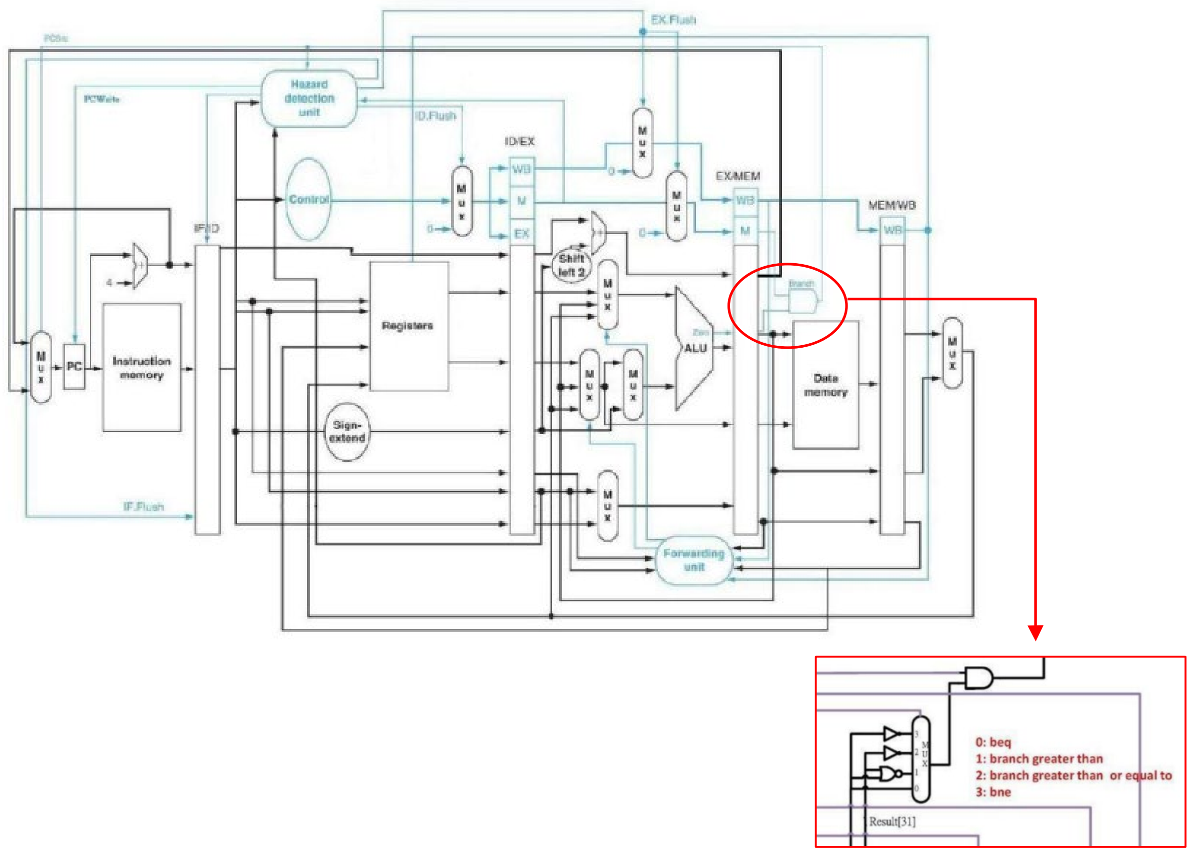
Computer Organization Lab5

Name:

ID:

Architecture diagrams:



(from CO Lab3.pdf)

Hardware module analysis:

Pros: can handle data hazard, double data hazard, and load-use hazard, thus improve the performance since it can save the time to stall or wait for other instructions to complete

Cons: the complexity is increased, making it more challenging to maintain the CPU, and has higher power consumption

- The following modules are the same as in Lab4:
  Adder, ALU, MUX_2to1, Shift_Left_Two_32, Sign_Extend

- MUX_3to1, MUX_4to1 are similar to MUX_2to1

- ALU_Ctrl

	Input									Output			
	fc5	fc4	fc3	fc2	fc1	fc0	op2	op1	op0	ctr3	ctr2	ctr1	ctr0
add	1	0	0	0	0	0	1	1	0	0	0	1	0
sub	1	0	0	0	1	0	1	1	0	0	1	1	0
and	1	0	0	1	0	0	1	1	0	0	0	0	0
or	1	0	0	1	0	1	1	1	0	0	0	0	1
slt	1	0	1	0	1	0	1	1	0	0	1	1	1
mult	0	1	1	0	0	0	1	1	0	1	0	0	0
addi	x	x	x	x	x	x	0	0	0	0	0	1	0
slti	x	x	x	x	x	x	1	0	1	0	1	1	1
beq	x	x	x	x	x	x	0	0	1	0	1	1	0
bne	x	x	x	x	x	x	0	0	1	0	1	1	0
bge	x	x	x	x	x	x	0	0	1	0	1	1	0
bgt	x	x	x	x	x	x	0	0	1	0	1	1	0
lw	x	x	x	x	x	x	0	0	0	0	0	1	0
sw	x	x	x	x	x	x	0	0	0	0	0	1	0

given function field　　ALU opcode　　ALU control

➔ ctr3 = fc4 & op1
ctr2 = (~fc2 & fc1 & op1) | op0
ctr1 = (~fc3 & ~fc2) | fc1 | (~op1)
ctr0 = (fc0 & op1) | (fc3 & fc1 & op1) | (op2 & ~op1)

➢ Decoder

	Input						Output						
	op5	op4	op3	op2	op1	op0	RD	alu op2	alu op1	alu op0	Src	RW	Brh
R-f	0	0	0	0	0	0	1	1	1	0	0	1	0
addi	0	0	1	0	0	0	0	0	0	0	1	1	0
slti	0	0	1	0	1	0	0	1	0	1	1	1	0
beq	0	0	0	1	0	0	x	0	0	1	0	0	1
bne	0	0	0	1	0	1	x	0	0	1	0	0	1
bge	0	0	0	0	0	1	x	0	0	1	0	0	1
bgt	0	0	0	1	1	1	x	0	0	1	0	0	1
lw	1	0	0	0	1	1	0	0	0	0	1	1	0
sw	1	0	1	0	1	1	x	0	0	0	1	0	0

given op field　RegDst　ALU opcode　ALUSrc　RegWrite　Branch

➔ RD = ~op3 & ~op2 & ~op1
alu_op2 = ~op2 & ~op0 & (~op3 | op1)
alu_op1 = ~op3 & ~op2 & ~op0
alu_op0 = ~op5 & (op2 | op1| op0)
Src = op5 | op3
RW = (~op2 & ~op0) | (op5 & ~op3)
Brh = op2 | (~op1 & op0)

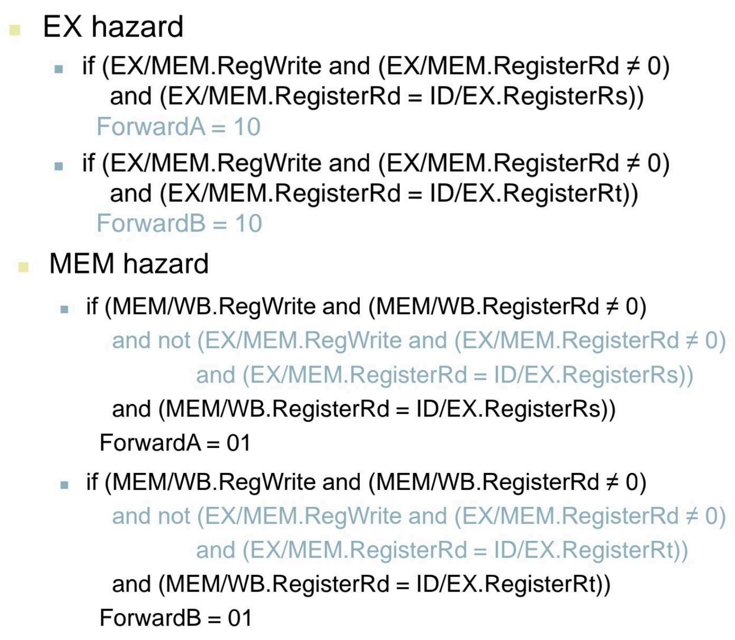
	Input						Output			
	op5	op4	op3	op2	op1	op0	BrhType	memread	memwrite	memtoreg
R-f	0	0	0	0	0	0	xx	0	0	1
addi	0	0	1	0	0	0	xx	0	0	1
slti	0	0	1	0	1	0	xx	0	0	1
beq	0	0	0	1	0	0	00	0	0	x
bne	0	0	0	1	0	1	11	0	0	x
bge	0	0	0	0	0	1	10	0	0	x
bgt	0	0	0	1	1	1	01	0	0	x
lw	1	0	0	0	1	1	xx	1	0	0
sw	1	0	1	0	1	1	xx	0	1	x

given op field　BranchType　MemRead　MemWrite　MemtoReg

➔ BrhType[1] = ~op1 & op0
BrhType[0] = op2 & op0
memread = op5 & ~op3
memwrite = op5 & op3
memtoreg = ~op0

➢ Forwarding

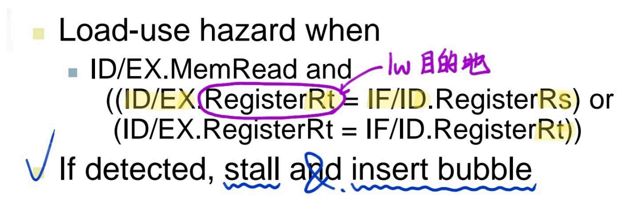
(from Ch4 ppt)



- EX hazard
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 10
  - if (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 10
- MEM hazard
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRs))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRs))
    ForwardA = 01
  - if (MEM/WB.RegWrite and (MEM/WB.RegisterRd ≠ 0)
    and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd ≠ 0)
    and (EX/MEM.RegisterRd = ID/EX.RegisterRt))
    and (MEM/WB.RegisterRd = ID/EX.RegisterRt))
    ForwardB = 01

➢ Hazard_Detector

(from Ch4 ppt)



- Load-use hazard when
  - ID/EX.MemRead and — lw 目的地
    ((ID/EX.RegisterRt = IF/ID.RegisterRs) or
    (ID/EX.RegisterRt = IF/ID.RegisterRt))

√ If detected, stall ~~and~~ & insert bubble

(besides, flush IF/ID, ID/EX, and EX/MEM pipeline registers when a branch launch)

```
if Branch == 1 :
        IF_flush <= 1;
```

ID_flush <= 1;
EX_flush <= 1;

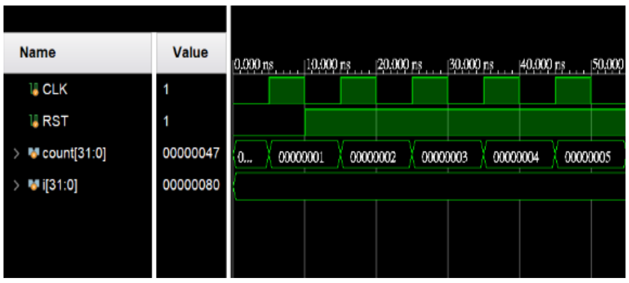
➢ Pipelined_CPU

Compare with Lab4, the Pipelined CPU this time includes two new modules, Hazard_Detector and Forwarding, to cope with hazard issues. Therefore, more signal controls are needed, and they are shown in blue in the diagram.

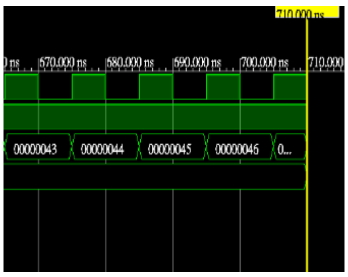
Besides, to implement four types of branch, I add a 4to1 MUX so that it can execute the various function according to the BranchType signal.

After all, the work is to connect the wires based on the architecture diagram and done!

<u>Finished part</u>:



…

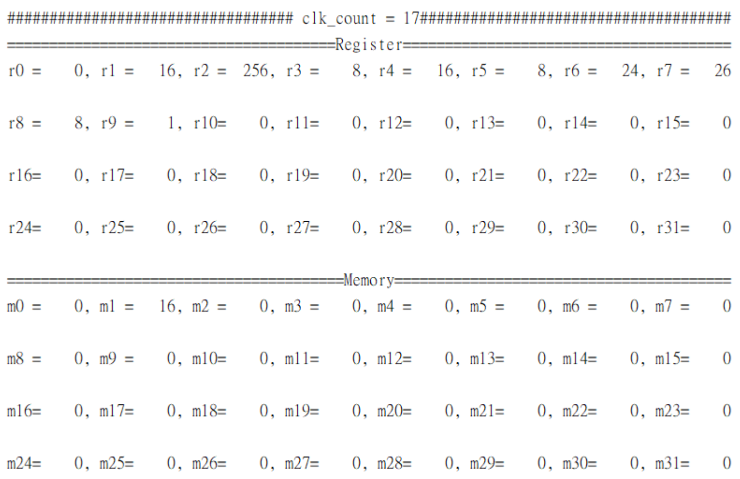


The first time both CLK and RST are triggered is the second cycle. From that time on, every time CLK is triggered, PC either reads in the next address or repeat the same one, determined by "pc_write" signal.

The whole process ends at the 70th cycle, which is the maximum clock count. (On the other hand, test_1 ends at the 17th cycle, by changing "MAX_COUNT" in testbench)

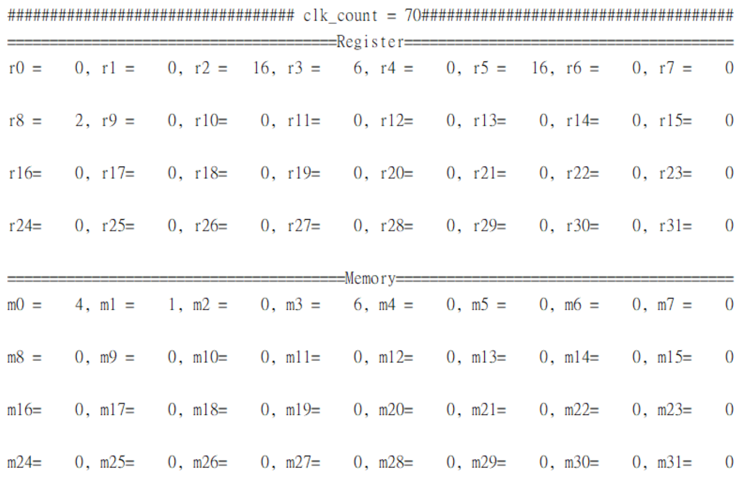
From the simulation results below, it is obvious that the CPU can correctly handle with data hazard and finish the execution in a limited time.

✓ test_1



```
################################# clk_count = 17####################################
=====================================Register=====================================
r0 =    0, r1 =   16, r2 =  256, r3 =    8, r4 =   16, r5 =    8, r6 =   24, r7 =   26

r8 =    8, r9 =    1, r10=    0, r11=    0, r12=    0, r13=    0, r14=    0, r15=    0

r16=    0, r17=    0, r18=    0, r19=    0, r20=    0, r21=    0, r22=    0, r23=    0

r24=    0, r25=    0, r26=    0, r27=    0, r28=    0, r29=    0, r30=    0, r31=    0

======================================Memory======================================
m0 =    0, m1 =   16, m2 =    0, m3 =    0, m4 =    0, m5 =    0, m6 =    0, m7 =    0

m8 =    0, m9 =    0, m10=    0, m11=    0, m12=    0, m13=    0, m14=    0, m15=    0

m16=    0, m17=    0, m18=    0, m19=    0, m20=    0, m21=    0, m22=    0, m23=    0

m24=    0, m25=    0, m26=    0, m27=    0, m28=    0, m29=    0, m30=    0, m31=    0
```

✓ test_2



```
################################# clk_count = 70####################################
=====================================Register=====================================
r0 =    0, r1 =    0, r2 =   16, r3 =    6, r4 =    0, r5 =   16, r6 =    0, r7 =    0

r8 =    2, r9 =    0, r10=    0, r11=    0, r12=    0, r13=    0, r14=    0, r15=    0

r16=    0, r17=    0, r18=    0, r19=    0, r20=    0, r21=    0, r22=    0, r23=    0

r24=    0, r25=    0, r26=    0, r27=    0, r28=    0, r29=    0, r30=    0, r31=    0

======================================Memory======================================
m0 =    4, m1 =    1, m2 =    0, m3 =    6, m4 =    0, m5 =    0, m6 =    0, m7 =    0

m8 =    0, m9 =    0, m10=    0, m11=    0, m12=    0, m13=    0, m14=    0, m15=    0

m16=    0, m17=    0, m18=    0, m19=    0, m20=    0, m21=    0, m22=    0, m23=    0

m24=    0, m25=    0, m26=    0, m27=    0, m28=    0, m29=    0, m30=    0, m31=    0
```

Problems you met and solutions:

At first I forgot to write default conditions in Hazard_Detector and Forwarding unit, so I got "x" value for some registers. To track how the signals be delivered, I add a few printing statements in testbench, and finally found out the problem.

In fact, I encountered more problems this time, but others are trivial (like wires wrong connection), so I don't elaborate on each situation here. But all the solution is to track the signals like mentioned above.

Summary:

The structure of the CPU was much more complicated than before, so I made many mistakes this time. To debug was distressing, since there are lots of wires and modules, and I didn’t know where to start with. Finally, I decided to track down some key signals patiently and found out the problems at last.

Although it took me many time to do the debug work, during the process, I got to know more about the signal delivery, and gained a deeper understanding of the working principle behind the CPU. I regard it as a good practice!