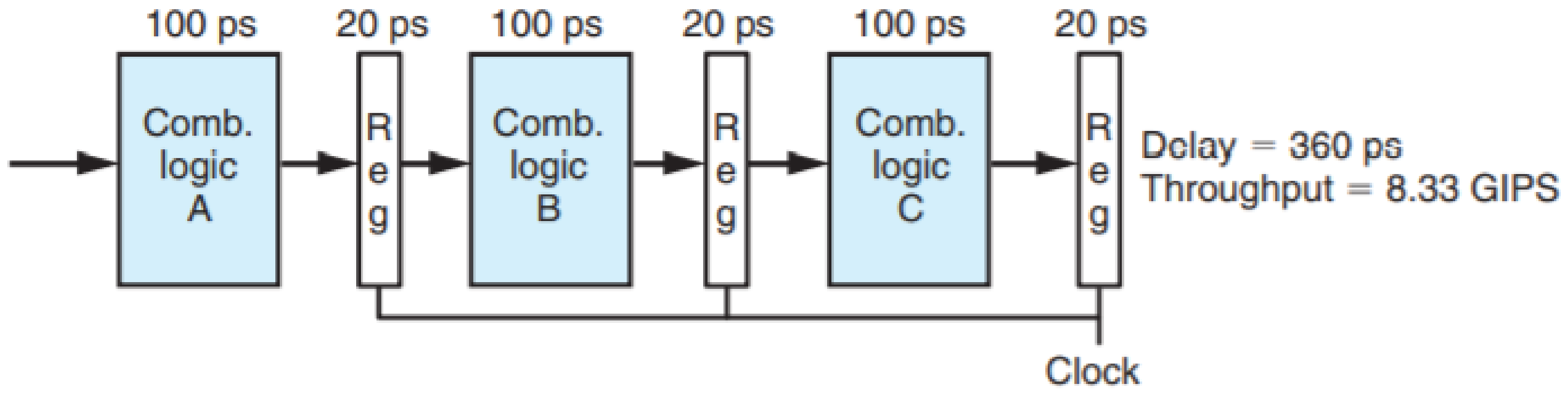
Processor Arch: Pipelined Memory Hierarchy

MADE BY: Zhong Zhineng & Song Yixin

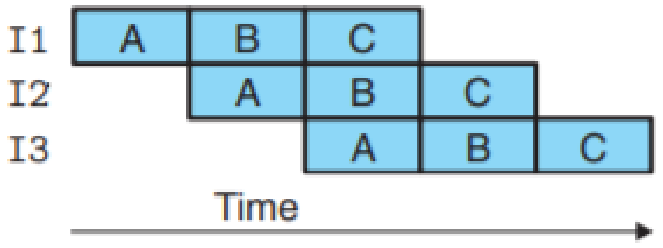
Processor Arch: Pipelined

Pipeline Basics

- Throughput and latency change



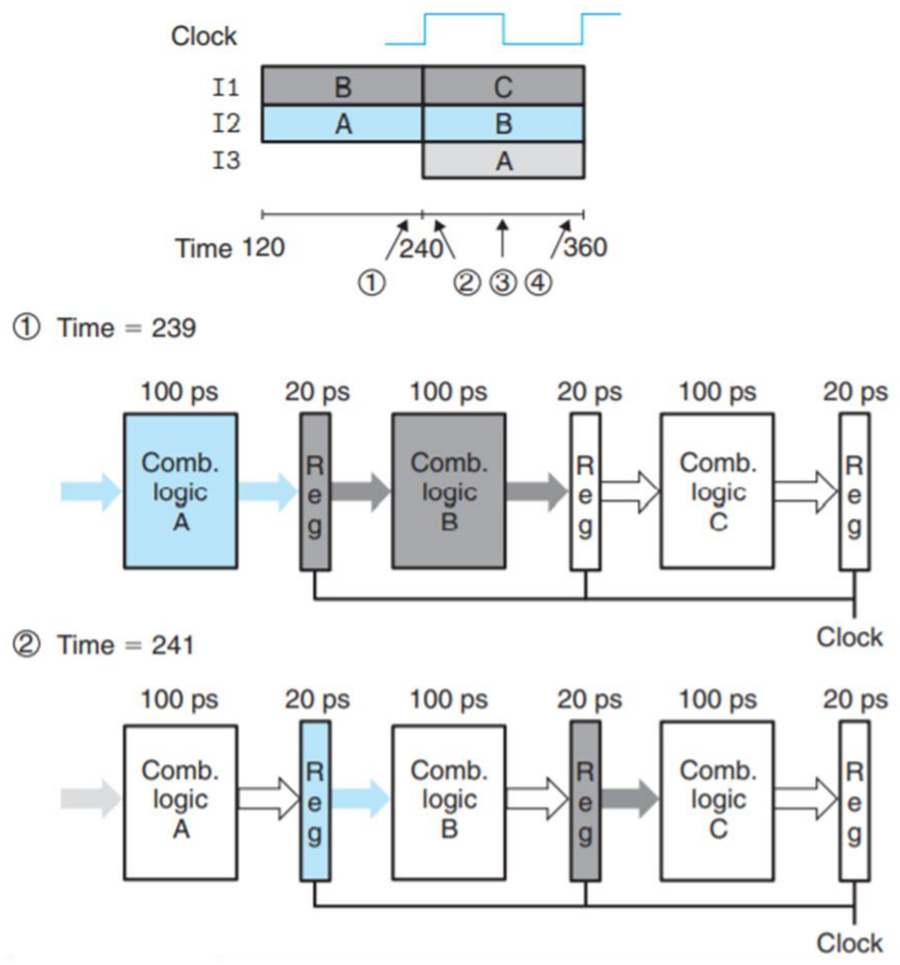
(a) Hardware: Three-stage pipeline

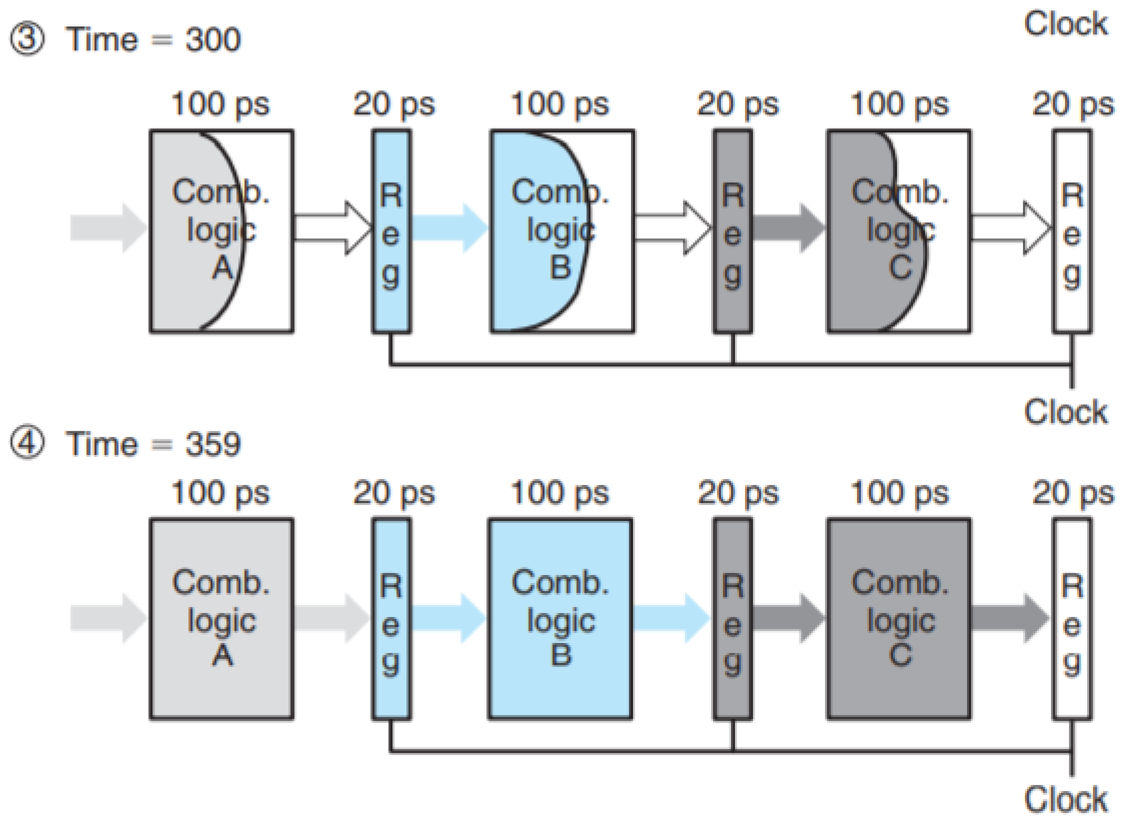


(b) Pipeline diagram

Pipeline Basics

- Critical moments



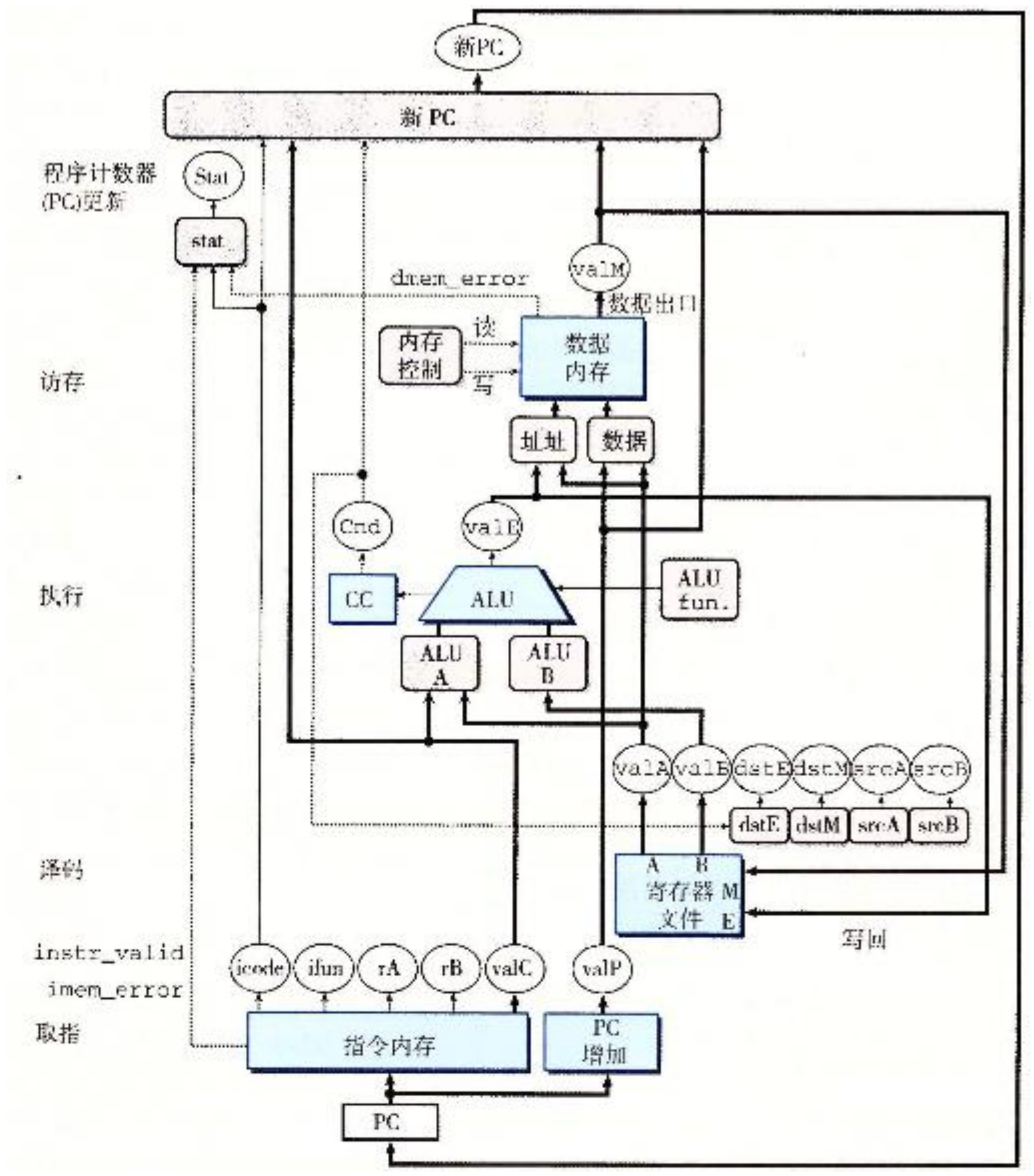


Pipeline Basics

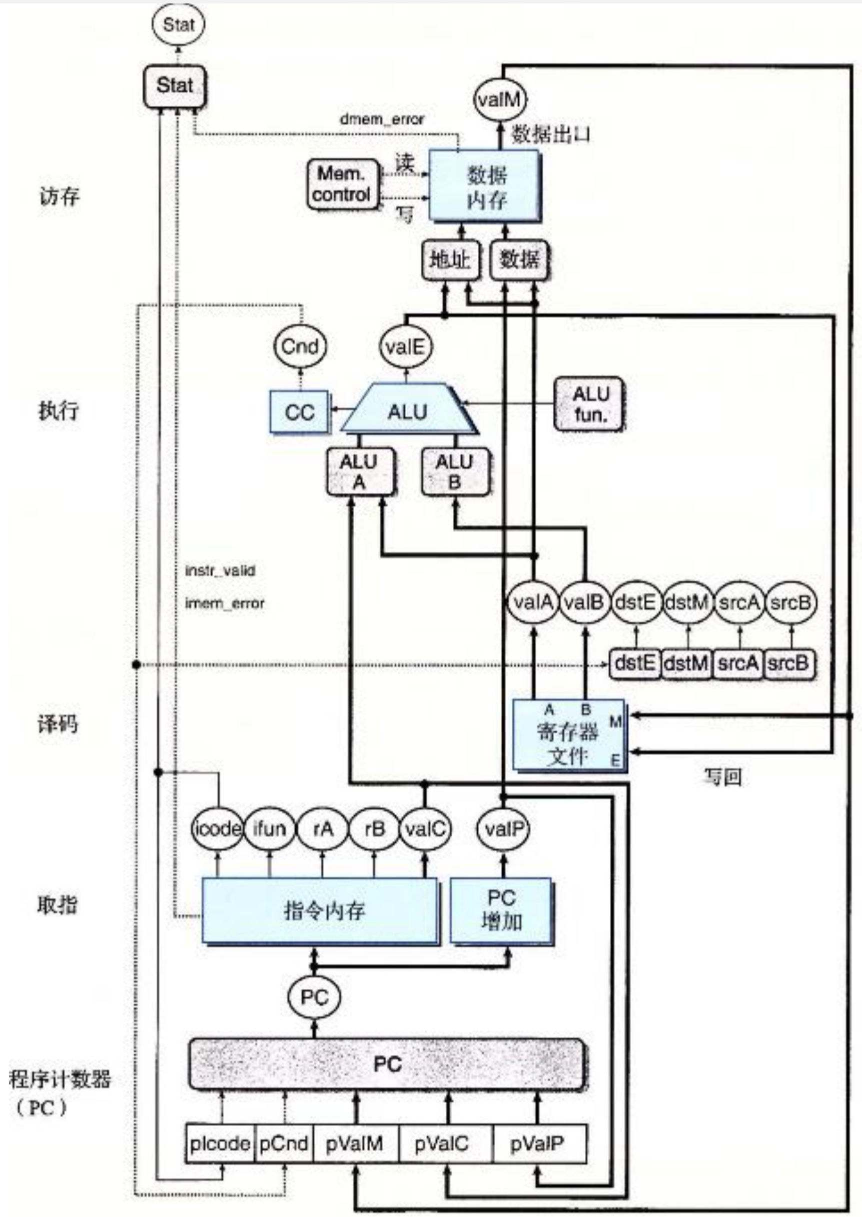
- **Limitations of pipelining**
  - Nonuniform partitioning
  - Decreasing returns to pipeline depth

- **Stages**
  - AMD Zen2(3th Gen Ryzen): 19 stages
  - Intel Ice Lake(10th Gen core): 14-19 stages

SEQ->SEQ+



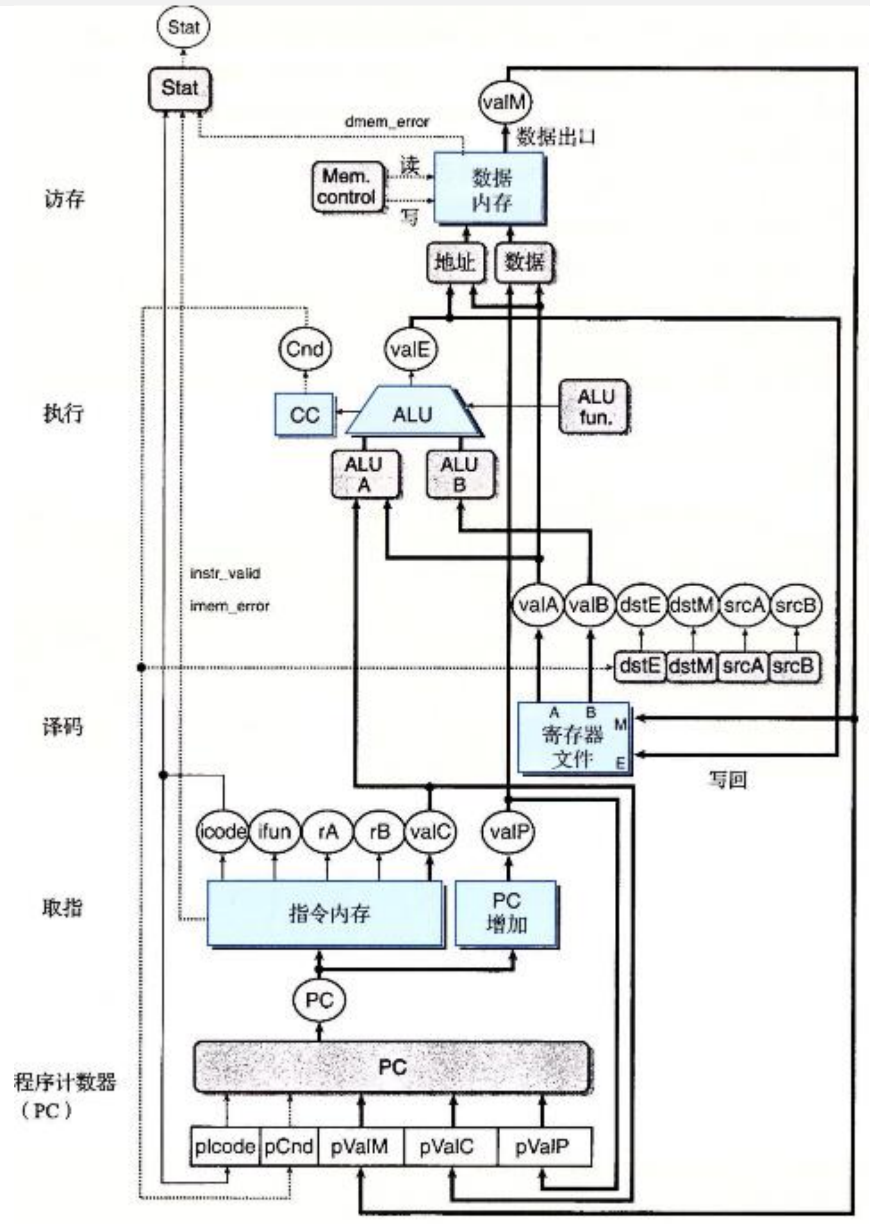
新PC
新 PC
程序计数器 (PC)更新
Stat
stat
dmem_error
valM
数据出口
内存 控制
读
写
数据 内存
访存
地址
数据
Cnd
valE
执行
CC
ALU
ALU fun.
ALU A
ALU B
valA
valB
dstE
dstM
srcA
srcB
dstE
dstM
srcA
srcB
译码
A
B
寄存器 文件
M
E
写回
instr_valid
imem_error
icode
ifun
rA
rB
valC
valP
取指
指令内存
PC 增加
PC



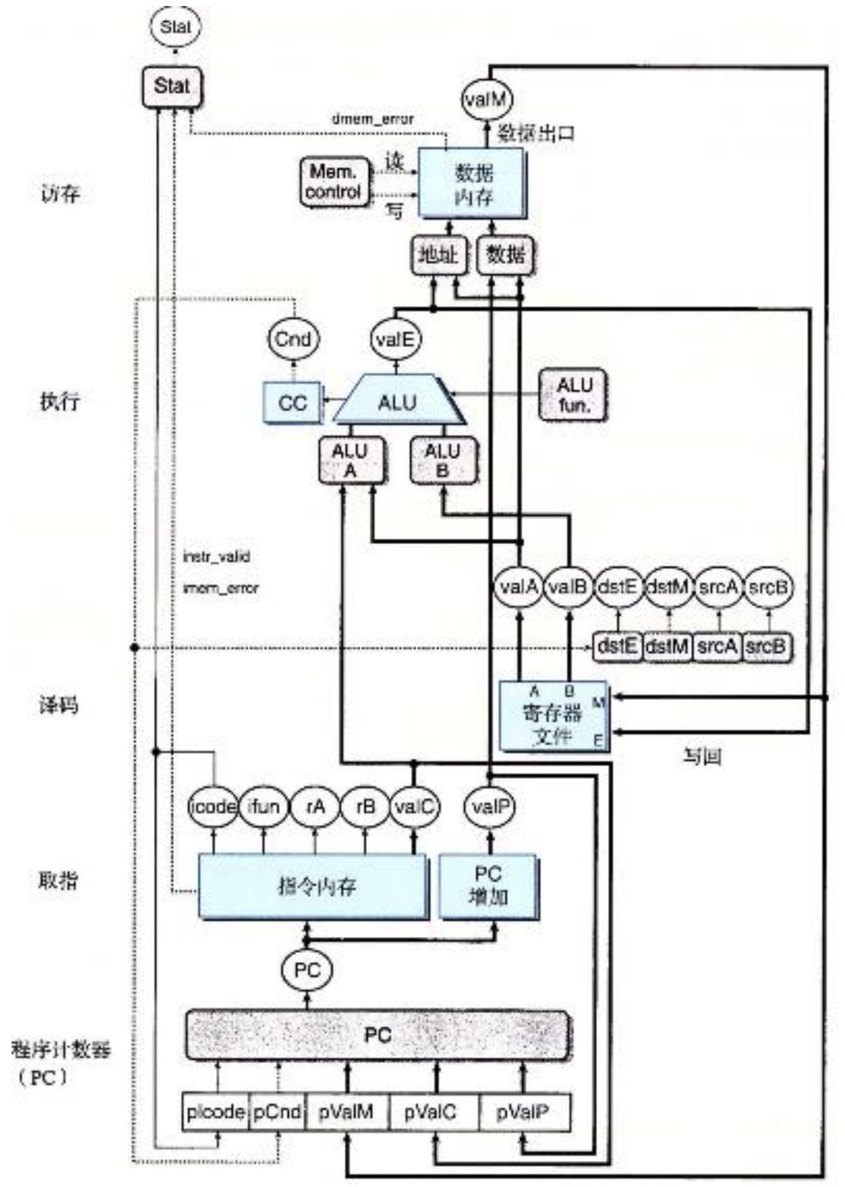
Stat
Stat
valM
dmem_error
数据出口
访存
Mem. control
读
写
数据 内存
地址
数据
Cnd
valE
执行
CC
ALU
ALU fun.
ALU A
ALU B
instr_valid
imem_error
valA
valB
dstE
dstM
srcA
srcB
dstE
dstM
srcA
srcB
译码
A
B
寄存器 文件
M
E
写回
icode
ifun
rA
rB
valC
valP
取指
指令内存
PC 增加
PC
程序计数器 （PC）
PC
pIcode
pCnd
pValM
pValC
pValP

SEQ->SEQ+

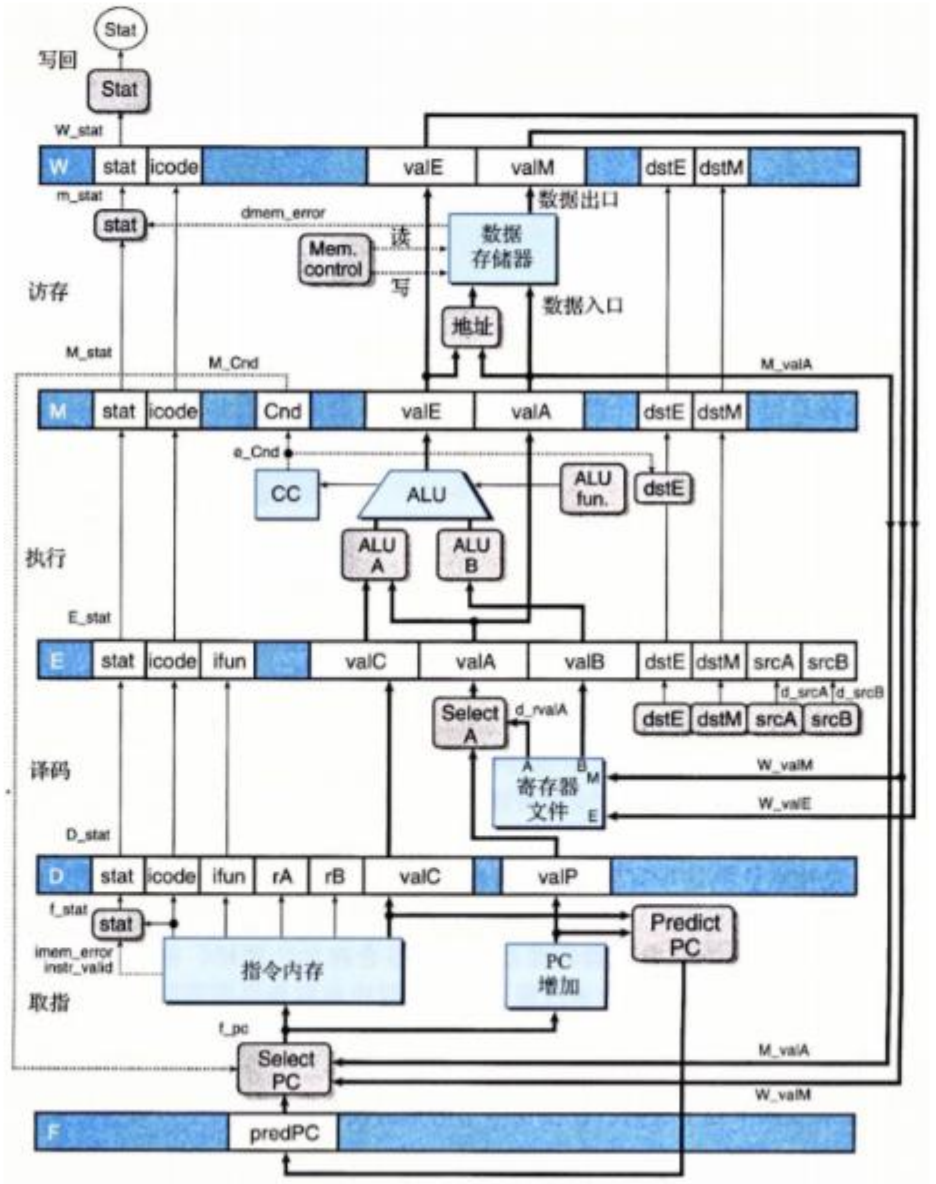
- Circuit retiming
- PC update stage
- No hardware PC registers



SEQ+->PIPE-



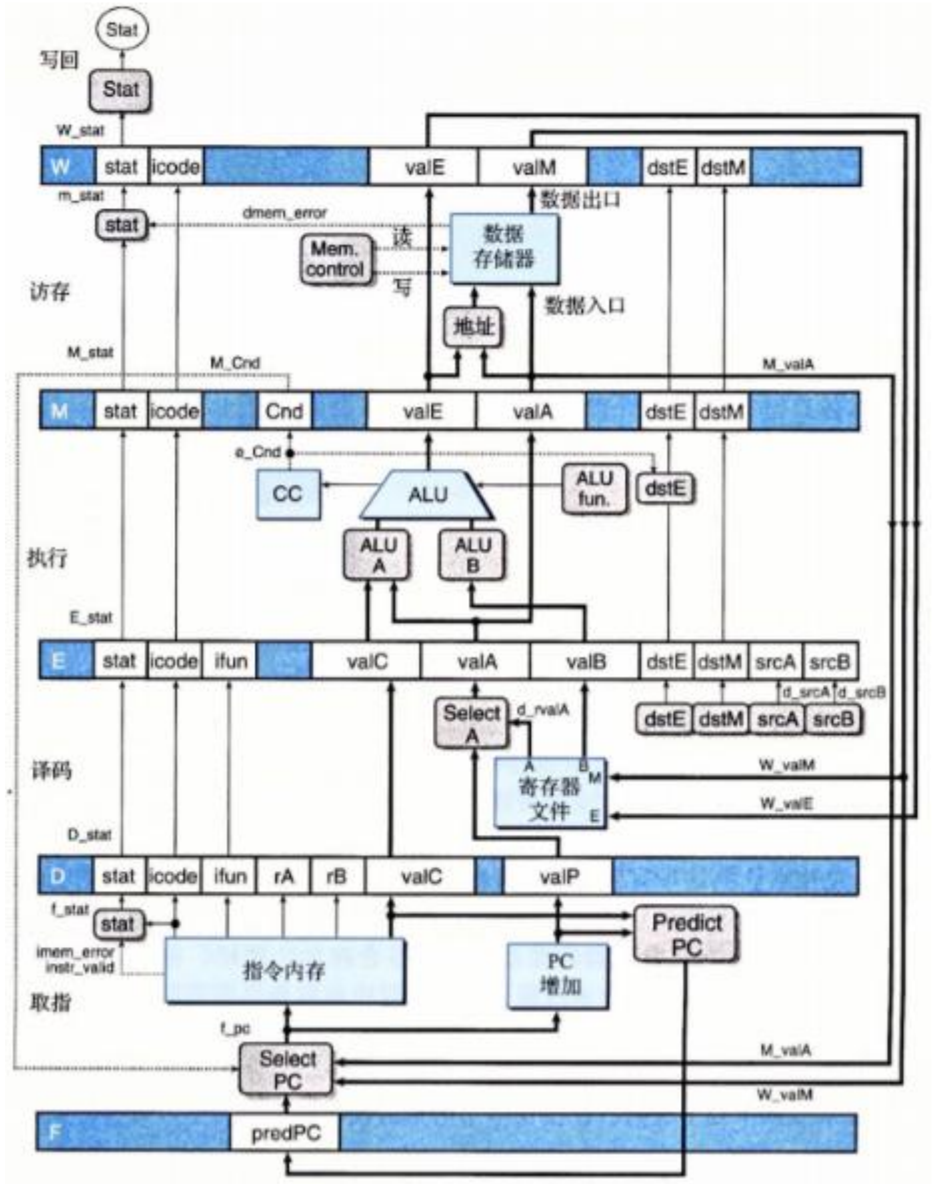
Stat
Stat
dmem_error
valM
数据出口
Mem. control
读
写
数据内存
地址
数据
访存
Cnd
valE
CC
ALU
ALU fun.
执行
ALU A
ALU B
instr_valid
imem_error
valA
valB
dstE
dstM
srcA
srcB
译码
寄存器文件
写回
icode
ifun
rA
rB
valC
valP
取指
指令内存
PC增加
PC
程序计数器（PC）
pIcode
pCnd
pValM
pValC
pValP



Stat
写回
Stat
W_stat
W
stat
icode
valE
valM
dstE
dstM
m_stat
stat
dmem_error
数据出口
Mem. control
读
写
数据存储器
访存
地址
数据入口
M_stat
M_Cnd
M_valA
M
Cnd
valA
e_Cnd
CC
ALU
ALU fun.
dstE
执行
ALU A
ALU B
E_stat
E
ifun
valC
valB
srcA
srcB
d_srcA
d_srcB
Select A
d_rvalA
译码
寄存器文件
W_valM
W_valE
D_stat
D
rA
rB
valP
f_stat
Predict PC
imem_error
instr_valid
指令内存
PC增加
取指
f_pc
Select PC
M_valA
W_valM
F
predPC

SEQ+->PIPE-

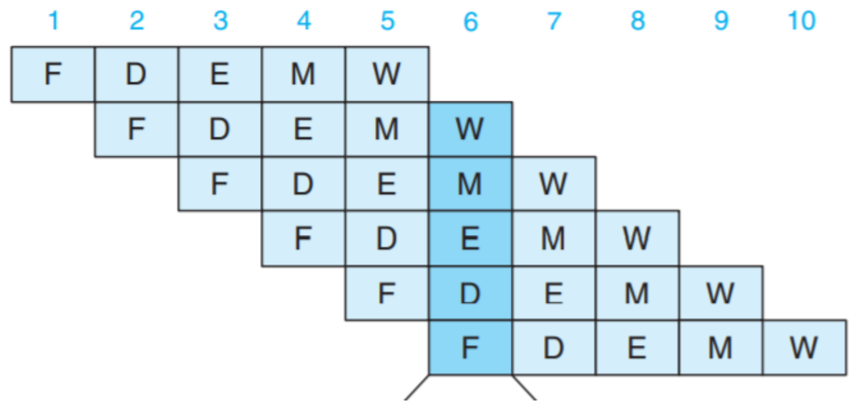
- Adding pipeline registers
- **Select PC**
- **Select A:** Since valP is only used in the Memory period of call and in the Execute period of jXX and both of them do not need valA, Select A module is used to reduce the number of registers.



Problems: Data/Control Dependency

- Data hazard

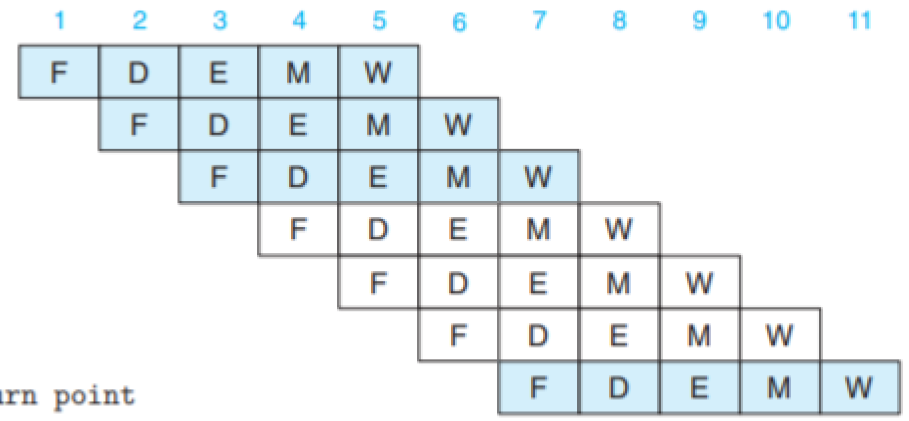
```
# prog2
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



1	2	3	4	5	6	7	8	9	10
F	D	E	M	W					
	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

- Control hazard

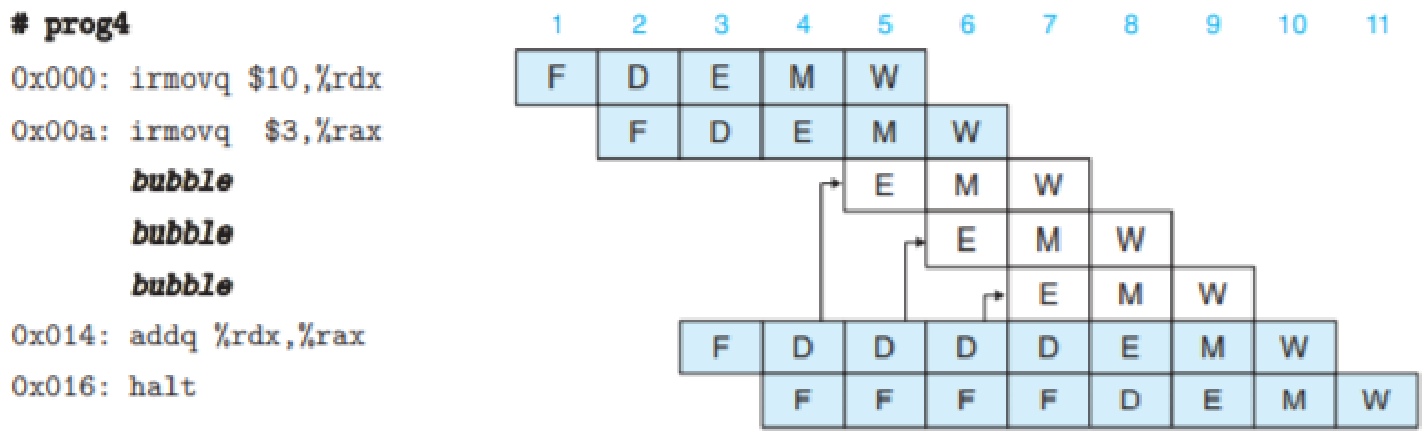
```
# prog7
0x000: irmovq Stack,%edx
0x00a: call proc
0x020: ret
       bubble
       bubble
       bubble
0x013: irmovq $10,%edx # Return point
```



1	2	3	4	5	6	7	8	9	10	11
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		
					F	D	E	M	W	
						F	D	E	M	W

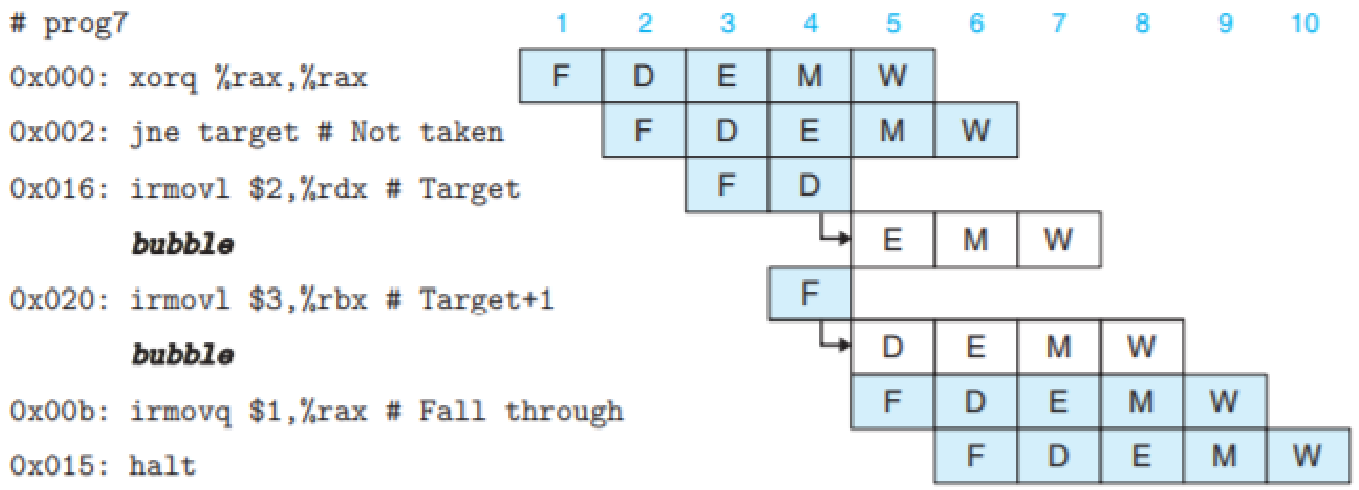
A Simple Solution: Bubbles and Stalls

Handling data hazard



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

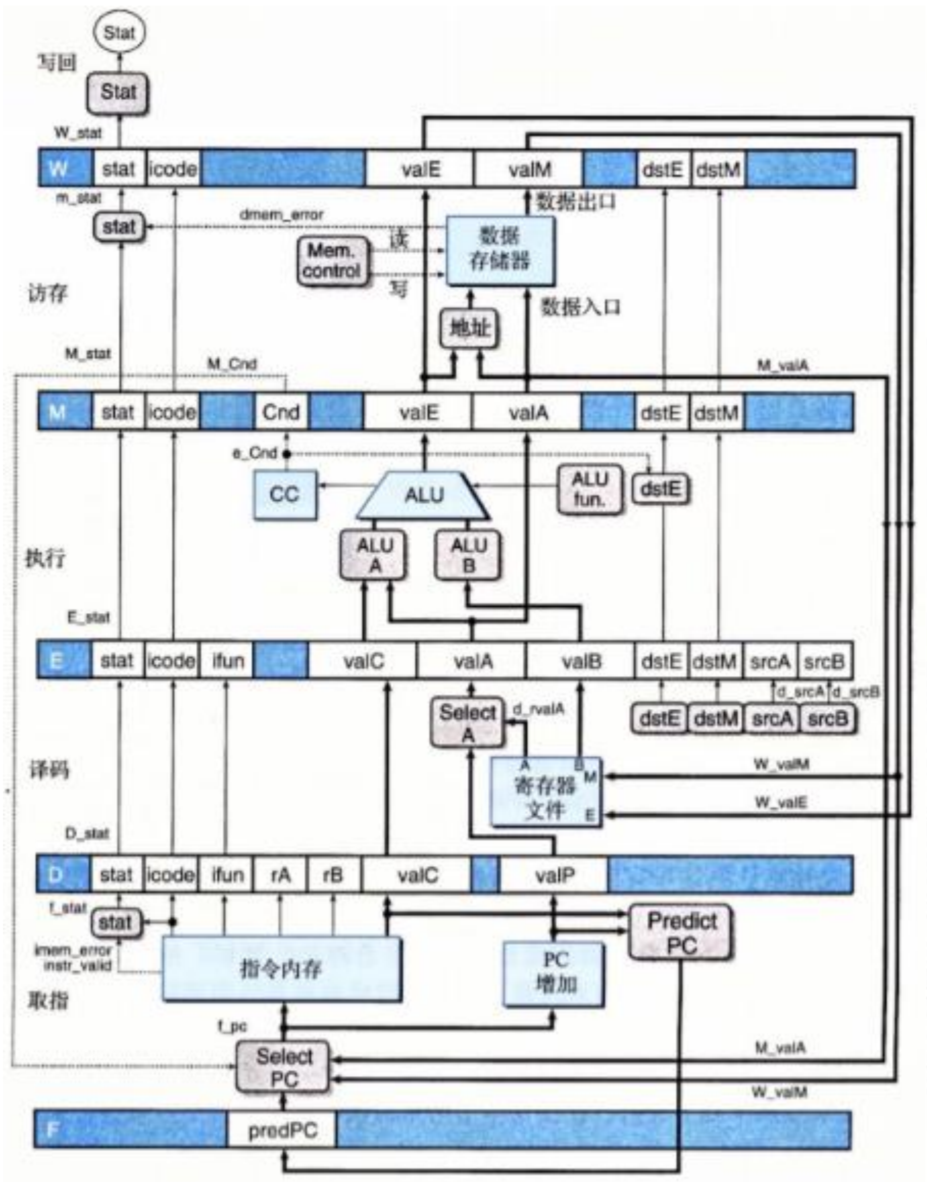
Handling control hazard

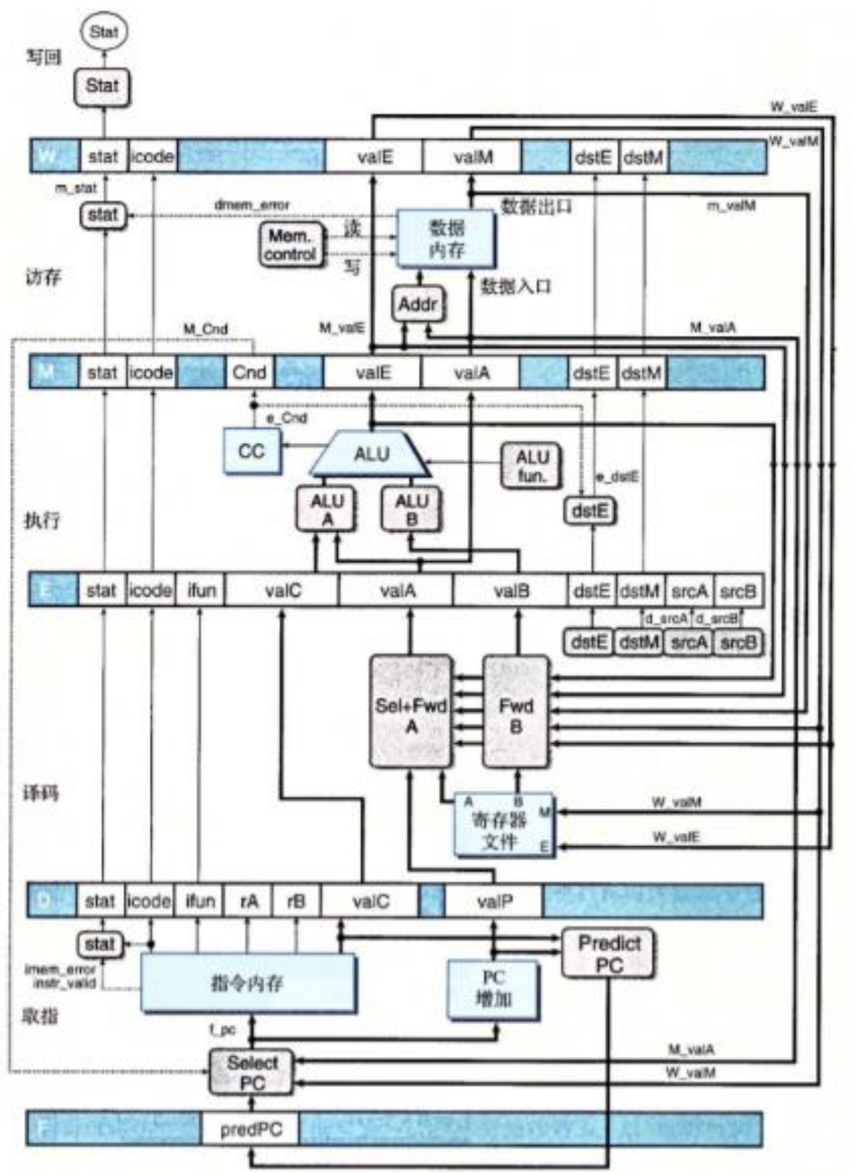


```
# prog7
0x000: xorq %rax,%rax
0x002: jne target # Not taken
0x016: irmovl $2,%rdx # Target
       bubble
0x020: irmovl $3,%rbx # Target+1
       bubble
0x00b: irmovq $1,%rax # Fall through
0x015: halt
```

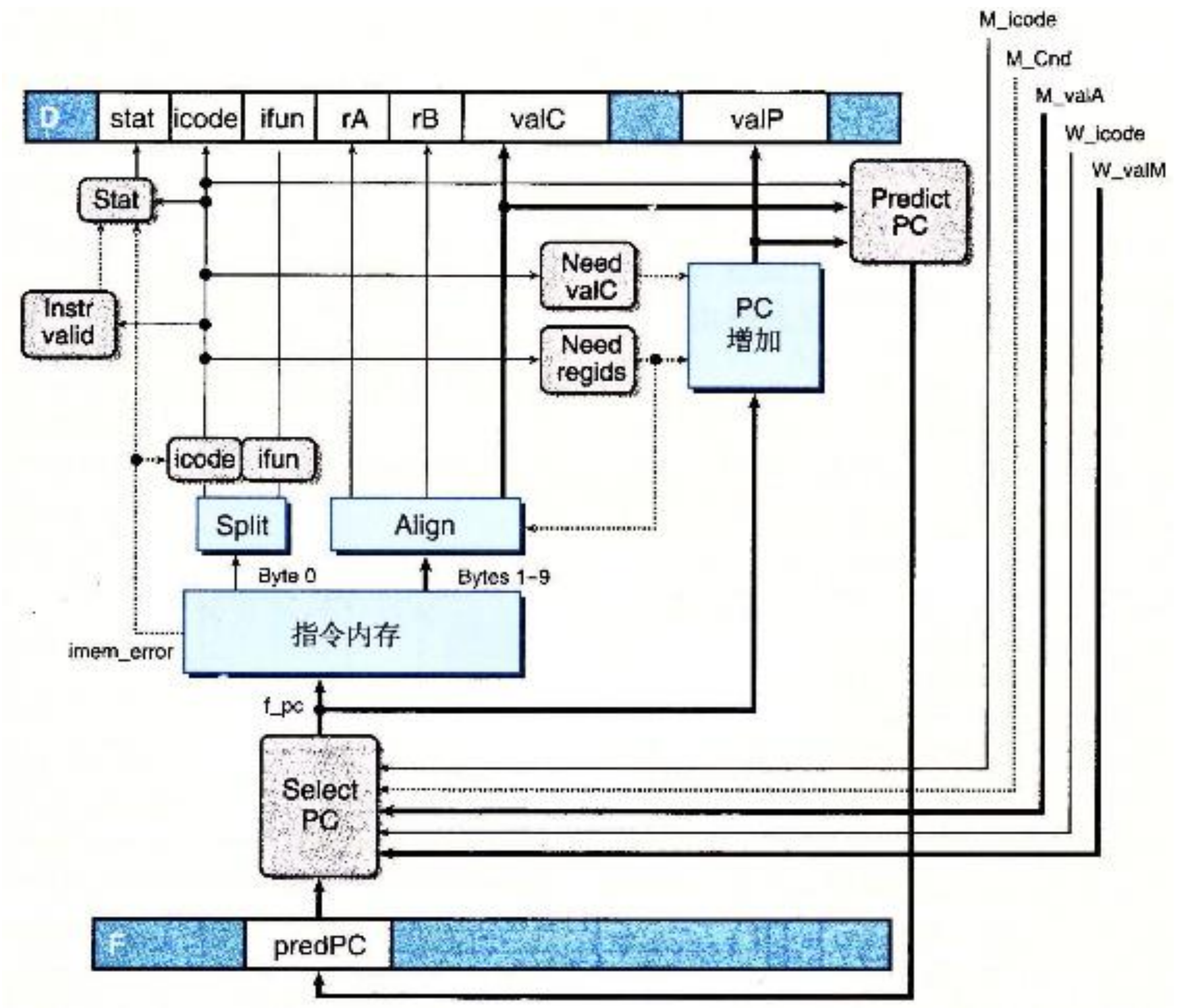
Another Solution: Forward

- Need the data that has not written back to the registers when decoding.
- **Principle: Try to use forward. If failed, use stall.**
- Sel+Fwd A
- Fwd B





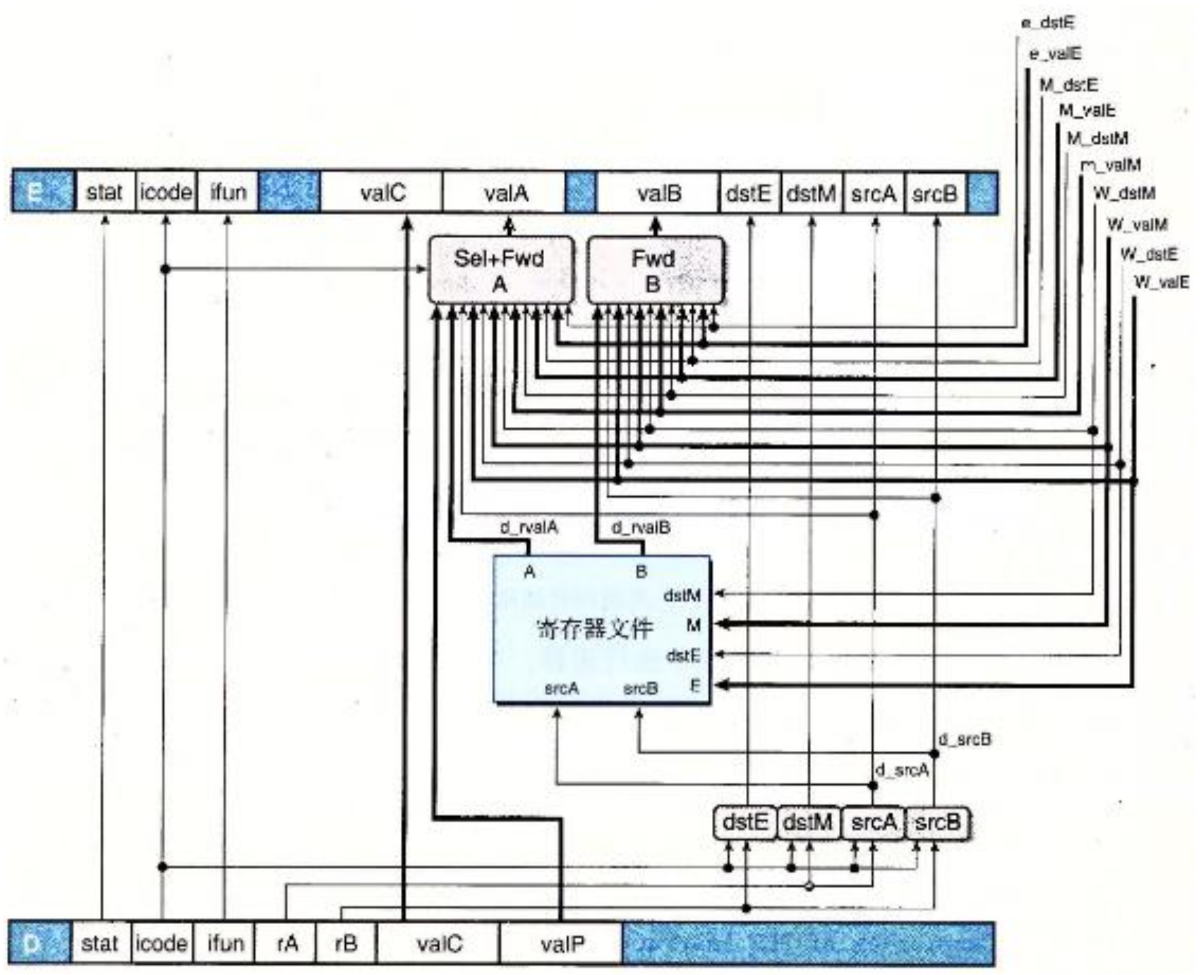
Modified HCL: Select PC & Fetch



HCL:

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

Modified HCL: Decode & Write back



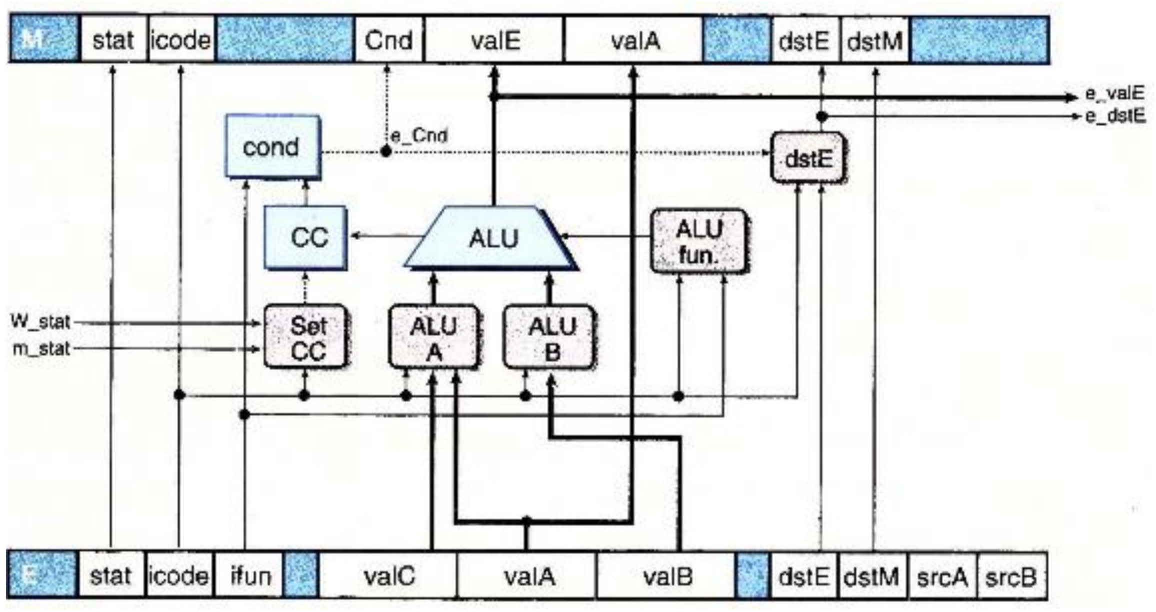
HCL:

```
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
];

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 write back
        d_srcB == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalB;  # Use value read from register file
];
```

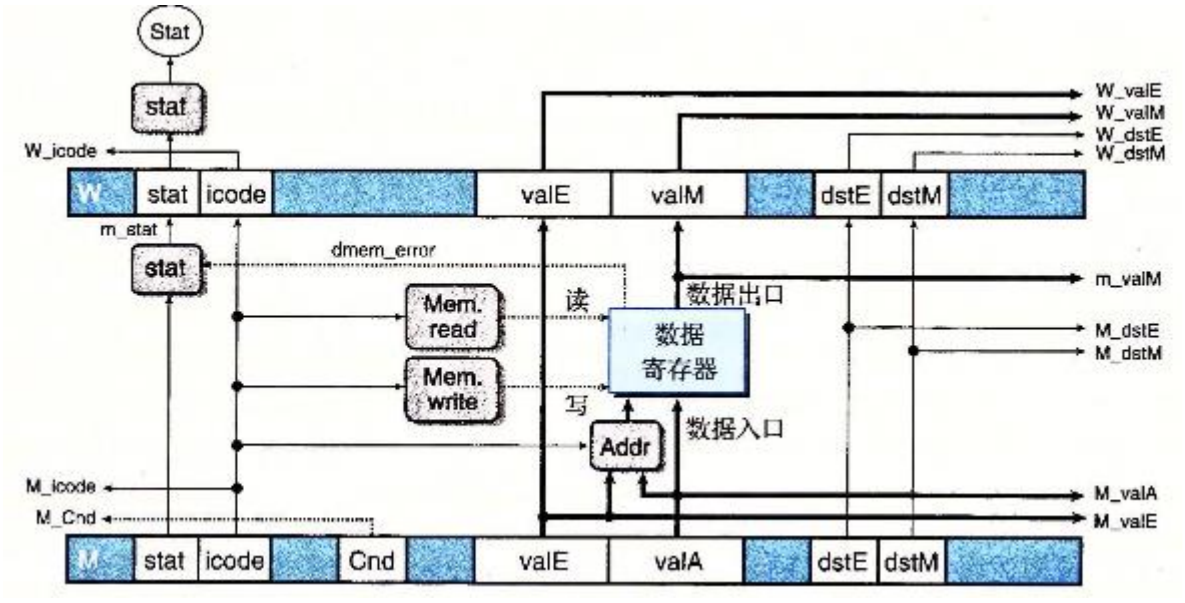
Pay attention to the choice order!

Modified HCL: Execute



HCL: left out

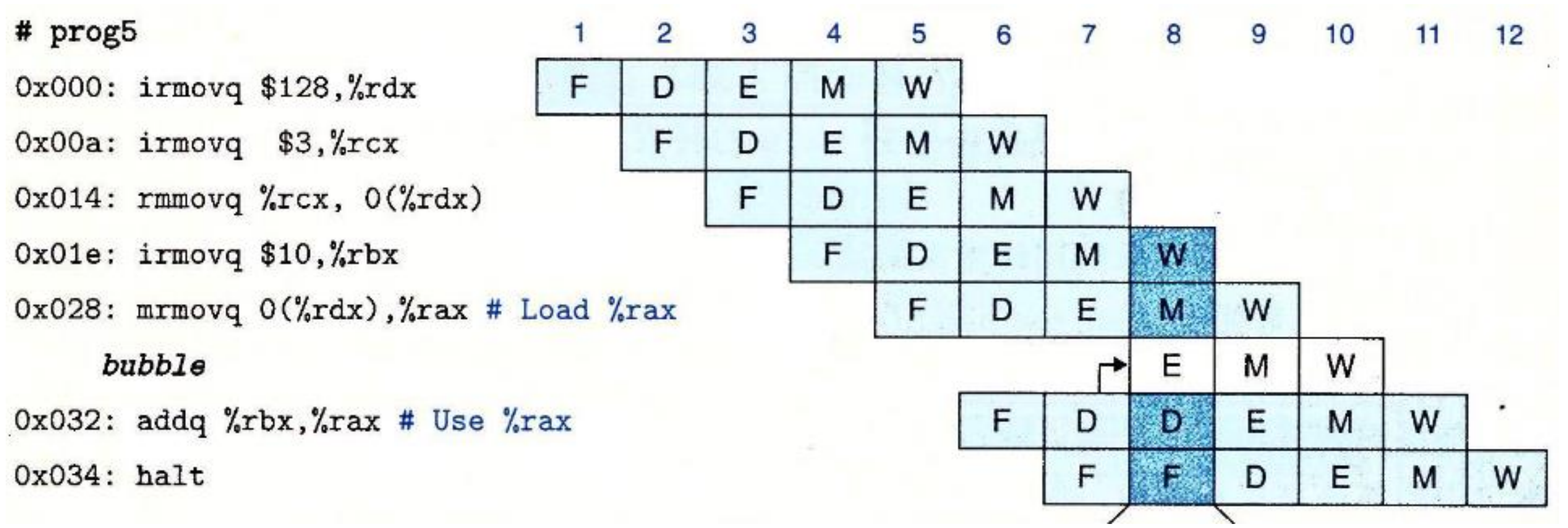
Modified HCL: Memory



HCL: left out

Hazard: Load/Use

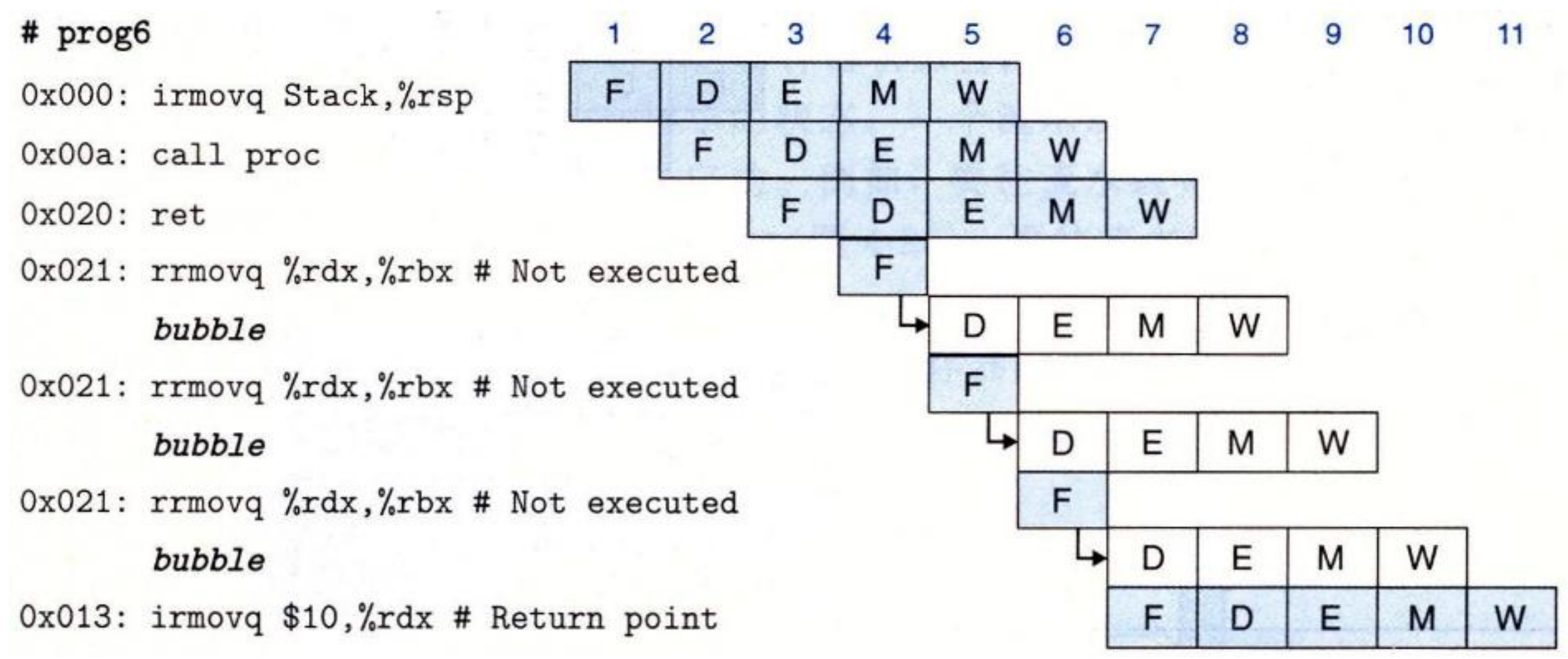
- **You cannot only use forwarding to solve all the problems…**
- Last instruction reads data from memory to a register, and present instruction needs the data in this register.
- Must stall and insert a bubble, then forward from memory stage.



# prog5	1	2	3	4	5	6	7	8	9	10	11	12
0x000: irmovq $128,%rdx	F	D	E	M	W							
0x00a: irmovq $3,%rcx		F	D	E	M	W						
0x014: rmmovq %rcx, 0(%rdx)			F	D	E	M	W					
0x01e: irmovq $10,%rbx				F	D	E	M	W				
0x028: mrmovq 0(%rdx),%rax # Load %rax					F	D	E	M	W			
bubble								E	M	W		
0x032: addq %rbx,%rax # Use %rax						F	D	D	E	M	W	
0x034: halt							F	F	D	E	M	W

Hazard: ret

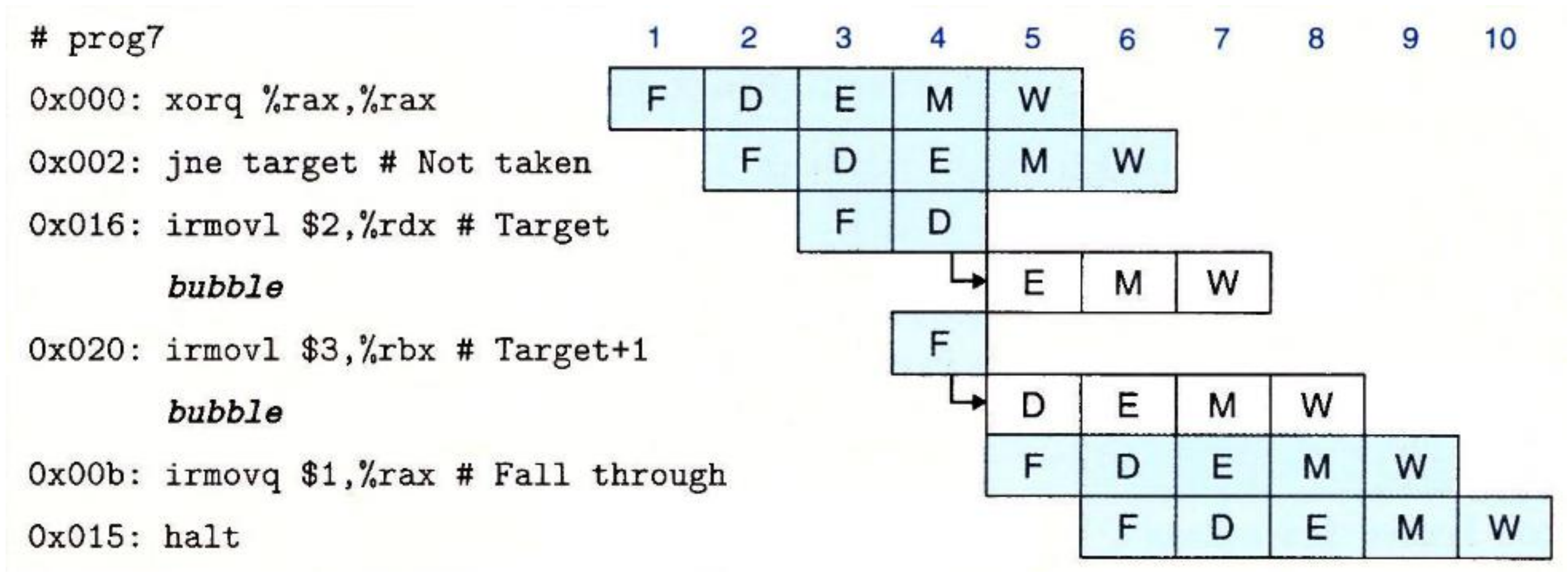
- The PC of the next instruction of ret will be known until memory stage.
- Insert three bubbles.



```
# prog6
0x000: irmovq Stack,%rsp
0x00a: call proc
0x020: ret
0x021: rrmovq %rdx,%rbx # Not executed
       bubble
0x021: rrmovq %rdx,%rbx # Not executed
       bubble
0x021: rrmovq %rdx,%rbx # Not executed
       bubble
0x013: irmovq $10,%rdx # Return point
```

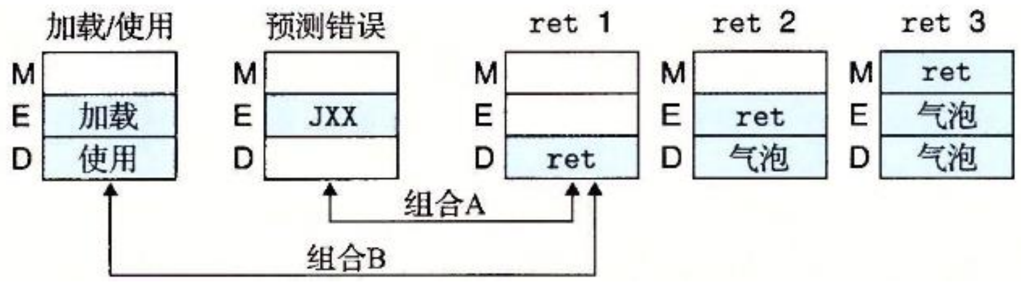
Hazard: Branch Misprediction

- After Execute stage, the right branch will be known.
- Insert two bubbles.



```
# 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
```

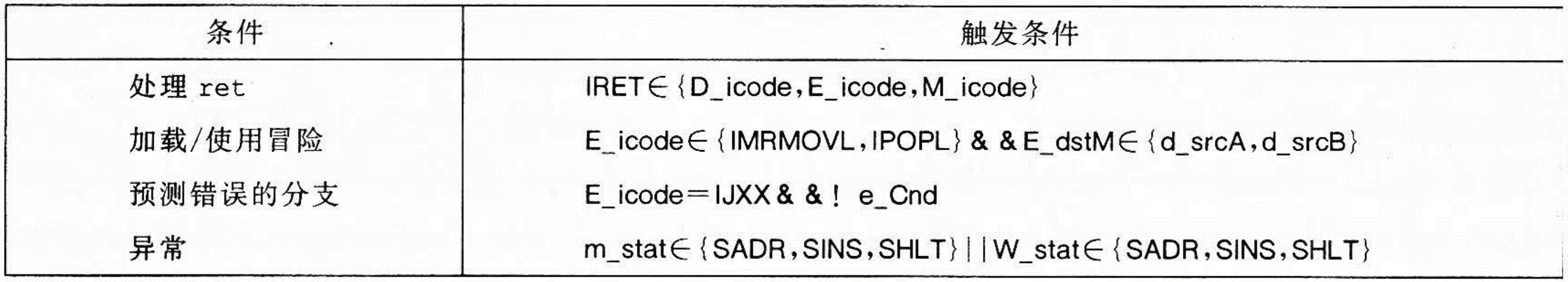
Hazard Combination



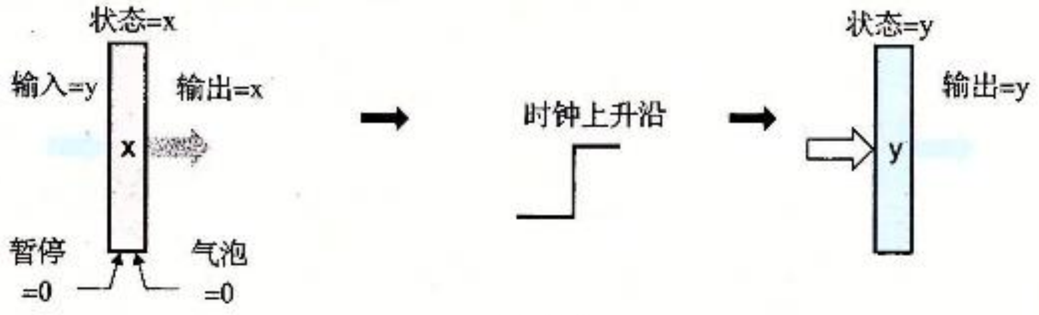
条件	流水线寄存器				
	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
预测错误的分支	正常	气泡	气泡	正常	正常
组合	暂停	气泡	气泡	正常	正常

条件	流水线寄存器				
	F	D	E	M	W
处理 ret	暂停	气泡	正常	正常	正常
预测错误的分支	暂停	暂停	气泡	正常	正常
组合	暂停	气泡＋暂停	气泡	正常	正常
期望的情况	暂停	暂停	气泡	正常	正常

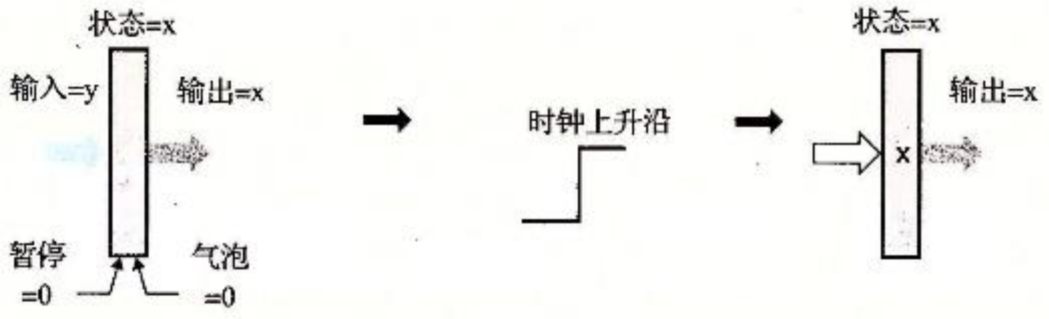
Hazard Detection & Control



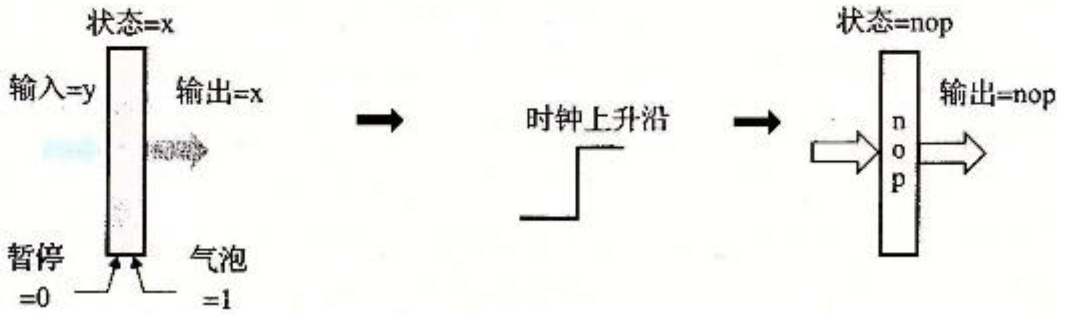
条件	触发条件
处理 ret	IRET∈{D_icode,E_icode,M_icode}
加载/使用冒险	E_icode∈{IMRMOVL,IPOPL} & & E_dstM∈{d_srcA,d_srcB}
预测错误的分支	E_icode=IJXX & & ! e_Cnd
异常	m_stat∈{SADR,SINS,SHLT} || W_stat∈{SADR,SINS,SHLT}



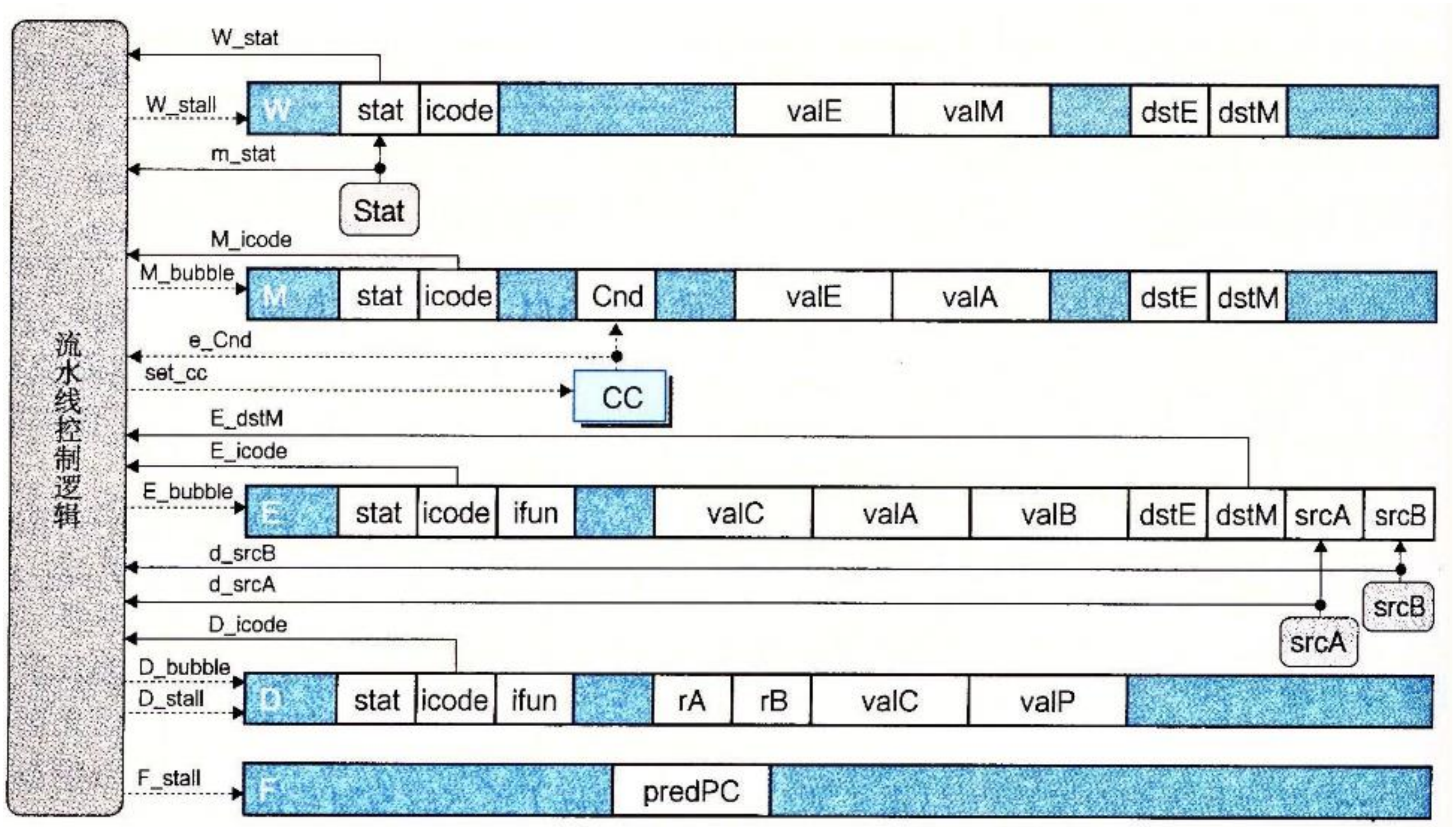
a）正常



b）暂停

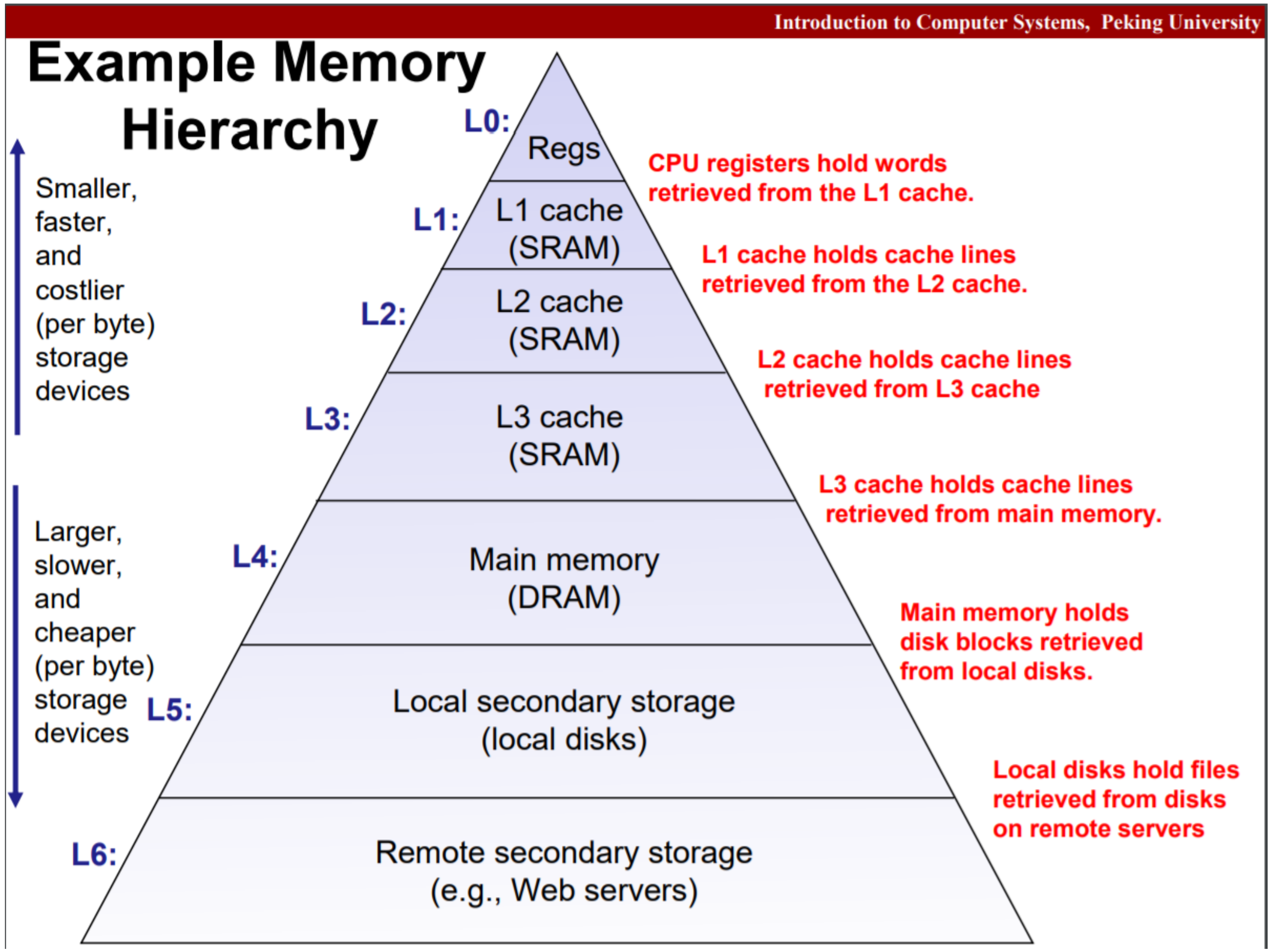


Implementing Pipeline Control



Memory Hierarchy

Example



Introduction to Computer Systems, Peking University
Example Memory Hierarchy
Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (e.g., Web servers)
CPU registers hold words retrieved from the L1 cache.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from L3 cache
L3 cache holds cache lines retrieved from main memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote servers

Storage Technology

- RAM: SRAM & DRAM
- Disk
- Bus structure

RAM

- Random Access Memory (RAM)
  - Volatile, expensive, compared to hard disk

- SRAM versus DRAM
  - SRAM doesn’t need refresh
    - faster and stable, more expensive
    - used as cache memories
  - DRAM
    - higher density, lower power consumption
    - used as main memory

DRAM: Access

- Row Access Strobe (RAS)
- Column Access Strobe (CAS)
- Memory module: Read & Write a word
- FPM DRAM, SDRAM, DDR SDRAM

ROM

- nonvolatile, compared to RAM
- PROM: only programmed once
- EPROM
- EEPROM -> flash memory
- firmware: stored in ROM

BUS

- Bus transaction: read and write
- System bus: connecting CPU and I/O bridge
- Memory bus: connecting I/O bridge and main memory
- I/O bus: disk, graphic card and other buses

DISK

- Capacity:
  - $Capacity\ = \left(\#\frac{bytes}{sector}\right) * \left(\#\frac{avg.sectors}{track}\right) * \left(\#\frac{tracks}{surface}\right) * \left(\#\frac{surfaces}{platter}\right) * \left(\#\frac{platters}{disk}\right)$
- Access time:
  - avg seek time + avg rotation time + avg transfer time

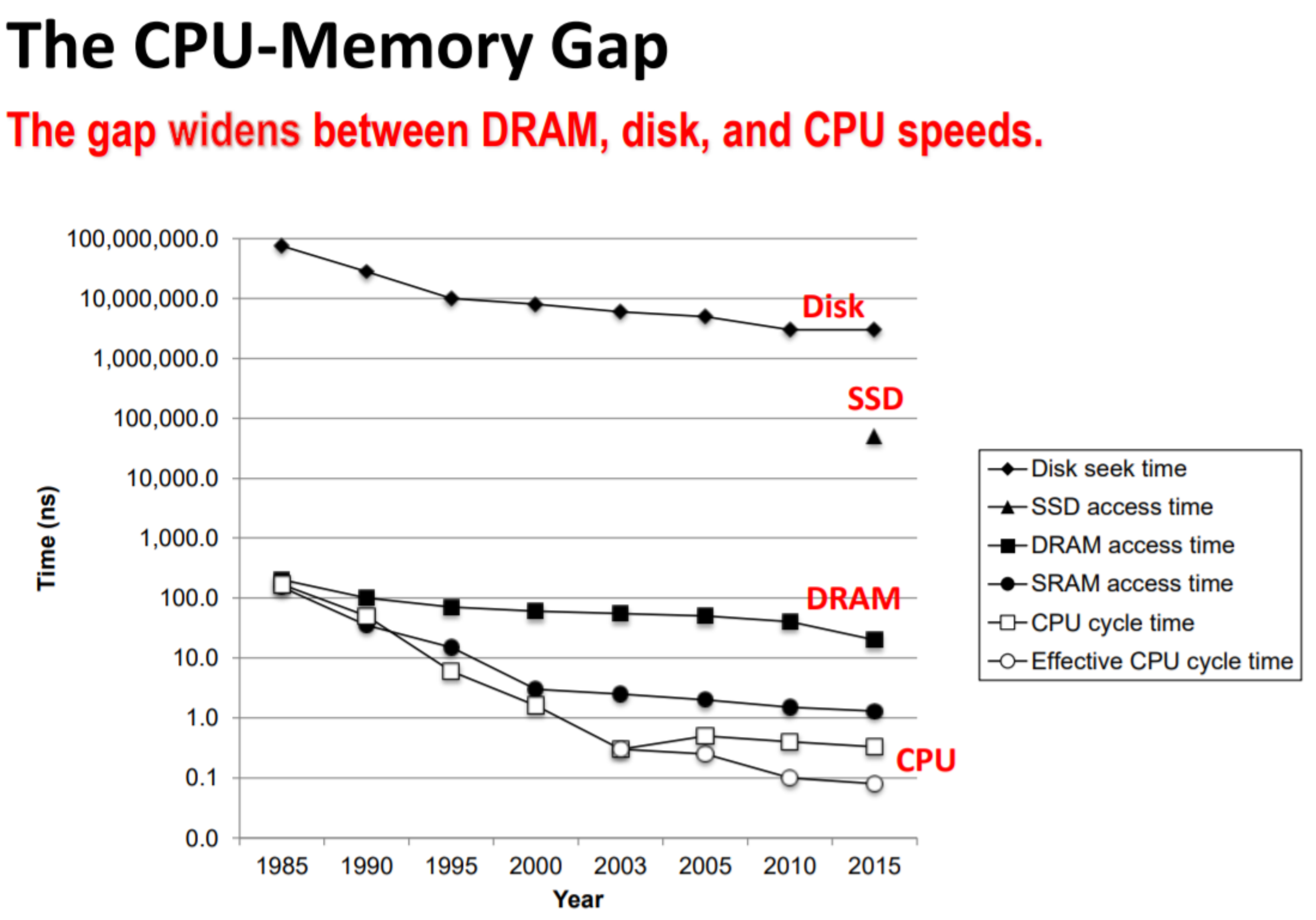
Unit Conversion

- K(kilo), M(mega), G(giga), T(tera): context dependent
- DRAM & SRAM: $K = 2^{10}$, $M = 2^{20}$, $G = 2^{30}$, $T = 2^{40}$
- Disk & network: $K = 10^{3}$, $M = 10^{6}$, $G = 10^{9}$, $T = 10^{12}$

SSD

- Solid State Disk (SSD)
- Sequential access faster than random access
- Write slower than Read
- Modifying a block page requires full page erasure and copy

Developing Tendency



Locality

- Temporal locality
- Spatial locality
- Data-access: temporal locality or spatial locality:
  - The smaller the step length, the better the spatial locality.
  - Repeating references to the same variable has the temporal locality.
- Instruction-fetch: both locality:
  - The smaller the loop body and the more the number of iteration, the better the locality.

Thanks for listening.