CS:APP Chapter 4 Computer Architecture Overview
第4章 处理器体系结构概述

任课教师：

宿红毅　张艳　黎有琦　颜珂

原作者：

Randal E. **Bryant and** David R. O'Hallaron

Carnegie Mellon University

课程概览 Course Outline

背景 Background

- 指令集 Instruction sets
- 逻辑设计 Logic design

顺序实现 Sequential Implementation

- 一个简单但是不是很快的处理器设计 A simple, but not very fast processor design

流水线 Pipelining

- 让更多的事情同时运行 Get more things running simultaneously

流水线实现 Pipelined Implementation

- 让它发挥作用 Make it work

高级主题 Advanced Topics

- 性能分析 Performance analysis
- 高性能处理器设计 High performance processor design

覆盖的内容 Coverage

我们的方法 Our Approach

- 针对特定指令集进行设计 Work through designs for particular instruction set
  - Y86-64是Intel x86-64的简化版本 Y86-64 – a simplified version of the Intel x86-64
  - 如果了解一个，或多或少会了解全部 If you know one, you more-or-less know them all
- 工作在“微体系结构”级 Work at “microarchitectural” level
  - 将基本硬件块组装进整个处理器结构中 Assemble basic hardware blocks into overall processor structure
    - » 内存、功能单元等 Memories, functional units, etc.
  - 用控制逻辑驱动以确保每条指令正确地流动 Surround by control logic to make sure each instruction flows through properly

覆盖的内容 Coverage

我们的方法 Our Approach

- 使用简单的硬件描述语言来描述控制逻辑 Use simple hardware description language to describe control logic
  - 可以扩展和修改 Can extend and modify
  - 通过模拟仿真进行测试 Test via simulation
  - 转换设计使用Verilog硬件描述语言 Route to design using Verilog Hardware Description Language
    - » 参见网站旁注：See Web aside ARCH:VLOG

安排 Schedule

Part A

- 指令集体系结构 Instruction set architecture
- 逻辑设计 Logic design

作业： 编写和测试汇编代码程序 *Assignment:* Write & test assembly code programs

Part B

- 顺序实现 Sequential implementation
- 流水线和初始流水线实现 Pipelining and initial pipelined implementation

作业： 增加新指令到顺序实现 *Assignment:* Add new instructions to sequential implementation

Part C

- 让流水线工作 Making the pipeline work
- 现代处理器设计 Modern processor design

作业： 优化程序+流水线以实现性能最大化 *Assignment:* Optimize program+pipeline for maximum performance

CS:APP Chapter 4
Computer Architecture
Instruction Set Architecture
指令集体系结构

任课教师:

宿红毅　　张艳　　黎有琦　　颜珂

原作者:

Randal E. **Bryant and** David R. O'Hallaron

Carnegie Mellon University

为何要研究CPU设计
Why do We Study the CPU Design?

理解基本的计算机组织结构 Understand basic computer organization

- 指令集体系结构 Instruction set architecture

深度探索CPU工作机制 Deeply explore the CPU working mechanism

- 指令是如何执行的 How the instruction is executed

有助于编程 Help you programming

- 完全理解计算机的组织和工作方式有助于编写更加稳定和高效的代码 Fully understand how computer is organized and works will help you write more stable and efficient code

指令集体系结构 #1
Instruction Set Architecture #1

什么是ISA What is ISA ?

- 汇编语言抽象 Assemble Language Abstraction
  - 处理器支持的汇编语言 assembly supported by a processor
- 机器语言抽象 Machine Language Abstraction
  - 字节级表示 Byte-level representation

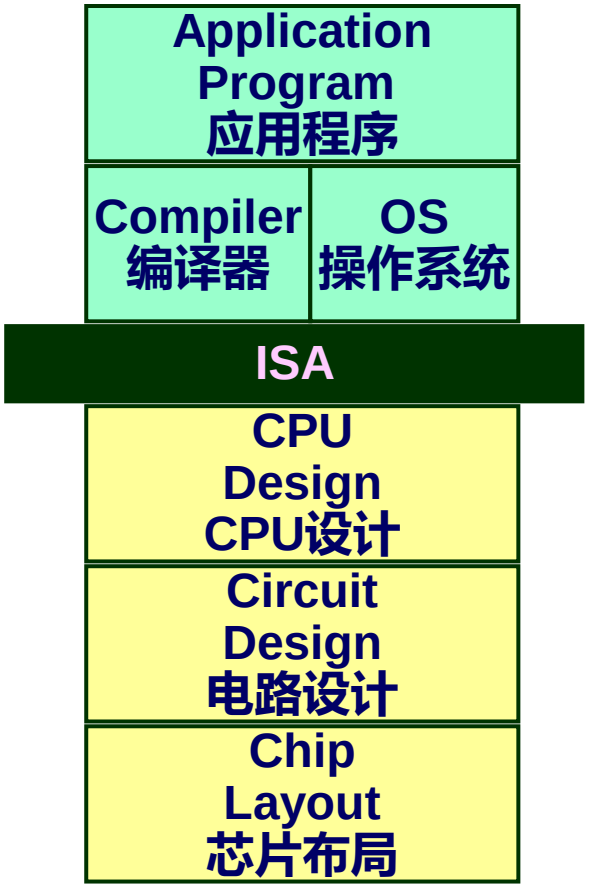
它提供什么? What does it provide?

- 真实计算机的一个抽象，隐藏了实现细节 An abstraction of the real computer, hide the details of implementation
  - 计算机指令语法 The syntax of computer instructions
  - 指令语义 The semantics of instructions
  - 执行模式 The execution model
  - 程序员可见的计算机状态 Programmer-visible computer status

指令集体系结构
Instruction Set Architecture

汇编语言视角 Assembly Language View

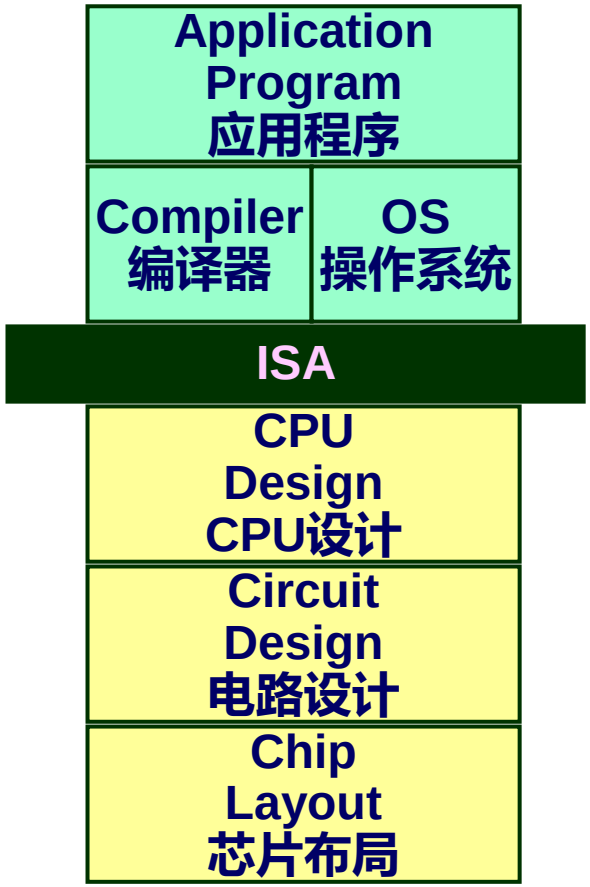
- 处理器状态 Processor state
  - 寄存器、内存。。。Registers, memory, …
- 指令 Instructions
  - `addq`, `pushq`, `ret`, …
  - 指令如何编码为字节序列 How instructions are encoded as bytes



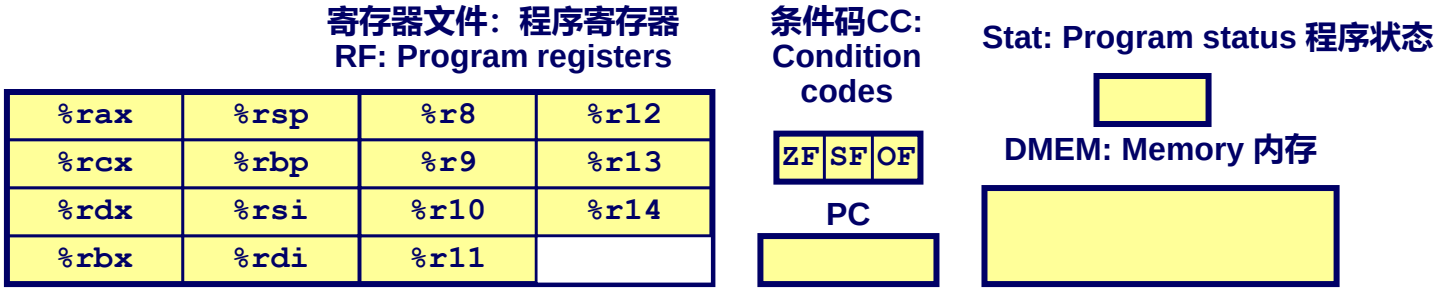
指令集体系结构
Instruction Set Architecture

抽象层次 Layer of Abstraction

- 之上：程序机器如何工作 Above: how to program machine
  - 处理器顺序执行指令 Processor executes instructions in a sequence
- 之下：需要构建什么 Below: what needs to be built
  - 使用各种技巧使其运行更快 Use variety of tricks to make it run fast
  - 例如同时执行多条指令 E.g., execute multiple instructions simultaneously



Y86-64处理器状态
Y86-64 Processor State



- 程序寄存器 Program Registers
  - 15个寄存器（省略%r15），每个64位 15 registers (omit `%r15`). Each 64 bits
- 条件码 Condition Codes
  - 算术或逻辑运算指令设置单个比特位标志 Single-bit flags set by arithmetic or logical instructions
    - » ZF: Zero零　　SF:Negative负数　　OF: Overflow溢出

Y86-64处理器状态
Y86-64 Processor State

寄存器文件：程序寄存器
RF: Program registers

%rax	%rsp	%r8	%r12
%rcx	%rbp	%r9	%r13
%rdx	%rsi	%r10	%r14
%rbx	%rdi	%r11	

条件码CC: Condition codes

ZF	SF	OF

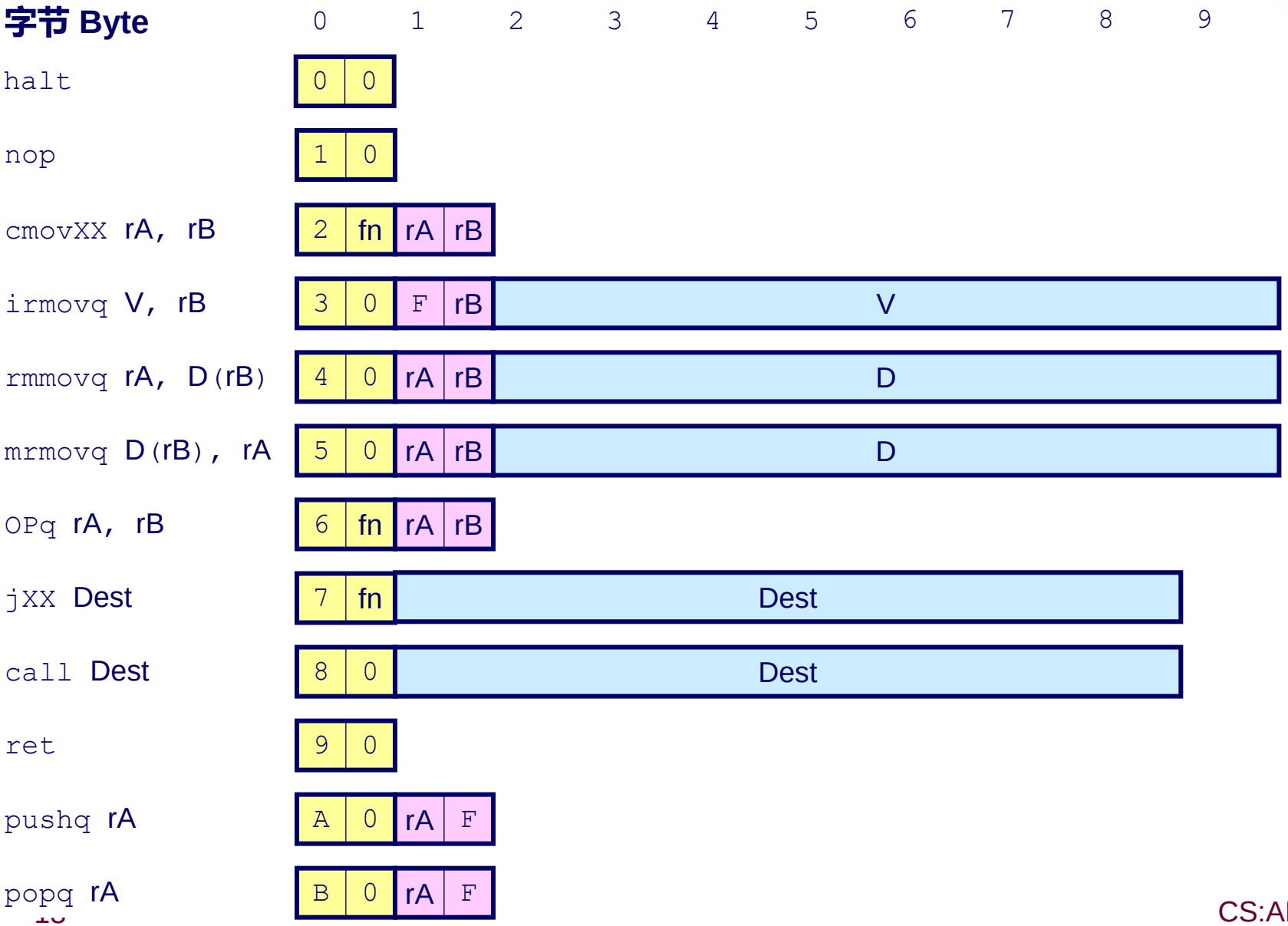
PC

Stat: Program status 程序状态

DMEM: Memory 内存

- 程序计数器 Program Counter
  - 指明下一条指令地址 Indicates address of next instruction
- 程序状态 Program Status
  - 指明正常运行还是一些错误情况 Indicates either normal operation or some error condition
- 内存 Memory
  - 字节寻址存储数组 Byte-addressable storage array
  - “字”采用小端字节顺序存储 Words stored in little-endian byte order

Y86-64 Instruction Set 指令集 #1

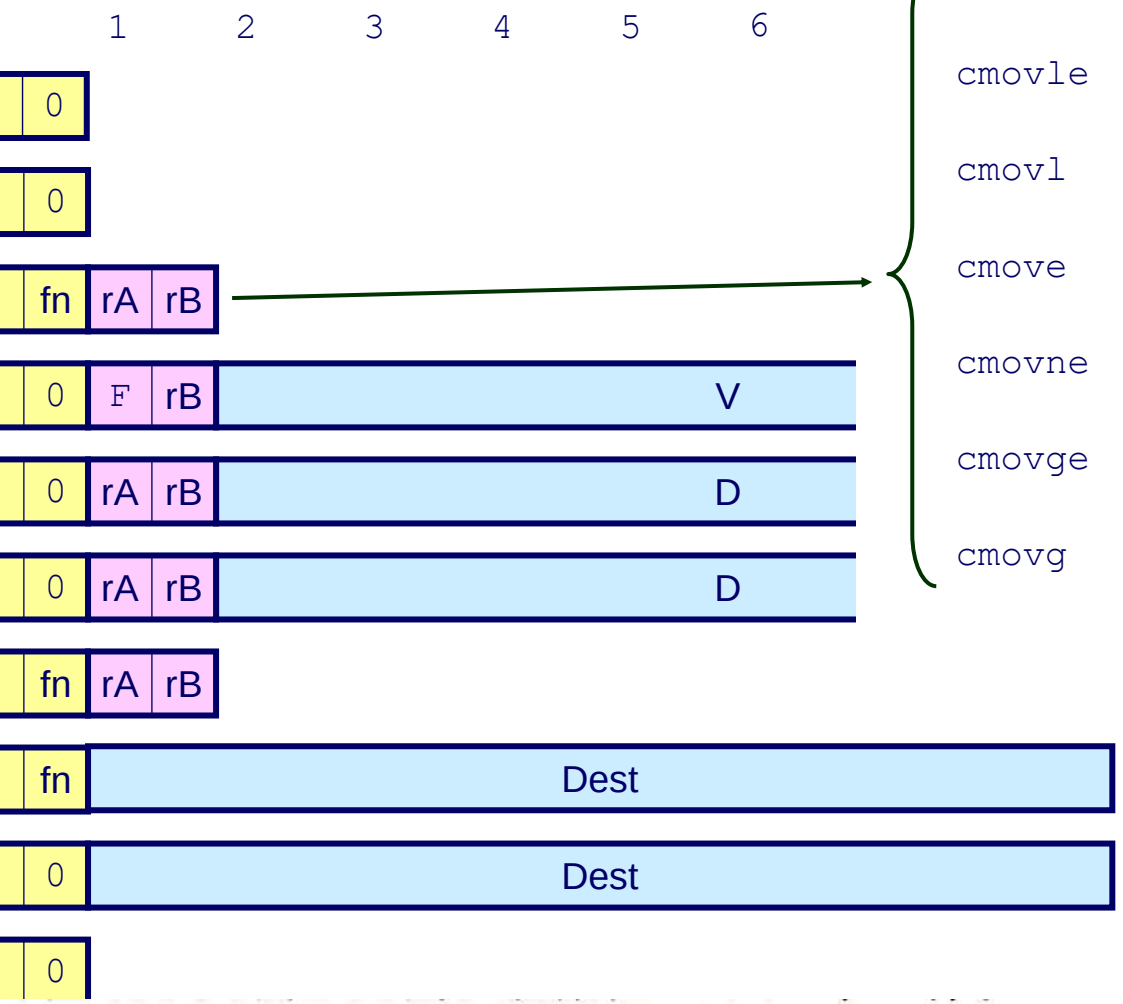


Y86-64 Instructions 指令

格式 Format

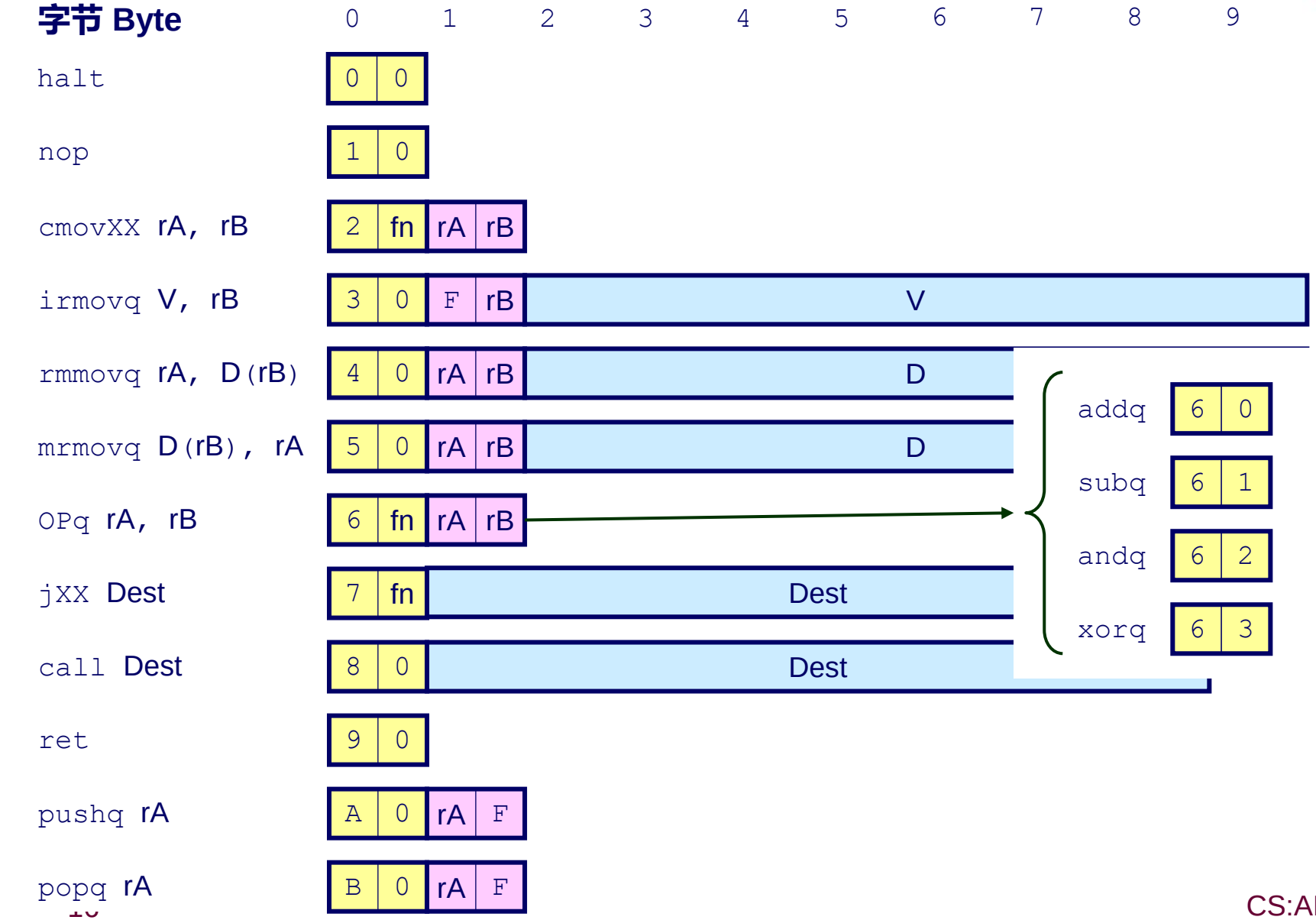
- 1-10字节信息从内存读出 1–10 bytes of information read from memory
  - 从第一个字节能够确定指令长度 Can determine instruction length from first byte
  - 与x86-64相比,指令类型不是很多，而且编码更简单 Not as many instruction types, and simpler encoding than with x86-64
- 每次访问和修改一部分程序状态 Each accesses and modifies some part(s) of the program state

Y86-64 Instruction Set 指令集#2

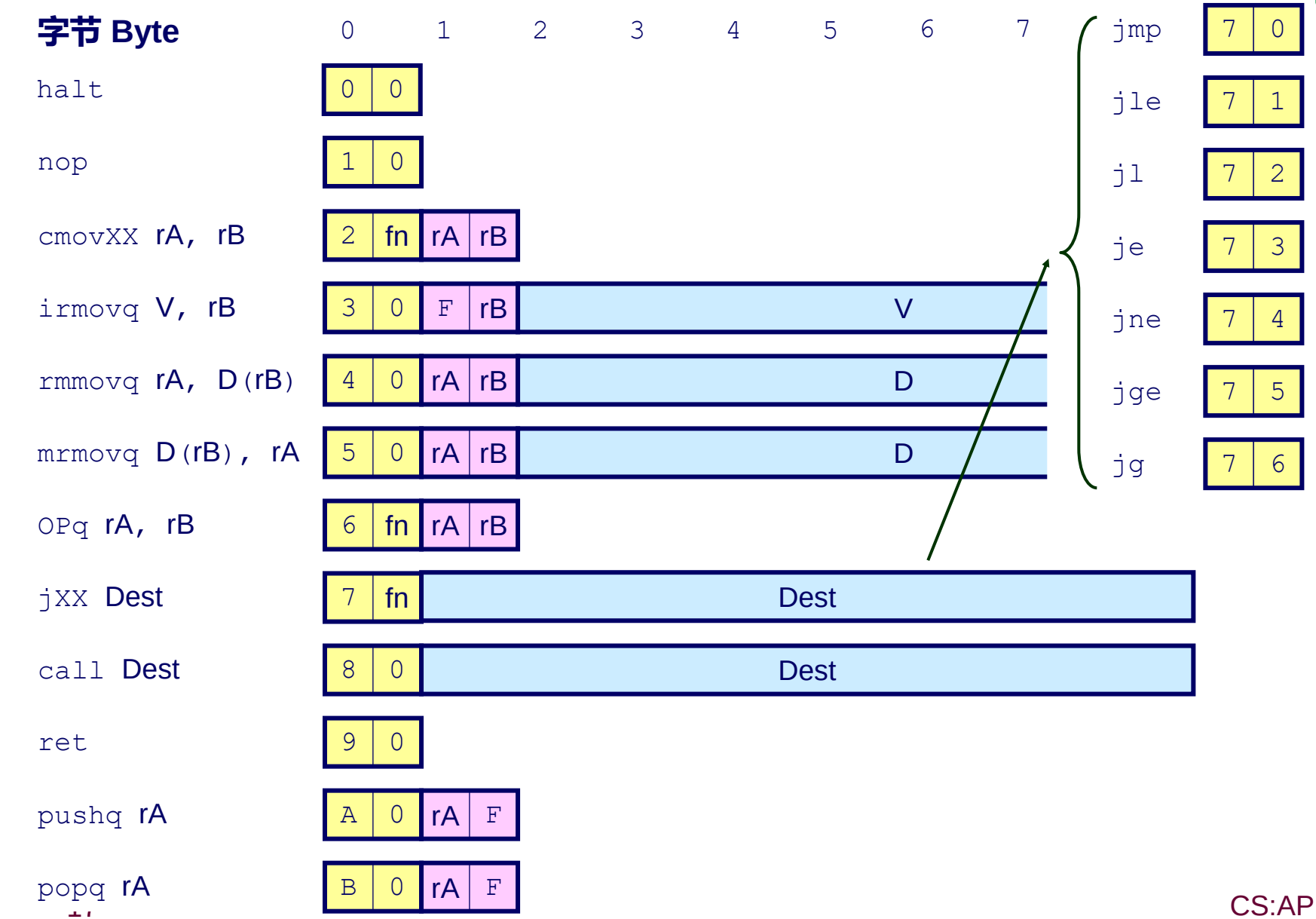


- halt 指令停止指令的执行。x86-64 中有一个与之相当的指令 hlt。x86-64 的应用程序不允许使用这条指令，因为它会导致整个系统暂停运行。对于 Y86-64 来说，执行 halt 指令会导致处理器停止，并将状态码设置为 HLT(参见 4.1.4 节)。

Y86-64 Instruction Set 指令集#3



Y86-64 Instruction Set 指令集 #4



编码寄存器 Encoding Registers

每个寄存器有4位ID Each register has 4-bit ID

%rax	0	%r8	8
%rcx	1	%r9	9
%rdx	2	%r10	A
%rbx	3	%r11	B
%rsp	4	%r12	C
%rbp	5	%r13	D
%rsi	6	%r14	E
%rdi	7	No Register	F

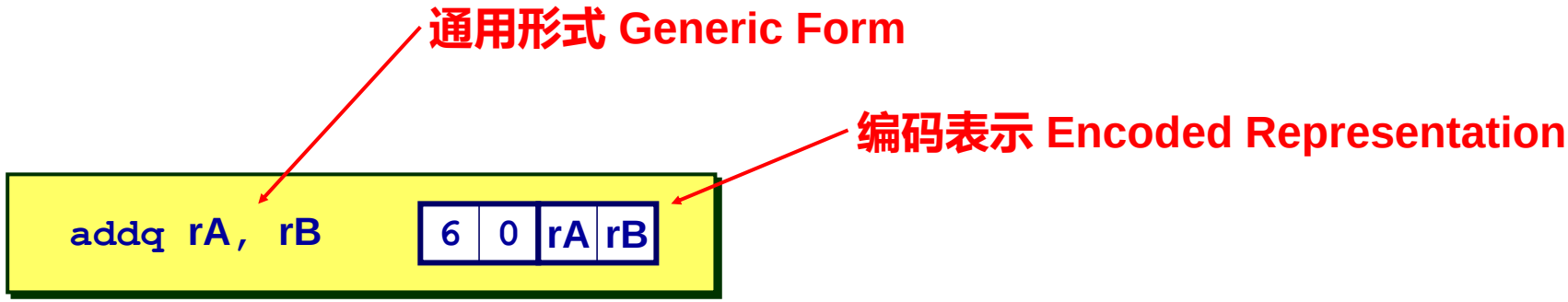
- 与x86-64编码相同 Same encoding as in x86-64

寄存器ID 15（0xF）指示“没有用到寄存器” Register ID 15 (0xF) indicates “no register”

- 在硬件设计多处将会用到 Will use this in our hardware design in multiple places

指令示例 Instruction Example

加法指令 Addition Instruction



- rB寄存器中的值加上rA寄存器中的值 Add value in register rA to that in register rB
  - 存储结果到rB寄存器 Store result in register rB
  - 注意Y86-64仅允许寄存器数据进行加法运算 Note that Y86-64 only allows addition to be applied to register data
- 根据结果设置条件码 Set condition codes based on result
- 例如 e.g., `addq %rax,%rsi` 编码 Encoding: `60 06`
- 两字节编码 Two-byte encoding
  - 第一个字节指明指令类型 First indicates instruction type
  - 第二字节给出源和目的寄存器 Second gives source and destination registers

算术和逻辑运算
Arithmetic and Logical Operations

指令代码 Instruction Code 功能码 Function Code

Add

Subtract (rA from rB)

And

Exclusive-Or

- 泛指为“OPq” Refer to generically as “OPq”
- 仅仅“功能码”编码不同 Encodings differ only by “function code”
  - 第一个指令字节低4位 Low-order 4 bits in first instruction word
- 设置条件码作为副作用 Set condition codes as side effect

传送操作 Move Operations

寄存器到寄存器 Register ➔ Register

`rrmovq` rA, rB

2	0

立即数到寄存器 Immediate ➔ Register

`irmovq` V, rB

3	0	F	rB	V

寄存器到内存 Register ➔ Memory

`rmmovq` rA, D(rB)

4	0	rA	rB	D

内存到寄存器 Memory ➔ Register

`mrmovq` D(rB), rA

5	0	rA	rB	D

- 类似x86-64传送指令 Like the x86-64 `movq` instruction
- 内存寻址格式更简单 Simpler format for memory addresses
- 给定不同名字保持区别 Give different names to keep them distinct

传送指令示例
Move Instruction Examples

x86-64	Y86-64
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx`
编码 Encoding:	`30 F2 cd ab 00 00 00 00 00 00`
`movq %rsp, %rbx`	`rrmovq %rsp, %rbx`
编码 Encoding:	`20 43`
`movq -12(%rbp),%rcx`	`mrmovq -12(%rbp),%rcx`
编码 Encoding:	`50 15 f4 ff ff ff ff ff ff ff`
`movq %rsi,0x41c(%rsp)`	`rmmovq %rsi,0x41c(%rsp)`
编码 Encoding:	`40 64 1c 04 00 00 00 00 00 00`

条件传送指令 Conditional Move Instructions

Move Unconditionally 无条件

`rrmovq` rA, rB	2	0	rA	rB

Move When Less or Equal 小于等于

`cmovle` rA, rB	2	1	rA	rB

Move When Less 小于

`cmovl` rA, rB	2	2	rA	rB

Move When Equal 等于

`cmove` rA, rB	2	3	rA	rB

Move When Not Equal 不等

`cmovne` rA, rB	2	4	rA	rB

Move When Greater or Equal 大于等于

`cmovge` rA, rB	2	5	rA	rB

Move When Greater 大于

`cmovg` rA, rB	2	6	rA	rB

- 泛指为“cmovXX” Refer to generically as “`cmovXX`”
- 仅“功能码”编码不同 Encodings differ only by “function code”
- 根据条件码的值 Based on values of condition codes
- rrmovq指令的变种 Variants of `rrmovq` instruction
  - （有条件）复制值从源到目的寄存器 (Conditionally) copy value from source to destination register

跳转指令 Jump Instructions

跳转（条件）Jump (Conditionally)

- 泛指作“jXX” Refer to generically as “jxx”
- 仅“功能码”fn编码不同 Encodings differ only by “function code” fn
- 根据条件码的值 Based on values of condition codes
- 与x86-64处理器相同 Same as x86-64 counterparts
- 编码完整目的地址 Encode full destination address
  - 与x86-64中见到的PC相对寻址不同 Unlike PC-relative addressing seen in x86-64

跳转指令 Jump Instructions

Jump Unconditionally 无条件

`jmp` Dest	7	0	Dest

Jump When Less or Equal 小于等于

`jle` Dest	7	1	Dest

Jump When Less 小于

`jl` Dest	7	2	Dest

Jump When Equal 等于

`je` Dest	7	3	Dest

Jump When Not Equal 不等

`jne` Dest	7	4	Dest

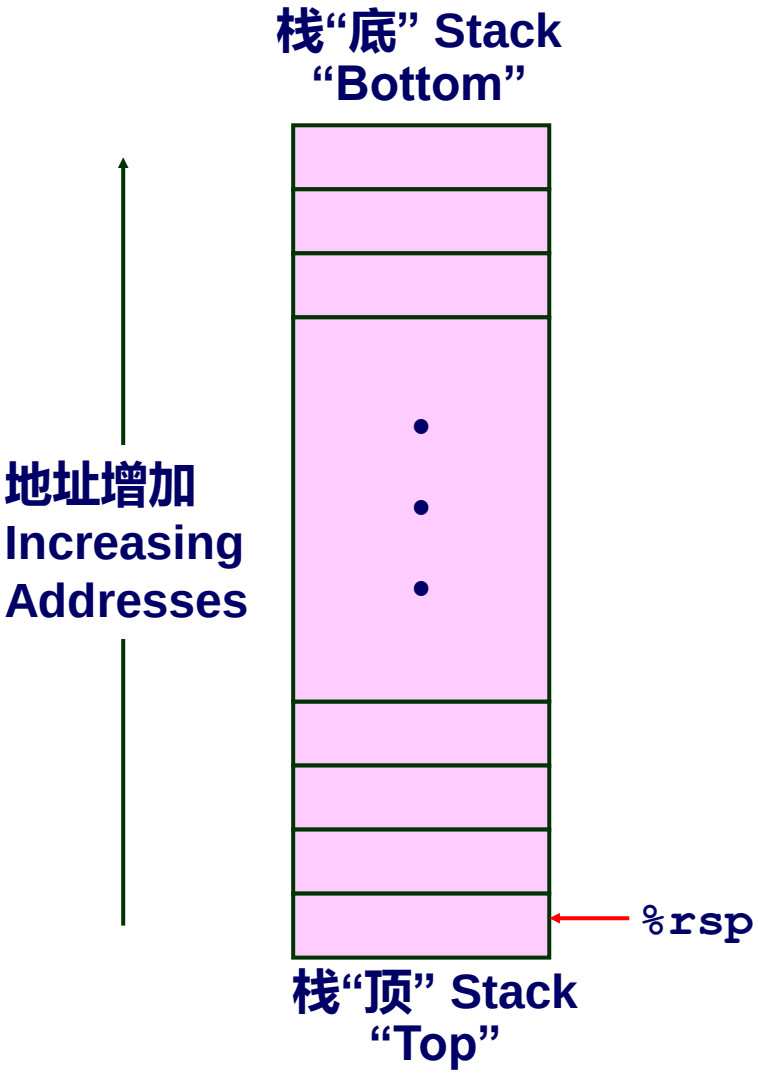
Jump When Greater or Equal 大于等于

`jge` Dest	7	5	Dest

Jump When Greater 大于

`jg` Dest	7	6	Dest

Y86-64程序栈 Y86-64 Program Stack



- 存储程序数据的内存区域 Region of memory holding program data
- 在Y86-64（和x86-64）中用于支持过程调用 Used in Y86-64 (and x86-64) for supporting procedure calls
- 栈顶由%rsp指示 Stack top indicated by `%rsp`
  - 栈顶元素的地址 Address of top stack element
- 栈向低地址方向生成 Stack grows toward lower addresses
  - 栈顶元素在栈的最低地址 Top element is at lowest address in the stack
  - 压栈时必须首先递减栈指针 When pushing, must first decrement stack pointer
  - 弹出后栈指针递增 After popping, increment stack pointer

栈操作 Stack Operations

`pushq rA`

A	0	rA	F

- %rsp减8 Decrement `%rsp` by 8
- 将rA中的字存储到%rsp指向的内存单元 Store word from rA to memory at `%rsp`
- 类似 x86-64 Like x86-64

`popq rA`

B	0	rA	F

- 从%rsp指向的内存读出字 Read word from memory at `%rsp`
- 存储到rA中 Save in rA
- %rsp加8 Increment `%rsp` by 8
- 类似x86-64 Like x86-64

子程序调用和返回
Subroutine Call and Return

`call Dest`

- 将下一条指令地址压入栈 Push address of next instruction onto stack
- 开始执行目的处指令 Start executing instructions at Dest
- 类似 x86-64 Like x86-64

`ret`

- 从栈弹出值 Pop value from stack
- 作为下条指令地址使用 Use as address for next instruction
- 类似x86-64 Like x86-64

杂项指令 Miscellaneous Instructions

- 不做任何事情 Don’t do anything

- 停止执行指令 Stop executing instructions
- x86-64有兼容指令，但是不能在用户模式使用 x86-64 has comparable instruction, but can’t execute it in user mode
- 我们用它来停止模拟器 We will use it to stop the simulator
- 这个编码确保程序运行到内存为零时会停机 Encoding ensures that program hitting memory initialized to zero will halt

状态码 Status Conditions

Mnemonic	Code
AOK	1

- 正常运行 Normal operation

Mnemonic	Code
HLT	2

- 遇到停机指令 Halt instruction encountered

Mnemonic	Code
ADR	3

- 遇到错误地址（指令或数据） Bad address (either instruction or data) encountered

Mnemonic	Code
INS	4

- 遇到非法指令 Invalid instruction encountered

预期的行为 Desired Behavior

- 如果AOK，则继续运行 If AOK, keep going
- 否则，停止程序执行 Otherwise, stop program execution

编写Y86-64代码 Writing Y86-64 Code

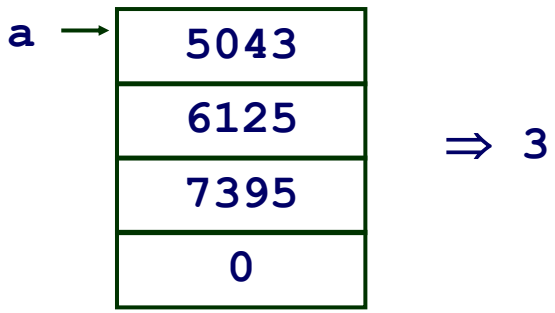
尝试尽可能使用C语言编译器 Try to Use C Compiler as Much as Possible

- 用C语言编写代码 Write code in C
- 编译成x86-64汇编 Compile for x86-64 with `gcc -Og -S`
- 移植成Y86-64汇编 Transliterate into Y86-64
- *现代编译器让这个工作变得更困难 Modern compilers make this more difficult*

代码示例 Coding Example

- 确定以空作结尾列表中元素数量 Find number of elements in null-terminated list

```
int len1(int a[]);
```



⇒ 3

Y86-64代码生成示例
Y86-64 Code Generation Example

首先尝试 First Try

- 编写典型的数组代码 Write typical array code

```
/* Find number of elements in
   null-terminated list */
long len(long a[])
{
  long len;
  for (len = 0; a[len]; len++)
       ;
  return len;
}
```

- 编译 Compile with `gcc -Og -S`

问题 Problem

- 在Y86-64中很难做数组索引 Hard to do array indexing on Y86-64
  - 因为没有缩放寻址方式 Since don't have scaled addressing modes

```
L3:
   addq $1,%rax
   cmpq  $0, (%rdi,%rax,8)
   jne L3
```

Y86-64代码生成示例#2
Y86-64 Code Generation Example #2

第二次尝试 Second Try

- 编写C语言代码模仿期望的Y86-64代码 Write C code that mimics expected Y86-64 code

```
long len2(long *a)
{
    long ip = (long) a;
    long val = *(long *) ip;
    long len = 0;
    while (val) {
        ip += sizeof(long);
        len++;
        val = *(long *) ip;
    }
    return len;
}
```

结果 Result

- 编译器生成和以前完全一样的代码 Compiler generates exact same code as before!
- 编译器将两个版本都转换成同样的中间格式 Compiler converts both versions into same intermediate form

Y86-64代码生成示例#3
Y86-64 Code Generation Example #3

```
len:
    irmovq $1, %r8          # Constant 1
    irmovq $8, %r9          # Constant 8
    irmovq $0, %rax         # len = 0
    mrmovq (%rdi), %rdx     # val = *a
    andq %rdx, %rdx         # Test val
    je Done                 # If zero, goto Done
Loop:
    addq %r8, %rax          # len++
    addq %r9, %rdi          # a++
    mrmovq (%rdi), %rdx     # val = *a
    andq %rdx, %rdx         # Test val
    jne Loop                # If !0, goto Loop
Done:
    ret
```

寄存器 Register	用途 Use
%rdi	a
%rax	len
%rdx	val
%r8	1
%r9	8

Y86程序 Y86 Programs

```
int Sum(int *Start, int Count)
{
    int sum = 0;
    while (Count) {
        sum += *Start;
        Start++;
        Count--;
    }
    return sum;
}
```

Y86 Assembly

IA64 code

```
1 sum:
2     movl $0, %eax
3     jmp .L2
4  .L3:
5     addq (%rdi), %rax
6     addq $8, %rdi
7     subq $1, %rsi
8  .L2:
9     testq %rsi, %rsi
10    jne .L3
11    rep; ret
```

Y86 code

*int Sum(int *Start, int Count)*

```
1 sum:
2     irmovq $8,%r8
3     irmovq $1,%r9
4     xorq %rax,%rax
5     andq %rsi,%rsi
6     jmp test
7 loop:
8     mrmovq (%rdi),%r10
9     addq %r10,%rax
10    addq %r8,%rdi
11    subq %r9,%rsi
12 test:
13    jne loop
14    ret
```

Y86目标程序 Y86 Object Program

```
1   # Execution begins at address 0
2         .pos 0
3         irmovq    stack, %rsp       # Set up stack pointer
4         call      main              # Execute main program
5         halt                        # Terminate program
6
```

符号名 Symbolic Name

自动生成程序的初始部分 Init part of program generated automatically

汇编器伪指令 Assembler directives

- .pos 0, .pos 0x200
- .align

Y86目标程序 Y86 Object Program

```
7   # Array of 4 elements
8         .align 8
9   array:
10        .quad 0x000d000d000d
11        .quad 0x00c000c000c0
12        .quad 0x0b000b000b00
13        .quad 0xa000a000a000
14
```

数据区 Data area

- Array指示数组的起始 array denotes the start of an array
- 对齐到8字节边界 Aligned on 8-byte boundary

Y86目标程序 Y86 Object Program

```
15  main:
16      irmovq    array,%rdi
17      irmovq    $4,%rsi
18      call      sum                 # sum(array, 4)
19      ret
20
```

Y86目标程序 Y86 Object Program

```
21  # long sum(long *start, long count)
22  # start in %rdi, count in %rsi
23  sum:
24      irmovq    $8,%r8          # Constant 8
25      irmovq    $1,%r9          # Constant 1
26      xorq      %rax,%rax       # sum = 0
27      andq      %rsi,%rsi       # Set CC
28      jmp       test            # Goto test
```

Y86目标程序 Y86 Object Program

```
29  loop:
30      mrmovq (%rdi),%r10      # Get *start
31      addq %r10,%rax          # Add to sum
32      addq %r8,%rdi           # start++
33      subq %r9,%rsi           # count--. Set CC
34  test:
35      jne loop                # Stop when 0
36      ret                     # Return
37
38 # Stack starts here and grows to lower addresses
39 .pos 0x200
```

Y86目标程序 Y86 Object Program

```
38   # Stack starts here and grows to lower addresses
39       .pos 0x200
40   stack:
```

符号名 Symbolic Name

程序员必须自己写汇编代码 Programmers must write assembly codes themselves

- 包括管理内存 including manage the memory
  - 例如为数组和栈分配内存 such as allocate memory for array and stack
  - 以及避免内存覆盖 as well as avoid the memory overwritting

Y86-64样例程序结构#1
Y86-64 Sample Program Structure #1

```
init:                       # Initialization
    . . .
    call Main
    halt

    .align 8                # Program data
array:
    . . .

Main:                       # Main function
    . . .
    call len    . . .

len:                        # Length function
    . . .

    .pos 0x100              # Placement of stack
Stack:
```

- 程序起始地址为零 Program starts at address 0
- 必须设置栈 Must set up stack
  - 栈的位置 Where located
  - 栈指针值 Pointer values
  - 确保不会覆盖代码 Make sure don't overwrite code!
- 必须初始化数据 Must initialize data

Y86-64程序结构#2
Y86-64 Program Structure #2

```
init:
      # Set up stack pointer
      irmovq Stack, %rsp
      # Execute main program
      call Main
      # Terminate
      halt

# Array of 4 elements + terminating 0
      .align 8
Array:
      .quad 0x000d000d000d000d
      .quad 0x00c000c000c000c0
      .quad 0x0b000b000b000b00
      .quad 0xa000a000a000a000
      .quad 0
```

- 程序起始地址为零 Program starts at address 0
- 必须设置栈 Must set up stack
- 必须初始化数据 Must initialize data
- 可以使用符号名 Can use symbolic names

Y86-64程序结构#3
Y86-64 Program Structure #3

```
Main:
      irmovq array,%rdi
      # call len(array)
      call len
      ret
```

设置对len的调用 Set up call to len

- 遵循x86-64过程规则 Follow x86-64 procedure conventions
- 数组地址作为参数传递 Push array address as argument

汇编Y86-64程序 Assembling Y86-64 Program

```
unix> yas len.ys
```

- 生成“目标代码”文件len.yo Generates “object code” file `len.yo`
  - 实际看起来像反汇编输出 Actually looks like disassembler output

```
0x054:                      | len:
0x054: 30f80100000000000000 |   irmovq $1, %r8          # Constant 1
0x05e: 30f90800000000000000 |   irmovq $8, %r9          # Constant 8
0x068: 30f00000000000000000 |   irmovq $0, %rax         # len = 0
0x072: 50270000000000000000 |   mrmovq (%rdi), %rdx     # val = *a
0x07c: 6222                 |   andq %rdx, %rdx         # Test val
0x07e: 73a000000000000000   |   je Done                 # If zero, goto Done
0x087:                      | Loop:
0x087: 6080                 |   addq %r8, %rax          # len++
0x089: 6097                 |   addq %r9, %rdi          # a++
0x08b: 50270000000000000000 |   mrmovq (%rdi), %rdx     # val = *a
0x095: 6222                 |   andq %rdx, %rdx         # Test val
0x097: 748700000000000000   |   jne Loop                # If !0, goto Loop
0x0a0:                      | Done:
0x0a0: 90                   |   ret
```

模拟运行Y86-64程序
Simulating Y86-64 Program

```
unix> yis len.yo
```

- 指令集模拟器 Instruction set simulator
  - 计算每条指令对处理器状态的影响 Computes effect of each instruction on processor state
  - 打印从开始到现在的状态变化 Prints changes in state from original

```
Stopped in 33 steps at PC = 0x13.  Status 'HLT', CC Z=1 S=0 O=0
Changes to registers:
%rax:    0x0000000000000000      0x0000000000000004
%rsp:    0x0000000000000000      0x0000000000000100
%rdi:    0x0000000000000000      0x0000000000000038
%r8:     0x0000000000000000      0x0000000000000001
%r9:     0x0000000000000000      0x0000000000000008

Changes to memory:
0x00f0: 0x0000000000000000      0x0000000000000053
0x00f8: 0x0000000000000000      0x0000000000000013
```

指令集体系结构 #3
Instruction Set Architecture #3

ISA定义了处理器分类 ISA define the processor family

- 两个主要分类：Two main kind: RISC and CISC
  - RISC: SPARC, MIPS, PowerPC, ARM
  - CISC: X86 (or called IA32)

采用同样的ISA，有很多不同的处理器 Under same ISA, there are many different processors

- 来自不同的制造商 From different manufacturers
  - X86 from Intel, AMD and VIA
- 不同的型号 Different models
  - 8086, 80386, Pentium, atom, core i7

RISC vs. CISC

ISA

- 指令集体系结构 Instruction set architecture
  - 指令由特定处理器支持 Instructions supported by a particular processor
  - 其字节级编码 Their byte-level encodings

CISC

- 复杂指令集计算机 Complex instruction set computer

RISC

- 精简指令集计算机 Reduced instruction set computer

CISC

从最早的计算机开始 Involved from the earliest computers

大型主机和超级小型机 Mainframe and Minicomputers

- 早在上世纪80年代 By the early 1980s
- 其指令集增长得非常庞大 their instruction sets had grown quite large
  - 操作环形缓冲区 Manipulating circular buffers
  - 执行十进制运算 performing decimal arithmetic
  - 评估多项式 evaluating polynomials

微型计算机 Microcomputer

- 出现在上世纪70年代，有有限的指令集 Appeared in 1970s, had limited instruction sets
  - 受单芯片上晶体管数量的限制 constrained by number of transistors on a single chip
- 到上世纪80年代早期，按照这个路线增加其指令集 By the early 1980s, followed the path to increase their instruction sets

RISC



开发于上世纪80年代早期 Developed in the early 1980s

- 哲学 philosophy
  - 对于更简单的指令集格式能够生成高效代码 One are able to generate efficient code for a simpler form of instruction set
  - 复杂指令很难用编译器生成而且很少使用 Complex instructions are hard to generated with a compiler and seldom used

John Cocke (1925-2002)

- 1987 ACM Turing Award 图灵奖

David Patterson(UC Berkeley), John Hennessy (Stanford U)

- 2017 ACM Turing Award 图灵奖

RISC

IBM	Power
IBM and Motorola	PowerPC
Sun Microsystems	SPARC
Digital Equipment Corporation	Alpha
Hewlett Pack	Pa-risc
MIPS Technologies	MIPS
Acorn Computers Ltd	ARM(Acorn RISC Machine)

RISC vs. CISC

CICS	早期RISC Early RISC
指令数很多 A large number of instructions	指令较少 Many fewer instructions (<100)
有些指令执行时间较长 Some instructions with long execution times	没有执行时间很长的指令 No instruction with a long execution time
可变长编码 Variable-length encodings. (x86-64 1~15 bytes)	固定长度编码 Fixed-length encodings, (4 bytes in usual)
多种格式寻址操作数 Multiple formats for specifying operands	简单的寻址格式 Simple addressing formats
算术和逻辑运算可以应用到内存和寄存器操作数 Arithmetic and logical operations can be applied to both memory and register operands	算术和逻辑运算只能用于寄存器操作数，装载/存储 体系结构 Arithmetic and logical operations only use register operands. *load/store Architecture*

RISC vs. CISC

CICS	早期RISC Early RISC
对机器级程序来说实现细节是不可见的 Implementation artifacts hidden from machine level programs	机器级程序来说实现细节是可见的 Implementation artifacts exposed to machine level programs
条件码 Condition codes	显式测试指令将测试结果存储在普通寄存器中以用于条件评估 explicit test instructions store the test results in normal registers for use in conditional evaluation
过程链接通过栈实现 Stack-intensive procedure linkage	过程参数传递和返回值采用寄存器实现 Registers are used for procedure arguments and return addresses

Y86-64

Y86-64指令集 The Y86-64 instruction set

- 包括了CISC和RISC二者的属性 Includes attributes of both CISC and RISC
- 可以看成采用CISC指令集（x86-64）同时应用RISC一些原则进行了简化 Can be viewed as taking a CISC instruction set (x86-64) and simplifying it by applying some of the principles of RISC

在CISC这方面 On the CISC side

- 条件码、可变长指令 condition codes, variable-length instructions
- 使用栈存储返回地址 uses the stack to store return addresses.

在RISC方面 On the RISC side,

- 装载/存储 体系结构 a load/store architecture
- 规整的指令编码 a regular instruction encoding
- 通过寄存器传递过程参数 passes procedure arguments through registers

复杂指令集 CISC Instruction Sets

- 复杂指令集计算机 Complex Instruction Set Computer
- IA32就是例子 IA32 is example

面向栈的指令集 Stack-oriented instruction set

- 使用栈传递参数，保存程序计数器 Use stack to pass arguments, save program counter
- 显式的入栈和出栈指令 Explicit push and pop instructions

运算指令能够访问内存 Arithmetic instructions can access memory

- `addq %rax, 12(%rbx,%rcx,8)`
  - 需要内存读和写 requires memory read and write
  - 复杂的地址计算方式 Complex address calculation

复杂指令集 CISC Instruction Sets

条件码 Condition codes

- 作为算术和逻辑运算指令的副作用设置条件码 Set as side effect of arithmetic and logical instructions

哲学 Philosophy

- 增加指令执行“典型的”编程任务 Add instructions to perform “typical” programming tasks

精简指令集 RISC Instruction Sets

- 精简指令集计算机 Reduced Instruction Set Computer
- IBM的内部项目，后来由Hennessy(斯坦福)和Patterson (伯克利)大规模推广 Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)

指令更少，更简单 Fewer, simpler instructions

- 为了完成任务可能需要更多指令 Might take more to get given task done
- 可以执行指令用小且快速的硬件 Can execute them with small and fast hardware

面向寄存器的指令集 Register-oriented instruction set

- 有更多的寄存器（典型32个） Many more (typically 32) registers
- 用于参数传递、返回指针和临时变量存储 Use for arguments, return pointer, temporaries

精简指令集 RISC Instruction Sets

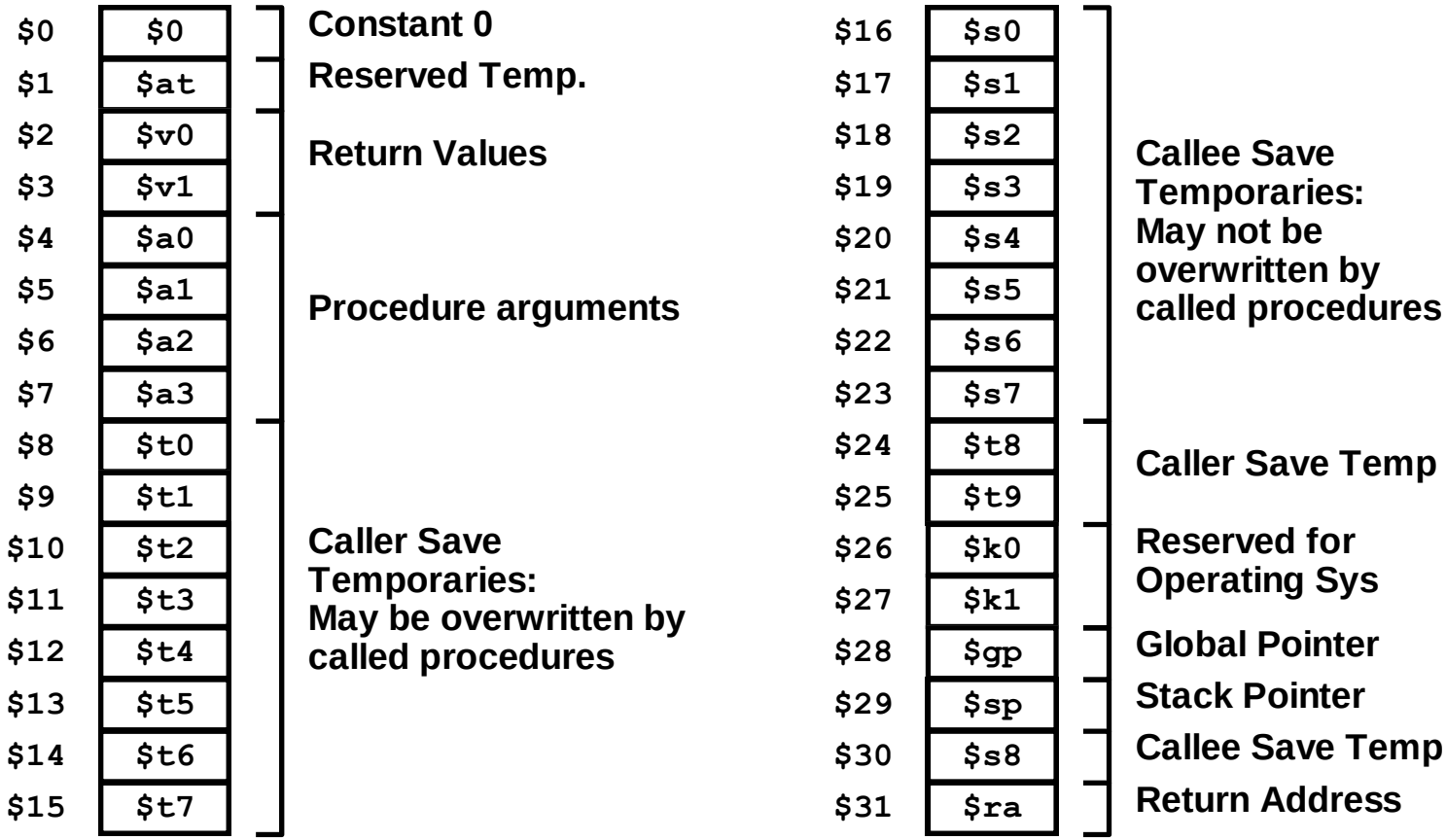
只有load和store指令能够访问内存 Only load and store instructions can access memory

- 类似于Y86-64的`mrmovq`和`rmmovq`指令 Similar to Y86-64 `mrmovq` and `rmmovq`

没有条件码 No Condition codes

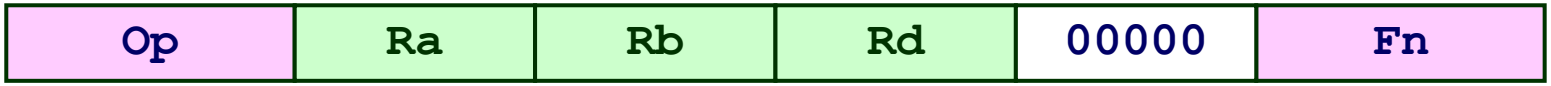
- 测试指令返回0/1在寄存器中 Test instructions return 0/1 in register

MIPS寄存器 MIPS Registers



MIPS指令示例
MIPS Instruction Examples

R-R

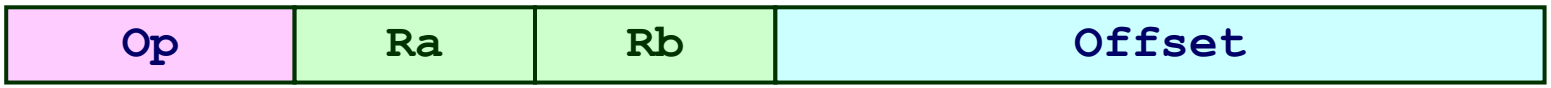



```
addu $3,$2,$1          # Register add: $3 = $2+$1
```

R-I

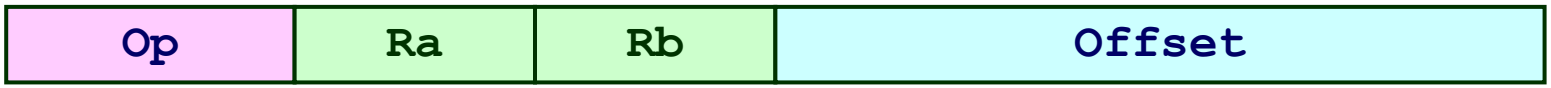

```
addu $3,$2, 3145       # Immediate add: $3 = $2+3145
sll $3,$2,2            # Shift left: $3 = $2 << 2
```

Branch




```
beq $3,$2,dest         # Branch when $3 = $2
```

Load/Store




```
lw $3,16($2)           # Load Word: $3 = M[$2+16]
sw $3,16($2)           # Store Word: M[$2+16] = $3
```

CISC vs. RISC

原始争论 Original Debate

- 争论十分激烈 Strong opinions!
- CISC支持者-编译器比较方便，很少的代码字节 CISC proponents---easy for compiler, fewer code bytes
- RISC支持者-优化编译器更佳，用简单芯片设计可以使运行更快 RISC proponents---better for optimizing compilers, can make run fast with simple chip design

CISC vs. RISC

目前状态 Current Status

- 对于台式机处理器，ISA选择并非技术问题 For desktop processors, choice of ISA not a technical issue
  - 有足够的硬件可以使任何事情运行的更快 With enough hardware, can make anything run fast
  - 代码兼容更重要 Code compatibility more important
- x86-64采纳了很多RISC功能 x86-64 adopted many RISC features
  - 更多的寄存器；使用它们作参数传递 More registers; use them for argument passing
- 对于嵌入式处理器，RISC更具优势 For embedded processors, RISC makes sense
  - 更小、更便宜、功耗更低 Smaller, cheaper, less power
  - 大部分手机使用ARM处理器 Most cell phones use ARM processor

小结 Summary

Y86-64指令集体系结构 Y86-64 Instruction Set Architecture

- 与x86-64类似的状态和指令 Similar state and instructions as x86-64
- 更简单的编码 Simpler encodings
- 某些方面介于CISC和RISC之间 Somewhere between CISC and RISC

ISA设计有多重要? How Important is ISA Design?

- 现在比以前要低一些 Less now than before
  - 有足够的硬件，几乎可以使任何事情都变得更快 With enough hardware, can make almost anything go fast

CS:APP Chapter 4
Computer Architecture
Logic Design
逻辑设计

任课教师：

宿红毅　张艳　黎有琦　颜珂

原作者：

Randal E. **Bryant and** David R. O'Hallaron

Carnegie Mellon University

逻辑设计概述
Overview of Logic Design

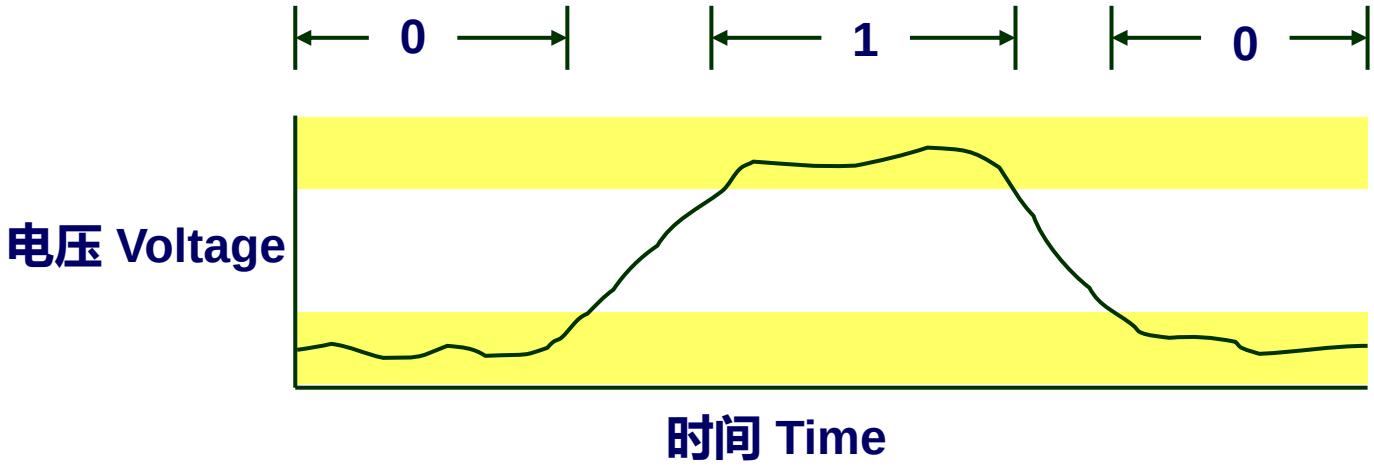
基础硬件需求 Fundamental Hardware Requirements

- 通信 Communication
  - 如何将值从一个地方传递到另一个地方 How to get values from one place to another
- 计算 Computation
- 存储 Storage

比特位是我们的朋友 Bits are Our Friends

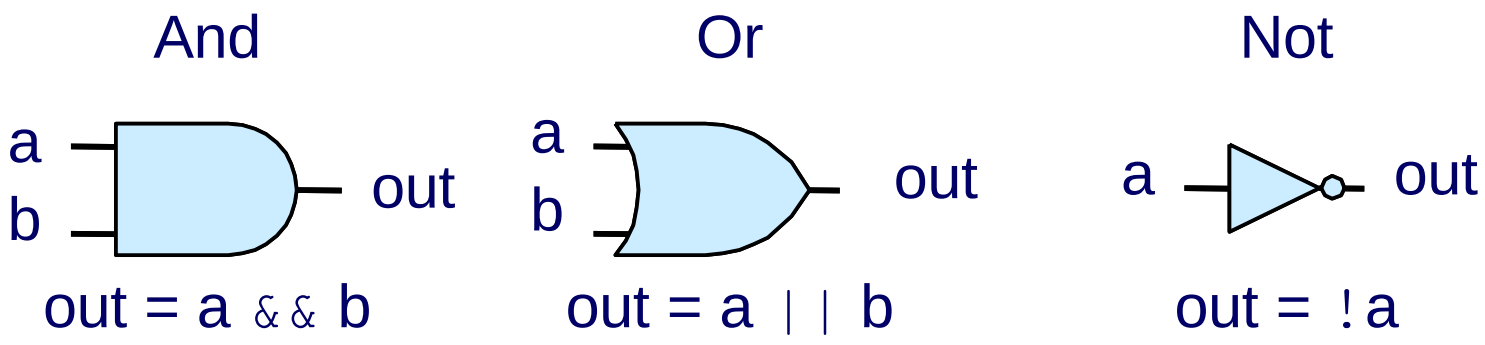
- 一切事情都可以用值0和1进行表达 Everything expressed in terms of values 0 and 1
- 通信 Communication
  - 在电缆上传递低或高电平 Low or high voltage on wire
- 计算 Computation
  - 计算布尔函数 Compute Boolean functions
- 存储 Storage
  - 存储信息比特位 Store bits of information

数字信号 Digital Signals

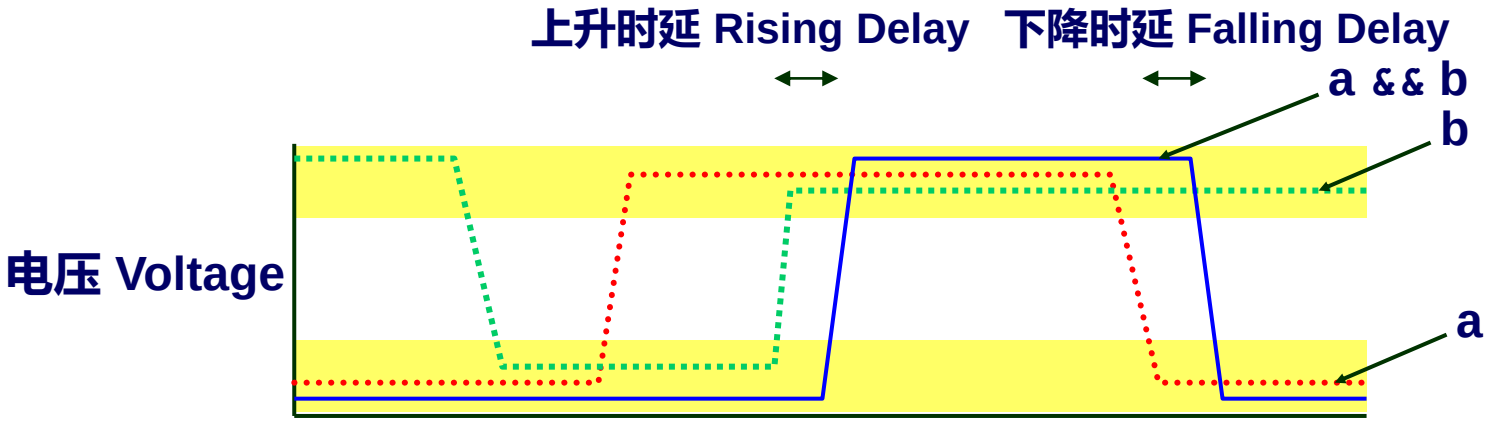


- 使用电压阈值抽取连续信号的离散值 Use voltage thresholds to extract discrete values from continuous signal
- 最简单的版本：1位信号 Simplest version: 1-bit signal
  - 要么在高电平范围（1）要么在低电平范围（0） Either high range (1) or low range (0)
  - 之间的电平值作为警戒范围 With guard range between them
- 不会受噪声或低质量电路元素较强影响 Not strongly affected by noise or low quality circuit elements
  - 可以使电路简单、小型和快速 Can make circuits simple, small, and fast

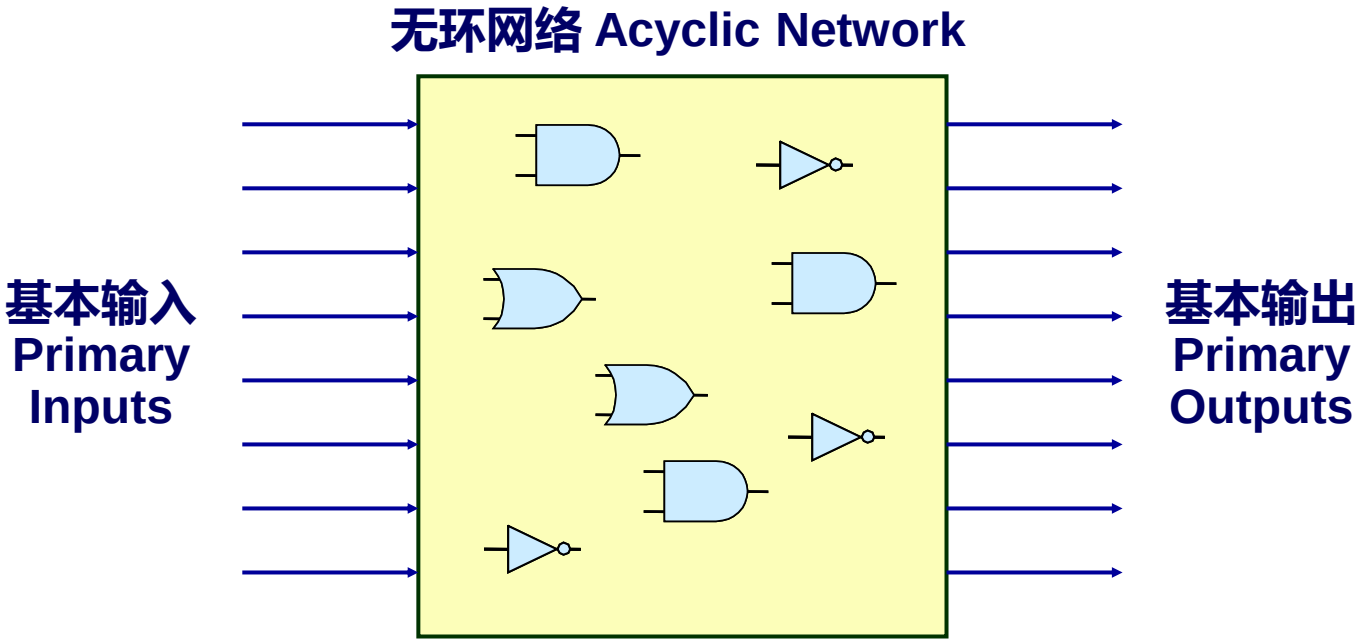
用逻辑门进行计算
Computing with Logic Gates



- 输出是输入的布尔函数 Outputs are Boolean functions of inputs
- 对输入的变化连续进行响应 Respond continuously to changes in inputs
  - 有一点小的时延 With some, small delay



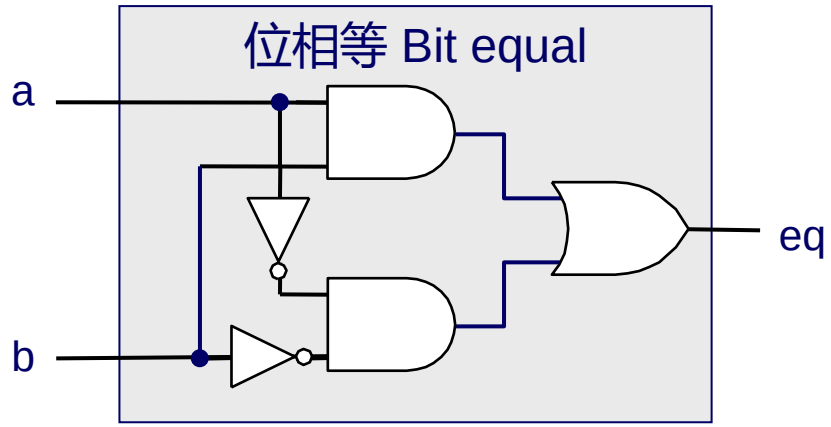
组合逻辑电路 Combinational Circuits



逻辑门的无环网络 Acyclic Network of Logic Gates

- 连续响应基本输入的变化 Continously responds to changes on primary inputs
- 基本输出变成（一些时延后）基本输入的布尔函数 Primary outputs become (after some delay) Boolean functions of primary inputs

位相等 Bit Equality



HCL表达式 HCL Expression

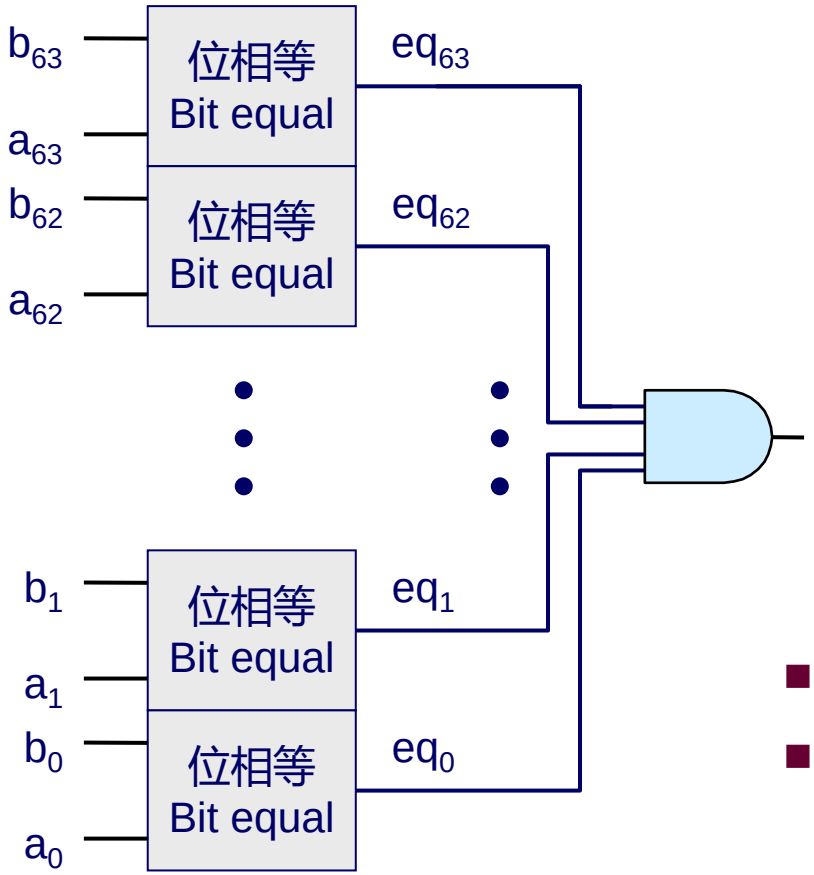
```
bool eq = (a&&b)||(!a&&!b)
```

- 如果a和b相等产生1 Generate 1 if a and b are equal

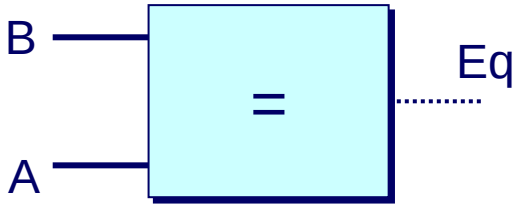
硬件控制语言（HCL） Hardware Control Language (HCL)

- 非常简单的硬件描述语言 Very simple hardware description language
  - 布尔操作与C语言逻辑操作有类似的语法 Boolean operations have syntax similar to C logical operations
- 我们将用它来描述处理器的控制逻辑 We'll use it to describe control logic for processors

字相等 Word Equality



字级表示 Word-Level Representation

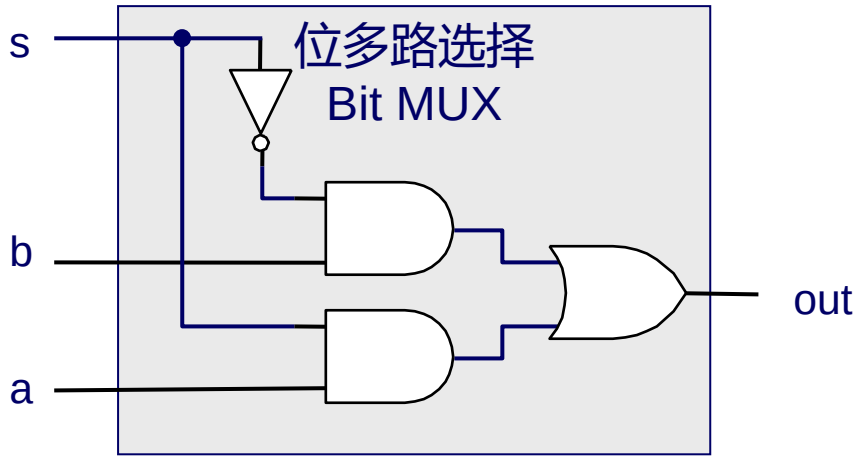


HCL表示 HCL Representation

```
bool Eq = (A == B)
```

- **64位字长 64-bit word size**
- **HCL表示 HCL representation**
  - **相等操作 Equality operation**
  - **产生布尔值 Generates Boolean value**

位级多路选择器 Bit-Level Multiplexor

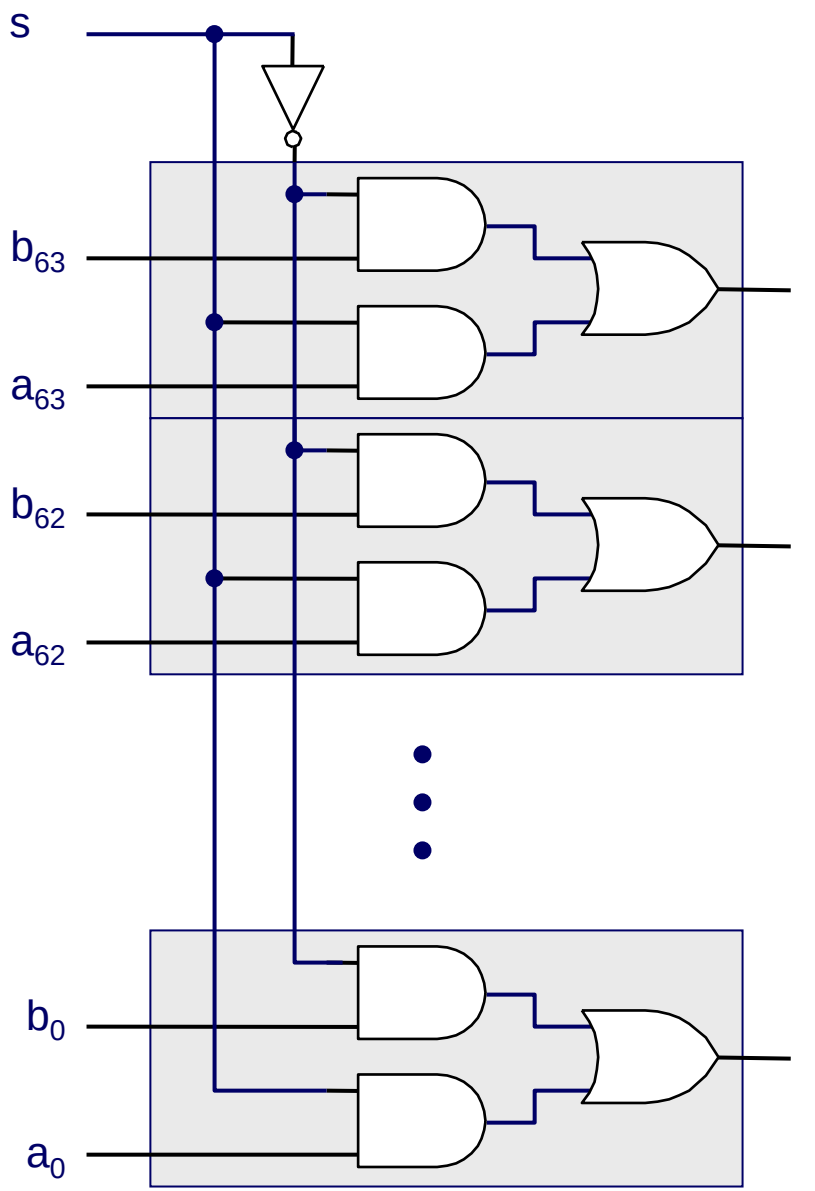


HCL表达式 HCL Expression

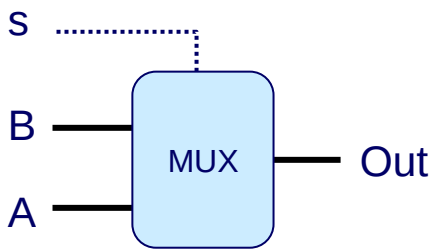
```
bool out = (s&&a)||(!s&&b)
```

- **控制信号s Control signal s**
- **数据信号a和b Data signals a and b**
- **当s为1时输出a，s为0时输出为b Output a when s=1, b when s=0**

字级多路选择器 Word Multiplexor



字级表示 Word-Level Representation



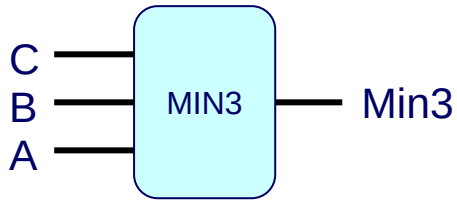
HCL表示 HCL Representation

```
int Out = [
  s : A;
  1 : B;
];
```

- 根据控制信号s选择输入字A还是B Select input word A or B depending on control signal s
- HCL表达式 HCL representation
  - Case表达式 Case expression
  - 一系列测试：值对 Series of test : value pairs
  - 第一个成功的测试作为输出值 Output value for first successful test

HCL字级示例
HCL Word-Level Examples

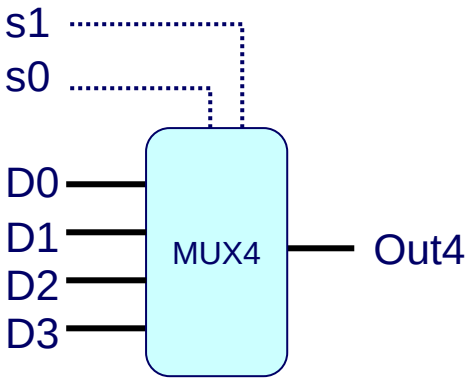
3个字中最小值 Minimum of 3 Words




```
int Min3 = [
  A < B && A < C : A;
  B < A && B < C : B;
  1              : C;
];
```

- 发现三个输入字中最小的 Find minimum of three input words
- HCL case表达式 HCL case expression
- 最后的case确保匹配 Final case guarantees match

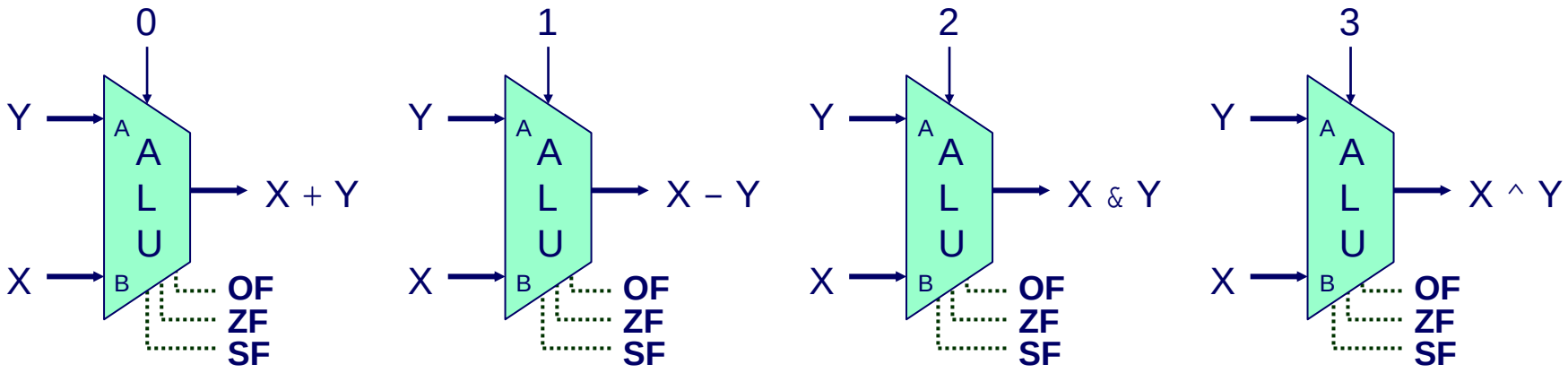
4路选择器 4-Way Multiplexor




```
int Out4 = [
  !s1&&!s0: D0;
  !s1     : D1;
  !s0     : D2;
  1       : D3;
];
```

- 根据两个控制位选择4个输入之一 Select one of 4 inputs based on two control bits
- HCL case表达式 HCL case expression
- 假设顺序匹配简化测试 Simplify tests by assuming sequential matching

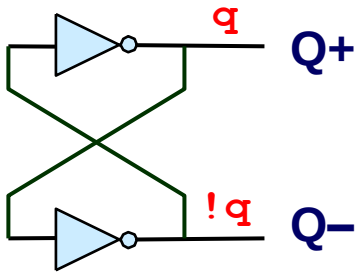
算术逻辑单元 Arithmetic Logic Unit



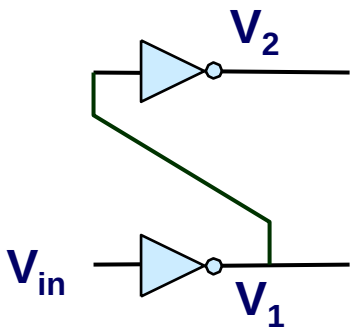
- 组合逻辑 Combinational logic
  - 连续响应输入 Continuously responding to inputs
- 控制信号选择计算的功能 Control signal selects function computed
  - 对应于Y86-64中的4种算术/逻辑运算 Corresponding to 4 arithmetic/logical operations in Y86-64
- 也计算条件码的值 Also computes values for condition codes

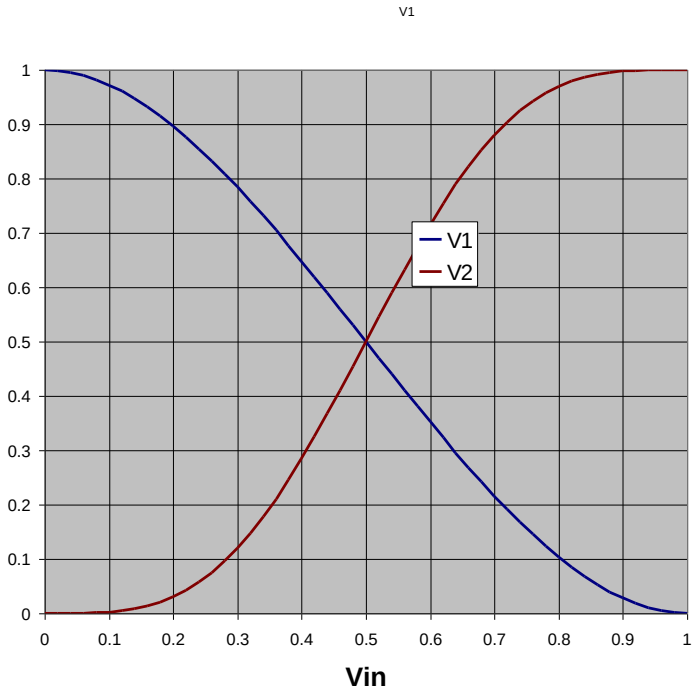
存储1位 Storing 1 Bit

双稳态元件 Bistable Element



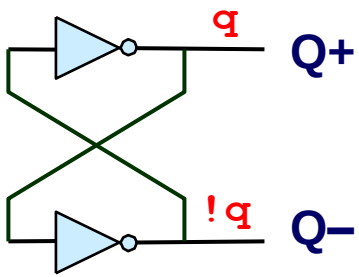
q = 0 or 1



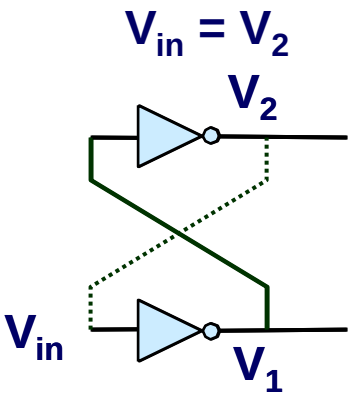


存储1位（续） Storing 1 Bit (cont.)

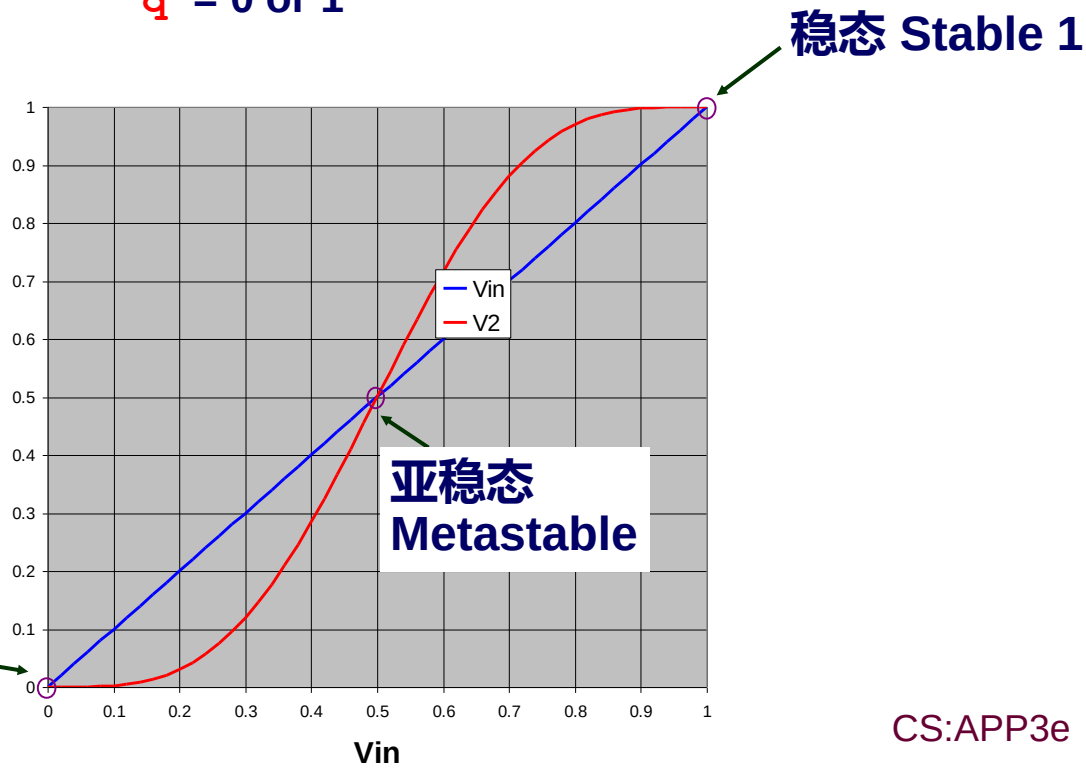
双稳态元件 Bistable Element



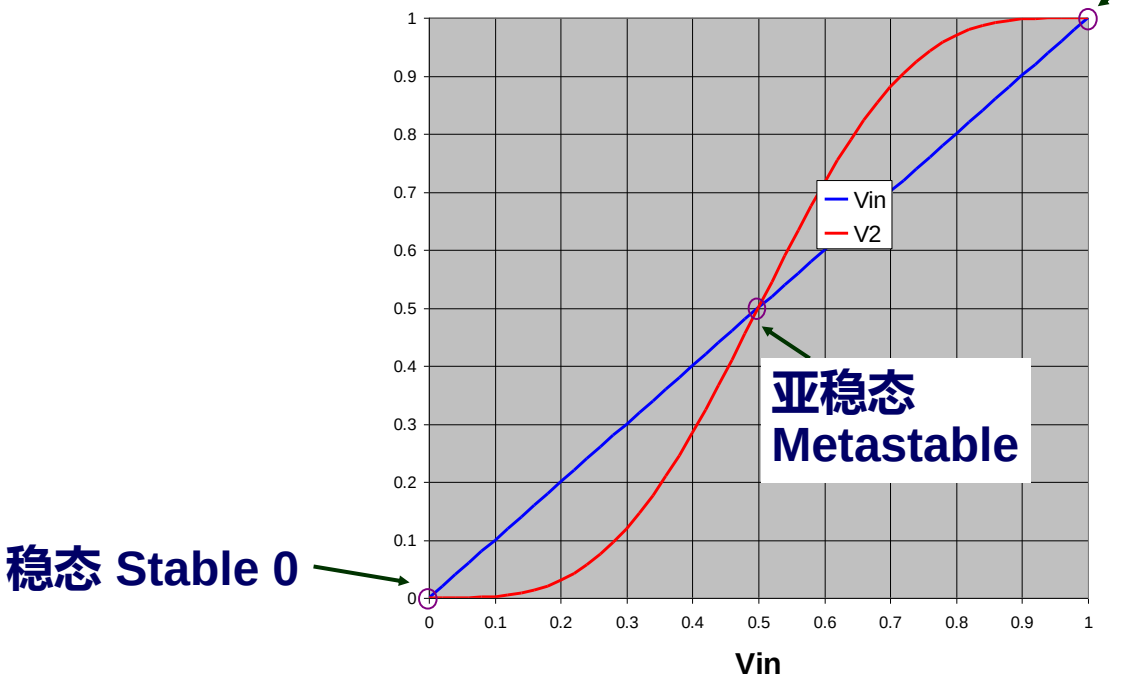
q = 0 or 1

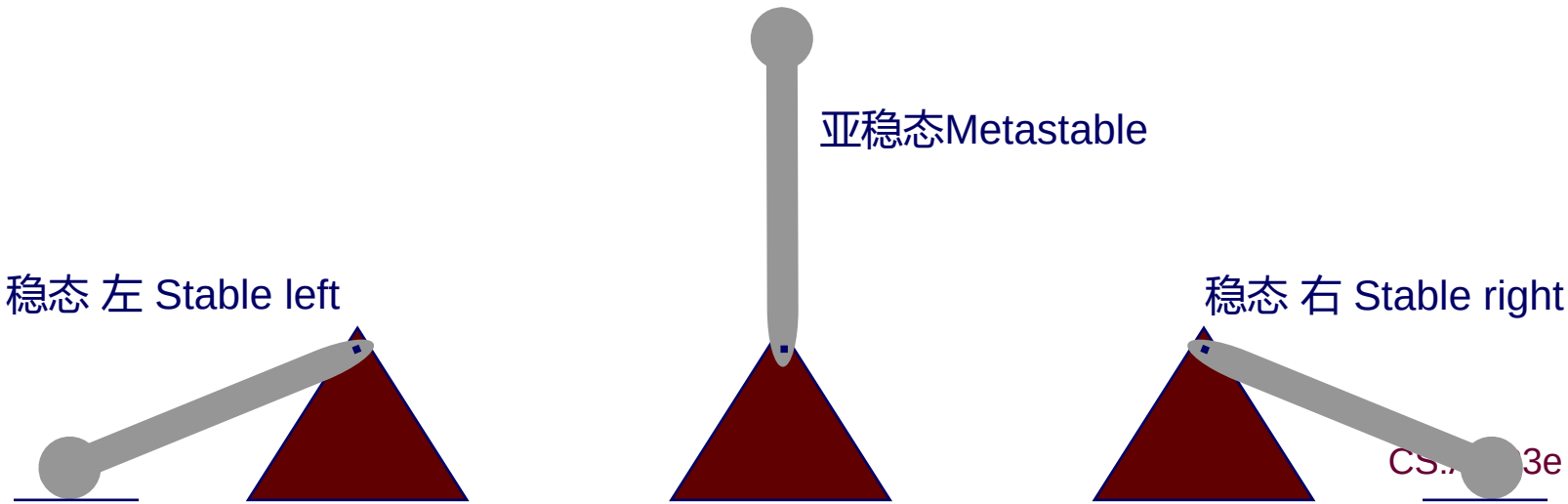


稳态 Stable 0



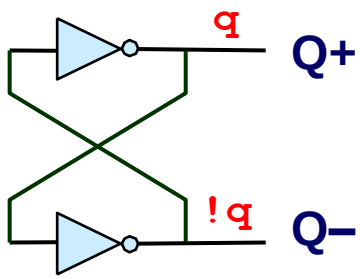
物理类比 Physical Analogy





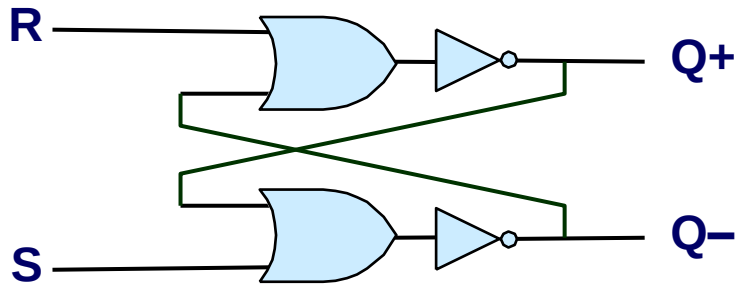
存储和访问1位
Storing and Accessing 1 Bit

双稳态元件 Bistable Element

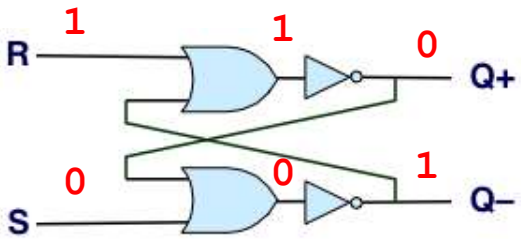


q = 0 or 1

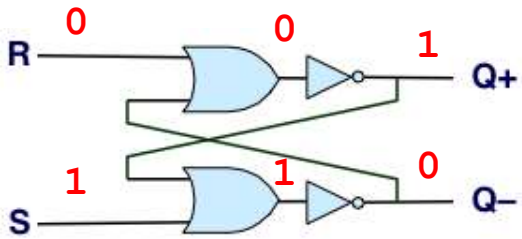
R-S锁存器 R-S Latch



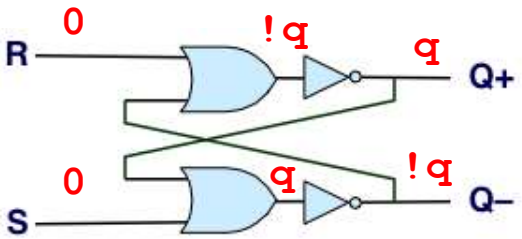
置0 Resetting



置1 Setting

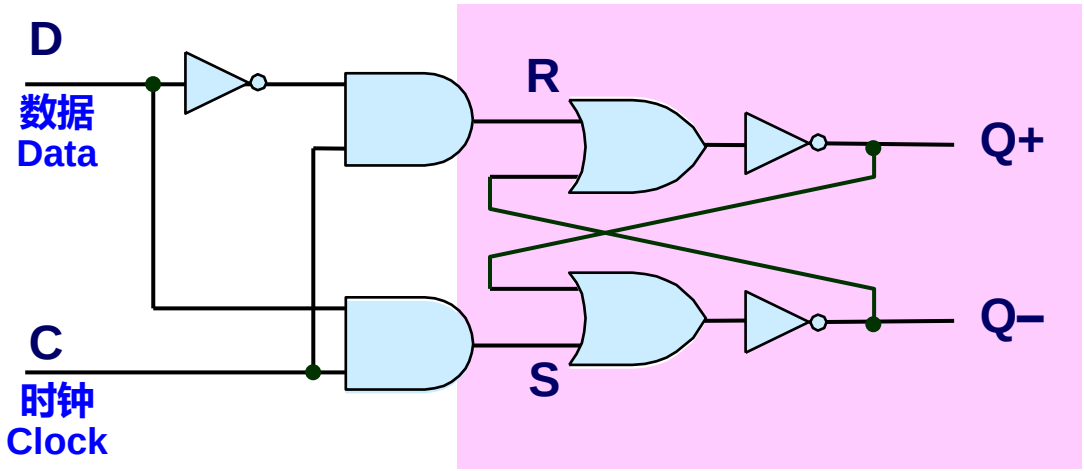


存储 Storing

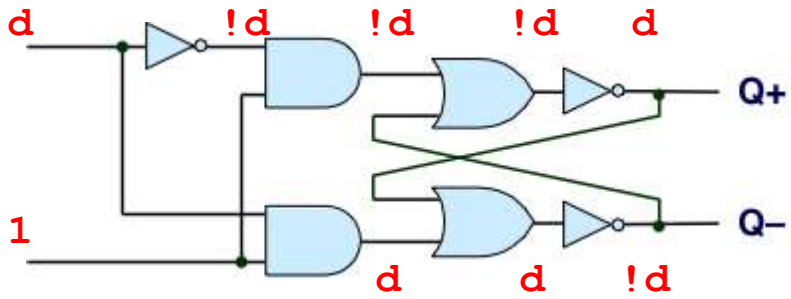


1位锁存器 1-Bit Latch

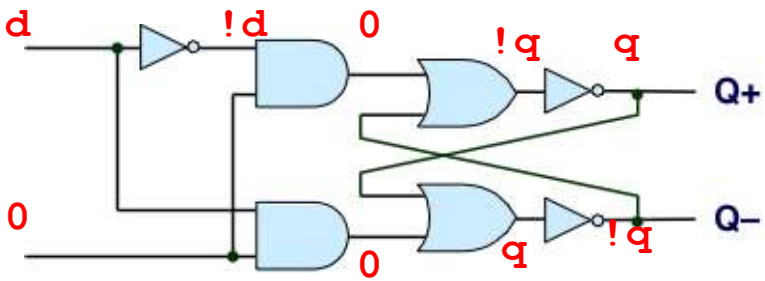
D锁存器 D Latch



锁定 Latching

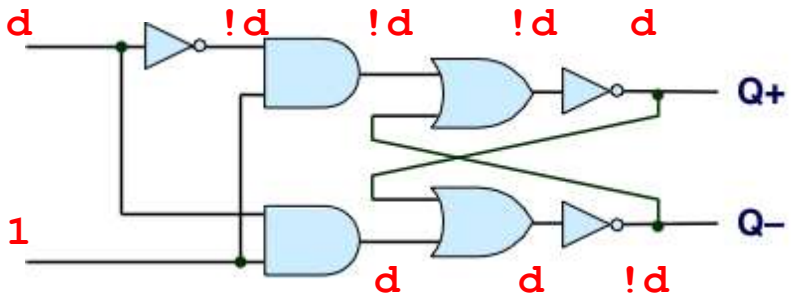


存储 Storing

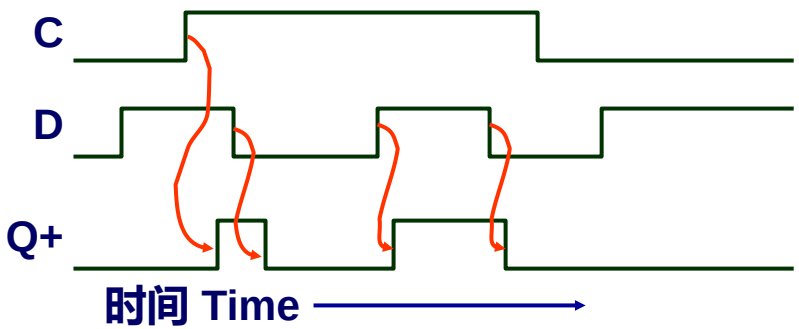


透明的1位锁存器 Transparent 1-Bit Latch

锁定 Latching

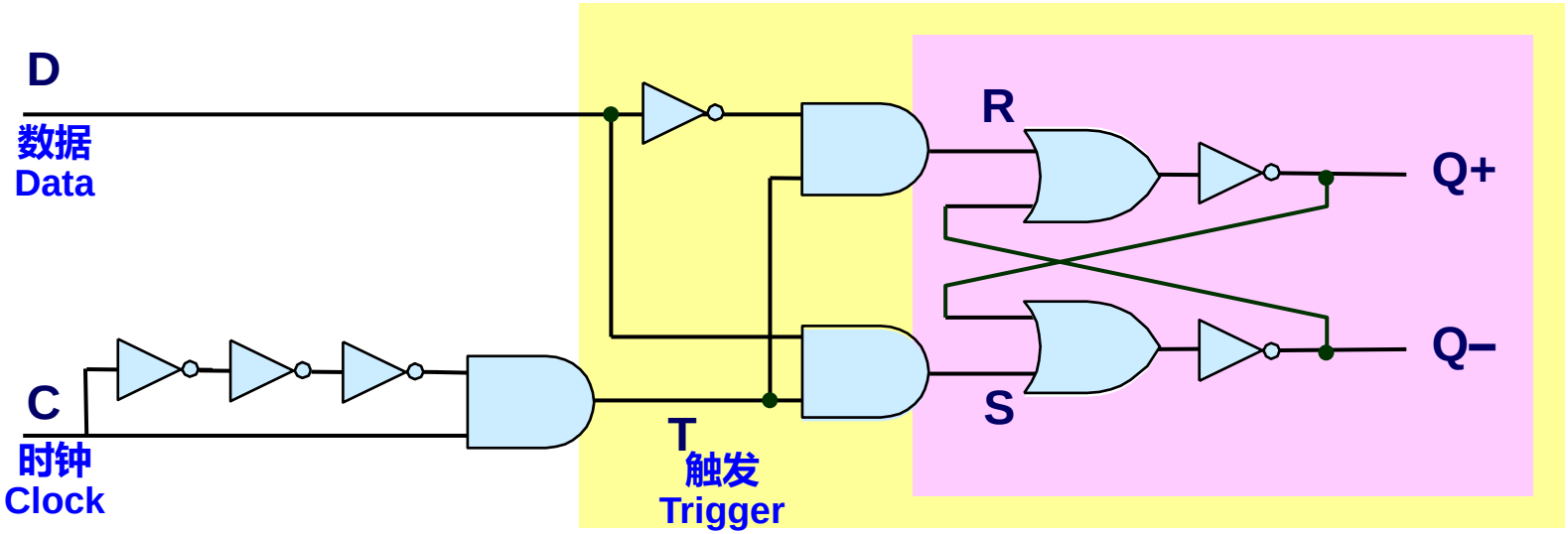


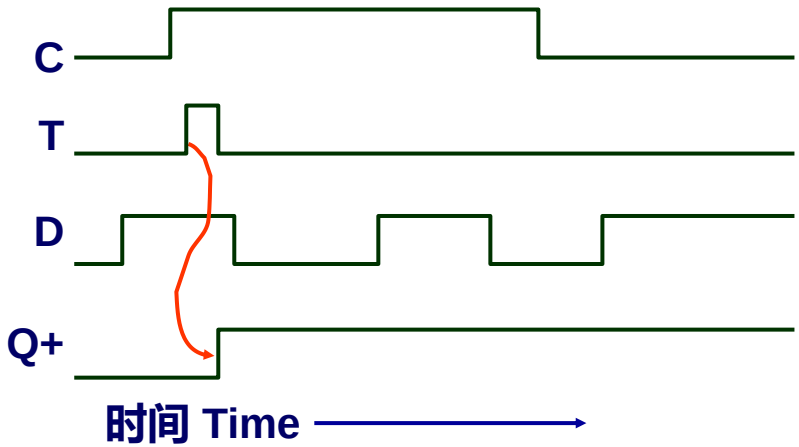
改变D Changing D



- 当在锁定模式时，组合逻辑传播从D到Q+和Q- When in latching mode, combinational propogation from D to Q+ and Q–
- 锁存的值取决于当C下降时D的值 Value latched depends on value of D as C falls

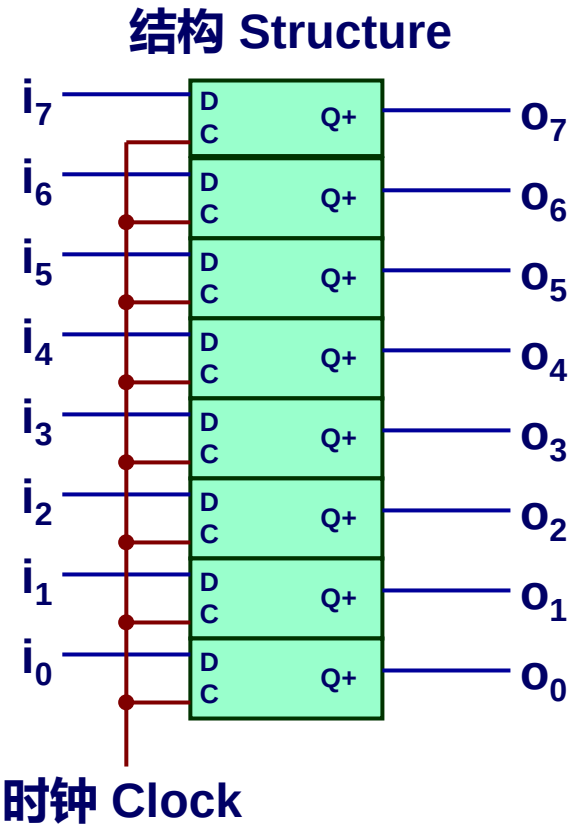
边沿触发的锁存器
Edge-Triggered Latch

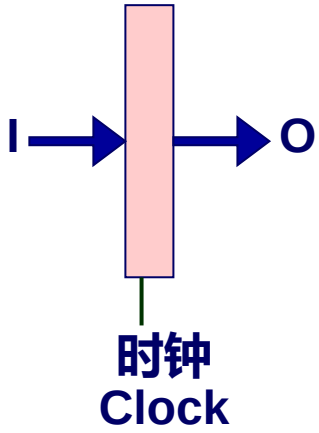




- 仅很短时间处于锁存模式 Only in latching mode for brief period
  - 上升时钟沿 Rising clock edge
- 锁存的值取决于当时钟上升时的数据 Value latched depends on data as clock rises
- 输出保持稳态在所有其它时间 Output remains stable at all other times

寄存器 Registers



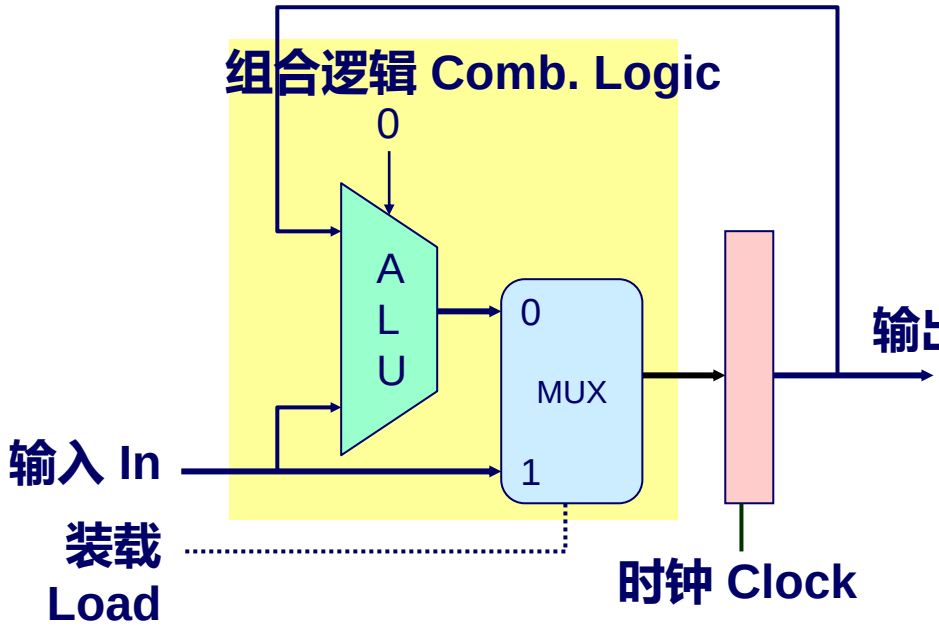


- 存储数据字 Stores word of data
  - 不同于汇编代码中看到的程序寄存器 Different from *program registers* seen in assembly code
- 边沿触发的锁存器集合 Collection of edge-triggered latches
- 在时钟上升沿装载输入 Loads input on rising edge of clock

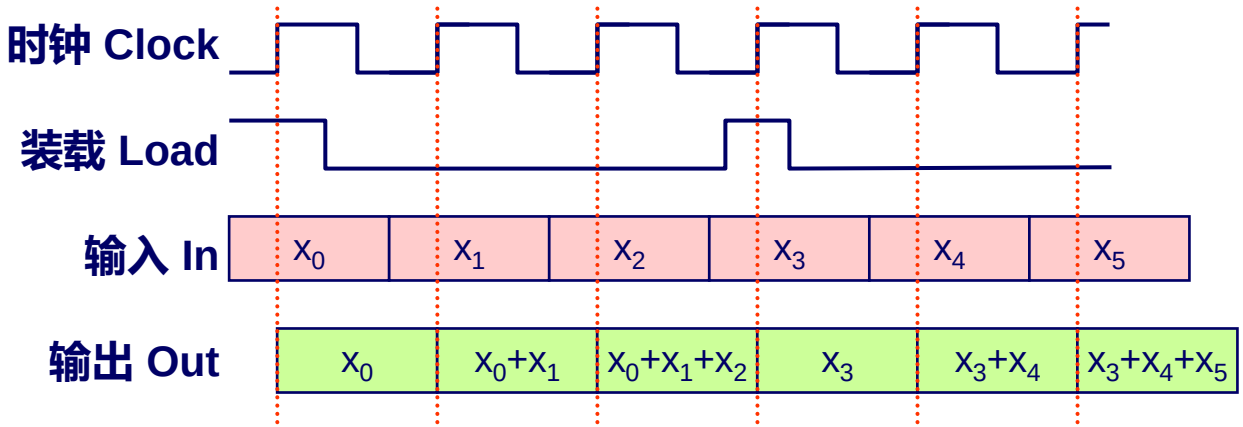
寄存器操作 Register Operation

- **存储数据位 Stores data bits**
- **大多数时间充当输入和输出之间的障碍 For most of time acts as barrier between input and output**
- **当时钟上升时，装载输入 As clock rises, loads input**

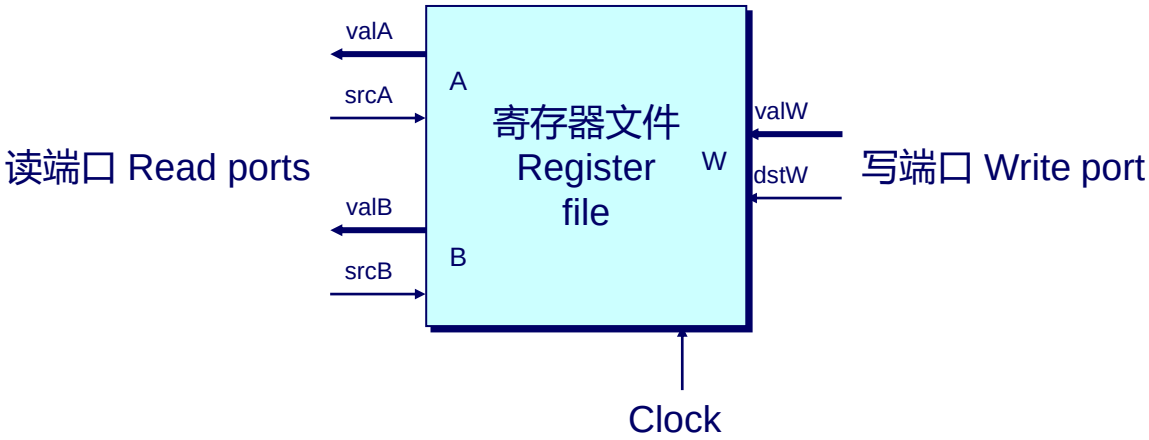
状态机示例 State Machine Example



- 累加器电路 Accumulator circuit
- 每个周期装载或累加 Load or accumulate on each cycle

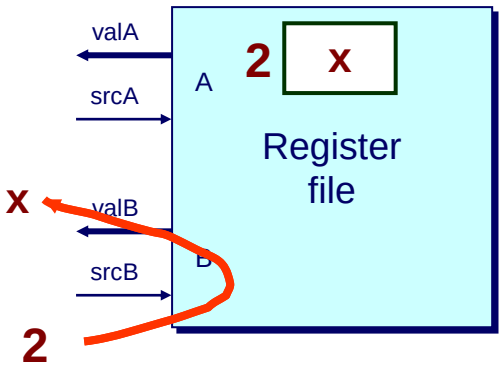


寄存器文件 Register File



- **存储单个字 Stores single word**
  - **地址输入指定读或写哪个字 Address input specifies which word to read or write**
- **寄存器文件 Register file**
  - **存储程序寄存器的值 Holds values of program registers**
  - **`%rax`, `%rsp`, etc.**
  - **寄存器标识符服务作为地址 Register identifier serves as address**
    - » ID 15（0xF）暗含不执行读或写 ID 15 (0xF) implies no read or write performed
- **多个端口 Multiple Ports**
  - **可以一个周期读和/或写多个字 Can read and/or write multiple words in one cycle**
    - » 每个有单独的地址和数据输入/输出 Each has separate address and data input/output

寄存器文件时序 Register File Timing

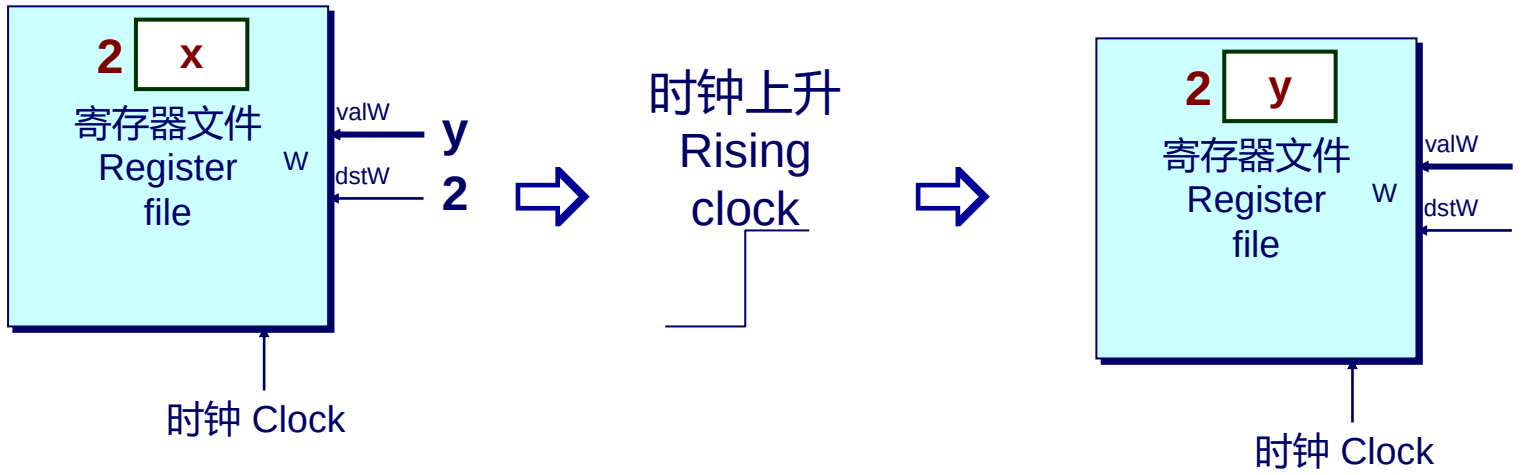


读 Reading

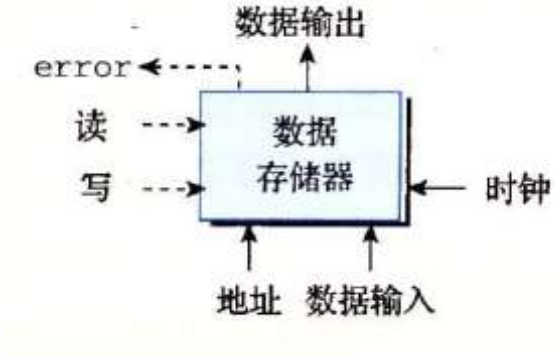
- 类似组合逻辑 Like combinational logic
- 根据输入地址产生输出数据 Output data generated based on input address
  - 一些时延后 After some delay

写 Writing

- 类似寄存器 Like register
- 仅在时钟上升沿更新 Update only as clock rises



随机访问存储器
Random-Access Memory



- **存储多个字 Stores multiple words**
  - **地址输入指定读或写哪个字 Address input specifies which word to read or write**
- **读写操作**
  - **当地址输入变化时读出字 Read word when address input changes**
    - » 给定地址，并将读信号设置为1，写信号设置为0，经过一定延迟或，输出数据在“数据输出线”上。
  - **当时钟上升时写入字 Write word as clock rises**
    - » 设定地址，写入的数据放在“数据输入”信号线上，设置写信号为1，控制时钟信号，时钟信号上升时候写入数据

硬件控制语言 Hardware Control Language

- 非常简单的硬件描述语言 Very simple hardware description language
- 仅可以表达有限的硬件操作 Can only express limited aspects of hardware operation
  - 我们想要探索和修改的部分 Parts we want to explore and modify

数据类型 Data Types

- `bool`: Boolean
  - `a`, `b`, `c`, …
- `int`: words
  - `A`, `B`, `C`, …
  - 不指定字长-字节还是64位字 Does not specify word size---bytes, 64-bit words, …

语句 Statements

- `bool a = bool-expr ;`
- `int A = int-expr ;`
- 按照返回值类型来分类 Classify by type of value returned

HCL操作 HCL Operations

布尔表达式 Boolean Expressions

- 逻辑操作 Logic Operations
  - `a && b, a || b, !a`
- 字比较 Word Comparisons
  - `A == B, A != B, A < B, A <= B, A >= B, A > B`
- 集合成员关系 Set Membership
  - `A in { B, C, D }`
    - » 等同于 Same as `A == B || A == C || A == D`

字表达式 Word Expressions

- Case表达式 Case expressions
  - `[ a : A; b : B; c : C ]`
  - 按顺序评估测试表达式 Evaluate test expressions `a`, `b`, `c`, … in sequence
  - 返回第一个成功测试的字表达式 Return word expression `A`, `B`, `C`, … for first successful test

小结 Summary

计算 Computation

- 由组合逻辑执行 Performed by combinational logic
- 计算布尔函数 Computes Boolean functions
- 连续对输入变化做出输出反应 Continuously reacts to input changes

存储 Storage

- 寄存器 Registers
  - 存储单个字 Hold single words
  - 在时钟上升时装载 Loaded as clock rises
- 随机访问存储器 Random-access memories
  - 存储多个字 Hold multiple words
  - 可能有多个读或写端口 Possible multiple read or write ports
  - 当地址输入变化时读出字 Read word when address input changes
  - 当时钟上升时写入字 Write word as clock rises

CS:APP Chapter 4
Computer Architecture
Sequential Implementation
顺序实现

任课教师:

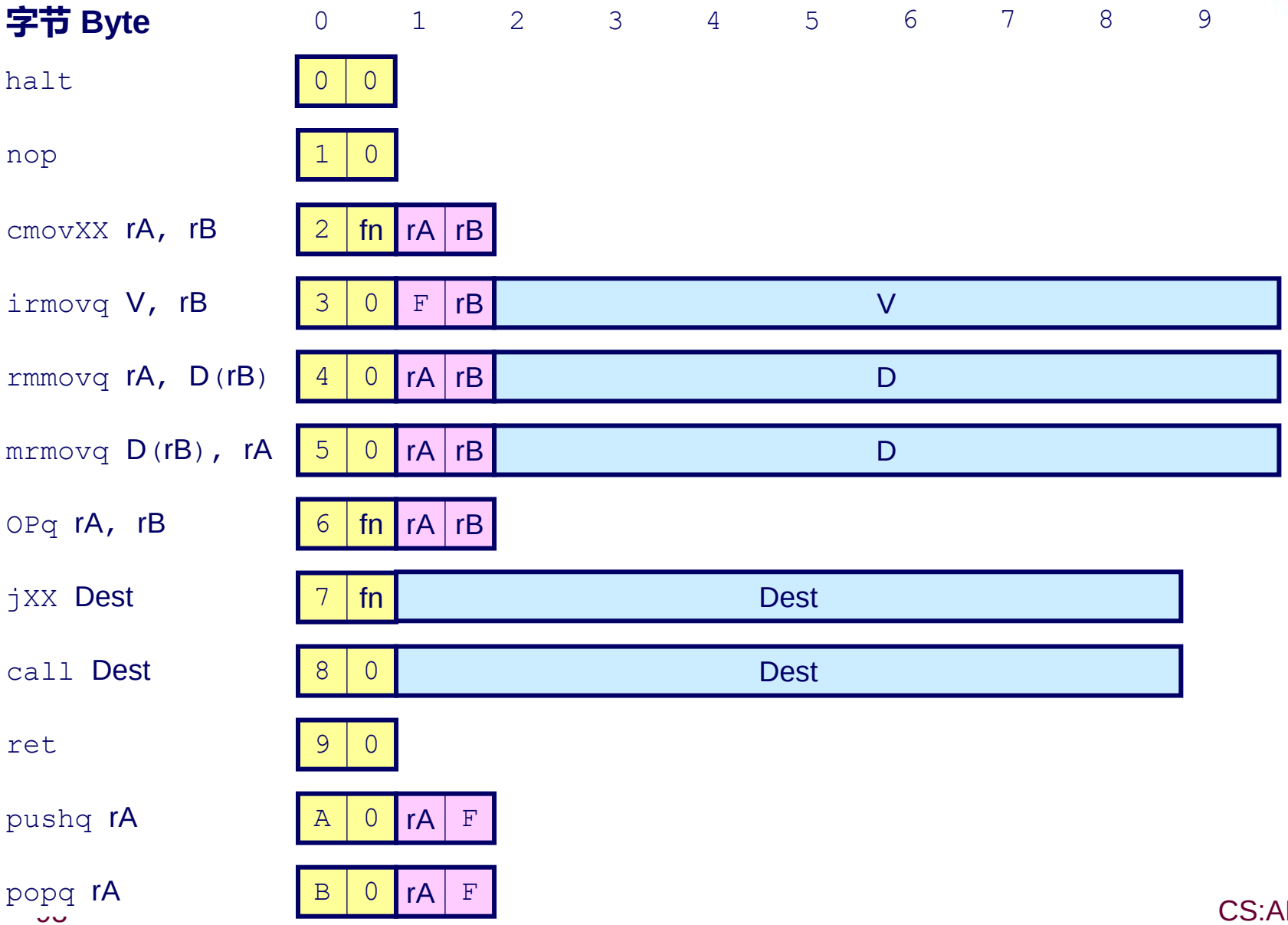
宿红毅　张艳　黎有琦　颜珂

原作者:

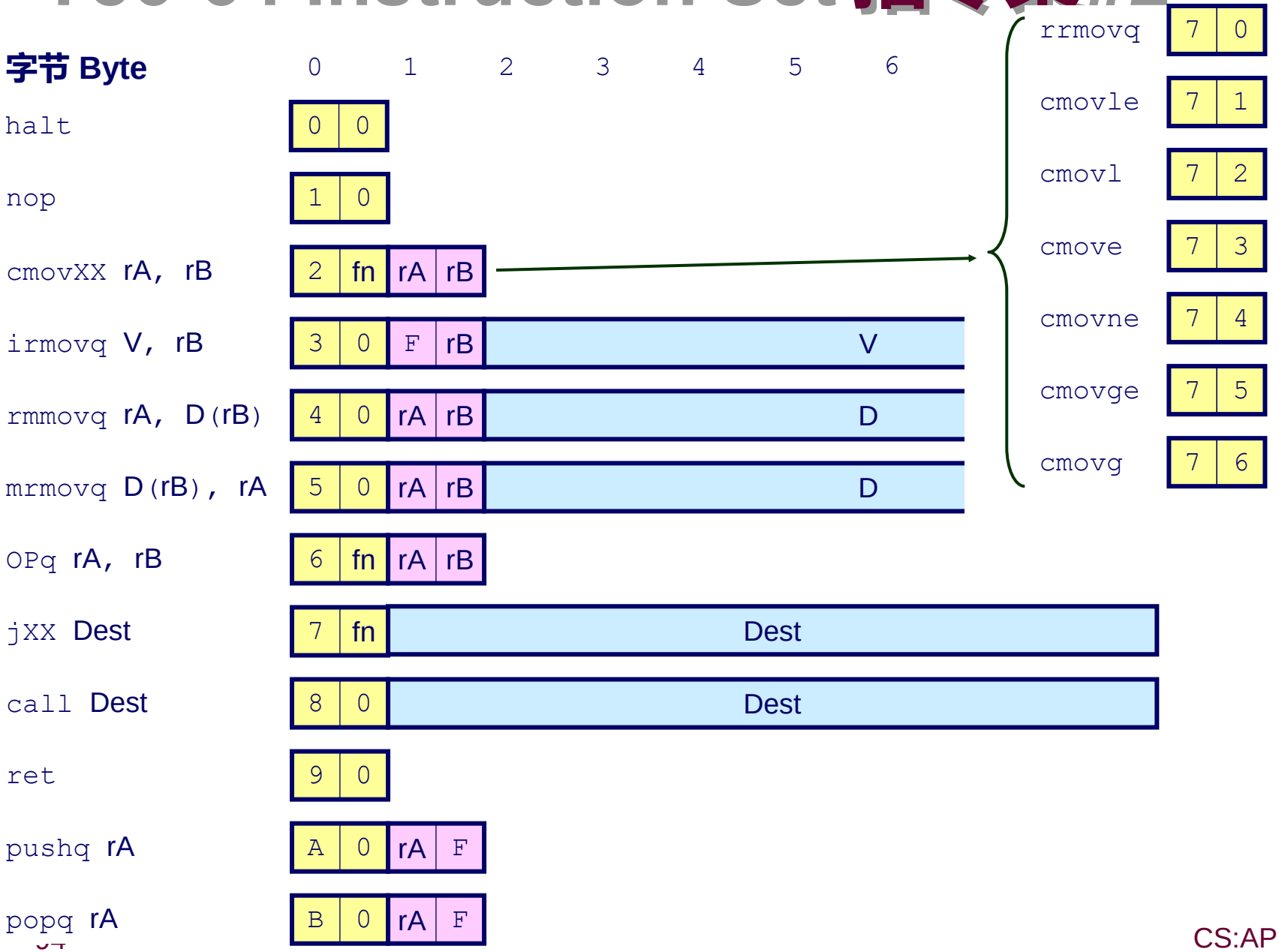
Randal E. **Bryant and** David R. O'Hallaron

Carnegie Mellon University

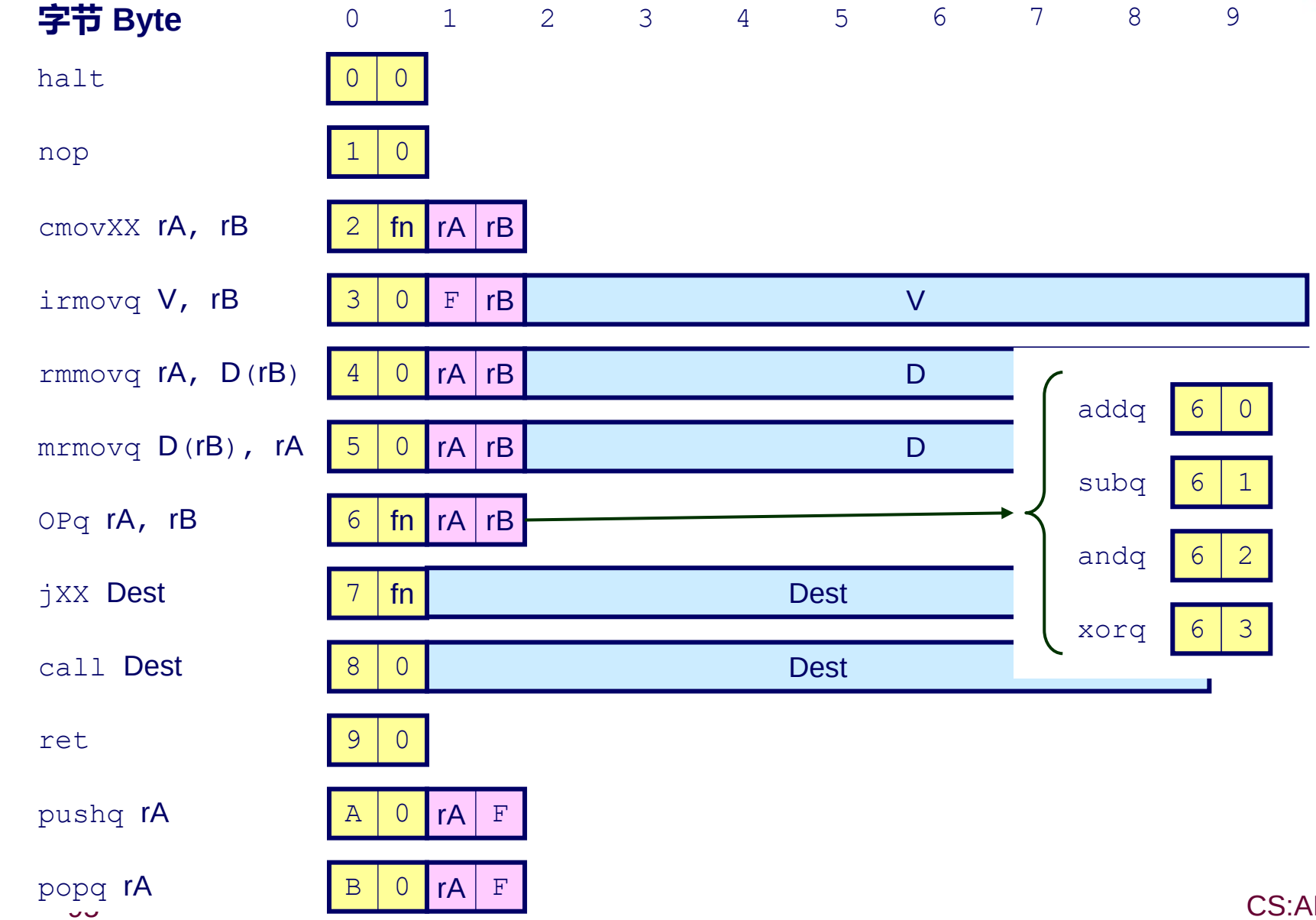
Y86-64 Instruction Set 指令集#1



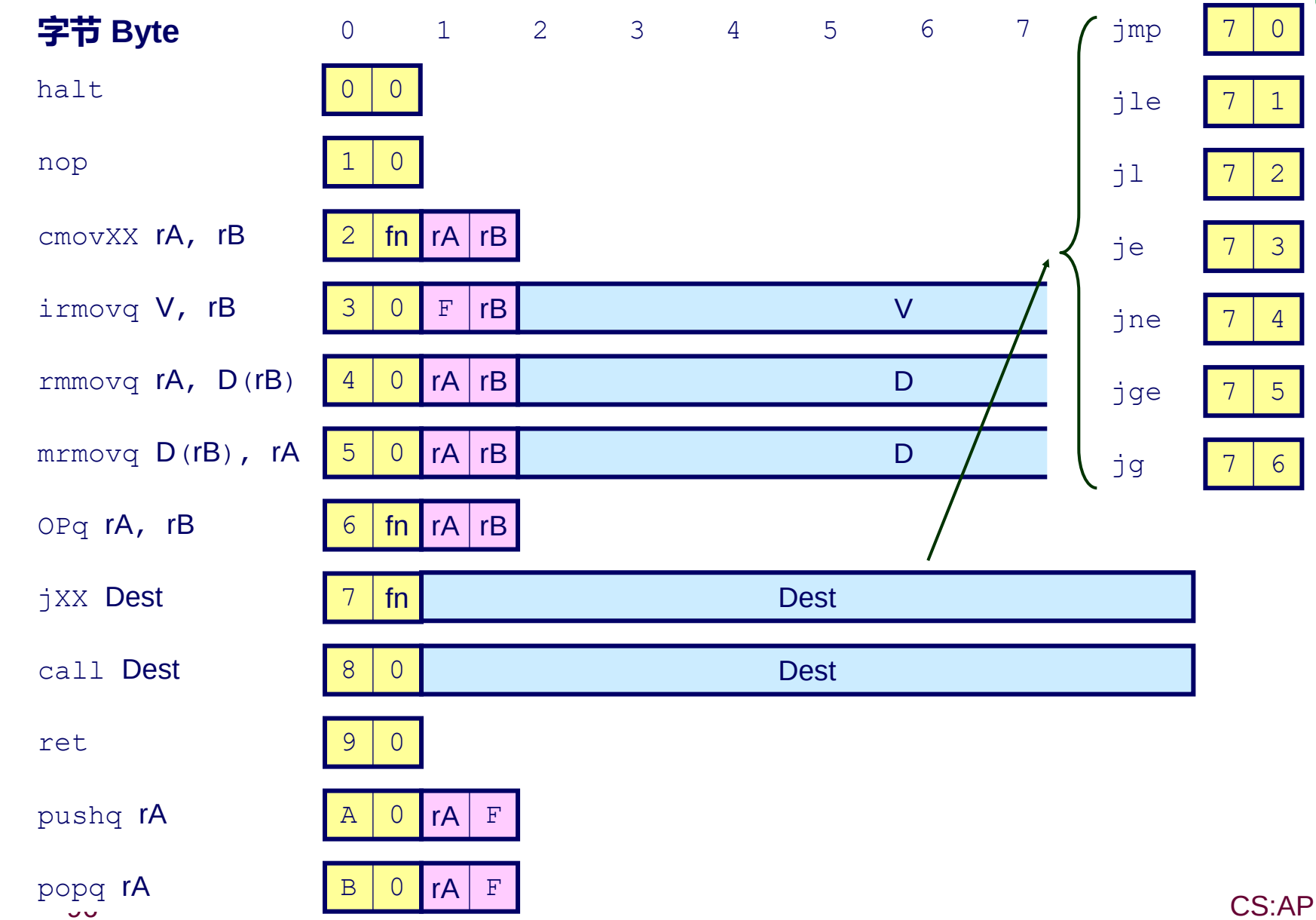
Y86-64 Instruction Set 指令集#2



Y86-64 Instruction Set 指令集#3



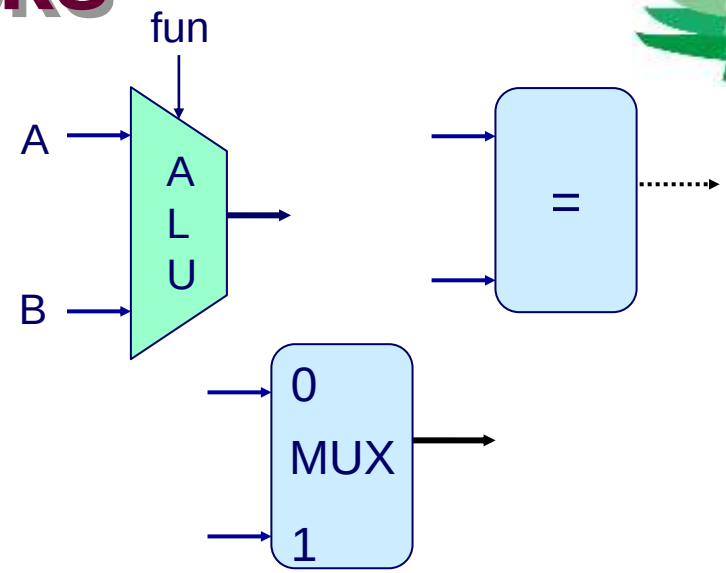
Y86-64 Instruction Set 指令集#4



构建块 Building Blocks

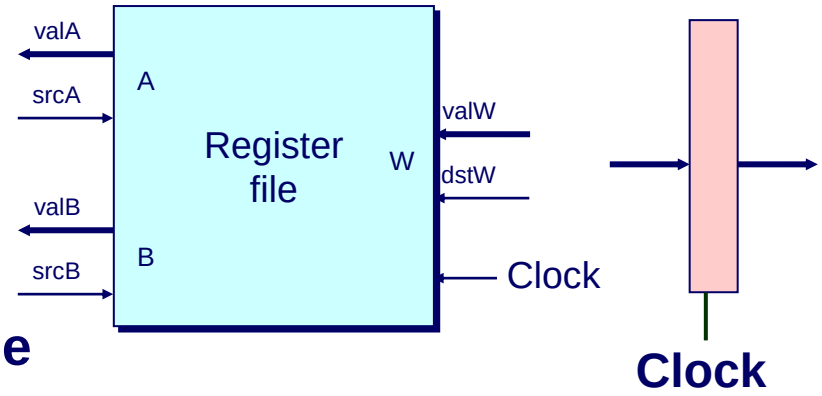
组合逻辑 Combinational Logic

- 计算输入的布尔函数 Compute Boolean functions of inputs
- 连续响应输入的变化 Continuously respond to input changes
- 对数据进行操作并实现控制 Operate on data and implement control



存储元素 Storage Elements

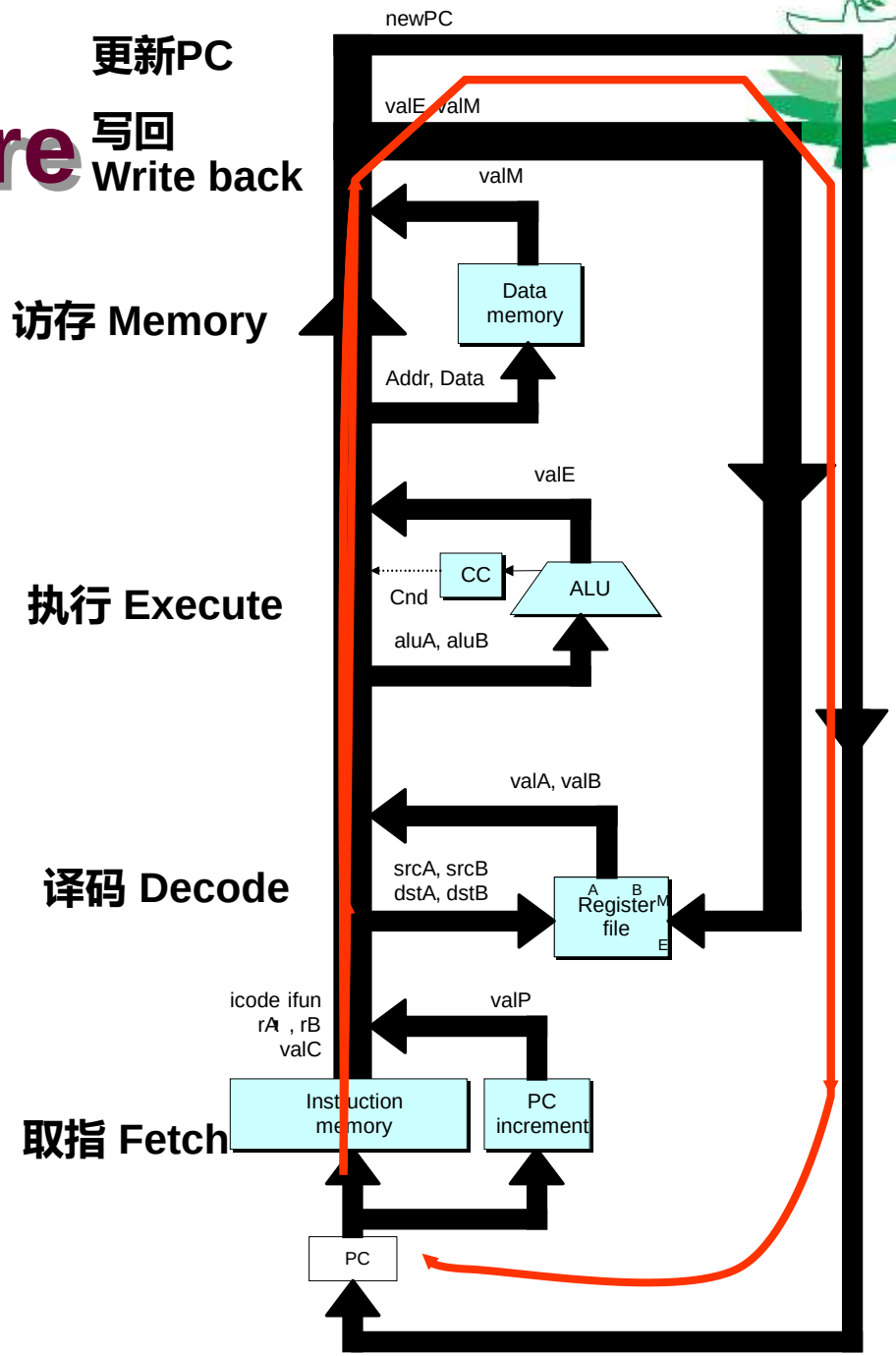
- 存储若干位 Store bits
- 可寻址的内存 Addressable memories
- 非寻址的寄存器 Non-addressable registers
- 仅在时钟上升时装载 Loaded only as clock rises



顺序硬件结构 SEQ Hardware Structure

状态 State

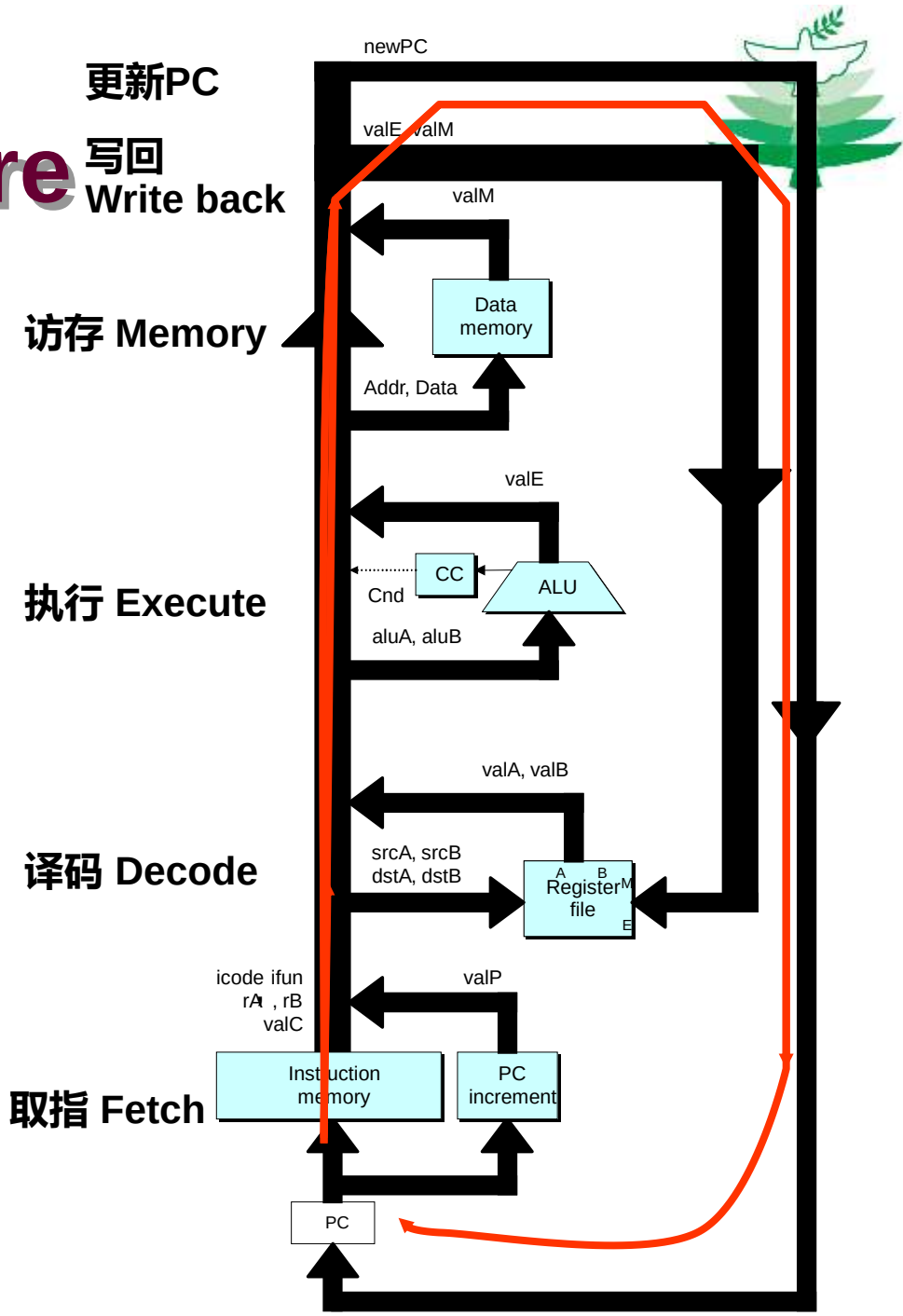
- 程序计数器寄存器 Program counter register (PC)
- 条件码寄存器 Condition code register (CC)
- 寄存器文件（堆） Register File
- 内存 Memories
  - 访问同样的内存空间 Access same memory space
  - 数据：读/写程序数据 Data: for reading/writing program data
  - 指令：读指令 Instruction: for reading instructions



顺序硬件结构 SEQ Hardware Structure

指令流 Instruction Flow

- 读PC指定地址处的指令 Read instruction at address specified by PC
- 分成阶段处理 Process through stages
- 更新程序计数器 Update program counter



顺序阶段 SEQ Stages

取指 Fetch

- 从指令内存读指令 Read instruction from instruction memory

译码 Decode

- 读程序寄存器 Read program registers

执行 Execute

- 计算值或地址 Compute value or address

访存 Memory

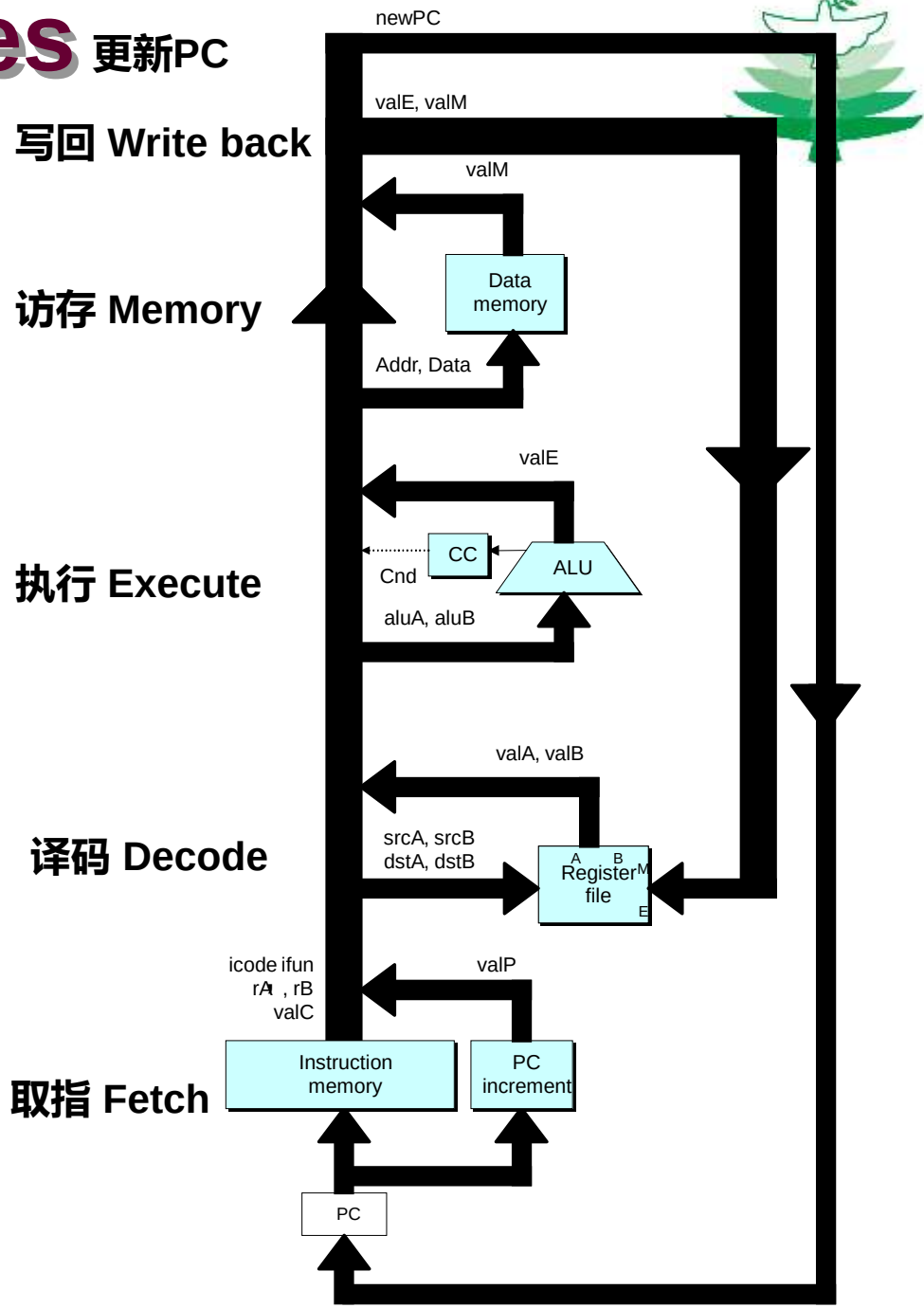
- 读或写数据 Read or write data

写回 Write Back

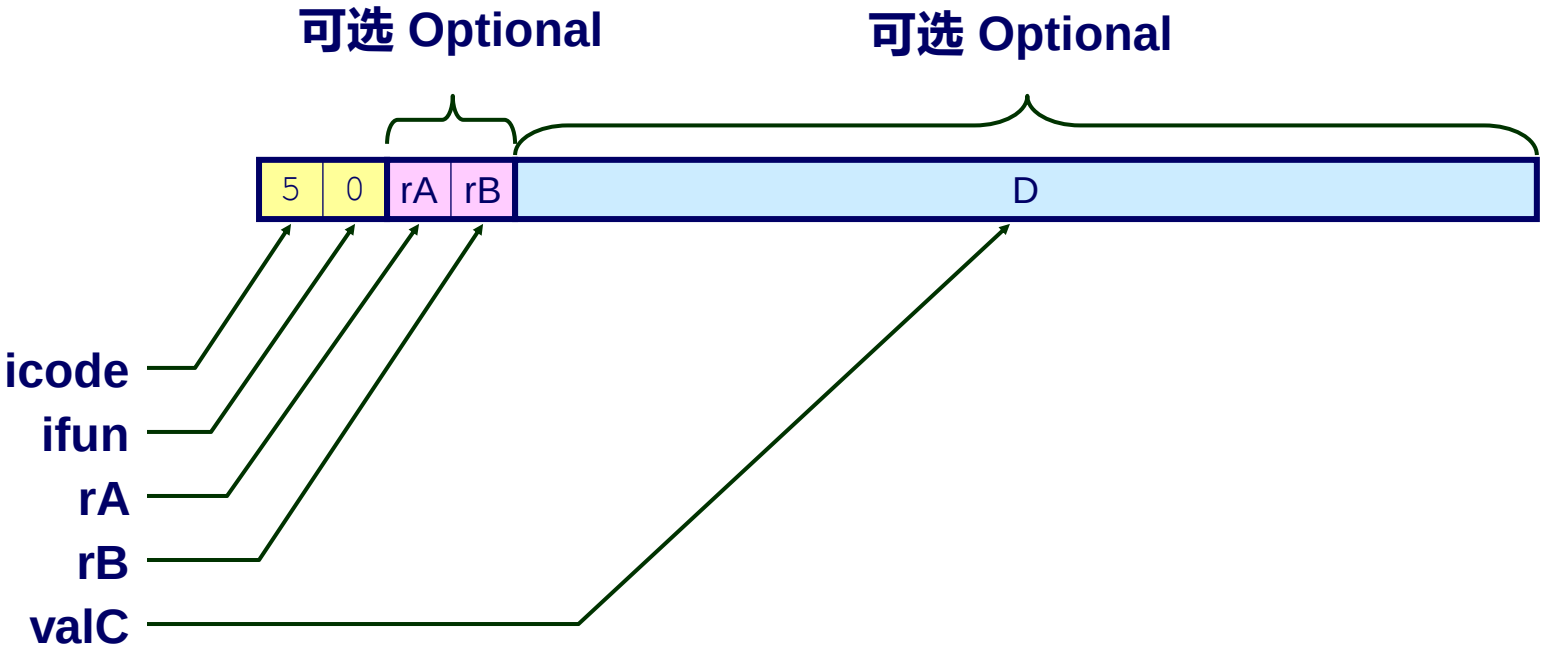
- 写程序寄存器 Write program registers

更新PC

- 更新程序计数器 Update program counter



指令译码 Instruction Decoding



指令格式 Instruction Format

- 指令字节 Instruction byte — icode:ifun
- 可选的寄存器字节 Optional register byte — rA:rB
- 可选的常量字 Optional constant word — valC

执行算术/逻辑操作
Executing Arith./Logical Operation

取指 Fetch

- 读2个字节 Read 2 bytes

译码 Decode

- 读操作数寄存器 Read operand registers

执行 Execute

- 执行运算 Perform operation
- 设置条件码 Set condition codes

访存 Memory

- 无操作 Do nothing

写回 Write back

- 更新寄存器 Update register

PC更新 PC Update

- PC加2 Increment PC by 2

每个阶段的计算：算/逻操作
Stage Computation: Arith/Log. Ops

	OPq rA, rB	
取指 Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	读指令字节 Read instruction byte 读寄存器字节 Read register byte 计算下一个PC Compute next PC
译码 Decode	valA ← R[rA] valB ← R[rB]	读操作数A Read operand A 读操作数B Read operand B
执行 Execute	valE ← valB OP valA Set CC	执行ALU操作 Perform ALU operation 设置条件码寄存器 Set condition code register
访存 Memory		
写回 Write back	R[rB] ← valE	写回结果 Write back result
PC更新 PC update	PC ← valP	更新PC Update PC

- 指令的执行公式化为一系列简单的步骤 Formulate instruction execution as sequence of simple steps
- 对所有的指令使用同样的通用形式 Use same general form for all instructions

执行传送类指令 Executing rmmovq

取指 Fetch

- 读10个字节 Read 10 bytes

译码 Decode

- 读操作数寄存器 Read operand registers

执行 Execute

- 计算有效地址 Compute effective address

访存 Memory

- 写入内存 Write to memory

写回 Write back

- 无操作 Do nothing

PC更新 PC Update

- PC增加10 Increment PC by 10

每个阶段的计算：传送类指令
Stage Computation: rmmovq

	rmmovq rA, D(rB)	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valC ← M_8[PC+2] valP ← PC+10	Read instruction byte Read register byte 读变址值 Read displacement D Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M_8[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

- 使用ALU进行地址计算 Use ALU for address computation

执行出栈指令 Executing popq

取指 Fetch

- 读2个字节 Read 2 bytes

译码 Decode

- 读栈指针 Read stack pointer

执行 Execute

- 栈指针增加8 Increment stack pointer by 8

访存 Memory

- 从老的栈指针读 Read from old stack pointer

写回 Write back

- 更新栈指针 Update stack pointer
- 写结果到寄存器 Write result to register

PC更新 PC Update

- PC增加2 Increment PC by 2

每个阶段的计算：出栈
Stage Computation: popq

	popq rA	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valP ← PC+2	Compute next PC
Decode	valA ← R[%rsp]	Read stack pointer
	valB ← R[%rsp]	Read stack pointer
Execute	valE ← valB + 8	Increment stack pointer
Memory	valM ← M_8[valA]	Read from stack
Write back	R[%rsp] ← valE	Update stack pointer
	R[rA] ← valM	Write back result
PC update	PC ← valP	Update PC

- 使用ALU增加栈指针 Use ALU to increment stack pointer
- 必须更新两个寄存器 Must update two registers
  - 弹出的值 Popped value
  - 新的栈指针 New stack pointer

执行条件传送 Executing Conditional Moves

取指 Fetch

- 读2个字节 Read 2 bytes

译码 Decode

- 读操作数寄存器 Read operand registers

执行 Execute

- 如果没有设置条件码，则将目的寄存器设置为0xF If !cnd, then set destination register to 0xF

访存 Memory

- 无操作 Do nothing

写回 Write back

- 更新寄存器（或不更新） Update register (or not)

PC更新 PC Update

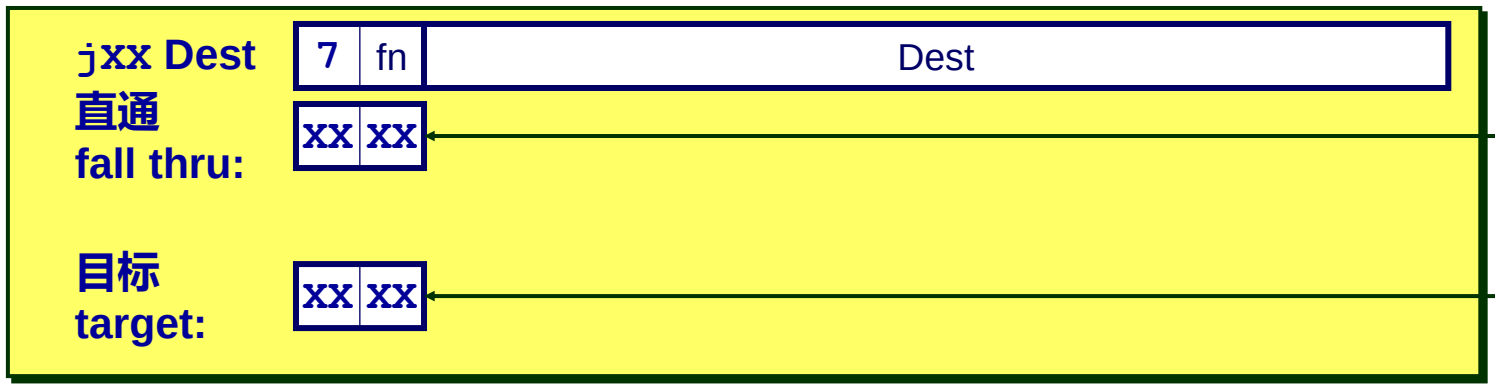
- PC增加2 Increment PC by 2

每个阶段的计算：条件传送
Stage Computation: Cond. Move

	cmovXX rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[rA] valB ← 0	Read operand A
Execute	valE ← valB + valA If ! Cond(CC,ifun) rB ← 0xF	Pass valA through ALU (Disable register update)
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

- 读寄存器rA并通过ALU Read register rA and pass through ALU
- 取消传送通过设置目的寄存器为0xF Cancel move by setting destination register to 0xF
  - 如果条件码与传送条件指明不需传送 If condition codes & move condition indicate no move

执行跳转类指令 Executing Jumps



取指 Fetch

- 读9个字节 Read 9 bytes
- PC增加9 Increment PC by 9

译码 Decode

- 无操作 Do nothing

执行 Execute

- 根据跳转条件和条件码确定是否进行分支转移 Determine whether to take branch based on jump condition and condition codes

访存 Memory

- 无操作 Do nothing

写回 Write back

- 无操作 Do nothing

PC更新 PC Update

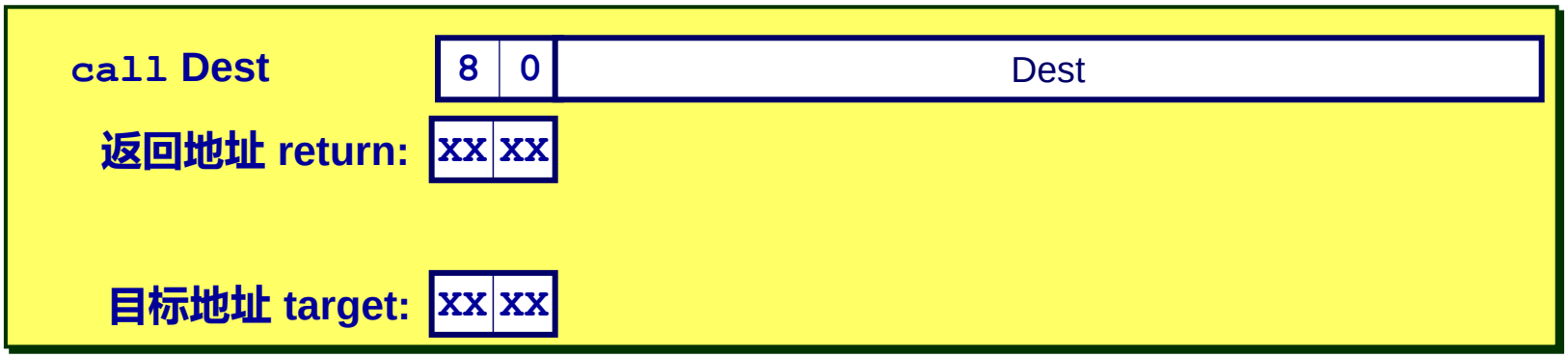
- 如果选择分支设置PC为目标地址，或者如果不选择分支则增加PC Set PC to Dest if branch taken or to incremented PC if not branch

每个阶段的计算：跳转
Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun ← M_1[PC] valC ← M_8[PC+1] valP ← PC+9	Read instruction byte Read destination address 下一条指令地址 Fall through address
Decode		
Execute	Cnd ← Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC ← Cnd ? valC : valP	Update PC

- 计算跳转和不跳转两个地址 Compute both addresses
- 根据条件码的设置和分支条件选择一个地址 Choose based on setting of condition codes and branch condition

执行过程调用 Executing `call`



取指 Fetch

- 读9个字节 Read 9 bytes
- PC增加9 Increment PC by 9

译码 Decode

- 读栈指针 Read stack pointer

执行 Execute

- 栈指针减8 Decrement stack pointer by 8

访存 Memory

- 将增加后的PC值写到栈指针新值 Write incremented PC to new value of stack pointer

写回 Write back

- 更新栈指针 Update stack pointer

PC更新 PC Update

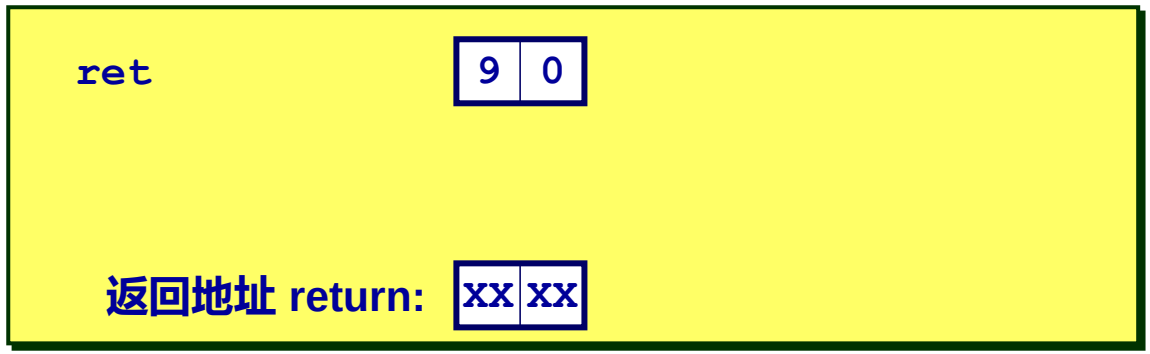
- 设置PC为目标地址 Set PC to Dest

每个阶段的计算：过程调用
Stage Computation: `call`

	`call` Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_8[PC+1]	Read destination address
	valP ← PC+9	Compute return point
Decode	valB ← R[`%rsp`]	Read stack pointer
Execute	valE ← valB + –8	Decrement stack pointer
Memory	M_8[valE] ← valP	Write return value on stack
Write back	R[`%rsp`] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- 使用ALU减栈指针 Use ALU to decrement stack pointer
- 存储增加后的PC Store incremented PC

执行过程返回 Executing ret



取指 Fetch

- 读1个字节 Read 1 byte

译码 Decode

- 读栈指针 Read stack pointer

执行 Execute

- 栈指针增加8 Increment stack pointer by 8

访存 Memory

- 从老的栈指针读返回地址 Read return address from old stack pointer

写回 Write back

- 更新栈指针 Update stack pointer

PC更新 PC Update

- 设置PC为返回地址 Set PC to return address

每个阶段的计算：过程返回
Stage Computation: `ret`

	`ret`	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
Decode	valA ← R[`%rsp`] valB ← R[`%rsp`]	Read operand stack pointer Read operand stack pointer
Execute	valE ← valB + 8	Increment stack pointer
Memory	valM ← M_8[valA]	Read return address
Write back	R[`%rsp`] ← valE	Update stack pointer
PC update	PC ← valM	Set PC to return address

- 使用ALU增加栈指针 Use ALU to increment stack pointer
- 从内存读返回地址 Read return address from memory

计算步骤 Computation Steps

		OPq rA, rB
取指 Fetch	icode,ifun	icode:ifun ← M_1[PC]
	rA,rB	rA:rB ← M_1[PC+1]
	valC	
	valP	valP ← PC+2
译码 Decode	valA, srcA	valA ← R[rA]
	valB, srcB	valB ← R[rB]
执行 Execute	valE	valE ← valB OP valA
	Cond code	Set CC
内存 Memory	valM	
写回 Write back	dstE	R[rB] ← valE
	dstM	
更新 PC update	PC	PC ← valP

读指令字节 Read instruction byte
读寄存器字节 Read register byte
读常量字 [Read constant word]
计算下一个PC Compute next PC
读操作数A Read operand A
读操作数B Read operand B
执行ALU操作 Perform ALU operation
设置/使用条件码寄存器Set/use cond. code reg
内存读/写[Memory read/write]
写回ALU结果 Write back ALU result
写回内存结果 [Write back memory result]
更新PC Update PC

- 所有指令遵循同样的通用模式 All instructions follow same general pattern
- 差别在于每步计算什么 Differ in what gets computed on each step

计算步骤 Computation Steps

		`call` Dest
取指 Fetch	icode,ifun	icode:ifun ← M_1[PC]
	rA,rB	
	valC	valC ← M_8[PC+1]
	valP	valP ← PC+9
译码 Decode	valA, srcA	
	valB, srcB	valB ← R[`%rsp`]
执行 Execute	valE	valE ← valB + –8
	Cond code	
内存 Memory	valM	M_8[valE] ← valP
写回 Write back	dstE	R[`%rsp`] ← valE
	dstM	
更新 PC update	PC	PC ← valC

读指令字节 Read instruction byte

读寄存器字节 [Read register byte]

读常量字 Read constant word

计算下一个PC Compute next PC

读操作数A [Read operand A]

读操作数B Read operand B

执行ALU操作 Perform ALU operation

设置/使用条件码寄存器 [Set /use cond. code reg]

内存读/写 Memory read/write

写回ALU结果 Write back ALU result

写回内存结果 [Write back memory result]

更新PC Update PC

- 所有指令遵循同样的通用模式 All instructions follow same general pattern
- 差别在于每步计算什么 Differ in what gets computed on each step

计算的值 Computed Values

取指 Fetch

- **icode** 指令代码 Instruction code
- **ifun** 指令功能 Instruction function
- **rA** 指令寄存器A Instr. Register A
- **rB** 指令寄存器B Instr. Register B
- **valC** 指令常量 Instruction constant
- **valP** 增加后的PC Incremented PC

译码 Decode

- **srcA** 寄存器ID Register ID A
- **srcB** 寄存器ID Register ID B
- **dstE** 目的寄存器 Destination Register E
- **dstM** 目的寄存器 Destination Register M
- **valA** 寄存器A的值 Register value A
- **valB** 寄存器B的值 Register value B

执行 Execute

- **valE** ALU结果 ALU result
- **Cnd** 分支/传送标志 Branch/move flag

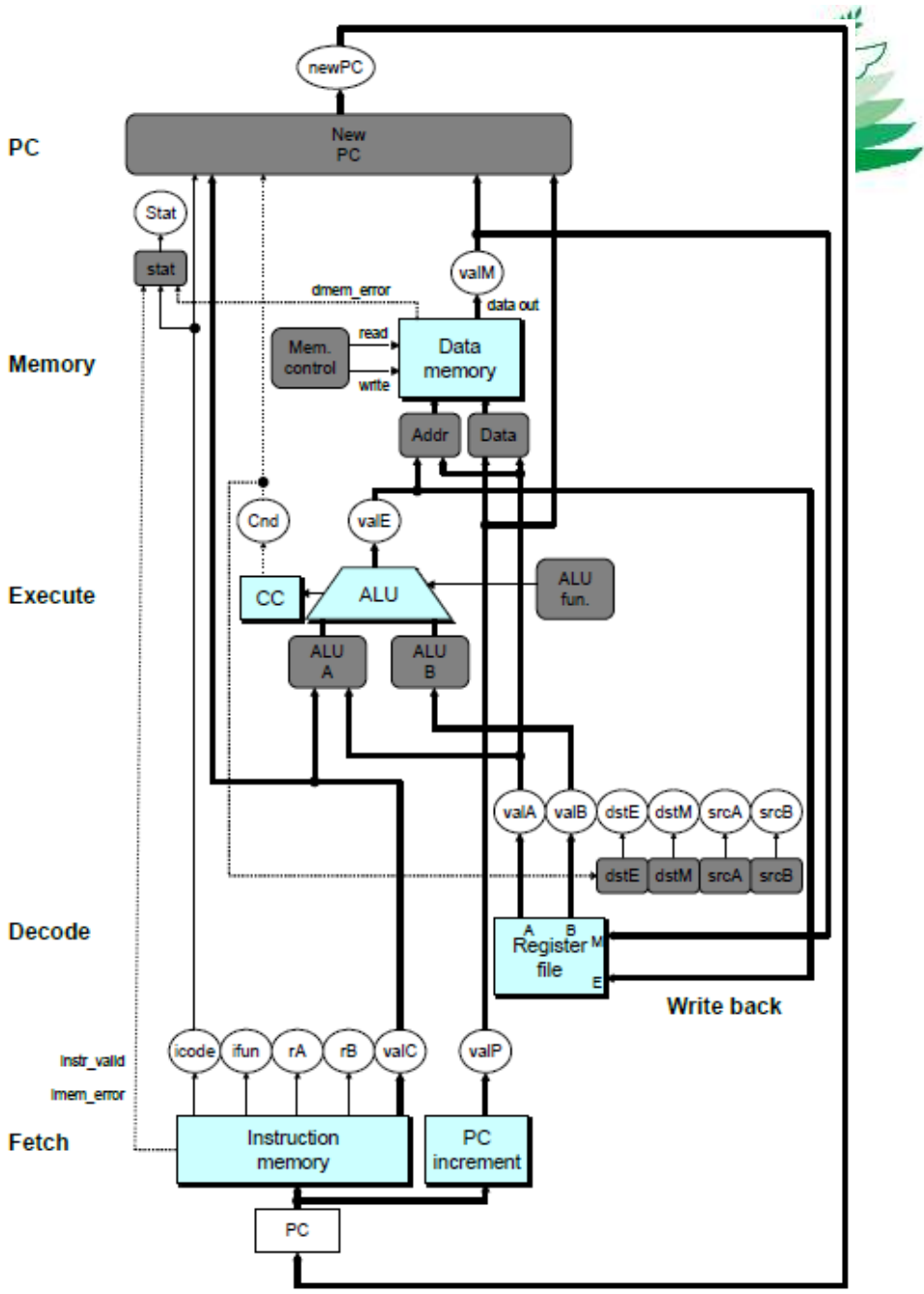
内存 Memory

- **valM** 内存的值 Value from memory

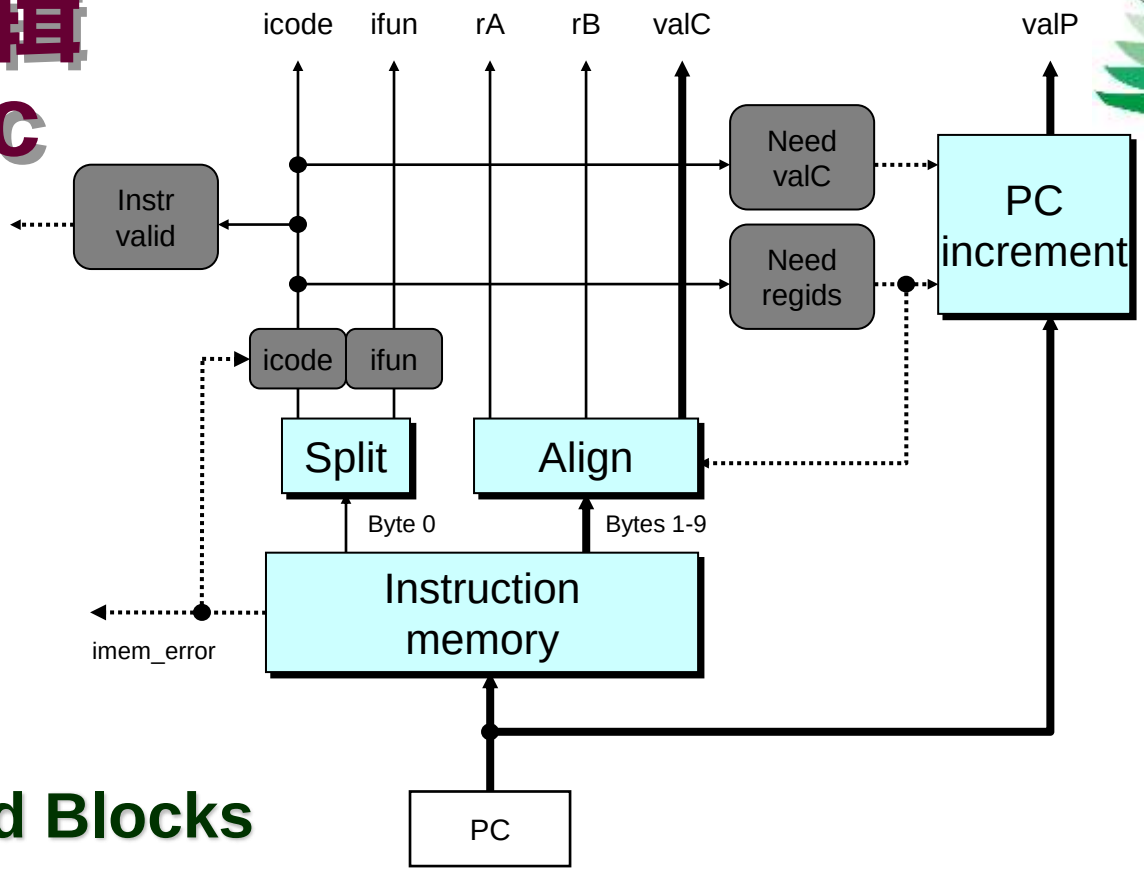
顺序处理器硬件 SEQ Hardware

关键 Key

- 蓝框：预设计的硬件块 Blue boxes: predesigned hardware blocks
  - 例如内存、ALU E.g., memories, ALU
- 灰框：控制逻辑 Gray boxes: control logic
  - HCL中描述 Describe in HCL
- 白椭圆框：信号标签 White ovals: labels for signals
- 粗线：64位字的值 Thick lines: 64-bit word values
- 细线：4-8位值 Thin lines: 4-8 bit values
- 虚线：1位值 Dotted lines: 1-bit values



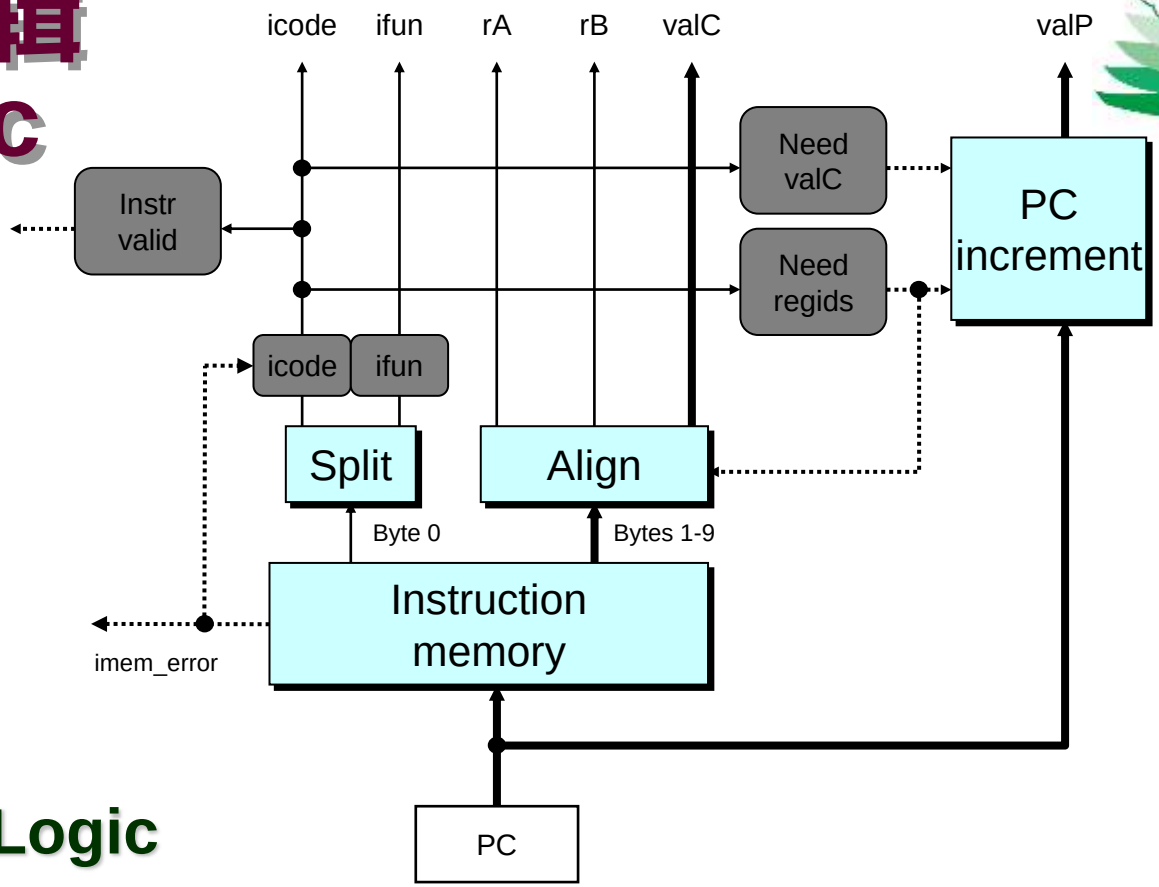
取指阶段逻辑 Fetch Logic



预定义块 Predefined Blocks

- PC：包含PC的寄存器 PC: Register containing PC
- 指令内存：读10个字节 Instruction memory: Read 10 bytes (PC to PC+9)
  - 不合法地址给出信号指示 Signal invalid address
- 分离器：指令字节分成icode和ifun两部分 Split: Divide instruction byte into icode and ifun
- 对齐：得到rA、rB和valC字段 Align: Get fields for rA, rB, and valC

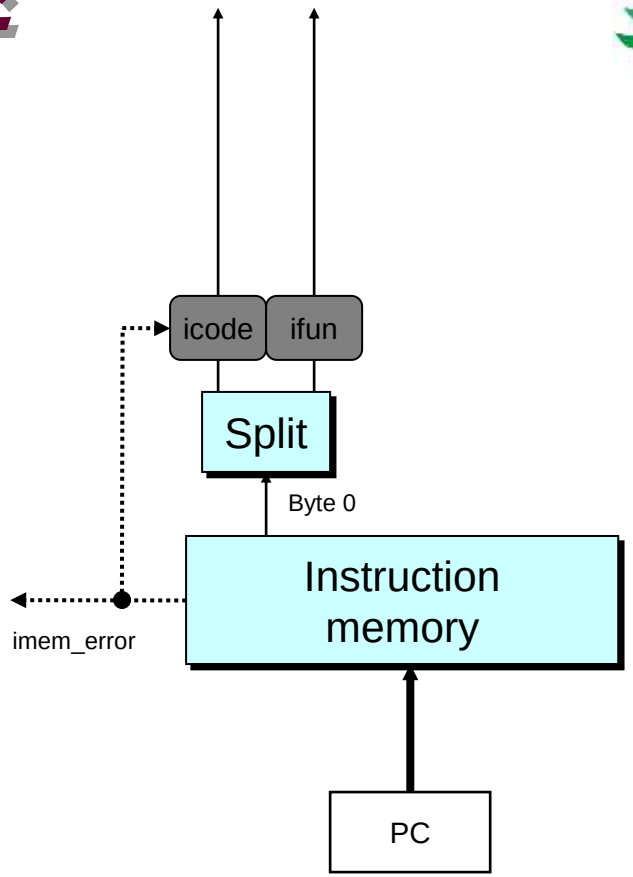
取指阶段逻辑 Fetch Logic



控制逻辑 Control Logic

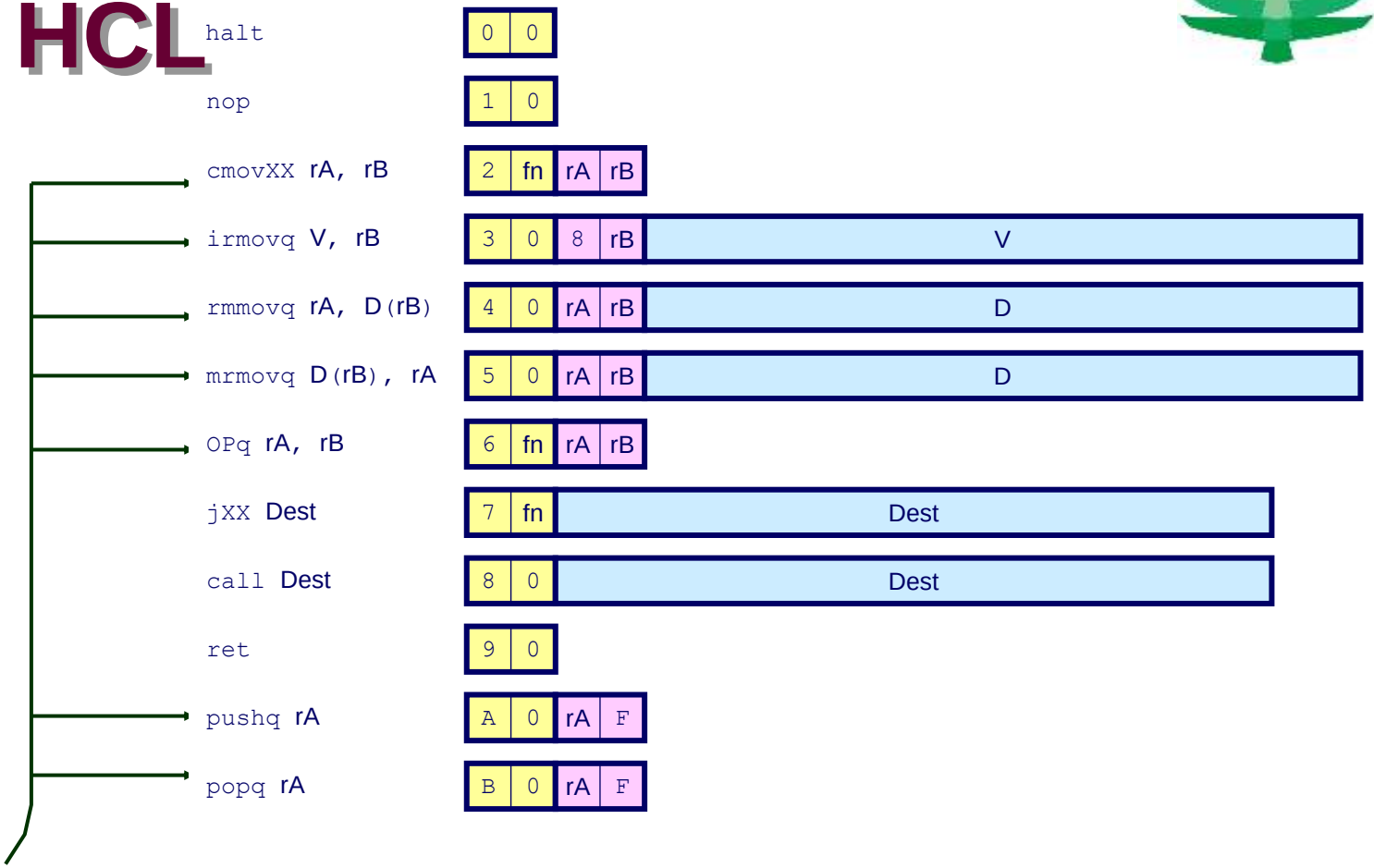
- Instr. Valid: 这条指令是否合法? Is this instruction valid?
- icode, ifun: 如果地址不合法，产生空指令 Generate no-op if invalid address
- Need regids: 这条指令有寄存器字节吗? Does this instruction have a register byte?
- Need valC: 这条指令有常量字吗? Does this instruction have a constant word?

取指控制逻辑HCL描述
Fetch Control Logic in HCL




```
# Determine instruction code
int icode = [
        imem_error: INOP;
        1: imem_icode;
];

# Determine instruction function
int ifun = [
        imem_error: FNONE;
        1: imem_ifun;
];
```

取指控制逻辑HCL描述 Fetch Control Logic in HCL

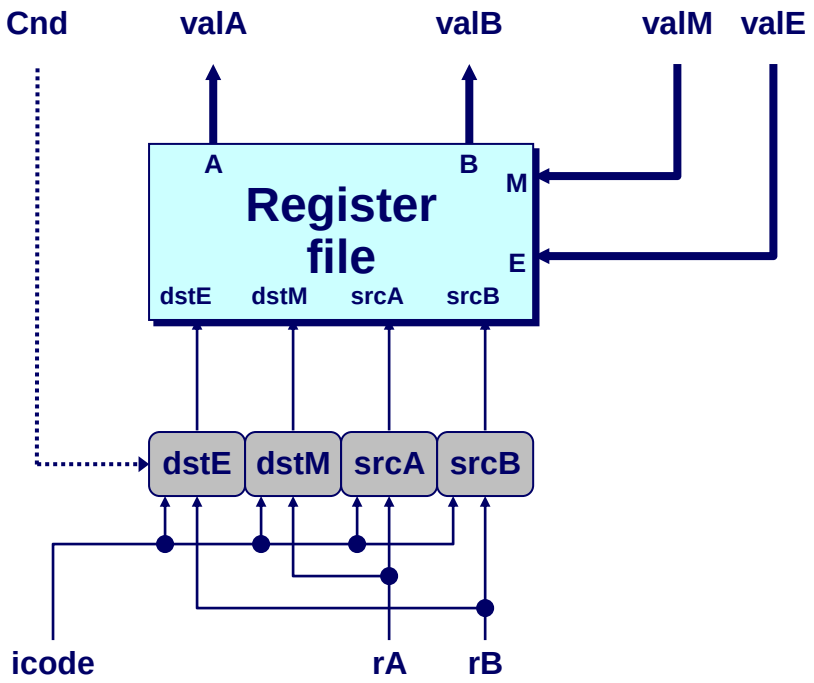



```
bool need_regids =
        icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                   IIRMOVQ, IRMMOVQ, IMRMOVQ };

bool instr_valid = icode in
        { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
          IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };
```

译码阶段逻辑 Decode Logic

寄存器文件（堆）Register File

- 读端口A和B Read ports A, B
- 写端口E和M Write ports E, M
- 地址是寄存器ID或15（0xF）（不访问）Addresses are register IDs or 15 (0xF) (no access)



控制逻辑 Control Logic

- srcA, srcB: 读端口地址 read port addresses
- dstE, dstM: 写端口地址 write port addresses

信号 Signals

- Cnd: 指明是否执行条件传送 Indicate whether or not to perform conditional move
  - 执行阶段计算 Computed in Execute stage

端口A的来源
A Source

	OPq rA, rB	
Decode	valA ← R[rA]	读操作数A Read operand A
	cmovXX rA, rB	
Decode	valA ← R[rA]	读操作数A Read operand A
	`rmmovq` rA, D(rB)	
Decode	valA ← R[rA]	读操作数A Read operand A
	`popq` rA	
Decode	valA ← R[`%rsp`]	读栈指针 Read stack pointer
	jXX Dest	
Decode		无操作数 No operand
	`call` Dest	
Decode		无操作数 No operand
	`ret`	
Decode	valA ← R[`%rsp`]	读栈指针 Read stack pointer

```
int srcA = [
        icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
        icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];
```

端口E的目的地址 E Destination

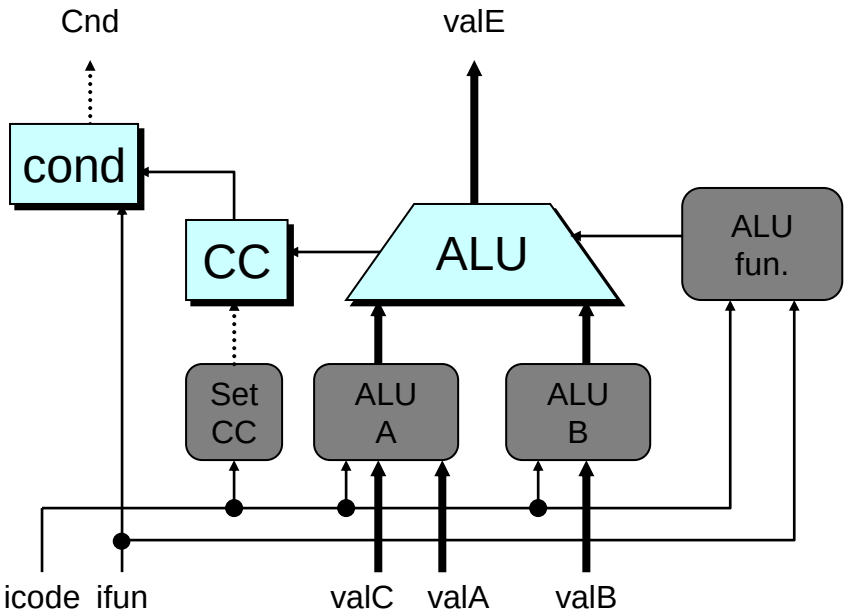
	OPq rA, rB	
Write-back	R[rB] ← valE	写回结果 Write back result
	cmovXX rA, rB	
Write-back	R[rB] ← valE	条件写回结果 Conditionally write back result
	`rmmovq` rA, D(rB)	
Write-back		无 None
	`popq` rA	
Write-back	R[`%rsp`] ← valE	更新栈指针 Update stack pointer
	jXX Dest	
Write-back		无 None
	`call` Dest	
Write-back	R[`%rsp`] ← valE	更新栈指针 Update stack pointer
	`ret`	
Write-back	R[`%rsp`] ← valE	更新栈指针 Update stack pointer

```
int dstE = [
        icode in { IRRMOVQ } && Cnd : rB;
        icode in { IIRMOVQ, IOPQ} : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];
```

执行阶段逻辑 Execute Logic

单元 Units

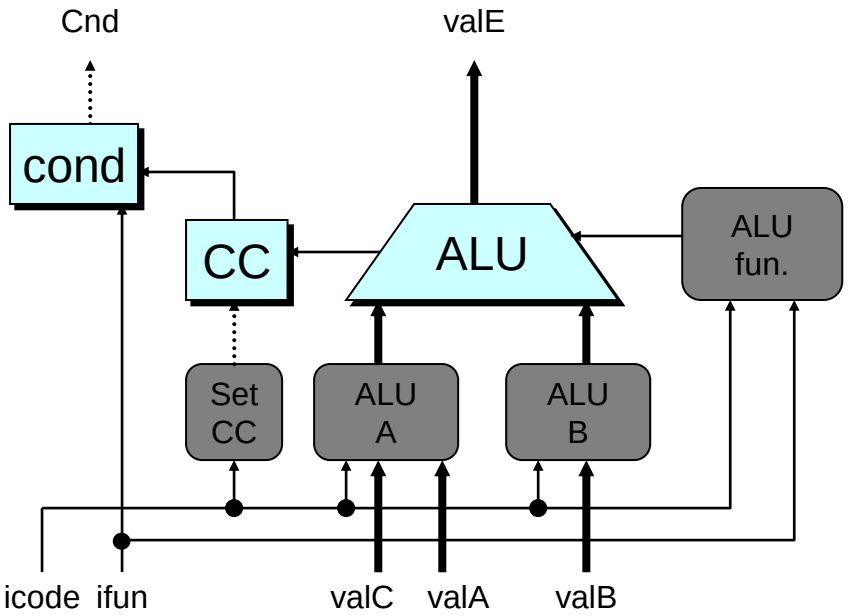
- 算术逻辑单元 ALU
  - 实现4个需要的功能 Implements 4 required functions
  - 产生条件码的值 Generates condition code values
- 条件码 CC
  - 有3个条件代码位的寄存器 Register with 3 condition code bits
- cond
  - 计算条件跳转/传送标志 Computes conditional jump/move flag



执行阶段逻辑 Execute Logic

控制逻辑 Control Logic

- Set CC: 装载条件码寄存器吗? Should condition code register be loaded?
- ALU A: ALU输入端A Input A to ALU
- ALU B: ALU输入端B Input B to ALU
- ALU fun: ALU的计算功能 What function should ALU compute?



ALU A 输入端 ALU A Input

	OPq rA, rB
Execute	valE ← valB OP valA

执行ALU操作

Perform ALU operation

	cmovXX rA, rB
Execute	valE ← 0 + valA

传递valA直通ALU

Pass valA through ALU

	rmmovq rA, D(rB)
Execute	valE ← valB + valC

计算有效地址

Compute effective address

	popq rA
Execute	valE ← valB + 8

增加栈指针

Increment stack pointer

	jXX Dest
Execute	

无操作 No operation

	call Dest
Execute	valE ← valB + –8

减少栈指针

Decrement stack pointer

	ret
Execute	valE ← valB + 8

增加栈指针

Increment stack pointer

```
int aluA = [
        icode in { IRRMOVQ, IOPQ } : valA;
        icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
        icode in { ICALL, IPUSHQ } : -8;
        icode in { IRET, IPOPQ } : 8;
        # Other instructions don't need ALU
];
```

ALU 运算 ALU Oper-ation

	OPq rA, rB
Execute	valE ← valB OP valA

执行ALU操作

Perform ALU operation

	cmovXX rA, rB
Execute	valE ← 0 + valA

传递valA直通ALU

Pass valA through ALU

	`rmmovl` rA, D(rB)
Execute	valE ← valB + valC

计算有效地址

Compute effective address

	`popq` rA
Execute	valE ← valB + 8

增加栈指针

Increment stack pointer

	jXX Dest
Execute	

无操作 No operation

	`call` Dest
Execute	valE ← valB + –8

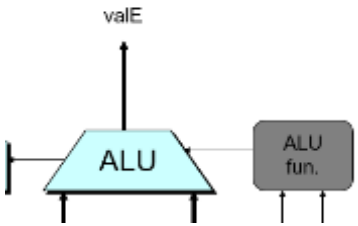
减少栈指针

Decrement stack pointer

	`ret`
Execute	valE ← valB + 8

增加栈指针

Increment stack pointer




```
int alufun = [
        icode == IOPQ : ifun;
        1 : ALUADD;
];
```

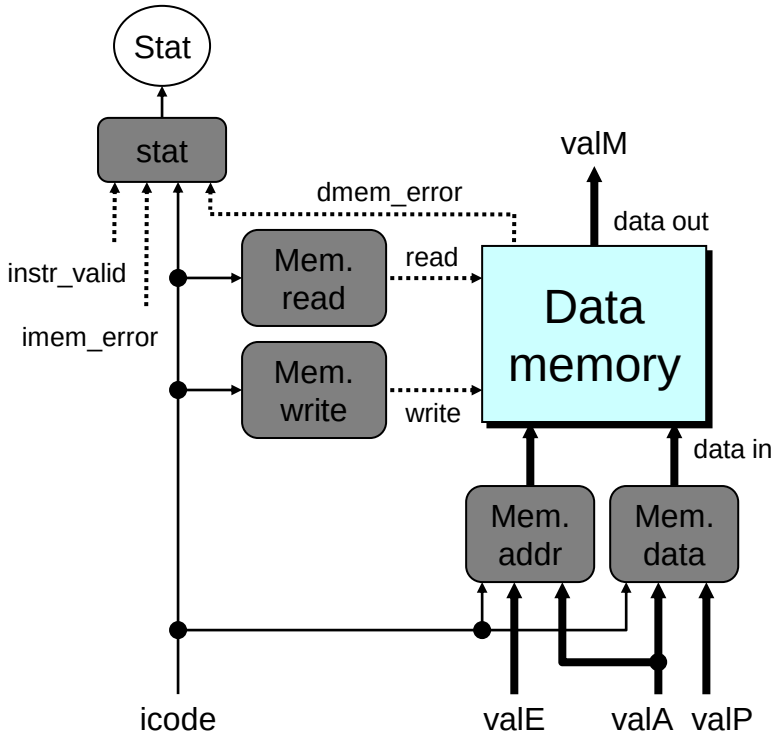
访存阶段逻辑 Memory Logic

访存 Memory

- 读或写内存字 Reads or writes memory word

控制逻辑 Control Logic

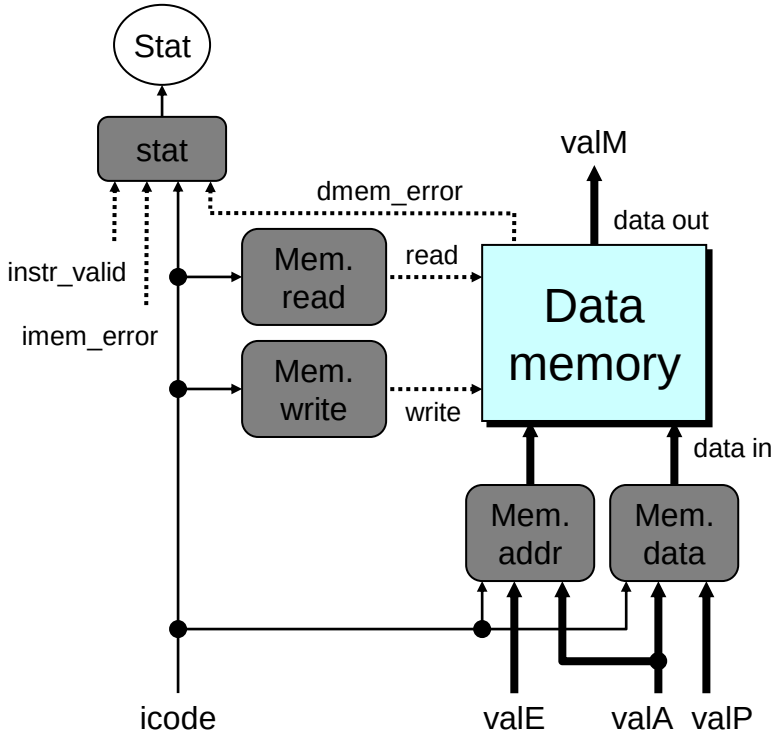
- stat: 指令状态 What is instruction status?
- Mem. read: 读字 should word be read?
- Mem. write: 写字 should word be written?
- Mem. addr.: 选择地址 Select address
- Mem. data.: 选择数据 Select data



指令状态 Instruction Status

控制逻辑 Control Logic

- stat: 指令状态 What is instruction status?




```
## Determine instruction status
int Stat = [
        imem_error || dmem_error : SADR;
        !instr_valid: SINS;
        icode == IHALT : SHLT;
        1 : SAOK;
];
```

访存地址 Memory Address

	OPq rA, rB	
Memory		无操作 No operation

	rmmovq rA, D(rB)	
Memory	M_8[valE] ← valA	写数据到内存 Write value to memory

	popq rA	
Memory	valM ← M_8[valA]	从栈读数据 Read from stack

	jXX Dest	
Memory		无操作 No operation

	call Dest	
Memory	M_8[valE] ← valP	写返回地址到栈 Write return value on stack

	ret	
Memory	valM ← M_8[valA]	读返回地址 Read return address

```
int mem_addr = [
        icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
        icode in { IPOPQ, IRET } : valA;
        # Other instructions don't need address
];
```

内存读 Memory Read

	OPq rA, rB	
Memory		无操作 No operation

	rmmovq rA, D(rB)	
Memory	M_8[valE] ← valA	写数据到内存 Write value to memory

	popq rA	
Memory	valM ← M_8[valA]	从栈读数据 Read from stack

	jXX Dest	
Memory		无操作 No operation

	call Dest	
Memory	M_8[valE] ← valP	写返回地址到栈 Write return value on stack

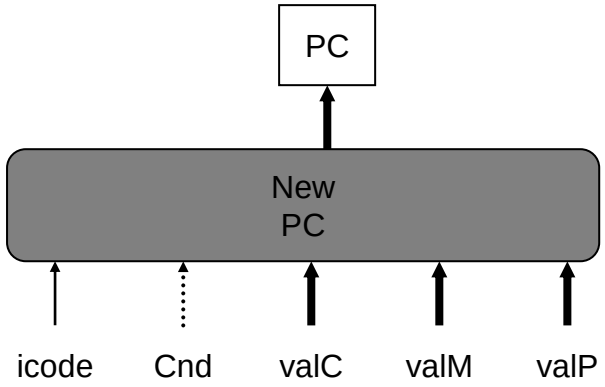
	ret	
Memory	valM ← M_8[valA]	读返回地址 Read return address

```
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET };
```

PC更新阶段逻辑 PC Update Logic

New PC

- 选择PC的下一个值 Select next value of PC



PC更新 PC Update

	OPq rA, rB
PC update	PC ← valP

更新PC Update PC

	`rmmovq` rA, D(rB)
PC update	PC ← valP

更新PC Update PC

	`popq` rA
PC update	PC ← valP

更新PC Update PC

	jXX Dest
PC update	PC ← Cnd ? valC : valP

更新PC Update PC

	`call` Dest
PC update	PC ← valC

设置PC为目的地址

Set PC to destination

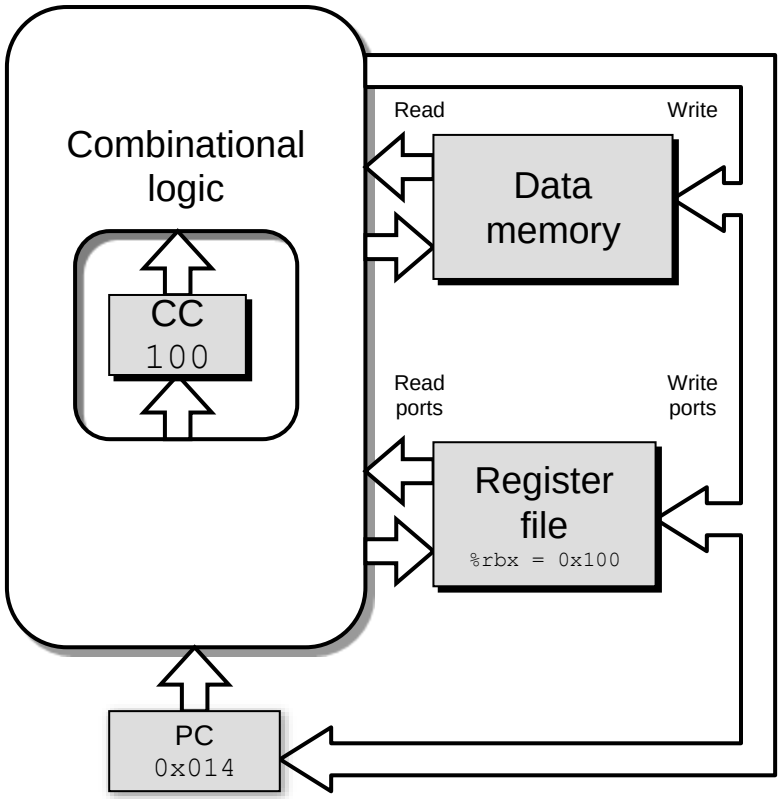
	`ret`
PC update	PC ← valM

设置PC为返回地址

Set PC to return address

```
int new_pc = [
        icode == ICALL : valC;
        icode == IJXX && Cnd : valC;
        icode == IRET : valM;
        1 : valP;
];
```

顺序处理器操作 SEQ Operation

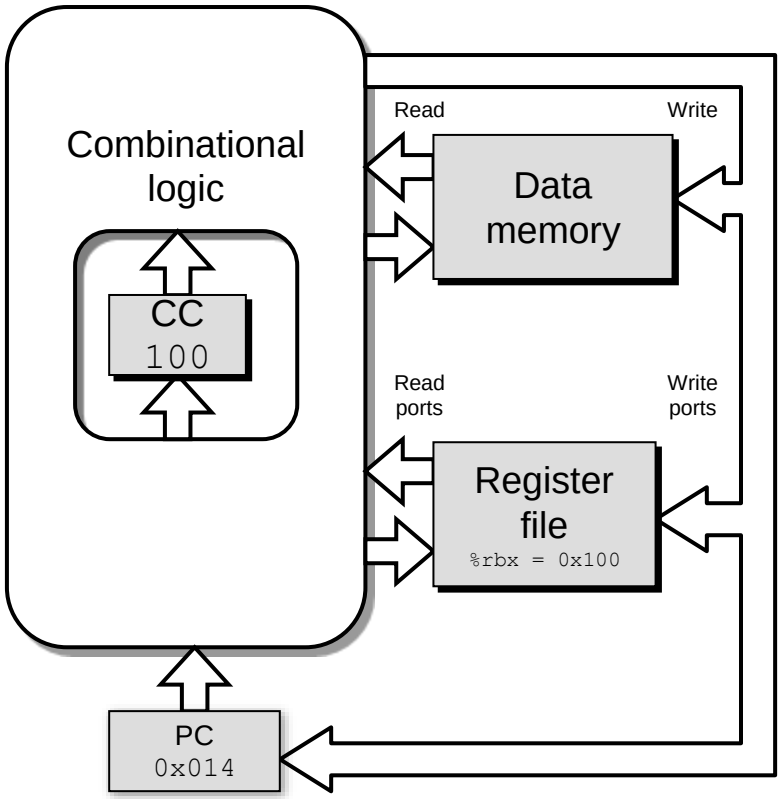


状态 State

- PC寄存器 PC register
- 条件码寄存器 Cond. Code register
- 数据内存 Data memory
- 寄存器文件（堆） Register file

所有更新在时钟上升沿 All updated as clock rises

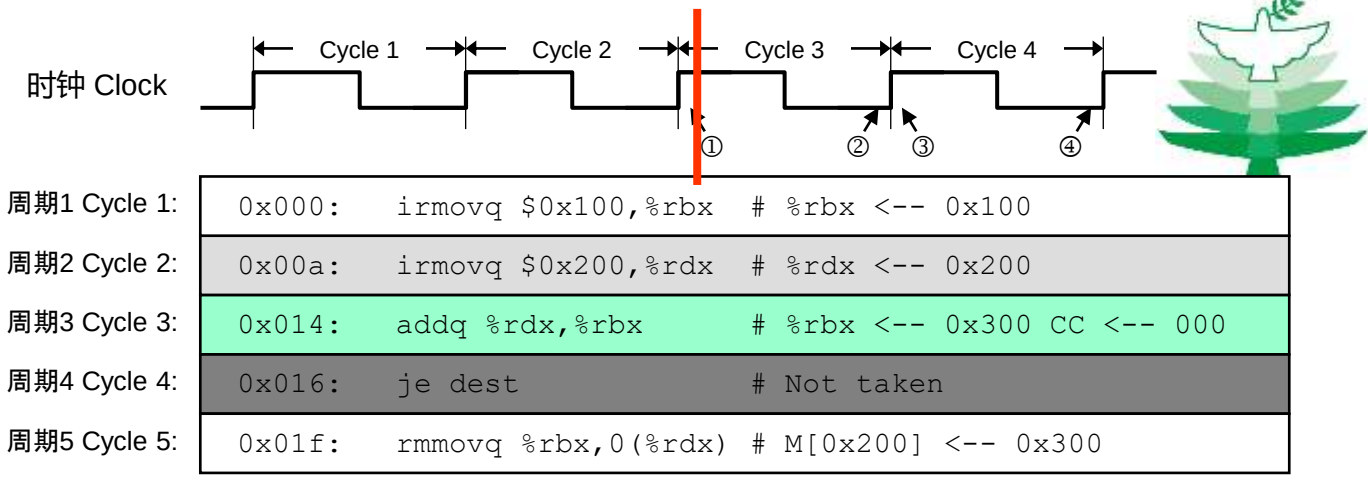
顺序处理器操作 SEQ Operation

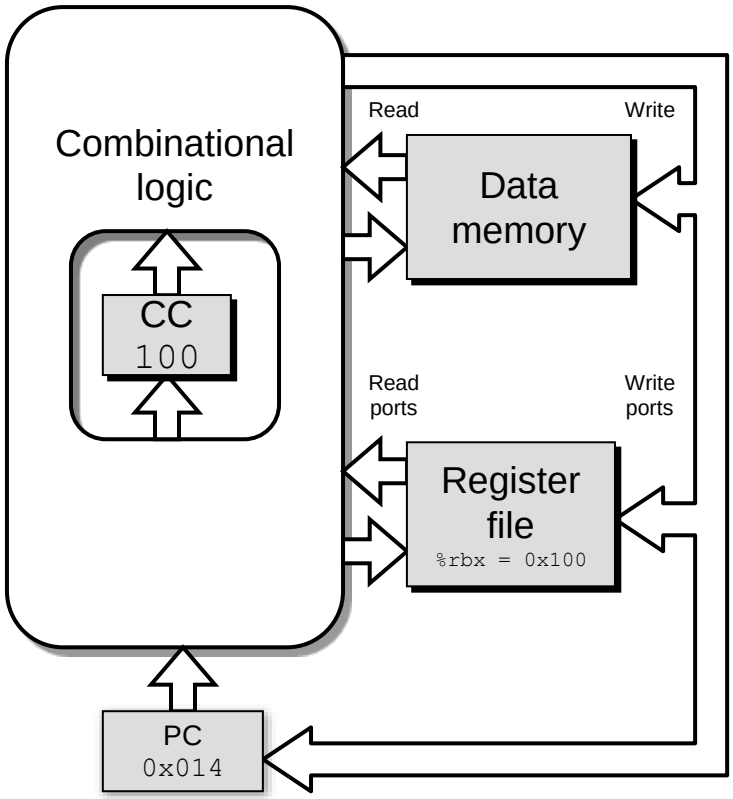


组合逻辑 Combinational Logic

- 算术/逻辑单元 ALU
- 控制逻辑 Control logic
- 内存读 Memory reads
  - 指令内存 Instruction memory
  - 寄存器文件（堆）Register file
  - 数据内存 Data memory

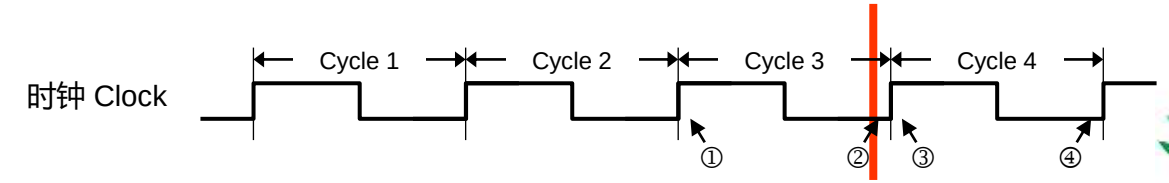
SEQ操作 SEQ Operation #2



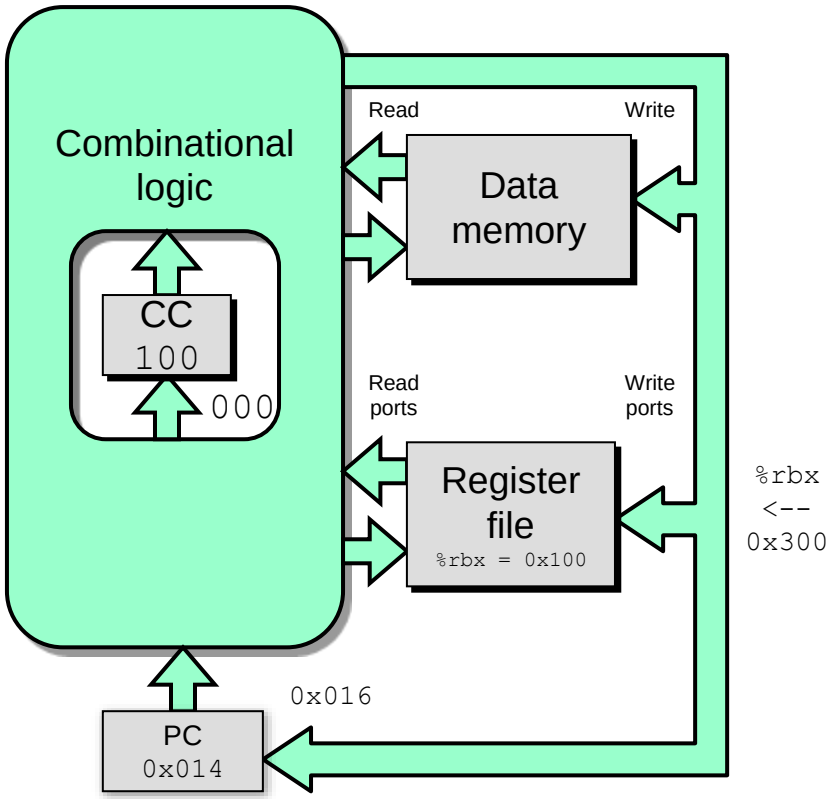


- 状态设置按照第二条irmovq指令 state set according to second `irmovq` instruction
- 组合逻辑开始反应状态改变 combinational logic starting to react to state changes

SEQ操作 SEQ Operation #3

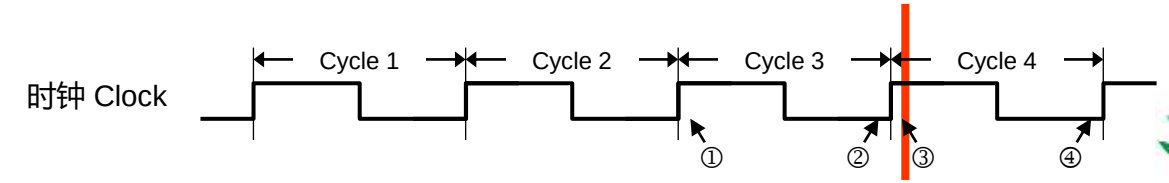


周期1 Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
周期2 Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
周期3 Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
周期4 Cycle 4:	0x016:	je dest	# Not taken
周期5 Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

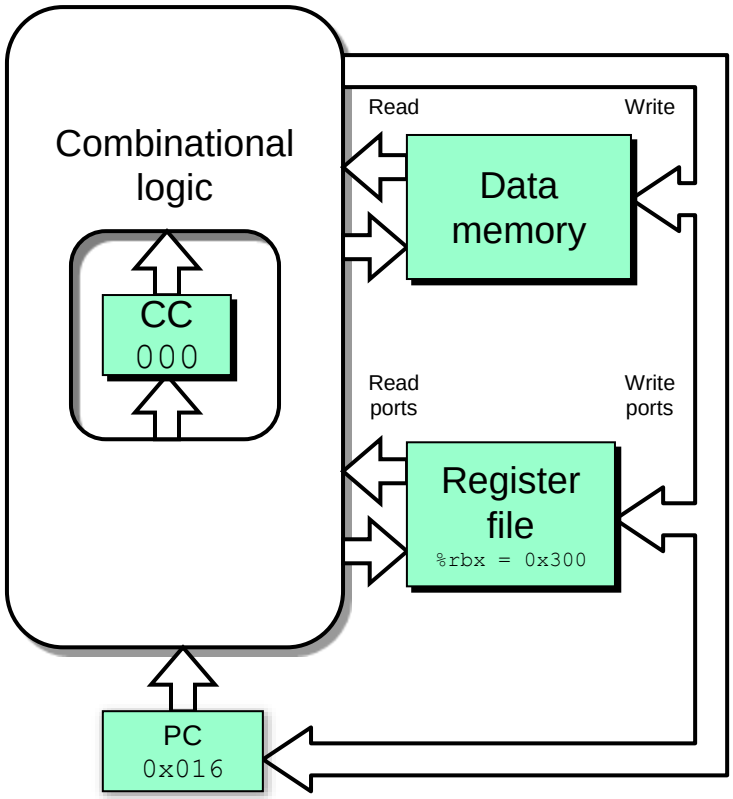


- 状态设置按照第二条irmovq指令state set according to second `irmovq` instruction
- 组合逻辑产生addq指令的结果 combinational logic generates results for `addq` instruction

SEQ操作 SEQ Operation #4

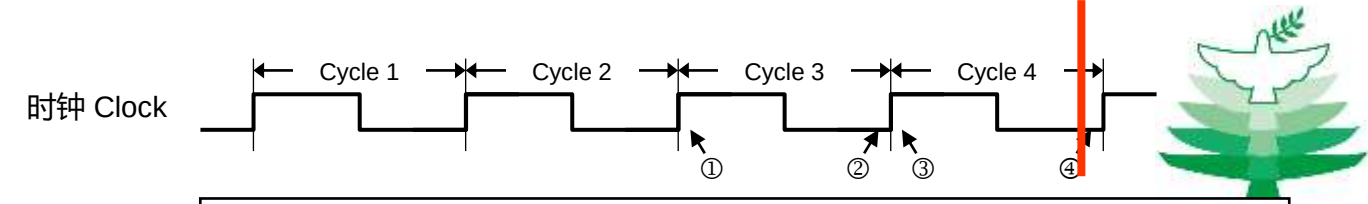


周期1 Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
周期2 Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
周期3 Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
周期4 Cycle 4:	0x016:	je dest	# Not taken
周期5 Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300

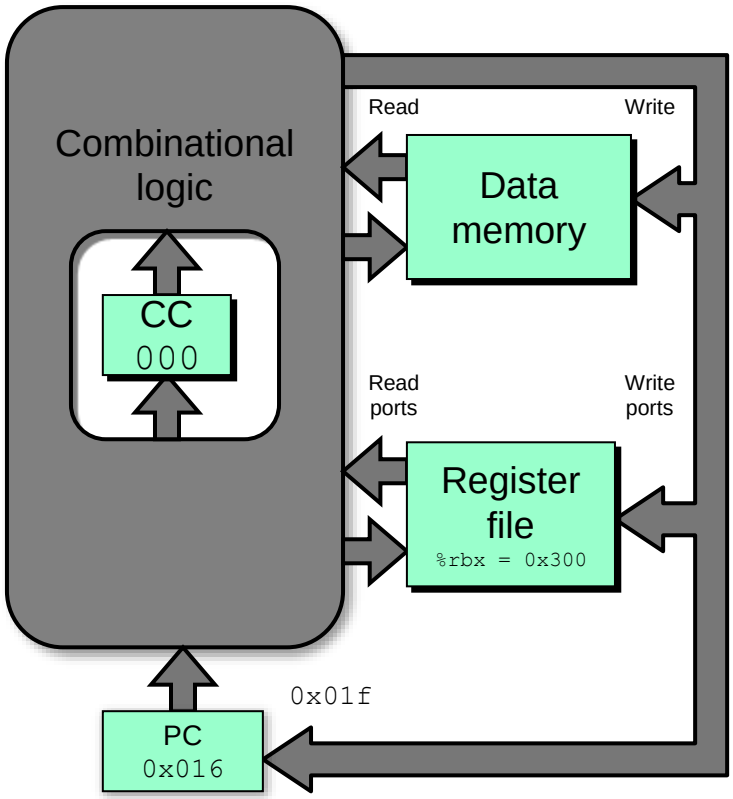


- 状态设置按照addq指令 state set according to addq instruction
- 组合逻辑开始反应状态改变 combinational logic starting to react to state changes

SEQ 操作 SEQ Operation #5



周期1 Cycle 1:	0x000:	irmovq $0x100,%rbx	# %rbx <-- 0x100
周期2 Cycle 2:	0x00a:	irmovq $0x200,%rdx	# %rdx <-- 0x200
周期3 Cycle 3:	0x014:	addq %rdx,%rbx	# %rbx <-- 0x300 CC <-- 000
周期4 Cycle 4:	0x016:	je dest	# Not taken
周期5 Cycle 5:	0x01f:	rmmovq %rbx,0(%rdx)	# M[0x200] <-- 0x300



- 状态设置按照addq指令 state set according to `addq` instruction
- 组合逻辑产生je指令的结果 combinational logic generates results for `je` instruction

顺序处理器小结 SEQ Summary

实现 Implementation

- 表达每条指令为一系列简单的步骤 Express every instruction as series of simple steps
- 每个指令类型遵循同样通用流程 Follow same general flow for each instruction type
- 装配寄存器、内存和预设计的组合逻辑块 Assemble registers, memories, predesigned combinational blocks
- 用控制逻辑进行连接 Connect with control logic

限制 Limitations

- 太慢不实用 Too slow to be practical
- 一个周期内必须传播通过指令内存、寄存器文件（堆）、ALU和数据内存 In one cycle, must propagate through instruction memory, register file, ALU, and data memory
- 需要运行时钟非常慢 Would need to run clock very slowly
- 硬件单元仅在时钟周期的一部分时间内是活动的 Hardware units only active for fraction of clock cycle

CS:APP Chapter 4
Computer Architecture
Pipelined Implementation
Part I
流水线实现
第一部分

任课教师：
宿红毅　张艳　黎有琦　颜珂

原作者：
Randal E. **Bryant and** David R. O'Hallaron

Carnegie Mellon University

概述 Overview

流水线的一般原理 General Principles of Pipelining

- 目标 Goal
- 难点 Difficulties

创建一个具有流水线的Y86-64处理器 Creating a Pipelined Y86-64 Processor

- 重新安排顺序处理器SEQ Rearranging SEQ
- 插入流水线寄存器 Inserting pipeline registers
- 数据和控制冒险问题 Problems with data and control hazards

真实世界的流水线：洗车
Real-World Pipelines: Car Washes

顺序 Sequential

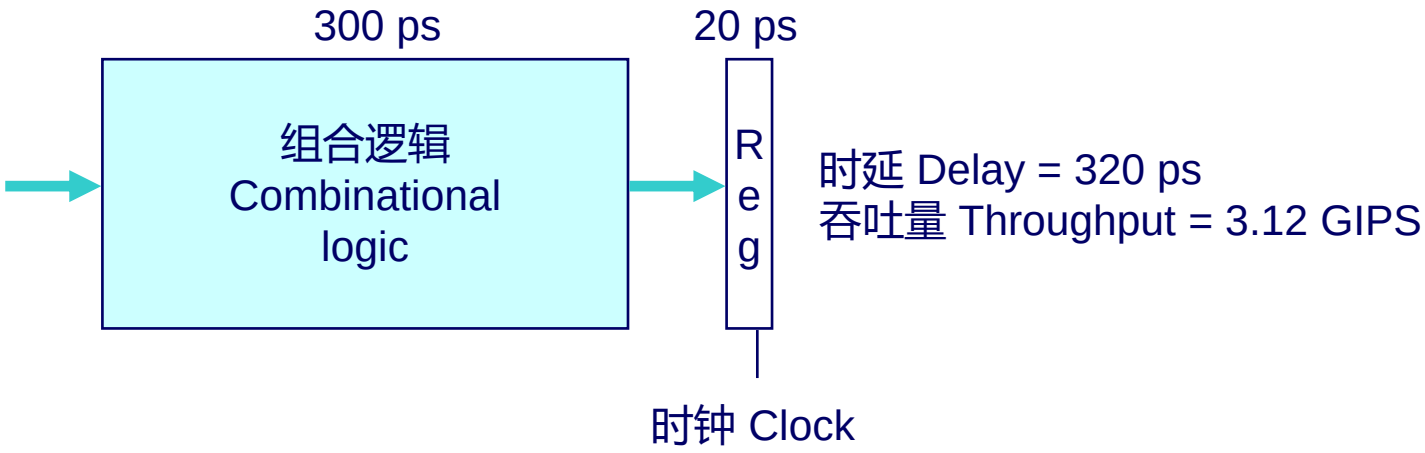
并行 Parallel

流水线 Pipelined

思想 Idea

- 将洗车过程分成若干独立的阶段 Divide process into independent stages
- 顺序移动目标通过各个阶段 Move objects through stages in sequence
- 在任何给定时间，在对多个目标进行处理 At any given times, multiple objects being processed

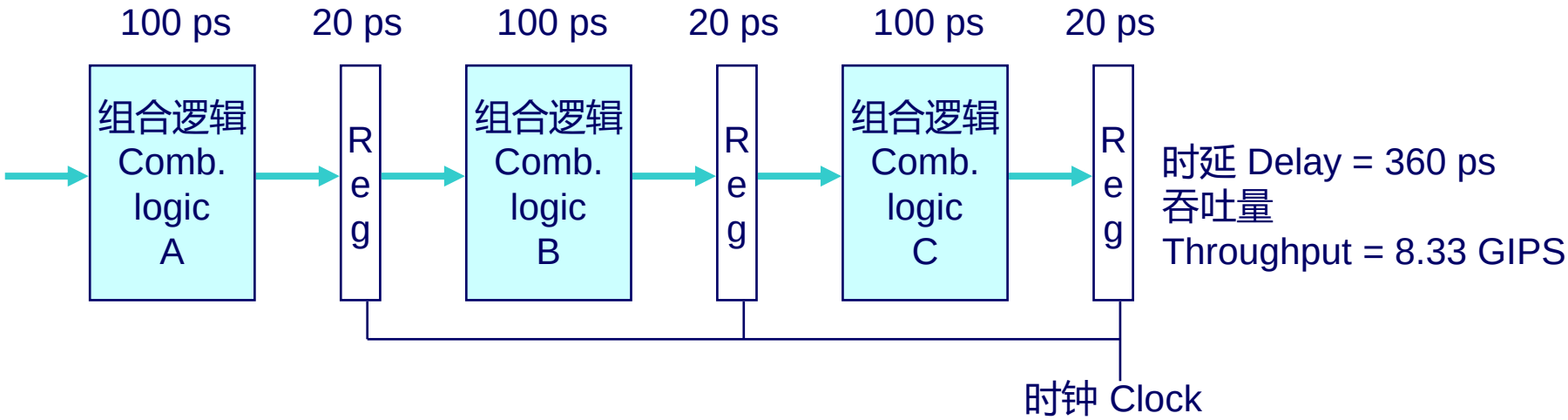
计算示例 Computational Example



系统 System

- 计算需要总计300ps Computation requires total of 300 picoseconds
- 另外20ps保存结果在寄存器中 Additional 20 picoseconds to save result in register
- 时钟周期必须至少320ps Must have clock cycle of at least 320 ps

3级流水线版本
3-Way Pipelined Version

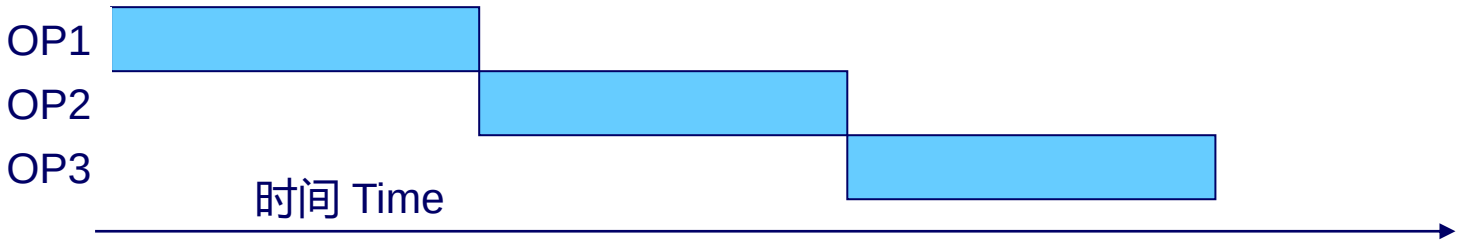


系统 System

- 将组合逻辑分成3块，每块需要100ps Divide combinational logic into 3 blocks of 100 ps each
- 只要上一个操作通过阶段A，就可以立即开始新的操作 Can begin new operation as soon as previous one passes through stage A.
  - 每隔120ps开始一个新操作 Begin new operation every 120 ps
- 总体时延增加 Overall latency increases
  - 从开始到结束需360ps 360 ps from start to finish

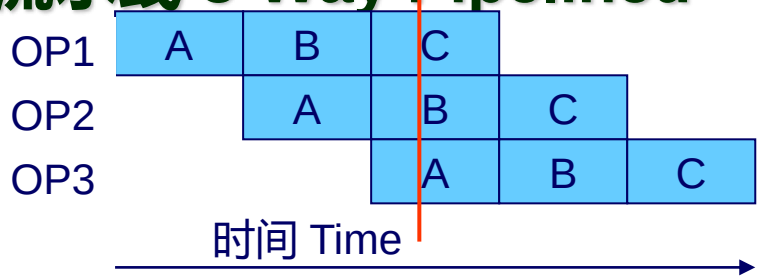
流水线图 Pipeline Diagrams

非流水线 Unpipelined



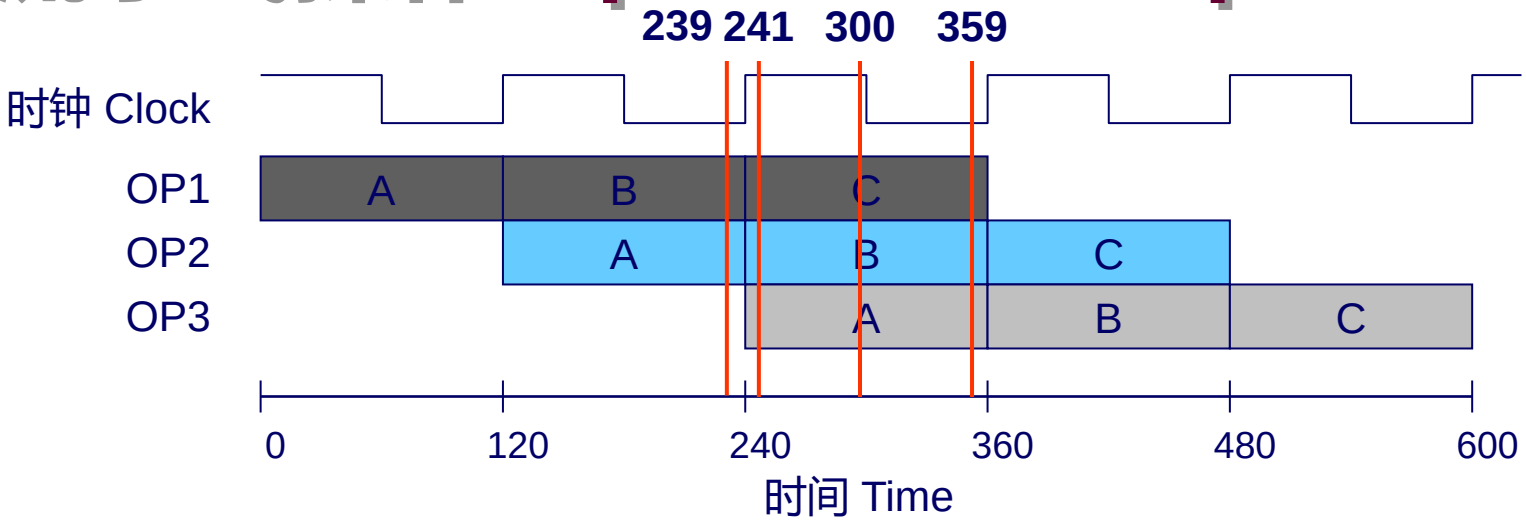
- 在上一个操作完成前不能开始新的操作 Cannot start new operation until previous one completes

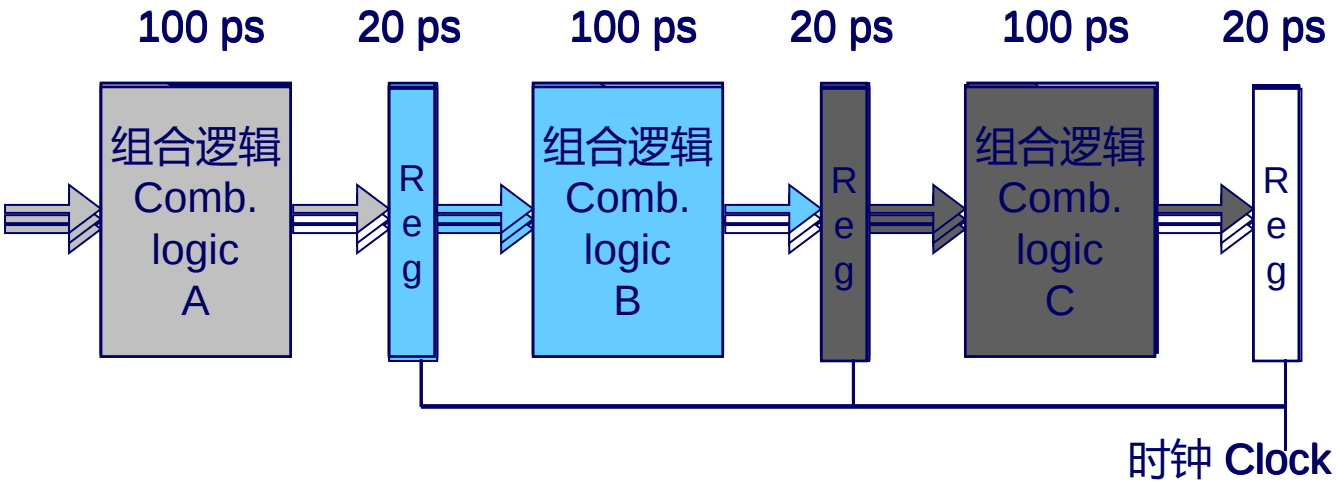
3级流水线 3-Way Pipelined



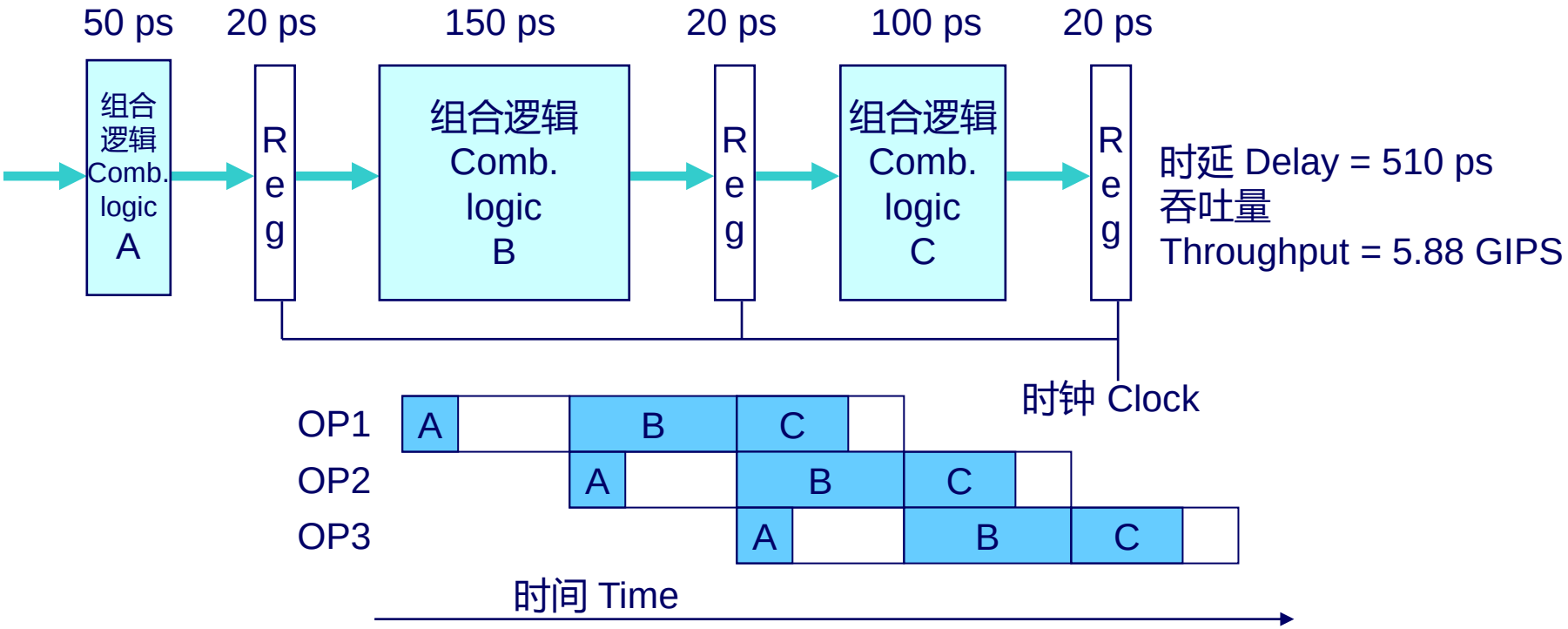
- 最多3个操作在同时处理 Up to 3 operations in process simultaneously

流水线操作 Operating a Pipeline



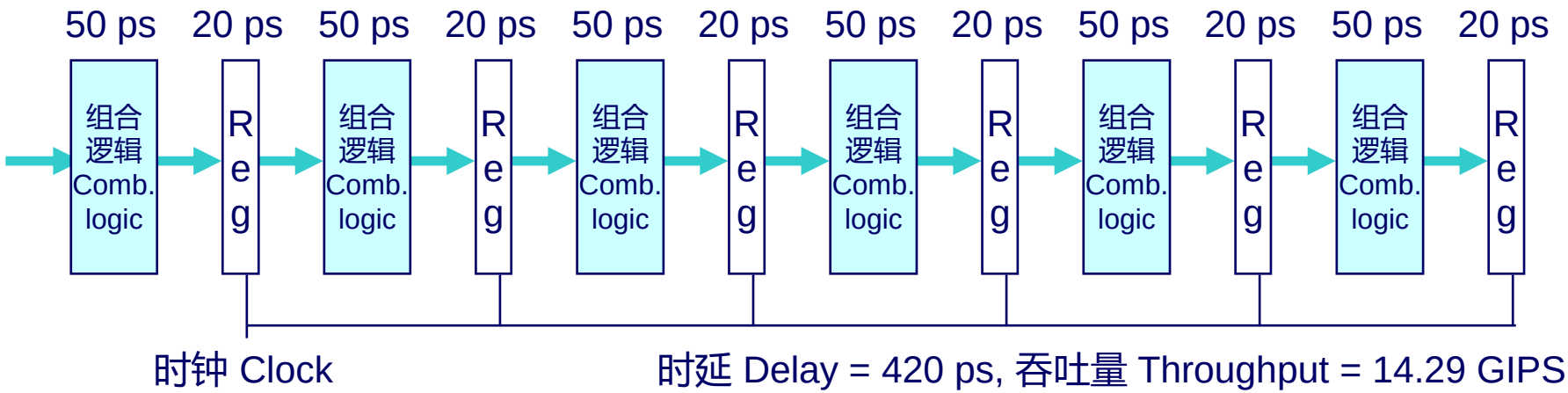


限制：非统一时延
Limitations: Non-uniform Delays



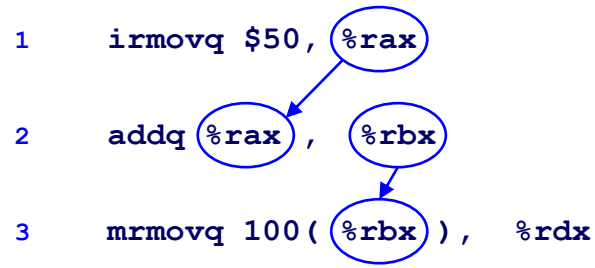
- 吞吐量受限于最慢的阶段 Throughput limited by slowest stage
- 其它阶段大部分时间都处于空闲状态 Other stages sit idle for much of the time
- 挑战在于把系统分成平衡的阶段 Challenging to partition system into balanced stages

限制：寄存器开销
Limitations: Register Overhead



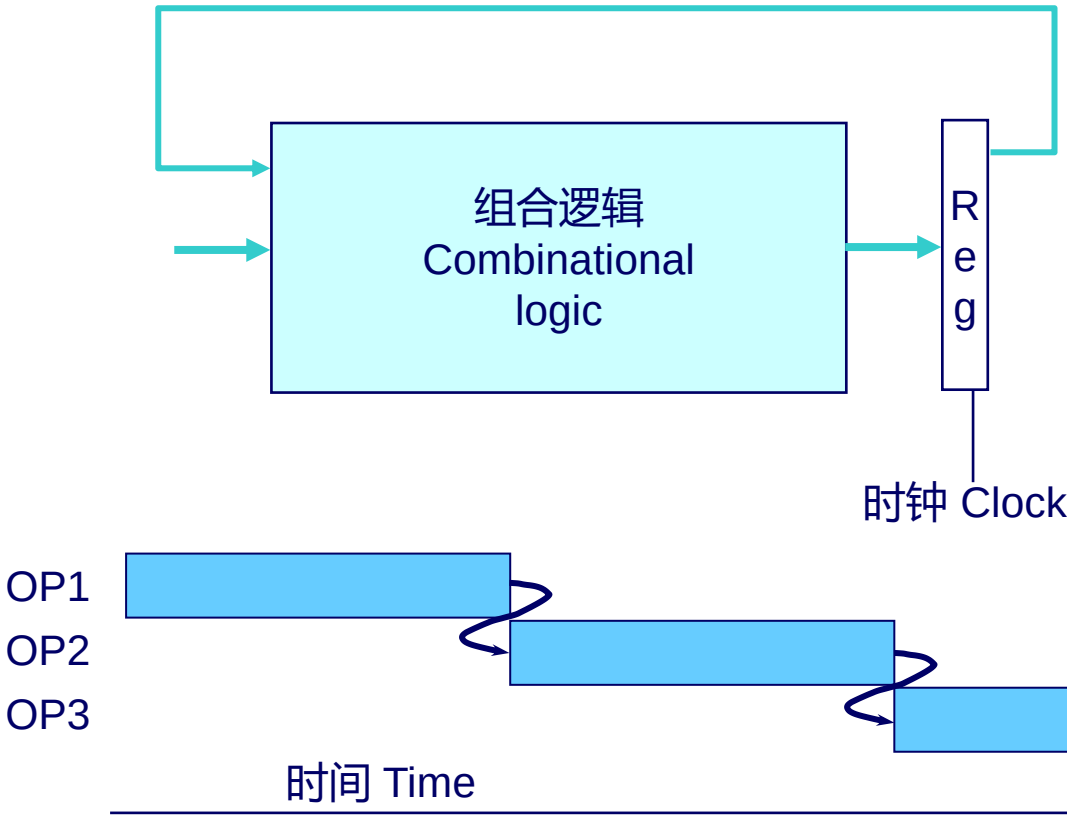
- **随着流水线深度加深，装载寄存器的开销变得越来越大 As try to deepen pipeline, overhead of loading registers becomes more significant**
- **时钟周期花费在装载寄存器的百分比：Percentage of clock cycle spent loading register:**
  - **1-stage pipeline: 6.25% 1阶段流水线：20/320**
  - **3-stage pipeline: 16.67% 3阶段流水线：60/360**
  - **6-stage pipeline: 28.57% 6阶段流水线：120/420**
- **现代处理器设计的高速度通过非常深度的流水线获得的 High speeds of modern processor designs obtained through very deep pipelining**

处理器中的数据相关
Data Dependencies in Processors



- 一条指令的结果用作另一条指令的操作数 Result from one instruction used as operand for another
  - 写后读（RAW）相关 Read-after-write (RAW) dependency
- 在实际程序中非常常见 Very common in actual programs
- 必须确保流水线能够正确处理这些情况 Must make sure our pipeline handles these properly
  - 得到正确的结果 Get correct results
  - 最小化对性能的影响 Minimize performance impact

数据相关 Data Dependencies



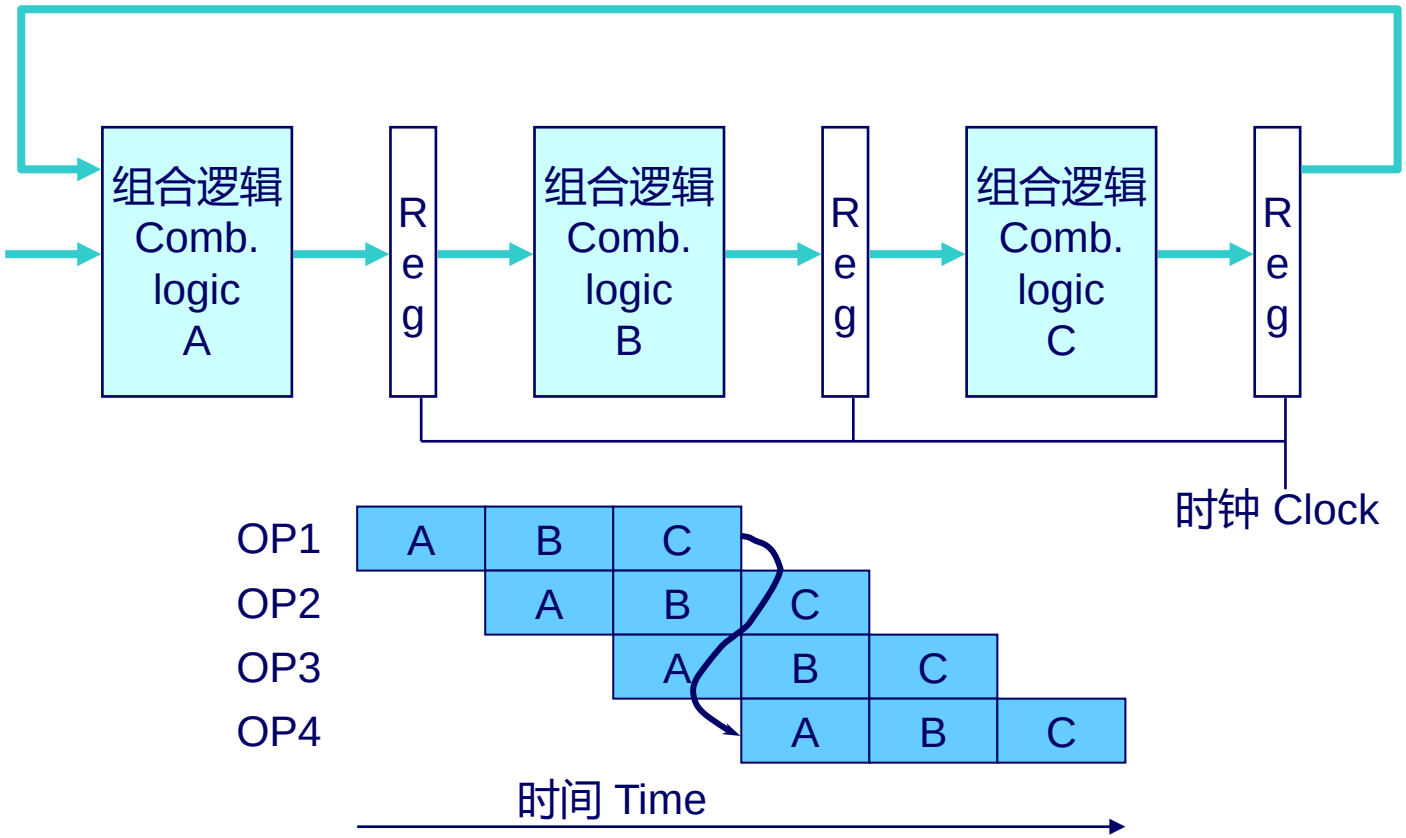
非流水线化（顺序）的处理器

系统 System

- 每个操作依赖于上一次操作的结果 Each operation depends on result from preceding one

数据冒险 Data Hazards

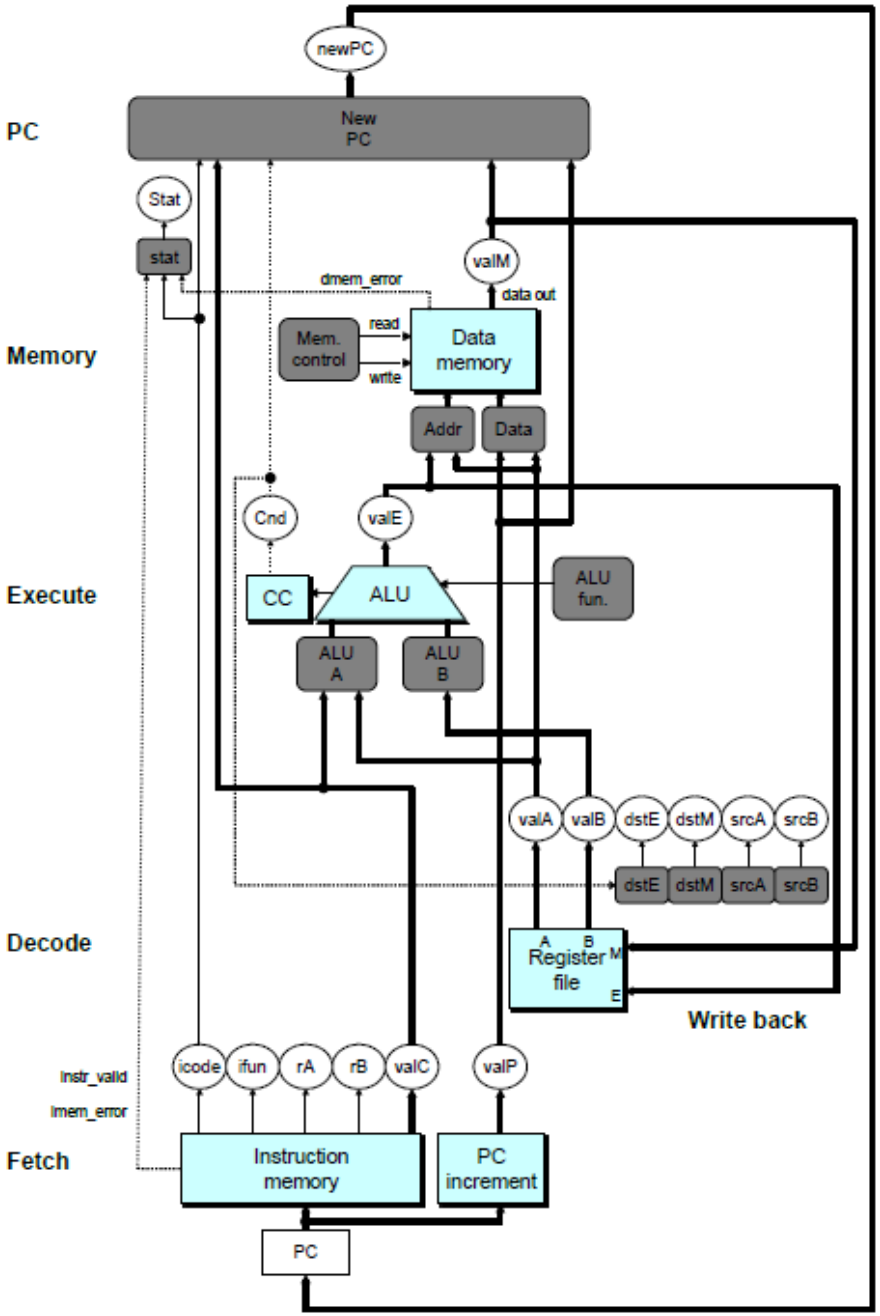
流水线化的处理器



- **结果没有及时反馈给下一次操作 Result does not feed back around in time for next operation**
- **流水线改变了系统的行为 Pipelining has changed behavior of system**

SEQ硬件
SEQ Hardware

- 顺序产生各个阶段 Stages occur in sequence
- 一次只有一个操作在进行处理 One operation in process at a time



SEQ+硬件
SEQ+ Hardware

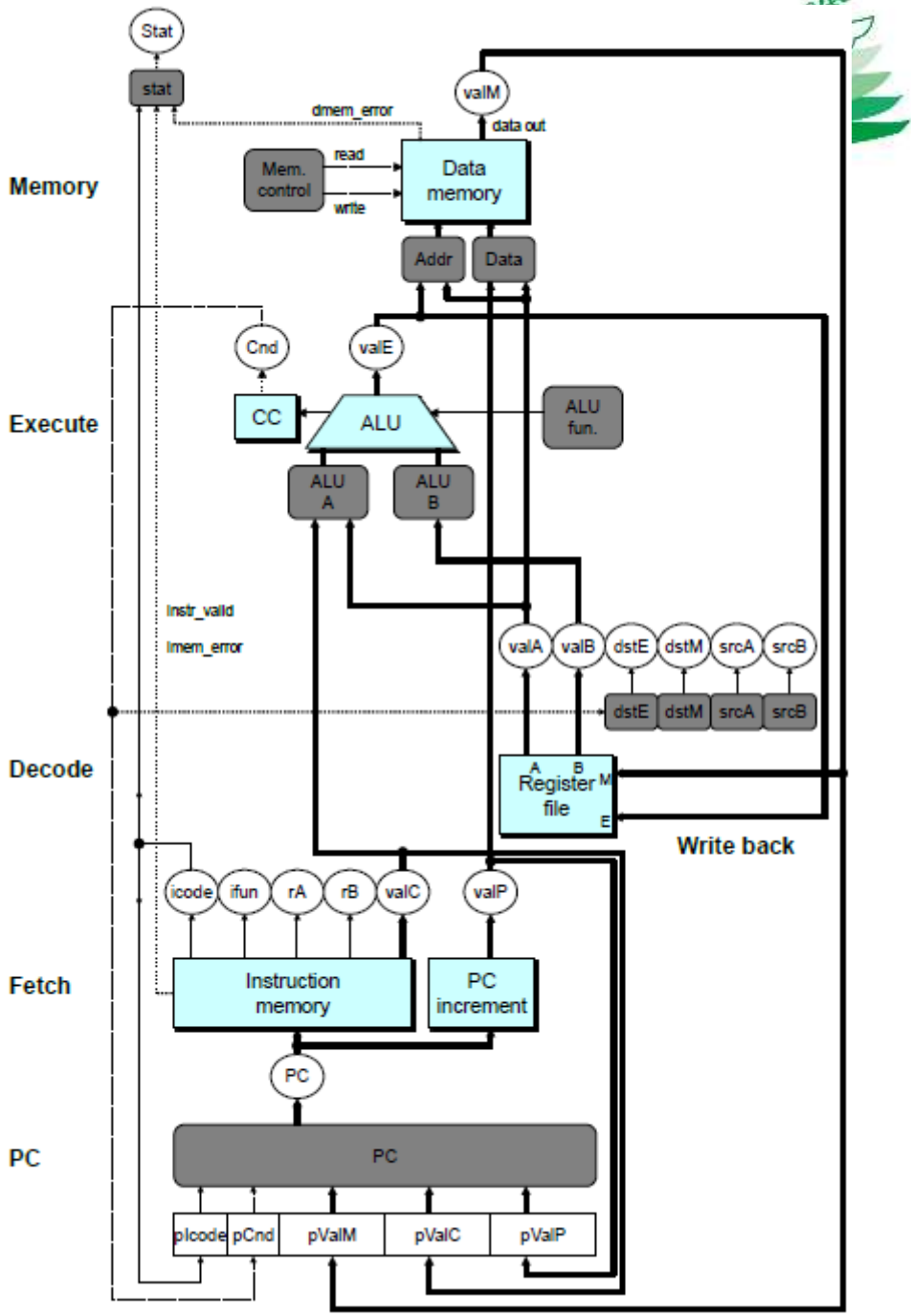
- 仍然是顺序实现 Still sequential implementation
- 记录PC阶段放在开始 Reorder PC stage to put at beginning

PC阶段 PC Stage

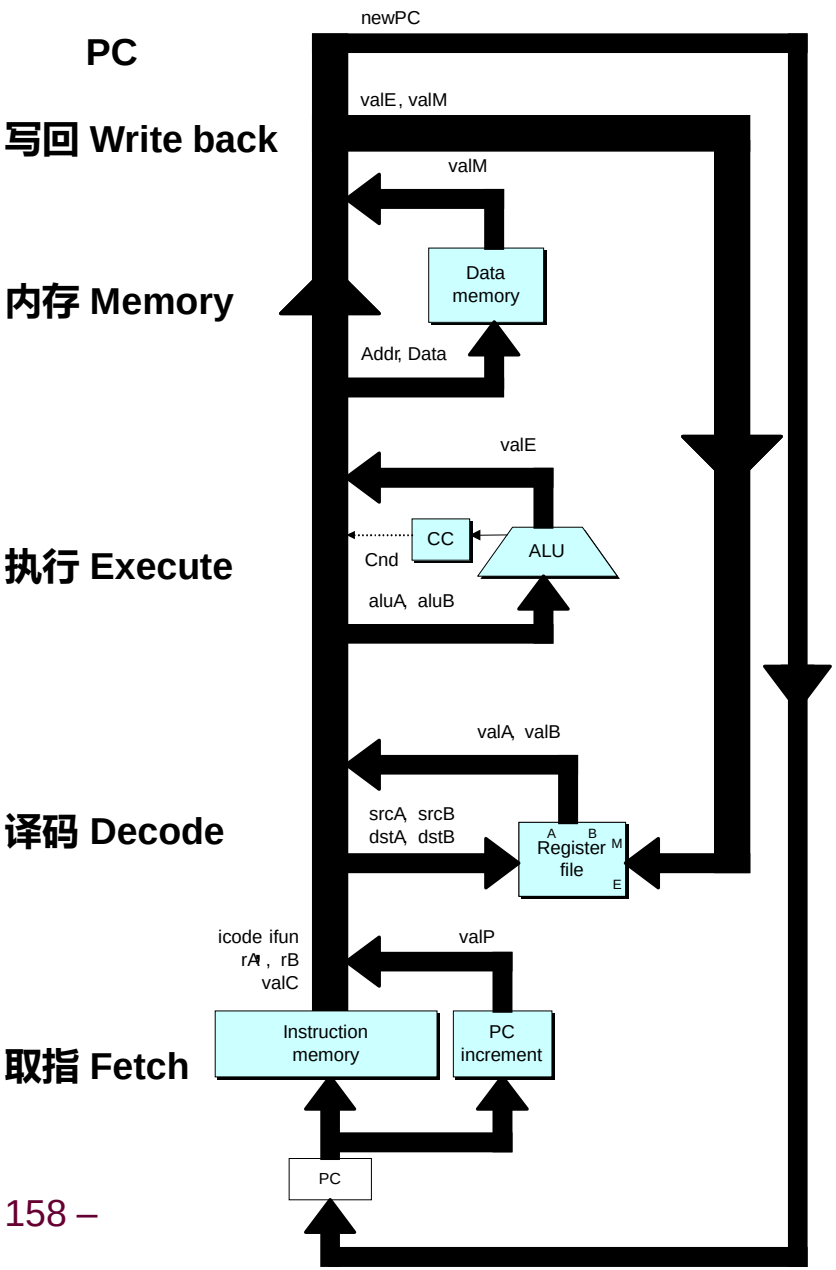
- 任务是为当前指令选择PC Task is to select PC for current instruction
- 根据上条指令计算的结果 Based on results computed by previous instruction

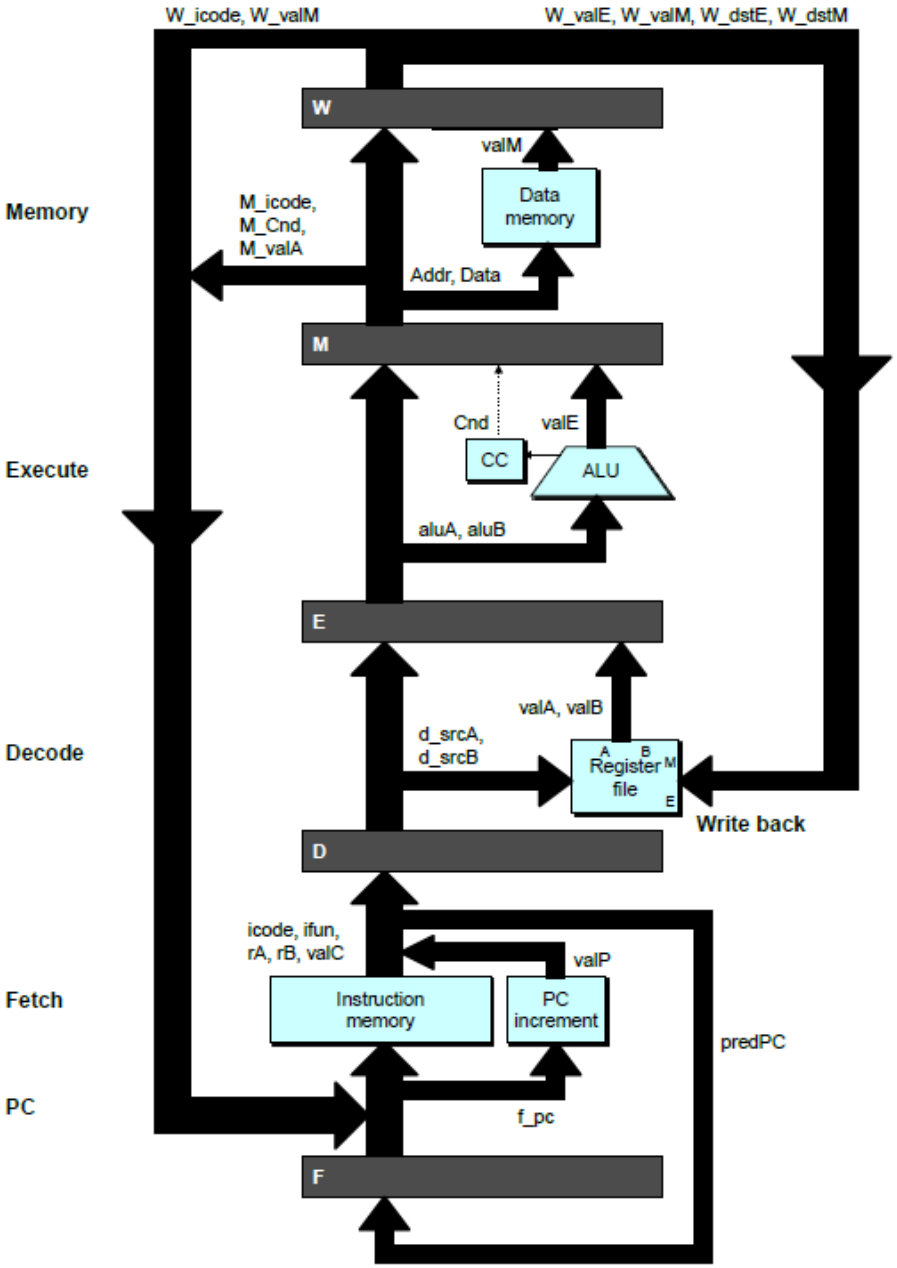
处理器状态 Processor State

- PC不再存储在寄存器中 PC is no longer stored in register
- 但是，可以根据其它存储信息确定PC But, can determine PC based on other stored information



增加流水线寄存器 Adding Pipeline Registers





流水线阶段 Pipeline Stages

取指 Fetch

- 选择当前PC Select current PC
- 读指令 Read instruction
- 计算PC增加值 Compute incremen

译码 Decode

- 读程序寄存器 Read program regis

执行 Execute

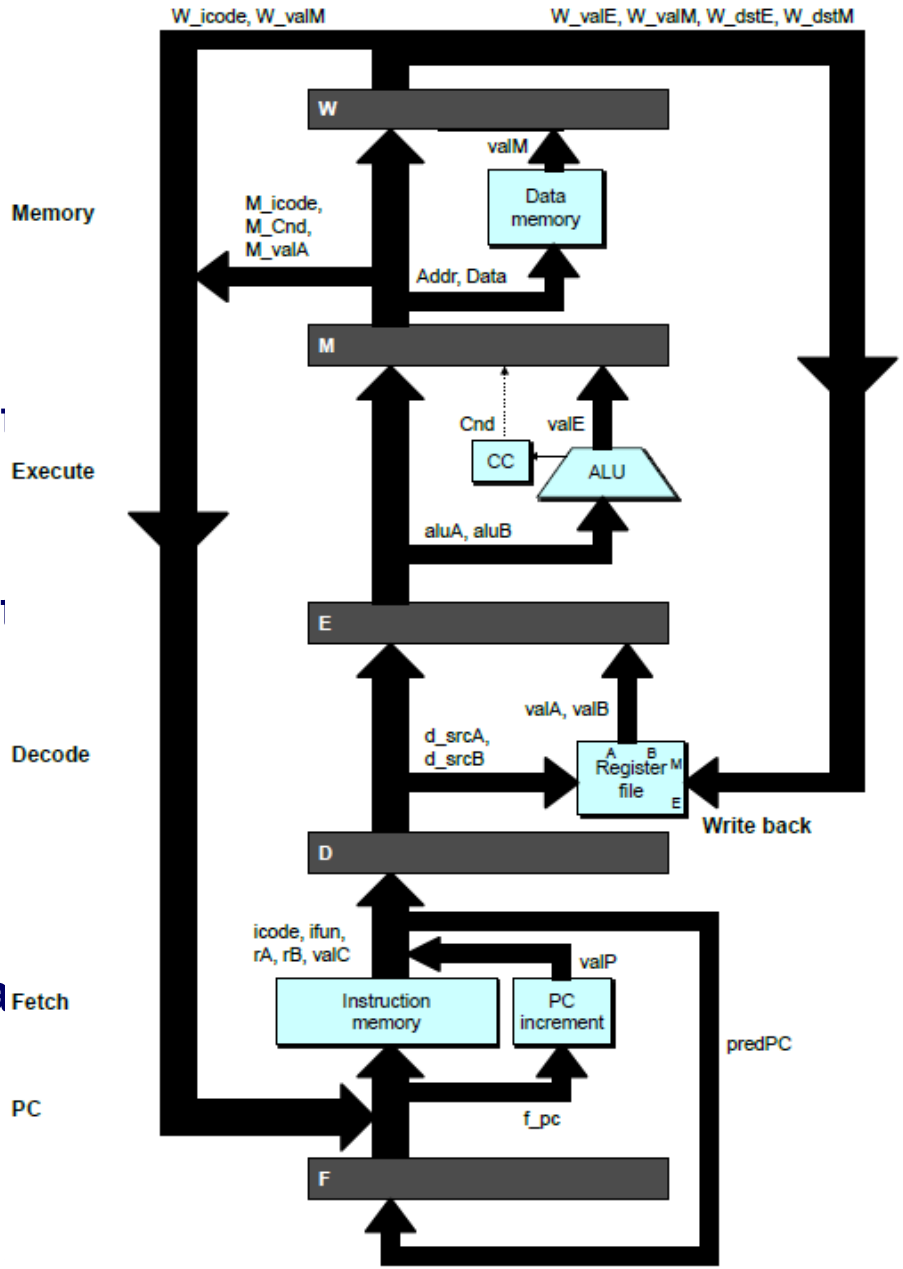
- 操作ALU Operate ALU

内存 Memory

- 读或写数据内存 Read or write data memory

写回 Write Back

- 更新寄存器文件（堆） Update regi file

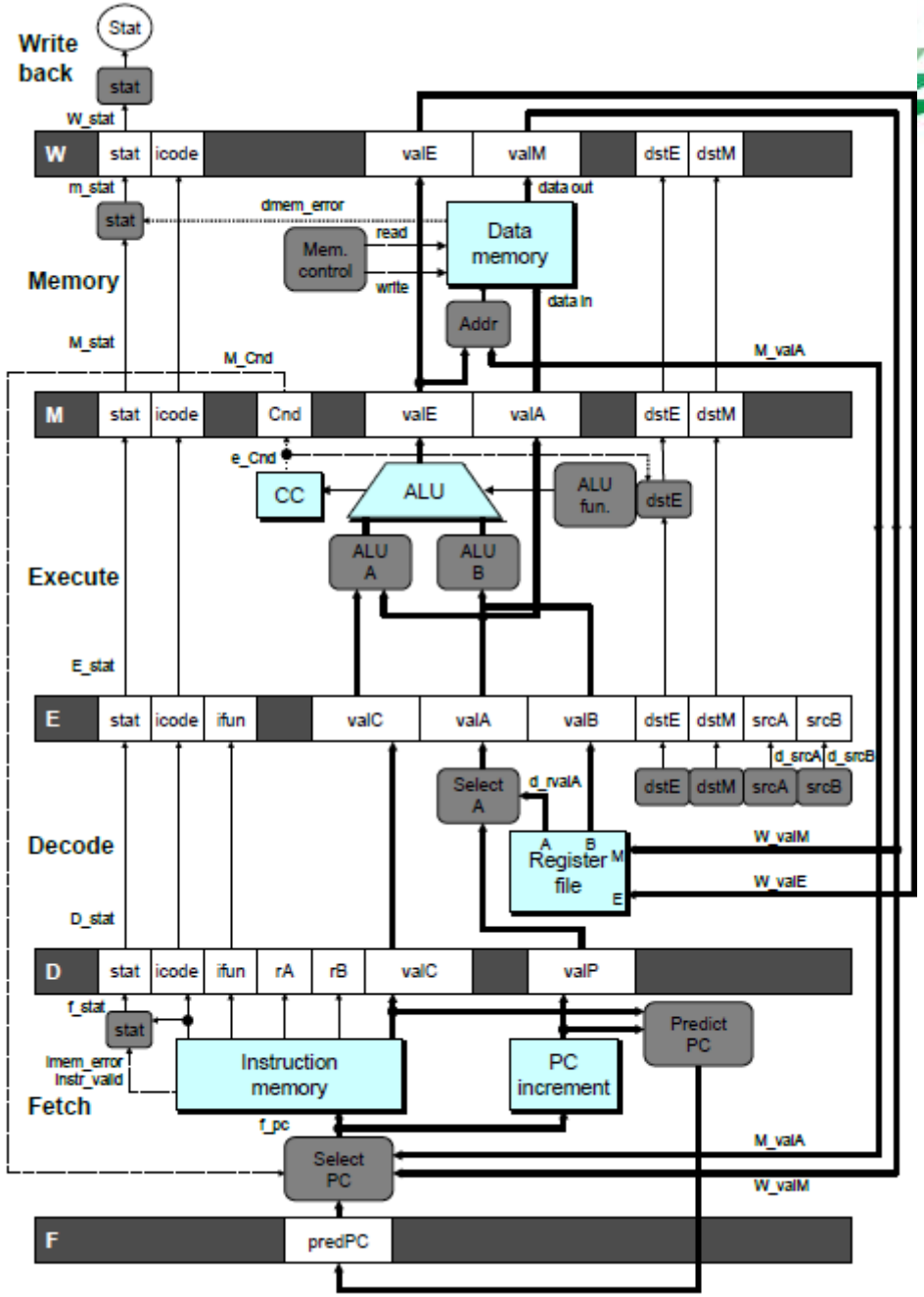


PIPE-硬件
PIPE- Hardware

- 流水线寄存器存储指令执行中的中间值 Pipeline registers hold intermediate values from instruction execution

转发（向前）路径 Forward (Upward) Paths

- 从一个阶段到下一个阶段传递值 Values passed from one stage to next
- 不能回跳到过去的阶段 Cannot jump past stages
  - 例如valC直传通过译码阶段 e.g., valC passes through decode



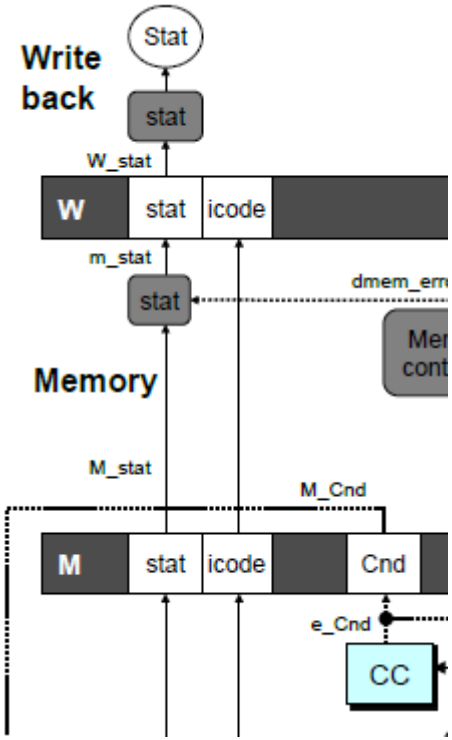
信号命名规则
Signal Naming Conventions

S_Field

- S阶段流水线寄存器中Field字段的值 Value of Field held in stage S pipeline register

s_Field

- S阶段中计算的Field字段的值 Value of Field computed in stage S



反馈路径 Feedback Paths

预测PC Predicted PC

- 猜测下一次PC的值 Guess value of next PC

分支信息 Branch information

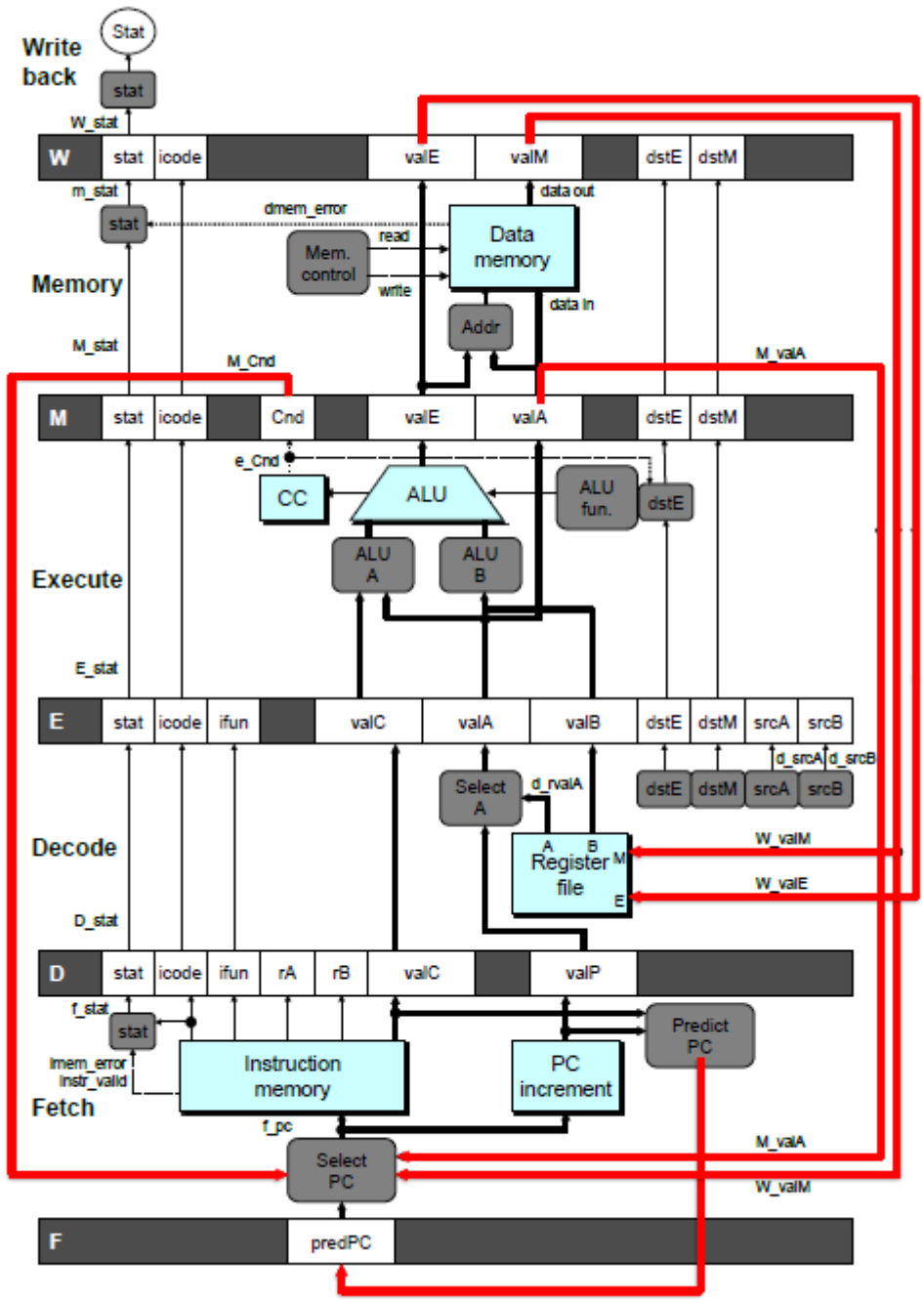
- 跳转/不跳转 Jump taken/not-taken
- 直落或目标地址 Fall-through or target address

返回点 Return point

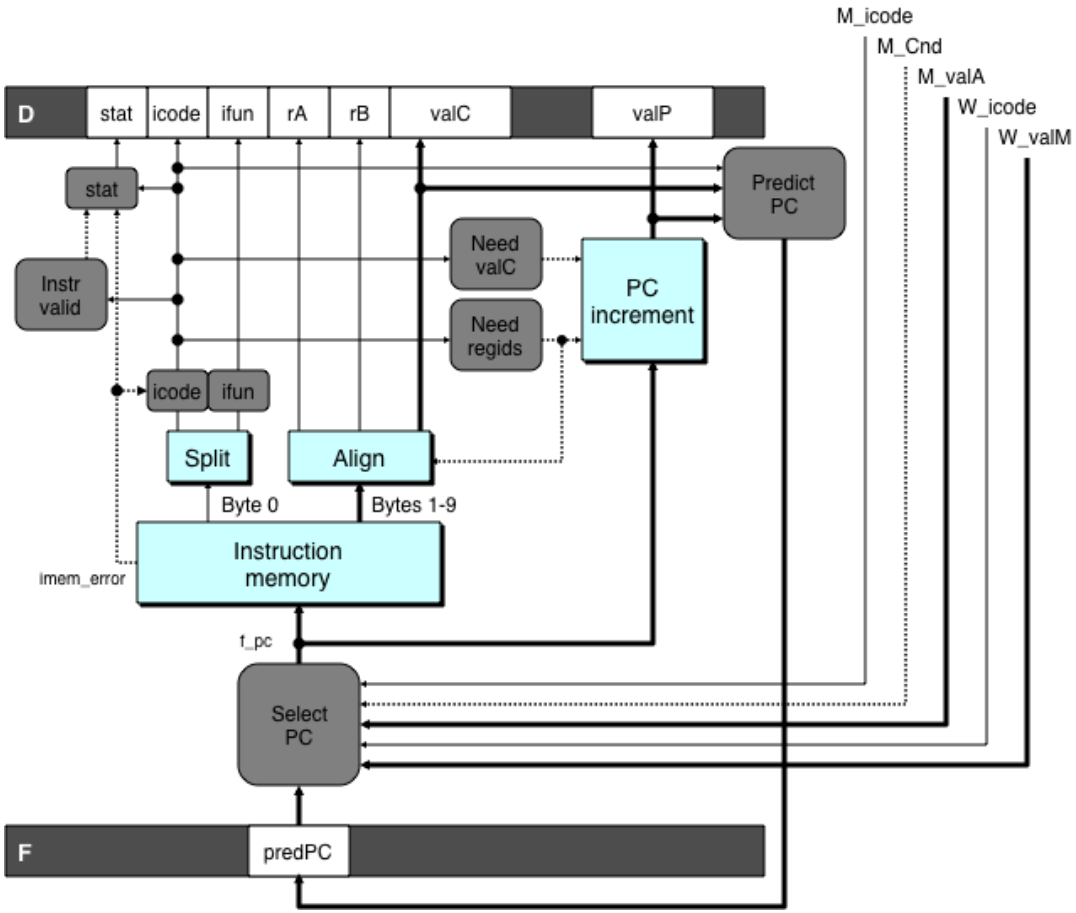
- 从内存读 Read from memory

寄存器更新 Register updates

- 寄存器文件的写端口 To register file write ports



预测PC Predicting the PC



- 当前指令已经完成取指阶段后，开始新指令取指阶段 Start fetch of new instruction after current one has completed fetch stage
  - 没有充足的时间来可靠地确定下一条指令 Not enough time to reliably determine next instruction
- 猜测哪条是下一条指令 Guess which instruction will follow
  - 如果预测不正确则恢复 Recover if prediction was incorrect

我们的预测策略
Our Prediction Strategy

不转换控制的指令 Instructions that Don't Transfer Control

- 预测下一个PC为valP Predict next PC to be valP
- 总是可靠的 Always reliable

过程调用和无条件跳转指令 Call and Unconditional Jumps

- 预测下一个PC为valC（目标地址） Predict next PC to be valC (destination)
- 总是可靠的 Always reliable

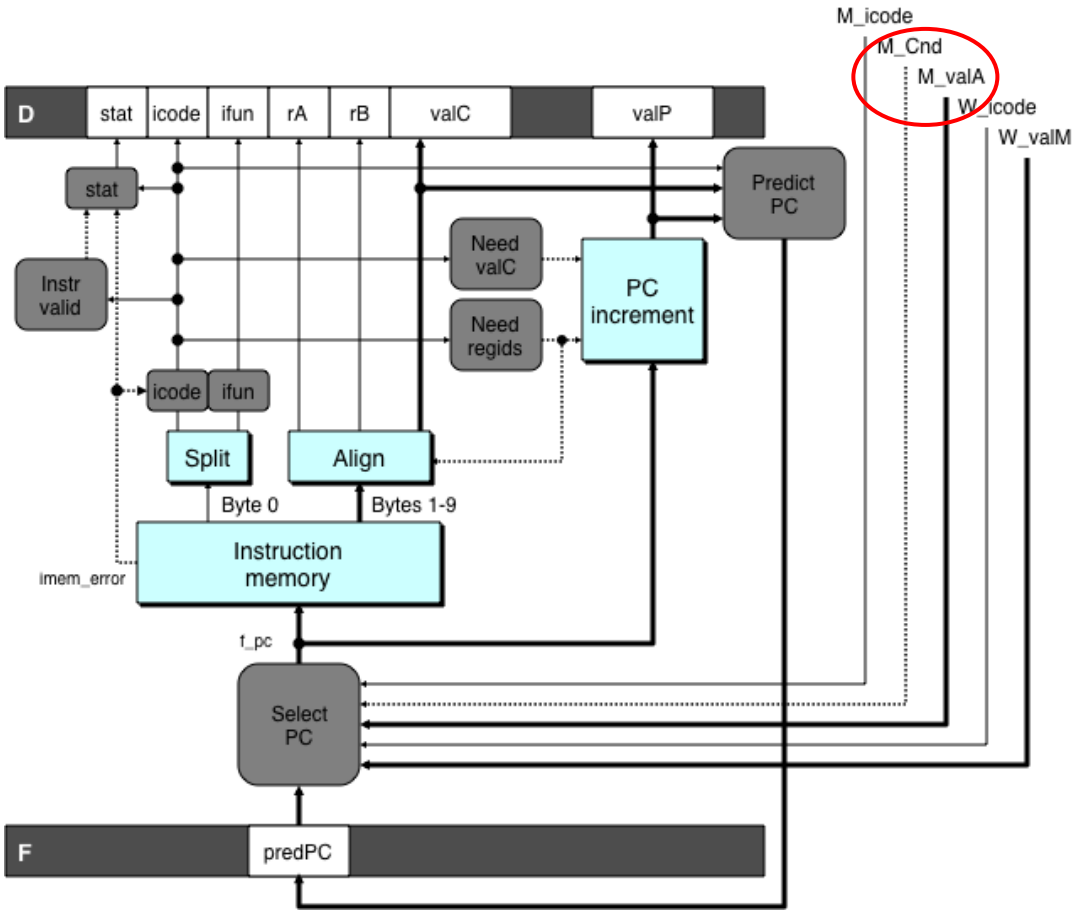
条件跳转指令 Conditional Jumps

- 预测下一个PC为valC（目标地址） Predict next PC to be valC (destination)
- 仅在选择分支时正确 Only correct if branch is taken
  - 典型的正确率为60% Typically right 60% of time

返回指令 Return Instruction

- 不进行预测 Don't try to predict

从PC预测错误中恢复 Recovering from PC Misprediction

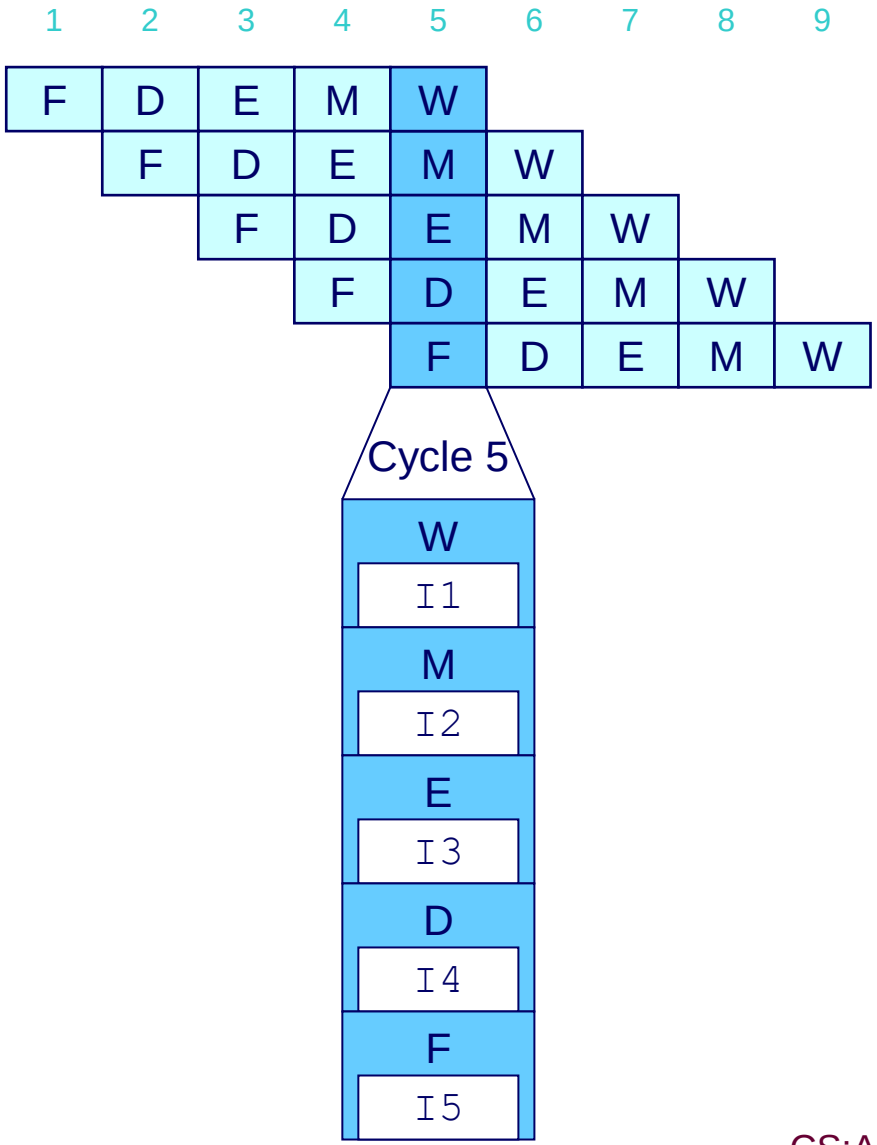


- 错误预测跳转 Mispredicted Jump
  - 一旦指令到达内存阶段，看分支条件标志 Will see branch condition flag once instruction reaches memory stage
  - 可以从valA（M_valA值）中得到直落PC Can get fall-through PC from valA (value M_valA)
- 返回指令 Return Instruction
  - 当返回指令到达写回阶段（W_valM）时得到返回PC Will get return PC when `ret` reaches write-back stage (W_valM)

流水线演示 Pipeline Demonstration

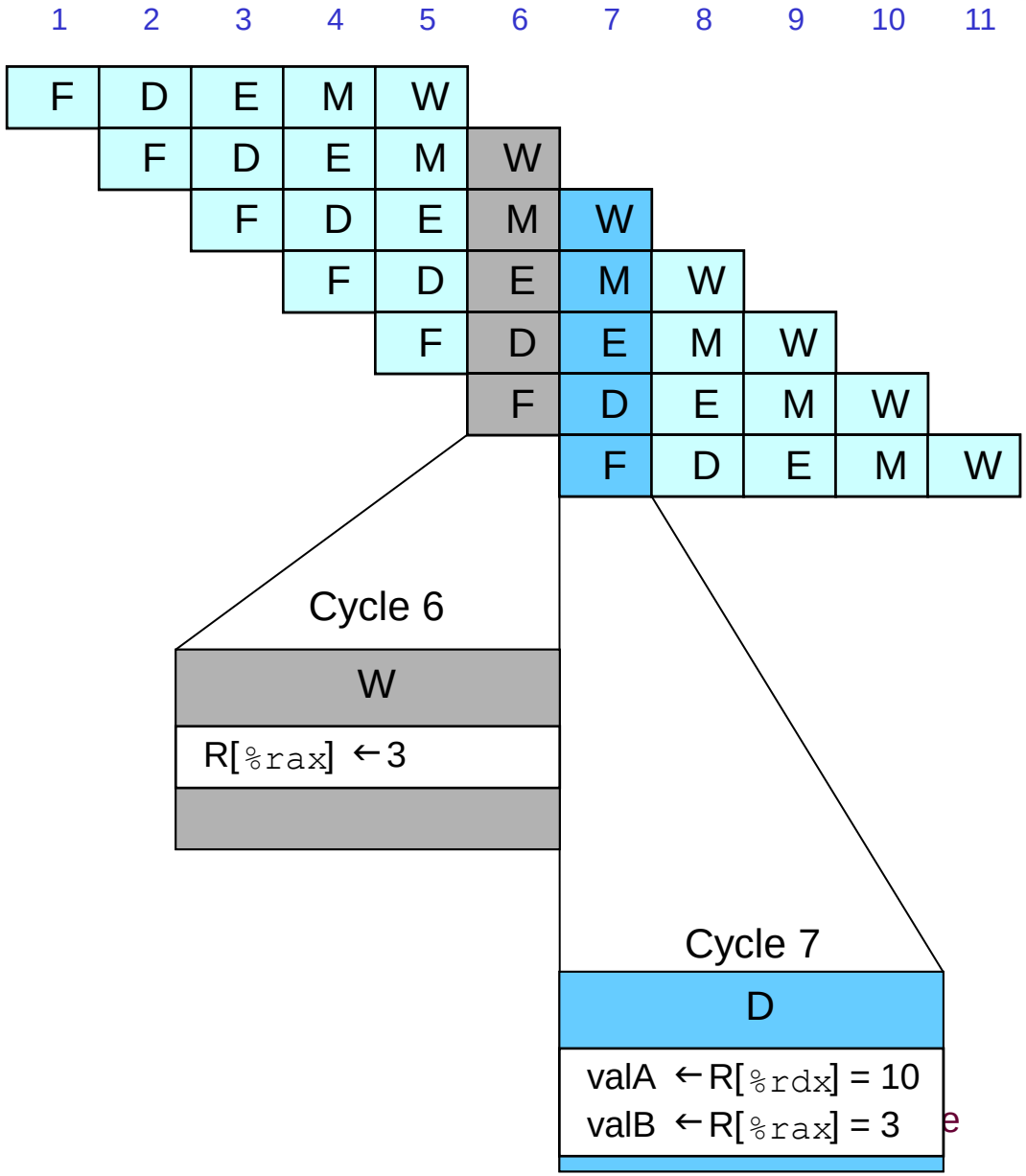
```
irmovq    $1,%rax   #I1
irmovq    $2,%rcx   #I2
irmovq    $3,%rdx   #I3
irmovq    $4,%rbx   #I4
halt                #I5
```

文件 File: `demo-basic.ys`



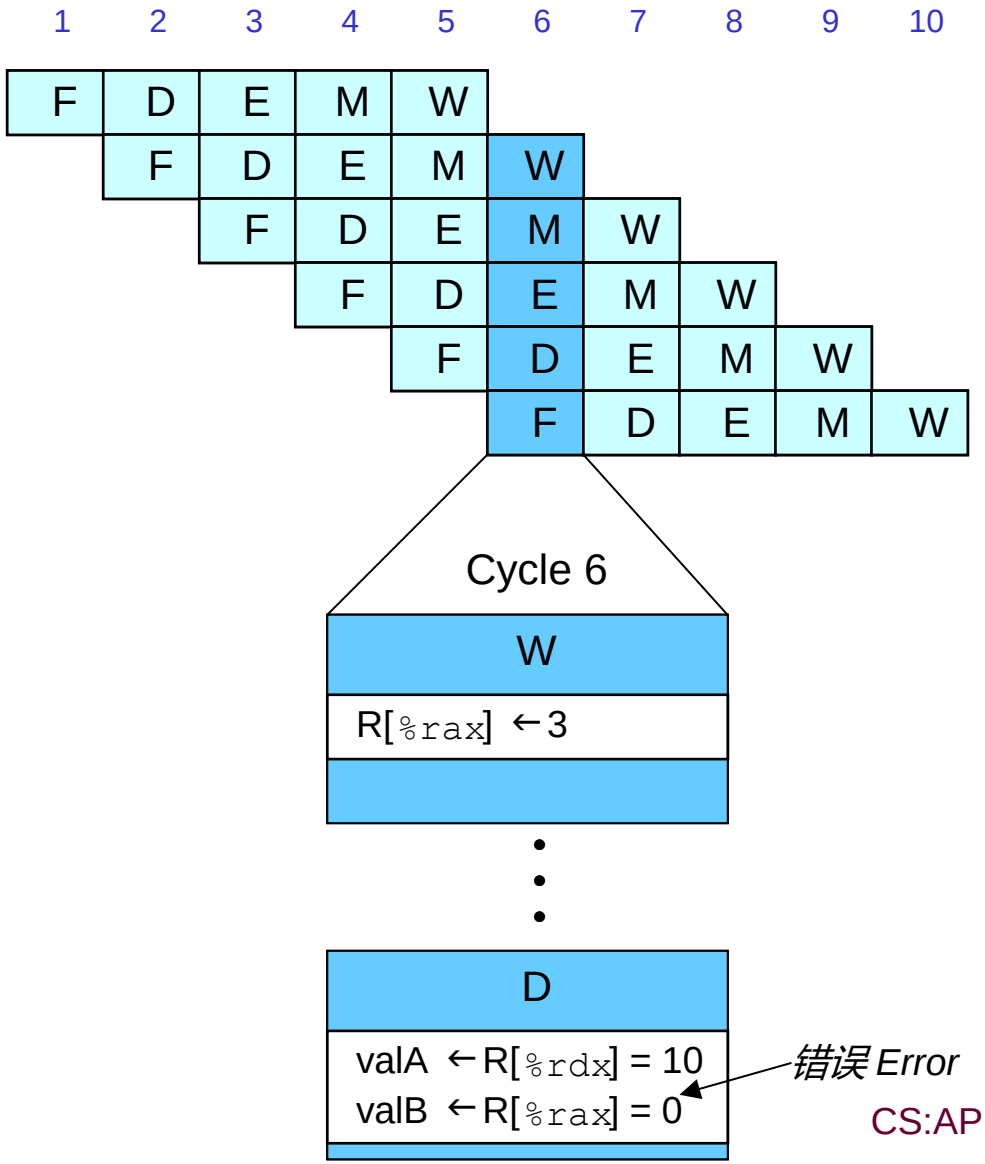
数据相关：3条空指令
Data Dependencies: 3 Nop's

```
# demo-h3.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: nop
0x017: addq %rdx,%rax
0x019: halt
```



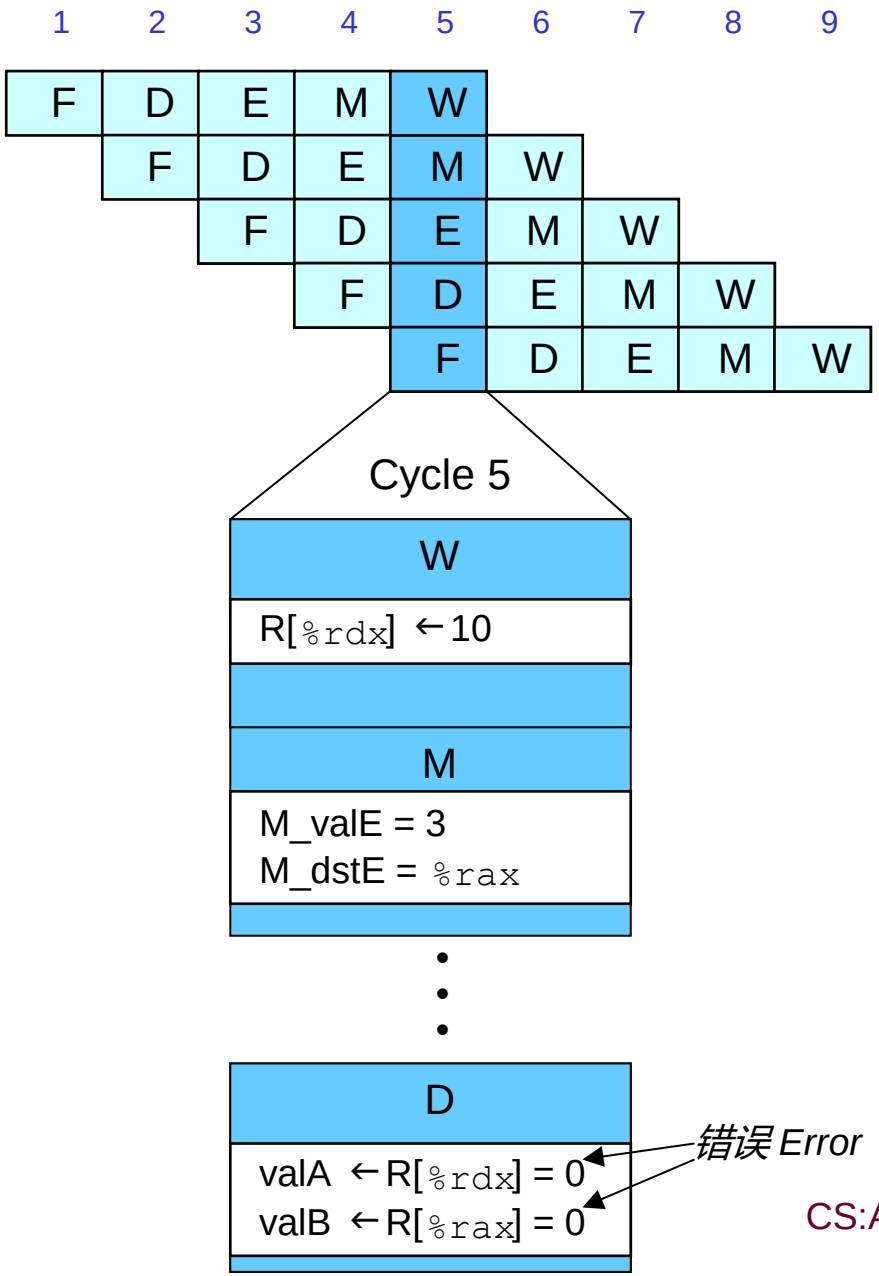
数据相关：2条空指令
Data Dependencies: 2 Nop's

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



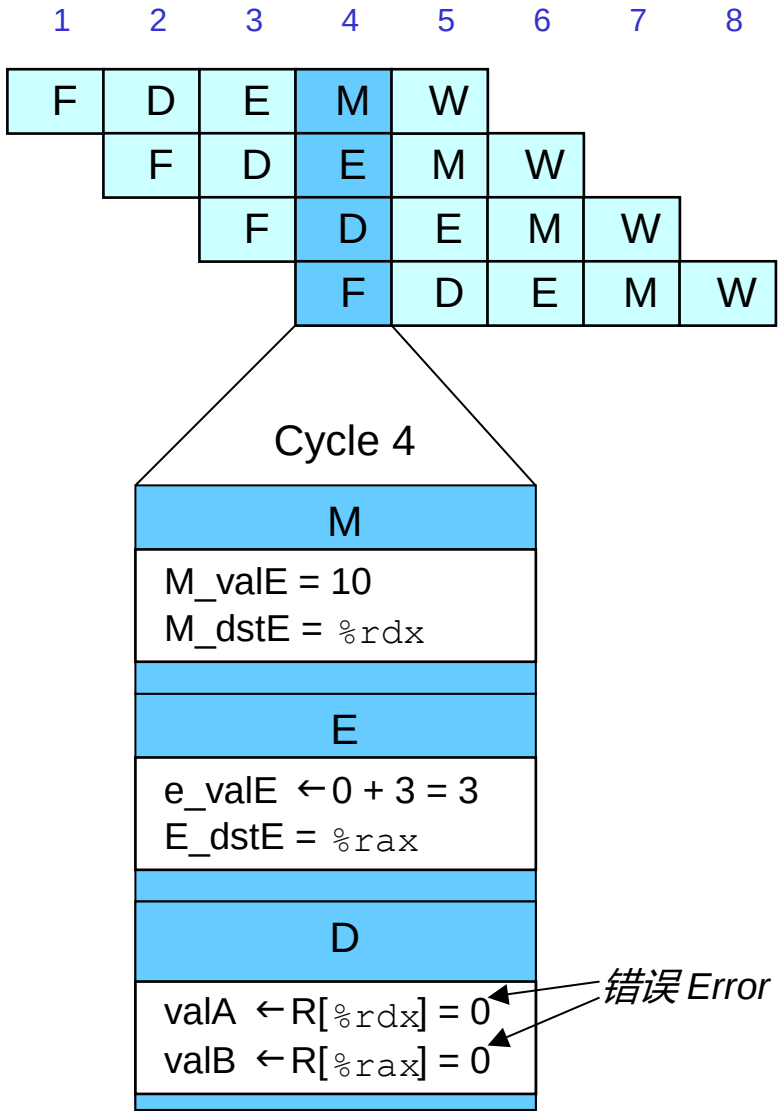
数据相关：1条空指令 Data Dependencies: 1 Nop

```
# demo-h1.ys

0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: addq %rdx,%rax
0x017: halt
```



数据相关：无空指令 Data Dependencies: No Nop

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



分支预测错误示例
Branch Misprediction Example

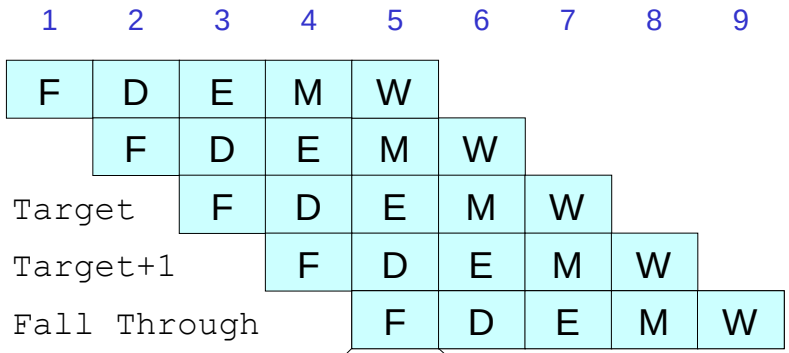
demo-j.ys

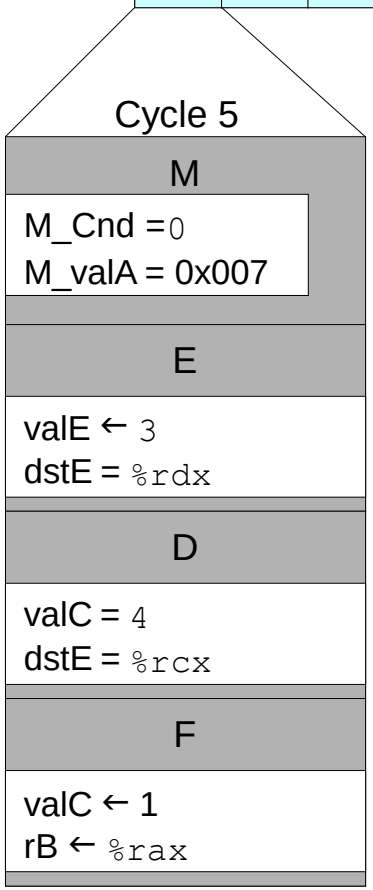
```
0x000:     xorq %rax,%rax
0x002:     jne  t                # Not taken
0x00b:     irmovq $1, %rax       # Fall through
0x015:     nop
0x016:     nop
0x017:     nop
0x018:     halt
0x019: t:  irmovq $3, %rdx       # Target (Should not execute)
0x023:     irmovq $4, %rcx       # Should not execute
0x02d:     irmovq $5, %rdx       # Should not execute
```

- 应该仅执行前8条指令 Should only execute first 8 instructions

分支预测错误跟踪 Branch Misprediction Trace

```
# demo-j
0x000:    xorq %rax,%rax
0x002:    jne t # Not taken
0x019: t: irmovq $3, %rdx # Target
0x023:    irmovq $4, %rcx # Target+1
0x00b:    irmovq $1, %rax # Fall Through
```





- **不正确地执行分支目标处的两条指令 Incorrectly execute two instructions at branch target**

返回示例
Return Example

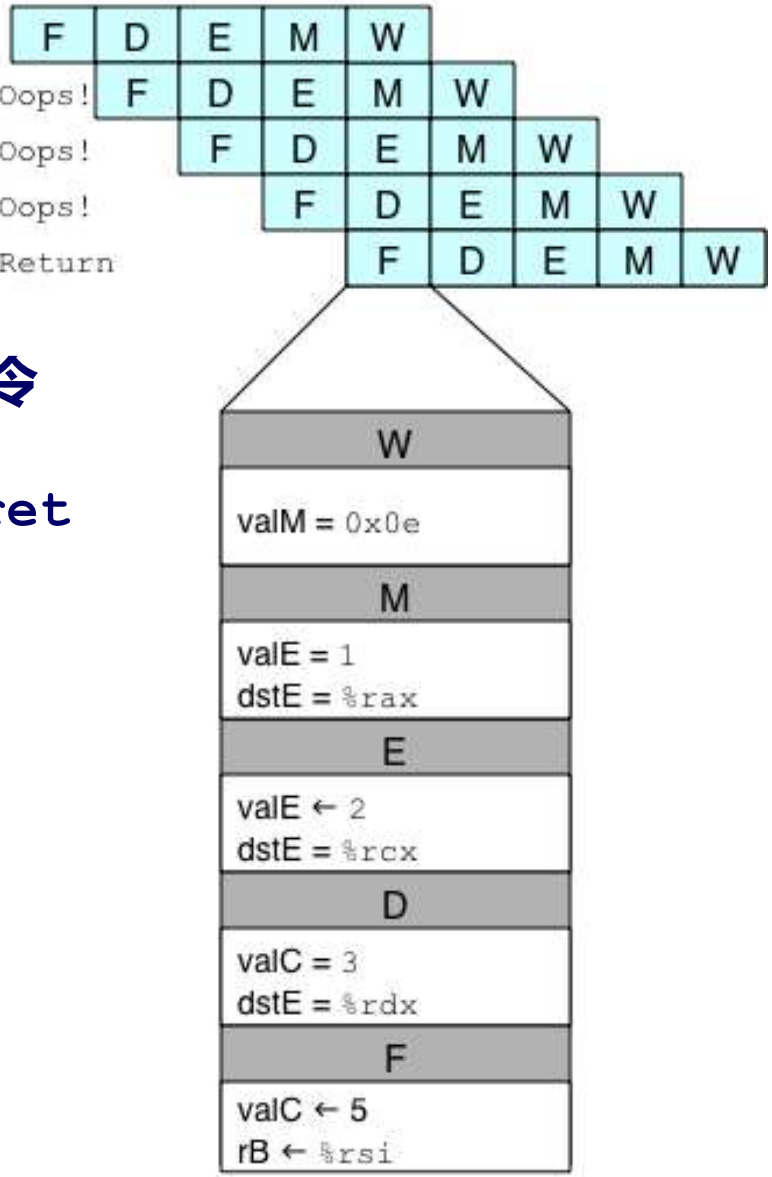
demo-ret.ys

```
0x000:     irmovq Stack,%rsp    # Intialize stack pointer
0x00a:     nop                  # Avoid hazard on %rsp
0x00b:     nop
0x00c:     nop
0x00d:     call p               # Procedure call
0x016:     irmovq $5,%rsi       # Return point
0x020:     halt
0x020:  .pos 0x20
0x020:  p: nop                   # procedure
0x021:     nop
0x022:     nop
0x023:     ret
0x024:     irmovq $1,%rax        # Should not be executed
0x02e:     irmovq $2,%rcx        # Should not be executed
0x038:     irmovq $3,%rdx        # Should not be executed
0x042:     irmovq $4,%rbx        # Should not be executed
0x100:  .pos 0x100
0x100:  Stack:                   # Initial stack pointer
```

- 需要很多空指令来避免数据冒险 Require lots of nops to avoid data hazards

不正确返回示例 Incorrect Return Example

```
# demo-ret
0x033:    ret
0x034:    irmovq $1,%rax # Oops!
0x03e:    irmovq $2,%rcx # Oops!
0x048:    irmovq $3,%rdx # Oops!
0x052:    irmovq $5,%rsi # Return
```



- 错误执行ret后面的3条指令
  Incorrectly execute 3 instructions following `ret`

流水线小结 Pipeline Summary

概念 Concept

- 把指令执行分成5个阶段 Break instruction execution into 5 stages
- 以流水线模式运行指令 Run instructions through in pipelined mode

限制 Limitations

- 当指令流太紧密时不能处理指令之间的相关性 Can't handle dependencies between instructions when instructions follow too closely
- 数据相关 Data dependencies
  - 一条指令写寄存器，然后一条指令读它 One instruction writes register, later one reads it
- 控制相关 Control dependency
  - 指令设置PC的方式，不是流水线正确预测的结果 Instruction sets PC in way that pipeline did not predict correctly
  - 预测失误的分支和返回 Mispredicted branch and return

修正流水线 Fixing the Pipeline

- 下一次课完成这个工作 We'll do that next time

CS:APP Chapter 4 Computer Architecture
Pipelined Implementation
Part II
流水线实现
第二部分

任课教师：

宿红毅　张艳　黎有琦　颜珂

原作者：

Randal E. **Bryant and** David R. O'Hallaron

Carnegie Mellon University

概述 Overview

使流水线处理器工作 Make the pipelined processor work!

数据冒险 Data Hazards

- 以寄存器 R 作为源的指令紧跟在以寄存器 R 作为目的的指令之后 Instruction having register R as source follows shortly after instruction having register R as destination
- 常见情况，不想减慢流水线 Common condition, don't want to slow down pipeline

控制冒险 Control Hazards

- 错误预测条件分支 Mispredict conditional branch
  - 我们的设计预测所有分支为选择分支 Our design predicts all branches as being taken
  - 朴素流水线执行两条额外指令 Naïve pipeline executes two extra instructions
- 获得ret指令的返回地址 Getting return address for `ret` instruction
  - 朴素流水线执行三条额外指令 Naïve pipeline executes three extra instructions

概述 Overview

使流水线处理器工作 Make the pipelined processor work!

确保它真正工作 Making Sure It Really Works

- **如果多种特殊情况同时发生会怎么样? What if multiple special cases happen simultaneously?**

流水线阶段 Pipeline Stages

取指 Fetch

- 选择当前PC Select current PC
- 读指令 Read instruction
- 计算PC增加值 Compute incremented PC

Memory

Execute

译码 Decode

- 读程序寄存器 Read program registers

执行 Execute

Decode

- 操作ALU Operate ALU

内存 Memory

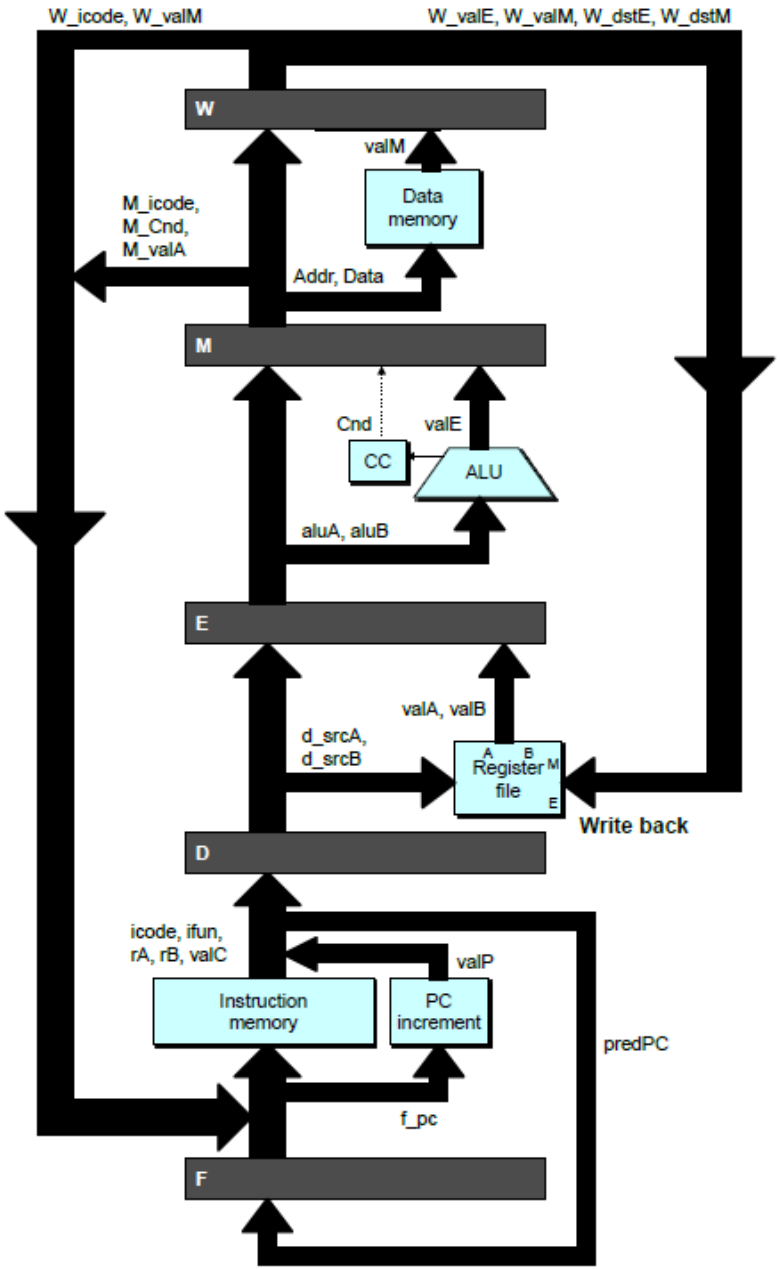
- 读或写数据内存 Read or write data memory

Fetch

PC

写回 Write Back

- 更新寄存器文件（堆） Update register file

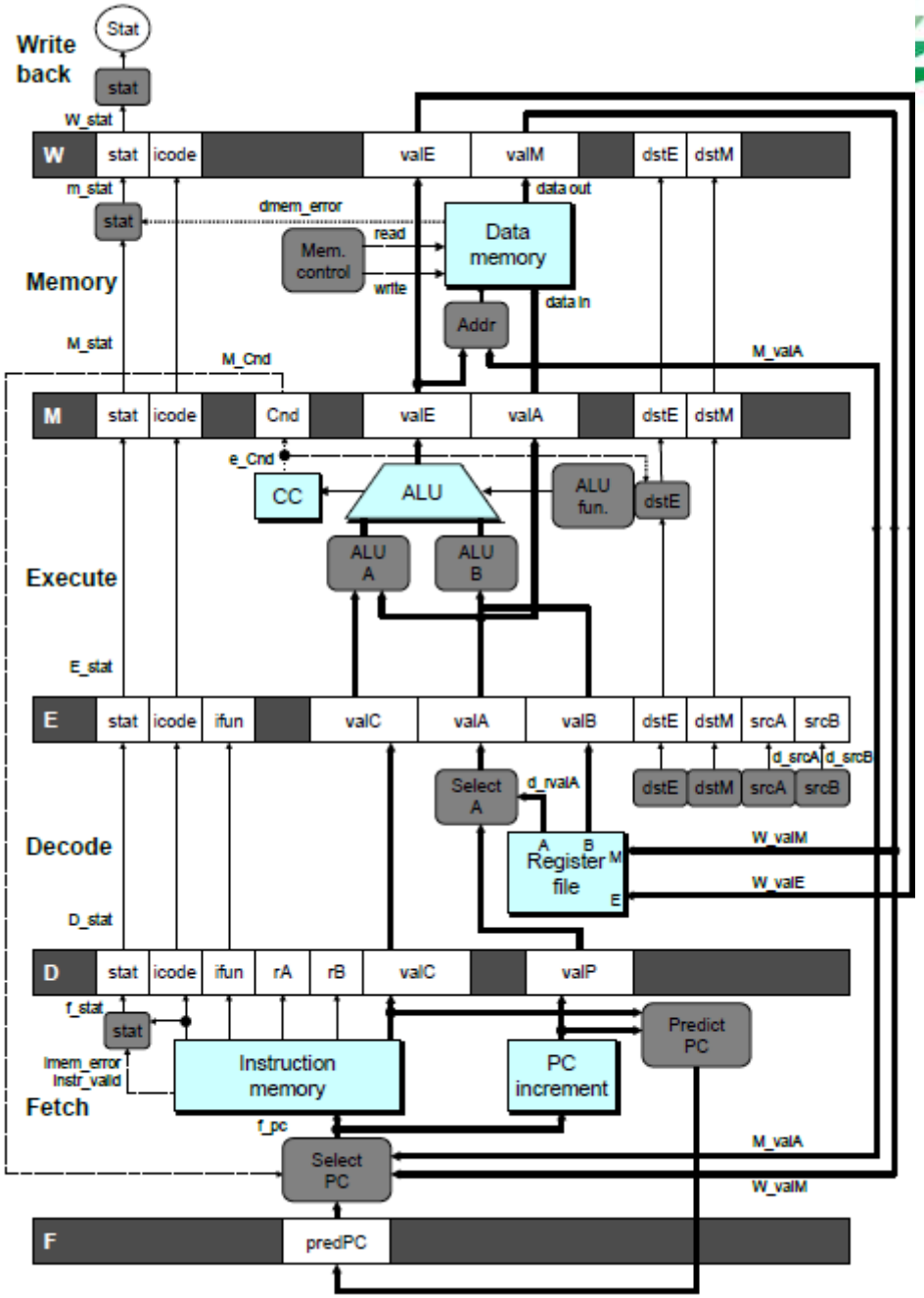


流水线硬件 PIPE- Hardware

- 流水线寄存器存储指令执行过程的中间值 Pipeline registers hold intermediate values from instruction execution

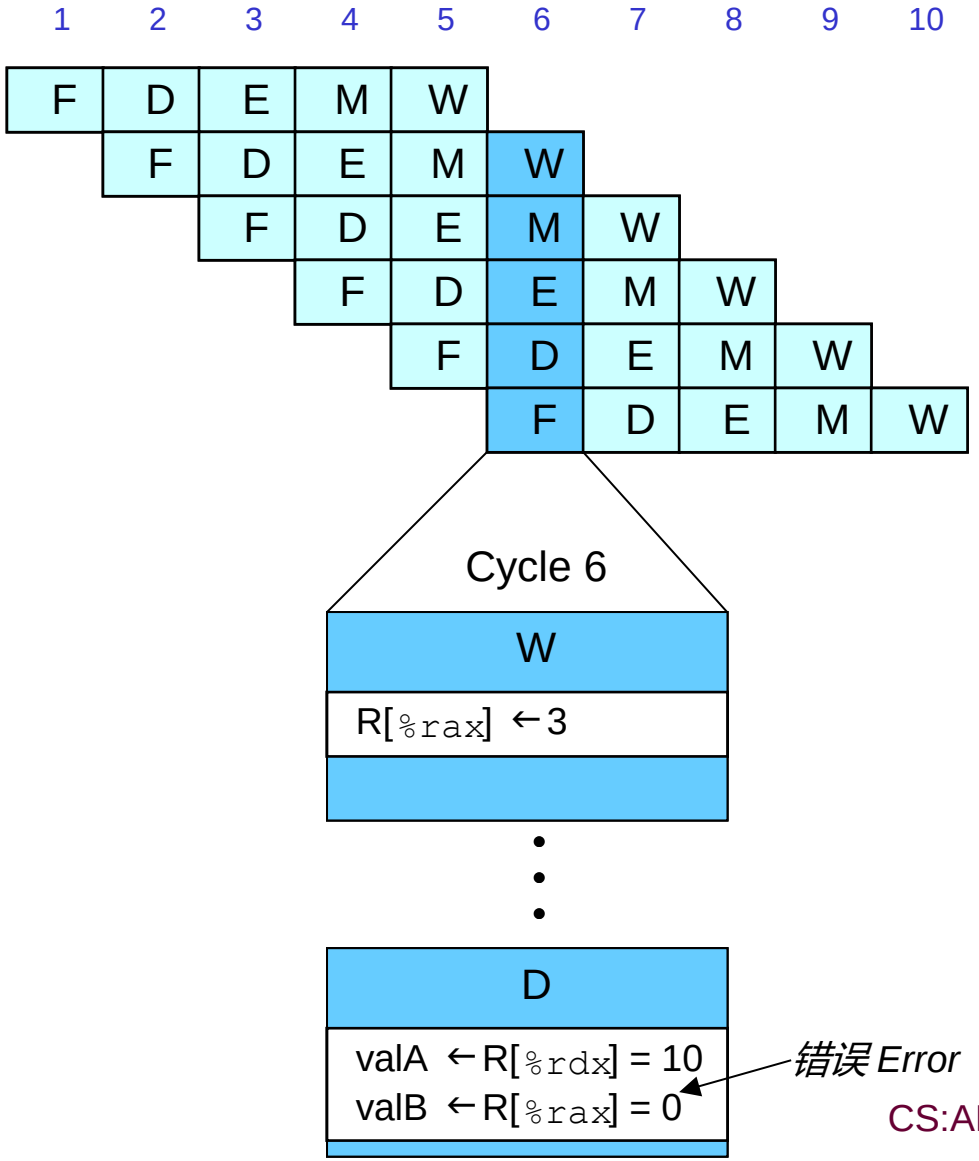
转发（向前）路径 Forward (Upward) Paths

- 从一个阶段向下一个阶段传递值 Values passed from one stage to next
- 不能跳转到过去阶段 Cannot jump past stages
  - 例如valC直传通过译码阶段 e.g., valC passes through decode



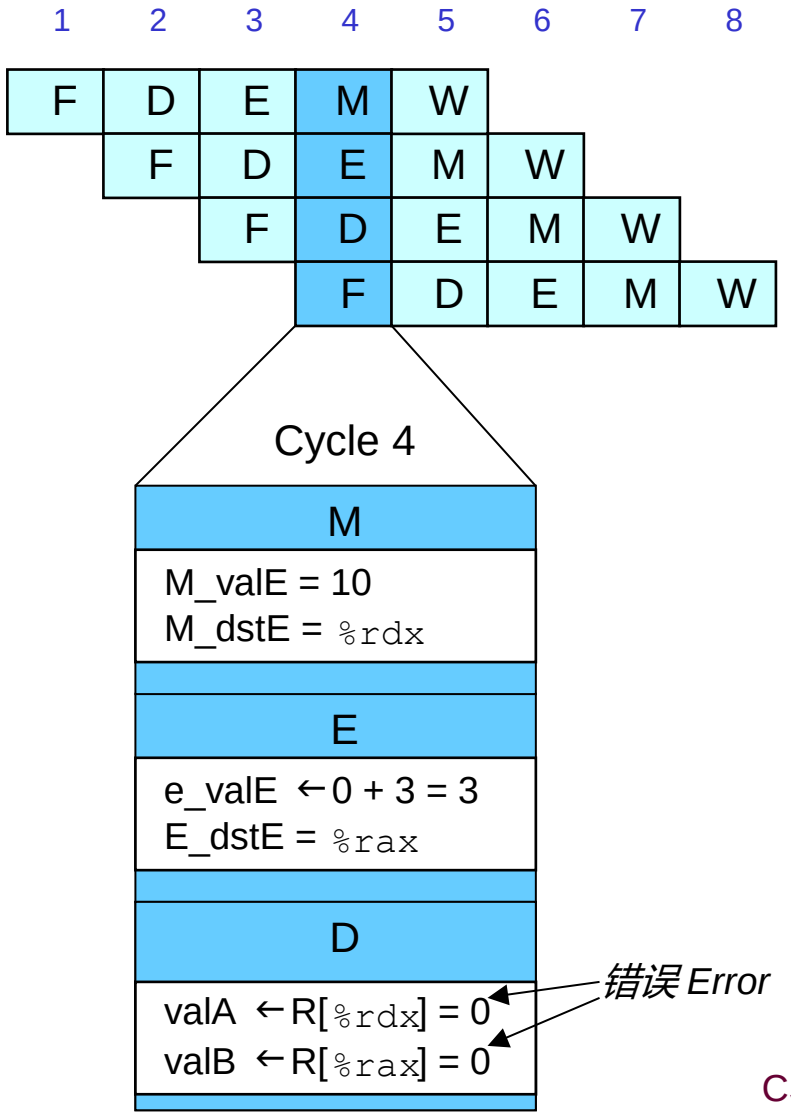
数据相关：2条空指令
Data Dependencies: 2 Nop's

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```



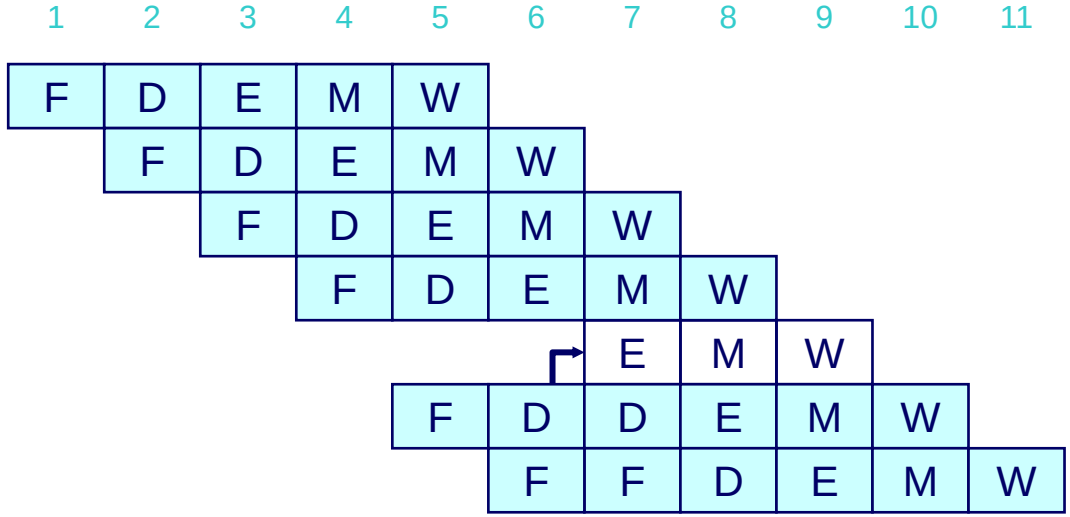
数据相关：没有空指令
Data Dependencies: No Nop

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



暂停解决数据相关
Stalling for Data Dependencies

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addq %rdx,%rax
0x018: halt
```



- 如果一条写寄存器指令后面指令流太紧密，需要慢下来 If instruction follows too closely after one that writes register, slow it down
- 保持指令在译码阶段 Hold instruction in decode
- 动态注入空指令到执行阶段 Dynamically inject nop into execute stage
- 气泡相当于虚拟空指令，只不过不存储在指令内存

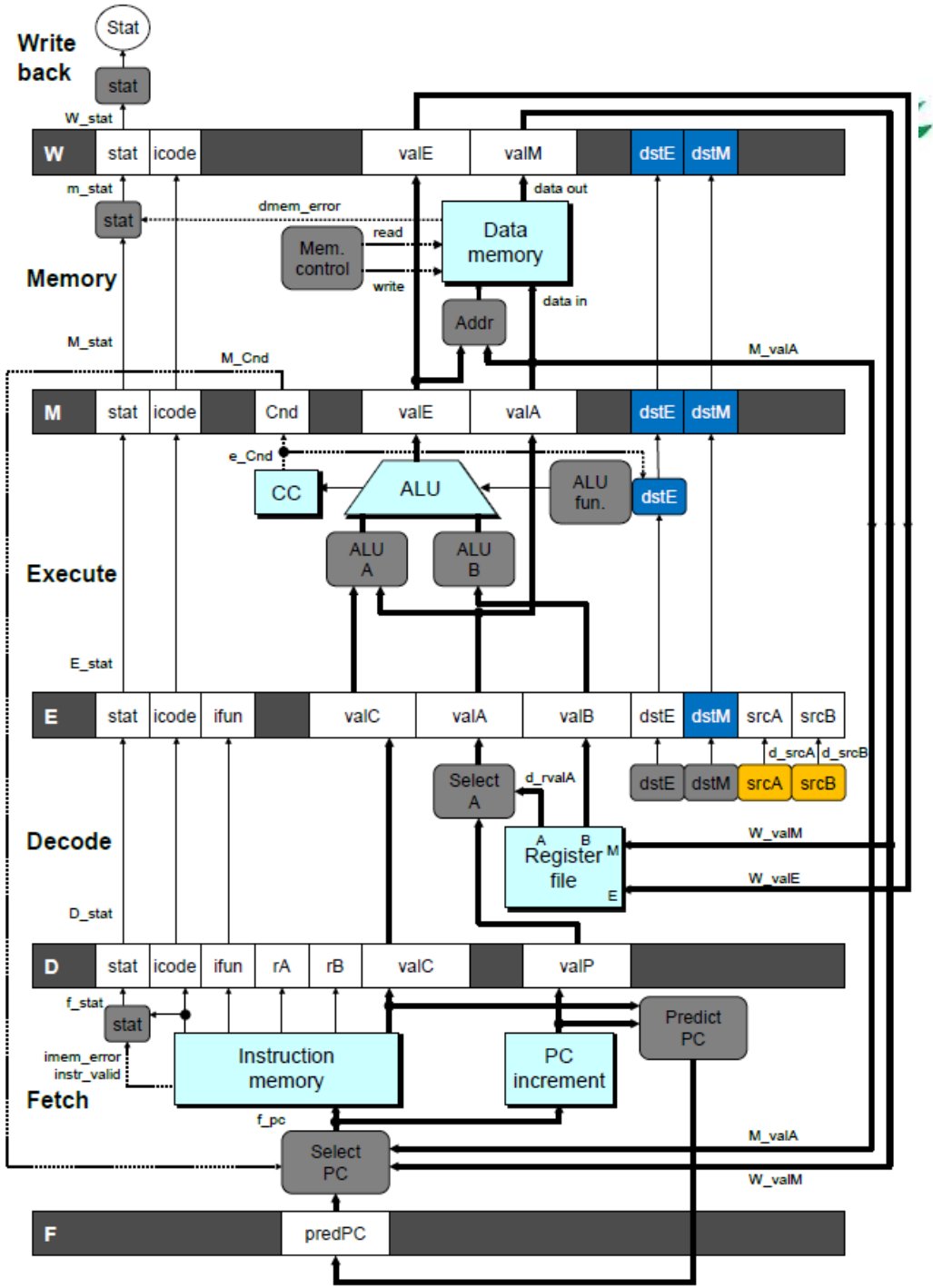
暂停条件 Stall Condition

源寄存器 Source Registers

- 译码阶段的当前指令的srcA和srcB srcA and srcB of current instruction in decode stage

目的寄存器 Destination Registers

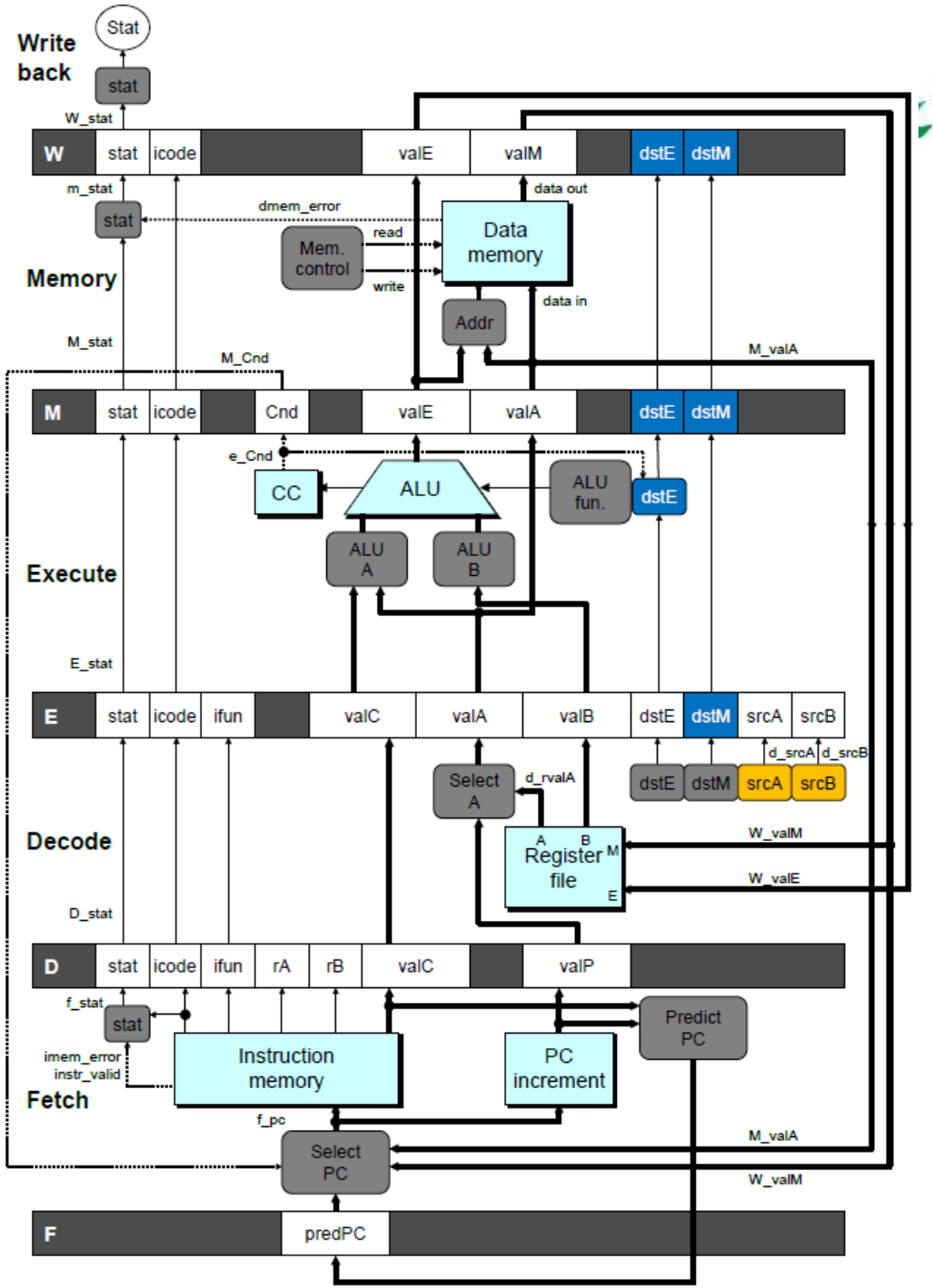
- dstE和dstM字段 dstE and dstM fields
- 指令在执行、访存和写回阶段 Instructions in execute, memory, and write-back stages



暂停条件 Stall Condition

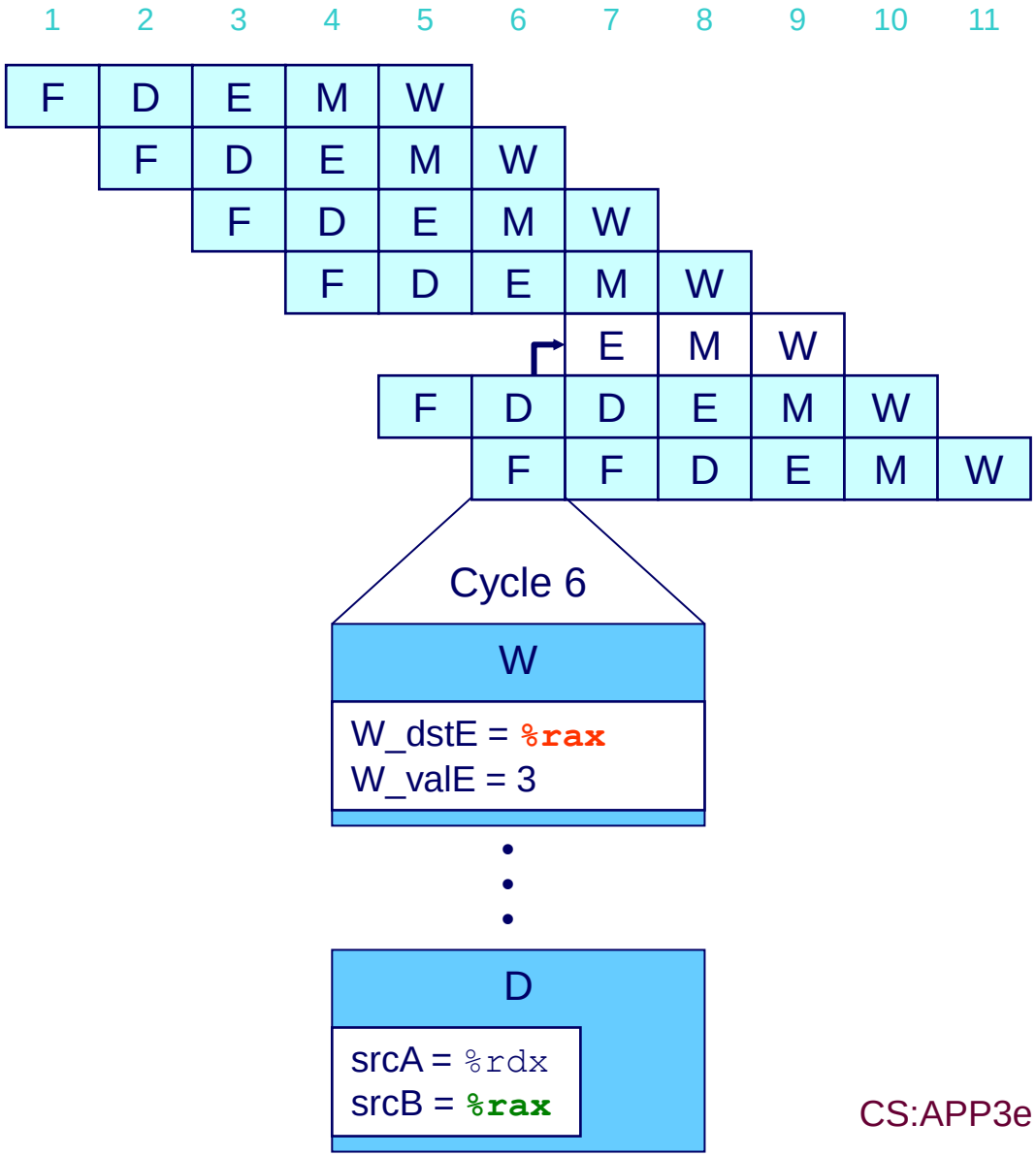
特殊情况 Special Case

- 不要暂停ID为15（0xF）的寄存器 Don't stall for register ID 15 (0xF)
  - 指明没有寄存器操作数 Indicates absence of register operand
  - 或失败的条件传送 Or failed cond. move



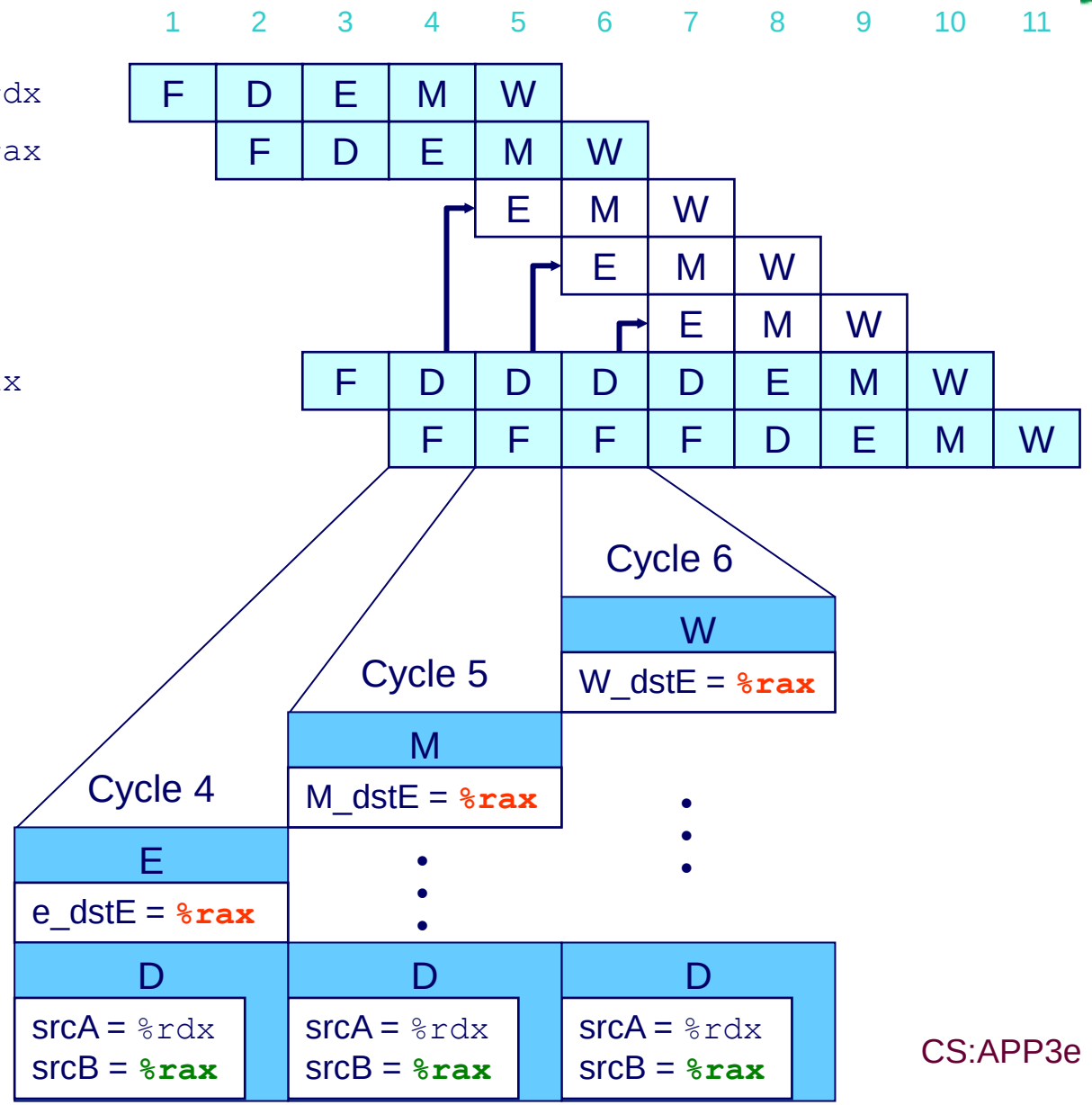
检测暂停条件 Detecting Stall Condition

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
       bubble
0x016: addq %rdx,%rax
0x018: halt
```



暂停三次 Stalling X3

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
       bubble
       bubble
       bubble
0x014: addq %rdx,%rax
0x016: halt
```



当暂停时发生了什么?
What Happens When Stalling?

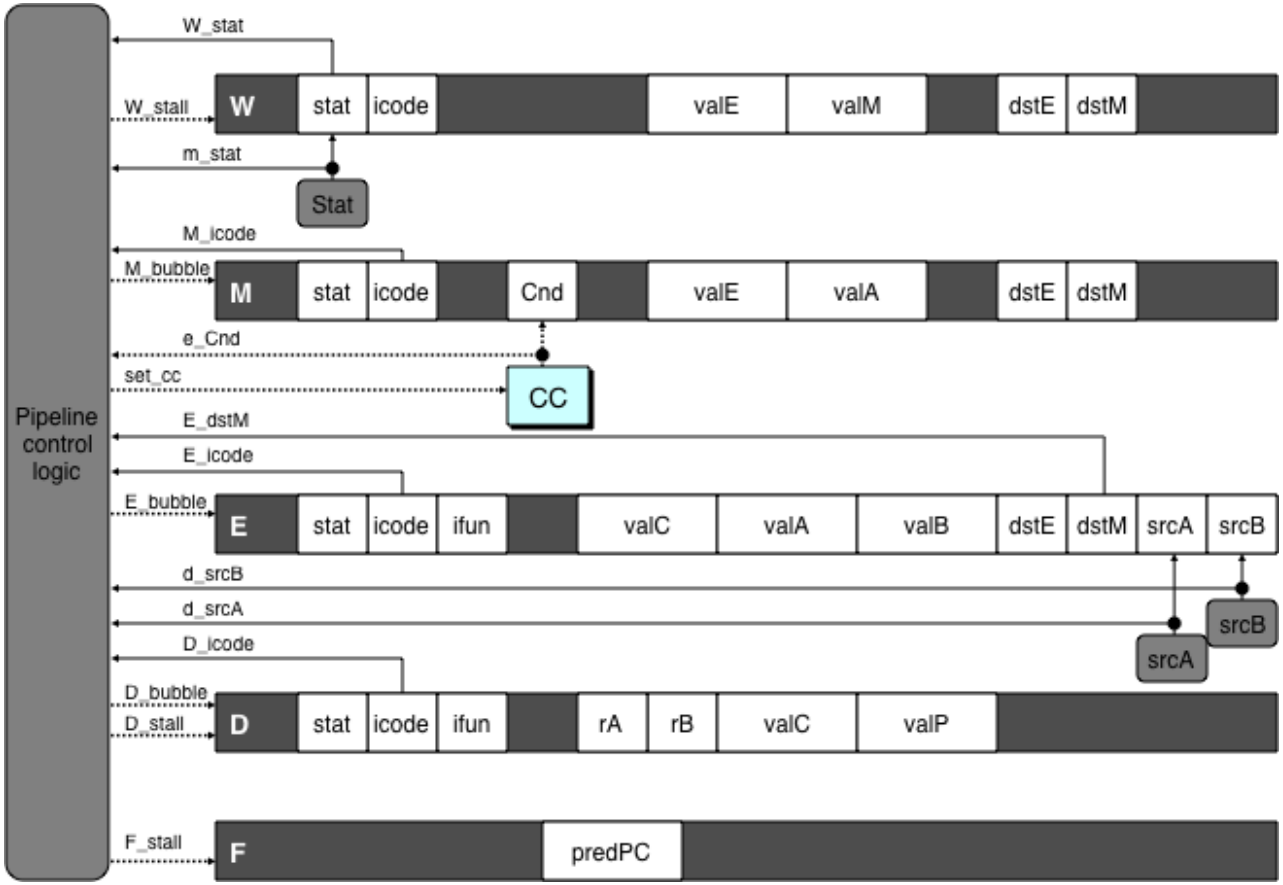
```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```

Cycle 8

Write Back	***bubble***
Memory	***bubble***
Execute	`0x014: addq %rdx,%rax`
Decode	`0x016: halt`
Fetch	

- 暂停指令保持在译码阶段 Stalling instruction held back in decode stage
- 后续指令停留在取指阶段 Following instruction stays in fetch stage
- 气泡注入到执行阶段 Bubbles injected into execute stage
  - 类似于动态产生空指令 Like dynamically generated nop's
  - 移动通过后面的阶段 Move through later stages

实现暂停 Implementing Stalling

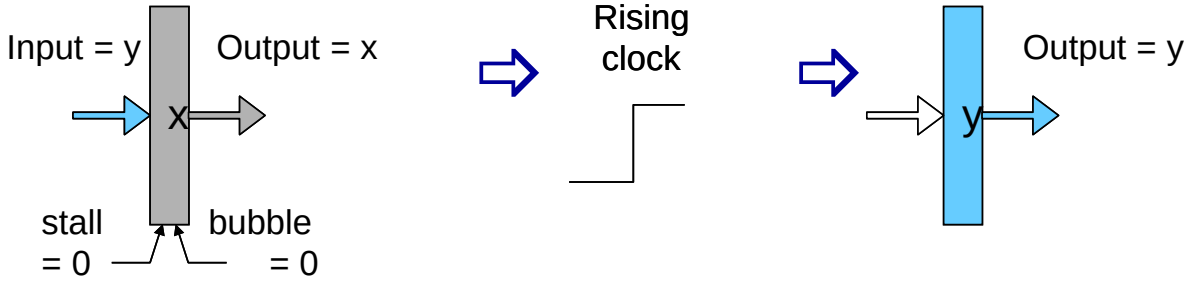


流水线控制 Pipeline Control

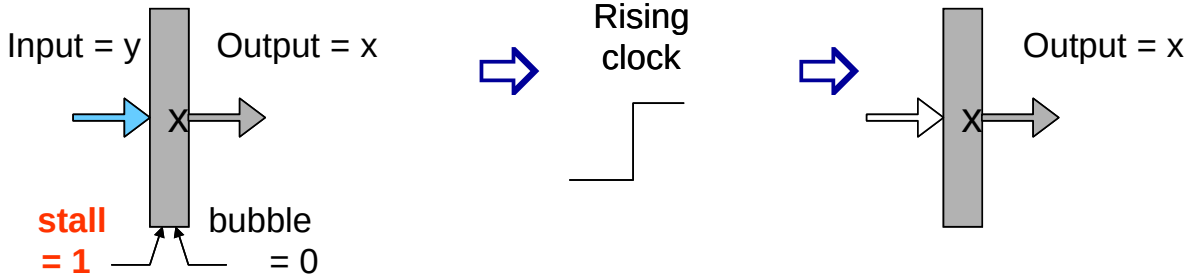
- 组合逻辑检测暂停条件 Combinational logic detects stall condition
- 设置模式信号指示流水线寄存器该如何更新 Sets mode signals for how pipeline registers should update

流水线寄存器模式
Pipeline Register Modes

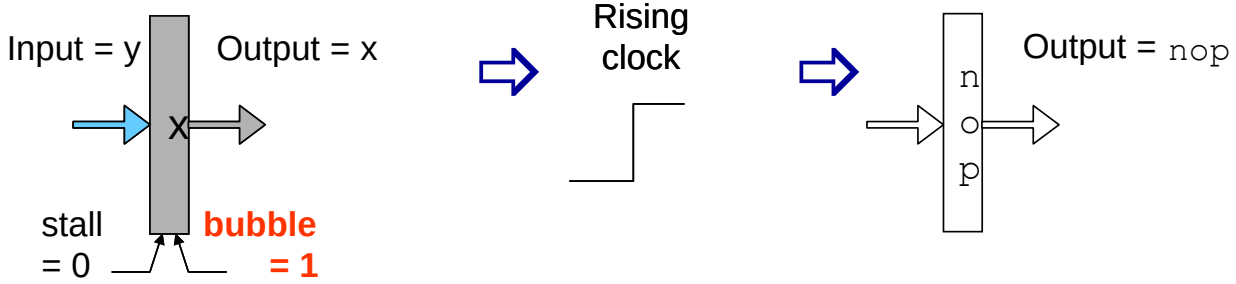
正常 Normal



暂停 Stall



气泡 Bubble



数据转发 Data Forwarding

朴素流水线 Naïve Pipeline

- 直到写回阶段完成才进行寄存器写操作 Register isn't written until completion of write-back stage
- 译码阶段就需要从寄存器文件(堆)读源操作数 Source operands read from register file in decode stage
  - 需要在阶段开始就在寄存器文件(堆)中才行 Needs to be in register file at start of stage

观察 Observation

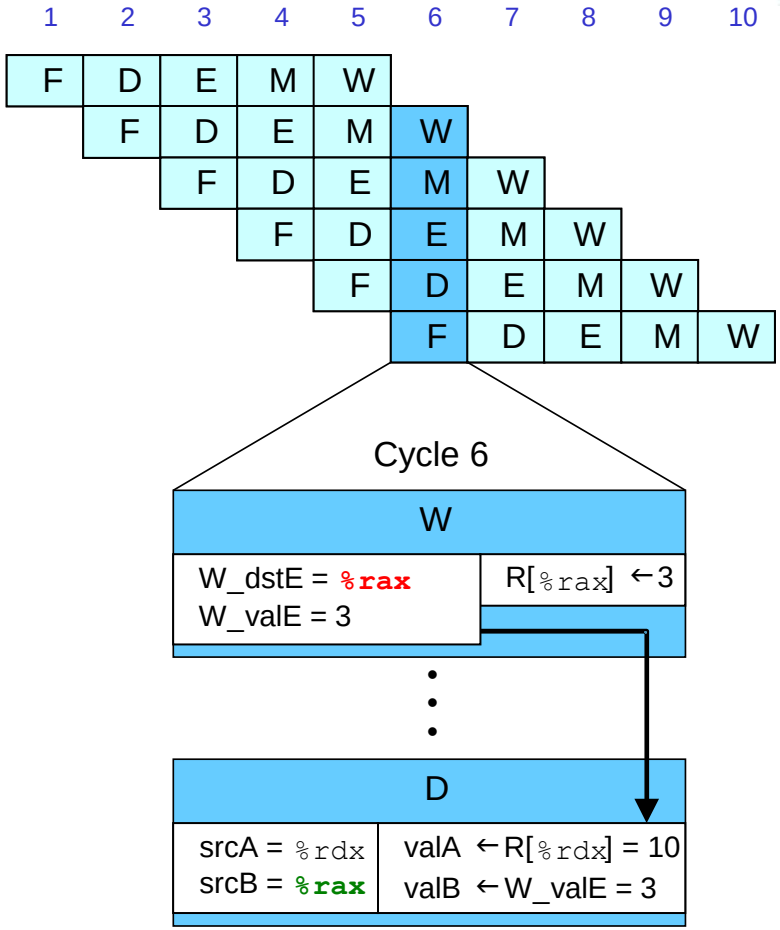
- 在执行或访存阶段产生值 Value generated in execute or memory stage

技巧 Trick

- 从生成指令处直接传递值到译码阶段 Pass value directly from generating instruction to decode stage
- 需要在译码阶段结束时可用 Needs to be available at end of decode stage

数据转发示例 Data Forwarding Example

```
# demo-h2.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: nop
0x015: nop
0x016: addq %rdx,%rax
0x018: halt
```

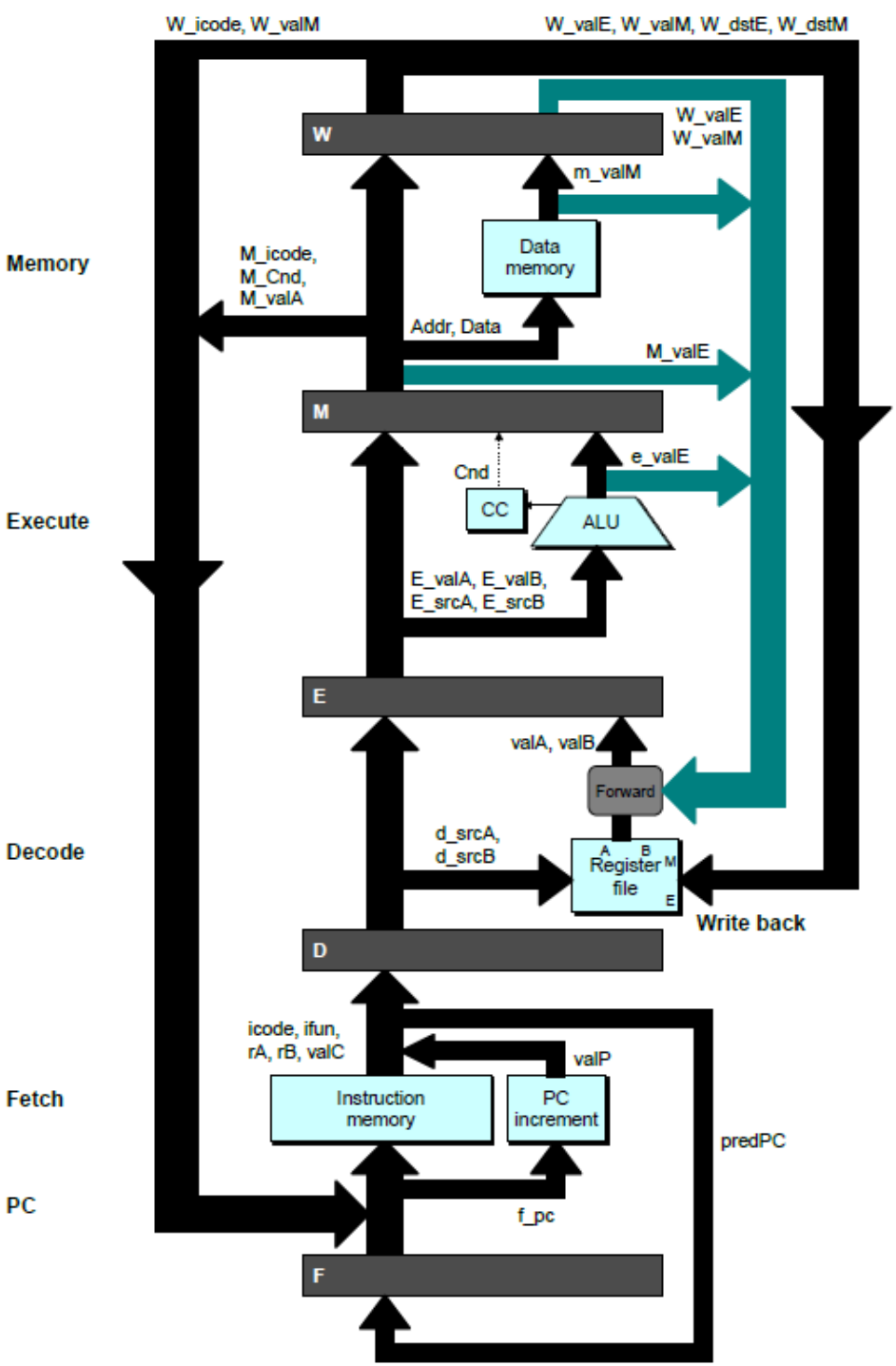


- irmovq在写回阶段 `irmovq` in write-back stage
- 目的值在W流水线寄存器中 Destination value in W pipeline register
- 转发为译码阶段的valB Forward as valB for decode stage

旁路路径 Bypass Paths

译码阶段 Decode Stage

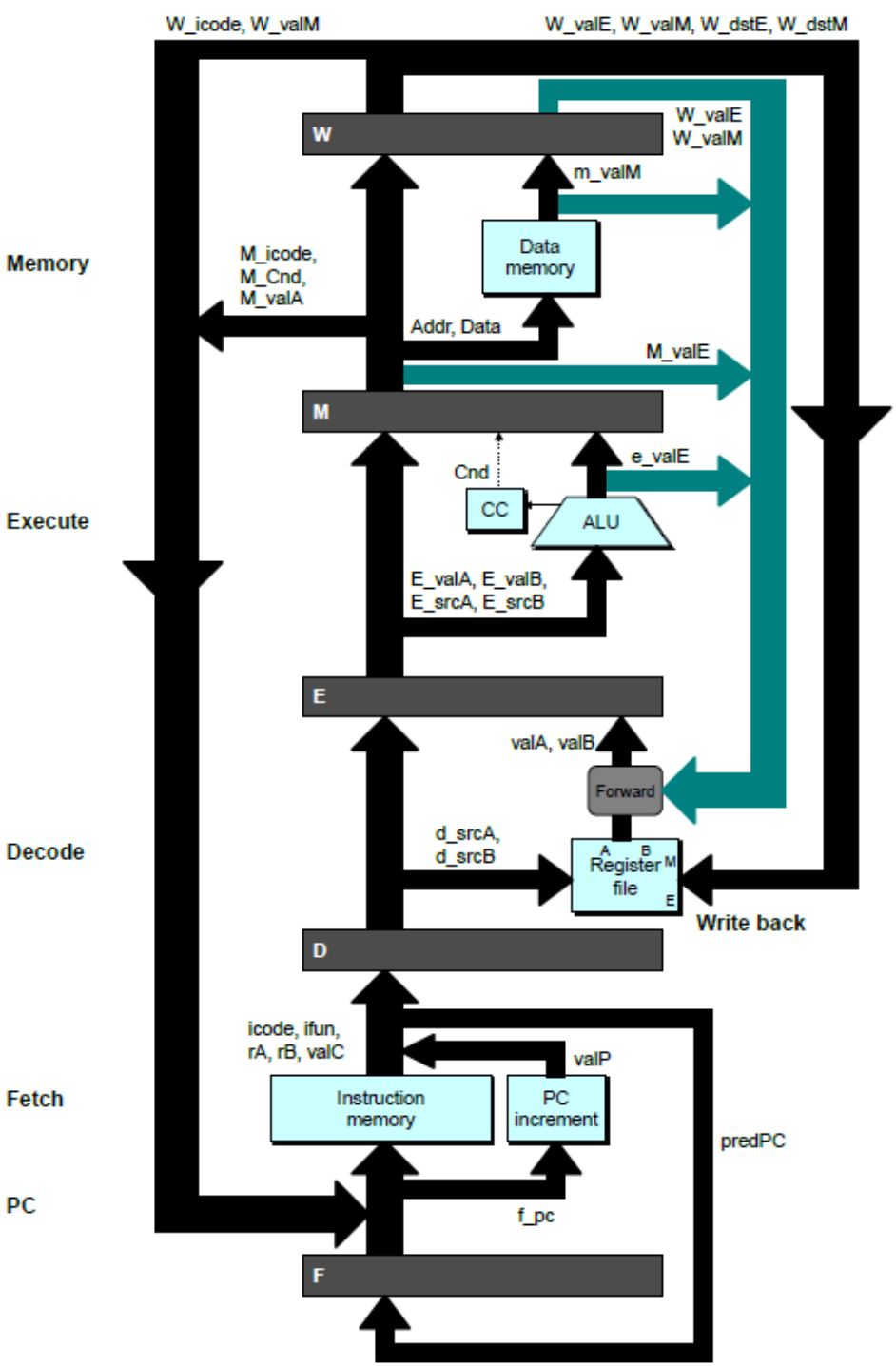
- 转发逻辑选择valA和valB
  Forwarding logic selects valA and valB
- 正常来自寄存器文件(堆)
  Normally from register file
- 转发：从后面流水线阶段得到valA或valB
  Forwarding: get valA or valB from later pipeline stage



旁路路径 Bypass Paths

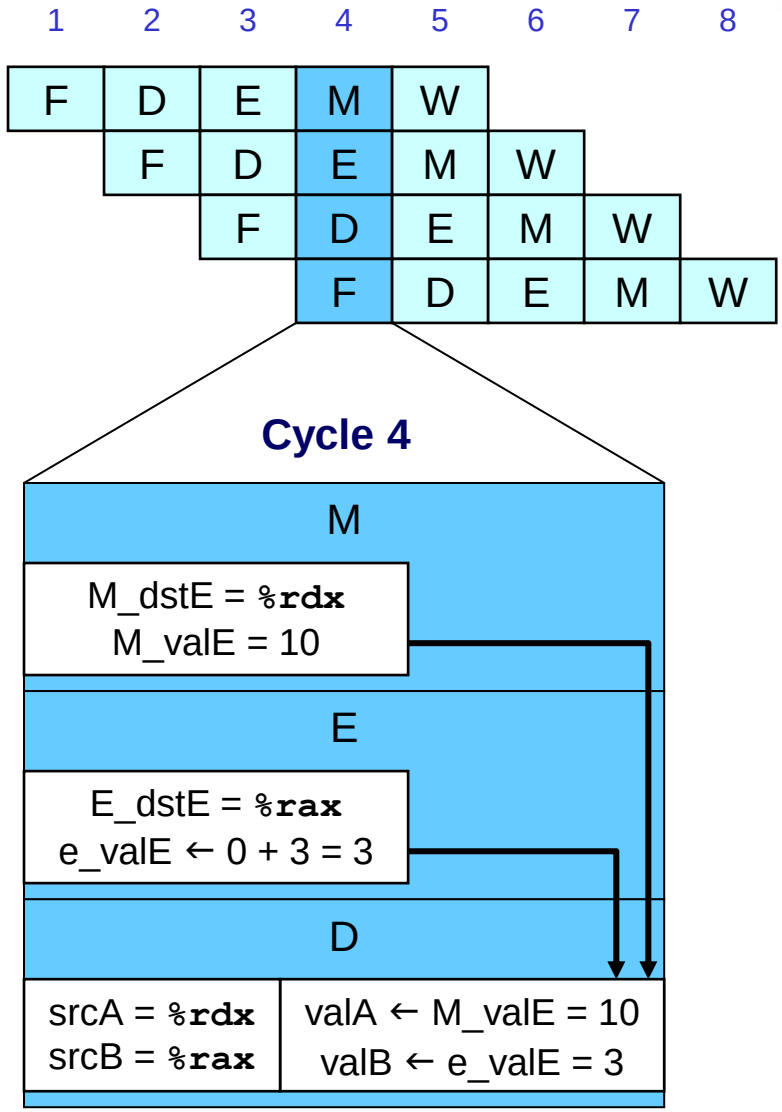
转发源 Forwarding Sources

- 执行阶段：valE Execute: valE
- 访存阶段：valE和valM Memory: valE, valM
- 写回阶段：valE和valM Write back: valE, valM



数据转发示例 Data Forwarding Example#2

```
# demo-h0.ys
0x000: irmovq $10,%rdx
0x00a: irmovq  $3,%rax
0x014: addq %rdx,%rax
0x016: halt
```



寄存器%rdx Register `%rdx`

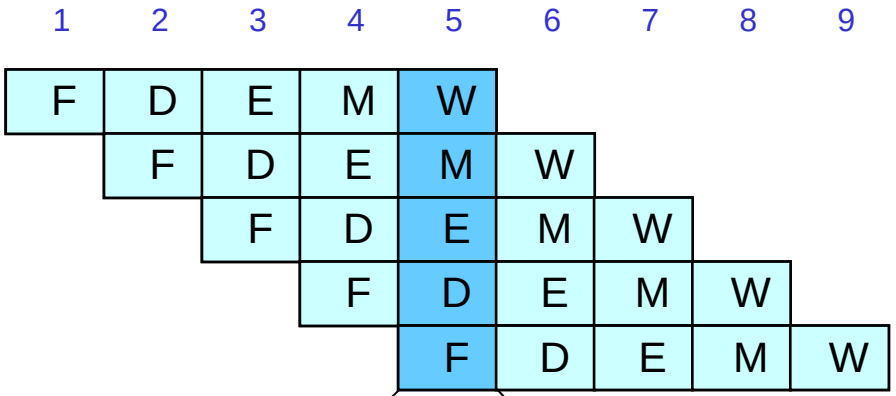
- 上个周期ALU产生
  Generated by ALU during previous cycle
- 从访存阶段转发作valA
  Forward from memory as valA

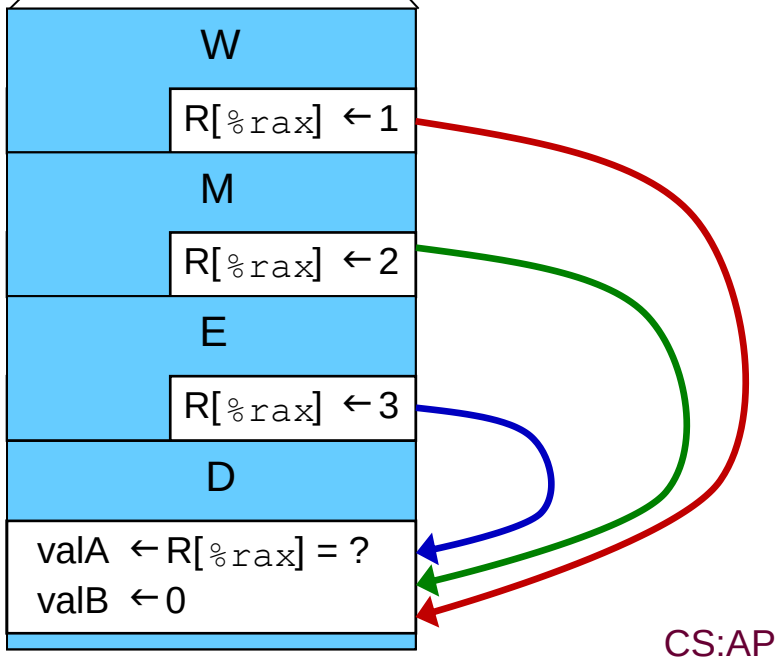
寄存器%rax Register `%rax`

- 刚由ALU产生的值 Value just generated by ALU
- 从执行阶段转发作valB
  Forward from execute as valB

转发优先级 Forwarding Priority

```
# demo-priority.ys
0x000: irmovq $1, %rax
0x00a: irmovq $2, %rax
0x014: irmovq $3, %rax
0x01e: rrmovq %rax, %rdx
0x020: halt
```





多种转发选择 Multiple Forwarding Choices

- 哪个应该有优先权 Which one should have priority
- 串行匹配语义 Match serial semantics
- 使用来自最早流水线阶段的匹配值 Use matching value from earliest pipeline stage

转发优先级HCL表达式

多个转发源

4.5.5 节中提到有 5 个不同的转发源，每个都有一个数据字和一个目的寄存器 ID：

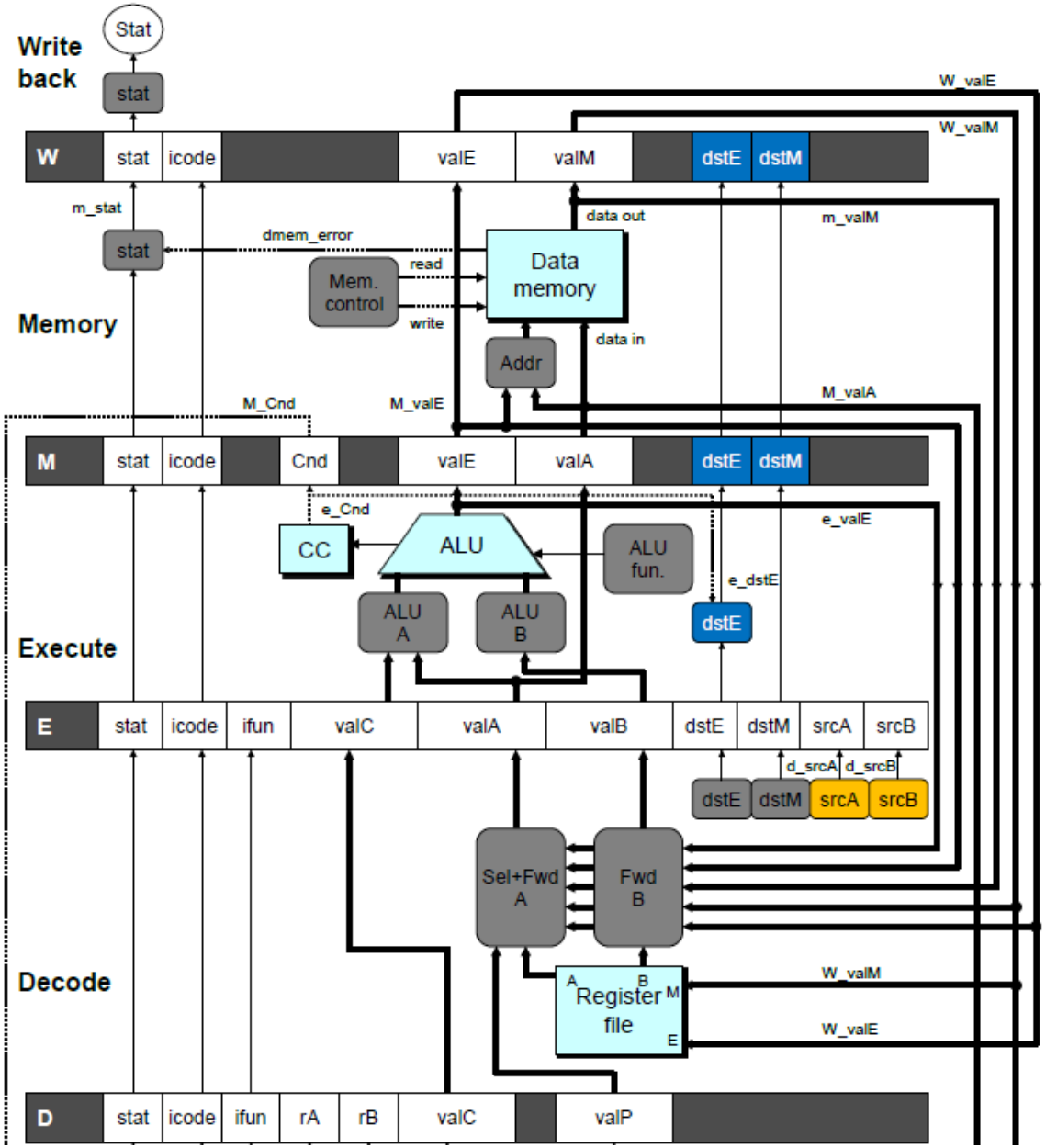
数据字	寄存器 ID	源描述
e_valE	e_dstE	ALU 输出
m_valM	M_dstM	内存输出
M_valE	M_dstE	访存阶段中对端口 E 未进行的写
W_valM	W_dstM	写回阶段中对端口 M 未进行的写
W_valE	W_dstE	写回阶段中对端口 E 未进行的写

如果不满足任何转发条件，这个块就应该选择 d_rvalA 作为它的输出，也就是从寄存器端口 A 中读出的值。

综上所述，我们得到以下流水线寄存器 E 的 valA 新值的 HCL 描述：

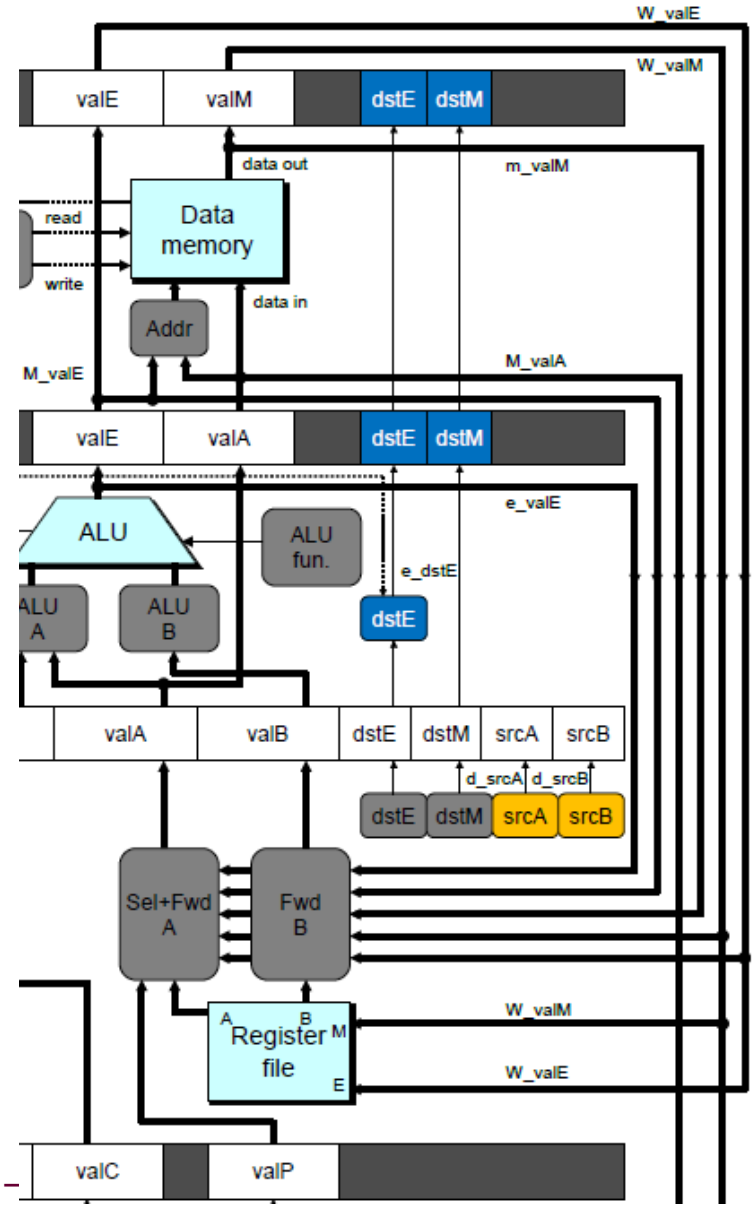
```
word d_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == e_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
        d_srcA == W_dstM : 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
];
```

实现转发 Implementing Forwarding



- 增加附加的反馈路径，从E、M和W流水线寄存器转发到译码阶段 Add additional feedback paths from E, M, and W pipeline registers into decode stage
- 创建逻辑块从多个源选择作译码阶段的valA和valB Create logic blocks to select from multiple sources for valA and valB in decode stage

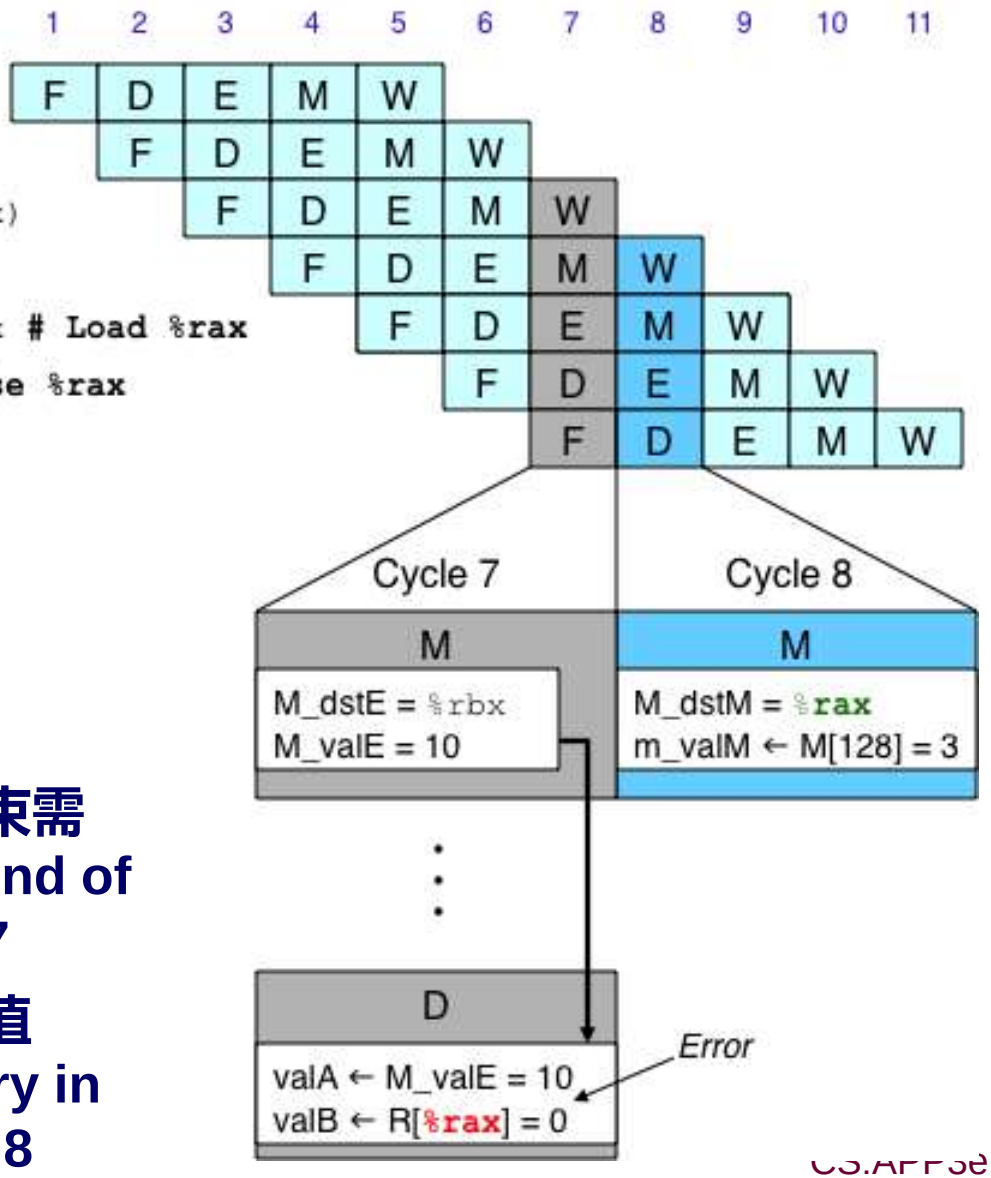
实现转发 Implementing Forwarding




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

转发的限制 Limitation of Forwarding

```
# demo-luh.ys
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
```

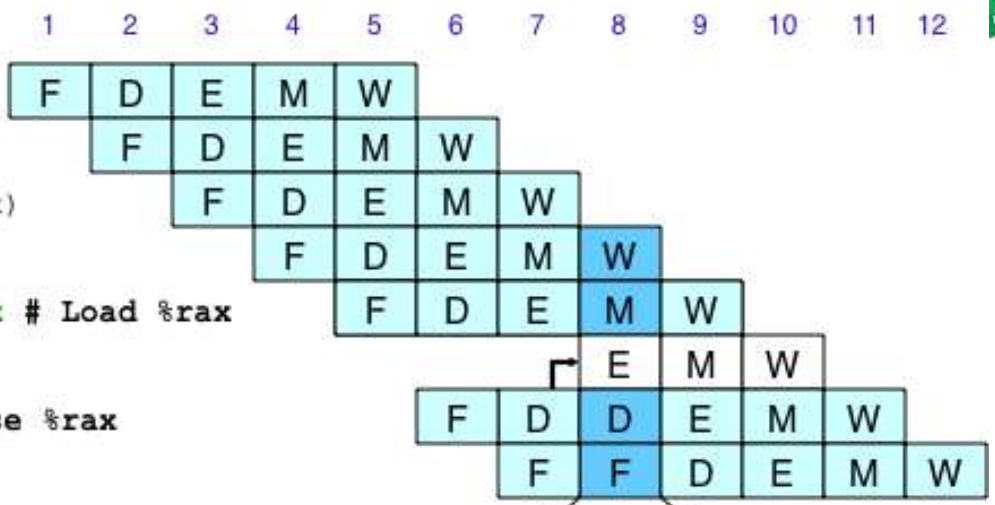


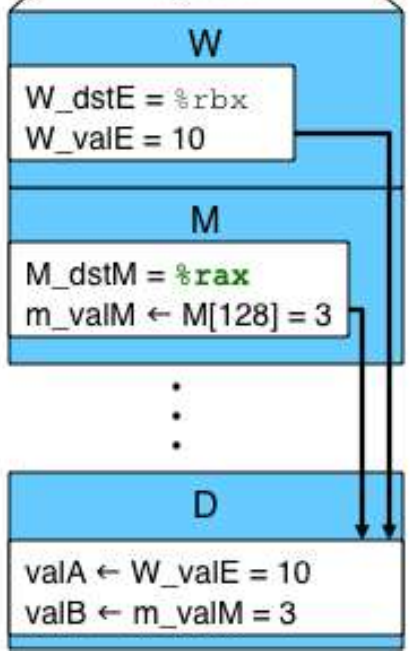
装载-使用相关 Load-use dependency

- 第7个周期译码阶段的结束需要值 Value needed by end of decode stage in cycle 7
- 第8个周期内存阶段读取值 Value read from memory in memory stage of cycle 8

避免装载/使用冒险 Avoiding Load/Use Hazard

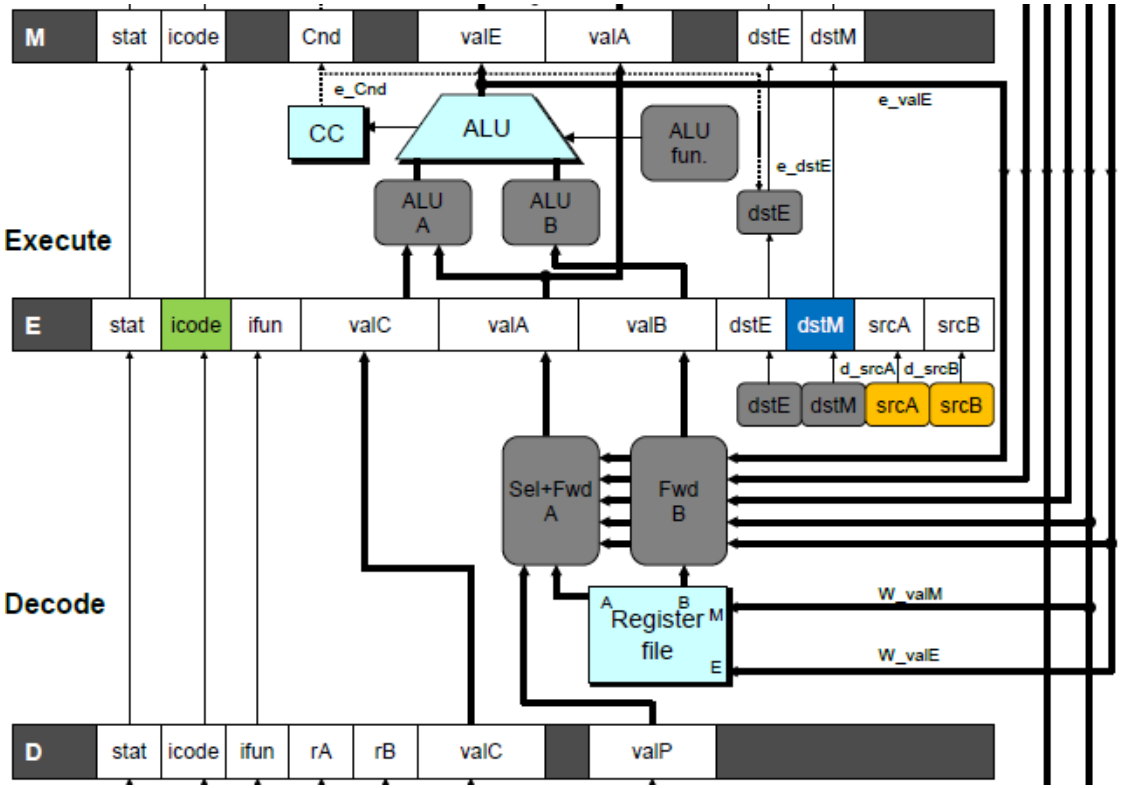
```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%rbx
0x028: mrmovq 0(%rdx),%rax # Load %rax
     bubble
0x032: addq %rbx,%rax # Use %rax
0x034: halt
```





- 暂停使用指令一个周期 Stall using instruction for one cycle
- 然后能够通过从内存阶段转发获取装载的值 Can then pick up loaded value by forwarding from memory stage

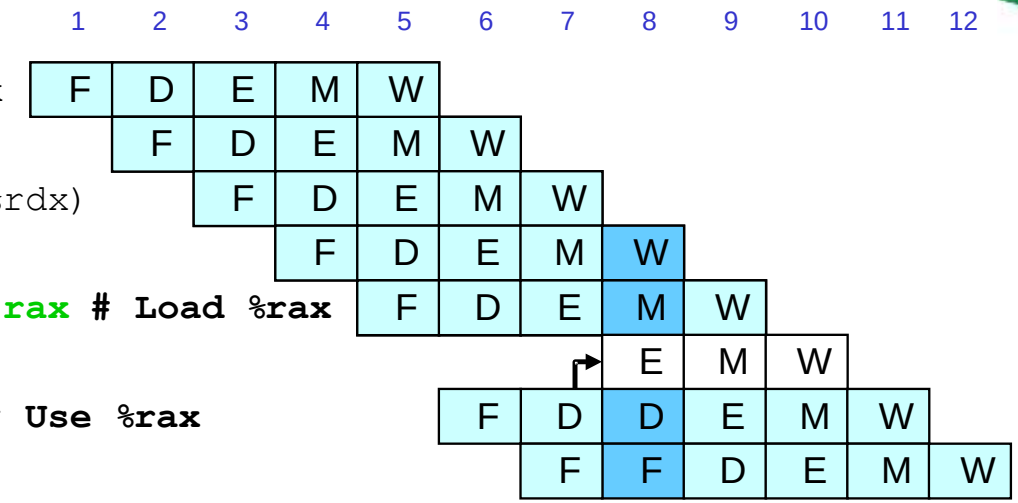
检测装载/使用冒险 Detecting Load/Use Hazard



状况 Condition	触发 Trigger
装载/使用冒险 Load/Use Hazard	`E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }`

装载/使用冒险控制 Control for Load/Use Hazard

```
# demo-luh.ys
0x000: irmovq $128,%rdx
0x00a: irmovq  $3,%rcx
0x014: rmmovq %rcx, 0(%rdx)
0x01e: irmovq $10,%ebx
0x028: mrmovq 0(%rdx),%rax # Load %rax
       bubble
0x032: addq %ebx,%rax # Use %rax
0x034: halt
```



- 暂停指令在取指和译码阶段 Stall instructions in fetch and decode stages
- 注入气泡到执行阶段 Inject bubble into execute stage

状况 Condition	F	D	E	M	W
装载/使用冒险 Load/Use Hazard	暂停 stall	暂停 stall	气泡 bubble	正常 normal	正常 normal

分支预测错误示例
Branch Misprediction Example

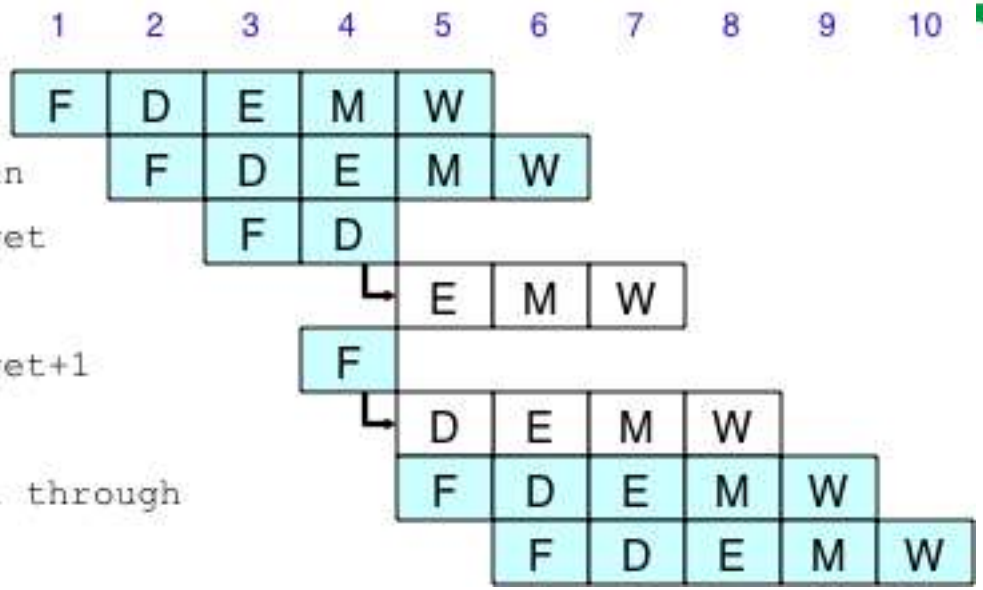
demo-j.ys

```
0x000:     xorq %rax,%rax
0x002:     jne  t                # Not taken
0x00b:     irmovq $1, %rax       # Fall through
0x015:     nop
0x016:     nop
0x017:     nop
0x018:     halt
0x019: t:  irmovq $3, %rdx       # Target
0x023:     irmovq $4, %rcx       # Should not execute
0x02d:     irmovq $5, %rdx       # Should not execute
```

- 应该仅仅执行前8条指令 Should only execute first 8 instructions

处理预测错误 Handling Misprediction

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



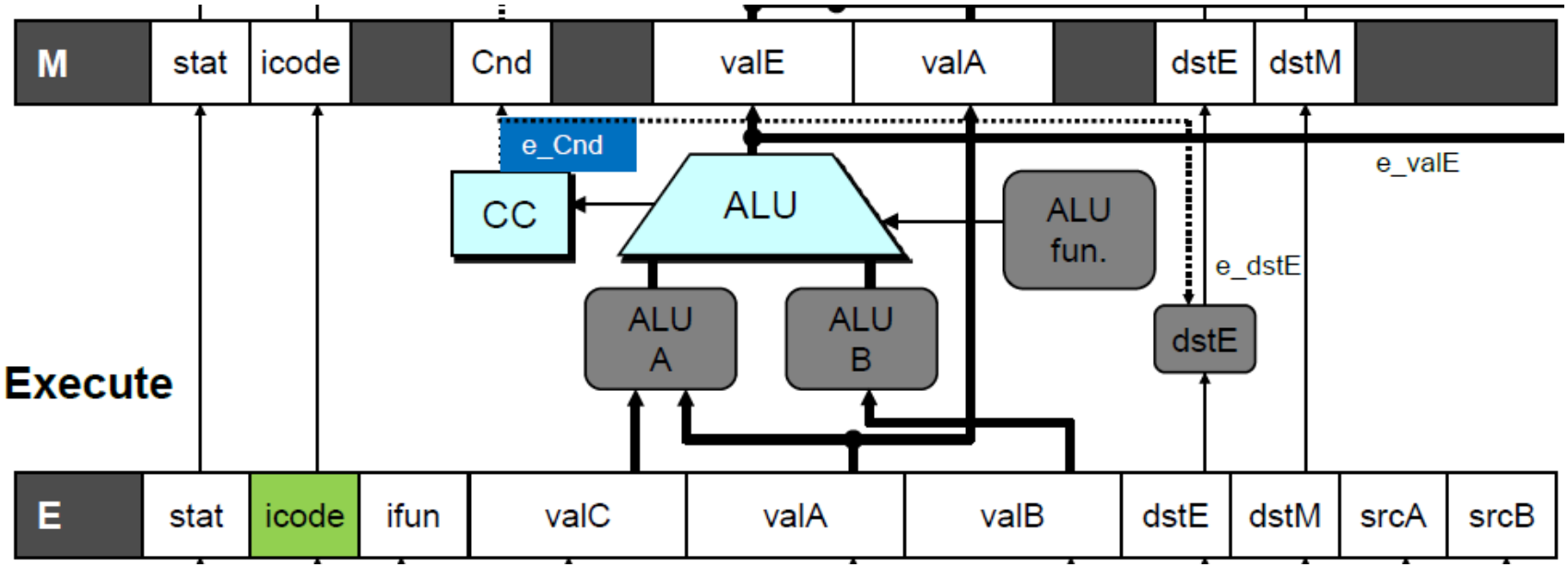
预测分支为选择分支 Predict branch as taken

- 取目标处2条指令 Fetch 2 instructions at target

当预测错误时取消 Cancel when mispredicted

- 在执行阶段检测出分支不选择 Detect branch not-taken in execute stage
- 在下一个周期，用气泡替代执行和译码阶段的指令 On following cycle, replace instructions in execute and decode by bubbles
- 还没有副作用发生 No side effects have occurred yet

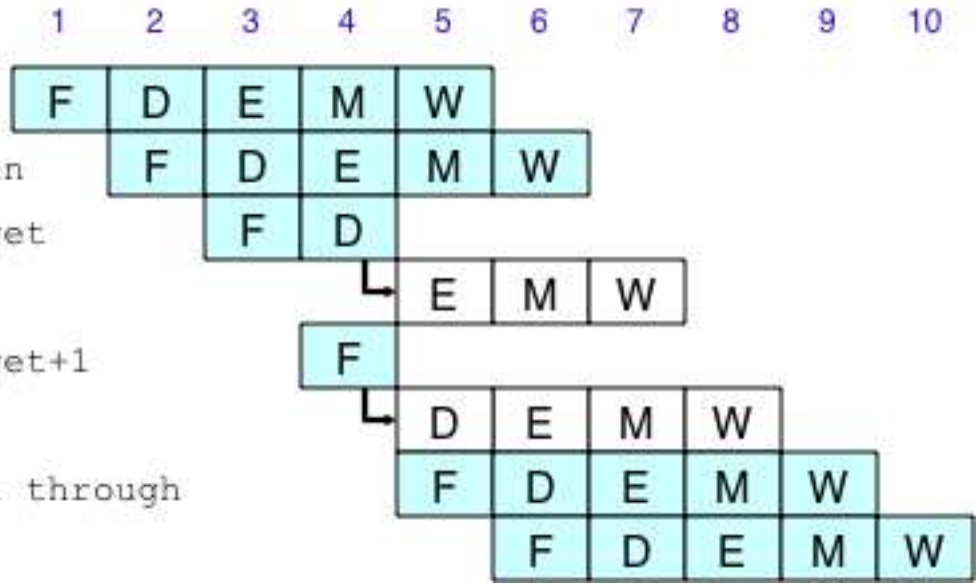
检测分支预测错误
Detecting Mispredicted Branch



状况 Condition	触发 Trigger
分支预测错误 Mispredicted Branch	`E_icode = IJXX & !e_Cnd`

预测错误控制 Control for Misprediction

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



状况 Condition	F	D	E	M	W
分支预测错误 Mispredicted Branch	正常 normal	气泡 bubble	气泡 bubble	正常 normal	正常 normal

返回示例 Return Example

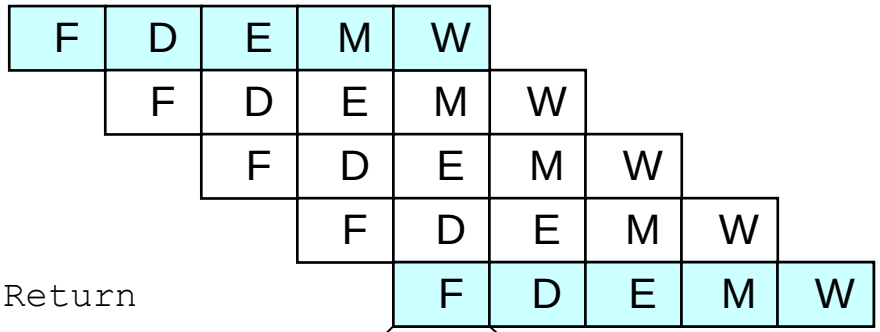
demo-retb.ys

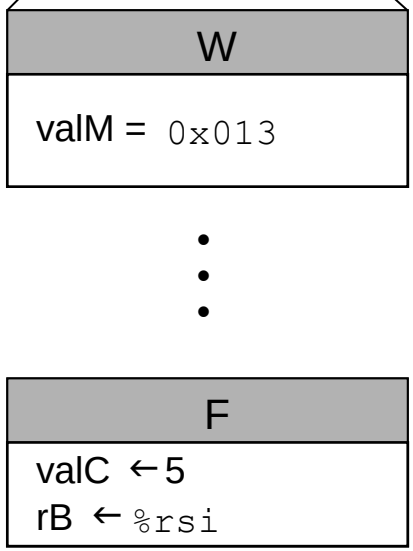
```
0x000:     irmovq Stack,%rsp  # Intialize stack pointer
0x00a:     call p             # Procedure call
0x013:     irmovq $5,%rsi     # Return point
0x01d:     halt
0x020:  .pos 0x20
0x020: p:  irmovq $-1,%rdi    # procedure
0x02a:     ret
0x02b:     irmovq $1,%rax     # Should not be executed
0x035:     irmovq $2,%rcx     # Should not be executed
0x03f:     irmovq $3,%rdx     # Should not be executed
0x049:     irmovq $4,%rbx     # Should not be executed
0x100:  .pos 0x100
0x100: Stack:                 # Stack: Stack pointer
```

- 以前执行三条附加的指令 Previously executed three additional instructions

正确返回示例 Correct Return Example

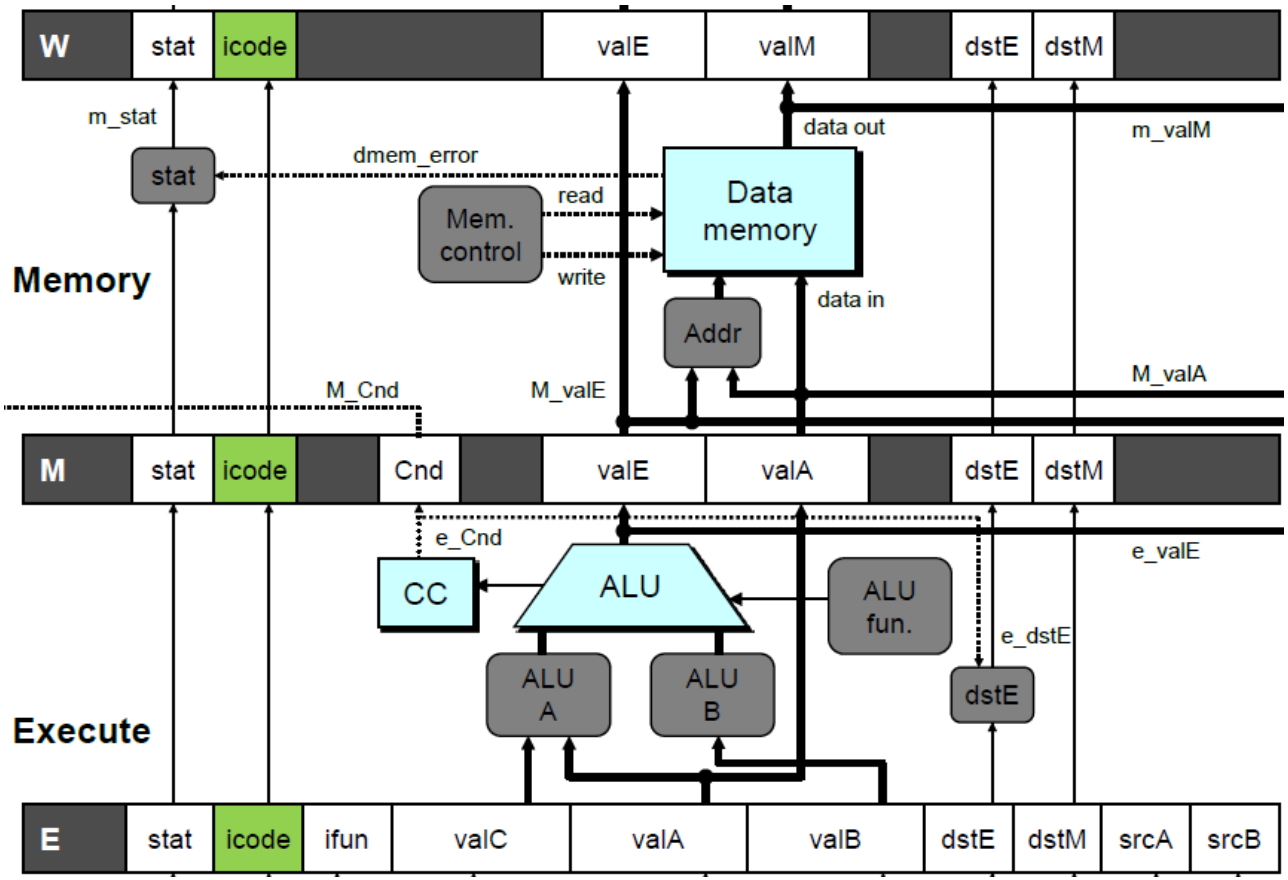
```
# demo-retb
0x026:      ret
            bubble
            bubble
            bubble
0x013:      irmovq $5,%rsi # Return
```





- 当ret通过流水线时，暂停取指阶段 As `ret` passes through pipeline, stall at fetch stage
  - 同时ret流过译码、执行和内存阶段 While in decode, execute, and memory stage
- 注入气泡到译码阶段 Inject bubble into decode stage
- 当到达写回阶段时释放暂停 Release stall when reach write-back stage

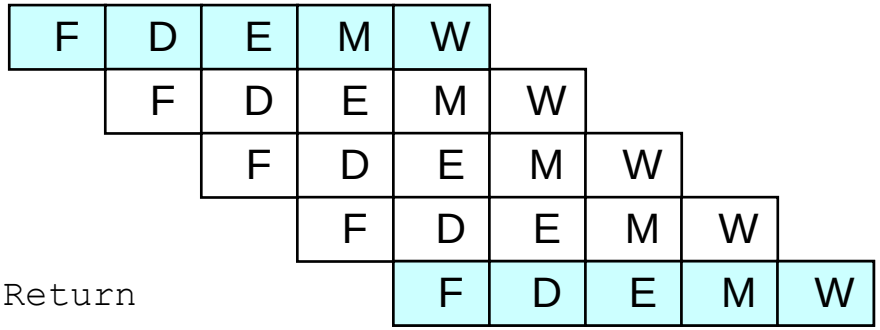
检测返回 Detecting Return



状况 Condition	触发 Trigger
处理ret Processing `ret`	`IRET in { D_icode, E_icode, M_icode }`

返回控制 Control for Return

```
# demo-retb
0x026:      ret
            bubble
            bubble
            bubble
0x014:      irmovq $5,%rsi # Return
```



状况 Condition	F	D	E	M	W
处理ret Processing `ret`	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal

特殊控制情况 Special Control Cases

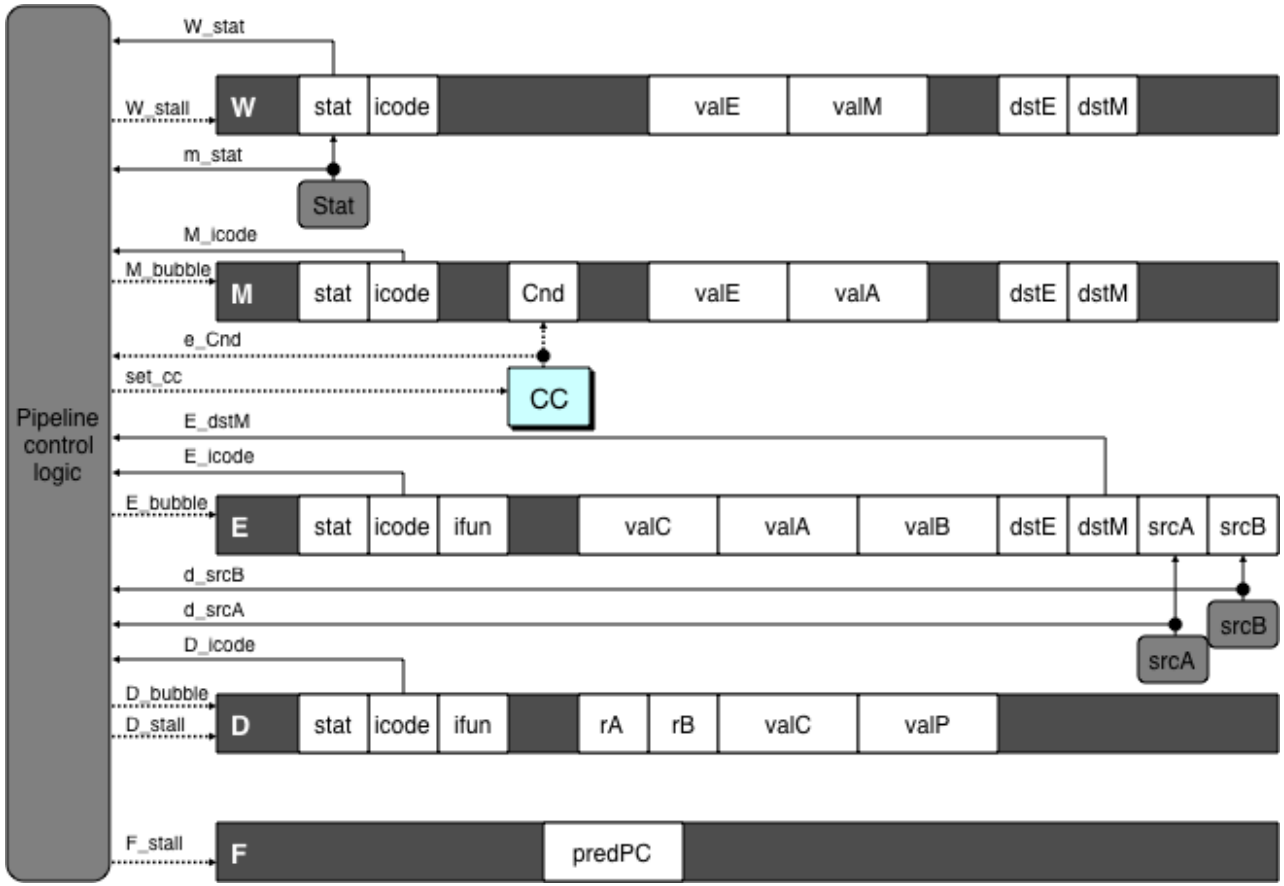
检测 Detection

状况 Condition	触发 Trigger
处理ret Processing `ret`	IRET in { D_icode, E_icode, M_icode }
装载/使用冒险 Load/Use Hazard	E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }
分支预测错误 Mispredicted Branch	E_icode = IJXX & !e_Cnd

动作（下一个周期） Action (on next cycle)

Condition	F	D	E	M	W
处理ret Processing `ret`	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal
装载/使用冒险 Load/Use Hazard	暂停 stall	暂停 stall	气泡 bubble	正常 normal	正常 normal
分支预测错误 Mispredicted Branch	正常 normal	气泡 bubble	气泡 bubble	正常 normal	正常 normal

实现流水线控制
Implementing Pipeline Control

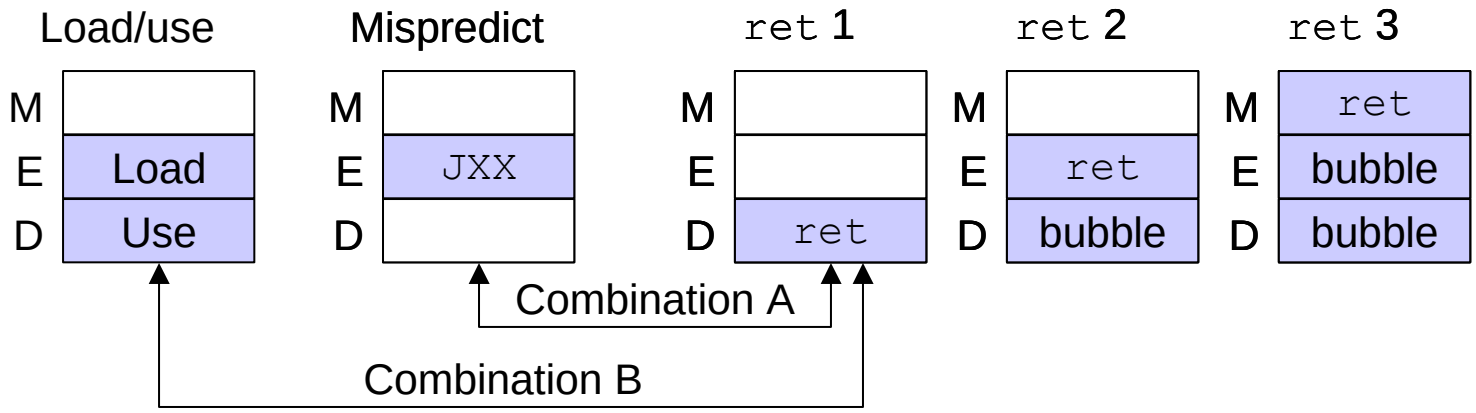


- 组合逻辑产生流水线控制信号 Combinational logic generates pipeline control signals
- 动作发生在下一个周期的开始 Action occurs at start of following cycle

流水线控制的初始版本
Initial Version of Pipeline Control

```
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_stall =
    # Conditions for a load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };

bool D_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Stalling at fetch while ret passes through pipeline
     IRET in { D_icode, E_icode, M_icode };

bool E_bubble =
    # Mispredicted branch
    (E_icode == IJXX && !e_Cnd) ||
    # Load/use hazard
    E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB };
```

控制组合 Control Combinations



- 在同一个时钟周期可能引起的特殊情况 Special cases that can arise on same clock cycle

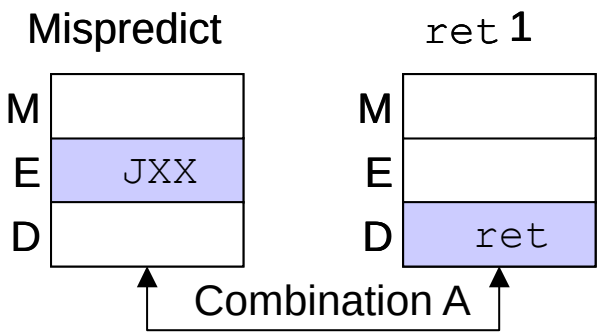
组合A Combination A

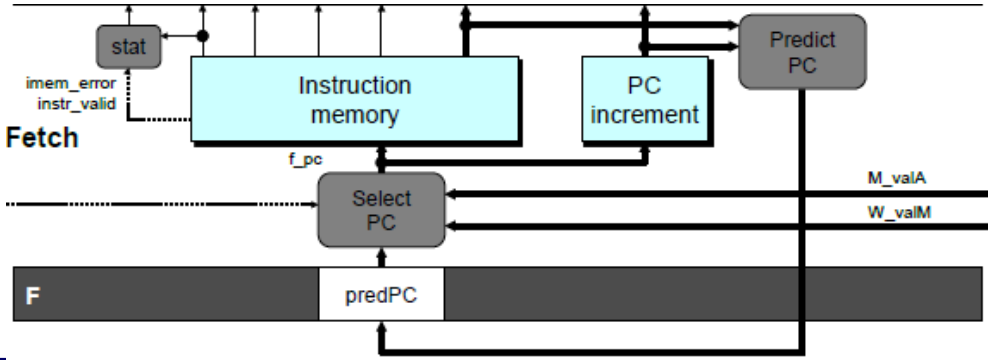
- 分支不选择 Not-taken branch
- Ret指令在分支目标处 `ret` instruction at branch target

组合B Combination B

- 指令从内存读到%rsp Instruction that reads from memory to `%rsp`
- 后跟ret指令 Followed by `ret` instruction

控制组合A Control Combination A

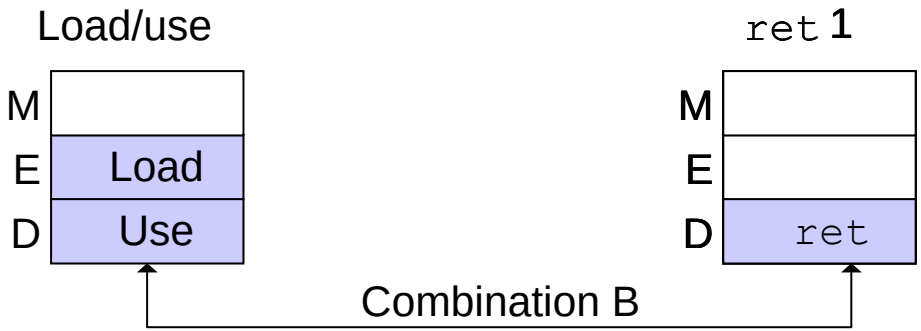




状况 Condition	F	D	E	M	W
处理ret Processing ret	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal
分支预测错误 Mispredicted Branch	正常 normal	气泡 bubble	气泡 bubble	正常 normal	正常 normal
组合 Combination	*暂停* *stall*	*气泡* *bubble*	*气泡* *bubble*	*正常* *normal*	*正常* *normal*

- 当分支预测错误时应该处理 Should handle as mispredicted branch
- 暂停F流水线寄存器 Stalls F pipeline register
- 但PC选择逻辑无论如何使用M_valM But PC selection logic will be using M_valM anyhow

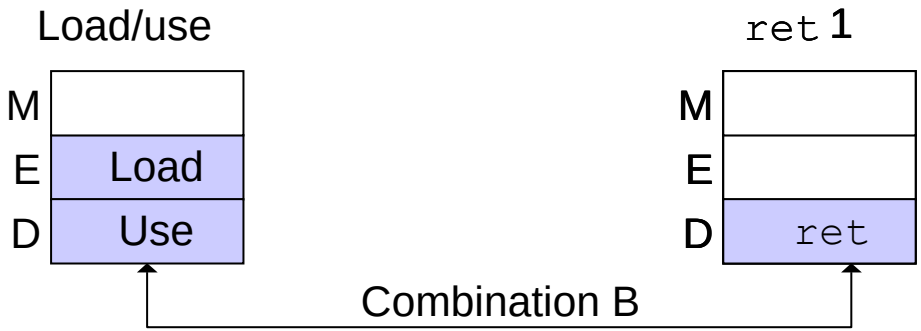
控制组合B Control Combination B



状况 Condition	F	D	E	M	W
处理ret Processing ret	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal
装载/使用冒险 Load/Use Hazard	暂停 stall	暂停 stall	气泡 bubble	正常 normal	正常 normal
组合Combination	*暂停* *stall*	*气泡+暂停* *bubble + stall*	*气泡* *bubble*	*正常* *normal*	*正常* *normal*

- 尝试注入气泡和暂停流水线寄存器D Would attempt to bubble *and* stall pipeline register D
- 处理器信号指示流水线错误 Signaled by processor as pipeline error

处理控制组合B Handling Control Combination B



状况 Condition	F	D	E	M	W
处理ret Processing ret	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal
装载/使用冒险 Load/Use Hazard	暂停 stall	暂停 stall	气泡 bubble	正常 normal	正常 normal
组合 Combination	*暂停* *stall*	*暂停* *stall*	*气泡* *bubble*	*正常* *normal*	*正常* *normal*

- 装载/使用冒险应该得到优先权 Load/use hazard should get priority
- ret指令保持在译码阶段一段附加周期 `ret` instruction should be held in decode stage for additional cycle

修正流水线控制逻辑
Corrected Pipeline Control Logic

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

状况 Condition	F	D	E	M	W
处理ret Processing ret	暂停 stall	气泡 bubble	正常 normal	正常 normal	正常 normal
装载/使用冒险 Load/Use Hazard	暂停 stall	暂停 stall	气泡 bubble	正常 normal	正常 normal
组合 Combination	*暂停* *stall*	*暂停* *stall*	*气泡* *bubble*	*正常* *normal*	*正常* *normal*

- 装载/使用冒险应该得到优先权 Load/use hazard should get priority
- ret指令保持在译码阶段一段附加周期 `ret` instruction should be held in decode stage for additional cycle

流水线小结 Pipeline Summary

数据冒险 Data Hazards

- 大多通过转发进行处理 Most handled by forwarding
  - 没有性能损失 No performance penalty
- 装载/使用冒险需要暂停一个周期 Load/use hazard requires one cycle stall

控制冒险 Control Hazards

- 当检测到分支预测错误时取消指令 Cancel instructions when detect mispredicted branch
  - 浪费两个时钟周期 Two clock cycles wasted
- 当ret通过流水线时暂停取指阶段 Stall fetch stage while `ret` passes through pipeline
  - 浪费三个时钟周期 Three clock cycles wasted

控制组合 Control Combinations

- 必须仔细分析 Must analyze carefully
- 第一个版本有细微的错误 First version had subtle bug
  - 仅在不寻常的指令组合中出现 Only arises with unusual instruction combination

CS:APP Chapter 4 Computer Architecture

Wrap-Up
结语

任课教师：
宿红毅　张艳　黎有琦　颜珂

原作者：
Randal E. **Bryant and** David R. O'Hallaron

Carnegie Mellon University

概述 Overview

流水线设计结语 Wrap-Up of PIPE Design

- 异常情况 Exceptional conditions
- 性能分析 Performance analysis
- 取指阶段设计 Fetch stage design

现代高性能处理器 Modern High-Performance Processors

- 乱序执行 Out-of-order execution

异常 Exceptions

- 在这种情景下处理器不能继续正常操作 Conditions under which processor cannot continue normal operation

原因 Causes

- 停机指令 Halt instruction (Current)
- 指令或数据地址不正确 Bad address for instruction or data (Previous)
- 不合法的指令 Invalid instruction (Previous)

典型期望的动作 Typical Desired Action

- 完成某些指令 Complete some instructions
  - 要么是当前要么是前面指令（取决于异常类型） Either current or previous (depends on exception type)
- 废弃其它指令 Discard others
- 调用异常处理程序 Call exception handler
  - 类似非预期的过程调用 Like an unexpected procedure call

我们的实现 Our Implementation

- 当指令引起异常时停机 Halt when instruction causes exception

异常示例 Exception Examples

在取指阶段检测到异常 Detect in Fetch Stage

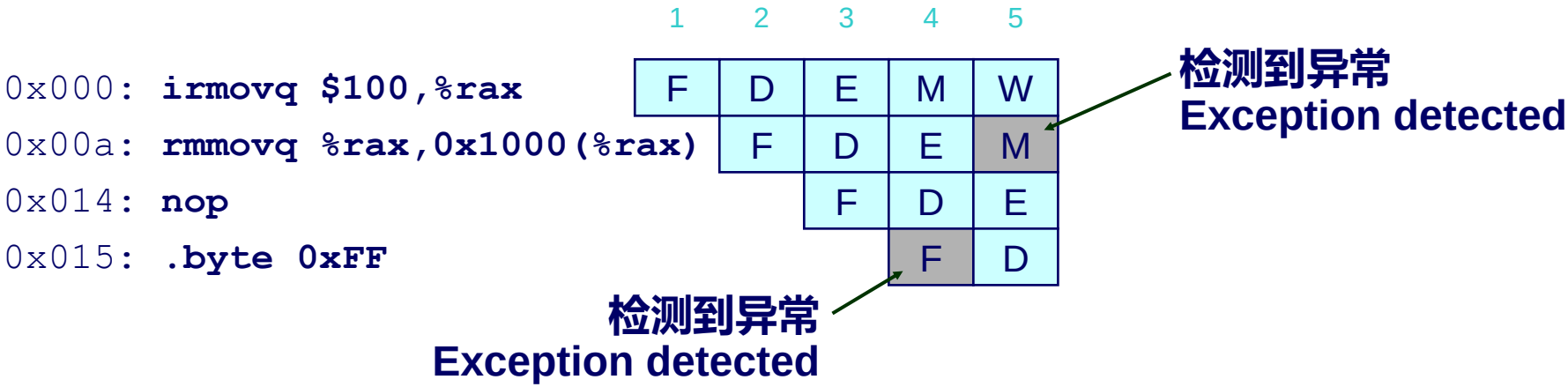
```
jmp $-1                    # Invalid jump target

.byte 0xFF                 # Invalid instruction code

halt                       # Halt instruction
```

在访存阶段检测到异常 Detect in Memory Stage

```
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # invalid address
```

流水线处理器中的异常#1
Exceptions in Pipeline Processor #1

```
# demo-exc1.ys
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # Invalid address
nop
.byte 0xFF                # Invalid instruction code
```



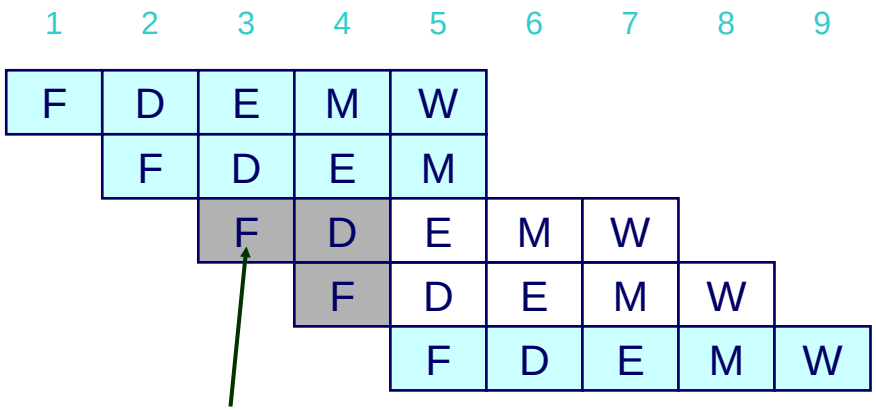
期望的行为 Desired Behavior

- rmmovq应该引起异常 rmmovq should cause exception
- 后续指令应该对处理器状态没有影响 Following instructions should have no effect on processor state

流水线处理器中的异常#2
Exceptions in Pipeline Processor #2

```
# demo-exc2.ys
0x000:     xorq %rax,%rax    # Set condition codes
0x002:     jne t             # Not taken
0x00b:     irmovq $1,%rax
0x015:     irmovq $2,%rdx
0x01f:     halt
0x020: t:  .byte 0xFF        # Target
```

```
0x000:     xorq %rax,%rax
0x002:     jne t
0x020: t:  .byte 0xFF
0x???: (I'm lost!)
0x00b:     irmovq $1,%rax
```

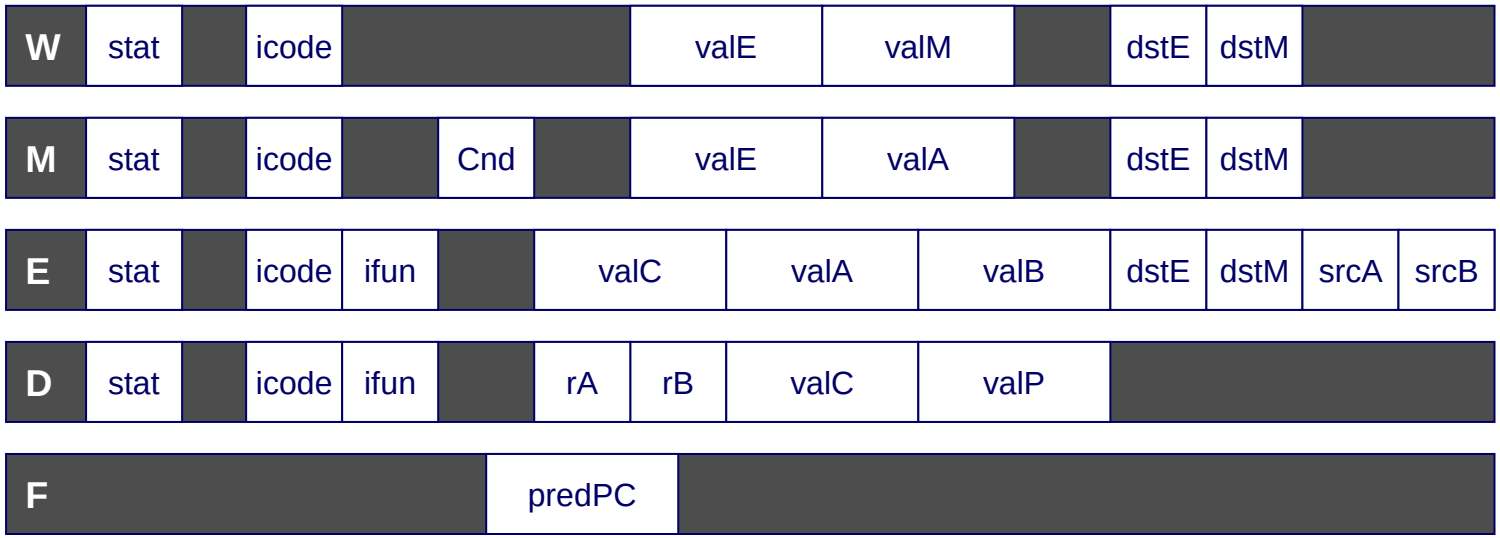


检测到异常 Exception detected

期望的行为 Desired Behavior

- 不应该发生异常 No exception should occur

维护异常排序 Maintaining Exception Ordering



- 流水线寄存器增加状态字段 Add status field to pipeline registers
- 取指阶段设置“AOK”、“ADR”（当取指地址错），“HLT”（停机指令）或“INS”（非法指令）状态之一 Fetch stage sets to either “AOK,” “ADR” (when bad fetch address), “HLT” (halt instruction) or “INS” (illegal instruction)
- 译码和执行阶段直接传递状态值 Decode & execute pass values through
- 访存阶段直接传递或设置成“ADR” Memory either passes through or sets to “ADR”
- 仅当指令到达写回阶段时触发异常 Exception triggered only when instruction hits write back

异常处理逻辑
Exception Handling Logic

取指阶段 Fetch Stage

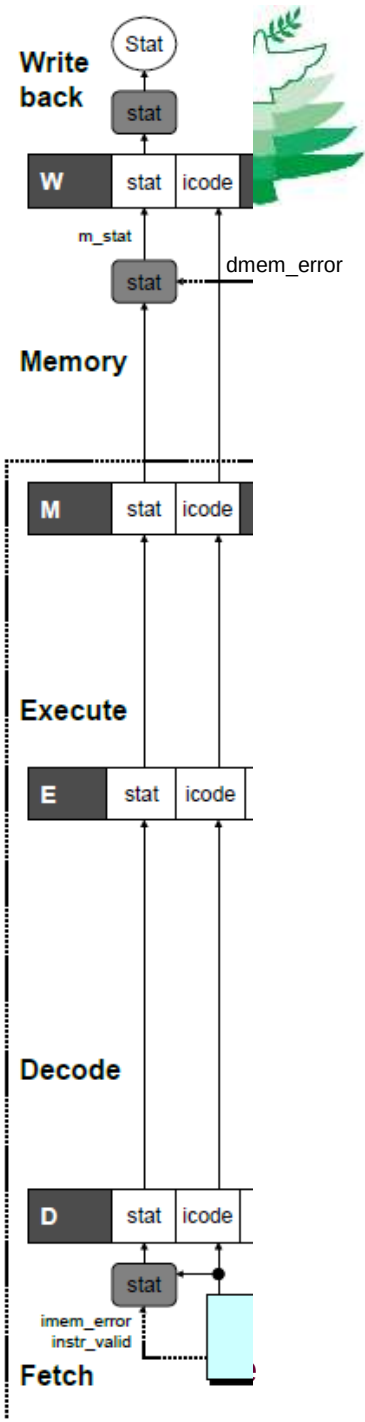
```
# Determine status code for fetched instruction
int f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
        1 : SAOK;
];
```

访存阶段 Memory Stage

```
# Update the status
int m_stat = [
        dmem_error : SADR;
        1 : M_stat;
];
```

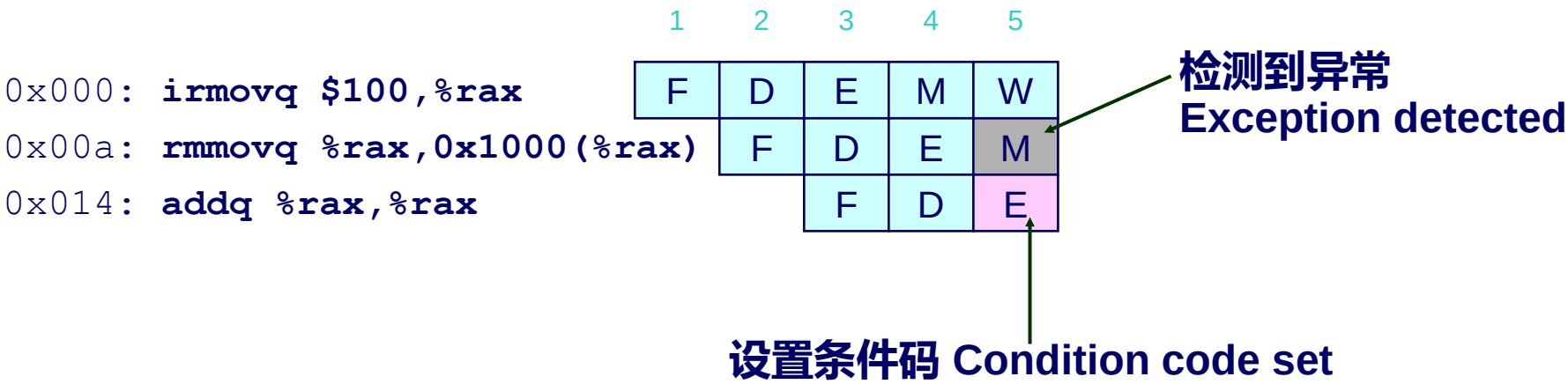
写回阶段 Writeback Stage

```
int Stat = [
        # SBUB in earlier stages indicates bubble
        W_stat == SBUB : SAOK;
        1 : W_stat;
];
```



流水线处理器中的副作用
Side Effects in Pipeline Processor

```
# demo-exc3.ys
irmovq $100,%rax
rmmovq %rax,0x10000(%rax) # invalid address
addq %rax,%rax            # Sets condition codes
```



期望的行为 Desired Behavior

- rmmovq应该引起异常 `rmmovq` should cause exception
- 后续指令应该不会有任何影响 No following instruction should have any effect

避免副作用 Avoiding Side Effects

出现异常应该取消状态更新 Presence of Exception Should Disable State Update

- 不合法指令转换成流水线气泡 Invalid instructions are converted to pipeline bubbles
  - 除非有状态指明异常状态 Except have stat indicating exception status
- 数据内存不会写到非法地址 Data memory will not write to invalid address
- 防止非法更新条件码 Prevent invalid update of condition codes
  - 在访存阶段检测异常 Detect exception in memory stage
  - 在执行阶段取消条件码 Disable condition code setting in execute
  - 必须在同一个时钟周期发生 Must happen in same clock cycle
- 在最后阶段处理异常 Handling exception in final stages
  - 当在访存阶段检测到异常 When detect exception in memory stage
    - » 下一个周期开始注入气泡到内存阶段 Start injecting bubbles into memory stage on next cycle
  - 当在写回阶段检测到异常 When detect exception in write-back stage
    - » 暂停异常指令 Stall excepting instruction
- 包括在HCL代码中 Included in HCL code

状态改变的控制逻辑
Control Logic for State Changes

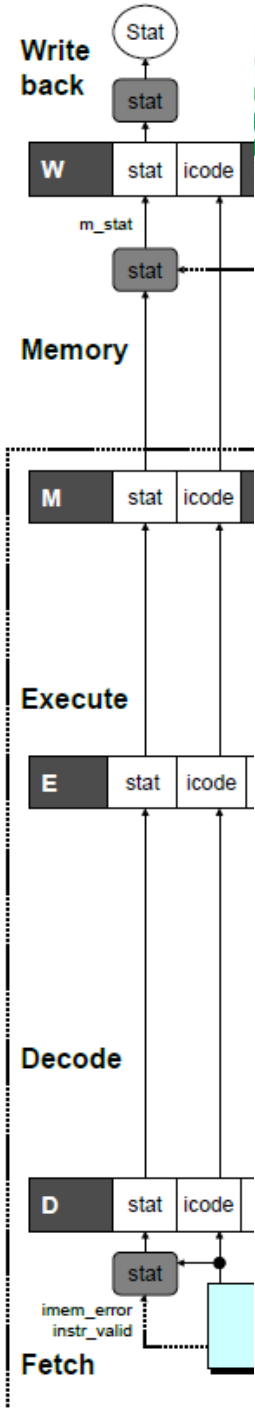
设置条件码 Setting Condition Codes

```
# Should the condition codes be updated?
bool set_cc = E_icode == IOPQ &&
        # State changes only during normal operation
        !m_stat in { SADR, SINS, SHLT }
        && !W_stat in { SADR, SINS, SHLT };
```

阶段控制 Stage Control

- 也控制内存更新 Also controls updating of memory

```
# Start injecting bubbles as soon as exception passes
through memory stage
bool M_bubble = m_stat in { SADR, SINS, SHLT }
        || W_stat in { SADR, SINS, SHLT };

# Stall pipeline register W when exception encountered
bool W_stall = W_stat in { SADR, SINS, SHLT };
```



现实生活中的其他异常处理 Rest of Real-Life Exception Handling

调用异常处理程序 Call Exception Handler

- PC压入栈 Push PC onto stack
  - 故障指令或下一条指令的PC Either PC of faulting instruction or of next instruction
  - 通常随着异常状态直接在流水线传递 Usually pass through pipeline along with exception status
- 跳转到处理程序地址 Jump to handler address
  - 通常是固定地址 Usually fixed address
  - 定义为ISA的一部分 Defined as part of ISA

实现 Implementation

- 还没有实现 Haven't tried it yet!

性能度量 Performance Metrics

时钟频率 Clock rate

- 用GHz度量 Measured in Gigahertz
- 阶段的功能划分和电路设计 Function of stage partitioning and circuit design
  - 保持每个阶段工作量较小 Keep amount of work per stage small

指令执行的速率 Rate at which instructions executed

- CPI：每条指令占用的周期数 CPI: cycles per instruction
- 平均每条指令需要多少时钟周期 On average, how many clock cycles does each instruction require?
- 流水线功能设计和基准测试程序 Function of pipeline design and benchmark programs
  - 例如分支预测错误的频率 E.g., how frequently are branches mispredicted?

流水线处理器的CPI CPI for PIPE

CPI ≈ 1.0

- 每个时钟周期取一条指令 Fetch instruction each clock cycle
- 几乎每个周期有效处理一条新指令 Effectively process new instruction almost every cycle
  - 尽管每条单独指令都有5个周期的时延 Although each individual instruction has latency of 5 cycles

CPI > 1.0

- 有时必须暂停或取消分支 Sometimes must stall or cancel branches

计算CPI Computing CPI

- C是时钟周期数 C clock cycles
- I是执行完成的指令数 I instructions executed to completion
- B是注入的气泡数 B bubbles injected (C = I + B)

$$CPI = C/I = (I+B)/I = 1.0 + B/I$$

- 因子B/I表示气泡的平均惩罚 Factor B/I represents average penalty due to bubbles

流水线处理器的CPI CPI for PIPE (Cont.)

B/I = LP + MP + RP

典型值 Typical Values

- LP是由于装载/使用冒险暂停导致的惩罚 LP: Penalty due to load/use hazard stalling
  - Load指令的占比 Fraction of instructions that are loads 0.25
  - Load指令需要暂停的比例 Fraction of load instructions requiring stall 0.20
  - 每次注入的气泡数 Number of bubbles injected each time 1

  ⇒ LP = 0.25 * 0.20 * 1 = 0.05
- MP是由于分支预测错误导致的惩罚 MP: Penalty due to mispredicted branches
  - 条件跳转指令的占比 Fraction of instructions that are cond. jumps 0.20
  - 条件跳转预测错误的比例 Fraction of cond. jumps mispredicted 0.40
  - 每次注入的气泡数 Number of bubbles injected each time 2

  ⇒ MP = 0.20 * 0.40 * 2 = 0.16

流水线处理器的CPI CPI for PIPE (Cont.)

B/I = LP + MP + RP

典型值 Typical Values

- RP是返回指令导致的惩罚 RP: Penalty due to `ret` instructions
  - 返回指令的占比 Fraction of instructions that are returns 0.02
  - 每次注入的气泡数 Number of bubbles injected each time 3

  ⇒ RP = 0.02 * 3 = 0.06
- 惩罚的净效果 Net effect of penalties 0.05 + 0.16 + 0.06 = 0.27

  ⇒ CPI = 1.27 (还不赖 Not bad!)

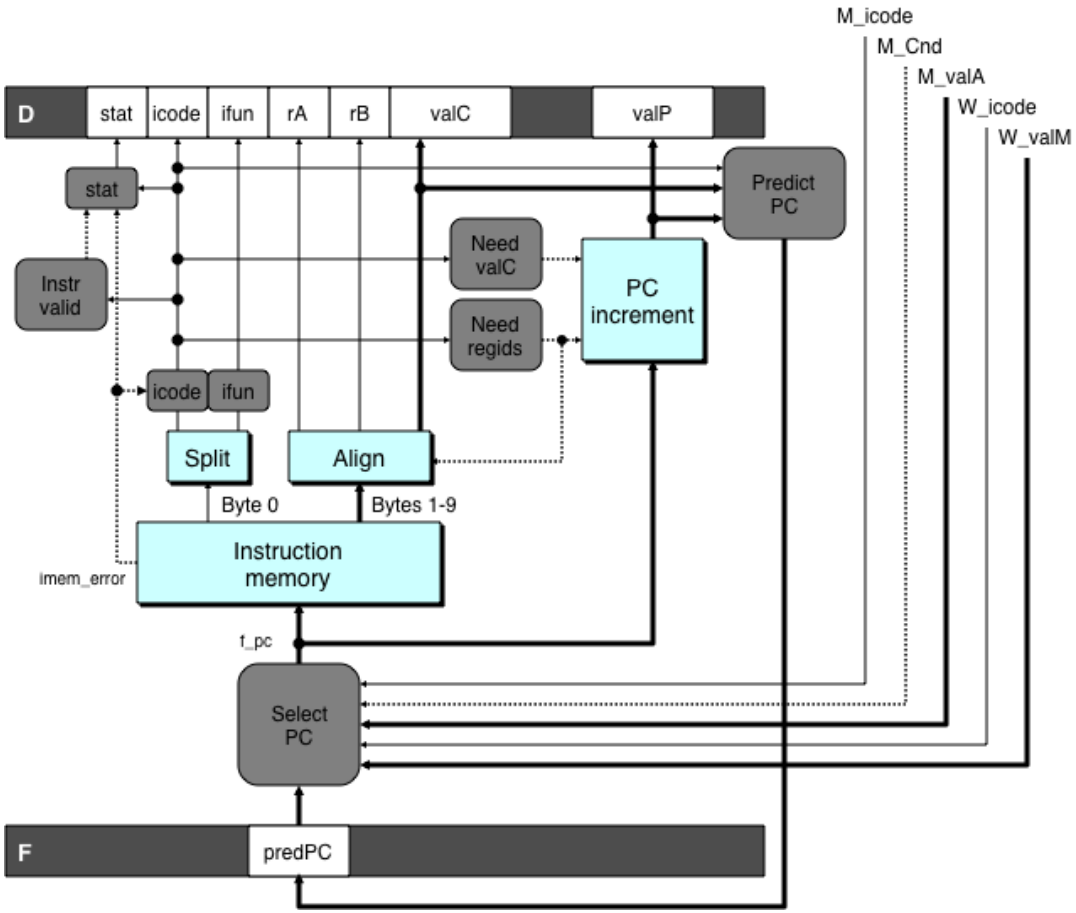
取指逻辑修正 Fetch Logic Revisited

在取指周期期间 During Fetch Cycle

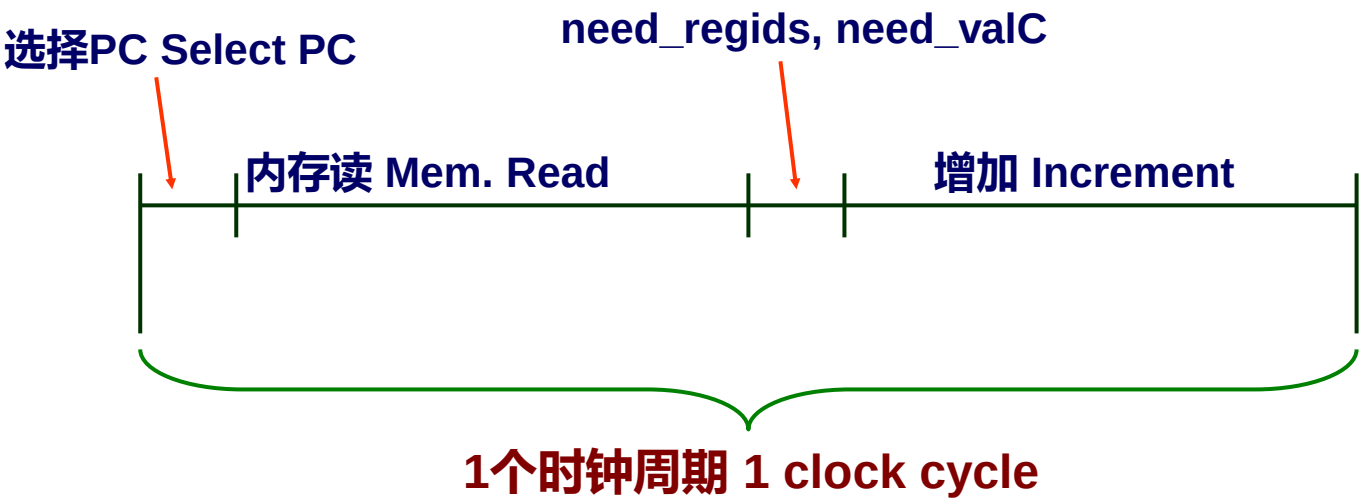
1. 选择PC Select PC
2. 从指令内存读字节 Read bytes from instruction memory
3. 检查icode确定指令长度 Examine icode to determine instruction length
4. 增加PC Increment PC

时序 Timing

- 步骤2和4需要较长的时间 Steps 2 & 4 require significant amount of time

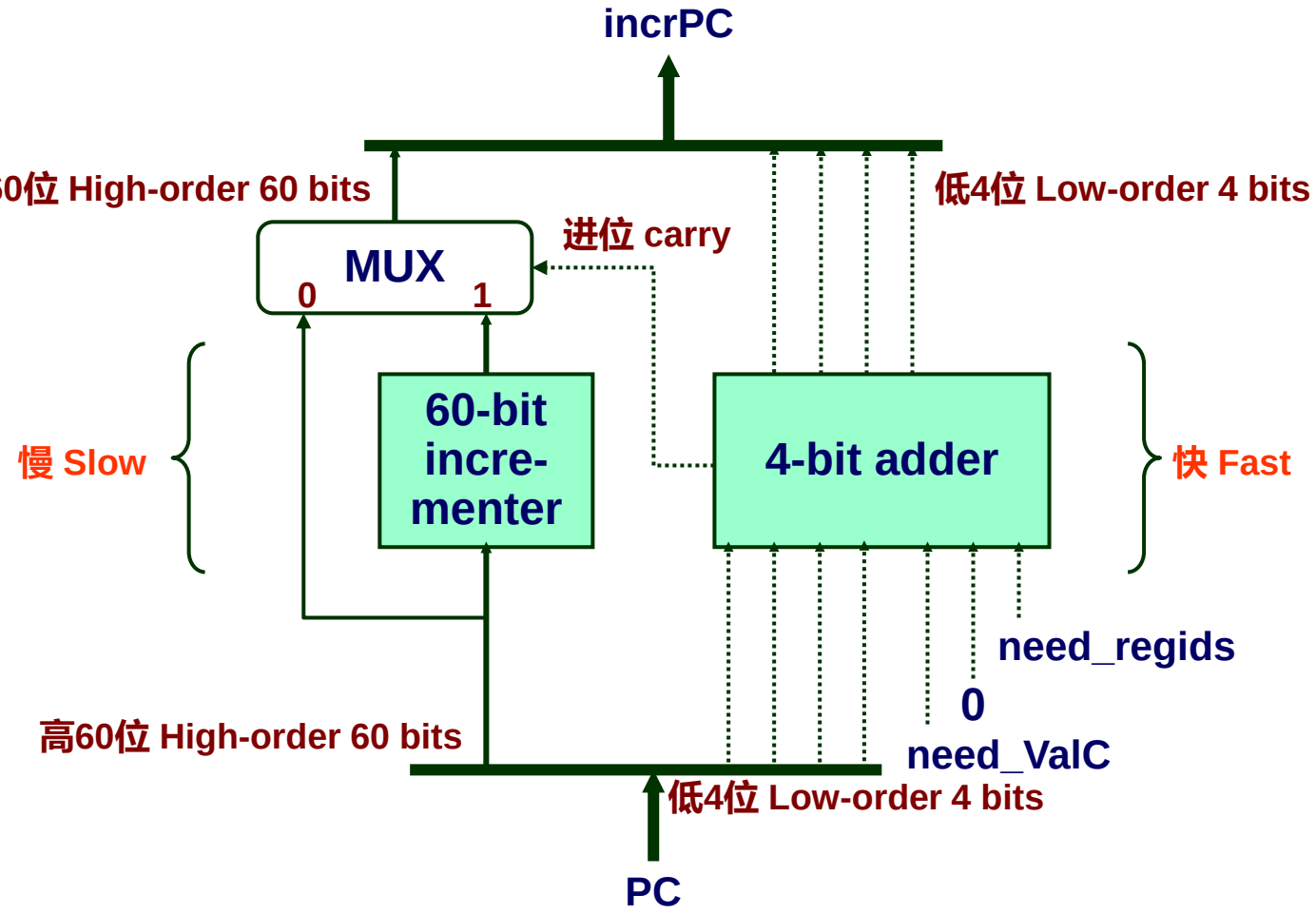


标准取指时序 Standard Fetch Timing

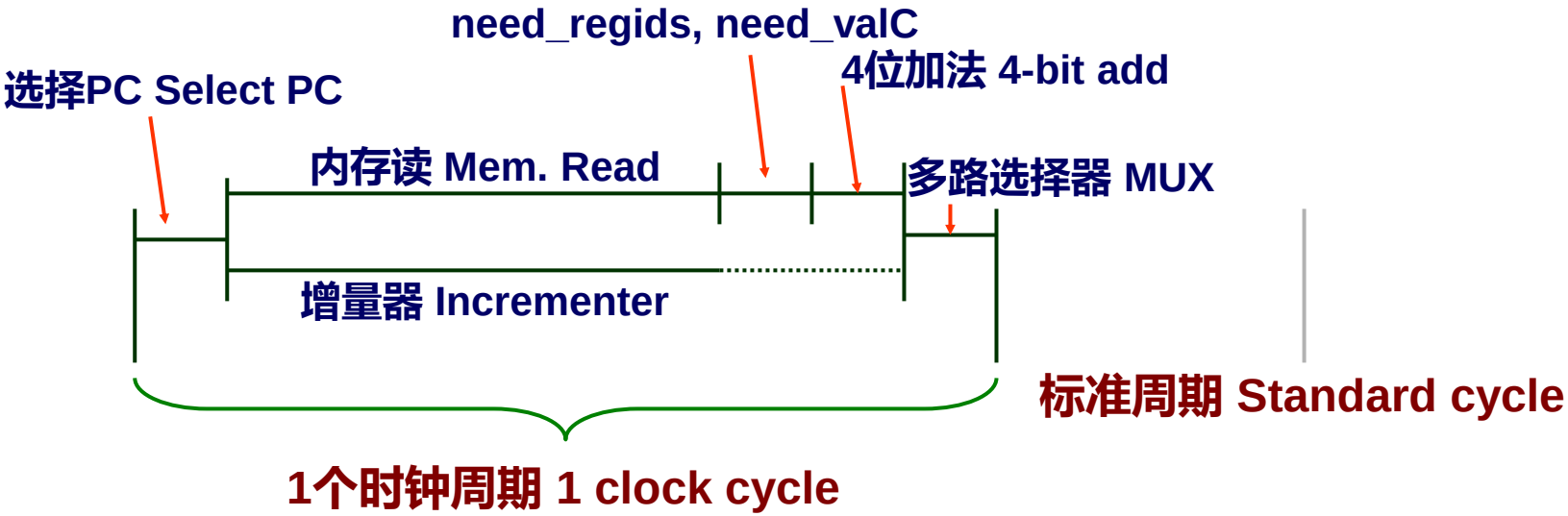


- 必须顺序执行每个操作 Must Perform Everything in Sequence
- 在知道PC需要增加多少前无法计算PC增加 Can't compute incremented PC until know how much to increment it by

快速的PC增加电路
A Fast PC Increment Circuit



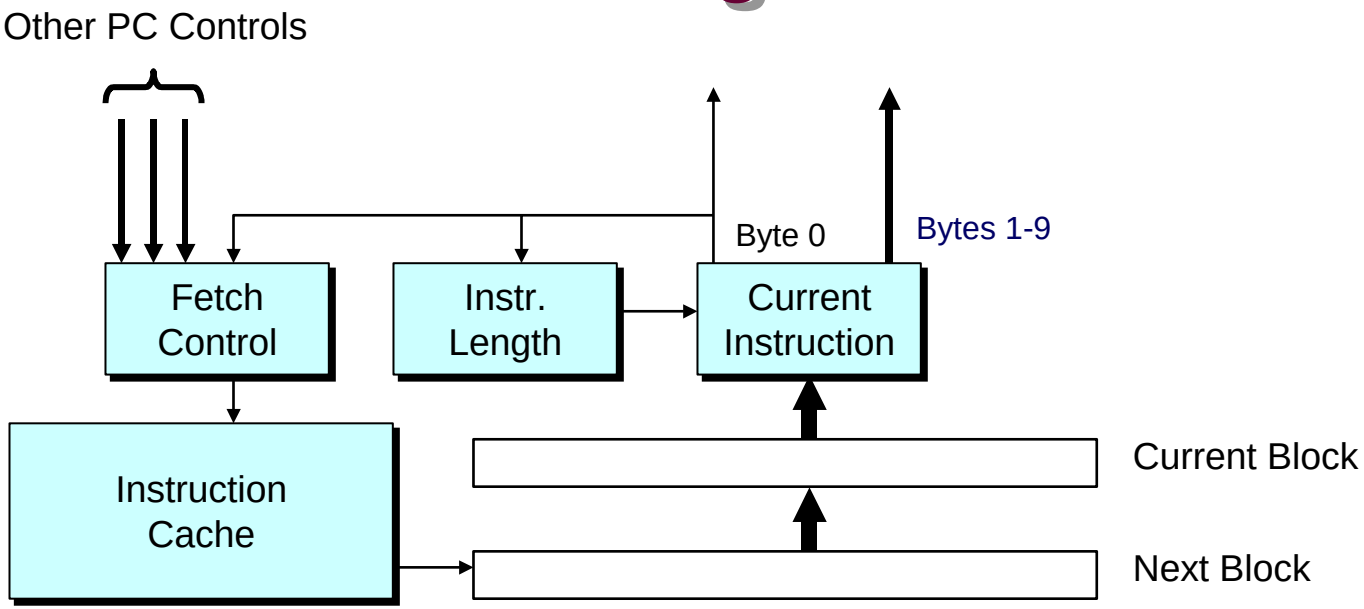
修改的取指时序 Modified Fetch Timing



60位的增量器 60-Bit Incrementer

- 只要选择PC后，立即开始行动 Acts as soon as PC selected
- 直到最后多路选择器才需要输出 Output not needed until final MUX
- 与内存读并行工作 Works in parallel with memory read

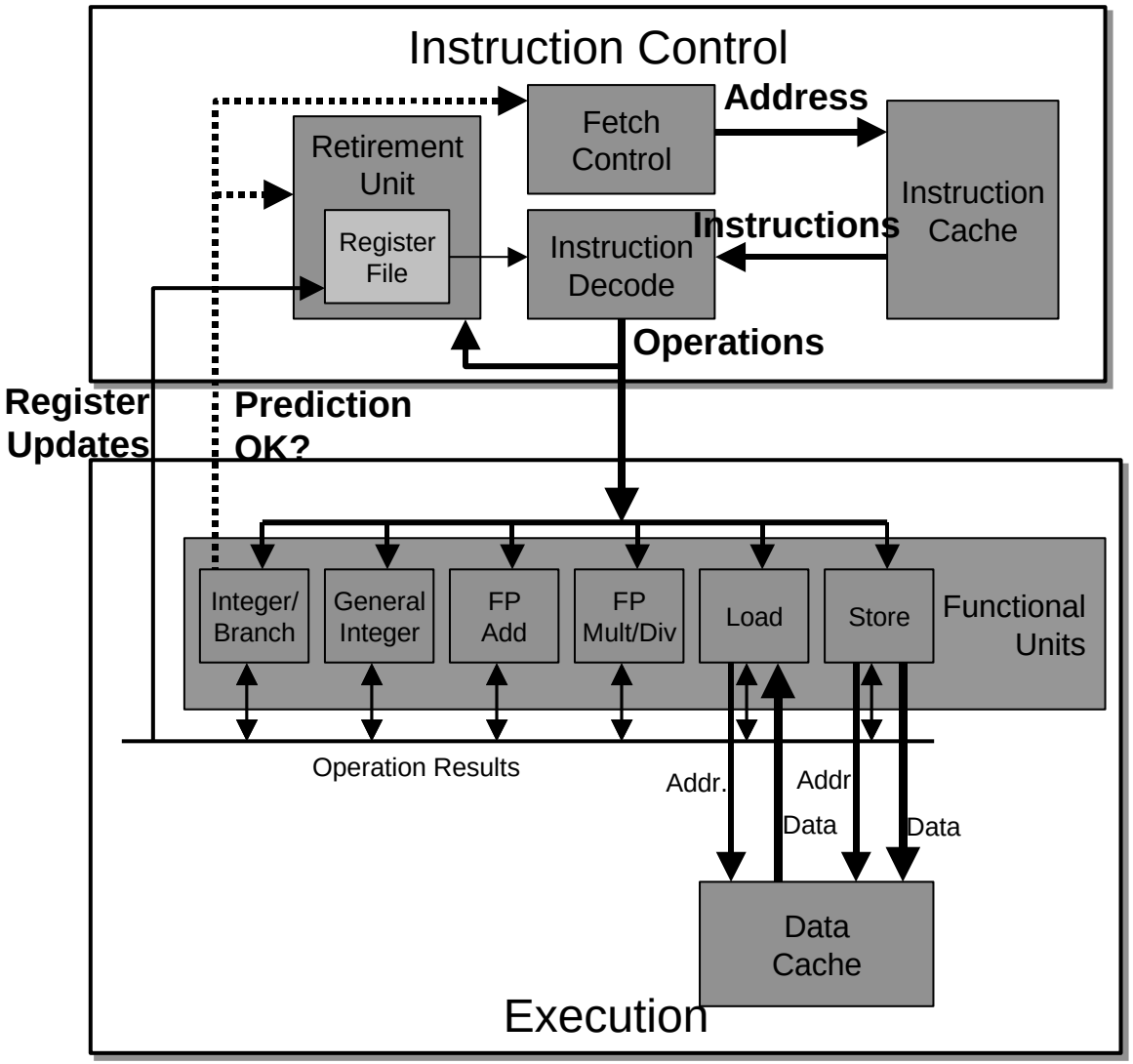
更真实的取指逻辑
More Realistic Fetch Logic



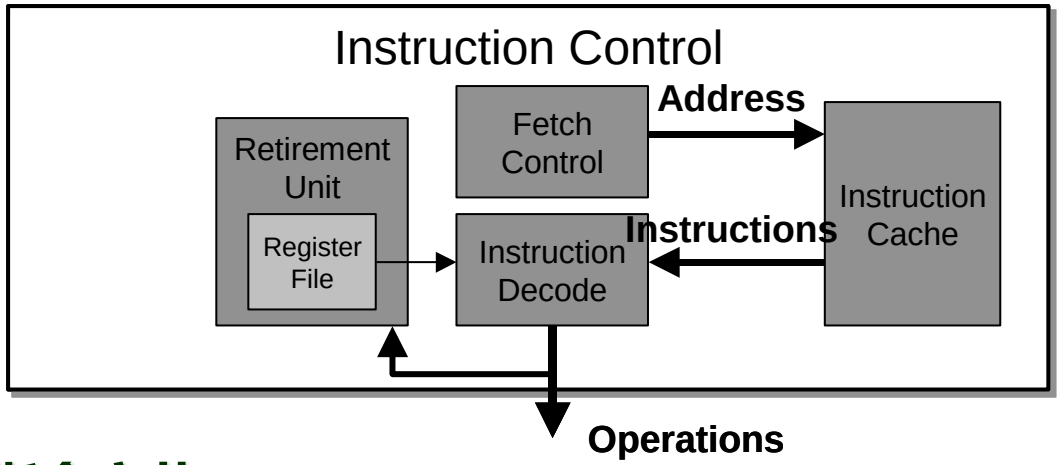
取指框 Fetch Box

- 集成进指令高速缓存 Integrated into instruction cache
- 取整个cache块（16或32字节） Fetches entire cache block (16 or 32 bytes)
- 从当前块选择当前指令 Selects current instruction from current block
- 提前工作取下一个块 Works ahead to fetch next block
  - 当到达当前块的尾部 As reaches end of current block
  - 在分支目标处 At branch target

现代CPU设计 Modern CPU Design



指令控制 Instruction Control



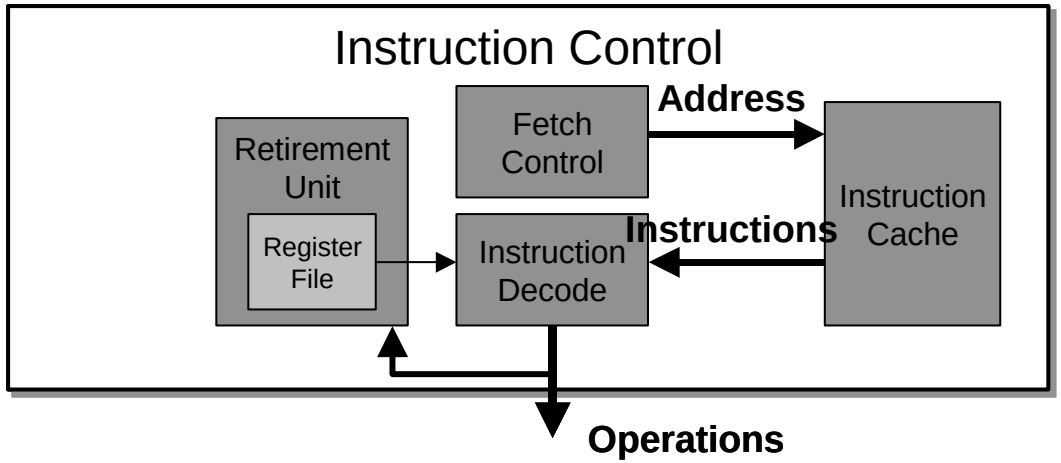
从内存抓取指令字节 Grabs Instruction Bytes From Memory

- 基于当前PC+分支预测的预测目标 Based on Current PC + Predicted Targets for Predicted Branches
- 硬件动态猜测是否选择/不选择分支和（可能的）分支目标 Hardware dynamically guesses whether branches taken/not taken and (possibly) branch target

翻译指令成操作 Translates Instructions Into *Operations*

- 执行指令所需要的原语步骤 Primitive steps required to perform instruction
- 典型指令需要1-3个操作 Typical instruction requires 1–3 operations

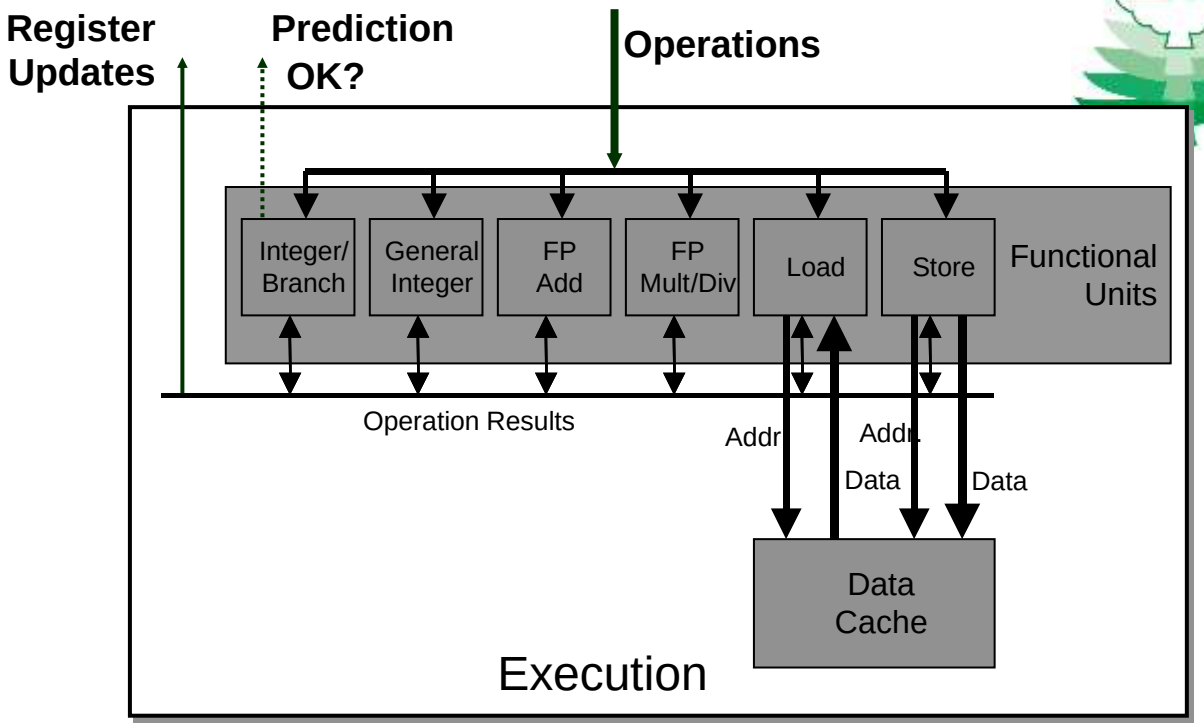
指令控制 Instruction Control



转换寄存器引用成标记 Converts Register References Into *Tags*

- 将一个操作的目的地址和后续操作的源地址链接在一起的抽象标识符
  Abstract identifier linking destination of one operation with sources of later operations

执行单元 Execution Unit



- 多功能单元 Multiple functional units
  - 每个单元可以独立地运行 Each can operate in independently
- 一旦操作数可用，就立即执行操作 Operations performed as soon as operands available
  - 没必要按照程序的顺序 Not necessarily in program order
  - 在功能单元范围内 Within limits of functional units
- 控制逻辑 Control logic
  - 确保行为等价于顺序程序执行 Ensures behavior equivalent to sequential program execution

Intel Haswell的CPU能力
CPU Capabilities of Intel Haswell

多条指令可以并行执行 Multiple Instructions Can Execute in Parallel

- 2条load指令 2 load
- 1条store指令 1 store
- 4条整数指令 4 integer
- 2条浮点乘法 2 FP multiply
- 1条浮点加/除 1 FP add / divide

有些指令占用1个以上周期，但是可以流水线化 Some Instructions Take > 1 Cycle, but Can be Pipelined

指令 Instruction	时延 Latency	周期/发射 Cycles/Issue
装载/存储 Load / Store	4	1
整数乘 Integer Multiply	3	1
整数除 Integer Divide	3—30	3—30
双/单精度浮点乘 Double/Single FP Multiply	5	1
双/单精度浮点加 Double/Single FP Add	3	1
双/单精度浮点除 Double/Single FP Divide	10—15	6—11

Haswell操作 Haswell Operation

- 动态翻译指令成“Uops” Translates instructions dynamically into “Uops”
  - 超过118位宽 ~118 bits wide
  - 包括操作、两个源地址和目的地址 Holds operation, two sources, and destination
- 执行Uops用“乱序”引擎 Executes Uops with “Out of Order” engine
  - Uop执行，当以下时候 Uop executed when
    - 操作数可用 Operands available
    - 功能单元可用 Functional unit available
  - 由“预留站”控制执行 Execution controlled by “Reservation Stations”
    - 保持跟踪uops之间的数据相关 Keeps track of data dependencies between uops
    - 分配资源 Allocates resources

高性能分支预测
High-Perforamnce Branch Prediction

对性能至关重要 Critical to Performance

- 典型地预测错误需要11-15周期惩罚 Typically 11–15 cycle penalty for misprediction

分支目标缓冲 Branch Target Buffer (BTB)

- 512个条目 512 entries
- 4位历史 4 bits of history
- 自适应算法 Adaptive algorithm
  - 能够识别重复的模式，例如替换选择-不选择 Can recognize repeated patterns, e.g., alternating taken–not taken

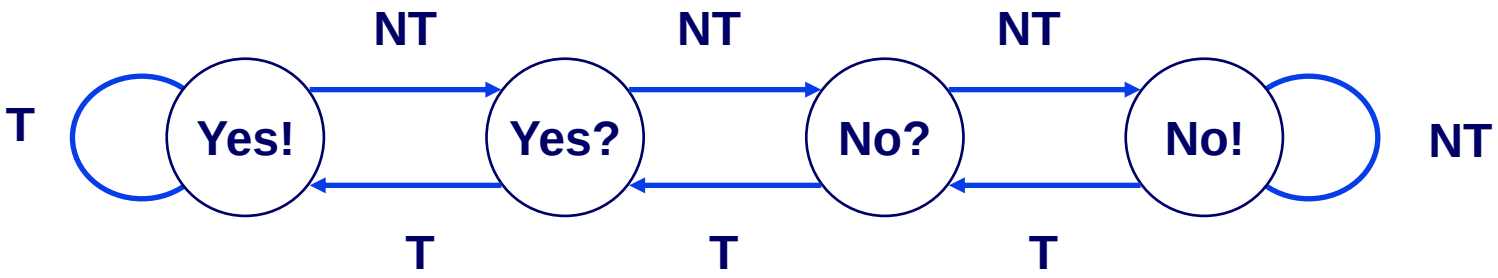
处理BTB缺失 Handling BTB misses

- 检测大约第6个周期 Detect in ~cycle 6
- 对负偏移预测选择，对正偏移预测不选择 Predict taken for negative offset, not taken for positive
  - 循环对条件分支 Loops vs. conditionals

示例分支预测 Example Branch Prediction

分支历史 Branch History

- 对有关以前分支指令的历史进行信息编码 Encode information about prior history of branch instructions
- 预测分支是否选择 Predict whether or not branch will be taken



状态机 State Machine

- 每次分支选择，向右转换 Each time branch taken, transition to right
- 当不选择，向左转换 When not taken, transition to left
- 预测分支选择当处于状态Yes！或Yes？ Predict branch taken when in state Yes! or Yes?

处理器小结 Processor Summary

设计技术 Design Technique

- 为所有指令创建统一的框架 Create uniform framework for all instructions
  - 想要在指令之间共享硬件 Want to share hardware among instructions
- 用控制逻辑位连接标准逻辑块 Connect standard logic blocks with bits of control logic

操作 Operation

- 状态存储在内存和时序寄存器中 State held in memories and clocked registers
- 由组合逻辑完成计算 Computation done by combinational logic
- 寄存器/内存时钟足以控制总体行为 Clocking of registers/memories sufficient to control overall behavior

处理器小结 Processor Summary

增强性能 Enhancing Performance

- 流水线增加了吞吐量和改进了资源利用率 Pipelining increases throughput and improves resource utilization
- 必须确保维持ISA行为 Must make sure to maintain ISA behavior