High-level Specification and Efficient Implementation of Pipelined Circuits

Maria-Cristina Marinescu

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
email: cristina@lcs.mit.edu

Martin Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
rinard@lcs.mit.edu

Abstract— This paper describes a novel approach to high-level synthesis of complex pipelined circuits, including pipelined circuits with feedback. This approach combines a high-level, modular specification language with an efficient implementation. In our system, the designer specifies the circuit as a set of independent modules connected by conceptually unbounded queues. Our synthesis algorithm automatically transforms this modular, asynchronous specification into a tightly coupled, fully synchronous implementation in synthesizable Verilog.

I. Introduction

An important conflict in hardware design is providing a simple, high-level way of specifying a system without sacrificing the efficiency of the resulting implementation. An efficient implementation is usually synchronous and is obtained as the result of globally scheduling all of the operations in the system. In contrast, designers usually find it easier to specify the system as a collection of reusable, concise, loosely-coupled components.

This paper describes an approach that meets both these challenges. The designer specifies the circuit as a set of independent modules connected by queues. Conceptually, the queues have unbounded length, which decouples the modules in the design. Unfortunately, implementing this abstraction directly in hardware using asynchronous queues produces a circuit with significant handshaking overhead between modules. Our synthesis algorithm therefore automatically transforms the modular, asynchronous specification into a tightly coupled, fully synchronous implementation in synthesizable Verilog. It is designed to handle complex pipelined circuits, including pipelined circuits with feedback.

It is important to understand the design advantages of this approach: the asynchrony at the specification level enables the designer to compose modules together into a complete system without the need to deal with complex global issues such as the coordinated assignment of operations to clock cycles. The approach scales to large circuits, and the circuits are easier to modify, debug, reuse, and formally verify.

The key idea behind our synthesis algorithm is to automatically compose the module specifications 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 efficiently in a completely synchronous, pipelined manner. We have built a prototype synthesizer that implements our synthesis algorithm and present experimental results from this synthesizer. We evaluate the efficiency of this implementation by measuring the area and clock cycle time of the circuits that it generates. For our benchmark design, our algorithm generates a circuit with area and clock cycle time comparable to those of a hand-written Verilog model that implements the same basic functionality.

This paper makes the following contributions:

- **Approach:** It presents a new approach to high-level synthesis. This approach combines the best of both worlds: a modular, asynchronous specification language and an automatically generated synchronous, fully pipelined implementation.
- **Algorithms:** It presents a **relaxation** algorithm for decreasing the clock cycle time and a coordinated **global scheduling** algorithm for mapping the individual operations of the modules into clock cycles. The latter is the enabling technology for efficient pipelining, as it allows the data to move together across the circuit even when the pipeline buffers are full.
- **Experimental Results:** It presents experimental results that demonstrate the effectiveness of the technique.

The remainder of the paper is organized as follows. Section II illustrates how a system is specified using rewrite rules. Section III presents the synthesis algorithm. Section IV presents the experimental results. Section V discusses related work. We conclude in Section VI.

II. Specification Example

We illustrate our approach by presenting a short example. Our example is a linear pipelined datapath with associated control functionality. Note that none of the techniques of our approach is specific to this particular class of circuits.

Specifying the behavior of a system consists of two steps:

- **Module Specification:** The designer specifies the behavior of each module as a set of update rules. Modules communicate *solely* using FIFO queues.

[*] This research was supported in part by NSF Grant CCR-9702297.

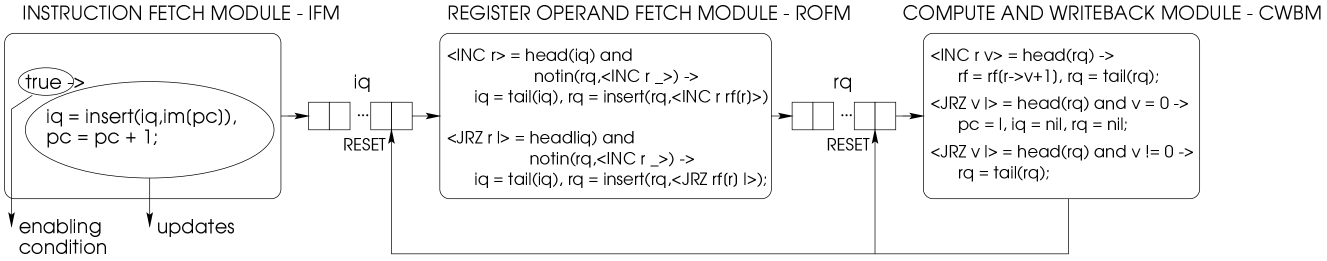
INSTRUCTION FETCH MODULE - IFM
REGISTER OPERAND FETCH MODULE - ROFM
COMPUTE AND WRITEBACK MODULE - CWBM
true ->
iq = insert(iq,im(pc)),
pc = pc + 1;
enabling condition
updates
iq
RESET
<INC r> = head(iq) and
notin(rq,<INC r _>) ->
iq = tail(iq), rq = insert(rq,<INC r rf(r)>)
<JRZ r l> = headliq) and
notin(rq,<INC r _>) ->
iq = tail(iq), rq = insert(rq,<JRZ rf(r) l>);
rq
RESET
<INC r v> = head(rq) ->
rf = rf(r->v+1), rq = tail(rq);
<JRZ v l> = head(rq) and v = 0 ->
pc = l, iq = nil, rq = nil;
<JRZ v l> = head(rq) and v != 0 ->
rq = tail(rq);

Fig. 1. Specification Example

- **State Declarations:** The designer specifies the state of the system as a set of typed variable declarations.

A. Modules

Fig. 1 shows the three functional modules in our example and the queues that interconnect them. Each module is implemented by a set of *update rules*. An update 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 [7] and term rewriting systems [2]. In our implementation, priority is given to rules according to the textual ordering in the specification.

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)`: Returns the rest of `q` after the first element.
- `insert(q,e)`: Returns the queue `q` after inserting the element `e` at the end of `q`.
- `notin(q,e)`: Returns *true* if the element `e` is not in `q`; otherwise returns *false*.
- `q = nil`: Resets the queue to be empty.

We illustrate the conceptual model of execution in our system by discussing the operation of the rules in our example. We would like to emphasize that this asynchronous model of execution is used primarily to reason about the abstract behavior of the modules and the correctness of the system. It does not directly reflect the actions of the generated circuit.

The condition for the rule in module IFM is `true`, which means that the rule 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 module ROFM remove instructions from `iq`, fetch the register operands, and insert them into `rq`. The first rule processes `INC` instructions, and the second one processes `JRZ` instructions. Both rules use a form of pattern matching similar to that found in ML and Haskell. The enabling condition of the first rule is `<INC r> = head(iq) and notin(rq, <INC r _>)`. If the first clause is true, the clause matches and *binds* the variable `r` to the register name argument of the `INC` instruction, to be used later in the rule when reffering 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 delays the operand fetch so that it fetches the value after the write (this translates into *stalling*[1]). If there is a pending instruction that will write `r`, the instruction is in `rq`. The clause `notin(rq, <INC r _>)` checks to make sure that there is no such instruction in `rq`. The rule as a whole is enabled and can execute only if there is no hazard. If enabled, the rule atomically executes the block in the right-hand-side of the arrow.

The other rules perform similar actions. The update `rf = rf[r->v+1]` from the first rule in the compute and writeback module sets element `r` of the register file `rf` to be `v+1`. The updates `iq = nil/rq = nil` clear the queues `iq/rq`.

B. State

```
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);
```

Fig. 2. State Variables and Type Declarations for Example in Fig. 1

Line `4` and `5` in Fig. 2 present the state declarations, which consist of the following state variables: a program counter `pc`, an instruction memory `im`, a register file `rf`, and two queues, `iq` and `rq`. 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

[1] This is not a particularity of our algorithm, but rather what the original description specifies. The machine could as easily generate, for example, bypassing logic if this choice is explicitly made in the specification.

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 and Haskell. To keep the example clear, the instruction set contains only an `INC` instruction, which increments the value in its single register argument, and 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.

III. Synthesis Algorithm

The synthesis algorithm takes an asynchronous specification and converts it into a synchronous implementation by generating a global schedule for all of the operations in the rules. This schedule enables the synchronous and concurrent execution of multiple rules per clock cycle and produces a circuit that, when no hazards are present, reads and writes each queue in the same cycle. It implements each queue as a finite hardware buffer.

The basic approach is to give each rule an opportunity to execute at each cycle. The challenge is to ensure that the final result at the end of the cycle correctly reflects the sequential, atomic execution of all of the rules that execute in that cycle. The algorithm meets this challenge by symbolically executing the rules in sequence, with each rule operating on the output of the previous[2] rule. The derived expression for each state variable represents its new value at the beginning of the next clock cycle.

The synthesis algorithm assumes that each operation is implemented by a dedicated hardware component. It is conceivable to modify the algorithm as to give the designer the option of trading parallelism for silicon area.

The algorithm consists of six phases:

- **Associating Versions With Each State Variable:** Order the rules[3] for symbolic execution and compute the version of each state variable that each rule accesses. The first rule will read version 0 of the variables and compute version 1. The second rule will read version 1 and compute version 2 and so on. By feeding the output of the previous rule into the next rule, we establish an initial schedule for symbolic execution.
- **Relaxation:** The result of the operation performed in the previous step suffers from an excessively long clock cycle, as rule execution is completely sequentialized. The goal of the relaxation is to shorten the critical path **within** each clock cycle. Whenever possible, the algorithm relaxes the calculation of the **enabling condition** for each rule so that it is evaluated in the initial state (at the beginning of the clock cycle) rather than in the state created by the previously executed rule. To maintain correctness, the updates still execute sequentially if they operate on the same state variable. This transformation ensures that each element of data traverses at most one module per clock cycle, producing an acceptable critical path for the circuit. By increasing the parallelism in this way, we shorten the clock cycle of the circuit, and, indirectly, increase its throughput. Relaxation does **not** insert or remove delays in/from the circuit.
- **Global Scheduling:** In the initial specification, queues have unbounded length. But the hardware implementation must have a finite, specific number of entries allocated for each queue. Given a designer-specified length for each queue, the synthesis algorithm must generate an implementation that does not exceed that length. In the actual hardware, a given length of 1 for each queue translates into the synthesis of a standard pipeline.
- **Symbolic Execution:** Next, the algorithm symbolically executes all of the rules in sequence. An expression is generated for each state variable that reflects all of the possible updates of that variable for that clock cycle. This expression represents the value of the variable in the next clock cycle. Since only a subset of the rules may fire in a given clock cycle, the expressions contain conditionals.
- **Optimizations:** The synthesis algorithm next applies a spectrum of optimizations geared towards avoiding unnecessary replication of hardware and eliminating false paths in the implementation. These optimizations currently include common sub-expression elimination and mutual exclusion testing for the expressions derived at symbolic execution. If an expression contains a value that will never actually occur in practice because the conditions required to obtain that value are mutually exclusive, its computation is eliminated from the expression. The mutual exclusion testing is implemented using resolution [3] and a set of reduction and simplification rules.

  Fig. 3 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 l>)
   t2 = <JRZ r l> = head(iq0) and
                        notin(rq0,<INC r l>)
   t3 = insert(iq0,im0[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
```

Fig. 3. Result of Symbolic Execution for `iq`

- **Verilog Generation:** In the final phase we generate synthesizable Verilog for the optimized expressions in the previous step. Each state variable is implemented as one or more registers, depending on its type; each memory variable as a library block. Queues are implemented as hardware registers. The derived expression for each state variable evaluates to the new value that gets written back into the state at the beginning of the next clock cycle.

We next discuss the two more complicated phases of the al-

[2] Previous and next refer to the textual ordering of the rules in the original specification.

[3] Our implementation uses textual ordering of the rules.

gorithm in turn.

A. Relaxation

The execution of a rule R can update state variables tested by a subsequent rule R'. If this is the case, then R' has to wait for R to execute and update the state, before testing its precondition. But if we can prove that the execution of R will not disable the enabling condition of R', we can relax the precondition of R' to test the state before R executes. This transformation exposes parallelism in the specification, reducing the length of the critical path of the circuit.

In our example, relaxing the rules pipelines the instruction fetch and execution over multiple clock cycles, thus reducing cycle time.

Relaxation is the process of replacing each version of each state variable with its earliest[4] safe version. An earlier version of v_j, named v_k, is safe if the following property holds:

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

This is an application of the following more general rule: Assume a predicate $P[e/d]$ (i.e. the predicate P with the expression d replaced by another expression e) implies P. Then for any rule with precondition P, we can (subject to liveness concerns) use the predicate $P[e/d]$ instead of P.

This transformation is valid because of two reasons:

- **Partial Correctness:** If a rule in the transformed numbering executes, the rule would also execute in the original numbering and yield the same result. This takes care of the safety issue.
- **Liveness:** Since the rule in the transformed numbering tests the initial state, if a rule is enabled in the original numbering but not in the transformed one, *some* rule executes in the transformed numbering. This ensures liveness.

The algorithm processes the rules in reverse order, repeatedly attempting to replace the current version of each variable in the enabling condition of the rule, with the previous corresponding version, starting from the immediately preceding rule. Fig. 4 shows how to obtain a new expression from an initial expression `Exp`, by replacing v_j in `Exp` with its earliest safe version. P_l stands for the enabling condition of the rule that contains `Exp`. A replacement is successful if either of the two is true:

- The enabling condition with the earlier version instead of the current one implies the enabling condition with the current version.
- The condition of the rule that computes the earlier version and the current enabling condition are mutually exclusive.

The relaxation algorithm is especially well suited for use with queues. An element inserted at the tail of the queue does not affect the element that was at the head of the queue before the insertion. Rules that test the first element of a queue remain enabled regardless of the number of elements inserted at the tail of the queue, provided that no rule previously removes the head of the queue. This property allows such a rule to test the initial version of the queue, rather than versions produced by earlier rules.

```
replace ($v_j$,Exp)
if version($v_j$) = 0
    then Exp
    else $v_k$ = earlier-version($v_j$)
         $P_k$ = rule-that-updates($v_j$)
         if ($P_k$,$P_l$) mutual exclusive
            then replace($v_k$,Exp[$v_k$/$v_j$])
            else if $P_l[v_k/v_j]$ implies $P_l$
                then replace($v_k$,Exp[$v_k$/$v_j$])
                else Exp
```

Fig. 4. Relaxation Algorithm

Conceptually, the algorithm could include an initial phase that can in many cases order the modules/rules so as to match the flow of data in the pipeline. Being able to put the rules in this order is sufficient (but not necessary) to ensure that they all test the initial version of each queue.

Currently, the framework does not provide a flexible support for trading concurrency for cycle time. By default, if two rules simultaneously evaluate their preconditions to true, they are both going to fire in the current clock cycle, either in parallel if there are no data dependencies or sequentially if there are.

B. Global Scheduling

The scheduler augments each rule that inserts an element into a queue to ensure that it never overflows any of the buffers that implement the queues in hardware. The basic approach is to assume all queues are within length at the beginning of the clock cycle and schedule only those rules for firing that are 1) enabled and 2) whose combined execution leaves the queue within its length at the *end* of the clock cycle. All the other rules remain unchanged. As part of this process, queue insertions are prioritized. In hardware, global scheduling corresponds to generating the control signals for the combinational logic.

Global scheduling is the enabling technology for efficient pipelining. The key insight is that, every clock cycle, the number of rules that can execute and insert into a queue `q` can be bigger than the number of empty slots in `q`, without causing `q` to overflow. The condition is that enough rules will also execute in that clock cycle and remove elements from `q`, leaving it within length at the end of the clock cycle. Applying this mechanism boosts the throughput of the circuit.

B.1 Basic Concepts

We define a **queue path** using a **rule graph**. The nodes in the graph are the rules. There is a directed edge between two rules if the first inserts items into a given queue and the second removes items from the same queue. By definition, the specification is acyclic if there are no cycles in the rule graph and cyclic if there are.

[4] Earlier here refers to the ordering established in the first step of the algorithm.

By definition, a rule is an appending rule if its set of updates contains at least one insertion of an element into some queue.

B.2 Acyclic Specifications

Acyclic specifications contain no cyclic queue paths. For acyclic specifications, the scheduling algorithm ensures that the queues do not overflow by computing an additional constraint as shown in Fig. 5. We use R' for rule R augmented with the corresponding additional constraints and $Room(q)$ for the number of empty locations in q at the beginning of the clock cycle. Function $eval(R')$ returns 0 if R' is $false$ and 1 if R' is $true$. $index(X)$ returns the set of indices of all the rules in X. The constraint counts the number of elements in each queue at the beginning of each clock cycle. It also considers queue removals and previous insertions to augment the enabling condition of each rule so that it does not execute if it would overflow the queue.

```
for each rule $R_i$ [in topological sort order]
    $Q = \{q$ | $R_i$ inserts into $q\}$
    if $Q$ != nil
    then for each $q \in Q$
        $I = \{R_j$ | $R_j$ inserts into $q\}$
        $D = \{R_k$ | $R_k$ removes from $q\}$
        $Select(R_i, q) =$
            if ($R_i$ is the only rule in $I$) or
             $\forall i1, i2 \in I, (i1, i2)$ mutually exclusive
            then $''\forall k \in index(D).$
                    $Room(q) + \Sigma eval(R'_k) > 0''$
            else $''\forall k \in index(D).\forall j \in index(I).j < i.$
                    $Room(q) + \Sigma eval(R'_k) > \Sigma eval(R'_j)''$
    else NOP
```

Fig. 5. Computing the Additional Constraints for a Rule in an Acyclic Specification

B.3 Generalization for Cyclic Specifications

Introducing additional enabling conditions raises the possibility of deadlock. For acyclic specifications, this is not an issue because the acyclicity ensures that the queues will eventually drain, enabling rules that were originally suspended for lack of space. But this line of reasoning does not hold for cyclic specifications. The key insight is that the additional enabling conditions need not introduce deadlock if there is a way to coordinate the removals and insertions of elements from all of the queues in the cycle so that the removal of each element leaves room for the insertion of the element behind it. The algorithm for cyclic specifications therefore analyzes groups of rules together to generate a global schedule that allows all of the data in a cycle to move together through the cycle.

We use the example in Fig. 6 to illustrate the operation of the algorithm for cyclic specifications. To simplify the presentation, we present the rules by themselves, omitting the module decomposition. We also omit the rule(s) that remove from queue `z` and any rules that do not affect the contents of queues `x` and `y`.

```
state x : queue(int) =  2 ;
state y : queue(int) =  3 ;
state z : queue(int);
0: t = head(x) ->
       y = insert(y,(t+3)&15), x = tail(x);
1: t = head(y) ->
       x = insert(x,(t+5)&15), y = tail(y);
2: t = head(x) and (t&3 = 0) ->
       z = insert(z,t), x = tail(x);
3: t = head(y) and (t&3 = 0) ->
       z = insert(z,t), y = tail(y);
// implementation constraints
length(x) = 1;
length(y) = 1;
```

Fig. 6. Cyclic Example

This example is modeled after a random number generation process that starts with two numbers (2 and 3) and repeatedly adds 3, then 5 to each number, retaining the lower 4 bits after each addition. The computation records the values of the numbers when their bottom 2 bits become 0. In our implementation, each number is stored in a queue, and the designer specifies that each queue has a single entry. Because of the cyclic nature of the specification, the numbers must move through the queues together — if they attempt to move separately, there is no room in the queues. The synthesis algorithm must therefore schedule the rules involved in the cycle (rules 0 and 1) together to coordinate their queue insertions and removals.

- **Idea:** The key idea is to find, for each rule that inserts an element into a queue q, the maximal sets of rules that have to execute together to preserve the "non-overflow" invariant of q, at the end of each clock cycle. To do this, the algorithm starts from each rule and traverses the rule graph on all possible paths, gathering for each rule that we go through, the conditions that would let that rule fire. We stop if either a rule is not an appending rule, so will always fire when its initial enabling condition becomes true, or if we already traversed that rule on the current path, so we already considered that the rule fires. Once we reach such a point there's no additional information on that path in the circuit that was not already collected at the first traversal. Nothing needs to be added to yield a correct solution. When all paths reach such points, the set of all rules that have to fire together becomes provably maximal.
- **Algorithm:** The scheduling algorithm processes each rule in the cyclic specification in turn. Fig. 7 shows the algorithm that produces the additional enabling condition for a rule R_i. $CrtPath$ keeps the currently explored path, for purposes of termination. This variable is initially empty for each symbolic execution of a rule. The symbolic execution of a rule terminates if either one of the two scenarios below is true:
- R_k is a non-appending rule and in this case $newR_k = R_k$. We call $newR_k$ what we derive from R_k after enhancing it with the additional constraints.
- R_k is a rule previously examined on the current path. This means we already assumed R_k fires on that path, so there's no need to explore further, therefore $newR_k = true$.

```
**SymbolicExecution** $(R_i, CrtPath)$
$Q = \{q$ | $R_i$ inserts into $q\}$
if $Q$ != nil
then for each $q \in Q$
    $I = \{R_j$ | $R_j$ inserts into $q\}$
    $D = \{R_k$ | $R_k$ removes from $q\}$
    $S =$ if ($R_i$ is the only rule in $I$) or
         $\forall i1, i2 \in I, (i1, i2)$ mutually exclusive
            then $D$
            else $D \cup I$
    for each rule $R_k \in S$
        $newCrtPath = CrtPath \cup R_k$
        $newR_k =$ if $R_k \in CrtPath$
            then $true$
            else SymbolicExecution($R_k$, $newCrtPath$)
        $newSelect(R_i, q) = Select(R_i, q)[newR_k / R'_k]$
        $Select(R_i, q) = newSelect(R_i, q)$
    $newR_i = (R_i$ and $Select(R_i, q))$
    $R_i = newR_i$
else $R_i$
```

Fig. 7. Computing the Additional Constraints for a Rule in a Cyclic Specification

Fig. 8. Comparative Clock Cycle and Area Estimates

Architecture	Cycle (MHz)	Area
RISC Pipelined Processor	88.89	23195.25
SCU RTL 98 DSP	90.91	22999.50

IV. Experimental Results

We have implemented a prototype synthesis system based on the algorithms presented in this paper. The algorithm generates synthesizable Verilog implementations at the RTL level. We wrote the specification of a 32-bit datapath, RISC-style, linearly pipelined processor with a complete instruction set[5], ran it through our synthesis algorithm, then synthesized the resulting Verilog model using the Synopsis Design Compiler to an industry standard .25 micron standard cell process. To serve as a reference point, we also synthesized, in the same environment, the Santa Clara University SCU RTL 98 DSP, a hand-written, standard 32-bit fixed point DSP that implements the same basic functionality. Table 8 shows area and clock cycle numbers for the two applications. Notice that the synthesized area is roughly the same, while clock-cycle-wise, our processor is within 3 percent of the hand coded version.

It took us less than five hours to develop the specification for the processor, which we believe is significantly faster than developing the DSP model by hand. Our specification contains 15 lines for state declarations and 21 lines of rule definitions for module specifications. The SCU RTL 98 DSP application, on the other hand, consists of approximately 885 lines of Verilog code. Our automatically generated implementation consists of about 1200 lines of synthesizable Verilog.

We have also tried our synthesis algorithm on several non-processor benchmarks. Table 9 shows cycle time and area numbers for a specification describing bubblesort for eight 8-bit numbers, a butterfly network similar to the ones used in bitonic sorting networks and in FFTs, and a cascaded FIR with 16 coefficients.

Fig. 9. Clock Cycle and Area Estimates for a Few Basic Data Processing Elements

Benchmark	Cycle (MHz)	Area
Bubblesort	107.06	5434
FFT	104.42	5411
FIR	105.01	3757

The running time of our system is roughly proportional to the complexity of the generated control. For all applications except the pipelined processor, our system required less than one minute to generate the Verilog output. For the processor, it took roughly half an hour. We tested the generated Verilog for each application, including the pipelined processor, using the Cadence NCVerilog simulator.

V. Related Work

HDLs like VHDL or Verilog use a model of concurrency in which processes communicate using signals. A signal is a direct physical connection with no buffering and with dynamic synchronization overhead. Designed for formal verification and synthesis of communication protocols, SUAVE [1] improves the communication features of VHDL by providing bounded or unbounded message buffers. The synchronous communication model is similar to those of CSP [13] and Occam [6]. Our approach is different in that it displays an asynchronous communication model at design level, while generating a synchronous implementation.

Another approach uses software languages such as C and C++. The Olympus/Hercules system is designed to support mainly ASIC synthesis from HardwareC [15], a C-like syntax behavioral language. HardwareC supports concurrency by providing synchronous queues with blocking send and receive constructs. In *Scenic* [9], the semantics of concurrency is similar to that of CSP and processes communicate via signals. In both approaches, the synchronous communication semantics force the designer think about the global timing when describing the system.

Systems based on hierarchical PBSs [18] (Production Based Specification) specify the control implicitly via the production hierarchy. The simplicity of PBS comes from the local nature of each production, allowing the designer not worry about the explicit construction of the global flow. PBS is closer to our description language in the sense that both describe external behavior rather than particular implementations of a system. Moreover, the actions for a given behavior are described locally, even if possibly simultaneous actions can be described elsewhere. On the other hand, the framework is synchronous.

Systems like Ptolemy [5], GRAPE [16], SPW from Cadence or COSSAP from Synopsys start from block diagram languages based on a dataflow semantics and are targeted to DSP design, mostly for minimizing memory usage and buffer memory. In SDF (Synchronous Data Flow), a static schedule for the block diagram is found that fires each *actor* in the

[5] The instruction set contains load, store, jump, ALU, multiply and variable shift operations.

dataflow graph at least once and does not change the net number of *tokens* queued on each edge. In our approach, not every update rule has to fire every clock cycle, the number of elements in the queues may vary in time and the desired lengths for the queues are specified by the designer. Unlike DDF (Dynamic Data Flow), which implements a run-time scheduler, our approach provides a statically scheduled model.

In synchronous languages like Esterel [4], Lustre [11], Signal [10] and Statecharts [12], the programmer thinks about a program as reacting instantaneously to external events. Processes are tightly coupled and deterministic, communication being realized by instantaneous broadcasting.

Classic work on pipelining optimization by Patel [17], Davidson, Shar and Thomas [8] starts from a given reservation table for the task flows in a system and develops methodologies for increasing the throughput of a pipeline. In our approach there is no initial knowledge of what gets assigned to each pipeline stage at each clock cycle; there is no notion of synchronicity.

Several specification and verification systems have taken an approach similar to ours, based on describing the behavior of a system by a state transition system [7, 13]. Closely related to our research, Hoe and Arvind [14] develop a method for hardware description and synthesis based on an operation-centric approach.

VI. Conclusions

This paper presents a new approach for hardware synthesis. The designer uses a design language based on connecting modules with asynchronous queues. The synthesis algorithm eliminates the inefficiency associated with a direct asynchronous implementation by automatically generating a coordinated global schedule for all operations in the system. This schedule is used to generate an efficient and fully pipelined synchronous implementation.

The primary advantages of this approach include good support for concurrency, modularity, debugging, and reuse in the design language. The use of update rules provides support for formal verification and concurrency, and enables concise, behavioral descriptions. This gives the resulting implementation a better chance to correctly reflect the designer's intent. The synthesis algorithm is the key to enabling the designer to use a convenient design language while obtaining an efficient hardware implementation of the design. The global scheduling and relaxation algorithms maximize the throughput. Relaxation also reduces the clock cycle time by parallelizing the evaluation of the enabling conditions of the rules. Global scheduling eliminates the need for handshaking hardware, while applying optimizations at a global level optimizes the combinational logic. Our experimental results provide encouraging evidence that the approach can deliver efficient implementations of high-level specifications. The approach also greatly improves on design time and has reasonable run-times of the synthesis algorithm. Our approach is well-suited to systems that are naturally described as a composition of interacting sub-systems. The class of pipelined circuits is one such system, as FIFO queues are a natural way to isolate pipe stages.

References

[1] P. Ashenden, R. Esser, and P. Wilsey. Communication and synchronization using bounded channels in SUAVE. In *Proceedings of the 1999 International Hardware Description Languages Conference and Exhibit (HDLCON99)*, 1999.

[2] F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[3] M. Ballantyne. Automatic deduction. Technical Report STAN-CS-82-937, Dept. of Computer Science, Stanford Univ., Stanford, Calif., October 1982.

[4] F. Boussinot and R. de Simone. The ESTEREL language. In *Proceedings of the IEEE*, pages 79(9):1293–1304, September 1991.

[5] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: a framework for simulating and prototyping heterogeneous systems. *Intl. J. of Computer Simulation*, 1995.

[6] A. Burns. *Programming in Occam 2*. Addison-Wesley, Reading, Mass., 1988.

[7] K. M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison-Wesley, Reading, Mass., 1988.

[8] E.S. Davidson, L.E. Shar, A.T. Thomas, and J.H. Patel. Effective control for pipelined computers. In *Proceedings of the 1975 Spring COMPCON.*

[9] A. Ghosh, J. Kunkel, and S. Liao. Hardware synthesis from C/C++. In *Design, Automation and Test in Europe Conference and Exhibition*, 1999.

[10] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with Signal. In *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, September 1991.

[11] N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous dataflow programming language Lustre. In *Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue*, September 1991.

[12] D. Harel. Statecharts: a visual approach to complex systems. In *Science of Computer Programming*, pages 8:231–274, 1987.

[13] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, N.J., 1985.

[14] J. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *VLSI: Systems on a chip*, Lisbon, Portugal, December 1999.

[15] D. Ku and G. De Micheli. HardwareC: a language for hardware design. Technical Report SCSL/CSL/TR-90-419, Computer Systems Laboratory, Stanford Univ., Stanford, Calif., August 1990.

[16] R. Lauwereins, P. Wauters, M. Ade, and J. A. Peperstraete. Geometric parallelism and cyclo-static data flow in grape-ii. In *Proc. IEEE Workshop on Rapid System Prototyping*, Grenoble, France, June 1994.

[17] Janak Patel. Pipelines with internal buffers. In *Proceedings of the Fifth Annual Symposium on Computer Architecture, ISCA78.*

[18] Andrew Seawright and Forrest Brewer. Synthesis from Production-Based Specifications. In *Proceedings of 29th Design Automation Conference*, 1992.