Open access • Proceedings Article • DOI:10.1109/WCADM.1995.514642

Single-rail handshake circuits — Source link

Anna Peeters, K. van Berkel

Institutions: Eindhoven University of Technology, Philips

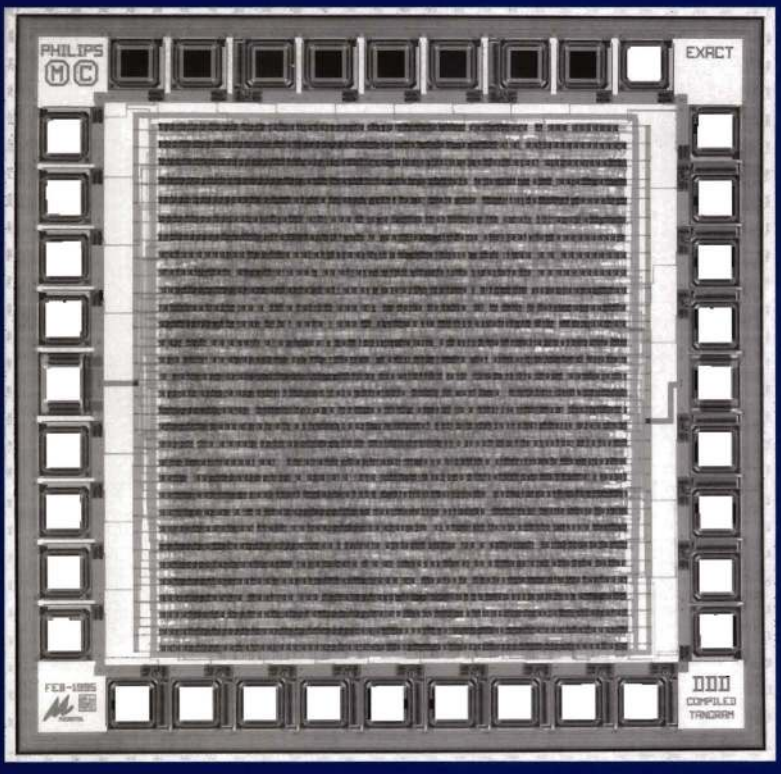
Published on: 30 May 1995

Topics: Asynchronous system, Handshake and Asynchronous communication

Single-rail handshake circuits

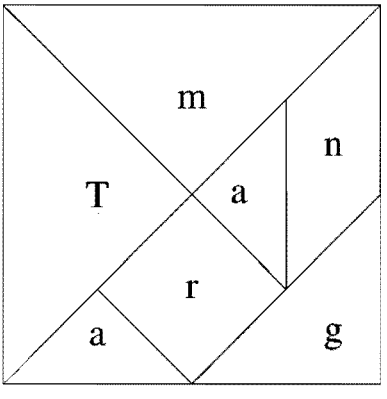
Single-Rail Handshake Circuits

Ad M.G. Peeters



Single-Rail Handshake Circuits

Ad M. G. Peeters



Cover: layout of the single-rail demonstrator IC, which is discussed in Chapter 7.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Peeters, Adrianus Marinus Gerardus.

Single-Rail Handshake Circuits / Ad M. G. Peeters. -

Proefschrift Technische Universiteit Eindhoven. -

Met lit. opg. - Met samenvatting in het Nederlands.

ISBN 90-74445-28-4

Trefw.: IC-design, VLSI, asynchronous circuits.

Single-Rail Handshake Circuits

PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR
AAN DE TECHNISCHE UNIVERSITEIT EINDHOVEN,
OP GEZAG VAN DE RECTOR MAGNIFICUS,
PROF. DR. J.H. VAN LINT,
VOOR EEN COMMISSIE AANGEWEZEN
DOOR HET COLLEGE VAN DEKANEN
IN HET OPENBAAR TE VERDEDIGEN
OP WOENSDAG 12 JUNI 1996 OM 16.00 UUR

DOOR

ADRIANUS MARINUS GERARDUS PEETERS

GEBOREN TE DONGEN

Dit proefschrift is goedgekeurd door de promotoren:

prof. dr. M. Rem

en

prof. S. B. Furber

en door de copromotor:

dr. ir. C. H. van Berkel.

The work described in this thesis has been carried out at Philips Research Laboratories Eindhoven while the author was employed on EXACT (ESPRIT project 6143) by Eindhoven University of Technology, under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Contents

Preface

In the Tangram project at Philips Research Laboratories Eindhoven, silicon compilation and asynchronous circuit techniques are combined to enable the fast design of low-power circuits. VLSI designers using the Tangram system are offered a powerful programming language, called Tangram, a compiler from Tangram to standard-cell netlists, and several tools that give fast and accurate feedback on performance aspects such as area, time, energy, and testability.

Handshake circuits form an intermediate representation in the compilation from Tangram to silicon. At this level one can still choose between various handshake protocols and data encodings. The first version of the Tangram system was based on the combination of a four-phase handshake protocol and double-rail data encoding.

These double-rail handshake circuits successfully demonstrated the low-power potential of compiled asynchronous circuits. The low-power advantage, however, came at too high costs, namely an unacceptable area overhead and a dedicated standard-cell library.

The challenge that was formulated for the research documented in this thesis was to remove these two roadblocks towards exploitation. To this end single-rail techniques were investigated and applied to handshake circuits.

The thesis introduces single-rail implementations of handshake circuits as a cost-effective way to realize low-power asynchronous circuits. The thesis presents an overview of several ways to combine single-rail data encoding with handshake protocols. A design-flow from Tangram to single-rail realizations in a generic standard-cell library is defined. This flow is applied successfully to a demonstrator IC: an error-detector that is part of a DCC player.

Much of the research that is documented in this dissertation was carried out in the context of EXACT[1]. The aim of this ESPRIT project was to demonstrate the low-power promise of asynchronous VLSI circuits. The single-rail DCC chip was used as one of the demonstrators in EXACT.

[1] EXploitation of Asynchronous Circuit Technologies (ESPRIT project 6143)

Acknowledgements

Throughout the years the Tangram Team at Philips Nat.Lab. has always been an inspirational environment to work in. I am especially indebted to Kees van Berkel for inviting me 'to spend a brief period' in the team, to learn more about asynchronous circuit design. This was is July 1991, and since then I have learned a lot and was allowed to work on some challenging problems. I would like to thank Eric van Utteren and the members of the team, Ronan Burgess, Joep Kessels, Marly Roncken, Frits Schalij, Hans van Gageldonk, and Rik van de Wiel, for their support and the pleasant cooperation.

Ever since my M.Sc. work in the Parallelism and Architecture group at Eindhoven University, Martin Rem has encouraged me to write a Ph.D. thesis on asynchronous silicon compilation. I am most grateful to him for this continuous support and for his help in focusing the thesis on single-rail handshake circuits.

I want to thank the members of the thesis committee —which, in addition to Martin Rem and Kees van Berkel, consisted of Steve Furber, Emile Aarts, and Jochen Jess— for the effort they put into judging this thesis.

During the period that I worked for EXACT, Andrew Bailey has been an ideal colleague. We had a lot of interesting discussions, not only on asynchronous circuit design, but also on politics, religion, and other facts of life.

The members of the Eindhoven VLSI club are acknowledged for the regular Friday-morning discussions. Rudolf Mak and Tom Verhoeff taught me to be critical. Jo Ebergen introduced me into the interesting field of asynchronous and delay-insensitive circuit design.

Manchester University was one of the partners in EXACT and during my work I had the opportunity to pay some visits to the AMULET group. The discussions with Steve Furber, Craig Farnsworth, Shiv Sikand, and Doug Edwards helped me in understanding more about micropipeline techniques.

ESPRIT Working Group 7225 (ACiD) is gratefully acknowledged for funding my visits to Manchester University and to workshops and conferences.

There is a lot more to life than working, and for me, running has become one of the major other occupations. It has always been an excellent way to relax after or during a working day and to find new inspiration. On the other hand, the sort of work I have been doing enabled me to train a lot and to turn running into a way of life. Although running is often considered to be an individual sport, I have always experienced it as a team sport, for which I would like to thank a lot of friends, especially the former Asterix core, the current PSV running group, the Nat.Lab. runners, and their respective trainers.

Chapter 1

Introduction

The market for portable consumer-electronic products is booming, and is expected to continue to grow at a high rate for many years to come. Portable telephones, digital assistants, notebooks, video games, buzzers, and portable digital audio, are products that already today are affordable to a lot of people. Several key technologies have made the introduction of all these portable devices possible. Display technology, audio and video compression, and IC technology are some of these enabling technologies.

Power consumption is a major issue for all these portable products. Low power consumption of a device may enable the use of less batteries, which results in a lighter and more appealing product, or alternatively, the product may be recharged less frequently.

An important source of power consumption is the digital signal processing inside these portable products. The ever decreasing dimensions of especially CMOS IC-technology have enabled the integration of more and more digital functions. Although CMOS was originally preferred because of its low-power consumption in comparison with other technologies, power consumption of digital CMOS ICs has now become a main concern. Digital ICs demand an ever increasing portion of the power budget of portable devices.

One of the alternatives that is investigated to reduce the power consumption of digital ICs is not to use a clock to drive the operation of the IC, but to apply asynchronous techniques instead. Although these techniques generally introduce some area overhead because they require some circuitry to replace the control by a clock, the extra cost may be affordable if it results in a significant reduction in energy consumption. Furthermore, the circuits should be implementable against reasonable design-costs, which in general means within a short design time.

This dissertation introduces *single-rail* implementation of *handshake circuits*

as a means to realize area-efficient low-power asynchronous VLSI circuits. Neither 'single-rail' nor 'handshake circuits' is a new concept, but the combination of the two is, especially in the context in which it is applied in this thesis, namely that of a silicon compiler and a standard-cell layout style.

Part of the work described here has been reported earlier in two papers at the 1995 London *Asynchronous Design Methodologies* working conference [65, 10]. For references to asynchronous literature the public 'asynchronous' bibliography as maintained at Eindhoven University of Technology has been used [64].

1.1 Tangram project

The project 'VLSI Programming and Silicon Compilation' at Philips Research Labs Eindhoven was initiated in 1986. The central idea was —and still is— to view VLSI design as a programming activity. This can be achieved by combining a powerful programming language and a good silicon compiler for that language. In the course of the project a VLSI programming language called 'Tangram' has been defined and a compiler from Tangram to VLSI circuits has been implemented. In the rest of the thesis the project is referred to as 'Tangram project.'

Handshake circuits form the intermediate representation in the fully automatic compilation of Tangram programs to VLSI circuits, and where introduced by Van Berkel in [8]. A handshake circuit is a network of handshake components, connected by point-to-point handshake channels. In such a circuit, all communication takes place via handshaking. VLSI programs written in Tangram are translated in a transparent way into handshake circuits, which are subsequently, on a component-by-component basis, replaced by gate netlists.

Handshake circuits can be realized in silicon in different ways. One of the degrees of freedom is the timing assumptions that are made in this mapping. The only timing assumption that was made originally in the implementation of handshake circuits was that of the *isochronic fork*, which is a fork (branch in a wire to different inputs) for which the difference in the delays between the two branches is shorter than the delays through the gates to which the fork is an input. The isochronic fork is introduced by Burns and Martin [20, 57] and is considered to be an essential and the 'weakest possible' compromise to true delay insensitivity [56] if such circuits are to be realized in CMOS, the dominating IC-technology of today. The resulting circuits are generally called *quasi delay insensitive* (QDI) [19]. Implementation aspects of isochronic forks are well understood [56, 7].

The restriction to QDI implementation of handshake circuits implies that for data communication a delay-insensitive encoding of the data should be used [82]. The most natural coding scheme then is the double-rail code, in which two wires per

bit are used, one to signal the communication of a '0,' the other for communicating a '1.' Double-rail operations on data, such as addition, require complex cells, not normally available in a standard-cell library. Therefore, a dedicated asynchronous cell library was developed.

With this approach a number of interesting circuits have been realized. In 1987 the feasibility of the approach was demonstrated with working silicon [6]. This led to the definition of the VLSI-programming language Tangram, the implementation of a compiler from Tangram to handshake circuits, the development of a cell library, the implementation of a compiler for handshake circuits, and the implementation of a tool set to support this design flow. A second demonstrator IC, used to validate the tools and the flow, was reported in 1993 [14]. In terms of complexity the most-notable Tangram-compiled ICs are the chips for the error decoder of the Digital Compact Cassette (DCC) player [11].

The DCC chip set proved that designs of industrially-relevant complexity could be handled by the tools and that interfaces to synchronous environments, standard protocols, and standard DRAMs could be straightforwardly implemented. More importantly, however, the circuits demonstrated an interesting power advantage over a commercial synchronous equivalent: the asynchronous circuit used only a fifth of the energy for the equivalent function.

1.2 Roadblocks

Although low power was recognized as a key advantage of Tangram circuits, two roadblocks prohibited practical application and exploitation of Tangram VLSI programming. The most important barrier was the 70 to 100% area overhead of the double-rail circuits over synchronous implementations of the same functions. This overhead is not industrially acceptable, even when the circuits are five times more energy efficient.

A second, increasingly important, obstacle was the use of the dedicated asynchronous standard-cell library, while the trend in industry has been towards generic standard-cell libraries. Such a library comprises standard logic gates (NAND, NOR, AND, OR, XOR), some complex gates (AND-OR-INVerts), inverters and buffers of a range of driving capabilities, and special functions such as adders, multiplexers, decoders, latches, and D-types. The advantage of generic libraries is that a design can easily (fast and cheap) be retargeted to different versions of a given technology (low power, high performance, minimal area), to different technologies (CMOS in various feature sizes, FPGA, gate array), and to different manufacturers.

Both the area overhead and the need for a dedicated cell library are due to the double-rail encoding of data that has been used. Since double-rail uses two wires

per bit in the encoding of data, the area of such a circuit is about twice that of a circuit in which only one wire per bit is used. (Each wire has to be driven by some cell, and hence implies transistors and circuit area.)

1.3 Challenges

The goal that was set for the work described in this thesis was to greatly reduce the area overhead of handshake circuits and, simultaneously, to map them onto a generic standard-cell library. Both points are essential when striving for acceptance, application, and production of asynchronous circuits on a larger scale.

The main candidate for area reduction is the datapath, where we should switch from an encoding based on two wires per bit to one that uses only one wire per bit. *Single-rail* data encoding [72], which uses one wire per bit, plus one additional wire to signal the validity of the data, is such an encoding. This form of data encoding is also known as bundled data [75] and was already applied in the sixties in the Macromodule project [74]. Especially for wide datapaths, single-rail encoding should lead to a reduction in the number of wires and transistors, and thus to a smaller area.

The choice for single-rail encoding introduces a new timing assumption. In QDI implementations we only had to consider isochronic forks, but in single-rail we have to take a more quantitative look at delays. The use of a data-valid wire introduces the *bundling constraint* [75]: the data-valid signal must arrive later than the valid data. This implies that the delay through the data-valid path should be larger than the worst-case delay that can be encountered in the datapath.

The bundling constraint implies new verification obligations in the design flow, especially in the context of the standard-cell layout style that we pursue. In this style, the actual delay of gates is only known when the layout is completed and before that we have to resort to estimated wire loads. We have to take the sensitivity of the delays to these estimates into account, most notably in comparing the delay of the datapath with that of the data-valid signal. If we want to minimize the verification effort, we have to combine optimistic estimates for the data-valid wire (low loads, short wires) with pessimistic estimates for the actual data (high loads, long wires). From this observation it is clear that one of the challenges in single-rail implementations is to combine manageable verification effort with acceptable performance.

An important advantage of single-rail datapaths is that datapath operators such as adders and exclusive ors can be found in any generic standard-cell library. This motivates our choice to try and implement *all* handshake components, both control and data, using only such a generic cell library.

The double-rail datapath was identified as the main source for area inefficien-

cies. The control path, however, must also taken into account when trying to reduce the area of handshake-circuit implementations. In going from a dedicated to a generic library the circuit area of the control is likely to increase. One possible remedy is to identify frequently occurring combinations of handshake components and to substitute these by more economic implementations. Another approach that might help is to exploit the richness of the generic library through peephole optimization.

Low power is considered to be the key strength of asynchronous circuits. We therefore also strive to improve the power efficiency of handshake circuit implementations. Area reduction, however, *is* the main concern, because the 70–100% area overhead of double-rail circuits is a serious handicap. We furthermore believe that, to some extent, less area implies shorter wires and thus less power and potentially more speed.

Another challenge in the step from double-rail to single-rail is *not* to affect the compilation from Tangram to handshake circuits, thereby keeping this compilation independent of the actual implementation style of the handshake circuit.

We furthermore want to maintain the push-button aspect of the compilation from handshake circuit to layout. This means that we should minimize the effort for post-layout verification. The safety margins in the data bundling, and the sensitivity to variation in processing and operation parameters are, therefore, important aspects of the single-rail design-flow.

Finally, the testability of single-rail circuits (against production faults) is of utmost importance when striving for practical applications.

1.4 Contributions

Throughout the thesis a systematic approach to single-rail handshake circuits is introduced. In going from double-rail to single-rail we basically increase the engineering content of handshake circuit implementations. This improves area, timing, and energy quality of the resulting VLSI circuits at the cost of a well-identified and manageable increase in design effort.

The main contributions of the research documented in this thesis are the following:

- An inventory of single-rail handshake protocols, leading to a surprisingly rich domain to choose from.

- The identification of a four-phase handshake protocol in which all four phases are productive (functional), thus removing an often mentioned disadvantage of four-phase handshaking, namely the redundancy of the return-to-zero phase.

- The definition and implementation of a push-button single-rail design flow, targeted at a generic standard-cell library.
- A single-rail demonstrator in the form of fully functional silicon.

1.5 Overview

In this chapter we have given some context and defined the main target of the research that is reported in the rest of the thesis. The structure of the thesis is as follows.

Chap 2. Introduction to the Tangram VLSI-programming approach. This chapter briefly introduces the VLSI-programming language Tangram, handshake circuits, the compilation from Tangram to these handshake circuits, and an overview of the Tangram toolbox.

Chap 3. Discussion of single-rail data encoding. Takes stock of various ways to implement handshake channels. Various single-rail data-valid schemes are introduced.

Chap 4. Implementation of single-rail handshake circuits. The choice of a 'good' data-valid scheme, based on the compilation scheme from Tangram to handshake circuits. The implementations of assignments, communication, iteration, and selection are discussed.

Chap 5. Implementation of handshake components, based on a data-valid scheme for the handshake channels. The delay assumptions in the implementation of the components are made explicit.

Chap 6. Single-rail design-flow, with emphasis on the role of peephole optimization at various levels of representation.

Chap 7. The viability of single-rail handshake circuits is demonstrated by the implementation of a DCC error detector.

Chap 8. Single-rail handshake circuits are compared with various other (synchronous and asynchronous) circuit technologies. This leads to an identification of the strengths, weaknesses, opportunities, and threats of single-rail handshake circuits. The chapter ends with an identification of the main remaining issues on the route to exploitation of asynchronous circuits.

Chapter 2

Tangram Handshake Circuits

Handshake circuits are the central architecture in the Tangram project. They form the intermediate representation in the compilation from Tangram to VLSI circuits. Essentially, handshake circuits abstract from all VLSI aspects, thus separating Tangram compilation aspects from VLSI technology details.

This chapter briefly sketches the role of Tangram as a VLSI-programming language. Subsequently, handshake circuits, as introduced by Van Berkel [8], are discussed. The compilation from Tangram programs to handshake circuits is addressed, which results in the identification of Tangram handshake circuits: handshake circuits as they can be obtained by compilation from a Tangram program. The efficient implementation of these Tangram handshake circuits is the subject of the rest of this thesis.

2.1 VLSI Programming

Silicon compilation, that is, the automatic generation of VLSI circuits from descriptions written in a high-level programming language, demands a powerful programming language and a good compiler. The programming language should abstract from VLSI circuit and technology details, thus allowing the designer to concentrate on application and programming issues. The silicon compiler should be able to translate these programs into *efficient* VLSI circuits automatically, which induces requirements for the programming language.

Two extreme approaches to silicon compilation can be identified. The first is to combine a traditional programming language, like C or Pascal, with a powerful compiler (or, perhaps more accurately, a synthesizer) that extracts parallelism, chooses efficient data-encodings, optimizes the amount of sharing, et cetera. The advantage of this style is that it puts minimal burden on the designer (programmer).

An important disadvantage is that it generally is unclear how to steer the compiler, for example towards a low-power solution. A small change in the input program that was thought to improve a design aspect, may actually decrease the quality of the resulting circuit. A familiar cause for this may be an unexpected side effect: the change may disable optimizations that have been applied to the original program.

The approach that has been chosen in the Tangram project represents the other extreme. It is based on a more dedicated programming language in combination with a highly *transparent* silicon compiler, which allows the designer to infer circuit costs and performance directly from the program. The language (Tangram) is similar to a traditional programming language but in addition

- offers language constructs to explicitly deal with parallelism, at any level of granularity;
- supports synchronized (CSP-like [43]) communication via channels;
- makes the sharing of hardware resources explicit, both for control structures and the datapath; and
- offers provisions for tuple construction and selection, type casting, and type fitting, which can be used to choose the most appropriate and efficient data encoding.

The transparency of the compiler enables the designer to evaluate the silicon area, timing, power consumption, and testability of the compiled circuit at the program level. This high-level feedback is an essential requirement to allow the designer to make trade-offs and to explore the design space.

This approach to silicon compilation is called *VLSI programming* [5], and the designer of such a program is referred to as a *VLSI programmer*.

The VLSI programmer typically is a system expert in the application field for which the design is intended. Detailed VLSI knowledge is not required, since this is hidden in the tools and the libraries. However, some silicon awareness (insight into the area and energy properties of a VLSI design) is essential to arrive at cost-effective solutions.

To support the VLSI programmer in the exploration, a set of tools have been defined and developed. These tools, and the VLSI programmer's view on the design flow are illustrated in Fig. 2.1.

Tangram Compiler. This compiler implements the syntax-directed translation of Tangram programs into handshake circuits, as described in [6, 16, 8].

For each production rule of the Tangram syntax a corresponding translation rule to handshake circuits exists in the compiler. This translation is therefore

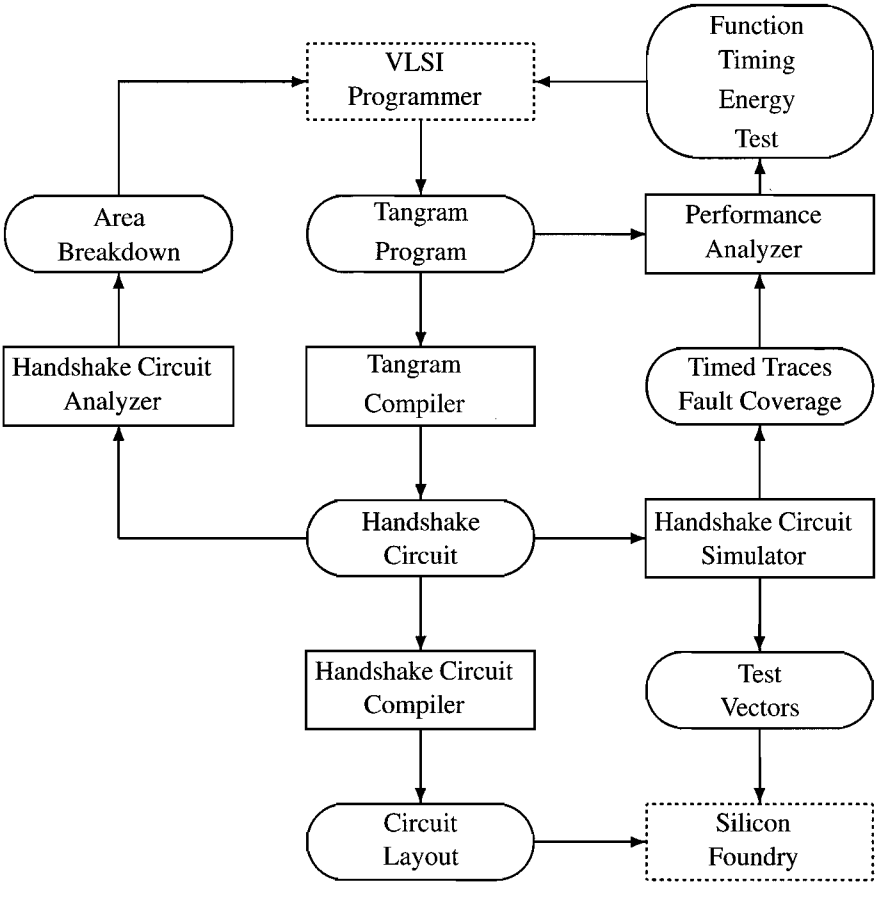


Figure 2.1: The Tangram Toolbox: boxes denote tools, ovals denote (design) representations.

highly transparent, and allows one to relate performance criteria like timing, area, power, and testability directly to the Tangram program.

Handshake Circuit Simulator. This discrete event simulator is based on detailed functional, timing, energy, and test models of handshake components. The test-coverage part of this simulator is described in detail by Van de Wiel [79]. The other design aspects have also been modeled in VHDL, which makes co-simulation of Tangram generated circuits and other designs relatively straightforward.

Performance Analyzer. This is a tool with a high-quality graphical user interface that is used to link simulation results to the Tangram program. After simulation, the VLSI-programmer can use this interactive tool to evaluate the performance and the energy consumption of the Tangram program, and to identify possible bottlenecks and 'hot spots.'

Handshake Circuit Analyzer. This tool generates area statistics from a handshake circuit. In contrast to the other performance criteria, area information is not translated back to the Tangram level.

Handshake Compiler. Compilation from handshake circuits to layout consists of several steps, which are detailed in Chapter 6. This compilation forms the central theme of this thesis and can be partitioned into two phases: component substitution and layout generation. Component substitution is the mapping of handshake components onto a cell library and is part of the Tangram tool set. The netlist that results from this phase can be input in any commercially available CAD framework for placement, routing, simulation, and verification.

In addition to the tools shown in the diagram and mentioned above there is also a faster simulator that can be used for functional and coarse-grain timing simulation. This simulator is based on the direct compilation of Tangram to C, without using handshake circuits as intermediate. The timing information is linked to the Tangram program using the same performance analyzer as for the handshake circuit simulator. A VLSI programmer typically uses this tool during the initial phase of the design, when fast iteration is more important than high accuracy.

In the final stage of the design other simulators, based on representations that are more detailed than the handshake circuit (netlist, layout) are also used. These simulations are typically part of a standard CAD framework, and are not dedicated to asynchronous or handshake circuits.

2.2 Handshake circuits

Handshake circuits form the intermediate representation in the compilation from Tangram programs to VLSI. It is the central architecture throughout this thesis. In this section we introduce handshake channels, handshake components (which communicate via such channels), and handshake circuits (networks of handshake components).

2.2.1 Handshake channels

Handshake signaling is a communication mechanism that establishes point-to-point synchronization. A handshake involves two partners which play different roles, called *active* and *passive*. The partners exchange so-called *request* and *acknowledge* signals. The passive partner waits for a request to arrive and after receipt of a request responds with sending an acknowledge. The active partner starts with issuing a request and then waits for the corresponding acknowledge to arrive. Such a combination of a request and an acknowledge is called a handshake.

Throughout this thesis we assume the active and passive roles to be fixed, which means that one partner will always be active, and the other will always be passive. The communication medium between the partners is called a *handshake channel.* We conform to the convention to denote an active handshake partner with a fat dot (●), and a passive partner with an open circle (○), see Fig. 2.2.

Figure 2.2: A handshake channel represents the communication medium between an active and a passive handshake partner.

Those who are used to think in terms of Petri-nets [67], places, and tokens, may think of handshake communication as the exchange of tokens. Initially the active partner has the token. Sending a request is then interpreted as passing the token from the active to the passive partner. An acknowledge is represented by sending the token from the passive to the active partner. The fat dots and open circles can then be thought of as indicating the initial distribution of the tokens.

A handshake essentially synchronizes the active and the passive partner. In addition to pure synchronization, handshakes can also establish data communication between the partners by encoding data in the request, in the acknowledge, or in both.

Handshake channels with no data encoded are called *nonput* channels. They connect two so-called nonput handshake partners, one active, one passive. A hand-

shake on a nonput channel establishes a synchronization only; no data is communicated.

The second type of handshake channels are those with data encoded in the request. These channels connect an active sender and a passive receiver. So, the sender takes the initiative for a communication action. One might say that the sender *pushes* the data through the channel, therefore these channels are referred to as *push*[1] channels. From a data-flow point of view push channels are *data driven.*

On a *pull*[1] handshake channel data is encoded in the acknowledge. Such a channel connects a passive sender and an active receiver. The sender issues data after receiving a request from the receiver, so one could say that the receiver *pulls* the data through the channel. From a data-flow point of view pull channels are *demand driven.*

The fourth type of handshake channels are *biput* channels, on which data is encoded in both the request and the acknowledge. One handshake then establishes the exchange of values between the two partners. The active partner now initiates the handshake by sending data to the passive partner. The passive partner then responds by sending data back. The handshake partners alternatingly act as sender and receiver.

The four types of handshake channels are depicted in Fig. 2.3. On data channels, arrows indicate the direction(s) of data-flow. In the context of Tangram only nonput, push, and pull channels are applied. Biput channels do not recur in the rest of the thesis. For push and pull handshake channels, the data that can be communicated on such a channel is called the *type* of the channel.

For a handshake channel a we use a_r to denote the request of a, and a_a for its acknowledge. If a is a push channel, the communication of a value x (encoded in the request), is denoted by $a_r(x)$. Similarly, the communication of a value x on pull channel a is denoted by $a_a(x)$.

Now that we have introduced symbols to denote events on handshake channels, we can use *traces* to record sequences of events that can be observed on handshake channels. The allowed sequences of events can then be specified using *commands.* Traces and commands are introduced by example, rather than via a separate formal section. For Trace Theory as used throughout this thesis the reader is referred to Van de Snepscheut [78], Hoare [44], Kaldewaij [47], Ebergen [29], or Verhoeff [83].

The handshake protocol prescribes that a handshake is initiated with a request and ended by an acknowledge. The allowed sequence of events on a nonput handshake channel a can thus be described by command $*(a_r\,;a_a)$. In this command

[1] The names push and pull were coined by Joep Kessels. Previously, these type of channels were referred to as straight and anti, respectively. Push and pull are also used as strategies in flow-shop control. A drawback of the push-oriented strategy then is the accumulation of unsold products. Other familiar terms, especially in the market place, are Technology Push and Market Pull.

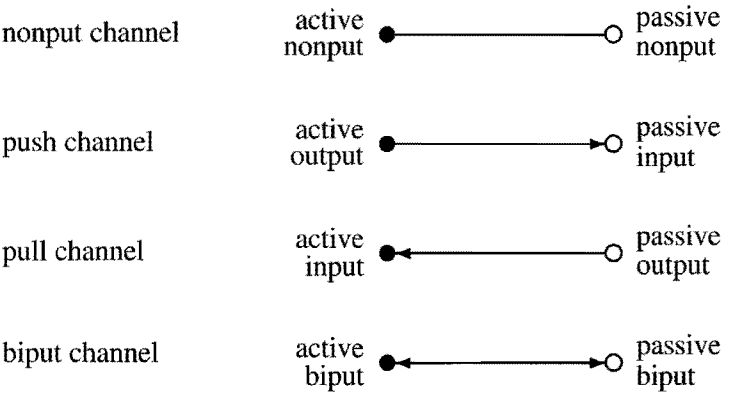


Figure 2.3: Symbols for the four types of handshake channels

the semicolon indicates that an acknowledge is allowed only after the request has occurred, and the star denotes repetition.

Throughout the thesis we use explicit typing of push and pull channels, especially in specifications, for which we use commands. All commands in which data plays a role are of the form $|[\, D \mid C \,]|$, where D (for declaration) introduces variables of some appropriate type, and C is a command in which these variables occur. For a push channel a of type T the behavior is prescribed by $|[\, x : T \mid *\,(a_r(x)\,;a_a)\,]|$. This command describes the alternating of requests in which data of type T is encoded and acknowledges. Similarly, pull channel b of the same type is specified by command $|[\, x : T \mid *\,(b_r\,;b_a(x))\,]|$.

2.2.2 Handshake components

Handshake components are components that use handshake channels to communicate with their environment. The interface of a handshake component to a handshake channel is called a *handshake port*. These ports are either active or passive, depending on whether the component plays the active or the passive role on that channel.

If data is encoded on a handshake channel, then the handshake ports are input or output ports, depending on whether the channel is push or pull, and whether the port is active or passive. For a handshake port connecting to handshake channel a, we use the following conventions to denote the type of the port.

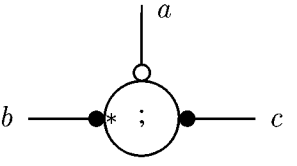
notation	port activity	channel type
a°	passive	nonput
$a^\bullet$	active	nonput
$a^\circ?T$	passive	push, T
$a^\bullet!T$	active	push, T
$a^\circ!T$	passive	pull, T
$a^\bullet?T$	active	pull, T

We use commands to specify the behavior (that is, the allowed sequences of events at the external interface) of handshake components. One restriction that all components satisfy is that the handshake protocol on all handshake channels involved is obeyed. This can always be readily verified from the command specifications that are given.

In the commands, input and output events can be distinguished. For passive ports, the request events are inputs and the acknowledge events are outputs. On active ports this is the other way around, which means that requests are outputs and acknowledges are input.

The *sequencer* is used to implement sequential composition of Tangram. It is a handshake component with one passive nonput port and two active nonput ports. The specification and the symbol for a sequencer with ports a, b, and c are shown below.

$$\text{SEQ}(a^\circ, b^\bullet, c^\bullet) = \\ *(a_r\,;b_r\,;b_a\,;c_r\,;c_a\,;a_a)$$



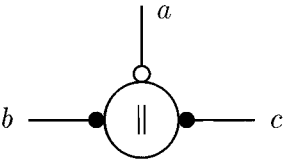
When this sequencer is activated along channel a it first performs a handshake along b, then a handshake along c, and then signals successful termination by completing the handshake along a. In the symbol for the sequencers we label the handshake channel that is first activated with a '$*$.'

From the command one can derive the behavior on the individual channels. If we focus on channel b, for instance, the behavior is characterized by $*(b_r\,;b_a)$. The sequencer thus satisfies the handshake protocol on channel b. This can be checked for all handshake channels that the sequencer connects to. Other handshake components can be verified in a similar way.

The *parallel* component implements Tangram's parallel composition. Its handshake interface is the same as that of the sequencer, that is, it has three nonput ports, one passive and two active. When activated along its passive port, it initiates handshakes on its active ports, then waits till both handshakes have completed, and signals this by sending an acknowledge on its passive channel. Its specification us-

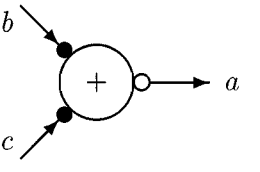
ing a command and the symbol for the component are shown below. Notice that in commands, sequential composition (denoted by ';') has higher priority than parallel composition (denoted by ' ||').

$$\text{PAR}(a^\circ, b^\bullet, c^\bullet) = \\ *(a_r\,;(b_r\,;b_a \parallel c_r\,;c_a)\,;a_a)$$



The sequencer and the parallel are both examples of control components. Of course, there are also handshake components that deal with data. An example of a data component is the *adder*, whose specification and symbol are given next. This component operates in a demand-driven fashion. It awaits a request on its result channel before it collects the operands, computes the result, and outputs this on the result channel.

$$\text{ADD}(a^\circ!T_a, b^\bullet?T_b, c^\bullet?T_c) = \\ |[\,x : T_b, y : T_c \\ |\ *\,(a_r\,;(b_r\,;b_a(x) \parallel c_r\,;c_a(y))\,;a_a(x+y)) \\ ]|$$

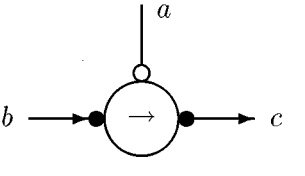


The adder is an example of a parameterized component. The inputs and output each have their own type, in which the type of the output is determined by that of the inputs. This is discussed in more detail in Chapter 5.

One may observe that the specification of the adder is very similar to that of the parallel component. The communication behavior is essentially the same; the only difference is that the adder encodes data in the acknowledges.

The *transferrer* is a handshake component with three different kinds of handshake ports, a passive nonput port, an active pull port and an active push port. When activated along its passive channel it actively fetches data from its pull port and subsequently forwards this data along its push port. The transferrer thus transfers data upon request and —in handshake circuits— is an interface between control and data components, or, phrased differently, between the control and the datapath. The symbol and command specification are given below.

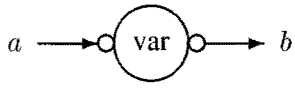
$$\text{TRF}(a^\circ, b^\bullet?T, c^\bullet!T) = \\ |[\,x : T \mid *\,(a_r\,;b_r\,;b_a(x)\,;c_r(x)\,;c_a\,;a_a)\,]|$$



The behavior of the transferrer, as specified in the command, is similar to that of the sequencer. The sequences of events are basically the same; in addition the transferrer has data encoded in the acknowledge of its input channel and in the request of its output channel.

The components addressed so far illustrate how operations on data can be performed, how data can be transferred, and how this can be controlled. An essential component that is missing is one to *store* data. The *handshake variable* implements exactly this. Its specification and symbol are given next.

$$\begin{aligned}&\text{VAR}(a^{\circ}?T, b^{\circ}!T) = \\ &\quad [\![\, x : T \mid *\,(a_r(x)\,;a_a \mid b_r\,;b_a(x))\,]\!]\end{aligned}$$



A variable can be engaged in a *write* handshake (on channel a) or a *read* handshake, and the environment must guarantee that these two will always be mutually exclusive. During a write handshake the content of the variable is updated; during a read handshake it is inspected.

All components, except the variable, are *receptive* [8]. This means that on all channels, if the handshake protocol on that channel allows for an input to occur, the component is willing to accept that input. (For a passive port, the request is the input, for an active port the acknowledge is the input.) More precisely, if a trace that is specified by the command can be extended with an input as far as the handshake protocol is concerned, then this extension is also specified by the command. The parallel component, for example, initially and after completion of a full cycle, can accept a request on its passive channel. On both active channels, an acknowledge is anticipated directly after the corresponding request is sent.

The specification of the variable as given in the command above implies that the variable is *not* receptive. During a read handshake, for instance, the component is not receptive for a write request. Tangram's compilation scheme guarantees mutual exclusion between read and write accesses [16, 8]. Therefore the non-receptive variable always suffices in the Tangram context.

2.2.3 Handshake circuits

Handshake channels can be used to connect handshake components into networks of handshake components that are called *handshake circuits*. The connections that are made by a handshake channel are point-to-point, and of course the type of the channel should match that of the handshake ports of the components. Furthermore, a channel must connect an active port to a passive port. A push handshake channel, for instance, can only connect an active output port to a passive input port.

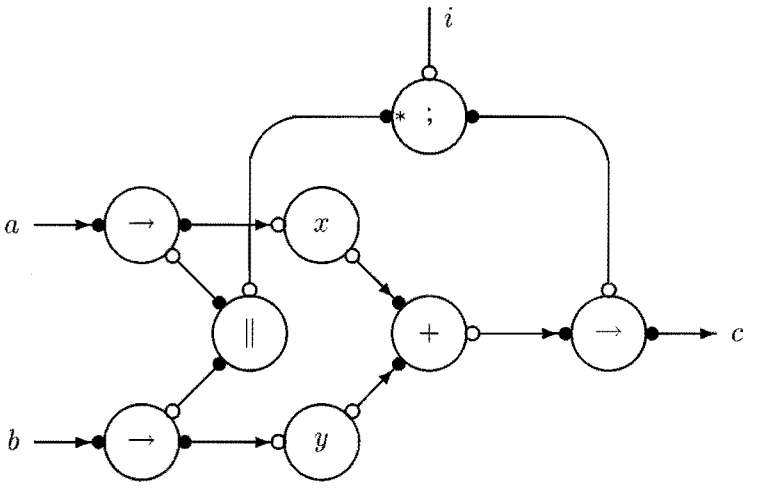


Figure 2.4: An example handshake circuit, built from the components introduced earlier in this chapter.

With the handshake components that have been introduced in the previous section, we can for instance build the handshake circuit of Fig. 2.4. When activated via channel i, this circuit first activates the left branch of the sequencer, which in turn activates two transferrers in parallel. These transferrers operate independently. The top transferrer collects a value from channel a and stores this in variable x, the other transferrer stores the value collected from b in y. After both transferrers have completed, the acknowledges are combined in the parallel and sent to the sequencer. This component then sends a request via its right-hand channel, which activates the transferrer that connects to output channel c. The transferrer collects its output from the adder, which in turn collects its input (the operands) from x and y.

The above description of an operational cycle is rather verbose. We can also specify the behavior of such a handshake circuit as if it where a handshake component. This requires abstraction from internal detail and structure. The formal side of this (parallel composition and hiding) falls outside the scope of this thesis. For this the reader is referred to Van Berkel [8]. A command that specifies the handshake circuit of Fig. 2.4 is the following.

$$
\begin{array}{l}
|[\, x : T_a, y : T_b \\
|\ *\,(i_r\,;(a_r\,;a_a(x)\ \|\ b_r\,;b_a(y))\,;c_r(x+y)\,;c_a\,;i_a) \\
]|
\end{array}
$$

Designing handshake circuits by abutting handshake components is not a very

productive way of designing these circuits. It makes more sense to design these circuits in an algorithmic language and to use a compiler to generate the handshake circuits.

The programming language should abstract from the handshaking details, such as the exact interleaving of the request and acknowledge events and the assignment of active and passive roles to handshake ports. Shorthands should, for instance, be offered for 'collect a value from channel a and store this in variable x.' This naturally leads to a CSP-like language, in which the operation of the handshake circuit can be described as `(a?x || b?y) ; c!(x+y).`

2.3 Tangram

Tangram is a VLSI programming language that abstracts from handshake circuits and enables the design of a circuit as a programming activity. Programs written in Tangram can automatically be compiled to handshake circuits.

A key strength of Tangram compilation is the transparency of the compilation scheme. The basic idea is that for each Tangram construct there is a handshake component that implements the same function. This transparency is illustrated in the first two examples in this section. Two additional examples illustrate the expressive power of Tangram.

A complete definition of Tangram, together with a description of the performance characteristics (in terms of area, time, and energy) of the corresponding silicon as they can be derived from a Tangram program, can be found in the Tangram manual [71].

2.3.1 Adder revisited

In the previous section we only gave the command that corresponds to the handshake circuit shown in Fig. 2.4. To obtain a complete (compilable) Tangram program we have to add typing information and declarations. A Tangram program that actually compiles to the handshake circuit of Fig. 2.4 is shown below, in which we have chosen to input eight-bit operands and to output a nine-bit result.

```
  int8 = type [0..255]
  int9 = type [0..511]
|
(a?int8 & b?int8 & c!int9).
begin
  x,y : var int8
|
  (a?x || b?y) ; c!(x+y)
end
```

2.3.2 Wagging buffer

The wagging FIFO is included as an example to illustrate the sharing of control and datapaths in handshake circuits compiled from Tangram. The Tangram program for a wagging FIFO is shown below.

```
  byte = type [0..255]
|
(a?byte & b!byte).
begin
  x,y : var byte
| /* two-place wagging FIFO */
  a?x ;
  forever do (b!x || a?y)
           ; (b!y || a?x)
          od
end
```

In this program we encounter three inputs from channel `a`, two of the form `a?x` and one `a?y`. Incoming messages on channel `a` are alternatingly sent to variables `x` and `y`. The program also contains two outputs to channel `b`, one from variable `x` and one from `y`. Compilation of this Tangram program to a handshake circuit yields the circuit of Fig. 2.5.

This handshake circuit contains three so-called *mixer* handshake components. A mixer component has three handshake ports, two passive (say a and b) and one active (say c). The command that specifies the behavior of the mixer is

$$*(\, a_r \,;\, c_r \,;\, c_a \,;\, a_a \;\mid\; b_r \,;\, c_r \,;\, c_a \,;\, b_a \,).$$

This command specifies the *nonput mixer*. In addition, data may be encoded in the request or the acknowledge, in which case the component is a *multiplexer* or a *demultiplexer*, respectively.

Mixer components are used for sharing. The component labeled 'dmx' in the handshake circuit in Fig. 2.5 is a demultiplexer. It splits the input stream on channel a to variables x and y. On the output side a multiplexer (labeled 'mux') merges data from x and y onto channel b.

The datapath of the handshake circuit is connected to the control path via four transferrers, which each control one of the data transfers of the Tangram program. Since there are two occurrences of `a?x`, the corresponding transferrer has to be shared via a control mixer (labeled '|').

Channel i represents the initiation channel, which is used to activate the handshake circuit. The component labeled '$\star$' is a *repeater* and implements the `forever do od` construct of the program. The handshake circuit clearly represents the syntactic structure of the Tangram program. Each handshake component, apart from

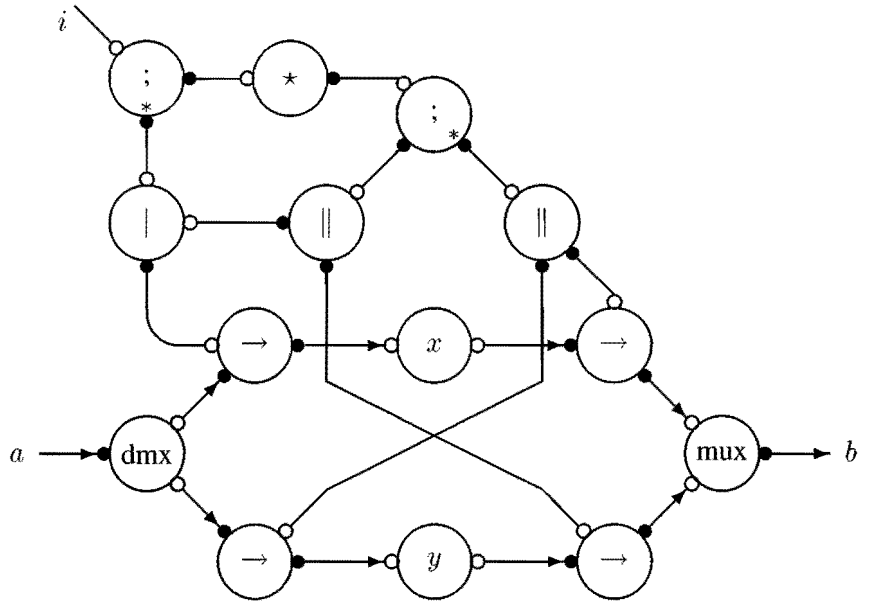


Figure 2.5: Handshake circuit for wagging FIFO

the mixers, can directly be related to the program. The mixers are required to implement sharing of hardware, which in the Tangram program shows as multiple occurrences of a construct.

2.3.3 Galois Field arithmetic

Tangram offers only two data types: booleans and integers from a specified range. Any other data type that is required can be programmed based on these two basic types, using tuple construction and selection, and type casting and fitting.

The construction of a user-defined type is exemplified in the Tangram program given in Fig. 2.6, which introduces the Galois Field data type for GF(2^8). This data type is used, for instance, in the Compact Disc program as discussed in [49] and the DCC decoder program from [11], which is used as basis for the demonstrator discussed in Chapter 7.

A Galois Field is characterized by a root of an irreducible polynomial. For the field described here this is polynomial $1 + x^2 + x^3 + x^4 + x^8$, and the root is denoted by α. Galois Field symbols are represented in the Tangram program as tuples of eight booleans. This type is called `gfsym`. Tupling in Tangram is denoted by << >>, in which items are separated by commas. The program introduces two con-

```
/* Definition of GF data type plus operations */
& gfsym    = type <<bool,bool,bool,bool,bool,bool,bool,bool>>
/* some useful GF constants */
& gff      = const false
& gft      = const true
& gfzero   = const <<gff,gff,gff,gff,gff,gff,gff,gff>>
& gfalpha  = const <<gft,gff,gft,gft,gft,gff,gff,gff>>
/* GF functions */
& gfiszero = func (s: gfsym): bool.
  -(((s.0+s.1)+(s.2+s.3))+((s.4+s.5)+(s.6+s.7)))
& gflrot   = func (s: gfsym): gfsym.
  <<s.7,s.0,s.1,s.2,s.3,s.4,s.5,s.6>>
& gfadd    = func (s,t: gfsym): gfsym.
  <<s.0#t.0,s.1#t.1,s.2#t.2,s.3#t.3,
    s.4#t.4,s.5#t.5,s.6#t.6,s.7#t.7>>
& mulalpha = func (s: gfsym): gfsym.
  <<s.7,s.0,s.1#s.7,s.2#s.7,s.3#s.7,s.4,s.5,s.6>>
& divalpha = func (s: gfsym): gfsym.
  <<s.1,s.0#s.2,s.0#s.3,s.0#s.4,s.5,s.6,s.7,s.0>>
```

Figure 2.6: Definition of Galois Field data type plus operations in Tangram.

stants of type `gfsym`: `gfzero`, which is the all-zero word, and `gfalpha`, which is the representation of α in `gfsym`.

Function `gfiszero` implements the check whether a `gfsym` equals `gfzero`. This function illustrates the use of tuple selection (denoted by a dot and the appropriate index, starting from 0) and the use of boolean operators for logical or (+) and negation (-). Rotation to the left over one position is defined in `gflrot`. The other three functions implement addition (bit-wise exclusive or, denoted by #), multiplication by α, and division by α. In addition to the functions shown here the CD and DCC programs require functions for inversion of a `gfsym` and multiplication of two `gfsyms`.

The Galois Field example illustrates that the definition of data types in a Tangram program does not differ from the definition of an abstract data type in any other programming language.

2.3.4 Carry-select adder

Tangram offers four basic operations for integers: addition (+), subtraction (-), negation (-), and comparison (=, <>, <, >, <=, >=). For these basic operations the VLSI programmer should be aware of their performance characteristics. The transparency of the compilation from Tangram to VLSI circuits can then be used to reason about the performance of (the silicon corresponding to) a Tangram program. The

```
/* type USi : UnSigned, i bits */
/* type  Bi : Boolean, i bits  */

  B9      = type <<bool,bool,bool,bool,bool,bool,bool,bool,bool>>
& BB9     = type <<bool,B9>>
& US8     = type [0..255]    /* < 2^8  */
& US9     = type [0..511]    /* < 2^9  */
& US16    = type [0..65535]  /* < 2^16 */
& US8US8  = type <<US8,US8>>
& US8B    = type <<US8,bool>>
& US16B   = type <<US16,bool>>
|
().
begin
  x,y,z : var US16
& cout  : var bool
| /* carry-select implementation of z := x+y */
  <<z,cout>> :=
  begin
    sumlow   = val (x cast US8US8.0 + y cast US8US8.0) cast US8B
  & sumhigh0 = val (x cast US8US8.1 + y cast US8US8.1) cast B9
  & sumhigh1 = val ( <<1,x cast US8US8.1>> cast US9
                   + <<1,y cast US8US8.1>> cast US9
                   ) cast BB9.1
  & sgn      = val sumlow.1
  & sgnb     = val -sumlow.1
  & sumhigh  = val << sumhigh1.0 * sgn + sumhigh0.0 * sgnb
                   , sumhigh1.1 * sgn + sumhigh0.1 * sgnb
                   , sumhigh1.2 * sgn + sumhigh0.2 * sgnb
                   , sumhigh1.3 * sgn + sumhigh0.3 * sgnb
                   , sumhigh1.4 * sgn + sumhigh0.4 * sgnb
                   , sumhigh1.5 * sgn + sumhigh0.5 * sgnb
                   , sumhigh1.6 * sgn + sumhigh0.6 * sgnb
                   , sumhigh1.7 * sgn + sumhigh0.7 * sgnb
                   , sumhigh1.8 * sgn + sumhigh0.8 * sgnb
                  >>
  |
    <<sumlow.0,sumhigh >> cast US16B
  end
end
```

Figure 2.7: Tangram program using carry-select addition.

adder that is offered, for instance, is a simple ripple-carry adder (this is the simplest implementation of addition of two numbers). Should the programmer require a different (faster) adder, then this can be programmed in Tangram rather straightforwardly. The implementation of `z:=x+y` with a carry-select adder, for instance, can be programmed in Tangram as shown in Fig. 2.7.

The carry-select Tangram program illustrates the use of a powerful language construct, namely type casting. This allows one to cast an expression to another type of the same size, in which the size is the number of bits used in the (unique) bit-vector representation. This requires that the VLSI programmer knows that integers in Tangram are represented as bit vectors, with the low-order bit representing the least-significant bit. This ordering of bits within a symbol is similar to the Big Endian ordering of bytes within a word [24].

With this information in mind the program can easily be understood. The program is based on splitting the 16-bit addition in two parts of 8 bits each. The low-order bits are added using the standard ripple-carry adder and the result is available as `sumlow`. For the high-order bits two values are computed, one based on a 'zero' carry-in (`sumhigh0`), the other based on a 'one' carry-in (`sumhigh0`). Based on the carry-out of the low-order addition (`sumlow.1`) the appropriate high-order addition is selected.

Another integer operation that is not directly supported in Tangram is multiplication. Tangram does allow, however, for the definition of a wide range of multipliers with various performance characteristics. An extensive treatment of Tangram programs for multiplication is given by Haans in [38].

Other examples of VLSI-programming in Tangram, with emphasis on programming for low power, can be found in reports by Van Berkel and Rem [17] and Kessels [48]. The examples given in this section illustrate the expressive power of Tangram. Common programs, such as those for integer multiplication and Galois Field arithmetic, could be put in a library of Tangram definitions, from which a VLSI programmer can then directly include a solution with the required performance characteristics.

2.4 Tangram handshake circuits

This thesis addresses the implementation of so-called *Tangram handshake circuits*. These are handshake circuits as they can be generated from a Tangram program. The restriction to Tangram handshake circuits allows us to exploit properties of the compilation scheme from Tangram to handshake circuits in the implementation. We try to keep the implementation of handshake circuits as general as possible, but at some points the restriction to Tangram allows for efficient implementations that

would not be suited in a more general context. The properties of the compilation scheme that are exploited are the following.

First of all, the Tangram compiler assures mutual exclusion between read and write accesses to handshake variables. For the Tangram programs this implies that so-called auto-assignments (like `x:=x+1`) are decomposed into two assignments (like `xx:=x+1; x:=xx`). This master-slave decomposition is implemented in the Tangram compiler. Removing this restriction from the Tangram compiler would require the introduction of master-slave variables, which in addition to normal read and write cycles allow for *read-modify-write* cycles.

Secondly, Tangram does not allow input communications in expressions. This implies that if an input has to be processed, it first has to be stored in a variable. This restriction could easily be removed from the language, since at the handshake circuit level inputs could easily be allowed in expressions. In the implementation of Tangram handshake circuits, however, we can take advantage of the restriction.

A third property of handshake circuits compiled from Tangram is that mixers of any type (nonput, multiplexer, demultiplexer) can be implemented non-receptively. The compiler assures mutual exclusion of mixer requests on different passive ports. The implementation of receptive mixers would require arbiter circuits to resolve possible conflicts.

Chapter 3

Single-Rail Data Encoding

For a cost-effective implementation of handshake circuits the efficient implementation of data storage, manipulation, and communication is of utmost importance. Single-rail implementations are promising since they only require one wire per bit for the data encoding, in contrast with double rail, for which two wires per bit are needed.

In this chapter single-rail data communication is introduced. The combination of single-rail with handshaking is then illustrated on two- and four-phase handshake protocols. In subsequent chapters, four-phase single-rail implementations of handshake components and handshake circuits are introduced. It turns out that the combination of single-rail and four-phase handshake protocols allows for a wide range of choices.

3.1 Single rail

In a single-rail data interface, schematically depicted in Fig. 3.1, one wire per bit is used to represent data. In addition to this there must be a means for the sender to inform the receiver that the data is stable and valid (called the *data-valid* signal), and a means for the receiver to indicate that this data is no longer required (the *data-release* signal).

The exchange of information between sender and receiver is organized as follows. The sender first puts valid data on the wires and then informs the receiver about this by sending a data valid signal. The receiver can then process this data and inform the sender when the data is no longer required by sending a data release signal.

The relation between sender and receiver in the single-rail protocol is essentially symmetrical. The sender may take an indefinite amount of time to prepare

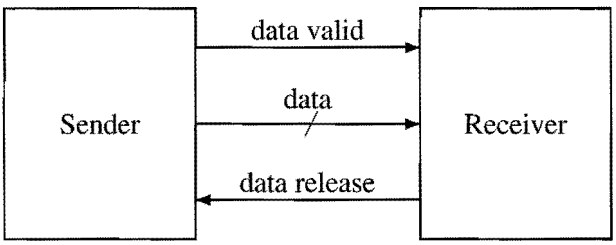


Figure 3.1: Single-rail data interface between sender and receiver.

new data. After the issue of the data-valid signal, however, it has to keep the data stable. The receiver can then prolong the data-valid period as long as required to fully assimilate the new data. Only after the issue of the data release signal by the receiver, the sender is relieved from its task to keep the data stable, and new data may be prepared.

The single-rail data communication scheme was already applied in the Macromodules project [23]. The scheme was then called *data validation* and the control signals were called *initiation* (valid) and *completion* (release) [74].

The term *single rail* was first used by Seitz to contrast it with double-rail encoding [72]. Sutherland, in his Turing Award lecture [75], popularized the use of the single-rail data encoding scheme under the name *bundled data*. This name emphasizes the importance of the so-called *bundling constraint*, that is, the obligation for the data-valid signal to arrive later than the data itself.

Both Seitz and Sutherland refer to the data valid wire as the request, and to the data release wire as the acknowledge. Per data transfer this is indeed a natural way to denote these signals, since a data communication then is initiated by a request and ended by an acknowledge. In the next section the relation between the handshake protocol and the single-rail scheme is discussed. Request and acknowledge are then signals that are distinguished in the handshake protocol and the interpretation as data valid and data release depends on the direction of the data flow.

3.2 Handshake channels

Tangram handshake circuits feature two kinds of handshake channels with data, namely, push and pull channels, cf. Chapter 2. These two types of handshake channels are shown in Fig. 3.2, together with the relation between the request and acknowledge of the handshake channel and the data valid and data release of the single-rail scheme. On push channels (Fig. 3.2, top) the data-valid signal is encoded in the request, and the data-release signal in the acknowledge. On pull channels (Fig. 3.2,

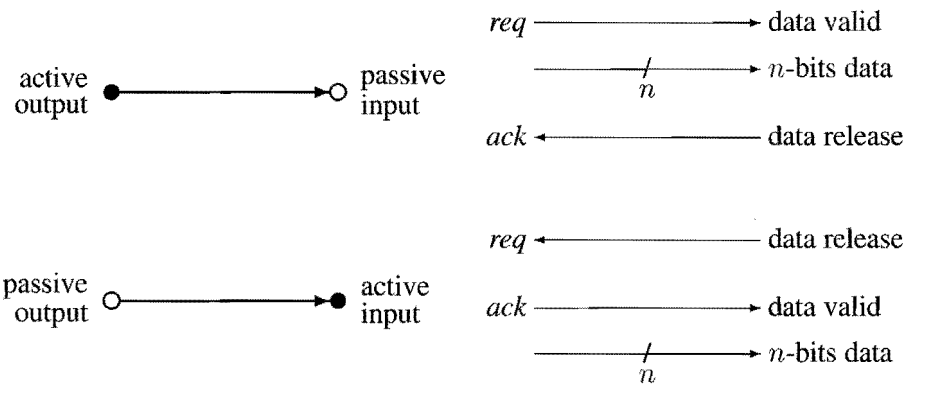


Figure 3.2: Single-rail push (top) and pull (bottom) handshake channels.

bottom) this is the other way around.

A communication on a push channel is initiated by the sender and starts with the issue of a data-valid signal. After assimilation of the new data the receiver responds by sending an acknowledge, which is interpreted as the data-release signal. On the handshake channel the data is thus valid *during* the handshake.

Communication on a pull handshake channel is initiated by the receiving party, which thereby requests new data. The first-ever request may be interpreted as a redundant data-release signal. The handshake is completed when the receiver acknowledges the request by issuing a data-valid signal. This implies that data is valid *between* handshakes and that the sender thus prepares new data during the handshake.

In literature on handshaking, communication data via pull channels is generally not addressed. Apparently, communicating data with handshakes in a demand-driven way is not very common. Martin, however, mentions that 'contrary to common belief, it is simpler to implement input commands with active ports than with passive ports' [57, Sec. 20.4].

3.3 Two phase

In the two-phase handshake protocol a request and an acknowledge wire are used to implement the handshaking between the active and the passive partner. If we assume both wires to be low initially, then the sequence of events that can be observed on such a channel is depicted in Fig. 3.3. Up and down handshakes can be distinguished, where after an up handshake both wires are high and after each down handshake both wires are low. Of course other initializations are also possible, and

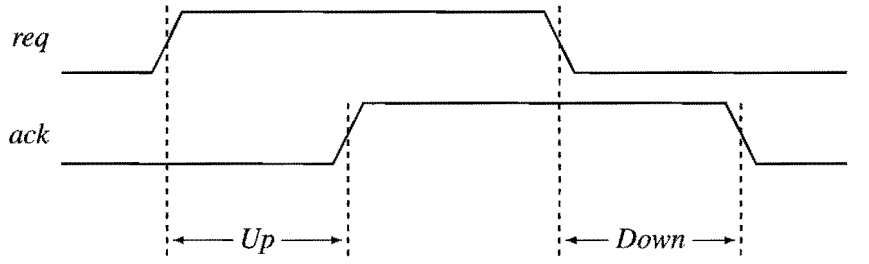


Figure 3.3: Two-phase handshaking

in particular it is not essential for the wires to start in the same state.

In the two-phase handshake protocol all transitions are functional, that is, each request followed by an acknowledge constitutes a complete handshake. Two-phase handshaking is also known as transition, two-stroke[1], two-cycle[1], and non-return-to-zero (NRZ)[2] signaling.

Since all transitions are functional the direction of dataflow directly determines the assignment of data valid and release to the handshake signals. On push channels every request event signals the beginning, and every acknowledge the end of a data-valid period. On pull channels this is the other way around. The timing diagrams for push and pull two-phase single-rail channels are depicted in Fig. 3.4, which shows a down handshake followed by an up handshake.

The data on the push channel is valid when the request and the acknowledge are in a different state (which can be detected with an exclusive-OR), whereas on the pull channel the data is valid when the request and the acknowledge are in the same state (which can be detected by an exclusive-NOR).

3.4 Four phase

A complete cycle in a four-phase handshake involves four (sequential) events, hence the name. One four-phase handshake consists of an up handshake followed by a down handshake, as shown in Fig. 3.5. (Of course other initial states could be chosen.) The down handshake is also referred to as the *return-to-zero* phase. Since four events

[1] Originally referring to combustion engines in which a complete fuel cycle in a cylinder requires only two piston strokes.

[2] Adopted from digital magnetic recording techniques. NRZ refers to a mode of recording in which the direction of writing current is reversed for every change in the binary sequence. One direction of surface magnetization then corresponds to a '1,' the other to a '0.' In NRZI encoding, the current is reversed only when a '1' is recorded. See Hoagland [42] for details.

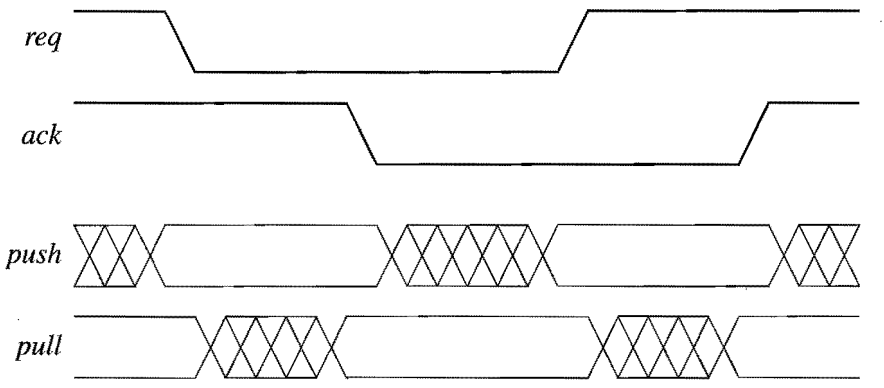


Figure 3.4: Data-valid schemes for two-phase single-rail channels. (The data is valid during the non-hashed periods.)

are used to designate a handshake, half of these events are essentially functionally redundant. Three interpretations can be distinguished that differ as to which events are labeled redundant. These interpretations are depicted in Fig. 3.5. In the *early* interpretation the return-to-zero phase is assumed to be functionally redundant. One might interpret this as a functional phase followed by a cooling down. The *late* view, in contrast, assumes the up handshake to be functionally redundant, such that a complete handshake can be interpreted as a warming up followed by a functional phase.

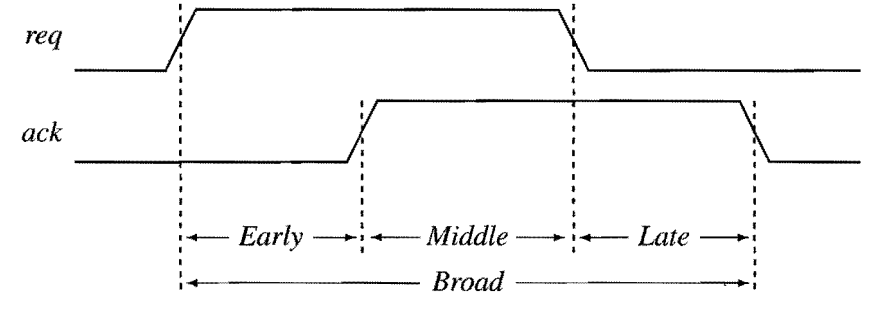


Figure 3.5: Four-phase handshaking

Four-phase handshake protocols are also known as return-to-zero (RTZ)[3], four-

[3]In digital magnetic recording, return-to-zero (RZ) recording refers to a mode in which a '1' is recorded by saturating a spot in one direction, and a '0' by saturating a spot in the reverse direction. Interestingly, a feature of this RZ encoding is that it allows for self-clocking, because an output signal is obtained during each bit interval. As Hoagland [42] put it: 'The RZ recording method (...) does

stroke[4], four-cycle[4], and level signaling.

3.4.1 Push channels

On a push channel the data-valid signal is encoded in the request and the data-release signal in the acknowledge. In the two-phase protocol this left us with only one choice, but in a four-phase protocol three conventions can be distinguished. If $req\uparrow$ is the data-valid signal, we can choose $ack\uparrow$ or $ack\downarrow$ as the data-release signal. In case $req\downarrow$ is the data-valid signal we can only choose $ack\downarrow$ as the data-release signal. These three conventions will be called early, broad, and late, respectively, and are depicted in the form of a timing diagram in Fig. 3.6.

The term 'broad' has been borrowed from Brunvand [18]. Early and late were coined by Craig Farnsworth. Early is referred to as narrow by Brunvand.

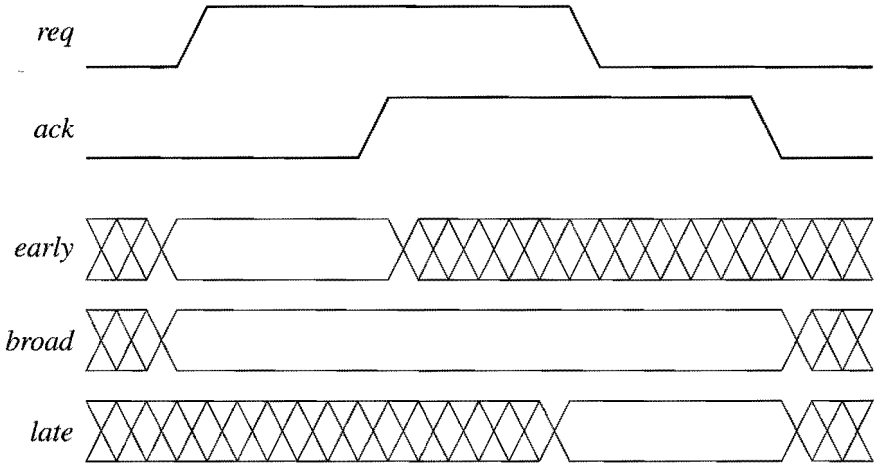


Figure 3.6: Data-valid schemes for *push* single-rail channels.

Although the data-valid periods are defined by the events that signal the beginning and the ending of such a period, the data-valid period can in all three cases be characterized by a Boolean function, by looking at the state of the control wires only. On a broad push channel, for instance, the data is valid when $req + ack$. On an early channel the data-valid period is defined by $req * \overline{ack}$, on a late channel by $\overline{req} * ack$.

possess a capability for almost asynchronous operation.'

[4]Referring to combustion engins in which a complete fuel cycle in a cylinder requires four separate piston strokes, namely intake, compression, power, and exhaust.

3.4.2 Pull channels

On a pull channel the data-valid signal is encoded in the acknowledge and the data-release signal in the request. In the two-phase protocol this leads to the observation that the data thus is valid between two handshakes, not during a handshake as on a push channel. In the four-phase protocol we can again identify three different data-valid schemes. The data-valid period can begin at $ack\uparrow$ and then end at $req\downarrow$ of the same handshake, or at $req\uparrow$ of the next handshake. The other choice is that the period begins at $ack\downarrow$ and ends at $req\uparrow$ of the next handshake. These three conventions are also called early, broad, and late, respectively, and are depicted as a timing diagram in Fig. 3.7.

The early convention for pull channels can also be called the *middle* data-valid scheme, since the data is valid during the middle phase of the handshake protocol, namely from $ack\uparrow$ till $req\downarrow$. Like on push channels, the data-valid period can be defined by looking at the *state* of the request-acknowledge signals. The broad data-valid period, for instance is characterized by $ack + \overline{req}$

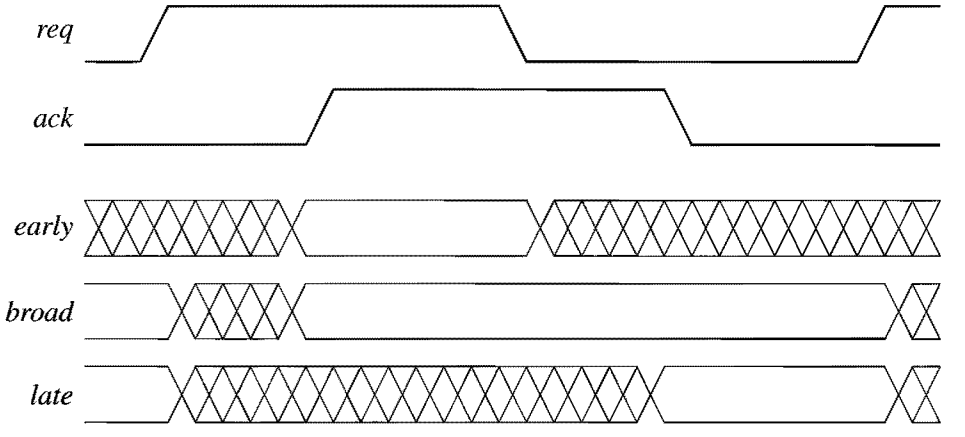


Figure 3.7: Data-valid schemes for *pull* single-rail channels.

3.4.3 Double rail

Although the double-rail data-encoding scheme can be applied in both two- and four-phase handshake protocols, it is generally only used in combination with four-phase handshaking. The up-going phase of the handshake is then used to assert the data, and in the return-to-zero phase all wires are reset. The data on a double-rail channel is completely defined if all pairs of wires are in a complementary state. If

all wires are low the channel is inactive, and the two wires of a pair must never be high at the same time.

If the data on a double-rail channel is completely defined, one might say that the data is valid, in the sense that the status of the one-wires encodes the message, and the zero-wires hold the complement (bitwise inverse) of the message. Both on push and on pull channels this interpretation of when the data is valid corresponds with the early data-valid scheme as defined for single-rail channels.

3.5 Single track

The single-track handshake protocol, as described by Van Berkel and Bink [9], combines the advantages of two- and four-phase handshaking. It has the minimum number of transitions, that is, two per handshake, and after each handshake the initial state is restored. This is achieved by combining the request and the acknowledge onto one wire. This idea can be combined with single-rail data encoding, which results in a data-valid scheme that is similar to that of the two-phase handshake protocol, see Fig. 3.8.

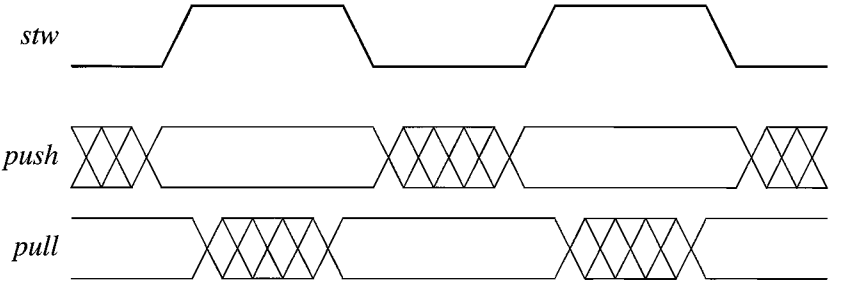


Figure 3.8: Data-valid schemes for single-track single-rail channels.

The single-track wire (denoted by *stw* in Fig. 3.8) combines request and acknowledge events. Each rising transition denotes a request event, and the falling transitions denote acknowledge events. If we assume the single-track wire to be low initially, as depicted in the figure above, then the data on push channels is valid when that wire is high, and on pull channels when *stw* is low.

3.6 Minimum-power schemes

In the definition of the data-valid schemes we have only defined what rules should be obeyed during the data-valid periods. The sender then has to keep the data stable

so that the receiver can safely interpret it. Data-valid periods are separated by what we might call *data-change* periods, during which a sender can prepare new data. For this period we have not defined any restrictions yet.

An additional restriction to the data-valid scheme could be to limit the number of transitions in the data-path to at most one per bit. This obviously is the minimum number of transitions that should be allowed since the sender must be able to switch between any two messages. We call data-valid schemes with this additional constraint *minimum-power* schemes, since they minimize the number of transitions in the data channels, and thus minimize the energy consumption (assuming a CMOS implementation, in which the energy consumption is dominated by switching energy [84]).

In general we allow for an arbitrary number of transitions in the datapath during data-change periods. This means that spurious transitions due to deep combinational logic are allowed. Minimum-power schemes are more strict, which limits the implementation freedom and results in more circuit area.

3.7 Extended data-valid schemes

The relation between sender and receiver on a handshake channel is symmetrical in all data-valid schemes that have been addressed so far. A sender may prolong the data-change period by postponing the data-valid signal, and a receiver may use the data-release signal to extend the data-valid period.

This symmetry is not always practical. Especially the relation between the data-release signal and the end of a data-valid period need not always be that strict. Two situations can be distinguished, namely, one in which the data remains valid after the data-release signal, and one in which the data may change even before the data-release signal has occurred.

3.7.1 Reduced schemes

One situation that can easily be envisioned is that the receiver cannot prolong the data-valid period, that is, that the data-valid period may end before the data-release signal arrives. For each data-valid scheme that has be defined earlier in this chapter such a *reduced* scheme can be defined.

A reduced broad scheme on a push channels means that the data is stable after the up-going edge of the request, but after a down-going edge of the request the data may change shortly. In reduced early or reduced late schemes on push channels the data is only valid during a short period after the data-valid signal. The exact duration of a reduced data-valid period may vary, and is not taken into account here.

A reduced data-valid scheme may, for instance, be caused by the use of dynamic logic. This may imply that after a data-valid signal there is only a limited window during which the data may be assumed stable. With most styles of dynamic-logic, however, the symmetry between sender and receiver can be restored by adding staticizers (trickle transistors).

Another reason to use a reduced data-valid period may be that the sender is informed via some other path (that is, not via the data-release signal) that the data it is keeping stable is no longer required. In the implementation of the handshake variable in Chapter 5 we encounter this situation.

3.7.2 Prolonged schemes

Although the data-release signal is used by a receiver to inform the corresponding sender that the data is no longer required, the data may remain stable for quite some time. Basically, for each channel we should not only look at the formal ending of the data-valid period (as signaled by the data-release signal), but we might try to determine when the data may actually start to change and thereby become invalid. For the different data-valid schemes that have been introduced, *prolonged* versions may be defined, in which the data remains valid for some period after the data-release signal.

In the implementation of Tangram assignments, as discussed in Chapter 4, pull channels are used with such a prolonged data-valid scheme. In that context, the data-valid period is actually under control of the environment, more specifically, the transferrer that controls the data-transfer. The data-valid scheme that is used in that context is the prolonged early scheme. This data-valid scheme can also be considered as a reduced form of the broad scheme.

3.8 Synchronous data-valid schemes

Single-rail data encoding essentially is very similar to the way data-validity is organized in a synchronous circuit. In a synchronous circuit the data is generally stored in flip-flops (D-types, FFs) which are controlled by a central clock. The timing assumptions are then formulated in terms of data-stability assumptions at the inputs and outputs of these flip-flops.

An example of a synchronous data-valid scheme is shown in Fig. 3.9. In this diagram *clk* refers to the global clock, D to the data-inputs of the flip-flops, and Q refers to their outputs.

For such a data-valid scheme, at least three parameters are important. First of all, the data at the input (D) of the flip-flops should be stable (valid) before the rising

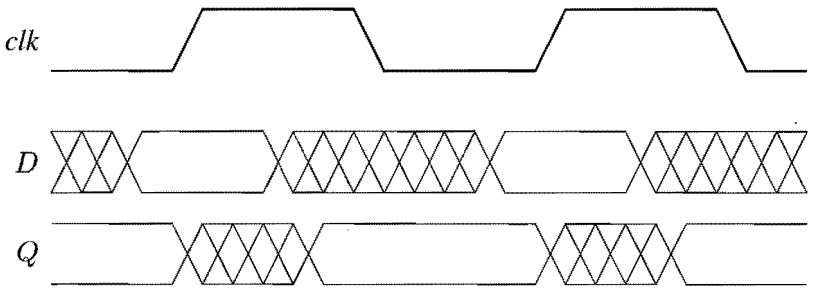


Figure 3.9: Data-valid schemes for a synchronous system, assuming positive-edge triggered storage elements and a global single-phase clock.

edge of the clock, by a margin that is called the *set-up time* of the data. After the clock edge the input data should remain stable for at least the *hold-time*. A third parameter that is specified is the time it takes for new data to be assimilated in the flip-flops and to propagate to the outputs (Q). This is generally called the *propagation time* through the flip-flops. Similarly, there is also a propagation time through the logic between the flip-flops, which is the time it takes for the new input data D to stabilize again. For a more extensive discussion of timing assumptions in a synchronous (clocked) circuit one may consult [77], [3, Chap. 8], and [84, Chap. 5].

One of the differences that can be noted between handshake data-valid schemes and synchronous data-valid schemes is that in the synchronous diagram we refer both to the current state (Q) and to the next state (D), whereas in the handshake schemes only the current state is recorded. In the implementation of the respective circuits this difference shows in the choice of storage elements. Synchronous circuits generally use flip-flops, whereas in asynchronous circuits latches are common.

In the single-rail data-valid schemes, the validity of the data is defined in relation to *local* signals (*req*, *ack*). Data-validity in a synchronous scheme is always related to a *global* signal: the clock. In terms of communication this means that in a synchronous system the time that is available for a communication is fixed, and predefined ('dictated') by the clock period. In asynchronous (handshake) communication the length of the data-valid period may vary between different handshake channels, and even between different communications along such a channel.

3.9 Options

In this chapter, several single-rail data-valid schemes have been introduced. Each scheme defines a contract between a sender and a receiver that communicate through

a handshake channel. Such a contract specifies when the receiver can reliably inspect the data and when the sender may change the data. In the next chapters the implementation of handshake components and handshake circuits is addressed. Handshake components communicate via handshake channels. In the single-rail implementation of handshake circuits we have to choose a data-valid scheme for each handshake channel.

The two-phase and single-track handshake protocol do not allow any choice in the assignment of data-valid and data-release to the request and acknowledge signals. They may still allow for some interesting implementations of handshake circuits and components, but as far as the data-valid schemes are concerned, there is not much of a choice.

The four-phase handshake protocol, in contrast, allows for early, broad, and late data-valid schemes. The challenge in the single-rail implementation of handshake circuits now is to make the best possible choice. This choice may depend on the performance criteria that is considered. A solution that is optimal in terms of (minimal) area requirements may not be optimal for maximum speed or minimal power. The main cost criterion in this thesis is area.

In the discussion of single-rail implementation of handshake circuits, we have chosen to follow a top-down approach. We first address the choice of a data-valid scheme for each handshake channel in a handshake circuit, based on the compilation scheme from Tangram to handshake circuits. For the handshake components this limits the number of alternatives that have to be considered. The choice of data-valid schemes for handshake channels in Tangram handshake circuits is covered in Chapter 4; the implementation of the components is discussed in Chapter 5.

Chapter 4

Handshake Circuits

The subject of this chapter is the implementation of handshake circuits. We focus on the four-phase single-rail implementation of the datapath and assume a four-phase implementation of the control. It turns out that, contrary to common belief the return-to-zero phase of the four-phase handshake can be scheduled such that it is fully productive. This alleviates an often mentioned disadvantage of four-phase over two-phase handshaking.

In the discussion of handshake circuits the compilation scheme from Tangram to handshake circuits is followed. We therefore first investigate the structure of Tangram handshake circuits, and then concentrate on the various compilation rules that introduce datapath components. From the structure of Tangram handshake circuits, an assignment of data-valid schemes to the handshake channels can already be evaluated. For each compilation rule this leads to the choice of an optimal data-valid scheme for the handshake channels and components that are involved.

If we would discuss the implementation of handshake circuits in general, then it would be natural to follow a bottom-up approach, that is, first elaborate on the implementation of the individual components, and then look at interconnections of these components. The advantage of the top-down approach that is pursued in this chapter is that it limits the number of alternatives that have to be considered in the implementation of handshake components, as discussed in Chapter 5. For most components only one combination of data-valid schemes on its channels has to be taken into account.

4.1 Structure

Handshake circuits can in general be split into a control part, a data part, and an interface part. The control part is then composed of handshake components that

only have control (nonput) handshake channels. The components that constitute the data part have a handshake interface that consists of data channels only. Interface components have both nonput and data handshake channels and thus form a natural separation between the control path and the data path of a handshake circuit.

Throughout this chapter the data components are furthermore partitioned into pull, push, and passive components. Push and pull components have at least one active and one passive handshake port. Push components are data components that have an interface consisting of push handshake channels only. Similarly, the handshake interface of pull components consists of pull channels. Passive components have only passive handshake ports. This partition in three types covers all Tangram data components. Tangram does not use all-active components, for example.

With respect to the above partitioning of Tangram handshake components, the general structure of a Tangram handshake circuit is as shown in Fig. 4.1. The figure schematically represents both the internal structure of a handshake circuit and the possible handshake interfaces to the environment. Non-handshake interfaces (both internally and externally) are not taken into account.

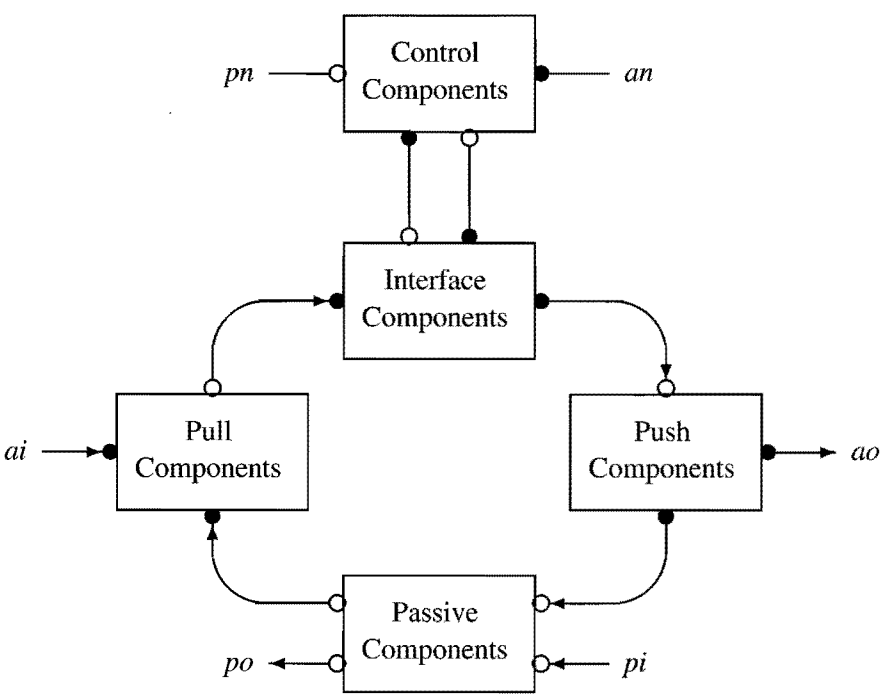


Figure 4.1: General structure of Tangram handshake circuits.

Synchronization between control components and the environment takes place

via nonput channels, and can be either active (*an*) or passive (*pn*). One of the passive nonput channels generally is the activation channel, along which the operation of the handshake circuit is initiated. This corresponds to starting the execution of the Tangram program. Control components may furthermore activate interface handshake components, and the other way around, interface components may activate control components.

Pull components are used in Tangram to implement operations on data, such as addition. The evaluation of such operators is demand driven and the operands are collected from passive components such as variables and constants. The general function of pull components is to collect, modify, and output data upon request. This data can also be collected from the environment of the handshake channel, either directly via an active input (*ai*), or indirectly through a passive component from a passive input (*pi*).

An example of a push component is the multiplexer, cf. Section 2.3.2. It is used to merge two streams of data onto one channel, and as such, implements sharing of push handshake channels. Push components also take care of the communication of data to the environment of the handshake circuit, either directly via an active output (*ao*), or indirectly through a passivator via a passive output (*po*).

The link between the control, push, and pull components are the interface components. The predominant interface between control and data in a (Tangram) handshake circuit is the transferrer, which controls a so-called *data transfer*, that is, a Tangram input action, output action, or assignment. Other interfaces are the *do* and *case* components, which deal with repetitive and conditional execution. These two components are often referred to as control components, but in this thesis they fall into the class of interface components because they deal with data for the evaluation of their guards.

In the rest of this chapter we first explore Tangram assignments, and then briefly look at input and output communications. After this iteration and selection are discussed. Initially, the interface components are studied independently. An important issue, however, is that of sharing, which reduces the implementation freedom of the interface components, since at the interface to the datapath (push or pull channels) they often have to be compatible. The impact of sharing is discussed at the end of this chapter.

4.2 Assignment

An investigation into general handshake data-transfers, with input, output, assignments, and combinations thereof would be too complex as a starting point. Since assignments constitute the majority of data-transfers in a Tangram program, we start

by zooming in on a relatively simple assignment, namely `z:=x+y`. This turns out to be a good starting point to explore the rich world of four-phase single-rail handshake implementations.

In the rest of this section we concentrate on the data-transfer action described by the Tangram fragment `z:=x+y`, in which `x`, `y`, and `z` are integer variables within some range and `+` stands for addition. We furthermore assume that there is also another assignment to `z`, resulting in a multiplexer on the write-port of the corresponding handshake variable. The corresponding handshake circuit is depicted in Fig. 4.2.

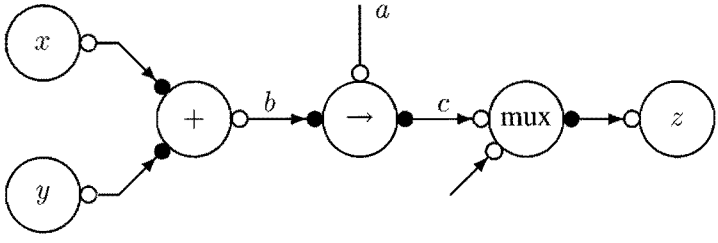


Figure 4.2: Compiled handshake circuit for `z:=x+y` with an additional multiplexer on z.

The assignment is initiated by a request signal on handshake channel a, which connects to the transferrer. This transferrer then sends a request for data along channel b. The adder component forwards this request to handshake variables x and y, which correspond to the Tangram variables `x` and `y`. The variables then assert data, which is processed by the adder and passes through the transferrer to channel c. The multiplexer forwards this data to the write port of handshake variable z, in which it is subsequently written. The acknowledge thereof passes through the multiplexer and the transferrer and finally arrives at channel a, which completes the data transfer.

For the realization of these components we have to choose a handshake protocol and a data-valid scheme per channel. In subsequent sections, several options pass in review. We begin with a two-phase implementation, which gives us a chance to identify in more detail the work that has to be done during this assignment. After this, several four-phase implementations are addressed.

4.2.1 Two-phase

One of the properties of two-phase handshake circuit implementations is that there is virtually no degree of freedom in the implementation, apart from post-optimization

issues. The implementation of the control part of the transferrer, for example, is as shown in Fig. 4.3. For the data-part we have chosen here to use wires only in the implementation. The two-phase pull data-valid scheme on channel b assures that the data on channel c is valid at least during the corresponding push data-valid period. Therefore, no latching is required in the transferrer.

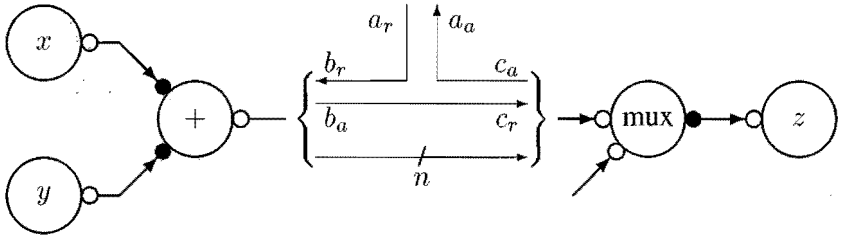


Figure 4.3: Data transfer with early transferrer.

Below we take stock of some of the tasks that have to be performed in the various data components during the data transfer. We do not elaborate upon their implementation yet; for that we refer to Chapter 5.

variable (x,y) The readports of variables x and y have to release new data upon request, and keep this stable between handshakes.

adder The function of the adder, apart from adding operands and producing a result, is to provide a data-valid signal at its output to signal the validity of the result. This can be established by delaying the request-acknowledge signals for a time that matches that of a worst-case addition.

multiplexer Two tasks can be distinguished in the multiplexer that relate to the datapath. First of all, the appropriate input has to be selected, and secondly, the control signals have to be delayed such that they provide a matched path with the delay that a bit encounters in passing through the multiplexer.

variable (z) Tangram's compilation scheme basically dictates normally-opaque implementations of the latches, though sometimes, via some post-optimization procedure, one can get away with this, see Section 8.1.3. Assuming a normally-opaque latch control, a data-transfer requires the latches to be opened, valid data should be allowed to enter (possibly via some matched delay), and the latches have to be closed again.

From the above description it is already clear that a direct two-phase implementation of these components leads to quite complex circuitry. Especially the implementations of the variable and the multiplexer require quite some hardware. We do

not further elaborate upon this here, but switch to the greener pastures of four-phase handshake protocols, which allow a variety of choices.

4.2.2 Early four-phase

The wire-only implementation of the transferrer in Fig. 4.3 resembles the double-rail implementation. When activated along channel a, this transferrer first performs an up handshake along b, then one along c, after which it sends an acknowledge on a. The return-to-zero phase follows the same cycle.

Given the wire-only transferrer of Fig. 4.3 we can determine the data-valid scheme on channel c directly from that on channel b. In this section we assume the early data-valid scheme on channel b, in the next section we look at late and broad data-validity.

An early data-valid scheme on channel b and the other pull channels implies that the data is guaranteed to be valid in the transferrer during the period as shown in Fig. 4.4. This means that one may assume an early data-valid scheme on c (the period labeled as 'II' in the figure) as well. This data-validity is prolonged a bit (during III), but valid data may not be assumed anymore after this.

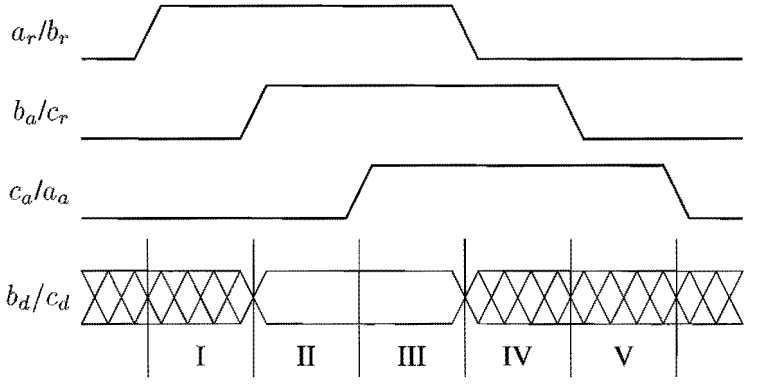


Figure 4.4: Early data transfer

This limited data-validity has an important impact on the data-transfer action and the implementation of the components involved. First of all, it implies that after the up-going phases of all handshakes (that is, directly after $a_a\uparrow$, which signals the end of phase II in Fig. 4.4) the data-transfer is basically complete, in the sense that new data must already be latched in the variable (z). Both the control phase (III) and the return-to-zero phases (IV, V) of the data-transfer are, therefore, functionally redundant as far as the actual transfer of data is concerned.

The functional redundancy of the return-to-zero phase of the handshake should be taken into account in the implementation of the delay-matching in the adder (and, in general, in pull components). To minimize the time spent in the redundant phase, one could consider using asymmetric delays. The implementation should be such that during phase I of the data-transfer, when data is collected from the latches, the actual delay-matching takes place. The return-to-zero phase (IV), however, does not contribute to the data-validity, and should be implemented as a quick reset to minimize the time wasted in this phase.

The implementation of the handshake variable with an early data-valid scheme on its write port requires quite a few gates, as is shown in Section 5.3.2. During the up phase of the write handshake the latches in the variable must be opened *and* closed again, for which a sequential control circuit is required.

In the multiplexer we must first make sure that during the up phase of the handshake, the data on the output channel comes from the selected input channel, and that it is valid. The first requirement can only be met after the input channel announces itself by starting a handshake (in this case, on channel c). After selecting the actual input, the data must be allowed to ripple through the multiplexer before a data-valid signal issued.

In conclusion, in the early scheme all work is performed in the up phase of the handshakes. The return-to-zero phase serves only to reset the request-acknowledge circuits and is a functionally redundant cooling-down phase. For the latches, operators, and multiplexers, this leads to inefficient implementations.

With the early data-valid scheme, all work has completed after the up phase on control channel a has finished. This can be exploited in the implementation of the control components. Since the return-to-zero phase in the datapath is redundant, we might just as well implement the control path such that its return-to-zero phase is also redundant. This means that we can implement the control components with the early protocol, which allows for some parallelism between return-to-zero and functional phases. This may result in overall faster circuits, since after the functional phase of each handshake a kind of quick reset can be implemented. In general, however, it leads to more complex implementations of the components, which reduces the possible speed advantage, and is generally not optimal for power and area.

In the context of micropipelines the early protocol is rather popular, see for instance [25]. The main reason for this is that some of the disadvantages mentioned here do not hold for micropipelines. Although asymmetrical delays have to be used, the implementation of the variables can be kept relatively simple. A more extensive discussion of this is given in Section 8.1.3.

4.2.3 Late and broad four-phase

If the transferrer is implemented as shown in Fig. 4.3, and the data-valid scheme on channel b is late, then the relation between the handshake signals and the data-validity is as shown in Fig. 4.5. For channel c, and thus for the multiplexer and the handshake variable that are involved in the data-transfer, this implies that late data-validity must be assumed.

The data-transfer is now a bit more efficient, since the up-going phase of the handshake can be used as a sort of 'warming up.' Phase I of the transfer is redundant, but in phase II the multiplexers can already be set in the requested state and the latches in the variable that is to be updated (z) can already be opened. The actual delay-matching then takes place during phase IV, after which the data is latched in phase V.

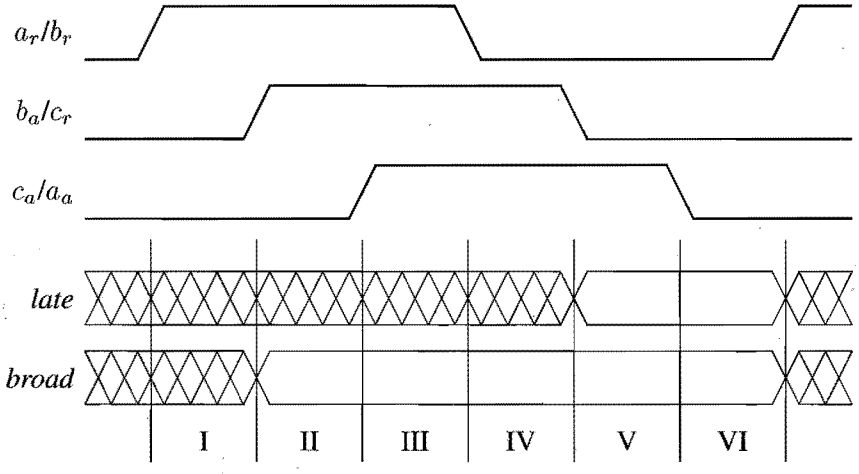


Figure 4.5: Data transfer with a late or broad data-valid scheme on b (and c).

If the late data-valid scheme is assumed on all pull channels, this means that the up-going phase of their handshakes cannot be exploited. For the implementation of operators, asymmetrical delays should thus be used that have minimal delay in the up phase of the handshake, and a matched delay in the down phase. (This in contrast to the early scheme, where the asymmetry is the other way around.)

The implementation of multiplexers and of assignments to variables is already a bit more efficient than in the early scheme, since the up phase of the handshake can be used to prepare the arrival of valid data.

If the data-valid scheme on channel b is broad, we may also assume the broad data-valid scheme on channel c, as may be concluded from Fig. 4.5. Phase I now

clearly is functional, since it establishes data-validity on channels b and c. The work on the push side of the transferrer, that is, in the multiplexer and the variable, can again safely be spread over the up and the down phases of the handshake, since the data on channel b is guaranteed to remain valid. Phase IV of the handshake, however, should be kept as short as possible (asymmetrical delays in the adder), since it does not add anything, as the data was already valid.

An apparent drawback of both the late and the broad scheme is that data is assumed to be valid *between* handshakes. For the passive components that are involved as data-sources in such an assignment (x, y), this implies that they have to keep data valid at the outputs after the handshake has completed. Especially for handshake variables this leads to area-inefficient implementations, since in each readport data must be latched.

Note that broad, early, or late on b leads to a *prolonged* broad, early, or late data-valid scheme on c, respectively. On the other hand, for an early, broad, or late data-valid scheme on channel c, only a *reduced* early, broad, or late data-valid scheme on b is required. In the next sections we exploit the last observation.

4.2.4 Mirrored four-phase

The implementations of data-transfer discussed so far are all based on an implementation of the transferrer in which the request/acknowledge signals follow the same path as the direction of the data-transfer, that is, the handshake on the input channel is started before that on the output channel. An interesting alternative is obtained when the transferrer is mirrored, as in Fig. 4.6.

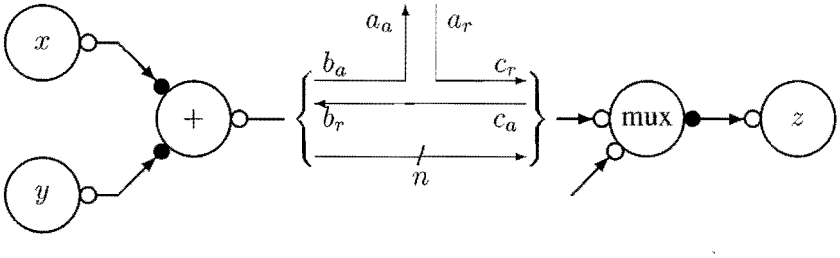


Figure 4.6: Data transfer with 'mirrored' transferrer.

Since the handshake on channel c has completed before the return-to-zero phase on channel b is started, the early data-valid scheme on channel b is the only viable option. From Fig. 4.7 one can see that in this implementation of the data-transfer, early data-validity on channel b is combined with the late data-valid scheme on channel c.

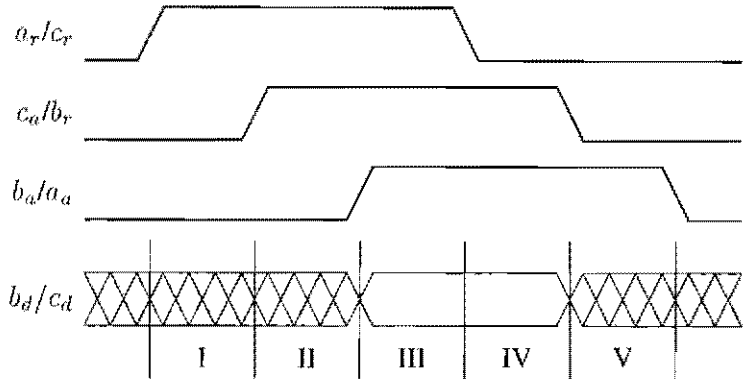


Figure 4.7: Mirrored data transfer

An advantage of this scheme is that, in the data-transfer, only the return-to-zero phase on channel b (denoted by V in Fig. 4.7) is redundant. Phase I can be used to set the multiplexer in the correct state, and to already open the latches in the variable. Phase II is then used for delay-matching, phase III adds an additional safety-margin to the bundle, and during phase IV the actual latching takes place.

A potential disadvantage of this mirrored implementation of the assignment is that the data may actually still be changing when the latches in the variable are already opened. This can cause the output of these latches to make more than one transition, which may not be optimal from a low-power point of view.

4.2.5 True four-phase

In the organization of data-transfer as we have discussed so far, we have looked at various combinations of assigning data-valid and data-release 'meanings' to transitions on handshake signals. In all choices that have passed in review we ended up with some form of (productive) redundancy of handshake events. Especially for the datapath operators this is a disadvantage, since it demands asymmetric implementation of delays, which is more complex (and less efficient) than symmetric delays.

The mirrored four-phase protocol implementation of the previous section has an interesting advantage, namely that both in the up and the down phase of the handshakes on the push channels, useful work could be done. The challenge now is to organize the data-transfer such that we can also spread the work on the pull channels over the up and down phases.

A straightforward way to spread the work in the pull components that are involved in the assignment over the up and the down phase is to let these phases each

contribute to the delay matching. Per phase we can then match half of the delay of the datapath.

If we spread the work on both the push and the pull channels over the up and the down phase, then we have basically assigned a meaning to all handshake events. On pull channels, for example, the up-going acknowledge signals that half of the delays have been matched, and the down-going acknowledge indicates that matching is complete and data is valid. Since all handshake events now have a function, we call this scheme the *true* four-phase scheme.

Since the data on pull channels is valid only after the completion of the handshake, the transferrer has to be implemented as shown in Fig. 4.8, provided that we want to keep the implementation simple, that is, without latches in the datapath. This implementation is actually the same as that used for two-phase and early, broad, and late data-transfer, cf. Fig. 4.3.

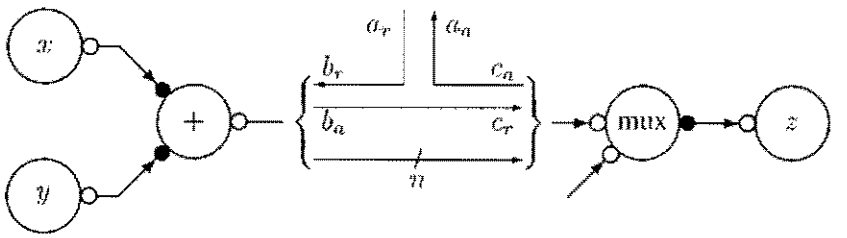


Figure 4.8: True four-phase data transfer: implementation of transferrer.

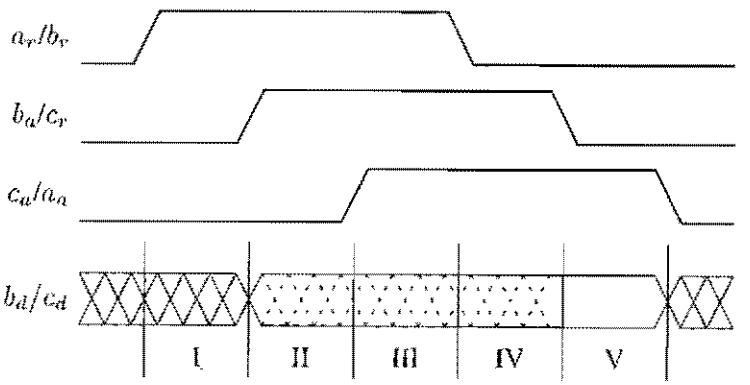


Figure 4.9: True four-phase data transfer: data-valid scheme, with dotted lines indicating 'half' validity of the data.

With reference to the true four-phase timing diagram of Fig. 4.9, the work is spread over the different phases of the handshakes on channels a, b, and c as follows.

I. The request signal ($b_r\uparrow$) is distributed, generally via forks, to the readports of the variables. In the acknowledge phase the delay of the data is matched.

II. Multiplexers are set and the delay through the multiplexer is matched. The latches in the variables are opened (switched from opaque to transparent).

III. The time required by the control circuit between $a_a\uparrow$ and $a_r\downarrow$ (at least one CMOS inversion) adds an additional safety margin to the data-bundle.

IV. The same path as in phase I is followed, which means that the delay in the datapath is again matched. After this the pull handshake components are back in their initial state, but the data is still valid.

V. Delay through the multiplexer is matched again in the control circuit of the multiplexer. Data is latched in the variables. The setting of the multiplexer switches should be maintained until the latches in the variables are closed again.

From the above description of the true four-phase implementation of the assignment we can deduce the required data-valid schemes on b and c. Although both the up and the down handshake on b have a function, it is the falling edge of the acknowledge that actually signals the validity of the data. A late data-valid scheme on b is thus assumed. For channel c this implies a prolonged late data-valid scheme. The other way around, however, we only need a 'pure' late data-valid scheme on channel c, which means that a reduced late data-valid scheme on b suffices.

In Tangram handshake circuits, mutual exclusion between read and write accesses to variables is assured. For assignments this implies that variables that are being written are not read during the same assignment. During a true four-phase data-transfer it is guaranteed that at least until the completion of the handshake on the nonput channel of the transferrer (channel a in the handshake circuit above) the variables that are read from are not written. We can thus keep the implementation of the variable simple (that is, without latching data in readports, see Section 5.3.2) and still guarantee the data at the input of the transferrer to be stable during at least the reduced late period from $b_a\downarrow$ till $a_a\downarrow$.

The true four-phase scheme has some obvious advantages. It allows for symmetric delay matching and efficient multiplexer and latch control circuits. It does depend, however, on the mutual exclusion between read and write accesses to variables, and can therefore be applied straightforwardly to Tangram handshake circuits. For handshake circuits in general, however, a true four-phase solution may not always be efficient.

A consequence of the true four-phase scheme is that control should be implemented with the broad four-phase protocol, such that mutual exclusion between data transfers is guaranteed during the complete four-phases of the handshake protocol. This assures that for two transferrers of which one implements an assignment to a variable x, and the other requires a read action from x, their activation channels will never be active (involved in a handshake) simultaneously.

4.2.6 Low power

The emphasis has so far been on minimizing the (productive) redundancy of the transitions in the request-acknowledge path on handshake channels. In this section we address the implementation of assignments with low power as the main target.

A separate investigation into low-power implementations is only meaningful if the implementations shown earlier suffer from power inefficiencies. The true four-phase scheme, which is optimal in terms of minimal redundancy, indeed does. In that context, for handshake variables we know that the value on a read port is no longer required to be stable as soon as a write handshake is initiated. The arrival of a write request essentially signals that all reduced late data-valid periods at the input of transferrers can be ended. During a write action on a variable we are thus free to change the data at the read ports. This may lead to quite some redundant transitions at the data wires of a read port, since not every write action is necessarily followed by a read action.

If we want to minimize the number of transitions on the handshake channels, we should allow for a maximum of only one transition per data-wire between data-valid periods. This restriction, in combination with spreading the actual work over the up and the down phases of the handshakes, is called the *low power* scheme. It thus is a minimum-power variant of the true four-phase scheme.

This low power variant mainly affects the implementation of the handshake variable. In the standard true four-phase scheme, we could latch the data in a write port, and update all read ports after (or during) a write handshake. In the low-power variant, we have to latch the data in the read port. Since read and write accesses to variables are still assured to be mutually exclusive, and the data is not allowed to change between handshakes, we may assume data at the input to the write port to be stable during all read actions. We thus do not have to latch data in write ports of variables, but effectively use the write channel to store the current state.

Although the low-power variant indeed minimizes the transitions on the handshake channels in the datapath, it is not necessarily a power-efficient solution. It is likely that the additional power consumption in the handshake variables, due to the complex read ports, sweeps away the power advantage that is gained by minimizing the number of transitions on the channels.

4.2.7 Comparison

The important characteristics of the data-valid schemes for assignments, which are addressed so far, are summarized in Table 4.1. The true four-phase scheme has the best prospects when it comes to efficiency, both with respect to area and time.

4.3 Communication

Input and output communications are special cases of data-transfer, for which the true four-phase scheme may not always be the best alternative. For communication, different schemes may be better suited, for instance, because they reduce the cost of some components or lead to more natural data-valid schemes at off-chip interfaces.

Before we start looking at input and output separately, we first investigate the combined effect, namely that of communication within a handshake circuit. The parallel composition of Tangram fragments `a!E` and `a?x` is semantically equivalent to assignment `x:=E`. Communication may thus be interpreted as a distributed assignment.

In the compilation scheme from Tangram to handshake circuits, both input and output are implemented with a transferrer. For output statement `a!E` its transferrer collects the value of expression `E` and sends this along channel `a`. The transferrer for input statement `a?x` collects a value via channel `a` and stores this in variable `x`. Since both communications along `a` are active we need a so-called *passivator* to synchronize these two handshakes.

The general structure of the handshake circuit that is involved in such a communication (or distributed assignment) is depicted in Fig. 4.10. The component in the center of the figure is the passivator. The left transferrer corresponds to the output action, the right one to the input action. The boxes labeled with 'mux' refer to a handshake circuit consisting of (push) multiplexers only, and allow access from multiple sources to their respective outputs. The box labeled 'dmx' refers to demultiplexers, which allow multiple input actions to (mutually exclusive) use an input channel.

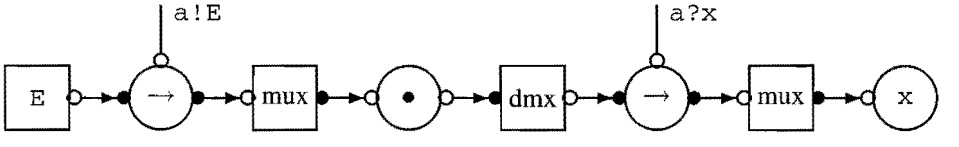


Figure 4.10: Synchronization within a handshake circuit through a passivator.

The challenge in the implementation of input, output, and synchronization is

Early scheme	
Control	Early or broad
Data	Early for both push and pull channels
Operators	Asymmetrical delay-matching (quick reset)
Variables	Complex latch control, simple read-port
RTZ-phase	Fully redundant; pure overhead (quick reset)
Late scheme	
Control	Broad, late would complicate variables
Data	Late for both push and pull channels
Operators	Asymmetrical delay-matching (quick set)
Variables	Simple latch control, complex read-ports
RTZ-phase	Functional; up-phase is half-functional
Broad scheme	
Control	Broad
Data	Broad for pull channels; broad or late for push channels
Operators	Asymmetrical delay-matching (quick reset)
Variables	Simple latch control, complex read-ports
RTZ-phase	Half redundant
Mirrored scheme	
Control	Broad
Data	Early for pull channels; late for push channels
Operators	Asymmetrical delay-matching (quick reset)
Variables	Simple latch control, simple read-ports
RTZ-phase	Redundant on pull channels, functional on push channels
True four-phase scheme	
Control	Broad
Data	Reduced late for pull channels; late for push channels
Operators	Symmetrical (halved) delays for matching
Variables	Simple latch control, simple read-ports
RTZ-phase	Fully functional
Low-power scheme	
Control	Broad
Data	Late for both push and pull channels
Operators	Symmetrical (halved) delays for matching
Variables	Latched readports, write port wires only
RTZ-phase	Fully functional

Table 4.1: Characteristics of different handshake implementation of assignments.

to find an implementation in which the data-valid schemes that are involved match nicely at the interfaces, and this against minimal cost.

The central component in the above handshake circuit is the passivator. This component can be implemented cheaply, that is, against constant costs, independent of the word size, if the data-valid scheme on its output is early. For any other data-valid scheme the costs are linear in the number of bits, see Section 5.3.3. For the input channel of the passivator the broad data-valid scheme leads to the cheapest realization, but against constant cost the passivator can also deal with early and late input channels.

These costs characteristics of the passivator have an impact on the implementation of input and output, which are discussed separately. We also discuss the implications for external communications, such as off-chip.

Input

The optimal choice for the passivator is to have an early data-valid scheme on its output. For the pull components between the passivator and the input transferrer this is no problem, as long as there are no operators in that path. Fortunately, Tangram does not allow inputs in expressions, and all input actions are of the form `a?x`. This means that inputs are stored immediately after collection, without intermediate modification through operators. Therefore, for the complete pull path from the passivator to the transferrer, the early data-valid scheme is fine.

For the multiplexers and the variables that are involved in an input action, a data-valid scheme that has data valid during the return-to-zero phase (that is, at least during the late period) is optimal.

The mirrored data-transfer scheme, as discussed in Section 4.2.4, exactly combines early pull with late push, and thus seems a good choice in the sense that it minimizes the implementation cost of the passivator, the multiplexer, and the variable.

Output

The handshake circuit around the output transferrer is very similar to that of a transferrer in an assignment. At the input side of the transferrer we may have operators which require delay-matching, and at the output side we encounter multiplexers that have to be switched to the correct state. Therefore, it seems a good choice to implement the output data-transfer based on the true four-phase protocol.

For the input of the passivator this implies that the data may already be valid during the up phase of the handshake, but it is safer to interpret the data during the

down phase. We should thus assume the late data-valid scheme at the input of the passivator.

Internal

For communication inside a handshake circuit, the combination of the choices for input and output as described above, leads to a solution in which mirrored input and true four-phase output are combined. An advantage of this scheme is that both transferrers that are involved can be implemented with wires only. Furthermore, the cost of the passivator is low, since its data part consists of wires only.

One should observe that concentrating the cost of communication in the passivator is better than distributing it over the transferrers, since the passivator is possibly shared between many input and output actions. For each passivator there will generally be multiple transferrers in the handshake circuit.

External

The choices that have been made for input and output also have an impact on communication at the boundary of handshake circuits with the external world, which in general is non-handshaking. There are four cases that can be considered, namely, input and output, which can both be active or passive.

For active input the mirrored transferrer assumes an early data-valid scheme on the associated pull channel. For the external of the handshake circuit it follows that the handshake circuit first requests for input, and then the up-going edge of the acknowledge is interpreted as the data-valid signal. The falling edge of the request signals the end of the data-valid period, which means that the input data need no longer be stable. The advantage of this input scheme is that the requirements for the environment are minimal. It must only hold the data valid between two handshake events, and between handshakes it can prepare new data. This is of course more efficient than preparing the data after the request, as would be required in the broad or late pull data-valid scheme, since then the handshake circuit would be temporarily stalled.

Passive input takes place through a passivator along a push channel. The implementation of this passivator can be easily tuned to any data-valid scheme. The cheapest implementation is possible if the data-valid scheme is broad, but then the environment should keep the data stable during the complete handshake. Generally this is no problem, and new data can be prepared between handshakes, without stalling the handshake circuit.

During active output (on a push channel) the late data-valid scheme is guaranteed, which means that the falling edge of the request is the data-valid signal. The

broad and early data-valid scheme can be realized against low cost, via data-valid converters, cf. Section 5.4.1. All data-valid schemes, however, allow the data to change after the handshake has completed. Except for the minimum power variants, the data may even change more than once. Especially for off-chip communication it may be effective to filter out these spurious transitions by latching the last communicated values at the interface.

For passive output via a passivator along its pull channel, the early data-valid scheme can be realized with the lowest cost. This scheme, however, allows the data to change between handshakes and, therefore, also allows for spurious transitions during this period. For off-chip passive output it may, therefore, be better to assure the broad data-valid scheme by latching the data in the passivator. This not only minimizes the external transitions, but also maximizes the time during which the environment can safely interpret the data, without stalling the handshake circuit (keeping it from making progress).

4.4 Iteration

A second component that interfaces between control and data is the do-component, which implements the control structure of Tangram's "`do G then S od`." The compiled handshake circuit for this Tangram statement is depicted in Fig. 4.11. Channel a is the initiation channel of the iteration and originates from some control component. Via channel b the guard (`G`) is evaluated, and channel c is the initiation channel of command `S`.

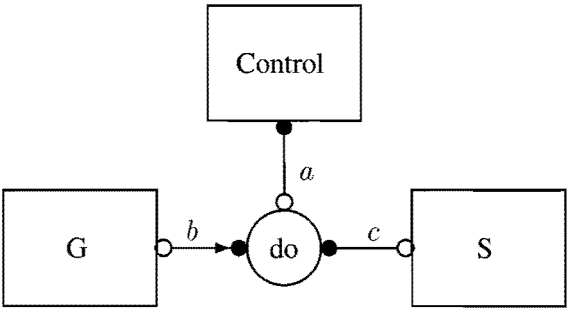


Figure 4.11: Handshake circuit that corresponds to Tangram's iteration statement `do G then S od`.

The box labeled 'G' implements the guard `G` of the statement and may contain both pull and passive components. The pull components then correspond to the operators that occur in expression `G`; the passive components correspond to the oper-

ands. Tangram does not allow for communication (input actions) in guards, which implies that the passive components are limited to variables and constants.

The handshake circuit that corresponds to command S of the construct is represented by the box labeled 'S.' It depends on the structure of S whether channel c connects to an interface component (for instance, if S is an assignment) or to a control component.

When activated along a, the do-component first evaluates the guard during a handshake along b, and then, depending on the outcome of that evaluation, either terminates by completing the handshake along a (if the evaluation resulted in false), or handshakes along c to execute the statement, and then reevaluates the guard. This cycle of handshakes along b and c continues until the guard is false.

The choice of a data-valid scheme for the data channels has impact both on the implementation of the pull components and on the implementation of the do-component itself. One important observation that has to be made is that during execution of the statement (that is, during the handshake on channel c) some of the variables that occur in the guard will typically change. So, as soon as the handshake on channel c is initiated, some of the variables in the passive-components block may change.

From the discussion on the implementation of the assignment we already know that in order to keep the implementation of the variables simple, that is, in order not to have to latch the data in the readports of the variable, we should assume a reduced data-valid scheme on the pull channels. The data on the pull channels may then change as soon as one of the variables involved changes. For this particular context this implies that the data on channel b should no longer be assumed to be valid if the handshake on channel c is initiated. This naturally has an impact on the implementation of the do-component, which is discussed in Section 5.2.3.

In a four-phase implementation, the true four-phase protocol on the pull channels gives rise to an implementation of the pull components (operators in the guard) that best exploits the four phases of the handshake protocol. Furthermore, choosing the same data-valid scheme on the data channels as for assignments allows sharing of datapaths between guards and expressions that occur in assignments, see Section 4.6.3.

4.5 Selection

Another interface between control and data originates from Tangram's `case` statement, which is the equivalent of the switch statement in the programming language C and the case statement in Pascal. A case statement selects, depending on the value of an expression, the corresponding alternative from an enumerated list. An ex-

ample of a case statement is the following.

```
case (x+y)
  is  0  then S0
  or  1  then S1
  or  4  then S4
  or  7  then S7
  si
```

If `x+y` evaluates to `0`, then `S0` is executed. Similar for the other alternatives that are listed. If the expression evaluates to a value that is not listed, the behavior of this statement is not specified. We may choose this situation equivalent to `stop` or `skip`, for instance. (In the Tangram compiler `skip` is chosen.)

The handshake circuit that corresponds to a case statement is shown in Fig. 4.12. Handshake channel a is the initiation channel of the statement. It connects to a transferrer, which (when activated) evaluates the expression via channel b, and sends the result of this along channel c. Box 'G' represents the handshake circuit for the expression and consists of pull components (for the operators) and passive components (for the operands). The case components form a sort of decoder that, dependent on the data that is input along channel c, activates one of the nonput channels connecting to the alternative statements.

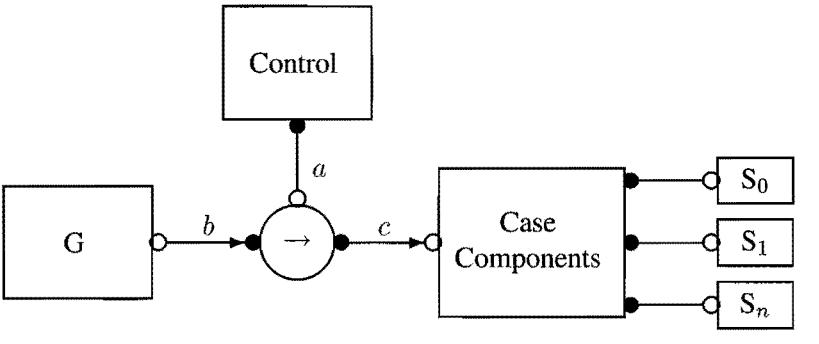


Figure 4.12: Structure of datapath around case components.

One thing that should be taken into account in the implementation of the handshake components that are involved in a case statement is that the statement that is selected may change variables that occur in the guard. If the variables are implemented with simple read ports, as in the true four-phase protocol, this implies that the data on the pull channels (and thus on b), may start to change as soon as a handshake on one of the nonput channels from the case components block is activated. This, of course, affects the implementation of the transferrer and the case

components.

The simplest situation is that in which the guard does not contain operators, but consists of a variable only, and, furthermore, that variable is not written by any of the alternatives. In that case no delay-matching has to be done and the guard remains stable during the complete handshakes on b and c. We can thus implement the transferrer with wires only (any of the two will do), and the data on c is stable during the complete four-phase handshake, which allows for a straightforward combinational implementation of the case components (see Section 5.2.2).

Another situation in which the case components can be implemented combinationally is if the guard is an expression that contains operators, but none of the alternatives modifies any of the variables that occur in the guard. In order to fully exploit the delay-matching potential of the operators, we should first complete the four-phase handshake before the handshake on c is initiated. This requires a transferrer with gates in the control, see Section 5.2.1.

The most complex situation occurs if some of the alternatives *do* affect variables that occur in the guard. In that case the guard is not stable, and as soon as a handshake is initiated the guard may change. There are two ways to handle this situation. One possible solution is to make to case components insensitive for such changes, at the cost of one asymmetric C-element per alternative, the other is to latch the guard before initiating the handshake on c. The first solution is discussed in Section 5.2.2, the second one can be implemented at the Tangram level.

In order to assure stable guards during the execution of an alternative in the case statement, the compiler from Tangram to handshake circuits could insert an auxiliary variable of the same type as the guard, and transform

```
case G is ... si
```

into

```
g:=G; case g is ... si.
```

This new case statement then has a simple and stable guard. Therefore, the corresponding transferrer can be implemented with wires only, and the case components can be implemented combinationally.

It depends on the type of the guard, more precisely, on the number of bits that are used to represent the guard, which alternative is the most economic. Making the case components insensitive to changes in the guard requires (on average) three transistors per alternative. Latching the guard requires ten transistors per bit, plus some small overhead (6), cf. Section 5.3.2. A guard of N bits can serve 2^N alternatives, so for the transistor count we have to weigh $10N + 6$ against $3 \cdot 2^N$. Given these numbers, four bits is nearly the break-even point, and a guard of at least five

bits should be latched. In a standard-cell technology it also makes sense to minimize cell area. If we use grids as a unit we might compare $6N+5$ and $3 \cdot 2^{N-1}$, which also indicates that from five bits onwards latching is more area-efficient.

4.6 Sharing

In the discussions so far we have ignored possible interferences between choices for data-valid schemes in various contexts. In this section some sources of interference are addressed.

4.6.1 Sequential sharing

One point at which there is only limited freedom in choosing data-valid schemes is the data-transfer. Datapaths in Tangram can, for instance, be shared, which means that hardware resources are reused. An example of this is the sequential composition of Tangram statements `z:=x+y` and `b!(x+y)`. Tangram provides means to share the hardware for expression `x+y`, so that only one adder is required. An example of such a Tangram program is the following, in which function `add` is *declared* (denoted by ':' in `add : func()`).

```
T   = type [0..7]
& TT = type [0..14]
|
(b!TT).
begin
  x,y : var T
& z   : var TT
& add : func(): TT . (x+y)
|
  z:=add() ; b!add()
end
```

The compiled handshake circuit for the above Tangram program is shown in Fig. 4.13. Only one adder component is used, and multiple (sequential) accesses to this are handled by the demultiplexer (dmx).

In the implementation of the adder a data-valid scheme has to be chosen for all channels. Since this adder may be part of an expression in an assignment and in an output communication it is clear that the choice for a particular data-valid scheme has impact on the implementation of both types of data-transfers.

The shared part of the handshake circuit consists of pull channels only. Fortunately, both for output and assignment we have chosen a reduced late data-valid

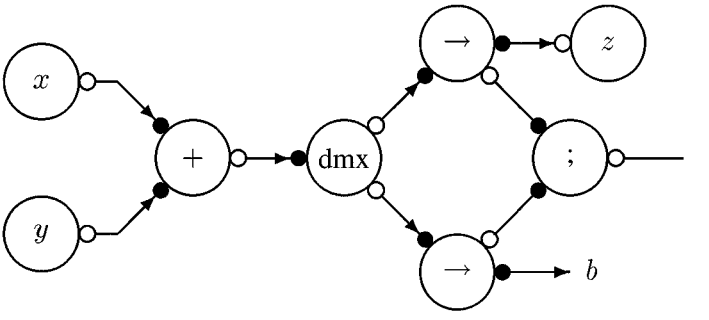


Figure 4.13: Compiled handshake circuit for sequential assignment and communication with sharing.

scheme for these channels, since this allowed the use of symmetric delay-matching, with the matching spread over the up and the down phase of the handshake.

In the example above the influence is limited to pull channels only, since the transferrers are still separate and may therefore be implemented differently. The influence may go further, as shown in the next section.

4.6.2 Parallel sharing

Sharing in Tangram may have an effect on both push and pull channels in the handshake circuit, since it may in some cases be interesting to share more than just an expression. An example where sharing clearly pays off is the parallel composition of Tangram statements `z:=x+y` and `b!(x+y)`. (One could replace the sequential composition of these statements in the Tangram program in the previous section by parallel composition. Unfortunately, this is not allowed in the current version of Tangram.) These statements both 'transfer' `x+y`, albeit to different destinations, namely to variable `z` and to channel `b`. Both from an energy and an area viewpoint it would pay off to share the hardware for both the expression and the transferrer, between the two data-transfers. This would imply, however, that the transfers are not really in parallel anymore, but are more-or-less synchronized, in the sense that they cannot complete the handshakes independently. The handshake circuit with the best area and energy statistics, in which both the expression and the transferrer are shared, is depicted in Fig. 4.14.

In this handshake circuit the assignment and the output are intertwined. For the transferrer and the adder a choice has to be made for an appropriate data-valid

[2]With the current specification and compilation of Tangram this handshake circuit cannot be generated.

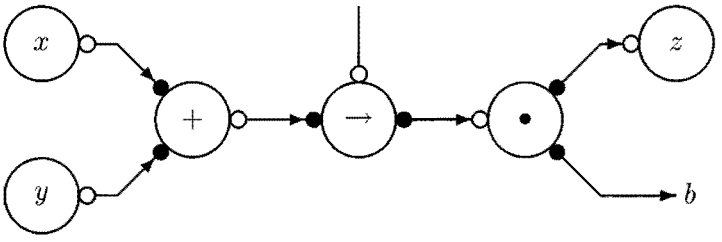


Figure 4.14: Handshake circuit for synchronized assignment and communication.[2]

scheme. In the fork component, which distributes the value to channel b and variable z, one could still choose to differentiate between the two output channels, and choose two different data-valid schemes. This would, however, complicate the implementation of the fork. With the choices that have been made in this chapter, the data-valid scheme at channel b will be late.

4.6.3 Sharing between do/trf

Declared functions can be shared between expressions in various contexts in Tangram. Above we addressed the sharing between various data-transfers, but we may also share a declared function between, say, the expression in an assignment, and the guard of an iteration, such as in the following Tangram fragment.

```
T  = type [-6..7]
& TT = type [-13..14]
|
().
begin
  x,y : var T
& z   : var TT
& add : func(): TT . (x+y)
& sub : func(): TT . (x-y)
|
  z:=add()
; ...
; do add()>sub() then ... od
end
```

From this example it should be obvious that we had best choose *one* data-valid scheme for *all* pull channels, and thus not distinguish between expressions that occur in assignments and in guards. For all components that occur in expressions we

choose the true four-phase scheme, which was already identified as the best choice in the context of assignments, earlier in this chapter.

4.7 Conclusion

In this chapter we have investigated the implementation of Tangram handshake circuits. More precisely, we have tried to find an 'optimal' assignment of data-valid schemes to handshake channels. The main target was to combine data-valid schemes such that all phases of the four-phase handshakes are productive.

For pull channels this leads to the choice for the reduced late data-valid scheme. This allows for low-cost implementation of both variables and operators. Delay-matching can be implemented with symmetric delays, since both the up-going and the down-going phase of the handshakes on the pull channels are scheduled before the data-valid signal. Although the above choice for pull channels was motivated by the implementation of Tangram assignments, sharing of datapaths basically dictates the same data-valid scheme to be chosen on all pull channels.

On push channels the late data-valid scheme has been chosen. In multiplexers this allows to use the up phase for setting the switches and the down phase for matching the rippling of the data. In the implementation of handshake variables, the up phase is used to open the latches, which are then closed again in the down phase. The late data-valid scheme thus allows for simple multiplexer and latch control circuits (details in Chapter 5).

The choice for data-valid schemes in which all phases are productive ('true' four-phase) has lead to a restriction of the implementation freedom for the control part of handshake circuits. The broad four-phase handshake protocol has to be chosen, since this is required to guarantee the mutual exclusion between read and write actions on handshake variables on which the choices for push and pull data-valid schemes have been based. Another reason to choose broad control is that it keeps the implementation of multiplexers simple, since for these components it assures mutual exclusion between accesses via different inputs.

This chapter intentionally does not cover the complete compilation scheme from Tangram to handshake circuits. The implementation of register files, RAM and ROM interfaces, and parameterized functions and procedures, for example, has not been addressed. These additional features, however, all fit naturally in the schemes as they have been discussed in this chapter.

In the next chapter we discuss the implementation of handshake components with emphasis on the variants with the data-valid schemes as they are required for the true four-phase implementation of handshake circuits.

Chapter 5

Handshake Components

Single-rail implementations of handshake components are discussed in this chapter. The partitioning that is introduced in the previous chapter is followed, that is, we distinguish interface components (transferrer, do), passive components (constant, passivator, variable), pull components (adder, negator, comparator), and push components (multiplexer).

The implementations of the various handshake components that are addressed in this chapter follow the same line of thought. The components are decomposed into a data and a handshake part. The data part implements the operations on the data, whereas the handshake part organizes the request and acknowledge signaling between a component and its environment. The data and control part may be connected via enable, control or select signals, but they can also be independent. Examples of both situations are addressed in this chapter.

In the implementation of handshake components several criteria can be taken into account, such as area, speed, energy, and testability. We choose minimal area as the most important design criterion, since the reduction of the area overhead of handshake circuits is the main motivation throughout the thesis.

In the implementation of some of the components we take into account that they are part of a Tangram handshake circuit. If such assumptions are made, this is explicitly stated in the text.

5.1 Implementation aspects

In this section we briefly introduce production rules as a means to abstract from the target (CMOS) technology. Furthermore, several timing assumptions are identified, which lead to a choice of specific timing assumptions in the implementation of handshake components.

5.1.1 Production rules

The target technology for handshake circuits is that of digital CMOS VLSI circuits. Such a circuit consists of CMOS gates which are interconnected by wires. A CMOS gate is an operator with inputs and, generally, one output. For the specification of CMOS gates we use *production rules*, as introduced by Martin. For an extensive treatment we refer to Martin [57]. A brief introduction is given in the rest of this section.

Production rules describe the dynamic behavior of CMOS gates. This is established by giving a pair of predicates U and D that specify when the output of the gate makes a transition. The behavior of an operator with single output z can be specified by the following pair of production rules

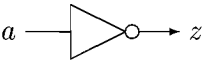
$$\begin{cases} U & \mapsto & z\uparrow \\ D & \mapsto & z\downarrow \end{cases}$$

Predicates U and D are Boolean expressions that range over the inputs of the operator. They are generally referred to as the *guards* of the operator. Production rule $U \mapsto z\uparrow$ can be interpreted as 'when U holds output z becomes true (goes high).' Similarly, $D \mapsto z\downarrow$ can be read as 'when D holds output z goes low.' These two rules, in combination with the rule that the output is stable otherwise, specify the behavior of the operator.

In Boolean expressions as they used in this thesis, '$*$' denotes Boolean AND and '$+$' stands for Boolean OR. The negation of a literal x (expression without operators) is denoted by $\overline{x}$; that of a compound expression X by $-X$.

One of the simplest examples to which production rules can be applied is an inverter, whose production rules and symbol are given next (input a, output z).

$$\begin{cases} \overline{a} & \mapsto & z\uparrow \\ a & \mapsto & z\downarrow \end{cases}$$



A restriction on the guards is that they must be mutually exclusive, that is, they may never be true simultaneously (or, alternatively, $-(U * D)$ should be invariantly true). A further restriction is that guards are required to be *stable*, which means that an output transition should not directly invalidate its own guard. This essentially is a restriction on the environment of the operator. An output change must be allowed to complete before the guard that causes it becomes false.

Changes on the inputs of an operator may or may not cause a change in the gate output. If an input transition causes the output to make a transition, it is called *productive*, otherwise it is called *void*.

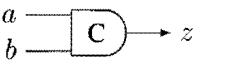
Two kinds of gates can be distinguished, namely, *combinational* and *sequential* gates. A gate is called combinational if the disjunction of the predicates in the production rules specifies the complete (input) state space, that is, if $U + D$ always holds. In that case we have $D = -U$. Combinational gates can also be specified using a Boolean equation of the form $z = U$.

A two-input OR-gate with inputs a and b and output z, is specified by $z = a + b$, and by the following pair of production rules (the symbol is also shown).

$$\begin{cases} a + b & \mapsto & z\uparrow \\ \overline{a} * \overline{b} & \mapsto & z\downarrow \end{cases}$$

A well-known sequential gate in the context of asynchronous design is the Muller C-element.[1] In its basic form, this gate has two inputs and one output. The output assumes the value of the inputs if these values are the same. The production rules and symbol of this *symmetric* C-element are as follows.

$$\begin{cases} a * b & \mapsto & z\uparrow \\ \overline{a} * \overline{b} & \mapsto & z\downarrow \end{cases}$$

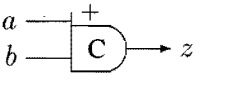


One may observe that the C-element behaves like an AND-gate when the output is low, and like an OR-gate when the output is high. The production rules allow for multiple void transitions. For example, when output z and input b are both low, input a may make multiple transitions, without affecting the state of the operator. In this state, the output will only change if a and b are high simultaneously.

Asymmetric C-elements are used frequently throughout this chapter. In an asymmetric C-element the guards of the production rules are not equivalent modulo negating of the inputs, as is the case for the symmetric C-element. In the symbol that we use for a C-element we indicate which input only contributes to the up guard, by labeling that input terminal with a '+.' Similarly, we use a '-' label if an input only contributes to the down guard. This notation has been adopted from Furber.

The most common asymmetric C-element is the one specified below. For this element, input b is the dominating input. By making input b low, we can force the output of the C-element low.

$$\begin{cases} a * b & \mapsto & z\uparrow \\ \overline{b} & \mapsto & z\downarrow \end{cases}$$



[1] named after D. E. Muller [61].

In addition to labeling inputs with '+' and '-', to denote in which guard they occur, we may also use bubbles to denote inverted inputs.

Sequential gates can also be specified in a single equation, specifying the output state as a fixed point. For a gate with output z, up guard U, and down guard D, equation $z = U + z * \overline{D}$ completely characterizes the behavior of the gate.

The initial state of a sequential gate is not uniquely defined. To force such a gate to a know state, should that be required, one can add a (re)set input to the gate, or control the inputs such that the state is known. Throughout this thesis we strive for self-initializing circuits, in which no extra (re)set input is used. In these circuits the gates in a sense cooperate to reach a know initial state.

Production rules specify the behavior of a gate, and in this, abstract from the exact timing behavior. After a productive input transition, the time it takes for the output to react is not specified. The impact of the abstraction, and the cases where we *do* require timing information, are discussed in the next section.

5.1.2 Timing assumptions

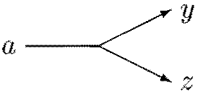
CMOS gates can be connected to form more complex structures. For these networks several timing assumptions can be made, especially with respect to nodes that fan out to multiple gates: so-called forks. In the rest of this section several timing assumptions are addressed. These timing assumptions differ in the amount of implementation knowledge that is used. More timing assumptions generally lead to circuits that are more efficient but that operate within a more limited window of operating conditions.

Fork

An extreme approach that can be followed in asynchronous circuit design is to make *no* timing assumptions, apart from those made in the operators, which is that productive input transitions eventually lead to output transitions, within some unknown but non-negative delay. This class of circuits is generally called *delay insensitive*.

In its purest form, even branches in wires are treated as separate components, known as *forks*. A fork is a component with one input and two outputs, in which the outputs follow the input, albeit with an unspecified delay that is possibly different for the two branches. The associated symbol and the specification with production rules are given below.

$$\begin{cases} a & \mapsto & y\uparrow, z\uparrow \\ \overline{a} & \mapsto & y\downarrow, z\downarrow \end{cases}$$



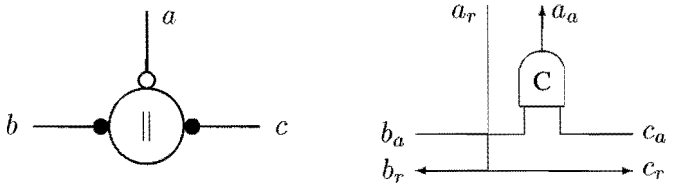


Figure 5.1: Two-phase, delay-insensitive implementation (right) of parallel component (left).

In a delay-insensitive circuit, transitions at both ends of the fork have to be detected before the input of the fork may be changed. This leads to circuits in which for every fork a corresponding C-element can be pinpointed that essentially combines the two paths again.

An example of the combined use of the fork and the C-element is the two-phase implementation of the parallel handshake component (for specification and symbol see Chapter 2). The behavior of a two-phase parallel component with passive port a and active ports b and c is described by command

$$*(a_r\uparrow ; (b_r\uparrow ; b_a\uparrow \parallel c_r\uparrow ; c_a\uparrow) ; a_a\uparrow ; a_r\downarrow ; (b_r\downarrow ; b_a\downarrow \parallel c_r\downarrow ; c_a\downarrow) ; a_a\downarrow).$$

This component can be implemented with a fork that forwards a_r to b_r and c_r, and a C-element that combines the two acknowledges (b_a and c_a) into one acknowledge a_a. The circuit diagram for this implementation is shown in Fig. 5.1. (The derivation of implementations in terms of production rules from such commands is discussed by Martin in [57].)

Isochronic fork

The isochronic fork is introduced by Burns and Martin [20, 57] and is considered to be an essential and the 'weakest possible' compromise to true delay insensitivity [56]. An isochronic fork is a fork for which we assume that the difference in delays between the two branches is less than the delays through the gates to which the fork is an input. Circuits in which the only timing assumption is that of the isochronic fork are generally called *quasi delay insensitive* (QDI) [19]. The term *speed independent* basically refers to the same class of circuits.

Implementation aspects of isochronic forks are well understood [56, 7]. They amount to limiting the spread in logic thresholds of gates, which is straightforward as long as the maximum transistor-stack height is two or three. Furthermore, the capacitive loads and driving strengths should be balanced such that transition times are uniform and especially 'slow' transitions are ruled out.

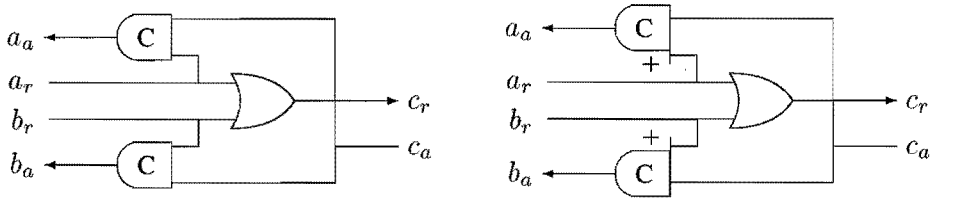
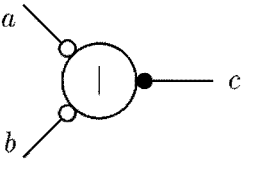


Figure 5.2: Four-phase, quasi delay-insensitive implementations of the nonput mixer with one (left) and three (right) isochronic forks.

Isochronic forks quite often combine with asymmetric C-elements, similar to the way forks combine with symmetric C-elements. An example of the application of isochronic forks is the four-phase implementation of the nonput mixer. The symbol and the command specification of the four-phase mixer with passive ports a and b and active port c are as follows:

$$
\begin{array}{l}
*(\ a_r\uparrow\,;c_r\uparrow\,;c_a\uparrow\,;a_a\uparrow\,;a_r\downarrow\,;c_r\downarrow\,;c_a\downarrow\,;a_a\downarrow \\
\quad \mid b_r\uparrow\,;c_r\uparrow\,;c_a\uparrow\,;b_a\uparrow\,;b_r\downarrow\,;c_r\downarrow\,;c_a\downarrow\,;b_a\downarrow \\
\ )
\end{array}
$$



The nonput mixer requires the handshakes on a and b to be mutually exclusive. For the four-phase command given above, this implies that the environment guarantees that a_r and b_r will never be high simultaneously.

From the command given above the gate-level implementation is readily derived. For each output the state that causes this output can be uniquely characterized using only the inputs of the mixer. This leads to the following three pairs of production rules.

$$
\left\{ \begin{array}{rcl} a_r + b_r & \mapsto & c_r\uparrow \\ -(a_r + b_r) & \mapsto & c_r\downarrow \end{array} \right.
\qquad
\left\{ \begin{array}{rcl} a_r * c_a & \mapsto & a_a\uparrow \\ \overline{a_r} * \overline{c_a} & \mapsto & a_a\downarrow \end{array} \right.
\qquad
\left\{ \begin{array}{rcl} b_r * c_a & \mapsto & b_a\uparrow \\ \overline{b_r} * \overline{c_a} & \mapsto & b_a\downarrow \end{array} \right.
$$

These production rules specify an OR-gate and two symmetric C-elements. The corresponding circuit is shown in Fig. 5.2 (left). The fork connecting c_a to the C-elements is isochronic. Both for the up and the down-going transition, only the transition on one branch of the fork is acknowledged. One may observe that the OR-gate basically is the reason that the isochronic fork is necessary.

The efficiency of the mixer implementation (in terms of area, time, and energy) can be improved by replacing the symmetric C-elements by asymmetric C-elements,

at the cost of two additional isochronic forks. To this end the production rules for a_a and b_a should be weakened to:

$$\left\{\begin{array}{rcl} a_r * c_a & \mapsto & a_a\uparrow \\ \overline{c_a} & \mapsto & a_a\downarrow \end{array}\right. \qquad \left\{\begin{array}{rcl} b_r * c_a & \mapsto & b_a\uparrow \\ \overline{c_a} & \mapsto & b_a\downarrow \end{array}\right.$$

The circuit that corresponds to this implementation is depicted in Fig. 5.2 (right). The forks connected to a_r and b_r are isochronic. The up transitions are acknowledged for both branches, whereas the down transition of the branch to the asymmetric C-element is ignored. The isochronic-fork assumption is that after a transition on c_r caused by any of the incoming requests, the asymmetric C-element that connects to the other branch of the fork has also observed the corresponding input transition.

The mixer implementation illustrates that circuit realizations can be made more efficient by making more timing assumptions. Especially the application of isochronic forks to replace symmetric C-elements by asymmetric variants is an interesting one that is often applied in the implementation of handshake components.

Extended isochronic fork

For an isochronic fork we assume that, once a transition on one branch has been observed, a node connected to the other branch has also detected this transition. With *extended isochronic forks* [15] we even go even a step further. For such a fork we also assume that transitions on different branches have completed (and have been observed by gates connected to these nodes) once one transition is observed, but now for the nodes that are one (inverting) CMOS stage further away from the fork.

These forks are called (extended) isochronic forks of depth one, and this notion can readily be generalized to any depth (counting inverting CMOS stages as a unit). The traditional isochronic fork then has depth zero. Isochronic forks of maximum depth two are used throughout this chapter.

An example of the use of an extended isochronic fork of depth one is the implementation of a 3-input C-element. With inputs a, b, and c, and output z, this C-element can be characterized by $z = a * b * c + z * (a + b + c)$. This specification can be decomposed into three equations, thus introducing two internal nodes, namely as $z = -(x * y)$, where $x = -(a * b * c)$ and $y = -(z * (a + b + c))$. This implementation is shown in Fig. 5.3.

For this realization of the 3-input C-element, the extended isochronic fork assumption guarantees that after a transition on node z' (which represents the output of any inverting CMOS gate connected to output z of the C-element) internal node y is stable. Therefore, after a change on z', the inputs of the C-element may safely change, without the risk of a hazard.

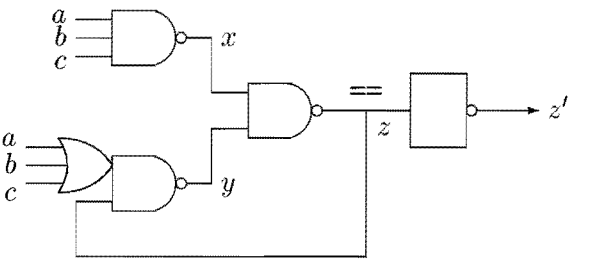


Figure 5.3: Implementation of three-input C-element based on OR-AND-INVert and an extended isochronic fork. The gate with output z' is included to show the reach of the extended fork.

One may observe that, for proper operation of the 3-input C-element, the extended fork assumption can be asymmetric. Node y should be stable before a gate connected to z' observes a productive transition on z' (and reacts). All extended forks that are used in this chapter have such an asymmetry.

Delay matching

With the extended isochronic fork we can stepwise enlarge the reach of an isochronic fork. The timing assumption of these forks, however, becomes increasingly tricky. For two long chains of inverting CMOS stages we cannot guarantee that the difference between the delays through these chains is bounded to, say, a (unit) gate delay.

In the implementation of extended forks we generally have to assure an asymmetry, that is, we have to guarantee that one path is not slower than another path. An extreme in this is *delay matching*, in which we have to assure that the delay of a path is larger than the worst case delay of several other paths.

Delay matching is an essential technique in the implementation of single-rail circuits. The implementation of this generally requires a safety margin, to account for variations in operating conditions, spread in capacitive loads, and variations in driving strengths. This is discussed in more detail in Section 5.5.2 and Chapter 6.

5.1.3 Assumptions in handshake circuits

In the previous section we have identified several timing assumptions that can be made in the design of handshake components and handshake circuits. In this section we choose which assumptions are applied in which part of the design of handshake components.

The implementation of control handshake components falls outside the scope of this chapter. These components are implemented such that they are QDI. The mixer and parallel implementations that are shown earlier in this section are examples of such QDI realizations.

The request-acknowledge part of data and interface components are mostly implemented QDI. Sometimes extended isochronic forks (depth one or two) are used. This is then indicated explicitly. Delay matching is used if timing assumptions have to be made about the logic in the datapath. In these assumptions, the extended isochronic fork occasionally suffices.

For all handshake components we strive for implementations that are *self initializable* [8]. In combination with the property that the activity graph of a Tangram handshake circuit is acyclic, this can be exploited to force a compiled circuit in a well-defined initial state after power-up of the IC, *without* the need of additional reset circuitry, by making only the electrical inputs to the circuit low. (The activity graph is the directed graph obtained from a handshake circuit by replacing the components by nodes and introducing a directed arc from one node to the other if there is a corresponding handshake channel for which the component of the first node is active, and that of the second node is passive.)

To make a circuit realization self-initializable we have to obey two rules. Firstly, when the (request) inputs of the passive ports are low, the (request) outputs of all active ports should go low. Secondly, when all inputs (active acknowledges and passive requests) are low, all outputs should go low. The implementation of the parallel and mixer components shown earlier obey these rules.

5.2 Interface components

In the components that interface between control and data we encounter both data and control handshake channels. In a four-phase implementation this leaves us with three alternatives for the four-phase nonput handshake protocol that is followed, namely, early, broad, and late. Throughout this chapter we assume the broad four-phase protocol for all nonput channels. This allows us to focus on the single-rail data issues, thereby reducing the scope of this chapter.

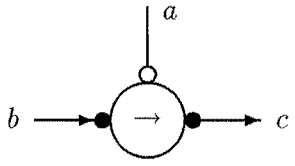
The interface components that are covered in this section are the transferrer, case components, and the do component.

5.2.1 Transferrer

The transferrer (which is discussed earlier in Chapter 2) has one nonput port, one pull port, and one push port. In the compilation scheme of Tangram the transferrer

is used to control the transfer of data in input, output, and assignment. (The transferrer is also used in the implementation of case-selection, as is discussed later in this chapter.) The specification and symbol of the transferrer are as follows:

$$\text{TRF}(a^\circ, b^\bullet?T, c^\bullet!T) = \\ |[\, x : T \mid *\,(a_r\,;b_r\,;b_a(x)\,;c_r(x)\,;c_a\,;a_a)\,]|$$



In the command, x is a local variable of the appropriate type (the type of the input and output channel), $b_a(x)$ denotes the receipt of a value x via channel b, and $c_r(x)$ the sending of the same value encoded in the request of channel c. The variable should not be interpreted as a piece of hardware, but rather as an auxiliary (or ghost) variable that is only used in the specification.

Global organization

Since the transferrer is used frequently in handshake circuits, it is essential to implement it efficiently. The transferrer *controls* the transfer of data from channel b to channel c. It should therefore be possible to implement it with wires only as far as the data is concerned. For the control some hardware may be required. We therefore start with the decomposition of the transferrer as shown in Fig. 5.4.

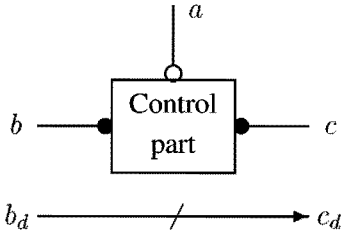


Figure 5.4: Decomposition of transferrer in control and data circuitry.

From the decomposition we can derive requirements for the control circuit. Since the data is not latched we have to make sure that the data-valid period on channel b encloses that on channel c. This means that the data-valid signal on b should precede that on c, and the data-release signal on c should precede that on b.

In all control circuits that we discuss we shall adhere to the broad four-phase protocol for the control on channel a, that is, for all implementations, the first transition during a data-transfer action will be $a_r\uparrow$, and the last one $a_a\downarrow$. Other handshake protocols for control can also be interesting but fall outside the scope of this chapter.

Wire-only transferrers

Two wire-only implementations of the transferrer control circuit exist, and they were both already introduced in the previous chapter. The circuits are shown in Fig. 5.5; the transition level specifications are given in Eqn. 5.1 and Eqn. 5.2.

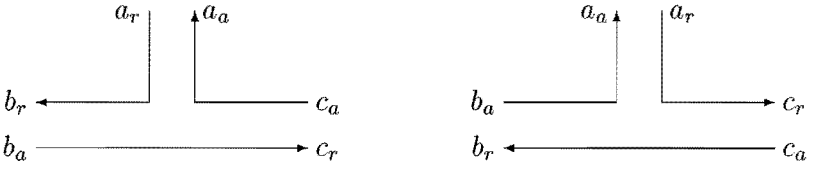


Figure 5.5: Wire-only realizations of the transferrer control circuit.

$$*(\ a_r\uparrow ; b_r\uparrow ; b_a\uparrow ; c_r\uparrow ; c_a\uparrow ; a_a\uparrow \\ ; a_r\downarrow ; b_r\downarrow ; b_a\downarrow ; c_r\downarrow ; c_a\downarrow ; a_a\downarrow \\ ) \tag{5.1}$$

$$*(\ a_r\uparrow ; c_r\uparrow ; c_a\uparrow ; b_r\uparrow ; b_a\uparrow ; a_a\uparrow \\ ; a_r\downarrow ; c_r\downarrow ; c_a\downarrow ; b_r\downarrow ; b_a\downarrow ; a_a\downarrow \\ ) \tag{5.2}$$

The transferrer in Fig. 5.5 (left), which corresponds to Eqn. 5.1, is also the implementation of the two-phase transferrer. In that case it naturally combines the data-valid schemes on b and c in the appropriate way.

In case the data-valid scheme on channel b is early the control circuit assures a (prolonged) early data-valid scheme on channel c. Similarly, broad combines with prolonged broad, and late with prolonged late.

The mirrored transferrer of Fig. 5.5 can only be used in combination with the late four-phase data-valid scheme on channel c. In that case the data-valid signal on channel b must be $b_a\uparrow$, which implies that either the early or the broad data-valid scheme on b should be assumed.

Parallel transferrers

The wire-only transferrers are all fully sequential. At the expense of some circuitry, parallelism might be introduced, for example to reduce the time spent in redundant phases of the handshakes. Two examples of such transferrers are given below.

The serial-parallel transferrer sequences the up-phases of the handshakes on b and c, but allows for parallelism during the return-to-zero phases on these channels. Its specification is given in Eqn. 5.3. This transferrer can be used in combination

with the early data-valid schemes on b and c, and in that case runs the cooling-down phases of the handshakes on these channels in parallel.

The control circuit for this transferrer consists of an AND-gate and an asymmetric C-element, see Fig. 5.6 (left). Since these gates add delay to the critical paths, it is not obvious beforehand that such an implementation results in faster overall operation than the wire-only implementation.

$$
\begin{array}{l}
*(\; a_r\uparrow \; ; \; b_r\uparrow \; ; \; b_a\uparrow \; ; \; c_r\uparrow \; ; \; c_a\uparrow \; ; \; a_a\uparrow \\
\quad ; \; a_r\downarrow \; ; \; (b_r\downarrow \; ; \; b_a\downarrow \; \| \; c_r\downarrow \; ; \; c_a\downarrow) \; ; \; a_a\downarrow \\
\quad)
\end{array}
\tag{5.3}
$$

The parallel-serial transferrer runs the up-going phases of b and c in parallel but sequences the down-going phases of the protocols, as specified in Eqn. 5.4. This transferrer can be used with late data-valid schemes on b and c. It then runs the warming-up phases of the handshakes in parallel and properly sequences the data-valid periods. An implementation of the control circuit is shown in Fig. 5.6 (right). When compared to the wire-only transferrer of Fig. 5.5 (left), it is again not obvious whether the cost of the additional gate-delays outweighs the advantage of the extra parallelism.

$$
\begin{array}{l}
*(\; a_r\uparrow \; ; \; (b_r\uparrow \; ; \; b_a\uparrow \; \| \; c_r\uparrow \; ; \; c_a\uparrow) \; ; \; a_a\uparrow \\
\quad ; \; a_r\downarrow \; ; \; b_r\downarrow \; ; \; b_a\downarrow \; ; \; c_r\downarrow \; ; \; c_a\downarrow \; ; \; a_a\downarrow \\
\quad)
\end{array}
\tag{5.4}
$$

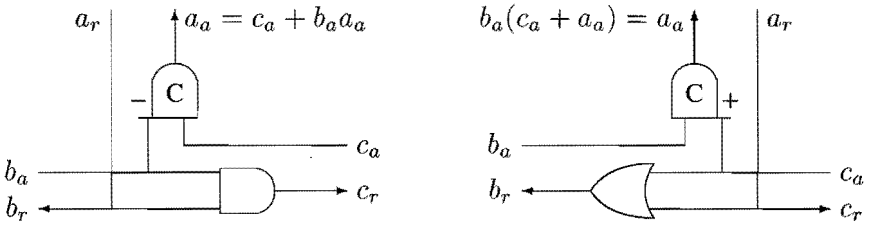


Figure 5.6: Control circuits for ser-par and par-ser transferrer.

Sequencer transferrer

In the implementation of case selection a transferrer is needed that performs the complete four-phase handshake on channel b before the handshake on channel c is initiated. This is required to exploit the delay-matching properties of the pull data-path fully before the data that is fetched is interpreted on the push side. The control

circuit of this transferrer is exactly the control sequencer, and its behavior is specified by Eqn. 5.5.

$$*(\ a_r\uparrow\ ;\ b_r\uparrow\ ;\ b_a\uparrow\ ;\ b_r\downarrow\ ;\ b_a\downarrow\ ;\ c_r\uparrow\ ;\ c_a\uparrow\ ;\ a_a\uparrow \\ ;\ a_r\downarrow\ ;\ c_r\downarrow\ ;\ c_a\downarrow\ ;\ a_a\downarrow \\ )\tag{5.5}$$

The implementation of this transferrer is based on the so-called S-element [8, p. 162], also known as the Q-element [57, p. 36]. The S-element is a handshake component with two nonput handshake ports, one passive and one active. During the up phase of the handshake on the passive port a complete handshake on the active port takes place. The down phase on the passive port has no effect on the active port and basically restores the initial state of the S-element. For the S-element in Fig. 5.7, with passive port d and active port b, the behavior is described by

$$*(d_r\uparrow\ ;\ b_r\uparrow\ ;\ b_a\uparrow\ ;\ b_r\downarrow\ ;\ b_a\downarrow\ ;\ d_a\uparrow\ ;\ d_r\downarrow\ ;\ d_a\downarrow).$$

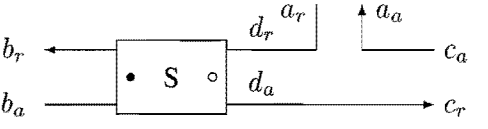


Figure 5.7: Control circuit for sequencer transferrer.

Data bundling

All transferrers described in this section are safe with respect to their bundling constraints. For the data-path they all consist of wires only. The control circuitry sometimes even adds an additional safety margin to the data-valid signal. This holds especially for transferrers that require more than wires for the control circuitry, such as in the last four implementations discussed above.

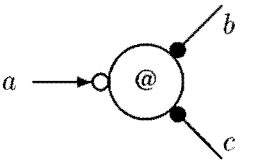
Even the wire-only transferrers may indirectly increase the safety margin of the data-valid signal. If the transferrer of Fig. 5.5 (right) combines the early data-valid scheme on b with the late scheme on c, for example, then the time from $a_a\uparrow$ till $a_r\downarrow$ (at least one inversion) is added to the safety-margin of the data-valid signal.

5.2.2 Case components

Case components are used in the implementation of Tangram's `case` construct to decode the expression to select the appropriate statement to be executed. The basic

case component has two nonput ports and one push data-port, along which it can be activated.

CASE(a°?bool, $b^\bullet$, $c^\bullet$) =
  ||[x : bool
  | * ($a_r(x)$;
      if x then b_r ; b_a else c_r ; c_a fi ;
      a_a)
  ]||



For the implementation of the case component the data-valid scheme on the data channel is critical. If the data-valid scheme on a is broad, then valid data may be assumed during the complete handshake on a. This implies that the handshakes on b and c apparently do not directly influence this data, which has impact on the compilation scheme for Tangram, as discussed in Section 4.5.

With the broad data-valid scheme on a, the case can be implemented as shown in Fig. 5.8 (left). The AND-gates decode the incoming data, and depending on the outcome either initiate a handshake along b or along c.

If the data-valid scheme on a is early, measures have to be taken to assure proper handshakes on b and c. The data (a_0) should then only influence the initiation of the handshake, whereas the return-to-zero should depend on the control signals only. This can be accomplished by replacing the AND gates with asymmetric C-elements, as shown in Fig. 5.8 (right). These asymmetric C-elements act as filters for redundant transitions on a_0.

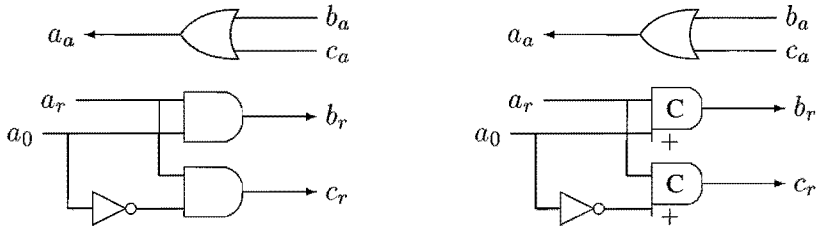


Figure 5.8: Gate realizations of case component for broad (left) and early (right) data-valid scheme on channel a.

If only a reduced early data-valid scheme can be assumed on channel a, even the filtering with asymmetric C-elements is not sufficient. If input a_0 changes from high to low while a_r and b_r are high, then the early implementation allows for a transition on c_r, which would imply that both alternatives are selected. This situation clearly is undesirable. To prevent this we can choose the implementation based on the production rules given next. With respect to the early realization, the up guards

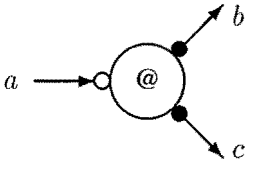
have been strengthened. This leads to an implementation based on cross-coupled asymmetric C-elements.

$$\left\{\begin{array}{rcl} a_0 * \overline{c_r} * a_r & \mapsto & b_r\uparrow \\ \overline{a_r} & \mapsto & b_r\downarrow \end{array}\right. \qquad \left\{\begin{array}{rcl} \overline{a_0} * \overline{b_r} * a_r & \mapsto & c_r\uparrow \\ \overline{a_r} & \mapsto & c_r\downarrow \end{array}\right.$$

The case component is a good example of a component for which the relation between the compilation scheme (from Tangram to handshake circuits) and the implementation of the components itself is important. In general, the handshakes on b and c may affect some of the variables that occur in the guard of the case statement. Therefore, if as a result of this the value on a may also change, then the above implementations lead to a reduced early data-valid scheme on a, and the implementation with cross-coupled asymmetric C-elements should be chosen. If the guard is latched after evaluation, however, the implementation based on AND gates is safe as well.

The basic case component can only be used to decode boolean (1-bit) guards. In a general case-construct n-bit guards have to be decoded. In Tangram this is accomplished by constructing trees of case components, for which the push case component is used. Its symbol and specification are given next. One may observe that these components basically are push components.

$$\begin{array}{l} \text{CASE}(a^\circ?\text{bool} \times T, b^\bullet!T, c^\bullet!T) = \\ \quad |[\, x : \text{bool}, y : T \\ \quad |\, *(a_r(x,y)\,;\text{if } x \text{ then } b_r(y)\,;b_a \text{ else } c_r(y)\,;c_a \text{ fi}\,;a_a) \\ \quad]| \end{array}$$

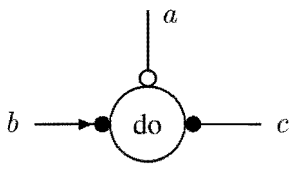


The implementation of this case component is similar to that of the basic case. The realization with AND gates can be used. Depending on the least significant bit the rest of the message is then sent either via b or via c. The forwarding of this part of the incoming data can be implemented by simply forking the data, such that on the outputs the same data-valid scheme may be assumed as on the input, but now (due to the control overhead) with an even larger safety margin.

5.2.3 Do component

The do component is used to implement Tangram's iteration ('while') construct. The command and the symbol for the do component are given next.

$$
\begin{array}{l}
\text{DO}(a^\circ, b^\bullet?\text{bool}, c^\bullet) = \\
\quad [\![x : \text{bool} \\
\quad | * (a_r ; b_r ; b_a(x) ; \\
\qquad \text{do } x \text{ then } c_r ; c_a ; b_r ; b_a(x) \text{ od } ; \\
\qquad a_a) \\
\quad]\!]
\end{array}
$$



Before we start implementing the do-component, we first take stock of some of the properties of the communication with its environment and the implications thereof on the four-phase implementation. First of all, the guard that has to be evaluated may be of arbitrary complexity, which implies that delay-matching on channel b is important. In the true four-phase scheme this means that we should perform a complete four-phase handshake on b before we interpret its data.

If the guard evaluates to true a four-phase handshake on channel c has to be performed. During this handshake some of the variables occurring in the guard will typically change. In the true four-phase scheme this implies that the guard may change as well. Consequently, the value of the guard is not stable after the handshake on c is initiated. If the guard evaluates to false, however, the handshake on a can be completed. Due to the broad four-phase control protocol this implies that all variables that occur in the guard will be stable during this handshake.

A possible gate-level realization of the do-component is depicted in Fig. 5.9. This implementation is based on the S-element and a variant of the case component. The function of the S-element is to assure valid data on channel b. At $d_a\uparrow$, b_0 is guaranteed to be stable. If b_0 is low (the guard is false) then the handshake on a is completed, which also resets the S-element. If b_0 is high (a true guard) then the handshake on c is performed. During this handshake the handshake on d is also completed, and upon completion of c a cycle is started by making d_r high. The asymmetric C-element assures the stability of c_r even if b_0 changes during the handshake.

The use of a NOR gate for a_a is a bit tricky. If we would follow the implementation of the case component, an asymmetric C-element would be used that sets at $d_a \wedge \overline{b_0}$ and resets at $\neg d_a$. We can get away with the simpler implementation, however, due to the stability of the guard when it is false during the handshake on a. The only protection that is required is against b_0 going low during the handshake on c. We could thus choose $a_a = d_a * \overline{d_0} * \overline{c_r} * \overline{c_a}$, which assures that a_a is low during a handshake on c. We can get rid of the $\overline{c_a}$ factor if we assume an isochronic fork of depth one at b_a and choose the implementation shown in Fig. 5.9. The assumption is that the path from $d_a\downarrow$ through the inverter to its output going high (one inversion) is faster than that from d_a going low to c_r going low (two inversions).

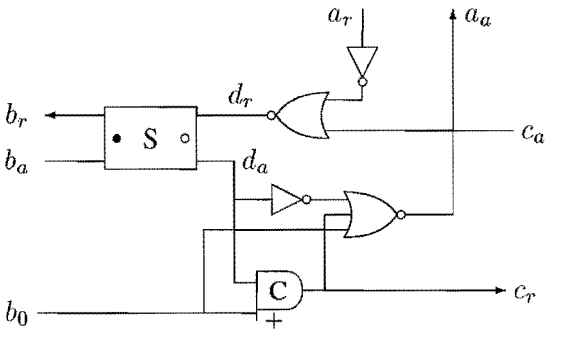


Figure 5.9: Implementation of do-component based on S-element and case component.

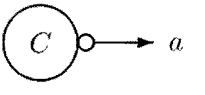
5.3 Passive components

Passive components provide and accept data upon request. The simplest passive component is the constant, which upon request always outputs the same value. The variable is one of the essential Tangram handshake components, since it is used to store and retrieve information. The passivator is used in synchronization and information exchange between parallel processes.

5.3.1 Constant

The constant has one passive pull handshake port along which it outputs its constant value (denoted by C, where $C \in T$ and where T denotes the type of the channel) upon request. The specification and the symbol of the constant are given next.

$$\text{CST}(a^{\circ}!T) = *(a_r\,;a_a(C))$$



The single-rail implementation of the constant is straightforward. Depending on the binary representation of the value C, some wires are tied low and the others tied high. The control part of the constant is implemented with wires only, since the data is always valid, and thus the acknowledge can be connected directly to the request.

All data outputs of the constant are tied high or low, which implies that the gates that these wires connect to are amenable for post optimization. This optimization is part of the single-rail design flow and is discussed in Chapter 6.

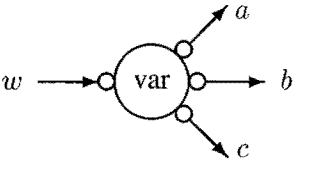
5.3.2 Variable

Tangram variables are implemented in the handshake circuit by components that we also call variables. Variables are used to store information and can be read and written. Tangram's compilation scheme assures that these read and write handshakes are mutually exclusive. It even assures a stronger property, namely, that in a data-transfer a variable is never both read and written. This property is stronger than mutual exclusion on reads and writes, since the latter would still allow for direct implementation of Tangram statements like `x:=x+1`. This statement could be compiled to a handshake circuit in which the read handshake is completed before the write handshake is initiated. In Tangram, however, an auxiliary variable is required, as in `y:=x; x:=y+1`. Variables that *do* allow for these so-called read-modify-write cycle (in which the current state of the variable is read and the new state is written within one assignment) fall outside the scope of this thesis.

An important characteristic of handshake variables is that they may have an unbounded number of independent read ports. One might say that the variable may have multiple subscribers that, independent of each other, can decide to inspect the value via a read action. A more precise specification of this is given below.

The specification and symbol of the variable are given next. As an example a variable with three read ports is used. In general the number of read ports is at least one.

$$\begin{aligned}&\text{VAR}(w^\circ?T, a^\circ!T, b^\circ!T, c^\circ!T) = \\ &\quad \text{var}(w, a, b, c)\end{aligned}$$



Read and write actions on variables are mutually exclusive, but multiple read actions may take place independently. Therefore, we choose the following specification.

$$\begin{aligned}\text{var}(w, a, b, c) = \; &|[\, x : T \\ &| * (w_r(x) ; w_a \mid *(a_r ; a_a(x)) \parallel *(b_r ; b_a(x)) \parallel *(c_r ; c_a(x))) \\ &]|\end{aligned}$$

Global organization

The variable can be decomposed into a data and a control part, as shown in Fig. 5.10 for a variable with one read port. Since read and write handshakes are mutually exclusive, we can implement the data part with latches. The control circuit has to generate the enable signals (*en*) for these latches.

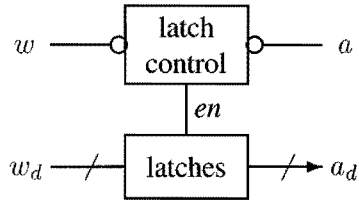


Figure 5.10: Decomposition of variable into control and data circuitry, connected by enable signals.

Since a variable may have any number of read ports, it is important to keep their implementation cost minimal. The absolute minimum is to implement the read ports with wires only. A way to achieve this is to connect the read acknowledge to the read request directly, and per bit to connect *all* read ports to the outputs of the same latch. This decomposition leads to an implementation in which the enable signals of the latches are controlled by the write port only, and one latch per bit is used, with the output broadcast to all read ports.

In the rest of this section we first discuss several implementations of latches and latch-control circuits. After that an alternative decomposition is given that is based on more expensive read ports.

Latches

An important choice that has to be made is what latch to apply for storing the data. Some degrees of freedom are (i) static versus dynamic storage, (ii) inverting versus non-inverting data, and (iii) the type of select signals. These issues are covered in detail by Weste [84, Ch. 5] and Bakoglu [3, Ch. 8].

All handshake components are implemented using static CMOS. For data and control paths that are activated with a sufficient high frequency, dynamic implementations could be used. This option, however, is not studied in this thesis. Furthermore, all variables are implemented with non-inverting latches. Inverting latches could be introduced in a post-optimization step, but this requires information about the context of the variables.

One of the simplest latches that can be chosen is depicted in Fig. 5.11. This latch is non-inverting, requires complementary enabling signals and is fully static. It contains two tri-state inverters, one connecting d to qb, the other providing the feedback by connecting q to qb. This feedback-stack is only required to make the latch static, and has to compensate leakage for current only. It can therefore be implemented with minimum-sized transistors.

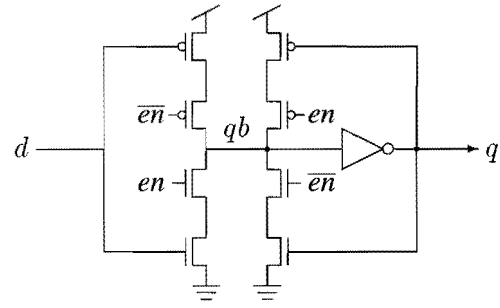


Figure 5.11: Multiplexer-like latch circuit. The feed-back stack only compensates for leakage and can be minimal sized.

The latch can be in two states, known as *transparent* (or open) and *opaque* (or closed, state-holding). With the enable signal (en) high and its complement ($\overline{en}$) low, the latch is transparent. As long as the latch is transparent, its output will follow the input. This means that if the input is stable at high or low, the output will settle to the same value.

If the enable signal is low, the latch is state-holding since the output q is in a feedback-loop to qb. Although generating the feedback directly from the output results is a simple latch, it also has some disadvantages. One consequence is that the minimum period during which the latch should be transparent depends on the output load on q. Some implications of this are sketched later in this section. A second consequence, however, is that the output may affect the internal state, for instance, if it is exposed to charge sharing effects due to a pass transistor connecting to q, or if q connects to an off-chip pin.

The symmetry can be broken by separating the feedback path from the output path, at the cost of only two additional minimal transistors, as shown in Fig. 5.12 (left). (Note that the tri-state inverters are identified by their enabling signal only, assuming the complement as disabling signal.) From this implementation two minimum transistors can be saved by replacing the tri-state inverter in the feedback path by a minimum sized inverter, as in Fig. 5.12 (right). Although this introduces a small fight during switching, it may still be advantageous because it reduces the load on the enable signals and it simplifies the layout.

The last two latches require minimal set-up and hold times and enable-pulses, because only the internal node qb has to be switched. After the latch is closed it still continues to update its output q, if the transition on q has not yet completed. An additional advantage of these latches is that they appear to have better testability properties than the simple latch [70].

The latches shown here all require complementary enable signals. It depends

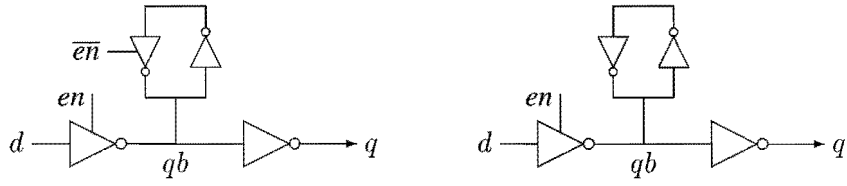


Figure 5.12: Latches with feedback decoupled from output, based on MUX cell (left) and RAM cell (right).

on the layout style whether one can best distribute both signals, or apply a local inversion of the enable signal. In a full-custom layout style it is generally better to distribute both signals, whereas in standard cells it may pay off to use a local inverter, since this reduces the number of external signals of the cell and thus the routing requirements.

An alternative that is not elaborated upon here is to use latches that require only one polarity of the enable signal. These are called true single-phase latches, and are introduced in [86] and [1]. In a synchronous environment these latches reduce clock-skew problems, precisely because only one signal has to be distributed. These latches are especially interesting when combined with logic operations, which was demonstrated in the design of the DEC Alpha [28]. Since single-phase latches require more transistors, it is not immediately clear whether they lead to really better asynchronous circuits. A case study can be found in [25].

Latch control

The latch-control circuit is the heart of the variable. It controls the enable signals of the latches such that they capture the valid data when required. An important restriction for the latch control to fulfill is that the latches should be closed during a write action *before* the data-valid period of the input data has ended. This is generally implemented by withholding the data-release transition on the write acknowledge until after the latches have closed.

Throughout this section we assume the broad data-valid scheme on the write channel, which means that the data is valid at least from $w_r\uparrow$ until $w_a\downarrow$. The implications of using other data-valid schemes are discussed later.

In the variable, the first time that it is known that the current state is no longer required is when a write action is initiated, that is, when $w_r\uparrow$ arrives. The only option, therefore, is to use a normally-opaque latch-control circuit. Since in the broad data-valid scheme the data is valid during the complete write handshake we can open the latches right after $w_r\uparrow$ and close them just before $w_a\downarrow$.

A circuit that achieves exactly this is shown in Fig. 5.13. It employs a driver to

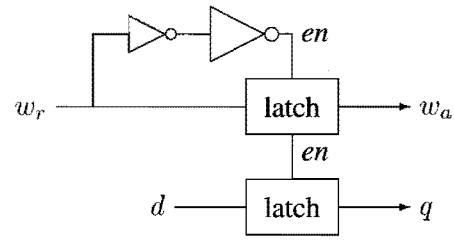


Figure 5.13: Matched-delay latch-control circuit

distribute the enable signal to all data latches and to the latch that generates the write acknowledge. This circuit can be used in combination with any type of enable-signal distribution, provided that after w_r going low there is enough time to switch the latch driving w_a before the enable signal goes low as well. With the latches of Fig. 5.12 this should be no problem. The generation of the write acknowledge signal via a latch automatically provides a matched path with the data.

The above control circuit may be a bit tricky in combination with the latch of Fig. 5.11, since it depends on the load on w_a whether $w_r\downarrow$ can really effect the state of the latch before (as a reaction to this transition) the enable signal goes low and thereby closes the latch. An extended isochronic fork of depth 2 is required to justify this assumption.

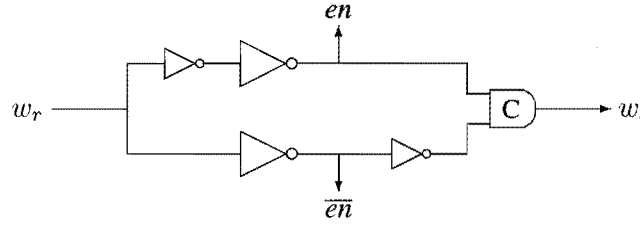


Figure 5.14: QDI latch-control circuit.

A very safe way of generating complementary enable signals and controlling these is to use the QDI latch-control circuit shown in Fig. 5.14. This is based on the two-phase control circuit described by Paver [63] with the two-to-four-phase converter (Toggle and XOR) removed. The larger sized inverters are included in the figure to indicate the insertion of buffers (drivers) and strong inverters to drive the potential high load on the enable signals (in variables consisting of many bits). The C-element in this circuit postpones the write-acknowledge signal until both enable signals (en and $\overline{en}$) have completed their transition, and have thus opened or closed all latches.

Whilst this control circuit is very safe, the C-element also introduces a delay, since it is in the critical path twice. A variant that enables faster cycle times is depicted in Fig. 5.15. Compared with the previous control circuit, the C-element is removed. Completion detection of transitions on the enable signals is done on $\overline{en}$ using an inverter. In combination with the latch of Fig. 5.11 the total transistor load on the $\overline{en}$-signal will be larger than that on the *en*-signal, since the former is connected to a minimum n-transistor and a default p-transistor, whereas the *en*-signal drives normal p-transistors and small n-transistors. (The load of a p-transistor is larger than that of an n-transistor.) This circuit thus essentially assumes an asymmetric extended isochronic fork of depth two on w_r.

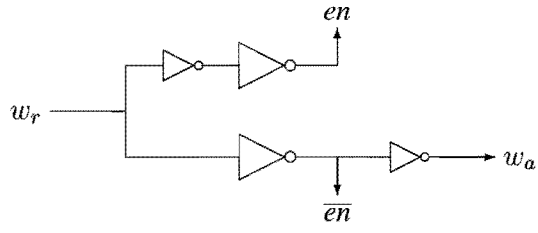


Figure 5.15: Extended QDI latch-control circuit.

Other ways of generating the write acknowledge signal are to connect it to the enable signal directly (thus saving an inverter), or to apply *fast forwarding*, that is, to generate w_a from w_r directly. This assumes that the latches are fast enough, and the environment is slow enough to still allow a sufficiently long transparent period of the latches. The time between the completion of a write handshake and the initiation of a read handshake should also suffice to complete the output transition of the latches.

If latches with local enable inverters are used —which only requires one enable signal to be distributed to the latch cells— a simplified version of the above control circuits can be used.

Data bundling

To illustrate what may go wrong when the latch of Fig. 5.11 opens for only a short period consider the events depicted in Fig. 5.16. The circuit simulated here consists of a 4-bit variable with the latch control of Fig. 5.15. The write-request and write-acknowledge are connected to an S-element to give a fast but realistic response time from $w_a \uparrow$ to $w_r \downarrow$. The enable signal is driven by a buffer, its complement by a standard inverter. The acknowledge is generated by a standard inverter from the $\overline{en}$-signal.

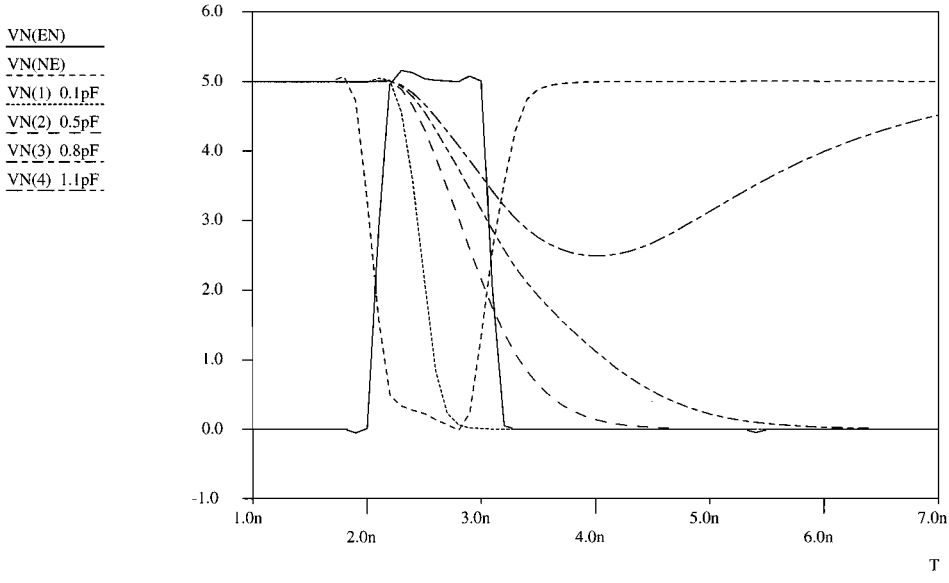


Figure 5.16: How to break the latching scheme.

The pulses generated on the enable and its complement are narrow. Together they allow for a 1 nanosecond period of transparency of the latch. It now depends on the output load of the particular latches whether this time suffices to make a transition.

Initially, the outputs of all latches are high and the inputs of all latches are low. The four latches are loaded with 0.1, 0.5, 0.8, and 1.1 picofarads. (In the technology that is assumed in the simulation, a typical load is in the range of 0.1 to 0.4 pF.) The latch with the 0.1pF load switches fast; its transition is completed before the latch closes again. The second latch has an 0.5pF output load. The open period of the latch still suffices to make a nearly complete swing. While the latch closes it completes its transition.

The third latch is critical. Due to the 0.8pF load it requires more time to make a complete transition. When the latch closes again, its output is still changing. The output now takes considerably more time to complete this transition. The fourth latch, with the 1.1pF load, does not make it. When the latch closes again its output has not yet passed the threshold. Therefore, when the feedback-path is created, the output of the latch overwrites the state of the latch (which was stored on the internal node).

The analysis done here is not carried out in great detail. There are two reasons for this. First, others have already characterized this mode of failure in detail. This type of problems is closely related to synchronization and metastability, as studied in, for example, [21, 68, 81, 50].

The second and more important reason is based on engineering practice. To build a working system, safety is an important issue. It is therefore useful to establish safety guidelines for the implementation of variables. At the output of latches that have a high fan-out, and thus face a high capacitive load, additional buffers are inserted to assure a timely transition (see Chapter 6).

Early/late data-valid schemes

In the above we assumed a broad data-valid scheme on the write channel of the variable. An implication of this was that we could open the latches during the full write handshake without wondering about valid input data. In this section we investigate the impact of having non-broad data-valid schemes on the write channel.

If the data-valid scheme on w is early, we have to adapt the latch-control circuits that we have seen so far. In the early scheme we may not assume valid data after $w_a\uparrow$, which implies that the latches have to be opened *and* closed in the up phase of the w-handshake. This can be achieved by using an S-element, basically to convert from early to broad. (See also Section 5.4.1.)

In case the data-valid scheme on the write channel is late we could of course also convert this to broad (by using a D-element). The latch-control circuits shown above, however, can also be used, since at the falling edge of w_r, which signals the beginning of the late data-valid period, the latches are still transparent. It now depends on the available setup time margins whether we have to delay the closing of the latch or not.

A consequence of making the latches transparent when the data is not yet valid, such as in the context of a late or a true four-phase data-valid scheme, is that the latches may toggle more than once, due to spurious transitions on the data inputs before the data is stable. These spurious transitions at the output of the latches are not optimal for low power. To prevent them one should enable the latches as late as possible, or, alternatively, use flip-flops instead of latches, such that the output does not switch before the latching has completed.

Minimum channel power

All implementations of the handshake variable discussed so far are based on a decomposition in which a write handshake results in the broadcast of the new state via all read ports. For the read handshake channels this possibly results in spurious data

transitions, since a write handshake is not necessarily followed by a read handshake. For the early, reduced late, and broad data-valid scheme on these read channels this is not a problem, since these transitions fall outside the data-valid period.

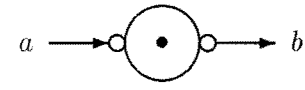
However, in the minimum power variants of these data-valid schemes the number of transitions on these channels are restricted to at most one per bit. To achieve this we have to implement the read ports of a variable with more than wires only. To achieve a minimum number of transitions we can only allow the data on a read handshake channel to change when a read request has been received. Therefore we have to latch the last-read data in all read ports independently. This requires a latch per bit per read port of the handshake variable.

A nice feature of these minimum power variants is that the write port can now be implemented with wires only, provided that the write channel also obeys the minimum power rule, and thus only changes data just before a write request arrives. To make this work, the other components should cooperate and also obey the minimum-power rule. For all handshake channels the value that was last communicated should be kept stable until new (valid) data arrives. This means that data has to be latched in more components, for instance in passivators and demultiplexers. It also limits the implementation freedom in the multiplexer, which complicates especially the control circuits for multi-input multiplexers.

5.3.3 Passivator

The passivator is used to synchronize two active partners during a data transfer. It synchronizes data exchange between an active sender and an active receiver. The symbol and specification of the passivator are given next.

$$
\begin{array}{l}
\text{PAS}(a^{\circ}?T, b^{\circ}!T) = \\
\quad [\![x, y : T \\
\quad [\, (a_r(x) \;\|\; b_r) \\
\qquad ; *(\, (a_a \,;\, a_r(y) \;\|\; b_a(x) \,;\, b_r) \\
\qquad\quad ; (a_a \,;\, a_r(x) \;\|\; b_a(y) \,;\, b_r)) \\
\quad]\!]
\end{array}
$$



The specification of the passivator is rather complex. Intuitively, one may expect the simpler command

$$[\![x : T \,[\, (a_r(x) \;\|\; b_r) \,;\, (a_a \;\|\; b_a(x))]\!].$$

This specification, however, is not receptive. It allows for a new request to arrive on a channel only after the acknowledge on the other channel has occurred. In the Tangram compilation scheme a receptive implementation of the passivator is required, since it is used to synchronize otherwise independent processes.

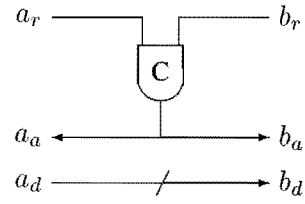


Figure 5.17: Latch-free passivator implementation

In a two-phase implementation, the specification dictates the use of latches for the datapath. After the acknowledge on a has been sent, the input data from a is no longer valid. On channel b, however, valid data must still be assured.

In contrast, in a four-phase implementation data-valid schemes for a and b exist that allow for a latch-free implementation of the passivator. In the compilation scheme of Tangram communication takes place only between independent parallel processes. Fortunately, the four-phase implementation of this parallel composition is such that it allows any interleaving of the intermediate phases of the four-phase handshake protocols on a and b without any danger of deadlock. The simplest implementation is to use a control passivator for the control of the data-valid signaling, as depicted in Fig. 5.17.

In this implementation the data-release signal of channel a is directly connected to the data-valid signal of channel b. This means that both the early and the late data-valid scheme on channel a lead to empty data-valid periods on channel b. The only choice therefore is to assume the broad data-valid scheme on a. The passivator in that case guarantees an early data-valid scheme on its output channel b.

The circuit depicted in Fig. 5.17 can thus be used to combine the broad data-valid scheme on a with the early scheme on b. For this combination, however, we can, at the expense of some extra circuitry, also use a circuit that allows a return-to-zero of b_a independent of whether a_r has already gone low.

This broad/early passivator can be used in a context where input is implemented with the early data-valid scheme for input communications (such as based on the mirrored transferrer) and the broad data-valid scheme for output (based on the sequencer transferrer).

Different data-valid combinations can be achieved by connecting data-valid converters to either of the channels. These data-valid converters are discussed in the next sections. If broad or late data-validity is required on the output channel, then the data must be latched in the passivator, since after the complete handshake on a data-validity is no longer guaranteed in any of the push data-valid schemes.

5.4 Push components

Two types of push components are discussed in this section, namely protocol converters and multiplexers.

5.4.1 Data-valid conversion

Connectors can be used to convert between data-valid schemes, if that is required. The symbol and the specification of the push connector are shown in Fig. 5.18.

$$\begin{aligned}&\text{CON}(a^{\circ}?T, b^{\bullet}!T) = \\ &\quad [\![\, x : T \mid * (a_r(x) \,;\, b_r(x) \,;\, b_a \,;\, a_a) \,]\!]\end{aligned}$$

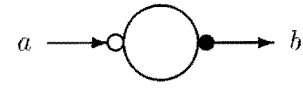


Figure 5.18: Push connector, which can be used for data-valid conversion.

The implementation of the connector is especially interesting if the data-valid schemes on channels a and b differ. A wide range of these type of components can be defined, but in the context of this thesis only the four-phase converters are of interest.

If the data-valid scheme on channel a is broad, and that on channel b is early or late, then the implementation of the converter can be done with wires only, since the data-valid period in these two schemes is a sub-period of the data-valid period in the broad scheme.

Two interesting cases are when the broad data-valid scheme on the output channel is combined with early or late data-valid schemes on the input channel. To establish a broad data-valid period on channel b we have to implement the converter such that the data-valid signal on channel a precedes $b_r\uparrow$, and the data-release signal on a is scheduled after $b_a\downarrow$, which is the data-release signal on b.

If the data-valid scheme on channel a is early, this leads to the specification shown below, in which a complete handshake on channel b is performed during the up handshake on a. The control component with this specification is known as the S-element [8, p. 162], also known as Q-element [57, p. 36], and is used in the implementation of the do-component in Section 5.2.3.

$$*(a_r\uparrow \,;\, b_r\uparrow \,;\, b_a\uparrow \,;\, b_r\downarrow \,;\, b_a\downarrow \,;\, a_a\uparrow \,;\, a_r\downarrow \,;\, a_a\uparrow)$$

In the case of a late data-valid scheme on channel a, the D-element [57, 8] can be used to convert to broad data-validity. The D-element schedules the handshake on b during the down handshake on a. The implementation of both converters is depicted in Fig. 5.19. As can be seen, the data part of both components is implemented with

wires only, and the control hardware takes care of the proper ordering of the data-valid and data-release signals.

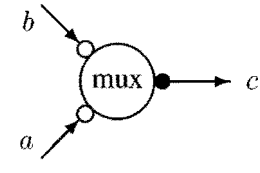
Figure 5.19: Implementation of early (left) and late (right) to broad converter for push channels.

Push protocol converters with a broad data-valid scheme on the output channel can for instance be used in the implementation of variables (which require a broad data-valid scheme on their write port), or in output communications (to allow the environment to use its favorite data-valid scheme).

5.4.2 Multiplexer

The multiplexer is used to merge (type compatible) data streams onto one channel. For proper operation the handshakes on the passive channels have to be mutually exclusive.

$$
\begin{array}{l}
\text{MUX}(a^{\circ}?T, b^{\circ}?T, c^{\bullet}!T) = \\
\quad [\![x : T \mid \\
\quad\quad * \, (\, a_r(x) \, ; c_r(x) \, ; c_a \, ; a_a \\
\quad\quad\quad \mid b_r(x) \, ; c_r(x) \, ; c_a \, ; b_a \\
\quad\quad) \\
\quad]\!]
\end{array}
$$



This specification of the multiplexer can easily be generalized to multi-input multiplexers. Multi-channel multiplexers form an attractive substitute for trees of binary multiplexers. First of all they potentially offer more-or-less uniform latency to all clients, and second they can be implemented efficiently. Therefore, the implementation of both binary and multi-channel multiplexers are addressed below.

Global organization

In any implementation of the multiplexer, three things have to be taken care of. First of all, the data of one of the incoming channels has to be forwarded to the output channel. Secondly, the associated data-valid signal should be derived from the incoming data-valid signals and should compensate for any delay that might be encountered by the data. The third thing that has to be done is to send the incoming

data-release signal (from the output channel) to the input channel that sent the data-valid signal.

We decompose the multiplexer into a control part, which takes care of the request and acknowledge signaling, and a data part, which takes care of the forwarding of the data. The interface between these two modules consists of select lines, which are signals from the control part to the data part that identify the input that has to be selected. This decomposition is schematically depicted in Fig. 5.20.

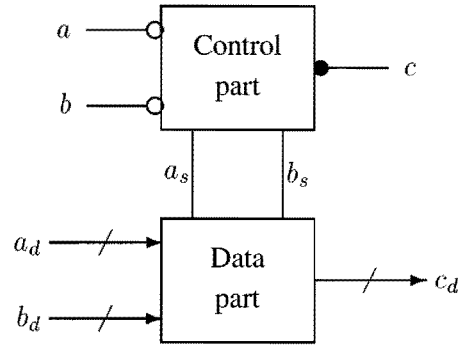


Figure 5.20: Global organization of a two-input multiplexer.

Below, we first discuss the implementation of the data part, and then that of the control part. One may already observe that apart from generating the select signals a_s and b_s, the function of the control part is equal to that of a nonput mixer. The exact specification of the select signals depends on the implementation of the data part.

Multiplexer cells

In the data section of the multiplexer we have to direct input data to the output, based on the status of the select lines. Two options for this are addressed here, namely a bus-like implementation based on tri-state cells, and a direct realization based on complex CMOS gates.

A solution that is especially attractive for many-input multiplexers is to use tri-state inverters in combination with a bus-keeper (a cell that keeps the bus stable when no other cell is driving it). One bit section of a four-input multiplexer, with inputs a, b, c, and d, and output q is shown in Fig. 5.21. In this solution at most one select signal is allowed to be high at any time. Otherwise an electrical conflict may occur, which results in short-circuit. A nice property of this implementation is that it basically latches the output of the multiplexer cell. This means that after valid data

has been assured, the corresponding select signal can return to zero. Especially for many-input multiplexers this simplifies the control circuit.

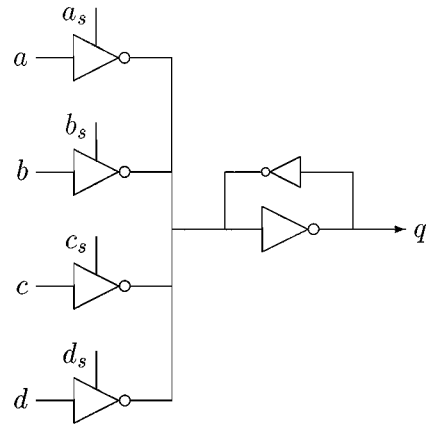


Figure 5.21: Circuit realization for one bit of the data section of latched four-input bus-like multiplexer.

A drawback of the tri-state solution is that a tri-state inverter requires both the true and the complement of its select signal. Especially in a standard-cell implementation this leads to area inefficiencies. One can choose to distribute both the true and the complement of the select signals (the 'dual rail' distribution), or invert the select signal in the multiplexer cell. In the latter case this inverter only drives one transistor, in the former case the multiplexer cells require excessive routing.

An alternative realization of the multiplexer is the direct implementation of the sum-of-products function $q = a_s * a + b_s * b + c_s * c$ (for a three-input multiplexer; generalization is straightforward). Especially if complex gates known as AND-OR-INVerts (AOIs) can be employed, this function can be realized efficiently. From an electrical point of view this realization does not impose any restriction on the implementation of the select signals.

Mixer-based control circuit

The function of a multiplexer control circuit is twofold. First of all, the control circuit has to generate appropriate select signals for the multiplexer cells. In addition to this the handshake signaling on the handshake channels has to be taken care of. The latter also involves the generation of a data-valid signal on the output channel, and depends on the data-valid scheme that is chosen and on the implementation of the multiplexer cells.

If we ignore the data-valid obligations and the select lines, then the control circuit can be implemented as a control mixer. So, an approach to the design of a multiplexer control circuit is to start with such a control circuit, and then add gates to control the select lines. The implementation of the nonput mixer is shown in Fig. 5.2.

Data at the output of a multiplexer cell need only be valid during the data-valid period. This implies that the state of the select lines outside this period may be arbitrarily chosen. Throughout this section we assume active-high select lines.

If the data-valid scheme is early, the data is supposed to be valid on a handshake channel during the period in which the request is high and the acknowledge is low. For the control circuit of the multiplexer one can thus choose, for input channel a, for example, $a_s = a_r * \overline{a_a}$. Naturally, we may also decide to keep the select lines stable for a longer period, and thus simplify the expression to $a_s = a_r$.

In case the data-valid scheme would be late, we can for similar reasons choose $a_s = a_a$. When a broad data-valid is required, however, we cannot simply use the request or the acknowledge signal as a select signal, but rather we have to choose $a_s = a_r + a_a$.

The forwarding of the data-valid signal in the mixer control circuit is implemented by the OR-gate. This does not take the delay-matching obligations with the datapath into account. To provide a perfect match the data-valid signal should be forwarded with the same circuitry as the data in the datapath. This means that for instance the multiplexer cell with $c_r = a_s * a_r + b_s * b_r$ could be used.

A disadvantage of the above implementations of the select lines is that the signals return-to-zero between handshakes. This clearly does not minimize the switching activity of the select lines. We therefore switch to a different kind of control circuit, based on complementary select lines.

Complementary-select control circuit

A different way to design a control circuit is to start with the select lines, that is, first design a circuit that controls the select lines, and later extend this circuit to include the handshake signaling as well.

From a low-power point of view it is best to minimize the switching activity of the select lines, because these signals may fanout to many multiplexer cells. We therefore opt for a solution in which the select lines only switch state when necessary. This implies that we (i) latch the select lines, and (ii) keep the select lines complementary, such that the last selected input remains selected between handshakes. If we furthermore choose to switch the select lines as early as possible in the handshake, that is, at an incoming request, this leads to the following specification of the

associated circuit.

$$\begin{cases} a_r & \mapsto & a_s\uparrow, b_s\downarrow \\ b_r & \mapsto & a_s\downarrow, b_s\uparrow \end{cases}$$

This specification can straightforwardly be implemented as a set-reset latch with complementary outputs. Two (cross-coupled) NORs suffice for this, and for high fanout select signals, two extra inverters can be added to provide sufficient driving strength.

The next issue is the forwarding of the request. The data-valid signal of handshake channel c is encoded in its request, so we have to be careful with the request signaling. If the data-valid scheme on the input channels of the multiplexer is early, and we want to have an early data-valid scheme on c as well, then we have to delay an incoming request sufficiently long to allow the select lines to switch and, subsequently, the data to propagate.

A specification of the request signal on c in which at least the switching of the select lines has been taken into account is the following.

$$\begin{cases} a_r * a_s + b_r * b_s & \mapsto & c_r\uparrow \\ \overline{a_r} * \overline{b_r} & \mapsto & c_r\downarrow \end{cases}$$

Since we decided to leave the select lines in the last selected state, we can implement the above specification by $c_r = a_r * a_s + b_r * b_s$. An advantage of this implementation is that the request forwarding can be done with exactly the same cell as the actual multiplexing of the data, as shown in Fig. 5.22.

The last task that has to be implemented is the generation of the acknowledges on the input channels. For the mixer this is specified as follows. (Only the specification of a_a is given. It is similar for b_a.)

$$\begin{cases} c_a * a_r & \mapsto & a_a\uparrow \\ \overline{c_a} & \mapsto & a_a\downarrow \end{cases}$$

It is again helpful that the select lines are being latched in the control circuit. This allows us to implement the above specification with an AND-gate, namely as $a_a = c_a * a_s$, instead of the asymmetric C-element that is actually specified.

The complete circuit realization of a two-input multiplexer is shown in Fig. 5.22. The width of the multiplexer may vary: multiple multiplexer cells can be straightforwardly connected to the select lines.

The delay that data encounters in the multiplexer of Fig. 5.22 is perfectly matched in the control, since the same circuit is used for forwarding the request as for the actual multiplexing of the data. Actually, the select lines are treated as extended isochronic forks; in the figure of depth two. This makes the circuit a safe implementation in any data-valid scheme. In the context of true four-phase communication, however, a simpler request circuit can be used.

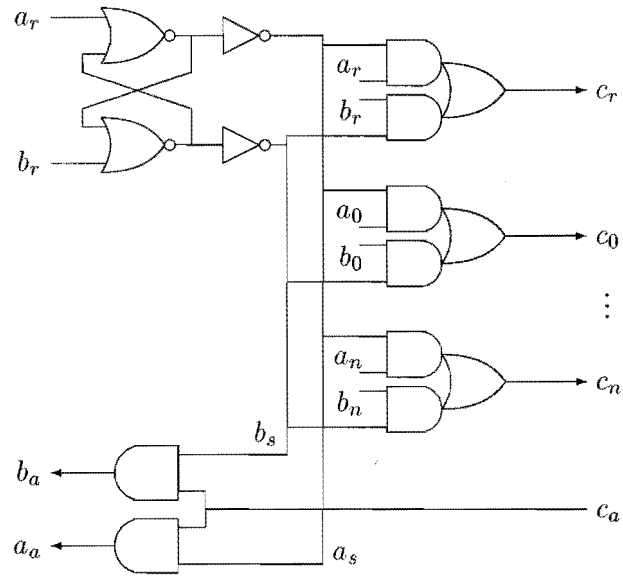


Figure 5.22: Two-input multiplexer, showing select, request, acknowledge, and (two) data signals.

Simplified control circuit

In the true four-phase protocol, data validity is established in two phases, and we can use both the up and the down phase of the handshake to match the delay of the datapath. This means that we can implement a quick forwarding of the request, for example, as quickly as possible, that is, with an OR-gate, as shown in Fig. 5.23.

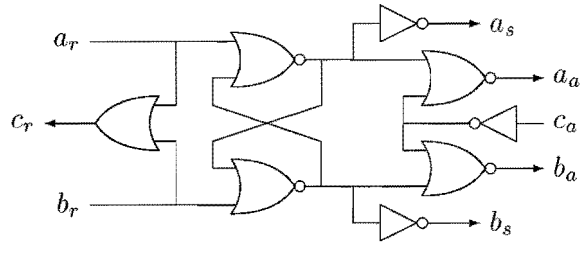


Figure 5.23: Simplified control circuit for two-input multiplexer based on quick forwarding of the data-valid signal.

A consequence of the quick forwarding of the request is that, from a speed-independent point of view, when the acknowledge arrives we cannot be sure that the select-lines have already switched to the correct state. In general, however, the

path from, for example, $a_r\uparrow$, via $c_r\uparrow$ to $c_a\uparrow$ requires more time than the path from $a_r\uparrow$ to $a_s\uparrow$. Therefore the acknowledge signals can generally be safely derived from the select signals, as shown in Fig. 5.23.

Biased control circuit

In the control circuits discussed so far it is assumed that both select signals are distributed, but of course one could also choose to have a local select-line inverter in each multiplexer cell. This then guarantees that the select lines are always complementary, but does not necessarily minimize the switching activity on the select lines.

An interesting option then is to choose the simple select generation circuit based on the nonput mixer. This can be used to implement a *biased* multiplexer, which by default selects some input, and only during the handshake on the other input selects that one. For multiplexers that have an unbalanced invocation profile, in the sense that one input is selected significantly more often than the other, a biased control circuit is as good as a latched control circuit.

Quantitative aspects

From the control circuits and multiplexer cells that are discussed above several multiplexers with different performance characteristics can be built.

One of the degrees of freedom is the safety margin that is taken into account in the delay matching of the data-valid signal with respect to the datapath. In the true four-phase protocol the falling edge of the request signal counts as the definite data-valid signal, but the delay-matching can be spread over the several phases of the handshakes.

From a low-power point of view it seems a good idea to use a control circuit that minimizes the number of transitions on the select lines. This is achieved in the control circuits that store the last selected state, such as the control circuits of Fig. 5.22 and Fig. 5.23. A disadvantage of this scheme, however, is that the outputs of the multiplexer cells always follow the last selected input, which may give rise to switching activity outside the data-valid period (that is, in between handshakes). Therefore, for circuits in which a lot of spurious data transitions are expected, a solution based on return-to-zero select lines may result in less switching activity. This can be combined with both latched tri-state and AOI multiplexer cells.

5.5 Pull components

Expressions in Tangram are compiled to pull components. Pull components are connected to pull handshake channels only. This means that expressions are evaluated in a demand-driven way.

We first briefly look at data-valid conversion on pull channels, and then discuss the implementation of operators (unary and binary) and the demultiplexer.

5.5.1 Data-valid conversion

Connectors can be used to convert between data-valid schemes, if that is required. The symbol for a pull connector with input port b and output port a is shown below.

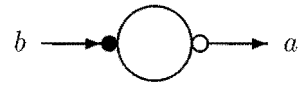


Figure 5.24: Pull connector, which can be used for data-valid conversion.

The interesting data-valid conversions are again those from early or late to broad. Surprisingly, it is the conversion from late to broad that can be implemented relatively cheaply, that is, with wires only for the data part. If the data-valid scheme on b is late, then the data-valid signal is $b_a\uparrow$. We have to schedule this before $a_a\uparrow$ to assure a broad data-valid scheme on a. This can be achieved by performing the full handshake on b during the up handshake on a, for which an S-element suffices.

Conversion from early to broad cannot be achieved without latching the data, because this would require to withhold the data-release signal on channel b ($b_r\downarrow$) until after the data-release signal on channel a. This data-release signal is $a_r\downarrow$ of the *next* handshake. So, a latch-free pull converter from early to broad would imply a form of deadlock for channel b. The converter can be implemented with latches that are opened at $b_r\uparrow$, and closed at $b_r\downarrow$.

The true four-phase protocol resulted in a reduced broad data-valid scheme for pull channels. To convert this to 'full' broad data-valid schemes the latching converter can be used as well.

5.5.2 Datapath operators

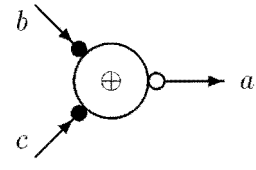
One of the more interesting aspects of single-rail handshake components is the implementation of operations on data. The data part of the single-rail operators is similar to well-known synchronous realizations; the interesting choices are in the kind

of delay matching or completion detection that must be applied to generate appropriate data-valid signals.

Tangram offers several unary and binary operators, both for booleans and for integers. Other operators and types can be explicitly programmed in the language. We pick two examples of binary operators, namely the XOR for booleans, and the subtractor for integers. The complete set of Tangram operators is described in the Tangram manual [71]. The two examples given here are intended to be representative.

First the generic specification of the binary operators is given. Operators are evaluated in a demand-driven way, that is, each cycle of operation is initiated by a request for a result. After this request is forwarded and the operands are collected, the computed value is sent as an acknowledge to the initial request, which completes the cycle.

$$
\begin{array}{l}
\text{BIN}(\oplus, a^{\circ}!T_a, b^{\bullet}?T_b, c^{\bullet}?T_c) = \\
\quad [\![x : T_b, y : T_c \\
\quad \mid * (a_r \,;\, (b_r \,;\, b_a(x) \;\|\; c_r \,;\, c_a(y)) \,;\, a_a(x \oplus y)) \\
\quad]\!]
\end{array}
$$



The type of the result channel, especially the number of bits that are needed to represent these, depends on those of the operand channels. This typing information is kept track of during the compilation process from Tangram to handshake circuits and is defined as follows.

$$T_a = T_b \oplus T_c = \{x, y : x \in T_b \wedge y \in T_c : x \oplus y\}$$

In the implementation of all operators we strive for the same data-valid scheme on all channels. For the circuit this means that, given a data-valid scheme on the inputs, the same data-valid scheme should also be guaranteed on the output channel.

Boolean operators

The implementation of boolean operators on data is simple, since the variety of operators is small and they all operate on 1-bit operands and deliver a 1-bit result. As an example the circuit realization of the logical XOR is shown in Fig. 5.25. The interesting aspects of this implementation are the C-element and the delay-element in the acknowledge path.

Intuitively one would say that the combined delay of the C-element and the delay-element should match that of the XOR. It then depends on the actual data-valid scheme at hand and the safety margins one wants to take into account, how

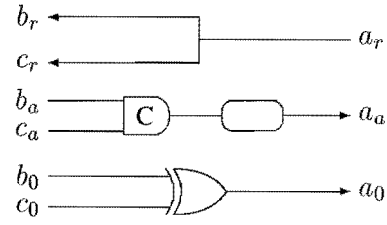


Figure 5.25: Implementation of logical XOR. The delay through the C-element is not taken into account in the delay-matching since it is always (in Tangram handshake circuits) eliminated during post-optimization.

the delay should be dimensioned. Fortunately, however, in Tangram handshake circuits the C-element is always redundant, in the sense that it always re-combines the forked request, possibly via some delayed paths. This issue is discussed in detail in Chapter 6 and relates to the wire-only implementation of the control circuit for read ports in variables and the fact that, in Tangram, no inputs are allowed in expressions. For now it suffices to know that the C-element is generally removed and thus the delay-element should take care of all the matching.

For the implementation of the delay element several options remain. We only discuss symmetric delay implementations, since asymmetric delays (with a fast reset or preset) are not required in the true four-phase protocol. For the matching of the delay of a gate an equivalent gate could be used. An AND-gate delay could for instance be matched by an AND-gate with inputs connected together. For the XOR, however, this is not a good idea, since any delay element based on an XOR would possibly result in hazards at its output.

We have chosen to use a unit-delay element as a basis for all delay matching. This allows for simple post-optimization rules, for instance for the elimination of parallel delays (cf. Chapter 6). The implementation is shown in Fig. 5.26. Given our standard-cell library, this implementation is such that (in a first-order approximation based on delay-versus-load characteristics) the sum of the delays of the NAND-gate (with the NOR-gate transistor load and zero wire load) and the NOR-gate (assuming an average total load C) matches the delay of an XOR-gate with average load C. Further quantitative analysis of delays and the matching thereof is given in Chapter 6.

There is something interesting about the relation between the two-phase specification and the four-phase implementation. According to the specification the handshakes on b and c should be able to complete independently. The four-phase implementation, however, synchronizes the two operands with a C-element, which may give rise to deadlock. Fortunately (or intentionally), in Tangram handshake

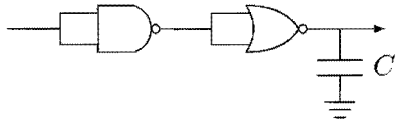


Figure 5.26: Implementation of a unit delay element

circuits this deadlock is guaranteed not to occur. This is of course related to the fact that the C-element can be removed, as mentioned above.

The reason behind the absence of deadlock is that for the evaluation of expressions (in which the operands occur) a reduced form of parallelism known as *mutual inclusion* suffices. At the Tangram level this means that the evaluation of an expression can be seen as an atomic action. If we would allow input communications in expressions, then with the implementation sketched here, communications occurring in one expression would be synchronized, that is, cannot complete independently. In this situation the C-element would not be removed in the post-optimization phase.

Integer operators

For the implementation of operations on integers, one first has to choose a representation of the integers. The choice of a particular integer representation has an important impact on the implementation of integer operations (addition, inversion). Two representations are frequently used in computer arithmetic, namely, one's and two's complement. Negation in one's complement amounts to bit-wise inversion (which can be done in constant time), and is more complex in two's complement, where it is equivalent to incrementing the bit-wise inverse. Addition and multiplication, however, are more straightforward in two's complement than in one's complement.

In Tangram, the choice has been made to represent signed integers in two's complement, and unsigned integers without the (zero-valued) sign-bit. Throughout this thesis we stick to this choice. Operations based on other numbering systems can be programmed directly in Tangram. Examples of multipliers in various representations are described by Haans [38].

For an integer variable x of N bits, we use x_i, with $0 \leq i < N$, to refer to the individual bits, in which x_0 refers to the least significant bit and x_{N-1} to the most significant bit. If x is a signed integer, then x_{N-1} is generally referred to as the sign-bit.

An unsigned variable x of N bits thus represents

$$(\sum i : 0 \leq i < N : x_i * 2^i),$$

which is a value in the range $[0 \ldots 2^N - 1]$. A signed variable x of N bits represents

$$(\sum i : 0 \leq i < N-1 : x_i * 2^i) - x_{N-1} * 2^{N-1},$$

which is in the range $[-2^{N-1} \ldots 2^{N-1} - 1]$ of signed numbers.

For the details of two's complement arithmetic we refer to Hwang [46]. In this section we focus on the single-rail aspects of the arithmetic operations and we do this by using subtraction as an example. For now it is sufficient to know that subtraction in two's complement in it simplest form requires a carry to ripple from the least significant to the most significant bit.

A schematic representation of the circuit realization of a 4-by-4 bit subtraction with a 4-bit result is shown in Fig. 5.27. Each box labeled with a '$-$' represents a half or full subtractor. The matched-delay path that is depicted consists of five unit delays. This, of course, is a tentative match. It depends on the actual implementation of the carry-path (always positive, or alternatingly positive and negative representation) and of the delay from the carry in to the result bit in the last subtractor cell. The C-element should be considered as a zero-delay element, since —in Tangram handshake circuits— it will be eliminated during post-optimization.

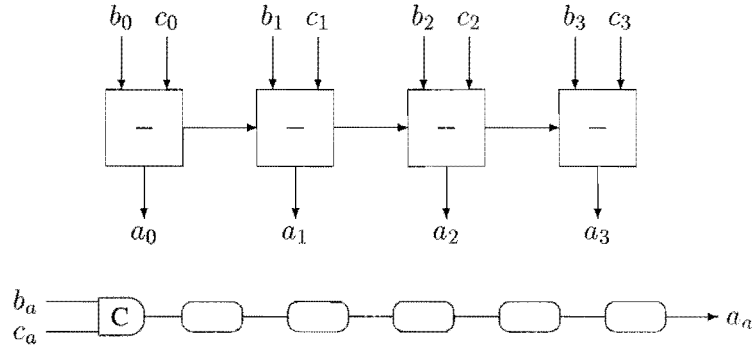


Figure 5.27: Implementation of subtraction with single-rail carry. The C-element should be interpreted as having zero delay. The delay chain matches the worst-case path through the subtractor.

An alternative for the independent delay-matching path as in Fig. 5.27 is to make the data-valid signaling part of the computation. Phrased differently, one could let the computation itself signal when it has completed. An example of such an approach is shown in Fig. 5.28, where a double-rail carry is used to control the subtraction. The computation is started by injecting a start-carry in the head-cell, and upon arrival of the carry at the tail (detected by the OR-gate) one can assume all bits to have settled to their correct value.

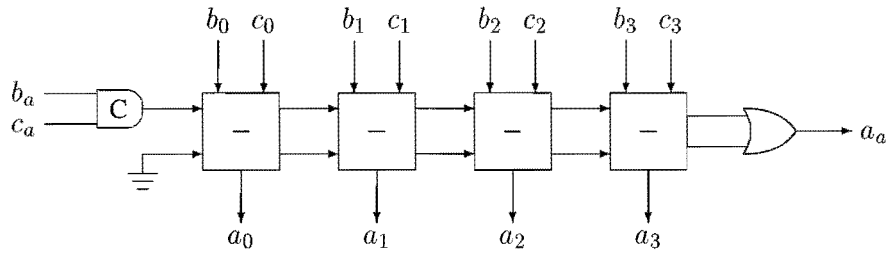


Figure 5.28: Implementation of subtraction with double-rail carry, with (redundant) C-element as initiator and OR-gate for completion detection.

In the implementation shown in Fig. 5.28 the carry is reset during the return-to-zero phase of the handshake. This implies that it can only be combined with the early data-valid scheme. In a late data-valid scheme we could make the carry ripple during this return-to-zero phase. If the data-valid scheme is broad or true four-phase, however, this is harder.

One of the often cited advantages of asynchronous circuits is their average-case promises, that is, in an asynchronous circuit it should be possible to take advantage of (or, profit from) average case processing times. The subtractor is a good example for this. The matched path in Fig. 5.27 anticipates on the worst-case processing time, that is, it can accommodate a full-length carry ripple plus the computation of the most significant bit. On average, however, the subtraction may complete faster. For this four-bit subtractor the difference is minimal, but for a 32-bit subtractor the difference between the average and the worst case is significant.

This observation was already made in 1955 in a paper by Gilchrist [36], and he proposes double-rail encoding to achieve this. Even with only a double-rail carry one can implement this, but then one should broadcast the data-valid signal of the inputs to more bits, such that multiple carries may start to ripple independently. Hwang, in his book on computer arithmetic, also discusses such double-rail techniques, see [46, pp. 75–78] on self-timed adders. This observation was also made by Seitz [72].

The ALU of the AMULET1[2] has been implemented using such a double-rail carry-scheme [34, 63]. It has been designed as a four-phase sub module, operating in an otherwise two-phase handshaking environment. The early data-valid scheme has been employed, which combines naturally with the dynamic CMOS domino logic that implements the data operations.

In this ALU, the evaluate signal (which signals both the completion of the pre-charge phase and the validity of input data) is distributed to all bit sections. Each

[2]An asynchronous version of the ARM, designed at Manchester University.

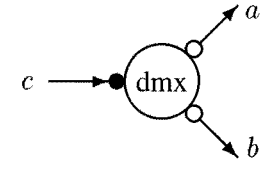
full adder can then generate a (double-rail encoded) carry-out signal as soon as the majority of the two operands and the carry-in have assumed the same value. To detect completion of the operation, the arrival of the carry at all stages is detected. Implementation details of this ALU can be found in [34].

Another way to achieve an exact data-valid signal is to use *current-sensing completion detection* [26, 27, 37]. This is a technique that exploits the property of CMOS that charge and discharge currents are some orders of magnitude larger than leakage currents. Therefore, when a computation has completed, and all wires have set to their final value, this can be detected. It is not obvious, however, at which grain size this should be applied. Furthermore, this does not combine very well with a standard-cell layout style, since completion detection requires isolated power or ground rails. It has also not been demonstrated yet how to efficiently implement this in CMOS.

5.5.3 Demultiplexer

The demultiplexer does not directly correspond to any language construct in Tangram. It is used to organize mutually exclusive access to an input channel, such as required in the Tangram program for the wagging buffer on p. 19. The demultiplexer can thus be considered as the pull variant of the control mixer, in the sense that it allows (mutually exclusive) sharing of pull channels, whereas the mixer implements sharing of nonput channels. Of course the multiplexer can be seen as the push variant of the mixer. The specification of the demultiplexer and its symbol are given next.

$$
\begin{array}{l}
\text{DMX}(a^{\circ}!T, b^{\circ}!T, c^{\bullet}?T) = \\
\quad [\![x : T \\
\quad | * (a_r ; c_r ; c_a(x) ; a_a(x) \\
\qquad | b_r ; c_r ; c_a(x) ; b_a(x) \\
\qquad) \\
\quad]\!]
\end{array}
$$



The implementation of the demultiplexer is simple. For the control circuit a nonput mixer suffices. The data from c can be forked to a and b, in which case the data-valid schemes on these outputs are the same as the data-valid scheme on the input. The control overhead even adds to the already existing safety margin.

A possible pitfall of course is the increased input load for the gates driving the data wires of channel c. For a safe implementation of something as simple as forking a good driver strategy is required to keep transition times bounded. Otherwise even forking may invalidate the delay matching. This issue is covered in more detail in Chapter 6.

If the number of transitions on the channels has to be minimized, then the two outputs should be independently latched. The particular latching scheme depends on the data-valid scheme of channel c. Thereby, an output of such a demultiplexer is similar to the readport of a variable with minimum channel transitions.

5.6 Summary

In this chapter we discussed the implementation of the handshake components that are essential to the cost and performance trade-offs as they are made in Chapter 4. Most attention was paid to the implementation of the transferrer, the variable, the multiplexer, and arithmetic. For all these components the true four-phase realizations —in which the work is partly performed in the up-phase of the handshake and completed during the down-phase— are area-efficient.

The implementation of handshake components in the datapath is split in a data part (for the operations on the data) and a control part (which is involved in the handshaking). The control and data part are sometimes connected by some control signals, but the implementation can often be addressed independently. The circuit realization of the data part generally is straightforward, and can directly be based on the implementation of equivalent synchronous functions.

In the implementation of the control part of the components, trade-offs are made between area and timing assumptions. For most circuits it holds that the more timing assumptions that are made, the smaller the resulting circuits are.

Chapter 6

Design Flow

One of the innovative aspects of our approach to single-rail asynchronous circuits is that they are compiled from a high-level VLSI programming language. This requires us to pay attention to two essential goals: on the one hand the resulting circuits should be push-button correct, whereas on the other hand they must be area, timing, and power (energy) efficient. It turns out that these two goals are not necessarily conflicting.

In this chapter we discuss the design flow from Tangram to (single-rail) silicon. This flow can be characterized as a transparent, syntax-direct compilation. The potential weakness of this building-block approach is that it leads to inefficiencies, especially at the boundaries of these building blocks.

An important aspect of the design flow, therefore, is the peephole optimization that is applied at several phases during the compilation process. This issue has previously not been highlighted, but receives considerable attention throughout this chapter. It is argued that the combination of simple building blocks and simple peephole optimization rules results in efficient silicon.

The design flow is treated step by step, and for each step the importance of peephole optimization is illustrated. This is accompanied by numerous examples.

6.1 Design flow

The design flow from Tangram to single-rail silicon is schematically depicted in Fig. 6.1. The boxes stand for representations of the design, and the arrows represent design steps. Peephole optimization is considered to be an important issue in any compiler and should be considered at all levels of representation [59, 76].

Tangram is the language in which the VLSI programmer describes a design. Several tools are available that give accurate feedback on timing, energy, area, and

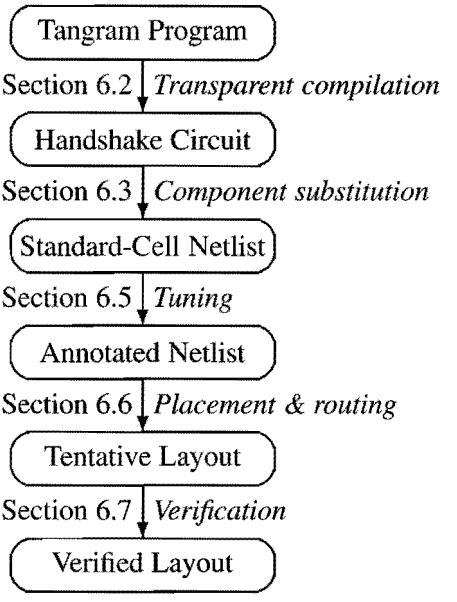


Figure 6.1: Push-button single-rail design flow.

testability at the level of the Tangram program. These tools were briefly touched upon in Chapter 2. For the single-rail design flow, Tangram is considered to be the starting point.

Since Tangram is the first level of representation that is encountered, it is also the first level at which a compiler could start optimizing. This, however, is not considered in this thesis. First of all, manipulations at the Tangram level are considered to be the domain of the VLSI programmer. Secondly, most optimizations that can be achieved at the Tangram level (if not all) can also be realized at the handshake circuit level, due to the transparent syntax-directed translation. Examples of this are given later.

In the rest of this chapter the design flow is discussed step-by-step, and after each step the importance of peephole optimization at that level of representation is illustrated by some examples.

6.2 Tangram compilation

Compilation from Tangram to handshake circuits is based on syntax-directed translation. During the compilation of a Tangram program an abstract syntax tree is constructed that represents the program. This is subsequently mapped onto a handshake

circuit. Details on the compilation may be found in [6, 16, 8]. Several examples are reviewed later in this chapter.

Since a handshake circuit is essentially just another representation of the syntax tree of the Tangram program, the VLSI programmer has detailed control over the handshake circuit that is generated, and thereby over the performance characteristics of the corresponding VLSI circuit. This property of the compiler is called *transparency*, and it basically means that properties of the VLSI implementation of a program can be related to the Tangram source program. This transparency is the basis of powerful analysis tools that give the VLSI programmer detailed information on performance characteristics of his Tangram program (area, speed, energy, testability) in a format that allows him (or her) to directly relate this to the program. The testability of a program, for example, can readily be analyzed at the Tangram level [79].

Compilation from Tangram to handshake circuits is independent of the style of implementation of these handshake circuits. It is not immediately obvious that this independency is optimal for all implementations of the handshake circuit. It turned out, however, that although the compiler from Tangram to handshake circuits had been designed with double-rail implementations of these handshake circuits in mind, these handshake circuits were indeed a viable starting point for single-rail implementation.

A potential weakness of transparent syntax-directed translation is that of inefficiencies at the handshake-circuit level. Therefore, peephole optimization is recognized as an important post-processing step. Throughout this chapter peephole optimization plays an important role. It turns out that these optimizations generally improve even the (intuitive) transparency of the compilation process, in the sense that they make it easier to derive performance characteristics of the silicon form the Tangram program.

6.2.1 Peephole optimization

One class of optimizations that can be applied at the handshake circuit level is to apply obvious improvements that have apparently been overlooked by the VLSI programmer, such as sharing of datapaths. Another important class of optimizations is that of multi-channel components.

Sharing

Compilation from Tangram to handshake circuits is (currently) based on explicit sharing, that is, the VLSI programmer has to indicate in the program which parts of the datapath have to be shared. The motivation behind this is that the designer

can make trade-offs at the Tangram level, for example to minimize the circuit area under the restriction that the performance criteria are met. In some cases, however, there is no trade-off, and one alternative (sharing) is obviously better than the other in *all* dimensions (area, time, energy, test). These cases are not recognized by the Tangram compiler, but could easily be recognized in the handshake circuit.

An example of a handshake circuit in which sharing is beneficial is shown in Fig. 6.2. The two transferrers collect data from the same source (variable x) and send it to the same destination (channel a). This handshake circuit might correspond to a Tangram program in which a statement such as `a!x` or `y:=x` occurs twice. In the first case channel a in the handshake circuit corresponds to channel `a` in the program, in the second case, channel a corresponds to the write port of the handshake variable that implements `y`.

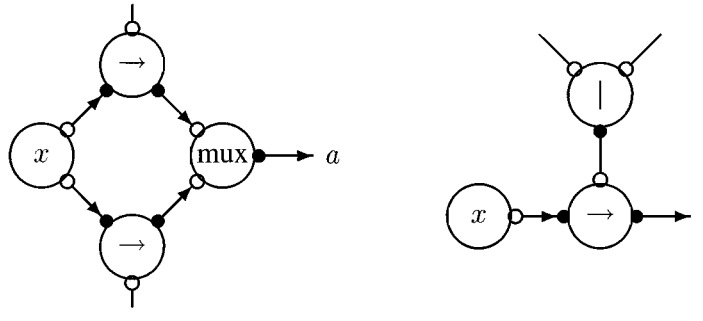


Figure 6.2: Sharing datapaths, showing pattern (left) and improved replacement (right).

The replacement of this handshake circuit is better in terms of area, since the multiplexer cells (one per bit) are saved. Since these cells and their select lines contribute to the switching activity in the original circuit, the substitute is also more energy efficient. Furthermore, the control circuit of the mixer is simpler than that of the multiplexer, which makes the replacement faster.

Surprisingly, the replacement above even improves the testability of the circuit, in the sense that it allows for shorter test traces. Testing the original realization involves testing the multiplexer, for which it is required to provide it with complementary data via each branch. This means that each transferrer should be activated twice. In the optimized circuit the mixer should be activated from each branch, which can be combined with complementary data-values to test the datapath.

Reordering

Reordering of handshake components may also lead to improvement. In the handshake circuit in Fig. 6.3 we can save one mixer and one sequencer, provided that the orientations of the sequencers are equivalent. This is another example of improved sharing that can also be achieved at the Tangram level. From the handshake circuit in Fig. 6.3 (left), and given the transparency of the compilation from Tangram to handshake circuits, one may conclude that the source Tangram program has the following form.

```
begin
    procA : proc(). ...
  & procB : proc(). ...
|   ...
  ; procA() ; procB()
    ...
  ; procA() ; procB()
    ...
end
```

The improved handshake circuit can be obtained directly from the Tangram compiler by introducing a declared procedure for `procA();procB()`, and then replace the two occurrences of this text by calls to that procedure.

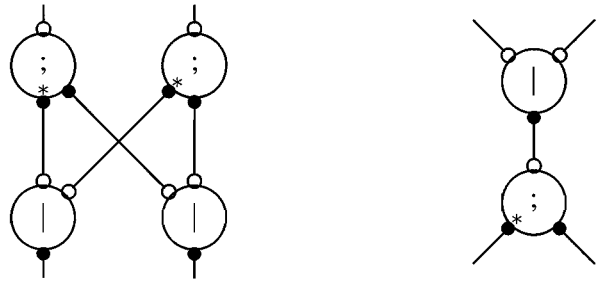


Figure 6.3: Sequencer-mixer reordering. Pattern (left) and optimized replacement (right).

A similar optimization in the datapath is shown in Fig. 6.4. For this substitution to be allowed, it is required that the split components are of the same type. The gain then is one multiplexer control circuit and one C-element. The width of the new multiplexer, is the sum of the two original multiplexers, and the split is of the same type as the originals.

The split-multiplexer reordering is harder to repair at the Tangram level. It may originate from two tuple assignments of the form `<<x,y>>:=...`, and it depends

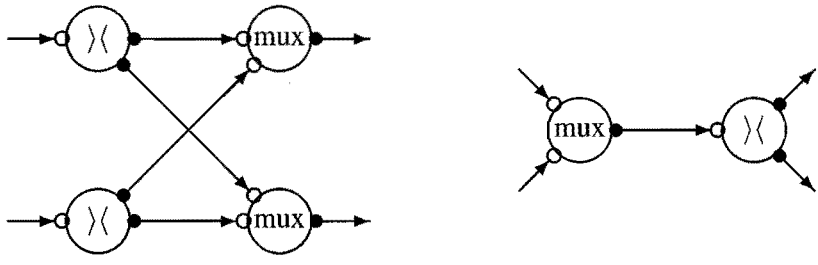


Figure 6.4: Split-multiplexer reordering. Pattern (left) and optimized substitute (right).

on the choices in the Tangram compiler whether it maximizes the sharing of these datapaths. Whereas at the Tangram level (or for a Tangram compiler) this may be hard to optimize, at the handshake circuit level it is a rather trivial optimization.

Multi-channel components

Nearly all handshake components that are used by the compiler have a fixed number of handshake ports. An exception is the variable, for which the number of read ports may vary. For all other components, trees of 'binary' handshake components are built if actually a multi-channel variant is required. These trees either have to be balanced by the compiler, or somehow the VLSI programmer must indicate the structure of such trees, for instance by the use of brackets, to force the compiler in a certain direction.

Examples of components that regularly occur in trees are control components like sequencer, parallel, and mixer, and data components like (de)multiplexer, split, and adder. Instead of building trees constructed from the binary versions of these components, one could use parameterized (multi-channel) versions.

The gate-level implementation of such a parameterized component can always be based on a (balanced) tree of the binary component. Generally, however, more efficient implementations exist. The approach that is followed in the Tangram design flow is to replace trees of components by multi-channel components, whenever the latter can be implemented more efficiently.

The control mixer generalizes trivially to multi-channels. As an example the three-channel mixer is shown in Fig. 6.5. For each additional passive port an extra asymmetric C-element and an extra input to the OR-gate are used. Larger OR-gates can be constructed from NANDs and NORs. In terms of area this multi-channel mixer is clearly an improvement over any tree realization. For a mixer with N passive ports only N asymmetric C-elements are required, against $2N - 2$ for an equi-

valent tree of binary mixers.

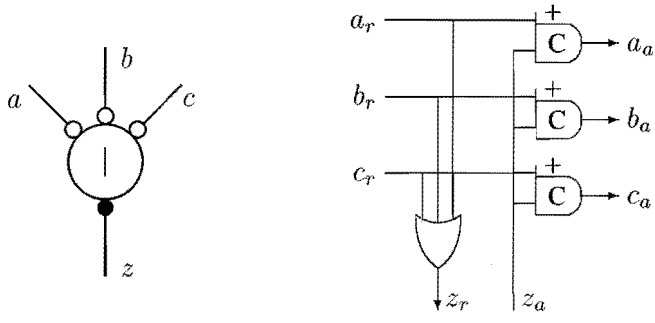


Figure 6.5: Three-channel mixer: symbol (left) and its implementation (right).

Since the fanout of the acknowledge signal of the active port increases, at first sight it seems that the multi-channel implementation loses in terms of energy per handshake. In a tree realization, however, depending on the path that the acknowledge has to follow, multiple asymmetric C-elements have to be switched. The total switched capacitance is in both cases linear in N, but since in the generalized mixer only one C-element is involved per cycle, this implementation requires less energy per cycle. Also in terms of speed the generalization is a clear win. Another control component for which the multi-channel generalization can be implemented efficiently is the sequencer. Large sequencers can, for instance, be based on Gray-code counters. Various ways to implement generalized sequencers are discussed by Bailey and Josephs in [2].

The multiplexer is an example of a component in the datapath for which a multi-channel generalization can be implemented a lot more efficiently than a tree of basic (two-channel) components. Especially implementations with a control circuit based on the (multi-channel) mixer, in which the select lines return-to-zero between handshakes, generalize readily to multiple inputs.

An advantage of the multi-channel realizations of components is that the transparency of the Tangram compilation scheme generally improves. The three components discussed above, for instance, can be implemented such that the cycle times for all handshakes are (almost) equal. This means that the incurred delay only depends on the number of channels that are involved, which is directly related to the number of occurrences in the Tangram program. With a tree implementation, the cycle time depends on the position of the leaf in the tree, and thus also on the balancing of the tree.

6.3 Component substitution

Compilation from handshake circuits to single-rail standard-cell netlists is implemented as a component-by-component substitution process. The basis of this substitution process is the choice of a standard-cell library. We have chosen a generic standard-cell library as a starting point. Such a library contains (N)AND, (N)OR, X(N)OR, gates, various inverters and buffers, quite a few complex gates, latches and possibly some special functions (full adder, multiplexer, decoder, etc.).

The implementation of some of the data components is discussed in Chapter 5. Most components are straightforward and can be defined as library elements, whereas others are parameterized and require netlist generators. The implementation of the adder component, for instance, depends on width (in terms of number of bits) of the operands, and on their representation (signed or unsigned).

All-in-all, the netlist generation is manageable, since all handshake components obey a four-phase handshake protocol on all channels, and for each channel it is uniquely determined whether on that channel the active or the passive role is required. However, due to the simple component-by-component substitution process, netlist generation results in a netlist that possibly contains inefficiencies. In a post-optimization step these are eliminated.

6.3.1 Peephole optimization

Especially at the gate-level netlist, peephole optimization is of utmost importance. The building-block approach to the implementation of handshake circuits results in inefficiencies at the boundaries of handshake components. Three sources for peephole optimization are distinguished: the control-part of data components, the data-part of data components, and the control components.

Data components: control part

An important step in the single-rail netlist generation is the elimination of the abundantly present redundant C-elements and parallel delays in the control part of the datapath. A lot of pull datapath components that were discussed in Chapter 5 use both a fork and a C-element in the request-acknowledge path. Since these components are generally involved in the evaluation of expressions, and expressions do not contain input statements, the C-element is usually redundant, in the sense that it synchronizes two branches of a forked wire.

The elimination of redundant C-elements is illustrated in Fig. 6.6 (left). If a wire forks to a C-element, then this C-element is functionally redundant. Since C-elements (by choice) do not play a role in the delay-matching, such C-elements can

safely be removed from the netlist. The pattern shown is generic, that is, the number of delay elements in sequence on the top branch may vary from zero to any number.

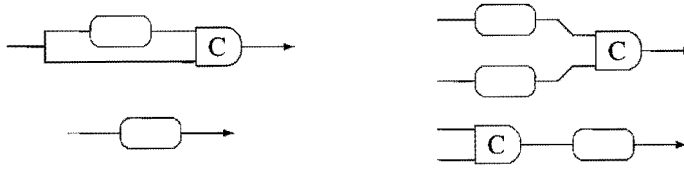


Figure 6.6: Two rules for the reduction of delay-matching and its associated control, namely redundant C-element elimination (left) and parallel-delay elimination (right).

Whereas the first rule reduces the number of C-elements, a second one is needed to reduce the number of (unit) delay elements. Such a rule is illustrated in Fig. 6.6 (right). Two parallel delays that lead to a C-element can also be placed after that C-element, since in terms of delay-management it does not matter whether the waiting takes place before or after the synchronization by the C-element.

The combination of the above two (generic) rules takes care of the automatic construction of the longest delay path and on the fly gets rid of parallel delay paths. The power of these rules is illustrated later in this chapter on several examples.

Data components: data part

Since Tangram offers only a limited set of basic operators and allows for the construction of other functions by programming, the data-part of the datapath is also an important source for improvement. Two issues can be distinguished here, namely operator reconstruction and technology mapping.

A simple example that illustrates the room for peephole optimization is Tangram expression `a*b*c`, in which `a`, `b`, and `c` are boolean variables. Since Tangram offers only the binary and-operator (`*`) the expression is compiled using two 'and' handshake components, each containing an AND gate. The composition of these two can of course be replaced by an implementation with a 3-input AND-gate. This type of optimization applies to other gates as well, and may be called *input extension* (though it essentially is a form of technology mapping). This optimization can be obtained by peephole optimization, or, alternatively, by introducing a 3-input 'and' handshake component, corresponding to a ternary operator.

There are, however, also optimizations that cannot easily be associated with multi-input components. The compilation of Tangram expression `a*b=c*d` (in which all operands are booleans), for instance, introduces two AND-gates and one XNOR-

gate. These AND-gates can be replaced by NAND-gates. This optimization by shifting inversions through gates is called *bubble shuffling*.

Another example is a decrementer, which is not a basic operator but can be constructed from a subtractor and a constant, as in `x-1`, in which x is an integer of, say, n bits. The Tangram compiler then generates a subtractor, with one n and one 1-bit operand. However, since the data-input of the 1-bit operand is constant (namely 1), the subtractor can be further optimized such that a true decrementer is obtained. Optimizations that involve constant inputs are known as *constant propagation*.

The optimization of the logic in the datapath is implemented as an incremental improvement step in the compilation, based on peephole optimization. Examples of peephole optimizations are constant propagation, bubble shuffling, and input extension, as shown above. This optimization step also involves so-called *technology mapping*, which is the link between the cell library that is assumed during netlist generation, and the larger standard-cell library that is available. Especially the availability of complex gates, such as AND-OR-INVerts, allows for some improvements in the netlist.

An important aspect of the logic optimization step is that the replacement circuits are always smaller than the original circuits, which is our primary concern. Secondly, the optimizations generally also improve the speed of the datapath, in the sense that they locally reduce the worst-case path. This means that after this step the delay matching can often be relaxed, in the sense that the delay-chain can be reduced in length. This issue is discussed later in more detail.

An alternative for the incremental improvement would be to use a logic synthesizer in combination with a technology mapper. For each node in the generated netlist its function can be derived from the cell that drives the node. A logic synthesizer could then optimize the combinational part of the netlist according to some criterion, for example logic depth, speed, or area. A combination of a powerful logic optimizer and a good technology mapper should be able to generate more efficient netlists than are obtained through the peephole strategy.

An experiment with Synergy™[1], however, suggests that the peephole approach is quite powerful. For the demonstrator that is discussed in Chapter 7, Synergy could not come up with a smaller netlist. Even worse, after flattening the expressions it turned out to be impossible to find a cost-equivalent solution, and the best approximation that was given was still 1–2% larger in terms of standard-cell area. This was probably (partly) due to the abundant number of XOR-gates in the netlist. Since logic optimizers internally generally use a sum-of-products or product-of-sums normal form, it may be hard to reconstruct XOR-gates after the optimization step, whereas during the peephole optimization step this is never a problem.

[1] Synergy is a logic optimizer from Synopsis

Control components

Although the focus in this thesis is the efficient implementation of datapath components, it should be noted that the control path of a handshake circuit can also be improved by a peephole optimization step. The effect of multi-channel components has already been discussed earlier. In this section we focus on improvements at the boundaries of neighboring handshake components.

An example of such a post-optimization is depicted in Fig. 6.7. The handshake circuit shows a fork that connects to two mixers. This might result from a Tangram program in which two shared procedures are called in parallel, such as in the last line in the fragment of a Tangram program shown below. The mixers organize multiple calls to the shared procedures, and the fork implements the parallel (synchronized) calls of the procedures.

```
begin
    procA : proc(). ...
  & procB : proc(). ...
|   ...
  ; procA()
    ...
  ; procB()
    ...
  ; procA() || procB()
end
```

Direct implementation of the handshake circuit, such as is obtained by substituting gate-level implementations for the components, results in a circuit with a subcircuit as shown in Fig. 6.7 (left). The request of channel a, denoted by a_r, is forked to two asymmetric C-elements, which originate from the mixers in the handshake circuit. The outputs of these asymmetric C-elements are combined in a standard Muller C-element.

This circuit exhibits a form of redundancy that is particularly hard to test. The path from a_r to a_a is a so-called *re-convergent path*, which means that the forking of a_r leads to a C-element that is functionally redundant. This can be resolved by replacing the circuit by the circuit shown in Fig. 6.7 (right). The redundant C-element is eliminated, such that synchronization with a_r in producing a_a takes place only once.

For other component interfaces that contain C-elements, similar optimization rules can be defined. Two-input C-elements can often be combined into multi-input C-elements (an optimization that was earlier referred to as input extension), some of which can be efficiently realized in a standard-cell library containing standard AND-OR-INVert gates and their duals.

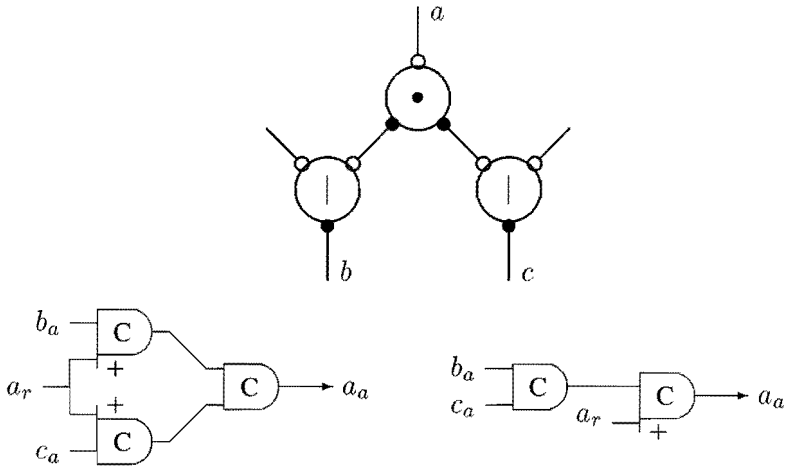


Figure 6.7: Fork-mixer optimization: handshake circuit (top), direct implementation of a subcircuit (left), and optimized subcircuit (right). The remaining C-elements can be further combined into one asymmetric three-input C-element.

Integrating C-elements, however, can be tricky. Although a C-element with three inputs can be decomposed into two two-input C-elements, the integration the other way around is not always possible. This depends on information about the behavior on the various inputs. Especially in a non-QDI environment, combining C-elements is not always allowed.

Implementation

The incremental netlist optimization step, based on peephole optimization, is implemented using a Philips in-house tool called VERA (VERification Assistant, [52, 53]). This is a rule-based pattern-matching algorithm based on LISP. Match patterns and their substitutes can easily be defined and added to the so-called *knowledge base*.

The actual netlist optimization amounts to applying rules in some predefined order. This order is important, both for the efficiency of the optimization step and for the costs of the final result. The first phase is the elimination of redundant C-elements and parallel delays. This step generally halves the number of elements in the netlist, which increases the efficiency of subsequent matching of patterns. Detailed technology mapping is performed as the last step, since this increases the number of library cells that are used.

6.4 Examples

In this section we illustrate the effect of the peephole optimization steps on pull datapaths, that is, on the implementation of Tangram expressions. These examples also illustrate the power of Tangram as a VLSI programming language. Tangram offers only a basic set of operators for data, but it has the expressive power to build other operators. The efficiency (in terms of area, speed) of these programmed implementations quite often is equal to what one would expect from a dedicated operator.

6.4.1 Parallel delays

The role of some of the peephole optimizations can be illustrated on the compilation of the Tangram expression `(x.0+x.1)<>(a-1)`, in which `a` is an integer, and `x` is a tuple of two integers. The compilation of this expression results in the handshake circuit shown in Fig. 6.8 (top). During a handshake on channel E, data is collected from handshake components 'x', 'a', and '1', and the expression is evaluated.

The infrastructure that is required to organize this data flow through the handshake channels is shown in Fig. 6.8 (middle). This is exactly the request-acknowledge circuitry that is generated during the component-by-component substitution process. In this control circuitry, all channels and components of the original handshake circuit can still be identified.

With the two peephole optimization rules of Fig. 6.6 this circuit can be greatly simplified. The C-element that corresponds to the split component (labeled $\rangle\langle$), for instance, receives its inputs from the adder, in which these are forked from one request signal. This C-element thus is redundant and can be removed from the netlist. Similarly, the inputs to the C-element in the adder are forked from one signal in the split. For the subtractor component, the inputs to the C-element are (indirectly) generated within the subtractor component itself.

So, by applying the rule for eliminating redundant C-elements one can already eliminate three C-elements, and only the right-most C-element remains. The inputs of this C-element, however, are both connected to a delay-element, and the rule for elimination of parallel delays can be applied.[2] After this the remaining C-element is also redundant and can be removed. What remains is a control circuit in which E_r is connected to E_a via a path of delay-elements only. This delay-line, by construction, matches the worst-case delay path of the expression.

The circuitry that is generated for the data part of the above handshake circuit is also simplified during peephole optimization. The netlist generator for the sub-

[2]The figure shows a simplified view on delays. Each delay-element represents a string of unit-delays, long enough to provide a worst-case match with its datapath.

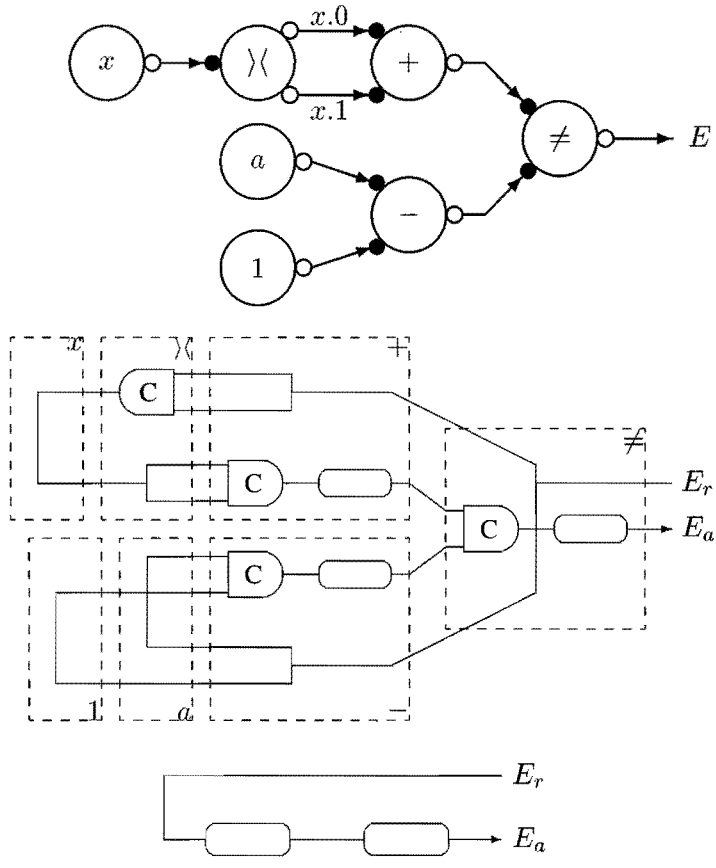


Figure 6.8: Post-optimization example showing handshake circuit (top), netlist for control circuit generated by direct substitution (middle), and post-optimized control circuit(bottom). The dashed boxes indicate the relation with the handshake circuit.

tractor takes advantage of the fact that '1' is an unsigned number and requires only one bit to represent. Therefore the subtractor is actually quite simple, and during peephole optimization only the low-order bit-section of the subtractor is affected. This may slightly reduce the worst-case delay of the subtractor, which means that the matching string of unit delays could possibly be shortened. This issue is further addressed in Section 6.5.

6.4.2 Galois-field arithmetic

The VLSI programming language Tangram offers only some basic operators as its starting set of operators. More complicated and dedicated functions can be pro-

grammed, in which bit-selection and type-casting are widely applied. An example of such a user-defined function is multiplying by *alpha*, where the operands and the result are from some Galois Field. Such a multiplication function is *gfalpha* of the Compact Disc Erco program (taken from [49]).

```
begin
  gfsym   = type <<bool,bool,bool,bool,bool,bool,bool,bool>>
& gfalpha = func (s: gfsym): gfsym.
            <<s.7,s.0,s.1<>s.7,s.2<>s.7,s.3<>s.7,s.4,s.5,s.6>>
& x,y : var gfsym
| ...
  y := gfalpha(x)
  ...
end
```

In this Tangram text, `gfsym` introduces eight-bit Galois Field symbols as a type, which is represented as a tuple of eight booleans. Function `gfalpha` multiplies `s` by *alpha*, basically by performing a shift-add step, where addition is modulo 2 (exclusive-or).

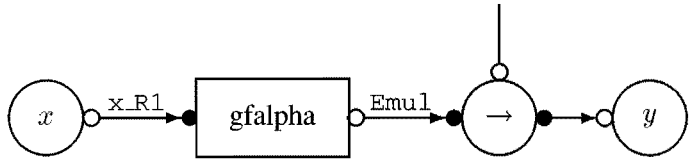


Figure 6.9: Handshake circuit

An invocation of this function, as in `y:=gfalpha(x)` (with `x` and `y` of type `gfsym`), results in considerable 'pull' handshake circuitry. The handshake circuit that corresponds to this assignment is shown in Fig. 6.9, in which we abstracted from the handshake circuit for function `gfalpha`. The textual representation of that handshake circuit is give in Fig. 6.10 (excerpt from compiled Tangram).

This combinational (pull) handshake circuit has two external pull channels. Channel `x_R1` is the readport of variable x, and the result of the expression appears on channel `Emul`. In the textual representation the external channels are listed in the header (between `( )`). The body (between `| [ ] |`) consists of a list of handshake components, in which each component has an associated type (between `[ ]`) and a list of handshake channels (between `( )`).

This handshake netlist contains seven split-components, which together implement the bit-selection of variable x. Channel `x_R1` connects to a readport of handshake variable x, and each split components splits off a one-bit channel. The bits are available on channels `s_2V0Q` through `s_2V7Q`. Bit 7 of this variable occurs

```
gfalpha (
    Emul PULL 8 PAS,
    x_R1 PULL 8 ACT
  ) =
  |[
    /* Substitution of function: gfalpha */
    SPLIT [PULL, 7, 1] (s_2_5, s_2V7Q, x_R1);
    SPLIT [PULL, 6, 1] (s_2_4, s_2V6Q, s_2_5);
    SPLIT [PULL, 5, 1] (s_2_3, s_2V5Q, s_2_4);
    SPLIT [PULL, 4, 1] (s_2_2, s_2V4Q, s_2_3);
    SPLIT [PULL, 3, 1] (s_2_1, s_2V3Q, s_2_2);
    SPLIT [PULL, 2, 1] (s_2_0, s_2V2Q, s_2_1);
    SPLIT [PULL, 1, 1] (s_2V0Q, s_2V1Q, s_2_0);
    JOIN [1] (s_2V7Ql, s_2V7Qr, s_2V7Q);
    JOIN [1] (s_2V7Qrl, s_2V7Qrr, s_2V7Qr);
    JOIN [1] (s_2V7Qrrl, s_2V7Qrrr, s_2V7Qrr);
    BIN [NEQ, 1, 1, 1] (Emul_rrl, s_2V1Q, s_2V7Qrl);
    BIN [NEQ, 1, 1, 1] (Emul_rrrl, s_2V2Q, s_2V7Qrrl);
    BIN [NEQ, 1, 1, 1] (Emul_rrrrl, s_2V3Q, s_2V7Qrrr);
    COMBINE [PULL, 1, 1] (Emul_rrrrrr, s_2V5Q, s_2V6Q);
    COMBINE [PULL, 1, 2] (Emul_rrrrr, s_2V4Q, Emul_rrrrrr);
    COMBINE [PULL, 1, 3] (Emul_rrrr, Emul_rrrrl, Emul_rrrrr);
    COMBINE [PULL, 1, 4] (Emul_rrr, Emul_rrrl, Emul_rrrr);
    COMBINE [PULL, 1, 5] (Emul_rr, Emul_rrl, Emul_rrr);
    COMBINE [PULL, 1, 6] (Emul_r, s_2V0Q, Emul_rr);
    COMBINE [PULL, 1, 7] (Emul, s_2V7Ql, Emul_r);
    /* End of function: gfalpha */
  ]|
```

Figure 6.10: Textual representation of compiled handshake circuit for *gfalpha*

four times in the expression, hence the three join-components connected to channel `s_2V7Q`. The function is basically computed by the three BIN components. The combine-components combine the result again in an eight-tuple, which is available on channel `Emul`.

Compilation of this handshake circuit into a gate netlist results in one C-element per handshake component (twenty in total), and three delay elements and XOR-gates, one for each bin-component. During peephole optimization, *all* C-elements and two delay elements are eliminated. What remains in the request-acknowledge part is a request wire connecting $\texttt{Emul}_r$ to $\texttt{x\_R1}_r$, and one delay element connecting $\texttt{x\_R1}_a$ to $\texttt{Emul}_a$. In the data part the three XOR gates remain. This resulting circuitry clearly is the optimum that would also be used in a dedicated GFALPHA component, would that be available in the library of handshake components.

6.4.3 Carry-select adder

Another illustrative example of the effect of peephole optimization is the carry-select adder of Section 2.3.4. The purpose of this adder is to provide a faster cycle time, which is achieved by splitting the 16-bit adder into two 8-bit parts. The carry-out of the least significant half can then be used to select either the high-order adder that assumed a zero carry-in or the one that assumed a one carry-in.

The control optimization of this carry-select adder is straightforward. The parallel delay paths of the two high-order adders can easily be merged, similarly for the parallel delays that correspond to the programmed AND-OR multiplexing. The result of this parallel-delay and redundant C-element elimination is a matched path that matches the sum of one 8-bit adder delay and the multiplexing delay.

The datapath logic that is generated for the carry-select adder is also improved during peephole optimization. First of all, the constants allow for simplification of the 9-bit high-order adder. Furthermore, the two high-order adders differ in their carry path, but since the two operands are the same, the full-adders can be merged, for example, by sharing the XORs in the sum paths of the full adders. A third improvement is achieved by combining the ANDs and ORs that implement the multiplexing of the high-order adders into AND-OR gates.

6.5 Fine tuning

The netlist that remains after the peephole optimization steps described in the previous sections still requires some attention before an actual layout can be made and sent to a foundry. Part of the work that still has to be done is related to the timing assumptions that have been made during the component-by-component substitution process. In Chapter 5 three delay assumptions are distinguished, namely, (i) isochronic fork, (ii) extended isochronic fork, and (iii) matched path. The most critical assumption in these is that of matched path. We first discuss the implications of this assumption, and then look into safety margins and the effect of peephole optimization.

6.5.1 Driver strategy

Translation from Tangram to handshake circuits is based on compositional rules. One might wonder why in such an approach, delay issues are not handled compositionally as well, such that after component substitution only peephole optimization would be required. The main reason for this is that on the component level one cannot predict the actual delay of a logical gate. In a first-order model, the delay of a gate can be modeled as the sum of an intrinsic delay and a resistive factor times the

capacitive load of the output of the gate. Input-slope dependencies and RC effects (interconnect resistance) are then ignored. At the level of handshake components the capacitive load is unknown, since both the fanout of the node and the required wiring cannot be predicted, but depend on the structure of the handshake circuit.

The compositionality is lost in components that fork data wires, such as fork, join, demultiplexer, and some arithmetic components. To maintain an exact delay-matching between the request-acknowledge path and the data one could insert buffers both in the data and the control part in all these situations, but that would be an overly pessimistic approach.

The approach followed in this thesis (especially in Chapter 5) is to ignore fanout issues during netlist generation and to solve this a posteriori. This requires a so-called *driver strategy* to be implemented after the peephole optimization step. A driver strategy is a procedure to resize transistors or to add drivers, such that timely transitions can be assured.

In the netlist that is obtained after peephole optimization one can, for each node, straightforwardly determine the cell that drives this node, the fanout of this node (the number of cells it connects to) and the total input load (capacitive loads of the inputs of these cells) of the node. The unknown factor before layout then remains the wiring capacitance, but this can be estimated from the fanout, based on layout statistics. This is sometimes called *forward annotation*, to contrast it with back annotation, which can be done once the layout is obtained and the actual circuit parameters are known. Given the capacitive load of a node one can determine the actual delay of the gate driving that node, and the associated duration of output transitions. These can then be adjusted by increasing or decreasing transistor sizes, or by adding additional drivers. The exact implementation of the driver strategy depends on the available repertoire of standard cells, which generally consists of standard gates (possibly in multiple driving strengths) and a range of inverters and buffers of different driving strengths.

The reason that such a driver strategy suffices to maintain a matched path relation between the request-acknowledge part and the data part is that, although the actual delay of a gate depends linearly on the capacitive load, the use of drivers reduces this to a logarithmic relation. More precisely, a wide range of capacitive loads (say, 0.1–10 picofarads) can be switched within small timing margins (say, 0.5–3 nanoseconds) assuming a sufficiently rich repertoire of drivers. For heavily loaded nodes in the datapath the matched properties should of course be verified. This issue is addressed later.

A driver strategy is not only required for proper delay matching, but also has a positive impact on the short-circuit power dissipation of the circuit. Timely transitions imply short periods in which both the path via n-transistors and the path via p-transistors are conducting [80]. In addition, the driver strategy results in overall

faster operation and thus has a positive effect on performance.

6.5.2 Tightening delays

The matched paths that are generated during the component substitution process are sometimes overly pessimistic in the sense that they add worst-case to worst-case. This leaves room for tightening the matched path by reducing the number of unit-delay elements.

Skew

Compilation of the Tangram expression `x+y+z`, in which the three operands are subranges of integers, results in a handshake circuit containing two adders, one for each + in the expression. In both adders the worst-case carry ripple is anticipated in the delay matching. Due to the organization of the handshakes, the total matched path of the combination of the adders is the linear composition of the two individual matched paths.

The question now arises how well this summed delay matches the worst case addition time of the combination of the two adders. The adders are both implemented using a straightforward ripple-carry adder. A feature of these adders is that the least significant bit is valid long before the most significant bit is valid. In general the data-validity of the bits is *skewed*, that is, the data-validity ripples from the low-order to the high-order bits. For an adder in isolation we then have to choose the latest data-validity. When the result of such an adder feeds into another adder, however, this second adder will also have this skew property. This means that the output of the second adder will be valid shortly after the output of the first adder is valid. The two carries more-or-less ripple in parallel, whereas the summed delay match anticipates a sequential rippling.

The skew problem is even worse in large combinational (parallel) multipliers. In a 16-by-16 multiplier with a 32-bit result, for instance, one could have 15 adders, each contributing 16 unit delays (240 in total), whereas the worst-case carry requires only about 32 unit delays. For the first example, with the two adders, the problem could be solved by introducing a multi-operand adder component, so that an appropriate matched path can be generated during the substitution process. For the multiplier, however, this approach would lead to a rather irregular handshake component, or dedicated multiplier handshake components would be required (which would make multiplication a Tangram primitive).

The cause of the skew problem is that we have chosen to use one data-valid signal per channel, rather than a data-valid signal per bit. A feasible alternative implementation strategy is to introduce a data-valid wire per bit, at least for pull com-

ponents. In the implementation of handshake variables we could then broadcast the data-valid signal to all bits, and the transferrers would have to determine the appropriate data-valid signal from the incoming data-valid signals per bit. This could be implemented using a tree of C-elements.

For most pull components the new approach would be straightforward. In adders and subtractors we could generate a delay chain to match the rippling of the carry. Per bit we could then detect whether the data-valid signal of the carry and that of the operands have arrived, and from this generate the data-valid signal of that bit. In the peephole-optimization process the longest path should then be selected and all other matching circuitry should be removed. For this the rules to eliminate parallel delays and redundant C-elements suffice.

Control overhead

A second source that possibly enables tighter delay matching is the control overhead. Consider the following Tangram fragment:

```
a?x
; z:= x+y
```

The delay-matching of expression `x+y` takes place during evaluation of that expression. The evaluation of the expression, however, starts as soon as `x` is updated, at least, given the implementation of the variable in Chapter 5, in which the value at all readports directly follows the state of the handshake variable. This means that the time spent in the sequencer (that implements the `;` of the program) could be used to shorten the delay of the adder. The contribution of the sequencer might be so small that it does not allow for the reduction of the matched path, but in general it may be that enough time is spend in the control to safely reduce some of the matched paths.

Peephole impact

During peephole optimization the logic in the data path is optimized. This operation may be called *function merging*, and also allows for tightening delays. An example to which this applies is the incrementer, in Tangram written as `+1`. Due to the constant input in the least-significant bit the adder can be simplified, and possibly the matched path can be shortened. Another example is the compilation of boolean expression `b*c+d`, which after component substitution results in an AND-gate, an OR-gate, and two delay elements. The AND- and OR-gate, however, can be combined into an AND-OR-gate, which is smaller, faster, and more energy efficient. This integration probably allows for the elimination of one delay-element.

Critical path

The above observations are all related to the fact that the matched path that is generated by the component substitution process is not affected during peephole optimization (apart from the elimination of parallel delays) and therefore may be overly pessimistic. After peephole optimization the actual critical paths should be identified, and the matched path should be adjusted to this.

Finding critical paths in netlists is computationally intensive, since critical paths are both load and data dependent. For the identification of critical paths timing analyzers can be exploited [22], which are tools that are common in synchronous circuit design. This issue is not further addressed in this thesis.

6.5.3 Isochronic forks

In the implementation of control components and in the implementation of the request-acknowledge part of the datapath (extended) isochronic forks are widely applied, for instance in the completion detection on high-fanout control wires, such as select lines in multiplexers and enable signals in variables. The implementation of these isochronic forks is an issue that must also be addressed after peephole optimization. The implementation of isochronic forks is well understood [7, 15]. The two main concerns are to bound the difference in logic-threshold voltages between gates that connect to an isochronic fork, and to bound the transition times on these isochronic forks.

The latter requirement is taken care of by the driver strategy, which precisely achieves this. With the insertion of drivers, steep transitions are guaranteed. This makes sure that the logical threshold voltages of different gates are passed at sufficient high speed to guarantee observation of the transition by different gates within a typical gate-delay.

The variation in logic thresholds of gates is kept minimal since only fully-static gates are used, and since the maximum stack height of n- or p-transistors is limited to three or four transistors in series. For the logic thresholds this limits the variation to at most 10% from the logic threshold of a standard inverter. The restriction on the variation in logic thresholds is thus guaranteed by the standard-cell library. (The limited variation also follows from the very small variation in switching thresholds of identical transistors on a single chip [66].)

6.5.4 Safety margins

The choice of a safety margin in delay matching is an important one. It is based on a trade-off between safety, performance, power, area, and verification effort.

One could choose to match the delays such that there is a 100% safety margin. In the context of the true four-phase protocol, this implies that delays in the request-acknowledge path should exactly match that of the datapath. Since the request-acknowledge path is traversed twice (both in the up- and the down-phase of the handshake) this effectively gives a safety margin of 100%.

Especially for long critical paths this is a safe approach that allows for considerable spread in environmental and processing conditions. For short paths, however, this approach may not be safe enough, since the margin may, in an absolute sense, still be small. Therefore, the safety margin that is used will in general have a multiplicative and an additive component. Matching of a delay of t_d nanoseconds can, for example, be done by a path of $(1+\alpha)*t_d+\beta$ nanoseconds, in which $\alpha > 0$ and $\beta > 0$. In this, α might be inversely proportional to t_d, to accommodate the fact that long paths are more likely to encounter both increasing and decreasing delay effects, and thus are likely to have a smaller relative spread than short paths.

The advantages of tighter delay matching are higher performance (shorter delays imply shorter cycle times and thus faster operation) and marginally smaller area (delays are implemented with gates, and smaller delays require fewer gates). An obvious disadvantage of tight matching is the greater verification effort that is required after layout. The matching always includes some uncertainty about the contribution of wiring to the delays, and after layout, when the wiring and the delays can be accurately characterized, the assumptions should be verified. With tight matching one is more likely to hit some more-than-average loaded nodes, which may affect the safety margins. Note that there should always be a safety margin to accommodate for performance variations due to variations in processing and operational conditions.

A possibly unexpected effect of tight delay-matching is that it may lead to more spurious transitions. In the true four-phase protocol, for instance, multiplexers are switched and latches are opened after about half the delay-matching has taken place. This means that from that point onwards the output of multiplexers and latches will follow the inputs. When these inputs are still changing, this possibly results in spurious transitions at the outputs that would not have occurred if a greater margin had been applied in the matching. So, tight matching can have a negative impact on the energy-efficiency of the datapath.

6.6 Placement and routing

After peephole optimization and fine tuning, the netlist can be placed and routed. This is a standard procedure, and therefore not thoroughly addressed here. One thing that should be noted, however, is that for large netlists, for example, consisting

of more than 100,000 gates, a step *after* placement is required, in which additional drivers are introduced.

For large circuits the wiring of some nodes may require long wires. Instead of over-dimensioning the drivers in the netlist to anticipate these exceptional cases, it is more attractive to adjust the driving strength of cells to the actual capacitive load that they face in the final layout. This capacitive load can be accurately estimated after placement of the cells, before actual routing has started. Even after routing it may be attractive to resize transistors, though in general this requires a liberal standard-cell library, which allows for variable transistor sizes.

Placement-driven optimization in combination with post-routing optimization of transistor sizes, is an interesting approach, both for higher performance and for better power management. Some ideas on this may be found in [60].

From the final layout the actual capacitive load of all nodes can easily be determined. This can then be used to back-annotate the netlist and start verification and accurate timing and power simulation.

6.7 Verification

After layout the wiring capacitances are known and it can be verified how realistic all timing assumptions have been, especially how safe the data-bundling is. The actual delay of the delay-matching chains can be determined straightforwardly. Determining the critical path of the combinational part is more complex, but can be done by a timing verifier [22, 41].

Rather than implementing the critical-path checks in the design flow, we have applied timing simulations based on accurate timing models of the standard cells. In these simulations the set-up and hold time constraints of the latches can be evaluated using various input scenarios. When there appears to be a safe margin, the silicon is likely to operate correctly under a reasonable range of operating conditions.

The verification process has been applied to multiple designs and uncovered no problems so far. To a large extent this is due to the conservative delay-matching that we have chosen. The design flow has also resulted in first-time right silicon, which is functional over a wide supply-voltage range, see Chapter 7 and [10]. This suggests that there is room for tighter delay-matching.

6.8 Conclusion

Some aspects of the single-rail design flow have been stressed in this chapter that are important for a *successful* route from Tangram to *efficient* single-rail silicon.

Especially the tight relation between simple and transparent compilation steps and peephole optimization is of utmost importance.

All compilation steps have been kept relatively simple and transparent. This can only lead to efficient implementation if sufficiently powerful peephole optimization is applied. The simple compilation steps and the representation at each level in a small set of primitives allow a straightforward and effective peephole optimization.

The implementation of the mixer component is a good example of where this approach pays off. Throughout the thesis the implementation with an OR-gate plus two asymmetric C-elements is used. At the gate level implementations exist that require less area. An advantage of our implementation, however, is that it straightforwardly generalizes to multi-channel implementations and in addition allows for peephole optimizations in which the asymmetric C-elements can quite often be eliminated.

Peephole optimization is an essential step in all phases of the design flow. The optimizations sketched in this chapter can all be characterized as *macho improvements*[3], which means that they lead to improvements in all cost and performance aspects. The primary concern in the optimizations has been to reduce area, but as a result of this the timing and energy consumption of the circuit also improves. Furthermore, even the testability is often improved, since structural redundancy (such as parallel equivalent gates and reconvergent paths) is eliminated from the netlist.

An important step in the design flow as discussed in this chapter is the netlist generation phase, in which each handshake component in a handshake circuit is replaced by a gate netlist, and subsequently the total gate netlist is improved via a peephole optimization step that is based on iterative improvement. This turned out to be a viable approach, both for the logic and the control in the datapath.

An alternative approach to iterative netlist improvement that is definitely worth pursuing is to employ logic synthesis tools. Datapath synthesis tools combine logic optimizers that manipulate logic expressions according to some performance criteria with powerful technology mappers that map these expressions onto some standard-cell library. Although these tools generally have computational problems with deep combinatorics (especially with XOR or adder dominated networks), they become increasingly powerful. Since at the level of handshake components these bottlenecks are already identified, the netlist generated from this level may be good starting point for a logic synthesizer.

One of the important trade-offs in the design flow is that of tight delay matching versus verification effort and performance. In the applications we had at hand, performance has never been the primary issue, but the emphasis has been on low power and area efficiency. When striving for highest possible performance, however, the

[3] term coined by Andrew Bailey

delay matching will have to be done as tightly as possible. This will come at the price of increased verification effort, and sometimes also at the price of more spurious transitions, and thus reduced energy efficiency. Moreover, it may in general lead to multiple place and route iterations, in which timing violations are removed in an iterative process. Another point that might deserve additional attention when striving for high performance is that of data-dependent completion times of computations. In a high-performance setting it may pay off to implement some of the components such that they take advantage of this data dependency.

Chapter 7

Demonstrator

7.1 Function

The choice of a demonstrator for the single-rail techniques presented in this thesis was highly motivated by the desire to compare it to alternative synchronous and asynchronous implementations of the same (or a comparable) function. Furthermore, for the demonstrator to be sufficiently convincing it should be of reasonable complexity and implement an industrially relevant (not self-defined) function.

Based on these motivations an error detector for the DCC player was chosen. This function was already realized as an asynchronous circuit, using double-rail encoding of the data [12]. Furthermore, information on several synchronous realizations of essentially the same function was also available.

The Digital Compact Cassette (DCC) player is an industrially relevant product (at least for Philips). It is a successor of the analog compact cassette system and allows for digital recording and playback of music (or other data) in digital format. The DCC system has furthermore been designed for backward compatibility, which means that it also plays analog cassettes. For a detailed overview of the system the reader is referred to [54, 45].

Since this thesis focuses on single-rail implementations, we take the Tangram program for the error detector as a starting point. Motivations behind the design and VLSI-programming insights can be found in [48]. The double-rail implementation of this function was first reported at the ISSCC, February 1994 [13], and later in some more detail in [11] (with emphasis on tools and design flow) and [12] (with emphasis on the asynchronous implementation).

The single-rail demonstrator has earlier been reported on at the Asynchronous Design Methodologies working conference in London, UK, May 1995 [10]. This chapter goes into somewhat more detail, and especially elaborates more on the design

steps that have been identified in Chapter 6. The importance of some peephole optimization steps is addressed.

Throughout the rest of this chapter the demonstrator will often be referred to as DDD, which is an acronym for *Dicy DCC Decoder*[1] (Dicy is the original name of the Tangram project).

7.2 Diagram

One of the motivations to choose the DDD as a vehicle was that a double-rail version was already available. This chip is part of a three-chip board comprising the Detector (DDD), an asynchronous (double-rail) Controller (DDR, for *Dicy DCC Remainder*[1]), and a commercially available 1Mbit DRAM. The board is used for error correction in an experimental DCC player, and can thus be used to test the chips in a working environment.

A diagram showing the context of the DDD is shown in Fig. 7.1. The chip-set is designed to operate in play mode only. Data from tape is first demodulated by the Channel Decoder, then corrected by the Error Corrector, and finally decompressed by the Source Decoder. A more detailed description of the function of the various modules is given in [12].

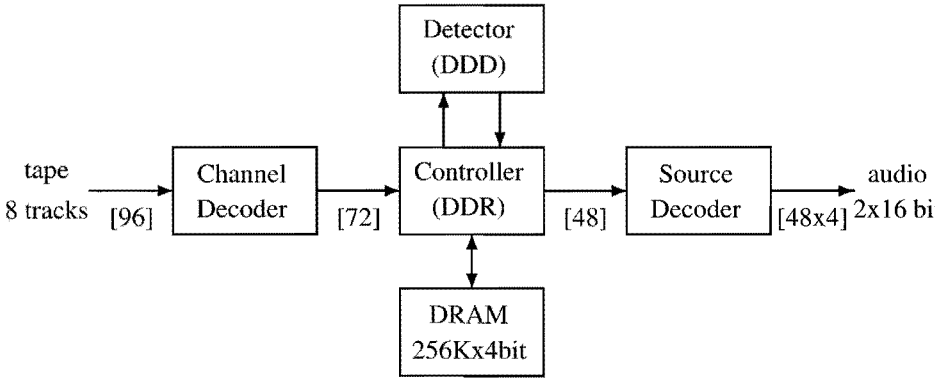


Figure 7.1: DCC codec in play mode [data rates in kilobytes per second]. The DDD, DDR, and DRAM together constitute the Error Corrector.

In order to be pin-compatible with the double-rail DDD the single-rail version has converters between single-rail and double-rail handshake channels. The diagram of the DDD with its double-rail interface is shown in Fig. 7.2. The converter

[1] coined by Kees van Berkel

elements, labeled D in the figure, convert between single and double-rail. Communication on channels t, c, l, and e is double-rail, and on channels T, C, L, and E single-rail. The converters have been made switchable, allowing the actual single-rail behavior also to be observed. This gives additional information about the on-chip behavior, which might be required if the circuit malfunctioned (which fortunately it did not).

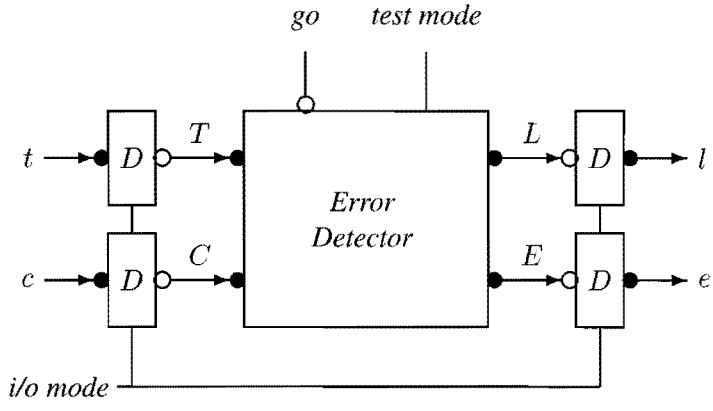


Figure 7.2: Single-rail error detector encapsulated with switchable conversion elements D from/to double-rail data-encoding.

The type information of the external channels can be understood from the header of the Tangram program for the DDD, which is given in the next section (Fig. 7.3). The implementation of the conversion elements is rather straightforward, and ideas can be found in [72]. Steven Vercauteren (IMEC, Belgium) proposed circuits for the converters using ASSASSIN [85], and a variant of these were actually implemented.

(One may observe that, from a cost (pin-count) viewpoint, it would have made more sense to add double-to-single rail converters to the original double-rail implementations. Also, the pins for channels C and E could have been made bidirectional so that they can be shared, since communications on these channels are mutually exclusive.)

The *go* channel in the diagram corresponds to the activation channel of the Tangram program. More specifically, it activates the repeater that implements the outer loop (`forever do ... od`) of the Tangram program. Since the repeater never issues an acknowledge on its passive port (or, alternatively, the Tangram program never completes) the acknowledge of the *go* channel can be eliminated. (After netlist generation this pin is tied low.)

Finally, a *test mode* signal has been added. The DDD contains a form of Tangram-

```
  gfsym = type <<bool,bool,bool,bool,bool,bool,bool,bool>>
& gfsymbool = type <<gfsym,bool>>  /* <<gfsym,erasure>> */
& int = type [0..31]
|
(T?<<bool,bool>> & C?gfsymbool & E!gfsym & L!int).
begin
  /* additional types, definitions, and declarations */
|
  forever
  do Syndromes()
   ; Euclid()
   ; Chien()
   ; Output()
  od
end
```

Figure 7.3: Structure of Tangram program for DDD.

programmed scan testing that is activated by making the test-mode signal high. This allows for shorter test sequences during functional test, since it allows short cuts in some of the iterations in the algorithm. Channels C and E are in test mode used as scan-in and scan-out channels. This kind of VLSI-programmed test has also been applied to the double-rail controller. A detailed report on this can be found in [69].

Throughout this chapter we focus on the single-rail core of Fig. 7.2, which excludes the conversion elements, but includes the (scan-) test facilities.

7.3 Tangram program

The signature of the Tangram program for the DDD is given in Fig. 7.3. This shows the typing and the direction of the external channels, and the main structure of the program. The program is explained in detail in [48].

The DDD is a Reed-Solomon type of decoder. At the beginning of a cycle it receives typing information via channel T (for Type). The type information determines whether the code word that has to be decoded consists of 24 or 32 bytes (called `gfsym` in the program), and how many parity symbols are used (4 or 6). The symbols are subsequently input sequentially via channel C (for Code word), in which each symbol is tagged with an erasure bit, indicating whether the channel decoder was successful. During reception of the code word the DDD on the fly computes whether the word contains an error. After this the number of errors is output via channel L (for Location). If the word contained errors (but not too many) then the

error patterns and the error locations are subsequently output via channel E (for Error) and L. In terms of a command the communication behavior of the DDD can be denoted by

$$*(T?\,;C?^{24|32}\,;L!\,;(L!\;\|\;E!)^{0\ldots6}).$$

The complete Tangram program for the DDD consists of 650 lines of code. About 120 of these concern the definition of Galois-Field arithmetic. This includes the definition of the types that are used, and the definition of operations on these types, such as addition, multiplication and division. Since the vast majority of the datapath logic in the DDD is in terms of Galois Field operations, the DDD program is somewhat atypical.

7.4 Handshake circuit

Translation from Tangram program to handshake circuit is performed in two steps: syntax-directed compilation followed by peephole optimization. In the peephole-optimization step of the compilation of the DDD, the rule for split-multiplexer reordering (as shown in Fig. 6.4) applies 11 times. This rule therefore accounts for the elimination of 11 multiplexer control circuits and 11 C-elements. Other reorderings in the handshake circuit could not be applied, since all possible sharing optimizations had been implemented in the Tangram program.

An important optimization step in the compilation from a Tangram program to a gate-level netlist is the combination of sequencers, mixers, multiplexers, and demultiplexers into multi-channel versions. Especially for mixers and multiplexers this allows for area-efficient implementations.

The handshake circuit that was initially generated by the Tangram compiler contained 2,009 handshake components. After reordering 1,987 components remained, and after the introduction of multi-channel components the handshake circuit for the DDD contained 1,768 handshake components. The breakdown into control and data, and a more detailed account of the distribution of handshake components are given in Table 7.1 (for control) and Table 7.2 (for data).

For the control it may be interesting to observe that the number of sequencers is about equal to the number of mixers. Furthermore, most parallel components are implemented with forks (synchronizers), which indicates that the Tangram program is highly sequential and the parallelism is limited to parallel assignments. Only 10 test components (called IF^t in [69]) were required to enhance the testability of the program (or, more precisely, the corresponding silicon).

In the data part, the number of variables and multiplexers are about equal, and the number of transferrers is about twice the number of multiplexers. The adder

Component	#	breakdown/comment
Sequencer	160	n 2 3 4 5 9 15 48
		# 37 12 6 3 1 1 1
Mixer	164	n 2 3 4 5 6 7 10 11
		# 30 17 3 9 3 2 2 1
Repeater	7	`forever do S od`
Parallel	1	
Fork	65	cheap parallel (synchronizer)
Join	7	
Run	11	`skip`, e.g. in if-then-else-fi construct
Duplicator	5	`for 2 do S od`
Case	15	`if G then St else Sf fi`
Do	16	`do G then S od`
Test	10	test component
Total	461	265 after multi-channel mixer/sequencer step

Table 7.1: Control components in DDD. For the sequencer and mixer both the number of 'binary' and multi-channel variants are given.

actually is an incrementer, and two out of four subtractors are decrementers. The datapath logic is dominated by XORs, which implement addition modulo 2 (used in Galois-Field arithmetic), and boolean ANDs (used in Galois-Field multiplication).

From the handshake circuit netlist an estimate of the contribution of the various handshake components to the standard-cell area can easily be generated. For the DDD this estimate (which does not take into account the peephole optimization at the netlist level) gives rise to the distribution shown in Table 7.3. In this list, redundant C-elements and delay elements have not been taken into account, since they are assumed to be removed (at least to a large extent) in the peephole optimization step.

7.5 Gate netlist

The netlist that is available after component substitution contains many redundant C-elements and parallel delays. Most redundant C-elements can already be predicted from the handshake component summary given in Table 7.2. For most pull data components it is known beforehand that they can be generated only by Tangram expressions, and that therefore the C-element will be redundant. This applies to the join, combine, split, and all binary operators, thus summing up to 1,285 re-

Component	#	breakdown/comment				
Variable	42	total: 410 bits				
Transferrer	76					
Multiplexer	54	before split-mux reordering				
	43	n	2	3	4	5
		#	8	8	5	1
Demultiplexer	4	n	2	4		
		#	1	1		
Boolean ops	430	type	inv	or	and	xor
		#	26	17	84	303
Integer ops	7	type	add	sub	less	adapt
		#	1	4	1	1
Split	320	pull type				
Split	37	push type, before split-mux reordering				
	26	after split-mux reordering				
Combine	261					
Constant	23					
Join	294					
Total	1526	1503 after multi-channel mux/dmx				

Table 7.2: Data components in DDD. For the (de)multiplexer both the number of 'binary' and multi-channel variants are given.

dundant C-elements. These are indeed all removed in the peephole optimization phase.

The number of unit-delay elements can also be estimated from the data components. Each binary boolean operator accounts for one delay element, which adds up to a total of 404 delay-elements. The adders and subtractors add 26 delay elements. The total number of delays therefore is 430. A lot of these elements turn out to be in parallel. After peephole optimization only 64 delay elements remain, which is 15% of the number of delays in the original netlist. This can be explained by the Galois-field arithmetic that has been programmed in Tangram. Some of the operations are specified at the bit-level, which means that at the word-level there may be many parallel delays. In an application that is dominated by adders, for instance, the percentage of parallel delays will be significantly lower.

Constant propagation is the elimination of constants (lows and highs) from the netlist by simplification of the gates to which these cells connect. These constants originate from two types of handshake components, namely constants and repeaters. These rules are rather trivial; so are the rules that eliminate cells with dangling

category	perc.	
control	36%	components of Table 7.1
communication	22%	(de)multiplexers, split
storage	23%	variables
logic	19%	datapath operators

Table 7.3: Area contribution of handshake components per category.

outputs from the netlist.

Bubble-shuffling optimizations are part of the technology-mapping phase of netlist optimization. The rule that allows to replace an AND-gate that feeds into an XOR-gate by a NAND and an XNOR applies 64 times in the DDD netlist, thus saving 128 transistors.

After the complete optimization step about 400 C-elements remain, both symmetric and asymmetric, and with two or three inputs. These C-elements are subsequently mapped onto combinations of complex gates and inverters. The exact mapping depends on the cells that are available in the library. A symmetric two-input C-element with input a and b and output z, for instance, can be realized by the sequential function $z = a * b + z * (a + b)$. This can be decomposed into a combinational gate and an inverter if either a negating majority with three inputs, or the function $-(a*b+c*d+e*f)$ is available. A decomposition into three gates is also possible, using $x = -(a*b)$, $y = -(z*(a+b))$, and $z = -(x*y)$. The first decomposition assumes an isochronic fork, the latter an extended isochronic fork, at the output of the C-element [15].

The last step in the netlist generation is the application of the driver strategy. For each gate output the expected capacitive load is estimated. If this exceeds the driving capability of the gate, that is, if it leads to too slow transitions, the driving strength is increased. For NOR, OR, AND, and NAND gates, this is achieved by replacing the gate by its inverse and adding a strong inverter. Inverters and buffers are simply replaced by stronger versions, and for other gates a buffer is inserted. For the DDD, the driver strategy adds some 300 transistors to the netlist, using unit-size transistors as unit. (A buffer of drive strength 8 contains 20 transistors, 2 parallel inverters for for the first stage, 8 for the second stage.)

7.5.1 Area breakdown

The total standard-cell area of the single-rail DDD is 1.47 mm^2 in a 0.8μ CMOS technology, which amounts to 4,783 gate equivalents, using the area of a two-input NAND-gate as a unit. The layout area, using double-layer metal, is 3.9 mm^2. A

photo of the layout is shown in Fig. 7.4.[2] This photo was also used in an article in the Scientific American of June 1995 [35].

An overview of the 3,364 standard cells that have been used is given in Table 7.4. The standard-cell dimensions are in terms of grids. One grid is 3.2 μm, the cell height is 10 grids, and the cell width varies.

The top two of the standard-cell list are latches and exclusive-ors. This should not come as a surprise since the datapath logic in the DDD is dominated by Galois Field arithmetic, which is to a large extent mapped onto XOR-gates.

The high inverter count may come as a surprise. Many inverters originate from C-elements, in which they implement the feedback to make the C-element static. The asymmetric C-element used in mixers, for instance, is implemented by the sequential function $z = a * (b + z)$, which is mapped onto an OAI-gate ($y = -a * (b + z)$) and an inverter ($z = -y$) that generates both the output and the feedback. This example also explains the high ranking of OAI-gate $z = -a * (b + c)$, which is used to implement the 270 asymmetric C-elements.

Two other high-ranking AOI-gates are $z = -(a * b + c * d)$ and $z = -(a * b + c * d + e * f)$, which are used in the implementation of (multi-channel) multiplexers. Some of the other AOI-types are reduced multiplexer cells that were originally fed by a constant. Two or three-input multiplexers also give rise to inverters.

Multi-channel sequencers are built from NAND and NOR-gates, and C-elements specified by $z = a * b * (c + z)$, in which a is used as a reset (this gives rise to an OAI and an inverter) [2].

[2]Both this layout (corresponding to SR2 in Table 7.5) and that of the single-rail IC (SR1 in the same table) were made by Gert-Jan Hekelaar and Arnold Gruijthuijsen of Microtel, for which they are gratefully acknowledged.

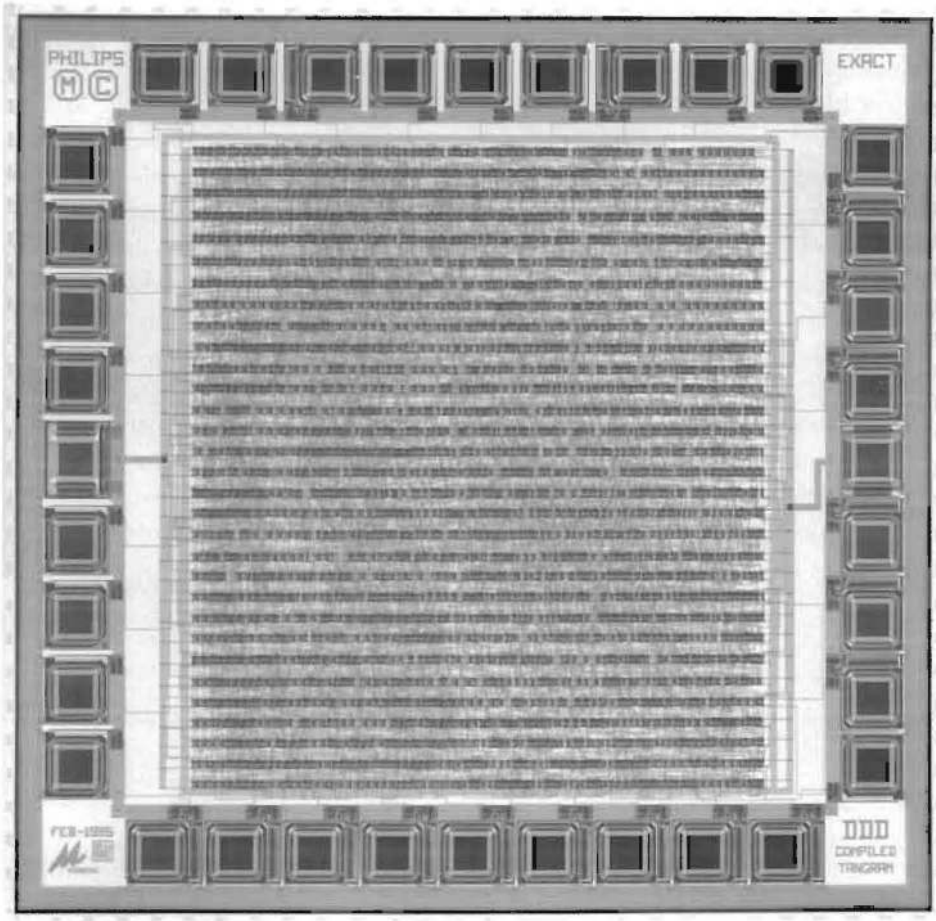


Figure 7.4: Layout of the single-rail DDD.

Function	#occ	#MOSts	(%)	#grids	(%)
$q = d * g + q * \overline{g}$	410	4,100	20.2	2,870	20.0
$z = (a \neq b)$	300	3,000	14.8	2,100	14.6
$z = -a$	735	1,470	7.2	1,470	10.2
$z = -(a * (b + c))$	346	2,076	10.2	1,384	9.6
$z = -(a * b)$	349	1,396	6.9	1,047	7.3
$z = -(a * b + c * d + e * f)$	114	1,368	6.7	912	6.4
$z = -(a + b)$	284	1,136	5.6	852	5.9
$z = -(a * b + c * d)$	170	1,360	6.7	850	5.9
$z = a * b$	112	672	3.3	448	3.1
$z = -(a + b * c)$	85	510	2.5	340	2.4
$z = -(a * b * (c + d))$	59	472	2.3	295	2.1
$z = a * b * c$	52	416	2.1	260	1.8
$z = -(a * b * c)$	60	360	1.8	240	1.7
$z = a + b$	60	360	1.8	240	1.7
$z = a\ (2)$	57	342	1.7	228	1.6
$z = -a\ (2)$	50	200	1.0	150	1.0
$z = (a = b)$	17	170	0.8	119	0.8
$z = -(a + b + c)$	29	174	0.9	116	0.8
$z = -(a * b + c + d)$	11	88	0.4	55	0.4
$z = a + b + c$	11	88	0.4	55	0.4
$z = a * b * c * d$	6	60	0.3	36	0.3
$z = -a\ (6)$	5	60	0.3	35	0.2
$d = (a = b \neq ci)$ $co = \mathrm{maj}(a, \overline{b}, ci)$	2	60	0.3	32	0.2
$z = -a\ (3)$	8	48	0.2	32	0.2
$z = -(a * b + c * d + e)$	5	50	0.2	30	0.2
$z = -a\ (4)$	6	48	0.2	30	0.2
$z = -((a * b + c) * d)$	6	48	0.2	30	0.2
$z = a + b + c + d$	4	40	0.2	24	0.2
$z = a\ (8)$	2	40	0.2	22	0.2
$z = -((a + b + c) * d)$	4	32	0.2	20	0.1
$z = -(a * b * c * d)$	3	24	0.1	15	0.1
$z = a\ (4)$	2	20	0.1	12	0.1
Total	3,364	20,288	100.0	14,349	100.0

Table 7.4: Standard-cell list of single-rail DDD, ranked according to the contribution to the total cell area. Inputs are labeled $a, b, \ldots$, output is z, except for latch and subtractor cells. The numbers between (.) denote relative drive strengths (for inverters and buffers).

7.5.2 Verification

Verification of netlists is always an important step in the design. For single-rail circuits, an essential aspect is the timing behavior. There are several modes of failure that can be envisioned. Most of the errors can be formulated in terms of latching the wrong data during assignments or communications.

First of all, latches may be enabled for a period that is too short to allow new data to be assimilated in the latches. In particular, this is a problem if the output of the latch is used in the feedback path, since then the output load of the latch directly affects the required pulse width on the latch enable. This type of problems can be expected when the environment of the variable is maximally fast, which is during a data-transfer in which no operations on data are involved, such as in a FIFO.

In the single-rail circuit some of these fast cycles were simulated at Spice level. The most critical scenario was very similar to the one shown in Fig. 5.16. The actual output load was 0.5pF, and the circuit was still fully functional with an output load of 1.3pF. At 1.5pF the circuit malfunctioned. This safety margin was considered sufficient to accommodate variations in processing or operation conditions.

The delay-matching was verified by varying the set-up times of the latches. For most of the latches it turned out that the input data was stable at 7ns before the actual closing of the latch. This was considered to be a sufficient safety margin. In general it may not be trivial to actually reconstruct the worst-case timing path for the logic in the datapath, since this may be highly data dependent. In the DDD, however, the maximum logic depth is less than 10, which simplifies finding the worst-case path.

7.6 Measurements

In June 1994 the netlist of a single-rail DDD, including test facilities and converters between single and double rail, was frozen. The resulting silicon arrived in November 1994 and proved fully functional, both on an HP82000 tester and in the experimental DCC player, in which it first played music on Wednesday, November 16, 1994.

Some interesting features of the error detector can be (and have been) tested in the experimental player. The error-correcting capacity of the code, for instance, is such that if one of the eight input tracks is corrupt, the other seven tracks contain sufficient information to reconstruct the missing information. In the player this feature has been tested by removing one of the tracks, thus forcing the DDD into its worst-case corners, with the most demanding signal-processing requirements. The player then indeed still produces audibly correct music, and a little extra stress (tapping the cassette), which makes the input uncorrectable, produces clicks (glitches) in the music.

Both the single-rail IC, and the double-rail IC that was reported earlier [13], were tested and characterized on an HP82000 tester. The results of the measurements for timing and power of these ICs are depicted in Fig. 7.5 and 7.6. In Table 7.5, which is presented in the next section, the circuits are referred to as SR1 and DR1. The measurements of the single-rail IC were reported earlier in [10] and received attention in an article in Electronic Design [55].

7.6.1 Speed

The measured execution frequencies are plotted in Fig. 7.5. In the measurements, the DCC-specified mix of 3,000 C1 words (length 24) and 2,3000 C2 words (length 32) per second has been used.

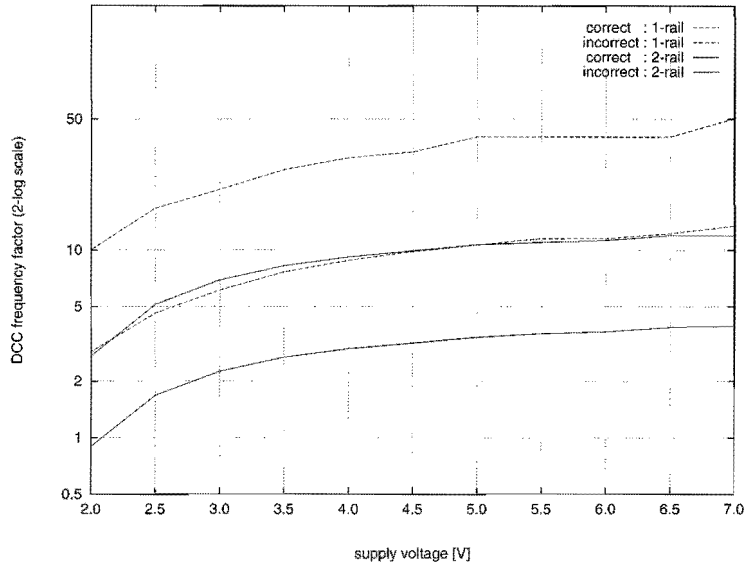


Figure 7.5: Measured execution frequencies versus supply voltage for correct and incorrect code words. The specified frequency for DCC application is used for normalization.

Both ICs are fully functional in the complete supply-voltage range from 1.2V to 7V. This indicates the robustness of these circuits. For a synchronous circuit one would need to adjust the clock frequency as the supply voltage is varied, to assure that the logic completes in time to meet the clock deadline. In the double-rail circuit this timing assumption has been replaced by that of the isochronic fork. Correct functioning is then guaranteed by the inherent completion detection on all double-rail handshake channels. In this way, the circuit naturally adjusts its speed to the

supply-voltage.

The robustness of the single-rail circuit may come as a surprise. In addition to the isochronic-fork assumption, the single-rail circuit uses matched paths to time the completion of datapath operations. The matched paths have been implemented with standard gates, just like the datapath operators. Furthermore, only fully static cells have been applied. The delay-elements have been implemented such that the timing dependencies on processing and operation conditions are similar to those of the operators in the datapath. Therefore these delay elements also slow down their operation as the supply voltage is lowered. Apparently, the safety-margins in the delay-matching are sufficient to guarantee correct operation over the complete supply-voltage range.

The single-rail IC still easily meets its performance constraints, with a safety factor of 2, at a supply voltage of 2V. At 5V, the decoder even has an excess in performance of a factor 10. One could thus operate the circuit at 2V, which gives at least a factor 9 reduction in power consumption. Alternatively, one could try to rewrite the Tangram program to one that requires less area, possibly at the cost of a higher power consumption.

The relative speed improvement of single-rail over double-rail is a factor 3. We estimate that a factor 2 is due to the differences in technology, layout, Tangram program, and control optimizations. The remaining factor of 1.5 then represents the actual single-rail contribution. This can be attributed to the absence of completion detection in the datapath, and the use of simpler cells (with fewer transistors in series) for the operations on data.

7.6.2 Power

The measured power dissipation of the single- and the double-rail ICs is shown in Fig. 7.6 for both correct and worst-case words. A logarithmic scale is used for power consumption. The DCC-specified mix of C1 and C2 words has been used, and the supply voltage was varied in steps of 0.5V. The measurements themselves are not too interesting. A comparison, however, leads to interesting observations.

First of all, one can observe that, for the single-rail IC, the ratio of power for incorrect versus correct words is close to 5. For the double-rail IC this ratio is only 3. The increased ratio for single-rail reflects an improvement in the underlying Tangram program, which was further optimized towards minimal activity for correct inputs.

A second observation that can be made is that the double-rail curves rise somewhat more steeply than the single-rail curves. This hints at a difference in the short-circuit contribution to the power consumption. We assume that the total switched capacitance per cycle is independent of the supply voltage. Given this, switching

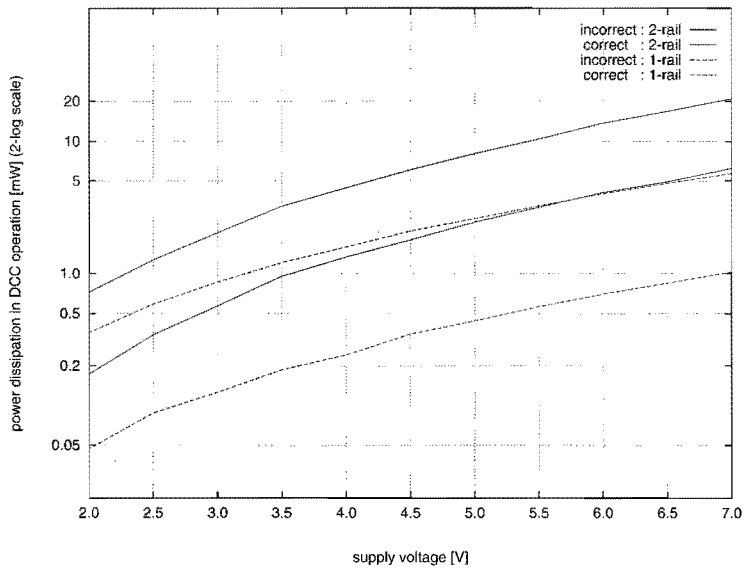


Figure 7.6: Measured power consumption versus supply voltage for correct and incorrect code words at DCC speed.

this capacitance results in a power consumption that is quadratic in the supply voltage. At a supply voltage of 2V the short-circuit dissipation is zero, which can be used to estimate the short-circuit power. The short-circuit power at 5V, as a fraction of the total power, can thus be estimated from the following formula.

$$1 - \left(\frac{5}{2}\right)^2 \times \frac{\text{power @ 2V}}{\text{power @ 5V}}$$

For the single-rail IC this indicates 15% short-circuit dissipation, for the double-rail IC it indicates 40%. The former is considered to be reasonable [80], whereas the latter is excessive, and is due to a poor driver strategy.

7.7 Evaluation

In Table 7.5 six circuit realizations of the DCC error detector are compared: two single-rail circuits, two double-rail circuits, and two synchronous circuits. The four asynchronous circuits have been obtained by compilation from a Tangram program. The synchronous circuits are part of existing ICs that are used in DCC products.

All circuits, except DR1, have been realized using the same standard-cell library, in a 0.8μ double-metal CMOS technology. DR1 was realized in a 1.0μ ded-

icated (asynchronous) standard-cell library. Data for DR1 have been scaled to the 0.8μ technology.

Data for the ICs have been measured at 5V, layouts have been simulated assuming nominal processing and typical operation conditions (5V, 25 degrees Celcius). Data for the IC-parts were kindly provided by Peters Arts and John Sherry (Philips Consumer Electronics). The cycle times in Table 7.5 are given only to compare the four asynchronous circuits. For the synchronous IC-parts this is not applicable.

Circuit DR1 is the chip that has been reported in [13]. Based on new insights the Tangram program was then changed a little, to further minimize the average-case activity (correct code words). From the new program three designs were realized, one single-rail chip with double-rail interfaces (SR1, reported in [10]), one single-rail layout (SR2), and one double-rail layout (DR2).

In the rest of this section we first discuss the various realizations in more detail, compare first and second generation of each technology (to illustrate the respective learning curves), and after this give a comparative evaluation of the three second generation circuits.

7.7.1 Single rail

Both single-rail circuits use the true four-phase protocol for data, and the broad four-phase protocol for control. Circuit SR1 contains single-to-double rail converters and can be put in the double-rail DCC test board. For sake of a better comparison with other (a)synchronous realizations of the same function we derived SR2, by stripping these converters (-400 MOSts). SR1 contained a power-supply switch [11], which was also removed (-150 MOSts). Additional peephole optimizations, not yet available at the time when SR1 was produced, lead to the saving of another 1,000 transistors.

The additional optimizations only explain part of the improvement of SR2 over SR1. Another factor is that the layout of SR2 is better SR1, due to more experience with the layout tool. The quality of the layout can, for instance, be measured by looking at the average wiring capacitance per node, which improved from 114fF for SR1 to 78fF for SR2. The ratio between core area and cell area improved from 2.8 to 2.6. This high ratio probably reflects the irregular structure of the DDD datapath, and the associated routing complexity.

The increased density of the layout results in better area, power, and timing results. In going from SR1 to SR2, the transistor count was reduced by 7%, and the core area by 13%. The total capacitance (transistor plus wiring) was reduced by 23%. To a first order approximation, this gain may be expected to be reflected as a gain in energy. The actual gain turns out to be larger, namely 30%. The optimizations (interfaces, peephole, and layout) also result in a 20% reduction of the cycle

		Single Rail		Double Rail		Synchronous	
		SR1	SR2	DR1	DR2	Sync1	Sync2
Quantity	unit	IC	layout	IC	layout	IC-part	IC-part
Area							
#transistors	1,000	21.8	20.3	44.0	30.8		
Cell area	mm^2	1.6	1.5	3.5	2.2		
Core area	mm^2	4.5	3.9	7.0	5.9	3.4	3.3
Cycle time							
C1, correct	μs	8.0	5.9	14.0	8.1		
C1, worst case	μs	26.8	23.2	38.4	31.1		
C2, correct	μs	8.7	7.3	14.4	10.3		
C2, worst case	μs	38.1	31.3	51.2	42.0		
DCC mix	ms	50	40	83	55		
Energy							
C1, correct	μJ	0.08	0.05	0.37	0.10	2.0	0.4
C1, worst case	μJ	0.36	0.31	1.12	0.63	2.4	0.8
C2, correct	μJ	0.08	0.06	0.41	0.15	2.7	0.6
C2, worst case	μJ	0.55	0.42	1.50	0.88	3.1	1.1
Power							
DCC mix	mW	0.50	0.35	2.3	0.80	12	2.7

Table 7.5: Comparison of six implementations of the error detector. Cycle time and energy are measured at 5V. The DCC mix comprises 3,000 C1 and 2,300 C2 codewords per second, 95% correct and 5% worst case.

time.

7.7.2 Double rail

The double-rail circuits both use the broad four-phase protocol for control and the four-phase double-rail protocol for data. The difference between DR1 and DR2 are the timing assumptions that have been made, the control optimizations that have been applied, and the cell library.

Circuit DR1 is fully QDI and is implemented in a dedicated standard-cell library which includes several asynchronous cells. To keep layout costs reasonable, special cells for double-rail arithmetic and logic were required. Layout DR2 is realized in a generic standard-cell library and, therefore, required extended isochronic forks [15]. Furthermore, the same control optimizations as for SR2 have been applied. The cells in the dedicated cell library used for DR1 are rather dense (in terms of

transistors per area), which explains the relatively small area reduction compared to the reduction in transistor count.

A further difference between DR1 and DR2 is the Tangram program (and handshake circuit) that was used as a starting point. A slight modification of the program for DR1 allowed a reduction in the activity for correct code words by about 50%, at the cost of only a small area increase (3%). Furthermore, the short-circuit dissipation was reduced by a better driver strategy (the same driver strategy as for SR1 and SR2).

In comparison to DR1, DR2 is 35% faster, 15% smaller, and requires only 35% of the energy.

7.7.3 Synchronous

Circuit Sync1 is part of a synchronous IC in an early-generation DCC player. It operates on a 6.14MHz clock, uses a ROM-based centralized controller and a small RAM to store intermediate results. The ROM and RAM together account for more than 50% of the power dissipation.

Sync2 is the successor of Sync1 and is optimized towards low power consumption. Architectural modifications allowed the halving of the clock frequency and elimination of the RAM. Furthermore, by means of clock gating (in the enabling of the ROM), power could be reduced further. The latter also decreased the power ratio for best over worst case. The contribution of the ROM to the power consumption was reduced to 6–10%.

7.7.4 Comparison

The data collected above allow for an interesting comparison for area and power. Furthermore the speed of single- and double-rail can be compared.

The main motivations for the single-rail work were area reduction and mapping onto a generic cell library. The original double-rail circuit had a 100% area overhead over the synchronous circuit that was available at that time (Sync1). Meanwhile, both the synchronous and the double-rail implementation have been improved, in which the double-rail overhead was reduced to 80%. The single-rail circuit, in contrast, has an area overhead of 20%, which is a significant reduction, but cannot be neglected.

Power efficiency is an important motivation for the work on asynchronous circuits and has also gained increased attention in synchronous designs. Interestingly, the factor 5 advantage in power that was reported in [13] for double-rail circuit DR1 over synchronous circuit Sync1 recurs for the power efficiency of the asynchronous

DDD	Sync	SR	DR
Area	100%	120%	180%
Energy	100%	15%	30%
Speed		100%	75%

Table 7.6: A simplified comparison of the best synchronous, single-rail, and double-rail realizations of the DCC error detector.

successor of DR1 (single-rail circuit SR1) over Sync2, the synchronous successor of Sync1.

In conclusion, the single-rail circuit is 20% larger than the synchronous version, but uses only 15% of the power. Compared to the best double-rail version it is 33% smaller, 25% faster, and requires 50% less energy. A simplistic summary of the main characteristics is shown in Table 7.6.

Testing has not been addressed in the above evaluation. For the synchronous circuits, test hardware was not taken into account. At the cost of about 3–5% (area and energy) these circuits can be made fully scan testable. The two single-rail circuits include partial scan-test facilities, accounting for 4% of the circuit area. With these facilities, both circuits are fully testable against stuck-at input faults. The double-rail IC is fully tested against stuck-at faults in a functional sequence involving four code words (giving rise to four complete algorithmic cycles). The testability of DR2 has not been analyzed in detail.

On an HP82000 tester, the complete test of the single-rail IC (SR1) takes 1.15ms (including scan in/out). The complete test for the double-rail IC (DR1) requires 2ms.

7.8 Conclusion

The single-rail DCC error detector demonstrated the viability of the single-rail implementation of handshake circuits in a generic standard-cell library. The power advantage of a factor 6 over the synchronous equivalent was sufficient to justify the 20% area overhead.

The delay-matching in the single-rail DDD has been implemented straightforwardly and conservatively, to maximize the chance to get a successful first-time-right single-rail chip. The robustness of the delay-matching was demonstrated by the correct operation over the complete supply-voltage range from 1.2V to 7V.

Chapter 8

Conclusion

The target that was set for the work described in this thesis was to reduce the silicon area of handshake-circuit implementations greatly, while restricting oneself to a generic standard-cell library. Furthermore, the low-power properties should not be lost and possibly be improved, and the design flow should be highly automated, with manageable verification effort after layout.

Throughout the previous chapters single-rail implementation of handshake circuits has been put forward as a way to achieve exactly these goals. The potential success of this approach has been demonstrated on a DCC error detector chip. This demonstrator has an interesting power advantage, at the cost of an acceptable area overhead, and was realized in a generic ('synchronous') standard-cell library.

In this final chapter we first compare single-rail circuits with other implementation techniques, both synchronous and asynchronous. Next we evaluate the strengths and weaknesses of single-rail handshake circuits. This leads to an inventory of the opportunities and threats for single-rail handshake circuits. The chapter is concluded with some thoughts on possible future directions.

8.1 Comparison

In this section we address the quality of single-rail handshake circuits, and we do that by comparing them to circuits realized via other design flows. Since in the Tangram project both a single-rail and a double-rail design flow from handshake circuits have been implemented, we have learned quite a lot about their relative merits. Therefore, this comparison is rather detailed and extensive.

We have chosen to use a four-phase protocol in combination with single-rail data-encoding. A different option would have been to use a two-phase handshake protocol. The disadvantages of two-phase implementation of single-rail handshake

circuits are also addressed in the comparison. Furthermore, brief comparisons with micropipelines and synchronous circuits are made.

8.1.1 Four-phase double-rail handshake circuits

The main difference between single and double-rail handshake circuits is the area efficiency of the resulting circuits, but the timing, power, and testability characteristics of these two also differ.

A first insight into the difference between single rail and double rail can be gained by looking only at the handshake channels, and ignoring the contribution of the handshake components. For a handshake channel, the number of wires can be used as a measure for area, and the number of transitions on these wires (per handshake) as a measure of energy.

For the double-rail channel $2N+1$ wires are used, whereas the single-rail variant only requires $N+2$ wires. For a 16-bit datapath this hints at an area-ratio of double rail over single rail of 1.8. If we look at the number of transitions per handshake, then a double-rail handshake requires exactly $2N+2$ transitions, and a single-rail handshake $\frac{1}{2}N+4$ (assuming uncorrelated data and no spurious transitions). A 16-bit double-rail handshake thus consumes 2.8 times the energy of the equivalent single-rail handshake.

From the above observations, we may expect single-rail circuits to be equally fast at the cost of a little more than half the area and only a third of the power of double-rail circuits. This comparison is rather naive, and is carried out in more detail in the rest of this section.

Standard-cell libraries

The motivation to choose double-rail encoding of data generally is to adhere to a QDI implementation style in the circuit. In combination with a generic standard-cell library this leads to a high area price that has to be paid.

One well-known work-around to get rid of this overhead is to use a dedicated standard-cell library, in which special double-rail operators are included. Circuit techniques based on weak feedbacks or partially dynamic realizations can then be exploited to reduce the area-inefficiency of fully static implementations [58, 73, 39].

Another way to partially eliminate the double-rail overhead is to relax the QDI timing assumption, and to allow for extended isochronic forks. A generic standard-cell library can then be used with little overhead over a dedicated library [15].

Single-rail handshake circuits can be realized efficiently in generic standard-cell libraries, as has been demonstrated in this thesis.

Speed

The most simplistic comparison of the speed of single-rail and double-rail circuits is obtained by looking only at the handshake protocol on the channels. Since they are both based on a four-phase protocol, a handshake on both channels requires four handshake events. So, from this simple viewpoint, single-rail and double-rail circuits are equally fast.

Several observations can be made, however, that lead to the conclusion that single-rail realizations are faster than their double-rail equivalents.

If we zoom in a bit further on the handshake channels, and look at the sequences of events, we observe that, on a single-rail channel, the timing (and therefore the speed) is determined by the four transitions that are required for a complete handshake on the request-acknowledge pair. On a double-rail handshake channel the slowest bit determines the length of a cycle. Completion detection is required to determine whether a complete message has arrived. This is typically implemented using a tree of C-elements, for instance in a double-rail variable. This completion detection introduces a timing overhead for double-rail circuits.

Another issue that has to be taken into account is the processing time for operations on data. Typically, a double-rail implementation of a function requires complex gates with stacks of pMOS and nMOS transistors that are higher than required for the equivalent single-rail implementation. This makes the gates slower and — maybe more important— reduces the possibilities for peephole optimization. Furthermore, the fanout of cells in a double-rail datapath is, on average, also higher. As a consequence, double-rail processing is generally slower than the equivalent single-rail processing.

In double-rail operations, the return-to-zero phase is redundant. If this is not implemented as a quick reset (for which extended isochronic forks can be used [15]), this time gives an additional overhead for double-rail timing. This effect can be canceled against the safety margin that has to be applied in the delay-matching in the single-rail datapath. Another way to circumvent the redundancy of the return-to-zero phase is to employ the *lazy active* protocol, as proposed by Martin [57].

A potential speed advantage of double-rail datapaths is the data dependency of timing. A double-rail adder, for instance, can easily be realized such that in each bit section of the adder the carry out is generated as soon as it can be determined [72, 58]. In the single-rail implementations that we have chosen and implemented, data-dependent processing times have not been exploited. Instead we have chosen to use a fixed matched delay that always accommodates the worst-case completion time of the datapath. For an incrementer, for instance, we always anticipate the full length carry ripple, although this is known to occur very rarely, and on average the carry ripples only two stages.

From the above observations, and from our experience, we conclude that single-rail datapaths are on average clearly faster than double-rail equivalents. This does not apply to wide, stand-alone adders or large combinational multipliers, for which the average processing time may easily be faster than the worst-case single-rail delay. On the other hand, however, if no operations on data are required (as in a FIFO) the completion detection required in the double-rail circuit becomes a really serious overhead.

The single-rail demonstrator that we have built turned out to be some 30% faster than a double-rail variant, in the sense that it performs the same function in only 70% of the time. It is hard to quantify the speed-potential of single-rail in comparison with double-rail, although from the discussion above a performance advantage may generally be expected. Based on the single-rail demonstrator, a rough estimate for the speed-advantage of single-rail over double-rail is that it is in the range of 20 to 40%. It should be noted that the advantage depends on the margins that are used for delay-matching, on the layout style, and on the design effort.

Energy

For double-rail handshake channels the number of transitions per communication is determined by the protocol. Each (four-phase) communication on an N-bit channel results in exactly $2N + 2$ transitions, independent of the data that is actually communicated. On a single-rail channel the number of transitions depends on the Hamming distance between the current data and the previous data. On average, assuming uncorrelated data, one would expect $\frac{1}{2}N + 4$ per communication. This suggests that a single-rail implementation uses only about a third (for practical N) of the energy of a double-rail implementation.

However, some countereffects exist in single-rail datapaths. Most notably, we did not restrict the transitions on the data wires in the period in which the data was *not* valid (data-change period). In the true four-phase scheme —which has actually been implemented— the state of latches is updated when a variable is written. All handshake components that read from this variable are therefore faced with changing inputs. Especially for arithmetic (adders etc.) this implies that they start using power as well, since the operators are permanently evaluated. This may have a considerable impact on the energy consumption, because the read ports may fanout to many logic gates. Apart from the sum of input capacitances of these gates that has to be switched, the output of these gates may also be affected. Especially in deep combinational logic (for example in multipliers), this can lead to a lot of redundant transitions.

Another effect that must be taken into account is that latches are made transparent during the up-going phase of the write handshake, and closed again during

the down-going phase. In between, the latches are transparent and the latch outputs follow the latch inputs. If the data was not yet valid at the input when the latch was opened, this leads to spurious transitions, which also reduces the energy efficiency.

Since double-rail combinatorics generally require more complex cells than the equivalent single-rail functions, double-rail datapaths typically have higher average fanout and higher average transistor stacks than single-rail datapaths. This effect works in favor of single-rail energy efficiency.

From the above observations and from practical experience (for instance, measurements on the DCC demonstrator) we estimate that a single-rail circuit consumes half the energy of its double-rail counterpart.

Area

In contrast to energy and timing, circuit area is relatively easy to compare. A first order insight can already be gained by looking at handshake channels. A single-rail channel for N bits requires $N + 2$ wires, against $2N + 1$ wires for the double-rail circuit. Since every wire requires a gate to drive it, one may expect the standard-cell ratio (d.r./s.r.) to be somewhat less than 2. This estimate is actually quite accurate, though in the datapath effects in favor of both single-rail and double-rail exist.

Double-rail operations on data require gates that are more complex than those used for single-rail operations, especially for simple boolean functions like AND, OR, and XOR, but also for adders [58], although these can sometimes be simplified somewhat by turning the return-to-zero phase into a quick reset [15]. All in all, where the single-rail datapath is relatively simple, the double-rail operators quite often are state-holding (to assure adherence to the handshake protocol), and require standard cells with high nMOS and pMOS stacks.

For multiplexers —which are frequently used components in Tangram handshake circuits— the single-rail and double-rail implementations are of comparable cost. A double-rail multiplexer requires two OR-gates per bit, whereas a single-rail multiplexer requires a complex gate of about the same size, both in transistors and cell area.

Completion detection in double-rail datapaths and delay-matching in single-rail datapaths both require circuitry. These two contributions are hard to compare quantitatively. The circuit area required for completion detection is linear in the width of the datapath to which this is applied. Delay-matching, in contrast, depends linear on the logic depth of the function that is computed.

The area of a single-rail circuit on average turns out to be 60% of that of the double-rail equivalent. To underscore this, and to illustrate the contribution of the control part, several Tangram programs have been compiled to netlists. For single-rail the generic library is used, whereas for double-rail a dedicated library is applied.

(With the extended isochronic fork assumption the double-rail circuits can be realized in the same silicon area using generic cells only, cf. [15].) The results are listed in Table 8.1.

	single rail		ratio single/double rail	
Design	#MOSts	control	#MOSts	cell area
FIFO (32b, 8st)	3.0k	14%	0.49	0.52
DCC detector	20.3k	36%	0.48	0.60
DCC controller	78.1k	54%	0.78	0.90
Speech codec	38.6k	21%	0.37	0.49
Router	28.5k	45%	0.58	0.70
average	33.7k	34%	0.54	0.64

Table 8.1: Comparison of single-double rail, cell area and transistor count. The double-rail numbers are based on a dedicated cell library, the single-rail numbers refer to implementations using a generic library.

As is to be expected, the gain in circuit area and transistor count depends on the percentage of the circuit required for control. Data-dominated applications, such as the FIFO, the DCC error detector, and the speech codec, profit highly from the single-rail implementation of the data. The DCC controller and the router are control dominated, so single-rail and double-rail make less of a difference.

Test

QDI circuits and (as a consequence of this) double-rail data encoding have often been praised for their good prospects for testability with respect to stuck-at faults [4, 40]. Indeed, on a double-rail channel a stuck-at fault implies that either during the up-phase or the down-phase of the handshake protocol, the sender does not cooperate and therefore stalls the receiver. This deadlock can be detected by a timeout, since as a result of the deadlock, the circuit does not meet predetermined response times.

However, it depends on the implementation of the components how well this simple scenario can be followed down to the gate level. The observability of stuck-at faults, for instance, highly depends on the acknowledge property of the double-rail operators, which is that every input transition is eventually followed by an output transition. At the gate-level, however, this property may be lost if the operators have to be implemented using a generic standard-cell library [15]. Since a dedicated standard-cell library is not a viable option (to us), this means that test hardware has to be added to make the design testable.

	single rail	double rail
timing	data independent	data dependent
energy	data dependent	data independent

Table 8.2: Data dependencies in single-rail and double-rail handshake circuits.

Testing single-rail datapaths may require some form of scan to enhance the observability of faults in the logic of the datapath. The application of these techniques to double-rail datapaths is described in [69]. To facilitate test-generation for single-rail circuits a fault simulator has been developed [79]. In this simulator, faults are modeled at the handshake circuit level, which allows for high-performance simulation and accurate feedback at the Tangram level. This enables the interactive design of test patterns for single-rail datapaths. In combination with scan facilities this approach has proven a very powerful means to achieve high-coverage test patterns.

Summary

From the above discussion we estimate that double-rail circuits are about 75% larger than single-rail circuits, consume twice the energy, and are some 25% slower.

An interesting insight that applies to the single-rail and double-rail handshake circuits as they are compared here, and as they are implemented in the Tangram project, is given in Table 8.2, in which we look at the data dependencies of the time and energy required for a computation. In a single-rail Tangram handshake circuit the timing in the datapath is data-independent and the energy consumption data-dependent. In the double-rail implementation this is the other way around.

8.1.2 Two-phase single-rail handshake circuits

Our single-rail handshake circuits are based on a four-phase single-rail (bundled-data) protocol. Many implementations of bundled data circuits, however, use a two-phase signaling protocol [33, 63, 75]. The use of such a two-phase protocol at first sight leads to faster and more power-efficient circuits, since only half of the control transitions are required. However, in the implementation of some of the essential handshake components it leads to complications; most notably in the variable and in components that implement sharing (mixers, multiplexers).

In the two-phase implementation of the variable, a latch-control circuit is needed that requires Toggles and Merges (XORs) to convert from the two-phase (transition-sensitive) write handshake to the four-phase (level-sensitive) enable signals. The CMOS implementation of the Toggle is complex [63]. An important consequence

of the use of Toggles and Merges is that they are in the critical timing path of the control, which leads to poor cycle times, and hence restricts the maximum attainable speed. The use of capture-pass latches [75] would allow for a simpler latch-control circuit, but then the latches are slow and require more transistors.

Sharing of datapaths and control paths is an important issue in Tangram handshake circuits. In a typical Tangram application, a significant fraction of the handshake components consists of mixers and multiplexers. The two-phase implementation of the mixer is known as the *call*-component [75]. The CMOS implementation of this is quite complex (see for instance [63]) since internally it essentially requires a conversion to four-phase signals. The two-phase multiplexer control circuit is based on this two-phase call component and also requires this conversion to four-phase signals for the control of the level-sensitive multiplexer switches. Especially for multi-input mixers and multiplexers, two-phase realizations are bulky. Compared to four-phase realizations they are significantly larger, have a longer cycle-time, and use more energy per cycle.

Another component with a relatively complex two-phase implementation is the passivator, a component that is used in input-output communication. In a two-phase implementation of a passivator the data has to be latched during each handshake, whereas in the true four-phase protocol we can do without latching.

The implementation of datapath operators (components that operate on data, such as adders and Boolean operators) is essentially the same in two and four-phase. The symmetric implementation of delays was traditionally claimed as an advantage of two-phase over four-phase datapath operators. With the true four-phase protocol that is applied in our single-rail circuits, however, four-phase delays can also implemented symmetrically.

A complication in two-phase implementations of handshake components is that for a lot of components self-initializable realizations do not exist. In all cases where Toggles or call components are required, dedicated reset hardware is needed to force the circuits into well-defined initial states. This reset circuitry has a negative impact on the speed, power, and area characteristics of the resulting circuits.

In conclusion, two-phase handshaking does not seem to be an attractive alternative for handshake circuits. Two-phase handshaking may appear to be more efficient at the handshake channels, but some of the essential handshake components become rather complex, which sweeps away any potential advantage.

On a speculative note, single-track handshaking (cf. Section 3.5) may be an alternative two-phase solution with interesting speed and energy prospects [9]. The current implementation, however, requires a dedicated standard-cell library, which makes it hard to apply to practical circuits directly, given the tendency towards generic standard-cell libraries. The testability of this implementation style is also still unclear.

8.1.3 Single-rail micropipelines

Micropipelines were introduced by Sutherland in his Turing Award lecture [75], in which he assigned the name to 'a particular simple form of event-driven elastic pipeline with or without internal processing.' Single-rail handshake signaling is used to communicate between pipeline stages and the interface between stages consists of a push handshake channel.

Micropipelines, and especially two-phase (transition signaling) realizations of them, have become rather popular. They have, for instance, been used in AMULET1, a micropipelined ARM processor designed in Manchester [32, 33, 31, 63].

Pipelines can straightforwardly be programmed in Tangram. There are some properties of pipelines that make them an interesting subclass of handshake circuits, most notably the increased freedom in the choice of latch-control circuits. In the rest of this section we first introduce the Tangram equivalent of a micropipeline, and then compare the compiled circuit with dedicated micropipeline implementations that are known from literature.

Tangram equivalent

An example of a Tangram program for a (micro)pipeline is shown below. The Tangram program consists of three stages. The first stage includes internal processing logic, represented by function `f`. The second stage is a 1-place FIFO element. The structure of this program can of course be replicated to obtain longer (also called 'deeper') pipelines. Furthermore, the pipeline may be forked and joined, by splitting and combining the dataflow. Bypasses and counterflows can also be added, but that is a VLSI programming exercise that falls outside the scope of this thesis.

```
begin
  int    = type [0..1023]
& f      = func (s:int):int . (...)
& a,b,c : chan int
|    begin x: var int | forever do a?x ; b!f(x) od end
  || begin y: var int | forever do b?y ; c!y od end
  || begin z: var int | forever do c?z ; ... od end
end
```

The exact circuit realization of this pipeline depends on the compilation to handshake circuits and the peephole optimizations that are applied. Two handshake circuit realizations are possible, with slightly different performance characteristics.

Direct compilation of this Tangram program results in the handshake circuit of Fig. 8.1. Only the first two stages are shown. In this circuit, the passivators implement the synchronization that is required for communication, for instance between

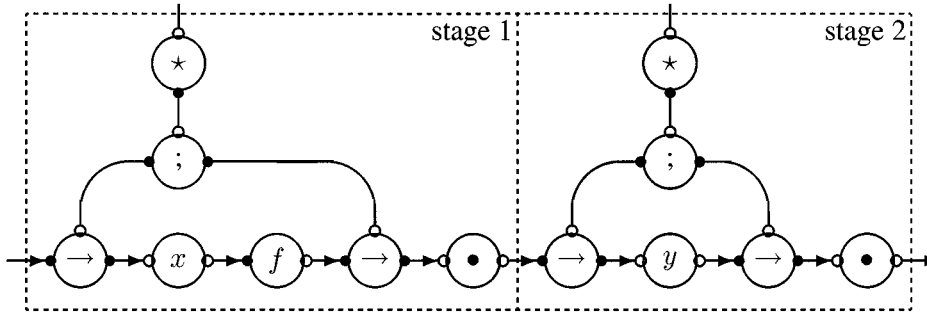


Figure 8.1: Handshake circuit for a Tangram-programmed pipeline with synchronization in datapath (passivators).

`b!f(x)` and `b?y`.

Both the double-rail and the two-phase single-rail implementation of passivators are rather expensive (in terms of area and energy). An alternative to the use of a passivator is to move the synchronization from the datapath to the control, via peephole optimization. This rule is introduced in [6] and results in the handshake circuit show in Fig. 8.2. Basically, this rule replaces the synchronization of two (otherwise independent) data transfers by one synchronized data transfer. The behavior of the substituted circuit is not fully equivalent, but allows for only a subset of the original behavior (this is detailed later). The join component implements the control synchronization.

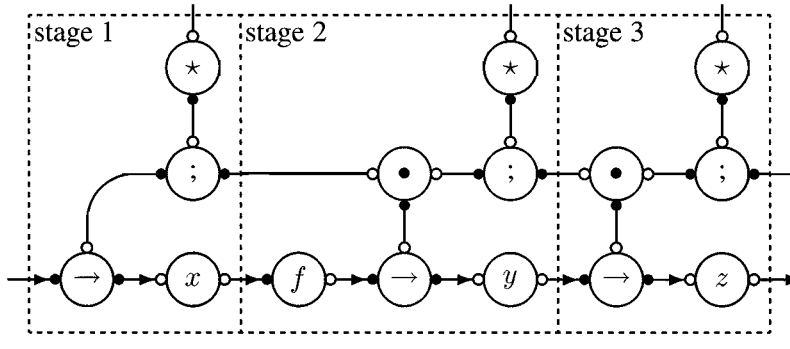


Figure 8.2: Handshake circuit for a Tangram-programmed pipeline with synchronization in control path (joins).

The compilation of the handshake circuit with the passivator (Fig. 8.1) is described in Section 4.3. The input transferrer in each stage (left) is implemented such that the latches in the associated variable are made transparent before the commu-

nication with the passivator of the previous stage is initiated. The output transferrers (right) and the passivator are implemented such that the actual synchronization in the passivator takes place only after the delay-matching has completed. With this implementation (which actually corresponds to the true four-phase compilation of the handshake circuit), each stage is implemented such that the latches are operated in what is called a *normally transparent* way. Only when the output is actually interpreted, are the latches opaque.

The behavior of the circuit that is generated for Fig. 8.2 is subtly different. In this peephole-optimized handshake circuit, latches in a variable are made transparent only after the two sequencers that are involved in a communication have synchronized through the join component. The latches are thus *normally opaque*. The latches in variable y, for instance, are opened after the first delay-matching phase through $f(x)$, rather than immediately after y has been copied to z, as in Fig. 8.1.

Which implementation of the pipeline control is preferred depends on the performance characteristics that are required. The direct implementation (Fig. 8.1) has reduced latency, but can consume more power when the data from f makes spurious transitions.

In four-phase, the implementation costs (circuit area) of the two handshake circuits are almost equal. In two-phase, however, the implementation of the passivator requires a latch per bit. Therefore, in two-phase, the handshake circuit from Fig. 8.2 should be chosen. The two-phase implementations of the transferrer and the sequencer consist of wires only. The join is implemented with a single C-element. The repeater is required only for initialization (start up) purposes and corresponds to the combination of the bubble and the reset of the C-element from Sutherland's implementation. The two-phase implementation of the handshake circuit from Fig. 8.2 thus is equivalent to the normally-opaque implementation as described in [63].

In the AMULET project at Manchester University a lot of different two-phase and four-phase micropipeline control circuits have been investigated, see for instance [32, 63, 25]. The conclusion seems to be that a four-phase control circuit is more area-efficient and leads to significant faster circuits. Farnsworth proposes to apply some of these control circuits in the implementation of Tangram-programmed pipelines [30]. Basically, any dedicated micropipeline control circuit can be applied in Tangram pipelines through peephole optimization. Pipeline structures can thus be programmed straightforwardly in Tangram and compiled to efficient circuit implementations.

8.1.4 Synchronous circuits

The main difference between an asynchronous and a synchronous datapath is in the enabling of the latches and flip-flops. The clock in a synchronous circuit enables

all flip-flops at all clock ticks, unless clock gating is applied. In a single-rail circuit, latches are enabled only if a data-transfer really has to take place. The distributed handshake control in combination with the high enable efficiency makes the single-rail circuits potentially highly power efficient.

For the DCC error detector we have demonstrated an interesting power advantage: the single-rail realization consumes only a fifth of the energy (and power) of the synchronous realization (see Chapter 7 and [10]). A more extensive discussion of the asynchronous low-power potential, annotated with various examples, and including a quantification of the power efficiency of synchronous circuits, can be found in [17].

In many asynchronous circuits the supply voltage can be tuned without risking malfunctioning of the circuit. This feature can be exploited to control the supply voltage such that the resulting performance of the circuit exactly matches the required performance [62], thus delivering the specified performance at minimum (average) voltage and hence at minimum power. In a synchronous implementation this adaptive scaling of the supply-voltage is less straightforward.

An important difference between synchronous circuits and single-rail asynchronous circuits is the way timing is dealt with. In a synchronous circuit the clock is specified such that the worst-case computation time can be accommodated within a full or half clock cycle (depending on the particular clocking scheme that is applied). In a single-rail circuit, delays are implemented on chip to achieve essentially the same goal: safe data transfer.

The handshake signaling in a single-rail circuit, which is used to replace the clock of a synchronous equivalent, in general implies that the asynchronous realization requires more silicon area. The area overhead depends on the application, but need not be more than 20%; a figure that was demonstrated for the DCC error detector.

There are several reasons why the area overhead can be relatively small. First of all, our single-rail circuits are based on latches rather than flip-flops. This means that storage effectively may require less area. (Sometimes, however, two latches are required to implement operations on state variables, in which case the latch advantage disappears.) The logic in a single-rail datapath is equivalent to the synchronous logic, so this does not result in an area overhead either. The distributed control, which is required to achieve the power efficiency of asynchronous circuits, *does* contribute to the extra area. Part of the area required for the asynchronous control cancels against the area required for the central synchronous controller, which is often implemented in ROM.

In situations where raw speed is the main design criterion, synchronous circuits are generally more applicable than asynchronous circuits based on handshakes. The handshaking overhead is part of the critical timing path and thus directly limits the

maximum attainable speed.

The availability of a global clock, in combination with flip-flops, makes the testability of a synchronous circuit a manageable task. Scan techniques can be straightforwardly applied to achieve complete observability of the state of the circuit. The overhead of full scan is limited to the larger scan flip-flops and the routing of a test control signal and the scan path. In a single-rail circuit, with its latch-based datapath and local enabling, full observability is less straightforward. The cost of full scan is high if this implies that all latches are simply replaced by flip-flops. In the Tangram project, partial scan in combination with design-for-test has been successfully applied to obtain full observability at acceptable (low) cost [69, 79].

8.2 Strengths

The main contribution of the single-rail design flow as advocated in this thesis is its *push-button* aspect. It turned out to be feasible to compile handshake circuits to single-rail netlists automatically, with high confidence in the correctness of the resulting silicon. This design flow has actually been implemented and resulted in first-time-right silicon in the form of a single-rail DCC error detector.

A second strength is that a *generic standard-cell library* proved to be sufficiently powerful to implement all asynchronous elements (C-elements, delays, and even arbiters [15]). Throughout the project this has helped us in easily retargeting the design-flow to other cell libraries. The addition of a new 0.5μ CMOS technology (not scaled from the original 0.8μ library, with a different latch, and several cells that were not available in the 0.8μ library) as target to the design-flow, for instance, was a matter of hours rather than days.

The only assumptions that have to be verified in the mapping onto the cell library are those of the isochronic fork, the timing assumptions in the actual implementation of the delay elements, and the driver strategy.

Throughout this thesis the power of *peephole optimization* has been illustrated in numerous examples and on various levels of representation. Silicon compilers based on syntax-directed translation are often criticized because they supposedly lead to inefficient circuit realizations. These critics generally neglect the combined power of simple representations and peephole optimizations, techniques that are common in software design and compilers. A clear advantage of peephole optimization is that additional rules can straightforwardly be added to the design flow. The effect of this leads to a learning curve, as is illustrated in Table 8.3 for single-rail implementations of the DCC error detector.

From the table one may observe that the area efficiency of the single-rail implementation of handshake circuits is still improving. The extra improvement in

Version	#MOSts	#cells	#gate eq.	perc.	date
SR core of chip	21,796	3,296	5,035	105%	Jun 94
SR layout	20,288	3,101	4,783	100%	Feb 95
SR netlist	19,912	3,017	4,684	98%	Dec 95
SR netlist	21,542	2,904	4,559	95%	Dec 95

Table 8.3: Single-rail learning curve for the DDD. The first three entries refer to the 0.8μ cell library that is also assumed throughout the rest of the thesis. The last entry applies to a richer (0.5μ) cell library. (The unit for gate equivalents is the cell-area of a two-input NAND.)

the last entry in the table is due to a slightly richer cell library, which is exploited in a few additional peephole rules. The transistor count is higher because of the different implementation of the latch.

A key advantage of single-rail handshake circuits is their *low power consumption*. The DCC error detector demonstrated a factor six advantage over synchronous, and a factor two over double-rail.

A nice feature of the true four-phase protocol, which is used in this thesis to implement the single-rail handshake circuits, is that (performance wise) it is an *efficient four-phase protocol*, in the sense that all phases of the handshake are functional. The down phase of a four-phase handshake is generally called return-to-zero phase, and is often also used like this, which means that all work is performed during the up phase of the handshake. We have been able (at least in the datapath) to spread the work over all four phases.

This true four-phase protocol allowed us to implement symmetric delays. Interestingly, Seitz [72, Sec. 7.8.2] does *not* mention this option when he addresses the implementation of single-rail datapaths:

> *The addition of a delay to any ordinary combinational net converts it to a self-timed combinational element operating on single-rail data. If 2-cycle request-acknowledge signaling is used, the delay should be symmetrical, and for 4-cycle signaling asymmetrical. Depending on the data-validity coding scheme employed, the outputs may need to be loaded in latches by a signal produced as Request and not Acknowledge.*

The data-valid scheme that is referred to in this text is the early scheme. This scheme indeed requires asymmetric delays, and has a (functionally) redundant return-to-zero phase.

8.3 Weaknesses

In its current status, a weakness of the single-rail design-flow is the conservative delay-matching that is used. During netlist generation, delays are matched on a worst-case basis. After the peephole optimization phase, some of these delays can be reduced because the datapath has actually become faster. Furthermore, control overhead can sometimes be discounted. None of these techniques have thus far been implemented. They would require the use of timing analyzers to estimate the worst-case computation times in the datapath and propagation times in the control and through delay elements.

Low power is an important selling point for single-rail handshake circuits, but there are also some weak points with respect to power. Most notably, we have not restricted the number of spurious transitions in the datapath. In deep combinatorics this may lead to a waste of power.

This also relates to another potential weakness, namely a loss in transparency with respect to power from the VLSI programmer's point of view. Most notably, in double-rail handshake circuits, the energy required for an assignment is determined by the number of bits in the variable, and its number of read ports (the number of occurrences of the variable in expressions). With single-rail implementations, the energy required for an assignment also depends on the structure of the expressions in which the variable occurs.

Another potential weak point is raw speed. Handshake circuits are aimed at low power, and achieve this by enabling datapaths only when required. This, however, introduces a handshaking overhead that cannot always be afforded, for instance, if high performance is required. In that case a synchronous solution, possibly combined with clock-gating, may result in better performance at acceptable cost.

8.4 Opportunities

Given the strengths and weaknesses of single-rail handshake circuits, the best opportunities are expected in applications in which low or medium range performance is sufficient, low power is a key issue, and where a small (20%) area overhead over synchronous circuits is acceptable.

The range of consumer-electronic products to which this applies includes portable, battery-powered products, such as phones, personal digital assistants (PDAs), personal communicators, CD players, and video games. One of the targets of the Tangram project therefore is to find an application in which asynchronous techniques make a difference, in the sense that they contribute to a longer battery lifetime or a lighter product.

Other potential applications for single-rail, or asynchronous techniques in general, are circuits that have to deal with a highly variable work load in a low-power environment. In such applications the property that handshake circuits consume zero stand-by power and require no additional wake-up time (for example to await a well-defined clock signal) may prove to be essential.

Furthermore, applications that are asynchronous by nature, such as sample-rate converters, or interfaces to DRAM, I^2C, and I^2S, can profit from single-rail implementations.

The target application for single-rail handshake circuits excludes high-performance microprocessors, since in that case any asynchronous control directly limits the maximum speed. Another implementation style that is not suited for single-rail circuits is that of pipelined digital signal processing in which clock rate and sample frequency are about the same. Handshaking is then pure overhead and potential average-case processing times cannot easily be exploited.

8.5 Threats

Although the potential advantage of single-rail handshake circuits over synchronous circuits has been demonstrated, the success of asynchronous circuits is still not guaranteed. On the one hand, advances in synchronous techniques may diminish some of the key advantages; on the other hand, weaker points of single-rail may turn out to be definite roadblocks.

A serious threat for the low-power advantage is clock-gating. Although this is not expected to achieve the same low level of granularity in enabling as single-rail techniques, it may be that in many applications clock gating closes a significant part of the low-power gap. Clock gating also requires additional gates, so the low-power advantage comes at the price of an increase in silicon area. This technique, however, could make the handshake overhead for the rest of the low-power gap unacceptable.

In today's technology, delay matching is relatively easy to implement because wiring capacitances can be predicted rather accurately before routing, and transition times can be bounded. With very small dimensions and many layers of interconnect, this situation may change. Given the need to accommodate variations in processing and operating conditions, it may become increasingly hard to implement the delay elements efficiently and still achieve acceptable performance.

Another threat for asynchronous circuits is that it is harder to come up with a cost-effective push-button test strategy (automatic generation of test hardware and test patterns) then for synchronous circuits. In a synchronous circuit one can chain the flip-flops together in a scan chain and then use the clock to step-wise operate both this chain and the circuit. The combination of the clock and the scan chain

result in both high controllability and high observability at relatively low cost.

Single-rail implementations of handshake circuits have two properties that make testing harder. First of all the handshake control results in circuits that are highly autonomous. This makes if harder to control the operation of the circuit during test, for instance, if IDDQ testing is required. The addition of test-hardware may then be required to introduce quiescent states in which IDDQ measurements can take place. Another complication is the use of latches in the datapath. Although this has advantages during normal operation of the circuit, it makes it harder to introduce scan chains in a straightforward way. One could of course replace the latches by scan flip-flops, but this would imply relatively high cost for testing. A more attractive solution is to try to combine latches into flip-flops.

8.6 Remaining issues

Single-rail has replaced double-rail as the implementation standard for handshake circuits in the Tangram project. Although the single-rail design flow has already resulted in working silicon, it is not yet mature enough to finish the investigation into single-rail techniques. Some of the work that still has to be done is addressed in this section.

8.6.1 Synthesis

Both in the control and the datapath we have followed an approach based on substituting gate implementations for handshake components and then using peephole optimization to eliminate inefficiencies at component boundaries. Although this has turned out to be an effective strategy, it may be that the use of dedicated synthesis tools results in better (smaller, faster) circuits.

For asynchronous controllers, quite a few automatic synthesis tools exist. These tools are directly applicable to the control part of Tangram handshake circuits. The ASSASSIN compiler from IMEC [85], for instance, can be applied to improve handshake control circuits [51]. A likely consequence of the synthesis approach is a loss in transparency.

Logic synthesis tools, which generally form a combination of logic optimizers and technology mappers, can straightforwardly be applied to the data part of Tangram handshake circuits. This might lead to more efficient realizations, although improvements of more then 10% are very unlikely.

8.6.2 Performance

The emphasis throughout the thesis has been on area and energy efficiency of the implementations of handshake circuits. This led to the choice of the true four-phase protocol for implementing operations in the datapath. Within this protocol, higher performance can be achieved in the datapath by tighter delay-matching and by eliminating completion detection. This requires additional timing assumptions to be made and thus comes at the cost of more verification effort and possibly more spurious transitions in the datapath.

Another way to achieve higher performance may be to choose the early variant of the four-phase handshake protocol, both in the control and the datapath. In the datapath, the early scheme does not lead to faster cycle times, since matching delays cannot be reduced (their implementation cost increases, due to the quick reset), and the complexity of latch-control circuits increases. In the control, however, the protocol allows the (then functionally redundant) return-to-zero phases to be in parallel with other (productive) phases. This may actually result in faster circuits. For a handshake variable, for instance, the true four-phase protocol requires at least eight *sequential* handshake events for a read plus write handshake. In the early scheme this number may be reduced to four sequential events. The higher degree of parallelism of the early scheme, however, generally leads to more C-elements (to resynchronize forked handshake events), which makes it hard actually to gain speed.

Bibliography

[1] Morteza Afghahi and Christer Svensson. A unified single-phase clocking scheme for VLSI systems. *IEEE Journal of Solid-State Circuits*, 25(1):225–232, January 1990.

[2] Andrew Bailey and Mark Josephs. Sequencer circuits for VLSI programming. In *Asynchronous Design Methodologies*, pages 82–90. IEEE Computer Society Press, May 1995.

[3] H. B. Bakoglu. *Circuits, Interconnections, and Packaging for VLSI*. Addison-Wesley, 1990.

[4] Peter Beerel and Teresa Meng. Semi-modularity and self-diagnostic asynchronous control circuits. In Carlo H. Séquin, editor, *Advanced Research in VLSI: Proceedings of the 1991 UC Santa Cruz Conference*, pages 103–117. MIT Press, March 1991.

[5] C. H. (Kees) van Berkel, Martin Rem, and Ronald W. J. J. Saeijs. VLSI programming. In *Proc. International Conf. Computer Design (ICCD)*, pages 152–156. IEEE Computer Society Press, 1988.

[6] C. H. (Kees) van Berkel and Ronald W. J. J. Saeijs. Compilation of communicating processes into delay-insensitive circuits. In *Proc. International Conf. Computer Design (ICCD)*, pages 157–162. IEEE Computer Society Press, 1988.

[7] Kees van Berkel. Beware the isochronic fork. *Integration, the VLSI journal*, 13(2):103–128, June 1992.

[8] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[9] Kees van Berkel and Arjan Bink. Single-track handshaking signaling with application to micropipelines and handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, March 1996.

[10] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, Frits Schalij, and Rik van de Wiel. A single-rail re-implementation of a DCC error detector using a generic standard-cell library. In *Asynchronous Design Methodologies*, pages 72–79. IEEE Computer Society Press, May 1995.

[11] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

[12] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. A fully-asynchronous low-power error corrector for the DCC player. *IEEE Journal of Solid-State Circuits*, 29(12):1429–1439, December 1994.

[13] Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalij. A fully-asynchronous low-power error corrector for the DCC player. In *International Solid State Circuits Conference*, pages 88–89, February 1994.

[14] Kees van Berkel, Ronan Burgess, Joep Kessels, Marly Roncken, and Frits Schalij. Characterization and evaluation of a compiled asynchronous IC. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 209–221. Elsevier Science Publishers, 1993.

[15] Kees van Berkel, Ferry Huberts, and Ad Peeters. Stretching quasi delay insensitivity by means of extended isochronic forks. In *Asynchronous Design Methodologies*, pages 99–106. IEEE Computer Society Press, May 1995.

[16] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)*, pages 384–389, 1991.

[17] Kees van Berkel and Martin Rem. VLSI programming of asynchronous circuits for low power. In Graham Birtwistle and Al Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 152–210. Springer-Verlag, 1995.

[18] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 262–265. IEEE Computer Society Press, November 1989.

[19] Steven M. Burns. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[20] Steven M. Burns and Alain J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and F. Leighton, editors, *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI*, pages 35–50. MIT Press, 1988.

[21] T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.

[22] Hsi-Chuan Chen and David Hung-Chang Du. Path sensitization in critical path problem. *IEEE Transactions on Computer-Aided Design*, 12(2):196–207, February 1993.

[23] Wesley A. Clark. Macromodular computer systems. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 335–336, Atlantic City, NJ, 1967. Academic Press.

[24] D. Cohen. On holy wars and a plea for peace. *Computer*, 14(10):48–54, October 1981.

[25] Paul Day and J. Viv Woods. Investigation into micropipeline latch design styles. *IEEE Transactions on VLSI Systems*, 3(2):264–272, June 1995.

[26] Mark E. Dean, David L. Dill, and Mark Horowitz. Self-timed logic using current-sensing completion detection (CSCD). In *Proc. International Conf. Computer Design (ICCD)*, pages 187–191. IEEE Computer Society Press, October 1991.

[27] Mark E. Dean, David L. Dill, and Mark Horowitz. Self-timed logic using current-sensing completion detection (CSCD). *Journal of VLSI Signal Processing*, 7(1/2):7–16, February 1994.

[28] Daniel W. Dobberpuhl et al. A 200-MHz 64-b dual-issue CMOS microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11):1555–1564, November 1992.

[29] Jo C. Ebergen. *Translating Programs into Delay-Insensitive Circuits*, volume 56 of *CWI Tract*. Centre for Mathematics and Computer Science, 1989.

[30] C. Farnsworth, D. A. Edwards, Jianwei Liu, and S. S. Sikand. A hybrid asynchronous system design environment. In *Asynchronous Design Methodologies*, pages 91–98. IEEE Computer Society Press, May 1995.

[31] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, S. Temple, and J. V. Woods. The design and evaluation of an asynchronous microprocessor. In *Proc. International Conf. Computer Design (ICCD)*. IEEE Computer Society Press, October 1994.

[32] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *Proceedings of VLSI 93*, pages 5.4.1–5.4.10, September 1993.

[33] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. AMULET1: A micropipelined ARM. In *Proceedings IEEE Computer Conference (COMPCON)*, pages 476–485, March 1994.

[34] Jim D. Garside. A CMOS VLSI implementation of an asynchronous ALU. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 181–207. Elsevier Science Publishers, 1993.

[35] W. Wayt Gibbs. Turning back the clock. *Scientific American*, 272(6), June 1995.

[36] Bruce Gilchrist, J. H. Pomerene, and S. Y. Wong. Fast carry logic for digital computers. *IRE Transactions on Electronic Computers*, EC-4(4):133–136, December 1955.

[37] E. Grass and S. Jones. Asynchronous circuits based on multiple localised current-sensing completion detection. In *Asynchronous Design Methodologies*, pages 170–177. IEEE Computer Society Press, May 1995.

[38] Jaco Haans. VLSI programming of multipliers in Tangram. Master's thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1992.

[39] Jaco Haans, Kees van Berkel, Ad Peeters, and Frits Schalij. Asynchronous multipliers as combinational handshake circuits. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 149–163. Elsevier Science Publishers, 1993.

[40] Pieter J. Hazewindus. *Testing Delay-Insensitive Circuits*. PhD thesis, California Institute of Technology, 1992.

[41] R. Hitchcock. Timing verification and the timing analysis program. In *Proc. ACM/IEEE Design Automation Conference*, pages 594–604, 1982.

[42] Albert S. Hoagland. *Digital Magnetic Recording*. John Wiley & Sons, 1963.

[43] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[44] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[45] Abraham Hoogendoorn. Digital compact cassette. *Proceedings of the IEEE*, 82(10):1479–1489, October 1994.

[46] Kai Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley & Sons, 1979.

[47] Anne Kaldewaij. *A Formalism for Concurrent Processes*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1986.

[48] Joep Kessels. VLSI programming of a low-power asynchronous Reed-Solomon decoder for the DCC player. In *Asynchronous Design Methodologies*, pages 44–52. IEEE Computer Society Press, May 1995.

[49] Joep Kessels, Kees van Berkel, Ronan Burgess, Marly Roncken, and Frits Schalij. An error decoder for the compact disc player as an example of VLSI programming. In *Proc. European Conference on Design Automation (EDAC)*, pages 69–74, 1992.

[50] Lindsay Kleeman and Antonio Cantoni. Metastable behavior in digital systems. *IEEE Design & Test of Computers*, 4:4–19, December 1987.

[51] Tilman Kolks, Steven Vercauteren, and Bill Lin. Control resynthesis for control-dominated asynchronous designs. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, March 1996.

[52] A. P. Kostelijk. Vera, a rule-balsed verification assistant for VLSI circuit design. In *Proc. of the VLSI 89 Conference*, pages 89–98, August 1989.

[53] A. P. Kostelijk. *Verification of electronic designs by reconstruction of the hierarchy*. PhD thesis, Eindhoven University of Technology, September 1994.

[54] G. C. P. Lokhoff. Digital compact cassette. *IEEE Transactions on Comsumer Electronics*, 37(3):702–706, August 1991.

[55] Lisa Maliniak. Single-rail DCC error detector ups speed and reduces sice and power. *Electronic Design*, page 48, June 12 1995.

[56] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[57] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64. Addison-Wesley, 1990.

[58] Alain J. Martin. Asynchronous datapaths and the design of an asynchronous adder. *Formal Methods in System Design*, 1(1):119–137, July 1992.

[59] W. M. McKeeman. Peephole optimization. *Communications of the ACM*, 8(7):443–444, 1965.

[60] Noel Menezes, Satyamurthy Pullela, and Lawrence T. Pileggi. Simulateous gate and interconnect sizing for circuit-level delay optimization. In *Proc. ACM/IEEE Design Automation Conference*, pages 690–695, 1995.

[61] David E. Muller and W. S. Bartky. A theory of asynchronous circuits. In *Proceedings of an International Symposium on the Theory of Switching*, pages 204–243. Harvard University Press, April 1959.

[62] L. S. Nielsen, C. Niessen, J. Sparsø, and C.H. van Berkel. Low-power operation using self-timed and adaptive scaling of the supply voltage. *IEEE Transactions on VLSI Systems*, 2(4):391–397, December 1994.

[63] N. C. Paver. *The Design and Implementation of an Asynchronous Microprocessor.* PhD thesis, Department of Computer Science, University of Manchester, June 1994.

[64] Ad Peeters. *The 'Asynchronous' Bibliography* (in BibTeX format). Uniform Resource Locator: `ftp://ftp.win.tue.nl/pub/tex/async.bib.Z`. Corresponding email address: `async-bib@win.tue.nl`.

[65] Ad Peeters and Kees van Berkel. Single-rail handshake circuits. In *Asynchronous Design Methodologies*, pages 53–62. IEEE Computer Society Press, May 1995.

[66] Marcel J. M. Pelgrom, Aad C. J. Duinmaijer, and Anton P. G. Welbers. Matching properties of MOS transistors. *IEEE Journal of Solid-State Circuits*, 24(5):1433–1440, 1989.

[67] J. Peterson. *Petri net theory and modeling of systems*. Prentice-Hall, 1981.

[68] Miroslav Pečhouček. Anomalous response times of input synchronizers. *IEEE Transactions on Computers*, 25(2):133–139, February 1976.

[69] Marly Roncken. Partial scan test for asynchronous circuits illustrated on a DCC error corrector. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 247–256, November 1994.

[70] Manoj Sachdev. Iddq and voltage testable CMOS flip-flop configurations. In *Proceedings of International Test Conference*, October 1995.

[71] Frits D. Schalij. Tangram manual. Technical Report UR 008/93, Philips Research, Eindhoven, 1993.

[72] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[73] J. Sparsø, C. D. Nielsen, L. S. Nielsen, and J. Staunstrup. Design of self-timed multipliers: A comparison. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 165–179. Elsevier Science Publishers, 1993.

[74] Mishell J. Stucki, Severo M. Ornstein, and Wesley A. Clark. Logical design of macromodules. In *AFIPS Conference Proceedings: 1967 Spring Joint Computer Conference*, volume 30, pages 357–364, Atlantic City, NJ, 1967. Academic Press.

[75] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[76] Andrew S. Tanenbaum, Hans van Staveren, and Johan W. Stevenson. Using peephole optimization on intermediate code. *ACM Transactions on Programming Lanuages and Systems*, 4(1):21–36, January 1982.

[77] Stephen H. Unger and Chung-Jen Tan. Clocking schemes for high-speed digital systems. *IEEE Transactions on Computers*, 35(10):880–895, October 1986.

[78] Jan L. A. van de Snepscheut. *Trace Theory and VLSI Design*, volume 200 of *Lecture Notes in Computer Science*. Springer-Verlag, 1985.

[79] Rik van de Wiel. High-level test evaluation of asynchronous circuits. In *Asynchronous Design Methodologies*, pages 63–71. IEEE Computer Society Press, May 1995.

[80] Harry J. M. Veendrick. Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits. *IEEE Journal of Solid-State Circuits*, 19(4):468–473, August 1984.

[81] Harry J.M. Veendrick. The behavior of flip-flops used as synchronizers and prediction of their failure rate. *IEEE Journal of Solid-State Circuits*, 15(2):169–176, 1980.

[82] Tom Verhoeff. Delay-insensitive codes—an overview. *Distributed Computing*, 3(1):1–8, 1988.

[83] Tom Verhoeff. *A Theory of Delay-Insensitive Systems*. PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1994.

[84] Neil H. E. Weste and Kamran Eshraghian. *Principles of CMOS VLSI Design: a Systems Perspective*. Addison-Wesley, 1993. Second Edition.

[85] Chantal Ykman-Couvreur, Bill Lin, and Hugo de Man. Assassin: A synthesis system for asynchronous control circuits. Technical report, IMEC, September 1994. User and Tutorial manual.

[86] Jiren Yuan, Ingemar Karlsson, and Christer Svensson. A true single-phase-clock dynamic CMOS circuit technique. *IEEE Journal of Solid-State Circuits*, 22(5):899–901, October 1987.

Summary

Nearly all digital ICs that are made today are *synchronous*, which means that for their operation they depend on a clock signal. This periodic signal essentially dictates the pace at which the circuit is supposed to work. At Philips Research Laboratories Eindhoven the design of *asynchronous* ICs is investigated. These circuits do not use a clock signal, but instead employ *handshaking* to control the computation.

An important potential benefit of these asynchronous ICs is low energy consumption, especially in applications in which the full computational potential of the IC is not always required. This thesis starts at the point where this advantage has been proven in an experimental DCC player, containing asynchronous ICs that only consume a fifth of the energy of their synchronous contemporaries. Two issues, however, hindered the application of asynchronous circuits at a large scale: the ICs were clearly too large compared to their synchronous counterparts and furthermore required a dedicated standard-cell library. These two problems formed the main motivation for the research documented in this thesis.

In order to understand the reason behind these handicaps we should have more insight into the design of asynchronous circuits, especially into the approach that is pursued at the Nat.Lab. The method is based on the combination of *VLSI-programming* and *silicon compilation*. In this approach a designer describes a function in a VLSI-programming language, called Tangram. Tangram is a conventional programming language with additional constructs for parallelism, communication, and reuse of hardware (sharing). Several tools are available to support the designer in making trade-offs with respect to the area, speed, energy consumption, and testability of a design.

The compilation from Tangram to silicon is performed in two steps. Firstly, the Tangram program is compiled into an intermediate representation that we call *handshake circuits*. In a second step the handshake circuit is mapped onto the required standard-cell library. Subsequently, a layout can be made and based on that an IC can be processed.

In the DCC chips, data was encoded using the *double-rail* scheme, which implies that two wires per bit are used. Such a data-encoding scheme requires only

a minimum of timing assumptions and, therefore, the compilation from handshake circuits to standard-cell libraries is simple. Although double-rail encoding leads to highly reliable communication and robust ICs, it is also the cause of the two problems identified above. The use of two wires per bit automatically leads to extra area compared to synchronous circuits, in which one wire per bit suffices. Furthermore, the efficient implementation of operations such as addition, in double-rail requires a number of dedicated cells that are not available in present standard-cell libraries.

The *single-rail* part in the title of this thesis refers to the use of only one wire per bit in combination with a separate wire to indicate the validity of the complete word. The use of one wire per bit directly leads to a significant area reduction compared to double-rail. Furthermore, using this encoding, operations on data can readily be realized in any common standard-cell library. In addition to these advantages, the resulting circuits are faster and more energy efficient.

One of the contributions of this thesis is an inventory of single-rail handshake protocols. It turns out that there are numerous ways to combine single-rail data encoding with handshake communication. This leads to a surprisingly rich domain of alternatives to choose from.

The innovation of the research as documented in this thesis is neither single-rail, nor handshake circuits in isolation, but the combination of these two in the context of silicon compilation and a standard-cell layout style. Moreover, in the implementation a four-phase handshake protocol is applied in which none of the phases is redundant. It was commonly believed that this was not possible.

The expansion of handshake circuits into single-rail realizations in a standard-cell library is extensively documented in this thesis. The compilation has also been automated completely in such a way that only minimal verification is required after layout generation. It was not beforehand obvious that this could indeed be realized.

An important aspect of the design flow that we have implemented is *peephole optimization*, a technique to improve the implementation efficiency stepwise that is well known in compiler construction. We have chosen to keep the various compilation steps and the respective representations simple. At each level peephole optimization is applied to replace frequent combinations of elements by more efficient ones (smaller, faster, and more energy efficient).

With the reimplementation of one of the DCC chips we have proven that the single-rail design flow can indeed be applied successfully. The single-rail IC turns out to be fully functional, both on a tester and in the experimental DCC player, which really plays music.

Compared to the previously realized double-rail IC, the single-rail version is a third smaller, uses only half the energy, and is a quarter faster. With respect to the best present synchronous counterpart the single-rail demonstrator IC is about a sixth larger, but uses only a sixth of the energy.

Single-rail handshake circuits appear to be an attractive technique to design low-power ICs against little additional costs. Especially for portable products, which are battery powered, the single-rail implementation of digital functions can make an interesting contribution to a longer battery lifetime, a lighter product, or more appealing functionality.

Samenvatting

Bijna alle digitale ICs die gemaakt worden zijn *synchroon*, wat betekent dat ze voor hun werking afhankelijk zijn van een klok. Dit periodieke signaal geeft in feite het tempo aan waarin de chip geacht wordt te werken. Op het Philips Natuurkundig Laboratorium wordt gewerkt aan de ontwikkeling van *asynchrone* ICs, die voor hun werking geen klok gebruiken. In plaats daarvan wordt *handshaking* gebruikt om de berekening te sturen.

Een belangrijk potentieel van deze asynchrone ICs is laag energieverbruik, met name in toepassingen waarin de volle rekenkracht van het IC niet altijd ingezet hoeft te worden. Dit proefschrift begint op het punt waar dit energievoordeel is aangetoond in een experimentele opstelling voor een DCC speler, met daarin asynchrone ICs die slechts een vijfde van de energie van hun synchrone tijdgenoten gebruiken. Twee feiten belemmerden echter grootschalige toepassing van deze asynchrone circuits: de ICs waren duidelijk te groot in vergelijking met synchrone tegenhangers en vereisten bovendien een eigen bibliotheek van standaard cellen. Aan beide problemen moest dringend iets gedaan worden.

Om te begrijpen waardoor deze handicaps werden veroorzaakt moeten we iets meer weten over het ontwerp van asynchone circuits zoals dat op het Nat.Lab. plaatsvindt. De ontwerpaanpak is gebaseerd op een combinatie van *VLSI-programmeren* en *siliciumcompilatie*. Binnen deze aanpak beschrijft een ontwerper de te ontwerpen functie in Tangram, een VLSI-programmeertaal met, behalve de gewoonlijke taalconstructen, ook constructen voor parallellisme, communicatie en hergebruik van hardware. Diverse gereedschappen ondersteunen de ontwerper bij het maken van afwegingen over oppervlakte, snelheid, energieverbruik en testbaarheid van zijn ontwerp.

De vertaling van Tangram naar silicium verloopt in feite in twee stappen. Allereerst wordt het Tangram programma vertaald naar een tussenrepresentatie die we *handshake circuits* noemen. Daarna wordt het handshake circuit afgebeeld op de gewenste standaard-cell bibliotheek en kan een layout en vervolgens een IC gemaakt worden.

Om de vertaling van handshake circuits naar standaard-cell bibliotheek eenvou-

dig te houden werd in het verleden een minimum aan timing-aannames gemaakt. De data werd daartoe gecodeerd in *double-rail*, wat inhoudt dat twee draden per bit gebruikt worden. Hoewel deze codering tot zeer betrouwbare communicatie en robuuste ICs leidt, is ze ook de oorzaak van de twee hierboven gesignaleerde problemen. Het gebruik van twee draden per bit leidt vanzelf tot extra oppervlakte ten opzichte van synchrone circuits, waarin één draad per bit volstaat. Bovendien vereist efficiënte implementatie van operaties zoals optellen in double-rail een aantal speciale cellen die men niet in een hedendaagse standaard-cell bibliotheek aantreft.

Het *single-rail* uit de titel van dit proefschrift verwijst naar het gebruik van één draad per bit in combinatie met een aparte draad om de geldigheid van die data aan te geven. Binnen deze codering kunnen operaties op data eenvoudig gerealiseerd worden in elke standaard-cell bibliotheek. Bovendien leidt dit tot een reductie in oppervlakte en worden de schakelingen sneller en zuiniger.

Eén van de bijdragen van dit proefschrift is een inventarisatie van de mogelijke manieren om single-rail codering van data te combineren met handshake communicatie. Dit leidt tot een verrassend groot aantal alternatieven en maakt het bovendien mogelijk om een weloverwogen keuze te maken voor de beste variant.

Het vernieuwende aspekt van het onderzoek zoals beschreven in dit proefschrift betreft niet single-rail of handshake circuits op zichzelf, maar de combinatie van deze twee in de context van siliciumcompilatie en een standaard-cell layout stijl. Bovendien is in de implementatie gebruik gemaakt van een vier-fasen handshake protocol waarin geen van de fasen overbodig is, iets waarvan algemeen werd aangenomen dat dit niet mogelijk was.

In dit proefschrift wordt de vertaling van handshake circuits naar single-rail realisaties in een standaard-cell bibliotheek uitgebreid beschreven. Deze vertaling is in zijn geheel ook geautomatiseerd en wel zodanig dat er na het maken van de layout slechts minimale verificatie noodzakelijk is. Vooral van dit laatste aspekt was op voorhand verre van duidelijk dat het realiseerbaar was.

Een belangrijk aspekt van de ontwerpaanpak die we hebben geïmplementeerd is *peephole optimalisatie*, een techniek om stapsgewijs de efficiency van implementaties te verbeteren, die bij compilerbouw vrij algemeen wordt toegepast. We hebben ervoor gekozen om de diverse vertaalstappen en de daarbij behorende representaties eenvoudig te houden. Op elk niveau wordt peephole optimalisatie toegepast om veel voorkomende combinaties van elementen te vervangen door efficiëntere tegenhangers (kleiner, sneller, zuiniger).

Aan de hand van een herimplementatie van een van de DCC chips is bewezen dat de single-rail ontwerpaanpak inderdaad succesvol kan zijn. Het single-rail IC blijkt volledig functioneel, zowel op een tester als in de experimentele DCC speler, waarmee werkelijk naar muziek geluisterd kan worden.

In vergelijking met de eerder gerealiseerde double-rail chip is de single-rail ver-

sie een derde kleiner, gebruikt hij de helft van de energie, en is hij een kwart sneller. Ten opzichte van de beste huidige synchrone tegenhanger is de single-rail proefchip ongeveer een zesde groter, maar gebruikt hij maar een zesde van de energie.

Single-rail handshake circuits lijken een aantrekkelijke manier om tegen geringe extra kosten energiezuinige ICs te ontwerpen. Vooral voor draagbare produkten, die door een batterij worden gevoed, kan een single-rail implementatie van digitale functies een interessante bijdrage leveren aan een langere levensduur, een lichter produkt, extra functionaliteit en, in het algemeen, tot een aantrekkelijker produkt.

Curriculum Vitae

Ad Peeters was born on February 26, 1966, in Dongen, the Netherlands. After attending the John F. Kennedy Atheneum in Dongen, he started his study Computing Science at Eindhoven University of Technology in September 1984. With an M.Sc. thesis on extensions to trace theory for expressing lifeness properties he graduated cum laude in September 1988.

Subsequently, he followed the two-year post-graduate designers course Software Technology at the Stan Ackermans Institute, Eindhoven University. As part of this course and inspired by the approach followed at Philips Nat.Lab., the design of delay-insensitive circuits was investigated. After completing the course in September 1990, he started to work towards a doctorate in the area of asynchronous silicon compilation.

When ESPRIT Project 6143 EXACT was initiated in July 1992, Ad started as a researcher on this project, actually working for Eindhoven University, but spending most of his time at Philips Nat.Lab., one of the partners in EXACT. Much of the research documented in this thesis was carried out during this period, which ended with the successful completion of EXACT in Summer 1995.

Since May 1995 Ad works as a research scientist in the IC Design Centre at Philips Nat.Lab. in Eindhoven, the Netherlands.

Current address

Philips Nat.Lab., Bldg. WAY-41
Prof. Holstlaan 4
5656 AA Eindhoven
The Netherlands
E-mail: apeeters@natlab.research.philips.com

Stellingen

behorende bij het proefschrift

Single-Rail Handshake Circuits

van

Ad M. G. Peeters

Technische Universiteit Eindhoven

juni 1996

1. Single-rail handshake circuits kunnen efficiënt worden gerealiseerd in synchrone standaard-cel bibliotheken. Met het toevoegen van een paar asynchrone cellen kan echter nog winst worden geboekt, in de zin dat circuits gemaakt kunnen worden die zowel kleiner, sneller, energiezuiniger als testbaarder zijn.

   [lit] Dit proefschrift.

2. De verlengde isochrone vork, in het bijzonder de asymmetrische variant daarvan, is een nuttig hulpmiddel bij het ontwerpen van asynchrone circuits. Het biedt de mogelijkheid om het gat tussen quasi-vertragingsongevoelig en data bundling stapsgewijs te dichten.

   [lit] Kees van Berkel, Ferry Huberts, and Ad Peeters, Stretching quasi delay insensitivity by means of extended isochronic forks. In *Proceedings of the 2nd Working Conference on Asynchronous Design Methodologies*, pp. 99–106. IEEE Computer Society Press, (1995).

3. De combinatie van een krachtige VLSI-programmeertaal, transparante vertaling en peephole optimalisatie is concurrerend met logische synthese.

   [lit] Dit proefschrift.

4. Asynchrone circuits zullen een steeds grotere rol gaan spelen in digitale VLSI systemen, zowel vanuit het oogpunt van laag energieverbruik als uit snelheidsoverwegingen.

5. Adiabatisch schakelen, meerwaardige logica en wave pipelining zijn onuitroeibare niches binnen het VLSI ontwerp.

6. Hoewel S. T. Dougherty in [2] beweert dat zijn bewijs van het niet bestaan van twee orthogonale latijnse vierkanten van orde zes anders is dan dat van D. R. Stinson in [1], zijn beide bewijzen op de volgorde van de argumenten na identiek. Ze zijn beide gebaseerd op lineare algebra over GF(2) en zijn bovendien, afgezien van het feit dat deelruimtes over GF(2) vaak lineaire codes worden genoemd, niet code-theoretisch van aard.

   [1] D. R. Stinson, A short proof of the nonexistence of a pair of orthogonal latin squares of order six, *Journal of Combinatorial Theory*, Series A, **36**, 373–376 (1984).

   [2] S. T. Dougherty, A coding theoretic solution to the 36 officer problem, *Designs, Codes and Cryptography*, **4**, 123–128 (1994).

7. Het is beter een bijdrage te leveren aan de ontwikkeling van zwakke economieën dan elke hulp te onthouden zolang nog niet alle mensenrechten in zo'n land worden gerespecteerd.

8. Het negeren van tijdsvariërende volatiliteit kan leiden tot significante afwijkingen van de veelgebruikte Black-Scholes formule voor het waarderen van financiële opties.

   [lit] F. Black and M. Scholes, The pricing of options and corporate liabilities, *Journal of Political Economy*, **81**, 637–659 (1973).

   [lit] R. J. Mahieu and P. C. Schotman, An application of stochastic volatility models. Submitted to *Journal of Applied Econometrics*, (1996).

9. De onder runderen voorkomende erfelijke afwijking 'paardenhoef' heeft als voordeel dat dieren met dit gebrek niet vatbaar zijn voor tussenklauwontsteking. Het verdient daarom wellicht aanbeveling om deze afwijking middels genetische manipulatie te cultiveren.

10. Het succes van de Nederlandse snijbloemenindustrie is voor een groot deel te wijten aan gesubsidieerde verwarming van kassen en accijnsloze brandstof voor vliegtuigen vanaf Schiphol. Samen met de Ethiopische aardbeien die we in het vroege voorjaar aantreffen in de supermarkten en de varkens die we voeren met soja uit hongergebieden om ze vervolgens levend naar Italië te brengen en ze daar te verslachten tot Parma-ham illustreert dit dat de kosten van brandstof onverantwoord laag zijn.

11. Bij de techniek van het hardlopen wordt de afstand van een voetplaatsing tot het zwaartepunt *rem*afstand (braking distance) genoemd. Dit suggereert ten onrechte dat een landing voor het zwaartepunt vermeden moet worden. Michael Johnson en Haile Gebrselassie weerleggen deze aanname op een ongeëvenaard wereldniveau.

12. Uit onderzoeken naar de relatie tussen fitheid en sportbeoefening blijkt gewoonlijk dat mensen gezonder zijn naarmate ze intensiever sporten. Bij een onderzoek in de Verenigde Staten bleek dat tieners die volgens het volwassenrecht zijn veroordeeld na hun detentie gemiddeld crimineler zijn dan tieners die volgens het jeugdrecht worden berecht (en dus gemiddeld lichtere straffen krijgen). Hoewel dit soort onderzoeken veel interessant cijfermateriaal opleveren definiëren ze eerder een kip-ei-probleem dan dat er harde conclusies aan verbonden kunnen worden.

13. De hogere aaibaarheid van het konijn ten opzichte van de muis hindert onderzoek naar het ontstaan van prenatale afwijkingen bij de mens.

14. Zowel in de industrie als in de sport wordt openheid over successen en mislukkingen, en dan vooral over de manier waarop deze zijn bereikt, onderschat als instrument ter verbetering.