Designing a Processor in Bluespec

by

Nirav Hemant Dave

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
[February 2005]
January 2005

Author .....
Department of Electrical Engineering and Computer Science
Jan 14, 2005

Certified by .....
Arvind
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Designing a Processor in Bluespec

by

Nirav Hemant Dave

Submitted to the Department of Electrical Engineering and Computer Science
on Jan 14, 2005, in partial fulfillment of the
requirements for the degree of
Master of Science

Abstract

In this thesis, we designed a 2-way out-of-order processor in Bluespec implementing the MIPS I integer ISA. A number of scheduling optimizations were then used to bring the initial design up to the same level of cycle-level concurrency as found in standard RTL-level designs. From this, a general design methodology is proposed to effectively express, debug, and optimize large Bluespec designs.

Thesis Supervisor: Arvind
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

There are many people who have in some way contributed to this work and any attempts at an exhaustive list would surely be incomplete. I shall try to properly acknowledge those few who have had the most significant impact.

First, I would like to thank my advisor Professor Arvind for his solid belief in the importance of this work and encouragement to make this into an actual thesis. Without his ever-present interest in this work, I am sure that I would have only a set of unfinished processor designs and a disheveled organization of personal beliefs as to how Bluespec was meant to be written.

I would also like to thank my parents who never let me forget that no matter how good I am, how well I do, or how proud they are of me, that I can always can and should be better.

I would also like to thank Ravi Nanavati, Mieszko Lis, Jacob Schwartz, and the rest of the Bluespec Inc. team for putting up with my confusion when I started, enduring the complaints about the compiler when it did not do as I expected, and sagely granting me the perspective on how difficult improving the compiler can be by assigning me to make the enhancements to the compiler which I had been asking for.

I would also like to thank my former and current officemates Charles O'Donnell, Michael Pellauer, Daihyun Lim, Ryan Newton, and Karen Brennan who put up both with my endless chatter about this research and english advice when they were trying to work.

I would also like to my Nickolas Fotopoulos, my long-time friend and housemate for putting up with my over-exuberance for all this time and for taking time out of his busy schedule which I strongly suspect was slotted originally for some much needed sleep.

Lastly, I would like to acknowledge the supreme help of Boston's cold winters which convinced me so often to spend the night at the office working diligently instead of trudging home.

Contents

List of Figures

Chapter 1

Introduction

The need to speed up the hardware design cycle has caused industry to look at more powerful tools for hardware synthesis from high-level descriptions. One of these tools is Bluespec. Bluespec is a strongly-typed hardware synthesis language which makes use of the Term Rewriting System(TRS)[2] to describe computation as a series of atomic state changes.

Bluespec has been used at Sandburst, Bluespec Inc., MIT, and CMU to describe a variety of complex hardware designs. Previous work has also shown that small but complex designs described using TRS, the formalism underlying Bluespec, are amenable to formal verification [1]. It has also been shown that a simple 5-stage MIPS pipeline and other similarly complex hardware designs can be synthesized from TRS's quite efficiently [2, 3, 6]. What remains to be seen is if the correctness-centric Bluespec design approach is able to generate RTL that is comparable to handwritten Verilog.

In this work, we explore the design of a 2-way superscalar processor core with a centralized reorder buffer system implementing the MIPS I ISA. Performance will be measured by the achievable amount of "cycle-level parallelism" of the individual atomic actions within the design. While there will not be an explicit focus on clock frequency, we will only consider microarchitectures which reflect a reasonable hardware design.

1.1 Organization

Chapter 2 gives a review of Bluespec's syntax and semantics. Chapter 3 discusses the high-level abstract design of an out-of-order superscalar MIPS I processor. In Chapter 4, we discuss how to most naturally translate this abstract design into Bluespec. In Chapter 5, we discuss how to improve the compilation results to meet our performance goal. In Chapter 6 we generalize our work in Chapter 5 to form a methodology to effectively represent, debug, and tweak large Bluespec designs. Finally, we present our conclusions in Chapter 7.

Chapter 2

Bluespec

Bluespec is a hardware description language (HDL) which compiles into TRS. This intermediate TRS description can then be translated through a compiler into either in Verilog RTL or a cycle-accurate C-simulation.

In Bluespec, a module is the representation of a circuit in Bluespec. It is the object which is compiled into RTL. Each Bluespec module roughly corresponds to a Verilog module. A module consists of three elements: first, state such as registers, flip-flops, and memories; second, rules which modify that state; lastly are interfaces which provide a mechanism for interaction with the internal structure of the module.

2.1 Bluespec Syntax

In Bluespec there are two types of modules. The first, shown in Figure 2-1, is a standard module with state elements including other modules, rules, and interface methods. The second is a primitive module which is just a wrapper around an actual Verilog module.

State elements are all specified explicitly in a module. The behavior of a module is represented by its rules each of which consists of a state change on the hardware state of the module (an *action*) and the conditions required for the rule to be valid (a *predicate*). It is valid to execute (*fire*) a rule whenever its predicate is true. The syntax for a rule is:

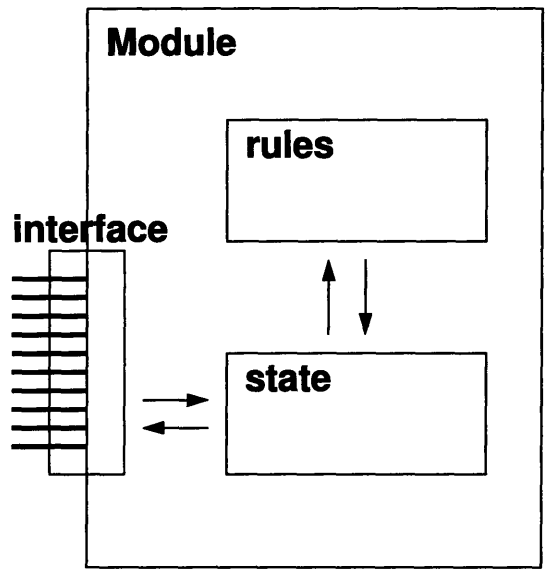

Figure 2-1: Diagram of a Standard Module (*courtesy of Bluespec Inc.*)

```
"RuleName":
 when predicate
  ==> action
```

The interface of a module is a set of methods through which the outside world interacts with the module. Each interface method has a predicate (a *guard*) which restricts when the method may be called. A method may either be a *read method* (a combinational lookup returning a value), an *action method*, or a combination of the two, an *actionvalue* method.

An actionvalue is used when we do not want a combinational lookup's result to be made available unless an appropriate action in the module also occurs. Consider a situation where we have a module consisting of a single FIFO and we want to provide a method which gives access to the head of the FIFO, and atomically causes the head value to be dequeued whenever it's used. Thus we would write the following, where `do` is used to signify an actionValue and `theFifo` is the FIFO instance.

```
getHeadOfFIFOmethod = do
```

```
theFifo.deq
return theFifo.first
```

The abstract model of execution of a Bluespec circuit is as follows. For any initial hardware state, we have some set of executable rules. Each cycle, we randomly select one of these rules and execute it, thereby changing the state.

This abstract model is of course very inefficient, so in the timing-dependant description we allow multiple rules to fire at once. To insure correctness we require that any transition from one state to another in the real description must be obtainable by a valid sequence of transitions (single rule firings) in the abstract system.

2.2 The Bluespec Compiler

The Bluespec compiler can translate Bluespec descriptions into either Verilog RTL or into a cycle-accurate C simulation (see Figure 2-2). It does this by first evaluating the high-level description of the design into a TRS description of rules and state. From this TRS description the compiler schedules the actions and transforms the design into a timing-aware hardware description. This task involves determining when rules can fire safely and concurrently, adding muxing logic to handle the sharing of state elements by rules, and finally applying boolean optimizations to simplify the design. From this timing-aware model, the compiler can then translate this into either a RTL or C implementation of the design.

2.2.1 Scheduling

We call the task of determining what subset of rules should fire on a cycle given its state and in what order should rules be fired in a single cycle, *scheduling*. Understanding how the Bluespec compiler schedules multiples rules for cycle-by-cycle execution is important for using Bluespec proficiently. Optimal selection of which

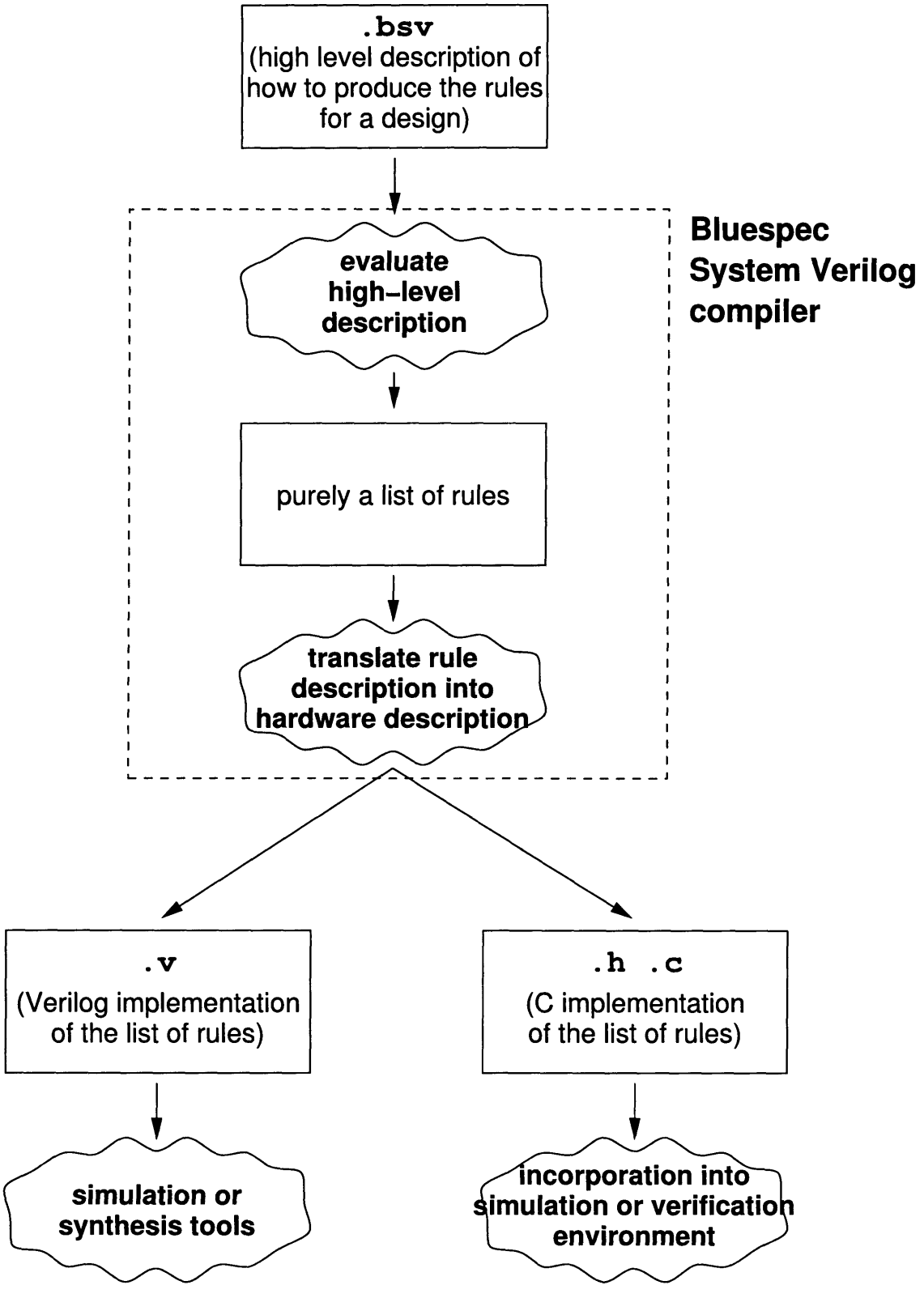

Figure 2-2: Diagram of Compiler Flow (*courtesy of Bluespec Inc.*)

subset of firable rules to fire in a single cycle is an NP-hard task, so the Bluespec compiler resorts to a quadratic time approximation.

Determining Rule Contents

Due to the complexity of determining when a rule will use an interface of a module, the Bluespec compiler assumes conservatively that an action will use any method that it could ever use. That is to say, if an action uses a method only when some condition is met, the scheduler will treat it as if were always using it. This leads the compiler to make to conservative estimations of method usage which in turn causes conservative firing conditions to be scheduled.

Determining Pair-wise Scheduling Conflicts

Once the components (methods and other actions) of all the actions have been determined, we find all possible conflicts between each atomic action pair. In the case that two rule predicates are provably disjoint, then we can say that there are no conflicts as they can never happen in the same cycle. Otherwise, the scheduling conflicts between them is exactly the set of scheduling conflicts between any pair of action components of each atomic action.

For example, consider rules "rule1" and "rule2" where rule1 reads some register r1 and rule2 writes it. Registers have the scheduling constraint "`_read` < `_write`", which means that calls to the `_read` method calls must happen before the `_write` method call in a single cycle. Thus this constraint is reflected in the constraints between rule1 and rule2 ("rule1 < rule2"). If rule1 were to also write some register r2 and rule2 were to read it we would have the additional constraint ("rule2 < rule1"). In this there is no consistent way of ordering the two rule, so we consider the rules conflicting with sequential ordering restrictions (as they will never happen together, it doesn't matter how they are ordered to happen concurrently).

Generating a Final Global Schedule

Once we have determined all the pair-wise conflicts between actions we create a total temporal ordering of the actions. To do this, the compiler orders the atomic actions by some metric of importance, which we shall call "urgency". It looks at each action in descending urgency order. When we look at an action, we place it to prevent the most conflicts with already ordered rules in the total ordering. Once its ordering has been determined, we say the rule can be fired in a cycle when both its predicate is met and there are no more urgent rule which conflict with it in that total ordering. Once the compiler has considered all atomic actions in turn, we have a complete schedule.

Chapter 3

High-level Processor Microarchitecture

We can view the out-of-order processor abstractly as the collection of units shown in Figure 3-1. Each of these abstract units map to a single Bluespec module. For a correct implementation, each Bluespec module must meet the associated abstract unit's requirements, which are listed below.

3.1 Modular Interfacing

The first basic and far-reaching consideration needed is how interaction between modules should be described in Bluespec. While the final hardware descriptions the compiler will generate will be essentially the same, the Bluespec representation for these different approaches are quite different. Thus the major consequence of this choice is the ease of description of our design.

The very first thing that occurs to us is that we could simply pass the interface of the receiving module into the sending module as shown below. Then a rule in the sending module could be invoked to call a receiving method in the receiving module. This a very natural to describe the interaction.

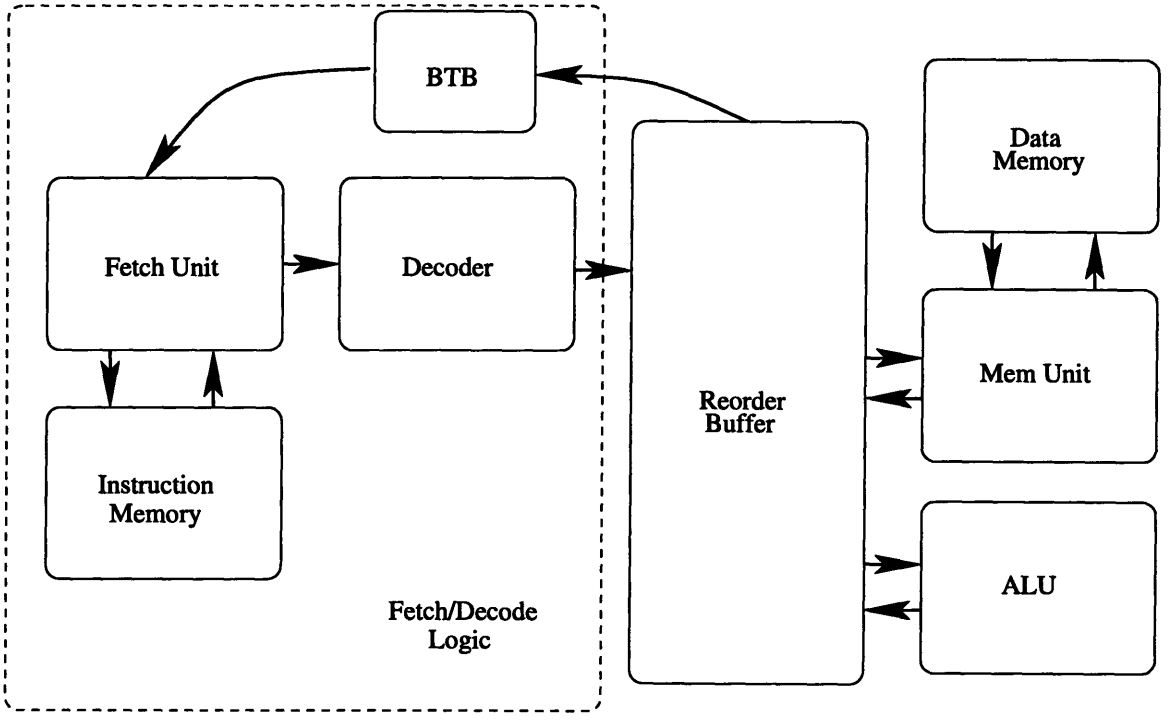

Figure 3-1: High Level Design of Processor

```
mkModuleSend receiving_module =
  module
    ...
    rules
      "send" when True ==>
       action
         receiving_module.receive(data)
```

This however has two problems. First, it is not possible for the sending module to be compiled modularly. This is because the implementation of the receiving module may change how the sending module is allowed to call it's interface methods. The second is that there is no way to send data in both directions. This is due to the fact that the Bluespec compiler forbids mutually recursive definitions, to simplify the compiler's job and to maintain better readability of code.

A better solution is to describe one module as a submodule of the other. In this organization, the outer module could then have rules to push data into the inner module to send data and rules to pull data out to receive it like shown in Figure 3-2.

This gives us the two-way communication, but not in a natural way.

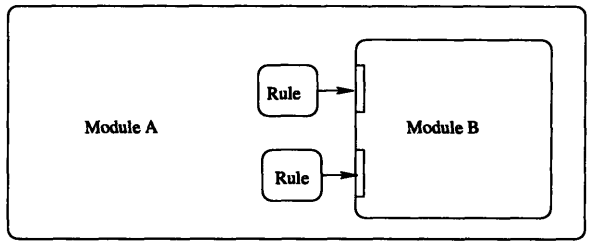

Figure 3-2: Submodule Organization

This organization also has problems. Although we can now compile each modularly, any time we change the submodule, we are forced to recompile the outer module. More problematic, the only modules which can access the interface of the submodule must be within the outer module. To make the methods available outside we must add a similar interface to the outer module to allow us to pass calls through to the submodule. This is both dissatisfying and unintuitive.

For our final solution we decided was to split each communication into two separate methods: an actionvalue method in the sender and a method which takes a value of the corresponding type in the receiving module. We could then make a wrapper module which contains a rule to call the two methods atomically in a natural way, as shown in Figure 3-3.

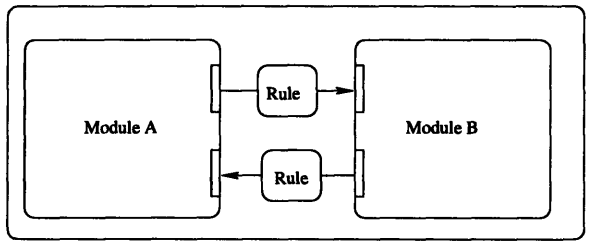

Figure 3-3: Separated Module Organization Style, Arrows Showing Data Flow

To better illustrate this consider the following example. `ModuleA` which passes `ModuleB` a 32-bit value (having type Int). Module B, then does some determines

whether the value it is given is odd and returns a boolean value (having type Bool) to ModuleA. Using the style the code would look something like:

```
mkModuleA =
  module

   <state and unrelated rules>

   interface
     -- method type: ActionValue (Int)
     sendToModuleB =
        actionvalue
           <action to remove value>
           return intValue

     -- method type: Bool -> Action
     getFromModuleB boolVal =
        action
          <action to handle bool>


mkModuleB =
  module

   (FIFO Bool) thefifo <- mkFIFO

   interface
     -- method type: ActionValue (Bool)
     sendToModuleA =
        actionvalue
           thefifo.deq
           return thefifo.first

     -- method type: Int -> Action
     getFromModuleA intVal =
        let
          isOddVal = isOdd intVal
        in
          action
            thefifo.enq isOddVal
```

This allows us to keep our module structure exactly as envisioned in the high-level design and keep our modules completely self-contained for modular compilation. It removes the restriction of having a single master-slave relationship between any module pair. Only the wrapper module and neither of the communicating modules need to be explicitly aware of the other when described. Because of this, we chose to use this style when describing the modules in our processor.

This organization has since been explicitly formalized into the Get-Put interfaces in the Bluespec Standard Library.

Now that we have determined how the modules interconnect, we must determine what each module is supposed to do. The rest of this chapter details the responsibilities of each module in the design.

3.2 Fetch

The Fetch Unit contains the Program Counter (PC) which it uses to make requests to the Instruction Memory. When it receives a response, the Fetch Unit passes the response on to the Decode Unit. The Fetch Unit determines which instruction to fetch, by consulting the Branch Table Buffer (BTB). The Fetch Unit contains an interface by which the Reorder Buffer(ROB) can notify it of the new PC whenever the ROB detects a branch misprediction. The Fetch Unit has an epoch register, which it uses to tag every instruction which it passes on to the Decode Unit. The epoch is a six-bit integer value which is incremented on every branch miss. The ROB ignores all incoming instructions whose epoch values do not match it's local current value as it implies that they are part of the mispredicted path.

3.3 Instruction Memory

The Instruction Memory takes requests from the Fetch Unit either during the same cycle or in later cycles. It responds with a list of instructions starting at the address requested. Responses must be returned in the same order as the requests were made.

3.4 Decode Unit

The Decode Unit takes 32-bit MIPS instructions from the Fetch Unit and decodes them. It then passes the results on to the ROB in the order it was received. The decode unit must maintain enough state to allow for some asynchronity between the fetch unit and the execution units.

3.5 ALU

The ALU must be able to take tagged instructions that are ready to execute from the ROB and execute those instructions. When the ALU returns a result, it must also send the associated tag of the instruction. No restrictions are placed on the ordering of the replies.

3.6 Memory Unit

The Memory Unit takes memory instructions (loads and stores) from the ROB with all operands resolved (the address index, the address offset, and the value). To simplify the complexity of the Memory Unit, we require that the memory instructions must be sent in program order. It is equally easy to express other more complex memory models in Bluespec. The Memory Unit makes any necessary memory accesses and returns the results to the ROB. Speculative stores must be kept until they are

either invalidated or committed via two methods accessible by the ROB. Since this is a uniprocessor model, we only have to enforce a relaxed memory model whereby memory instructions to the same address must occur in order.

3.7 Data Memory

The Data Memory is very similar to the Instruction Memory. It must handle requests from the Memory Unit and respond with the appropriate data on read requests. Requests must be handled sequentially. More specifically, a write request's results must be observed by all read requests which occur later in time.

3.8 Reorder Buffer

The Reorder Buffer (ROB) keeps track of the ordering of instructions it receives. It tracks data dependencies between instructions, and passes the instruction results values to instructions waiting for them. Whenever possible the ROB commits the oldest instructions that have been executed by writing the results back into the register file.

In this design, the ROB unit also contains the branch execution logic. On branch misses, it marks all the false path instructions as killed and increments the ROB's current epoch value. It also notifies the Fetch/Decode Logic of the correct program counter and the new epoch. Subsequent instructions which do not have the correct epoch will be thrown away when they are put into the ROB.

We make the assumption that responses from functional units may not occur in the same cycle as a request to the functional unit (i.e. there are no purely combinational functional units). There are no timing requirements placed on the design of the Fetch/Decode Logic by the ROB[5].

Chapter 4

Initial Implementation

In this section we discuss the initial design which matches the MIPS ISA. This design reflects what we thought to be a "natural" description of the design. The design described here matches the abstract requirements but does not achieve the optimal cycle-level parallelism. Most of the complexity in this design is found in the ROB module. Because of this we will cover the ROB is much more depth.

Design improvements will be made in Chapter 5. As the design was clearly split into definite units, and the interactions were explicitly determined in Section 3.1, we need only look at each module's implementation in isolation.

4.1 Satellite Modules

This section covers all the modules except for the ROB which is discussed in Section 4.2.

4.1.1 Instruction Memory

The instruction memory is implemented as a simple one-level hierarchy. On receiving an address, the memory enqueues the four-instruction block starting at that address,

which it then returns the next cycle. There is no additional circuitry to check for repetitions or overlaps of sent requests, as the Fetch Unit is expected to handle any such optimizations.

4.1.2 Branch Table Buffer Unit

The Branch Table Buffer Unit (BTB) consists of a table of a direct-mapped cache of 8 address to next-address mappings. On a lookup from the outside world for the next PC value, the BTB checks if the value is in its cache. If so, it combinationally returns the recorded value. If not, it assumes the branch is not taken and returns the next instruction's address (PC + 4). On an update by the ROB, the table entry associated with the update's instruction address is updated in the BTB's cache.

4.1.3 Fetch Unit

The Fetch Unit state consists mainly of a PC, a nextPC, and an epoch register. The PC value is used to as the request to the Instruction Memory. On the resulting response from the Instruction Memory, the BTB Unit is consulted. If it returns that the nextPC instruction is a taken branch, the PC is replaced with the nextPC value, the nextPC value is replaced by the prediction determined by the BTB Unit, and we send only the first instruction on to the Decoder. If it is determined not to be a taken branch, we change the PC to the BTB's address prediction (PC + 8), nextPC to 4 more than that, and we send the first two instructions to Decode.

All instructions sent from the Fetch Unit are marked with the epoch value. This value serves to tell the ROB which of the instructions it receives from the Fetch/Decode logic are valid. On a branch misprediction notification, the Fetch Unit increments its epoch value, and changes the PC and nextPC to the appropriate values. It then marks all the following instructions with the new epoch value. Thus the Reorder Buffer can tell when it starts receiving instructions from the correct path.

4.1.4 Decode Unit

On an insert from the Fetch Unit, the Decode Unit decodes the given instruction(s) and enqueues them into a 2-way FIFO. The interface to the ROB, may then extract one or two instructions per cycle as needed.

4.1.5 ALU Unit

The ALU takes a single cycle for all instructions. On a request it computes the result and enqueues it into a FIFO. The result is then dequeued from the FIFO when the result is taken. All requests are marked with a tag. The ALU returns the same tag as it received with each instruction it is given.

4.1.6 Data Memory

The Data Memory is a two-way cache which handles word-aligned addresses. As part of each write request is a 4-bit mask to signify which bytes in the word are to be written.

4.1.7 Memory Unit

The Memory Unit receives all memory instructions in program order from the ROB. When received, store instructions are saved in a store buffer capable of holding 4 requests, and a response containing any errors is returned to the ROB. Loads are placed into a LoadRequest buffer. When a load is able to be fired (after all previous stores have been handled), a request is made to memory. On the response from memory, the result is enqueued to be sent back to the reorder buffer. Store instructions are removed from the store buffer on commits and invalidations from the ROB. In the case of commits, they are then sent to the Data Memory. In the case of invalidations (due to the instruction being on a false path), all pending stores found to be invalid

are discarded.

4.1.8 Register File

The register file has 2 write ports and 4 read ports. R0 is hardwired to zero. In addition to the 32 general purpose registers, there are two special registers (HI and LO) used by 64-bit instructions (multiplies and divisions). These can be accessed through separate read and write interfaces.

4.2 Reorder Buffer

The Reorder Buffer is by far the most complicated of the modules. Its performance is pivotal to the performance of the design as a whole. As such, we shall invest a great deal more time in describing its implementation than the other blocks

4.2.1 Storage

Instructions are kept in an ordered list of N "slots." each slot contains an instruction and the associated values required for its execution, as well as the operand values, the result, and some internal state to described the state of the slot and possible instruction held in it. We use a headTag and a tailTag pointer to represent the oldest slot used and the next slot in which an incoming instruction will be placed respectively. To differentiate having the circular slot buffer being full and being empty we assert that at least one slot must always remain empty. Below is the Bluespec description of a slot.

```
struct Slot =
 tag      :: ROBTag          -- the Slot's tag
 state    :: Reg State
 ia       :: Reg IA
```

```
insType  :: Reg InstrType
opcode   :: Reg (Bit osz)  -- opcode size
tv1      :: Reg TagOrValue -- operand 1
tv2      :: Reg TagOrValue -- operand 2
imm      :: Reg Imm        -- immediate field
dval     :: Reg Value      -- result
destReg  :: Reg RegOrHiLo
predIa   :: Reg PredIA     -- for branches
```

Each slot consists of a number of registers, as shown above, which represent an instruction template: the instruction address (IA), the predicted instruction address (predIA), the slot's state, and two operand registers tv1 and tv2 that store either the tag of the slot generating the value or the actual value of the operand. We could have represented each slot as a single register, but by using a multiple register design, we help the compiler partition the data and generate better schedules.

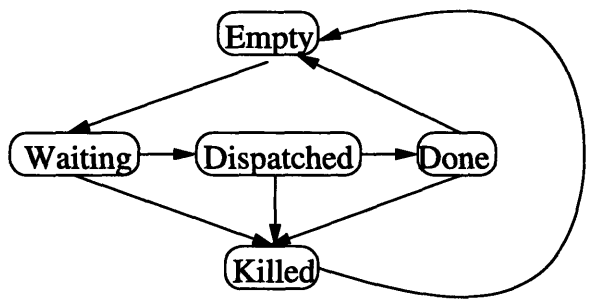

Figure 4-1: State Transitions of a Slot in the Reorder Buffer

The state of a slot is either Empty, Waiting, Dispatched, Killed or Done. The state transition diagram is shown in Figure 4-1. Empty signifies that the slot has no instruction in it. Empty instructions only exist within the region that the headTag and tailTag denote as non-active. When an instruction is inserted into a slot, it enters the Waiting state where it will wait for its operand values to be resolved into actual values. After both operand values have been resolved, the instruction in that slot can be dispatched. This consists of changing the slot's state value to Dispatched and sending the instruction to the appropriate functional unit. When the result is sent

back to the ROB and written into the slot, the slot enters the Done state. At this point it can be committed and made Empty again. At any time, the branch resolution rule can set the state of non-empty slots on a mispredicted path to Killed.

Instructions are committed in the ROB in the order they were inserted. To complete an instruction one increments the headTag and writes the associated slot's state register as Empty. To insert an instruction, the ROB increments the tailTag, places the instruction into the slot at which the tailTag pointed.

4.2.2 Lookup Organization

We decided to keep track of the speculative state of the register file via a combinational lookup through the slots. This could also be done with an additional structure which kept the speculative value or tag reference of each register. The state would get copied during branch instructions and restored if the branch was mispredicted. This however is significantly more complicated to implement and so the simpler method was chosen.

4.2.3 Design Complications for the Reorder Buffer

To match the MIPS I ISA we need to add a few additional complications to our design.

First, the MIPS I ISA has a branch delay slot. When a branch instruction is killed, we must not invalidate the instruction directly after it. If we resolve the branch before this instruction has been inserted, the delay slot instruction will have the wrong epoch. To prevent this from happening, we assert that branch instructions cannot be resolved until the next instruction has been inserted into the ROB.

Secondly, some instructions generate 64-bit results (i.e. multiply and divide instructions). To keep from having to double the size of the result field in each slot, we place these instructions into two consecutive slots with the high order bits in the

first slot, and the low order bits in the second. The slots will then be treated as an atomic unit until the slots are committed.

4.2.4 The ROB Module

Below is a stylized Bluespec description of our initial design of the ROB. The sz value is a integer which the ROB is passed at instantiation. It represents the number of slots in the ROB. We can change this number to any value larger than 2 and maintain correctness.

```
mkROB :: Interger -> Module ROB
mkROB sz = -- sz is # of slots
  module
   let
    minTag = 0
    maxTag = sz

    --auxiliary functions
    --(e.g. mkSlot & incrTag)

    -- state elements
    rf            :: RegFile           <- mkRegFile

    curEpoch      :: Reg Epoch         <- mkReg 0

    headTag       :: Reg ROBTag        <- mkReg minTag
    tailTag       :: Reg ROBTag        <- mkReg minTag

    handlemissReg :: Reg (IA,PC,Epoch) <- mkReg (0,0)

    slotList      :: List Slot         <- mapM (mkSlot)
                                             (upto minTag maxTag)

    rules
       <rules>

    interface
       enqueueInst inst     = ...
       getALUInstr          = ...
```

```
getMEMInstr          = ...
updateALU tag result = ...
updateMEM tag result = ...
missvalues           = ...
```

4.2.5 Per Slot Rules

Many of the actions which the ROB needs to be perform occur on a per slot basis. These actions are best represented as a set of rules, each of which operate on exactly one of the slots.

It may initially appear that generating these rules for each slot can be quite difficult and restrictive, but due to Bluespec's good static elaboration, this task can be easily done. We do this by writing a function to generate rules for a single given slot as shown below. Then we can map this function, in the same way we would in a standard functional language, over the list of all the slots producing a list of rules for those slots. We can then add this rule list to our current list of rules for the ROB. This also gives us the additional benefit of not limiting the number of slots in the ROB when we make a design.

```
let
  mkRules i = -- makes a slot's rules
    rules
      <rules>
 in
   mapM mkRules (upto minTag maxTag)
```

The specific rules of this type are described below.

Operand Update Rules

There are two separate rules per slot which update the tagged values with the actual values. They look as follows:

```
"update TagOrValue 1":
 when
     (T tag) <- slotJ.tv1
  ==> let
         slotTag = (getSlot tag)
       in
         action
           if (slotTag.state == Done) then
              slotJ.tv1 :=(V slotTag.dval)
            else
              noAction
```

The above rule checks to see if the instruction in some slot, slotJ, associated with the given tag has been executed and if so, it writes the value into the operand register.

Rules to Dispatch to Functional Units

Additionally, for each slot there is a slot dispatch rule per functional unit which takes waiting instructions and places them into the FIFOs which will dispatch to the appropriate functional units. The tag value that is enqueued along with the data sent is the unique number associated with the particular slot.

```
"Dispatch to ALU":
 when (slotJ.state == Waiting),
      (V v1) <- slotJ.tv1,
      (V v2) <- slotJ.tv2,
      (ALUTYPE == slotJ.instType)
  ==> let
         aluInst = (aluInstfromSlot slotJ)
       in
         action
            slotJ.state := Dispatched
            fifo2ALU.enq aluInst
```

4.2.6 Additional Rules

Besides the rules defined for each slot, there are a number of other rules in the ROB. These are described below.

Branch Execution Rule

Branch instructions are executed by checking the result, killing all instructions after the branch, and passing the values of the new PC and epoch value to the fetch unit. These killed instructions are left in the list to be removed by the commit rule. This means that the tailTag is not modified on a branch miss.

```
"Resolve Branch":
 when canFireBranch
  ==> let
        inst = fifo2branch.first
        correctIA = (calcNewIA inst)
        slotJ = (getSlot inst.tag)
      in
        fifo2branch.deq
        slotJ.state := Done

        if (correctIA /= inst.predIA) then
           action
             -- Send information on branchmiss
             handlemissReg := (correctIA,inst.IA,nextEpoch)
             curEpoch      := (nextEpoch)
         else
           noAction
```

Commit Rule

Commits are done by removing the oldest instruction from the slot list and writing back any results to the register file for any instruction which weren't killed.

```
"Commit":
 when headTag /= tailTag,
      slotJ <- (getSlot headTag),
      slotJ.state == Done,
      not slotJ.err
  ==> action
        headTag := (incrTag headTag)
        slotJ.state := Empty
        (rf.write slotJ.destReg slotJ.dval)
```

4.2.7 Interface Methods

The methods on the ROB correspond exactly to the appropriate Get or Put half of each communication channel it needs.

EnqueueInst Method

The enqueueInst interface does two combinational lookups to see if the two operands were generated by another instruction in the ROB. It writes either the tag of the associated slot, or the value from the register file as appropriate into the operand registers. It also marks the slot as waiting to be dispatched (i.e. the state is Waiting).

```
enqueueInst inst =
   let
      --slot to write into
      slotJ = getSlot tailTag

      --structure with values to write
      slotVals = (getSlotValues inst)
   in
     action
        tailTag := incrTag tailTag
        writeSlot SlotJ slotVals
  when (not slotListFull)
```

Methods to get Dispatched Instructions

The interface to get the instruction from the ROB and hand it to a functional unit consists solely of a dequeue from the associated FIFO.

```
-- type: ActionValue -> ALUInstr
getALUInstr = do
                fifo2ALU.deq
                return fifo2ALU.first
```

Branch Miss Information Interface

The interface to get the new branch information which is sent to the Fetch Unit just returns the value associated in the handlemissReg register. This register is set by the branch execute rule.

```
missvalues = handlemissReg
```

Functional Unit Result Writeback Methods

Writebacks from the functional units write into the appropriate slot. This slot is determined by the tag which is sent with the result.

```
updateMEM tag result =
   let
      slotJ = (getSlot tag)
    in
      action
         slotJ.state := Done
         slotJ.err   := result.err
         slotJ.dval  := result.value
```

Chapter 5

Design Refinements

This section will discuss the problems inherent in the initial design and their sources. We will then describe possible solutions, whether through the design modifications or through changes to the compiler, and detail any problems that still require further consideration.

5.1 Separating Rules to Simplify Organization

A number of rules may read and write any of the slots in the ROB. However, during a given cycle they only operate on a small subset of the slots. As the compiler will consider these rules to always access all of the slots' state, it finds a number of false conflicts which causes inefficiencies in the scheduling.

A prime example of this is found in the commit logic of the ROB. A 2-way commit action on all slots will only operate on a pair of consecutive slots. In our initial implementation we use a single rule to do all of the commits. The scheduler then infers that this one rule changes all of the slots, and causes the commit rule to conflict with actions which affect other slots. This is clearly not correct.

If we break the rule into a set of smaller rules as shown below, where each rule in the set handles a different subset of slots, it becomes clear to the compiler that

committing only ever operates on two consecutive slots at any one time, removing these false conflicts.

```
-- function to generate one particular sub-rule

mkSlotRuleCommit :: Integer -> Rules
mkSlotRuleCommit j =
       let
           slotJ = (List.select slots j)
           slotJ_plus_1 = (List.select slots incr(j))
           (slotJ,slotJP0) = (List.select slotPairList j)
           jb = fromInteger j  -- The tag j represented in bits
      in
        "commit_subrule":
         when True
          ==> action
                <do_commit_action on slot and slot_plus_1>


...

-- add ALL of the commit sub-rules to the ROB rule list

addRules (List.joinRules (List.map mkSlotRuleCommit
                                    (upto 0 (numSlots - 1))))
```

In the case of interface methods this strategy becomes more complicated. Because we cannot split up a method into many different methods (as it would change the interface), we must instead change the method into a dummy action which is responsible solely for transporting the methods operand data (input) into the module. We do this by writing the associated data into a state unit like a register or FIFO. Once the data has been placed into the state element, it is globally accessible in the module and we can split the interface method's original action into multiple rules as we did with rules.

While this solves the problem of conflicts it does add an extra cycle of latency

between when the outside world called the method and when the module executed it. To fix this we replace the state element with an "RWire", which is described in Section 5.4.

5.2 Inter-Module Latency

The natural way of organizing hardware to meet the Get-Put style interface, is to have a FIFO on the output path of the provider of the information. This can be seen in Figure 5-1. When results are ready, they are enqueued into the FIFO. Then, when the interface method is called they are taken from the head of the FIFO.

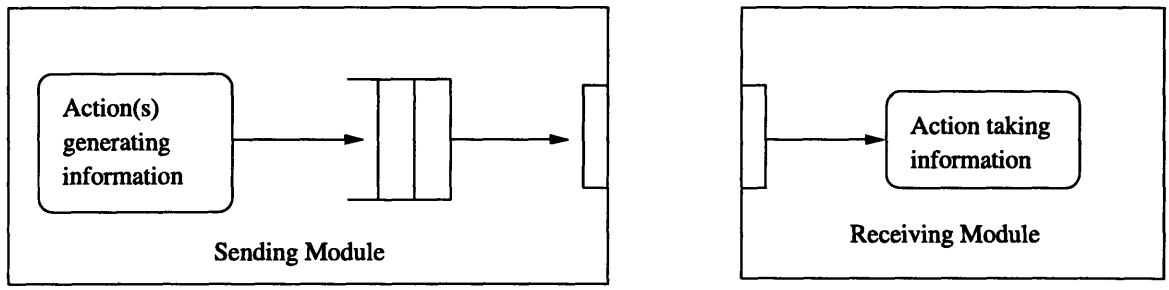

Figure 5-1: FIFO-based Get-Put Style Communication Channel

This methodology works very well when there is only one source of information. For example, the Decode unit provides the ROB with all the instructions that it needs to process at once, so when the decoder receives undecoded instructions from the Fetch Unit, it can enqueue them into its output FIFO on the same cycle. This means that an instruction will be ready to leave the decoder the cycle after it was placed into the decoder.

However, in the case of dispatching instructions from the ROB, a FIFO does not offer the appropriate latency. When an instruction in a slot is ready to execute it must first be enqueued into the FIFO going to the appropriate FU (as shown in Figure 5-2). This adds an extra cycle to dispatch the instruction. As this is on the critical

path, we want to remove this extra cycle. The natural solution is to merge all of the dispatch rules to a functional unit and the method together into one atomic rule, as in Figure 5-3. This rule will search for an appropriate instruction, mark it as sent, and send the value. This allows us to avoid using a FIFO entirely.

This solves the cycle latency issue but causes a new problem. The new method now appears to affect each of the slots, which as was discussed in Section 5 causes inefficient scheduling. This will be resolved in the next section.

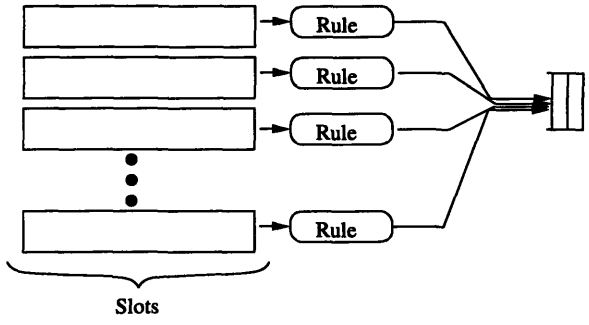

Figure 5-2: Initial design with an extra cycle of latency

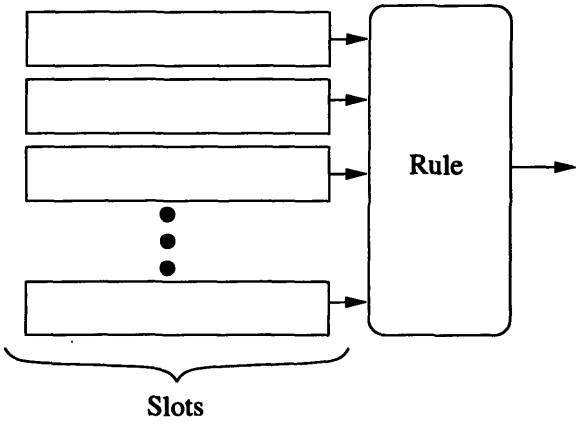

Figure 5-3: Final design which removes latency

5.3 Improving compilation with Disjointness Information

Some rules will never fire together, but the compiler still spends extensive time to to see if they conflict. This can be quite expensive computationally if the action predicates cannot be easily observed to be disjoint. A instance of this in the ROB can be found between the rule which inserts an instruction into a slot and the rule which updates an operand value for an instruction in the same slot. The former requires the slot state be "Empty" and the later requires the slot state be "Waiting". As such, even though they may both write to the same exact state, they do not have any scheduling conflicts. By stating explicitly in the predicate of the insert rule that it only operates on empty slots the compiler can determine easily that it is mutually exclusive with a number of other rules. This optimization causes a substantial reduction in compile time.

5.4 Removing False Conflicts from High-Level Information

In our initial design, there are a few situations where the compiler is not intelligent enough to determine that two rules do not conflict. The best example of this is in the state register in the slots of the ROB. At a high level, it is clear that we cannot do any of grouping of rules larger than one on a particular slot at once (insert, dispatch, writeback, commit). However, these actions can fire concurrently, and so the compiler must verify that the instructions do not write to the same data at once. This requires more high-level reasoning than is reasonable for current compiler technology.

One solution is to explicitly disambiguate the writes on conflicting registers. To do this we will consider a hypothetical register with multiple non-conflicting write ports.

These ports are totally ordered so that the port with highest priority (the lowest numbered port) which does a write will be observed. All others will be ignored. With this new register we can solve the problem by having each rule write into a different port wherever there is a conflict.

The question then is how to allow this to happen in Bluespec. With the atomic action mindset, each rule or method happens in isolation and so it cannot tailor its actions based on what others are doing. In the case of this register we would like each port's action to be known by a central rule which would take this global knowledge of the writes occurring and choose to do the appropriate write.

We achieve this by using RWires. An RWire is similar to a Verilog wire, but with an exposed write enable bit (reads can see if a write is being performed that cycle). Alternatively it can be viewed as a register where writes are performed before reads in a cycle, with no ability to save values, and with a validation bit associated with data. RWires allow us to have a global name space as in standard RTL. This in turn lets us accomplish some optimizations that previously were not available. It is worthy of note that RWires are "unsafe" in that they allow the introduction of timing dependencies into our model and therefore must be used with caution.

This multi-ported register consists of a series of methods which all writes into its own RWire. There is also a rule which reads the values from the RWires and determines which value (if any exist) to write into the register.

With this new multi-ported register we are now able to explicitly tell the compiler how to solve a large number of its write conflicts by placing conflicting rules writing different ports with the appropriate relative priorities.

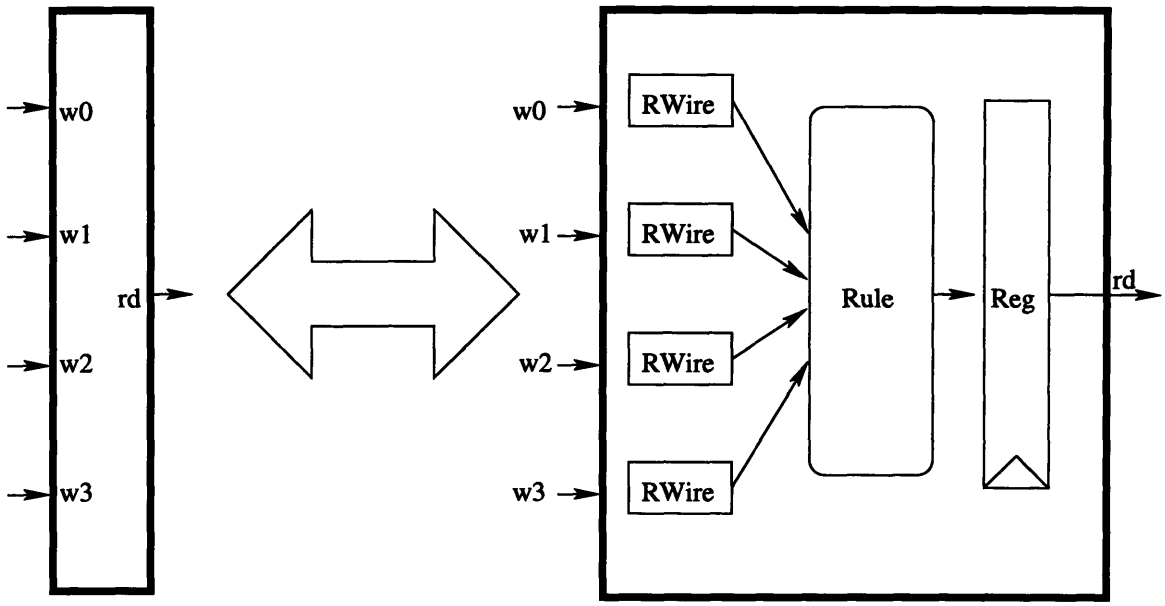

Figure 5-4: Multi-Ported Register(MPReg) as seen from outside and internally

5.5 Removing Redundant Data Reads

With the methods described previously, almost all of the actions we expect to happen concurrently no longer have conflicts. However, one important case remaining is the conflict between inserting a value into the reorder buffer and removing one. The reason for this is read-write sequencing. When we insert a value into the ROB, we check the tail and head pointers to verify that there is space and then we update the tail pointer. When we commit we check whether the oldest instructions are done, mark the slots as empty and then update the head pointer. Each rule is reading a the value of state which the other changes. As the Bluespec compiler will not implicitly, forward values from one rule to another combinationally, it cannot find a consistent ordering.

However this is simply a problem of representation. We know that any slot marked Empty is free, so we can insert an instruction regardless of what the head pointer is. Similarly, we know that any instruction marked Done cannot be Empty, and so we do not need to check the tail pointer when we commit.

While this allows us to remove one conflict, a sequential ordering conflict remains. This is discussed in Section 5.8.

5.6 Cleaning up Writeback Muxing

Our ROB must handle writebacks from each of the functional units concurrently to be considered a realistic model of a processor. With the method described in Section 5.4, we were able to circumvent this problem with a multi-ported register with two ports. However, unlike the case of the state register, there is only ever one writer which writes to each port with a certain number. With this high-level knowledge, we can simplify our model, as seen in Figures 5-5 and 5-6, to merge all corresponding RWires into one RWire per writing method. While this does not change the generated hardware, it helps simplify the user's and compiler's views of the ROB.

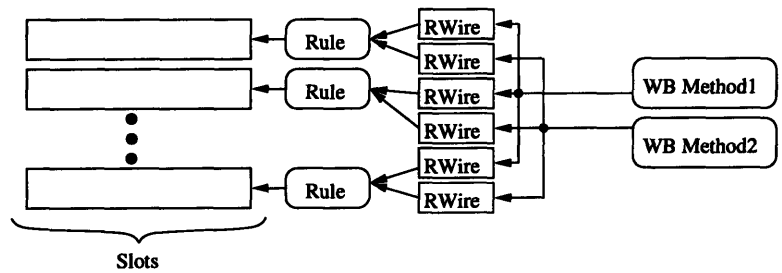

Figure 5-5: Initial Rule Organization for Writebacks

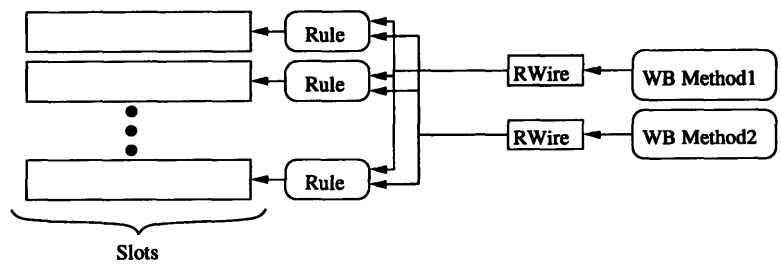

Figure 5-6: Final Organization

5.7 Reducing Branch Misprediction Penalty

When the ROB is sending branch update information to the Fetch Unit on a branch misprediction we have an extra cycle of latency between when the information is calculated in the ROB and the cycle that the ROB notifies the Fetch Unit and BTB. Using the method described in Section 5.2 we could fold the branch-execute rule into the branch update methods. Unfortunately, the current implementation has the update operation split between two methods to represent passing the data to both the BTB unit and the Fetch Unit. We could combine these two methods into one value method, and split it up outside of the reorder buffer, but this makes the method description less understandable. To avoid this we look at other ways of reducing this latency.

A solution inspired by Section 5.4 is to insert an RWire in between the methods and the branch-execute rule to provide the combinational path. However, because the consumer of the information is externally controlled, the ROB cannot know that the method will be fired whenever the information is created. Ultimately, we need a FIFO which allows for concurrent `enq` sequenced before `deq` on the same clock cycle. This way we can still have a combinational path, but will not suffer from data loss if other modules are not ready to take the information when it is presented. The design of such a FIFO is shown in Figure 5-7.

Once we have verified that the other units will always be able to take values that the reorder buffer generates, we can replace these bypassing FIFOs with a RWire to reduce the hardware generated.

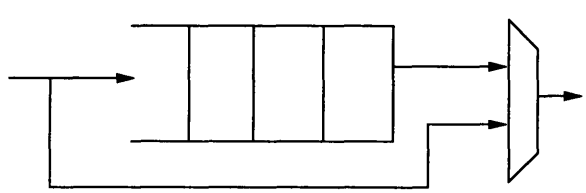

Figure 5-7: Design of Bypassing FIFO

5.8 Removing Conflicts with Special Registers

After completing the above improvements to the design, we only achieve fifty percent cycle-time throughput. The inefficiency lies with the insert and commit rules. The insert rule reads from the register file while the commit rule writes into the register file. Therefore the scheduling constraints on a register force all reads to occur before writes. Thus, this use of the register file requires the insert rule to occur logically first in a single cycle. However, the commit rule reads from the slot which the insert rule writes to, which means that the reverse ordering must happen in a cycle. This cyclic dependency causes a conflict.

At a high level, this is not a true conflict, as the values being stored into the register file are duplicated in the slots, so the insert rule will always get the correct answer when it searches through the slots, no matter when the commit happens in relation to it. We would like to solve this by just telling the compiler to completely ignore this conflict.

We do this by replacing all the registers in the register file with ConfigRegs, a special register with no sequencing requirement between read and write. This removes one of the sequential orderings, specifically the one forcing the insert to happen before the commit. With this requirement gone, the rules can be scheduled to fire concurrently.

While this works, it is somewhat unappealing as we have lost the safety associated with having the sequential conflicts involving those registers. A better solution is to replace the portions of the slot being read by the commit rule with registers in which writes happen before reads (in Bluespec, a BypassReg). This exactly represents a hardware latch. This reverses one of the sequential orderings and allows rules to be composed sequentially, without us having to override the compiler's correctness checking.

5.9 Performance Results

After the design improvements described in the previous sections, our two-way superscalar design was able to achieve a maximum IPC of 0.5 with a branch misprediction penalty of 3 cycles.

Though most of the more than many-hundred-fold improvement in compilation was due to improvements in the compiler's implementation of scheduling algorithms, a factor of ten improvement was due entirely to organizational improvements and conflict resolutions in the design.

Chapter 6

General Methodology for Large-Scale Design

In the process of designing this out-of-order processor, we were able to find a number of conventions and strategies which greatly improved debugging and organization of large designs. This chapter discusses these insights.

6.1 Improving Debugging

As with any complicated system, showing that the design conforms correctly to your high-level specification is difficult. The debugging methodology that hardware designers use for Verilog does not work for Bluespec. The following new debugging methods have shown themselves to be immensely helpful in finding problems with our processor design.

6.1.1 Saving Intermediate Values Via RWires

When simulating Verilog, the designer is able to probe any value in his specification. However, since Bluespec adds its own hidden signals and optimizations, intermediate

values can be lost after being passed through the compiler.

Often when debugging, these intermediate values are extremely useful, so it would be beneficial if there was a way to force the compiler to keep the value. Our solution is to write debugging information into RWires. With this one could probe the output port of the wire and get the value. Using this approach also provides a validation bit which tells us if the value we are looking at is currently valid.

It should be noted that the compiler's `-inline-RWire` flag cannot be used in conjunction with this, as it will remove all RWires from the design.

6.1.2 Retaining Accurate Timing while Displaying Values

Displaying information to the screen is extremely useful for debugging. However, displaying information from registers causes implicit reads. This may cause different scheduling decisions by the compiler for designs that with displays than those made for designs without them. Consider the following rules:

```
"Rule_1":
 when True
  ==> action
        if (b >= 5)
           regA := b*2
         else
           regB := regB + 1

"Rule_2":
 when True
  ==> action
        if (c >= 4)
           regA := regA + 1;
         else
           regC := regC + 1
```

Currently these rules do not conflict, as they can be fired in the same cycle with `Rule_2` sequenced before `Rule_1`. However, if we add a display statement for debug-

ging as follows:

```
"Rule_1":
 when True
  ==> action
        $display "rule1: regA: %d regB: %d " regA regB
        if (b >= 5)
           regA := b*2
         else
           regB := regB + 1

"Rule_2":
 when True
  ==> action
        $display "rule2: regA: %d regC: %d" regA regC
        if (c >= 4)
           regA := regA + 1;
         else
           regC := regC + 1
```

Then both rules now both read and write **regA**, therefore they cannot be scheduled in the same cycle like originally described.

To fix this problem we must separate the display statements from the associated rules. But we still only want to display register values when the associated rule is fired. We accomplish this with careful use of RWires. Instead of displaying the data,we write to a special RWire which has a zero-bit data. This signals when the display should happen. Then we create a special rule which handles the register value display. The above example turns into:

```
"Rule_1":
 when True
  ==> action
        rule1_display_RWire.wset ()
        if (b >= 5)
           regA := b*2
         else
           regB := regB + 1
```

```
"Rule_1_display":
 when Just () <- rule1_display_RWire.wget
  ==> action
        $display "rule1: regA: %d regB: %d " regA regB

"Rule_2":
 when True
  ==> action
        rule2_display_RWire.wset ()
        if (c >= 4)
           regA := regA + 1;
         else
           regC := regC + 1

"Rule_2_display":
 when Just () <- rule2_display_RWire.wget
  ==> action
        $display "rule1: regA: %d regC: %d " regA regC
```

This enforces that the register value displays will never cause our original rules to fire as predicted before. However, the displays themselves may not fire because of other dependencies. We can discover which of these may not fire as expected by use of a `fire-when-enabled` pragma when compiling. Some reasonably simple scheduling analysis will show why displays do not fire, but there is currently no mechanical way to guarantee displays will always fire without causing scheduling changes.

6.2 Determining When Rules Fire

Sometimes it is useful to know when rules are ready to be fired and when they are actually fired in a simulation. The `-keep-fires` flag forces the compiler to preserve the CAN_FIRE and WILL_FIRE signals for each rule. These signify when a rule's implicit conditions are met and whether the scheduler decided to execute the rule respectively. With these, a user can determine whether or not a rule is firing as expected.

6.3 Improving Compile Time

As designs get bigger, compilation time of Bluespec increases hyper-linearly. To help mitigate this effect, we have worked out a number of strategies which are effective at bringing compilation times under control.

6.3.1 Modular Organization

One of the biggest issues with any large design is how to divide it into smaller, more manageable parts. Modules and there interfaces provide a clean abstraction boundary to help designer's accomplish this. However, we encounter a problem arises when we wish to compile a many-module Bluespec design. In many naïve design organizations, the designer has neglected to make hard module boundaries between intercommunicating modules, choosing instead to pass interfaces into the module instantiations. As such, the design is not able to compile those modules separately. The monolithic design compilation which must then take place is unnecessarily long.

To properly maintain a strong division between different modules in a design, a designer should use the Get-Put methodology described in Section 3.1. This allows both faster compilation as well as the ability to selectively recompile a subset of the blocks in a design.

6.3.2 Mutual Exclusion Inclusion

Some of the information that the compiler spends a significant time trying to determine is whether any particular pair of rules is mutually exclusive. Although for almost all of the rule pairs in a design this information can be found almost instantaneously, the compiler has to brute force at the problem for a long time to determine mutual exclusivity for certain pairs.

The designer can simplify the compiler's task by making the information more ob-

vious. This can be achieved by changing the representation to simplify the compiler's task. Alternatively, the designer could add a scheduling pragma which explicitly notes the correct conflict relationship between the rules. The former takes more thought, but is guaranteed be maintain Bluespec's safety properties. The latter easily sidesteps a lot of computation but shifts the weight of correctness of your ordering assertion to the designer and should therefore be used sparingly.

6.3.3 Removing Unnecessary Conflicts

Some modules have actions which would operate correctly even if receiving a stale register value. A standard example of this occurs in register files, where reads and writes often have no ordering associated between each other as the designer has already explicitly worked around stale read values in the design. In this case, the compiler needlessly forces the read methods to occur before any write methods. To make the compiler ignore this unnecessary sequencing, we replace the associated registers with ConfigRegs, which are standard registers but without any temporal relationship between reads and writes in a cycle.

6.3.4 No-inlining Function Calls

Often there are large pieces of logic which we would like to duplicate multiple times. An ideal example is the circuitry which decodes a single instruction in a superscalar design. The logic block can be repeated multiple times without any changes. The most natural way to express these is via function application. Unfortunately, when the compiler tries to compile a module, by default, it expands out each instance of the logic and optimizes each separately. Clearly, we would like the compiler to optimize only the logic block only once. Therefore we attach a `no-inline` pragma to the function. This causes the compiler to compile the function as a separate module and then instantiate that module within the design. This prevents the compiler from

having to evaluate the function more than once.

6.4 Improving Scheduling

When designing in Bluespec, the designer should have a good understanding of the concurrency he expects from the design. However, the compiler must ultimately be able to find the concurrency in the design for it to exploited. While this helps the designer catch bugs in his design, it may also cause false restrictions to concurrency to occur. This section discusses methods which were found to be effective at getting to the heart of such scheduling problems.

6.4.1 Determining Conflicts From Compiler Output

The first step to handling a scheduling issue is to gain a good understanding of what exactly is causing the problem. To do this we make use of a number of debugging flags provided by the compiler.

First, we want to see a quick and dirty description of the conflicts which the compiler determined. We do this by using the dump schedule flag, `-dschedule`. This shows us the atomic actions in urgency order followed by the actions which will prevent the rule from firing. So the following output:

```
parallel: [esposito: [RL_Rule1 -> []],
           esposito: [RL_Rule2 -> []],
           esposito: [RL_Rule3 -> [RL_Rule1, RL_Rule2]],
           esposito: [RL_Rule4 -> [RL_Rule2]]]
```

says the rules are looked at in order `Rule1`, `Rule2`, `Rule3`, and finally `Rule4`. `Rule1` and `Rule2` will always fire when enabled and `Rule3` will not fire when `Rule1` or `Rule2` are to be fired. `Rule4` will not fire with `Rule2` is fired.

From this we can see where rules are being prevented from firing. If a rule is preventing another rule from firing and they should be able to fire concurrently, then those rules' interaction represents concurrency the compiler cannot find in the design. This line of reasoning will not pick out all possible conflicts because when rules which can be shown to be mutually exclusive do not appear in this form. Having unwanted concurrency loss there is much less likely as the designer writing the action predicates is likely to notice if they are mutually exclusive.

To see mutual exclusion data you must look at the more verbose `-show-schedule` which also contains all the same information as `-dschedule` but in a more verbose form.

Once we determine which conflicts we are interested in, we can get a closer look at the scheduling interaction between rules with the `-show-rule-rel` flag. This flag takes as operands one wished to see the conflict analysis between.

```
Scheduling info for rules "ruleA" and "ruleB":
predicates are not disjoint
    <>
    conflict:
    calls to
      RWire1.wget vs. RWire1.wset
      reg1.get vs. reg1.set
    <
    conflict:
    calls to
      RWire1.wget vs. RWire1.wset
    no resource conflict
    no cycle conflict
    no <+ conflict
```

In the above example, the compiler believes `RuleA` and `RuleB` are not disjoint, which means that the compiler could not prove that the rules are mutually exclusive. The rules have two scheduling restrictions between them, a read of the RWire `RWire1` by `RuleA` and a write into it by `RuleB`, and a read of `reg1` by `RuleA` and a write of

it by `RuleB`. RWires require that all reads must happen any writes on a cycle. This means that that `RuleA` must happen after `RuleB` in a cycle. Registers require that all reads happen before any writes to the register in a single cycle. This implies that `RuleB` must happen before `RuleA`.

Looking at the "<" conflict `RWire1.wget vs. RWire1.wset` we see that if `RuleA` is scheduled before `RuleB` the RWire conflict is the only conflict preventing parallelism.

At this point, we have a good idea of exactly what parts of the rules are causing the scheduling conflict. Hopefully, we can now change the design to correctly change the schedule.

6.4.2 Verifying Rule Firing

Sometimes rules are not executable when we would expect them to be. This is due to the fact that by default the compiler will add implicit conditions to the rule, even if the it is necessary only part of the time. For example, in the rule shown below, we want to take one value from either of the two FIFOFs, whenever there is a value and we can add to the output FIFO. A FIFOF is a FIFO with externally visible "not Empty" and "not Full" signals. It is natural for the designer to expect that the implicit conditions only prevent the rule from firing when both FIFOFs are empty or the output FIFO is full, but the implicit rules are added to the entire predicate, meaning that the compiler will decide the rule can only fire when **both** FIFOFs have a value and the output FIFO is not full.

```
"Rule_merge_fifofs":
 when True
  ==> action
        if fifof1.notEmpty
          action
           fifoout.enq(fifof1.first)
           fifof1.deq
         else if (fifof2.notEmpty)
```

```
 action
  fifoout.enq(fifof2.first)
  fifof2.deq
else
 noAction
```

The most straightforward method for the user to determine exactly what the implicit condition for a rule is by looking at the predicate listing from the `-show-schedule` flag. This states what the compiler believes is the full predicate on the action. If the designer finds that the reason that a rule is not firing is due to implicit conditions being added incorrectly, he can add the `-use-aggressive-conditioning` flag. This flag makes the compiler take into account whether implicit conditions are caused by predicated actions. By using this flag in the above case we would get the expected predicate:

```
(fifof1.notEmpty || fifof2.notEmpty) && fifoout.notFull
```

instead of:

```
fifof1.notEmpty && fifof2.notEmpty && fifoout.notFull
```

The reason this transformation is not done by default is that in general this results in a noticeable increase in hardware and timing length, and the optimization is generally not needed.

6.4.3 Checking Rule Ordering

When the compiler cannot determine a temporal ordering for methods in a module, it will make an arbitrary choice. Normally this is not a problem, but sometimes it causes unexpected conflicts. An example of this occurs when we separately compile two modules which have two independent Get-Put communication interfaces between

them. Since the channels are independent, there is no ordering between methods on the same rule so the compiler chooses an random ordering in each module. However, when we connect the two method pairs together into a two separate rules we now have to deal with the artificial orderings we added when compiling each module. If the compiler's choices are not consistent for the two modules, we will find that there is an ordering conflict. To fix this we need to tell the compiler that the choices that is makes cannot be completely arbitrary. We give the compiler artificial timing restrictions for the methods by use of the `internal_scheduling` pragma as shown below. With this we can specify conflicts between any two atomic actions.

```
(* internal_scheduling = "atomic_action1 SB atomic_action2" *)
```

Chapter 7

Conclusion

Translating high-level descriptions into a Bluespec design is relatively simple if the design does not have to worry about performance. Since we can look at each atomic action in isolation, the task of debugging is greatly simplified. In this method the only places where scheduling needs to be considered are places where optimizations have been taken to reduce hardware, such as replacing what is a logical FIFO with a register because the consuming action always takes the value on the next cycle. However, this sort of change should be a final optimization and should be applied only after the entire design has been verified and shown to meet the implicit assumptions necessary for the replacement. Therefore, this sort of constraint shouldn't affect verification much.

Once cycle-level performance considerations are added, the problem becomes more complicated. If we avoid the use of "unsafe" module components, such as RWires and ConfigRegs (which either ignore conflicts or reintroduce rule concurrency into the model), then this only requires the designer to verify that the schedule ordering and conflict analysis matches his high-level analysis. The designer can do this analysis separately from correctness analysis.

If the designer cannot achieve the necessary cycle-time performance using only safe

module components, then he has no choice but to add unsafe components to meet the requirements. Because of this, the designer must be aware of exactly what the rules do when scheduled together. The designer must analyze how rules interacting with the same unsafe modules behave when scheduled and verify that this matches his intention. While this can be difficult, this analysis is exactly the same analysis that a designer would have to use if the designer were to use a normal RTL language to handle this concurrency.

While the design methodology presented in this paper makes analysis of composition and conflicts of atomic actions relatively straightforward, further work needs to be done to help automate and simplify the scheduling analysis that the designers must do. In addition, a new strategy for handling scheduling conflicts was proposed [4] recently. This uses type of state, the Ephemeral History Register or EHRs, to allow the compiler to change the design meet the designer's desired cycle-level concurrency. This new approach provides many clear benefits, but it is still unclear how one should approach the design process. Further consideration of this is needed.

Bibliography

[1] Arvind and X. Shen, *Using Term Rewriting Systems to Design and Verify Processors*, IEEE, Micro Special Issue on Modelling and Validation of Micro-processors Vol. 19(3): pp. 36-46, 1999.

[2] J. C. Hoe, *Operation-Centric Hardware Description and Synthesis*, in Dept. of Electrical Engineering and Computer Science: Massachusetts Institute of Technology, 2000, p. 139.

[3] J. C. Hoe, and Arvind, *Synthesis of Operation-Centric Hardware Descriptions*, presented at IEEE/ACM International Conference on Computer Aided Design (ICCAD), 2000.

[4] D. Rosenband *The Ephemeral History Register: Flexible Scheudling for Rule-Based Designs*, presented at ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2004.

[5] N. Dave, *Designing A Reorder Buffer in Bluespec*, presented at ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE), 2004.

[6] *Bluespec Testing Results: Compairng RTL Tool Output to Hand-Designed RTL*, `http://bluespec.com/images/pdfs/InterraReport042604.pdf`, 2004.