Computer System Architecture

— Computation, ISA and Datapath

陈鸿峥

December, 2018

1. 背景、历史、预备知识
2. 指令集系统概述
3. RISC-V指令集总览
4. 单周期CPU
5. 多周期CPU概述
6. RISC-V的内存模型
7. RISC-V扩展

1

背景、历史、预备知识

1.1

从一切的源头讲起—计算模型

一切的开端

我们必须知道，我们必将知道。
Wir müssen wissen, wir werden wissen.
—大卫希尔伯特(David Hilbert),1930

一切的开端

我们必须知道，我们必将知道。
Wir müssen wissen, wir werden wissen.
—大卫希尔伯特(David Hilbert),1930

希尔伯特23个问题：第二个问题 — 算术系统的相容性

1. 数学是完备的吗(complete)?
2. 数学是一致的吗(consistent)?
3. 数学是可判定的吗(decidable)?

一切的开端

我们必须知道，我们必将知道。
Wir müssen wissen, wir werden wissen.
—大卫希尔伯特(David Hilbert),1930

希尔伯特23个问题：第二个问题 — 算术系统的相容性

1. 数学是完备的吗(complete)?
2. 数学是一致的吗(consistent)?
3. 数学是可判定的吗(decidable)?

- 哥德尔(Gödel)不完备性定理, 1931

一切的开端

我们必须知道，我们必将知道。
Wir müssen wissen, wir werden wissen.
—大卫希尔伯特(David Hilbert),1930

希尔伯特23个问题：第二个问题 — 算术系统的相容性

1. 数学是完备的吗(complete)?
2. 数学是一致的吗(consistent)?
3. 数学是可判定的吗(decidable)?

- 哥德尔(Gödel)不完备性定理, 1931
  - **形式化武器**

一切的开端

我们必须知道，我们必将知道。
Wir müssen wissen, wir werden wissen.
—大卫希尔伯特(David Hilbert),1930

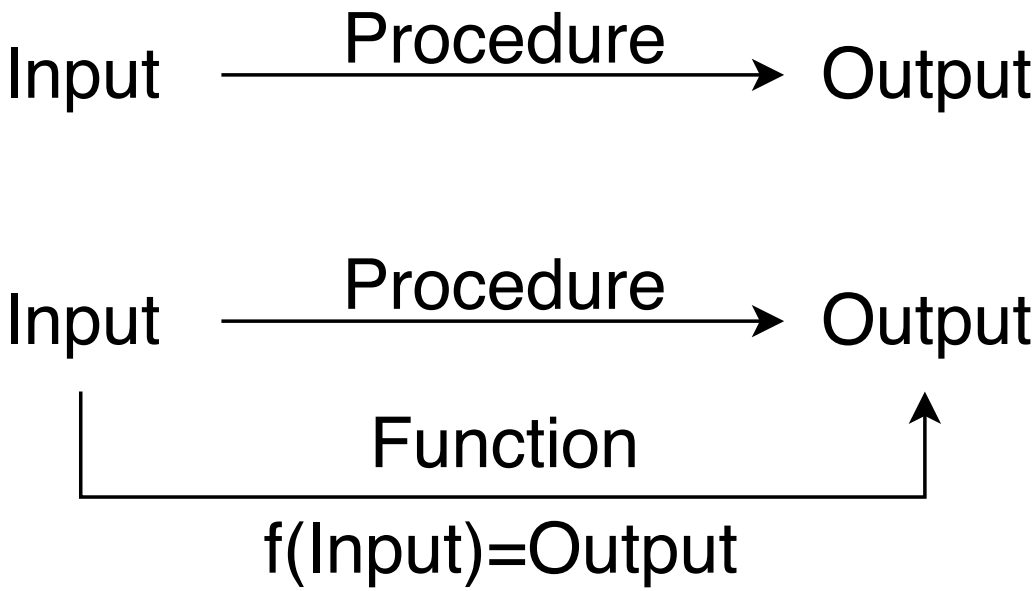
希尔伯特23个问题：第二个问题 — 算术系统的相容性

1. 数学是完备的吗(complete)?
2. 数学是一致的吗(consistent)?
3. 数学是可判定的吗(decidable)?

- 哥德尔(Gödel)不完备性定理, 1931
  - **形式化**武器
  - **一阶谓词逻辑**是完备的

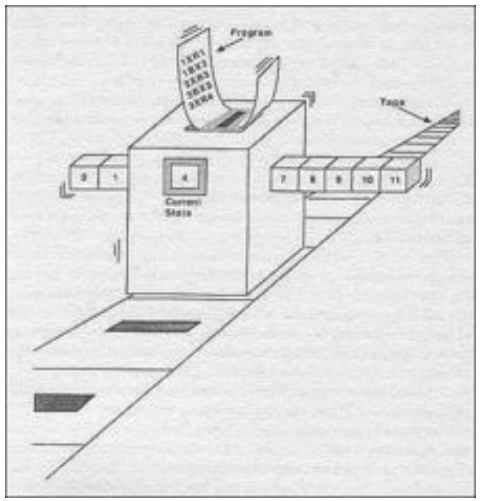
什么是计算？

什么是计算？


Input
Procedure
Output
Input
Procedure
Output
Function
f(Input)=Output

图灵机

Alan Turing, *On Computable Numbers, With an Application to the Entscheidungsproblem*, 1936



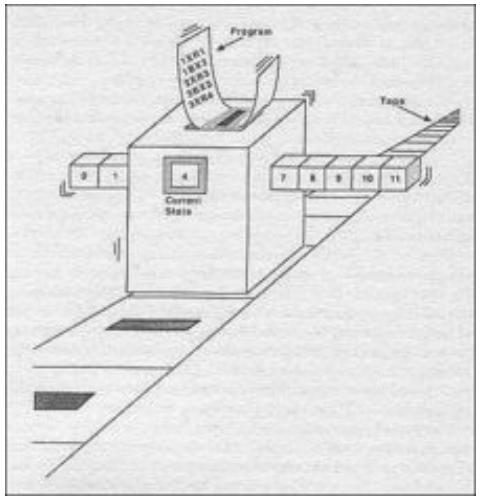
- 纸带(tape)
- 读写头(head)
- 状态寄存器(state register)
- 有限状态表(table)

基本操作:

- 读符号
- 不修改或者写符号
- 左移或右移纸带

图灵机

Alan Turing, *On Computable Numbers, With an Application to the Entscheidungsproblem*, 1936

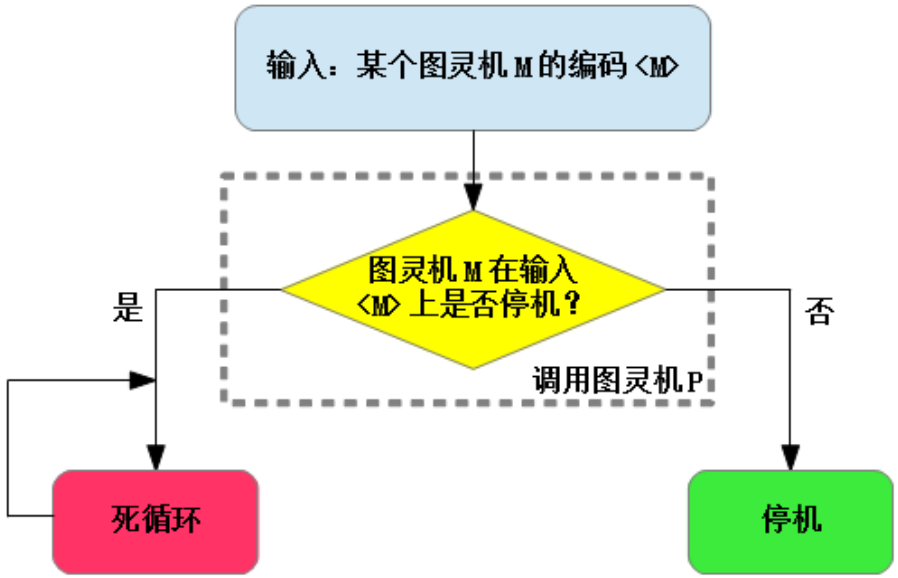


停机问题：是否存在这样的图灵机能够判断一个程序在给定输入上是否会停机/结束？

图灵机

Alan Turing, *On Computable Numbers, With an Application to the Entscheidungsproblem*, 1936

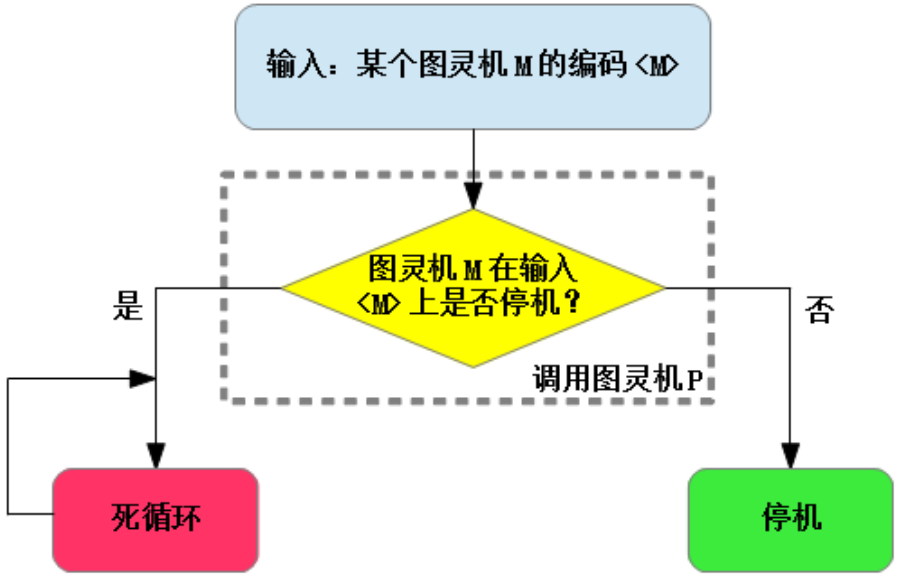
停机问题：是否存在这样的图灵机能够判断一个程序在给定输入上是否会停机/结束？



图灵机

Alan Turing, *On Computable Numbers, With an Application to the Entscheidungsproblem*, 1936

停机问题：是否存在这样的图灵机能够判断一个程序在给定输入上是否会停机/结束？



即使是完备的数学系统，也是不可判定的！

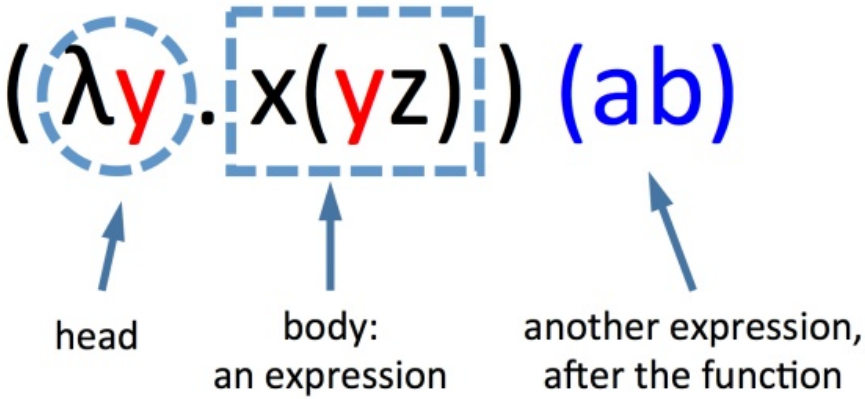
λ-Calculus

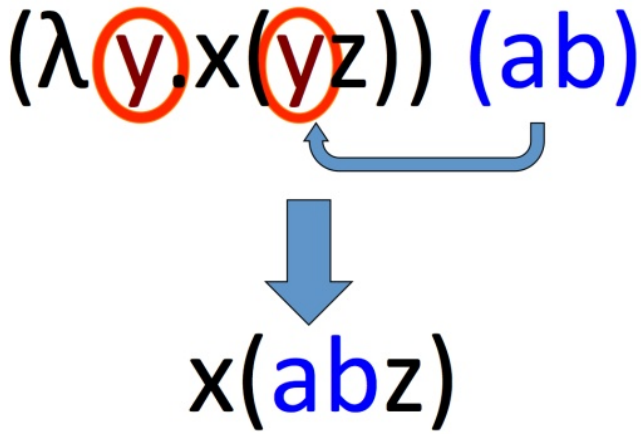
Alonzo Church, *An unsolvable problem of elementary number theory*, 1936

- 变量(variable)：x
- 抽象(abstraction)：$(\lambda x.M)$
- 应用(application)：(MN)

Operation:

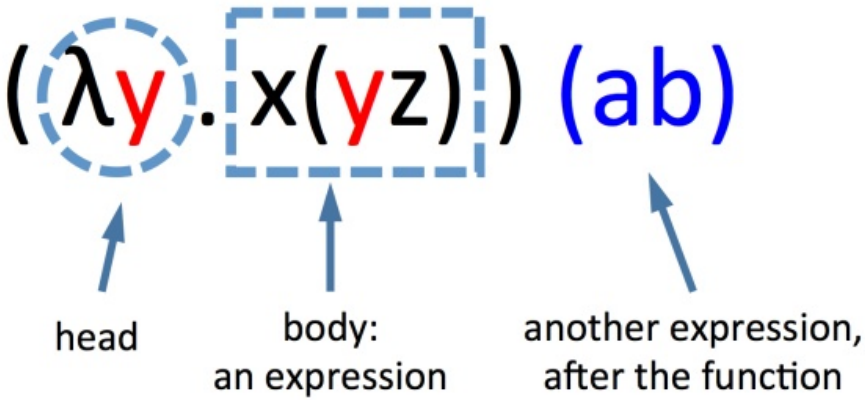
- α转换(conversion)：$(\lambda x.M[x]) \to (\lambda y.M[y])$
- β规约(reduction)：$((\lambda x.M)E \to (M[x \leftarrow E]))$

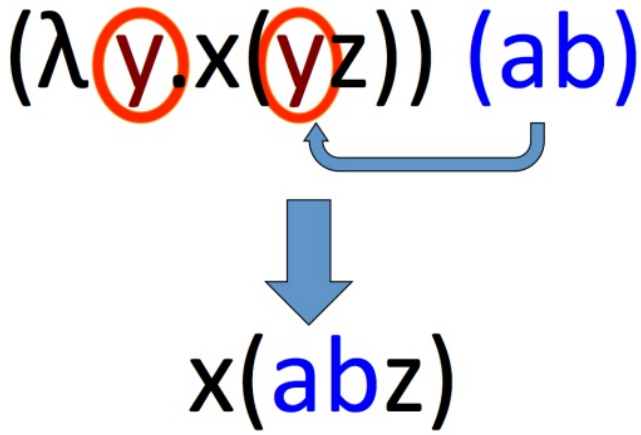




λ-Calculus

Alonzo Church, *An unsolvable problem of elementary number theory*, 1936





没有数据和程序之分!
一切都是λ项，它们既是程序，也是数据

图灵完备

图灵机与λ-演算等价!（图灵完备）

图灵完备

图灵机与λ-演算等价!（图灵完备）

命令式语言(imperative)	函数式语言(functional)
图灵系	丘奇系
面向计算机硬件的抽象	面向数学的抽象
指令序列	表达式（函数是一等公民）
C/C++/Python/Java	Lisp/Haskell/ML/Coq
顺序执行 有副作用，依赖外部环境（如IO） 变量对应着存储单元	对求值顺序不相关，易于并行 无副作用，纯函数，不出现竞争冒险 变量只是代数名称

图灵完备

图灵机与λ-演算等价!（图灵完备）

命令式语言(imperative)	函数式语言(functional)
图灵系	丘奇系
面向计算机硬件的抽象	面向数学的抽象
指令序列	表达式（函数是一等公民）
C/C++/Python/Java	Lisp/Haskell/ML/Coq
顺序执行 有副作用，依赖外部环境（如IO） 变量对应着存储单元	对求值顺序不相关，易于并行 无副作用，纯函数，不出现竞争冒险 变量只是代数名称

若不考虑内存的限制，绝大多数编程语言都是图灵完备的

计算模型

什么是可计算的？(Computability)

计算模型

什么是可计算的？(Computability)

Church-Turing Thesis
所有可以有效计算的函数都可以被通用图灵机计算

- **机械计算**就是图灵机能做的计算 ← 可计算理论的基石
- **目前**任何计算装置（乃至大脑、超算）都不能超过图灵机的能力（不考虑速度，只考虑可计算性）

怎么去计算？

λ-演算更像是智慧的推理

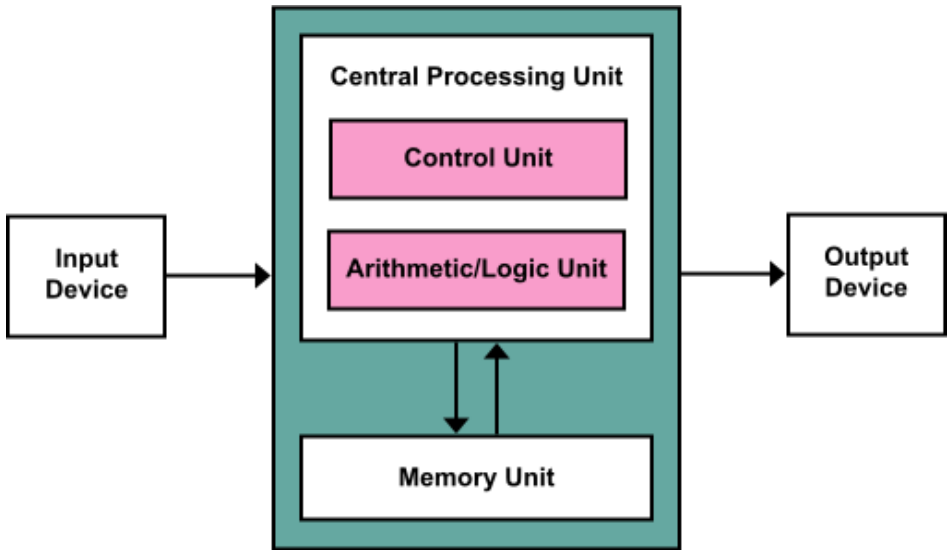
而图灵机真正抓住了 **机械计算** 的神韵

1.2

冯诺依曼体系结构

冯诺依曼体系结构

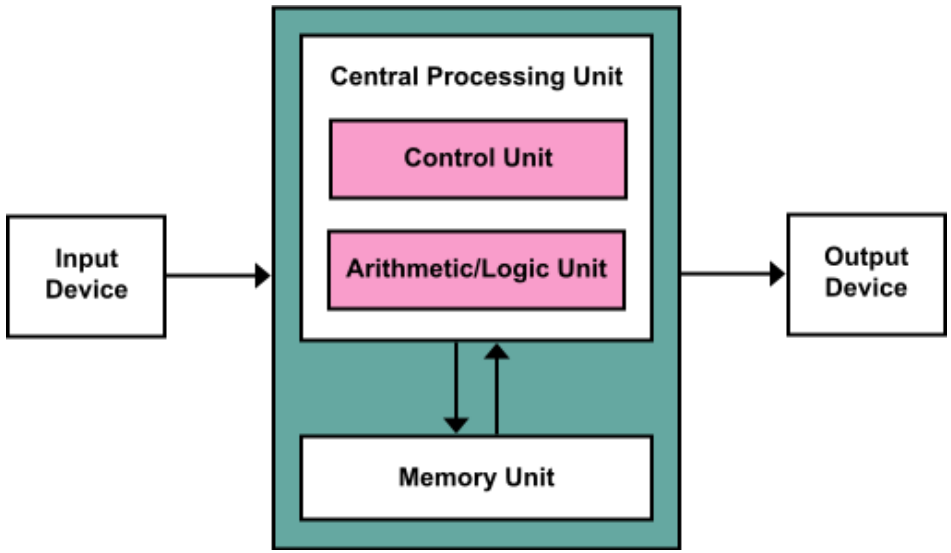
- 图灵机是**理论**的最简单的模型，但具有非常强的计算力
- 冯诺依曼(von Neumann)体系结构则是对图灵机的物理**实现**(EDVAC, 1945)



- **存储器！二进制！**

冯诺依曼体系结构

- 图灵机是**理论**的最简单的模型，但具有非常强的计算力
- 冯诺依曼(von Neumann)体系结构则是对图灵机的物理**实现**(EDVAC, 1945)



- **存储器！二进制！**
- 所有指令与数据都**无区别**地存储在计算机中

1.3

数据的表示与大小

数据的表示

数据的表示

- 原码(sign-magnitude)：最高位为符号位
- 反码(**ones'** complement)：负数按位取反
- 补码(**two's** complement)：模2^n运算，反码$+1$

进制

- 二进制0b10
- 八进制034
- 十六进制0xFF或FFH

基本数据单位

数据单位

- 位(bit)：处理信息的最小单位
- 字节(Byte)：存储的基本单位，$1\text{B}=8\text{bit}$
- 字(Word)：随机器而变，32位机 $1\text{Word}=4\text{B}=32\text{bit}$
- 字长(Word length)：数据通路的宽度/字的长度

内存大小

$$1\text{KiB}=2^{10}\text{B} \qquad 1\text{MiB}=2^{10}\text{KiB} \qquad 1\text{GiB}=2^{10}\text{MiB}$$

2

指令集系统概述

指令集背景

Machine Language (Binary)[1945]

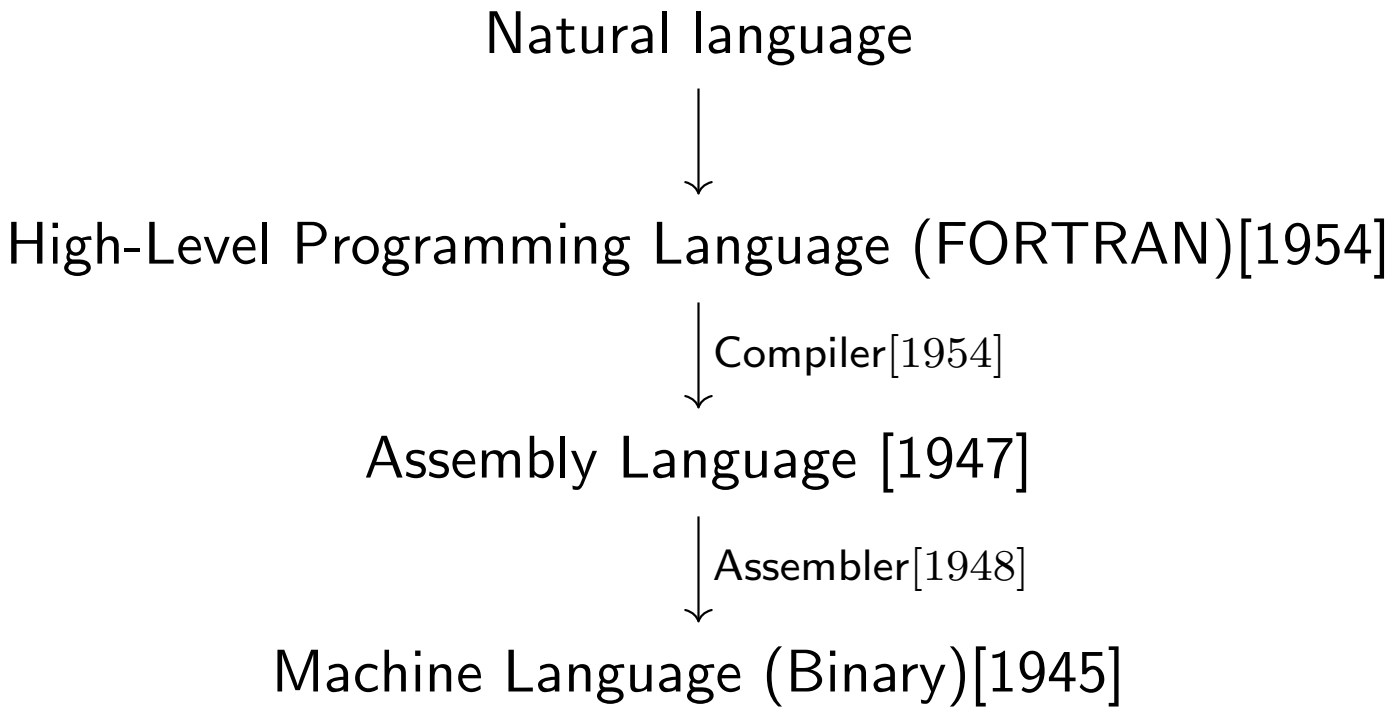
指令集背景

Assembly Language [1947]

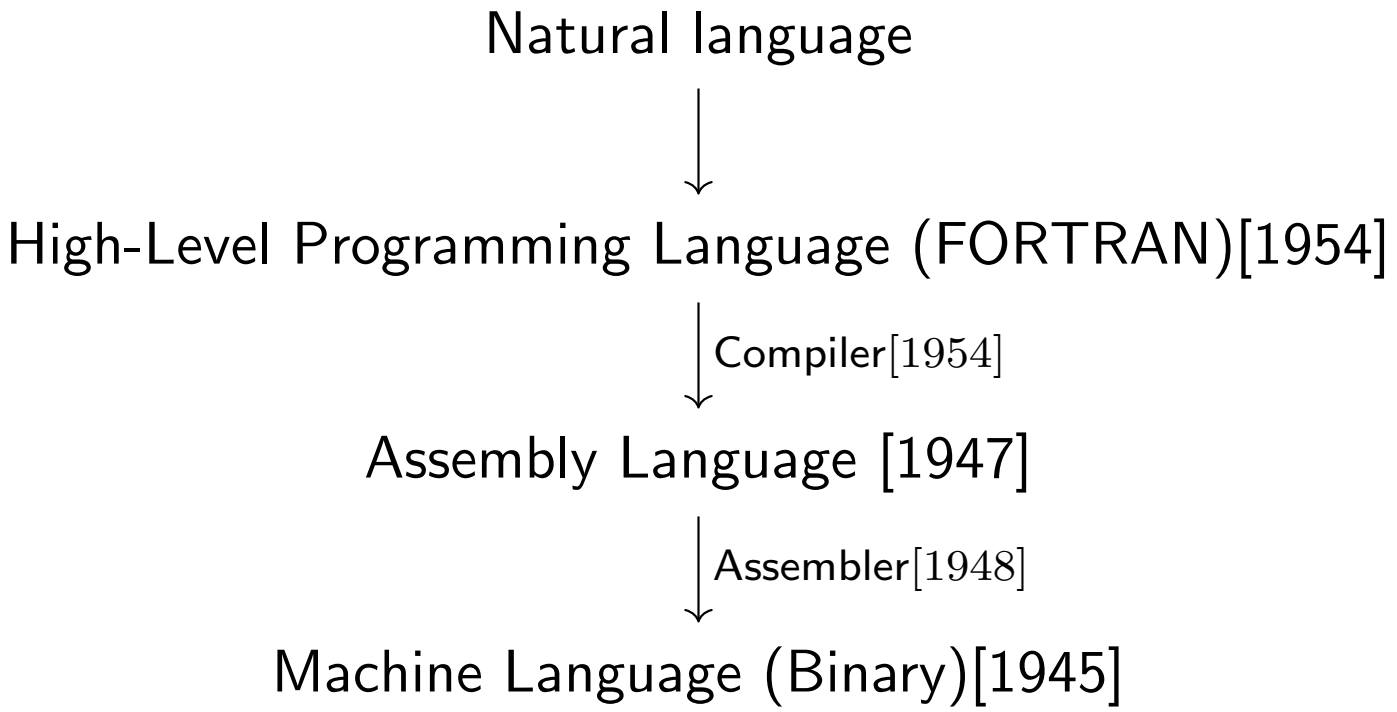
$\downarrow Assembler[1948]$

Machine Language (Binary)[1945]

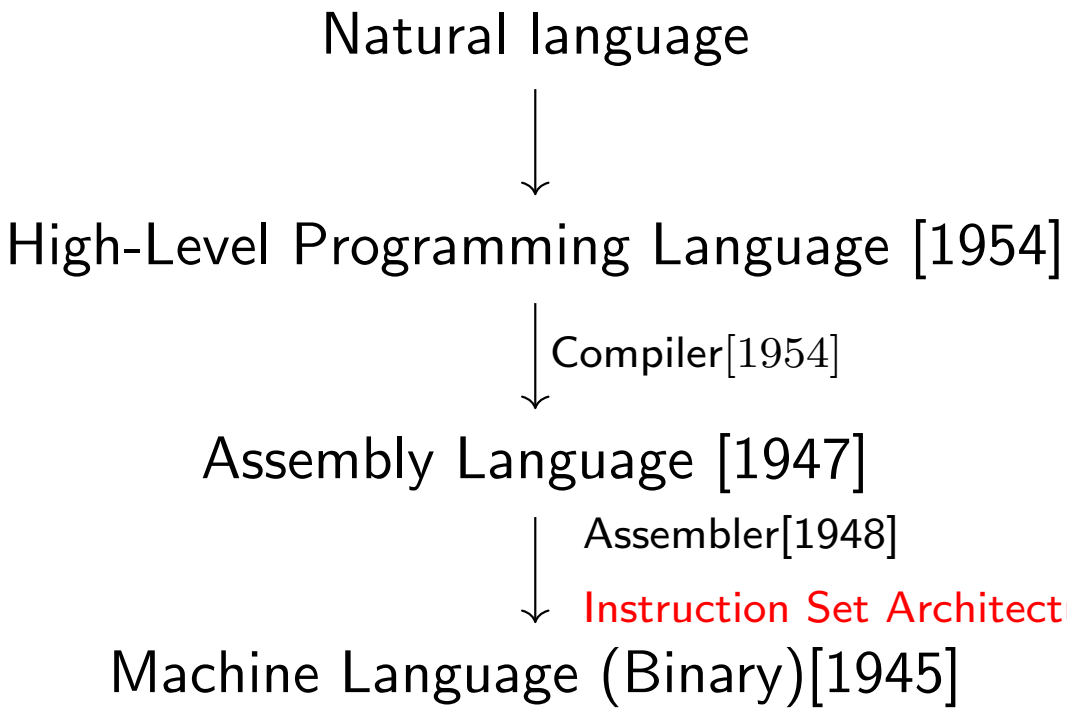
指令集背景


Natural language
High-Level Programming Language (FORTRAN)[1954]
Compiler[1954]
Assembly Language [1947]
Assembler[1948]
Machine Language (Binary)[1945]

指令集背景


Natural language
High-Level Programming Language (FORTRAN)[1954]
Compiler[1954]
Assembly Language [1947]
Assembler[1948]
Machine Language (Binary)[1945]

指令集背景



Assembly:
add $1, $0, 1
sw $1, $0(4)

Machine:
0000 0010 1100 0111
0001 1101 0100 1000

指令系统概述

- ISA is the hardware/software interface
  - Defines set of programmer visible state
  - Defines data types
  - Defines instruction semantics (operations, sequencing)
  - Defines instruction format (bit encoding)
  - Examples: *MIPS, Alpha, x86, IBM 360, VAX, ARM, JVM*
- Many possible implementations of one ISA
  - 360 implementations: model 30 (c. 1964), zEnterprise196 (c. 2010)
  - x86 implementations: *8086 (c. 1978), 80186, 286, 386, 486, Pentium, Pentium Pro, Pentium-4, Core i7, AMD Athlon, AMD Opteron, Transmeta Crusoe, SoftPC*
  - MIPS implementations: *R2000, R4000, R10000, ...*
  - JVM: *HotSpot, PicoJava, ARM Jazelle, ...*

指令系统的作用

- 硬件设计者角度：
  - ISA为CPU提供功能需求
  - ISA设计目标：易于硬件逻辑设计
- 系统程序员角度：
  - 通过ISA使用硬件资源
  - ISA设计目标：易于编写编译器

ISA determines the performance and the cost of computers

指令集计算机

复杂指令集计算机	精简指令集计算机
CISC, Complex Instruction Set Computer	RISC, Reduce Instruction Set Computer
出现较早(1970s)，大而全	出现较晚(1980s)，小而精
指令周期长，专用寄存器，微程序控制，难编译优化生成高效目标代码，效率低	指令周期短，大量通用寄存器，组合逻辑电路控制，优化编译系统
变长指令字	定长指令字
x86	ARM(Advanced RISC Machine), MIPS, SPARC

处理器性能评价

$$\frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} * \frac{\text{Cycles}}{\text{Instruction}} * \frac{\text{Time}}{\text{Cycle}}$$

- Instructions per program depends on source code, compiler technology and ISA
- Cycles per instructions (CPI) depends upon the ISA and the microarchitecture
- Time per cycle depends upon the microarchitecture and the base technology

rest of this lecture →

Microarchitecture	CPI	cycle time
Microcoded	>1	short
Single-cycle unpipelined	1	long
Pipelined	1	short

指令集系统

1. 什么操作？操作码
2. 操作对象？操作数
3. 如何找到操作对象？寻址方式

指令=操作码+地址码

指令集系统

1. 数据传送指令：store、load
2. 算术运算指令：加减乘除
3. 逻辑运算指令：与或非
4. 输出输出指令：I/O
5. 系统控制指令：启动IO设备、存取特殊寄存器指令
6. 程序控制指令：转移、跳转、返回、中断

3

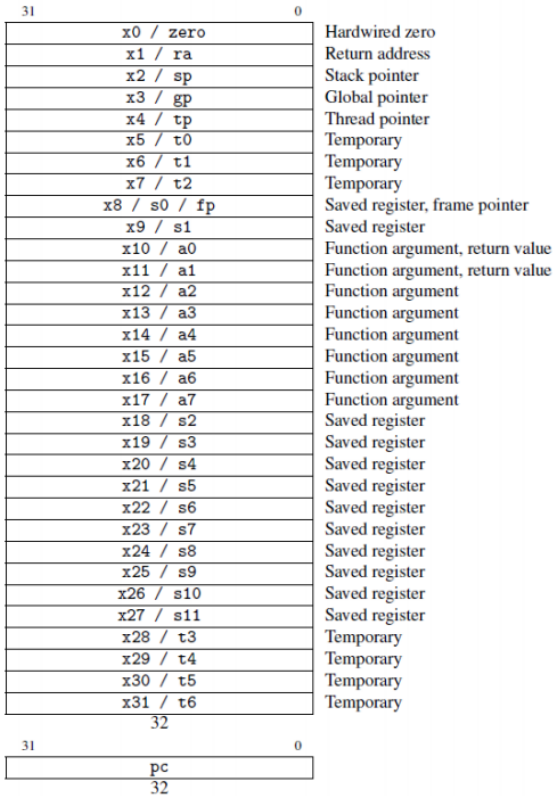
RISC-V指令集总览

RISC-V指令集

- 所有指令都是32位宽(4字节)，按字地址对齐
- 基本的RV32I共47条指令

Registers

x1-x31, x0=0, Program Counter (PC)



31 — 0	
x0 / zero	Hardwired zero
x1 / ra	Return address
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporary
x6 / t1	Temporary
x7 / t2	Temporary
x8 / s0 / fp	Saved register, frame pointer
x9 / s1	Saved register
x10 / a0	Function argument, return value
x11 / a1	Function argument, return value
x12 / a2	Function argument
x13 / a3	Function argument
x14 / a4	Function argument
x15 / a5	Function argument
x16 / a6	Function argument
x17 / a7	Function argument
x18 / s2	Saved register
x19 / s3	Saved register
x20 / s4	Saved register
x21 / s5	Saved register
x22 / s6	Saved register
x23 / s7	Saved register
x24 / s8	Saved register
x25 / s9	Saved register
x26 / s10	Saved register
x27 / s11	Saved register
x28 / t3	Temporary
x29 / t4	Temporary
x30 / t5	Temporary
x31 / t6	Temporary

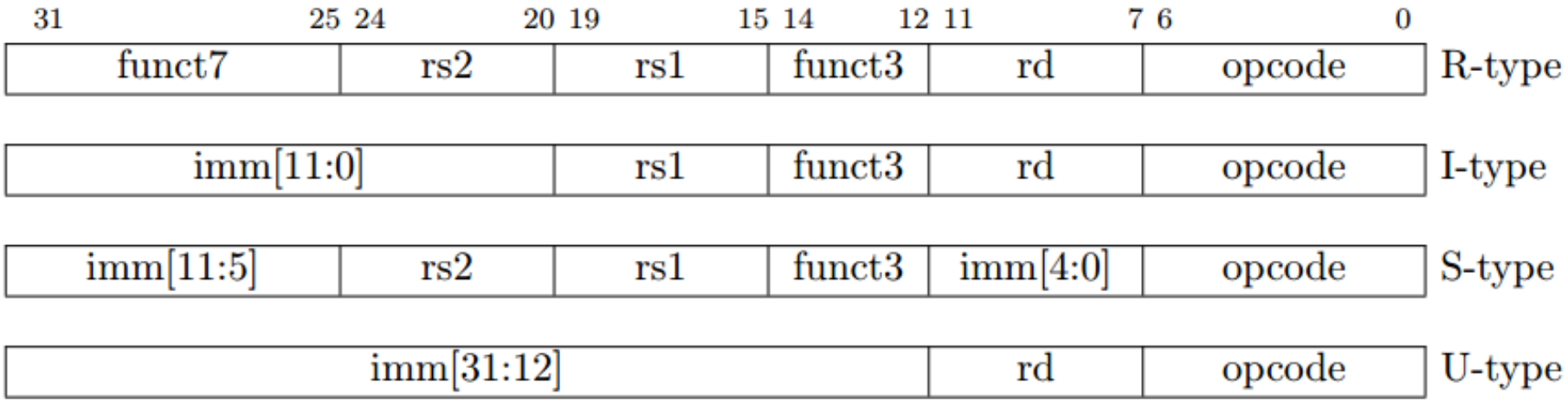
32

31 — 0
pc

32

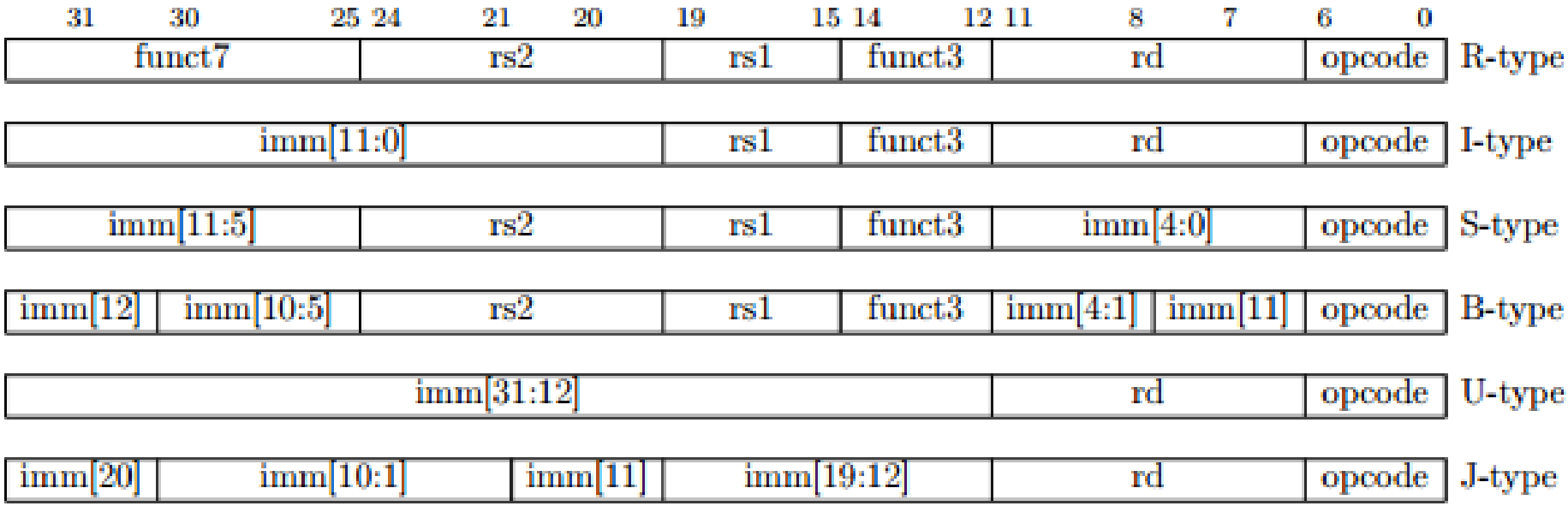
- RISC-V共32个通用寄存器，有零寄存器，且PC不是通用寄存器
- x86-32仅8个寄存器，且无零寄存器
- arm-32有16个寄存器，但PC不作为单独寄存器

Instruction Formats



- rs1, rs2, rd are at the same position (MIPS is not)
- Immediates are put leftmost, sign 31 (sign-extension can be done before ID)
- 12bits regular + 20bits load upper
- Only has sign-extended imm
- The last bit used for extension (64-bit)

Instruction Formats



- rs1, rs2, rd are at the same position (MIPS is not)
- Immediates are put leftmost, sign 31 (sign-extension can be done before ID)
- 12bits regular + 20bits load upper
- Only has sign-extended imm
- The last bit used for extension (64-bit)

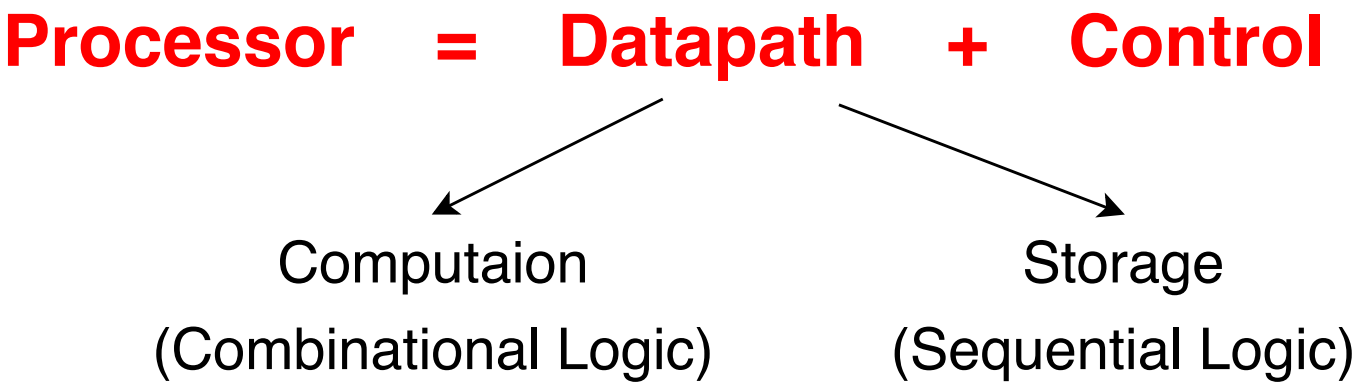
4

单周期CPU

4.1

Introduction

Central Processor Unit (CPU)



CPU指令执行过程

2-phases:

1. Fetch phase
2. Execute phase

5-stages CPU:

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. Execution (EXE)
4. Access memory (MEM)
5. Write back (WB)

核心指令

Main instructions:

1. `add rd, rs1, rs2`
2. `addi rd, rs, imm`
3. `lw rd, rs(imm)`
4. `sw rs, rd(imm)`
5. `beq rd, rs, offset`
6. `j address`

Examples:

- `add $3, $2, $1`
- `add $3, $2, 10`
- `lw $3, $2(4)`
- `sw $3, $2(4)`
- `beq $3, $2, -2`
- `j 0x00000050`

4.2

Some Basic Modules

All problems in computer science can be solved by another

abstraction layer.

隐藏low-level细节，给high-level提供简单模型

布尔逻辑层

0-1: 布尔运算能够用物理器件模拟（如继电器、二极管等等）

布尔逻辑层

0-1: 布尔运算能够用物理器件模拟（如继电器、二极管等等）

物理电路层→布尔逻辑层

计算器件层

1-2: 基本数学运算能够用布尔运算模拟

A	B	A+B	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

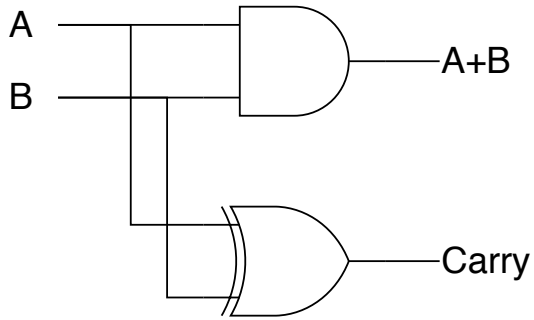
计算器件层

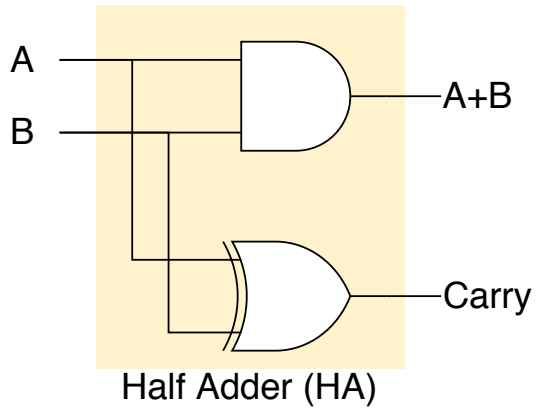
1-2: 基本数学运算能够用布尔运算模拟

A	B	A+B	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

$$A + B = A \oplus B$$

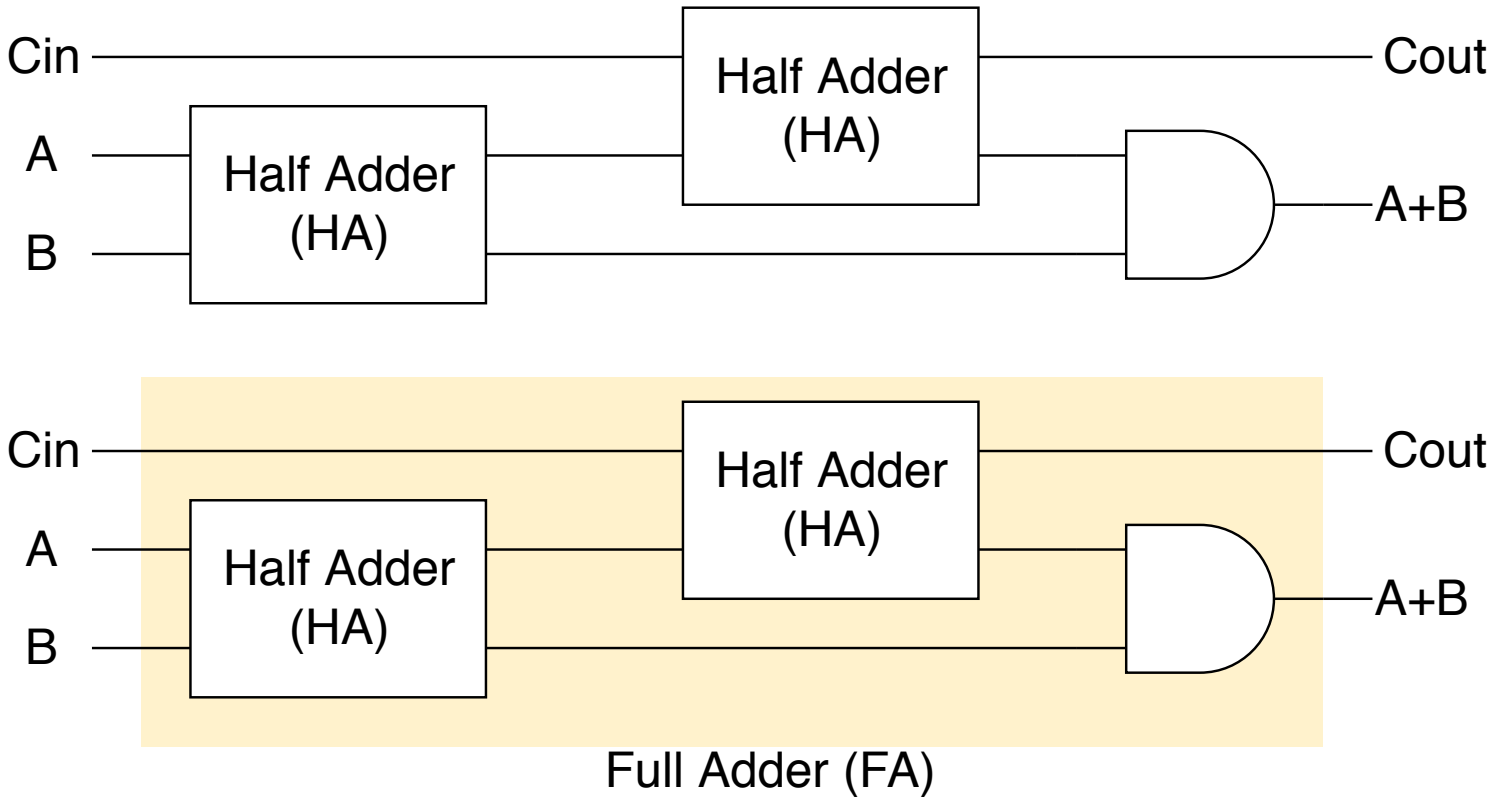
$$Carry = AB$$





计算器件层

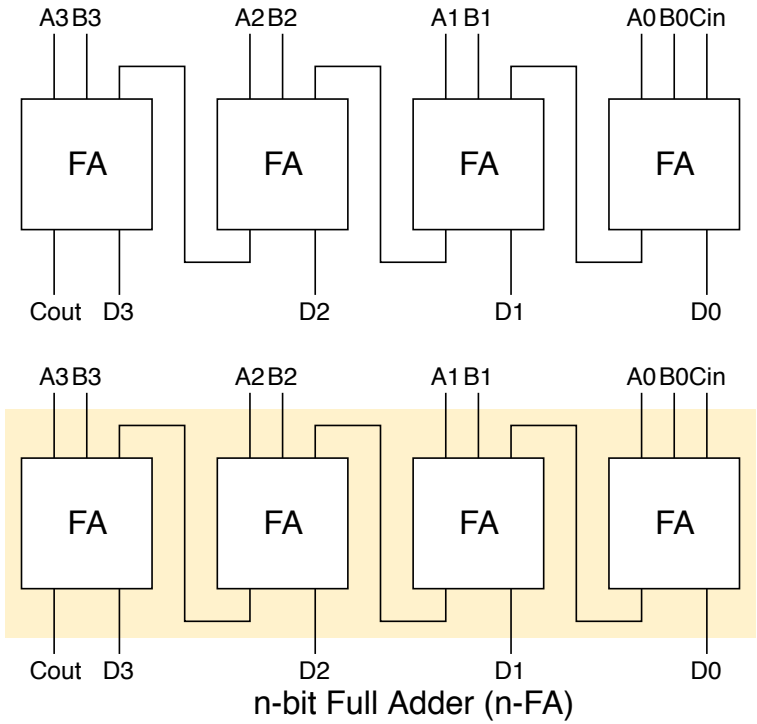
1-2: 基本数学运算能够用布尔运算模拟



物理电路层→布尔逻辑层→计算器件（**组合逻辑**）

计算器件层

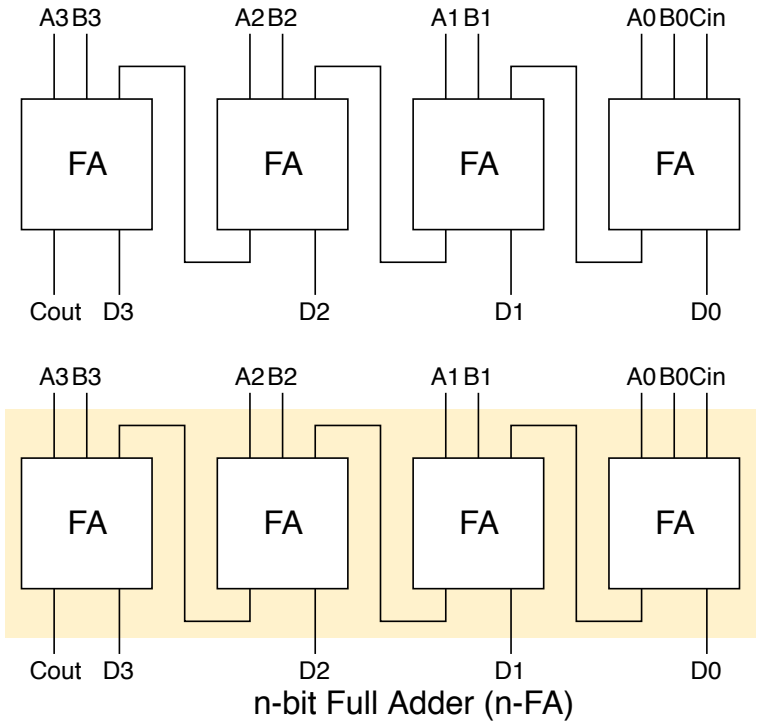
1-2: 基本数学运算能够用布尔运算模拟



物理电路层→布尔逻辑层→计算器件（**组合逻辑**）

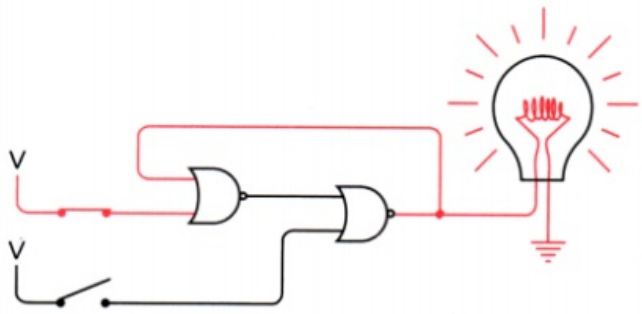
计算器件层

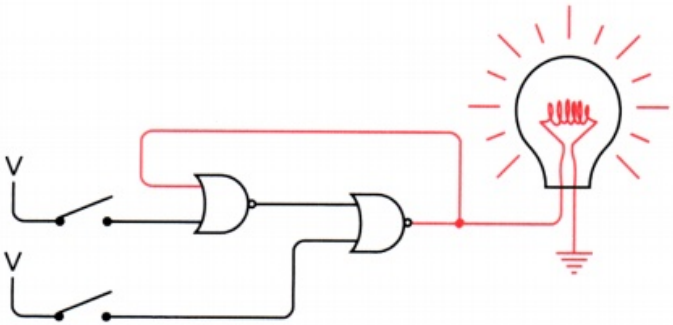
1-2: 基本数学运算能够用布尔运算模拟



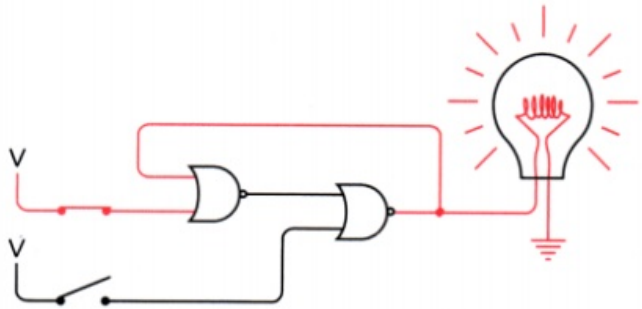
物理电路层→布尔逻辑层→计算器件（**组合逻辑**）

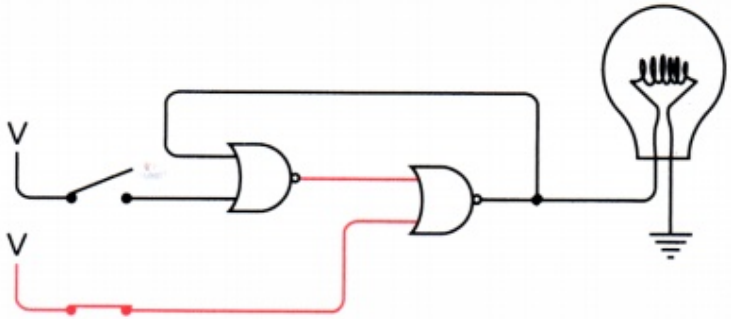
计算器件层（时序逻辑）

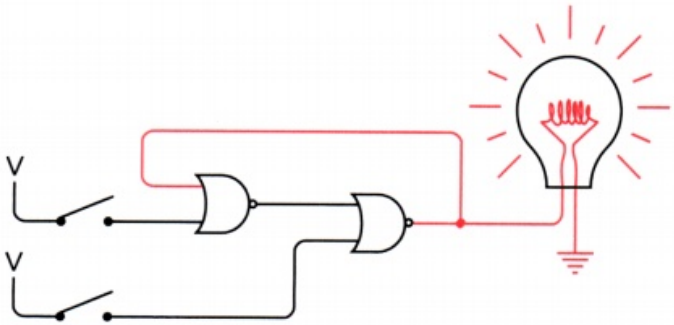


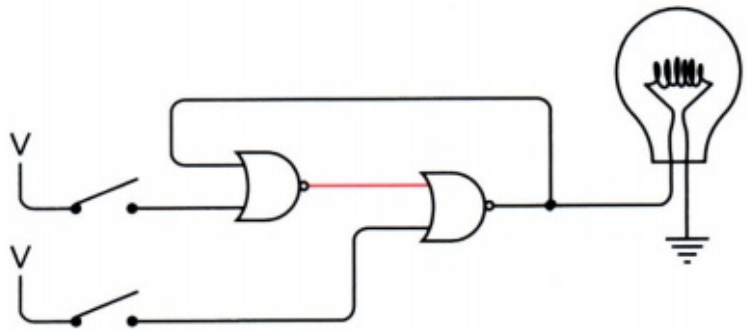


计算器件层（时序逻辑）

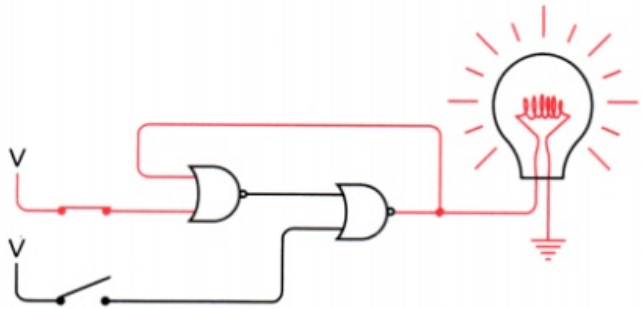


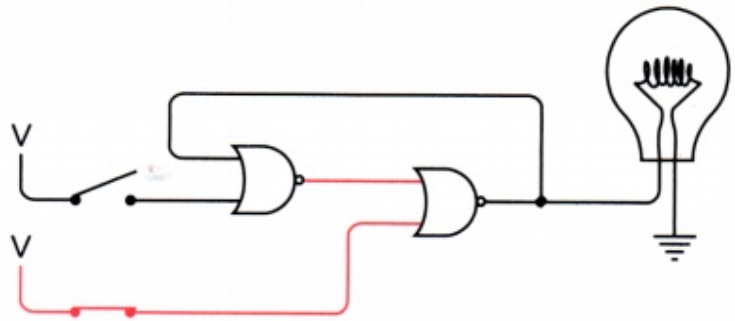


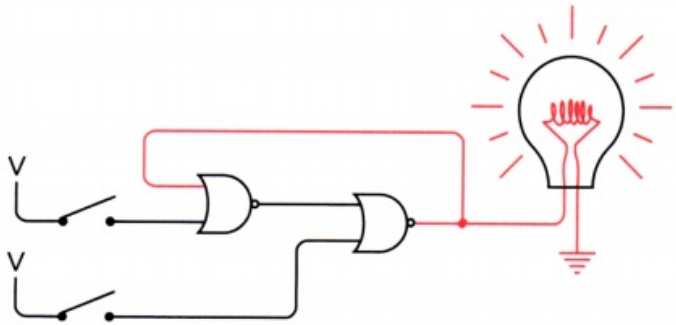


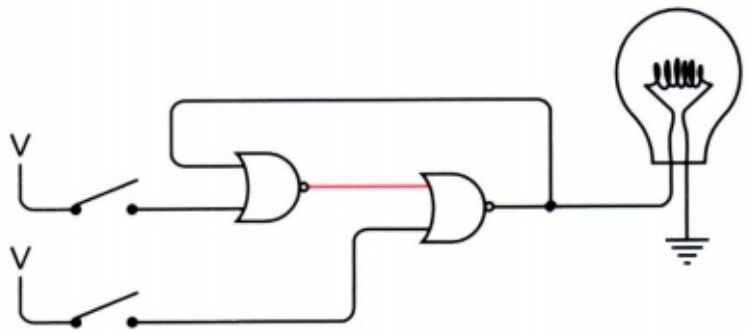


计算器件层（时序逻辑）







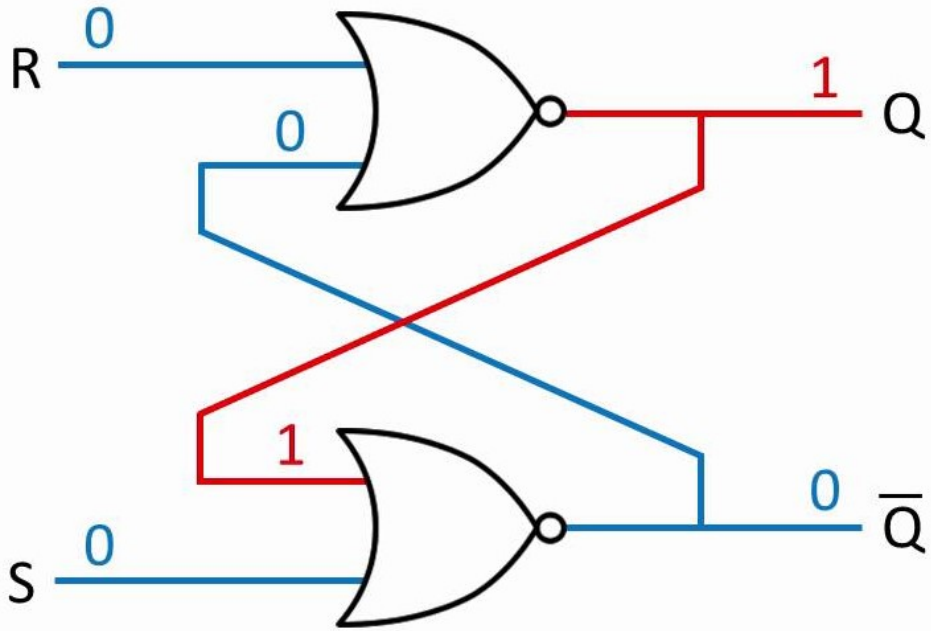


锁存器(Latch)！记忆信息！

计算器件层（时序逻辑）

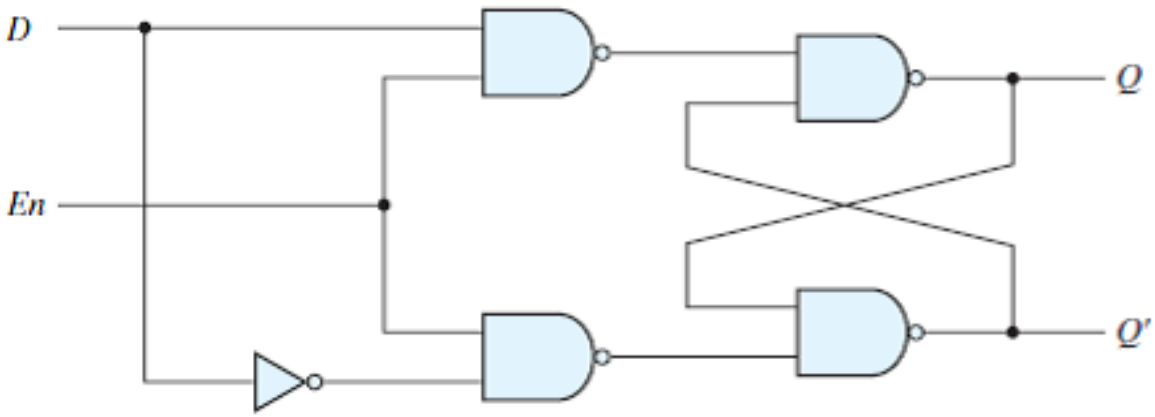
SR Latch

S	R	Q	$\overline{Q}$
0	0	1 0	0 1
0	1	0	1
1	0	1	0
1	1	0	0



计算器件层（时序逻辑）

D Latch



(a) Logic diagram

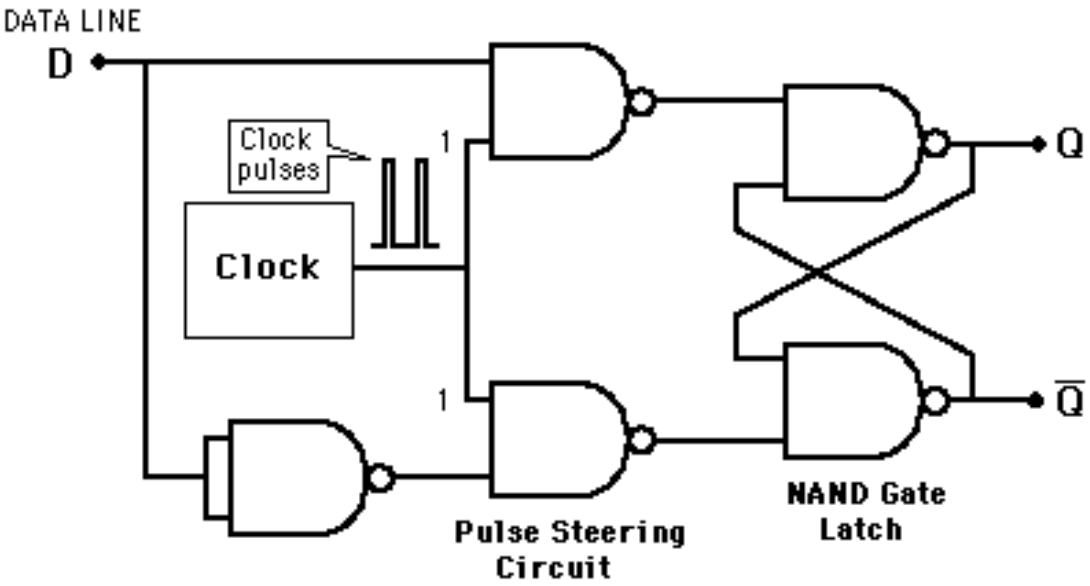
En	D	Next state of Q
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

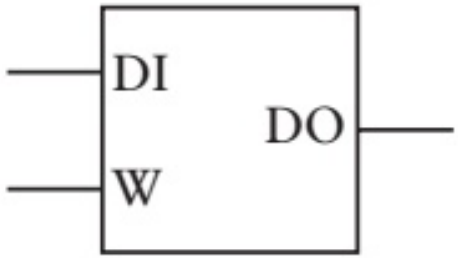
(b) Function table

FIGURE 5.6
D latch

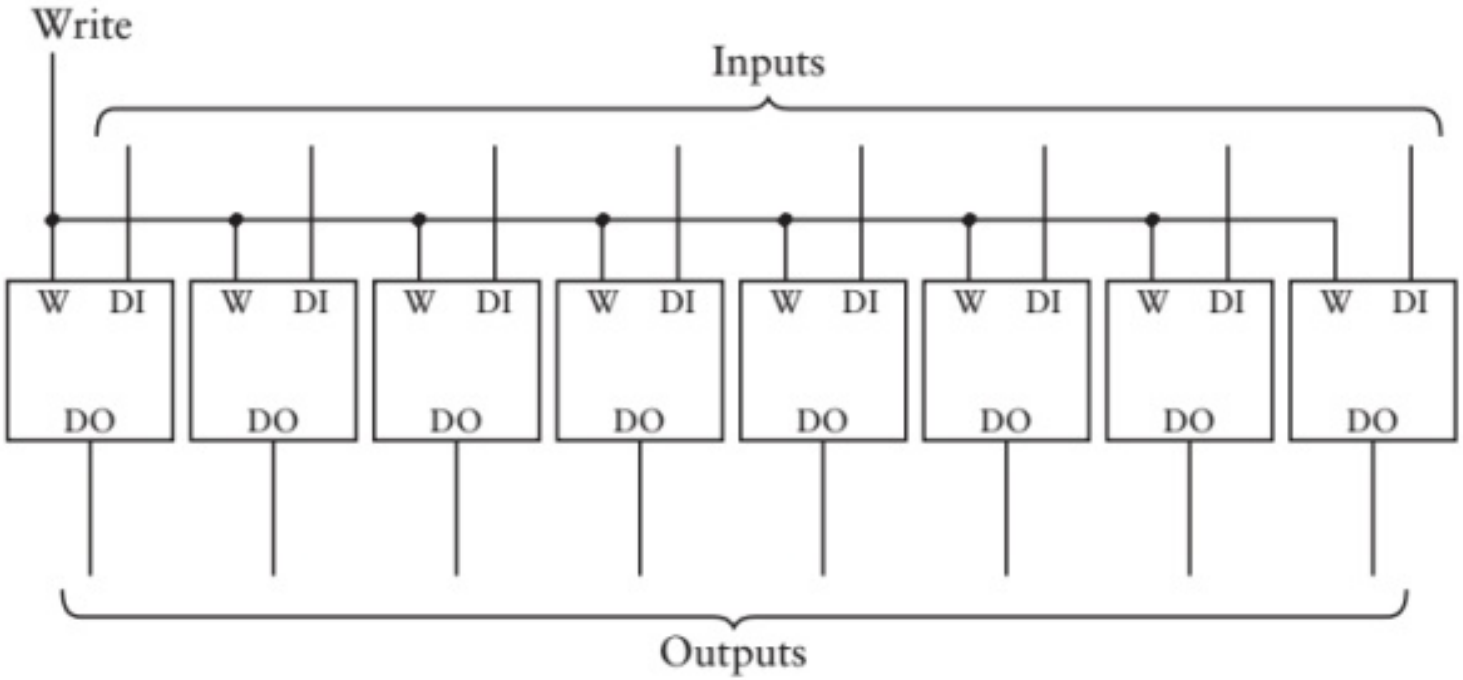
计算器件层（时序逻辑）

D Flip-flop (触发器) – 对**时钟边沿**敏感





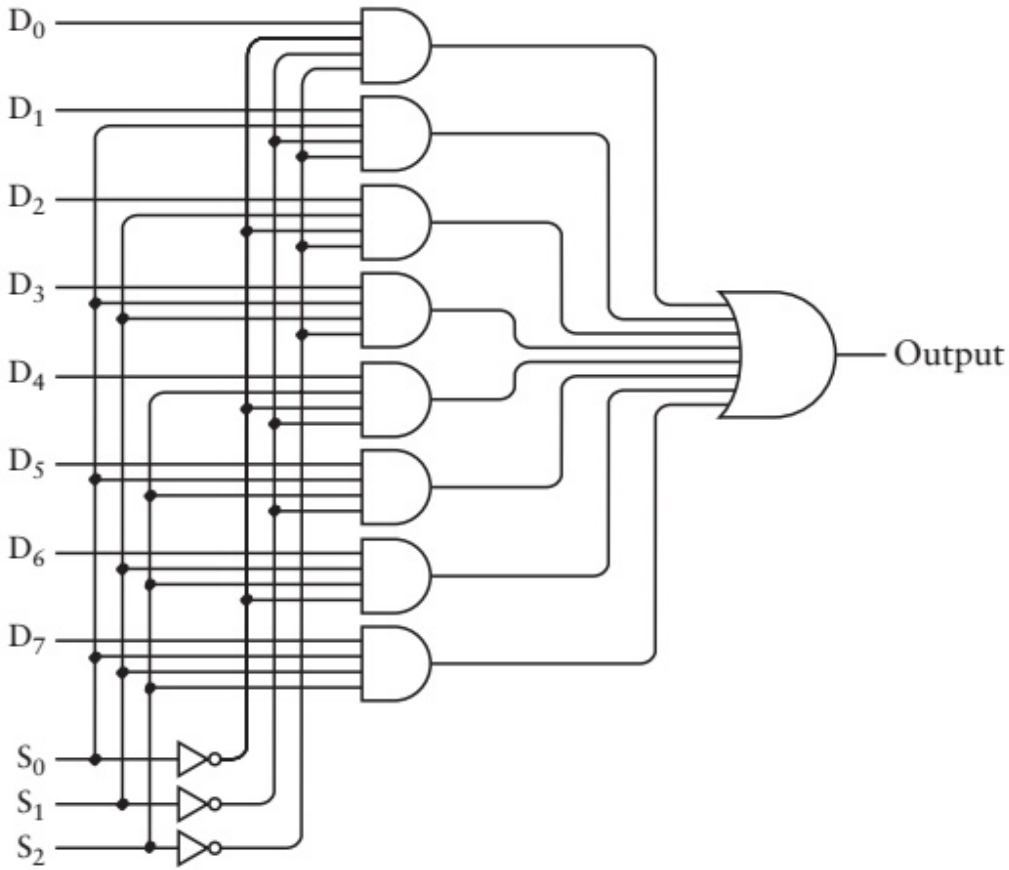
计算器件层（时序逻辑）


Write
Inputs
W DI
DO
Outputs


Data In
8
DI
8-Bit Latch
DO
8
Data Out
Write
W

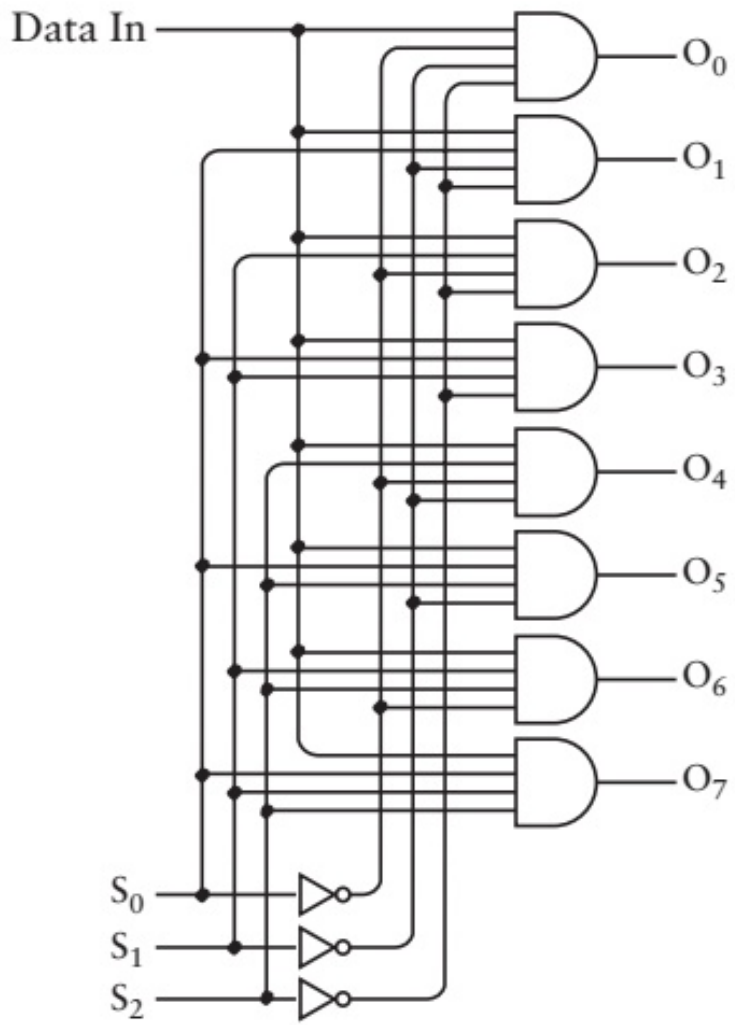
计算器件层

8-1 Multiplexer



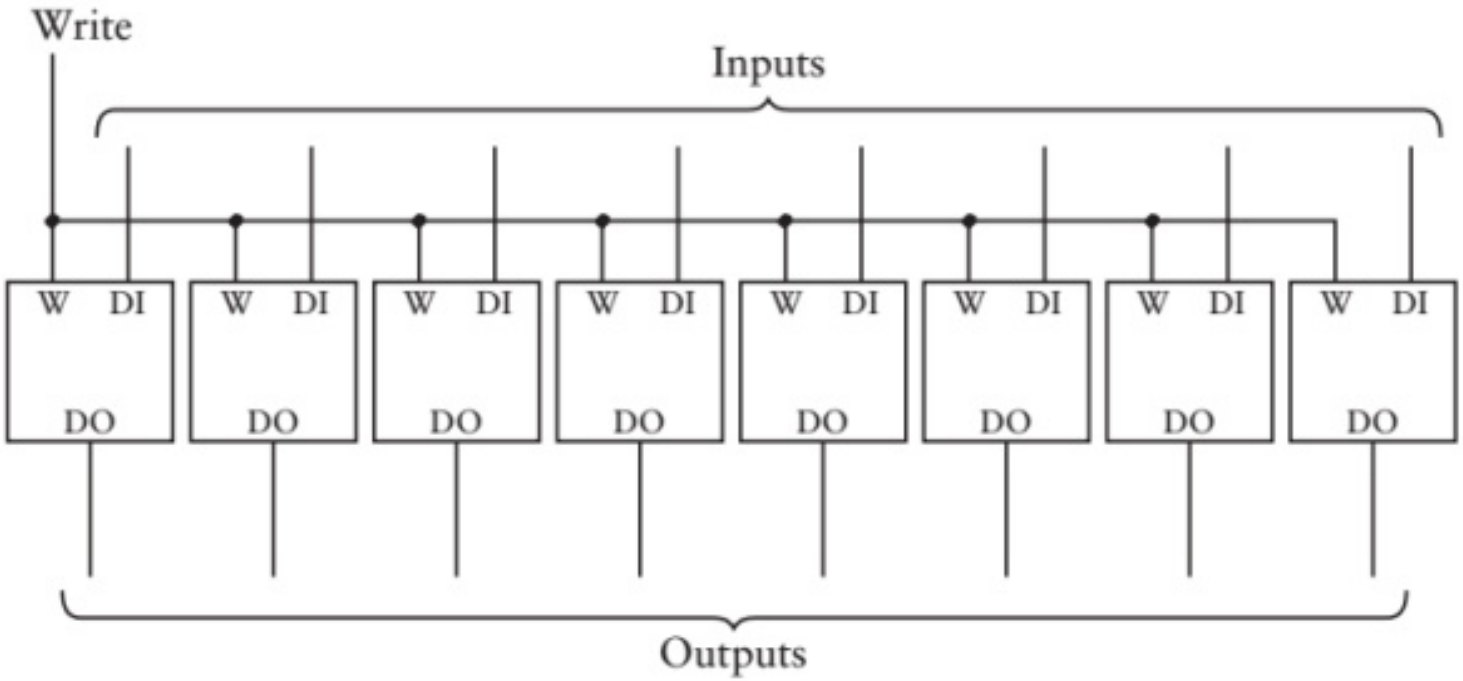
计算器件层

3-8 Decoder



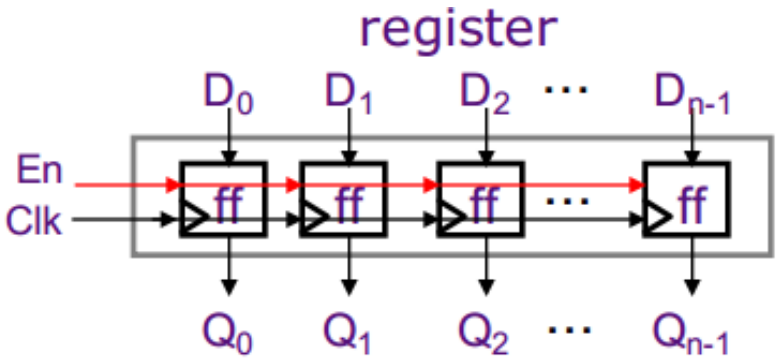
计算器件层

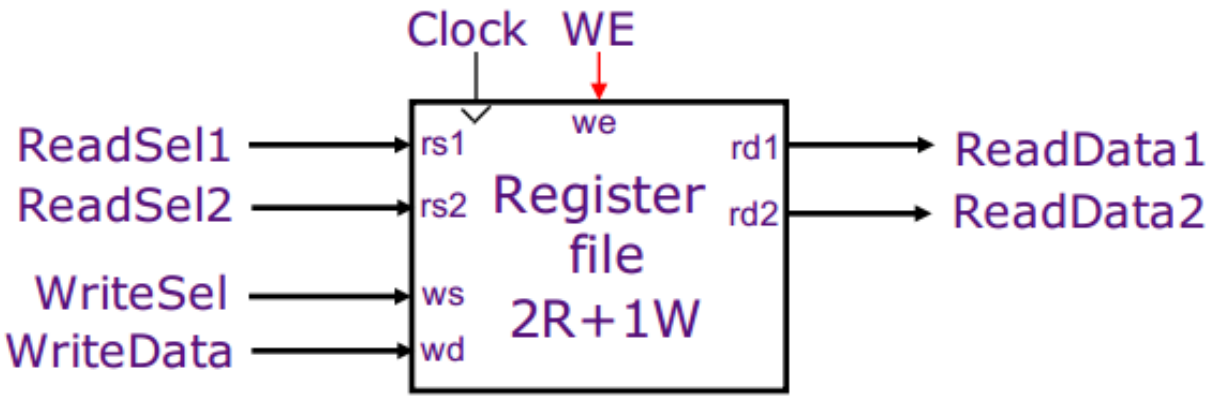
8-bit Memory (RAM)



计算器件层

寄存器堆 (Register File)





No timing issues in reading a selected register

0-2: 从理论到实践的第一步

物理电路层→布尔逻辑层→计算器件/模块

0-2: 从理论到实践的第一步

物理电路层→布尔逻辑层→计算器件/模块

2-3: 用基本器件实现通用处理器(CPU)

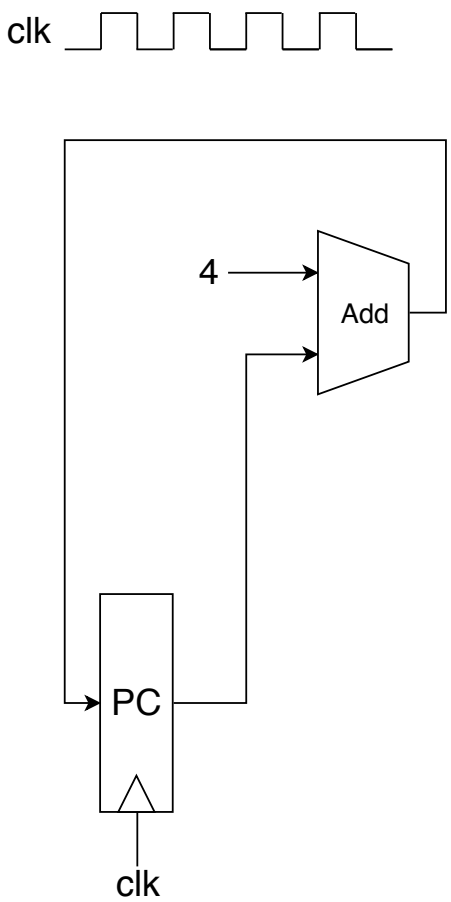
架构层

4.3

Common Datapath

共同通路(PC)

Program Counter (PC)



按字编址

0x00000000

0x00000004

0x00000008

0x0000000C

...

4.4

R-R Instructions

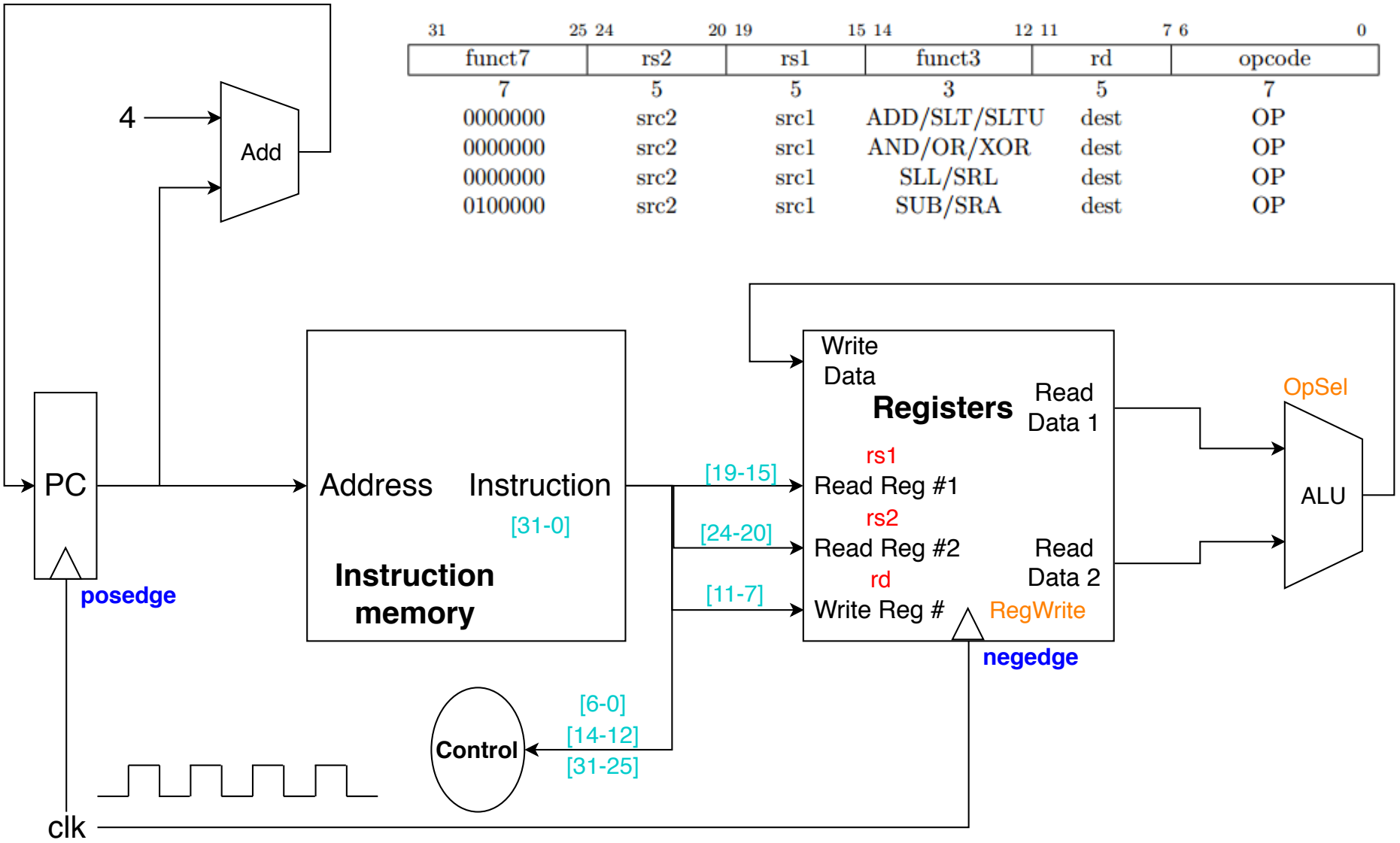
Register-Register Instructions

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

- `add rd, rs1, rs2`
- `and rd, rs1, rs2`
- `sll rd, rs1, rs2`
- `sub rd, rs1, rs2`

Register-Register Instructions

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP



Arithmetic Logic Unit (ALU)

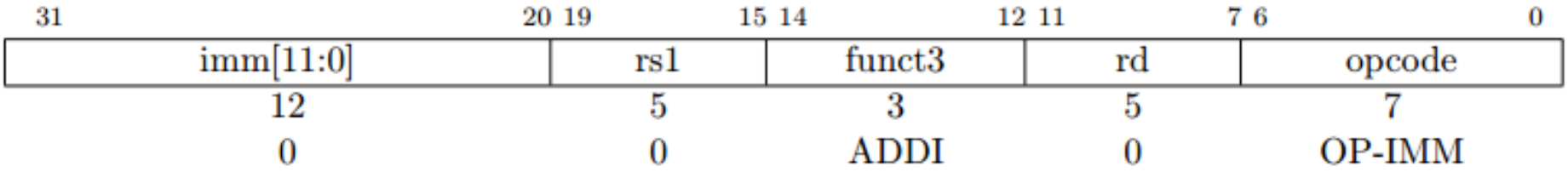
OpSel[2:0]	Function
000	$Y = A + B$
001	$Y = A - B$
010	$Y = B << A$
011	$Y = A \mid B$
100	$Y = A\&B$
101	$Y = (A < B)\ ?\ 1\ :\ 0$
110	$Y = (((A < B)\&\&(A[31] == B[31]))$ $\|\|\quad((A[31] == 1\&\&B[31] == 0)))$ $?\ 1\ :\ 0$
111	$Y = A \oplus B$

Register-Register Instructions

NO OPERATION (NOP)

`nop` $\rightarrow$ `addi x0, x0, 0`

doesn't change any user-visible state, except for advancing PC

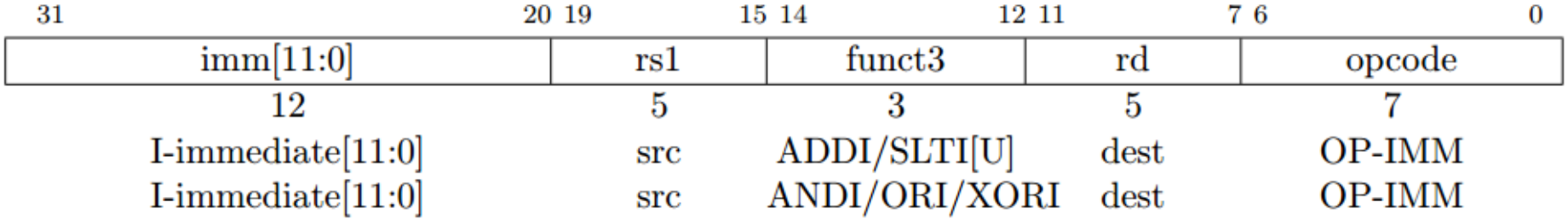


31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

4.5

R-I Instructions

Register-Immediate Instructions



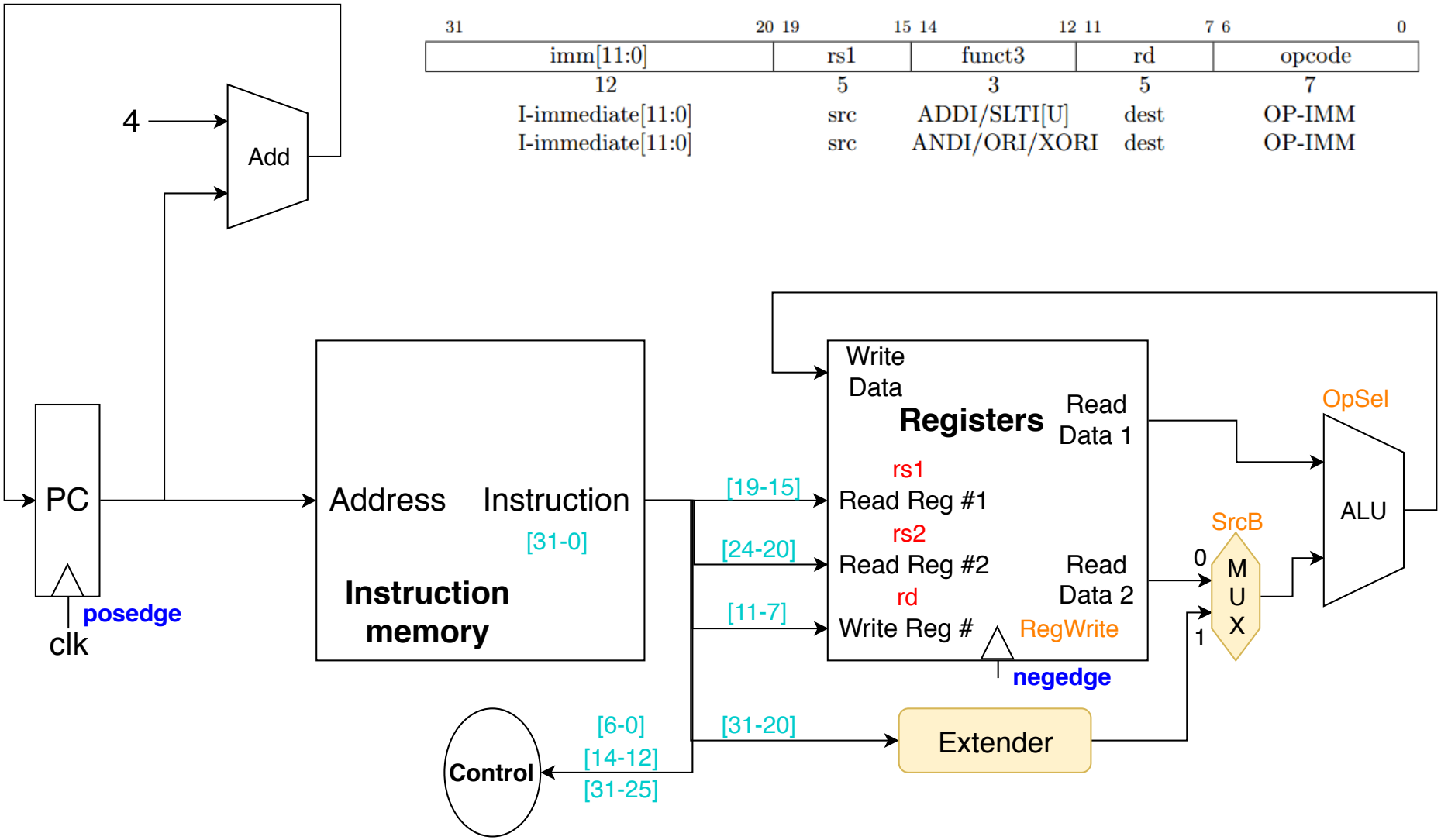
31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

All **sign-extended**

- `addi rd, rs1, imm` → imm=0, `mv rd, rs1`
- `sltiu rd, rs1, imm` → imm=1, `seqz rd, rs`
- `xori rd, rs1, imm` → imm=-1, `not rd, rs`

Register-Immediate Instructions

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

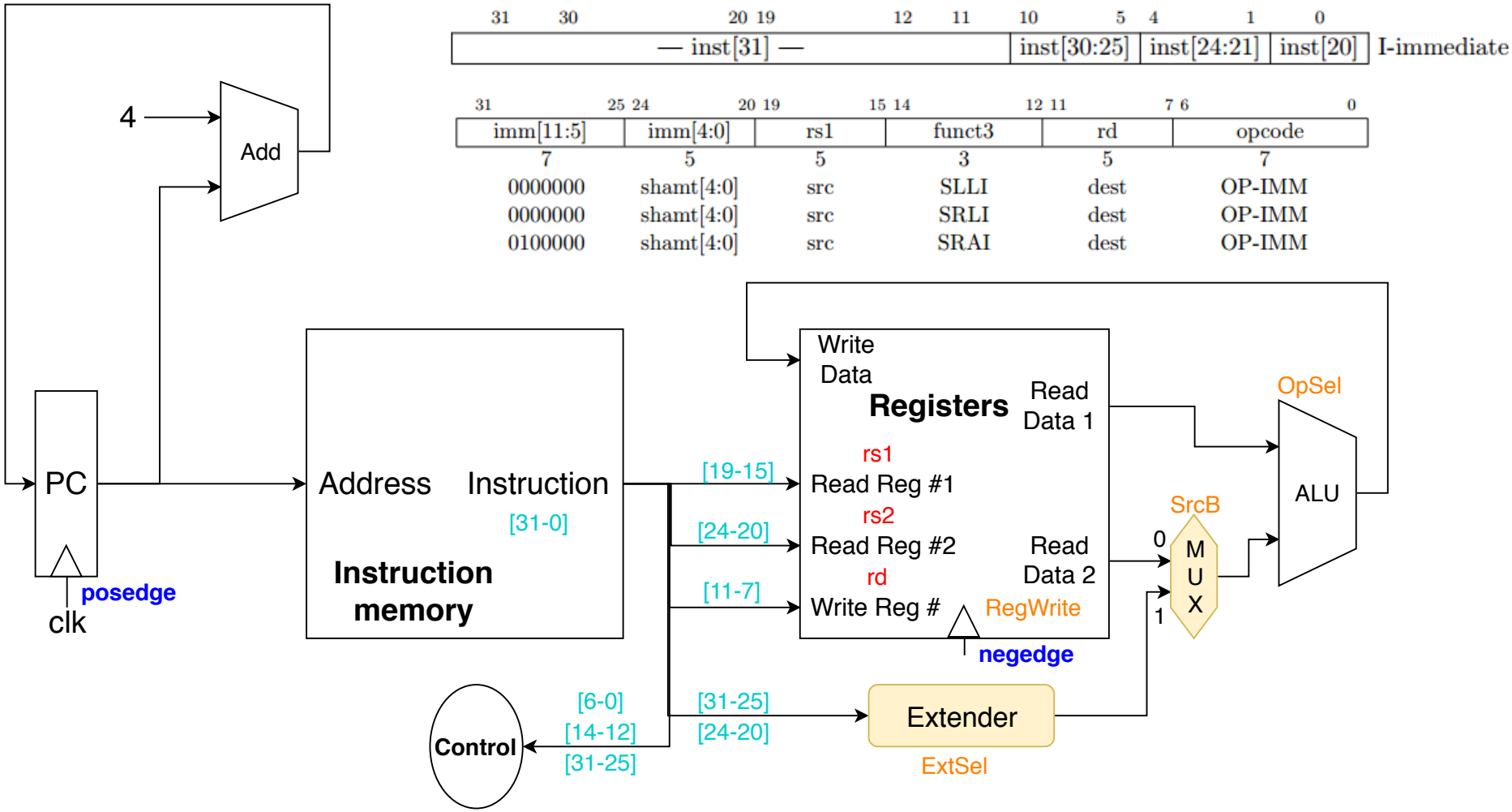


Register-Immediate Instructions

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM

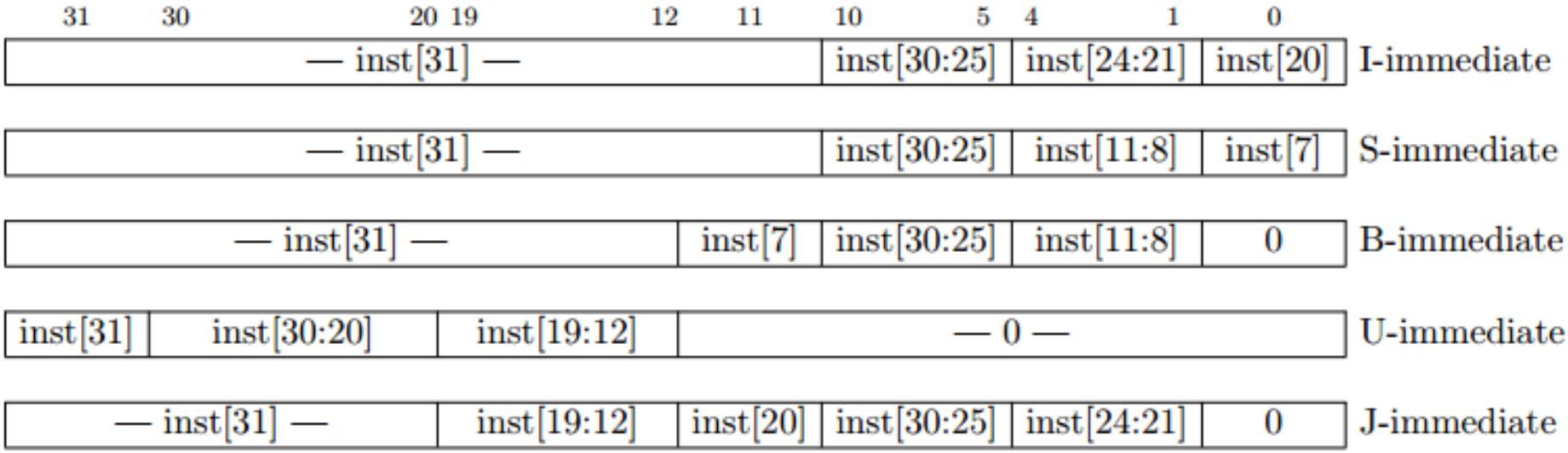
- Shift left (logic): `slli rd, rs1, imm`
- Shift right (logic): `srli rd, rs1, imm`
- Shift right (arithmetic): `srai rd, rs1, imm`
- Need not `slai`

Register-Immediate Instructions



Problem unfixed: How to extend for `srai`? Or set a threshold in ALU?

Immediates format



shift uses I-immediate

4.6

Load and Store

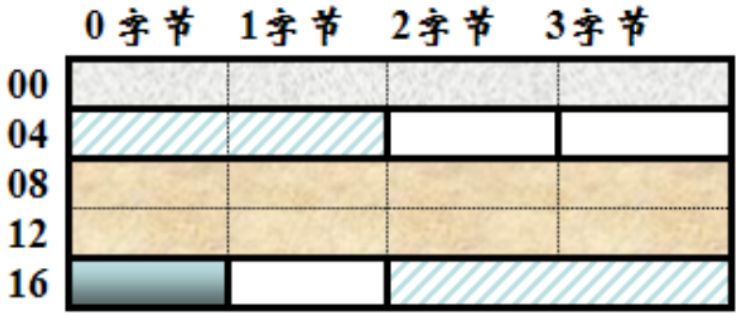
两种存储方式

- 小端(Little-endian)存储: **RISC-V**、x86、iOS、Android
- 大端(Big-endian)存储: MIPS、JPEG、Photoshop

0x12345678

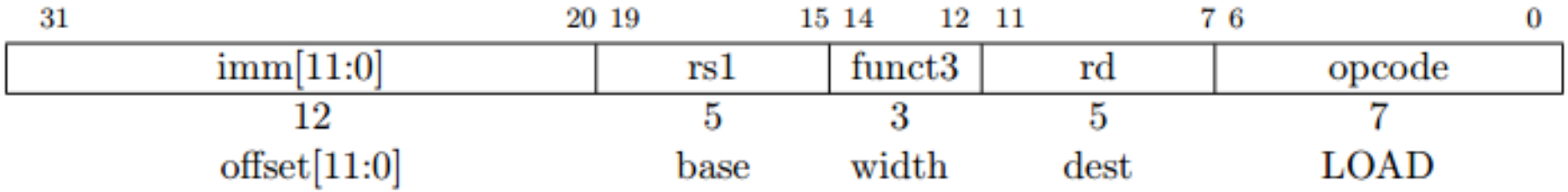
Address	00	04	08	0C
Little	78	56	34	12
Big	12	34	56	78

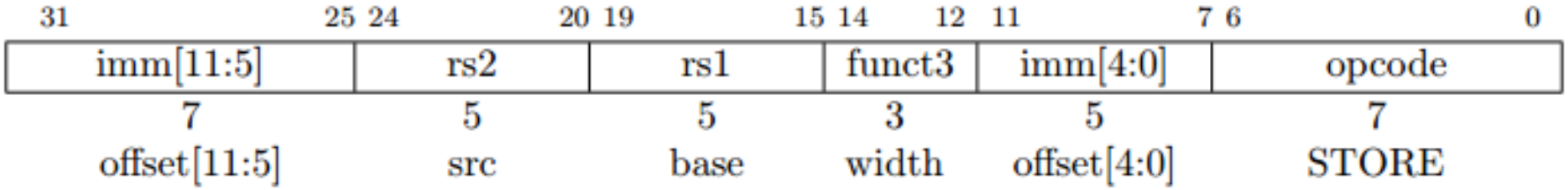
无边界对齐(misaligned)！提供细粒度访问（半字节）



Load & Store

RISC-V is **Load-Store Architecture**, only `lw` and `sw` can access memory
Other arithmetic instructions can only operate on CPU registers

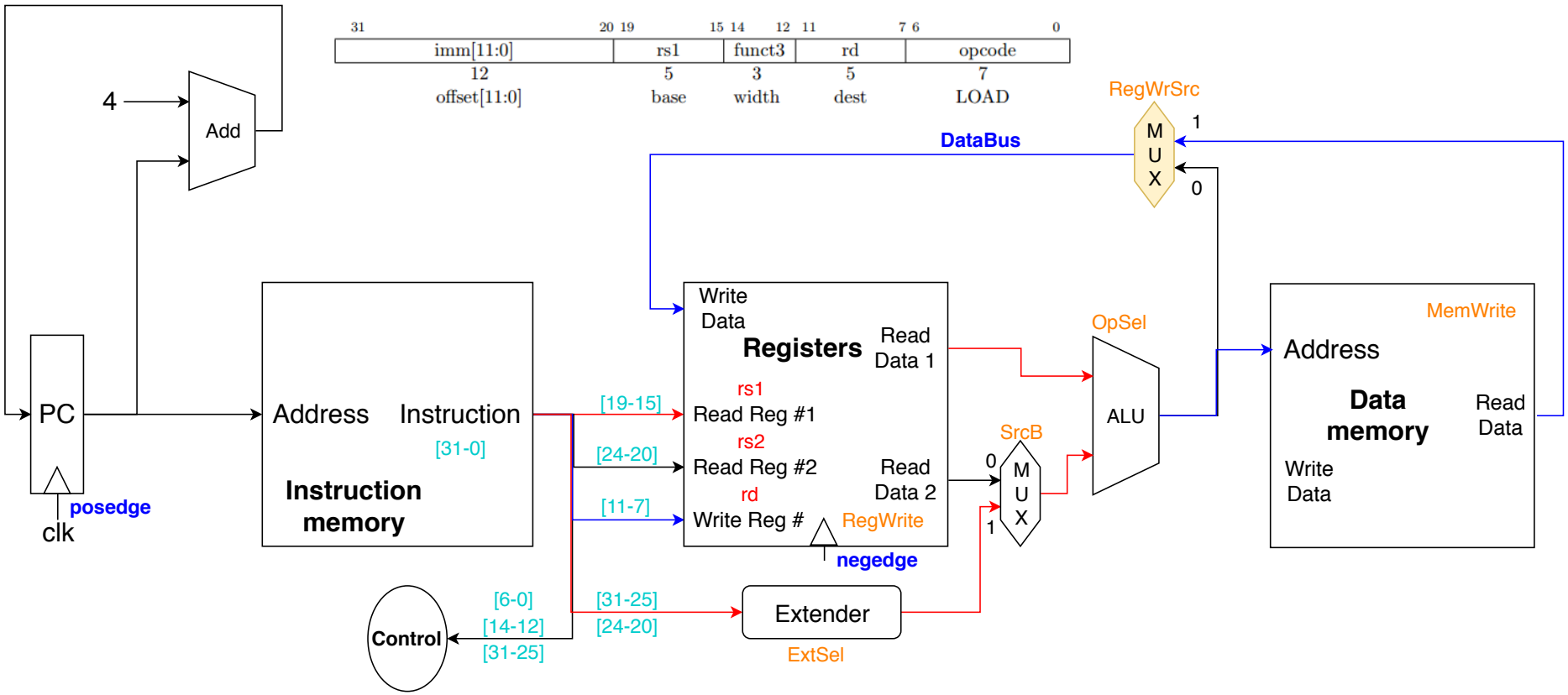




- `lw rd, base(imm)`
- `sw rs, base(imm)`

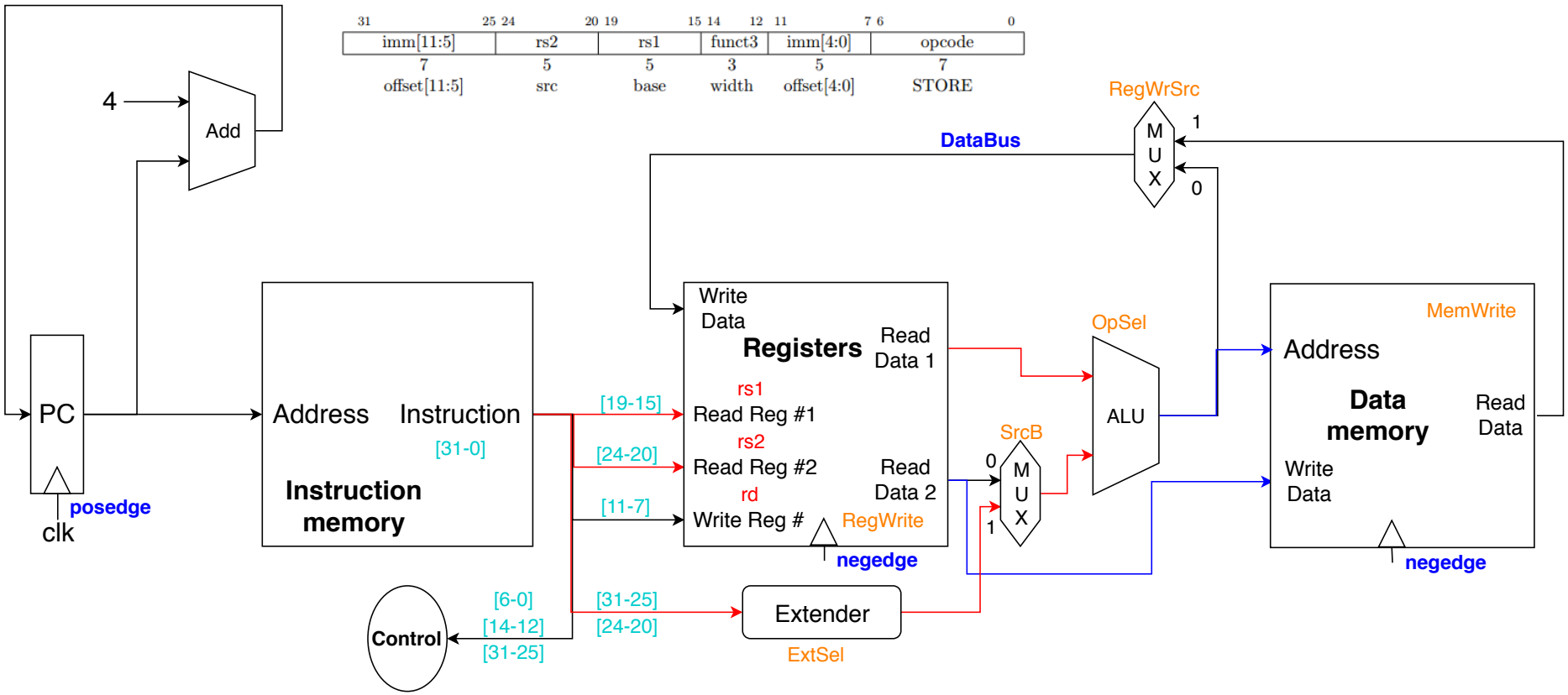
Load

```
x[rd] = sgnext(M[x[rs1] + sgnext(offset)][31:0])
```



Store

`M[x[rs1] + sgnext(offset)] = x[rs2][31:0]`

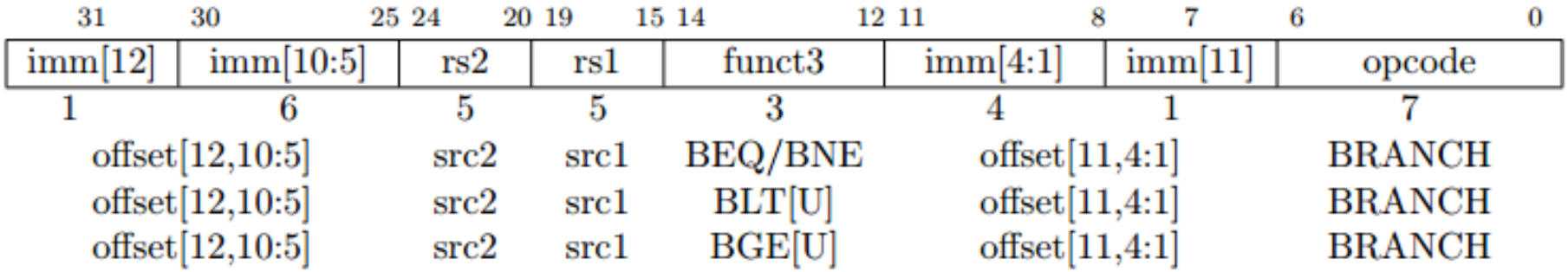


4.7

Branch

Branch

- Used in loop
- Important for pipelining (Branch prediction)
- address range ± 4KiB

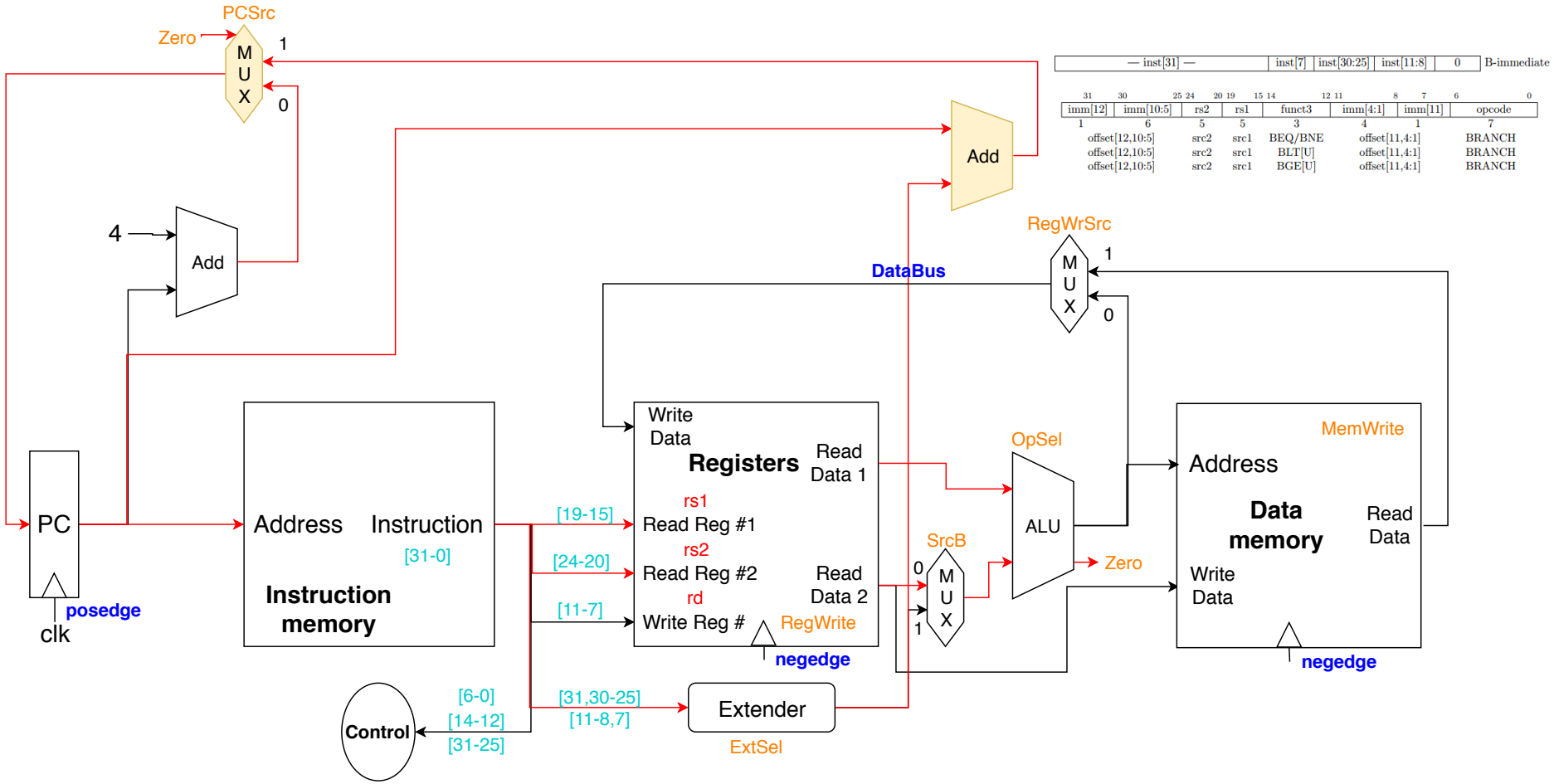


31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]		BRANCH

- `beq rs1, rs2, imm`

Branch

`if (rs1 == rs2) pc += sgnext(offset)`偏移量均为2的倍数，但地址必须4的倍数，否则抛异常



Use ExtSel (maybe 3 bits) to control which type of imm

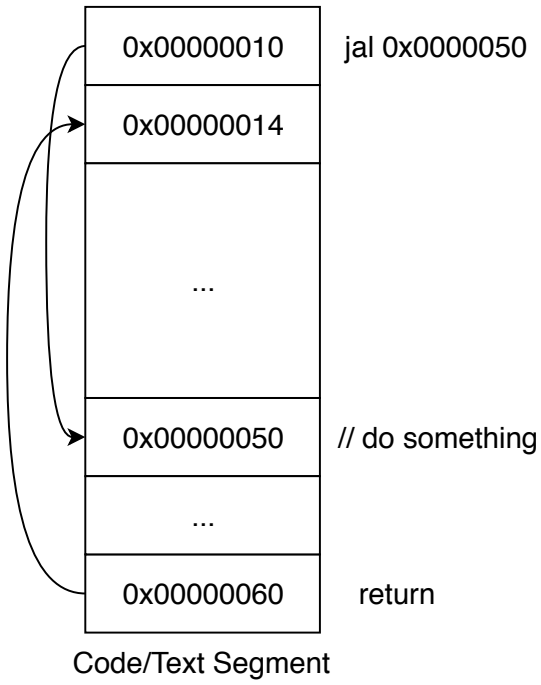
4.8

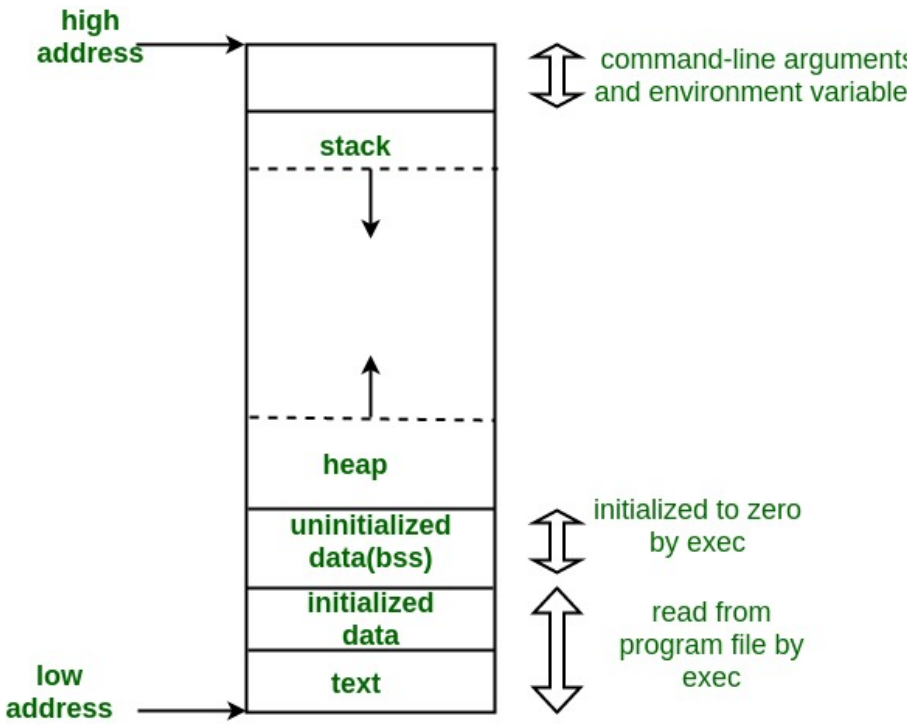
Jump

Function Call

堆栈、保护现场、恢复现场

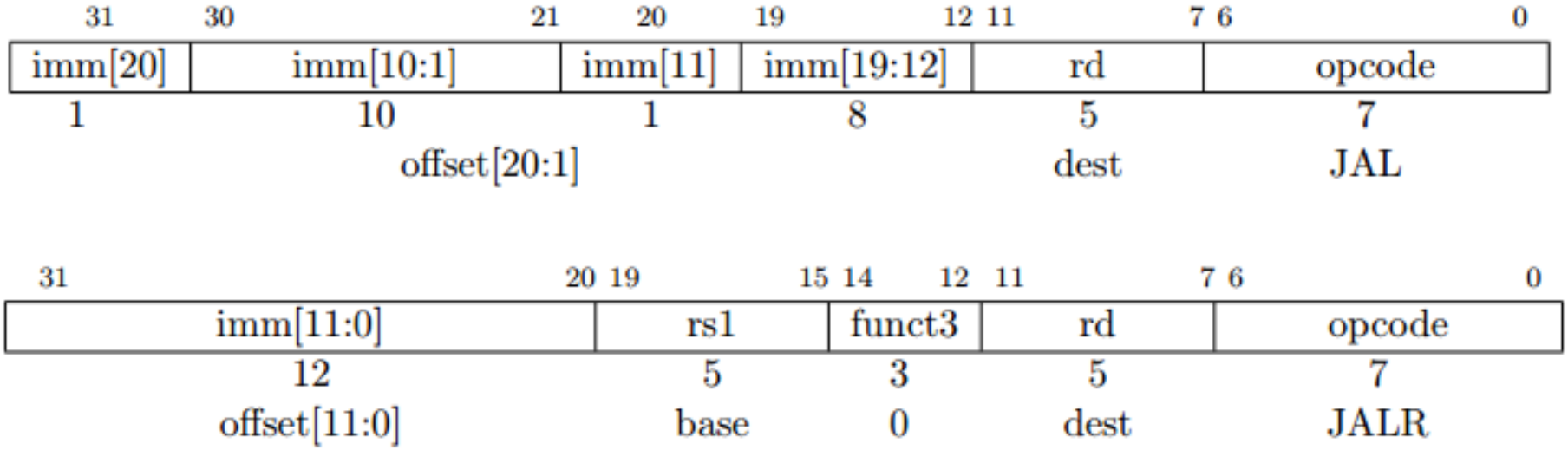
```
void foo (void)
{
    // do something
}

int main (void)
{
    foo();
    return 0;
}
```





Jump

Address range ±1MiB

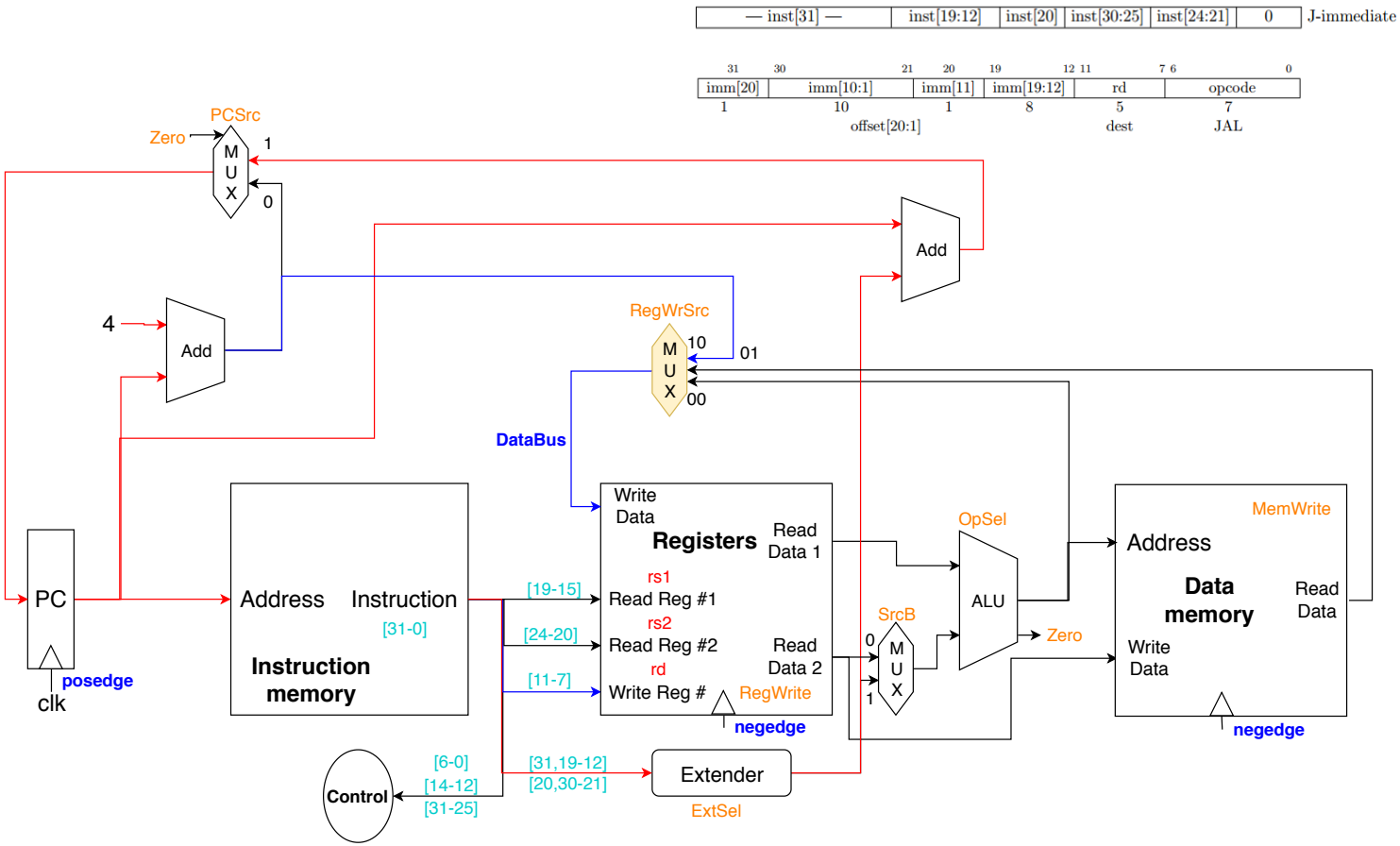


- `jal rd, offset`
- `jalr rd, offset(rs1)`

Use `lui` and `auipc` to jump anywhere in a 32-bit pc-relative address range

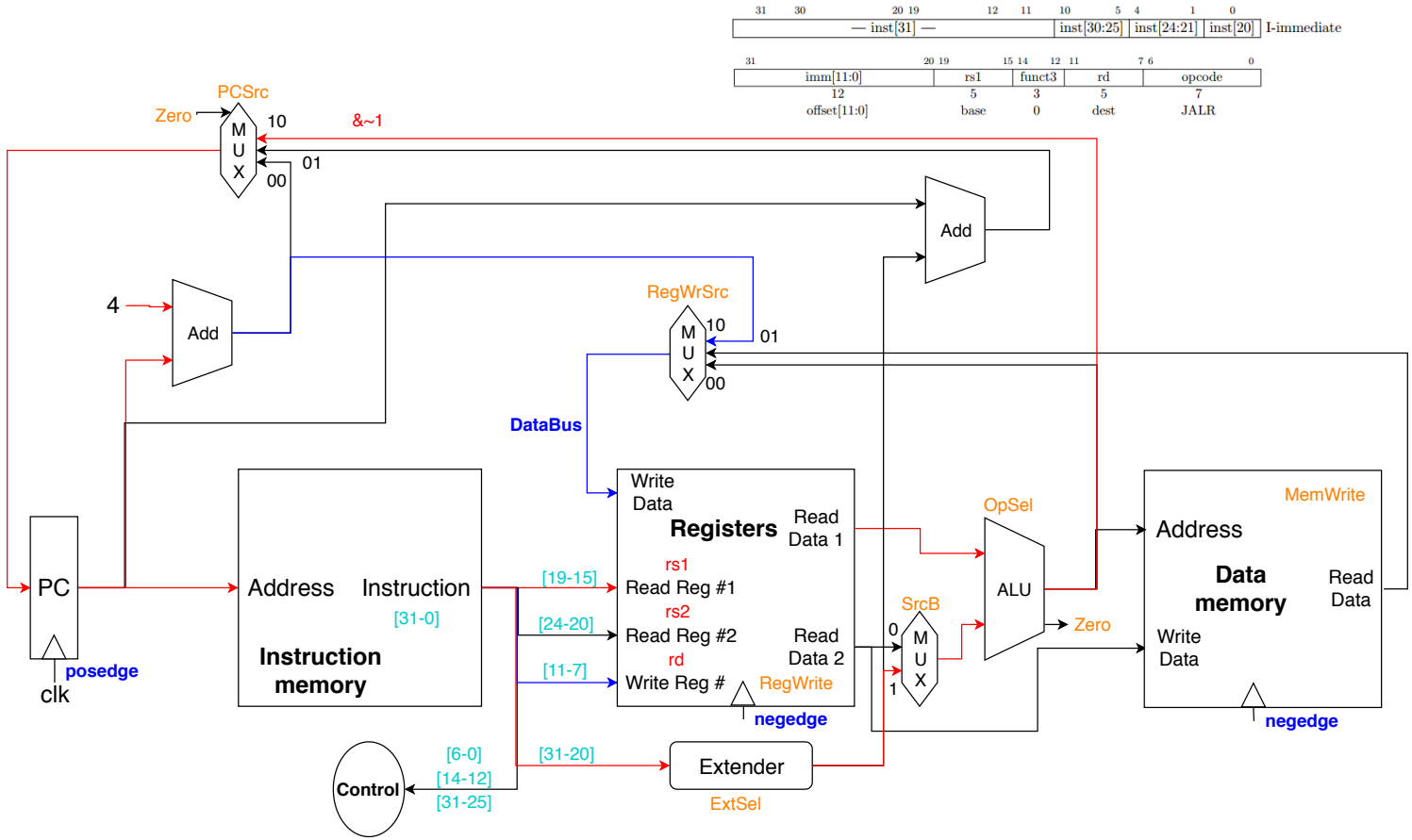
Jal

```
x[rd] = pc+4; pc += sgnext(offset)
```



Jalr

`x[rd]=pc+4; pc=(x[rs1]+sext(offset))&~1;` 偏移量同样是2的倍数，地址非4的倍数抛异常



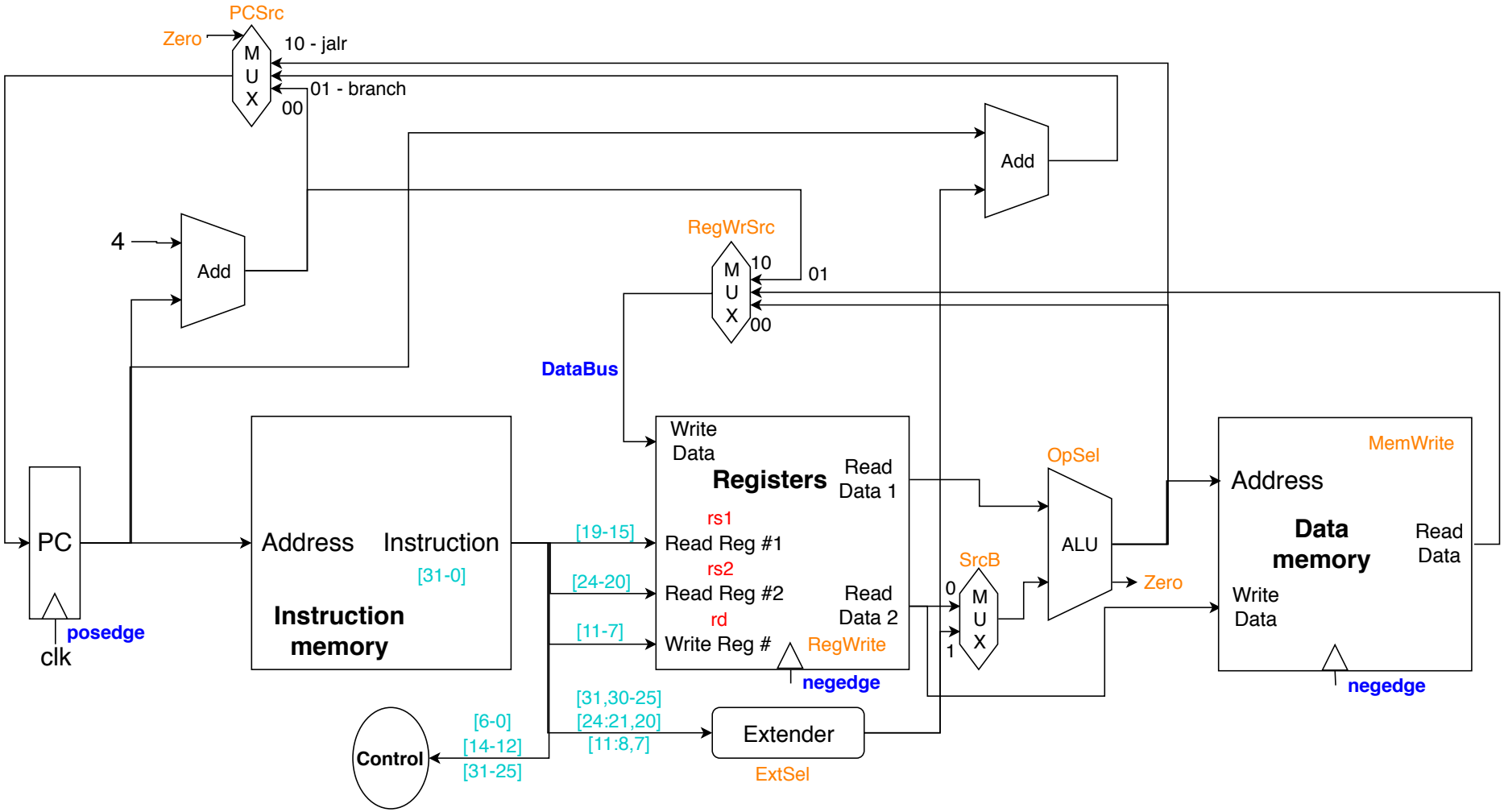
Question again: Where to implement `&~1`?

4.9

Single Cycle CPU Summary

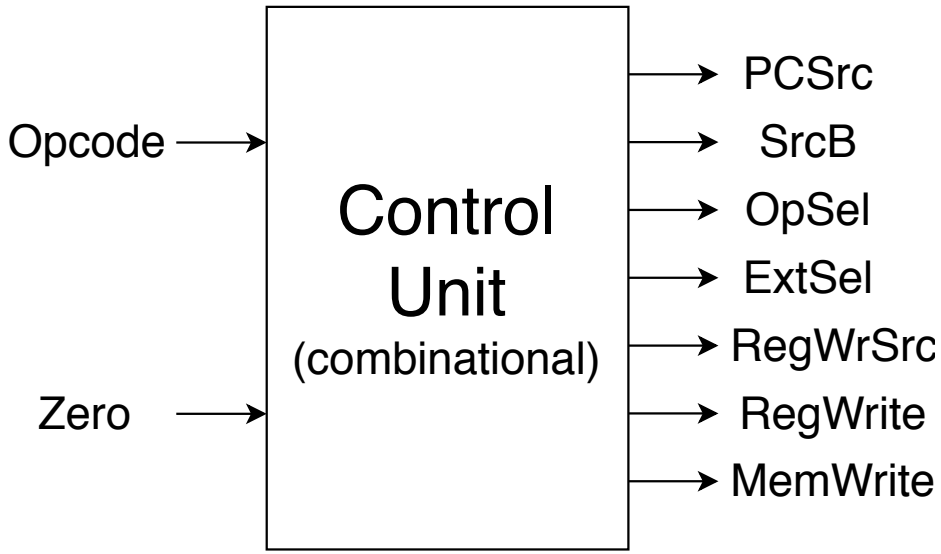
Harvard Architecture (Aiken and Mark)

Separate program and data memory



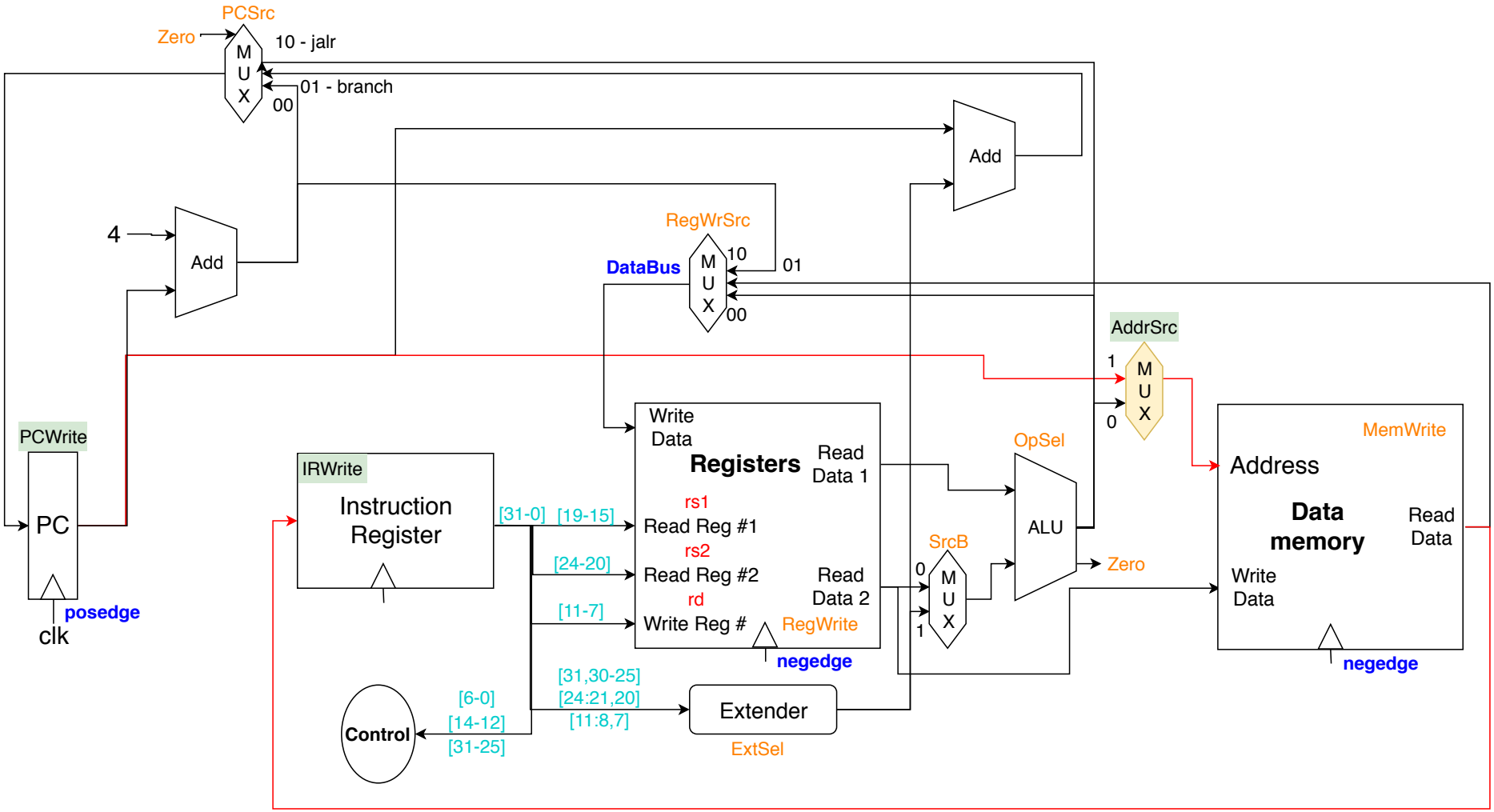
Control Signals

Hardwired control is pure combinational logic



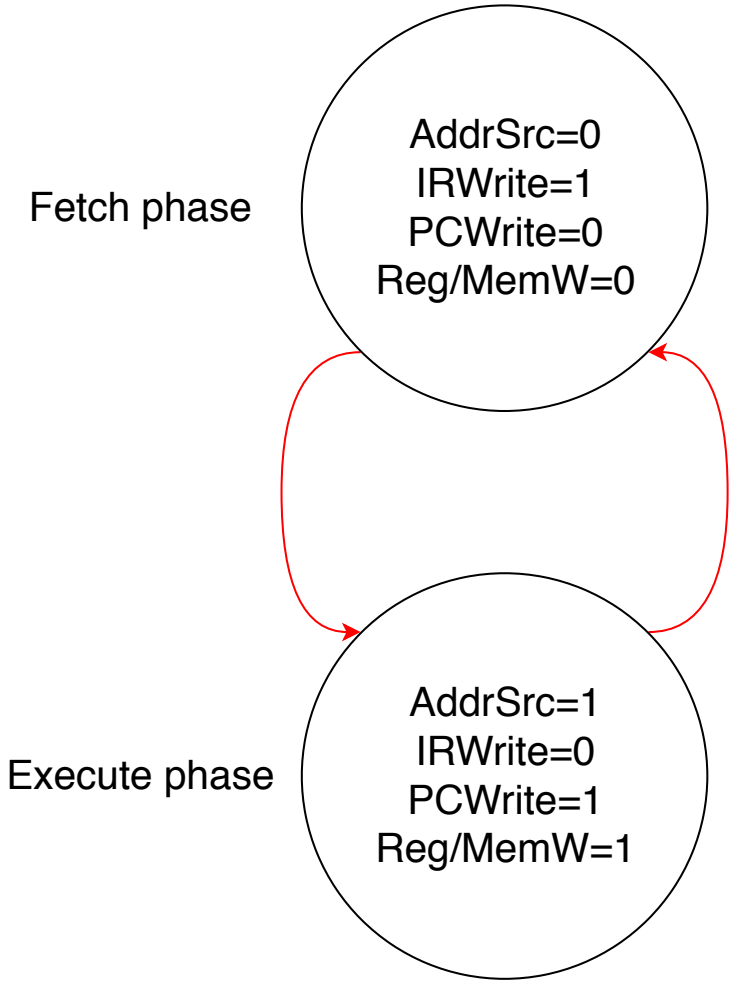
Princeton Architecture (von Neumann)

The same program and data memory



State Transition

Two-state controller (A filp-flop can be used to remember)



So, Princeton and Harvard, which one is better?

Clock Rate vs CPI

We will assume

- clock period is sufficiently long for all of the following steps to be "completed":

1. instruction fetch
2. decode and register fetch
3. ALU operation
4. data fetch if required
5. register write-back setup time

$$\Rightarrow t_C > t_{IFetch} + t_{RFetch} + t_{ALU} + t_{DMem} + t_{RWB}$$

Clock Rate vs CPI

$$t_{C\text{-}Princeton} > \max\{t_M, t_{RF} + t_{ALU} + t_M + t_{WB}\}$$
$$t_{C\text{-}Princeton} > t_{RF} + t_{ALU} + t_M + t_{WB}$$
$$t_{C\text{-}Harvard} > t_M + t_{RF} + t_{ALU} + t_M + t_{WB}$$

Suppose $t_M >> t_{RF} + t_{ALU} + t_{WB}$

$$t_{C\text{-}Princeton} = 0.5 * t_{C\text{-}Harvard}$$

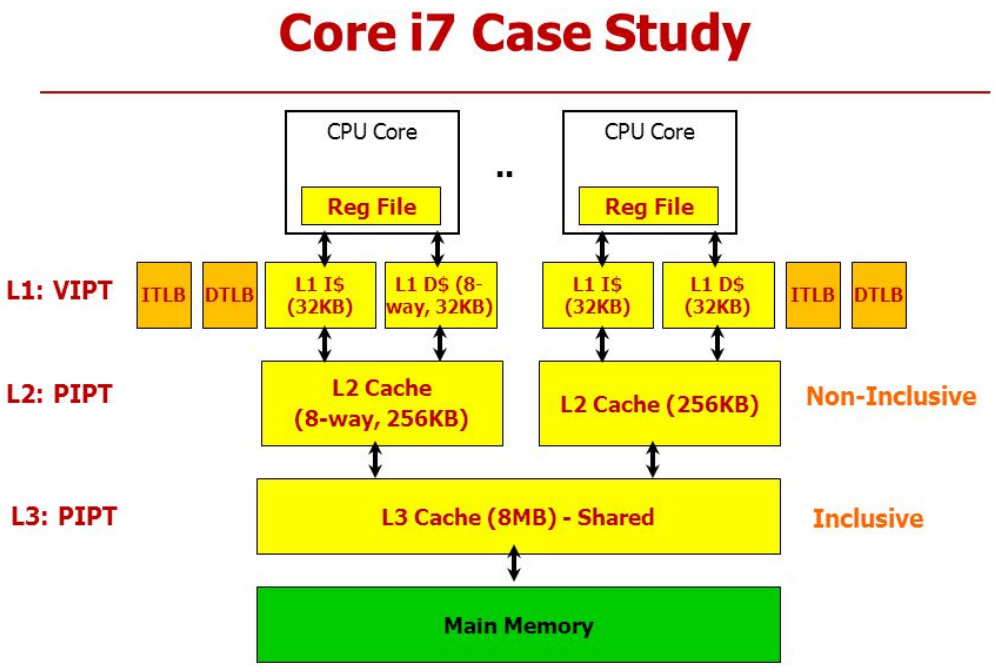
$$CPI_{Princeton} = 2$$
$$CPI_{Harvard} = 1$$

No difference in performance!

Is it possible to design a controller for the Princeton architecture with CPI < 2 ?

Two Architectures Summary

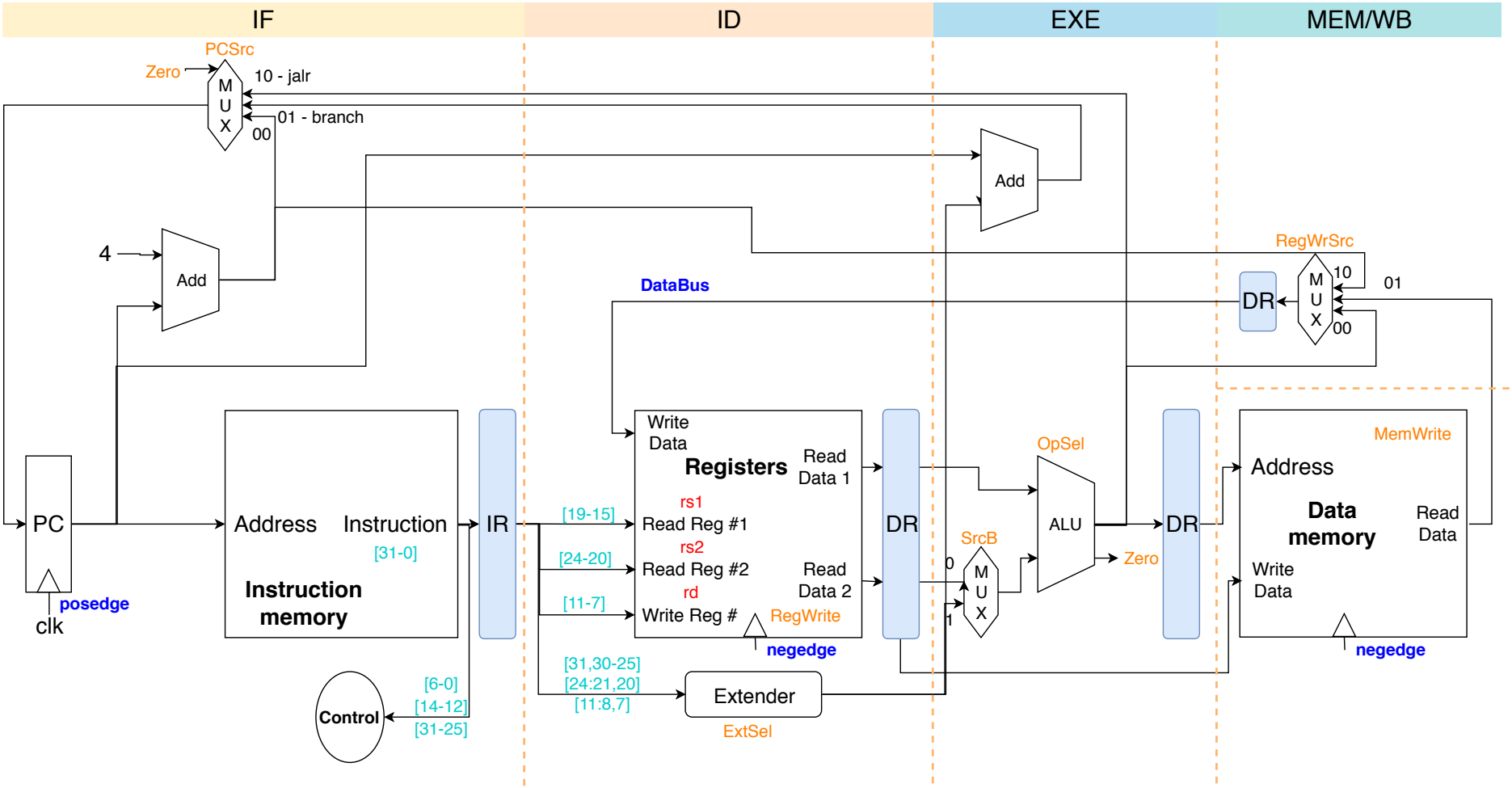
- Princeton: The same program and data memory, simple, two cycles, first place
- Harvard: Different program and data memory, complex, one cycle, ignored until 1970s



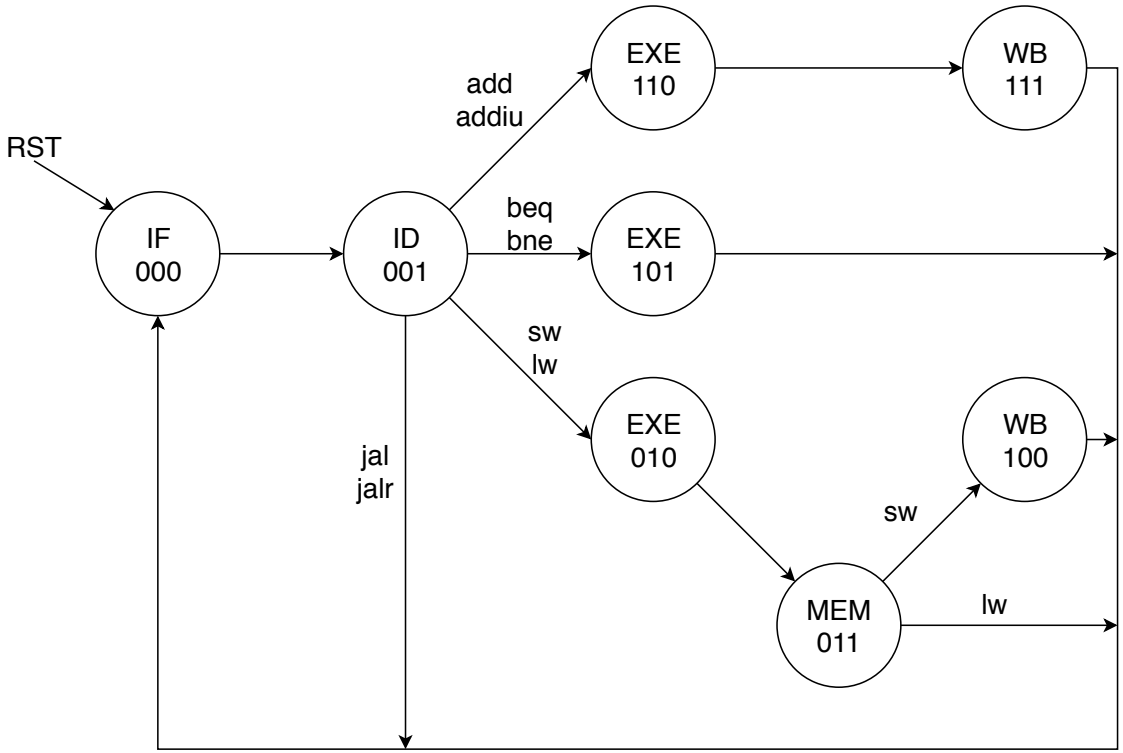
5

多周期CPU概述

Multi-Cycle CPU Datapath



Multi-Cycle CPU State Transition



6

RISC-V的内存模型

内存模型

内存模型：系统与程序员之间的规范，限定了**共享内存**的多线程/多核处理器的**读写顺序**

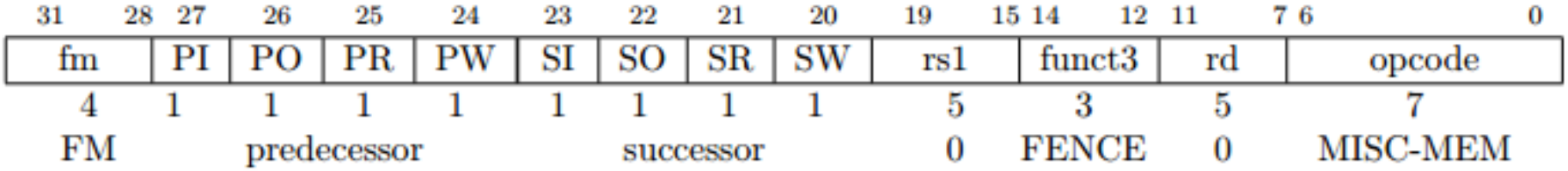
程序员（实施） → 高级语言（可移植规范） —编译器（实施）→ 汇编语言/指令集（不可移植规范） → 硬件（实施）

内存模型的强弱：

- 强：程序员编程简单，但限制了编译器指令重排/硬件流水
- 弱：硬件实施更简便、灵活、高效，但编程麻烦

内存模型

RISC-V Weak Memory Ordering (RVWMO)



FENCE seperates pred and succ (A sync)

Predecessor (P), successor (S)
Device input (I), output (O), read (R), write (W)

Hardware implementation? Emmm...

7

RISC-V扩展

RV32I的扩展

- RV32M: 乘除法
- RV32F/RV32D: 单双精度浮点数
- RV32A: 原子指令
- RV32C: 压缩指令(16位)
- RV32V: 向量架构（避免SIMD）
- RV64: 64位

以及一些未来可选扩展

Opcode Map

inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	$\geq$ 80b

Table 8.1: RISC-V base opcode map, inst[1:0]=11

31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	SB-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	UJ-type

RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI rd,imm
imm[31:12]				rd	0010111	AUIPC rd,imm
imm[20|10:1|11|19:12]				rd	1101111	JAL rd,imm
imm[11:0]			rs1 000	rd	1100111	JALR rd,rs1,imm
imm[12|10:5]		rs2	rs1 000	imm[4:1|11]	1100011	BEQ rs1,rs2,imm
imm[12|10:5]		rs2	rs1 001	imm[4:1|11]	1100011	BNE rs1,rs2,imm
imm[12|10:5]		rs2	rs1 100	imm[4:1|11]	1100011	BLT rs1,rs2,imm
imm[12|10:5]		rs2	rs1 101	imm[4:1|11]	1100011	BGE rs1,rs2,imm
imm[12|10:5]		rs2	rs1 110	imm[4:1|11]	1100011	BLTU rs1,rs2,imm
imm[12|10:5]		rs2	rs1 111	imm[4:1|11]	1100011	BGEU rs1,rs2,imm
imm[11:0]			rs1 000	rd	0000011	LB rd,rs1,imm
imm[11:0]			rs1 001	rd	0000011	LH rd,rs1,imm
imm[11:0]			rs1 010	rd	0000011	LW rd,rs1,imm
imm[11:0]			rs1 100	rd	0000011	LBU rd,rs1,imm
imm[11:0]			rs1 101	rd	0000011	LHU rd,rs1,imm
imm[11:5]		rs2	rs1 000	imm[4:0]	0100011	SB rs1,rs2,imm
imm[11:5]		rs2	rs1 001	imm[4:0]	0100011	SH rs1,rs2,imm
imm[11:5]		rs2	rs1 010	imm[4:0]	0100011	SW rs1,rs2,imm
imm[11:0]			rs1 000	rd	0010011	ADDI rd,rs1,imm
imm[11:0]			rs1 010	rd	0010011	SLTI rd,rs1,imm
imm[11:0]			rs1 011	rd	0010011	SLTIU rd,rs1,imm
imm[11:0]			rs1 100	rd	0010011	XORI rd,rs1,imm
imm[11:0]			rs1 110	rd	0010011	ORI rd,rs1,imm
imm[11:0]			rs1 111	rd	0010011	ANDI rd,rs1,imm
0000000		shamt	rs1 001	rd	0010011	SLLI rd,rs1,shamt
0000000		shamt	rs1 101	rd	0010011	SRLI rd,rs1,shamt
0100000		shamt	rs1 101	rd	0010011	SRAI rd,rs1,shamt
0000000		rs2	rs1 000	rd	0110011	ADD rd,rs1,rs2
0100000		rs2	rs1 000	rd	0110011	SUB rd,rs1,rs2
0000000		rs2	rs1 001	rd	0110011	SLL rd,rs1,rs2
0000000		rs2	rs1 010	rd	0110011	SLT rd,rs1,rs2
0000000		rs2	rs1 011	rd	0110011	SLTU rd,rs1,rs2
0000000		rs2	rs1 100	rd	0110011	XOR rd,rs1,rs2
0000000		rs2	rs1 101	rd	0110011	SRL rd,rs1,rs2
0100000		rs2	rs1 101	rd	0110011	SRA rd,rs1,rs2
0000000		rs2	rs1 110	rd	0110011	OR rd,rs1,rs2
0000000		rs2	rs1 111	rd	0110011	AND rd,rs1,rs2
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	SCALL
000000000001			00000	000	00000	1110011	SBREAK
110000000000			00000	010	rd	1110011	RDCYCLE rd
110010000000			00000	010	rd	1110011	RDCYCLEH rd
110000000001			00000	010	rd	1110011	RDTIME rd
110010000001			00000	010	rd	1110011	RDTIMEH rd
110000000010			00000	010	rd	1110011	RDINSTRET rd
110010000010			00000	010	rd	1110011	RDINSTRETH rd