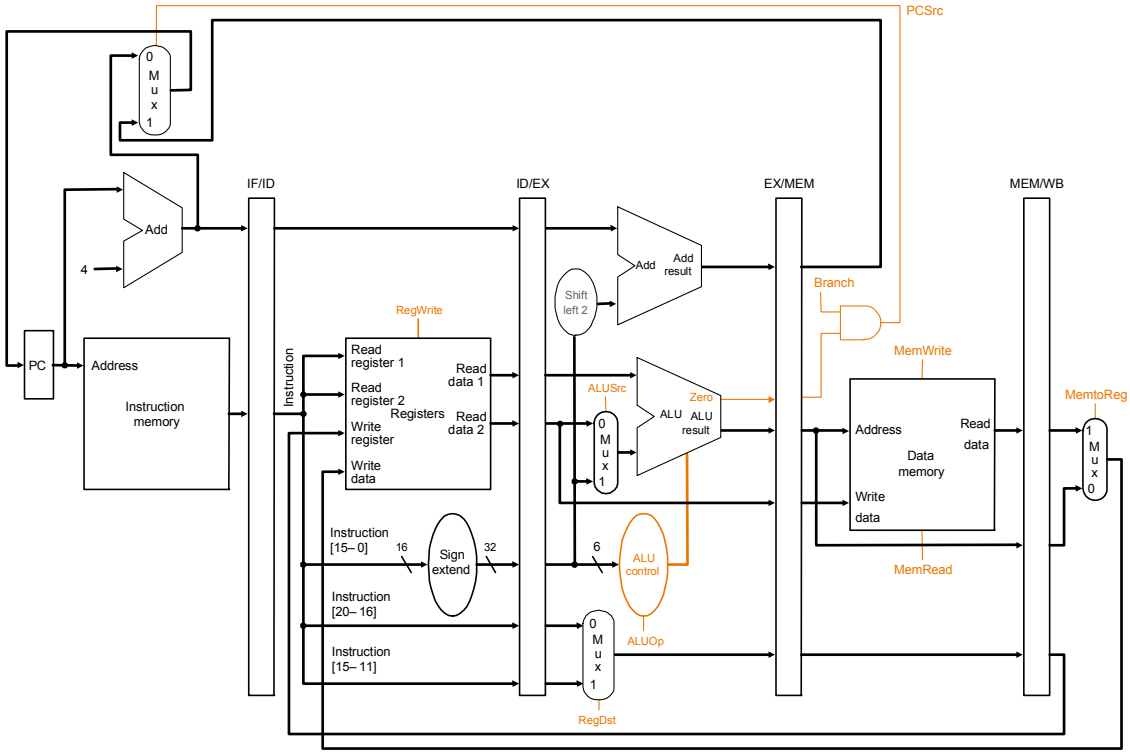
# Control Signal for Pipeline Processor

# Pipeline Control

- Three steps are needed
  - Step 1: Begin with the same control signal with single-cycle datapath control
  - Group control lines into five groups according to pipeline stage
    - Since control signals are associated with components active during a single pipeline stage
  - Set control signals during each pipeline stage

# Pipelined Datapath with Control – Step 1

**Start with Same control signals as the single-cycle datapath**



# Pipeline Control Signals- Step 2

- Group control lines into five groups according to pipeline stage
  - instruction fetch / PC increment (IF)
  - instruction decode / register fetch (ID)
  - execution / address calculation (EX)
  - memory access (M)
  - write back (WB)

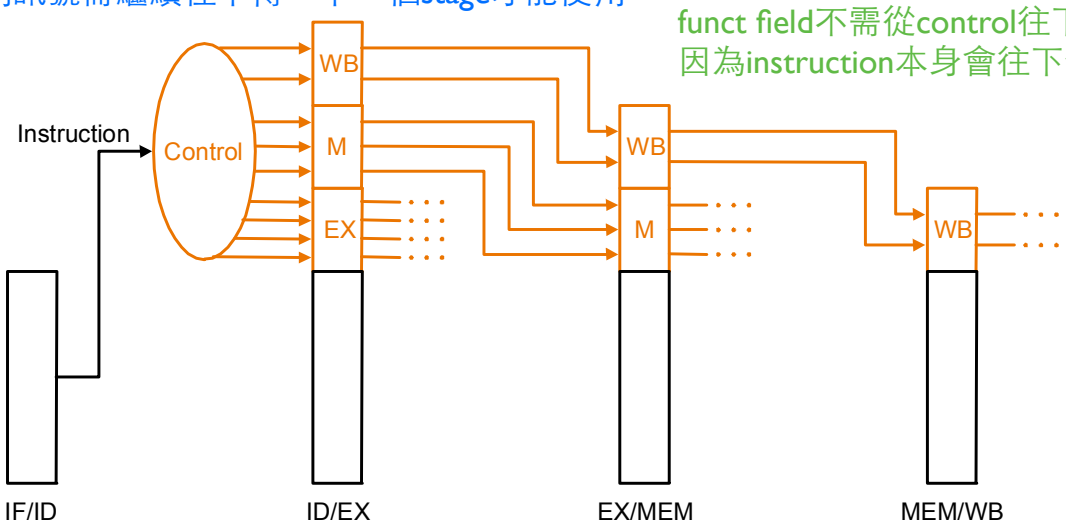
} Nothing to control as instruction memory read and PC write are always enabled

} (the following table)

| | EX Execution/Address Calculation stage control lines | | | | M Memory access stage control lines | | | WB Write-back stage control lines | |
|---|---|---|---|---|---|---|---|---|---|
| Instruction | Reg Dst | ALU Op1 | ALU Op0 | ALU Src | Branch | Mem Read | Mem Write | Reg write | Mem to Reg |
| R-format | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 |
| | X | 0 | 0 | 1 | 0 | 0 | 1 | 0 | X |
| | X | 0 | 1 | 0 | 1 | 0 | 0 | 0 | X |

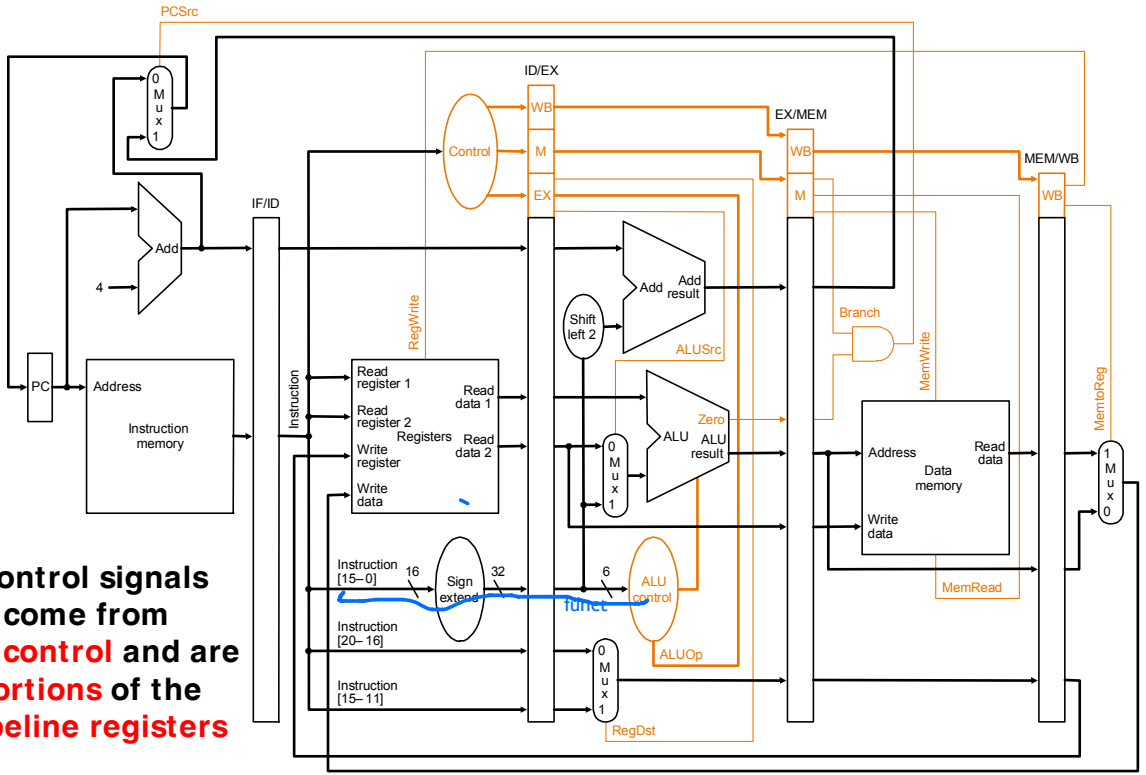
# Pipeline Control Implementation

- *Pass control signals along just like the data* – extend each pipeline register to hold needed control bits for succeeding stages
  控制訊號需繼續往下傳，下一個stage才能使用




- *Note*: The 6-bit *funct field* of the instruction required in the EX stage to generate ALU control can be retrieved as the 6 least significant bits of the immediate field which is sign-extended and passed from the IF/ID register to the ID/EX register

# Pipelined Datapath with Control – Step 3



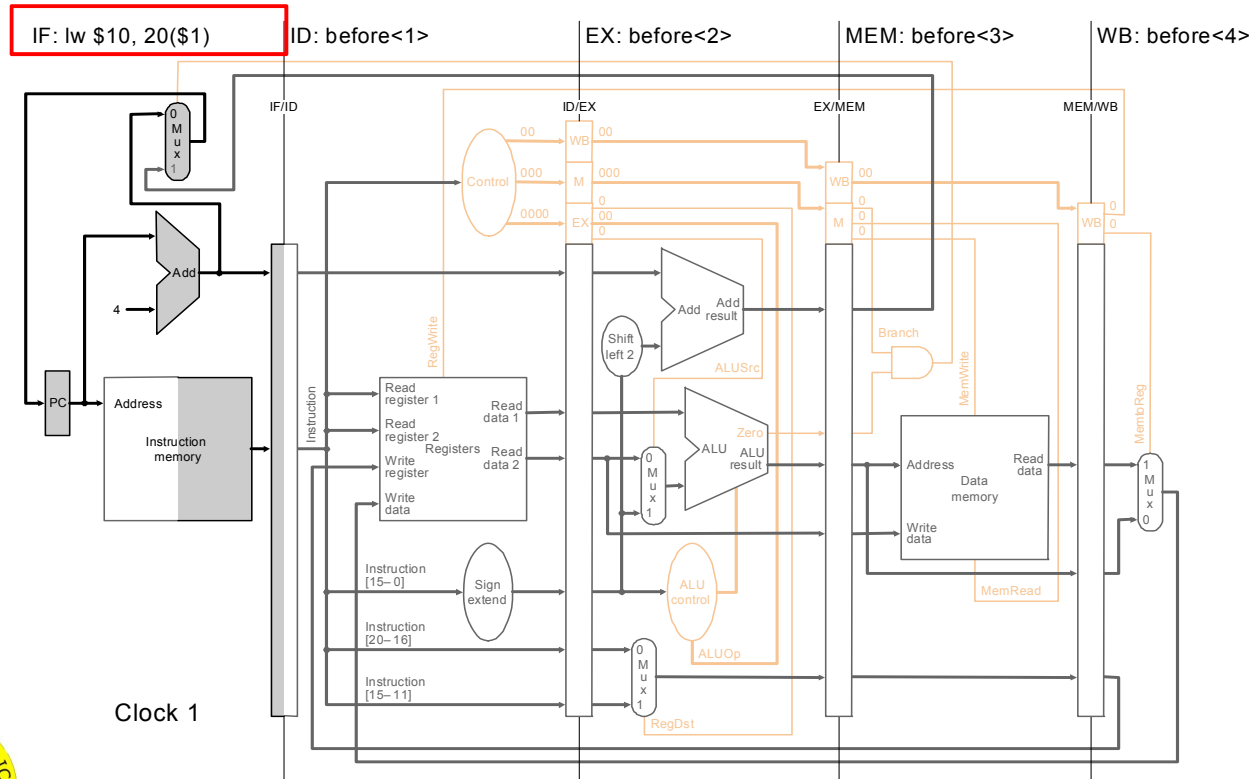

**Control signals come from the control and are portions of the pipeline registers**

# An Example

```
lw   $10, 20($1)
sub  $11, $2, $3
and  $12, $4, $5
or   $13, $6, $7
add  $14, $8, $9
```

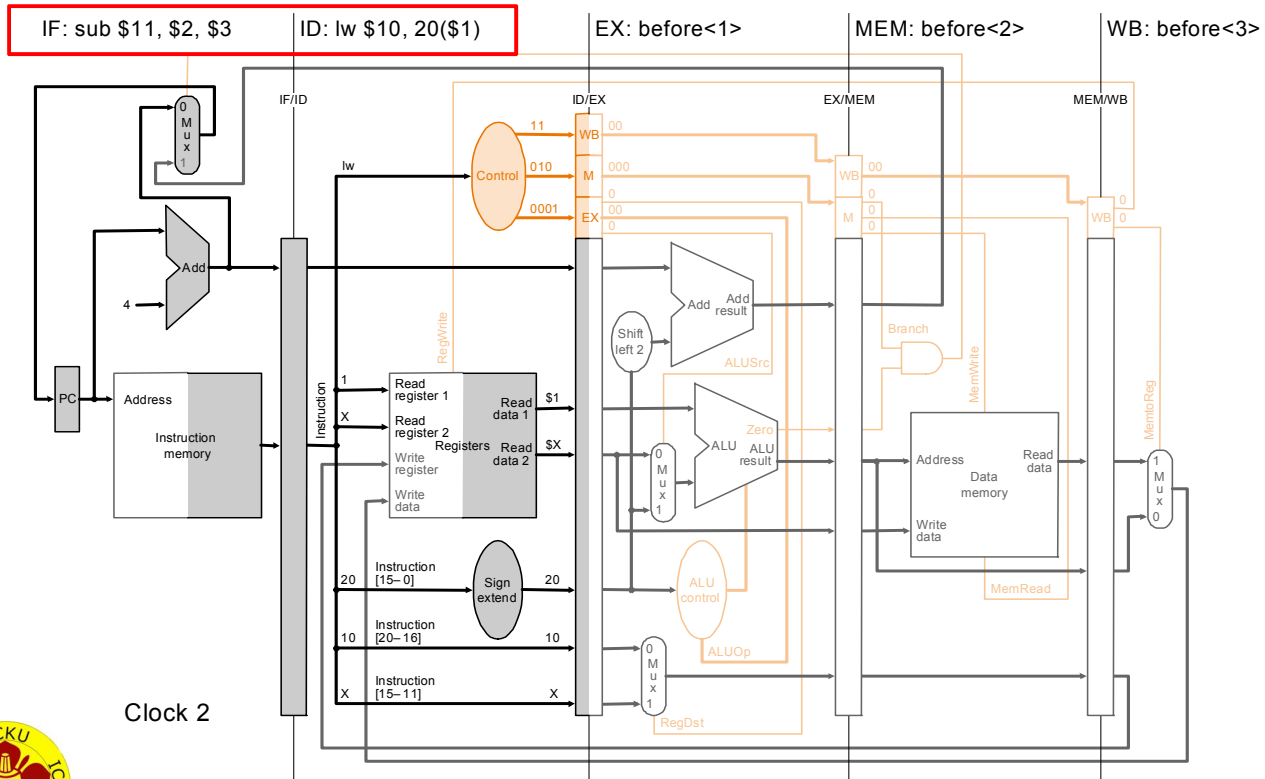
# Cycle 1

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```



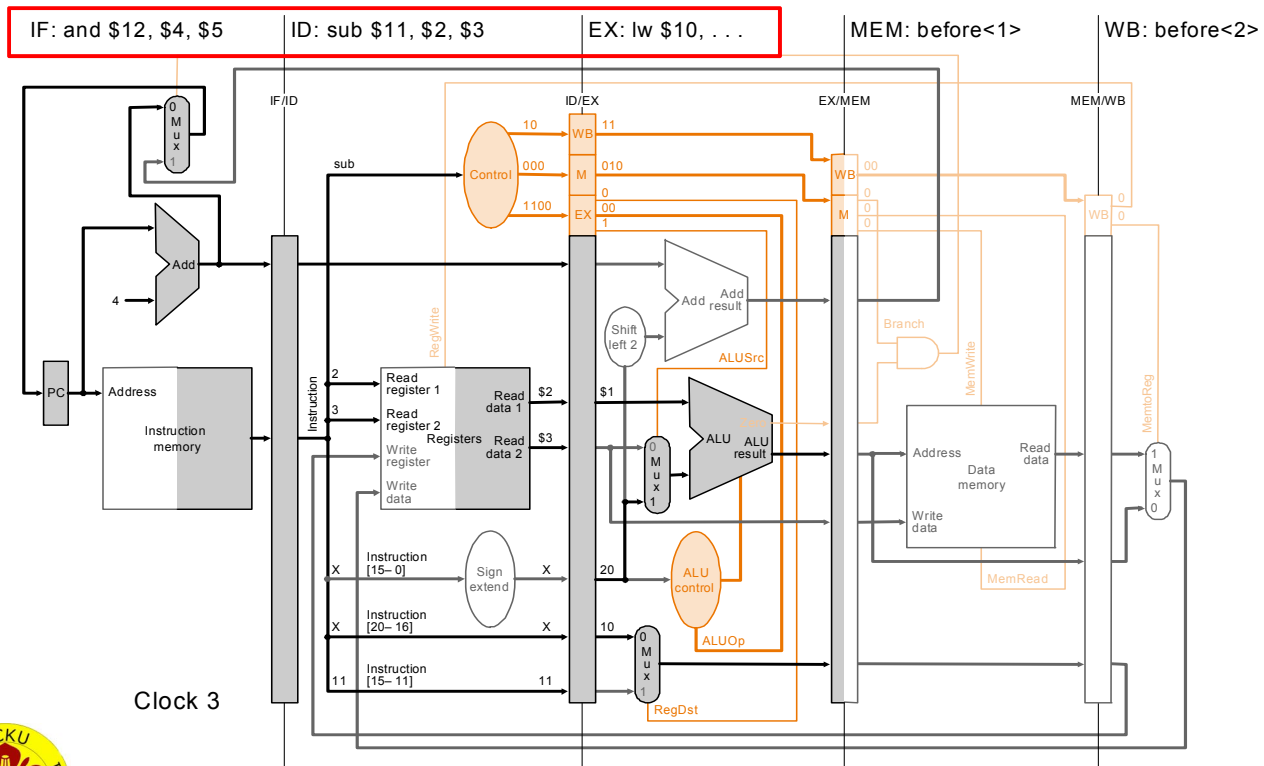
```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

# Cycle 2



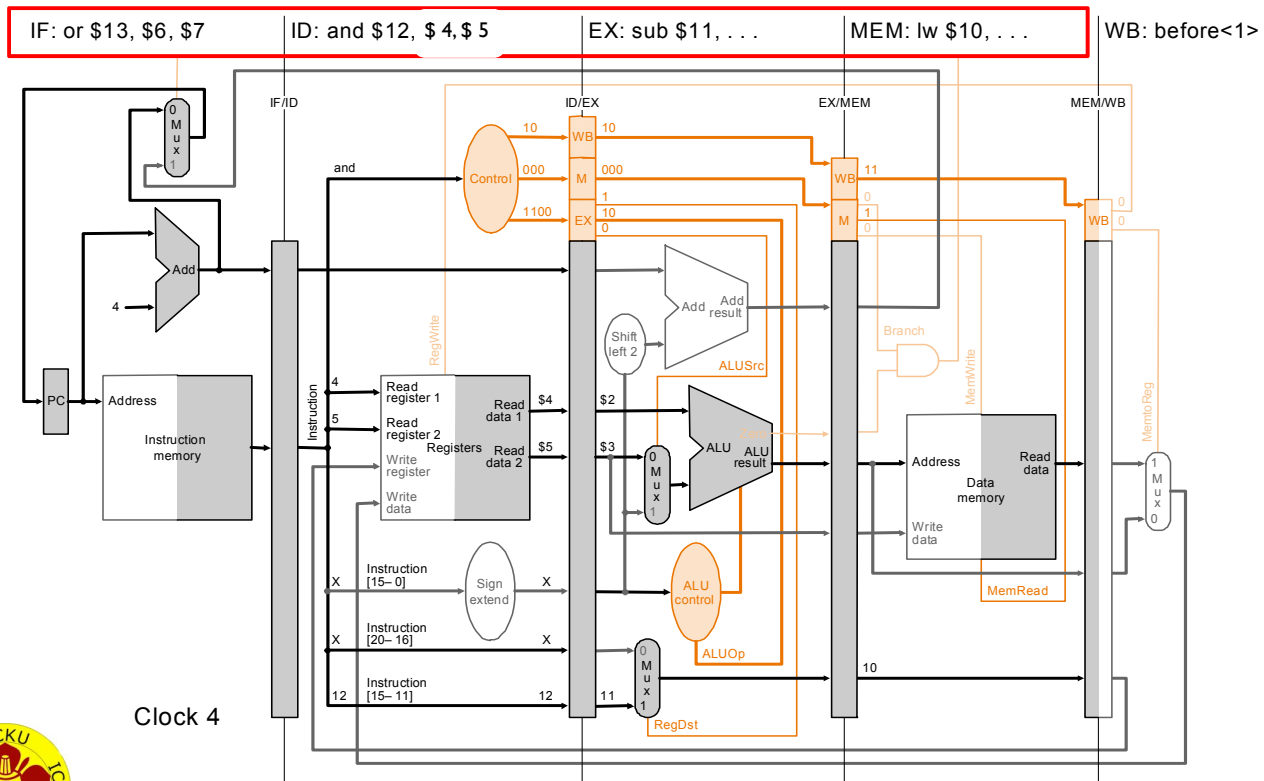
```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

# Cycle 3



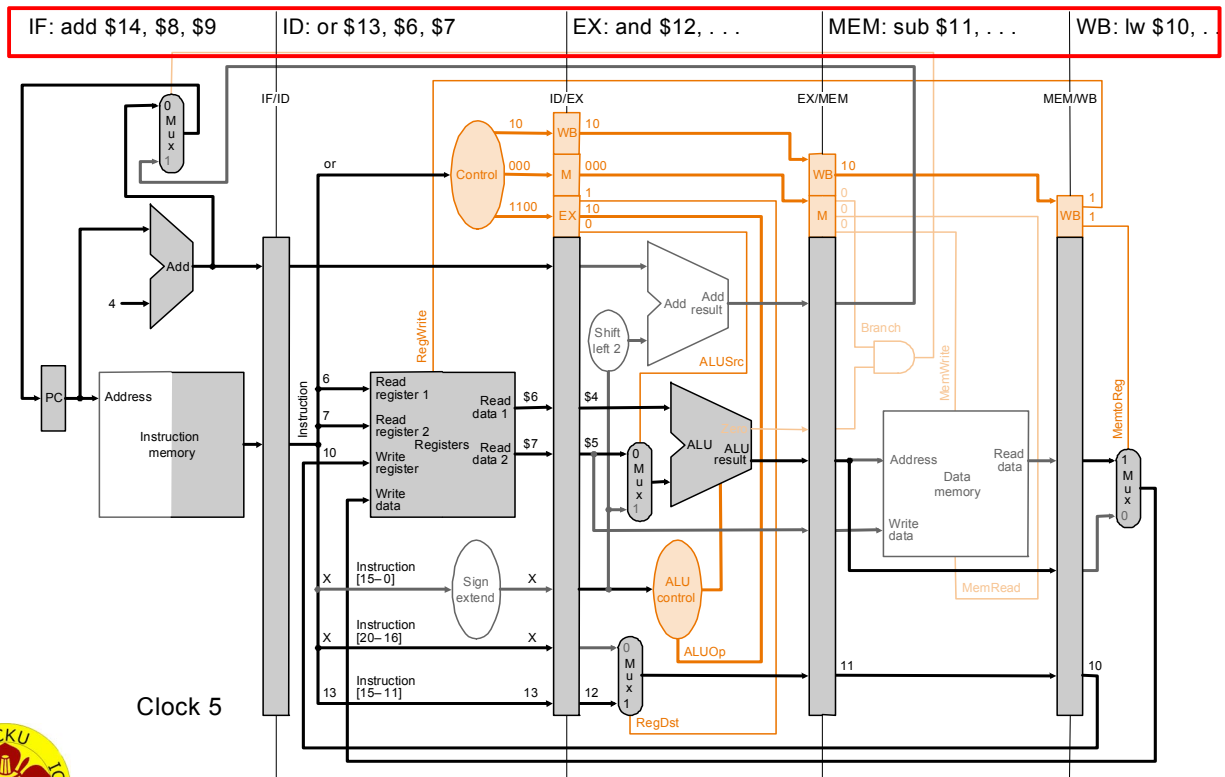
```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

# Cycle 4



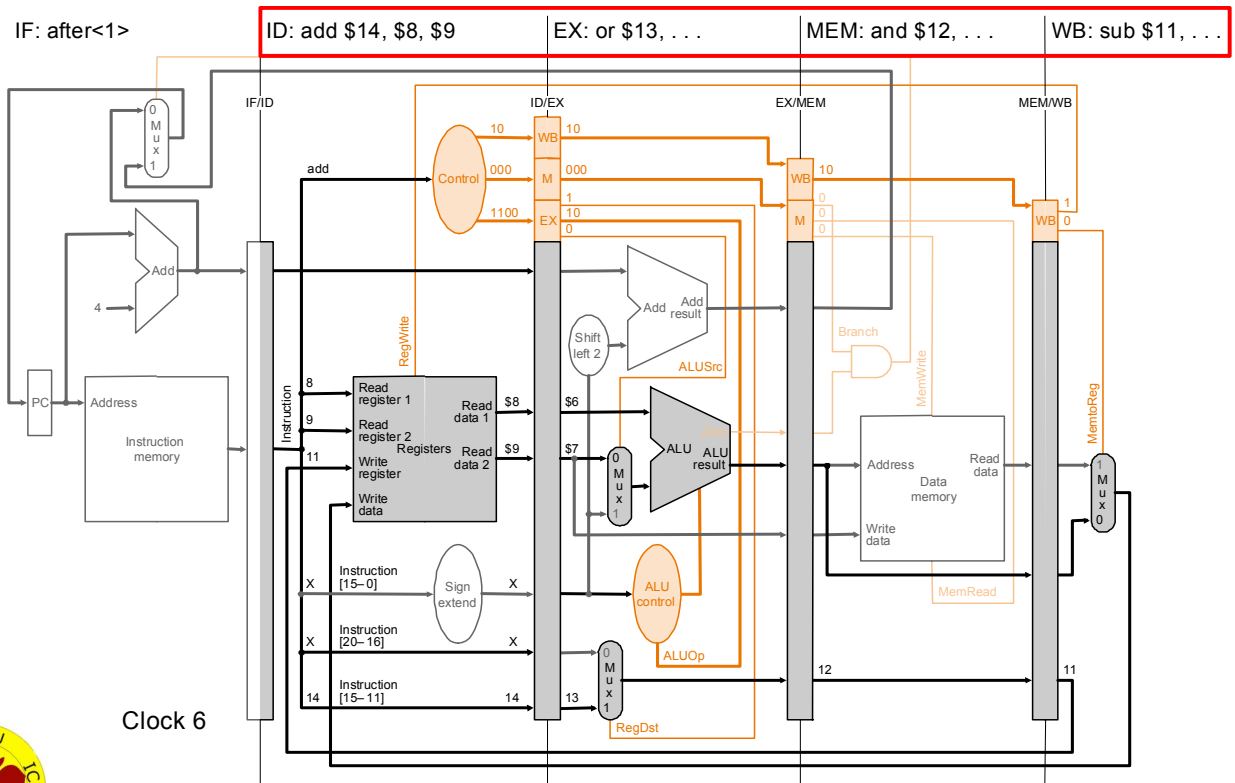
```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

# Cycle 5



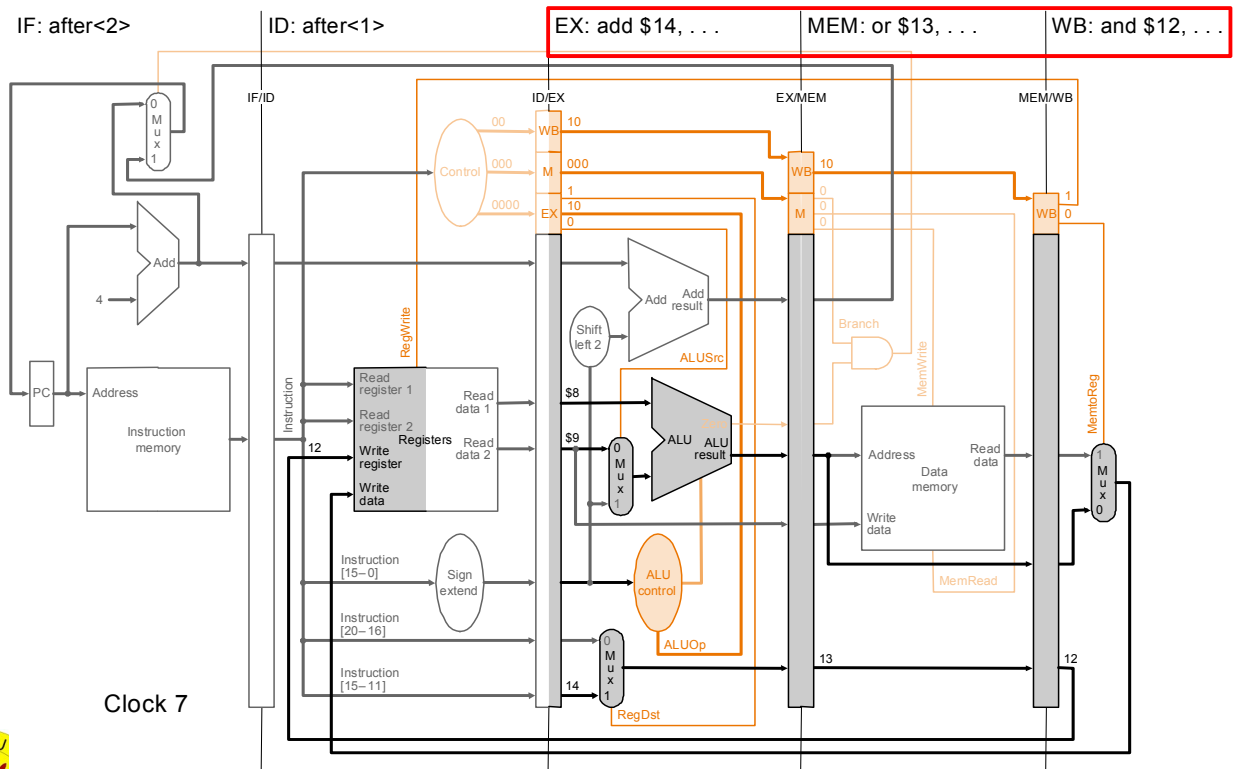
```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

# Cycle 6



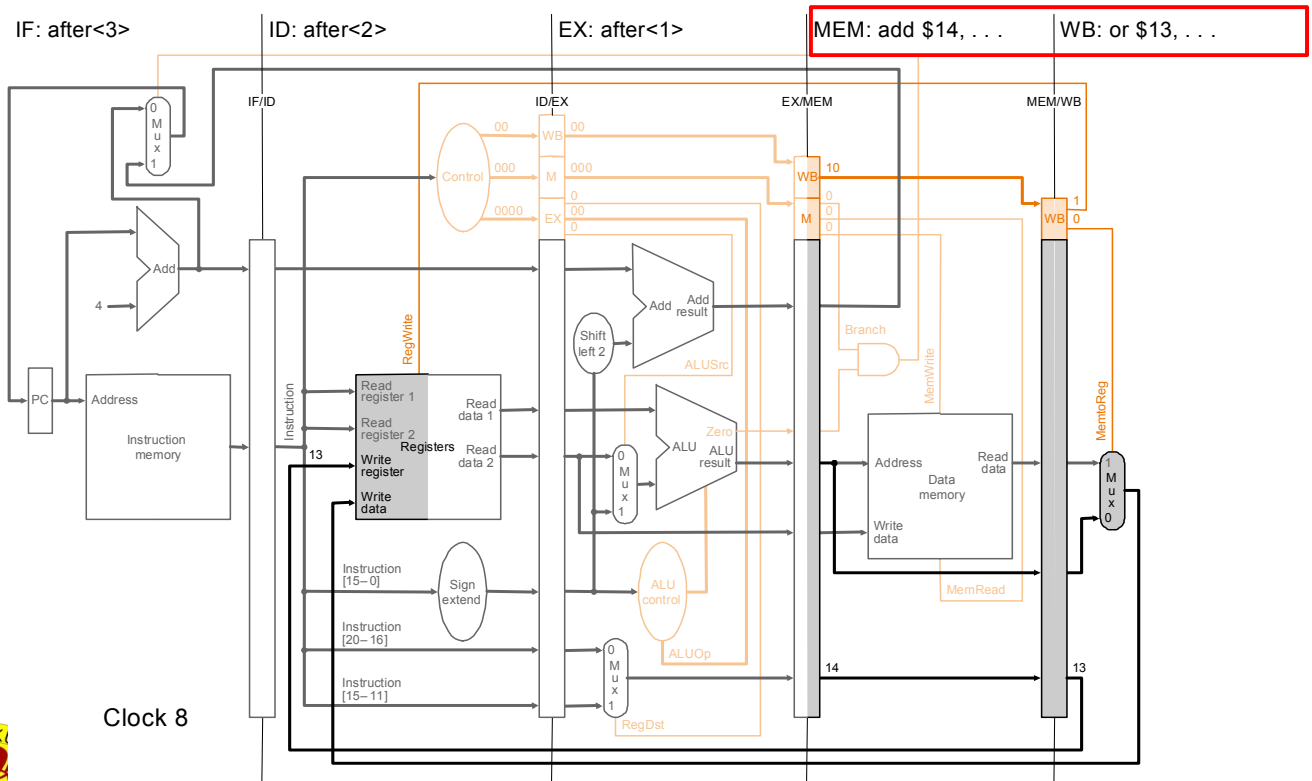
# Cycle 7

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```



```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

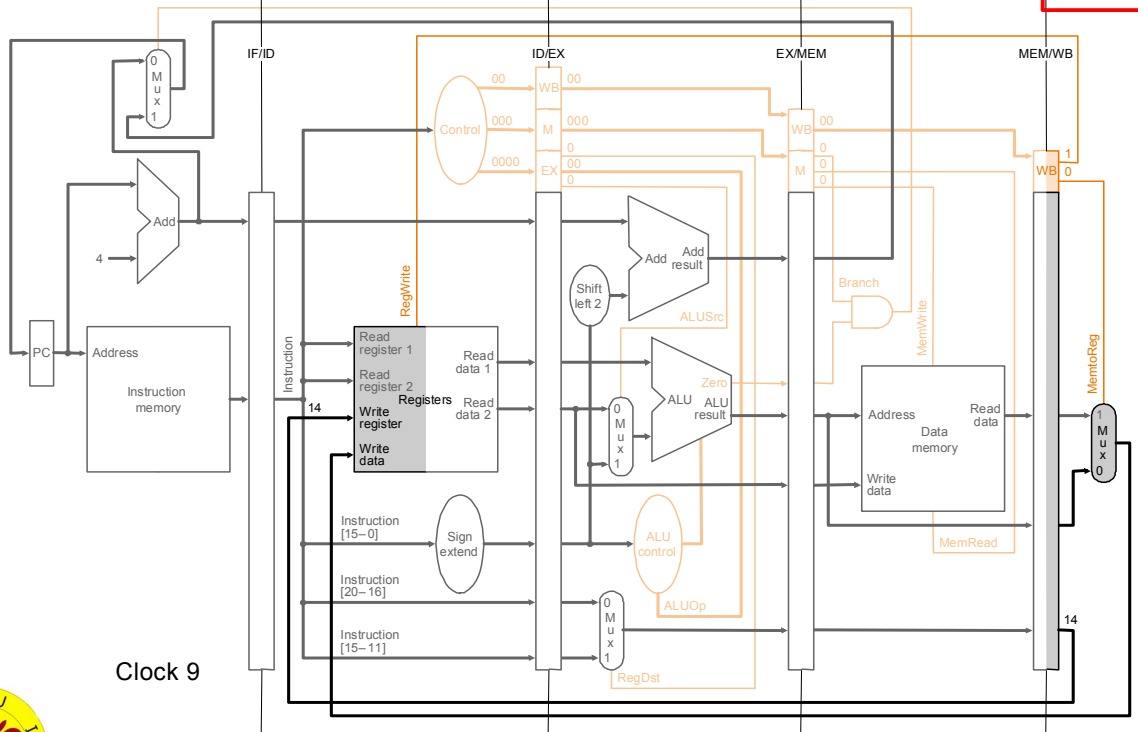
# Cycle 8



# Cycle 9

```
lw  $10, 20($1)
sub $11, $2, $3
and $12, $4, $5
or  $13, $6, $7
add $14, $8, $9
```

IF: after<4> | ID: after<3> | EX: after<2> | MEM: after<1> | WB: add $14, . . .



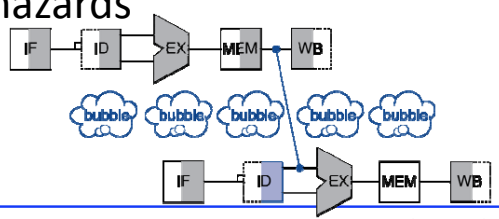

Clock 9

# Outline

- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards

# Pipeline Hazards

- Pipeline Hazards:
  - Structural hazards: attempt to use the same resource in two different ways at the same time
  - Data hazards: attempt to use data item before ready
    - Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: attempt to make decision before condition is evaluated
    - Branch instructions
- Can always resolve hazards by waiting (add nops /bubble)
  - pipeline control must detect the hazard
  - take action (or delay action) to resolve hazards



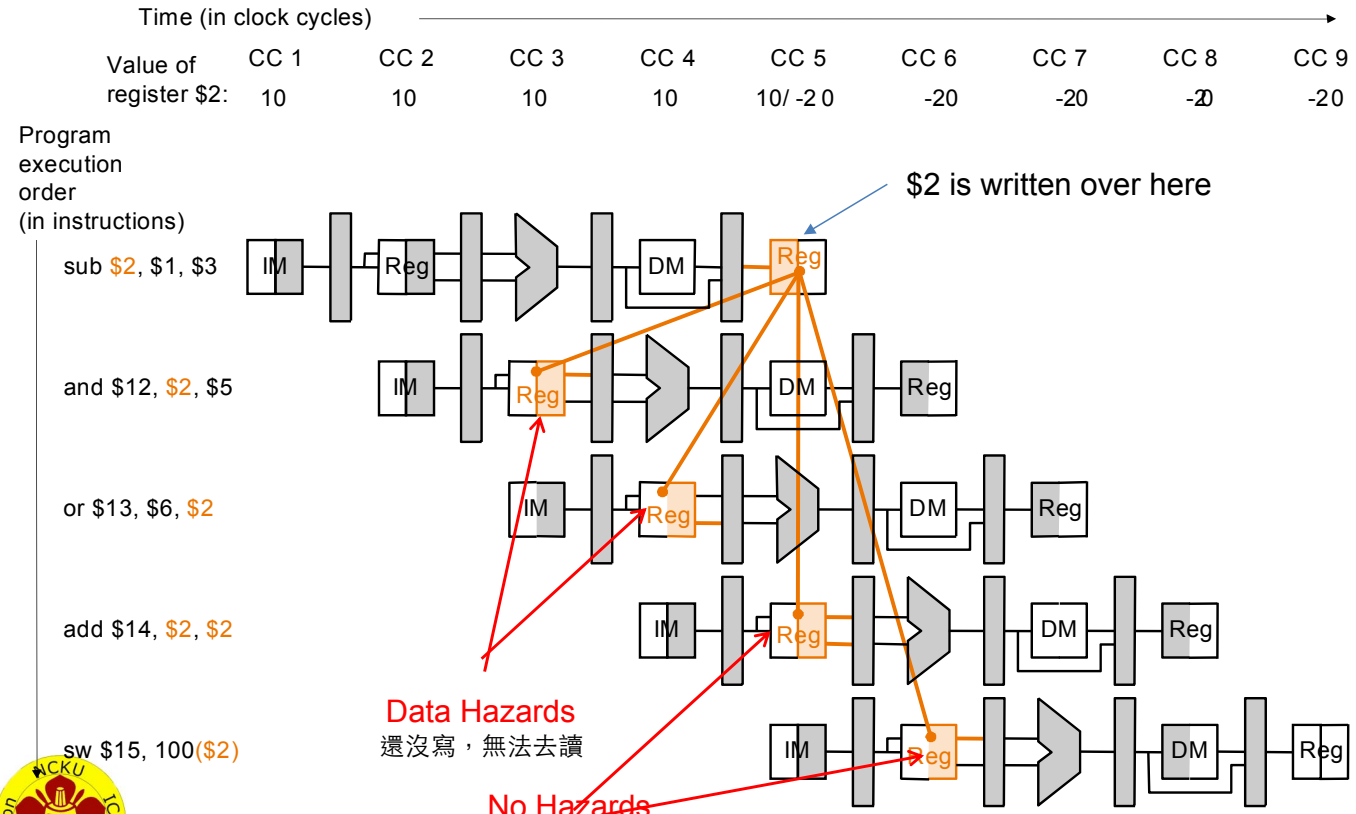
# Data Hazards in ALU instructions

- Consider this sequence

```
sub $2, $1, $3
and $12, $2, $5
or  $13, $6, $2
add $14, $2, $2
sw $15, 100, $2
```

- We can resolve hazards with forwarding
  - But how do we detect when to forward?

# Data Hazards

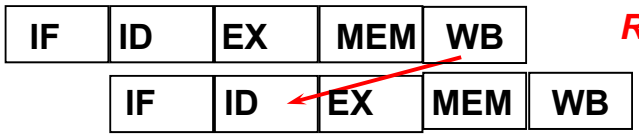


資料相依性

# Three Types of Data Dependency (RAW, WAR, WAW)

- RAW (read after write):
  i2 tries to read operand before i1 writes it

```
sub $2 $1 $3
add $4 $3 $2
```



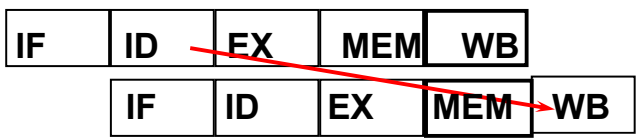

***RAW is Data Dependence also cause data hazard***

一定要寫完才能讀

- WAR (write after read):
  i2 tries to write operand before i1 reads it

```
add $4 $2 $3
sub $2 $1 $3
```

  – WAR is not a issue in MIPS 5-stage pipeline because all instructions take 5 stages, and reads are always in stage 2, and writes are always in stage 5, the following instruction never corrupt the previous instruction



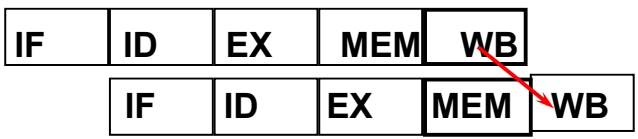

***Do not cause stall !!!***
***WAR is not data hazard***

# Types of Data Dependency (RAW, WAR, WAW)

Three types: (inst. i1 followed by inst. i2)

sub $2 $1 $3
sub $2 $1 $3

- WAW (write after write):
  i2 tries to write operand before i1 writes it
  - WAW is not an issue in MIPS 5-stage pipeline because all instructions take 5 stages, and writes are always in stage 5




***Do not cause stall !!!***
***WAW is not data hazard***

Quick summary: Data 間有相依性不一定會產生hazard
Three data dependency: RAW, WAR, WAR
only RAW may cause data hazard in MIPS

# Exercise

- Identify the data dependency (RAW, WAW, WAR) in the following instruction

```
lw $1,40($6)
add $6, $2, $2
sw $6, 50($1)
```

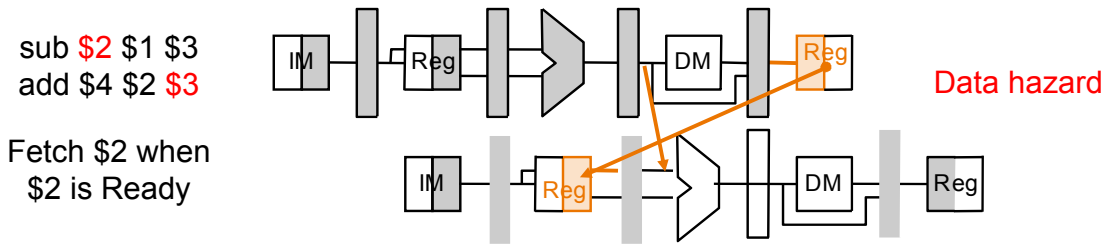
Ans:

```
lw $1,40($6)
      W       R
add $6, $2, $2
      W    R    R
sw $6, 50($1)
      R       R
```

I1 to I2: WAR on $6
I1 to I3: RAW on $1
I2 to I3: RAW on $6

# Hardware Solution: Forwarding

- Idea: fetch "fresh" data as early as possible




- Two steps:

Step 1: Detect data hazard:
Is the datum just produced required by the following inst.?
Step 2: Forward intermediate data to resolve hazard
If yes, then forward the requested datum to the requesting inst. immediately.

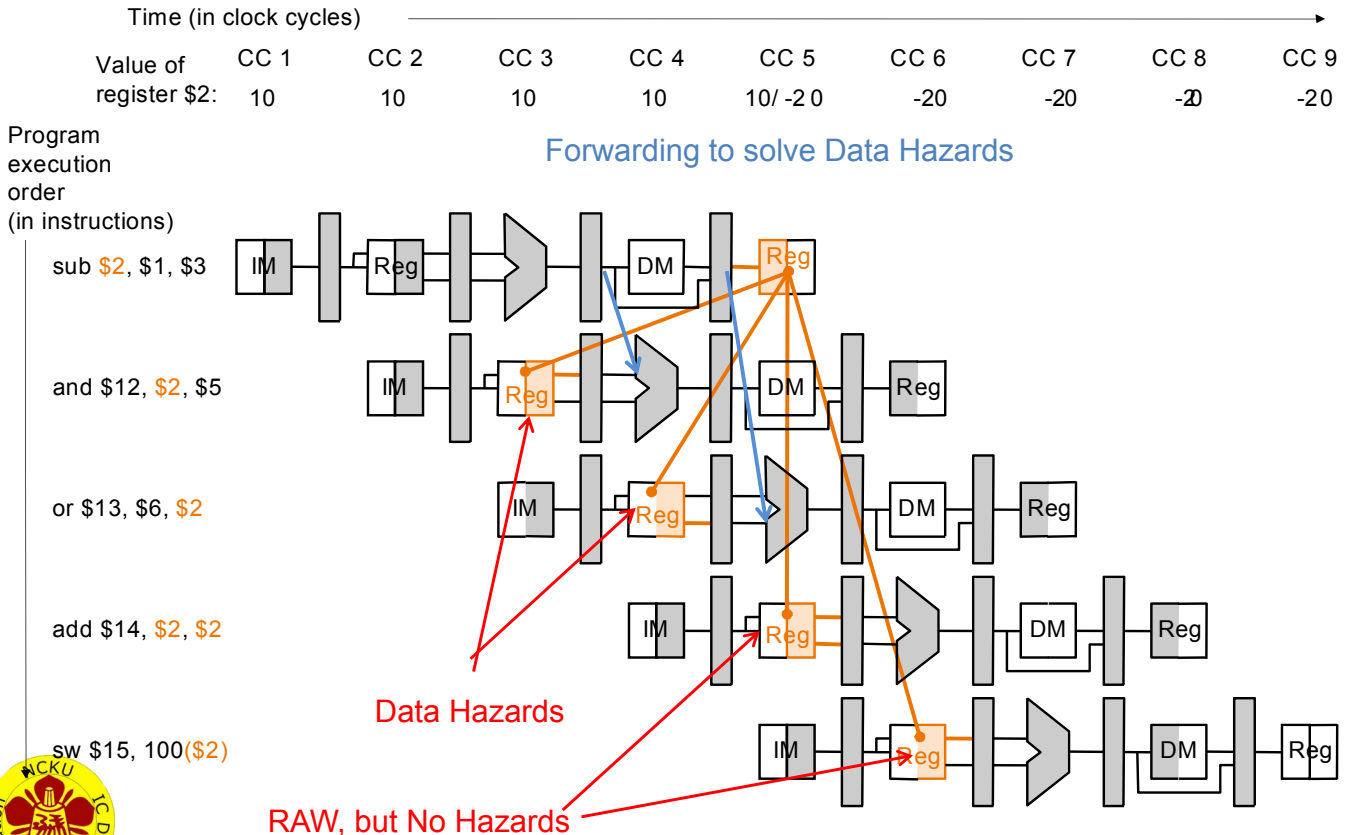
# Data Hazards in ALU Instructions

- Consider this sequence:

```
sub  $2,   $1, $3
and  $12, $2, $5
or   $13, $6, $2
add  $14, $2, $2
sw   $15, 100( $2)
```

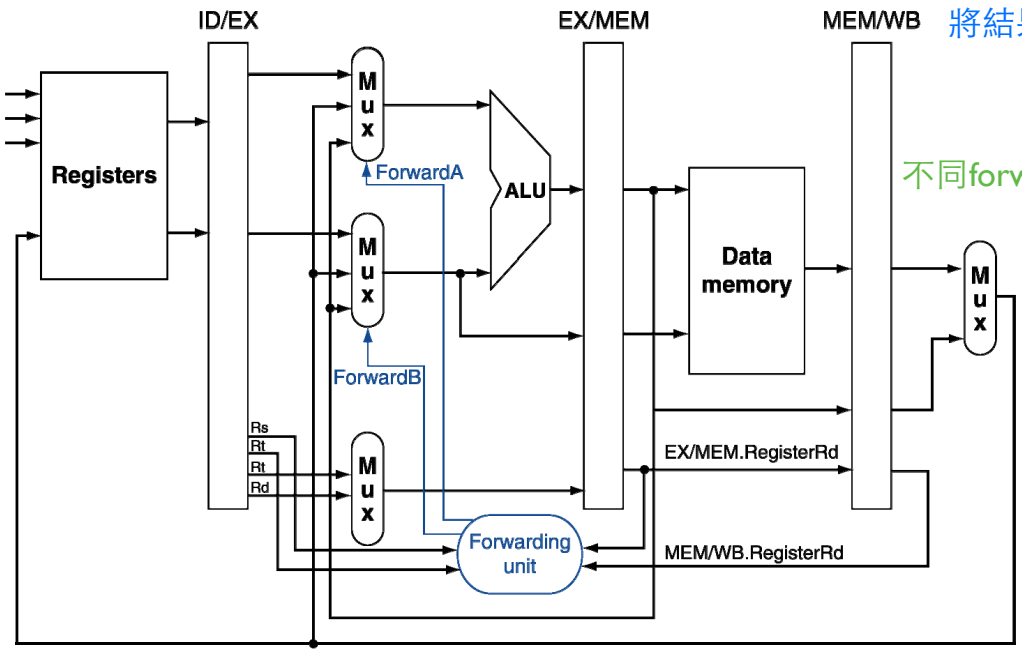
- We can resolve hazards with forwarding
  - How do we detect when to forward?

# Data Hazards



# Forwarding Hardware: Multiplexor Control

- Add Forwarding unit and ForwardA and ForwardB control signal to control mux (See next slides)



# Forwarding Hardware: Multiplexor Control

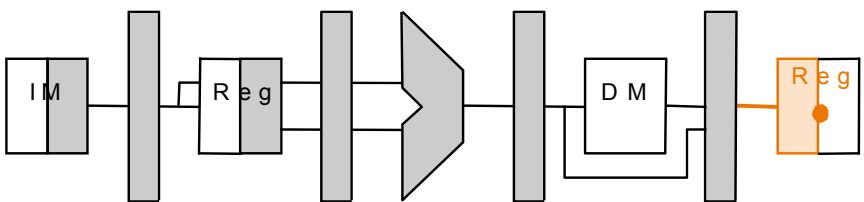
| Mux control | Source | Explanation |
|---|---|---|
| ForwardA = 00 | ID/EX | The first ALU operand (Rs) comes from the register file |
| ForwardA = 10 | EX/MEM | The first ALU operand is forwarded from prior ALU result |
| ForwardA = 01 | MEM/WB | * The first ALU operand is forwarded from data memory or an earlier ALU result |

| | | |
|---|---|---|
| ForwardB = 00 | ID/EX | The second ALU operand (Rt) comes from the register file |
| ForwardB = 10 | EX/MEM | The second ALU operand is forwarded from prior ALU result |
| ForwardB = 01 | MEM/WB | * The second ALU operand is forwarded from data memory or an earlier ALU result |

* Depending on the selection in the rightmost multiplexor (see datapath with control diagram)

# Detecting the Need to Forward

- Pass register numbers along pipeline
  - e.g., ID/EX.RegisterRs = register number for Rs sitting in ID/EX pipeline register
- E.g.: ALU operand register numbers in EX stage are given by
  - ID/EX.RegisterRs, ID/EX.RegisterRt




sub $2 $1 $3

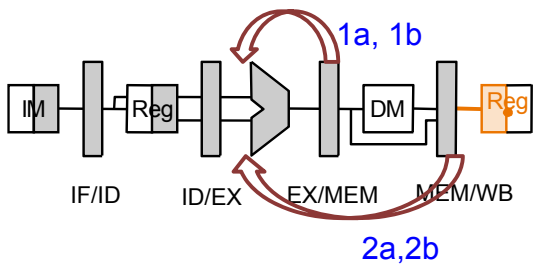
At the first cycle,
IF/ID.RegisterRs=$1
IF/ID.RegisterRt=$3
IF/ID.RegisterRd=$2

# Detecting Data hazard

- Data hazards when
  - 1a. EX/MEM.RegisterRd = ID/EX.RegisterRs
  - 1b. EX/MEM.RegisterRd = ID/EX.RegisterRt
  - 2a. MEM/WB.RegisterRd = ID/EX.RegisterRs
  - 2b. MEM/WB.RegisterRd = ID/EX.RegisterRt

Fwd from EX/MEM pipeline reg

Fwd from MEM/WB pipeline reg




```
sub $2, $1, $3
and $12, $2, $5
```

Hazard is detected when the and is in EX stage and the sub is in MEM stage because destination 和 source相同就需forward

EX/MEM.RegisterRd = ID/EX.RegisterRs = $2 (1a)

# Detecting EX hazard

- Forwarding is needed only if earlier instruction will write to a register!
  - Check if EX/MEM.RegWrite, MEM/WB.RegWrite is 1

```
sub $2, $1, $3
and $12, $2, $5
```

- And only if Rd for that instruction is not $zero
  - When Rd=$zero, result is always zero
  - Check if EX/MEM.RegisterRd ≠ 0,MEM/WB.RegisterRd ≠ 0

```
sub $0, $1, $3
and $12, $0, $5
```

RD是0不需forward

Quick Summary

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 (MEM/WB.RegisterRd=ID/EX.RegisterRs)) ForwardA =01

If(MEM/WB.RegWrite and (MEM/WB.RegisterRd !=0)
and (MEM/WB.RegisterRd=ID/EX.RegisterRt)) ForwardB=01

# Additional rule for MEM hazard

– if the later instruction is going to write the same register, even if there is register number match as in conditions above

```
Inst1   add $7, $7, $9
Inst 2  add $7, $7, $10
Inst 3  add $7, $7, $11
```

**Inst 1 does not need to forward to inst 3 because inst2 has more recent data**

– if the destination register of the later instruction is $0 – in which case there is no need to forward value ($0 is always 0 and never overwritten)

```
sub     $2, $1, $3
and     $0, $2, $5
```

**Inst 1 does not need to forward to inst2 because inst2 always produce 0 ($zero)**

destination為0不需forward

Inst1 If(MEM/WB.RegWrite and (MEM/WB.RegisterRd !=0)
Inst2 and not (EX/MEM.RegWrite and (EX/MEM.RegisterRd != 0)
Inst3 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 (EX/MEM.RegisterRd != ID/EX.RegisterRt)
and (MEM/WB.RegisterRd=ID/EX.RegisterRt)) ForwardB=01

# Double Data Hazard

- Consider the sequence:

```
add  $1, $1, $2
add  $1, $1, $3
add  $1, $1, $4
```

Wrong

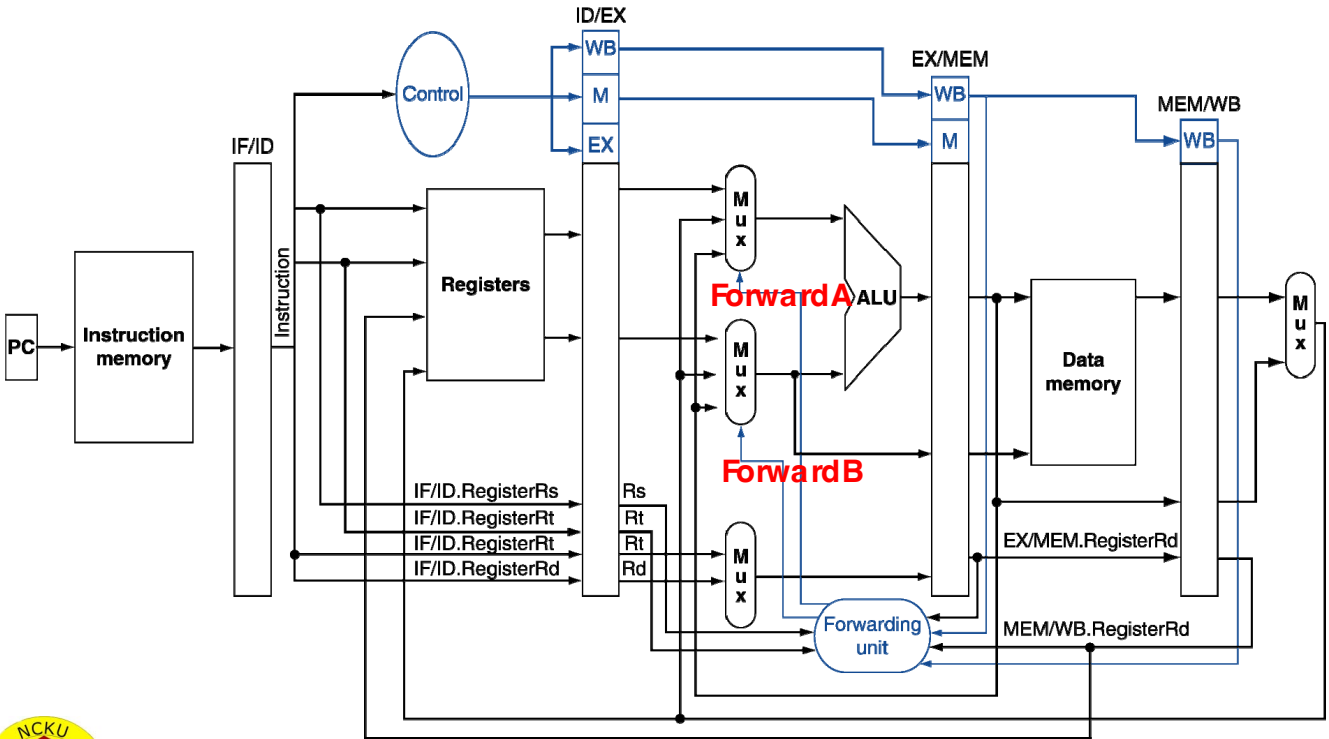
```
add  $1, $1, $2
add  $1, $1, $3
add  $1, $1, $4
```

- Both hazards occur
  - Want to use the most recent
- Revise MEM hazard condition
  - Only forward if no EX hazard condition exists (i.e. no data between second and third instruction)

# Revised Forwarding Condition

- 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

# Datapath with Forwarding



# Outline

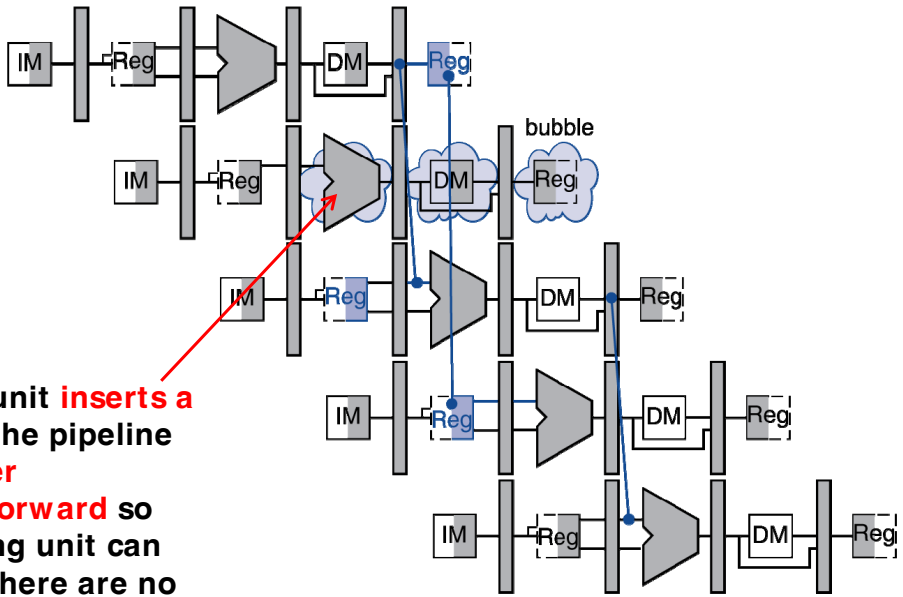
- A pipelined datapath
- Pipelined control
- Data hazards and forwarding
- Data hazards and stalls
- Branch hazards

# Stalling Resolves a Hazard

- Same instruction sequence as before for which forwarding by itself could not resolve the hazard:

```
lw  $2, 20($1)
and $4, $2, $5
or  $8, $2, $6
add $9, $4, $2
Slt $1, $6, $7
```

lw需等待一個cycle
才能寫入register



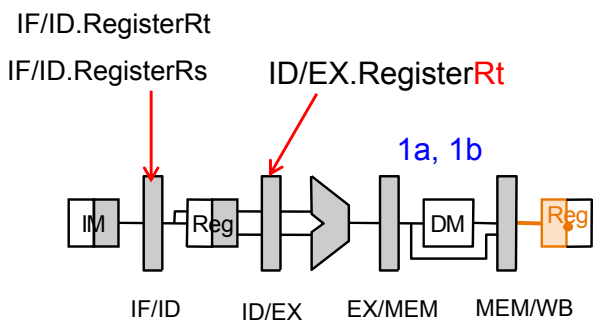

- **Hazard detection unit inserts a 1-cycle bubble in the pipeline**
- **All pipeline register dependencies go forward so then the forwarding unit can handle them and there are no more hazards**

# Hazard Detection Logic to Stall

- Recall: lw instruction format: `lw  Rt, offset(Rs)`
- Hazard detection unit implements the following check if to stall

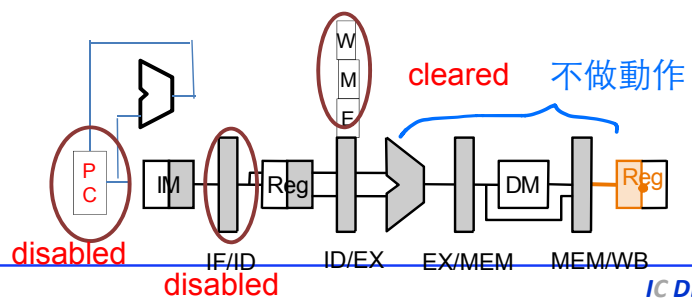
Destination Register in lw instruction.

if ( ID/EX.MemRead // if the instruction in the EX stage is a load…
and ( ( ID/EX.RegisterRt = IF/ID.RegisterRs ) // and the destination register
or ( ID/EX.RegisterRt = IF/ID.RegisterRt ) ) ) // matches either source register
// of the instruction in the ID stage, then…
stall the pipeline( insert a bubble )



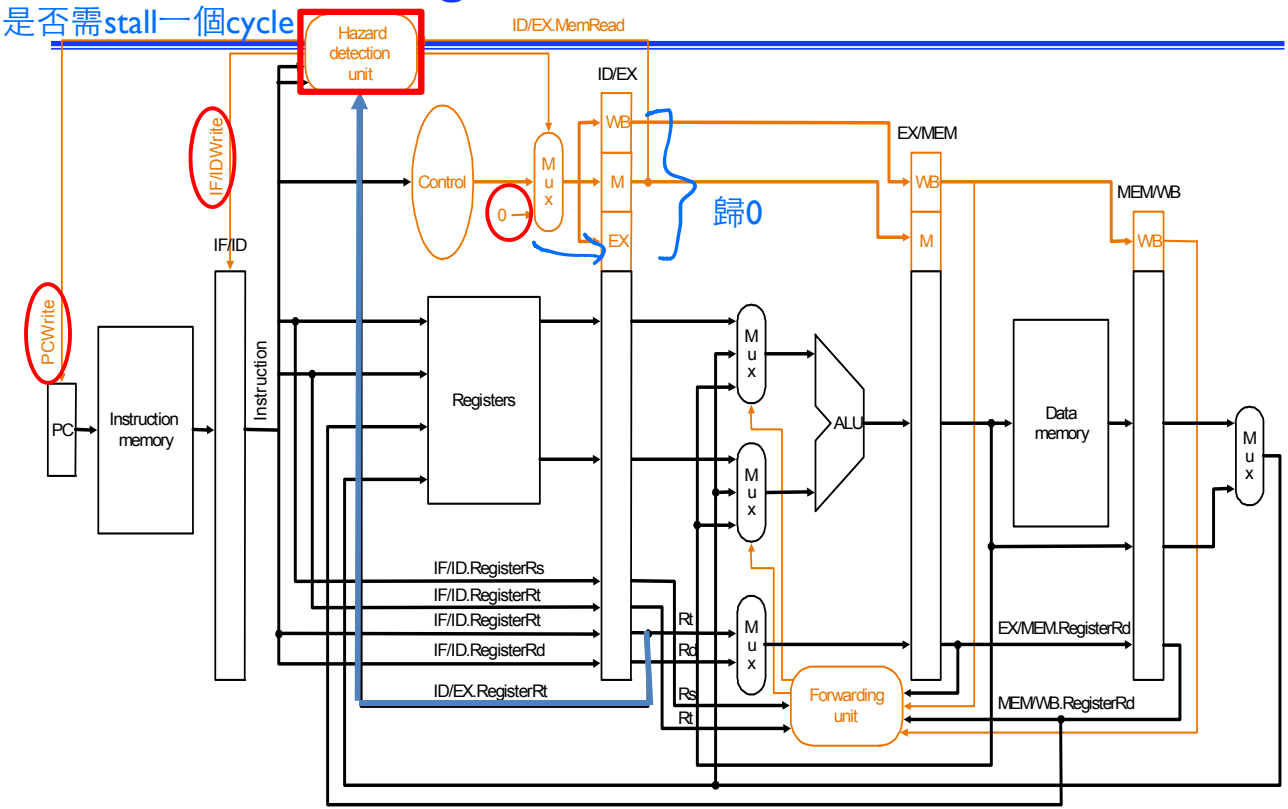
# How to Stall the Pipeline

- If the conditions to stall is valid, then the *pipeline needs to stall only 1 clock cycle* after the load as after that the forwarding unit can resolve the dependency
- How the hardware stalls the pipeline 1 cycle:
  - disable write on IF/ID register => this will cause the instruction in the ID stage to repeat, i.e., *stall*
  - disable write on PC => this will cause the instruction in the IF stage to repeat, i.e., *stall* 不讀下一指令
  - changes all the EX, MEM and WB control fields in the ID/EX pipeline register to 0 => , so the instruction just behind the load becomes a nop – a bubble is said to have been inserted into the pipeline



# Adding Hazard Detection Unit

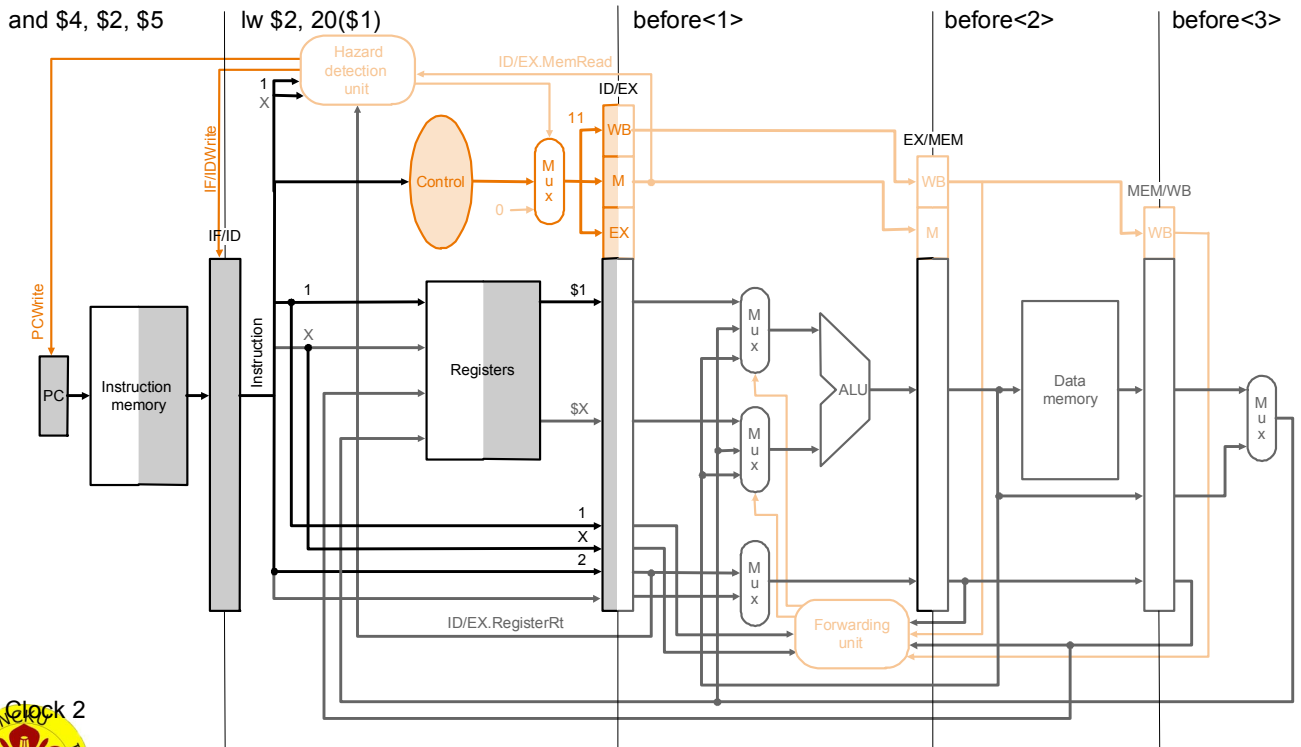
判斷是否需stall一個cycle




**Datapath with forwarding hardware, the hazard detection unit and controls wires – certain details, e.g., branching hardware are omitted to simplify the drawing**

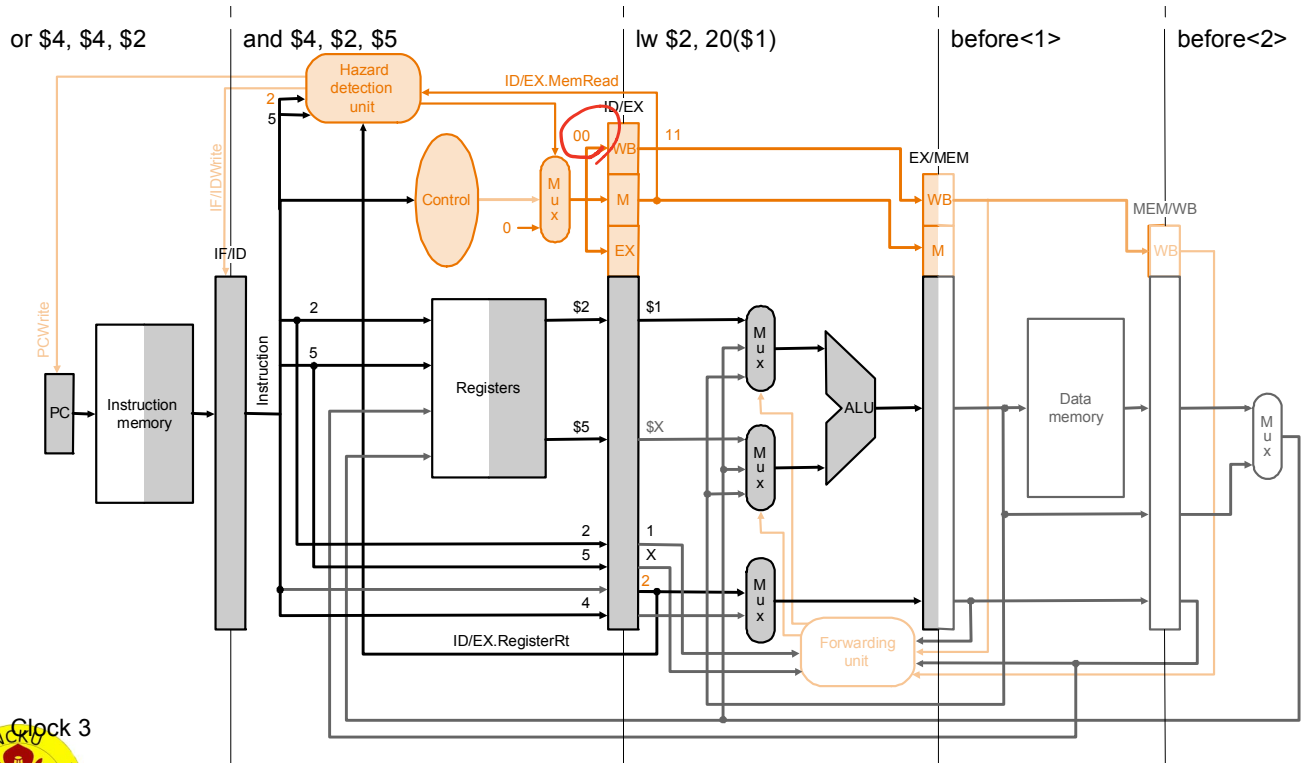
# Cycle 2

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```



```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

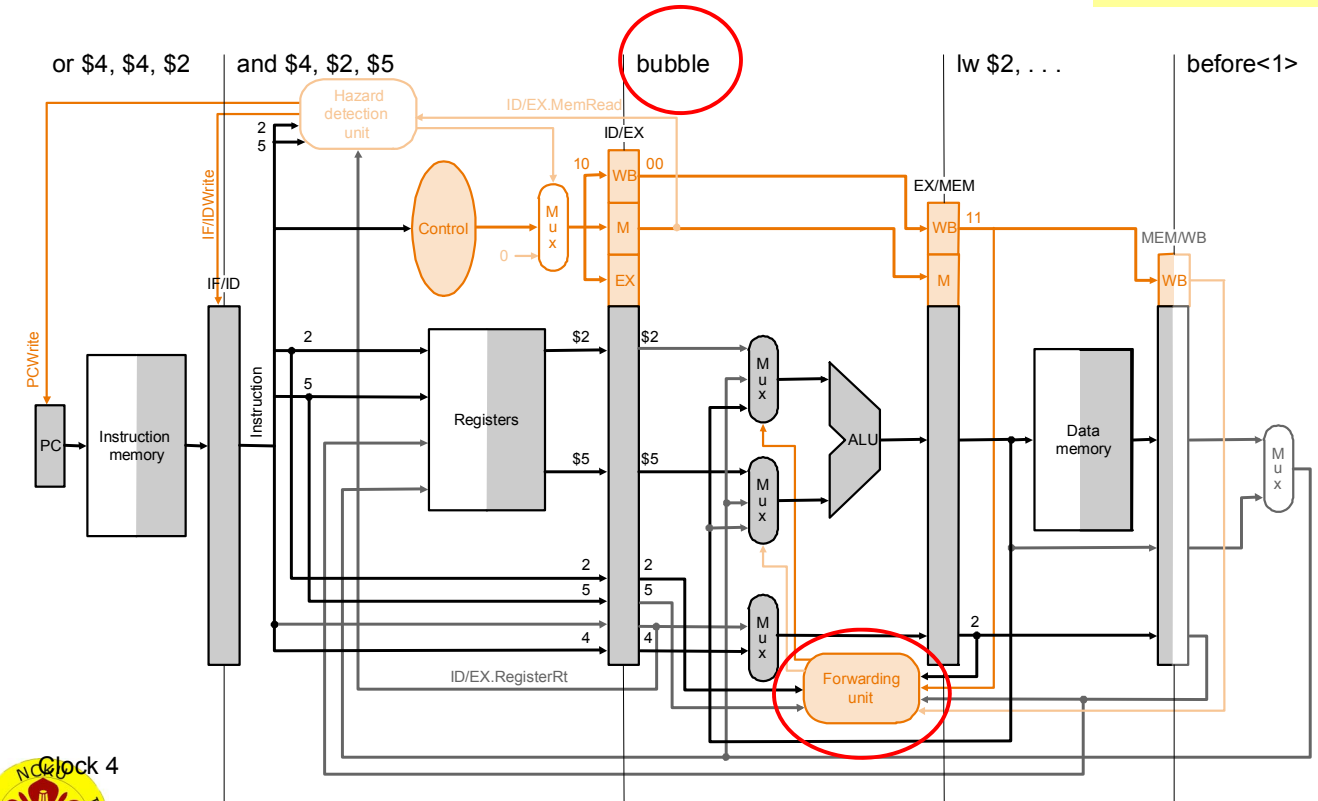
# Cycle 3




Clock 3

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

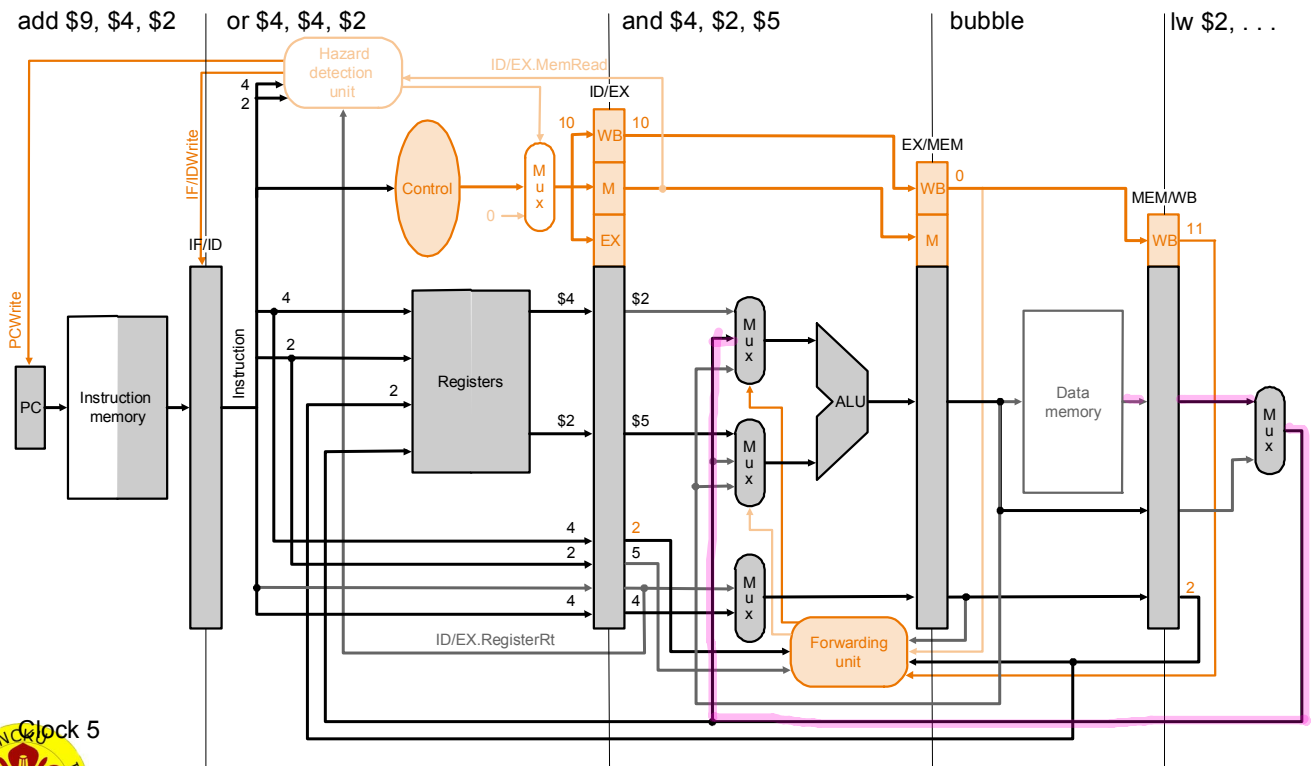
# Cycle 4




Clock 4

# Cycle 5

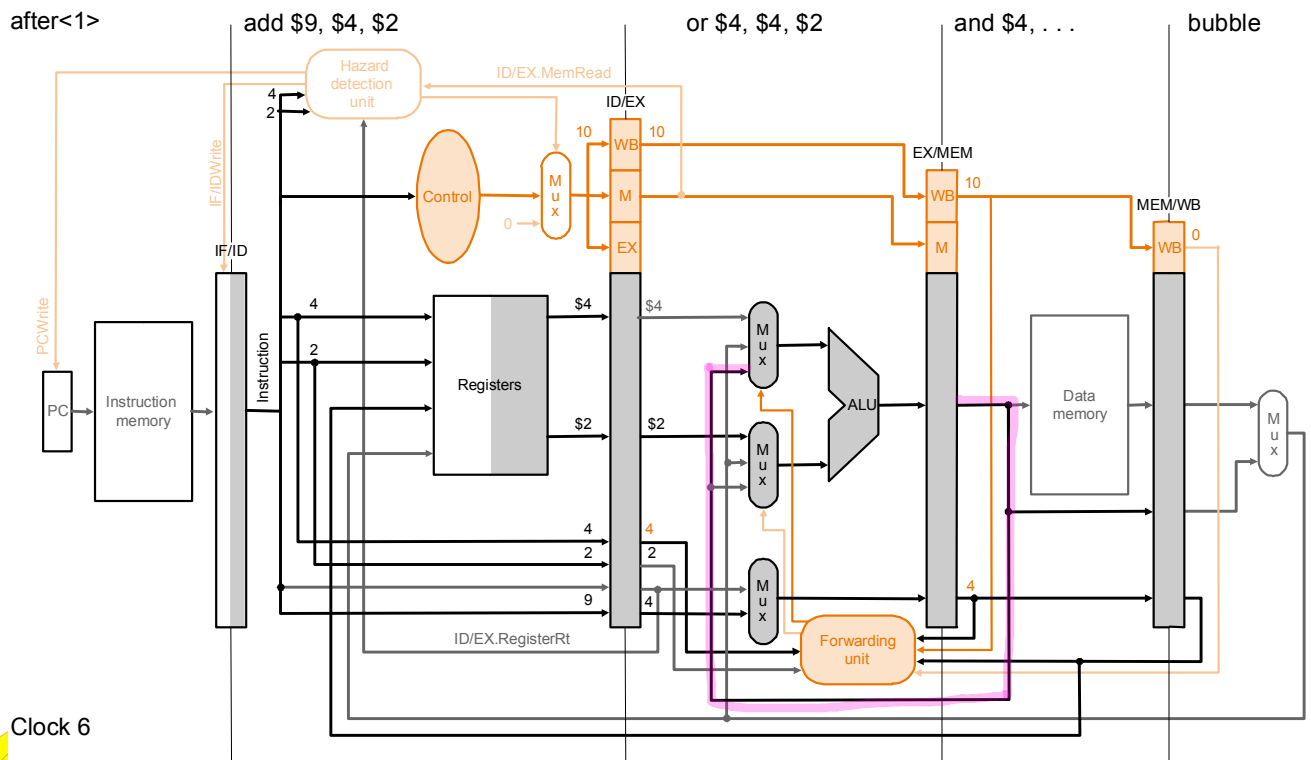
```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```




Clock 5

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

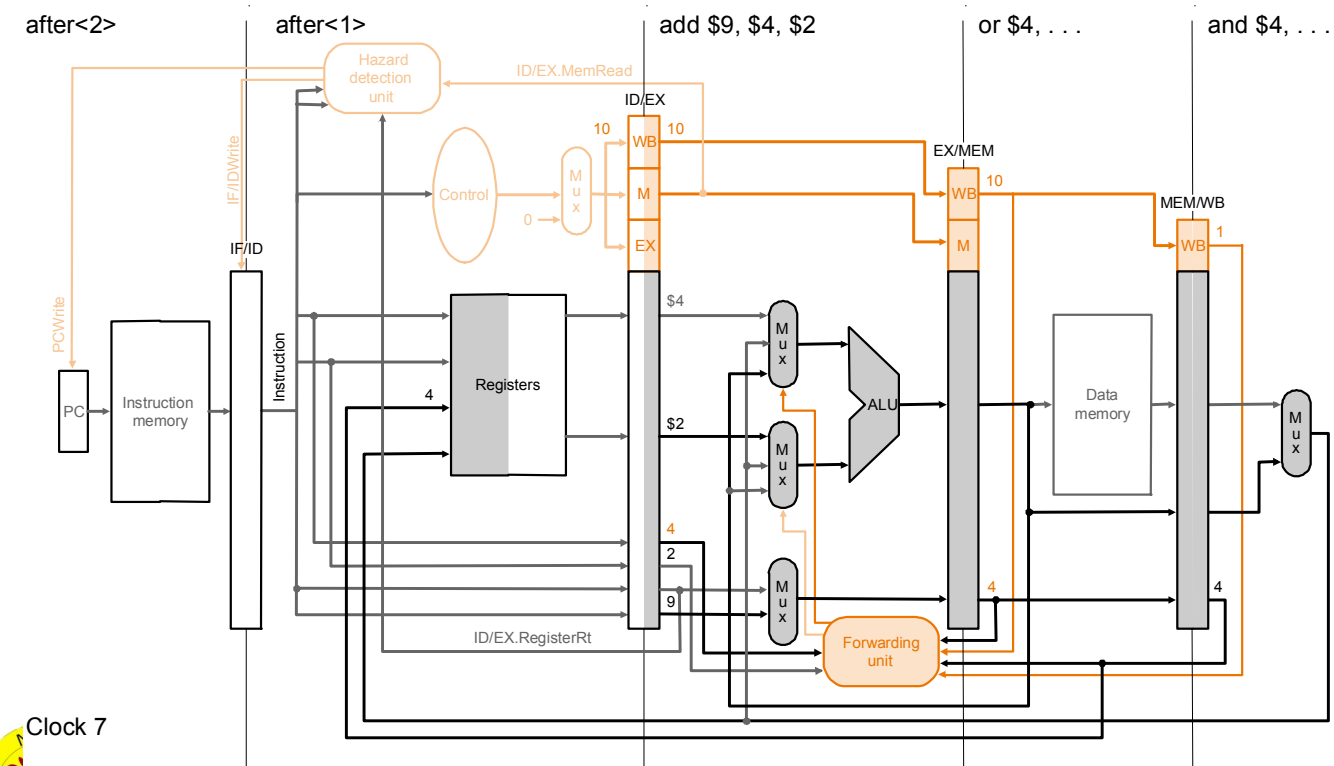
# Cycle 6




Clock 6

```
lw  $2, 20($1)
and $4, $2, $5
or  $4, $4, $2
add $9, $4, $2
```

# Cycle 7




Clock 7

# Backup Slides