

| 章节 | 内容 | 备注 |
|---|---|---|
| 第一章 概论 | 性能评价 | CPU time、CC、CR、IC、CPI、MIPS、FLOPS RISC、CISC |
| 第二章 指令是硬件 机 器 的 语 言——计算 机指令系统 | 指令系统 MIPS汇编 算术、逻辑指令 转移指令、子程序 寻址方式 | MIPS汇编、MIPS指令格式、MIPS寻址方式、MIPS典型操作数种类、 MIPS典型指令：ADD、LW、SW、BEQ、BNE... |
| 第三章 计算机中数 的表示、转 换与运算 | 补码加减运算 原码乘除运算 IEEE 754浮点表示，加减运算 | 定点数补码加减算法及溢出判断 定点数一位原码乘除算法 IEEE 754浮点数及加减乘除 |
| 第四章 处理器—— 数据通路与 控制器的设 计 | 单个组件设计 MIPS典型指令单周期数据通道 控制器设计 | 指令的执行过程；Datapath概念；R-Type/Load/Store/BEQ/BNE/J Datapath 执行、设计和定性定量分析 |
| 第五章 存储体系结 构 | 存储器层次体系 位扩展字扩展 Cache 虚拟存储 | 存储分类；DRAM及刷新；位扩展、字扩展；cache组织结构、容量、相联 性、命中率、失效损失、写策略的定性分析与性能分析；两级cache性能分 析；虚拟存储器、页表、TLB的基本概念和原理 |
| 第六章 接口处理器 和外部设备 | I/O概论 总线概述 | 中断概念和原理、DMA概念和原理 |

Computer Organization and Design

Chapter 1

Computer Abstractions and Technology

Understanding Performance

- **Algorithm**
 - Determines number of operations executed
- **Programming language, compiler, ISA**
 - Determine number of machine instructions executed per operation
- **Processor and memory system**
 - Determine how fast instructions are executed
- **I/O system (including OS)**
 - Determines how fast I/O operations are executed

Levels of Program Code

- High-level language program (in C)

```
void swap (int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

one-to-many

C compiler

- Assembly language program (for MIPS)

```
swap:  sll    $2, $5, 2
       add    $2, $4, $2
       lw     $15, 0($2)
       lw     $16, 4($2)
       sw     $16, 0($2)
       sw     $15, 4($2)
       jr     $31
```

one-to-one

assembler

- Machine (object, binary) code (for MIPS)

```
000000 00000 00101 0001000010000000
000000 00100 00010 0001000000100000
. . .
```

Performance Metrics

- ❑ **Purchasing perspective**

- | given a collection of machines, which has the
 - best performance ?
 - least cost ?
 - best cost/performance?

- ❑ **Design perspective**

- | faced with design options, which has the
 - best performance improvement ?
 - least cost ?
 - best cost/performance?

- ❑ **Both require**

- | basis for comparison
- | metric for evaluation

- ❑ Our goal is to understand what factors in the architecture contribute to overall system performance and the relative importance (and cost) of these factors

Amdahl定律

- **Amdahl定律**：加快某部件执行速度所获得的系统性能加速比，受限于该部件在系统中所占的重要性。

$$\text{系统加速比} = \frac{\text{改进后的系统性能}}{\text{改进前的系统性能}}$$

$$\text{系统加速比} = \frac{\text{改进前的总执行时间}}{\text{改进后的总执行时间}}$$

$$S_p = \frac{T_0}{T_e}$$

系统加速比依赖于两个因素

$$\text{可改进比例 } F = \frac{\text{改进前可改进部分占用的时间}}{\text{改进前整个任务的执行时间}}$$

$$\text{部件加速比 } S = \frac{\text{改进前改进部分的执行时间}}{\text{改进后改进部分的执行时间}}$$

$$T_e = T_0(1-F) + T_0 \cdot \frac{F}{S} = T_0 \left[(1-F) + \frac{F}{S} \right]$$

$$S_p = \frac{T_0}{T_e} = \frac{1}{(1-F) + \frac{F}{S}}$$

- 作业：某计算机系统有三个部件可以改进，其加速比分别为：

$$S_1=30, S_2=20, S_3=10$$

- 如果部件1和部件2的可改进比例为30%，那么当部件3的可改进比例为多少时，系统的加速比才可以达到10？

$$T_e = T_0(1 - F_1 - F_2 - F_3) + T_0 \cdot \frac{F_1}{S_1} + T_0 \cdot \frac{F_2}{S_2} + T_0 \cdot \frac{F_3}{S_3}$$

$$= T_0 \left[(1 - F_1 - F_2 - F_3) + \frac{F_1}{S_1} + \frac{F_2}{S_2} + \frac{F_3}{S_3} \right]$$

$$S_p = \frac{T_0}{T_e} = \frac{1}{(1 - F_1 - F_2 - F_3) + \frac{F_1}{S_1} + \frac{F_2}{S_2} + \frac{F_3}{S_3}}$$

$$10 = \frac{1}{(1 - 0.3 - 0.3 - F_3) + \frac{0.3}{30} + \frac{0.3}{20} + \frac{F_3}{10}}$$

$$S_p = \frac{1}{(1 - 0.3 - 0.3 - 0.2) + \frac{0.3}{30} + \frac{0.3}{20} + \frac{0.2}{10}} = 0.245$$

$$S_p = \frac{T_0}{T_e} = \frac{1}{1 - \sum_i F_i + \sum_i \frac{F_i}{S_i}}$$

Amdahl定律的作用

$$S_p = \frac{T_0}{T_e} = \frac{1}{(1-F) + \frac{F}{S}}$$

- 系统整体性能的提高有一极大值 $S_p \leq \frac{1}{(1-F)}$
- 如果仅仅对计算机中的一部分做性能改进，则改进越多，系统获得的效果越小；
- 指明了设计原则：按各部分所占的时间比例来分配资源；
- 指出了两种改进设计提高性能的方法：
 - ☆ 优先考虑高频事件，使之尽量快速实现
 - ☆ 减小(1-F)，进一步提高高频事件的使用频度
- 给出了定量比较不同设计方案的方法。

Response Time and Throughput

- Response time
 - How long it takes to do a task
 - Important to **individual** users
- Throughput
 - Total work done per unit time
 - e.g., tasks/transactions/... per hour
 - Important to **data center** managers
- How are response time & throughput affected by
 - Replacing the processor with a faster version?
 - Adding more processors?
- We'll focus on response time for now...

Relative Performance

- Define Performance = $1/\text{Execution Time}$
- “X is n time faster than Y”

$$\begin{aligned} & \text{Performance}_X / \text{Performance}_Y \\ &= \text{Execution time}_Y / \text{Execution time}_X = n \end{aligned}$$

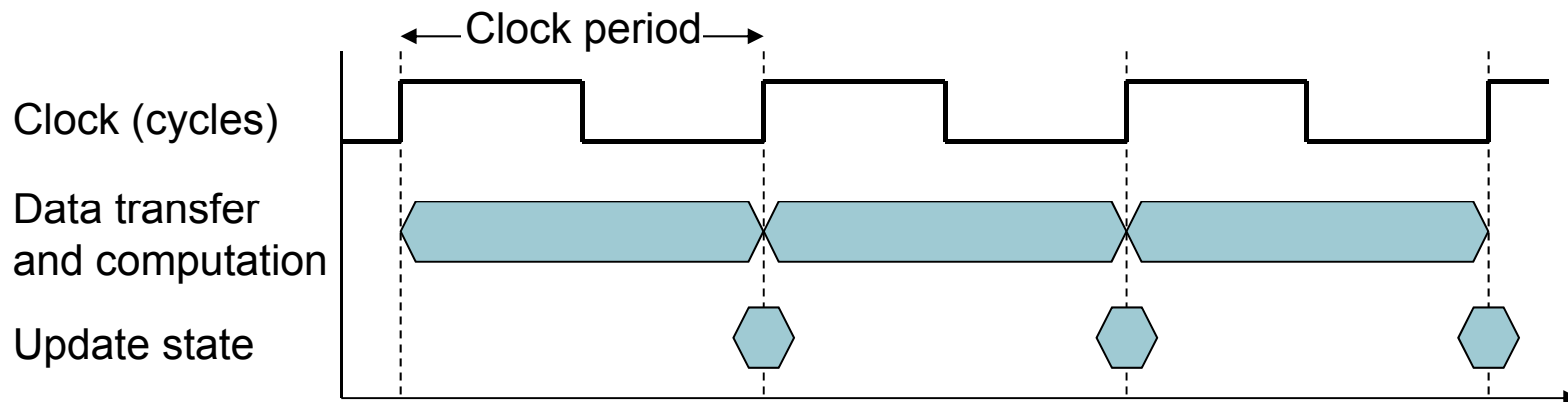
- **Example:** time taken to run a program
 - 10s on A, 15s on B
 - $\text{Execution Time}_B / \text{Execution Time}_A$
 $= 15\text{s} / 10\text{s} = 1.5$
 - So A is 1.5 times faster than B

Measuring Execution Time

- Execution time: seconds/program
- **Elapsed time (wall clock time)**
 - Total response time, including all aspects
 - Processing, I/O, OS overhead, idle time
 - Determines system performance
- **CPU time**
 - Time spent processing a given job
 - Discounts I/O time, other jobs' shares
 - Comprises user CPU time and system CPU time
 - Different programs are affected differently by CPU and system performance

CPU Clocking: Review

- Operation of digital hardware governed by a constant-rate clock



- **Clock period:** duration of a clock cycle
 - e.g., $250\text{ps} = 0.25\text{ns} = 250 \times 10^{-12}\text{s}$
- **Clock frequency (rate):** cycles per second
 - e.g., $4.0\text{GHz} = 4000\text{MHz} = 4.0 \times 10^9\text{Hz}$

CPU Clocking: Review

- ❑ Clock rate (clock cycles per second in MHz or GHz) is inverse of clock cycle time (clock period)

$$CC = 1 / CR$$

10 nsec clock cycle \Rightarrow 100 MHz clock rate

5 nsec clock cycle \Rightarrow 200 MHz clock rate

2 nsec clock cycle \Rightarrow 500 MHz clock rate

1 nsec (10^{-9}) clock cycle \Rightarrow 1 GHz (10^9) clock rate

500 psec clock cycle \Rightarrow 2 GHz clock rate

250 psec clock cycle \Rightarrow 4 GHz clock rate

200 psec clock cycle \Rightarrow 5 GHz clock rate

CPU Time

$$\begin{aligned}\text{CPU Time} &= \text{CPU Clock Cycles} \times \text{Clock Cycle Time} \\ &= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}\end{aligned}$$

- Performance improved by
 - Reducing number of clock cycles
 - Increasing clock rate
- Hardware designer must often trade off clock rate against cycle count
 - Many techniques that decrease the number of clock cycles also increase the clock cycle time

CPU Time Example

- A program runs on computer A with a 2 GHz clock in 10 seconds. What clock rate must computer B run at to run this program in 6 seconds? Unfortunately, to accomplish this, computer B will require 1.2 times as many clock cycles as computer A to run the program.

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6s}$$

$$\begin{aligned}\text{Clock Cycles}_A &= \text{CPU Time}_A \times \text{Clock Rate}_A \\ &= 10s \times 2\text{GHz} = 20 \times 10^9\end{aligned}$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6s} = \frac{24 \times 10^9}{6s} = 4\text{GHz}$$

Instruction Count and CPI

$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$

$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- **Instruction Count for a program**
 - Determined by program, ISA and compiler
- **Average cycles per instruction**
 - Determined by CPU hardware
 - If different instructions have different CPI
 - Average CPI affected by instruction mix

CPI Example

- **Computer A**: Cycle Time = 250ps, CPI = 2.0
- **Computer B**: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\ &= 1 \times 2.0 \times 250\text{ps} = 1 \times 500\text{ps} \leftarrow \text{A is faster...}\end{aligned}$$

$$\begin{aligned}\text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\ &= 1 \times 1.2 \times 500\text{ps} = 1 \times 600\text{ps}\end{aligned}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{1 \times 600\text{ps}}{1 \times 500\text{ps}} = 1.2 \leftarrow \text{...by this much}$$

CPI in More Detail

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^n (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^n \left(\text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)$$

Relative frequency

CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C. **What is avg. CPI?**

| Class | A | B | C |
|------------------|---|---|---|
| CPI for class | 1 | 2 | 3 |
| IC in sequence 1 | 2 | 1 | 2 |
| IC in sequence 2 | 4 | 1 | 1 |

- Sequence 1: IC = 5**

- Clock Cycles
 $= 2 \times 1 + 1 \times 2 + 2 \times 3$
 $= 10$
- Avg. CPI = $10/5 = 2.0$

- Sequence 2: IC = 6**

- Clock Cycles
 $= 4 \times 1 + 1 \times 2 + 1 \times 3$
 $= 9$
- Avg. CPI = $9/6 = 1.5$

Performance Summary

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

| | Instruction_count | CPI | clock_cycle |
|----------------------|-------------------|-----|-------------|
| Algorithm | X | X | |
| Programming language | X | X | |
| Compiler | X | X | |
| ISA | X | X | X |
| Core organization | | X | X |
| Technology | | | X |

A Simple Example

| Op | Freq | CPI _i | Freq x CPI _i | | | |
|--------|------|------------------|-------------------------|-----|-----|------|
| ALU | 50% | 1 | .5 | .5 | .5 | .25 |
| Load | 20% | 5 | 1.0 | .4 | 1.0 | 1.0 |
| Store | 10% | 3 | .3 | .3 | .3 | .3 |
| Branch | 20% | 2 | .4 | .4 | .2 | .4 |
| | | | $\Sigma =$ 2.2 | 1.6 | 2.0 | 1.95 |

- How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

CPU time new = 1.6 x IC x CC so 2.2/1.6 means 37.5% faster

- How does this compare with using branch prediction to shave a cycle off the branch time?

CPU time new = 2.0 x IC x CC so 2.2/2.0 means 10% faster

- What if two ALU instructions could be executed at once?

CPU time new = 1.95 x IC x CC so 2.2/1.95 means 12.8% faster

Computer Organization and Design

2.1-2.14

A vertical blue line starts from the top left of the slide and extends downwards. A horizontal blue line crosses this vertical line, extending to the right. The text 'Instructions: Language of the Computer' is positioned to the right of the vertical line, below the horizontal line.

Instructions: Language
of the Computer

Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination

add a, b, c # a gets b + c
- All arithmetic operations have this form
- *Design Principle 1*: Simplicity favours regularity/简单有利于规律性
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

$f = (g + h) - (i + j);$

- Compiled MIPS code:

```
add t0, g, h    # temp t0 = g + h
add t1, i, j    # temp t1 = i + j
sub f,  t0, t1  # f = t0 - t1
```

Register Operands

- Arithmetic instructions use **register operands**
- MIPS has a **32 × 32**-bit register file
 - Use for frequently accessed data
 - Numbered 0 to 31
 - 32-bit data called a “word”
- **Assembler names**
 - **\$t0, \$t1, ..., \$t9** for temporary values
 - **\$s0, \$s1, ..., \$s7** for saved variables
- ***Design Principle 2***: Smaller is faster
 - Reason for small register file
 - c.f. main memory: millions of locations

Register Operand Example

- C code:

$f = (g + h) - (i + j);$

- f, \dots, j in $\$s0, \dots, \$s4$

- Compiled MIPS code:

`add $t0, $s1, $s2`

`add $t1, $s3, $s4`

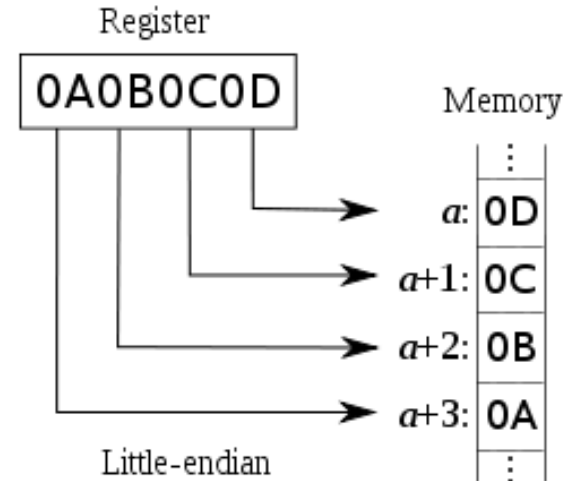
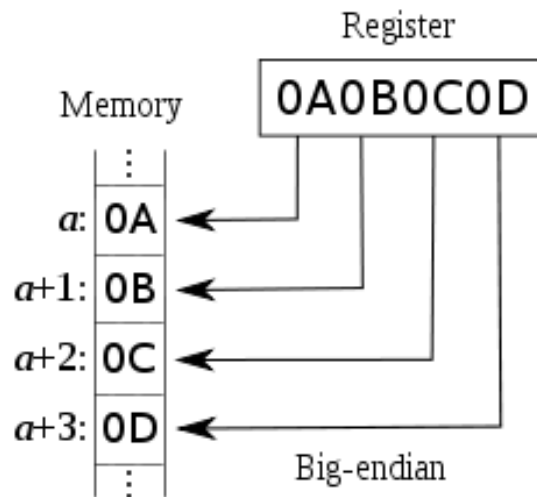
`sub $s0, $t0, $t1`

Memory Operands

- Main memory used for composite data
 - Arrays, structures, dynamic data, ...
 - Only a small amount of data can fit in registers
- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- Memory is **byte** addressed
 - Each address identifies an 8-bit byte
- Words are aligned in memory
 - Address must be a multiple of 4
- MIPS is **Big Endian**
 - Most-significant byte at least address of a word
 - *c.f.* Little Endian: least-significant byte at least address

Big vs. Little Endian

- ❑ **Big Endian:** leftmost byte is word address
IBM 360/370, Motorola 68k, **MIPS**, Sparc, HP PA
- ❑ **Little Endian:** rightmost byte is word address
Intel 80x86, DEC Vax, DEC Alpha (Windows NT)



Memory Operand Example 1

- C code:

`g = h + A[8];`

- `g` in `$s1`, `h` in `$s2`, base address of `A` in `$s3`

- Compiled MIPS code:

- Index 8 requires offset of 32

- 4 bytes per word

```
lw    $t0, 32($s3)    # load word
add   $s1, $s2, $t0
```

offset

base register

Memory Operand Example 2

- C code:

$A[12] = h + A[8];$

- h in $\$s2$, base address of A in $\$s3$

- Compiled MIPS code:

- Index 8 requires offset of 32

```
lw    $t0, 32($s3)    # load word
add   $t0, $s2, $t0
sw    $t0, 48($s3)    # store word
```

Registers vs. Memory

- Registers are faster to access than memory
 - Operating on memory data requires loads and stores
 - More instructions to be executed
- **Compilers** use registers for variables as much as possible
 - Only “spill” to memory for less frequently used variables
 - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

`addi $s3, $s3, 4`

- No subtract immediate instruction

- Just use a negative constant

`addi $s2, $s1, -1`

- *Design Principle 3:* Make the common case fast

- Small constants are common
- Immediate operand avoids a load instruction

The Constant Zero

- MIPS register 0 (**\$zero**) is the constant 0
 - Cannot be overwritten
- Useful for common operations
 - E.g., move between registers
`add $t2, $s1, $zero`

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$

- Example

- $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Using 32 bits

- 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$

- Example

- $$\begin{aligned} &1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2 \\ &= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= -2,147,483,648 + 2,147,483,644 = -4_{10} \end{aligned}$$

- Using 32 bits

- $-2,147,483,648$ to $+2,147,483,647$

2s-Complement Signed Integers

- Bit 31 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^n - 1)$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111
- By default, all data in MIPS is a signed integer
 - add vs. addu

Signed Negation

- Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111\dots111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
 - $+2 = 0000\ 0000 \dots 0010_2$
 - $-2 = 1111\ 1111 \dots 1101_2 + 1$
 $= 1111\ 1111 \dots 1110_2$

Sign Extension

- How to represent a number using more bits?
 - e.g. 16 bit number as a 32 bit number
 - Must preserve the numeric value
- Replicate the sign bit to the left(向左复制符号位)
 - c.f. (confer) unsigned values: extend with 0s(补充0)
- Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110
- In MIPS instruction set
 - addi : extend immediate value
 - l b, l h: extend loaded byte/halfword
 - beq, bne: extend the displacement

Representing Instructions

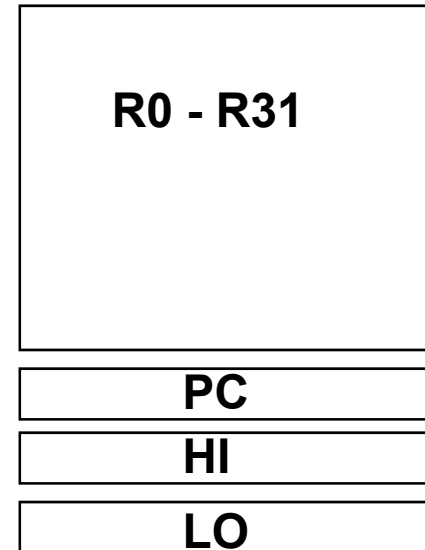
- Instructions are encoded in binary
 - Called **machine code**
- MIPS instructions
 - Encoded as 32-bit instruction words
 - Small number of formats encoding operation code (opcode), register numbers, ...
 - Regularity!
- Register numbers
 - **\$t0 – \$t7** are reg's 8 – 15
 - **\$t8 – \$t9** are reg's 24 – 25
 - **\$s0 – \$s7** are reg's 16 – 23

MIPS-32 ISA

❑ Instruction Categories

- | Computational
- | Load/Store
- | Jump and Branch
- | Floating Point
 - coprocessor
- | Memory Management
- | Special

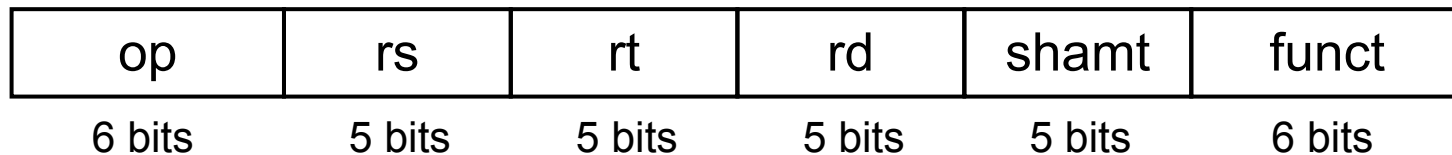
Registers



3 Instruction Formats: **all 32 bits wide**

| | | | | | | |
|----|-------------|----|-----------|----|-------|----------|
| op | rs | rt | rd | sa | funct | R format |
| op | rs | rt | immediate | | | I format |
| op | jump target | | | | | J format |

MIPS R-format Instructions



■ Instruction fields

- **op**: operation code (opcode)
- **rs**: first source register number
- **rt**: second source register number
- **rd**: destination register number
- **shamt**: shift amount (00000 for now)
- **funct**: function code (extends opcode)

R-format Example

| | | | | | |
|--------|--------|--------|--------|--------|--------|
| op | rs | rt | rd | shamt | funct |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

add \$t0, \$s1, \$s2

| | | | | | |
|---------|-------|-------|-------|-------|--------|
| special | \$s1 | \$s2 | \$t0 | 0 | add |
| 0 | 17 | 18 | 8 | 0 | 32 |
| 000000 | 10001 | 10010 | 01000 | 00000 | 100000 |

$00000010001100100100000000100000_2 = 02324020_{16}$

Hexadecimal(十六进制)

- Base 16
 - Compact representation of bit strings
 - 4 bits per hex digit

| | | | | | | | |
|---|------|---|------|---|------|---|------|
| 0 | 0000 | 4 | 0100 | 8 | 1000 | c | 1100 |
| 1 | 0001 | 5 | 0101 | 9 | 1001 | d | 1101 |
| 2 | 0010 | 6 | 0110 | a | 1010 | e | 1110 |
| 3 | 0011 | 7 | 0111 | b | 1011 | f | 1111 |

- Example: eca8 6420
 - 1110 1100 1010 1000 0110 0100 0010 0000

MIPS I-format Instructions



- Immediate arithmetic and load/store instructions
 - **rt**: destination or source register number
 - **Constant**: -2^{15} to $+2^{15} - 1$
 - **Address**: offset added to base address in rs
- **Design Principle 4**: Good design demands good compromises
 - Different formats complicate decoding, but allow 32-bit instructions uniformly (不同的格式使得解码复杂化, 除非允许32位指令均匀)
 - Keep formats as similar as possible

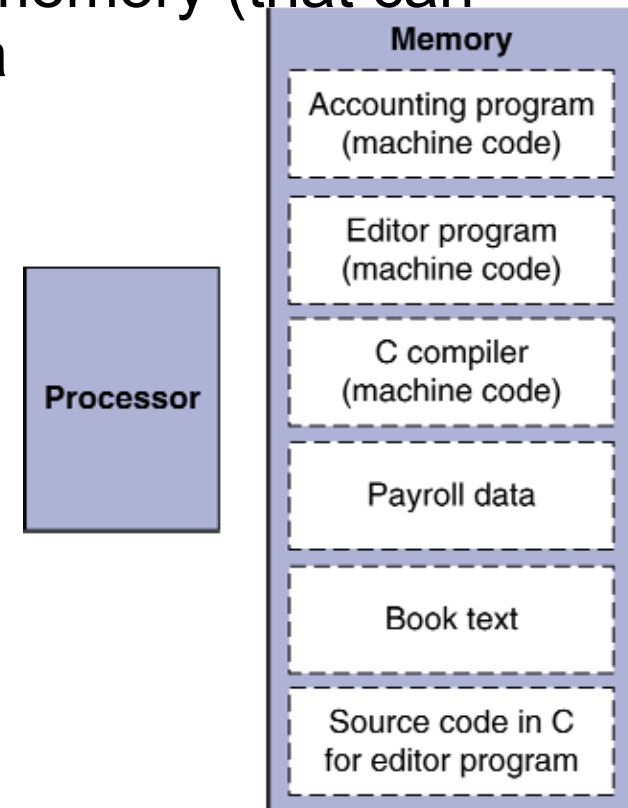
Two Key Principles of Computer Design

1. Instructions are represented as numbers and, as such, are indistinguishable from data(指令被表示为数字，因此，从数据是无法区分的)
2. Programs are stored in alterable memory (that can be read or written to) just like data

The BIG Picture

❑ Stored-program concept

- | Programs can be shipped as files of binary numbers – binary compatibility
- | Computers can inherit ready-made software provided they are compatible with an existing ISA – leads industry to align around a small number of ISAs



Logical Operations

- Instructions for bitwise manipulation

| Operation | C | Java | MIPS |
|-------------|----|------|-----------|
| Shift left | << | << | sll |
| Shift right | >> | >>> | srl |
| Bitwise AND | & | & | and, andi |
| Bitwise OR | | | or, ori |
| Bitwise NOT | ~ | ~ | nor |

- Useful for extracting and inserting groups of bits in a word

Shift Operations

| | | | | | |
|--------|--------|--------|--------|--------|--------|
| op | rs | rt | rd | shamt | funct |
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

- **shamt**: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - `sl l` by i bits multiplies by 2^i
- Shift right logical
 - Shift right and fill with 0 bits
 - `srl` by i bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
 - Select some bits, clear others to 0

and \$t0, \$t1, \$t2

| | |
|------|---|
| \$t2 | 0000 0000 0000 0000 0000 1101 1100 0000 |
| \$t1 | 0000 0000 0000 0000 0011 1100 0000 0000 |
| \$t0 | 0000 0000 0000 0000 0000 1100 0000 0000 |

OR Operations

- Useful to include bits in a word
 - Set some bits to 1, leave others unchanged

or \$t0, \$t1, \$t2

| | |
|------|---|
| \$t2 | 0000 0000 0000 0000 0000 1101 1100 0000 |
| \$t1 | 0000 0000 0000 0000 0011 1100 0000 0000 |
| \$t0 | 0000 0000 0000 0000 0011 1101 1100 0000 |

NOT Operations

- Useful to invert bits in a word
 - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
 - $a \text{ NOR } b == \text{NOT} (a \text{ OR } b)$

`nor $t0, $t1, $zero`

Register 0: always
read as zero

\$t1 0000 0000 0000 0000 0011 1100 0000 0000

\$t0 1111 1111 1111 1111 1100 0011 1111 1111

Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- `beq rs, rt, L1`
 - if (`rs == rt`) branch to instruction labeled L1;
- `bne rs, rt, L1`
 - if (`rs != rt`) branch to instruction labeled L1;
- `j L1`
 - unconditional jump to instruction labeled L1

Compiling If Statements

- C code:

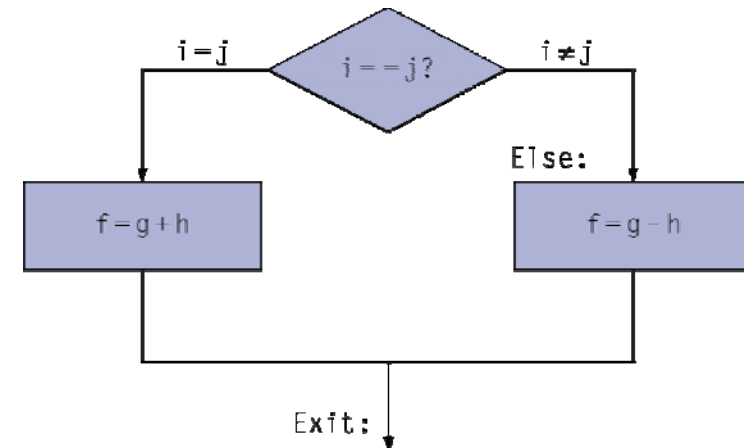
```
if (i == j) f = g+h;  
else f = g-h;
```

- f, g, ... in **\$s0, \$s1, ...**

- Compiled MIPS code:

```
        bne $s3, $s4, Else  
        add $s0, $s1, $s2  
        j   Exit  
Else:   sub $s0, $s1, $s2  
Exit:   ...
```

Assembler calculates addresses



Compiling Loop Statements

- C code:

while (save[i] == k) i += 1;

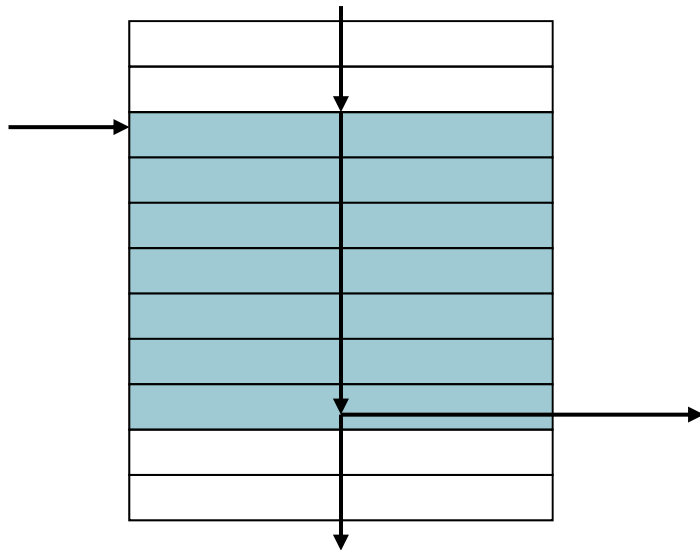
- i in **\$s3**, k in **\$s5**, address of save in **\$s6**

- Compiled MIPS code:

```
Loop:  sll    $t1, $s3, 2    # $t1 = i * 4
       add   $t1, $t1, $s6
       lw    $t0, 0($t1)
       bne   $t0, $s5, Exit
       addi   $s3, $s3, 1
       j     Loop
Exit:  ...
```


Basic Blocks

- A basic block is a sequence of instructions with
 - No embedded branches (except at end)
 - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- Set result to 1 if a condition is true
 - Otherwise, set to 0
- `sl t rd, rs, rt`
 - if ($rs < rt$) $rd = 1$; else $rd = 0$;
- `sl ti rt, rs, constant`
 - if ($rs < \text{constant}$) $rt = 1$; else $rt = 0$;
- Use in combination with `beq`, `bne`

```
sl t $t0, $s1, $s2 # if ($s1 < $s2)
bne $t0, $zero, L  # branch to L
```

Branch Instruction Design

- Why not **blt**, **bge**, etc?
- Hardware for $<$, \geq , ... slower than $=$, \neq
 - Combining with branch involves more work per instruction, requiring a slower clock
 - All instructions penalized!
- **beq** and **bne** are the common case
- This is a good design compromise

Signed vs. Unsigned

- Signed comparison: `sl t, sl ti`
- Unsigned comparison: `sl tu, sl tui`
- Example
 - `$s0` = 1111 1111 1111 1111 1111 1111 1111 1111
 - `$s1` = 0000 0000 0000 0000 0000 0000 0000 0001
 - `sl t $t0, $s0, $s1 # signed`
 - $-1 < +1 \Rightarrow \$t0 = 1$
 - `sl tu $t0, $s0, $s1 # unsigned`
 - $+4,294,967,295 > +1 \Rightarrow \$t0 = 0$

Procedure Calling

- Procedures enable **structured** programs
 - Easier to understand and reuse code
- Steps required
 1. Place parameters in registers
 2. Transfer control to procedure
 3. Acquire storage for procedure
 4. Perform procedure's operations
 5. Place result in register for caller
 6. Return to place of call

Register Usage

- **\$a0 – \$a3**: arguments (reg's 4 – 7)
- **\$v0, \$v1**: result (i.e. return) values (reg's 2 and 3)
- **\$t0 – \$t9**: temporaries
 - Can be overwritten by callee
- **\$s0 – \$s7**: saved
 - Must be saved/restored by callee
- **\$gp**: global pointer for static data (reg 28)
- **\$sp**: stack pointer (reg 29)
- **\$fp**: frame pointer (reg 30)
- **\$ra**: return address (reg 31)

Procedure Call Instructions

- Procedure call: jump and link

`j al ProcedureLabel`

- Address of following instruction put in `$ra`
- Jumps to target address

- Procedure return: jump register

`j r $ra`

- Copies `$ra` to program counter
- Can also be used for computed jumps
 - e.g., for case/switch statements

Leaf Procedure Example

- C code:

```
int leaf_example (int g, h, i, j)
{ int f;
  f = (g + h) - (i + j);
  return f;
}
```

- Arguments g, ..., j in **\$a0, ..., \$a3**
- f in **\$s0** (hence, need to save **\$s0** on stack)
- Result in **\$v0**

Leaf Procedure Example

- MIPS code:

| | | | | |
|---------------|-------|---------|--------|--------------------|
| leaf_example: | | | | |
| addi | \$sp, | \$sp, | -4 | Save \$s0 on stack |
| sw | \$s0, | 0(\$sp) | | |
| add | \$t0, | \$a0, | \$a1 | Procedure body |
| add | \$t1, | \$a2, | \$a3 | |
| sub | \$s0, | \$t0, | \$t1 | |
| add | \$v0, | \$s0, | \$zero | Result |
| lw | \$s0, | 0(\$sp) | | Restore \$s0 |
| addi | \$sp, | \$sp, | 4 | |
| jr | \$ra | | | Return |

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
 - Any saved registers being used \$s0-\$s7
 - Its return address
 - Any arguments and temporaries needed after the call
 - e.g., suppose main program calls procedure A with an argument 3 in \$a0 and then using **jal A**
 - Then if procedure A calls procedure B via **jal B** with an argument of 7 placed in \$a0, there is a conflict over the use of register \$a0
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
int fact (int n)
{
    if (n < 1) return 1;
    else return n * fact(n - 1);
}
```

- Argument n in **\$a0**
- Result in **\$v0**

Non-Leaf Procedure Example

- MIPS code:

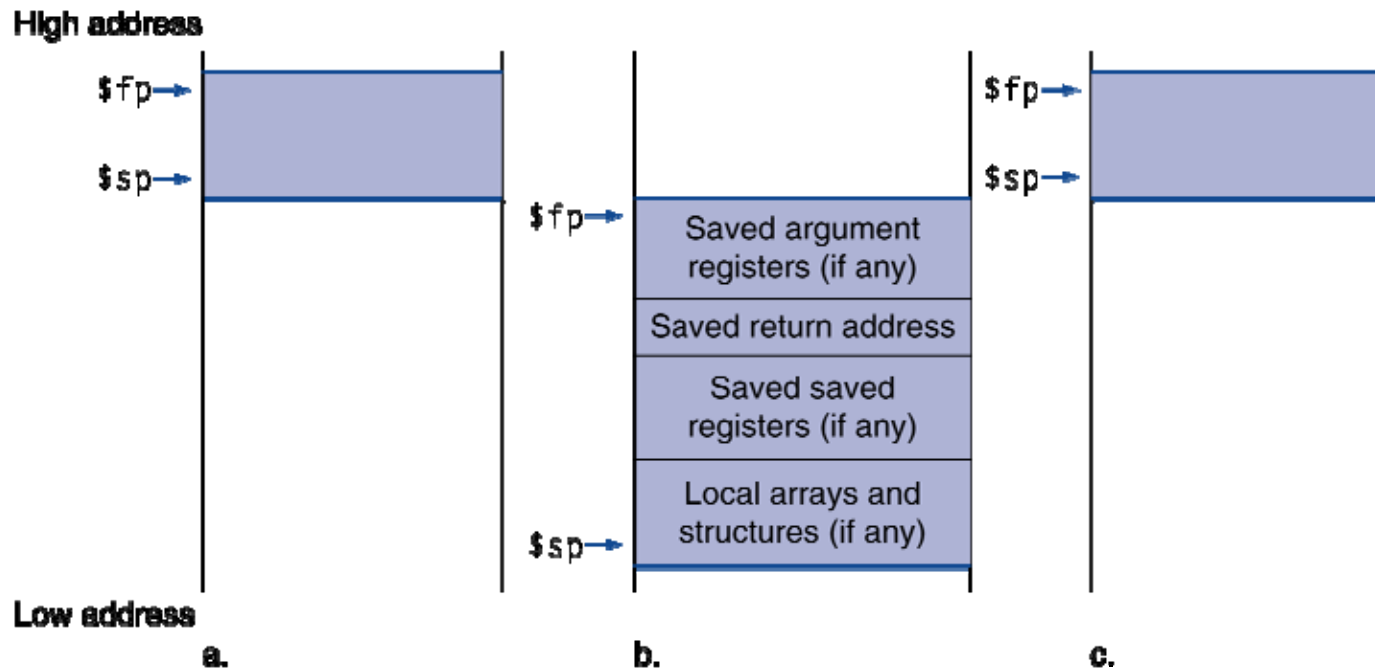
| | | |
|-------|---------------------|----------------------------|
| fact: | | |
| addi | \$sp, \$sp, -8 | # adjust stack for 2 items |
| sw | \$ra, 4(\$sp) | # save return address |
| sw | \$a0, 0(\$sp) | # save argument |
| slti | \$t0, \$a0, 1 | # test for n < 1 |
| beq | \$t0, \$zero, L1 | |
| addi | \$v0, \$zero, 1 | # if so, result is 1 |
| addi | \$sp, \$sp, 8 | # pop 2 items from stack |
| jr | \$ra | # and return |
| L1: | addi \$a0, \$a0, -1 | # else decrement n |
| | jal fact | # recursive call |
| lw | \$a0, 0(\$sp) | # restore original n |
| lw | \$ra, 4(\$sp) | # and return address |
| addi | \$sp, \$sp, 8 | # pop 2 items from stack |
| mul | \$v0, \$a0, \$v0 | # multiply to get result |
| jr | \$ra | # and return |

MIPS Register Conventions

| Name | Register number | Usage | Preserved on call? |
|-----------|-----------------|--|--------------------|
| \$zero | 0 | The constant value 0 | n.a. |
| \$v0-\$v1 | 2-3 | Values for results and expression evaluation | no |
| \$a0-\$a3 | 4-7 | Arguments | no |
| \$t0-\$t7 | 8-15 | Temporaries | no |
| \$s0-\$s7 | 16-23 | Saved | yes |
| \$t8-\$t9 | 24-25 | More temporaries | no |
| \$gp | 28 | Global pointer | yes |
| \$sp | 29 | Stack pointer | yes |
| \$fp | 30 | Frame pointer | yes |
| \$ra | 31 | Return address | yes |

Register 1, called **\$at**, is reserved for the assembler and registers 26-27, called **\$k0-\$k1**, are reserved for the operating system.

Local Data on the Stack



- Local data allocated by callee
 - e.g., C automatic variables
- Procedure frame (activation record)
 - Used by some compilers to manage stack storage

Byte/Halfword Operations

- To manipulate text (e.g. ASCII characters) byte operations are essential
- MIPS byte/halfword load/store
 - String processing is a common case

`lb rt, offset(rs)` `lh rt, offset(rs)`

- Sign extend to 32 bits in rt

`lbu rt, offset(rs)` `lhu rt, offset(rs)`

- Zero extend to 32 bits in rt

`sb rt, offset(rs)` `sh rt, offset(rs)`

- Store just rightmost byte/halfword

String Copy Example

- C code (naïve):

- Null-terminated string

```
void strcpy (char x[], char y[])  
{ int i;  
  i = 0;  
  while ((x[i]=y[i])!='\0')  
    i += 1;  
}
```

- Addresses of x, y in **\$a0, \$a1**
- i in **\$s0**

String Copy Example

- MIPS code:

| | | |
|---------|--------------------------|---------------------------|
| strcpy: | | |
| | addi \$sp, \$sp, -4 | # adjust stack for 1 item |
| | sw \$s0, 0(\$sp) | # save \$s0 |
| | add \$s0, \$zero, \$zero | # i = 0 |
| L1: | add \$t1, \$s0, \$a1 | # addr of y[i] in \$t1 |
| | lbu \$t2, 0(\$t1) | # \$t2 = y[i] |
| | add \$t3, \$s0, \$a0 | # addr of x[i] in \$t3 |
| | sb \$t2, 0(\$t3) | # x[i] = y[i] |
| | beq \$t2, \$zero, L2 | # exit loop if y[i] == 0 |
| | addi \$s0, \$s0, 1 | # i = i + 1 |
| | j L1 | # next iteration of loop |
| L2: | lw \$s0, 0(\$sp) | # restore saved \$s0 |
| | addi \$sp, \$sp, 4 | # pop 1 item from stack |
| | jr \$ra | # and return |

32-bit Constants

- Most constants are small
 - 16-bit immediate is sufficient
- For the occasional 32-bit constant
 - `lui rt, constant`
 - Copies 16-bit const to left 16 bits of rt; Clears right 16 bits of rt
- Example: how can we load 32 bit constant into `$s0`?

0000 0000 0111 1101 0000 1001 0000 0000

`lui $s0, 61`

| | |
|---------------------|---------------------|
| 0000 0000 0111 1101 | 0000 0000 0000 0000 |
|---------------------|---------------------|

`ori $s0, $s0, 2304`

| | |
|---------------------|---------------------|
| 0000 0000 0111 1101 | 0000 1001 0000 0000 |
|---------------------|---------------------|

- Typically assembler is responsible for breaking a large constant and reassembling into a register (using the reserved register `$at`)

Branch Addressing

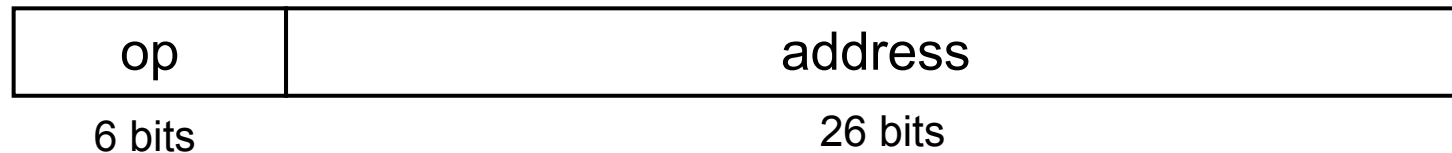
- Branch instructions specify
 - Opcode, two registers, target address
- If addresses had to fit in 16-bits, then no program could be bigger than 2^{16}



- (Pseudo伪)PC-relative addressing
 - Target address = PC + offset \times 4
 - PC already incremented by 4 by this time
 - Almost all loops and *if* statements $< 2^{16}$ words

Jump Addressing

- Jump (**j** and **j al**) targets could be anywhere in text segment
 - Encode full address in instruction



- Direct jump addressing
 - Target address = $PC_{31...28} : (\text{address} \times 4)$

Target Addressing Example

- Loop code from earlier example
 - Assume Loop at location 80000

```

Loop: sll    $t1, $s3, 2      80000
      add    $t1, $t1, $s6    80004
      lw     $t0, 0($t1)      80008
      bne    $t0, $s5, Exit   80012
      addi   $s3, $s3, 1      80016
      j      Loop            80020
Exit: ...                    80024
    
```

| | | | | | |
|------------------|-------|----|---|---|----|
| 0000:(20000 * 4) | | | | | |
| 0 | 0 | 19 | 9 | 4 | 0 |
| 0 | 9 | 22 | 9 | 0 | 32 |
| 35 | 9 | 8 | 0 | | |
| 5 | 8 | 21 | 2 | | |
| 8 | 19 | 19 | 1 | | |
| 2 | 20000 | | | | |

Add 2 words to address of following instruction

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example: how can we increase branching distance for the following instruction?

beq \$s0, \$s1, L1



bne \$s0, \$s1, L2

j L1

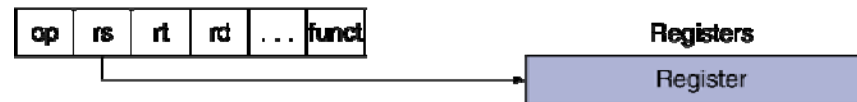
L2: ...

Addressing Mode Summary

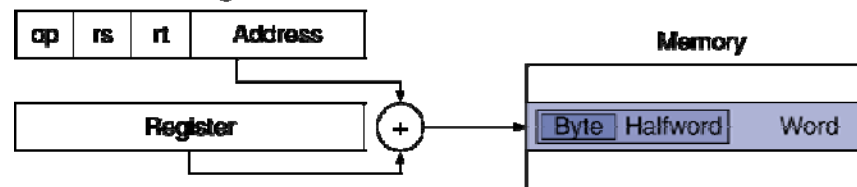
1. Immediate addressing



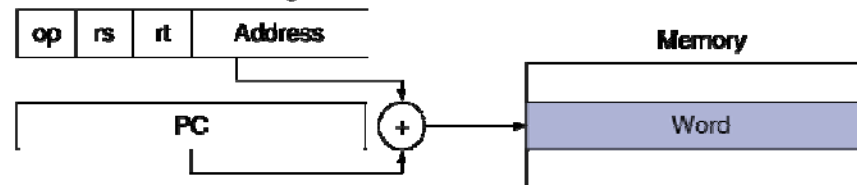
2. Register addressing



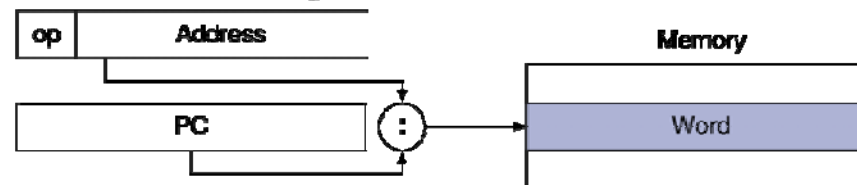
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



Concluding Remarks

- Design principles
 1. Simplicity favors regularity
 2. Smaller is faster
 3. Make the common case fast
 4. Good design demands good compromises
- Layers of software/hardware
 - Compiler, assembler, hardware
- MIPS: typical of RISC ISAs
 - c.f. x86

Concluding Remarks

- Measured MIPS instruction execution frequencies in benchmark programs
 - Consider making the common case fast

| Instruction class | MIPS examples | SPEC2006 Int | SPEC2006 FP |
|-------------------|--------------------------------------|--------------|-------------|
| Arithmetic | add, sub, addi | 16% | 48% |
| Data transfer | lw, sw, lb, lbu, lh, lhu, sb, lui | 35% | 36% |
| Logical | and, or, nor, andi, ori, sll, srl | 12% | 4% |
| Cond. Branch | beq, bne, slt, slti, sltiu | 34% | 8% |
| Jump | j, jr, jal | 2% | 0% |

Computer Organization and Design

3.1-3.6

Arithmetic for Computers

Computer Organization and Design

4.1-4.4

The Processor

定点数的加减运算及实现

- 一、补码加减运算及运算器
- 二、机器数的移位运算

一、补码加减运算及运算器

- 1、补码加减运算方法
- 2、补码加减运算的溢出判断
- 3、补码加减运算器

1、补码加减运算方法

- 补码的加减运算的公式是：
 - $[X+Y]_{\text{补}} = [X]_{\text{补}} + [Y]_{\text{补}}$
 - $[X-Y]_{\text{补}} = [X]_{\text{补}} + [-Y]_{\text{补}}$
- 特点：
 - 使用补码进行加减运算，符号位和数值位一样参加运算。
 - 补码的减法可以用加法来实现，任意两数之差的补码等于被减数的补码与减数相反数的补码之和。

求补运算： $[Y]_{\text{补}} \rightarrow [-Y]_{\text{补}}$

- 求补规则：将 $[Y]_{\text{补}}$ 包括符号位在内每一位取反，末位加1。
- 若 $[Y]_{\text{补}} = Y_0, Y_1, \dots, Y_n$ ，则：

$$[-Y]_{\text{补}} = \overline{Y_0} \overline{Y_1} \dots \overline{Y_n} + 1$$

- ⊕ 若 $[Y]_{\text{补}} = Y_0.Y_1, \dots, Y_n$ ，则：

$$[-Y]_{\text{补}} = \overline{Y_0} \overline{Y_1} \dots \overline{Y_n} + 0.0 \dots 01$$

- ⊕ 例： $[X]_{\text{补}} = 0.1101$ ，则： $[-X]_{\text{补}} = 1.0011$

- ⊕ $[Y]_{\text{补}} = 1.1101$ ，则： $[-Y]_{\text{补}} = 0.0011$

补码加减运算举例

- 例：已知 $X=+1011$ ， $Y=-0100$ ，用补码计算 $X+Y$ 和 $X-Y$ 。

- 写出补码：

$$[X]_{\text{补}} = 0,1011 \quad [Y]_{\text{补}} = 1,1100 \quad [-Y]_{\text{补}} = 0,0100$$

- 计算：

$$\begin{array}{r} 0,1011 \\ + 1,1100 \\ \hline 0,0111 \end{array}$$

$$\blacklozenge [X + Y]_{\text{补}} = 0, 0111$$

$$\begin{array}{r} 0,1011 \\ + 0,0100 \\ \hline 0,1111 \end{array}$$

$$\blacklozenge [X - Y]_{\text{补}} = 0, 1111$$

2、补码加减运算的溢出判断

- 当运算结果超出机器数的表示范围时，称为溢出。计算机必须具备检测运算结果是否发生溢出的能力，否则会得到错误的结果。
- 对于加减运算，可能发生溢出的情况：同号（两数）相加，或者异号（两数）相减。
- 确定发生溢出的情况：
 - 正数相加，且结果符号位为1；
 - 负数相加，且结果符号位为0；
 - 正数 - 负数，且结果符号位为1；
 - 负数 - 正数，且结果符号位为0；

常用的判溢方法（补码加减运算）

■ （1）单符号位判溢方法

- 当最高有效位产生的进位和符号位产生的进位不同时，加减运算发生了溢出。

- $V = C_1 \oplus C_f$

■ （2）双符号位判溢方法

- X和Y采用双符号位补码参加运算，正数的双符号位为00，负数的双符号位为11；当运算结果的两符号位 S_{f1} 、 S_{f2} 不同时（01或10），发生溢出。

- $V = S_{f1} \oplus S_{f2} = X_{f1} \oplus Y_{f1} \oplus C_f \oplus S_f$

- $S_{f1} S_{f2} = 01$ ，则正溢出； $S_{f1} S_{f2} = 10$ ，则负溢出。

双符号位判溢方法举例

- 例：用补码计算 $X+Y$ 和 $X-Y$
 - (1) $X=+1000$, $Y=+1001$
 - (2) $X=-1000$, $Y=1001$

$$\begin{array}{r} [X]_{\text{补}} \quad 00, 1000 \\ + [Y]_{\text{补}} \quad 00, 1001 \\ \hline [X+Y]_{\text{补}} \quad 01, 0001 \\ S_{f1} S_{f2}=01, \text{正溢出} \end{array}$$

$$\begin{array}{r} [X]_{\text{补}} \quad 11, 1000 \\ + [Y]_{\text{补}} \quad 00, 1001 \\ \hline [X+Y]_{\text{补}} \quad 00, 0001 \\ S_{f1} S_{f2}=00, \text{无溢出} \end{array}$$

$$\begin{array}{r} [X]_{\text{补}} \quad 00, 1000 \\ + [-Y]_{\text{补}} \quad 11, 0111 \\ \hline [X-Y]_{\text{补}} \quad 11, 1111 \\ S_{f1} S_{f2}=11, \text{无溢出} \end{array}$$

$$\begin{array}{r} [X]_{\text{补}} \quad 11, 1000 \\ + [-Y]_{\text{补}} \quad 11, 0111 \\ \hline [X-Y]_{\text{补}} \quad 10, 1111 \\ S_{f1} S_{f2}=10, \text{负溢出} \end{array}$$

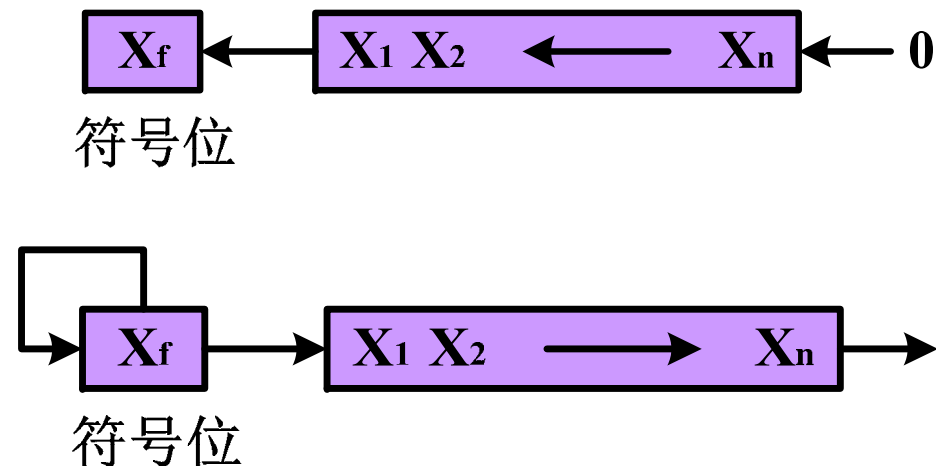
二、机器数的移位运算

- 二进制数据（真值）每相对于小数点左移一位，相当于乘以2；每相对于小数点右移一位，相当于除以2。
- 计算机中的移位运算分为：
 - 1、逻辑移位：将移位的数据视为无符号数据，各数据位在位置上发生了变化，导致无符号数据的数值（无正负）放大或缩小。
 - 2、算术移位：将移位的数据视为带符号数据（机器数）。算术移位的结果，在数值的绝对值上进行放大或缩小，同时，符号位必须保持不变。
 - 3、循环移位：所有的数据位在自身范围内进行左移或者右移，左移时最高位移入最低位，右移时最低位移入最高位。

补码的算术移位

- **算术左移**：符号位不变，高位移出，低位补0。
 - 为保证补码算术左移时不发生溢出，**移位的数据最高有效位必须与符号位相同**。
 - **在不发生溢出的前提下**，用硬件实现补码的算术左移时，直接将数据最高有效位移入符号位，不会改变机器数的符号。

⊕ **算术右移**：符号位不变，低位移出，高位正数补0，负数补1，即高位补符号位。



补码的算术移位举例

■ 例：设 $X = 0.1001$, $Y = -0.0101$ ，求

- $[X]_{\text{补}} = 0.1001$
- $[2X]_{\text{补}} = 1.0010$ (溢出)
- $[X/2]_{\text{补}} = 0.0100$
- $[Y]_{\text{补}} = 1.1011$
- $[2Y]_{\text{补}} = 1.0110$
- $[Y/2]_{\text{补}} = 1.1101$

定点数的乘法运算及实现

- 一、计算机中乘除运算的实现方法
- 二、原码乘法算法
- 三、原码乘法的硬件实现

二、原码乘法算法

1、手工乘法算法

- 手工计算 1011×1101 ，步骤：
- 手工算法：对应每1位乘数求得1项位积，并将位积逐位左移，然后将所有的位积一次相加，得到最后的乘积。
- 乘法的机器算法：从乘数的最低位开始，每次根据乘数位得到其位积，乘数位为0，位积为0，乘数位为1，则位积为被乘数；用原部分积右移1位加上本次位积，得新部分积；初始部分积为0。

$$\begin{array}{r} 1011 \\ \times 1101 \\ \hline 1011 \\ 0000 \\ 1011 \\ 1011 \\ \hline 10001111 \end{array}$$

二、原码乘法算法

■ 2、原码一位乘法算法：

假设 $[X]_{\text{原}} = X_s X_1 X_2 \dots X_n$ ， $[Y]_{\text{原}} = Y_s Y_1 Y_2 \dots Y_n$ ， $P = X \cdot Y$ ， P_s 是积的符号：

- 符号位单独处理 $P_s = X_s \oplus Y_s$
- 绝对值进行数值运算 $|P| = |X| \cdot |Y|$

■ 例如： $X = +1011$ ， $Y = -1101$ ，用原码一位乘法计算 $P = X \cdot Y$ 。

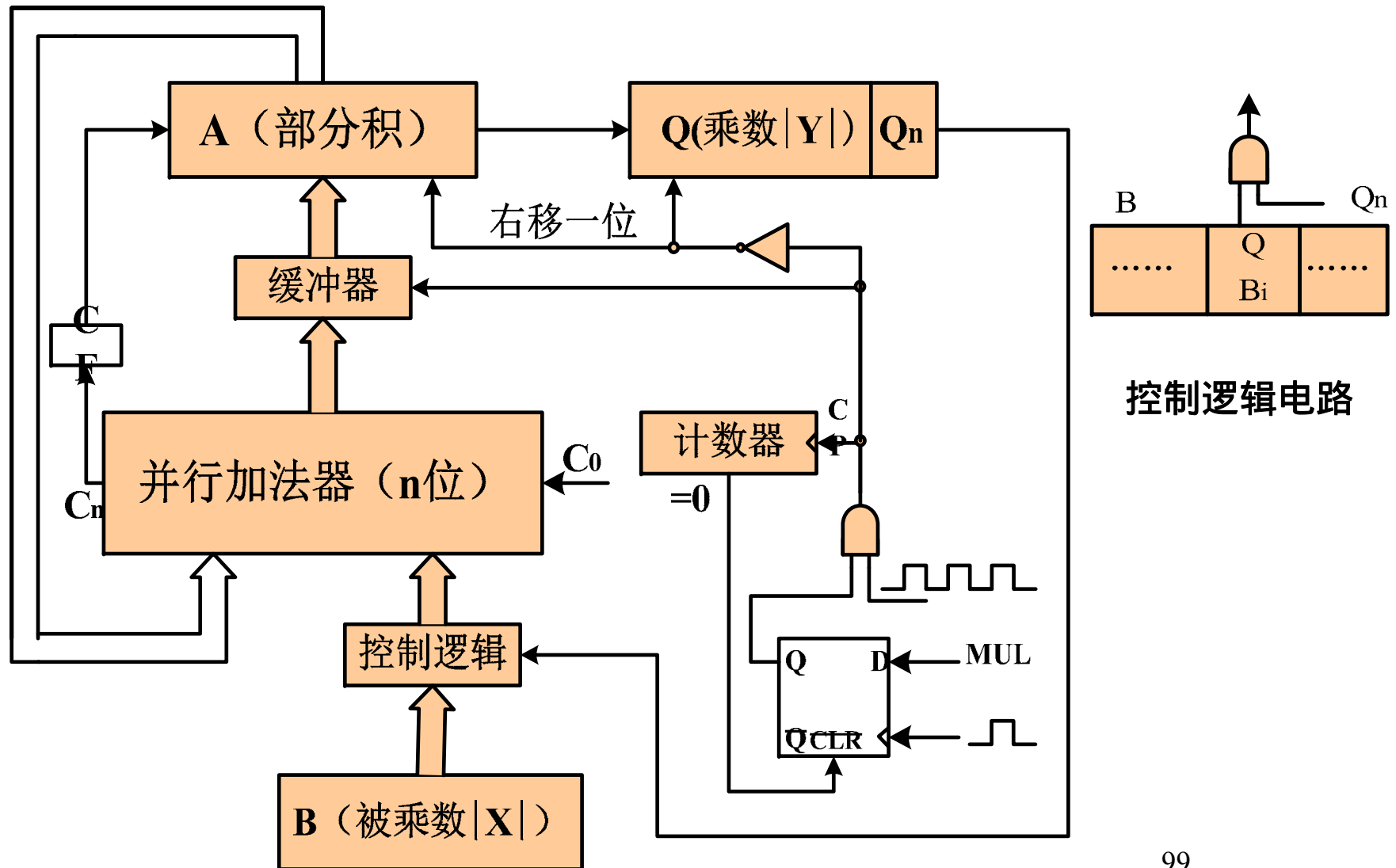
举例

- $[X]_{\text{原}} = 0, 1011$
- $[Y]_{\text{原}} = 1, 1101$
- $P_s = X_s \ Y_s$
 $= 0 \ 1 = 1$
- $|P| = |X| \cdot |Y|$

$[P]_{\text{原}} = 1, 10001111$

| 部分积 | 乘数Y | 操作说明 |
|-----------|----------------|---------------|
| 0, 0000 | 1 1 0 <u>1</u> | |
| + 0, 1011 | | $Y_4=1, + X $ |
| 0, 1011 | | |
| 0, 0101 | 1 1 1 <u>0</u> | 右移一位 |
| + 0, 0000 | | $Y_3=0, +0$ |
| 0, 0101 | | |
| 0, 0010 | 1 1 1 <u>1</u> | 右移一位 |
| + 0, 1011 | | $Y_2=1, + X $ |
| 0, 1101 | | |
| 0, 0110 | 1 1 1 <u>1</u> | 右移一位 |
| + 0, 1011 | | $Y_1=1, + X $ |
| 1, 0001 | | |
| 0, 1000 | 1 1 1 1 | 右移一位 |

三、原码乘法的硬件实现

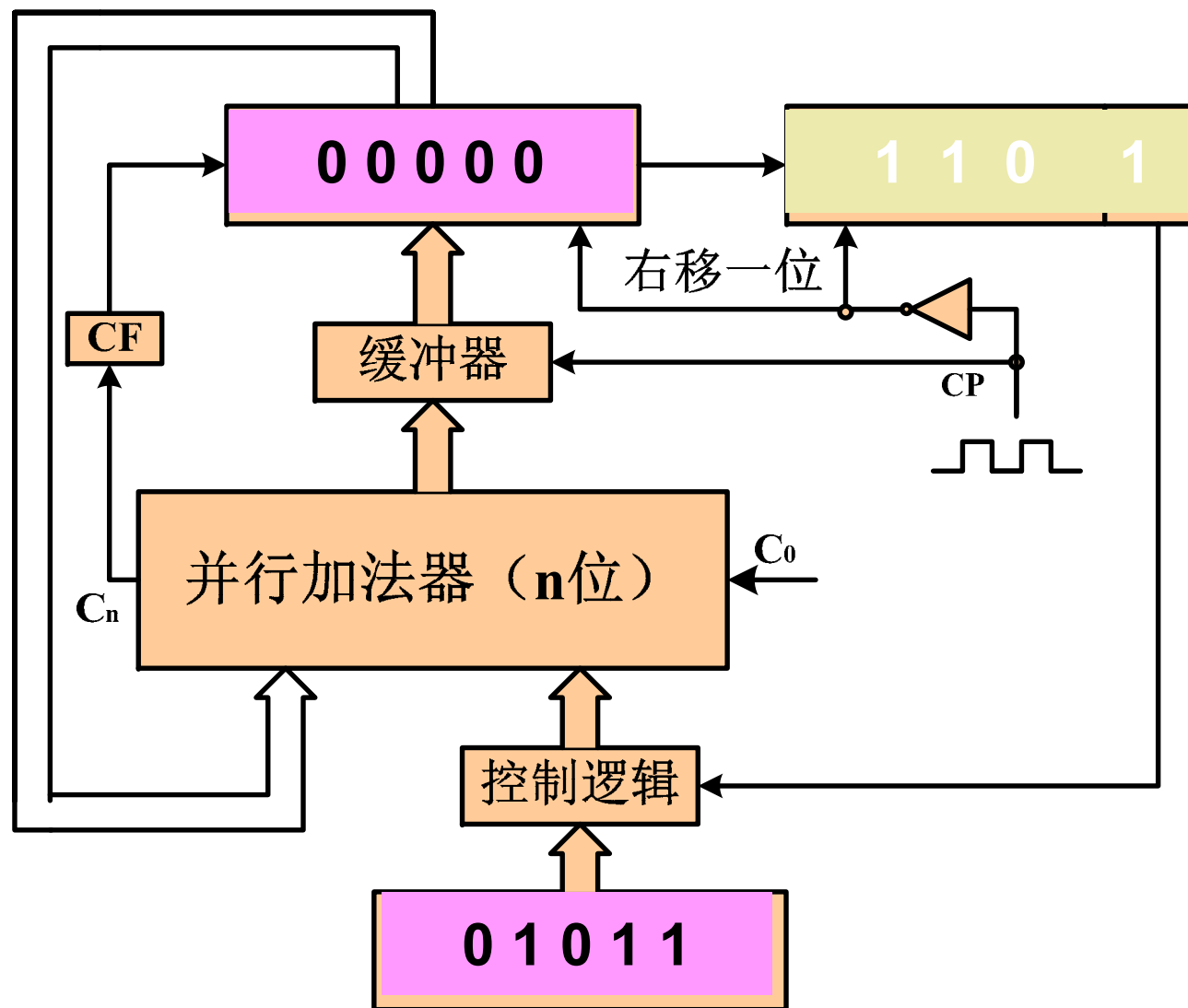


原码一位乘法

为各寄存器给初值

00000

1101



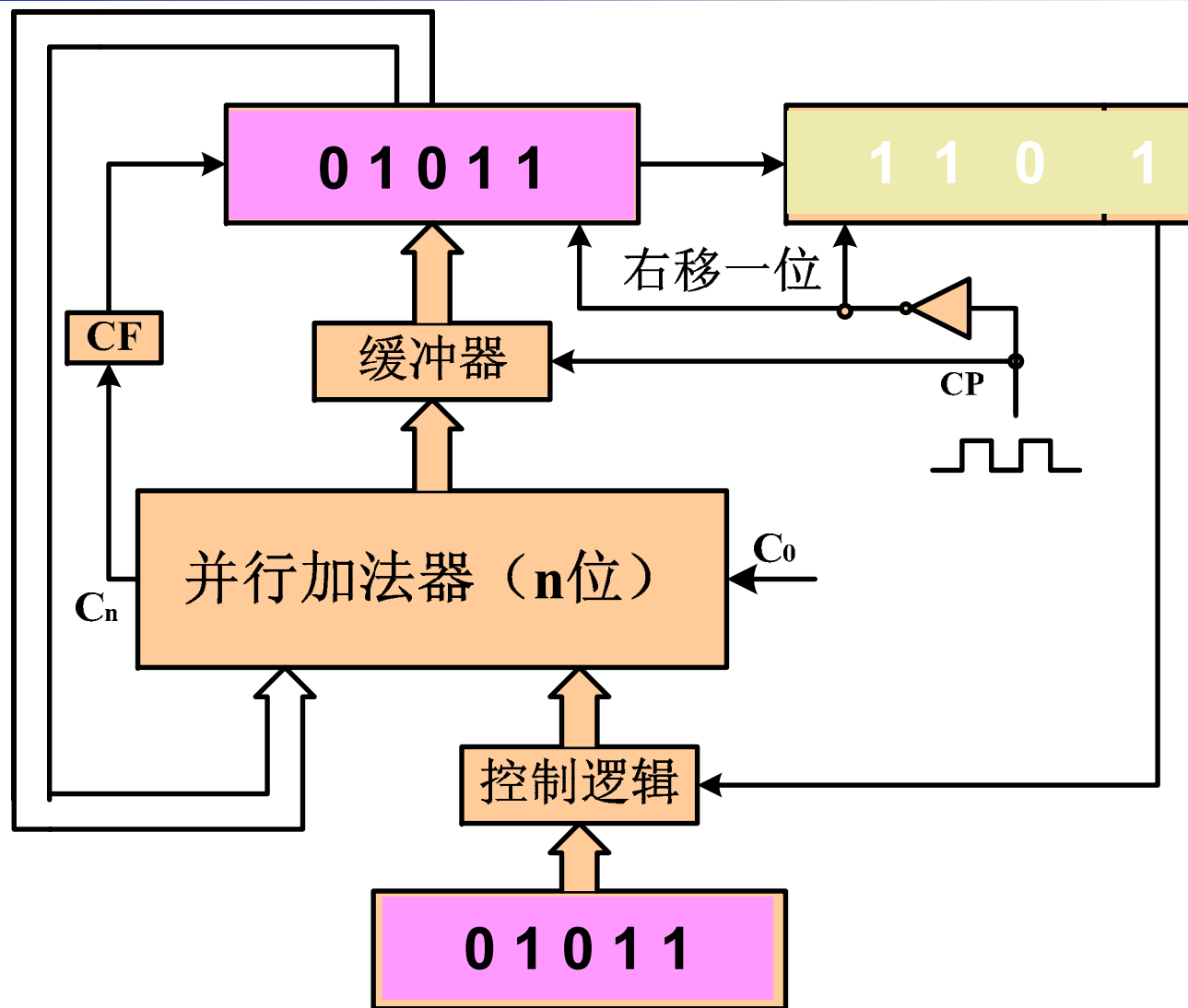
第一次求部分积 加运算: $+|X|$

00000

1101

01011

1101



第一次求部分积

右移1位

00000

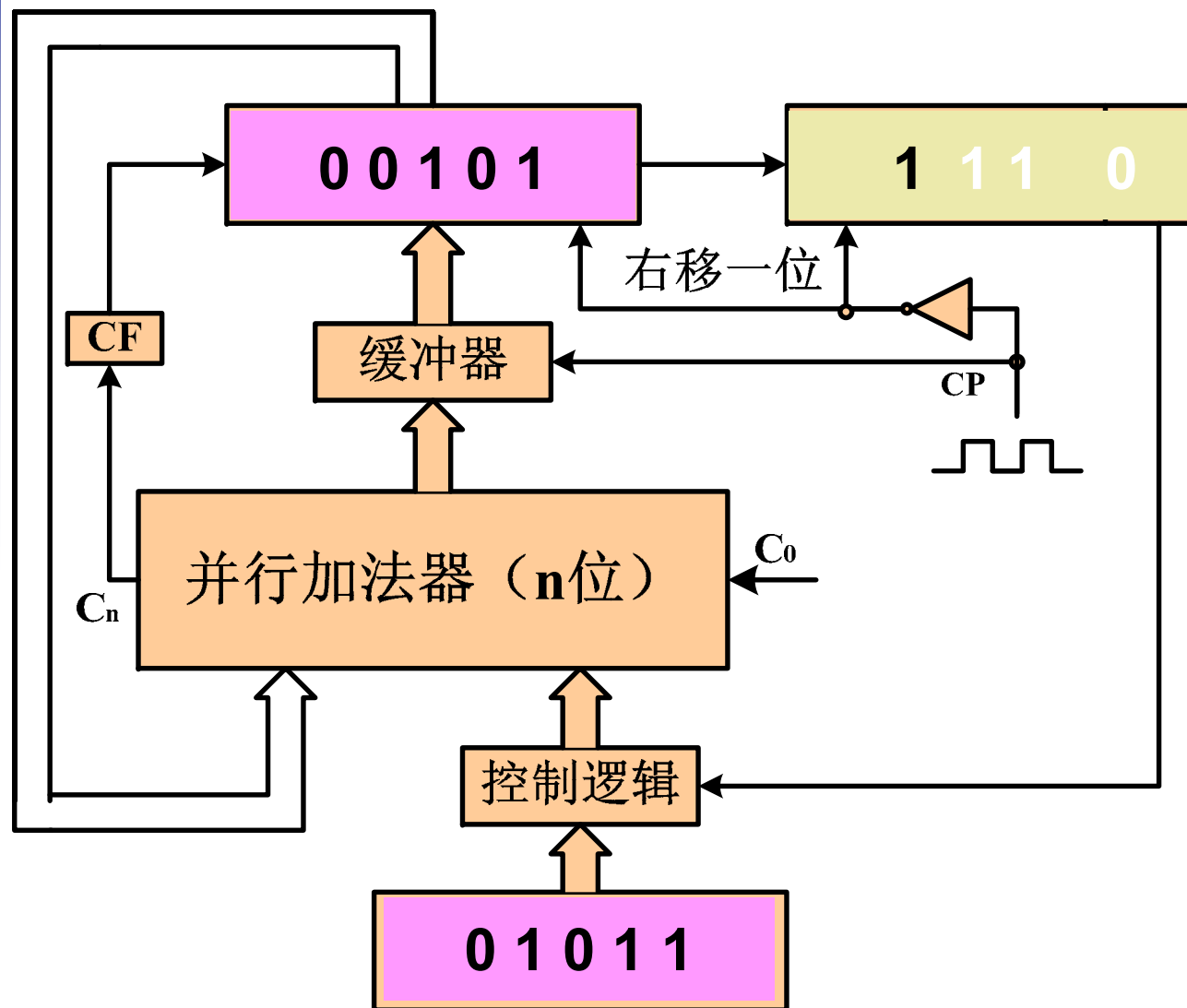
1101

01011

1101

00101

1110



第二次求部分积

加运算：+ 0

00000

1101

01011

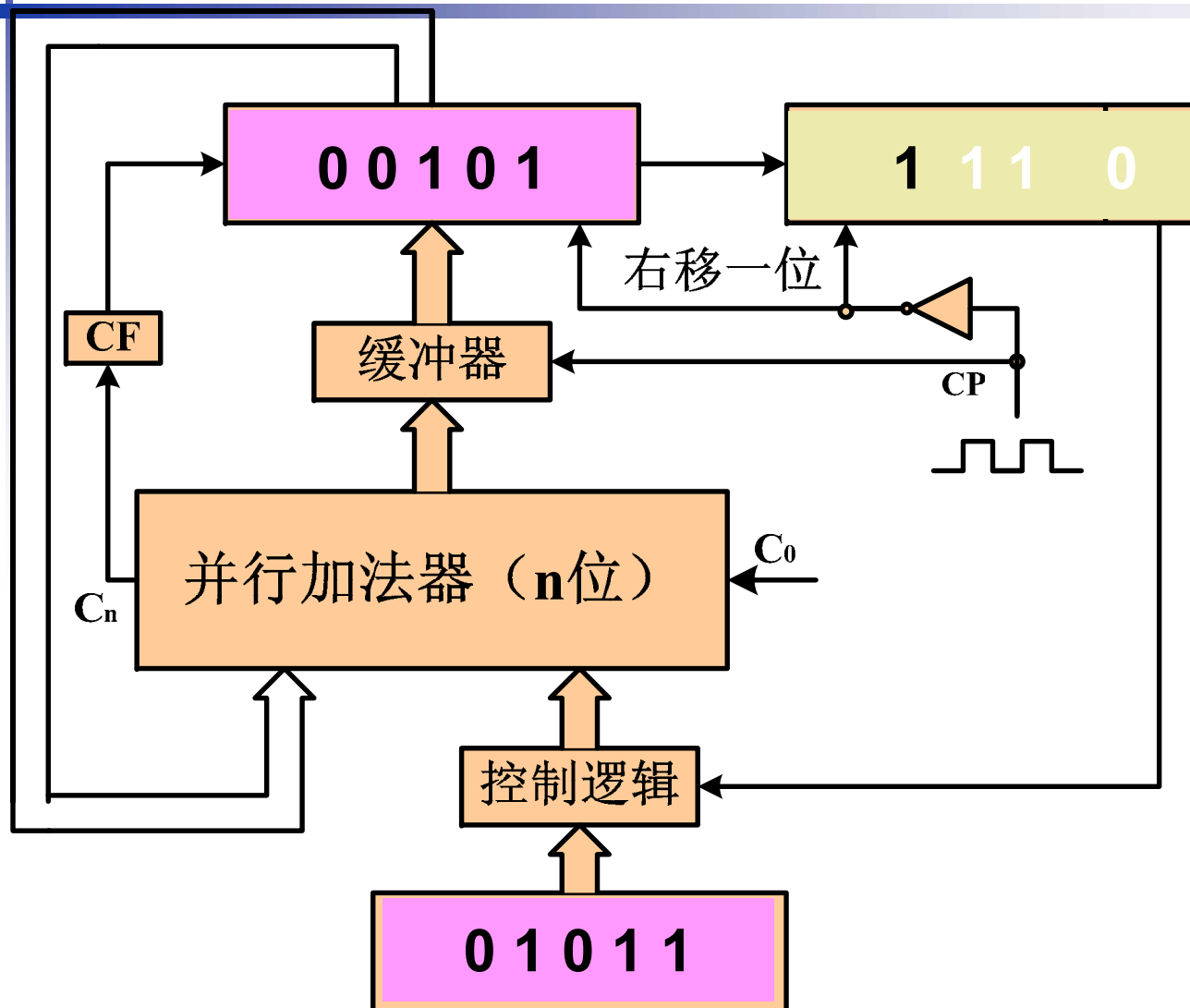
1101

00101

1110

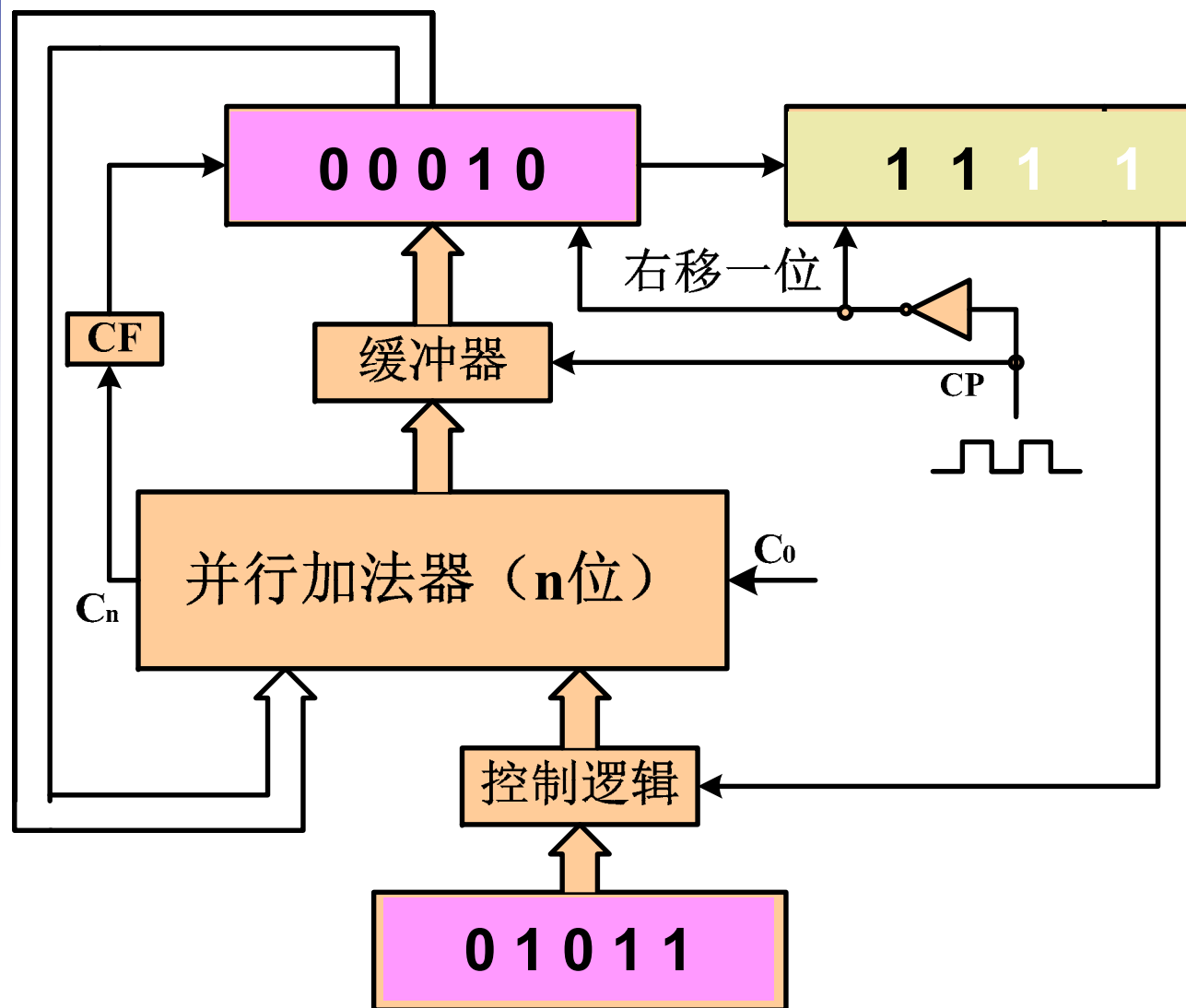
00101

1110



第二次求部分积

右移1位



00000

1101

01011

1101

00101

1110

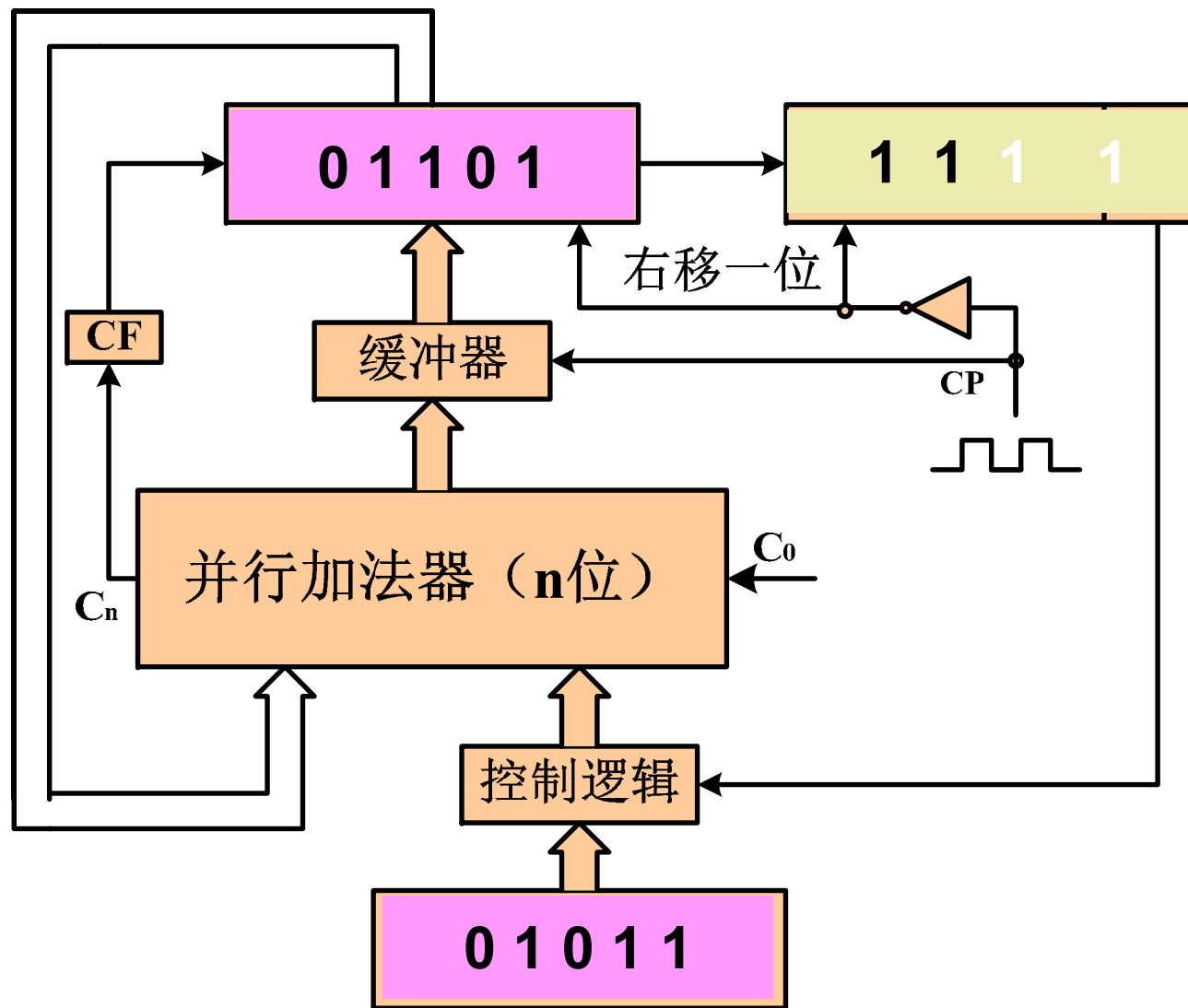
00101

1110

00010

1111

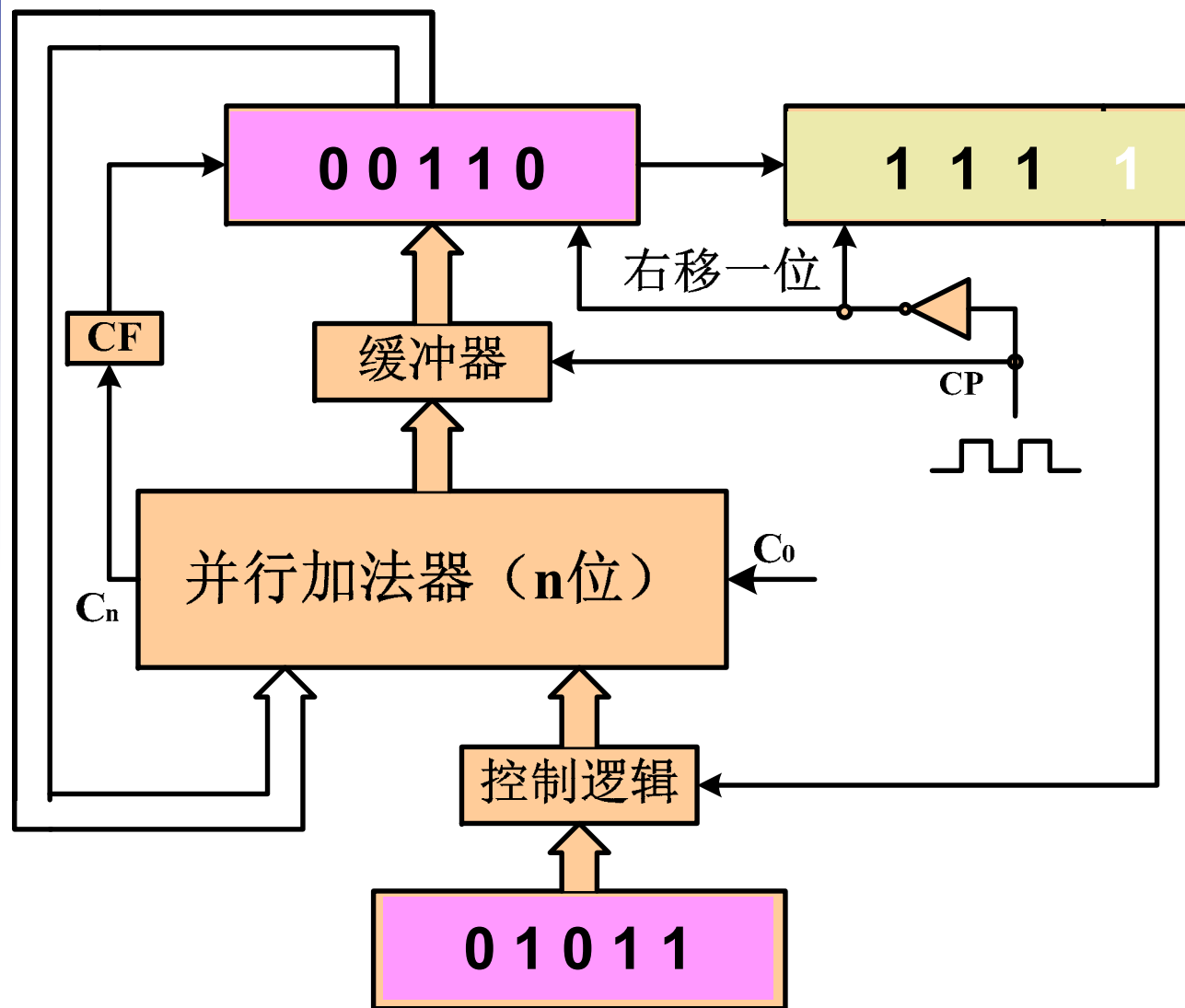
第三次求部分积 加运算：+|X|



| | |
|-------|------|
| 00000 | 1101 |
| 01011 | 1101 |
| 00101 | 1110 |
| 00101 | 1110 |
| 00010 | 1111 |
| 01101 | 1111 |

第三次求部分积

右移1位



00000

1101

01011

1101

00101

1110

00101

1110

00010

1111

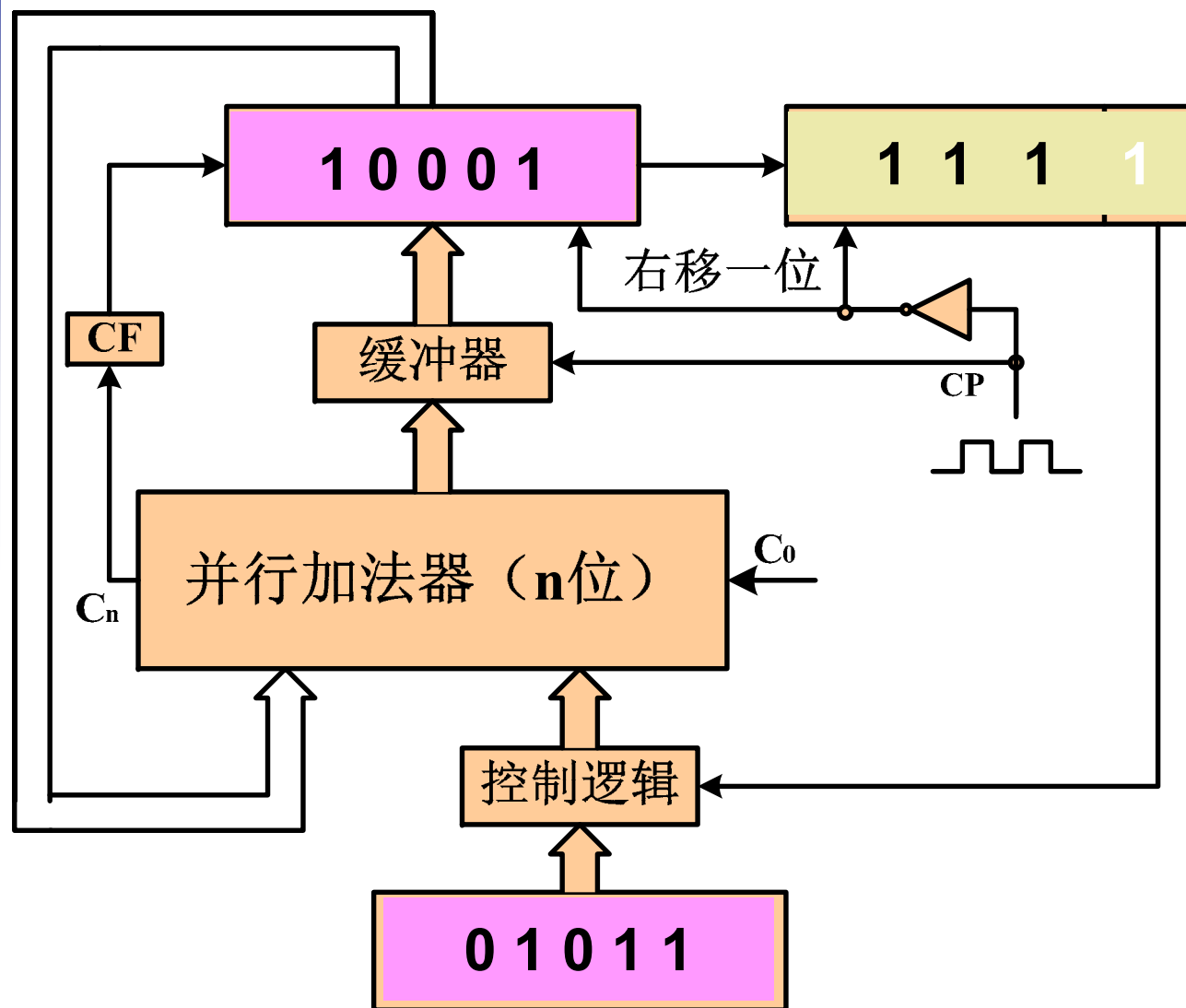
01101

1111

00110

1111

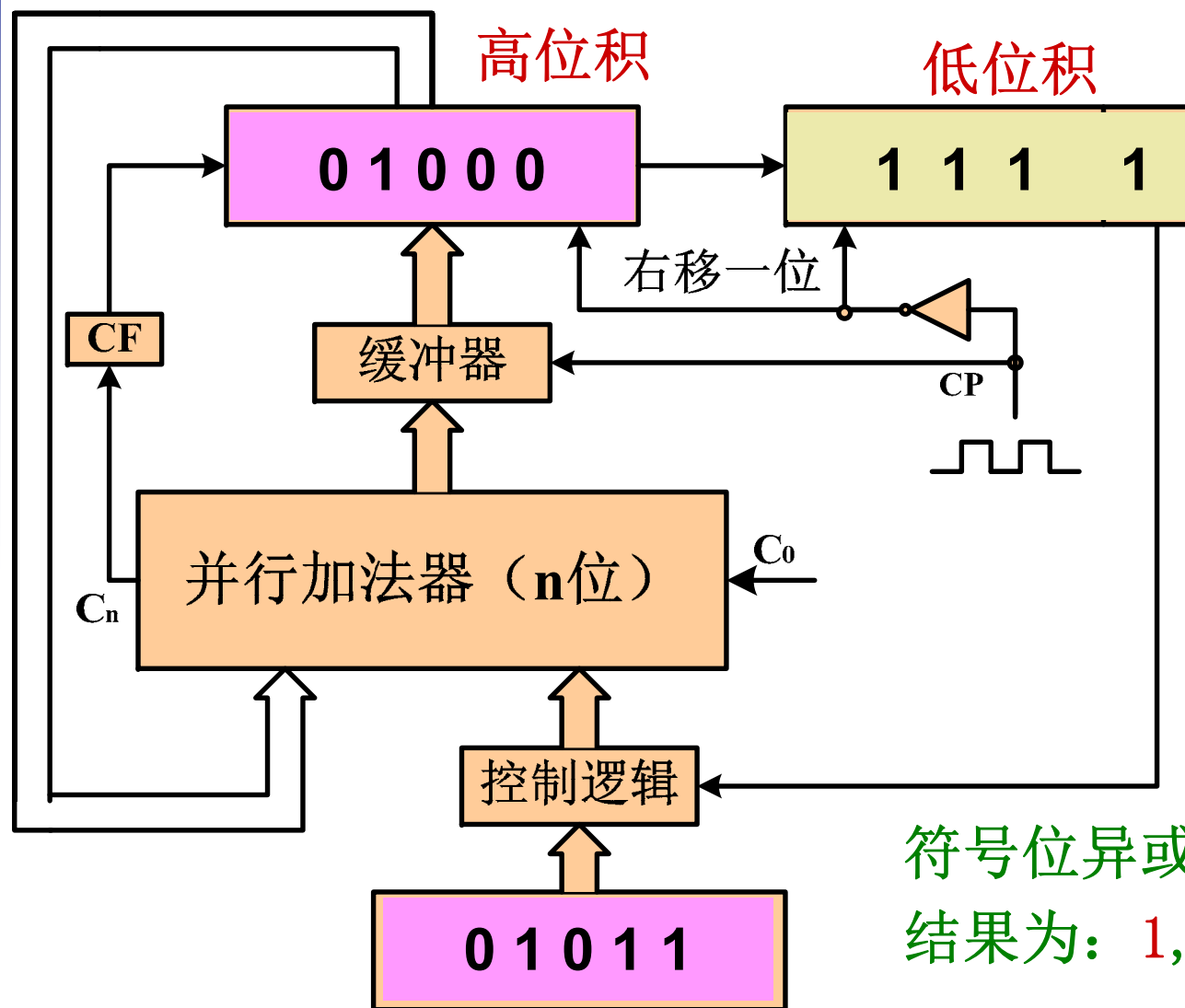
第四次求部分积 加运算: +|X|



| | |
|-------|------|
| 00000 | 1101 |
| 01011 | 1101 |
| 00101 | 1110 |
| 00101 | 1110 |
| 00010 | 1111 |
| 01101 | 1111 |
| 00110 | 1111 |
| 10001 | 1111 |

第四次求部分积

右移1位



| | |
|-------|------|
| 00000 | 1101 |
| 01011 | 1101 |
| 00101 | 1110 |
| 00101 | 1110 |
| 00010 | 1111 |
| 01101 | 1111 |
| 00110 | 1111 |
| 01000 | 1111 |

3.4 Division

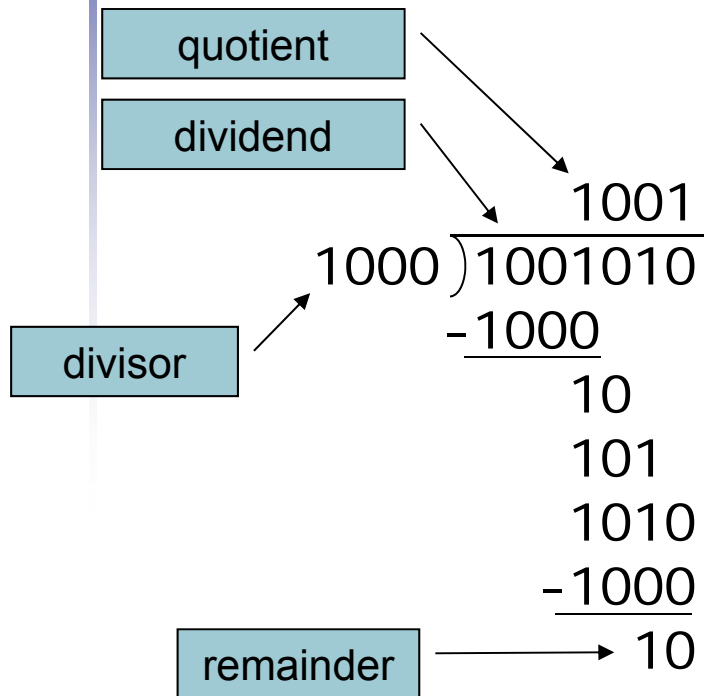
- 十进制整数除法回顾
- 二进制整数除法
- 原码1位除法思想
- 恢复余数除法及其优化
- 加减交替除法
- 定点小数除法
- 快速除法概况

3.4 Division-- Decimal Division

$$\begin{array}{r} 021 \\ 6 \overline{)128} \\ \underline{0} \\ 1 \\ 12 \\ \underline{12} \\ 0 \\ 08 \\ \underline{6} \\ 2 \end{array}$$

$$\begin{array}{r} 02 \\ 62 \overline{)128} \\ \underline{00} \\ 12 \\ 128 \\ \underline{124} \\ 4 \end{array}$$

Division -- Binary Division



n -bit operands yield n -bit quotient and remainder

计算机实现 n 位二进制除需解决的问题：

1) 余数（部分余数）与除数如何对齐？

固定的位数 $2*n$ (n) 位

2) 如何进行试商？

通过比较余数和除数大小(减法)

3) 计算（次数？）结束的条件？

设置固定 $n+1$ 次迭代

4) 如何处理符号位？

单独处理符号位

3.4 Division

- 十进制整数除法回顾
- 二进制整数除法
- 原码1位除法思想
- 恢复余数除法及其优化
- 加减交替除法
- 定点小数除法
- 快速除法概况

(1) 原码1位除法

设有两个n位定点整(小)数：

被除数：X，其原码为 $[x]_{\text{原}} = x_f \cdot x_{n-1} \cdots x_1 x_0$

除数：Y，其原码为 $[y]_{\text{原}} = y_f \cdot y_{n-1} \cdots y_1 y_0$

商：Q = X/Y, 其原码为

$$[Q]_{\text{原}} = (x_f \quad y_f) \cdot (x_{n-1} \cdots x_1 x_0 / y_{n-1} \cdots y_1 y_0)$$

原码1位除的思想为：

1) 商的符号运算 $q_f = x_f \quad y_f$

2) 商的数值运算，实质是两个正整数求商（余）的过程

恢复余数法和加减交替法

设计二进制正整数求商和余数的算法

假设 $X = x_{n-1} \dots x_1 x_0$, $Y = y_{n-1} \dots y_1 y_0$,

X/Y 的商为 $Q = q_{n-1} \dots q_i \dots q_1 q_0$

请同学们设计一个算法来计算商 Q 中的每一位取值 q_i ?

提示1 : 令 $X = Y * Q + R = Y * (q_{n-1} \dots q_1 q_0) + R$

$$X = q_{n-1} * Y * 2^{n-1} + \dots q_i * Y * 2^i + \dots q_1 * Y * 2 + q_0 * Y + R$$

提示2 : 如果 $X \geq Y * 2^{n-1}$, 那么 $q_{n-1} = 1$, 否则为 $q_{n-1} = 0$

正常使用主观题需2.0以上版本雨课堂

作答

正整数求商算法

假设 $X = Y * Q + R$, 其中 Q 为商 , R 为余数

算法 : 正整数求商和余数算法

输入 : n 位的二进制被除数 X 和除数 Y

输出 : 商 Q 和余数 R

step 0: 令 $t = n-1$, $Q = 0$; $R = X$

step 1. $R = R - Y * 2^t$

if $R \geq 0$

$q_t = 1$;

else $q_t = 0$; $R = R + Y * 2^t$

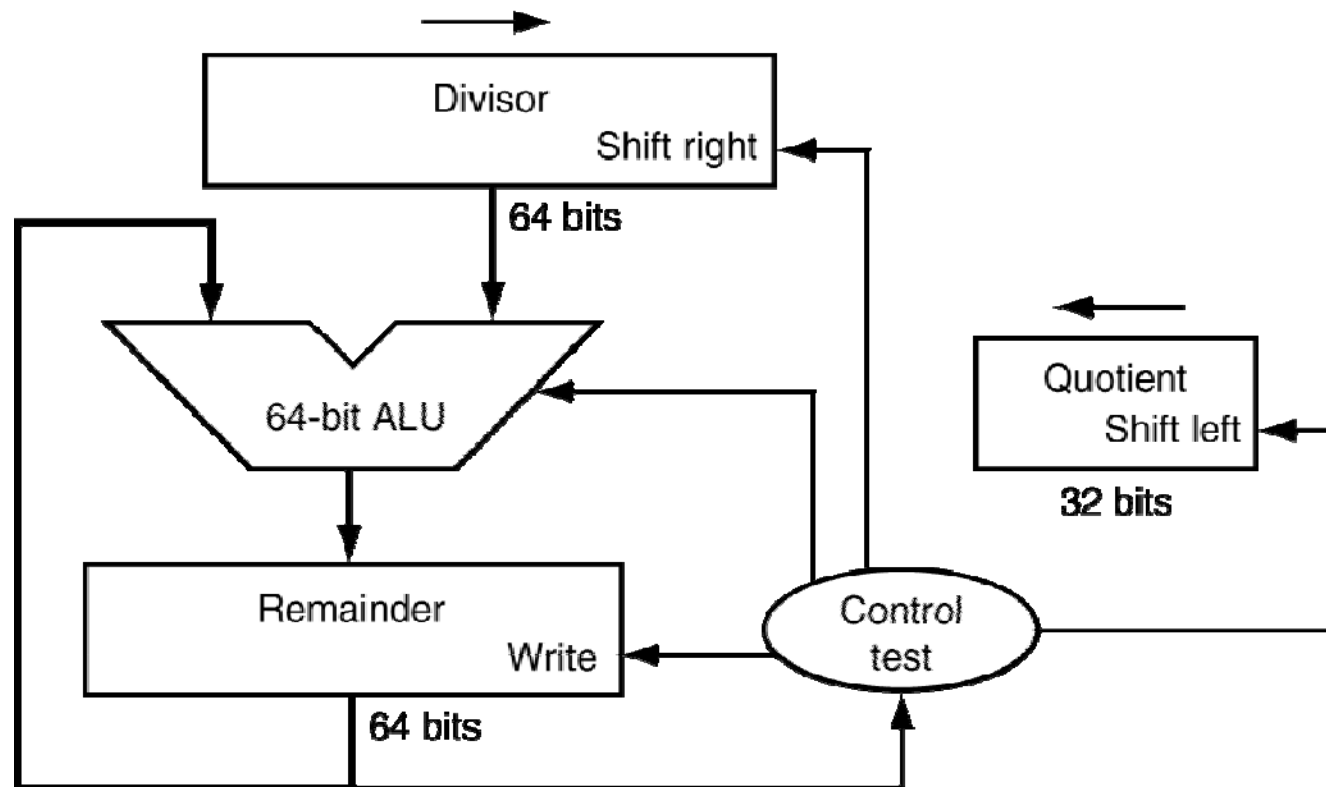
step 2. $t = t - 1$

step 3 if $t \geq 0$ goto step1

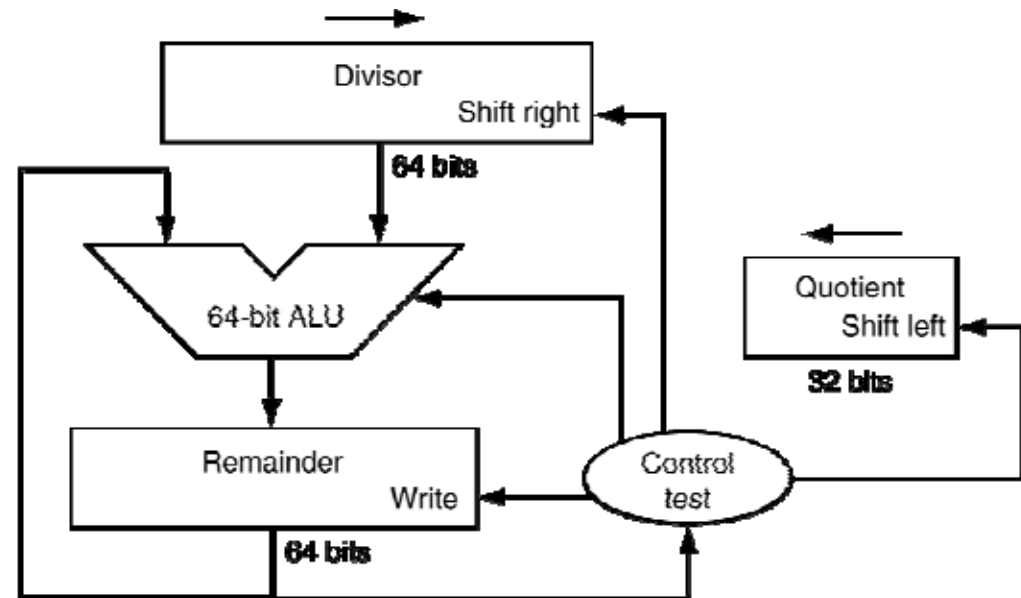
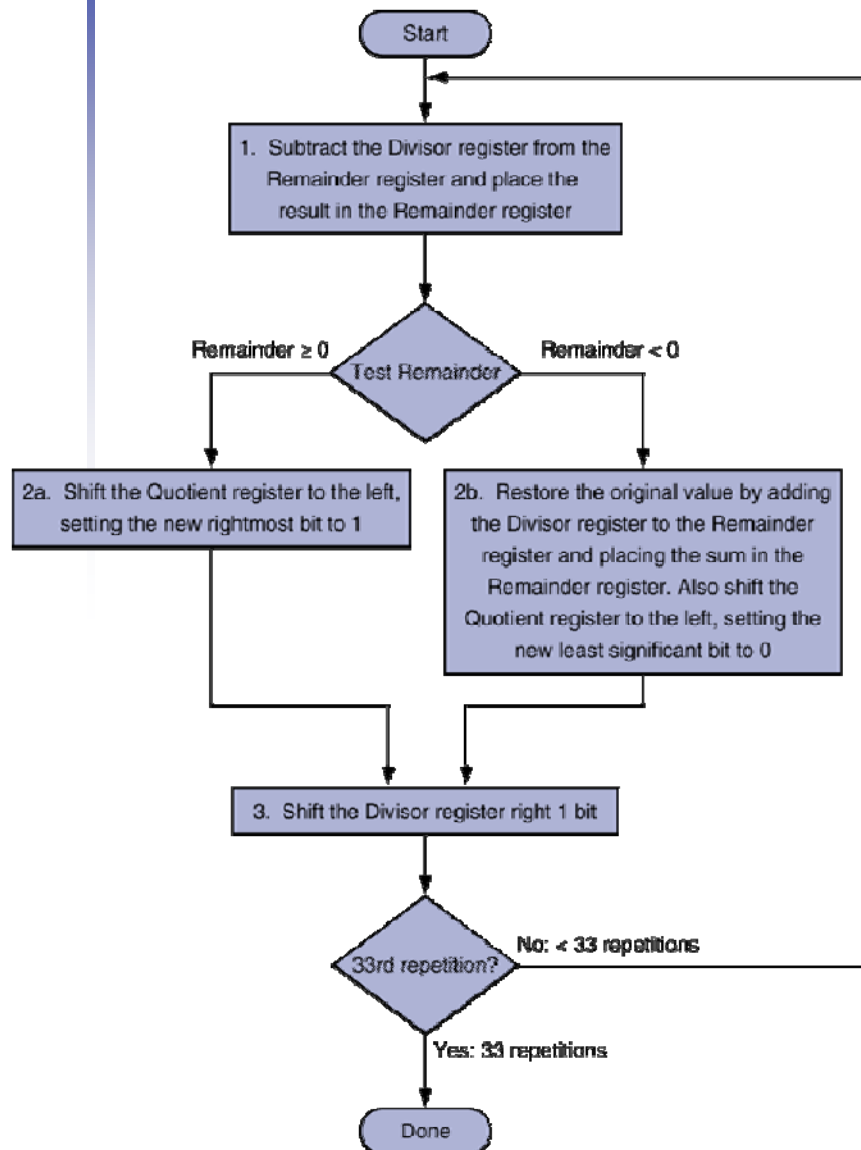
step 4 $R = X$

step 5 输出 R, Q

Integer Division Hardware



计算机中恢复余数算法流程



恢复余数法的基本特点

- **对齐方式**：除数和被除（余数）均为 2^n 位
除数左移 n 位扩展到 $2n$ 位，被除数则通过无符号扩展
- **用1试商**：每次试商1，即采用余数-除数
根据余数符号位判断是否有错。若有错误需要恢复余数。
- **移位处理**：除数右移1位，商左移1位置最低位**值**
- **运算次数**： n 位运算需要执行 $n+1$ 次计算

Demo

令

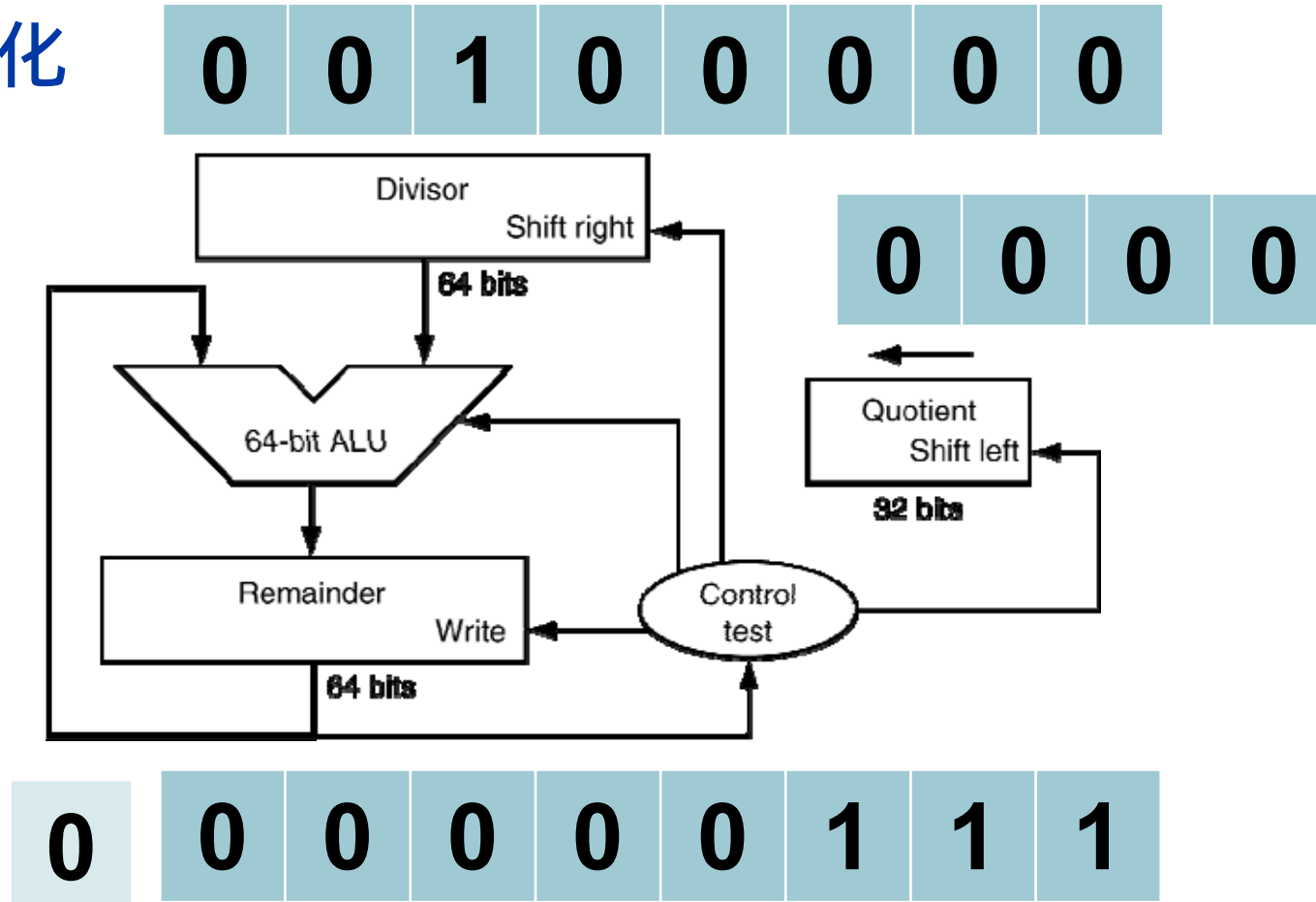
$$|x| = [0111]_2 \quad |y| = [0010]_2$$

求解 $|x/y|$

Integer Division Example

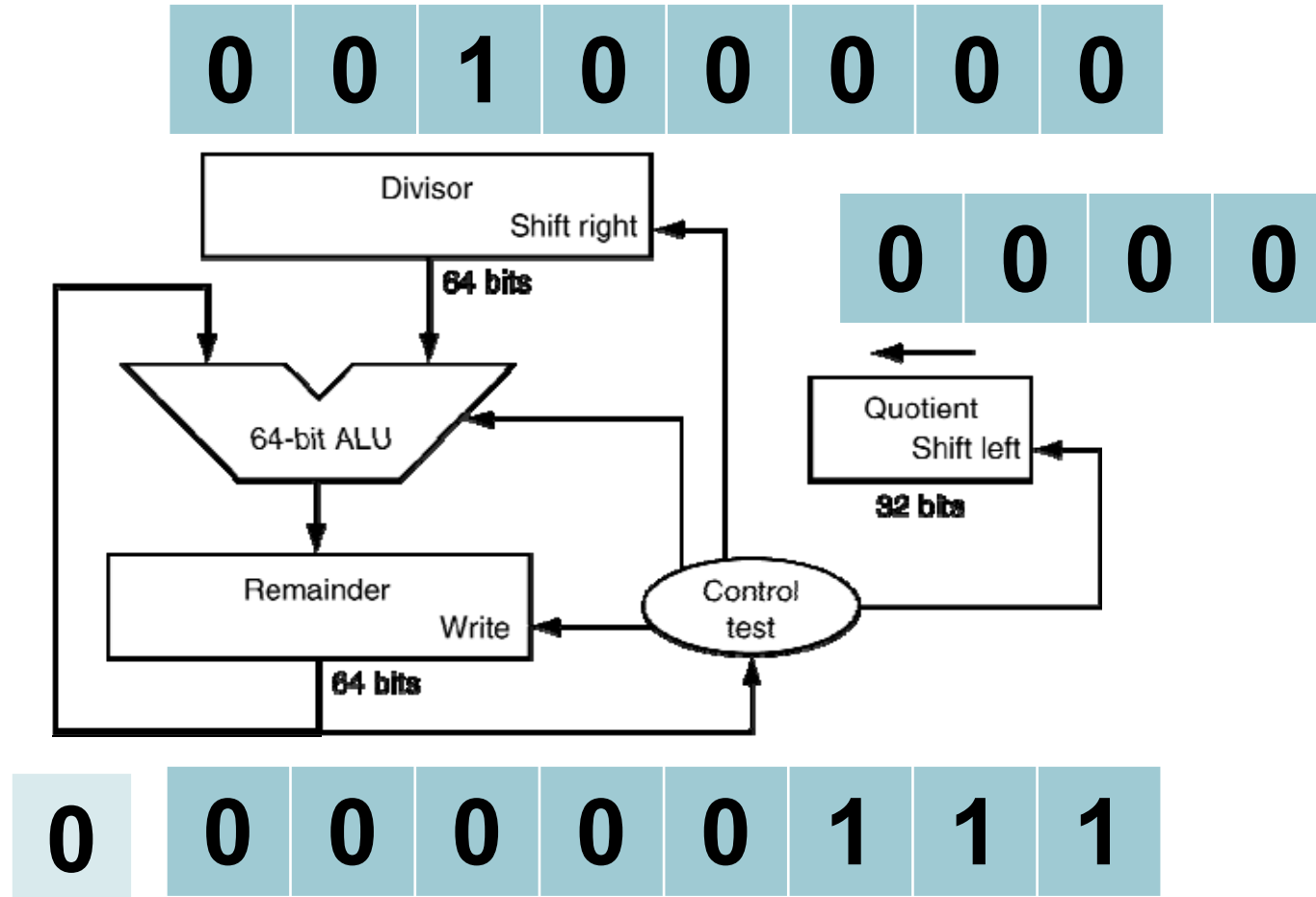
$$|x| = [0111]_2 \quad |y| = [0010]_2$$

初始化



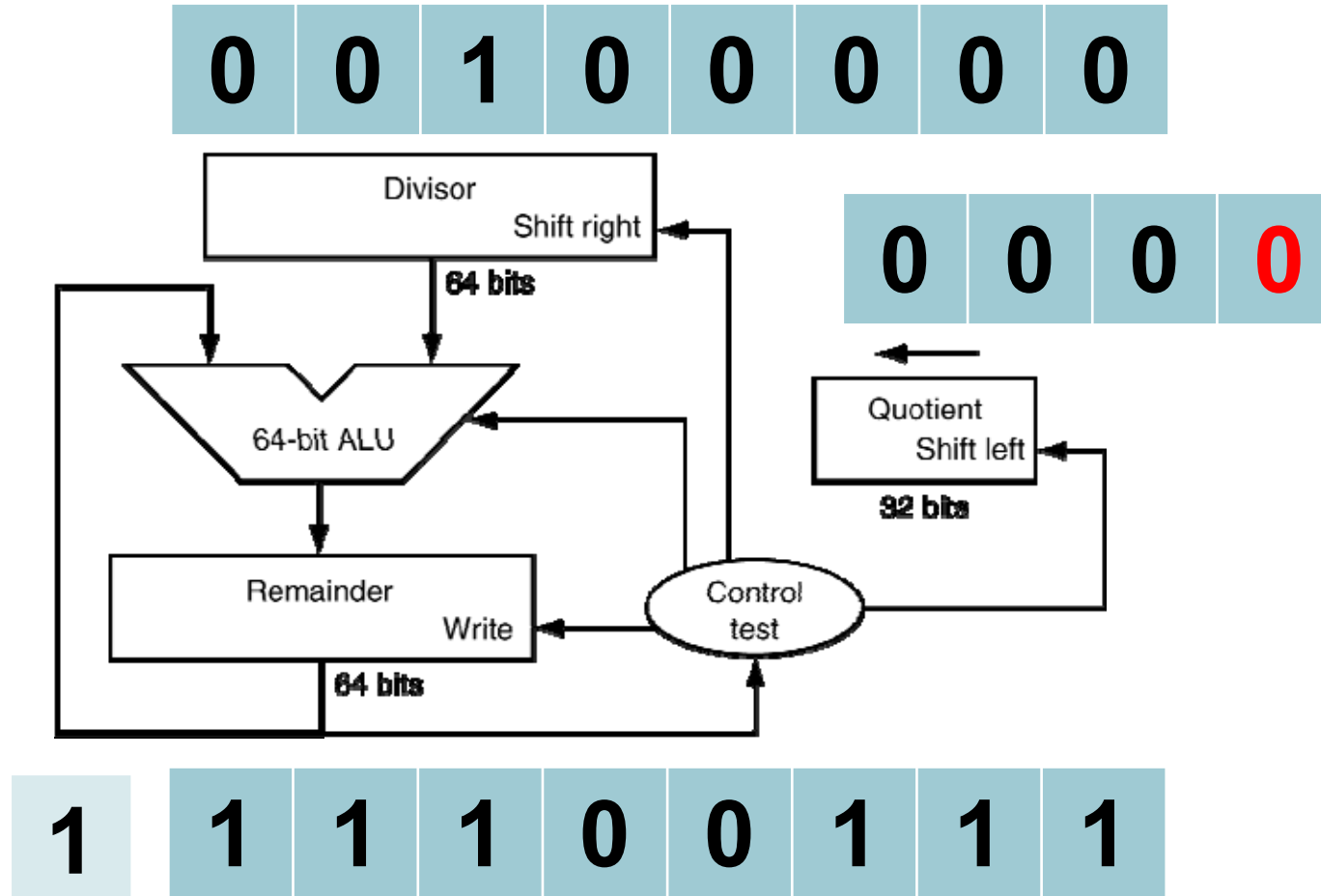
Integer Division Example

Step 1.1 余数-除数,试商



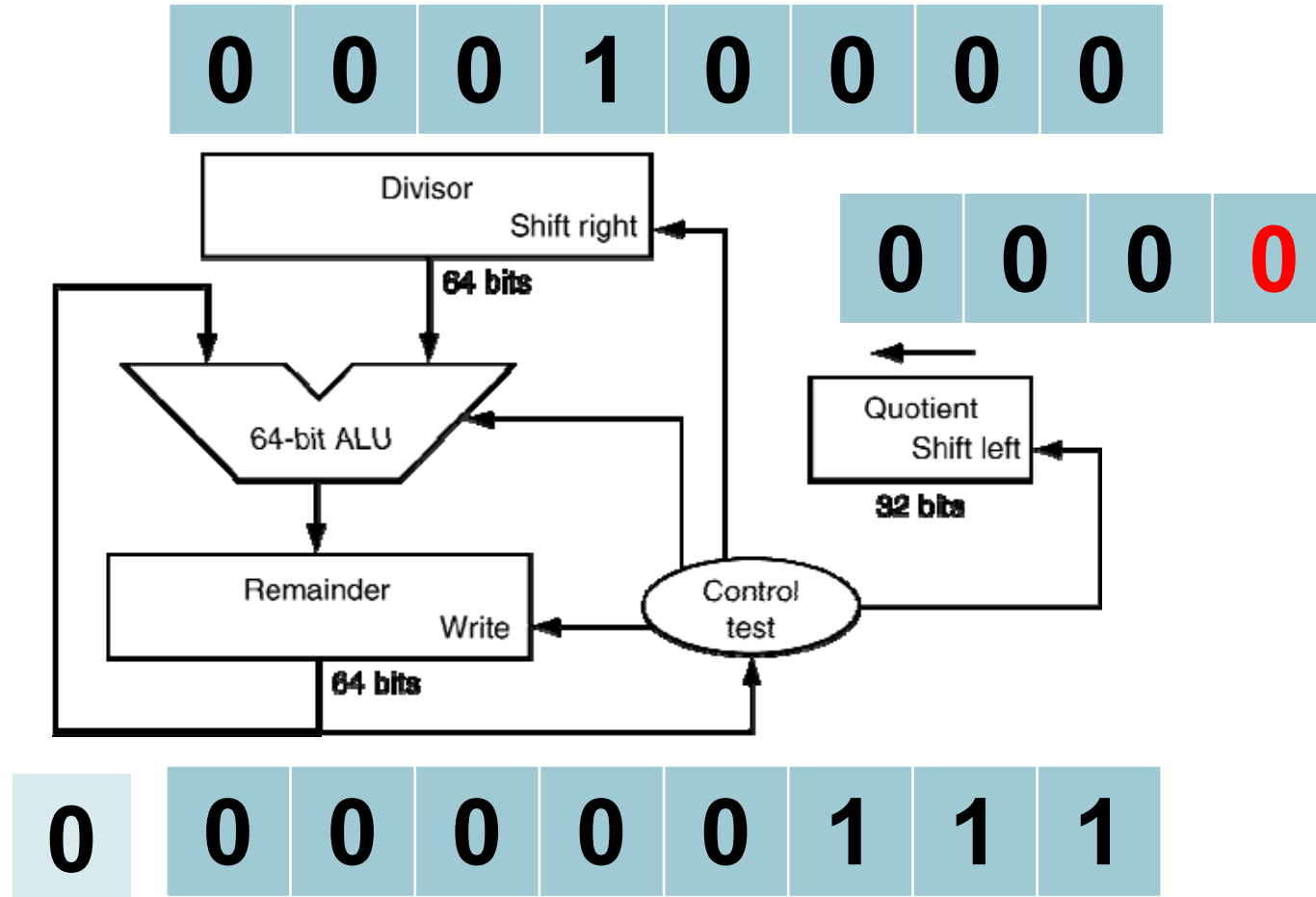
Integer Division Example

Step 1.2 余数<0,商左移1位, 置0



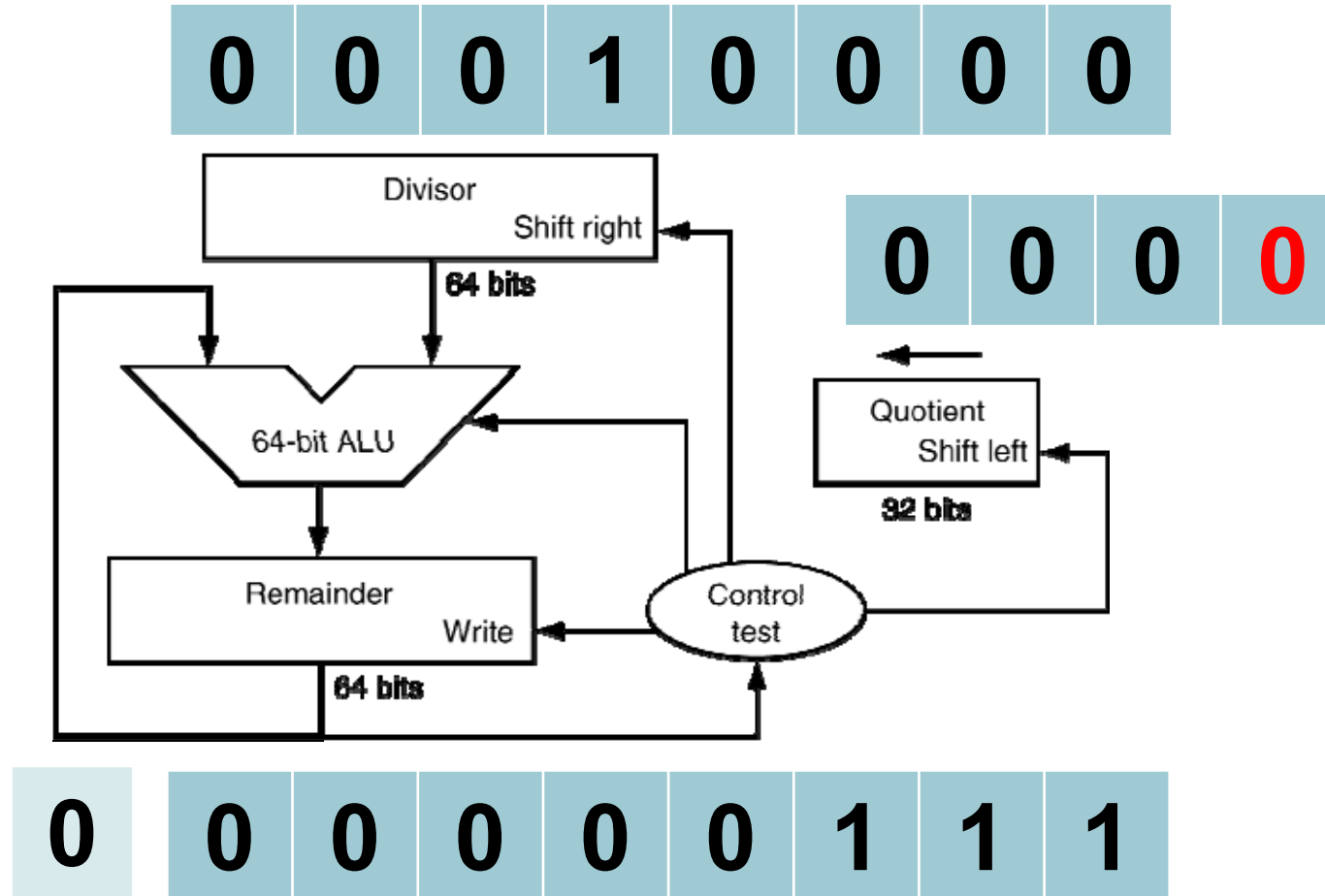
Integer Division Example

Step 1.3 恢复余数，除数右移1位



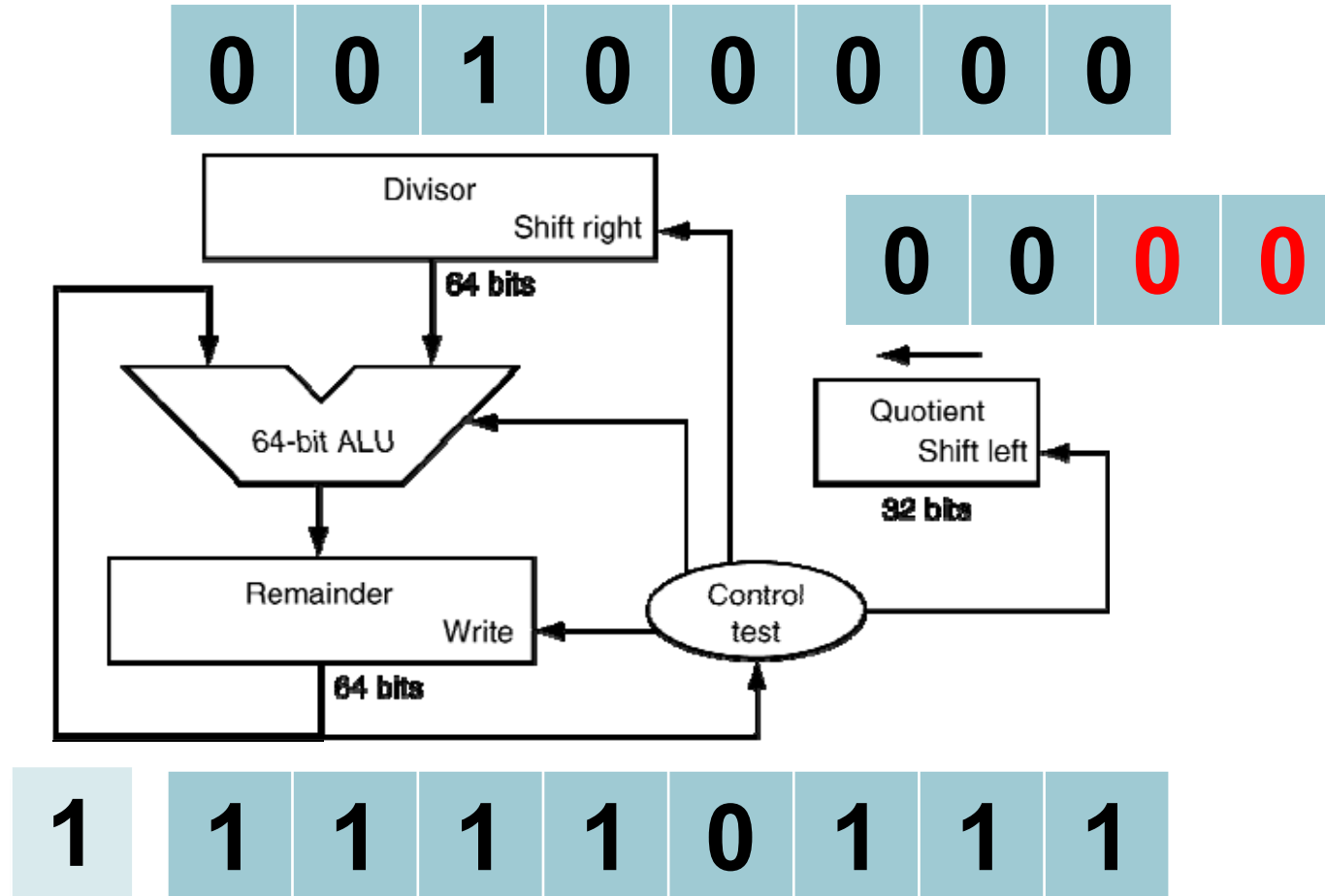
Integer Division Example

Step 2.1 余数-除数 试商



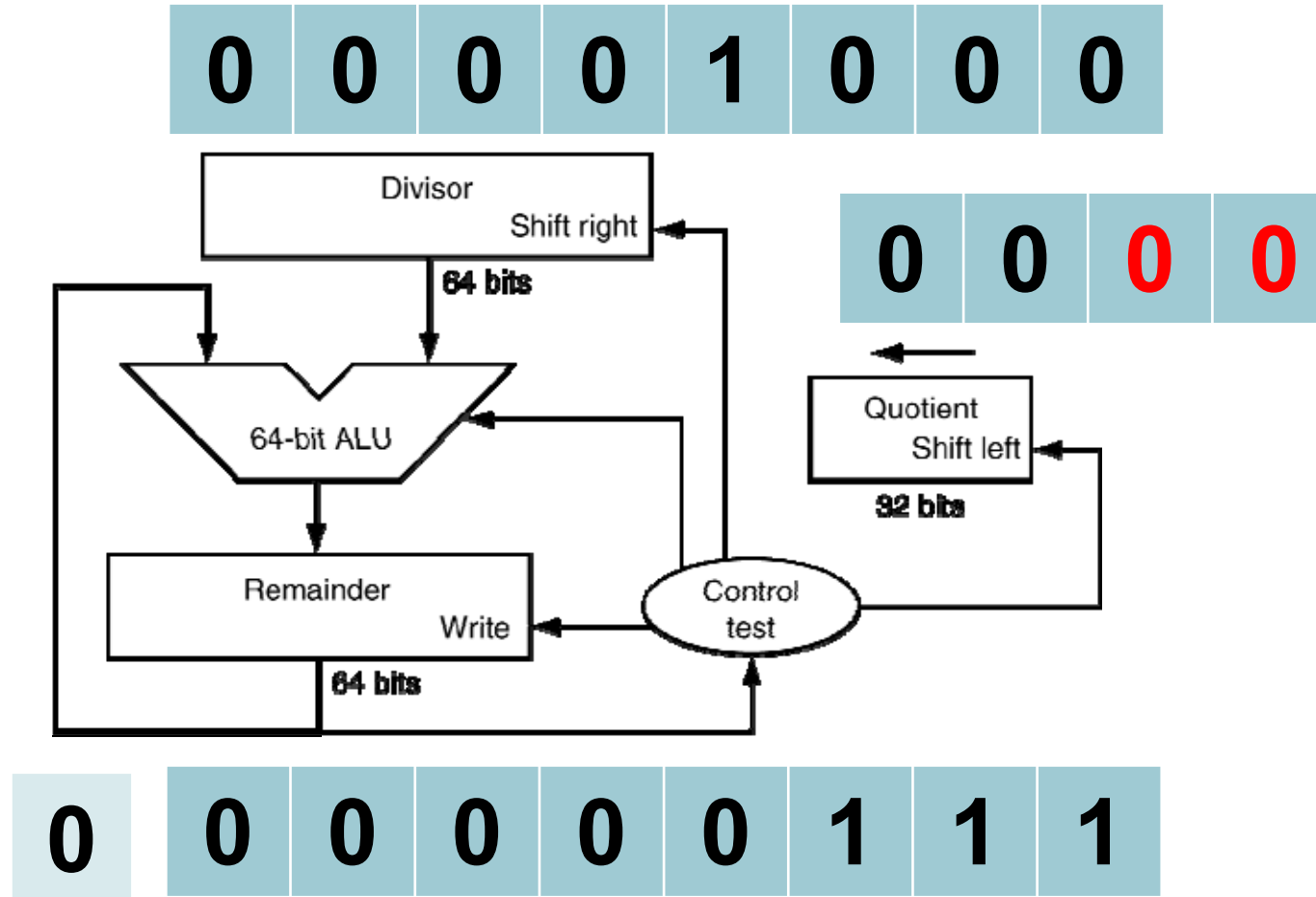
Integer Division Example

Step 2.2 余数<0,商左移1位, 置0



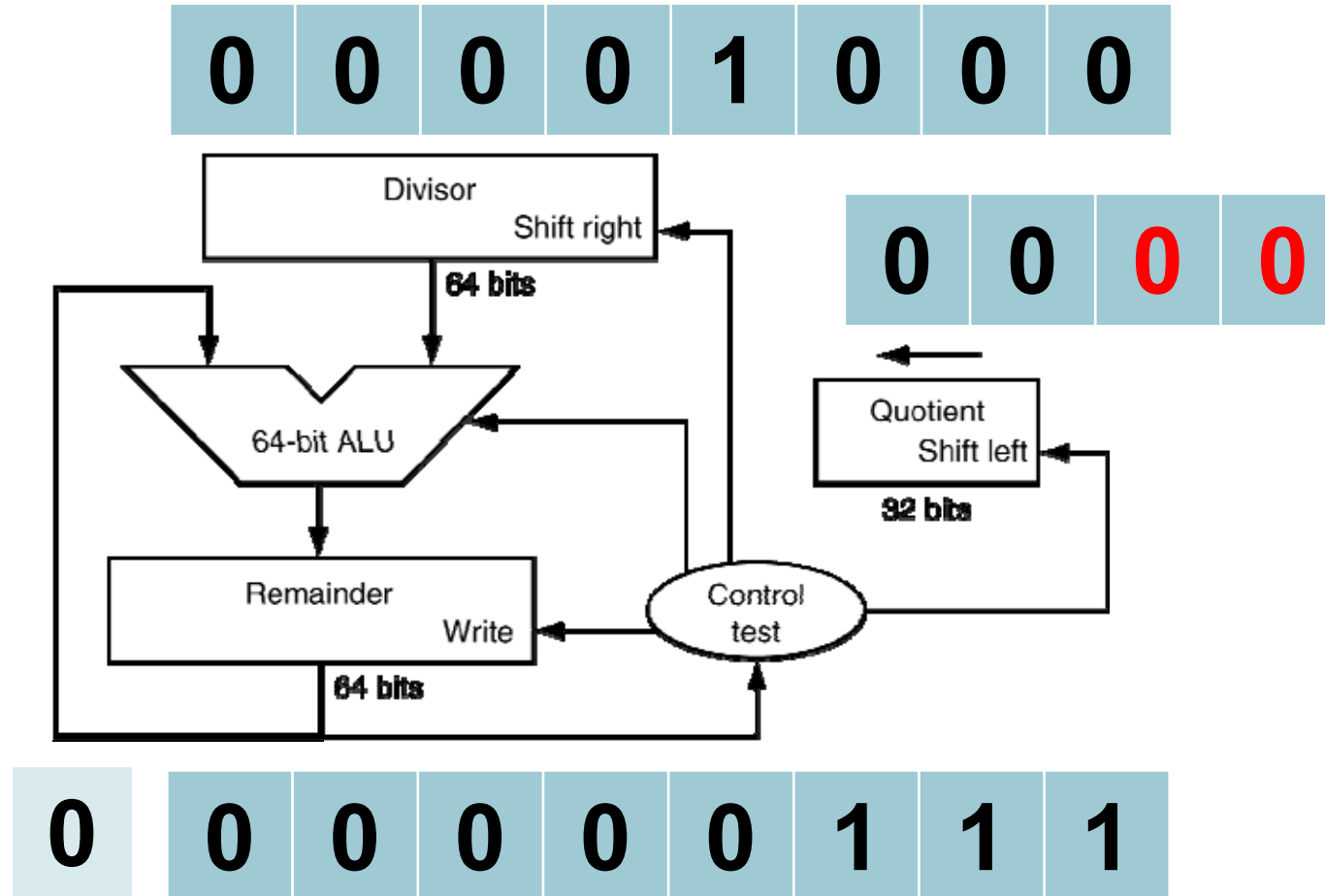
Integer Division Example

Step 2.3 恢复余数，除数右移1位



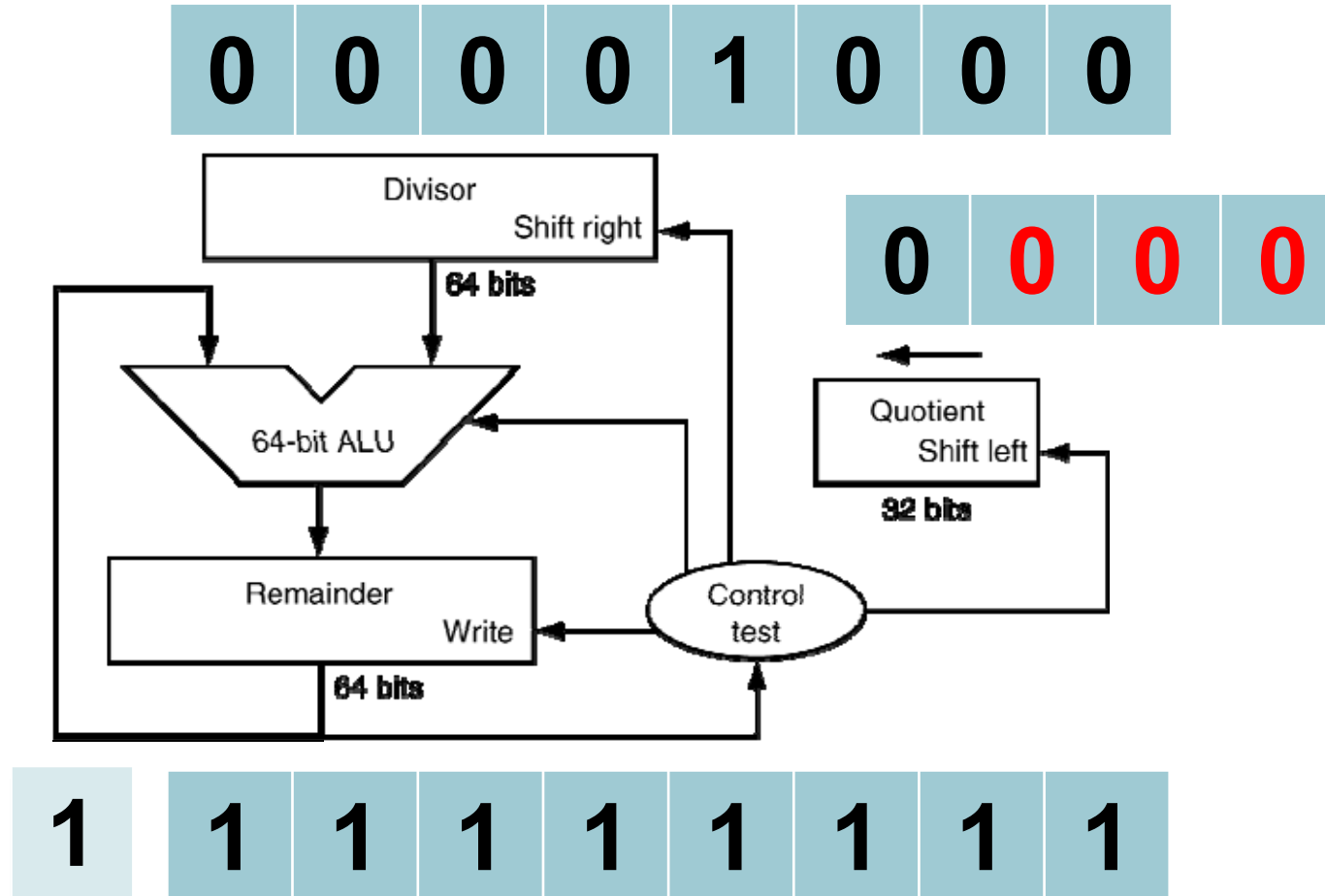
Integer Division Example

Step 3.1 余数-除数 试商



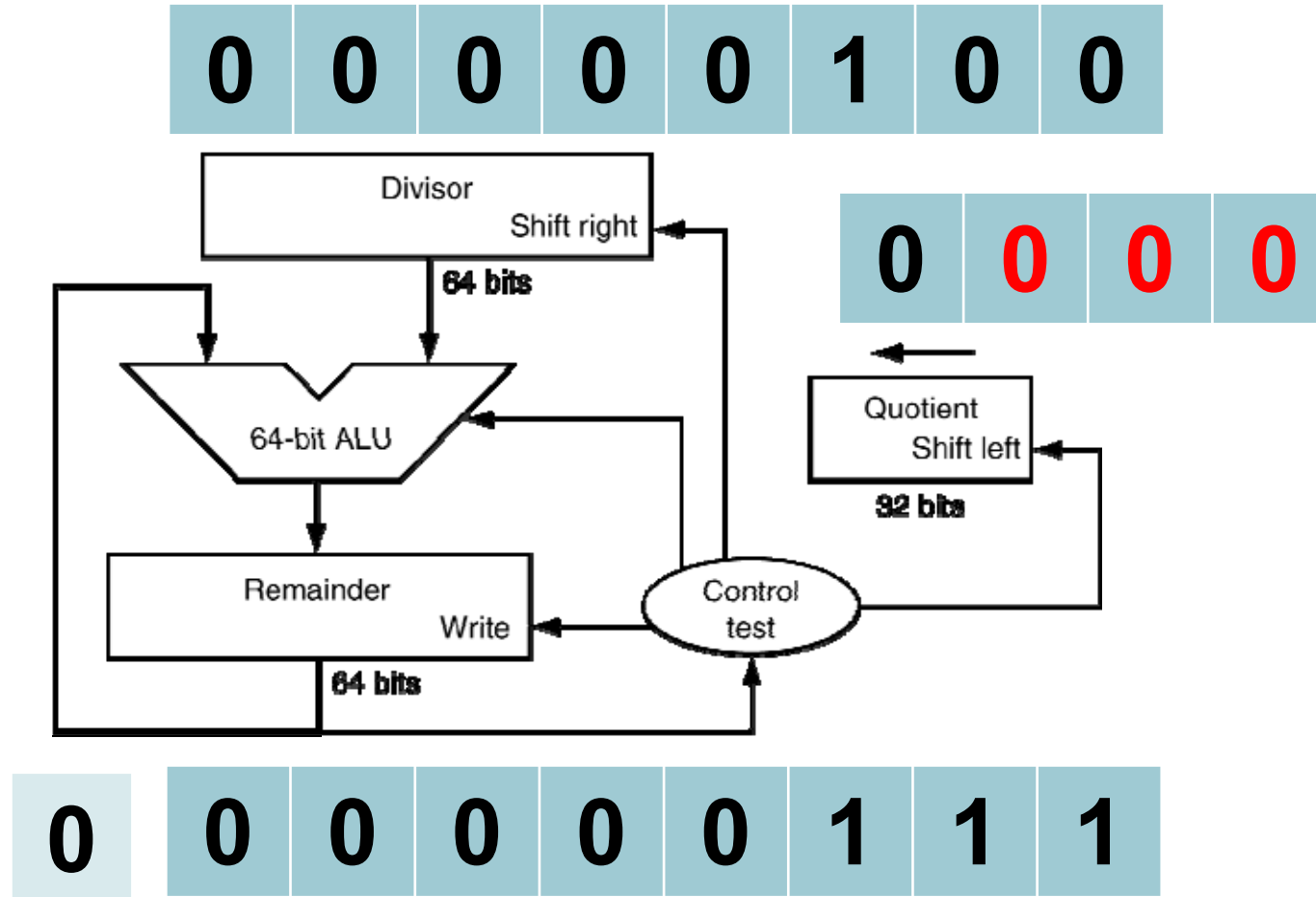
Integer Division Example

Step3.2 余数<0,商左移1位, 置0



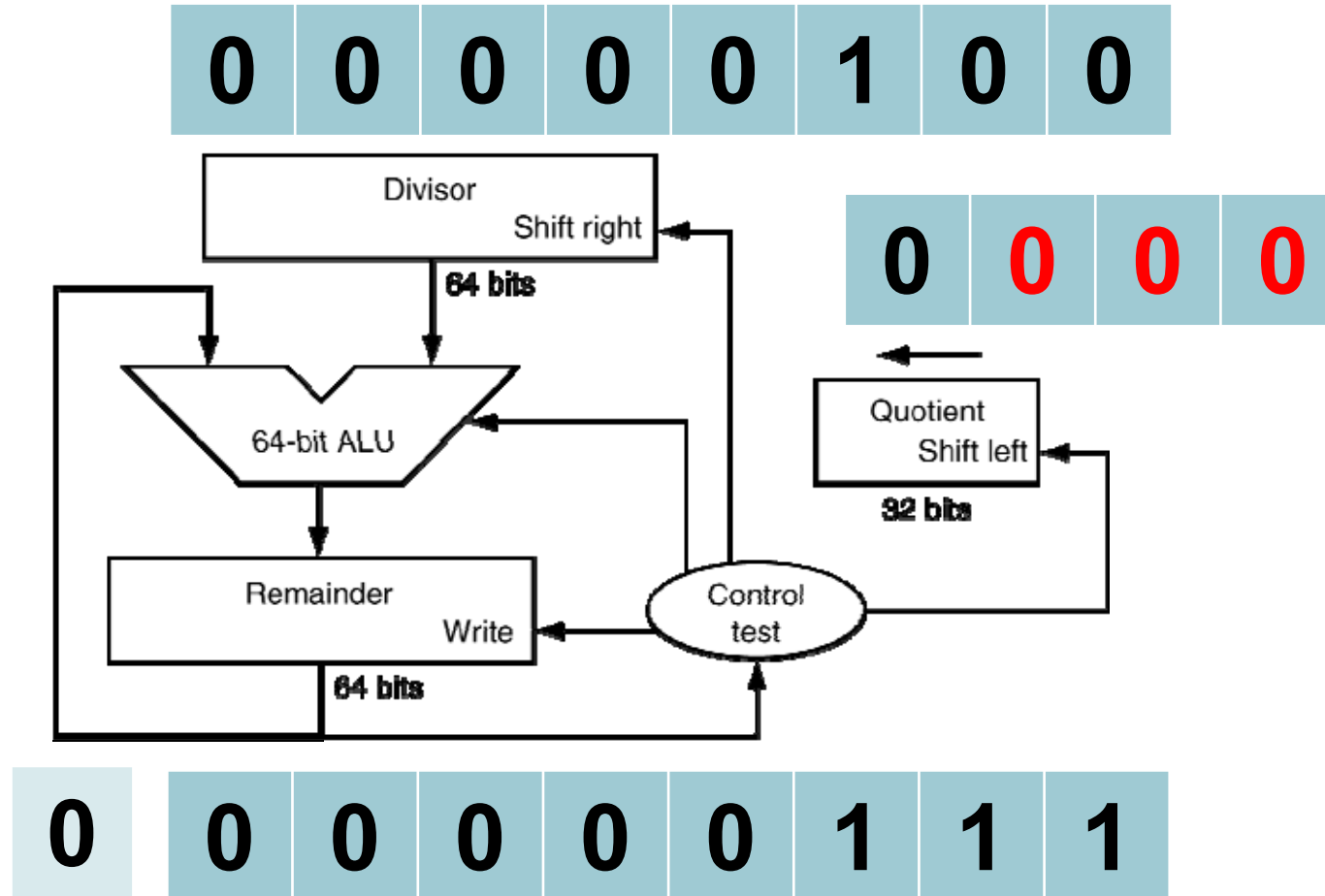
Integer Division Example

Step 3.3 恢复余数，除数右移1位



Integer Division Example

Step 4.1 余数-除数 试商

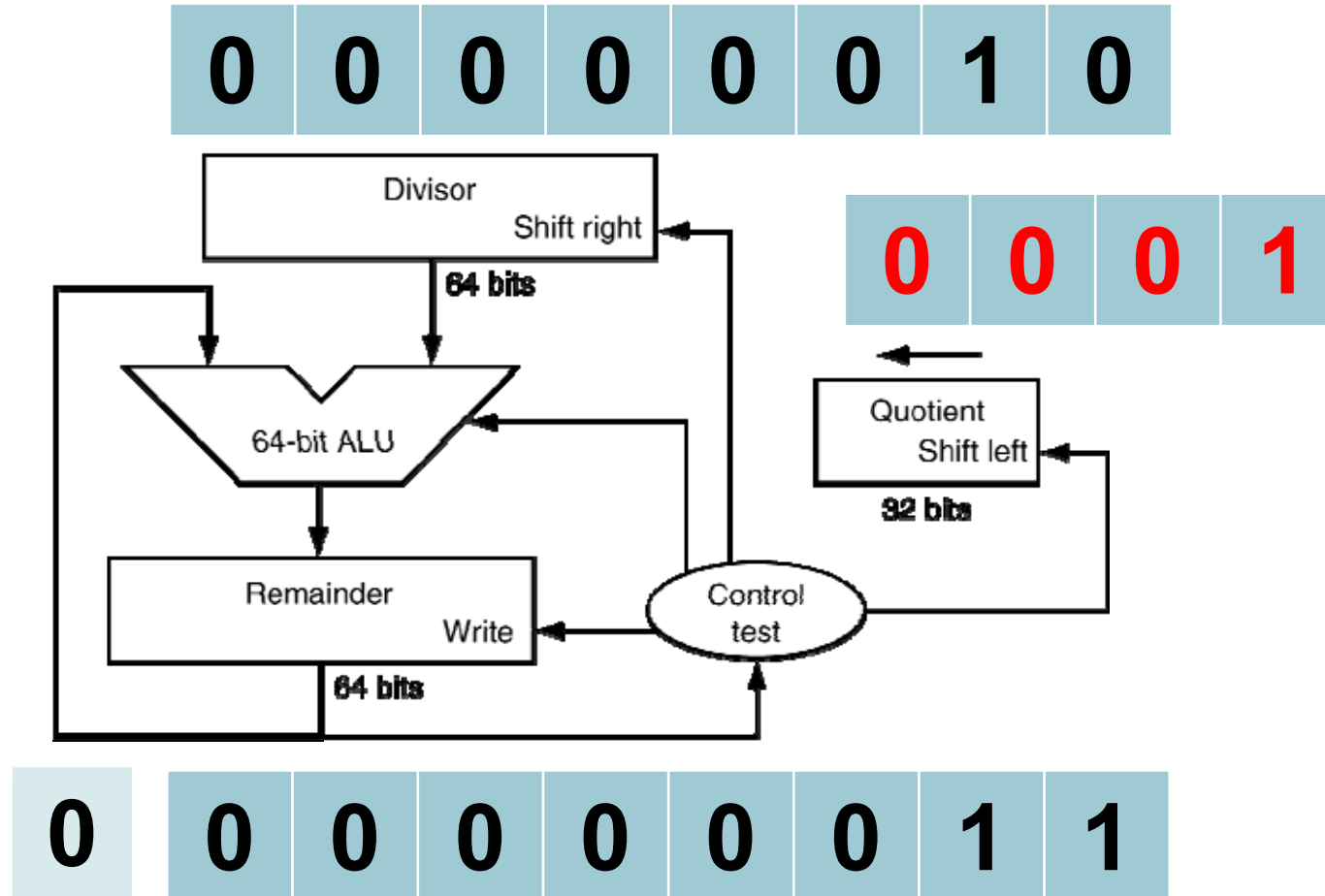


Step4.2 余数>0,商左移1位,置1



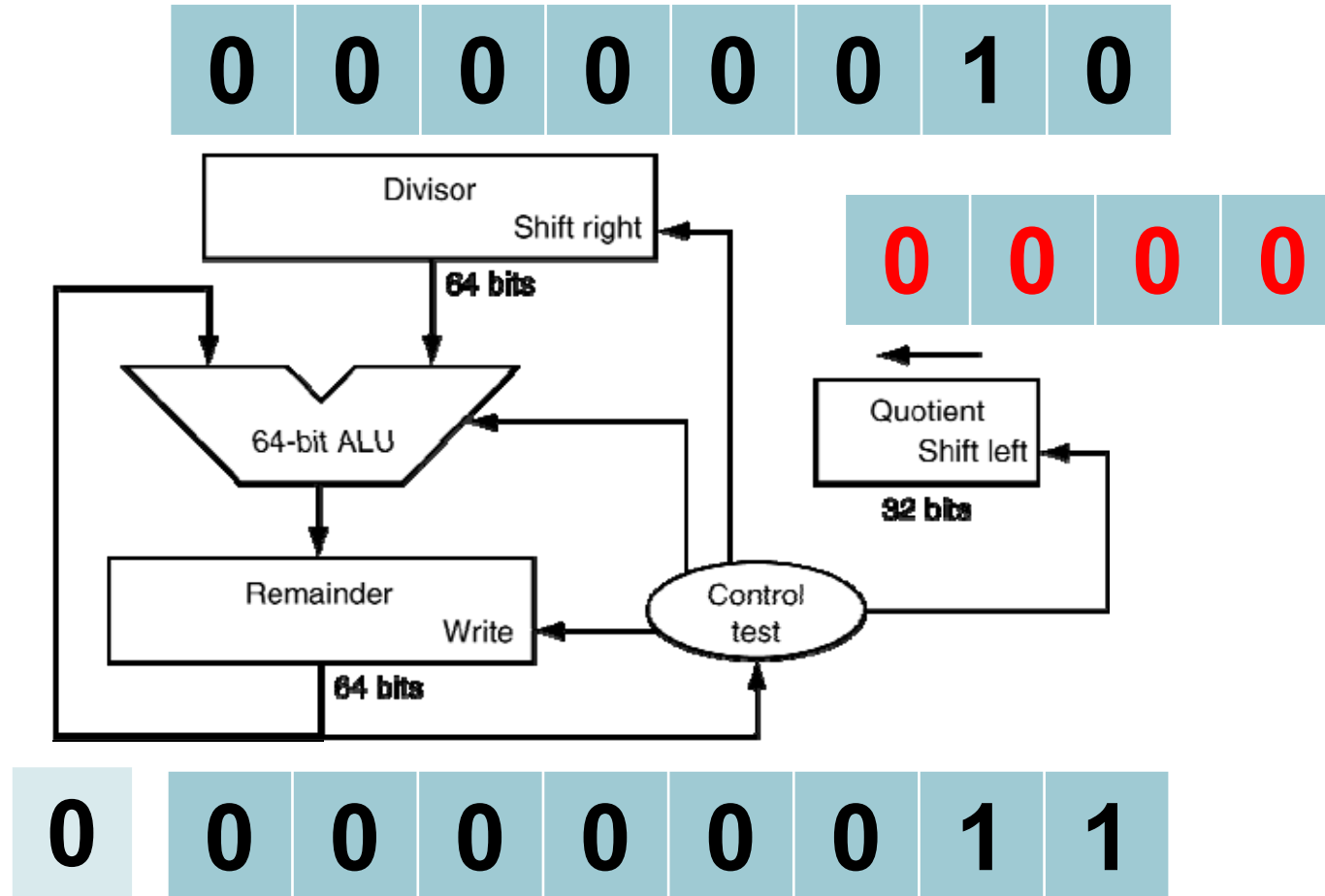
Integer Division Example

Step 4.4 除数右移1位



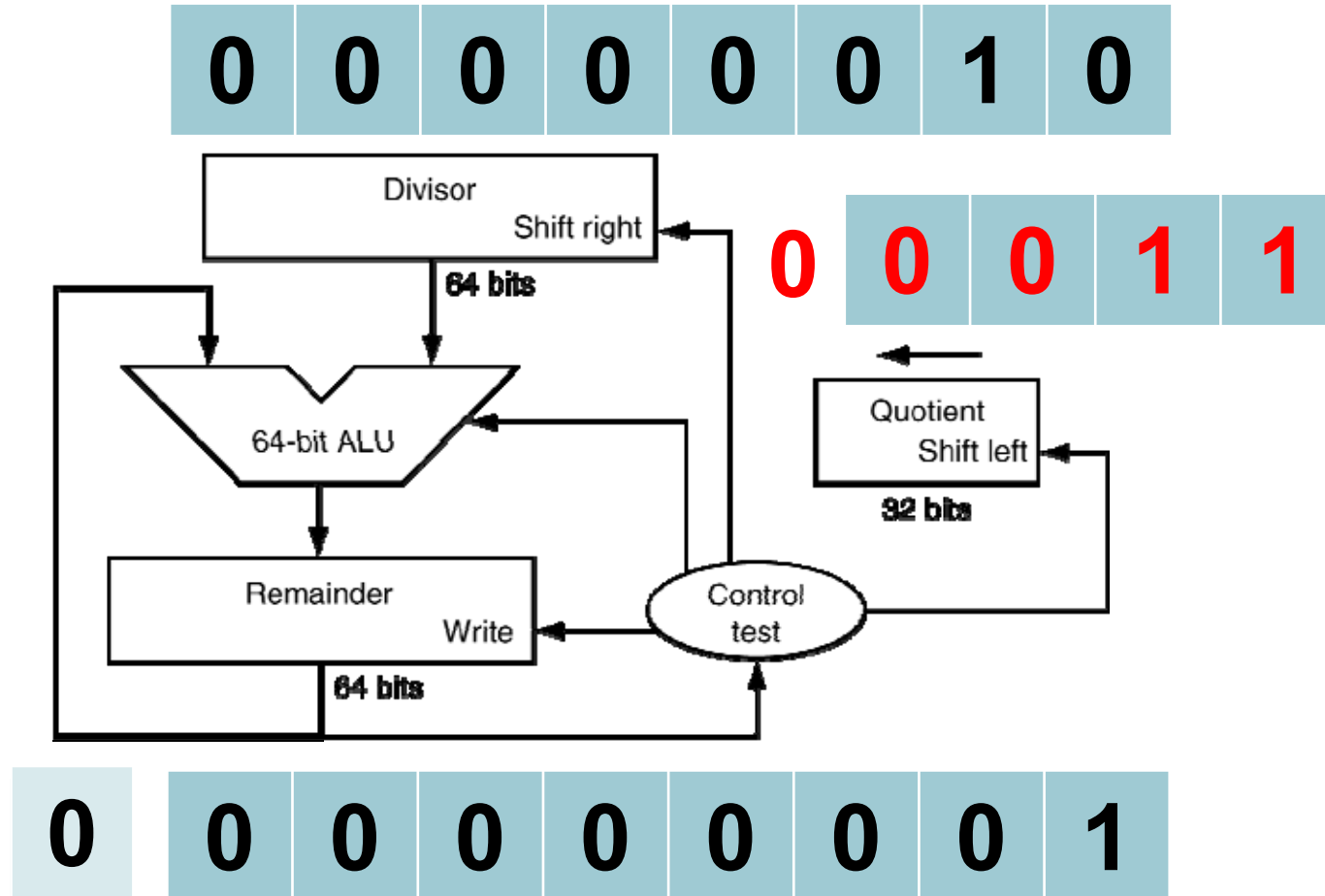
Integer Division Example

Step 5.1 余数-除数 试商



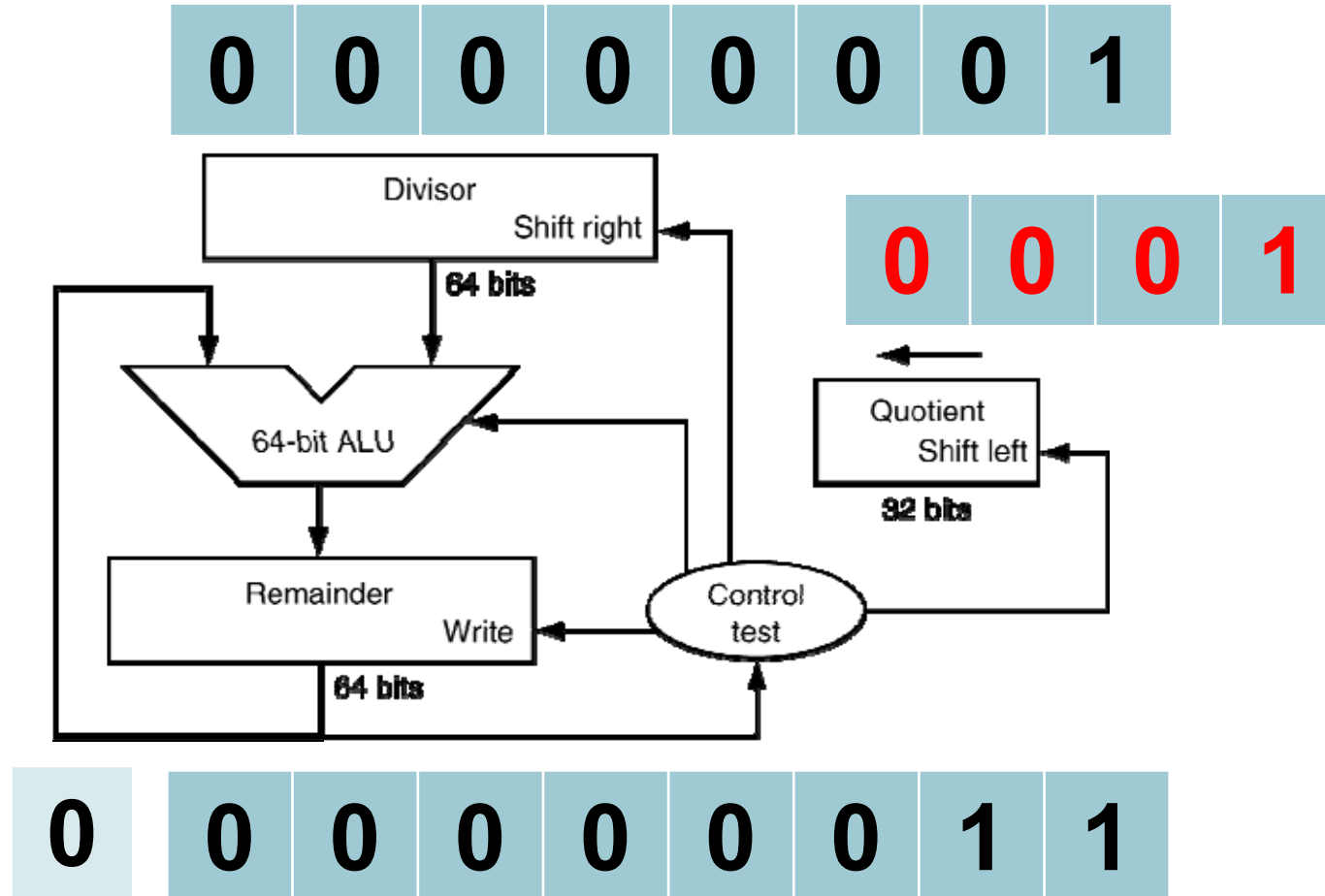
Integer Division Example

Step5.2 余数>0,商左移1位, 置1



Integer Division Example

Step 5.3 除数右移1位(可选)



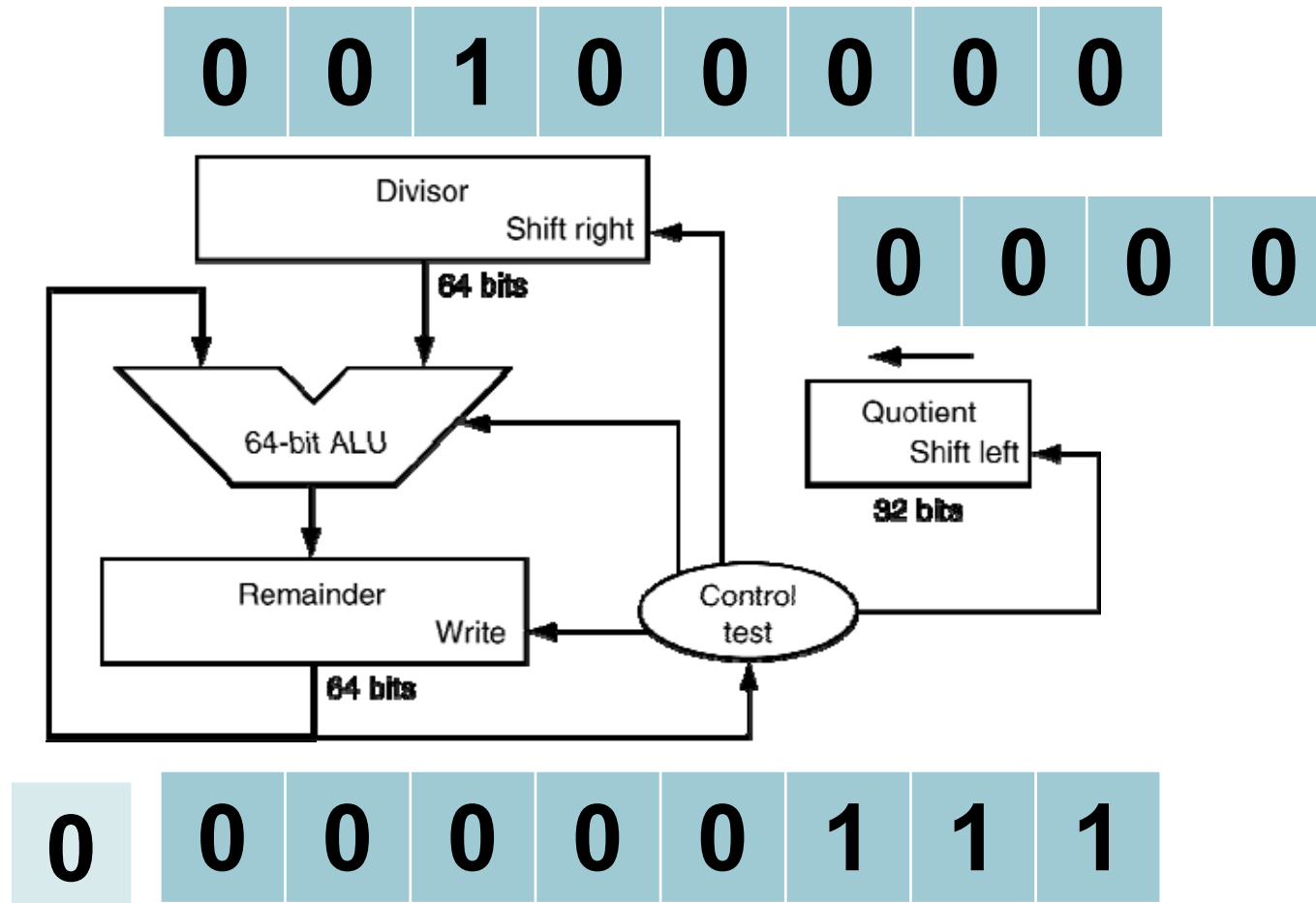
请同学们再观察一遍 $|x|/|y|$ 的过程

$$X = [0111]_2 \quad Y = [0010]_2$$

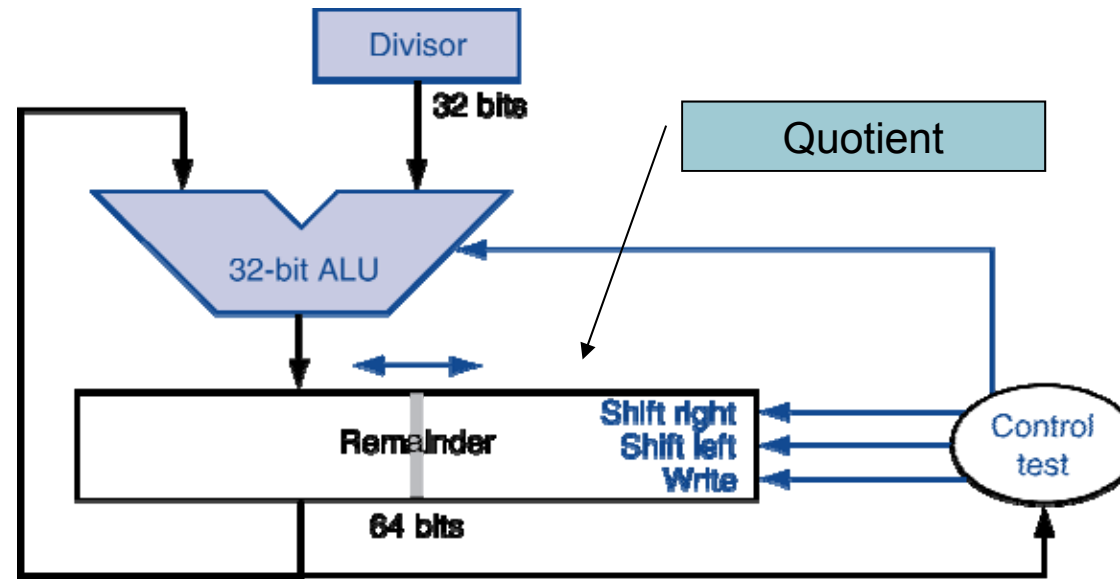
| 迭代次数 | 步骤 | 商 | 除数 | 余数 |
|------|--|----------------------|--|---|
| 0 | 初始化 | 0000 | <u>0010 0000</u> | 0 0000 0111 |
| 1 | 余数=余数-除数 试商, 余数<0 恢复余数 商左移1位, 最低位置0 除数右移 | 0000 0000 0000 | 0010 0000 0010 0000 0001 0000 | 1 1110 0111 0 0000 0111 0 0000 0111 |
| 2 | 余数=余数-除数 试商, 余数<0 恢复余数 商左移1位, 最低位置0 除数右移 | 0000 0000 0000 | 0001 0000 0001 0000 0000 1000 | 1 1111 0111 0 0000 0111 0 0000 0111 |
| 3 | 余数=余数-除数 试商, 余数<0 恢复余数 商左移1位, 最低位置0 除数右移 | 0000 0000 0000 | 0000 1000 0000 1000 0000 0100 | 1 1111 1111 0 0000 0111 0 0000 0111 |
| 4 | 余数=余数-除数 试商1, 余数>=0 商左移1位, 最低置1 除数右移 | 0000 0001 0001 | 0000 0100 0000 0100 0000 0010 | 0 0000 0011 0 0000 0011 0 0000 0011 |
| 5 | 余数=余数-除数 试商1, 余数>=0 商左移1位, 最低置1 除数右移 | 0000 0010 0011 | <u>0000 0010</u> 0000 0010 0000 0001 | 0 0000 0001 0 0000 0001 0 0000 0001 |

Integer Division

$$|x| = [0111]_2 \quad |y| = [0010]_2$$



Optimized Integer Divider

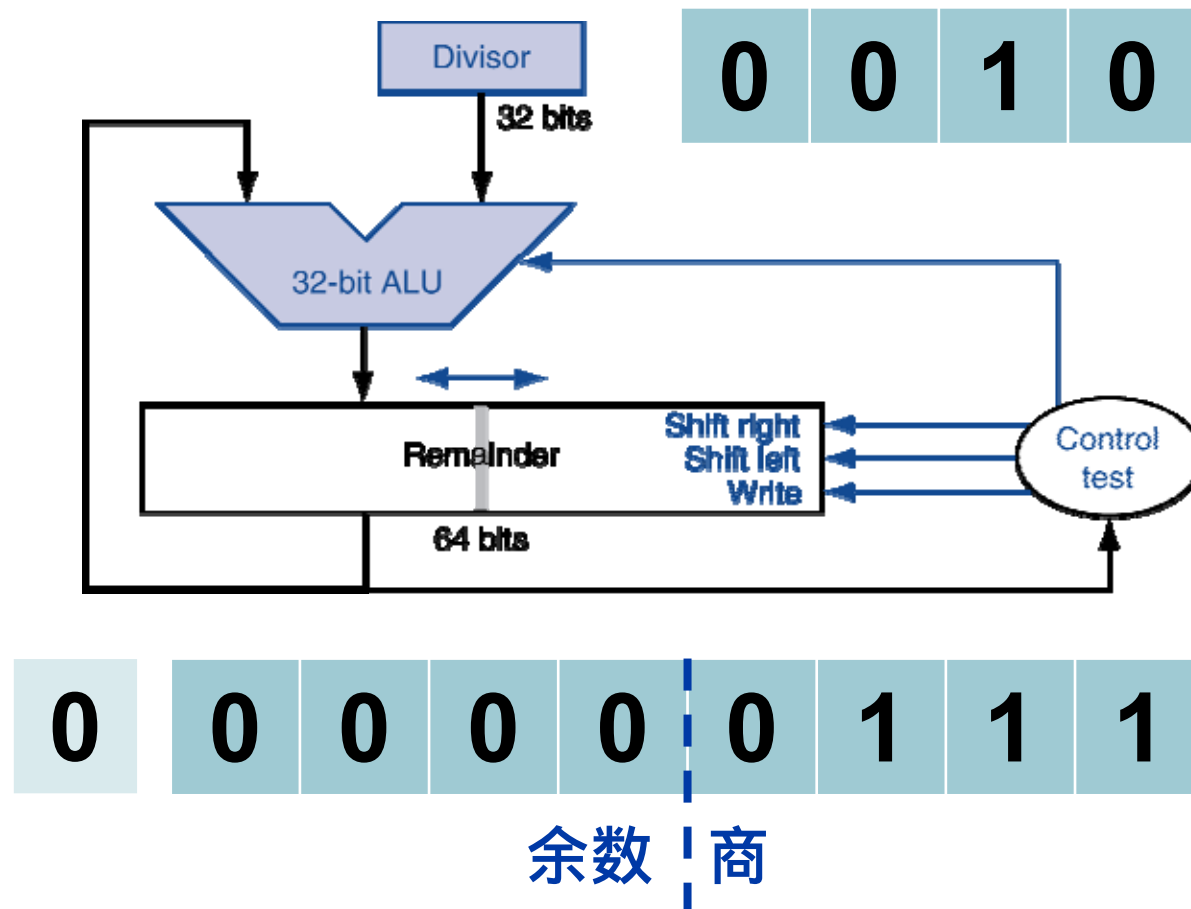


- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
 - Same hardware can be used for both

Optimized Integer Divider

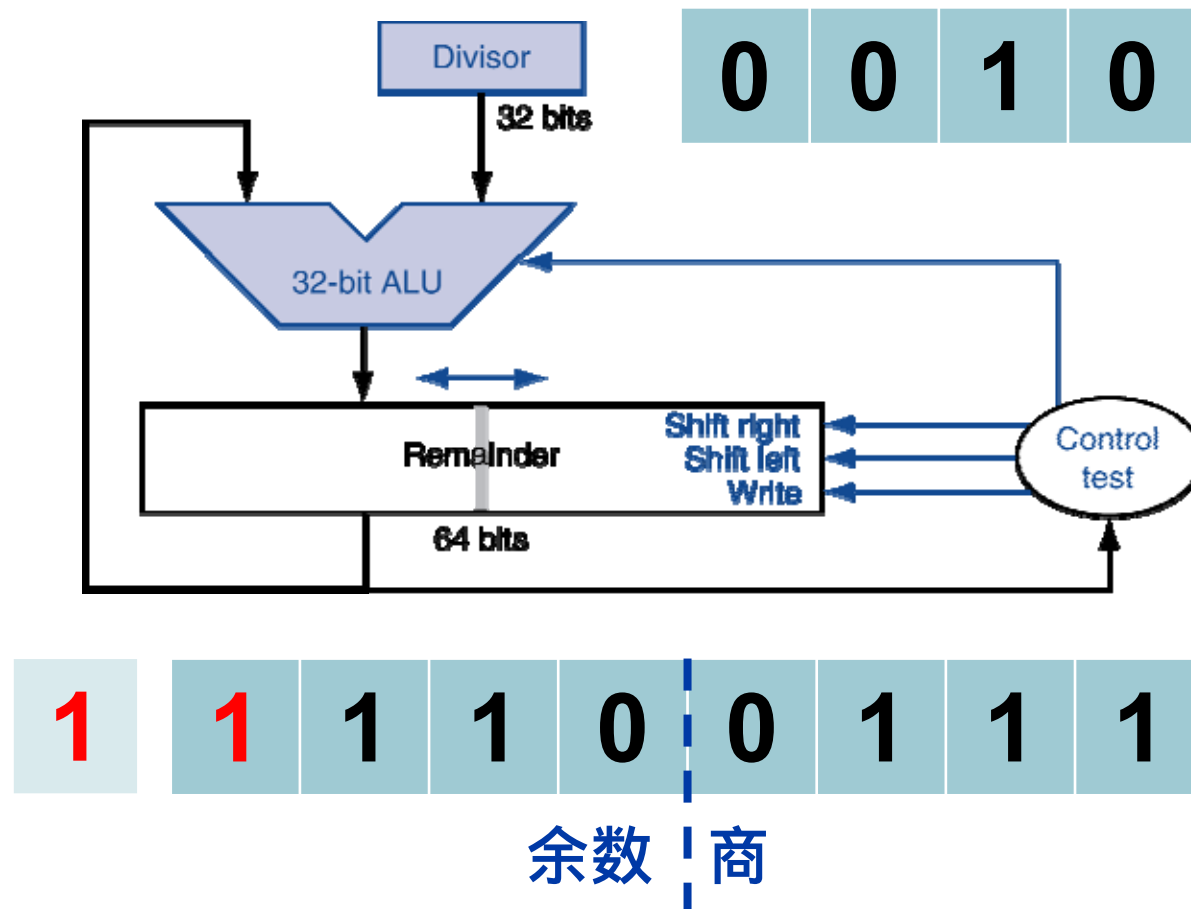
Step 0 初始化

被除数 = $[0111]_2$ 除数 = $[0010]_2$



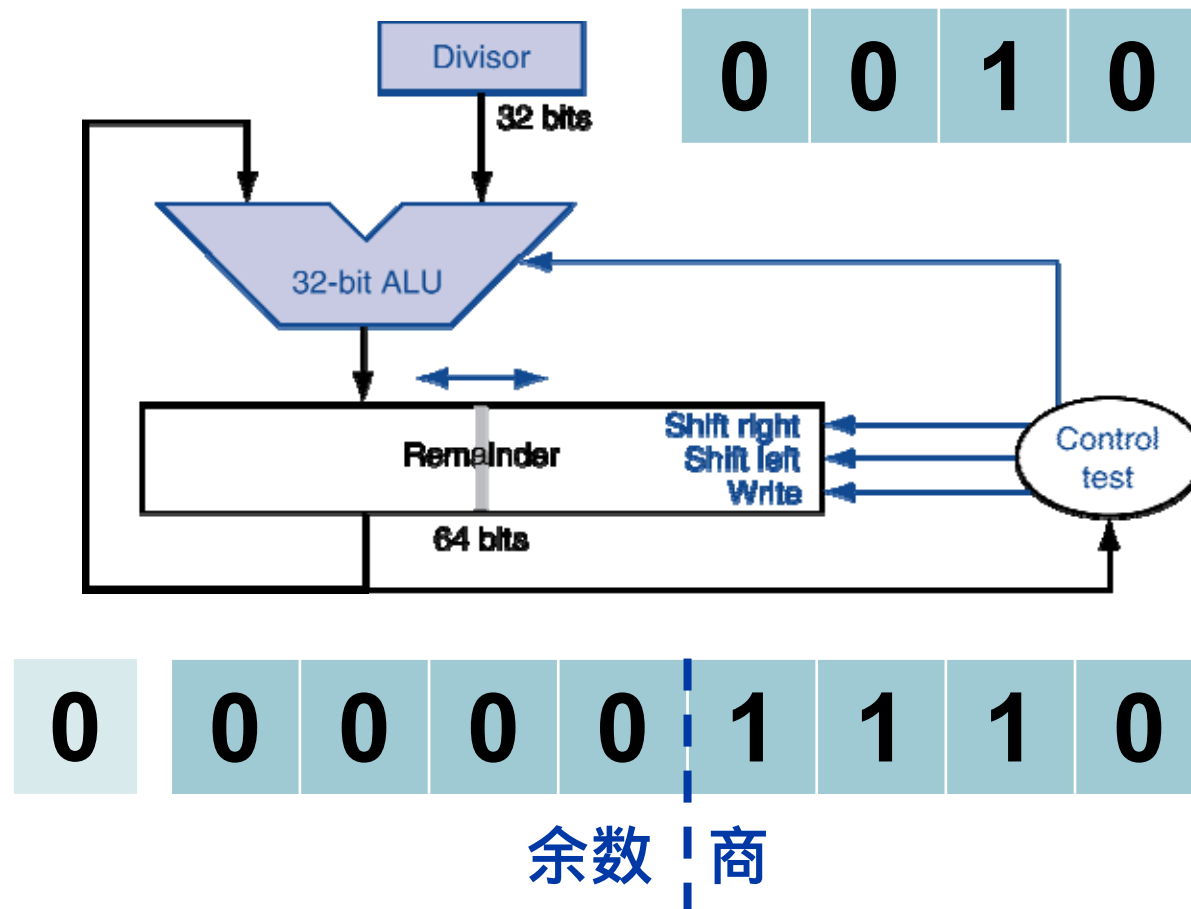
Optimized Integer Divider

Step 1.1 余数=余数-除数 试商



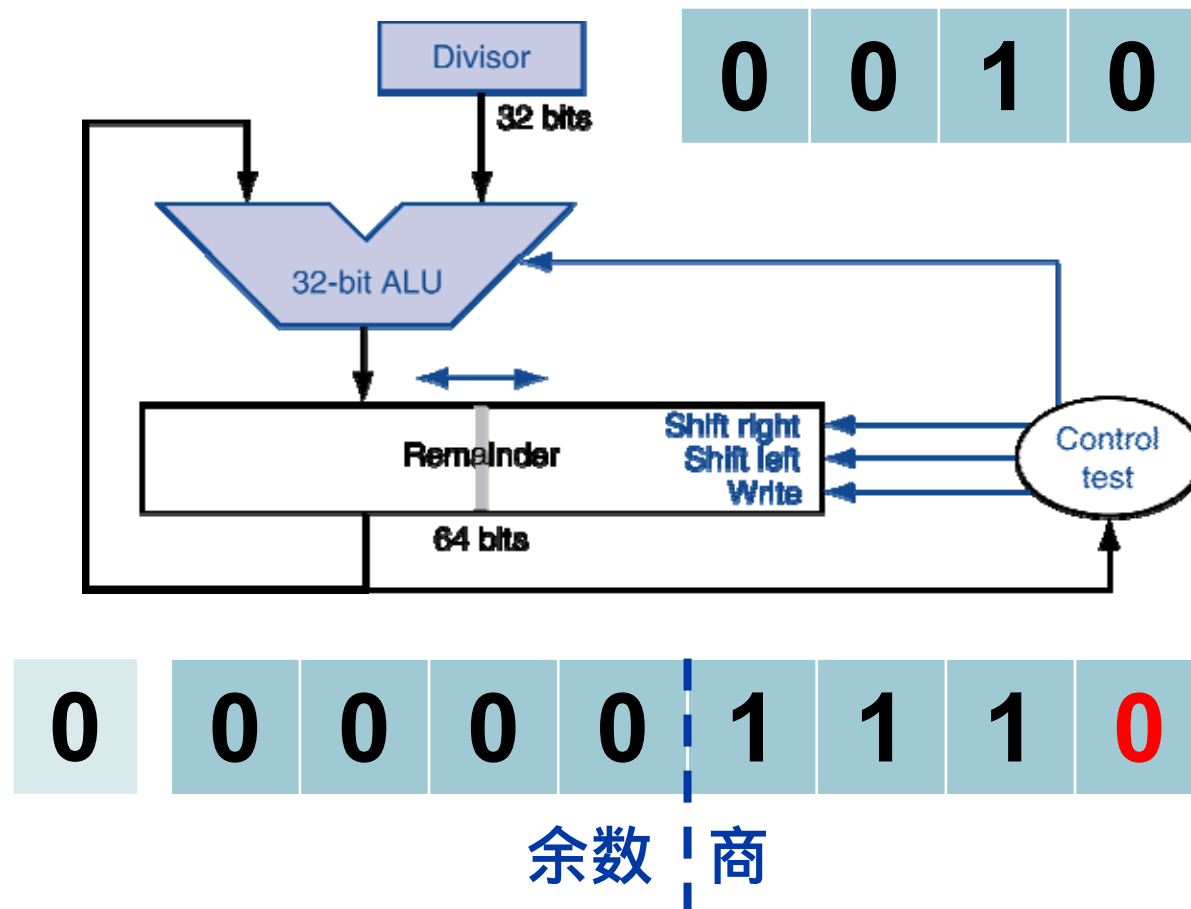
Optimized Integer Divider

Step 1.2 余数<0 恢复余数，余数和商 左移1位



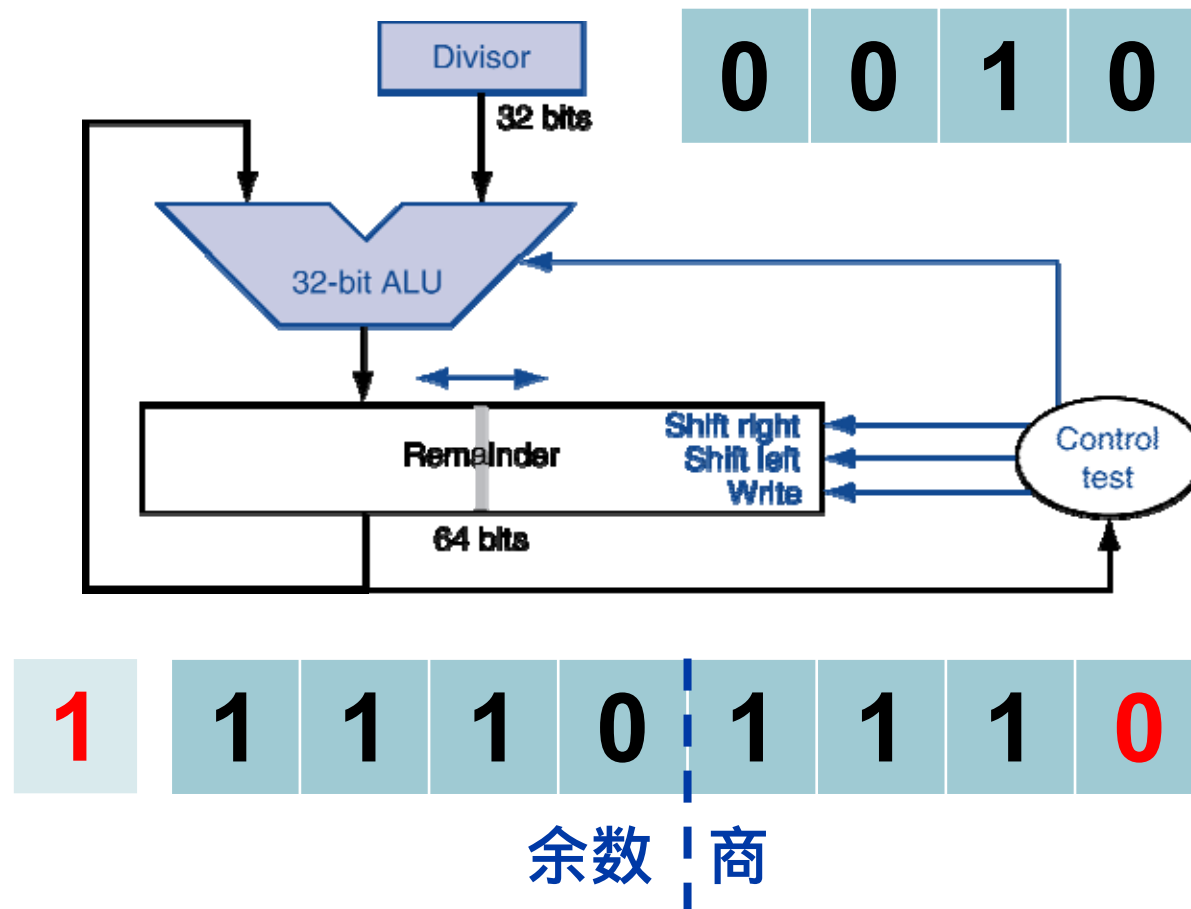
Optimized Integer Divider

Step 1.3 商的 最低位置0



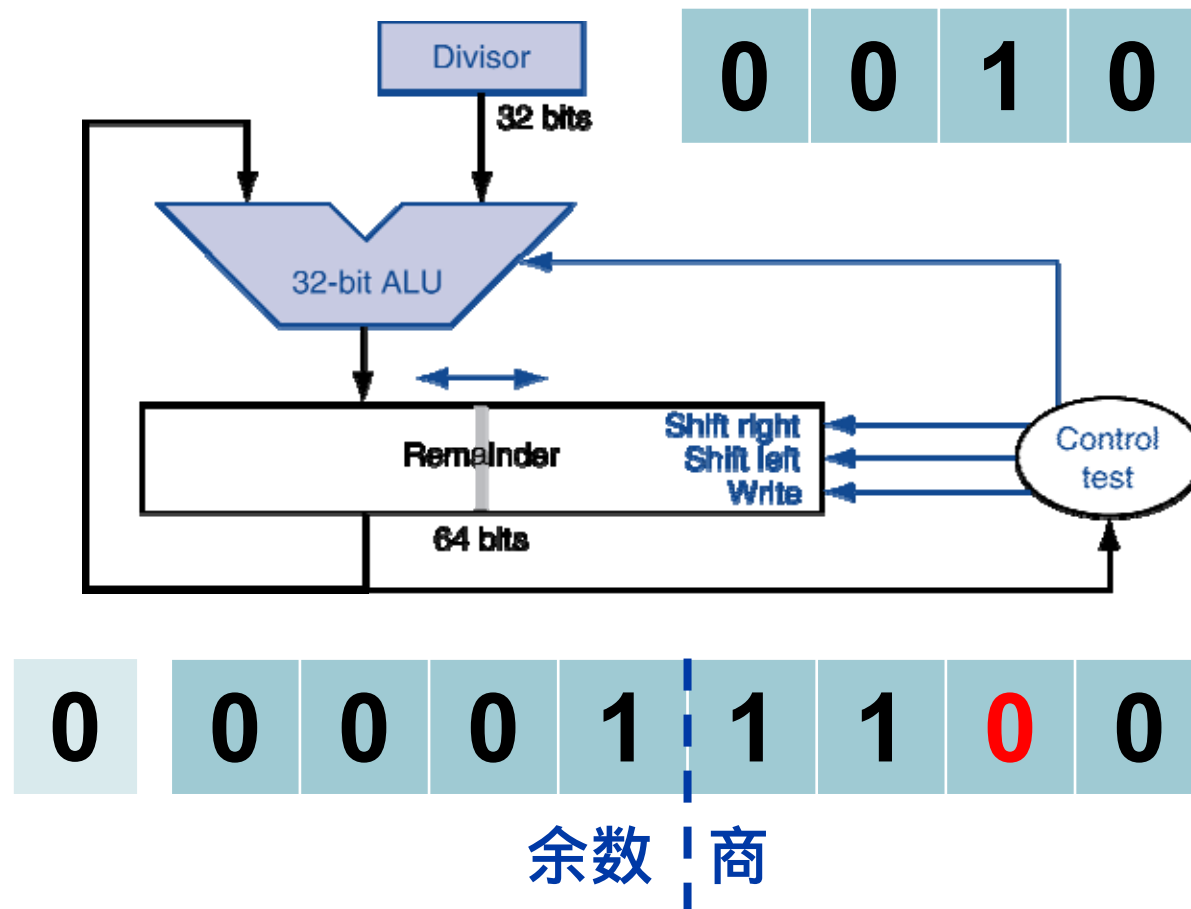
Optimized Integer Divider

Step 2.1 余数=余数-除数 试商



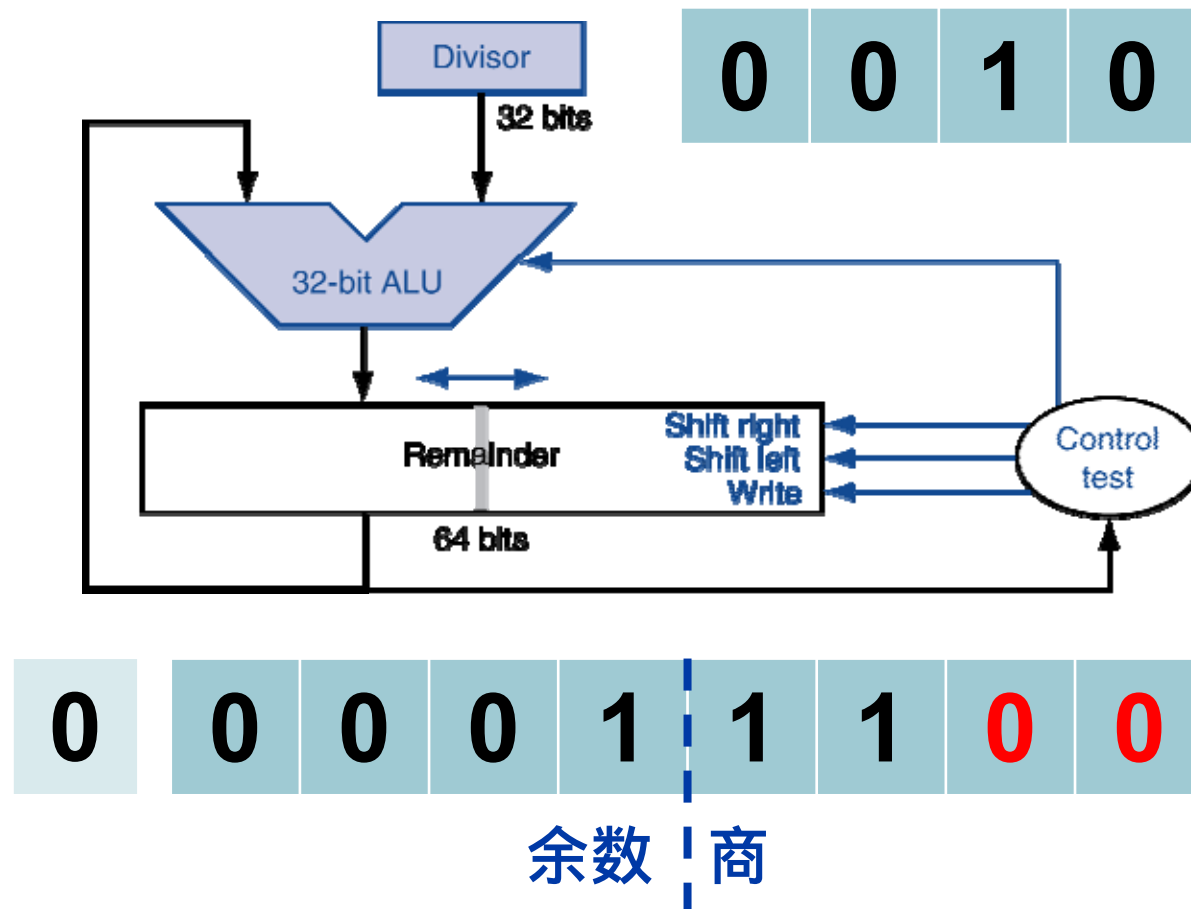
Optimized Integer Divider

Step 2.2 余数<0 恢复余数，余数和商 左移1位



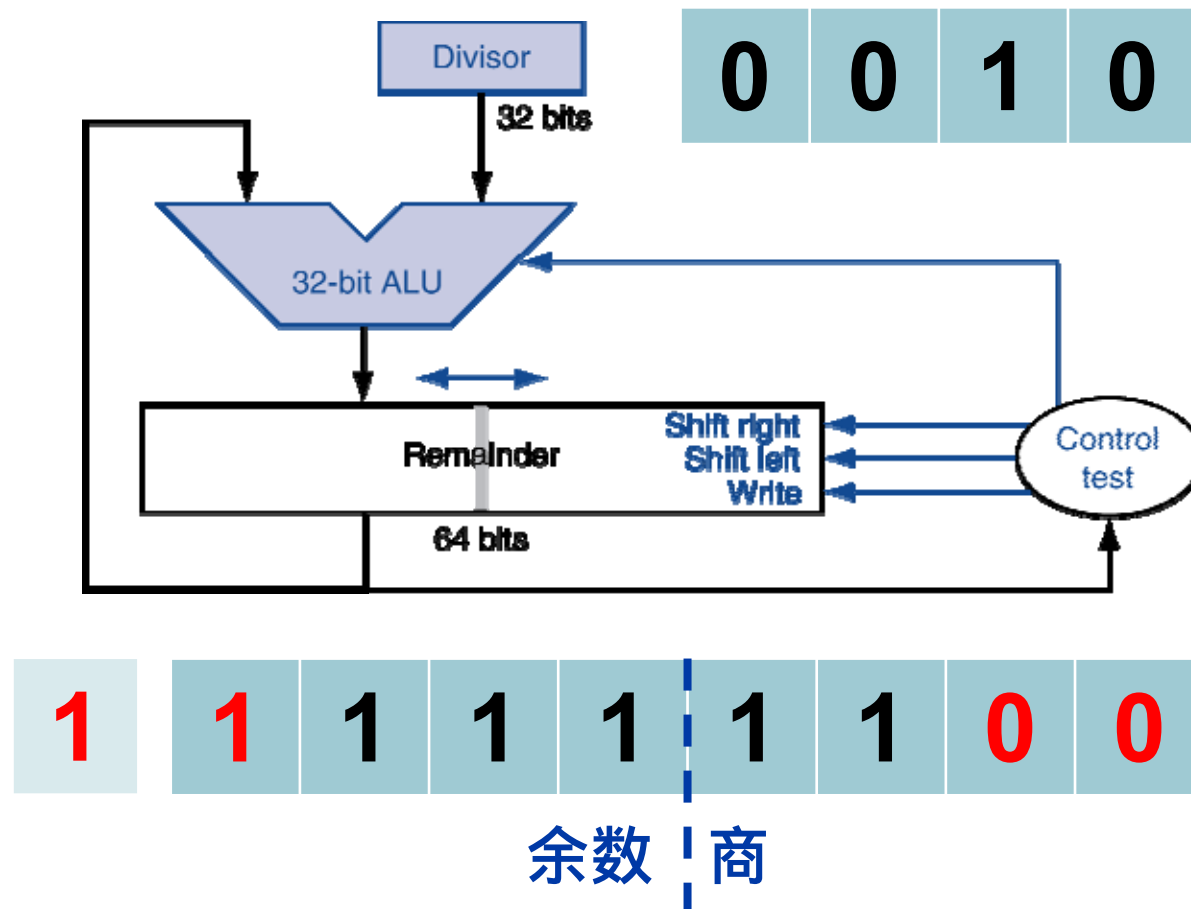
Optimized Integer Divider

Step 2.3 商的 最低位置0



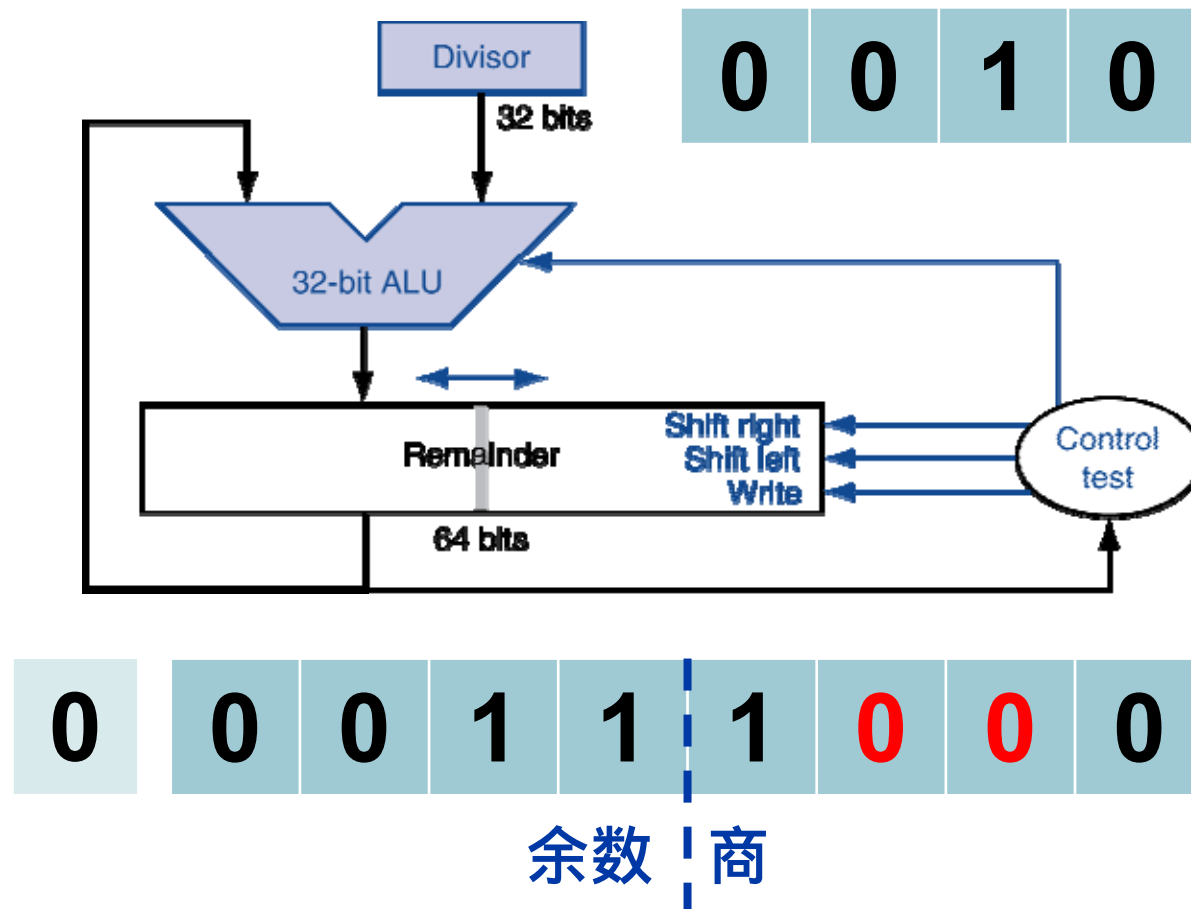
Optimized Integer Divider

Step 3.1 余数=余数-除数 试商



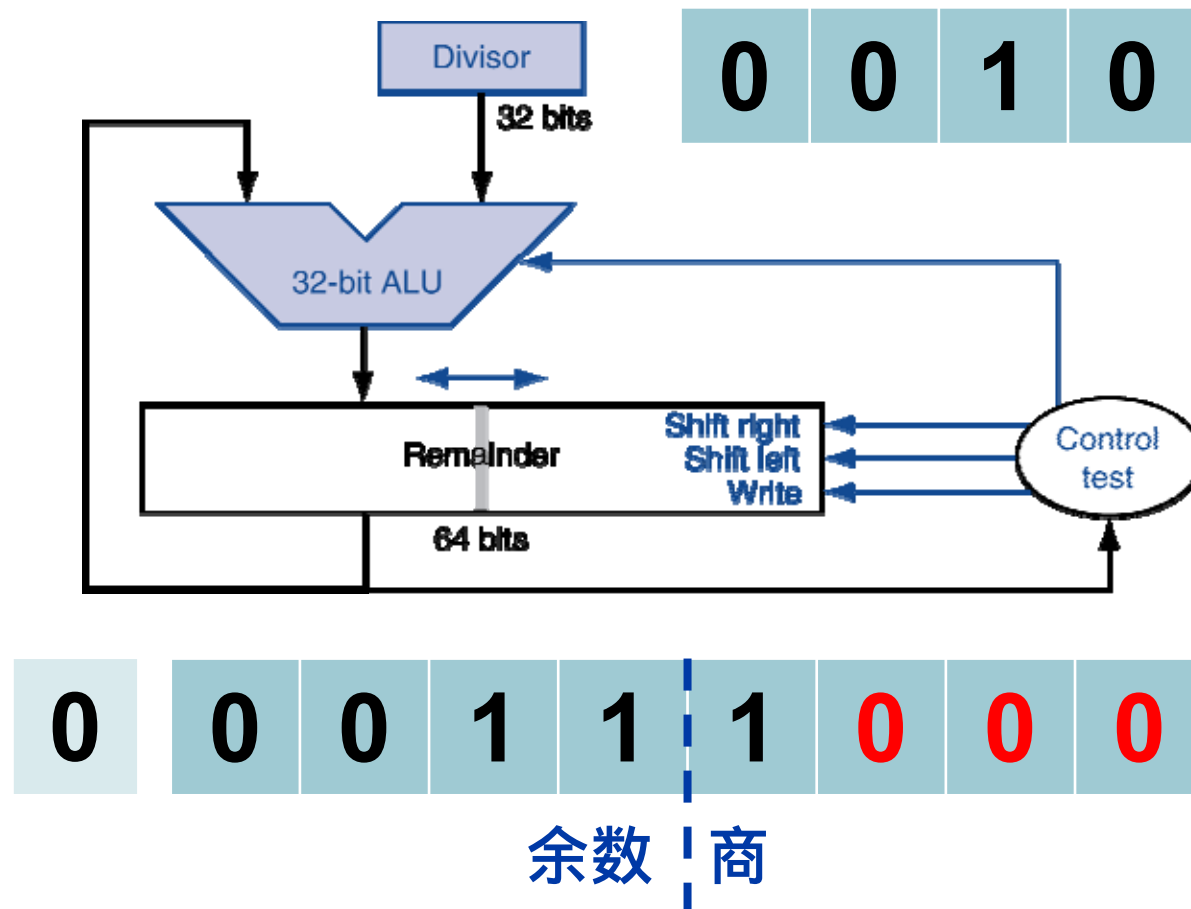
Optimized Integer Divider

Step 3.2 余数 < 0 恢复余数，余数和商 左移1位



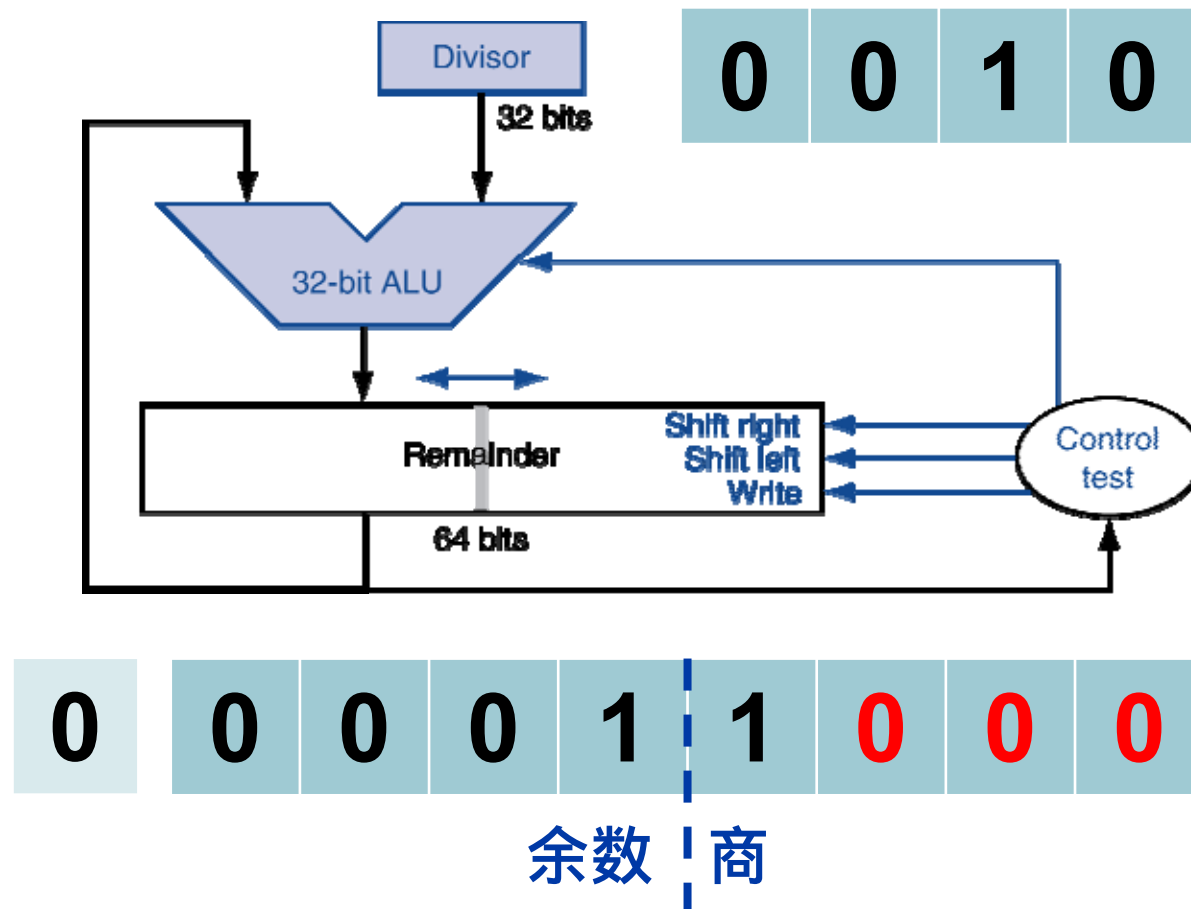
Optimized Integer Divider

Step 3 余数最低位置0



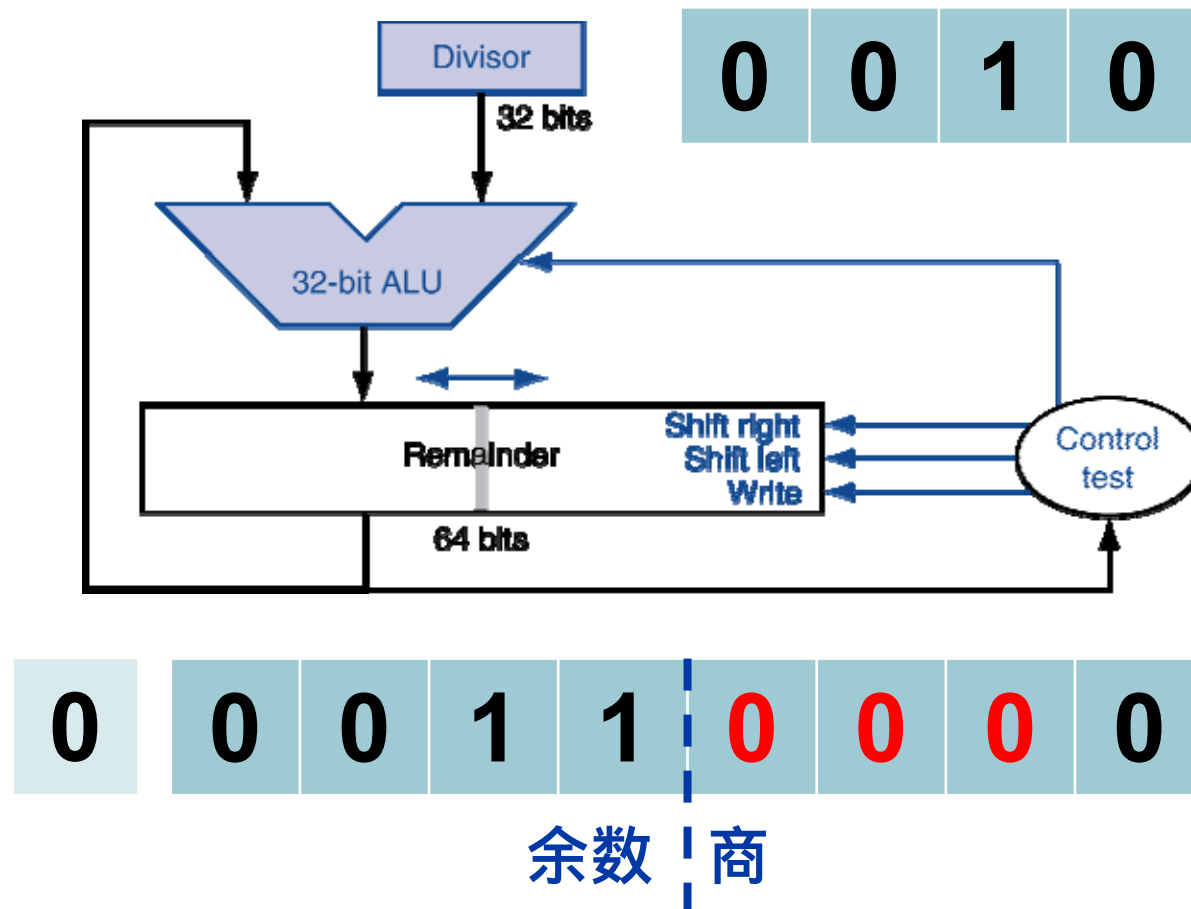
Optimized Integer Divider

Step 4.1 余数=余数-除数 试商



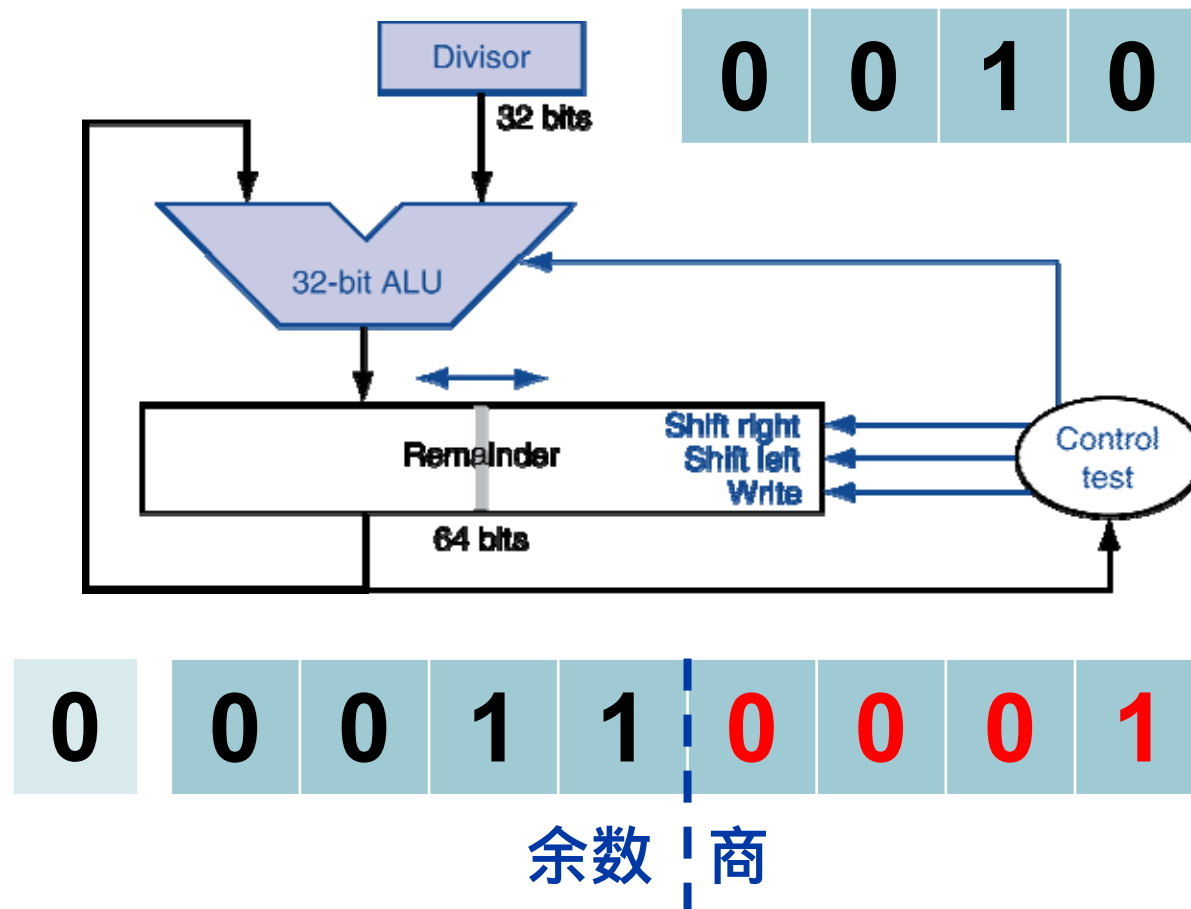
Optimized Integer Divider

Step 4.2 余数>0 , 余数-商左移1位



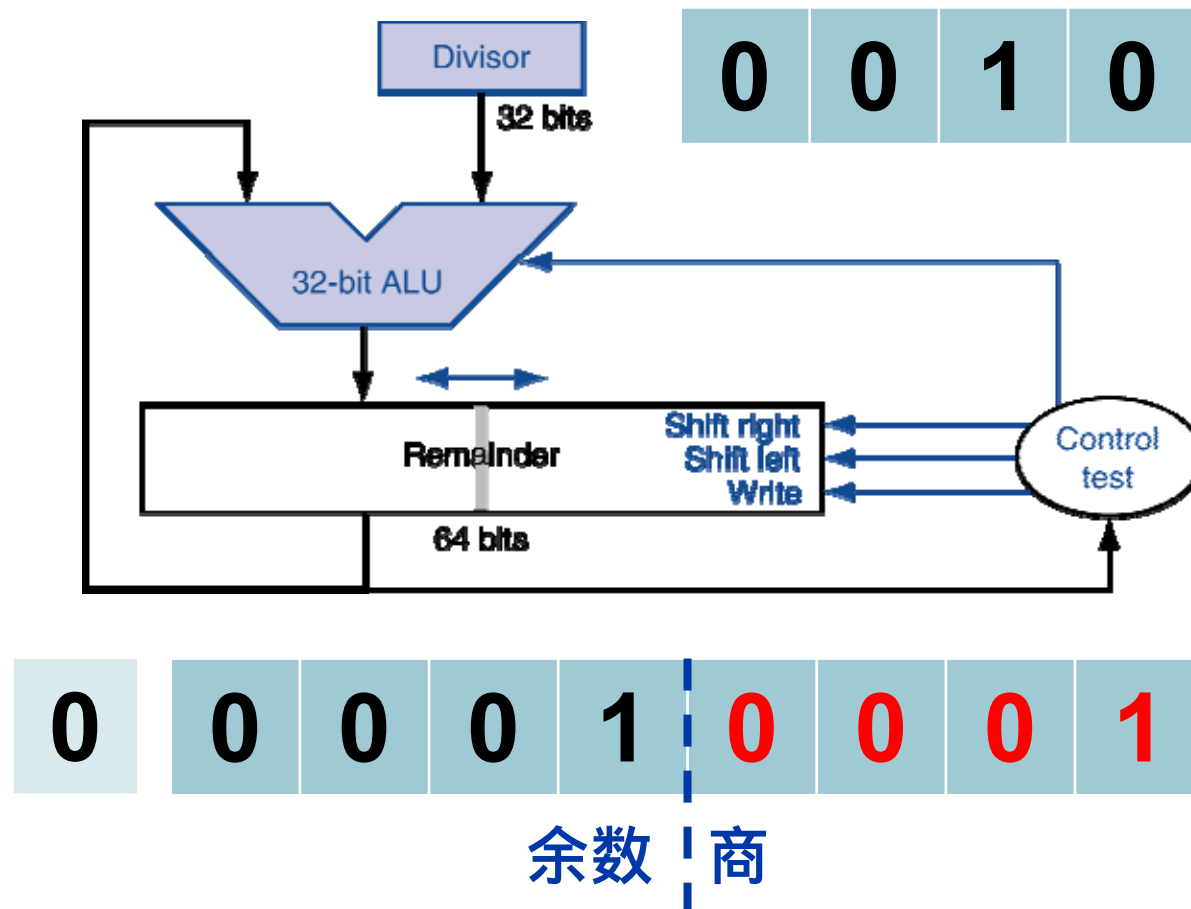
Optimized Integer Divider

Step 4.3 余数最低位置1



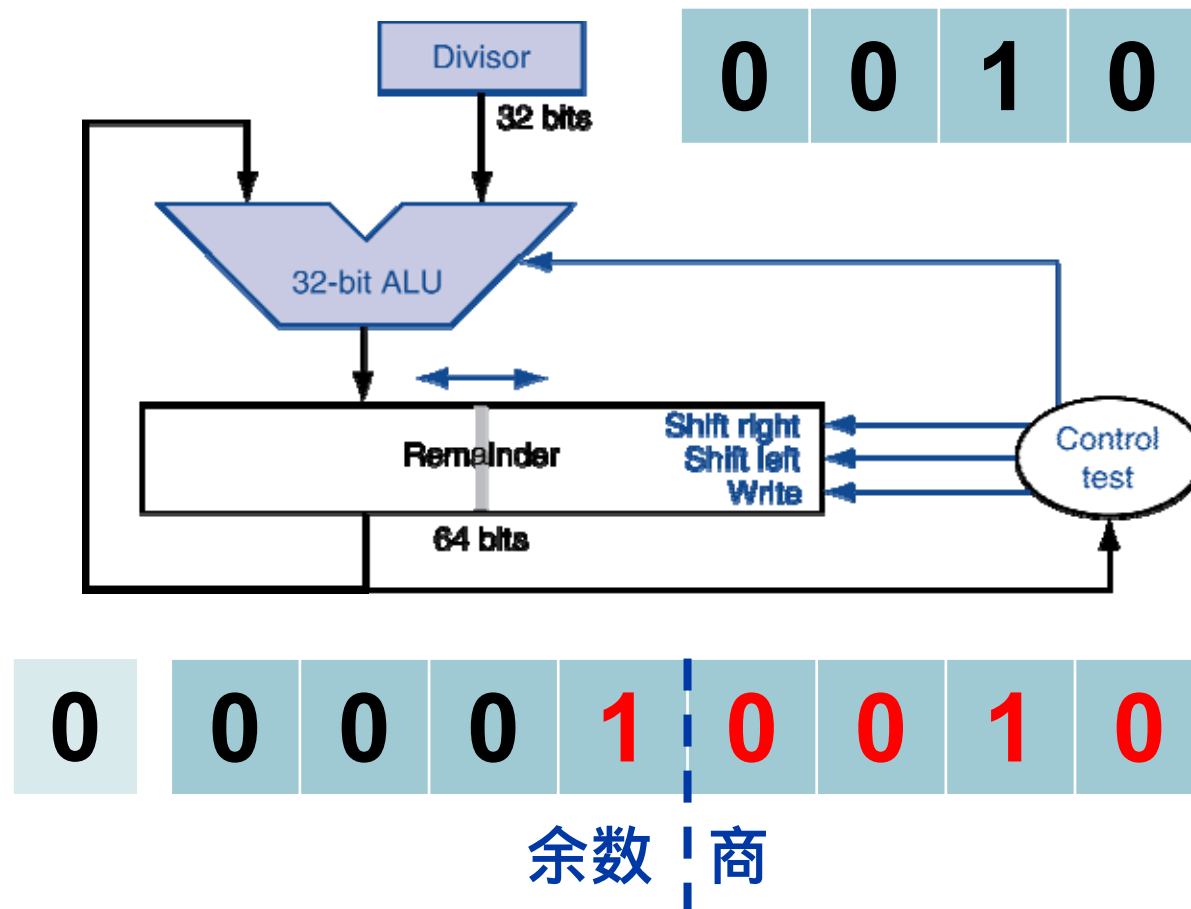
Optimized Integer Divider

Step 5.1 余数=余数-除数 试商



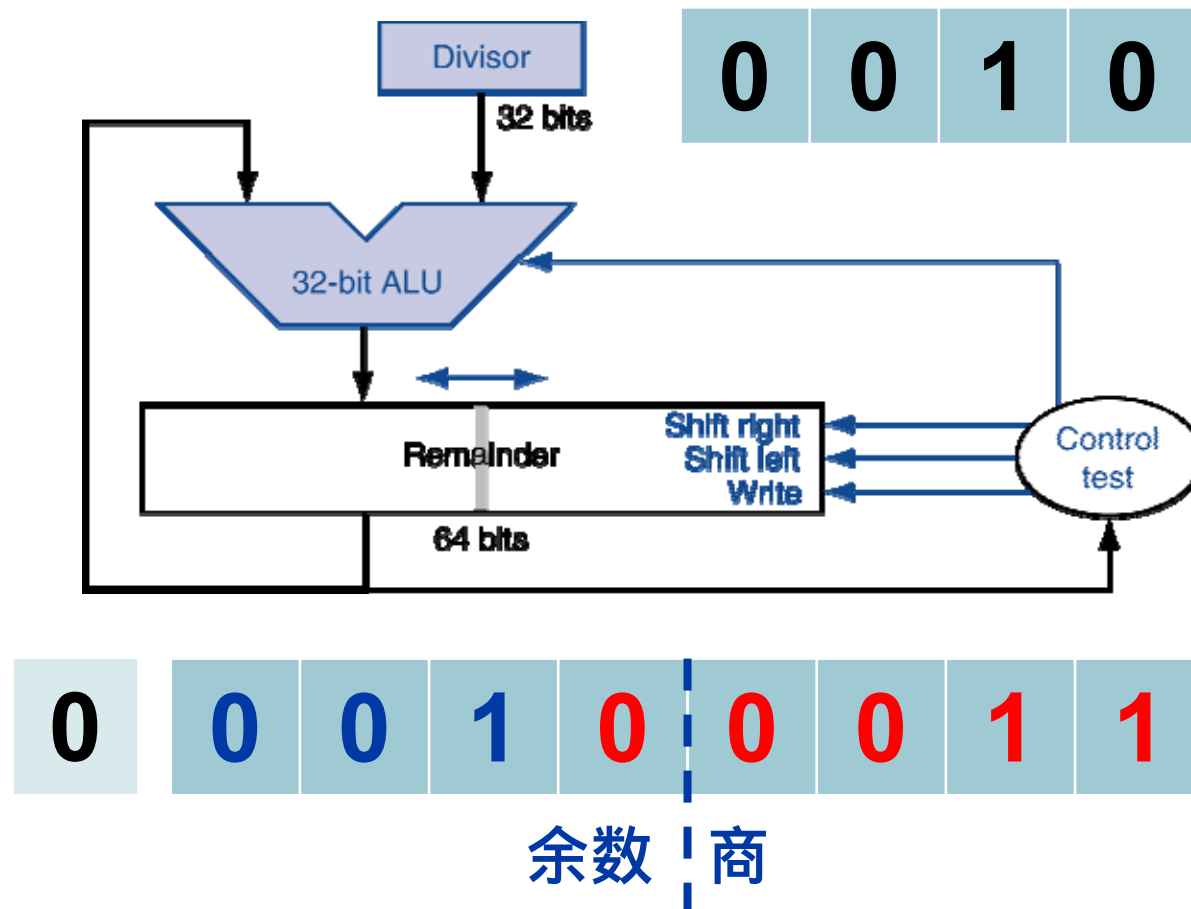
Optimized Integer Divider

Step 5.2 余数>0 , 余数-商左移1位



Optimized Integer Divider

Step 5.3 余数最低位置1



令 $X = [0111]_2$ $Y = [0010]_2$ 优化的除法汇总

| 迭代次数 | 步骤 | 除数 | 余数 商 |
|------|--|------|---|
| 0 | 初始化 | 0010 | 0 0000 0111 |
| 1 | 余数=余数-除数 试商 余数<0 恢复余数并左移1位 余数最低位置0 | 0010 | 1 1110 0111 0 0000 1110 1 0000 1110 |
| 2 | 余数=余数-除数 试商 余数<0 恢复余数并左移1位 余数最低位置0 | 0010 | 1 1110 1110 0 0001 1100 0 0001 1100 |
| 3 | 余数=余数-除数 试商 余数<0 恢复余数并左移1位 余数最低位置0 | 0010 | 1 1111 1110 0 0011 1000 0 0011 1000 |
| 4 | 余数=余数-除数 试商 余数>0, 余数-商左移1位 余数最低位置1 | 0010 | 0 0001 1000 0 0011 0000 0 0011 0001 |
| 5 | 余数=余数-除数 试商 余数>0, 余数-商左移1位 余数最低位置1 | 0010 | 0 0001 0001 0 0001 0010 0 0010 0011 |

商为 $= [0011]_2$ 余数为 $= [0000\ 0001]_2$

原码1位除法-加减交替除法

实际上常用的是**加减交替法**，其运算规则：

- 商的符号单独处理 $q_f = x_f \quad y_f$
 - 试商运算并左移
 - 当余数为正时，商1，余数左移一位，减除数；
 - 当余数为负时，商0，余数左移一位，加除数；
 - 上述步骤重复 $n+1$ 次（ n 位尾数，1位为符号位）得到商的绝对值，最后一步余数不左移
 - 最后一步余数为负值时，需要加上 $|y|$ 得到正确的余数
- 特点：**计算步数固定，控制简单。

加减交替除法正确性说明

对于**恢复余数法**，假设第 i 次余数为 R_i

➤ 如果 R_i 为正数，那么 R_{i+1} 的值为：

$$R_{i+1} = 2R_i - |y|$$

➤ 如果 R_i 为负数，那么 R_{i+1} 的值为：

恢复余数法：先恢复余数然后，左移1位，再减去 $|y|$

$$R_{i+1} = 2(R_i + |y|) - |y| = 2R_i + |y|$$

加减交替法：余数左移加上 $|y|$

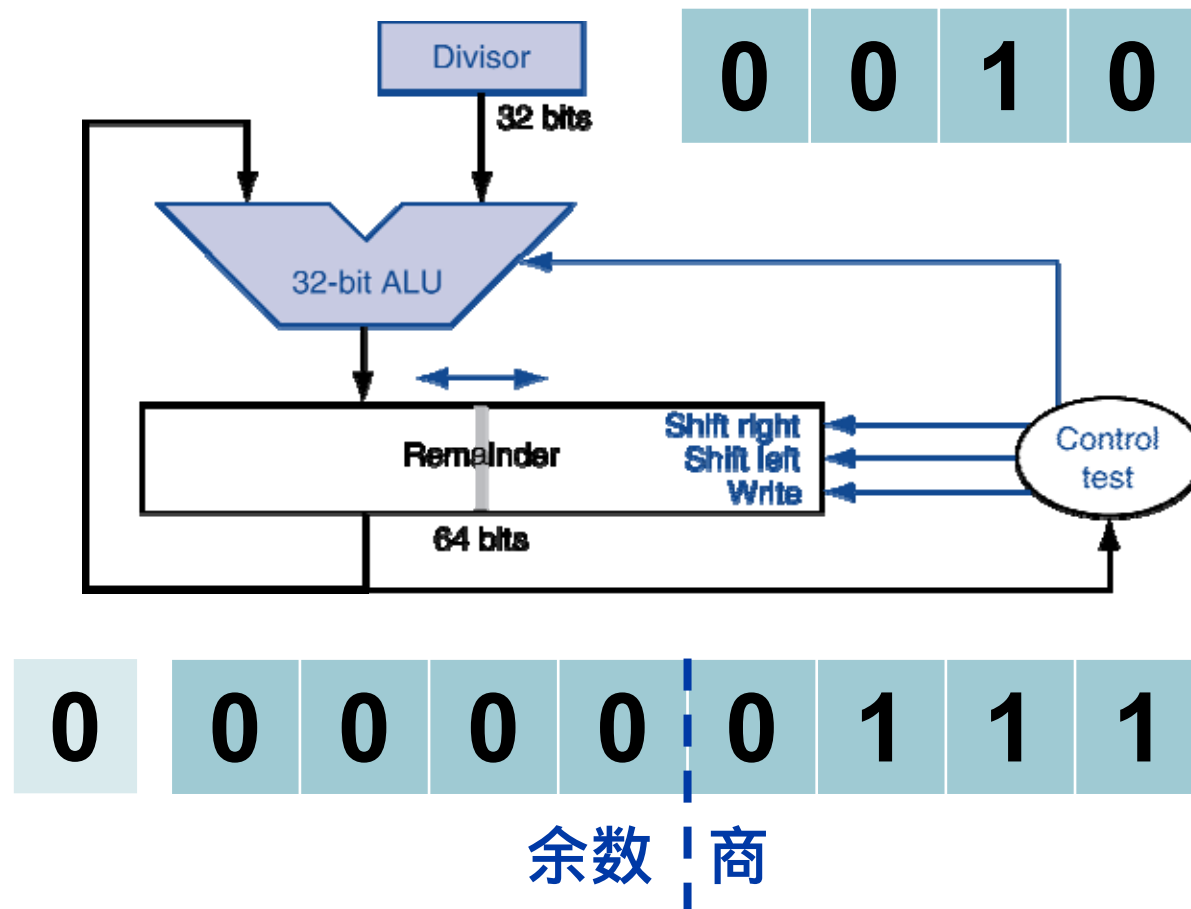
$$R_{i+1} = 2R_i + |y|$$

因此这两种算法计算结果是一样的！

加減交替法

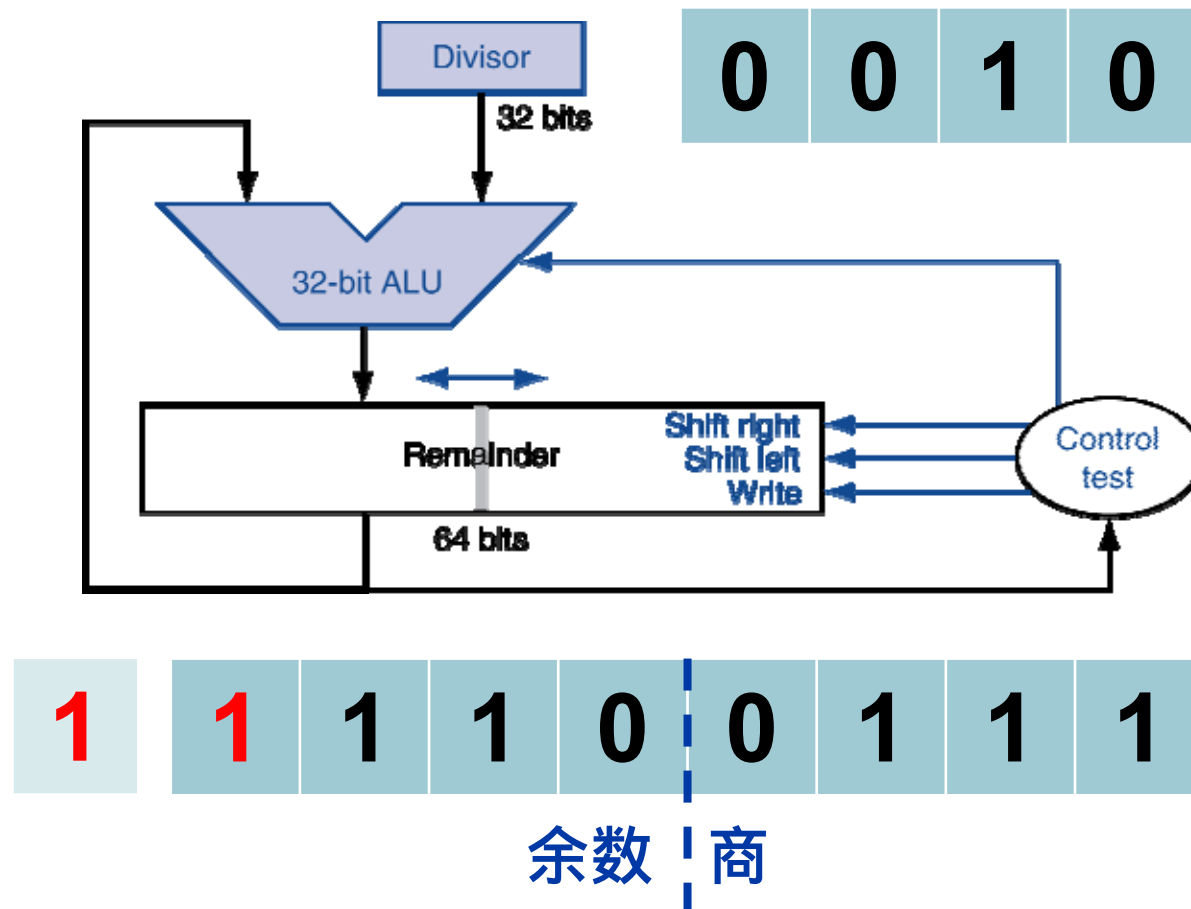
Step 0 初始化

被除数 = $[0111]_2$ 除数 = $[0010]_2$



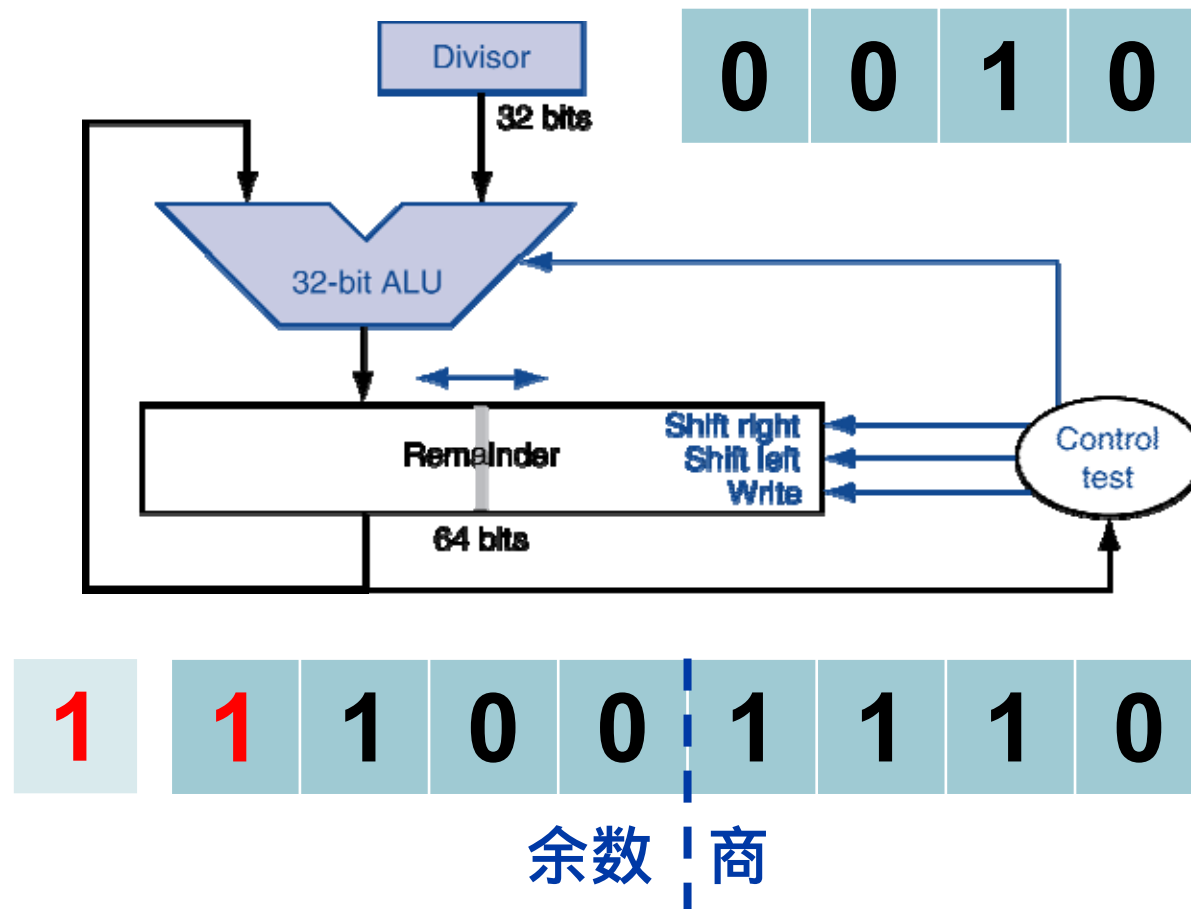
加减交替法

Step 1.1 余数>0, 余数-除数 试商



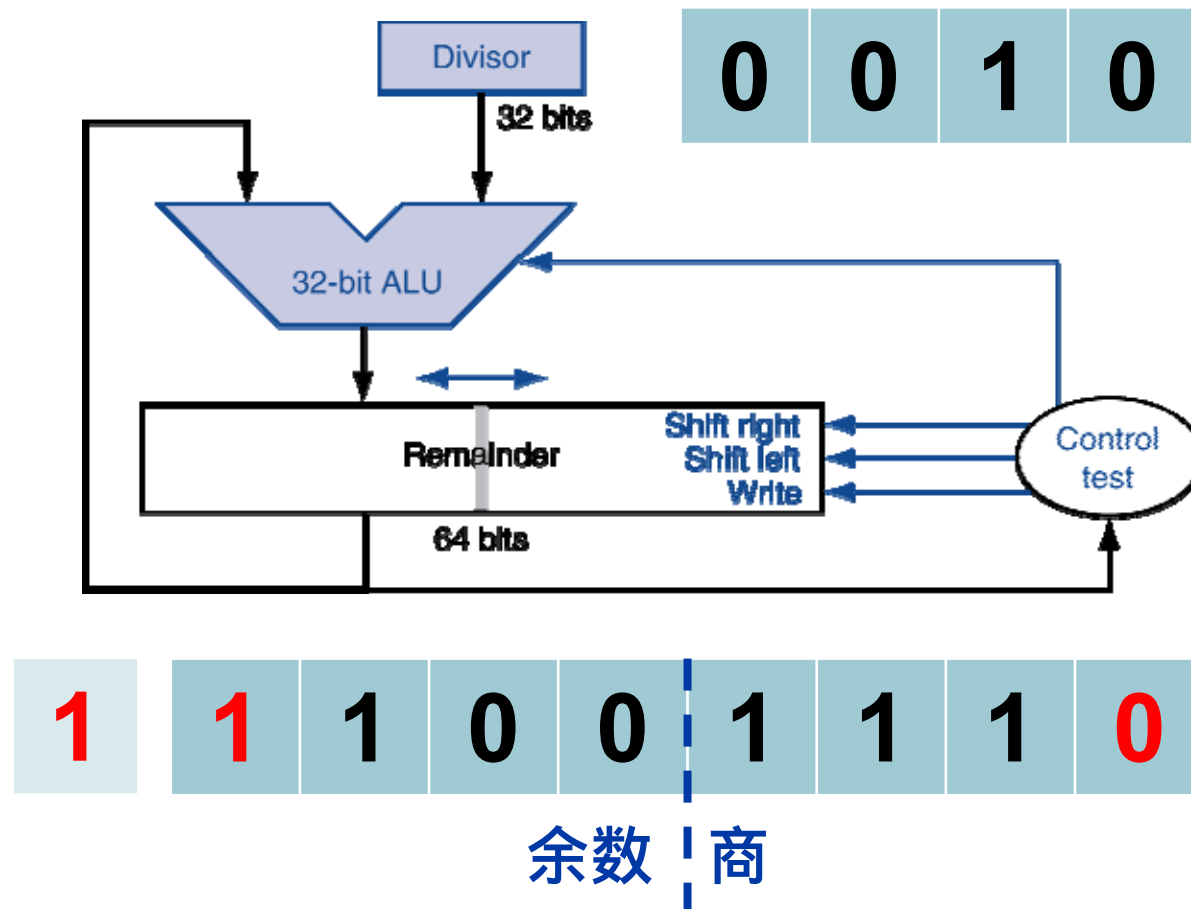
加減交替法

Step 1.2 余数和商 左移1位



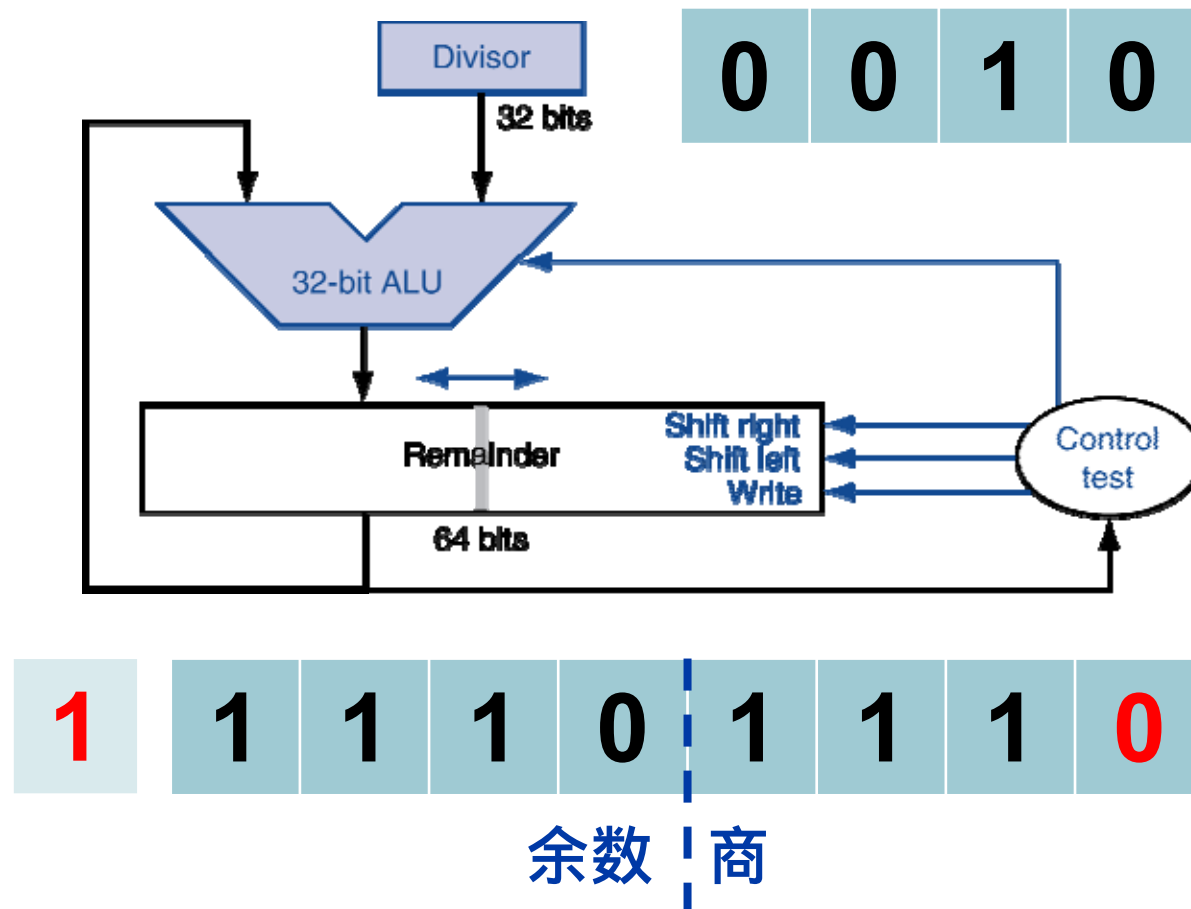
加減交替法

Step 1.3 余数 <0 ，商的最低位置0



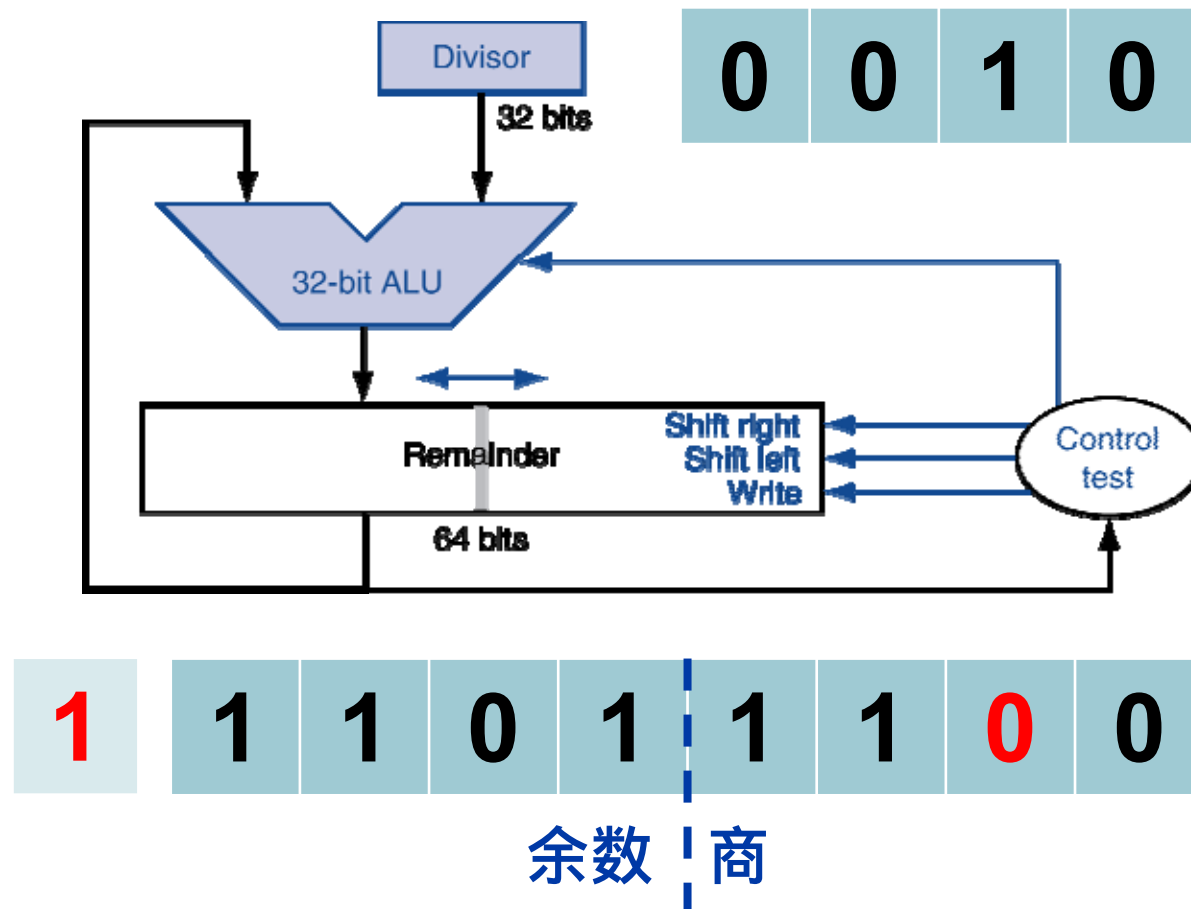
加減交替法

Step 2.1 余数<0, 余数+除数 试商



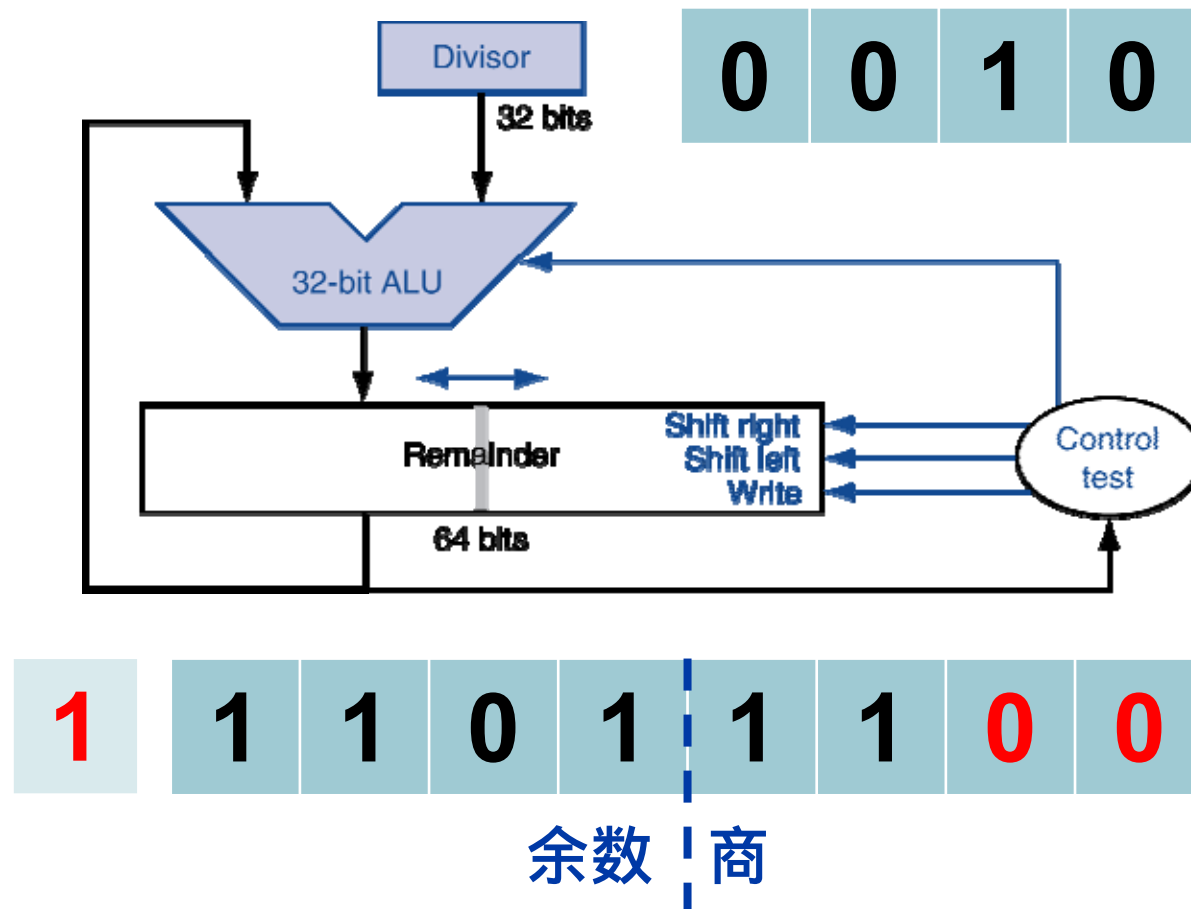
加減交替法

Step 2.2 余数和商 左移1位



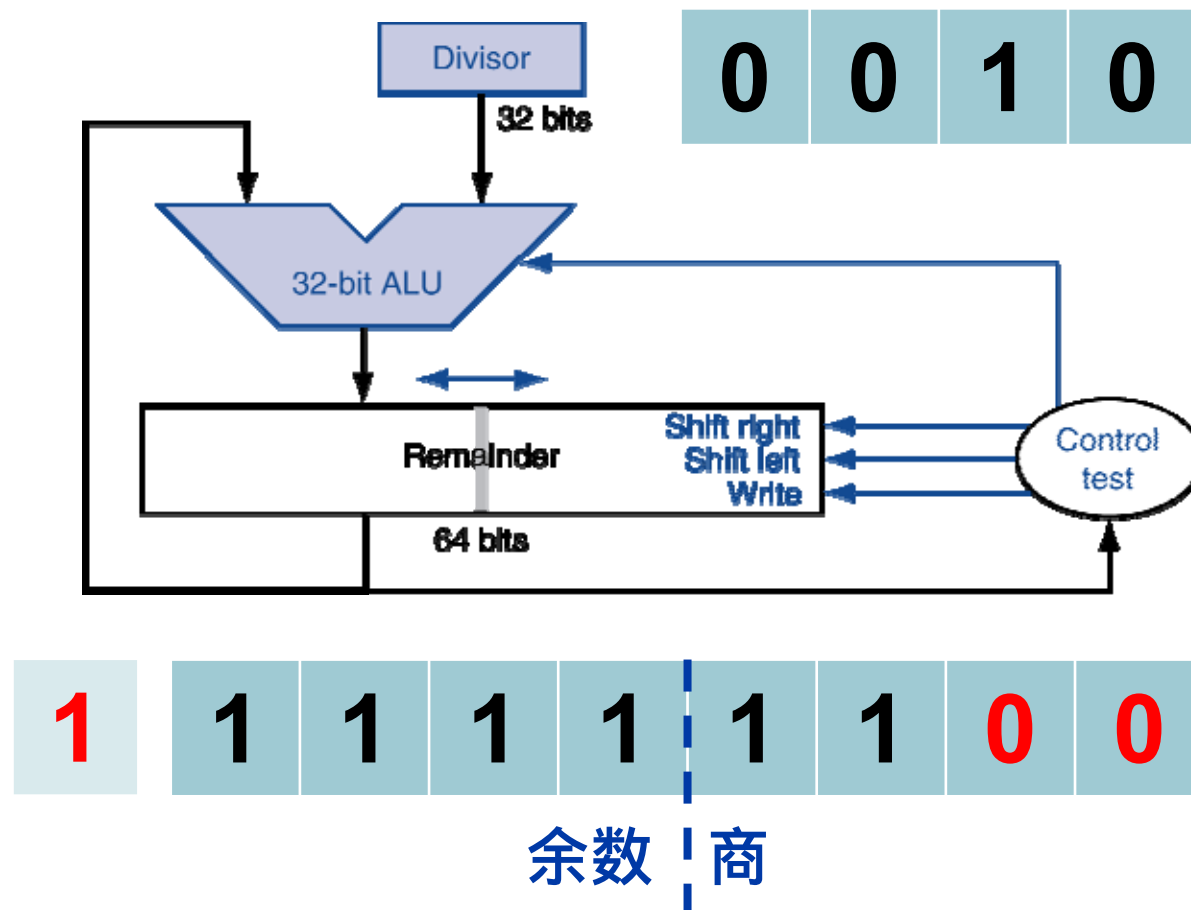
加減交替法

Step 2.3 余数 <0 ，商的最低位置0



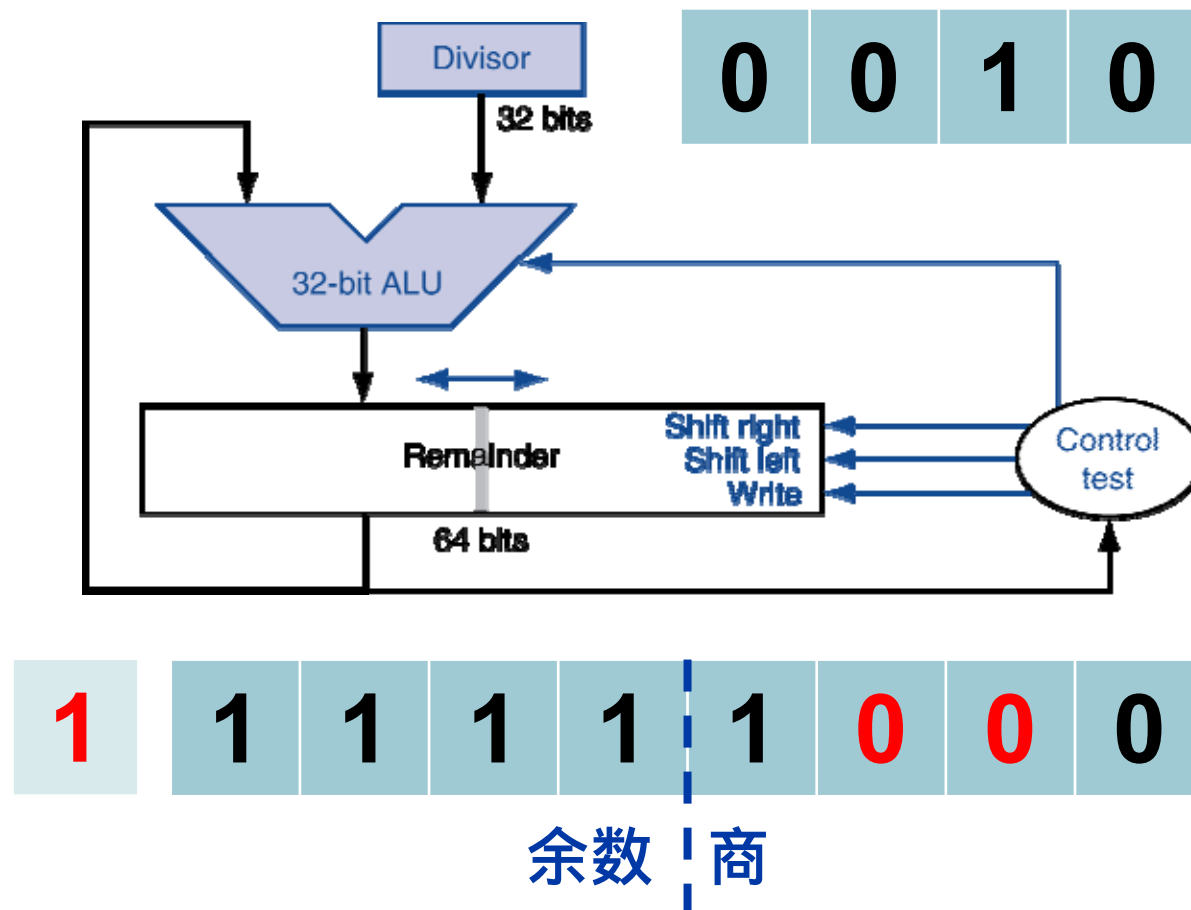
加减交替法

Step 3.1 余数<0, 余数+除数 试商



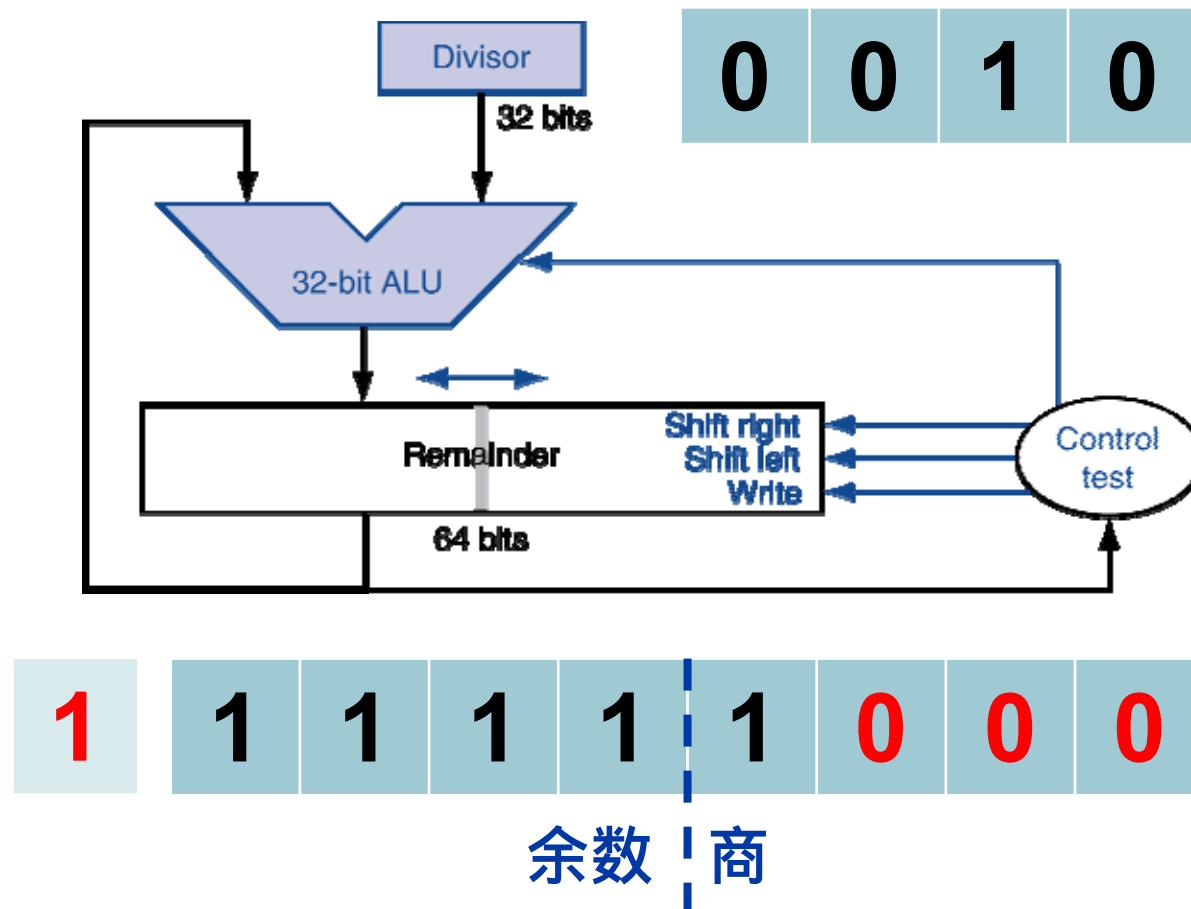
加減交替法

Step 3.2 余数和商 左移1位



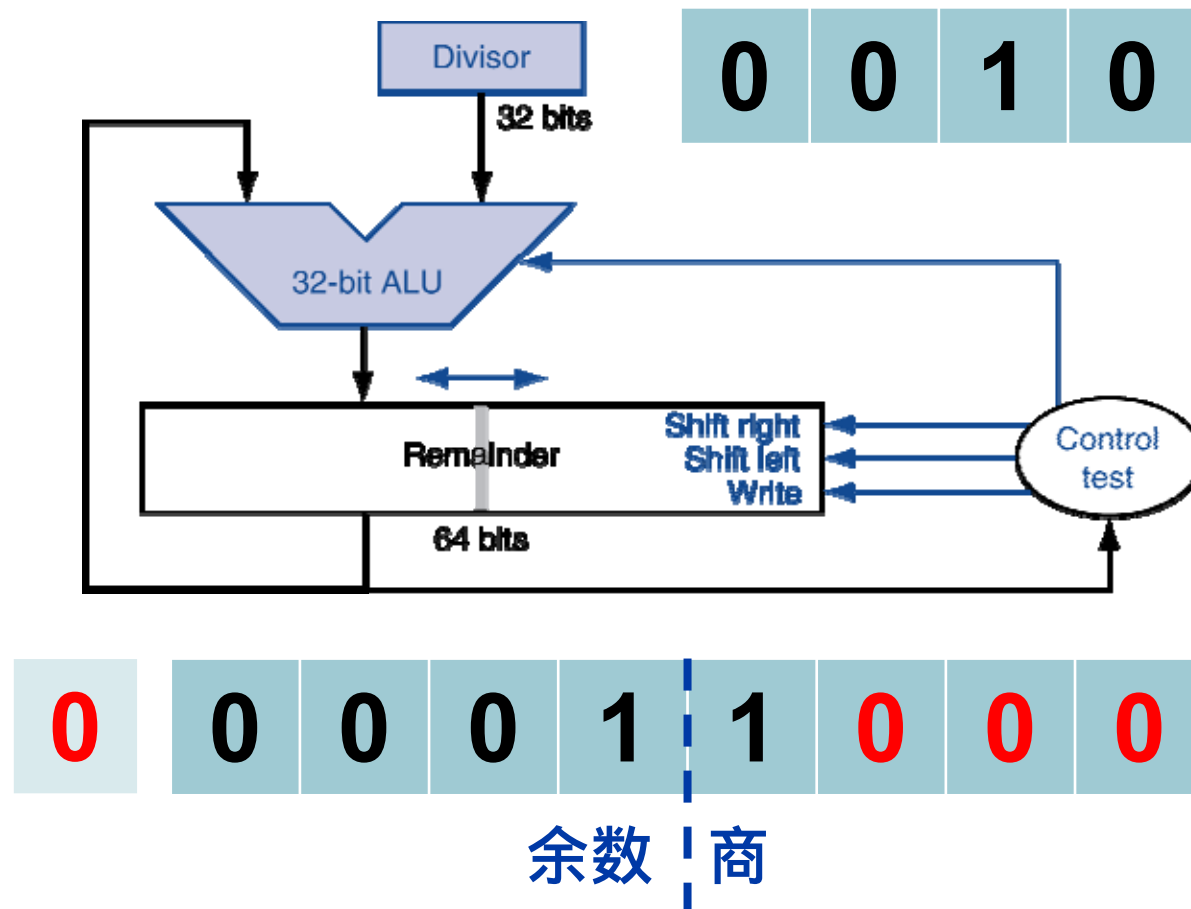
加減交替法

Step 3.3 余数 <0 ，商的最低位置0



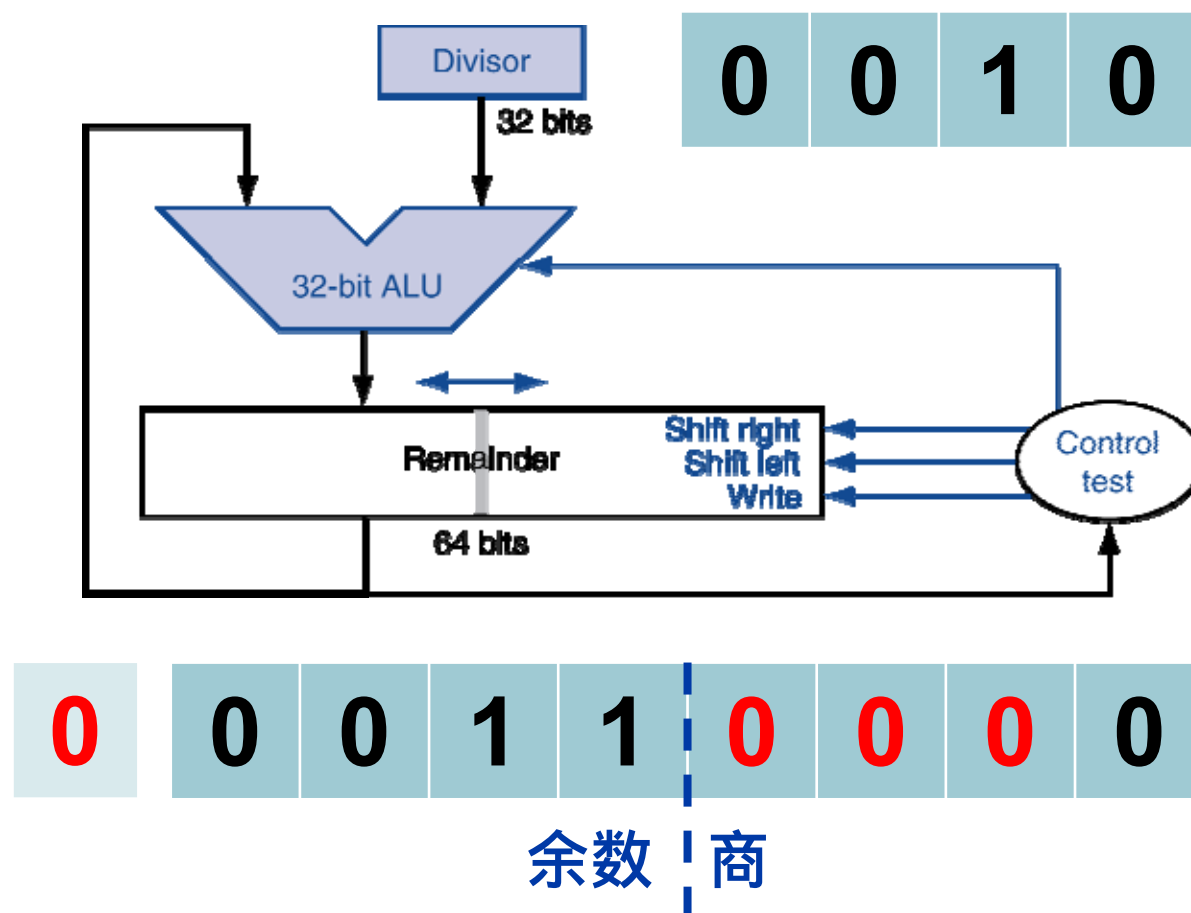
加減交替法

Step 4.1 余数<0, 余数+除数 试商



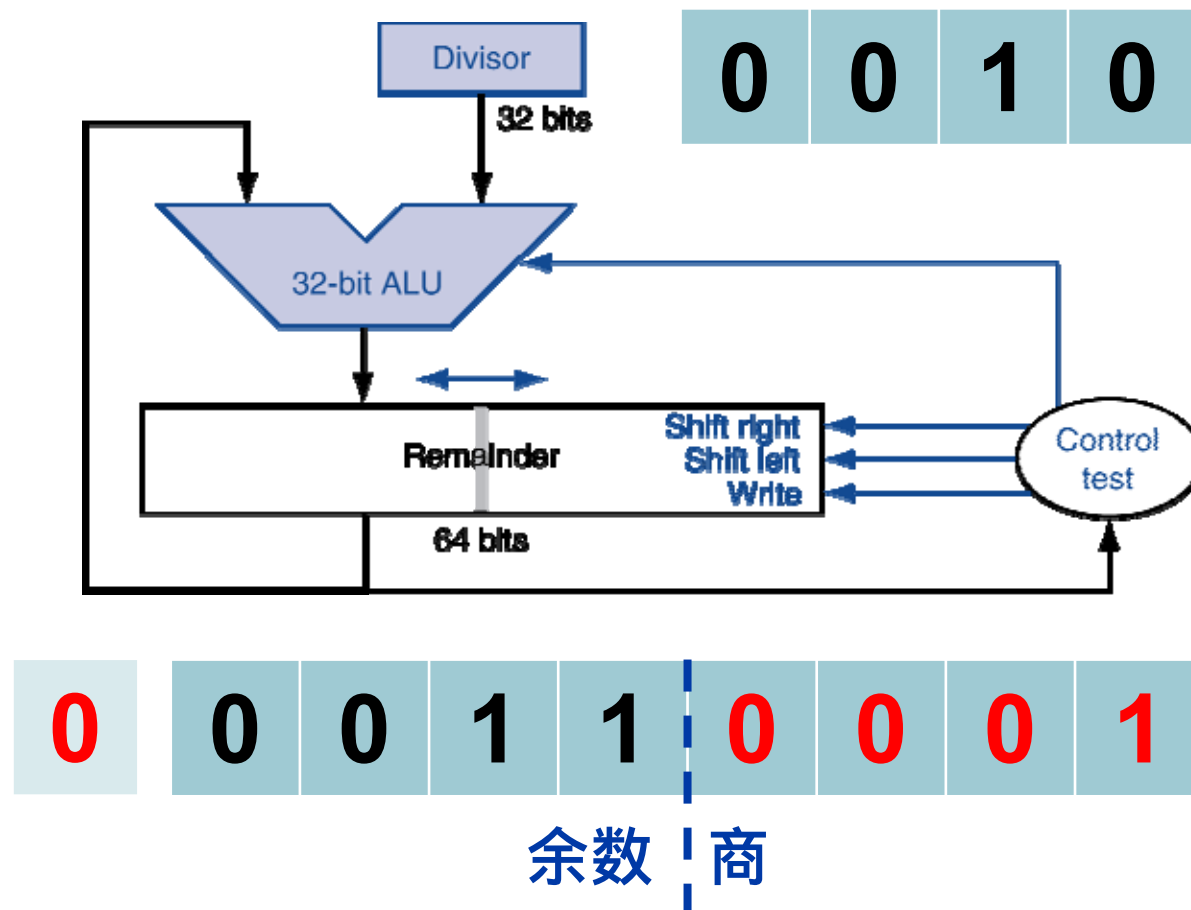
加減交替法

Step 4.2 余数和商 左移1位



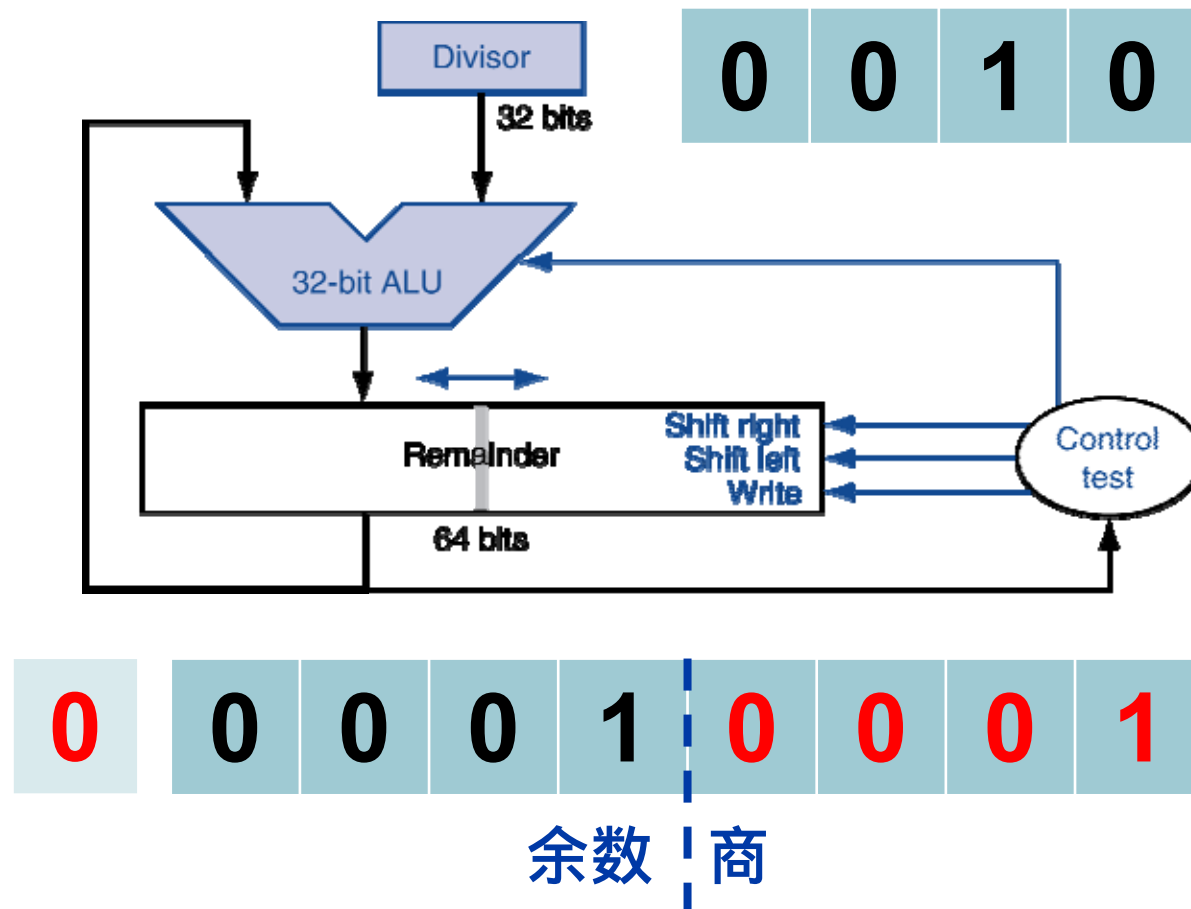
加減交替法

Step 4.3 余数 ≥ 0 ，商的最低位置1



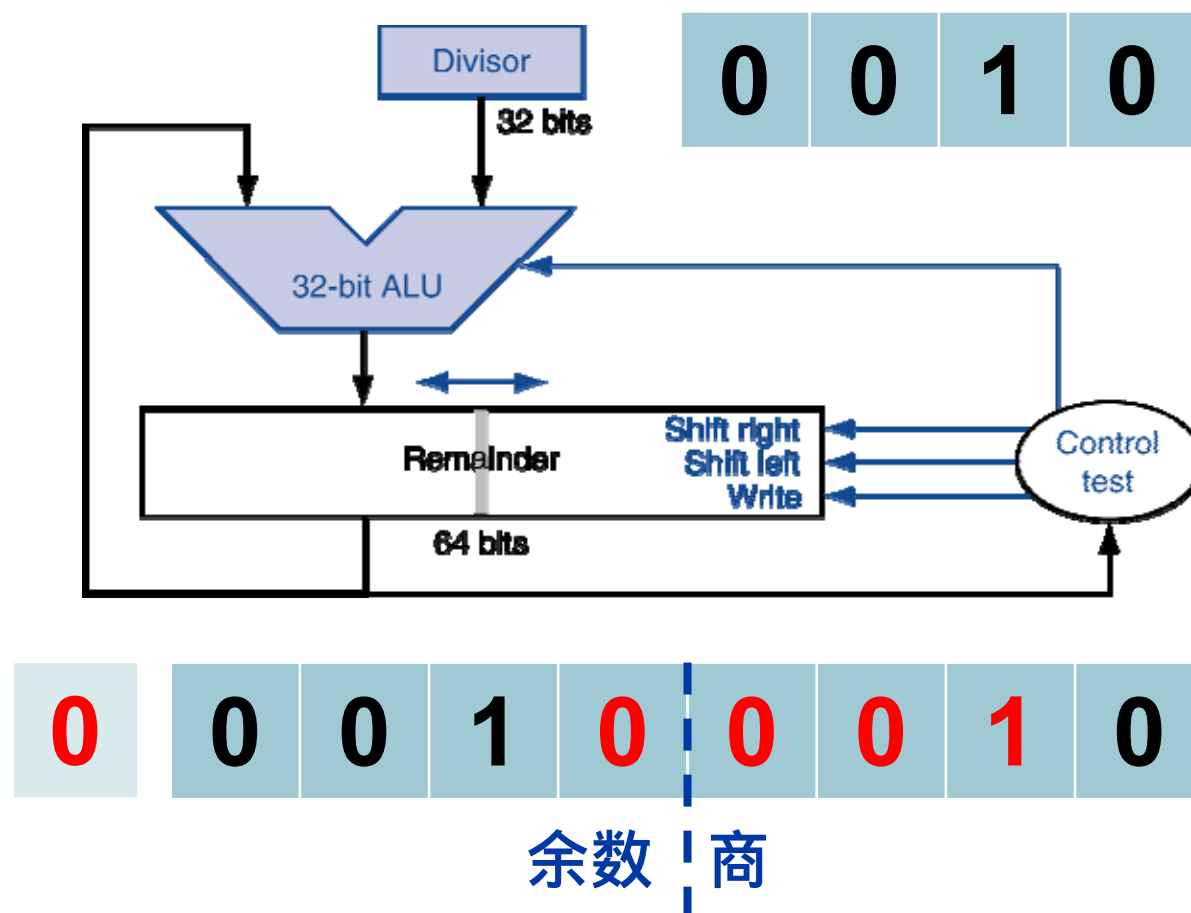
加减交替法

Step 5.1 余数>0, 余数-除数 试商



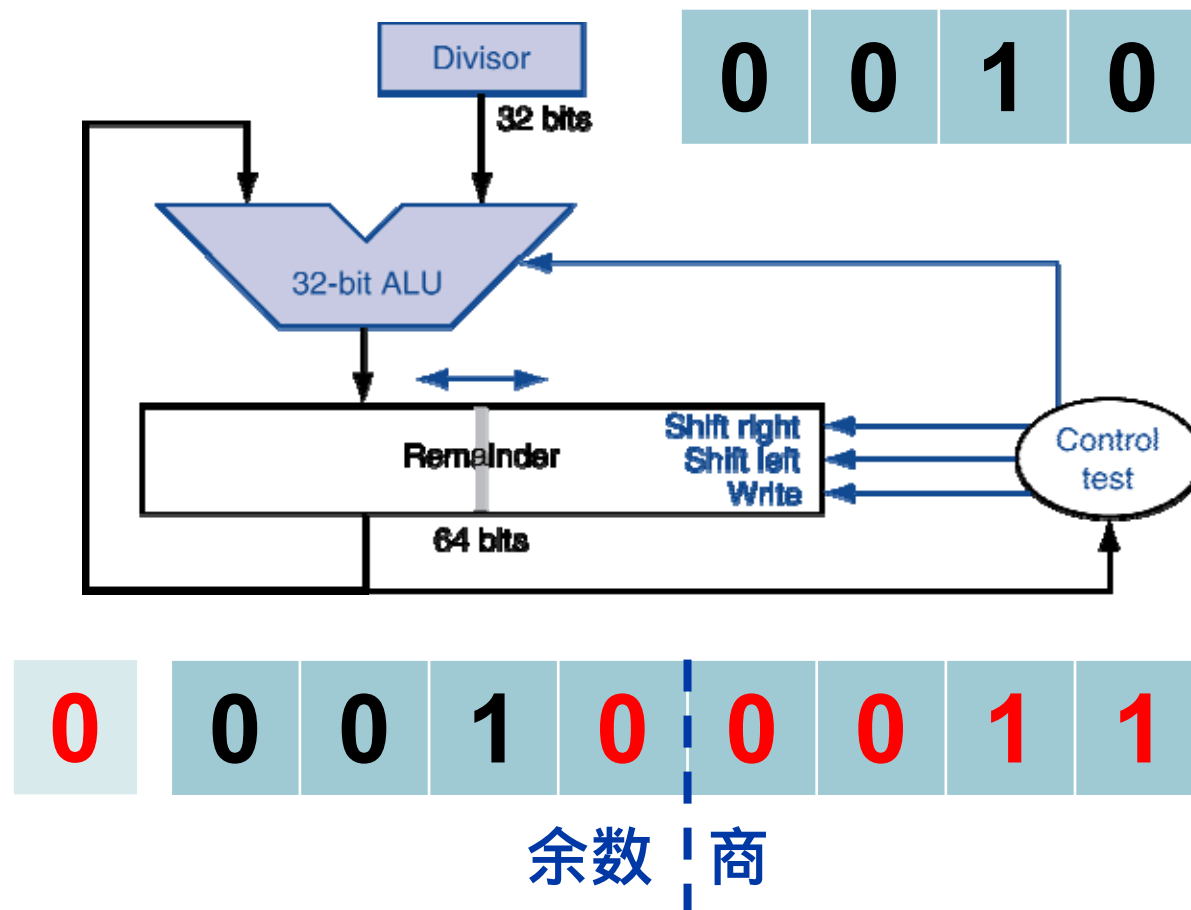
加減交替法

Step 5.2 余数和商 左移1位



加減交替法

Step 5.3 余数 ≥ 0 ，商的最低位置1



令 $x = [0111]_2$ $y = [0010]_2$ 加减交替计算过程

| 迭代次数 | 步骤 | 除数 | 余数 商 |
|------|--|------|---|
| 0 | 初始化 | 0010 | 0 0000 0111 |
| 1 | 当前余数为正，余数=余数-除数 余数<0，左移1位 余数最低位置0 | 0010 | 1 1110 0111 1 1100 1110 1 1100 1110 |
| 2 | 当前余数<0,余数=余数+除数 余数并左移1位 余数最低位置0 | 0010 | 1 1110 1110 1 1101 1100 1 1101 1100 |
| 3 | 当前余数<0，余数=余数+除数 余数<0 恢复余数并左移1位 余数最低位置0 | 0010 | 1 1111 1100 1 1111 1000 1 1111 1000 |
| 4 | 当前余数<0余数=余数+除数 余数>0，余数左移1位 余数最低位置1 | 0010 | 0 0001 1000 0 0011 0000 0 0011 0001 |
| 5 | 当前余数>0,余数=余数-除数 余数>0，余数左移1位 余数最低位置1 | 0010 | 0 0001 0001 0 0010 0010 0 0010 0011 |

商为 $= [0011]_2$ 余数为 $= [0000 0001]_2$

3.4 Division

- 十进制整数除法回顾
- 二进制整数除法
- 原码1位除法思想
- 恢复余数除法及其优化
- 加减交替除法
- 定点小数除法
- 快速除法概况

定点小数原码1位除法(加减交替)

算法：小数求商和余数算法

输入：n位的二进制小数被除数X和除数Y

输出：商Q和余数R

step 0: 初始化, $R=X$, $t=0$

step 1. if $R>0$ $R=R-X$ else $R=R+X$

step 2. left_shift(R), left_shift(Q), $t=t+1$

step 3 if $R\geq 0$ $Q(n)=1$ else $Q(n)=0$

step 4 if $t < n$ goto step 1;

step 5 输出 R , Q

定点小数原码1位除法

算法推广到定点小数时需要注意：

- 被除数后面补零扩展n位

与整数除法一项，被除数的有效值保持不变。

- 参与运算的小数必须是规格化小数

绝对值 ≥ 0.5 ，原码表示的小数点后第一位为0

- 加法器运算时采用双符号位

这是运算结果可以是 $[-2, 2)$ 的值。

定点小数除法采用双符号位补码（模4补码）

定点小数的补码采用2个符号位来表示数字的符号

例如：00.1101 或者11.1010

如果其表示范围为 $[-1, 1)$ 最终的结果为 10 或则 01，则表示其结果存在溢出。

如果采用双符号位，计算结果都是正确的无需特别处理。

加减交替法推广到定点小数的除法运算

| 被除数x / 余数 r | 商q | 说明 |
|---|--------|---|
| $\begin{array}{r} 00.1001 \\ +[-y]_{\text{补}} 11.0101 \\ \hline 11.1110 \end{array}$ | 0.0000 | $[x]_{\text{原}} = 00.1001,$ $[y]_{\text{补}} = 00.1011,$ $[-y]_{\text{补}} = 11.0101$ x减y 余数 $r_0 < 0$, r和q左移一位, 商0 |
| $\begin{array}{r} \leftarrow 11.1100 \\ +[y]_{\text{补}} 00.1011 \\ \hline 00.0111 \end{array}$ | 0.0000 | 加y 余数 $r_1 > 0$ r和q左移一位, 商1 |
| $\begin{array}{r} \leftarrow 00.1110 \\ +[-y]_{\text{补}} 11.0101 \\ \hline 00.0011 \end{array}$ | 0.0001 | 减y 余数 $r_2 > 0$ r和q左移一位, 商1 |
| $\begin{array}{r} \leftarrow 00.0110 \\ +[-y]_{\text{补}} 11.0101 \\ \hline 11.1011 \end{array}$ | 0.0011 | 减y 余数 $r_3 < 0$ 商0, r和q左移一位 |
| $\begin{array}{r} \leftarrow 11.0110 \\ +[y]_{\text{补}} 00.1011 \\ \hline 00.0001 \end{array}$ | 0.0110 | 加y 余数 $r_4 > 0$ 商1, 仅q左移一位 |
| 00.0001 | 0.1101 | |

得: $q = x/y = 0.1101$

余数 $r = 2^{-4} r_4 = 0.00000001$

Faster Division

- 除法运算无法向乘法一样并行化
 - 因为减法操作需要根据余数的符号位条件执行
- 快速除法器 **SRT division**
 - 通过查表的方式来获取多位 (2/3/4) 的商
 - 在后续的步骤中矫正错误取值
 - 需要多次迭代
- 其它快速除法器
 - 函数迭代算法(Newton-Raphson等 近似计算)

Atkins, Daniel. (1968). Higher-Radix Division Using Estimates of the Divisor and Partial Remainders. IEEE Transactions on Computers, pp. 925-934.

王县,倪晓强,邢座程, 浮点除法算法的分析与研究, 第十二届计算机工程与工艺学术年会, 2008

MIPS Division

- Use HI/LO registers for result
 - **HI**: 32-bit remainder
 - **LO**: 32-bit quotient
- Instructions
 - `div rs, rt` / `divu rs, rt`
 - Use `mfhi` , `mflo` to access result
 - No overflow or divide-by-0 checking
 - Software must perform checks if required

$$[X]_{\text{原}} = 0.1001$$

$$[Y]_{\text{原}} = 0.1101$$

请采用加减交替法计算 $[X/Y]_{\text{原}}$ 的结果

15分钟完成，选择适当的算法需要给出具体的步骤

得： $q = x/y = 0.1011$

余数 $r = 0.00000001$

正常使用主观题需2.0以上版本雨课堂

作答

参考答案 $x=[0.1001]$ $y=[0.1101]$, 计算 x/y 的值

| 被除数x / 余数 r | 商q | 说明 |
|---|--------|---|
| $\begin{array}{r} 00.1001 \\ +[-y]_{\text{补}} 11.0011 \\ \hline 11.1100 \end{array}$ | 0.0000 | $[x]_{\text{原}} = 00.1001,$ $[y]_{\text{补}} = 00.1101,$ $[-y]_{\text{补}} = 11.0011$ x减y 余数 $r_0 < 0$, r和q左移一位, 商0 |
| $\begin{array}{r} \leftarrow 11.1000 \\ +[y]_{\text{补}} 00.1101 \\ \hline 00.0101 \end{array}$ | 0.0000 | 加y 余数 $r_1 > 0$ r和q左移一位, 商1 |
| $\begin{array}{r} \leftarrow 00.1010 \\ +[-y]_{\text{补}} 11.0011 \\ \hline 11.1101 \end{array}$ | 0.0001 | 减y 余数 $r_2 < 0$ r和q左移一位, 商0 |
| $\begin{array}{r} \leftarrow 11.1010 \\ +[y]_{\text{补}} 00.1101 \\ \hline 00.0111 \end{array}$ | 0.0010 | 减y 余数 $r_3 > 0$ 商1, r和q左移一位 |
| $\begin{array}{r} \leftarrow 00.1110 \\ +[-y]_{\text{补}} 11.0011 \\ \hline 00.0001 \end{array}$ | 0.0101 | 加y 余数 $r_4 > 0$ 商1, 仅q左移一位 |
| 00.0001 | 0.1011 | |

得: $q = x/y = 0.1011$

余数 $r = 2^{-4} r_4 = 0.00000001$

Floating Point

- Representation for non-integral numbers
 - Including very small and very large numbers
- Like scientific notation
 - -2.34×10^{56} ← normalized
 - $+0.002 \times 10^{-4}$ ← not normalized
 - $+987.02 \times 10^9$ ← not normalized
- In binary
 - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types **float** and **double** in C

Floating Point Standard

- Defined by IEEE 754 Standard
- Developed in response to divergence of representations
 - Portability issues for scientific code
- Now almost universally adopted
 - In every computer invented since 1980
- Two representations
 - **Single** precision (**32-bit**)
 - **Double** precision (**64-bit**)
 - reduces chance of **underflow** and **overflow**

IEEE Floating-Point Format

single: 8 bits

double: 11 bits

single: 23 bits

double: 52 bits

| S | Exponent | Fraction |
|---|----------|----------|
|---|----------|----------|

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 \Rightarrow non-negative, 1 \Rightarrow negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
 - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
 - Significand is Fraction with the “1.” restored
- Exponent: excess representation: **actual exponent + Bias**
 - Ensures exponent is unsigned
 - Single: Bias = 127; Double: Bias = 1023

Single-Precision Range

- Exponents 00000000 and 11111111 reserved
- **Smallest value**
 - Exponent: 00000001
 \Rightarrow actual exponent = $1 - 127 = -126$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- **Largest value**
 - exponent: 11111110
 \Rightarrow actual exponent = $254 - 127 = +127$
 - Fraction: 111...11 \Rightarrow significand ≈ 2.0
 - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000...00 and 1111...11 reserved
- **Smallest value**
 - Exponent: 000000000001
 \Rightarrow actual exponent = $1 - 1023 = -1022$
 - Fraction: 000...00 \Rightarrow significand = 1.0
 - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- **Largest value**
 - Exponent: 111111111110
 \Rightarrow actual exponent = $2046 - 1023 = +1023$
 - Fraction: 111...11 \Rightarrow significand ≈ 2.0
 - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

■ Relative precision

- all fraction bits are significant
- **Single:** approx 2^{-23}
 - Equivalent to $23 \times \log_{10} 2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
- **Double:** approx 2^{-52}
 - Equivalent to $52 \times \log_{10} 2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

| Single precision | | Double precision | | Object represented |
|------------------|----------|------------------|----------|-----------------------------|
| Exponent | Fraction | Exponent | Fraction | |
| 0 | 0 | 0 | 0 | 0 |
| 0 | Nonzero | 0 | Nonzero | \pm denormalized number |
| 1–254 | Anything | 1–2046 | Anything | \pm floating-point number |
| 255 | 0 | 2047 | 0 | \pm infinity |
| 255 | Nonzero | 2047 | Nonzero | NaN (Not a Number) |

Floating-Point Example

- Represent -0.75
 - $-0.75 = -0.11_2 = (-1)^1 \times 1.1_2 \times 2^{-1}$
 - $S = 1$
 - Fraction = $1000\dots00_2$
 - Exponent = $-1 + \text{Bias}$
 - Single: $-1 + 127 = 126 = 01111110_2$
 - Double: $-1 + 1023 = 1022 = 011111111110_2$
- Single: $10111111101000\dots00$
- Double: $101111111111101000\dots00$

Floating-Point Example

- What number is represented by the single-precision float

11000000101000...00

- $S = 1$
- Fraction = $01000...00_2$
- Exponent = $10000001_2 = 129$

- $$\begin{aligned}x &= (-1)^1 \times (1 + .01_2) \times 2^{(129 - 127)} \\&= (-1) \times 1.25 \times 2^2 \\&= -5.0\end{aligned}$$

Floating-Point Addition

- Consider a 4-digit decimal example
 - $9.999 \times 10^1 + 1.610 \times 10^{-1}$
- 1. Align decimal points
 - Shift number with smaller exponent
 - $9.999 \times 10^1 + 0.016 \times 10^1$
- 2. Add significands
 - $9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$
- 3. Normalize result & check for over/underflow
 - 1.0015×10^2
- 4. Round and renormalize if necessary
 - 1.002×10^2

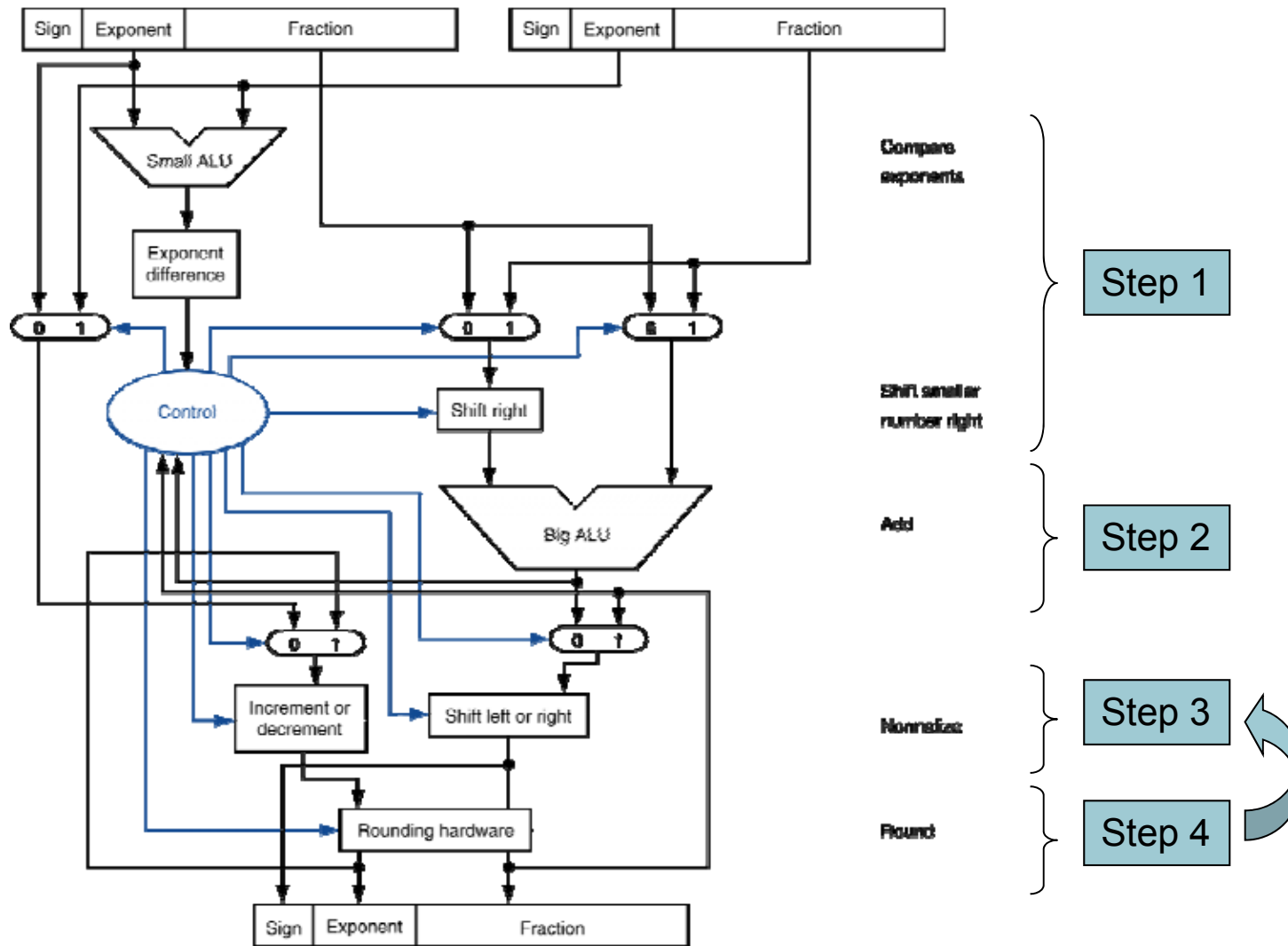
Floating-Point Addition

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + -0.4375)
- 1. Align binary points
 - Shift number with smaller exponent
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
 - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
 - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
 - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
 - Much longer than integer operations
 - Slower clock would penalize all instructions
- FP adder usually takes several cycles
 - Can be pipelined

FP Adder Hardware



Floating-Point Multiplication

- Consider a 4-digit decimal example
 - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
 - For biased exponents, subtract bias from sum
 - New exponent = $10 + -5 = 5$
- 2. Multiply significands
 - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^5$
- 3. Normalize result & check for over/underflow
 - 1.0212×10^6
- 4. Round and renormalize if necessary
 - 1.021×10^6
- 5. Determine sign of result from signs of operands
 - $+1.021 \times 10^6$

Floating-Point Multiplication

- Now consider a 4-digit binary example
 - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5×-0.4375)
- 1. Add exponents
 - Unbiased: $-1 + -2 = -3$
 - Biased: $(-1 + 127) + (-2 + 127) = -3 + 254 - 127 = -3 + 127$
- 2. Multiply significands
 - $1.000_2 \times 1.110_2 = 1.1102 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
 - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
 - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: $+ve \times -ve \Rightarrow -ve$
 - $-1.110_2 \times 2^{-3} = -0.21875$

Computer Organization and Design

4.1-4.4

The Processor

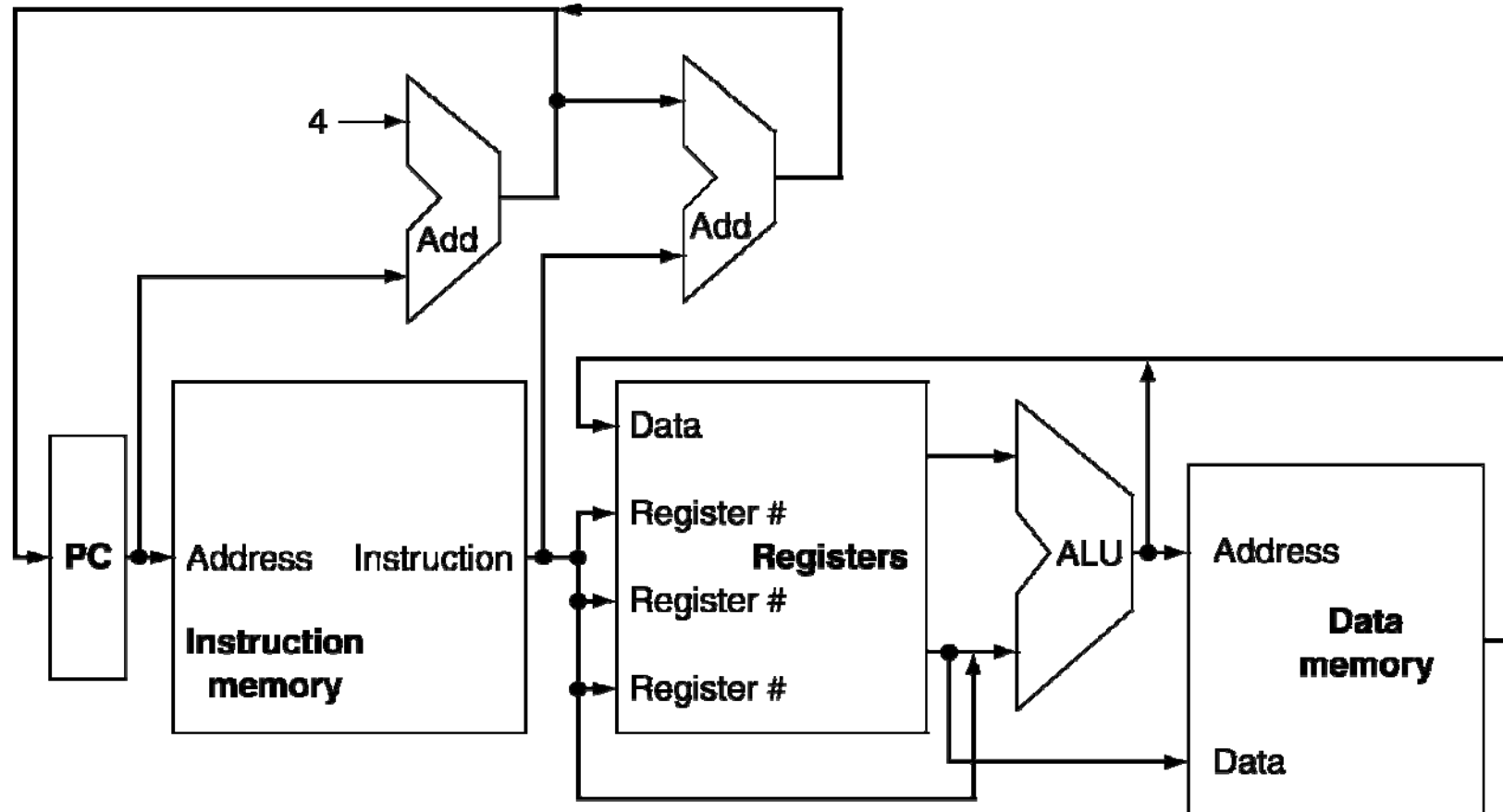
Introduction

- CPU performance factors
 - Instruction count
 - Determined by ISA and compiler
 - CPI and Clock cycle time
 - Determined by CPU hardware
- We will examine two MIPS CPU implementations
 - A simplified version(组成原理, 单总线CPU)
 - A more realistic pipelined version(系统结构, 流水线)
- Simple instruction subset, shows most aspects
 - Memory reference: `lw, sw`
 - Arithmetic/logical: `add, sub, and, or, slt`
 - Control transfer: `beq, j`

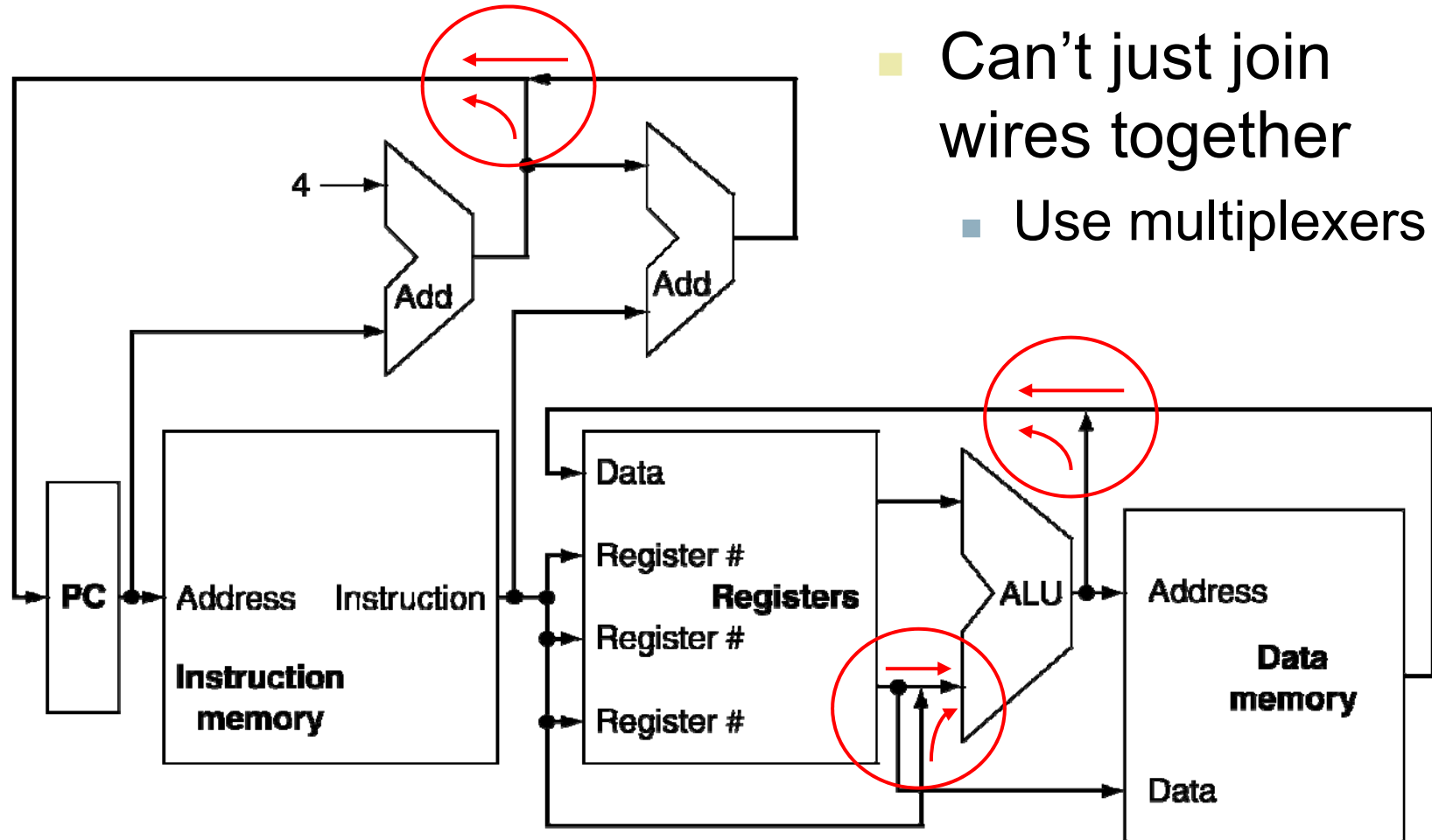
Instruction Execution: Main Steps

- PC \rightarrow instruction memory, fetch instruction
- Register numbers \rightarrow register file, read registers
- Depending on instruction class
 - Use ALU to calculate(无论哪种类型, MIPS的执行大致相同, 与具体指令无密切关系)
 - Arithmetic result
 - Memory address for load/store
 - Branch target address
 - Write result to register file (optional)
 - Access data memory for load/store (optional)
 - PC \leftarrow target address or PC + 4

CPU Overview



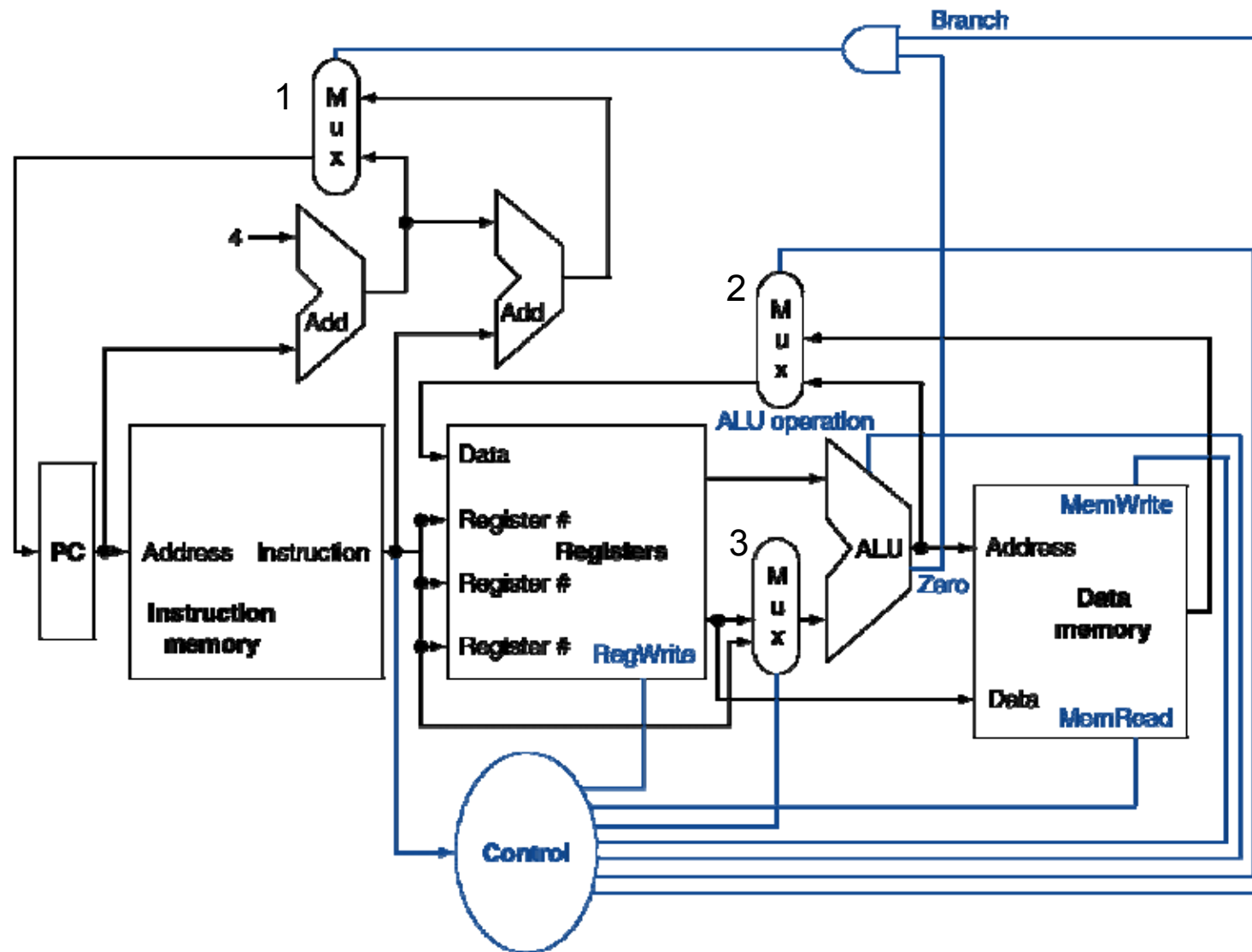
Multiplexers



- Can't just join wires together
 - Use multiplexers

Reference: Appendix C

Control



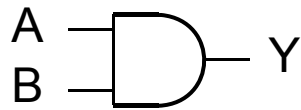
Logic Design Basics

- Information encoded in binary
 - Low voltage = 0, High voltage = 1
 - One wire per bit
 - Multi-bit data encoded on multi-wire buses
- Combinational element(组合单元)
 - Operate on data
 - Output is a function of input
- State (sequential) elements(时序/状态单元)
 - Store information

Combinational Elements

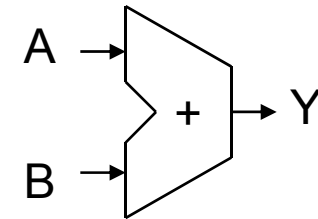
- AND-gate

- $Y = A \& B$



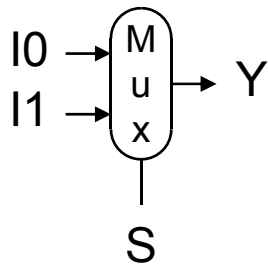
- Adder

- $Y = A + B$



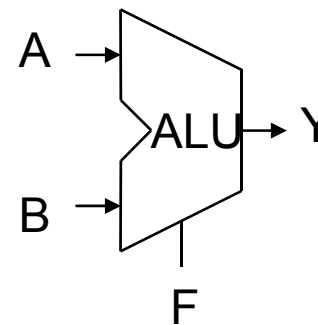
- Multiplexer

- $Y = S ? I1 : I0$



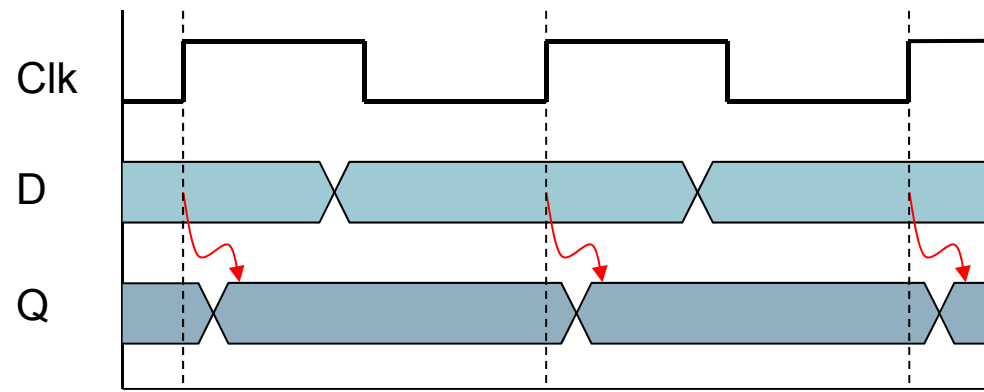
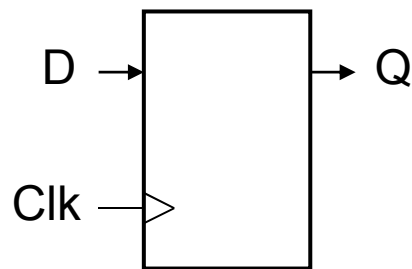
- Arithmetic/Logic Unit

- $Y = F(A, B)$



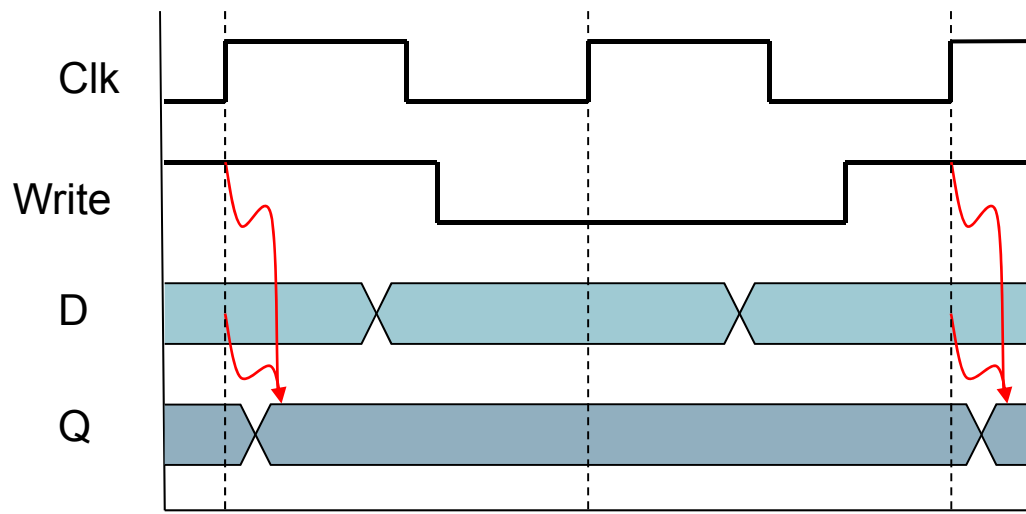
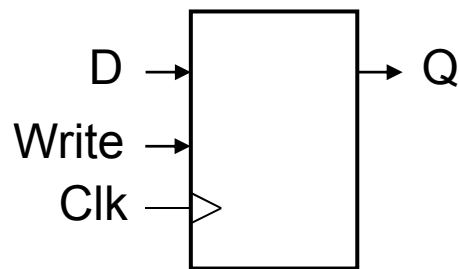
Sequential Elements

- **Register:** stores data in a circuit
 - Uses a clock signal to determine when to update the stored value
 - Edge-triggered: update when Clk changes from 0 to 1



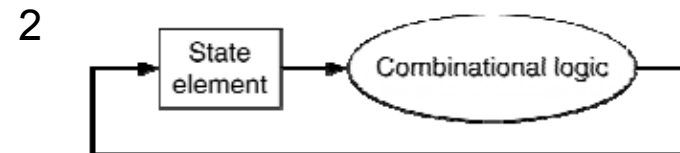
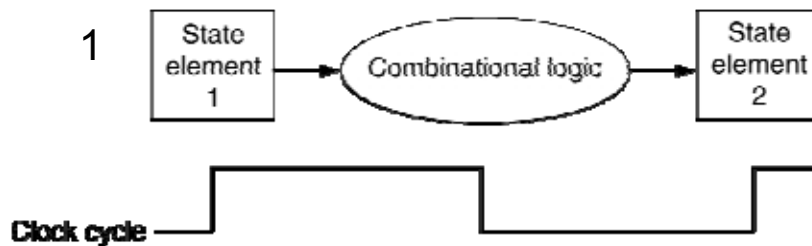
Sequential Elements

- Register with write control
 - Only updates on clock edge when write control input is 1
 - Used when stored value is required later



Clocking Methodology

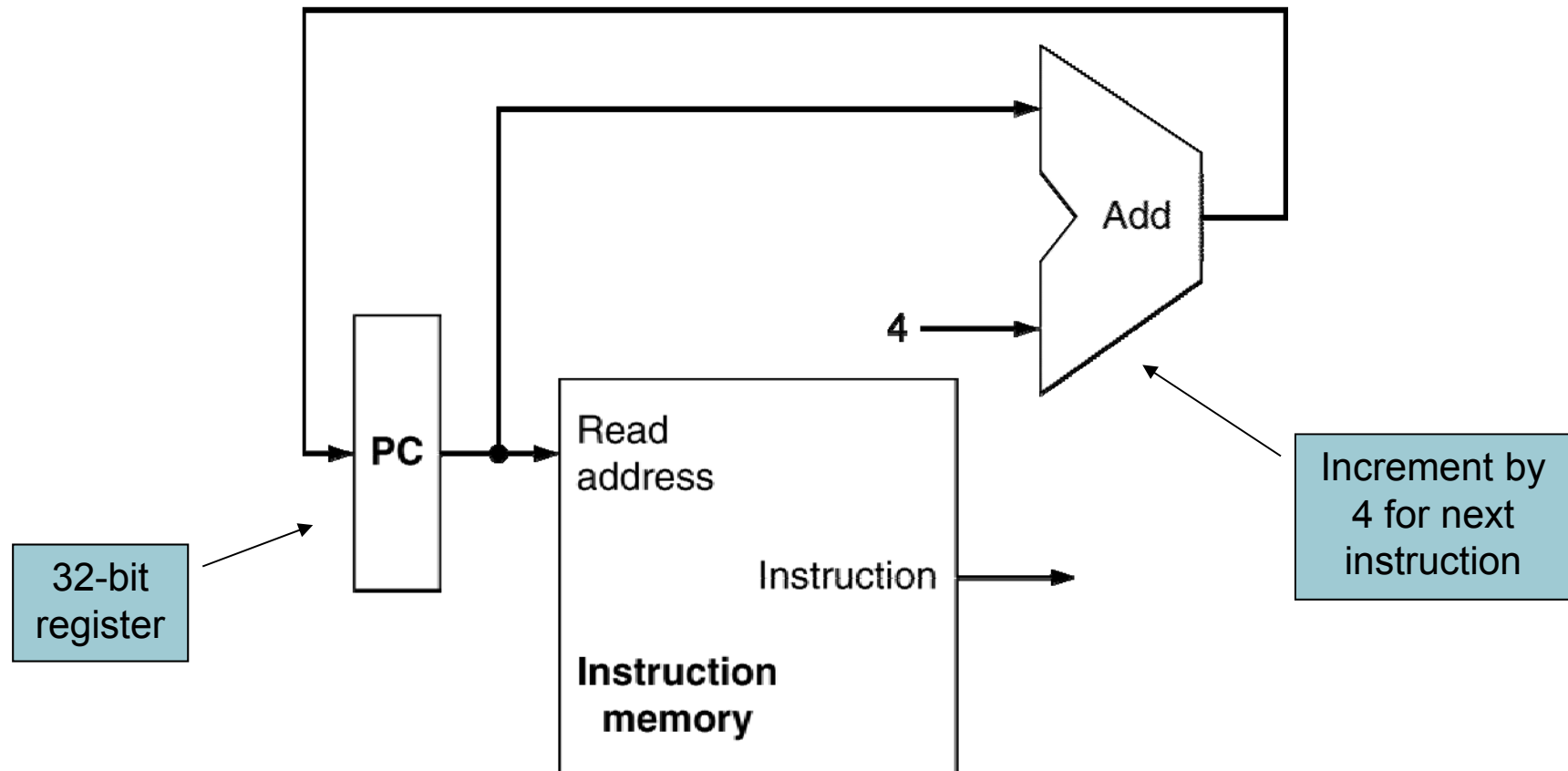
- Combinational logic transforms data during clock cycles
 - Between clock edges
 - Input from state elements, output to state element
 - Longest delay determines clock period
 - Must keep “critical path” as short as possible



Building a Datapath

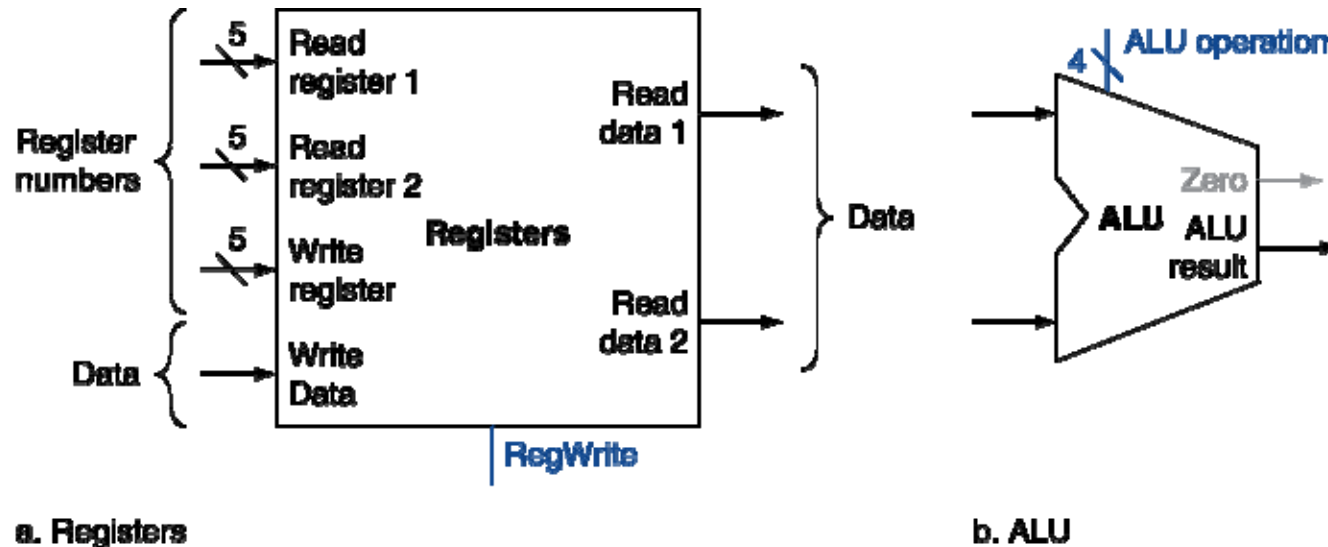
- **Datapath**
 - Elements that process data and addresses in the CPU
 - Registers, ALUs, mux's, memories, ...
- We will build a MIPS datapath incrementally
 - Refining the overview design

Instruction Fetch



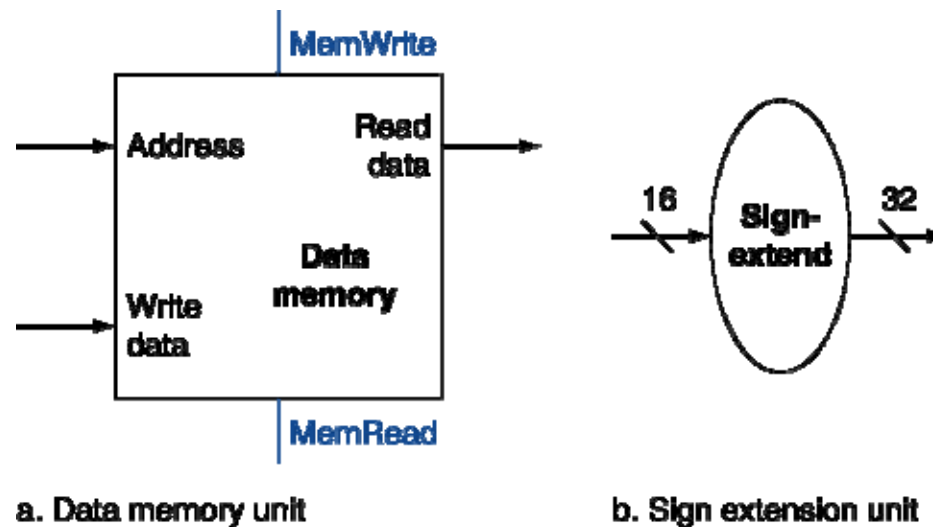
R-Format Instructions

- Read two register operands
- Perform arithmetic/logical operation
- Write register result



Load/Store Instructions

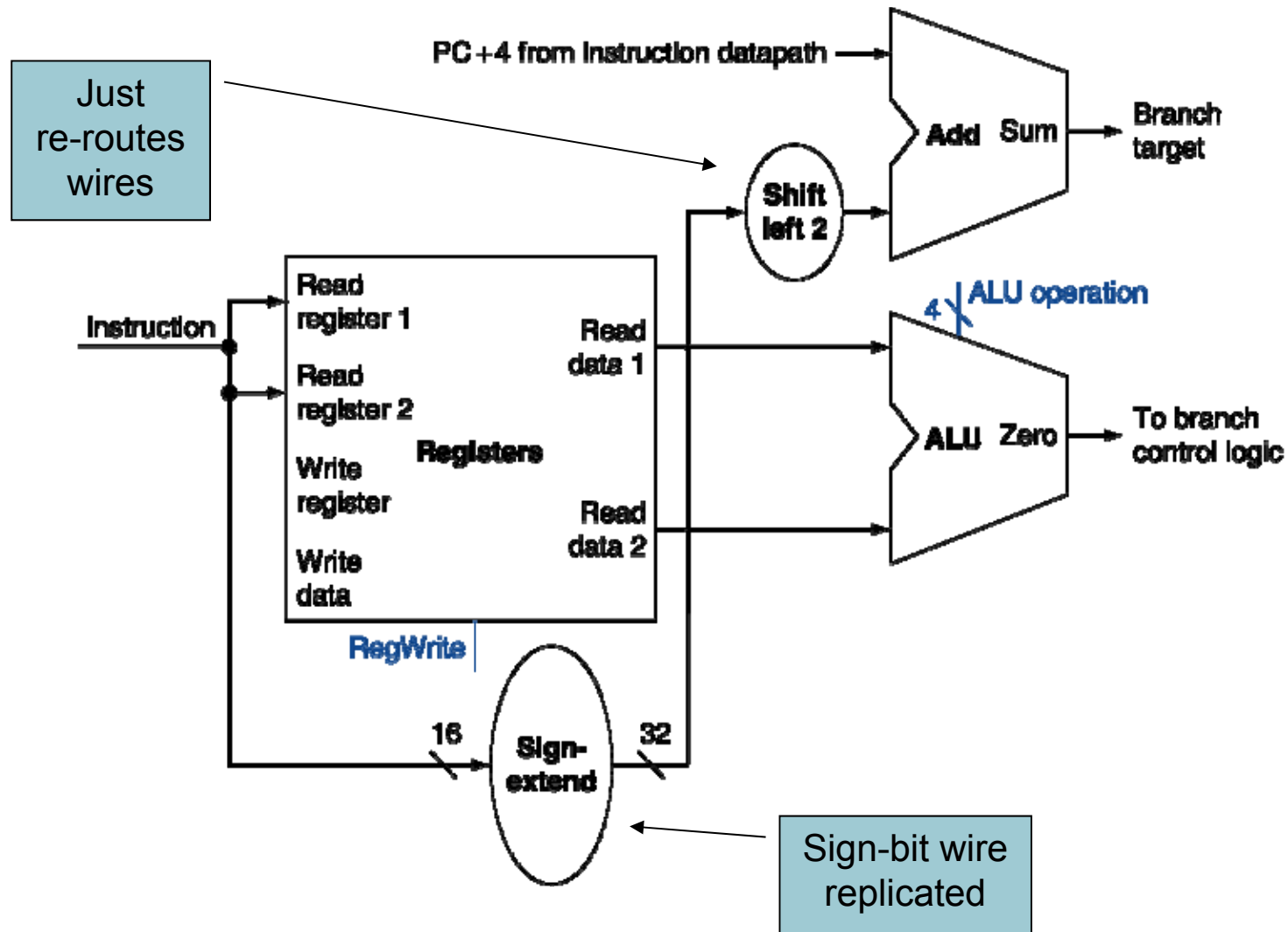
- Read register operands
- Calculate address using 16-bit offset
 - Use ALU, but sign-extend offset
- **Load**: Read memory and update register
- **Store**: Write register value to memory



Branch Instructions

- Read register operands
- Compare operands
 - Use ALU, subtract and check Zero output
- Calculate target address
 - Sign-extend displacement
 - Shift left 2 places (word displacement)
 - Add to PC + 4
 - Already calculated by instruction fetch

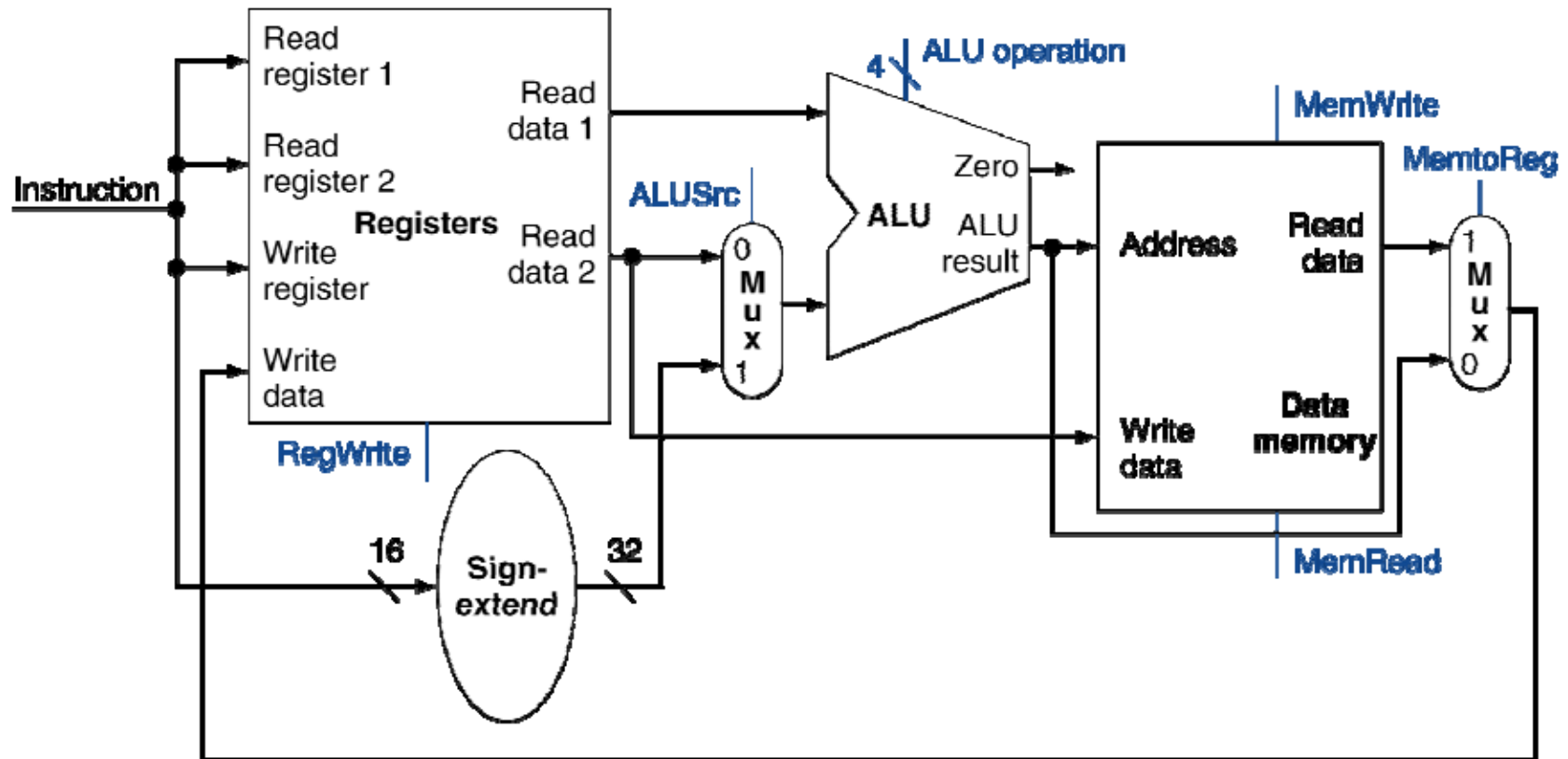
Branch Instructions



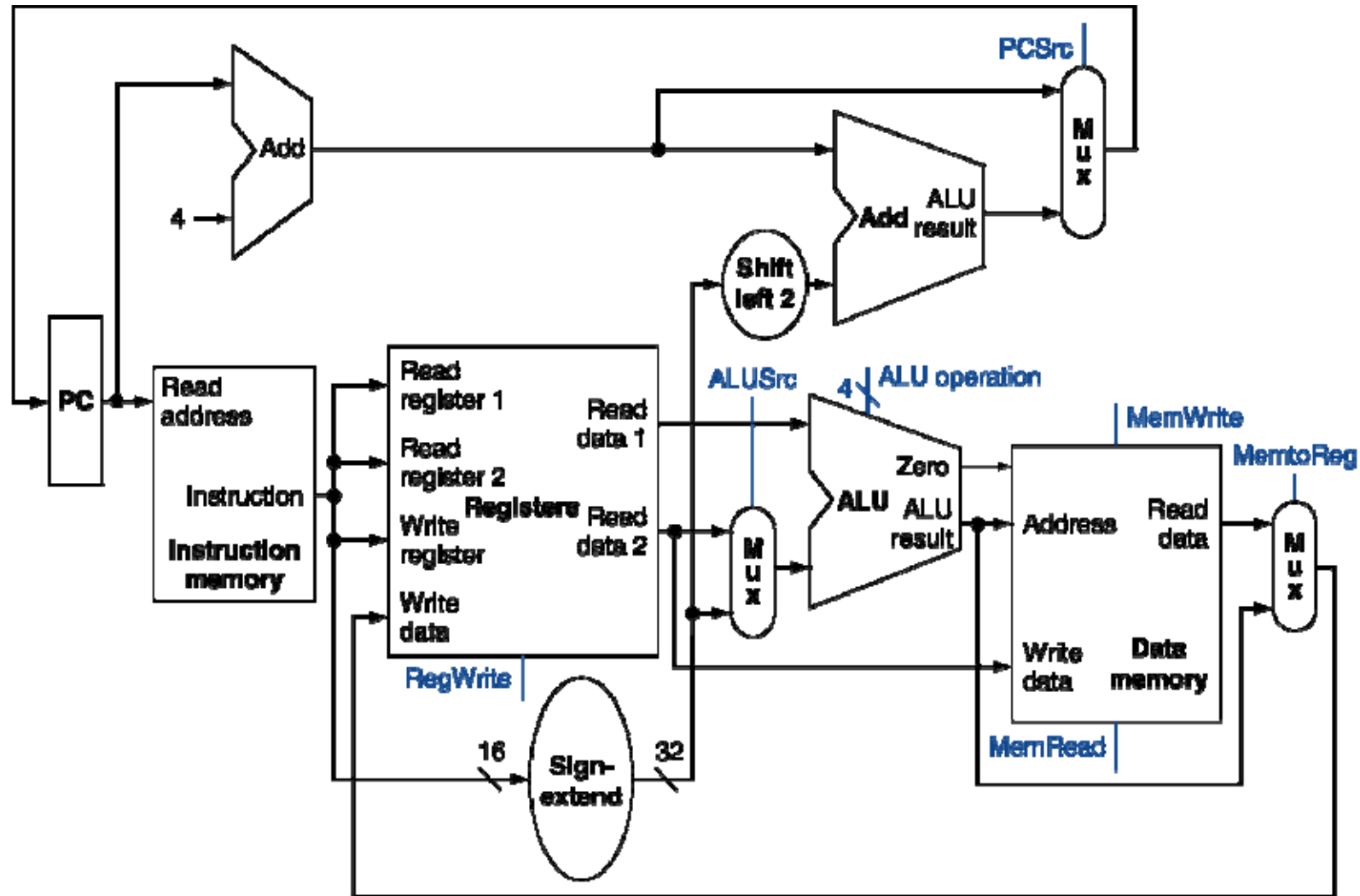
Composing the Elements

- Now that we have examined the datapath components needed for the individual instruction classes, we can combine them into a single datapath and add the control to complete the implementation. This simplest datapath will attempt to execute all instructions in one clock cycle
 - No datapath resource can be used more than once per instruction
 - Any element needed more than once must be duplicated
 - Hence, we need separate instruction and data memories
- Use multiplexers where alternate data sources are used for different instructions

R-Type/Load/Store Datapath



Full Datapath



ALU Control

- ALU used for
 - **Load/Store**: $F = ?$
 - **Branch**: $F = ?$
 - **R-type**: F depends on funct field


| ALU control | Function |
|-------------|------------------|
| 0000 | AND |
| 0001 | OR |
| 0010 | add |
| 0110 | subtract |
| 0111 | set-on-less-than |
| 1100 | NOR |

ALU Control

- Assume 2-bit ALUOp derived from opcode
 - Combinational logic derives ALU control

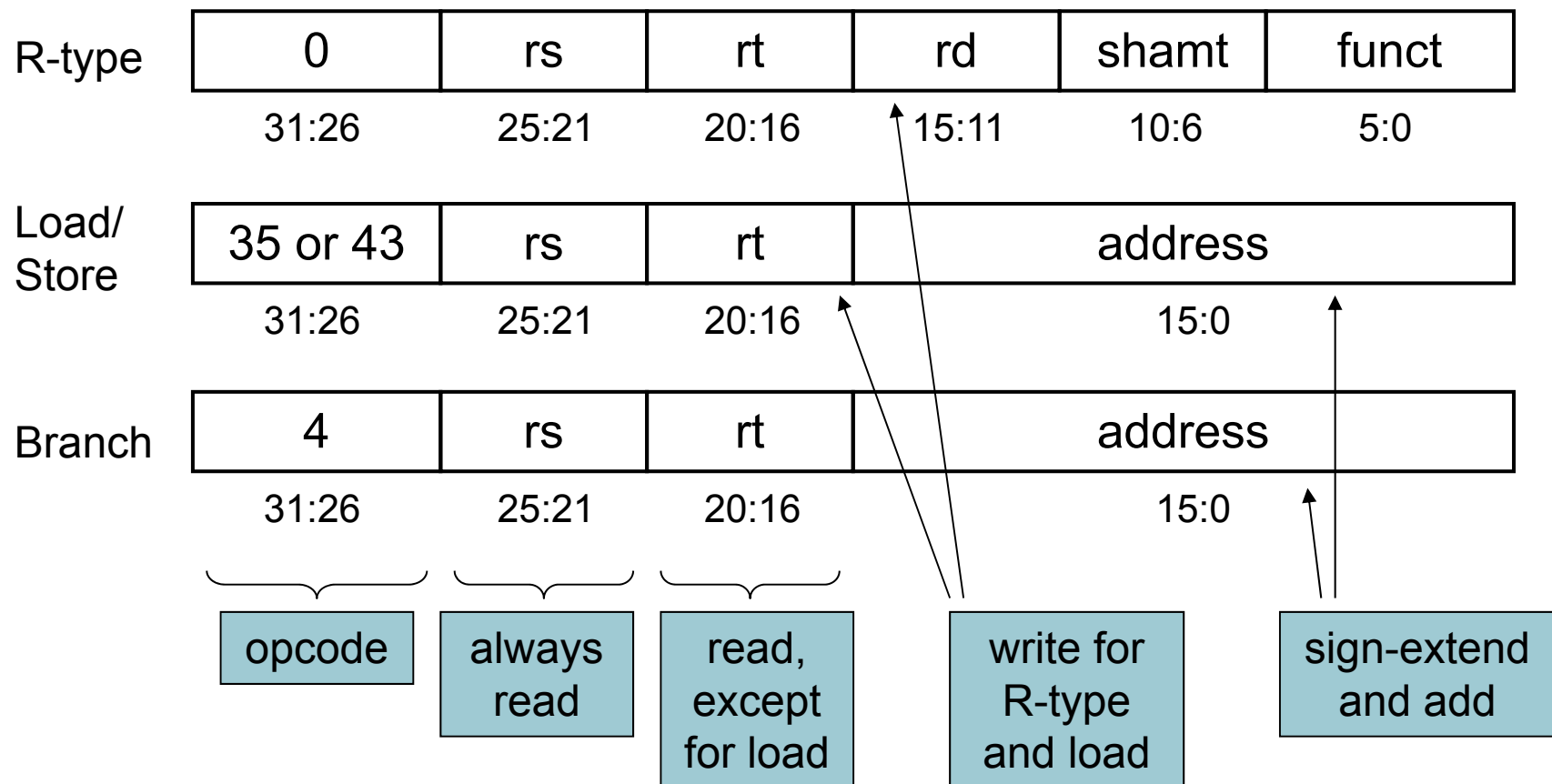
| opcode | ALUOp | Operation | funct | ALU function | ALU control |
|--------|-------|------------------|--------|------------------|-------------|
| lw | 00 | load word | XXXXXX | add | 0010 |
| sw | 00 | store word | XXXXXX | add | 0010 |
| beq | 01 | branch equal | XXXXXX | subtract | 0110 |
| R-type | 10 | add | 100000 | add | 0010 |
| | | subtract | 100010 | subtract | 0110 |
| | | AND | 100100 | AND | 0000 |
| | | OR | 100101 | OR | 0001 |
| | | set-on-less-than | 101010 | set-on-less-than | 0111 |

Generated
output



The Main Control Unit

- Control signals derived from instruction



The effect of each of the seven control signals

| Signal name | Effect when deasserted | Effect when asserted |
|-------------|--|---|
| RegDst | The register destination number for the Write register comes from the rt field (bits 20:16). | The register destination number for the Write register comes from the rd field (bits 15:11). |
| RegWrite | None. | The register on the Write register input is written with the value on the Write data input. |
| ALUSrc | The second ALU operand comes from the second register file output (Read data 2). | The second ALU operand is the sign-extended, lower 16 bits of the instruction. |
| PCSrc | The PC is replaced by the output of the adder that computes the value of PC + 4. | The PC is replaced by the output of the adder that computes the branch target. |
| MemRead | None. | Data memory contents designated by the address input are put on the Read data output. |
| MemWrite | None. | Data memory contents designated by the address input are replaced by the value on the Write data input. |
| MemtoReg | The value fed to the register Write data input comes from the ALU. | The value fed to the register Write data input comes from the data memory. |

The setting of the control lines

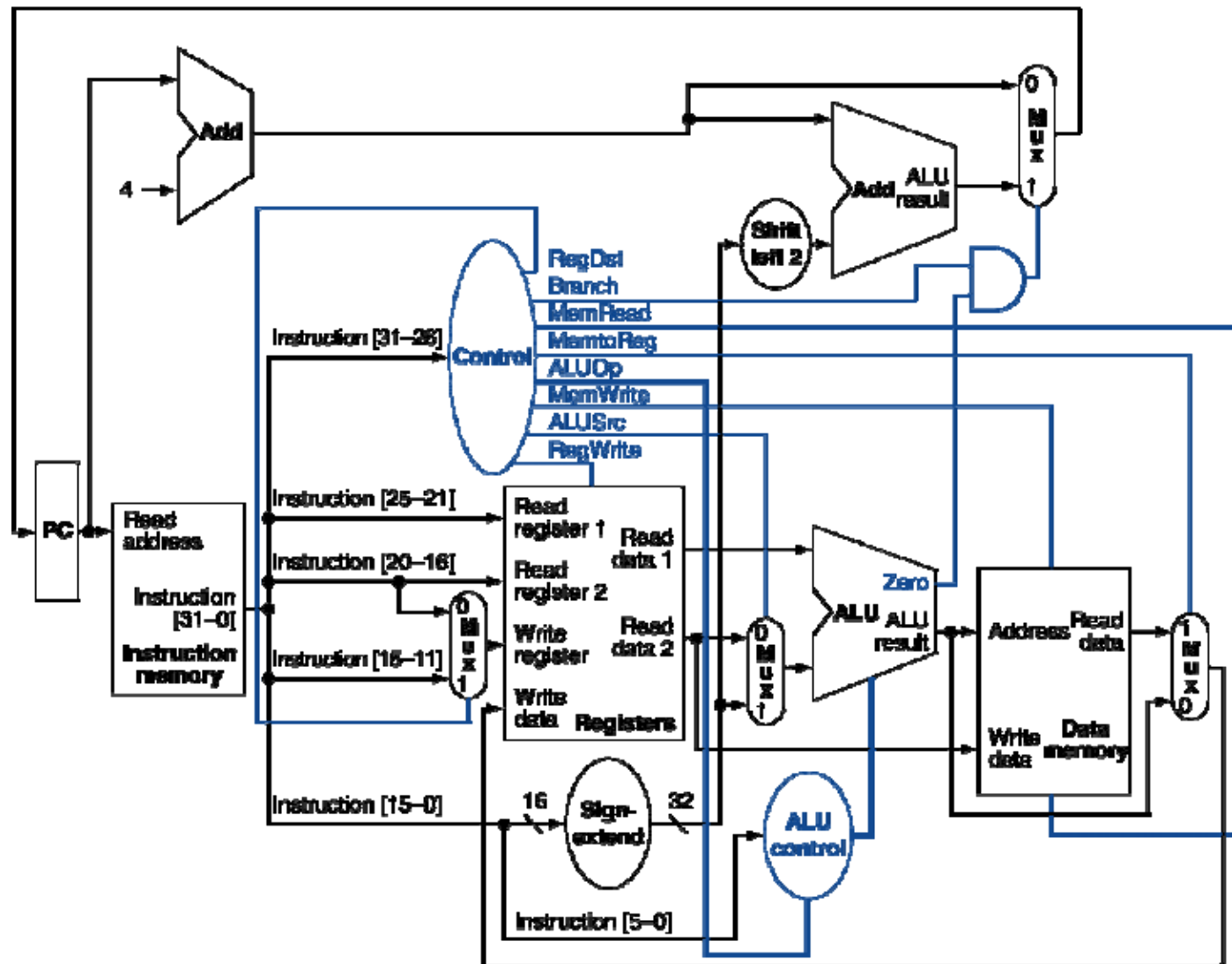
| Instruction | RegDst | ALUSrc | Memto-Reg | Reg-Write | Mem-Read | Mem-Write | Branch | ALUOp1 | ALUOp0 |
|-------------|--------|--------|-----------|-----------|----------|-----------|--------|--------|--------|
| R-format | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| sw | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| beq | X | 0 | X | 0 | 0 | 0 | 1 | 0 | 1 |

1. The first row of the table corresponds to the R-format instructions. For all these instructions, the source register fields are rs and rt, and the destination register field is rd; this defines how the signals ALUSrc and RegDst are set.
2. Furthermore, an R-type instruction writes a register (RegWrite = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with PC + 4; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high.
3. The ALUOp field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field.
4. The second and third rows of this table give the control signal settings for lw and sw. These ALUSrc and ALUOp fields are set to perform the address calculation. The MemRead and MemWrite are set to perform the memory access. Finally, RegDst and RegWrite are set for a load to cause the result to be stored into the rt register.

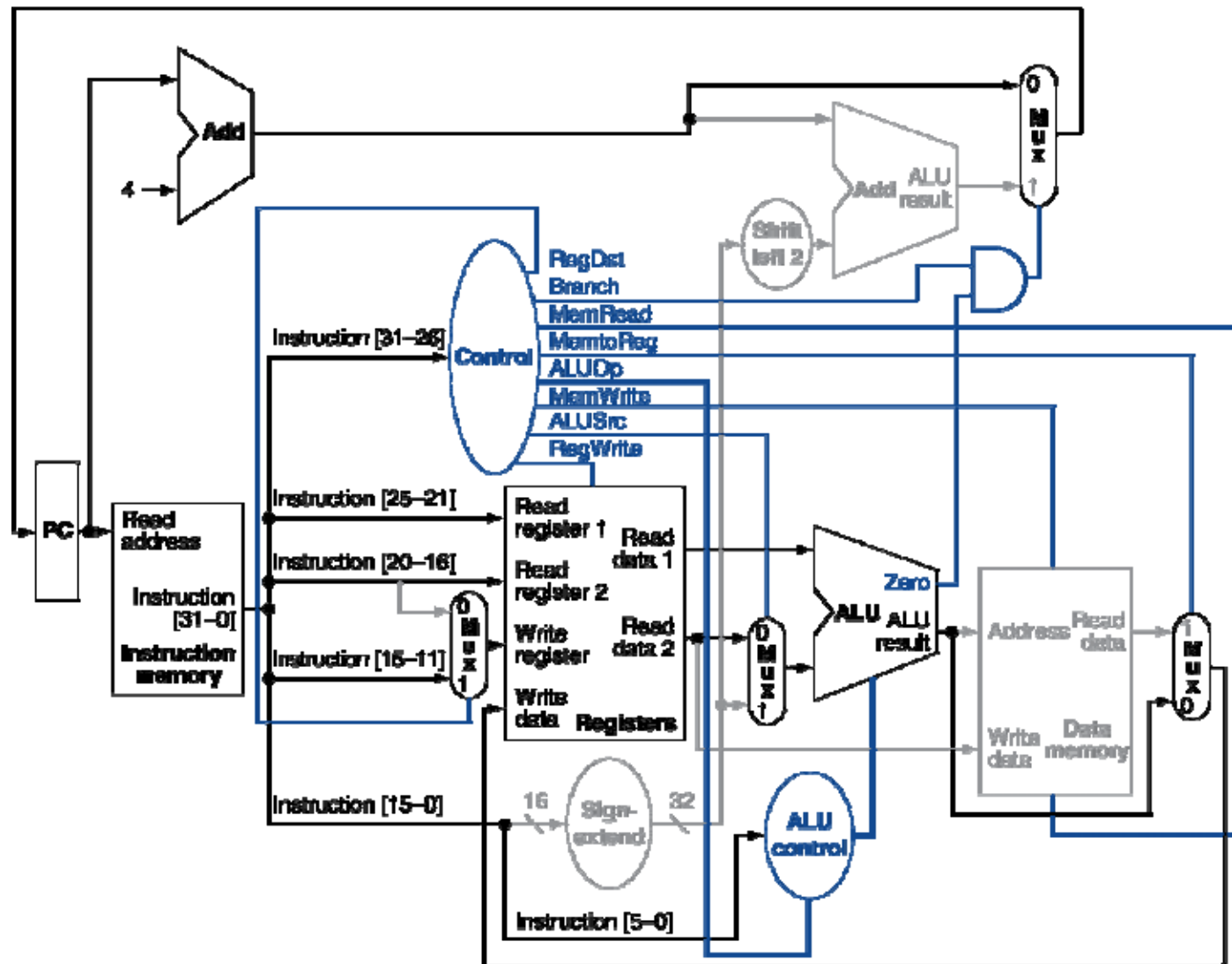
| Instruction | RegDst | ALUSrc | Memto-Reg | Reg-Write | Mem-Read | Mem-Write | Branch | ALUOp1 | ALUOp0 |
|-------------|--------|--------|-----------|-----------|----------|-----------|--------|--------|--------|
| R-format | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| lw | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| sw | X | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 |
| beq | X | 0 | X | 0 | 0 | 0 | 1 | 0 | 1 |

5. The branch instruction is similar to an R-format operation, since it sends the rs and rt registers to the ALU. The ALUOp field for branch is set for a subtract (ALU control = 01), which is used to test for equality.
6. Notice that the MemtoReg field is irrelevant when the RegWrite signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry MemtoReg in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to RegDst when RegWrite is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

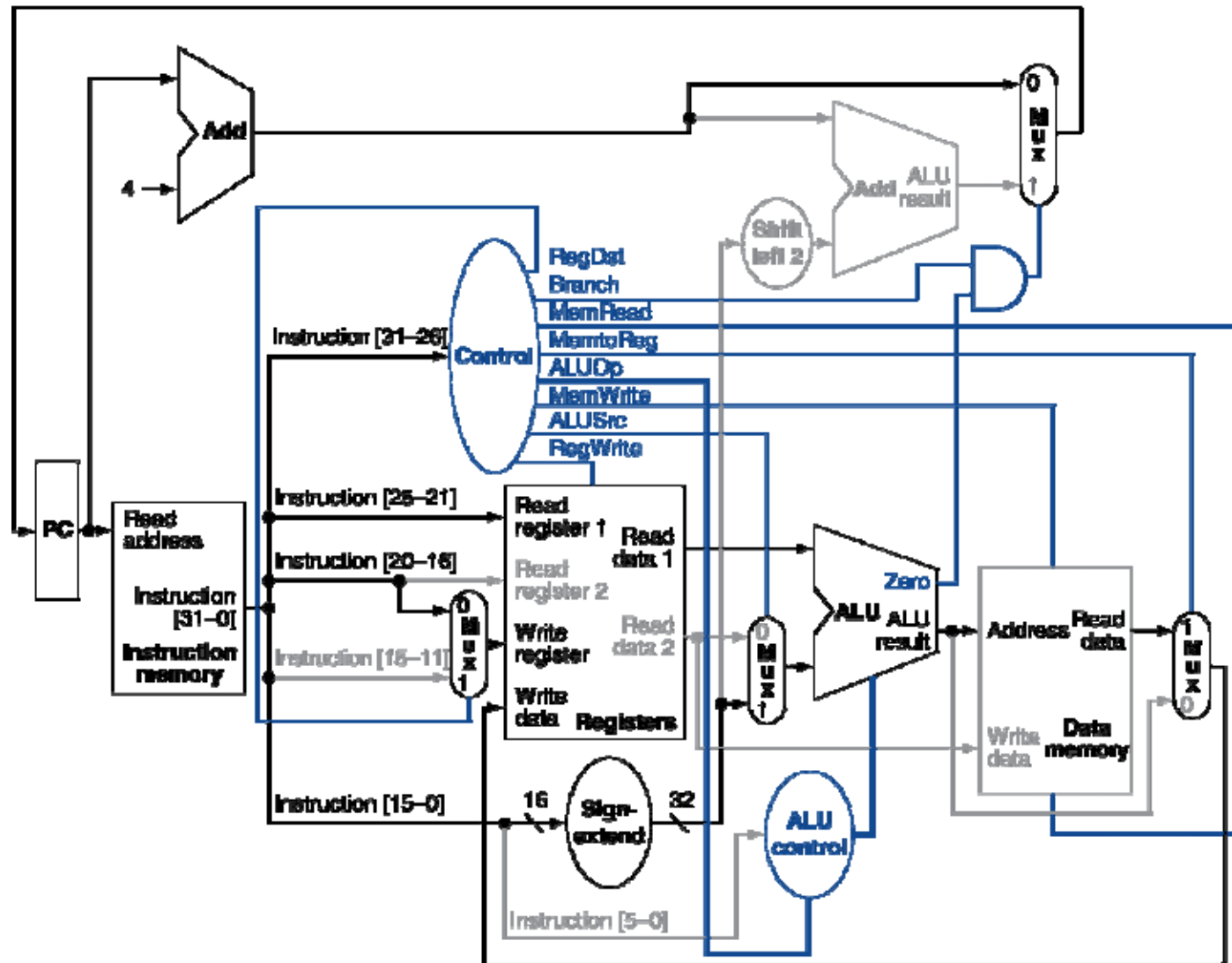
Datapath With Control



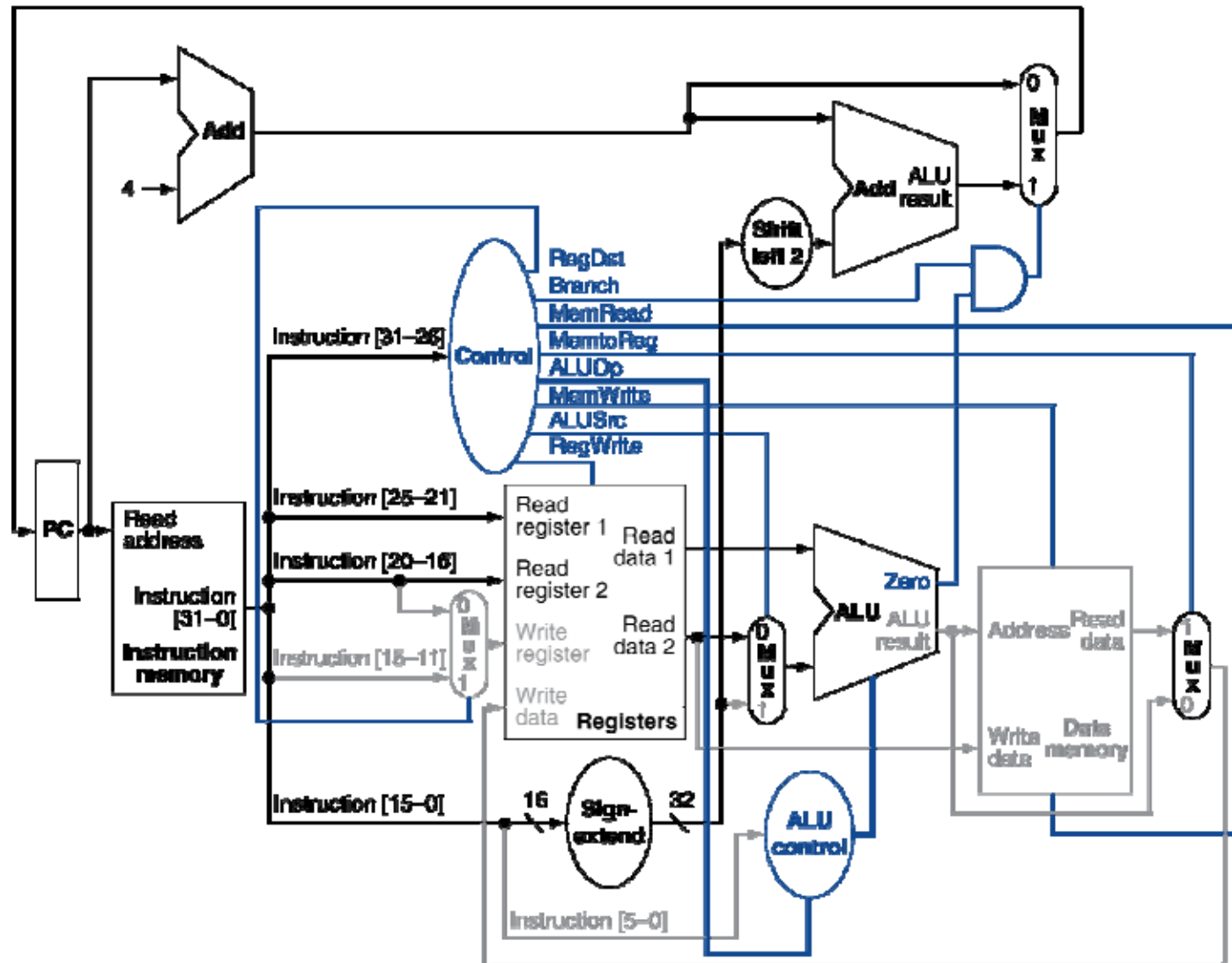
R-Type Instruction



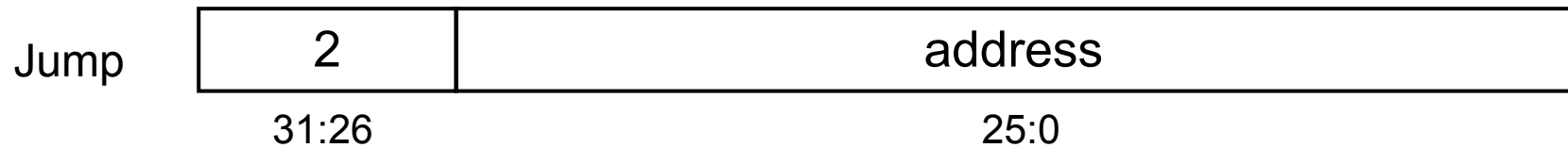
Load Instruction



Branch-on-Equal Instruction

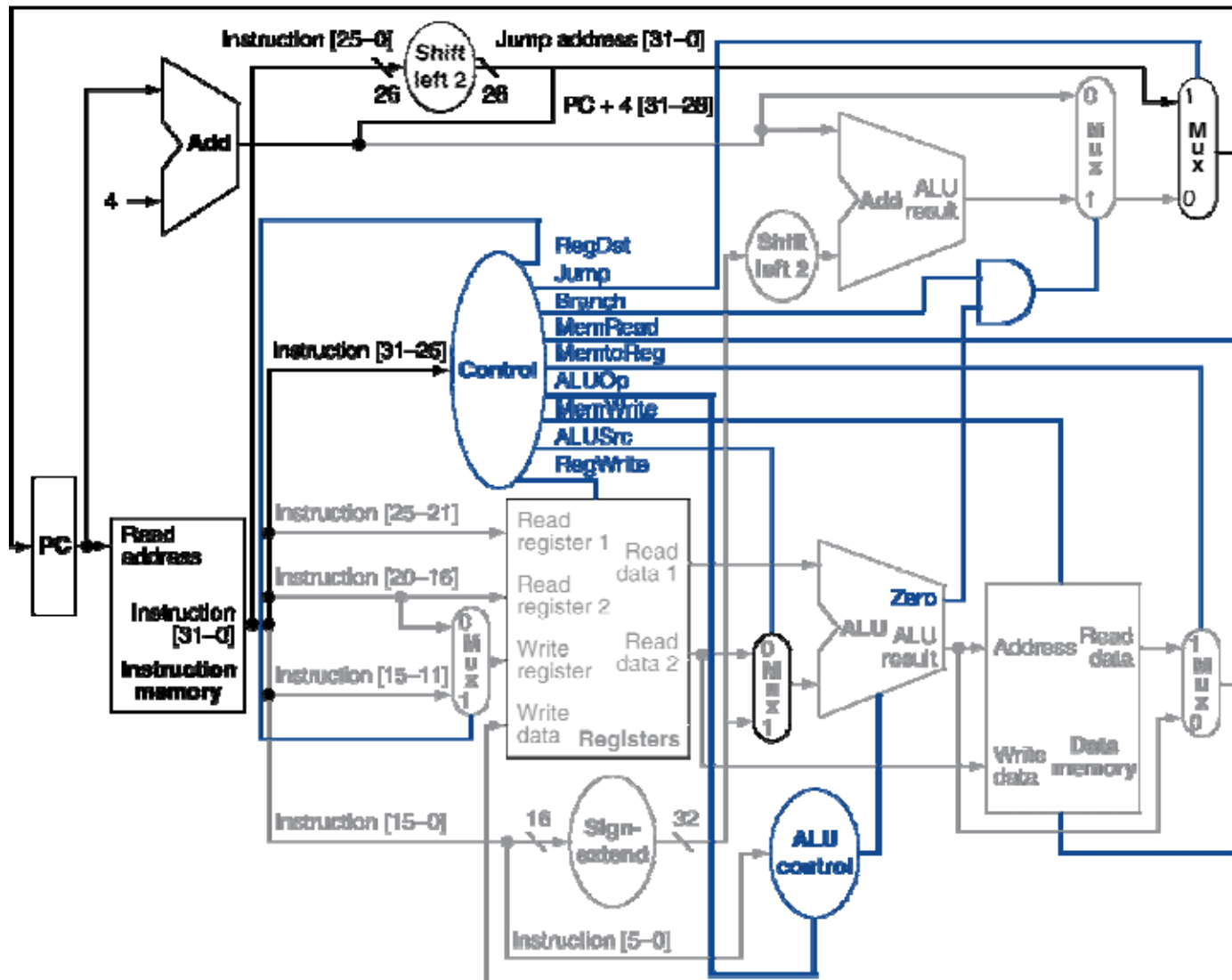


Implementing Jumps



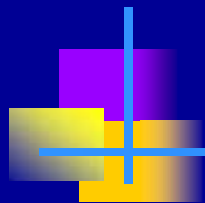
- Jump uses word address
- Update PC with concatenation of
 - Top 4 bits of old PC
 - 26-bit jump address
 - 00
- Need an extra control signal decoded from opcode

Datapath With Jumps Added



Performance Issues

- Longest delay determines clock period
 - Critical path: **load** instruction
 - Instruction memory → register file → ALU → data memory → register file
- Useless to try implementation techniques to reduce delay for different instructions
 - Load instruction still the bottleneck and will still determine clock period
- Violates design principle
 - Making the common case fast!
- We will improve performance by **pipelining**



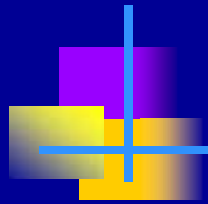
单周期 CPU数据通路的设计

(Verilog+Vivado+Nexys 4版)



主要内容

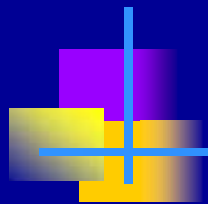
- CPU功能、结构与原理
- 单周期数据通路设计



CPU功能、结构与原理

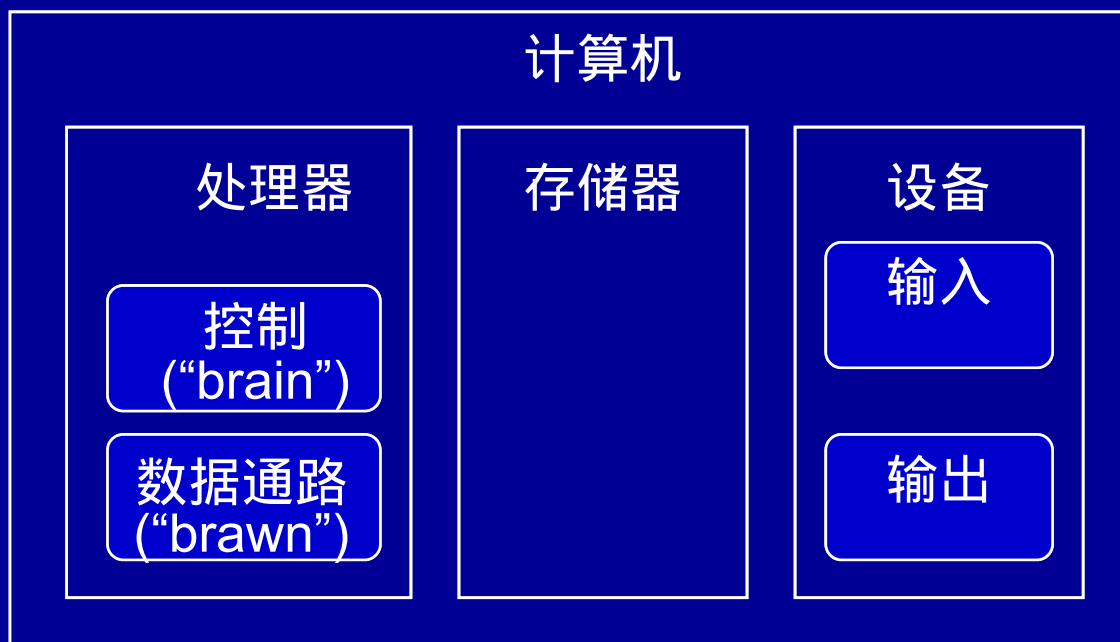
■ CPU的基本功能

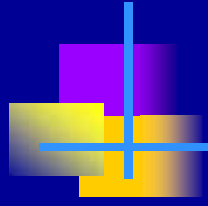
- 指令控制——控制指令执行顺序
- 操作控制——根据指令的功能，产生相应的控制信号
- 时序控制——对各种操作进行时间上的控制
- 数据加工——对数据进行算术/逻辑运算或其它处理
- 端口访问——对来自存储器或I/O端口的数据进行访问
- 中断处理——处理运行过程中的异常情况



CPU功能、结构与原理

- 计算机的基本结构
 - 处理器
 - 控制器
 - 数据通路
 - 存储器
 - 输入
 - 输出



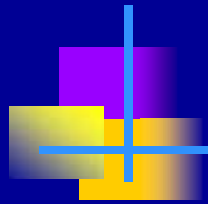


CPU功能、结构与原理

- CPU的基本结构

- **处理器 (CPU):** 实现指令集架构中的指令

- **数据通路:** 处理器的一部分，包含了完成处理器所要求的操作所必须的硬件 (“the brawn”)
 - **控制:** 处理器的一部分（也在硬件中），用以告诉数据通路需要做什么 (“the brain”)



CPU功能、结构与原理

■ CPU执行指令的过程

■ 取指

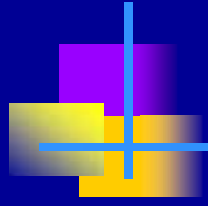
- 根据程序计数器 (PC) 从内存中取指令, PC的值为该指令在内存中存放的地址;

■ 计算PC的值

- 能自动计算PC的值以确定下一条指令的地址;

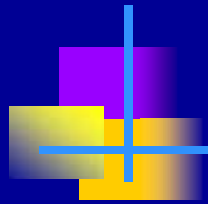
■ 译码

- 对指令操作码进行译码, 以产生控制信号来控制指令进行相应的操作;



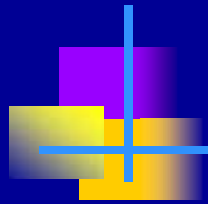
CPU功能、结构与原理

- CPU执行指令的过程
 - 取操作数
 - 根据指令字段的内容，选择从存储器读取数据或直接从寄存器取数。
 - 算术/逻辑运算
 - 根据译码结果，进行算术/逻辑运算或者计算操作数的地址；
 - 结果写回
 - 根据指令的要求对运算（处理）后的数据进行写回操作，如写存储器或写寄存器；
 - 时钟控制



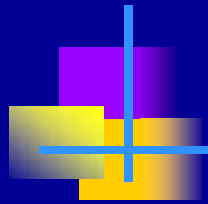
CPU功能、结构与原理

- 组成指令功能的四种基本操作
 - 读取某一主存单元的内容，并将其装入某个寄存器；
 - 把一个数据从某个寄存器存入给定的主存单元或输出到给定的端口地址中
 - 把一个数据从某个寄存器送到另一个寄存器或者ALU；
 - 进行某种算术运算或逻辑运算，将结果送入某个寄存器。



CPU功能、结构与原理

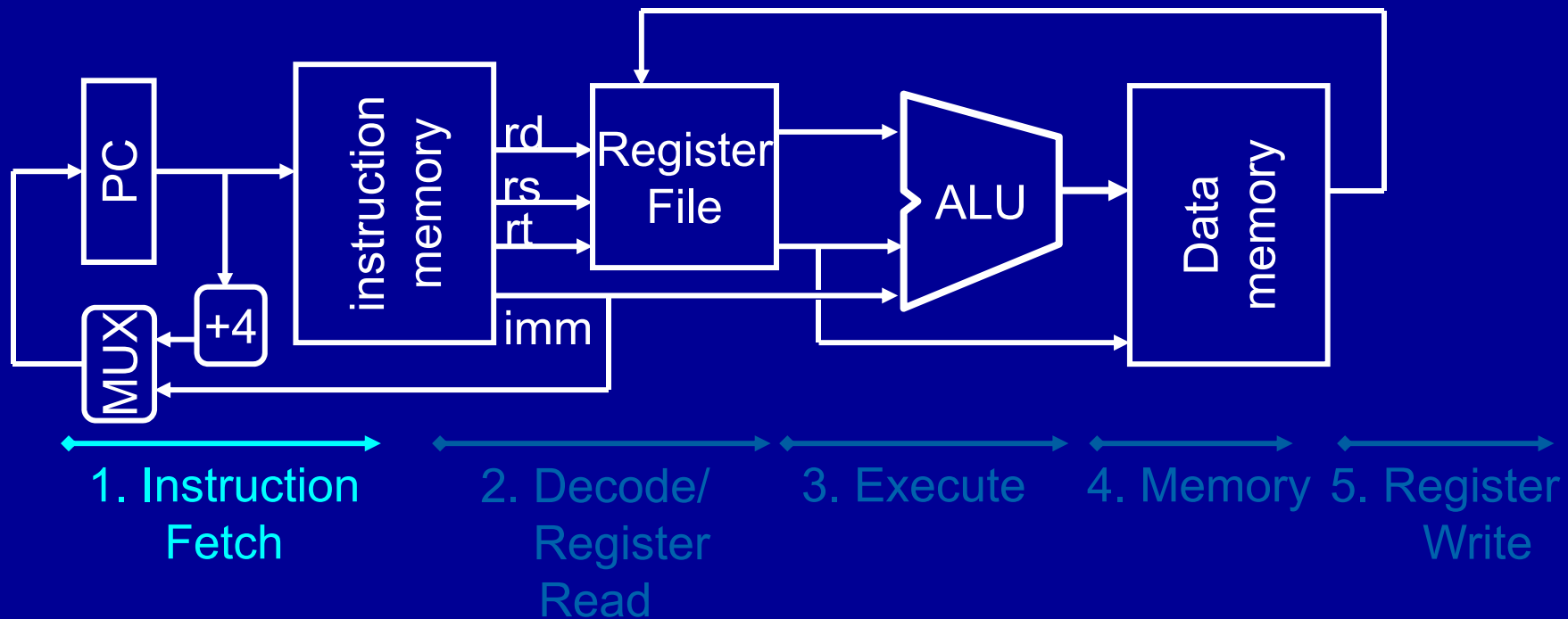
- 组成指令功能的四种基本操作
 - 读取某一主存单元的内容，并将其装入某个寄存器；
 - 把一个数据从某个寄存器存入给定的主存单元或输出到给定的端口地址中
 - 把一个数据从某个寄存器送到另一个寄存器或者ALU；
 - 进行某种算术运算或逻辑运算，将结果送入某个寄存器。



CPU功能、结构与原理

- 操作功能的形式化描述
 - 描述语言称为寄存器传送语言RTL (Register Transfer Language)，本实验所用的RTL规定如下
 - 用 $R[r]$ 表示寄存器 r 的内容；
 - 用 $M[addr]$ 表示读取主存单元 $addr$ 的内容；
 - 传送方向用“ \leftarrow ”表示，传送源在右，传送目的在左；
 - 程序计数器PC直接用PC表示其内容；
 - 用 $OP[data]$ 表示对数据 $data$ 进行OP操作。

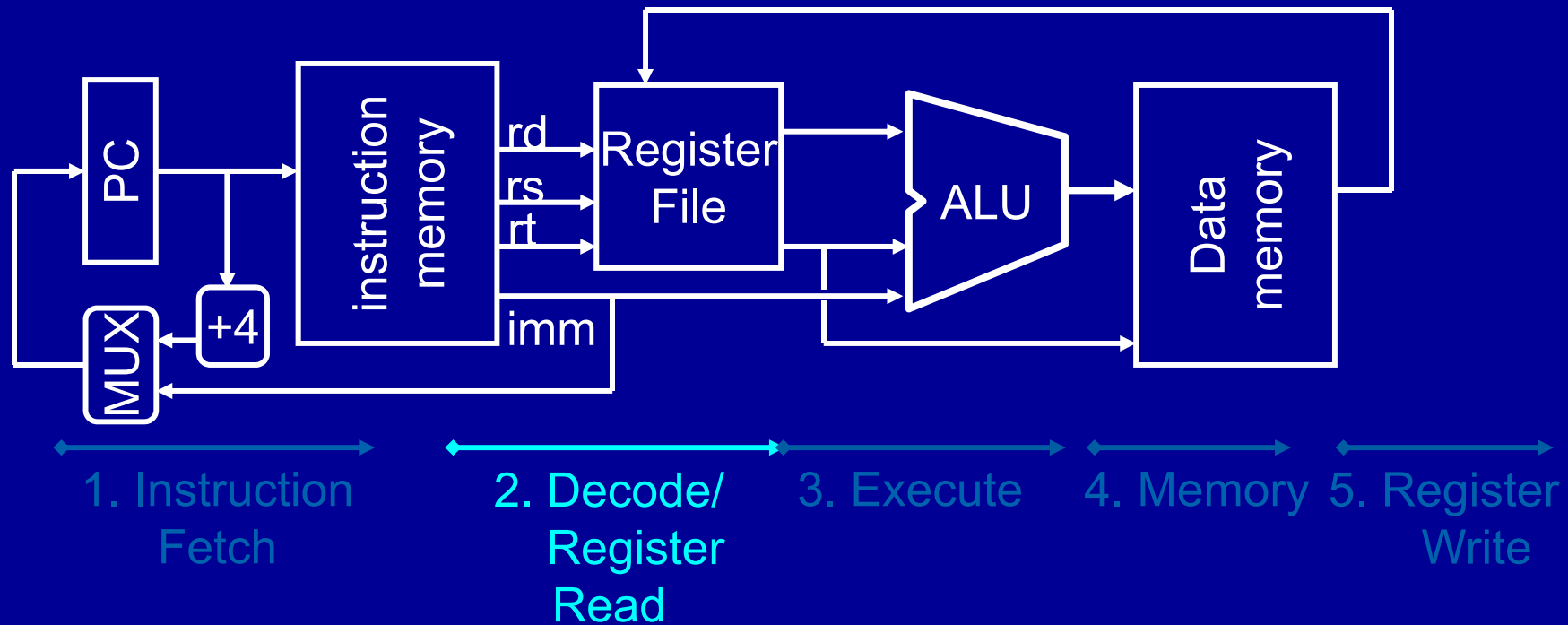
单周期数据通路的设计



Phase 1: *Instruction Fetch* (IF)

- Fetch 32-bit instruction from memory
- Increment PC ($PC = PC + 4$)

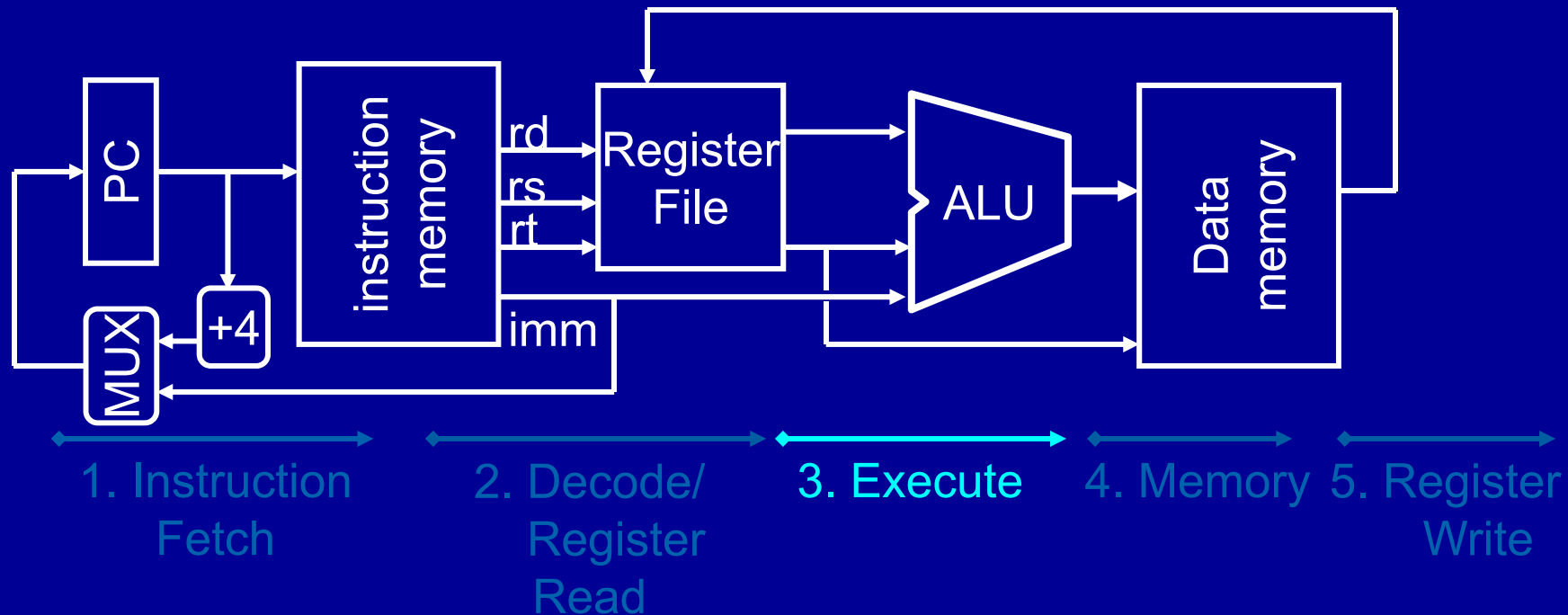
单周期数据通路的设计



Phase 2: *Instruction Decode* (ID)

- Read the opcode and appropriate fields from the instruction
- Gather all necessary registers values from Register File

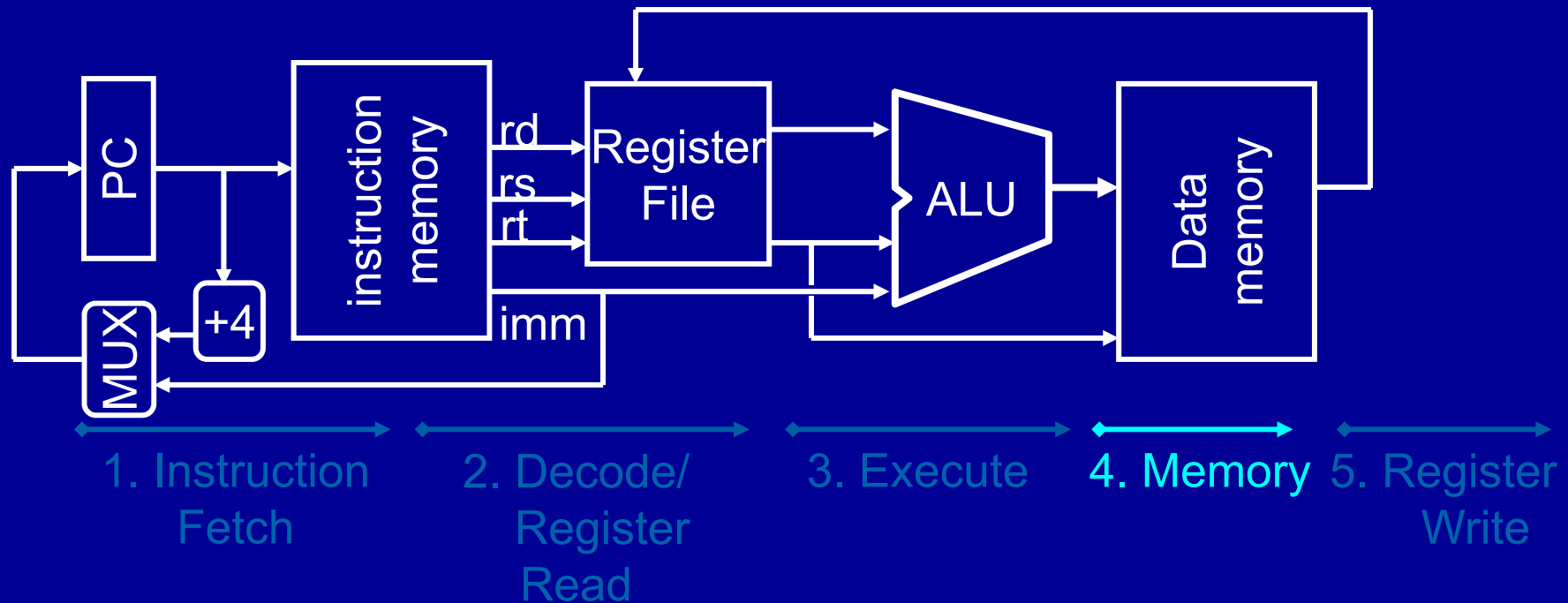
单周期数据通路的设计



Phase 3: *Execute* (EX)

- ALU performs operations: arithmetic (+, -, *, /), shifting, logical (&, |), comparisons (slt, ==)
- Also calculates addresses for loads and stores

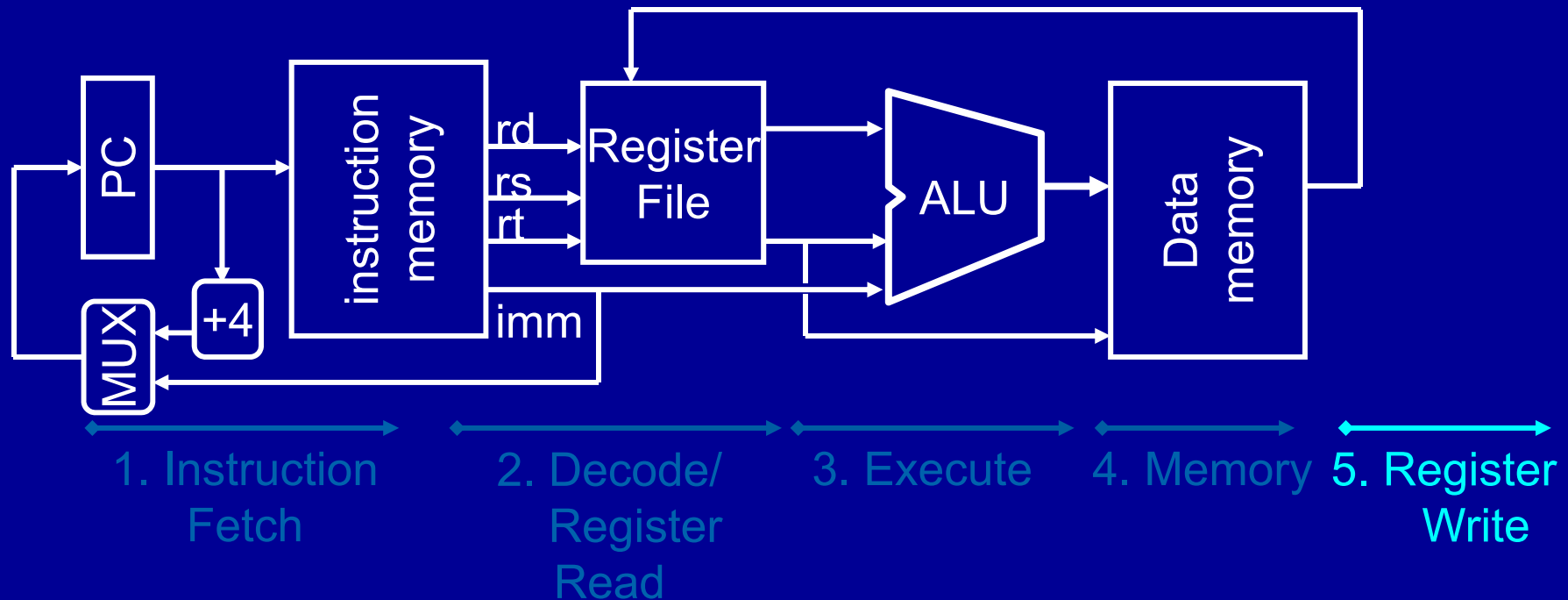
单周期数据通路的设计



Phase 4: *Memory Access* (MEM)

- Only load and store instructions do anything during this phase; the others remain idle or skip this phase
- Should be fast due to caches

单周期数据通路的设计



Phase 5: *Register Write* (WB for “write back”)

- Write the instruction result back into the Register File
- Those that don't (e.g. sw, j, beq) remain idle or skip this phase



单周期数据通路的设计

- 单周期数据通路的设计思想
 - 分析和确定每条指令所需数据通路部件；
 - 在给出数据通路部件的同时，确定它们对应的控制信号。

单周期数据通路的设计

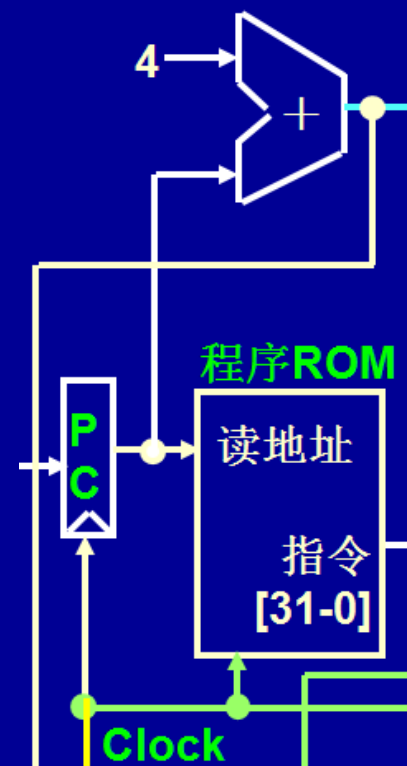
- 单周期数据通路的设计步骤
 - 取指部件(Instruction Fetch Unit)

每条指令都有的公共操作：

- 取指令：M[PC]
- 更新PC： $PC \leftarrow PC + 4$

顺序：先取指令，再改PC的值

数据通路部件：取指部件（PC、指令存储器）

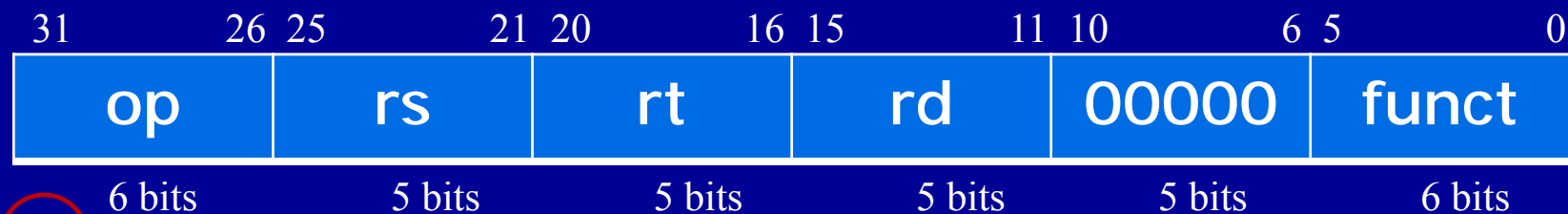


单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ R型指令数据通路

■ 非移位指令和jr



😊 指令特点：

- 1) 指令类型由op和funct决定；
- 2) R[rs] , R[rt]为ALU的源操作数；
- 3) rd为目标寄存器地址，ALU将计算结果写入目的寄存器；

单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ R型指令数据通路

■ 非移位指令和jr

☺ 指令功能的RTL描述：

- $M[PC], PC \leftarrow PC+4$
- $R[rd] \leftarrow R[rs] \text{ op } R[rt]$

☺ 所需器件

① 取指部件；② 寄存器组；③ ALU

☺ 新加控制信号2个（译码后产生）

① ALU_ctr — ALU控制信号，用以完成多种运算（稍后再讲）

② RegWrite — 寄存器堆控制信号，用于运算结果写回

MIPS instruction format

R-format



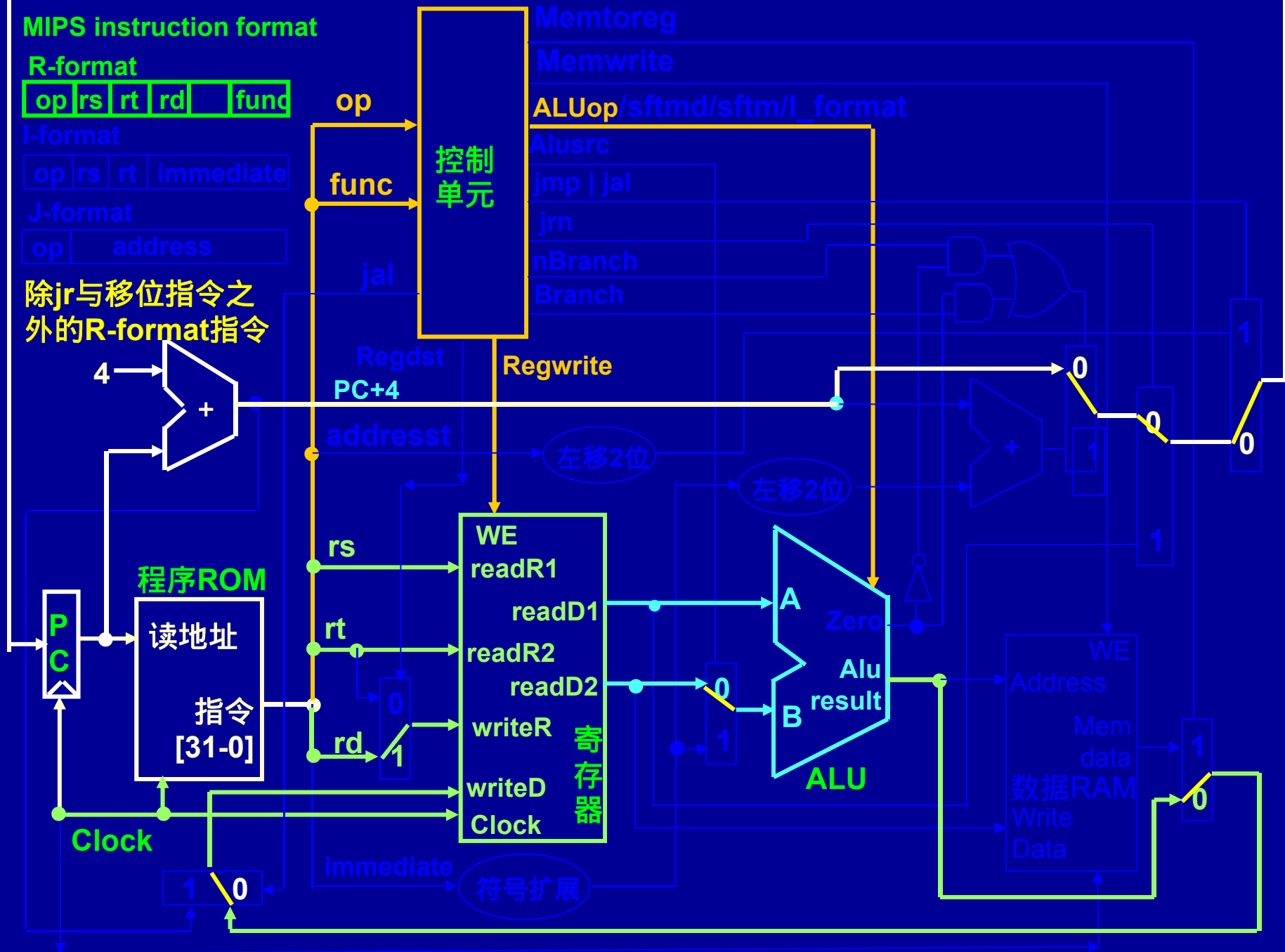
I-format



J-format



除jr与移位指令之外的R-format指令

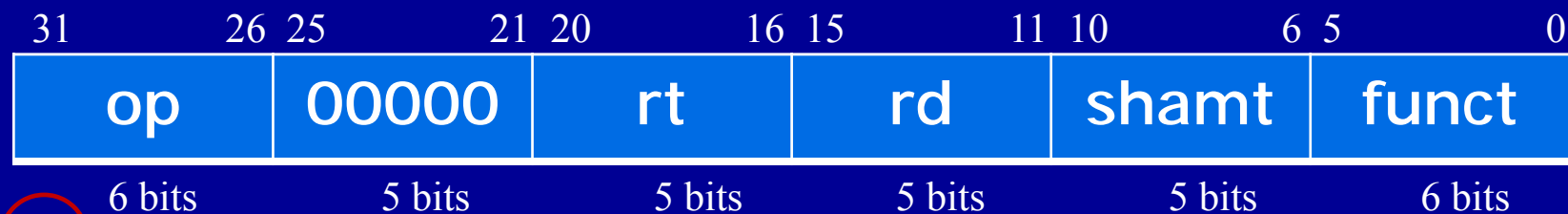


单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ R型指令数据通路

■ 移位指令



😊 指令特点：

- 1) 指令类型由op和funct决定；
- 2) R[rt]，shamt为ALU的源操作数，shamt从取指部件传入
- 3) rd为目标寄存器地址，ALU将计算结果写入目的寄存器；



单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ R型指令数据通路

☺ ■ 移位指令 指令功能的RTL描述：

- $M[PC], PC \leftarrow PC+4$
- $R[rd] \leftarrow R[rt] \text{ op shamt}$

☺ 所需器件

① 取指部件；② 寄存器组；③ ALU

☺ 新加控制信号1个（译码后产生）

① sftmd — ALU控制信号，标明是移位运算

MIPS instruction format

R-format

op 0 rt rd sh func

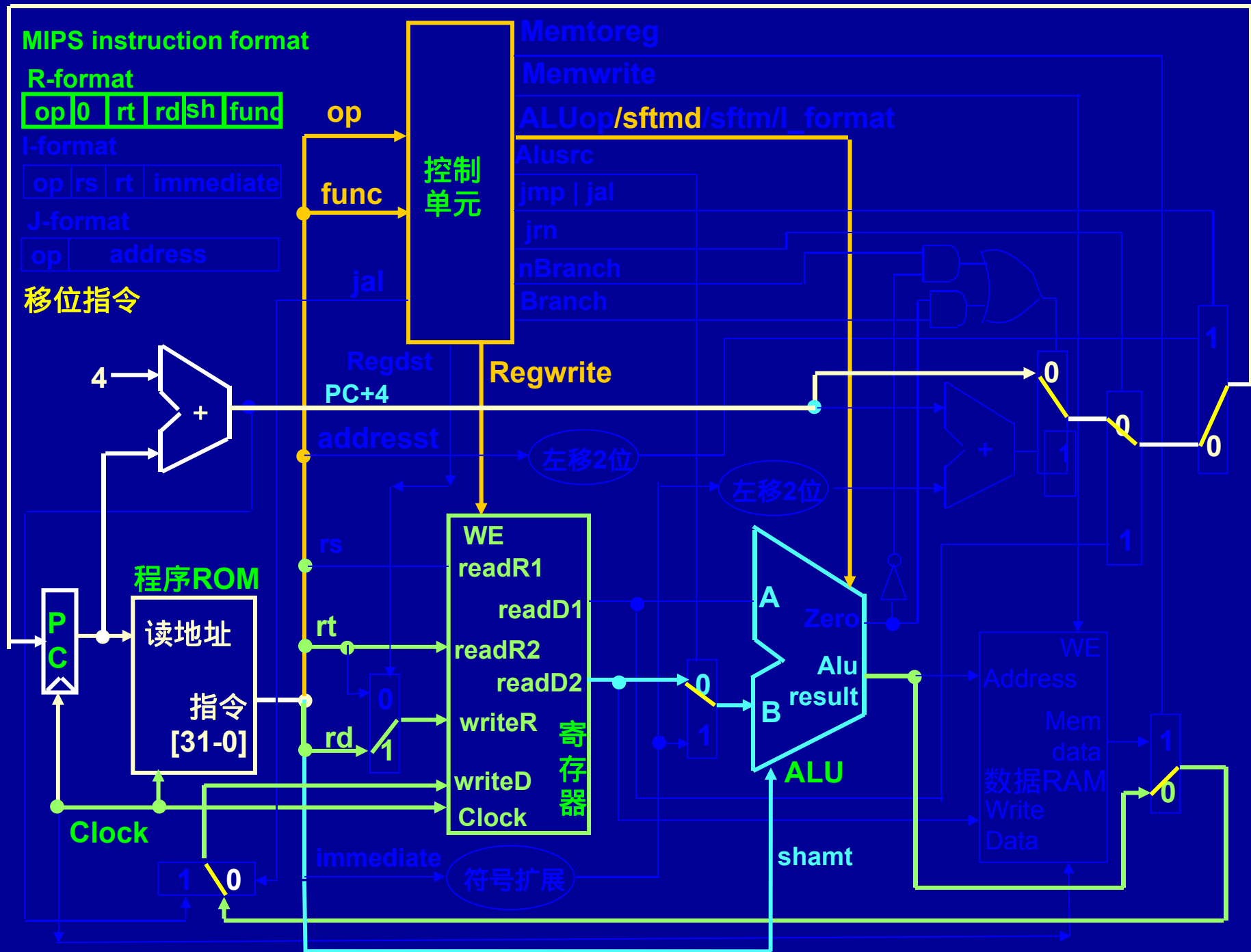
I-format

op rs rt immediate

J-format

op address

移位指令

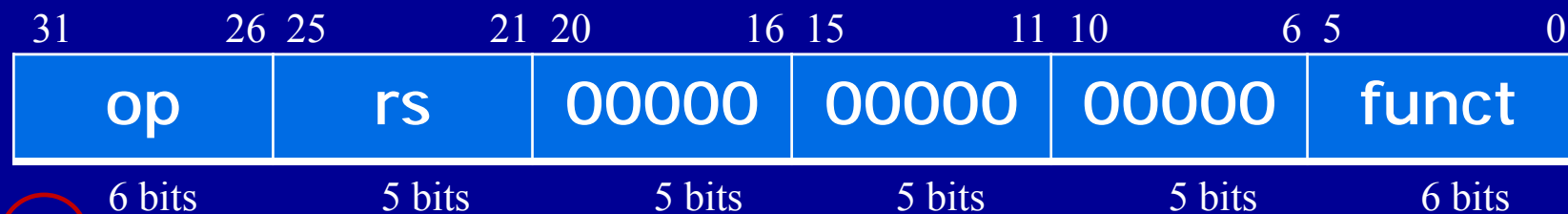


单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ R型指令数据通路

■ jr指令



😊 指令特点：

- 1) 指令类型由op和funct决定；
- 2) R[rs]为源操作数；
- 3) 将rs的值给PC，改变了 $PC \leftarrow PC+4$ ；



单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ R型指令数据通路

■ jr指令

😊 指令功能的RTL描述：

- $M[PC]$
- $PC \leftarrow R[rs]$

😊 增加器件

① 2选1数据选择器 —— 选择PC赋哪一个值，由新的控制信号jrn来控制。

MIPS instruction format

R-format



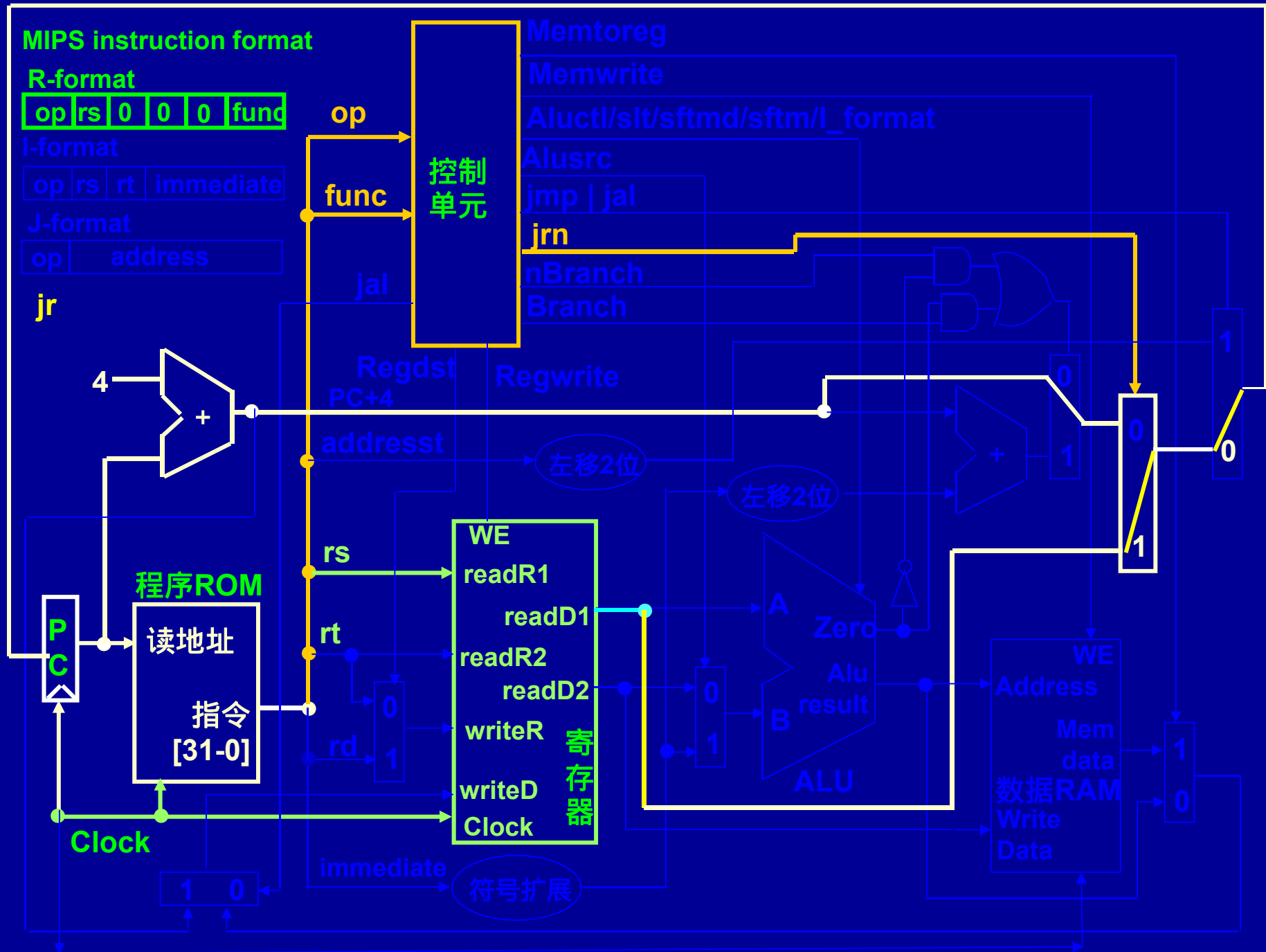
I-format



J-format



jr

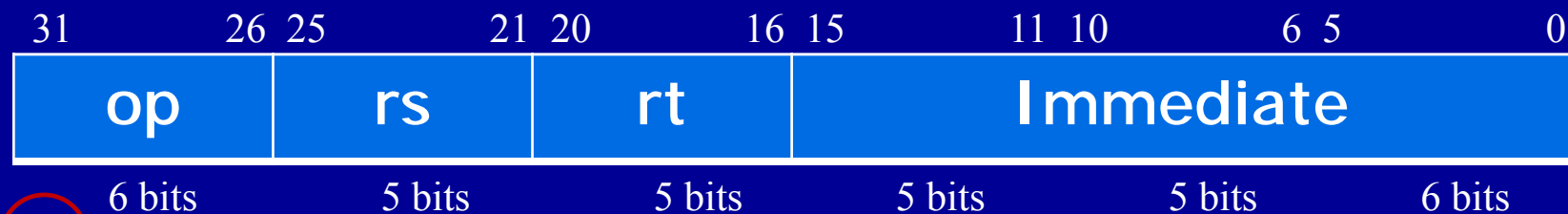


单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ I型指令数据通路

■ **addi, addiu, andi, ori, xori, lui, slti, sltiu**指令



😊 指令特点：

- 1) 指令类型由op决定；
- 2) R[rs] , Immediate为ALU的源操作数 , lui指令没有rs;
- 3) 目标寄存器为rt;

单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ I型指令数据通路

■ **addi, addiu, andi, ori, xori, lui, slti, sltiu**指令

☺ 指令功能的RTL描述：

- $M[PC] ; PC \leftarrow PC + 4 ;$
 - $R[rt] \leftarrow R[rs] \text{ op ZeroExt}(\text{imm16})$
 - $R[rt] \leftarrow R[rs] \text{ op SignExt}(\text{imm16})$
- } 立即数扩展并与rs的内容作运算

思考：

- 1) R型指令与此类指令目标寄存器不一致怎么办？
- 2) 有一个源操作数也和R型指令不一致怎么办？
- 3) 立即数扩展有无符号和有符号之分怎么办？

单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ I型指令数据通路

■ **addi, addiu, andi, ori, xori, lui, slti, sltiu**指令

😊 增加器件

- ① 2选一数据选择器——选择不同的目标寄存器，
- ② 2选一数据选择器——源操作数选择
- ③ 立即数扩展器。

😊 增加控制信号

- ① **I_format** —— 说明进行的是上述几种指令；
- ② **RegDst** —— 为1选择rd是目标寄存器
- ③ **ALUSrc** —— 为1选择立即数作为第二源操作数；

MIPS instruction format

R-format



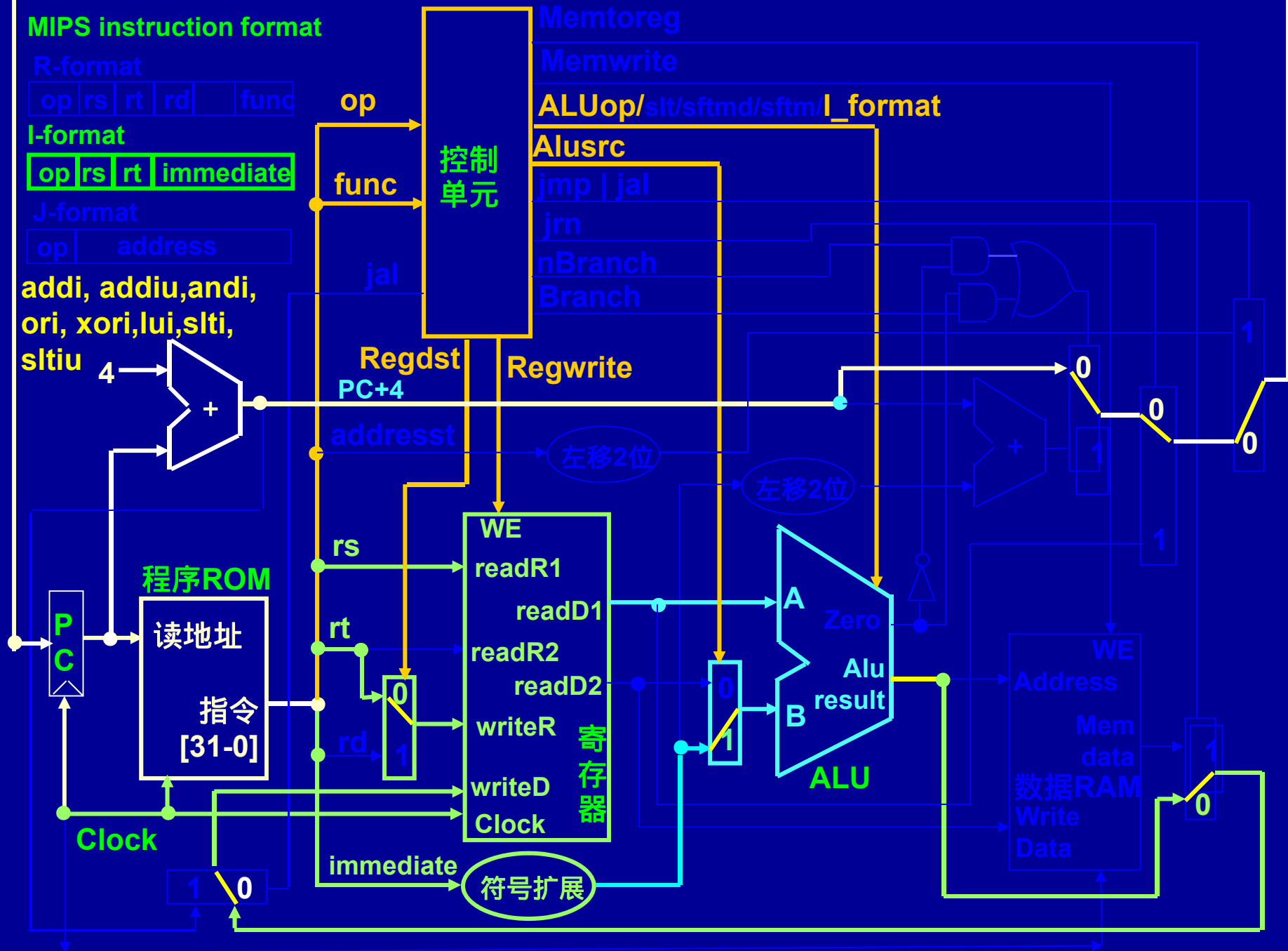
I-format



J-format



addi, addiu, andi,
ori, xori, lui, slti,
sltiu

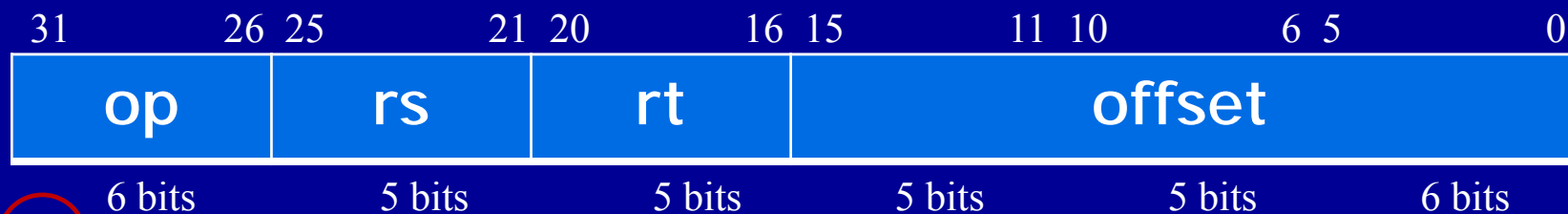


单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ 访存指令数据通路

■ lw指令



😊 指令特点：

- 1) 指令类型仅由op决定；目标寄存器为rt。
- 2) 唯一一条从存储器中读取数据的指令；
- 3) 指令中的16位立即数为偏移量offset，需符号扩展成32位；
- 4) rs和符号扩展后的立即数作为ALU的数据输入，ALU的计算结果为要访问的存储单元的地址。



单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ 访存指令数据通路

■ lw指令

☺ 指令功能的RTL描述：

- $M[PC] ; PC \leftarrow PC + 4$
- $Addr \leftarrow R[rs] + \text{SignExt}(\text{imm16}) ;$ 计算存储单元地址
- $R[rt] \leftarrow M[Addr] ;$ 从存储器中取出的数据送rt



单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ 访存指令数据通路

■ lw指令

😊 增加器件

- ① 数据存储器——所需信号引脚为write_data、read_data、address、Memwrite、CLK；
- ② 2选1数据选择器——运算类指令和lw指令写入的目的寄存器的结果来源不同。选择器的控制信号为MemtoReg。

😊 增加控制信号

- ① MemtoReg——选择写入寄存器的是存储器数据还是运算器数据；

MIPS instruction format

R-format



I-format



J-format



lw

Memtoereg

Memwrite

ALUop/slt/sftmd/sftm/l format

Alusrc

jmp | jal

jrn

nBranch

Branch

控制单元

Regwrite

Regdst
PC+4

addressst

左移2位

左移2位

程序ROM

读地址

指令
[31-0]

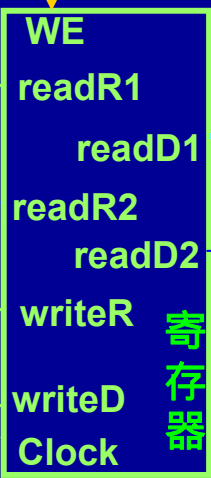
Clock

1

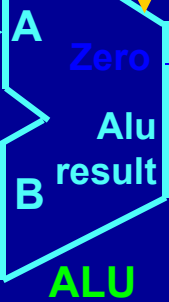
0

immediate

符号扩展



寄存器



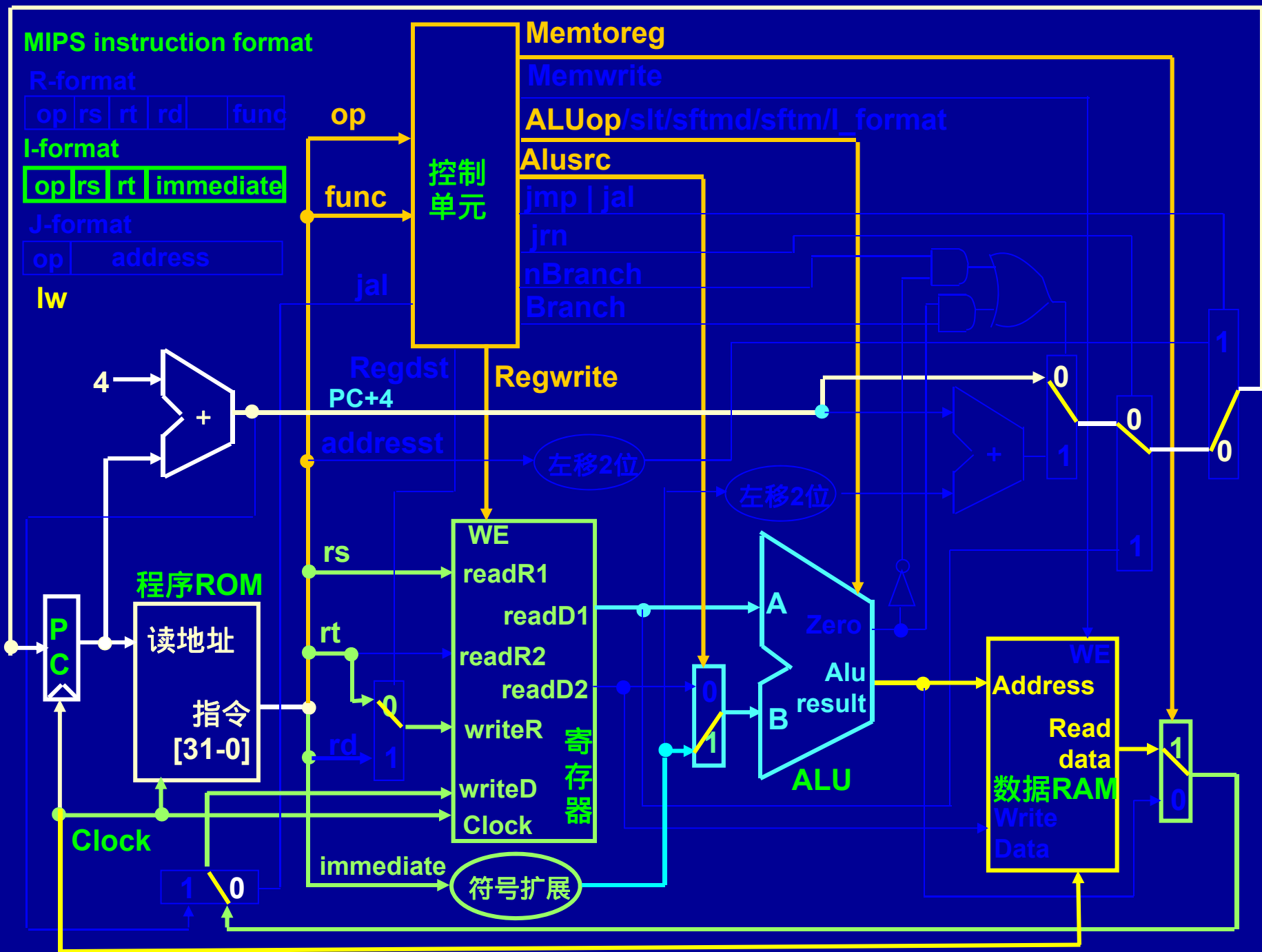
ALU

Address

Read data

数据RAM

Write Data

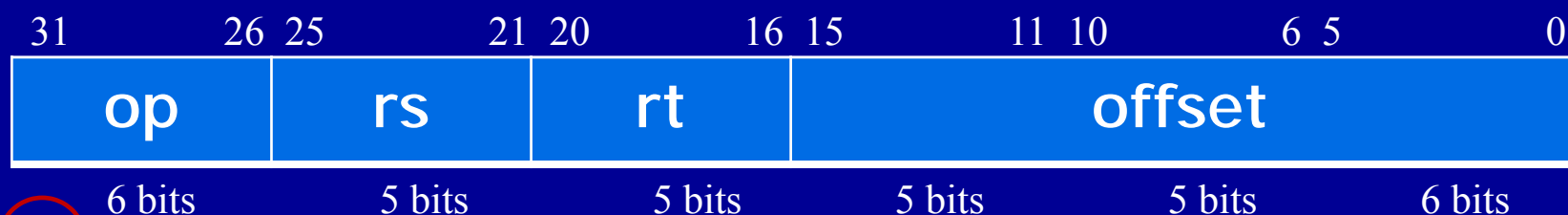


单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ 访存指令数据通路

■ sw指令



指令特点：

- 1) 指令类型仅由op决定。
- 2) 唯一一条向存储器写数据的指令；
- 3) 指令中的16位立即数为偏移量offset，需符号扩展成32位；
- 4) rs和符号扩展后的立即数作为ALU的数据输入，ALU的计算结果为要访问的存储单元的地址。



单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ 访存指令数据通路

■ sw指令

☺ 指令功能的RTL描述：

- $M[PC] ; PC \leftarrow PC+4$
- $Addr \leftarrow R[rs] + \text{SignExt}(\text{imm16}) ;$ 计算存储单元地址
- $M[Addr] \leftarrow R[rt] ;$ 将rt中的数写到存储器中



单周期数据通路的设计

- 单周期数据通路的设计步骤
 - 访存指令数据通路
 - sw指令

😊 增加控制信号

① Memwrite —— 存储器写允许信号；

MIPS instruction format

R-format



I-format



J-format



SW

MemtoReg

Memwrite

ALUop/slt/sltmd/sltm/l format

Alusrc

jmp | jal

jrn

nBranch

Branch

控制单元

Regdst

Regwrite

PC+4

address

左移2位

左移2位

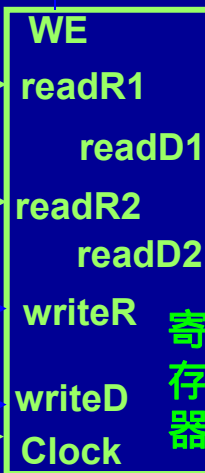
程序ROM

读地址

指令
[31-0]

Clock

immediate



寄存器

符号扩展

Zero

Alu result

B

ALU

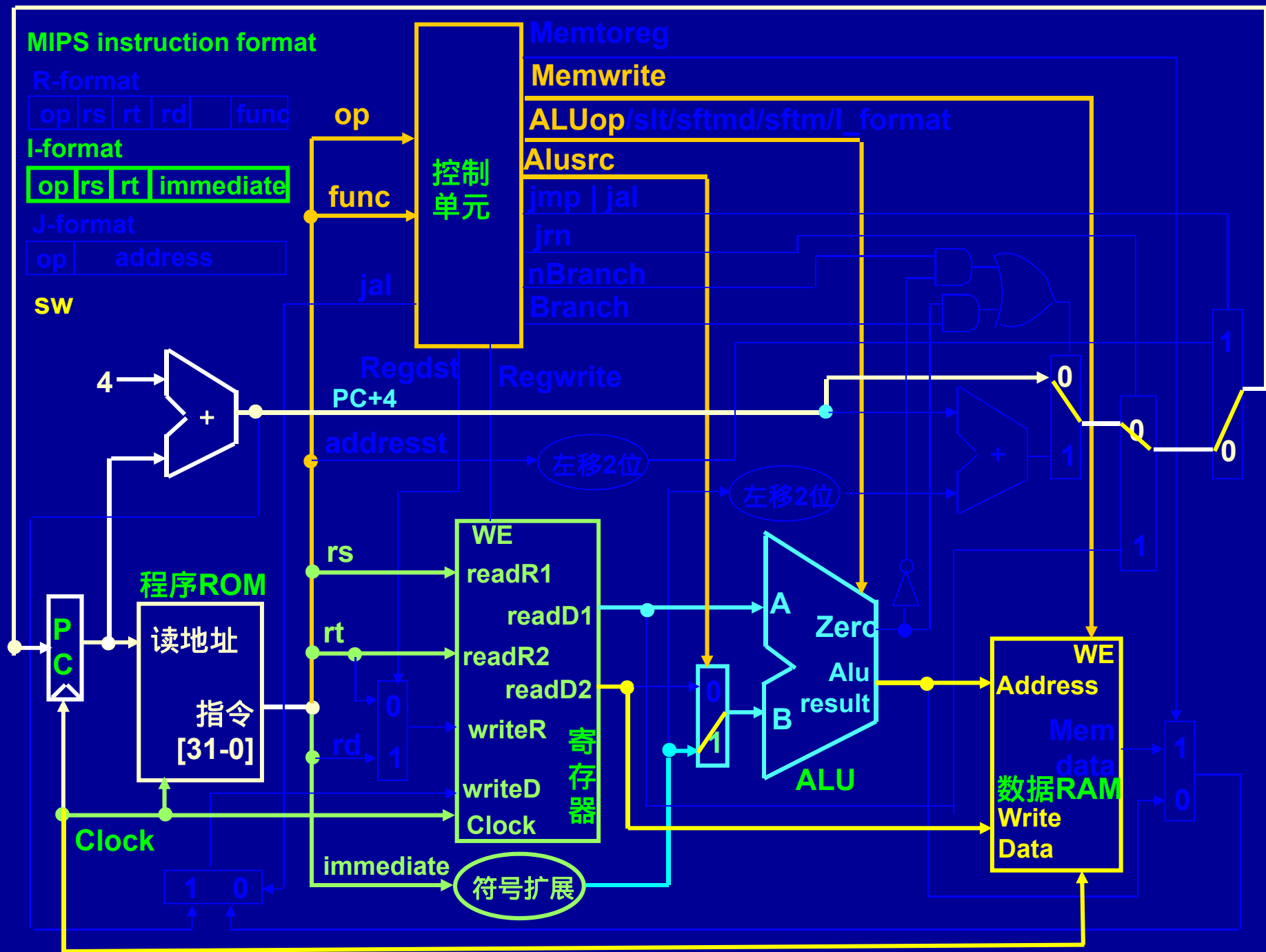
Address

Mem data

数据RAM

Write Data

WE

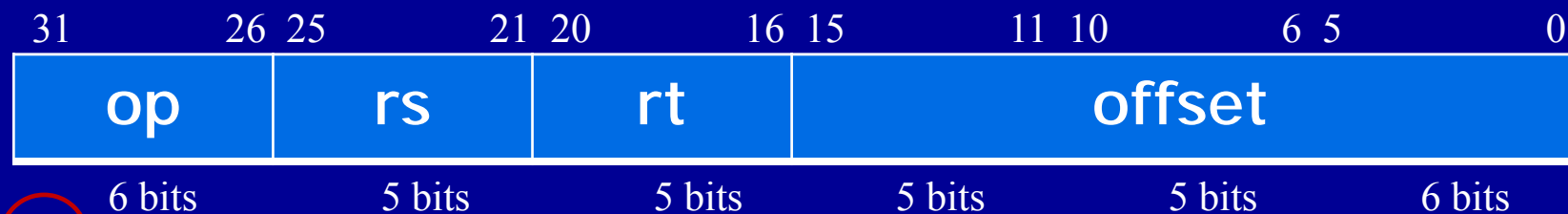


单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ 分支指令数据通路

■ beq指令



😊 指令特点：

- 1) 指令类型仅由op决定；
- 2) $R[rs]$ 与 $R[rt]$ 相减，并以差为0作为继续执行的条件；
- 3) 指令中的16位立即数为偏移量offset，需乘以4后做符号扩展成32位；然后和 $PC+4$ 相加作为新的PC值。该指令在差为0的时候会改变PC值

单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ 分支指令数据通路

■ beq指令

功能： M[PC]；

$Zero \leftarrow R[rs] - R[rt]$, if $Zero = 1$, then go to label
else $pc = pc + 4$; 执行beq下面一条指令

$Label = pc + 4 + (SignExt)offset \ll 2$

😊 增加器件

- ① 加法器——用于符号扩展后的地址计算；
- ② 2选一数据选择器——选择是PC+4还是label;
- ③ 与门。

😊 增加控制信号

Branch——表明是beq指令；
取指单元要将PC+4传到执行单元

MIPS instruction format

R-format

op rs rt rd func

I-format

op rs rt immediate

J-format

op address

beq

控制单元

MemtoReg

Memwrite

ALUop/slt/sftmd/sftm/l format

Alusrc

jmp | jal

jrn

nBranch

Branch

Regdst

Regwrite

PC+4

addressst

左移2位

左移2位

程序ROM

读地址

指令
[31-0]

Clock

immediate

WE
readR1
readD1
readR2
readD2
writeR
writeD
Clock

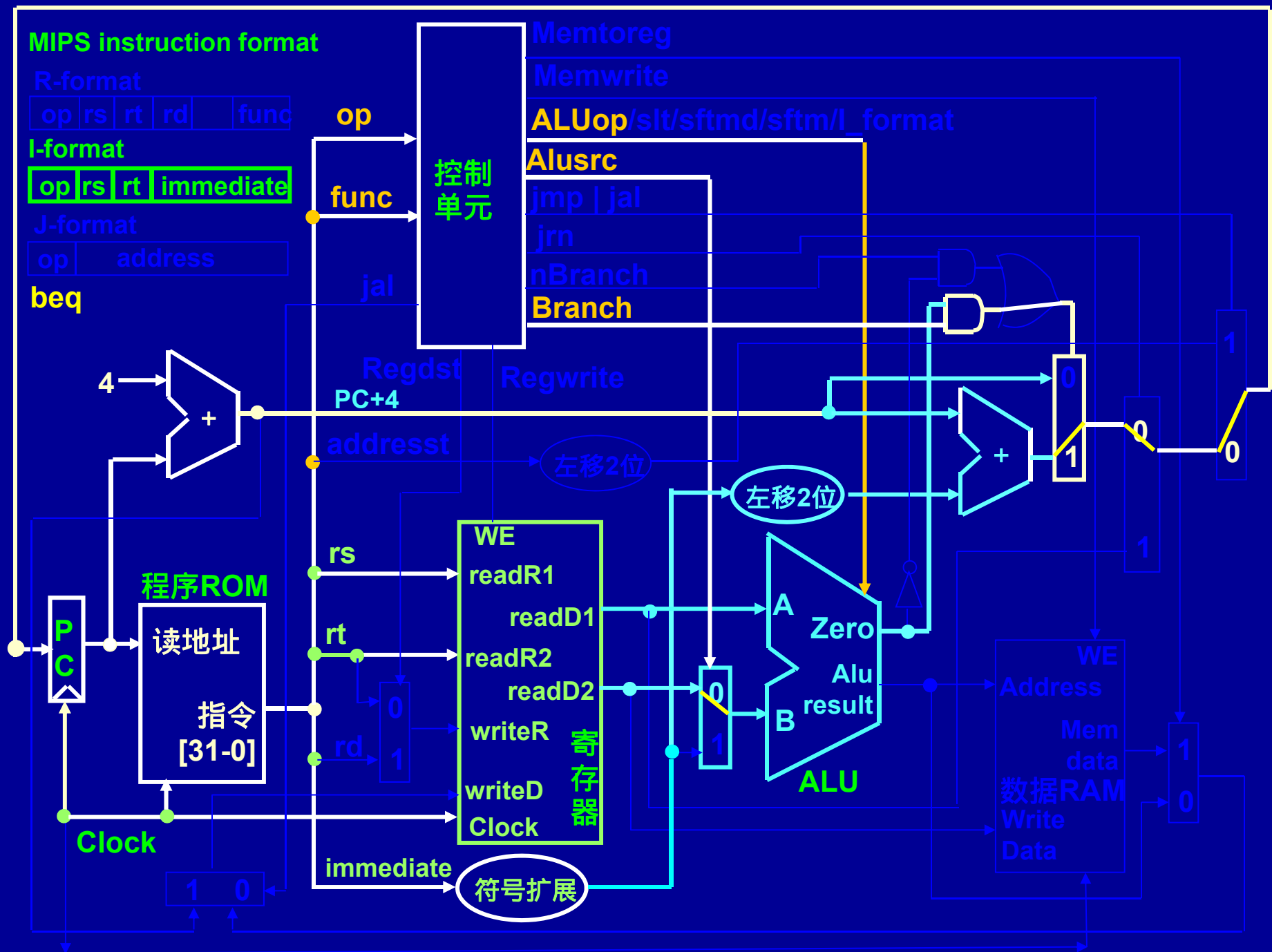
寄存器

符号扩展

ALU

Zero
Alu
result

WE
Address
Mem
data
数据RAM
Write
Data

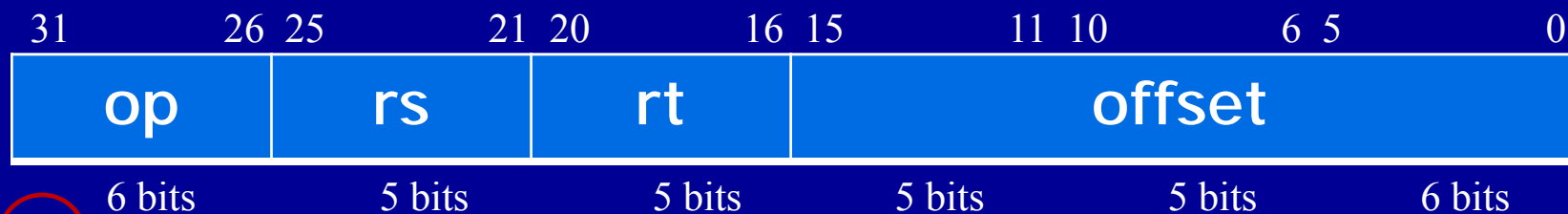


单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ 分支指令数据通路

■ bne指令



😊 指令特点：

- 1) 指令类型仅由op决定；
- 2) $R[rs]$ 与 $R[rt]$ 相减，并以差不为0作为继续执行的条件；
- 3) 指令中的16位立即数为偏移量offset，需乘以4后做符号扩展成32位；然后和 $PC+4$ 相加作为新的PC值。该指令在差不为0的时候会改变PC值

单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ 分支指令数据通路

■ bne指令

功能： M[PC];

$Zero \leftarrow R[rs] - R[rt]$, if $Zero \neq 1$, then go to label
else $pc = pc + 4$; 执行beq下面一条指令

$Label = pc + 4 + (SignExt)offset \ll 2$

😊 增加器件

- ① 反相器；
- ② 与门；
- ③ 或门。

😊 增加控制信号

nBranch——表明是bne指令；
取指单元要讲PC + 4传到执行单元

MIPS instruction format

R-format



I-format



J-format



bne

控制单元

MemtoReg

Memwrite

ALUOp slt/sltmd/sltm/l format/

Alusrc

jmp | jal

jrn

nBranch

Branch

Regdst

Regwrite

PC+4

address

左移2位

左移2位

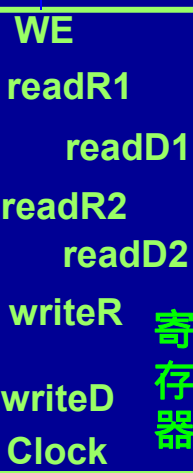
程序ROM

读地址

指令
[31-0]

Clock

immediate



寄存器

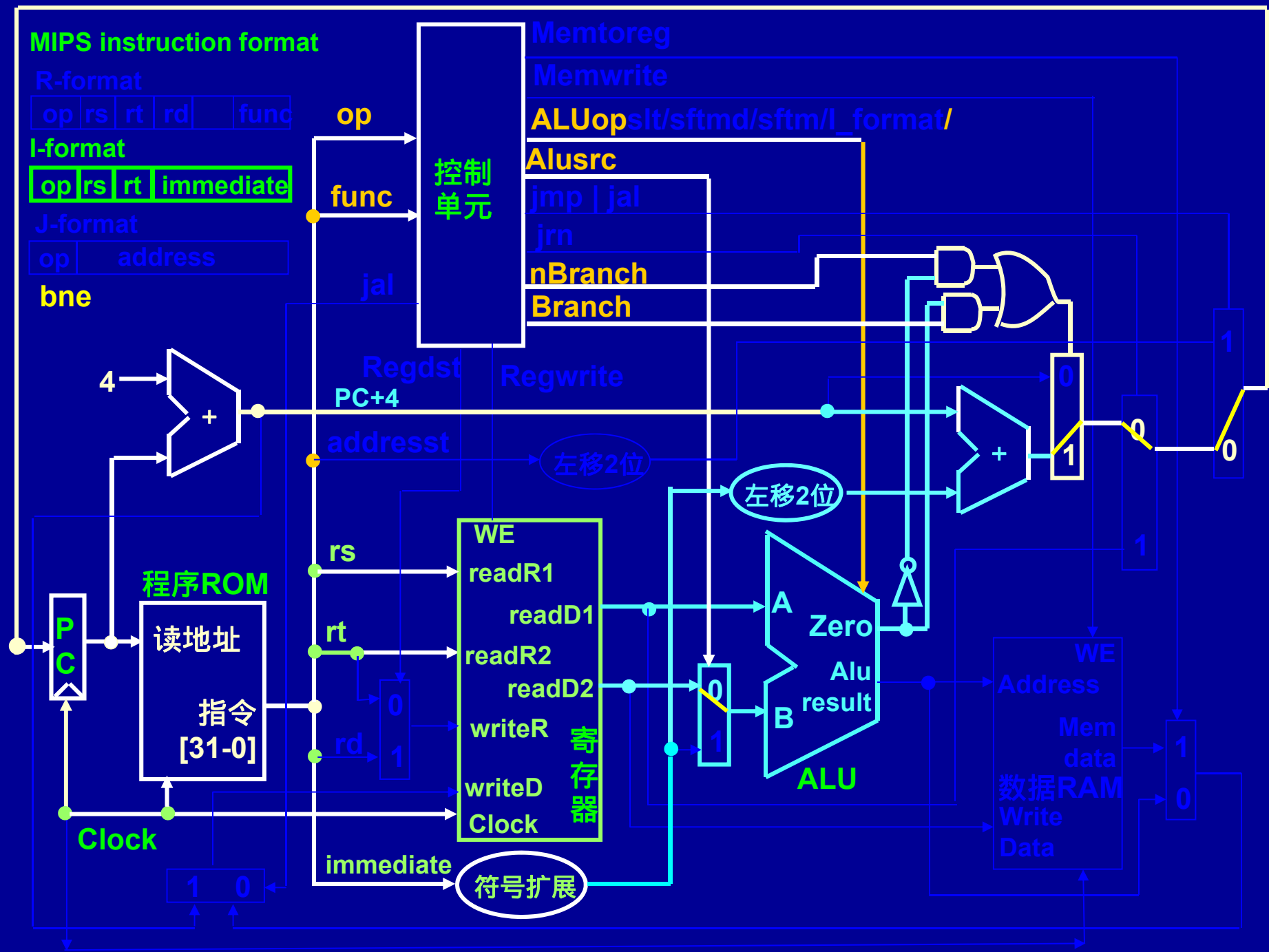
符号扩展

ALU

Zero
Alu result



数据RAM



单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ J类指令数据通路

■ j指令



😊 指令特点：

1) 指令类型仅由op决定。

2) 指令中的26位立即数为地址，需乘以4后作为新的PC值。
该指令会改变PC值



单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ J类指令数据通路

■ j指令

功能： $M[PC]$

$PC \leftarrow \text{ZeroExt}(\text{address} \ll 2)$

😊 增加器件

① 2选—数据选择器——选择正确的新PC值;

😊 增加控制信号

① jmp——表明是jmp指令；

MIPS instruction format

R-format

op rs rt rd func

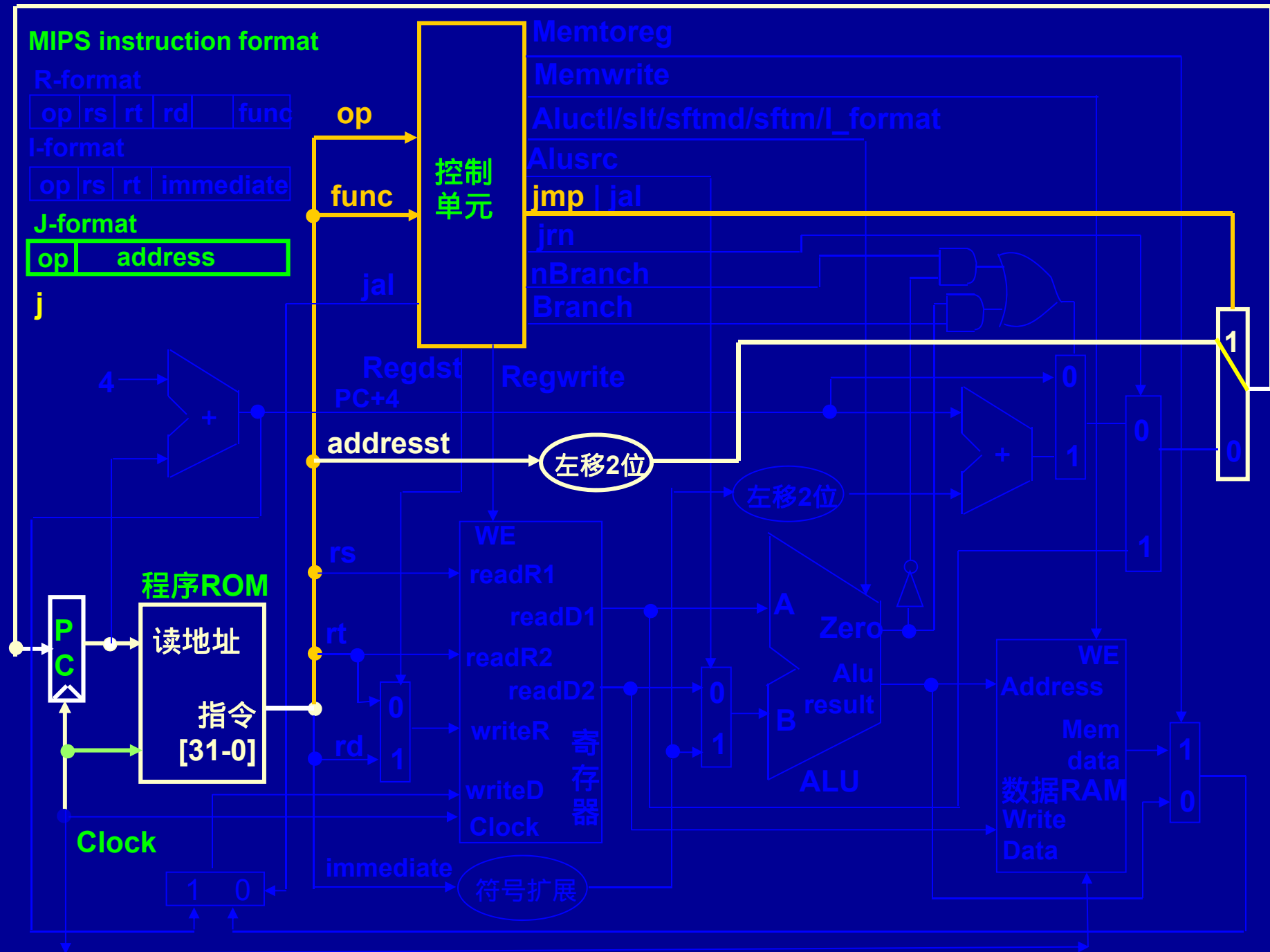
I-format

op rs rt immediate

J-format

op address

j

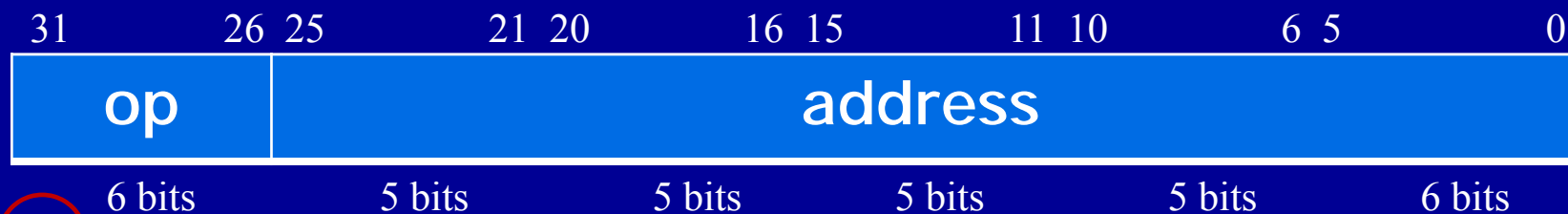


单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ J类指令数据通路

■ jal指令



指令特点：

- 1) 指令类型仅由op决定。
- 2) 将PC+4的值存放到\$31中
- 3) 指令中的26位立即数为地址，需乘以4后作为新的PC值。
该指令会改变PC值

单周期数据通路的设计

■ 单周期数据通路的设计步骤

■ J类指令数据通路

■ jal指令

功能： $M[PC]$

$R[31] \leftarrow PC+4$

$PC \leftarrow \text{ZeroExt}(\text{address} \ll 2)$

😊 增加器件

① 2选一数据选择器——选择PC+4写\$31, 控制信号是jal;

😊 增加控制信号

① **jal**——表明是jal指令；

MIPS instruction format

R-format

op rs rt rd func

I-format

op rs rt immediate

J-format

op address

jal

控制单元

MemtoReg

Memwrite

AluCtrl/slt/sltmd/sltm/l_format

AluSrc

jmp | jal

jrn

nBranch

Branch

Regwrite

Regdst
PC+4

addressst

左移2位

左移2位

程序ROM

读地址

指令
[31-0]

Clock

31

1/0

immediate

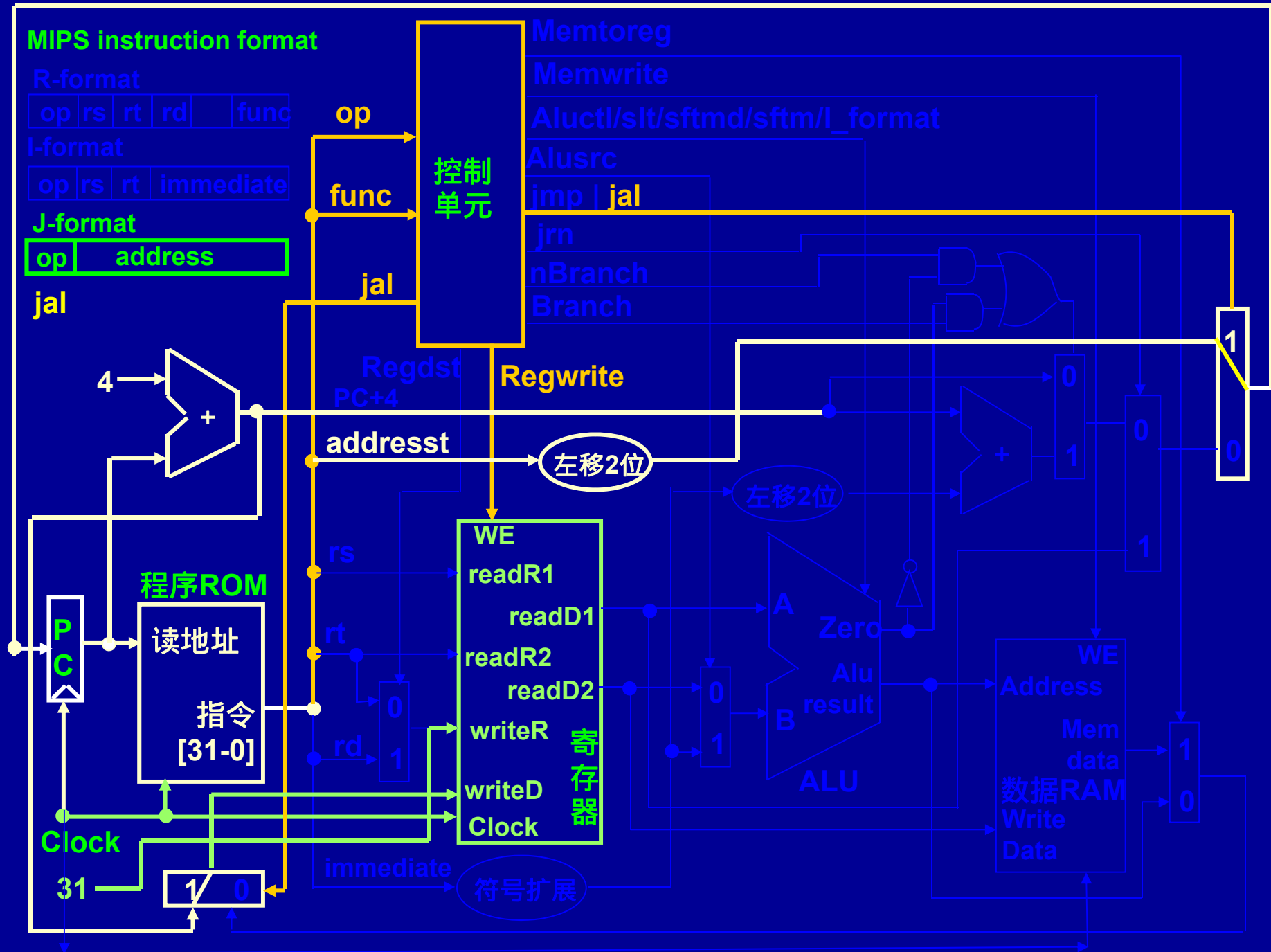
符号扩展

WE
readR1
readD1
readR2
readD2
writeR
writeD
Clock

寄存器

A
Zero
Alu result
B
ALU

WE
Address
Mem data
数据RAM
Write Data



MIPS instruction format

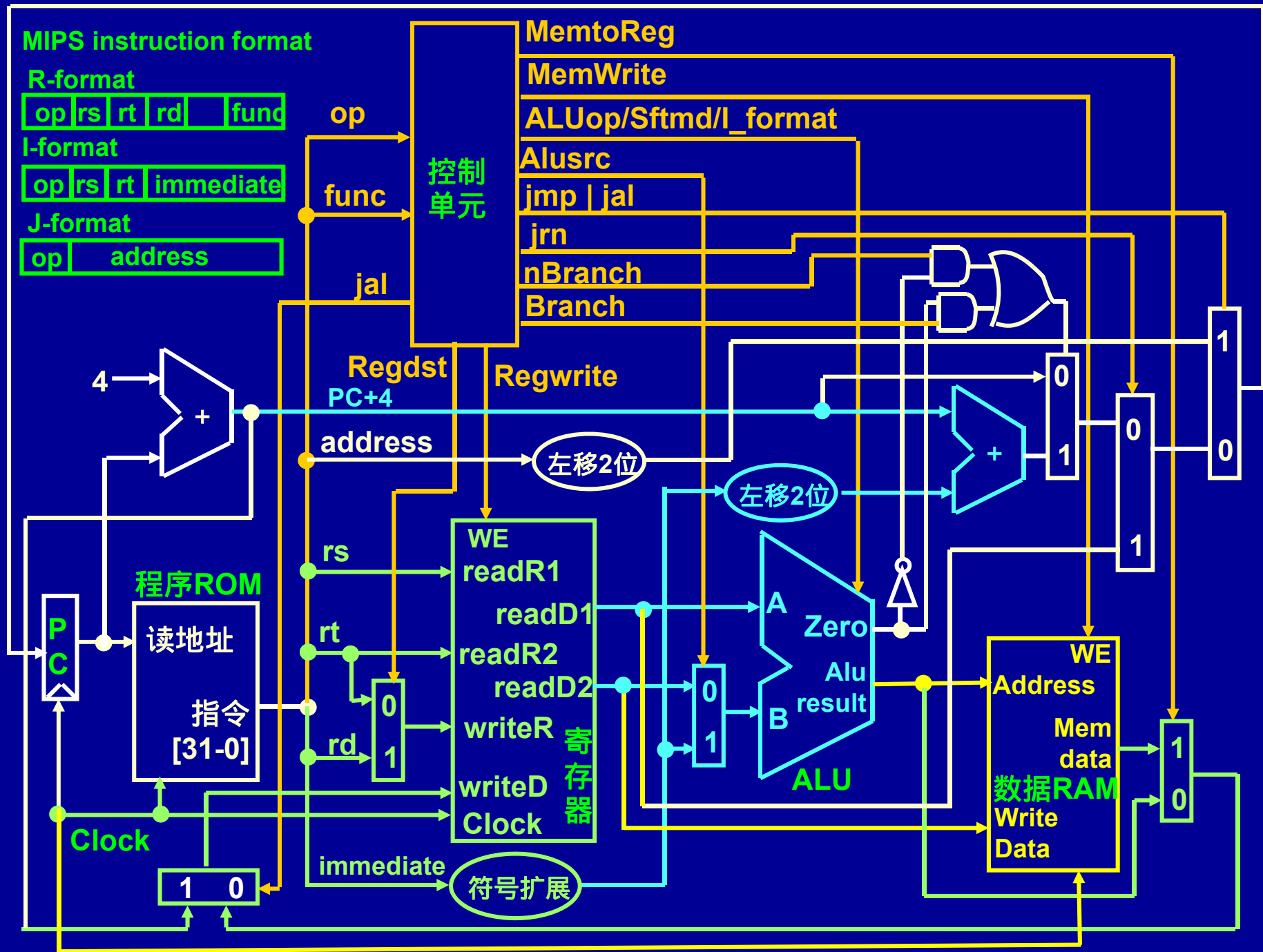
R-format



I-format



J-format



Lecture : Pipelining

College of Computer Science
Chongqing University

Overview

- Pipelining improves performance by overlapping multiple instructions
 - Takes advantage of parallelism among the actions needed to execute an instruction.
 - The number of instructions performed per second is increased
 - The time needed for one instruction is not changed, and may be increased.
- The **throughput** of an instruction pipeline is the measure of how often an instruction exits the pipeline.
- Invisible to high level (OS, programs)

Pipeline Stages

We can divide the execution of an instruction into the following 5 “classic” stages:

IF: Instruction Fetch

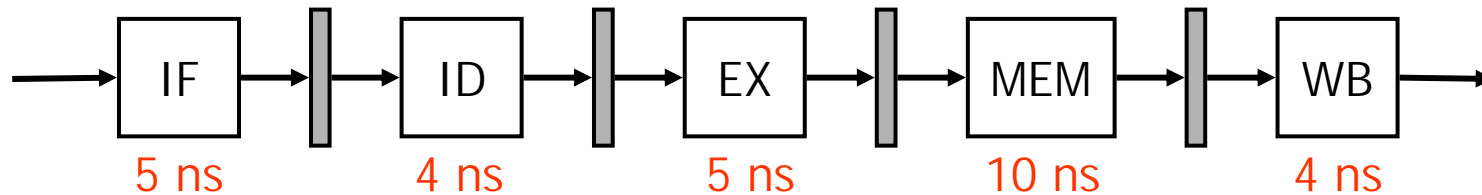
ID: Instruction Decode (w/ register fetch)

EX: Execution

MEM: Memory Access

WB: (Register) Write Back

Pipeline Throughput and Latency

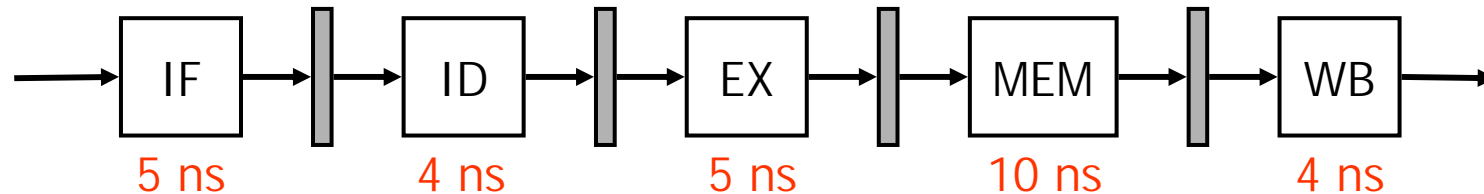


Consider the pipeline above with the indicated delays. We want to know what is the *pipeline throughput* and the *pipeline latency*.

Pipeline throughput: instructions completed per second.

Pipeline latency: how long does it take to execute a single instruction in the pipeline.

Pipeline Throughput and Latency



Pipeline throughput: how often an instruction is completed.

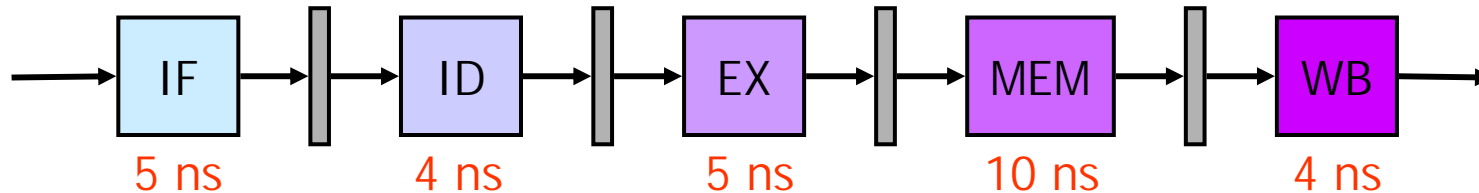
$$\begin{aligned} &= 1instr / \max[lat(IF), lat(ID), lat(EX), lat(MEM), lat(WB)] \\ &= 1instr / \max[5ns, 4ns, 5ns, 10ns, 4ns] \\ &= 1instr / 10ns \quad (\text{ignoring pipeline register overhead}) \end{aligned}$$

Pipeline latency: how long does it take to execute an instruction in the pipeline.

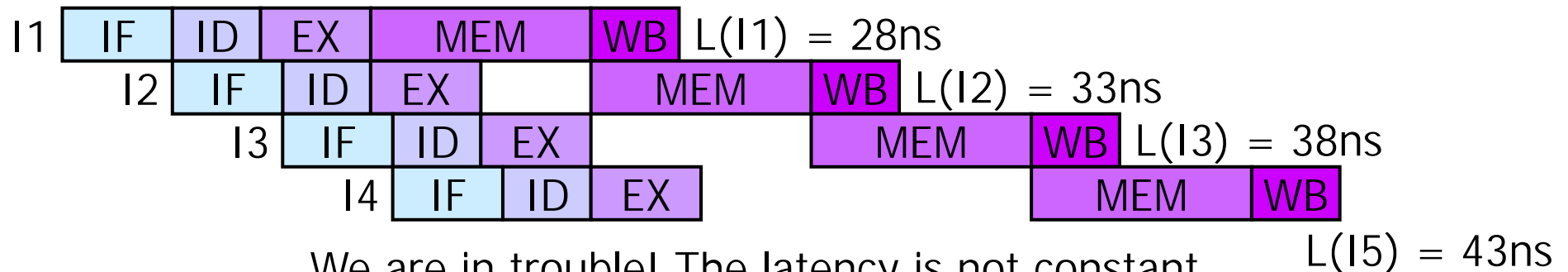
$$\begin{aligned} L &= lat(IF) + lat(ID) + lat(EX) + lat(MEM) + lat(WB) \\ &= 5ns + 4ns + 5ns + 10ns + 4ns = 28ns \end{aligned}$$

Is this right?

Pipeline Throughput and Latency

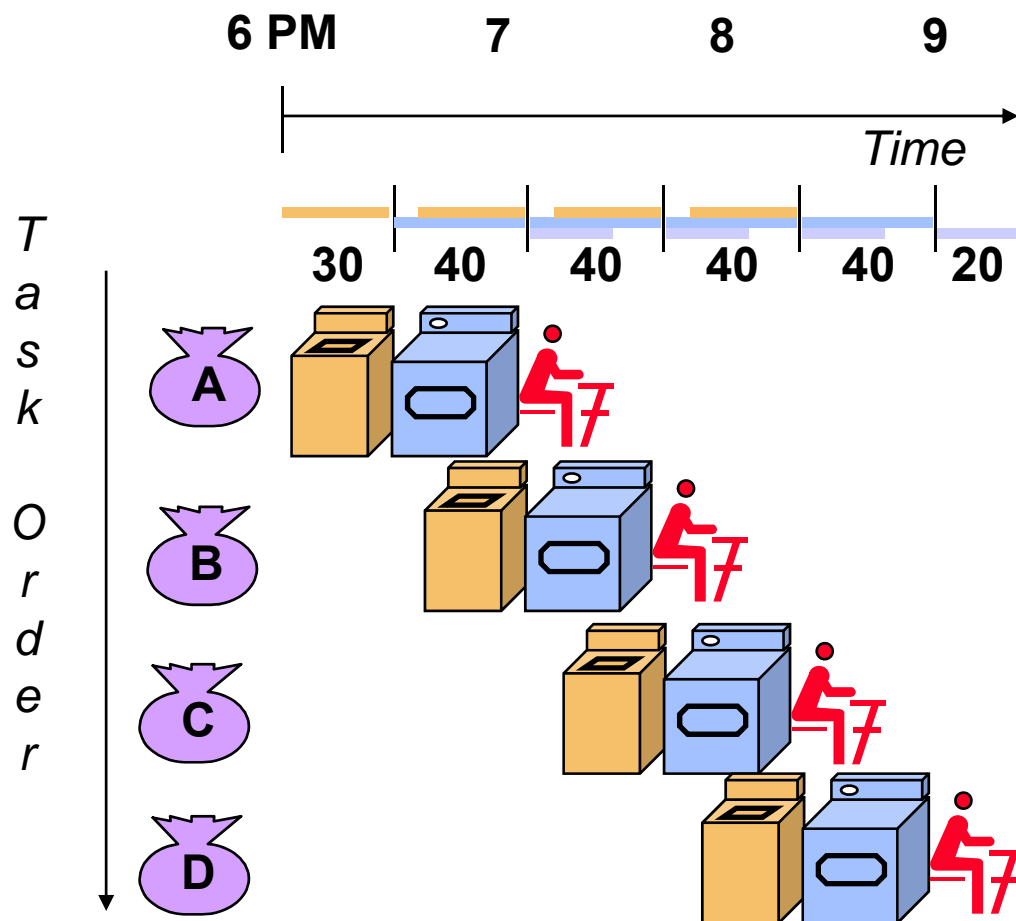


Simply adding the latencies to compute the pipeline latency, only would work for an isolated instruction



We are in trouble! The latency is not constant.
This happens because this is an unbalanced pipeline. The solution is to make every state the same length as the longest one.

Pipelining Lessons



- Pipelining doesn't help **latency** of a single task, it helps **throughput** of the entire workload
- Pipeline rate limited by the **slowest** pipeline stage
- **Multiple** tasks operating simultaneously
- Potential speedup = **number of pipe stages**
- Unbalanced lengths of pipe stages reduces speedup
- Time to "**fill**" pipeline and time to "**drain**" it reduces speedup

Other Definitions

- Pipe stage (or pipe segment)
 - A decomposable unit of the fetch-decode-execute paradigm
- Pipeline depth
 - Number of stages in a pipeline
- Machine cycle
 - Clock cycle time
- Latch
 - Per phase/stage local information storage unit

Design Issues

- Balance the length of each pipeline stage

$$\text{Throughput} = \frac{\text{Depth of the pipeline}}{\text{Time per instruction on unpipelined machine}}$$

- Problems
 - Usually, stages are not balanced
 - Pipelining overhead
 - Hazards (conflicts)
- Performance (throughput → CPU performance equation)
 - Decrease of the CPI
 - Decrease of the cycle time

We use RISC architecture to illustrate

- All operations on data apply to data in registers, and typically change the entire register (32 or 64 bits per register).
- The only operations that affect memory are load and store.
 - Load and store operations that load or store less than a full register (e.g., a byte, 16 bits, or 32 bits) are often available.
- The instruction formats are few in number, with all instructions typically being one size.

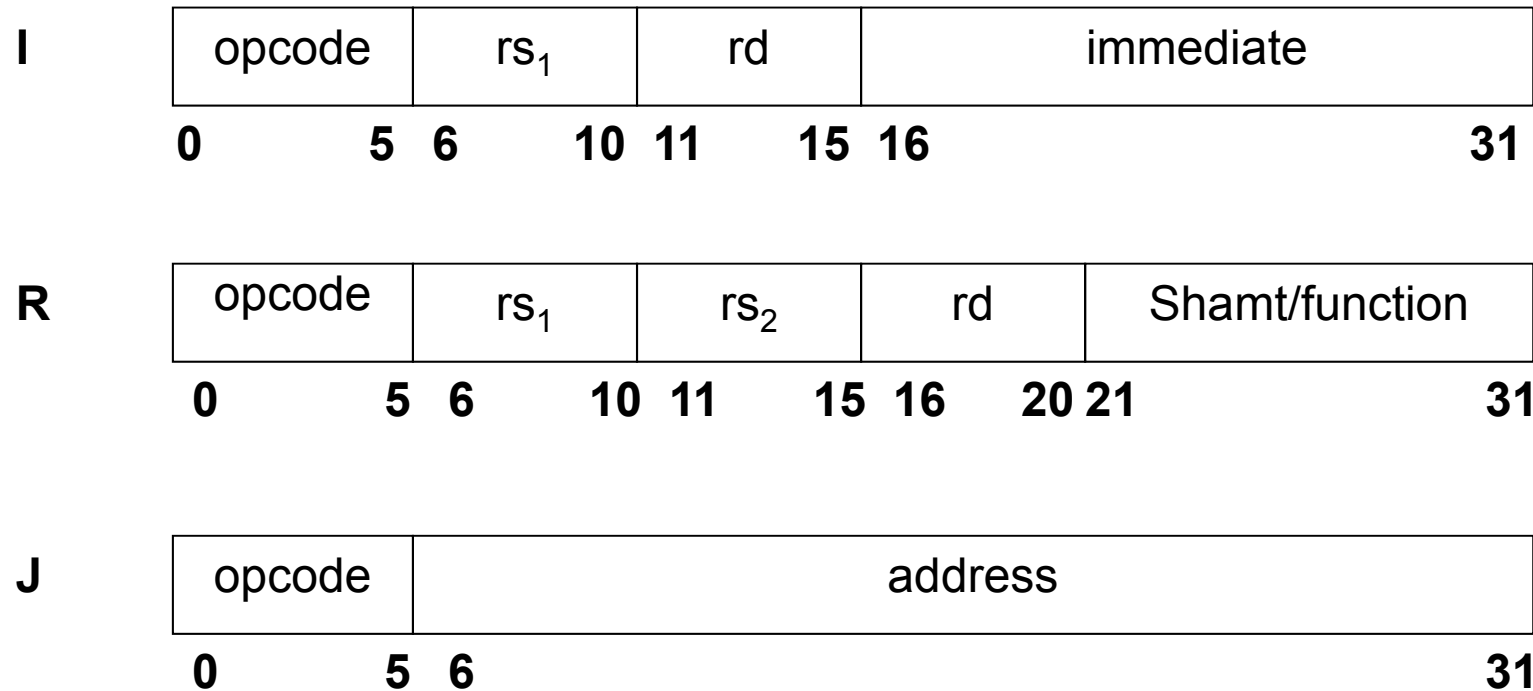
MIPS Instructions, one of RISC ISAs

- ALU instructions
 - Take either two registers or a register and a sign-extended immediate, operate on them, and store the result into a third register.
 - add (DADD), subtract (DSUB), and logical operations (such as AND or OR),
 - Immediate versions of these instructions use the same mnemonics with a suffix of I.
- There are both signed and unsigned forms of the arithmetic instructions;
 - the unsigned forms, which do not generate overflow exceptions— and thus are the same in 32-bit and 64-bit mode—have a U at the end (e.g., DADDU, DSUBU, DADDIU)..

MIPS Instructions, one of RISC ISAs

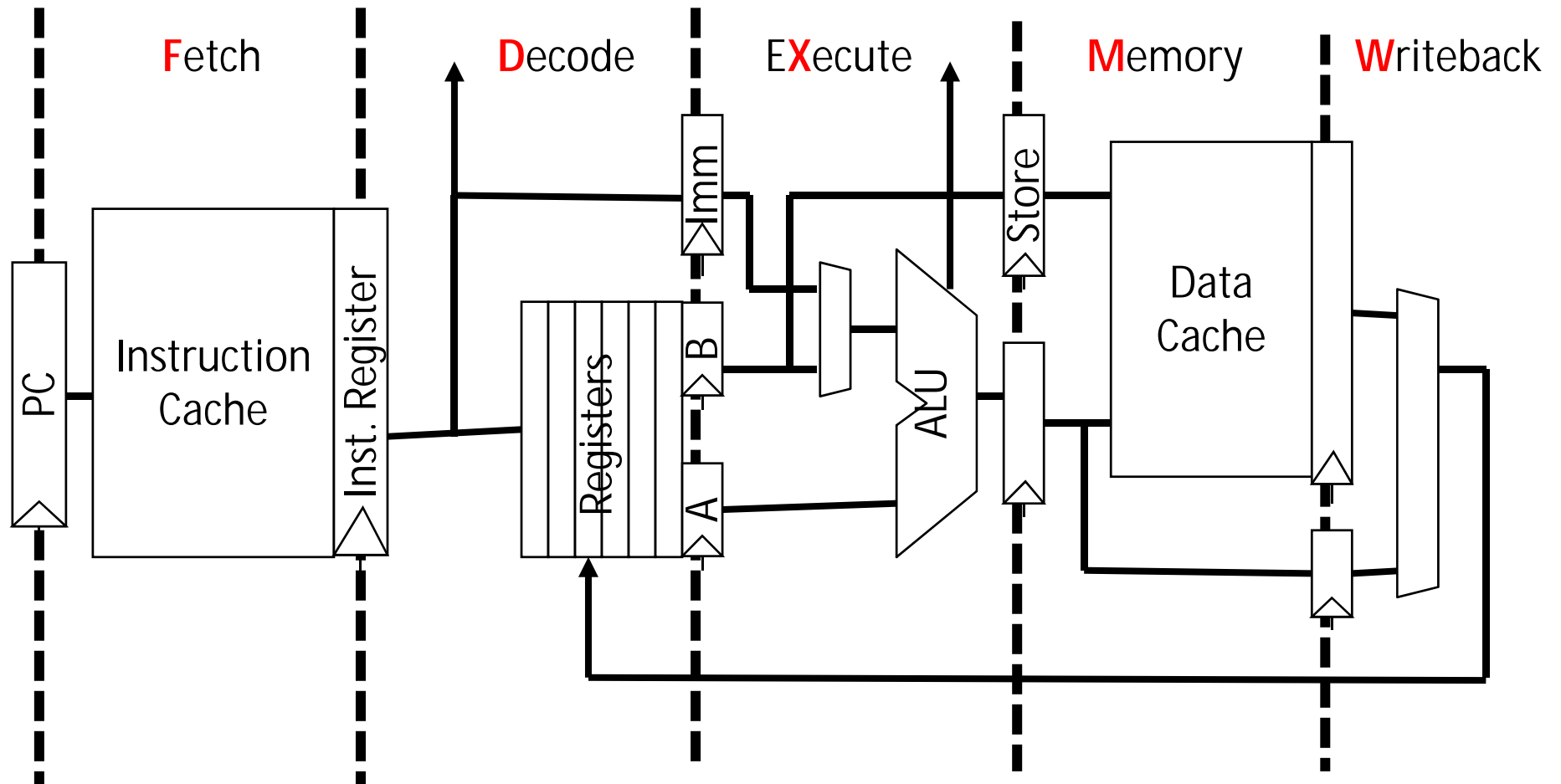
- Load and store instructions
 - base register + an immediate offset = effective address.
 - In the case of a load instruction, a 2nd register acts as the destination.
 - In the case of a store, the 2nd register operand is the source of the data that is stored into memory.
- Branch and jump instructions:
 - MIPS uses a set of comparisons between a pair of registers or between a register and zero.
 - The branch destination is obtained by adding a sign-extended offset (16 bits in MIPS) to the current PC.

MIPS Instruction Formats (all 4-byte long)



Fixed-field decoding

Classic 5-Stage RISC Pipeline



*This version designed for regfiles/memories
with synchronous reads and writes.*

The 1st stage of the 5-stage MIPS pipeline: Instruction Fetch (IF)

- Fetch the current instruction from memory according to the program counter (PC).
- Update the PC to the address of the next instruction in sequence by adding 4 (since each instruction is 4 bytes) to the PC.

$IR \leftarrow \text{Mem}[PC];$

$NPC \leftarrow PC + 4$

The 2nd stage of the 5-stage MIPS pipeline: Instruction Decoding (ID) w/ Register Fetch

- Decode the instruction and read the registers;
- Do the equality test on the registers as they are read, for a **possible** branch;
- Sign-extend the offset field of the instruction **in case** it is needed;
- Compute the **possible** branch target address by adding the sign-extended offset to the incremented PC;
 $A \leftarrow \text{Regs}[\text{IR}_{6..10}]$
 $B \leftarrow \text{Regs}[\text{IR}_{11..15}]$
 $\text{Imm} \leftarrow ((\text{IR}_{16})^{16} \# \# \text{IR}_{16..31})$
- In an aggressive implementation, branch instruction can be completed \rightarrow by storing the target address into the PC, if the condition test yielded true;

3rd Instruction cycle: Execution (EX)

- Execution or effective address calculation
 - In a load-store architecture, no instruction needs to simultaneously calculate a data address and perform an operation on the data
 - ALU will work on one of the two possibilities
- Do one of the following operations:
 - Memory reference: Base register + Immediate Offset
 - $\text{ALUOutput} \leftarrow A + \text{Imm}$
 - Register - Register ALU instruction: specified by op code
 - $\text{ALUOutput} \leftarrow A \text{ func } B$
 - Register - Immediate ALU instruction: specified by op code
 - $\text{ALUOutput} \leftarrow A \text{ func Imm}$

4th Instruction cycle: Memory Access (MEM)

- Memory access
 - Memory reference
 - $PC \leftarrow NPC$
 - $LMD \leftarrow Mem[ALUOutput]$ (load)
 - $Mem[ALUOutput] \leftarrow B$ (store)

5th Instruction cycle: Write Back (WB)

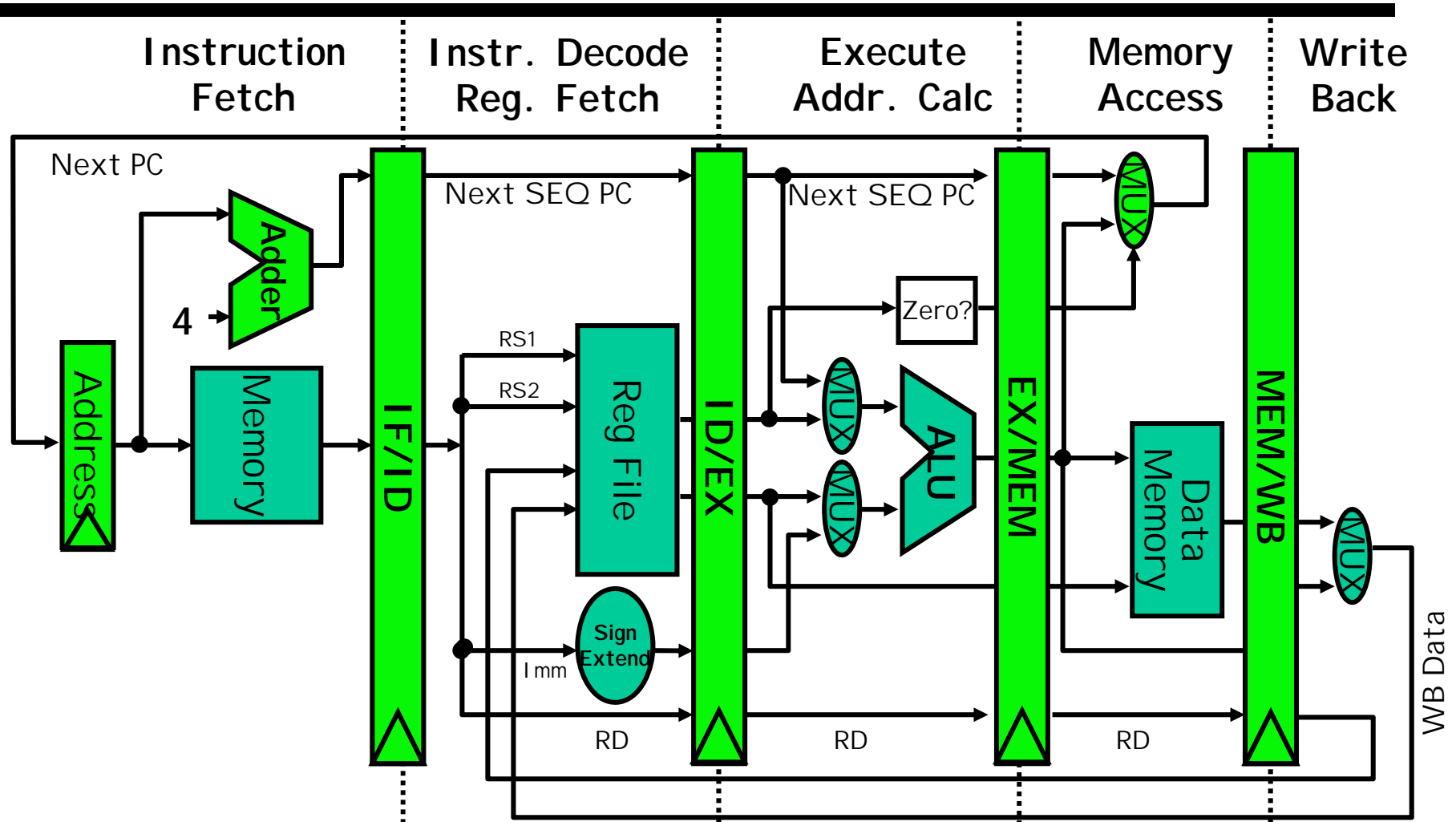
- Write-back (WB)
 - Register - register ALU instruction
 - $\text{Regs}[\text{IR}_{16..20}] \leftarrow \text{ALUOutput}$
 - Register - immediate ALU instruction
 - $\text{Regs}[\text{IR}_{11..15}] \leftarrow \text{ALUOutput}$
 - Load instruction
 - $\text{Regs}[\text{IR}_{11..15}] \leftarrow \text{LMD}$

Summary: The classic 5-stage pipeline

- Branch: 2 cycles, finish at ID;
- Store 4 cycles, finish at MEM;
- All other instructions: 5 cycles, finish at WB.
- Assuming 12% branch instructions, 10% store instructions, leads to an overall CPI of 4.54.

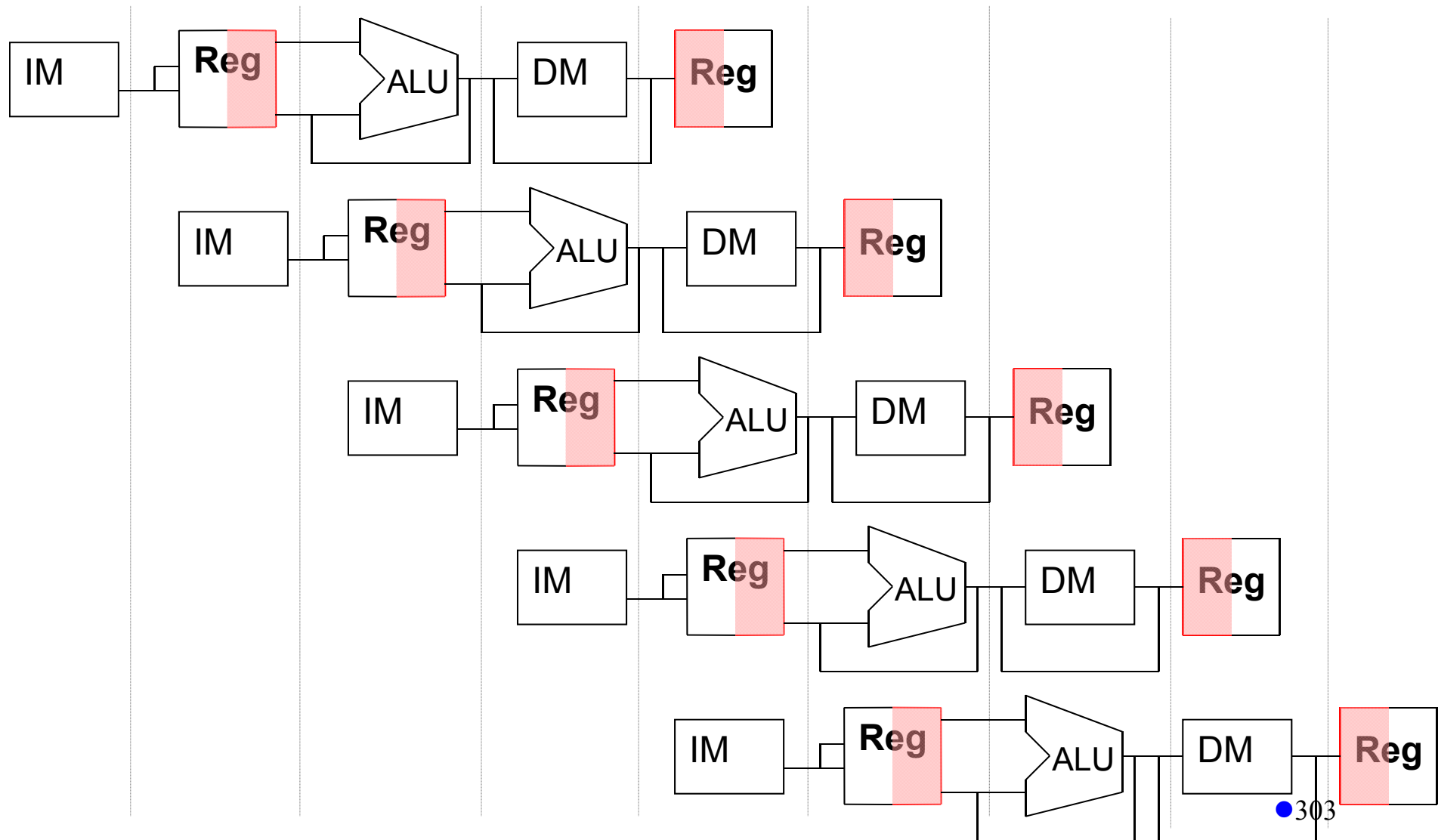
| Instruction number | Clock number | | | | | | | | |
|---------------------|--------------|----|----|-----|-----|-----|-----|-----|----|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Instruction i | IF | ID | EX | MEM | WB | | | | |
| Instruction $i + 1$ | | IF | ID | EX | MEM | WB | | | |
| Instruction $i + 2$ | | | IF | ID | EX | MEM | WB | | |
| Instruction $i + 3$ | | | | IF | ID | EX | MEM | WB | |
| Instruction $i + 4$ | | | | | IF | ID | EX | MEM | WB |

Pipelined MIPS Datapath Schematic



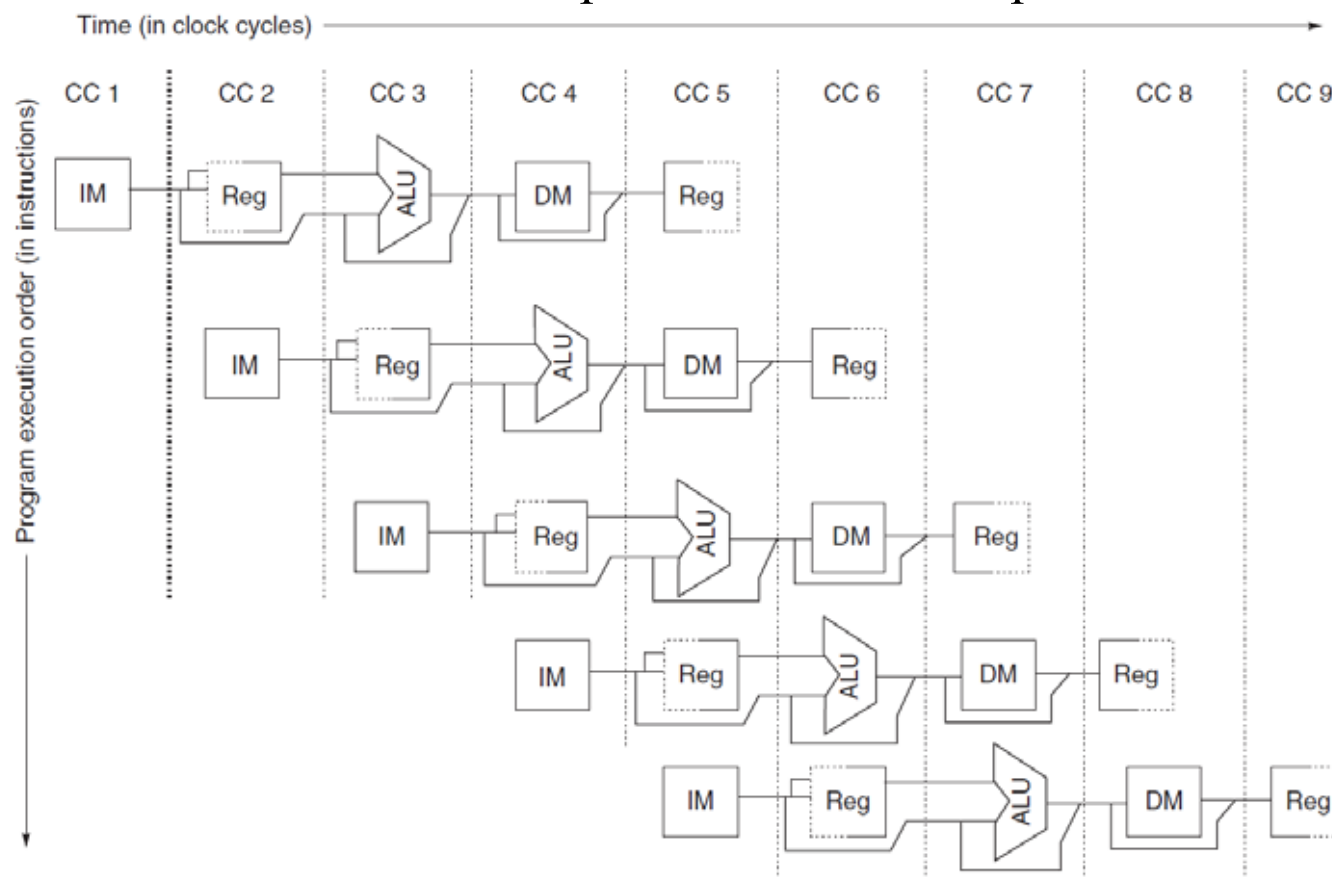
- Data stationary control
 - local decode for each instruction phase / pipeline stage

Pipeline Resources



The challenge on designing a pipelined datapath

- Need to make sure that a hardware unit is not asked to perform two operations in the same cycle.
 - For example, an ALU cannot be asked to compute an effective address and perform a subtract operation at the same time.



IM: instruction memory

DM: data memory

CC: clock cycles

Several Observations

- Use separate instruction and data memories, implemented by separate instruction and data caches;
 - It eliminates a conflict between instruction fetch and data memory access.
- Reg is used in both ID (for reading) and WB (for writing). Hence, two reads (distinct) and one write every clock cycle;
 - Notice that Write is always from earlier instruction and Reads are from later instructions.
 - Perform Write on the 1st half of the cc (rising edge of clock) and Read in the 2nd half (falling edge of clock).
- Operations on PC are not shown.
 - PC must be incremented every clock, and must be done at IF in preparation for the next instruction.
 - A branch does not change the PC until the end of ID stage.
 - This causes a problem, will be discussed shortly.
 - Needs an independent adder only for PC addition/subtraction.

Performance Issues in Pipelining

- Pipelining increases performance by running multiple instructions simultaneously. But it does not reduce the latency of every single instruction.
 - Due to pipeline overhead, the latency of an instruction is in fact increased slightly.
- Pipeline overhead 1: Imbalanced pipeline stages:
 - Slowest stage dominates the total throughput.
- Pipeline overhead 2: Pipeline registers and clock skew
 - Registers introduce setup time into clock frequency.
 - Clock skew: We assume all registers see the clock edges at the exact same time. In real work, due to physical design imperfection, some registers see edges earlier than others – clock skew.
- Hazards: we will discuss it in the next lecture.

MIPS Pipelining: Basic Performance Issues

- Pipelining

- improve CPU Throughput (reciprocal of CPU Time)
- slightly increases execution time
- result: program run faster!

$$[\text{time per instruction}]_{\text{pipelined}} = \frac{[\text{time per instruction}]_{\text{nonpipelined}}}{\text{number of pipeline stages}}$$

- Pipelining can be seen as either

- decreasing the CPI of a multi-cycle un-pipelined implementation
- decreasing the CCT of a single-cycle un-pipelined implementation

$$\text{CPU Time} = \text{IC} \times \text{CPI} \times \text{CCT}$$

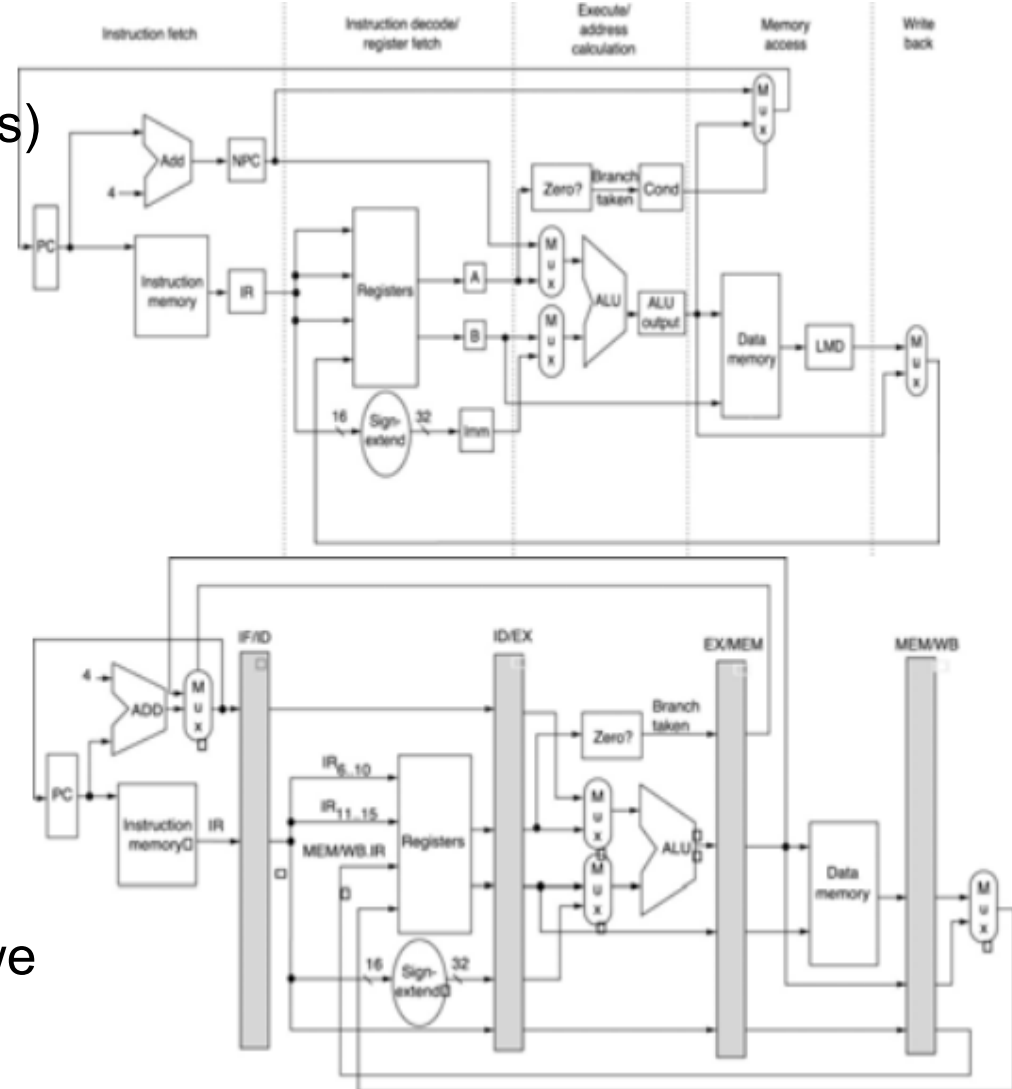
| | IC | CPI | CCT |
|-----------------|----|-----|-----|
| Program | ✓ | | |
| Compiler | ✓ | | |
| ISA | ✓ | ✓ | |
| HW organization | | ✓ | ✓ |
| HW technology | | | ✓ |

A question

- Consider the unpipelined processor. Assume that it has a 1ns clock cycle and that it uses 4 cycles for branches and stores and 5 cycles for other operations. Assume that the relative frequencies of branch and store operations are 15% and 10%, respectively.
- Consider a pipelined processor. Assume the slowest stage takes 1ns and clock skew and register setup add 0.2 ns to the clock period.
- How much speedup in the instruction execution rate will we gain from an ideal pipeline?

Example: Pipelining Speedup vs. Multi-cycle Non-Pipelined Implementation

- **Instruction CPI**
 - 4 cycles (branches and stores)
 - 5 cycles (other instructions)
- **Average CPI = 4.75 assuming**
 - 15% branches
 - 10% stores
- **Average Instruction ExecTime is 4.75ns, assuming CCT = 1ns**
- **Ideal CPI = 1 (almost always)**
- **Average Instruction ExecTime is 1.2ns with CCT = 1ns and clock overhead = 0.2 ns**
- **Speedup is 3.96x for ideal pipeline**
 - as we'll see soon, in reality we need to sum the pipeline stalled clock cycles



PIPELINE HAZARDS

Pipeline Hazards and Their Classification

- Pipeline hazard situation
 - The next instruction cannot execute in the following clock cycle
- Three different flavors
 - Structural Hazards
 - arise from resource conflict: the HW cannot support all possible instruction combinations simultaneously in overlapped execution
 - Data Hazards
 - arise when an instruction depends on the result of a previous instruction in a way exposed by the pipeline overlapped execution
 - Control Hazards
 - arise from the pipelining of branches, jumps...
- Hazards may force pipeline stalling
 - Instructions issued after the stalled one must stall also (and fetching is stalled) while all those issued earlier must proceed

Performance of Pipelines with Stalls

Pipelining can be thought of as decreasing the CPI: This is for the case where both unpipelined version and pipelined version take the same amount of stages for instructions. Unpipelined version just does not fetch new instruction until the previous one finishes. The clock cycles of two versions are roughly same.

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}} \\ &= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}} \\ &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}}\end{aligned}$$

Clock cycle unpipelined
= Clock cycle pipelined
If we ignore the pipeline overhead.

With Stalls

$$\begin{aligned}\text{CPI pipelined} &= \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction} \\ &= 1 + \text{Pipeline stall clock cycles per instruction}\end{aligned}$$

Speedup, assuming all instructions take the same amount of stages

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1 + \text{Pipeline stall cycles per instruction}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

Performance of Pipelines with Stalls

Pipelining can be thought of as reducing the clock cycle: This is for the case where unpipelined version takes only one cycle for all instructions, while pipelined version take multiple stages. Hence the clock cycle of unpipelined version is more than that of the pipelined version.

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{\text{CPI unpipelined}}{\text{CPI pipelined}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} \\ &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}\end{aligned}$$

In cases where the stages are perfectly balanced and there is no overhead, the clock cycle of pipelined version is smaller than that of the unpipelined version by a factor equal to the pipeline depth:

$$\text{Clock cycle pipelined} = \frac{\text{Clock cycle unpipelined}}{\text{Pipeline depth}}$$

$$\text{Pipeline depth} = \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}}$$

$$\begin{aligned}\text{Speedup from pipelining} &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \frac{\text{Clock cycle unpipelined}}{\text{Clock cycle pipelined}} \\ &= \frac{1}{1 + \text{Pipeline stall cycles per instruction}} \times \text{Pipeline depth}\end{aligned}$$

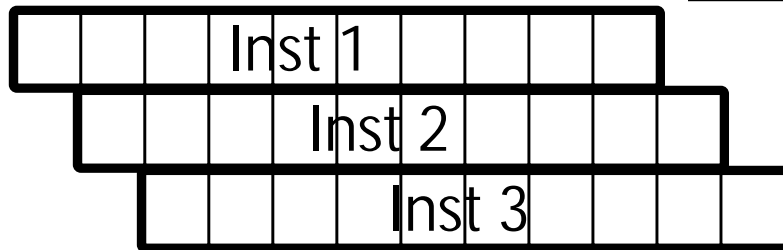
Structural Hazards

- Overlapping instructions require:
 - Pipelining of functional units
 - Duplication of resources
- Structural Hazard happens when the pipeline cannot accommodate some combination of instructions
- Common reasons for Structural Hazard
 - A unit is not fully pipelined → needs to finish current instruction before serving the next,
 - e.g. an instruction takes more than one clock cycle to go through.
 - Some resource has not been duplicated enough,
 - e.g. a register file has only one write port, but more than one instructions in pipeline may want to write at one stage.
- Consequences
 - **Stall** the pipeline until the resource becomes available, which creates a **Bubble**
 - Increase CPI from its ideal value

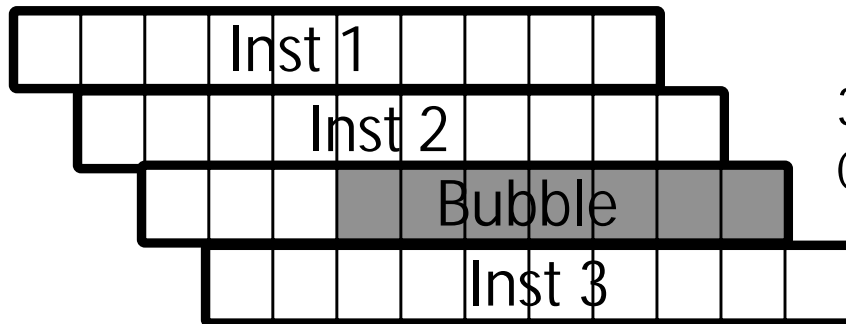
Pipeline CPI Examples

Measure from when first instruction finishes to when last instruction in sequence finishes.

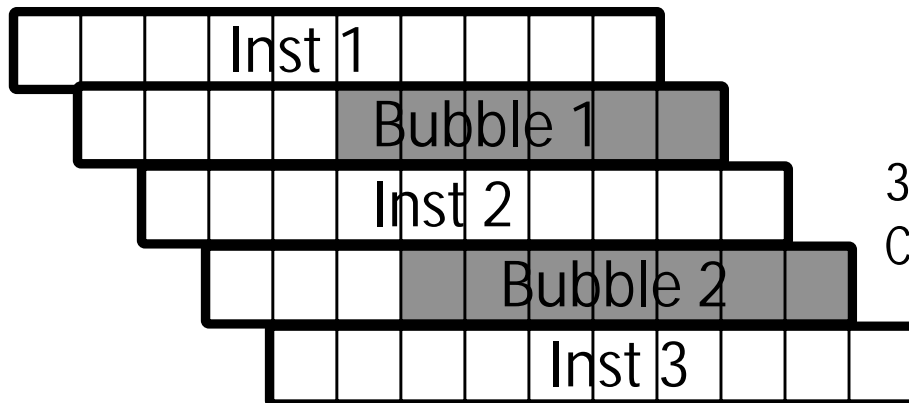
Time →



3 instructions finish in 3 cycles
 $CPI = 3/3 = 1$



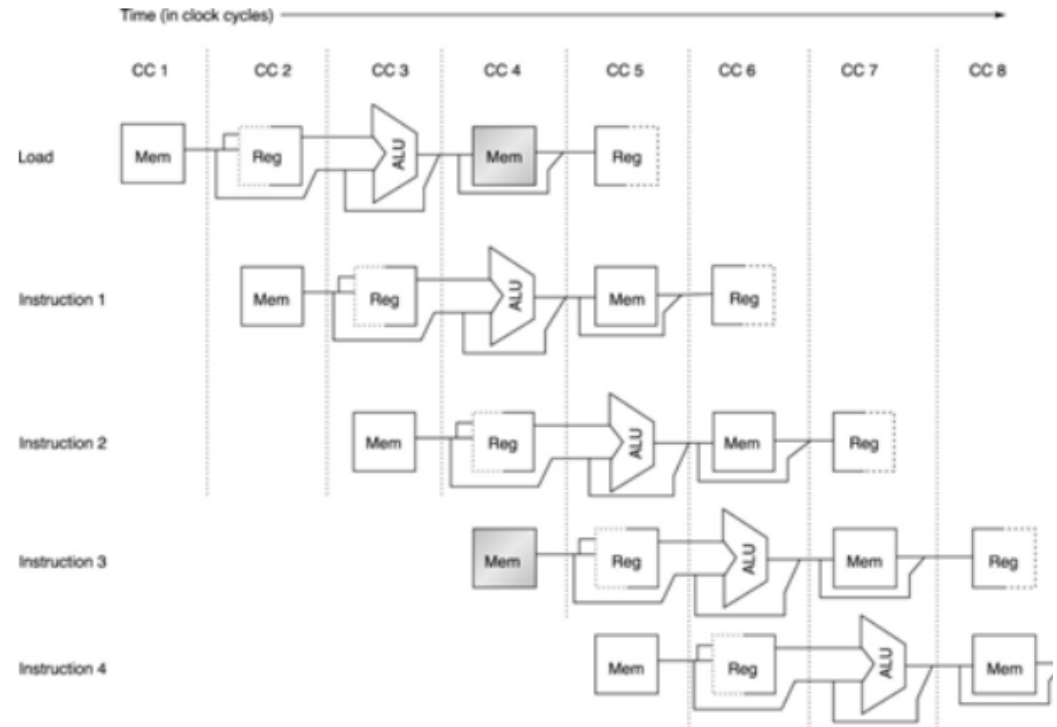
3 instructions finish in 4 cycles
 $CPI = 4/3 = 1.33$



3 instructions finish in 5 cycles
 $CPI = 5/3 = 1.67$

Structural Hazards: Examples

- Some pipelined processors have shared a single memory for data and instructions. As a result, when an instruction contains a data memory reference (at MEM), it will conflict with the instruction reference for a later instruction (at IF)



| instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|-----------------|----|----|----|-----|-----|-----|----|-----|----|
| LW R10, 20(R1) | IF | ID | EX | MEM | WB | | | | |
| SUB R11, R2, R3 | | IF | ID | EX | MEM | WB | | | |
| ADD R12, R3, R4 | | | IF | ID | EX | MEM | WB | | |
| STALL | | | | | | | | | |
| ADD R14, R5, R6 | | | | | IF | ID | EX | MEM | WB |

Resolving Structural Hazards

- Structural hazard occurs when two instructions need same hardware resource at same time
 - Can resolve in hardware by stalling newer instruction till older instruction finished with resource
- A structural hazard can always be avoided by adding more hardware to design
 - E.g., if two instructions both need a port to memory at same time, could avoid hazard by adding second port to memory
- Classic RISC 5-stage integer pipeline has no structural hazards by design
 - Many RISC implementations have structural hazards on multi-cycle units such as multipliers, dividers, floating-point units, etc., and can have on register writeback ports

Structural Hazards Consideration / Trade-off

- Completely avoiding structural hazards is very costly :
 - Pipelined functional units are more costly (pipelined FP ALU are costly)
 - Duplication of resources (separate instruction and data caches means doubled memory and bandwidth)
- If Structural Hazard is rare, just let it be.
 - For example, if FP operations are rare, then just use a non-pipelined FP ALU.

Impact of Stalls on the Performance of Pipelined Implementation - Example

- Implementation without structural hazards
 - ideal CPI = 1
 - Clock Cycle = 1ns
- Implementation with the “load” structural hazard
 - Clock Cycle = 0.9ns
- Suppose that
 - 40% of the executed instructions are loads or stores
- Which implementation is faster?
 - $(\text{averageInstructionTime})_{\text{noHaz}} = 1 * 1 = 1$
 - $(\text{averageInstructionTime})_{\text{haz}} = (1 + 0.4 * 1) * 0.9 = 1.26$
 - implementation without structural hazard is 1.26 times faster than the other

Impact of Stalls on the Performance of Pipelined Implementation

$$\text{speedupFromPipelining} = \frac{[CPI]_{\text{nonpipelined}} \times [CCT]_{\text{nonpipelined}}}{[CPI]_{\text{pipelined}} \times [CCT]_{\text{pipelined}}}$$

$$\begin{aligned}[CPI]_{\text{pipelined}} &= \text{idealCPI} + \text{pipelineStallCyclesPerInstruction} \\ &= 1 + \text{pipelineStallCyclesPerInstruction}\end{aligned}$$

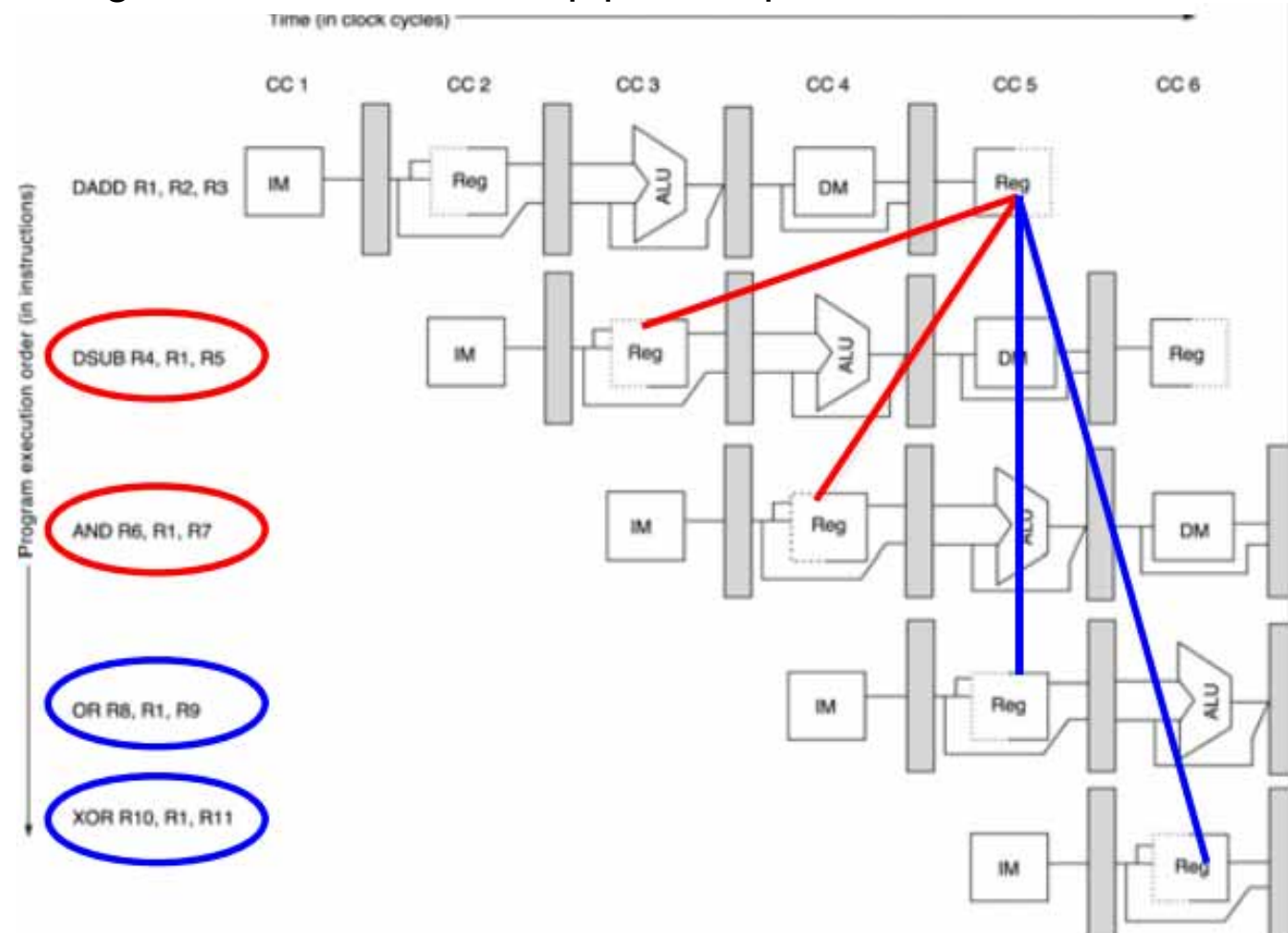
CCT = Clock Cycle Time

- Special case: ignoring the pipeline clock-period overhead and assume that
 - the pipeline stages are perfectly balanced
 - all instructions take the same number of cycles in the non-pipelined implementation (equal to the pipeline depth)

$$\text{speedupFromPipelining} = \frac{\text{pipelineDepth}}{1 + \text{pipelineStallCyclesPerInstruction}}$$

Data Hazards

Data Hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor

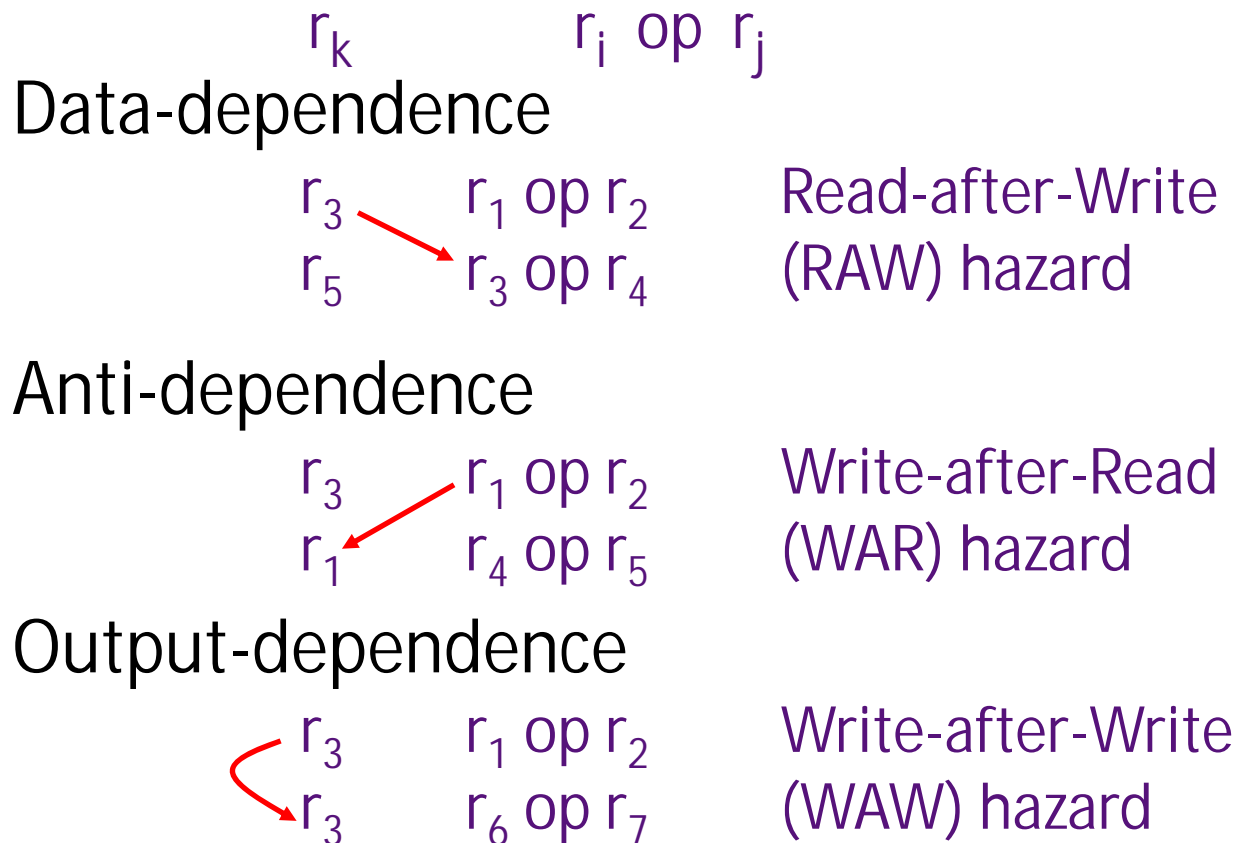


The red lines show data hazards.

This data hazard is called RAW (read after write)

Types of Data Hazards

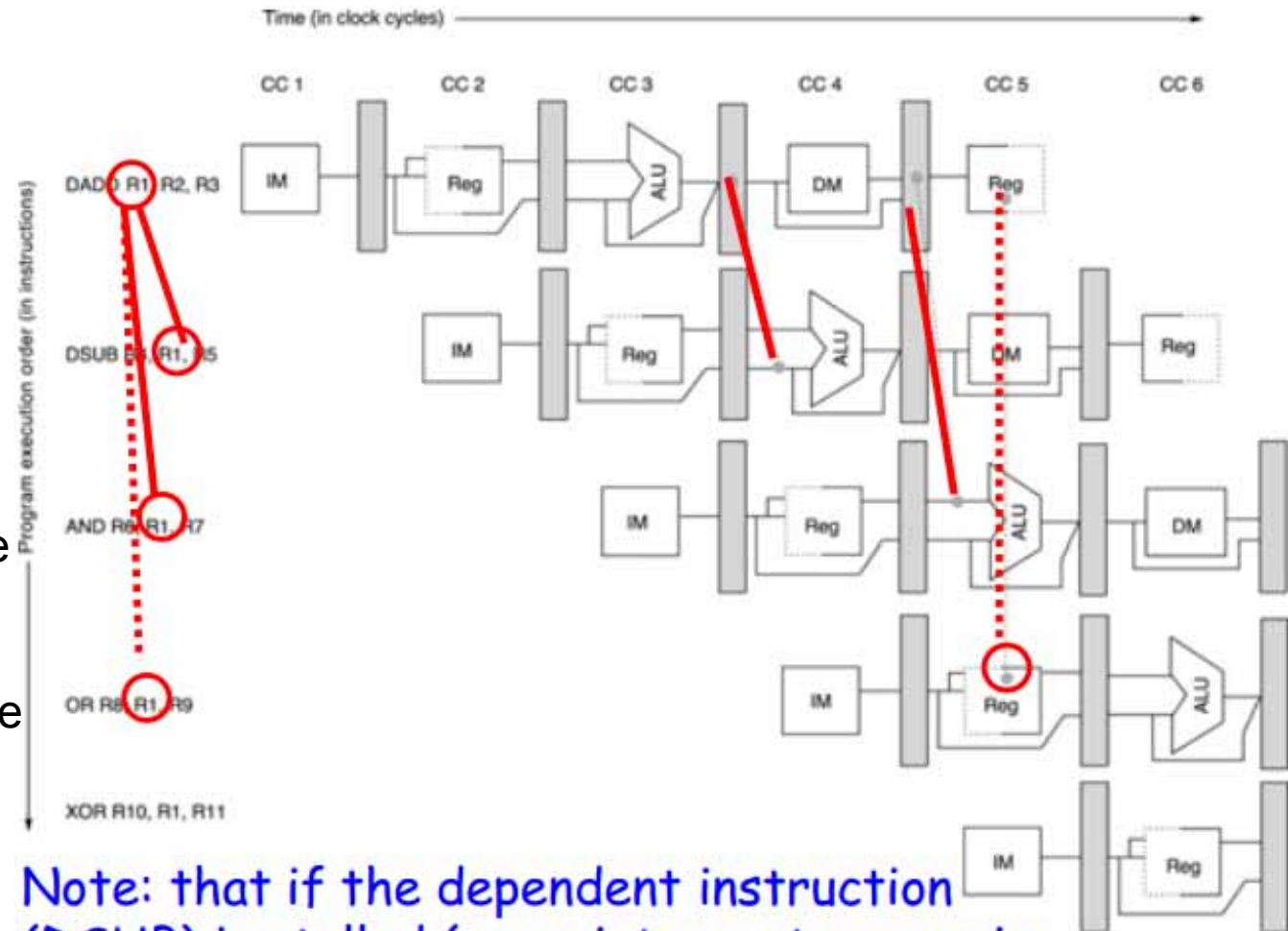
Consider executing a sequence of register-register instructions of type:



Data Hazards: Forwarding (or By-Passing)

- ALU result is always fed back to the ALU input from both EX/MEM and MEM/WB

• If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file.



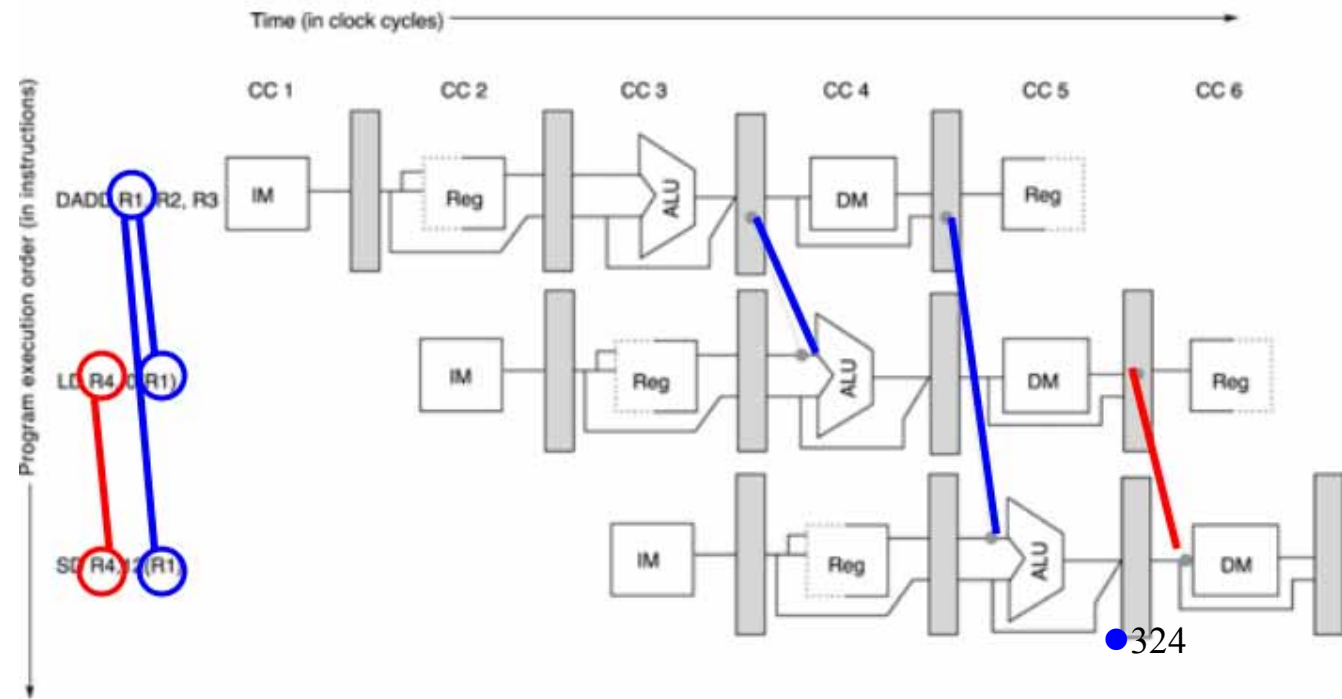
Note: that if the dependent instruction (DSUB) is stalled (or an interrupt occurs in between the two instructions) then the forward path will not be activated

Generalized Forwarding

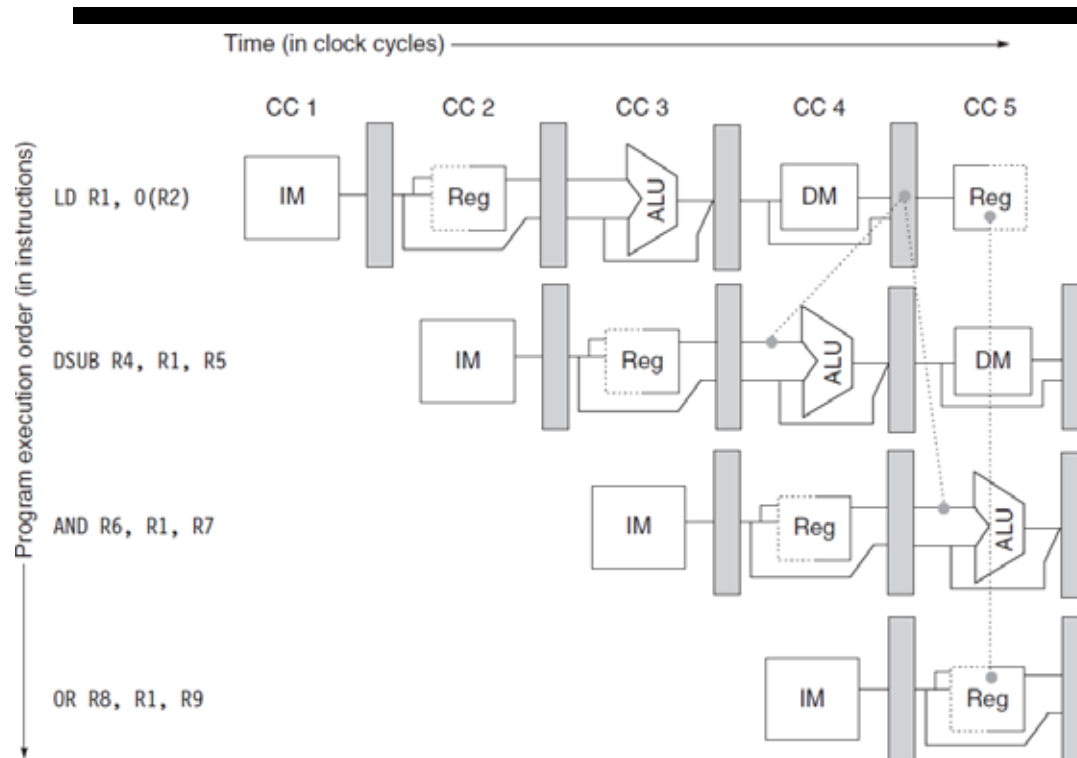
Sometime we need to forward results not only from the immediately previous instruction but also possibly from an instruction started 2 cycles earlier.

Generalized Forwarding: A result can be forwarded from a pipeline register of one unit to the input of another unit.

Note, without forwarding, units only receive operands from its own pipeline register.



Not all Data Hazards can be solved by Forwarding



A pipeline **interlock** (a hardware unit) detects a hazard and stalls the pipeline until the hazard is cleared.

| | | | | | | | | | |
|------|----------|----|----|----|-------|----|-----|-----|--------|
| LD | R1,0(R2) | IF | ID | EX | MEM | WB | | | |
| DSUB | R4,R1,R5 | | IF | ID | stall | EX | MEM | WB | |
| AND | R6,R1,R7 | | | IF | stall | ID | EX | MEM | WB |
| OR | R8,R1,R9 | | | | stall | IF | ID | EX | MEM WB |

Three Strategies for Data Hazards

- Interlock

- Wait for hazard to clear by holding dependent instruction in issue stage

- Bypass

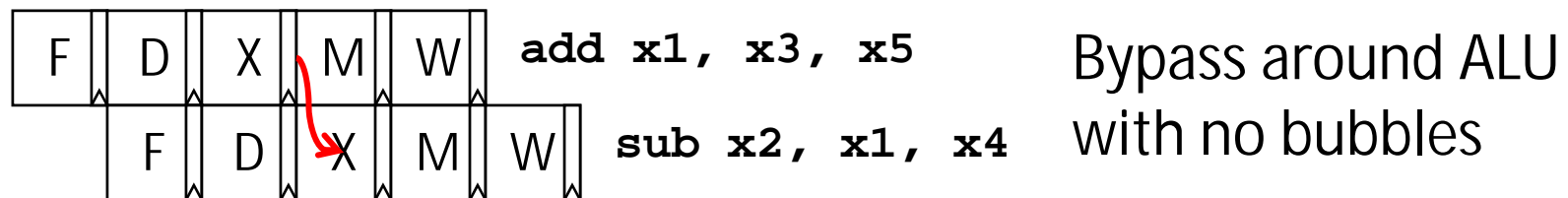
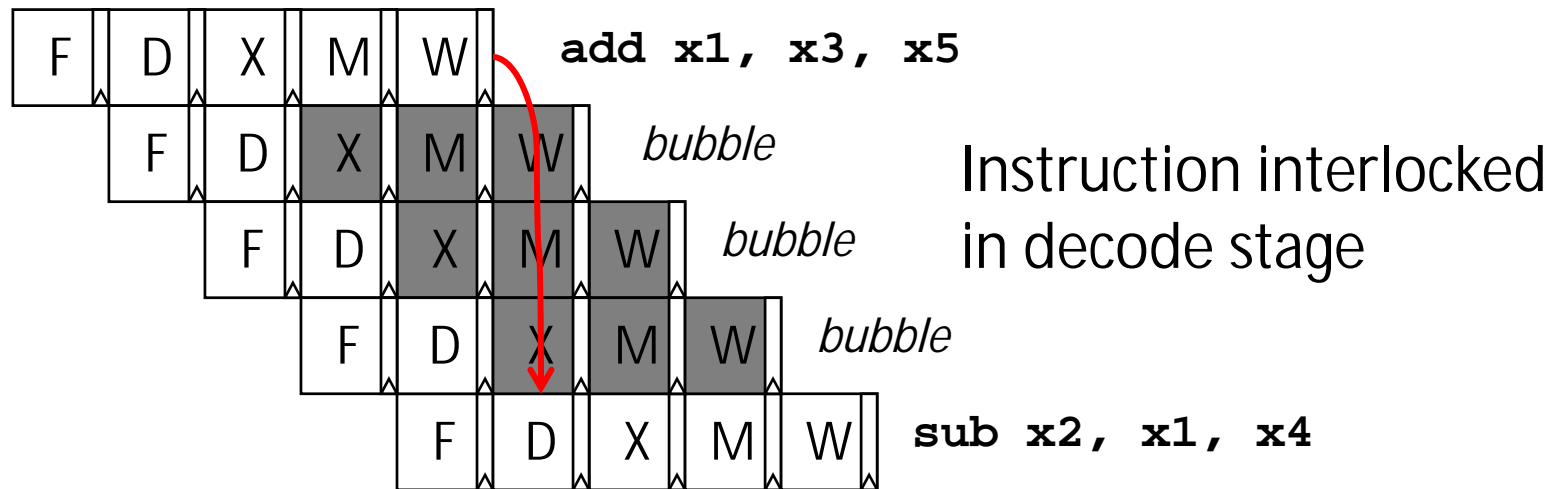
- Resolve hazard earlier by bypassing value as soon as available

- Speculate

- Guess on value, correct if wrong

Interlocking Versus Bypassing

add x1, x3, x5
 sub x2, ~~x1~~, x4



Control / Branch Hazards

Greater Performance Loss than Data Hazards

When a branch is executed, there could be two outcomes:

1. If a branch is taken (**taken branch**), PC will be changed to the branch address at the end of ID after the completion of the address calculation and comparison.
2. Otherwise it is called **untaken branch** or fall through, PC is incremented by 4 at the end of IF, PC + 4

Control Hazard: Where should the next IF get the instruction?

Simple solution: Always do 2 back-to-back IFs in the next instruction execution when a Branch instruction is decoded.

The 1st IF fetches from PC+4, like a stall.

Depending on the outcome of ID of Branch, the 2nd IF either re-fetches the same instruction from PC+4 in case fall through, or fetch the instruction from the address calculated by ID of the previous instruction.

| Branch instruction | IF | ID | EX | MEM | WB | | |
|----------------------|----|----|----|-----|----|-----|-----|
| Branch successor | | IF | IF | ID | EX | MEM | WB |
| Branch successor + 1 | | | | IF | ID | EX | MEM |
| Branch successor + 2 | | | | | IF | ID | EX |

Reducing Pipeline Branch Penalties

One stall cycle for every Branch will yield a performance loss of 10% to 30% depending on the branch frequency!

- Tech. 1: freeze or flush pipeline, like the two back-to-back IF
 - Compiler holds or deletes any instructions after the branch until the branch destination is known
 - Simple in hardware/software, but the penalty is fixed and can't be reduced.
- Tech 2: Guess Branch always Not Taken/Fall Through
 - Always fetch IF from PC+4.
 - In case Guess is right, continue as normal.
 - In case Guess is wrong, since the instruction is at the ID stage, instead of decoding it, change it to a NOOP.
 - Fetch the instruction from the address outcome of Branch instruction.

| | | | | | | | | | |
|----------------------------|----|----|----|-----|-----|-----|-----|-----|----|
| Untaken branch instruction | IF | ID | EX | MEM | WB | | | | |
| Instruction $i + 1$ | | IF | ID | EX | MEM | WB | | | |
| Instruction $i + 2$ | | | IF | ID | EX | MEM | WB | | |
| Instruction $i + 3$ | | | | IF | ID | EX | MEM | WB | |
| Instruction $i + 4$ | | | | | IF | ID | EX | MEM | WB |

| | | | | | | | | | |
|--------------------------|----|----|------|------|------|------|-----|-----|----|
| Taken branch instruction | IF | ID | EX | MEM | WB | | | | |
| Instruction $i + 1$ | | IF | idle | idle | idle | idle | | | |
| Branch target | | | IF | ID | EX | MEM | WB | | |
| Branch target + 1 | | | | IF | ID | EX | MEM | WB | |
| Branch target + 2 | | | | | IF | ID | EX | MEM | WB |

Computer Organization and Design

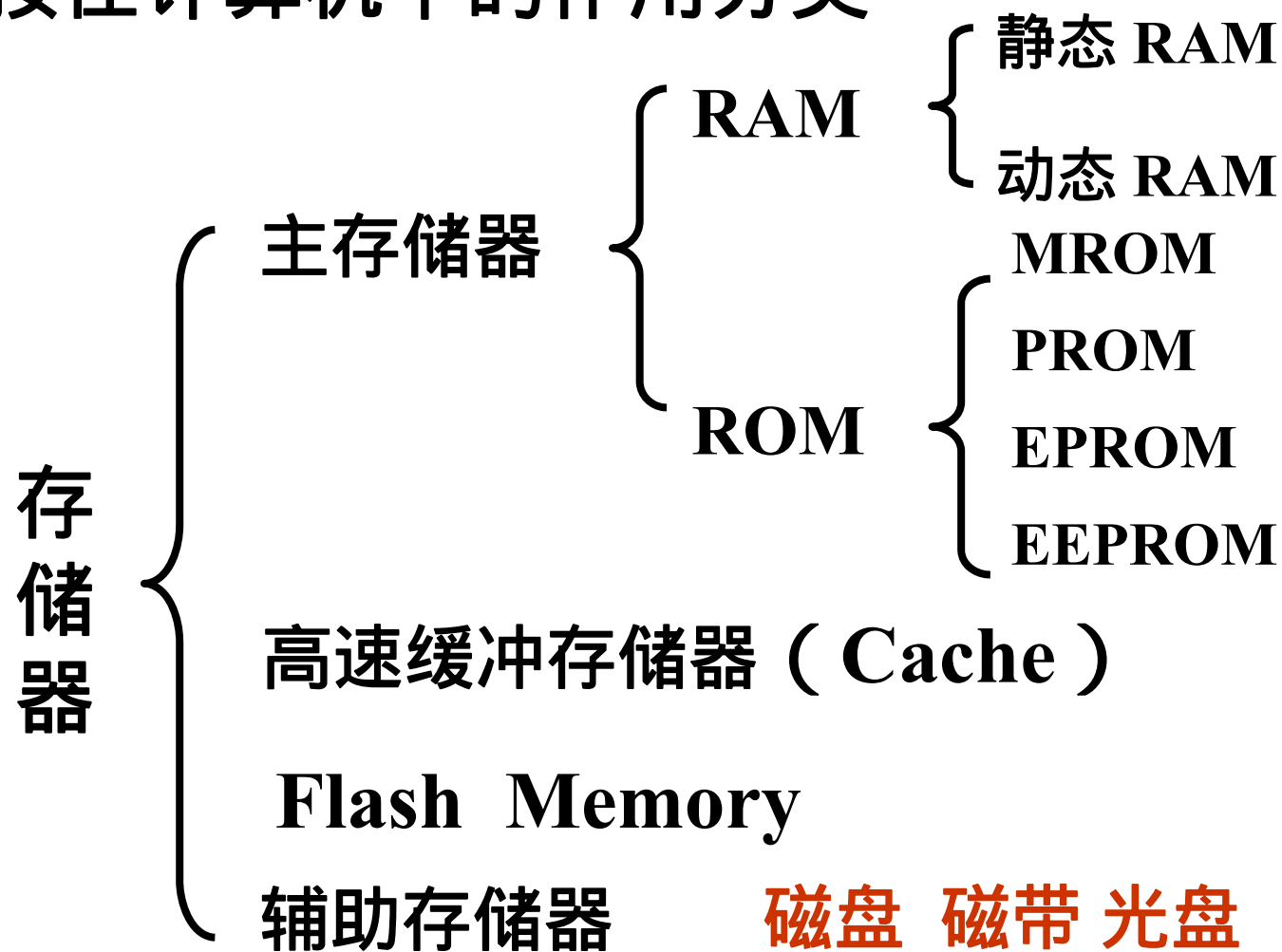
5.1-5.5

A vertical blue line starts from the top left of the slide and extends downwards. A horizontal blue line crosses this vertical line, positioned above the main text. The horizontal line has a gradient, fading from dark blue on the left to light blue on the right.

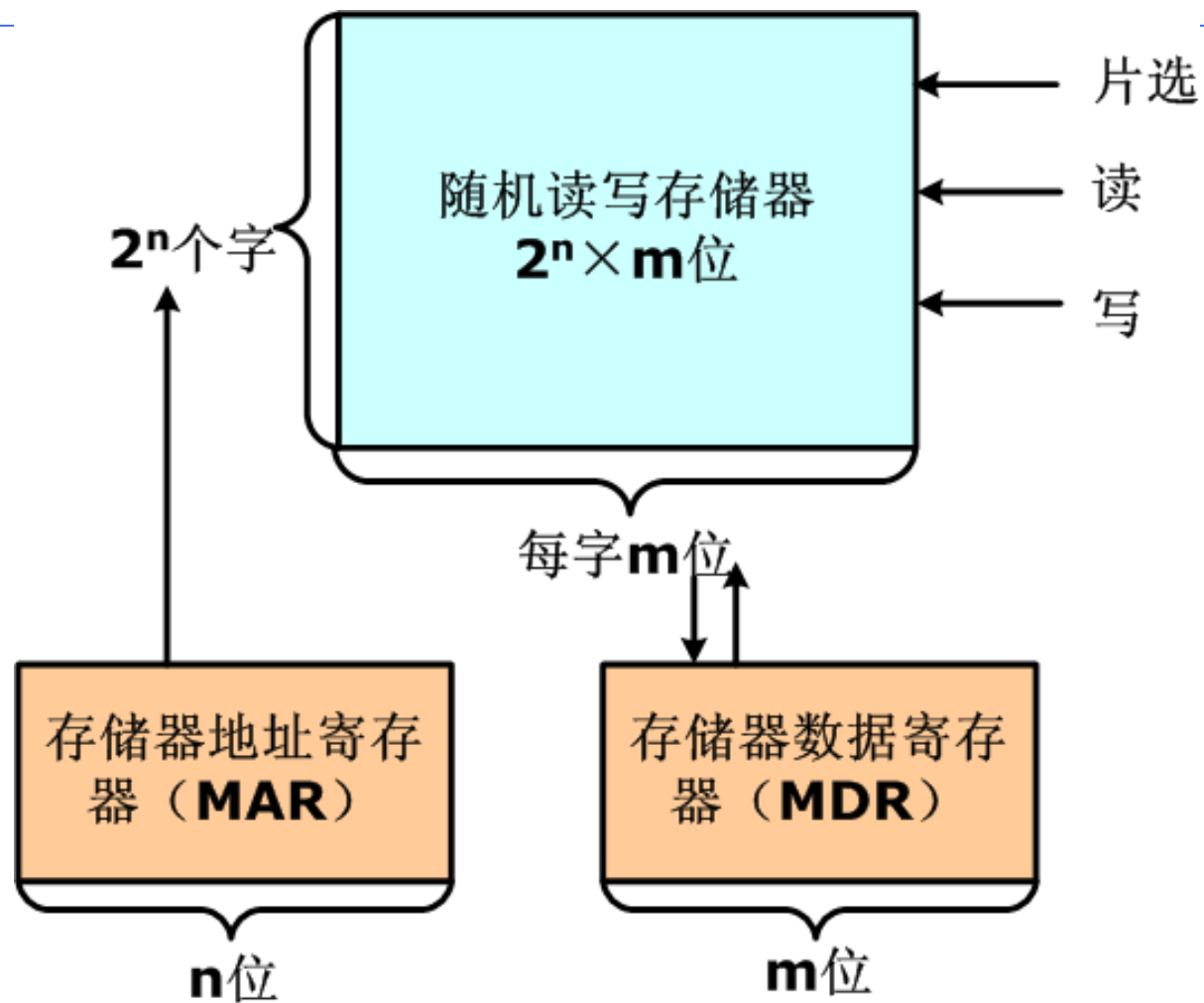
Large and Fast: Exploiting
Memory Hierarchy

小结

3. 按在计算机中的作用分类



一、随机读写存储器RAM

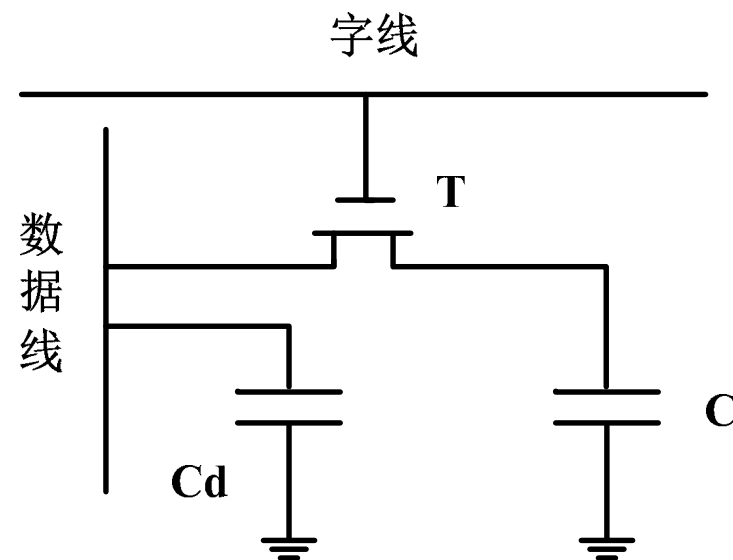


2、动态存储器（DRAM）

- （1）DRAM存储位元
- （2）DRAM存储器
- （3）DRAM的刷新方式
- （4）DRAM存储器的特点

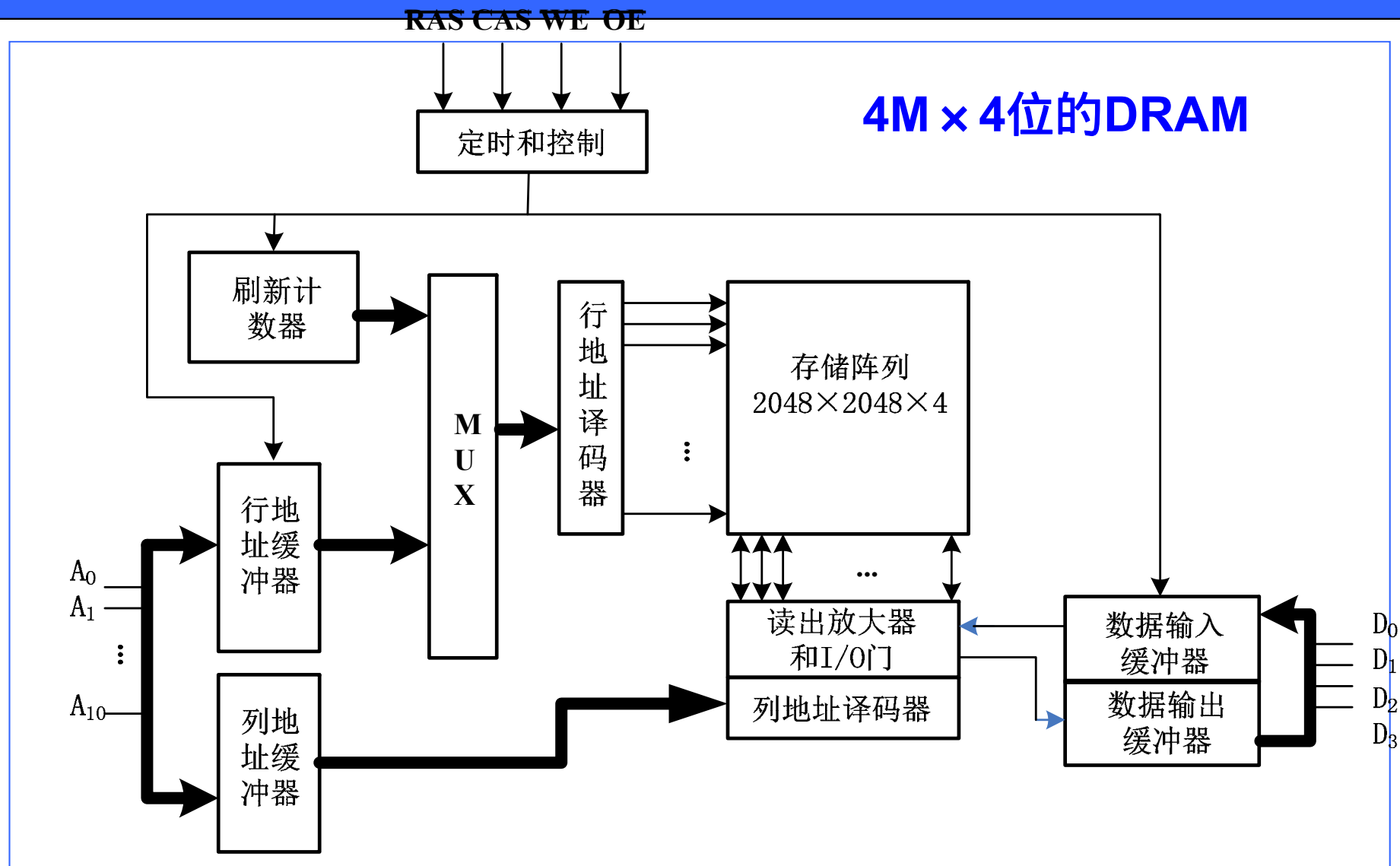
(1) DRAM存储位元

- “1”状态：电容C上有电荷
- “0”状态：电容C上无电荷
- 再生：读出后信息可能被破坏，需要重写。
- 刷新：经过一段时间后信息可能丢失，需要重写。



单管MOS动态存储器结构

(2) DRAM存储器

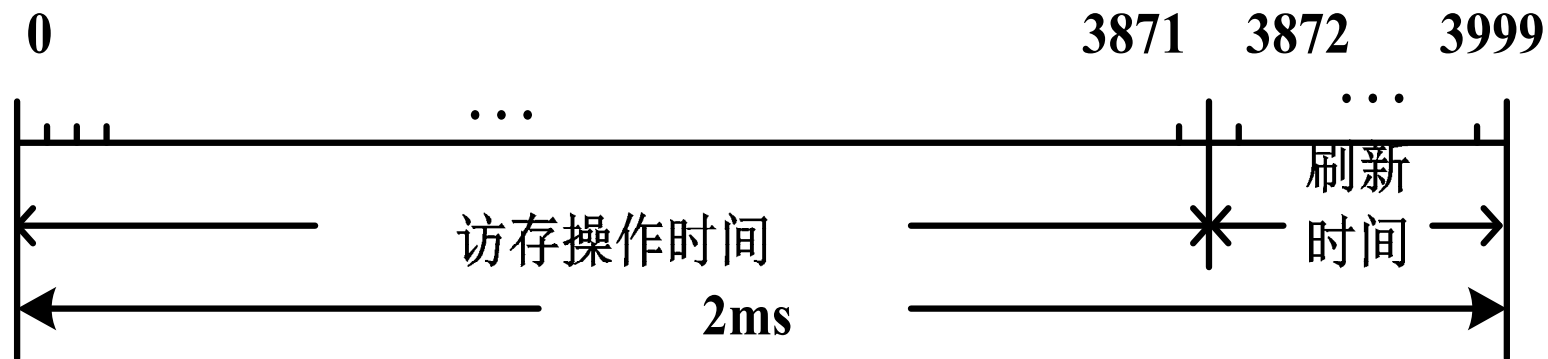


(3) DRAM的刷新方式

- **刷新周期**：从上一次刷新结束到下一次对整个DRAM全部刷新一遍为止，这一段时间间隔称为刷新周期。
- **刷新操作**：即是按行来执行内部的读操作。由刷新计数器产生行地址，选择当前要刷新的行，读即刷新，刷新一行所需时间即是一个存储周期。
- **刷新行数**：单个芯片的单个矩阵的行数。
 - 对于内部包含多个存储矩阵的**芯片**，各个矩阵的同一行是被同时刷新的。
 - 对于**多个芯片**连接构成的DRAM，DRAM控制器将选中所有芯片的同一行来进行逐行刷新。
- **单元刷新间隔时间**：DRAM允许的最大信息保持时间；一般为2ms。
- **刷新方式**：集中式刷新、分散式刷新和异步式刷新。

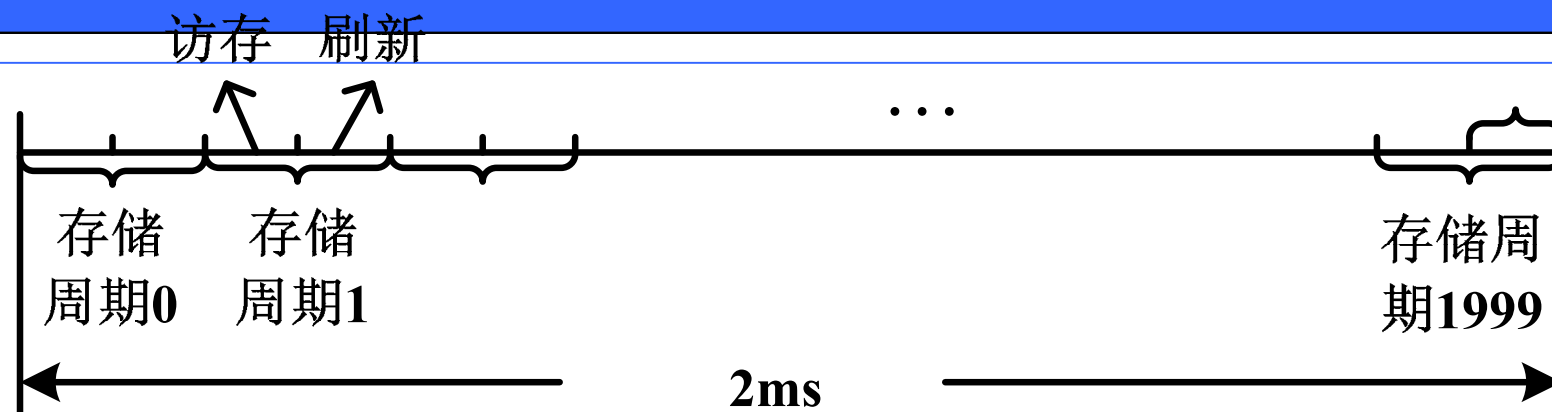
例：64K×1位DRAM芯片中，存储电路由4个独立的
128×128的存储矩阵组成。设存储器存储周期为500ns，单
元刷新间隔是2ms。

⊕集中式刷新



- 在2ms单元刷新间隔时间内，集中对128行刷新一遍，所需时间 $128 \times 500\text{ns} = 64\mu\text{s}$ ，其余时间则用于访问操作。
- 在内部刷新时间（64μs）内，不允许访存，这段时间被称为**死时间**。

✦ 分散式刷新

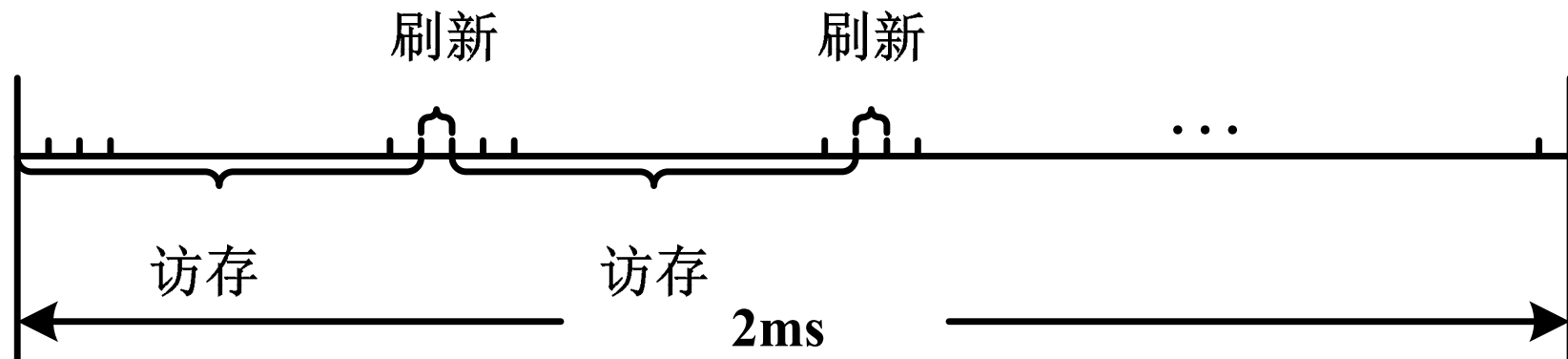


- 在任何一个存储周期内，分为访存和刷新两个子周期。
 - 访存时间内，供CPU和其他主设备访问。
 - 在刷新时间内，对DRAM的某一行刷新。
- 存储周期为存储器存储周期的两倍，即 $500\text{ns} \times 2 = 1\mu\text{s}$ 。
- 刷新周期缩短，为 $128 \times 1\mu\text{s} = 128\mu\text{s}$ 。在2ms的单元刷新闻隔时间内，对DRAM刷新了 $2\text{ms} \div 128\mu\text{s}$ 遍。

异步式刷新

异步刷新采取折中的办法，在2ms内分散地把各行刷新一遍。

- 避免了分散式刷新中不必要的多次刷新，提高了整机速度；同时又解决了集中式刷新中“死区”时间过长的问题。
- 刷新信号的周期为 $2\text{ms}/128=15.625\mu\text{s}$ 。让刷新电路每隔 $15\mu\text{s}$ 产生一个刷新信号，刷新一行。

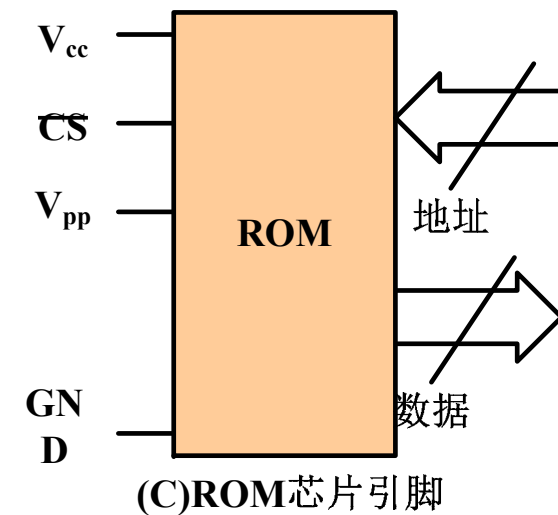
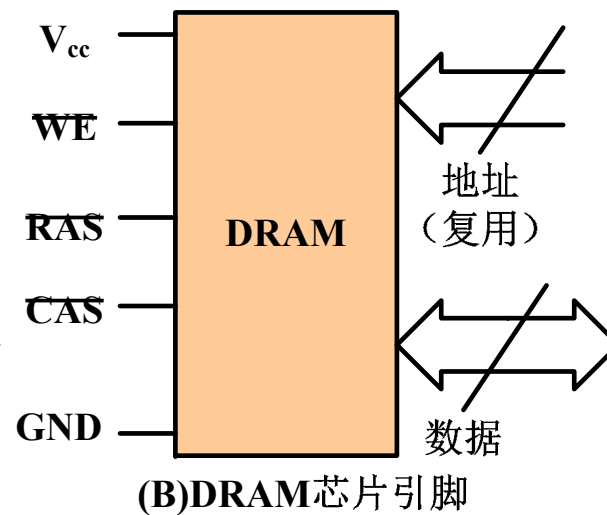
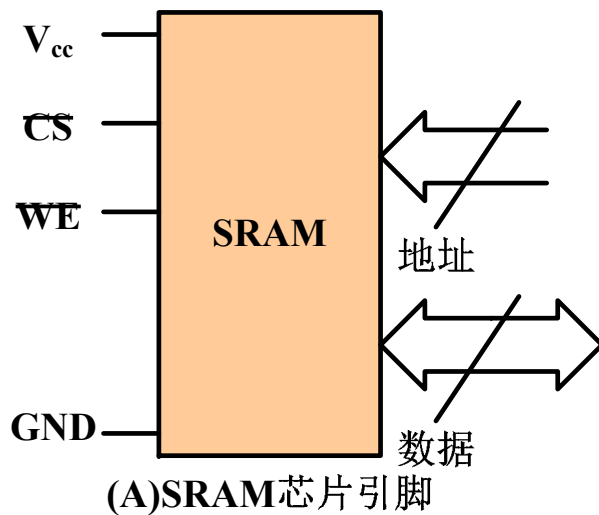


1 主存储器与CPU的连接

- 一、背景知识——存储芯片简介
- 二、存储器容量扩展的三种方法
- 三、主存储器与CPU的连接

一、背景知识——存储芯片简介

□ 存储芯片的引脚封装



二、存储器容量扩展的三种方法

□ 1、位扩展

从字长方向扩展

□ 2、字扩展

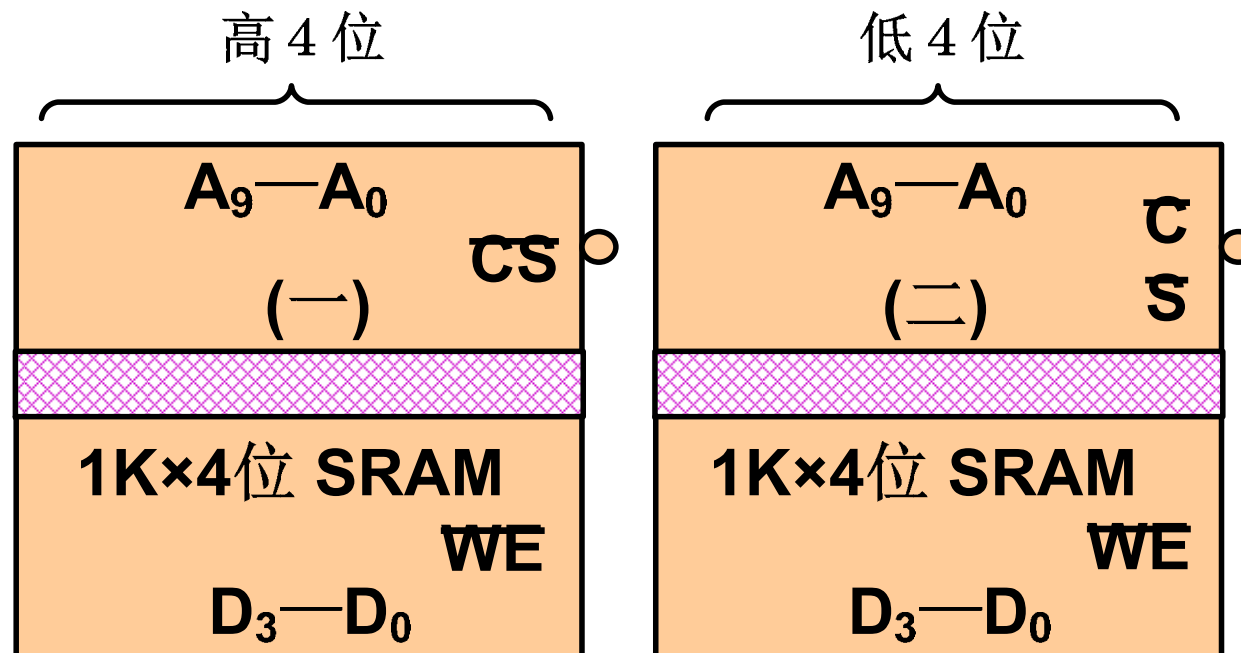
从字数方向扩展

□ 3、字位扩展

从字长和字数方向扩展

1、位扩展

□ 要求：用 $1K \times 4$ 位的SRAM芯片 \rightarrow $1K \times 8$ 位的SRAM存储器

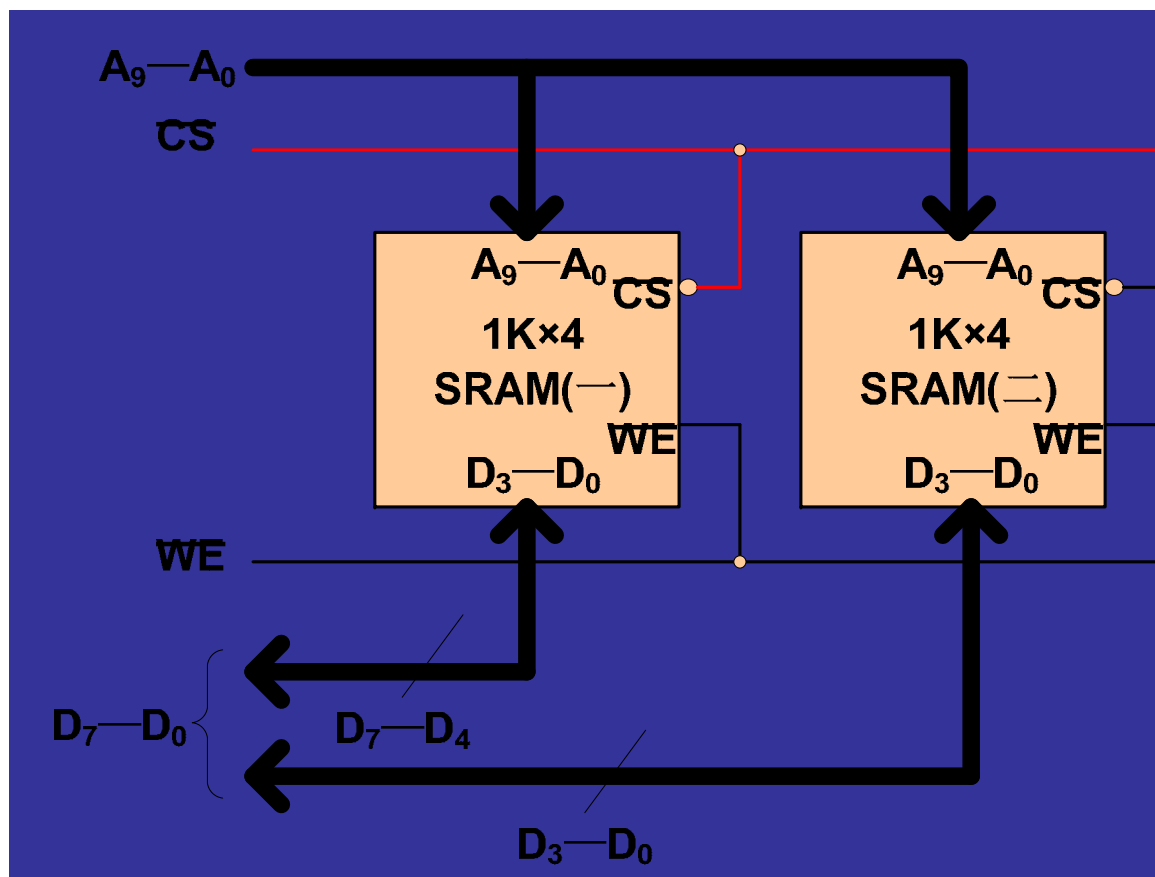


1、位扩展

□ 容量 = $2^{10} \times 8$ 位

□ 举例验证:

读地址为0 的
存储单元的内
容



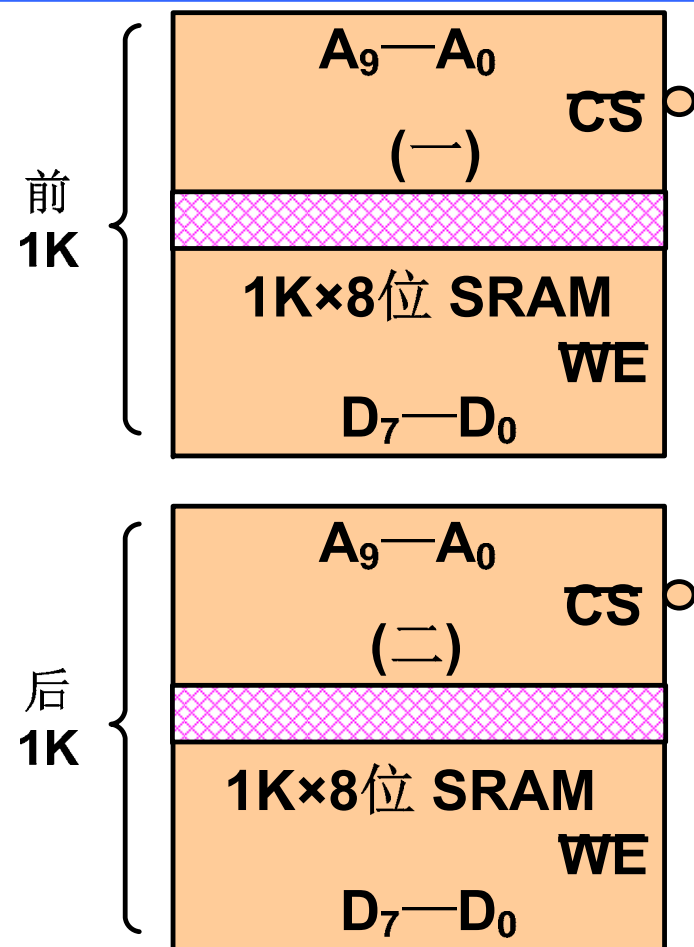
1、位扩展

- 要点：
- （1）芯片的地址线A、读写控制信号WE#、片选信号CS#分别连在一起；
- （2）芯片的数据线D分别对应于所搭建的存储器的高若干位和低若干位。

2、字扩展

□要求：

用 $1K \times 8$ 位的SRAM芯片
→ $2K \times 8$ 位的SRAM存储器



2、字扩展

□ 分析地址：

- A_{10} 用于选择芯片
- $A_9 \sim A_0$ 用于选择芯片内的某一存储单元

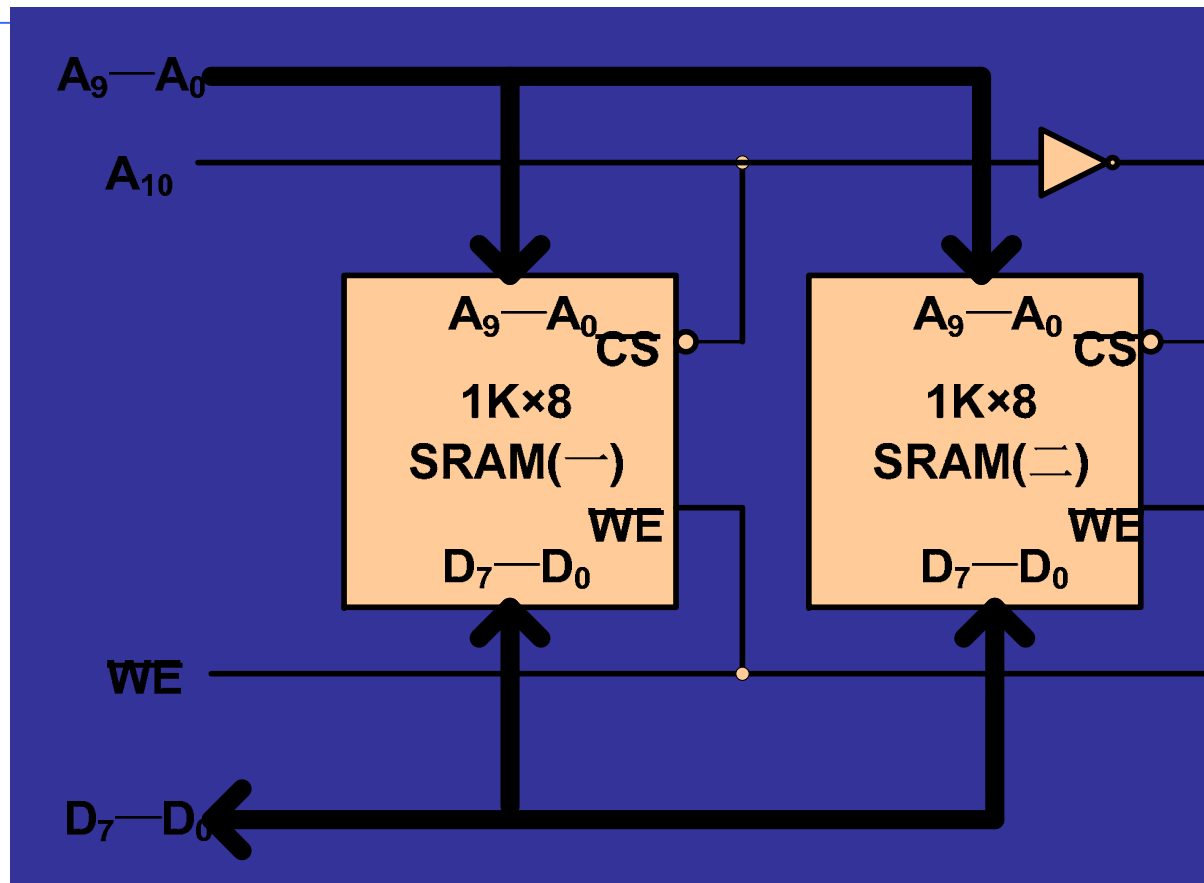
| A_{10} | A_9 | ~ | A_0 | |
|----------|-------|---|-------|---------|
| 0 | 0 | ~ | 0 | 前 1K |
| ⋮ | ⋮ | ⋮ | ⋮ | |
| 0 | 1 | ~ | 1 | |
| 1 | 0 | ~ | 0 | 后 1K |
| ⋮ | ⋮ | ⋮ | ⋮ | |
| 1 | 1 | ~ | 1 | |

2、字扩展

□容量 = $2^{11} \times 8$ 位

□举例验证:

- 读地址为 0 的存储单元的内容
- 读地址为 10 ... 0 的存储单元的内容



2、字扩展

- 要点：
- (1) 芯片的数据线D、读写控制信号WE#分别连在一起；
- (2) 存储器地址线A的低若干位连接各芯片的地址线；
- (3) 存储器地址线A的高若干位作用于各芯片的片选信号CS#。

3、字位扩展

- 需扩展的存储器容量为 $M \times N$ 位，已有芯片的容量为 $L \times K$ 位 ($L < M, K < N$)

$$\frac{M \times N}{L \times K}$$

- 用 M/L 组 芯片进行字扩展；
- 每组内有 N/K 个 芯片进行位扩展。

三、主存储器与CPU的连接

- 1、根据CPU芯片提供的地址线数目，确定CPU访存的地址范围，并写出相应的二进制地址码；
- 2、根据地址范围的容量，确定各种类型存储器芯片的数目和扩展方法；
- 3、分配CPU地址线。**CPU地址线的低位**（数量 = 存储芯片的地址线数量）直接连接**存储芯片的地址线**；**CPU高位地址线**皆参与形成存储芯片的**片选信号**；
- 4、连接数据线、R/W#等其他信号线，MREQ#信号一般可用作地址译码器的使能信号。
- 需要说明的是，主存的扩展及与CPU连接在做法上并不唯一，应该具体问题具体分析

例

- **例：**设CPU有16根地址线，8根数据线，并用MREQ#作访存控制信号（低电平有效），用R/W#作读/写控制信号（高电平为读，低电平为写）。现有下列存储芯片：1K*4位SRAM；4K*8位SRAM；8K*8位SRAM；2K*8位ROM；4K*8位ROM；8K*8位ROM；及3：8译码器和各种门电路。
- **要求：**主存的地址空间满足下述条件：最小8K地址为系统程序区（ROM区），与其相邻的16K地址为用户程序区（RAM区），最大4K地址空间为系统程序区（ROM区）。
- 请画出存储芯片的片选逻辑，存储芯片的种类、片数
- 画出CPU与存储器的连接图。

解：首先根据题目的地址范围写出相应的二进制地址码。

| A ₁₅ | A ₁₄ | A ₁₃ | A ₁₂ | A ₁₁ | A ₁₀ | A ₉ | A ₈ | A ₇ | A ₆ | A ₅ | A ₄ | A ₃ | A ₂ | A ₁ | A ₀ | |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|--------------------------|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | } 最小8K 系统区 |
| | | ... | ... | | | | | | | | | | | | | |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | } 相邻 16K 用户程 序区 |
| | | ... | ... | | | | | | | | | | | | | |
| 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | | ... | ... | | | | | | | | | | | | | |
| 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |
| | | ... | ... | | | | | | | | | | | | | } 最大4K 系统区 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | | ... | ... | | | | | | | | | | | | | |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | |

解题

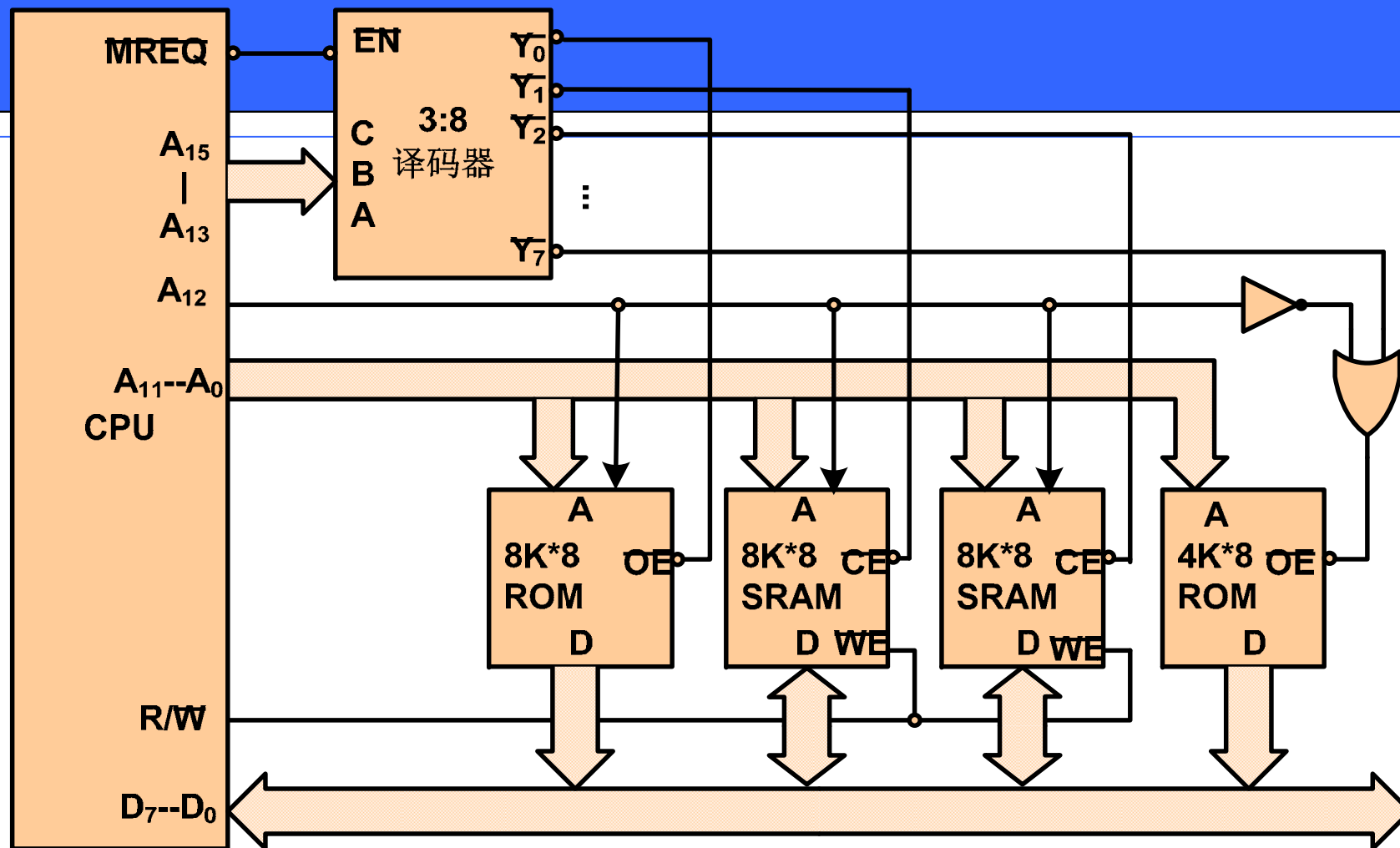
□ 第二步：选择芯片

- 最小8K系统程序区 \leftarrow 8K*8位ROM，1片
- 16K用户程序区 \leftarrow 8K*8位SRAM，2片；
- 4K系统程序工作区 \leftarrow 4K*8位ROM，1片。

□ 第三步，分配CPU地址线。

- CPU的低13位地址线 $A_{12}\sim A_0$ 与1片8K*8位ROM和两片8K*8位SRAM芯片提供的地址线相连；将CPU的低12位地址线 $A_{11}\sim A_0$ 与1片4K*8位ROM芯片提供的地址线相连。

□ 第四步，译码产生片选信号。



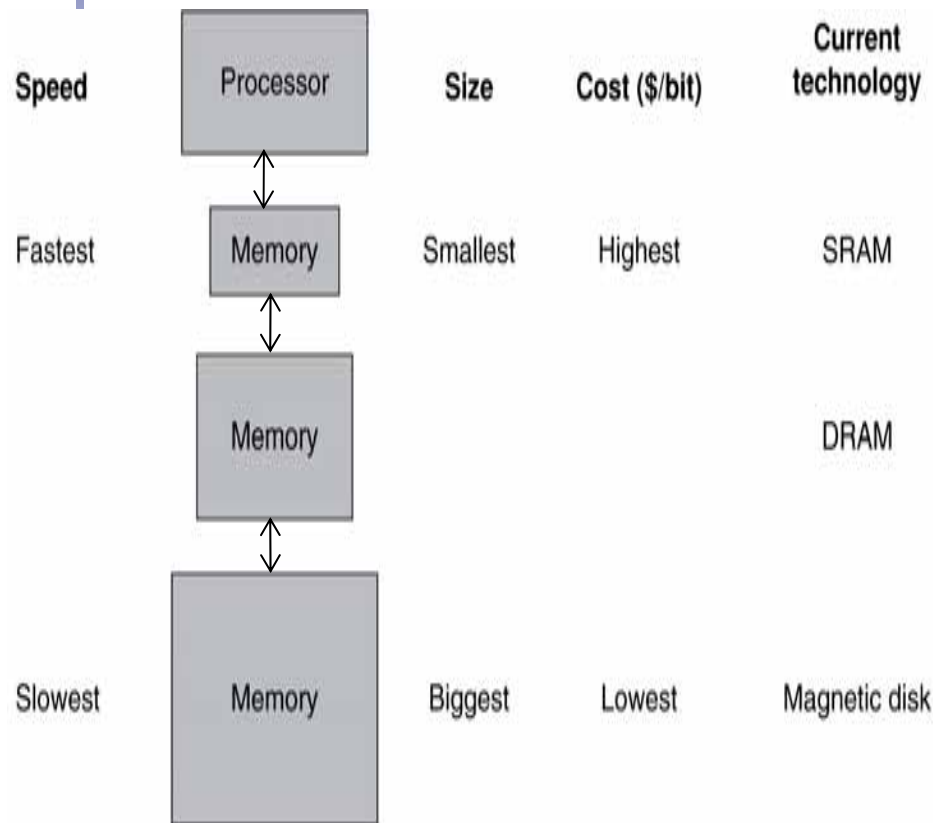
Principle of Locality

- Programs access a small proportion of their address space at any time
- **Temporal locality**
 - Items accessed recently are likely to be accessed again soon
 - e.g., instructions in a loop, induction variables
- **Spatial locality**
 - Items near those accessed recently are likely to be accessed soon
 - e.g., sequential instruction access, array data

Taking Advantage of Locality

- Memory hierarchy
- Store everything on disk
- Copy recently accessed (and nearby) items from disk to smaller DRAM memory
 - Main memory
- Copy more recently accessed (and nearby) items from DRAM to smaller SRAM memory
 - Cache memory attached to CPU

Memory Hierarchy Levels



- Block : unit of copying
 - May be multiple words
- If accessed data is present in upper level
 - **Hit**: access satisfied by upper level
 - Hit ratio: hits/accesses
- If accessed data is absent
 - **Miss**: block copied from lower level
 - Time taken: miss penalty
 - Miss ratio: misses/accesses = $1 - \text{hit ratio}$
 - Then accessed data supplied from upper level

Cache Memory

- Cache memory
 - The level of the memory hierarchy closest to the CPU
- Given accesses X_1, \dots, X_{n-1}, X_n

| |
|-----------|
| X_4 |
| X_1 |
| X_{n-2} |
| |
| X_{n-1} |
| X_2 |
| |
| X_3 |

a. Before the reference to X_n

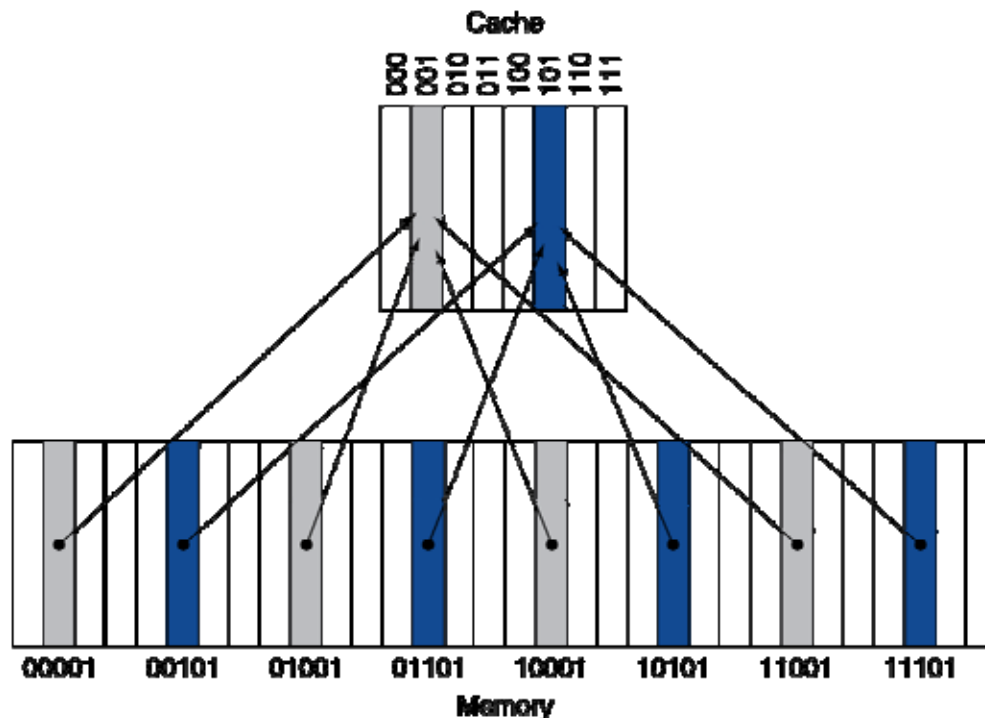
| |
|-----------|
| X_4 |
| X_1 |
| X_{n-2} |
| |
| X_{n-1} |
| X_2 |
| X_n |
| X_3 |

b. After the reference to X_n

- How do we know if the data is present?
- Where do we look?

Direct Mapped Cache

- Location determined by **address**
- **Direct mapped**: only one choice
 - (Block address) modulo (#Blocks in cache)



- #Blocks is a power of 2
- Use low-order address bits for block addr

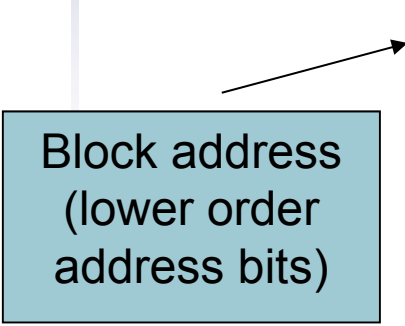
Tags and Valid Bits

- How do we know which particular block is stored in a cache location?
 - Store block address as well as the data
 - Actually, only need the high-order bits
 - Called the **tag**
- What if there is no data in a location?
 - **Valid bit**: 1 = present, 0 = not present
 - Initially 0

Cache Example

- 8-blocks, 1 word/block, direct mapped
- Initial state

Block address
(lower order
address bits)



| Index | V | Tag | Data |
|-------|---|-----|------|
| 000 | N | | |
| 001 | N | | |
| 010 | N | | |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | N | | |
| 111 | N | | |

Cache Example

| Word addr | Binary addr | Hit/miss | Cache block |
|-----------|-------------|----------|-------------|
| 22 | 10 110 | Miss | 110 |

| Index | V | Tag | Data |
|------------|----------|-----------|-------------------|
| 000 | N | | |
| 001 | N | | |
| 010 | N | | |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

Cache Example

| Word addr | Binary addr | Hit/miss | Cache block |
|-----------|-------------|----------|-------------|
| 26 | 11 010 | Miss | 010 |

| Index | V | Tag | Data |
|------------|----------|-----------|-------------------|
| 000 | N | | |
| 001 | N | | |
| 010 | Y | 11 | Mem[11010] |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

Cache Example

| Word addr | Binary addr | Hit/miss | Cache block |
|-----------|-------------|----------|-------------|
| 22 | 10 110 | Hit | 110 |
| 26 | 11 010 | Hit | 010 |

| Index | V | Tag | Data |
|-------|---|-----|------------|
| 000 | N | | |
| 001 | N | | |
| 010 | Y | 11 | Mem[11010] |
| 011 | N | | |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

Cache Example

| Word addr | Binary addr | Hit/miss | Cache block |
|-----------|-------------|----------|-------------|
| 16 | 10 000 | Miss | 000 |
| 3 | 00 011 | Miss | 011 |
| 16 | 10 000 | Hit | 000 |

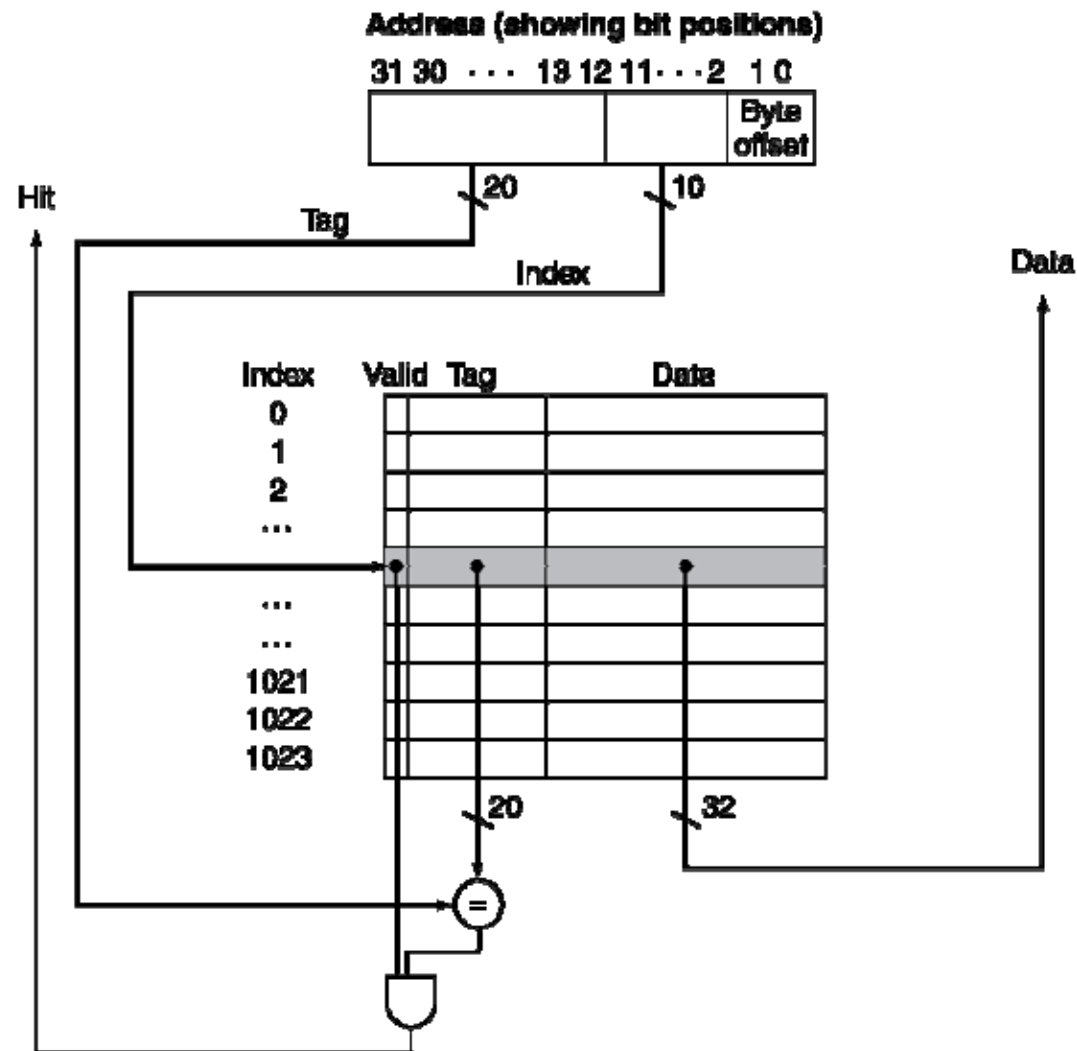
| Index | V | Tag | Data |
|------------|----------|-----------|-------------------|
| 000 | Y | 10 | Mem[10000] |
| 001 | N | | |
| 010 | Y | 11 | Mem[11010] |
| 011 | Y | 00 | Mem[00011] |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

Cache Example

| Word addr | Binary addr | Hit/miss | Cache block |
|-----------|-------------|----------|-------------|
| 18 | 10 010 | Miss | 010 |

| Index | V | Tag | Data |
|------------|----------|-----------|-------------------|
| 000 | Y | 10 | Mem[10000] |
| 001 | N | | |
| 010 | Y | 10 | Mem[10010] |
| 011 | Y | 00 | Mem[00011] |
| 100 | N | | |
| 101 | N | | |
| 110 | Y | 10 | Mem[10110] |
| 111 | N | | |

Address Subdivision



Cache Field Sizes

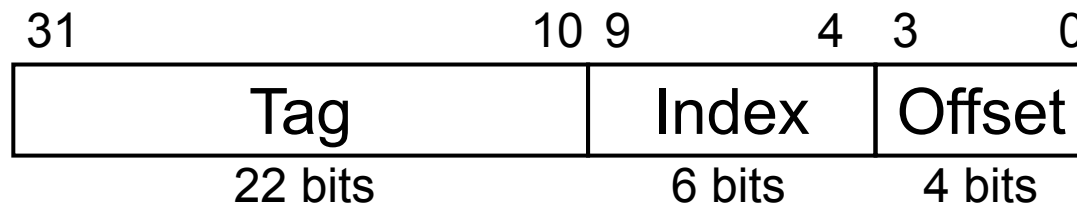
- The number of bits in a cache includes both the storage for data and for the tags
 - 32-bit byte address
 - For a direct mapped cache with 2^n blocks, n bits are used for the **index**
 - For a block size of 2^m words (2^{m+2} bytes), m bits are used to address the word within the block and 2 bits are used to address the byte within the word
- What is the size of the **tag** field?
 - $32 - (n+m+2)$
- The total number of bits in a direct-mapped cache is then $2^n \times (\text{block size} + \text{tag field size} + \text{valid field size})$

Cache Field Sizes

- How many total bits are required for a direct mapped cache with 16KB of data and 4-word blocks assuming a 32-bit address?
 - 16KB is **4K (2^{12}) words**
 - With a block size of 4 words, there are **1024 (2^{10}) blocks**
 - Each block has 4x32 or 128 bits of data plus a tag which is 32 – (10 + 2 + 2) = 18 bits, plus a valid bit = **19 bits**
 - So the total cache size is
$$2^{10} \times (4 \times 32 + 18 + 1) = 2^{10} \times 147 = \text{147Kbits} = 18.375\text{KB}$$
 - or about **1.15x** as many as needed just for storage of the data/存储16KB数据的cache需要18.375KB的容量，是数据存储量的1.15倍

Example: Larger Block Size

- 64 blocks, 16 bytes/block
 - To what block number does address 1200 map?
 - Block no. = (Block addr) modulo (#Blocks in cache)
- Block address = Byte addr/Bytes per block
 $= \lfloor 1200/16 \rfloor = 75$
- Block number = 75 modulo 64 = 11/1200-1215之间所有地址都映射于该块



Block Size Considerations

- Larger blocks should reduce miss rate

- Due to spatial locality

- But in a fixed-sized cache

- Larger blocks \Rightarrow fewer of them

- More competition \Rightarrow increased miss rate

- Larger blocks \Rightarrow pollution

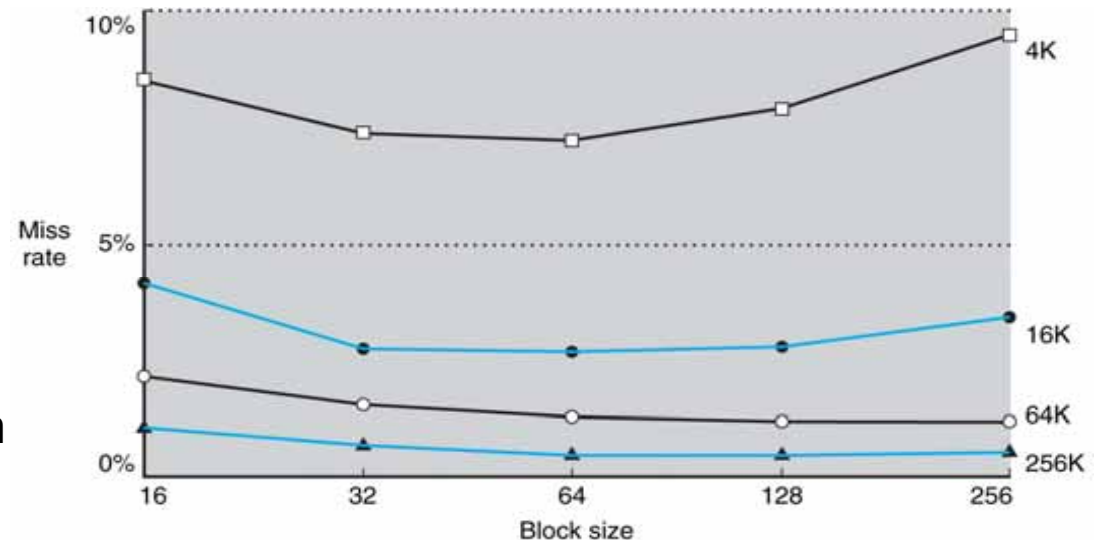
- Larger miss penalty

- Larger blocks \Rightarrow more time required to fetch

- Can override benefit of reduced miss rate

- Early restart and critical-word-first can help

- **Early restart:** resume execution as soon as requested word in block is returned
 - Rather than wait for entire block to be returned
- **Critical word first:** organize memory so that requested word is transferred first
 - Remainder of block transferred afterwards



Cache Misses

- On cache hit, CPU proceeds normally
- On cache miss
 - Stall the CPU pipeline-流水线阻塞
 - Instead of interrupts which require saving context
 - Fetch block from next level of hierarchy
 - Instruction cache miss
 - Restart instruction fetch
 - Data cache miss
 - Complete data access

Handling Cache Writes

- On data-write hit, could just update the block in cache
 - But then cache and memory would be inconsistent
- **Write through:** also update memory
- But makes writes take longer
 - e.g., if base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
 - Effective CPI = $1 + 0.1 \times 100 = 11$
- Solution: **write buffer**
 - Holds data waiting to be written to memory
 - CPU continues immediately
 - Only stalls on write if write buffer is already full

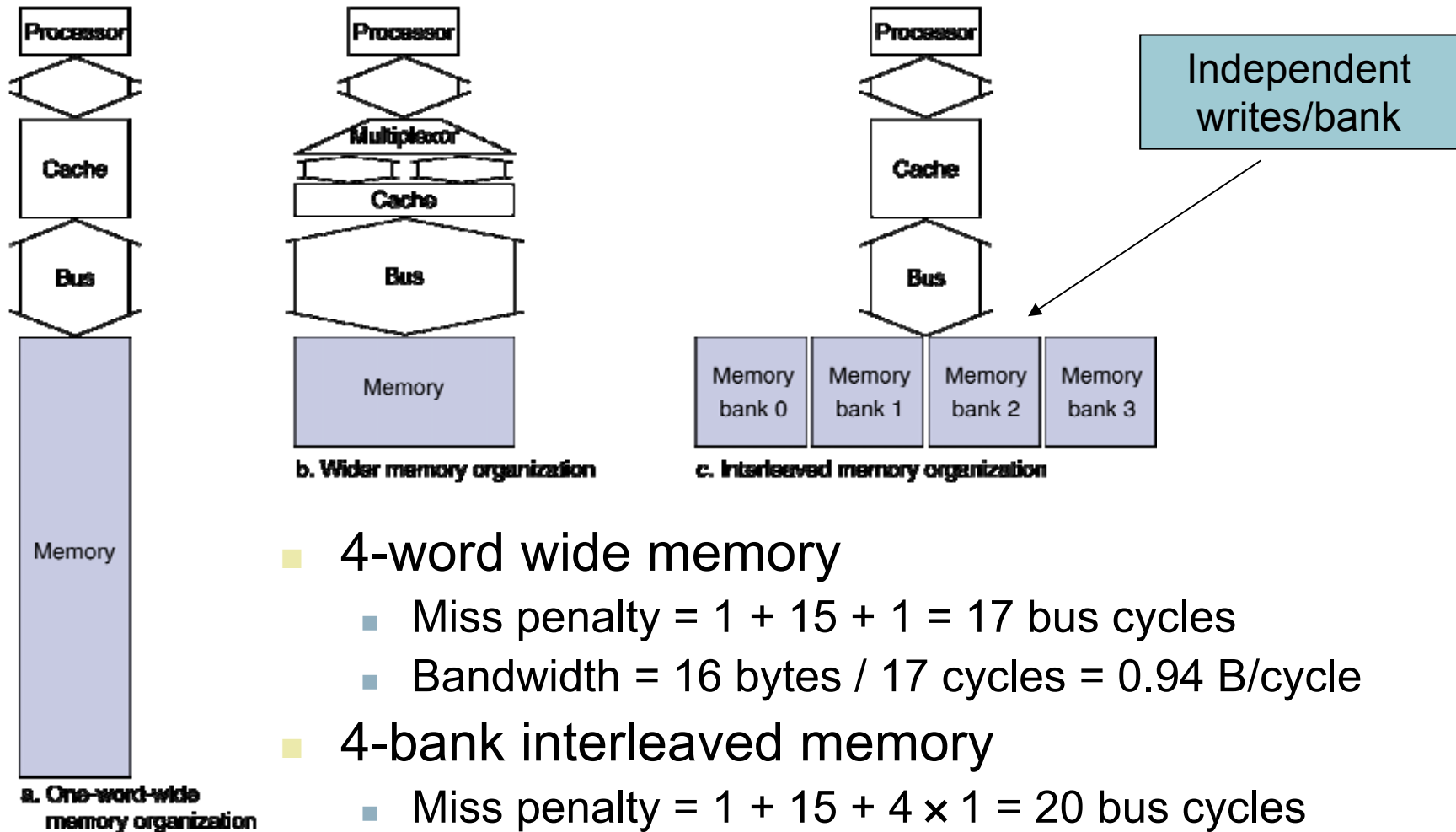
Write-Back

- Alternative Solution: On data-write hit, just update the block in cache
 - Keep track of whether each block is **dirty**(被修改的位)
- When a **dirty** block is replaced
 - Write it back to memory
 - Can use a write buffer to allow replacing block to be read first
 - without waiting for the block to be written into memory first

Main Memory Supporting Caches

- Use DRAMs for main memory
 - Fixed width (e.g., 1 word)
 - Connected by fixed-width clocked bus
 - Bus clock is typically slower than CPU clock
- Example cache block read
 - 1 bus cycle for address transfer
 - 15 bus cycles per DRAM access
 - 1 bus cycle per data word transfer
- For 4-word block, 1-word-wide DRAM
 - Miss penalty = $1 + 4 \times 15 + 4 \times 1 = 65$ bus cycles
 - Bandwidth = $16 \text{ bytes} / 65 \text{ cycles} = 0.25 \text{ B/cycle}$
 - 1 GHz clocked DRAM would have 250 MB/sec bandwidth

Increasing Memory Bandwidth



- 4-word wide memory
 - Miss penalty = $1 + 15 + 1 = 17$ bus cycles
 - Bandwidth = $16 \text{ bytes} / 17 \text{ cycles} = 0.94 \text{ B/cycle}$
- 4-bank interleaved memory
 - Miss penalty = $1 + 15 + 4 \times 1 = 20$ bus cycles
 - Bandwidth = $16 \text{ bytes} / 20 \text{ cycles} = 0.8 \text{ B/cycle}$

Advanced DRAM Organization

- DRAMs include clocks to eliminate memory-processor synchronization time
 - Synchronized DRAM (SDRAM)
- DRAM bits are organized as a rectangular array
 - DRAM accesses (and buffers) an entire row
 - Burst mode/突发传输: supply successive words from a row (buffer) with reduced latency
- Double data rate (DDR) DRAM
 - Transfer on rising and falling clock edges
- Quad data rate (QDR) DRAM
 - Separate DDR inputs and outputs

Measuring Cache Performance

- Components of CPU time
 - Program execution cycles
 - Includes cache hit time
 - Memory stall cycles
 - Mainly from cache misses
- With simplifying assumptions:
 - Write buffer stalls negligible

Memory stall cycles

$$= \frac{\text{Memory accesses}}{\text{Program}} \times \text{Miss rate} \times \text{Miss penalty}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}$$

Cache Performance Example

- Given
 - I-cache miss rate = 2%
 - D-cache miss rate = 4%
 - Miss penalty = 100 cycles
 - Base CPI (without memory stalls) = 2
 - Load & stores are 36% of instructions
- How much faster would a processor run with a perfect cache that never missed?
- Miss cycles per instruction
 - I-cache: $0.02 \times 100 = 2$
 - D-cache: $0.36 \times 0.04 \times 100 = 1.44$
- Actual CPI = $2 + 2 + 1.44 = 5.44$
 - Ideal CPU is $5.44/2 = 2.72$ times faster

Cache Performance Example

- What if we speed-up the computer by reducing the CPI_{ideal} to 1, without changing clock rate?
 - Actual $CPI = 1 + 3.44 = 4.44$
 - Ideal CPU is $4.44/1 = 4.44$ times faster
 - Amount of execution time spent on memory stalls rises from $3.44/5.44 = 63\%$ to $3.44/4.44 = 77\%$
- What if we increased clock rate?
 - Similar increase in performance lost due to cache misses
- Remember Amdahl's law!

Average Access Time

- **Hit time** is also important for performance
- Average memory access time (**AMAT**)
 - $AMAT = \text{Hit time} + \text{Miss rate} \times \text{Miss penalty}$
- Example
 - CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, miss rate/instruction = 5%
 - $AMAT = 1 + 0.05 \times 20 = 2\text{ns}$
 - 2 cycles per instruction

Performance Summary

- When CPU performance increased
 - Miss penalty becomes more significant
- Decreasing base CPI
 - Greater proportion of time spent on memory stalls
- Increasing clock rate
 - Memory stalls account for more CPU cycles
- Can't neglect cache behavior when evaluating system performance

Associative Caches

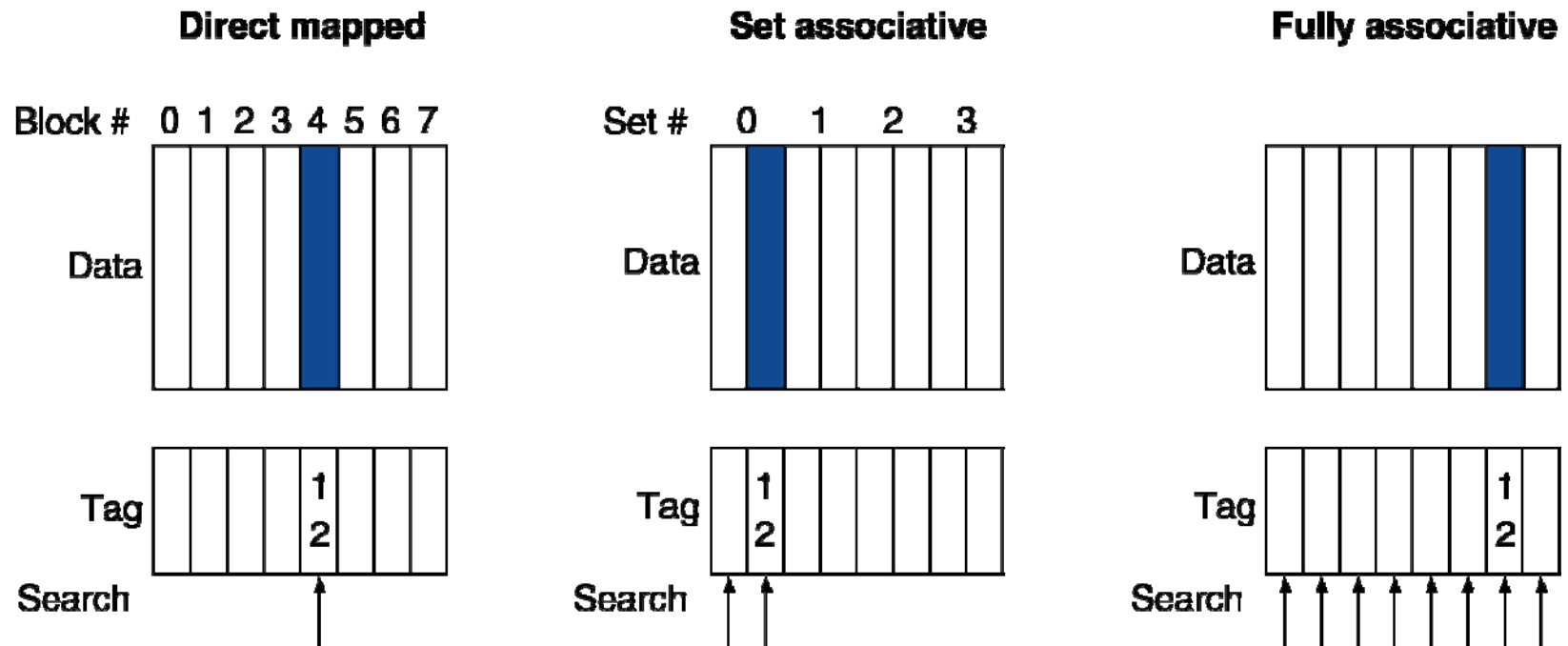
- Fully associative

- Allow a given block to go in any cache entry
- Requires all entries to be searched at once
- One comparator per cache entry (**expensive!**)

- *n*-way set associative

- Each set contains n entries
- Block number determines which set
 - (Block number) modulo (#Sets in cache)
- Search all entries in a given set at once
- n comparators (**less expensive**)

Associative Cache Example



Spectrum of Associativity

- For a cache with 8 entries

**One-way set associative
(direct mapped)**

| Block | Tag | Data |
|-------|-----|------|
| 0 | | |
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |

Two-way set associative

| Set | Tag | Data | Tag | Data |
|-----|-----|------|-----|------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |

Four-way set associative

| Set | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|-----|-----|------|-----|------|-----|------|-----|------|
| 0 | | | | | | | | |
| 1 | | | | | | | | |

Eight-way set associative (fully associative)

| Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data | Tag | Data |
|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|-----|------|
| | | | | | | | | | | | | | | | |

Associativity Example

- Compare 4-block caches
 - Direct mapped, 2-way set associative, fully associative
 - Block access sequence: 0, 8, 0, 6, 8
- Direct mapped
 - Block address to cache block mapping
 - 0 modulo 4 = 0; 6 modulo 4 = 2; 8 modulo 4 = 0;

| Block address | Cache index | Hit/miss | Cache content after access | | | |
|---------------|-------------|----------|----------------------------|---|--------|---|
| | | | 0 | 1 | 2 | 3 |
| 0 | 0 | miss | Mem[0] | | | |
| 8 | 0 | miss | Mem[8] | | | |
| 0 | 0 | miss | Mem[0] | | | |
| 6 | 2 | miss | Mem[0] | | Mem[6] | |
| 8 | 0 | miss | Mem[8] | | Mem[6] | |

Associativity Example

■ 2-way set associative

- 0 modulo 2 = 0; 6 modulo 2 = 0; 8 modulo 2 = 0;

| Block address | Cache index | Hit/miss | Cache content after access | | | |
|---------------|-------------|----------|----------------------------|--------|-------|--|
| | | | Set 0 | | Set 1 | |
| 0 | 0 | miss | Mem[0] | | | |
| 8 | 0 | miss | Mem[0] | Mem[8] | | |
| 0 | 0 | hit | Mem[0] | Mem[8] | | |
| 6 | 0 | miss | Mem[0] | Mem[6] | | |
| 8 | 0 | miss | Mem[8] | Mem[6] | | |

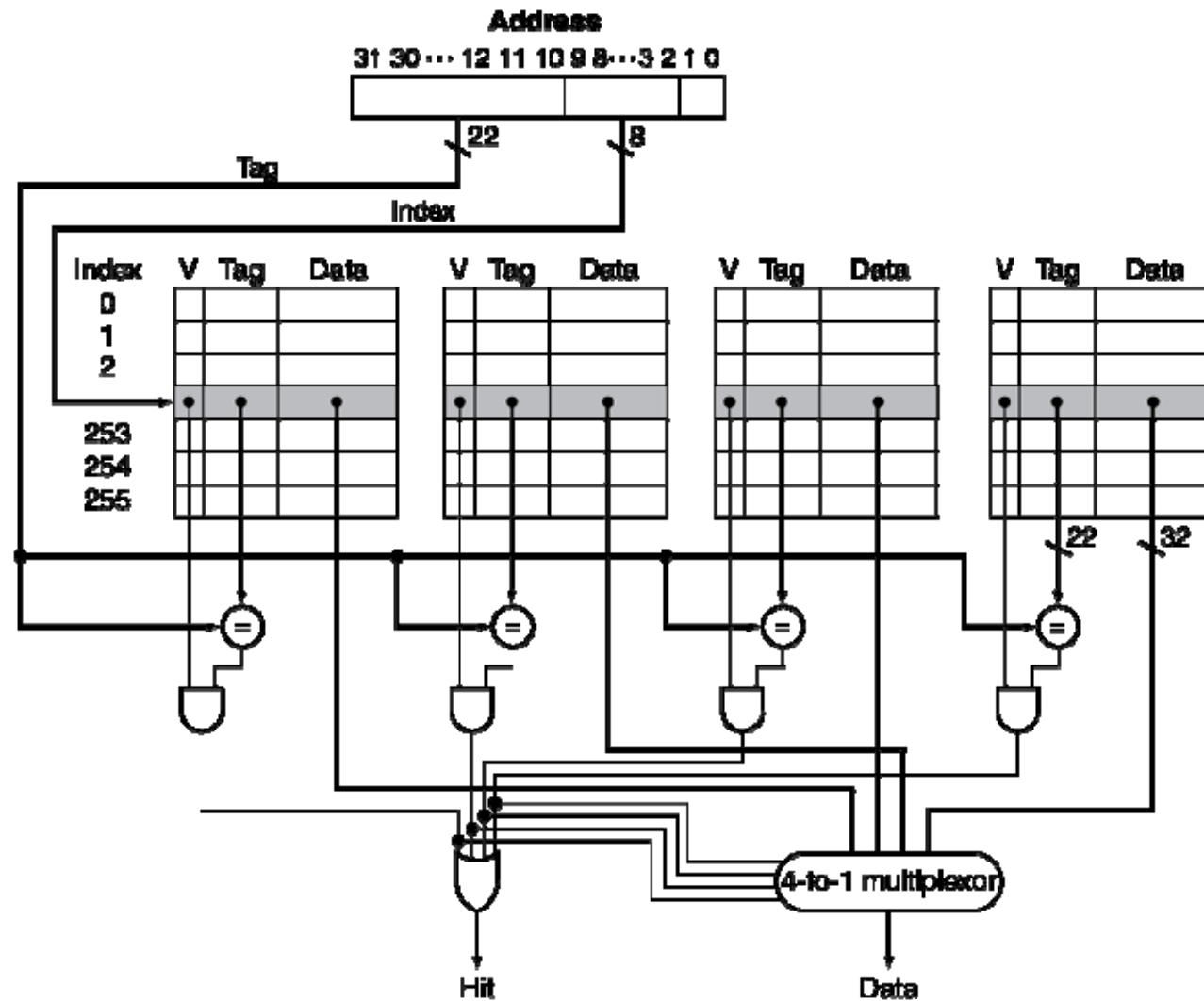
■ Fully associative

| Block address | | Hit/miss | Cache content after access | | | |
|---------------|--|----------|----------------------------|--------|--------|--|
| 0 | | miss | Mem[0] | | | |
| 8 | | miss | Mem[0] | Mem[8] | | |
| 0 | | hit | Mem[0] | Mem[8] | | |
| 6 | | miss | Mem[0] | Mem[8] | Mem[6] | |
| 8 | | hit | Mem[0] | Mem[8] | Mem[6] | |

How Much Associativity

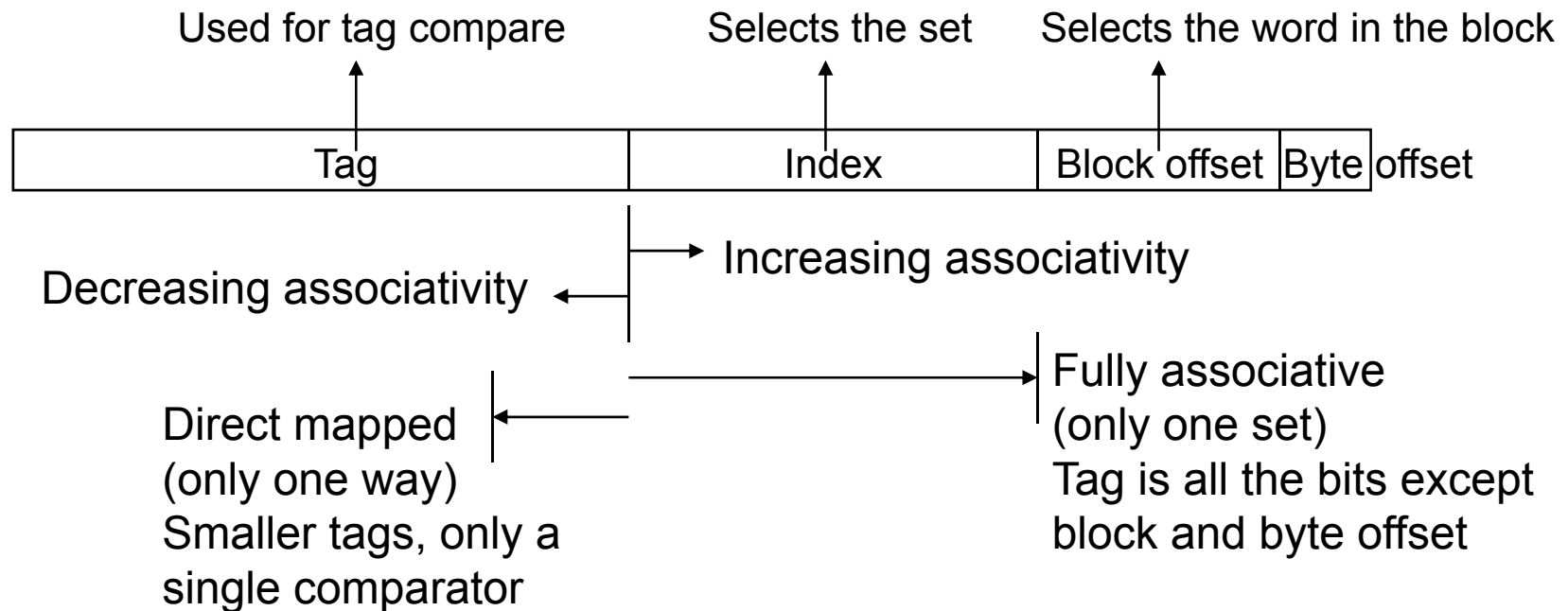
- Increased associativity decreases miss rate
 - But with diminishing returns
- Simulation of a system with 64KB D-cache, 16-word blocks, SPEC2000; Miss Rates:
 - 1-way: 10.3%
 - 2-way: 8.6%
 - 4-way: 8.3%
 - 8-way: 8.1%

Set Associative Cache Organization



Range of Set Associative Caches

- For a fixed size cache, each increase by a factor of two in associativity **doubles** the number of blocks per set (i.e., the number of ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit



Replacement Policy

- Direct mapped
 - no choice
- Set associative
 - Prefer non-valid entry, if there is one
 - Otherwise, choose among entries in the set
 - Least-recently used (LRU)
 - Choose the one unused for the longest time
 - Simple for 2-way, manageable for 4-way, too hard beyond that
 - Random
 - Gives approximately the same performance as LRU for high associativity

Multilevel Caches

- Primary cache attached to CPU
 - Small, but fast
- Level-2 cache services misses from primary cache
 - Larger, slower, but still faster than main memory
- Main memory services L-2 cache misses
- Some high-end systems include L-3 cache

Multilevel Cache Example

- Given
 - CPU base CPI = 1, clock rate = 4GHz
 - Miss rate/instruction = 2%
 - Main memory access time = 100ns
- How much faster will the processor be with a secondary cache with 5 ns access time, and is large enough to reduce miss rate to 0.5%?
- With just primary cache
 - **Miss penalty** = $100\text{ns} / 0.25\text{ns} = 400$ cycles
 - **Effective CPI** = $1 + 0.02 \times 400 = 9$

Example (cont.)

- Now add L-2 cache
 - Access time = 5ns
 - Global miss rate to main memory = 0.5%
- Primary miss with L-2 hit
 - Penalty = $5\text{ns}/0.25\text{ns} = 20$ cycles
- Primary miss with L-2 miss
 - Extra penalty = 400 cycles
- $\text{CPI} = 1 + 0.02 \times 20 + 0.005 \times 400 = 3.4$
- Performance ratio = $9/3.4 = 2.6$

Multilevel Cache Considerations

- **Primary cache**

- Focus on minimal hit time

- **L-2 cache**

- Focus on low miss rate to avoid main memory access
- Hit time has less overall impact

- **Results**

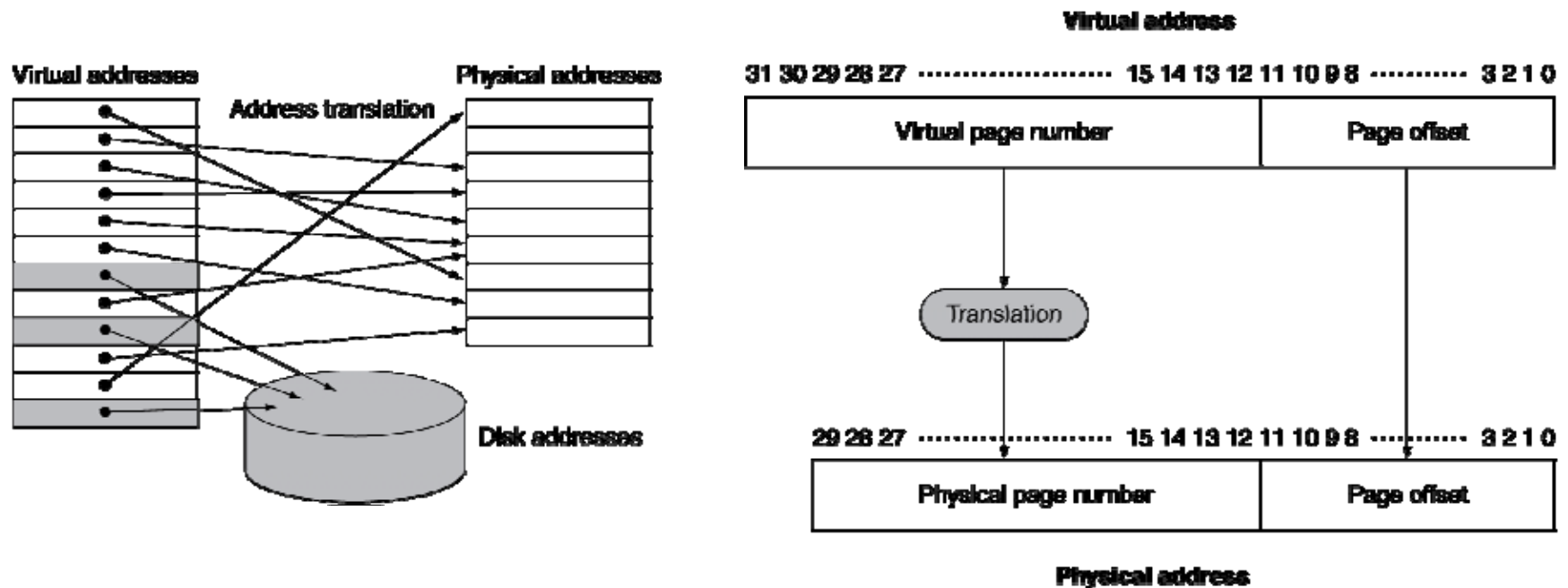
- L-1 cache usually smaller than a single cache
- L-1 block size smaller than L-2 block size

Virtual Memory

- Use main memory as a “cache” for secondary (disk) storage
 - Managed jointly by CPU hardware and the operating system (OS)
- Programs share main memory
 - Each gets a private virtual address space holding its frequently used code and data
 - Protected from other programs
- CPU and OS translate virtual addresses to physical addresses
 - VM “block” is called a **page**
 - VM translation “miss” is called a **page fault**

Address Translation

- Fixed-size pages (e.g., 4K)



- Main memory can have 1GB while virtual address space is 4 GB

Page Tables

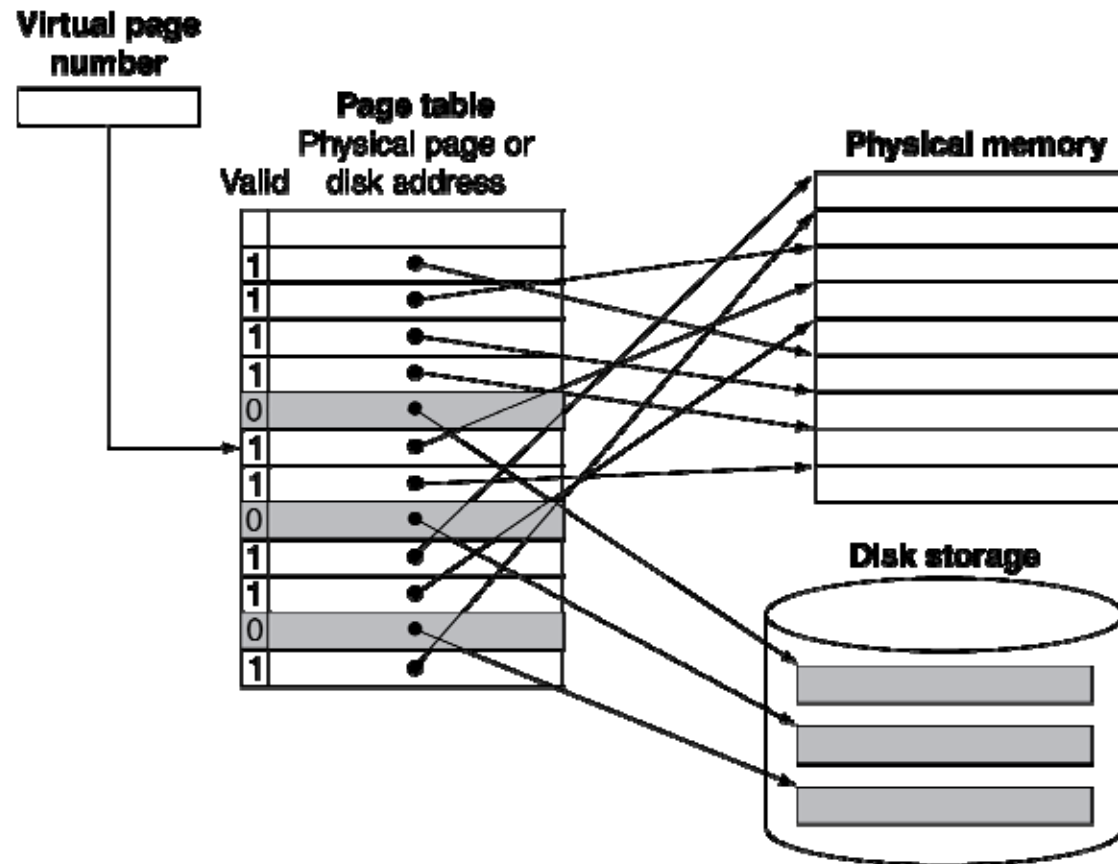
- Stores placement information
 - Array of page table entries (PTEs), indexed by virtual page number
 - Page table register in CPU points to page table in physical memory
- If page is present in memory
 - PTE stores the physical page number
 - Plus other status bits (referenced, dirty, ...)
- If page is not present
 - PTE can refer to location in swap space(为进程的全部虚拟地址空间预留的磁盘空间) on disk

Page Fault Penalty

- On page fault, the page must be fetched from disk
 - Takes **millions** of clock cycles!
 - Handled by OS code
- Try to minimize page fault rate
 - Fully associative placement
 - Smart replacement algorithms

| Age Group | Percentage |
|-----------|------------|
| 18-24 | 10% |
| 25-34 | 20% |
| 35-44 | 25% |
| 45-54 | 20% |
| 55-64 | 15% |
| 65-74 | 10% |
| 75-84 | 5% |
| 85+ | 5% |

Mapping Pages to Storage



Page Table Size Calculation

- 32-bit virtual address space
- 4 KB pages, 4 bytes/PTE
- What is the total page table size?
- No. of page table entries = $2^{32}/2^{12} = 2^{20}$
- Size of page table = 2^{20} PTE x 4 bytes/PTE = 4 MB per program in execution at any given time
- What if there are hundreds of programs in execution?
- How can we handle 64-bit addresses that would need 2^{52} PTEs?

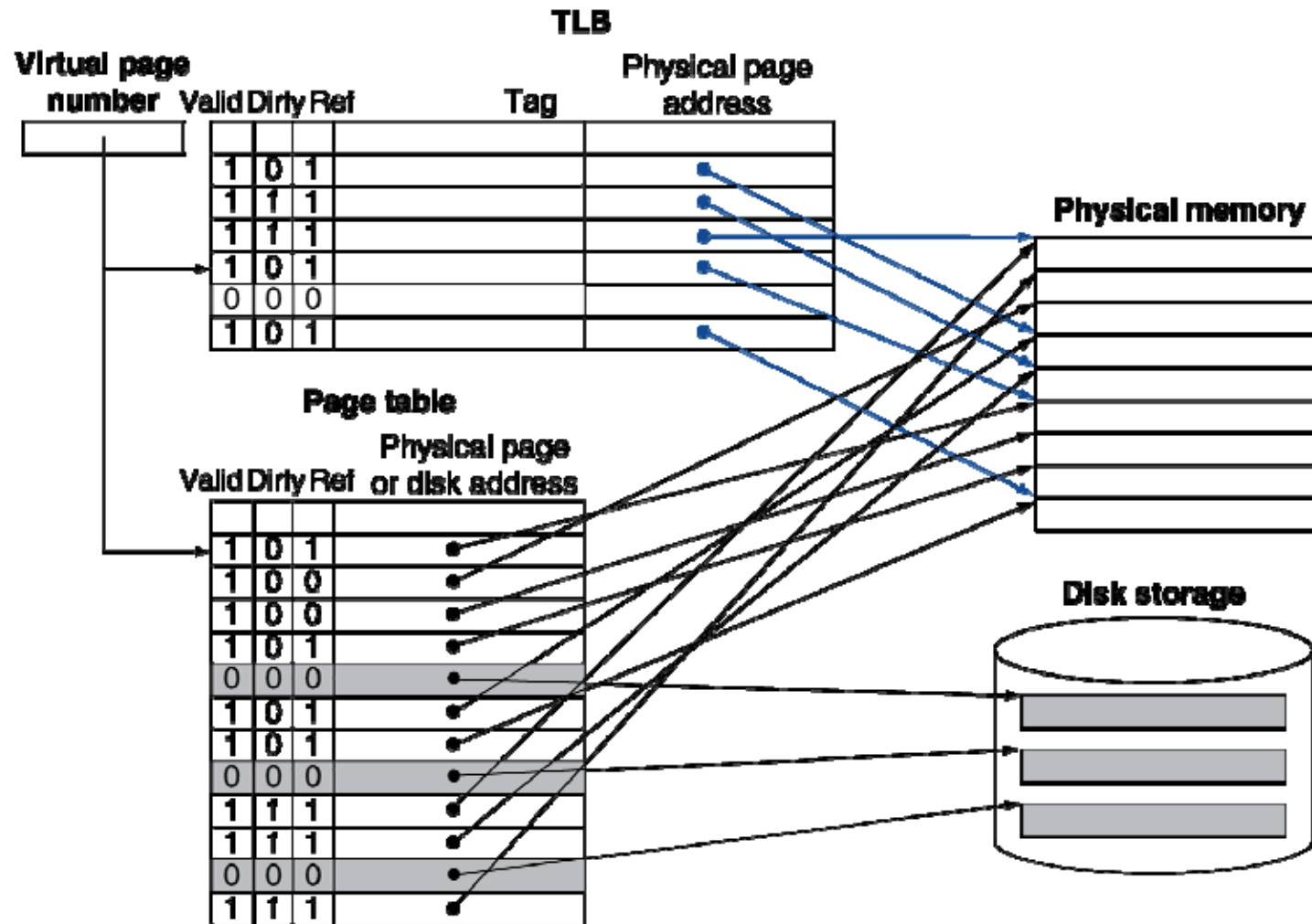
Replacement and Writes

- To reduce page fault rate, prefer **least-recently used (LRU)** replacement
 - Reference bit (aka use bit) in PTE set to 1 on access to page
 - Periodically cleared to 0 by OS
 - A page with reference bit = 0 has not been used recently
- Disk writes take millions of cycles
 - Block at once, not individual locations/整页复制而不是单字操作
 - **Write through** is impractical
 - Use write-back
 - Dirty bit in PTE set when page is written

Fast Translation Using a TLB

- Address translation would appear to require extra memory references
 - One to access the PTE
 - Then the actual memory access
- But access to page tables has good locality
 - So use a fast cache of PTEs within the CPU
 - Called a **Translation Look-aside Buffer (TLB)**
 - Typical: 16–512 PTEs, 0.5–1 cycle for hit, 10–100 cycles for miss, 0.01%–1% miss rate
 - Small TLBs are typically fully associative, large TLBs have small associativity (due to cost issues)
 - Misses could be handled by hardware or software
 - Write back scheme for writing reference, dirty bits to PTE

Fast Translation Using a TLB



TLB Misses

- If page is in memory
 - Load the PTE from memory and retry
 - 10's of cycles
 - Could be handled in hardware
 - Can get complex for more complex page table structures
 - Or in software
 - Raise a special exception, with optimized handler
- If page is not in memory (page fault)
 - OS handles fetching the page and updating page table
 - Then restart the faulting instruction
 - 1,000,000's of cycles
- **TLB misses more frequent than page faults**

Page Fault Handler

- Use faulting virtual address to find PTE
- Locate page on disk
- Choose page to replace
 - If dirty, write to disk first
- Read page into memory and update page table
- Make process runnable again
 - Restart from faulting instruction

TLB, VM, Cache Event Combinations

| TLB | Page Table | Cache | Possible? Under what circumstances? |
|------|------------|--------------|--|
| Hit | Hit | Hit | Yes – what we want! |
| Hit | Hit | Miss | Yes – although the page table is not checked if the TLB hits |
| Miss | Hit | Hit | Yes – TLB miss, PA in page table |
| Miss | Hit | Miss | Yes – TLB miss, PA in page table, but data not in cache |
| Miss | Miss | Miss | Yes – page fault |
| Hit | Miss | Miss/ Hit | Impossible – TLB translation not possible if page is not present in memory |
| Miss | Miss | Hit | Impossible – data not allowed in cache if page is not in memory |

The Memory Hierarchy

The BIG Picture

- Common principles apply at all levels of the memory hierarchy
 - Based on notions of caching/基于高速缓存的概念和原理
- At each level in the hierarchy
 - Block placement
 - Finding a block
 - Replacement on a miss
 - Write policy

Characteristics of Memory Hierarchy

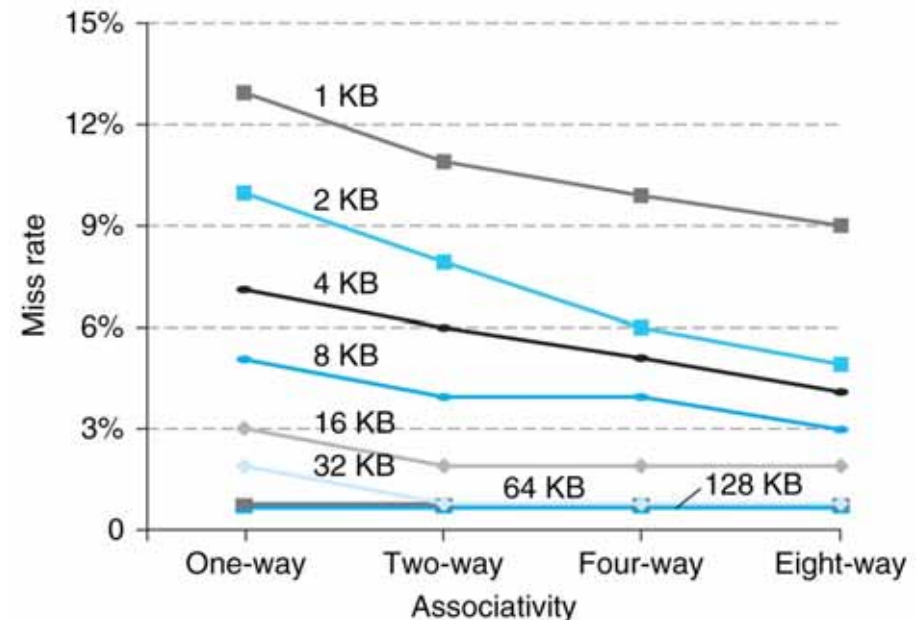
| Feature | Typical values for L1 caches | Typical values for L2 caches | Typical values for paged memory | Typical values for a TLB |
|----------------------------|------------------------------|------------------------------|---------------------------------|--------------------------|
| Total size in blocks | 250–2000 | 15,000–50,000 | 16,000–250,000 | 40–1024 |
| Total size in kilobytes | 16–64 | 500–4000 | 1,000,000–1,000,000,000 | 0.25–16 |
| Block size in bytes | 16–64 | 64–128 | 4000–64,000 | 4–32 |
| Miss penalty in clocks | 10–25 | 100–1000 | 10,000,000–100,000,000 | 10–1000 |
| Miss rates (global for L2) | 2%–5% | 0.1%–2% | 0.00001%–0.0001% | 0.01%–2% |

- **Four Questions for Memory Hierarchy**
 - Q1: Where can a block be placed in the upper level?
 - Q2: How is a block found if it is in the upper level?
 - Q3: Which block should be replaced on a miss?
 - Q4: What happens on a write?

Q1. Block Placement?

- Determined by associativity

- Direct mapped (1-way associative)
 - One choice for placement
- n-way set associative
 - n choices within a set
- Fully associative
 - Any location



- Higher associativity reduces miss rate
 - Increases complexity, cost, and access time

Q2. Finding a Block?

| Associativity | Location method | Tag comparisons |
|-----------------------|---|-----------------|
| Direct mapped | Index | 1 |
| n-way set associative | Set index, then search entries within the set | n |
| Fully associative | Search all entries | #entries |

- Hardware caches
 - Reduce comparisons to reduce cost
 - Usually **set associative** or (less commonly) **direct mapped**
- Virtual memory systems (with page tables)
 - Almost always **fully associative**, as misses are very expensive
 - Software can use sophisticated replacement schemes to further reduce miss rate

Q3. Replacement on Miss?

- Choice of entry to replace on a miss
 - Least recently used (LRU)
 - Complex and costly hardware for high associativity
 - Random
 - Close to LRU, easier to implement
- Virtual memory
 - LRU approximation with hardware support

Q4. Write Policy?

- Write-through

- Update both upper and lower levels
- Simplifies replacement, but may require write buffer

- Write-back

- Update upper level only
- Update lower level when block is replaced
- Need to keep more state

- Virtual memory

- Only write-back is feasible, given disk write latency

Sources of Cache Misses

- **Compulsory强制** (cold start or process migration进程迁移, first reference):
 - If you are going to run “millions” of instruction, compulsory misses are insignificant
 - **Solution:** increase block size (increases miss penalty; very large blocks could increase miss rate)
- **Capacity:**
 - Cache cannot contain all blocks accessed by the program
 - **Solution:** increase cache size (may increase access time)
- **Conflict** (collision):
 - Multiple memory locations mapped to the same cache location
 - **Solution 1:** increase cache size (may increase access time)
 - **Solution 2:** increase associativity (may increase access time)

Cache Design Trade-offs: Summary

| Design change | Effect on miss rate | Negative performance effect |
|------------------------|----------------------------|---|
| Increase cache size | Decrease capacity misses | May increase access time |
| Increase associativity | Decrease conflict misses | May increase access time |
| Increase block size | Decrease compulsory misses | Increases miss penalty. For very large block size, may increase miss rate due to pollution. |



重庆大学
CHONGQING UNIVERSITY

计算机学院

COLLEGE OF COMPUTER SCIENCE

输入输出(I/O)控制

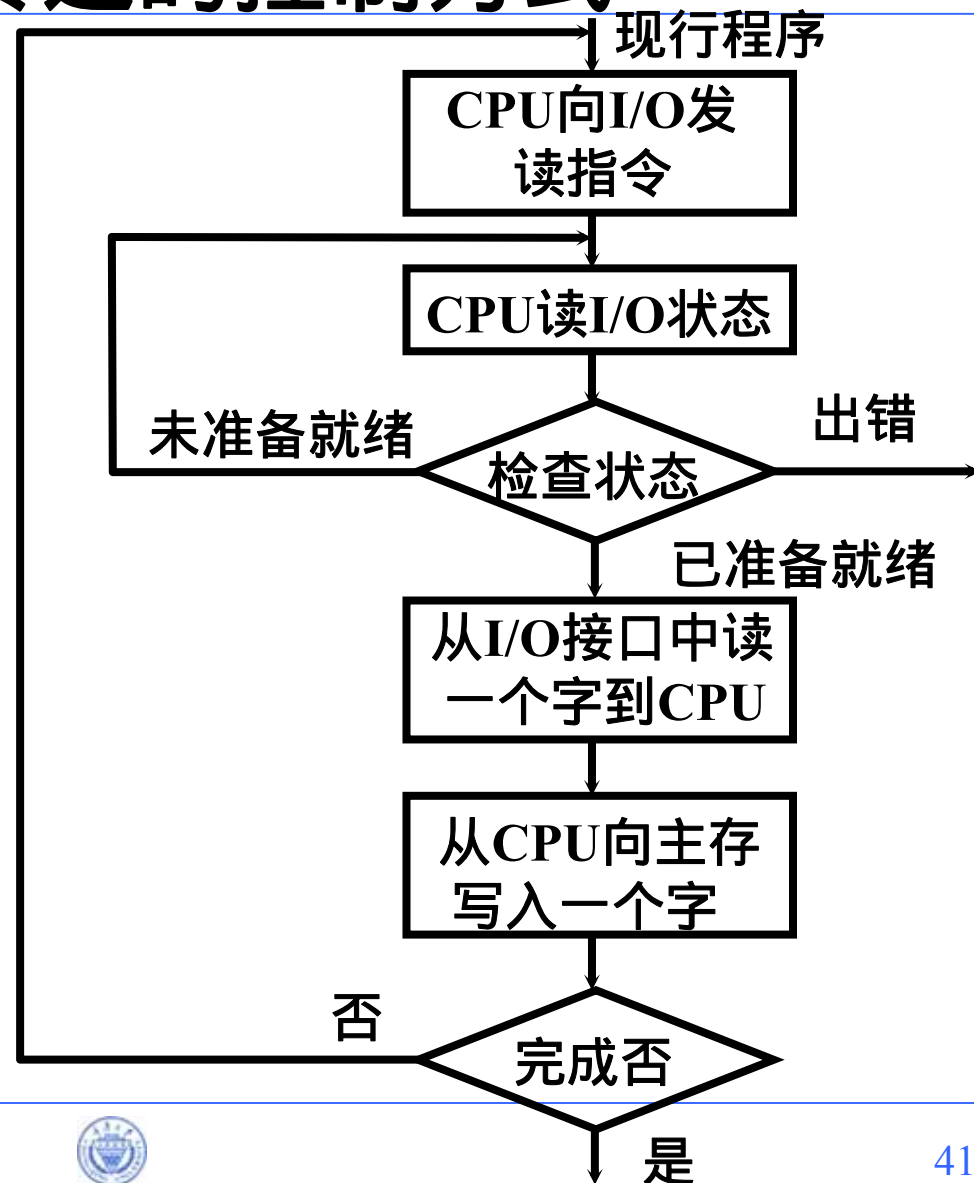
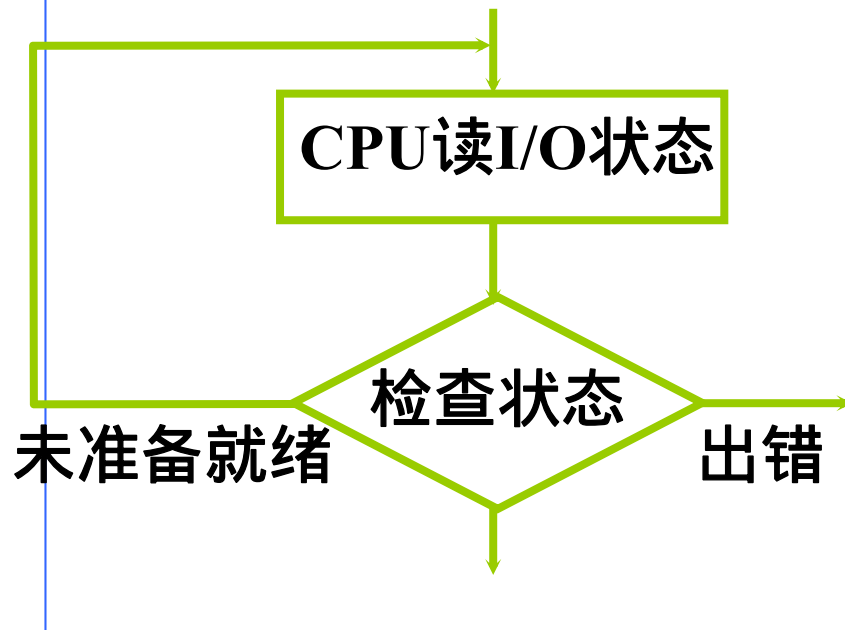
6.0 概述

I/O 与主机信息传送的控制方式

1. 程序查询方式

CPU 和 I/O 串行工作

踏步等待



6.0 概述

1. 程序查询方式（Polling）

■ 特点

- 何时对何设备进行输入或输出操作完全受CPU控制
- CPU通过指令对设备进行测试才能知道设备的工作状态。设备空闲、准备就绪、正在忙等
- 数据的输入和输出都要经过CPU
- 用于连接低速外围设备，如终端、打印机等

■ 优点

- 灵活性好。可以很容易地改变各台外围设备的优先级

■ 缺点

- 实现处理机与外围设备并行工作困难

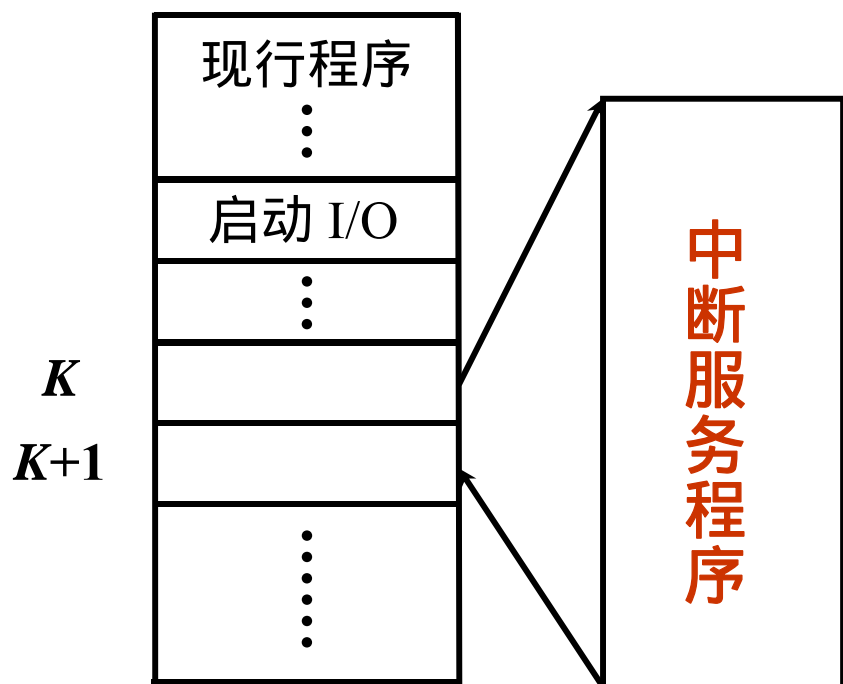
6.0 概述

2. 程序中中断方式

I/O 工作 { 自身准备
与主机交换信息

CPU 不查询

CPU 暂停现执行程序



CPU 和 I/O 并行工作

没有踏步等待现象

中断现执行程序

6.0 概述

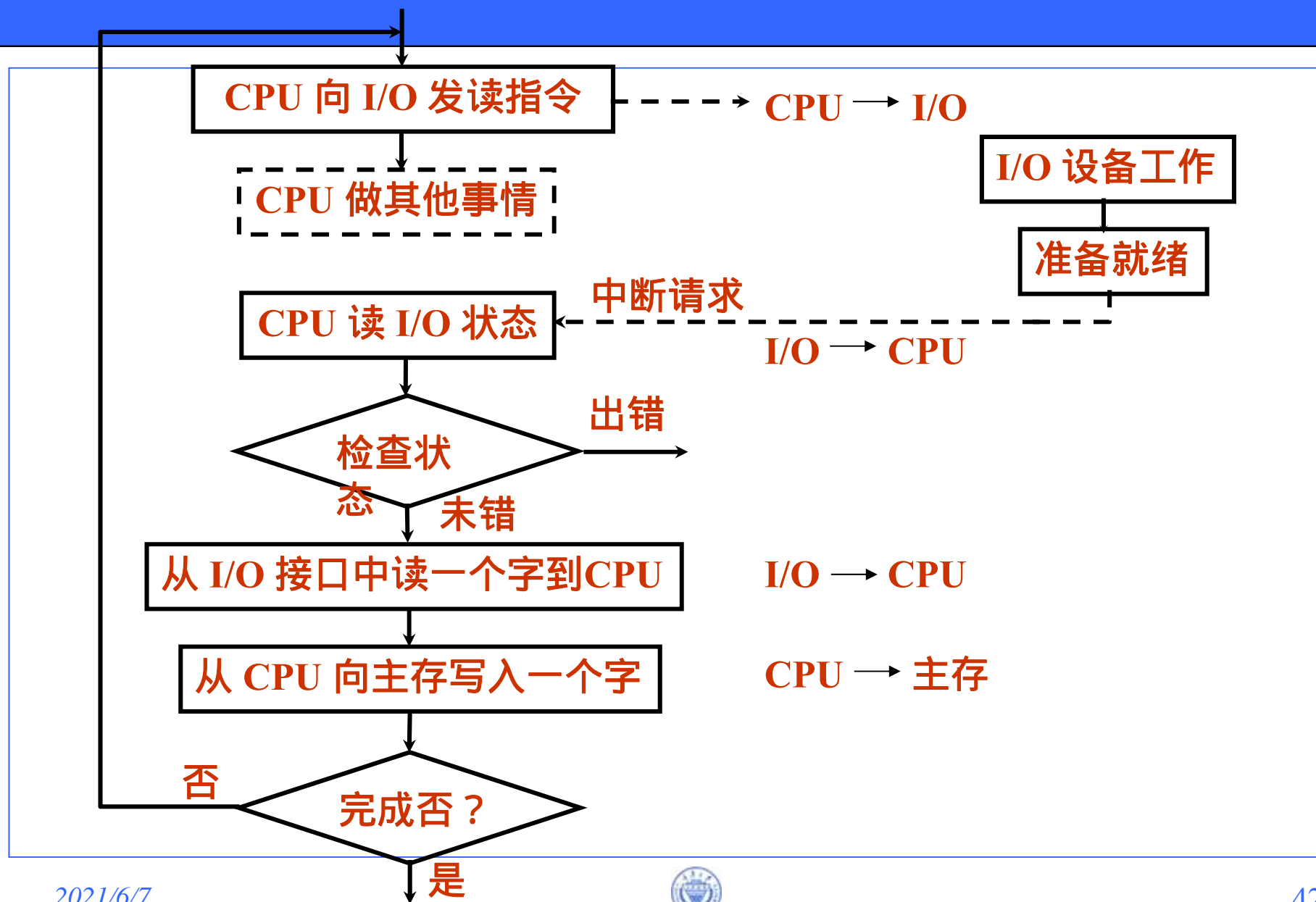
2. 程序中断方式

■ 特点

- ♣ CPU与外围设备能够并行工作
- ♣ 能够处理例外事件。例如，电源掉电、非法指令、地址越界、数据溢出、数据校验错、页面失效等
- ♣ 数据的输入和输出都要经过CPU
- ♣ 灵活性好
- ♣ 用于连接低速外围设备

在现代计算机系统中，中断输入输出方式的作用已经远远超出了为外围设备服务的范畴，成为现代计算机系统中非常重要的一个组成部分。

■ 程序中断方式流程



6.0 概述

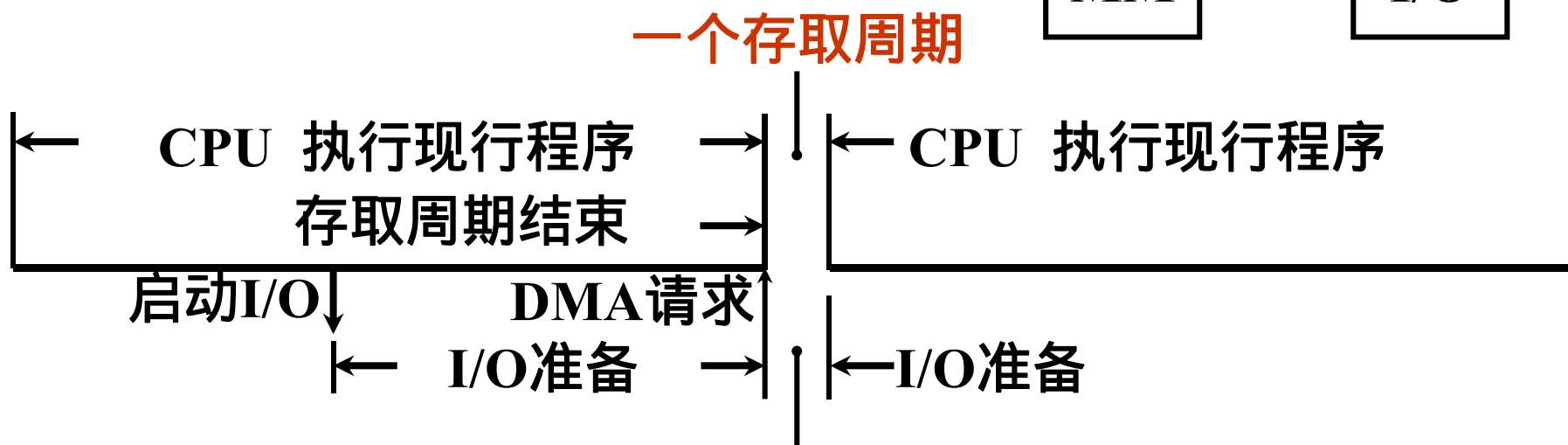
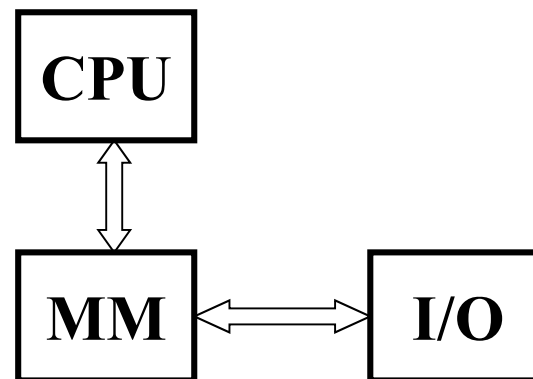
3. DMA 方式

主存和 I/O 之间有一条直接数据通道

不中断现行程序

周期挪用（周期窃取）

CPU 和 I/O 并行工作



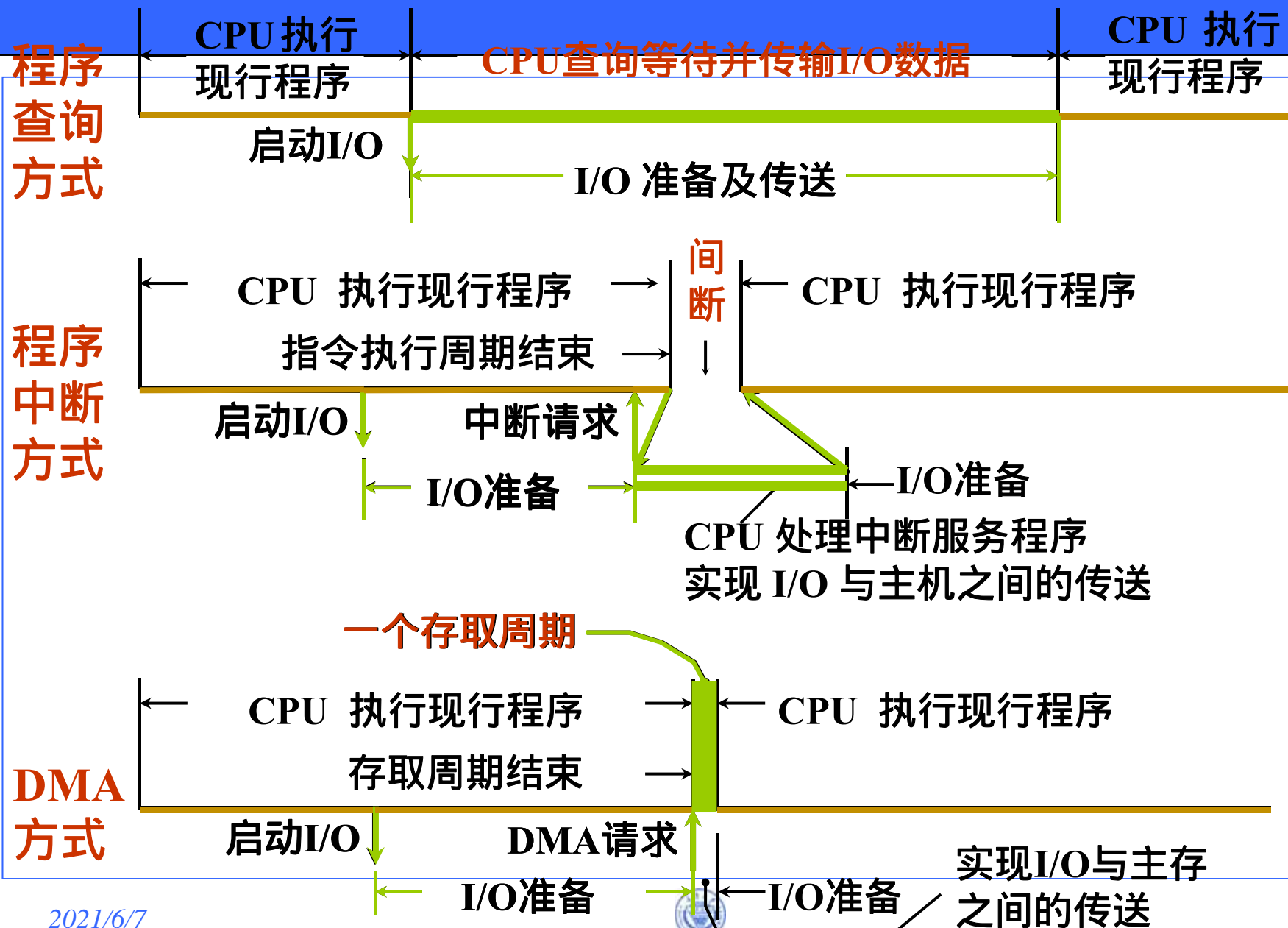
实现I/O与主存之间的传送

6.0 概述

■ 特点

- 外围设备访问请求直接发往主存储器
- 不需要CPU做保存现场和恢复现场等工作
- 在DMA控制器中，需要设置数据寄存器、设备状态或控制寄存器、主存地址寄存器、设备地址寄存器和数据交换个数计数器
- 在DMA方式开始和结束时，需要处理机进行管理
 - 在DMA方式开始之前对DMA控制器进行初始化。传送主存缓冲区首地址、设备地址、数据块的长度等，并启动设备开始工作
 - 在DMA方式结束之后，向CPU申请中断，对数据缓冲区进行后处理
- 数据的传送过程不需要CPU的干预

■ 三种方式的 CPU 工作效率比较





第 6.1 节

中断及程序中断控制传送

6.1.1 中断的有关问题

■ 中断的类型

3. 单重中断系统和多重中断系统

- 单重中断系统：执行中断服务程序时，不能再响应其它中断的系统称单重中断系统
- 多重中断系统：执行中断服务程序时，还可响应更高优先级中断的系统称多重中断系统——中断嵌套

6.1.1 中断的有关问题

■ 中断的类型

4. 可屏蔽中断和不可屏蔽中断

- 可屏蔽中断：可不响应或暂不响应，或有条件的响应的中断

- 不可屏蔽中断：必须立即处理的、不能回避和禁止的中断。断电中断是具有最高优先级的不可屏蔽中断

用程序方法有选择性地封锁部分中断，使之不发出中断请求，而允许其余中断发出中断请求并得到响应

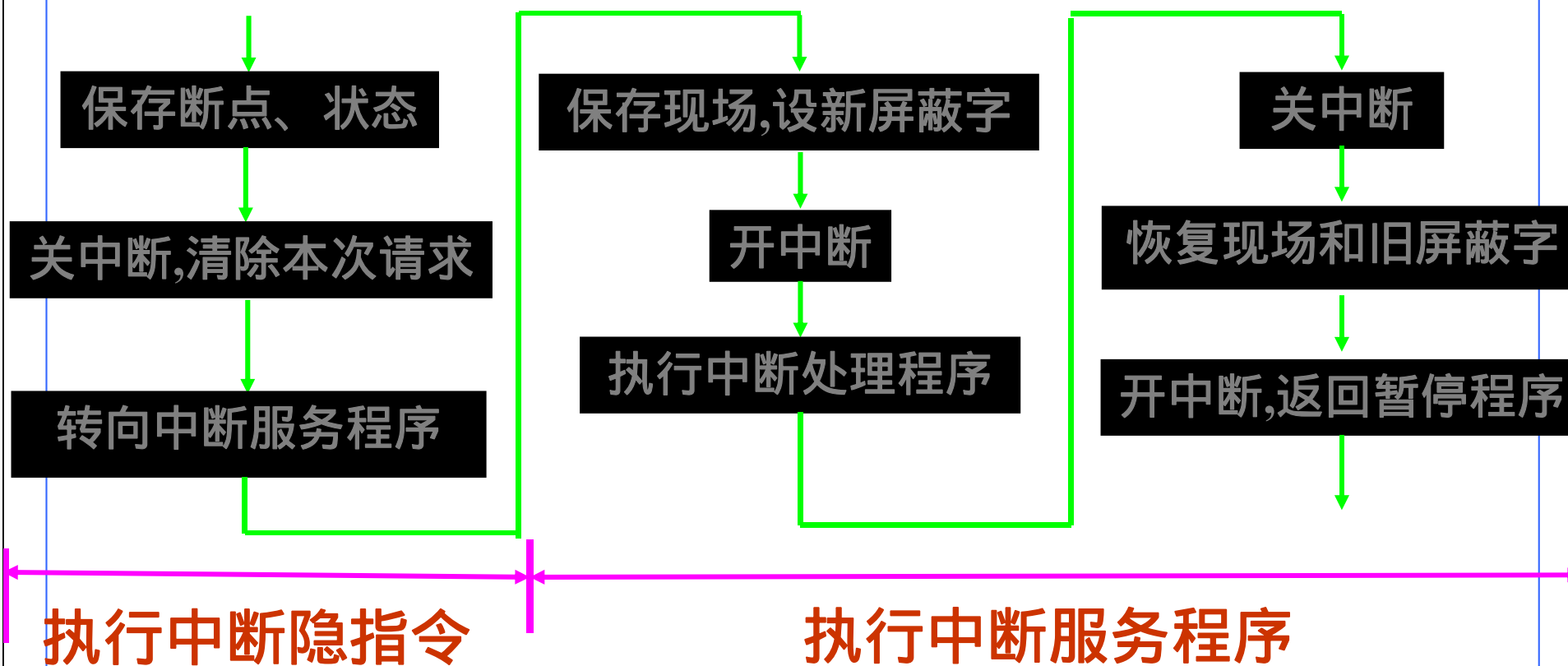
6.1.2 中断系统的结构组成

■ 中断系统的任务（中断应解决的问题）

1. 接收并保存中断请求（硬件完成）
2. 进行中断判优（软、硬件均可）
3. 实施中断响应（硬件完成）
4. 实现中断处理（软件完成）
5. 返回被暂停(中止)的程序（软件完成IRET）

6.1.2 中断系统的结构组成

□ 中断响应和中断处理过程



6.1.2 中断系统的结构组成

■ 中断屏蔽技术与多重中断的实现

1) 中断屏蔽技术

- **硬件**：设置屏蔽触发器及其相关电路
- **软件**：依中断源的**优先级**为每个中断源**预先**设1个中断屏蔽码。屏蔽码与中断源的**优先级**别一一对应

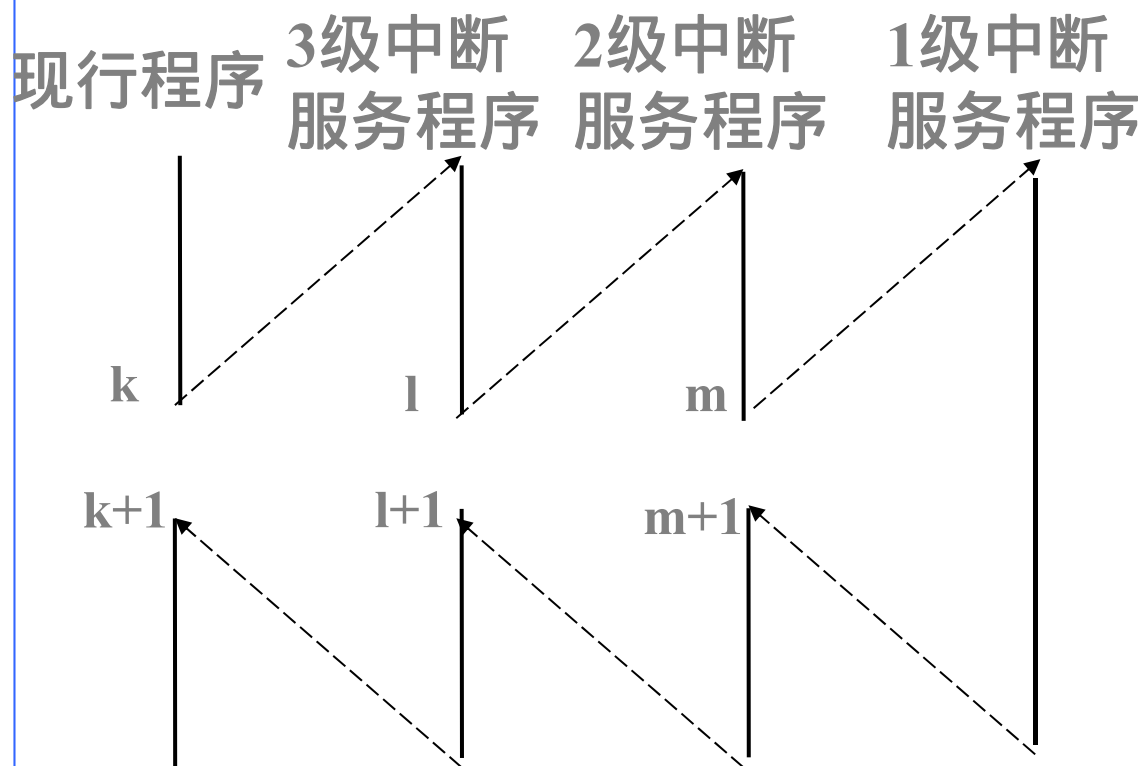
| 中断源序号 | 优先级 | 中断屏蔽码 |
|-------|-----|----------|
| 1 | 1 | 11111111 |
| 2 | 2 | 01111111 |
| 3 | 3 | 00111111 |
| 4 | 4 | 00011111 |
| 5 | 5 | 00001111 |
| 6 | 6 | 00000111 |
| 7 | 7 | 00000011 |
| 8 | 8 | 00000001 |

进入中断服务程序保护现场之后，通过屏蔽指令送新的屏蔽码达到中断屏蔽目的，实现了多重中断。

6.1.2 中断系统的结构组成

■ 中断屏蔽与多重中断的实现

2) 多重中断的实现



■ 借助中断屏蔽寄存器和屏蔽码，通过屏蔽指令实现多重中断

1级中断的屏蔽码

11111111

2级中断的屏蔽码

01111111

3级中断的屏蔽码

00111111

6.1.2 中断系统的结构组成

■ 中断的全过程（小结）

■ 中断请求

■ 中断判优

■ 中断响应

■ 中断处理

■ 中断返回

} 软件判优时两步合为一步

} 软件实现

6.1.2 中断系统的结构组成

■ 中断服务程序流程

1. 中断服务程序的流程

(1) 保护现场

{ 程序断点的保护
寄存器内容的保护

中断隐指令完成
进栈指令

(2) 中断服务

对不同的 I/O 设备具有不同内容的设备服务

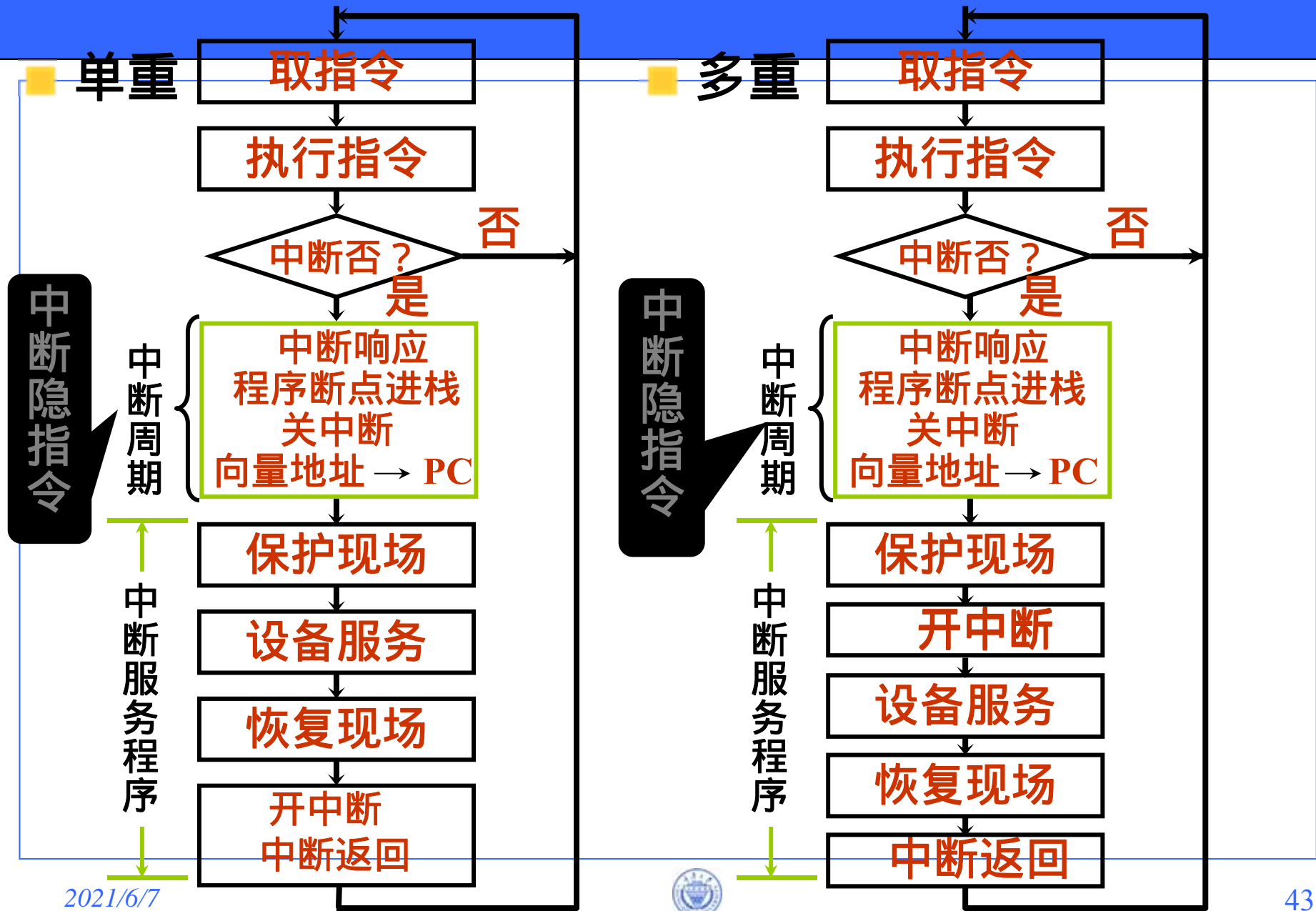
(3) 恢复现场

出栈指令

(4) 中断返回

中断返回指令

2. 单重中断和多重中断的服务程序流程



例：有四个中断源 D_1 、 D_2 、 D_3 和 D_4 ，它们的中断优先级从高到低分别是1级、2级、3级和4级。这些中断源的正常中断屏蔽码和改变后的中断屏蔽码见下表。每个中断源一位，共4位屏蔽码。

| 中断源名称 | 中断优先级 | 正常中断屏蔽码 $D_1 D_2 D_3 D_4$ | 改变后的中断屏蔽码 $D_1 D_2 D_3 D_4$ |
|-------|-------|------------------------------|--------------------------------|
| D1 | 1 | 1 1 1 1 | 1 0 0 0 |
| D2 | 2 | 0 1 1 1 | 1 1 0 0 |
| D3 | 3 | 0 0 1 1 | 1 1 1 0 |
| D4 | 4 | 0 0 0 1 | 1 1 1 1 |



解：

如果4个中断源都使用正常的中断屏蔽码，处理机的中断服务顺序将严格按照中断源的中断优先级进行。

如果改变中断屏蔽码，当 D_1 、 D_2 、 D_3 和 D_4 这4个中断源同时请求中断服务时，处理机实际为各个中断源服务的先后次序就会改变。

处理机响应的顺序是 D_1 、 D_2 、 D_3 、 D_4

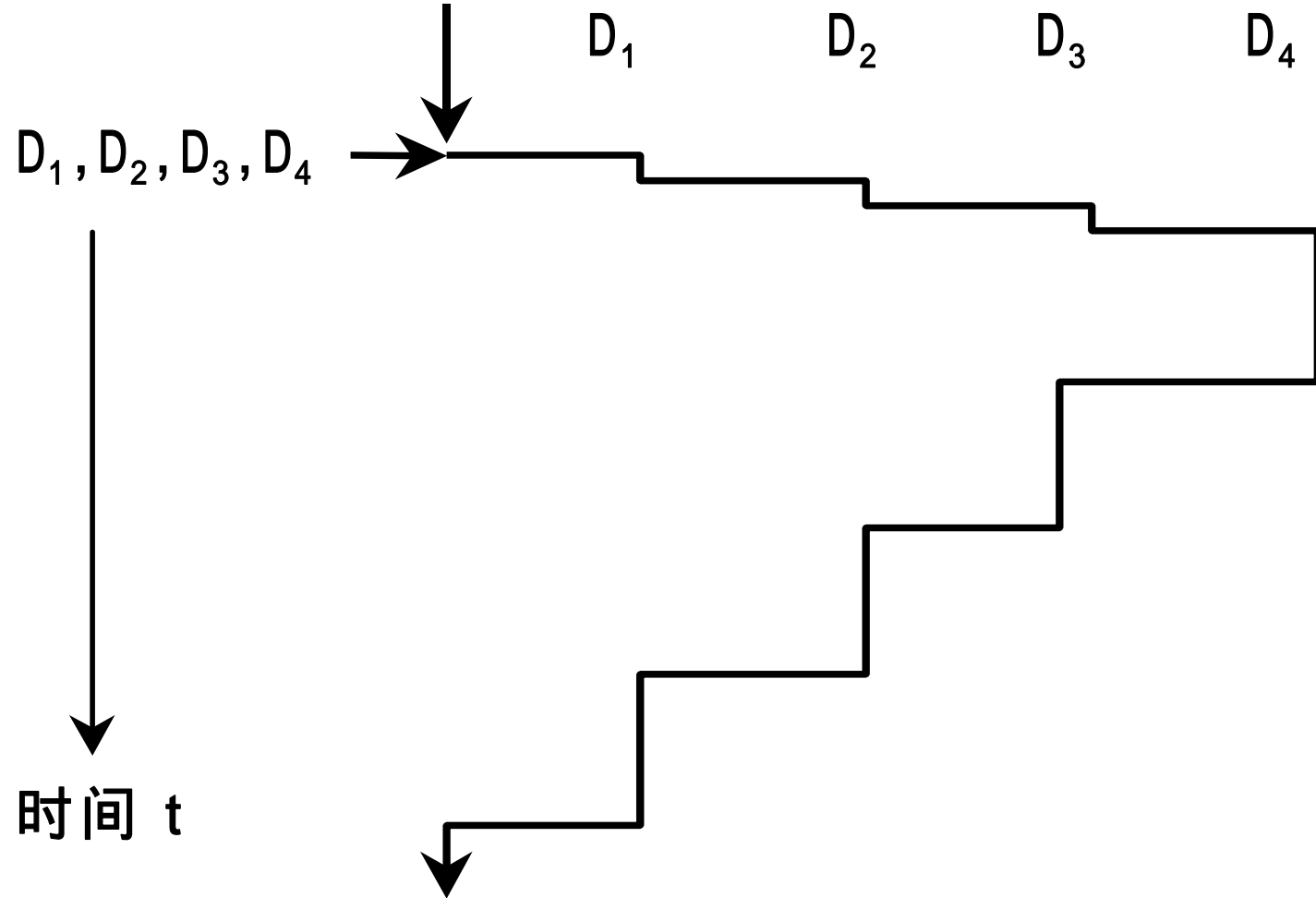
实际服务的顺序是 D_4 、 D_3 、 D_2 、 D_1



中断请求

主程序

中断服务程序





6.2

内存直接存取(DMA)

6.2.1 DMA概述

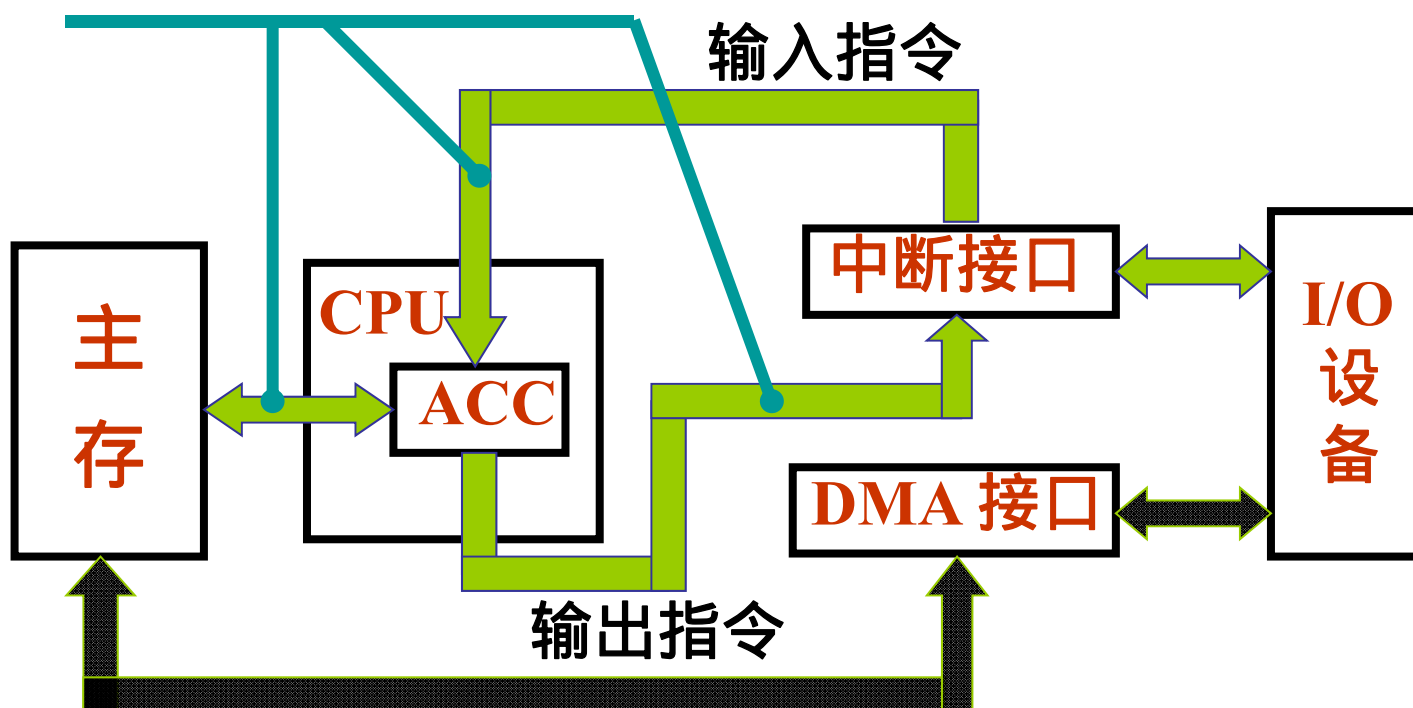
- 直接存储器服务方式又称为DMA（Direct Memory Access）方式
 - 以主存为中心，完全用硬件开辟外设和主存之间直接数据传输通路
 - 能使CPU工作效率更高的一种控制方式
 - DMA方式主要用来连接高速外围设备。例如，磁盘存储器，磁带存储器等

6.2.1 DMA概述

■ DMA 方式的特点

■ DMA和程序中断两种方式的数据通路

中断方式数据传送通路



DMA方式数据传送通路

6.2.1 DMA概述

■ DMA方式的特点

- 外围设备访问请求直接发往主存储器
- 不需要CPU做保存现场和恢复现场等工作
- DMA控制器中，需要设置数据寄存器、设备状态或控制寄存器、主存地址寄存器、设备地址寄存器和数据交换个数计数器
- DMA开始和结束时，需要处理机进行管理
 - 在DMA方式开始前，对DMA控制器进行初始化。传送主存缓冲区首地址、设备地址、数据块的长度等，并启动设备开始工作
 - 在DMA方式结束后，向CPU申请中断，对数据缓冲区进行后处理
- DMA执行中，数据的传送过程不需要CPU的干预

6.2.1 DMA概述

DMA工作方式

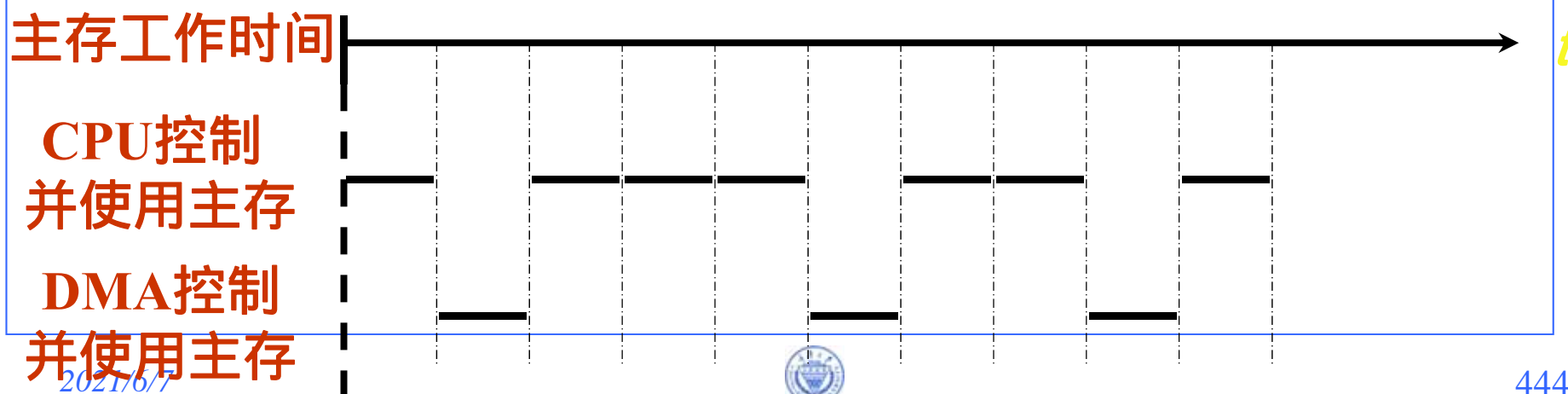
1. 周期挪用（或称周期窃取）

DMA 要求访问主存有三种可能：

- CPU 此时不访存（无冲突）
- CPU 正在访存（需等待）
- CPU 与 DMA 同时请求访存（冲突）

♣ DMA必须优先于CPU访存？

♣ CPU 将总线控制权让给 DMA



6.2.1 DMA概述

1.周期挪用（简单中断）

■ 处理方法

- ♣ PD需要传送时，CPU暂停一个或若干个存储周期让给DMA
- ♣ PD交换一个单位数据后，CPU继续工作

■ 使用场合

- ♣ PD访存周期>>主存存储周期，单位数据准备时间的间隔大，单位数据准备时间至少四倍于存储周期

■ 优缺点

- ♣ 存储器和CPU效率均较高，但在数据输入或输出过程中实际上占用了CPU的时间
- ♣ 控制较复杂

■ DMA大多采用这种方式

6.2.3 DMA的工作过程

1. DMA 传送过程

■ 初始化处理、数据传送、结束处理

(1) 初始化处理 (CPU程序实现)

■ CPU通过一条I/O指令取状态

♣ 查询外设是否闲置良好

■ CPU通过几条I/O指令预置如下信息

♣ 通知 DMA 控制逻辑传送方向 (入/出)

♣ 设备地址 → DMA 的 DAR

♣ 主存地址 → DMA 的 ABR

♣ 传送字数 → DMA 的 WC

■ 用I/O指令启动外设,CPU继续执行现行程序

(2) DMA 传送过程示意

CPU

数据传送

DMA请求

初始化处理：

主存起始地址 → DMA
设备地址 → DMA
传送数据个数 → DMA
启动设备

数据传送：

继续执行主程序
同时完成一批数据传送

结束处理：

中断服务程序
做 DMA 结束处理

继续执行主程序



否

允许传送？

是

主存地址送总线
数据送I/O设备(或主存)
主存地址 加 1
传送个数 减 1

否

数据块
传送结束？

是

向CPU申请 程序中断

6.2.3 DMA的工作过程

(3) 结束处理

校验送入主存的数据是否正确

是否继续用 DMA

测试传送过程是否正确，错则转错误处理程序

由中断服务程序完成