VLSI 系統設計
-期末專題報告

Report

系級	電機 113
學號	F64096114
姓名	郭家佑

1、 系統簡介

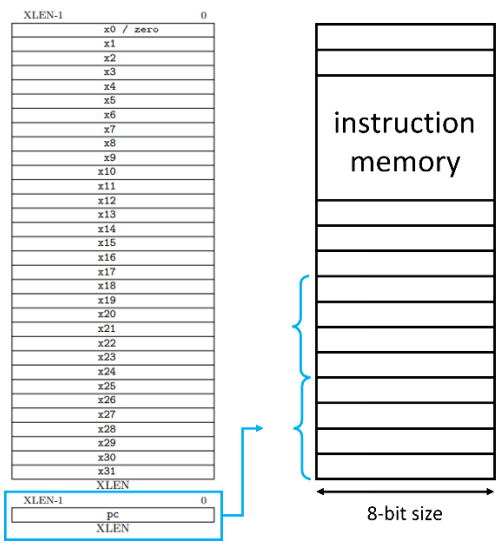
(1) 指令集格式

本次的指令集參考了 RISC-V 中的基本架構 RV32I，主要可以分成 R,I,S,B,U,J type 六種指令，每一種指令有不同的 opcode

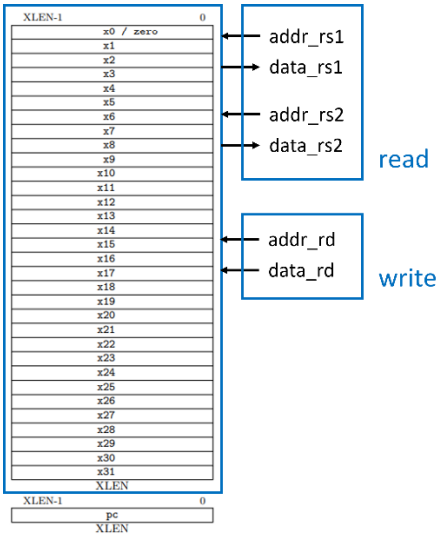
i. RV32I 特色

a. Program counter

- store the address pointing to instruction memory
- used to +4 as the next instruction



▲PC



▲Register file

b. Register file

- 32 registers compose this file
- two ports to read data
- one port to write data
- 5-bit (log232) address pointers

ii. Instruction type

R type 執行 Register 之間的運算

Ex: add => rs1+rs2 存入 rd

0000000	rs2	rs1	000	rd	0110011	R add
0100000	rs2	rs1	000	rd	0110011	R sub
0000000	rs2	rs1	001	rd	0110011	R sll
0000000	rs2	rs1	010	rd	0110011	R slt
0000000	rs2	rs1	011	rd	0110011	Rsltu

0000000	rs2	rs1	100	rd	0110011	R xor
0000000	rs2	rs1	101	rd	0110011	R srl
0100000	rs2	rs1	101	rd	0110011	R sra
0000000	rs2	rs1	110	rd	0110011	R or
0000000	rs2	rs1	111	rd	0110011	R and

I type 執行 Register 和常數間運算或 memory 讀取等

Ex: addi => rs1+imme 存入 rd

imm[11:0]	rs1	000	rd	0010011	I addi
imm[11:0]	rs1	010	rd	0010011	I slti
imm[11:0]	rs1	011	rd	0010011	I sltiu
imm[11:0]	rs1	100	rd	0010011	I xori
imm[11:0]	rs1	110	rd	0010011	I ori
imm[11:0]	rs1	111	rd	0010011	I andi

imm[11:0]	rs1	000	rd	0000011	I lb
imm[11:0]	rs1	001	rd	0000011	I lh
imm[11:0]	rs1	010	rd	0000011	I lw
imm[11:0]	rs1	100	rd	0000011	I lbu
imm[11:0]	rs1	101	rd	0000011	I lhu

S type 執行儲存至 data memory 的指令

Ex: sh => 將 rs2[15:0](half word)存入 mem(rs1+imm)的地址中

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw

B type 執行有條件的跳轉指令

Ex: beq => 比對 rs1、rs2 若相等則跳轉至 imme+current pc 的位置

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu

U type 執行常數的輸入(較 I type 的常數更大)

Ex: lui => 將 imm 存入 register rd 中的[31:20]

imm[31:12]	rd	0110111	U lui
imm[31:12]	rd	0010111	U auipc

J type 執行無條件跳轉

Ex: jalr => 跳轉至 imm+current pc 的位置，並將 current pc+4 存入 rd

imm[20|10:1|11|19:12]	rd	1101111	J jal

(2) 指令集格式欄位的名稱、長度、說明

Formats	32 Bits (RV32I)																													
	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R type	func7							rs2					rs1					func3			rd					opcode						
I type	imme[11:0]												rs1					func3			rd					opcode						
S type	imme[11:5]							rs2					rs1					func3			imme[4:0]					opcode						
B type	{ imme[12], imme[10:5] }							rs2					rs1					func3			{ imme[4:1], imme[11] }					opcode						
U type	imme[31:12]																				rd					opcode						
J type	{ imme[20], imme[10:1], imme[11], imme[19:12] }																				rd					opcode						

所有指令固定為 32bits

[6:0] 為 opcode 可判斷為上述六種 type 中的哪一種

[11:7] 為 register destination ，即最後存回 register 中的地址

[14:12] 為 function3 用來判斷其為各種 type 中的哪一項指令

[19:15] 為 register 1 ，即運算中被用到的第一號 register 的地址

[24:20] 為 register 2 ，即運算中被用到的第二號 register 的地址

[31:24] 為 function3 同樣用來判斷其為各種 type 中的哪一項指令

若該種指令不需要上述某些部件則以 imme(立即數)代替，而常數的排列方式也依各種指令或有多少不同處，可從上表見得。

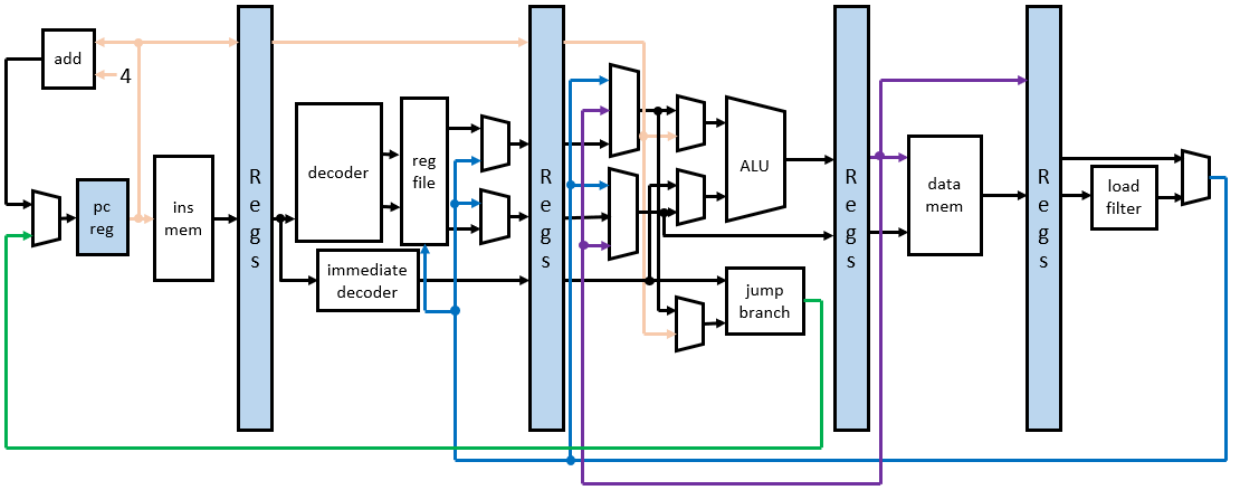
(3) Branch 指令與 Jump 指令的定址

本次 CPU 設計中 branch 指令為有條件跳轉，依照 function3 的值去選擇不同的條件，之後比對 rs1、rs2 之間的關係是否滿足跳轉的條件，其跳轉地址由 instruction 上的常數(imm) + 目前的 program counter 來獲得。

Jal、Jalr 為無條件跳轉指令，並且都會將目前執行的 program counter +4 存入 rd register 中，兩者的差異在於 jal 的跳轉地址由常數+目前的 program counter 來獲得，而 jalr 由 rs1 + 常數來獲得。

(4) 架構說明

i. 架構圖及說明



本次的 CPU 架構為 5 stage pipeline CPU，5 個 stage 可分為

Fetch： 選擇、更新 pc，並讀取 instruction。

Decode：

分解 instruction 中的各種欄位和常數，並提出 regfile 中所需的值。

Execute：

利用 ALU 計算指令中傳回 regfile 的值、data memory 讀寫時的地址、跳轉條件是否符合等。 Divider 負責執行除法運算。
Jump_branch :

負責運算正常跳轉時的跳轉地址。

Memory：

處理 data memory 的 store 和 load 指令。

Write_back:

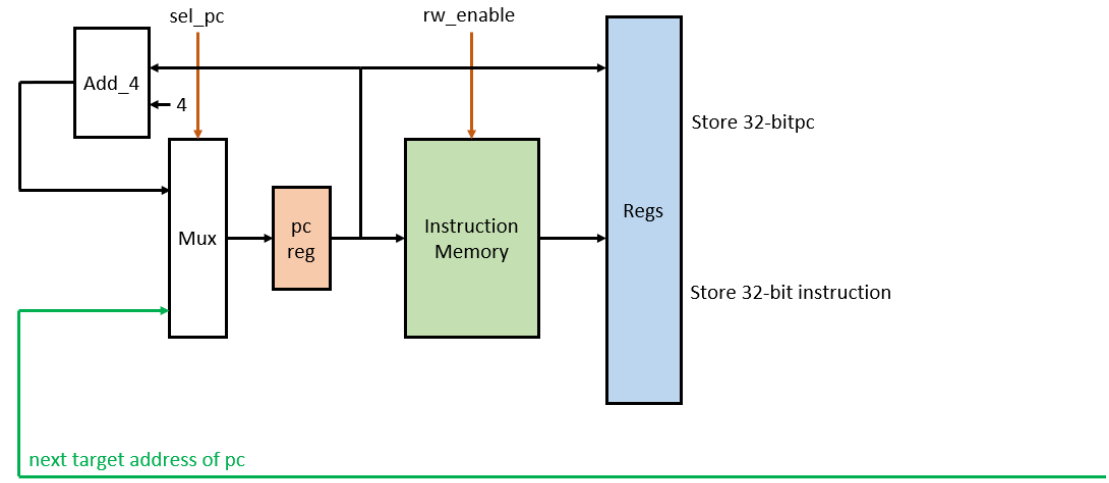
選擇寫回 regfile 中的值，load 值或 ALU 運算的結果。

regs：

利用四個 reg 去分割 CPU 成五個 stage 並儲存各個 stage 要傳去下個 stage 的值並在 posedge clk 時一同傳遞。

ii. 個別元件說明

Fetch

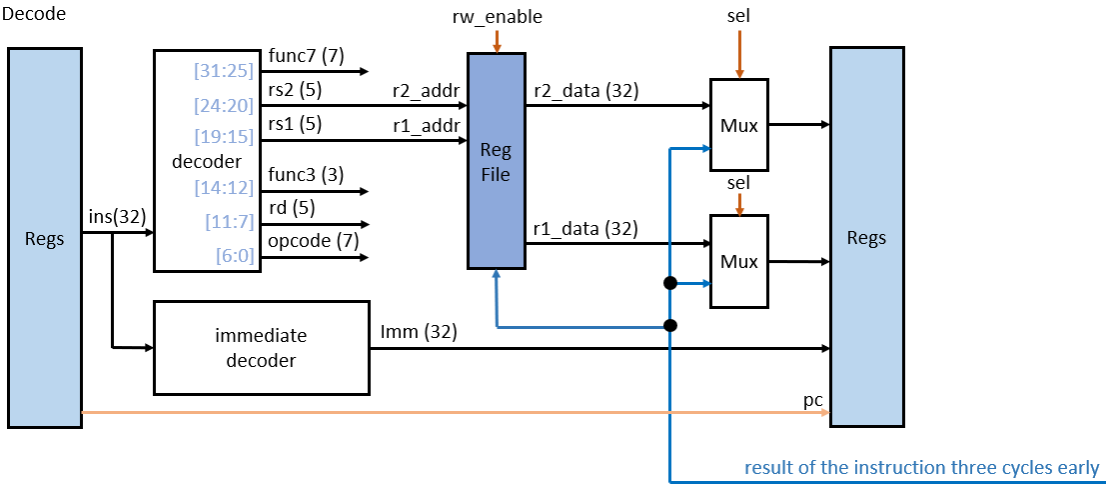


Add：將目前的 pc+4，1 代表 1byte，4 代表 1 word

Pc_reg：隨著 posedge clk 更新由 mux 所選擇的 pc

Instruction_memory：儲存指令，需以 pc 為地址來讀取指令

Decode



Decoder：

將前面提到指令中的 opcode、rd、function3 等 controller 需要的值切割出來

Imm_decoder：

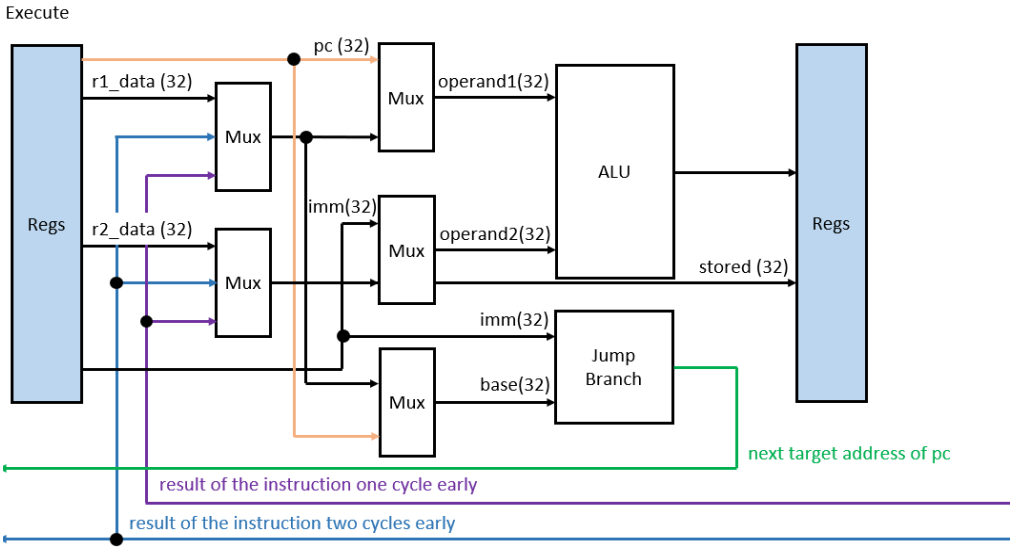
將每個指令中所含的常數提出並延長成 32bits

Reg_file：

內含 32 個 register 每次都會在 decoder 傳遞 rs1、rs2 index 後取出兩者內存取的值，在 write_back stage 時也會將回傳值存入 rd

Branch_predict：

特殊運算中預測跳轉時計算跳轉地址



ALU_mux：

連到 ALU 的兩個 mux 中上方的 mux 選擇(pc or rs1)、下方的 mux 選擇(rs2 or imm)。

ALU：

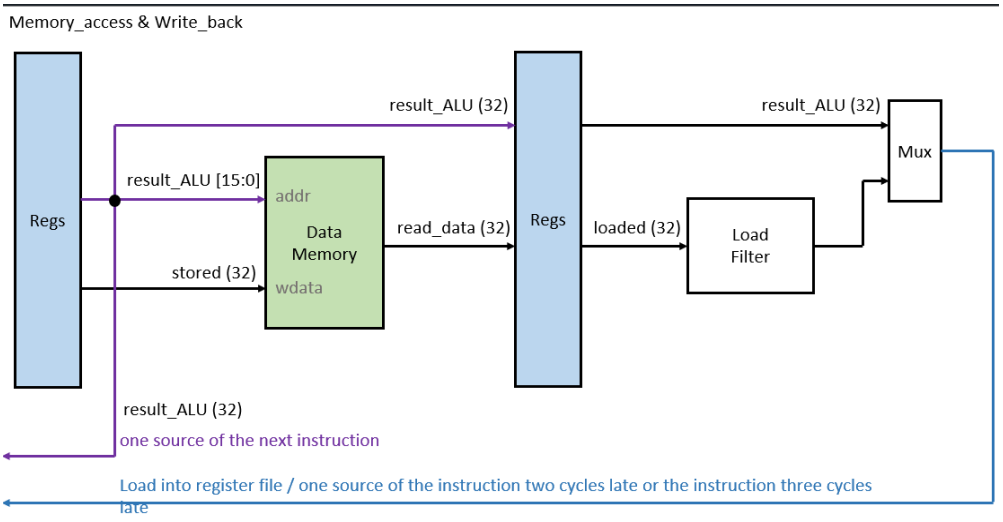
計算指令中傳回 regfile 的值、data memory 讀寫時的地址、Branch 條件是否符合等。

JB_mux：

選擇(pc(branch、jal) or rs1(jalr))。

Jump_branch：

用來計算跳轉之地址並將值傳到 fetch stage 的 mux。



Data_memory：

儲存、讀取時以 rs1+imm 作為 address，儲存 rs2 值進入 memory。

Load_filter：

load 時皆讀取 32bits，但當指令為 load halfword 或 load byte 時則需要將不需要的部分 flush 掉。

Write_back_mux：

選擇傳回 regfile 的值為 ALU_result 或是讀取出的資料。

2、 系統目前可執行之指令

imm[31:12]				rd	0110111	U lui
imm[31:12]				rd	0010111	U auipc
imm[20|10:1|11|19:12]				rd	1101111	J jal
imm[11:0]		rs1	000	rd	1100111	I jalr
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	B beq
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	B bne
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	B blt
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	B bge
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	B bltu
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	B bgeu
imm[11:0]		rs1	000	rd	0000011	I lb
imm[11:0]		rs1	001	rd	0000011	I lh
imm[11:0]		rs1	010	rd	0000011	I lw
imm[11:0]		rs1	100	rd	0000011	I lbu
imm[11:0]		rs1	101	rd	0000011	I lhu
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	S sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	S sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	S sw

imm[11:0]	rs1	000	rd	0010011	I addi
imm[11:0]	rs1	010	rd	0010011	I slti
imm[11:0]	rs1	011	rd	0010011	I sltiu
imm[11:0]	rs1	100	rd	0010011	I xori
imm[11:0]	rs1	110	rd	0010011	I ori
imm[11:0]	rs1	111	rd	0010011	I andi
0000000 shamt	rs1	001	rd	0010011	I slli
0000000 shamt	rs1	101	rd	0010011	I srli
0100000 shamt	rs1	101	rd	0010011	I srai
0000000 rs2	rs1	000	rd	0110011	R add
0100000 rs2	rs1	000	rd	0110011	R sub
0000000 rs2	rs1	001	rd	0110011	R sll
0000000 rs2	rs1	010	rd	0110011	R slt
0000000 rs2	rs1	011	rd	0110011	Rsltu
0000000 rs2	rs1	100	rd	0110011	R xor
0000000 rs2	rs1	101	rd	0110011	R srl
0100000 rs2	rs1	101	rd	0110011	R sra
0000000 rs2	rs1	110	rd	0110011	R or
0000000 rs2	rs1	111	rd	0110011	R and

目前 CPU 可以執行上方表格中 37 種指令和特殊設計中的浮點數加減法、乘法和除法。

3、 系統驗證方法與結果分析

(1) CPU 測試資料：這次利用三種測試來測驗我們的 CPU

第一種：將所有用到的指令分別測試以確認每種指令皆可執行

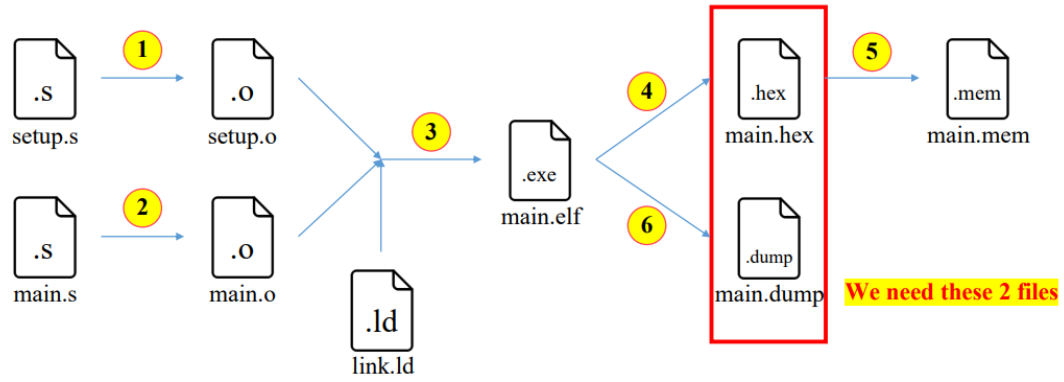
第二種：執行 merge sort 分別測試三種(32、15、27 筆測資)，將測資由小至大排列。

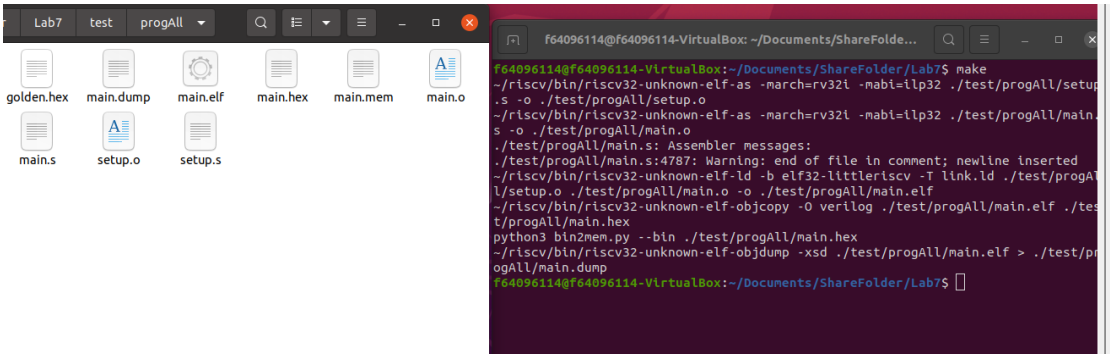
第三種：計算費氏數列中第五、第十和第十二項數分別為何

(撰寫的 code 內容可參考.test/progAll)

(2) Machine code (IR)生成過程

先專寫 RISC-V 組合語言(基本測試+merge sort+費氏數列)，將寫好的 main.s 與要設定的環境(如 main、result 位置等) setup.s 放入 linux 環境利用 assembler+linker 把他轉為執行檔 main.exe，最後反組譯回我們所要的 IR main.hex 檔及可方便 debug 的 main.dump 檔以及記憶體印象檔。

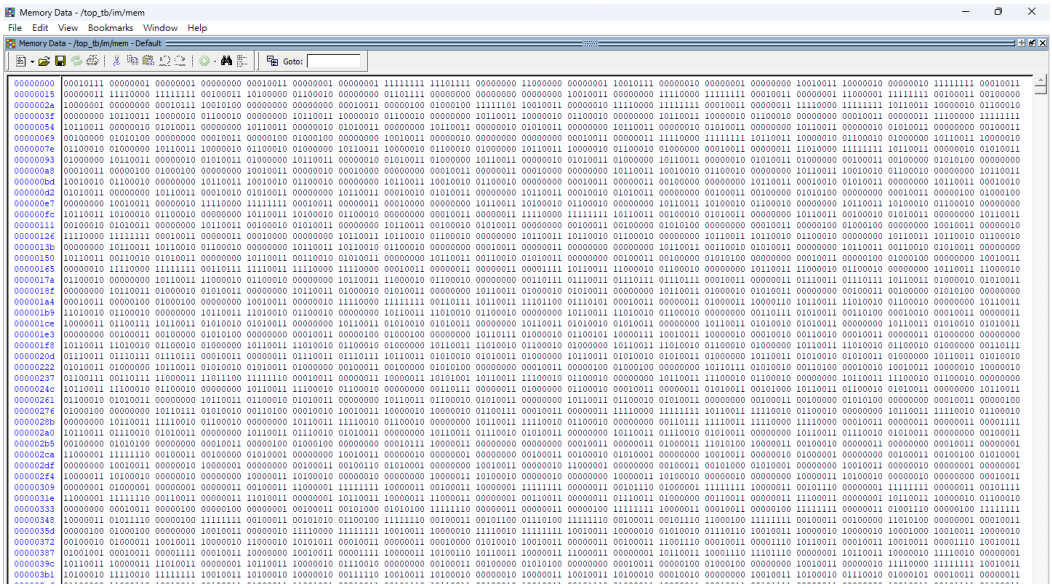




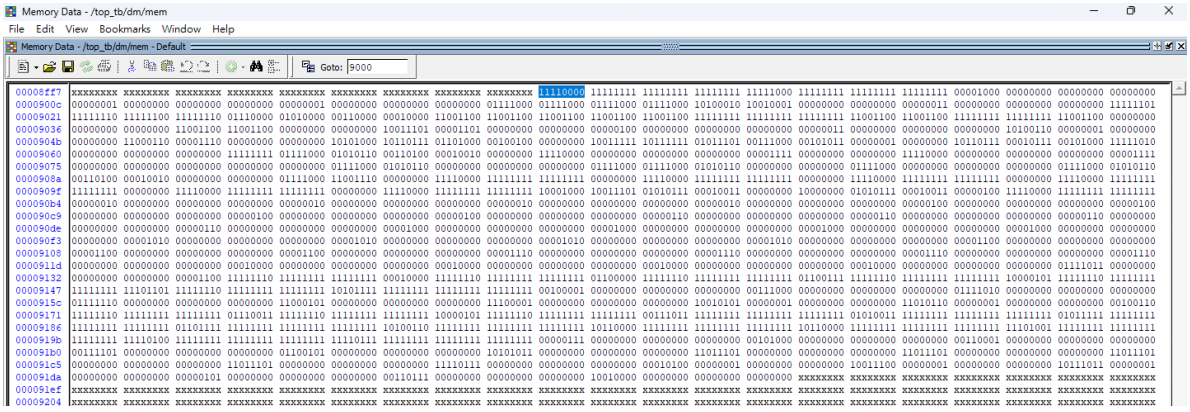
(3) 驗證方法

我們將 instruction read 進 instruction memory，並開始執行 CPU，實行過程中會將算好的答案寫入 data memory，待所有運算結束後比對 Golden data(正確答案)和我們寫入的答案，並標記其在 memory 中的位置、寫入的值、是否通過等

- Insert Instruction to instruction memory

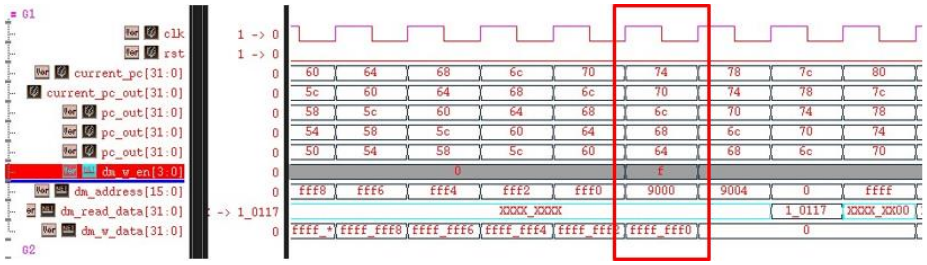


- Use the result in data memory (0x9000) to compare with golden file



4、 結果分析

(1) “單一”指令正確性



dm_w_en=f，表示存入 1 word(ffff_fff0 == -16)進入 address 9000

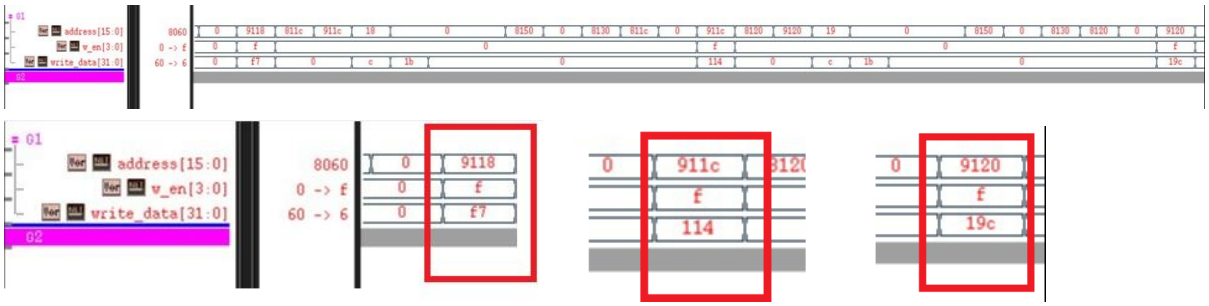
```
add:
  li t0, 0xffffffff # -1
  li t1, 0xffffffff # -1
  add t0, t0, t1    # t0 = -2
  add t0, t0, t1    # t0 = -3
  add t0, t0, t1    # t0 = -4
  add t0, t0, t1    # t0 = -5
  add t0, t0, t1    # t0 = -6
  li t1, 0xfffffffe # -2
  add t0, t1, t0    # t0 = -8
  add t0, t1, t0    # t0 = -10
  add t0, t1, t0    # t0 = -12
  add t0, t1, t0    # t0 = -14
  add t0, t1, t0    # t0 = -16
  sw t0, 0(s0)
  addi s0, s0, 4
```

```
DM['h9000] = fffffff0, pass
DM['h9004] = fffffff8, pass
DM['h9008] = 00000008, pass
DM['h900c] = 00000001, pass
DM['h9010] = 00000001, pass
DM['h9014] = 78787878, pass
DM['h9018] = 000091a2, pass
DM['h901c] = 00000003, pass
DM['h9020] = fefcfefd, pass
DM['h9024] = 10305070, pass
DM['h9028] = cccccccc, pass
DM['h902c] = ffffffcc, pass
```

以 add 為例，運算出答案為 fffffff0(-16)後輸入 data memory 地址為 9000 的位置，並比對 golden data 後確定為正解

(2) 程式正確性

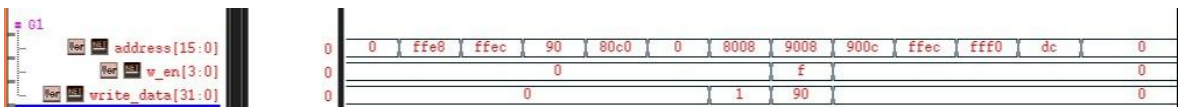
i. 排序(需自行撰寫組合語言)




```
DM['h9114] = 000000dd, pass
DM['h9118] = 000000f7, pass
DM['h911c] = 00000114, pass
DM['h9120] = 0000019c, pass
DM['h9124] = 000001bb, pass
```

從 nWave 中可見得從上到下分別儲存了 f7(247)至地址 9118、114(276)至地址 911c、19c(412)至地址 9120，數字從小往大排列符合設計理念，且和 golden data 相符

ii. 費氏數列(需自行撰寫組合語言)

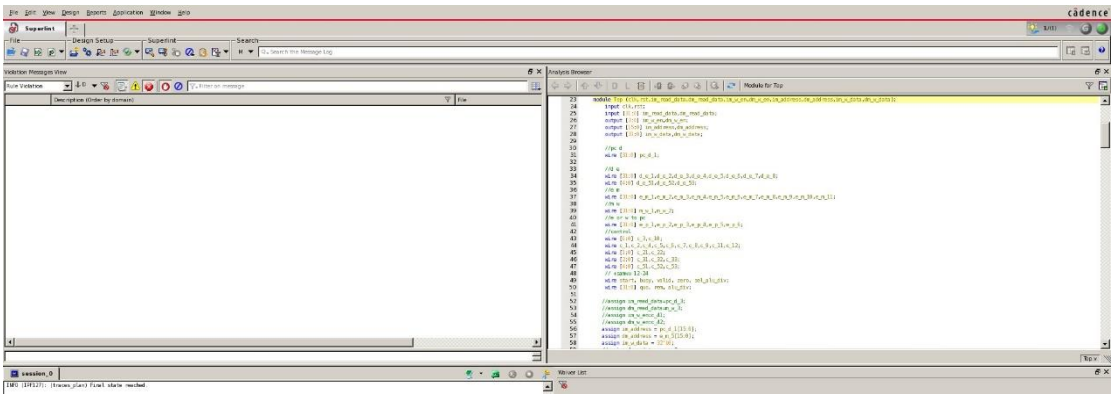



```
DM['h9000] = 00000005, pass
DM['h9004] = 00000037, pass
DM['h9008] = 00000090, pass
```

將費氏數列中第十二個數 90(144)存入 9008，符合設計理念，且和 golden data 相同

5、 SuperLint 與 ICC 檢查結果

(1) SuperLint 檢查結果



主要有三大類問題:

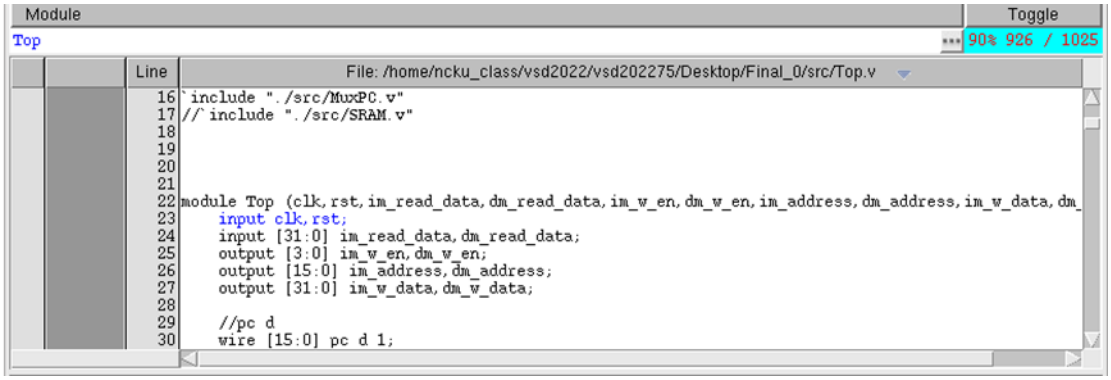
i. Coding style 不佳: 利用講義上注意的事項一一解決，大部分都是 else、default 等問題。

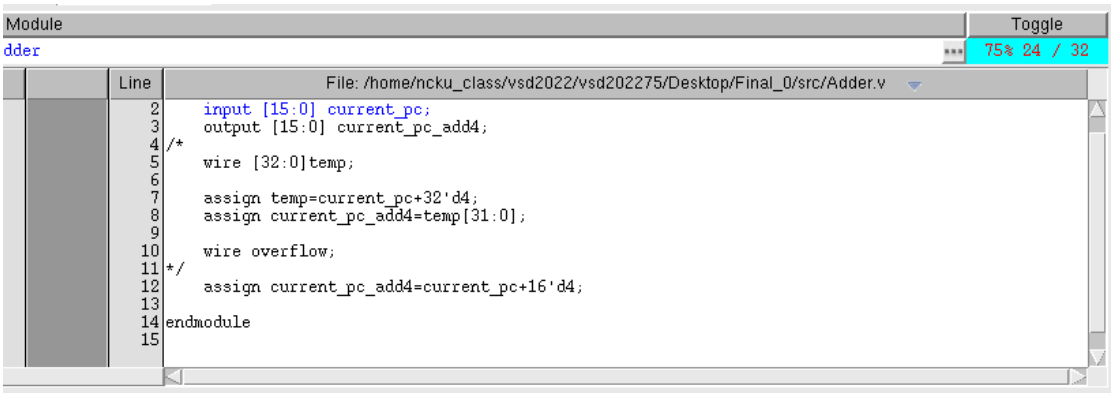
ii. Overflow 問題 : 主要是有些加減法運算只要存放的結果沒有比原有的位元長度還多 1bits，系統就會假設會有 overflow 問題發生並提醒要更改。

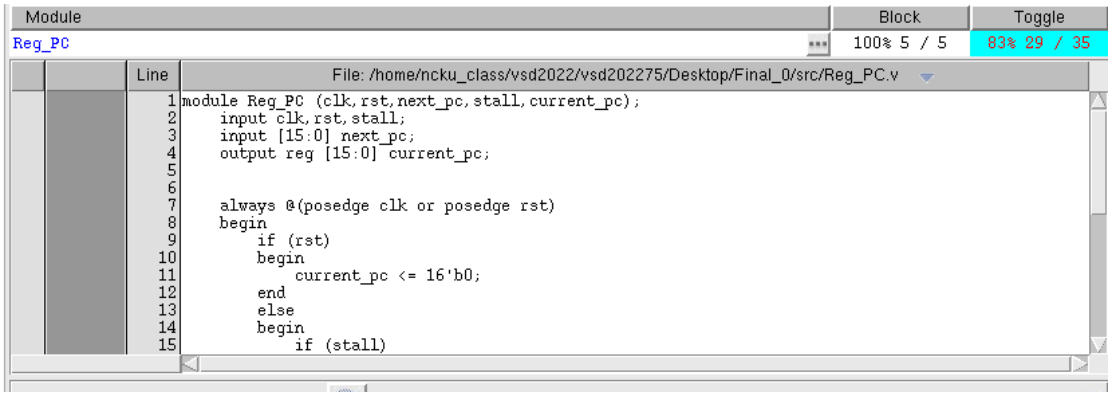
iii. 有些判斷式內的判斷條件根本不會發生，必須修改掉。

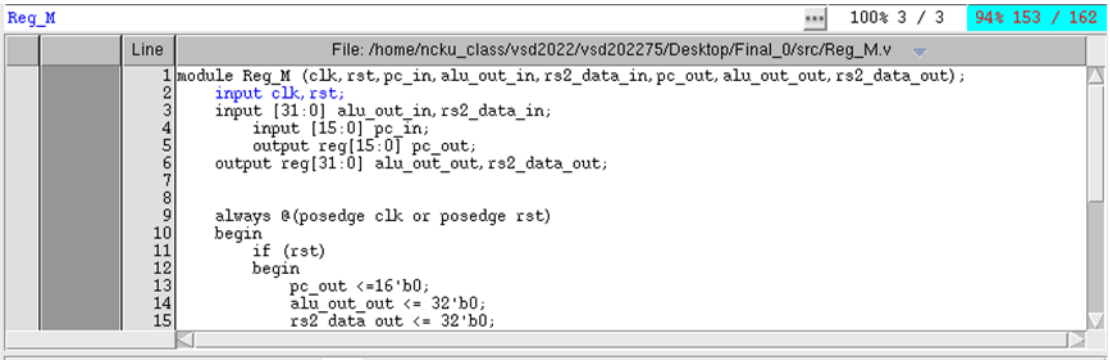
(2) ICC 檢查結果

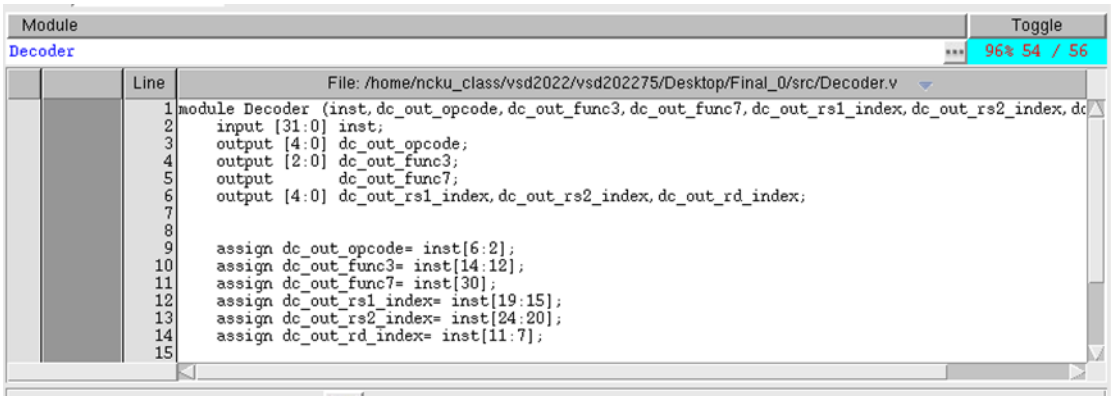
為了有效提升 ICC 檢查結果，除了盡可能增加各種類型的指令使 cpu 每個功能都能確實執行到之外，我們發現也可以透過簡化硬體的方法來達成，在我們所使用的 ISA 架構下，cpu 的 address 傳遞的資料實際上最大就是 16bits，而當初方便起見我們傳輸線都設定為 32bits，且相關硬體內容也是 32bits 的大小，也就是說其實有很多地方無法 toggle 到，因此進行硬體簡化，將多餘 32bits 的部分簡化成 16bits，而 ICC 的結果也有顯著提升，特別是 adder_pc 的部分，由原本 40 多%提升到 75%，將近兩倍之多。


Module
Toggle
Top
90% 926 / 1025
Line
File: /home/ncku_class/vsd2022/vsd202275/Desktop/Final_0/src/Top.v
16 `include "./src/MuxPC.v"
17 // include "./src/SRAM.v"
18
19
20
21
22 module Top (clk, rst, im_read_data, dm_read_data, im_w_en, dm_w_en, im_address, dm_address, im_w_data, dm_
23 input clk, rst;
24 input [31:0] im_read_data, dm_read_data;
25 output [3:0] im_w_en, dm_w_en;
26 output [15:0] im_address, dm_address;
27 output [31:0] im_w_data, dm_w_data;
28
29 //pc d
30 wire [15:0] pc d 1;


Module
Toggle
dder
75% 24 / 32
Line
File: /home/ncku_class/vsd2022/vsd202275/Desktop/Final_0/src/Adder.v
2 input [15:0] current_pc;
3 output [15:0] current_pc_add4;
4 /*
5 wire [32:0]temp;
6
7 assign temp=current_pc+32'd4;
8 assign current_pc_add4=temp[31:0];
9
10 wire overflow;
11 */
12 assign current_pc_add4=current_pc+16'd4;
13
14 endmodule
15


Module
Block
Toggle
Reg_PC
100% 5 / 5
83% 29 / 35
Line
File: /home/ncku_class/vsd2022/vsd202275/Desktop/Final_0/src/Reg_PC.v
1 module Reg_PC (clk, rst, next_pc, stall, current_pc);
2 input clk, rst, stall;
3 input [15:0] next_pc;
4 output reg [15:0] current_pc;
5
6
7 always @(posedge clk or posedge rst)
8 begin
9 if (rst)
10 begin
11 current_pc <= 16'b0;
12 end
13 else
14 begin
15 if (stall)


Reg_M
100% 3 / 3
94% 153 / 162
Line
File: /home/ncku_class/vsd2022/vsd202275/Desktop/Final_0/src/Reg_M.v
1 module Reg_M (clk, rst, pc_in, alu_out_in, rs2_data_in, pc_out, alu_out_out, rs2_data_out);
2 input clk, rst;
3 input [31:0] alu_out_in, rs2_data_in;
4 input [15:0] pc_in;
5 output reg[15:0] pc_out;
6 output reg[31:0] alu_out_out, rs2_data_out;
7
8
9 always @(posedge clk or posedge rst)
10 begin
11 if (rst)
12 begin
13 pc_out <=16'b0;
14 alu_out_out <= 32'b0;
15 rs2_data_out <= 32'b0;

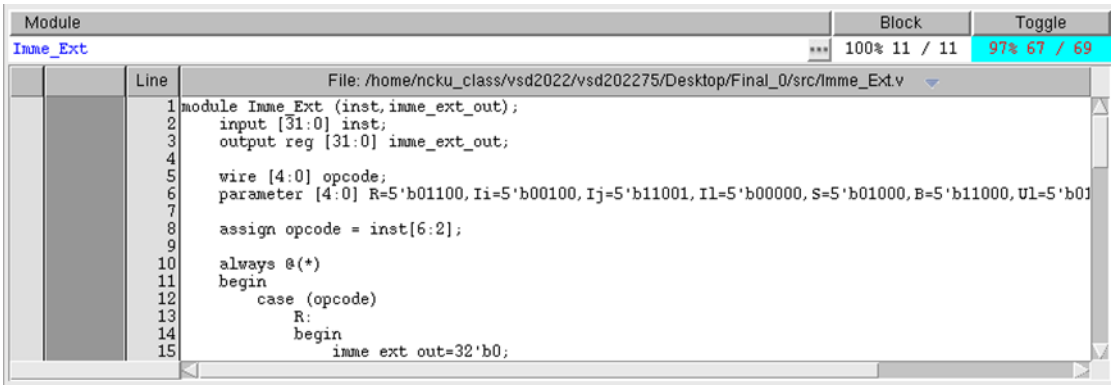


Module | Toggle

Decoder | 96% 54 / 56

Line | File: /home/ncku_class/vsd2022/vsd202275/Desktop/Final_0/src/Decoder.v

```
1  module Decoder (inst, dc_out_opcode, dc_out_func3, dc_out_func7, dc_out_rs1_index, dc_out_rs2_index, dc
2      input [31:0] inst;
3      output [4:0] dc_out_opcode;
4      output [2:0] dc_out_func3;
5      output       dc_out_func7;
6      output [4:0] dc_out_rs1_index, dc_out_rs2_index, dc_out_rd_index;
7
8
9      assign dc_out_opcode= inst[6:2];
10     assign dc_out_func3= inst[14:12];
11     assign dc_out_func7= inst[30];
12     assign dc_out_rs1_index= inst[19:15];
13     assign dc_out_rs2_index= inst[24:20];
14     assign dc_out_rd_index= inst[11:7];
15
```

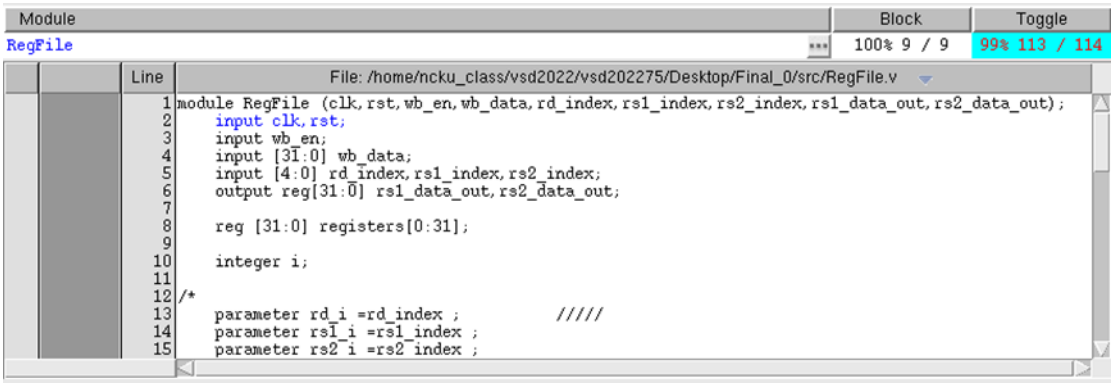


Module | Block | Toggle

Imme_Ext | 100% 11 / 11 | 97% 67 / 69

Line | File: /home/ncku_class/vsd2022/vsd202275/Desktop/Final_0/src/Imme_Ext.v

```
1  module Imme_Ext (inst,imme_ext_out);
2      input [31:0] inst;
3      output reg [31:0] imme_ext_out;
4
5      wire [4:0] opcode;
6      parameter [4:0] R=5'b01100, Ii=5'b00100, Ij=5'b11001, Il=5'b00000, S=5'b01000, B=5'b11000, Ul=5'b01
7
8      assign opcode = inst[6:2];
9
10     always @(*)
11     begin
12         case (opcode)
13             R:
14             begin
15                 imme ext out=32'b0;
```

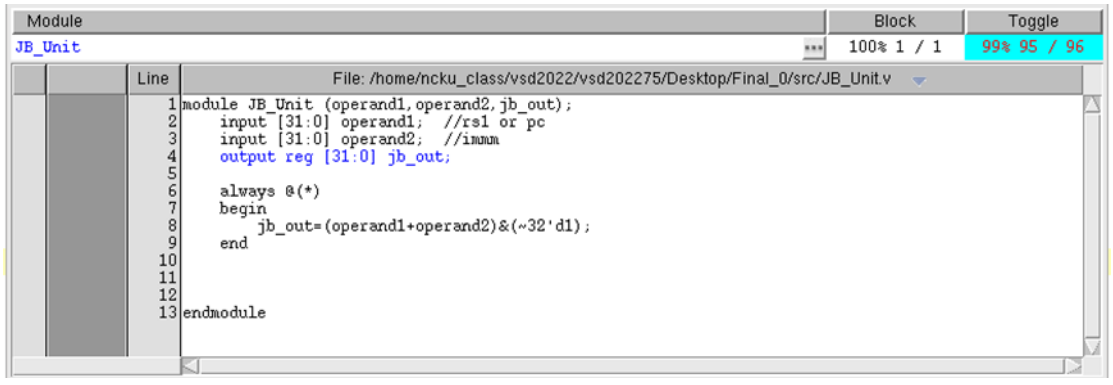


Module | Block | Toggle

RegFile | 100% 9 / 9 | 99% 113 / 114

Line | File: /home/ncku_class/vsd2022/vsd202275/Desktop/Final_0/src/RegFile.v

```
1  module RegFile (clk, rst, wb_en, wb_data, rd_index, rs1_index, rs2_index, rs1_data_out, rs2_data_out);
2      input clk, rst;
3      input wb_en;
4      input [31:0] wb_data;
5      input [4:0] rd_index, rs1_index, rs2_index;
6      output reg[31:0] rs1_data_out, rs2_data_out;
7
8      reg [31:0] registers[0:31];
9
10     integer i;
11
12 /*
13     parameter rd_i =rd_index ;          /////
14     parameter rs1_i =rs1_index ;
15     parameter rs2_i =rs2_index ;
```

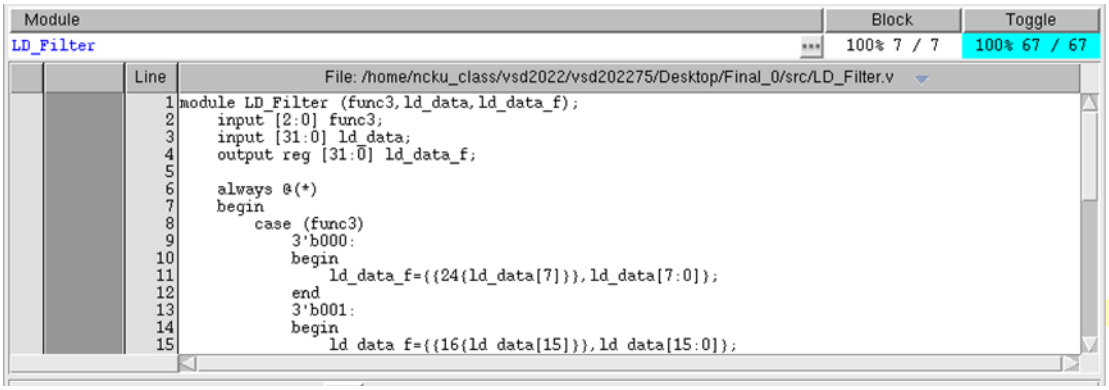


Module | Block | Toggle

JB_Unit | 100% 1 / 1 | 99% 95 / 96

Line | File: /home/ncku_class/vsd2022/vsd202275/Desktop/Final_0/src/JB_Unit.v

```
1  module JB_Unit (operand1,operand2,jb_out);
2      input [31:0] operand1;  //rs1 or pc
3      input [31:0] operand2;  //imm
4      output reg [31:0] jb_out;
5
6      always @(*)
7      begin
8          jb_out=(operand1+operand2)&(~32'd1);
9      end
10
11
12
13 endmodule
```



6、 合成結果

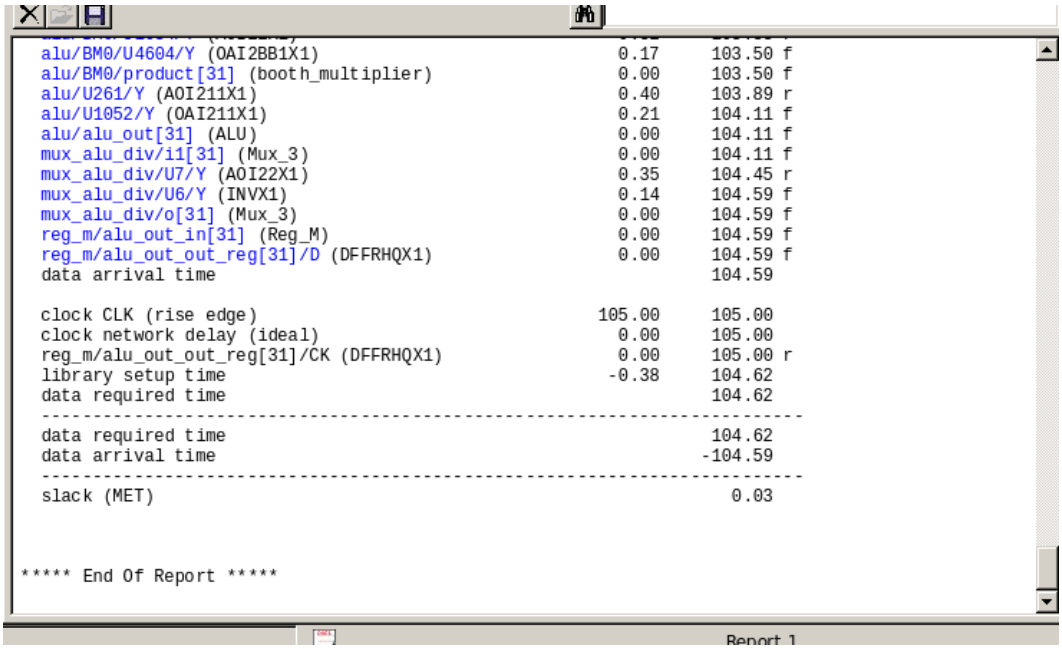
(1) 速度(需附上截圖，Setup time 和 Hold time slack 都>0):

i. Clk period=70ns

(在 RC wire load 沒有去除的情況下，若去掉 constraint period 可到 44ns)

由於 CPU 內部支援快速乘法器導致 critical path 變很長，所以加上特殊硬體後 clk period 最快只能到 70ns。

ii. Setup time slack



iii. Hold time slack

```
  reg_w/ld_data_in[0] (Reg_W)                              0.00      35.24 f
  reg_w/ld_data_out_reg[0]/D (DFFRHQX1)                    0.00      35.24 f
  data arrival time                                                  35.24

  clock CLK (rise edge)                                   35.00      35.00
  clock network delay (ideal)                              0.00      35.00
  reg_w/ld_data_out_reg[0]/CK (DFFRHQX1)                   0.00      35.00 r
  library hold time                                       -0.03      34.97
  data required time                                                 34.97
  --------------------------------------------------------------------------
  data required time                                                 34.97
  data arrival time                                                 -35.24
  --------------------------------------------------------------------------
  slack (MET)                                                         0.27


design_vision>
```

iv. Pre-simulation(synthesized) pass

```
$finish called from file "top_tb.v", line 139.
$finish at simulation time           1934065000
           V C S   S i m u l a t i o n   R e p o r t
Time: 1934065000 ps
CPU Time:    437.340 seconds;       Data structure size:   7.2Mb
Sun Jan  8 22:34:11 2023
CPU time: 9.554 seconds to compile + 7.866 seconds to elab + 11.878 seconds to link + 437.377 seconds in si
[vsd202271@cic-svr ~/final]$
```

v. No latch in the design!

```
        in routine Reg_M line 6 in file
                './src/Reg_M.v'.
=================================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=================================================================================
|   alu_out_out_reg   | Flip-flop |  32   |  Y  | N  | Y  | N  | N  | N  | N  |
|  rs2_data_out_reg   | Flip-flop |  32   |  Y  | N  | Y  | N  | N  | N  | N  |
|     pc_out_reg      | Flip-flop |  32   |  Y  | N  | Y  | N  | N  | N  | N  |
=================================================================================

Inferred memory devices in process
        in routine Reg_PC line 7 in file
                './src/Reg_PC.v'.
=================================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=================================================================================
|   current_pc_reg    | Flip-flop |  32   |  Y  | N  | Y  | N  | N  | N  | N  |
=================================================================================

Inferred memory devices in process
        in routine Reg_W line 7 in file
                './src/Reg_W.v'.
=================================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=================================================================================
|   alu_out_out_reg   | Flip-flop |  32   |  Y  | N  | Y  | N  | N  | N  | N  |
|   ld_data_out_reg   | Flip-flop |  32   |  Y  | N  | Y  | N  | N  | N  | N  |
=================================================================================

Inferred memory devices in process
        in routine RegFile line 18 in file
                './src/RegFile.v'.
=================================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
=================================================================================
|    registers_reg    | Flip-flop | 1024  |  Y  | N  | N  | N  | N  | N  | N  |
=================================================================================
Statistics for MUX_OPs
=================================================
Log  History
design_vision>
```

(2) 面積(需附上截圖)

```
Number of nets:                         34986
Number of cells:                        18093
Number of combinational cells:          16241
Number of sequential cells:              1645
Number of macros/black boxes:               0
Number of buf/inv:                       2346
Number of references:                      25

Combinational area:              400096.067348
Buf/Inv area:                     15693.955400
Noncombinational area:            96349.177177
Macro/Black Box area:                 0.000000
Net Interconnect area:          2443758.884521

Total cell area:                 496445.244525
Total area:                     2940204.129047
design_vision>
```

Total cell area: 496445um^2

(3) 功耗(需附上截圖)

```
Global Operating Voltage = 1.62
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000pf
    Time Units = 1ns
    Dynamic Power Units = 1mW    (derived from V,C,T units)
    Leakage Power Units = 1pW


  Cell Internal Power  = 970.2130 uW   (84%)
  Net Switching Power  = 184.1920 uW   (16%)
                         ---------
Total Dynamic Power    =   1.1544 mW  (100%)

Cell Leakage Power     =  20.1878 uW


                 Internal         Switching           Leakage            Total
Power Group      Power            Power               Power              Power   (   %    )  Attrs
--------------------------------------------------------------------------------------------------
io_pad             0.0000           0.0000              0.0000           0.0000  (   0.00%)
memory             0.0000           0.0000              0.0000           0.0000  (   0.00%)
black_box          0.0000           0.0000              0.0000           0.0000  (   0.00%)
clock_network      0.0000           0.0000              0.0000           0.0000  (   0.00%)
register           0.8734        1.1360e-02          3.2245e+06           0.8879  (  75.60%)
sequential     2.6378e-05        2.6092e-07          6.7374e+03       3.3376e-05  (   0.00%)
combinational  9.6832e-02           0.1728           1.6957e+07           0.2866  (  24.40%)
--------------------------------------------------------------------------------------------------
Total              0.9702 mW        0.1842 mW       2.0188e+07 pW        1.1746 mW

***** End Of Report *****
```

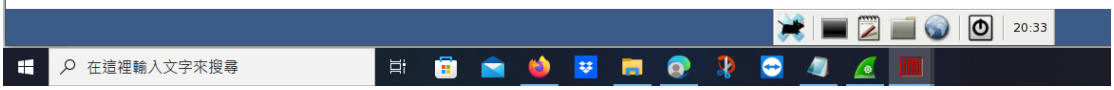
total static power : 20.1878 uW

total dynamic power : 1.1544 mW

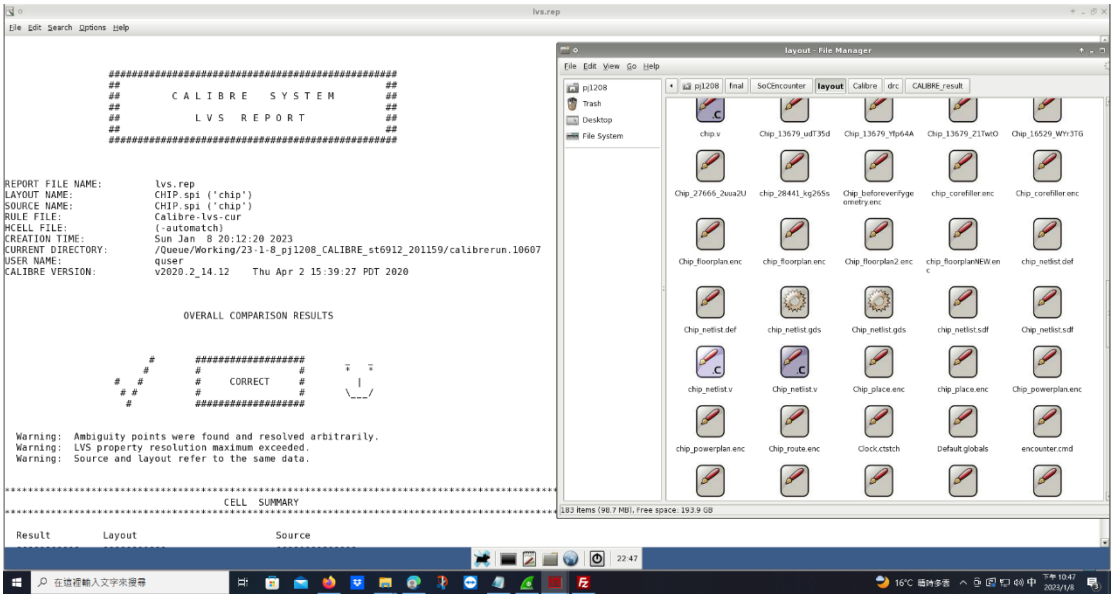
7、 Layout 結果

(1) DRC

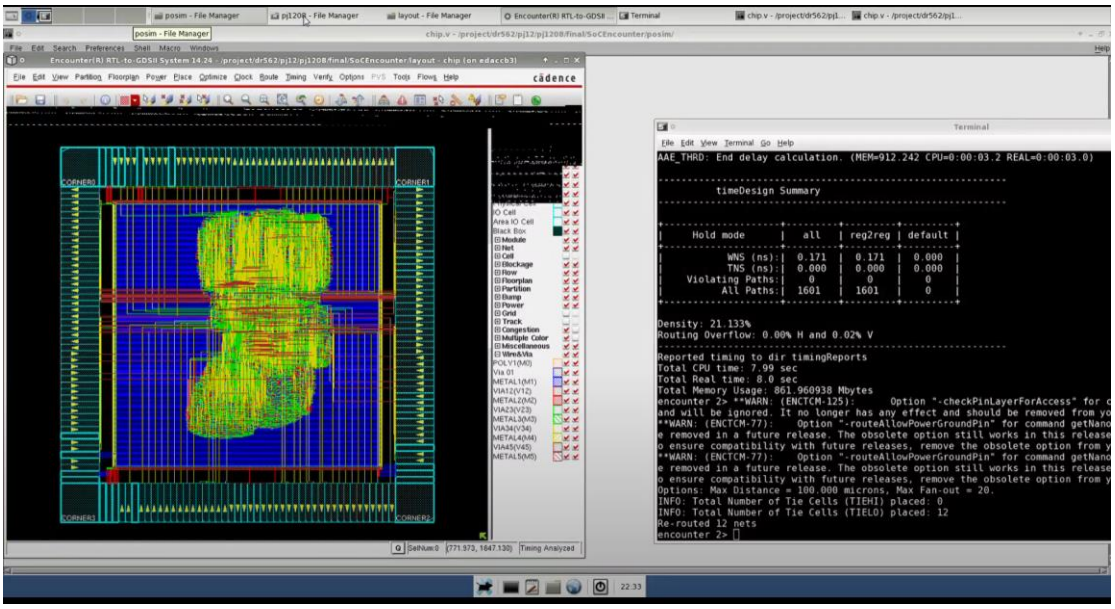
```
RULECHECK CTM.A.1 ..................... TOTAL Result Count = 0     (0)
RULECHECK CTM.R.5 ..................... TOTAL Result Count = 0     (0)
RULECHECK MIM_M5.W.1 .................. TOTAL Result Count = 0     (0)
RULECHECK MIM_M5.S.1 .................. TOTAL Result Count = 0     (0)
RULECHECK MIM_M5.S.2 .................. TOTAL Result Count = 0     (0)
RULECHECK MIM_M5.E.3 .................. TOTAL Result Count = 0     (0)
RULECHECK MIMVIA.S.1 .................. TOTAL Result Count = 0     (0)
RULECHECK MIMVIA.S.2 .................. TOTAL Result Count = 0     (0)
RULECHECK MIMVIA.E.1 .................. TOTAL Result Count = 0     (0)
RULECHECK MIMVIA.E.2 .................. TOTAL Result Count = 0     (0)
RULECHECK MIMVIA.C.1 .................. TOTAL Result Count = 0     (0)
RULECHECK MIMVIA.R.1 .................. TOTAL Result Count = 0     (0)
RULECHECK MIMVIA.R.3 .................. TOTAL Result Count = 0     (0)
RULECHECK SBD.W.1_SBD.W.1.1 ........... TOTAL Result Count = 0     (0)
RULECHECK SBD.W.2_SBD.W.2.1 ........... TOTAL Result Count = 0     (0)
RULECHECK SBD.W.3 ..................... TOTAL Result Count = 0     (0)
RULECHECK SBD.W.4 ..................... TOTAL Result Count = 0     (0)
RULECHECK SBD.S.1 ..................... TOTAL Result Count = 0     (0)
RULECHECK SBD.E.1 ..................... TOTAL Result Count = 0     (0)
RULECHECK SBD.E.2 ..................... TOTAL Result Count = 0     (0)
RULECHECK SBD.O.1 ..................... TOTAL Result Count = 0     (0)
RULECHECK SBD.E.1.1 ................... TOTAL Result Count = 0     (0)
RULECHECK SBD.E.3 ..................... TOTAL Result Count = 0     (0)
RULECHECK SBD.R.2 ..................... TOTAL Result Count = 0     (0)
RULECHECK SBD.R.3 ..................... TOTAL Result Count = 0     (0)
RULECHECK SBD.R.4 ..................... TOTAL Result Count = 0     (0)
RULECHECK SBD.R.5 ..................... TOTAL Result Count = 0     (0)
--------------------------------------------------------------------------------
--- SUMMARY
---
TOTAL CPU Time:                  9
TOTAL REAL Time:                 9
TOTAL Original Layer Geometries: 74329 (831039)
TOTAL DRC RuleChecks Executed:   366
TOTAL DRC Results Generated:     97528 (207356)
```

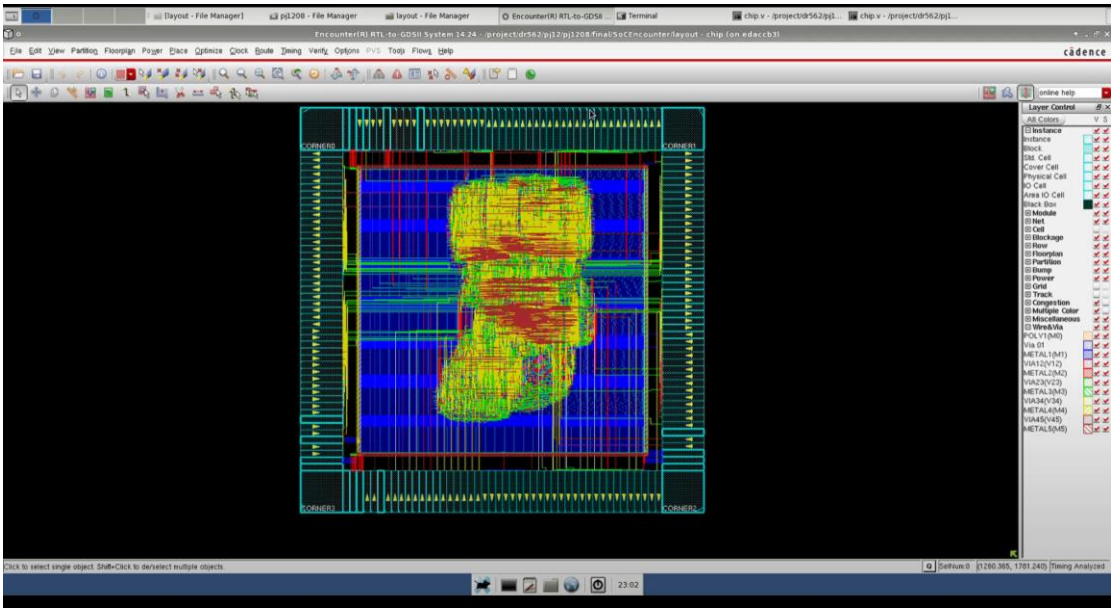


(2) LVS



(3) Layout





詳細過程可參考影片:
https://drive.google.com/drive/folders/1KQXQewABxRKX_CL1gAn0fu6nHqsbmqWL?usp=share_link

8、 管線化

(1) 設計與劃分

採用典型的 RISC CPU pipeline 架構，共有五 stage 的管線劃分，其中依序由 D_reg,E_reg,M_reg,W_reg 來區隔。

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock cycle	1	2	3	4	5	6	7

i. IF stage 讀取指令:

instruction memory 根據 pc register 提供的地址，將正確的指令內容傳遞到 D_reg，準備下一級輸出。

ii. ID stage 指令解碼:

Decoder 從 D_reg 所得到的指令內容進行解碼，並使 register file 預備好所需的傳遞內容，最後傳入 E_reg，準備下一級輸出。

iii. EX stage 執行:

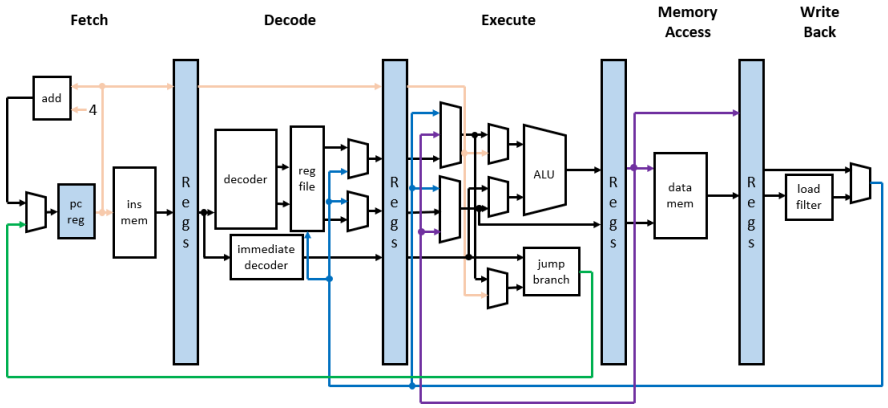
Alu 接收來自 E_reg 或是 MEM 層的內容並執行運算，將結果傳入 M_reg，準備下一級輸出，branch,juump 指令亦會在此層級執行跳轉。

iv. MEM stage 記憶體存取:

將上一級 M_register 傳遞過來的內容存入 data memory，或是從中取出，並將可能被用到的內容往下一級 W_reg 傳入。

v. WB stage 寫回暫存器:

將 W_reg 的傳遞內容寫入 register file 或是 data memory。

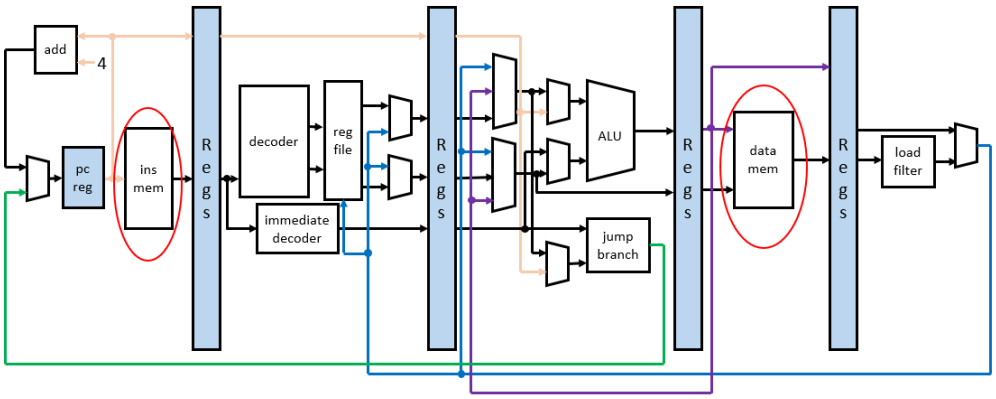


(2) 管線化的危障

此 pipeline 模式將會出現 3 種 hazard 分別為 structure hazard ,control hazard ,data hazard

i. structure hazard

源自於硬體資源不足，當 cpu 要在同一 cycle 內 fetch 和 load 資料時會產生衝突，因此將 memory 切成 IM 與 DM 兩塊，使兩者在不同的 cycle 執行。



ii. control hazard

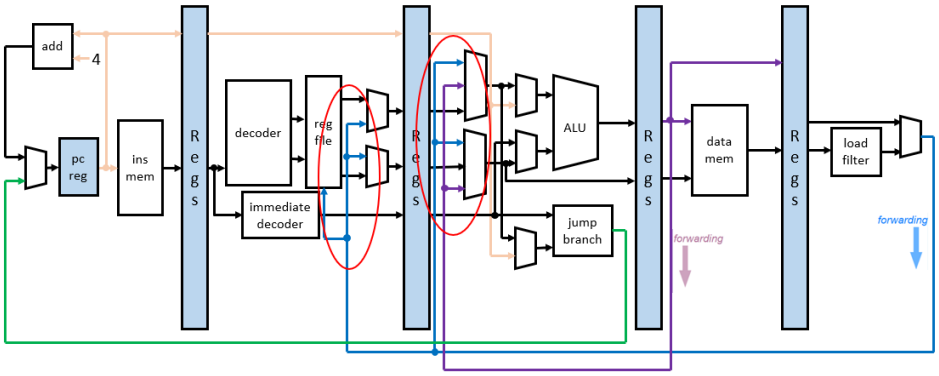
當執行到 branch 或 jal 等跳轉指令時，必須等到 EX stage 才能得知是否要跳轉，若要跳轉則須把前兩條指令改為 nop，controller 的訊號也要改成 nop，而 D_reg ,E_reg 將 output 改為 nop 訊號，在此之後 pc 便會指到對的指令位址，因此每次跳轉時會產生 stall 而拖慢 cpu 執行速度，為此我們有設計 branch prediction 因應此情形。

iii. data hazard

當指令用到前一條指令的 write register data 時，會發生 data 來不及更新的情況，而發生錯誤，可分為下列兩種。

a. (R-type/Ii-type/IL-type/S-type/B-type/Ij-type) +(R-type/Ii-type/IL-type/S-type/B-type/Ij-type):

將 WB stage 中最新的 data forward 到 EX/ID stage，MEM 中最新的 data forward 到 EX stage，其中透過 mux 來控制是否要選擇使用 forwarding 的 data。

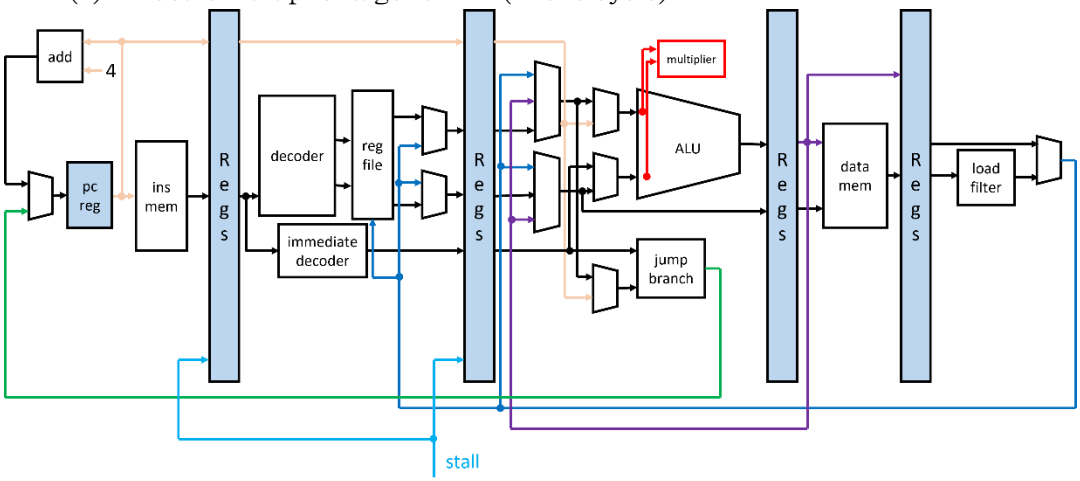


b. (R-type/Ii-type/IL-type/S-type/B-type/Ij-type) +(IL-type/Ul-type/Ua-type):

load use hazard 必須等到 data memory 拿到資料後才能 forward，所以只能從 WB stage forward 到 ID stage，而其中將加入一個 stall 等待從 memory 拿取資料的時間，並將 E_reg flush 掉。

9、 特殊設計(例如 cache、支援浮點數運算等額外設計都可提出)

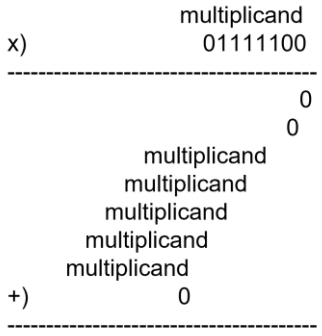
(1) Booths multiplier algorithm：(in one cycle)

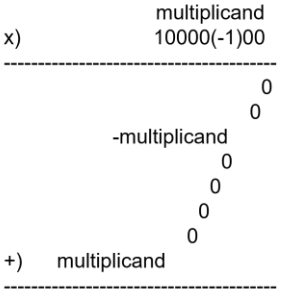


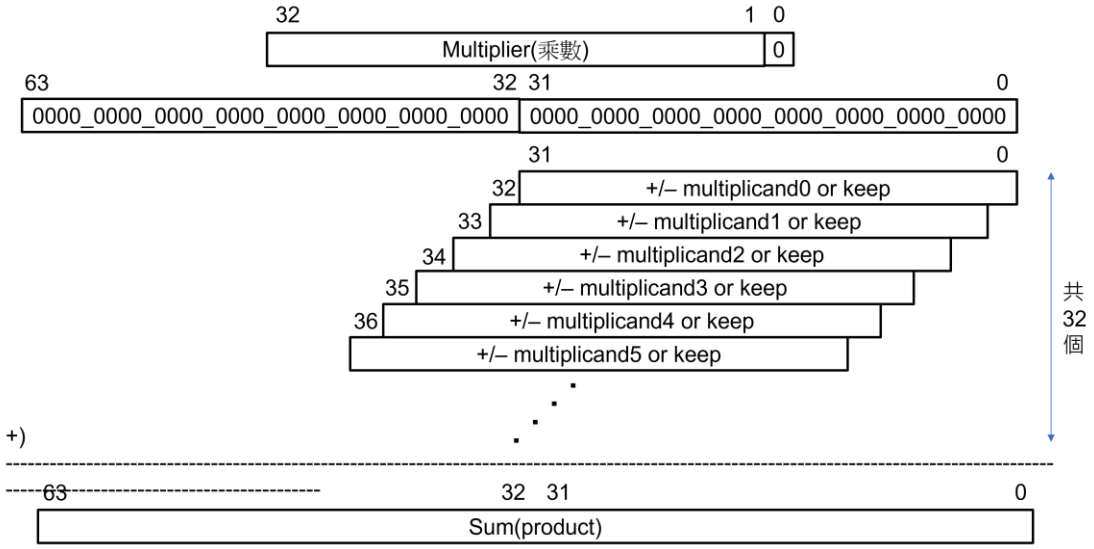
i. Extend 32-bit multiplier to 33-bit multiplier' = {32'd{multiplier},1'd0}

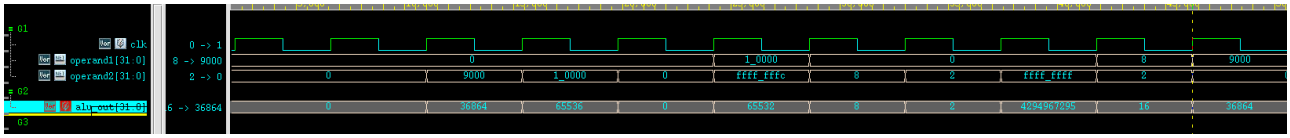
ii. Check multiplier'[1:0], multiplier' [2:1], multiplier' [3:2], … , multiplier' [32:31] at the same time. If equal to 01/10, add/sub shifted multiplicand By the way, the number multiplicand shifting depends on the number of multiplier's bit checked.

iii. Sum of these 32 add/sub shifted multiplicands is the product_result



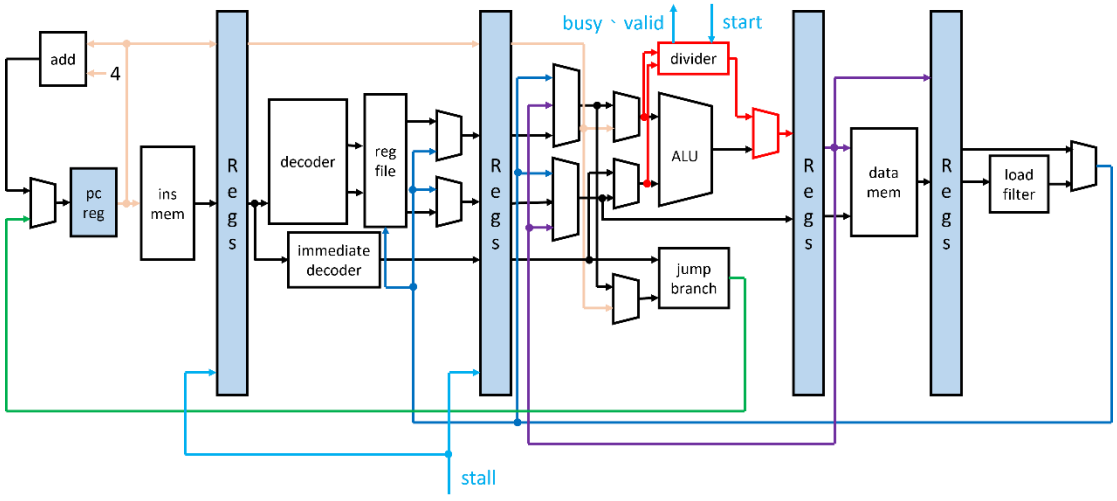






可看到範例為 8*2=16。當 operand1=8 及 operand2=2 時，經過 1 個 cycle 後乘法器會算出正確的結果為 16。

(2) divider algorithm :



1st cycle :

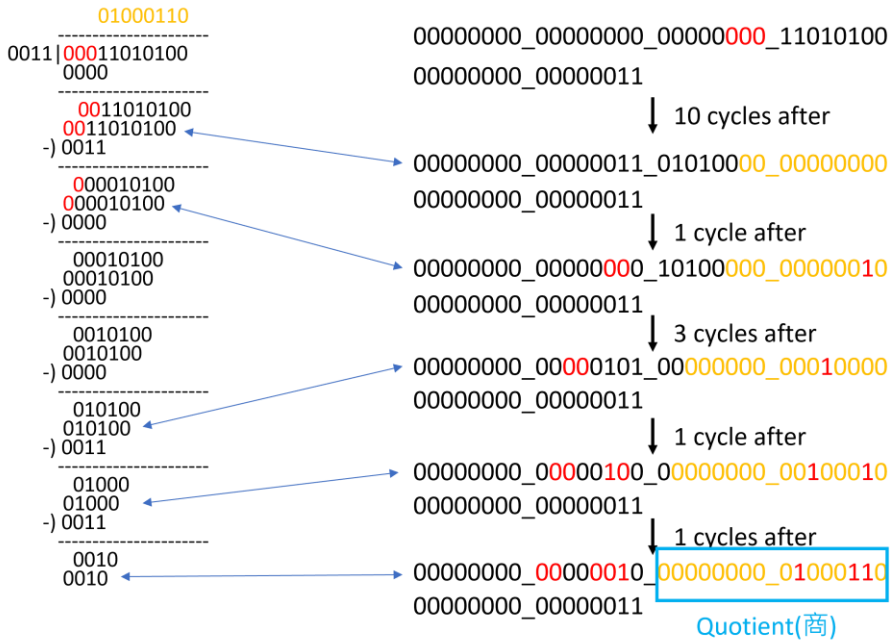
1. Prepare 2 operands
   1 operand : dividend_expand = {31'd0, 32'd{dividend}, 1'd0}
   1 operand : divisor
2. Give the start signal (inform the subtractor is ready)

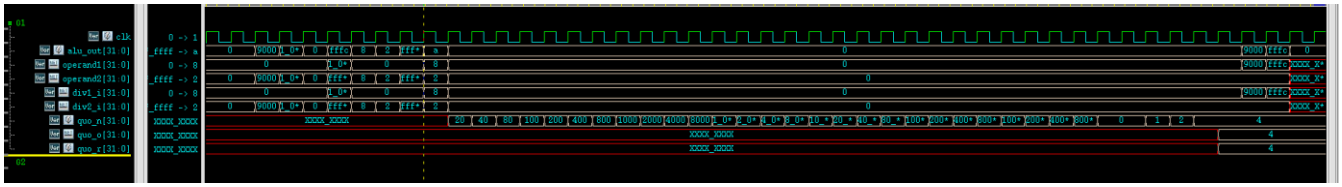
2nd ~ 32th cycle :

1. Give the busy signal : show the subtractor is working
2. Comparison : dividend_expand[63:32] : divisor
   a. if (dividend[63:32] > divisor) dividend_expand[0] = 1;
   b. else dividend_expand[0] = 0;
3. Left rotate 1 bit

32th cycle (Final result) :

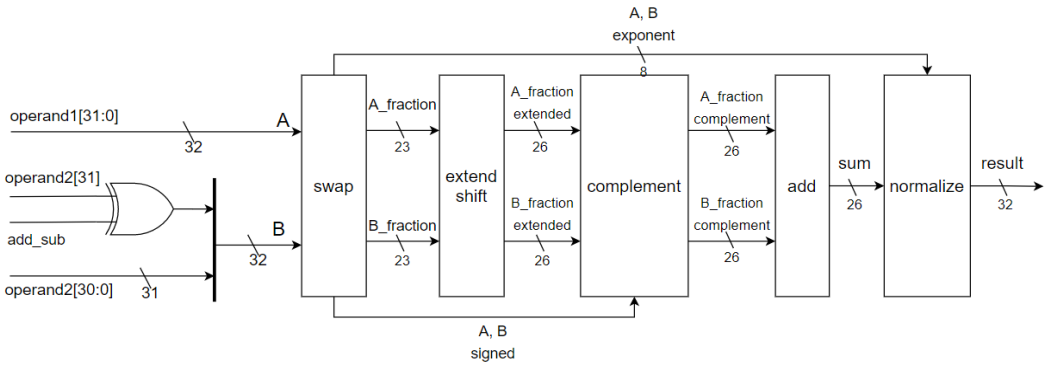
1. remainder : dividend_expand[63:32]
2. quotient : dividend_expand[31:0]
3. Give the valid signal to show that the "divide" operation finish.



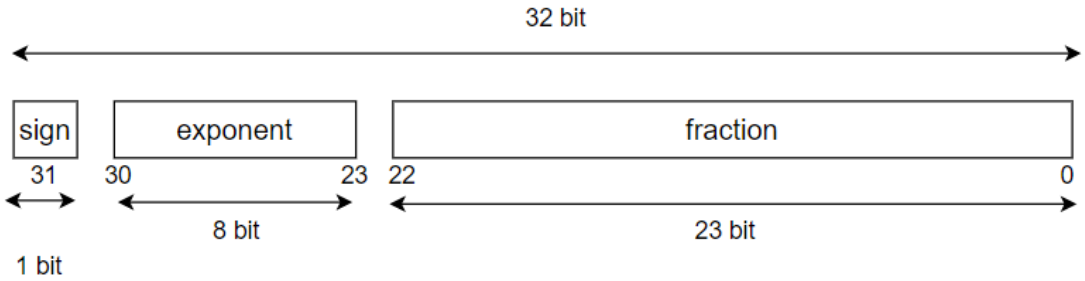


可看到範例為 8/2=4。當 operand1=8 及 operand2=2 時，經過 32 個 cycle 後除法器會算出正確的結果為 4。

(3) Float point substractor, adder:

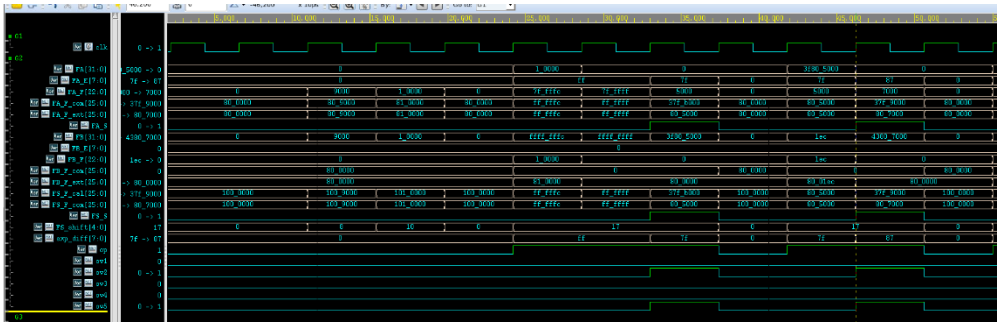


上圖是 float point 加減法運算的流程圖，input 為兩個 float point 格式的數值:operand1, operand2，以及一個控制加減法的 add_sub(0 為加, 1 為減)，因為浮點數形式的數值第 31 位是 signed bit，可以很容易的和加減法互相消除，所以首先第一個步驟要先將加減法一律更改為加法，所以將 add_sub 和 operand2 的 signed bit 做 xor，將得到的數值做為 operand2 的 signed bit，接下來的步驟是要將 operand1, operand2 比較 exponent 的大小，將 exponent 比較大的數值放在 operand1 的位置，比較小的放在 operand2，接下來就是將 fraction 隱藏的小數點前的 1 補回去，之後將 operand2 的 exponent 加減以及將 fraction shift，將 operand2 的 exponent 和 operand1 的 exponent 數值相同。



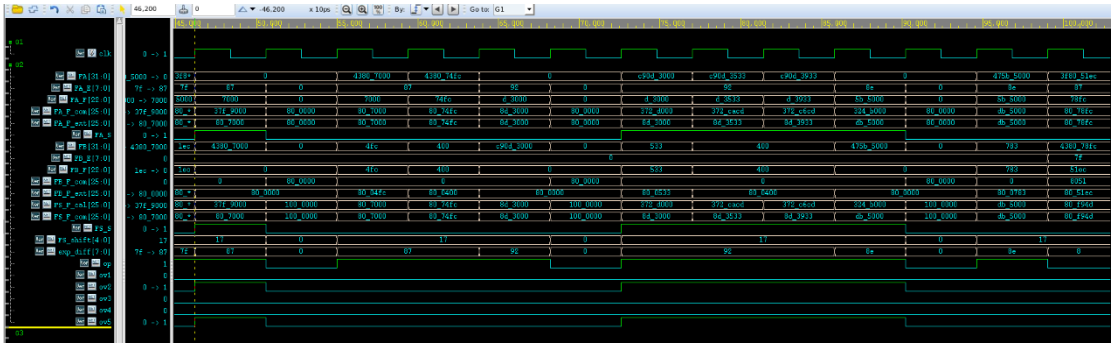
接下來就是依照兩個數值的 signed bit，將其 fraction 轉換為二的補數的形式，這樣就可以直接做加法，所以轉換完後就直接將兩個數值的 fraction 相加，相加後為二的補數形式，所以要依據將加後的數值的第一個 bit，將它當作 signed bit，如果 signed bit 為 1，就要將後面的數值轉換回正數，之後就是依照 fraction 的表示方式，在小數點前省略一個 1，並且只能有一個 1 的形式將 fraction 的小數點做 shift，並且依照 shift 更新 exponent，最後得到的就是答案了，將其作為 result 輸出。

驗證結果:

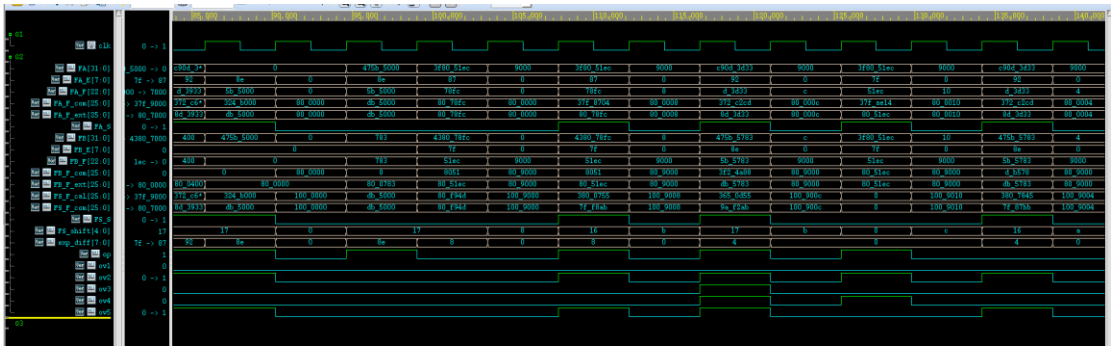


▲先 load 所需要的測資，因為是 32bits 的大數，所以必須用 lui+addi 來產生
mem0:00111111100000000101000111101100
mem1:01000011100000000111100011111100
mem2:11001001000011010011110100110011
mem3:01000111010110110101011110000011

instruction
mem0+mem1
mem2+mem3
mem0-mem1
mem2-mem3
mem3-mem3



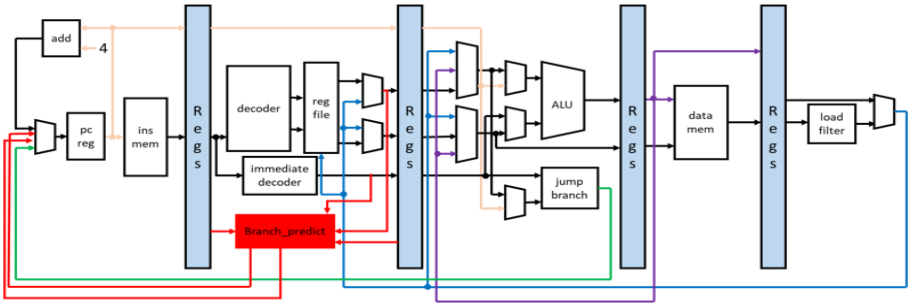
▲可看見測資(1)、(2)結果皆正確
(1)
00111111100000000101000111101100(1.0025)
+
01000011100000000111100011111100(256.9452)
=
01000011100000000111110010100111O(257.9477)
(2)
11001001000011010011110100110011(-578515.15651)
+
01000111010110110101011110000011(56151.51)
=
11001000111111110000111101110101(-522363.64651)
11000011011111111111000101010110



▲可看見測資(3)、(4)結果皆正確

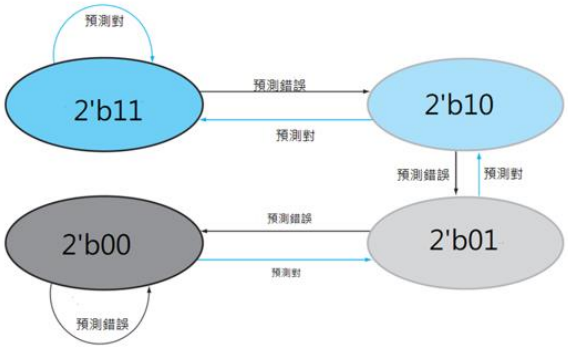
(3)
00111111100000000101000111101100(1.0025)
-
01000011100000000111100011111100(256.9452)
=
11000011011111111111000101010101(-255.9427)
(4)
11001001000011010011110100110011(-578515.15651)
-
01000111010110110101011110000011(56151.51)
=
11001001000110101111001010101011(-634666.66651)

(4) Branch Prediction



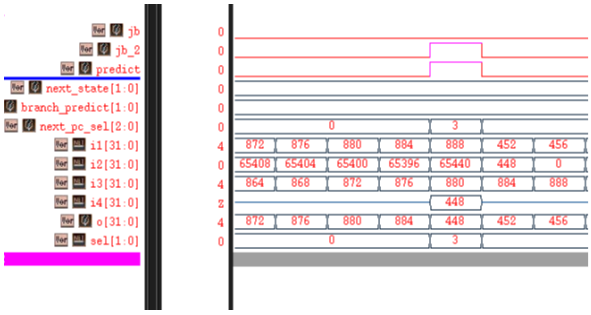
Branch_prediction 計算預測跳轉的 pc 地址(算法同 jump_branch)，並且儲存目前 decode stage 的 pc+4 作為預測錯誤時回復的地址

Controller 負責確認是否執行預測跳轉

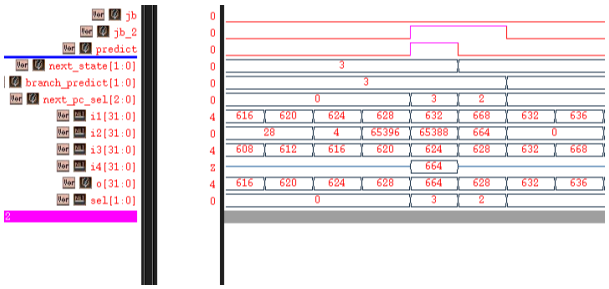


i. 本次的 Branch_prediction 有四個 stage，其中 11、10 會預測跳轉，00、01 則只執行正常跳轉
ii. 所需 rs1、rs2 == Execute、Memory stage 中的 rd 時不執行預測跳轉
iii. 和正常跳轉不同，預測跳轉只需要 flush 掉 fetch 和 decode 間的 register，若預測錯誤時一樣 flush 掉同一個 register
iv. 預測後令 predicted 為 1，並進行檢查，檢查過後在降回 0

結果：



預測正確：o(下一個 pc)選擇 Branch_predict 計算出的跳轉地址，並且 jb_2==1，flush 掉 reg_d，下個 posedge clk 檢查發現預測正確選擇 pc+4



預測錯誤，628□664□628
預測後下一個 clk 檢查到錯誤，再次 flush 掉 reg_d 並且將下一個 pc 定為 decode stage 跳轉時的+4
並且預測 stage 從 3 掉至 2

10、 問題與討論

(1) 關於單一指令需要超過 1 cycle 的指令如何優化：

i. 討論原因：

在 pipeline-CPU 當中如果有一個指令的執行階段超過 1-cycle，CPU 的 fetch 階段以及 execute 階段會 stall (停佇)，使得電腦會有許多單元會進入 idle 的狀態，而且電腦需要經常的使用這種指令的時候，就會造成大量資源和時間的浪費，使得 CPU 的整體效率變的很低。

ii. 討論結果：

可以考慮使用”Speculative execution” (預測執行) 的方式處理，這個是一種優化的方式，讓其他獨立的指令先行，使的在資源過剩的情況下能夠繼續動作。和 branch prediction 很像，但是不再受限於會跳轉的條件判斷或是只適用於迴圈運算的指令條件。

方法優點：

a.執行效率大幅提升，不會受限於執行週期長的指令。

方法缺點：

a. 需要有運算資源被閒置，不會搶奪正在執行指令的資源。

b.增加控制器的複雜度，需要判斷哪些指令沒有資料的相依性。

c.安全性的議題 (關於 speculative cpu 被攻擊的新聞可以在網路上查到)

(2) 關於 cache 在 cpu 應該使用 write-back 還是 write-through

i. 討論原因：

在更改 cache 內容的時候，什麼時候寫回 memory 會影響到後續指令的正確性，以及對 CPU 的效率也有影響。

a. 使用 write-back：更新 cache 之後對 cache 進行標記，不會同步更新 memory 的內容，直到 cache 標記的值要被其他資料覆蓋住時，才會更新 memory 的內容。

b. 使用 write-through：更新 cache 的時候同時對 memory 進行更新。

ii. 討論結果：

a. 使用 buffer 進行緩衝，在合適的時間進行更新。

甲. 這是常見的方式處理 cache 對 memory 更新動作。

乙. 通常會在資料被連續更新兩次以上時，將 buffer 的內容更新給 memory。

b. 根據 access memory 的代價以及對後續指令進行評估選擇。

甲. 因為使用 cache 的原因在於”重複存取同樣位置的記憶體”，所以先檢查後續指令會不會重複存取這個位置決定要不要進行 memory 的更新。

乙. 如果後續指令有很長時間不同的位置使用 write-back

丙. 如果後續指令會重複讀取相同的位置使用 write-through

丁. 但這樣的設計會比較適合 cache 小的 CPU，因為 cache 的更新會很快，只用 write-through 效率太低落，但用 write-back 可能會等不到下一個使用，就會被其他資料覆蓋。

戊. 缺點會是需要硬體檢查後續的指令。

11、 參考資料

大部分資料皆在雲端資料夾裡
https://drive.google.com/drive/folders/1gkNXiBWA3KkTxwD9GrSSlMST1YkHnX9U?usp=share_link

快速乘法器
https://zh.wikipedia.org/zh-tw/%E5%B8%83%E6%96%AF%E4%B9%98%E6%B3%95%E7%AE%97%E6%B3%95
https://blog.csdn.net/zhouxuanyuye/article/details/105742178

除法器
https://ithelp.ithome.com.tw/articles/10161144
小數運算:
https://blog.csdn.net/maxwell2ic/article/details/81076475

branch prediction:
https://zh.wikipedia.org/wiki/%E5%88%86%E6%94%AF%E9%A0%90%E6%B8%AC%E5%99%A8
https://ithelp.ithome.com.tw/m/articles/10265705