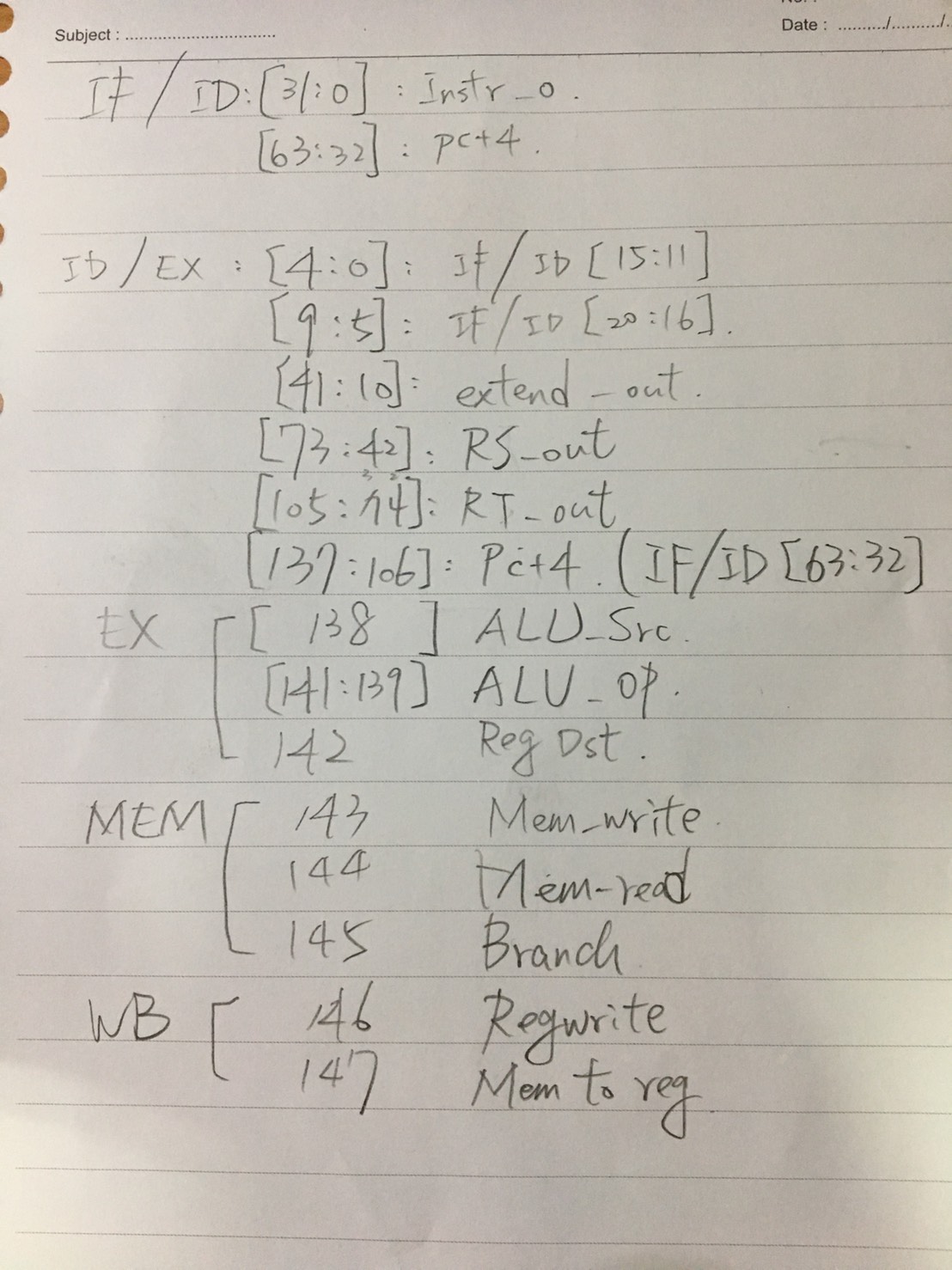
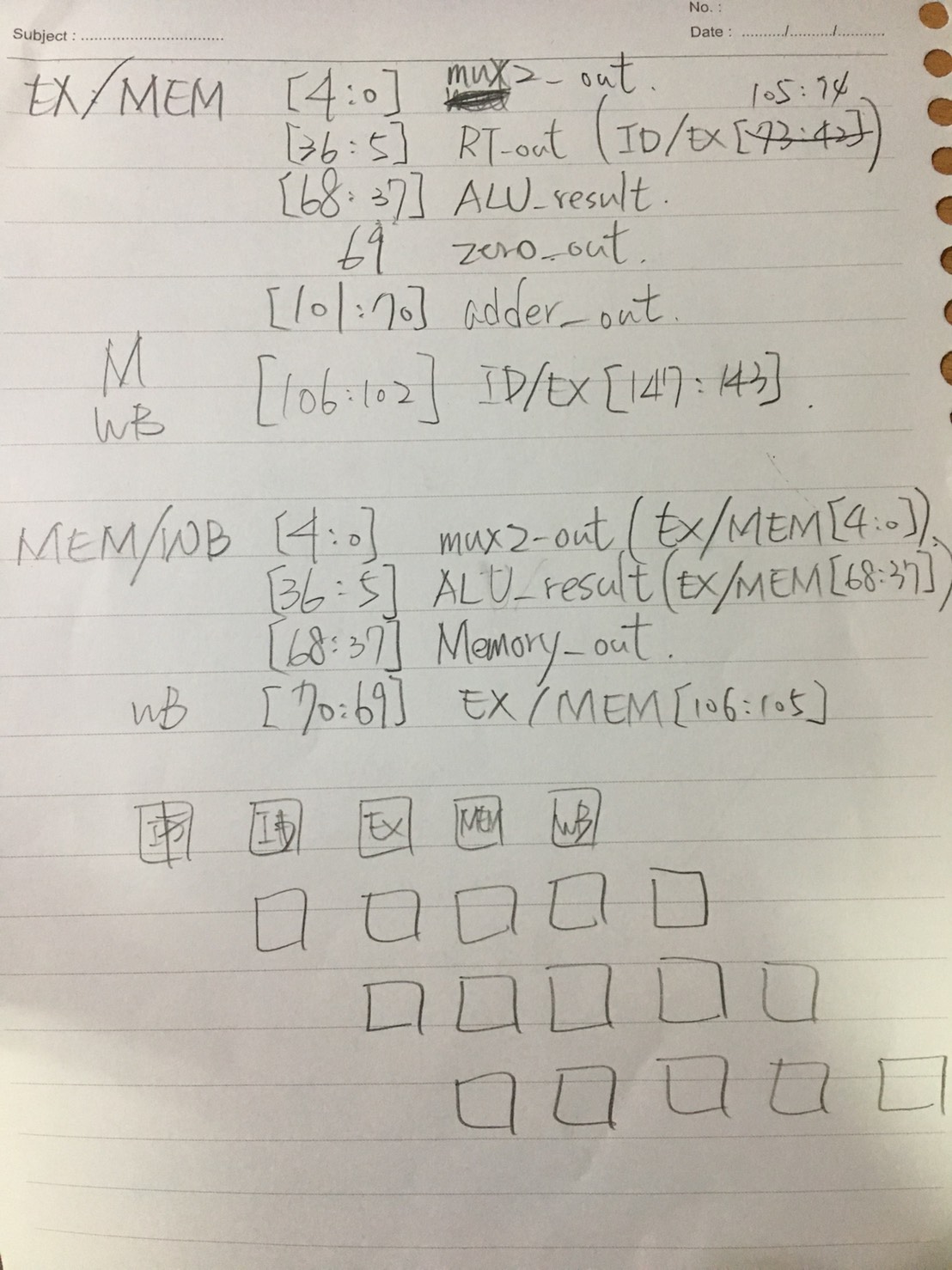
**Computer Organization**

**Source code and the note:**

****

****

**Machine code in test2:**

**00100000000000010000000000010000 //addi $1,$0 ,16**

**00100000000000110000000000001000 //addi $3,$0 ,8**

**00000000000000000000000000000000 // NOP**

**00000000000000000000000000000000 //NOP**

**00100000001000100000000000000100 //addi $2,$1 ,4**

**10101100000000010000000000000100 //sw $1,4($0)**

**10001100000001000000000000000100 //lw $4,4($0)**

**00000000011000010011000000100000 //add $6,$3 ,1**

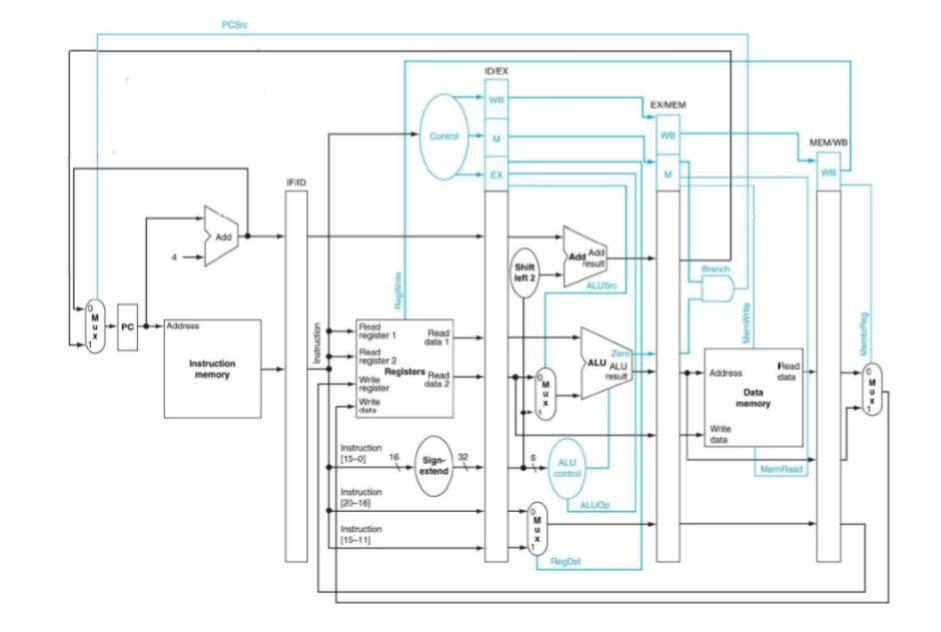
**00100000001001110000000000001010 //addi $7,$1 ,10**

**00000000100000110010100000100010 //sub $5,$4 ,$3**

**00100000000010010000000001100100 //addi $9,$0 ,100**

**00000000111000110100000000100100 //and $8,$7 ,$3**

**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。

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:

本次作業中有四個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ALU src2的輸入，控制指令是Decoder的RegSrc。

如果RegSrc=0，output為RT data output

如果RegSrc=1，output為Sign extention output

* Mux2:控制register file RDaddr的輸入，控制指令是Decoder的RegDst，如果RegDst=0，output為instr\_out[20:16]

如果RegDst=1，output為instr\_out[15:11]

* Mux3:控制回傳到register的值，控制指令是MemtoReg\_o

如果MemtoReg\_o=00，output是alu\_result

如果MemtoReg\_o=01，output是memory\_out

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

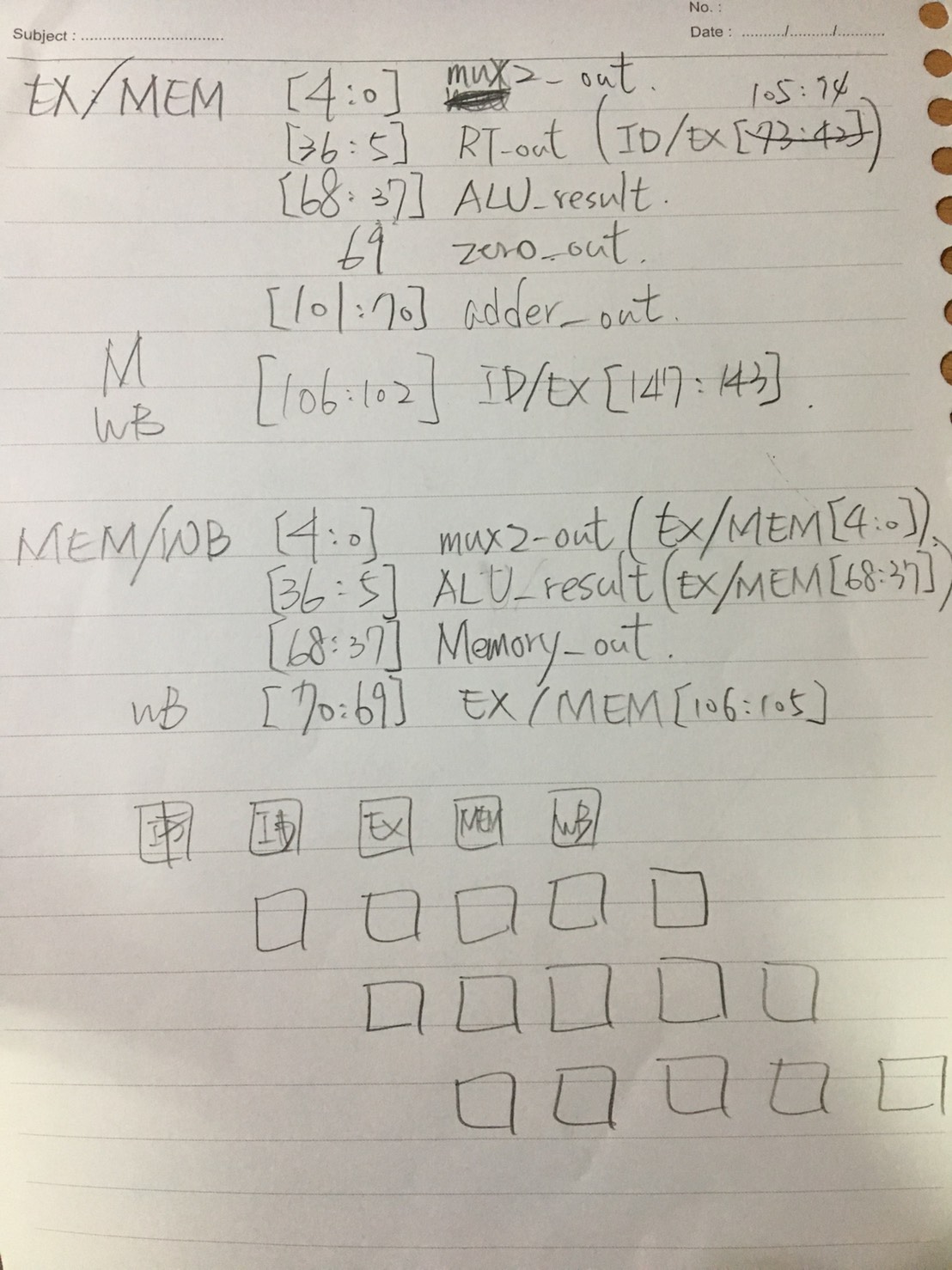
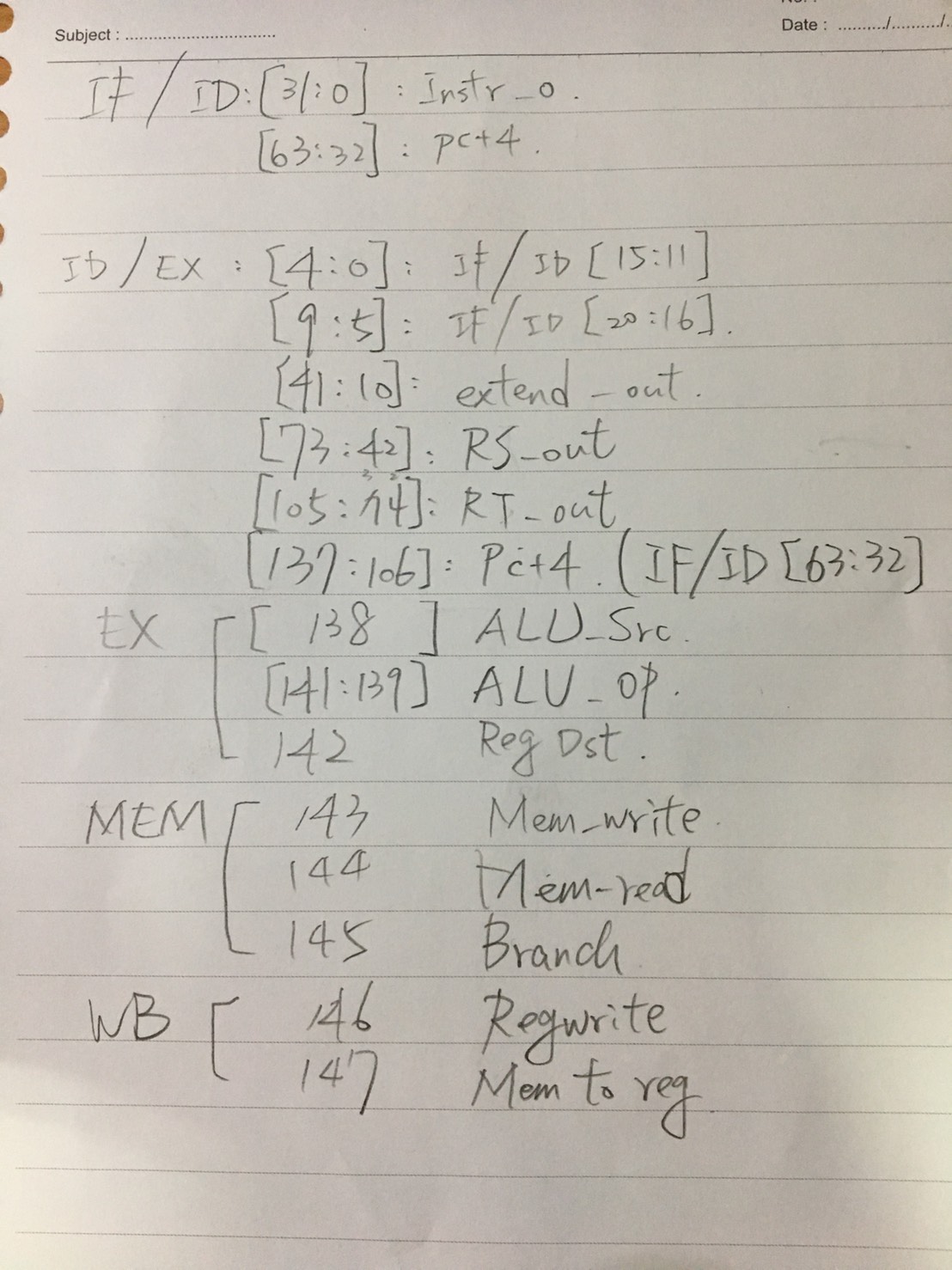
* ALU\_op=011(addi): ALU\_control=0010
* ALU\_op=100(slti): ALU\_control=0111
* ALU\_op=001(beq):ALU\_control=0110

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。



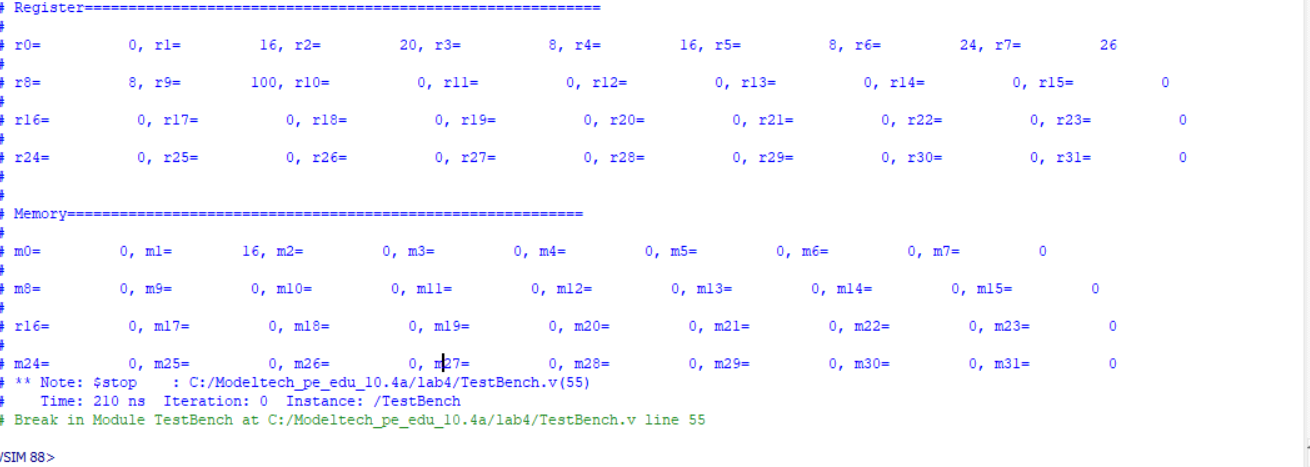
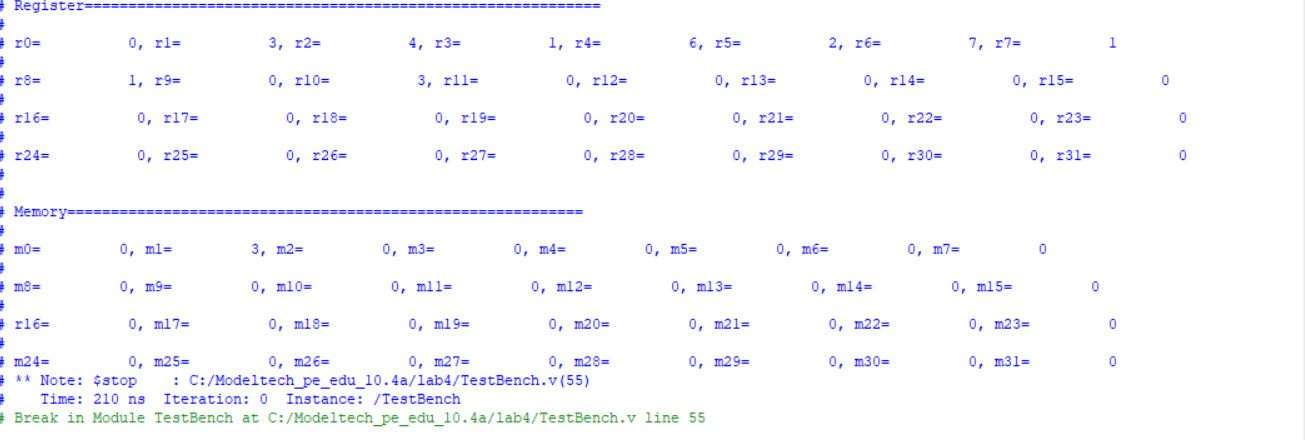
IF\_ID\_in = { pc\_plus4 , instr\_out }

ID\_EX\_in = { MemtoReg\_out , RegWrite\_out , Branch\_out , MemRead\_out , MemWrite\_out , RegDst\_out , ALU\_op\_out , ALUSrc\_out , IF\_ID\_out[63:32] , RTdata\_out , RSdata\_out , Extend\_out , IF\_ID\_out[20:16], IF\_ID\_out[15:11] }

EX\_MEM\_in = { ID\_EX\_out[147:143] , sum\_out , zero\_out , result\_out , ID\_EX\_out[105:74] , mux2\_out }

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

**Finished part:**

****

**Problems you met and solutions:**

**Pipeline register的設計上較複雜，造成資料寫入位置錯誤。RTdata\_out在EX\_MEM\_in中的位置為ID\_EX\_out[105:74]，可是不小心登記成RSdata\_out的位置(ID\_EX\_out[73:42])，造成memory儲存資料時的錯誤。**

**Summary:**

**透過這次lab，認知到Pipeline 的形式能讓CPU處理資料上更有效率，也能更有系統地整理datapath的細節。**