4.4 GENERAL PRINCIPLES OF PIPELINING

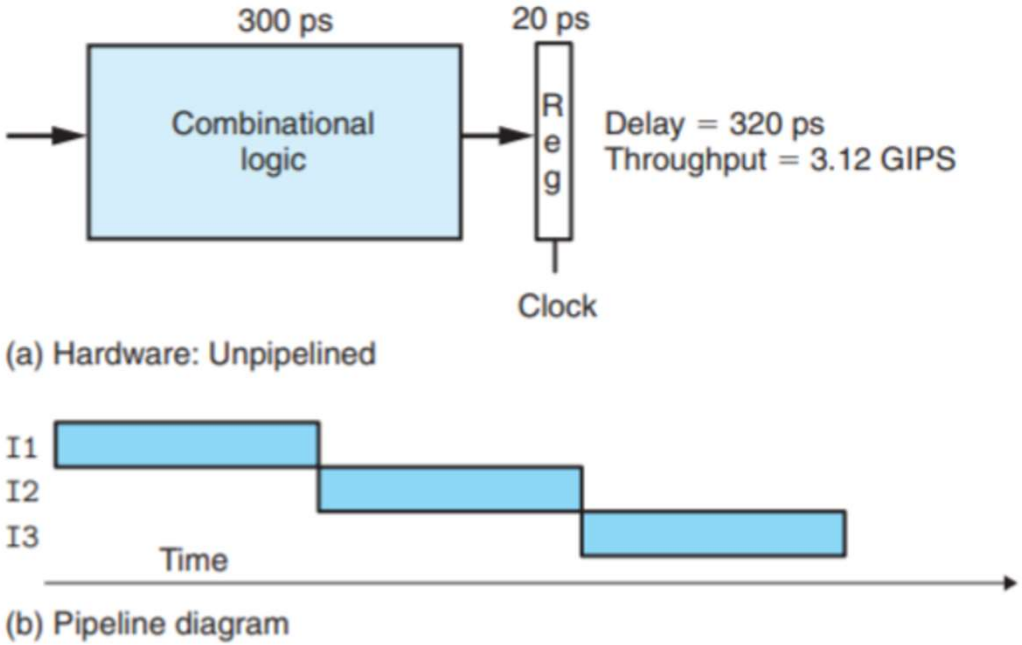
- SEQ(sequential processor) 太慢
- 將硬體分成很多管線，藉此同時執行很多指令
- 例子只看計算部分

評估方式

- latency
    - 執行一個指令的時間
    - picoseconds (ps)

- throughput
    - 單位時間內能處理多少指令
    - giga-instructions per second(GIPS)

NONPIPELINE



計算方式

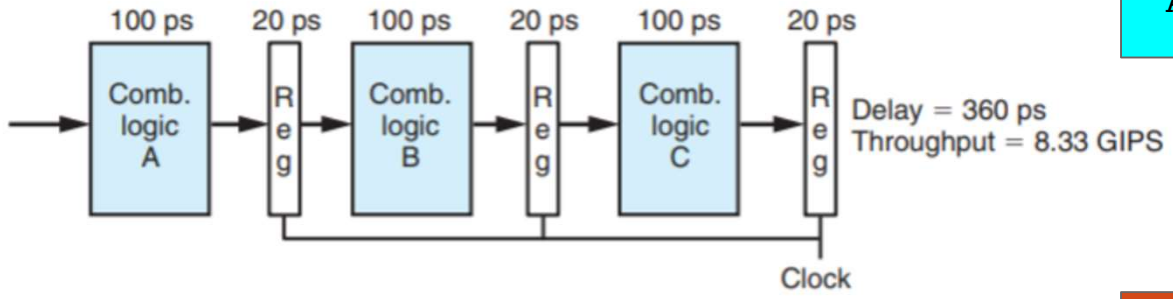
$$Throughput = \frac{1\ \text{instruction}}{(20 + 300)\ \text{picoseconds}} \cdot \frac{1{,}000\ \text{picoseconds}}{1\ \text{nanosecond}} \approx 3.12\ \text{GIPS}$$

r: pipeline register 的 讀取時間

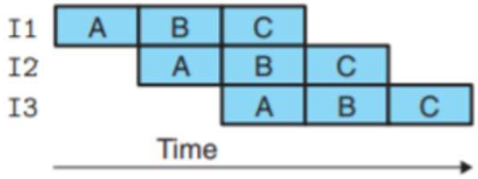
x: 細分後 comb logic 的執行時間

$$\text{Thoughput} = \frac{1\ instruction}{(r+x)\ ps} * \frac{1000\ ps}{1\ ns} \quad \text{GIPS}$$

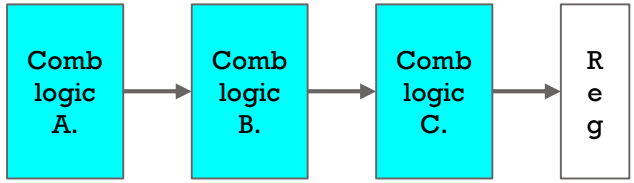
PIPELINE

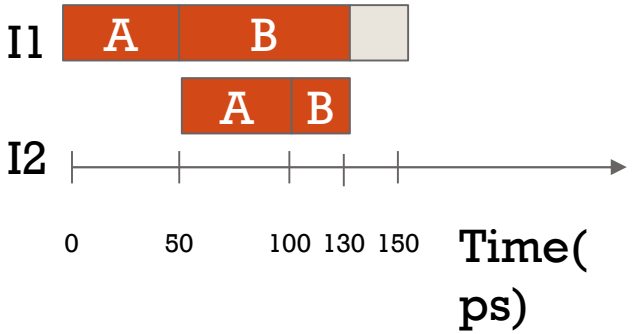


(a) Hardware: Three-stage pipeline



(b) Pipeline diagram

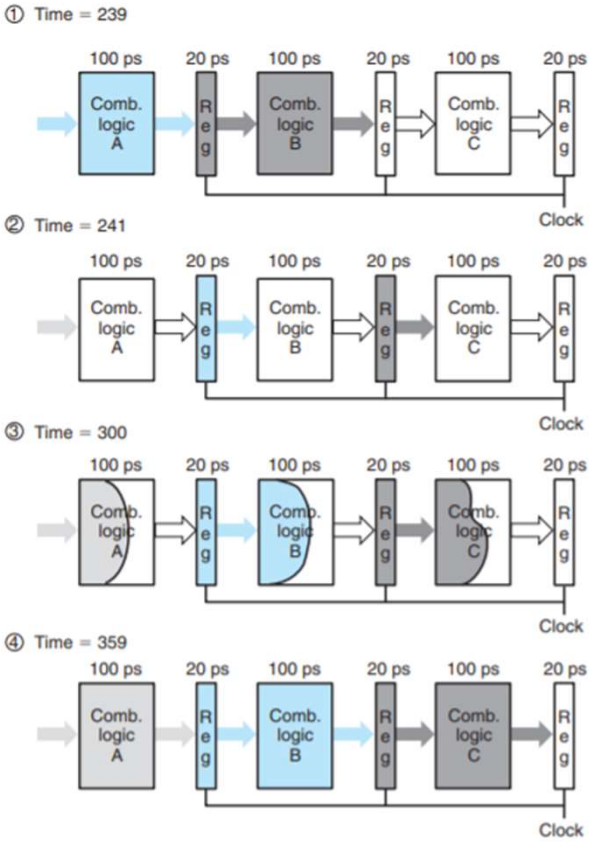


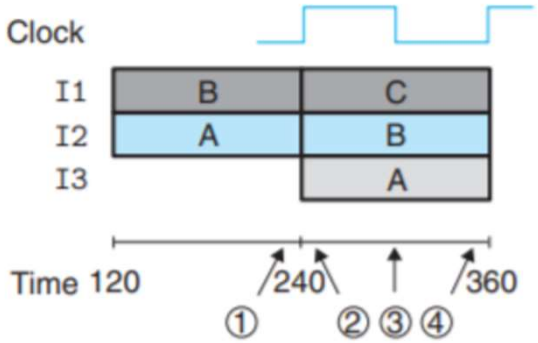


比較

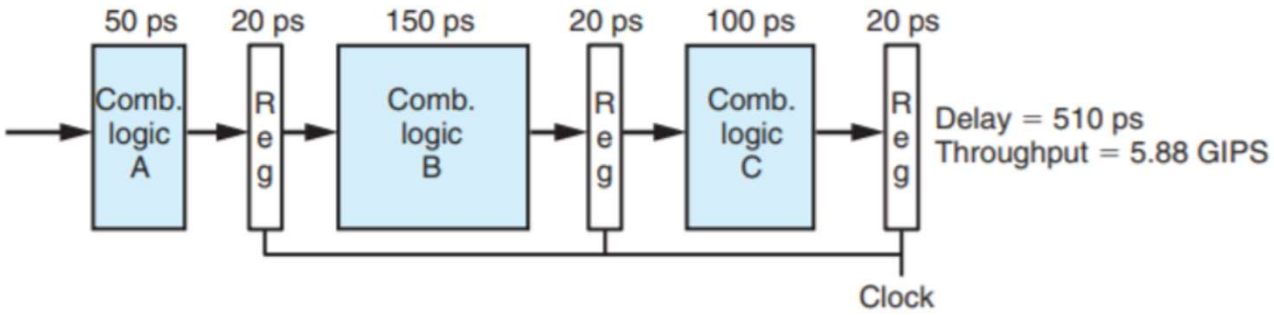
system	thoughput(GIPS)	latency(ps)	clock(ps)
nonpipeline system	3.12	320ps	320
pipeline system	8.33	360ps	120

A DETAILED LOOK AT PIPELINE OPERATION

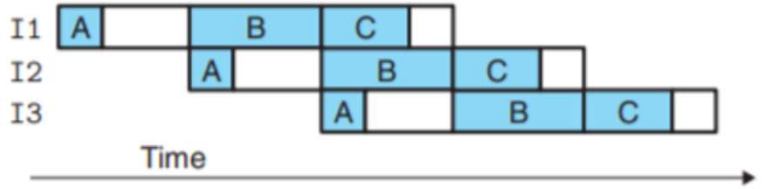




PIPELINE的限制:1.無法平分

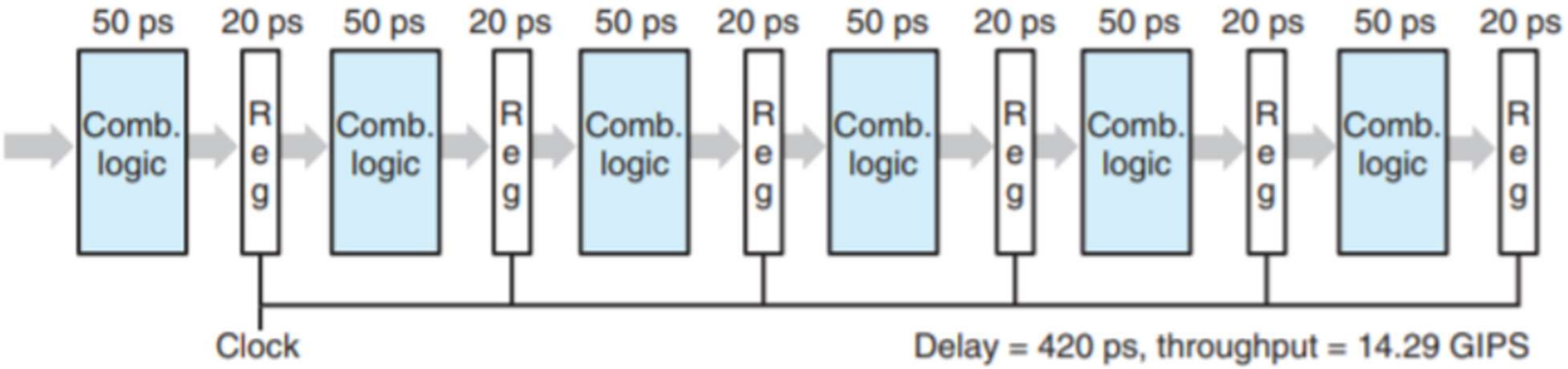


(a) Hardware: Three-stage pipeline, nonuniform stage delays



(b) Pipeline diagram

PIPELINE的限制:2.極限

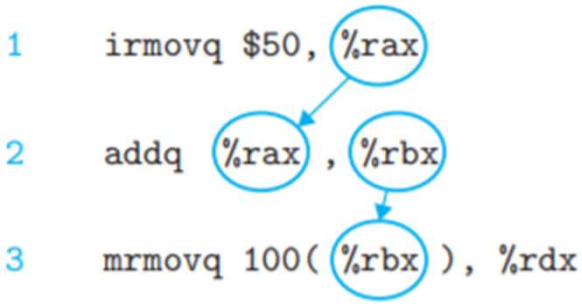


比較

pipeline system	thoughput(GIPS)	latency(ps)	clock(ps)
平分成3份	8.33	360ps	120
平分成6份	14.29	420ps	70

PIPELINING A SYSTEM WITH FEEDBACK

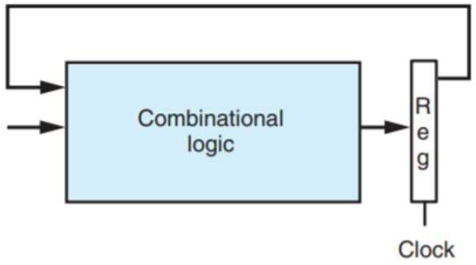
data dependency



control dependency

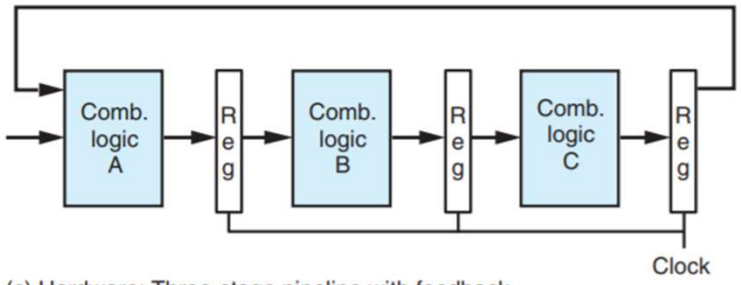
```
1   loop:
2       subq %rdx,%rbx
3       jne targ
4       irmovq $10,%rdx
5       jmp loop
6   targ:
7       halt
```

FEETBACK

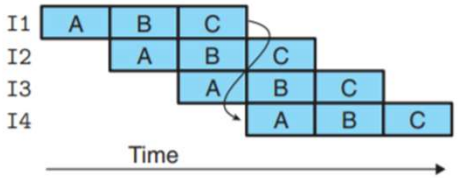


(a) Hardware: Unpipelined with feedback

(b) Pipeline diagram



(c) Hardware: Three-stage pipeline with feedback



(d) Pipeline diagram

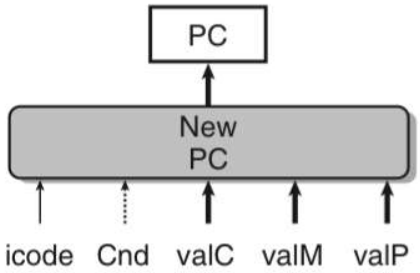
4.5 Pipelined Y86-64 Implementation

目錄

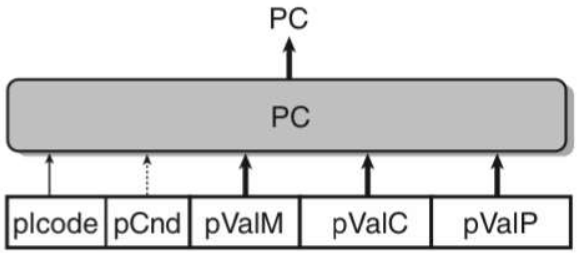
- SEQ變SEQ+的過程
- Insert Pipeline Registers
- PIPE-架構
- PIPE的實作、PCL(Pipeline Control Logic)
- Performance Analysis and Unfinished Business

SEQ+:Rearranging the Computation Stages

- 對SEQ做改良，稱之為SEQ+
- 把PC 的更新擺在第一順位
- 同一時間還是只能執行一個指令

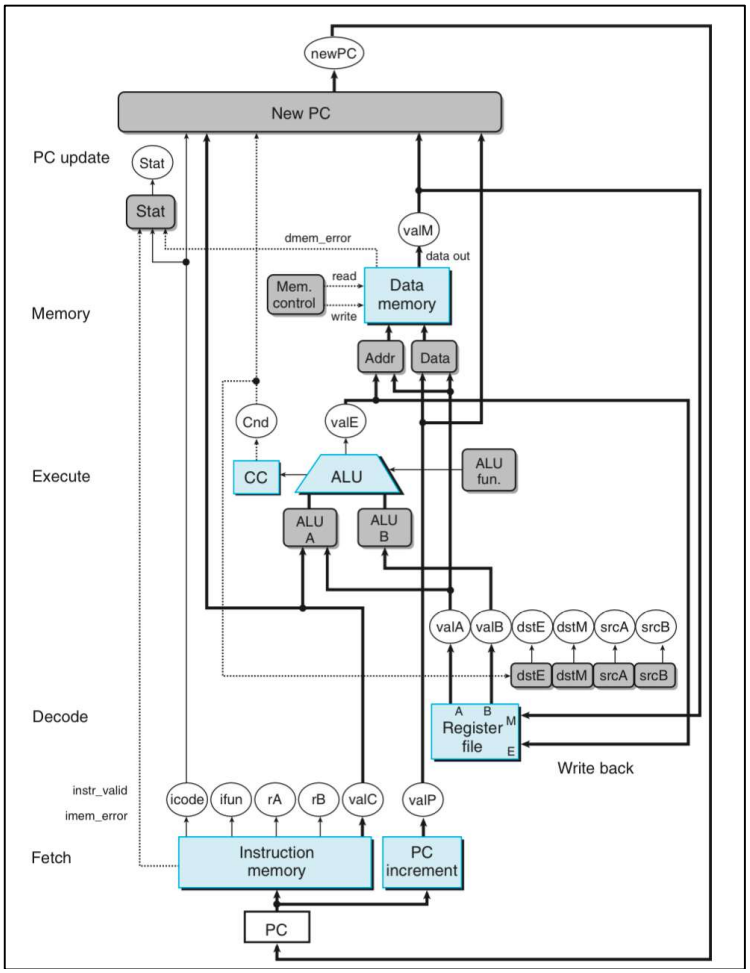


(a) SEQ new PC computation

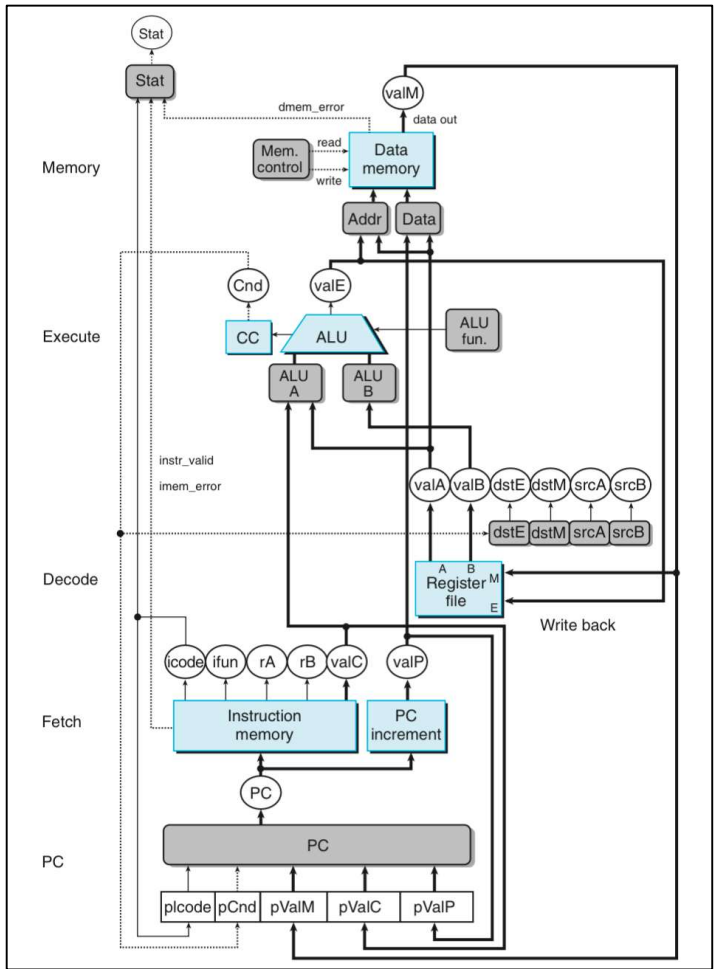


(b) SEQ+ PC selection

SEQ


newPC
New PC
PC update
Stat
Stat
dmem_error
valM
data out
Memory
Mem. control
read
write
Data memory
Addr
Data
Cnd
valE
Execute
CC
ALU
ALU fun.
ALU A
ALU B
valA
valB
dstE
dstM
srcA
srcB
dstE
dstM
srcA
srcB
Decode
A
B
Register file
M
E
Write back
instr_valid
imem_error
icode
ifun
rA
rB
valC
valP
Fetch
Instruction memory
PC increment
PC

SEQ+


Stat
Stat
valM
dmem_error
data out
Memory
Mem. control
read
write
Data memory
Addr
Data
Cnd
valE
Execute
CC
ALU
ALU fun.
ALU A
ALU B
instr_valid
imem_error
valA
valB
dstE
dstM
srcA
srcB
dstE
dstM
srcA
srcB
Decode
A
B
Register file
M
E
Write back
icode
ifun
rA
rB
valC
valP
Fetch
Instruction memory
PC increment
PC
PC
pIcode
pCnd
pValM
pValC
pValP

Inserting Pipeline Registers

- 原本的白色圈圈（可能是電路訊號）變成白色框框（實際的硬體元件）
- 總共5個pipeline register，內含許多資料

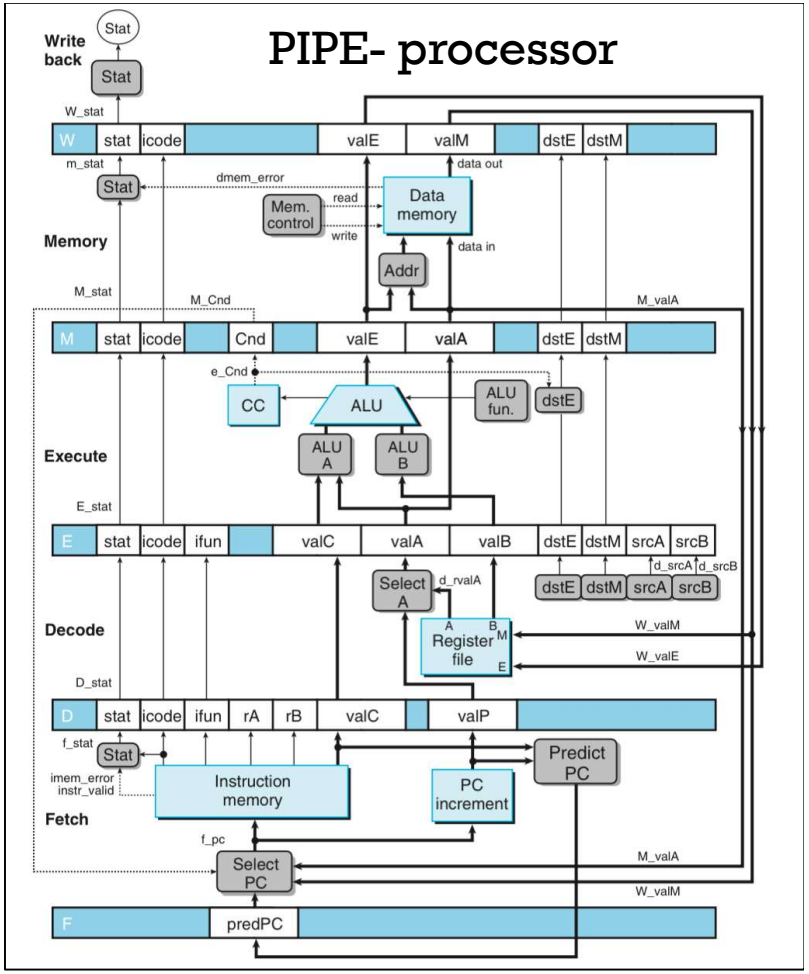
  F: PC的預測值

  D: 新取得的指令

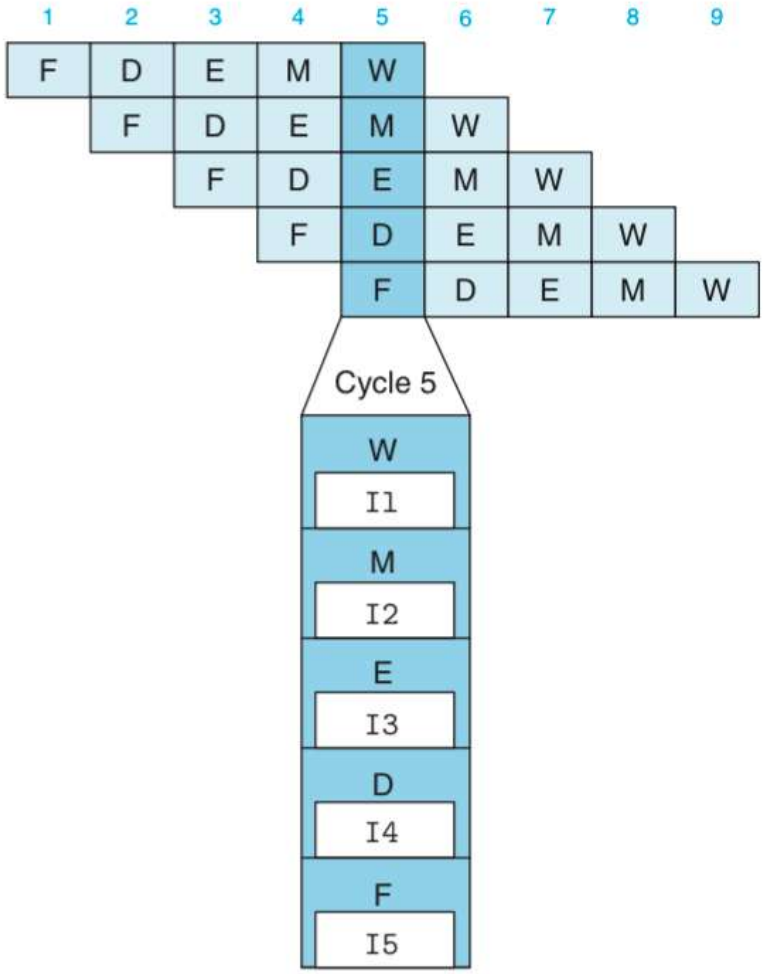
  E: 新解碼的指令、register file的數值

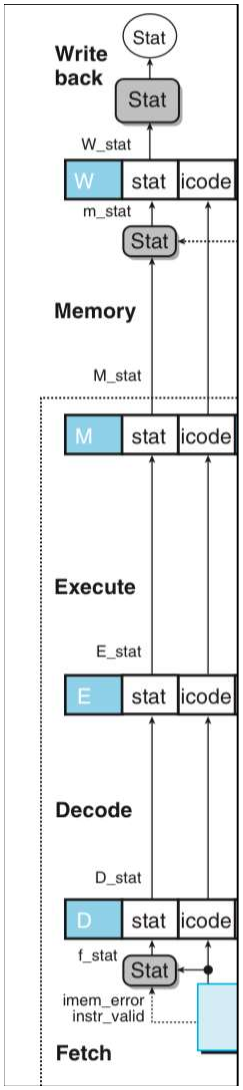
  M: 指令執行結果、分支狀況

  W: 運算結果、回傳address



```
irmovq      $1,%rax     #I1
irmovq      $2,%rbx     #I2
irmovq      $3,%rcx     #I3
irmovq      $4,%rdx     #I4
halt                    #I5
```



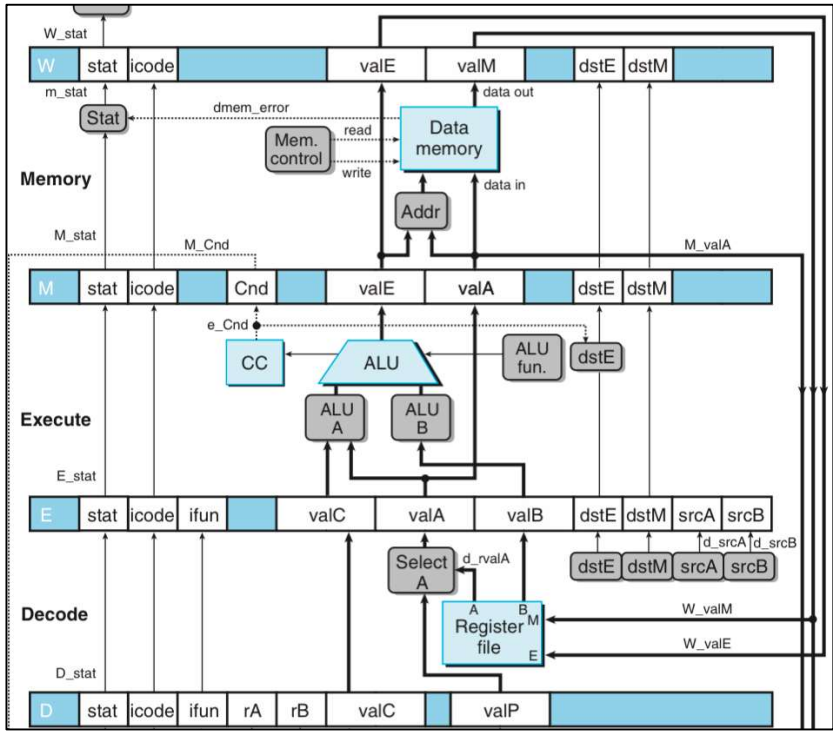


Rearranging and Relabeling Signals

- PIPE-會一次執行多個指令，不能把指令的執行結果亂存
- 存在pipeline register 的stat → D_stat E_stat M_stat W_stat
- 計算完的Stat → f_stat m_stat

Rearranging and Relabeling Signals

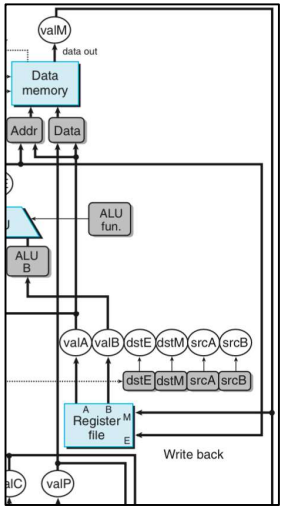
- dstE 和 dstM:
  - 在SEQ+中，可以直接傳回register file
  - 但在PIPE-中，因為很多指令同時進行，所以要等到指令跑到write-back stage才能回傳
  - Keep all information of a particular instruction in a single pipeline stage

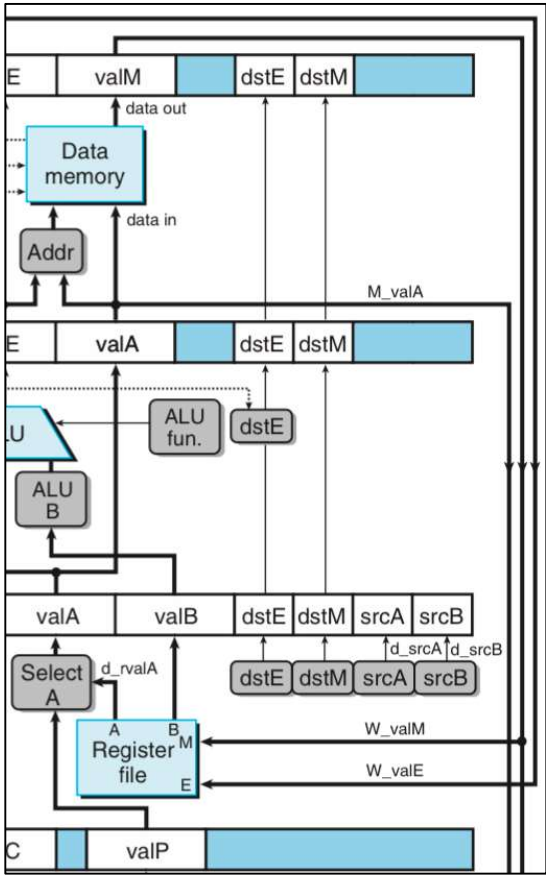


Rearranging and Relabeling Signals

- Decode Stage 的 "Select A" Block:
  - Call 和 jump 指令只需要valP，不需要d_rvalA
  - Reduce pipeline register state by merging
  - 取代SEQ+裡的Data

SEQ+





PIPE-

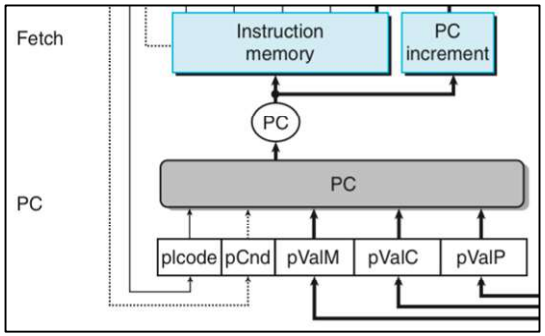
Next PC Prediction

- 要在fetch完現在的指令之後馬上決定下個指令的位址：
  - call 和 unconditional jump 是 valC
  - call.jump.return以外的指令是valP

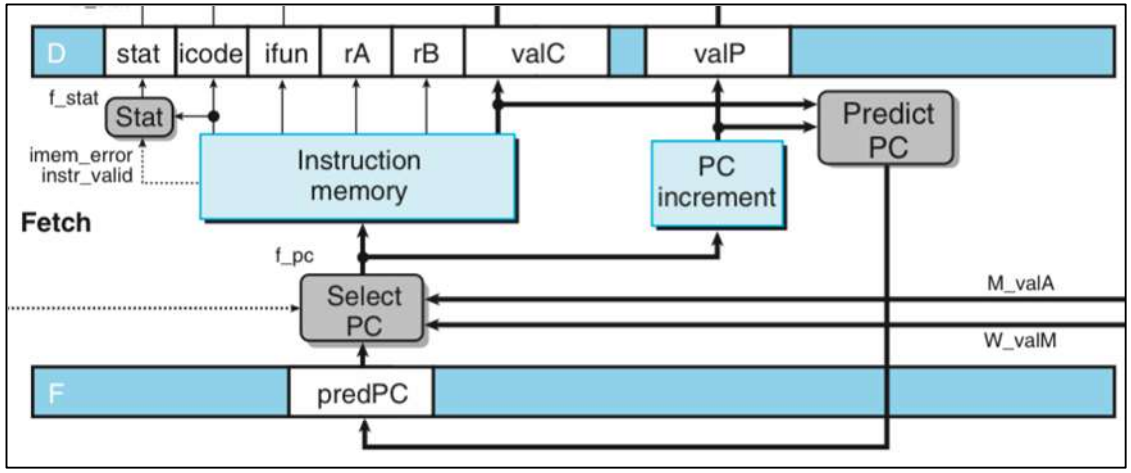
- 如果有control hazard發生，就無法馬上取得正確指令位址：
  - Mispredicted conditional jump:
    - Branch Prediction: 預測conditional jump成不成立
    - 在這裡我們預測它們全部成立（回傳valC）
  - Return:
    - 太多可能性，所以不會預測
    - 繼續執行其他指令直到return 走到 write-back stage

Next PC Prediction

- PIPE- 的 “Select PC” block 取代了SEQ+ 的 “PC” block
- Select PC 從 M_valA（mispredicted conditional jump 走到 memory）、W_valM（return 走到 write-back 時）、predPC 中選擇一個



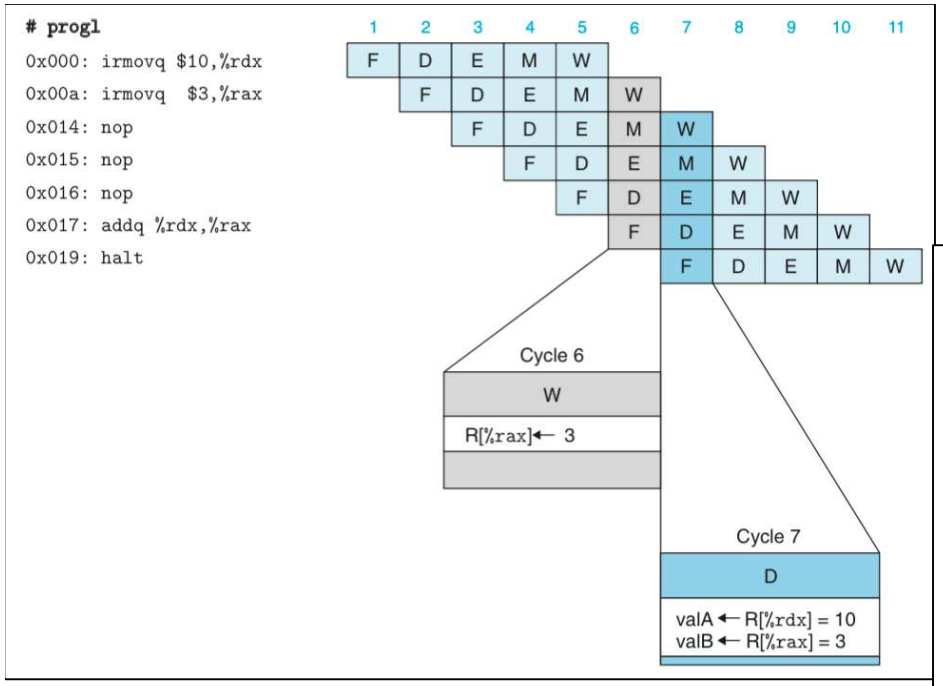
SEQ+

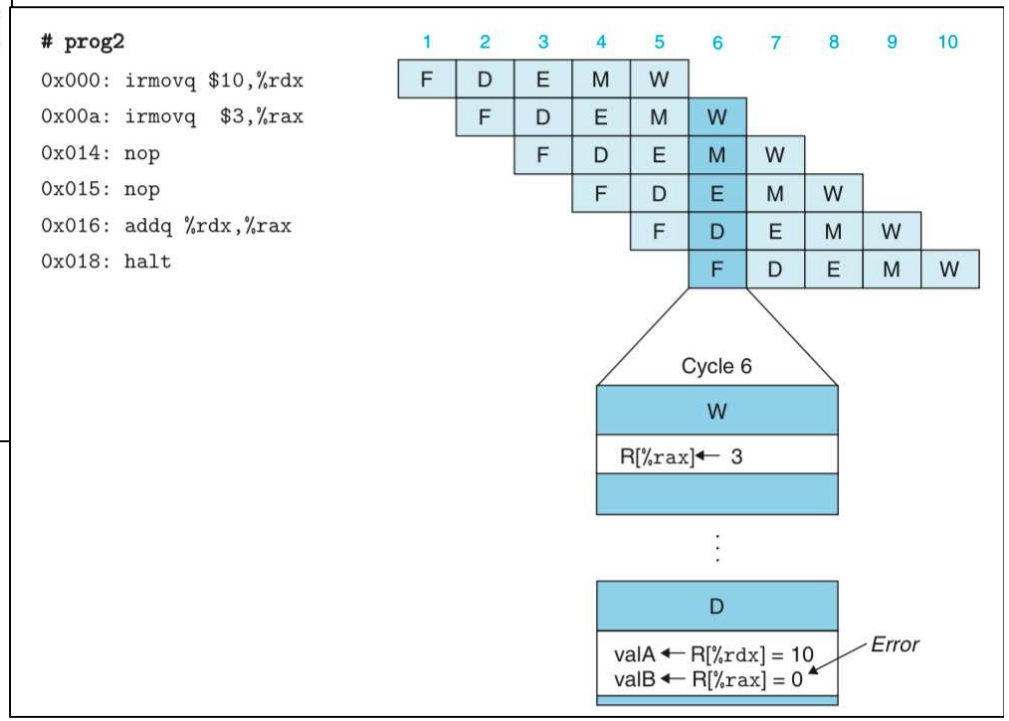


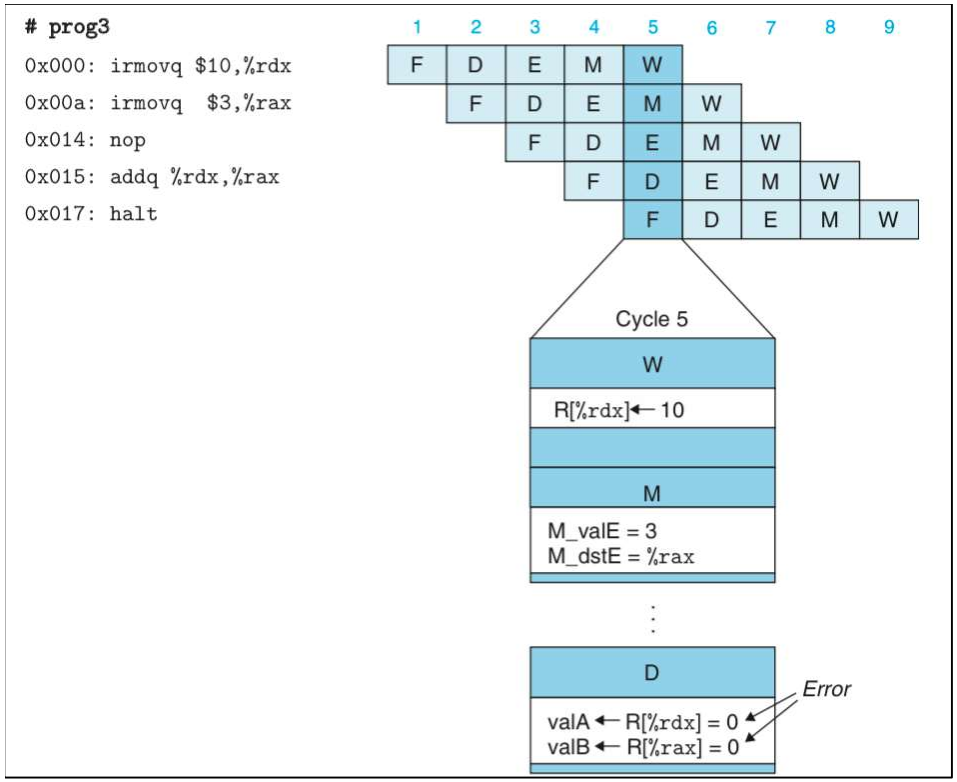
PIPE-

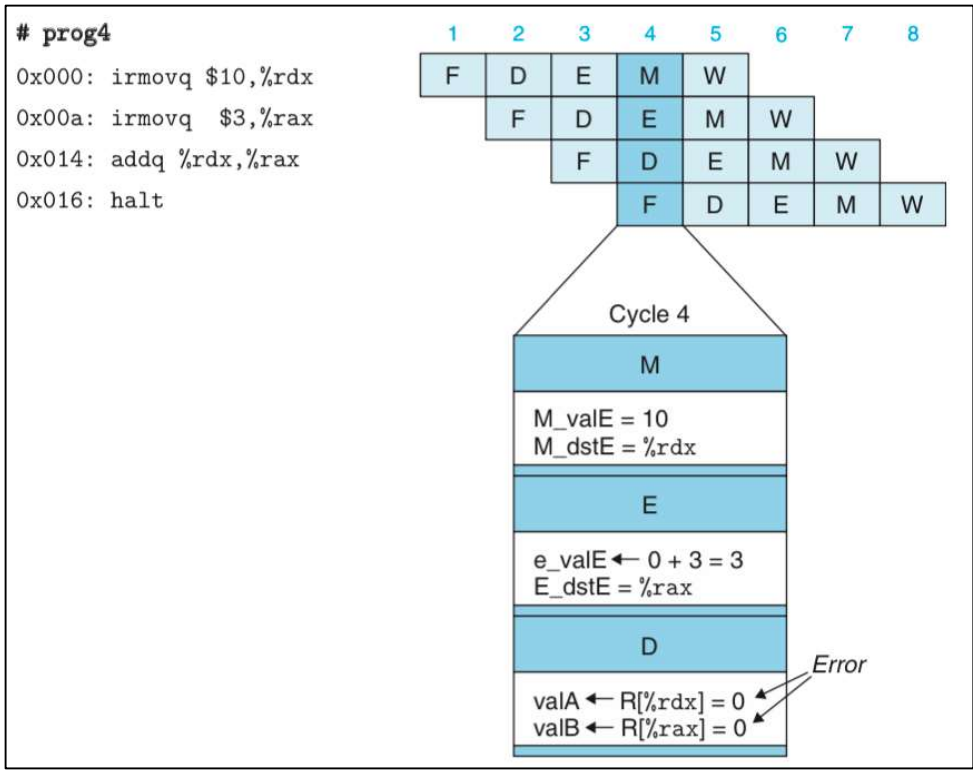
Pipeline Hazards

- 在pipeline時，必須考慮到指令和指令間的dependency。
- 分成data dependency 和 control dependency
- 這些dependency可能造成hazards（風險）
- 因此，hazards 也分為data hazards 和 control hazards


prog1
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: halt
1 2 3 4 5 6 7 8 9 10 11
F D E M W
Cycle 6
W
R[%rax]← 3
Cycle 7
D
valA ← R[%rdx] = 10
valB ← R[%rax] = 3


prog2
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
1 2 3 4 5 6 7 8 9 10
F D E M W
Cycle 6
W
R[%rax]← 3
D
valA ← R[%rdx] = 10
valB ← R[%rax] = 0
Error


prog3
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
1 2 3 4 5 6 7 8 9
F D E M W
F D E M W
F D E M W
F D E M W
F D E M W
Cycle 5
W
R[%rdx]← 10
M
M_valE = 3
M_dstE = %rax
D
valA ← R[%rdx] = 0
valB ← R[%rax] = 0
Error

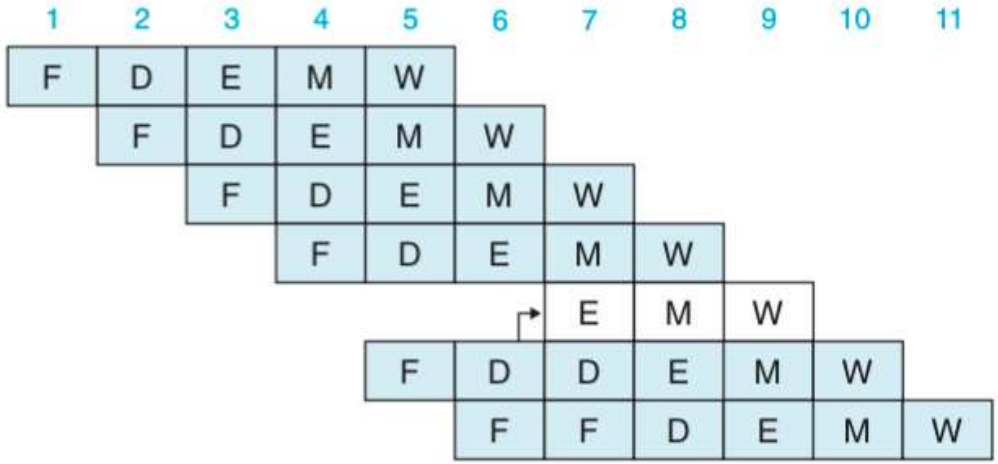

prog4
0x000: irmovq $10,%rdx
0x00a: irmovq $3,%rax
0x014: addq %rdx,%rax
0x016: halt
1 2 3 4 5 6 7 8
F D E M W
F D E M W
F D E M W
F D E M W
Cycle 4
M
M_valE = 10
M_dstE = %rdx
E
e_valE ← 0 + 3 = 3
E_dstE = %rax
D
valA ← R[%rdx] = 0
valB ← R[%rax] = 0
Error

Avoid Data Hazards

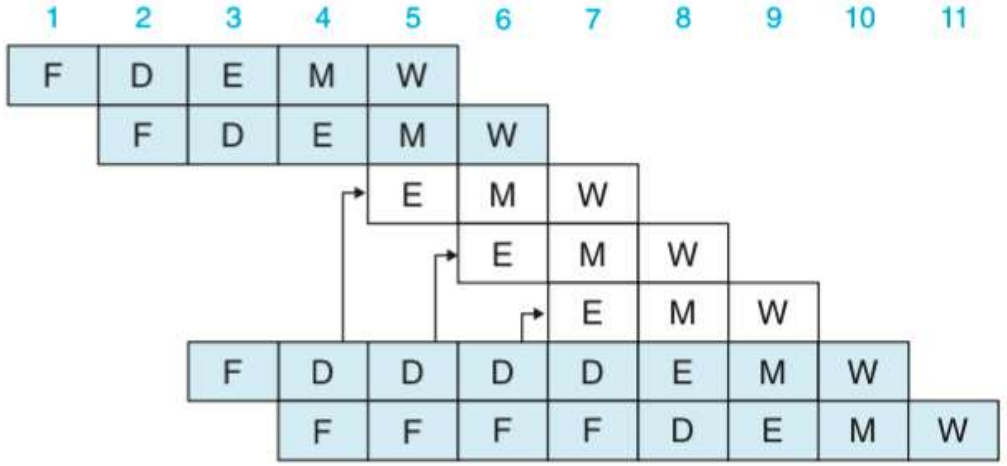
- 方法1：Stalling
  - 這個指令所需的資料還沒 write-back 之前就是 data hazard
  - 當PCL偵測到data hazards時，就必須再執行一次指令的decode
  - 每次讓指令停留在decode stage時，就在execute stage引入bubble
  - 等同於插入nop指令，不會對程式或記憶體造成任何影響

  - 缺點：連續很多個指令都要用同一個register，但這些指令又需要stalling，一直stall下去就爆炸啦！

```
# prog2
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addlq %rdx,%rax
0x018: halt
```



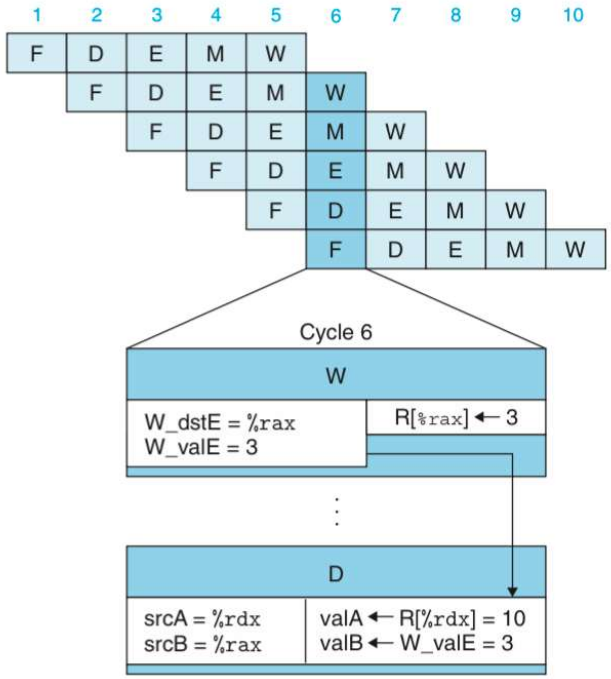

```
# prog4
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
       bubble
       bubble
       bubble
0x014: addq %rdx,%rax
0x016: halt
```



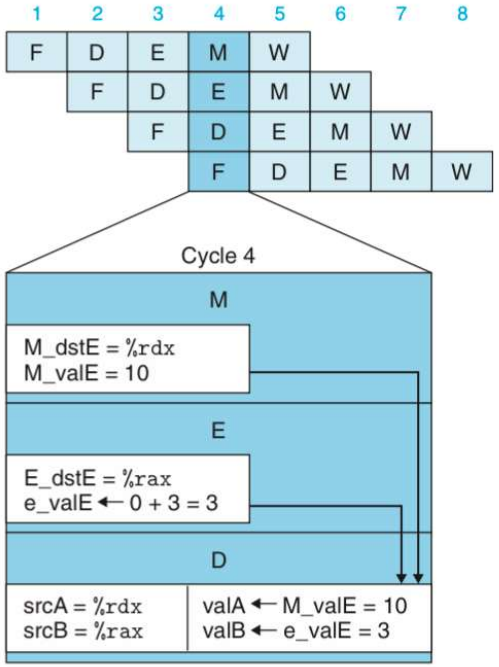
Avoid Data Hazards

- 方法2：Forwarding(Bypassing)
  - 把前面指令在後面階段計算完的值直接傳到valA和valB
  - 可以在該指令decode stage的clock cycle結束前完成
  - 需要加控制元件到硬體結構上

```
# prog2
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```




```
# prog4
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

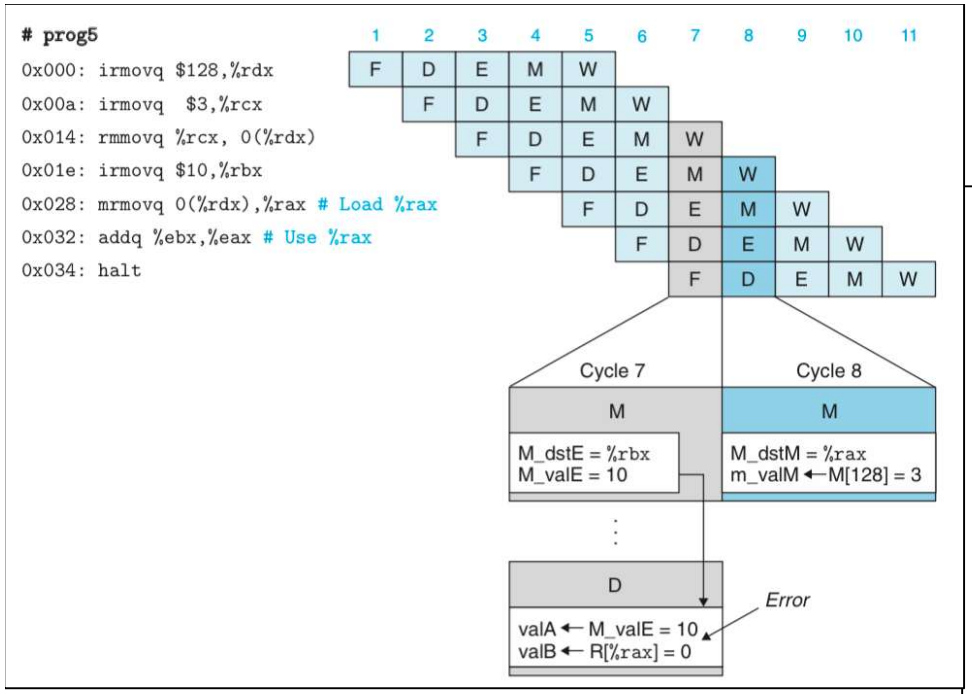


Avoid Data Hazards

Load/Use data hazards:

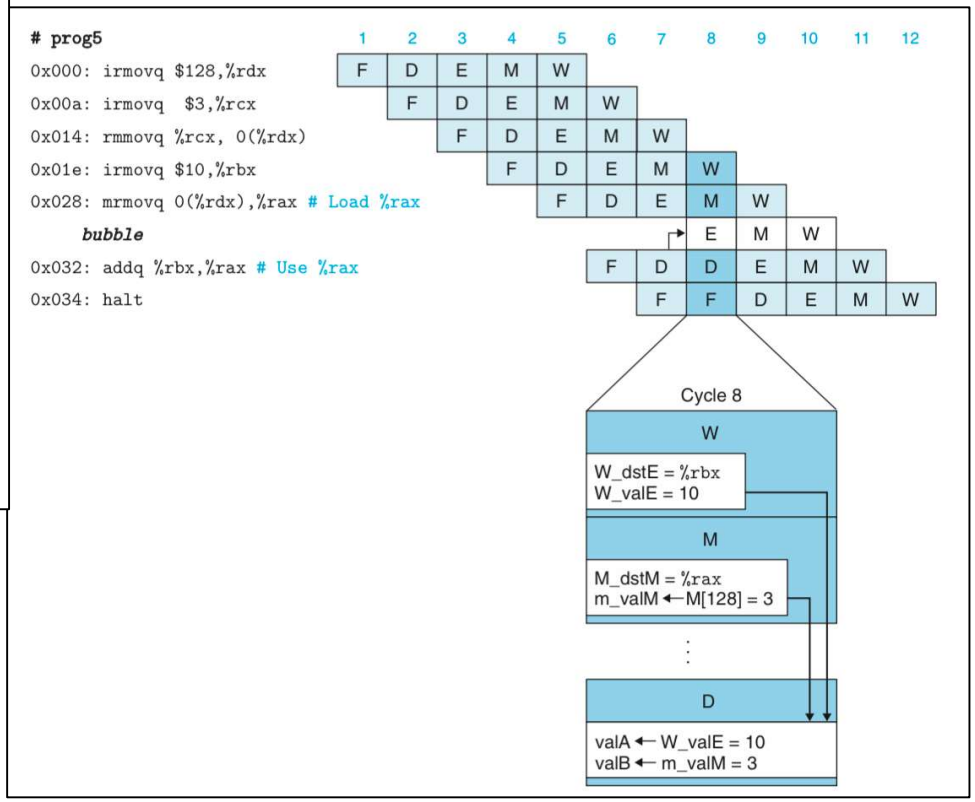
- 指令位在execute stage
- mrmovq/popq 指令要到memory stage才能讀取memory
- 但此時下個指令需要用到 register 或 forward 過來的值
- 使用forwarding根本來不及

- 解決辦法：forwarding + stalling!



forwarding

forwarding + stalling

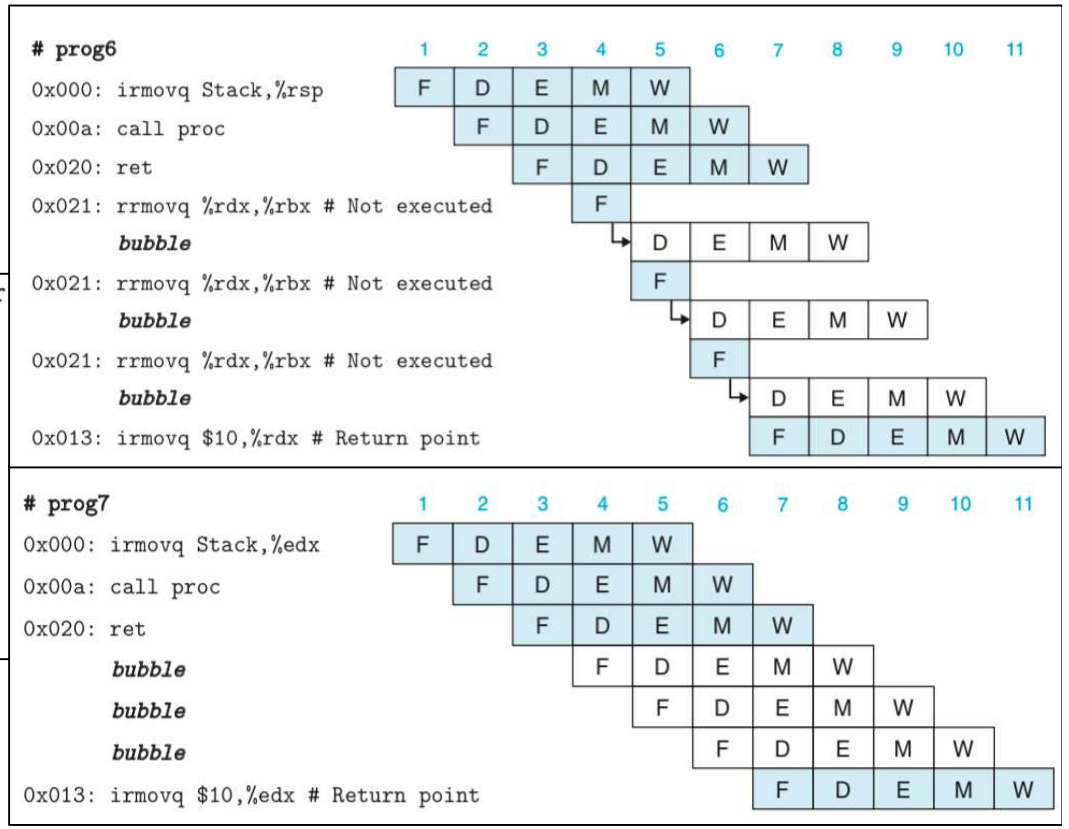


Avoid Control Hazards

- 狀況1：Return instruction
  - 走到write-back stage時，select PC才能收到return address
  - 中間有3個clock cycle無法做任何事

  - 解決辦法：引入3個bubble

```
0x000:    irmovq stack,%rsp  #   Initialize stack pointer
0x00a:    call proc          #   Procedure call
0x013:    irmovq $10,%rdx    #   Return point
0x01d:    halt
0x020: .pos 0x20
0x020: proc:                 # proc:
0x020:    ret                #   Return immediately
0x021:    rrmovq %rdx,%rbx   #   Not executed
0x030: .pos 0x30
0x030: stack:                # stack: Stack pointer
```



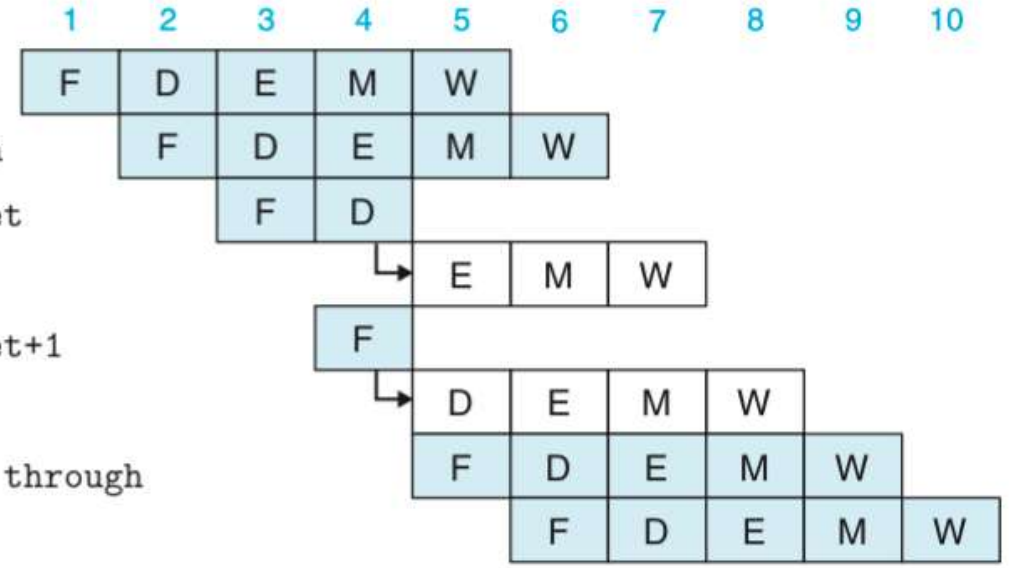
Avoid Control Hazards

- 狀況2：Mispredicted conditional jump
  - 指令走到execute stage發現不用跳
  - 此時2個不該被執行的指令已經通過fetch stage

- 解決辦法：Cancel(squash) misfetched instructions
  - 在2個錯誤指令的下個clock cycle引入bubble
  - 同時 fetch 正確應該被執行的下個指令

```
0x000:     xorq %rax,%rax
0x002:     jne  target          # Not taken
0x00b:     irmovq $1, %rax      # Fall through
0x015:     halt
0x016: target:
0x016:     irmovq $2, %rdx      # Target
0x020:     irmovq $3, %rbx      # Target+1
0x02a:     halt
```

```
# prog7
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovl $2,%rdx # Target
       bubble
0x020: irmovl $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```



（　）1.引入bubble等於stalling

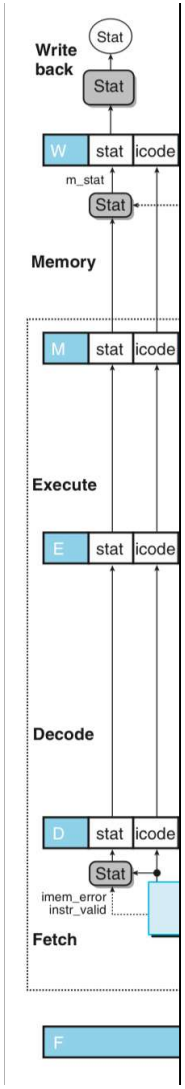
（　）2.引入幾個bubble就等於幾個clock cycle wasted

1.X　2.O

Exception Handling

- 這部分的設計很複雜困難
  - 無法預測發生時間
  - PIPE 把異常狀態偵測分在 fetch 和 memory stage 處理
  - 程式狀態可能在 execute . memory . write-back stage 被改變
- 我們期望處理器在遇到內部異常的時候能夠：
  - 1. Set the appropriate status code, as listed below
  - 2. Complete the excepting instruction and cause no effect
  - 3. Halt

1	AOK	Normal operation	
2	HLT	halt instruction encountered	→ Fetch stage 偵測
3	ADR	Invalid address encountered	→ Memory stage 偵測
4	INS	Invalid instruction encountered	→ Fetch stage 偵測



Exception Handling

- 狀況1：很多異常同時發生，規則上要回報最先開始執行的指令
- 狀況2：偵測到異常，但指令因為mispredicted branch被取消，要避免回報異常狀況

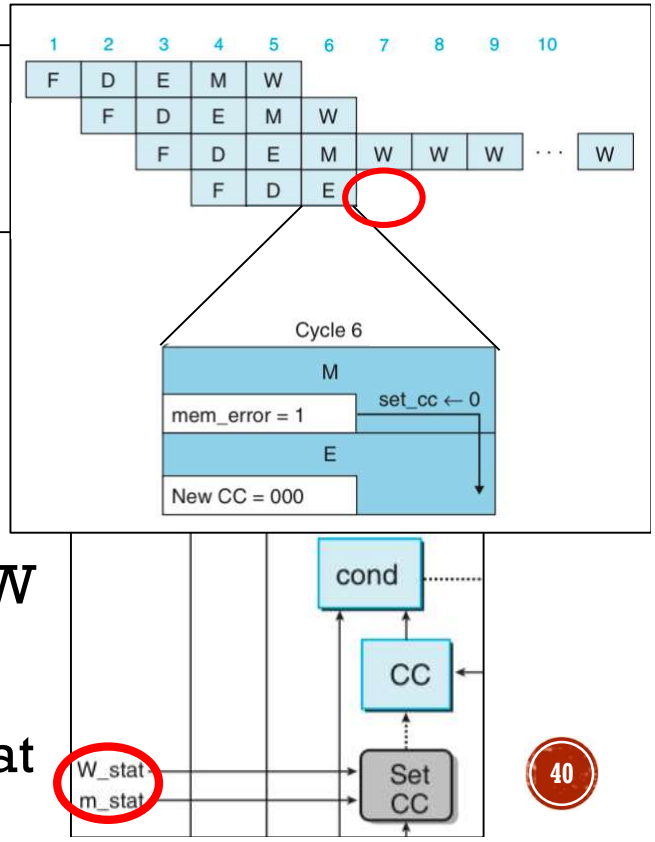
```
0x000: 6300                    |    xorq %rax,%rax
0x002: 7416000000000000000     |    jne  target       # Not taken
0x00b: 30f00100000000000000    |    irmovq $1, %rax   # Fall through
0x015: 00                      |    halt
0x016:                         | target:
0x016: ff                      |    .byte 0xFF        # Invalid instruction code
```

- 狀況1和狀況2的解決辦法：

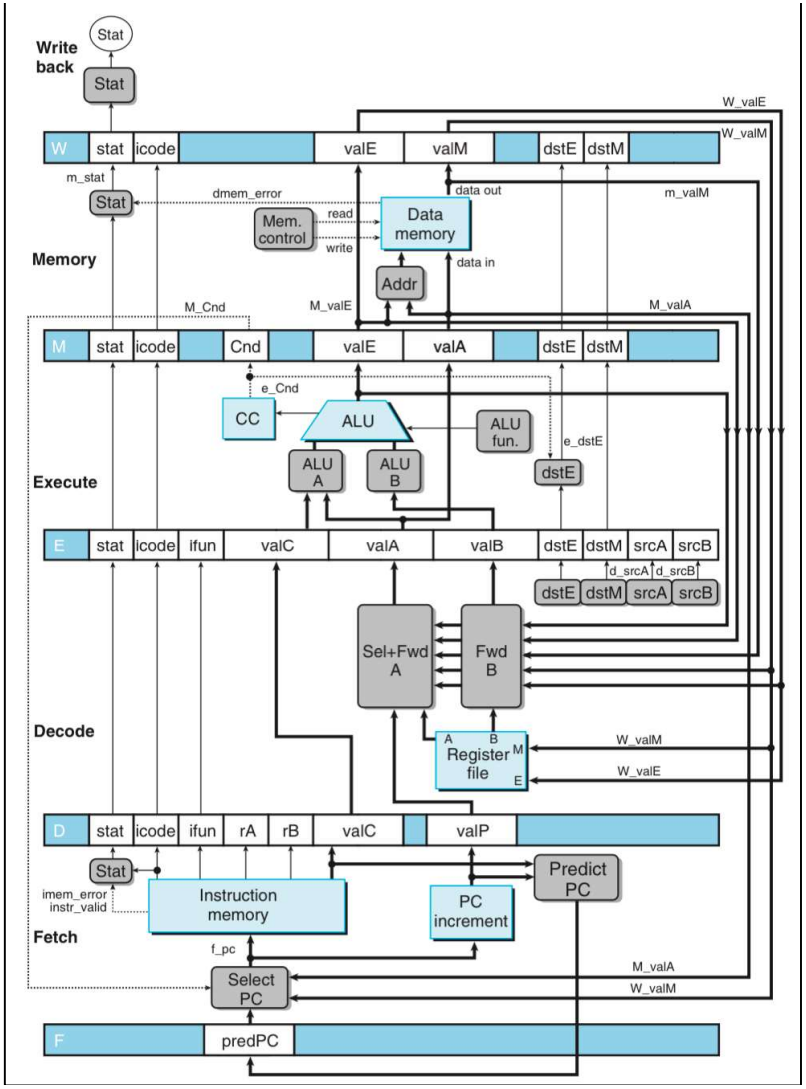
  在每個 pipeline register 都加入 status code

Exception Handling

```
1       irmovq $1,%rax
2       xorq %rsp,%rsp    # Set stack pointer to 0 and CC to 100
3       pushq %rax        # Attempt to write to 0xfffffffffffffff8
4       addq  %rax,%rax   # (Should not be executed) Would set CC to 000
```



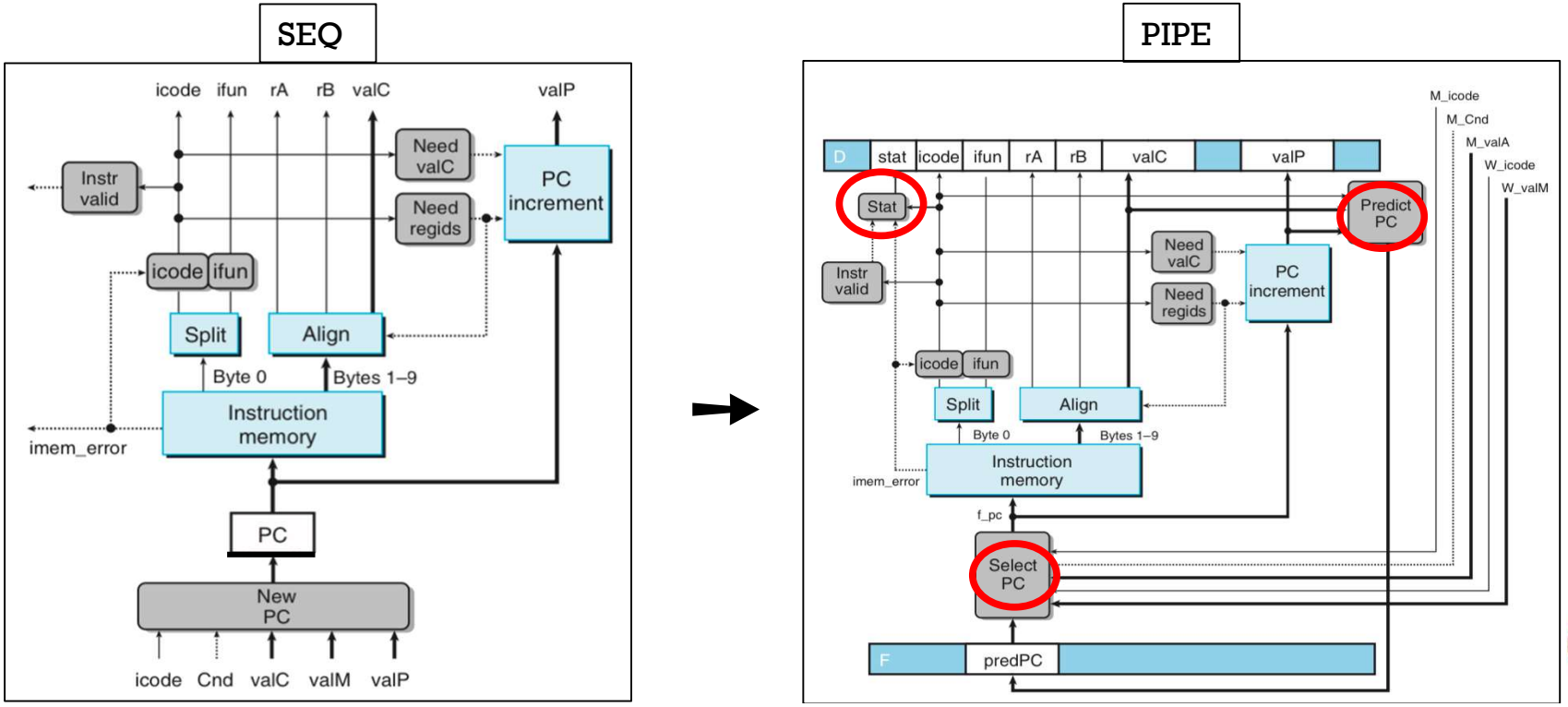
- 狀況3：異常指令還沒被回報時，要阻止
  即將被改變的系統狀態被改變
- 狀況3解決辦法：
  1. 把錯誤指令stall在pipeline register W
  2. 在memory stage注入bubble
  3. 在execute stage加入W_stat 和 m_stat



PIPE hardware structure

PIPE Stage Implementations

- PC Selection and Fetch Stage



PIPE Stage Implementations

- PC Selection and Fetch Stage的改變
  - 多了 PC prediction

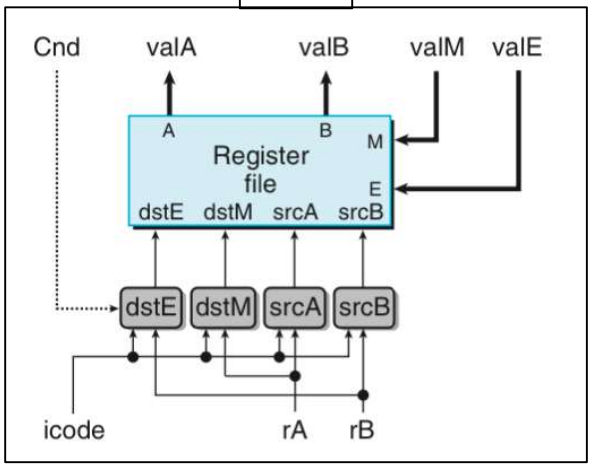
```
word f_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
        1 : f_valP;
];
```

```
word f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        M_icode == IJXX && !M_Cnd : M_valA;
        # Completion of RET instruction
        W_icode == IRET : W_valM;
        # Default: Use predicted value of PC
        1 : F_predPC;
];
```

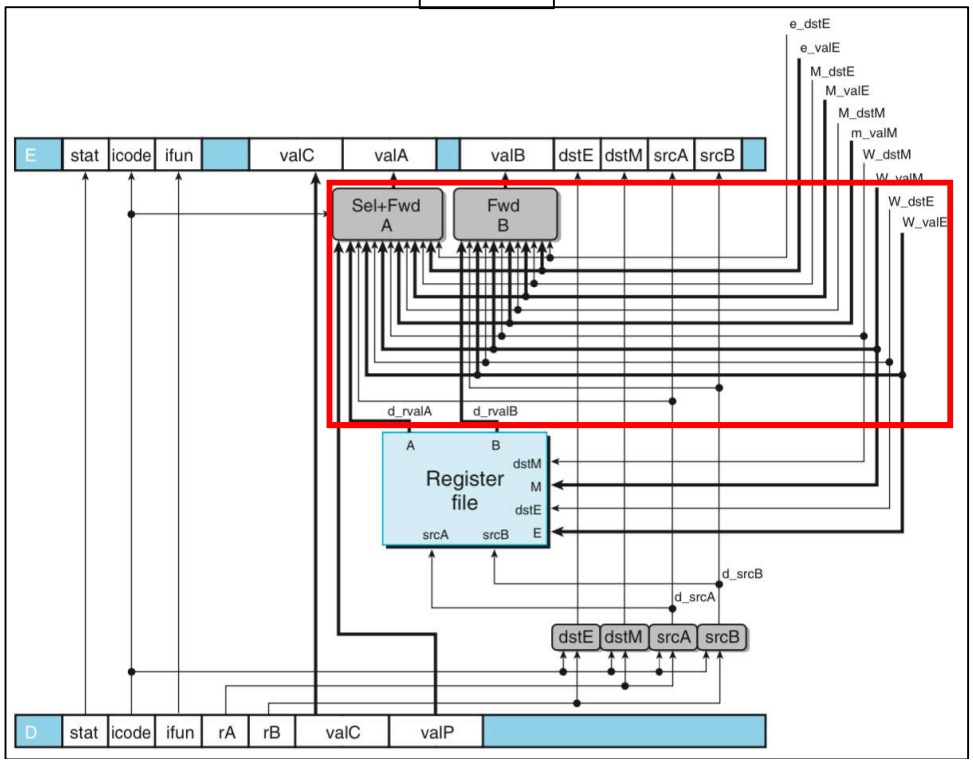
PIPE Stage Implementations

- Decode and Write-Back Stage

SEQ



PIPE

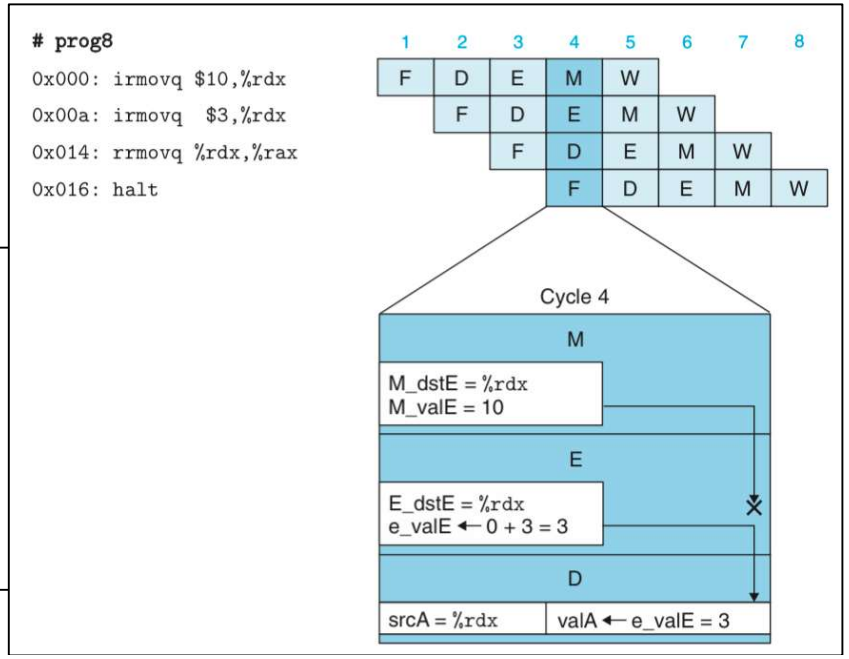


PIPE Stage Implementations

- Decode and Write-Back Stage的改變
  - 多了“Select A” Block
  - 多了 Forwarding

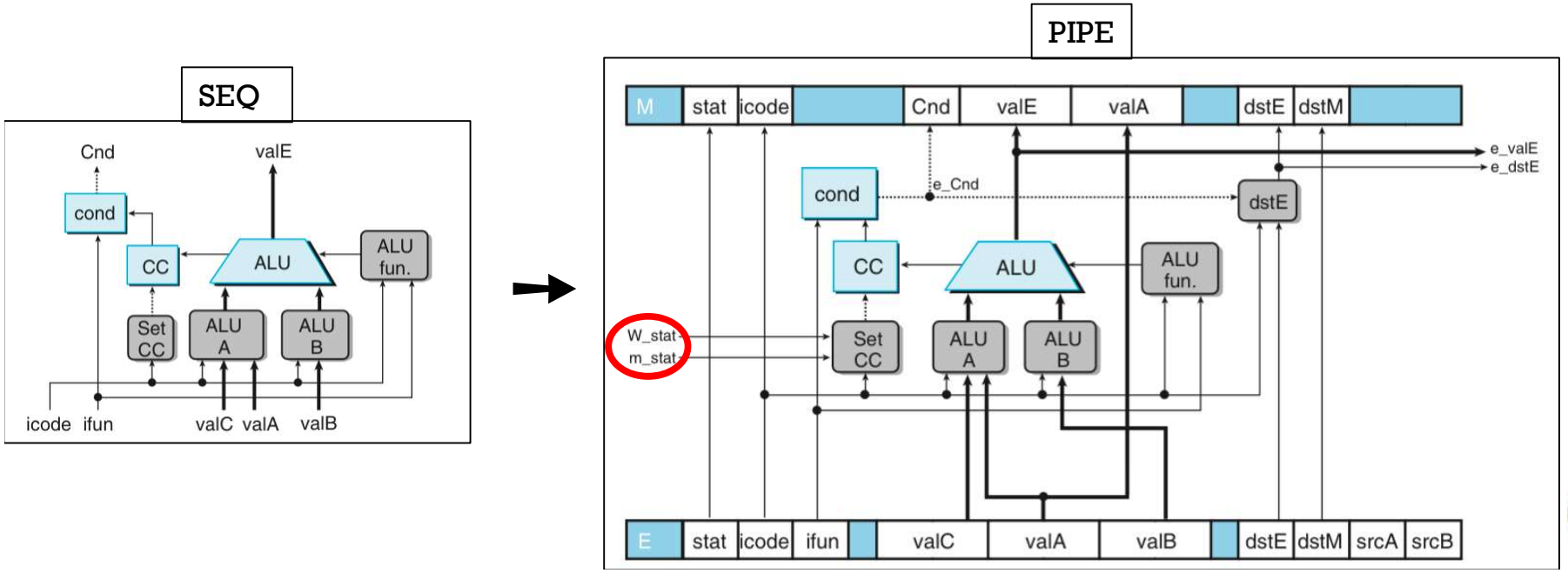
※以最後開始執行的指令為主

```
word d_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == e_dstE : e_valE;     # Forward valE from execute
        d_srcA == M_dstM : m_valM;     # Forward valM from memory
        d_srcA == M_dstE : M_valE;     # Forward valE from memory
        d_srcA == W_dstM : W_valM;     # Forward valM from write back
        d_srcA == W_dstE : W_valE;     # Forward valE from write back
        1 : d_rvalA;  # Use value read from register file
];
```



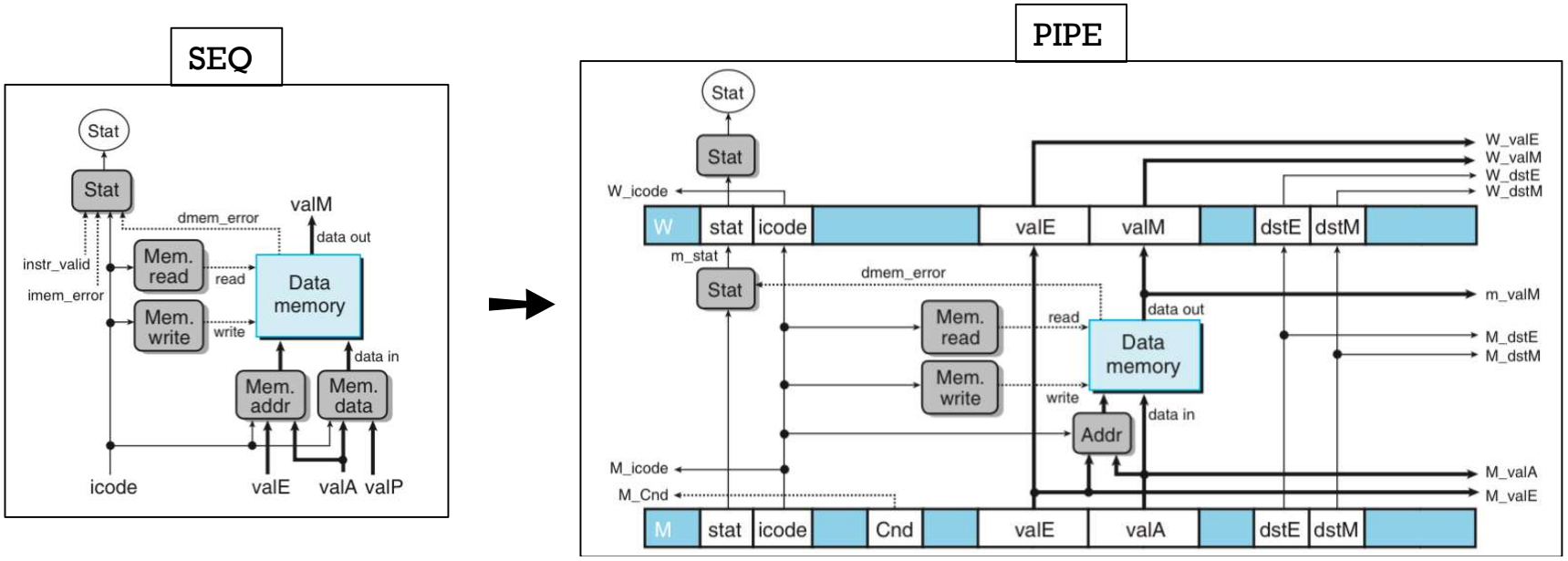
PIPE Stage Implementations

- Execute Stage的改變
  - W_stat 和 m_stat：偵測異常狀況



PIPE Stage Implementations

- Memory Stage的改變：
  - 因為 “Select A” Block ，valP 不見了

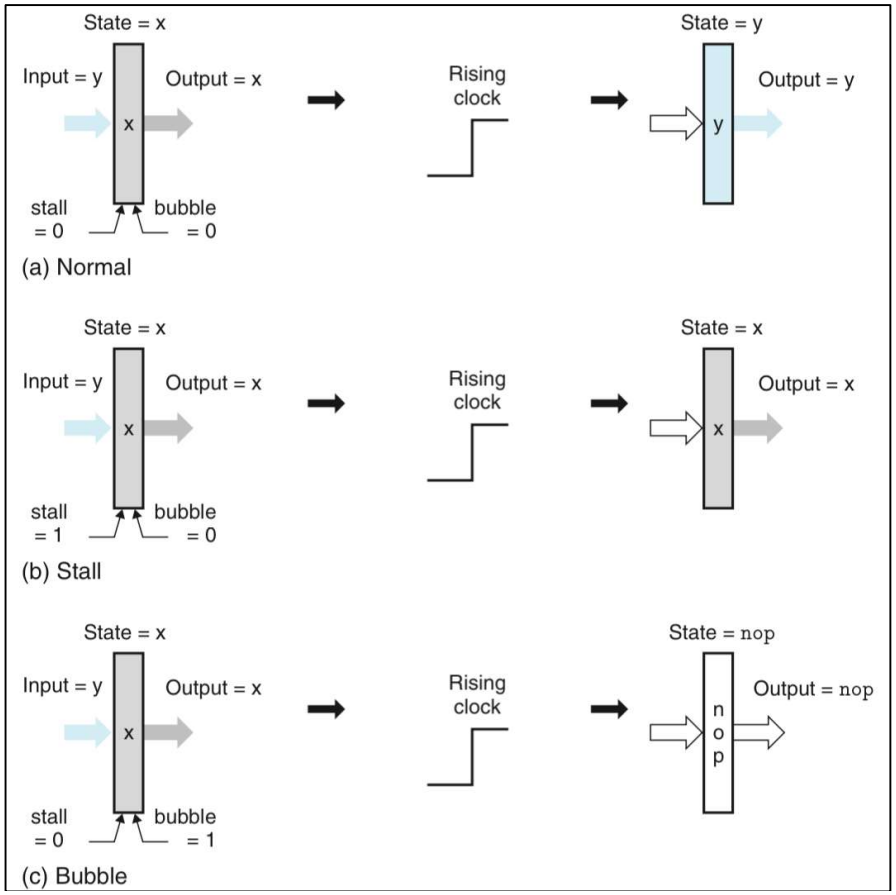


Pipeline Control Logic

- 建構出 PCL 之後才算真正完成 PIPE 的設計
- PCL 必須偵測和處理其他 PIPE 結構沒有滿足的4個control condition：

Condition	Trigger
Processing `ret`	IRET ∈ {D_icode, E_icode, M_icode}
Load/use hazard	E_icode ∈ {IMRMOVQ, IPOPQ} && E_dstM ∈ {d_srcA, d_srcB}
Mispredicted branch	E_icode = IJXX && !e_Cnd
Exception	m_stat ∈ {SADR, SINS, SHLT} || W_stat ∈ {SADR, SINS, SHLT}

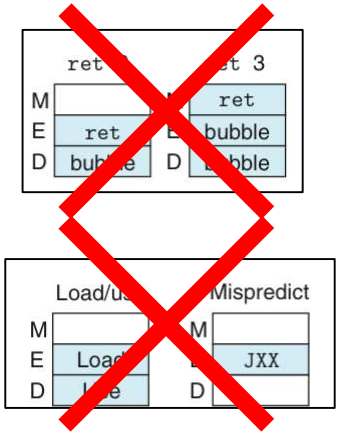
Pipeline Control Mechanisms



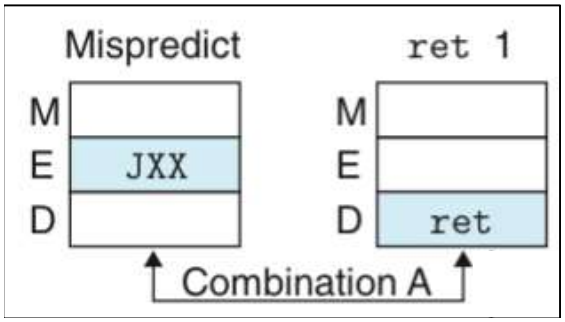
- Bubble:
  - D & E: 把 icode 設為 INOP
  - E: 把 dstE.dstM.srcA.srcB 設為 RNONE

	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal

Combination of Control Condition



- Combination A :
  - Return 是 jump 到某分支後的指令
  - 必須偵測到misprediction，取消 return 指令
- 解決辦法：1. 按處理 mispredicted branch 的方式處理
  2. stall pipeline register F

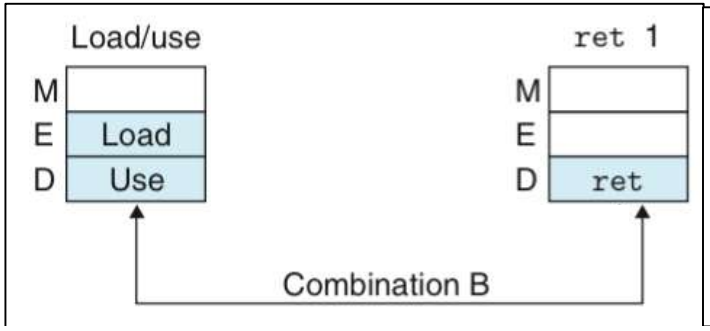


	Pipeline register				
Condition	F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Mispredicted branch	normal	bubble	bubble	normal	normal
Combination	stall	bubble	bubble	normal	normal

Combination of Control Condition

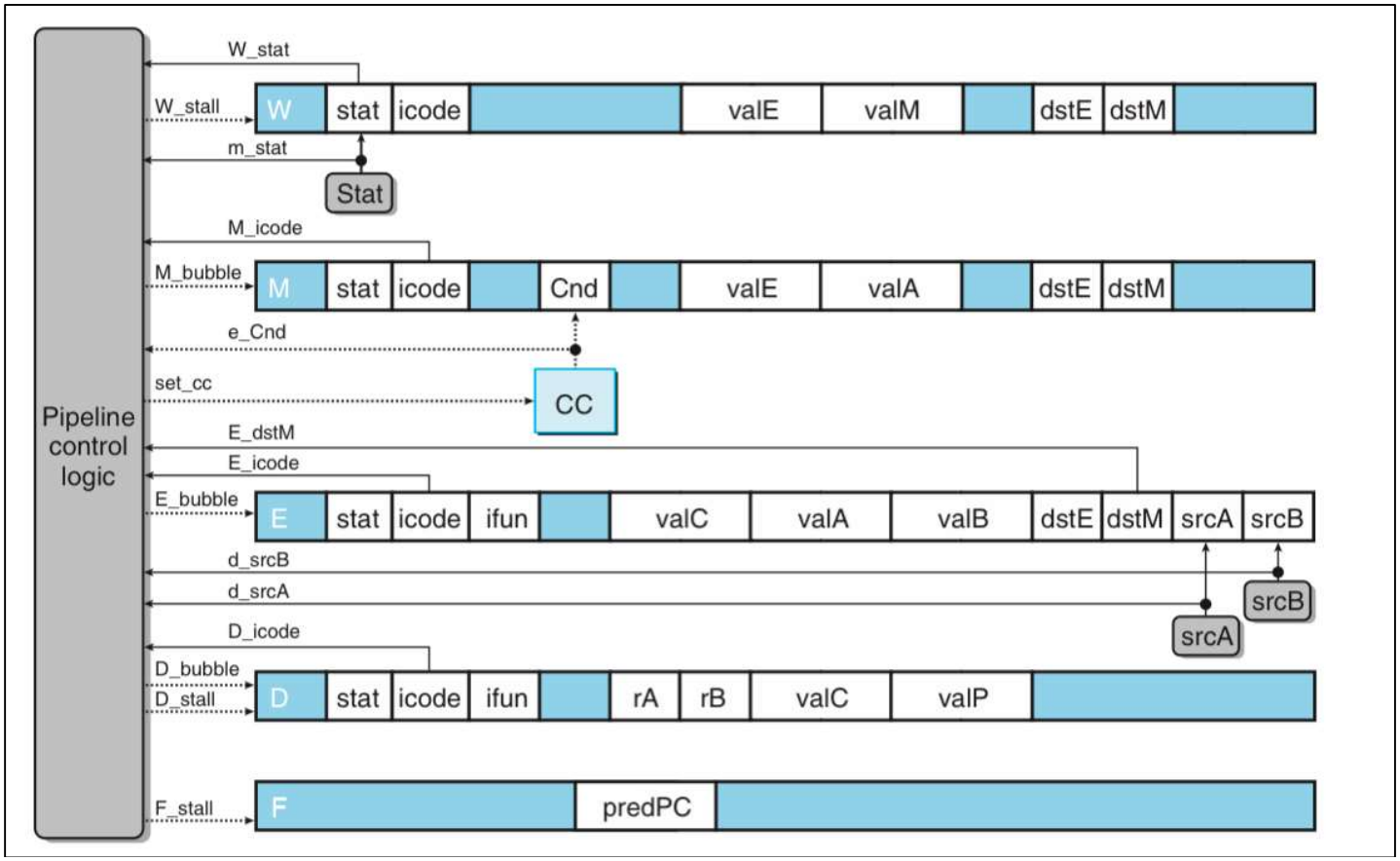
- Combination B :
  - Load 指令用了 %rsp，而 return 要把 %rsp 的位址 pop 掉
  - 必須把 return 指令留在 decode stage
- 解決辦法：1.先 stall return 指令

  2.下一個 clock cycle 再按處理return 指令的方式處理



Condition	Pipeline register F	D	E	M	W
Processing `ret`	stall	bubble	normal	normal	normal
Load/use hazard	stall	stall	bubble	normal	normal
Combination	stall	bubble+stall	bubble	normal	normal
Desired	stall	stall	bubble	normal	normal

Control Logic Implementation



Control Logic Implementation

```
bool F_stall =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB } ||
        # Stalling at fetch while ret passes through pipeline
        IRET in { D_icode, E_icode, M_icode };
```

```
bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }) &&
          IRET in { D_icode, E_icode, M_icode };
```

Performance Analysis

- CPI (Cycles Per Instruction):
  - C_i : execute stage 正常執行的 cycle 數
  - C_b : execute stage 執行被注入的 bubble 的 cycle 數
  - penalty (p) = C_b/ C_i : 表示平均每個執行的指令有幾個bubble被注入
  - p = lp (load penalty) + mp (mis.. penalty) + rp (return penalty)

$$\text{CPI} = \frac{C_i + C_b}{C_i} = 1.0 + \frac{C_b}{C_i} = 1.0 + lp + mp + rp$$

Performance Analysis

- Goal : design a pipeline that CPI is 1
- Our PIPE has CPI 1.27
- We can try more to deal with mp

Cause	Name	Instruction frequency	Condition frequency	Bubbles	Product
Load/use	*lp*	0.25	0.20	1	0.05
Mispredict	*mp*	0.20	0.40	2	0.16
Return	*rp*	0.02	1.00	3	0.06
Total penalty					0.27

Unfinished Business

- 1. Multicycle Instructions
  - Floating-point multiplication : 3~4 cycles
  - Integer multiplication/division : up to 64 cycles

  - 解決辦法：
    - (1). expand capabilities of execute stage
    - (2). two special hardware functional units

Unfinished Business

- 2.Interfacing with the Memory System
  - Memory accessed may be far away
  - Require translation from virtual to physical addresses

  - 解決辦法：
    - normal : first-level caches * 2 + TLB
    - cache miss with missing data not in disk : stall for 3~20 clock cycles
    - cache miss with missing data in disk : call the OS page fault handler