Computing without Clocks:
Micropipelining the ARM Processor

Steve Furber
Department of Computer Science, University of Manchester
Oxford Road, Manchester M13 9PL, England.

Abstract

High-performance VLSI microprocessors are becoming very power hungry; this presents an increasing problem of heat removal in desk-top machines and of battery life in portable machines. Asynchronous operation is proposed as a route to more energy efficient computing. In his 1988 Turing Award Lecture, Ivan Sutherland proposed a modular approach to asynchronous design based on "Micropipelines". The AMULET group at Manchester University has developed an asynchronous implementation of the ARM microprocessor based on micropipelines as part of a broad investigation into low power techniques. The design is described in detail, the rationale for the work is presented and the characteristics of the chip described. The first silicon from the design arrived in April 1994 and an evaluation of it is presented here.

1 Motivation

The motivation for the work described in this chapter stems from the need for approaches to the design of VLSI devices which result in chips with lower power consumption, and from the potential which asynchronous logic appears to have to meet this need.

1.1 Power Trends

VLSI techniques with lower power consumption are needed for at least two different reasons.

Firstly, the portable equipment market is growing rapidly. Products such as laptop computers and personal digital assistants rely on batteries for power, hence battery life is an important specification point. Mobile telephones are increasingly using significant computational functions, and again battery life is a key feature in the marketplace.

Secondly, the trend in CMOS VLSI is towards very high dissipation. Chips like the DEC Alpha and the TI Viking SPARC illustrate the trend towards unmanageable power consumption where 20 to 30 watts is not unusual for a high-performance CMOS processor today. If current design practice were to continue, by the year 2000 we would expect to see a 0.1µm 5V CMOS processor dissipating 2kW! Of course, there are trends towards lower supply voltages which reduce the power somewhat, but 3v (and later 2v) operation only reduces power by a factor of 3 (and then 6). More is needed if the technology of the next century is not to have its performance potential compromised by power issues.

The desirability of lower power consumption is now widely recognised and many conferences have sessions and keynote speeches on the subject. There have been workshops dedicated to low power, and manufacturers of commercial processors are incorporating power management features into their products. The goal of more power-efficient design is not contentious; what is contentious is the proposition that asynchronous logic may have a role to play in achieving this goal.

1.2 Asynchronous Logic

There has recently been a resurgence of interest in asynchronous logic design techniques, which had for two previous decades been largely neglected by designers. One reason for this renewed interest is the observation that synchronous logic is beginning to run into serious limits. As VLSI devices incorporate increasing numbers of transistors it is becoming increasingly difficult to maintain the global synchrony on which synchronous designs depend. Clock skew is already a problem at board level, and is increasingly becoming a problem on a single chip. Most notable amongst recent victories over clock skew is the DEC Alpha [1], where the clocking system is a major feat of engineering and the statistics of the clock drivers (such as the proportion of silicon area they occupy, the sizes of the transistors, the peak currents, etc.) are viewed with some awe by most ‘normal’ designers. But the fact that these feats were needed on the Alpha serves here to demonstrate the point that synchronous design is beginning to approach its limits.

Asynchronous logic abandons global synchrony in favour of locally generated timing signals and is therefore unaffected by clock skew. It also displays a number of specific advantages over clocked logic, particularly when low power consumption is an objective:

- Power is only used to do useful work. In synchronous design the clocks are applied to all units whether or not they are doing anything useful. Recent developments in power management may gate off clocks from some areas of the design, but these typically are applied at a coarse granularity. Straightforward asynchronous design only activates units when they are required to do useful work.

- Designs can be optimised for typical conditions. When timing is defined by a fixed clock, all operations must complete within a fixed period. Synchronous designs therefore expend considerable silicon resource on making rare worst-case operations fast. This usually results in complex circuits which use more power than is necessary under typical conditions. An asynchronous design can allow worst case operations to proceed more slowly and focus the use of resource on operations which occur frequently.

Therefore asynchronous design appears to have considerable potential for power-efficient design. However, asynchronous design is not new; in the early days of computing many designs used asynchronous approaches, but these were later abandoned in favour of synchronous styles because of the inherent difficulties of asynchronous operation. Are these difficulties still an obstacle to a wide acceptance of asynchronous logic?

This question remains to be answered. The following observations may go some way towards addressing it:

- Synchronous designs are getting increasingly complex and, where power-efficiency is important, clock gating is used with increasingly fine granularity. In the limit, clock gated synchronous circuits look very like fully asynchronous circuits, so perhaps this complexity is unavoidable where power-efficiency is a goal.
- The clock in a synchronous circuit has a role similar to a global variable in a computer program. Object-oriented programming styles are increasingly accepted as ways to increase programmer productivity by precluding global variables and encapsulating data. Asynchronous logic styles with well defined interfaces have many of the properties of object-oriented programming styles and may show the same benefits in designer productivity for systems above a certain level of complexity.
- Mathematical techniques are being developed and applied with increasing success to address some of the difficulties inherent in asynchronous design, such as proving circuits are free from deadlock.

Whilst the above arguments do not constitute conclusive proof that asynchronous design has advantages over clocked design sufficient to justify an immediate abandoning of synchronous styles by all designers, they do at least suggest that asynchronous design is worth investigating again to see to what extent its potential for power-efficiency can be realised. The AMULET group at the University of Manchester was established late in 1990 to carry out such an investigation. This chapter documents the first work of this group, the design and evaluation of AMULET1, an asynchronous implementation of the ARM microprocessor.

The chapter begins with a description of Sutherland's micropipelines, which were used as the basis for the asynchronous design style employed on AMULET1. Details of the event control cells which manage the interactions between micropipelines are presented along with discussions of various low-level VLSI design issues. The ARM processor is introduced and the asynchronous organisation of AMULET1 described. The results of the evaluation of the first silicon are presented, and the chapter closes with a discussion of the conclusions which may be drawn from the work so far and suggested directions for future work.

2 Sutherland's Micropipelines

The asynchronous design style used in AMULET1 is based on Sutherland's Micropipelines [2] which employ a 2-phase bundled data interface for sending data between functional units. 2-phase (or transition) signalling uses both rising and falling edges in turn to signal the same event; rising and falling edges are equivalent and carry the

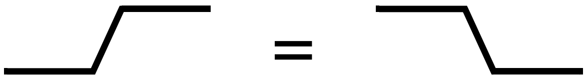


Figure 1: Transition signalling

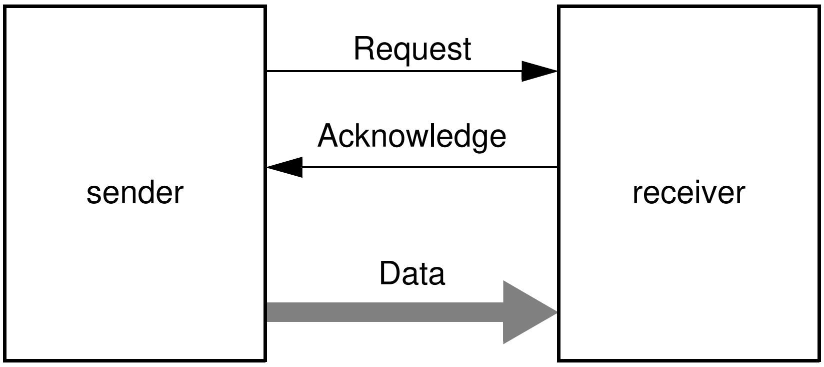


Figure 2: A bundled data interface

same information (figure 1).

A bundled data interface passes a binary value encoded conventionally on a bus from sender to receiver along with a *Request* wire which indicates when the data is valid and an *Acknowledge* wire which indicates when the data has been received (figure 2).

The communication protocol is illustrated by the timing diagram in figure 3. A valid data value is placed on a conventional bus by the sender which then indicates the availability of the data by causing a transition on the *Request* wire. The receiver senses this transition, accepts the data and then causes a transition on the *Acknowledge* wire, completing the transfer. The sender may then issue another data value in a similar manner. Note that only the order of these events is significant; the delays between them are arbitrary (though long delays will, of course, reduce performance). Also note that the Request and Acknowledge wires use 2-phase signalling; rising and falling edges are both significant and have the same meaning.

In order to design circuits based on micropipelines, a few event control blocks are required. The basic set of event control blocks proposed by Sutherland is illustrated in figure 4.

The *Muller C-gate* performs the rendezvous function for events; it waits until it has received an event on both of its inputs before issuing an event on its output.

The *XOR* (exclusive-OR*)* gate performs the merge function for events; a transition on either input results in a transition on its output.

The *toggle* cell transfers an event from its input to its two outputs alternately; the first event to arrive is issued to the output marked with a dot, the second to the unmarked output, and so on.

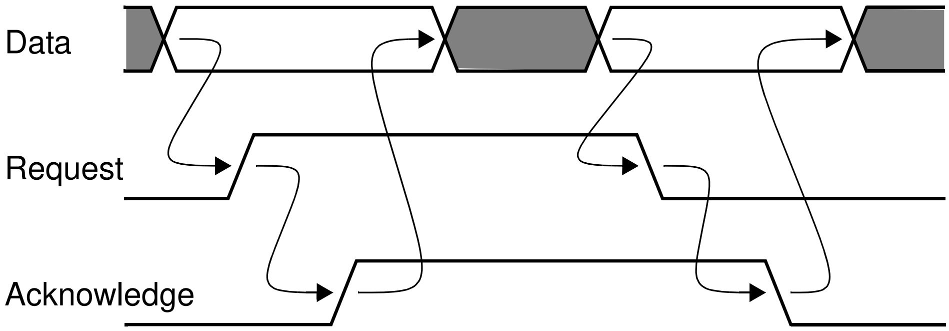


Figure 3: The 2-phase bundled data convention

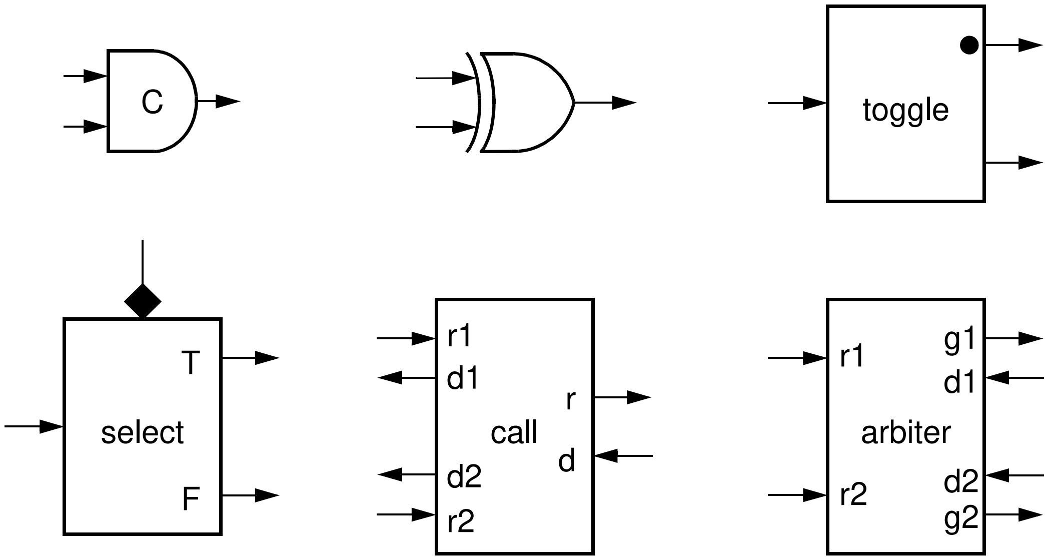


Figure 4: Event logic modules

The *select* block allows a boolean, usually derived from the binary data, to direct the input event to the True or False output.

The *call* block allows two independent processes to share a common subprocess; requests on r1 and r2 are routed through to r, and the completion signal on d is routed back to the caller. The calling processes must be mutually exclusive; if they are not, they must access the call block through an arbiter.

The *arbiter* accepts asynchronous requests on r1 and r2 and grants only one of them at a time on g1 or g2. When the shared resource is given up (signalled by an event on d1 or d2) the arbiter will then grant the other request if it is pending. The arbiter must be designed to allow for metastability in its internal circuitry, but if this is done correctly reliable operation is possible.

In order to construct a micropipeline circuit, it is necessary to employ event con-

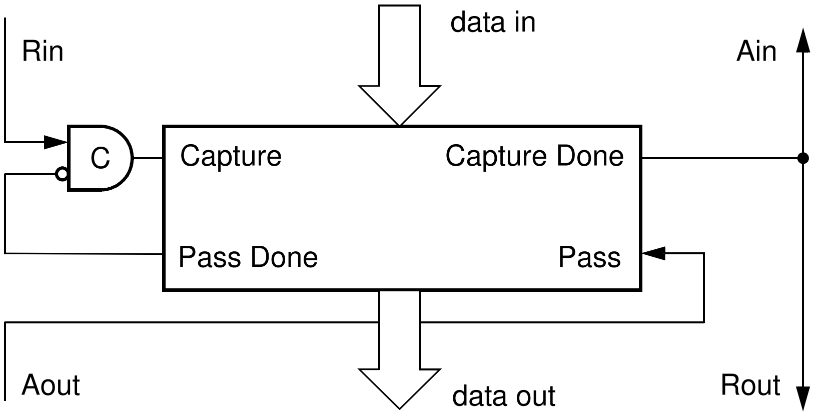


Figure 5: A micropipeline event driven register

trolled latches to hold the data stable. The high-level view of a micropipeline latch is shown in figure 5. The latch begins in a transparent state. The C-gate has a ‘bubble’ on the lower input, indicating that in its initial state this input is primed (as though an event has already arrived). An event on the input request wire (Rin) indicates that the input data is valid. This event passes through the C-gate to the latch ‘capture’ input, causing the data to be latched. When the latch has captured the data it issues an event on ‘capture done’, which is copied back as the input acknowledge (the sender may now remove the data) and forward as the output request (the output data is now valid). The output data is held stable until an acknowledge is received from the output channel, whereupon the latch is put back into ‘pass’ (transparent) mode and the C-gate re-primed ready for the next input request. Note that this request may already have arrived; the C-gate waits until the output has acknowledged and the input requested before firing and thereby ensures the correct operation of the latch whatever the relative timings on the input and output sides.

The event register in figure 5 can be replicated to form a first-in first-out (FIFO) buffer as shown in figure 6. A data value can be placed into the left event register by signalling an event on Rin, whereupon it will be passed along to the right register at a speed determined only by the properties of the logic technology from which it is constructed. The right register will then issue an event on Rout and hold the data stable until it receives an acknowledge on Aout. In the meantime further values may be entered from the left until the FIFO is full (i.e. each event register holds a data value), whereupon the first stage will not acknowledge the last value entered and the input process will stall until a value is removed by the output process.

The FIFO has two important parameters which define its performance:

- Latency - the speed at which new data passes through an empty FIFO. This parameter may not be significant in some applications where a steady stream of values flows through the FIFO. But when micropipelines are applied to the design of a microprocessor, the FIFO is likely to be flushed from time to time and then the latency will influence the speed with which a new processing thread may be established.

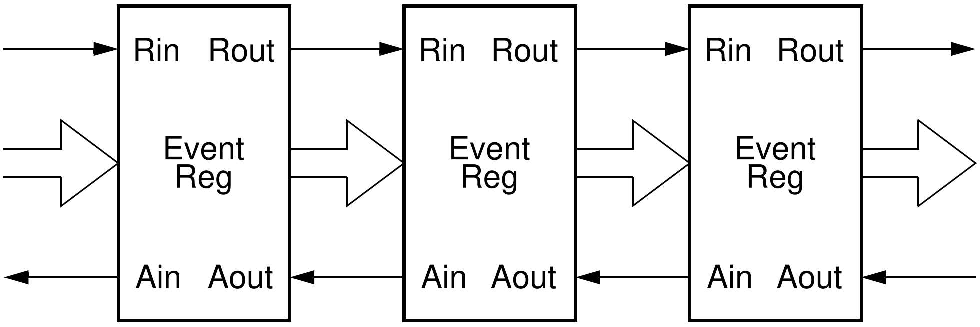


Figure 6: A FIFO - the canonical micropipeline

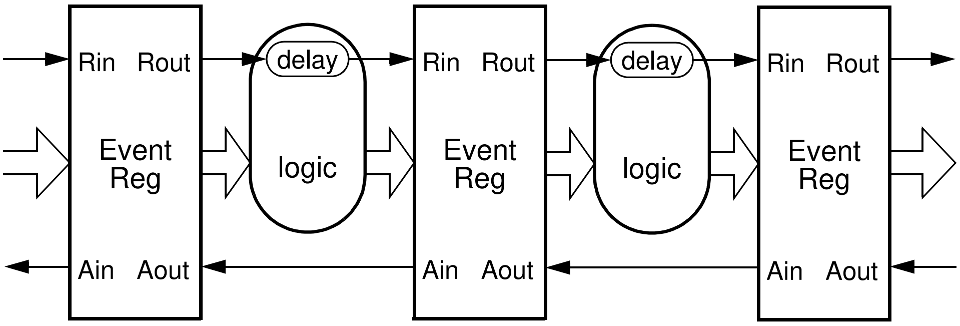


Figure 7: A FIFO with processing

- Throughput - the maximum sustainable data rate. This parameter plays the same role as the maximum clock rate in a synchronous pipeline.

One of the features of the micropipeline FIFO is its elasticity. Data can be entered at any rate up to a maximum defined by the throughput parameter and limited only by the FIFO becoming full, and it can be removed at any rate up to a similar maximum limited only by the FIFO becoming empty. Within these limits the structure will buffer a variable number of data values and pass new input values to the output at a speed limited only by the process technology. Compare this with a clocked pipeline where the input and output are regimented by the clock, and the input to output delay is a fixed number of clock cycles.

The micropipeline FIFO is fine as a buffer, but performs no logical processing on the data that flows through it. By interspersing logic between the event registers in a FIFO, a micropipeline with processing functions can be built (figure 7). The logic in the data route incurs some delay in the arrival of the input data at the next register, so a corresponding delay must be introduced into the request line to ensure that the bundling constraint is met at the input of the next register. This delay element may be implemented in a number of ways; examples on AMULET1 include:

- A 33rd register read bit which always makes a transition whenever a register is read. This 33rd bit is used to signal completion of the register read process.

- A data dependent delay which follows the longest carry path in the ALU. The time which must be allowed for the ALU to complete depends on the data values being processed (and the function being evaluated; logical functions are much faster than addition and subtraction). The delay which is introduced to match the ALU delay similarly is a function of the operand values, giving a data dependent delay.

The need to produce matched delays in micropipelines is a source of concern to many designers of clocked chips. Whilst not wishing to underplay the difficulties inherent in this approach, it should be noted that such delay matching is not uncommon in current synchronous designs. CMOS PLAs frequently use self-timed paths to allow dynamic operation and to eliminate DC currents, and SRAMs use self-timed paths to turn off sense amplifiers in order to save power. Delay matching is not particularly difficult in a full-custom design environment where the tolerances of the target process are well understood, but it becomes increasingly problematic where process portability is important, or where automatically laid out standard cell design is to be used, or where field programmable gate arrays are the target implementation technology. The problem can be solved in all of these cases, but usually at the cost of increasing margins and therefore compromising performance.

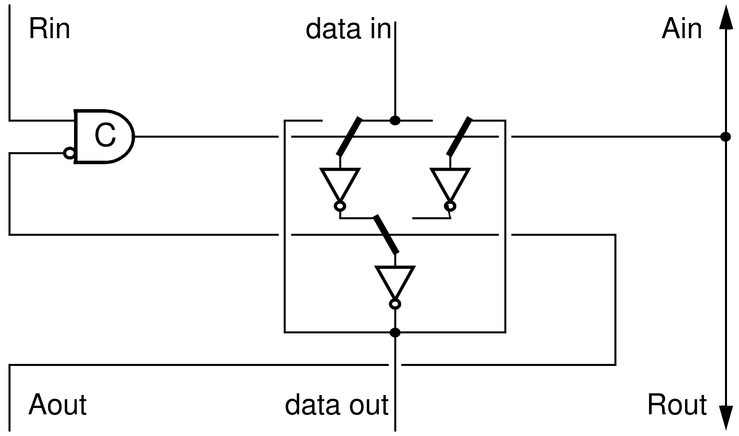


Figure 8: Sutherland's 'capture-pass' latch

3 Micropipeline Latch Structures

Sutherland proposed a latch for use in micropipelines which operates directly from the 2-phase capture and pass signals. The principle of this latch is illustrated in figure 8. Here the latch is shown in its initial, transparent, condition. An event on the capture wire (the output of the C-gate) switches the upper two multiplexers over, holding the data in a feedback loop and isolating the output from the input. A subsequent event on the 'pass' wire (from Aout) switches the output multiplexer over, reconnecting the input through to the output.

Note in this figure how the event wires pass across the latches so that the capacitive load of the latches affects the speed of operation of the control circuits. This is a common factor in all the latch designs presented here and allows the event paths to 'measure' the latch loading to ensure correct operation under all conditions by preventing the control circuits from operating faster than the latch control wires can switch.

AMULET1 does not use the Sutherland capture-pass latch. Instead, conventional level-sensitive transparent latches are used with a more complex control circuit as shown in figure 9. Here the XOR gate and toggle operate as a 2-phase to 4-phase conversion circuit. After initialisation the latch is transparent. A capture event from the C-gate closes the latch and the toggle steers that event to Rout and Ain. The pass event from Aout puts the latch back into its transparent state, then the toggle steers the 'pass done' event back to re-arm the C-gate. Note again how the events which change the state of the latch are 'measured' to ensure that the control circuits wait for the latch to complete its operation.

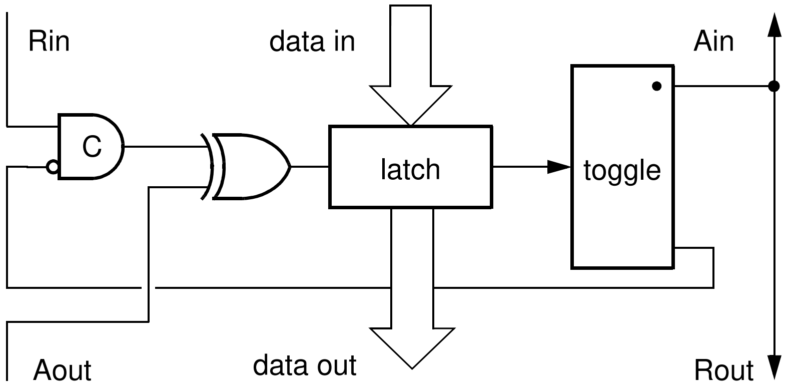


Figure 9: An event latch based on a conventional transparent latch

This latch was used in preference to the 'capture-pass' latch mainly because it is more area efficient for wide buses than Sutherland's latch. Its operation is very safe with respect to bundling constraints. Input data which changes very close to its request will be latched with a good set-up time into the latches since the C-gate, XOR and latch enable line delays all increase the input margins. Rout is delayed further by the latch enable and toggle delays so the output bundle is well margined.

The latch has a latency (Rin to Rout time) of around 10ns on a 5V 1µm CMOS process (worst case temperature, power supply and process parameters) and when several such latches are built into a FIFO structure the bandwidth corresponds to a latch cycle time of around 30ns.

The latch circuit shown in figure 9 is widely used on AMULET1. There are some places, however, where its latency is too high. One such place is the instruction prefetch buffer. When a branch is taken this buffer is flushed and the branch target instruction must pass through 5 latch stages in this buffer before it can begin execution, which at 10ns per stage would add significantly to the branch cost. In such cases a slightly modified 'fast forward' form of the latch is used. This latch control struc-

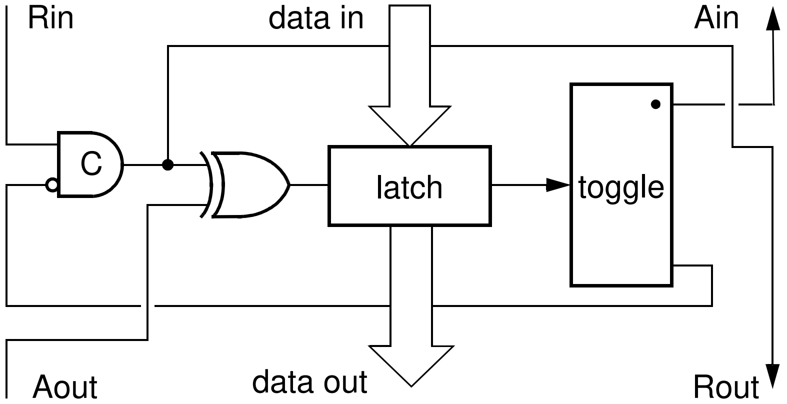


Figure 10: A ‘fast forward’ event latch

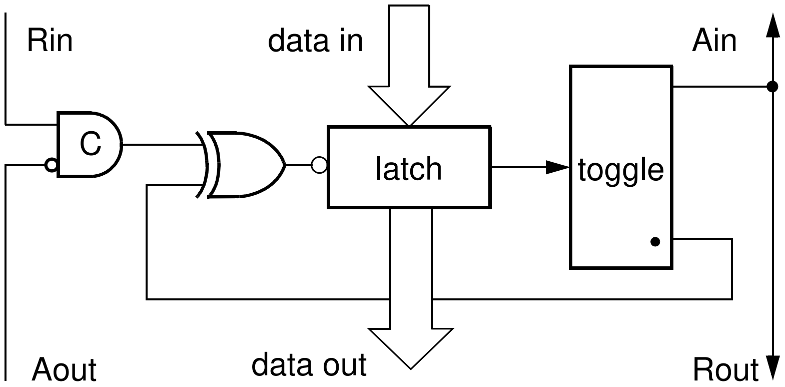


Figure 11: A blocking (normally closed) event latch

ture is shown in figure 10.

The change to the control circuit is to take Rout directly from the C-gate output rather than from the toggle. This is allowable (under certain conditions) since Rout indicates merely that the output data is valid; it need not be delayed until the latches have closed. This latch has a faster forward propagation time (around 2ns Rin to Rout) than the standard latch, giving the 5-stage instruction buffer a refill latency of 10ns rather than the 50ns it would have using standard latches.

The fast-forward latch is less safe on bundling constraints and requires the latch input to output time to be no slower than the C-gate delay. It also requires the Rout to Aout delay to be above a minimum value. This is readily satisfied when several of these latches are composed into a FIFO; under these conditions the latch operates safely and has a cycle time around 20ns.

All the above latches are transparent when empty, so transients on the input data buses will propagate through the latches, causing power to be dissipated uselessly down the pipeline. In some cases it is highly desirable to eliminate this effect in order to save power, and indeed this may be achieved by inserting a ‘blocking’ (normally closed) latch at the top of the pipeline. A control circuit for a blocking latch is shown in figure 11.

The blocking latch uses the first event from the C-gate to open the latches, letting the data through to the output. The toggle returns this event to the XOR, where it then closes the latch and is passed through the toggle to Rout and Ain. Aout simply re-primes the C-gate. This latch is very safe with respect to the bundling constraints, but is rather slower than the previous latches with an Rin to Rout time of around 20ns on 1µm CMOS (worst case) and around 40ns cycle time when built into a FIFO.

4 Event Control Elements

The basic library of event control elements proposed by Sutherland was introduced in figure 4. In this section the transistor structures used for these elements in AMULET1 will be introduced and relevant design issues raised.

The circuit used for the 2-input Muller C-gate is shown in figure 12. Series transistor stacks pull the internal node high or low when both inputs are at the same logic level and the weak feedback inverter retains the previous state when the inputs are at differing levels. Since the input levels are unknown at initialisation, reset circuitry (connected to *Cdn*, the active-low ‘clear-down’ signal) is required to force the output to zero independently of the inputs.

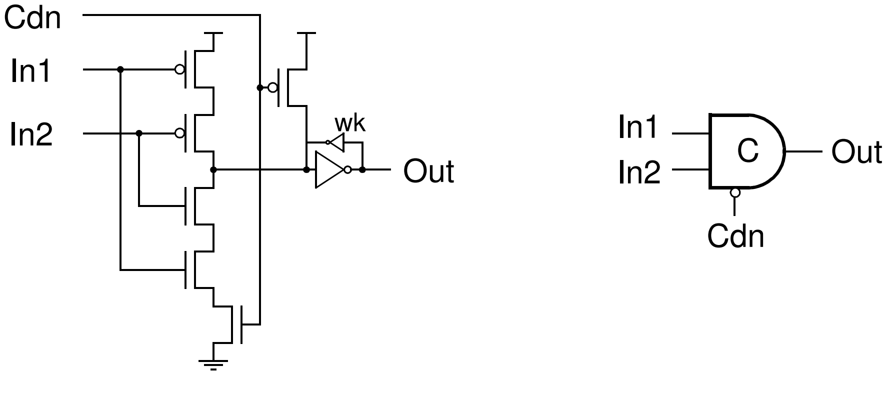


Figure 12: Muller C-gate schematic and symbol

This circuit is similar to Sutherland’s dynamic C-gate, with the addition of the weak feedback inverter to give fully static operation. It should be noted that this circuit has unusual logic thresholds on its inputs since when one input is switching the other input will turn off one of the series transistor stacks. The switching threshold for a rising input is therefore V_{tn}, and for a falling input V_{tp}. The C-gate therefore switches early on the transition of the input and may detect the transition significantly before another gate connected to the same signal. To avoid this causing any problems it is advisable to ensure that all event signals have fast rising and falling edges.

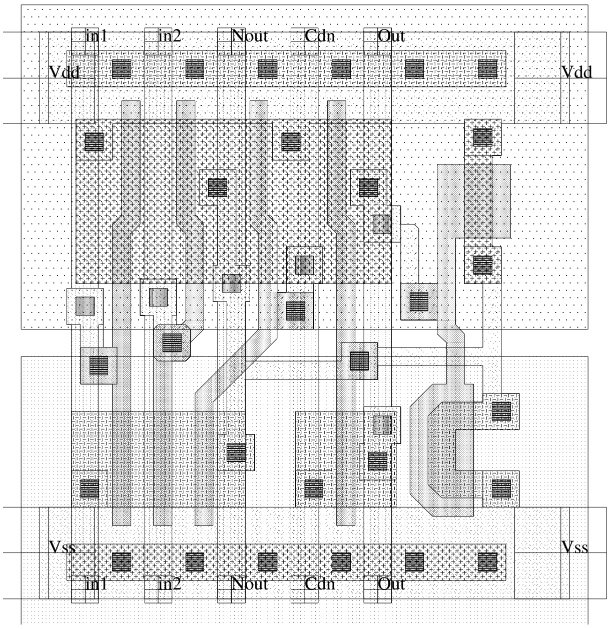
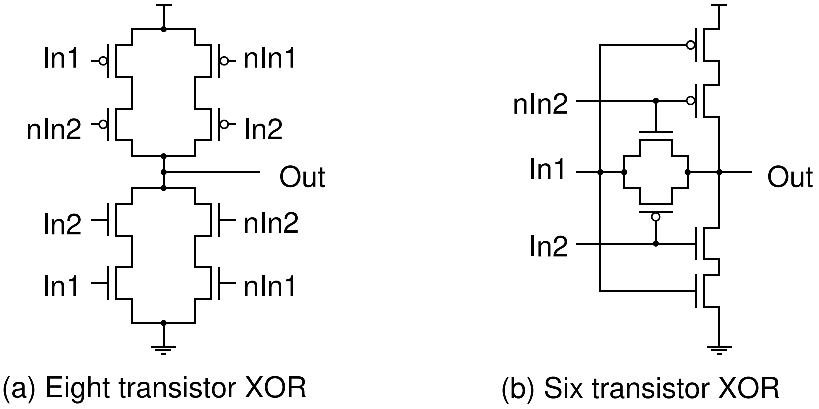


Figure 13: Muller C-gate layout

The observant reader will have noted that whilst all the introduction to micropipelines referred to transitions rather than levels, the implementation of the C-gate is clearly level sensitive. This is best understood by observing that provided the C-gate inputs are initialised to the same level (zero), they must thereafter each make the same number of transitions and must therefore retain the same level relationship, allowing a level-sensitive implementation of the C-gate. This also allows the primed version of the C-gate (as used in all the latches) to be implemented by simply adding an inverter to the 'bubble' input of a standard C-gate. The VLSI layout of the Muller C-gate is shown in figure 13

Two circuits were considered in detail for the XOR gate; these are both illustrated in figure 14. The most compact layout is achieved with the 6 transistor design. However this design exhibits 'charge-sharing' under certain input conditions, causing glitches on In1. These glitches are not large, but any glitch on an event wire is a potential source of erroneous operation (particularly if the same wire is connected to a low threshold device such as a C-gate) so all glitches must be thoroughly investigated. There is a danger that, in using circuits which are known to produce glitches, the designer will neglect to investigate other glitches which are the source of real problems. Therefore the 6 transistor circuit was rejected in favour of the larger 8 transistor design which is not prone to glitching (but which also requires the inverse of both its inputs).

A circuit which does not appear in Sutherland's cell library is a transparent latch for events. It is used as a building block for other event control elements where its basic role is to block events, and as it may be closed at initialisation it requires reset circuitry to ensure the output is initialised to zero. (Transparent latches for data are simpler as they do not require reset circuitry; they will be introduced later.)



(a) Eight transistor XOR

(b) Six transistor XOR

Figure 14: Alternative XOR schematics

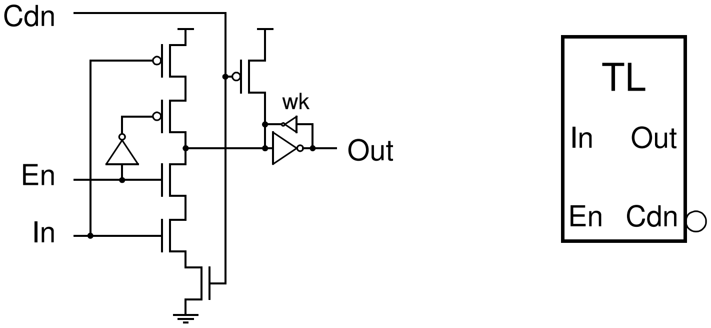


Figure 15: Transparent latch schematic and icon

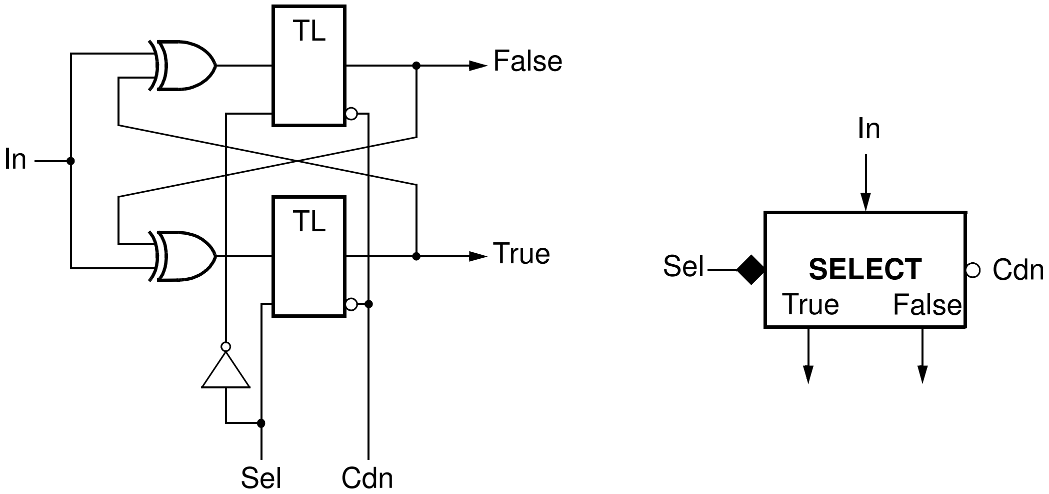


Figure 16: Select block schematic and icon

The circuit used for the event transparent latch is shown in figure 15 along with the icon used to represent this schematic at higher levels of the design. The circuit is similar to that for the C-gate, comprising a tri-state gate, a weak feedback storage node and reset circuitry.

The select block schematic and icon are shown in figure 16. This circuit uses the event transparent latch to control the flow of the input event to one output or the other. The boolean control input selects the path to be open; the input event passes to the inputs of both latches, through the open latch, then it returns to cancel the event on the input of the closed latch. The boolean input must not change near to the time at which an input event occurs.

Another element which does not appear in Sutherland's cell library but which is a useful building block is the 'decision-wait'. This circuit is a generalisation of the C-gate, performing a rendezvous between one event line and either of a pair of event lines. A circuit schematic and icon are shown in figure 17. Events can arrive in any order on the 'fire' input and either 'A1' or 'A2' (but not both), then an output event is generated on 'Z1' or 'Z2' according to which 'A' input made a transition. Once the output has fired the circuit is ready to perform the next rendezvous between 'fire' and either the same or the other 'A' input. The circuit is based around two C-gates; the 'fire' input is applied to both, then an 'A' input fires, generating the corresponding 'Z' output and cancelling the 'fire' input on the other gate via the XOR. This technique of 'removing' an event which has already arrived works because the C-gate implementation is level rather than transition sensitive.

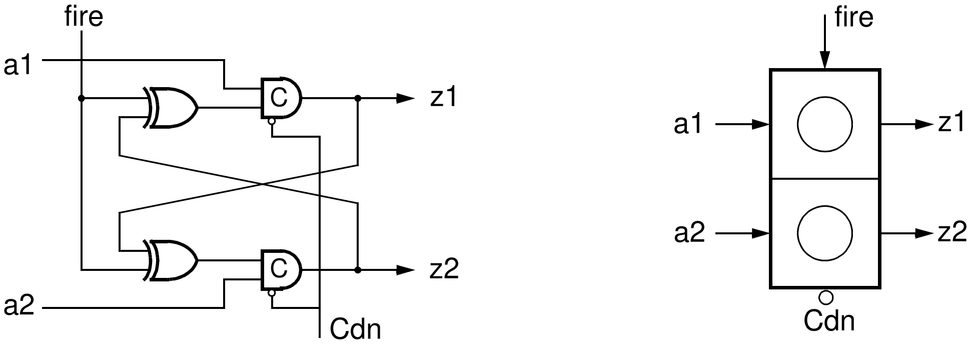


Figure 17: Decision-wait schematic and icon

The decision-wait can be composed with an XOR gate to construct the Call block as shown in figure 18. The two request inputs are merged through the XOR to produce the request output and the decision-wait is used to store the identity of the source of the active request. The 'done' response is then steered by the decision-wait back to the appropriate calling process. The layout of the complete Call block is shown in figure 19

The toggle appears, in principle, to be quite simple to implement. In practice, this was the most problematic of all the event control blocks because the circuit contains an inherent race hazard. The principle of the toggle implementation is shown in figure 20. The input event alternately opens and closes two latches in anti-phase, allow-

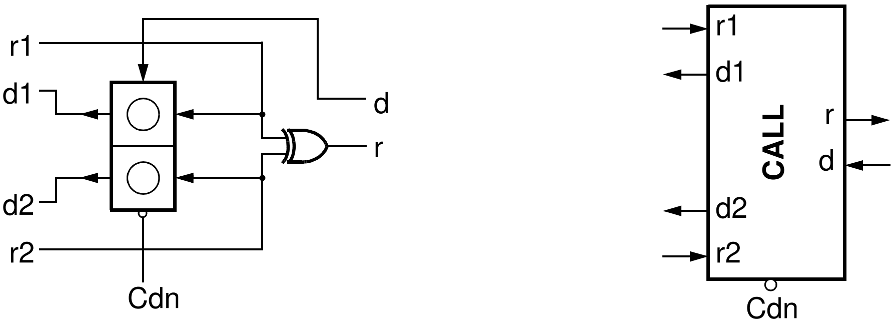


Figure 18: Call block schematic and icon

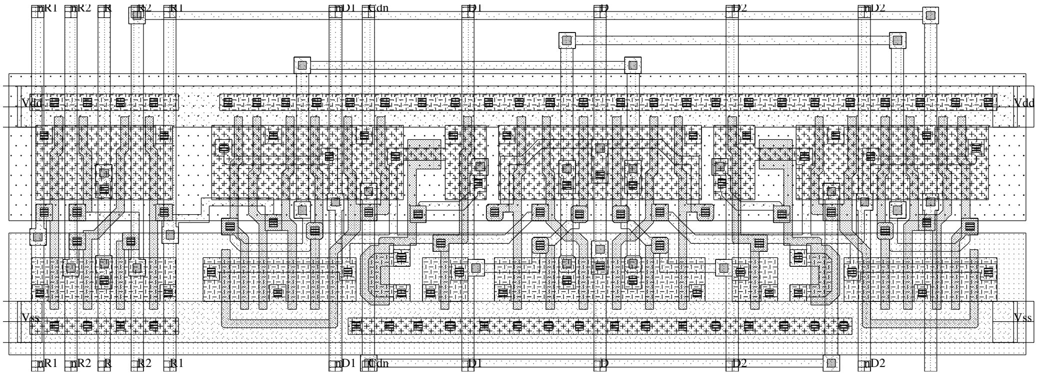


Figure 19: Call block layout

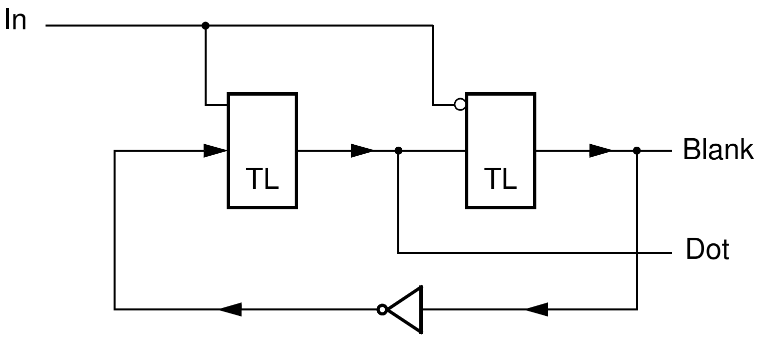
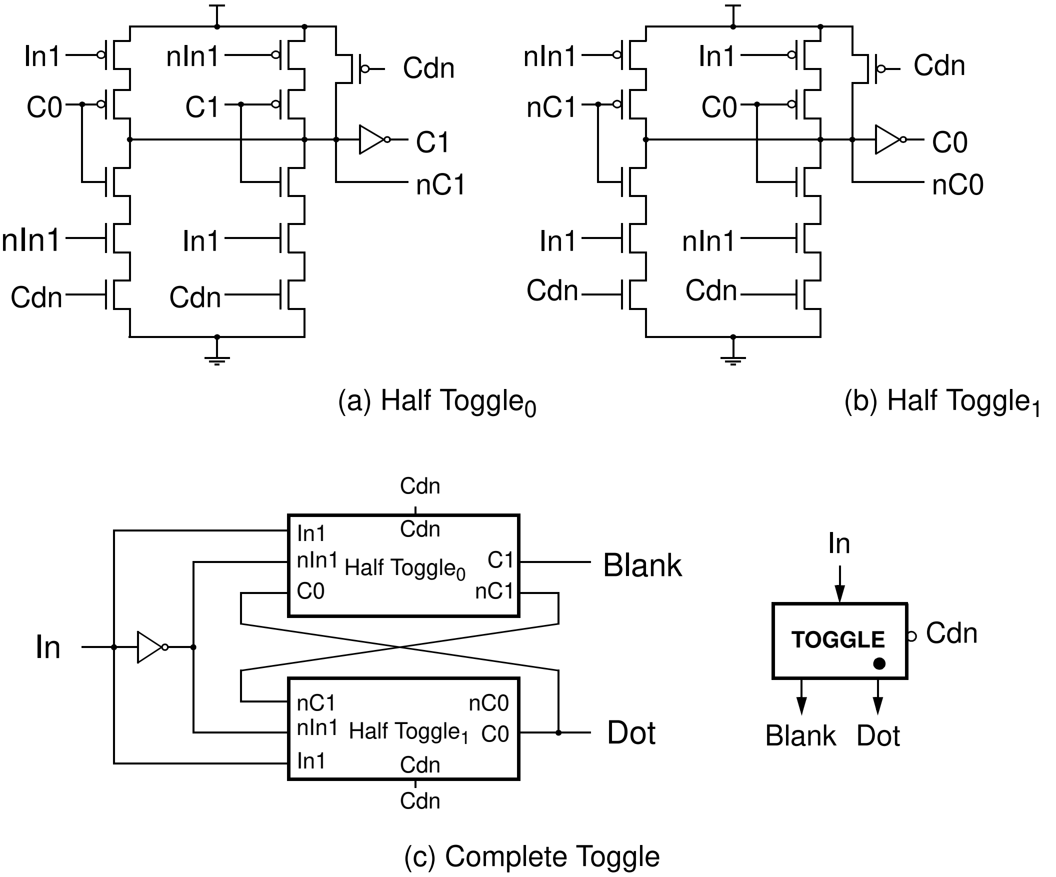


Figure 20: Toggle principle

ing a boolean value to go round a closed loop with one inversion each time round the loop. An implementation must take into account the finite rise and fall time of the input and ensure that there is no point in the cycle where both latches are open at the same time. A toggle which is to be used from a cell library must operate correctly over a wide range of input and load conditions, and early implementations of the principle of figure 20 required a non-overlapping two-phase clock generator (built into the cell) to ensure safe operation. A more successful design was based on the principle of the 6 NAND gate TTL D-type latch, but the final design used in AMULET1 was developed by Jay Yantchev (at Oxford) using an algebraic approach to derive an implementation from the specification.

The circuit of the Yantchev toggle is shown in figure 21. This circuit follows the principle illustrated in figure 20 closely, but succeeds in minimising the race problem by carefully interlocking the two latches. Even so, the circuit can fail if the input and its complement are not carefully aligned to each other, but alignment can be ensured by building the inverter which generates the complement into the cell itself rather than allowing the external circuit to produce it; under these conditions the circuit is very robust. It is interesting to note, in passing, that the Yantchev toggle is the most significant contribution from 'formal' approaches to AMULET1.



(a) Half Toggle$_0$

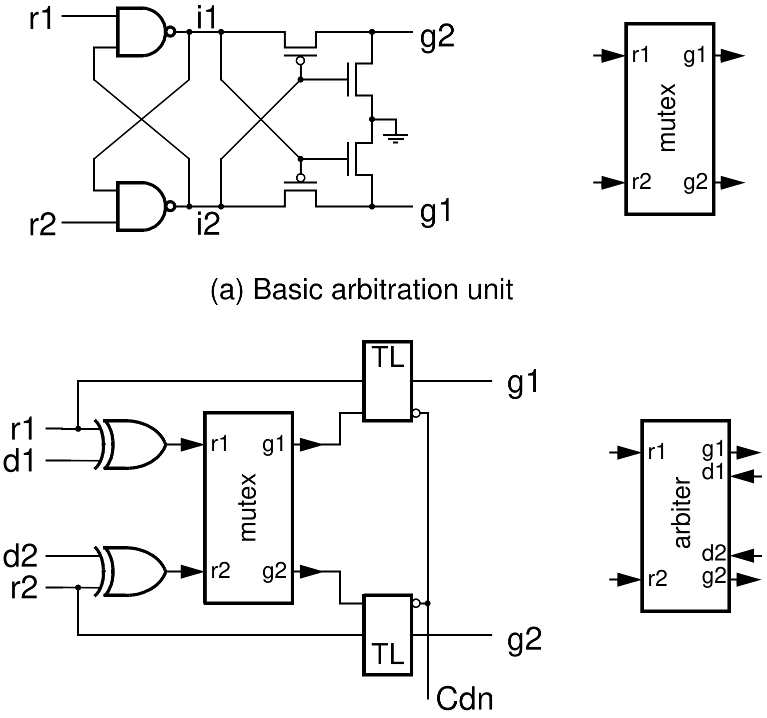
(b) Half Toggle$_1$

(c) Complete Toggle

Figure 21: The Yantchev toggle circuit

The arbiter is required to ensure the mutual exclusivity of the grants to two asynchronous requesting processes. As this amounts to making a discrete decision on a continuous set of possible input conditions, it is fundamentally impossible to guarantee that the decision will be made within a bounded time. The circuit must be designed to cope with internal metastability under a range of input conditions. A suitable circuit schematic is shown in figure 22. The basic arbitration unit is a 4-phase level-sensitive mutual exclusion element with a cross-coupled pair of NAND gates forming an R-S flip-flop. The request inputs are normally low, but switch high to indicate an active request. If both switch high at the same time, the R-S flip-flop may go metastable. The output circuit will not, however, issue either grant until the difference between the internal nodes i1 and i2 exceeds V_{tp}, by which time the R-S flip-flop has exited from its metastable state.

The micropipeline (2-phase) arbiter is constructed from the mutual exclusion element by adding suitable 2-phase to 4-phase conversion circuits. The circuit shown in figure 22 is suitable provided there is a reasonable delay between the ‘done’ input and a subsequent ‘request’ on the same input.



(a) Basic arbitration unit

(b) Micropipeline arbiter

Figure 22: Arbiter schematics

It is worth noting here a fundamental difference between synchronous and asynchronous circuits. In a synchronous circuit the designer must accept some residual probability of failure whenever an asynchronous input enters the clocked domain of the circuit. This probability can be made very small, at the cost of reduced performance, but can never be zero. The designer of an asynchronous circuit can, on the other hand, use an arbiter which will never fail, though it may take a very long time to make a decision. If the system in question is, for example, a flight control computer, it would make little difference to the unfortunate passengers whether a failure is due to synchronization failure in a clocked system or terminal indecision in an asynchronous system. But note that the synchronous system must accept the worst-case cost of synchronizing to the desired level of reliability on every asynchronous sampling process, whereas the asynchronous system typically incurs the average cost and only very rarely comes near to the worst-case cost.

The circuit used to construct pipeline data latches on AMULET1 is shown in figure 23. This circuit is the same as that used on ARM6 for data latches, comprising a transmission gate which, when enabled, overdrives the weak feedback which at other times retains the state on the forward inverter. A practical control circuit is illustrated in figure 24, which shows the buffer circuits used to drive 32 latch loads and a C-gate used to detect that both the latch enable and its complementary signal have switched before firing the toggle.

To illustrate the use of some of the event control cells that have been introduced, circuits are shown below for a conditional pipeline fork (figure 25) and a conditional pipeline join (figure 26). The first of these shows the last stage of one pipeline which

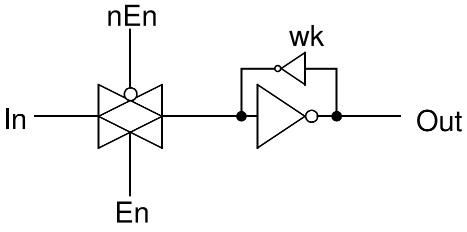


Figure 23: Data latch circuit

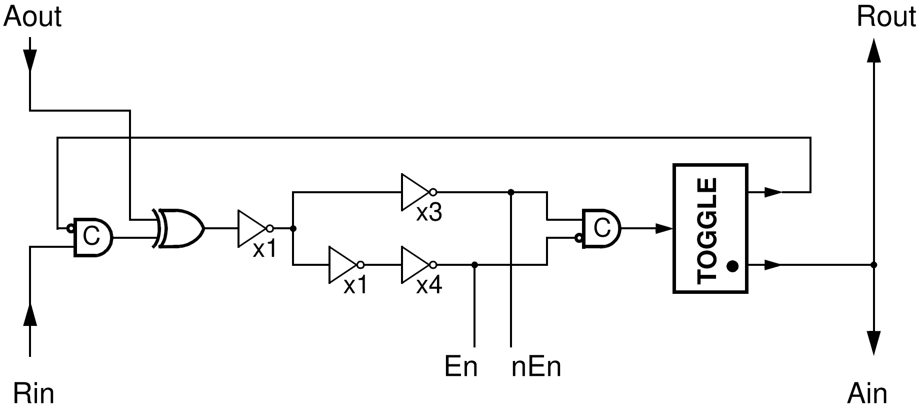


Figure 24: A practical event register control circuit

always sends data down the left output pipeline, but only sends data down the right output pipeline if a boolean in the current data value indicates that it should. The select block implements the decision by steering the event appropriately. Note how the 'False' select output event just bypasses the request to the right pipeline and that both output pipelines are called in parallel, with a C-gate used to wait until both have completed. (A simpler, but slower, circuit could be used to call first the left pipeline and then, conditionally, the second. This would save the cost of the C-gate.)

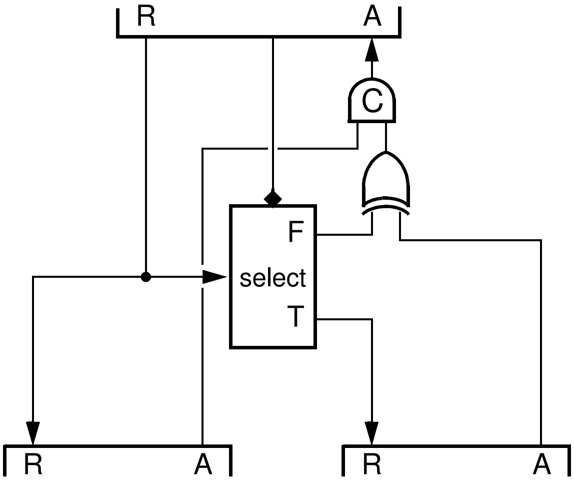


Figure 25: Conditional pipeline fork circuit

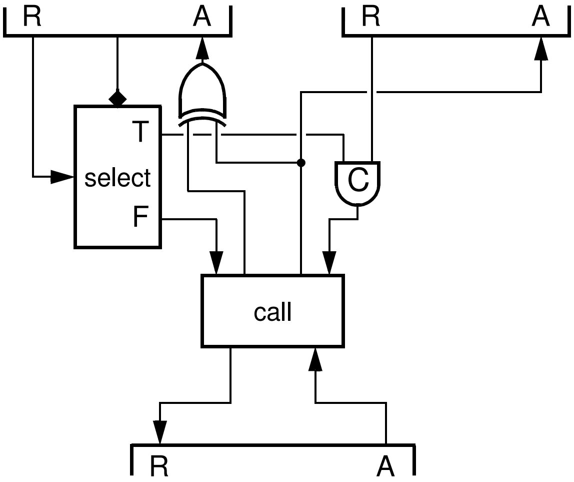


Figure 26: Conditional pipeline join circuit

The conditional pipeline join circuit has a boolean in the left input pipeline which determines whether or not there should be a rendezvous with the right input pipeline before the output pipeline is called. Again, a select block makes the decision by steering its input event. One output uses a C-gate to rendezvous with the second input pipeline, the other bypasses the rendezvous. These two events could now be merged through an XOR to form the request to the output pipeline, but that would leave the problem of handling the acknowledge properly, which may or may not need to go to the right hand input pipeline. A select block on the acknowledge path could be used to achieve the required effect, but rather than taking the same decision again the circuit in figure 26 uses a call block to remember the original decision and steers the acknowledge accordingly.

4.1 Cell Areas

The areas of various cells are shown in figure 27, with that of a standard inverter for reference. Note that although it is possible to extend the number of outputs from a select block, this leads to a very large cell size, so in practice selects were limited to two outputs and several 2-way selects were cascaded where more outputs were required.

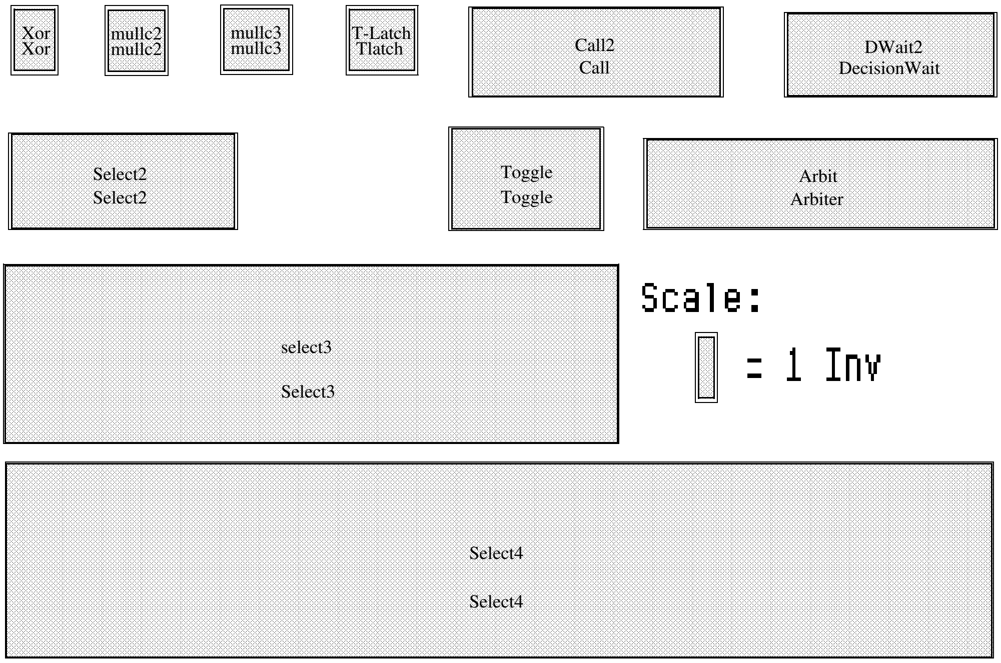
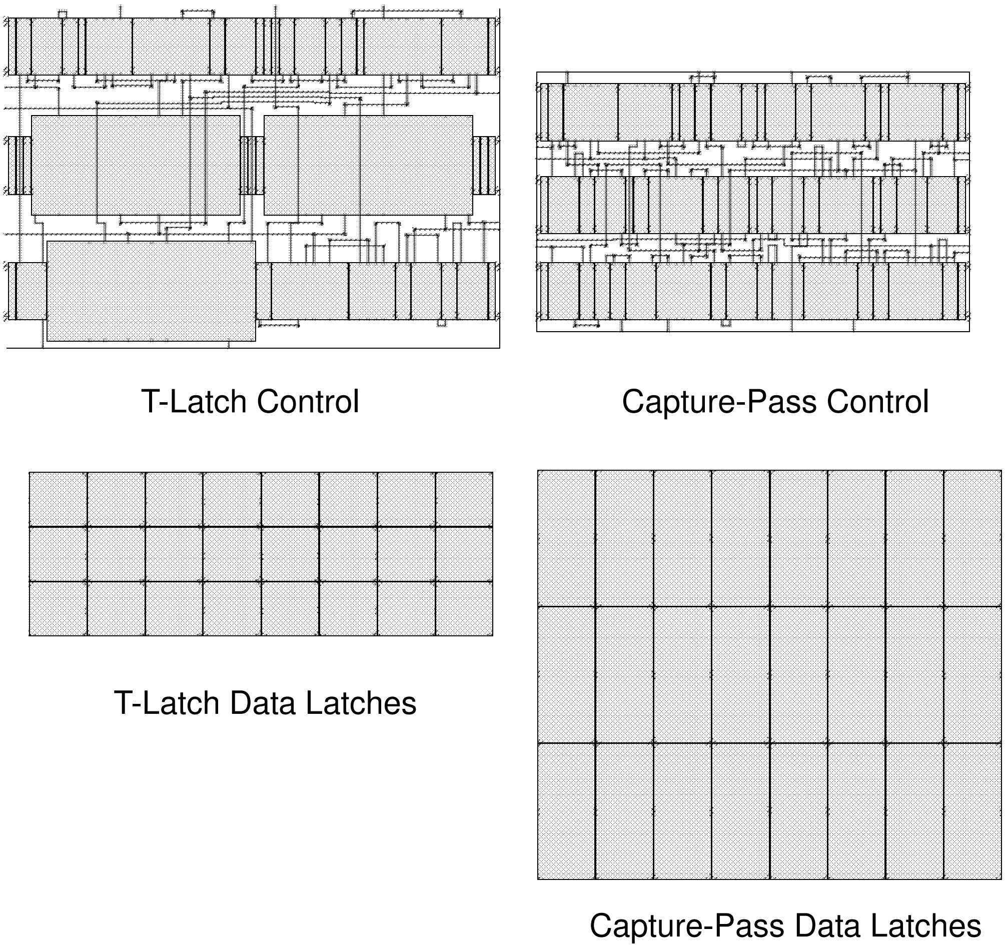


Figure 27: Layout areas of event control blocks and a standard inverter.

Earlier it was stated that capture-pass latches took significantly more silicon area than standard transparent latches, though the control for standard latches is larger. We can now look at this in more detail. The layout areas for 24 data latches and the control circuits are shown in figure 28. From these areas it can be shown that standard latches do, indeed, save on silicon area whenever there are more than 8 data bits in the latch [3]. On AMULET1 latches typically are 32 bits wide, so the area saving from using standard latches is significant.

This concludes the introduction to micropipelines and related engineering issues.



T-Latch Control

Capture-Pass Control

T-Latch Data Latches

Capture-Pass Data Latches

Figure 28: Latch area comparisons

We will now go on to see how micropipelines may be applied to the design of an asynchronous microprocessor. The next section gives an introduction to the ARM microprocessor, then subsequent sections describe the micropipeline implementation of the ARM, the results from the test silicon and the conclusions that can be drawn from the work so far.

5 The ARM Processor

AMULET1 is an asynchronous implementation of the ARM microprocessor [4]. This section gives some background on the ARM and the company which was responsible for developing it, Acorn Computers Limited, which is based in Cambridge, England.

In 1982 Acorn launched the BBC microcomputer, based on the 8-bit 6502 microprocessor, which established Acorn's education market in the UK and certain other countries. (Over 1.5 million BBC machines have been sold worldwide.)

Design of the 32-bit ARM1 was started in 1983. This was the first implementation of the ARM, based on a 3µm CMOS process. The first ARM1 samples worked in April 1985 after 6 man-years of development, establishing the ARM as the world's first commercial development of the pioneering RISC ideas from Berkeley and Stanford Universities. The design was then moved onto a 2µm CMOS process, and the first ARM2 samples were delivered, fully functional, in 1987 and went into production in the Acorn Archimedes personal computer. The Archimedes sold in reasonable numbers, but real volume came with the cost-reduced A3000 which brought low-cost RISC performance into UK education in 1989. In the same year, first samples of the ARM3 were delivered, a 1.5 micron CMOS design incorporating an ARM2 macrocell and 4k bytes of fully associative cache memory on the same chip.

In 1990 Advanced RISC Machines Limited was formed by Acorn, Apple and VLSI Technology as a separate company established to deliver ARMs to a much wider market. In 1992 the ARM610 was developed for the Apple Newton PDA product, and since then the number of ARM licensees and chips incorporating ARM macrocells has increased regularly, establishing the ARM as a world standard architecture for a range of low-cost and low-power applications.

5.1 ARM6 Programmers' Model

The ARM has a load/store architecture with 16 visible registers available to the programmer [4,5]. Register 15 is the program counter; all other registers being general purpose with the only special use being that subroutine return addresses are placed into register 14. Exceptions are handled in protected modes which switch in private registers in place of user registers 13 and 14, the former usually pointing to a private stack in main memory and the latter containing the exception return address. Fast interrupt mode also has some private work registers. The register organization is shown in figure 29.

In addition to the general registers the ARM also has a Current Program Status Register (CPSR) visible in every mode, and a Saved Program Status Register (SPSR) for each non-user mode (figure 30). Each of these registers is used to save processor mode, interrupt enable status and condition code flag bits (figure 31).

The ARM6 instruction set is summarised in figure 32. In common with other RISC processors, ARM separates those instructions which perform data processing functions from those which move data between memory and registers. The most unusual features of the instructions set include:

- All instructions are conditionally executed.
- Load and store multiple register instructions are included which transfer any subset of the currently visible registers.

The former reduces the number of branches which are required, allowing, for example, some if-then-else clauses to be compiled without a branch instruction. The latter improves the efficiency of procedure entry and exit, and increases the rate at which data block moves can be carried out.

This is the architecture that is re-implemented on AMULET1 using a fully asynchronous design style based on micropipelines. The AMULET1 chip has the functionality of the ARM6 macrocell, omitting only the coprocessor instructions and

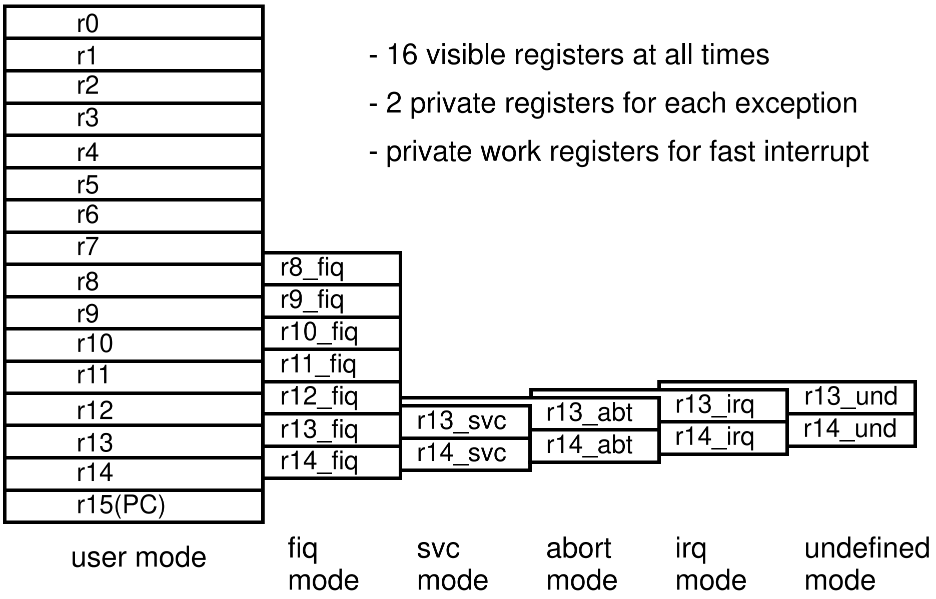


Figure 29: ARM register organization

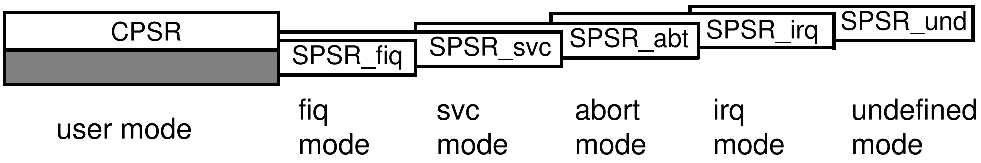


Figure 30: ARM program status registers

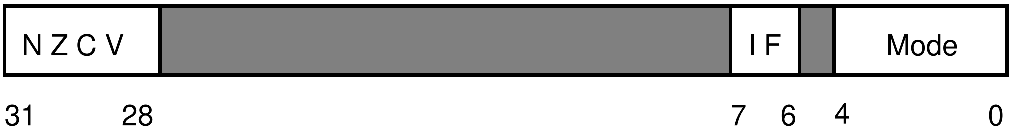
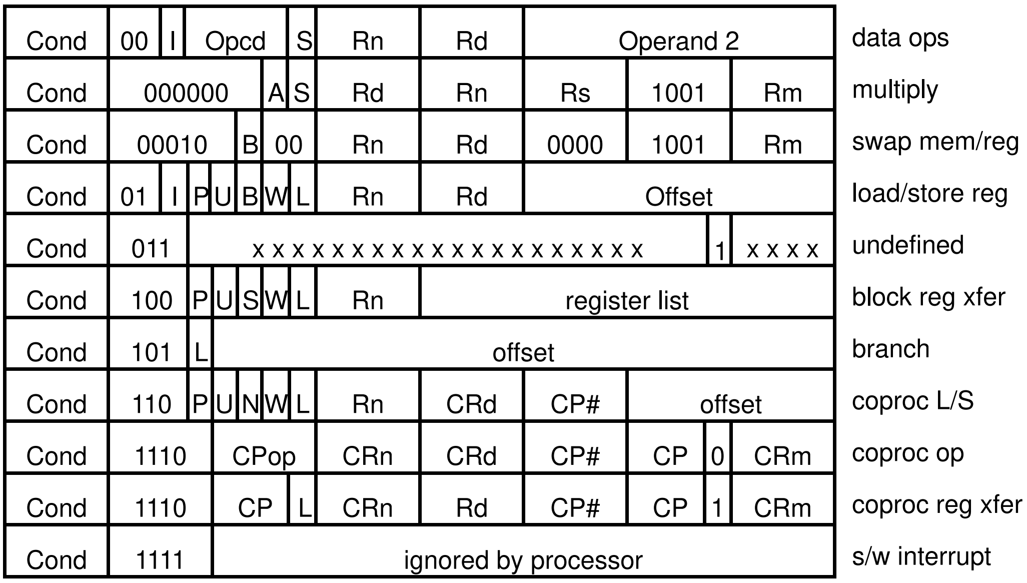


Figure 31: ARM PSR format

support for the 26-bit address space modes which ARM6 has for compatibility with the older ARM chips.

6 A Micropipelined ARM

The design of AMULET1 [3,6,7] begins from a consideration of the environment in which the processor will be used and the interfaces through which the processor should communicate with that environment. For instance, should the chip be



Cond	00	I	Opcd	S	Rn	Rd	Operand 2				data ops
Cond	000000	A	S	Rd	Rn	Rs	1001	Rm			multiply
Cond	00010	B	00	Rn	Rd	0000	1001	Rm			swap mem/reg
Cond	01	I	P	U	B	W	L	Rn	Rd	Offset	load/store reg
Cond	011	x x x x x x x x x x x x x x x x x x x x	1	x x x x							undefined
Cond	100	P	U	S	W	L	Rn	register list			block reg xfer
Cond	101	L	offset								branch
Cond	110	P	U	N	W	L	Rn	CRd	CP#	offset	coproc L/S
Cond	1110	CPop	CRn	CRd	CP#	CP	0	CRm			coproc op
Cond	1110	CP	L	CRn	Rd	CP#	CP	1	CRm		coproc reg xfer
Cond	1111	ignored by processor									s/w interrupt

Figure 32: ARM6 instruction set

designed to use an existing interface so that it can plug into an existing environment?

It was decided to use a micropipeline interface between the chip and its environment. This approach was chosen because the longer term plan was to build the AMULET1 core into a larger chip as a macrocell with on-chip memory (possibly in the form of a cache). Such an on-chip memory could benefit greatly from operating as a micropipeline, so a micropipeline interface is a natural way to connect the memory to the processor core. (In retrospect this decision was probably a mistake, since it made testing the AMULET1 prototypes rather difficult. Two-phase control circuits are easy to implement in VLSI but are much harder to work with at board-level.)

The top level interface is shown in figure 33. The MMU and memory are not part of the AMULET1 design task, but are shown here to illustrate the environment in which the processor will be employed. The processor produces one output bundle containing the memory address, the ‘write’ data (if any) and control bits. The control bits include a read enable, a write enable, an opcode fetch bit (to indicate whether an instruction or a data item is to be fetched), a privilege mode bit and two bits which hint at sequential address behaviour. A second (input) bundle is used to transfer read data back to the processor.

Note that at this stage no assumption is made about the pipeline depth of the memory subsystem. Indeed, if the memory includes a cache, the effective pipeline depth may depend on whether the cache is hit or not. The memory must, however, return results in the same order as the requests were issued.

As the processor handles memory faults as exact exceptions, it must internally prevent any state change after issuing a request for data from memory until it knows that the request will succeed. Therefore a fault/no fault response from the MMU is time critical on data transfers, and the design employs a dual-rail encoded abort response signal from the MMU. A transition on one wire signifies ‘abort’, causing exception entry, whereas a transition on the other wire signifies ‘no abort’, allowing

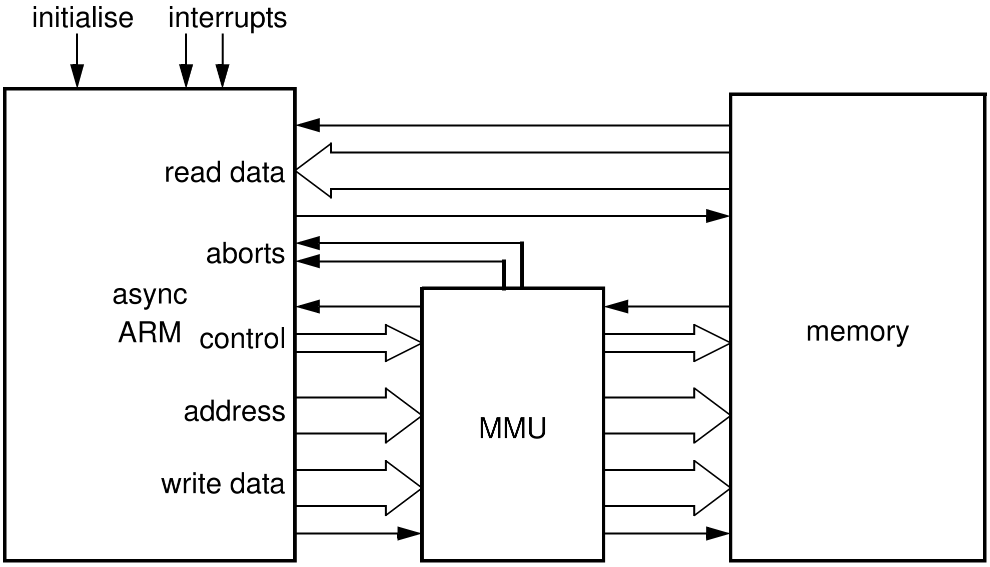


Figure 33: AMULET1 top-level interface

the processor to proceed. (For instruction prefetches a boolean flag returned with the instruction is sufficient to indicate an unsuccessful access; the trap is then initiated when the instruction enters the decode stage.) Conventional level-sensitive interrupt inputs are provided for compatibility with existing peripheral chips, though this is not really a satisfactory model for an asynchronous system as there is no control on the timing of the release of an interrupt line. A system initialisation input completes the top level interface.

6.1 Processor Organisation

The internal organisation of the processor is shown in figure 34. The processor state is held in the register bank, which has two read ports for operand access. The operands are processed by the execution unit and the result written back through a write port. The write port also allows data to be written into the register bank from memory. The third input port is used to allow the PC (which resides in the address interface unit) to be available as register 15 as required by the ARM instruction set.

The address interface unit produces sequential addresses autonomously and only requires input from the execution unit when the address sequence is changed; this may be a temporary change for a data access or a permanent change for a branch.

Write data are copied from one of the register bank read operand buses and then synchronised with the appropriate address for issue to memory. Values returned from memory are split into data and instruction streams and processed accordingly.

6.2 The Address Interface

The address interface includes a PC word-increment loop (see figure 35). The ARM instruction set specifies that (in most cases) R15 has the value PC+8 (exposing the

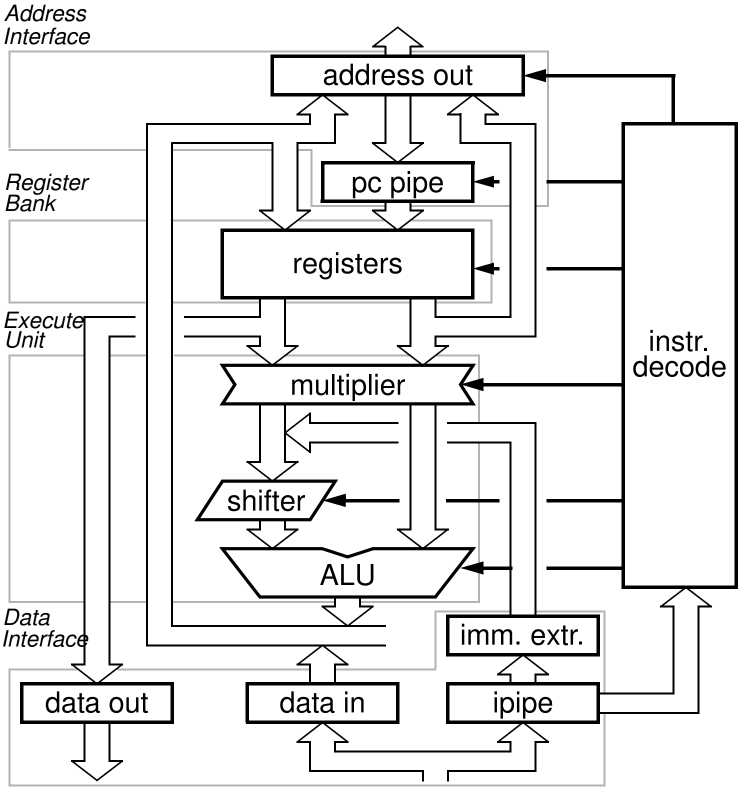


Figure 34: AMULET1 internal organization

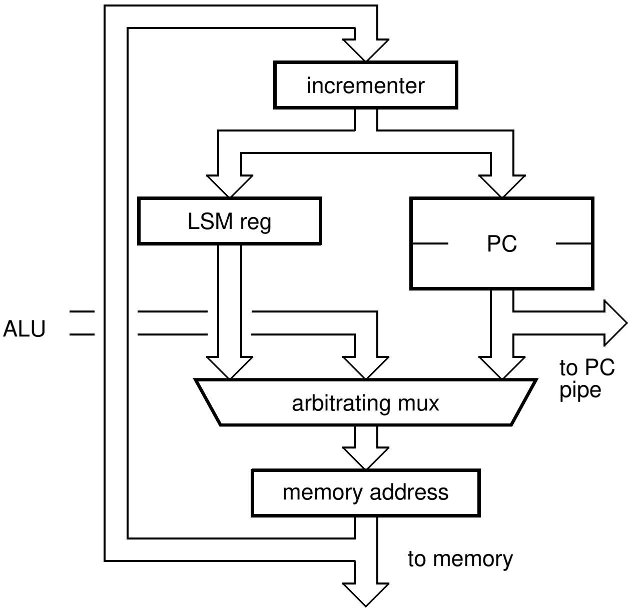


Figure 35: The address interface

depth of the synchronous pipeline implementation in the original ARM), so for code compatibility that behaviour is copied here. The first address, 0, is produced in the memory address register. After passing through the incrementer, the first value offered to the PC pipeline (which buffers PC values until they are needed as R15 during execution of the fetched instruction; see figure 46) is therefore 4. However the first value is 'thrown away', so the first value actually copied into the PC pipeline is 8. This skew between the PC pipeline and the fetched instruction stream persists and maintains the alignment of all future instructions with PC+8. The skew causes incorrect pairing of the instructions immediately following a taken branch, but as these are not executed this is of no consequence. Correct pairing is re-established by the time the new instruction stream begins execution.

The incrementer loop operates autonomously so a new address from the ALU arrives asynchronously. An arbiter is required to ensure safe interruption of the incrementer loop, and the precise point where the loop is interrupted is therefore non-deterministic, resulting in the non-deterministic depth of prefetching beyond a branch. The interruption may be transient (for a data access) or permanent (for a branch). In the former case the PC value is preserved in the PC register; in the latter case the old value is thrown away and the new one takes over the loop.

The ARM instruction set includes multiple register loads and stores of arbitrary subsets of the visible registers. Here the memory addresses are sequential, so the incrementer loop is temporarily usurped to produce these addresses. In this case the PC is held in the PC register, and the looping value passes through the load/store multiple (LSM) register.

6.3 The Register Bank

A high-level view of the operation of the register bank is shown in figure 36. An instruction specifies two source register addresses (a and b) and the destination regis-

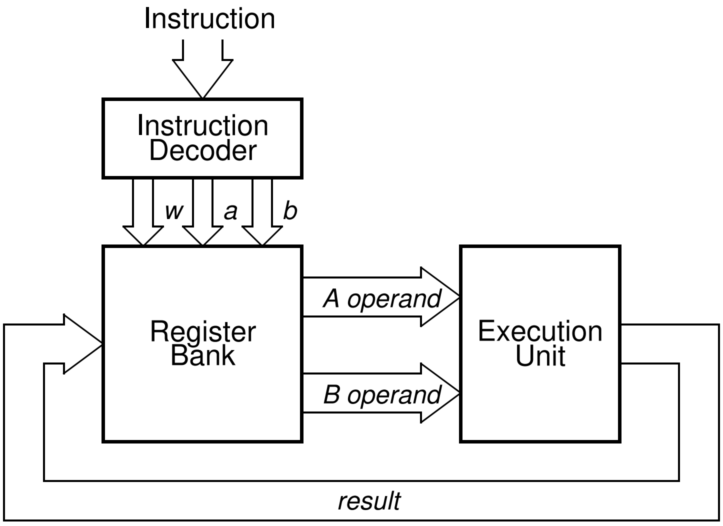


Figure 36: High-level view of register bank operation

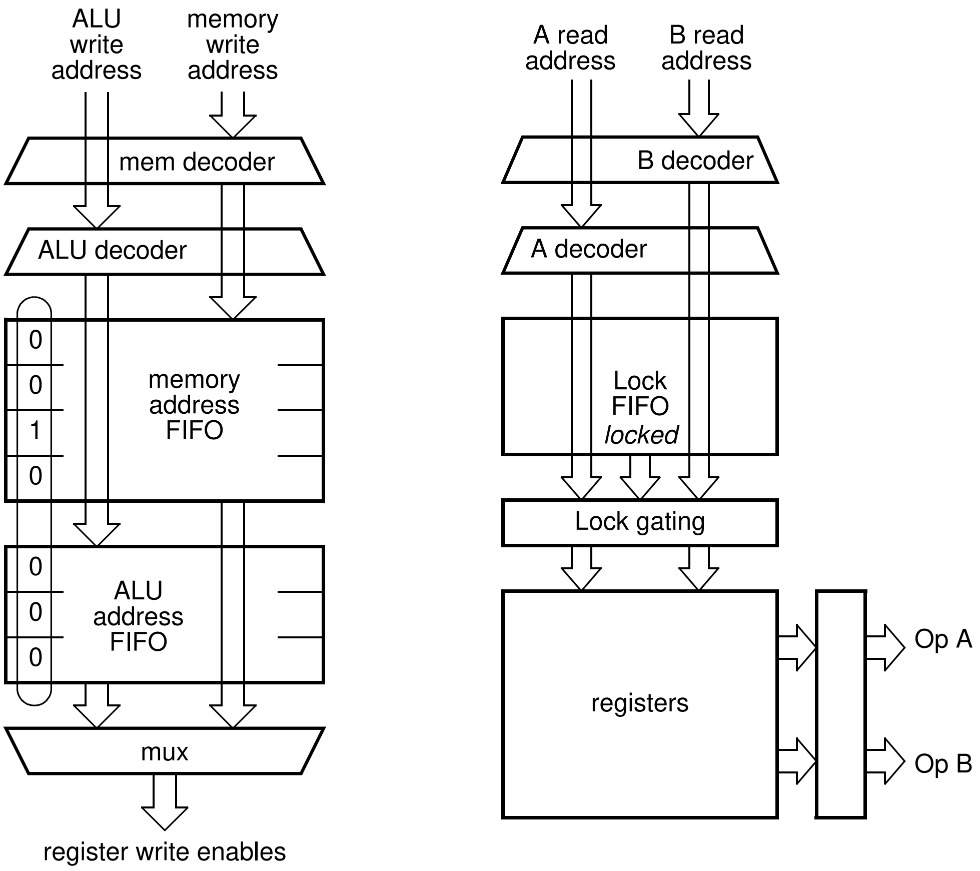


Figure 37: Register lock FIFO and read logic

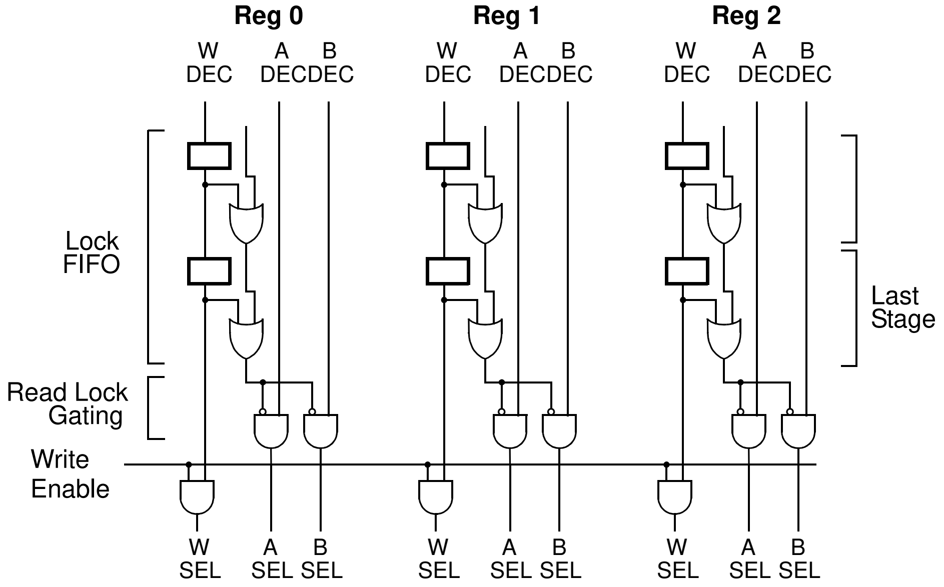


Figure 38: Register read lock gating logic

ter address (w). The register bank then produces the source operands and waits for the result to be returned before placing it in the destination. In pipelined operation, however, the register bank may produce source operands for several instructions before the result from the first one returns. The unit must therefore handle multiple pending write operations register locking to prevent access to stale register values and the asynchronous interaction between read and write operations. All these issues

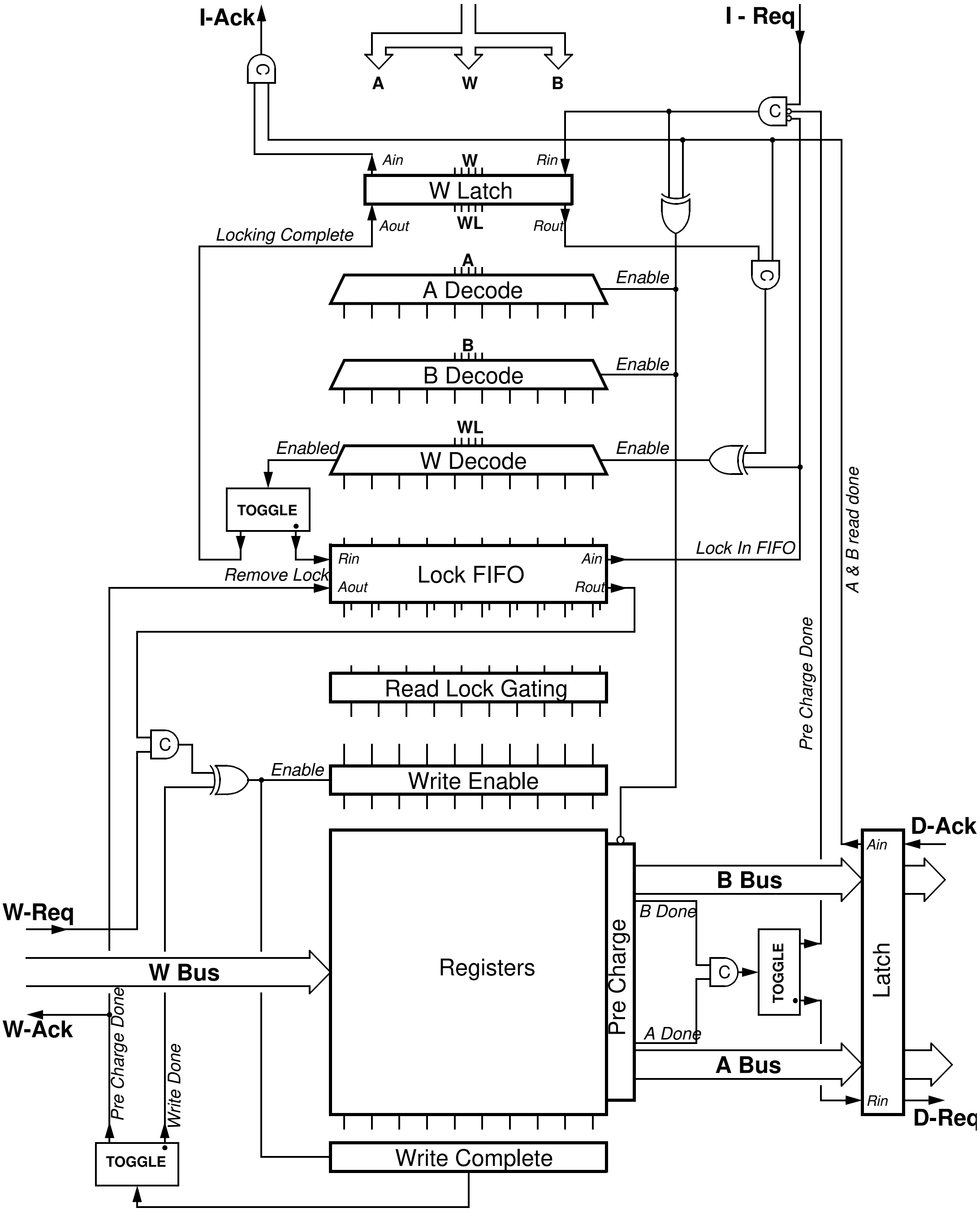


Figure 39: Register bank organization

are resolved in a single regular structure,the register lock FIFO [8], as illustrated in figure 37. (Note that in our design there are two FIFOs to allow internal results to overtake data from - potentially much slower - memory.) Here write addresses are queued in a fully decoded form, so each stage of the FIFO contains at most one ‘1’. A column of the FIFO contains all the information about pending writes to a particular register and a logical OR of the column provides the lock control. The OR function is not normally permissible across such an asynchronous structure but is possible here because, as data propagates through the FIFO, a ‘1’ is copied to the new latch before it is removed from the old one and the lock output is glitch-free.

The lock information is used to delay the decoded read word lines until the correct value is available (details of the lock FIFO logic are shown in figure 38). Multi-

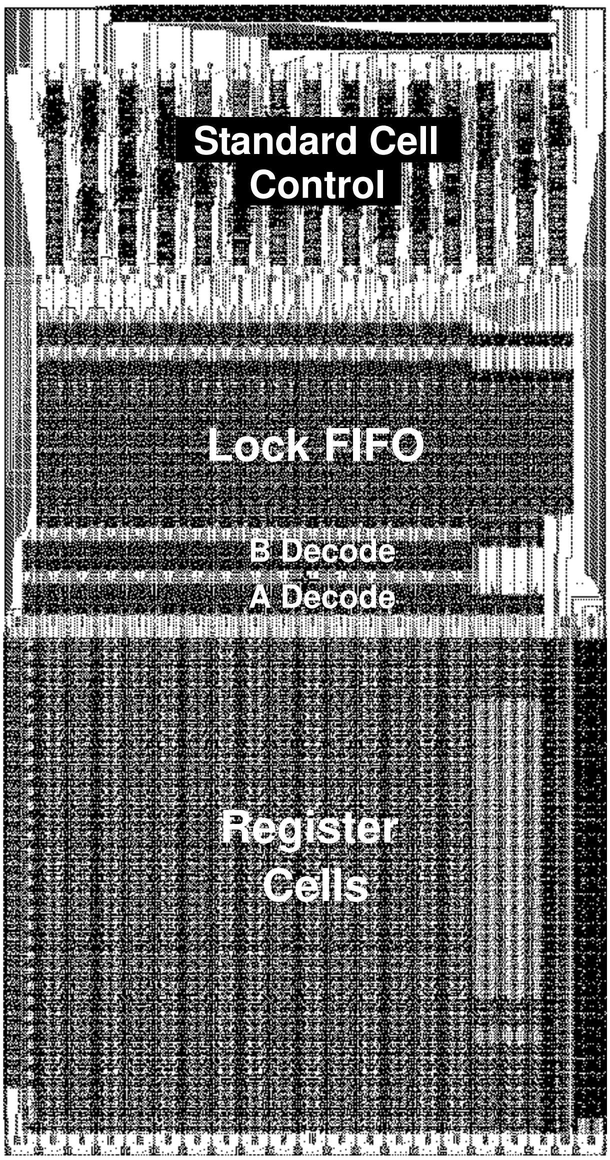


Figure 40: The register bank layout

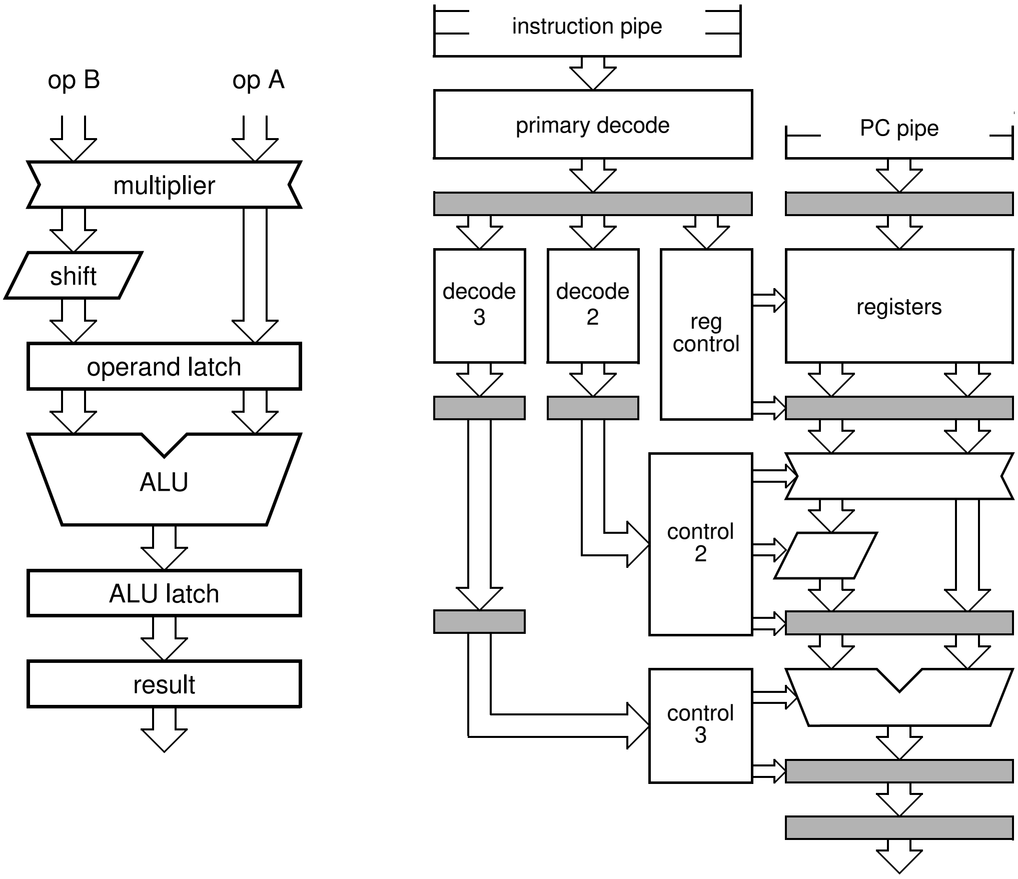


Figure 41: The execution pipeline and control structures

ple locks on a single register are handled correctly and no arbiter is required to manage the asynchronous interaction between reads and writes; these proceed independently when there is no interdependency and the lock mechanism synchronises them when a dependency occurs.

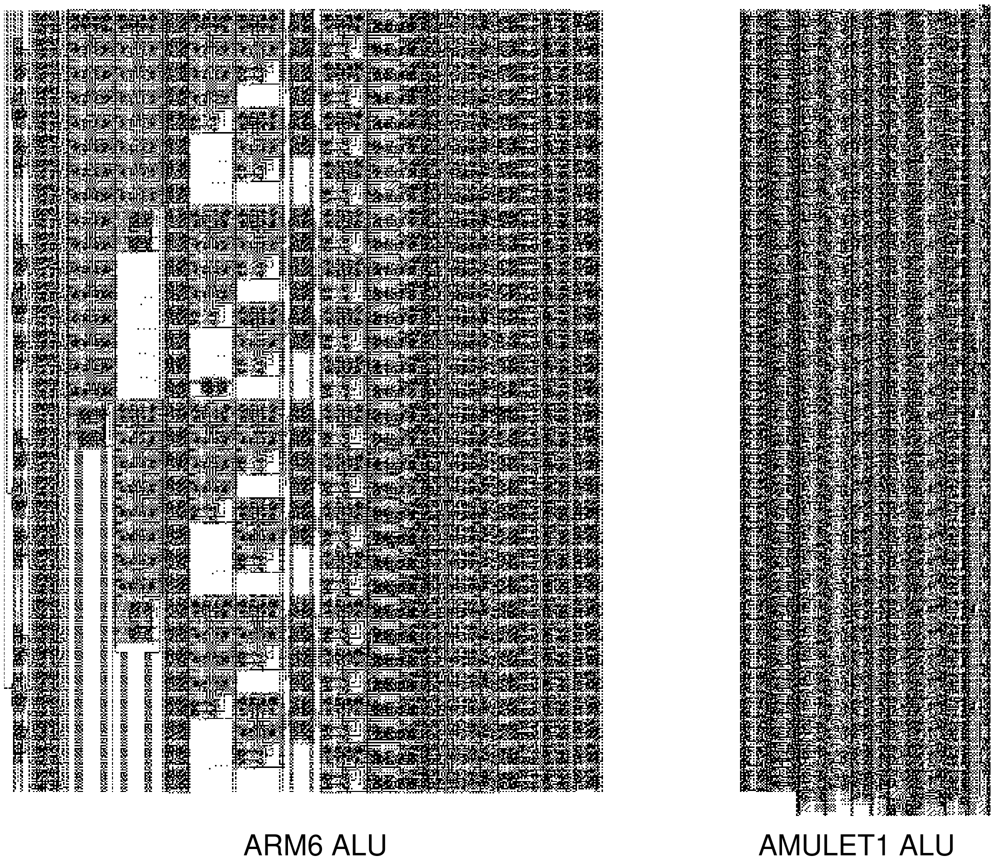
The complete organization of the register bank is shown in figure 39 and the resulting layout is shown in figure 40.

6.4 The Execution Unit

The functional units in the execution pipeline are shown in figure 41. The register operands first pass through a multiplier, which either passes them on immediately or replaces them by partial product and partial carry outputs from a carry-save multiplication unit. A barrel-shifter then modifies one of the operands before both are placed into a pipeline latch. The operands are then combined in the ALU which has a data-dependent delay and a latch to allow a dynamic structure to operate with static external behaviour. A result latch passes the output to its next destination (either a register or the address unit).

The sequential positioning of the function units is perhaps not ideal for performance. However the ARM instruction set supports shift and ALU operations in a single instruction, forcing the barrel shifter to be in series with one of the ALU inputs. Multiplications also use the ALU for the carry propagate addition which is required to combine the partial product and partial carry. When no multiplication is required, the cost of passing the operands through the multiplier is equivalent to passing them through the multiplexer which would be needed to bypass the multiplier, so overall this arrangement would appear to be the best compromise for the target instruction set.

The instruction decode and execute pipe control logic are also illustrated in figure 41. Here the pipeline latches are shaded to highlight the structure. Prefetched instructions are queued before being passed to the primary decode logic which produces multiple pipeline bubbles for the more complex instructions, sends appropriate read and write addresses to the register bank for each bubble and passes information on to the secondary and tertiary decode logic. Note that although the shaded pipeline latches are aligned to emphasise the matching of the pipeline depths of the parallel structures, synchronisation only occurs when the pipelines interact, for instance where 'control 2' connects into the multiplier and where 'control 3' governs the ALU.



ARM6 ALU

AMULET1 ALU

Figure 42: ALU size comparisons

As an example of an area where asynchronous logic appears to offer particular advantages, consider the ALU [9]. Clocked ALUs (such as that on the ARM6) must ensure that the worst case data operands can be processed within the clock cycle time and considerable logic is added to the ALU to make these rare worst cases complete as fast as possible. An asynchronous ALU can allow rare worst cases to take longer and can therefore dedicate the logic resource to making typical cases go fast. A comparison of cell size between the AMULET1 and ARM6 ALUs is shown in figure 42.

6.5 Data Operation Datapath Activity

The operation of the datapath during a simple data processing instruction is illustrated in figure 43. The active buses are shaded for register-register and register-immediate instructions. Note that in these figures the instruction does not occupy all the shaded buses at the same time; the execution is pipelined, and at any one time different resources may be in use for several different instructions.

The only difference between the two illustrations in figure 43 is the source of the second ALU operand, which comes either from the register bank or from the immediate field extractor.

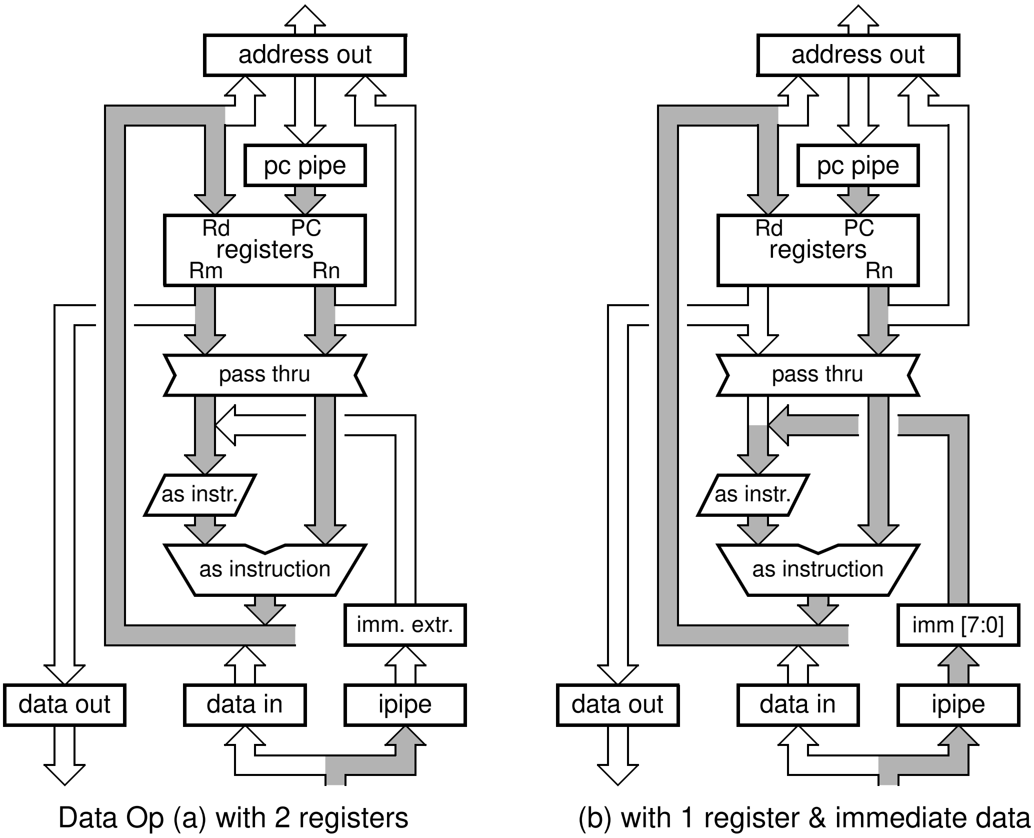


Figure 43: Data operation datapath activity

6.6 Tracking the PC

A high-level view of the mechanism which gives each instruction access to the correct PC+8 value in R15 is shown in figure 44. PC values are autonomously generated in the address interface and issued to memory as addresses for instruction fetches. As they are issued, the PC+8 value associated with each instruction is copied into the PC pipeline. Instructions which return from memory are placed in the instruction FIFO and as they are removed at the end of the FIFO to be decoded each instruction is associated with its PC+8 value. The decoded instruction first passes its register addresses to the register bank and the PC+8 value is passed with these to be used as the value to be read if R15 is accessed as one of the source registers.

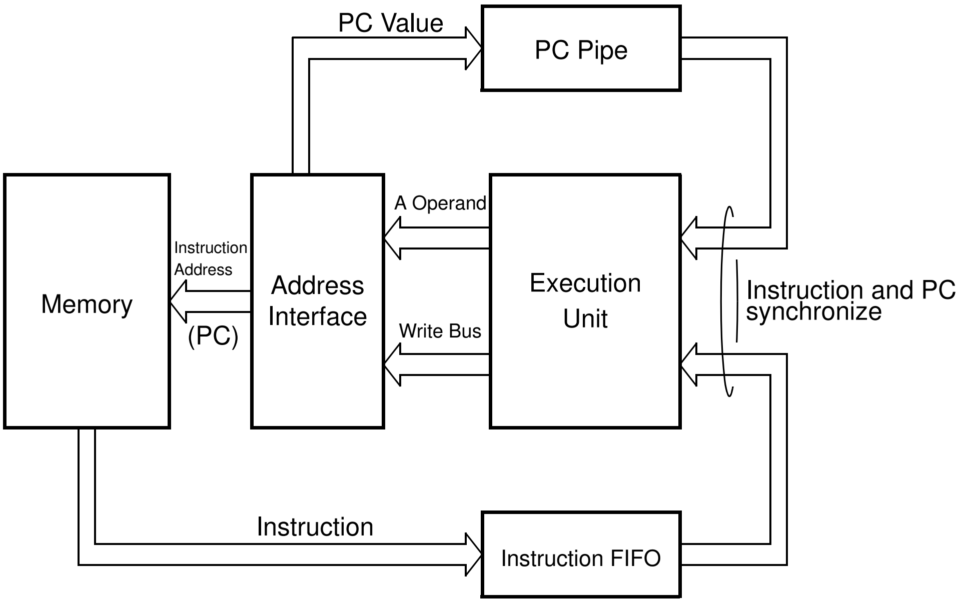
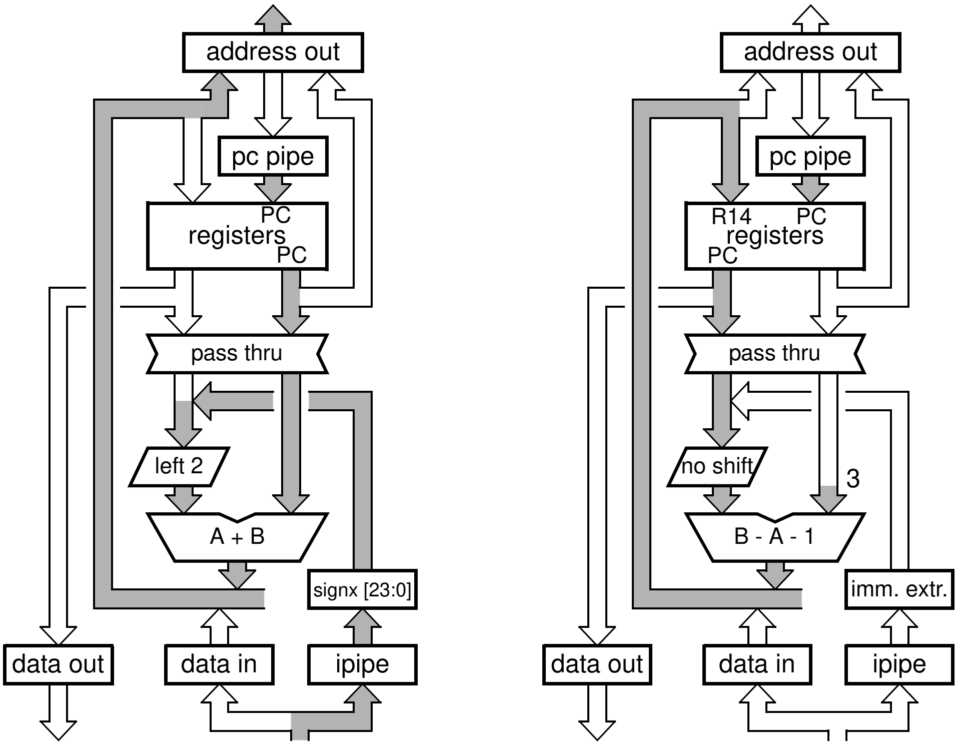


Figure 44: Tracking the PC

6.7 The Branch Mechanism.

The datapath activity for branch and branch-with-link instructions is shown in figure 45. A branch uses a single cycle to add the offset to the current PC value and then to issue that as a new instruction fetch address. Branch-with-link has a second cycle which constructs a return address by subtracting 4 from the R15 value, which contains PC+8, and placing this return address in R14.

After taking a branch the processor must reject instructions prefetched after the branch from the same stream, however the number of rejected instructions is nondeterministic as the branch target is injected into the address interface asynchronously to the operation of the PC incrementing loop. So how can the processor recognise which instructions are to be rejected and which instructions come from the branch target?



(a) B & BL- change PC (b) BL- save return address

Figure 45: Branch datapath activity

The mechanism used is described as instruction stream 'colour'. Every instruction fetch is issued with a particular colour corresponding to the current operating colour of the processor. If the fetch colour of an instruction which comes to be executed does not match the current operating colour the instruction is rejected. Every branch changes the fetch and operating colours, so the operating colour changes immediately and the fetch colour changes from the branch target instruction.

The colour is checked at two positions inside the processor. It is checked at the ALU result stage for precise operation, and at the entry to primary decode for efficient operation. Rejecting an instruction early is efficient, since it saves the time and power that instruction would otherwise consume, but to reject instructions only at the decode stage would require every instruction to be held up there until it was known that the preceding instruction would not change the operating colour. This would compromise performance. Instead, instructions are allowed past the decode stage speculatively if their colour is correct at that point. If a preceding instruction then changes the operating colour after this instruction has entered decode, the instruction will be rejected at the ALU result stage.

6.8 Exact Exceptions

The most difficult aspect of the design of most processors is the provision for exceptions which arise during the execution of an instruction. The processor must allow for the recovery of sufficient information for the exception to be handled and execution of the original program resumed as though nothing had happened. The simplest mechanism which allows this recovery is the exact exception, where the processor stops at the end of the faulting instruction with at most reversible changes from its state at the start of that instruction. Resumption of the faulting program then only requires the reversal of the state changes and a return to re-execute the faulting instruction. More complex mechanisms allow the processor to roll on past this instruction before the fault is discovered and require considerably more ‘history’ to support rewinding and recovery.

AMULET1 uses the exact exception mechanism for load and store accesses to memory to allow an MMU to support a virtual memory system. It must therefore prevent any instruction following a load or store from completing until it is known that the load or store will complete successfully. Instructions are stalled at the ALU stage until an abort/no abort response is returned from the MMU, and if an abort is indicated the operating colour is changed immediately, causing instructions behind the faulting one to be discarded. The abort/no abort information is also passed to the address interface where it controls the bottom of the X pipe (figure 46 shows details of the PC pipelines) using control logic shown in figure 47. This shows how a decision-wait is used either to throw away the bottom entry in the exception pipeline, or to copy that value into the exception latch (X latch). Any value which is copied into the exception latch then causes the exception entry mechanism to be initiated.

The logic which allows the exception latch to break into the instruction stream to cause an exception entry is shown in figure 48. The normal instruction stream enters from the top of this figure. The C-gate waits until the interrupt arbiter, the PC pipe-

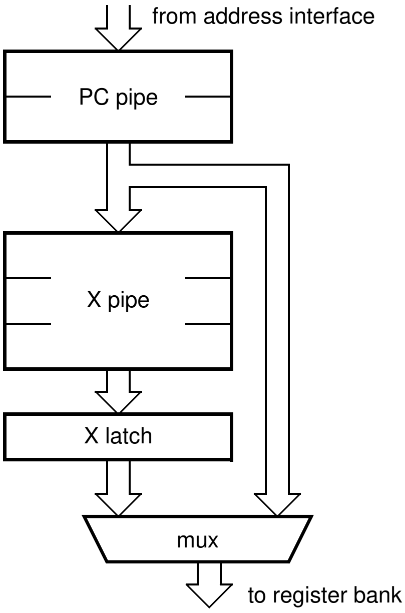


Figure 46: The PC pipelines

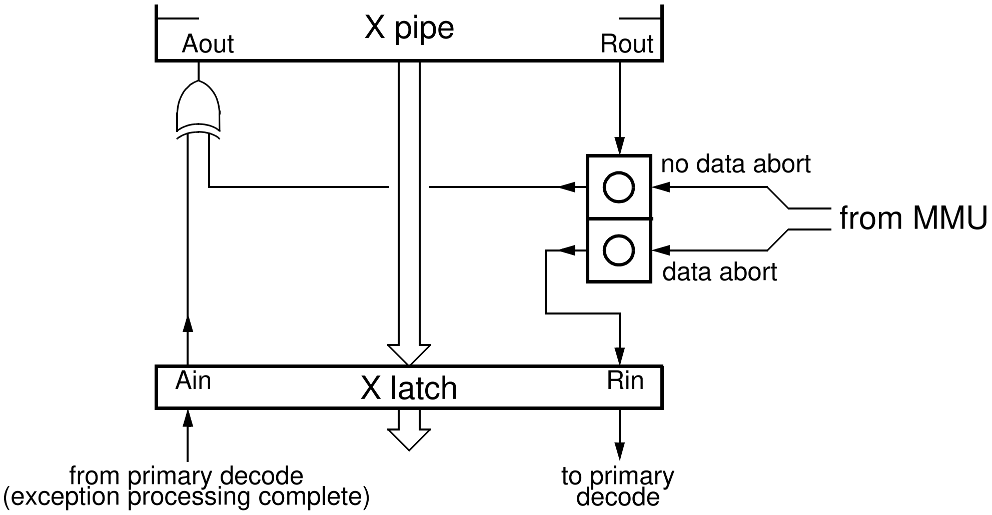


Figure 47: The Exception Latch control logic

line and the instruction pipeline are all ready (ARBrq, PCrq and IPrq respectively) and then the instruction colour (Icol) and the current operating colour (OPcol) are checked to see if they are the same. If not, the select block rejects the instruction by sending the request event straight back to the acknowledge (IPack). When a data abort happens, the operating colour is changed but no further instruction fetch

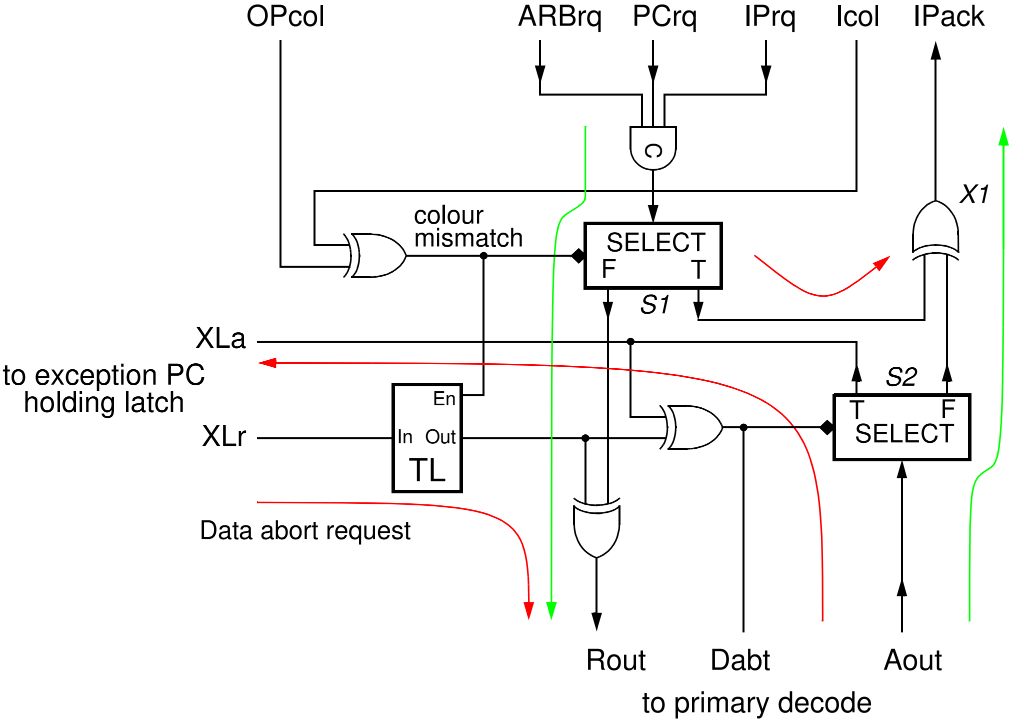


Figure 48: The Data Abort exception entry logic

requests are issued, so eventually a colour mismatch will arise. The data abort will also copy a PC value into the exception latch, causing a request on XLr. This request is held up by the transparent latch until the colour mismatch is detected, whereafter no further requests will go to Rout from S1 so the XLr request can be passed safely to Rout, along with the Dabt boolean, causing the instruction decoder to enter the data abort exception sequence. After the sequence has been issued by the primary decoder, the Acknowledge is passed by S2 back to XLa and clears the Dabt boolean, re-establishing the normal instruction path. In the meantime the exception entry sequence will have caused instruction fetching to resume so the processor will continue normally.

6.9 Chip Composition

The complete pipeline structure of AMULET1 is illustrated in figure 49. The chip was implemented with full-custom datapath components and compiled standard cell control circuits using tools from Compass Design Automation. The floorplan of the chip is shown in figure 50. The control circuits included two PLAs which were generated using a tool supplied by ARM Limited and modified to give access to external circuitry to the completion signal (which already existed inside the ARM design where it controlled power-down and precharging). The VLSI layout of the chip is shown in figure 51.

7 Results

AMULET1 has been fabricated on two CMOS processes: a 1µm process at ES2 and a 0.7µm process at GEC Plessey Semiconductors. Both devices have been evaluated on a test card which connects, via a serial line, to development tools from ARM Limited; the monitor program in the test card ROM is the same as that used in similar evaluation cards for the ARM6. Both prototype devices are functional and execute programs produced by standard ARM development tools such as the assembler and C compiler. There are three minor design flaws which relate to the operation of interrupts and have relatively straightforward software work-arounds. A summary of the devices' characteristics is shown in table 1 with those of ARM6 for comparison.

The devices have been characterised over varying voltage and temperature and display the usual property of asynchronous devices, namely the ability to adapt automatically to changing environmental conditions. The variation of performance and power-efficiency with voltage is shown in figure 52, using the Dhrystone benchmark as an indicator of performance and using the 1µm part. (The 0.7µm part operates at twice the speed but does not have the facility to measure core power consumption.) The voltage range used for these tests is limited by the other circuitry on the test card below 3.5V; the processor appears to operate in isolation down to 2.5V.

Variation of speed with temperature has also been measured. Here the test devices display a normal increase of their delays of 0.3% per °C and operate correctly between -50°C and 120°C.

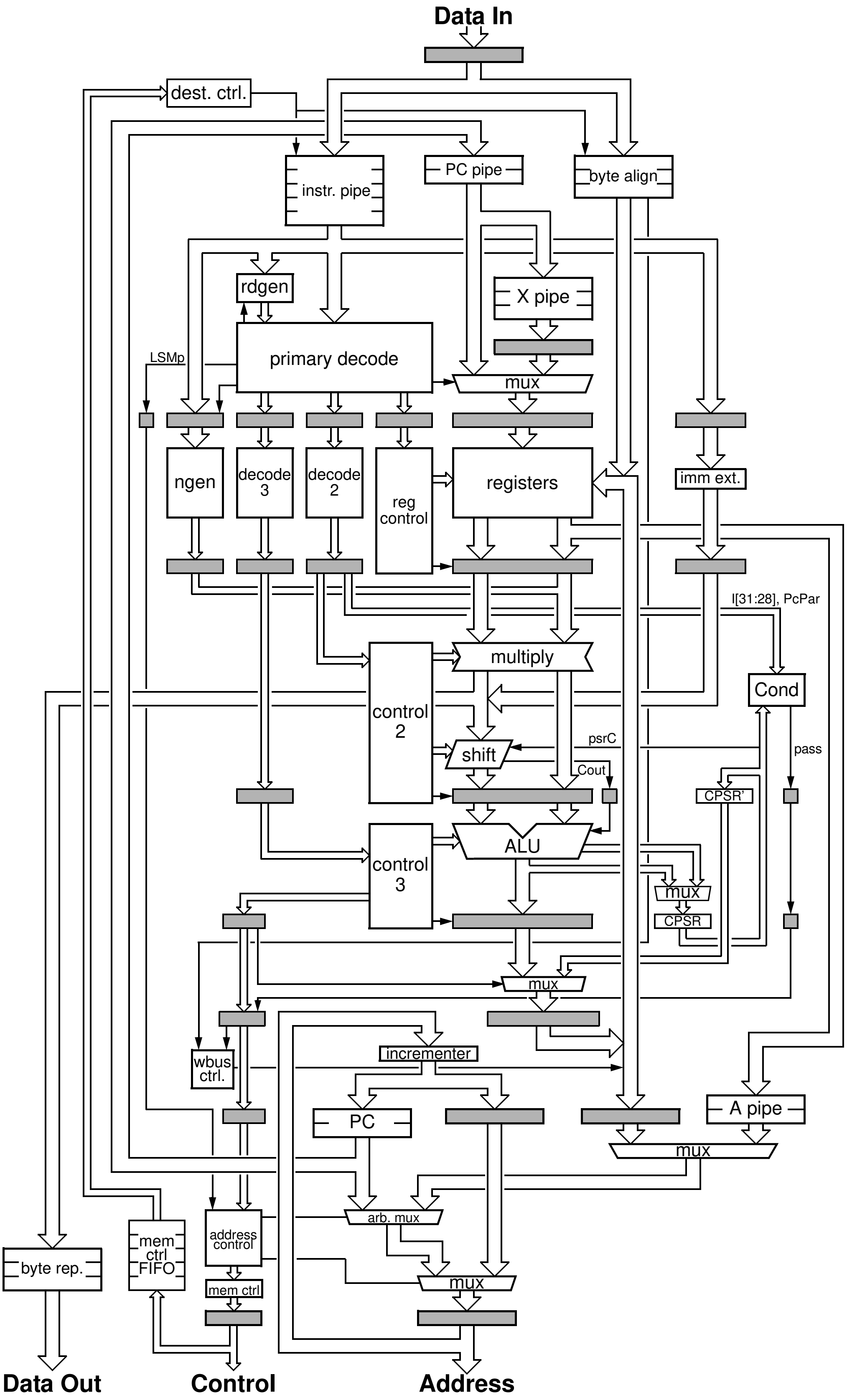


Figure 49: AMULET1 complete pipeline organisation

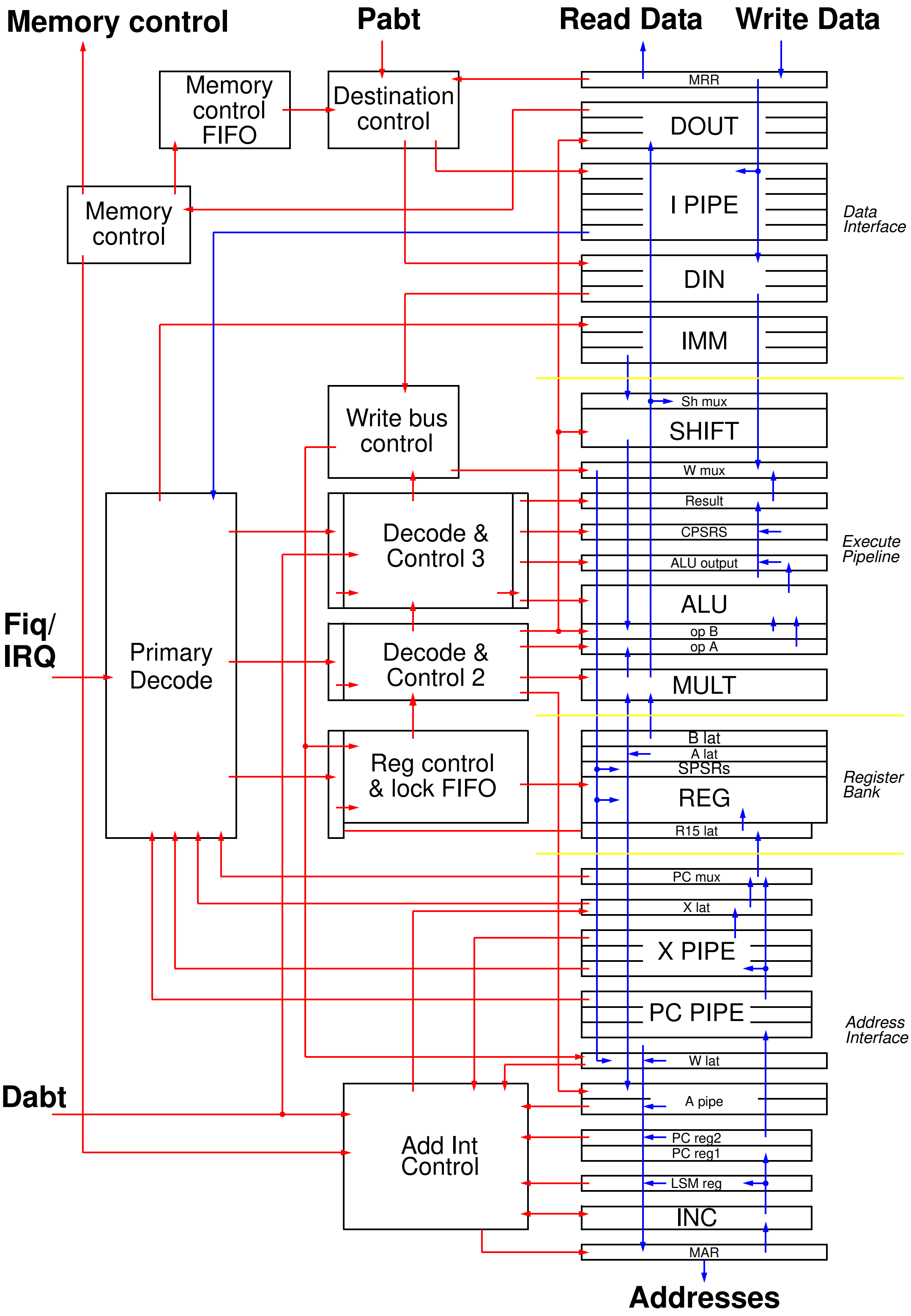


Figure 50: AMULET1 chip floorplan

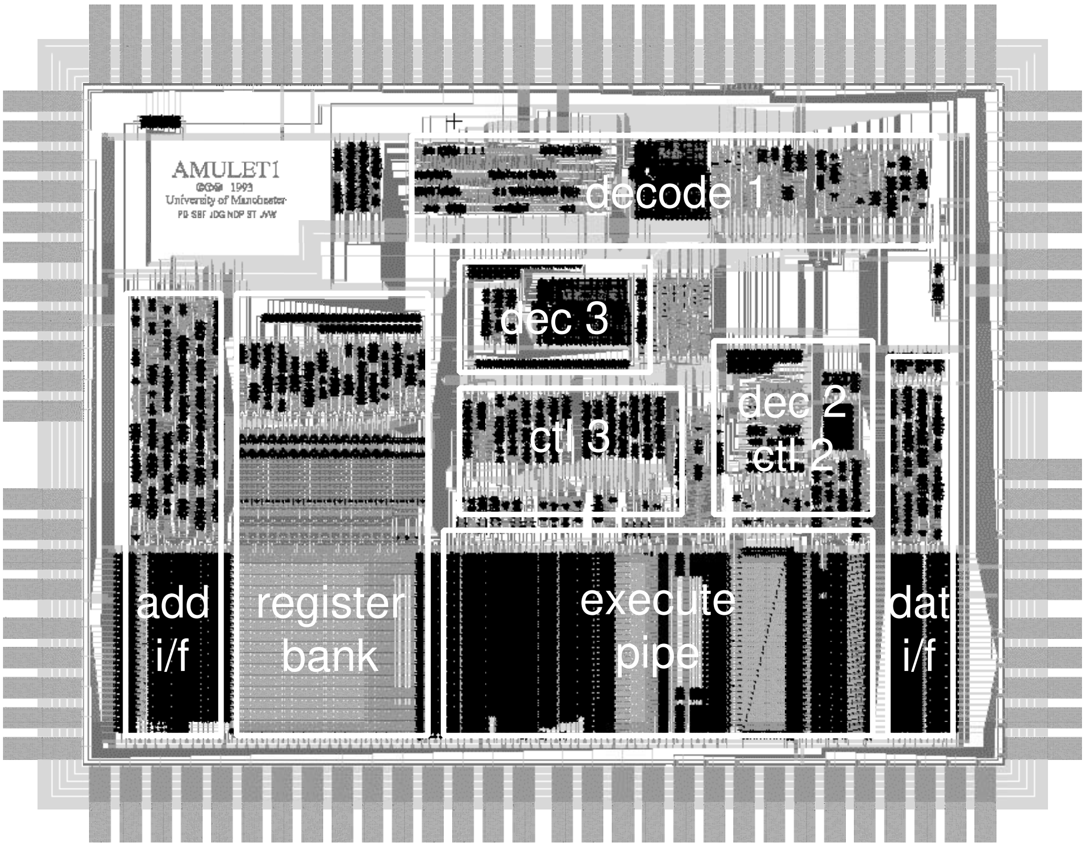


Figure 51: AMULET1 VLSI layout

	AMULET1/ES2	AMULET1/GPS	ARM6
Process	1µm	0.7µm	1µm
Area (mm^2)	5.5 x 4.1	3.9 x 2.9	4.1 x 2.7
Transistors	58,374	58,374	33,494
Performance	20.5 kDhry.	~40 kDhry.[1]	31 kDhry.
Multiplier	5.3ns/bit	3ns/bit	25ns/bit
Conditions	5V, 20°C	5V, 20°C	5V, 20MHz
Power	152mW	N/A[2]	148mW
MIPS/W	77	N/A	120

Table 1: Characteristics of AMULET1 and ARM6

[1] estimated maximum performance.

[2] the GPS part does not have separate core supplies for power measurement

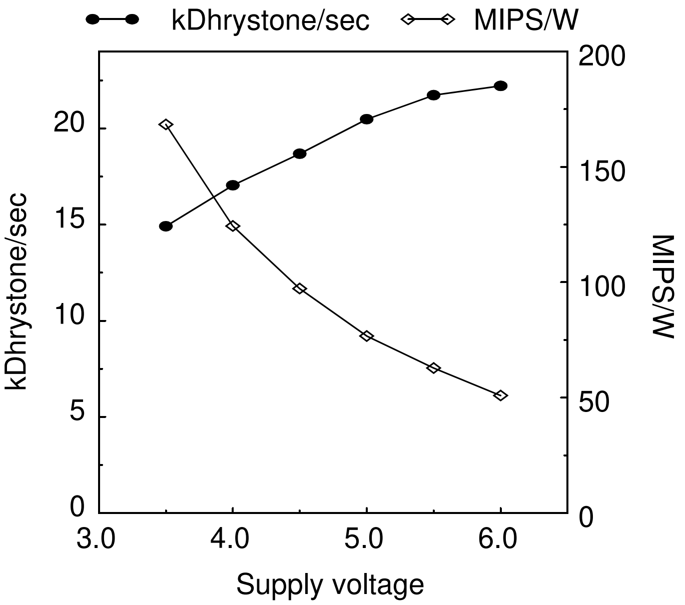


Figure 52: 1µm AMULET1 performance and power-efficiency against voltage

8 Design Tools

The AMULET1 design was developed using a conventional design flow based on proprietary tools from ARM Limited and commercial VLSI CAD tools from Compass Design Automation (see figure 53). In line with standard practice, simulations were run on transistor-level netlists extracted from the physical layout in all the process corners to cover all four combinations of fast and slow n- and p-transistors. Any serious difference between matched paths in their dependence on rising and falling delays is exposed by these skewed simulations, and on AMULET1 a few paths were adjusted it compensate for asymmetries of this sort.

Although these tools allowed the design to be developed successfully and working silicon to be produced, some aspects of asynchronous designs make conventional simulation inadequate for verification. Asynchronous circuits are prone to deadlock and simulation only establishes that deadlock does not arise under the simulated conditions; it cannot prove that liveness is guaranteed under all conditions.

In order to increase confidence in the liveness properties of the design, random delays were added into all the C-gate models in some simulation runs to simulate different time orderings of events. This only serves to increase confidence and still cannot approach an exhaustive test of all possible event orderings.

It should be noted in passing that the tendency for incorrect circuits to deadlock was a significant help during debugging! The behavioural simulator could be set up to maintain a circular buffer of past events. Then, when the model deadlocked, this buffer would contain enough history to identify the source of the problem. When similar problems arise with a clocked design all that may emerge is the wrong answer at the end of a long run, and it can be much harder to identify the time at which the

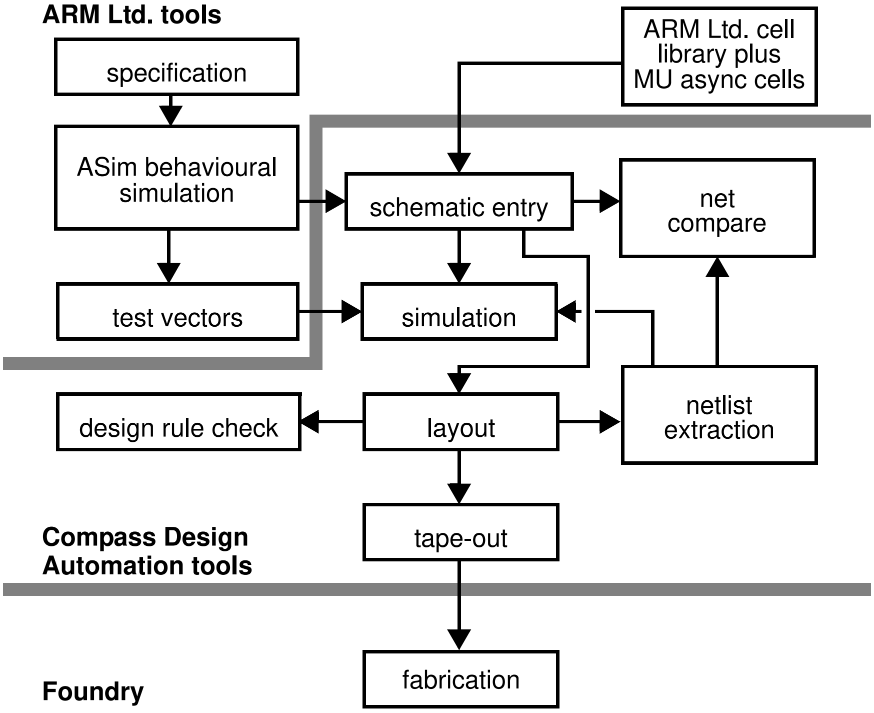


Figure 53: The design flow used to develop AMULET1

error arose and then the source of the error.

In addition to the standard tools, a tool was built to check that bundled data values never changed within the Request-Acknowledge period (with added margins) to ensure that the bundling constraints were never violated. Again, only the timing under the simulated conditions could be checked, not all possible timings under all possible conditions.

Though the above procedures increase confidence, they are not sufficient for general asynchronous design work; the present design is very conservative, and as margins are reduced better tools will be needed to achieve satisfactory confidence levels that the design will operate correctly.

8.1 Design Flaws

Though the first AMULET1 silicon is functional, there are some errors in the design which it may be instructive to review.

- an MSR or MRS executed in a non-user mode immediately after a STM accesses the wrong SPSR, due to an error in the design of logic which allows supervisor code to store the user's registers.
- if an interrupt is disabled and the flag is re-written (again to disable the interrupt) the interrupt may be transiently enabled and entered.

- an exception which arises during the execution of a load or store which fails its condition codes (and is therefore not, in fact, executed) saves the wrong SPSR value if the memory system is sufficiently slow.
- an LDM which restores the CPSR, but is aborted during data transfer, does not leave the CPSR unaffected.

In the absence of aborts, and with sufficiently fast memory, only the first two design errors manifest themselves, and then only in non-user code. Therefore the chip will run general compiled or hand-written user code. The first two errors were identified and diagnosed soon after the first samples were tested. The third error is due to a latch being wired incorrectly, and took a considerable time to diagnose due to its dependency on the memory speed. This caused the error to manifest itself in the part from one source, but not from the other, in the same test card, since the different silicon speeds made the memory in the test card look slower than the critical speed to the former and not to the latter! The fourth error was not found whilst testing the silicon but was discovered whilst considering the redesign for AMULET2; the silicon was checked to confirm that the error existed in AMULET1.

None of the errors are due to problems with self-timing or the asynchronous control structure. As is frequently the case with engineering design, the errors arose not in the areas the designers focussed on but in the peripheral detail. More thorough testing of the design under simulation would have revealed these errors before fabrication.

9 Future Enhancements

The AMULET1 design was shipped for fabrication at the end of February, 1993. Since then work has been underway to enhance the design with respect to both its performance and its power efficiency. Significant improvements have been made in many areas of the design. These improvements cover all aspects of the design including the transistor structures used for data latches, the approach to event control, architectural features to reduce pipeline stalls, through to the compiler.

9.1 Compiler Optimization

As some compiler improvements also enhance the performance of AMULET1, these will be described first.

The compiler used to evaluate AMULET1 is the standard ARM Limited C compiler. As the ARM6 displays no sensitivity to code order, the compiler makes no attempt to separate dependent instructions. AMULET1, on the other hand, shows a strong dependency of performance on code order. The register bank lock FIFO ensures that instructions wait until their operands are available, but whenever an instruction is forced to wait, the pipeline is stalled and performance is lost.

For example, this is the code for a standard string comparison which loads and sign extends the bytes for each string:

```
strcmp
       LDRB   a3,[a1],#1
       MOV    a3,a3,LSL #24
       MOV    a3,a3,ASR #24
       LDRB   a4,[a2],#1
       MOV    a4,a4,LSL #24
       MOV    a4,a4,ASR #24
       ...
```

This code causes register locks between the first and second instruction while the data load accesses memory and a shorter stall between the second and third instructions. The same delays are incurred again in the second group of three instructions. Effectively there are two independent threads in these six instructions which can be interleaved almost to double the execution speed. This interleaving fits the second three instructions in the gaps left by stalls during the execution of the first three.

```
strcmp
       LDRB   a3,[a1],#1
       LDRB   a4,[a2],#1
       MOV    a3,a3,LSL #24
       MOV    a4,a4,LSL #24
       MOV    a3,a3,ASR #24
       MOV    a4,a4,ASR #24
       ...
```

There are other optimizations which improve the speed of the code running on AMULET1 (generally without impacting the speed on ARM6), such as replacing a single register load multiple (which passes four cycles down the execution pipeline) with a single register load (which passes one) and being careful about which instructions are left in a branch shadow (where some instructions may take several cycles in the execution pipeline before being annulled by the colour checking process). Producing optimised code for AMULET1 is harder than for a clocked processor because instruction dependencies have cost functions which are not constrained to discrete multiples of a clock cycle and individual instruction costs can be data-dependent.

Note that the increased inter-instruction dependencies (compared with ARM6) introduce other pressures on the compiler. Interleaving independent threads may increase register allocation pressure, so register allocation and code ordering interact, and ARM does not have very many registers compared with most modern RISC architectures.

Compiler optimization for asynchronous architectures allows for some simple improvements as outlined above, but it is not clear how far optimization can go. The lack of a discrete cost function makes this a new research area!

9.2 Improved Latch Mechanisms

The data latch used on AMULET1 is the same as that used on ARM6. Investigations into alternative latches suggest that there may be potential improvements to be gained in both performance and power-efficiency from switching to latches which do not require complementary latch enable signals. An example of such a latch is the Svensson latch as used on the DEC Alpha processor [10]. The Alpha uses a dynamic form of the latch; the transistor schematic of a variant of the latch with weak feedback to ensure fully static operation is shown in figure 54. The advantage of the single enable signal may be seen by comparing the latch control circuit for a conventional latch with complementary enable signals (figure 55) with that for a Svensson latch (figure 56).

The two forms of latch are compared in table 2. A Svensson latch with carefully sized transistors can give twice the speed of operation at one third of the energy per cycle of the ARM6 latch currently used on AMULET1. It achieves this by removing the need for the inverted enable signal (saving a C-gate and reducing the buffer loads) and by minimising the load each bit places on the remaining enable line.

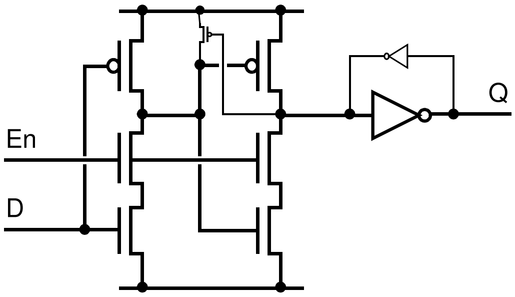


Figure 54: A fully static version of the Svensson latch

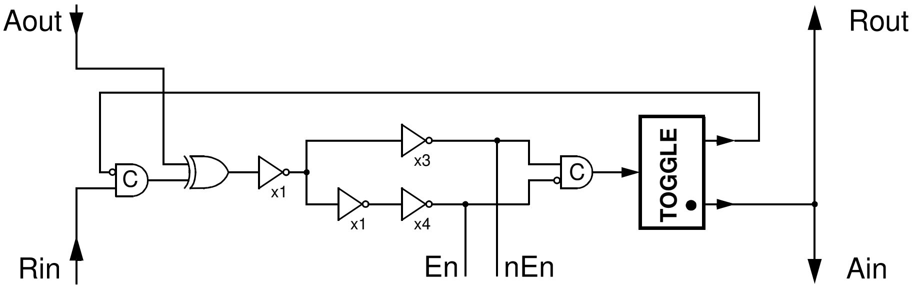


Figure 55: Conventional transparent latch control circuit

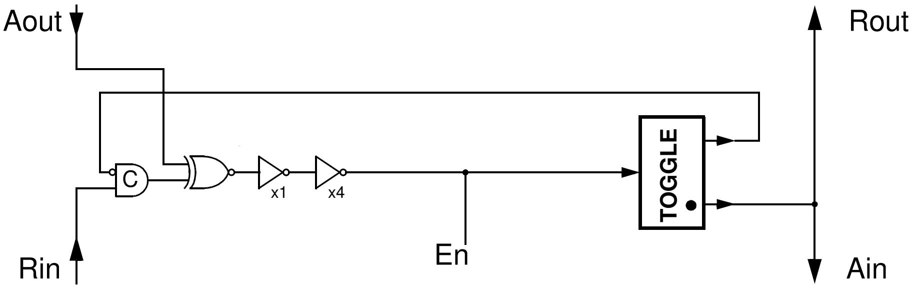


Figure 56: Single phase latch control circuit

	Svensson	AMULET1
Transistors per latch bit	11	6
Inversions in datapath	3	1
Clock load	2N	1N+1P
clock capacitance	1	3
Invs Rin -> Rout	8	11
FIFO cycle time	12ns	25ns

Table 2: Comparison of Svensson and AMULET1 latches

9.3 Four-Phase Control

All the latches described here (apart from the Sutherland capture-pass latch) use 2- to 4-phase conversion control circuits to interface the 2-phase (transition event) control environment to the level-sensitive latches. Clearly the control environment itself could be converted to 4-phase (level-sensitive) operation. Although 4-phase circuitry may appear conceptually simpler at first sight, it is in practice considerably harder to design efficient 4-phase control circuits because the return-to-zero events have no specific meaning so the communication protocol does not define when they should happen. If the return-to-zero event follows the same path around the control circuitry as the active event, the circuit tends to be very slow. Therefore the designer has to expend considerable effort in deciding the best way to handle the recovery phase within each part of the circuit.

Despite these difficulties, the performance advantages of 4-phase control seem to be significant. A 4-phase control circuit suitable for use with Svensson style latches is shown in figure 57. (Note that the C-gates do not require symmetric n- and p-transistor stacks in this circuit.)

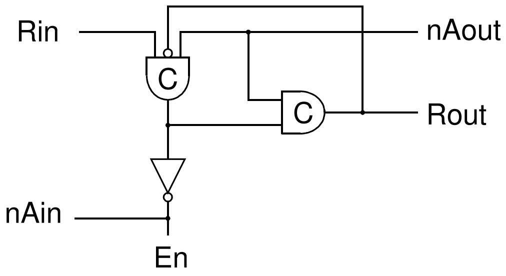


Figure 57: 4-phase pipeline latch control circuit

9.4 Engineering Margins

The self-timed circuits used on AMULET1 raise the question of what margin should be allowed in a matched signal path. If the engineering margin is too small incorrect operation could result. If the margins are too large, performance will be compromised. In order to ensure the best matching between the data route and the matching event path, the paths should be identical, close together on the chip and have the same physical orientation.

On AMULET1 the timing margins were around 20-30% on identical layout (e.g. the 33rd register bit) and 100% on standard cells paths with the same nominal delay. Since no failures on the sample chips are attributable to mismatched paths, these margins appear acceptable, at least for prototype devices. Self-timed paths on clocked chips often use much smaller margins (e.g. below 10%), but here there are usually far fewer paths on a chip and a correspondingly lower probability of failure due to one of them being out of tolerance.

The factors which need to be assessed are process, voltage and temperature variations, volume production yields and the number of potential failure points in the design.

9.5 Restructuring the Pipeline

The AMULET1 pipeline is now considered to be deeper than is optimal. To reduce the latency we propose to bypass the shifter in most cycles and to sideline the multiplier except when it is needed. The revised pipeline is illustrated in figure 58 along with the existing pipeline for comparison.

To be effective the new pipeline organization requires the shifter to be used less frequently than on AMULET1. This may be achieved by extending the immediate operand extraction logic to align all immediate operands and by causing the control logic to detect when the shifter is required for register operands, bypassing it in other cases.

The benefits of this change are lower execution unit latency and reduced power (the shifter will be activated less often). The costs are more datapath logic (a partial shifter in the immediate extraction block) and more complex control logic.

There are other places in AMULET1 where the pipeline can be slimmed without loss of performance. The memory control pipe can be reduced from 5 stages to 2 or 3. The PC and instruction pipes can be reduced by 1 stage; this reduces prefetching (saving power) and improves performance. The exception pipe is longer than is useful, and reducing the A pipe by 1 stage probably will not affect performance.

Overall there are 10 stages in various pipelines which are contributing little to the performance of the chip and are consuming power.

9.6 The Last Result Register

Register bypassing depends on different stages of the pipeline being in step and is not directly applicable in an asynchronous pipeline. However, typical programs display

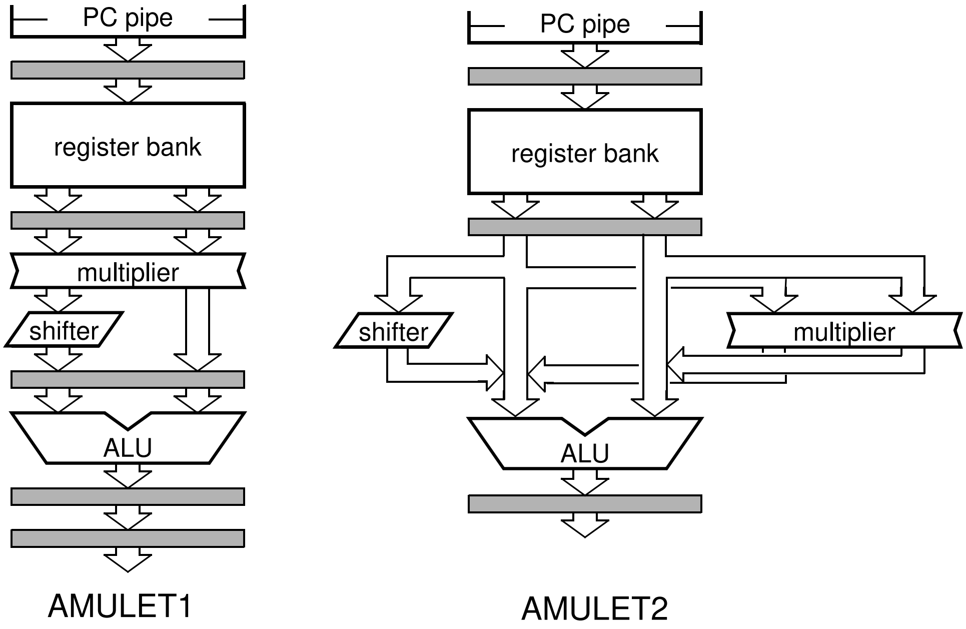


Figure 58: The proposed simplification of the AMULET2 execution pipe

frequent dependencies between consecutive instructions. Therefore an efficient way to feed results to the next instruction is needed. The 'last result register' is a proposed mechanism which is similar to register bypassing but is usable in an asynchronous pipeline.

The last result register operates as follows:

- when an instruction is decoded, the destination register is recorded in the instruction decoder

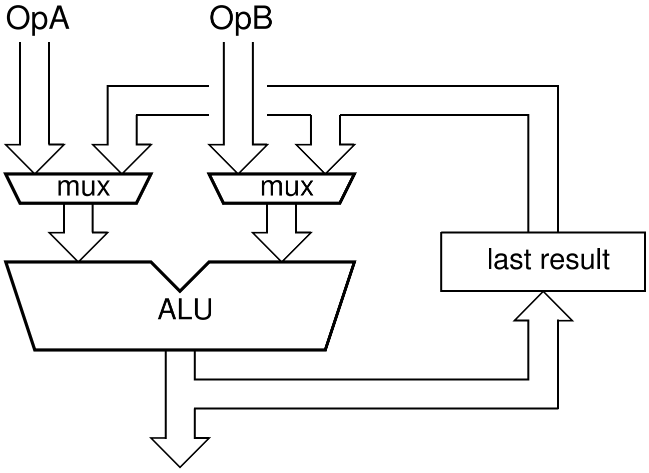


Figure 59: The last result mechanism

- when the next instruction enters decode, it compares its source registers with the previous destination register
- if a match is found, that instruction bypasses the read of the matching register, thereby avoiding stalling on the register lock
- instead the operand is collected from the last result register (figure 59) in the execution pipeline

Note that the last result mechanism is rather more restricted in its use than conventional register bypassing. Only the results of unconditional instructions (i.e. instructions with the condition ALWAYS) can be used. The inclusion of this feature will also move the optimization point for compiled code, and the last result register will make the use of dual register read ports relatively infrequent; is their inclusion still justified?

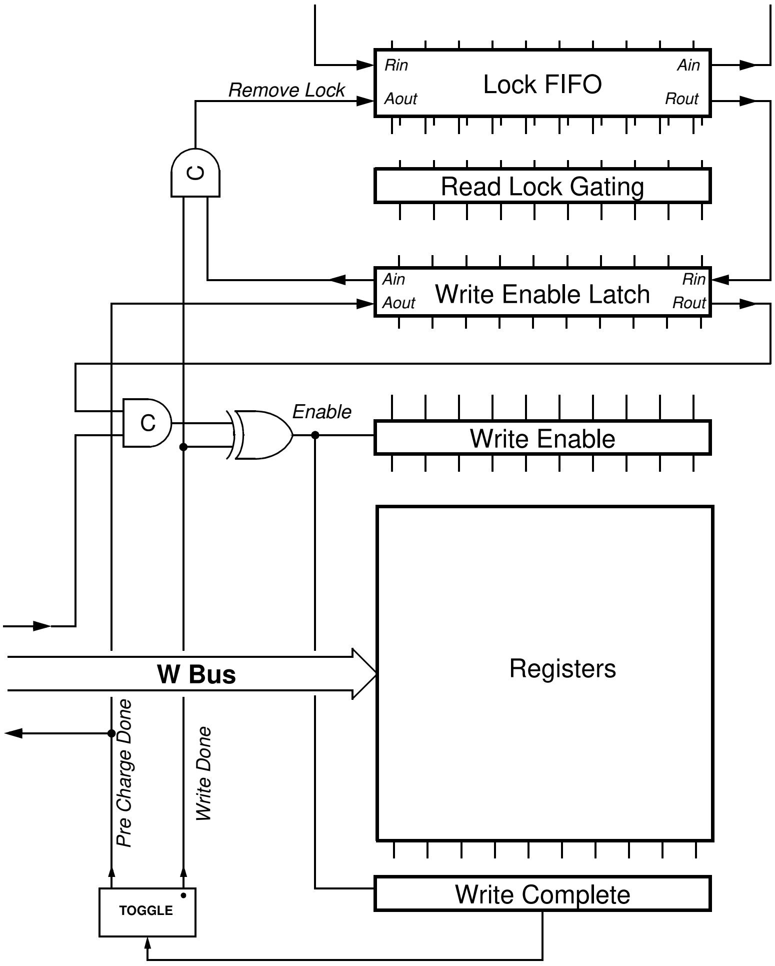


Figure 60: Improved register write logic

9.7 Improved Register Write Logic

The current register bank design releases a read which is stalled awaiting a pending write by clearing the bottom entry of the lock FIFO. As the lock FIFO also controls the register write word lines, its output must remain stable until the write has completed and the write word line disabled. The register will have been written well before this sequence terminates, so the read will have been delayed considerably longer than is necessary.

A modification to the register bank control logic (figure 60) adds a separate latch to hold the register write word line stable, allowing the bottom of the lock FIFO to be cleared as soon as the data has been written to the register. This will release the read operation significantly earlier, improving the performance on code which causes read stalls (which includes most existing ARM code).

10 Conclusions

The AMULET1 design demonstrates the feasibility of designing complex asynchronous circuits, and whilst it does not offer a direct advantage over clocked designs at this stage, there is a lot of room for improvement over the present design.

The architectural features which make synchronous processors go fast do not all transfer easily to asynchronous designs (e.g. register forwarding) and there is a need for new architectural features to be developed for asynchronous designs to replace them (e.g. the register locking FIFO and the last result register).

Micropipelines offer a good framework for the design of high-performance asynchronous circuits. They are amenable to use with conventional CAD tools, though further tool development could assist the design process.

Asynchronous techniques are enjoying a resurgence of interest amongst the VLSI design community because they offer the potential for high performance and low power whilst avoiding the increasing problem of clock skew. A major objection to asynchronous design has, until now, been the issue of the feasibility of developing asynchronous designs at levels of complexity which are representative of commercial circuits. The AMULET1 work overcomes this objection by demonstrating that such circuits are now feasible. Whilst many questions still remain, this work represents a step forward towards the commercial exploitation of asynchronous circuits.

11 Acknowledgements

The work described in this chapter involved several members of the AMULET group at Manchester University. Paul Day and Nigel Paver were the full-time research staff on the AMULET1 project. They contributed many of the ideas incorporated in the design and carried out most of the design work. Without their skill and commitment, we would not now have working silicon. Steve Temple joined the research staff towards the end of the design phase; he made a significant contribution to the final chip composition and designed and carried out the tests on the silicon. Viv Woods and Jim Garside are members of academic staff associated with the project and both made substantial contributions to the design effort.

The AMULET1 design work described in this chapter was carried out as part of ESPRIT project 5386, OMI-MAP (the Open Microprocessor systems Initiative - Microprocessor Architecture Project). Subsequent work has been supported as part of ESPRIT project 6909, OMI/DE-ARM (the Open Microprocessor systems Initiative - Deeply Embedded ARM Applications project). The author is grateful for this support from the CEC.

The author is also grateful for material support in various forms from Advanced RISC Machines Limited, Acorn Computers Limited, Compass Design Automation Limited, VLSI Technology Limited and GEC Plessey Semiconductors Limited. The encouragement and support of the OMI-MAP and OMI/DE-ARM consortia are also acknowledged.

12 References

1. Dobberpuhl, D. W. et al., "A 200-MHz 64-b Dual-Issue CMOS Microprocessor", IEEE Journal of Solid-State Circuits, Vol. 27, No. 11, Nov. 1992, pp. 1555-1565.

2. Sutherland, I.E., "Micropipelines", The 1988 Turing Award Lecture, Communications of the ACM, Vol. 32, No. 6, June, 1989, pp. 720-738.

3. Paver, N.C., "The Design and Implementation of an Asynchronous Microprocessor", PhD Thesis, University of Manchester, June 1994.

4. Furber, S.B., "VLSI RISC Architecture and Organization", Marcel Dekker Inc., New York, 1989.

5. van Someren, A., and Atack, C., "The ARM RISC Chip: A Programmer's Guide", Addison-Wesley, 1993.

6. Furber, S.B., Day, P., Garside, J.D., Paver, N.C. and Woods, J.V., "A Micropipelined ARM", Proceedings of the IFIP TC 10/WG 10.5 International Conference on Very Large Scale Integration (VLSI'93), Grenoble, France, September 1993. Ed. Yanagawa, T. and Ivey, P. A. Pub. North Holland.

7. Furber, S.B., Day, P., Garside, J.D., Paver, N.C. and Woods, J.V., "AMULET1: A Micropipelined ARM", Proceedings of the IEEE Computer Conference, March 1994.

8. Paver, N.C., Day, P., Furber, S.B., Garside, J.D. and Woods, J.V., "Register Locking in an Asynchronous Microprocessor", 1992 IEEE International Conference on Computer Design: VLSI in Computers & Processors. October 1992.

9. Garside, J.D., "A CMOS VLSI Implementation of an Asynchronous ALU", IFIP Working Conference on Asynchronous Design Methodologies, April 1993. Ed. Furber, S. B. and Edwards, M. D. Pub. North Holland.

10. Yuan, J., and Svensson, C., "High-Speed CMOS Circuit Techniques", IEEE Journal of Solid-State Circuits, Vol. 24, No. 1, February 1989, pp. 62-70.