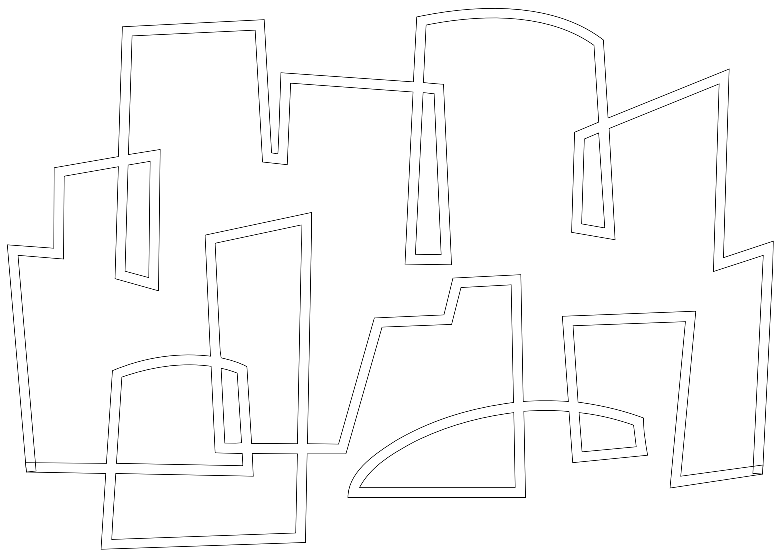
High-level synthesis: An Essential Ingredient for Designing Complex ASICs

Computation Structures Group Memo 473
April 24, 2004

Arvind (MIT CSAIL)
Rishiyur S. Nikhil (Bluespec, Inc.)
Daniel L. Rosenband (MIT CSAIL)
Nirav Dave (MIT CSAIL)

Submitted for publication

This report describes research done at the Computer Science and Artificial Intelligence Laboratory at the Massachusetts Institute of Technology. Funding for this is provided in part by the Defence Advanced Research Projects Agency under contract DARPA NBCH3039004.



High-level synthesis: An Essential Ingredient for Designing Complex ASICs

Abstract

It is common wisdom that synthesizing hardware from higher-level descriptions than Verilog will incur a performance penalty. We present a case study that shows that this need not be the case. If the higher-level language has suitable semantics, it is possible to synthesize hardware that is competitive with hand-written Verilog RTL. Differences in the hardware quality are dominated by architecture differences, and so it is more important to be able to explore multiple hardware architectures easily, which is enabled by using a higher-level language.

1 Introduction

Five to ten million-gate ASICs are commonplace today. Their design typically takes 18 to 24 months and costs somewhere between \$10M to \$20M. An ASIC has a selling window of 6 to 8 months in the market place, and consequently, if the chip is delayed by much more than six months the customer is likely to leapfrog to the next generation chip, which is likely to be cheaper, faster, or have more features. Currently, in spite of a myriad of verification tools, three verification engineers are needed for each designer in a typical ASIC team. The verification task is exacerbated rather than abated by use of pre-existing IP blocks. Only a small fraction of ASICs complete development in time to make money. Consequently, ASIC development has come to be viewed as an expensive and highly risky proposition.

Another casualty of the increasingly compressed development timeline is a thorough exploration of architectural alternatives. An alternative microarchitecture can often result in far greater time and area savings than any tweaking of a specific architecture. Consider adding a pipeline stage or functional unit, multiplexing an expensive resource, or doubling the datapath width while halving the clock rate. Determining the impact of these alternatives would require such a massive redesign as to be impractical using current methodologies. Although adding or removing datapaths and memories is relatively straightforward, the subsequent redesign and re-verification of the control logic is not.

Commercial developments in CMOS technology make it likely that 50 million gate ASICs will be feasible by 2010. Such large ASICs will be common place only if EDA tools can keep up with the growing size and complexity of designs. What is needed is a high-level design methodology and accompanying tools that will allow complex digital systems to be realized by reasonably sized teams in a short time frame. The central themes of any such methodology have to be *correctness by construction*, *predictable functionality* and *predictable performance*. The methodology should make it as easy to use pre-existing IP blocks as it is to use procedural and data abstraction libraries in software, and should provide a framework that will simplify exploration of a large architectural design space by automatically generating correct control logic for any composition of instantiated library elements. To be successful, the quality of hardware synthesis from these high-level descriptions must approach that of hand-designed blocks so that designers are not tempted to break the abstractions.

This paper partially evaluates the Bluespec hardware design methodology that purports to have most of the characteristics described above. The methodology is based on synthesis from high-level hardware descriptions expressed as guarded atomic actions [10]. Guarded atomic actions form the basis of Bluespec [2], which has been developed over the last four years, first at Sandburst Corporation and, now at Bluespec Inc.

Our method of evaluation is to take a small but non-trivial design problem and explore many different micro-architectures to implement it. We compare these micro-architectures in terms of area, clock-cycle time, efficiency in solving the problem, robustness to changes in component characteristics and flexibility in dealing with changes in problem specification. We also compare some of the Bluespec generated results against hand-coded verilog. The problem we have chosen is the much studied "Longest Prefix Match" search engines which are present in all Internet routers. A solution must pass the same test suite on the same test bench to be acceptable.

Based on our study we conclude that 1. The differences in area and timing between different micro-architectural solutions are far greater than the differences in hand-written Verilog and Bluespec generated Verilog; 2. If both Bluespec and Verilog are written by the same designer, the Bluespec compiler routinely generates code that is comparable to hand-written Verilog; 3. If the Verilog design is cleverly optimized, the Bluespec designer can usually but not always imitate the Verilog designer to produce comparable results; and 4. Architectural exploration is easier and quicker in Bluespec than in Verilog because the Bluespec methodology preserves correctness at every step and encourages the use of modules. 5. Though it is hard to quantify, Bluespec designs often take much less time to develop the first working model than comparable Verilog designs. But this advantage can be dissipated by the extra time needed for "performance tuning".

Paper Organization In Section 2, we provide a background on guarded atomic actions as an HDL and briefly explain automatic synthesis from them. Sections 3, 4, and 5 discuss the "Longest Prefix Match" problem, alternative design solutions, and how they are coded in Bluespec. Section 6 presents results for designs using this methodology. Related work in high-level hardware design is discussed in Section 7. Section 8 provides our conclusions and some ideas for further evaluation.

2 Guarded atomic actions as HDL

2.1 Guarded atomic actions and modules

In the Bluespec methodology, the designer first instantiates the state elements of the system (registers, FIFOs, memories, etc.) explicitly. That is, in Bluespec there is no mysterious "inferencing" of state from the program. Every bit of state (even a register) is a *module instance*, and clients of a module interact with it using *interface methods*. A call to an interface method looks like a procedure call, but every method also involves an *implicit condition*: a "ready" wire that specifies if the module can currently perform the requested method's action. For registers, the methods are the usual read() and write() operations, and the implicit conditions are always True (and will be optimized away). For FIFOs, the methods are the usual enq() and deq() operations. The implicit condition for enq() is True if the FIFO is not full; the implicit condition for deq() is True if the FIFO is not empty. A FIFO module may be visualized as shown in Figure 1.

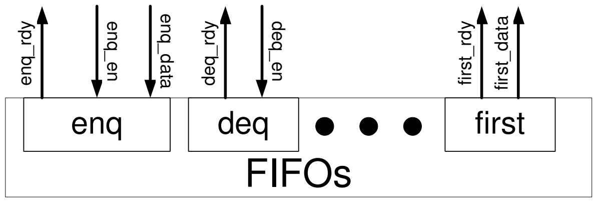


Figure 1: A FIFO Module.

Next, the designer describes the behavior of the system using a collection of *guarded atomic actions or rules*, which express the conceptual operations on the state of the system. Each rule specifies the condition under which it is enabled, and a consequent *allowable* (i.e., not compulsory) state transition. Two rules may access and update common state, but rules are written without regard to such interaction. In particular, rules have *atomic* semantics, i.e., the effect of each rule can be expressed and reasoned about in isolation, as if the rest of the system was frozen (see, for example, Arvind and Shen[1]). A precise and useful semantics emerges from the fact that any legitimate behavior of the system can be understood as a series of atomic actions on the state. Indeed, this is key to the high-level nature of rules: all the control circuitry and muxing needed to manage potential interactions between rules is produced by automatic synthesis as discussed in the next section.

2.2 Synthesis from guarded atomic actions

We briefly outline the synthesis approach of Hoe and Arvind [9, 10]. A rule consists of a guard and a body and may be written in the following form:

$$\text{Rule } R_i : \text{when } \pi(s) \text{ ==> } s := \delta(s)$$

where π is the guard (predicate) and $s := \delta(s)$ is the body of rule R_i. Function δ is used to compute the next state of the system from the current state s. The execution model for a set of rules is to non-deterministically pick a rule whose predicate is true and then to atomically execute that rule's body. The execution continues as long as some predicate is true:

```
while (some $\pi$ is true) do
1) select any $R_i$, s.t. $\pi(s)$ is true
2) $s := \delta(s)$
```

There is a straightforward translation from rules into hardware as shown in Figure 2. Assuming all state is accessible (no port contention), each π and δ can be implemented easily as combinational logic. A hardware scheduler and control circuit then needs to be added so that in every cycle the scheduler dynamically picks one δ function whose corresponding π condition is satisfied and the control circuit updates the state of the system with the result of the selected δ function. The cycle time in such a synthesis is determined by the slowest π and the slowest δ functions. Although correct, such an implementation has unsatisfactory performance because it is often possible to execute several rules simultaneously such that the result of the execution matches an execution in which the selected rules are applied in some sequential order. Thus, the challenge in generating efficient hardware from sets of atomic actions is to generate a scheduler which in every cycle picks a maximal set of rules that can be executed simultaneously.

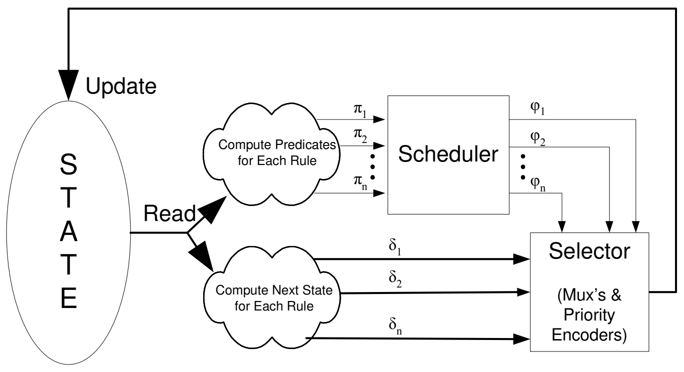


Figure 2: Synthesis from Guarded Atomic Actions.

It is easy to see that two rules can execute simultaneously if they are "conflict free", that is, they do not update the same state and neither updates the state accessed (i.e., "read") by the other rule. Arvind and Hoe further observed that two rules (R1 and R2) can execute simultaneously if one rule (R2) does not read any of the state that the other rule (R1) writes. In this case, simultaneous execution of R1 and R2 appears the same as sequential execution of R1 followed by R2. For this to hold, R2 writes must take precedence over writes to the same state by R1 and the execution of R1 must not disable R2. Such rules are called "sequentially composable" in [9, 10]. Hoe showed that from these pair-wise relationships between rules one can deduce if a group of rules can be scheduled concurrently. Figure 2 shows the circuit that is generated in Hoe's synthesis flow. The predicates (π's) are computed for each rule using a combinational circuit. The scheduler is designed to select a maximal subset of applicable rules with the constraint that the outcome of a scheduling step can be explained as atomic firing of rules in some sequence. Based on which rules the scheduler chooses to enable (ϕ's), the selector block then combines the update functions (δ's) from the chosen rules and updates the current state with the resulting values.

An aggressive "mutual exclusion" analysis of rules is performed to eliminate scheduling cases that cannot arise logically. Without such an analysis one may unnecessarily commit resources, such as ports. One also needs a policy for selecting among the maximal schedules because different maximal sets can have different resource requirements. Construction of good schedulers is the most important problem in the synthesis of atomic actions. Recent work has made it possible to provide much better control over scheduling via modular composition and scheduling annotations (see, for example, Rosenband and Arvind[15], Nordin and Hoe[12] and Rosenband[14]).

3 A Design Problem: Longest Prefix Match Function

The Longest Prefix Match (LPM) function is used in Internet Protocol (IP) packet routers to determine the output port to which an input packet should be forwarded based on the destination IP address (IPA) in the packet's header. For IPv4 packets the IPA is 32 bits while for IPv6 it is 128 bits. We will consider solutions for IPv4 packets with an eye towards generalization to IPv6 packets.

LPM is based on a routing table which conceptually consists of an ordered set of prefixes, each associated with an output port. Each prefix is a string of length $\leq$ 32 bits. High-end routers can contain more than 100K prefixes. Since more than one prefix can match an incoming IPA, the port associated with the longest matching prefix is selected as the output port. Thus, in a properly ordered table, the first prefix match on a sequential search produces the correct output port. It is a requirement that the router maintain the ordering of packets between the same source and destination.

Packets must be processed at line rate (10 Gb/s today). This leaves a budget of typically $<$ 10 memory references per packet. Memory size must be kept small for cost, board area, access rate and power reasons. A flat table would contain 2^{32} entries and is not feasible; sparse data structures are necessary. Finally, routes may change while the system is online.

Although many clever data structures have been devised for LPM function, we use a simplified but representative example to illustrate our design approach. The lookup table is organized into three levels as shown in Figure 3. The first 16 bits of the IPA selects an entry from a root table containing 64K entries. If we find a leaf (output port number), we're done. Otherwise, we find a pointer to a 2nd-level table of 256 entries, which is indexed with the next 8 bits of the IPA. Again, we either find a leaf or a pointer to a 3rd-level table of 256 leaves indexed by the remaining 8 bits of the IPA. Each packet thus requires from 1 to 3 memory accesses. The data structure can be computed offline from any given routing table containing prefixes with lengths $\leq$ 32 bits.

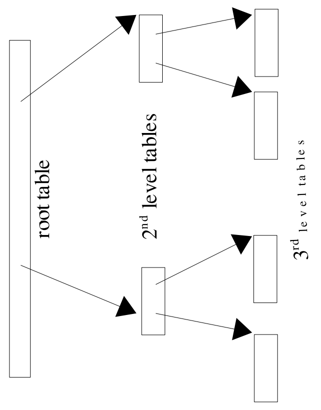


Figure 3: Data structure for Longest Prefix Match.

Figure 4 shows a software implementation of LPM, written in a variant of C, extended with a Verilog-like bit extraction facility. Automatic generation of hardware from such a specification is close to impossible if the hardware is required to sustain 10 Gbps line rate.

```
int lpm(IPA  ipa)
{
    int p;
    p = RAM [rootTableBase + ipa[31:16]];
    if (isLeaf(p)) return p;
    p = RAM [p + ipa [15:8]];
    if (isLeaf(p)) return p;
    p = RAM [p + ipa [7:0]];
    return p; // must be a leaf
}
```

Figure 4: Software version of LPM algorithm.

4 Some architectural alternatives

We now describe three radically different architectural alternatives for a hardware implementation of LPM, and discuss their attributes, strengths and weaknesses. A key point is that the entire data structure must be kept in a single memory because of pin limitations and memory management flexibility. Assume that the memory is pipelined, and has a fixed multi-cycle latency L. Thus in a pipelined implementation, the memory port is a shared resource across different stages of the pipeline, and at any given time the memory pipe will contain requests interleaved from different packets.

Statically scheduled pipeline The first design, whose schema is shown in Figure 5 and Figure 6, is a "rigid" pipeline architecture. For the first L cycles, we launch the first memory requests from each of the first L IPAs (assuming packets are available). Since each packet makes three memory access, no new packet is injected for the next $2L$ cycles. This guarantees that when the first memory response arrives, we can launch the second memory request for the first IPA (if it needs a second access), and so on. In summary, we completely statically schedule the pipeline, knowing the memory latency L and the maximum number of memory requests (three) for each IPA.

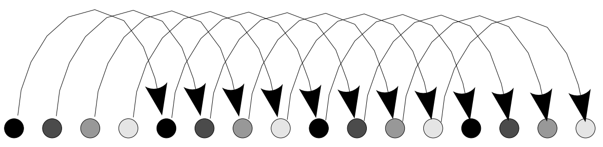


Figure 5: Statically scheduled memory references

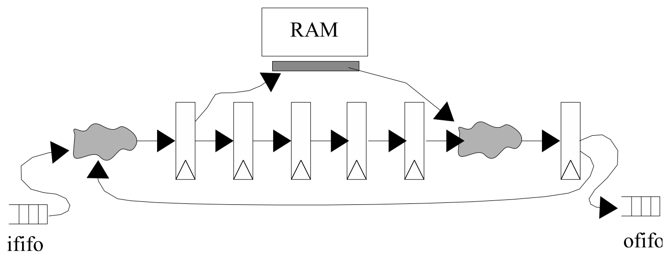


Figure 6: Rigid pipeline architecture

There are several issues with this design. The memory is fully utilized only if packets arrive at the highest rate (minimum-sized packets at line rate) and if they all need three memory references. Thus, the memory bandwidth is sized for the worst case, instead of the expected case. The latency and throughput in processing packets is fixed at the worst case (the length of the pipe) even if the actual workload contains packets requiring fewer memory references. The whole pipeline must be re-planned if we are given a memory with a different latency. Finally, additional complex control is required to insert memory accesses for routing software to update the data structure online.
As we shall see in Section 6.2, even for statically scheduled pipeline there are several alternatives for organizing the state elements with very different implications for area and timing.

Flexible pipeline Figure 7 shows the second design, which has a "flexible" pipeline architecture. As each IPA arrives on `ififo`, `Stage0` launches its first memory request into `mport0` and keeps the IPA in `fifo0`. `Stage1` collects this IPA and the corresponding response from `mport0`. If done, it places the result and a "done" bit into `fifo1`, else it launches the second memory request into `mport1` and places the IPA and a "not done" bit into `fifo1`. `Stage2` and Stage3 act in a similar manner to Stage 1 and the result finally ends up in `ofifo`. Notice, all fifo's except `ififo` must be of size L for full memory utilization.

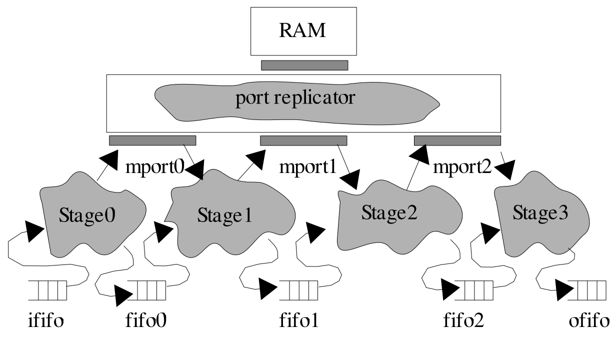


Figure 7: Flexible pipeline architecture.

The single memory is accessed through a "port replicator" module that takes requests as they arrive in any order on `mport0`, `mport1` and `mport2` and forwards them to the memory. Results from the memory are distributed back to `mport0`, `mport1` and `mport2`. Since the order of request arrivals is unpredictable, book-keeping circuitry (e.g., tagging) is required to return memory responses to the correct ports.
Although this design possibly requires more hardware and control logic (FIFOs, a port replicator with tagging, etc.), in many ways it is more robust than the previous design. For example, it is robust to changes in memory latency. It is relatively straightforward to extend it to a fourth port for updating the routing data structure online. Packets that require fewer memory accesses can traverse the pipeline faster, and so the design can exhibit a better latency and throughput than the worst case.

Circular pipeline Finally, Figure 8 shows a circular pipeline architecture, which in a way is a folded version of the flexible pipeline. The `Input` stage takes an arriving

IPA from `ififo` and a "ticket" from the Completion Buffer and launches a memory request using the high 16 bits of the IPA and places a (ticket,IPA[15:0],State$_0$) tuple into `cfifo`. Based on the memory response p and the first tuple (ticket,IPA,State$_j$) in `cfifo`, either the `Completion` or `Circulate` stage executes. If the lookup is done the `Completion` stage forwards (ticket,p) to the Completion Buffer else the `Circulate` stage places (ticket,IPA,State$_j+1$) into `cfifo` and launches another memory request.
Thus, each IPA goes around the pipe as many times as the number of memory references it needs, and the result finally goes to the Completion Buffer. Since IPAs need varying numbers of memory references, results may arrive at the Completion Buffer out of order; the tickets allow it to output them into `ofifo` in the right order. In the worst case, for 100% memory utilization, the size of the Completion Buffer must be $2L$, though in practice a smaller buffer may do. The number of IPA's in the circular pipeline must not exceed the capacity of `cfifo` to avoid deadlocks and thus, a new IPA should have a lower priority to enter `cfifo` than an IPA already in the pipeline. The size `cfifo` must be L for 100% memory utilization.

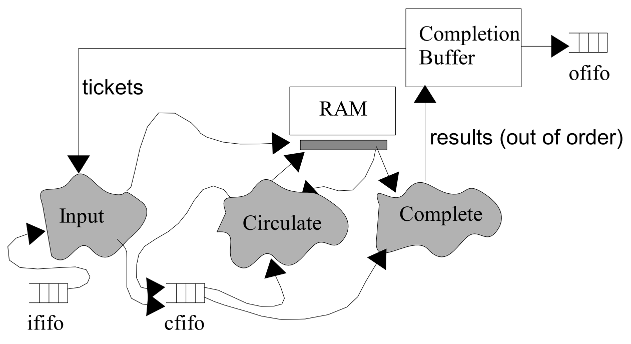


Figure 8: Circular pipeline architecture.

This last design arguably has the simplest memory architecture (single port, no interleaving issues), but the most complex control (Completion Buffer with tickets and reordering). On the other hand it is very robust to different memory latencies and it generalizes easily to LPM on IPv6 (128-bit) addresses which require more memory references.

5 Coding in Bluespec: Correctness by Construction

We now give examples of how pieces of the above designs are coded in Bluespec. These examples illustrate how designing with guarded atomic actions frees designers from worrying about global coordination allowing them to focus on the much simpler task of local correctness.
The `Input` stage in the Circular Pipeline is expressed as follows:

```
Input:
  when (True) {
    ipa = ififo.pop();
    tkt = compBuf.getTicket();
    cfifo.enq {tkt, ipa[15:0], State0};
    RAM.request(base_addr + ipa[31:16]);
  }
```

Although the explicit condition in the rule is `True`, the rule is not enabled until all the implicit conditions are also `True`, i.e., until `ififo` contains an IP address, the completion buffer is willing to yield a ticket, and `cfifo` has room (is not full). In one succinct rule we have expressed a conceptual operation controlled by a complex set of conditions.
The completion and re-circulate rules are expressed as follows:

```
p = RAM.getResult();

Complete:
  when (isLeaf(p)) {
    {tkt,ipa,s} = cfifo.pop();
    compBuf.done {tkt, p};
  }


Circulate:
  when (!isLeaf(p)) {
    {tkt,ipa,s} = cfifo.pop();
    cfifo.enq {tkt, ipa, s+1};
    RAM.request(compute_addr(p,s,ipa));
  }
```

Both the `Input` and the `Circulate` rules enqueue into `cfifo`. To avoid deadlocks, the `Circulate` rule has to be given priority, or some other mechanism (such as an up-down counter) is needed to ensure that no more than L requests are enqueued into the circular pipeline. Given the priority between the rules, the Bluespec compiler automatically synthesizes the appropriate control logic. Similarly, control circuitry to manage the concurrent access to the shared completion buffer by the `Input` and `Complete` rules is automatically generated.
Atomicity of the actions in a rule plays a crucial role in avoiding races between enqueuing and dequeuing of `cfifo` and sending a request to the memory. The designer can reason about each rule in isolation to ensure that it is doing the right thing, without worrying about interactions with other rules. The synthesis method is guaranteed to preserve these atomic semantics while producing highly concurrent clocked synchronous hardware.
As another example, we consider the port replicator for shared access to a RAM from the Flexible Pipeline architecture of Figure 7. Figure 9 shows the organization of a 2-way port replicator. When a request arrives on `port0` and `port1`, respectively, the `In0` or `In1` stages forward it to the shared port and place the tag "0" or tag "1" into the FIFO. When a response arrives from the shared port and "0" is at the head of the FIFO, the `Out0` stage forwards the result to `port0`. If "1" is at the head of the FIFO, the `Out1` stage forwards the result to `port1`.

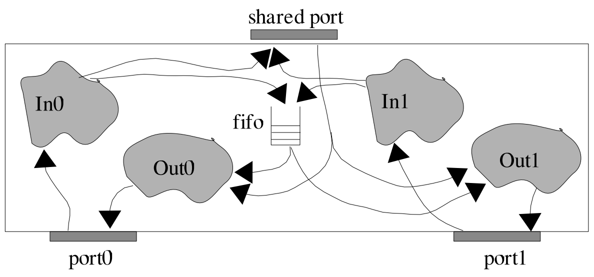


Figure 9: 2-way port replicator.

The behavior of `In0` and `Out0` are expressed as follows (`In1` and `Out1` have similar rules):

```
In0:
  when (True) {
    req0 = port0.deq();
    sharedPort.enq (req0);
    fifo.enq (Tag0);
  }

Out0:
  when (fifo.first() == Tag0) {
    resp0 = sharedPort.deq();
    fifo.deq();
    port1.enq (resp0);
  }
```

Note that `In0` and `In1` can conflict: if requests arrive simultaneously on ports 0 and 1, both attempt to forward their requests into the shared port, and both attempt to enqueue a tag into the FIFO. Similarly, `Out0` and `Out1` both examine the shared port response and the head of the FIFO, and one of them dequeues from the FIFO. All the control circuitry for managing this interaction is synthesized automatically. How does the designer assure himself that the identity of memory requests is accurately reflected in the tag FIFO, i.e., that `In0` and `In1` don't send requests to the memory in one order and enqueue their tags in the opposite order? Once again, atomicity comes to the rescue. It ensures that the order in the FIFO is exactly the same as the order of memory requests. Assuming the memory itself maintains FIFO order, the desired property is obtained.

Now suppose we wish to generalize the port replicator to have the following features:

- N-way replication (not just 2 or 3)
- Requests of arbitrary type T1
- Responses of arbitrary type T2
- Parameterized by memory latency L

Bluespec solves this by allowing powerful composition of circuit elements which include Actions, Rules, Interfaces and Modules. Bluespec permits arbitrary programming with these objects, and thus, uses software expressivity only to describe circuit *structure*; behavior is specified entirely by Atomic Rules. In this approach, software expressivity does not face any ad hoc limits such as "synthesizable subsets." This approach is in sharp contrast with Behavirol Verilog and other C-based high-level synthesis approaches which use software expressivity to describe circuit *behavior*.

6 Experimental Results

All the Bluespec and Verilog codes for various designs were written by the authors. All designs were simulated using a shared test-bench. The memory to which the designs interface is fully pipelined and has a latency of 4 cycles. Requests are inserted into the LPM design whenever possible, results are dequeued whenever possible. A simple compiler was used to translate prefix tables into appropriate data structures. The test data consisted of 9920 requests with an average of 1.908 dependent memory references per request.

Bluespec designs were compiled using the Bluespec Compiler version 3.8.12, available from Bluespec Inc. The generated verilog was compiled to the TSMC 0.18μm library using Synopsys Design Compiler version 2003.12. So as to achieve accurate timing and area results, the timing constraints were tuned to be within 500ps of the timing that the design could achieve. The worst case (slow process, low voltage) timing model was used. We divide area results by the area of a two input NAND2X1 gate (9.98μm^2) to obtain the reported gate counts.

6.1 Bluespec versus hand written Verilog comparison

The table in Figure 10 shows the best Bluespec and Verilog synthesis results for each of the three previously described longest prefix match architectures. The designs are nearly identical in both area and cycle times. The number of register bits varies slightly between the Verilog and Bluespec implementations because of small design choice differences and because the Bluespec compiler generates slightly different data and state machine encodings. The total gate count (combinational logic and registers) is within 8% in all designs, cycle time is within 7%, and as expected, simulation results between the Bluespec and Verilog designs match exactly.

The differences we observe are within the noise margin of what we obtain from repeated compilation with Synopsys design-compiler with slightly different timing constraints. In general, the Bluespec results indicate slightly faster designs and slightly larger area. This can be explained through the generation of lower level code by the Bluespec compiler which in some cases makes the designs slightly faster and, because of differing logic structuring, marginally larger. In comparison to the Verilog vs. Bluespec tradeoffs, area, cycle time and execution performance vary far more between designs with differing high-level and micro-level architectural choices.

Comparing the architectures we find that the smallest design, the static pipeline, is one seventh the size of the largest

	Language	Timing Constraint	Reg Bits	Comb Gates	% diff	Total Gates	% diff	Cycle Time (ns)	% diff	Through-put (cycles/ lookup)	Avg Latency (cycles)	Memory Utilization
Static	BS	3.3ns	252	837	2%	2391	5%	3.32	-7%	3.001	17.0	63.5%
Static	Verilog	3.3ns	240	818		2271		3.56		3.001	17.0	63.5%
Flexible	BS	4.7ns	1219	6190	9%	15910	8%	4.7	0%	1.908	20.972	99.9%
Flexible	Verilog	4.7ns	1144	5685		14759		4.7		1.908	20.972	99.9%
Circular	BS	3.6ns	778	2391	3%	8170	1%	3.67	2%	1.908	14.814	99.9%
Circular	Verilog	3.6ns	778	2331		8103		3.62		1.908	14.814	99.9%

Figure 10: Results table 1

	Comments	Reg Bits	Comb Gates	% diff from best	Total Gates	% diff from best	Cycle Time (ns)	% diff from best
Static	BS; Initial; 3.3ns	252	1719	105%	3375	41%	3.69	11%
Static	BS; Data alignment; 3.3ns	252	1038	24%	2606	9%	3.48	5%
Static	BS; No type conversion; 3.3ns	252	948	13%	2478	4%	3.61	9%
Static	BS; Nest case (BEST); 3.3ns	252	837	0%	2391	0%	3.32	0%
Static	Verilog; Replicated; 3.3ns	243	7151	775%	8898	292%	3.60	1%
Static	Verilog; BEST; 3.3ns	240	818	0%	2271	0%	3.56	0%

Figure 11: Results table 2

design, the flexible pipeline. Also, as expected, both the flexible and circular pipeline execute more efficiently than the static pipeline. In our test case 99.9% of the available memory bandwidth is utilized by the flexible and circular pipeline whereas the static architecture achieves only 63.5% memory utilization. These results are not entirely surprising, but we were impressed with how much smaller the circular pipeline is than the flexible pipeline. Our initial expectation was that the flexible and circular pipeline would be roughly equal in size. Because an optimized circulating loop and more efficient register allocation could be used in the circular design, using the Bluespec language we were quickly able to determine that the circular pipeline is the superior design compared to the flexible pipeline. Through simulation the designer can then choose whether the static pipeline provides sufficient performance, or whether an area penalty should be incurred and the circular design be chosen.

6.2 Tweaking the Bluespec code

We also implemented two Verilog implementations of the static pipeline: one replicated much of the computation that occurs in the pipeline by unfolding the feedback of Fig. 6 into a linear pipe, the other (BEST) was highly optimized. Figure 11 shows, even in Verilog it is easy for two reasonable implementations to have dramatically different results. In this example the replicated implementation uses almost nine times the combinational logic! The replicated design could easily have been the implementation of choice by a non-expert designer, and encourages the notion that micro-architecture is far more important than language results.

The initial Bluespec implementation of the static pipeline had over two times the combinational logic than the optimized verilog implementation. Although better than the replicated design, a factor of two is usually an unacceptable penalty to pay knowingly for the use of a higher-level language. They illustrate that in some cases the abstractions that the language provides can introduce an overhead, but that this overhead can be overcome by carefully crafting of the code. Our steps: 1) By carefully laying out data types several muxes and adders could be removed. 2) By simplifying types we can eliminate type-conversion logic. 3) A case statement was restructured to clarify that it was exhaustive. The circular pipeline took similar optimizations to achieve comparable performance to the verilog implementation.

7 Related work

7.1 Designing using atomic actions

The approach advocated in this paper was first used for verification by Arvind and Shen [1]. Synthesis was later explored by Hoe and Arvind [9, 10]. Augustsson developed the Bluespec language [2] which introduced the notion of modules and a two-level language. In addition to internal work at Sandburst and Bluespec, this approach has been used by at James Hoe and his student, Roland Wunderlich, at the Carnegie Mellon University. In cooperation with Intel, they have developed a version of the Itanium microachitecture running at 100 MHz on a 6M-gate FPGA. The FPGA board is housed in a dual processor PC chassis and can exchange data with the other processor over the system bus at the rate of 800 MBytes/s. In another effort, Nirav Dave has used Bluespec in the design and synthesis of reorder buffer [6].

Though the use of guarded atomic actions in verification has been much explored [4, 11, 5, 16, 3], there are only a few attempts at synthesis [13].

7.2 Modelling using C-based HDLs

There are several approaches to synthesis based on sequential and parallel C, e.g., [17], [7, 8]. These approaches have rarely approached the quality of synthesis that one gets from Verilog. There are myriad reasons for this gap. As one example, consider the difficulty of synthesizing control from parallel C.

Suppose we code an architecture such as the Circular pipeline of Figure 8 in some dialect of parallel C, i.e., C extended with a notion of processes, together with some constructs for process synchronization such as semaphores, events and channels. (Such dialects include SystemC). Each of the stages – input, circulation, and completion – becomes a separate sequential *process*. But still two major sources of complexity remain.

First, there is the issue of managing concurrent access to shared resources, such as the enqueues into `cfifo` by the `Input` and `Circulate` rules. When writing just for simulation this is not a problem – a simple lock will do the job. But when writing for synthesis, data paths must be properly multiplexed and controlled.

Second, there is the issue of complex control. Both the input and circulate/completion stages interact through `cfifo` and through the Completion Buffer. The input stage is only active if `ififo` is not empty, `cfifo` is not full, and the Completion Buffer has a ticket to issue. The circulate/completion stage is only active if a result from the RAM is available, if `cfifo` is not empty, and, depending on the RAM result, if either the Completion Buffer is ready to accept a result or if `cfifo` is not full. It is easy to make synchronization errors when using primitives like semaphores, events and channels. Furthermore it is not clear if such complex synchronization code in C can be synthesized automatically. These synchronization issues were handled automatically in our approach based on guarded atomic actions.

In summary, while we believe that the parallel C can be a fine medium to express behavior for simulation and perhaps, for hardware-software co-verification, it is not a good vehicle for expressing high-quality *synthesizable* hardware designs.

8 Conclusions

This study shows that high-level synthesis from guarded atomic actions as embodied in Bluespec provides a useful tool for microarchitectural exploration in the design of complex ASICs. The differences in area and time between different micro-architectures dominate differences between Bluespec-generated Verilog and hand-written Verilog.

Bluespec also provides a way of capturing the idioms commonly used in hardware design in a form that allows pervasive reuse. Existing hardware description languages only allow reuse at the level of fixed RTL modules with interfaces defined by sets of wires and cycle-level timing. Bluespec supports factoring of concepts such as buffered pipelines, completion buffers, and arbiters, into standard libraries. A designer can then instantiate these concepts with application-specific data types and connect them arbitrarily. The compiler will then synthesize an optimized design, including automatic generation of control logic. This approach raises the level of abstraction in hardware design without sacrificing hardware efficiency and may turn out to be the most crucial ingredient in designing large and complex ASICs in the future.

References

[1] Arvind and X. Shen. Using term rewriting systems to design and verify processors. *IEEE Micro*, 19(3):36–46, 1998.

[2] L. Augustsson, J. Schwarz, and R. S. Nikhil. Bluespec Language definition, 2001. Sandburst Corp.

[3] R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proc. 2nd. Ann. ACM Symp. Principles of Distributed Computing*, pages 131–142, 1983.

[4] K. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison Wesley, 1988.

[5] D. L. Dill. The Murphi verification system. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 390–393, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[6] N. Dave. Designing a Reorder Buffer in Bluespec. In *Proc. MEMOCODE'04*, June 2004.

[7] D. Gajski. *High-level synthesis: introduction to chip and system design.* Kluwer Academic, Boston, 1992.

[8] D. Gajski. *SpecC : specification language and methodology.* Kluwer Academic, Boston, 2000.

[9] J. C. Hoe. *Operation-Centric Hardware Description and Synthesis.* PhD thesis, MIT, June 2000.

[10] J. C. Hoe and Arvind. Synthesis of operation-centric hardware descriptions. In *IEEE/ACM Intl. Conf. on Computer Aided Design (ICCAD)*, pages 511–518, 2000.

[11] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley Professional (Pearson Education), 2002.

[12] G. Nordin and J. C. Hoe. Synchronous Extensions to Operation-Centric Hardware Description Languages. In *Proc. MEMOCODE'04*, June 2004.

[13] J. Plosila and K. Sere. Action systems in pipelined processor design. In *Proc. 3rd. Intl. Symp. on Adv. Res. in Asynchronous Circuits and Systems*, pages 156–166, 1997.

[14] D. L. Rosenband. The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs. In *Proc. MEMOCODE'04*, June 2004.

[15] D. L. Rosenband and A. 2003. Modular Scheduling of Guarded Atomic Actions. In *Proc. 41st DAC*, June 2004.

[16] J. Staunstrup and M. Greenstreet. From High-Level Descriptions to VLSI Circuits. *BIT*, 28(3):620–638, 1988.

[17] Synopsys. Behavioral Compiler/Behavioral Synthesis. See: www.synopsys.com/products/beh_syn/beh_syn.html.

Acknowledgements

Daniel Rosenband was supported under a grant from Compaq Computer Corporation.

Nirav Dave was supported under a grant from the Defence Advanced Research Projects Agency (contract DARPA NBCH3039004).

We would also like to thank Ravi Nanavati and Jacob Schwartz for their assistance in Bluespec coding; Jacob Schwartz for his assistence with LaTeX; and Winnie Cheng for her help in coding an earlier version of the circular pipeline in Verilog.