Computer System Architecture

— Computation, ISA and Datapath

陈鸿峥

April, 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

- 哥德尔(Gödel)不完备性定理, 1931

  任何包含了算术的数学系统都不可能同时拥有完备性和一致性!

一切的开端

- 哥德尔(Gödel)不完备性定理, 1931
  任何包含了算术的数学系统都不可能同时拥有完备性和一致性!
  - **形式化武器**

一切的开端

- 哥德尔(Gödel)不完备性定理, 1931

  任何包含了算术的数学系统都不可能同时拥有完备性和一致性!

  - **形式化武器**

    形式系统 $\underset{\text{映射}}{\overset{\text{映射}}{\longleftrightarrow}}$ 自然数

一切的开端

- 哥德尔(Gödel)不完备性定理, 1931

  任何包含了算术的数学系统都不可能同时拥有完备性和一致性!

  - **形式化武器**

    形式系统 $\underset{\text{映射}}{\overset{\text{映射}}{\longleftrightarrow}}$ 自然数

    程序/数据 $\underset{\text{解码}}{\overset{\text{编码}}{\longleftrightarrow}}$ 二进制数

一切的开端

- 哥德尔(Gödel)不完备性定理, 1931

  任何包含了算术的数学系统都不可能同时拥有完备性和一致性!

  - **形式化武器**

    形式系统 $\underset{\text{映射}}{\overset{\text{映射}}{\longleftrightarrow}}$ 自然数

    程序/数据 $\underset{\text{解码}}{\overset{\text{编码}}{\longleftrightarrow}}$ 二进制数
  - **一阶谓词逻辑是完备的**
    - 命题逻辑：$\wedge, \vee, \neg$
    - 量化：$\forall, \exists$

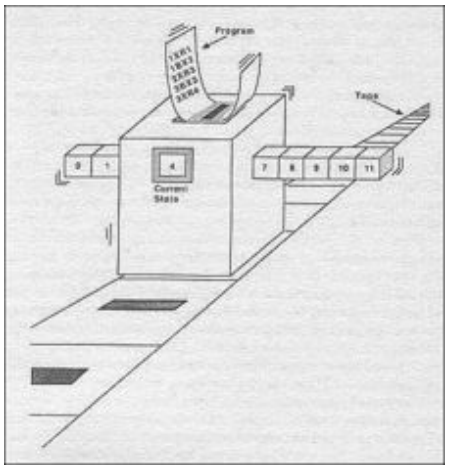
一切的开端

对于完备的系统，真理一定可以被证明，那么

- 它是否可以被**判定**呢？
- 是否能找到一种**机械计算**的方法来判断命题的真伪呢

一切的开端

- **图灵(Turing)机，1936**
  On Computable Numbers, With an Application to the Entscheidungsproblem



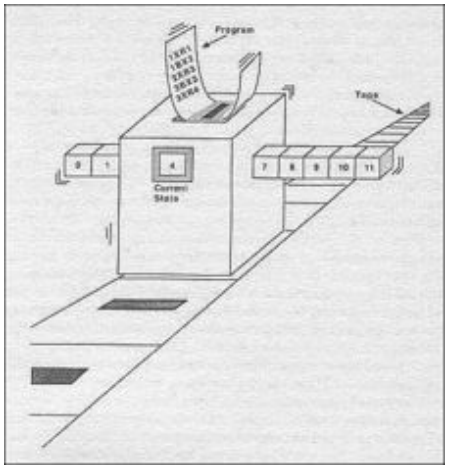
- **无限长**的纸带(tape)
- 读写头(head)
- 状态寄存器(state register)
- **有限**状态表(table)

基本操作：

- 读符号
- 不修改或者写符号
- 左移或右移纸带

一切的开端

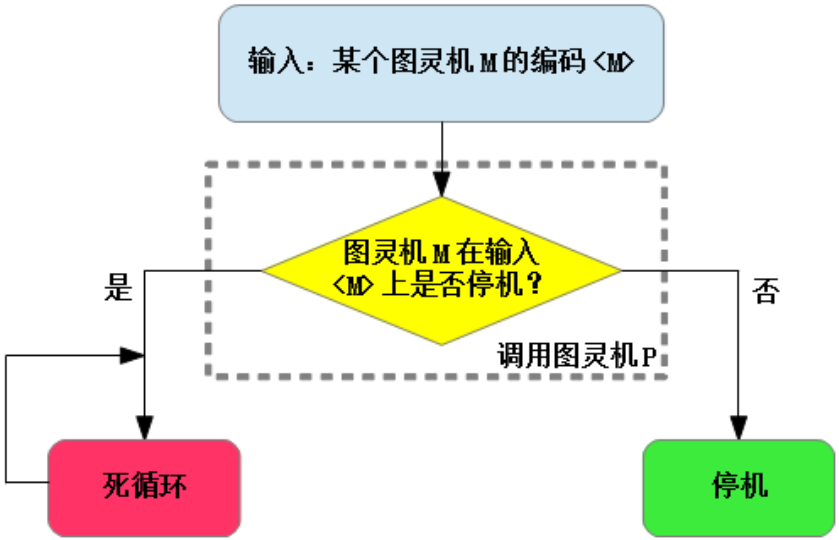
- **图灵(Turing)机，1936**



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？

一切的开端

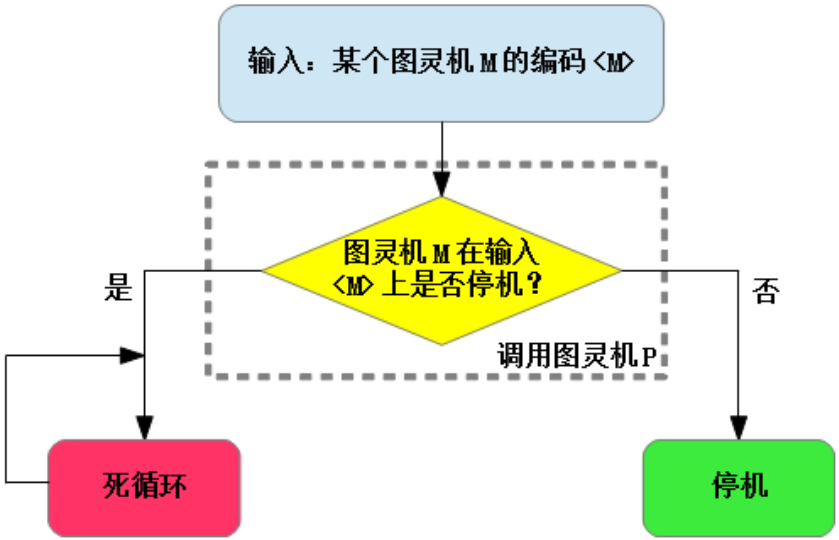
- **图灵(Turing)机，1936**
  停机问题：是否存在这样的图灵机能够判断一个程序在给定输入上是否会停机/结束？
  证明方法：反证法，输入图灵机R关于自身的编码$\langle R \rangle$



图灵机R相当于**完整的程序**，P相当于**一个函数**，M则是**输入数据**

一切的开端

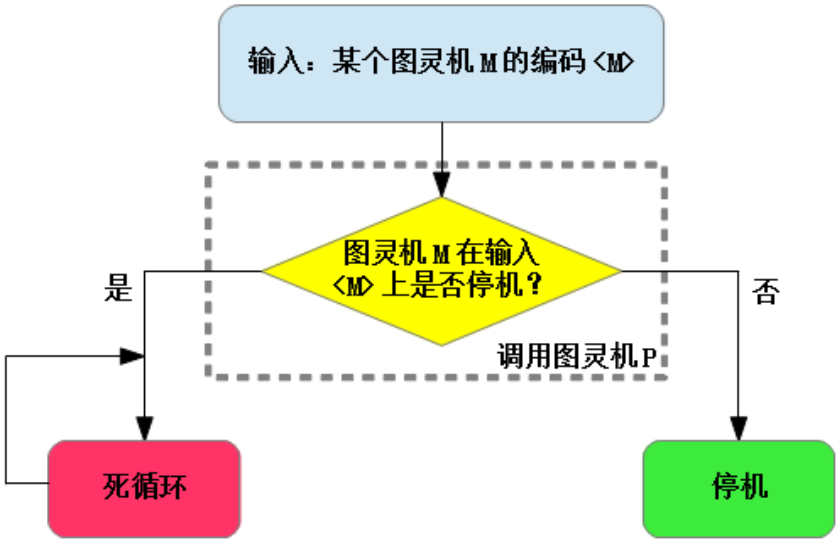
- **图灵(Turing)机，1936**
  停机问题：是否存在这样的图灵机能够判断一个程序在给定输入上是否会停机/结束？
  证明方法：反证法，输入图灵机R关于自身的编码$\langle R \rangle$



图灵机R相当于**完整的程序**，P相当于**一个函数**，M则是**输入数据**
再一次，**程序即数据!**

一切的开端

- **图灵(Turing)机，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？
  证明方法：自我指涉+自我否定→矛盾!



即使是完备的数学系统，也是**不可判定**的!

希尔伯特的幻想破灭!

故事该结束了？

λ-Calculus

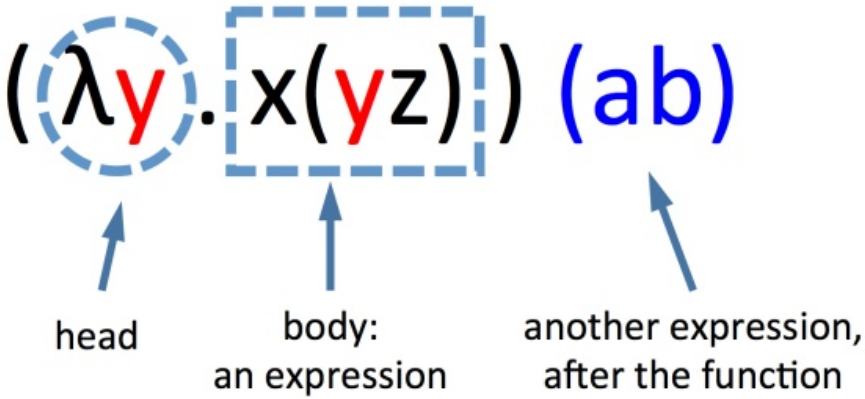
Alonzo Church, *An unsolvable problem of elementary number theory*, 1936

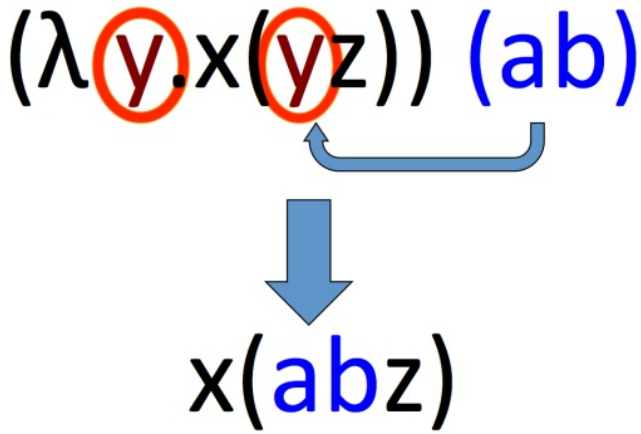
λ-item:

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

Operation:

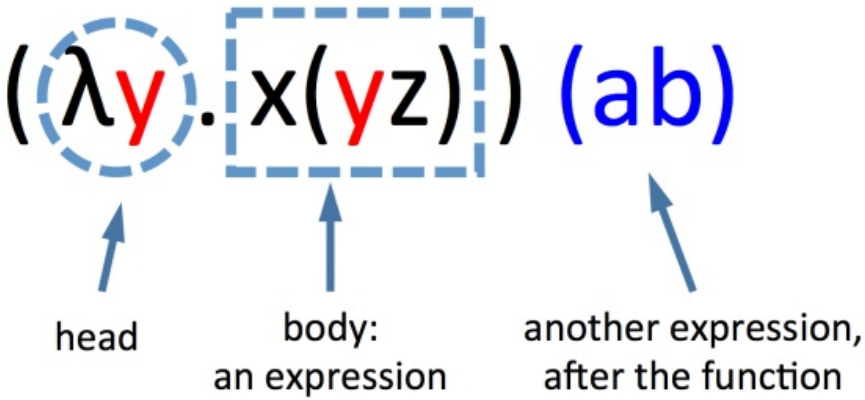
- α转换(conversion)：$(\lambda x.M[x]) \to (\lambda y.M[y])$
- β规约(reduction)：$((\lambda x.M)E \to (M[x \leftarrow E]))$

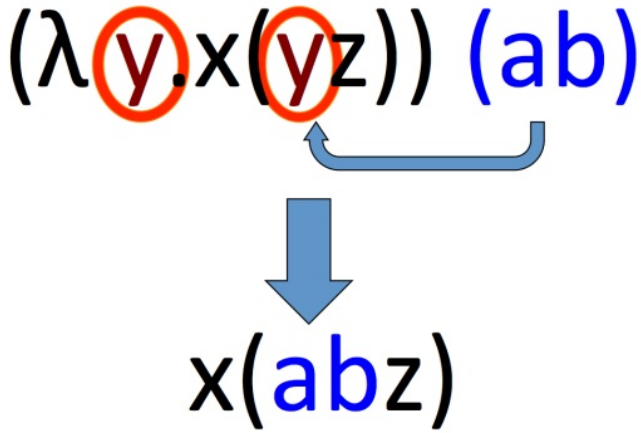




λ-Calculus

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

图灵机与λ-演算等价!

命令式语言(imperative)	函数式语言(functional)
图灵系(图灵机)	丘奇系(λ-演算)
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 C++11和Python中的lambda表达式
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

编程语言影响力网络：`https://exploring-data.com/vis/programming-languages-influence-network/`

图灵完备

如果一个系统（指令集、编程语言、元胞自动机等）能够模拟图灵机，则它是**图灵完备**的

- 图灵完备的基本条件是有条件分支(`if`)、跳转指令(`goto`)，及修改内存的能力
- 若不考虑内存的限制，绝大多数编程语言都是图灵完备的

图灵完备

如果一个系统（指令集、编程语言、元胞自动机等）能够模拟图灵机，则它是**图灵完备**的

- 图灵完备的基本条件是有条件分支(`if`)、跳转指令(`goto`)，及修改内存的能力
- 若不考虑内存的限制，绝大多数编程语言都是图灵完备的
  - C++ Template: ✓

图灵完备

如果一个系统（指令集、编程语言、元胞自动机等）能够模拟图灵机，则它是**图灵完备**的

- 图灵完备的基本条件是有条件分支(`if`)、跳转指令(`goto`)，及修改内存的能力
- 若不考虑内存的限制，绝大多数编程语言都是图灵完备的
  - C++ Template: ✓
  - Regular Expression: ✗

计算模型

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(Computability)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）

计算模型

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(Computability)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）

* 超计算(Hyper-computation)模型:
  - 谕示机(Oracle machine)
  - 量子图灵机(QTM)?

计算模型

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(Computability)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）

* 超计算(Hyper-computation)模型:
  - 谕示机(Oracle machine)
  - 量子图灵机(QTM)？区分可计算性和时间复杂度!

λ-演算更像是智慧的推理

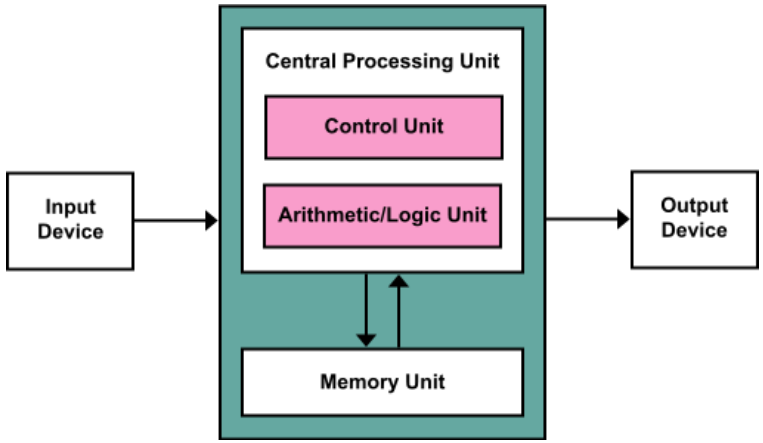
而图灵机真正抓住了 **机械计算** 的神韵

1.2

冯诺依曼体系结构

冯诺依曼体系结构

- 图灵机是**理论**模型，简单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[1], 1945)

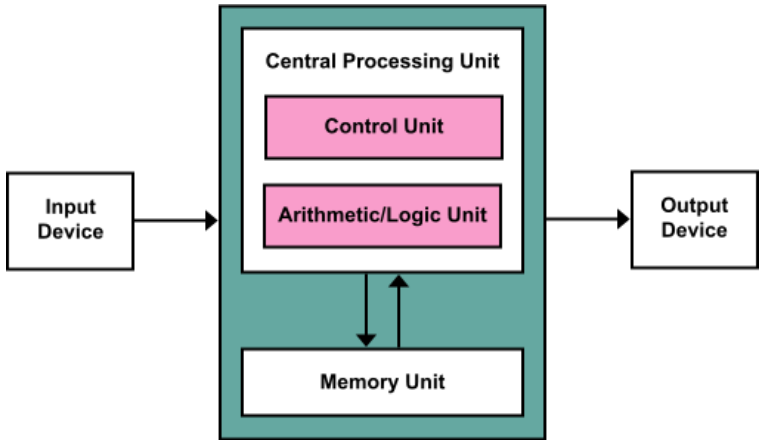


- **存储器！二进制！**

[1]区别于第一台**通用**电子计算机, ENIAC, 1943

冯诺依曼体系结构

- 图灵机是**理论**模型，简单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[1], 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]区别于第一台**通用**电子计算机, ENIAC, 1943

冯诺依曼体系结构

为什么采用二进制？

- 二进制只有两种状态，容易找到具有两个稳定状态并且状态转换容易控制的物理器件（物理实施）
- 二进制可靠性强，噪声容限高，且节省大量的空间资源开销（数字电路）
- 二进制编码运算规则简单，0、1与二值逻辑一致，容易实现逻辑运算（布尔代数）

程序即数据

程序即数据的思想衍生出了后来很多分支

- 元编程(metaprogramming)：能够写程序的程序，输出数据是一个程序，如C宏

程序即数据

程序即数据的思想衍生出了后来很多分支

- 元编程(metaprogramming)：能够写程序的程序，输出数据是一个程序，如C宏
- 性能剖析器(profiler)：对程序的分析，输入数据是一个程序，如杀毒软件

程序即数据

程序即数据的思想衍生出了后来很多分支

- 元编程(metaprogramming)：能够写程序的程序，输出数据是一个程序，如C宏
- 性能剖析器(profiler)：对程序的分析，输入数据是一个程序，如杀毒软件
- 编译器：程序的优化，本身是一个程序，输入、输出数据也都是程序

程序即数据

程序即数据的思想衍生出了后来很多分支

- 元编程(metaprogramming)：能够写程序的程序，输出数据是一个程序，如C宏
- 性能剖析器(profiler)：对程序的分析，输入数据是一个程序，如杀毒软件
- 编译器：程序的优化，本身是一个程序，输入、输出数据也都是程序
- 神经网络：TVM Relay (ongoing)，既然神经网络是一个程序，那能否类似编译一样进行优化？
  Key idea：图灵完备，自动优化，高阶微分

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 $1\text{Word}=4\text{B}=32\text{bit}$
  但注意在操作系统中，word为16位，dword为32位
- 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

2.1

指令集背景

指令集背景

Machine Language (Binary)[1945]

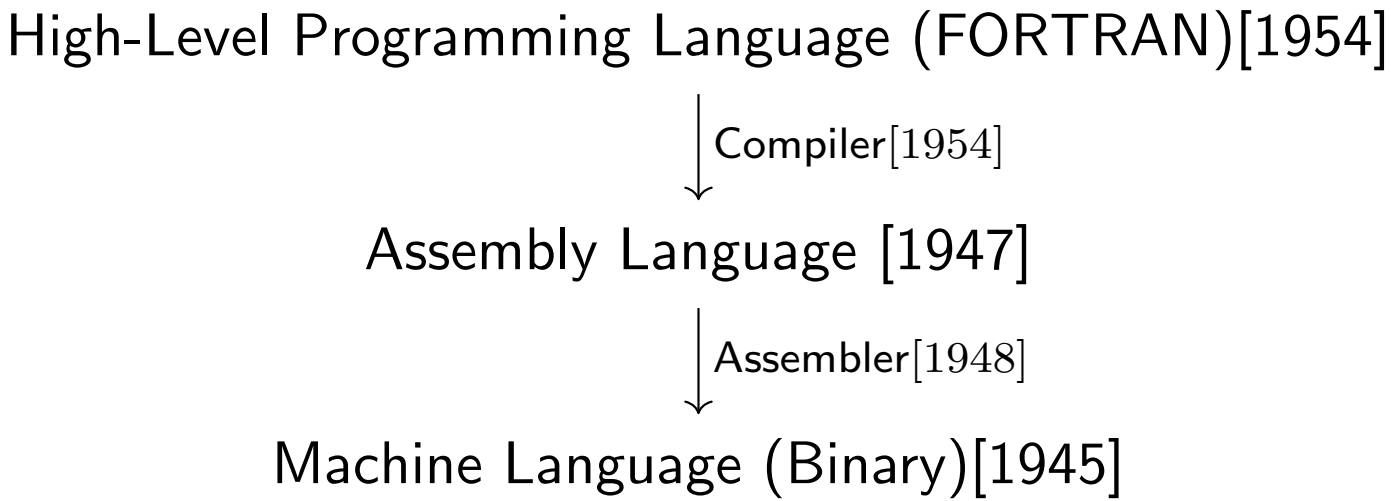
指令集背景

Assembly Language [1947]

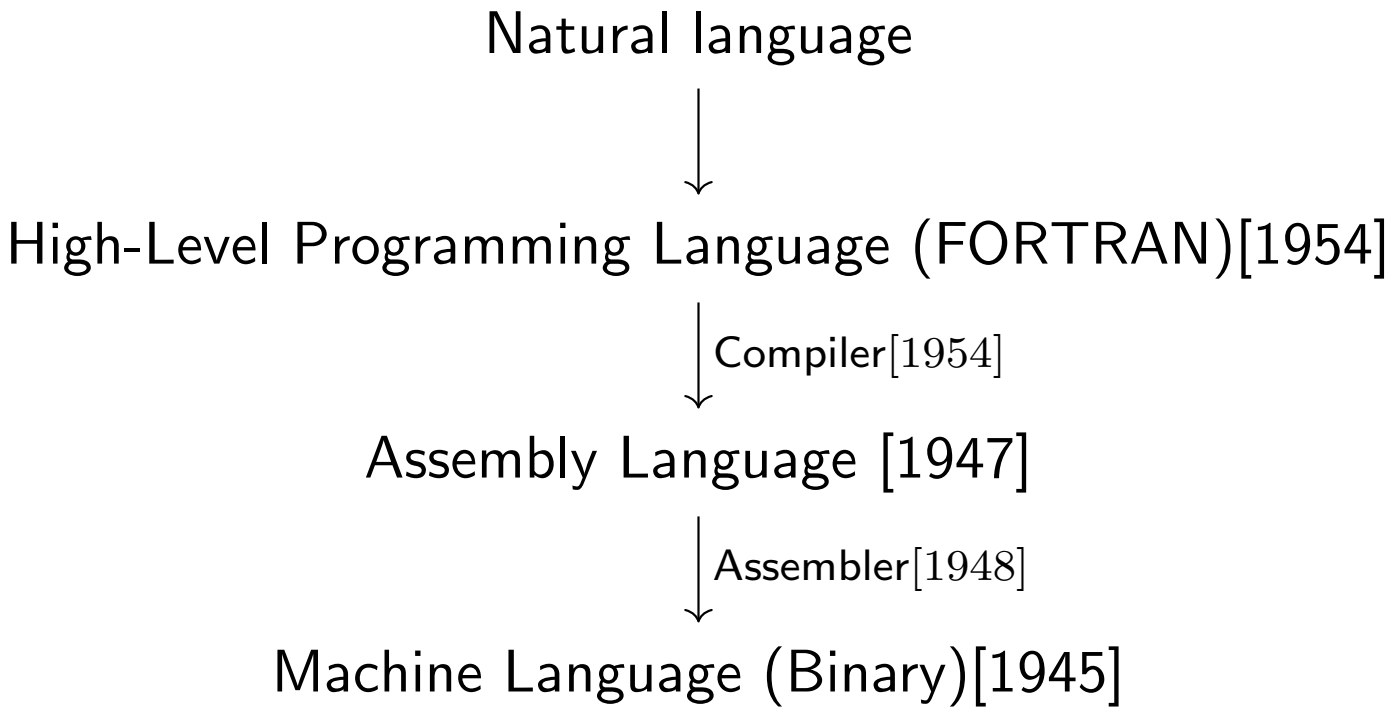
$\downarrow Assembler[1948]$

Machine Language (Binary)[1945]

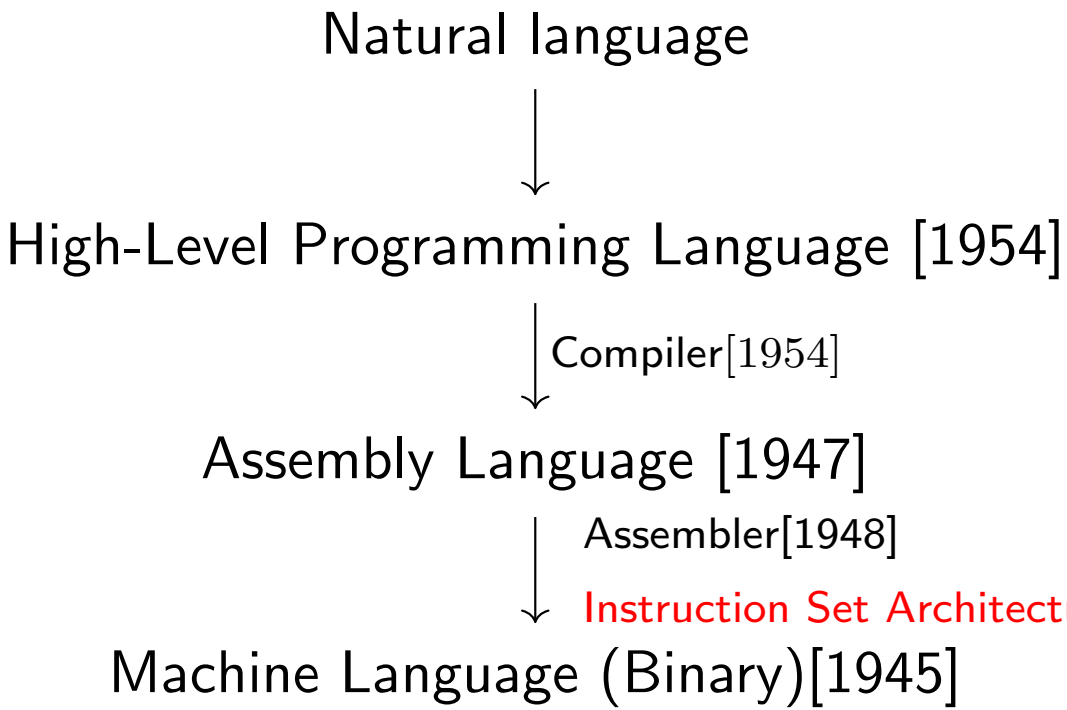
指令集背景


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

2.2

指令集计算机

CISC与RISC之争

变长指令集

CISC(Complex Instruction Set Computer)的天下：1970s

- 内存少→程序要小→代码密度高
- 编译器弱→寄存器分配难，无法针对微结构进行深层优化
- mem-mem，reg-mem模式

CISC与RISC之争

变长指令集

CISC(Complex Instruction Set Computer)的天下：1970s

- 内存少→程序要小→代码密度高
- 编译器弱→寄存器分配难，无法针对微结构进行深层优化
- mem-mem，reg-mem模式

泥沼：

- 越来越多指令添加进指令集
- 研发成本提升，周期变长
- 殃及软件，甚至编译器都不知如何用这么庞大的指令集

CISC与RISC之争

定长指令集

John Cocke (IBM), David Patterson (UCB), John Hennessy (Stanford)

RISC(Reduce Instruction Set Computer)崛起：1980s

- 寄存器着色：Graph coloring
- 编码整洁，流水线设计容易

CISC与RISC之争

定长指令集

John Cocke (IBM), David Patterson (UCB), John Hennessy (Stanford)

RISC(Reduce Instruction Set Computer)崛起：1980s

- 寄存器着色：Graph coloring
- 编码整洁，流水线设计容易

存在的问题：

- 指令定长，大常数/地址需要拆分为多块
- 代码密度不高，某些情况下浪费指令缓存的容量和带宽

CISC与RISC之争

CISC与RISC交互融合：

- 编译技术的提升：寄存器重命名
- 体系结构的发展：微程序
- Intel x86为“兼容”需要，保留CISC风格，同时借鉴了RISC思想
- RISC引入代码密度更高的新指令集，以弥补指令缓存等劣势

指令集计算机

复杂指令集计算机	精简指令集计算机
CISC	RISC
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，编译系统易优化
变长指令字	定长指令字
x86	ARM(Advanced RISC Machine), MIPS, SPARC

处理器性能评价

$$\text{计算机的性能(Performance)} = 1/\text{执行时间(Execution time)}$$

按照单位（量纲）进行换算即可

$$\begin{aligned}\text{CPU执行时间(s)} &= \text{执行程序所需CPU时钟周期(cyc)} \times \text{时钟周期(s/cyc)} \\ &= \text{指令数目(ins)} \times \text{CPI(cyc/ins)} \times \text{时钟周期(s/cyc)} \\ &= \text{指令数目(ins)} \times \text{CPI(cyc/ins)}/\text{CPU主频(cyc/s)}\end{aligned}$$

程序性能对执行事件的影响：

	指令数	CPI	时钟周期
算法、编程语言、编译器	×	×	
指令集	×	×	×
计算机组成		×	×
实现技术			×

2.3

指令集系统基本内容

指令集系统

1. 什么操作？

指令集系统

1. 什么操作？操作码

指令集系统

1. 什么操作？操作码
2. 操作对象？

指令集系统

1. 什么操作？操作码
2. 操作对象？操作数

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？

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

$$\text{指令} = \text{操作码} + \text{地址码}$$

寻址方式

立即数寻址	直接给出操作数本身，无需访存快速，操作数大小受地址字段长度限制，大量使用	MOV AX, 1000H
存储器直接寻址	操作数在存储器中，直接给出操作数在存储器中的地址，寻址空间受指令地址字段长度限制，较少使用	MOV AX, [1000H]
存储器间接寻址	存储器中的内容是操作数的地址，需二次寻址	
寄存器直接寻址	直接给出寄存器编号，无需访存速度快，地址范围有限，可用通用寄存器较少，使用最多，提高性能常用手段	MOV AX, BX
寄存器间接寻址	寄存器中的内容是操作数的地址，二次寻址	MOV AX, [BX]
相对寻址（偏移）	相对**当前指令(PC)**位移量为A的单元，跳转指令	EA=(PC)+A
基址寻址（偏移）	相对基址(B)位移量为A的单元，**OS页面（重定位）**，面向系统，程序逻辑空间与存储器物理空间的无关性	EA=(B)+A
变址寻址（偏移）	相对形式地址A（数组基址）位移量为(I)的单元，X为数组元素大小，面向用户	EA=(I)+A, I=(I)±X
堆栈寻址	从寄存器到堆栈或反过来，指令短	EA=栈顶(SP)
复合寻址	间接寻址+相对/变址寻址	间接相对EA=(PC)+A 相对间接EA=((PC)+A)

注意，(X)代表X地址/寄存器内的内容，如((X))代表寄存器间接寻址。注意看题目中是按**字**编址还是按**字节**编址。但从80年代开始，几乎所有机器都采用字节编址(byte addressing)。

操作指令

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

为什么是这些指令？

操作指令

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

为什么是这些指令？ **图灵完备!**

指令集设计的难点

指令的位数十分有限

- 完备性：可以解决任何可解的问题
- 有效性：简洁、无歧义、加速常用操作
- 规整性：简单源于规整，相同位置含义相同
- 兼容性：向前/向后兼容

对于RISC来说，如果操作码多了，意味着操作数少了；
反之，操作码少操作数多
需要**极其精巧的设计与权衡!**

3

RISC-V指令集总览

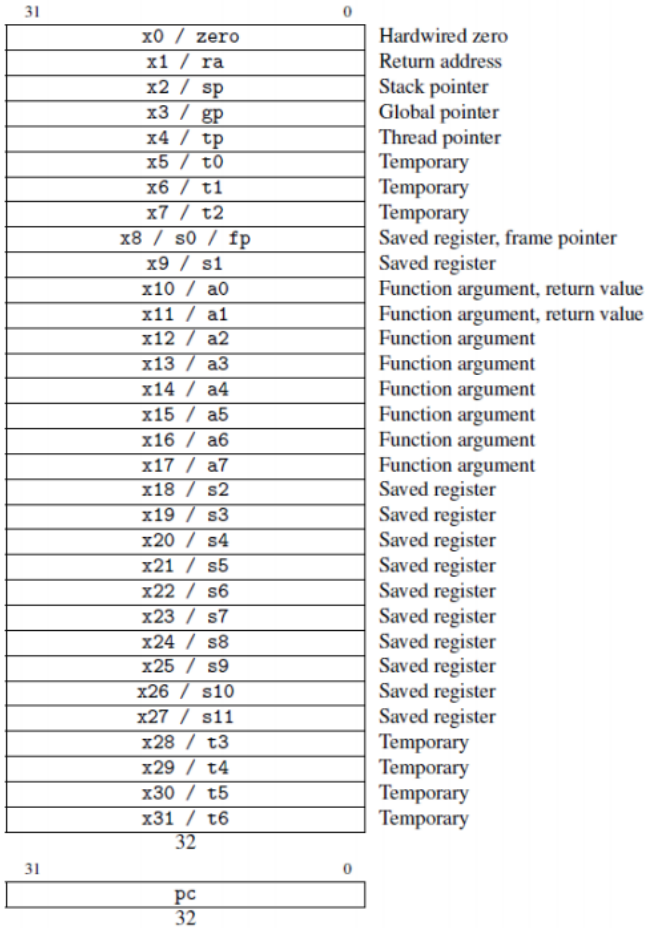
RISC-V指令集

这里主要介绍32位指令，64位类似

- 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($2^6 = 64$)
- 仅提供一种数据寻址模式（寄存器+立即数）

Registers

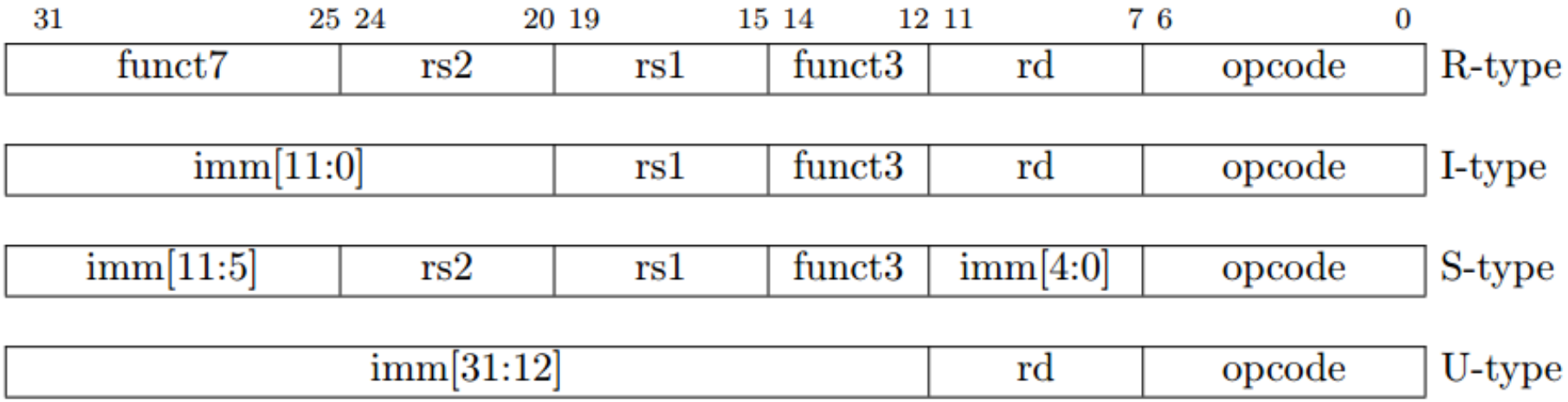
x1-x31, x0=0, Program Counter (PC)



- 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，且PC包含在内

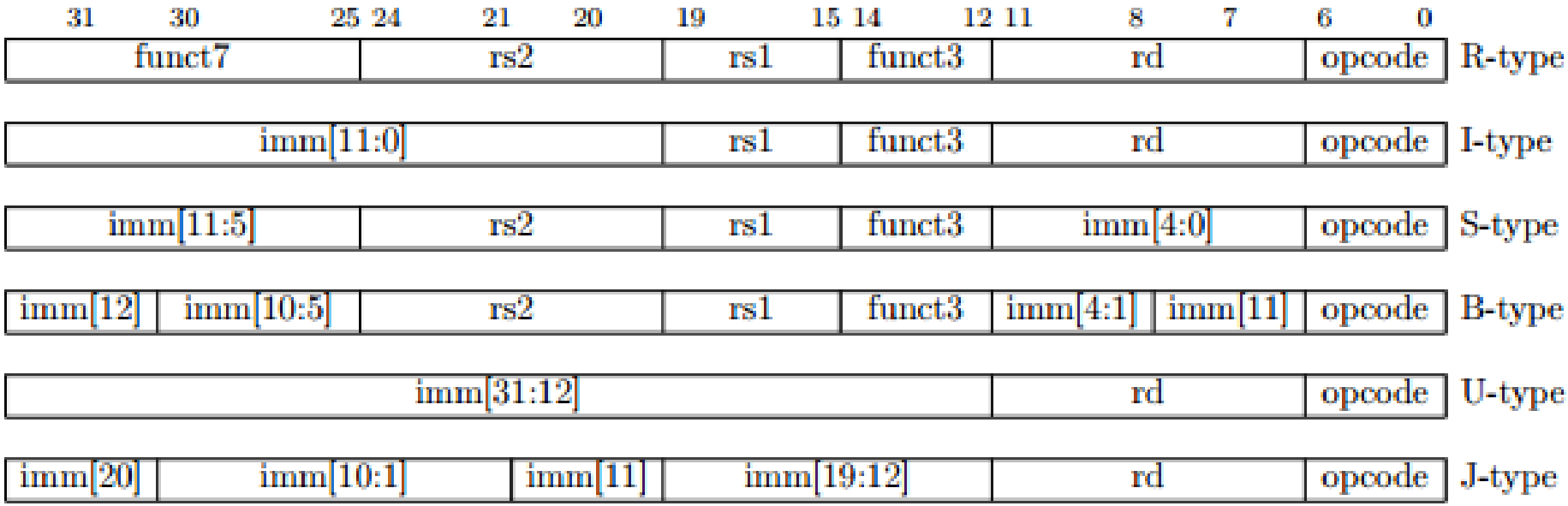
为什么x86寄存器数目这么少？

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)

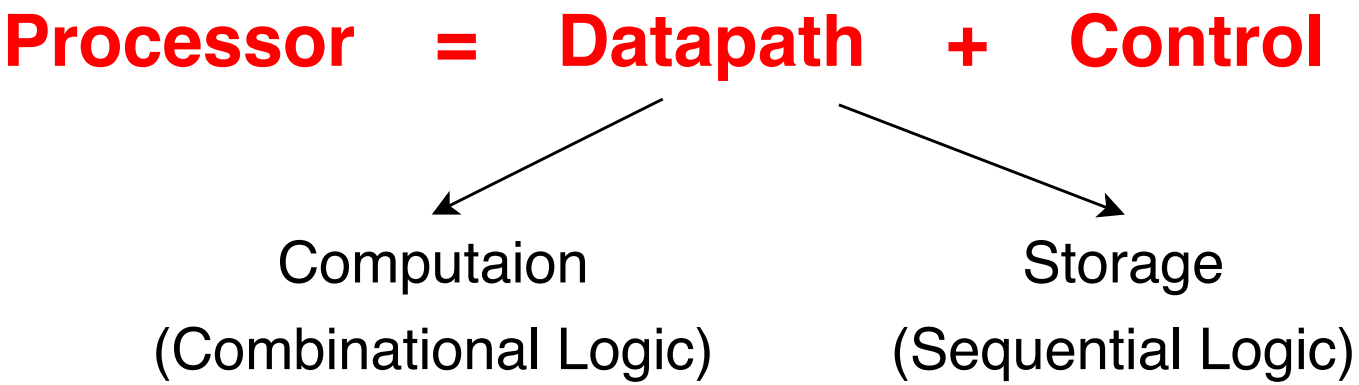
4

单周期CPU

4.1

Introduction

Central Processor Unit (CPU)



既然**指令和数据**在计算机中都以二进制形式表示，那么CPU到底是怎么区别它们的？

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 x3, x2, x1`
- `add x3, x2, 10`
- `lw x3, x2(4)`
- `sw x3, x2(4)`
- `beq x3, x2, -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

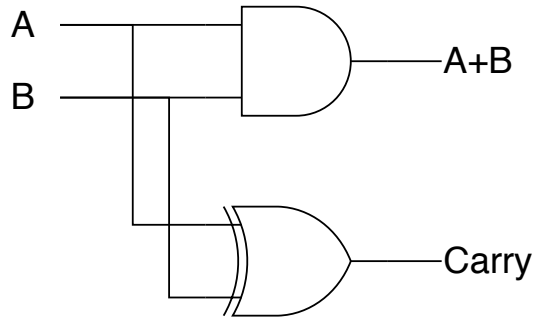
计算器件层

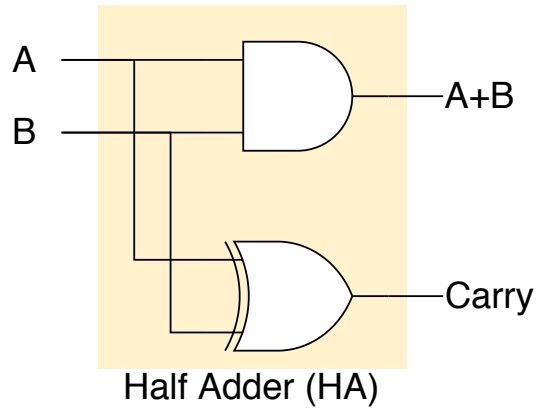
1-2: 基本数学运算能够用布尔运算模拟

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

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

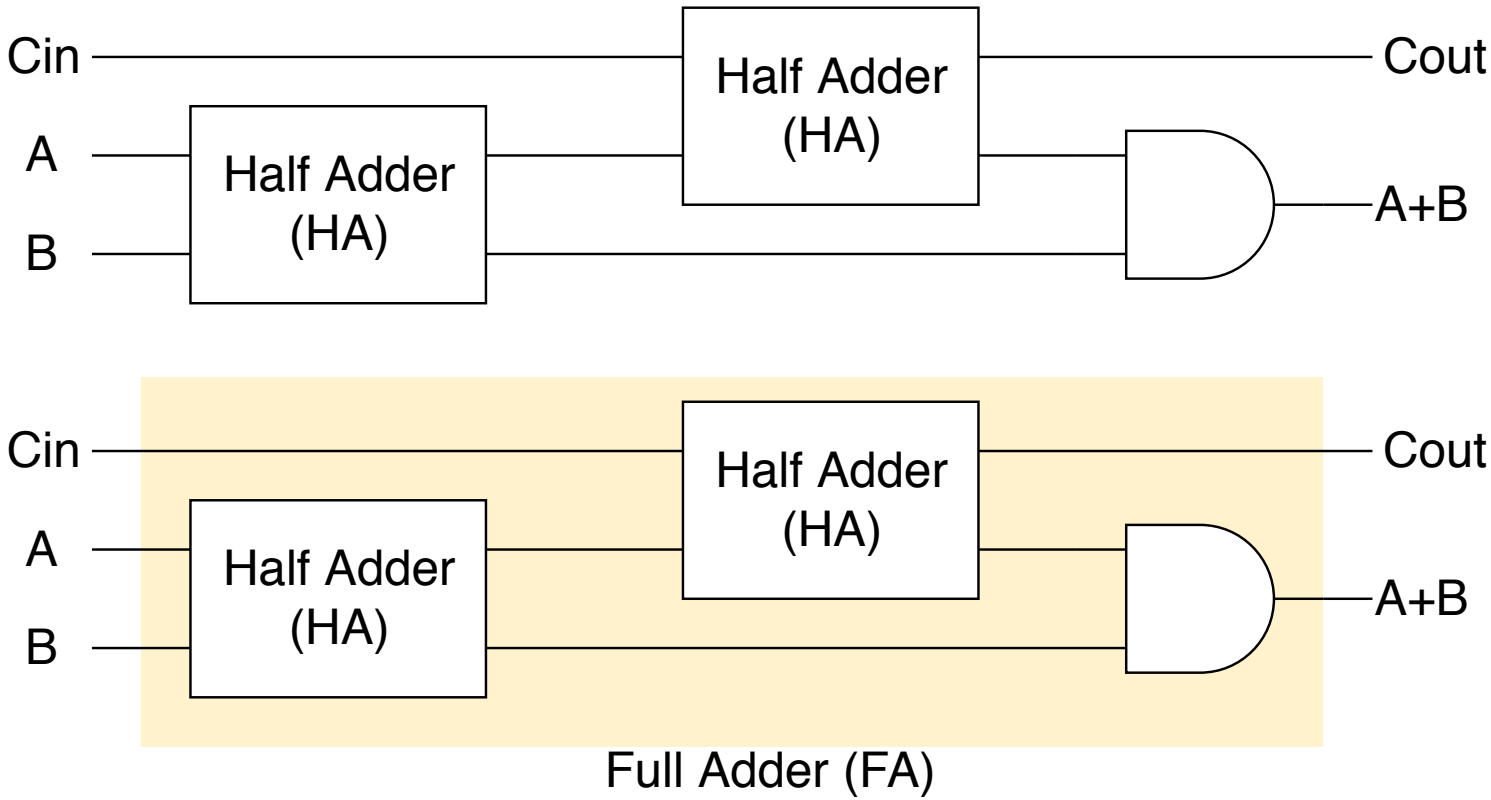
$$Carry = AB$$





计算器件层

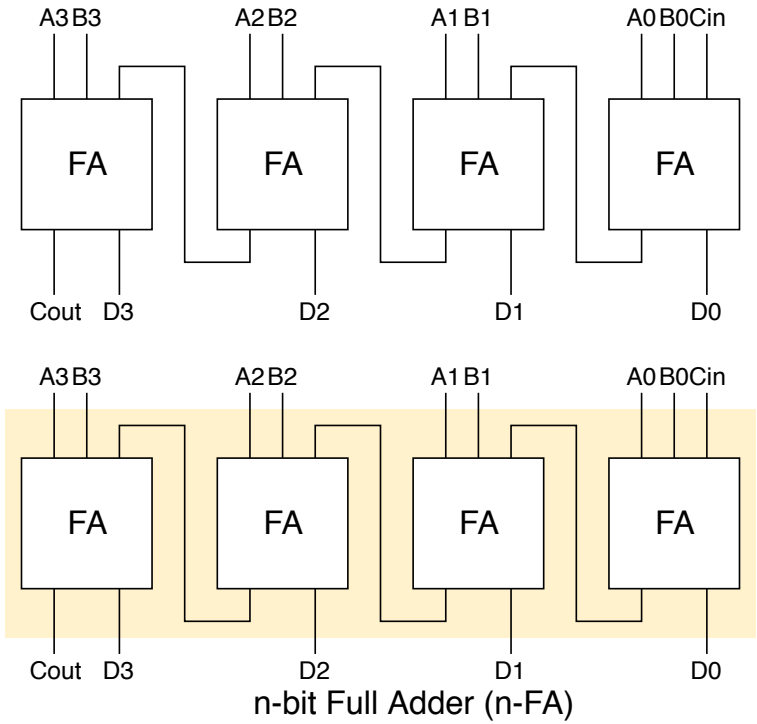
1-2: 基本数学运算能够用布尔运算模拟



物理电路层→布尔逻辑层→计算器件（**组合逻辑**）

计算器件层

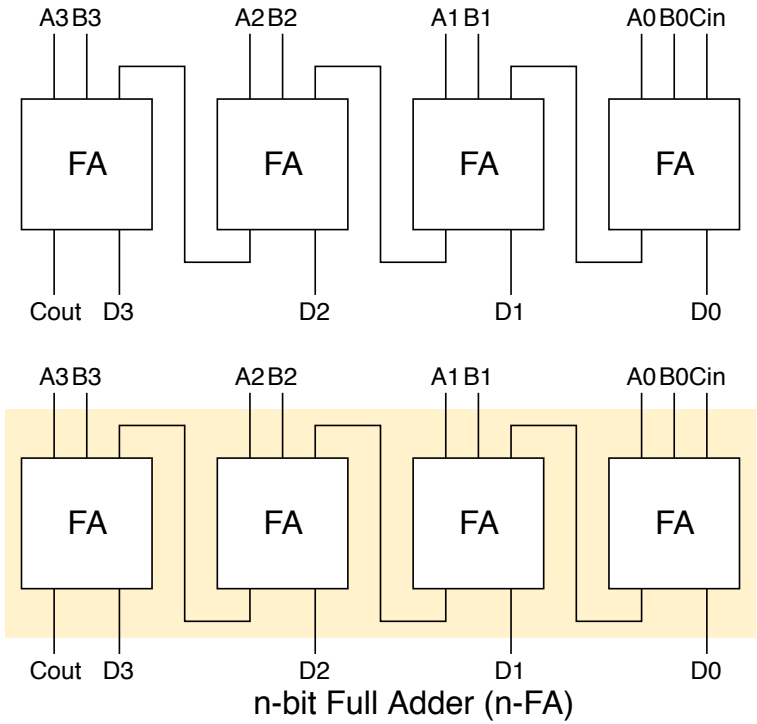
1-2: 基本数学运算能够用布尔运算模拟



物理电路层→布尔逻辑层→计算器件（**组合逻辑**）

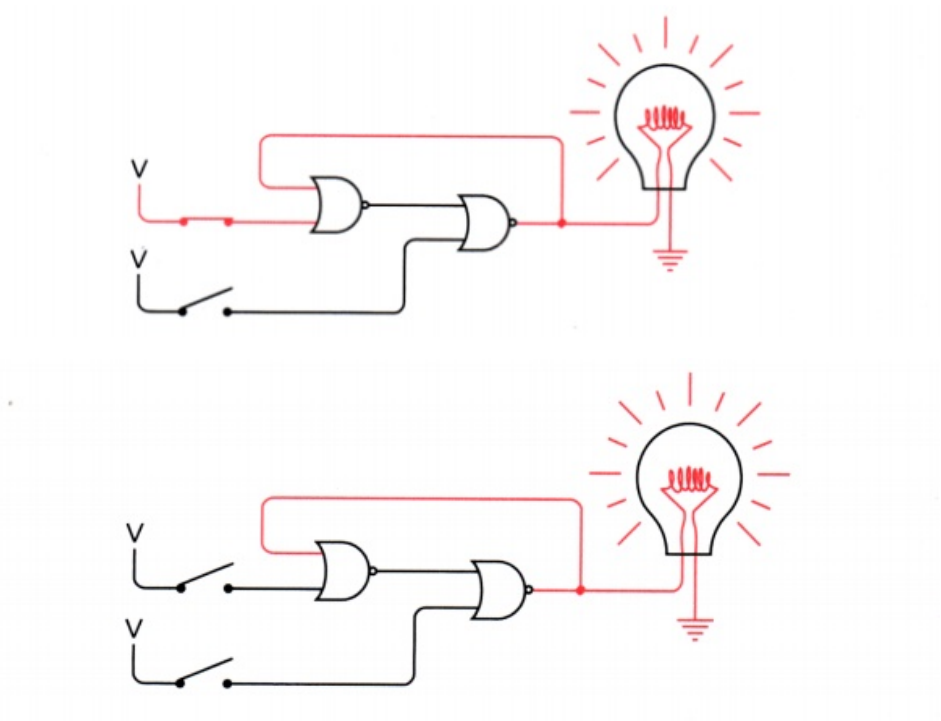
计算器件层

1-2: 基本数学运算能够用布尔运算模拟

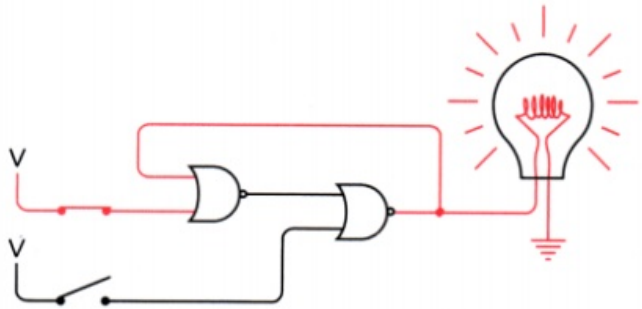


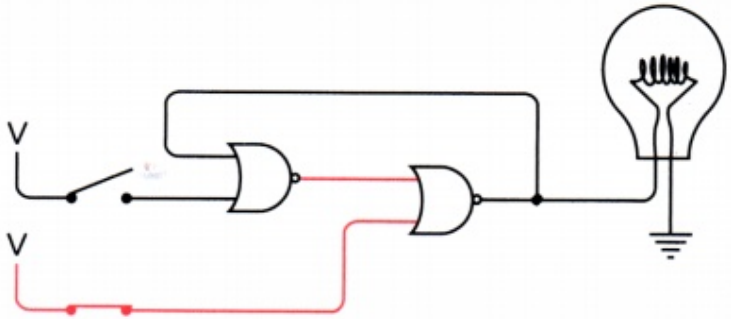
物理电路层→布尔逻辑层→计算器件（**组合逻辑**）

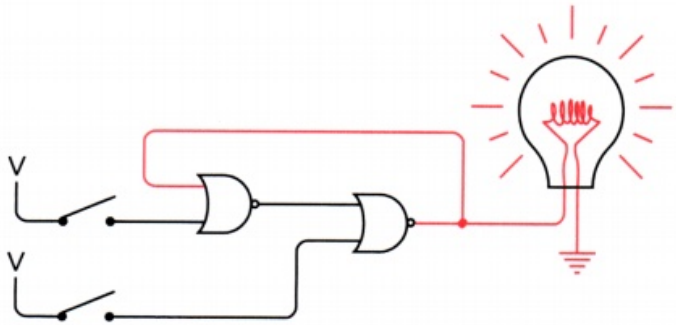
计算器件层（时序逻辑）

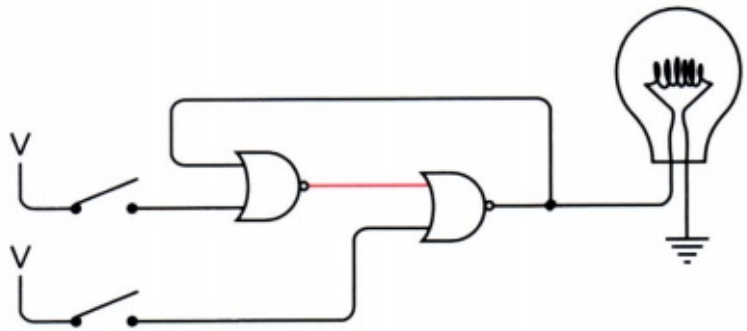


计算器件层（时序逻辑）

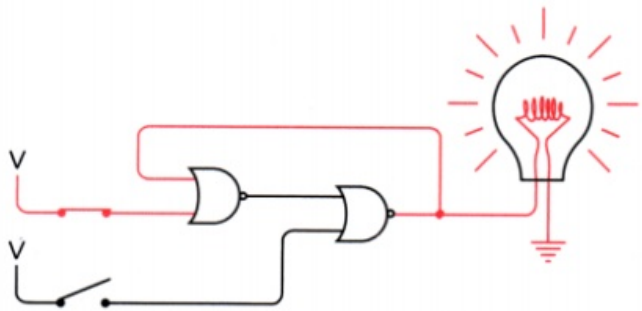


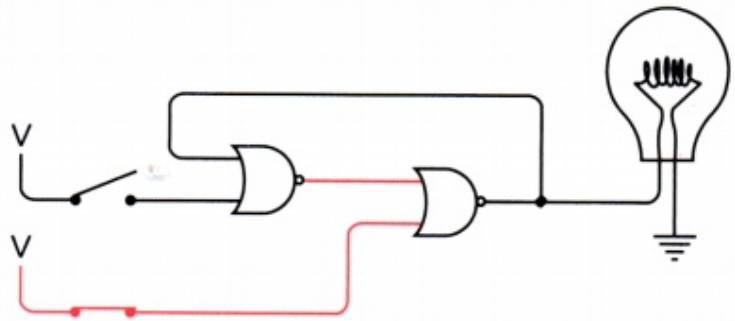


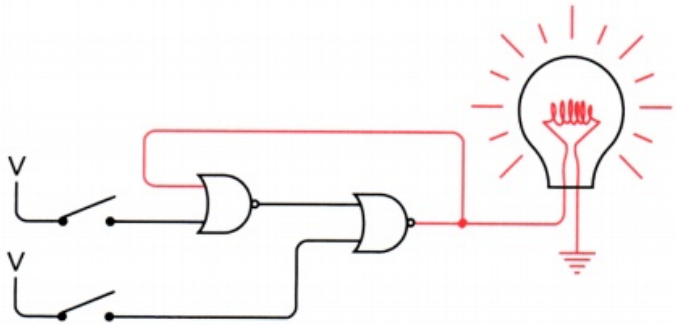


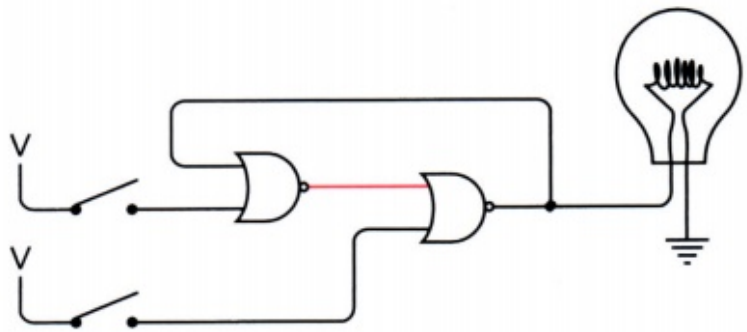


计算器件层（时序逻辑）







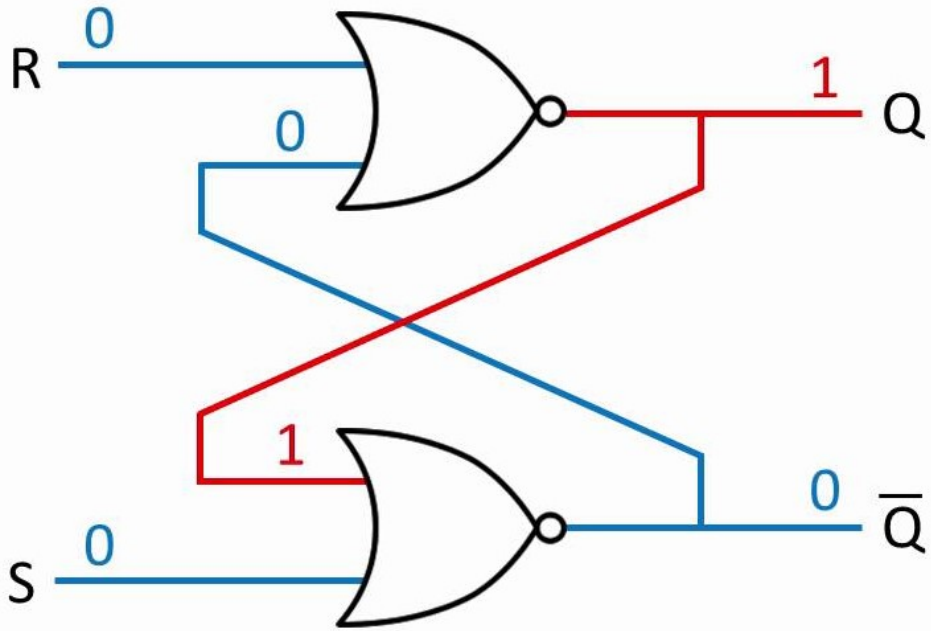


锁存器(Latch)！记忆信息！

计算器件层（时序逻辑）

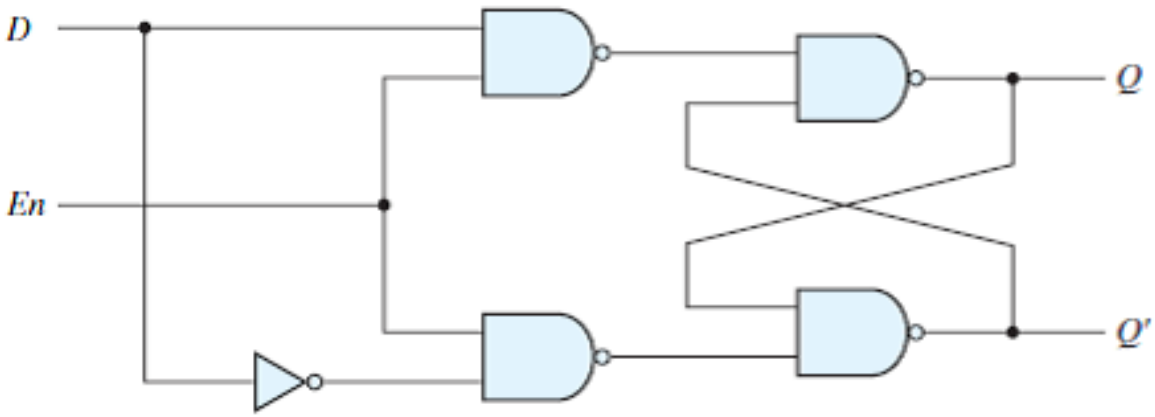
SR Latch

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

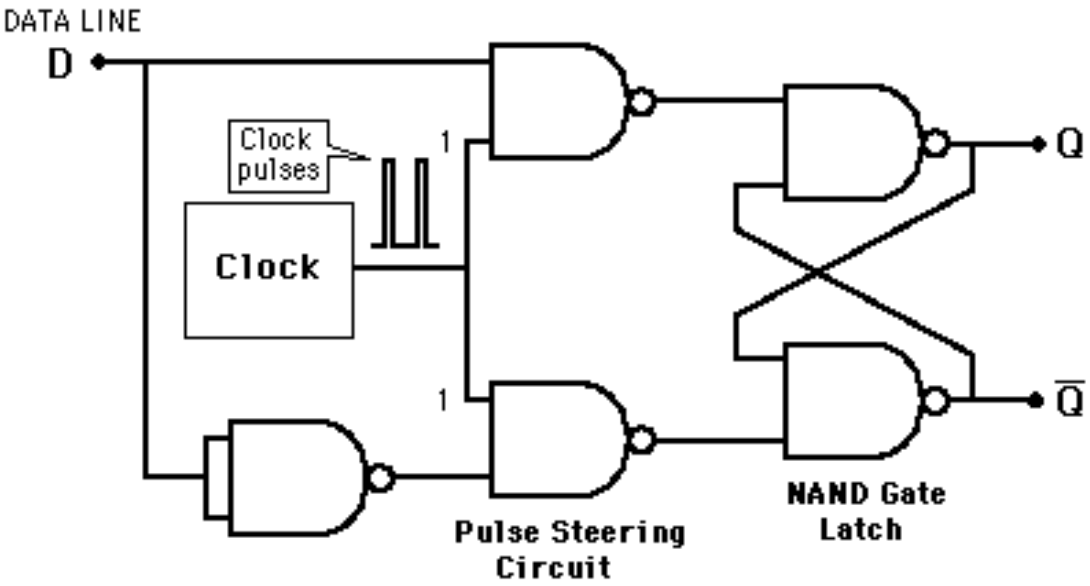
En	D	Next state of Q
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

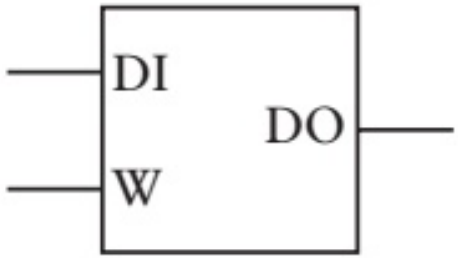
(b) Function table

FIGURE 5.6
D latch

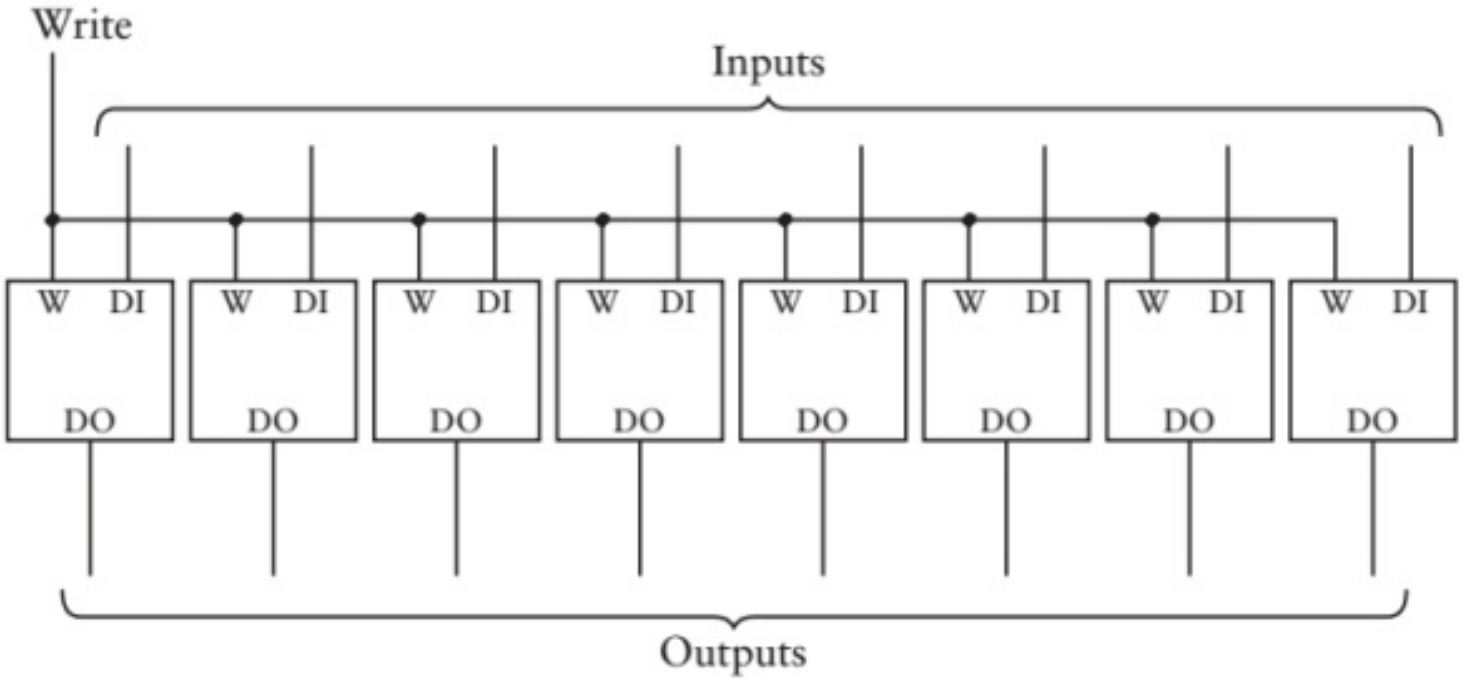
计算器件层（时序逻辑）

D Flip-flop (触发器) – 对**时钟边沿**敏感





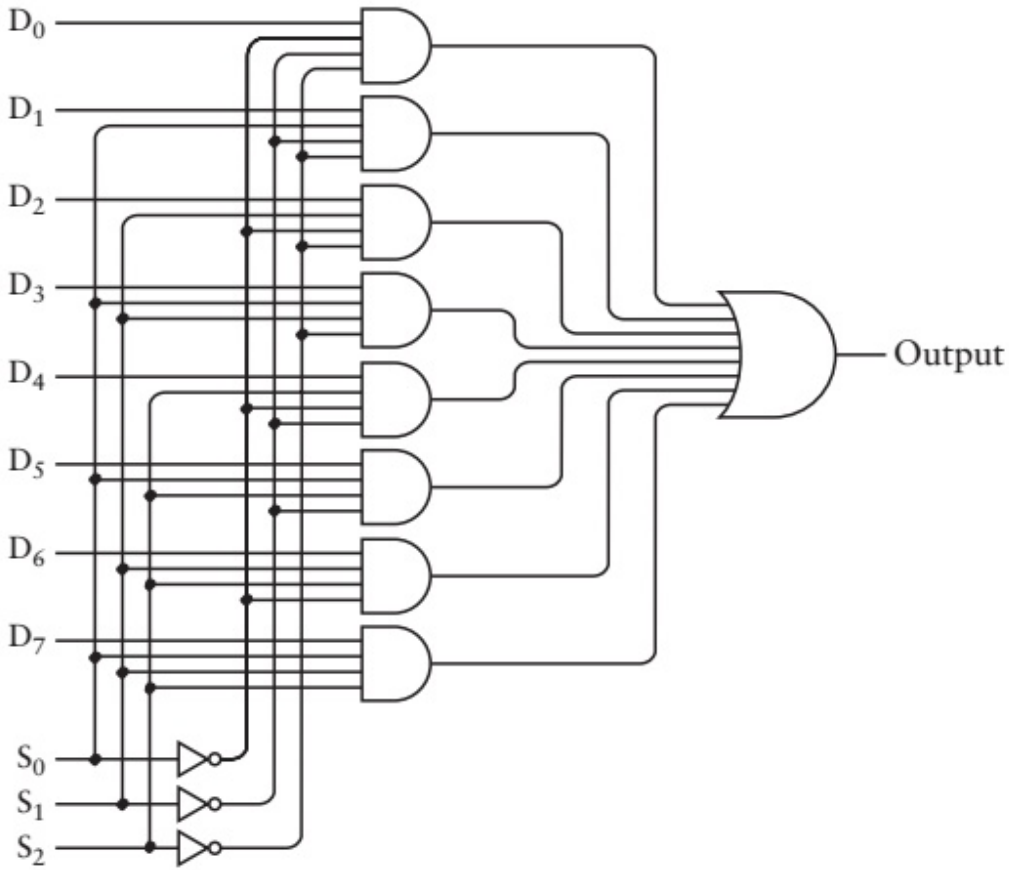
计算器件层（时序逻辑）


Write
Inputs
W DI
DO
Outputs


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

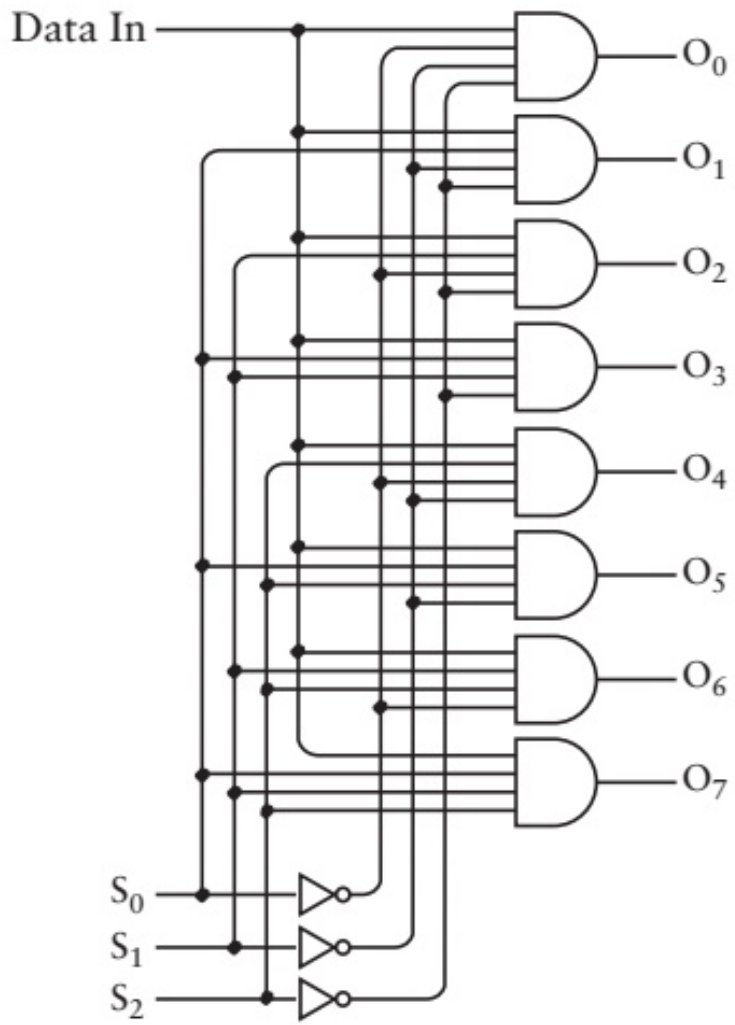
计算器件层

8-1 Multiplexer



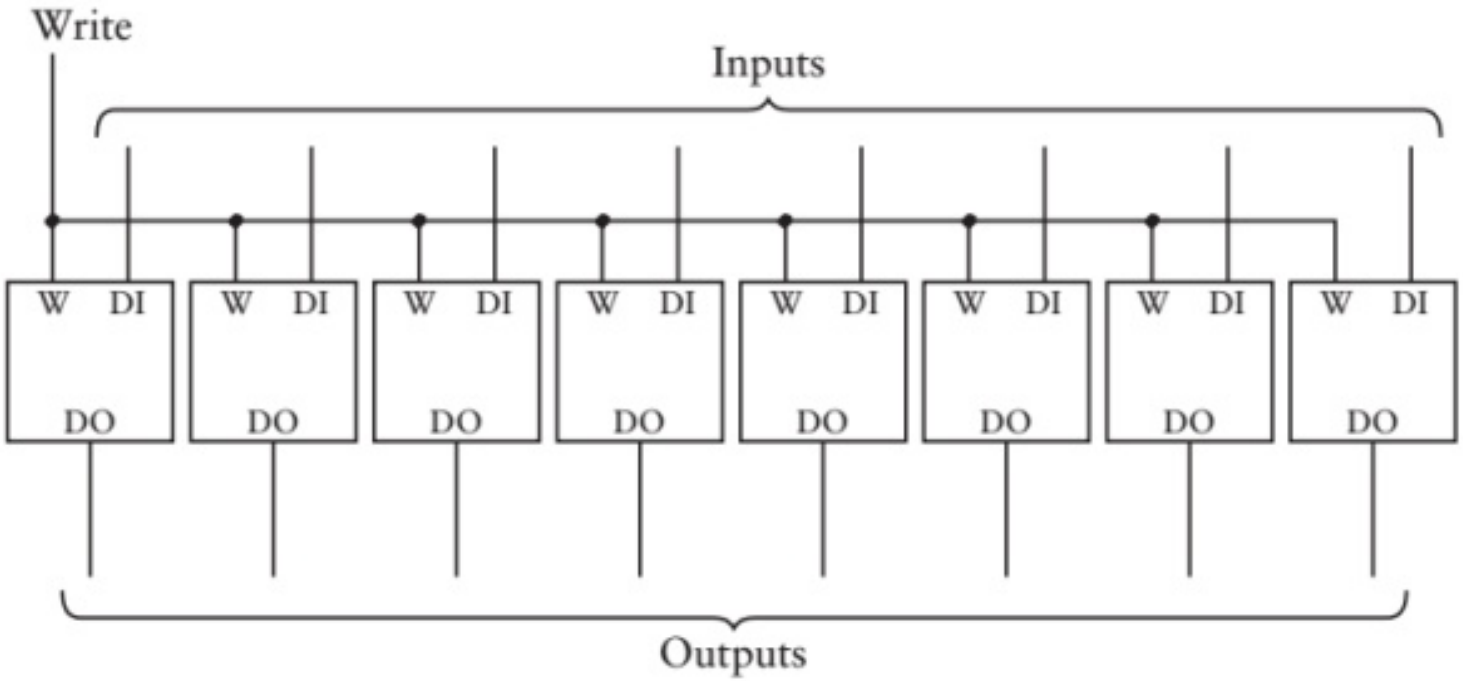
计算器件层

3-8 Decoder



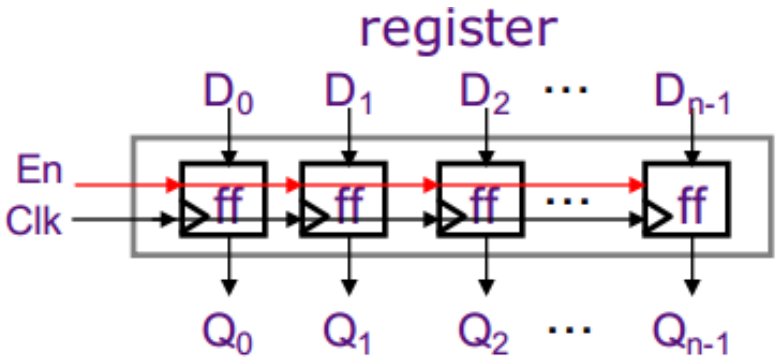
计算器件层

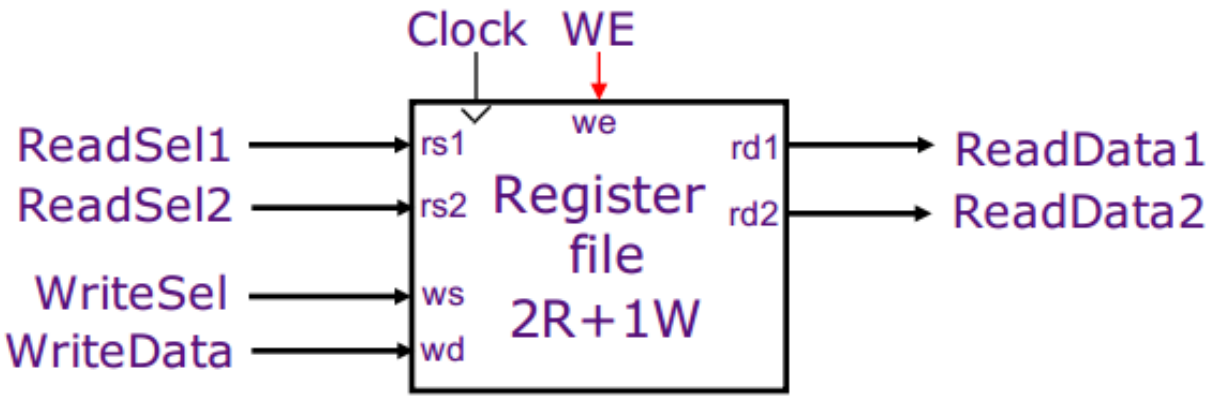
8-bit Memory (RAM)



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)

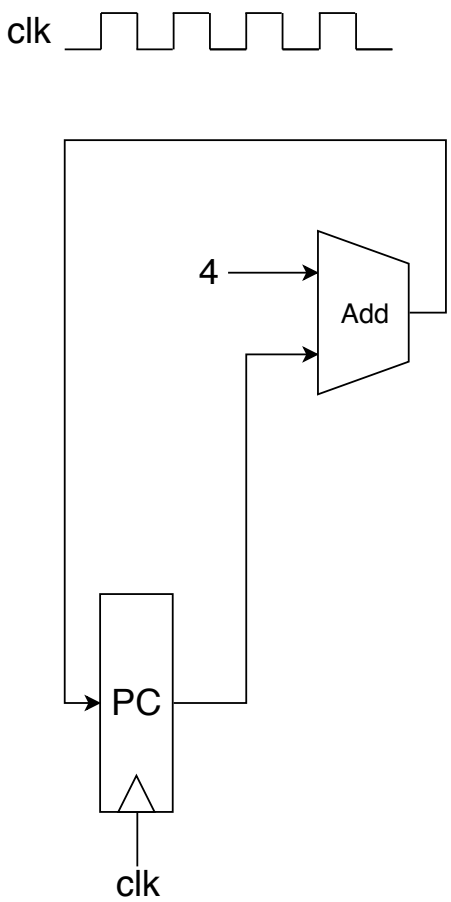
架构层

4.3

Common Datapath

共同通路(PC)

Program Counter (PC)



按字编址

```
                main:
0x00000000          xor x1, x1, 0
0x00000004          add x2, x1, 1
0x00000008          beq x3, x1, x2
0x0000000C          jal x3, 1
...
```

4.4

R-R Instructions

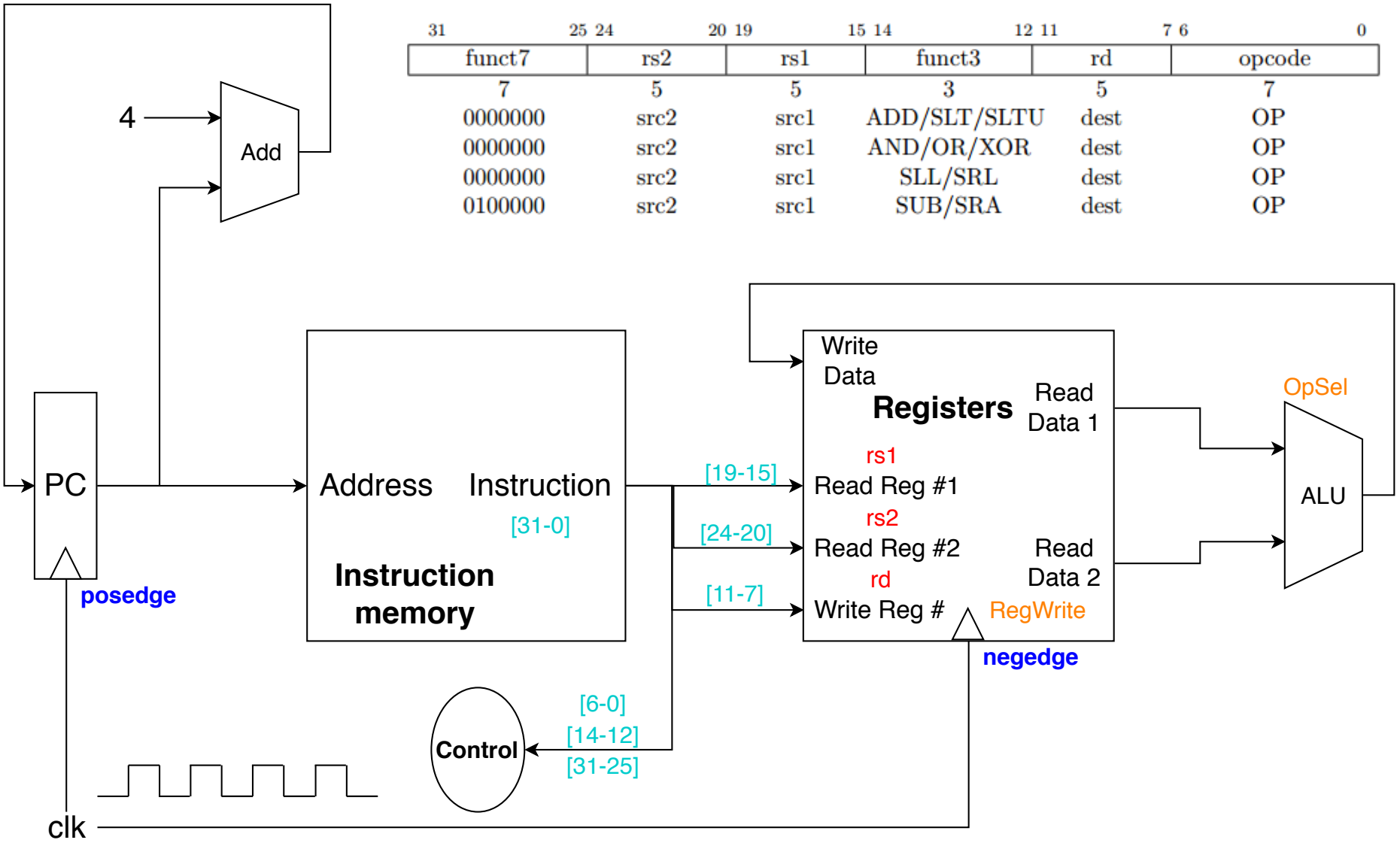
Register-Register Instructions

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)

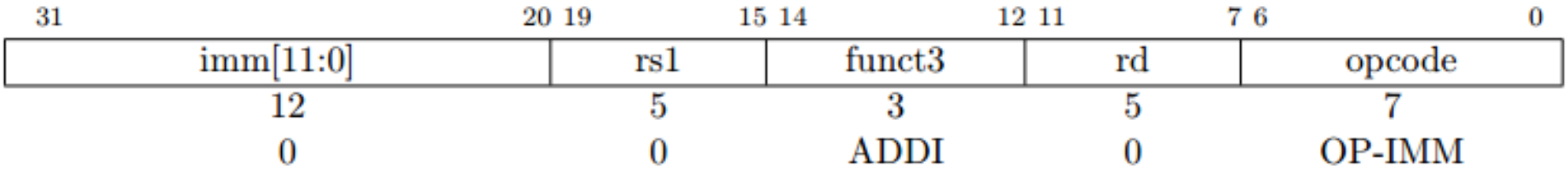
OpSel[2:0]	Function
000	$Y = A + B$
001	$Y = A - B$
010	$Y = B << A$
011	$Y = A \mid B$
100	$Y = A\&B$
101	$Y = (A < B)\ ?\ 1\ :\ 0$
110	$Y = (((A < B)\&\&(A[31] == B[31]))$ $\|\|\quad((A[31] == 1\&\&B[31] == 0)))$ $?\ 1\ :\ 0$
111	$Y = A \oplus B$

Register-Register Instructions

NO OPERATION (NOP)

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

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

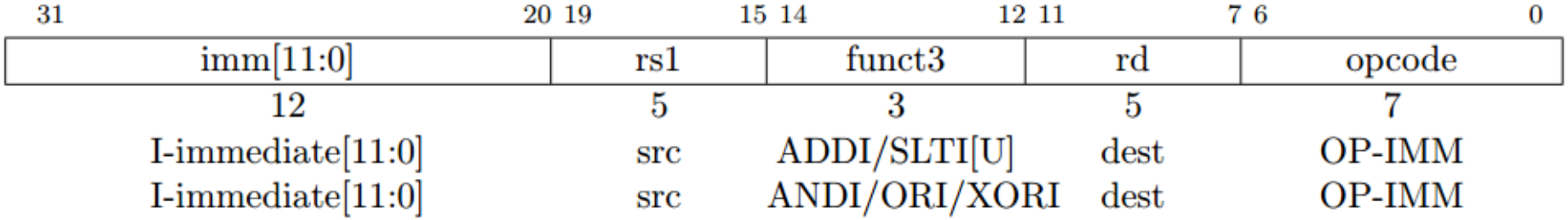


31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
0	0	ADDI	0	OP-IMM

4.5

R-I Instructions

Register-Immediate Instructions



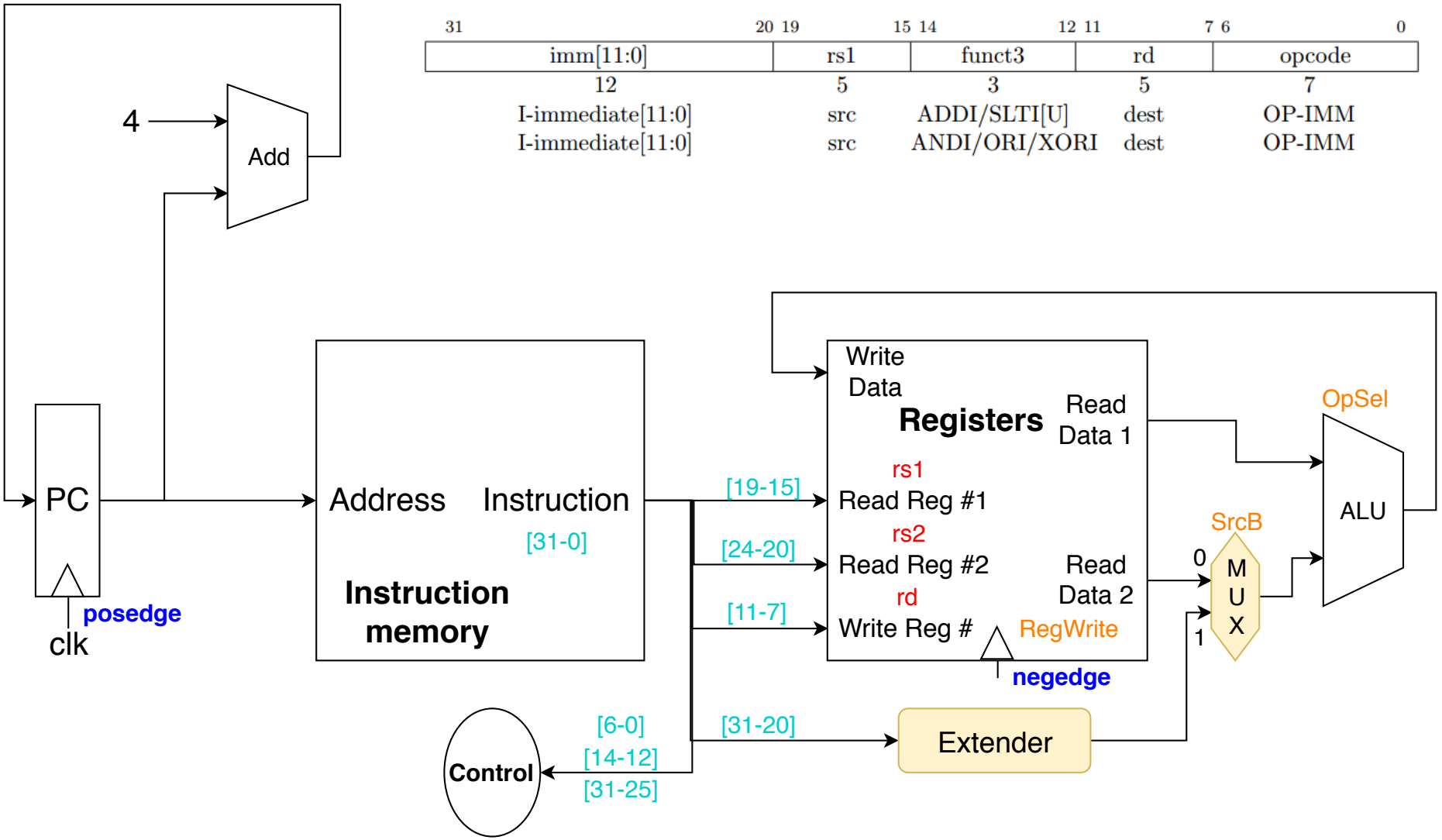
31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

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

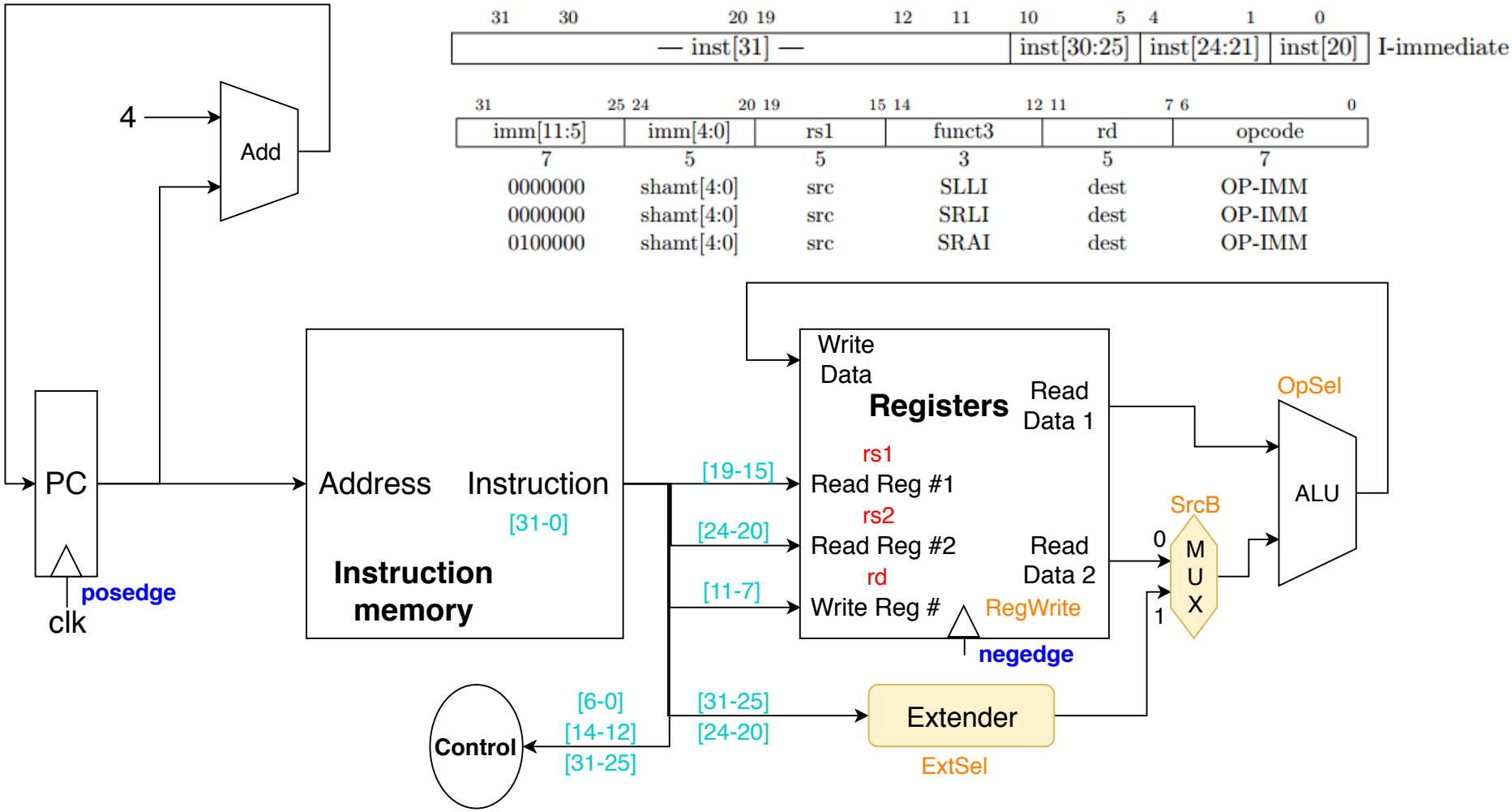


Register-Immediate Instructions

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

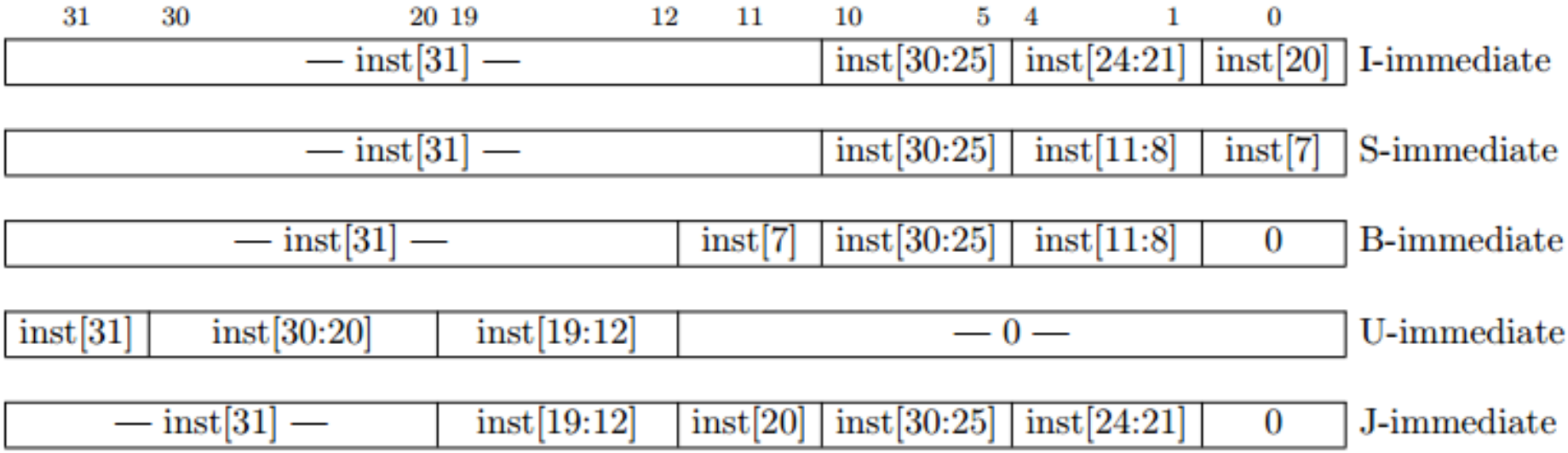
- Shift left (logic): `slli rd, rs1, imm`
- Shift right (logic): `srli rd, rs1, imm`
- Shift right (arithmetic): `srai rd, rs1, imm`
- Need not `slai`

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

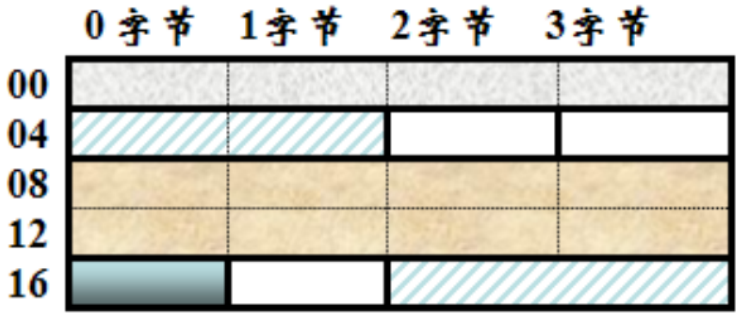
两种存储方式

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

0x12345678

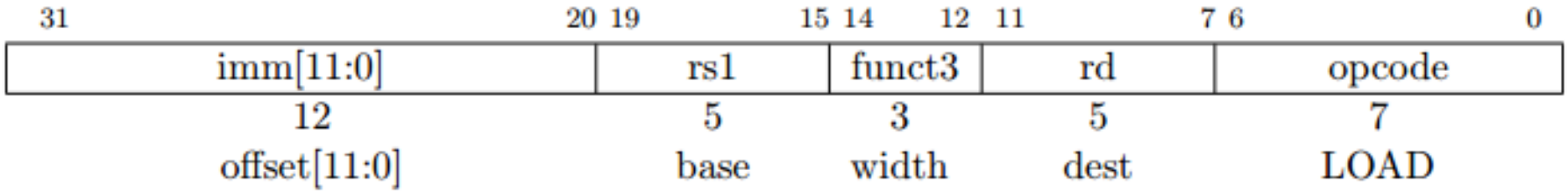
Address	00	04	08	0C
Little	78	56	34	12
Big	12	34	56	78

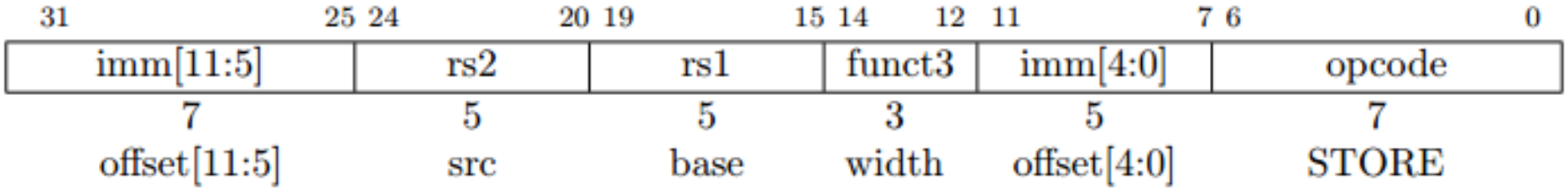
无边界对齐(misaligned)！提供细粒度访问（半字节）



Load & Store

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

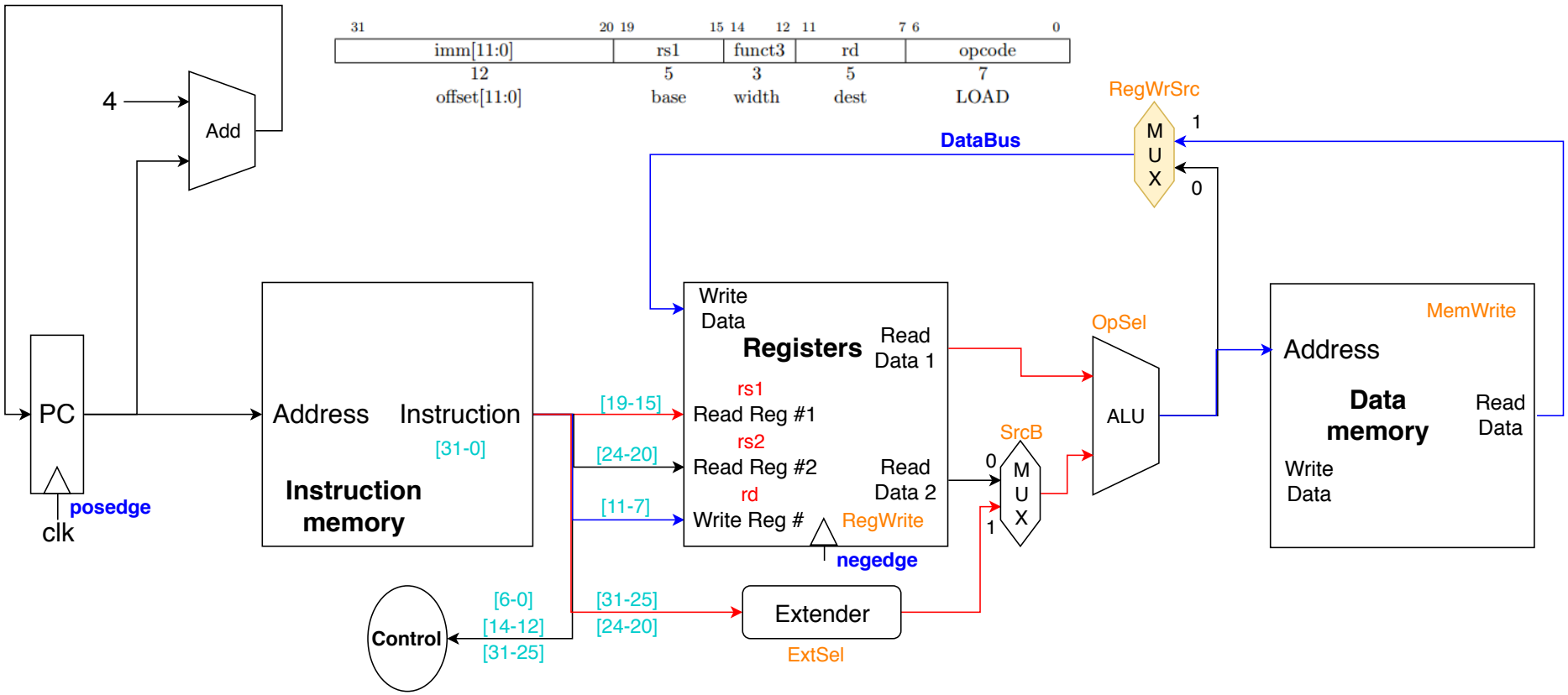




- `lw rd, base(imm)`
- `sw rs, base(imm)`

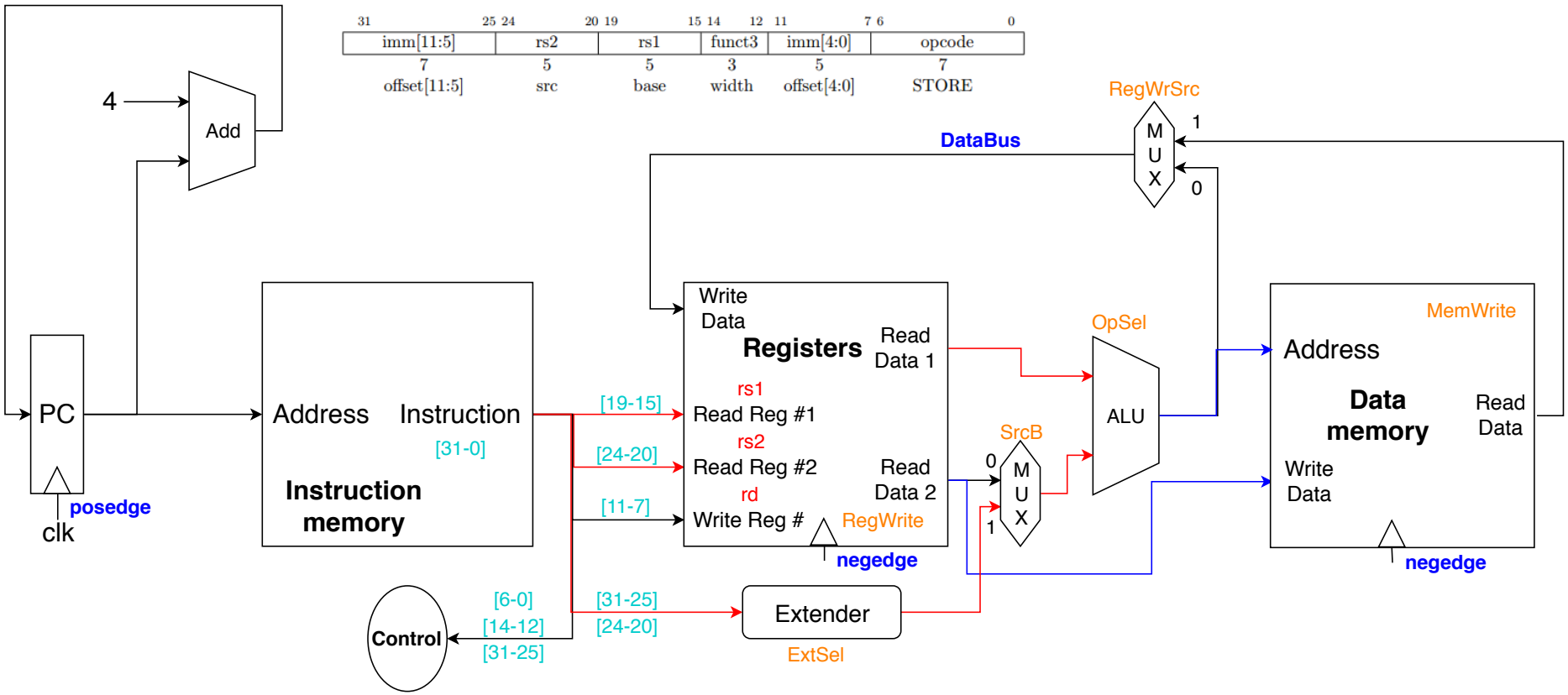
Load

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



Store

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

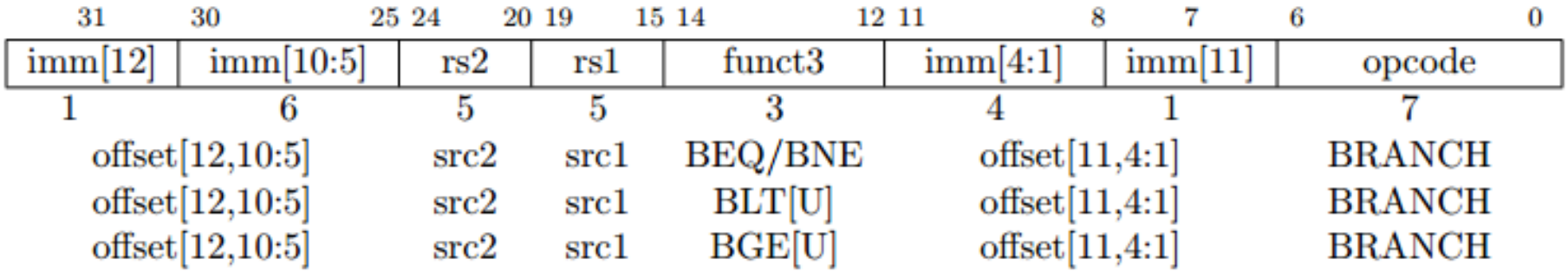


4.7

Branch

Branch

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

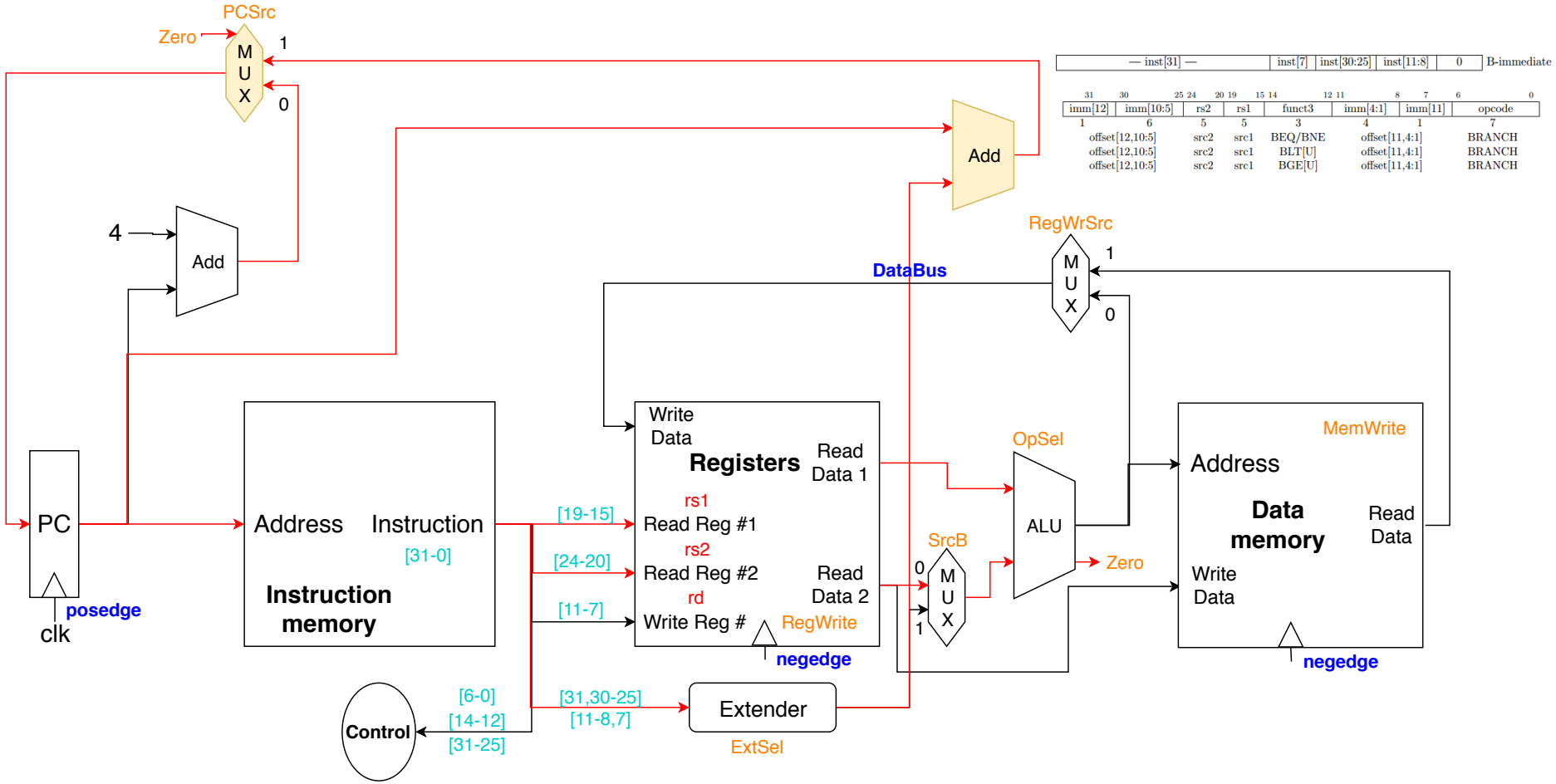


31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]		BRANCH

- `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

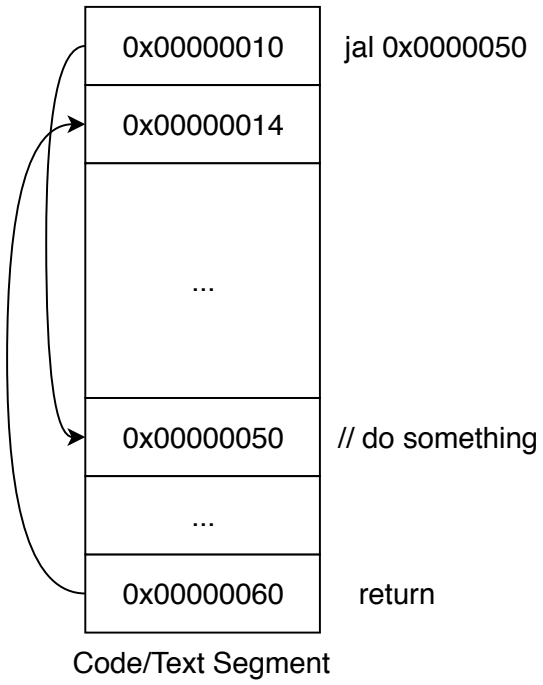
4.8

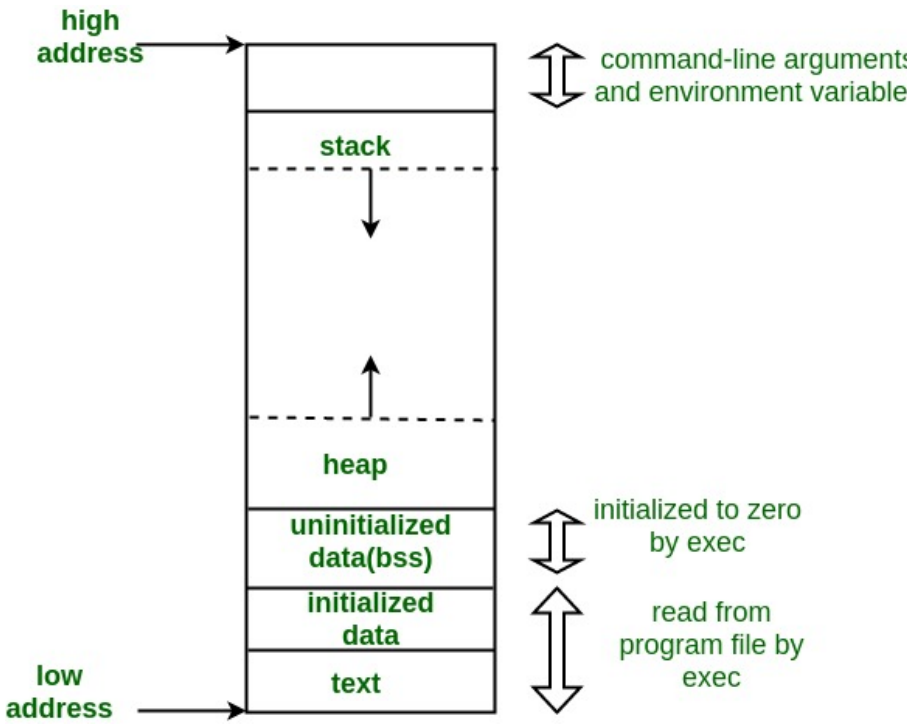
Jump

Function Call

堆栈、保护现场、恢复现场

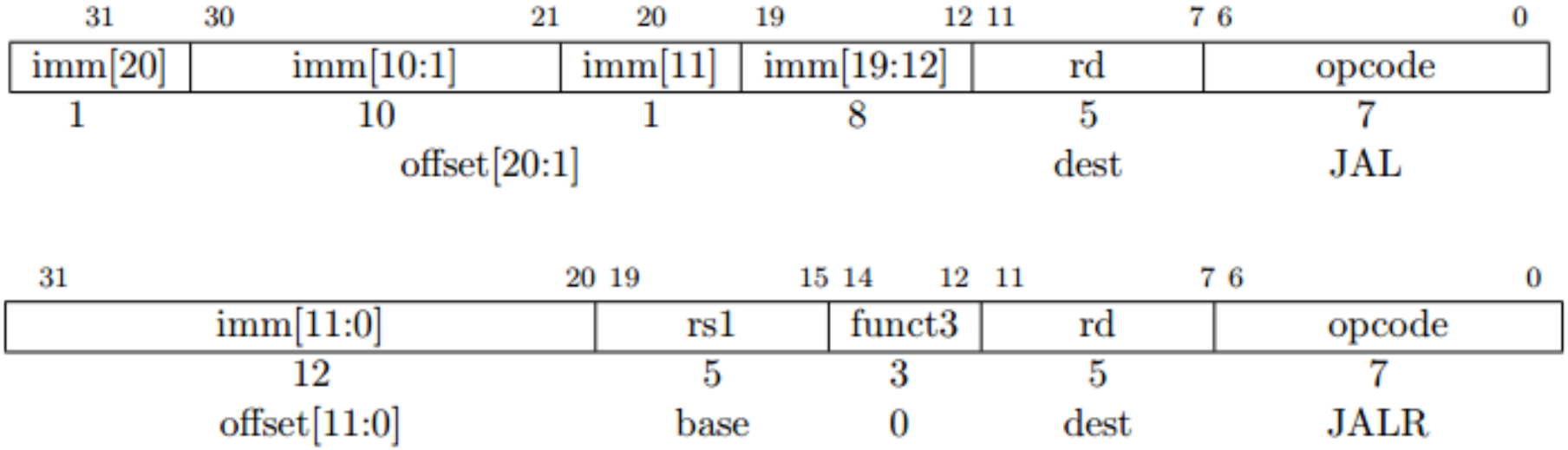
```
void foo (void)
{
    // do something
}

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





Jump

Address range ±1MiB

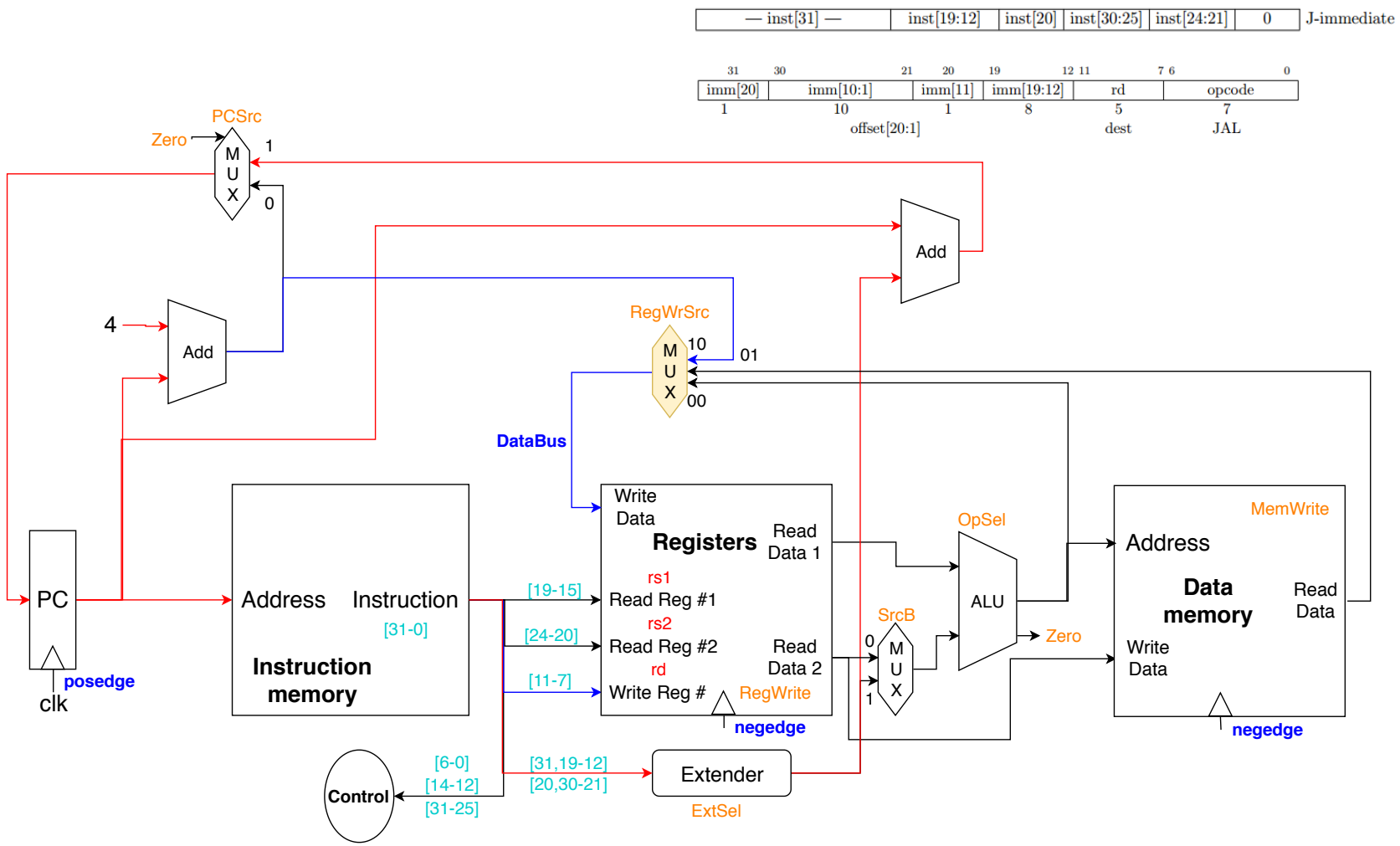


- `jal rd, offset`
- `jalr rd, offset(rs1)`

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

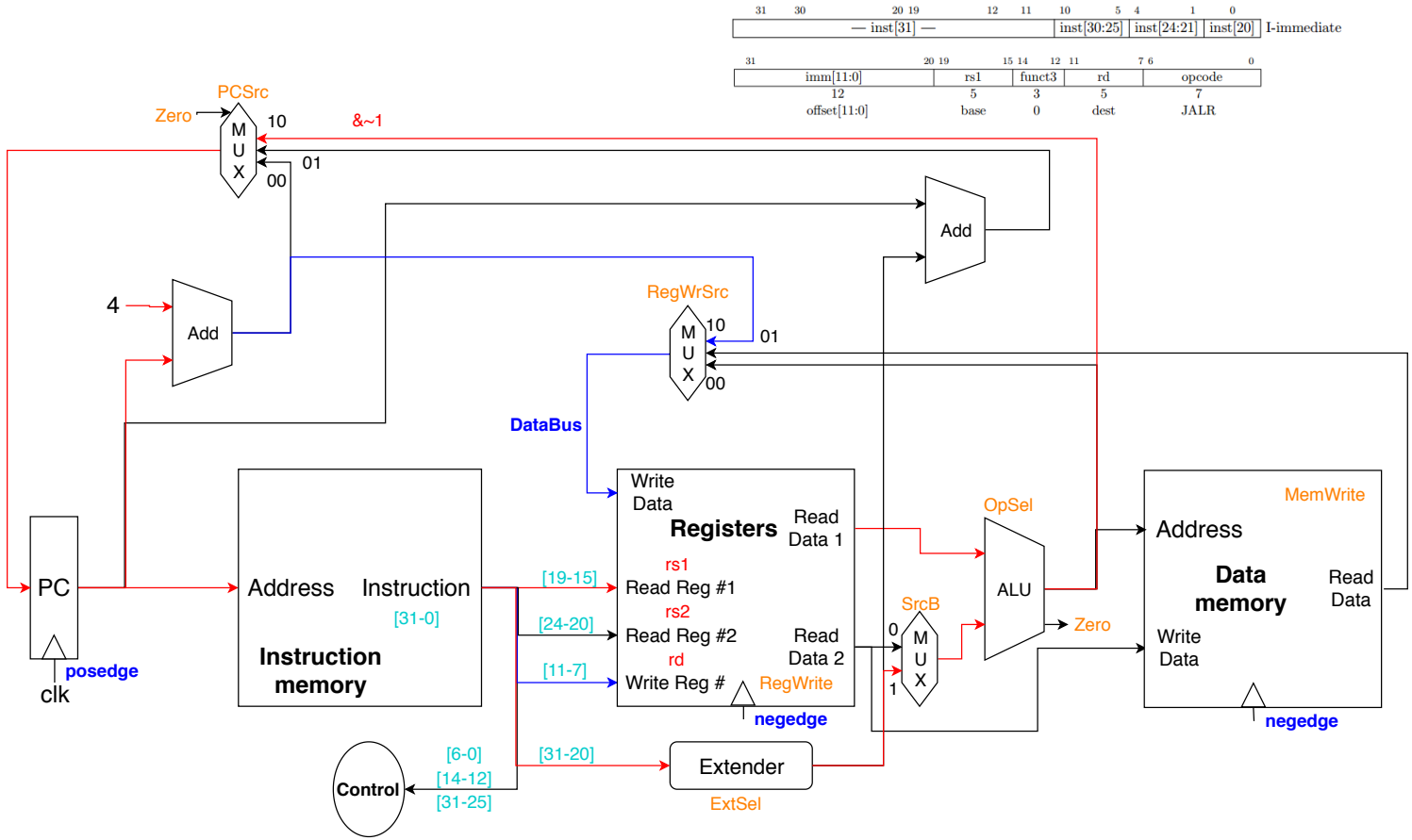
Jal

```
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



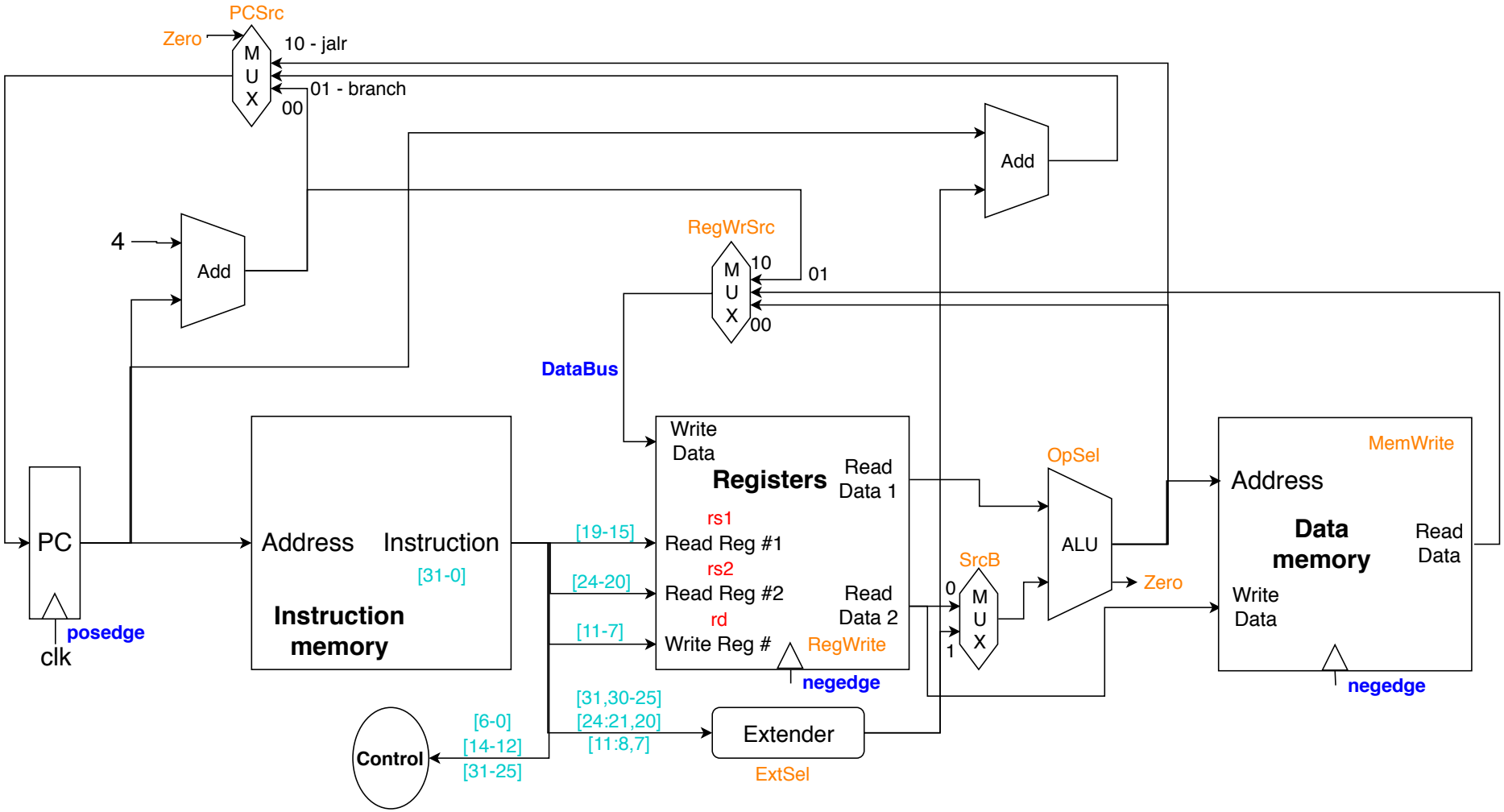
Question again: Where to implement `&~1`?

4.9

Single Cycle CPU Summary

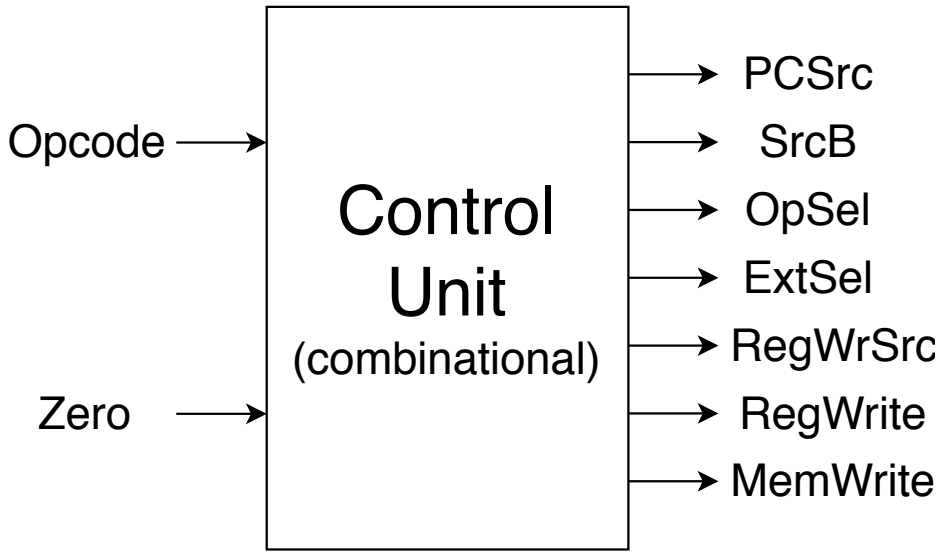
Harvard Architecture (Aiken and Mark)

Separate program and data memory



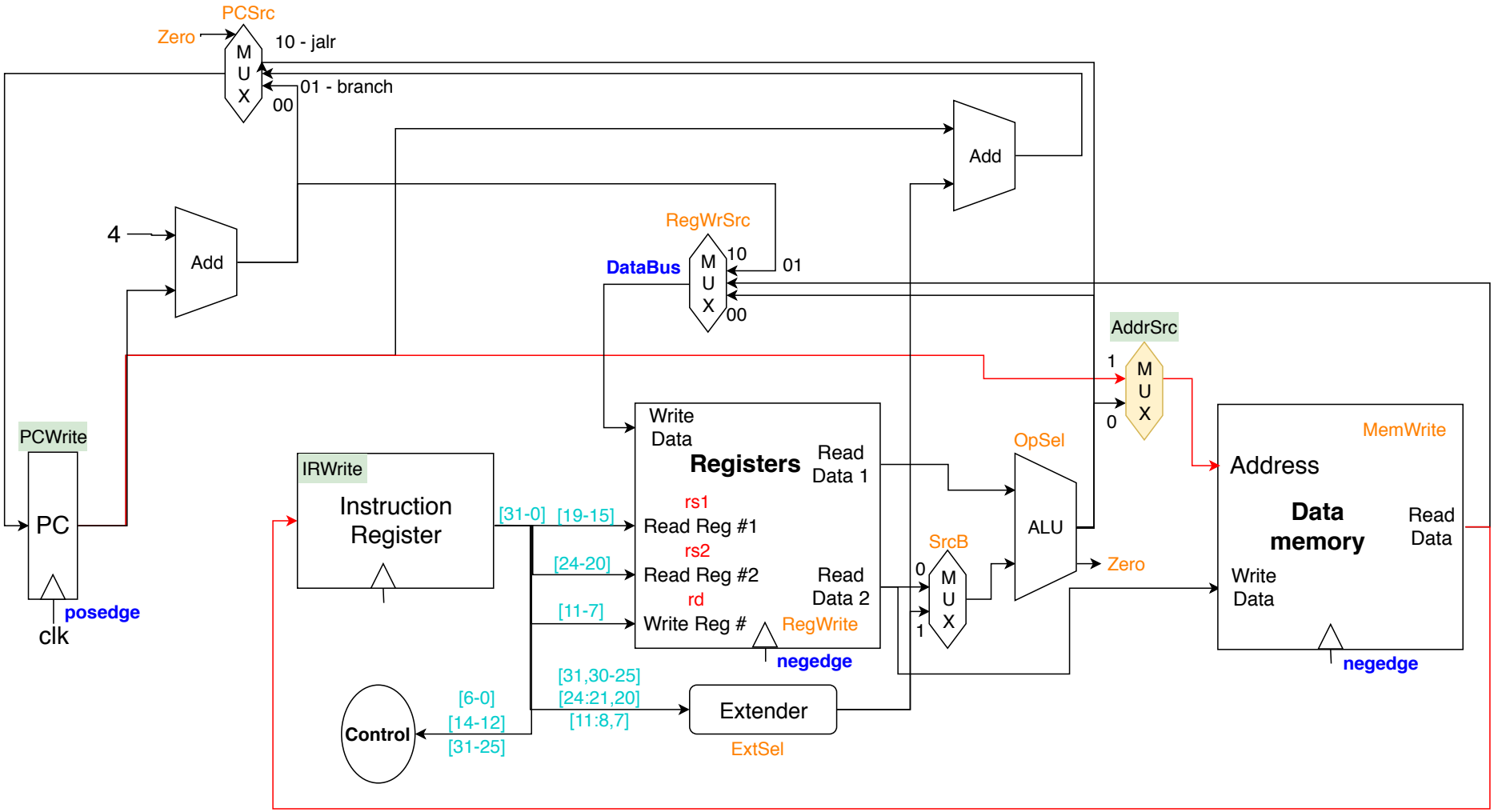
Control Signals

Hardwired control is pure combinational logic



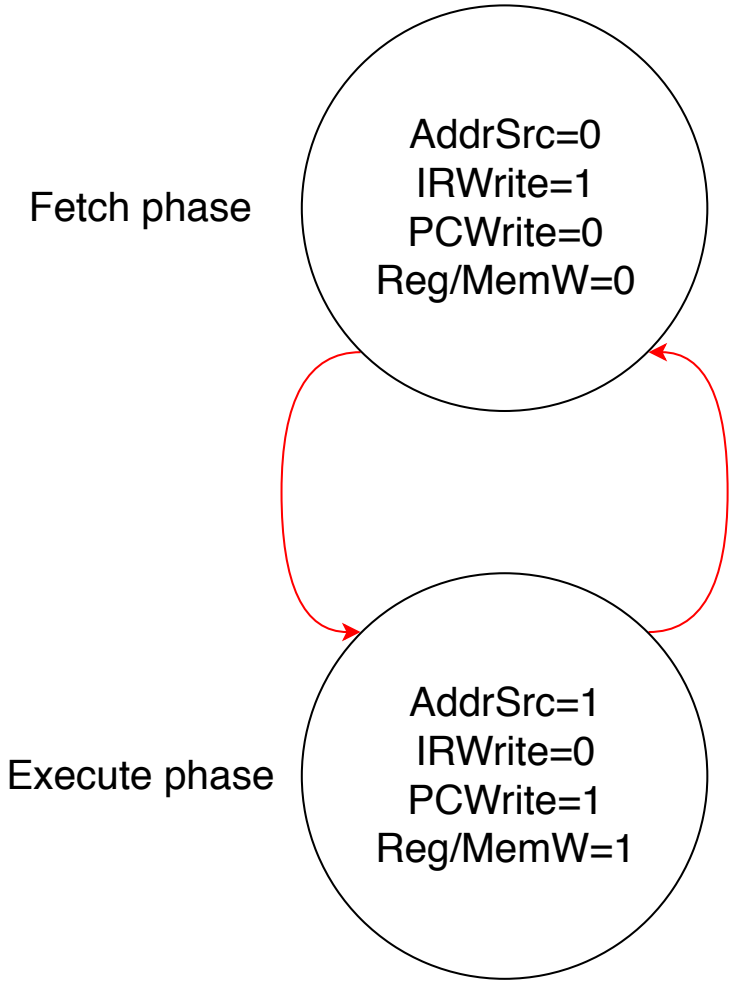
Princeton Architecture (von Neumann)

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

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

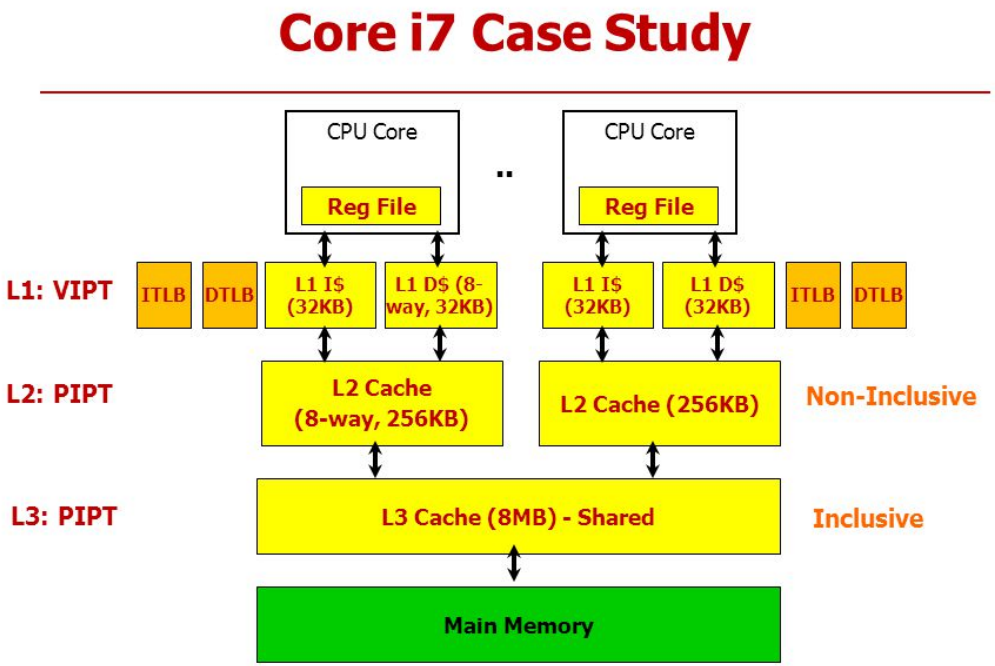
$$CPI_{Princeton} = 2$$
$$CPI_{Harvard} = 1$$

No difference in performance!

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

Two Architectures Summary

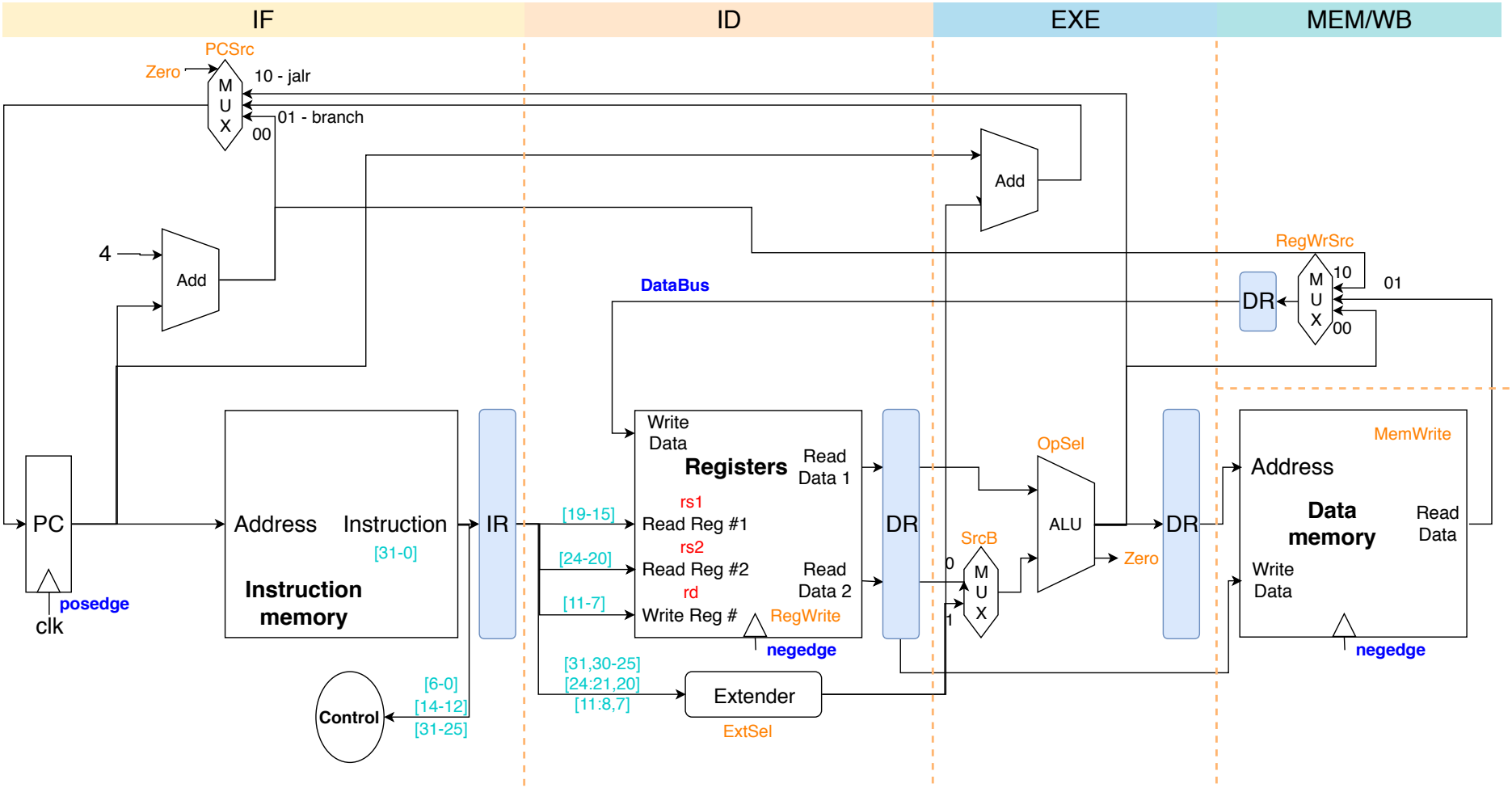
- Princeton: The same program and data memory, simple, two cycles, first place
- Harvard: Different program and data memory, complex, one cycle, ignored until 1970s



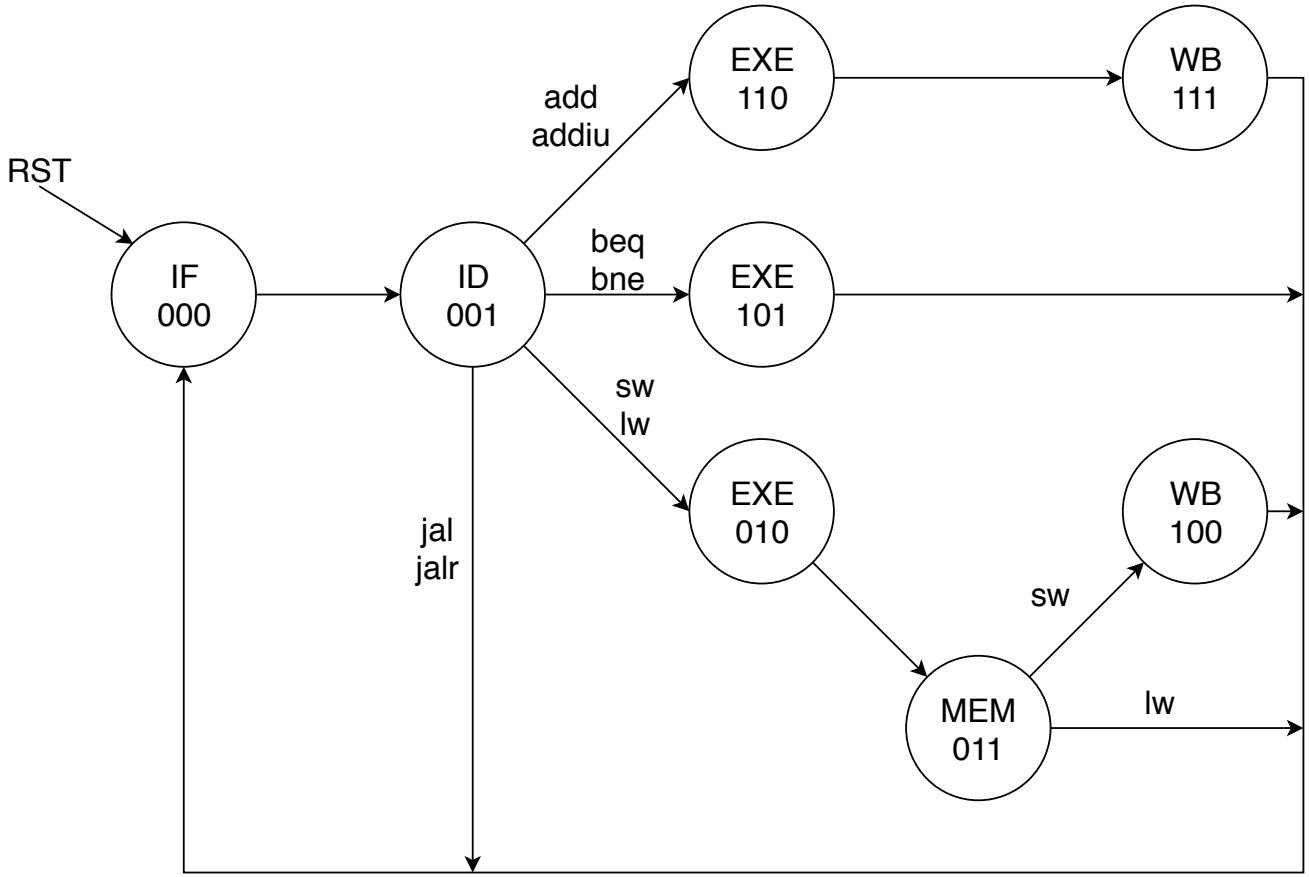
5

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**

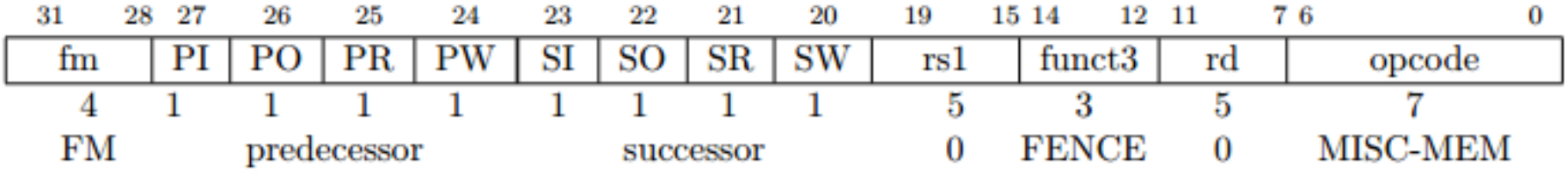
程序员（实施） → 高级语言（可移植规范） —编译器（实施）→ 汇编语言/指令集（不可移植规范） → 硬件（实施）

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)



31–28	27	26	25	24	23	22	21	20	19–15	14–12	11–7	6–0
fm	PI	PO	PR	PW	SI	SO	SR	SW	rs1	funct3	rd	opcode
4	1	1	1	1	1	1	1	1	5	3	5	7
FM	predecessor				successor				0	FENCE	0	MISC-MEM

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
- RV32S：特权指令

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