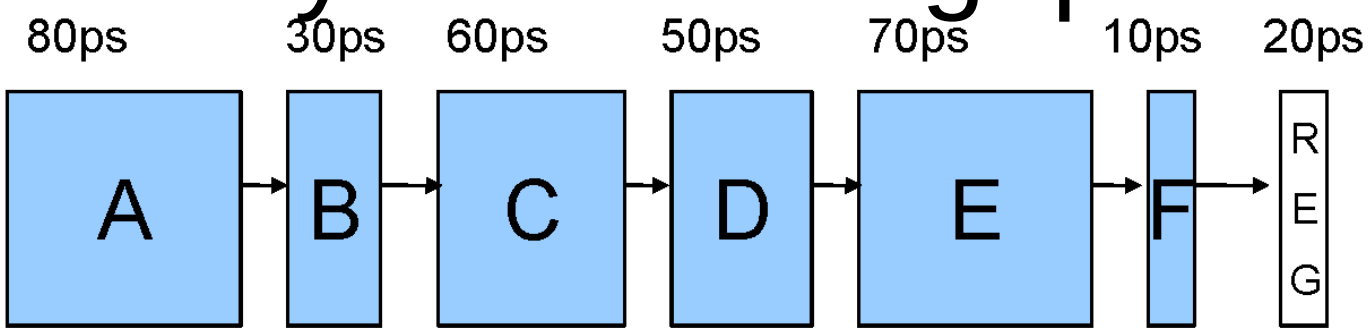
Pipelined Design

תרגול 12

Latency and Throughput

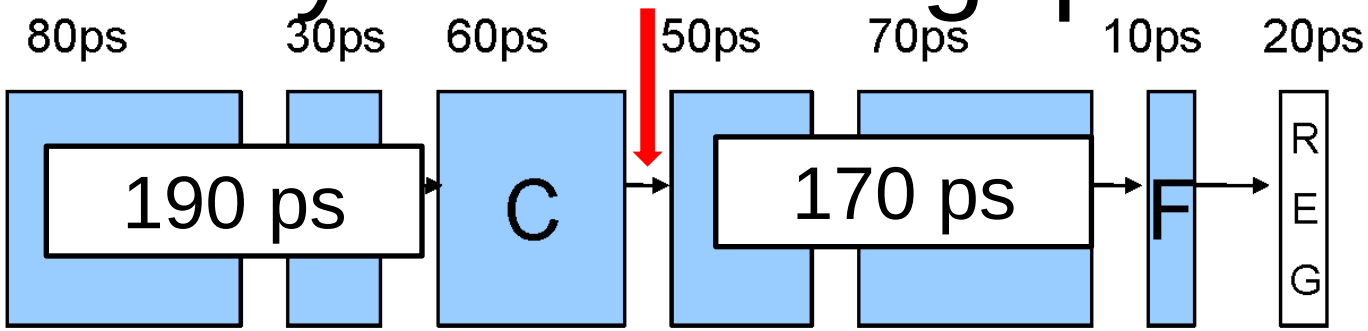
- Latency – Total time to perform a single operation from start to end.
- Throughput – operations / latency ration or GOPS, giga-operations per second.
- The clock cycle is always bounded by the slowest operation.

Latency and Throughput



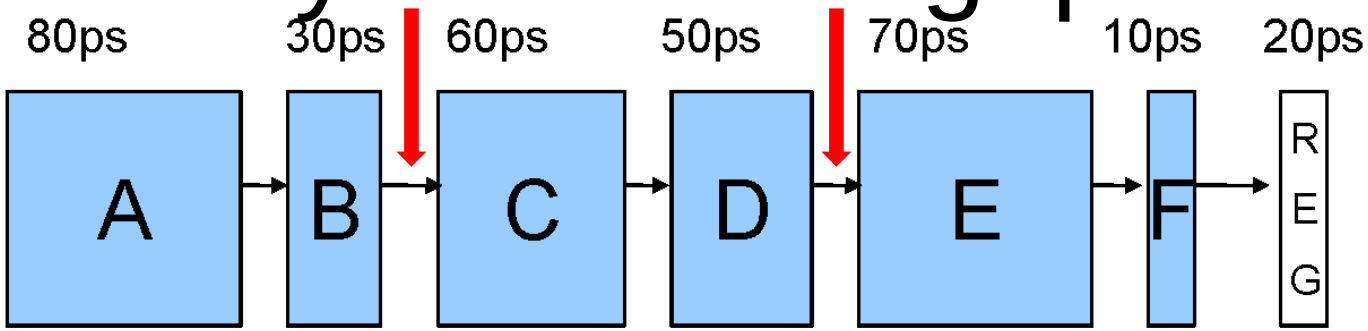
- $throughput = \frac{1\ operation}{operation\ time/\ cycle\ time} * 1000$
- 1000 for normalize it.
- The shorter the cycle time, the larger the throughput
- Cycle time ≥ slowest unit.

Latency and Throughput



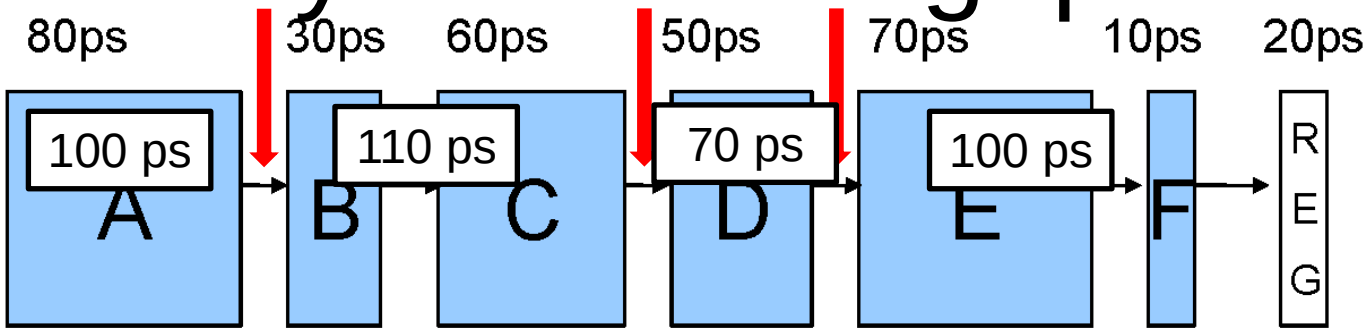
- How to maximize throughput using an additional register?
  - Put it between C and D
- $X_1 = total\ time\ of\ left\ side\ units$
- $X_2 = total\ time\ of\ right\ side\ units$
- Place the register that will give the $\min\max(X_1, X_2)$ time

Latency and Throughput



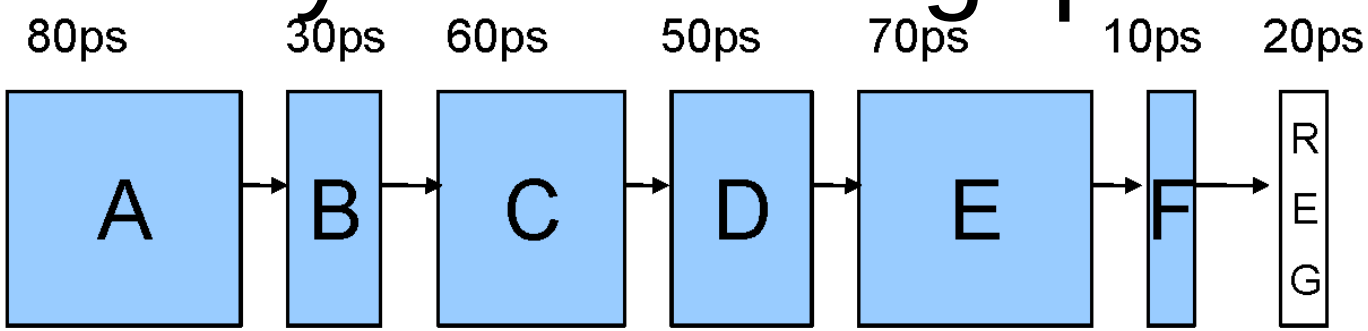
- How to maximize throughput using an additional register?
  - Put it between C and D
- How to maximize throughput using 2 additional registers?
  - AB, CD, EF
- $X_1 = total\ time\ of\ left\ side\ units$
- $X_2 = total\ time\ of\ middle\ part\ units$
- $X_3 = total\ time\ of\ right\ side\ units$
- Place the register that will give the $\min\max(X_1, X_2, X_3)$ time

Latency and Throughput



- How to maximize throughput using an additional register?
  - Put it between C and D
- How to maximize throughput using 2 additional registers?
  - AB, CD, EF
- How to maximize throughput using 3 additional registers?
  - A, BC, D, EF
- What is the cycle time, latency and throughput in that case?
  - Cycle time: 110ps, latency: 440ps,
    throughput: (1/110)*1000 = 9.09 GOPS

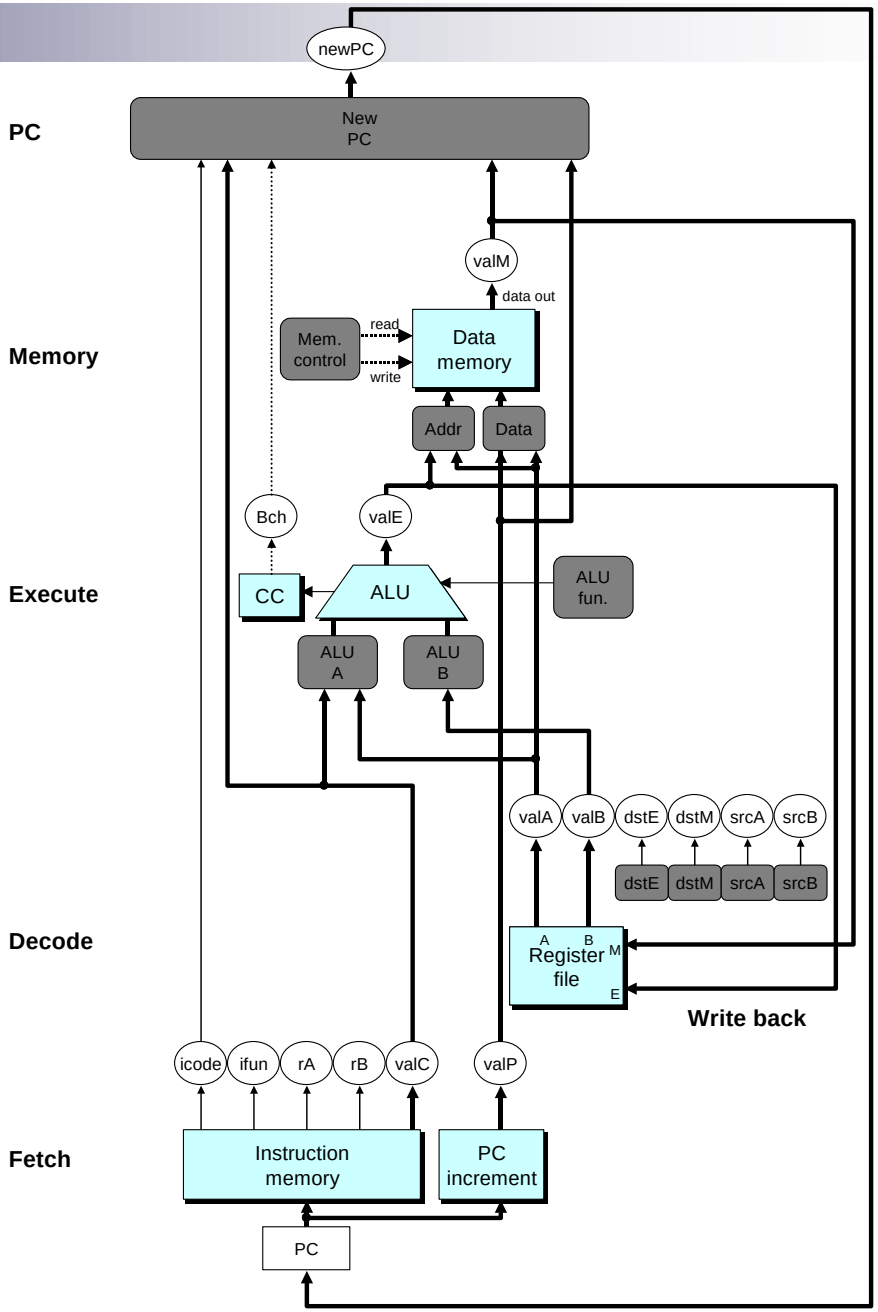
Latency and Throughput



- How to maximize throughput using an additional register?
  - Put it between C and D
- How to maximize throughput using 2 additional registers?
  - AB, CD, EF
- How to maximize throughput using 3 additional registers?
  - A, BC, D, EF
- What is the cycle time, latency and throughput in that case?
  - Cycle time: 110ps, latency: 440ps,
    throughput: (1/110)*1000 = 9.09 GOPS
- How to get max throughput with min stages?
  - 5 stages, since we cannot go lower than 100ps for a clock cycle

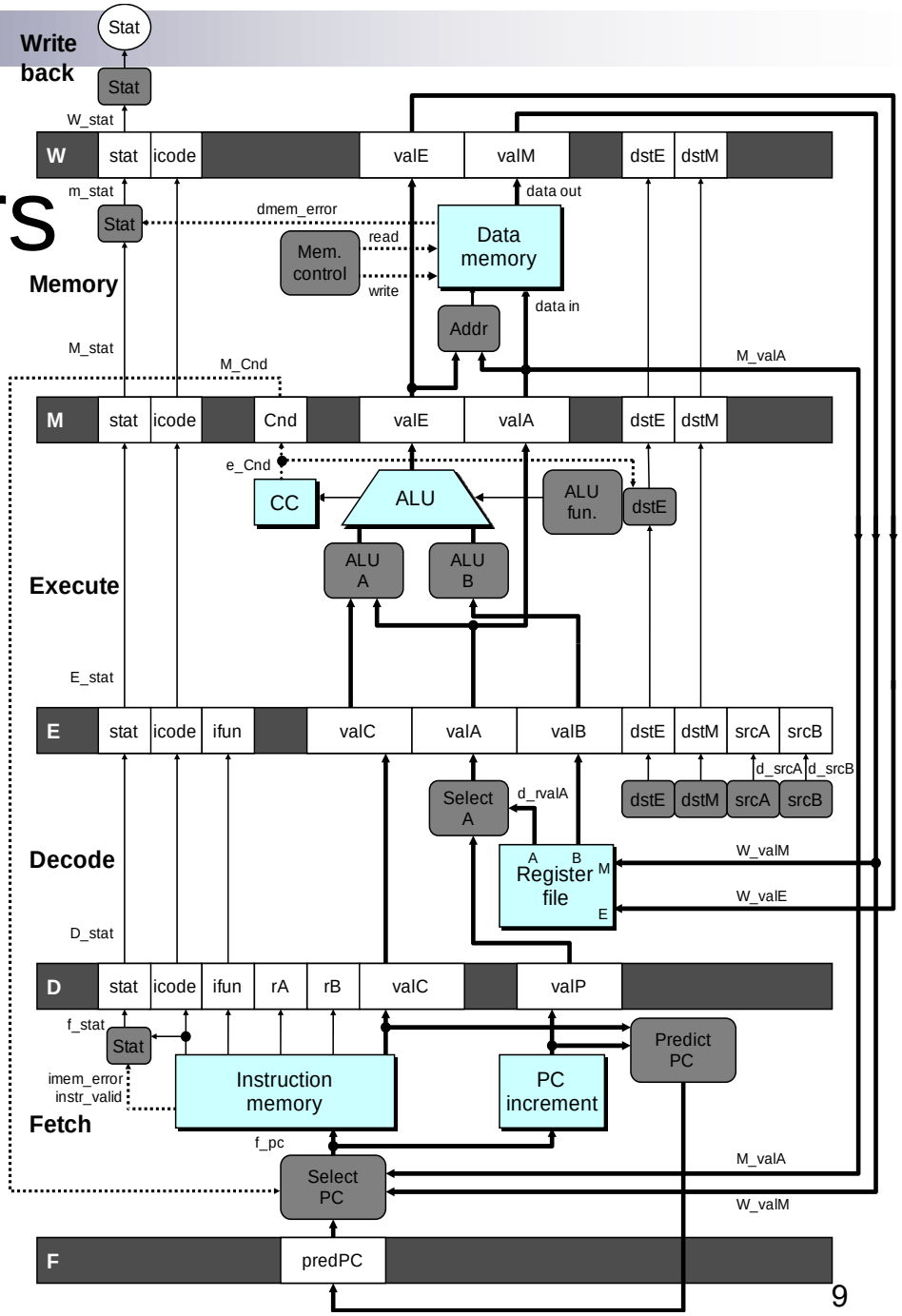
SEQ Hardware (Reminder)

- At each cycle, only a single instruction is processed
- Let's add pipeline registers in order to increase throughput



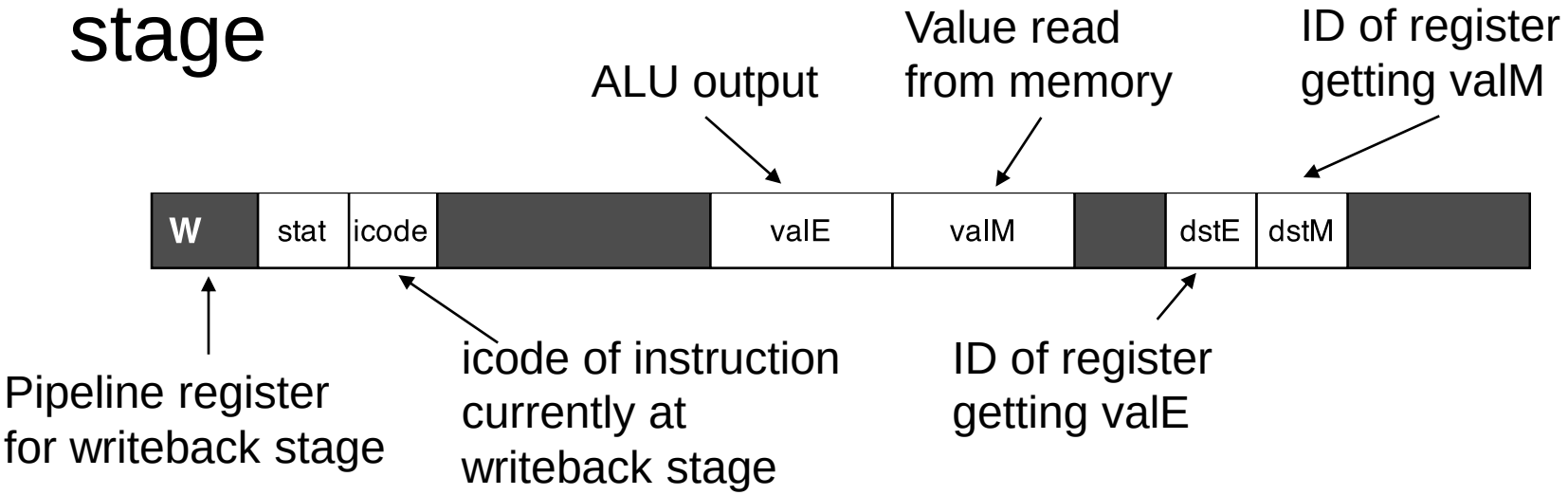
Pipeline Registers

- Select A
  - Since only jxx and call need valP at further stages (E and M, resp.)
- Predicted PC
  - Compute next PC value as first step of instruction execution



Pipeline Registers – A Closer Look

- Each stage has its pipeline register
  - F for fetch, D for decode, etc.
- A pipeline register holds data associated with the instruction being processed at this stage



Data Hazards

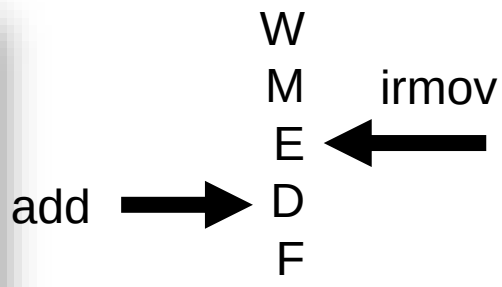
- Data dependencies in processors
- Results from once instruction being used as an operand for another (nop example)
- Stalling / forwarding / load-use hazard

```
irmovq   $50,%rax

addq     %rax,%rbx

mrmovq   100(%rbx),%rdx
```

Data Hazards

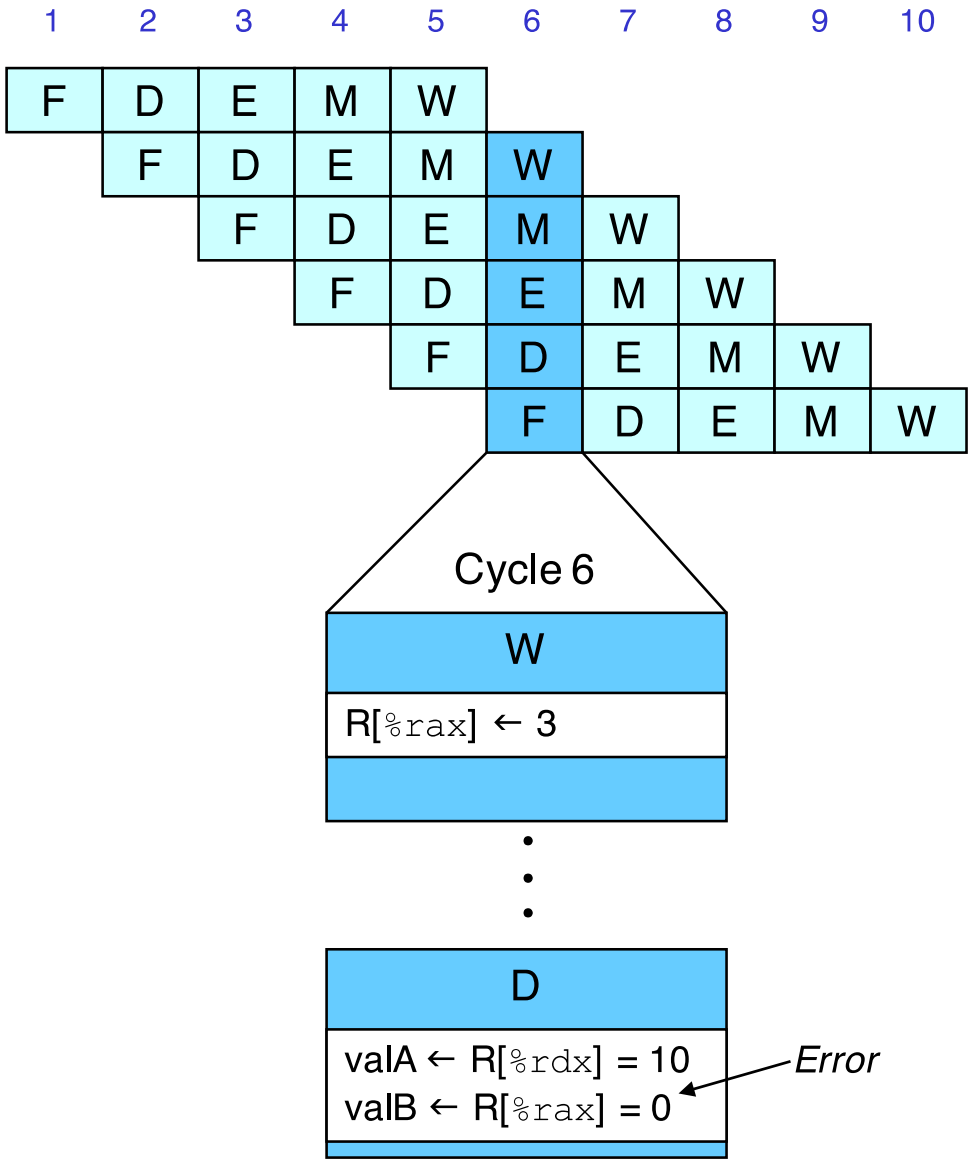
- Data dependencies in processors
- Results from once instruction being used as an operand for another (nop example)
- Stalling / forwarding / load-use hazard

```
irmovq   $50,%rax

addq     %rax,%rbx

mrmovq   100(%rbx),%rdx
```



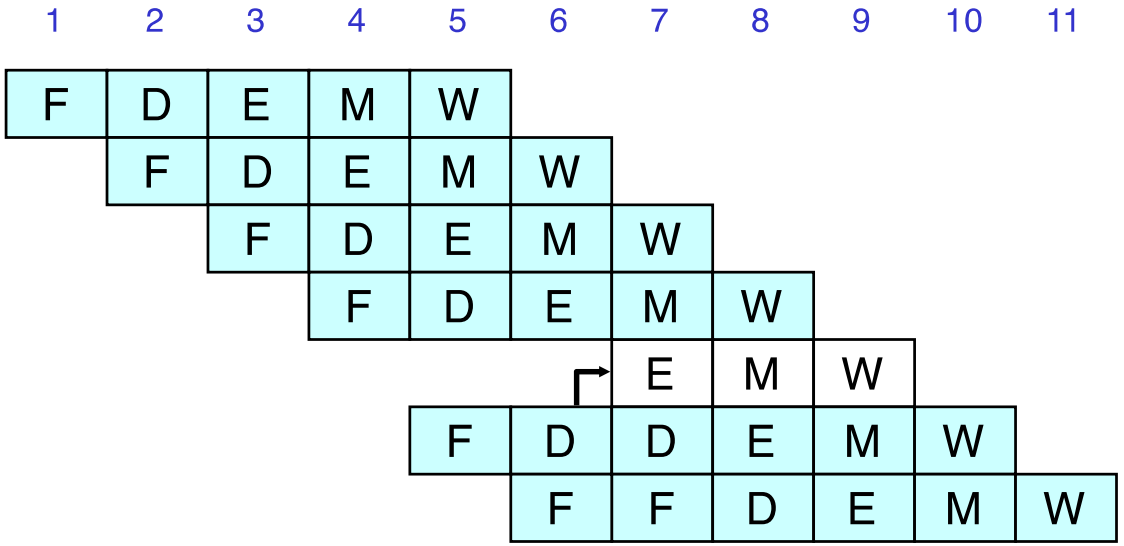
Data Dependencies: 2 nop's

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



Stalling Solution

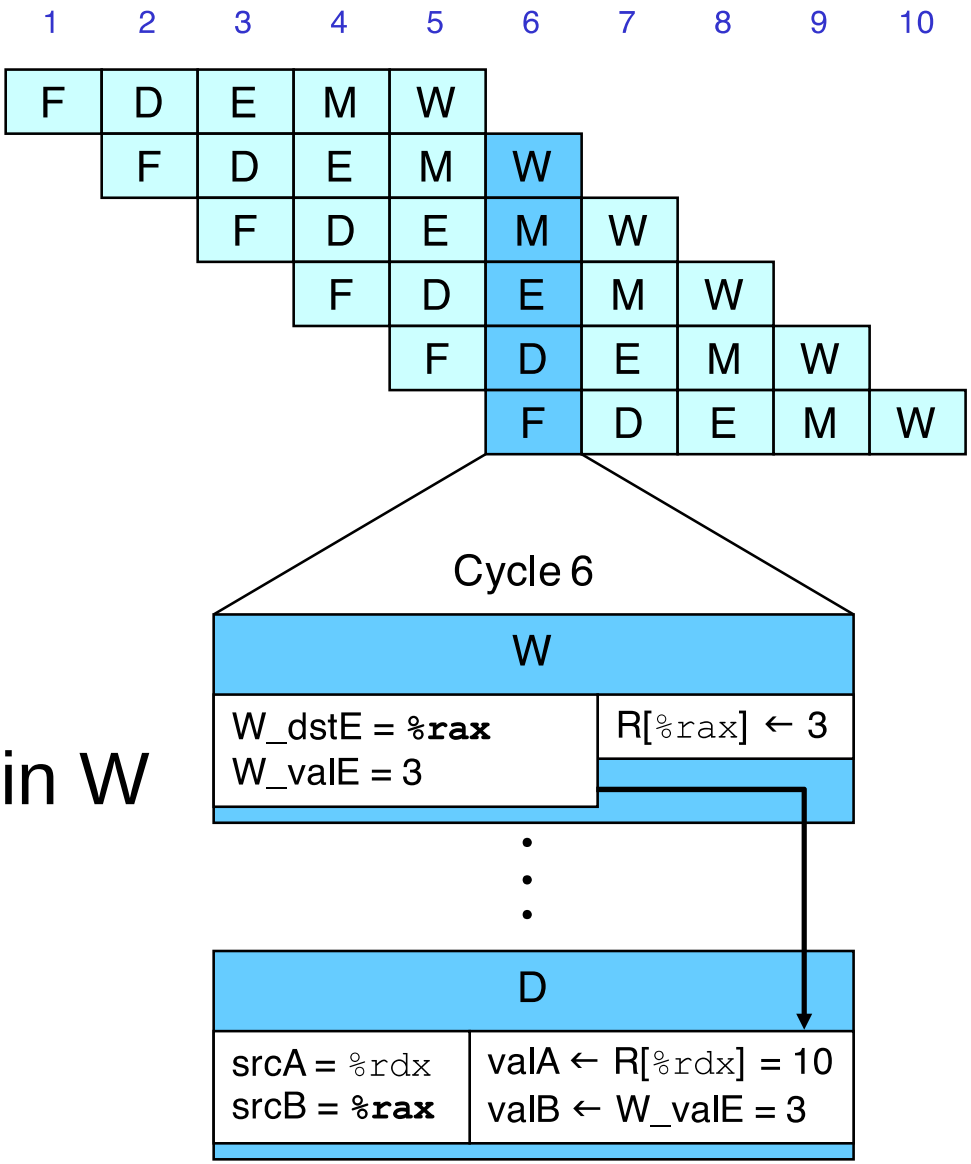
```
# prog2
0x000:  irmovq $10,%rdx
0x00a:  irmovq  $3,%rax
0x014:  nop
0x015:  nop
        bubble
0x016:  addq %rdx,%rax
0x018:  halt
```



- If instruction follows too closely after one that writes register, slow it down
- Hold instruction in decode
- Dynamically inject nop into execute stage

Data Forwarding Solution

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



- irmovq in write-back stage
- Destination value in W pipeline register
- Forward as valB for decode stage

Control Logic for Forwarding

- 5 forwarding resources

Source description	Data word	(Dest.) Register ID
ALU output	e_valE	E_dstE
Value read from memory	m_valM	M_dstM
Forward ALU output of instruction currently at memory stage	M_valE	M_dstE
Forward memory output of instruction currently at writeback stage	W_valM	W_dstM
Forward ALU output of instruction currently at writeback stage	W_valE	W_dstE

Control Logic for Forwarding (2)

- HCL implementation for operand valA:

```
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 writeback
   d_srcA == W_dstE : W_valE; # Forward valE from writeback
   1 : d_rvalA; # Use value read from register file
];
```

- Give priority to the earliest forwarding resource
  - Holds latest instruction setting the register

Control Logic for Forwarding (2)

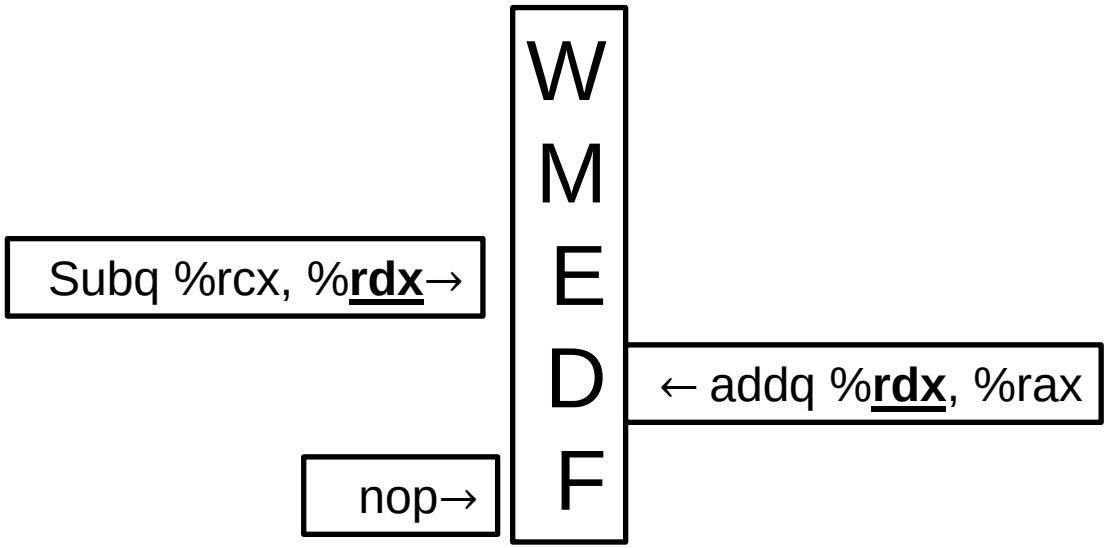
- HCL implementation for operand valA:

```
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 writeback
   d_srcA == W_dstE : W_valE; # Forward valE from writeback
   1 : d_rvalA; # Use value read from register file
];
```

- Give priority to the earliest forwarding resource
  - Holds latest instruction setting the register

Code:

subq %rcx, %**rdx**
addq %**rdx**, %rax
nop



Control Logic for Forwarding (3)

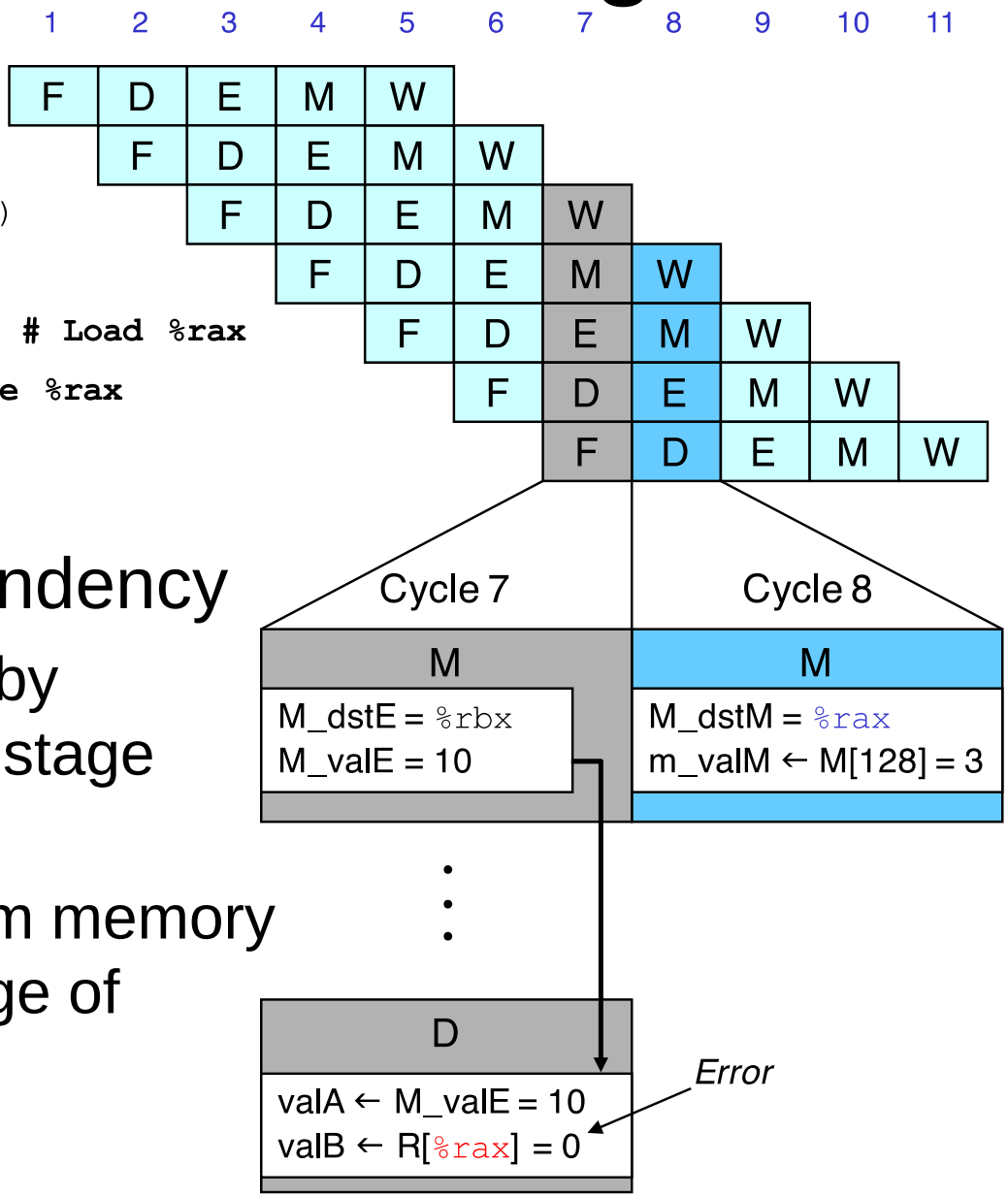
- HCL implementation for operand valB:

```
word d_valB = [
   d_srcB == e_dstE : e_valE; # Forward valE from execute
   d_srcB == M_dstM : m_valM; # Forward valM from memory
   d_srcB == M_dstE : M_valE; # Forward valE from memory
   d_srcB == W_dstM : W_valM; # Forward valM from writeback
   d_srcB == W_dstE : W_valE; # Forward valE from writeback
   1 : d_rvalB; # Use value read from register file
];
```

- As before, give priority to the earliest forwarding resource

Limitation of Forwarding

```
# prog5
0x000:  irmovq $128,%rdx
0x00a:  irmovq  $3,%rcx
0x014:  rmmovq %rcx,  0(%rdx)
0x01e:  irmovq $10,%rbx
0x028:  mrmovq 0(%rdx),%rax  # Load %rax
0x032:  addq %rbx,%rax  # Use %rax
0x034:  halt
```



- Load-use dependency
  - Value needed by end of decode stage in cycle 7
  - Value read from memory in memory stage of cycle 8

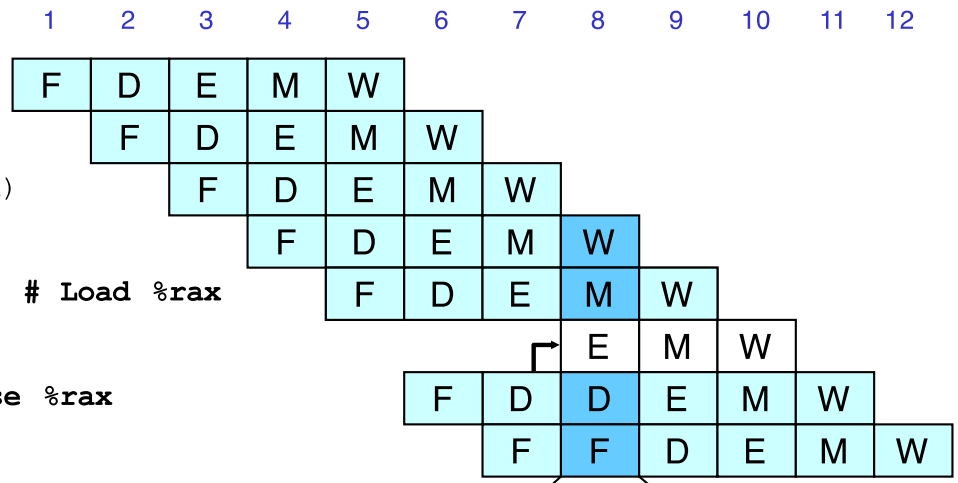
Detecting a Load/Use Hazard

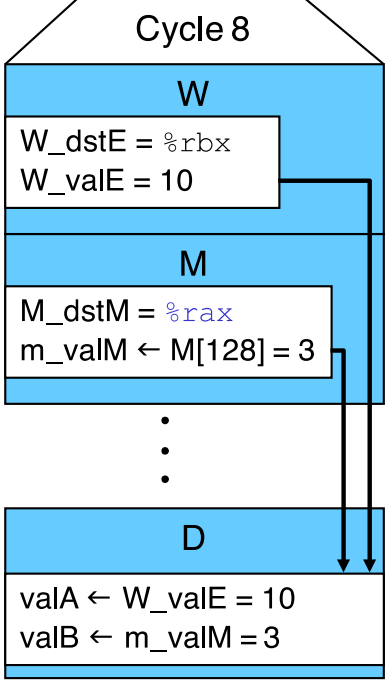
- Only mrmovq,popq read from memory
- Therefore, if:
  - Either mrmovq,popq is in the execute stage
  - An instruction requiring the destination register is in the decode stage
- In HCL:

```
E_icode in {IMRMOVQ, IPOPQ} &&
                    E_dstM in {d_srcA,d_srcB}
```

Avoiding Load/Use Hazard

```
# prog5
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax  # Load %rax
     bubble
0x032: addq %rbx,%rax  # Use %rax
0x034: halt
```





- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage

Avoiding Load/Use Hazard (2)

- In case of a load/use hazard:
  - Hold back the instruction in the decode stage
  - On the next cycle, inject a bubble to the execute stage

Pipeline Register				
F	D	E	M	W
stall	stall	bubble	normal	normal

Control Hazards

- PC prediction (Fetch stage)
  - jxx, ret → ?
  - call, jmp → valC
  - Other → valP

Control Hazards

- Branch prediction strategies
  - Always-taken
  - Never-taken
  - Backward taken, forward not taken
- Why are forward branches less common?
- How can we deal with stack prediction (ret)?

Control Logic

- Processing “ret”
  - Must stall until instruction reaches write back
- Load/Use hazard
  - Must stall between read memory and use
- Mispredicted branch
  - Removing instructions from the pipe

- Added the forwarding logic
- 5 forwarding sources