EECS 470 Final Project-A 3-way superscalar Out-of-Order machine with P6 style register renaming

Group2

April 22, 2022

1 Introduction

This is the project report of a 3-way out-of-order RSIC-V CPU using P6 style register renaming, from Group2 in EECS470 course, University of Michigan, Ann Arbor. In final project, we implemented a P6 style register renaming architecture CPU. It works as a 3-way out-of-order superscalar machine and all the other features of it will be described in detail as follows.

2 Design Overview

All the modules we implemented are as listed in the flowchart fig. 1. In fetch stage, we can fetch at most 3 instructions in order. But the fetch number would be less than 3 due to data dependence between the instructions and structure hazard of ROB, RS or CDB.

In the dispatch stage, The decoder would decode the information of the instruction and read the register value from register file. Then it would put them in ROB and RS, record the *rob_tag* in map table. RS would use a MUX to choose T1/T2 and V1/V2 from other place in pipeline or RRF.

In the issue stage, instructions in RS with zero T1/T2 would request to issue. If more than 3 instructions request, we would use 16-to-3 priority selector to choose which one can go to pipeline register.

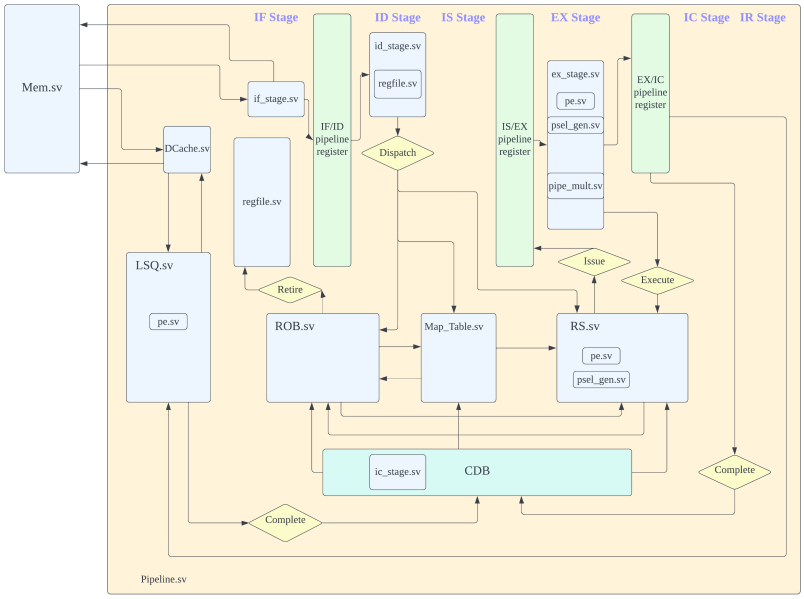


Figure 1: 3-way P6 style CPU Design Flowchart

In the execute stage, we have 3 ALUs with 1 cycle latency and 3 8-stage multiplier. If more than 3 insns complete at one cycle, a priority selector would choose 3 results to EX/IC pipeline register. For those haven't been chosen, we would stall the whole pipeline and wait for them.

In the complete stage, load/store instructions would go to the in-order LSQ and wait for the valid data from d-cache. The head of the LSQ would go to CDB and broadcast the rob_tag and value in all modules.

In the retire stage, the head of the rob would be written into register file. If exception bit detected, all pipeline register and module would be cleared.

3 3-way Superscalar Out-of-Order Design Features

3.1 Pipeline Stages

Our processor is designed to have 6 stages in total: Fetch, Dispatch, Issue, Execute, Complete, Retire.

At fetch stage, the processor fetches at most 3 instruction in order. The actual number of instructions fetched is determined by the dispatched number signal received from RS module. Noticing that there are 2 instructions per memory data, there are two ways we fetch the data. From fig. 2, when the third bit of first PC is 0, we fetch in the way circled by green, otherwise red. The outputs of fetch stage are inst, valid, PC and NPC for current instruction.

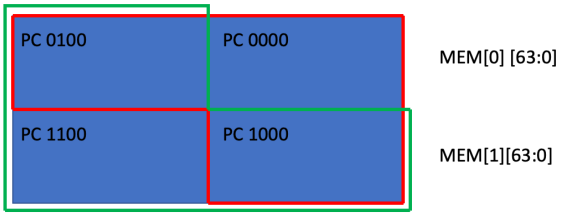


Figure 2: 2 ways of fetching inst

Like the classic P6 architecture, we divide decode stage into dispatch and issue stage to support OoO execution. At dispatch stage, inst, PC and NPC are bypassed as output. At the meantime, we decode src_reg_1 and src_reg_2 as output for use of branch prediction unit. Both RRF and decoder modules function at dispatch stage. RRF has 2 ports for write and read respectively. Each port has at most three wires that naturally support 3-way write&read to register files. At the same time, internal forwarding is supported. Three decoder works in parallel to decode the instruction for each packet in. Dispatch stage sequentially assigns valid bit to every packet until WAW, WAR, RAW dependencies are found.

The decoded packets are sent to ROB, RS, MT and LSQ. RS assigns entries to be dispatched with tag and register value based on MT, CDB and ROB status, position of load and store queue from status of LSQ. At the same time, all values from dispatch packet including PC, NPC, inst, halt, alu_func, cond_branch, uncond_branch, opa_select, opb_select, rd_mem, wr_mem are assigned to each entry. ROB read tags from MT and fetch corresponding register values for RS. At the same time it assign entries for allocated instructions. MT assign ROB tag for instruction allocated in RS and update the ready tag to be 0. LSQ instantiates new entry to assign rob tag for each instruction allocated. The detail description can be found at each subsection. Noticing the dispatch number is co-determined by different modules. We assume that ROB is sufficient for the whole pipeline and there is no structure harzard. Then the dispatch number is the minimal of reservation station available size and the number of dispatch packets valid. Due to time constraint, we do not add the structure hazard of LSQ into the dispatch number for the final submission version. But it should be considered whenever there is load or store in our instructions to be dispatched.

At issue stage, MT uses priority selector to find all instructions those are ready to issue and pick three of them.

At execution stage, MT will clear the busy bit. Three instances of ALU, branch prediction and mult modules calculate ALU results, whether to take branches, multiplication results respectively. There is structure hazard where multiplication execution completes the same time as ALU result.

Priority selector is used to determine which goes to CDB first.

At complete stage, MT update ready bits of entries with same destination rob tags as complete instructions to be 1. ROB updates complete bit of corresponding entry to be 1. Note that ALU module and LSQ might have instruction complete at the same time. We resolve this hazard by always assigning higher priority to ALU unit and LSQ is only able to complete whenever there is spare slot in CDB.

At retire stage, The rob moves head forward by at most 3 slots. When the instruction has a destination register, it writes value to corresponding register files. Or if the instruction is a store, ROB will send a signal to LSQ to enable it to retire the head and write the data into DCache. At the same time, MT will clean the ready bit of entries with register index the same as retired rob register index and rob tag the same as retired rob number. RS clean the rob tags that are retired and assign the register values into the value fields. The number of instructions retired at each cycle can only be the minimal of number of consecutive completed rob entries and the number of rob entries until first exception bit met. Whenever a exception is found, the ROB will send a clear_all bit to all the rest of processor to clear all the existing state and send the fetch stage next PC to be fetched.

There is one special hazard that is due to implementation. When the same rob tag is used in dispatch, complete stage, and retire stage, the rob tag and ready bit assigned to MT must be assigned based on different situations. The dispatch has the highest priority, while completion second, and retire third. At each cycle, if the destination rob tag is the same as any of three of CDB rob tags, or the retired rob number in ROB, the ready bit in MT is always 0. Likewise, when the register index of CDB that has completed is the same as any of the three register indices in the retired rob entry, the rob tag is always set to CDB rob tag and the ready bit is 1.

3.2 Re-Order Buffer

Our re-order buffer (ROB) consists of 64 entries. Each entry contains target register index, register value, complete bit to indicate whether the entry is completed, exception bit to indicate whether the entry is an exception, and *alu_result* to record the branch target. The ROB uses a head and tail pointer to mark the first and last non-empty entry. Both head and tail pointer can be incremented by three at most to support three-way superscalar. For convenience, we would not use $ROB\#0$ and *rob_tag* is zeros indicates invalid.

When a new cycle starts, the head pointer will move based on the number of instructions that are going to retire in this cycle. The *retire_num* is calculated by checking the complete bit and exception bit of the three entries starting from the head pointer. The tail pointer moves based on the number of instructions that are going to be dispatched. The *dispatch_num* is calculated by checking the remaining empty entries of both ROB and RS.

When an instruction is dispatched, we record its destination register in *reg_index*. After an instruction is completed, ROB use its attached ROB entry index to find its corresponding entry and set the complete bit high and record the register value in *reg_val*. Completed instructions can go to retire stage in the next cycle. When an instruction can retire without exception bit being high, we clear the ROB entry and send the register value to register file. When an instruction is going to retire with the exception bit high, all ROB entries are cleared, and head and tail pointers are set to zero.

3.3 Reservation Station

The reservation station(RS) consists of 16 entries. each entry is not matched with certain function units names. In the dispatch stage, the instruction information decoded from decoder can be written into any available RS entry, the RS entry would record all about the instruction, including whether it's a branch, multiplication, load, store or wfi. For the choice of the source register value, if the required value haven't been calculated in the pipeline, RS would record a *rob_tag* in T1/T2 and wait for instruction with *rob_tag* to be finished. We have a MUX to forward the register value with *rob_tag* if it's somewhere in the pipeline. the possible position could be in ROB, CDB or RRF. The *ready_bit* in Map Table indicates the value in ROB or not. RS would monitor the CDB to get the source value which just completes.

In the issue stage, one instruction can issue when T1/T2 are both zero and all the sources values are ready. If more than 3 instructions requests to issue, we would use a priority selector(16 to 3) to select up to 3 instructions to issue. In the execute stage, corresponding RS entry is cleared. In the complete stage, the CDB *rob_tag* would search in T1/T2 find matches. If matches found, the *rob_tag* would be cleared and the values broadcast on CDB can be written into corresponding RS entry.

When an exception happens, RS will be informed by ROB, then RS will clear all entries.

3.4 Map Table

Map Table has 32 entries. Each entry would record the *rob_tag* and *check_ready* bit. In the dispatch stage, The *reg_index* and corresponding *rob_tag* of dispatched instructions are written into map_table. The information recorded in map table will be sent to ROB and RS continuously. In the complete stage, if the *rob_tag* in CDB matches the *rob_tag* in Map Table, it would set the *check_ready* mark "+". This "+" bit helps us know whether the required value is ready in ROB. In the retire stage, both the *rob_tag* and the "+" ready bit corresponding to the retired register would be cleared. If all of the three stages happened in one cycle, we assume that the retire stage would perform firstly, then the complete stage and finally the dispatch stage. So we add conflict detect logic and tend to retain the dispatch result then the complete and finally the retire result.

3.5 Multiple Function Unit with Varing Latencies

There are six main function units. There are three units for ALU functions, and three units for multiplication functions. The ALU units calculate arithmetic instructions that are not multiplication functions or load/store address. For conditional branch instructions, it would also compare the two values to know whether the branch would actually be taken. The ALU functional units complete execution in one cycle. The multiplication function units complete execution in 8 cycles. It contains 8 mult-stages and all stages are pipelined. So it can accept a new input every cycle if there is no structure hazard in the CDB. When a unit finishes execution, it attempts to broadcast its value on the CDB. We add an 6-3 priority selector here to accept multiplication functions units result firstly which ensures that the multiplication functions would never be stalled. If the ALU units are stalled due to CDB full, the whole pipeline would stall until the CDB is available. Also, if exception happens, all functional units will clear their registers.

3.6 Load-Store Queue (LSQ)

To support the operation of the two types of instructions that interface with memory — LOAD and STORE, we implemented the LSQ structure. The LSQ has 16 entries. Every entry can be used for both the load and store. This LSQ makes LOAD and STORE run in order to solve hazard. In one cycle, up to 3 load and store can dispatch, issue and execute, but only one load or store can be sent to CDB or interact with dcache.

When a LOAD or a STORE dispatches, the LSQ will record the information of the instruction and the tail pointer increments. The head and tail pointers are used just like how they are used in ROB and RS. When one entry gets data and address calculated from ALU, it can be sent to dcache. For load, only when it is at the head of both LSQ, LOAD command can be sent to dcache. But for store, it also needs to be the head of ROB.When the LSQ receives the signal that indicates dcache has finished the load or store, the corresponding entry can be put on CDB. Since other function units have higher priorities than LSQ, LSQ can do complete only when the CDB has empty slots. After complete, the entry retires and head pointer moves forward.

3.7 DCache

We implement one-way directed-map data cache without advanced features due to time constraint. The inputs for it are response, tag and data from memory; command, data, memory size and memory address from processor. The outputs are command, memory address and memory data that DCache sends to memory; the data DCache giving back to processor and one bit indicates whether the data is valid. The major data structures in the DCache are data entries with a table for current state.

We use finite state machine to encode states of DCache for each cache line and it mainly consists of four states:IN_CACHE, DIRTY, COMMIT, IN_MEM. The stage transfer graph can be shown in fig. 3 We have in total 32 cache lines as specified. At the very beginning when the processor wants to read or write data from DCache, we spare one cycle before the DCache starts functioning. This is because we want to reset some signals after a new instruction comes in that is used by prior instruction like $write\_commit, load\_commit, send\_command$.

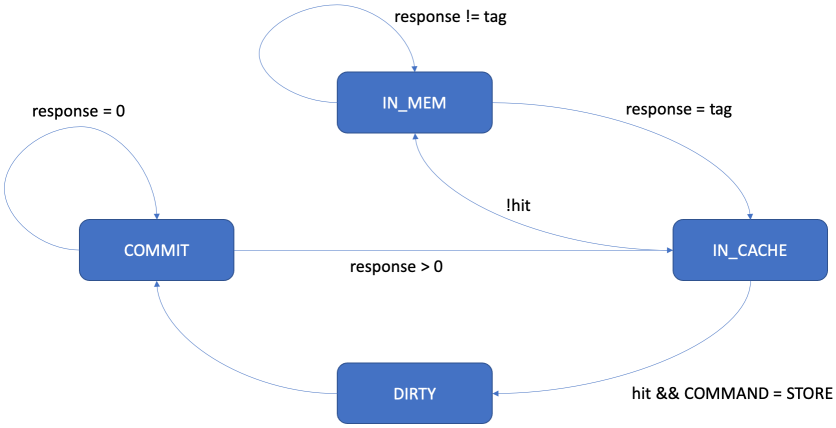


Figure 3: State Diagram of DCache

3.7.1 IN_CACHE

For each cycle, current index and current tag are decoded from the memory address from processor. IN_CACHE stage represents the data with current index(not necessary current tag) is in DCache and it is consistent with the data in memory. After the first cycle, we evaluate for current instruction, no matter it's a STORE or LOAD, whether we have a cache hit or miss. The hit only goes to 1 if the current tag from processor matches the tag stored in DCache, the valid bit for current index is 1 and the state for current index is IN_CACHE. if hit happens and the command from processor is a LOAD, we directly send the data back to processor and set the valid out bit to 1 while we set command for memory to be NONE. If hit happens while the command is STORE, we transfer the state for current index to be DIRTY while setting valid out bit to be 0. Otherwise, if the command is not NONE, the DCache state will transfer to IN_MEM which indicates we need to acquire new data from memory.

3.7.2 IN_MEM

IN_MEM indicates the data the processor want to access is still in memory but not in DCache. This is equivalent to a cache miss. No matter the command from processor is a LOAD or STORE, we need to firstly transfer the data from memory to cache. DCache will firstly send command LOAD to memory and wait for response. When the response is valid, DCache updates its memory tag to be the response for later use. DCache at this point will wait for the tag from memory and the valid out bit to processor is always 0. When DCache receives the tag from memory, it means the DCache can access the data sent from memory. Correspondingly, DCache will update data in that cache line. After that, if the command from processor is LOAD, the state will be transferred to IN_CACHE and if STORE, to DIRTY. DCache spends majority cycles on this state from getting response to fetching data from memory and it depends on memory speed.

3.7.3 DIRTY

DIRTY state means the command from processor is a STORE and the address the processor want to write to is in cache and the data in cache is updated while the update hasn't been pushed into memory, where there is a inconsistency. At this point DCache will write the data in cache line in BYTE, HALF or WORD by the input mem_size. After this is done, the state will be transferred into COMMIT. The DIRTY state takes 1 cycle and the valid out bit is 0.

3.7.4 COMMIT

COMMIT is the state that DCache push the updated data into memory. This state takes cycles until the LOAD command sending to memory is accepted. Considering we do not have arbiter to manage data flow to ICache and DCache, these is no congestion so the cycle spent on this is state is 1. The state will automatically transfer to IN_CACHE after one cycle.

4 Advanced Features

The following are the advanced features of our processor.

4.1 3-Way Superscalar

Our pipeline is three-way superscalar, which means it fetches three instructions, possibly dispatches, issues, executes, completes and retires three instructions at most. Theoretically, this would decrease the CPI significantly. However, CPI doesn't behave as expected due to memory latency, blocking cache and data dependencies. We expect to load 3 instructions(2 address) per cycle from I-cache so we need a 2-way I-cache. And it would cost many cycles for the I-cache to load from memory. If there are too many load or store, the blocking d-cache would be the bottleneck. In the dispatch stage, each instruction would compare the source and destination registers with the older instructions. It would not be dispatched if any of the WAR, WAW and RAW data hazards is detected in this cycle. So it can't dispatch 3 instructions according to data dependence.

4.2 GUI debugger

We developed our 3-way superscalar GUI debugger based on VTUBER in project3 as shown in Figure4. In the pipeline window, there are 6 stages, including the fetch(IF), dispatch(ID), issue(IS), execute(EX), complete(IC) and retire(IR), and the debugger would show the three instructions in each stage. The "V" or "I" in the following three lines indicates whether the instructions are valid or not. we extend the pipeline register window to 3 ways and add ROB, RS and Map Table window in the bottom. In the stage and module window, the debugger would show us the important registers' value and how they change each time. For example, we would can learn about which instructions are fetched into the pipeline according to the NPC in the IF stage. And if some instructions are not issued, we would refer to the T1/T2 and V1/V2 values in RS window. After the branch has taken and exception happens, we would check if all modules and pipeline have been cleared.

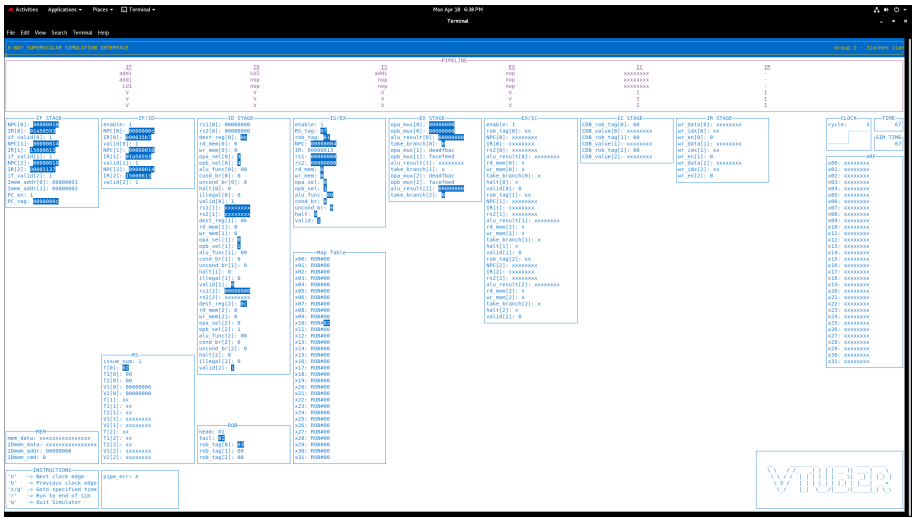


Figure 4: GUI debugger

5 Test and Results

5.1 Test Strategy

To test the whole processor, we test every module separately first to make sure its function is correct. When integrating the modules, we also do it from simpler version. We tested the processor without LSQ, Dcache and Memory, to make sure the processor can run correctly when there is no LOAD or STORE. After that, we grouped all modules and simulated with more complex programs.

We mainly use several methods to debug. Basically, we add display statements in testbench. Such information will be shown in pipeline.out. We have changed our pipeline.out to 6 stages and in each stage it can display 3 instructions one cycle. Pipeline.out displays the NPC and intruction of each stage and the writeback-reg and value in every cycle. It also shows the Memory and bus command to help us to debug the LOAD and STORE. Pipeline.out helps us to know the process of every instruction clearly — which instruction make the pipeline break off and the possible reasons. After that, We check the waveform of VCS, based on the information provided by pipeline.out. We can pinpoint the signal that has something wrong by analysing the waveforms step by step. If the pipeline can run smoothly, we compare the program.out and writeback.out with the correct files to do futher debug. We also designed a shell script to run all testcases and do comparison automatically. If we want to check all important register value in certain cycle more quickly, we would use our visual debugger to show them all.

5.2 Synthesis Strategy

5.2.1 Synthesis problems

The same as testing, we synthesised small modules first and finally the whole processor. Here are some problem we faced and how we dealt with them.

To solve some unsynthesizable problems caused by $clog2, we used the priority encoder module to encode index.

```
// for(int i = 0; i < 3; i ++) begin
//     if(issue_gnt_bus[(i + 1) * `NUM_RS-1 -: `NUM_RS] != 0) begin
//         issue_idx[i] = $clog2(issue_gnt_bus[(i + 1) * `NUM_RS-1 -: `NUM_RS]);
//     end
// end
```

Figure 5: $*clog*2 function synth bug

```
read_file -f ddc [list "psel_gen/psel_gen_REQS3_WIDTH16.ddc"]
set_dont_touch psel_gen_REQS3_WIDTH16
read_file -f ddc [list "pe/pe_OUTWIDTH4_INWIDTH16.ddc"]
set_dont_touch pe_OUTWIDTH4_INWIDTH16
```

Figure 6: solution

Since we can not use uncertain values in for-loops, we use valid signals instead, which will be more straightforward in the picture.

```
    for (int i = 0; i < retire_num; i++)begin
        rob_packet_out[i].retire_R_out = rob_entry[(head + i - 1) % SIZE + 1].reg_idx;
        rob_packet_out[i].retire_V_out = rob_entry[(head + i - 1 ) % SIZE + 1].reg_val;
        rob_packet_out[i].retire_valid = rob_entry[(head + i - 1 ) % SIZE + 1].valid;
        rob_packet_out[i].retire_rob_tag_out = (head + i - 1) % SIZE + 1;
        rob_packet_out[i].NPC = rob_entry[(head + i - 1 ) % SIZE + 1].NPC;
        rob_packet_out[i].inst = rob_entry[(head + i - 1 ) % SIZE + 1].inst;
        rob_packet_out[i].halt = rob_entry[(head + i - 1 ) % SIZE + 1].halt;
    end
end
```

Figure 7: uncertain for-loop bug

```
    for (int i = 0; i < 3; i++)begin
        if(retire_num_valid[i])begin
            rob_packet_out[i].retire_R_out = rob_entry[(head + i - 1) % SIZE + 1].reg_idx;
            rob_packet_out[i].retire_V_out = rob_entry[(head + i - 1 ) % SIZE + 1].reg_val;
            rob_packet_out[i].retire_valid = rob_entry[(head + i - 1 ) % SIZE + 1].valid;
            rob_packet_out[i].retire_rob_tag_out = (head + i - 1) % SIZE + 1;
            rob_packet_out[i].NPC = rob_entry[(head + i - 1 ) % SIZE + 1].NPC;
            rob_packet_out[i].inst = rob_entry[(head + i - 1 ) % SIZE + 1].inst;
            rob_packet_out[i].halt = rob_entry[(head + i - 1 ) % SIZE + 1].halt;
        end
    end
end
```

Figure 8: solution

The third one is, we need to give all the registers used in combination and flip-flop blocks initial values. Similarly, we need to take all the if states into consider, in case of getting X values in synthesizing test.

Another interesting thing we found is that, when running synthesis of small module, we need to make the parameters of it the same with the ones in upper level modules. And then, the post synthesis files' name would be changed into a name with parameter numbers as illustrated in the picture.

5.2.2 Synthesis Tool

We used Synopsys vcs Xpropagaion simulator as a tool to help debug synthesis. X-Propagation can affect the behavior of control logic, make simulation work like post synthesis. For example,

```
read_file -f ddc [list "psel_gen/psel_gen_REQS3_WIDTH16.ddc"]
set_dont_touch psel_gen_REQS3_WIDTH16
read_file -f ddc [list "pe/pe_OUTWIDTH4_INWIDTH16.ddc"]
set_dont_touch pe_OUTWIDTH4_INWIDTH16
```

Figure 9: vg file name

```
always@*
    if(s)
r = a;
    else
        r = b ;
```

Figure 10: if statement with X input

S	A	b	V-merge	T-merge	X-merge
X	0	0	0	0	X
X	0	1	1	X	X
X	1	0	0	X	X
X	1	1	1	1	X

Figure 11: Xpropagation outputs

when the condition of the if statement is unknown, the output is always an X. So we must avoid X signal and give initial values to all our registers.

Case statement is similar to if statement. The truth table shows when the value of the case expression is unknown, The output is always an X with xprop tag added.

```
case (s)
    1'b0: r = a;
    1'b1: r = b;
endcase
```

Figure 12: case statement with X input

s	A	b	V-merge	T-merge	X-merge
X	0	0	r(t-1)	0	X
X	0	1	r(t-1)	X	X
X	1	0	r(t-1)	X	X
X	1	1	r(t-1)	1	X

Figure 13: Xpropagation outputs

Edge sensitivity expressions must be handled carefully, for example asynchronous flip-flops. In Verilog, a posedge expression will occur for the following transitions, Verilog will optimistically consider all of these transitions as if a rising edge of the signal occurred, which is not necessarily true. For example, let's consider the 0-to-X transition. X can represent either a 0 or a 1, which means a rising transition may have happened, or may not have happened. Both cases need to be considered.

In Verilog, latches are implemented with an if statement that does not have an else branch, as shown below. Because of the missing branch, whenever the clock into the latch is an X, this causes a merge of the current value of the latch with the data input. So we get the truth table for this situation like this.

5.3 Simulation Results

In both pre-synthesis simulation and post-synthesis simulation, we pass all the .s files correctly. The minimal clock period is 12.8 ns.

```
always@(posedge clk, negedge rst)
   if (! rst)
      q <= 1'b0;
   else
      q <= d ;
```

Figure 14: Edge Sensitive Process-D-flip flop

Clk	V-merge	T-merge	X-merge
0 -> 1	d	d	d
0 -> X	d	merge(d,q(t-1))	X
0 -> Z	d	merge(d,q(t-1))	X
X -> 1	d	merge(d,q(t-1))	X
Z -> 1	d	merge(d,q(t-1))	X

Figure 15: Xpropagation outputs

```
always@(*)
   if(g)
      q <= d ;
```

Figure 16: Latch code

G	d	V-merge	T-merge	X-merge
X	0	q(t-1)	merge(0,q(t-1))	X
X	1	q(t-1)	merge(1,q(t-1))	X

Figure 17: Xpropagation outputs

6 Performance and Analysis

6.1 Compared with one-way and two-way processor

To analyse the performance of the 3-way OoO processor, we set the process can only fetch, issue and retire one cycle to get a 1-way OoO processor. The CPIs of two processors are shown as follows fig. 18.

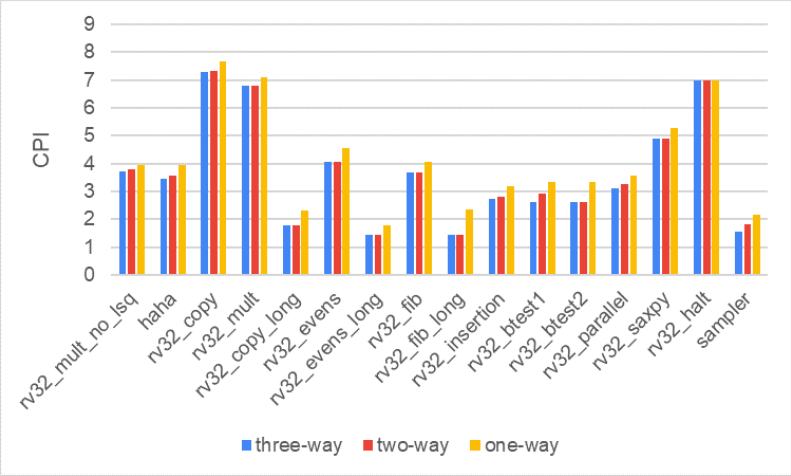


Figure 18: CPI comparison with one-way,two-way and three-way superscalar processor

We expect that 3-way superscalar can drastically decrease our CPI. In some cases with weak dependence of data and few load or stores, we can observe a decrease of CPI compare to one way or two way. But for those test cases with strong data dependence and lots of load or stores, the CPI is not changing too much because it's mainly limited by data hazard and bad d-cache performance. Our bottle neck in superscalar is that our d-cache is one way and blocking. The pipeline have to wait many cycles before load a value from memory or store value into memory. So the pipeline retire no instructions after load or store for many cycles, which damage our CPI significantly. If we have time to develop a 3 way non-blocking d-cache, the advantage of the 3-way superscalar would be more obvious.

6.2 Multiplier Analysis

To analysis the performance of the multiplier stage number, we change the multiplier from 6 stages to 8 stages and compare the execute time of all programs as shown in fig. 19. Switching from 6

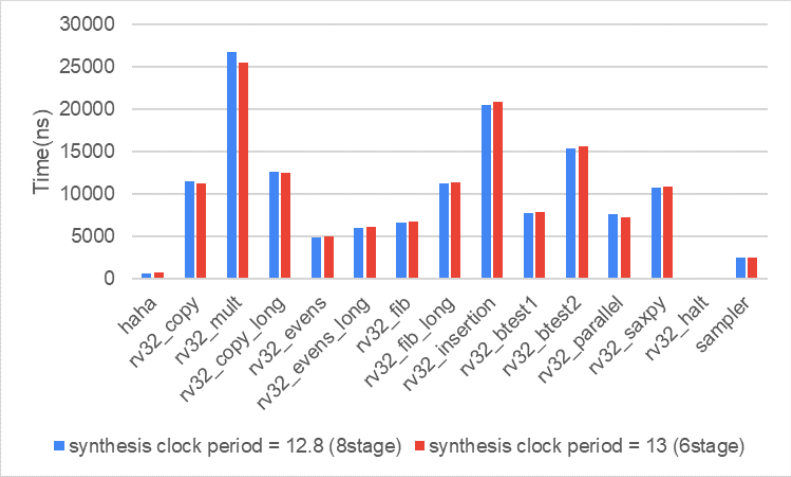


Figure 19: Execute time comparison between 6-stage multiplier and 8-stage multiplier

stage multiplier to an 8 stage multiplier improves our clock period from 13.0 ns to 12.8 ns. The shorter clock period would decrease the total execution time for the program which contains less mult instruction. However, the 8 stage multiplier would take more time for the mult instruction to be completed. Specifically, copy_long.s, copy.s, mult.s and parallel.s took longer to execute with 8 stage multiplier. Because all of these programs have a large number of multiplies. And some instructions are dependent on the multiplications. So the execution time would actually be longer than 6 stage multiplier though under a better clock period.

6.3 Cache Line Size Analysis

To analyse cache, we change the cache line size t We compare the impact of different setting of

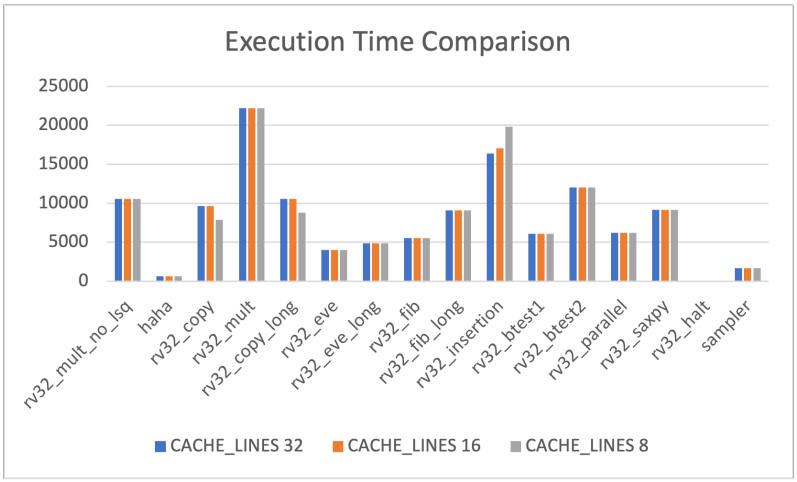
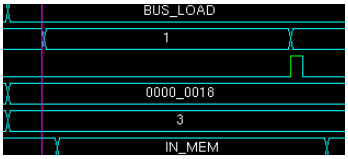


Figure 20: CPI comparison between different cache line size.

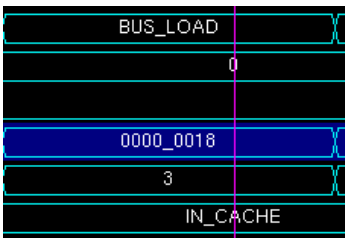
cache line size to the performance of our processor. When no load or store operation is involved, like mult_no_lsq, the execution time is exact the same.

When load or store are involved, we expect the CPI of processor with bigger cache line should be higher than the one with smaller.

For simplicity, we will use $8 to represents processor with cache line 8 setting and $32 with cache line 32. Taking insertion.s as example, we can see that the execution time of $8 increases from 16360 ns to 19840 ns by 21.27%. This is caused by the conflict of cache line from our observation. From section 6.3, at clock 401, the processor tries to load data from address 0x0018. While the



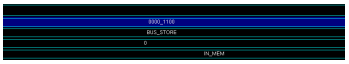
(a) wave for cache line 8, insertion.s



(b) wave for cache line 32, insertion.s

data of is in cache for $32, the state of data is in memory for $8. The cache has been occupied by other data with same cache address and is in conflict with current tag for cache[3]. Thus the processor requires more time to load the data back from memory again which consumes more cycles. This happens very often for this program and eventually causes a big delay of execution.

(a) wave for cache line 8, copy.s



(b) wave for cache line 32, copy.s

However, this is not always the case. From fig. 20, the execution time for copy.s of $8 is slightly better than $32. That is because at cycle 41, when the processor try to store a value into memory address 0x0018, while the data of $32 is in memory, that of $8 is in cache. So $8 only needs to transfer to DIRTY, COMMIT, and back to IN_CACHE state to finish the store but $32 needs to load data first, which consumes more cycles.

In conclusion, although it happens that cache with smaller cache line size could have better execution time than that with larger cache line size, from a perspective of probability, the execution time of cache with larger cache size should be shorter as there is less chance for conflict.

7 Fixed Bugs and Critical Path

7.1 Bugs and Fixes

Bugs and Problems	Method to Fix
Speculating on load dependencies and forwarding optimally using LSQ	Change to an in-order LSQ
Developing the 3-way non-blocking I-cache	Connect directly to the memory.
d-cache can't be developed based on the given I-cache	Rewrite the d-cache using FSM strategy.
Map table ready bit set wrong	Add conflict detect logic and set priority from dispatch, complete to retire
Multiple branch together causing target PC pass error	Record each target PC value in ROB after branch complete
Ex_stage alu-module and brcond-module result error	Add $signed() function to all signed register
Jal and Jalr instruction register value wrong	Put PC + 4 to CDB instead of alu_result
Load can't distinguish signed and unsigned value	Differentiate the signed and unsigned value according to Mem_Size[2] bit
$clog2() can't synthesis	Using priority encoder to calculate the log value.

7.2 Critical path and Synthesis Clock

We get the critical path of the pipeline from the .rep file after synthesis. We find it start from the rob/head_reg and end in map_table/mt_entry_reg[rob_tag] which takes 12.53 ns. It's

$rob\_head \rightarrow rob\_packet\_out \rightarrow rob\_packet\_in \rightarrow map\_table/rob\_packet\_in[complete\_reg\_idx] \rightarrow map\_table/mt\_entry\_reg[rob\_tag]$. It means that when the CDB broadcasts the rob_tag to Map Table, we first look up the $ROB\#$ in ROB and then set the ready bit in Map table according to the ROB output. We can remove this critical path by searching directly in the Map Table using CAM.

Our synthesis clock is chosen as 12.8 ns to make sure that all slack can meet the requirement.

8 Future Work and Involvement

8.1 Future Work

- Fix unknown bugs when running .c programs. We need more time to debug what's happening to our .c programs.
- A 3-way I-cache and arbiter. Currently, we put the programs directly in the memory assuming zero latency. We need an efficient non-blocking I-cache which supports 3-way superscalar.
- Write-back non-blocking d-cache. Our CPI is limited by the one-way d-cache. If we want to show significant advantage of 3-way superscalar to one-way, non-blocking d-cache is required.
- Branch predication. We want to use the branch predication module to improve the performance of btest1.s and btest2.s.

8.2 Involvement

- Lingfei Huo. Pipeline Stages(35%) ROB(50%) RS(50%) MT(50%) dCache(100%) Test(20%) Debug(50%) Report(20%)
- Jiachen Jiang. Pipeline Stages(35%) ROB(50%) RS(50%) MT(50%) Multiple FU(100%) GUI_debugger(100%) Test(20%) Debug(50%) Report(20%)
- Yuanzhi Xiong. Pipeline Stages(15%) LSQ(100%) Synthesis(33%) Test(20%) Report(20%)
- Chenhao Ma. Pipeline Stages(15%) Synthesis(33%) Test(20%) Report(20%)
- Peiqing Zhang. Synthesis(33%) Test(20%) Report(20%)

9 Conclusion

We have implemented a 3-way superscalar out-of-order core based on P6 architecture. Our final achieved synthesis clock period is 12.8 ns. The overall average CPI of all assembly test cases is 3.628. We passed all the assembly test cases but fails the .c programs. Passing all the assembly test cases shows that most of our design choice works but failing on other programs means that there are still some hidden bugs we haven't found when running long programs. We would continue working on it if we still get access to the VCS license.

A Symbol

- inst: instruction
- PC: program counter
- NPC: next program counter
- src_reg#: source register number
- RRF: register file
- ROB: reorder buffer
- MT: map table
- RS: reservation station
- LSQ: load store queue