VLSI System Design (Graduate Level)

Fall 2021

HOMEWORK I

REPORT

Must do self-checking before submission:

Compress all files described in the problem into one tar

All SystemVerilog files can be compiled under SoC Lab environment

All port declarations comply with I/O port specifications

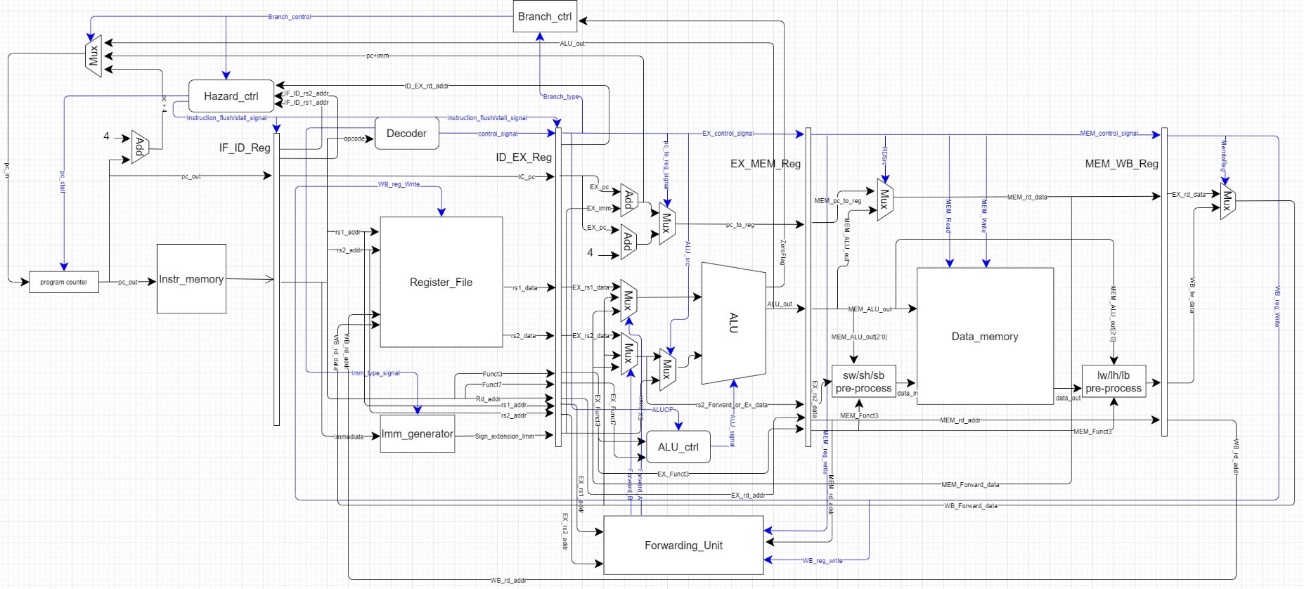
Organize files according to File Hierarchy Requirement

No any waveform files in deliverables

Student name: \_\_\_陳韋綸\_\_\_\_\_\_\_\_

Student ID: \_\_\_\_\_P76091412\_\_\_\_\_

* **Block diagram:**



* **Summary**

在這次的作業中，最重要的地方在於pipeline CPU的這4根pipeline register的控制，並且在遇到data\_hazard的時候要如何判斷才能把需要的值Forward回去，讓ALU可以產生正確的值，而在發生branch hazard的時候需要把已經進入管線當中錯誤的指令和資料都清空，讓指令順序可以正確的被執行，最後在發生load use的時候要stall pipeline把load指令和發生load use的那個指令之間隔開，等於插一個NOP指令進去，讓load 出來的data可以Forward回去。

* **Design explanation**

將CPU利用4個pipeline register分成5個stage，ID、ID、EXE、MEM、WB。

IF:將program counter送到IM裡面，抓取所對應的指令

ID:將讀出來的指令做decode並設定好控制信號線，此外也要從Register File拿取所需要的register裡的資料

EXE:會進行ALU的運算並且決定branch或是jump指令是否要跳以及計算出跳躍位址

MEM:會根據MemRead和MemWrite決定要對DM做讀或是寫

WB；將結果寫回Register File裡

此外在這次的作業中我將要送入IM和DM的clk用反向(~clk)的方式送入，如此一來只需要一個cycle的時間就可以完成給address和拿data這2個行為。

* **Number of lines of RTL code**

1. Top.sv: 53
2. CPU.sv: 456
3. Register\_File.sv: 33
4. Program\_Counter.sv: 19
5. Immediate.sv: 39
6. Hazard\_control.sv: 26
7. Forward\_unit.sv: 24
8. Decoder.sv: 168
9. Branch\_Ctrl.sv: 21
10. ALU.sv: 112
11. ALU\_control.sv: 88
12. SRAM\_wrapper.sv: 99

Total of lines: 1138

* **Result of Superlint**

**一張含有 文字 的圖片

自動產生的描述**

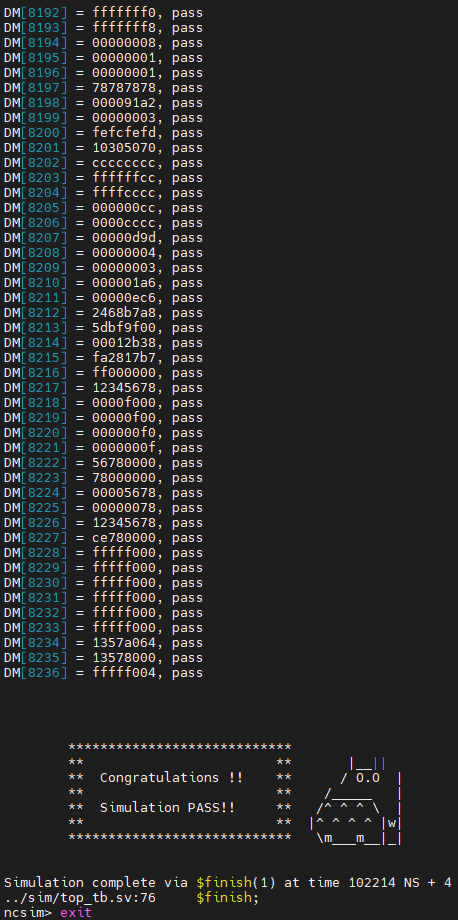
最多的warning是因為Module name跟filename不一致而產生的，所以只要把module name改成與filename相同即可解決warning。

一張含有 桌 的圖片

自動產生的描述

* **Performance(Gate-level simulation)**

1. Program0(37 instructions)



1. Program1(bubble\_sort)

if((&array\_addr)[j+1] < (&array\_addr)[j])

{

temp = (&array\_addr)[j+1];

(&array\_addr)[j+1] = (&array\_addr)[j];

(&array\_addr)[j] = temp;

}

一張含有 文字 的圖片

自動產生的描述

1. Program2(multiplication)

一張含有 文字 的圖片

自動產生的描述

mul1 = 64’h000000001a2b3c4d

mul2 = 64’hffffffff98765432

result = mul1\*mul2=64’hf56a8809cfb90b0a

1. Program3(利用遞迴的方式求得GCD)

int gcd(int a ,int b)

{

if(a==0) return b;

if(b==0) return a;

return gcd(b,a%b);

}

一張含有 文字 的圖片

自動產生的描述

* **Area Report**

一張含有 文字 的圖片

自動產生的描述

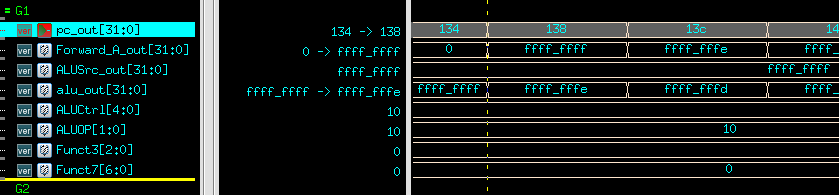
* **Timing Report**

**一張含有 文字 的圖片

自動產生的描述**

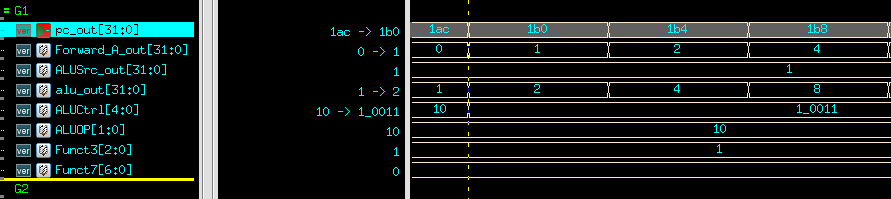
* **Verification Wave**

1. **R\_type**
2. **ADD**



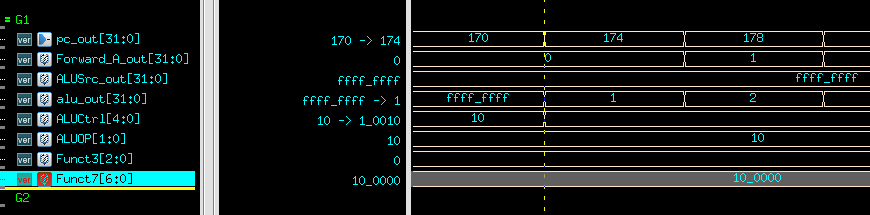
當pc\_out的值為138時，代表EXE\_stage中正在處理的指令為130的add t0,t0,t1，因為ALUOP為2’b10所以會依靠Funct3和Funct7來判斷要執行哪種行為，因此Funct3為3’b000並且Funct7為7’b0000000所以ALUCtrl被設成5’b00010，進行ADD，input\_1(Forward\_A\_out)為32’hffffffff(10進位為-1)，input\_2(ALUSrc\_out)為32’hffffffff(10進位為-1)，所以output(alu\_out)即為32’hfffffffe(10進位為-2)。

1. **SLL**



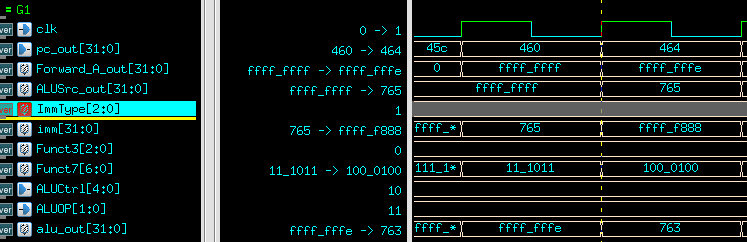
當pc\_out的值為1b0時，代表EXE\_stage中正在處理的指令為1a8的sll t0,t0,t1，因為ALUOP為2’b10所以會依靠Funct3和Funct7來判斷要執行的行為，因此Funct3為3’b001且Funct7為7’b0000000，所以ALUCtrl被設為5’b10011，進行SLL，input\_1(Forward\_A\_out)為32’h1(10進位為1)，input\_2(ALUSrc\_out)為32’h1(10進位為1)，將1左移1位，所以output(alu\_out)即為32’h2(10進位為2)。

1. **SUB**



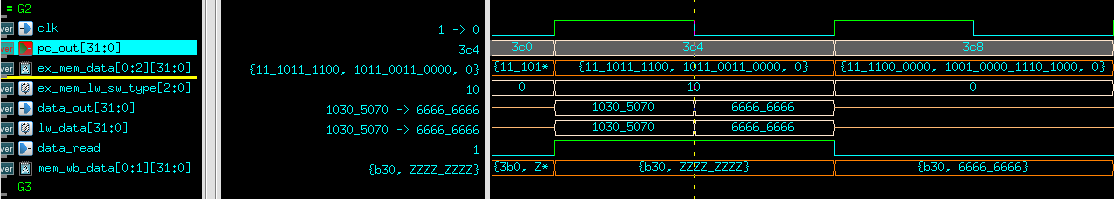
當pc\_out的值為174時，代表EXE\_stage中正在處理的指令為16c的sub t0,t0,t1，因為ALUOP為2’b10所以會依靠Funct3和Funct7來判斷要執行的行為，因此Funct3為3’b001且Funct7為7’b1000000，所以ALUCtrl被設為5’b10010，進行SUB，input\_1(Forward\_A\_out)為32’h0(10進位為0)，input\_2(ALUSrc\_out)為32’hffffffff(10進位為-1)，0減掉-1，所以output(alu\_out)即為32’h1(10進位為1)。

1. **I\_type**
2. **ADDI**



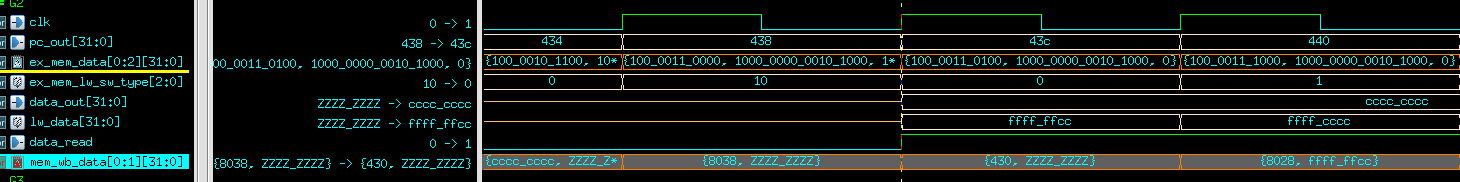
當pc\_out為460時，代表EXE\_stage正在處理458的addi t0,t0,-1指令，因為Imm\_type=3’b001，所以要把Imm做signed-extension變成765，因此ALU的input\_1(Forward\_A\_out)為32’hffffffe(10進位是-2)，input\_2(ALUSrc\_out)為32’h00000765，因此(-2)+765等於763為alu\_out的值。

1. **Load\_word**



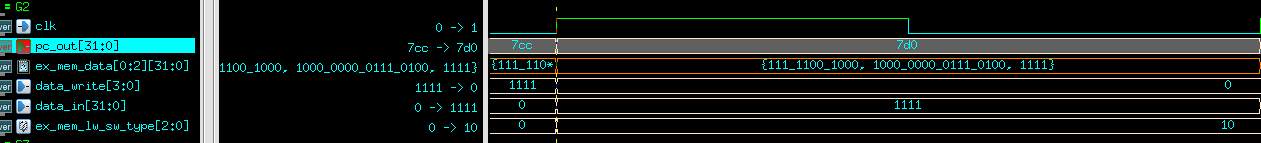
當pc\_out為3c4時，代表Mem\_stage正在處理3b8的lw t0 0(t1)指令，所以在這個時候data\_read會被拉高，又因為我把要送入SRAM的clk變成(~clk)傳入，所以在falling edge時就會把需要的資料準備好並傳出來，也就是data\_out，而又因為ex\_mem\_lw\_sw\_type=3’b010代表要接收的是1個word大小的資料，所以data\_out會把1個word的資料量都傳給lw\_data，然後在下一個clock cycle在把lw\_data傳給mem\_wb\_data[1]裡面。

1. **Load\_Byte**



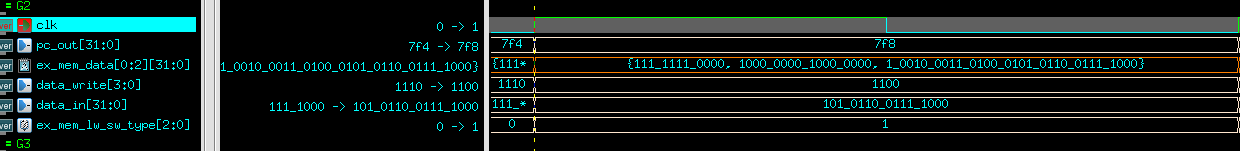
當pc\_out為43c時，代表Mem\_stage正在處理430的lb t1,-16(s0)指令，所以此時data\_read會被拉高，又因為我們是傳(~clk)給SRAM所以在falling edge的時候就會準備好資料並且傳出來，就是data\_out=32’hcccccccc，不過根據ex\_mem\_lw\_sw\_type=3’b000得知這次要的是一個byte的資料量，所以它會再根據ex\_mem\_data[1](此為傳送到SRAM的address)裡面的最後2個bit(2’b00)判斷要從data\_out裡拿最後1個byte的資料，然後再經過signed extension傳到lw\_data=32’hffffffcc中，之後再隨著pipeline傳到mem\_wb\_data[1]寫回register file中。

1. **S\_type**
2. **Store\_word**



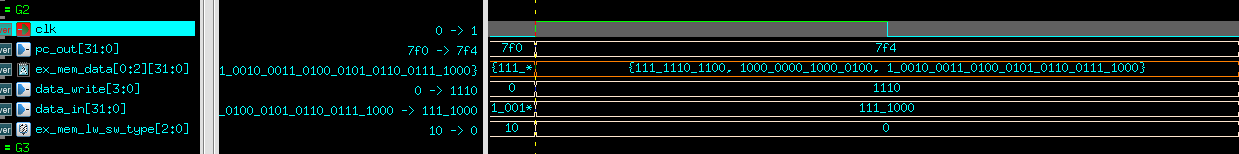
當pc\_out為7d0時，表示Mem\_stage正在處理7c4的sw t0,-4(s0)指令，又因為ex\_mem\_lw\_sw\_type=3’b010代表現在要存的是一個word的資料量，所以會把data\_write設成4’b0000，並且data\_in=32’h00001111即為要存入的資料。

1. **Store\_half\_word**



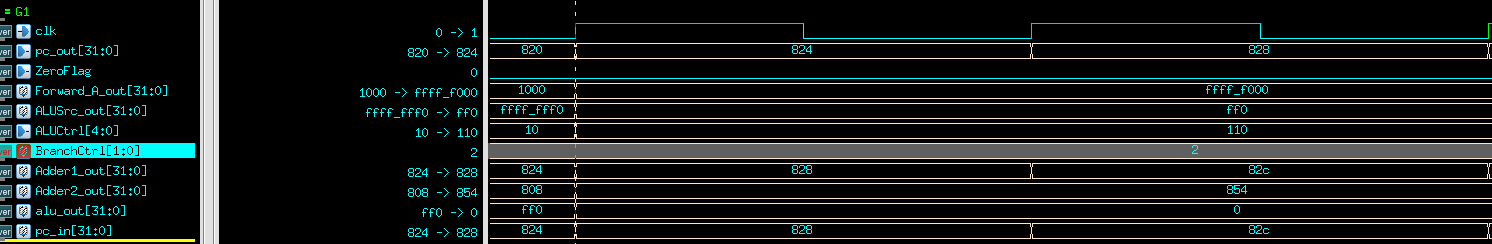
當pc\_out為7f8時，表示Mem\_stage正在執行7ec的sh t5,-12(s0)指令，又因為ex\_mem\_lw\_sw\_type=3’b001代表現在要存的是half-word大小的資料量，所以它會根據ex\_mem\_data[1]的第2個bit判斷需要的data是[31:16]或是[15:0]，如果是1就是[31:16]，反之。因為它的第2個bit為0，所以它就把ex\_mem\_data[2]裡面的[15:0]傳到data\_in並做signed extension=32’h00005678，即為要存入的資料。

1. **Store\_byte**



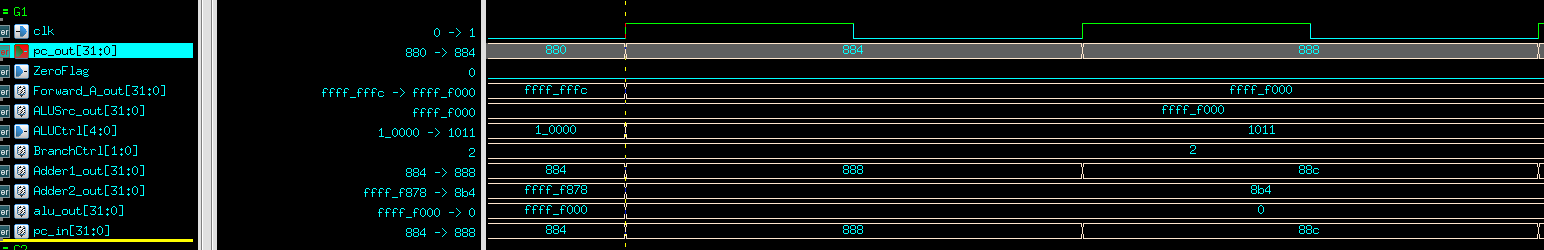
當pc\_out為7f4時代表Mem\_stage正在執行7e8的sb t5 -8(s0)的指令，又因為ex\_mem\_lw\_sw\_type=3’b000代表現在要存的是1個byte大小的資料量，所以它會根據ex\_mem\_data[1]的最後2個bit判斷需要的data是[31:24]、[23:16]、[15:8]或是[7:0]，如果是2’b00就是[7:0]，依序下去。因為它的最後2個bit為2’b00，所以它就把ex\_mem\_data[2]裡面的[7:0]傳到data\_in並做signed extension=32’h00000078，即為要存入的資料。

1. **B\_type**
2. **Beq**



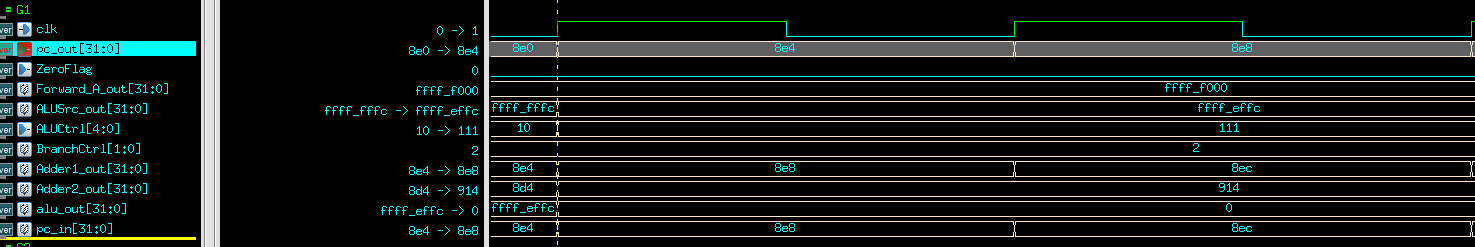
在pc\_out為824時，表示Mem\_stage正在執行81c的beq t0,t1,854指令，此時ALU的input\_1(Forward\_A\_out)為32’hfffff000，input\_2(ALUSrc\_out)為32’h00000ff0，ALUCtrl=5’00110，會去比較2個input是否相等並把結果傳到ZeroFlag，如果相等就等於1否則為0，於是ZeroFlag顯示這2個數不相同，所以會把BranchCtrl設成3’b010，讓Adder1\_out(pc+4)通過Mux傳給pc\_in，而不是Adder2\_out(pc+imm，beq要跳到的位址)。

1. **Bne**



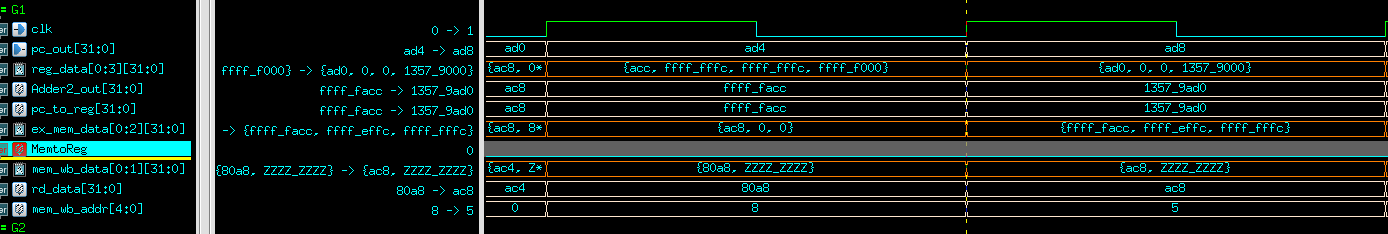
當pc\_out為884時，表示Mem\_stage正在執行87c的bne t0,t1,8b4指令，此時ALU的input\_1(Forward\_A\_out)為32’hfffff000，input\_2(ALUSrc\_out)為32’hfffff000，ALUCtrl=5’01101，會去比較2個input是否不相等並把結果傳到ZeroFlag，如果不相等就等於1否則為0，於是ZeroFlag顯示這2個數是相同的，所以會把BranchCtrl設成3’b010，讓Adder1\_out(pc+4)通過Mux傳給pc\_in，而不是Adder2\_out (pc+imm，beq要跳到的位址)。

1. **Blt**



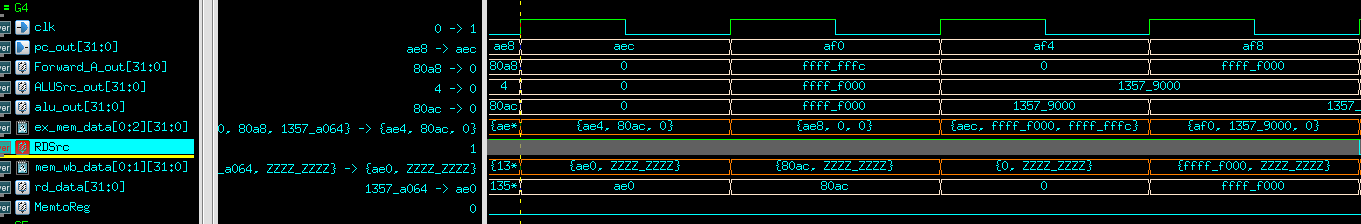
當pc\_out為8e4時，表示Mem\_stage正在執行8dc的blt t0,t1,914指令，此時ALU的input\_1(Forward\_A\_out)為32’hfffff000，input\_2(ALUSrc\_out)為32’hfffeffc，ALUCtrl=5’00111，會去比較input\_1是否小於input\_2並把結果傳到ZeroFlag，如果小於就等於1否則為0，於是ZeroFlag顯示input\_1大於input\_2，所以會把BranchCtrl設成3’b010，讓Adder1\_out(pc+4)通過Mux傳給pc\_in，而不是Adder2\_out (pc+imm，beq要跳到的位址)。

1. U\_type
2. **Auipc**



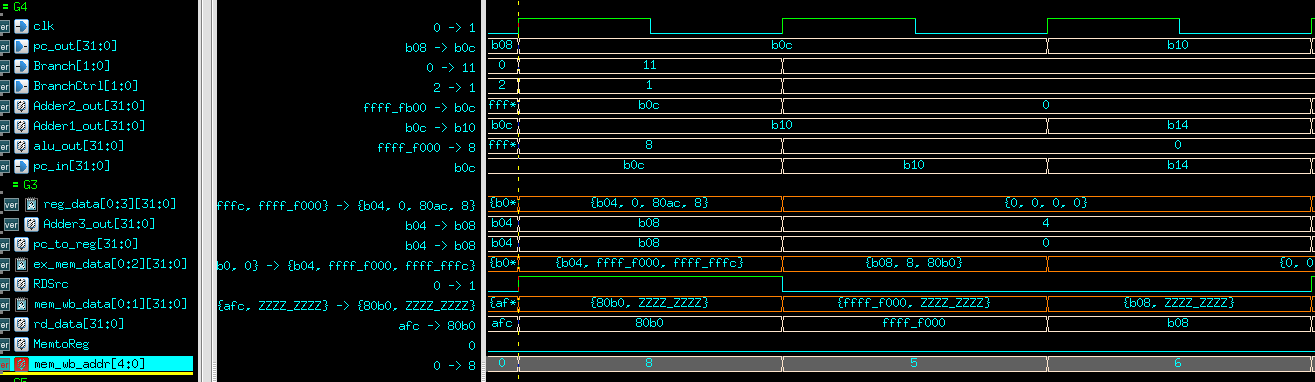
當pc\_out為ad4時，表示EX\_stage正在處理acc的auipc t1,0xfffff，在這個stage的pc為reg\_data[0]=32’h00000acc，而此時的imm為reg\_data[3]=32’hfffff000，因此在經過Adder2之後得到Adder2\_out=32’hfffffacc，之後再通過Mux送到pc\_to\_reg，下一個clk再傳入ex\_mem\_data[0]，即為rd所需要的資料(rd=pc+imm)。

1. **Lui**



當pc\_out為af0時，表示EX\_stage正在處理ae8的lui t1,0xfffff，此時ALU會單純只把input\_2(ALUSrc\_out)=32’hfffff000輸出而已，所以alu\_out=32’hfffff000，然後下一個clk再傳給ex\_mem\_data[1]，再下一個clk傳到mem\_wb\_data[0]，之後把值給到rd\_data再寫到Register\_file裡。

1. **J\_type**
2. **Jal**



當pc\_out為b0c時，表示EX\_stage正在處理b04的jal t1,b0c指令，branch收到opcode的Branch=2’b11知道要做jal，所以把BranchCtrl設成2’b01讓Adder2\_out(pc+imm)=32’h00000b0c通過Mux而不是Adder1\_out(pc+4)=32’h00000b10，所以在下一個clk，pc\_out的指令仍然為b0c，另一方面因為EX\_stage在執行jal指令所以會讓Adder3\_out(pc+4)的值通過Mux而不是Adder2\_out(pc+imm)，再給到pc\_to\_reg，然後下一個clk再隨著pipeline傳到ex\_mem\_data[0]，之後再傳到mem\_wb\_data[0]裡，最後再給到rd\_data，寫入t1暫存器裡。

* **Lesson Learn**

這次作業中最困難的點在於要如何解決3種hazard，而我的解法如下:

1. **Data Hazard**

例子:

Add t1, t1, t2

Sub t3, t1, t2

And t4, t1, t2

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Clk1 | Clk2 | Clk3 | Clk4 | Clk5 | Clk6 | Clk7 |
| Add | IF | ID | EX | MEM | WB |  |  |
| Sub |  | IF | ID | EX | MEM | WB |  |
| And |  |  | IF | ID | EX | MEM | WB |

從例子中可以看到Sub和And所需要的input data都是從Add產生的(紅字部分)，但是因為指令是用pipeline進行的，所以Add在產生完相加的結果之後就要依靠Forwarding unit把EXE的結果Forward給Sub和And，如此一來發生data dependency的指令才不會產生錯誤的結果。

1. **Branch Hazard**

**例子:**

Beq t1, t2, 458(假設要跳)

Add s0, s1, s2

Lw t5 0(s0)

…….

Xor s3, s4,s5(第458行指令)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Clk1 | Clk2 | Clk3 | Clk4 | Clk5 | Clk6 | Clk7 | Clk8 |
| Beq | IF | ID | EX | MEM | WB |  |  |  |
| Add |  | IF | ID | NOP | NOP | NOP |  |  |
| Lw |  |  | IF | NOP | NOP | NOP | NOP |  |
| Xor |  |  |  | IF | ID | EX | MEM | WB |

在這個例子中，如果Beq指令要跳到第458行指令，那在他後面進來的2個指令(Add和Lw)都是錯誤的指令執行順序，所以一旦知道要跳之後就會把ID\_EX\_Reg和IF\_ID\_Reg清成0，就相當於是插入NOP指令，然後讓要被執行的第458行指令(Xor)進入pipeline中。

1. **Load Use**

例子:

Lw t2, 20(s0)

Add t1, t2, t3

Or t4, t2, t3

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Clk1 | Clk2 | Clk3 | Clk4 | Clk5 | Clk6 | Clk7 | Clk8 |
| Lw | IF | ID | EX | MEM | WB |  |  |  |
| Add |  | IF | Stall | ID | EX | MEM | WB |  |
| Or |  |  | Stall | IF | ID | EX | MEM | WB |

在這個例子中，因為Add、Or會用到load word指令的t2，但是因為load use是沒辦法用Forwarding的方式解決，所以一定要暫停pipeline，因此只要Hazard control發現有load use產生，就會把IF\_ID\_Reg暫停，並且把ID\_EX\_Reg裡面的值清0，而EX\_MEM\_Reg和MEM\_WB\_Reg則不受影響隨著pipeline傳送data。