



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

第四讲：指令集与流水线架构

主讲：陶耀宇

2025年春季

• 课程作业情况

- **第1次作业截止日期：3月18号晚11:59**
 - 6次作业可以使用总计6个Late day
 - Late Day耗尽后，每晚交1天扣除20%当次作业分数
- **第1次lab时间：3月10晚上线 - 4月10晚11:59**
 - **2个基础任务 (50%+50%) + 1个Bonus (可2选1, 50%)**
- **第2次lab时间：4月10-6月10**

- 课程作业情况

- 第1次lab时间：3月10-4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

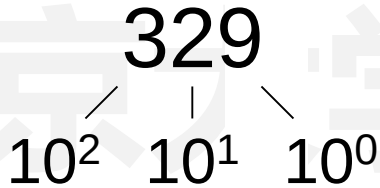
目录

CONTENTS

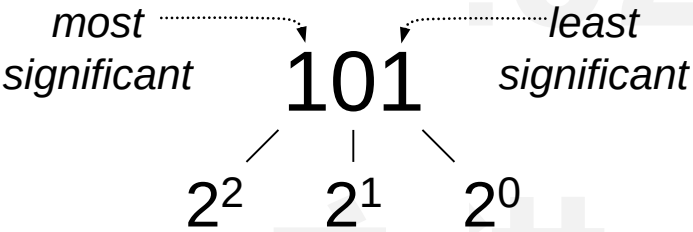


- 01. 晶体管与逻辑门电路基础**
- 02. 电路延迟分析与逻辑功效**
- 03. 动态逻辑电路与时序电路**
- 04. 复杂计算单元与数据格式**

- 最基础的二进制数据格式 – 原码（无符号数）



$$3 \times 100 + 2 \times 10 + 9 \times 1 = 329$$



$$1 \times 4 + 0 \times 2 + 1 \times 1 = 5$$

2^2	2^1	2^0	
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

An n -bit unsigned integer
represents 2^n values:
from 0 to $2^n - 1$

• 有符号数的表现格式 – Sign and Magnitude

一个n bit数可以表示 2^n 不同的值

- 近一半赋值到正整数($1 \sim (2^{n-1}-1)$)
近一半赋值到负整数($(-(2^{n-1}-1)) \sim (-1)$)
- 还剩下两个值：表示0

正整数

同无符号数– 最高位为0

00101 = 5

负整数

对于原码来说，将最高位设为1代表负数，其他比特同无符号数一样

10101 = -5

总结

萌

• 有符号数的表现格式 – 补码

原码(sign-magnitude)有什么问题？

- 0有两种重复表示 (+0 and -0)
- 计算电路复杂
- 对负数做加法时，实际上需要减法操作
- 需要考虑减法中的借位操作

00101	(5)	01001	(9)
+ 11011	(-5)	+ 10111	(-9)
00000	(0)	00000	(0)

2的补码表示方法可以让计算电路更简单

- 对于每个正数 X ，保证其相反数 $(-X)$ 满足 $X + (-X) = 0$ ，其中的加法为忽略最高位进位的普通加法

- 有符号数的表现格式 – 补码

若数字为正数或是0

- 正常二进制表示方法

若数字为负数

- 写出和它互为相反数的那个正数
- 翻转每一个比特
- 最后加1

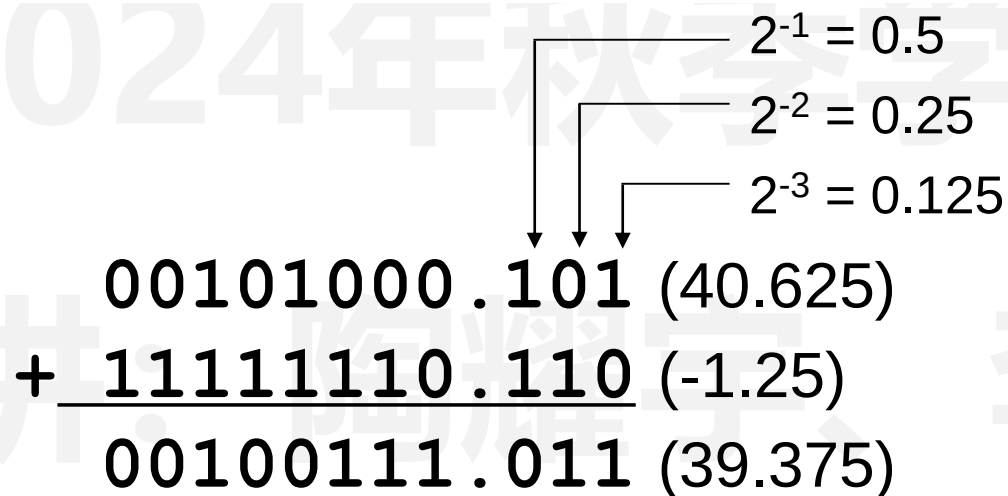
00101	(5)	01001	(9)
11010	(1's comp)	10110	(1's comp)
+ 1		+ 1	
11011	(-5)	10111	(-9)

- 定点数 – Fixed-point

如何表示分数？

- 使用二进制小数点来分开2的正数次幂和负数次幂（同十进制相似）
- 2的补码加法和减法依然成立

➤ 前提时小数点对齐


$$\begin{array}{r} 00101000.101 \quad (40.625) \\ + 11111110.110 \quad (-1.25) \\ \hline 00100111.011 \quad (39.375) \end{array}$$

$2^{-1} = 0.5$
 $2^{-2} = 0.25$
 $2^{-3} = 0.125$

No new operations -- same as integer arithmetic.

- 特别大和特别小的数：浮点数 – Floating-point

Large values: 6.023×10^{23} -- requires 79 bits

Small values: 6.626×10^{-34} -- requires >110 bits

使用科学计数法的等效： $F \times 2^E$

需要表示分数F (fraction), 指数E (exponent), and 符号位S(sign).

IEEE 754 浮点数标准(32-bits):



$$N = -1^S \times 1.\text{fraction} \times 2^{\text{exponent}-127}, \quad 1 \leq \text{exponent} \leq 254$$

$$N = -1^S \times 0.\text{fraction} \times 2^{-126}, \quad \text{exponent} = 0$$

- Single-precision IEEE floating point number:
10111110100000000000000000000000
- ↑ ↑ ↑
sign *exponent* *fraction*

- Sign is 1 – number is negative.
- Exponent field is 01111110 = 126 (decimal).
- Fraction is 0.100000000000... = 0.5 (decimal).

Value = $-1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75$.

- 卷积 – AI计算中最常用的算子

卷积最初是一种信号处理滤波操作 –

“AI神经网络也可看作是一种滤波”

WinoGrad算法起源于1980年，
是Shmuel Winograd提出用来减少FIR滤波器计算量的一个算法

输入 $d = [d_0, d_1, d_2, d_3]$ ，卷积核 $g = [g_0, g_1, g_2]$

1维卷积可表达为

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \end{bmatrix}$$

输出尺寸 卷积核尺寸 6次乘法

- 卷积 – AI计算中最常用的算子

输入 $d = [d_0, d_1, d_2, d_3]$, 卷积核 $g = [g_0, g_1, g_2]$

1维卷积可表达为

Winograd

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} r_0 \\ r_1 \end{bmatrix} \quad \rightarrow \quad F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

存在大量重复元素

$$\begin{aligned} m_1 &= (d_0 - d_2)g_0 & m_2 &= (d_1 + d_2) \frac{g_0 + g_1 + g_2}{2} \\ m_4 &= (d_1 - d_3)g_2 & m_3 &= (d_2 - d_1) \frac{g_0 - g_1 + g_2}{2} \end{aligned}$$

预算好一次g的加减后可重复复用 -> 4次乘法

- 卷积 – AI计算中最常用的算子

[element-wise multiplication](#) ([Hadamard product](#))

输入 $d = [d_0, d_1, d_2, d_3]$, 卷积核 $g = [g_0, g_1, g_2]$

1维卷积可表达为

$$F(2,3) = \begin{bmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{bmatrix} \begin{bmatrix} g_0 \\ g_1 \\ g_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

$$m_1 = (d_0 - d_2)g_0 \quad m_2 = (d_1 + d_2) \frac{g_0 + g_1 + g_2}{2}$$

$$m_4 = (d_1 - d_3)g_2 \quad m_3 = (d_2 - d_1) \frac{g_0 - g_1 + g_2}{2}$$

1D Winograd

$$Y = A^T [(Gg) \odot (B^T d)]$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{bmatrix}$$

$$g = [g_0 \quad g_1 \quad g_2]^T$$

$$d = [d_0 \quad d_1 \quad d_2 \quad d_3]^T$$

• 卷积 – AI计算中最常用的算子

将一维卷积运算定义为 $F(m, r)$, m 为Output Size, r 为Filter Size, 则输入信号的长度为 $m + r - 1$, 卷积运算是对应位置相乘然后求和, **输入信号每个位置至少要参与1次乘法**, 所以乘法数量最少与输入信号长度相同, 记为

$$\mu(F(m, r)) = m + r - 1$$

在行列上分别进行一维卷积运算, 可得到二维卷积, 记为 $F(m \times n, r \times s)$, 输出为 $m \times n$, 卷积核为 $r \times s$, 则输入信号为 $(m + r - 1)(n + s - 1)$, 乘法数量至少为

$$\begin{aligned}\mu(F(m \times n, r \times s)) &= \mu(F(m, r))\mu(F(n, s)) \\ &= (m + r - 1)(n + s - 1)\end{aligned}$$

若是直接按滑动窗口方式计算卷积, 一维时需要 $m \times r$ 次乘法, 二维时需要 $m \times n \times r \times s$ 次乘法, **远大于上面计算的最少乘法次数**。

使用Winograd算法计算卷积快在哪里? 一言以蔽之: **快在减少了乘法的数量**, 将乘法数量减少至 $m + r - 1$ 或 $(m + r - 1)(n + s - 1)$ 。

- 卷积 – AI计算中最常用的算子

$$Y = A^T [(Gg) \odot (B^T d)]$$

g : 表示卷积核

d : 表示输入信号

G : 卷积核变换矩阵, 尺寸为 $(m + r - 1) \times r$

B^T : 输入变换矩阵, 尺寸 $(m + r - 1) \times (m + r - 1)$

A^T : 输出变换矩阵, 尺寸 $m \times (m + r - 1)$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$g = [g_0 \quad g_1 \quad g_2]^T$$

$$d = [d_0 \quad d_1 \quad d_2 \quad d_3]^T$$

计算过程可分为4步:

(1) 输入变换

(2) 卷积核变换

1D Winograd

(3) 外积

(4) 输出变换

• 卷积 – AI计算中最常用的算子

$$Y = A^T [(Gg) \odot (B^T d)]$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$$

$$G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

$$g = [g_0 \ g_1 \ g_2]^T$$

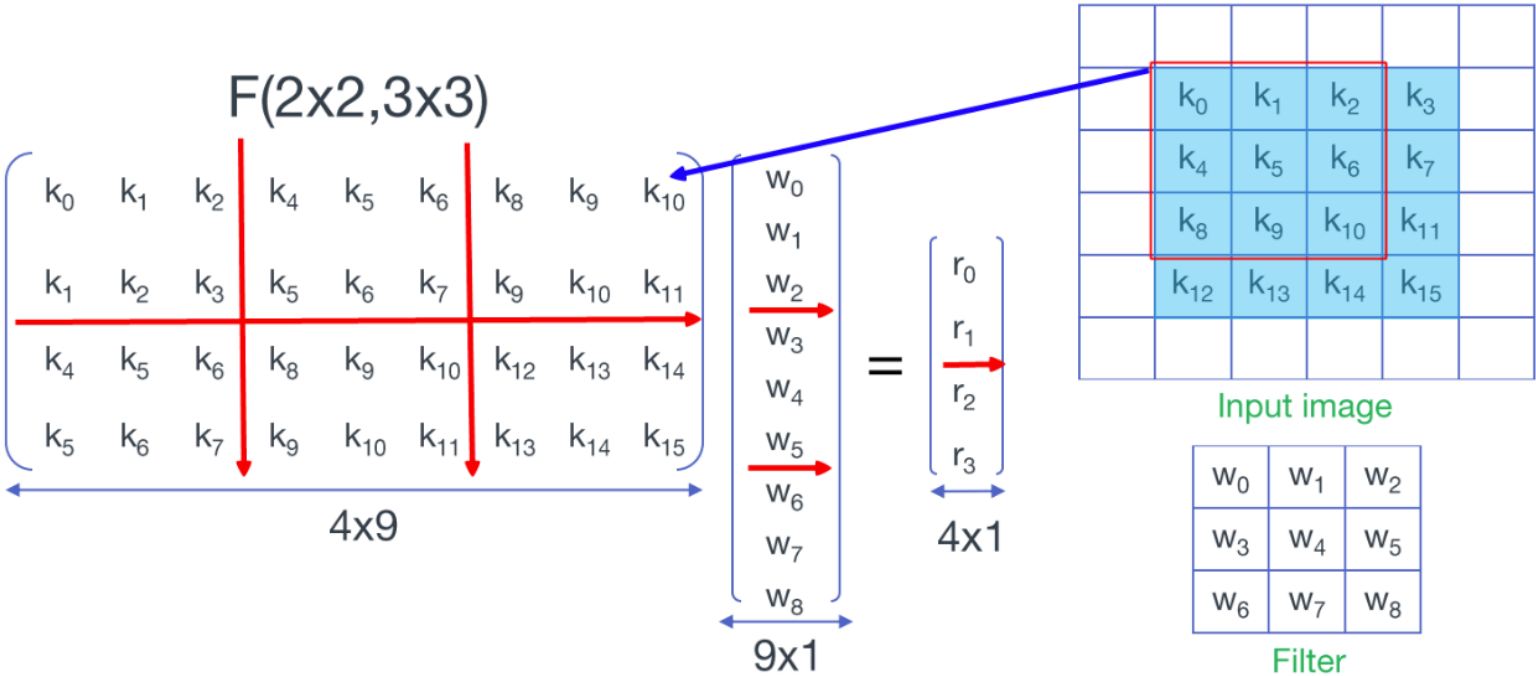
$$d = [d_0 \ d_1 \ d_2 \ d_3]^T$$

2D Winograd

A minimal 1D algorithm $F(m, r)$ is **nested with itself** to obtain a minimal 2D algorithm $(m \times m, r \times r)$.

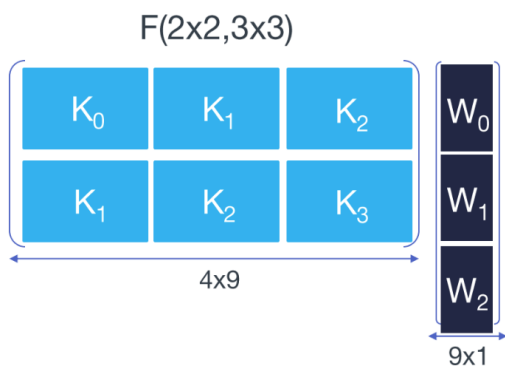
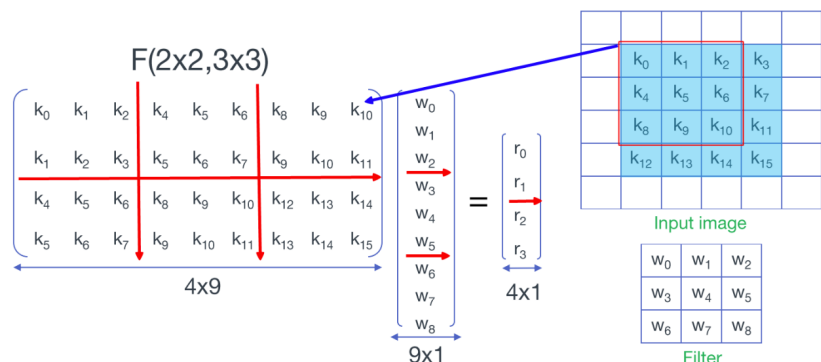
$$Y = A^T [[GgG^T] \odot [B^T dB]] A$$

g 为 $r \times r$ Filter, d 为 $(m + r - 1) \times (m + r - 1)$



• 卷积 – AI计算中最常用的算子

令 $D_0 = [k_0, k_1, k_2, k_3]^T$, 即窗口中的第0行元素, $D_1 D_2 D_3$ 表示第1、2、3行; $W_0 = [w_0, w_1, w_2]^T$,



$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{bmatrix}$$

$$= \begin{bmatrix} R_0 \\ R_1 \end{bmatrix} = \begin{bmatrix} K_0 W_0 + K_1 W_1 + K_2 W_2 \\ K_1 W_0 + K_2 W_1 + K_3 W_2 \end{bmatrix}$$

$$= \begin{bmatrix} A^T [(GW_0) \odot (B^T D_0)] + A^T [(GW_1) \odot (B^T D_1)] + A^T [(GW_2) \odot (B^T D_2)] \\ A^T [(GW_0) \odot (B^T D_1)] + A^T [(GW_1) \odot (B^T D_2)] + A^T [(GW_2) \odot (B^T D_3)] \end{bmatrix}$$

$$= A^T [[G[W_0 W_1 W_2]G^T] \odot [B^T[d_0 d_1 d_2 d_3]B]] A$$

$$= A^T [[GgG^T] \odot [B^T dB]] A$$

$$\begin{bmatrix} K_0 & K_1 & K_2 \\ K_1 & K_2 & K_3 \end{bmatrix} \begin{bmatrix} W_0 \\ W_1 \\ W_2 \end{bmatrix} = \begin{bmatrix} R_0 \\ R_1 \end{bmatrix} = \begin{bmatrix} M_0 + M_1 + M_2 \\ M_1 - M_2 - M_3 \end{bmatrix}$$

Matrix multiply F(2,3)! 4 multiplications

$$M_0 = (K_0 - K_2) \cdot W_0$$

$$M_1 = (K_1 + K_2) \cdot \frac{W_0 + W_1 + W_2}{2}$$

$$M_3 = (K_1 - K_3) \cdot W_2$$

$$M_2 = (K_2 - K_1) \cdot \frac{W_0 - W_1 + W_2}{2}$$

卷积加速模块设计

• NVDLA中的Winograd卷积核设计

在MAC并行计算单元中计算

$$A^T \left[\left[GgG^T \right] \odot \left[B^T dB \right] \right] A$$

预先算好

在输入datapath计算

For m = 3, r = 2

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{bmatrix}, G = \begin{bmatrix} 1 & 0 \\ \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} \\ 0 & 1 \end{bmatrix}$$
$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

For m = 4, r = 3

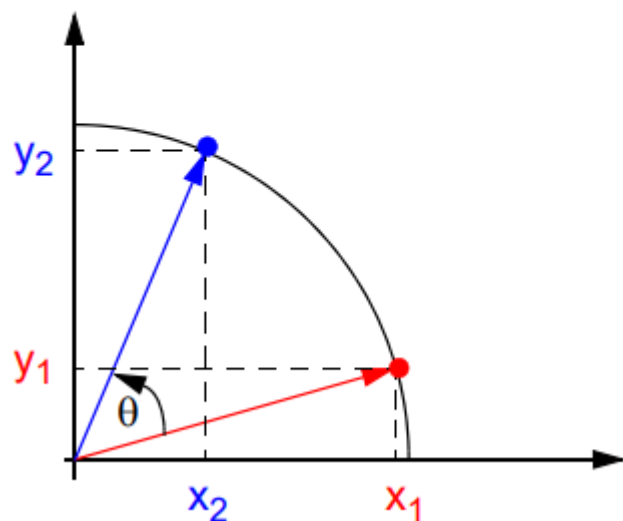
$$B^T = \begin{bmatrix} 4 & 0 & -5 & 0 & 1 & 0 \\ 0 & -4 & -4 & 1 & 1 & 0 \\ 0 & 4 & -4 & -1 & 1 & 0 \\ 0 & -2 & -1 & 2 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 & 0 \\ 0 & 4 & 0 & -5 & 0 & 1 \end{bmatrix}$$
$$G = \begin{bmatrix} \frac{1}{4} & 0 & 0 \\ -\frac{1}{6} & -\frac{1}{6} & -\frac{1}{6} \\ -\frac{1}{6} & \frac{1}{6} & -\frac{1}{6} \\ \frac{1}{24} & \frac{1}{12} & \frac{1}{6} \\ \frac{1}{24} & -\frac{1}{12} & \frac{1}{6} \\ 0 & 0 & 1 \end{bmatrix}$$
$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & 2 & -2 & 0 \\ 0 & 1 & 1 & 4 & 4 & 0 \\ 0 & 1 & -1 & 8 & -8 & 1 \end{bmatrix}$$

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 $\cos \theta$ （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

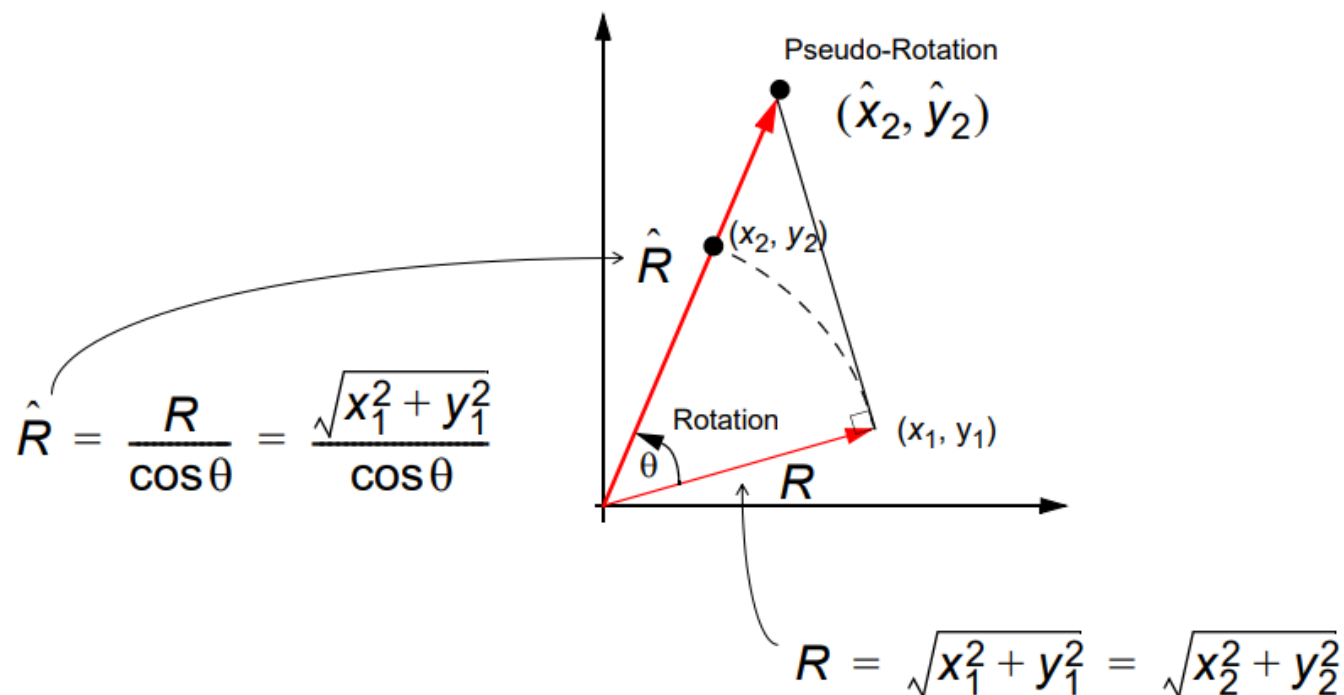
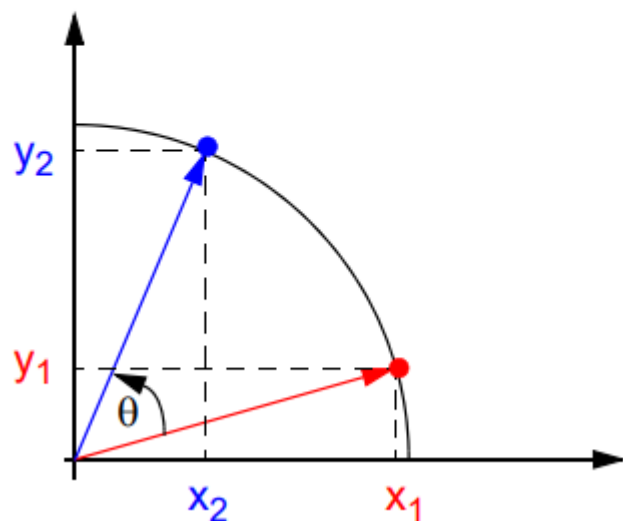
伪旋转 (pseudo rotations)

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

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

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

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



• CORDIC可以用来实现多种复杂非线性函数

伪旋转 (pseudo rotations)

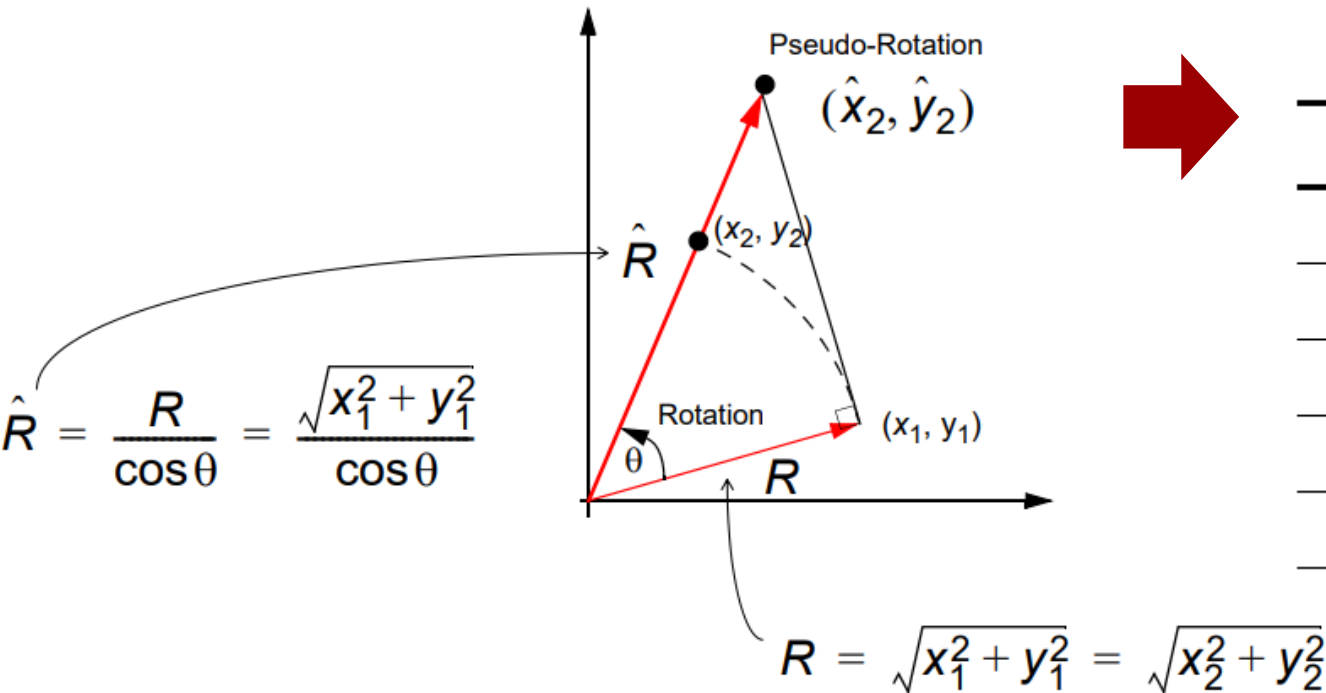
$$\hat{x}_2 = \cancel{\cos \theta}(x_1 - y_1 \tan \theta) = x_1 - y_1 \tan \theta$$

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

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	$\tan\theta$	Angle, θ	$\cos\theta$
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

13次伪旋转后，需要乘以
 $1/0.607252941 =$
 1.6467602

$$\cos 45 \times \cos 26.5 \times \cos 14.03 \times \cos 7.125 \dots \times \cos 0.0139 = 0.607252941$$

- 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 $d_i = +/- 1$

2 shifts

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

3 additions

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

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

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

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

Scaling Factor

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

where $d_i = +/- 1$

2 shifts

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

3 additions

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

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

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

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

n = number of iterations

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

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

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

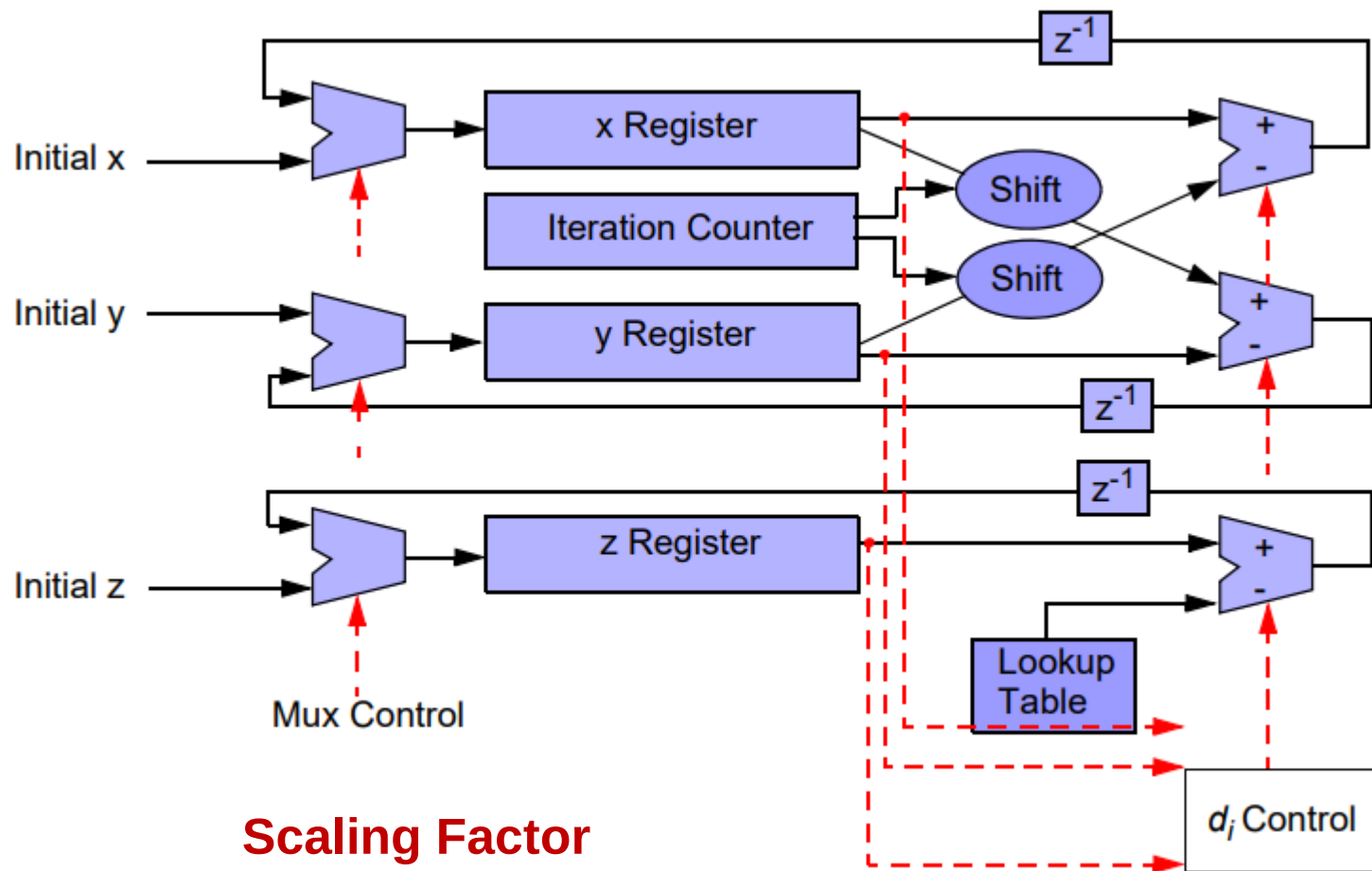
$$z^{(i+1)} = z^{(i)} - d_i\theta^{(i)}$$

where $d_i = +/- 1$

2 shifts

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

3 additions



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

目 录

CONTENTS



01. 指令集架构基础

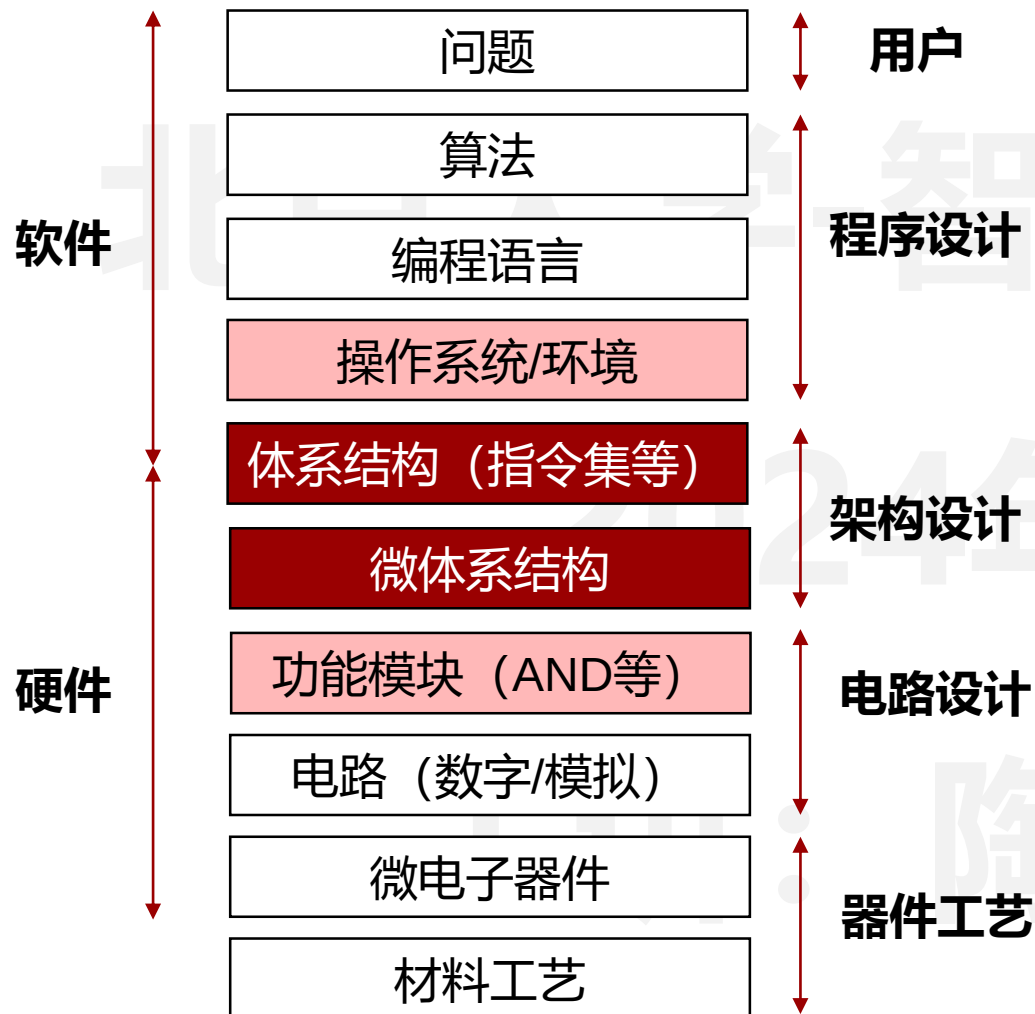
02. 指令集设计基础

03. 流水线架构基础

04. 流水线架构优化

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?

- 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  $\leftarrow$  pc+1  
    gpr[rd] = gpr[rs] + gpr[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:
 - $(\text{instructions/program}) * (\text{cycles/instruction}) * (\text{seconds/cycle})$
- 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

- 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)

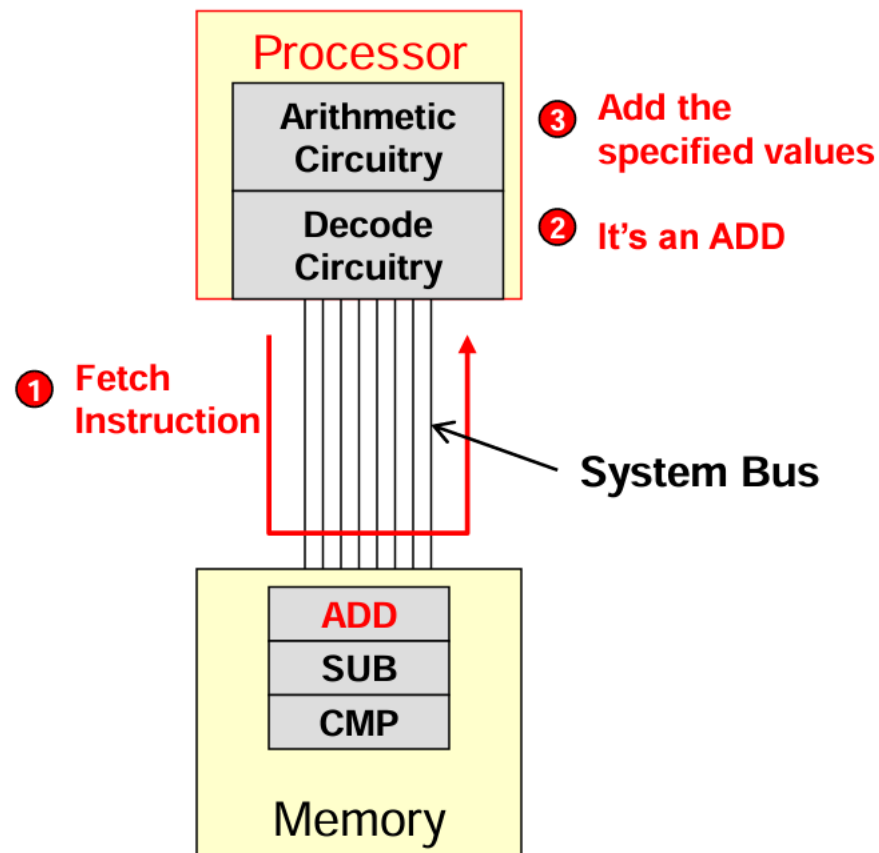
传统冯诺依曼架构的指令集

- 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
 - 此指令是ADD, SUB或是其他？
- **Decode**：解码这个指令
 - 执行特定操作
- **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**



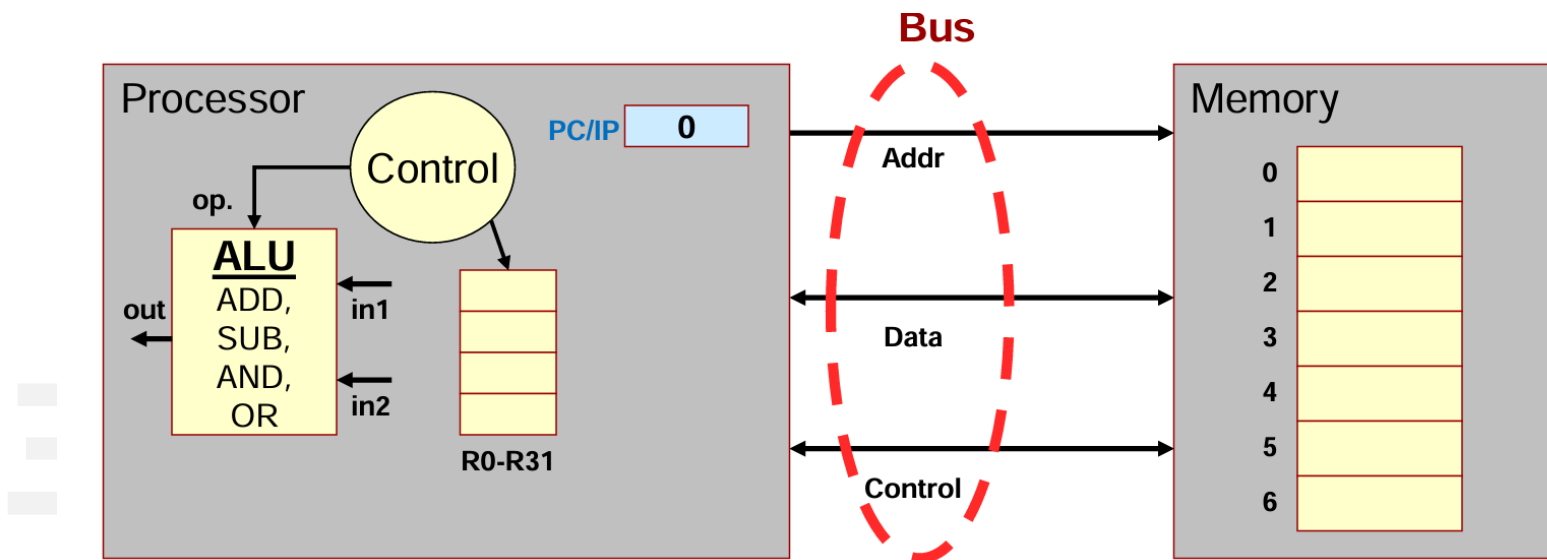
传统冯诺依曼架构的指令集

- 需要3类指令：读取、写回和运算

- 处理器中3种主要的组成

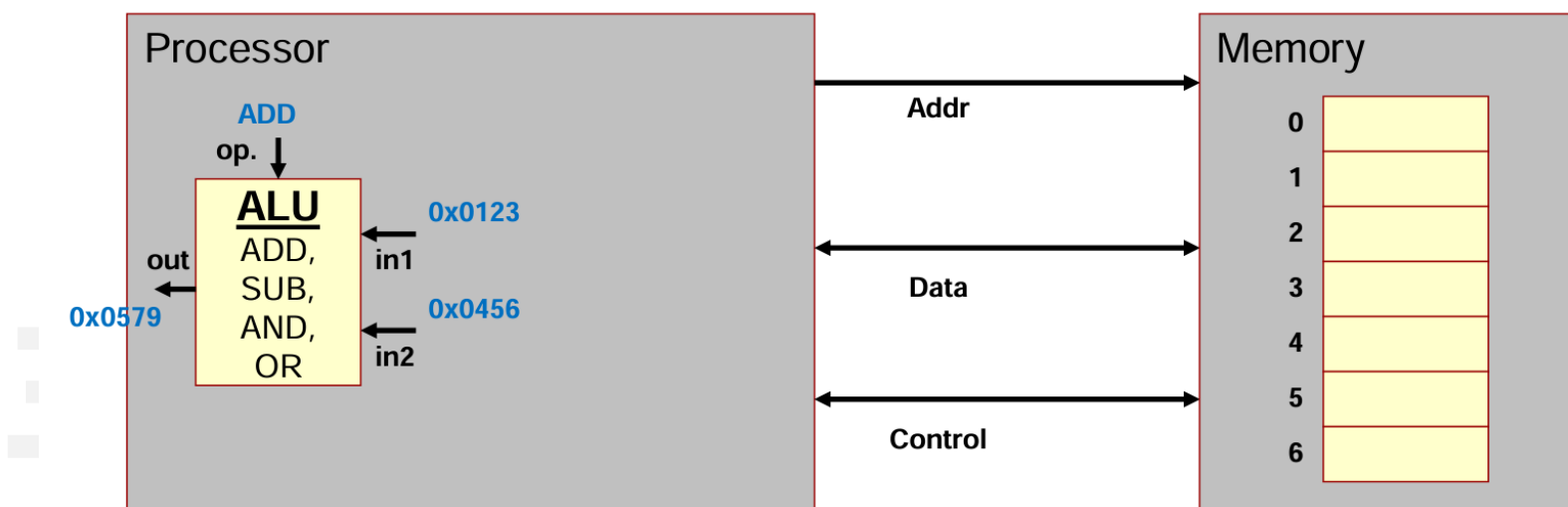
- ALU（算术逻辑单元）
- 寄存器
- 控制电路

- 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

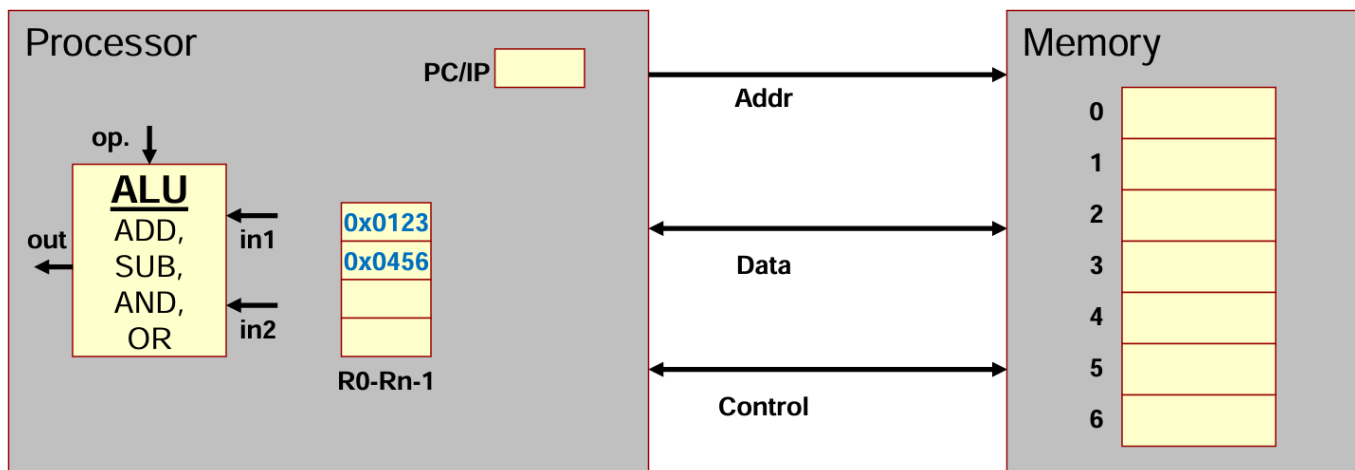


传统存算分离的指令集架构 – 核心部件1: Register

- Register负责将ALU运算结果暂存在靠近ALU的地方

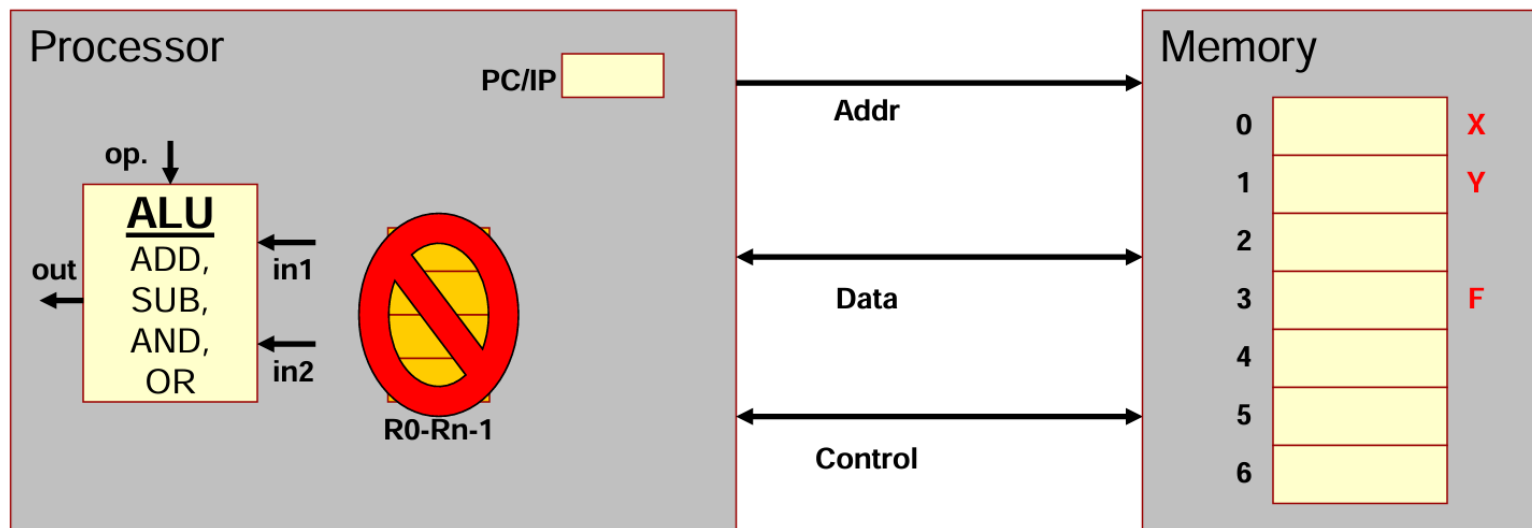
- 访问存储器的时间通常比处理器要慢
- 寄存器在处理器内部提供了快速的暂时存储位置

- 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）



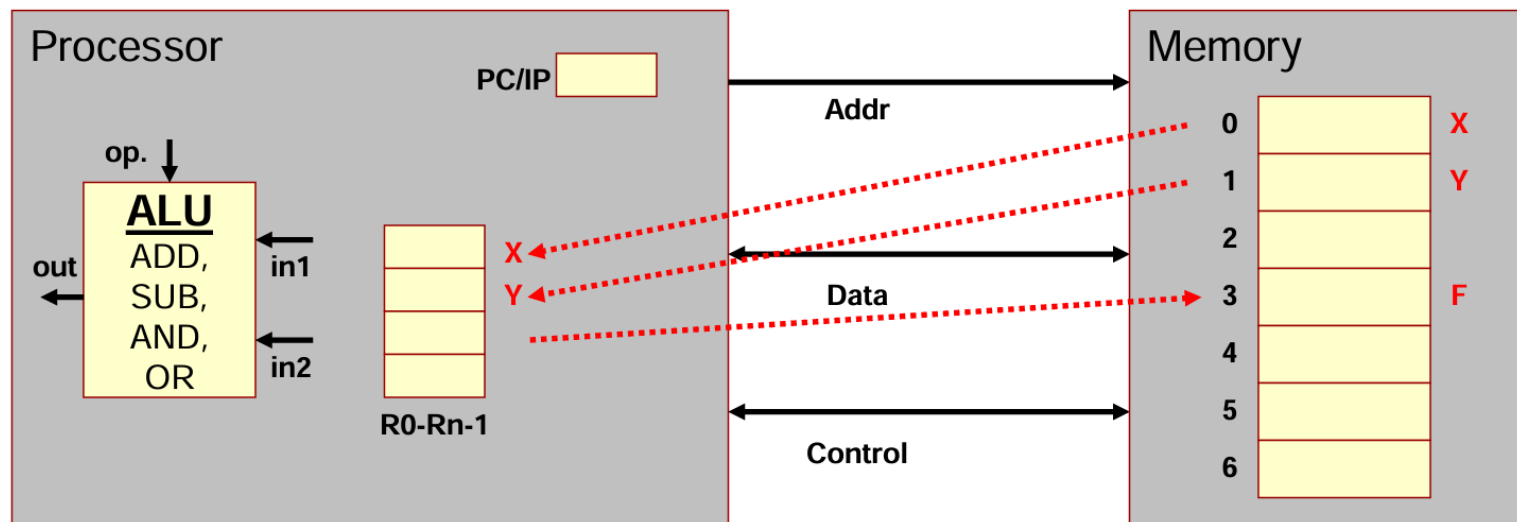
传统存算分离的指令集架构 – 核心部件1: Register

- 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



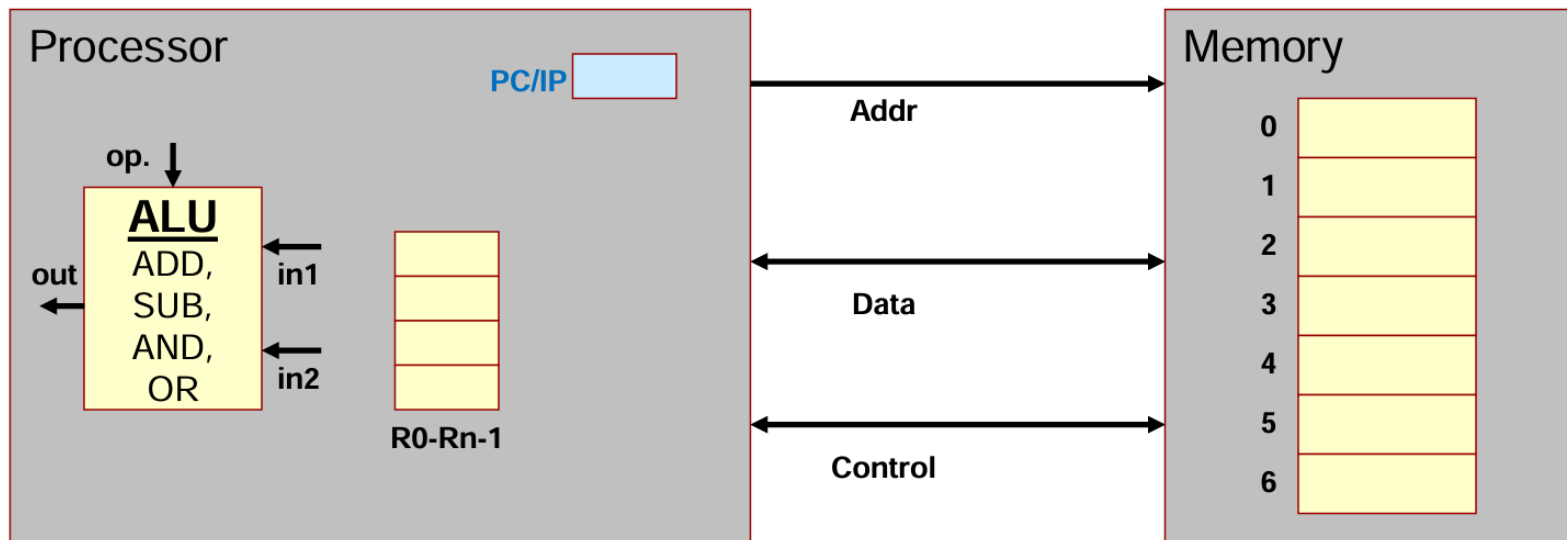
传统存算分离的指令集架构 – 核心部件1: Register

- 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



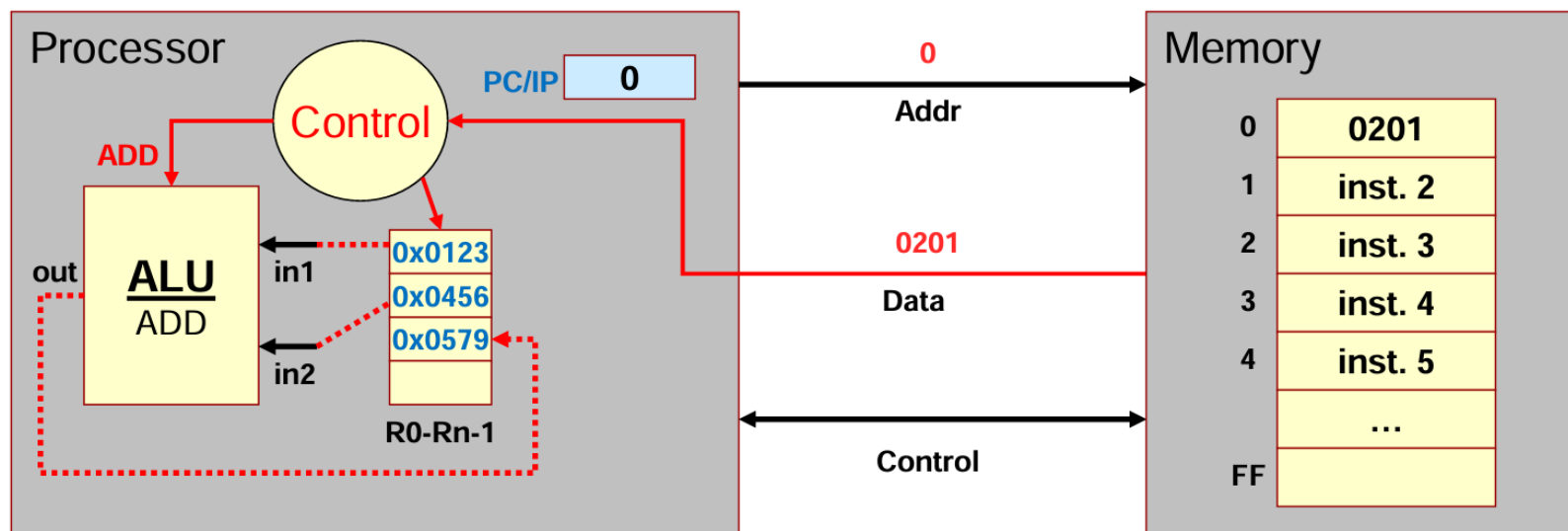
传统存算分离的指令集架构 – 核心部件1: Register

- 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

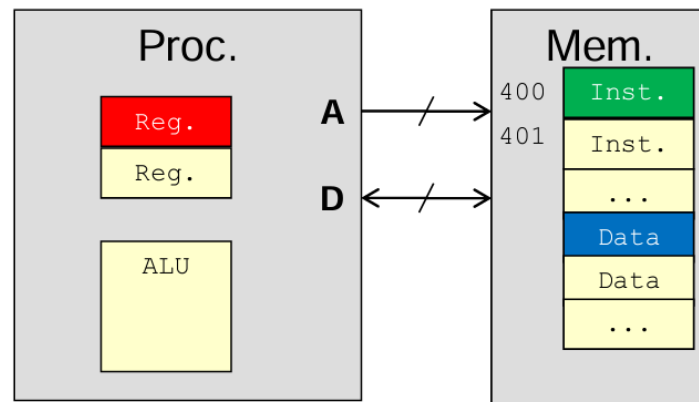
- 数据可以存储在register、主存memory或指令内部

- 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



- 01. 指令集架构基础**
- 02. 指令集设计基础**
- 03. 流水线架构基础**
- 04. 流水线架构优化**

主流指令集都有哪些？

- X86、MIPS、ARM、RISC-V

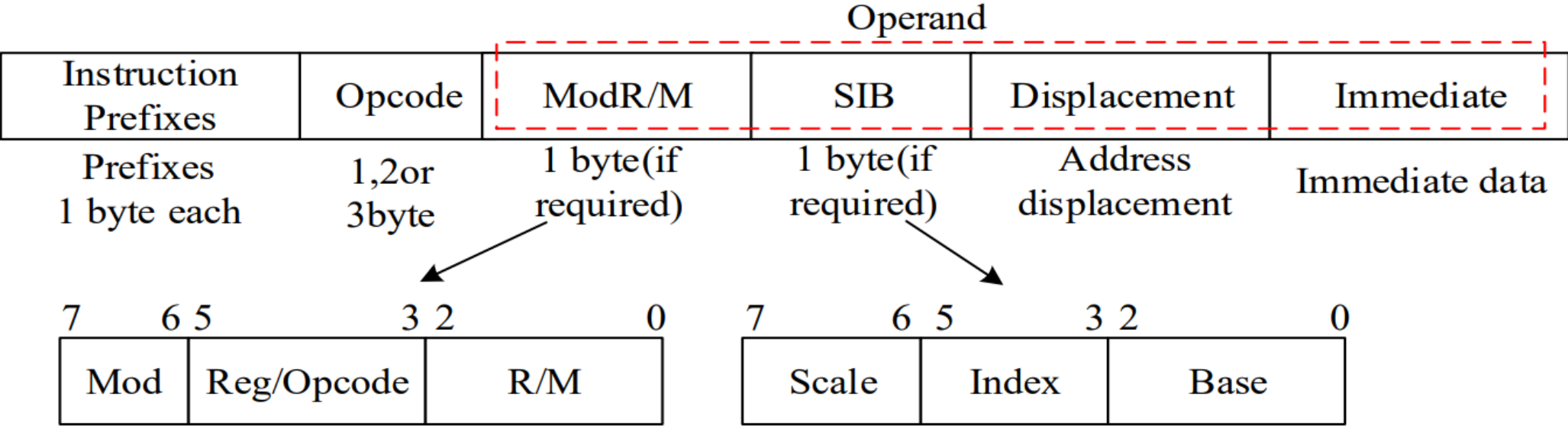
指令集	类型	运营公司	特点	代表厂商
X86	CISC	Intel、AMD	功能强大、通用性、兼容性、实用性	Intel、AMD
MIPS	RISC	MIPS	简洁、优化、高扩展性、寄存器多	Intel、IBM、龙芯、Oracle、Toshiba
ARM	RISC	ARM	低功耗、低成本、适用于移动设备	苹果、华为、谷歌
RISC-V	RISC	RISC-V基金会	完全开源、架构简单、移植性高、开源工具链	数百家大学、科研机构和企业
CUDA	RISC	Nvidia	张量高并发处理、图像处理	Nvidia

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）

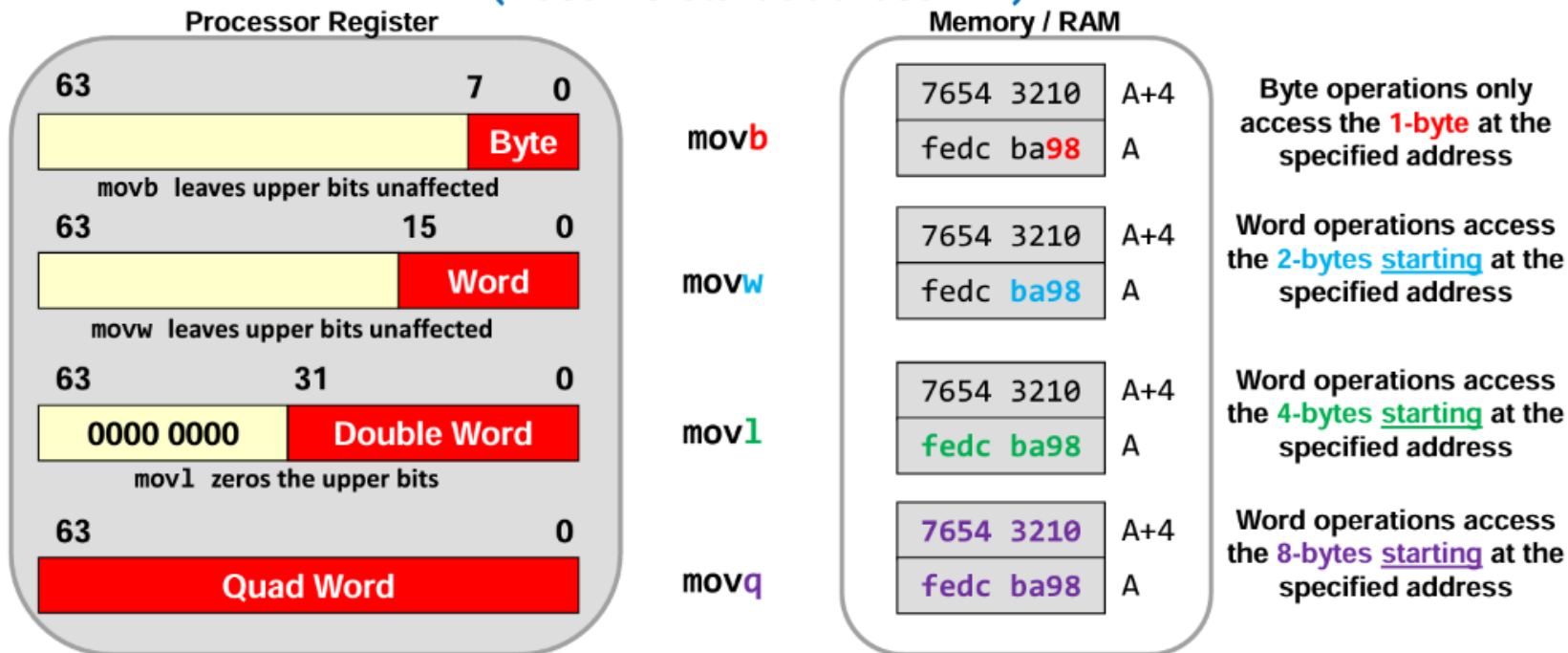


主讲：陶耀宇、李萌

指令集的分类1 – 传输指令

- 指令集可以看做链接软件和一个硬件的一个协议
 - 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

(Assume start address = A)



指令集的分类1 – 传输指令：指令的地址模式

• 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	$R[\text{rax}] = \text{imm.}$
Register	r_a	movl %rdx,%rax	$R[\text{rax}] = R[\text{rdx}]$
Direct Addressing	imm	movl 2000,%rax	$R[\text{rax}] = M[2000]$
Indirect Addressing	(r_a)	movl (%rdx),%rax	$R[\text{rax}] = M[R[r_a]]$
Base w/ Displacement	$\text{imm}(r_b)$	movl 40(%rdx),%rax	$R[\text{rax}] = M[R[r_b]+40]$
Scaled Index	$(r_b, r_i, s^\dagger)$	movl (%rdx,%rcx,4),%rax	$R[\text{rax}] = M[R[r_b]+R[r_i]*s]$
Scaled Index w/ Displacement	$\text{imm}(r_b, r_i, s^\dagger)$	movl 80(%rdx,%rcx,2),%rax	$R[\text{rax}] = M[80 + R[r_b]+R[r_i]*s]$

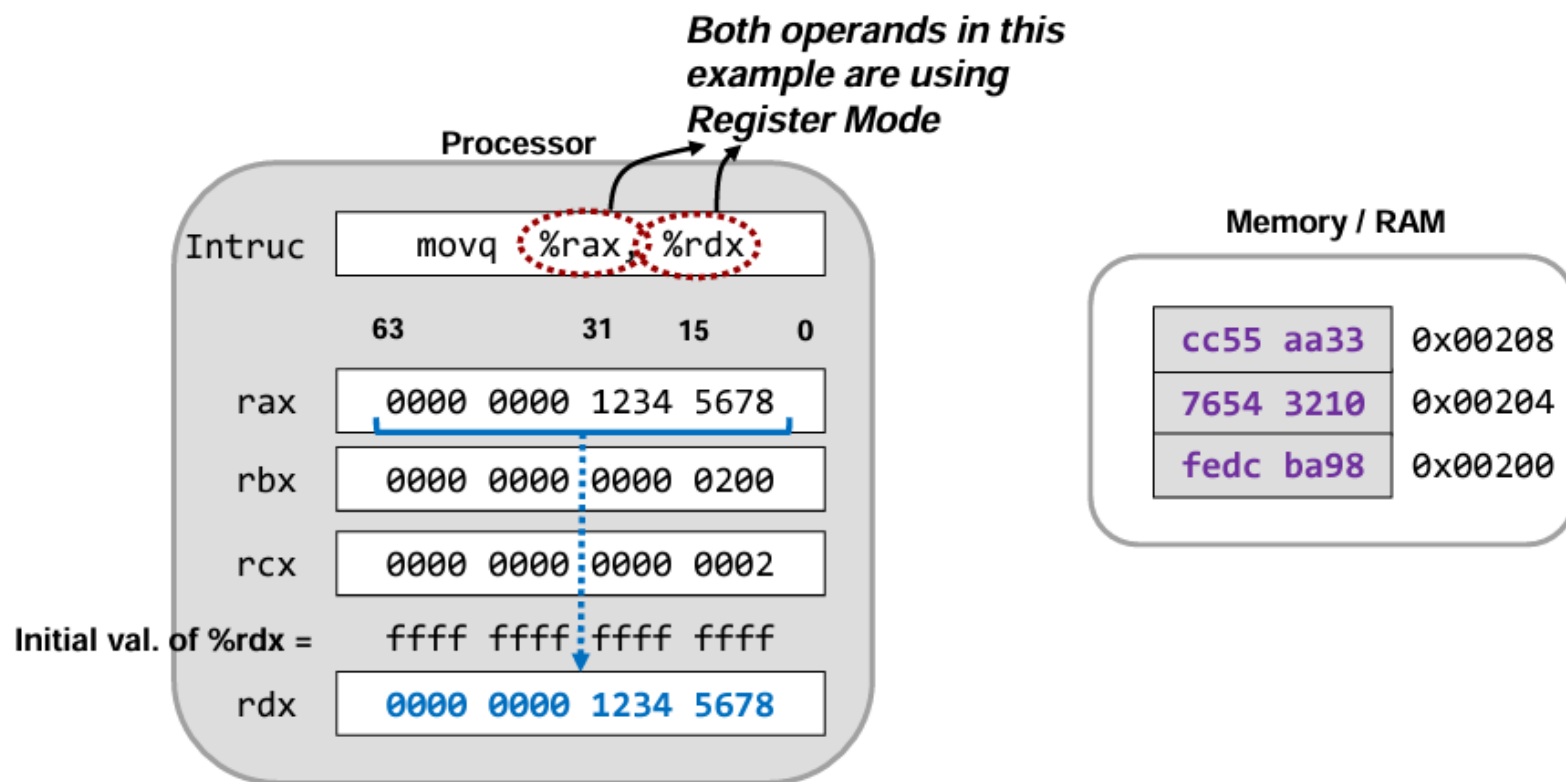
† Known as the scale factor and can be {1,2,4, or 8}

Imm = Constant, $R[x]$ = Content of register x, $M[\text{addr}]$ = Content of memory @ addr.

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

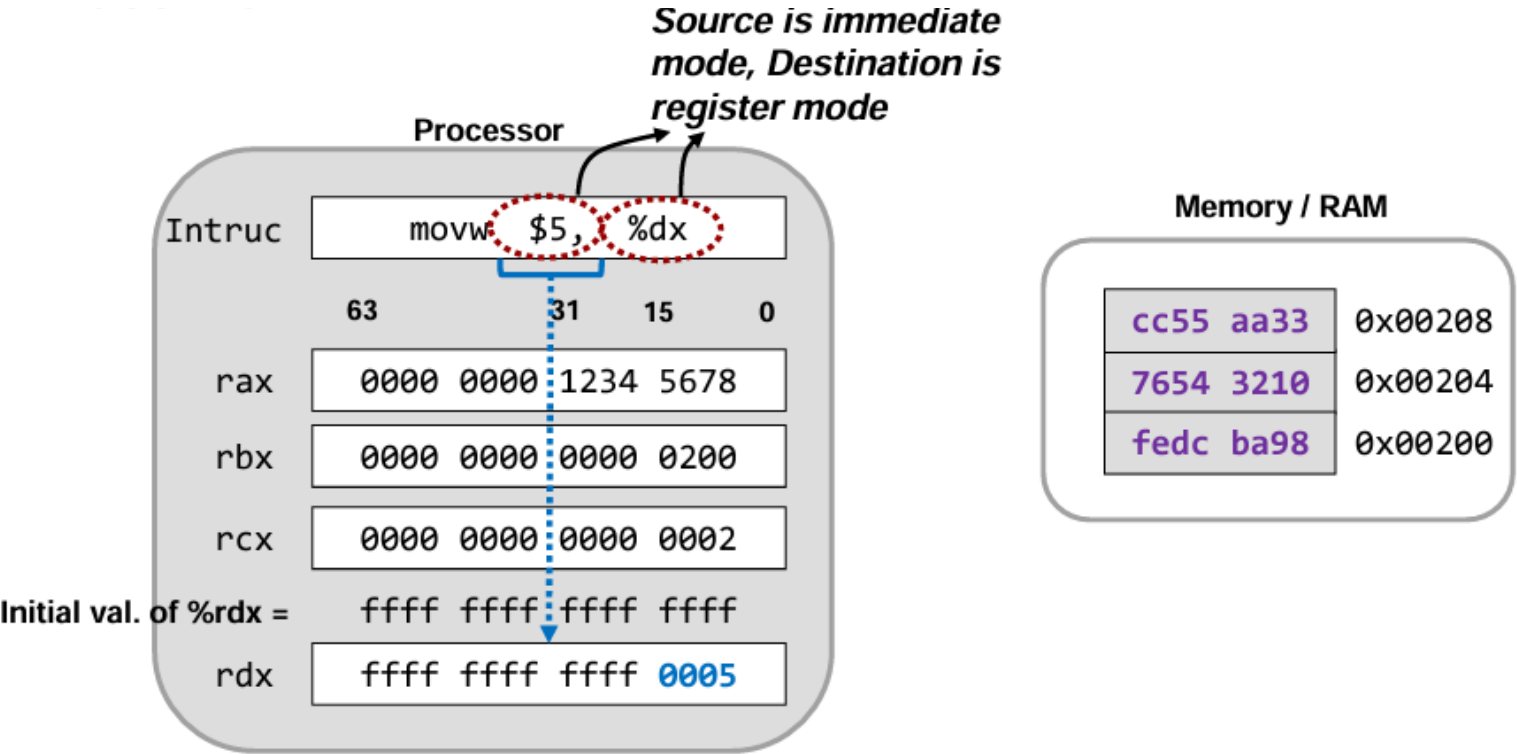
指令集的分类1 – 传输指令: Register Mode

- 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

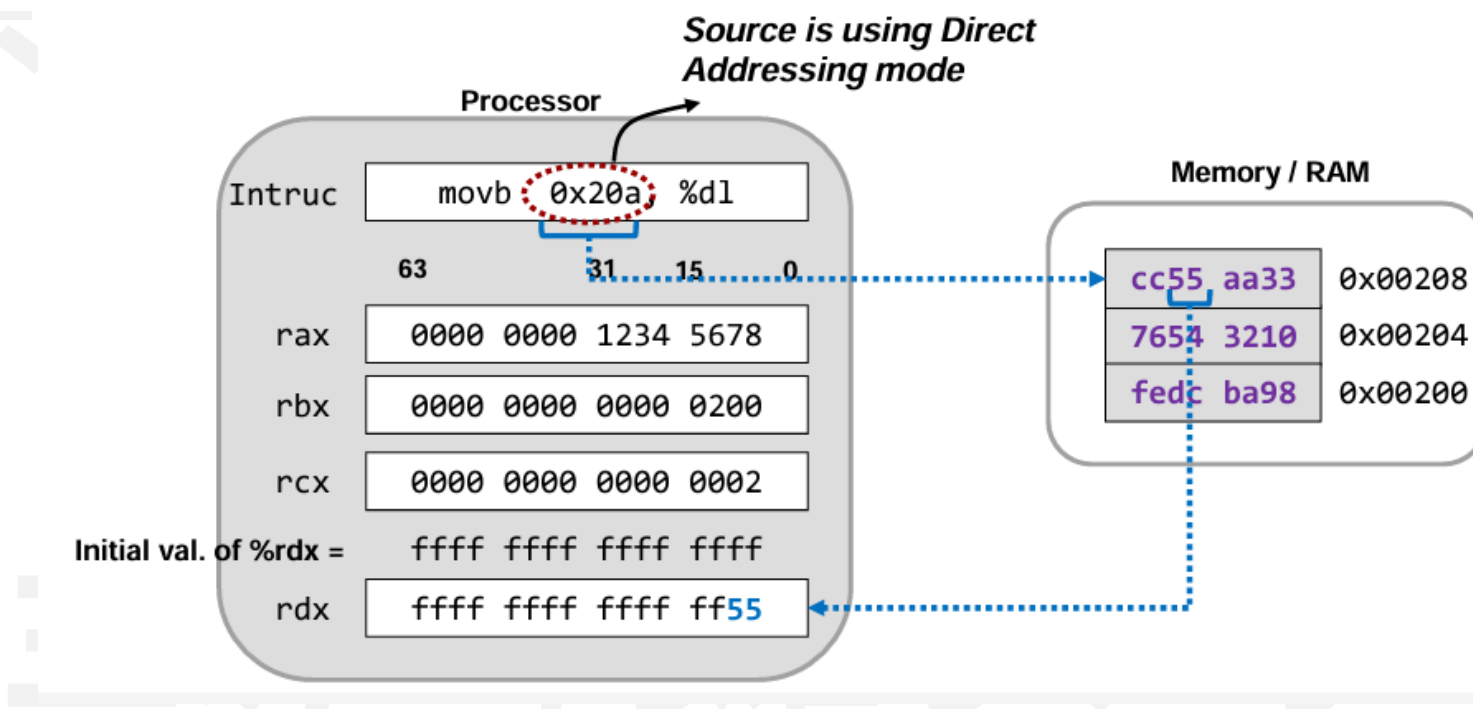
- 指定指令中的立即数作为操作数
- 符号 '\$' 表示立即数，并且可以指定用十六进制或是十进制



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



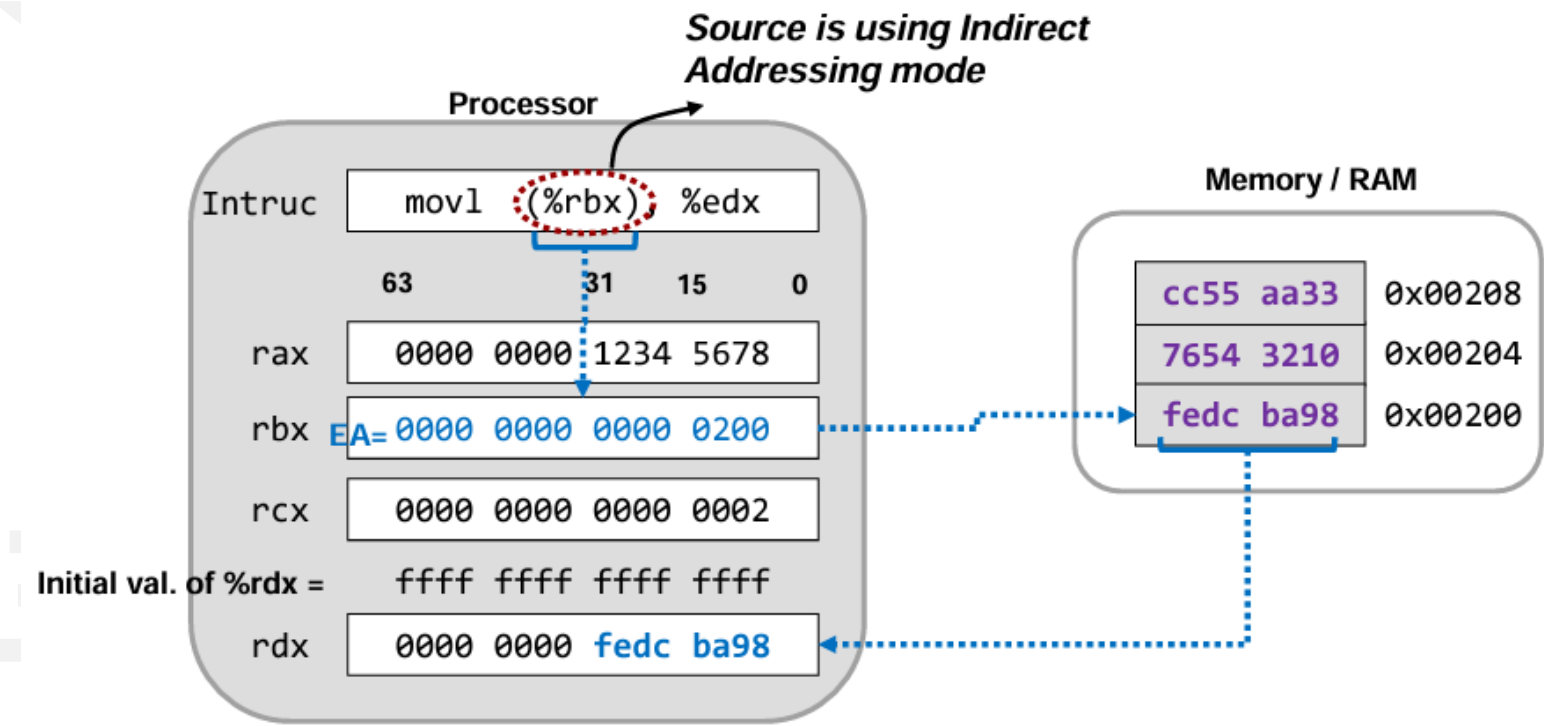
指令集的分类1 – 传输指令： Indirect Addressing Mode

- 指定存储器内容为真正操作数在存储器中的有效地址

y

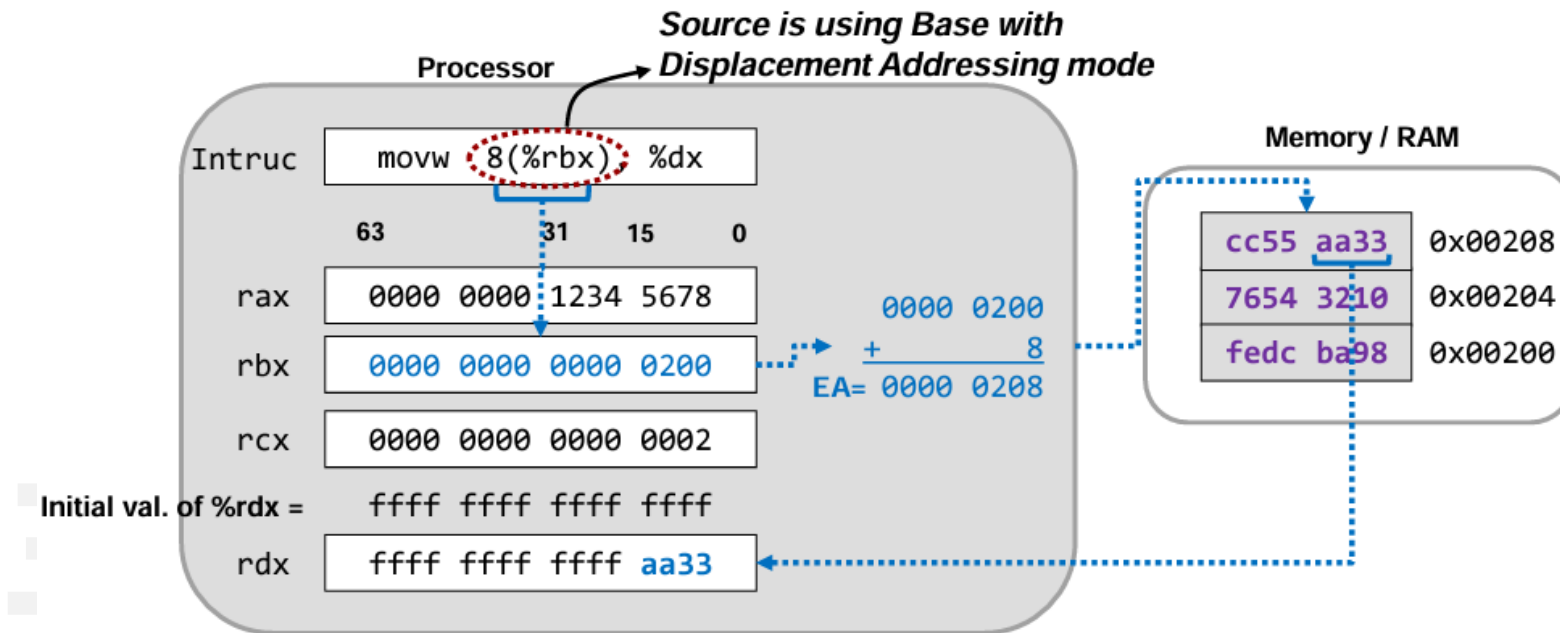
类似于指针

- 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



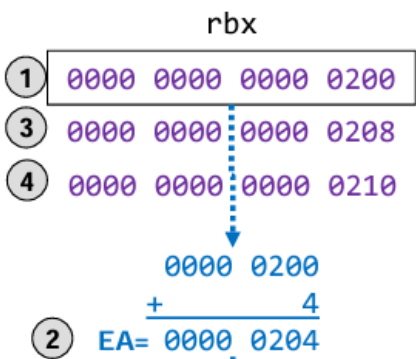
指令集的分类1 – 传输指令： Base/Indirect with Displacement Addressing Mode

- 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



Memory / RAM

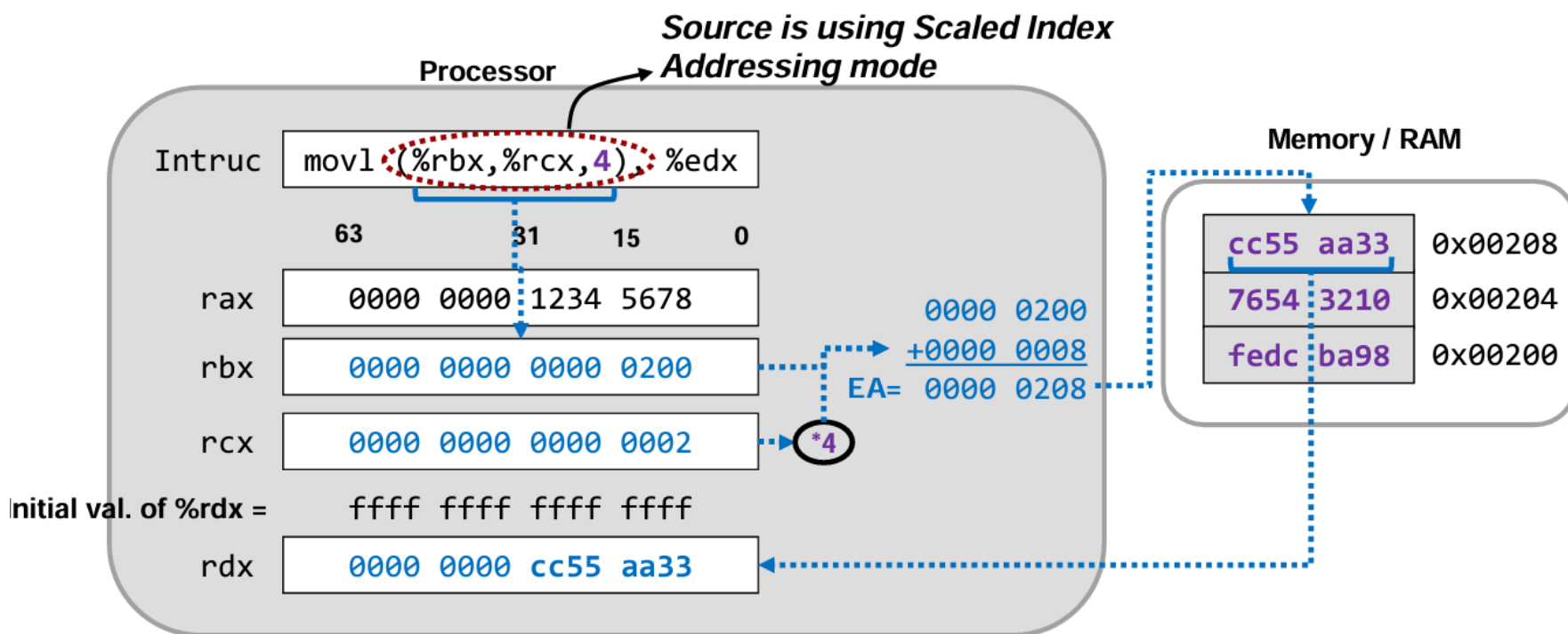
data[2].y	0000 0002	0x00214
data[2].x	0000 0001	0x00210
data[1].y	0000 0002	0x0020c
data[1].x	0000 0001	0x00208
data[0].y	0000 0002	0x00204
data[0].x	0000 0001	0x00200

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

Assembly

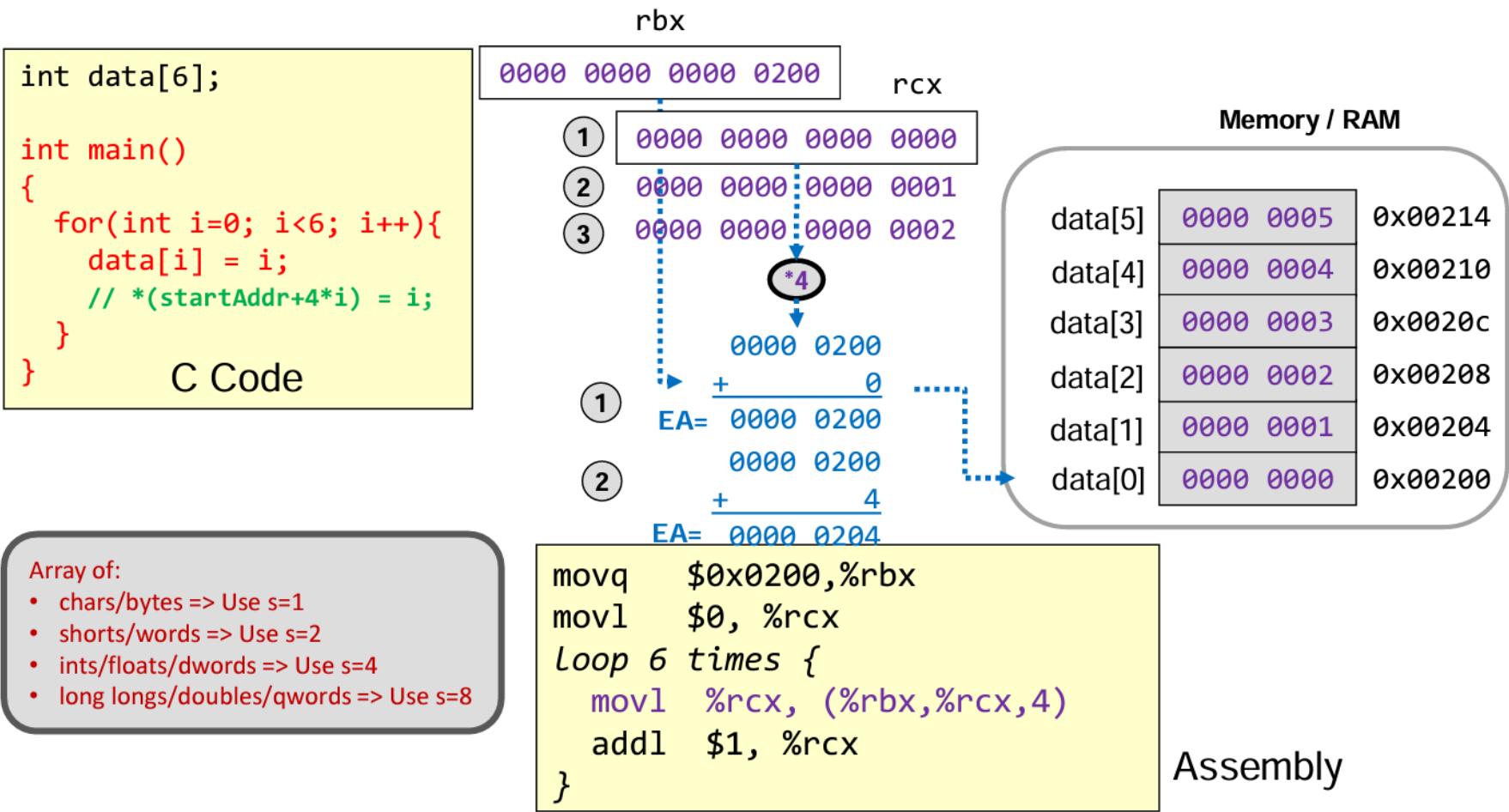
指令集的分类1 – 传输指令: Scaled Index Addressing Mode

- 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



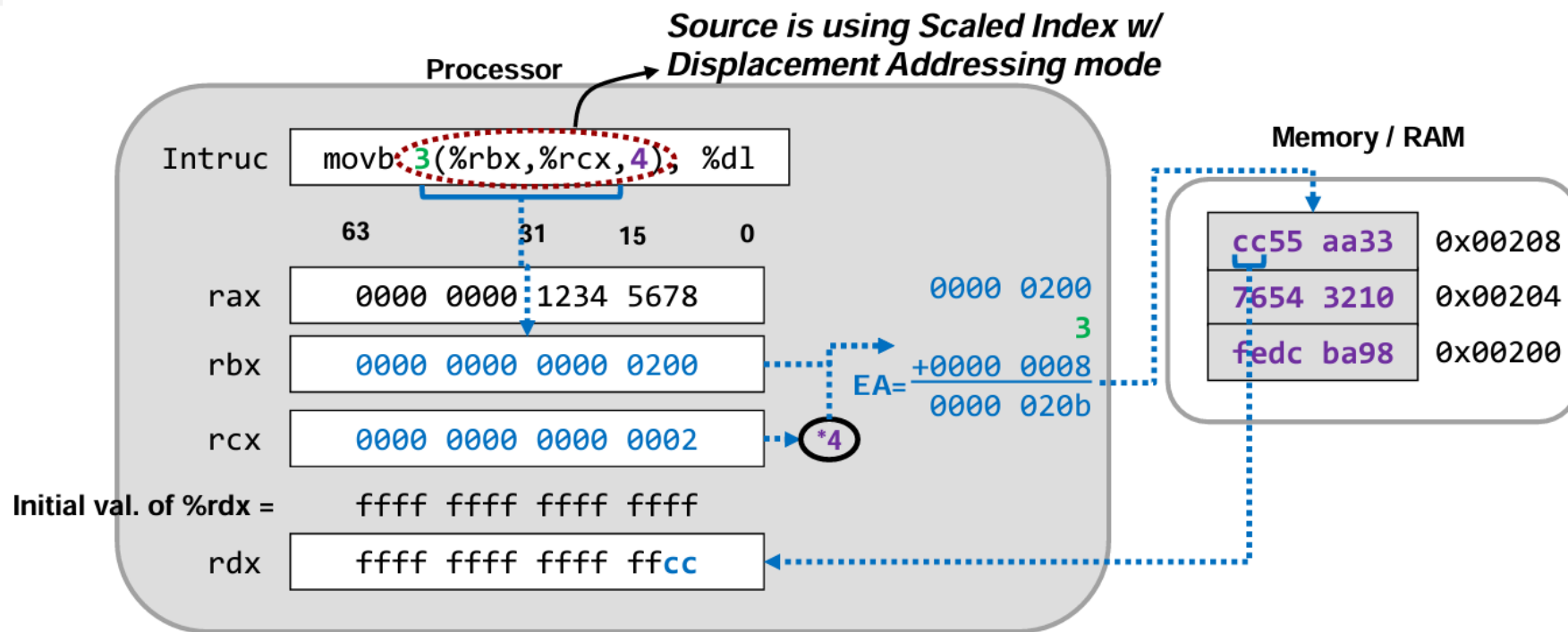
指令集的分类1 – 传输指令: Scaled Index Addressing Mode

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



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

- 集合Scale和Displacement: 地址 = $d(\%reg1, \%reg2, s)$ [$s = 1, 2, 4, \text{ or } 8$]
- 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

Processor Registers		Memory / RAM	
0000 0000 0000 0200	rbx	cdef 89ab	0x00204
0000 0000 0000 0003	rcx	7654 3210	0x00200
		f00d face	0x001fc
		dead beef	0x001f8

– movq (%rbx), %rax	cdef 89ab 7654 3210	rax
– movl -4(%rbx), %eax	0000 0000 f00d face	rax
– movb (%rbx,%rcx), %al	0000 0000 f00d fa76	rax
– movw (%rbx,%rcx,2), %ax	0000 0000 f00d cdef	rax
– movsbl -16(%rbx,%rcx,4), %eax	0000 0000 ffff ffce	rax
– movw %cx, 0xe0(%rbx,%rcx,2)	0000 0000	0x002e8
	0003 0000	0x002e4

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

Initial Conditions

- 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

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

Memory / RAM	
7654 3210	0x00204
0f0f ff00	0x00200
Processor Registers	
ffff ffff 1234 5678	rdx
0000 0000 cc33 aa55	rax
0000 0000 cc34 cd55	rax
ffff ffff de69 23cd	rax
ffff ffff de69 2300	rax
ffff ffff de69 230f	rax
ffff ffff de69 2301	rax
7655 5555	0x00204
0f0f ff00	0x00200

主讲：陶耀宇、李萌

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

- 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

If(condition 0)

XXXXXX
XXXXXX
XXXXXX
XXXXXX

else

XXXXXX
XXXXXX

while(condition 1)

XXXXXX
XXXXXX
XXXXXX
XXXXXX

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

then continue the code.

Jump to address 497000

- 与上层操作系统OS对接，例如OS可更改register内容等

北京大学-智能硬件体系结构

编译器设计内容

2024年秋季学期

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
imm _{11:0}				rs1	funct3	rd	op
imm _{11:5}		rs2		rs1	funct3	imm _{4:0}	op
imm _{12,10:5}		rs2		rs1	funct3	imm _{4:1,11}	op
imm _{31:12}						rd	op
imm _{20,10:1,11,19:12}						rd	op
fs3	funct2	fs2	fs1	funct3		fd	op
5 bits	2 bits	5 bits	5 bits	3 bits		5 bits	7 bits

R-Type
I-Type
S-Type
B-Type
U-Type
J-Type
R4-Type

- 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

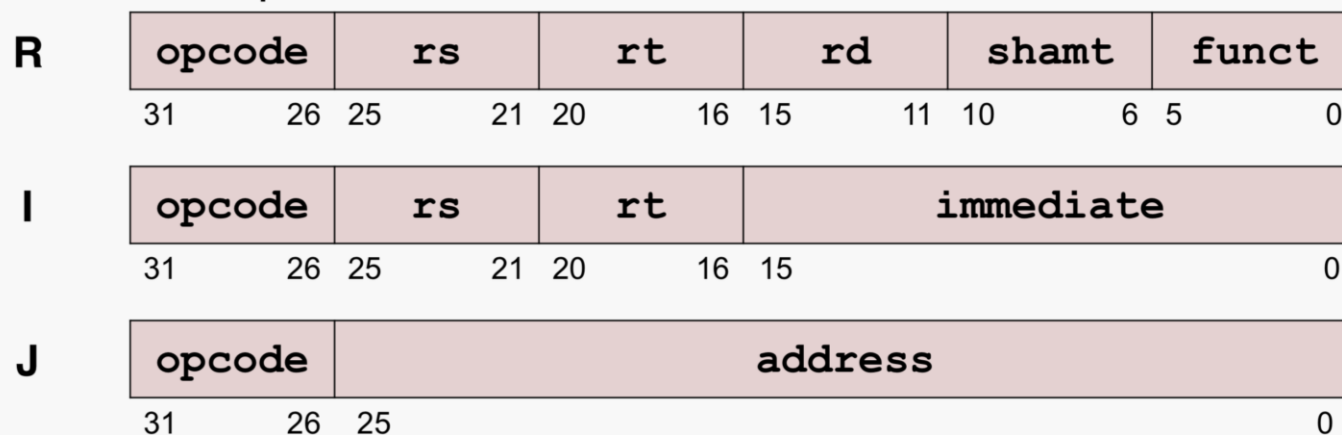
Figure B.1 RISC-V 32-bit instruction formats

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)

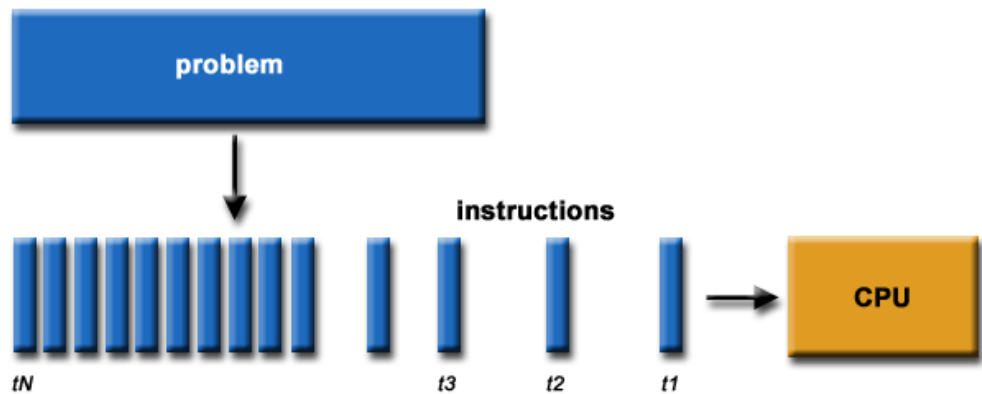
• Opcode

- 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

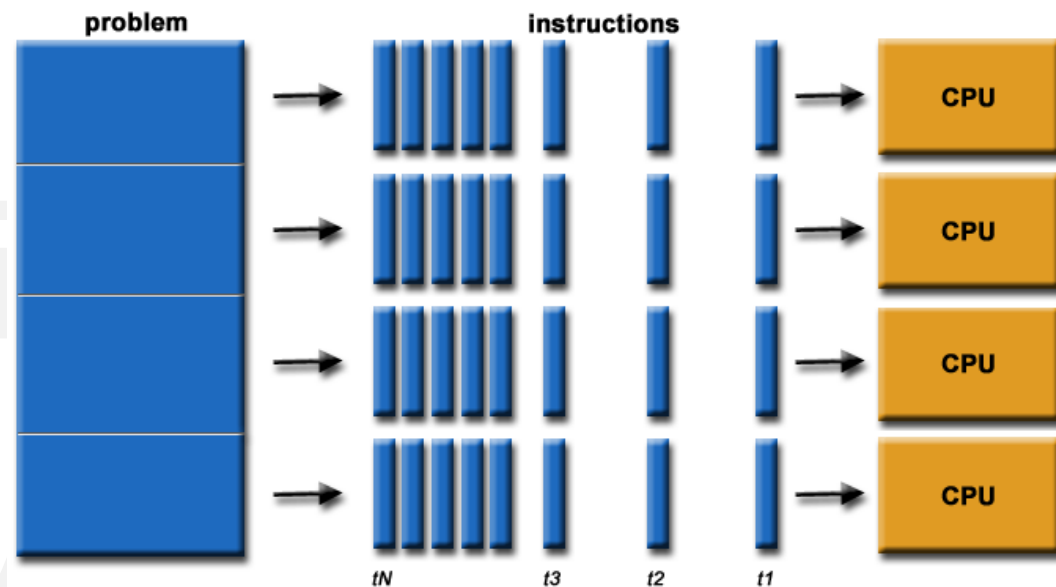


代表性指令集：GPU的CUDA指令集

- SISD、SIMD、MISD、MIMD的扩展

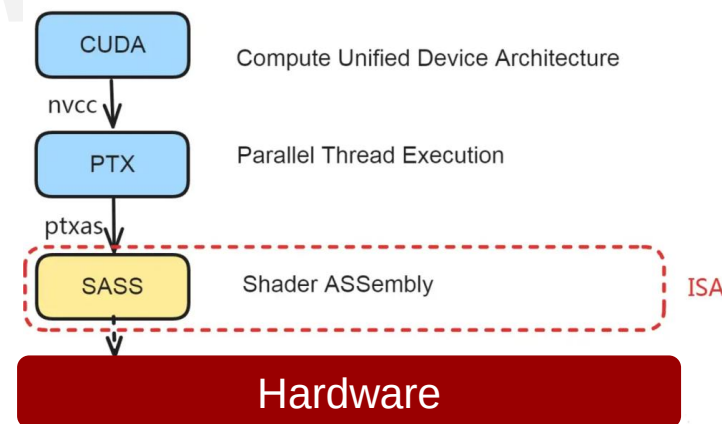


并行计算



		DATA STREAM	
		Single	Multiple
INSTRUCTION STREAM	Single	Single Instruction Single Data SISD	Single Instruction Multiple Data SIMD
	Multiple	Multiple Instruction Single Data MISD	Multiple Instruction Multiple Data MIMD

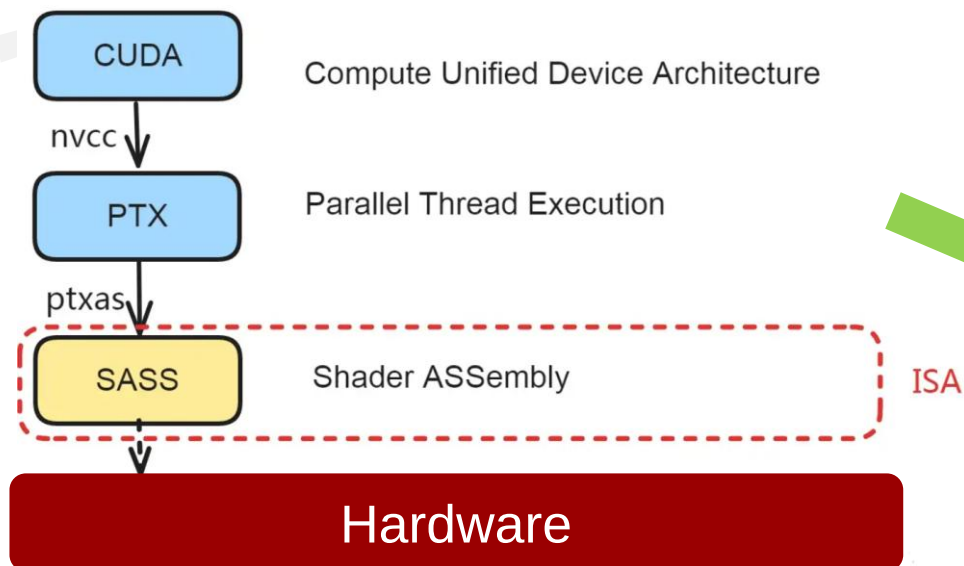
CUDA指令集的层次



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

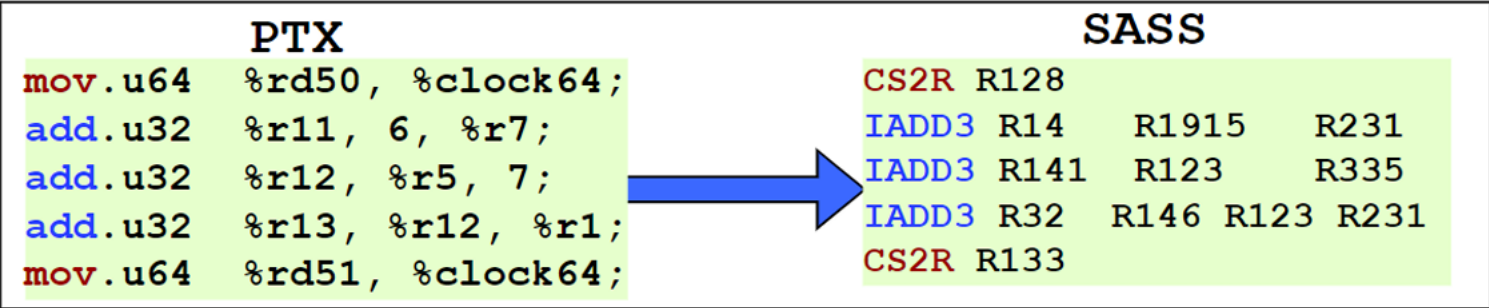
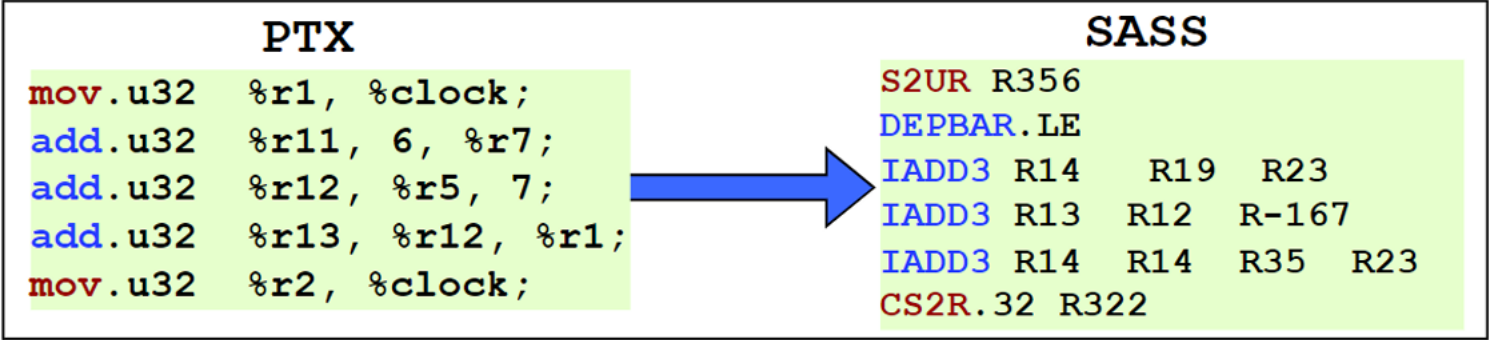


从SASS上抽象出来的一种更上层的软件编程模型，介于CUDA C/C++和SASS之间

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



Mapping of PTX to SASS

代表性指令集： GPU的CUDA指令集

• 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	Rsqr 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.m.f32	multiple instructions	525	clz.b32	FLO.U32 + IADD	7
div.m.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	fn.s.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			