北京大学
PEKING UNIVERSITY

人工智能的硬件基石
从物理器件到计算架构

第五讲：复杂计算单元与指令集

主讲：陶耀宇

2026年春季

注意事项

北京大学 PEKING UNIVERSITY

- 课程作业情况

- **作业将在3月底-4月中旬、4月中旬-5月初、5月中旬-6月初**

  第一次作业时间：3.30-4.20（3周），今晚上传

- **第1次lab时间：4月10日-5月10日**
- **第2次lab时间：5月10日-6月15日**
- **助教安排硬件Verilog/SystemVerilog编写及设计、验证全流程入门介绍，有兴趣的同学请积极参与！**

注意事项

北京大学 PEKING UNIVERSITY

CLAB配置情况

- 目前CLAB平台账号已配置好，请各位同学保证能够**正常登录CLAB平台**
- 配置教程请各位同学**尽快完成CLAB配置**并且能够能录远程桌面，以免耽误后续LAB的正常进行
- **第三次verilog助教课**是否需要?
- CLAB配置问题请联系助教**李中源**同学

注意事项

北京大学 PEKING UNIVERSITY

- **课程作业情况**

- **第1次lab时间：4月10-5月10**
  - **2个基础任务（50%+50%）+ 1个Bonus任务（可2选1，50%）**
  - 基础任务提供完整环境（Verilog+TB+SW Reference，10个测试激励）
    - **基础任务1：面向卷积加速的1D Winograd电路**
    - **基础任务2：面向三角函数加速的Cordic电路**
  - Bonus任务（利用基础任务代码，自行编写Verilog+TB+SW Reference和测试激励）
    - 将1D Winograd扩展至2D Winograd
    - 将Cordic扩展至支持离散傅里叶变化DFT
  - **请撰写<=3页的实验报告**，简单解释代码结构与设计思路

目 录

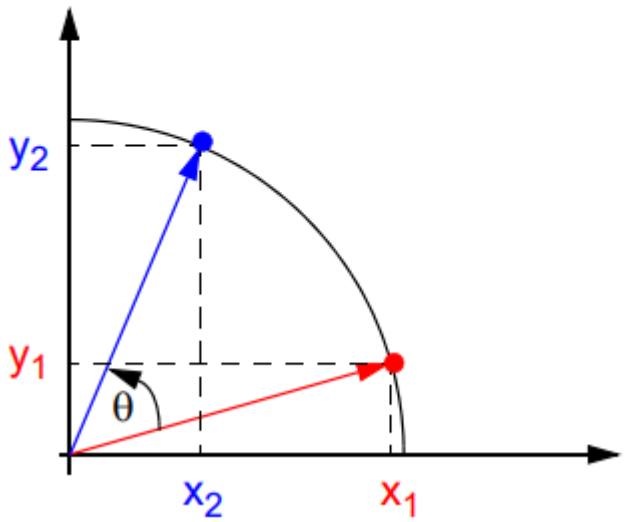
CONTENTS

复杂模块电路设计2 - CORDIC模块设计

- CORDIC可以用来实现多种复杂非线性函数

$$x_2 = x_1\cos\theta - y_1\sin\theta$$
$$y_2 = x_1\sin\theta + y_1\cos\theta$$

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}$$



$$x_2 = x_1\cos\theta - y_1\sin\theta = \cos\theta(x_1 - y_1\tan\theta)$$
$$y_2 = x_1\sin\theta + y_1\cos\theta = \cos\theta(y_1 + x_1\tan\theta)$$

$$\hat{x}_2 = \cancel{\cos\theta}(x_1 - y_1\tan\theta) = x_1 - y_1\tan\theta$$
$$\hat{y}_2 = \cancel{\cos\theta}(y_1 + x_1\tan\theta) = y_1 + x_1\tan\theta$$

当θ足够小接近于0时，先忽略cosθ（后面会scale回来）

伪旋转（pseudo rotations）

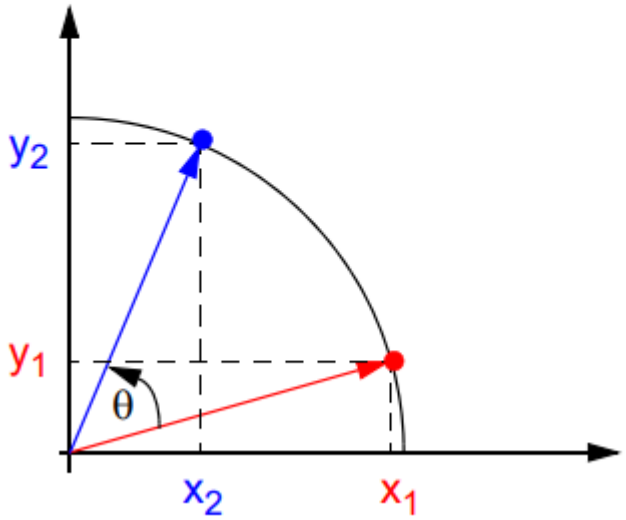
复杂模块电路设计2 - CORDIC模块设计

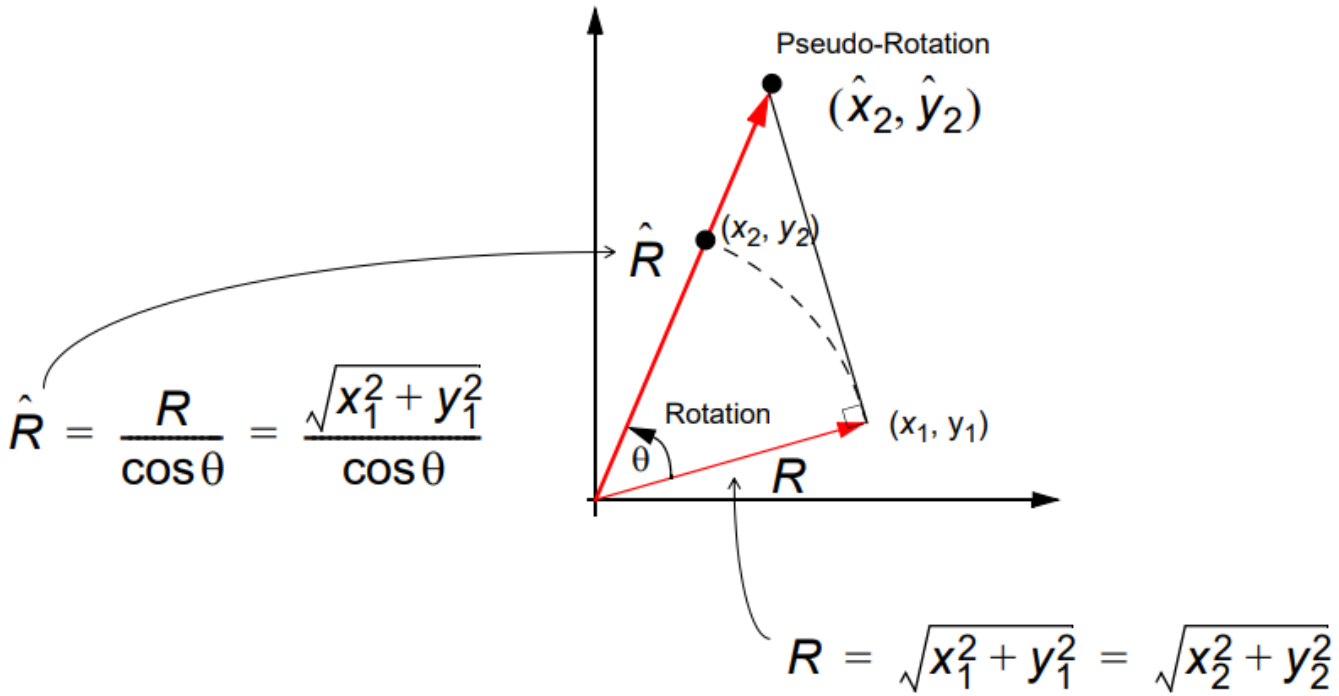
- CORDIC可以用来实现多种复杂非线性函数

$$x_2 = x_1\cos\theta - y_1\sin\theta$$
$$y_2 = x_1\sin\theta + y_1\cos\theta$$

伪旋转（pseudo rotations）

$$\hat{x}_2 = \cancel{\cos\theta}(x_1 - y_1\tan\theta) = x_1 - y_1\tan\theta$$
$$\hat{y}_2 = \cancel{\cos\theta}(y_1 + x_1\tan\theta) = y_1 + x_1\tan\theta$$





复杂模块电路设计2 - CORDIC模块设计

- CORDIC可以用来实现多种复杂非线性函数

伪旋转（pseudo rotations）

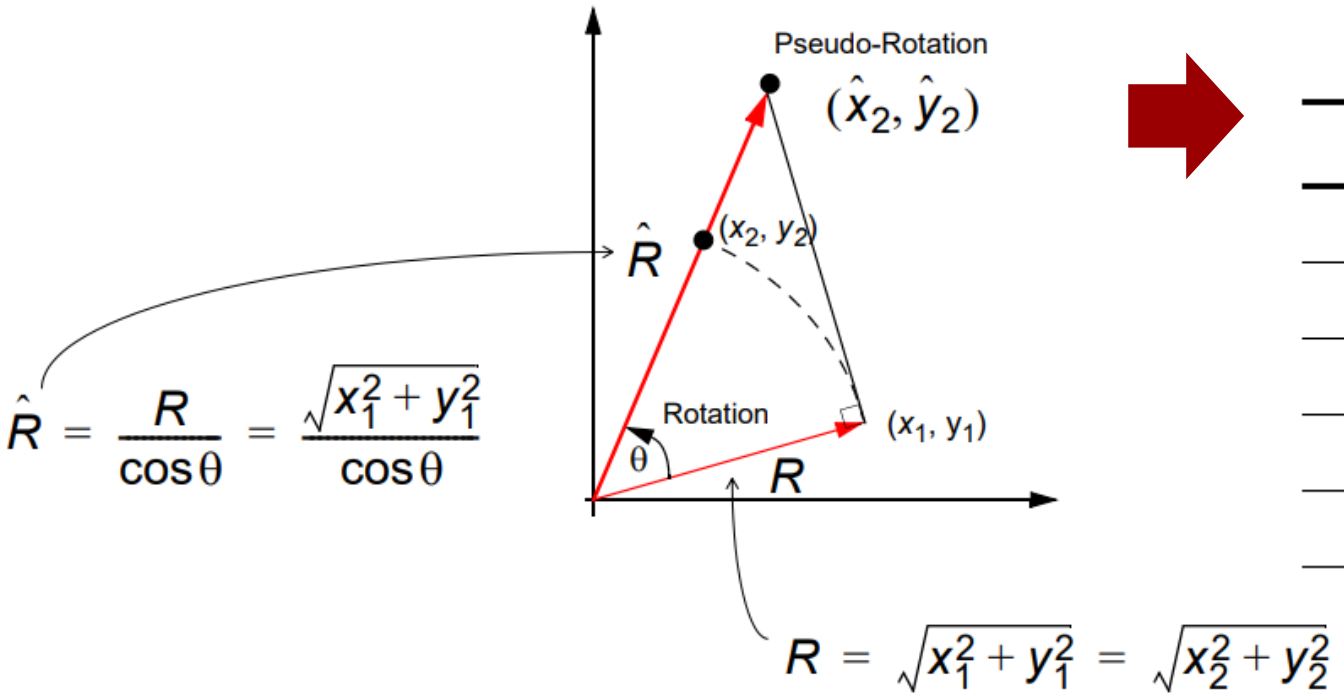
$$\hat{x}_2 = \cos\theta(x_1 - y_1\tan\theta) = x_1 - y_1\tan\theta$$

$$\hat{y}_2 = \cos\theta(y_1 + x_1\tan\theta) = y_1 + x_1\tan\theta$$

选择旋转角度 $\tan\theta^i = 2^{-i}$

$$\hat{x}_2 = x_1 - y_1\tan\theta = x_1 - y_1 2^{-i}$$

$$\hat{y}_2 = y_1 + x_1\tan\theta = y_1 + x_1 2^{-i}$$



i	θ^i (Degrees)	$\tan\theta^i = 2^{-i}$
0	45.0	1
1	26.555051177...	0.5
2	14.036243467...	0.25
3	7.125016348...	0.125
4	3.576334374...	0.0625

复杂模块电路设计2 - CORDIC模块设计

- CORDIC可以用来实现多种复杂非线性函数

1st iteration: rotate by 45°; 2nd iteration: rotate by 26.6°, 3rd iteration: rotate by 14°

i	tanθ	Angle, θ	cosθ
1	1	45.0000000000	0.707106781
2	0.5	26.5650511771	0.894427191
3	0.25	14.0362434679	0.9701425
4	0.125	7.1250163489	0.992277877
5	0.0625	3.5763343750	0.998052578
6	0.03125	1.7899106082	0.999512076
7	0.015625	0.8951737102	0.999877952
8	0.0078125	0.4476141709	0.999969484
9	0.00390625	0.2238105004	0.999992371
10	0.001953125	0.1119056771	0.999998093
11	0.000976563	0.0559528919	0.999999523
12	0.000488281	0.0279764526	0.999999881
13	0.000244141	0.0139882271	0.99999997

cos 45 x cos 26.5 x cos 14.03 x cos 7.125 ... x cos 0.0139 = 0.607252941

13次伪旋转后，需要乘以

1/0.607252941 =

1.6467602

复杂模块电路设计2 - CORDIC模块设计

- CORDIC可以用来实现多种复杂非线性函数

$$\hat{x}_2 = x_1 - y_1 \tan\theta = x_1 - y_1 2^{-i}$$

$$\hat{y}_2 = y_1 + x_1 \tan\theta = y_1 + x_1 2^{-i}$$

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i} y^{(i)})$$

$$y^{(i+1)} = y^{(i)} + d_i(2^{-i} x^{(i)})$$

$$z^{(i+1)} = z^{(i)} - d_i \theta^{(i)} \quad \text{(Angle Accumulator)}$$

where d_i = +/- 1

The symbol d_i is a decision operator and is used to decide which direction to rotate.

2 shifts

1 table lookup ($\theta^{(i)}$ values)

3 additions

复杂模块电路设计2 - CORDIC模块设计

北京大学 PEKING UNIVERSITY

- CORDIC可以用来实现多种复杂非线性函数

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i}y^{(i)})$$

$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$

$$z^{(i+1)} = z^{(i)} - d_i\theta^{(i)}$$

where d_i = +/- 1

2 shifts

1 table lookup ($\theta^{(i)}$ values)

3 additions

Scaling Factor

$$K_n = \prod_n 1/(\cos\theta^{(i)}) = \prod_n (\sqrt{1+2^{(-2i)}})$$

$$K_n = \prod_n 1/(\cos\theta^{(i)}) = \prod_n \left(\sqrt{1+\tan^2\theta^{(i)}}\right) = \prod_n \left(\sqrt{1+2^{(-2i)}}\right)$$

$$K_n \to 1.6476 \text{ as } n \to \infty$$

$$1/K_n \to 0.6073 \text{ as } n \to \infty$$

n = number of iterations

复杂模块电路设计2 - CORDIC模块设计

北京大学 PEKING UNIVERSITY

- CORDIC可以用来实现多种复杂非线性函数

$$x^{(i+1)} = x^{(i)} - d_i(2^{-i}y^{(i)})$$

$$y^{(i+1)} = y^{(i)} + d_i(2^{-i}x^{(i)})$$

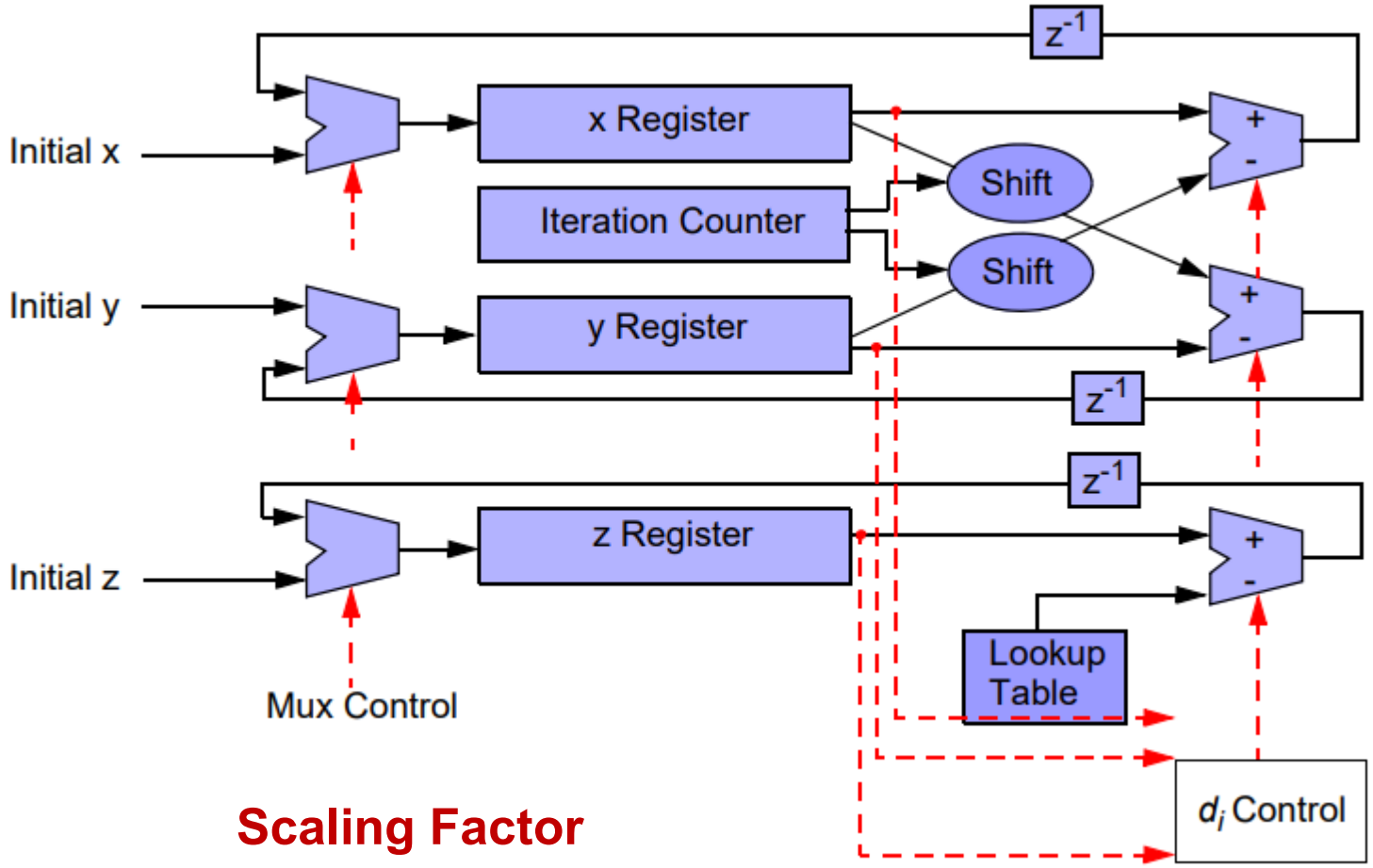
$$z^{(i+1)} = z^{(i)} - d_i\theta^{(i)}$$

where d_i = +/- 1

2 shifts

1 table lookup ($\theta^{(i)}$ values)

3 additions



Scaling Factor

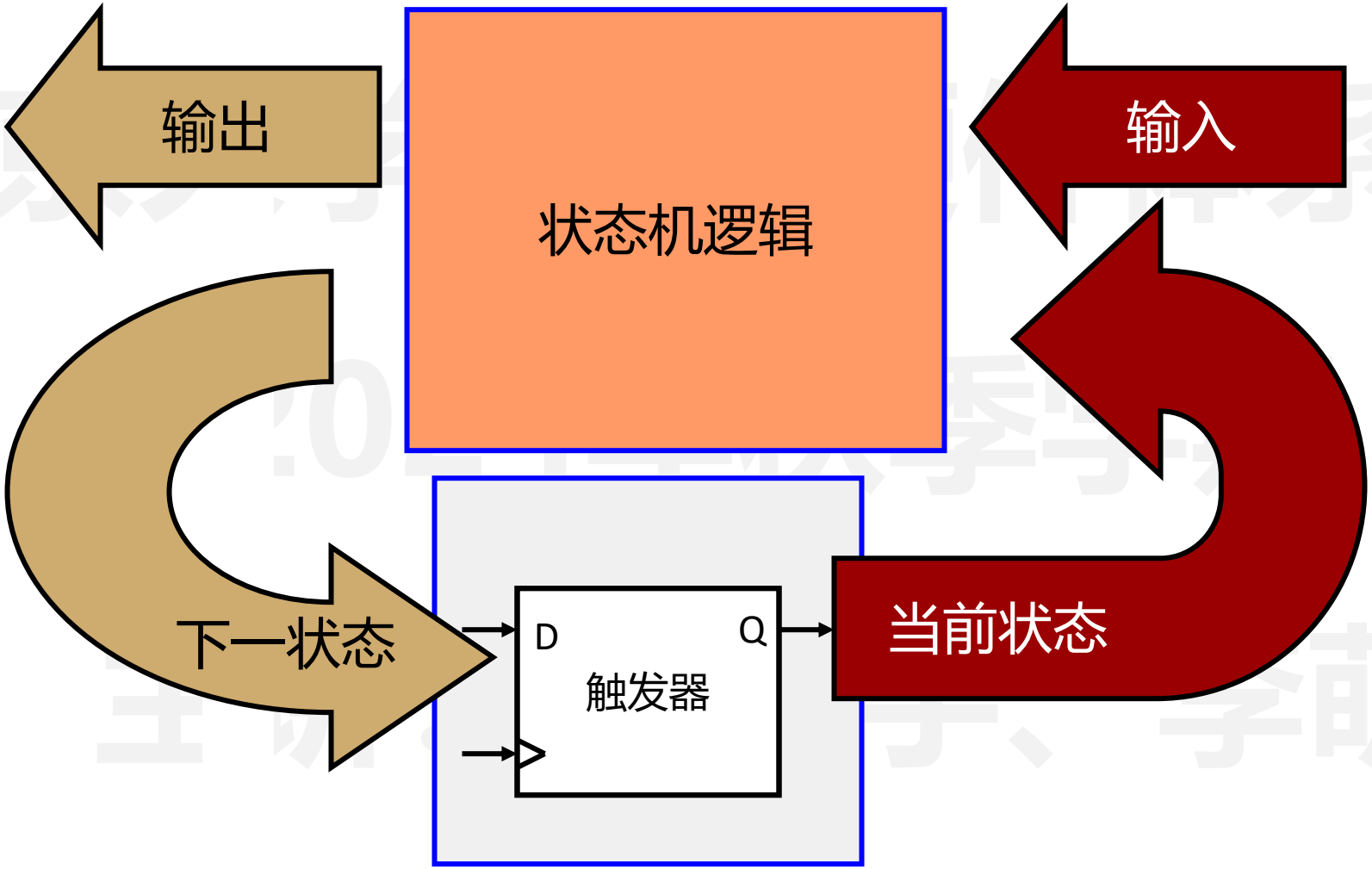
$$K_n = \prod_n 1/(\cos\theta^{(i)}) = \prod_n (\sqrt{1+2^{(-2i)}})$$

目 录

CONTENTS

为什么需要状态机

- 控制电路的基石



为什么需要状态机

- 控制电路的基石

状态机实例1 – 控制一个红绿灯

- 仅考虑红灯和绿灯，灯转换的速度不快于每次30s（0.033 Hz 时钟）
- 2个输出
  - NSlight：1=南北向为绿灯；0=南北向红灯
  - EWlight：1=东西向为绿灯；0=东西向为红灯
- 2个输入
  - Nscar：1=南北向有车等；0=南北向无车等
  - Ewcar：1=东西向有车等；0=南北向无车等
- 规则
  - 交通灯切换到另一个方向当且仅当另一方向有车等
  - 否则，保持当前交通灯不变

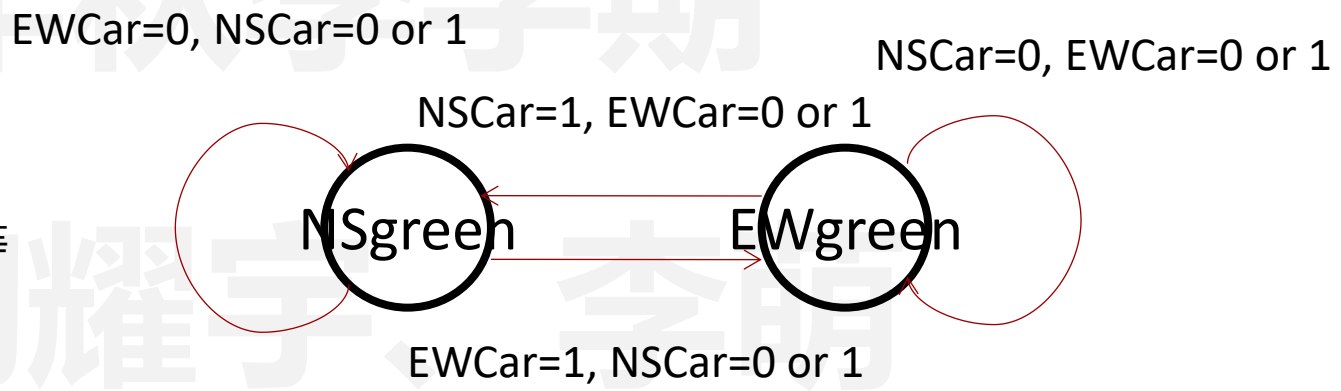
为什么需要状态机

- 控制电路的基石

状态机实例1 – 控制一个红绿灯

- 2个输出
  - NSlight：1=南北向为绿灯；0=南北向红灯
  - EWlight：1=东西向为绿灯；0=东西向为红灯
- 2个输入
  - Nscar：1=南北向有车等；0=南北向无车等
  - Ewcar：1=东西向有车等；0=南北向无车等
- 规则
  - 交通灯切换到另一个方向当且仅当另一方向有车等
  - 否则，保持当前交通灯不变

- 需要2个状态
  - Nsgreen、EWgreen



为什么需要状态机

- 控制电路的基石

状态机实例1 – 控制一个红绿灯

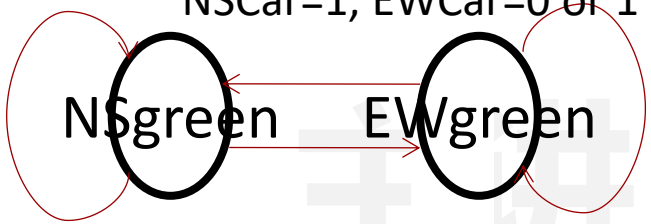
- 需要2个状态
  - Nsgreen、EWgreen

Current state	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

EWCar=0, NSCar=0 or 1

NSCar=0, EWCar=0 or 1

NSCar=1, EWCar=0 or 1



EWCar=1, NSCar=0 or 1

Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1

$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$

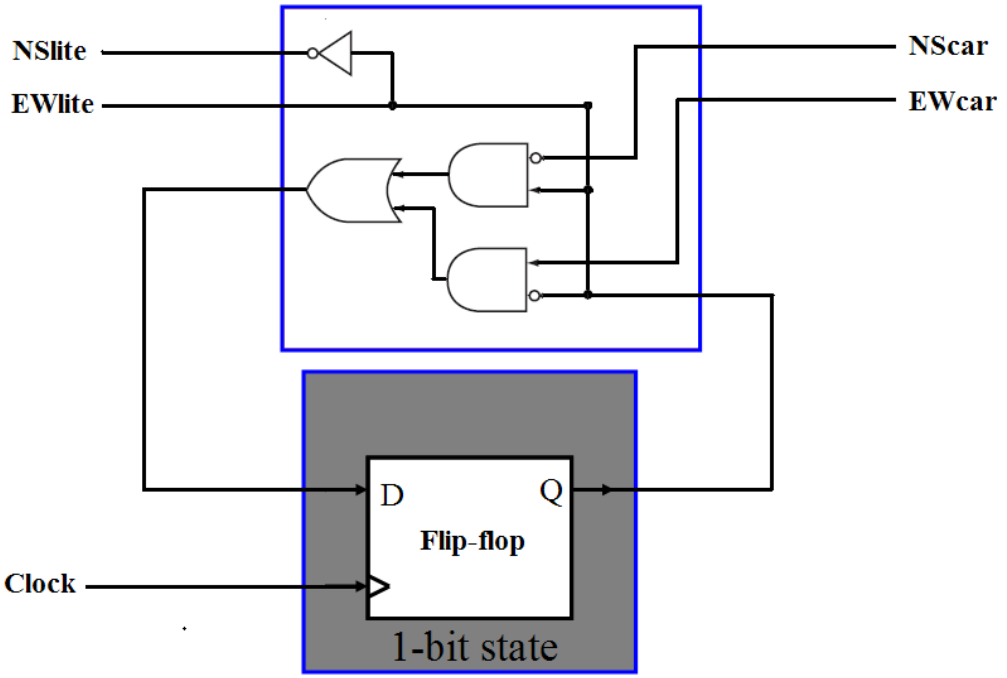
为什么需要状态机

- 控制电路的基石

状态机实例1 – 控制一个红绿灯

Current state	Inputs		Next state
	NScar	EWcar	
NSgreen	0	0	NSgreen
NSgreen	0	1	EWgreen
NSgreen	1	0	NSgreen
NSgreen	1	1	EWgreen
EWgreen	0	0	EWgreen
EWgreen	0	1	EWgreen
EWgreen	1	0	NSgreen
EWgreen	1	1	NSgreen

Current state	Outputs	
	NSlite	EWlite
NSgreen	1	0
EWgreen	0	1



$$\text{NextState} = (\overline{\text{CurrentState}} \cdot \text{EWcar}) + (\text{CurrentState} \cdot \overline{\text{NScar}})$$

$$\text{NSlite} = \overline{\text{CurrentState}}$$

$$\text{EWlite} = \text{CurrentState}$$

为什么需要状态机

- 控制电路的基石

Step 1 – 定义状态并画出状态转换图

Step 2 – 给每一个状态赋值并更新状态转换图

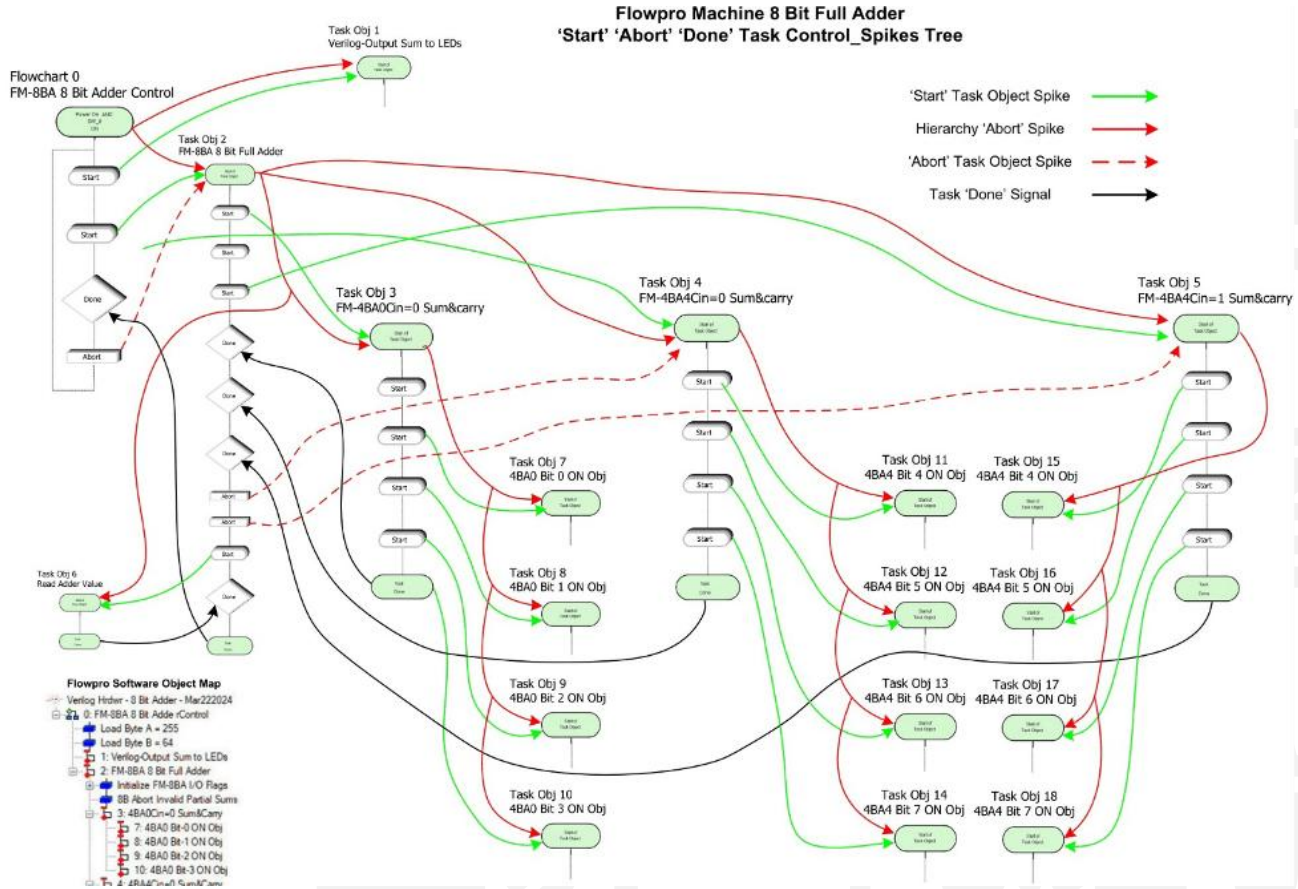
Step 3 – 根据状态转换图写出下一状态和输出的逻辑表达式

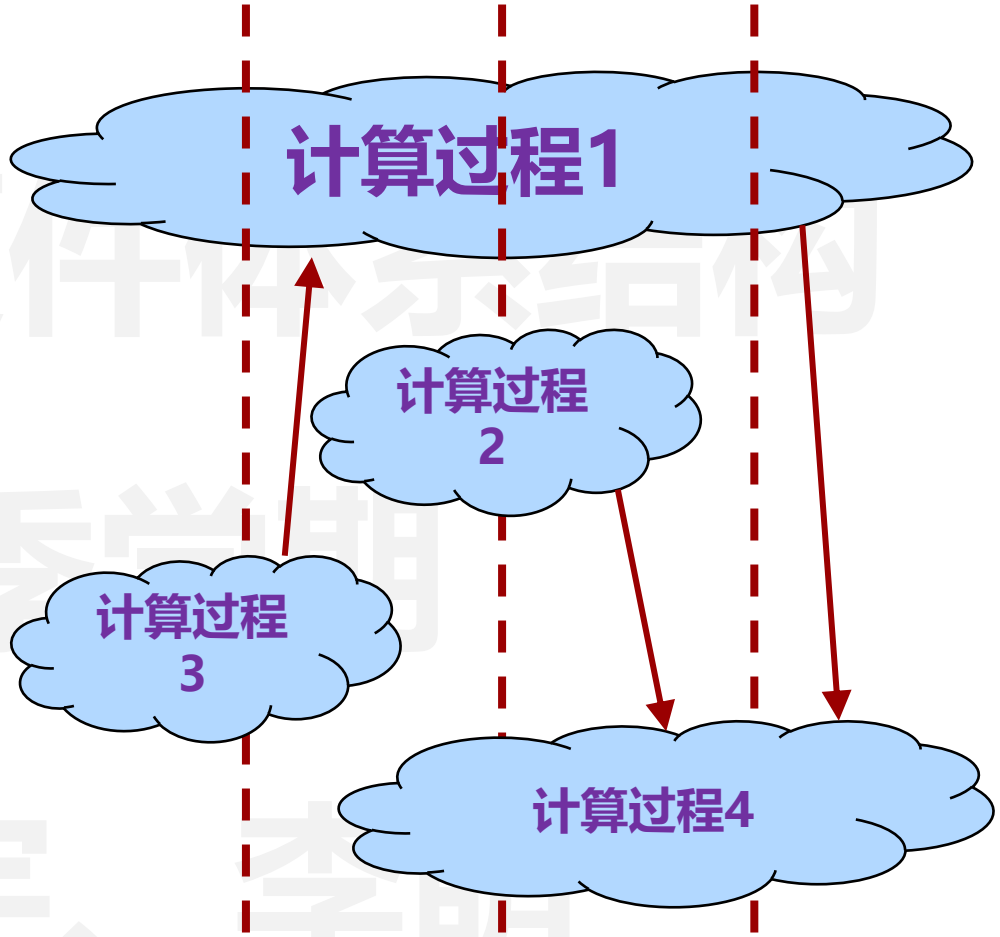
Step 4 – 画出实际电路图

状态将在每一个时钟上升沿更新

电路时序的基本概念

- 电路为什么需要一个时钟?



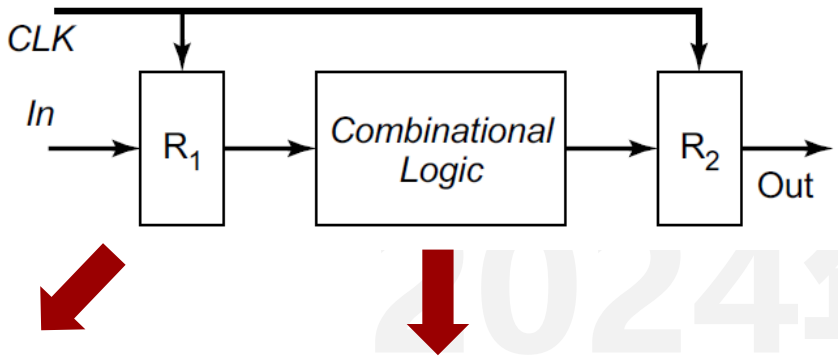


无时钟：非常难以控制每一个信号的有效时间

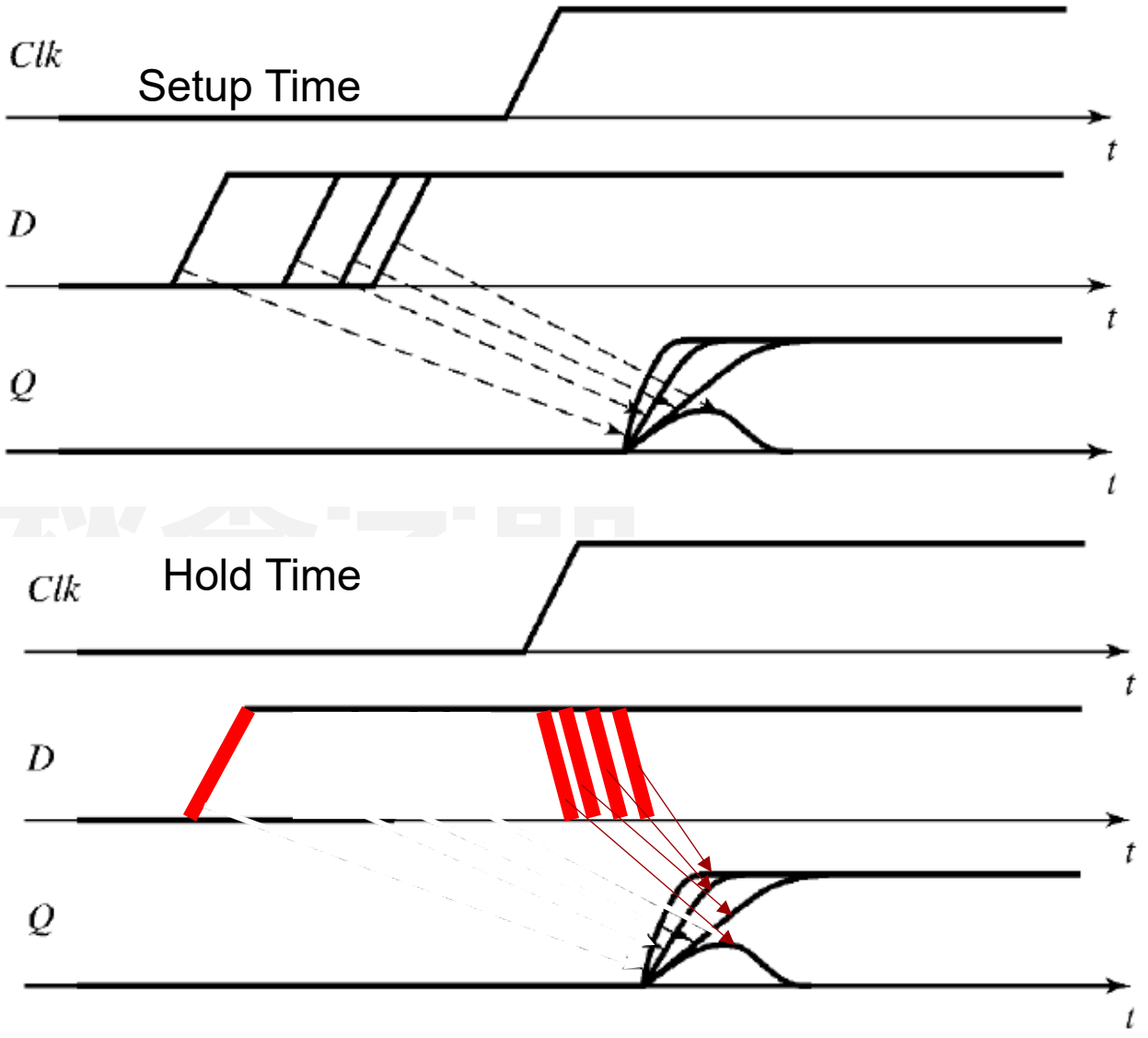
引入时钟：每隔一段计算将结果同步一次

电路时序的基本概念

- 同步时序（Synchronous Timing）

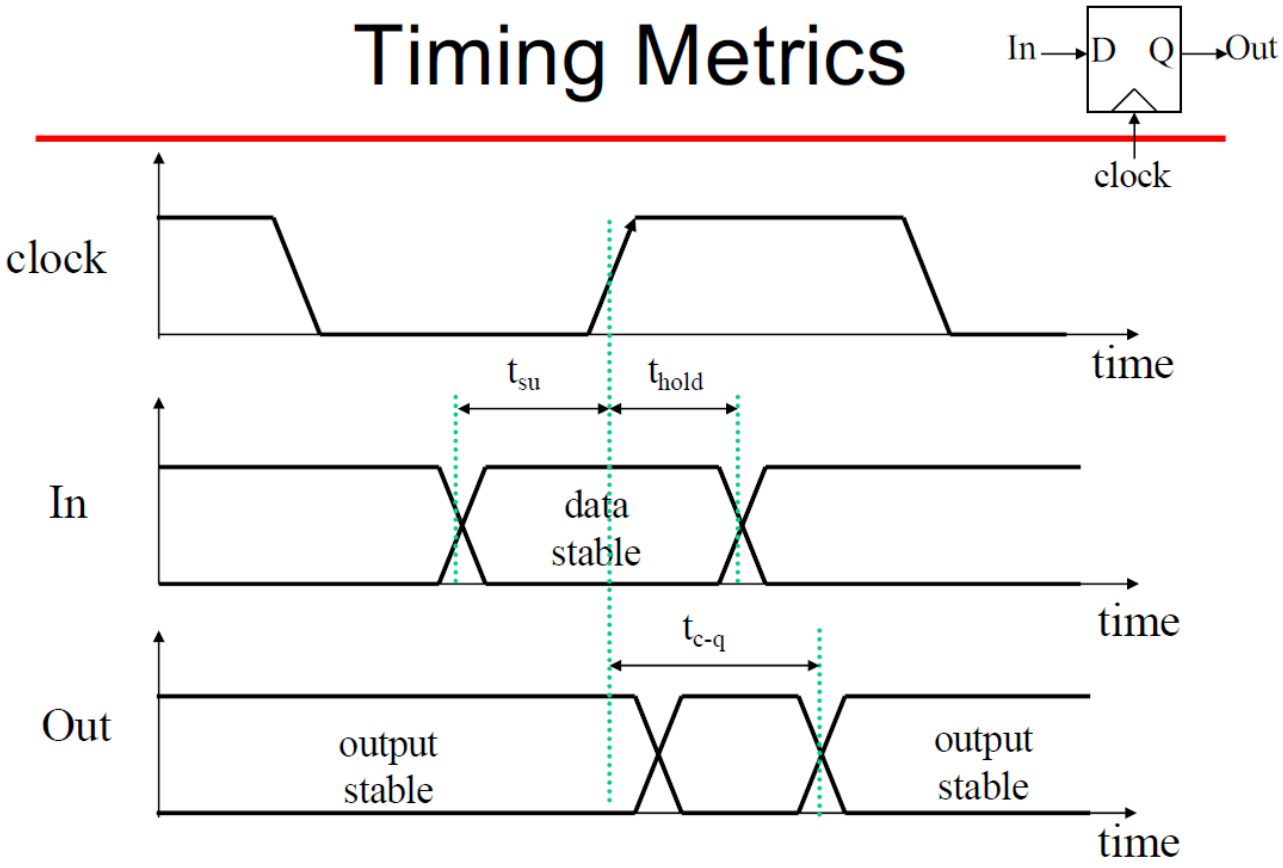


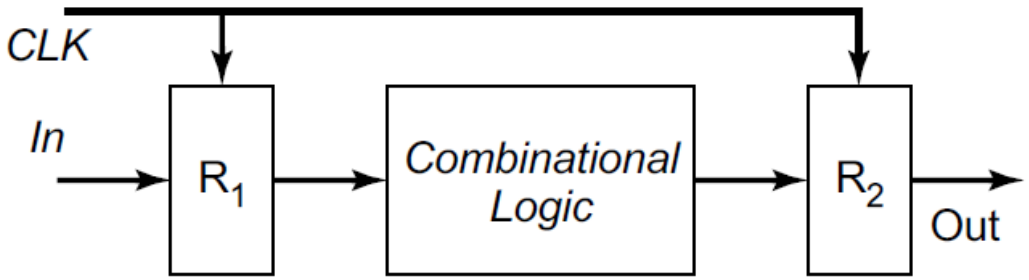
触发器　组合逻辑（各种逻辑门电路）
(Register)



电路时序的基本概念

- 同步时序（Synchronous Timing）





$$T_{c\text{-}q} + t_{plogic,min} \geq t_{hold}$$

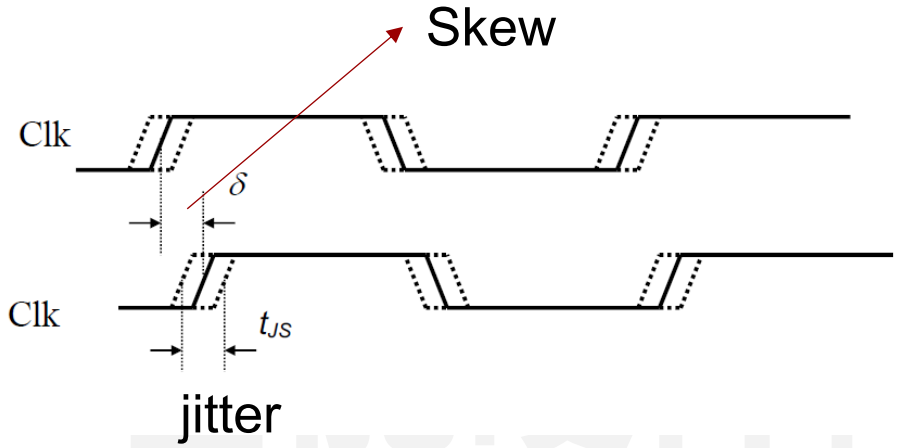
min

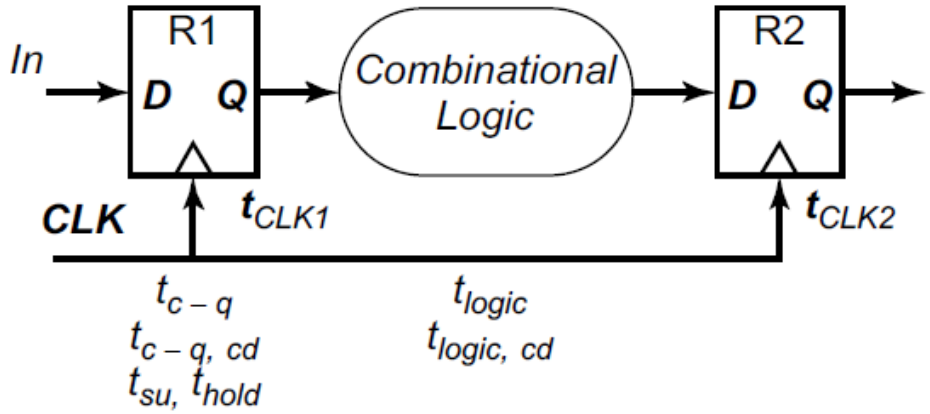
$$T \geq t_{c\text{-}q} + t_{plogic,max} + t_{su}$$

max

电路时序的基本概念

- 时钟的不稳定性

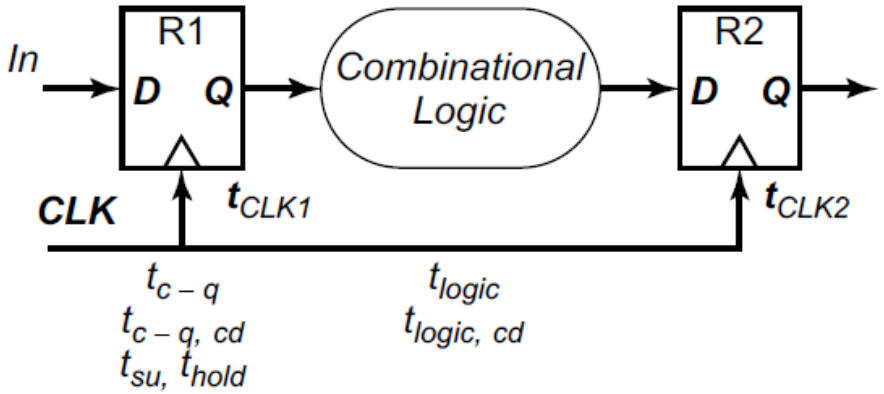




Minimum cycle time:

$T \geq t_{c\text{-}q} + t_{su} + t_{logic} - \delta$

最坏情况为接收边沿过早到达（negative δ）



Hold time constraint:

$t_{(c\text{-}q,\ cd)} + t_{(logic,\ cd)} > t_{hold} + \delta$

最坏情况为接收边沿过晚到达（正偏差）
数据和时钟之间的竞争
Cd：contamination delay（最快可能延迟）

目 录

CONTENTS

为什么需要指令集?

- 指令集可以看做链接软件和硬件的一个协议

ISA (instruction set architecture)

A well-defined hardware/software interface

一个定义完善的软/硬件接口

软件与硬件之间的“协议”

- 对硬件支持操作、模式和存储位置的功能定义
- 对如何调用获取硬件资源的精确描述

以下内容不通过ISA定义

- 具体如何实现操作
- 在不同场景下，不同操作速度的快慢
- 不同操作的功耗

为什么需要指令集?

北京大学 PEKING UNIVERSITY

- **指令集可以看做链接软件和硬件的一个协议**

- 编程者可见的变量
  - 编程计数器，通用计数器，存储器，控制寄存器
- 编程者可见的行为（状态转换）
  - 要做什么，什么时候做

Example “register-transfer-level” description of an instruction

```
if imem[pc]==“add rd, rs, rt”
then
        pc ⇐ pc+1
        gpr[rd]=gpr[rs]+grp[rt]
```

- A binary encoding

ISAs last 25+ years (because of SW cost)…

…be careful what goes in

指令集的分类

- **RSIC和CISC两种指令集**

- “Iron” law:
  - (instructions/program) * (cycles/instruction) * (seconds/cycle)
- CISC (Complex Instruction Set Computing) **例如X86等商用指令集**
  - 用复杂的指令改善了“instruction/program”
  - 对软件编程者友好，代码量小
- RISC (Reduced Instruction Set Computing) **例如MIPS/ARM/RISC-V**
  - 通过许多单周期指令来改善“cycles/instruction”
  - 增加了“instruction/program”，但代价不大
    - 编译器对此帮助很大
  - 有时会改善时钟周期长度
    - 精简指令允许了更激进的代码与硬件实现

指令集设计思路

北京大学 PEKING UNIVERSITY

- 兼顾软件可编程性、硬件可实现性和兼容性
- **Programmability**
  - 可以高效且容易的表达程序
- **Implementability**
  - 能够设计出高性能的硬件实现
    - 低功耗设计
    - 高可靠性设计
    - 低开销设计
- **Compatibility**
  - 在编程语言与程序更新迭代后，能够保持可编程性与硬件可实现性
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, Pentium-II, Pentium-III, Pentium4, …
  - MIPS、RISC-V、ARM…

软件编译过程

- 软件代码通过编译器和指令集，编译成硬件可直接运行的汇编代码

- Demo of assembler
  - $ g++ -Og -c -S file1.cpp
- Demo of hexdump
  - $ g++ -Og -c file1.cpp
  - $ hexdump -C file1.o | more
- Demo of objdump/disassembler
  - $ g++ -Og -c file1.cpp
  - $ objdump -d file1.o

```
void abs(int x, int* res)
{
  if(x < 0)
    *res = -x;
  else
    *res = x;
}
```

Original Code

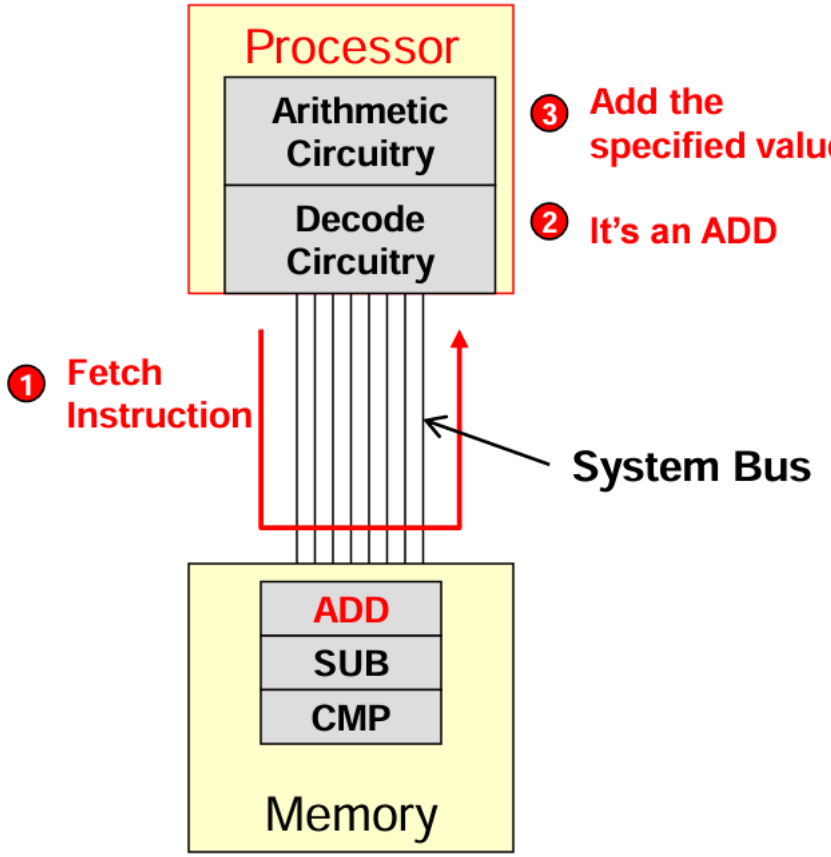
```
Disassembly of section .text:

0000000000000000 <_Z3absiPi>:
   0:   85 ff    test   %edi,%edi
   2:   79 05    jns    9 <_Z3absiPi+0x9>
   4:   f7 df    neg    %edi
   6:   89 3e    mov    %edi,(%rsi)
   8:   c3       retq
   9:   89 3e    mov    %edi,(%rsi)
   b:   c3       retq
```

Compiler Output
(Machine code & Assembly)
Notice how each instruction is turned into binary (shown in hex)

传统冯诺依曼架构的指令集

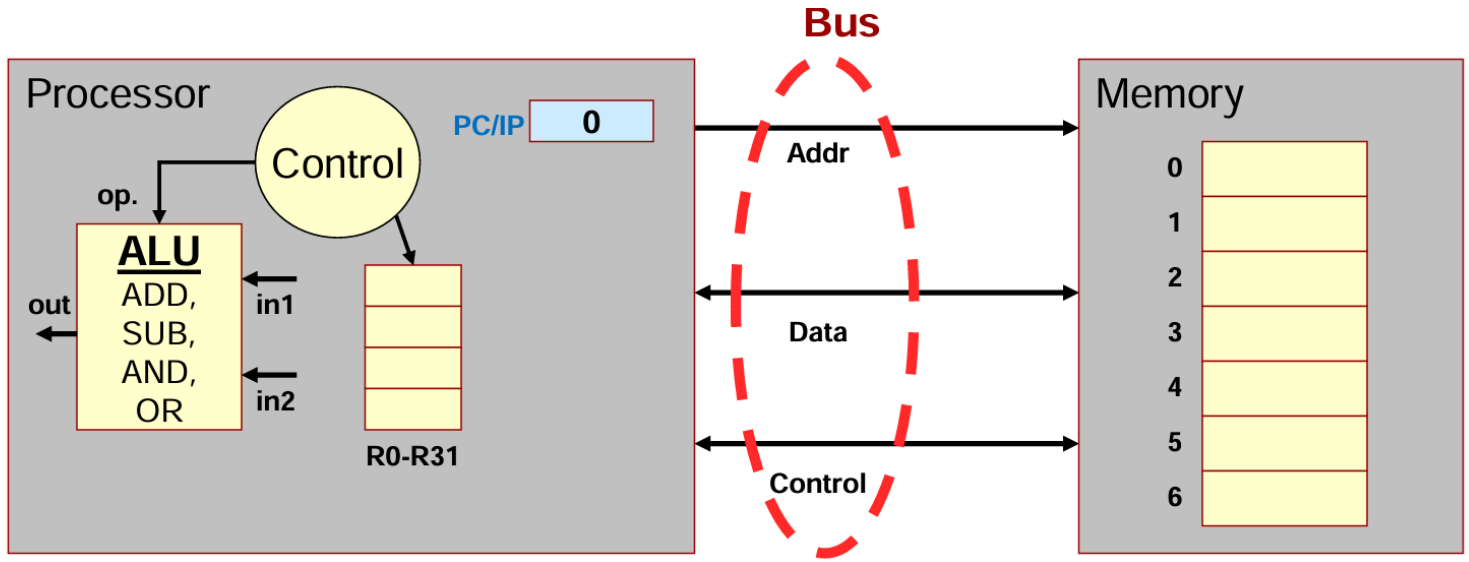
- 需要3类指令：读取、写回和运算

- 来回执行以下同样三种类型的指令
  - Fetch：从存储器中取出指令
  - Decode：解码这个指令
    - 此指令是ADD，SUB或是其他?
  - Execute：执行指令
    - 执行特定操作
- 每个指令运行的过程被称为指令周期



传统冯诺依曼架构的指令集

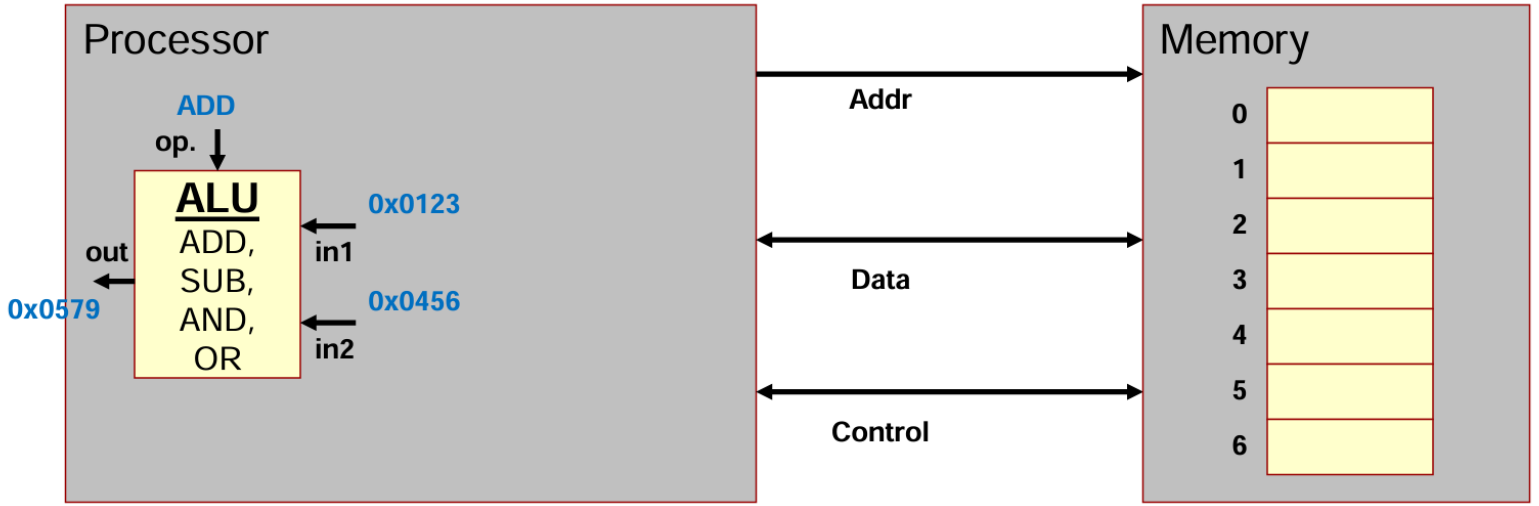
- 需要3类指令：读取、写回和运算
  - 处理器中3种主要的组成
    - ALU（算术逻辑单元）
    - 寄存器
    - 控制电路
  - 通过地址、数据和控制总线（bus）与存储器和I/O连接



传统存算分离的指令集架构 – 核心部件1：ALU

- ALU是指令集架构的核心部件，负责完成所有实际的计算功能

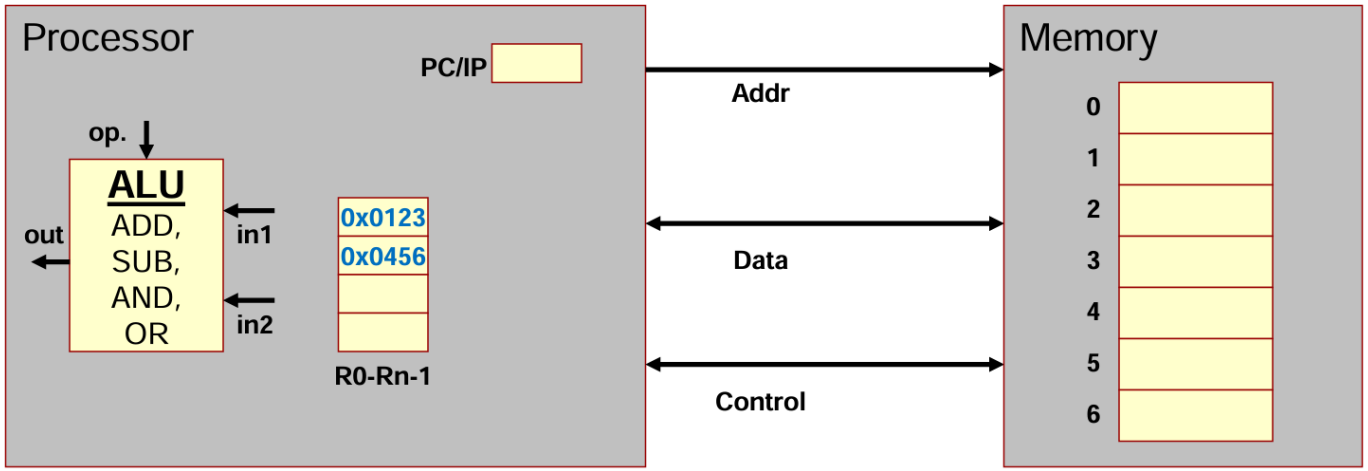
  - 执行加减、逻辑运算等(AND,OR,etc.)算术操作的数字电路



传统存算分离的指令集架构 – 核心部件1：Register

- Register负责将ALU运算结果暂存在靠近ALU的地方

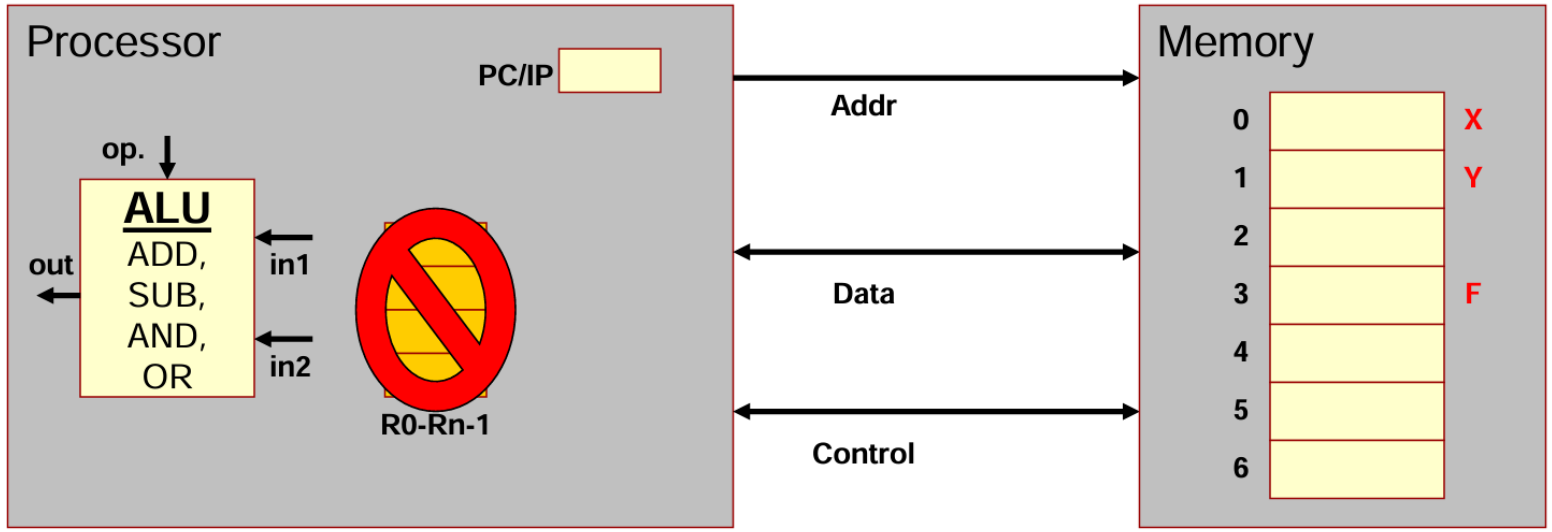
  - 访问存储器的时间通常比处理器要慢
  - 寄存器在处理器内部提供了快速的暂时存储位置



- 软件指令可以调用寄存器用于编程与编译
- 编程/编译使用这些寄存器作为输入（源位置）和输出（目标位置）

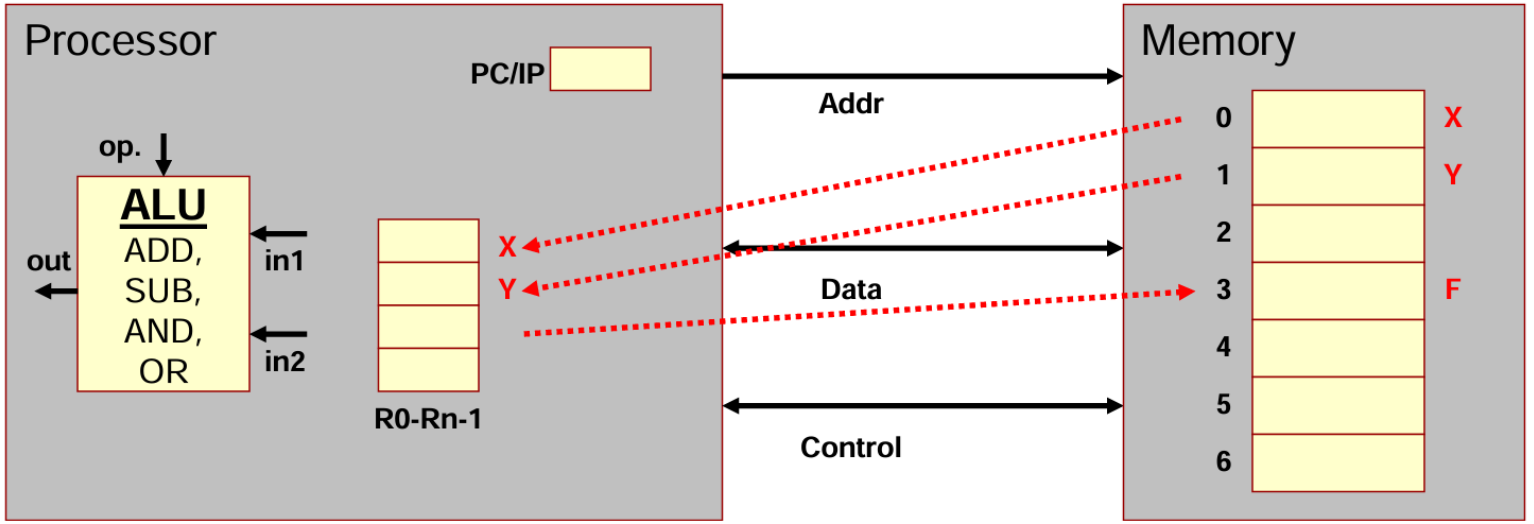
传统存算分离的指令集架构 – 核心部件1：Register

- Register的存在大幅减少了长延时的Memory访问
  - 无寄存器时计算 F = (X+Y)-(X*Y):
    - 需要ADD，MUL，SUB这些指令
    - 无寄存器
      - ADD：从存储器加载X和Y，存储计算结果到存储器
      - MUL：再次从存储器加载X和Y，存储计算结果到存储器
      - SUB：加载ADD和MUL的计算结果，存储计算结果到存储器
      - **共9次访存**



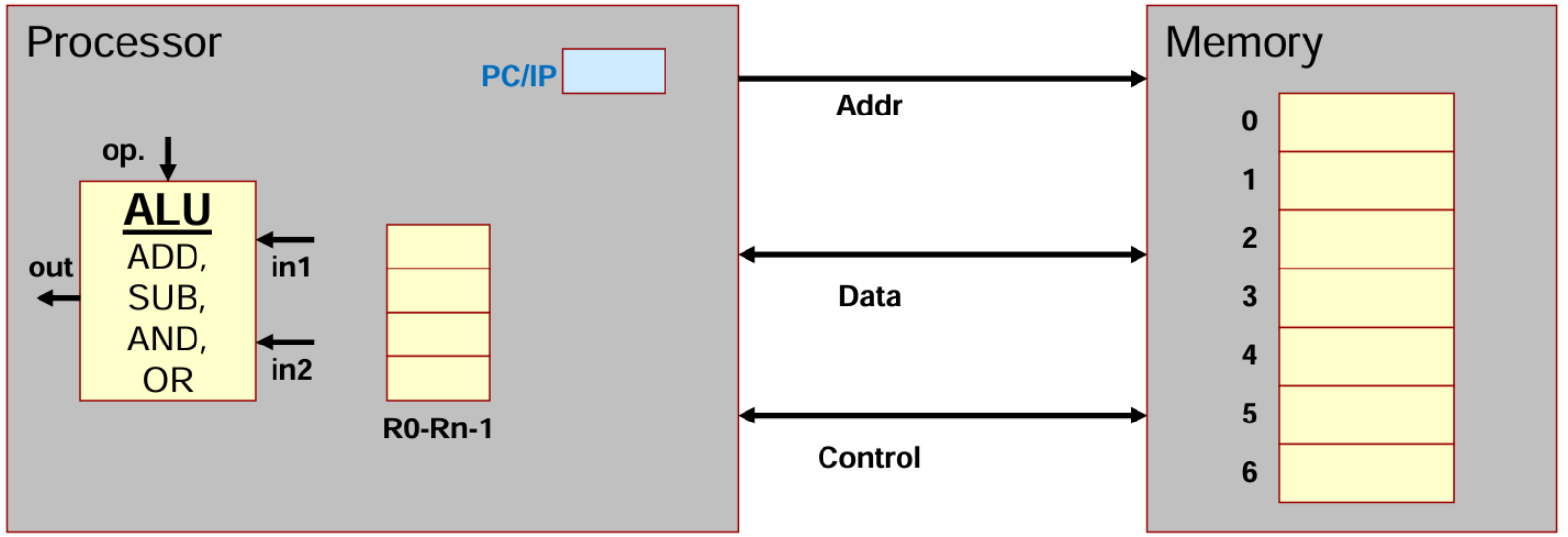
传统存算分离的指令集架构 – 核心部件1：Register

- Register的存在大幅减少了长延时的Memory访问
  - 使用寄存器时计算 F = (X+Y)-(X*Y):
    - 从存储器加载X和Y到寄存器R0，R1
    - ADD：计算R0+R1并存储到R2
    - MUL：计算R0*R1并存储到R3
    - SUB：计算R2-R3并存储到R4
    - 存储R4到存储器
    - 3次访存



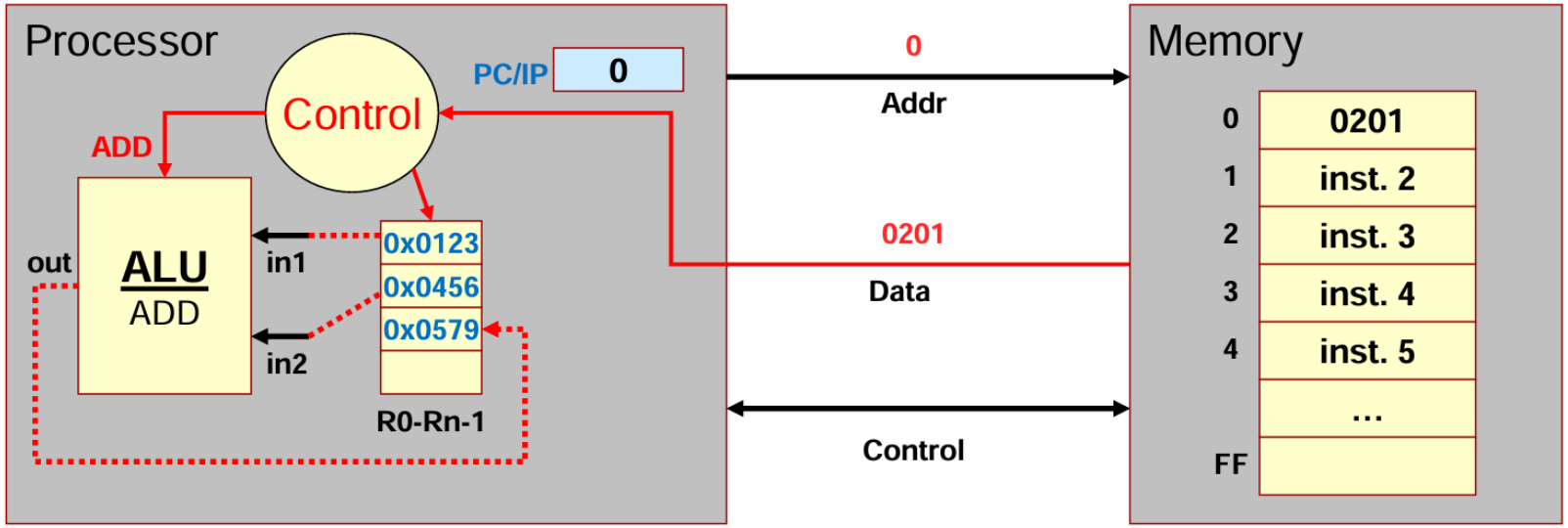
传统存算分离的指令集架构 – 核心部件1：Register

- Register还包括用于记录程序状态与指令状态的PC/IP
  - 要使处理器正确运行需要一些状态信息
  - 如程序计数器/指令指针（PC/IP）寄存器
    - 上文讲到处理器在译码与执行指令之前要从存储器获取指令
    - PC/IP寄存器保留下一个要获取指令的地址



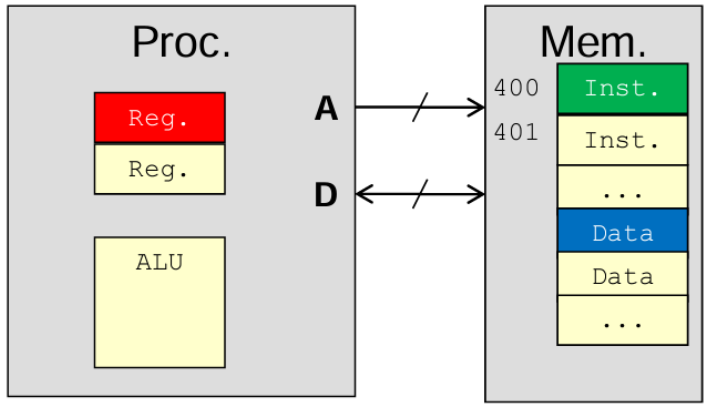
简单指令集架构的操作流程

- 指令从Memory中读取，ALU进行运算（可内含加、减、乘、除、逻辑、复杂计算单元等）
  - 假设0x0201是R2=R0+R1这个ADD指令的机器码
  - 控制逻辑将会是
    - 选取源寄存器（R0 和 R1）
    - 告诉ALU做加法
    - 选取目标寄存器（R2)



指令集数据的位置

- 数据可以存储在register、主存memory或指令内部
  - 源操作数存储在以下三个位置内
    - 寄存器值(eg. %rax)
    - 主存中的值（eg. 0x0200e8)
    - 在指令内部存储的常量（又叫“立即数immediate”）[eg. ADDI $1,D0]
      - $表示是常量或者是立即数
  - 目标操作数存储在
    - 寄存器
    - 存储器（由其地址指定）



目 录

CONTENTS

主流指令集都有哪些?

- X86、MIPS、ARM、RISC-V

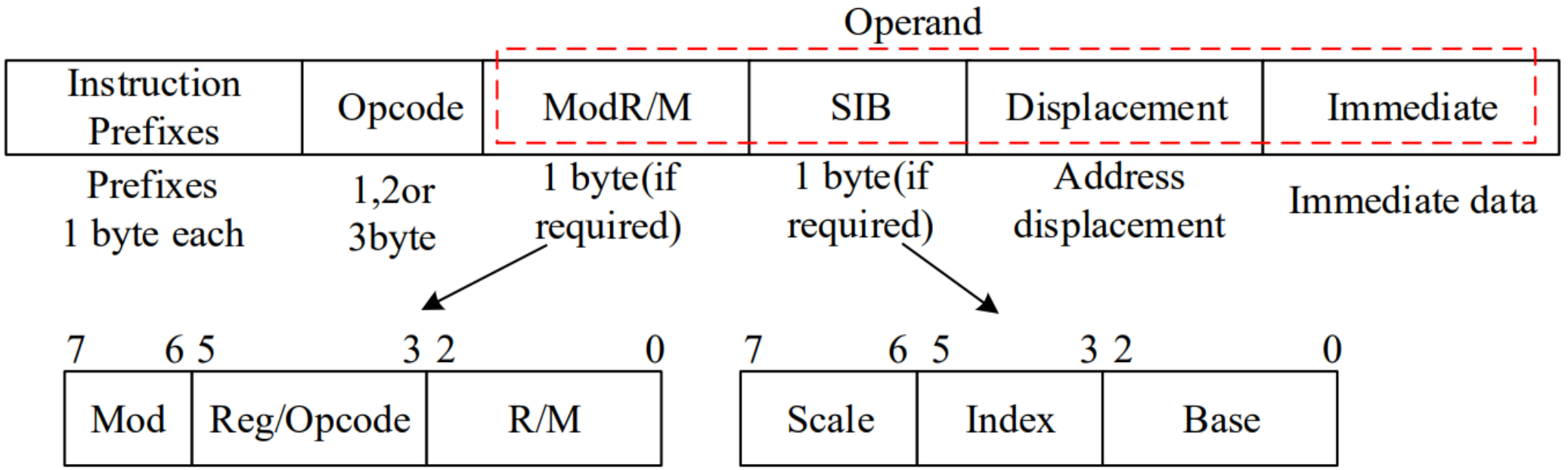
指令集	类型	运营公司	特点	代表厂商
X86	CISC	Intel、AMD	功能强大、通用性、兼容性、实用性	Intel、AMD
MIPS	RISC	MIPS	简洁、优化、高扩展性、寄存器多	Intel、IBM、龙芯、Oracle、Toshiba
ARM	RISC	ARM	低功耗、低成本、适用于移动设备	苹果、华为、谷歌
RISC-V	RISC	RISC-V基金会	完全开源、架构简单、移植性高、开源工具链	数百家大学、科研机构和企业
CUDA	RISC	Nvidia	张量高并发处理、图像处理	Nvidia

指令集架构一般需要哪些指令?

- **四大类：传输指令、运算指令、控制指令、系统指令**
  - 数据传输（mov指令）
    - 在处理器和存储器之间传输数据（加载load/保存save变量）
    - 其中一个操作数必须是处理器的寄存器（不能在两个存储器位置之间传输数据）
    - 通过指令的后缀来指定具体大小（movb，movw，movl，movq）
  - 算术逻辑单元ALU操作
    - 其中一个操作数必须是处理器的寄存器
    - 通过指令来指定大小和操作（addl，orq，andb，subw）
  - 控制指令
    - 无条件/有条件跳转（cmpq，jmp，je，jne，jl，jge）
    - 子例程调用（call，ret）
  - 系统指令
    - 只能通过OS或其他“监督”软件使用的指令（eg. int to access certain OS capabilities, etc.)

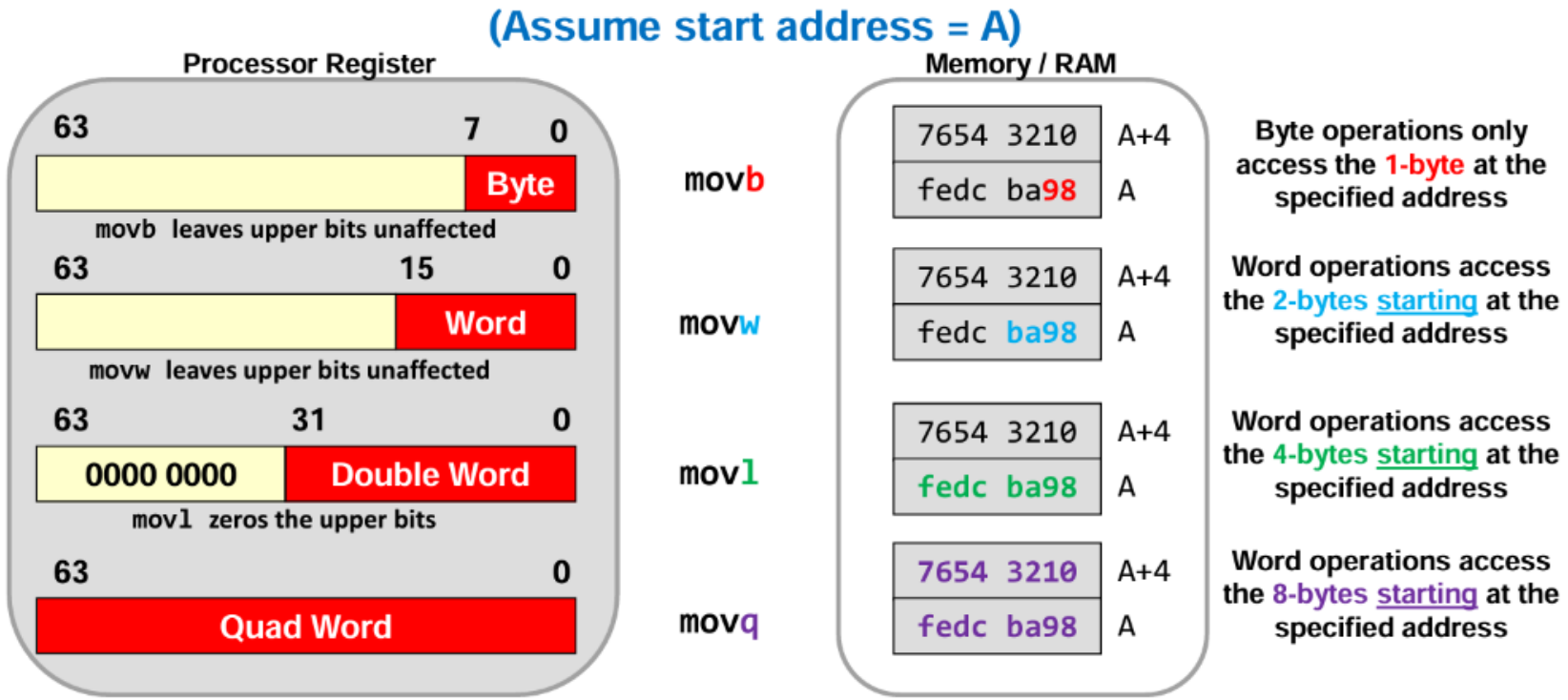
代表性指令集：X86一种典型的CISC指令

- 指令长度可变，较为复杂（多周期指令等）



指令集的分类1 – 传输指令

- 指令集可以看做链接软件和硬件的一个协议
  - 在处理器寄存器和存储器之间传输数据
  - 通过指令的后缀来指定具体大小（mov[bwlq]）
  - 起始地址应该能被访问地址大小整除



指令集的分类1 – 传输指令：指令的地址模式

北京大学 PEKING UNIVERSITY

- 指令集一般包含多种地址模式 – 以广泛商用的X86或MIPS为案例

X64使用16个64bit寄存器，寄存器的低字节可以独立作为32-，16-，8- 比特寄存器来被访问，他们的名字如下

8-byte register	Bytes 0-3	Bytes 0-1	Byte 0
%rax	%eax	%ax	%al
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

Name	Form	Example	Description
Immediate	$imm	movl $-500,%rax	R[rax] = imm.
Register	r_a	movl %rdx,%rax	R[rax] = R[rdx]
Direct Addressing	imm	movl 2000,%rax	R[rax] = M[2000]
Indirect Addressing	(r_a)	movl (%rdx),%rax	R[rax] = M[R[r_a]]
Base w/ Displacement	imm(r_b)	movl 40(%rdx),%rax	R[rax] = M[R[r_b]+40]
Scaled Index	(r_b,r_i,s†)	movl (%rdx,%rcx,4),%rax	R[rax] = M[R[r_b]+R[r_i]*s]
Scaled Index w/ Displacement	imm(r_b,r_i,s†)	movl 80(%rdx,%rcx,2),%rax	R[rax] = M[80 + R[r_b]+R[r_i]*s]

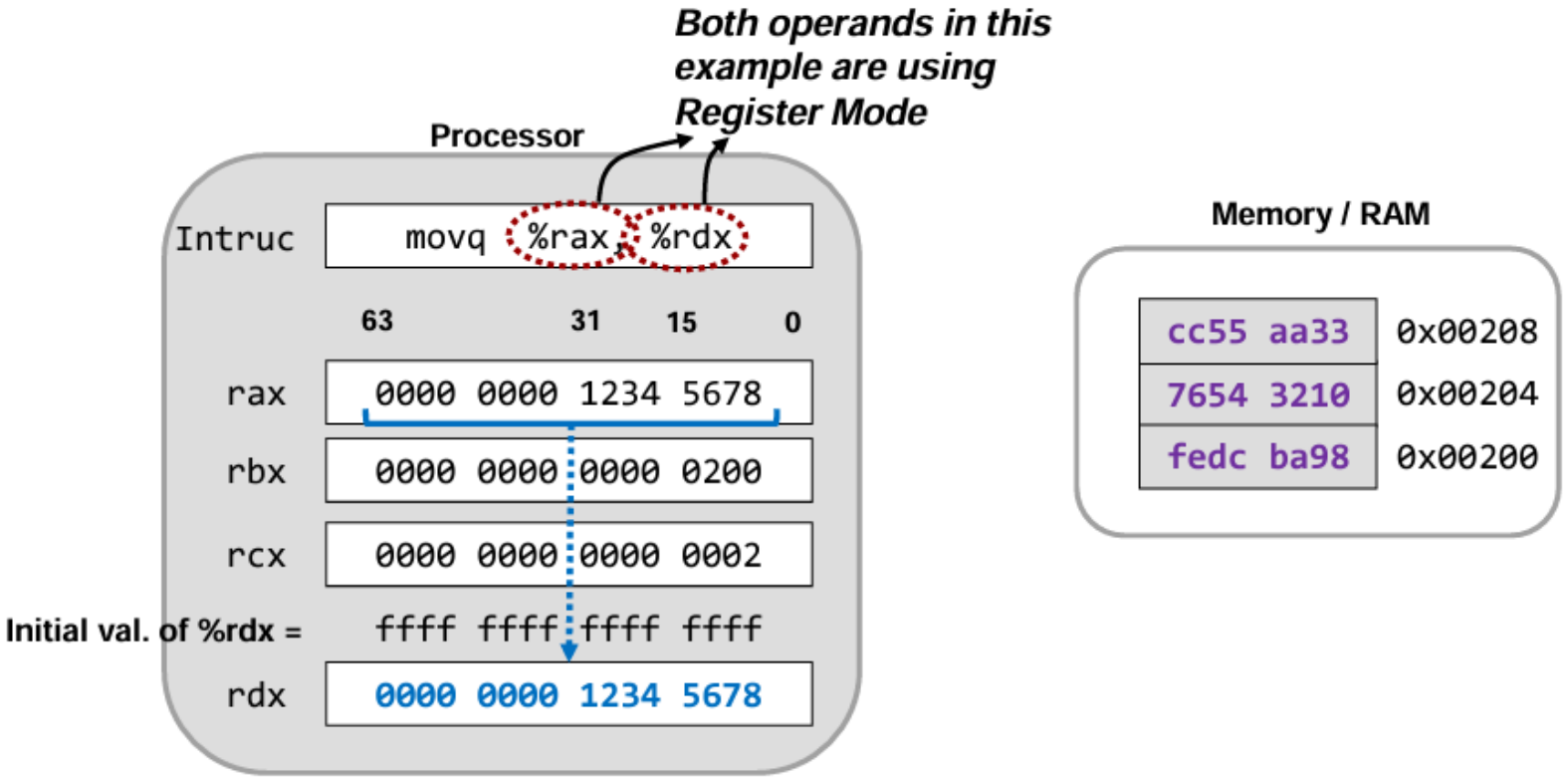
†Known as the scale factor and can be {1,2,4, or 8}

Imm = Constant, R[*x*] = Content of register *x*, M[addr] = Content of memory @ addr.

Purple values = effective address (EA) = Actual address used to get the operand

指令集的分类1 – 传输指令： Register Mode

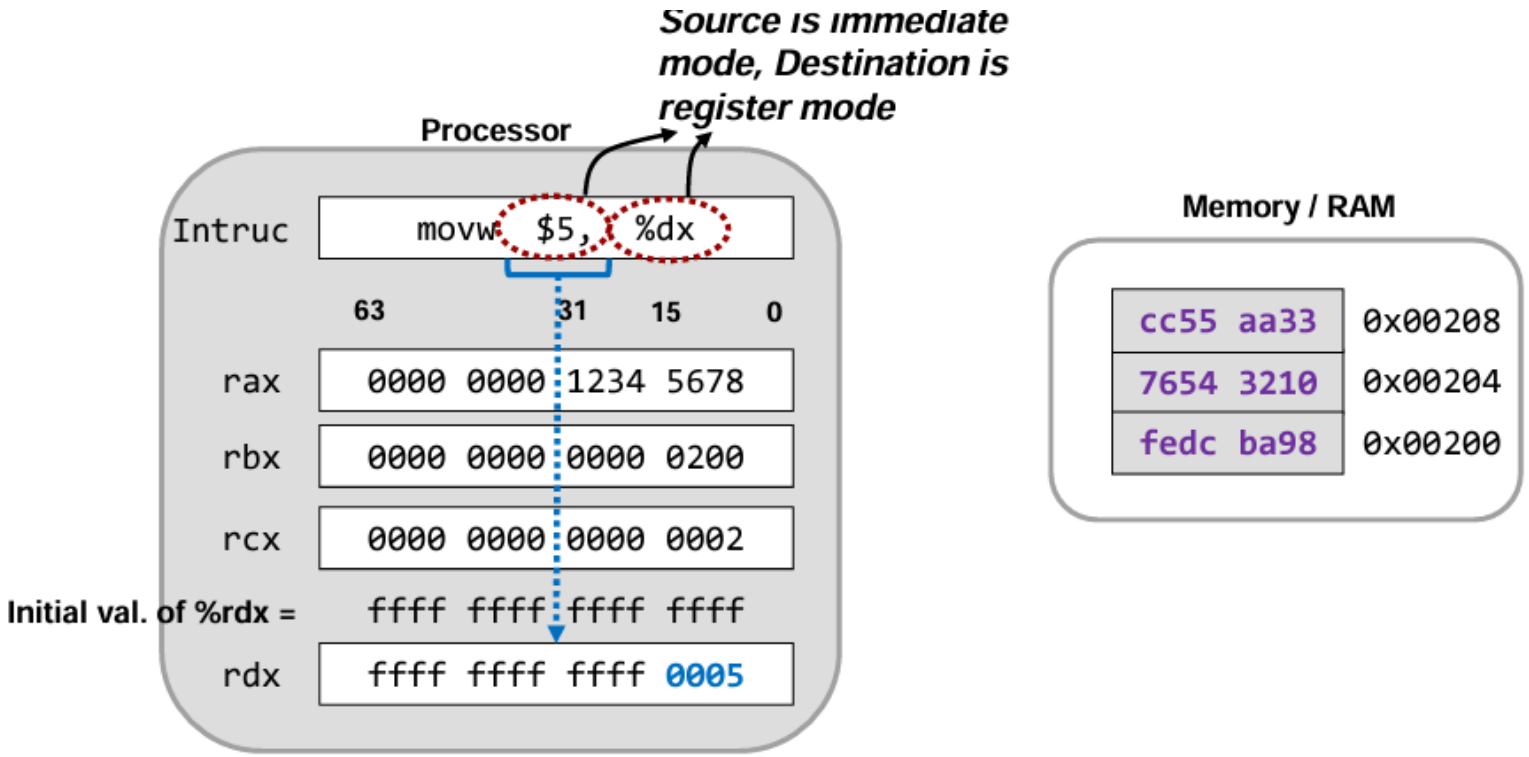
- 指定寄存器的内容作为操作数



指令集的分类1 – 传输指令： Immediate Mode

- 指定指令中的立即数作为操作数

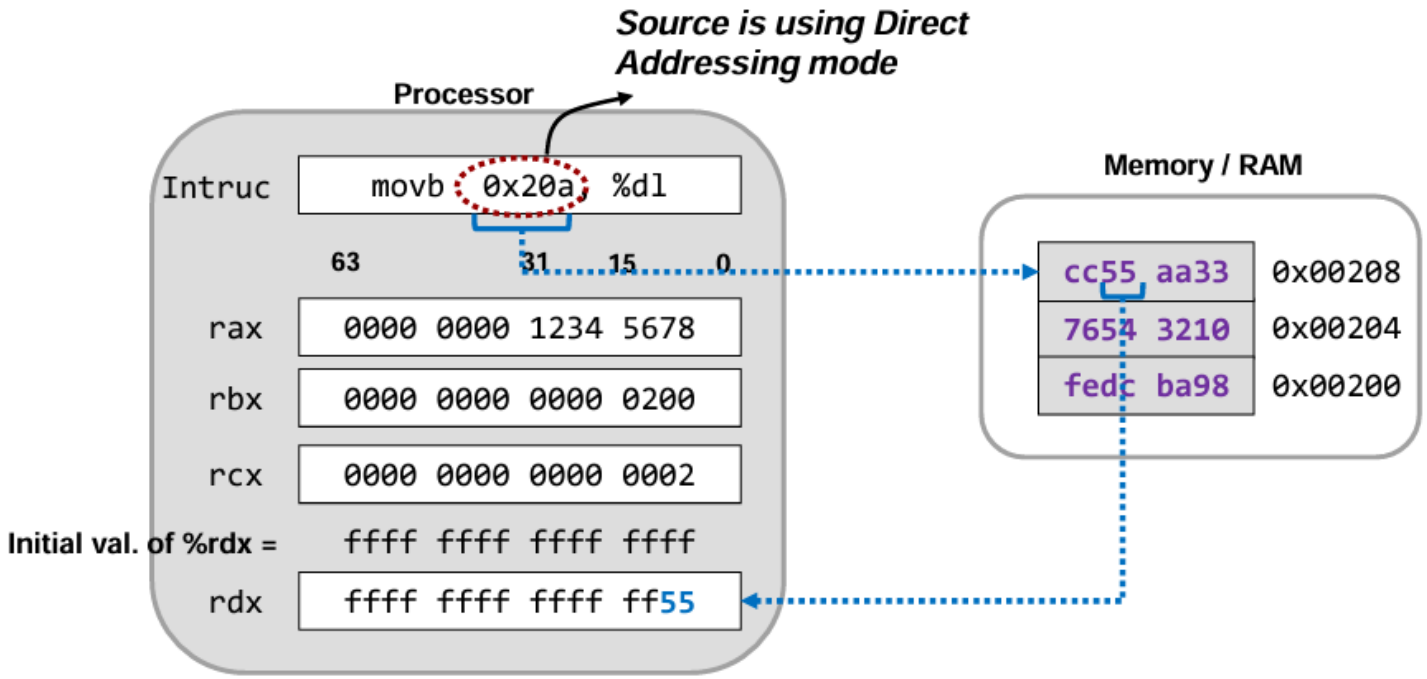
  - 符号’$’表示立即数，并且可以指定用十六进制或是十进制



指令集的分类1 – 传输指令： Direct Addressing Mode

- 指定真正操作数所在位置的存储器地址常量

- 地址可以用十六进制或是十进制

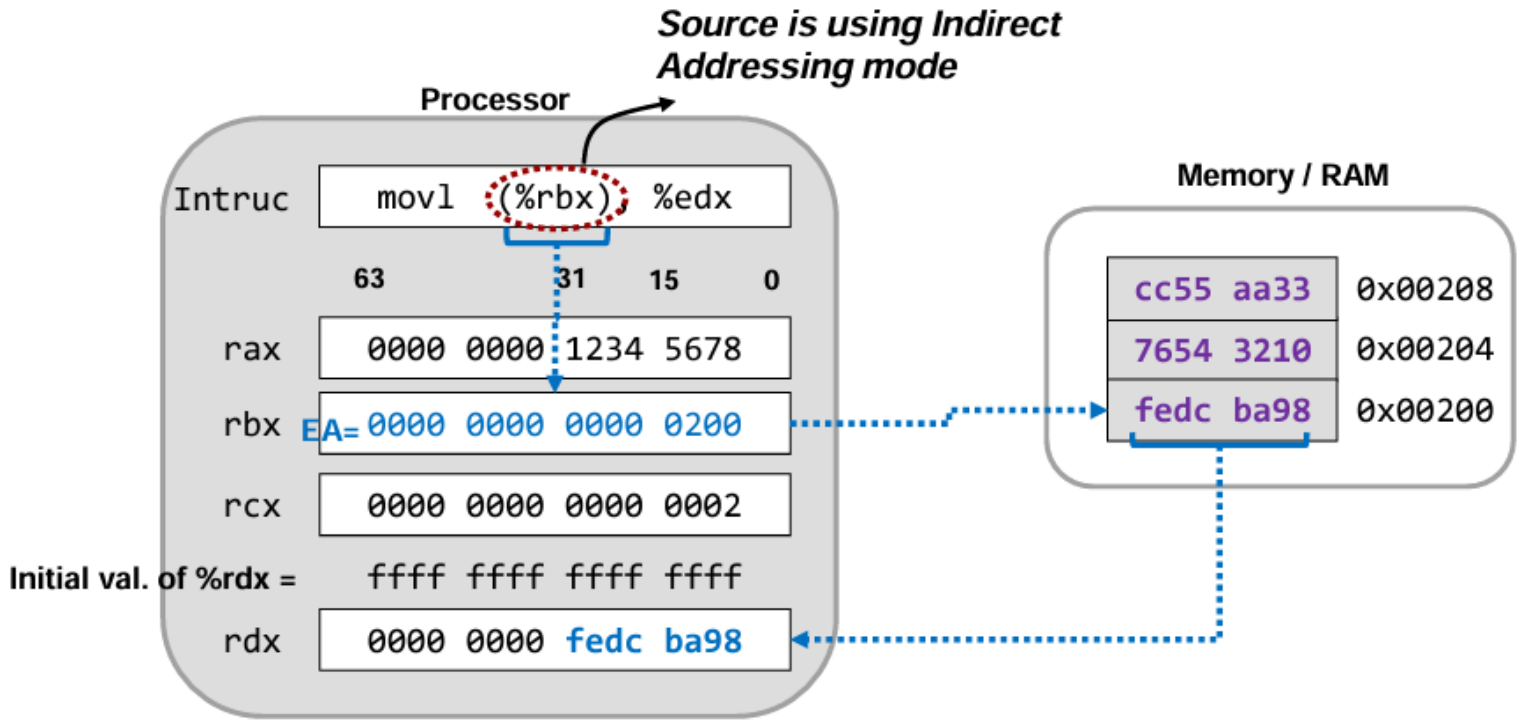


指令集的分类1 – 传输指令： Indirect Addressing Mode

- 指定存储器内容为真正操作数在存储器中的有效地址

类似于指针

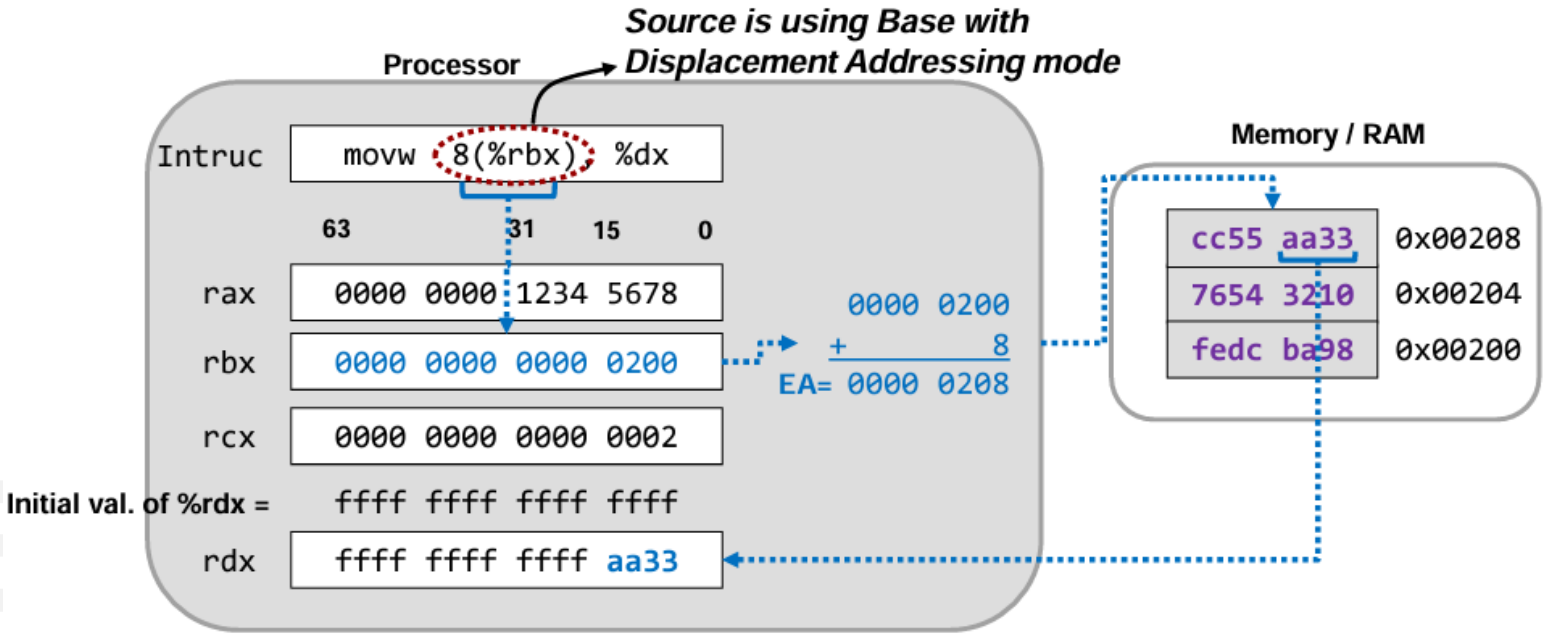
- 圆括号表示间接寻址模式



指令集的分类1 – 传输指令： Base/Indirect with Displacement Addressing Mode

- 采用d(%reg)来指定地址

  - 给寄存器值加一个常量，并用求和结果作为实际操作数在存储器中的有效地址



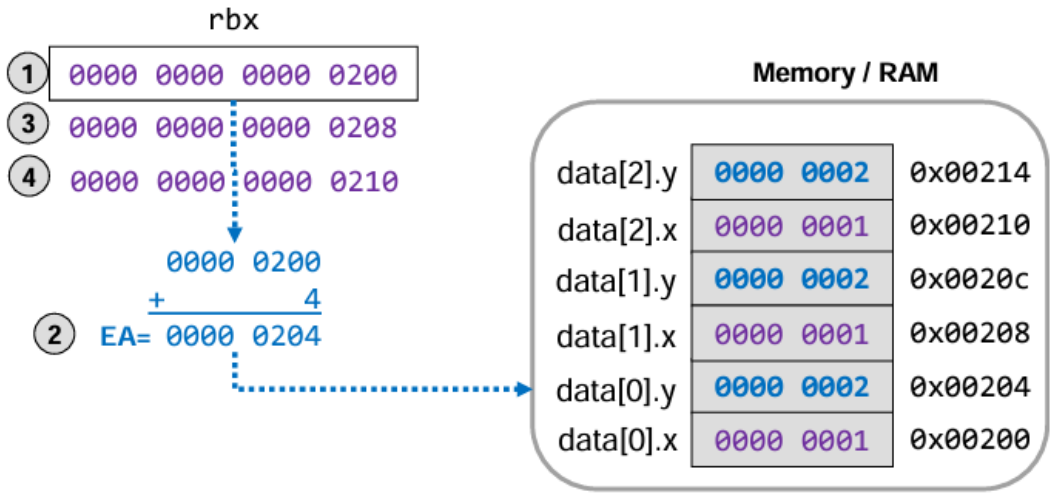
指令集的分类1 – 传输指令： Base/Indirect with Displacement Addressing Mode

- 为什么需要Base/Indirect with Displacement Addressing实际案例

  - Useful for access members of a struct or object

```
struct mystruct {
  int x;
  int y;
};
struct mystruct data[3];

int main()
{
  for(i=0; i<3; i++){
    data[i].x = 1;
    data[i].y = 2;
  }
}
```

C Code

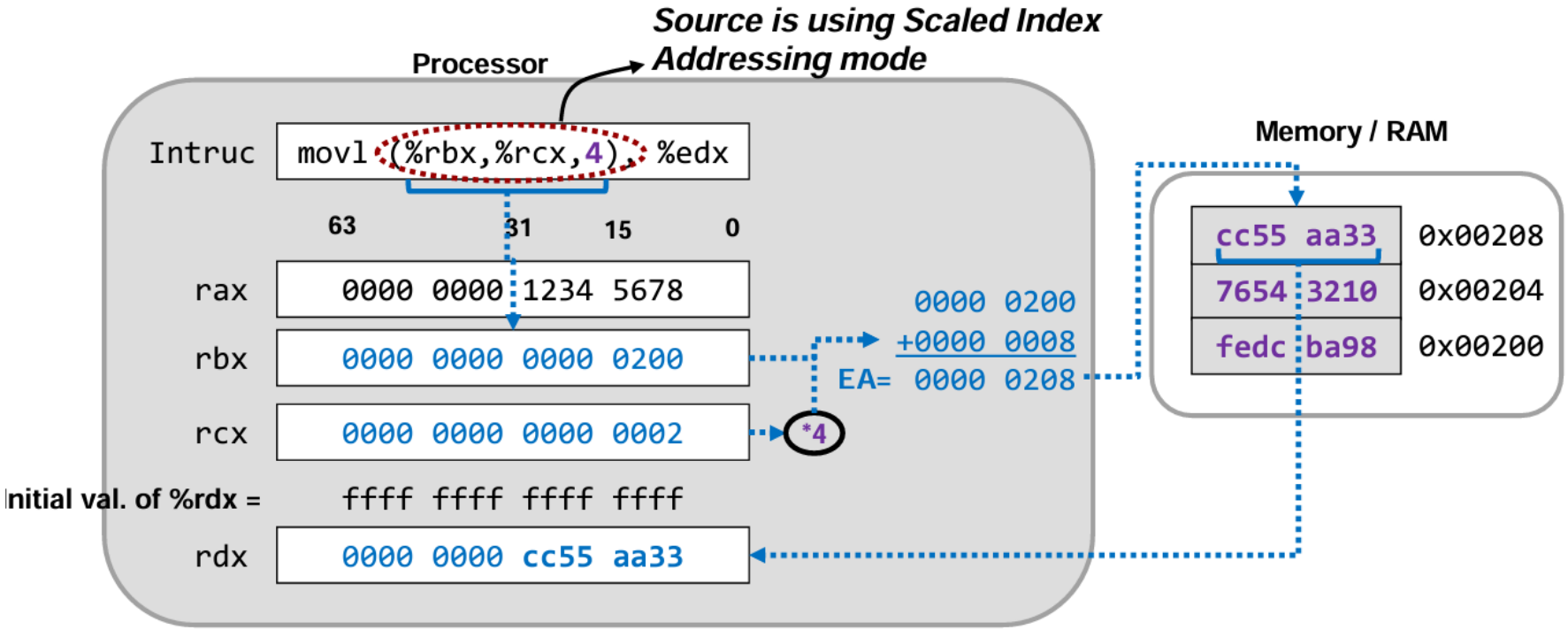



```
movq   $0x0200,%rbx
Loop 3 times {
1 3 4 movl  $1, (%rbx)
2     movl  $2, 4(%rbx)
      addq  $8, %rbx
}
```

Assembly

指令集的分类1 – 传输指令： Scaled Index Addressing Mode

- 地址格式：Form: (%reg1,%reg2,s) [s = 1, 2, 4, or 8]
  - 用%reg1+%reg2*s作为实际操作数在存储器中的有效地址

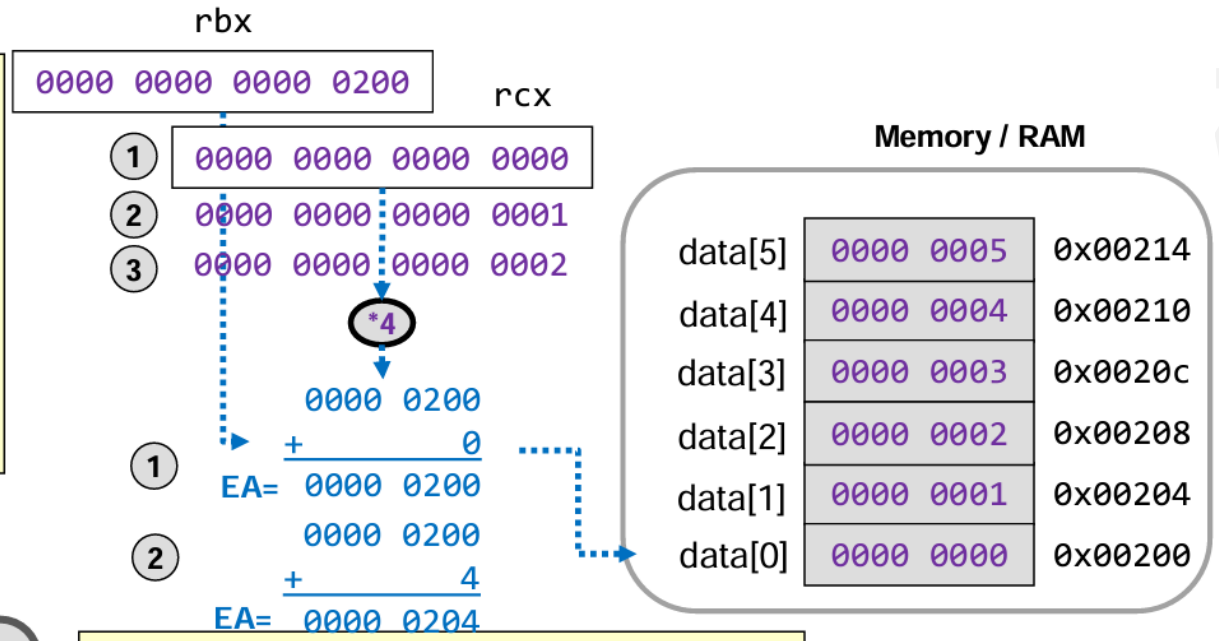


指令集的分类1 – 传输指令： Scaled Index Addressing Mode

- 为什么需要Scaled Index Addressing Mode实际案例
  - Useful for accessing array elements

```
int data[6];

int main()
{
  for(int i=0; i<6; i++){
    data[i] = i;
    // *(startAddr+4*i) = i;
  }
}
```

C Code



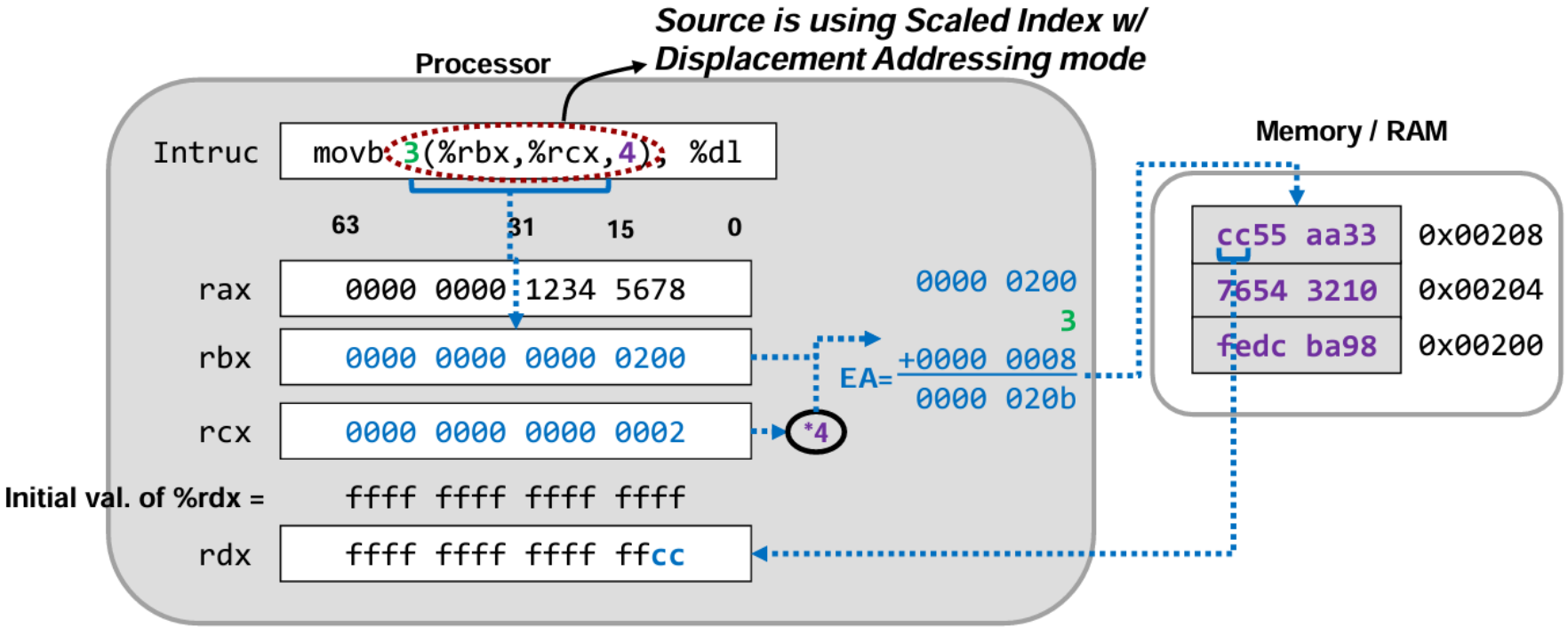
Array of:
- chars/bytes => Use s=1
- shorts/words => Use s=2
- ints/floats/dwords => Use s=4
- long longs/doubles/qwords => Use s=8

```
movq    $0x0200,%rbx
movl    $0, %rcx
Loop 6 times {
  movl  %rcx, (%rbx,%rcx,4)
  addl  $1, %rcx
}
```

Assembly

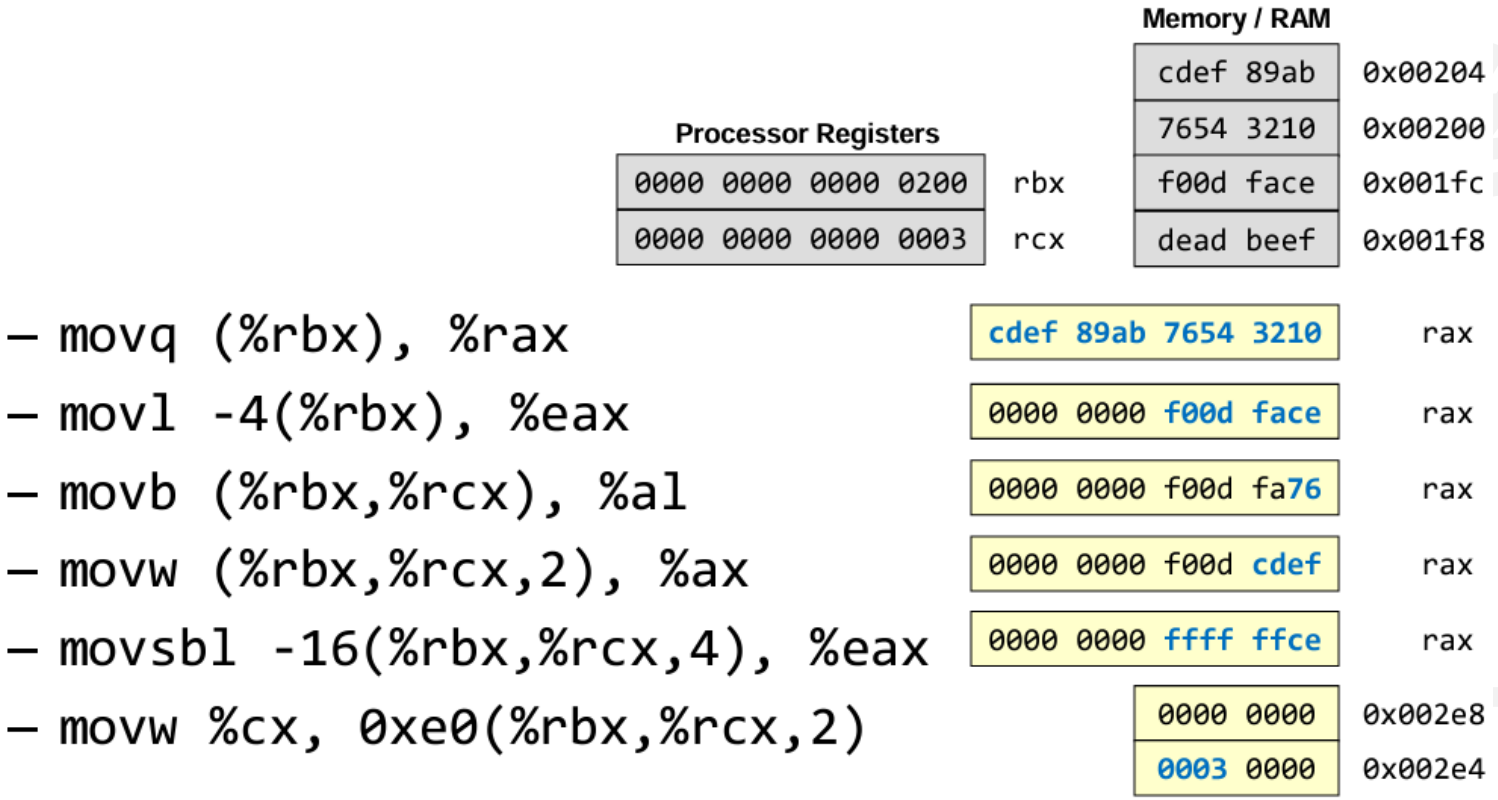
指令集的分类1 – 传输指令： Scaled Index w/ Displacement Addressing Mode

- 集合Scale和Displacement：地址 = d(%reg1,%reg2,s) [s = 1, 2, 4, or 8]
  - 用d+%reg1+%reg2*s作为实际操作数在存储器中的有效地址



指令集的分类1 – 传输指令： Addressing Mode案例

- 实际程序中可能由多种Addressing Mode共同组成



指令集的分类2 – 计算指令

- 利用ALU来完成实际计算任务

C operator	Assembly	Notes
+	add[b,w,l,q] src1,src2/dst	src2/dst += src1
-	sub[b,w,l,q] src1,src2/dst	src2/dst -= src1
&	and[b,w,l,q] src1,src2/dst	src2/dst &= src1
|	or[b,w,l,q] src1,src2/dst	src2/dst |= src1
^	xor[b,w,l,q] src1,src2/dst	src2/dst ^= src1
~	not[b,w,l,q] src/dst	src/dst = ~src/dst
-	neg[b,w,l,q] src/dst	src/dst = (~src/dst) + 1
++	inc[b,w,l,q] src/dst	src/dst += 1
--	dec[b,w,l,q] src/dst	src/dst -= 1
* (signed)	imul[b,w,l,q] src1,src2/dst	src2/dst *= src1
<< (signed)	sal cnt, src/dst	src/dst = src/dst << cnt
<< (unsigned)	shl cnt, src/dst	src/dst = src/dst << cnt
>> (signed)	sar cnt, src/dst	src/dst = src/dst >> cnt
>> (unsigned)	shr cnt, src/dst	src/dst = src/dst >> cnt
==, <, >, <=, >=, != (src2 ? src1)	cmp[b,w,l,q] src1, src2 test[b,w,l,q] src1, src2	cmp performs: src2 – src1 test performs: src1 & src2

指令集的分类2 – 计算指令

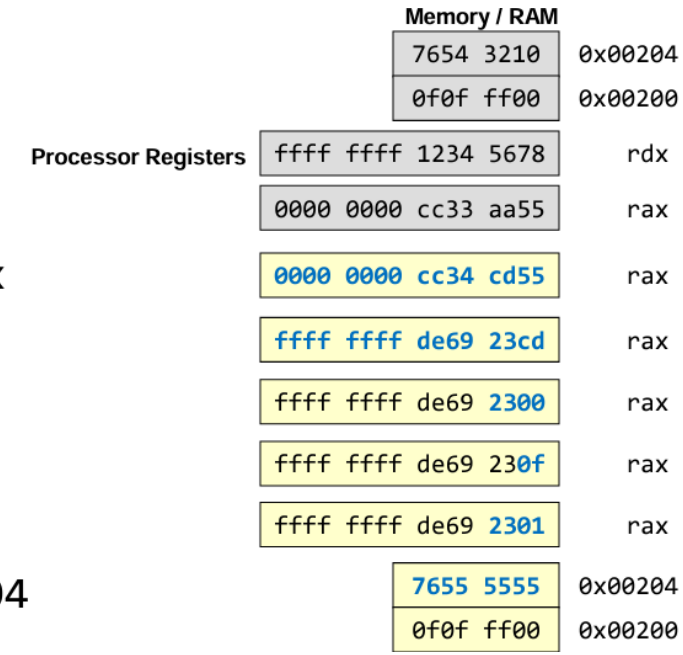
- 利用ALU来完成实际计算任务

- 基于给定数据尺寸执行算术/逻辑操作
- 限制：两个操作数都不能是存储器

- Format
  - add[b,w,l,q] src2, src1/dst
  - Example 1: addq %rbx, %rax (%rax += %rbx)
  - Example 2: subq %rbx, %rax (%rax -= %rbx)

Work from right->left->right

- Initial Conditions
  - addl $0x12300, %eax
  - addq %rdx, %rax
  - andw 0x200, %ax
  - orb 0x203, %al
  - subw $14, %ax
  - addl $0x12345, 0x204



指令集的分类2 – 计算指令：实际案例

- 计算指令配合传输指令完成一个代码的编译过程

```
// data = %edi
// val  = %esi
// i    = %edx
int f1(int data[], int* val, int i)
{
  int sum = *val;
  sum += data[i];
  return sum;
}
```

Original Code

```
f1:

    movl    (%esi), %eax
    addl    (%edi,%edx,4), %eax


    ret
```

Compiler Output

```
struct Data {
  char c;
  int d;
};

// ptr  = %edi
// x    = %esi
int f1(struct Data* ptr, int x)
{
  ptr->c++;
  ptr->d -= x;
}
```

Original Code

```
f1:

    addb    $1, (%edi)
    subl    %esi, 4(%edi)


    ret
```

Compiler Output

指令集的分类3 – 控制指令

- 控制指令地址跳跃

适用于if、case判断语句以及for、while等循环语句等等

将会在后续分支预测等内容深入讲解


```
Address    Instruction
004937F7   MOV EAX,200
004937FC   MOV EDX,50
00493801   ADD EAX,67F0
00493806   MOV ECX,490AB3
0049380B   JMP 00497000

;Pretend there is a lot of code inbetween here.

00497000   DEC EDX
00497001   MOV DWORD [49E6CC],EDX
00497007   MOV EAX,EDX
```

Jump to address 497000

then continue the code.

指令集的分类4 – 系统指令

- 与上层操作系统OS对接，例如OS可更改register内容等

编译器设计内容

主讲：陶耀宇、李萌

代表性指令集： RISC-V一种典型的RISC指令

- 完全开源，扩展性较好，指令种类多

31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	R-Type
$imm_{11:0}$		rs1	funct3	rd	op	I-Type
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	S-Type
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	B-Type
$imm_{31:12}$				rd	op	U-Type
$imm_{20,10:1,11,19:12}$				rd	op	J-Type
fs3	funct2	fs2	fs1	funct3	fd	op	R4-Type
5 bits	2 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

Figure B.1 RISC-V 32-bit instruction formats

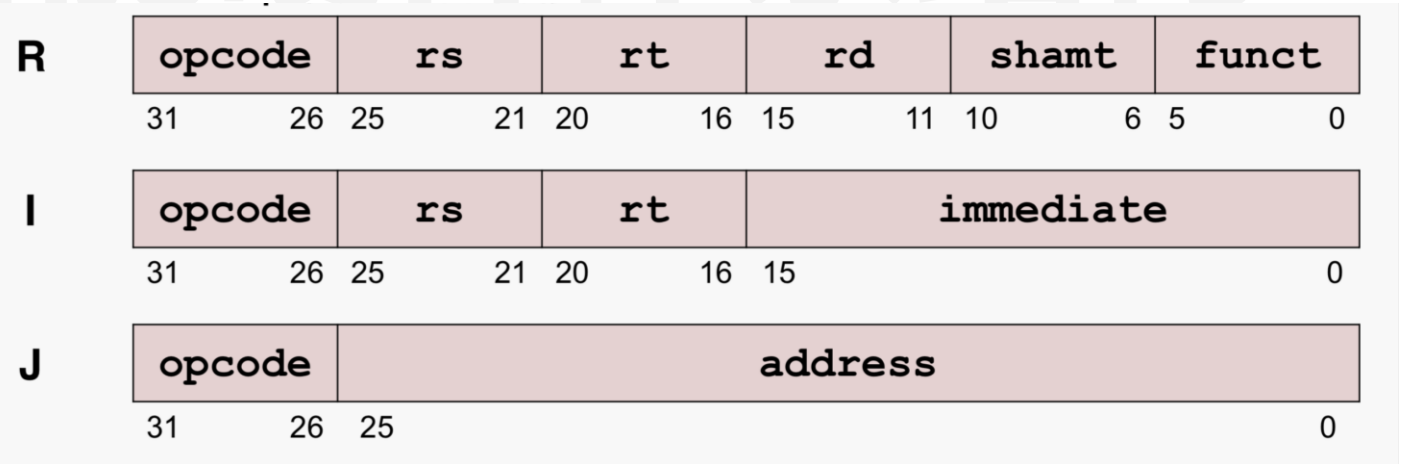
- `imm`: signed immediate in $imm_{11:0}$
- `uimm`: 5-bit unsigned immediate in $imm_{4:0}$
- `upimm`: 20 upper bits of a 32-bit immediate, in $imm_{31:12}$
- `Address`: memory address: rs1 + SignExt($imm_{11:0}$)
- `[Address]`: data at memory location Address
- `BTA`: branch target address: PC + SignExt({$imm_{12:1}$, 1'b0})
- `JTA`: jump target address: PC + SignExt({$imm_{20:1}$, 1'b0})
- `label`: text indicating instruction address
- `SignExt`: value sign-extended to 32 bits
- `ZeroExt`: value zero-extended to 32 bits
- `csr`: control and status register

代表性指令集： MIPS一种典型的RISC指令

- 指令长度固定，相对简单（单周期指令）
  - 3种CPU指令， 都是32比特对齐words
- I-type (Instruction)
- J-type (Jump)
- R-type (Register)

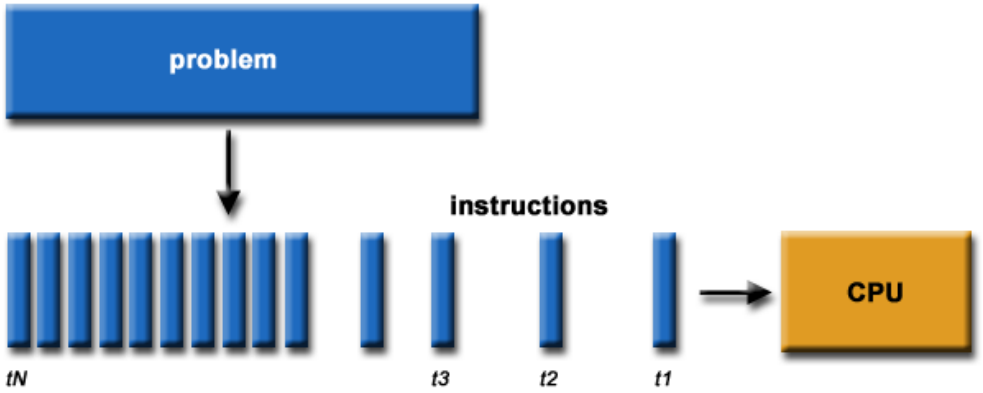
•Opcode

- 6-bit operation code
- There are 3 different register specifiers:
  - RD - 5-bit destination register
  - RS - 5-bit source register
  - RT - 5-bit target register

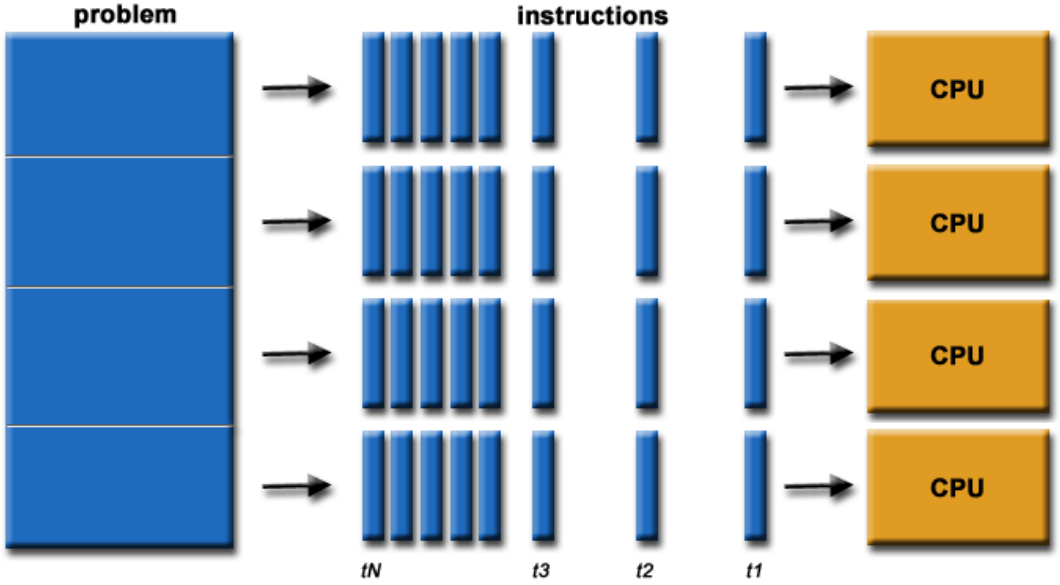


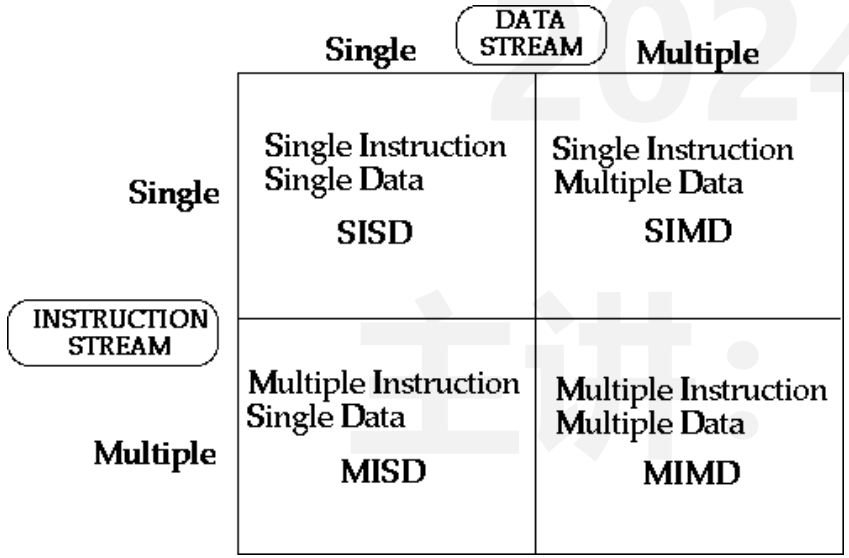
代表性指令集： GPU的CUDA指令集

- SISD、SIMD、MISD、MIMD的扩展

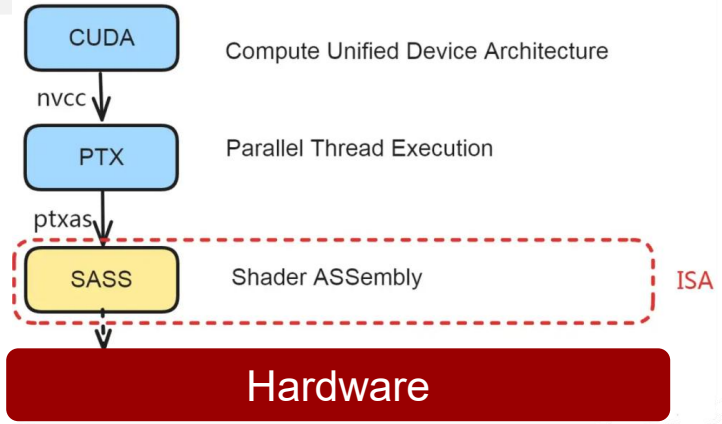


并行计算



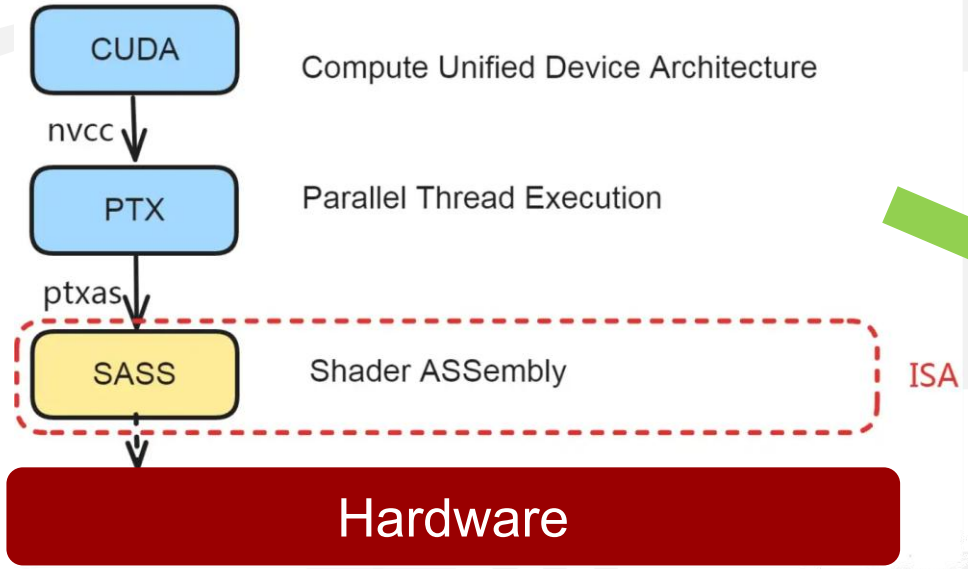


CUDA指令集的层次



代表性指令集： GPU的CUDA指令集

- PTX和SASS



CUDA C/C++程序编译后，一般NVCC会同时生成PTX和SASS，用户也可以指定只生成其中一种。SASS是机器码的硬件指令集，编译的SM版本与当前GPU的SM版本不对应的话是不能运行的

从SASS上抽象出来的一种更上层的软件编程模型，介于CUDA C/C++和SASS之间

SASS指令集与GPU的SM架构有直接对应关系，一旦硬件架构设计完成就不再改变

代表性指令集： GPU的CUDA指令集

- PTX和SASS

PTX

```
mov.u32   %r1, %clock;
add.u32   %r11, 6, %r7;
add.u32   %r12, %r5, 7;
add.u32   %r13, %r12, %r1;
mov.u32   %r2, %clock;
```

SASS

```
S2UR R356
DEPBAR.LE
IADD3 R14   R19  R23
IADD3 R13  R12  R-167
IADD3 R14  R14  R35  R23
CS2R.32 R322
```

PTX

```
mov.u64   %rd50, %clock64;
add.u32   %r11, 6, %r7;
add.u32   %r12, %r5, 7;
add.u32   %r13, %r12, %r1;
mov.u64   %rd51, %clock64;
```

SASS

```
CS2R R128
IADD3 R14   R1915   R231
IADD3 R141  R123    R335
IADD3 R32  R146 R123 R231
CS2R R133
```

Mapping of PTX to SASS

代表性指令集： GPU的CUDA指令集

北京大学 PEKING UNIVERSITY

- PTX和SASS

PTX	SASS	cycles	PTX	SASS	cycles
Add / sub instruction			**Min/Max instructions**		
add.u16	UIADD3	2	Min.u16	ULOP3.LUT+UISETP.LT.U32.AND+USEL	8
addc.u32	IADD3.X	2	min.u32	IMNMX.U32	2
add.u32	IADD	2	min.u64	UISETP.LT.U32.AND+2*USEL	8
add.u64	UIADD3.x+ UIADD3	4	min.s16	PRMT+IMNMX	4
add.s64	UIADD3.x+UIADD3	4	min.s32	IMNMX	2
add.f16	HADD	2	Min.s64	UISETP.LT.U32.AND+UISETP.LT.AND.EX+2*USEL	8
add.f32	FADD	2	min.f16	HMNMX2+PRMT	4
add.f64	DADD	4	min.f32	FMNMX	2
Mul instruction			min.f364	DSETP.MIN.AND+IMAD.MOV.U32+UMOV+FSEL	10
mul.wide.u16	LOP3.LUT+IMAD	4	**Neg instruction**		
mul.wide.u32	IMAD	4	neg.s16	UIADD3+UPRMT	5
mul.lo.u16	LOP3.LUT+IMAD	4	neg.s32	IADD3	2
mul.lo.u32	IMAD	2	neg.s64	IMAD.MOV.U32+HFMA2.MMA+MOV+UIADD3	10
mul.lo.u64	IMAD	2	neg.f32	FADD or IMAD.MOV.U32 *	2
mul24.lo.u32	PRMT + IMAD	3	neg.f64	DADD+(UMOV)	4
mul24.hi.u32	UPRMT+USHF.R.U32.HI+IMAD.U32+PRMT	9	**FMA instruction**		
mul.rn.f16	HMUL2	2	fma.rn.f16	HFMA2	2
mul.rn.f32	FMUL	2	fma.rn.f32	FFMA	2
mul.rn.f64	DMUL	4	fma.rn.f64	DFMA	4
MAD Instruction			**Sqrt Instruction**		
mad.lo.u16	LOP3.LUT+IMAD	4	sqrt.rn.f32	[multiple instrs including MUFU.RSQ]	190-235
mad.lo.u32	FFMA	2	sqrt.approx.f32	[multiple instrs including MUFU.SQRT]	2-18
mad.lo.u64	IMAD	2	sqrt.rn.f64	[multiple insts including MUFU.RSQ64]	260-340
mad24.lo.u32	SGXT.U32 + IMAD	4	**Rsqrt Instruction**		
mad24.hi.u32	USHF.R.U32.HI+UIMAD.WIDE.U32+2*UPRMT+IADD3	11	rsqrt.approx.f32	[multiple insts including MUFU.RSQ]	2-18
mad.rn.f32	FFMA	2	rsqrt.approx.f64	MUFU.RSQ64H	8-11
mad.rn.f64	DFMA	4	**Rcp Instruction**		
Sad Instruction			rcp.rn.f32	[multiple insts including MUFU.RCP]	198
sad.u16/s16	(2*LOP3) +ULOP3+ VABSDIFF	6	rcp.approx.f32	[multiple insts including MUFU.RCP]	23
sad.u32/s32	VABSDIFF +IMAD (1 IMAD + 1 Umov for 3 instrs)	3	rcp.rn.f64	[multiple insts including MUFU.RCP64H]	244
sad.u64/s64	UISETP.GE.U32.AND+UIADD+IADD	10	ex2.approx.f32	FSTEP + FMUL + MUFU.EX2 + FMUL	14

代表性指令集： GPU的CUDA指令集

- PTX和SASS

Div / Rem Instruction			Pop Instruction		
rem/div.u16/s16	multiple instructions	290	popc.b32S	POPC	6
rem/div.s32/u32	multiple instructions	66	popc.b64	2*UPOPC + UIADD3	7
rem/div.u64/s64	multiple instructions	420	**Clz Instruction**		
div.rn.f32	multiple instructions	525	clz.b32	FLO.U32 + IADD	7
div.rn.f64	multiple instructions	426	clz.b64	UISETP.NE.U32.AND+USEL+UFLO.U32+2*UIADD3	13
Abs Instruction			**Bfind Instruction**		
abs.s16	PRMT+IABS+PRMT	4	bfind.u32	FLO.U32	6
abs.s32	IABS	2	bfind.u64	FLO.U32+ISETP.NE.U32.AND+IADD3+BRA	164
abs.s64	UISETP.LT.AND+UIADD3.X +UIADD3+2*USEL	11	bfind.s32	FLO	6
abs.f16	PRMT	1	bfind.s64	multiple instructions	195
abs.ftz.f32	FADD.FTZ	2	**testp Instruction**		
abs.f64	DADD or (DADD+UMOV)	4	testp.normal.f32	IMAD.MOV.U32+2*ISETP.GE.U32.AND	0 or 6
Brev Instruction			testp.subnor.f32	ISETP.LT.U32.AND	0 or 6
brev.b32	BREV + SGXT.U32	2	testp.normal.f64	2*UISETP.LE.U32.AND+2*UISETP.GE.U32.AND	13
brev.b64	2*UBREV+MOV	6	testp.subnor.f64	UISETP.LT.U32.AND+2*UISETP.GE.U32.AND.EX	8
copysign Instruction			**Other Instruction**		
copysign.f32	2*LOP3.LUT or 1.5*LOP3.LUT	4	sin.approx.f32	FMUL + MUFU.SIN	8
copysign.f64	2*ULOP3.LUT+IMAD.U32+*MOV	6	cos.approx.f32	FMUL.RZ+MUFU.COS	8
and/or/xor Instruction			lg2.approx.f32	FSETP.GEU.AND+FMUL+MUFU.LG2+FADD	18
and.b16	LOP3.LUT or 1.5*LOP3.LUT	2	ex2.approx.f32	FSETP.GEU.AND+2*FMUL+MUFU.EX2	18
and.b32	LOP3.LUT	2	ex2.approx.f16	MUFU.EX2.F16	6
and.b64	ULOP3.LUT	2-3	tanh.approx.f32	MUFU.TANH	6
Not Instruction			tanh.approx.f16	MUFU.TANH.F16	6
not.b16	LOP3.LUT	2	bar.warp.sync;	NOP	changes
not.b32	LOP3.LUT	2	fns.b32	multiple instructions	79
not.b64	2*ULOP3.LUT	4	cvt.rzi.s32.f32	F2I.TRUNC.NTZ	6
lop3 Instruction			setp.ne.s32	ISETP.NE.AND	10
lop3.b32	IMAD.MOV.U32+LOP3.LUT	4	mov.u32 clock	CS2R.32	2
cnot Instruction			**Bfi Instruction**		
cnot.b16	ULOP3.LUT+ISETP.EQ.U32.AND+SEL	5	bfi.b32	3*PRMT+2*IMAD.MOV+SHF.L.U32+BMSK+LOP3.LUT	11
cnot.b32	UISETP.EQ.U32.AND+USEL	4	bfi.b64	UMOV+USHF.L.U32+(UIADD3+ULOP3.LUT)*	5
cnot.b64	multiple instructions	11	**dp4a.u32/s32 Instruction**		
bfe Instruction			dp4a.u32.u32	IMAD.MOV.U32+IDP.4A.U8.U8	135-170
bfe.s32/.u32	3*PRMT+2*IMAD.MOV+SHF.R.U32.HI+SGXT/.U32	11	**dp2a.u32/s32 Instruction**		
bfe.u64	UMOV+USHF.L.U32+(UIADD3+ULOP3.LUT)*	5	dp2a.lo.u32.u32	IMAD.MOV.U32+IDP.2A.LO.U16.U8	135-170
bfe.s64	multiple instructions	14			