北京大学
PEKING UNIVERSITY

智能硬件体系结构

第五讲：指令集与流水线架构

主讲：陶耀宇、李萌

2025年春季

注意事项

北京大学 PEKING UNIVERSITY

- **课程作业情况**

- **第1次作业将于10.17日（今晚11:59）发布，截止日期：11月7号晚11:59**
  - 3次作业可以使用总计7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：10月17日（今晚11:59）发布 - 11月10日晚11:59**
  - **3个子任务（60%+30%+10%）**
- 第2次lab时间：11月10日-12月7日

注意事项

北京大学 PEKING UNIVERSITY

- **课程作业情况**

- **Clab 配置流程已经上传到课程网站上，请尽快完成配置。**

Project

Lab 0

本课程Lab将依托于CLab进行。你需要在CLab上创建用于本课程的主机，具体步骤见CLab使用指南。

> ❗ *本课程主机会在课程结束后随时被回收，请及时备份你的数据。*

要进行本课程的实验，你可能需要一些基本的Linux操作系统知识。如果你对Linux操作系统不熟悉，可以参阅 MIT 的 missing semeter （中文翻译版：计算机教育中缺失的一课）。

URL: https://aiarchpku.com/2025Fall/project/

Clab配置如有问题可以联系李中源助教

注意事项

北京大学 PEKING UNIVERSITY

- **课程作业情况**

- **Lab 1. 1D Winograd**

  - Winograd算法是一种用于加速卷积运算的高效算法，尤其在深度学习的卷积神经网络（CNN）中广泛应用。它通过减少乘法次数来优化计算，特别适合小尺寸卷积核
  - 本实验中你将实现一个基础的1D Winograd，并使用流水线进行优化

  - 本实验分为三个部分：
    - 组合逻辑的1D Winograd实现（60 Points）
    - 流水线的1D Winograd实现（30 Points）
    - 基于流水线的1D Winograd进一步优化（10 Points）

  - **实验完成后需要提交实验报告到教学网上，详情信息见课程网站 Projects 一栏**
  - **如有问题可以联系老师和詹喆助教**

注意事项

- 课程作业情况

- Lab 1. 1D Winograd

准备实验所需的环境和文件

1. **环境准备**：本实验需要使用 CLab for EDA 环境，请参考 Lab0 进行环境搭建。
2. **文件准备**：在课程共享文件的 `/mnt/nfs/` 文档下，你能看到 `Lab1` 文件夹，该文件夹包含了本实验所需的所有文件。该目录是只读的，你需要将其复制到自己的家目录下进行实验，可以使用如下命令：

```
cp -r /mnt/nfs/Lab1/ ~/Lab1
cd ~/Lab1
```

该目录中包括两个任务所需的文件夹，分别为 `winograd_comb` 和 `winograd_pipeline`，每个文件夹中都包含了实验所需的源代码、测试平台、Makefile等文件。

注意事项

- 课程作业情况

- Lab 1. 1D Winograd

任务 1：基于组合逻辑的 1D Winograd 卷积计算单元

任务 1 中，你需要实现一个纯组合逻辑的，卷积核大小固定的 1D winograd 卷积计算单元。计算单元的部分代码已经提供，你需要填补空缺的代码。

实验步骤

1. `winograd_comb` 文件夹包含了此任务需要的文件。文件结构如下：

```
├── run（进行仿真的目录）
│   ├── Makefile
│   └── winograd.f
├── dc（进行综合的目录）
│   ├── Makefile
│   ├── rpts（综合结果报告）
│   └── ...
└── src（Verilog 源代码目录）
    ├── input_transform.v（此处需要填空）
    ├── output_transform.v（此处需要填空）
    ├── weight_transform.v（此处需要填空）
    ├── winograd_1d_tb.v
    └── winograd_1d.v（此处需要填空）
```

TOP module 位于 `winograd_1d.v` 中。

2. 补充各 `.v` 文件中的空缺，完成代码；

Testbench 运行方法

- **不显示波形**

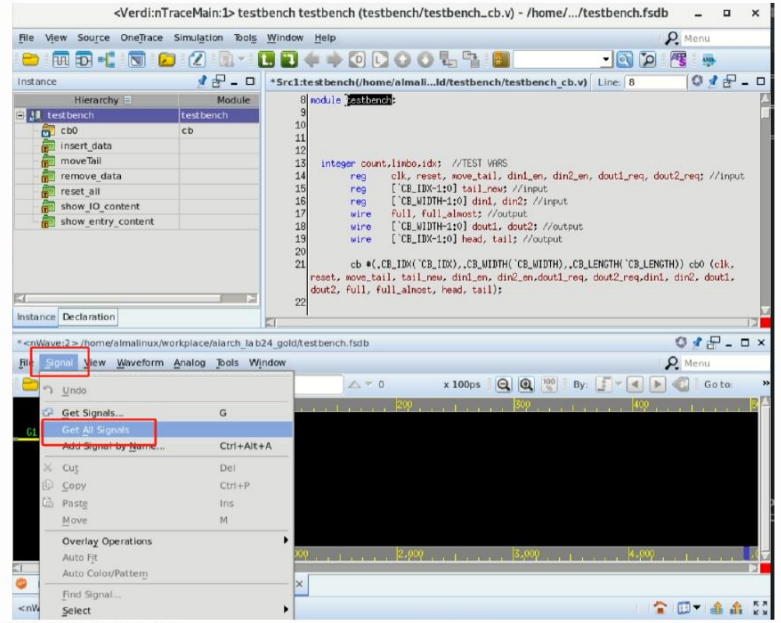
进入 `run` 目录，运行 `make rerun` 即可运行模拟，同时不会显示波形，由于该操作不依赖于GUI，可以在VS Code中直接使用。由于在testbench中设置了 `$monitor` 命令，会输出不同时刻的计算结果，可以用于验证结果是否正确。

```
[Case     0] PASS at 55000: Expected r1_res=        160, r2_res=       -210, Results match expectations.
[Case     1] PASS at 65000: Expected r1_res=          0, r2_res=          0, Results match expectations.
[Case     2] PASS at 75000: Expected r1_res=    1769220, r2_res=   -1993901, Results match expectations.
[Case     3] PASS at 85000: Expected r1_res= 1073676289, r2_res= 1073676289, Results match expectations.
[Case     4] PASS at 95000: Expected r1_res= 1073741824, r2_res= 1073741824, Results match expectations.
[Case     5] PASS at 105000: Expected r1_res=          3, r2_res=          3, Results match expectations.
[Case     6] PASS at 115000: Expected r1_res=          2, r2_res=         -2, Results match expectations.
[Case     7] PASS at 125000: Expected r1_res=  145063936, r2_res=   93028352, Results match expectations.
[Case     8] PASS at 135000: Expected r1_res= 2147483646, r2_res= 2147483646, Results match expectations.
[Case     9] PASS at 145000: Expected r1_res=-2147483648, r2_res=-2147483648, Results match expectations.
$finish called from file "../src/winograd_1d_tb.v", line 138.
$finish at simulation time                225000
           V C S   S i m u l a t i o n   R e p o r t
```

- **显示波形（使用verdi）**

配合Verdi，可以显示电路运行过程中的波形，便于进行debug。进入 `run` 目录，运行 `make all` 即可进行模拟并显示波形。由于依赖GUI，**这一步骤需要使用远程桌面。**

如运行成功，会自动弹出Verdi窗口，在Verdi窗口中，选择下方 `Signal – Get All Signals` 即可显示所有波形。



4. 清除运行产生的临时文件

只要运行 `make clean` 即可清除所有运行产生的临时文件

注意事项

- **课程作业情况**

- **Lab 1. 1D Winograd**

任务 2： 基于流水线的 1D Winograd 卷积计算单元

虽然如任务1中所做的那样，1D winograd可以完全通过组合逻辑完成计算，但是这会导致较大的延迟，阻碍频率和吞吐的提升，因此在任务2中，我们需要引入流水线来解决这一问题。

实验步骤

1. **尝试综合纯组合逻辑的设计**：进入 `winograd_comb/dc` 目录，运行 `make dc` 进行综合。这里我们设置不同的时钟周期，进行两次综合。
   - **第一次综合**：直接综合，使用默认的 20ns 时钟周期。建议保存 `dc` 目录下的 `rpts` 目录，以便后续对比。
   - **第二次综合**：修改时钟周期为 4ns，重新运行 `make dc` 进行综合。具体操作是：
     - 在 `winograd_comb/dc/syn_tcl/syn.tcl` 文件中，将 `set CLK_PERIOD 20.0` 修改为 `set CLK_PERIOD 4.0`；
     - 保存文件后，重新运行 `make dc` 进行综合。
     - 同样建议保存 `rpts` 目录，以便对比。

   综合完成输出如下：

```
check_error -verbose
0
#error_info
exit

Memory usage for this session 1107 Mbytes.
Memory usage for this session including child processes 1407 Mbytes.
CPU usage for this session 208 seconds ( 0.06 hours ).
Elapsed time for this session 105 seconds ( 0.03 hours ).

Thank you...
```

综合完成后，你可以在 `rpts` 目录下查看综合报告。你可以在 `winograd_1d.area.rpt` 文件中查看面积情况。此处我们重点关注 `winograd_1d.timing.max.rpt` 文件，该文件中包含了时序约束的满足情况。你会发现，纯组合逻辑的设计无法满足 4ns 的时序约束。该文件的内容可能像是：

```
Startpoint: r3_w[10] (input port clocked by clk)
Endpoint: r1_res[31] (output port clocked by clk)
Path Group: clk
Path Type: max

    ...

Point                       Fanout     Trans      Incr       Path
------------------------------------------------------------------------------

    ...

------------------------------------------------------------------------------
data required time                                            3.50
data arrival time                                            -4.83
```

```
------------------------------------------------------------------------------
slack (VIOLATED)                                             -1.33
```

这里的 `slack` 即为时序裕量，负值表示不满足时序约束。这里意味着，`r3_w[10]` 到 `r1_res[31]` 之间的路径延迟超过了 4ns 的时序约束，无法在 4ns 内完成计算。

2. **引入流水线**：进入 `winograd_pipeline` 目录，该目录下的文件结构与 `winograd_comb` 目录类似，区别在于 `src` 目录下的 `winograd_1d.v` 文件中你需要补充流水线的相关代码。这里你需要将任务1中完成的 `input_transform.v`、`weight_transform.v`、`output_transform.v` 复制到 `winograd_pipeline/src/` 目录下，并在 `winograd_1d.v` 中实例化这些模块。具体如何完成可以参考 `winograd_1d.v` 文件中的提示。

3. **验证功能**：进入 `winograd_pipeline/run` 目录，运行 `make rerun` 或 `make all` 来验证功能，确保你的设计与任务1中的设计功能一致。

4. **综合并验证时序**：进入 `winograd_pipeline/dc` 目录，运行 `make dc` 进行综合，综合完成后，你可以在 `rpts` 目录下查看综合报告，重点关注 `winograd_1d.timing.max.rpt` 文件，确保时序约束被满足。此时你应该会看到 `slack` 为正值，表示时序约束被满足。

   此外，你可以在 `winograd_1d.area.rpt` 文件中查看面积报告，在 `winograd_1d.power.rpt` 文件中查看功耗报告，了解不同设计的面积和功耗情况。

在任务2中，你已经成功实现了一个基于流水线的1D Winograd卷积计算单元，并且满足了时序约束。接下来，你可以尝试对设计进行优化，以进一步提升运行频率。

实验步骤

1. **通过时序报告了解瓶颈**：通过查看 `winograd_pipeline/dc/rpts/winograd_1d.timing.max.rpt` 文件，可以看到slack最小的路径（该文件是按照slack升序排列的）。你可以通过查看这些路径，了解设计中的瓶颈所在。此外，你也可以尝试将时钟周期进一步缩短（例如2ns），重新进行综合，查看此时是否发生了时序违例。

   这里预期瓶颈主要出现在大位宽的乘法器上。

2. **优化设计**：你可以尝试使用流水线乘法器，将大位宽的乘法器替换为分拆为位宽较小的流水线乘法器。这样可以减少单个乘法器的延迟，从而提升整体设计的频率。

3. **验证功能和时序**：在进行优化后，重复任务2中的步骤3，确保优化后的设计功能正确且满足时序约束。

4. **比较优化前后的性能**：使用 2ns 甚至更短的时钟周期，重新进行综合，并比较优化前后的时序报告，查看slack的变化，了解优化带来的提升。同时，你也可以比较面积和功耗的变化，了解优化的代价。

实验报告

请将你的实验过程和结果整理成实验报告，内容包括：

1. 任务1的实现过程和结果截图；
2. 对任务2中纯组合逻辑设计的综合结果分析（包括时序报告截图和分析）；
3. 任务2中流水线设计的实现过程和结果截图；
4. 对比分析任务2中纯组合逻辑设计 (20ns) 和流水线设计 (4ns) 的综合结果（包括时序、面积、功耗等方面的对比）；
5. **（可选）**任务3的优化过程和结果截图，以及优化前后的对比分析。

目 录

CONTENTS

复杂模块电路设计1 - 卷积加速模块

北京大学 PEKING UNIVERSITY

- 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 = [d0,d1,d2,d3]，卷积核g = [g0, g1, g2]

1维卷积可表达为

输入尺寸 = 2 + 3 -1 = 4

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} r0 \\ r1 \end{bmatrix}$$

输出尺寸　卷积核尺寸

6次乘法

卷积加速模块设计

- 卷积 – AI计算中最常用的算子

输入d = [d0,d1,d2,d3]，卷积核g = [g0, g1, g2]

1维卷积可表达为

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} r0 \\ r1 \end{bmatrix}$$

存在重复元素

Winograd

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} m1 + m2 + m3 \\ m2 - m3 - m4 \end{bmatrix}$$

$$m1 = (d0 - d2)g0 \quad m2 = (d1 + d2)\frac{g0 + g1 + g2}{2}$$

$$m4 = (d1 - d3)g2 \quad m3 = (d2 - d1)\frac{g0 - g1 + g2}{2}$$

预算好一次g的加减后可重复复用 -> 4次乘法

卷积加速模块设计

- 卷积 – AI计算中最常用的算子

element-wise multiplication（Hadamard product）

输入d = [d0,d1,d2,d3]，卷积核g = [g0, g1, g2]

1维卷积可表达为

$$F(2,3) = \begin{bmatrix} d0 & d1 & d2 \\ d1 & d2 & d3 \end{bmatrix} \begin{bmatrix} g0 \\ g1 \\ g2 \end{bmatrix} = \begin{bmatrix} m1+m2+m3 \\ m2-m3-m4 \end{bmatrix}$$

$$m1 = (d0 - d2)g0 \quad m2 = (d1 + d2)\frac{g0+g1+g2}{2}$$

$$m4 = (d1 - d3)g2 \quad m3 = (d2 - d1)\frac{g0-g1+g2}{2}$$

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

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

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

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} m1+m2+m3 \\ m2-m3-m4 \end{bmatrix}$$

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

$$d = [d0 \quad d1 \quad d2 \quad d3]^T$$

1D Winograd

卷积加速模块设计

北京大学 PEKING UNIVERSITY

- **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$(m+r-1)(n+s-1)$。

卷积加速模块设计

- 卷积 – AI计算中最常用的算子

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

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

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

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

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

$$d=[d0 \quad d1 \quad d2 \quad d3]^T$$

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$m\times(m+r-1)$

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

卷积加速模块设计

- 卷积 – AI计算中最常用的算子

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

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

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

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

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

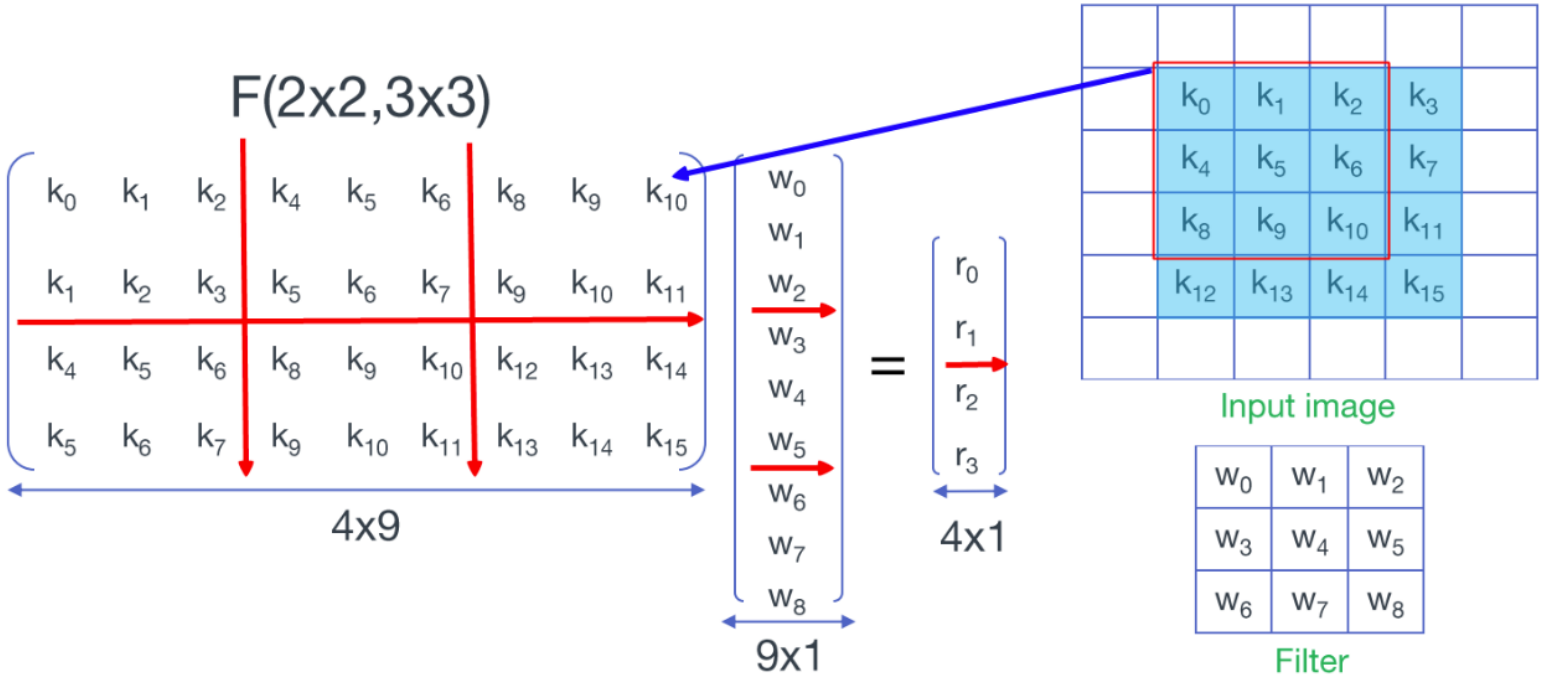
$$d = [d0 \quad d1 \quad d2 \quad d3]^T$$

2D Winograd

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

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

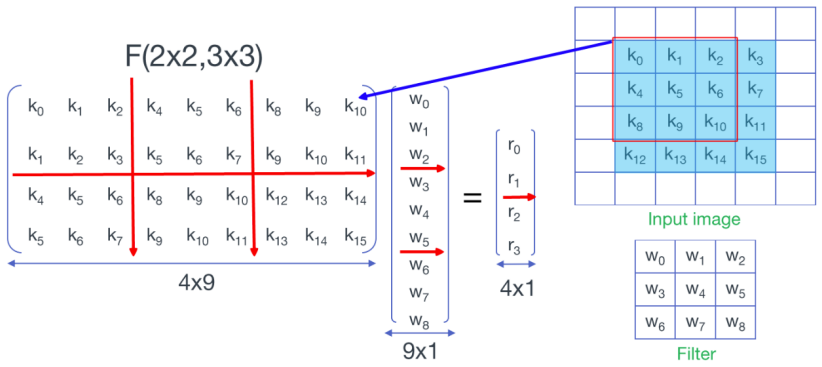
g为$r \times r$ Filter，d为$(m+r-1) \times (m+r-1)$



卷积加速模块设计

- **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；$W_0 = [w_0, w_1, w_2]^T$，

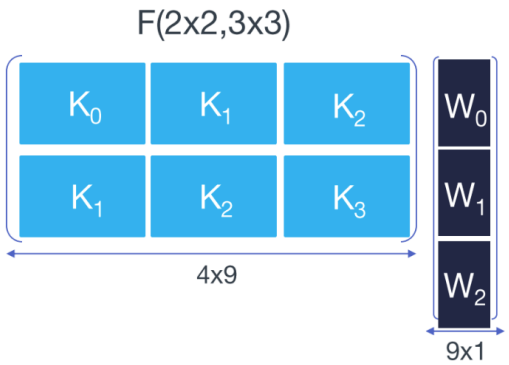


$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \end{bmatrix} = \begin{bmatrix} R_0 \\ R_1 \end{bmatrix} = \begin{bmatrix} K_0W_0 + K_1W_1 + K_2W_2 \\ K_1W_0 + K_2W_1 + K_3W_2 \end{bmatrix}$$

$$= \begin{bmatrix} A^T\left[(GW_0) \odot (B^TD_0)\right] + A^T\left[(GW_1) \odot (B^TD_1)\right] + A^T\left[(GW_2) \odot (B^TD_2)\right] \\ A^T\left[(GW_0) \odot (B^TD_1)\right] + A^T\left[(GW_1) \odot (B^TD_2)\right] + A^T\left[(GW_2) \odot (B^TD_3)\right] \end{bmatrix}$$

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

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



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

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

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

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

$$B^T=\begin{bmatrix}4&0&-5&0&1&0\\0&-4&-4&1&1&0\\0&4&-4&-1&1&0\\0&-2&-1&2&1&0\\0&2&-1&-2&1&0\\0&4&0&-5&0&1\end{bmatrix}$$

$$G=\begin{bmatrix}\frac{1}{4}&0&0\\-\frac{1}{6}&-\frac{1}{6}&-\frac{1}{6}\\-\frac{1}{6}&\frac{1}{6}&-\frac{1}{6}\\\frac{1}{24}&\frac{1}{12}&\frac{1}{6}\\\frac{1}{24}&-\frac{1}{12}&\frac{1}{6}\\0&0&1\end{bmatrix}$$

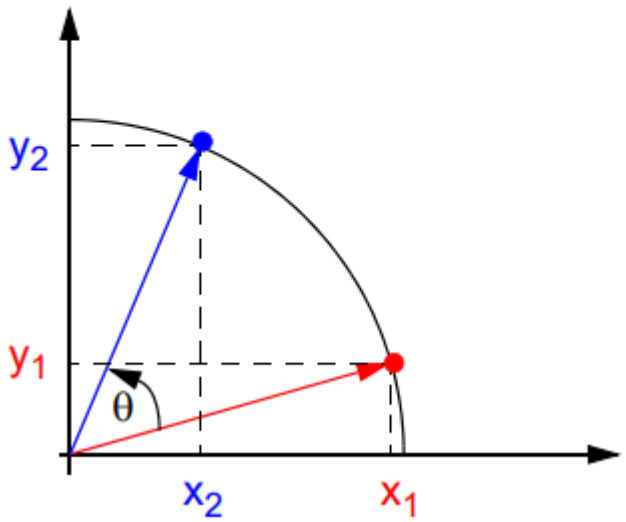
$$A^T=\begin{bmatrix}1&1&1&1&1&0\\0&1&-1&2&-2&0\\0&1&1&4&4&0\\0&1&-1&8&-8&1\end{bmatrix}$$

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）

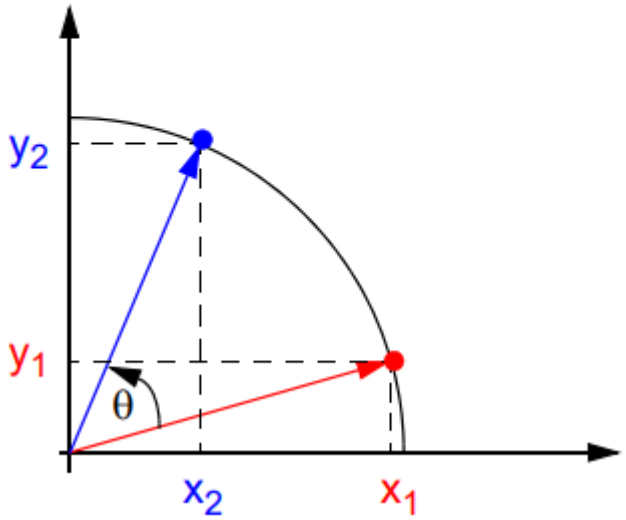
复杂模块电路设计2 - CORDIC模块设计

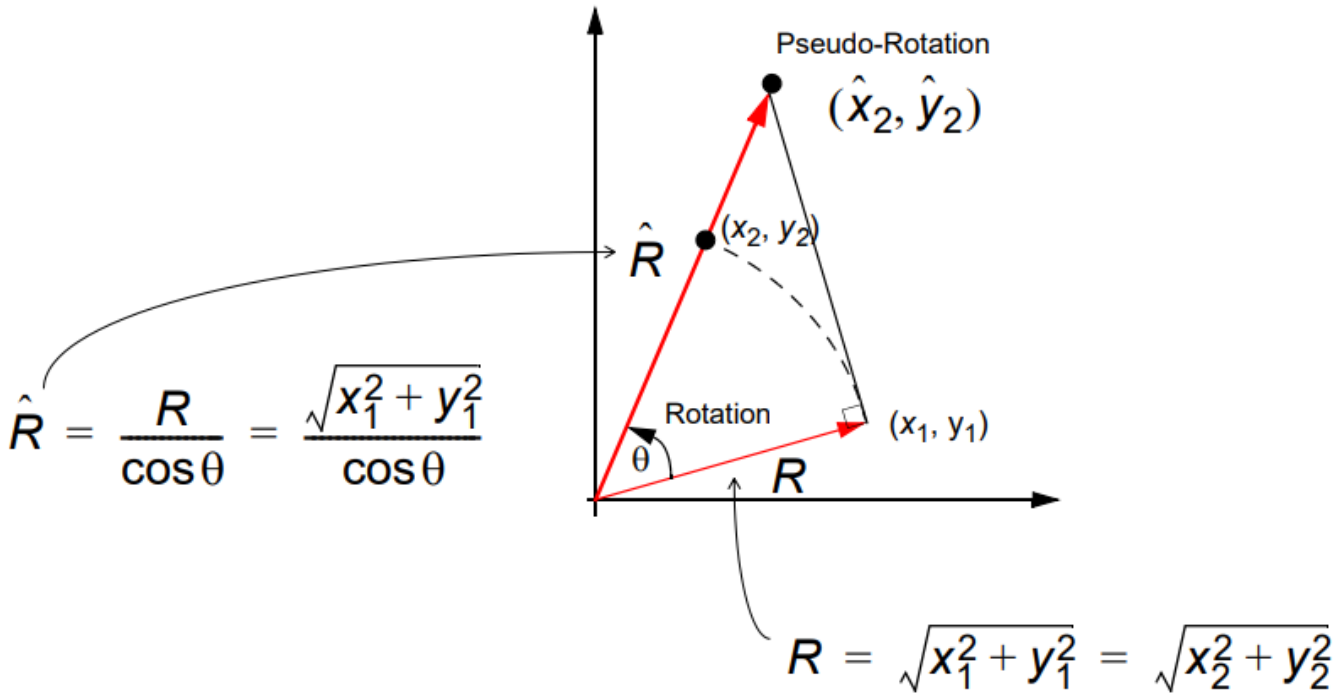
- CORDIC可以用来实现多种复杂非线性函数

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

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）

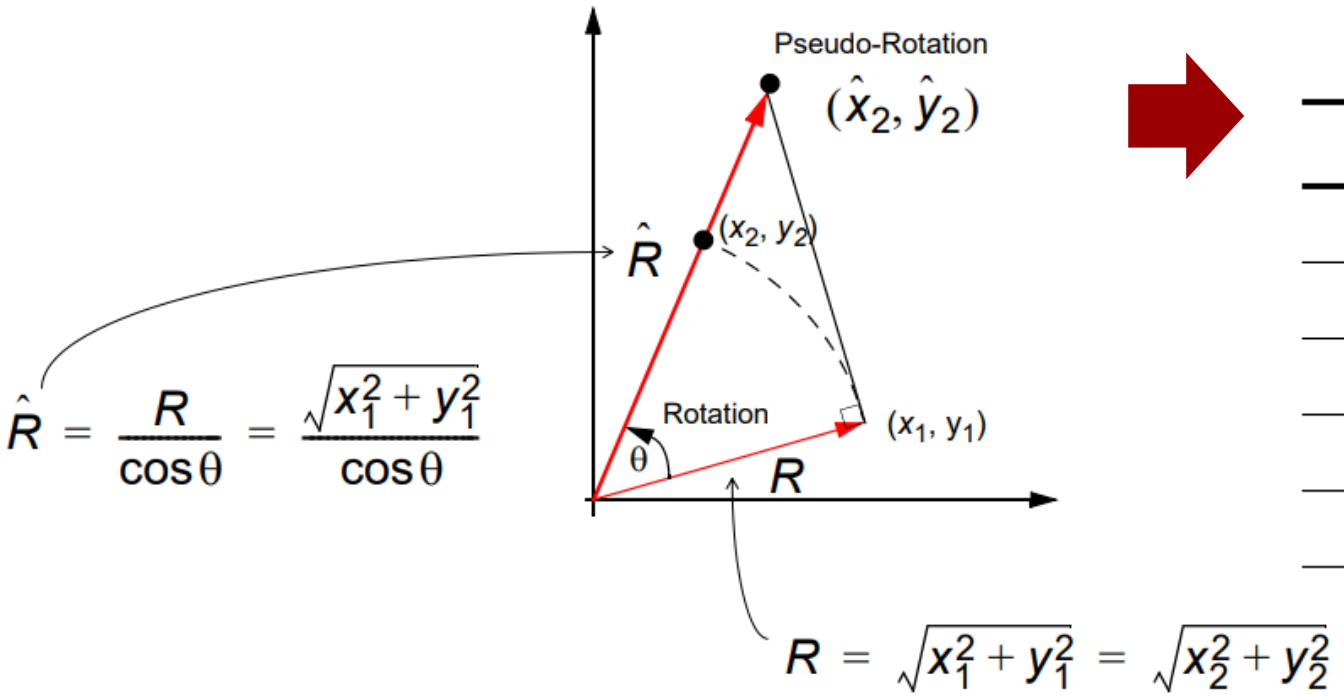
$$\hat{x}_2 = \cos\theta(x_1 - y_1\tan\theta) = x_1 - y_1\tan\theta$$

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

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

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

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

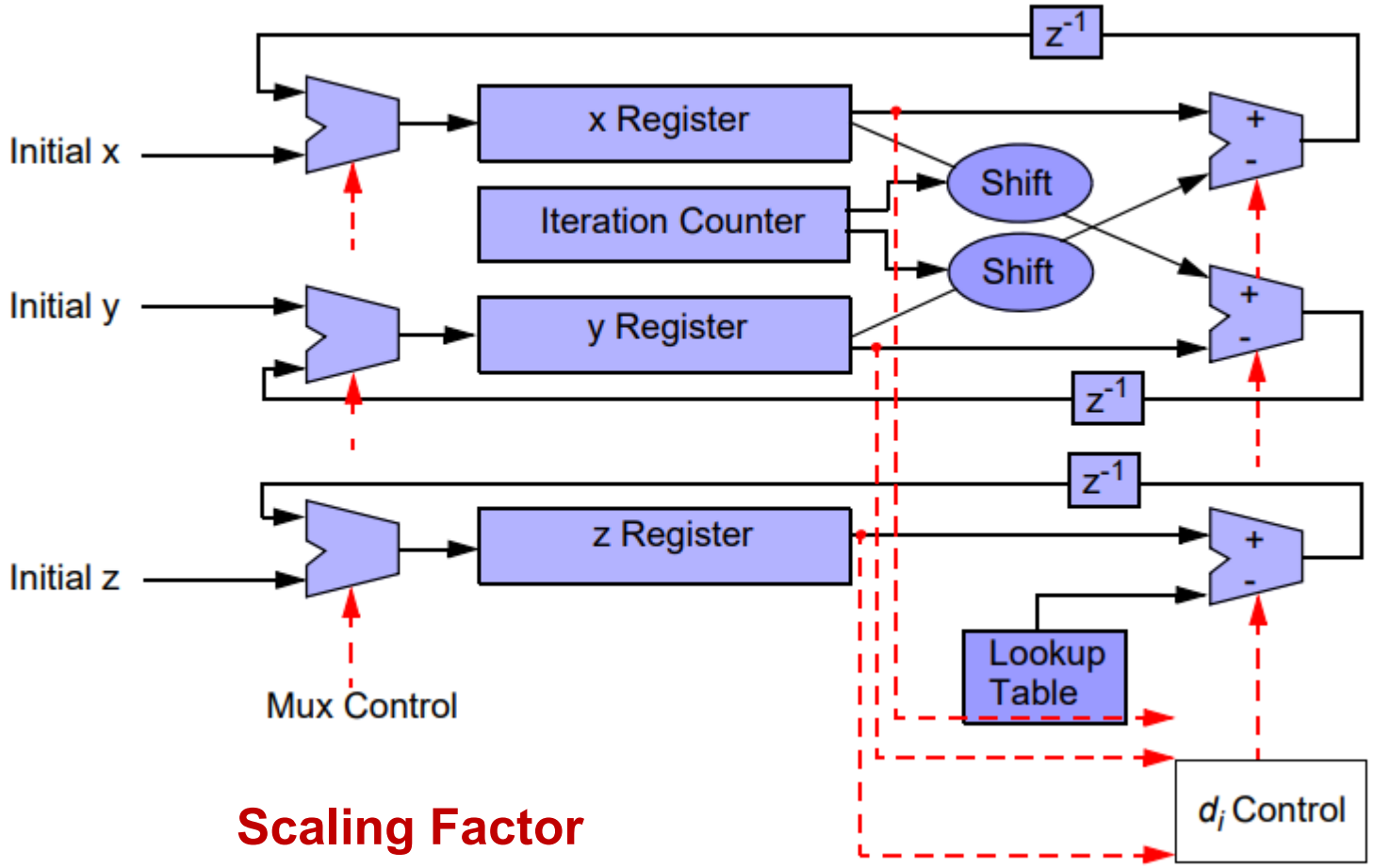
$$z^{(i+1)} = z^{(i)} - d_i\theta^{(i)}$$

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

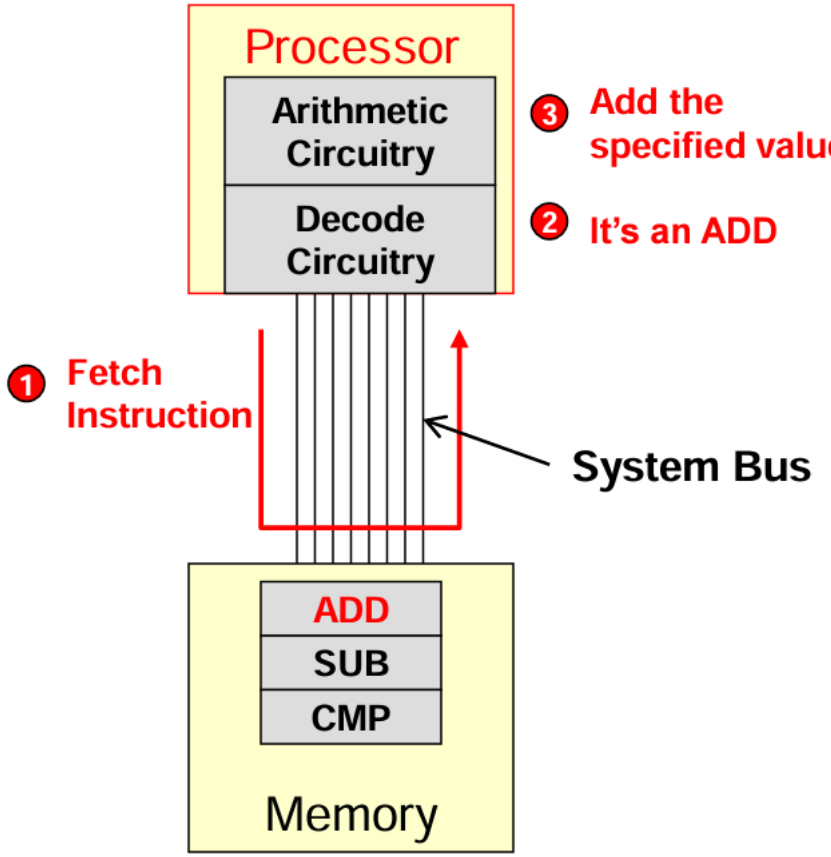
```
Disassembly of section .text:

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

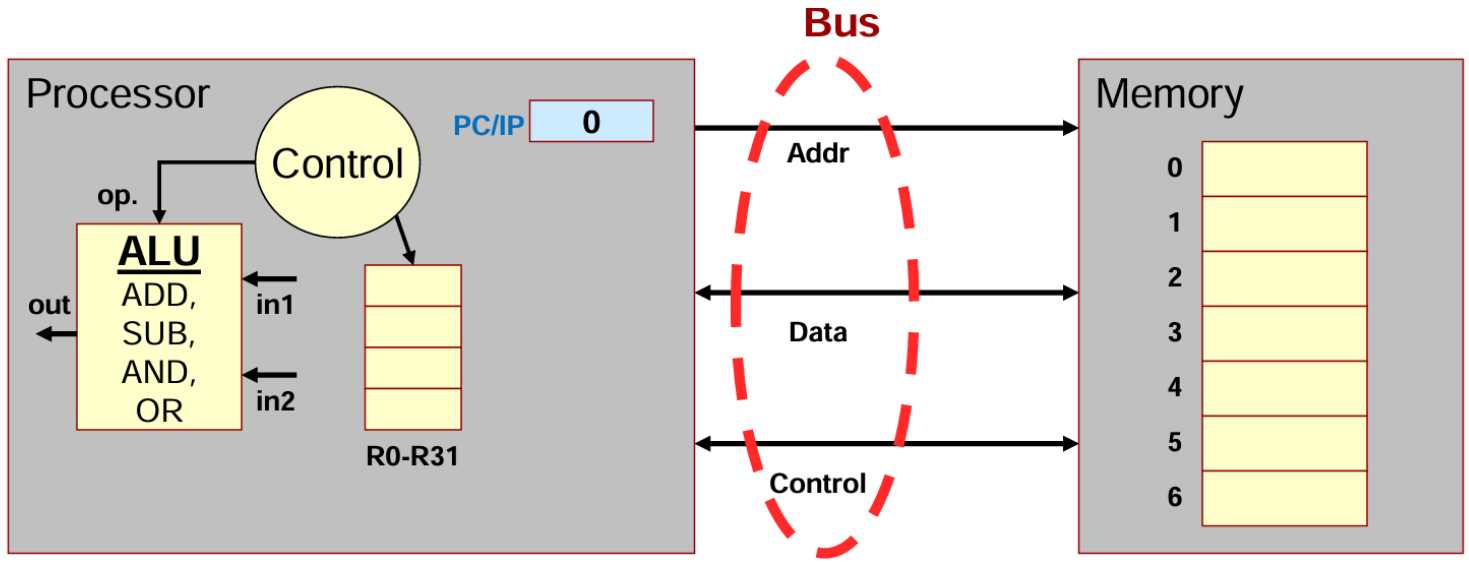
- 需要3类指令：读取、写回和运算

- 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

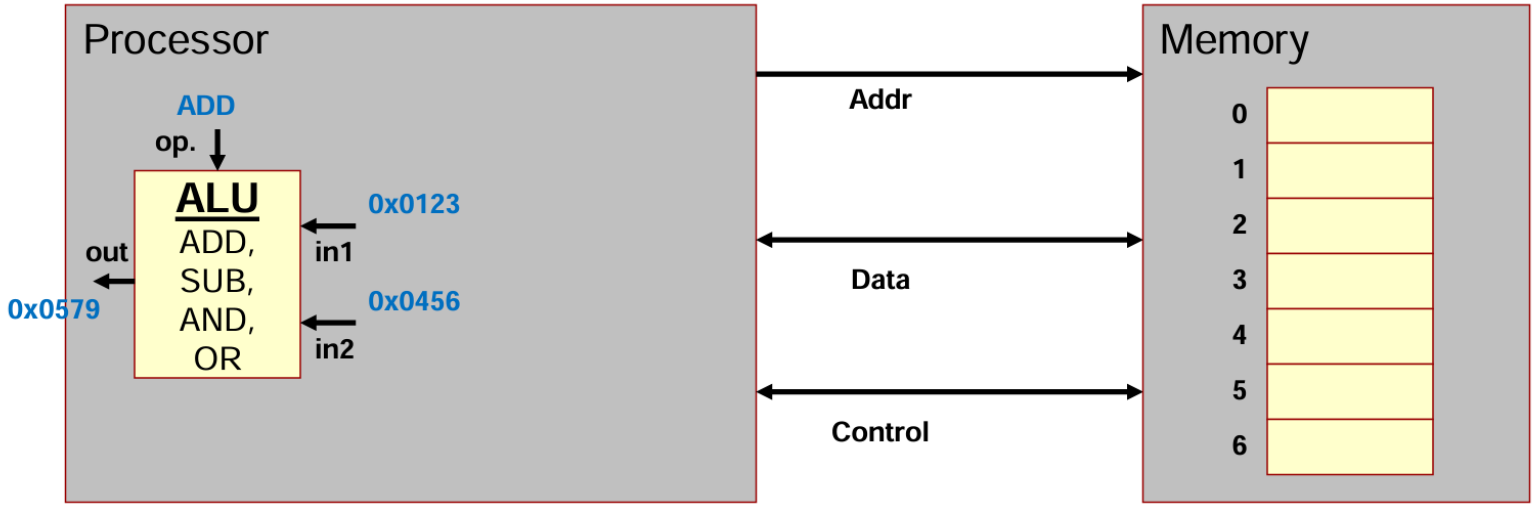
- 需要3类指令：读取、写回和运算
  - 处理器中3种主要的组成
    - ALU（算术逻辑单元）
    - 寄存器
    - 控制电路
  - 通过地址、数据和控制总线（bus）与存储器和I/O连接



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

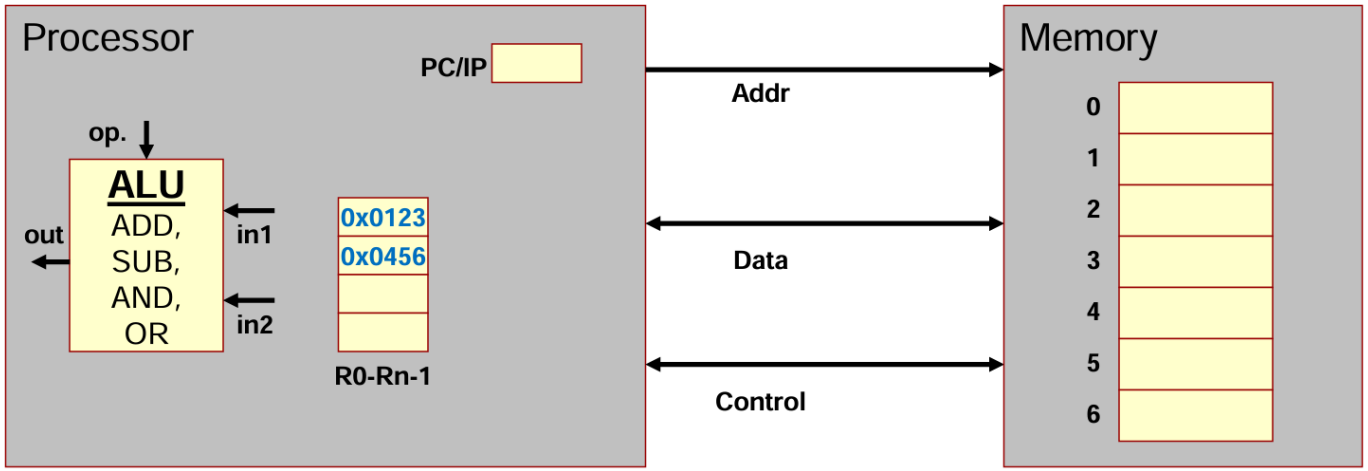
  - 执行加减、逻辑运算等(AND,OR,etc.)算术操作的数字电路



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

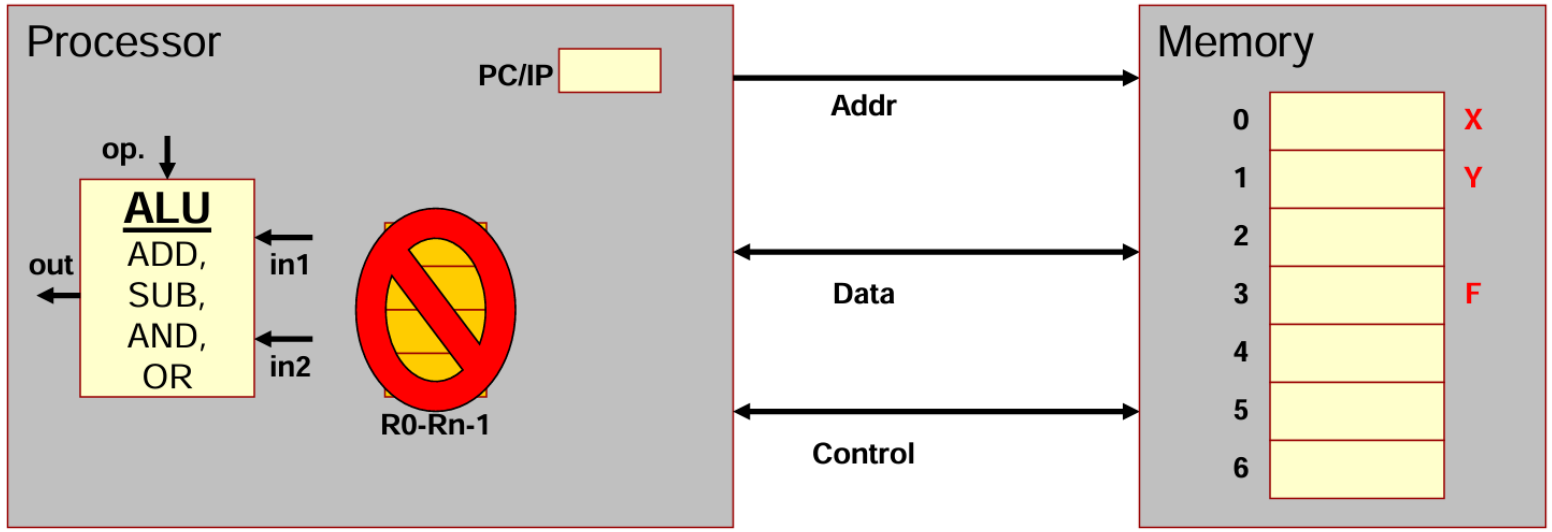
  - 访问存储器的时间通常比处理器要慢
  - 寄存器在处理器内部提供了快速的暂时存储位置



- 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）

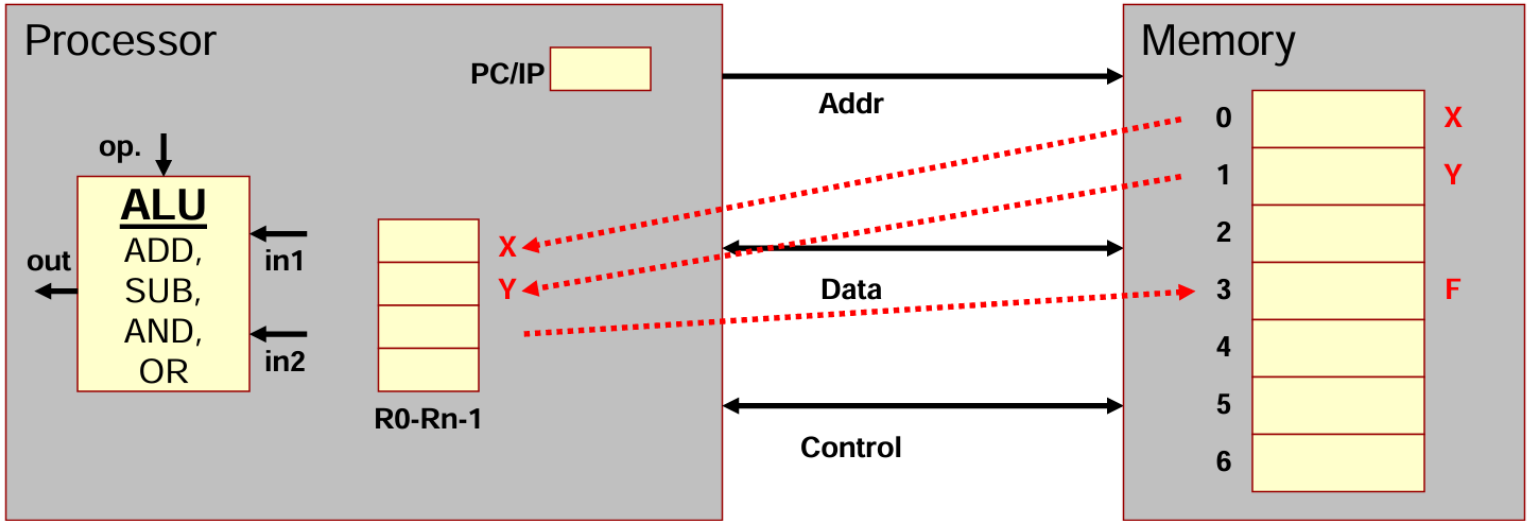
传统存算分离的指令集架构 – 核心部件1：Register

- 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**



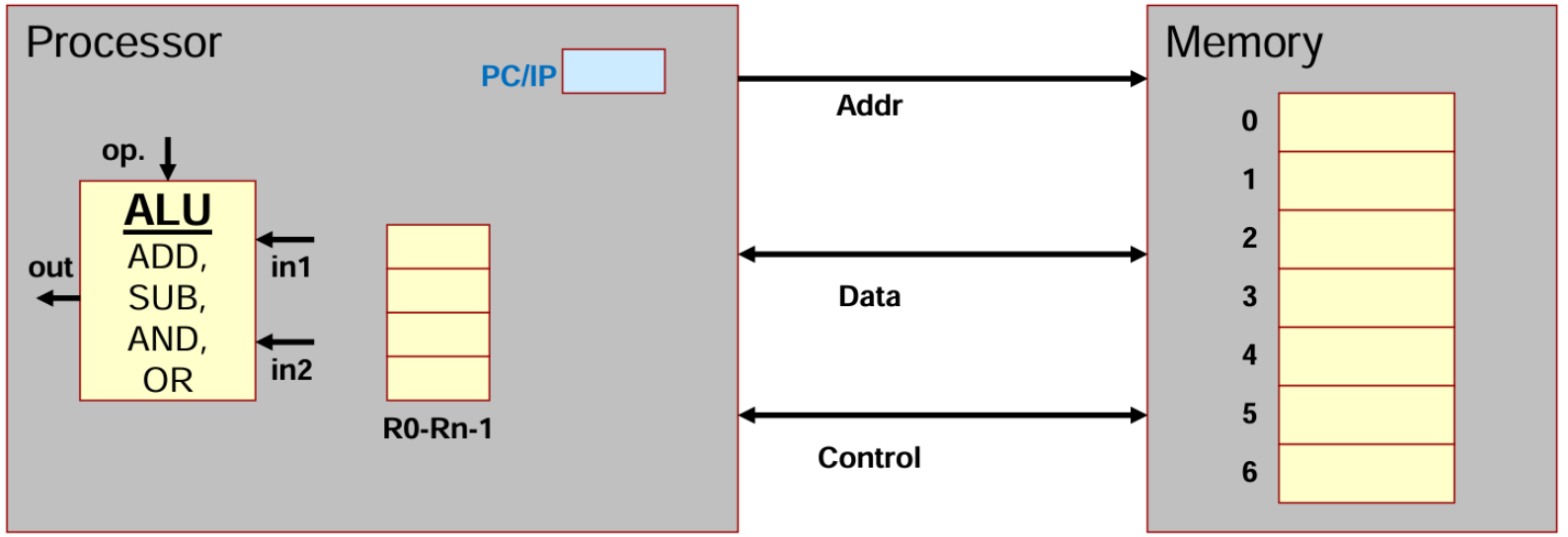
传统存算分离的指令集架构 – 核心部件1：Register

- 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



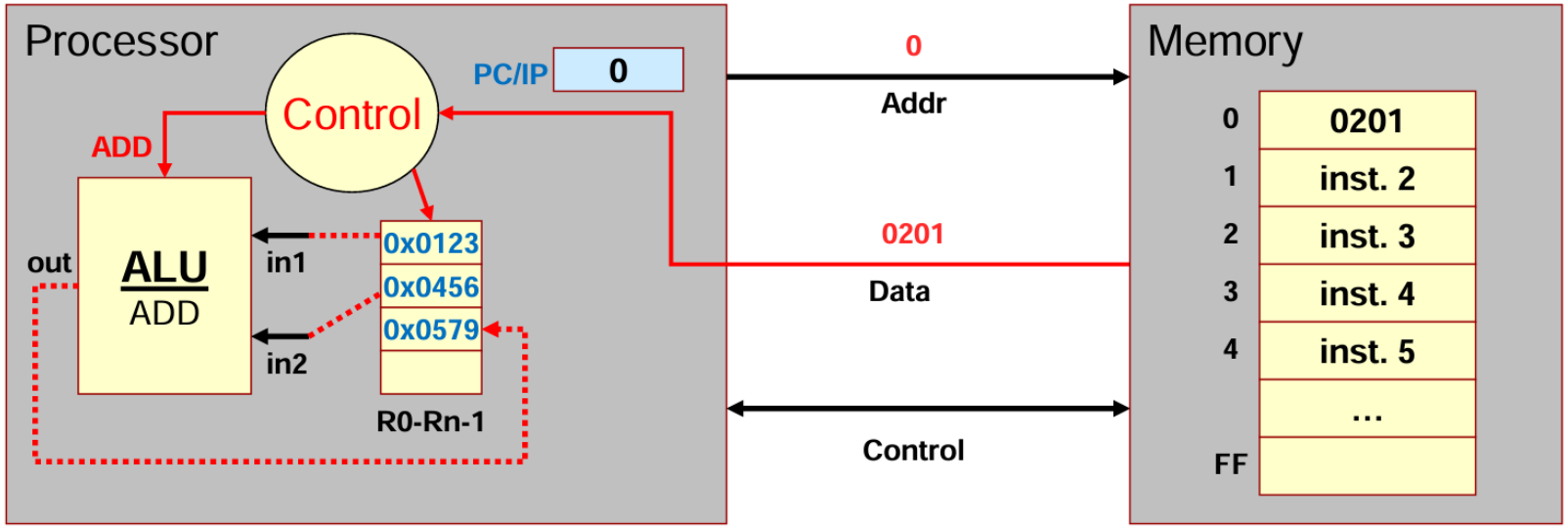
传统存算分离的指令集架构 – 核心部件1：Register

- 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



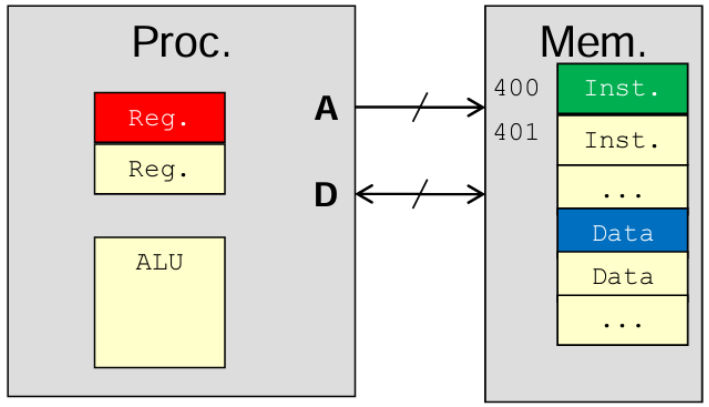
简单指令集架构的操作流程

- 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

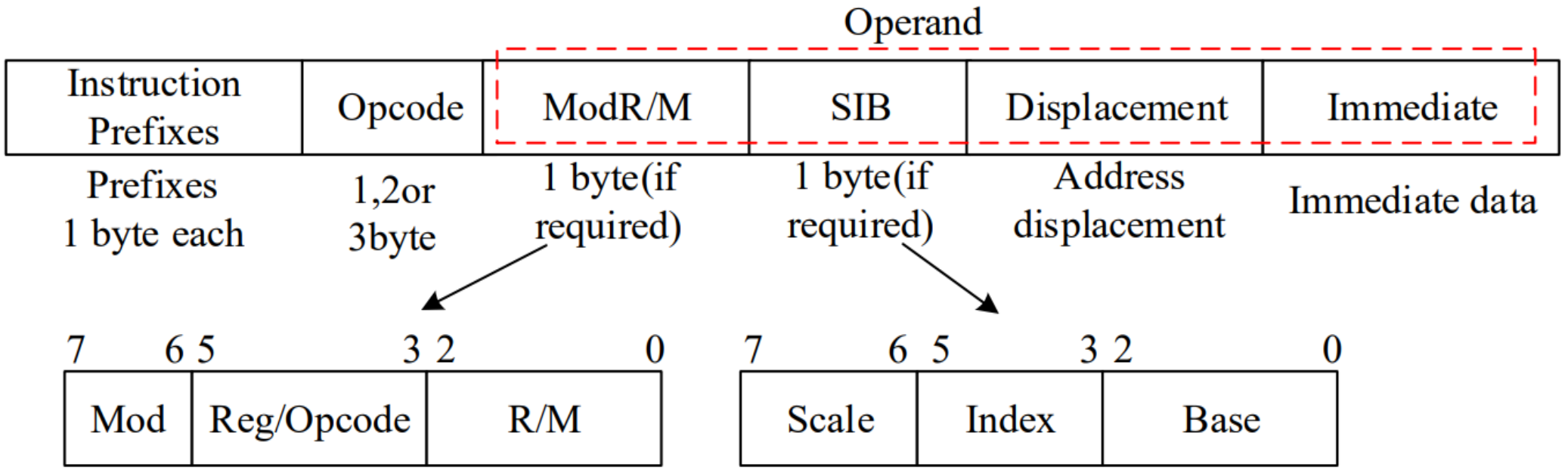
指令集	类型	运营公司	特点	代表厂商
X86	CISC	Intel、AMD	功能强大、通用性、兼容性、实用性	Intel、AMD
MIPS	RISC	MIPS	简洁、优化、高扩展性、寄存器多	Intel、IBM、龙芯、Oracle、Toshiba
ARM	RISC	ARM	低功耗、低成本、适用于移动设备	苹果、华为、谷歌
RISC-V	RISC	RISC-V基金会	完全开源、架构简单、移植性高、开源工具链	数百家大学、科研机构和企业
CUDA	RISC	Nvidia	张量高并发处理、图像处理	Nvidia

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.)

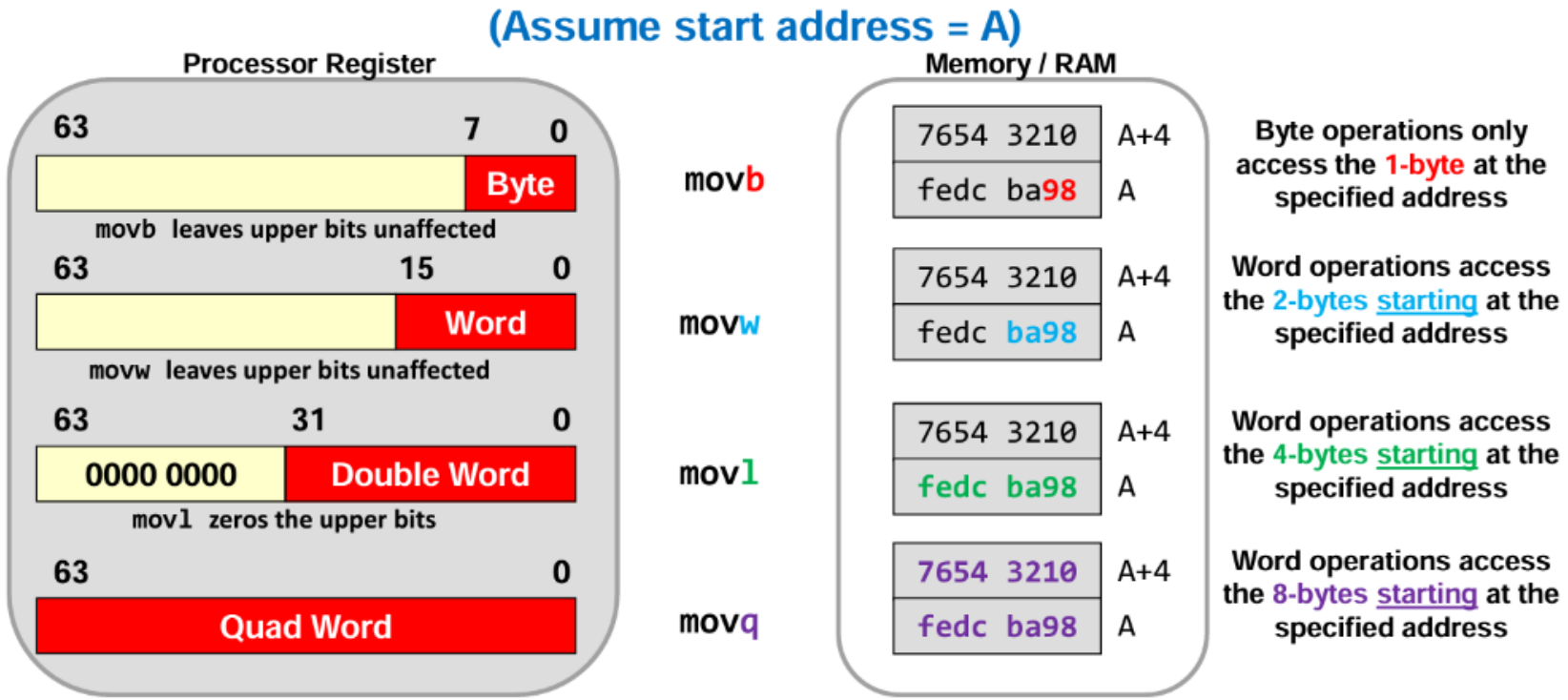
代表性指令集：X86一种典型的CISC指令

- 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]

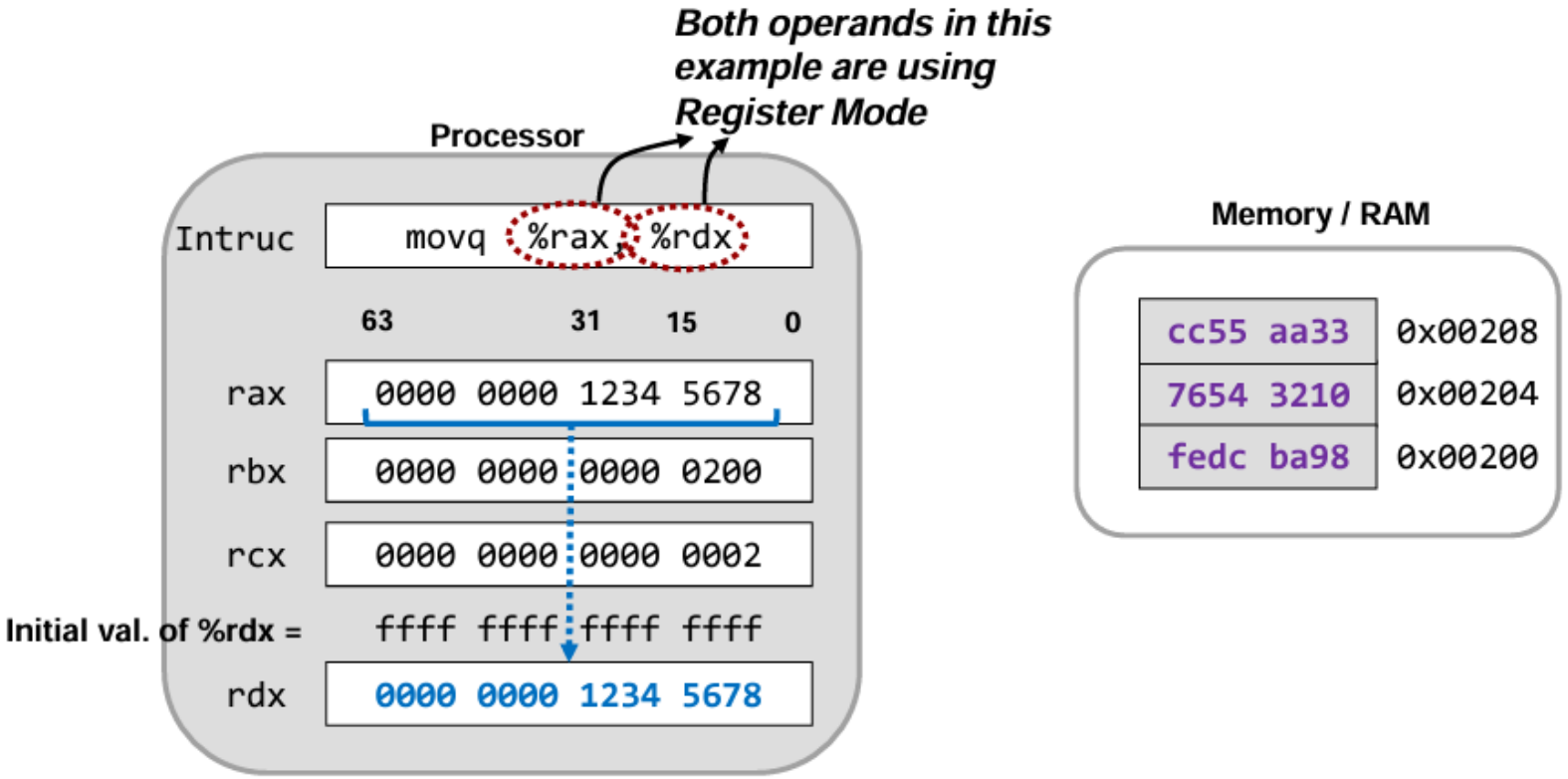
†Known as the scale factor and can be {1,2,4, or 8}

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

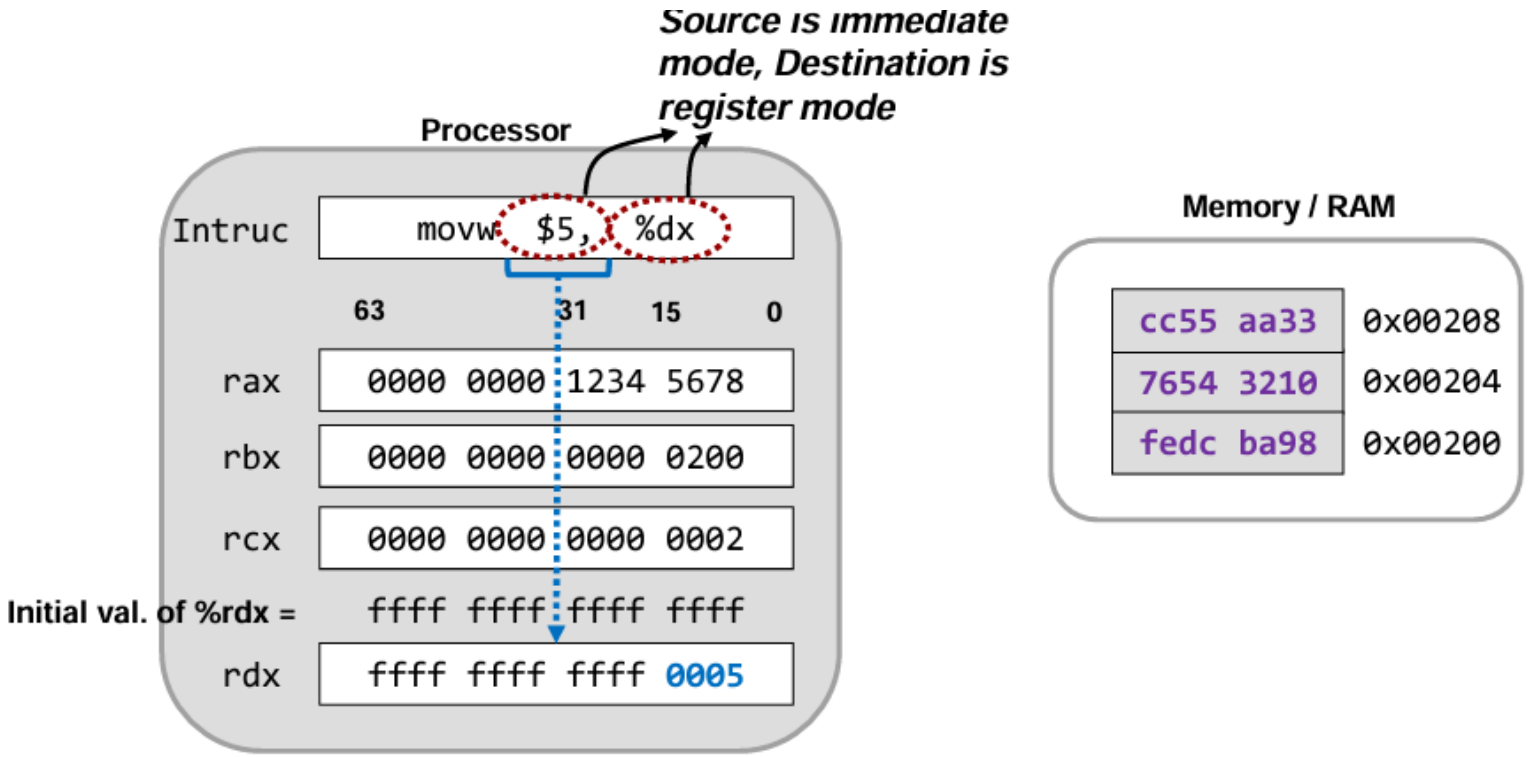
- 指定寄存器的内容作为操作数



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

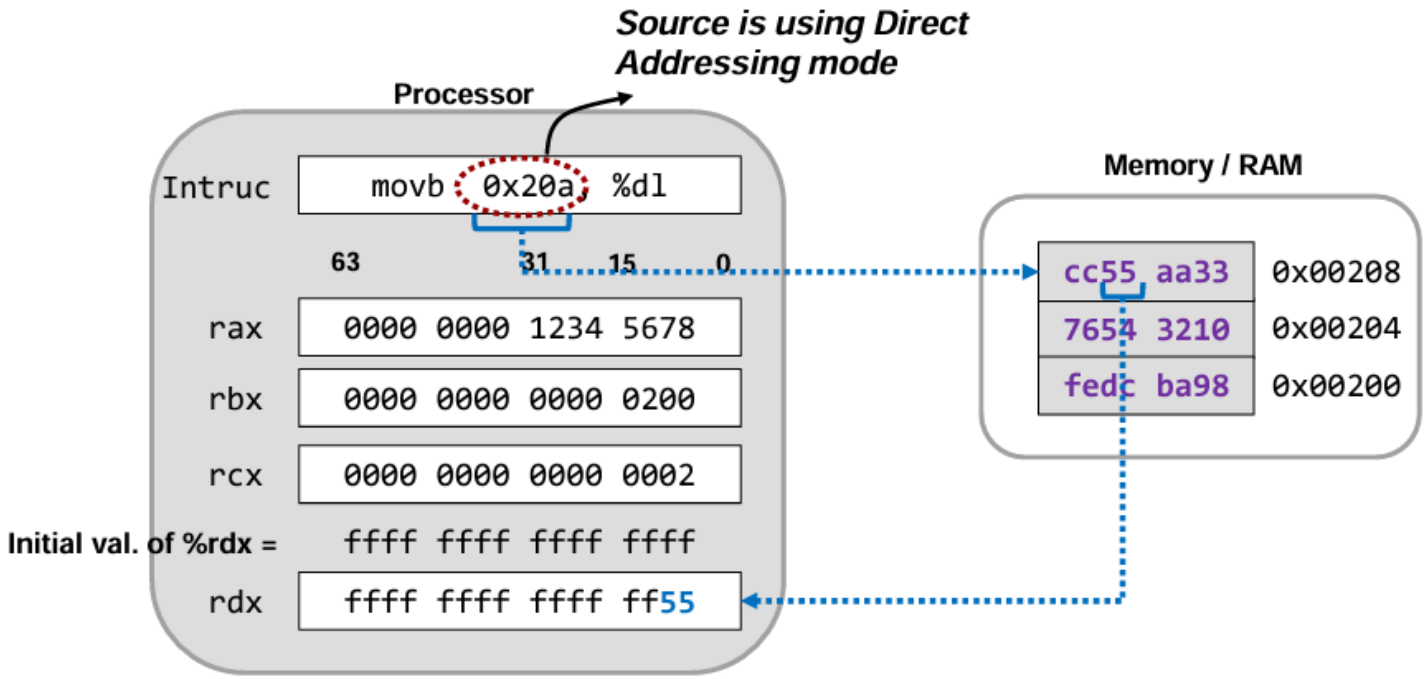
  - 符号’ $’ 表示立即数，并且可以指定用十六进制或是十进制



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

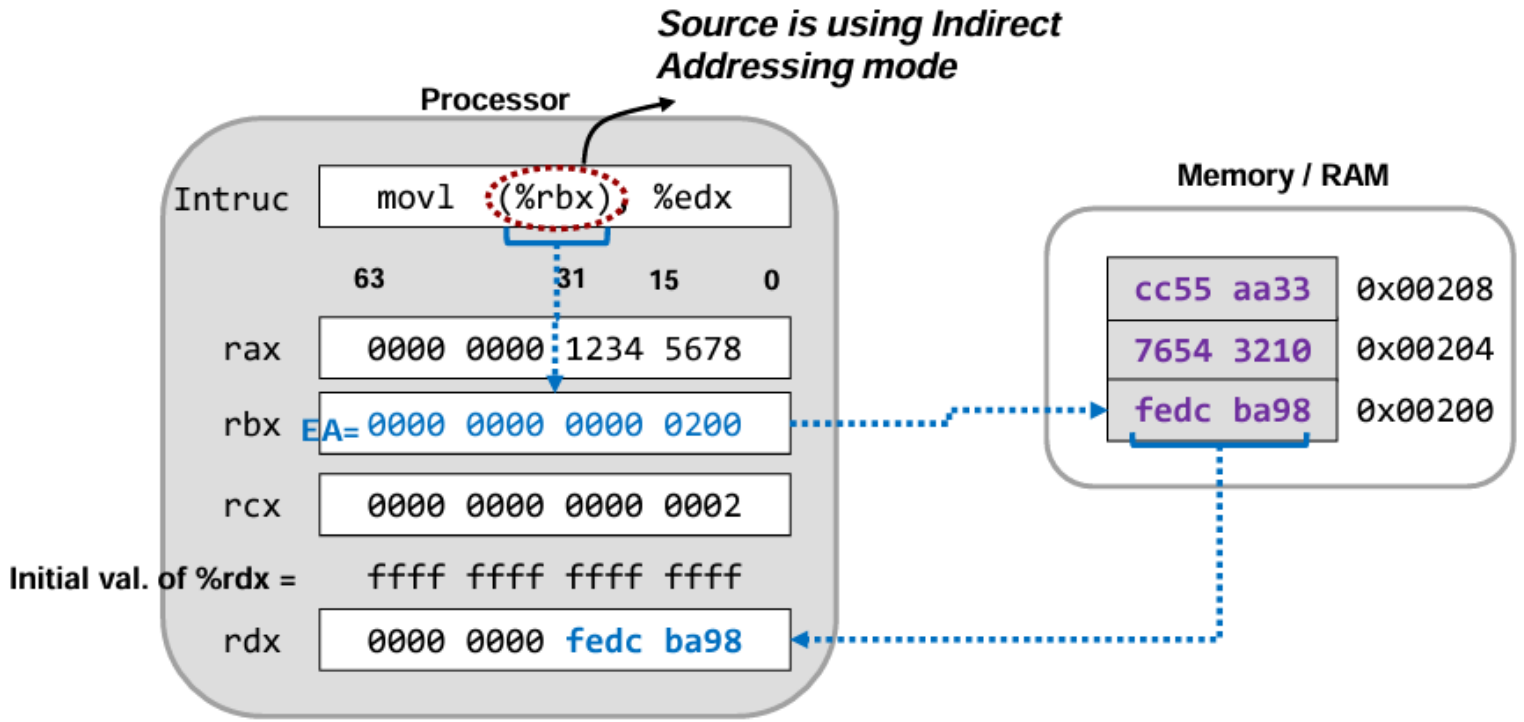


指令集的分类1 – 传输指令： Indirect Addressing Mode

- 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

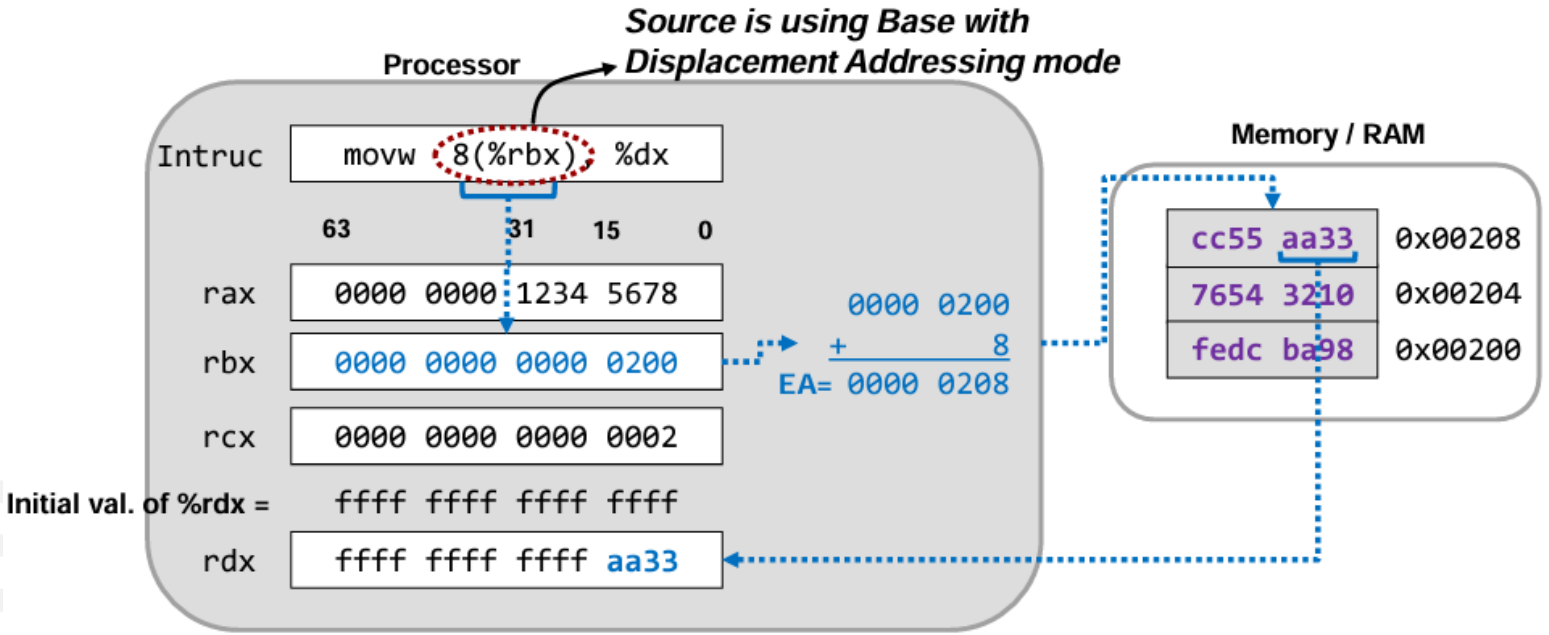
- 圆括号表示间接寻址模式



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



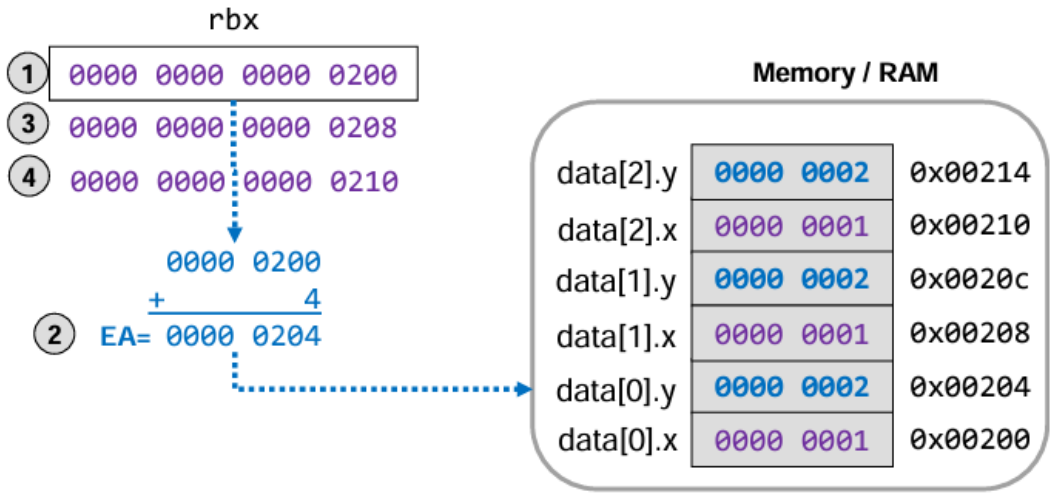
指令集的分类1 – 传输指令： Base/Indirect with Displacement Addressing Mode

- 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

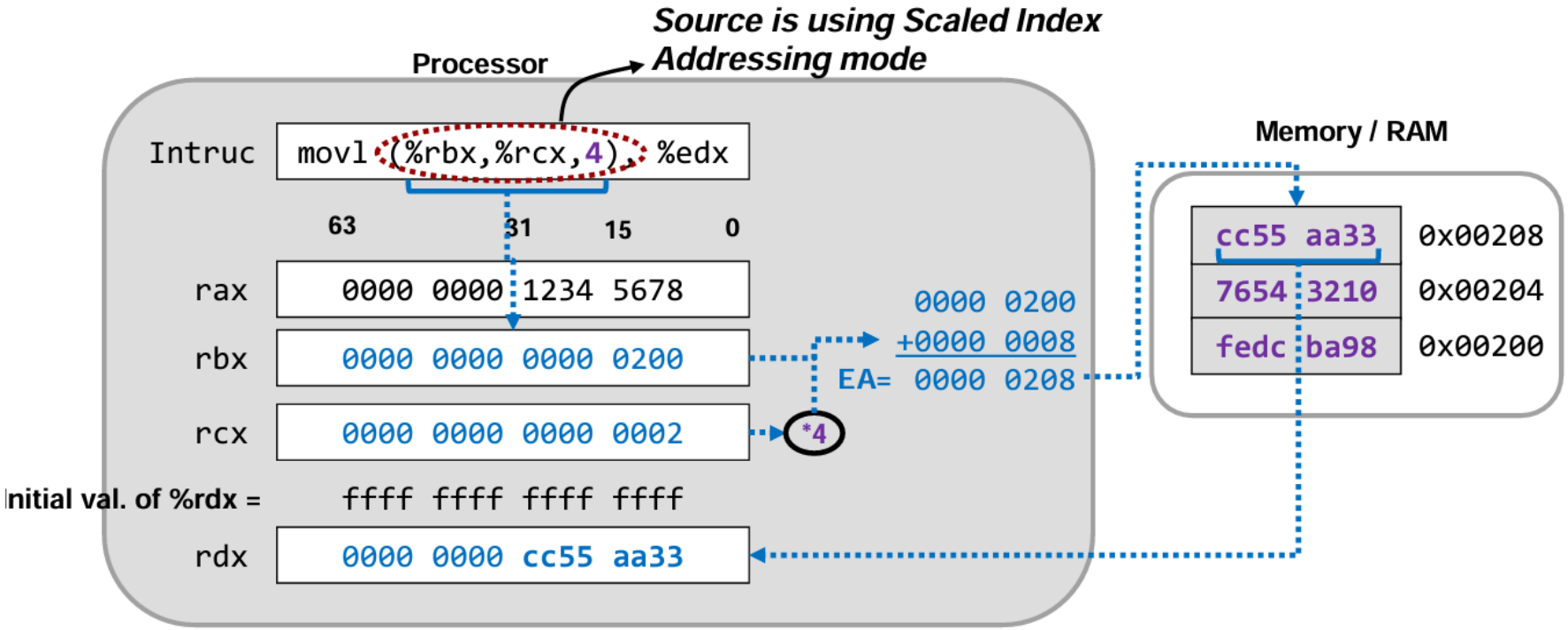



```
movq   $0x0200,%rbx
Loop 3 times {
1 3 4 movl  $1, (%rbx)
2     movl  $2, 4(%rbx)
      addq  $8, %rbx
}
```

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

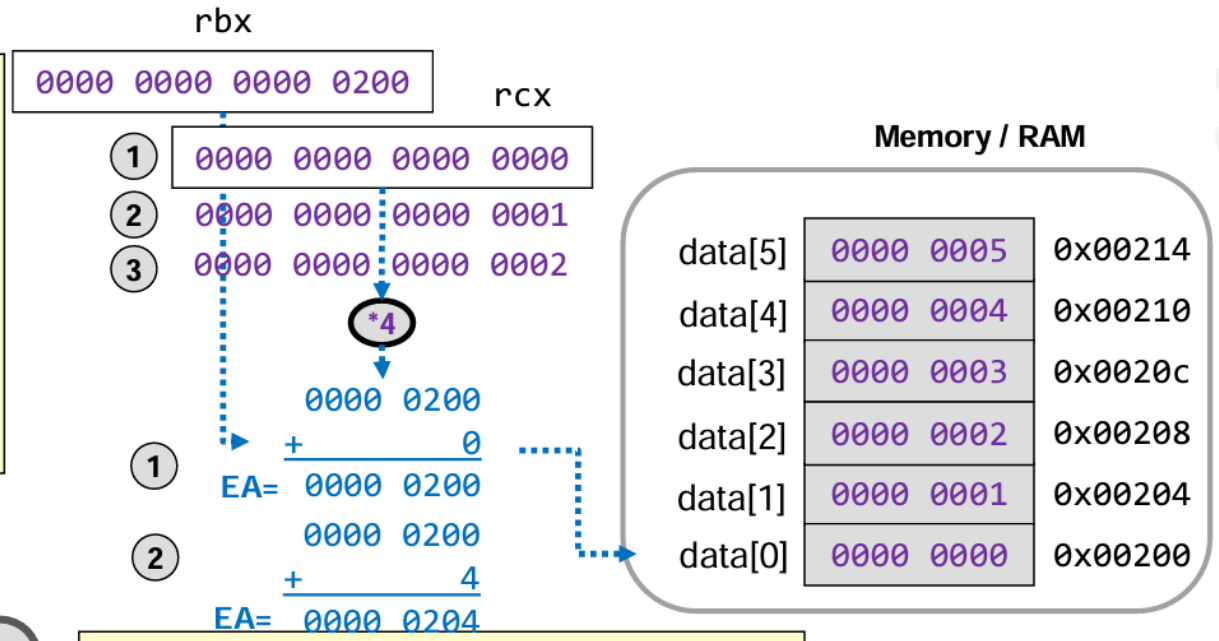


指令集的分类1 – 传输指令： Scaled Index Addressing Mode

- 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



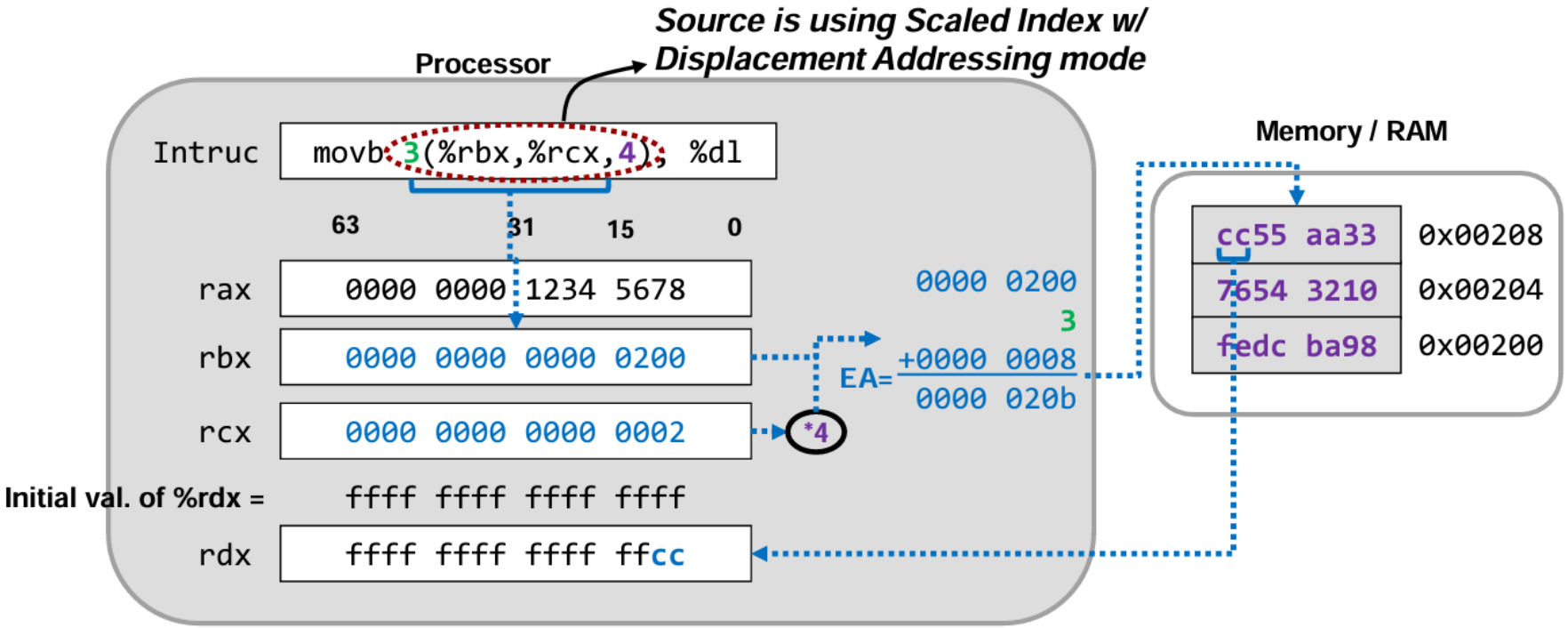
Array of:
- chars/bytes => Use s=1
- shorts/words => Use s=2
- ints/floats/dwords => Use s=4
- long longs/doubles/qwords => Use s=8

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

Assembly

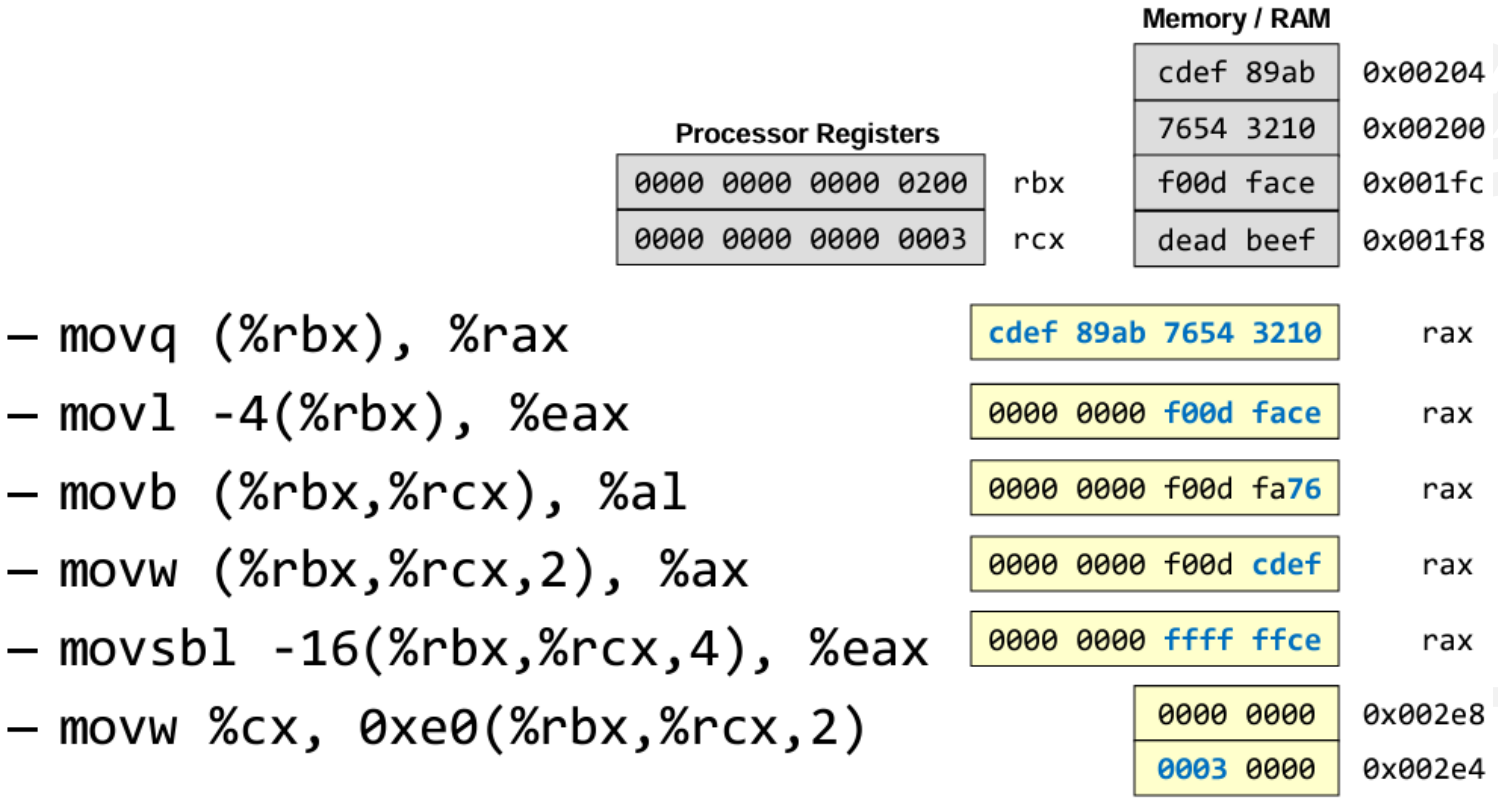
指令集的分类1 – 传输指令： Scaled Index w/ Displacement Addressing Mode

- 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

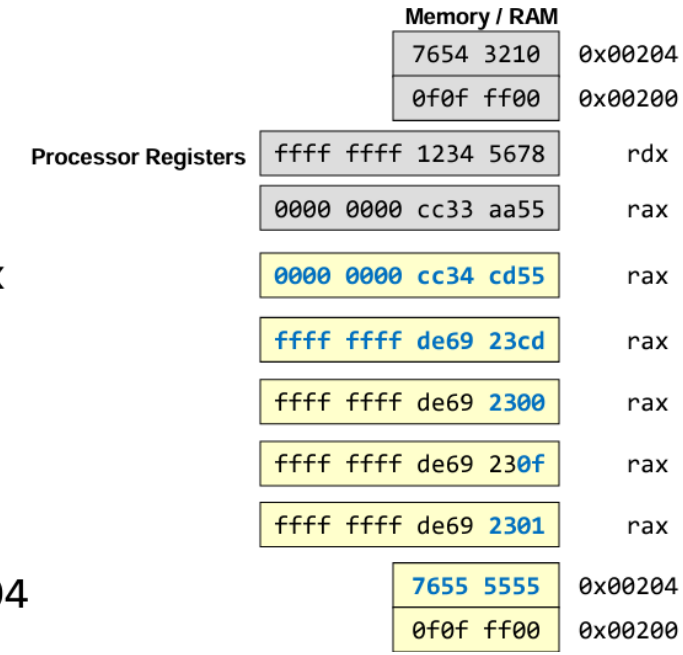
- 利用ALU来完成实际计算任务

- 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

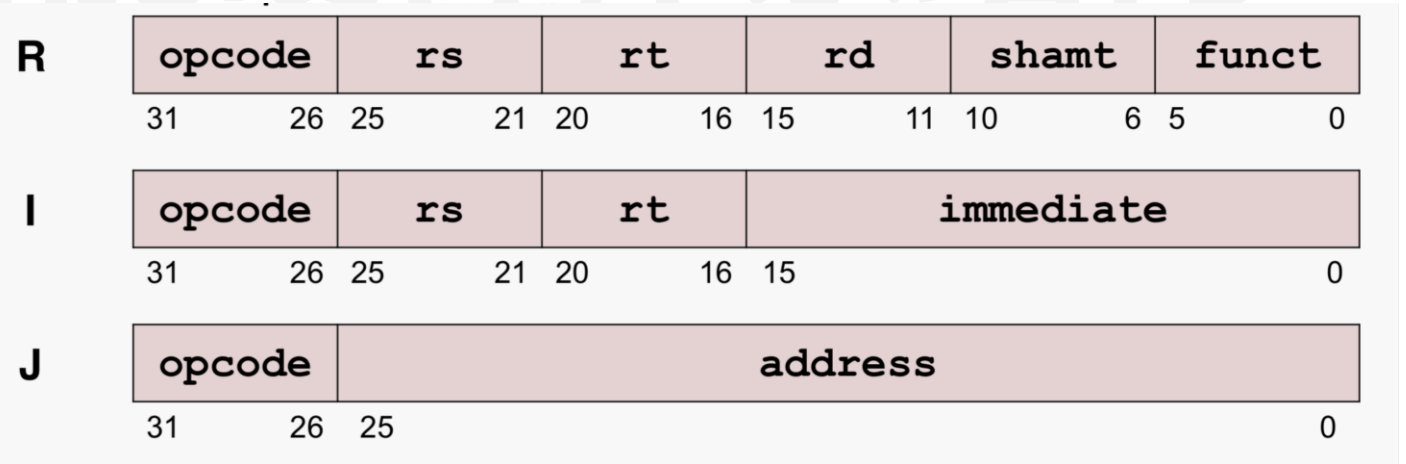
- `imm`: signed immediate in $imm_{11:0}$
- `uimm`: 5-bit unsigned immediate in $imm_{4:0}$
- `upimm`: 20 upper bits of a 32-bit immediate, in $imm_{31:12}$
- `Address`: memory address: rs1 + SignExt($imm_{11:0}$)
- `[Address]`: data at memory location Address
- `BTA`: branch target address: PC + SignExt({$imm_{12:1}$, 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)

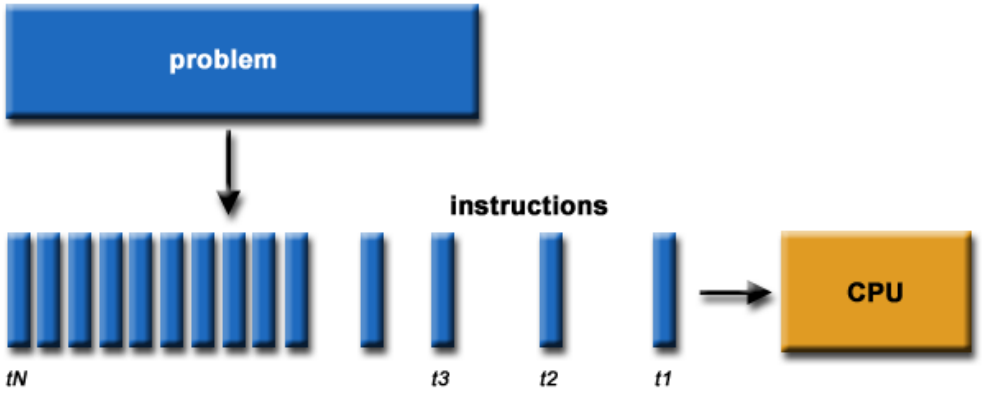
•Opcode

- 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

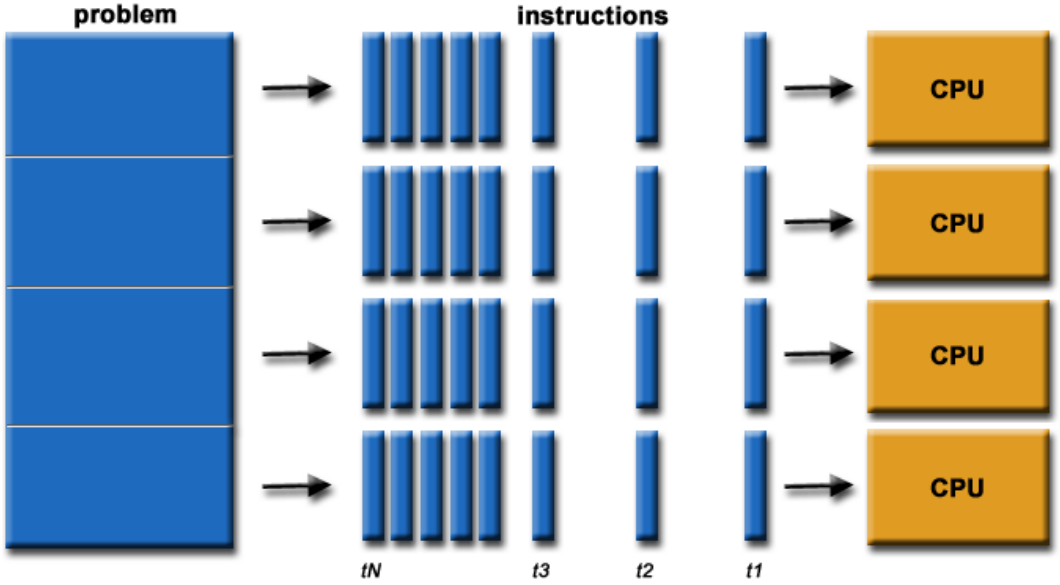


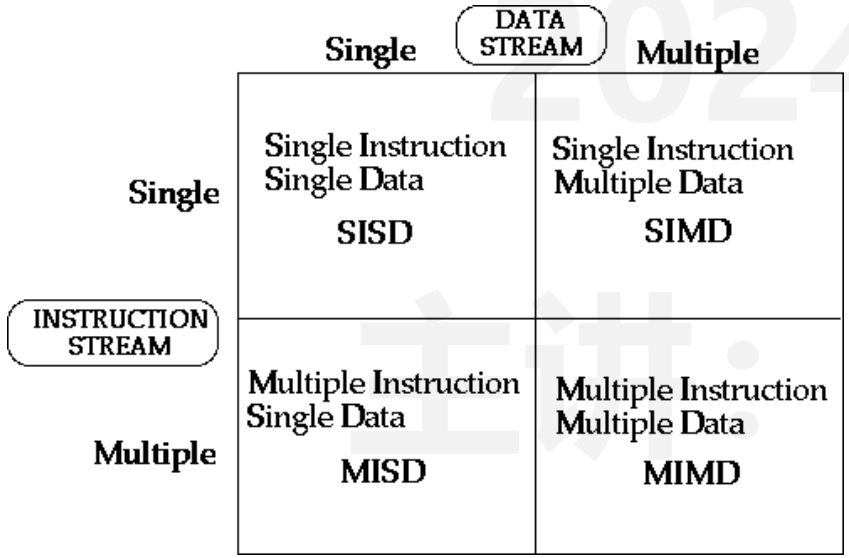
代表性指令集： GPU的CUDA指令集

- SISD、SIMD、MISD、MIMD的扩展

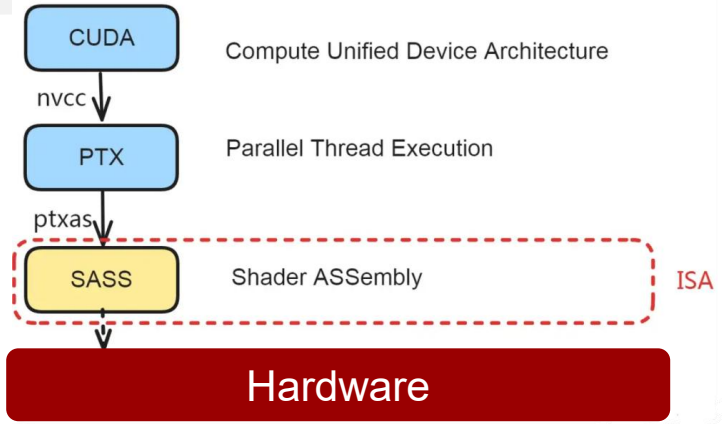


并行计算



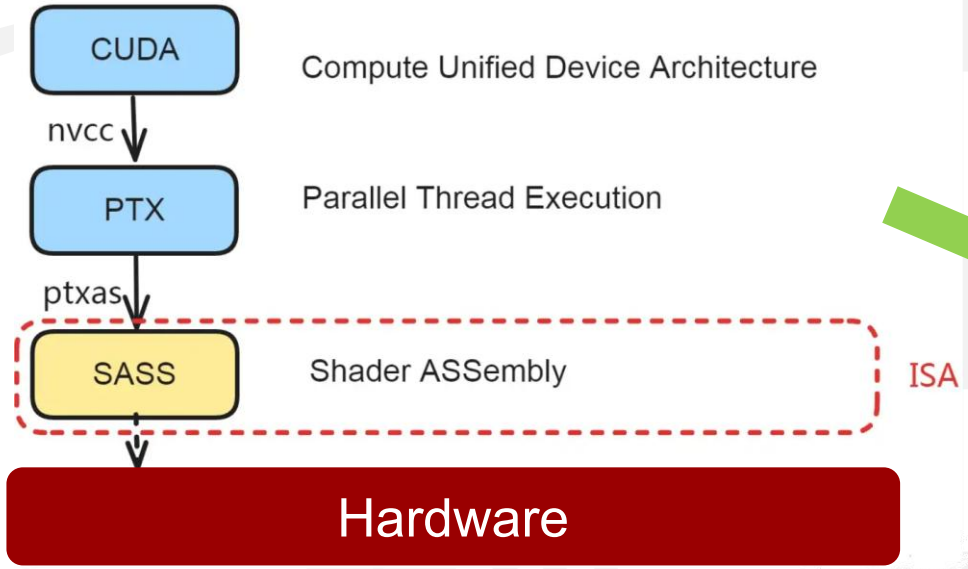


CUDA指令集的层次



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

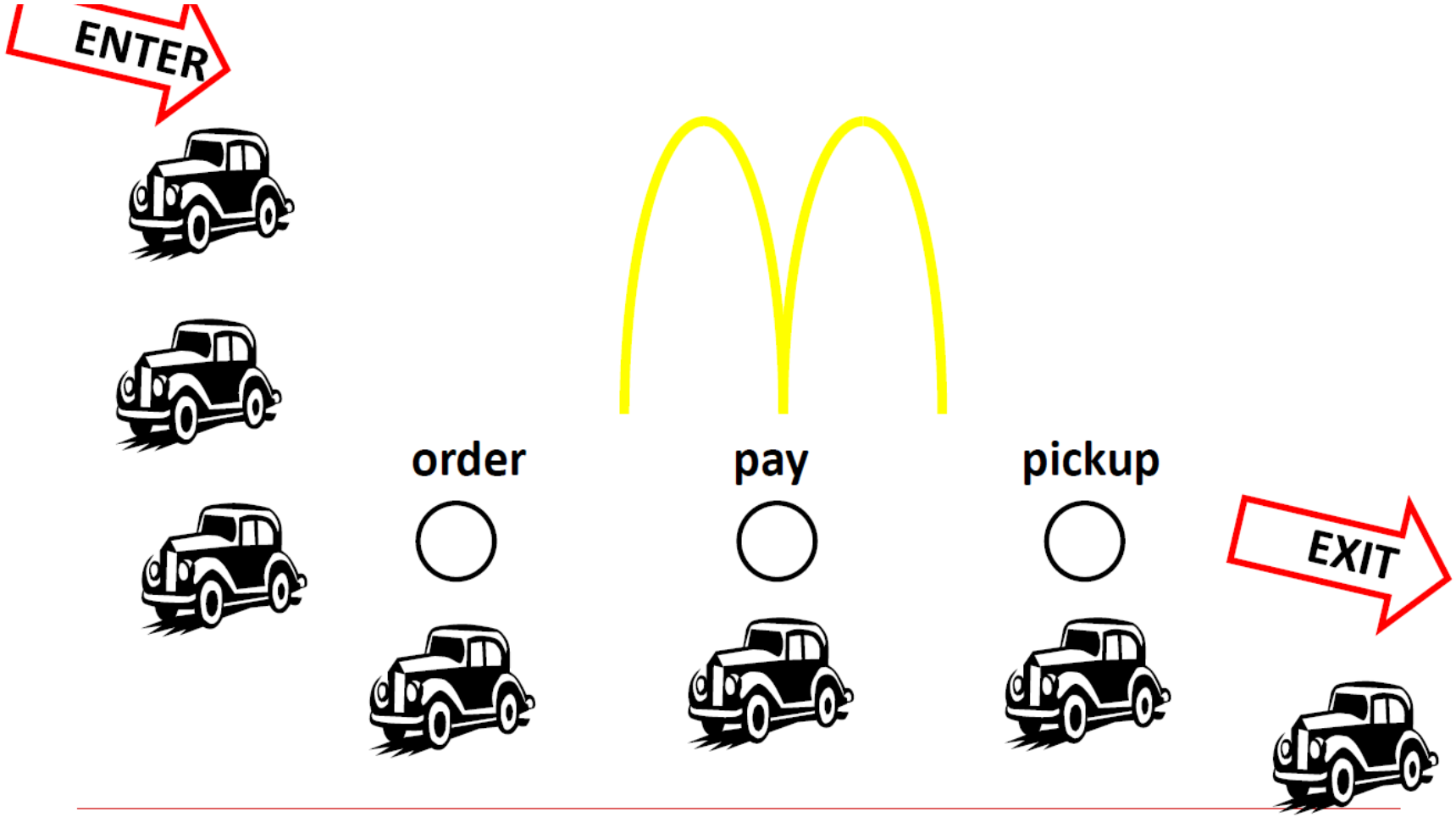
Div / Rem Instruction			Pop Instruction		
rem/div.u16/s16	multiple instructions	290	popc.b32S	POPC	6
rem/div.s32/u32	multiple instructions	66	popc.b64	2*UPOPC + UIADD3	7
rem/div.u64/s64	multiple instructions	420	**Clz Instruction**		
div.rn.f32	multiple instructions	525	clz.b32	FLO.U32 + IADD	7
div.rn.f64	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			

目 录

CONTENTS

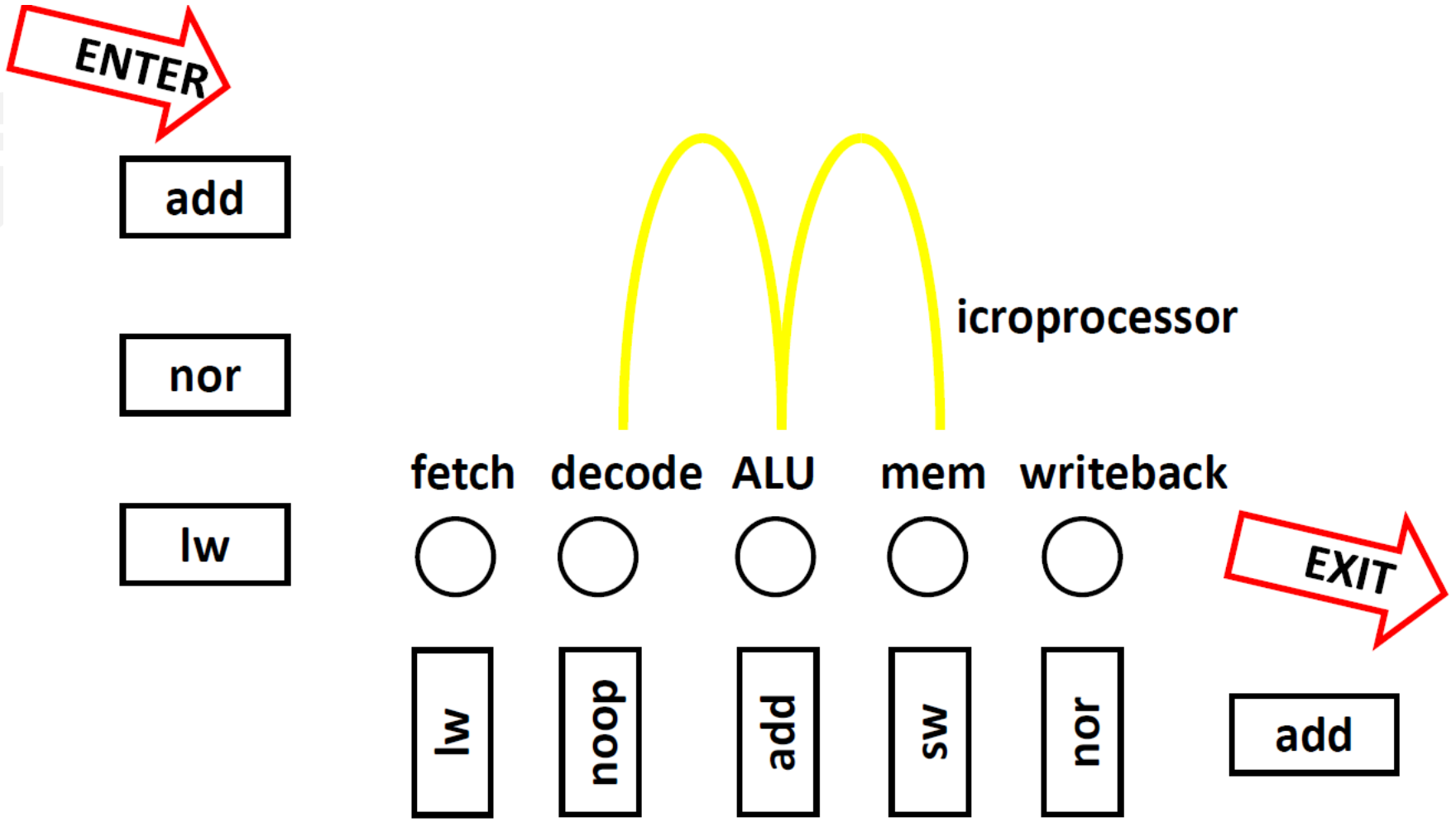
回顾：什么是流水线架构

- 流水线式运行方式 - 提高吞吐率的有效手段



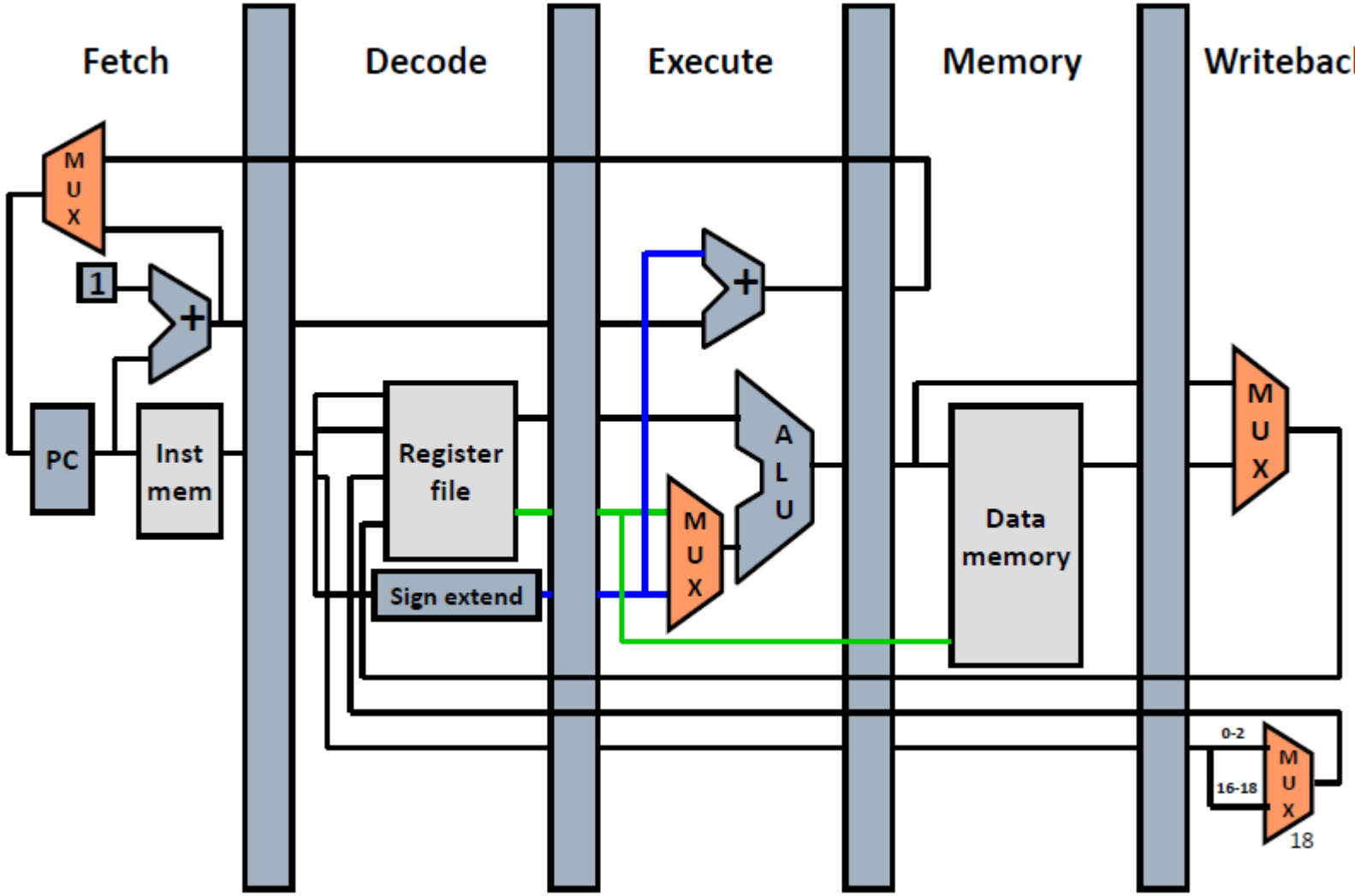
回顾：什么是流水线架构

- 流水线式运行方式 - 提高吞吐率的有效手段（提高instruction/cycle，CPI）



最基本的单条流水线设计示意图

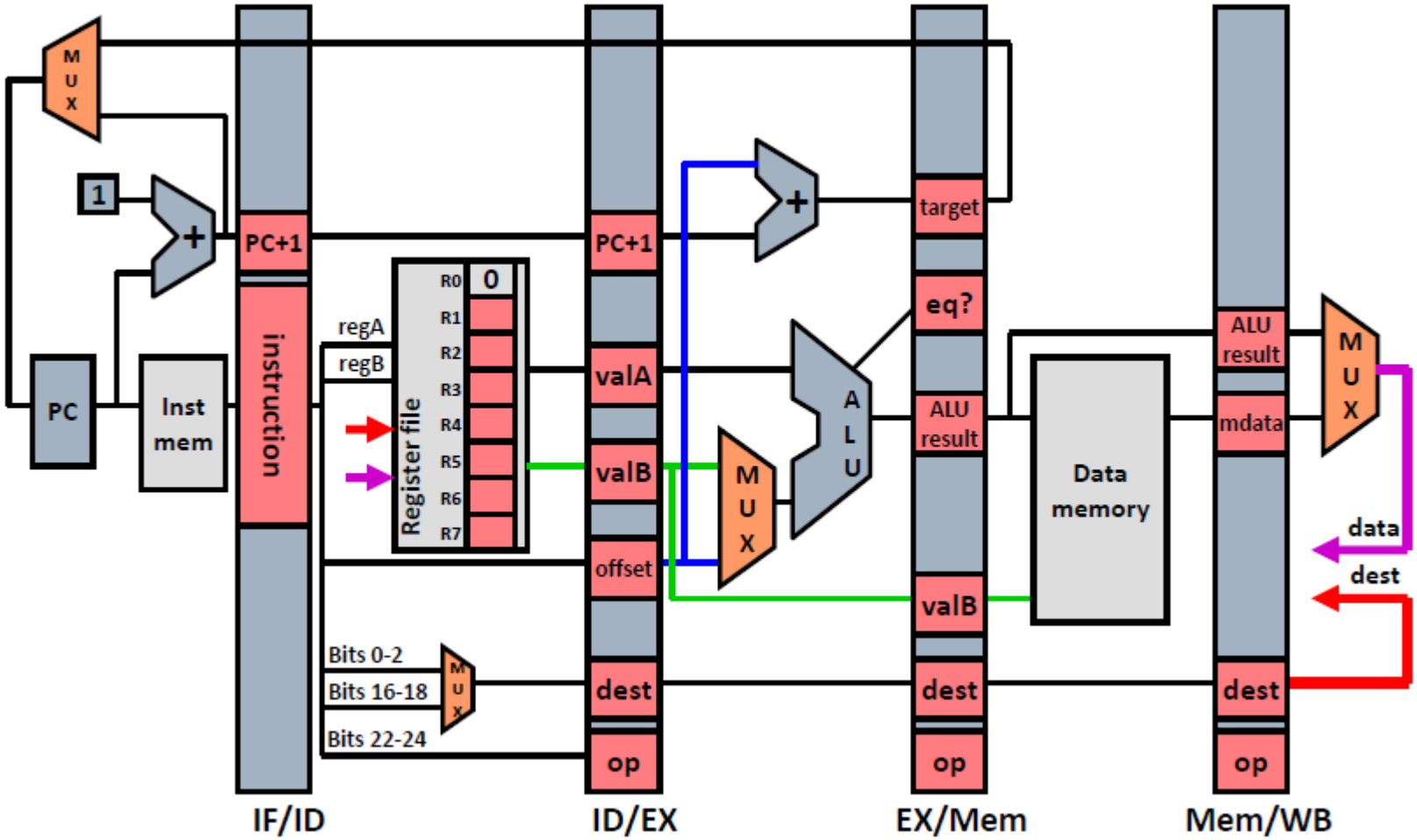
- 5级流水线设计：Fetch、Decode、Execute、Memory、Writeback



5级流水线的实际案例

- 假设运行以下指令在5级流水线上

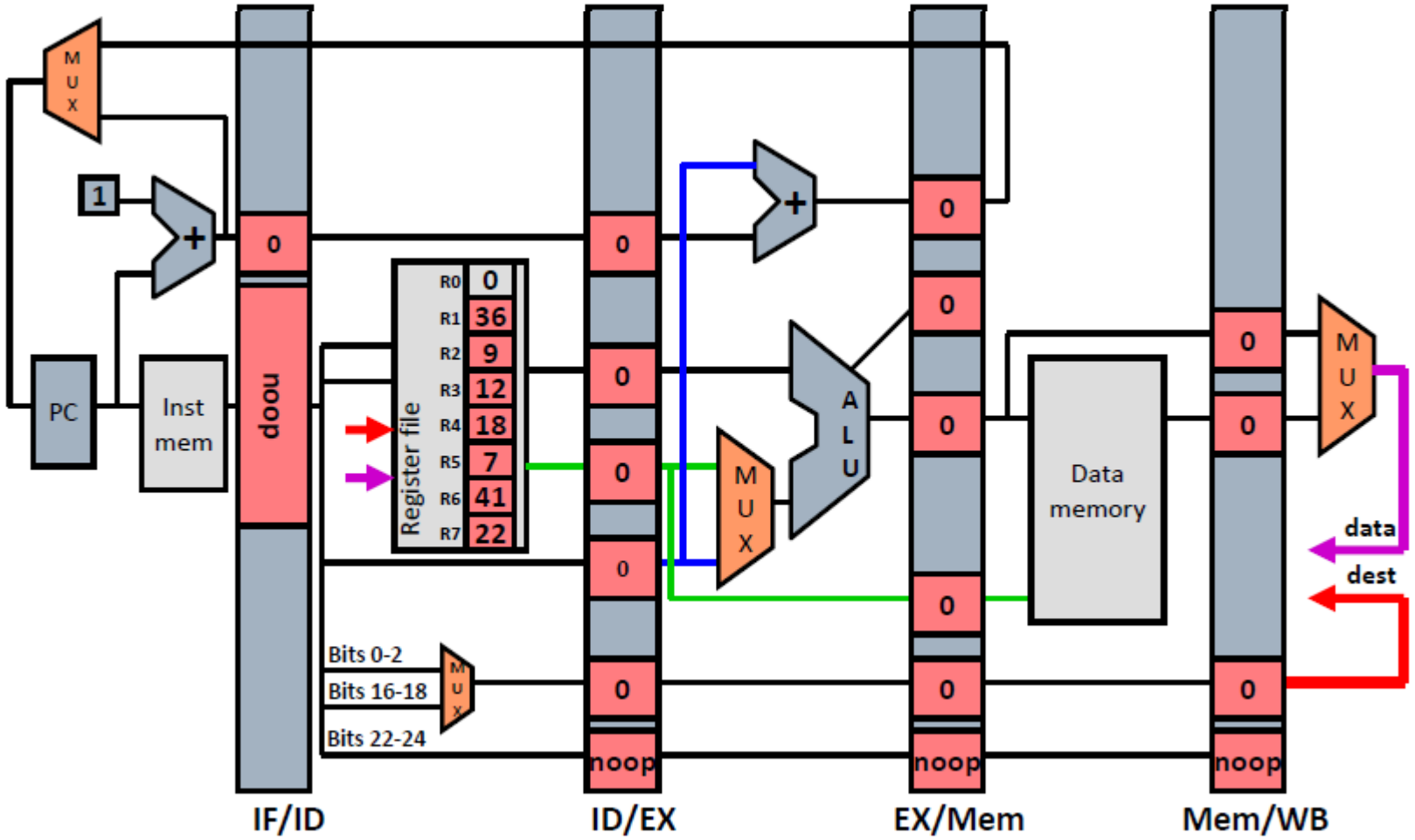
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



5级流水线的实际案例

- 假设运行以下指令在5级流水线上

```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```

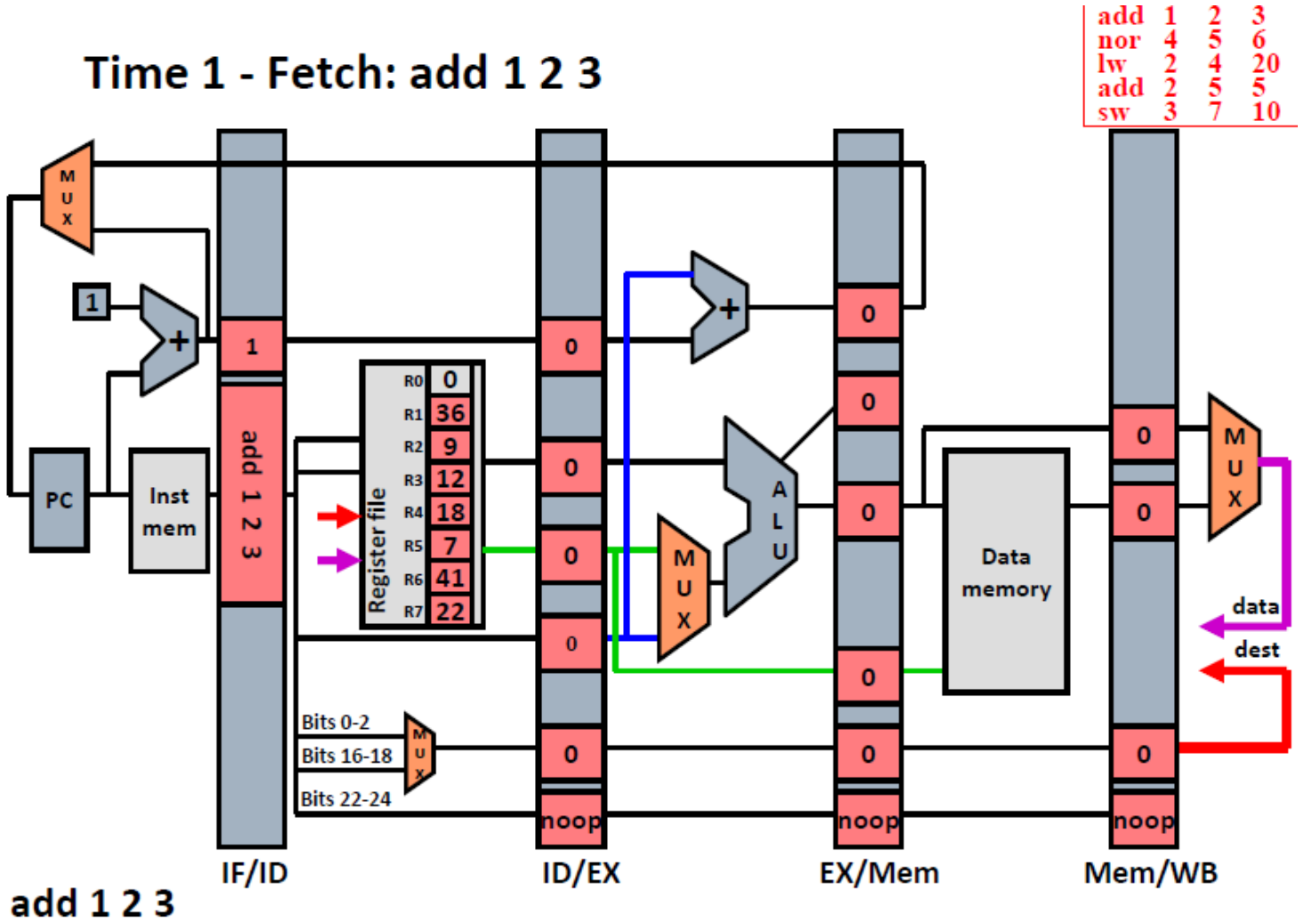


初始状态：t0

5级流水线的实际案例

- 假设运行以下指令在5级流水线上

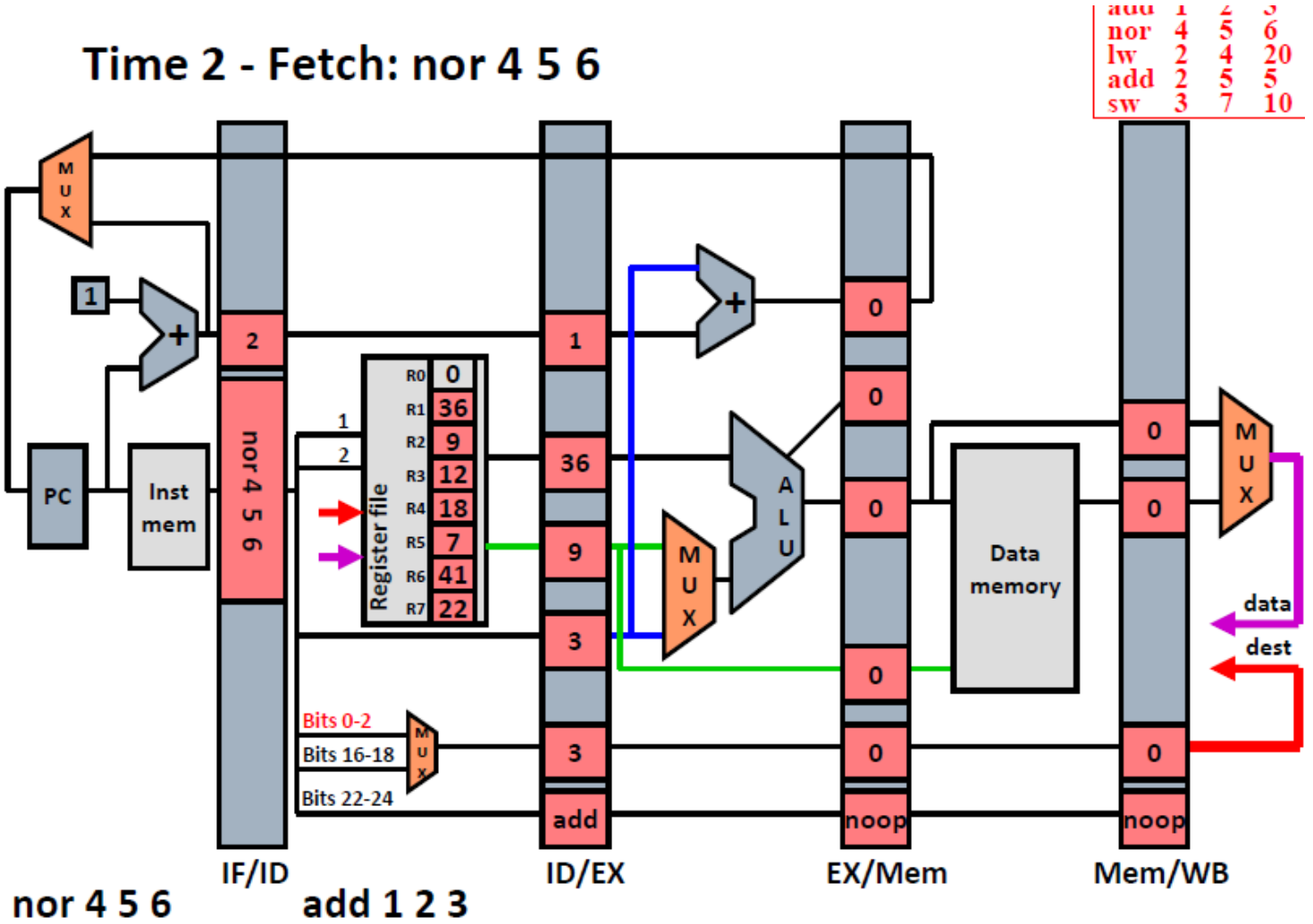
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



5级流水线的实际案例

- 假设运行以下指令在5级流水线上

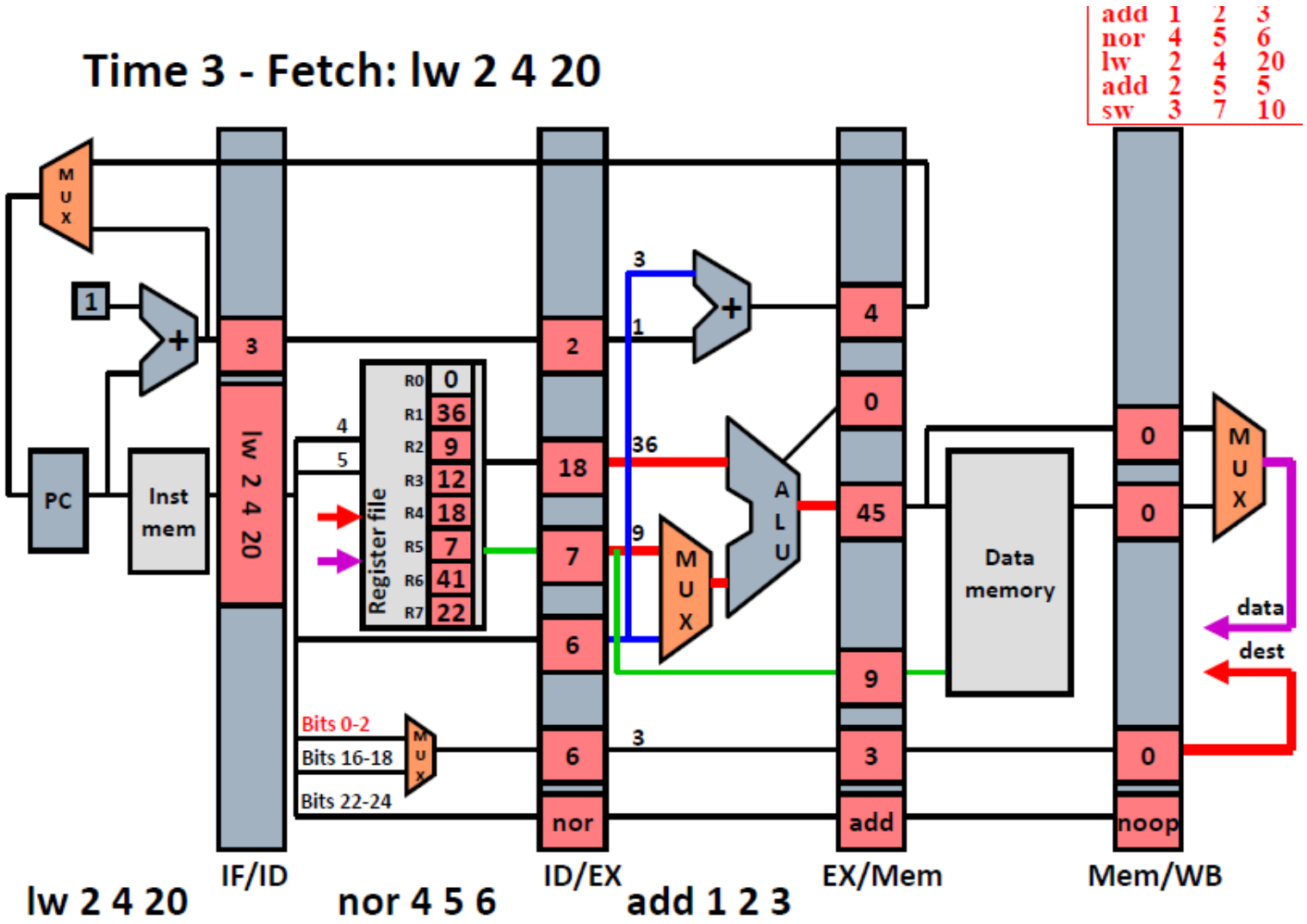
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



5级流水线的实际案例

- 假设运行以下指令在5级流水线上

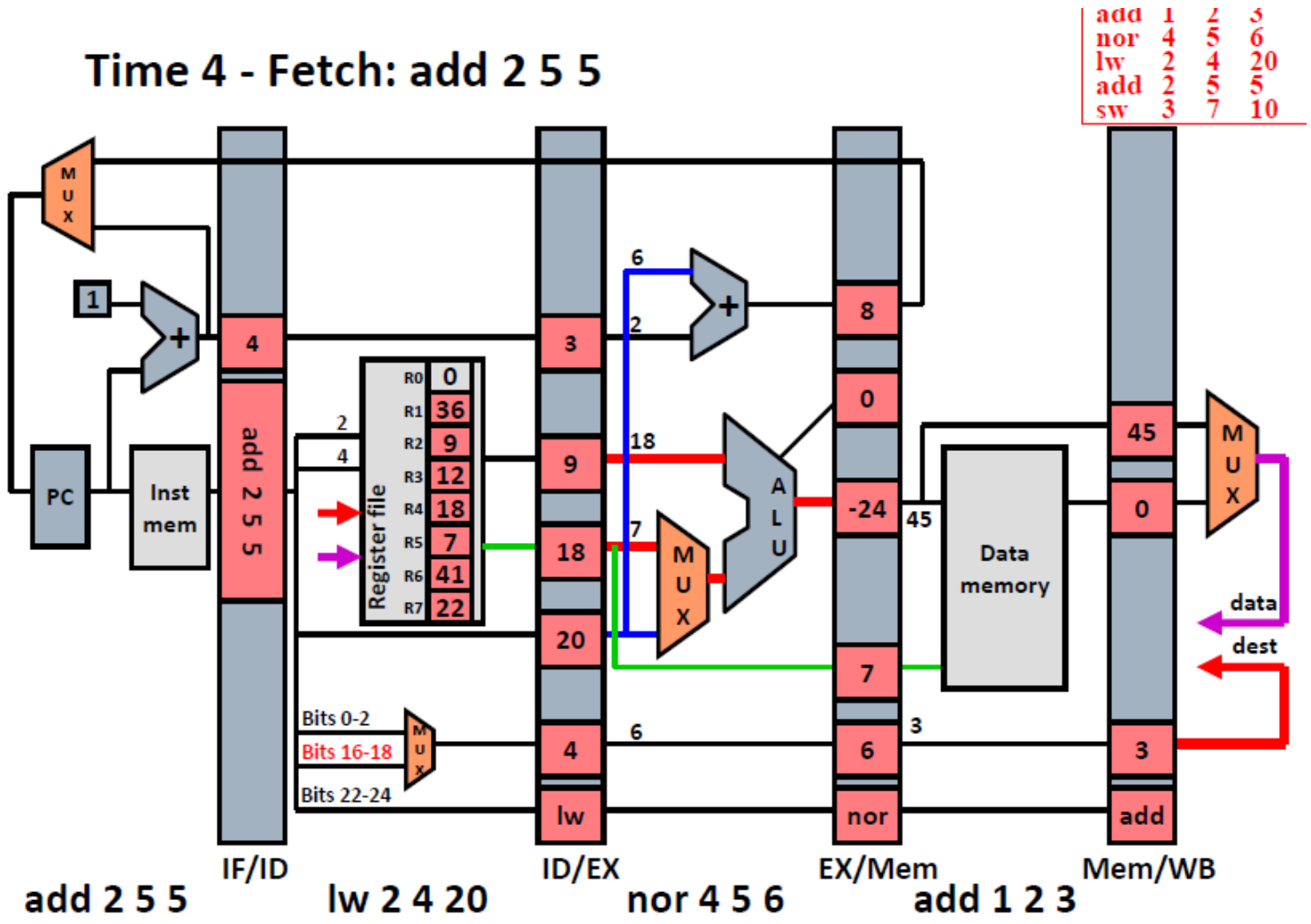
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



5级流水线的实际案例

- 假设运行以下指令在5级流水线上

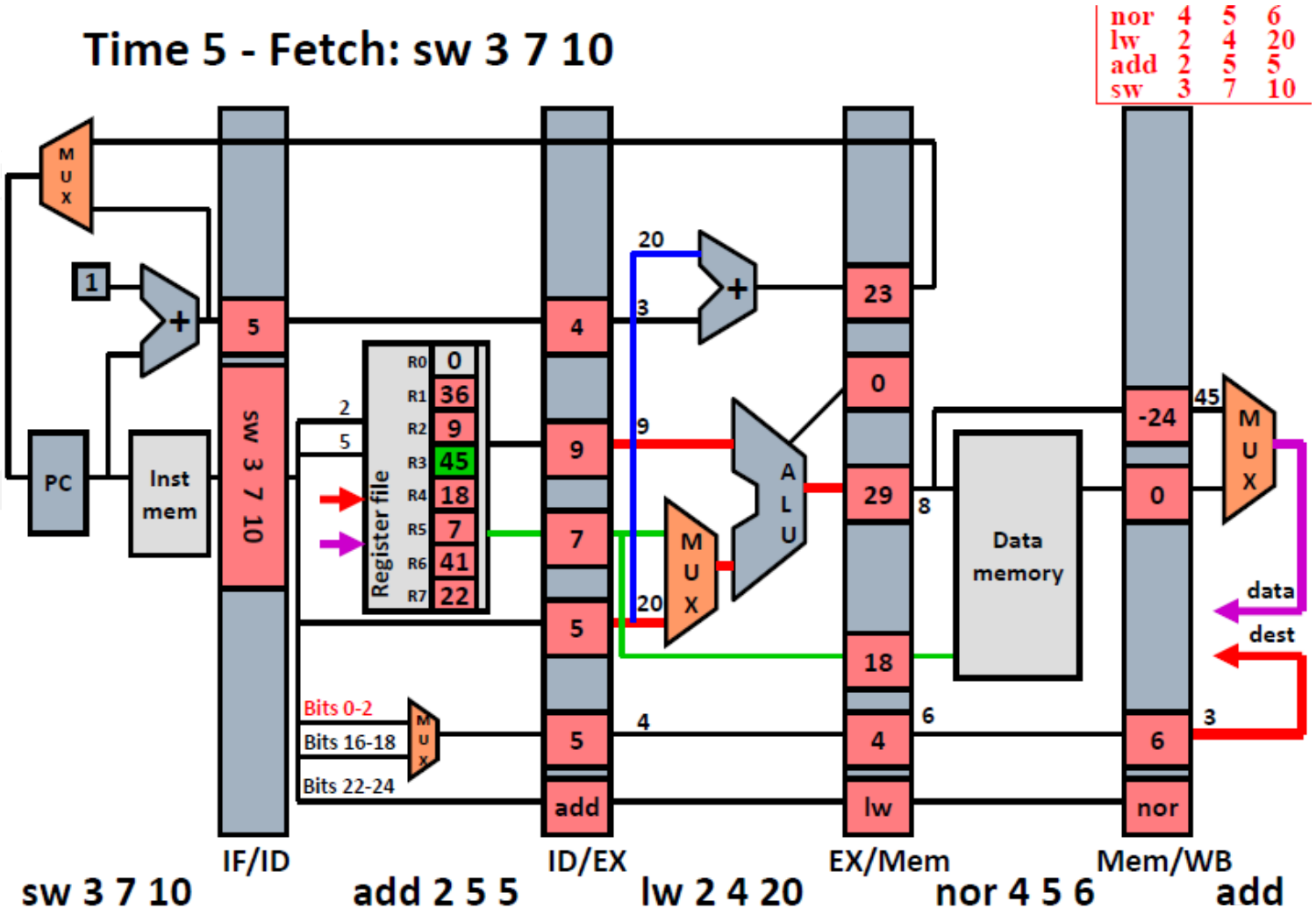
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



5级流水线的实际案例

- 假设运行以下指令在5级流水线上

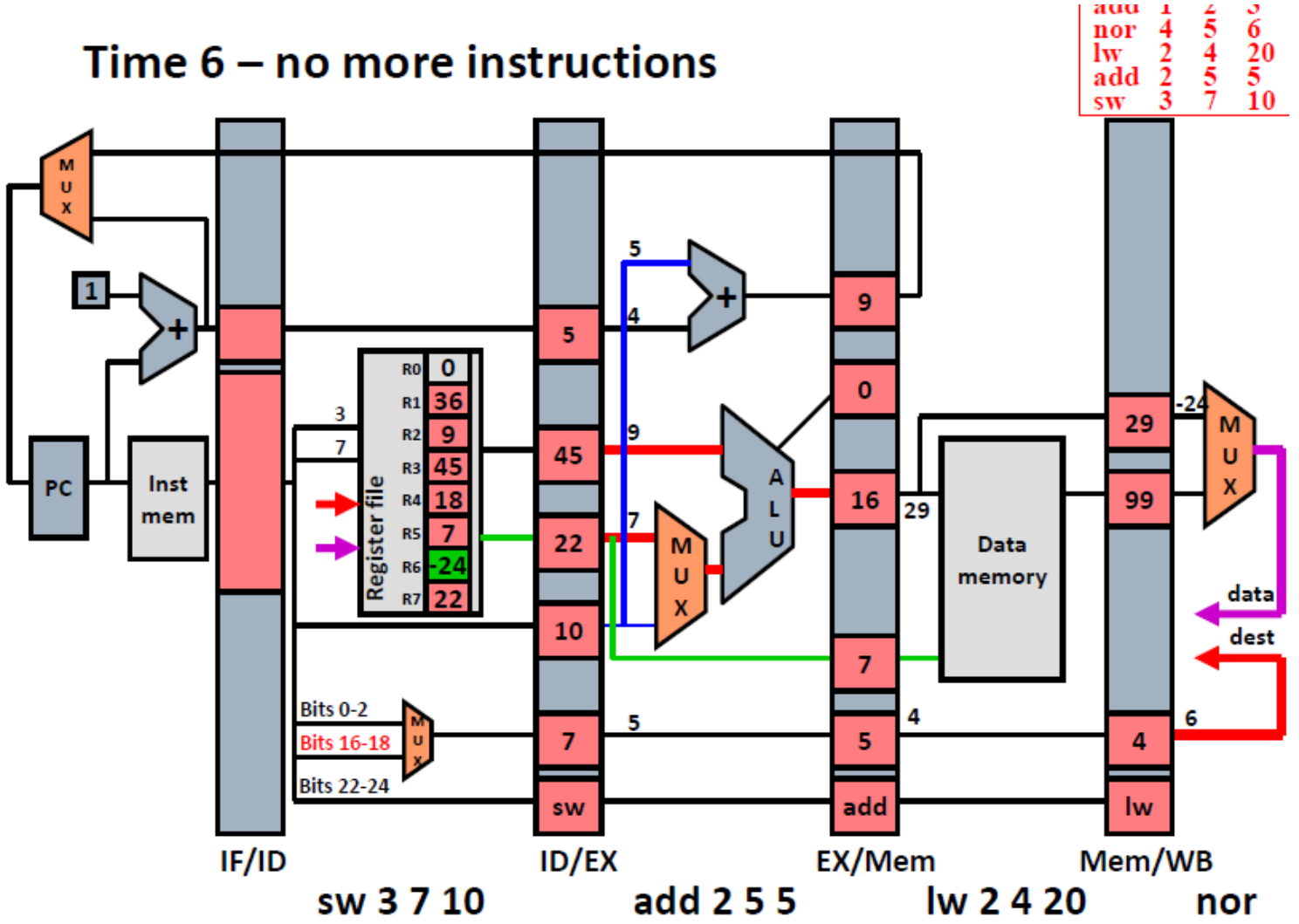
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



5级流水线的实际案例

- 假设运行以下指令在5级流水线上

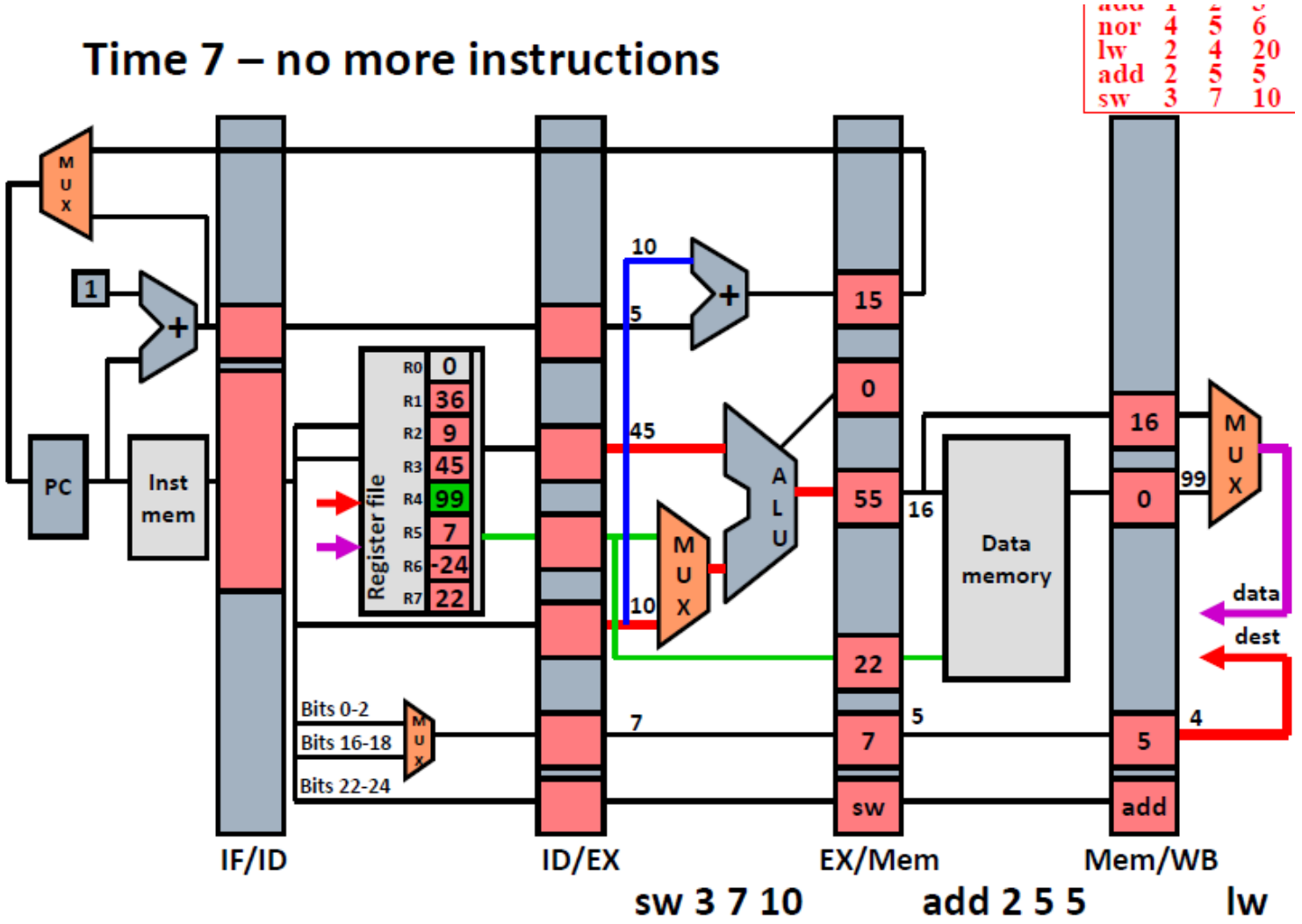
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



5级流水线的实际案例

- 假设运行以下指令在5级流水线上

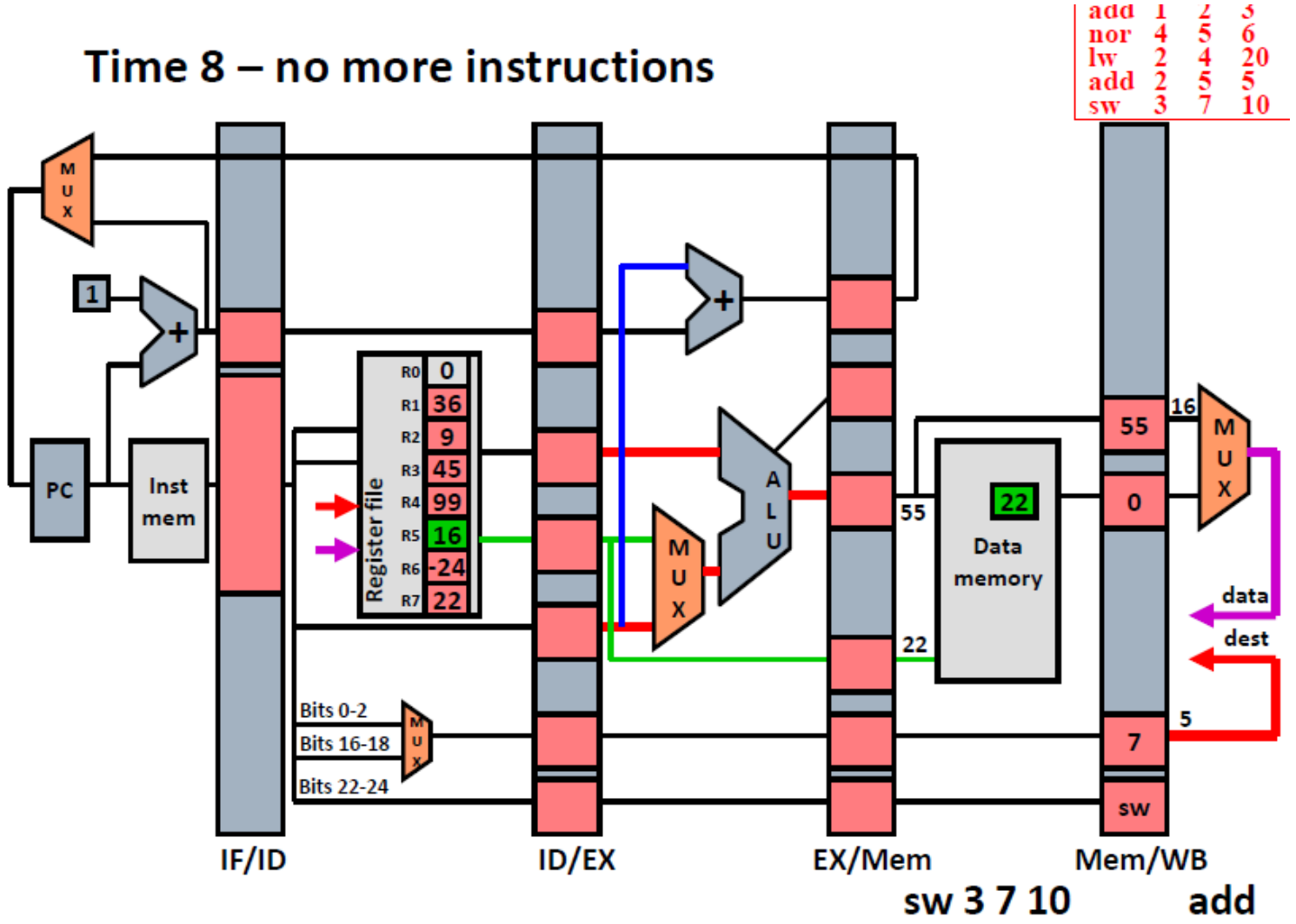
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



5级流水线的实际案例

- 假设运行以下指令在5级流水线上

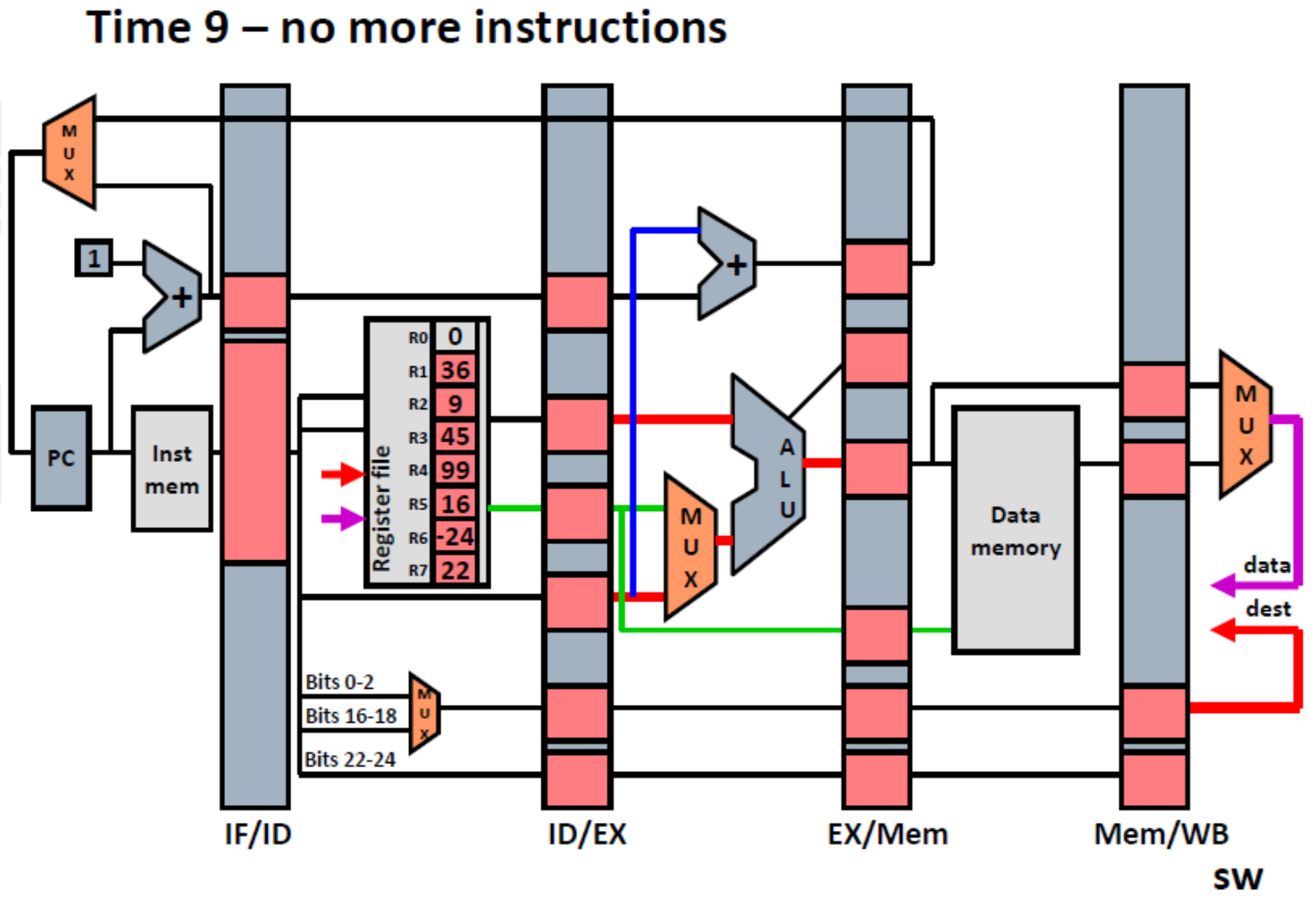
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



5级流水线的实际案例

- 假设运行以下指令在5级流水线上

```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



5级流水线的实际案例

- 假设运行以下指令在5级流水线上

```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```