# Superscalar SMIPS Processor
# Group 8

Andy Wright
acwright@mit.edu

Leslie Maldonado
lmald02@mit.edu

May 15, 2013

## 1 Project Objective

For our 6.375 project, we added superscalar execution to the SMIPS processor from the end of Lab 6. Superscalar execution is when a processor executes two instructions in the same cycle. This is only possible if the two instructions don't have dependencies between them that prevent them from being issued in the same cycle. Since superscalar processors issue multiple instructions per cycle, it is typical for these processors to get an instruction per cycle (IPC) measurement greater than one.

Our superscalar processor will features identical execution modules that share a single data memory. The additional execution pipelines increase the complexity of handling control and data hazards, and the shared memory structure introduces new structural hazards that need to be handled. Due to these hazards, the dispatching logic for the processor needs to be thorough.

This project started off as two way superscalar, but later we took advantage of Bluespec's parameterization capabilities and added support for N way superscalar processors.

## 2 Background

As processors start executing instructions in parallel (either through pipelining or superscalar), multiple different hazards appear. These hazards, if not resolved correctly, can cause the program to perform undesired actions and have incorrect results. Most of the complexities in this project involve properly handling these hazards.

### 2.1 Control hazards

Control hazards occur at branch instructions in pipelined processors. After the fetch of a branch instruction, the next instruction to fetch isn't known for sure until the branch instruction makes it to the execution module. To resolve this hazard, a pipeline bubble can be introduced, stalling the execution of further instructions until the result of the branch is known. This is very inefficient because branches are a common instruction in programs often taking up on the order of 10% of all instructions.

Our processor uses a prediction module to predict which instructions should be read after a branch. The execute module handling the branch instruction calculates the next `PC` and compares it to the predicted next `PC`. In the case that the prediction is wrong, the execute module sends a signal to the fetch module and the other execute module to kill all instructions in flight that occur after the branch in the program. This is done by tracking the epoch in each module and modifying it when there is a mispredict. The execute module will also send a signal to the fetch module to reset the `PC` to the actual value for the branch calculated by the execute module.

## 2.2 Data hazards

There are multiple hazards that occur from the processor reading and writing registers in a different order than the original program described.

### 2.2.1 RAW

A read-after-write (RAW) hazard is when a read is scheduled to occur after a write to the same register. This becomes an error if the read is performed before the write completes. This problem comes up often in pipelined processors because the next instruction is started before the previous instruction finishes. The following instructions show a RAW hazard.

```
add $t2 , $t0 , $t1
sub $t4 , $t2 , $t3
```

This hazard become a problem if `sub` is decoded and `$t2` is read before `add` finishes. To avoid this error, registers that are in the process of being updated can be tracked, and instructions depending on these results can be stalled creating a bubble in the pipeline. These bubbles slow down the processor. To avoid the slow down, the result of the earlier operation can be forwarded to the operand of the later instruction, bypassing the register file.

In our processor we have a scoreboard that keeps track of writes. An instruction will not be issued if it has a register to read that has a pending write in-flight. Also, a second instruction won't be issued in the same cycle as a first instruction if the first instruction is writing to a register read by the second instruction.

### 2.2.2 WAR

A write-after-read (WAR) hazard is when a write is scheduled after a read from the same register. In the following code, a WAR error would occur if the result of `sub` was written to `$t0` before it was read for the `add` instruction.

```
add $t2 , $t0 , $t1
sub $t0 , $t3 , $t4
```

This hazard is normally not an issue for in-order pipelined processors.

In our processor, instructions are issued in (partial) order and reads occur in an earlier pipeline stage than writes so we do not need to worry about WAR hazards.

### 2.2.3 WAW

A write-after-write (WAW) error occurs when a write is scheduled after a write to the same register. A WAW hazard can cause an error in the output of a program if the second write is performed before the first. The following code has an example of a possible WAW hazard with two back-to-back writes to `$t0`.

```
add $t0, $t1, $t2
sub $t0, $t3, $t4
```

WAW hazards can't cause errors in processors that commit in order.

In our processor, instructions are issued and committed in (partial) order. We do not need to worry about these instructions being issued in the opposite order, but we do need to worry about these instructions being issued in the same cycle however. Our dispatch logic checks to see if the instructions are writing to the same register. If they are, then the later instruction is stalled until the first instruction is finished. Furthermore, our scoreboard architecture is only able to track one write to a specific register at a time, so two writes to the same register can't be in-flight at the same time.

## 2.3 Structural hazards

There is also a potential hazard when there are two instructions that require the same physical resources such as a memory, a fifo, or an ALU.

In our processor, we have two execute modules that share a single data memory causing a potential structural hazard. This creates a structural hazard when two instructions are available to be issued in the same cycle, but they both need to access the memory.

Branch misprediction uses a redirect fifo that is read by the fetch module to correct the `PC`. There is only one input in the misprediction fifo, so the two execution modules need to share. This means two branch instructions that can be issued together causes a structural hazard.

Another hazard comes from writing to the coprocessor. The coprocessor used in our processor only has one write port, so two writes to the coprocessor in the same cycle creates another structural hazard.

All these structural hazards are dealt with in the dispatch logic by making sure the first and second instructions to issue will not be using the same hardware. The dispatch logic blocks two memory instructions, two branches, and two coprocessor writes because of these hazards.

# 3 High Level Design and Test Plan

To test our processor we use the existing SMIPS Sce-Mi interface to communicate with the design on the FPGA. The processor and memories changed, but we kept the Sce-Mi interface the same.

We leveraged the existing testing support for the SMIPS processor introduced in Lab 5 and 6 to test our superscalar SMIPS processor. The existing test suite contains a SMIPS compiler, functional test programs, and performance test programs.

The existing functional test programs test various portions of the SMIPS processor for correctness. The existing performance test programs test the processor's IPC for various workloads. We tested each processor design for accuracy to ensure the processor is functional, and performance to see how different design decisions affect the performance of the processor.

Since we only saw IPCs greater than one for one of the benchmark programs, and only for certain latency configurations, we created our own benchmark that would get very high IPCs for our architecture. The new benchmark is called circadd, and it is a series of additions with circular dependencies that can be issued in parallel. The code for the functional portion of the benchmark is shown below.

```
void circadd( int in[], int out[] )
{
    int a,b,c,d,e,f,g,h,i;
    a = in[0]; b = in[1]; c = in[2]; d = in[3];
    e = in[4]; f = in[5]; g = in[6]; h = in[7];
    for( i = 0 ; i < 256 ; i++ )
    {
        a = a + b;
        b = b + c;
        c = c + d;
        d = d + e;
        e = e + f;
        f = f + g;
        g = g + h;
        h = h + a;
    }
    out[0] = a; out[1] = b; out[2] = c; out[3] = d;
    out[4] = e; out[5] = f; out[6] = g; out[7] = h;
}
```

## 4 Microarchitectural Description

### 4.1 Overview

Our design is built off of the SMIPS processor from the end of Lab 6. The processor is split into modules as shown in figure 1. Each module is composed of methods that are called by top level rules. The top level rules call groups of methods that are in the same pipeline stage. This makes separate parallel modules such as the execution modules move synchronously.

### 4.2 Instruction memory

To support superscalar execution, multiple instructions are needed from memory per cycle. To meet this increased demand, the instruction memory needs to be able to provide multiple words per cycle. Since we have $N$ execution pipelines, our instruction memory module was expanded to allow for $N$-word unaligned reads. Also, its interface was modified so the

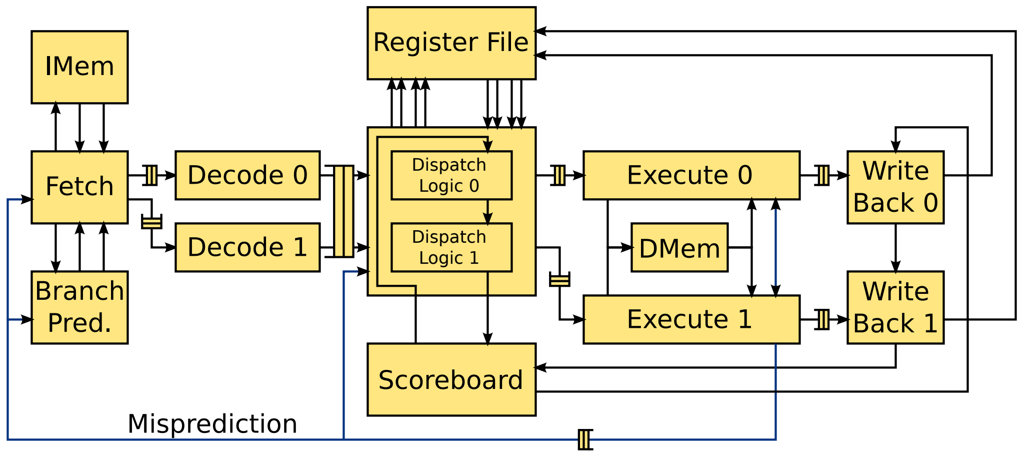

Figure 1: Top level view of the processor

response method returns a vector of the $N$ instruction words read from memory. A block diagram of the instruction memory module is shown in Figure 2.

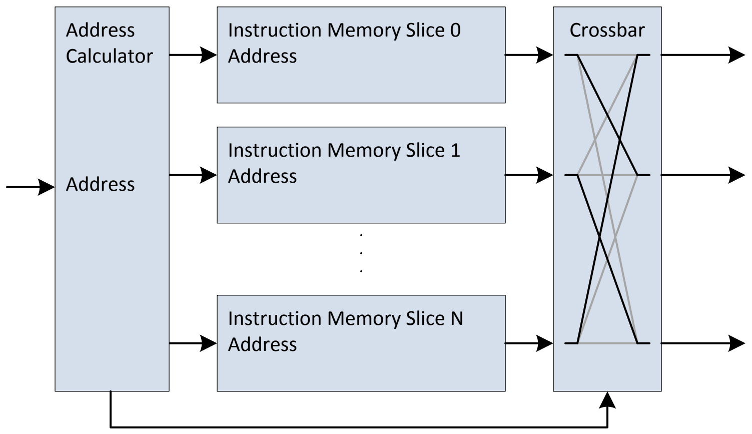
Address Calculator
Address
Instruction Memory Slice 0
Address
Instruction Memory Slice 1
Address
Instruction Memory Slice N
Address
Crossbar

Figure 2: Block diagram of instruction memory

Internally, the memory is split into $2^{\lceil \log_2 N \rceil}$ BRAM stripes. That means for a 2-way superscalar processor, then there are 2 BRAM stripes, but for 3-way superscalar, there are 4 BRAMs. This split is done so that no matter what address you are reading from, the next $N - 1$ words are all in different memories. In the case with two BRAM stripes, one BRAM holds even word addresses and one BRAM holds odd word addresses. The BRAMs are fed truncated addresses with minor modifications depending on the least significant bits of the address. The outputs of the BRAMs are a rotated version of the desired output, so muxes are used to reorder the output to the desired order. For $N$s that are not a power of two, the instruction memory is designed the same as the next highest power of two, except there are fewer output muxes for reordering.

## 4.3 Fetch

The fetch unit requests reads from the instruction memory from `PC`,`PC+4`,. . . , `PC+4*(N-1)`. It then queries the branch predictor using `PC+4*(N-1)` to get the next value for `PC`. The fetch unit also reads instructions from memory and packages them into a structure containing the instruction's `PC`, the next predicted `PC`, and fetch's epoch.

The fetch unit also receives the start `PC` from the host. The fetch unit stores that into `PC` and calculates `PC+4`, . . . , `PC+4*(N-1)` at the same time.

Fetch's interface is a connection to IMem, a connection to the branch predictor, and a vector of $N$ fifos that lead to decode modules. Fetch also receives the output of a redirect fifo for mispredictions.

## 4.4 Branch predictor

The branch predictor needs to predict the next N instructions since multiple instructions will be fetched in the same cycle. Two branch predictors were implemented. The simplest predictor is a `PC+4`/`PC+8`/. . . predictor. When `PC+4` is sent to the branch predictor from fetch, it calculates `PC+8`,`PC+12`,. . . and sends them back to fetch as the next predicted instruction addresses.

The other predictor is a hybrid predictor that uses a branch target buffer (BTB) to calculate the next `PC`, and then it adds 4 to that to get `PC+4` and others. The predictor is done this way because of limitations in the instruction memory. The current instruction memory only allows consecutive reads, so the branch predictor assumes consecutive addresses.

## 4.5 Decode

The decode module simply decodes the instruction into a format that is easier for the processor to use. Unlike the decode rule in the SMIPS processor from lab 5 and 6, this decode module does not read from the register file. The decode module receives input words from a fifo fed by fetch, and it writes to a superscalar fifo that leads to the dispatch logic.

## 4.6 Superscalar Fifo

Between the decode modules and the dispatch logic is a superscalar fifo. This fifo takes in $N$ elements at a time and provides one to $N$ elements each cycle. The fifo is constructed in a similar manner as the instruction memory. There are $N$ separate fifos used to make the superscalar fifo. Each fifo stores every $N^{th}$ item. The output of the fifo has a state variable that keeps track of which fifo has the next element. After dequeues occur, the state variable is updated to match which fifo now has the next element. There is a mux for each output of the superscalar fifo that selects which fifo is being read from. The state variable is used to calculate which fifo is being read from.

The interface for the superscalar fifo is made up of $N$ fifo subinterfaces. There are restrictions about how the fifo subinterfaces can be used, but the modules around the fifo always satisfy those restrictions. The interface also contains a clear method to remove all the data from the fifos and reset the state. A block diagram of the 2-way superscalar fifo can be seen in Figure 3.

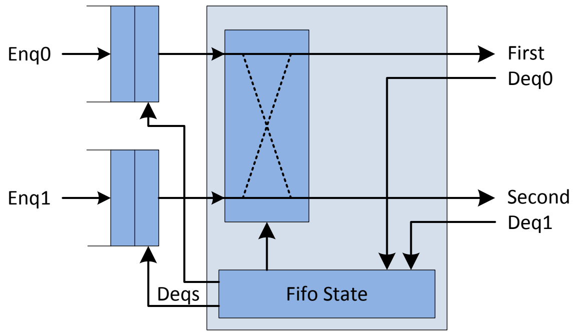

Figure 3: Block diagram of 2-way superscalar fifo

## 4.7 Register file

The register file needs to support N reads and writes per cycle since there are multiple execution modules. The register file's interface was expanded to an interface that contains a vector of $N$ of the old register file interfaces. Bluespec prevents multiple writes to the same register in the same clock cycle, and when there are two write ports in the same module, that is a possibility. To prevent Bluespec from stalling writes, the registers are implemented with EHRs and each write port writes to a different index in the EHR. The register file has implementations with and without bypassing. The register file with bypassing uses EHRs to allow the read ports to read from pending writes.

## 4.8 Dispatch logic

The core of the complexity that arises from superscalar execution is handled in the dispatch logic module. This module is used to avoid RAW and WAW hazards, along with structural hazards with the single data memory. Also this module is used to read from the register file when instructions are dispatched.

### 4.8.1 Issuing the first instruction

The first instruction loaded from the decoders has priority over the others, so if any instructions are going to be issued during the current cycle, it will at least be the first instruction. The first instruction only needs to look at previous instructions in flight to see if it can be issued. The scoreboard keeps track of all the register writes for instructions in flight, so it is used to see if there will be a conflict when issuing the first instruction. A local scoreboard keeps track of whether the first instruction is a branch and also if it is a data memory instruction. This local scoreboard is used to ensure that two branches or two data memory instructions will not be issued in the same cycle.

### 4.8.2 Issuing the next instruction

If the first instruction can be issued, then it is time to look at the second instruction to see if it can be issued in parallel. The second instruction cannot use any registers that will be written to by previous instructions in flight, so it also checks the scoreboard for possible hazards. If the scoreboard shows no hazards, then the dispatch logic looks to see if there is a data hazard conflict with the first instruction. The check is done by comparing the destination register of the first instruction with the registers used in the second instruction. If there is no overlap, then there is no data hazard to worry about. If the second instruction is a branch, it also checks the local scoreboard to see if the first instruction is already a branch.

Lastly, the dispatch logic needs to check for structural hazards. In our case, the only structural hazard we are worried about is the shared data memory between the two execution cores. The dispatch logic checks for this using the local scoreboard by making sure both instructions will not be using data memory. If they are not both data memory instructions, then the two instructions are ready to be issued in parallel.

### 4.8.3 Issuing

When an instruction is issued, its destination register is added to the scoreboard, and its source registers are read from the register file. Once that is done, it is sent to the fifo corresponding to the desired execution module.

### 4.8.4 Stalling

If the first instruction has a conflict and cannot be issued, then there is nothing done. The first instruction if stalled along with all other instructions. If a later instruction has a conflict, it is stalled along with the instructions that go after it, but the earlier instructions are still able to be issued. When there is a stalled instruction in the dispatch logic, the instruction is not dequeued from the fifo into dispatch logic.

### 4.8.5 Implementation

The dispatch logic module, as implemented, only looks at a single instruction. Multiple dispatch logic modules are chained together to get superscalar execution. The module has a method in it that takes in a scoreboard, tries to issue the instruction (including reading from the register file), and then it returns an updated scoreboard. The scoreboard passed from module to module also includes information about hardware hazards such as memory instructions and branch instructions. The scoreboard also passes along the information if an instruction has stalled yet. If an earlier instruction stalls, then all other instructions must stall too. A figure showing how dispatch logic modules are chained together can be seen in Figure 4.

## 4.9 Scoreboard

The scoreboard keeps track of all instructions that have already been issued but haven't been written back yet. Keeping track of these instructions allows the dispatch logic to easily detect potential RAW hazards.

The scoreboard is implemented with a bit vector EHR so changes to the scoreboard can be seen in the same cycle. There are two ports of the EHR, one for dispatch logic to use to search and insert, and one for writeback to remove. Depending on the register file used (bypass or non bypass), the reads and changes made by dispatch logic will either go before or after the changes made by writeback.

When the scoreboard is used by the dispatch logic modules, the scoreboard is combined with another data structure that keeps track of memory instructions, branch instructions, and if any previous instructions have been stalled yet. Each dispatch logic module takes in the current scoreboard as a variable, and it returns an updated scoreboard for other dispatch modules to use. This flow can be seen in Figure 4.

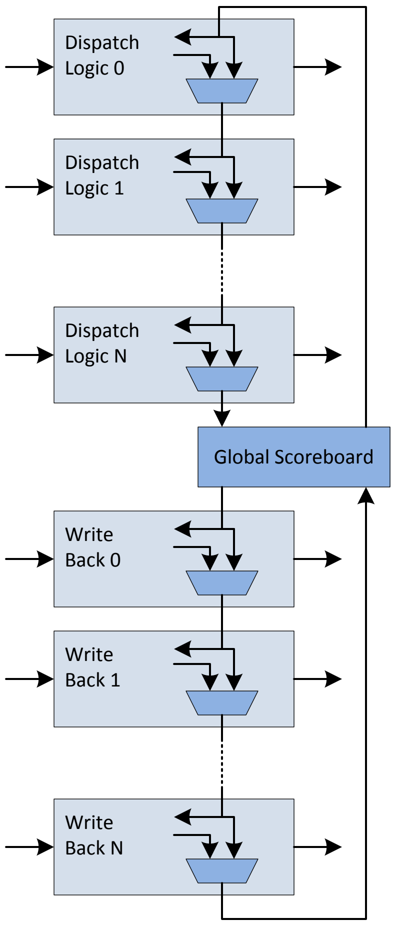

Figure 4: Flow of scoreboard through dispatch logic and write back

## 4.10 Execute

The execute module is based off of the execute rules from lab 6. Execute contains an ALU and it has access to a data memory. Each execute module has to share the same data memory, but that is taken care of by a combination of the dispatch logic and a memory sharing module.

Without a memory sharing module, Bluespec notices that there is a chance both rules will be using the memory, and it will not allow them to fire in the same cycle. The memory sharing module is a wrapper for the original data memory that creates $N$ virtual ports for the same memory. All the execute modules can write to their virtual port in the same cycle, but the wrapper gives priority to one of the execute module over the others, and only one of the write requests is passed to the wrapped memory. This allows the execute rules to fire in the same cycle.

The execute module also broadcasts notices of mispredicted branches, and it receives notices of mispredicted branches from the other execute module. To simplify the communication between the two execute modules, the execute modules are assigned priorities 0 to $N-1$. When $N$ instructions are issued in the same cycle, the earliest instruction is issued to execute 0 and the latest instruction is issued to execute $N-1$. This simplifies the case when one of the issued instructions is a mispredicted branch. By using an EHR for the execute epoch, changes to the epoch are seen by all the execute modules with instructions that are later in the program without affecting execute modules with earlier instructions. This implementation of the execute epoch can be seen in Figure 5.

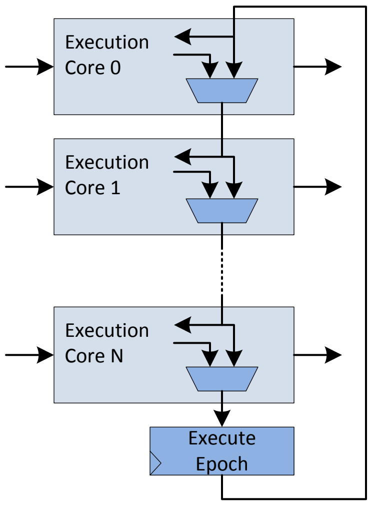

Figure 5: Block diagram of the EHR execute epoch implementation

The execute module's interface includes an epoch that is a single register from an EHR. Higher priority execute modules receive the lower numbered EHR registers, and the lower priority execute modules receive the higher numbered EHR registers. This allows all the execute modules to use the same code, and their priority is determined by how they are connected. Data is sent to and from the module through fifos, and a mispredict fifo is used to send corrected **??**s to the fetch module.

### 4.11 Write back

The write back module takes data from the execute module and writes the results to the register file or the coprocessor if needed. If a register was written to, its entry is removed from the scoreboard.

### 4.12 Coprocessor

The coprocessor is the module used for host/cpu communications. It keeps track of the number of cycles used and the number of instructions processed. It also allows for printing to the screen of the host. Coprocessor registers can be used as source and destination registers so they are also tracked in the scoreboard.

To allow for multiple instructions to be issued in the same cycle, we duplicated the number of read ports in the coprocessor.

To prevent a structural hazard, the dispatch logic does not allow two coprocessor writes to be issued in the same cycle.

## 5 Implementation Evaluation

### 5.1 Compilation Problems

When trying to simulate a 4-way superscalar processor, we ran the Bluespec compiler and it didn't finish within 5 hours of compile time. When we checked on its status we saw it was using a massive amount of memory ( 10 GB) and it probably wasn't going to finish. We explored a few compiler flags to see if we could reduce memory usage, but nothing worked.

We originally had the same problem with the 3-way superscalar processor, but after switching to the newer setup script and expanding the allowable stack size for Bluespec, the 3-way superscalar processor compiles in less than 15 minutes.

### 5.2 Structural Hazards

Throughout the design of the superscalar processor, we dealt with handling structural hazards. When dispatching instructions, we have to make sure no two instructions are going to be using the same hardware. We took the time to design it this way, but the Bluespec compiler does not have this information that we have. When the Bluespec compiler is used to compiler a design that has two rules that may calling the same action, the Bluespec compiler is cautious and doesn't allow the rules to fire at the same time, even though we know there will not be a conflict. We get around this by making wrappers for shared hardware. These wrappers normally share the action by assigning a fixed priority to all of the ports of the wrapper. If the dispatching logic works, then this fixed priority doesn ot matter.

### 5.3 Aggressive Conditions

The following code was not compiling with `-aggressive-conditions`, but it worked without that compiler flag.

```
rule even_filter;
    let x = input_fifo.first();
    input_fifo.deq();

    if( (x%2) == 0 ) begin
        ehr[0] <= True;
    end else begin
        ehr[1] <= False;
    end

    if( ehr[1] == True ) begin
        output_fifo.enq(x);
    end
endrule
```

Without `-aggressive-conditions`, `even_filter` had the following raised guard:

`input_fifo.notEmpty() && output_fifo.notEmpty()`

With `-aggressive-conditions`, the raised guard was

`input_fifo.notEmpty() && ((ehr[1] == False) || (output_fifo.notEmpty()))`

This depends on `ehr[1]` which depends on `ehr[0]` which is written by the rule `even_filter`. This dependence on on itself is what the bluespec compiler was complaining about.

One way to fix this is to add `output_fifo.notEmpty()` to the guard. This prevents the `(ehr[1] == False)` part of the guard from being raised, therefore preventing the self dependence.

This problem came up in our scoreboard because we were writing to an EHR and reading from a later port in the same EHR in the same rule. We fixed this by adding additional guards to the rule.

## 5.4 Area

The area results from synthesis for some of the modules can be seen below.

```
Number of Slice Registers:    16,539 out of  69,120   23%
Number of Slice LUTs:         24,152 out of  69,120   34%

+-----------------------------------------------------------------------+
| Module         | Slices*   | Slice Reg | LUTs       | LUTRAM | BRAM/FIFO |
+-----------------------------------------------------------------------+
| Proc           | 2800/6346 | 4581/8001 | 4750/11564 | 0/184  | 0/128     |
| Cop            | 893/893   | 1988/1988 | 1043/1043  | 0/0    | 0/0       |
| DMemoryShared  | 111/209   | 68/111    | 162/323    | 0/24   | 0/64      |
| IMemory        | 183/234   | 20/94     | 236/370    | 0/48   | 0/64      |
| RFile          | 1035/1035 | 1099/1099 | 2786/2786  | 0/0    | 0/0       |
| Scoreboard     | 174/174   | 64/64     | 411/411    | 0/0    | 0/0       |
+-----------------------------------------------------------------------+
```

Other modules do not appear in this table because they are not currently using synthesis boundaries.

## 6 Design Exploration

We explored the tradeoffs of different design choices inside the processor. We looked at 5 main locations where bypassing can be implemented, between fetch and decode, between dispatch and execute, between execute and writeback, in the redirect fifo, and in the register file. We started with all of the locations bypassed and trimmed the critical path down until it cannot be trimmed anymore. Table 1 shows the configurations we are looking at.

| Processor | Dispatch to Execute | Redirect | Registerfile |
|---|---|---|---|
| A | BypassFifo | BypassFifo | Bypass |
| B | BypassFifo | BypassFifo | Normal |
| C | CFFifo | BypassFifo | Bypass |
| D | CFFifo | BypassFifo | Normal |
| E | CFFifo | CFFifo | Normal |

Table 1: Processor bypassing configurations for design exploration. Fetch to decode and execute to write back are bypass in all processors.

For all these configurations, the fetch to decode fifo is a bypass fifo, and the execute to write back fifo is a bypass fifo also. Processor E's critical path does not pass through either of those fifos, so it does not make sense to explore those design points.

Table 2 shows IPCs for each processor, and Table 3 shows IPS for each processor.

|  | IPC for Benchmarks |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| Processor | circadd | median | multiply | qsort | towers | vvadd | Clock Frequency |
| A | 1.96 | 1.12 | 1.56 | 1.22 | 1.07 | 1.65 | 41.2 MHz |
| B | 1.64 | 0.73 | 1.01 | 0.72 | 0.83 | 1.24 | 51.2 MHz |
| C | 1.63 | 0.66 | 0.97 | 0.69 | 0.80 | 1.23 | 59.5 MHz |
| D | 1.40 | 0.52 | 0.70 | 0.48 | 0.61 | 0.90 | 81.5 MHz |
| E | 1.39 | 0.51 | 0.68 | 0.48 | 0.57 | 0.90 | 92.5 MHz |

Table 2: Instructions per cycle (IPC) comparison

|  | IPS for Benchmarks |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|
| Processor | circadd | median | multiply | qsort | towers | vvadd | Clock Frequency |
| A | 80.8 M | 46.1 M | 64.3 M | 50.3 M | 44.1 M | 68.0 M | 41.2 MHz |
| B | 84.0 M | 37.4 M | 51.7 M | 36.9 M | 42.5 M | 63.5 M | 51.2 MHz |
| C | 97.0 M | 39.3 M | 57.7 M | 41.1 M | 47.6 M | 73.2 M | 59.5 MHz |
| D | 114.1 M | 42.4 M | 57.1 M | 39.1 M | 49.7 M | 73.4 M | 81.5 MHz |
| E | 128.6 M | 47.2 M | 62.9 M | 44.4 M | 52.7 M | 83.3 M | 92.5 MHz |
| Scalar |  | 72.4 M | 79.0 M | 74.6 M | 84.5 M | 91.1 M | 109.7 MHz |

Table 3: Instructions per second (IPS) comparison

From these results, we see that superscalar dominates scalar in IPC, but due to the penalty for superscalar executaion, scalar still beats superscalar in IPS for most benchmarks.

Other than the scalar processor, no super scalar processor stands out as better than the rest. There is not even a clear winner between processor A and processor E, even though they have very different configurations and clock frequencies.