High Performance SMIPS Processor

Jonathan Eastep
6.884 Final Project Report

May 11, 2005

1 Introduction

1.1 Description

This project will focus on producing a high-performance, single-issue, in-order, pipelined SMIPS processor. I will investigate branch prediction and use of load data before tag check as ways to improve performance over a baseline implementation without increasing the cycle time.

1.2 Approach

I will start by building a baseline, single-issue, in-order, 6-stage pipeline with full-bypassing, a 16KB direct-mapped instruction cache with a line size of 16 bytes, a 32KB write back/write allocate, direct-mapped data cache with a line size of 16 bytes, rudimentary predict-not-taken branch prediction (to make it easier to add other forms of branch prediction later), and a "perfect" dual-ported, unified L2 cache with a 6 cycle latency.

I will then build a powerful ISA simulator/debugger that can compare a trace from a simv run against expected ISA simulation values and verify that the Bluespec model is functioning as expected. Additional features will include symbolic and numeric break-points, run-to and jump-to capabilities, on-the-fly modification of architectural state (including instruction memory), and dumps of architectural state.

As a design space exploration, I will then build three additional machines that incorporate incremental improvements over the baseline implementation. The four machines will compare as follows:

- Machine A: the baseline implementation previously described.
- Machine B: add to Machine A the use of load data before data cache tag check.
- Machine C: add to Machine B a simple predict-backward-branch-taken, forward-not branch prediction scheme.
- Machine D: add to Machine B a branch history table of 2-bit saturating counters for branch prediction.

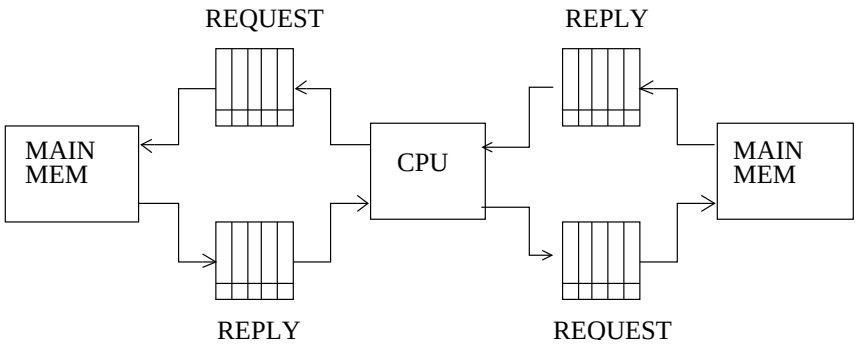

Figure 1: SMIPS Processor UTL Description Diagram

2 Unit Transaction Level Description

It is useful for high-level understanding to examine what I am setting out to build from the Unit Transaction Level (UTL). As depicted in Figure 1, the baseline machine (and the other three, in fact) may be characterized as two UTL units: the memory unit (shown twice in the diagram for convenience) and the CPU. Running between the units are several queues. Between the CPU and the memory unit are instruction memory request and reply queues. On the other side, between the CPU and the memory unit are data memory request and reply queues.

2.1 Processor State

The processor has four major state elements: the memory (instruction and data), the register file, the program counter, and the exception program counter. The instruction and data memory are in the UTL memory unit while the program counter, the exception program counter, and the register file are in the CPU.

2.2 UTL Transactions

2.2.1 Memory Unit – CPU

The CPU may make a request for instruction memory by enqueuing the PC and some tag information into the instruction memory request queue. The memory unit will dequeue the request, read from its memory state, and place a reply in the instruction reply queue. The CPU will then dequeue the reply to get an instruction, decode it, access the register file, and obtain any immediate values from the instruction itself. It will then perform an operation and potentially store a result as specified by the result of the instruction decode.

2.2.2 CPU – Memory Unit

The transactions between the CPU and the memory system involving data are similar to those involving instruction memory. The CPU will calculate a memory address in its ALU and enqueue a request in the data memory request queue. If the request was a load, the CPU will stall until it receives a reply. If it was a store, the CPU continues execution. The memory system will dequeue

the request the CPU made, identify its type, and if it is a load request, will read from its memory state and place a reply in the data memory reply queue. If the request was a store, the memory system will store the data in its memory state. If the CPU made a load request, it will wake up when it receives a reply from the memory system and complete the store into its register file state.

3 Test Strategy

The Bluespec model will make extensive use of the $write compiler directive to dump trace information. The trace will include such things as what instructions are in the various pipe stages, what their operands are, what the calculated results are, what address to load memory from or store memory to, what values are being inputted to bypass muxes, what the PC for an instruction in a given stage is, what value is being written to the register file, etc. Having a visible account of what is going on in the pipeline should help for basic debugging.

For debugging the processor against complex programs, however, I will write an ISA simulator to read in a trace from a run of the Bluespec model and test the trace against expected values looking for disagreements. This should guard against erroneous calculations, instructions entering commit that should have been squashed, errors in the dynamic instruction stream, etc. In addition to this diffing capability, the simulator will support symbolic and numeric breakpoints, run-to and jump-to capabilities, on-the-fly modification of architectural state (including instruction memory), and flexible dumping. These features should prove useful in debugging any benchmark programs that I write–if I have time to write extra benchmark programs, of course.

For benchmarking support in the hardware, I will create a module with methods for incrementing or resetting benchmark counters and dumping counter state. I will place all benchmarking related hardware in a module separate from the main processor so that it is easy to remove when doing synthesis and place and route. Having detailed benchmarking information such as the number of cycles the processor stalls for load hazards, for branch penalties, or for cache misses should help significantly when trying to figure out what changes to make to the pipeline to increase processor performance.

4 Micro-Architectural RTL Design

4.1 Baseline RTL

The baseline implementation consists of six stages: fetch, decode, execute, mem, tag check, and write back (see Figure 2). There are paths from execute, mem, tag check, and write back to decode where bypass muxing is performed. Since the SMIPS ISA always places the register specifiers for instruction operands in the same bits of the instruction, decode can read from the register file immediately without doing any decoding; decode exploits this fact.

Jump instructions resolve in decode and use the path from decode to fetch to communicate to fetch that there needs to be a PC update. For branches, the processor evaluates the branch condition and calculates the branch target in the execute stage. The path from execute back to

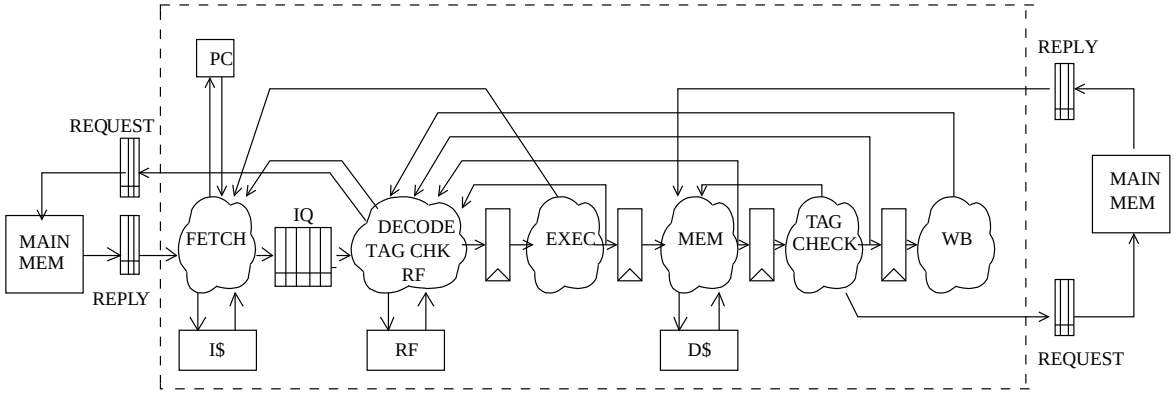

Figure 2: 6 Stage SMIPS RTL Model

fetch communicates to fetch that there needs to be a PC update in the event that a branch is taken.

The baseline implementation handles instruction and data caching, and the cache access is split into two stages: the cache SRAM access and the tag check. The fetch stage accesses the instruction cache SRAM and decode does the tag check. Decode starts using instruction cache data before the tag check completes, so decode will squash its result and the instruction to be placed in the instruction queue by the fetch stage if the tag check indicates that there was an instruction cache miss. On a miss, decode will issue a request to main memory for the line and stall. Fetch will dequeue the reply when it is available, and execution will continue.

For load and store instructions, the processor calculates the memory address in the execute stage; the processor uses the address in the mem stage to index into the data cache, and the next stage, tag check, checks the tag, initiating a cache miss and a memory request for the data in the event that the tags do not match. A data cache miss will stall the whole pipeline except for the portion of fetch that handles an instruction cache miss. When a reply comes back from the memory system, the tag check stage will update the data cache appropriately and execution will continue. If the instruction was a store, the data cache will be updated in the write back stage. Stores are bypassed to loads in the event that you have a load from the same address as a store that is still in the pipeline. If it turns out that there are two consecutive load/store instructions in the pipeline that access the same cache line and the first causes a miss, the data cache miss handler will patch up the second load/store instruction with the appropriate data when it finishes servicing the miss.

4.2 Baseline Bluespec Implementation

In the previous section, we examined the baseline RTL of the processor. This section focuses on the actual Bluespec implementation. The Bluespec represents each pipeline stage with a rule or a set of explicitly mutually exclusive rules. For example, the fetch stage is broken up into a rule that carries out the normal fetch operation of reading from the instruction cache SRAM and a rule that executes when servicing an instruction cache miss. Figure 3 gives a diagram for one of the

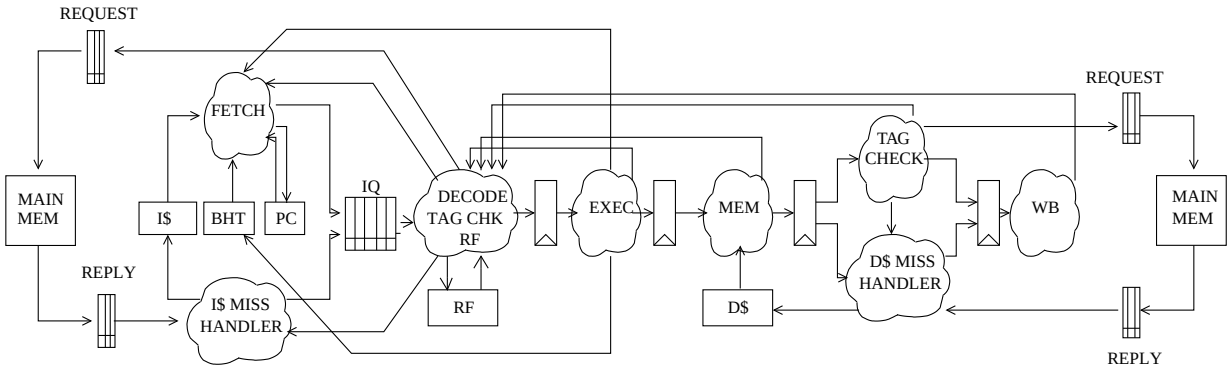

Figure 3: Bluespec Model for Machine D

machines I implement as part of my design space exploration (more on this later) that illustrates the mutually exclusive rules, and Figure 4 gives a rough idea of what the code looks like. The function serviceICacheMiss() contains combinational logic that evaluates whether or not the pipeline is currently servicing an instruction cache miss. In general, for clarity and readability, I try to use functions for the control logic that determines whether or not a rule will fire. The signals that serviceICacheMiss() and serviceDCacheMiss() use to determine what boolean value to return are set by the decode and tag check stages, respectively.

```
rule iCacheMissHandler( serviceICacheMiss() );
...
endrule
rule fetch ( !serviceICacheMiss() && !serviceDCacheMiss() );
...
endrule
```

Figure 4: Control Logic With Functions

Examination of Figure 2 and Figure 3 should raise concern over how rules communicate in the same cycle and execute without conflicts in Bluespec. The bypass paths and PC redirection associated with jumps and branches require same-cycle communication; I accomplish this through the use of RWire. And I make very liberal use of ConfigReg to eliminate the possibility of unexpected conflicts between rules that share a resource such as a pipeline register.

5 Design Space Exploration

For my design space exploration, I build three additional machines that incorporate incremental improvements over the baseline implementation. The four machines in all, again, compare as follows:

- Machine A: the baseline implementation previously described.

- Machine B: add to Machine A the use of load data before data cache tag check.
- Machine C: add to Machine B a simple predict-backward-branch-taken, forward-not branch prediction scheme.
- Machine D: add to Machine B a branch history table of 2-bit saturating counters for branch prediction.

5.1 Minimizing Load Hazards

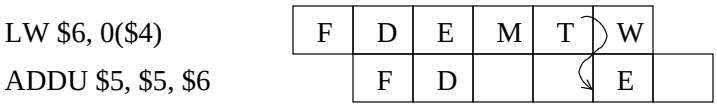

Figure 5: Load Hazard

Machine B address the fact that if there is an instruction in decode that depends on the result of an earlier load, the instruction in decode cannot reliably be allowed to progress until the load passes tag check. As indicated in Figure 5, the worst case can create a two-cycle stall. In Machine B, instead of paying the two-cycle stall, I build upon the fact that the data cache usually hits: we can somewhat speculatively use a value from the data cache SRAM before we have checked the tag. In our pipeline, this amounts to allowing bypassing from the mem stage to decode for loads. Figure 6 illustrates with a pipeline diagram.

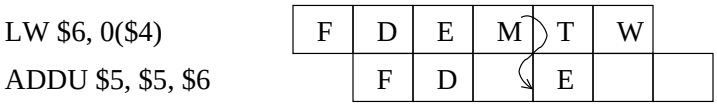

Figure 6: Load Hazard Reduced

In terms of additional bypass logic, there is little extra cost because the baseline already bypasses from the mem stage if the instruction in the mem stage is not a load. If anything, decode gets a little simpler because it now only has to look at the execute stage to see if there is a dependence on load data to determine whether or not to stall; a dependence found on an instruction in a later stage will not require a stall because the data will be available for bypass. The additional cost in our pipeline is in the squash logic. Since we allow instructions to begin execution based on load data that we will not know is valid until the cycle after it is potentially used, it may be necessary to squash instructions if we get a data cache miss.

As depicted in Figure 7, the tag check stage will need to be able to examine the instructions in previous stages to see what registers they sourced to see if any instructions need to be squashed. Instructions will now consequently need to carry their source information with them down the pipe. The baseline implementation already has logic in place to stall stages in the event of a data cache miss, so we exploit that and do the squashing writes to the pipeline registers in the data cache miss handler free of conflicts from the rules that would normally also be trying to write the pipeline

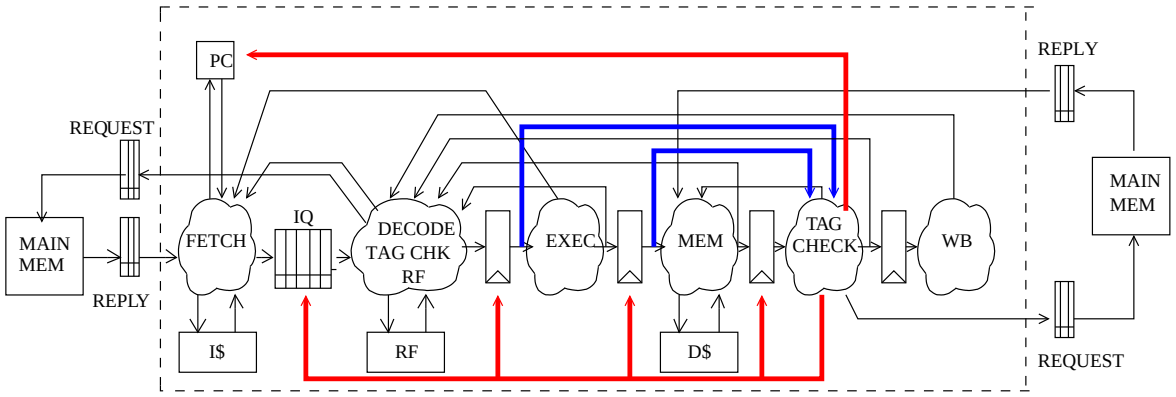

Figure 7: Early Load Use Pipeline Modifications

registers. Any squashed instructions must be replayed, so the data cache miss handler changes the PC to the PC of the earliest squashed instruction (the one closest to write back) if any instructions are squashed at all.

After placement and route and a comparison of the numbers for Machine A (our baseline implementation) and Machine B (our baseline with early load use enhancement) we see in Figure 8 that the area numbers and the timing are comparable. I should mention that the area numbers in these charts are inaccurate, but consistently so; I did not synthesize or place and route the cache memories. That said, the thing to note is that the IPC for one of our more memory intensive benchmarks improved considerably. I should mention that the improvement appears more dramatic than it is in actuality because the baseline implementation does not actually bypass a load value until write back even though it could obviously be done at the end of the tag check stage. This is a bug in the baseline implementation. IPC numbers for other benchmarks will follow in a later figure, but it looks like incorporating early use of load data is a design win nonetheless.

- Without early use of load data:
  - PR area: 632,471um^2
  - PR timing: 3.68ns, 272MHz
  - IPC (vvadd): .47
- With early use of load data:
  - PR area: 672,744um^2
  - PR timing: 3.81ns, 263MHz
  - IPC (vvadd): .70

Figure 8: PR Results for Early Load Use

5.2 Eliminating the Branch Penalty

The baseline implementation predicts that a branch is not taken, so in the case that that assumption is correct, the baseline already saves a cycle over a naive approach that just stalls until a branch resolves. When the assumption is wrong, however, the instruction in fetch must be squashed (see Figure 9). Machines C and D try to improve performance by making a better guess as to whether or not a branch is going to be taken, thereby reducing the number of cycles lost to squashing. They do so by adopting different prediction policies.

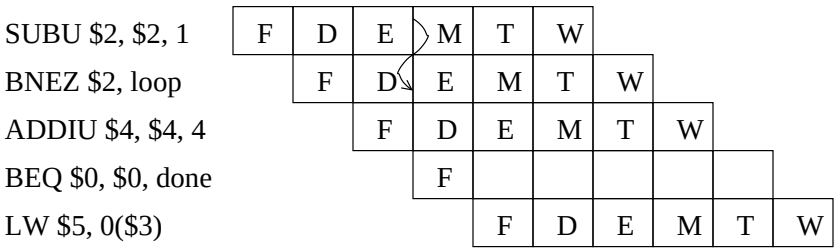

Figure 9: Branch Penalty

Machine C uses a simple but effective predict-backward-branches-taken, forward-not branch prediction scheme. To incorporate this form of branch prediction, the pipeline need undergo very little change. Jumps behave as they did in the baseline implementation since there is nothing to predict about a jump; it is always taken. But for branches, in decode, when we have identified that we have a branch, we look at the sign bit of the offset. If it is negative, we have a backward branch and we predict that it is taken; if the sign bit is positive, we have a forward branch and we predict that it is not taken. We communicate to fetch this information, but we also need to communicate the PC to jump to if the sign bit is negative. We therefore need to have determined the branch target before the instruction enters execute. This requires an additional adder in decode as you can see in Figure 10 (the block for branch predictor logic is shown as the general case; the simple scheme does not require any extra logic in fetch). If we predict that a branch will be taken and update the PC accordingly but determine in the next cycle in execute that we were wrong in our prediction, execute again redirects the instruction stream and squashes the instruction in fetch. The logic to do this already exists in the baseline (when a branch is taken the instruction in fetch is squashed), so there is no change to make there.

Machine D uses a 1024 entry branch history table of 2-bit saturating counters for the branch prediction. To make this modification to the baseline implementation, we add a branch history table SRAM in fetch. We read the SRAM in fetch then pass the counter value to decode (see Figure 10). Decode uses the branch prediction information to assert to fetch a branch taken or not and passes the information along to execute. In execute, if the prediction was wrong, we correct it by redirecting the PC in the usual way. But execute now has the additional responsibility of updating the counter in the branch history table. If the prediction was to take and that was correct, the counter is incremented, saturating at three of course. If the prediction was not to take and that was correct, the counter is decremented, saturating at zero. If the prediction was to take and that was wrong, the counter is decremented. If the prediction was not to take, and that was wrong, the

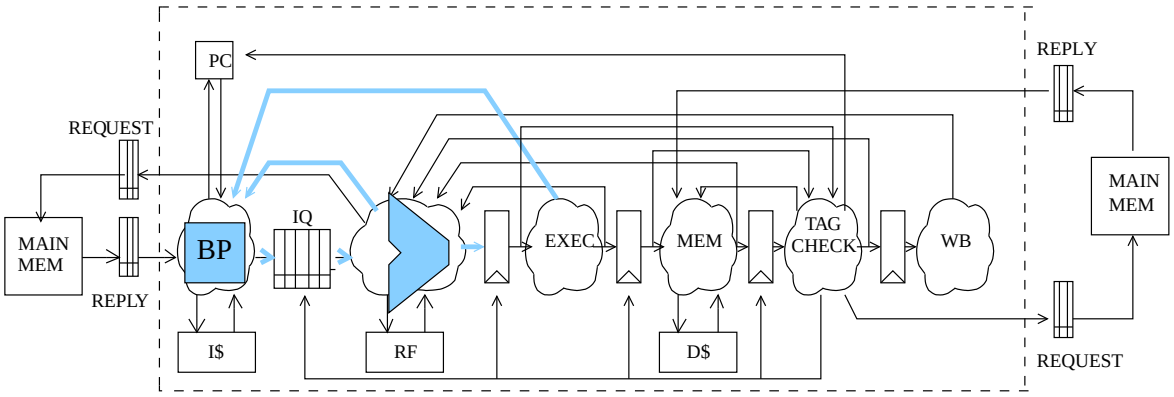

Figure 10: Branch Prediction Pipeline Modifications

counter is incremented. The increments and decrements are done in the common-sense way one would expect, and a counter value greater than or equal to two obviously correlates to a branch predicted taken.

- With early use of load data and predict backward taken, forward not:
  - PR area: 638,645um^2
  - PR timing: 3.35ns, **298MHz**
  - IPC (vvadd): .75
  - IPC (qsort): **.905**
- With early use of load data and 2-bit saturating counter branch prediction (1024 2-bit entries):
  - PR area: 766,642um^2
  - PR timing: 3.51ns, 285MHz
  - IPC (vvadd): .75
  - IPC (qsort): .906

Figure 11: PR Branch Prediction Comparison

As Figure 11 illustrates, place and route numbers show a significant increase in area associated with adding the 1024 entry branch history table (again the cache SRAMs have not been synthesized or placed and routed). These numbers are for machines that include logic for early use of load data, but I should mention two things: first, I cleaned up the Bluespec code before acquiring the placement and route numbers. This leads to a somewhat unfair comparison against the baseline implementation. I would have liked to have left the Bluespec code in a state consistent with that of the baseline implementation to make a fair comparison, but the compiler started running out of memory; I had to clean up the code to get it to compile again. Second, I would presume that the branch history table was synthesized and placed and routed rather inefficiently so the area increase shown is more dramatic than it should be with a good memory generator.

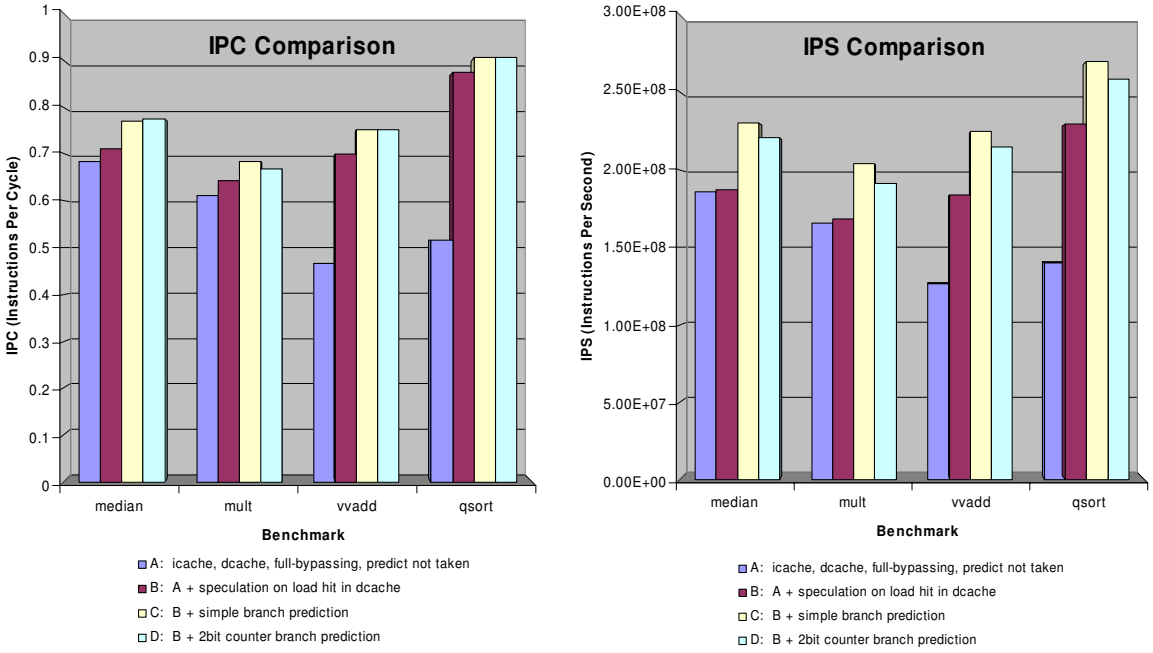

Figure 12: IPC and IPS Comparisons

It is interesting to note that the cycle time increased when I added the branch history table, and did so without a noticeable increase in IPC for the benchmarks available. For the benchmarks studied, therefore, we conclude that the branch history table implementation is actually suboptimal. Nevertheless, having a smarter form of branch prediction–whether it be the simple predict-backward-taken, forward-not or 2-bit saturating counters–seems to give a bit of a performance boost over blindly predicting not taken. The boost is not very dramatic, of course, because the processor front-end is rather short. Figure 12 gives a nice summary of the IPC and the Instructions Per Second (IPS) of the machines studied. IPS takes into consideration the place and route timing data to give a better metric for performance than the IPC. And lastly, Figure 13 gives a performance comparison of the various machines against the baseline implementation (whether or not it is a fair fight). The normalized performance is the ratio of the IPS of a given machine to the baseline's IPS.

6 Conclusion

The first take-away point is that it is not IPC that really matters. As we saw with the simple predict-backward-branch-taken, forward-not scheme versus the 2-bit saturating counter scheme, while the 2-bit counters may have yielded slightly higher IPC, the cycle time increased, making the 2-bit counters suboptimal for the benchmarks studied. The IPS is a better metric because it incorporates placement and route timing information. The idea is well-known, I am certain, but this experiment definitely highlights the point. I do feel like I owe it to the branch history table to

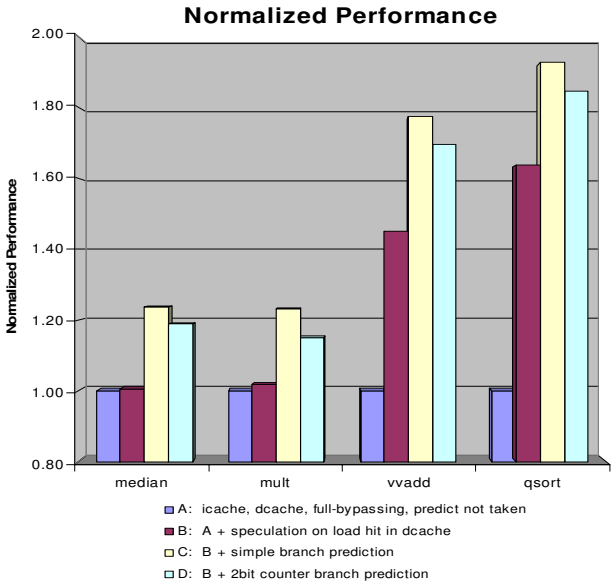

Figure 13: Normalized Performance

run some benchmarks that exemplify when it proves the victor over the simpler scheme, but I did not have time at the time of this writing.

The second point to make is that this design turned out to be pretty high performance as I hoped! I have never built a machine before, obviously, but an IPC of .75-.90 on the reasonable benchmarks supplied with a fairly realistic 6-cycle latency L2 model, seems pretty good considering the fact that one cannot do any better than an IPC of 1 with a single-issue machine. I was also impressed by how fast the design turned out. I was weary of the scheduler logic the Bluespec compiler was going to produce, but was pleasantly surprised when, after cleaning up my Bluespec code, I pushed my design through place and route, achieving a clock frequency of nearly 300MHz. It has certainly been a fun, rewarding experience, and I feel like I have come to grasp intimately the basic principles of computer architecture. That said, thank you to the 6.884 staff for their guidance and patience.

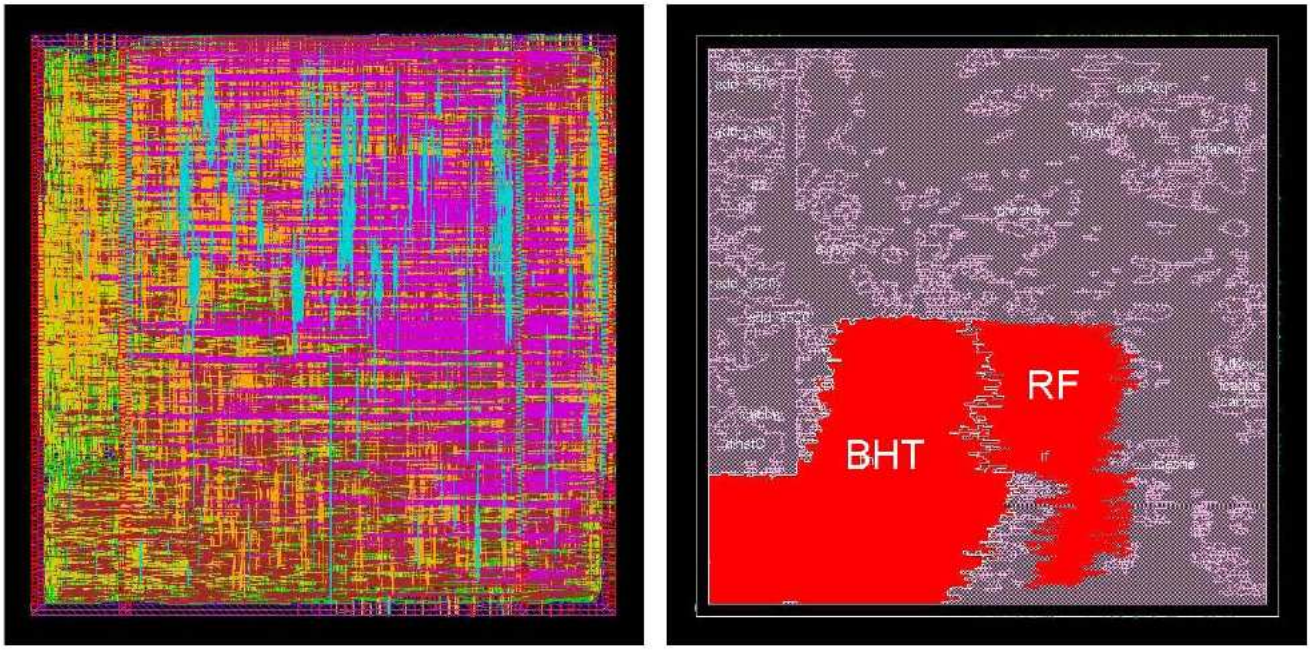

Figure 14: Final Layout