Spring 2004 CS 152
Final Project

8-Stage Deep-Pipelined MIPS Processor

Members:
Otto Chiu (cs152-ae)
Charles Choi (cs152-bm)
Teddy Lee (cs152-ac)
Man-Kit Leung (cs152-al)
Bruce Wang (cs152-am)

Table Of Contents

0. Abstract

The goal of our final project was to improve the performance of the 5-stage pipelined processor from previous labs. Aiming at this goal, we converted our processor into a 8-stage deep-pipelined one (22 pt.). Since an increase in the number of branch delay slots is an intrinsic drawback to adding more pipeline stages, we decided to add a branch predictor to cut down the number of stalled cycles in most cases (8 pt.). This problem also appears during a `jr` instruction. Thus, we installed a jump-target predictor (8 pt.) to abate the stalls associated with the instruction. In addition, we implemented a write-back cache (7 pt.) and added a victim-cache (4 pt.) to minimize first-level cache miss penalties. Finally, we added to our multiplier/divider the ability to handle signed numbers (4 pt.). Our project implemented a total of 53 pt. out of the required 37.5 pt. for our group.

We implemented our design successfully and thoroughly. We noted significant improvement in performance. The final clock speed for our processor is 27 MHz.

1. Division of Labor

The project specifications allowed us to split up the work by the major components: branch predictor, write-back cache, victim cache, jump-target predictor, and signed multiplier/divider. The entire group was involved in changing and verifying the existing design. Here is a more detailed division of labor:

Otto: Branch predictor, signed `mult/div`
Charles: Branch predictor, memory controller
Teddy: Datapath, `mult/div`, testbenches
Man-Kit: Write-back cache, victim cache
Bruce: Write-back cache, jump-target predictor

2. Detailed Strategy

2.1 Datapath.

We noticed from lab 5 that our critical path involves the memory components. Thus, we knew that we would need to concentrate in splitting the memory paths. From the timing analyzer for lab 5, we saw that it took more than 10ns to perform lookup on memory. In order to achieve the 28ns cycle time requirement, we began by splitting up each memory stage because the timing analysis tool told us that those stages together with forwarding paths took significantly more time than other stages. The idea here is to progressively split up the stages with long critical paths. Since a lot of work is involved in splitting a stage, we cut the critical paths by shifting components across stages whenever possible as an alternative. By doing this, we potentially introduced extra cycle delays, but this is partially remedied by the higher clock speed. We have split up our pipeline into the following stages:

IF	PR	ID	EX	MR	MW	WB	FW

Figure 1: Pipeline Stages

2.2 Stage Summary

IF: instruction fetch
PR: predict and register
- Branch predict
- Jump-target predict
- Register file read/write

ID: decode
- Mult/Div

EX: execute
- Resolve branches

MR: memory read
MW: memory write
WB: write back
FW: forward
- forward WB value to ID stage

We initially split up our memory stages to a tag-lookup stage and a data-lookup stage. However, we soon moved the data-lookup parallel to the tag-lookup and registered the data in the MW stage because we found that the critical path happened with data-lookup + forward logics. By moving data-lookup one stage earlier we can split up data-lookup from the forwarding logics. In addition, the routing time is shortened.

2.3 Stage Details

IF:
The instruction is fetched in this stage. We simultaneously look up the tag file and the

instruction cache and then check the tag to determine whether we should output it or stall. Our original design was doing only the tag lookup in this stage, but we realized that we could lookup the instruction at the same time.

PR:
The register file is located here. It was moved from the ID stage to here because of the amount of forwarding logic in the ID stage. We also do our branch and jump-target prediction in this stage.

ID:
We decode the instruction in this stage. Most of the forwarded values are forwarded to this stage. Our multiplier and divider are also located in this stage.

EX:
Arithmetic operations except for multiplication and division in this stage. We also resolve the branch and jump predictions we have made in this stage.

MR:
Like the IF stage, we could do a tag lookup and a cache read in the same cycle. But since we need to know whether there has been a cache hit before we can write to cache, we cannot perform stores in this stage.

MW:
We perform stores to cache in this stage. When we have a sequence of store followed by a load to the same address, the store and the load would be happening on the same cycle. Since the result from writing and reading from the same line at the same time in a RAM is undefined, we could not let the cache handle this case by itself. But when we have a store followed by a load to the same address, we know that the value loaded definitely should be the value that we stored. Thus, we simply latch in the value of the store, and output the value on the load.

WB:
A typical write back stage writing results back to register. We also have forwarding paths from here to ID and EX.

FW:
This stage only forwards values to the ID stage. We decided to add this stage because we wanted to forward to the ID stage instead of the PR stage. The details are explained below.

2.4 Forwarding Paths

The forwarding paths that we had in lab 5 stayed in our design. Because we added 3 stages to our pipeline, we had to introduce more forwarding paths. Spliting the data cache stage into MR and MW stages meant that we had to add a forwarding path from the MR stage to the ID stage, as well a path from the MR stage to EX stage.

We added a forwarding path from the FW stage to ID to handle the case where there are 5 instructions between the write and the read. Since our register lookup is in the 2^{nd} stage of the pipeline, and our write back stage is the 7^{th} stage, the register lookup would occur before the register read.

But since we did not want to increase the length of the PR stage and WB stage, we decided to add the FW stage to forward it back to the ID stage. Since the register file is in the PR stage now, the ID stage only contains the controller and forwarding muxes. Therefore forwarding to the ID stage seems most appropriate.

2.5 Branch Instructions.

Instead of resolving branches in the ID stage to get exactly one branch delay slot, we needed to move the branch comparators into the EX stage. This is done to cut the levels of logic in the ID stage. However, this has introduced 2 extra delay slots in addition to the one specified by the ISA. This problem led us to put in a branch predictor. We will discuss our implementation of such a predictor later.

2.6 Branch Predictor

In this project, we experimented with 5 different kinds of branch prediction scheme:

1. GAs
2. Gshare
3. Gshare/Bimodal
4. Always true
5. Always false

We will give the performance of each predictor in some of our test benches, but we will first describe how we implemented each. Since an always true or always false branch predictor is trivial to implement, we will only show the statistics for them.

We used the simple 4-state FSM in Figure 2 to predict whether to take branch or not. The transitional edges are labeled by the actual resolved branch.

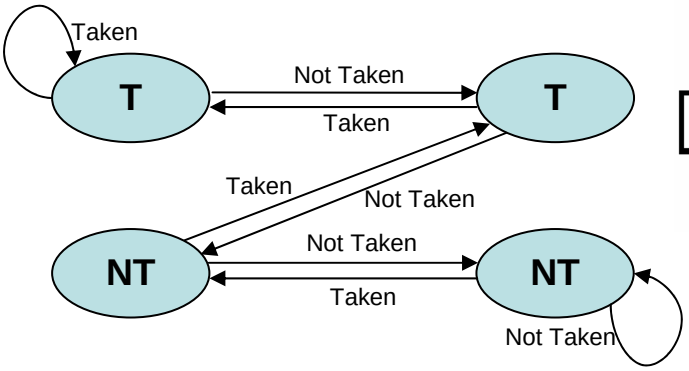


Figure 2: Branch Prediction FSM

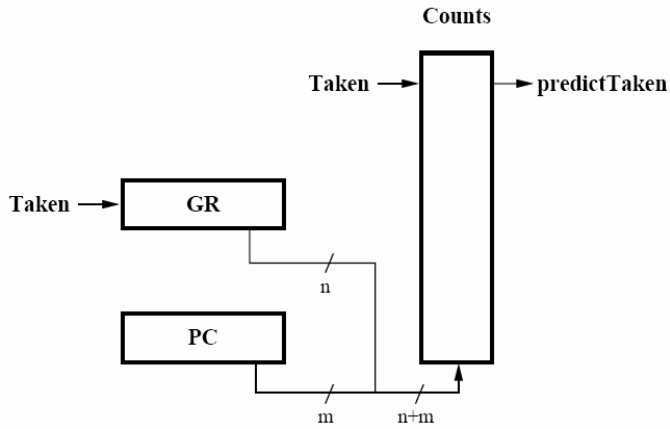


Figure 3: GAs

2.6.1 GAs:

In this prediction scheme, we use a global branch history register (GBHR) to pick from the set of local predictors (see Figure 3). We experimented with several different combinations of global history-local table sizes and found that increasing the GBHR size does not always translate into better predictor accuracy. This is because it takes the predictor a longer time to learn the global pattern for each branch.

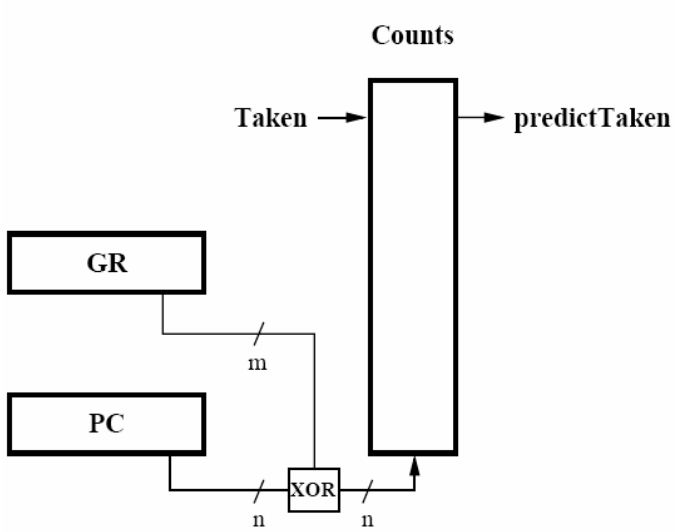


Figure 4: Gshare

2.6.2 Gshare:

To reduce the aliasing problem associated with the GAs predictor, the gshare predictor XORs the upper bits of the indexing PC with the GBHR to produce a new branch history table (BHT) index. This scheme seems to perform slightly better than GAs. A 7-bit GBHR with 512 entry BHT performed the best in our tests.

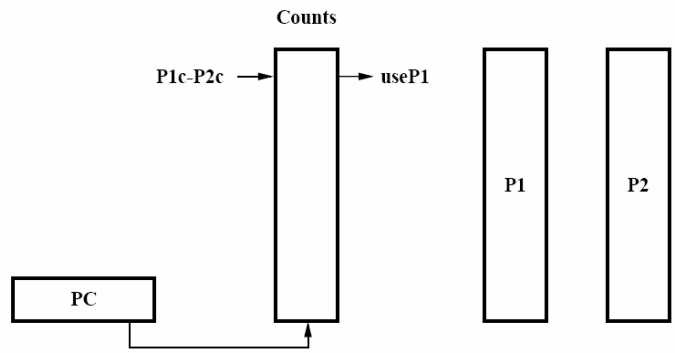


Figure 5: Combined Bimodal/Gshare

2.6.3 Combined Bimodal/Gshare.

This predictor is actually made up of two separate predictors: bimodal and gshare. It contains an extra table that holds the history to which predictor is doing better at a particular branch. Base on that information, it will return the prediction of the better-performing predictor. This special table (labeled as Counts in Figure 5) contains a FSM similar to that of Figure 2. This

predictor can be implemented to take just as much time as the bimodal or gshare predictors. This can be done by reading the 3 tables simultaneously and selecting the desired output asynchronously based on Counts. To our surprise, this prediction scheme did not outperform gshare predictor. We conclude that this is dependent on the application and may not be true in general.

2.7 Write-Back Cache.

We already had this working in lab 5. Basically, we added an extra dirty bit to each cache set to determine whether a line about to be overwritten needs to be written back to memory.

2.8 Victim Cache & Write Buffer.

The victim cache served as a small backup to reduce the penalty of conflict misses. It is used as a write buffer as well, thus in addition to the valid bit, a dirty bit is stored in the victim cache as well. Our victim cache contains 4 lines of data, and 8 words each line. We used registers to implement our victim cache, tag file and the LRU file associates with it. The victim cache costs us:
(4 entries x (8 words + 27bit tag)) + LRU files. The victim cache is fully associative, so we need a LRU file to keep track of which line to be replaced. The replacement policy for our victim cache is LRU. The LRU module is composed of 4 registers connected in series. When read or write to a particular LRU, the write enable signals below the LRU block being written are turned on. For example, if we're using the top LRU block in Figure 6, Write Enable 0 to 3 are all turned on. If the second block is being updated, then Write Enable 1 to 3 are turned on.

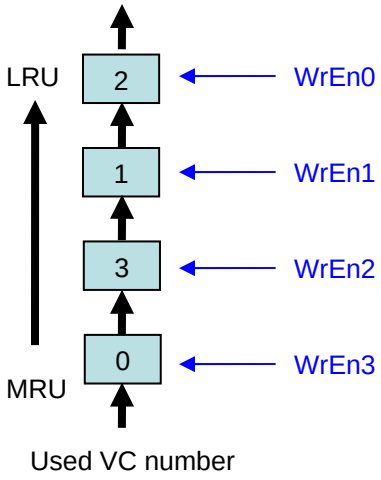


Figure 6: 4 Entries LRU file for Victim Cache

When there's a victim cache hit, we have to swap the data inside our victim cache with the data being kicked out from the data cache. This is done by checking if it is victim cache hit in our first data cache access stage(DT), and then swap the data in the second data access stage(DF). By doing so, we don't have to stall for a memory access if it is a victim cache hit.

2.9 Jump Target Predictor.

Since most jr instructions in programs are jumps to $ra, we decided that we would simply always use the address in $ra as the predicted target. In order to do this without stalls, we added an extra output port to the register file that outputs the value in $ra. The accuracy of our predictor is shown in the Results section.

2.10 (Un)Signed Multiply/Divide.

We built on our multiply/divide unit from Lab 4. Instead of re-implementing our multiply algorithm which calculates x0, x1, x2, and x3 in 1 cycle, we simply added the 2-bit Booth algorithm to handle signed multiplication. Adding the signed divide functionality to the unit required much less work because we only need to initialize the divisor and the dividend, reuse the existing divider, and fix the signs of the quotient and remainder at the very end. The MIPS convention to signed divide is described in the COD text.

In our original design in which `multu` took 1 cycle to compute x0, x1, x2, and x3, we found out that the coprocessor became part of the critical path. This occurs when one of the operands uses the forwarded value. To fix this problem, we decided to add registers to hold the value of the multiplicand and multiplier, which effectively end the critical path at the input to the unit. The tradeoff is that the entire mult/div coprocessor will take an extra cycle to complete because it will not start until the EX stage.

3. Testing

3.1 General testing.

Most of the modules we used in this lab are inherited from lab5, and the only new modules we have added in this lab are the victim cache and the branch predictor. The other big change to our processor is the 7-stage pipeline, and we expect it to have a lot more data hazards than the 5 stage pipeline we had. Our methodology is to build up the 7 stage pipeline, test it thoroughly with all the test programs from lab4 and lab5, and only after it is fully bug free then we will split the pipeline again for better clock speed. This can speed up the testing phase when we split up the stages later because most of the changes (cache controls, forwarding, hazard detection) make to the 5-stage pipeline are done in the 7-stage version.

3.2 Victim Cache test.

This is a very stressful load/save test. It has a couple of sequences of lw / sw that will request the Victim Cache to continuously swap data with the Data Cache, and then force feed the Victim Cache with sw to fill it up with Dirty lines so that the Memory Arbiter to do writes to the SDRAM to free up the Victim Cache.

With this test we found out that there is a problem with the dual port BlockRAM created with CoreGen. If we assign read address and write address at the same time the data out from the BlockRAM will go don't care (xxxx) afterwards. We have to create our own LRU file to fix this problem because parallel read and write operation is needed if we have to read and write from the same cache line in the same clock cycle (A write following a read in the pipeline will cause that).

3.3 Branch Predictor.

We simply used the quick sort program that is given to us after lab 5 checkoff. Quick sort has a much more irregular branch pattern, comparing to the other tests where most of the branches are taken.

3.4 (Un)Signed Multiply/Divide.

Because we implemented the 2-cycle fast `multu`, we needed to test those cases thoroughly. To make sure that each 4 of the instructions work correctly, we had 4 separate testbenches that feed random values to the coprocessor 2048 times. We had an option on the testbench to turn on debugging, which shows the results of each operation, but we also have an error count that tells us the number of errors resulted from the tests.

4. Results

Our processor is the only one that is able to run the original race program OGR the Golomb Ruler program correctly (at 27MHz) with correct results. Even though we lost to the Group One in the Corner race (2:34 to 2:51 in real time), but we won in terms of CPI.

Group One: CPI x Instruction Count / 36MHz = 154sec
Our group: CPI x Instruction Count / 27Mhz = 171sec

So Group One's CPI / Our group = 1.2

Our critical path is from the data tagfile to data cache control, back to the instruction cache control, and then back to the instruction cache. This is the special level 0 boot logic we did in lab 5 to avoid instruction compulsory misses during program execution.

4.1 Write-Back & Victim Cache:

The increase in performance with the addition of the Write-Back and Victim cache was obvious, since we had already implemented our WB cache and a semi-Victim cache in our lab5 (**semi-Victim because, in lab5, we only kicks out dirty cache lines to the victim cache, the

write buffer. And the write buffer actually brings back cache line to the Data cache if there is a hit in the write buffer). Our cycle count, in lab5, was much less than the other two groups without WB and V caches. The speedup was on a magnitude of around 2 to 3 (other group's cycle count for quick_sort 0x4d55 vs. our group's 0x1d40).

4.2 Branch Predictor

Accuracy:

Test / BP	Always true	Gshare	GShare/Bimodal
cycle	75.7%	81.1%	81.1%
base	98.6%	98.6%	98.6%
hammer	99.0%	99.0%	99.0%
q_sort	51.4%	65.1%	64.9%
extra	97.6%	97.0%	96.8%

An accurate branch predictor is very important to our processor. Without a branch predictor, our pipeline would need to stall for 2-cycles in addition to the delay slot. In the case of correct prediction, we would not need to stall, but a misprediction translates into 2-cycles of stalls, excluding the delay slot.

Take quick sort as an example, if we use an always true predictor, the cycle count is 0x1b33. Whereas our gshare predictor required only 0x1a34.

We chose to use the gshare predictor instead of the combined gshare/bimodal one because gshare was more accurate in the cases above. In addition, they produce similar results, but the combined predictor uses significantly more resources than gshare alone. That is why we chose gshare.

The high-level implementation illustrations are taken from the following source:

''Combining Branch Predictors.'' Scott McFarling. WRL Technical Note TN-36, June 1993.

4.3 Deep Pipelining

A well-made deep pipeline decreases the critical path, such that the time spent in each stage is nearly the same. In that case, the CPI increases, but the clock rate increases by a much greater factor. Thus gaining a speedup overall. With the exception of the FW stage, we were able to spread the time spent in each stage evenly.

When we ran quicksort, we gained a speedup of 2.32.

Lab5: 5328 cycles / 9MHz
Final: 6911 cycles / 27MHz
Speedup = Lab5 / Final = 2.32

5. Conclusion

We underestimated the delay from fetching data from block rams and store it into block rams. In our original design, we did the tag check in the first stage of 2 memory access stages and based on the hit or miss signal to select the data requested. However, this kind of approach gave us a RAM to RAM critical path of 36ns (27.7MHz). That is far from our desired clock frequency (35MHz). If we have more time, we could have removed our present critical path by only saving the instructions to the data cache and flushing them out to DRAM before we exit level 0 boot. We estimate that this could bring our clock speed to above 35 MHz. We anticipate our next critical path is in the execute stage, which is shorter than 27 ns.

The lesson that we learned from this project is to start with a simple design. Also we learned that implementing more features does not necessarily translate into higher performance. Personally, we all learned to work together as a group and had great fun.

6. Appendix I (Notebooks)

Here are the links to each group member's notebook along with the total number of hours each person spent on the project.

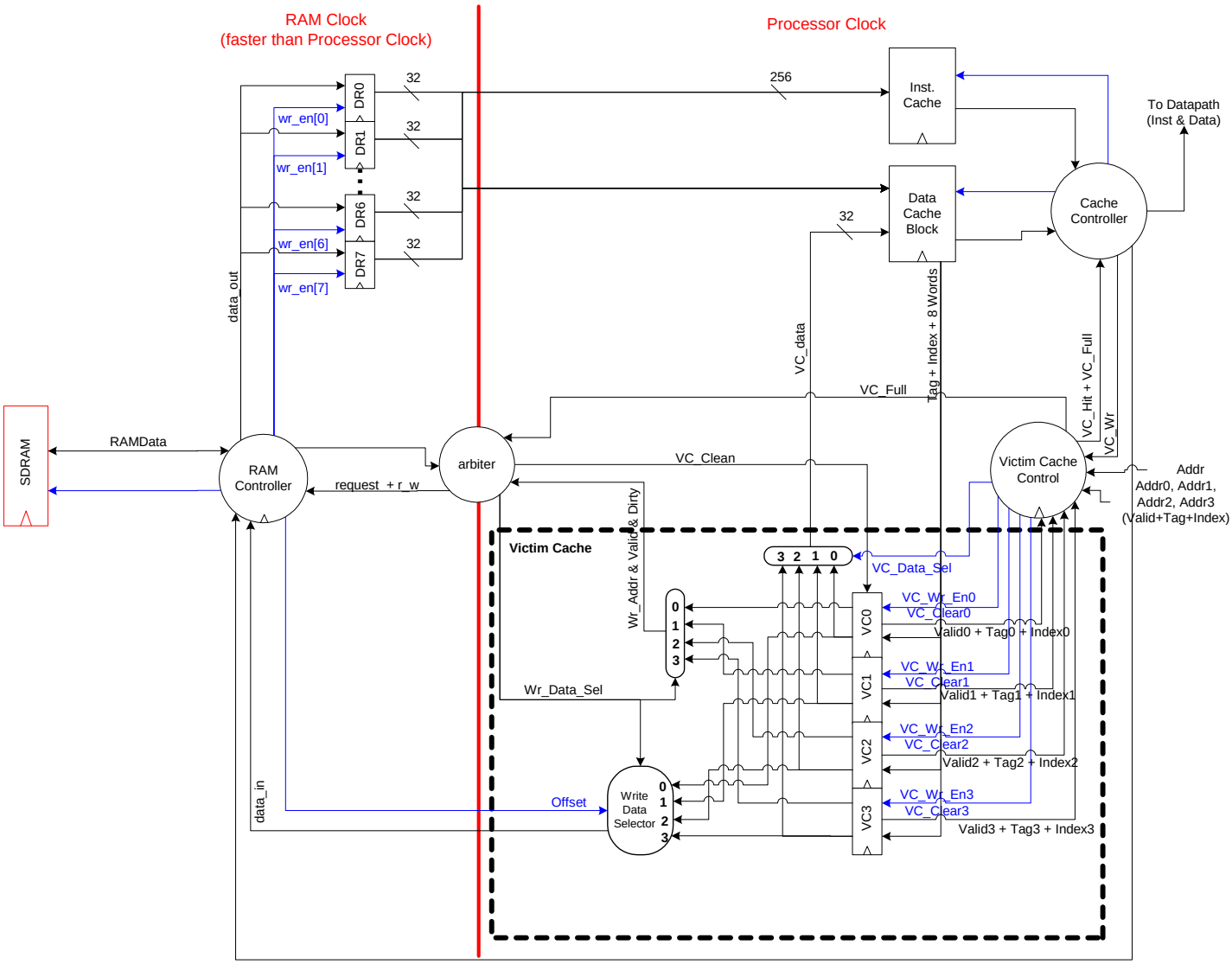
Otto Chiu, 140 hours
Charles Choi, 140 hours
Teddy Lee, 147 hours
Man-Kit Leung, 143 hours
Bruce Wang, 150 hours

We have spent a total of 720 hours as a group.

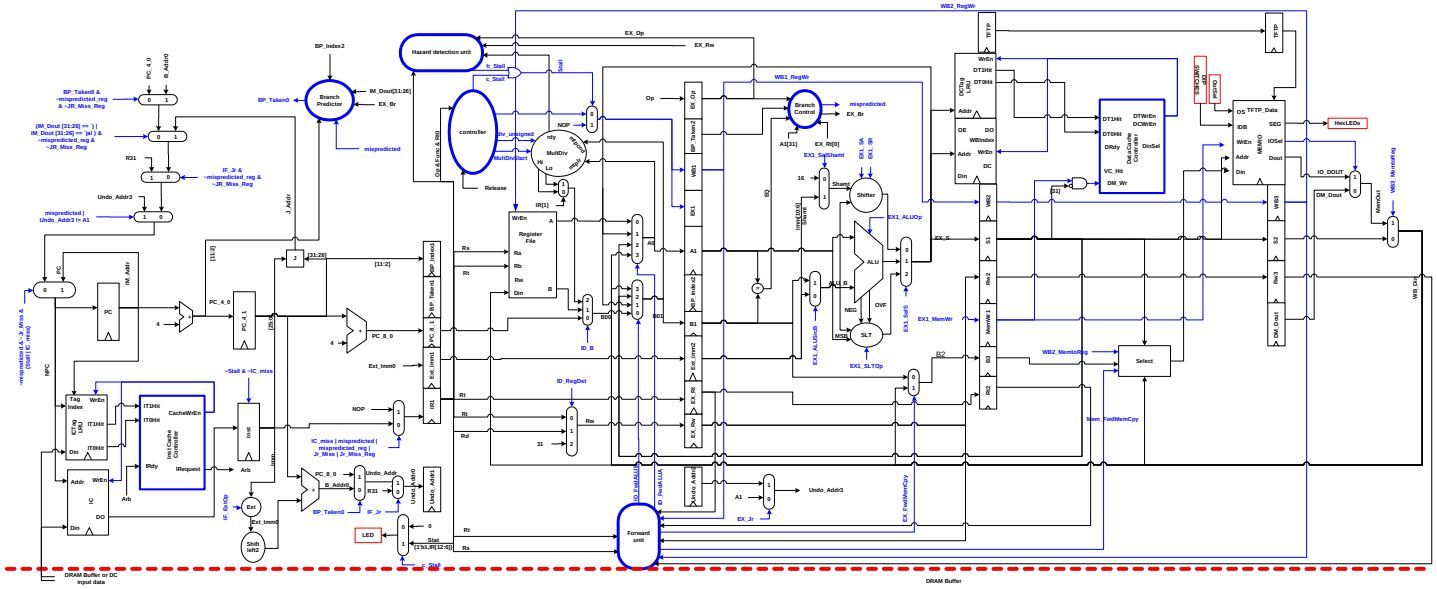
7. Appendix II (Schematics)

We used Visio as our schematic editor. Here are the links to the files as well as their screenshot.

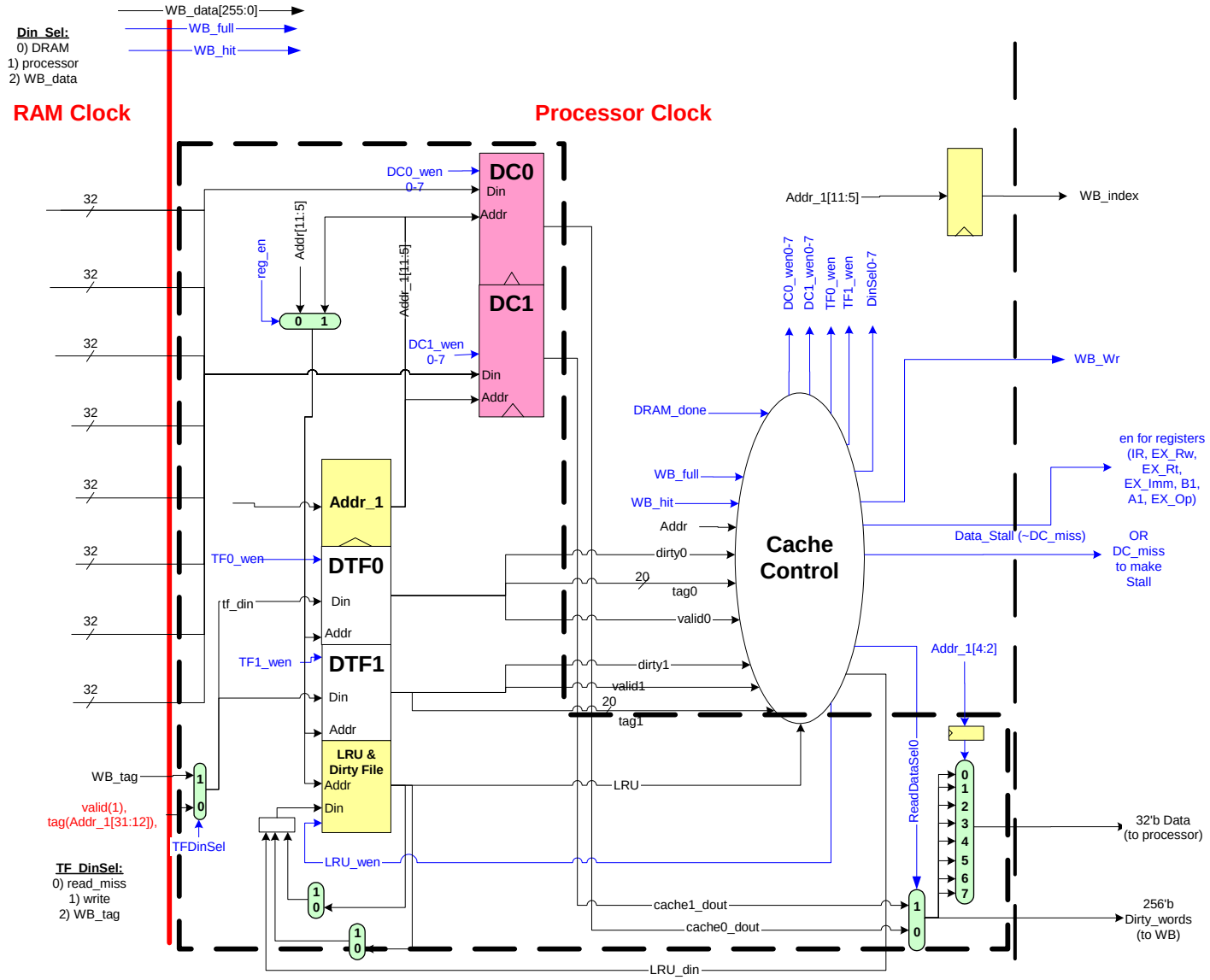
Victim Cache and Memory System (for crossing clock boundary) [cache_design.vsd]



8-Stage Deep Pipeline Datapath (datapath.vsd)



Data Cache (also to cross clock boundary) [dc_detail.vsd]



8. Appendix III (Verilog Files)

alu.v	ascii_romv1.1.v	bht.v
bin2HexLED.v	bootrom.v	bp.v
BP_Resolver.v	bts32.v	cache.v
cache_ctrl.v	controller.v	counter.v
datapath_v.v	data_mem.v	data_tagfile.v
debouncer.v	def.v	dinselect.v
D_cache.v	VC_Control.v	edge_detector.v
extend.v	forward.v	fpga_top2.v
hdetect.v	HiLoReg.v	inst_mem.v
inst_tagfile.v	I_cache.v	I_cache_ctrl.v
j.v	lab4group02blackbox.v	lru_file.v
memio.v	memory_arbiter.v	memory_control.v
monitor.v	mt48lc16m16a2.v	multAdder.v
multControl.v	multDatapath.v	mx2.v
mx3.v	mx4.v	mx5.v
ram128_32bit.v	ram128_32bit_dual.v	reg32.v
regfile.v	shifter.v	slt.v
VC_LRU.v	tftp_mem.v	top_level.v
util.v	VC.v	VC_Block.v

9. Appendix IV (Test Files)

testCache.v	testDiv.v	testDivu.v
testFM.v	testMult.v	testMultu.v
base_v1.0.s	beq.s	cycle_v1.1.s
extra_v1.0.s	hammer_v1.0.s	quick_sort_v1.0.s
test2.s (test instr. cache)	test3.s (test LRU for cache)	testSigned.s
testUnsigned.s	unfair_testbench.s	d_cache_tb.v
test_BP.v		

10. Appendix V (Timing)

```
================================================================================
Timing constraint: Default period analysis for net "processor_clock"

 7782172 items analyzed, 0 timing errors detected. (0 setup errors, 0 hold errors)
 Minimum period is  36.799ns.
--------------------------------------------------------------------------------
Delay:                  36.799ns (data path - clock path skew + uncertainty)
  Source:               DP/S1_dff/Q[7] (FF)
  Destination:          CACHE_UNIT/dc/dtf/B5.A (RAM)
  Data Path Delay:      36.762ns (Levels of Logic = 12)
  Clock Path Skew:      -0.037ns
  Source Clock:         processor_clock rising
  Destination Clock:    processor_clock rising
  Clock Uncertainty:    0.000ns

  Data Path: DP/S1_dff/Q[7] to CACHE_UNIT/dc/dtf/B5.A
    Delay type         Delay(ns)  Logical Resource(s)
    -------------------------  -------------------
    Tcko                  0.992   DP/S1_dff/Q[7]
    net (fanout=50)       2.878   S1[7]
    Topcyg                1.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_99
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_91
    net (fanout=1)        0.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_91/O
    Tbyp                  0.149   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_82
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_109
    net (fanout=1)        0.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_109/O
    Tbyp                  0.149   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_100
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_55
    net (fanout=1)        0.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_55/O
    Tbyp                  0.149   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_46
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_73
    net (fanout=1)        0.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_73/O
    Tbyp                  0.149   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_64
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_19
    net (fanout=1)        0.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_19/O
    Tbyp                  0.149   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_10
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_37
    net (fanout=1)        0.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_37/O
    Tbyp                  0.149   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_28
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_118
    net (fanout=1)        3.877   CACHE_UNIT/dc/M_VCC/I_118_n_1
    Tilo                  0.468   CACHE_UNIT/dc/M_VCC/VC_Hit1
    net (fanout=20)       2.357   CACHE_UNIT/dc/VC_Data_Sel[1]
    Tilo                  0.468   CACHE_UNIT/dc/M_VCC/VC_Hit
    net (fanout=26)       5.002   CACHE_UNIT.VC_Hit
    Tilo                  0.468   CACHE_UNIT/dcc/request_0_i_0_or6
    net (fanout=5)        1.153   CACHE_UNIT/dcc/N_428
    Tilo                  0.468   CACHE_UNIT/dcc/G_286
    net (fanout=11)       4.903   CACHE_UNIT/N_497
    Tilo                  0.468   CACHE_UNIT/dc/un1_reg_en
    net (fanout=2)        8.948   CACHE_UNIT/dc/un1_reg_en_n
    Tbeck                 2.418   CACHE_UNIT/dc/dtf/B5.A
    -------------------------  -------------------------
    Total                36.762ns (7.644ns logic, 29.118ns route)
```

```
                                (20.8% logic, 79.2% route)

--------------------------------------------------------------------------------
Delay:                  36.786ns (data path - clock path skew + uncertainty)
  Source:               DP/S1_dff/Q[7] (FF)
  Destination:          CACHE_UNIT/dc/dtf/B9.A (RAM)
  Data Path Delay:      36.745ns (Levels of Logic = 12)
  Clock Path Skew:      -0.041ns
  Source Clock:         processor_clock rising
  Destination Clock:    processor_clock rising
  Clock Uncertainty:    0.000ns

  Data Path: DP/S1_dff/Q[7] to CACHE_UNIT/dc/dtf/B9.A
    Delay type         Delay(ns)  Logical Resource(s)
    -------------------------  -------------------
    Tcko                  0.992   DP/S1_dff/Q[7]
    net (fanout=50)       2.878   S1[7]
    Topcyg                1.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_99
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_91
    net (fanout=1)        0.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_91/O
    Tbyp                  0.149   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_82
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_109
    net (fanout=1)        0.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_109/O
    Tbyp                  0.149   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_100
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_55
    net (fanout=1)        0.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_55/O
    Tbyp                  0.149   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_46
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_73
    net (fanout=1)        0.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_73/O
    Tbyp                  0.149   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_64
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_19
    net (fanout=1)        0.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_19/O
    Tbyp                  0.149   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_10
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_37
    net (fanout=1)        0.000   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_37/O
    Tbyp                  0.149   CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_28
                                  CACHE_UNIT/dc/M_VCC/un2_VC_Hit1_0.I_118
    net (fanout=1)        3.877   CACHE_UNIT/dc/M_VCC/I_118_n_1
    Tilo                  0.468   CACHE_UNIT/dc/M_VCC/VC_Hit1
    net (fanout=20)       2.357   CACHE_UNIT/dc/VC_Data_Sel[1]
    Tilo                  0.468   CACHE_UNIT/dc/M_VCC/VC_Hit
    net (fanout=26)       5.002   CACHE_UNIT.VC_Hit
    Tilo                  0.468   CACHE_UNIT/dcc/request_0_i_0_or6
    net (fanout=5)        1.153   CACHE_UNIT/dcc/N_428
    Tilo                  0.468   CACHE_UNIT/dcc/G_286
    net (fanout=11)       4.903   CACHE_UNIT/N_497
    Tilo                  0.468   CACHE_UNIT/dc/un1_reg_en
    net (fanout=2)        8.931   CACHE_UNIT/dc/un1_reg_en_n
    Tbeck                 2.418   CACHE_UNIT/dc/dtf/B9.A
    -------------------------  -------------------------
    Total                36.745ns (7.644ns logic, 29.101ns route)
                                  (20.8% logic, 79.2% route)

--------------------------------------------------------------------------------
Delay:                  36.707ns (data path - clock path skew + uncertainty)
  Source:               CACHE_UNIT/dc/dtf/B5.B (RAM)
  Destination:          CACHE_UNIT/dc/dtf/B9.A (RAM)
```

```
  Data Path Delay:      36.694ns (Levels of Logic = 8)
  Clock Path Skew:      -0.013ns
  Source Clock:         processor_clock rising
  Destination Clock:    processor_clock rising
  Clock Uncertainty:    0.000ns

  Data Path: CACHE_UNIT/dc/dtf/B5.B to CACHE_UNIT/dc/dtf/B9.A
    Delay type         Delay(ns)  Logical Resource(s)
    -------------------------  -------------------
    Tbcko                 3.414   CACHE_UNIT/dc/dtf/B5.B
    net (fanout=2)        4.973   CACHE_UNIT/dc/tf_dout1[2]
    Topcyg                1.000   CACHE_UNIT/dc/un3_DT1Hit_0.I_90
                                  CACHE_UNIT/dc/un3_DT1Hit_0.I_82
    net (fanout=1)        0.000   CACHE_UNIT/dc/un3_DT1Hit_0.I_82/O
    Tbyp                  0.149   CACHE_UNIT/dc/un3_DT1Hit_0.I_46
                                  CACHE_UNIT/dc/un3_DT1Hit_0.I_73
    net (fanout=1)        0.000   CACHE_UNIT/dc/un3_DT1Hit_0.I_73/O
    Tbyp                  0.149   CACHE_UNIT/dc/un3_DT1Hit_0.I_64
                                  CACHE_UNIT/dc/un3_DT1Hit_0.I_55
    net (fanout=1)        0.000   CACHE_UNIT/dc/un3_DT1Hit_0.I_55/O
    Tbyp                  0.149   CACHE_UNIT/dc/un3_DT1Hit_0.I_10
                                  CACHE_UNIT/dc/un3_DT1Hit_0.I_37
    net (fanout=1)        0.000   CACHE_UNIT/dc/un3_DT1Hit_0.I_37/O
    Tbyp                  0.149   CACHE_UNIT/dc/un3_DT1Hit_0.I_28
                                  CACHE_UNIT/dc/un3_DT1Hit_0.I_19
    net (fanout=3)        4.934   CACHE_UNIT/I_19_n_1
    Tilo                  0.468   CACHE_UNIT/dcc/DC_miss_0_and2_i_0_or6
    net (fanout=17)       0.419   CACHE_UNIT/dcc/N_409
    Tilo                  0.468   CACHE_UNIT/dcc/DC_miss_0_and2_i_0
    net (fanout=44)       4.771   CACHE_UNIT.reg_en
    Tilo                  0.468   CACHE_UNIT/dc/tf_addra[2]
    net (fanout=37)      14.142   CACHE_UNIT/dc/tf_addra[2]
    Tback                 1.041   CACHE_UNIT/dc/dtf/B9.A
    -------------------------  -------------------------
    Total                36.694ns (7.455ns logic, 29.239ns route)
                                  (20.3% logic, 79.7% route)

--------------------------------------------------------------------------------

================================================================================
```