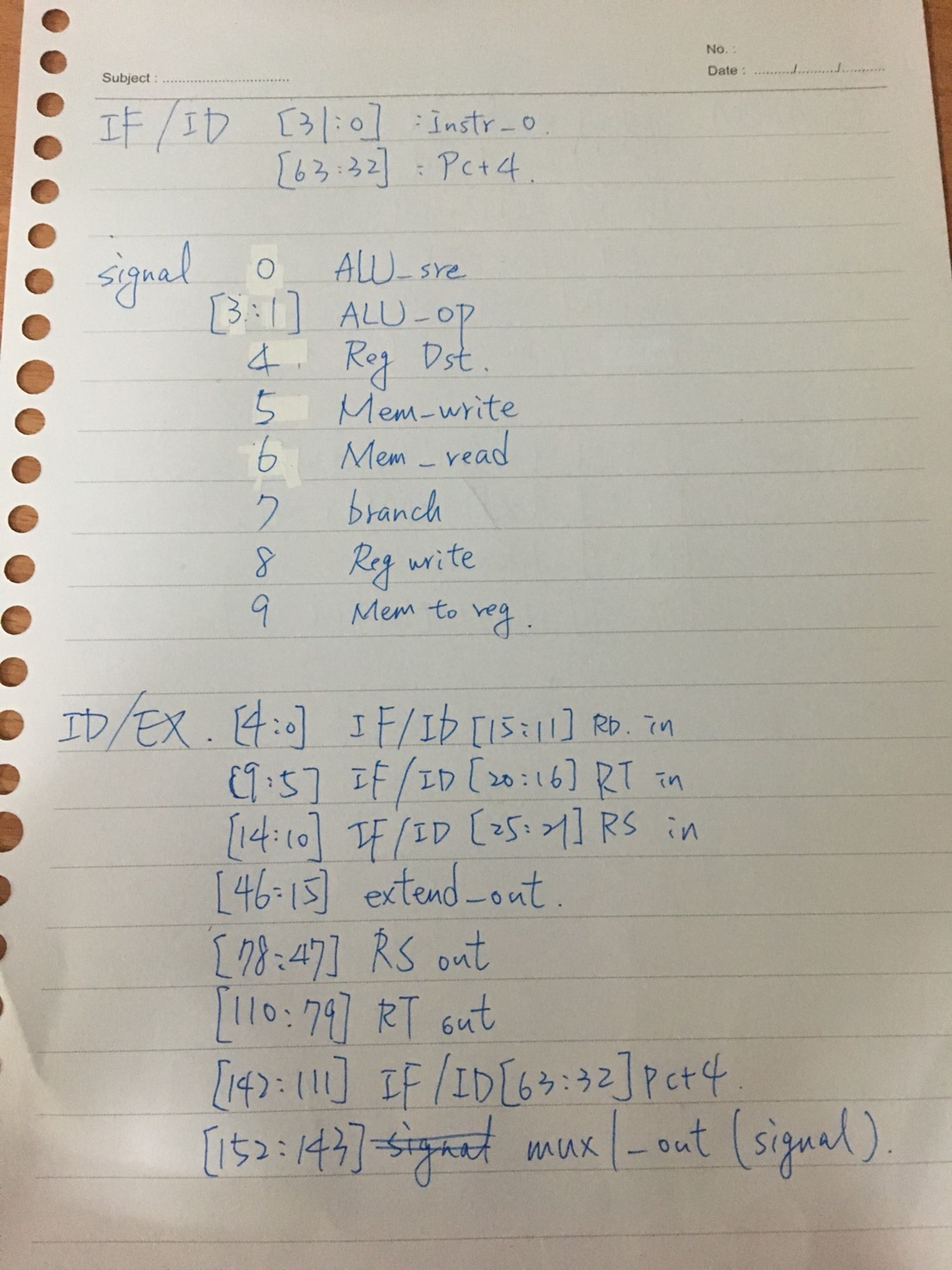
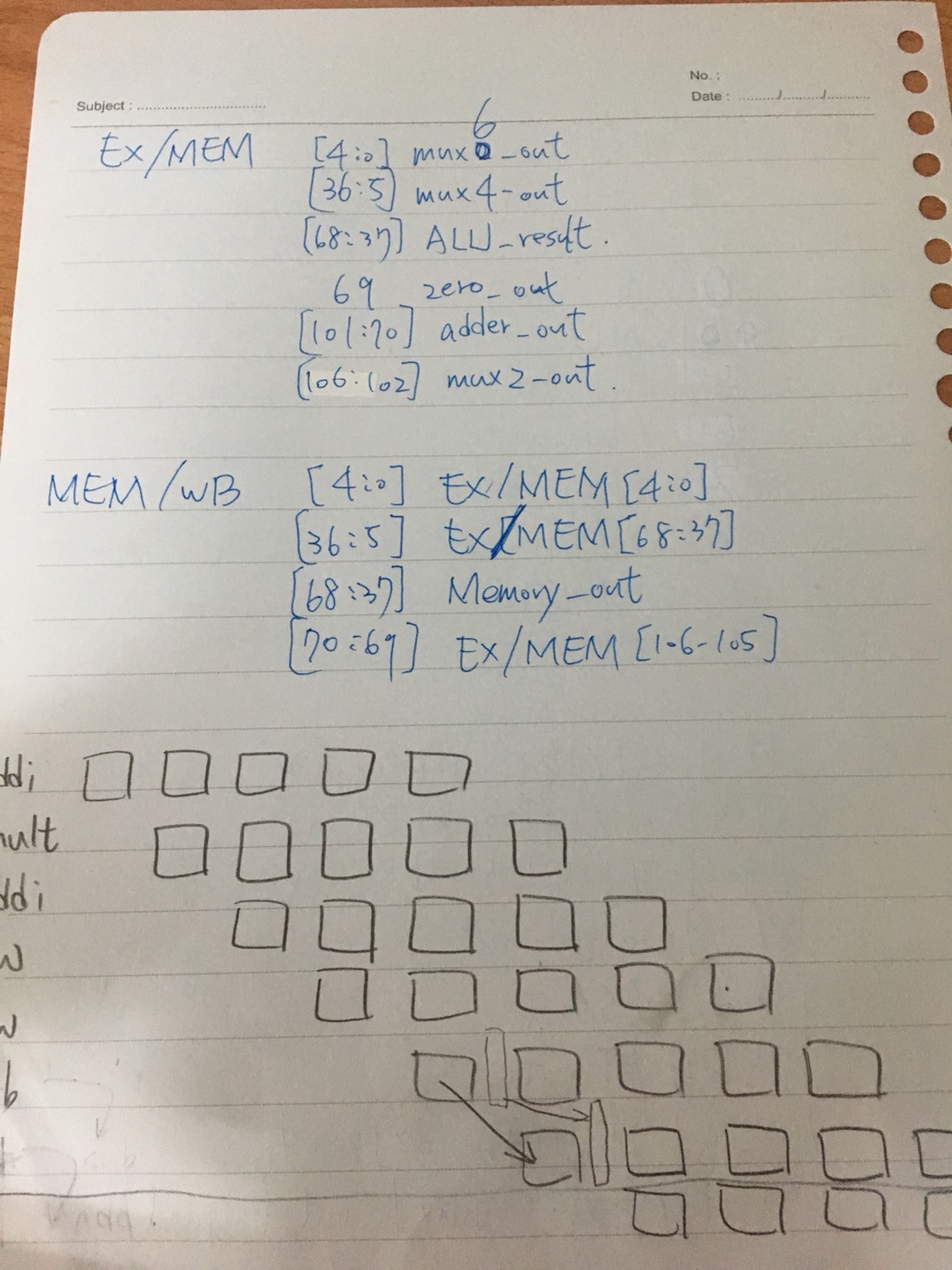
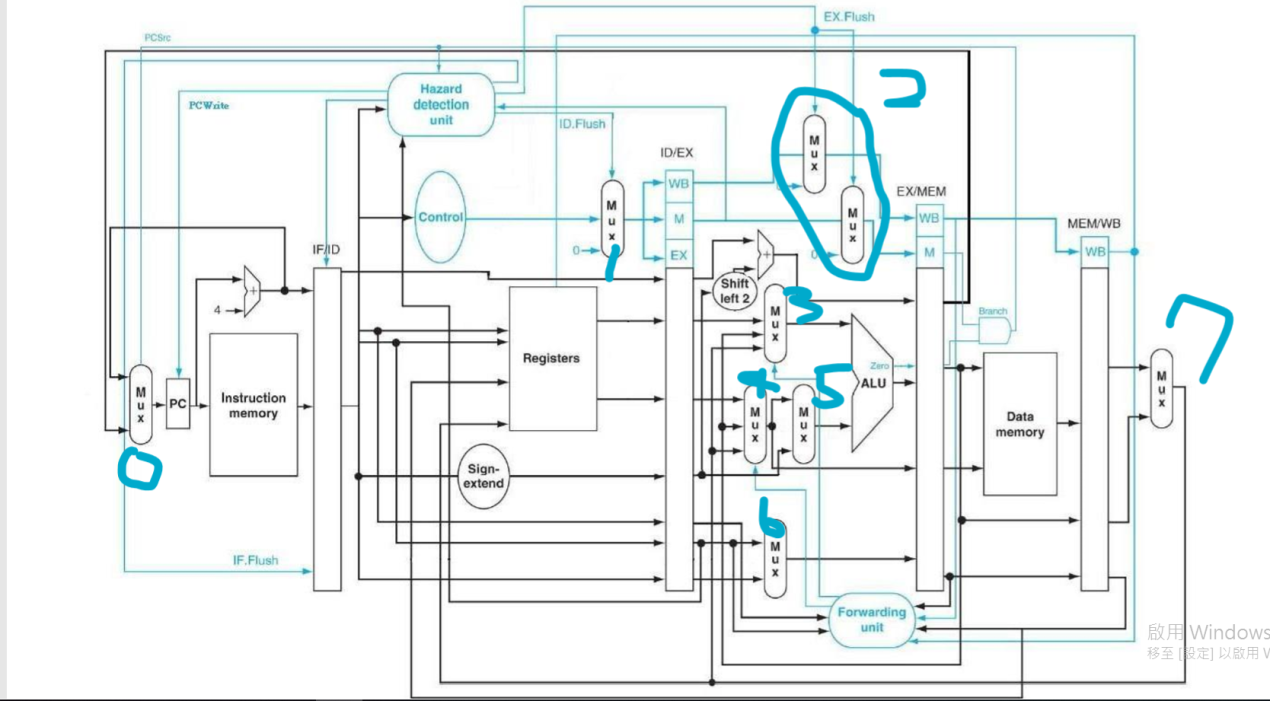
**Computer Organization**

**Source code and the note:**

****

****

**Architecture diagrams:**

****

**Hardware module analysis:**

PC :

控制本次輸入的instruction的位置，一種情況是上一次的instruction+4，另一種是根據branch所跳的位置來讀取insruction。

本次作業中因要配合hazard detection unit，增加了PC unit的stall功能，

可以暫存目前的pc位置一個clock cycle的時間，如果來自hazard detection unit的pc\_write\_out為1時，PC unit便會stall一個clock cycle。

Stall 做法:

assign pc\_out\_o = pc\_delay2;

always @(posedge clk\_i) begin

if(~rst\_i) begin

pc\_delay1 <= 32'b0;

pc\_delay2 <= 32'b0;

end

else begin

pc\_delay1 <= pc\_in\_i;

pc\_delay2 <= pc\_delay1;

if( pc\_write\_i == 1'b0) begin

pc\_delay2 <= pc\_in\_i;

end

end

end

Adder unit:

本次作業中有兩個Adder，第一個是PC+4，第二個是PC+4 + Shift left 的output。

* Adder1:將目前的instruction位置加上4，成為下個instruction位置的選擇之一。
* Adder2:將PC+4 加上的Shift left 的output，成為下個instruction位置的選擇之一。

IM:

根據PC傳送的位置尋找instruction code，並將instruction code輸出到後續的Unit中

Mux Unit:

本次作業中有八個Mux unit。

* Mux0:控制下一個instruction的位置，控制指令是pc\_src(EX\_MEM\_out[104] & EX\_MEM\_out[69])

如果temp3=0，output是PC+4

如果temp=1，output是 PC+4 + Shift left output

* Mux1:控制來自decoder的control signal的輸出，控制指令是hazard detection unit的ID\_flush。

如果ID\_flush=0，output為control signal

如果ID\_flush=1，output為 10’b0

* Mux2:控制MEM和WB的control signal的輸出，控制指令是hazard detection unit的EX\_flush。

如果EX\_flush=0，output為ID\_EX\_out[152:148]( MEM和WB的control signal )

如果EX\_flush=1，output為5’b0

* Mux3:控制alu src1的值，控制指令是forwardA

如果MemtoReg\_o=00，output是ID\_EX\_out[78:47]

如果MemtoReg\_o=10，output是EX\_MEM\_out[68:37]

如果MemtoReg\_o=01，output是mux7\_out

* Mux4:控制alu src2的值，控制指令是forwardB

如果MemtoReg\_o=00，output是ID\_EX\_out[110:79]

如果MemtoReg\_o=10，output是EX\_MEM\_out[68:37]

如果MemtoReg\_o=01，output是mux7\_out

* Mux5:控制alu src2的值，控制指令是alu\_src

如果alu\_src=0，output是Mux4\_out

如果alu\_src=1，output是ID\_EX\_out[46:15]

* Mux6:控制register file RDaddr的輸入，控制指令是Decoder的RegDst，如果RegDst=0，output為ID\_MEM\_out [9:5]

如果RegDst=1，output為ID\_MEM\_out[4:0]

* Mux7:控制回傳到register的值，控制指令是MemtoReg\_o

如果MemtoReg\_o=0，output是memory\_out

如果MemtoReg\_o=1，output是alu\_result

Decoder:

根據讀入的Op code，判斷出各指令，並根據目前指令控制RF、ALU control unit、Mux unit以及branch control。

* Branch\_o :000100(beq)
* MemtoReg\_o: 100011(lw)
* jump\_o[1]:Opcode:000000
* jump\_o[0]:001000(addi)、000000、100011(lw)、101011(sw)、000100(beq)
* MemRead\_o:100011(lw)
* MemWrite\_o:101011(sw)
* ALU\_op\_o [2]:001010(slti)
* ALU\_op\_o [1]:000000
* ALU\_op\_o [0]:000100 (beq)
* ALUSrc\_o: 001000(addi)、100011(lw)、101011(sw)、001010(slti)
* RegWrite\_o: 000000、001000(addi)、100011(lw)、001010(slti)
* RegDst\_o:000000

RF:

暫存instruction中的各個register的值，輸出RS和RT的data，並根據Reg\_write來判斷是否寫入ALU result到 RD。

IM:

將資料寫入儲存，或是將儲存的資料讀到register中。

SE unit:

將instr[15:0]共16bit的data extend成32bit的data，而其數值並未改變。

Shift left unit

將SE output 往左shift 2bit，作為往後adder的input之一。

ALU control unit:

根據Decoder的 ALU\_op及funct\_code來判斷該指令，並輸入相對應的ALU\_control到ALU進行運算。

* ALU\_op=010(R-type)

1. Funct\_code=100000(add): ALU\_control=0010
2. Funct\_code=100010(sub): ALU\_control=0110
3. Funct\_code=100100(and): ALU\_control=0000
4. Funct\_code=100001(or): ALU\_control=0001
5. Funct\_code=101010(slt): ALU\_control=0111
6. Funct\_code=011000(mult):ALU\_control=1000

* ALU\_op=011(addi): ALU\_control=0010
* ALU\_op=100(slti): ALU\_control=0111
* ALU\_op=001(beq):ALU\_control=0110

ALU:

根據從ALU control unit輸入的ALU\_control，將src1和src2進行運算，並輸出zero及result。

Pipeline register:

暫存前一個區段傳輸的資料，讓下一個區段的unit能取用，並避免資料遺失的問題，這次lab中有四個Pipeline register。

本次作業中因要配合hazard detection unit，增加了Pipeline register的stall以及flush功能。

stall可以暫存目前的pc位置一個clock cycle的時間，如果來自hazard detection unit的pc\_write\_out為1時，PC unit便會stall一個clock cycle。

Flush可以將register內的資料清空，當來自hazard detection unit的flush\_out為1時，便執行此功能。

Stall 及 flush:

assign data\_o = data\_delay2;

always@(posedge clk\_i) begin

if(~rst\_i) begin

data\_delay1 <= { size {1'b0} };

data\_delay2 <= { size {1'b0} };

end

else begin

data\_delay1 <= data\_i;

data\_delay2 <= data\_delay1;

if ( write\_i == 1'b0 ) begin

data\_delay2 <= data\_i;

end

if ( flush\_i == 1'b1 ) begin

data\_delay2 <= { size {1'b0} };

end

end

end

四個Pipeline register:

IF\_ID = { pc\_plus4 , instr\_out }

ID\_EX = { mux1\_out , IF\_ID\_out[63:32] , RTdata\_out , RSdata\_out , Extend\_out , IF\_ID\_out[25:21], IF\_ID\_out[20:16], IF\_ID\_out[15:11] }

EX\_MEM = { mux2\_out , sum\_out , zero\_out , result\_out , mux4\_out , mux6\_out }

MEM\_WB = { EX\_MEM\_out[106:105] , memory\_out , EX\_MEM\_out[68:37] , EX\_MEM\_out[4:0] }

Hazard detection unit:

檢測pipeline結構中是否發生hazzard，以下為判斷的條件:

ID\_EX\_mem\_read\_i && ( ( ID\_EX\_RT\_i == IF\_ID\_RS\_i ) || ( ID\_EX\_RT\_i == IF\_ID\_RT\_i ) )

檢測lw的指令及register是否和下個指令的RS register或RT register是否會發生hazard。

如果發生Hazard，Hazard detection unit會透過pc\_write和IF\_ID\_register\_write來stall接下來的instruction，等待資料從memory取出後再透過forward繼續執行。

Forwarding unit:

將資料在寫入register前提早送給之後的instructions，用以下四個條件判斷是否使用forward:

EX\_MEM\_reg\_write\_i && ( EX\_MEM\_RD\_i != 5'b0000 ) && ( EX\_MEM\_RD\_i == ID\_EX\_RS\_i )

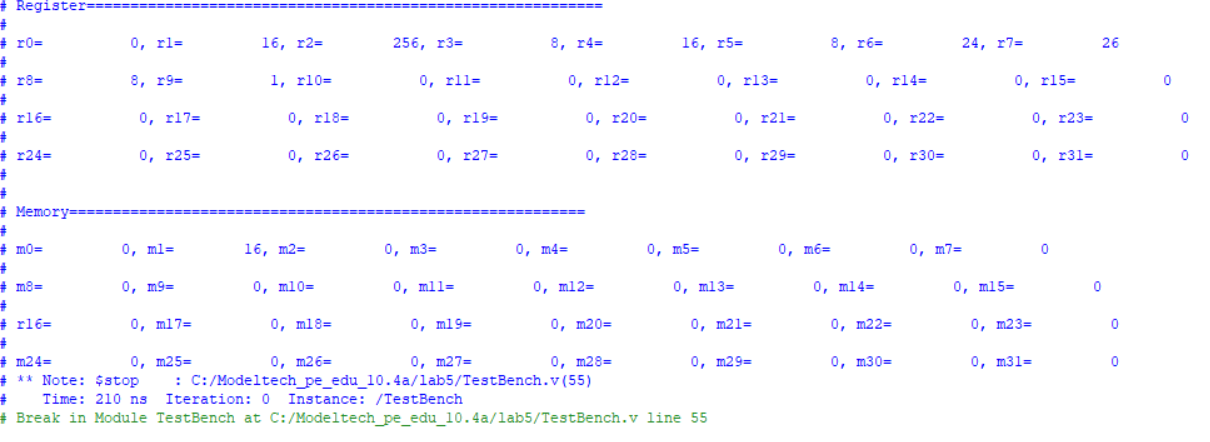
EX\_MEM\_reg\_write\_i && ( EX\_MEM\_RD\_i != 5'b0000 ) && ( EX\_MEM\_RD\_i == ID\_EX\_RT\_i )

MEM\_WB\_reg\_write\_i && ( MEM\_WB\_RD\_i != 5'b0000 ) && ( MEM\_WB\_RD\_i == ID\_EX\_RS\_i )

MEM\_WB\_reg\_write\_i && ( MEM\_WB\_RD\_i != 5'b0000 ) && ( MEM\_WB\_RD\_i == ID\_EX\_RT\_i )

如果資料可以forward，Forwarding unit會使用forwardA和forwardB來控制EX階段中的ALU輸入，選擇要輸入的資料。

**Finished part:**

****

**Problems you met and solutions:**

**本次lab中較困難的部分是hazard detection unit的設計和資料stall的實際操作，我參考了**[**http://www.cnblogs.com/oomusou/archive/2009/06/15/verilog\_dly\_n\_clk.html**](http://www.cnblogs.com/oomusou/archive/2009/06/15/verilog_dly_n_clk.html)**中的stall做法，可是stall的unit的always block的觸發條件需要更改，如果照之前的觸發條件，stall的值有可能會發生錯誤。**

**Summary:**

**透過這次lab，學習到Pipeline 系統中hazard detection和forwarding的操作，也更有系統地整理cpu datapath的細節。**