Two-way Superscalar Processor With Local Branch Predictor And Direct mapped Cache

Wentao Zhang Ruiqi Wang Yuanqing Lou
Xingyu Wang Qipeng Wang
Electrical Engineering and Computer Science, College of Engineering
University of Michigan, Ann Arbor
{zwtao, rickywrq, qingqing, xingyuw, felixwqp}@umich.edu

***Abstract*—This paper presents the implementation of a two-way superscalar processor with P6 semantic for the course Computer Architecture (EECS470) at the University of Michigan. This report introduces a design of a two-way superscalar out-of-order processor with local branch predictor and associative cache. The measures of effectiveness are also analyzed after the design section. The analysis discussed some advanced features, such as prefetcher, instruction cache, local branch predictor and load-store queue, and their positive influence in the overall performance in the processor.**

***Index Terms*—direct mapped cache, local branch predictor, two-way superscalar processor**

I. BACKGROUND

In the final project at the University of Michigan's Computer Architecture (EECS 470) course, students in a group of five need to design an out of order processor for the RISC-V instruction set.

The processor we design performs worse than the processors on the market due to the limitation of time and knowledge. But this project serves to let the students be familiar with the basic component and design flow of a modern processor.

In the following section, we will show how the two-way superscalar processor with local branch predictor and associative cache is designed. The scope of the audience of our project and how our processor will be tested will also be introduced after the project design section. The proposed visual which helps the audience to understand our processor design will be placed throughout the handout.

II. PROJECT DESCRIPTION

A. MOTIVATION

This project is aimed to provide a working processor to run the assembly code efficiently. Although the expected performance of the designed processor is much worse than those processors on the market, this project will lay a solid foundation of the basic processor components' understanding and have a hand on the basic processor schematic and design rules.

B. INST FETCHING SYSTEM

Our inst stage (Fig.2) consists of three parts:

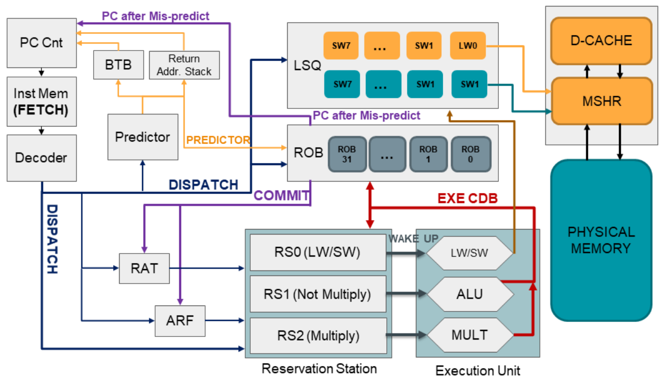


Fig. 1. Overview of the processor we designed. The processor is a two-way superscalar out of order processor with local branch predictor and four-way set-associative cache. The schematic we use is P6 with a general-purpose LSQ. The cache supports two-way read and write with data pre-fetching.

1) *Instruction cache:* The instruction cache stores the fetched instructions. It has no write port. It is directly mapped cache with cache line of size 4B. The total size is 256B. It supports one way read only. The miss signal will trigger the prefetcher.

2) *Prefetcher:* The prefecther will prefetch the instructions from the memory once it is triggered. In our design, it will prefetch 8 instructions. It won't be triggered when a miss happens but that instruction is actually in the middle way of prefetching process.

3) *Instruction cache controller:* Controls the interface between the prefetcher and memory. It will send and receieve the memory response from the memory arbitrator and tell prefecther whether the instruction prefeching process succeeded or not.

C. PC SYSTEM

Our PC stage consists of 4 parts:

1) *PC Register:* The PC register will be responsible for fetch instructions from the instruction cache (Fig.3). In our design, it only stores the PC for the first instruction. The second instructions will always be fetched at PC+4. The stored PC comes from the results of BTB, RAS, predictor, and decoder.

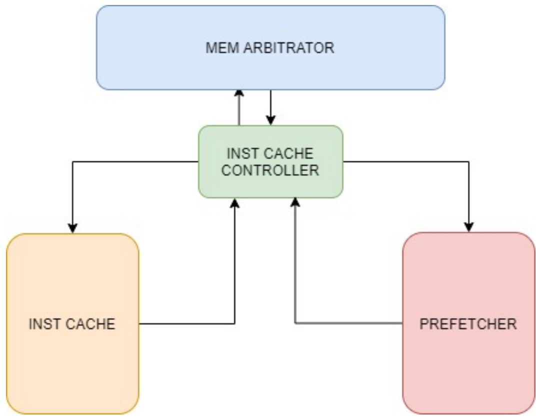


Fig. 2. Overall view of the inst stage part. The inst cache controller will determine whether enable the prefetcher. The prefetcher will send the address to the controller to load the needed instruction.

2) Local Branch Predictor: We use 2-bit local branch predictor. The size of it is 16. It will be updated when RoB committed the branch instructions.

3) Return Address Stack (RAS): The RAS depth is 4. It will push or pop the address based on whether the destination register is linked or not (Based on the Risc-V definition.)

4) Branch Target Buffer (BTB): The size of BTB is the same as the local branch predictor, which is 16. BTB will not record the JALR branch address, but it will record the JAL branch address.

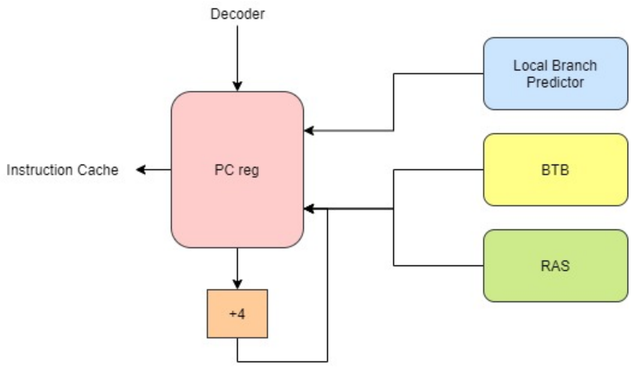


Fig. 3. Overall view of the PC register part. The decoded information from the decoder will tell PC reg to read the value from Local Branch Predictor and BTB or RAS or just current PC value plus four. In the same cycle, the instruction cache will read the current value in the PC reg and send the instruction at that address to the decoder.

D. COMPUTING SYSTEM DESIGN

The purpose of the computing system is to execute the instruction based on the value in the register file. The computing system is designed as a two-way superscalar with P6 schematic. The size of the Rename Table (RAT) and Architecture Register File (ARF) is 32, which meets the requirement of the RISC-V data set. We divide the instructions into four categories, Adder related, Multiplier related, Branch related and Memory related. Separate RS is used for the above four kinds of instructions. The central data bus (CDB) is designed to be two-way to meet the limitation post by the project requirements. To have a high prediction rate, we will implement a 2-bit local branch predictor, the Branch Target Buffer (BTB) and Return Address Stack (RAS).

1) Decoder: Decode the instruction (Fig.4). The decoder we are using is basically the same as that in a previous project. Since we are building a two-way superscalar processor, we have two decoders running in parallel. Each decoder gives information of an instruction such as PC, NPC(next PC), registers' indexes and values and types of executions. The results of the decoder is packed into a so-called decode_packet, which will be send to Rename Table(RAT), Reservation Station(RS) and Reorder Buffer(RoB) for further use.

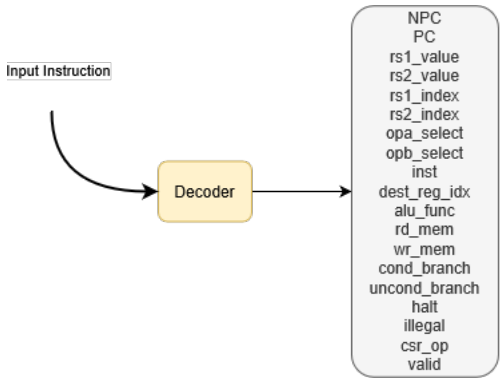


Fig. 4. Overall view of decoder. Decoder takes an instruction as input and outputs the decode packet.

2) Architecture Register File (ARF) : This is the place to store the temporary data, which will be used during the execution process. Some instructions write values to registers, so the values of registers in ARF will be updated whenever such instruction commits. The size of it is determined by the instruction set. Here, the size is 32.

3) Rename Table (RAT): The purpose of the rename table is to solve the false data dependency among the input instructions, by mapping the index of the destination register of an instruction to the index of the Reorder buffer (RoB) entry where this instruction is assigned to. The renamed registers will be sent to Reservation Station(RS). Whenever RoB commits an instruction, registers with tag "RoB" and value equal to the RoB entry number of the commited instruction will be set to have tag "ARF" and value equal to the output value of the committed instruction.

4) Reservation Station (RS): This is the place for the instructions to wait for the data it needs and then send it to the execution unit to be executed (Fig.6). The purpose of it is to achieve the "out of order" execution sequence for the input instructions. In this way, we can speed up the processing speed. An RS entry stores the source operands of an instruction, the corresponding RoB entry number, the type of operation and whether this instruction is branch or not. In our design, there are three types of RS corresponds to three different types of instructions. The reason that we do this separation is that

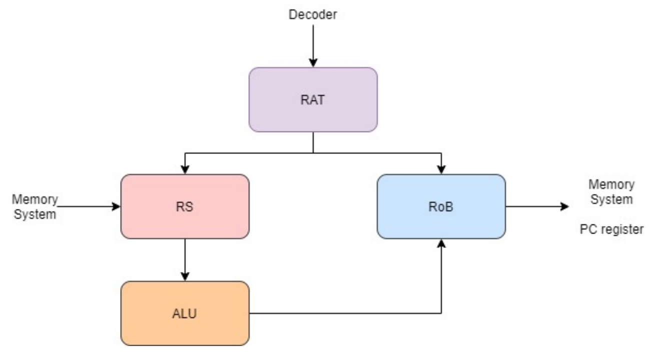


Fig. 5. Overview of computing system part of the processor. The RAT will solve the data dependency by renaming and sent the resolved instruction to both RS and RoB. The RS will reserve the instruction until all the data it needs are ready. After that, it will send it to the ALU and delete that instruction from RS. The execution result will send to RoB and RS. The RoB will send the executed result to the PC reg and memory system when that instruction is ready to be committed.

different instructions takes different number of clock cycles to run. The first type is RS-ALU. This RS stores instructions with operation that can be finished within one clock cycle. This type of operation includes add, bit shift, compare, etc. The second type is RS-MULT. This RS stores instructions with multiplication. Multiplication is more complicated than add when executing on hardware. If multiplication is designed to be executed in one cycle, our clock period will be very large because the critical path of multiplication is much longer than that of other operations. To reduce the clock period, we choose to split multiplication into several parts. Each part will be done in one cycle. The last type is RS-LWSW. This RS stores load/store instructions which will access memory. Since we have Load Store Queue(LSQ) specifically for storing load/store instructions, it's better to build an RS specially for these instructions.

RS will receive input from EXE-CDB, MEM-CDB and ROB-OUT every cycle. RS entry will update tags and values of operands of the stored instruction. The process is, first, check the tag of the operand. If the tag is "RoB", then check the RoB entry number of the input packet and the value of the operand. If they are equal, update the operand by changing its tag to "ARF" and value to the value of the input packet. Instructions will stay in RS until all of its source operands have tag "ARF". Once ready, instructions in RS-ALU and RS-MULT will be send to ALU-execution unit and MULT-execution unit, respectively. Instructions in RS-LWSW will be broadcasted directly from RS, which will be received by LSQ.

5) EXECUTION UNIT: Two types of execution units are used in our design (Fig. 7). One is the single-stage bit-wise operation ALU and the other one is the multi-stage pipe lined multiplier. There are 7 ALU and 7 MULTs in the execution unit. The stage number multiplier used is 4. The stage number is parameterized. Thought our processor is two-way superscalar, our execution units don't necessarily do two operations per clock cycle. Instead, our design has made an

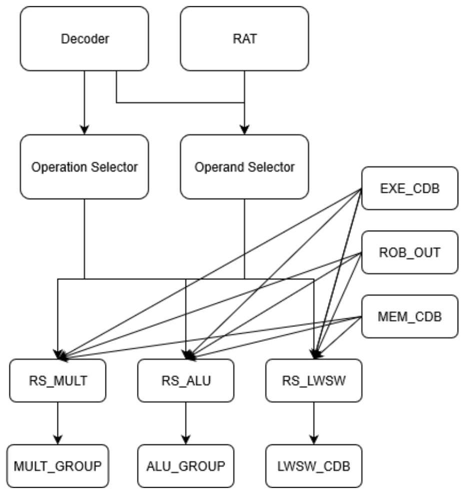


Fig. 6. Overall view of Reservation Station(RS). Whenever a valid decode packet is received, RS will determine the operation and operands of the input instruction and store it in the corresponding RS. Every clock cycle, each RS entry will receive the packet from EXE-CDB, MEM-CDB and ROB. RS will update the operands based on the input information.

improvement such that every instruction in RS that is ready at this clock cycle can be sent to the execution unit. This will make our processor more parallel and will greatly improve the performance.

6) Reorder buffer(RoB): The purpose of RoB is to ensure the processed data is written in the original order into the memory and the register. There are 32 entries in the RoB. The head and tail register provides the information of the which two instructions to be committed and where to store the next two instructions. The two-way EXECDB provides the executed results from the function units. The RoB will record the branch predicted result and target. If misprediction happens, the exception signal of that rob entry will raise and the exception will be solved at commit by raising the clean signal.

7) Central Data Bus (EXECDB, MEMCDB, LW_SW_CDB): This serves for transferring data among all the above parts. The size of it is two-way to meet the requirement of the project. EXECDB broadcasts the results from execution unit, MEMCDB broadcasts the results of load instructions. Both two CDBs will be the input of RS to update tags and values of source operands of an instruction. LW_SW_CDB broadcasts addresses and values of load/store instructions.

E. MEMORY SYSTEM DESIGN

The purpose of the memory system is to read and write the data into memory. We have caches and Load Store Queue (LSQ) in the memory system. Both of them aim to shrink the time to read and write the data to memory.

1) Data Cache: The purpose of the cache is to reduce the latency in the data transferring process. Comparing with the memory latency, which is over 10 cycles in our project setting (and possibly more in real-world processors), the cache in our

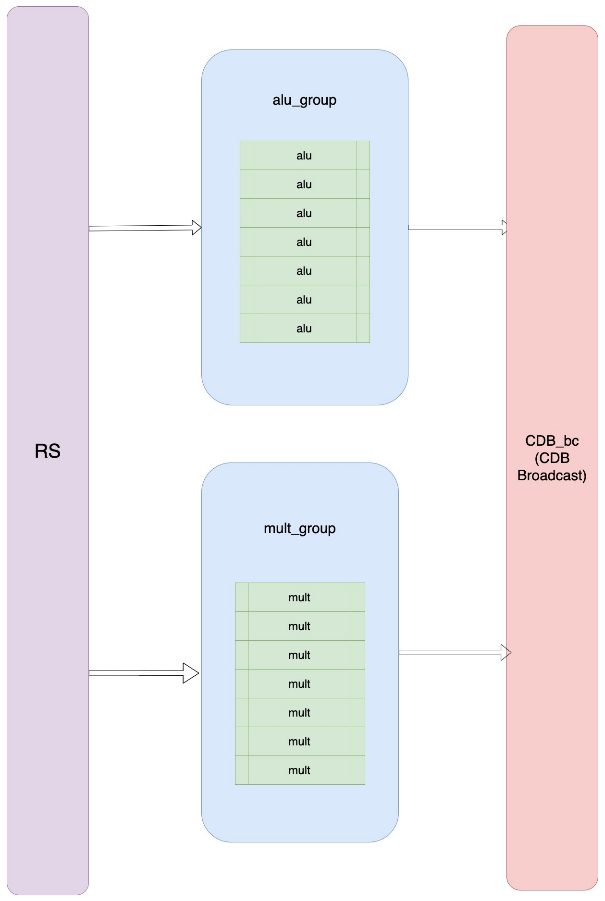


Fig. 7. Specific Design of Execution and connection with CDB broadcasting.

design can be accessed in only one cycle. As a result, the CPI of a memory intensive program is expected to decrease, and, thus, the performance of a processor will improve a lot by using caches.

The data cache is direct-mapped with 8B cache lines. We will use separate caches for instructions(I-cache) and data(D-cache). The instruction cache's size is 1KB and the data cache's size is 4KB. The data cache is designed to be read-allocate, write-back, single input and single output, and support internal forwarding when read or write happens at the same cycle as allocation. This means that there will be an allocation from the memory to the cache whenever there is a read or write and the tag is mismatched or the entry is invalid. The data will become dirty after being written and write-back to memory when the entry is evicted. In addition, when a read or write causes eviction and allocation, the cache will internally forwarding the allocation data to the output, with desired data modification, and store the data into the cache entries.

As shown in Fig.8, our data cache will divide the address of any data into four parts, offset (2:0), index (7:3), tag(15:8), unused(31:16). The control bits stored in our cache entries are a tag, a valid bit and a dirty bit. The data size of a data cache entry is 64bits.

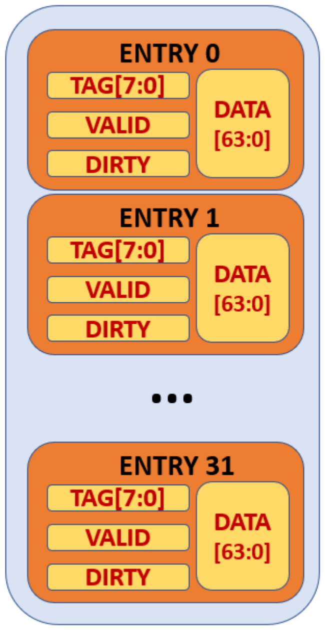


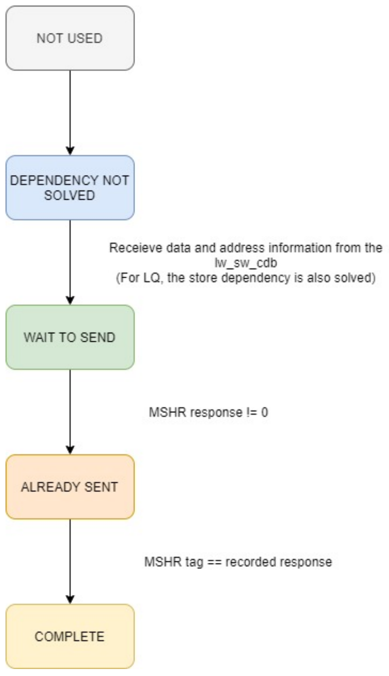
Fig. 8. The design sketch of the data cache. The cache has 32 direct-mapped entries. Each of the entry will store 64 bits of data, 8 bits of tag, 1 bit valid and 1 bit dirty.

The cache is accessed with an address and a read or write request. The cache will respond with a grant signal. If the read or write can be completed directly at the next clock rising edge, the grant signal will be high. Otherwise, the grant signal will remain low. The entries in the cache will become valid after allocation and become dirty if a write instruction has modified the data inside the entry. If the data requested by the instructions are not available in the cache, the grant signal will be low. In this case, the MSHR will look into the memory and access the cache again with an allocation request. The data cache will forward data to read or modify data for write when the allocation is combined with a read or write.

2) *Load Store Queue (LSQ):* The purpose of LSQ is to read the data from the memory as soon as all the data dependency that instruction needs are resolved (Fig.9). Since the memory latency is very high compared with the local register and cache, the existence of the LSQ will greatly improve the performance of a processor by getting the data from memory before it is required by the other instructions.

In our processor, we implemented a seperate LSQ, with each size to be 8. There is no data forwarding feature in LSQ.

The SQ will store the sw instructions in the dispatch stage. SQ will receive the sw address and data information from lw_sw_CDB. When the instruction address and data are solved by RS and the SQ detects this instruction arrives at the RoB head and it is a valid instruction, it will send the sw to the MSHR. Once it receives the completion signal from MSHR, it will tell the RoB that this store is ready to be committed. The LQ will copy all of the SQ entries every time there is a lw dispatched. The SQ copy will be updated by cdb and MSHR to LSQ packet. The lw will be send to MSHR only when its address is solved and all previous sw dependency is solved. Once the MSHR sends the load package back to LQ, it will change the state to be COMPLETED. It will be committed when this instruction hits the head of RoB.



NOT USED
DEPENDENCY NOT SOLVED
Receieve data and address information from the lw_sw_cdb
(For LQ, the store dependency is also solved)
WAIT TO SEND
MSHR response != 0
ALREADY SENT
MSHR tag == recorded response
COMPLETE

Fig. 9. LSQ state detail. When dispatch the instructions, LSQ will record the instructions if they are lw or sw. For sw instructions, it will stay in WAIT TO SEND state after it receives the data and address from the lw_sw_cdb, until this instruction reaches the head of RoB. For lw, after receives the address from cdb, it will check the sq copy to see whether all store dependency are solved. If solved, it will send the corresponding packet to the MSHR and reaches COMPLETE state after it receives the result from MSHR. The lw will be committed when it reaches the head of RoB.

3) MSHR: The MSHR controls the interface between LSQ, data cache and memory arbitrator (Fig.10). The size of it is set to be 15. The MSHR will receive the packet from LSQ. SQ packet has higher priority than LQ packet. The basic idea of MSHR is that it first checks whether the data is in the data cache, if yes, then it will send the data back to LSQ. If not, it will evict the dirty data to memory and allocate data from the memory, then ask data cache for the required data. During the time waiting memory for the allocate data and evicting data, the MSHR will change another LSQ packet to visit the data cache.

4) Memory Arbitrator: This modules solves the inst cache controller and MSHR memory access conflict. In our design, the MSHR request has the higher priority.

III. ANALYSIS

A. Overall Performance

Figure 12 shows the CPI (cycles per instruction) of our processor running different testcases. The processor has an average CPI 3.05 on the assembly test cases (without some trivial cases.) The clock period is 24.8 ns, which is relatively large. The critical path is between RS and execution units. The second longest critical path has 14.8 ns delay, which is from MSHR to LSQ. The first critical path could be reduced by adding latch between the RS and the execution units and shrink the size of the RS. The second critical path can be reduced by reducing the MSHR size.

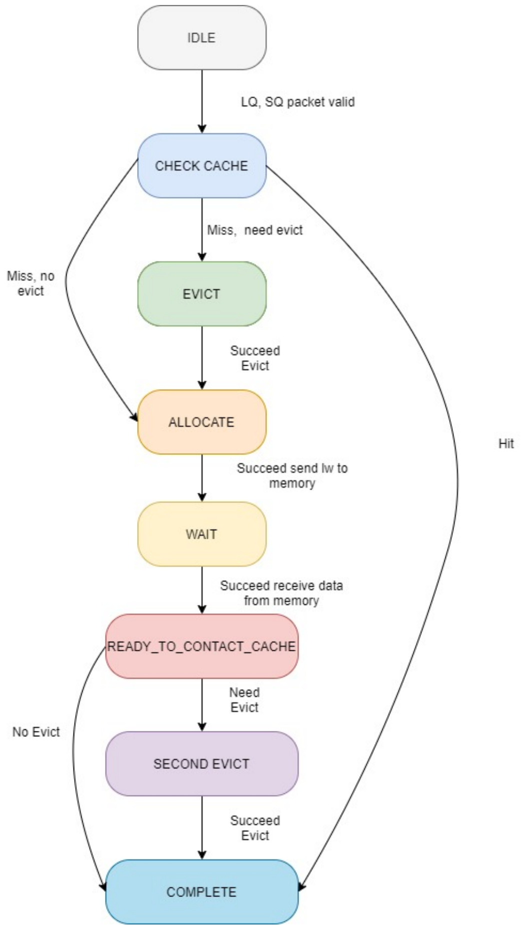


Fig. 10. The MSHR state detail. The MSHR will execute the instructions out of order. In one cycle, the MSHR will receive at most one valid LQ packet and one valid SQ packet. The valid packet will trigger one idle MSHR entry to CHECK CACHE state and send the entry place plus 1 as the MSHR response to LQ and SQ. If there is no empty entry left, the MSHR response will be 0. Every cycle MSHR will allow one MSHR entry to contact the data cache. Here, there are two states need to contact cache, CHECK CACHE and READY TO CONTACT CACHE. And we set CHECK CACHE state has higher priority. When check data cache, it will tell MSHR whether it's a hit, needs eviction or allocation. If hit, MSHR entry will grab the data, store it, and ready to be selected to send the data to LSQ. If not, the MSHR will entry the corresponding state to get and send the required data to memory. After MSHR gets the data it needs and allowed to contact cache, it will read or write the cache based on the instruction. The SECOND EVICT state serves the special situation. During the gap between CHECK CACHE and READY TO CONTACT CACHE, if the same cache block has been written new data, then it needs second time eviction. Once this is done, the MSHR entry will reach the COMPLETE state. After selected to send data to LSQ, it will go to IDLE state.

B. Prefetcher and Inst cache

We perform an analysis on how much instructions we should prefetch when there is a miss in the instruction cache. Because

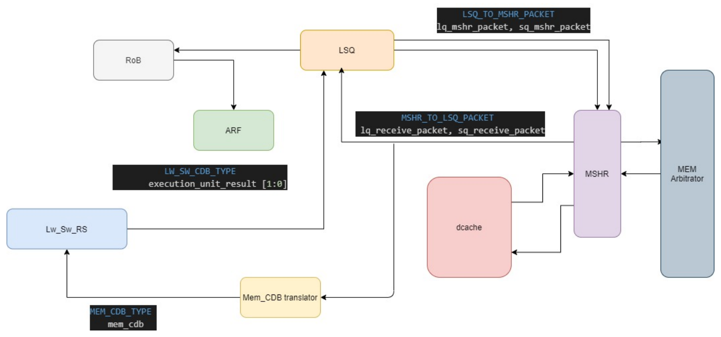


Fig. 11. Overview of the memory system part. The LSQ will send the packet to the MSHR. The MSHR will execute the lw or sw instructions with data cache only. The interface between memory and MSHR only serves for the needs of data cache.

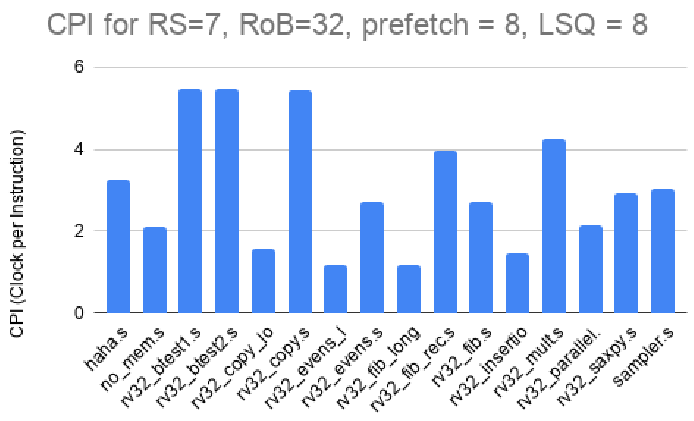


Fig. 12. The detail of our final design in terms of CPI over the test cases. The btest test cases has a high CPI because the test cases contains a lot of branches and these branches are not predictive. Also, the branch gap is big, so the prefetcher also not boosts the performance.

this value will change based on the implementation of the instruction cache, here we keep the instruction cache to be direct mapped, 32 × 8 cache line in size. We varied the prefetch length from 4 to 8. We can see (Fig. 20) that for some test cases, prefetch 4 instructions will have significantly lower cache hit rate compared to length 6 and 8. Prefetching with length 8 has the lowest CPI (Fig.14) in all test cases, so we can expect that the memory latency due to insturction fetching can be significantly decreased with a prefetching size of eight. As a result, we finally choose 8 as our prefetch length.

C. Local Branch Predictor

Here we will discuss the performance of our two-bit local branch predictor, whose size is 16. We can see that the hit rate (Fig.15) of our local branch predictor can reach 0.7 in most of the test cases. Comparing with the 0.25 prediction rate of always-not-taken branch predictor, our local branch predictor has a much higher accuracy. In terms of CPI, we can see that the local branch predictor improves the CPI by about 0.3 on average. (Fig.16).

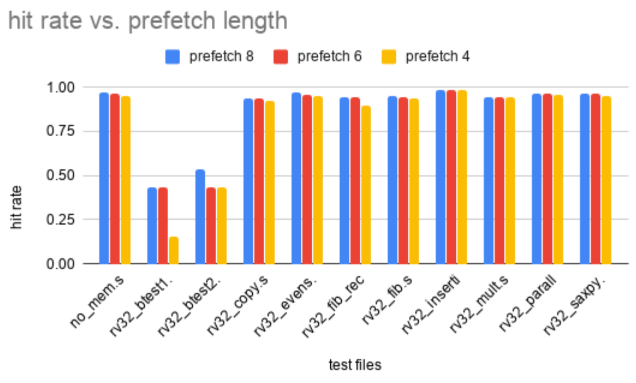


Fig. 13. The inst cache hit rate varies with the prefetch length. When set the prefetch length as 8, the inst cache hit rate can reach about 0.93 for most test cases. The btest has a low hit rate due to the natural of the test cases.

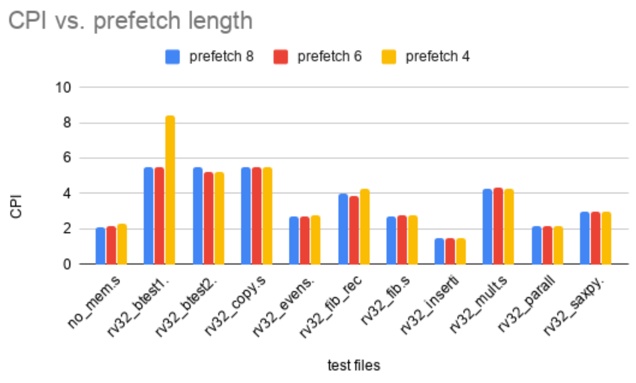


Fig. 14. The CPI varies with the prefetch length. Here we delete some short test cases to ensure the correctness. We can see that prefetch length 8 has slightly CPI improvement compared with length 6, but has great improvement compared with length 4.

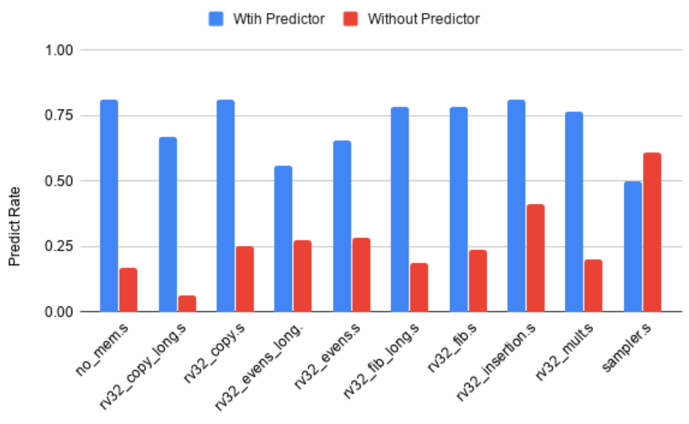


Fig. 15. The detail of predict rate on each test cases. We can see there exists a big improvement between the always-not-taken branch predictor and the local branch predictor in terms of the predict rate. The sampler.s test case shows that local branch predictor has a worse predict rate. This may be caused by the branch pattern in the test case.

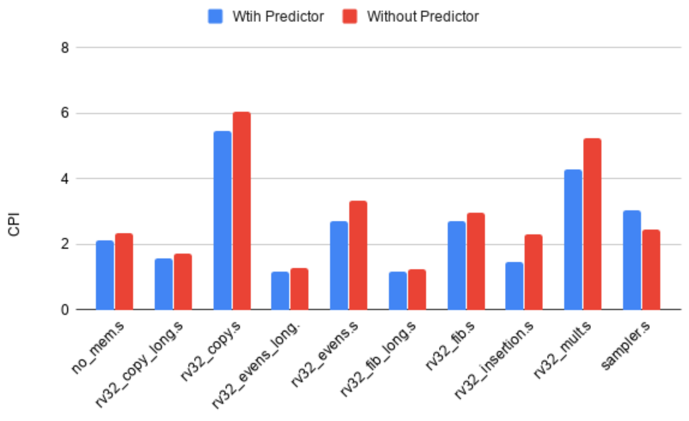


Fig. 16. The detail of CPI on each test cases. The improvement in prediction rate gives us about 0.3 clock improvement in average. It is worth noticing that in sampler.s test case, the local branch predictor version has a higher CPI compared with the always-not-taken branch predictor.

D. ROB and RS SIZE

Here we analysis the RoB and RS size. Since the RoB and RS both serve as a buffer for the instructions, increase the RoB size will increase the RS stall rate. So here we fixed our RS size to be 7 for each type of RS. By varing the RoB size, we can see how it will influence our CPI (Fig.17). We set the Rob size to be 32 because itthe improvement it gives us is small when it is set to be 64. And big RoB size will hurt the clock period.

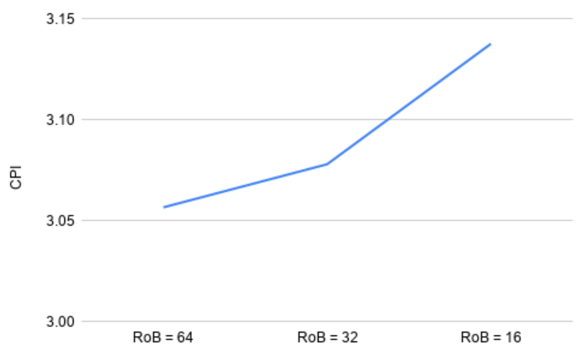


Fig. 17. This picture shows how CPI is infected by RoB size. We can see that CPI drops 0.07 when increase RoB size from 16 to 32, but only drops 0.02 when increases to 64. So we decided to leave RoB size to be 32.

E. LSQ

We analysis the size of LSQ. We keep the RoB size to be 32. We varies the size of LQ and SQ from 4 to 8 (Fig.18). The size of LQ and SQ are the same, although this can be analyzed later. We can see that when LSQ size is 2, there exists a huge lost in terms of CPI. The size 8 gives us the best CPI. Considering the LSQ is not on our critical path, so we set the LSQ size to be 8 for better CPI.

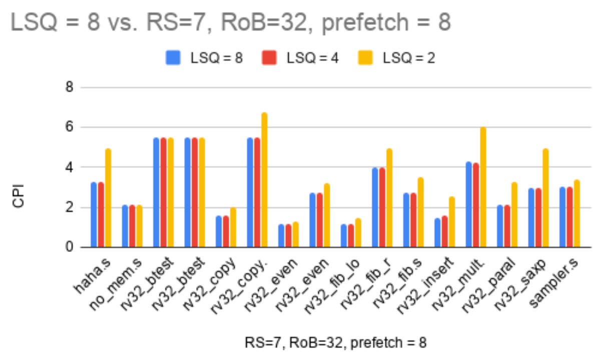
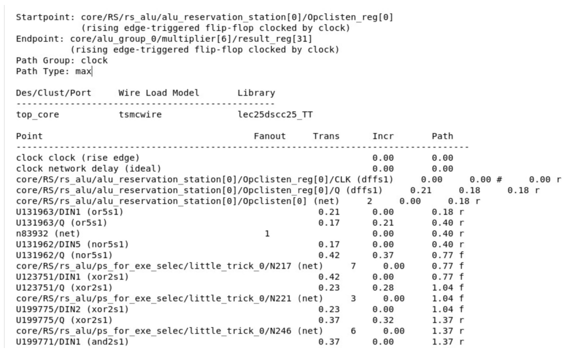


Fig. 18. This picture shows how CPI is infected by the LSQ size. LSQ size of 2 will hurt the performance on most of the test cases. The size of 8 gives us the better performance in most of the test cases.

F. Critical Path

In our design, the critical path happens at the RS to Function unit. The cause of it is our priority selector and the large number of RS entries. Due to the time limit, we did not improve our critical path. But the possible solutions will be add latches to the wake-up and selection logic. Or decrease the RS entry number.

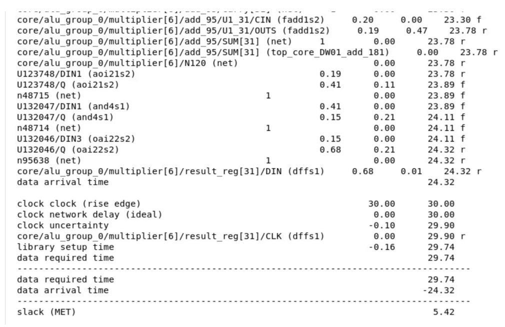


```
Startpoint: core/RS/rs_alu/alu_reservation_station[0]/Opclisten_reg[0]
            (rising edge-triggered flip-flop clocked by clock)
Endpoint: core/alu_group_0/multiplier[6]/result_reg[31]
          (rising edge-triggered flip-flop clocked by clock)
Path Group: clock
Path Type: max

Des/Clust/Port     Wire Load Model       Library
------------------------------------------------
top_core           tsmcwire              lec25dscc25_TT

Point                                        Fanout     Trans      Incr       Path
-----------------------------------------------------------------------------------
clock clock (rise edge)                                            0.00       0.00
clock network delay (ideal)                                        0.00       0.00
core/RS/rs_alu/alu_reservation_station[0]/Opclisten_reg[0]/CLK (dffs1)     0.00      0.00 #     0.00 r
core/RS/rs_alu/alu_reservation_station[0]/Opclisten_reg[0]/Q (dffs1)     0.21      0.18      0.18 r
core/RS/rs_alu/alu_reservation_station[0]/Opclisten[0] (net)     2     0.00      0.18 r
U131963/DIN1 (or5s1)                                    0.21       0.00       0.18 r
U131963/Q (or5s1)                                       0.17       0.21       0.40 r
n83932 (net)                                  1                    0.00       0.40 r
U131962/DIN5 (nor5s1)                                   0.17       0.00       0.40 r
U131962/Q (nor5s1)                                      0.42       0.37       0.77 f
core/RS/rs_alu/ps_for_exe_selec/little_trick_0/N217 (net)      7      0.00       0.77 f
U123751/DIN1 (xor2s1)                                   0.42       0.00       0.77 f
U123751/Q (xor2s1)                                      0.23       0.28       1.04 f
core/RS/rs_alu/ps_for_exe_selec/little_trick_0/N221 (net)      3      0.00       1.04 f
U199775/DIN2 (xor2s1)                                   0.23       0.00       1.04 f
U199775/Q (xor2s1)                                      0.37       0.32       1.37 r
core/RS/rs_alu/ps_for_exe_selec/little_trick_0/N246 (net)      6      0.00       1.37 r
U199771/DIN1 (and2s1)                                   0.37       0.00       1.37 r
```

Fig. 19. Critical path in rep part 1



```
core/alu_group_0/multiplier[6]/add_95/U1_31/CIN (fadd1s2)     0.20      0.00     23.30 f
core/alu_group_0/multiplier[6]/add_95/U1_31/OUTS (fadd1s2)     0.19      0.47     23.78 r
core/alu_group_0/multiplier[6]/add_95/SUM[31] (net)     1          0.00       23.78 r
core/alu_group_0/multiplier[6]/add_95/SUM[31] (top_core_DW01_add_181)     0.00     23.78 r
core/alu_group_0/multiplier[6]/N120 (net)                          0.00       23.78 r
U123748/DIN1 (aoi21s2)                                  0.19       0.00       23.78 r
U123748/Q (aoi21s2)                                     0.41       0.11       23.89 f
n48715 (net)                                  1                    0.00       23.89 f
U132047/DIN1 (and4s1)                                   0.41       0.00       23.89 f
U132047/Q (and4s1)                                      0.15       0.21       24.11 f
n48714 (net)                                  1                    0.00       24.11 f
U132046/DIN3 (oai22s2)                                  0.15       0.00       24.11 f
U132046/Q (oai22s2)                                     0.68       0.21       24.32 r
n95638 (net)                                  1                    0.00       24.32 r
core/alu_group_0/multiplier[6]/result_reg[31]/DIN (dffs1)     0.68      0.01     24.32 r
data arrival time                                                             24.32

clock clock (rise edge)                                           30.00       30.00
clock network delay (ideal)                                        0.00       30.00
clock uncertainty                                                 -0.10       29.90
core/alu_group_0/multiplier[6]/result_reg[31]/CLK (dffs1)          0.00       29.90 r
library setup time                                                -0.16       29.74
data required time                                                            29.74
-----------------------------------------------------------------------------------
data required time                                                            29.74
data arrival time                                                            -24.32
-----------------------------------------------------------------------------------
slack (MET)                                                                    5.42
```

Fig. 20. Critical path in rep part 2

IV. Workload Distribution

- Wentao Zhang (20%): Inst Cache, Inst Controller, Prefetcher, Inst stage, BTB, Local Branch Predictor, PC

Stage, RAT, RoB, ARF, LSQ, MSHR, Top Module, Test, Debug, Analysis

- Ruiqi Wang (20%): Reservation Station, Data Cache, Testbench, Top Module, Test, Debug, Analysis
- Yuanqing Lou (20%): Reservation Station, Testbench, Top Module, Test, Debug, Analysis
- Xingyu Wang (20%): RoB, Local Branch Predictor, Return Address Stack, PC Register, Testbench, Test, Debug
- Qipeng Wang (20%): Execution Unit, CDB broadcast, Top Module, Testbench, Test, Debug

V. Conclusions & Future Improvements

We have build a two-way superscalar processor with local branch predictor and direct mapped cache. Our design has clock period of 24.8 ns, with average CPI(cycle per instruction) of 3.05. When running long test cases, our CPI is about 3, which is very satisfying. However, there are still a few improvements we can make.

First, introduce the data forwarding in the LSQ. During the analysis, we found that the temporal locality is pretty high in the test programs. Also, the memory latency is very high. So implementing data forwarding will increase the performance a lot.

Second, pipeline the wake-up and select part in RS. In our design, wake-up and select part is our critical path, which greatly limits our clock period. By pipelining it, we can improve our clock period.

With these improvements, we are confident that our clock period will be less than 20 ns, and our average CPI will be close to 2.5.

VI. Code Segment

```

//this module is a new parameterized priority selector
//Modification: lengths of input values can be set to parameter.

//`timescale 1ns/100ps
module PS(
    //inputs
    req, //ready_to_issue list
    avail, //ready_to_receive list
    //outputs
    gnt, //gnt list, chosen from req
    occupy, // occupy list, chosen from avail
    bus// hard to explain, example:
//
req = 10101010
// avail = 10111110
//
gnt = 10101010
//occupy = 10111000
//
bus = 10000000 00000000 00100000 00001000 00000010 00000000 00000000 00000000
);

//parameters
parameter WIDTH = 7;//length of req
parameter LENGTH = 7;//length of avail

//inputs
input [WIDTH-1:0] req;
input [LENGTH-1:0] avail;

//outputs
output logic [WIDTH-1:0] gnt;
output logic [LENGTH-1:0] occupy;
output logic [WIDTH*LENGTH-1:0] bus;

//internal stuff
logic [3:0] num_req;
logic [3:0] num_avail;
logic [WIDTH-1:0] req_transformed;//req with all its 1s
logic [LENGTH-1:0] avail_transformed;//avail with all it
logic [WIDTH*WIDTH-1:0] temp_bus;//hard to explain
//example: given gnt = 01011010
//then temp_bus will be 00000000 00000000 00000000 00000


//assign req_transformed and avail_transformed
little_trick0 little_trick_0(.a(req),.b(req_transformed)
little_trick1 little_trick_1(.a(avail),.b(avail_transfor


//assign num_req, num_avail
always_comb begin
    num_req = 0;
    num_avail = 0;
    for (int i = 0; i < WIDTH; i = i + 1)
    begin
        num_req = num_req + req[i];
    end
    for (int i = 0; i < LENGTH; i = i + 1)
    begin
        num_avail = num_avail + avail[i];
    end
end

//assign gnt and occupy
int j;
int k;
always_comb begin
    if (num_req > num_avail)
    begin
        occupy = avail;
        gnt = 0;
        j = 0;
        for (int i = 0; i < WIDTH; i = i + 1)
        begin
            if (req[i] && avail_transformed[
            begin
                gnt[i] = 1;
                j = j + 1;
            end
        end
    end
    else if (num_req < num_avail)
    begin
        gnt = req;
        occupy = 0;
        k = 0;
        for (int i = 0; i < LENGTH; i = i + 1)
        begin
            if (avail[i] && req_transformed[
            begin
                occupy[i] = 1;
                k = k + 1;
            end
```

```
                end
        end
        else
        begin
                gnt = req;
                occupy = avail;
        end
end

//assign temp_bus and bus
int m;
always_comb begin
        m = 0;
        temp_bus = 0;
        for (int i = 0; i < WIDTH; i = i + 1)
        begin
                if (gnt[i]) begin
                        temp_bus[m*WIDTH +: WIDTH] = 1<<i;
                        m = m + 1;
                end
        end
end

int h;
always_comb begin
        h = 0;
        bus =0;
        for (int i = 0; i < LENGTH; i = i + 1)
        begin
                if (occupy[i])
                begin
                        bus[i*WIDTH +: WIDTH] = temp_bus[h*WIDTH +:WIDTH];
                        h = h + 1;
                end
        end
end
endmodule
```