Superscalar SMIPS Processor

Group 2
Qian (Vicky) Liu   Cliff Frey

1 Introduction

Our project is the implementation of a superscalar processor that implements the SMIPS specification. Our primary goal is to achieve high performance on a single thread of execution. Relatively large area or power consumption is acceptable.

Our major architectural features are an out-of-order execution unit based off of Tomasulo's algorithm, and an in order commit mechanism that supports precise exceptions. We also support speculative fetch and a memory system that returns results out of order.

The out-of-order execution unit enables the processor to do the maximum amount of computation possible given what results have currently been computed. This improves IPS when memory latency is high or variable. The tradeoff is that each instruction may take more cycles to execute, and complex control logic is necessary to move the result of a computation to where it is needed.

A major shortcoming of Tomasulo's algorithm is that it does not support precise exceptions. However, computer architectures that do not this feature are unpopular since it is quite difficult, or maybe even impossible, for programmers to debug in the event of an exception. The commit mechanism adds support for precise exceptions by allowing the processor to roll back changes to the processor state caused by instructions that should not have been executed due to exception causing events or an interrupt signal.

2 Design

2.1 UTL Design

At the most basic level, our design is only really broken up into two isolated modules: the fetch unit and the execute unit. These units can be isolated with one buffer of messages in between them. It is possible to further isolate different parts of the execution unit, but this seems less valuable. The different parts of the execution core are fairly highly connected, and some of the interactions between the modules will be buffered, while other interactions will not be. A possible breakdown of modules is shown in Figure 1.

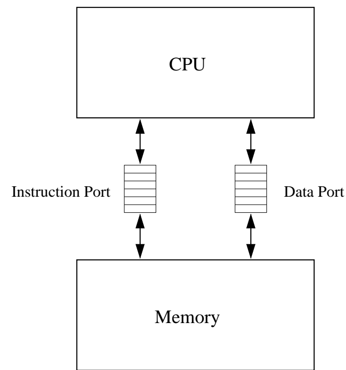

Figure 1: UTL Design. Abstractly, our design is broken up into these modules that send messages to each other. The commit unit is not in our first implementation.

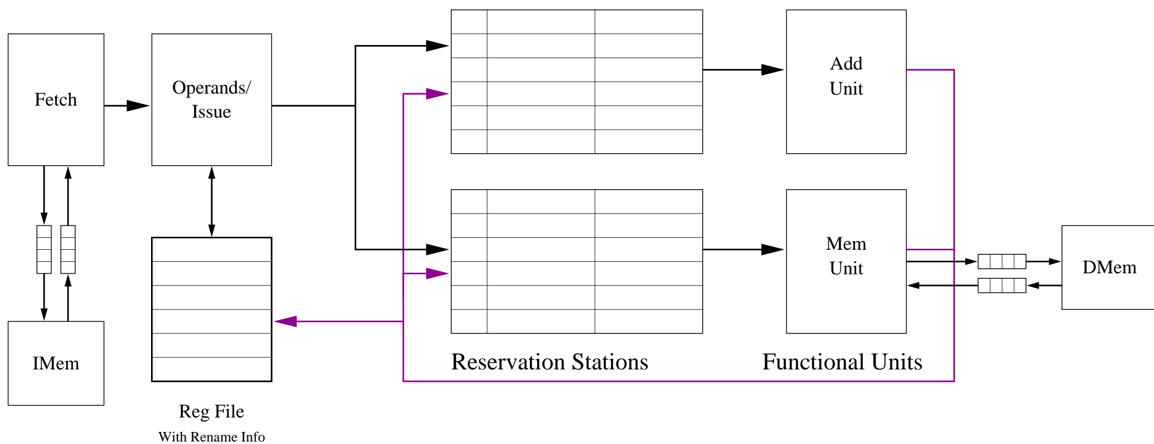

Figure 2: Design of execute stages. Our implementation of Tomasulo's algorithm. The CDB is shown in purple

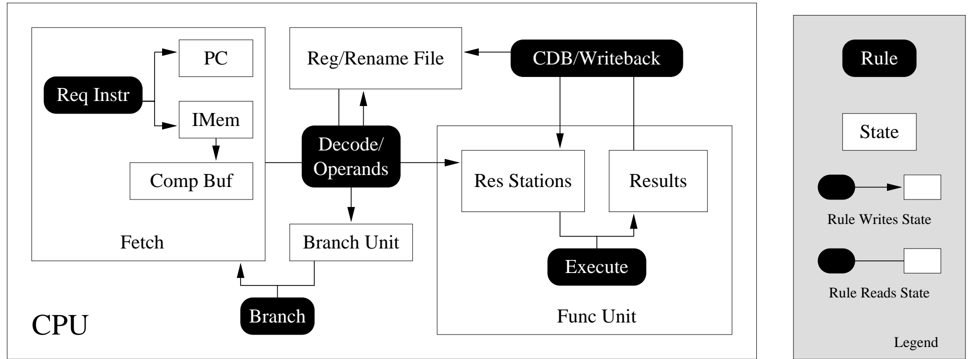

Figure 3: Bluespec design. Our BlueSpec module design for a superscalar processor using Tomasulo's algorithm. The diagram shows the various modules in our system (rectangles), their hierarchy, and the rules (black rounded rectangles) that act upon the modules.

2.2 Modular Design

The execution unit is based off of Tomasulo's algorithm. The implementation depends on various data structures maintained in hardware. Figure 2 shows the main components and paths between the modules.

The main register file contains actual register values, or tags corresponding to results that are currently being computed.

The Operands/Issue unit takes instructions, reads the operands out of the main register file, and issues the instructions to an appropriate reservation station. This unit also does a write to the main register file that marks the destination register as a result that is currently being computed.

There are also reservation stations. Each decoded instruction is placed in a reservation station waiting for a specific type of execution unit. These stations hold the operands (or tags corresponding to pending operands). These operands listen to the common data bus (CDB) and update themselves when the results have been computed.

Each cycle, each functional unit can take an instruction that has all of its operands ready and begin executing it. Once the result of the instruction is ready, it is stored in a result register for that functional unit.

Each cycle, all of the result registers from the functional units are inspected and, if available, a result is chosen to be broadcast on the CDB. This broadcast updates instructions that depend on that result that are waiting in various reservation stations. It also updates the main register file with the result if the tag in the reg file matches the tag of this result.

3 Bluespec Design

The Bluespec design structure somewhat matches our modular design in that the Fetch Unit survived. Most other units have recursive dependencies, and were therefore broken up into different state elements and rules that governed them. The decode, writeback, and branch rules tie together the entire CPU module. The Fetch

Unit takes care of updating the PC, making branch predictions, and queuing up fetched instructions. Each Functional Unit is responsible for maintaining its own reservation stations and driving the execution of ready instructions. Finally the Register File is its own unit that takes care of renaming registers in addition to reads from and writes to registers. Refer to Figure 3 for a diagram of the relationship between the rules and the units.

3.1 Fetch Unit

The Fetch Unit contains a completion buffer that serves as the instruction queue, the program counter, and a reference to instruction memory. The unit has two rules to make requests and get responses from instruction memory. The Fetch Unit interface allows the decode rule to drain the next fetched instruction from the completion buffer, and it allows the branch rule to set the new PC when a branch is taken. Upon changing the PC, the unit must clear the buffer and discard the corresponding memory requests.

3.2 Functional Units

In our design, the different Functional Units make up a distributed Execution Unit, with all of the units sharing a common interface. Each unit contains its own set of Reservation Stations, subunits for performing its respective operation, and a result register. Currently, we have implementations of adder, logic, shifter, branch, coprocessor, and load/store units.

Each unit is responsible for having a set of rules that actually execute instructions. For simple functional units, this is just one rule that finds a ready instruction in the reservation stations, calculates the result, and stores it in the result register. A "ready instruction" is an instruction where all of the operands have values rather than tags that refer to results of other instructions. Each unit differs in how it implements the execution of an instruction. For example, the load/store unit will require multiple rules (for making requests and receiving results), while the branch unit will calculate the branch target address and whether the branch should be taken or not.

The Functional Unit interface exposes the Reservation Stations to the CPU level rules. This interface allows the decode rule to queue decoded instructions into the stations and for the writeback rule to update the stations with the latest result. The interface also allows the writeback rule to access the results computed and to update all the reservation stations.

The branch unit is special because its results are not broadcast on the CDB. Instead, a special branch rule in the CPU looks at the branch unit's results, and determines if the branch should be taken or not. This unit was heavily modified once we added in support for precise exceptions.

3.3 Register File

The initial register file unit is very simple because we do not support precise exceptions. Each register either holds a value or the tag of an instruction that will produce the value. The Register File interface allows the

decode rule to read out operand values and the writeback rule to update the registers with the latest result. In addition, the decode rule needs to tell the Register what the current instruction's tag and what register the instruction writes to. There are no rules in this unit since all the changes will be driven by calls to the Register File methods.

3.4 Decode Rule

The decode rule is responsible for getting an instruction from the Fetch Unit and deciding which functional unit to issue the instruction to. This rule determines the type of the instruction and reads the value of the operands from the Register File. It also does a write to the main register file that updates the destination register to be a tag for the result of this instruction. In addition, the decode rule checks the result that is currently being written back because the updated value is not yet available from the register file. Each instruction is also issued a tag that is a combination of the register the instruction writes to and a unique number. This tag is given to the register file for renaming purposes. In the end, the decoded instruction is queued into the appropriate Functional Unit's reservation station.

The decode rule is only allowed to fire if there is not branch in progress as our design does not support speculative execution.

3.5 Writeback/CDB Rule

The writeback rule broadcasts one result out of the available ones in the various functional units. The priority order in which we broadcast results is loads/stores, other multistage computations (i.e., multipliers, dividers), and then single stage computations. The writeback rule will update all the reservation stations and the register file with the result and the tag of the instruction that produced it. This rule also updates an RWire that is checked by the decode rule.

3.6 Branch Rule

The branch rule monitors the branch unit. In the case that a branch rule is executed, this rule will update the PC via the Fetch Unit if the branch is taken. This model was changed once we implemented a commit stage.

4 Design Explorations

4.1 Supporting Precise Exceptions

One of the major drawbacks of Tomasulo's algorithm is its inability to support precise exceptions. A processor that supports precise exceptions must allow all instructions before the exception causing instruction to execute normally, but it cannot allow any instruction that comes after to make processor state changes that are visible to the programmer. In Tomasulo's design, since instructions are executed and written back

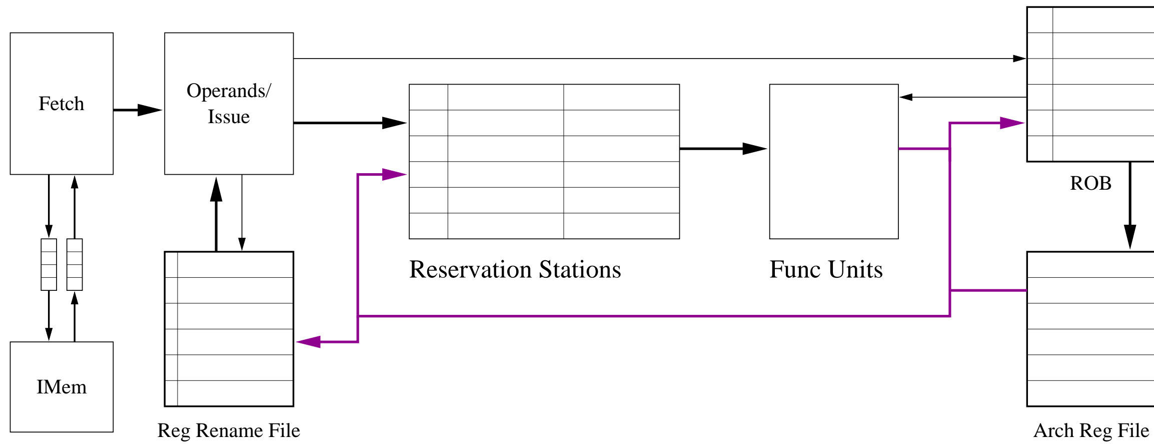

Figure 4: Updated hardware design. Our updated hardware design which now supports precise exceptions. The CDB is shown in purple.

out of order, there is no easy way to determine which instructions came before or after the exception causing instruction and no mechanism for nullifying the latter instructions. Furthermore, there is no way to roll back changes to the register file that are caused by instructions that come afterwards because the register renaming scheme we currently use does not remember previous values of a register. Finally, our system cannot revert external changes, such as stores and and writes to coprocessor registers, that might have executed before the exceptional instruction was identified.

We wanted to support precise exceptions by extending our original design rather than overhauling our original implementation. With this in mind, we decided to add a fifth stage, the commit stage, to our machine. The commit stage graduates instructions in order by using a reorder buffer (ROB), and this feature allows the processor to determine which instructions came after an exception causing instruction.

As instructions are graduated from the reorder buffer, their results are written to a new architectural register file. The architectural register file also contains all 31 registers, however the values in this register file are always consistent with an in order execution of the instructions. Our original register file is not changed in any way; it is still updated by the out-of-order write back stage. By adding the architectural register file, we have a way of recovering our rename register file after an exception happens.

The final change to our system involved making external changes occur at commit rather than at execution time. For example, rather than changing the PC right when a branch is resolved in the Branch functional unit, we cache the new PC and when the branch instruction reaches the commit stage, we retrieve the cached information and tell the fetch unit the new PC to fetch from.

Refer to Figure 4 for the high level hardware design of our new processor. We translated this design quite faithfully into BlueSpec. We added two new modules, the ROB and the architecture regfile, and we added two rules that power the commit stage. One of the rules, the Commit rule, handles normal commits, and the other rule, the Undo rule, disposes of instructions that come after an exception occurs. Only one of

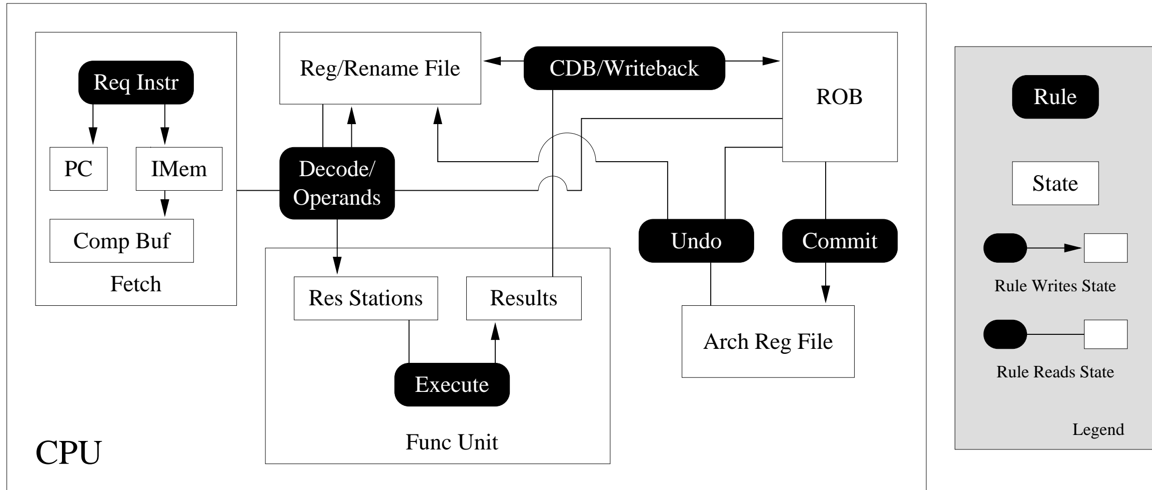

Figure 5: Updated BlueSpec design. Our updated BlueSpec rules and module design which now supports precise exceptions. Note that not all rule dependencies are actually shown in this figure. For instance, the Writeback rule modifies the reservation stations. Also, the Commit and Undo rules can modify the functional units in the case of taken branches or executed store instructions.

the two rules will ever be allowed to fire. Figure 5 shows our updated BlueSpec rule and module design.

4.2 Bluespec Design

This section goes into more detail about implementation of the new modules we added and the changes we made to the existing rules and modules. Figure 6 is a detailed diagram of how the rules and the modules interact, as well as how data flows in our processor.

4.2.1 ROB

The ROB is simply a completion buffer where each entry holds the PC, the register to write to, the result value of the instruction, and an error code. The main purpose of the error code is to flag instructions that cause exceptions, but in our current system, it is slightly abused in that it is also by the commit rules to determine if this instruction was a branch, store, or coprocessor write.

The ROB's interface is identical to the completion buffer interface: reserve a token, complete an entry, and drain out an entry. In addition, there is a method, getLastReservedToken, that returns the value of the last reserved token.

4.2.2 Architectural Register File

The Architectural Register File is implemented as a register file of 31 registers. There is one read port and one write port. Each entry is a 32 bit value.

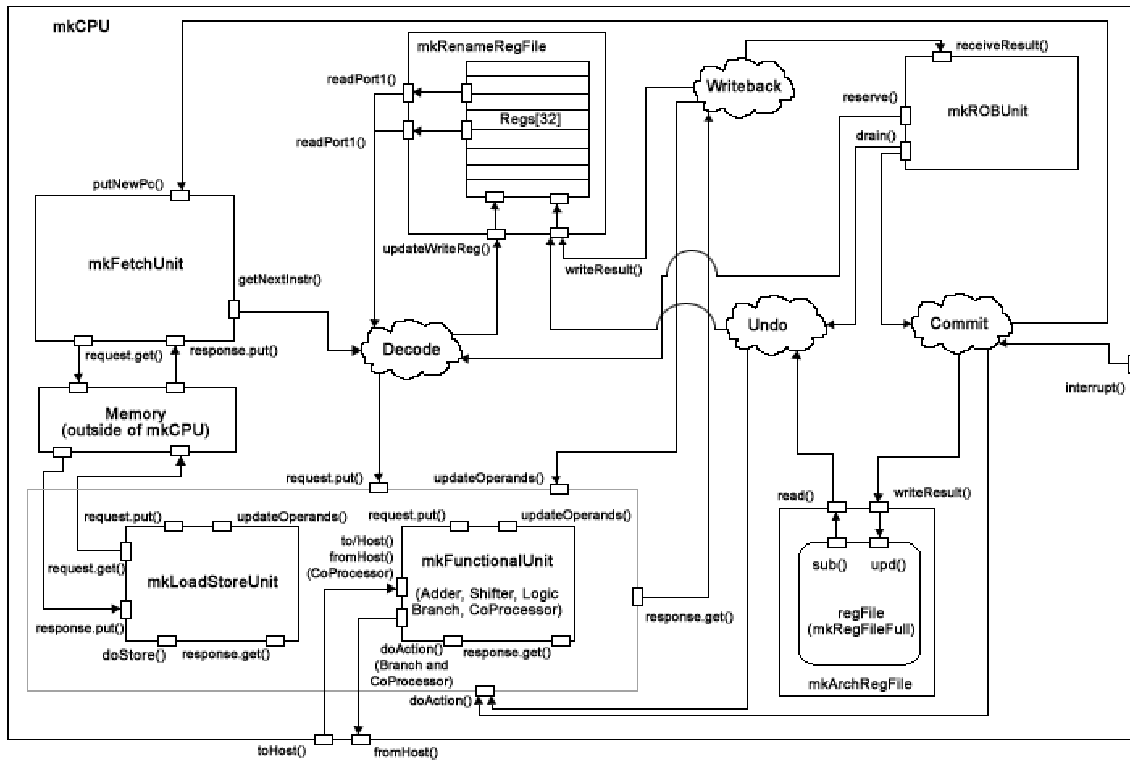

Figure 6: Detailed BlueSpec design diagram. Shows the interfaces of all our modules and how the rules interact with the modules. Some methods are not included, and some global processor state is missing, such as if we are currently handling an exception or a branch.

4.2.3 Commit Rule

The Commit rule is fired when the processor is not handling an exception. Each cycle, the Commit rule drains an entry from the ROB and checks the error code. If the error code signals an exception, such as a misaligned load or addition overflow, the commit rule will flag that the processor is now in exception handling mode. The results of the exception causing instruction are not stored, the PC of the instruction is stored to the epc register through the Coprocessor unit, and the PC is set to the address of the interrupt vector (0x1100 in sMIPS) through the fetch unit.

If the error code reveals that the entry corresponds to a store or coprocessor write, the Commit rule tells the load/store unit to actually queue up the store request or do the write. If the instruction was a taken branch, the Commit rule gets the new PC from the Branch unit and sets that as the PC through the Fetch unit. Otherwise if the error code denotes that the instruction is normal, the Commit rule will simply writes the result in the appropriate architecture register.

4.2.4 Undo Rule

The Undo rule is fired only when the processor is handling an exception, which is only flagged by the Commit rule. The Undo rule drains one entry from the ROB each cycle until the ROB is empty. For each instruction drained, if the error code denotes that the instruction is a store, a coprocessor write, or a branch, the Undo rule tells the appropriate functional unit to abort that operation and abandon any cached information. Since the functional units do not cause any external changes before the instruction commits, there is no need to roll back anything.

If the error code denotes that the instruction caused an exception or is a regular instruction, the Undo rule gets the register that the instruction writes to and copies the value of that register from the Architectural Regfile over to the Rename Regfile. Since no results are written into the Architectural Regfile after an exception is encountered, the values in the Architectural Regfile do not contain changes caused by instructions that came after the excepting instruction.

Our design recovers only the registers that need to be recovered rather than copying the entire Architectural Regfile over to the Rename Regfile. Also, we have to wait until all the decoded instructions finish executing before we finish handling an exception. If several load instructions were decoded past the exception causing instruction, this can take quite some time. We chose this design because it was easier to adapt our original design in this way than to add a global clear signal to all the units and copy all of the contents of the Architectural Regfile over to the Rename Regfile.

4.2.5 Decode Rule Changes

The decode rule requests a token from the ROB for each instruction. This token replaces the instruction token in our original design and is used for renaming in the Rename Regfile. The decode rule also stalls when the processor is handling an exception.

4.2.6 Functional Units Changes

The Functional units were changed in several ways to support precise exceptions. In our original design, not every instruction needed to go to the writeback stage since some instructions do not write back results. However, because the ROB needs to keep track of the order of all instructions, we changed functional units to issue placeholder results even for instructions such as stores.

A second change moved the execution of external changes to commit time rather than execution time. In the Branch unit, this involved remembering the new PC of a branch that is resolved to be taken. At commit time, the Commit rule takes the new PC and sends it to the Fetch unit. Execution of additional branches is delayed until taken branches are committed. In the Coprocessor Unit, the value to be written is stored and the register is actually written when the Commit rule comes across the instruction. This unit is also stalled until writes complete.

The changes to the Load/Store unit were more extensive since our original unit was incorrect and allowed the reordering of stores and loads. We changed the reservation station in this unit to behave like a FIFO so loads cannot be reordered with respect to stores. When a store instruction is ready, the address and the value are saved in a one-slot cache. When the Commit rule calls `doStore(True)`,the cached information is turned into a store request and actually queued up. No other store instruction can be executed until the previous store is committed. If the instruction is a load, the address of the load is checked with information in the cache. If the addresses match, a load request is not made to memory, and instead, the cached value is used as the result.

4.3 Wrestling BlueSpec Method Conflicts

Our original design could not reach high performance partially because there were many methods that could not be called in the same cycle. Pretty much every one of our modules that is accessed from multiple rules ended up having this issue.

The completion buffer of instruction fetches should be able to handle `reserve`, `complete`, and `drain` all in the same cycle. Similarly, the reorder buffer needs to support those three operations simultaneously as well. The main register renaming file needs to support renaming a register with a tagged value and the update of a tagged value to a real result in the same cycle. And last, the reservation stations need to support placing a new instruction in, removing a ready instruction, and updating operands all in the same cycle.

It is fundamentally impossible to accomplish all of these different tasks by only using multiple methods. For instance, with the reservation stations, if each of these operations are placed in their own methods, the methods will conflict, because all of the methods will do reads of the station status bits, and potentially do writes to the station status bits. However, it is actually safe for these methods to be run in parallel because each method only modifies one kind of status, but the BlueSpec compiler cannot infer that much information.

There are two solutions that are possible to the problem. One solution is to just make one method that

Fetch	Issue	Exec	CDB/WriteBack	Commit
load				
divide	load			
add	divide	load (not ready)		
	add	load (not ready) divide (not ready)		
		load divide add		
		divide add	load	
		divide	add	load
			divide	
				divide
				add

Figure 7: Some instructions moving through the pipeline. All three instructions entered the execute stage, and then all of the results became ready during the same cycle (the load took 3 cycles, the divide took 2 cycles, and the add only takes 1 cycle). The results are then moved out onto the CDB one at a time, in a fairly arbitrary order (ranked by which unit, not by program order). In this case, the divide and add instructions were reordered, but the commit stage actually commits the instructions in program order.

does all of the functionality that is needed. However, this is exceptionally difficult to use, because this one method must be called from just one rule, meaning that you must do nearly everything in your system in one rule. The much more reasonable solution is to make each method copy its arguments onto a *RWire*. There can then be a rule inside of the module that will look at all of the relevant RWires and update all of the state within the module. It is very important that this rule either is activated every time one of the methods is called, so that state updates are not mixed. As a general rule of thumb, any module using this model should do *all* state updates in that rule (do not update state in any of the methods) so that there is no risk of conflicts.

Using this trick, the methods in the reservation stations and the register rename file were changed such that none of them conflicted. The reorder buffer and fetch unit were changed to use a hand made completion buffer based off of the lab 4 material that used `ConfigReg` components to avoid method conflicts.

Once all of these changes were made, all of our main rules could fire at the same time, resulting in many fewer pipeline bubbles. As traces in our results section show, it is possible for all fetch rules, the decode rule, the execute rules, the writeback rule, and the commit rule to all fire in the same cycle.

5 High Level Pipeline Design

Due to variable memory latencies and computation costs, instructions do not take a constant amount of time to go through our processor. The first time that our processor references an instruction is when the address of that instruction has been calculated by the fetch unit. This address is then sent to the instruction memory. Some number of cycles later, the actual instruction is returned from memory. The instruction is then decoded, its operands are fetched (if available), and it is issued to an appropriate reservation station. The instruction may wait here for its operands to be calculated or for its functional unit to become ready

to execute it. If it is a simple integer instruction, the functional unit will take only one cycle to calculate and broadcast the results on the CDB. If the instruction is a load, it may be many cycles before the memory returns a result. Finally, in our reorder buffer implementation, the instruction may wait in the ROB until all instructions before it (in program order) have been fully executed. Once the instruction is committed, it has essentially been executed and it no longer uses any resources on the processor.

Our maximum instruction window will be determined by the size of many of our buffers, and the maximum delay of memory. Assuming that we do not have a ROB, the size of the instruction window will essentially be the sum of the maximum number of instructions being fetched at one time, the number of instructions being issued at one time, the number of reservation stations that we have, and the maximum number of instructions that are in any of the functional units at a given time. With a ROB, the maximum instruction window is simply the number of instructions being fetched at once plus the number of entries in the ROB.

5.1 Structural Hazards

The most obvious structural hazard possibility is that more than one result could arrive each cycle, but only one result can be transmitted on the CDB. We resolve this hazard by storing results in each functional unit until they have been transmitted over the CDB. This also means that each functional unit will not allow more instructions to enter it than it has spaces in its output buffer. This solution makes sense because other than memory operations, most operations are either very fast (single cycle integer operations), and then the output buffer is really just a pipeline register, or very slow (like divide instructions), and having an output buffer does not seem expensive.

Another way of looking at this is to see how back-pressure works in our system. The execution core can exert back pressure on the fetch unit by not taking instructions from it. This will cause the fetch unit's completion buffer to fill up, and eventually the fetch unit will stop making new requests to main memory.

The functional units can exert back pressure on the issue stage by not emptying the reservation stations. If the decoder can not issue an instruction because there is no reservation station available, then it will stop issuing instructions until there is room.

The CDB takes results one at a time from the functional units. The functional units are self stalling in that they will not take in new inputs unless they have room for the results.

And finally, in the precise exception supporting implementation, the ROB exerts back pressure by requiring decoded instructions to reserve a spot in the ROB before they are issued.

6 Verification

Most of our testing is at the CPU module level. By loading different programs into memory, we test many different aspects of our implementation. Many of these tests are constructed with knowledge of our specific

implementation, and try to isolate specific systems.

We also modified the memory system to support delaying responses based on the data that was actually being fetched. When reading data where the high 2 bytes were `0xDE1A`, the memory system would look at the low 2 bytes of the word, and wait that many cycles before returning a result. For instance, loading the value `0xDE1A0010` would take 16 cycles.

`simple_inthandling.S`, `simple_2int_handling.S` Verifies that basic exception detection and handling works. The second test file contains two back to back addition overflow exceptions.

`regfile_after_int.S` Verifies that operations do not get committed that happen after an exception causing instruction, even if the execution in reordered.

`proper_renaming.S` Assigns many different results to the same register, that are ready in different orders, making sure that the broadcast of results and the assigning of tags works.

`full_rstation.S` Verifies the case where we run out of reservation stations in a unit. In this case, the decode stage stalls before issuing more instructions.

`add1_test.S`, `add2_test.S` Simply has many back-to-back dependent adds (`add1`), or non-dependent adds (`add2`) so that we can look at the trace output and see what is happening.

`simple_loadstore_hazard.S`, `simple_loadstore_hazard2.S` Creates two situations where loads and stores from the same address can be executed out of order. The tests ensure that the correct value is returned from memory.

`self_test.S`, `all_test.S`, etc. These test the basic SMIPS functionality.

These are most of the tests that we actively used in verifying our design. It is by no means complete, and it is very likely that our design has flaws. For instance, there is no test that checks an exception instruction in a branch-delay slot.

In addition, we did not test any of the functionality that we did not actually implement. This includes all of the coprocessor except for the EPC and to/from host registers, as well as multiply and divide instructions.

7 Results

We successfully implemented a SMIPS CPU that can handle precise exceptions and execute instructions out of order.

We have size and timing numbers from both our original working implementation and our optimized implementation that includes precise exception support. These numbers are shown in figure 8.

More directly, changing our BlueSpec data structures so that their methods did not conflict doubled our IPC from .25 to .5 on the qsort benchmark.

Implementation	Area	Frequency
Base (no exceptions, unoptimized)	0.236 mm^2	300 MHz
Final (exceptions, optimized)	0.338 mm^2	294 MHz

Figure 8: Area and frequency measurements after synthesis for our first working implementation, which had no precise exception support and had many conflicting BlueSpec methods, and our final implementation.

Build Phase	Target	Time (min:sec)
BlueSpec	`mkCPU.v`	1:34
VCS	`simv`	0:04
tests	`run-tests`	0:12
Synopsis	`synth`	16:23
Encounter	`pr`	18:45

Figure 9: Time to build various parts of our system on an AMD 3000+ linux machine with 1GB of memory. Each build phase assumes that previous build phases have completed.

Our build process used `bsc-3.8.43-rh7` and `vcs_mx7.1.2`. Incremental compile times are shown in figure 9.