# CO Final Cheatsheet

> [!NOTE] Note
>
> 1. 并不是最终使用的版本
> 2. ***与其抄很多知识点，不如抄点例题！***

# 1 Chapter 1

## 1.1 performance

- elapse time 总时间
- cpu time 除去 I/O 等其他因素的时间
- clock cycle time 时钟周期
- clock rate 时钟频率
- 同样的代码，如果 ISA 相同，则 #inst 相同
- power of cpu $P = C \times V^2 \times F$
- amdahl's law $T_{\text{improved}} = \frac{T_{\text{affected}}}{\text{improvement factor}} + T_{\text{unaffected}}$
- MIPS: 每秒钟百万指令数

## 1.2 great ideas

- design for moore's law
- use abstraction to simplify design
- make the common case fast
- performance via parallelism
- performance via pipelining
- performance via prediction
- hierarchy of memories
- dependability via redundancy

# 2 Chapter 2 Instructions

## 2.1 指令表

| | 6 | 1 | 5 | 5 | 3 | 5 | 7 |
|---|---|---|---|---|---|---|---|
| R | funct7 | | rs2 | rs1 | funct3 | rd | opcode |
| I | i[11:0] | | | rs1 | funct3 | rd | opcode |
| I | funct6 | i[5:0] | | rs1 | funct3 | rd | opcode |
| S | i[11:5] | | rs2 | rs1 | funct3 | i[4:0] | opcode |
| SB | i[12, 10:5] | | rs2 | rs1 | funct3 | i[4:1, 11] | opcode |
| UJ | i[20, 10:1, 11, 19:12] | | | | | rd | opcode |
| U | i[31:12] | | | | | rd | opcode |

*References/CO-InstTypes.webp*

| Category | Inst | Example & Comments | | | FMT | OpCode | Funct3 | Funct6/7 |
|---|---|---|---|---|---|---|---|---|
| Arithmetic | add | Add | add rd, rs1, rs2 | rd = rs1 + rs2 | R | 0110011 | 000 | 0000000 |
| | sub | Subtract | sub rd, rs1, rs2 | rd = rs1 - rs2 | R | 0110011 | 000 | 0100000 |
| | addi | Add imm | addi rd, rs1, -20 | rd = rs1 + (-20) | I | 0010011 | 000 | n.a. |
| | slt | Set if less than | slt rd, rs1, rs2 | rd = rs1 < rs2 ? 1 : 0 | R | 0110011 | 010 | 0000000 |
| | sltu | slt, unsigned | sltu rd, rs1, rs2 | rd = rs1 < rs2 ? 1 : 0 | R | 0110011 | 011 | 0000000 |
| | slti | slt, imm | slti rd, rs1, imm | rd = rs1 < imm ? 1 : 0 | I | 0010011 | 010 | n.a. |
| | sltiu | slt, imm & unsigned | sltiu rd, rs1, imm | rd = rs1 < imm ? 1 : 0 | I | 0010011 | 011 | n.a. |
| | mul | mul, lower 64 of result | mul rd, rs1, rs2 | rd = rs1 * rs2 (lower 64) | R | - | - | - |
| | mulh | mul, upper 64 of result | mulh rd, rs1, rs2 | rd = (rs1 * rs2) >> 64 | R | - | - | - |
| | mulhu | mulh, unsgn * unsgn | mulhu rd, rs1, rs2 | rd = (rs1 * rs2) >> 64 | R | - | - | - |
| | mulhsu | mulh, sgn * unsgn | mulhsu rd, rs1, rs2 | rd = (rs1 * rs2) >> 64 | R | - | - | - |
| | div | divide | div rd, rs1, rs2 | rd = rs1 / rs2 | R | - | - | - |
| | divu | divide unsigned | divu rd, rs1, rs2 | rd = rs1 / rs2 | R | - | - | - |
| | rem | remainder | rem rd, rs1, rs2 | rd = rs1 % rs2 | R | - | - | - |
| | remu | remainder unsigned | remu rd, rs1, rs2 | rd = rs1 % rs2 | R | - | - | - |
| Data transfer | ld | Load dword | ld rd, 40(rs1) | rd = [rs1 + 40] | I | 0000011 | 011 | n.a. |
| | sd | Store dword | sd rs2, 40(rs1) | [rs1 + 40] = rs2 | S | 0100011 | 011 | n.a. |
| | lw | Load word | - | - | I | 0000011 | 010 | n.a. |
| | lwu | Load unsgn word | - | - | I | 0000011 | 110 | n.a. |
| | sw | Store word | - | - | S | 0100011 | 010 | n.a. |
| | lh | Load half word | - | - | I | 0000011 | 001 | n.a. |
| | lhu | Load unsgn hword | - | - | I | 0000011 | 101 | n.a. |
| | sh | Store hword | - | - | S | 0100011 | 001 | n.a. |
| | lb | Load byte | - | - | I | 0000011 | 000 | n.a. |
| | lbu | Load unsgn byte | - | - | I | 0000011 | 100 | n.a. |
| | sb | Store byte | - | - | S | 0100011 | 000 | n.a. |
| | lui | Load upper imm | lui rd, 0x12345 | rd = 0x12345000 | U | 0110111 | n.a. | n.a. |
| | auipc | Add upper imm to PC | auipc rd, 0x12345 | rd = PC + 0x12345000 | U | 0010111 | n.a. | n.a. |
| Logical | and | And | and rd, rs1, rs2 | rd = rs1 & rs2 | R | 0110011 | 111 | 0000000 |
| | or | Or | or rd, rs1, rs2 | rd = rs1 \| rs2 | R | 0110011 | 110 | 0000000 |
| | xor | Exclusive or | xor rd, rs1, rs2 | rd = rs1 ^ rs2 | R | 0110011 | 100 | 0000000 |
| | andi | And imm | andi rd, rs1, imm | rd = rs1 & imm | I | 0010011 | 111 | n.a. |
| | ori | Or imm | ori rd, rs1, imm | rd = rs1 \| imm | I | 0010011 | 110 | n.a. |
| | xori | Xor imm | xori, rd, rs1, imm | rd = rs1 ^ imm | I | 0010011 | 100 | n.a. |
| Shift | sll | shift left logical | sll rd, rs1, rs2 | rd = rs1 << rs2 | R | 0110011 | 001 | 0000000 |
| | srl | shift right logical | srl rd, rs1, rs2 | rd = rs1 >> rs2 (zExt) | R | 0110011 | 101 | 0000000 |
| | sra | shr arithmetic | sra rd, rs1, rs2 | rd = rs1 >> rs2 (sExt) | R | 0110011 | 101 | 0100000 |
| | slli | shl logical imm | slli rd, rs1, imm | rd = rs1 << imm | I | 0010011 | 001 | 000000 |
| | srli | shr logical imm | srli rd, rs1, imm | rd = rs1 >> imm (zExt) | I | 0010011 | 101 | 000000 |
| | srai | shr arith imm | srai rd, rs1, imm | rd = rs1 >> imm (sExt) | I | 0010011 | 101 | 010000 |
| Conditional Branch | beq | branch if equal | beq rs1, rs2, offset | if (rs1==rs2) PC+=offset | SB | 1100011 | 000 | n.a. |
| | bne | branch if not equal | bne rs1, rs2, offset | if (rs1!=rs2) PC+=offset | SB | 1100011 | 001 | n.a. |
| | blt | br if less than | blt rs1, rs2, offset | if (rs1<rs2) PC+=offset | SB | 1100011 | 100 | n.a. |
| | bge | br if greater or eq | bge rs1, rs2, offset | if (rs1>=rs2) PC+=offset | SB | 1100011 | 101 | n.a. |
| | bltu | blt, unsigned | bltu rs1, rs2, offset | if (rs1<rs2) PC+=offset | SB | 1100011 | 110 | n.a. |
| | bgeu | bge, unsigned | bgeu rs1, rs2, offset | if (rs1>=rs2) PC+=offset | SB | 1100011 | 111 | n.a. |
| Unconditional Branch | jal | jump and link | jal rd, offset | rd=PC+4; PC+=offset | UJ | 1101111 | n.a. | n.a. |
| | jalr | jump and link reg | jalr rd, 100(rs1) | rd=PC+4; PC=rs1+100 | I | 1100111 | 000 | n.a. |

*References/CO-Insts.webp*

> 这张图不用抄，如果考到了相关指令会给其作用的解释，但是可以记一下每一种指令是什么 type
> 或者记录一下简单的 opcode?

- `lr.d/lr.w` load reserved `lr.d rd, (rs1)`
  - 从 `(rs1)` 加载到 `rd`
  - 并在硬件上保留一个 `reservation` 标记
- `sc.d/sc.w` store conditional `sc.d rd, rs2, (rs1)`
  - 检查 `reservation` 标记是否有效，如果有效才将 `rs2` 的值写入 `(rs1)`，并标记 `rd=0`

```
again:
    lr.d x10, (x20)
    sc.d x11, (x20), x23
    bne x11, x0, again  # branch again if store failed
    addi x23, x10, 0
```

```
    addi x12, x0, 1
again:
    lr.d x10, (x20)  # read lock
    bne x10, x0, again  # 设置了一个锁，解锁条件为 sd x0, 0(x20) 被执行，使得跳转条件不满足
    sc.d x11, (x20), x12  # attempt to store
    bne x11, x0, again
```

## 2.1.1 加载 32-bit 立即数

# 32-bit Constants

**Example 2.19 Loading a 32-bit constant**

- The 32-bit constant:

**0000 0000 0011 1101 0000** ***1***001 0000 0000 $(976*16^3+2304=4000000)_{10}$

- RISC V code:

lui s3, 976 # 976 decimal = 0000 0000 0011 1101 0000 binary

| | | | |
|---|---|---|---|
| 0000 0000 0000 0000 | 0000 0000 0000 0000 | 0000 0000 0011 1101 0000 | 0000 0000 0000 |

(The value of s3 afterward is: 0000 0000 0011 1101 0000 0000 0000 0000)

addi s3, s3, 2304 # 2304 decimal = 1001 0000 0000 binary

| | | | |
|---|---|---|---|
| 1111 1111 1111 1111 | 1111 1111 1111 1111 | 1111 1111 1111 1111 1111 | 1001 0000 0000 |

The value of s3 afterward is: 实际补码值2304 become -1792 / sign extended

| | | | |
|---|---|---|---|
| 0000 0000 0000 0000 | 0000 0000 0000 0000 | 0000 0000 0011 1100 1111 | 1001 0000 0000 ✗ |

S3 + 2304 become S3 - 1792

| | | | |
|---|---|---|---|
| 0000 0000 0000 0000 | 0000 0000 0000 0000 | 0000 0000 0011 1101 0000 | 1001 0000 0000 √ |

结果 + 4096（2^12）可以修正这个问题，所以就是bit 12 + 1



这样结果就是两个立即数的拼接!

## 2.2 寄存器表

| Register | ABI Name | Description | Saver |
|---|---|---|---|
| x0 | zero | Hard-wired zero | — |
| x1 | ra | Return address | Caller |
| x2 | sp | Stack pointer | Callee |
| x3 | gp | Global pointer | — |
| x4 | tp | Thread pointer | — |
| x5 | t0 | Temporary/alternate link register | Caller |
| x6–7 | t1–2 | Temporaries | Caller |
| x8 | s0/fp | Saved register/frame pointer | Callee |
| x9 | s1 | Saved register | Callee |
| x10–11 | a0–1 | Function arguments/return values | Caller |
| x12–17 | a2–7 | Function arguments | Caller |
| x18–27 | s2–11 | Saved registers | Callee |
| x28–31 | t3–6 | Temporaries | Caller |
| f0–7 | ft0–7 | FP temporaries | Caller |
| f8–9 | fs0–1 | FP saved registers | Callee |
| f10–11 | fa0–1 | FP arguments/return values | Caller |
| f12–17 | fa2–7 | FP arguments | Caller |
| f18–27 | fs2–11 | FP saved registers | Callee |
| f28–31 | ft8–11 | FP temporaries | Caller |



*References/CO-Regs.webp*

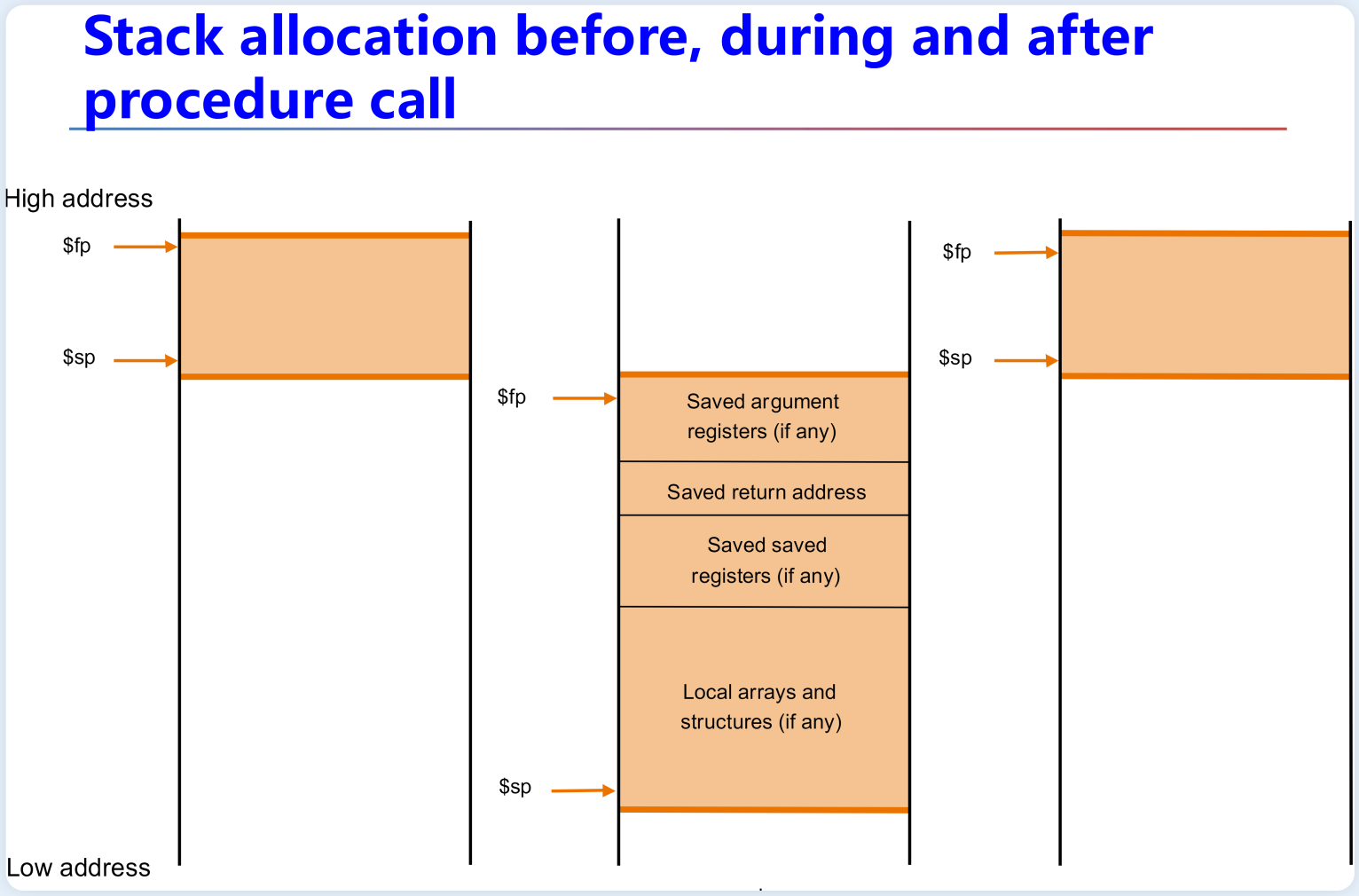
> - `ra` 在进入函数的第一步就要保护
> - `s` 类型寄存器使用前要保护

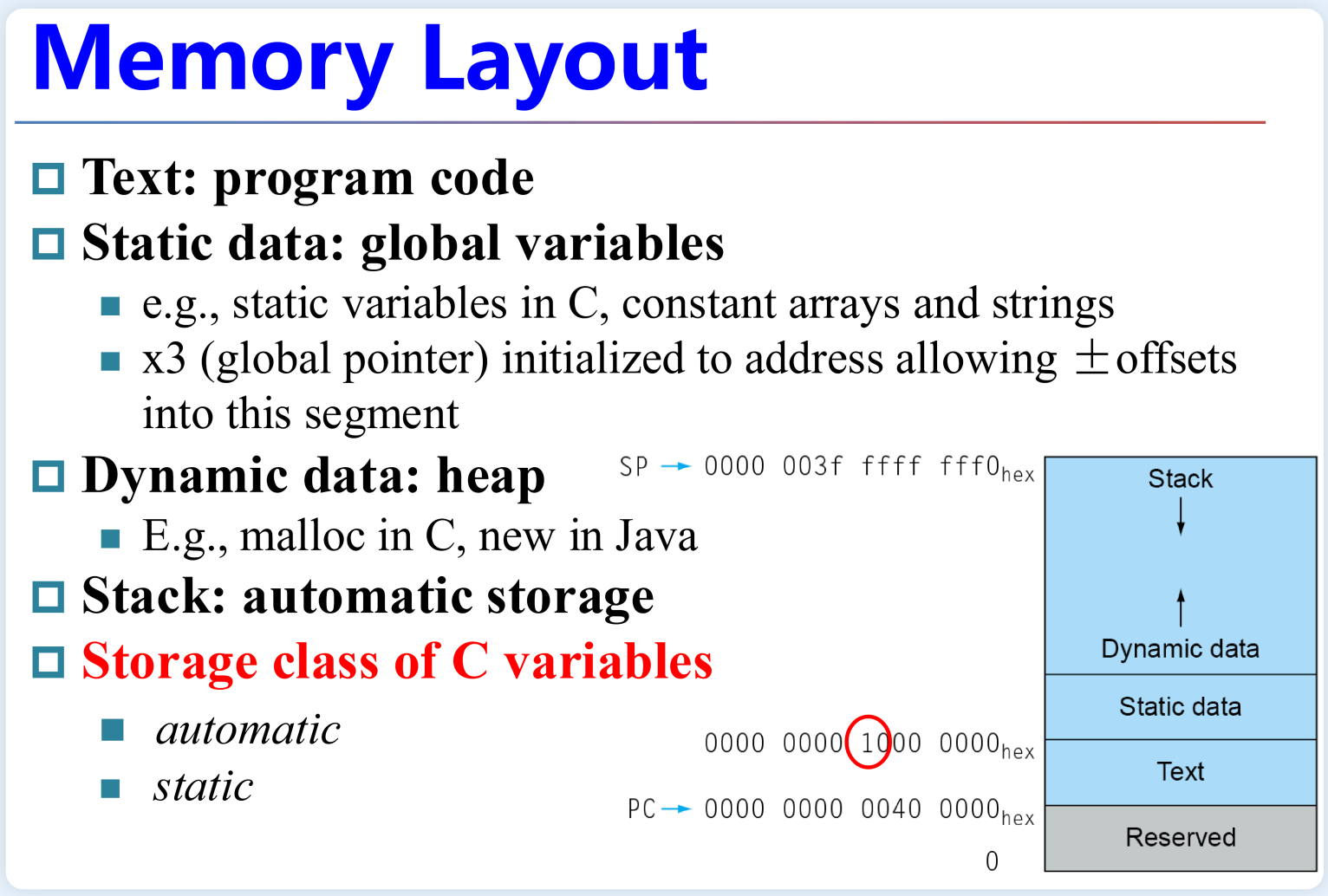
```
fact:
    addi sp, sp, -16  # 栈对齐，sp 分配空间的单位为 16 bytes = 128 bits
    sd ra, 8(sp)  # 保护返回地址，每个参数都对齐到 64 位 double word
    sd a0, 0(sp)  # 保护参数，便于递归
    addi t0, a0, -1
    bge t0, zero, L1  # 如果不是叶子，跳到 L1
    addi a0, zero, 1
    addi sp, sp, 16  # 堆栈平衡
    jalr zero, 0(ra)  # 返回
L1:
    addi a0, a0, -1
    jal ra, fact  # 递归调用
    add t1, a0, zero  # a0 为调用返回值
    ld a0, 0(sp)
    ld ra, 8(sp)  # 恢复参数
    add sp, sp, 16  # 堆栈平衡
```

```
    mul a0, a0, t1
    jalr zero, 0(ra)  # 返回
```

## 2.3 Memory layout

简单记一下就行 >




- little endians: least significant digit 的地址小，例如 `12345678h -> 78 56 34 12`
- fp: 栈帧指针
- sp: 栈顶指针

# 3 Chapter 3 Arithmetics

## 3.1 数制表示

- unsigned
- signed

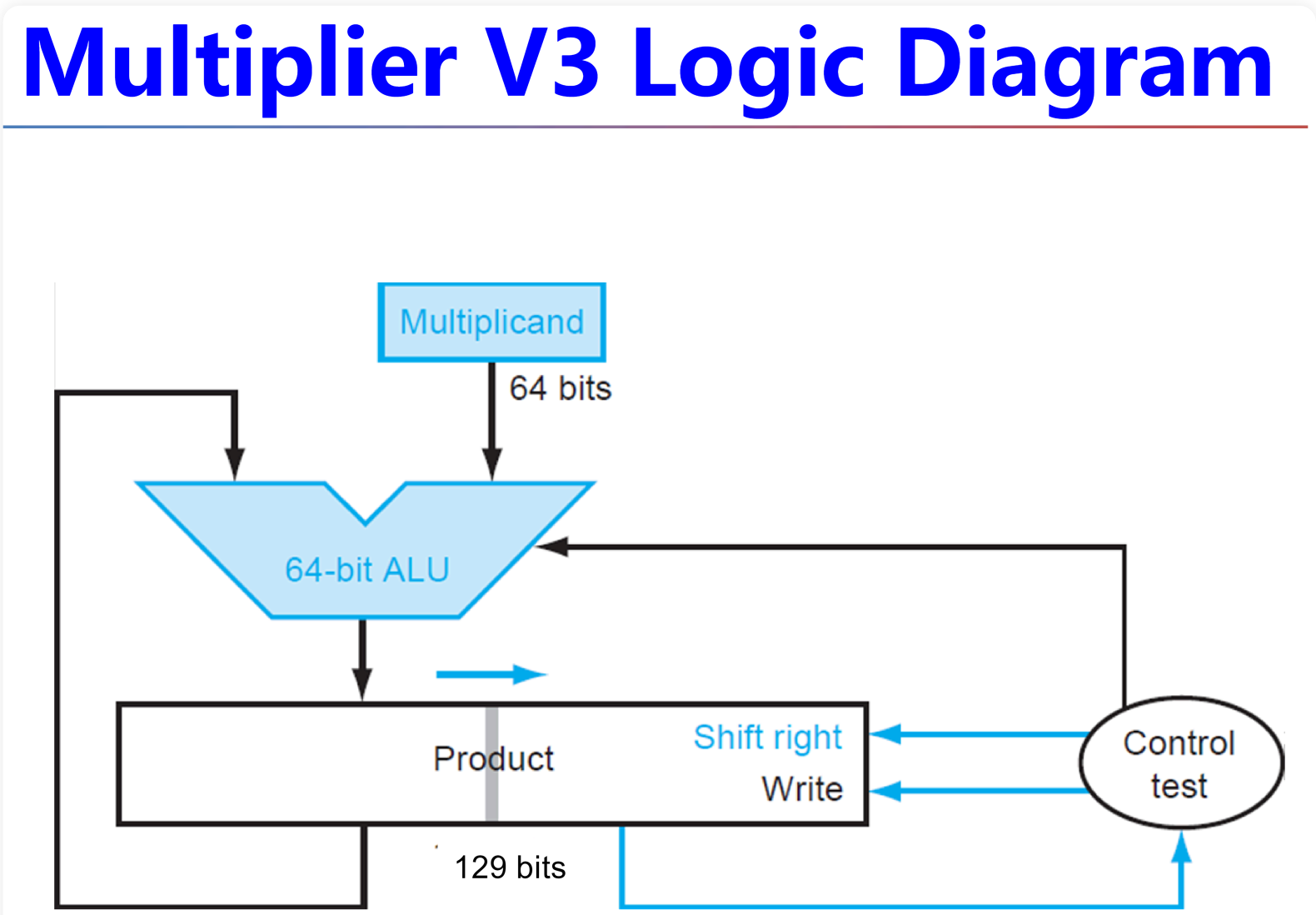
- sign-magnitude
- 2's complement
- *移码*，例如加 bias = -128

## 3.2 Add, Sub

- 减就是加上补码
- overflow
  - 负数相加得到正数
  - 正数相加得到负数

## 3.3 Mul

> CO 03 Arithmetic for Computer > 4 Multiplication

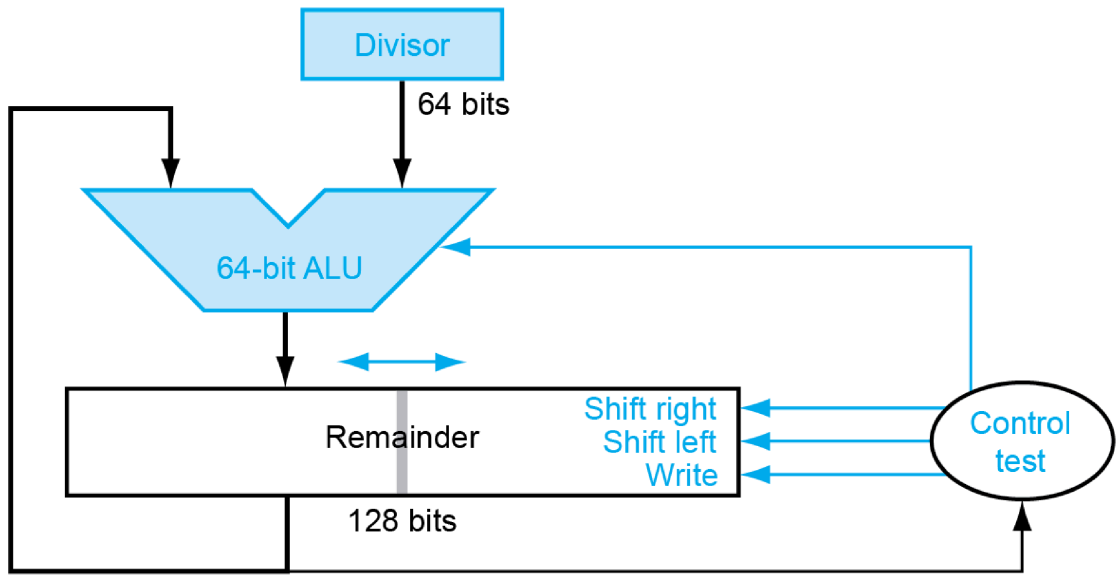



## 3.4 Div

> CO 03 Arithmetic for Computer > 5 Division

# Modified Division

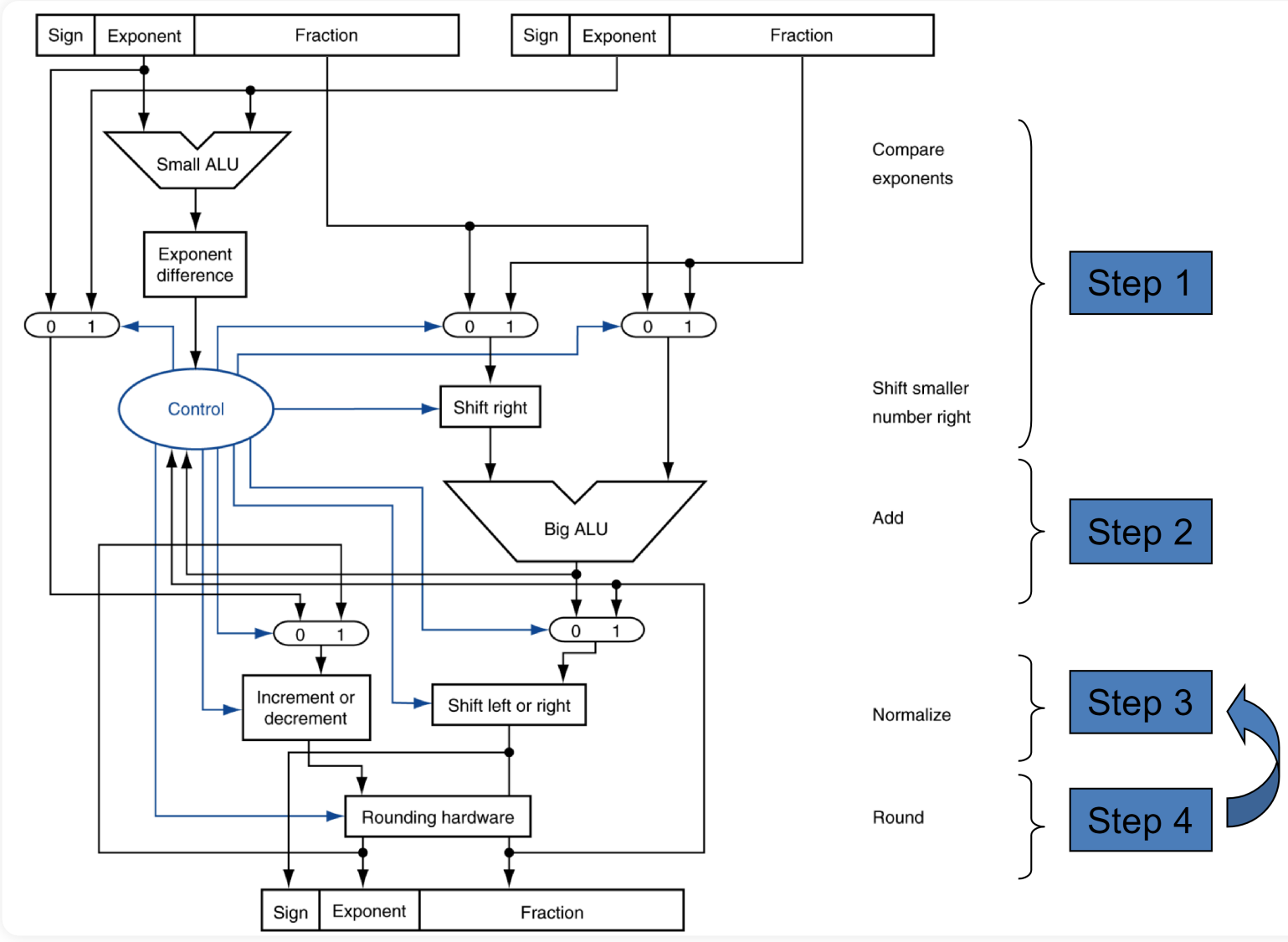
- Reduction of Divisor and ALU width by half
- Shifting of the remainder
- Saving 1 iteration
- Remainder register keeps quotient No quotient register required




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

## 3.5 Floating point

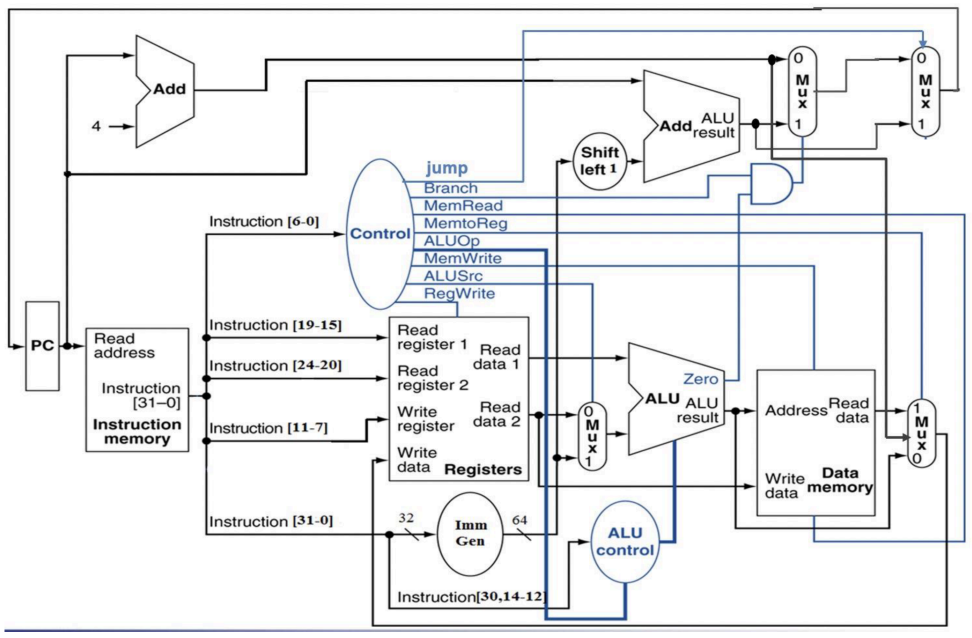
- IEEE 754 `sign | exp | frac`
  - 32-bit: `sign 1 | exp 8 | frac 23`
  - 64-bit: `sign 1 | exp 11 | frac 52`
  - infinity: `exp = 111...1, frac = 000...0` 无穷也分正负
  - nan: `exp = 111...1, frac != 000...0` illegal or undefined result
- rounding `guard (frac[-1]) | round (frac[-2]) | sticky (frac[-3])`
  - sticky 表示 round 后面是否还有 1，如果有 1 那么 sticky 就是 1
  - round to nearest even 精度是 0.5ulp



# 4 Chapter 4 CPU

## 4.1 Single cycle cpu

### Truth tables & Circuitry of main Controller



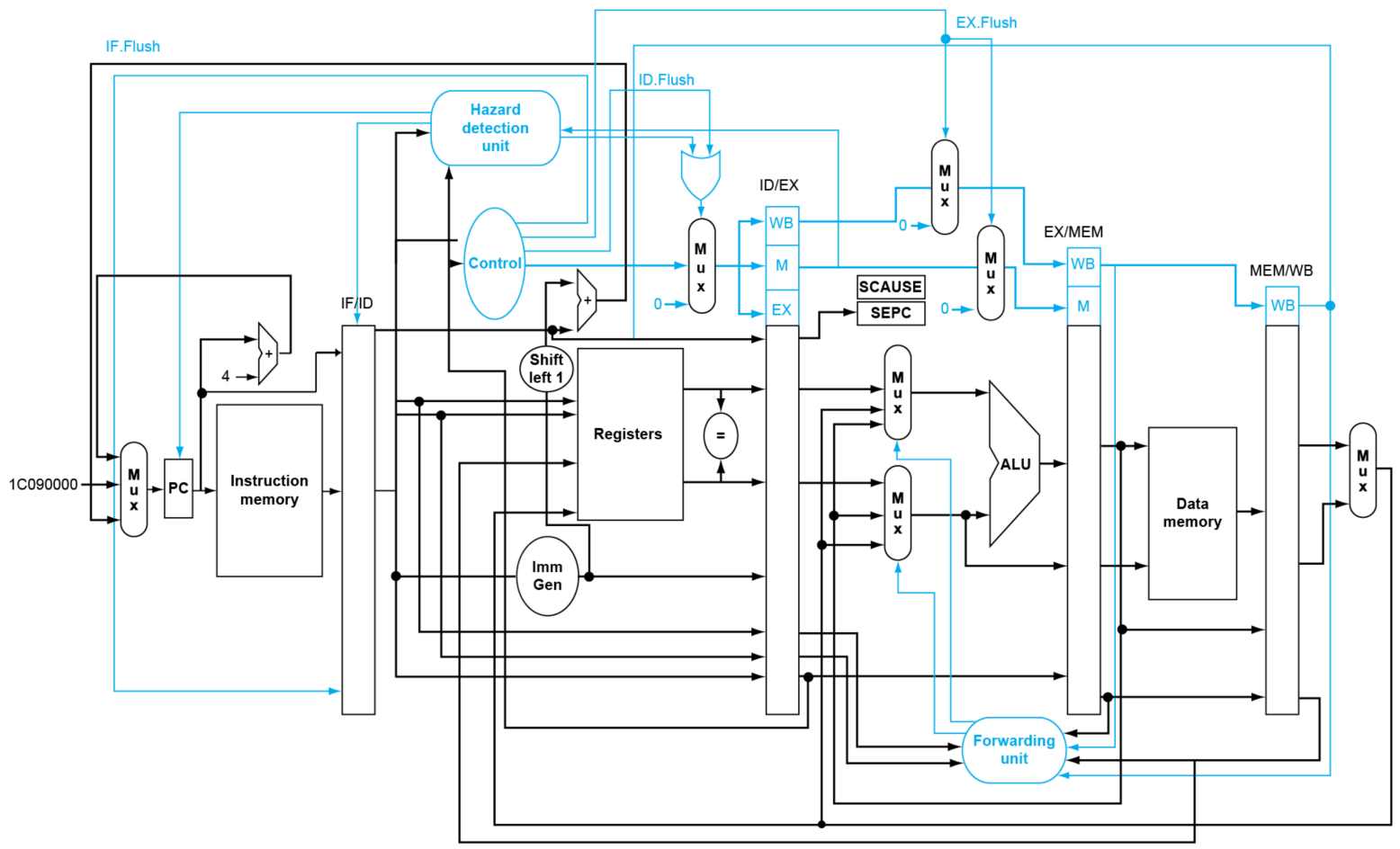

| 输入 | | 输出 | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Instruction | OPCode | ALUSrcB | Memto-Reg | Reg Write | Mem Read | Mem Write | Branch | Jump | ALU Op1 | ALU Op0 |
| R-format | 0110011 | 0 | 00 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| Ld(I-Type) | 0000011 | 1 | 01 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| sd(S-Type) | 0100011 | 1 | X | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| beq(SB-Type) | 1100111 | 0 | X | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| Jal(UJ-Type) | 1101111 | X | 10 | 1 | 0 | 0 | 0 | 1 | X | X |

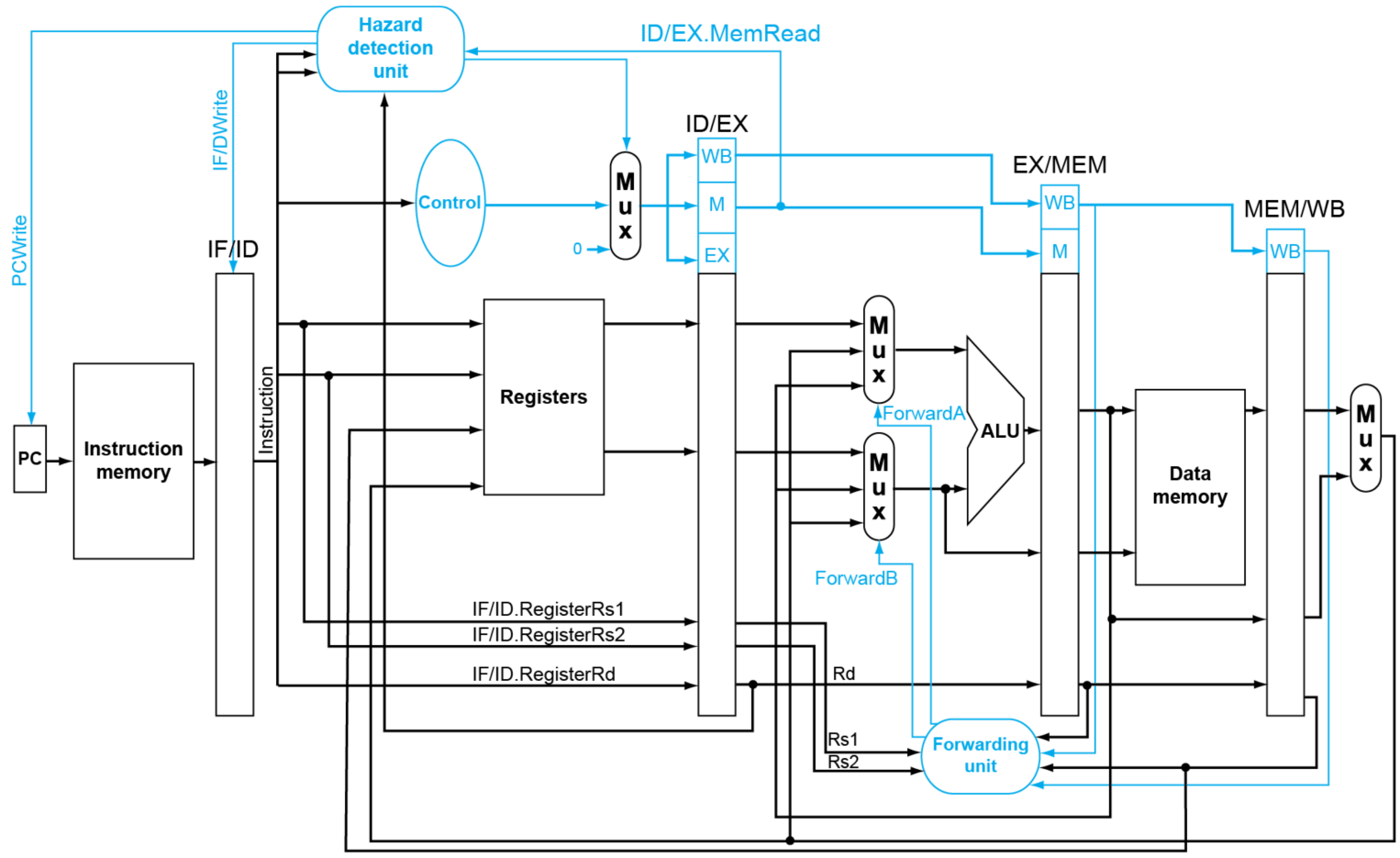
**题型**

- 信号是什么
- datapath 流程/需要的 unit
- 实现一定的指令扩展，信号？datapath?

## 4.2 Pipelined cpu

### Pipeline with Exceptions






- 多/单周期流水线示意图样例

## 4.2.1 Data Hazard

### 4.2.1.1 EX hazard

```
if (
    EX/MEM.RegWrite
    and (EX/MEM.Rd != 0)
    and (EX/MEM.Rd == ID/EX.Rs1)
) ForwardA = 10
```

### 4.2.1.2 Mem hazard *load use*

```
if (
    MEM/WB.RegWrite
    and (MEM/WB.Rd != 0)  // 有效 WB
    and not (
        EX/MEM.RegWrite
        and (EX/MEM.Rd != 0)  // 上一条指令有效 WB
        and (EX/MEM.Rd == ID/EX.Rs1)
    )  // 下一条指令不需要进行 EX hazard forwarding
    and (MEM/WB.Rd == ID/EX.Rs1)
) ForwardA = 01
```

### 4.2.1.3 Stall *load use*

```
if (
    ID/EX.MemRead
    and (
        (ID/EX.Rd == IF/ID.Rs1)
        or (ID/EX.Rd == ID/IF.Rs2)
    )
) stall the pipeline
```

> load use: stall + mem forwarding

## 4.2.2 Branch Hazard

- branch 可以提前到 ID，但是至少仍然产生一个 bubble
- dynamic prediction: 1/2-bit predictor
- branch target buffer，执行到 branch 就根据预测结果跳到 target 执行，不用再计算 pc

### 4.2.3 Exceptions

- `PC -> SPEC (Supervisor Exception Program Counter)`
- `ERROR CODE -> SCAUSE (Supervisor Exception Cause Register)`
- 跳转到 handler
  - `if restartalbe`: 修正错误，并返回到 `SPEC`
  - `else` 终止，向操作系统上报错误
- `PC` 前面需要扩展一个 `MUX` 来进行选择

**4.16** In this exercise, we examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies:

| IF | ID | EX | MEM | WB |
|---|---|---|---|---|
| 250 ps | 350 ps | 150 ps | 300 ps | 200 ps |

Also, assume that instructions executed by the processor are broken down as follows:

| ALU/Logic | Jump/Branch | Load | Store |
|---|---|---|---|
| 45% | 20% | 20% | 15% |

**4.16.1** [5] <§4.5> What is the clock cycle time in a pipelined and non-pipelined processor?

**4.16.2** [10] <§4.5> What is the total latency of an `ld` instruction in a pipelined and non-pipelined processor?

**4.16.3** [10] <§4.5> If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

**4.16.4** [10] <§4.5> Assuming there are no stalls or hazards, what is the utilization of the data memory?

**4.16.5** [10] <§4.5> Assuming there are no stalls or hazards, what is the utilization of the write-register port of the “Registers” unit?

# 5 Memory Hierarchy

- 金字塔
- disk access time 计算示例
- 几种表示例
- 几种 cache 结构示例
  - direct mapped
  - fully associative
  - set asso

## 5.1 Write

- write miss 如果目标不在内存里
  - write around: 直接写到 mem
  - write allocate: 先加载到 cache 再执行正常的写
- 正常的写操作 write strategy
  - write through: 总是写入到 mem
    - -> 问题：write stall -> 使用 write buffer 解决

    - write back 需要

## 5.2 Perf

- AMAT = hit time + miss time = hit time + miss rate * miss penalty
- CPU Time = CPU exe. clock cycles + mem-stall clock cycles

# 6 IO

- throughput & response time
- (average) disk access time
    - (average) seek time
    - rotational latency 平均转半圈
    - transfer time: 读取和传输一个 sector
    - disk controller: 控制器延迟