

A Design Flow Based on Modular Refinement

Nirav Dave, Man Cheuk Ng, Michael Pellauer, & Arvind
Computer Science and Artificial Intelligence Lab
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139
Email: {ndave, mcn02, pellauer, arvind}@mit.edu

Abstract—We propose a practical methodology based on modular refinement to design complex systems. The methodology relies on modules with latency-insensitive interfaces so that the refinements can change the timing contract of a module without affecting the overall functional correctness of the system. Such refinements can exacerbate the unit testing problem for modules whose specifications admit a set of output behaviors for the same input (*non-determinism*), or modules whose input behavior may be affected by past outputs (*feedback*). We avoid the difficult problem of generating appropriate unit tests for such modules by using system-level tests as unit tests to verify the correctness of refined modules. We illustrate our methodology by showing how one might develop a microprocessor with an in-order pipeline. We then develop a superscalar pipeline using the in-order pipeline as the starting point. Our methodology leverages the effort of design exploration to reduce the effort of specifying interface contracts and unit testing.

1. Introduction

Suppose a complex hardware system S is composed of modules (A_0, B_0, C_0, \dots) that interact with each other. By the modular refinement of S , we mean that we can change a module A_0 to A_1 to get a new system $S' = (A_1, B_0, C_0, \dots)$ that is still functionally correct but is better by some metric such as area, power, performance, etc. Modular refinement has many obvious benefits:

- 1) *Modular refinement lets us replace a module in a working system without a deep understanding of the rest of the implementation.* A module may need to be refined because it does not meet the area or timing requirements or because we want to experiment with a different algorithm to implement that module. While we need to check the correctness of the refined module (unit testing), modular replacement does not exacerbate the verification problem by requiring the retesting of other modules.
- 2) *A system that can be refined modularly allows independent teams to work on different parts of the design in parallel.*
- 3) *Modular replacement facilitates architectural exploration.* Many variants of an architecture can be produced via mix-and-match selection of module implementations.
- 4) *Modular refinement helps with the reuse of preexisting modules.* Since it encourages interfaces and designs that work in many contexts, it is likely that we can use modules in different designs without modification.
- 5) *Modular refinement is helpful in “performance debugging”.* By allowing the designer to isolate the effects of each module, we get a better understanding of where important tradeoffs or bottlenecks exist.

Modular refinement requires a method to verify the correctness of a module with respect to either a specification or a

prior implementation of the module. Consequently, modular refinement can be done only with respect to those properties of the design that can be stated formally or informally. For example, if the specifications require the two implementations to have equivalent Finite State Machines (FSMs) then there is no room for refinements that change the timing behavior of a module. Modular refinements that allow timing behavior changes permit greater latitude in improving the design’s timing, power and area than FSM-compatible changes. Such modular refinements are essential for our methodology to succeed. Surprisingly such refinements are uncommon in practice; there are two main reasons for this:

- 1) It is difficult to write interface specifications which are precise and understandable. These specifications are necessary so that designers of different modules understand what they may assume and what guarantees they must fulfill (the so-called *interface contract*).
- 2) It is hard to construct tests to verify that a refined module is correctly implemented. This is especially true, if the interface contract allows nondeterminism or if constructing a valid input stream depends on the output stream.

At first glance, these difficulties are exacerbated by modular refinement. Each new module added is another place for the interface contract to be worked out, and for new tests to be applied (*unit testing*).

In this paper we describe a design flow that addresses these difficulties by having the designers leverage information about interfaces in module variants. In particular we discuss properties of interfaces which are needed to support modular development (Section 2). Second, we show how the testing problem can be mitigated by using whole-system tests as unit tests, and how refined modules can be used to generate new tests for the rest of the system. We combine this observation with step-wise refinement to develop a complete design flow (Section 3). Finally, we show the use of this design flow via a practical example: developing an in-order processor from an initial specification (Section 4) to a modular two-stage pipeline (Section 5) to a four-stage pipeline with a more realistic memory subsystem (Section 6). Finally we turn our design into a superscalar processor that executes two instructions at a time (Section 7).

We present each of these steps by showing designs for a minimal 4-instruction ISA (Add, Load, Store, Bz) in a made up language of Guarded Atomic Actions. In reality, we have used the SMIPS ISA (a simplified MIPS ISA with 35 instructions) and implemented each of the designs presented

in Bluespec SystemVerilog [1]. Every design presented in this paper has been synthesized to the gate level and can easily be taken to ASIC or FPGA implementations.

The main contribution of this paper is a design flow that exploits modular refinement to iteratively refine interface contracts and the test suites which check them.

2. Language Support for Interface Contracts

Modular refinement relies on designers having sufficient understanding of interface contracts. While it is possible to find well-documented interface specifications for things like *standard* buses, in general complete interface specifications do not exist for most modules. Specifications are likely to be complicated when complex timing and concurrency issues are present. Such specifications take enormous effort to create and may not be justified when the interfaces are primarily for the internal use of the design team.

We think it is unlikely that designers would be willing to write precise interface definitions for someone else's benefit. However, any interface property that is required to express a design in an hardware description language (HDL) and enforced by the language tools is generally considered to be an aid instead of a burden by the designer. One can draw an analogy with *strong static typing* in software languages like Java, or Haskell. Type safety does not represent extra effort by the user – instead the user thinks of his program in terms of types. Thus, enforcement of type safety guarantees a level of correctness while simultaneously encouraging good design practice.

It is noteworthy that static types provide only a partial guarantee of correctness: there are always deep and important properties of programs that cannot be captured via types. For this reason we divide interface properties into three distinct categories. First, properties whose correctness is guaranteed by the tools, *e.g.*, type safety. Second, properties which are stated rigorously, but possibly informally, by the designer, *e.g.*, FIFO ordering. Such properties may or may not be enforced by the tools. Lastly, *implicit properties* which the designer may be assuming without being aware of it *e.g.*, that a module only ever gets at most N outstanding request at a time. In the rest of the section we discuss general interface properties which are relevant in all designs. We suggest that an HDL should have direct representations for these concepts so that tools can enforce them. We believe these guaranteed properties can make a design flow based on modular development and refinement practical.

2.1. Interface as a Collection of Methods

The interface specification for a Verilog or VHDL module is a list of ports, which is nothing but a grouping of wires. In Verilog, these groupings are all interpreted as bitvectors; interpreting the data (its type) is left to programming and design conventions and is not enforced by the tools. As an example, one can confuse a clock input with a boolean or a 32-bit integer from a 32-bit floating point number. This confusion leads to great difficulties in debugging modular designs. We think that instead of viewing module interfaces as a collection of wires, it should be viewed as a collection of methods as in an object-oriented software language. A module

instantiation should be viewed as an object and the compiler should implicitly connect the wires representing the method call to the wires of the ports representing the called method.

2.2. Event-based Timing Specifications

When designing synchronous hardware systems it is tempting to write specifications with cycle-level timing, such as “The signal on READ_ACK appears 5 cycles after the signal on READ_EN.” These specifications, though easy to verify, are too restrictive for modular refinement. Consider the case where a module is not meeting the clocking constraint. The most natural solution may be to add an extra pipeline stage in the block; this is ruled out by our specification. Furthermore, the rest of the system may have been implemented relying on this behavior, and may too rigid to accept this change. Instead of encoding the precise timings, it is preferable to have *valid bits* indicating the validity of all inputs and outputs of a module.

2.3. Atomicity Specifications

One would like the modular interfaces to prevent an operation on a module if the module is not ready for it. For example, the interface of a FIFO module should prohibit enqueueing into a full FIFO or dequeueing from an empty FIFO. The user can explicitly add a predicate *guard* to every operation and the compiler can automatically stall an action calling for this operation as long as the guard is not ready. This can be done in a synchronous language like Verilog by conventions, but no tools exist to prevent the user from inadvertently misusing such conventions. The question is what should happen if we want to do two operations together and one cannot happen, *e.g.*, reading a value and then enqueueing it into a FIFO. A reasonable thing to do is to stall both as we described them both as happening together, in essence merging the two guards, a straightforward task for the user. When we extend this to arbitrary operations and allow conditional calls, this task becomes prone to errors. The most natural way to ensure correctness is to let the user specify the operations that should be performed together (*atomically*) and have the guards be automatically combined.

2.4. Concurrency Specifications

A common source of errors in using a module is in understanding the behavior of using two parallel operations in the same cycle. Consider a FIFO module which supports concurrent enqueues and dequeues; such FIFOs are used everywhere in pipeline designs. Even with the assertion that we only call enqueue and dequeue when appropriate, our specification is still incomplete. Should an enqueue in a full FIFO be permitted because a dequeue is happening in the same cycle? Should a dequeue be permitted from an empty FIFO on which a concurrent enqueue is happening? Both FIFOs are semantically meaningful, but have different functional behavior. One defines the functionality to be “dequeue before enqueue” and the other “enqueue before dequeue.” There is a fair amount of accumulated wisdom which says that the functionality of concurrent operations should be explainable as a sequence of the individual actions [3], [6], [13].

Interface definitions should specify which operations can be performed concurrently and what is the resulting functionality when they are performed concurrently.

Ideally, the above properties should be directly represented in the HDL and enforced by the compiler. One existing abstraction which captures all of these features is Guarded Atomic Actions [4], [6]. We present our design flow in the context of this abstraction.

3. The Design Flow

In this section we present our design flow which can mitigate the upfront cost of specifying interface contracts and building unit test suites.

3.1. The Interaction of Modularization and Testing

All designers practice modularization to an extent – as a practicality, they partition their designs on boundaries where the partitions (and thus interface contracts) are intuitive, *e.g.*, a processor’s memory unit. These boundaries are understood sufficiently that it is straightforward for informed designers to produce many refined designs.

However, difficulties arise when the designers must subsequently demonstrate the correctness of a refinement (via test suite). For some modules, *e.g.*, adder, register file, it is relatively easy to specify functional correctness, though implementations can differ substantially in regards to the timing characteristics of the module. If the system is not latency-insensitive then even two functionally equivalent modules with different timing characteristics can break the whole system because other modules may be making assumptions with regards to the specific timing of the refined module.

Some intuitively obvious interfaces are too complicated for someone to design a unit test suite to fully exercise them. Consider a processor’s memory unit interface. Can the memory system internally reorder memory requests? Can the processor issue more than one request to the same address? Must the responses come back in order? The answer to these questions often depends upon the higher-level semantics of the system. Does the system specification admit nondeterminism? A simple form of nondeterminism may be just returning cache-hit/cache-miss information for the same load request while a more difficult type of nondeterminism may be when the memory system can return entirely different values because of shared memory.

When a module interacts with the rest of the system in a complex manner, it may not even be clear what constitutes a valid input for unit tests. In contrast whole-system tests are often easier to generate than the tests for individual modules. Microprocessors provide a good example of this relative difficulty. It is easy to specify, using a software simulator for example, the correct answers for any input program. This is not so for most internal units like the reorder buffer or TLB. In the next section we discuss how whole-system tests can be leveraged as unit tests.

3.2. Recasting Whole-System Tests as Unit Tests

Suppose we have a system S composed of two modules (A_0, B_0). Furthermore suppose we have a test suite that S

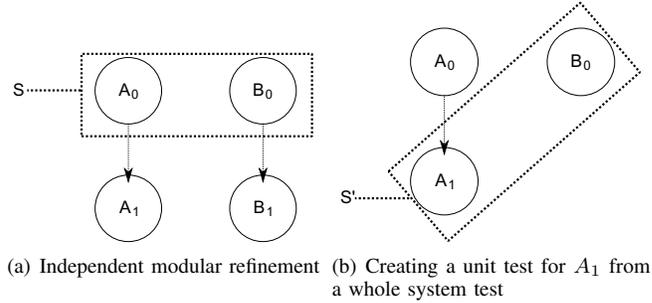


Fig. 1. Unit testing using whole-system tests

passes in this configuration. Now, two design teams independently refine A_0 to A_1 and B_0 to B_1 , respectively (shown in Figure 1). Now the design team working on A can use the whole-system tests with B_0 as unit tests for A_1 and similarly the other team can use these same tests with A_0 for B_1 .

When a whole-system test fails after the refinement of a module, it is important to understand the possible reasons why the refined module may not work in an old design context:

- *The implicit interface contract is more constrained* than what the designer changing a module believes. For example, if the module produces messages in a different order or at a different timing it may break the whole design. The way to fix this problem is to make the implicit constraints explicit in some manner, and renegotiate a new interface specification. If the newly refined module has been unit tested independently, then the failure of a whole-system test indicates that the initial system design was wrong and the designer is required to create a replacement.
- *The implicit interface contract is too unconstrained* in the sense that designers view meeting the interface contract as too hard or expensive. The solution here is again to renegotiate the contract. For example a processor may demand that memory requests are returned in-order though the memory system is permitted to execute them out-of-order.

Often as we do refinements, we expand the whole-system test suite to better exercise new features of the design. Whenever the test suite is expanded, we have to run the new tests on all the old versions of the system. If they fail, it is a sign of a latent error. At this stage, we have to either fix or reject the failing designs. It should also be noted that if a refinement changes the timing characteristics of a module and the whole-systems tests work then we gain more confidence in the correctness of the modules that were not refined.

3.3. The Design Flow

Now we present a design flow based on these ideas. We think that complex designs should be done by step-wise changes, with the goal of producing a working system at each step. In order to describe our design flow we have to assume that the designer has a good idea of these intermediate steps. For a superscalar processor we may want to start with a single-cycle unpipelined design, refine it to include the more realistic multi-cycle memory systems and functional units, increase the pipeline depth to accommodate these changes, and then convert the working pipeline to a low performance superscalar

taking multiple cycles for each stage to do what should be done in one cycle in the final design. From here we can increase the parallelism to allow each stage to do its task in the desired number of cycles.

The first step is to create an initial working design. The focus of this design should be to serve a baseline executable specification. Given this, our design flow progresses by repeatedly applying one of the following operations:

- *Re-modularize the system* by partitioning a module into two or more submodules. This is done to mitigate complexity of further design changes or to allow independent teams to work in parallel. This is straightforward for parts which have a clean interaction model with the rest of the design, *i.e.*, modules that do not call other modules. Good examples of such modules are register files, FIFOs, ALUs etc. It is important to understand the concurrency requirements for the use of such “component modules”. For the rest, the designer should consider if there are reasonable divisions of labor which may be applied to the design and then try to draw module boundaries accordingly.

Any such re-modularization generates new interfaces. It is important to understand the properties of these interfaces that are not enforced by the language tools.

- *Refining a module by changing its implementation.* This step requires either unit testing for this module or expanding the whole-system test suite to exercise new corner cases. If the refined module is unit tested then we have to make sure that the old modules pass the whole-system tests in conjunction with the refined module. If the test suites have been expanded then we again have to make sure that the earlier versions of the design pass these tests. We can only keep those modules in our database that pass every whole-system test.
- *Change an interface contract* to enable further refinements. Such a change has to be done carefully because it *always* requires changing all the affected modules simultaneously, and may require significant effort before the modified system again passes the whole-system test suites.

In the rest of the paper we present the development of a two-way superscalar processor using our design flow starting with a single-cycle unpipelined implementation. Most of the intermediate designs we build for debugging the final design, in practice may turn out to be useful as working variants of the final design. Similarly many component modules may find uses in other very different designs.

4. Baseline: A Single-Cycle Processor

In this section, we will explain the language of Guarded Atomic Actions (GAAs) using the example of our processor. A GAA, or rule, consists of a state change specification (*action*) and a predicate (*guard*) which signifies when it is valid for the state change to occur. Any rule whose guard is true can be, but does not have to be, executed. When there are many rules that can be executed, the semantics say that we arbitrarily choose one to execute. Thus, in a GAA system all behaviors are explainable in terms of a sequence of rule firings.

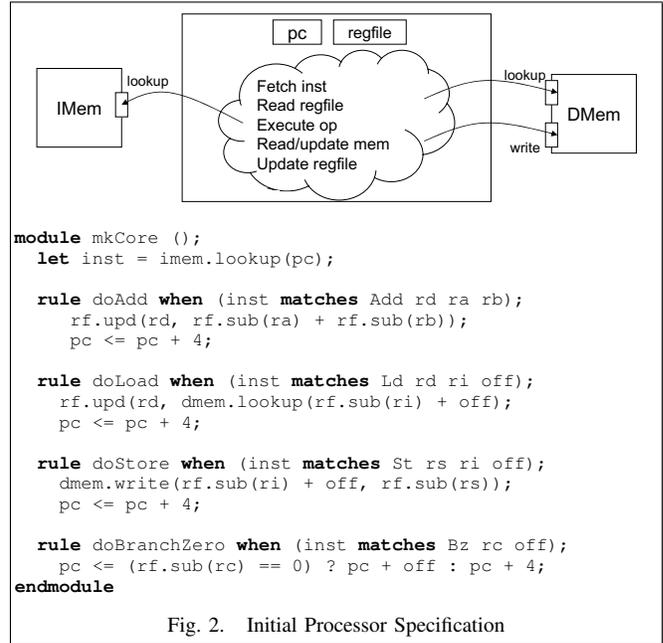


Figure 2 gives the rules for a baseline processor. This processor is easy to verify and will serve as the functional specification for our refinements. The instructions themselves are represented with a tagged union data type, and extracted with the `matches` syntax. For example, `inst matches (Bz rc off)` matches a branch-if-zero instruction with two parameters, a condition register and an offset, which are stored in the variables `rc` and `off`.

In this example, the rules are mutually exclusive and only one of them can be enabled at a time. Therefore there are no scheduling issues. In fact we could have written these four rules as a case statement on the value of `inst` without loss of generality. We may use the case syntactic form when convenient.

4.1. Guarded Module Interfaces

Rules define the internal behavior of a module. GAA modules interact with each other through guarded interfaces. Consider the memory module used by the processor in Figure 2. It uses the following interface:

```

interface Memory;
  method Value lookup(Addr a);
  method Action write(Addr a, Value v);
endinterface

```

The `write` method’s type is `Action`, meaning that it performs a state update but does not return a result. Action methods must be called from within rules, or from other action methods. On the other hand, the `lookup` method simply returns a `Value` — it is side-effect free (combinational logic) and therefore it may be called outside of rules. Realistically, a load to memory may have all kinds of side effects, such as initiating cache fills or updating recently-used data. We will tackle refining this illustrative memory interface to a more realistic version in Section 6.

Another module that is used in this design is a register file with two read ports and a write port. Since the add rules call all these methods concurrently, we need to understand

the semantics of value methods and action methods executing concurrently. For our design to work correctly, both the value methods *i.e.*, `rf.sub(ra)` and `rf.sub(rb)` must produce values before the action methods `rf.upd(rd,v)` updates the register file. This is the contract the `rf` module interface must fulfill. Later we will see situations where we may want slightly different concurrent semantics that will make the `rf` behaves like a bypass register file, *i.e.*, the update is visible to the reads in the same cycle.

The interface methods corresponding to the physical memory and register files are always ready to be applied, *i.e.*, have no guards. If `dmem.lookup` did have a guard then the `doLoad` rule would be enabled only when `dmem.lookup`'s guard is ready.

4.2. Synthesized Hardware

Our GAA semantic model assumes that all the reads take place before all writes in a rule. Consequently `x <= y; y <= x` will swap the values of `x` and `y`. This extends to methods. Thus when a method which does a read of the state happens in the same rule as one which does the update, the read method is presumed to happen first. To synthesize hardware, we further assume that each rule executes in one clock cycle. Consequently, such swaps can be done in hardware without introducing temporary state. To convert this description into synchronous hardware, we need to select which rules should be executed in each cycle. In general, since multiple rules can be enabled simultaneously, a scheduler must be synthesized to select the rules for execution [6]. For performance, the compiler tries to generate a scheduler which lets as many rules as possible execute in a single cycle without violating the one-rule-at-a-time semantics [8].

An implementation of the unpipelined, single-cycle processor would have a very slow clock: in a single clock cycle, it fetches an instruction from memory, reads the register file, performs the operation including a data fetch or store, and commits the result (see Figure 2). Even so, we can simulate the design in order to confirm that the design is behaving correctly. Thus, this design can serve as a kind of *executable specification* which we can use to verify further refinements.

This design can be tested by running a number of micro- and full benchmarks to exercise every instruction. We have used a suite of tests consisting of five small applications (`vvadd`, `multiply`, `towers`, `median`, `qsort`) for which code was generated from C by an SMIPS compiler.

5. The First Refinement: Pipelining

In this section we focus on turning our unpipelined processor into a two-stage pipelined processor, where the first stage will fetch and decode instructions and the second stage will do the execution (see Figure 3). This transformation will reveal several subtle concurrency issues, especially in regards to the component modules.

5.1. Two-Stage Pipeline

To implement a two-stage pipeline, we introduce a FIFO buffer between the stages to hold partially completed instructions, *i.e.*, the instructions where we have replaced the register operand names with the actual values from the register file.

The fetch rule requests the instruction at `pc`, decodes it, looks up the operands in the register file and puts the results in the FIFO buffer (see Figure 3). The execute rule gets an instruction from the buffer, does the appropriate operation and writes back the result. Since the instructions are supposed to behave as if they happen atomically and in order, it is possible that there is a *data hazard* where the fetch rule does not read a register update corresponding to an instruction currently in the FIFO buffer. To avoid such data hazards, we do not read the register file when there is an instruction in the FIFO buffer that is going to modify the register that the fetched instruction wants to read. Implementing this *stall* condition requires an ability to search the FIFO buffer for instructions writing to the appropriate register name.

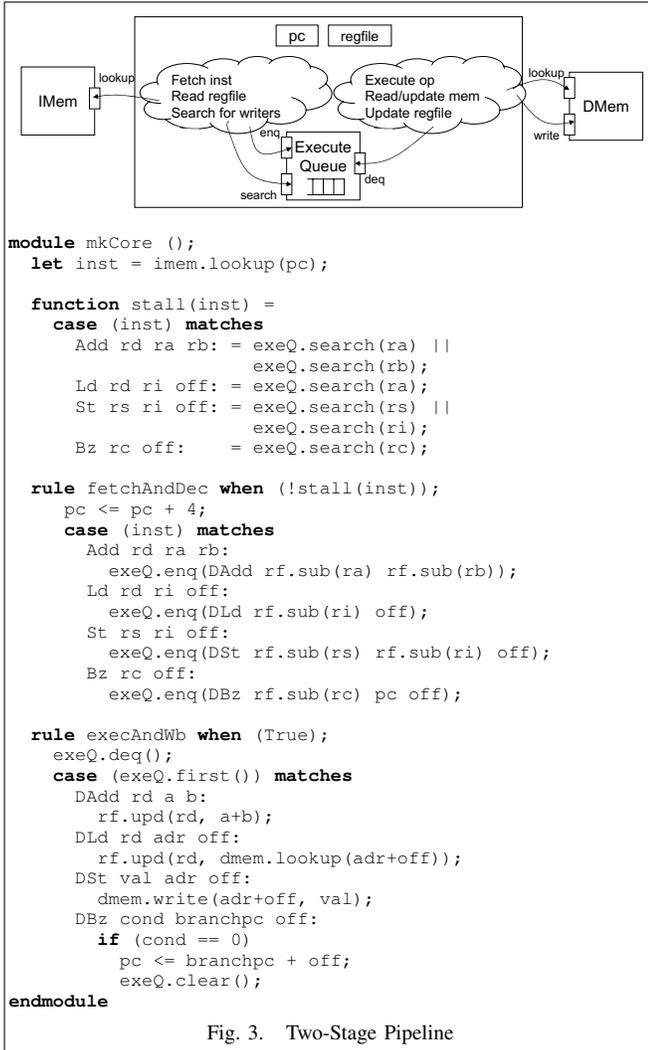
A pipelined machine is expected to execute one instruction while fetching the next. However, we cannot update the `pc` correctly until the instruction is executed. Thus it is necessary to speculate the address of the next instruction and initiate fetches even before the execution of the previous instruction completes. This is accomplished via a branch prediction mechanism, which in its simplest form always predicts the next sequential instruction, *i.e.*, `pc+4` (see the `Bz` case in the `fetchAndDec` rule of Figure 3). When execute happens, it verifies that the next instruction address was predicted correctly. If not, it updates the `pc` with the correct value and removes the wrong path instructions by clearing the intermediate buffer (see the `execAndWb` rule's `DBz` case).

Verifying the correctness of these rules is not difficult – it is straightforward to write benchmark programs that exercise various stall conditions and speculative branch executions. However, the designer's responsibility is not over until the right degree of pipelining is achieved. This, as we will see momentarily, requires arguments about semantics of concurrent execution of various method calls.

5.2. Concurrency Issues in Components

We can safely execute the rules in any order and often even concurrently because the stall condition guarantees that the fetch of operands does not happen in case of a data hazard. The only exception is when a branch prediction turns out to be wrong. In this case, the backend has to change the `pc`. If we require the frontend to read out the new value in the same cycle, it will create a long critical path. We choose to disallow fetches from happening in the same cycle as a mispredicted branch. In the GAA jargon, we say that the fetch and the branch resolution rules conflict. The designer can specify that the branch resolution rule should have priority to the compiler.

For fetch and execute to happen concurrently, we need to understand what functional behavior we want. Do we want the system to behave as if execute happens and then fetch or the other way around? A moment's thought will reveal that we want the execute rules to happen before the fetch rule, as execute is holding the older instruction and making space in our FIFO buffer for the new one. However, this effect cannot be achieved unless our components (*e.g.*, FIFO buffer and register file) behave in a manner matching this ordering. Note that since the execute rules dequeue and fetch enqueues in the same FIFO, we need the FIFO to behave as if the dequeue happens before the enqueue when they happen concurrently. These

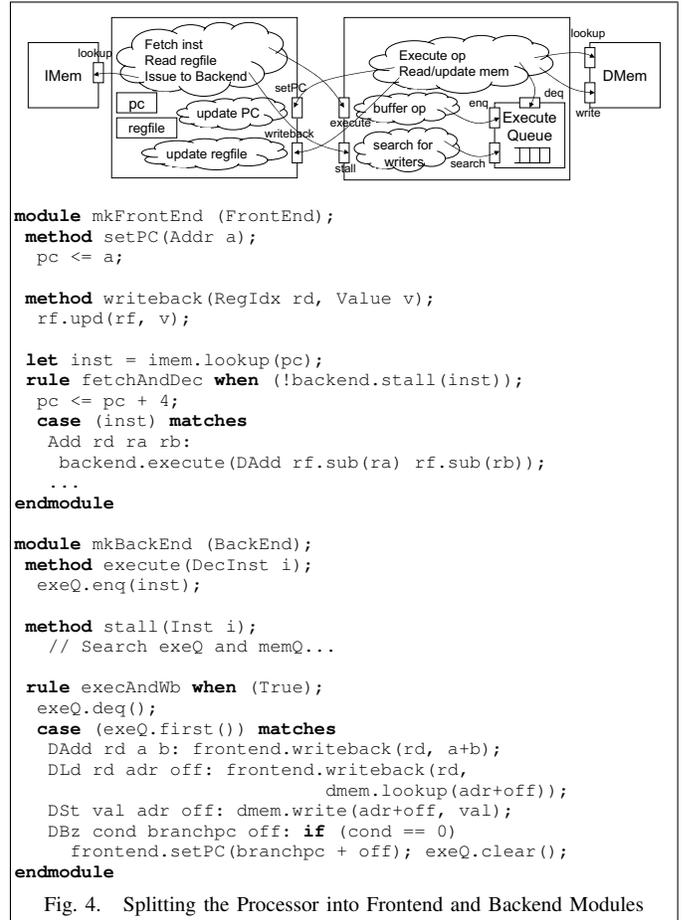


semantics also permit concurrent enqueueing and dequeuing in a one-element FIFO. Furthermore, when we search the FIFO in the fetch rule, it must ignore the value being dequeued in the same cycle. Similarly, the register file must behave as if writes happen before reads, *i.e.*, the write is visible to the read call, effectively requiring a data bypass. This is different from the original register file in our single-cycle processor.

Lack of space does not permit us to discuss the detailed implementation of these components. Regardless of the concurrency model we assume for FIFOs and register files, our pipeline will work correctly, though they may not have the ideal concurrency. The concurrency model is of paramount importance in understanding performance; a correct design with too many pipeline “bubbles” can be debugged by analysis of the components’ concurrency specifications.

5.3. Modularizing the Two-Stage Pipeline

Before we undertake more refinements like replacing the magic memory with a more realistic memory system, we would like to partition this design into two modules as shown in Figure 4. The instruction FIFO has been associated to the execute unit, while the register file has been assigned to the fetch unit. Given this organization, we need to expose the operations on these submodules that are used by the other module via interfaces. This leads to the following interfaces:



```

interface FrontEnd;
  method Action writeback(RegIdx r, Value v);
  method Action setPC(Addr a);
endinterface

```

```

interface BackEnd;
  method Bool stall(Inst i);
  method Action execute(DecodedInst i);
endinterface

```

Since modularization may hide modules (*e.g.*, the register file) by encapsulating them within another module, all method calls to the such hidden modules have to be made via the method calls of the encapsulating module. For instance, in the fetch unit the use of the `stall` function has to be replaced by a call to the backend method `backend.stall`. Similarly, the call to enqueue into the FIFO buffer must be replaced by the call `backend.execute`. Note that such calls make this interface organization recursive. One way to understand the semantics of this recursion is by considering *in situ* replacement of method calls with their bodies which results in the original non-modular code in Figure 3.

Our language semantics guarantees enforcement of types as well as proper use of guards in method calls. But there are deeper or design specific properties of these interfaces that the designer must understand before refinements can be undertaken.

- 1) There are FIFO constraints in the way backend processes instructions; the updates to `rf` and `pc` must be issued in an order that is consistent with the input instruction stream.

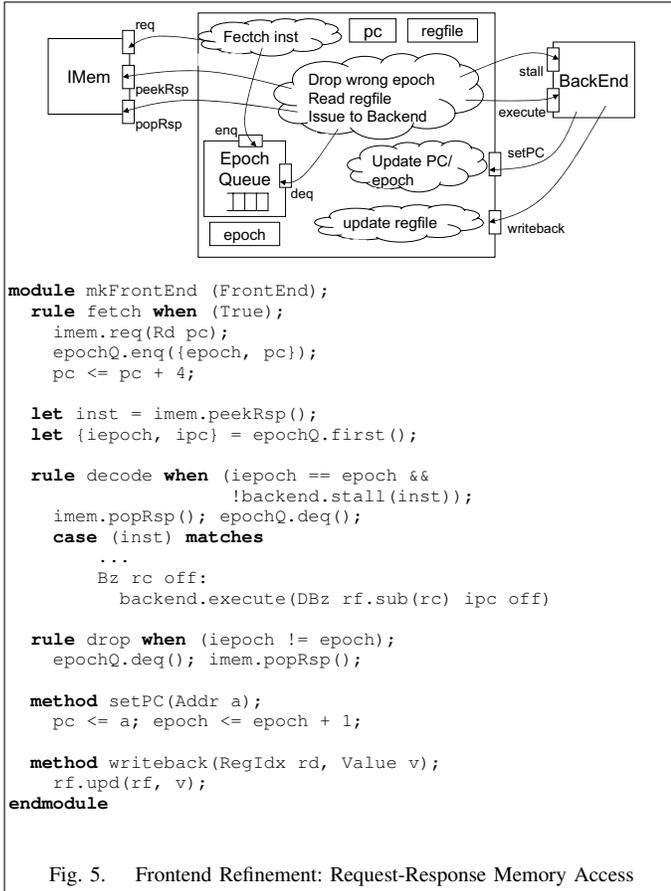


Fig. 5. Frontend Refinement: Request-Response Memory Access

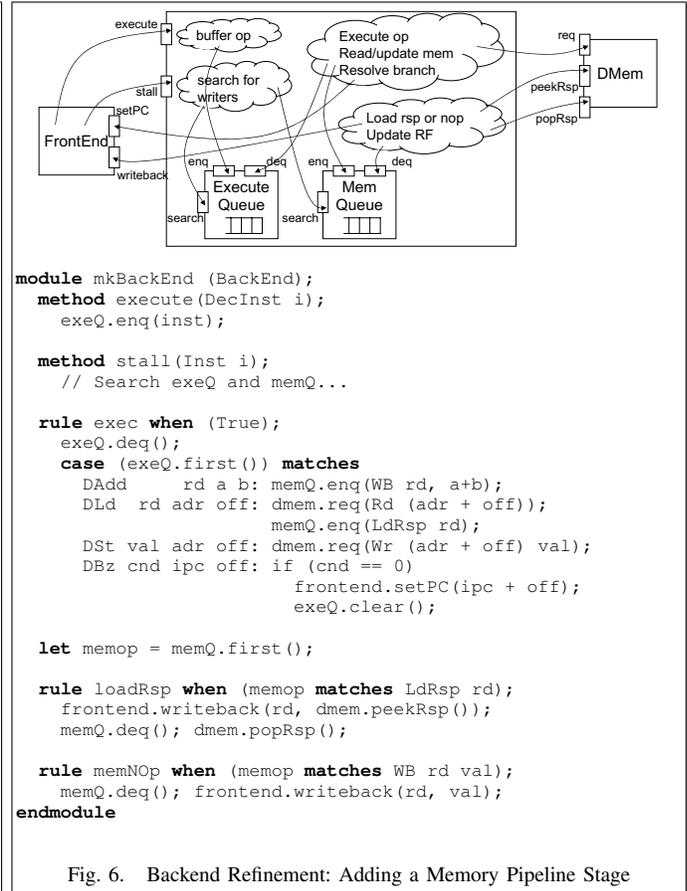


Fig. 6. Backend Refinement: Adding a Memory Pipeline Stage

- 2) The frontend must guarantee that after it processes `setPC` the next instruction enqueued in the backend corresponds to this `pc`.
- 3) The frontend also must guarantee that after it processes `writeback`, all subsequent instructions sent to the backend observe this update.
- 4) The frontend is required to check that enqueueing an instruction is safe by consulting the `stall` method. Consequently, the backend is required to make sure that the `stall` function is true whenever it is not safe to enqueue.

None of these properties will be checked by the compiler but rule atomicity and guarded interfaces offer powerful tools to the designer for controlling behavior. As an example, consider what happens when the backend wants to call the `setPC` method. The `execAndWb` rule gets stuck until such a call can be made, which in turn will prevent the next instruction from being processed. Consequently, things will come out in order. Rule atomicity also guarantees that either both or neither `setPC` nor `exeQ.clear` are executed which is paramount for correctness.

For performance reasons, we want our pipeline to allow full throughput. This means that in most cycles we should be able to add an instruction (via `execute`) to the backend and have a result returned (via `writeback`). These happen in two separate operations so they must be ordered in some way in our design, this choice determines if the written instruction affects the `stall` method and if the frontend needs to be aware of the write in the same cycle. This corresponds to deciding how the frontend orders `writeback`

with the operation calling `backend.execute` and how the backend orders `execute` and the operation which calls `frontend.writeback`. Though in our current two-stage design it is clear we want `writeback` to happen first in a cycle, this may not be the case for further refinements. The order at some level is not important; the first priority is that the decision is made consistently, so that we get the desired parallelism. The compiler will make sure that *if contradictory concurrency orderings are chosen the system will simply treat these rules as conflicting and execute them one at a time*.

6. The Second Refinement: Replacing the Magic Memory System

A multi-cycle memory has a different interface than a magic memory. The `lookup` method of magic memory has to be split into two methods – one to send a request and one to pick up the response. Often a multi-cycle memory is implemented in a pipelined manner and consequently may accept several requests before producing any responses. The following is a possible interface for such a memory:

```

interface Memory;
  method Action req(MemReq a);
  method Value peekRsp();
  method Action popRsp();
endinterface

```

To make a request, we send a memory request consisting of either just an address for reads or an address and a value for writes. To keep the interface simple, we assert that the memory subsystem returns results in order. The response value

can be accessed via the `peekRsp` method and is taken out of the memory subsystem when `popRsp` is called. Though a realistic implementation of the memory is too complicated to discuss in this paper, we can view it as if the old memory places the result of lookups in an intermediate FIFO, on which the response methods `peekRsp` and `popRSP` operate.

To use this new memory in our frontend, we split our single fetch rule into two, effectively pipelining the frontend into a fetch and decode stage. The first half makes memory requests and changes the `pc`, the second takes memory responses, decodes them and passes them to the backend. Since we need the `pc` value to decode the instruction, that value is also passed from the fetch rule to the decode rule via a FIFO. Since memory accesses may take many cycles which cost instructions to stay in the queue for a long time, we want our FIFO to hold multiple instructions so that the memory system can overlap miss penalties.

At first blush this appears to be a correct implementation of the module. However, when we run our tests, the system fails at certain branch instructions. In the current design, when there are outstanding instructions waiting between fetch and decode and `setPC` is called, these instructions will not be from the correct addresses. Ideally, we could just clear the FIFO and clear all instruction memory requests when we execute the `setPC` method. However, the operation is too complicated in general to allow an instantaneous clearing of the memory system. Instead, we employ the notion of *epochs* to assure that instructions on the mispredicted path are removed one by one as they come out.

The frontend keeps a register with the epoch it is operating in. Every fetched instruction is tagged with this epoch. When `setPC` is called, we increment the frontend's epoch. Before we pass a decoded instruction to the backend, we check that its epoch matches the current frontend epoch. If so, we send it on, otherwise, we know that it was on the wrong path and drop it.

Changing the data memory read interface to allow multi-cycle operations is quite similar to the instruction memory, requiring us to split the operation into an execute and write-back stage. The code for both the refined frontend and backend is given in Figures 5 and 6.

7. Changed Interfaces: Making a Superscalar Processor

“Final” designs are often not final; finished designs are often adapted or used in future designs. A good methodology should facilitate such uses. To illustrate this, we further refine our pipeline into a two-way superscalar processor.

7.1. Changing the Interface Specification

Our old design is very similar to our new design. The instruction set remains the same, making our test suite equally applicable. The only functional change is that now instead of operating on one instruction per stage we need to handle two instructions in parallel. This requires us to double most of our commonly used resources, *e.g.*, taking the register file from having two read and one write ports to four read and two write ports. Similarly, to make use of this extra hardware the instruction memory will need to support returning two

instructions at a time. Since instructions are likely to be read in sequence we can get away with changing the two instructions per read, giving us most of the benefit of two read ports without the high hardware cost.

The same cannot be said about the data memory as consecutive accesses are usually to unrelated addresses. However, since most instructions are not memory operations, it is reasonable to use the same interface as before and only support one memory operation per pair sent from the frontend. This requires that the frontend not to pass such instructions pair. This is similar in nature to its previous task of making sure that instructions do not have data dependencies to currently active instructions. Now that we pass two instructions, check must also occur between pairs.

The last interface consideration to worry about is what to do if we have only one instruction to pass from the frontend to the backend. We need to signify that the other instruction is not valid. To achieve this, we change the type of the second instruction in a pair as a *Maybe* instruction, which adds a predicate bit to signify if the instruction is valid.

After these changes we are left with the following changes to the interfaces:

- The frontend passes an instruction and a predicated instruction to the backend. Only one memory operation is allowed per pair and is always the first instruction.
- The backend may update two registers at once.
- The instruction memory data width is doubled

7.2. Obtaining an Initial Design

Having defined our new interfaces, we would like to generate an initial design as quickly as possible. Given the modules we had constructed in Section 6, we should be able to cobble together a minimal working model. The changes to the instruction memory is straightforward. The simplest modification we can make to the frontend is to keep it single-way. Whenever we decode an instruction we send an instruction pair with an invalid second instruction. This fulfills the interface obligations. The backend is slightly more complicated as we need to handle *both* instructions in a bundle to meet our obligations. We can leverage the old design by merely serializing the execution of the instruction pair into two cycles.

Notice that neither the frontend nor the backend are truly exercising the interface. The frontend does not make use of half of its interface to the backend. Similarly the backend does not write back both instruction results concurrently. This will have significance during testing.

7.3. Refining the Modules

The first frontend refinement task is to allow it to read two instructions at a time. Given that we've already pipelined our frontend, we would like to base our changes on this.

Most of the changes are fairly structural. As we mentioned above the first step is to widen the instruction memory response to two instructions and we need to extend the register file to have more read and write ports. To appropriately keep track of when instructions can be sent to the frontend need to add another `stall` method to check the second instruction.

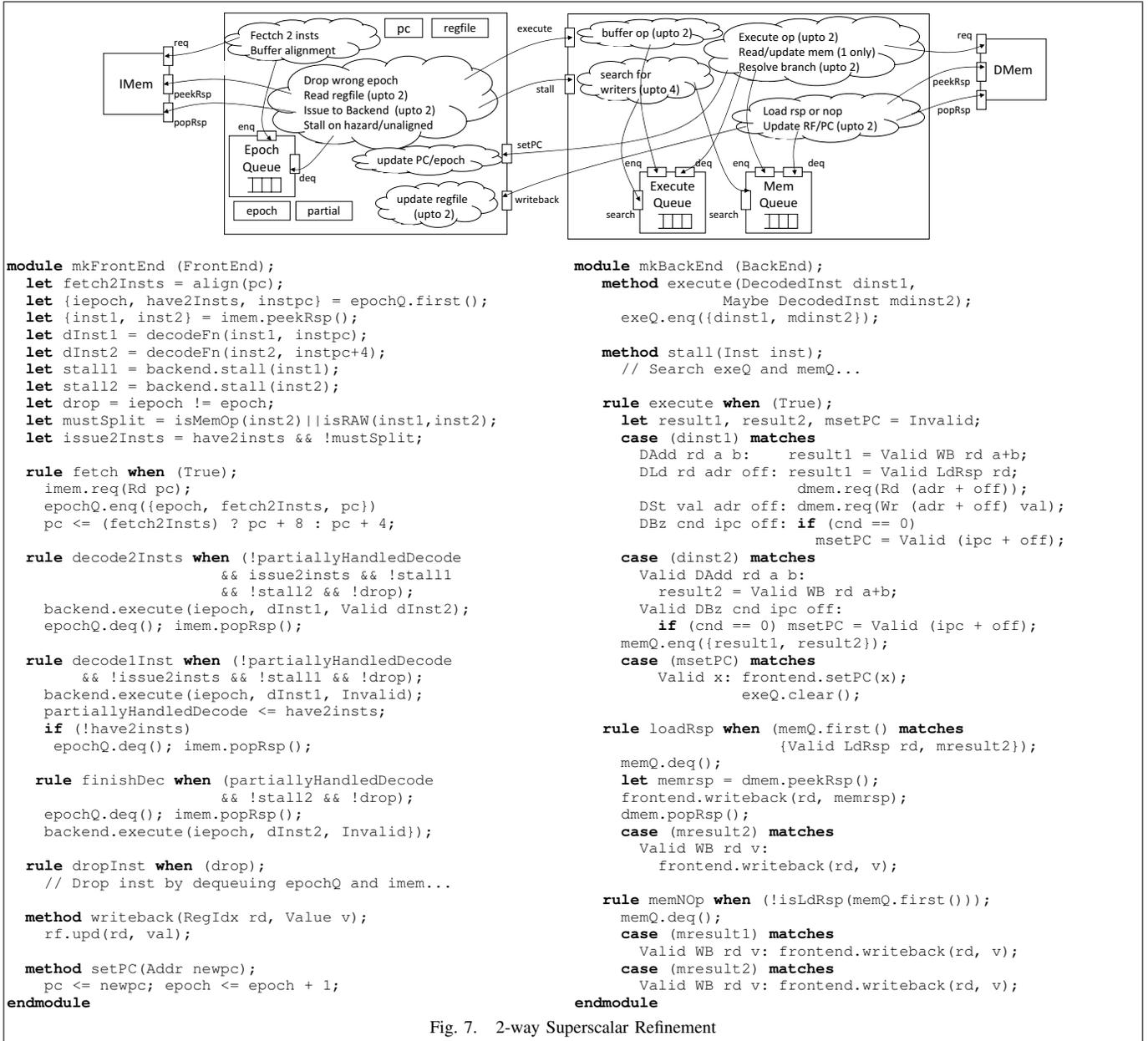


Fig. 7. 2-way Superscalar Refinement

Having modified all the necessary submodels we can change the rules of the frontend itself. For the fetch rule we have two different behaviors. If the instruction is aligned to instruction pairs (represented by the `align` function), do two instructions, otherwise we only handle one from our instruction memory read. We update the `pc` accordingly.

Changing the decode rule is trickier than the fetch rule. Though the fetch rule may be able to pass two instructions, we may need to split them to maintain our interface contract. Specifically we need to split pairs when: 1) the instructions fetched are not aligned to the instructions pair boundary 2) the second instruction is a memory operation 3) the second instruction has a data dependency with the first instruction.

To implement this we need to add state (the boolean register `partiallyHandledDecode`) to signify if we are part way through a split instruction pair decode.

To finish our new frontend we need to modify our handling

of rules from the backend now that they are returned in pairs.

Though modifying the backend is simple, lack of space does not permit us to discuss the refinement to the backend. The resulting code is listed in Figure 7.

7.4. Results

We present a few quantitative results to show that all of these designs were actually implemented. No conclusions should be drawn based on these numbers as much more testing and refinement is needed for such architectural exploration.

Figure 8 shows the performance results of our final SMIPS superscalar compared to the original non-superscalar processor. As we can see, the IPC improvements on the five testbench applications range from 6.25% (towers) to 16% (vvard).

We synthesize the two designs on a FPGA (Virtex-5 LX110T) using Xilinx ISE 10.1. Both designs meet our 50MHz target clock frequency after place-and-route (PAR). As

expected, the superscalar processor uses more flip-flops (6529 vs. 6165) as well as logic (13687 LUTs vs. 10170 LUTs) than the non-superscalar processor.

	median	multiply	qsort	towers	vvadd
Non-superscalar	0.44	0.51	0.46	0.48	0.50
Superscalar	0.47	0.57	0.51	0.51	0.58

Fig. 8. IPC of the non-superscalar and the 2-way superscalar processors

8. Related Work

A lot of research has gone into producing verifiers (a combination of human and machine) that try to extract all the appropriate high-level information from a design. This can be daunting because the design may have already been subjected to all of the designers' clever optimizations. Since RTL interfaces are given in terms of synchronous implementation behavior, extraction of useful high-level properties often requires the verifier to examine the full scope of a design, and ignore any inherent modularity in the system. Despite all this complexity, a great deal of progress has been made. Researchers have shown that the use of uninterpreted functions [7], [10], [14], systematic proof deductions [11], and other general approaches [5] coupled with domain-specific knowledge makes it possible to handle fairly large and complex systems.

Two areas where modular refinement has been explored extensively are 1) the cycle-accurate modeling of processors on FPGAs [12]; and 2) the latency insensitive refinements of synchronous sequential circuits [2], [9], [15]. In these works the timing of each module is specified *a priori* in terms of "model cycles" and any change in this model timing potentially causes a functional error. The goal is to allow implementations to take different number of FPGA cycles (ASIC cycles) to simulate a model-cycle without compromising the functionality. That is, one should be able to recreate the model cycle behavior regardless of the actual implementation cycles taken. Though all these works require reasoning about interfaces, the methodology presented in this paper is about *flexible* as opposed to *rigid* interfaces. Also in this paper there is one-to-one correspondence between the cycle-time to execute a rule and actual implementation clock.

9. Discussion

We have presented a methodology based on modular development and step-wise refinement to develop complex digital systems. Even though the potential advantages of modular refinements are myriad, it is rarely used in practice. We have argued that this is because of a lack of support for refinements in design languages. If the interface definitions supported by a language are too rigid, they leave no room for refinements. If the semantics of the language do not enforce a particular property of interfaces, then at best a designer can impose some discipline or conventions on himself. Experience has shown that properties that are not enforced by the tools or language semantics are rarely maintained in practice. Good language support for modularity requires one to write down the interface properties just to express the design. This approach requires dramatically less effort than writing separate interface specifications (for documentation or formal proof).

When modular refinement is combined with step-wise refinement we get a systematic way of developing complex systems with distributed teams. Teams are able to use modules developed by other teams to enhance their own unit testing capability by using common whole-system tests. When a module works in different contexts our confidence in its correctness grows. Another major benefit of our methodology is that it produces many different working implementations of a specification. Many of these implementations are extremely useful in architectural exploration and performance debugging.

It will be interesting to explore if the language semantics can also be used to capture the FIFO property of input-output behavior, which is a requirement in many specifications. Alternatively, one could apply other formal techniques to prove the FIFO property. We have found this remarkably difficult in practice.

Acknowledgments

This project was partially funded by NSF grants CCF-0541164 on Complex Digital Systems.

References

- [1] Bluespec, Inc., Waltham, MA. *Bluespec SystemVerilog Version 3.8 Reference Guide*, November 2004.
- [2] Luca P. Carloni, Kenneth L. Mcmillan, and Alberto L. Sangiovanni-vincentelli. Latency insensitive protocols. In *Computer Aided Verification*, pages 123–133. Springer Verlag, 1999.
- [3] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.
- [4] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as Rule Composition. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.
- [5] Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of the IEEE/ACM International Conference on Computer-aided Design (ICCAD 2000)*, pages 245–252, 2000.
- [6] James C. Hoe and Arvind. Synthesis of Operation-Centric Hardware Descriptions. In *Proceedings of ICCAD'00*, San Jose, CA, 2000.
- [7] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessors control. In David L. Dill, editor, *Proceedings of the sixth International Conference on Computer-Aided Verification CAV*, Stanford, California, USA, 1994.
- [8] Michal Karczmarek and Arvind. Synthesis from multi-cycle atomic actions as a solution to the timing closure problem. In *ICCAD '08*, Piscataway, NJ, USA, 2008. IEEE Press.
- [9] Sava Krstic, Jordi Cortadella, Mike Kishinevsky, and John O'Leary. Synchronous elastic networks. In *Formal Methods in Computer Aided Design, 2006. FMCAD '06*, Nov. 2006.
- [10] S. Lahiri, S. Seshia, and R. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *FMCAD '02*, volume 2517 of *LNCIS*, pages 142–159. Springer-Verlag, November 2002.
- [11] K. L. McMillan. A methodology for hardware verification using compositional model checking. *Science of Computer Programming*, 37(1–3):279–309, 2000.
- [12] Michael Pellauer, Muralidaran Vijayaraghavan, Michael Adler, Arvind, and Joel Emer. A-ports: an efficient abstraction for cycle-accurate performance models on fpgas. In *FPGA '08*, 2008.
- [13] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.
- [14] Jens U. Skakkebaek, Robert B. Jones, and David L. Dill. Formal verification of out-of-order execution using incremental flushing. In *Computer Aided Verification*, pages 98–109, 1998.
- [15] Muralidaran Vijayaraghavan and Arvind. Bounded dataflow networks and latency-insensitive circuits. In *Formal Methods and Models for Codesign, 2009. MEMOCODE '09. 7th IEEE/ACM International Conference on*, pages 171–180, July 2009.