DSD Final Report

Group5

電機三 b09901014 廖苡鈞
電機三 b09901057 陳俊霖
電機三 b09901101 劉心宇

Outline

I. Baseline

A. Structure

1. Architecture
2. Modules of 5 stages
3. Forwarding and Hazard Unit

B. AT Value

II. Extension

A. Branch Predictor

1. Architecture
2. Branch Predictor Module
3. Comparison

B. Compress

1. Structure
2. Testbench

C. Qsort

1. Compression
2. Branch Prediction
3. AT Value

D. Conclusion

1. Future improvement
2. What have we learned

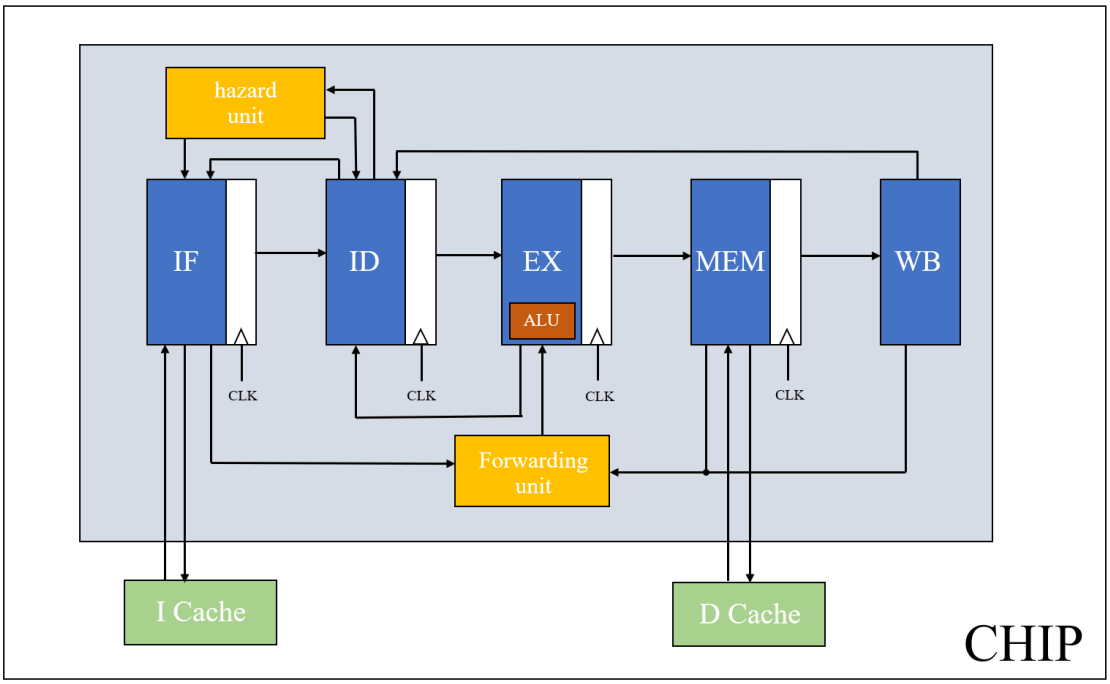
- # Baseline

A. Structure

1. Architecture

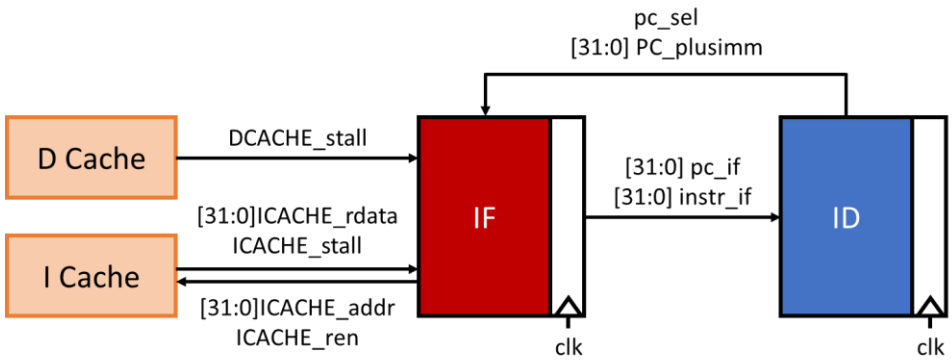
We create 5 modules corresponding to each stage, Forwarding unit, hazard unit and 2 caches are connected to them as well. To ensure the correctness of this pipeline, registers are set in the end of each stage.

The following figure is the architecture of our RISC-V chip. Direction of each arrow means I/O direction between two modules.



2. Modules of 5 stages

- **Instruction Fetch (IF)**

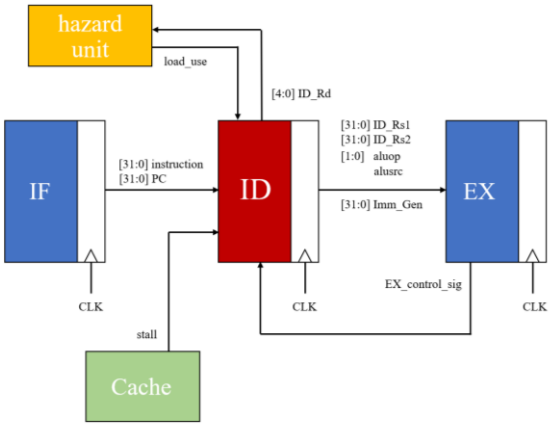


The main function of this module is to calculate Program Counter (PC) and request corresponding instruction from **Instruction Cache** (I Cache). The following figure indicts I/O relationship between **IF** and other modules.

	normal	stall	pc_sel
Instruction	ICACHE_rdata	Instruction	0
PC	PC + 4	PC	PC_plusimm

- **Instruction Decode (ID)**

The following figure indicts I/O relationship between **ID** and other modules



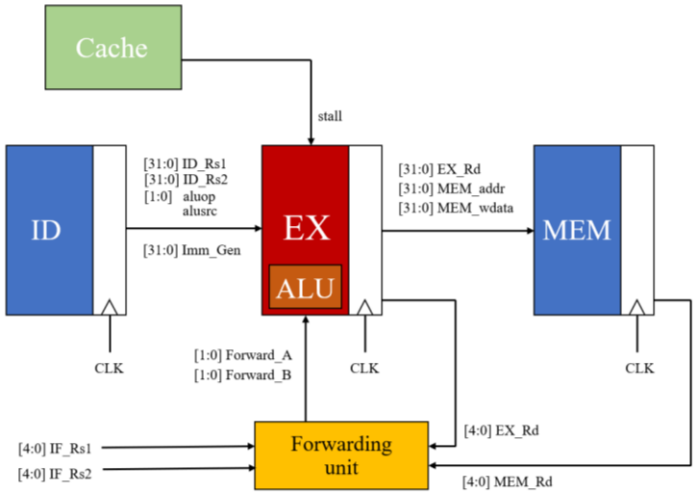
.

The function of the ID stage is to read/write the data of reg file, pass the correct rs1 data, rs2 data, immediate data and control signal to EX stage, detect whether to jump or branch, pass the new address to IF stage when jump and branch occur, detect whether to stall and pass the stall signal to IF stage.

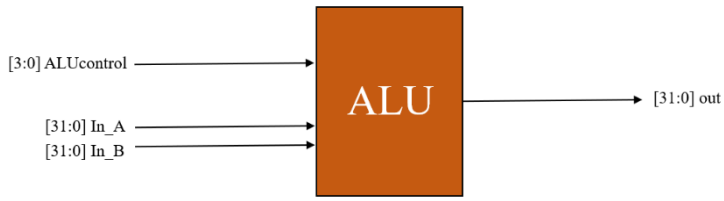
Because we use the positive edge triggered DFF in reg file, when data in WB stage is used in ID stage, cannot directly use the data in reg file, need to forwarding it from WB stage.

- **Instruction Execution (EX)**

The following figure indicts I/O relationship between **EX** and other modules.



Arithmetic is the main function of **EX**. There is an **ALU** module connected in **EX**, which I/O signals are show below.

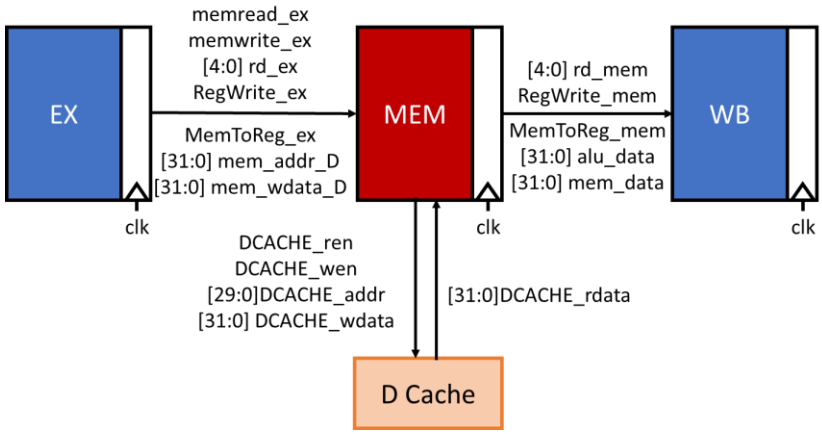


This table indicts *ALUcontrol* and corresponding operation.

ALUcontrol	operation
4'b0000	AND
4'b0001	OR
4'b0010	ADD
4'b0110	SUB
4'b0111	SLT
4'b1001	XOR
4'b1010	SLL
4'b1011	SRA
4'b1100	SRL

- **Data Memory (MEM)**

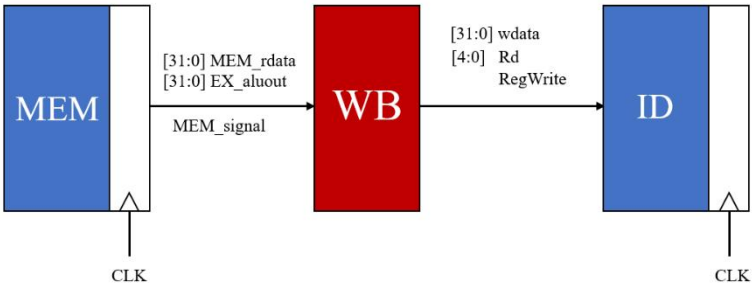
  The following figure indicts I/O relationship between **MEM** and other modules.



The main function of this state is to communicate with D cache in order to get the data required or to write data into data memory. If stall happens, pipeline stages remain the same value and thus are stopped. The data read from D cache and the data calculated from EX stage will both be sent to WB stage.

- **Write Back (WB)**

  The following figure indicts I/O relationship between **WB** and other modules.



There is no register in WB since data should write back to **ID**, where the register file is, immediately. However, data will write back to the register file by sequential circuit in **ID** as a result that **Forwarding Unit** has to deal with that case.

3. Forwarding and Hazard Unit

- **Forwarding Unit**

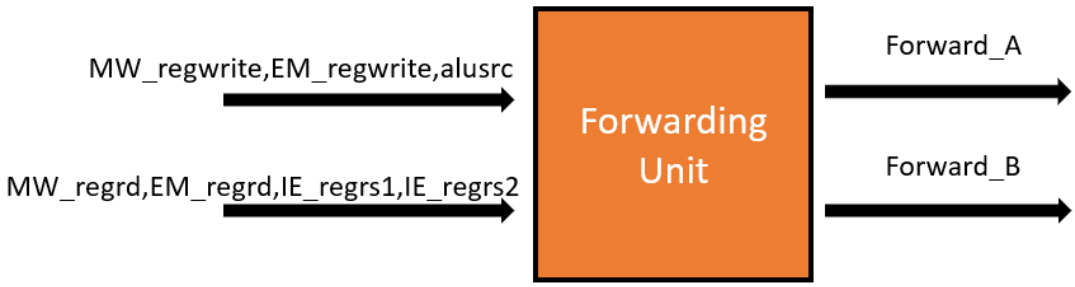
```
module Forwarding_unit(ForwardA, ForwardB, MW_regwrite, MW_regrd, EM_regwrite, EM_regrd, IE_regrs1, IE_regrs2, alusrc);
    input            MW_regwrite,EM_regwrite,alusrc;
    input      [4:0] MW_regrd,EM_regrd,IE_regrs1,IE_regrs2;
    output     [1:0]  ForwardA;
    output     [1:0]  ForwardB;


    assign ForwardA = (EM_regwrite && (EM_regrd == IE_regrs1))? 2'b10:
                      (MW_regwrite && ~(EM_regwrite && (EM_regrd == IE_regrs1))&&(MW_regrd==IE_regrs1))? 2'b01:2'b00;

    assign ForwardB = (EM_regwrite && (EM_regrd == IE_regrs2))? 2'b10:
                      (MW_regwrite && ~(EM_regwrite && (EM_regrd == IE_regrs2))&&(MW_regrd==IE_regrs2))? 2'b01:
                      (alusrc)? 2'b11:2'b00;

endmodule
```

When a instruction uses the data that has been changed and hasn't been stored in the reg file, have to forwarding the data from MEM stage or WB stage. Besides, choose the most recently changed data if the data is in the MEM stage and WB stage at the same time.



ForwardA	Meaning
2’b10	Forwarding the data from the MEM stage
2’b01	Forwarding the data from the WB stage
2’b00	No forwarding

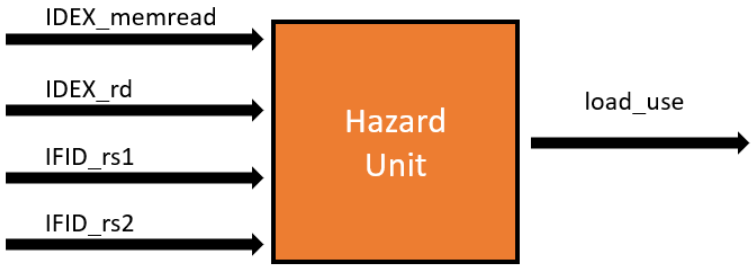
ForwardB is as the same way as ForwardA

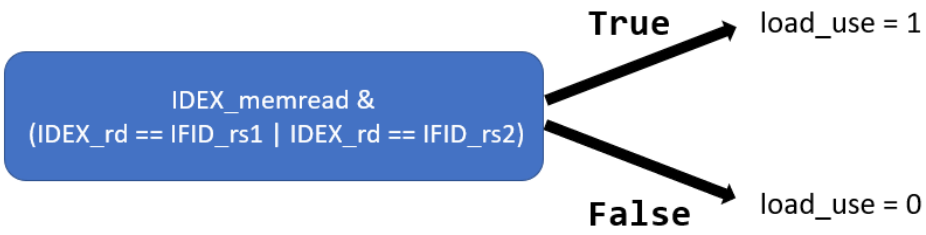
.

- **Hazard Unit**

```
module hazard_unit(
    input IDEX_memread,
    input [4:0] IDEX_rd, IFID_rs1, IFID_rs2,
    output load_use
);
    assign load_use = (IDEX_memread & (IDEX_rd == IFID_rs1 | IDEX_rd == IFID_rs2))? 1:0;
endmodule
```

Hazard unit is used to detect whether the load-use case occurs. Load-use case occurs when the instruction in EX stage is LW and the instruction in ID stage uses the rd of the instruction in EX stage as rs1 or rs2. If Load-use case occurs, pull load_use signal to high and stall the IF and ID stage.





B. AT Value

1. Area: 238238 um^2
2. sdc time: 10 ns
3. tb time: 10 ns
4. Cycle (no hazard)：1915 ns
5. Cycle (has hazard)：24245 ns

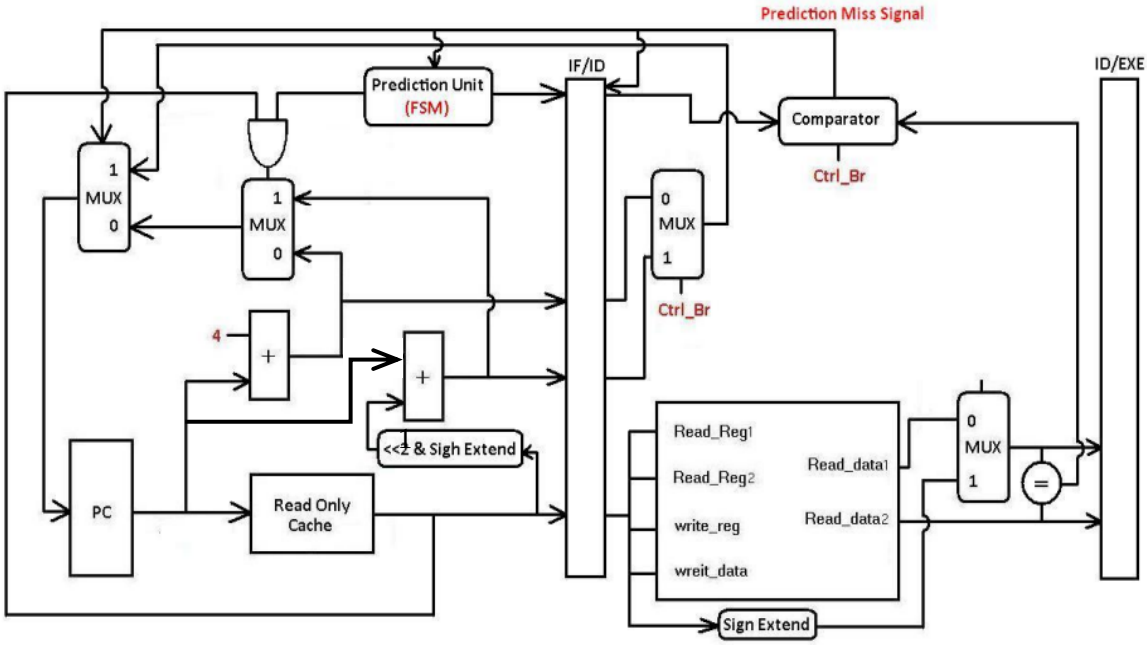
- # Extension

A. Branch Predictor

We implement 5 kind of branch predictor: all not taken, all taken, 1-bit predictor, 2-bit predictor, 2-level 1-bit predictor relatively.

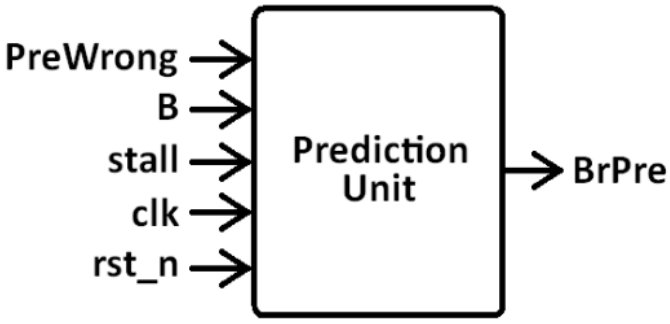
1. Architecture

This is the architecture of branch predictor in IF and ID stage.

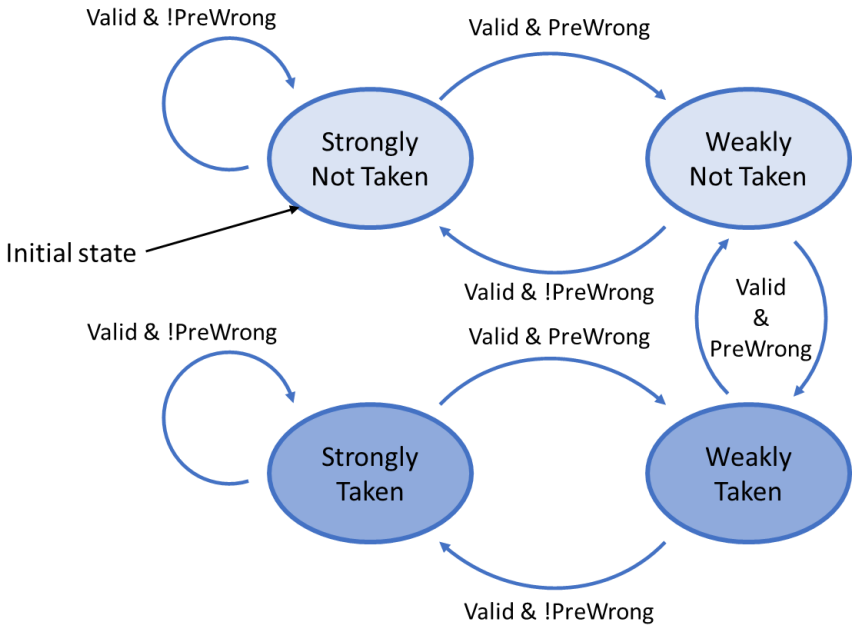


Branch instruction is decoded in IF stage, and Branch Predictor Unit (BPU) will determine whether we should branch. For BrPre signal, 0 stands for not branch and 1 stands for branch. It will be sent to ID stage to check whether we predicted correctly. If not, ID stage sends back PreWrong signal and the correct PC. Also, BPU should take corresponding actions.

2. Branch Predictor Module

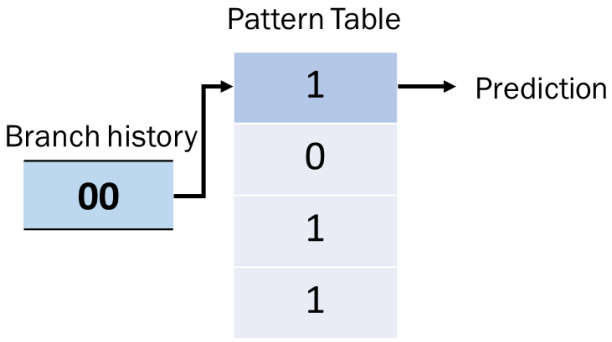


We implemented 3 kinds of BPU: always not taken, 2-bit predictor and 2-level predictor.



- 2-bit Predictor:

  In 2-bit predictor, state is controlled by Valid and PreWrong signal. Valid signal indicates the cycle that PreWrong signal is valid, and state should change according to the PreWrong signal. We chose strongly not taken to be the initial state.

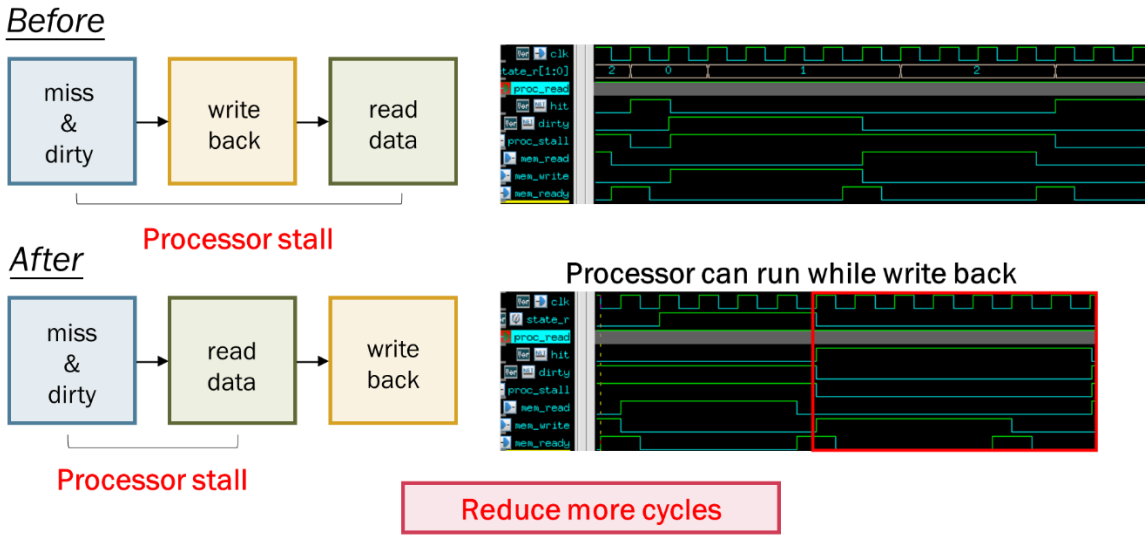


  2-level predictor remembers the history of the last 2 occurrences of the branch and uses one saturating counter for each of the 4 history patterns. It works efficiently on regularly recurring pattern. We chose to store 0 in pattern table initially.

3. Comparison

Branch Prediction	Timing (ns)	Area (um^2)	AT
Always not taken	4565	X	X
2-bit predictor	4495	228449	1.03*10^9
2-level predictor	4155	229795	0.95*10^9

As we can see, 2-level predictor works the best in this case, since the test pattern is regular.



B. Compress

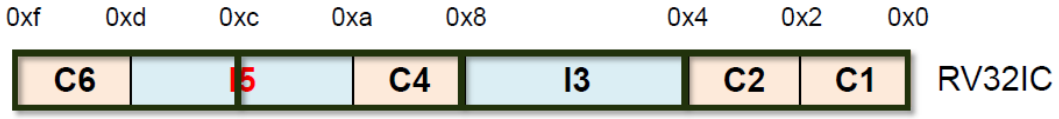
1. Structure

- Cache

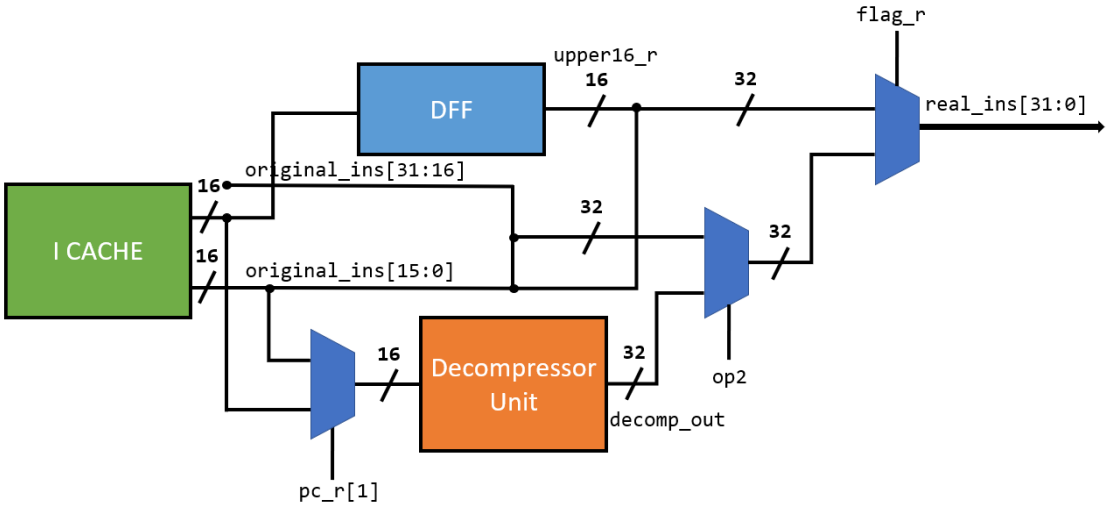
For our first idea, maintain the structure of the I cache to reduce the complexity of compression. Hence, the alignment of the I cache is as same as original one. Also, I cache outputs 32-bit data at once.

- IF stage

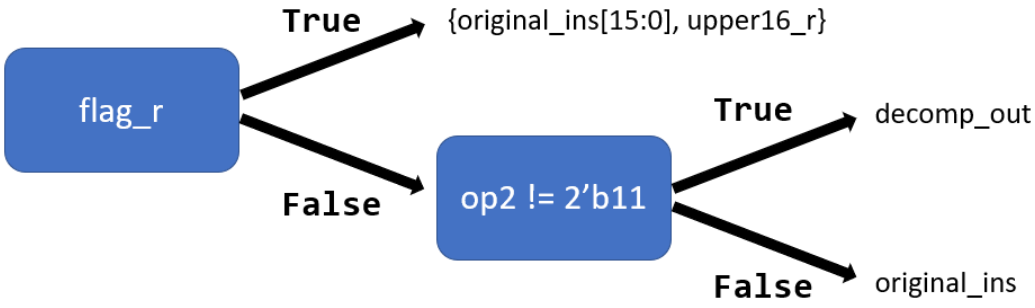
Since the alignment of the I cache is 32-bit and I cache outputs 32-bit data at once. Need to stall when fetching the non-compressed data whose address is between 0x2~6, 0x6~a, 0xa~d, or 0xd~next block's 0x2. Also, have to store the upper 16-bit of the non-compressed data. Need two cycles to fetch the cross-word-aligned data, i.e. data whose address is between 0x2~6, 0x6~a, 0xa~d. Need at least two cycles to fetch the cross-block data, i.e. data whose address is between 0xd~next block's 0x2. Since the data of next block may not be in the I cache, need to stall for several cycles to fetch the data of next block from memory.
For example, since the address of I5 instruction is between 0xa~d, need two cycles to fetch the correct instruction.



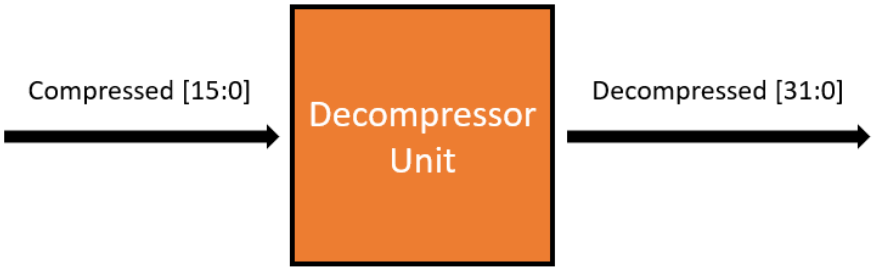
Below is the structure of handling the compression instruction. flag_r denotes whether the instruction is cross-word-aligned data. op2 is the 2-bit opcode of the current compressed instruction. pc_r[1] denotes the place of the current instruction. Store the upper 16-bit of the non-compressed data in DFF.



Below is the determination of real_ins.



- Decompressor unit



Convert the 16-bit compressed data into 32-bit decompressed data.

1. First, use the [1:0] of the compressed data to divide the 16 instructions into 3 types.

Compressed [1:0]	Instructions
2'b00	LW, SW
2'b01	SLLI, JR, MV, JALR, ADD
2'b10	NOP, ADDI, JAL, J, SRLI, SRAI, ANDI, BEQZ, BNEZ

2. For each type, convert the 16-bit compressed data into 32-bit decompressed data by specifying their opcode and funct3.

A. 2'b00: LW, SW

Compressed [15]	Instructions
0	LW
1	SW

B. 2’b01: SLLI, JR, MV, JALR, ADD

Compressed [15:13]	Instructions
3’b000	ADDI
3’b001	JAL
3’b100	SRLI, SRAI, ANDI
3’b101	J
others	BEQZ, BNEZ

For SRLI, SRAI, ANDI:

Compressed [11:10]	Instructions
2’b00	SRLI
2’b01	SRAI
others	ANDI

For BEQZ, BNEZ:
Directly place Compressed[13] to Decompressed[12].

C. 2’b10: NOP, ADDI, JAL, J, SRLI, SRAI, ANDI, BEQZ, BNEZ

Compressed [15]	Instructions
0	SLLI
1	JR, MV, JALR, ADD

For JR, MV, JALR, ADD:

Compressed [6:2]	Instructions
0	JR, JALR
otherwise	MV, ADD

For JR, JALR:
Directly place Compressed[12] to Decompressed[7].

For MV, ADD:

Compressed [12]	Instructions
0	MV
1	ADD

2. Testbench

- sdc_cycle = 4.0
- tb_cycle = 4.0
- Always not taken

	Timing (ns)	Area (um^2)	AT
Decompress	2494	240870	600729780
Compress	2114	253210	544887728

C. Qsort

1. Compression

- sdc_cycle = 4.0
- Always not taken

	Tb_cycle	Timing (ns)	Area (um^2)	AT
Decompress	4.0	673446	240870	1.62e11
Compress	4.2	624378	253210	1.58e11

From the result above, compress is better than decompress.

2. Breach Prediction

test1:

- sdc_cycle = 4.0
- tb_cycle = 4.6
- Total 8327 branch instructions

Branch Prediction	Prediction wrong	Error rate	Timing (ns)	Area (um^2)	AT
2-level 1-bit saturation predictor	4778	57.38%	667738	270044	1.80e11
2-bit predictor	4610	55.36%	667434	267729	1.786e11

test2:

- sdc_cycle = 4.0
- tb_cycle = 4.6
- Total 8327 branch instructions

Branch Prediction	Prediction wrong	Error rate	Timing (ns)	Area
2-bit predictor	4610	55.36%	667434	bigger
Always not taken	4885	58.66%	667351	smaller

From the result above, always not taken has the lowest AT value .

3. AT Value

Sdc cycle: 2.9 ns
Tb cycle: 2.9 ns
Timing: 437225.75 ns
Area: 284341 um^2
AT Value: 1.24*10^11

D. Conclusion

1. Future improvement:

In the implement of compression, we choose the most basic method. This method has to stall for more case. Hence, we could optimize our design by change the algorithm. For example, read 48 bit at one time or change the I-cache alignment from 32-bit to 16-bit.

2. What have we learned:

In this final project, we first learned how to cooperate with teammate, including how to divide the labor, how to communicate. Because this is a huge and complex project, we have more chance to communicate and cooperate. We think we make huge progress in this aspect.