A Synthesis Algorithm for Modular Design of Pipelined Circuits

Maria-Cristina Marinescu
and Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
cristina@lcs.mit.edu,rinard@lcs.mit.edu

Abstract: This paper presents a synthesis algorithm for pipelined circuits. The circuit is specified as a collection of independent, loosely-coupled modules connected by queues. The synthesis algorithm transforms this asynchronous, modular specification into a synchronous, tightly-coupled, and fully pipelined circuit in which queues are implemented as finite buffers. Data is read from the buffers at the begining of each clock cycle, new values are computed, then the new results are written back into the buffers at the end of each clock cycle.

We have implemented a prototype synthesizer that is capable of automatically generating synchronous, fully pipelined implementations of modular specifications. This paper presents experimental results from this synthesizer.

1. INTRODUCTION

One successful way to manage the complexity of building very large-scale systems is to specify them as a collection of independent, loosely-coupled modules connected by streams, queues or pipes. Our present work describes a synthesis algorithm for arbitrarily complex and general pipelined circuits, starting from a modular, compact high-level specification.

The designer specifies the circuit as a set of modules connected by queues. The behavior of each module is specified using a set of rewrite

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35498-9_57
L. M. Silveira et al. (eds.), *VLSI: Systems on a Chip*

rules. Each rule reads data from one or more input queues, uses the data to compute new values, then writes the new values out to the output queues. Conceptually, each module executes independently and asynchronously with respect to the other modules. Because the queues insulate the modules from each other, the designer can use a modular design approach. He or she can first focus on developing each module in isolation, then use queues to connect the modules into a complete specification.

A primary advantage of this approach is that it enables the designer to reason about the behavior and correctness of each module in isolation without worrying about the concurrent behavior of the entire system. This reduces human effort and makes specifications simple, compact, clear, less prone to mistakes, and more easily verified. It also promotes the reuse of existing modules in new specifications. Finally, modular specifications are more suitable for automatic synthesis and simulation than non-modular ones and have good scalability characteristics. This model has proved to be useful in the Unix operating system and in various parallel programming models [Arvind and Nikhil, 1990; Gregory, 1987; Newton and Browne, 1992]. More recently it has been used to successfully model complicated hardware designs, where it has shown great promise in enabling very concise, clear specifications [Arvind and Shen, 1999; Poyneer et al., 1998].

A straightforward synthesis algorithm would implement this model directly in hardware. The problem with this approach is the queue management overhead. If the queues are implemented as asynchronous connections between independently operating modules, the system as a whole suffers from synchronization overhead as modules dynamically handshake to transfer data.

This paper presents an alternative approach: a synthesis algorithm that produces a tightly coupled, fully synchronous implementation of a set of modules connected with queues. The basic idea behind the synthesis algorithm is to automatically compose the module definitions to derive, at the granularity of individual clock cycles, a global schedule for the operations of the entire system, including the removal and insertion of queue elements. The resulting implementation executes in a completely synchronous, pipelined manner. At the beginning of each clock cycle, the modules read their inputs from the input queues and compute the next result. At the end of the clock cycle, the results are written to the output queues, overwriting the inputs from the beginning of the clock cycle. This synthesis algorithm delivers the best of both worlds: it allows the designer to use a modular, high-level specification and obtain an efficient, fully synchronous circuit.

The remainder of the paper is organized as follows. Section 2 presents an example illustrating our synthesis approach. Section 3 presents the synthesis algorithm and Section 4 presents the experimental results. Section 5 discusses related work; we conclude in Section 6.

2. EXAMPLE

We next present an example that shows how to use our approach to synthesize a simple pipelined processor. We use a processor as our example because we expect it will be familiar to a wide audience. Our approach and synthesis algorithms are, of course, generally applicable to wide range of circuits, not just processors.

Our example processor has an instruction memory, a program counter and a register file. Figure 1.1 presents the simplified pipeline that we use to implement the processor. The instruction fetch stage fetches instructions from the instruction memory into the instruction buffer; the register fetch stage moves the instruction from the instruction buffer to the register buffer, replacing the register names in the instruction with the contents of the corresponding registers. The compute and writeback phase computes the results and writes them back into the register file.

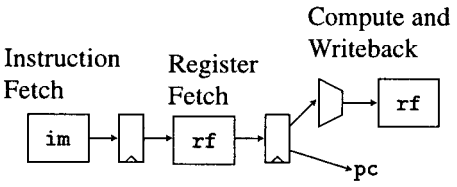

Figure 1.1 Simple Pipeline for Example

2.1 PROCESSOR STATE

Figure 1.2 presents the declaration of the processor state, which consists of the program counter `pc`, the instruction memory `im`, the register file `rf`, and two queues, `iq` and `rq`. Lines 4 and 5 declare the state as a set of state variables; lines 1 through 3 contain the type declarations for these variables. The type declarations include a 3 bit register name type `reg`, an 8 bit integer type `val`, an 8 bit integer type `loc` which represents the locations of instructions in the instruction memory, an instruction type `ins`, and a type `irf` for instructions whose register operands have been fetched from the register file. The instruction type is a tagged union type, similar to those found in ML [Milner et al., 1990]

and Haskell [Hudak et al., 1992]. Each instruction can be either an `INC` instruction, which increments the value in its single register argument, or a `JRZ` instruction, which tests the value in its register argument and, if the value is zero, jumps to the location in its location argument.

```
1 type reg = int(3), val = int(8), loc = int(8);
2 type ins = <INC reg> | <JRZ reg loc>;
3 type irf = <INC reg val> | <JRZ val loc>;

4 var pc : loc, im : ins[N], rf : val[8];
5 var iq = queue(ins), rq = queue(irf);
```

Figure 1.2 State Variables and Type Declarations for Example

2.2 QUEUES

Queues provide buffered, first-in, first-out connections between modules. There are several operations that modules can perform on a queue `q`:

- `head(q)`: Retrieves the first element in the queue.
- `tail(q)`: The rest of the queue `q` after the first element. Usually used to specify the new value of the queue after removing the first element.
- `insert(q,e)`: The queue `q` after inserting the element `e` at the tail of the queue `q`. Usually used to specify the new value of the queue after inserting a new element.
- `notin(q,e)`: Tests if the element `e` is not in the queue `q`.

Our specification models the pipeline buffers `iq` and `rq` in our example as queues.

2.3 UPDATE RULES

Figure 1.3 presents the code that implements the modules in our example. There are three modules, one for each pipeline stage. Each module is implemented by a set of *update rules*. Each rule has an enabling condition and a set of updates to the state. When the enabling condition evaluates to `true`, the rule is enabled and can execute, in which case its updates are atomically applied to the state. Conceptually, the execution of the system repeatedly chooses an enabled rule and executes it.

This is a standard model of asynchronous execution found, for example, in systems such as Unity [Chandy and Misra, 1988] and term rewriting systems [Baader and Nipkow, 1998].

```
// Instruction Fetch Stage
1: if true then iq = insert(iq,im[pc]); pc = pc+1;

// Register Operand Fetch Stage
2: if <INC r> = head(iq) and notin(rq, <INC r _>) then
     iq = tail(iq); rq = insert(rq, <INC r rf[r]>);
3: if <JRZ r l> = head(iq) and notin(rq, <INC r _>) then
     iq = tail(iq); rq = insert(rq, <INC rf[r] l>);

// Compute and Writeback Stage
4: if <INC r v> = head(rq) then
     rf = rf[r->v+1]; rq = tail(rq);
5: if <JRZ v l> = head(rq) and v = 0 then
     pc = l; iq = nil; rq = nil;
6: if <JRZ v l> = head(rq) and !(v = 0) then
     rq = tail(rq);
```

Figure 1.3 Update Rules for Example

We illustrate the execution of the system by going through the set of rules. The condition for the instruction fetch rule, rule 1, is `true`, which means that it is always enabled. When it executes, it fetches an instruction from the instruction memory and inserts it into the instruction queue `iq`. It also increments the program counter `pc` to set up the next fetch.

The two rules in the operand fetch stage, rules 2 and 3, remove instructions from the instruction queue, fetch the register operands, and insert them into the `rq`. Rule 2 processes `INC` instructions, and rule 3 processes `JRZ` instructions. Both rules use a form of pattern matching similar to that found in ML and Haskell. Consider rule 2. The enabling condition is `<INC r> = head(iq) and notin(rq, <INC r _>)`. The first clause of this condition, `<INC r> = head(iq)`, is true if an `INC` instruction is the first instruction in the instruction queue `iq`. Furthermore, if there is such an instruction, the clause matches and *binds* the variable `r` to the register name argument of the `INC` instruction. The variable `r` can then be used later in the rule to refer to this operand.

The second clause, `notin(rq, <INC r _>)` uses the binding to check for a read before write hazard. If there is a pending instruction waiting to execute that will write the register `r`, the machine must delay the operand fetch so that it fetches the value after the write. If there is a pending instruction that will write the register `r`, the instruction is in the `rq` queue. The clause `notin(rq, <INC r _>)` checks to make sure that there is no such instruction in `rq`, and the rule as a whole is enabled and can execute only if there is no hazard.

If the rule is enabled, it fetches the register operand and inserts the instruction, along with this operand, into the next queue in the pipeline, the `rq` queue. It also removes the instruction from the instruction queue. The other rules perform similar activities, removing elements from queues, processing the data in the elements to generate results, then inserting the results into the next queue or writing the result back into the register file. In particular, the update `rf = rf[r->v+1]` from the first rule in the compute and writeback stage, rule 4, sets element `r` of the register file `rf` to be `v+1`.

2.4 SYNTHESIS

In the abstract model of computation described above, the modules execute in a completely decoupled way. The rules execute whenever they are enabled, with the queues carrying results between modules. In effect, the queues decouple the modules, enabling the designer to focus on each module in turn. This design methodology scales to very large systems, including systems with hierarchically defined modules. The only problem is that an efficient hardware implementation must be tightly coupled and synchronous. Ideally, the stages of the processor would execute in a strict pipeline, with the queues implemented as hardware buffers and each stage reading the value from the previous stage in the same clock cycle as the new value is written into the register. The next section presents a synthesis algorithm that accomplishes this goal.

3. ALGORITHM

Given a system specification, the synthesis algorithm combines the operations in the rules first into a global schedule, then into a synchronous circuit that implements the specification. The basic approach is, at each clock cycle, to give each rule an opportunity to execute. If a rule is enabled at that cycle, it will execute. The challenge with this approach is to ensure that the final result at the end of the cycle correctly reflects the atomic execution of all of the rules that executed in that cycle. We meet this challenge by symbolically executing the rules in sequence, with

each rule operating on the output of the previous rule. The final result is an expression for each state variable. This expression is the new value of the state variable in the next clock cycle, and reflects the combined updates of all the rules that executed in the previous clock cycle.

To avoid the problem of an excessively long clock cycle, the algorithm, when possible, relaxes the enabling condition at each rule so that it is evaluated in the initial state, at the beginning of the clock cycle, rather than in the state produced by the previously executed rule. In particular, this technique ensures that data from state variables moves through at most one module in each clock cycle, which in turn ensures that the critical path of the circuit does not cross module boundaries. The clock cycle time of the system is therefore determined by the modules, not how they are connected together. The algorithm consists of the following phases:

- **Rule Numbering:** The algorithm numbers rules for symbolic execution, determining the intermediate state in which each rule will be evaluated. Figure 1.4 illustrates the numbering of all different versions of the state variables for all the rules in our previous example. As this example shows, the numbering is set up so that each rule reads the version of the state variables produced by the previous rule.

- **Relaxation:** When possible, the algorithm relaxes the calculation of the enabling condition for each rule so that it is evaluated in the initial state, not the intermediate state from the previous rule. This transformation has the effect of limiting the critical path that determines the length of the clock cycle.

- **Queue Finitization:** In the initial specification, the queues have unbounded length. Based on input from the designer, the algorithm chooses a finite length for each queue. It then modifies the rules to ensure that no queue ever exceeds its finite length. The key issue is to ensure that no rule ever executes if there will be no room for its result in the output queues. This is more difficult than it may sound, because each rule must take into account the number of items in the queue at the beginning of the clock cycle, the number of elements inserted and removed by rules before it in the evaluation order, and the number removed by rules after it in the evaluation order.

- **Symbolic Execution:** The algorithm symbolically executes the rules in sequence to obtain an expression for each state variable.

The expression is the value of the variable in the next clock cycle. Because rules may not be enabled in a given state and may therefore not execute, the expressions contain conditionals.

- **Optimizations:** The algorithm optimizes the representation by performing common sub-expression elimination to eliminate any duplication, and mutual exclusion testing to eliminate executions that can never actually occur (i.e. false paths in the circuit).

- **Verilog Generation:** The algorithm generates one or more hardware registers for each state variable, depending on its type. For each state variable, the value in the next clock cycle is determined by the combinational logic implementing the corresponding determined expression.

We next discuss the more complicated phases of the synthesis algorithm.

```
if true then iq1= insert(iq0,im[pc0]); pc1= pc0+1;
if <INC r> = head(iq1) and notin(rq1, <INC r _>) then
    iq2= tail(iq1); rq2=insert(rq1, <INC r rf1[r]>);
if <JRZ r l> = head(iq2) and notin(rq2, <INC r _>) then
    iq3= tail(iq2); rq3 = insert(rq2, <INC rf2[r] l>);
if <INC r v> = head(rq3) then
    rf4= rf3[r->v+1]; rq4= tail(rq3);
if <JRZ v l> = head(rq4) and v = 0 then
    pc5= l; iq5= nil; rq5 = nil;
if <JRZ v l> = head(rq5) and !(v = 0) then
    rq6= tail(rq6);
```

Figure 1.4 Numbered Rules for Example

3.1 RELAXATION

The rule numbering in Figure 1.4 suffers from an excessively long clock cycle. Consider, for example, the system starting out with nothing in any of the queues. The last version of the state variables reflects the entire fetch and execution of the next instruction. Obviously, we would like the fetch and execution to be pipelined over multiple clock cycles. We achieve this goal by relaxing the versions tested in the enabling conditions of each rule — we replace each version of each state variable with the earliest safe version. An earlier version of v_j, name v_k, is safe if the following property holds:

If the rule's enabling condition, C is true with v_j replaced by v_k, then it is also true with v_j, i.e. $C[v_k/v_j]$ implies C.

This transformation is valid for two reasons:

- **Safety:** After the transformation, each rule is enabled in a subset of the states in which it was enabled before the transformation, and, if enabled, produces the same result as before the transformation. So each execution of the transformed system is also an execution of the original system.
- **Liveness:** The transformation never completely disables a rule — the transformed enabling condition tests the original state, and the rule executes if it is enabled in this state.

Figure 1.5 presents the transformed system in our example. A key property that enables this transformation is that if a rule that tests the element at the head of a queue is enabled, it remains enabled if additional elements are inserted at the tail of the queue. This property makes it possible to relax the rules in the example so that they test the initial version of each queue instead of the version produced by earlier rules.

In many cases, the algorithm can order the rules to perform queue operations in the following order: first checks of the form `notin(q,e)` that test that an element is not in a queue, then insertions into the tail of the queue, then tests that the head of the queue satisfies a given property, then removals from the head of the queue. Being able to put the rules in this order is sufficient (but not necessary) to ensure that the algorithm will be able to relax the enabling conditions so that they all test the initial version of each queue.

```
if true then iq1= insert(iq0,im[pc0]); pc1= pc0+1;
if <INC r> = head(iq0) and notin(rq0, <INC r _>) then
    iq2= tail(iq1); rq2=insert(rq1, <INC r rf1[r]>);
if <JRZ r l> = head(iq0) and notin(rq0, <INC r _>) then
    iq3= tail(iq2); rq3 = insert(rq2, <INC rf2[r] l>);
if <INC r v> = head(rq0) then
    rf4= rf3[r->v+1]; rq4= tail(rq3);
if <JRZ v l> = head(rq0) and v = 0 then
    pc5= l; iq5= nil; rq5 = nil;
if <JRZ v l> = head(rq0) and !(v = 0) then
    rq6= tail(rq6);
```

Figure 1.5 Relaxed Rules for Example

The relaxation algorithm proceeds as follows. It processes the rules of the system in the order in which they are numbered. At each rule,

it repeatedly attempts to replace the current version of the state variables in the enabling condition with the previous version. This attempt succeeds if the enabling condition with the previous version of the state variables implies the enabling condition with the current version or if the enabling conditions are mutually exclusive. The implication test and mutual exclusion tests are performed using a combination of resolution [Ballantyne, 1982] and a set of simplification and reduction rules, and operate on the enabling conditions once they have been transformed into conjunctive normal form.

3.2 QUEUE FINITIZATION

When the queues are implemented in hardware, there is a specific number of entries allocated for the queue, and the synthesis algorithm must generate a circuit that does not exceed that length. The algorithm therefore analyzes the rules to determine the circumstances under which a queue may grow beyond its hardware limit. It then modifies the enabling conditions to ensure that the queues never exceed the limit.

Conceptually, the generated circuit maintains several counters for each queue: a counter L_q that contains the number of elements in q at the beginning of the clock cycle, a counter I_q that maintains, for each rule, the net number of elements that preceding rules insert into q (this number is the number of elements inserted minus the number removed), and a counter R_q that maintains, for each rule, the number of elements that succeeding rules remove from q. Both of these counters are dynamically generated using combinational logic, and count only insertions and removals from rules that are enabled in the current clock cycle. There is also the hardware limit $\mathtt{N}_q$ of the maximum number of queue entries.

The basic idea is to augment the enabling condition for each rule that inserts an element into q so that it does not execute unless a subsequent rule clears the queue or $L_q + I_q - R_q < \mathtt{N}_q$. Because the values of the counts depend directly on the enabling conditions, it may be more efficient to simply test combinations of enabling conditions rather than computing the counts explicitly. Figure 1.6 presents our example after the application of the queue finitization algorithm. In this figure, $\mathtt{length}(q) = L_q + I_q$.

Note that because the values of I_q and R_q affect the enabling conditions, it is possible for there to be a cycle of dependences between the different values of these counters. This occurs, for example, when there is a cycle of rules waiting for each other to remove elements from queues. In the worst case, there may simply be no way to avoid deadlock without changing the hardware to add more space in the queues.

```
if length(iq0) < Niq or
  (<INC r> = head(iq0) and notin(rq0, <INC r _>)) or
  (<JRZ r l> = head(iq0) and notin(rq0, <INC r _>)) or
  (<JRZ v l> = head(rq0) and v = 0) and
  length(rq0) < Nrq or length(iq0) < Niq or
  <INC s _> = head(rq0) or
  (<JRZ v l> = head(rq0) and v = 0) or
  (<JRZ v l> = head(rq0) and !(v = 0) then
    iq1 = insert(iq0,im[pc0]); pc1 = pc0+1;
if <INC r> = head(iq0) and
  notin(rq0, <INC r _>) and
  length(rq0) < Nrq or
  <INC s _> = head(rq0) or
  (<JRZ v l> = head(rq0) and v = 0) or
  (<JRZ v l> = head(rq0) and !(v = 0) then
    iq2 = tail(iq1); rq2=insert(rq1, <INC r rf1[r]>);
if <JRZ r l> = head(iq0) and
  notin(rq0, <INC r _>) and
  length(rq0) < Nrq or
  <INC s _> = head(rq0) or
  (<JRZ v l> = head(rq0) and v = 0) or
  (<JRZ v l> = head(rq0) and !(v = 0) then
    iq3 = tail(iq2); rq3 = insert(rq2, <INC rf2[r] l>);
if <INC r v> = head(rq0) then
    rf4 = rf3[r->v+1]; rq4 = tail(rq3);
if <JRZ v l> = head(rq0) and v = 0 then
    pc5 = l; iq5 = nil; rq5 = nil;
if <JRZ v l> = head(rq0) and !(v = 0) then
    rq6 = tail(rq6);
```

Figure 1.6 Rules in Example After Queue Finitization

But even if there are cycles of rules waiting for each other to remove elements, it may still be possible for the synthesis algorithm to generate a deadlock-free circuit without increasing the queue length.

The key insight in this case is that finitization will not introduce deadlock if there is a way for existing elements to be removed from all of the queues so that there is room for new elements. Assume the sequence of rules $R_j R_{j+1} ... R_{j+m}$ with enabling conditions $C_j C_{j+1} ... C_{j+m}$ creates a cycle for the current rule R_i with queue q, where $\texttt{length}(q) = \texttt{N}_q + R_q$. If C_i implies $\forall 0 \leq l \leq m, C_{j+l}$, then all of the rules in the cycle can execute. Otherwise, none of them can.

3.3 SYMBOLIC EXECUTION

Symbolic execution determines a new value for each state variable at the end of the clock cycle in terms of the values at the start of the clock cycle. It does this by substituting out the intermediate versions of each state variable. The result is an expression, in the original versions of the state variables, for each use of each state variable in the system. The versions at the last rule are latched back into the state variables at the end of the clock cycle, and provide the initial values for the start of the next clock cycle.

3.4 OPTIMIZATIONS

To improve the quality of the synthesized circuit, the compiler optimizes the expressions, using common sub-expression elimination and mutual exclusion testing. If an expression contains a value that will never actually occur in practice because the conditions required to obtain the value are mutually exclusive, the computation of that value is eliminated from the expression. A typical example is a value obtained if both a `JRZ` and an `INC` instruction is at the head of the instruction queue. Obviously, the instruction must be either a `JRZ` instruction or an `INC` instruction, but not both. So such a value will never be computed in the actual circuit. The mutual exclusion testing is implemented using resolution, simplification, and reduction.

We illustrate the symbolic execution and optimizations principle by presenting the final value of the instruction queue `iq`. If there is a taken branch, the instruction queue is cleared. If there is already an instruction at the head of the instruction queue that can go through the register fetch stage, the final result is obtained by inserting the new instruction into the tail of the queue and removing the instruction from the head of the queue. Otherwise, the circuit checks to see if there is an empty entry

in the instruction queue. If so, it fetches another instruction; if not, the instruction queue does not change.

Figure 1.7 presents the result of the expression evaluation; note the introduction of the temporary variables `t1`, `t2`, `t3`, and `t4`. These variables will turn directly into combinational logic in the final implementation of the circuit.

```
let
  t1 = <INC r> = head(iq0) and notin(rq0, <INC r _>)
  t2 = <JRZ r l> = head(iq0) and notin(rq0, <INC r _>)
  t3 = insert(iq0, im[pc0])
  t4 = tail(t3)
iq6 =
  if <JRZ v l> = head(rq0) and v = 0 then nil
  else if t1 then t4
  else if t2 then t4
  else if length(iq0) < Niq) then t3
  else iq0
```

Figure 1.7 Result of Symbolic Execution for `iq`

3.5 VERILOG GENERATION

The final step is to generate synthesizable Verilog for the circuit. The basic approach is that each state variable is implemented as one or more hardware registers, with the expressions generated during the symbolic execution providing the new values for the state variables at the end of each clock cycle.

The Verilog generation is straightforward. The algorithm generates combinational logic that computes the value of each expression, then connects the computed values to the inputs of the hardware registers that implement the corresponding state variables. In the future we may explore implementations that use more complicated synthesis algorithms for operations that are expensive to implement in combinational logic.

The compiler currently uses Verilog arrays to implement memories such as the instruction memory in our example. Current memory implementations are single ported, but we are exploring ways to obtain multi-ported memories, either under the control of the designer or automatically as part of the synthesis algorithm. We are also exploring the use of SRAM or DRAM to implement larger memories. Queues are implemented as registers.

Benchmark	Cycle Time	Area (NAND2 gates)	Map Effort	Constraints
Bubblesort	9.59ns	≈ 5121	medium	Clk = 10
Butterfly	9.61ns	≈ 5170	medium	Clk = 10
Processor	10.48ns	≈ 4830	low	none

Table 1.1 Benchmark Characteristics

4. EXPERIMENTAL RESULTS

We have implemented a prototype synthesis system based on the algorithm presented in this paper. We have used this algorithm to generate synthesizable Verilog implementations for the benchmarks described below. These Verilog implementations were tested using the NCVerilog by Cadence, then synthesized using the Synopsys Design Compiler to an industry standard .25 micron standard cell process.

The first benchmark implements Bubblesort for eight 8-bit numbers. The second benchmark implements a butterfly network similar to the ones used in bitonic sorting networks and in FFTs. The last benchmark is an 8-bit pipelined processor specification. The synthesized, gate level model of the processor was then regression tested using the ASIC vendor's simulation libraries in order to confirm correct synthesized functionality. Table 1.1 presents several benchmark characteristics.

5. RELATED WORK

The synthesis of hardware from various description languages has been and continues to be an active area of research [Micheli, 1994]. In this section we discuss systems for specifying custom microprocessors, synchronous dataflow languages, and recent work using term rewriting systems.

There is a large market for embedded processors customized for a specific application. Researchers have proposed to support the development of such systems by providing languages that allow the designer to quickly describe a customized architecture [Pyo et al., 1992; Park and Walker, 1988]. The research presented in our paper, on the other hand, is designed to support the development of arbitrary circuits, not just microprocessors.

Other researchers have proposed a design methodology based on synchronous dataflow [Ho et al., 1998]. While the resulting specifications contain modules, the connections between modules are synchronous, which forces the designer to understand the global timing of the circuit when designing each module. The research presented in our paper uses asynchronous queues to connect modules. The synthesis algorithm

automatically derives the global schedule of operations in the circuit, freeing the designer from the need to understand the global timing.

The research most closely related to ours is the work of James Hoe and Arvind on the synthesis of circuits specified as term rewriting systems [Hoe and Arvind, 1999; Hoe et al., 1997]. The basic goal is the same: to synthesize synchronous implementations of modular, queue-based specifications. There are differences, however, in the synthesis algorithms. In particular, their approach executes multiple rewrite rules in the same cycle only if they are completely independent. In our approach, multiple dependent rules may execute in the same clock cycle, with the final result reflecting the combined effect.

6. CONCLUSION

Understanding how to manage the complexity of building large-scale systems is a difficult, challenging, and important problem. This paper presents an approach based on specifying the system as a set of independent, parallel modules connected by queues. This approach enables the designer to control the complexity of the design process by first developing each module in isolation, then using queues to combine the modules and specify the complete system.

The successful use of this design methodology for circuits requires a synthesis algorithm that can translate the asynchronous, loosely-coupled specification into a synchronous, fully pipelined circuit. This paper presents such an algorithm. Our initial experimental results from an implementation of this algorithm provide encouraging evidence that it can be used to deliver efficient pipelined implementations of modular specifications that use queues.

References

Arvind and Nikhil, R. (1990). Executing a program on the MIT tagged-token dataflow architecture. *IEEE Transactions on Computers*, 39(3).

Arvind and Shen, X. (1999). Design and verification of processors using term rewriting systems. Technical Report CSG Memo 419, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Baader, F. and Nipkow, T. (1998). *Term Rewriting and All That.* Cambridge University Press.

Ballantyne, M. (1982). Automatic deduction. Technical Report STAN-CS-82-937, Dept. of Computer Science, Stanford Univ., Stanford, Calif.

Chandy, K. M. and Misra, J. (1988). *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Mass.

Gregory, S. (1987). *Parallel Logic Programming in PARLOG: The Language and Its Implementation.* Addison-Wesley, Reading, Mass.

Ho, W., Lee, E., and Messerschmitt, D. (1998). High level data flow programming for digital signal processing. *VLSI Signal Processing, III*, pages 385–395.

Hoe, J. and Arvind (1999). Hardware synthesis from term rewriting systems. Technical report, Laboratory for Computer Science, Massachusetts Institute of Technology.

Hoe, J., Rinard, M., and Arvind (1997). An exercise in high-level architectural descriptions using a synthesizable subset of term rewriting systems. Technical Report CSG Memo 403, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA.

Hudak, P., Peyton-Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W., and Peterson, J. (1992). Report on the programming language Haskell: a non-strict, purely functional language (version 1.2). *SIGPLAN Notices*, 27(5).

Micheli, G. D. (1994). *Synthesis and Optimization of Digital Circuits.* McGraw-Hill.

Milner, R., Tofte, M., and Harper, R. (1990). *The Definition of Standard ML.* The MIT Press, Cambridge, Mass., Cambridge, MA.

Newton, P. and Browne, J. C. (1992). The CODE 2.0 graphical parallel programming language. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, Washington, DC.

Park, N. and Walker, A. (1988). Sehwa: A software package for synthesis of pipelines from behavioral specifications. *IEEE Transactions on Computer-Aided Design*, 7(3).

Poyneer, L., Hoe, J., and Arvind (1998). A TRS model for a modern processor. Technical Report 408, Computation Structures Group, MIT Laboratory for Computer Science.

Pyo, I., Su, C., Huang, I., Pan, K., Koh, Y., Tsui, C., Chen, H., Cheng, G., Liu, S., Wu, S., and Despain, A. M. (1992). Application-driven design automation for microprocessor design. In *Proceedings of 29th Design Automation Conference.*