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

第六讲：指令集与流水线架构

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）

- **第1次lab时间：4月13日-5月13日**
- **第2次lab时间：5月13日-6月13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

- **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**

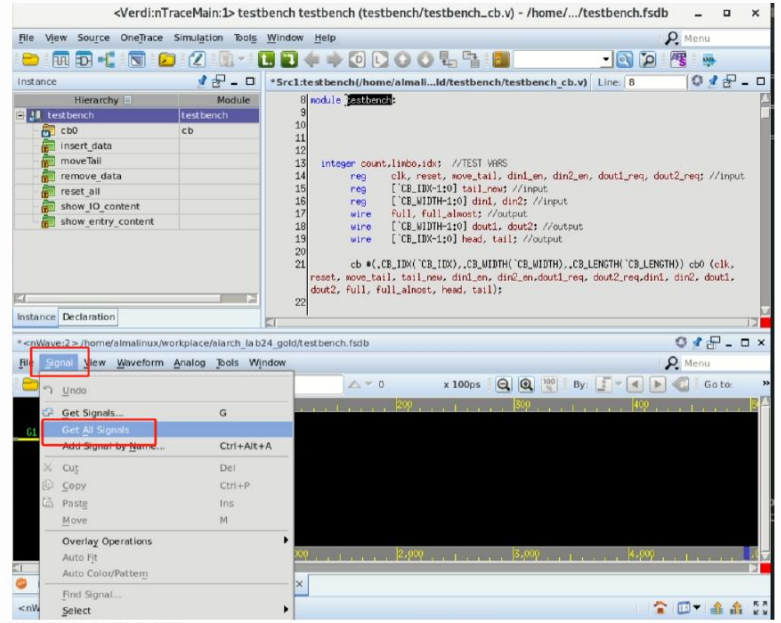
进入 `run` 目录，运行 `make rerun` 即可运行模拟，同时不会显示波形，由于该操作不依赖于GUI，可以在VS Code中直接使用。由于在testbench中设置了 `$monitor` 命令，会输出不同时刻的计算结果，可以用于验证结果是否正确。

```
[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

主流指令集都有哪些?

- X86、MIPS、ARM、RISC-V

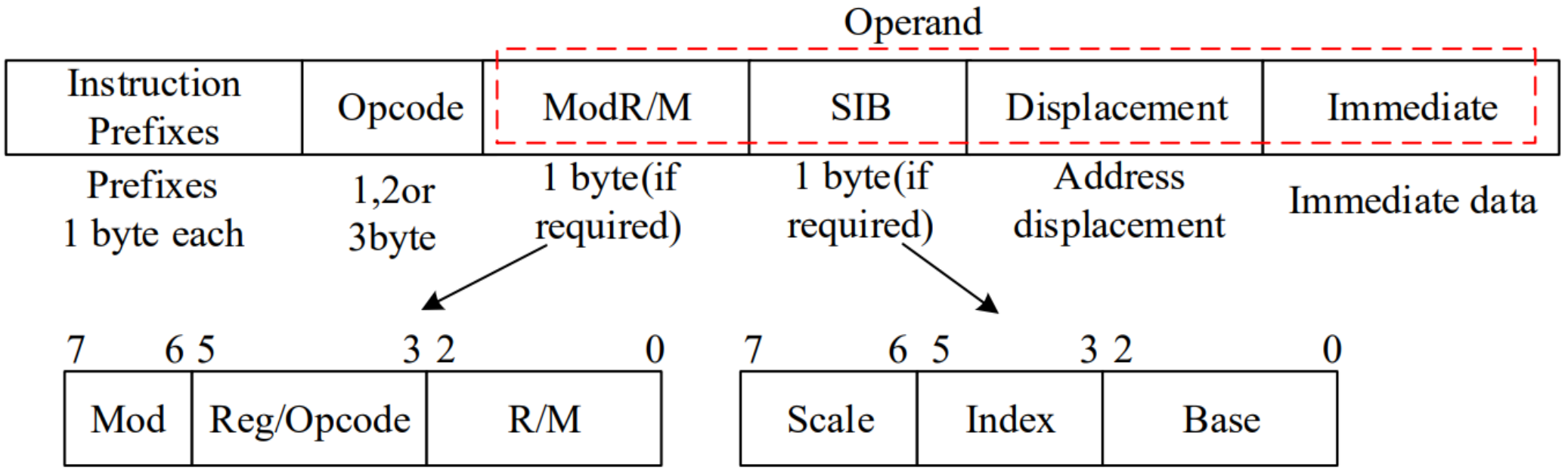
指令集	类型	运营公司	特点	代表厂商
X86	CISC	Intel、AMD	功能强大、通用性、兼容性、实用性	Intel、AMD
MIPS	RISC	MIPS	简洁、优化、高扩展性、寄存器多	Intel、IBM、龙芯、Oracle、Toshiba
ARM	RISC	ARM	低功耗、低成本、适用于移动设备	苹果、华为、谷歌
RISC-V	RISC	RISC-V基金会	完全开源、架构简单、移植性高、开源工具链	数百家大学、科研机构和企业
CUDA	RISC	Nvidia	张量高并发处理、图像处理	Nvidia

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.)

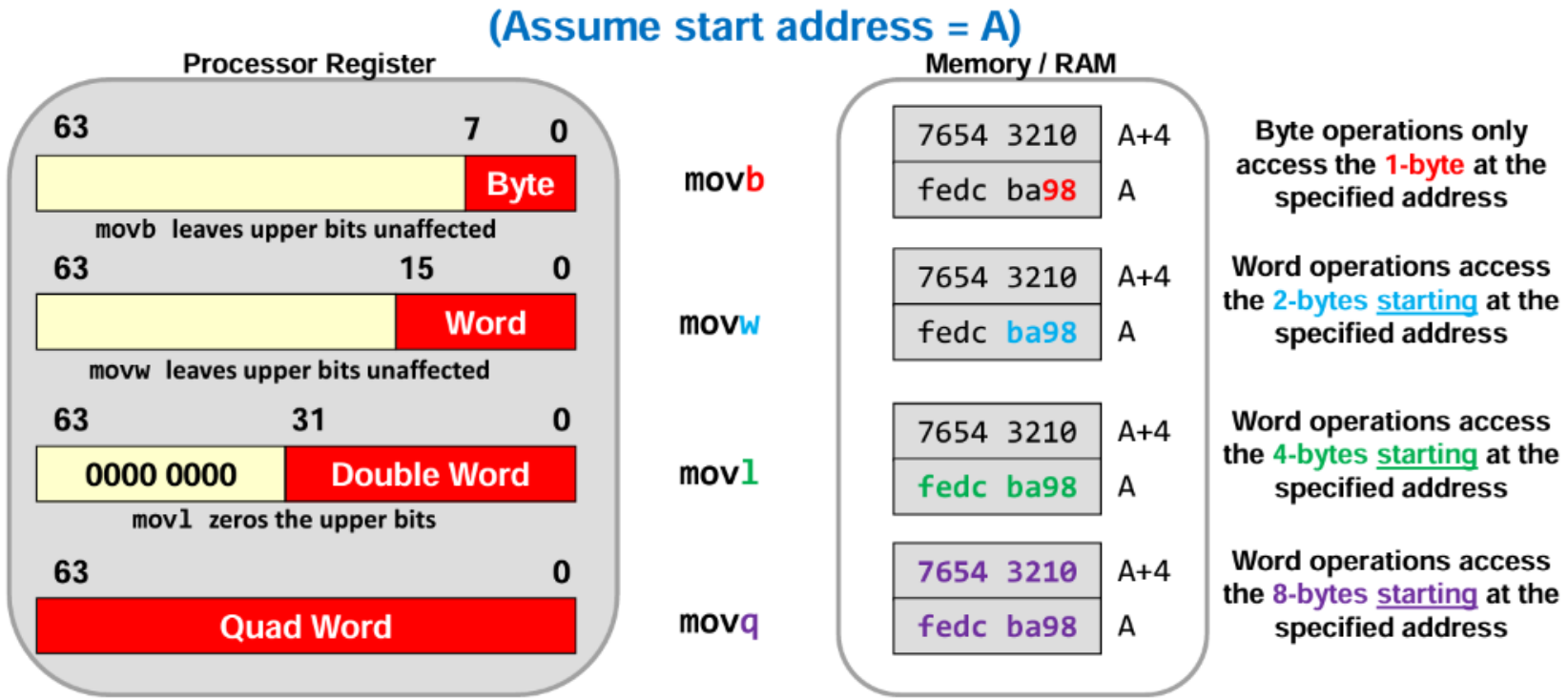
代表性指令集：X86一种典型的CISC指令

- 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]

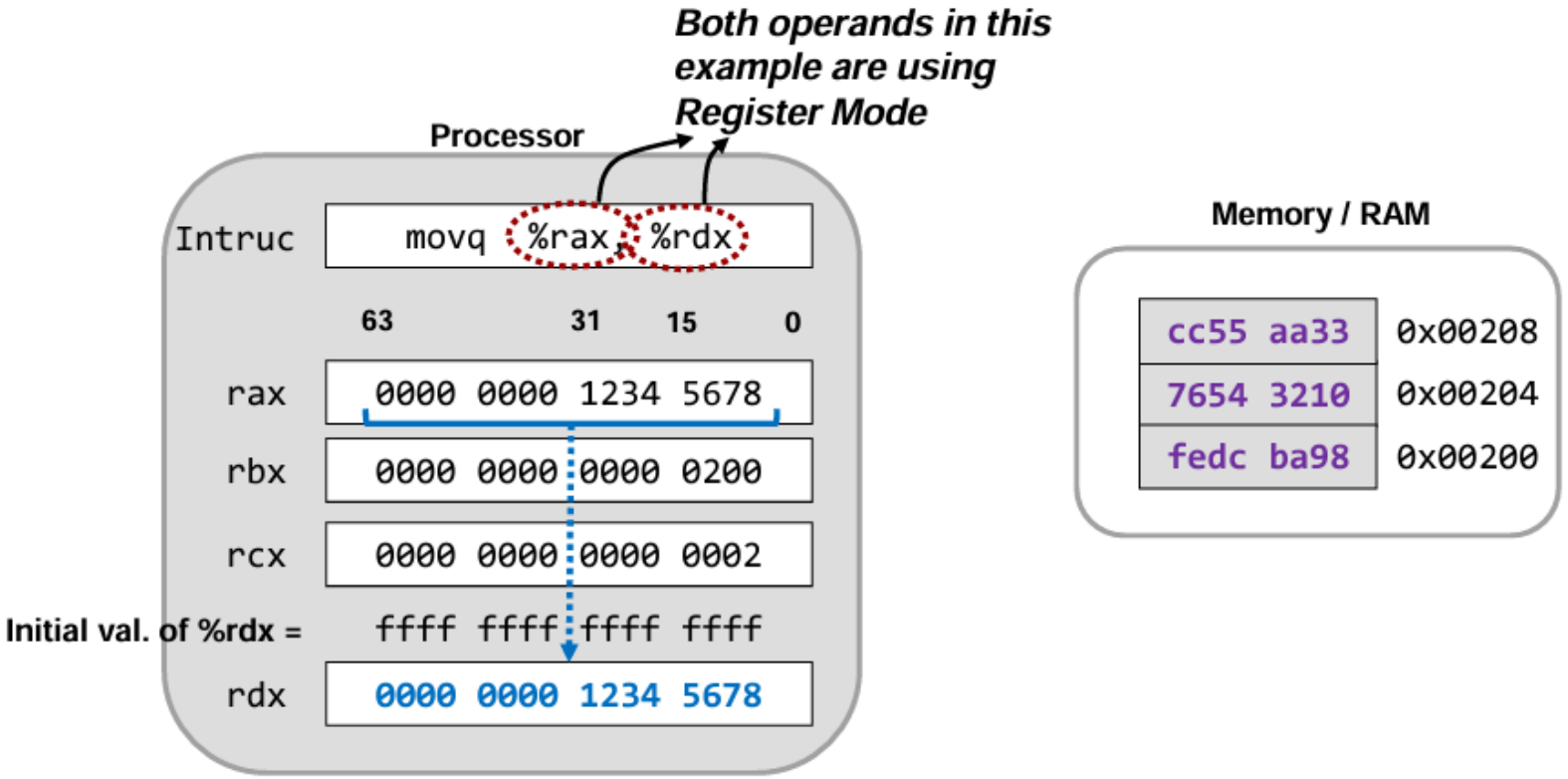
†Known as the scale factor and can be {1,2,4, or 8}

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

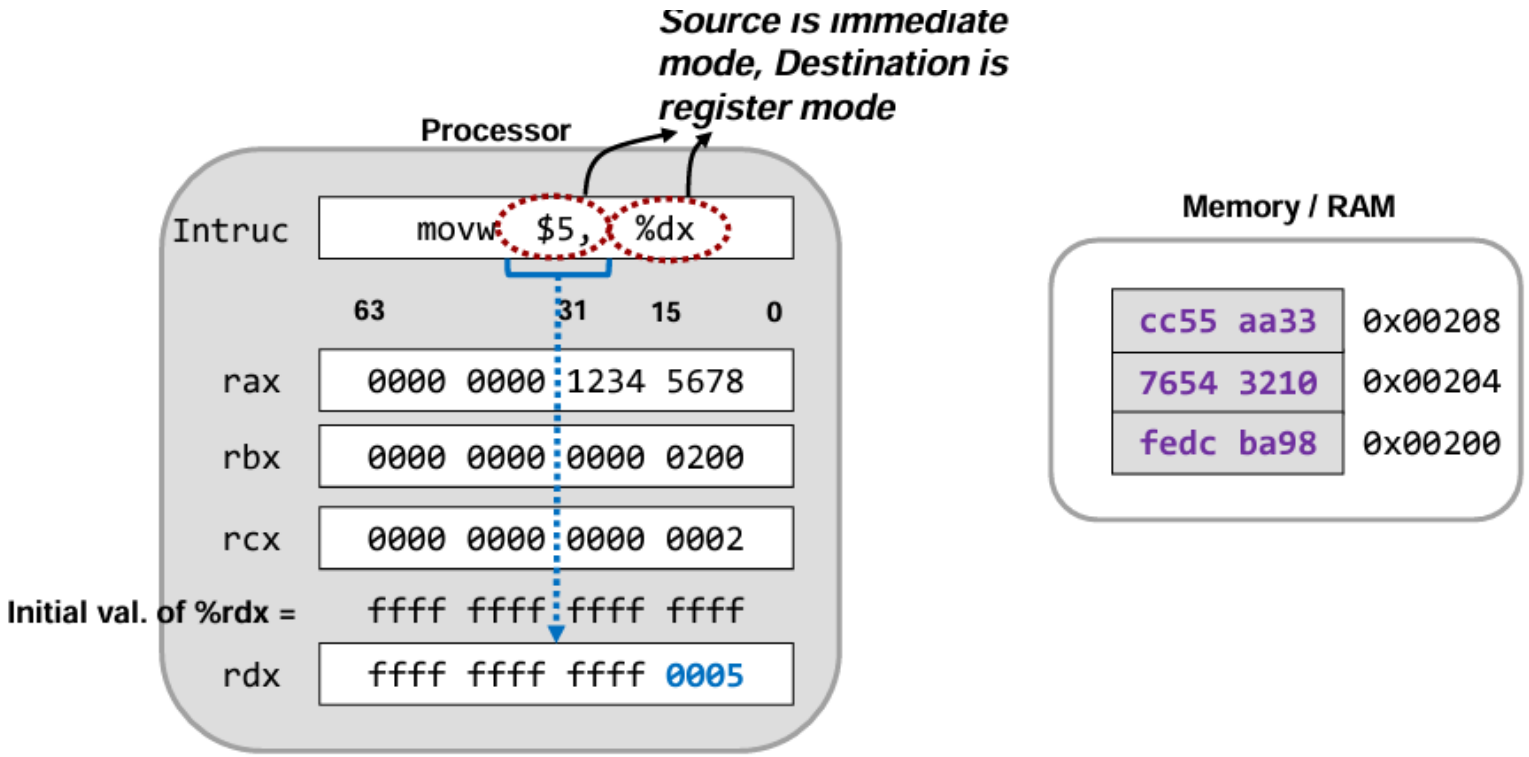
- 指定寄存器的内容作为操作数



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

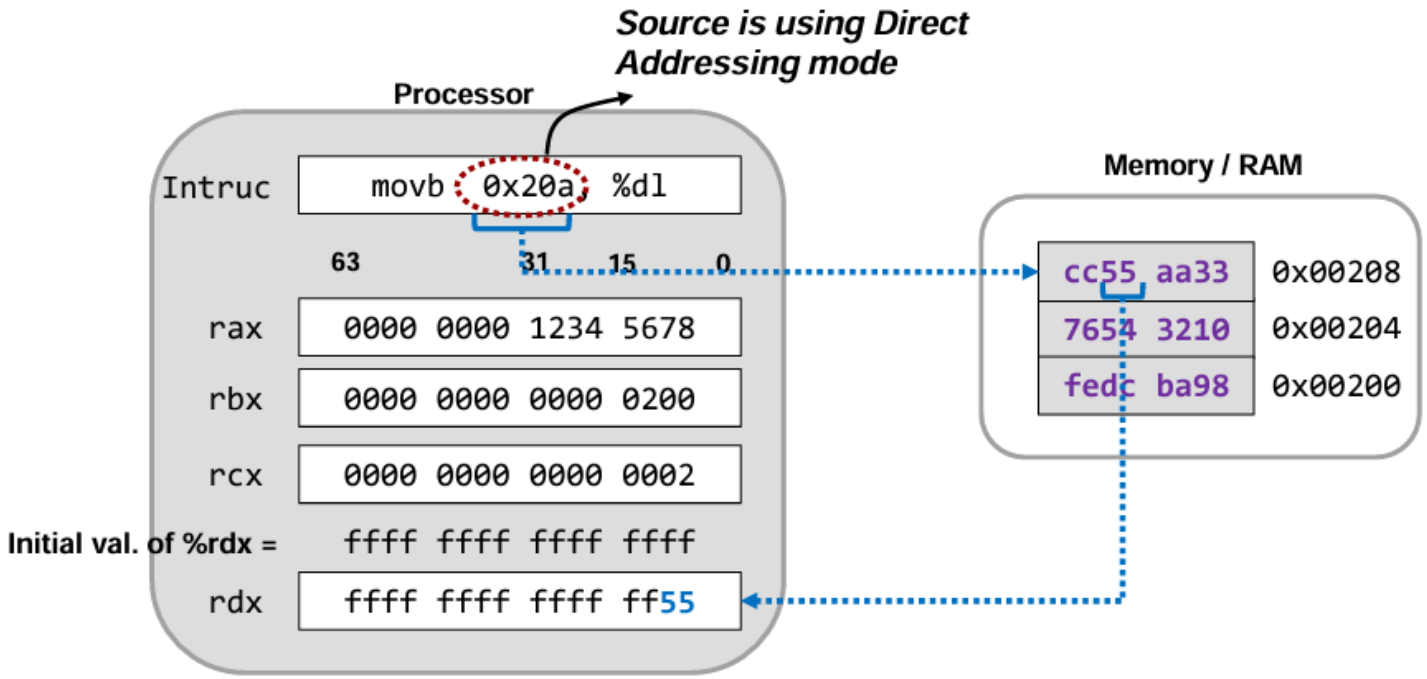
  - 符号’ $’ 表示立即数，并且可以指定用十六进制或是十进制



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

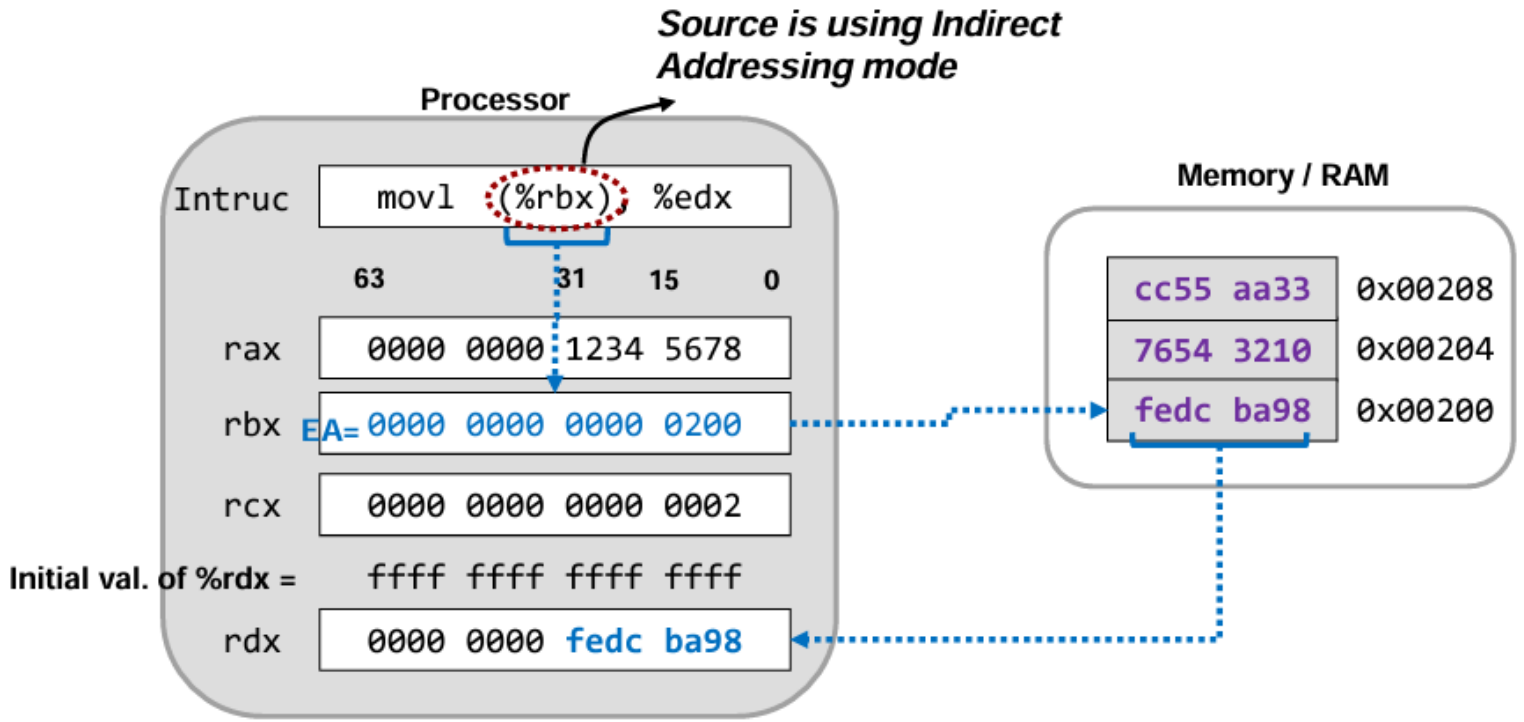


指令集的分类1 – 传输指令： Indirect Addressing Mode

- 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

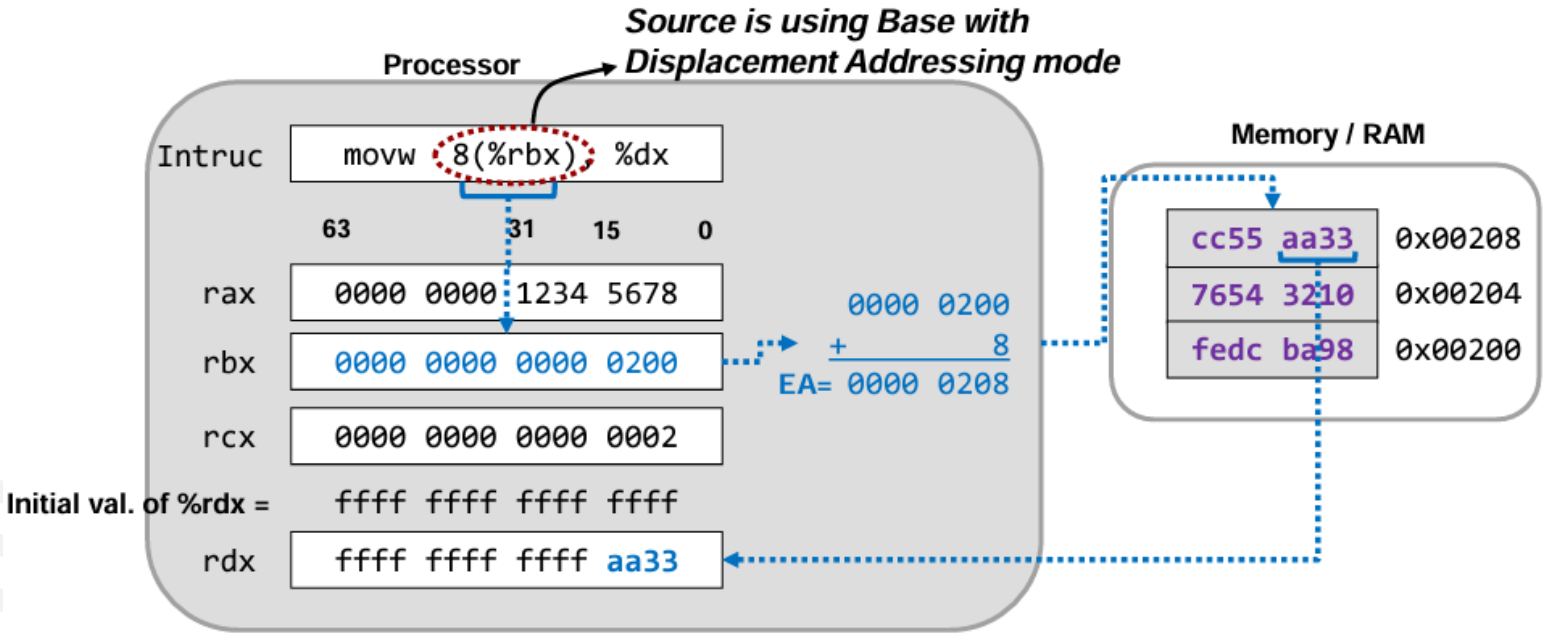
- 圆括号表示间接寻址模式



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

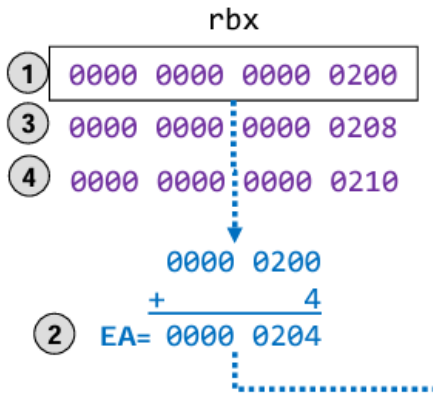
- 为什么需要Base/Indirect with Displacement Addressing实际案例

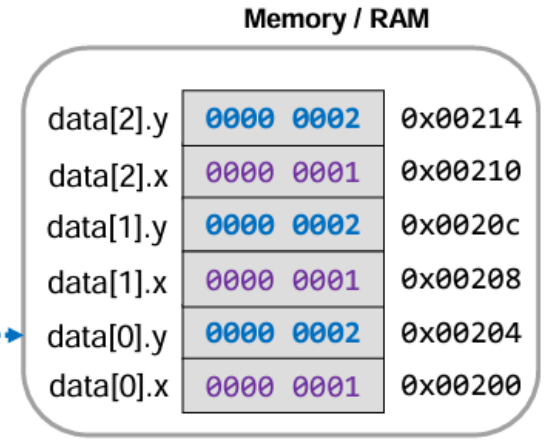
  - Useful for access members of a struct or object

```
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



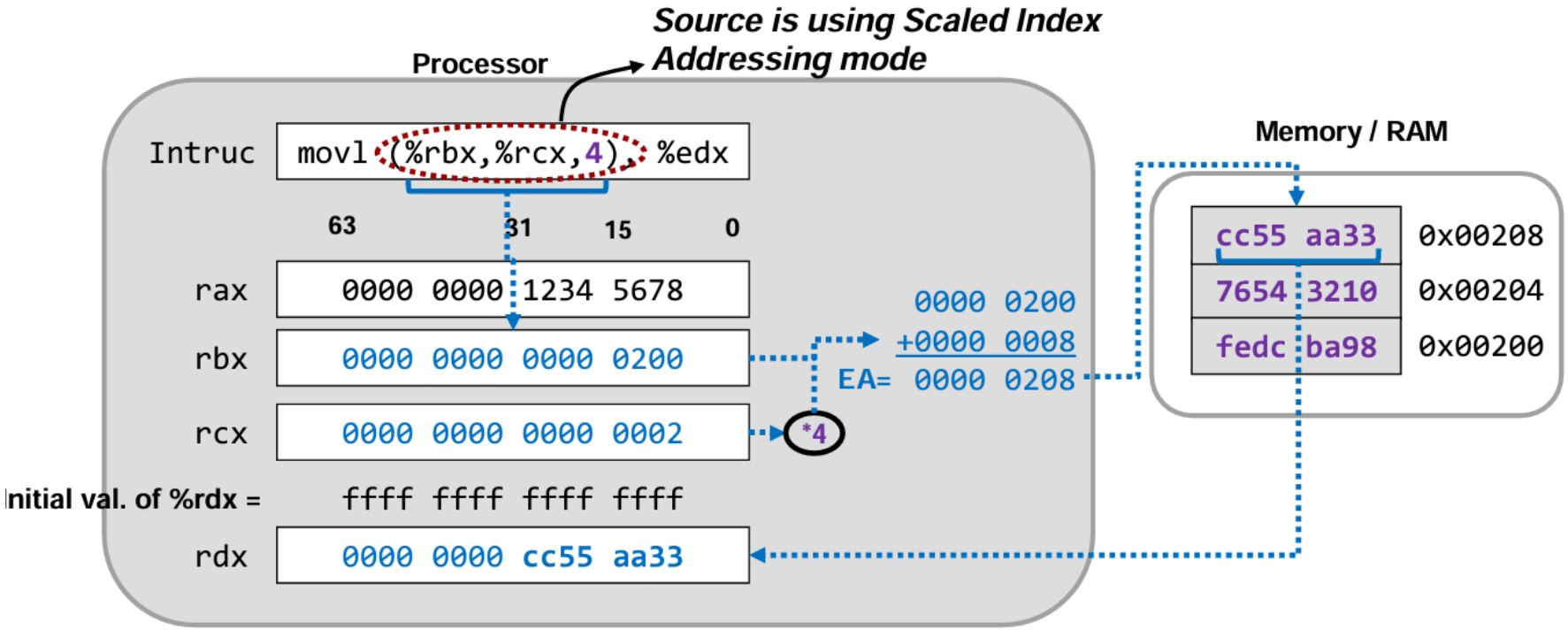



```
movq   $0x0200,%rbx
loop 3 times {
1 3 4 movl  $1, (%rbx)
2     movl  $2, 4(%rbx)
      addq  $8, %rbx
}
```

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



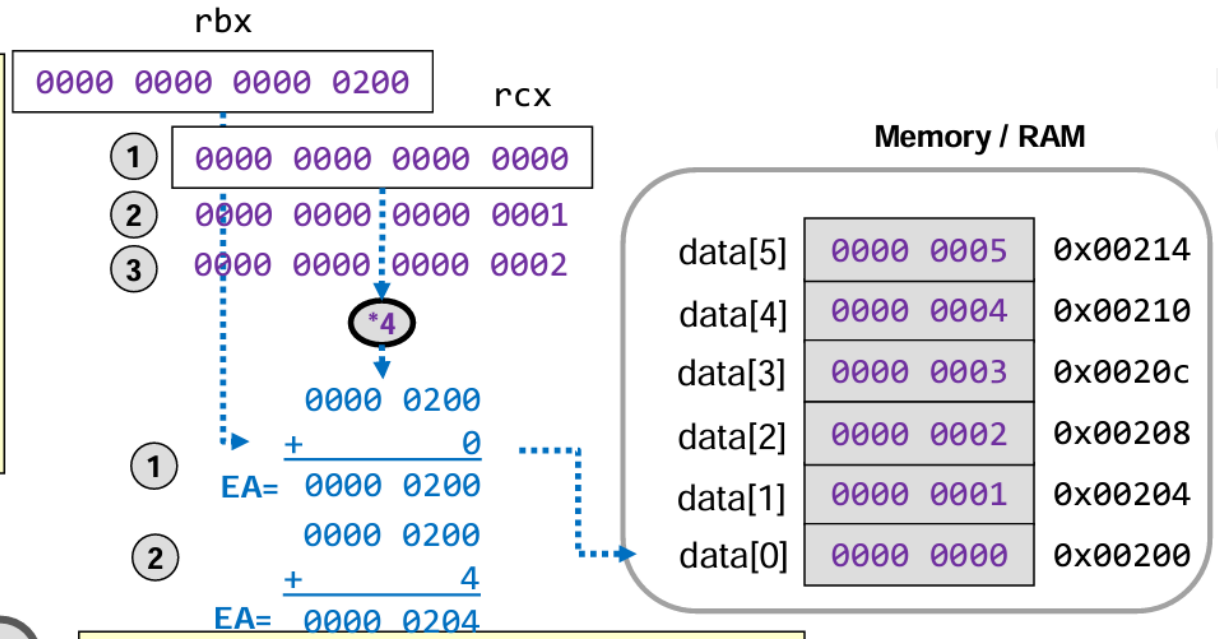
指令集的分类1 – 传输指令： Scaled Index Addressing Mode

- 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



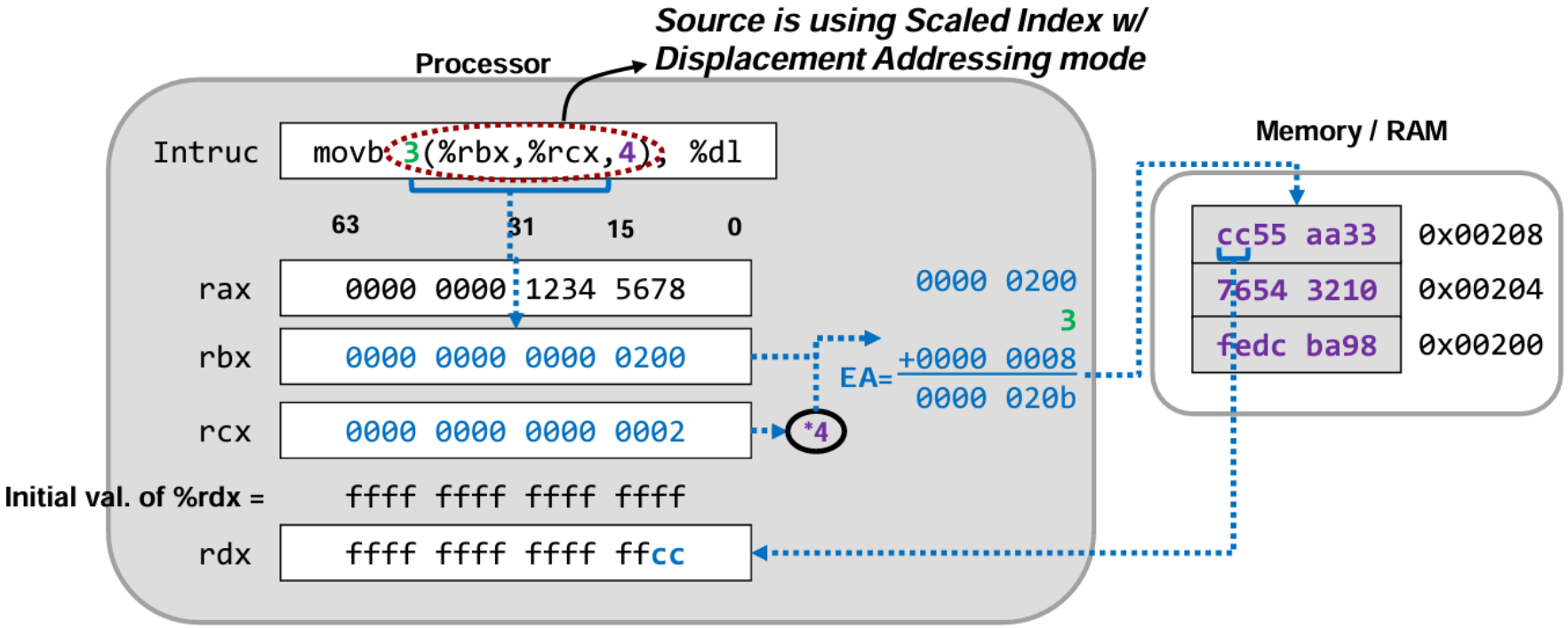
Array of:
- chars/bytes => Use s=1
- shorts/words => Use s=2
- ints/floats/dwords => Use s=4
- long longs/doubles/qwords => Use s=8

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

Assembly

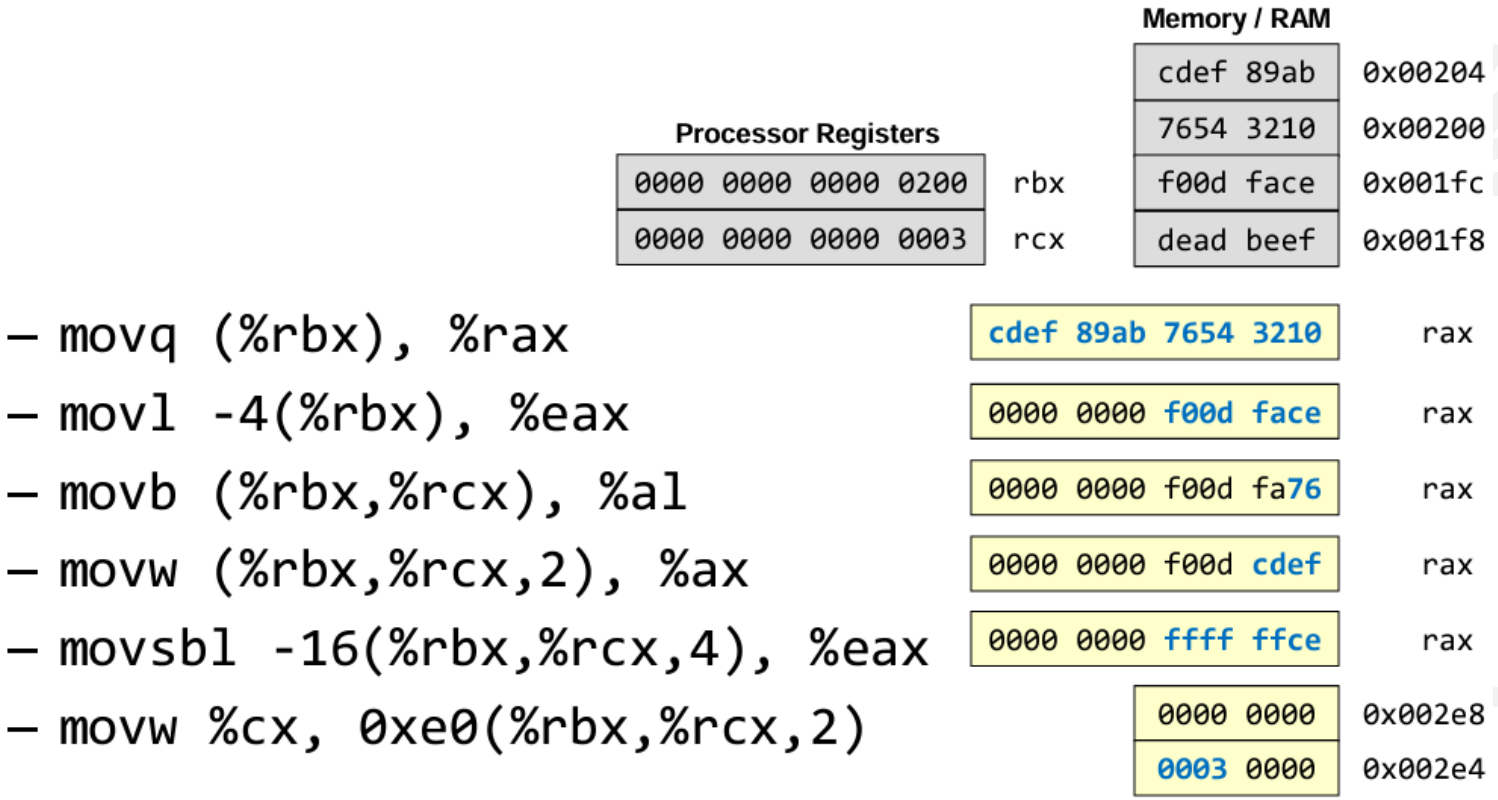
指令集的分类1 – 传输指令： Scaled Index w/ Displacement Addressing Mode

- 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

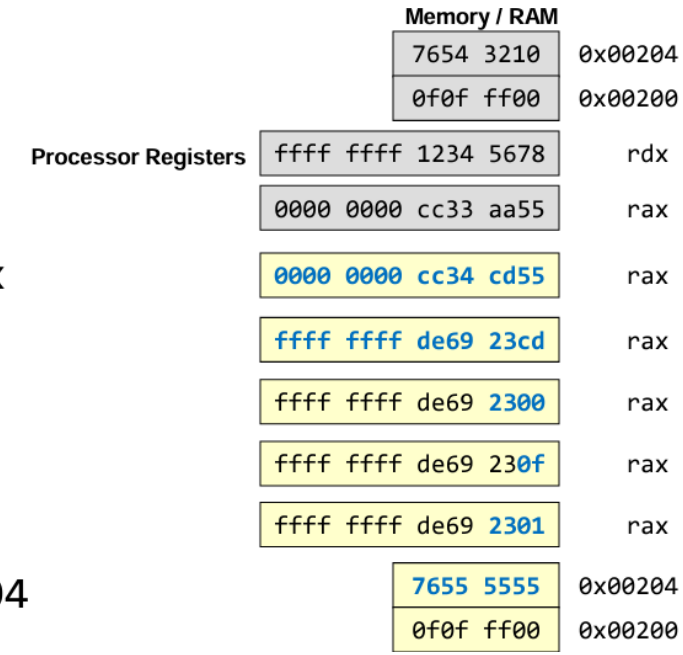
- 利用ALU来完成实际计算任务

- 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

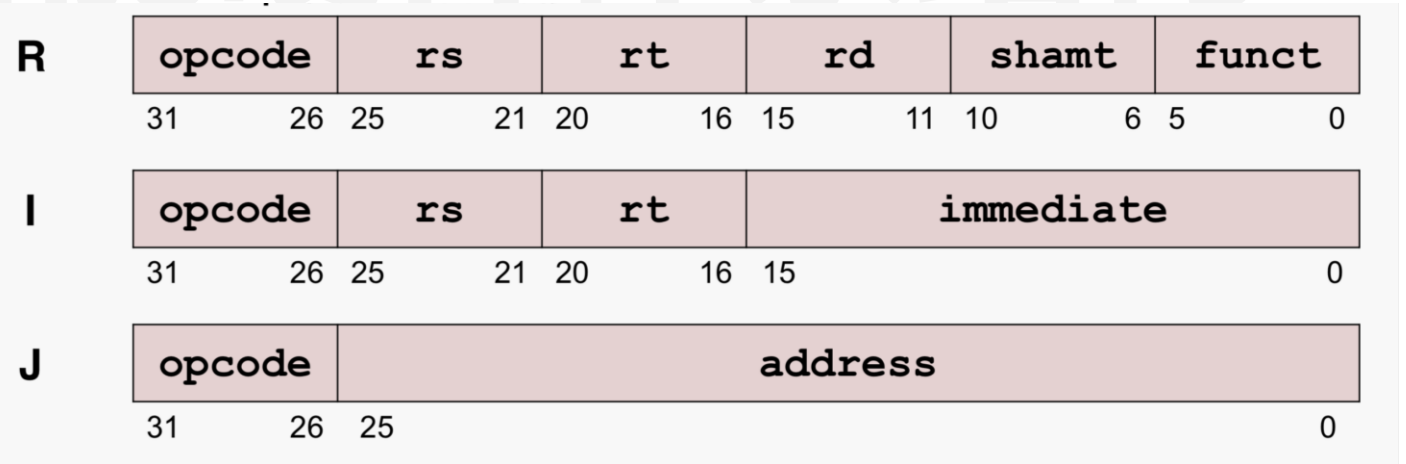
- `imm`: signed immediate in $imm_{11:0}$
- `uimm`: 5-bit unsigned immediate in $imm_{4:0}$
- `upimm`: 20 upper bits of a 32-bit immediate, in $imm_{31:12}$
- `Address`: memory address: rs1 + SignExt($imm_{11:0}$)
- `[Address]`: data at memory location Address
- `BTA`: branch target address: PC + SignExt({$imm_{12:1}$,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

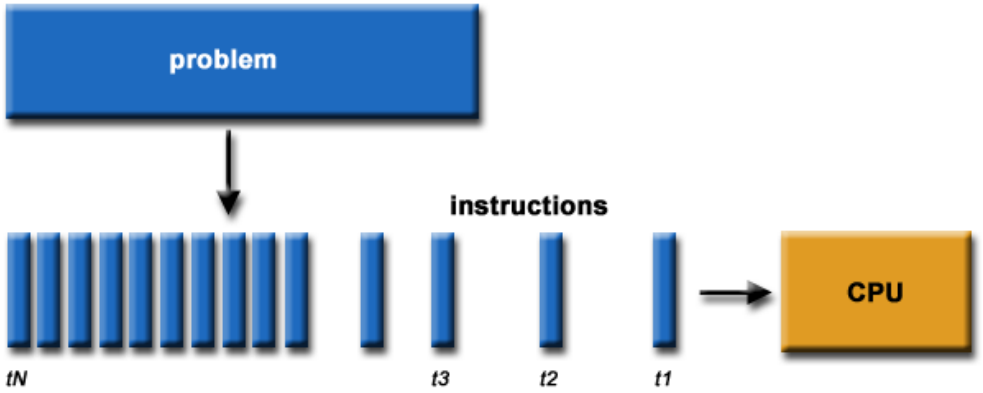
•Opcode

- 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

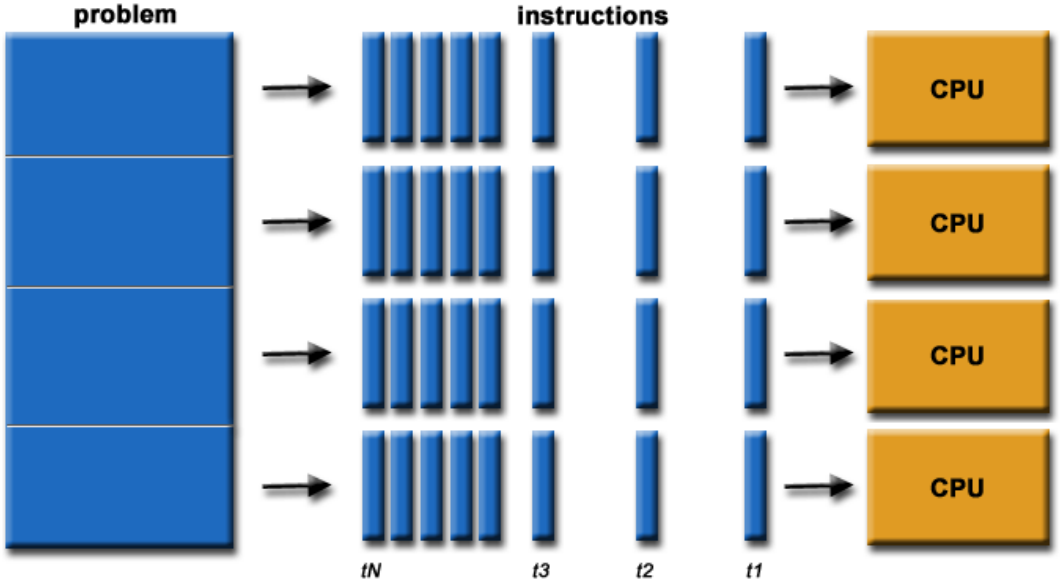


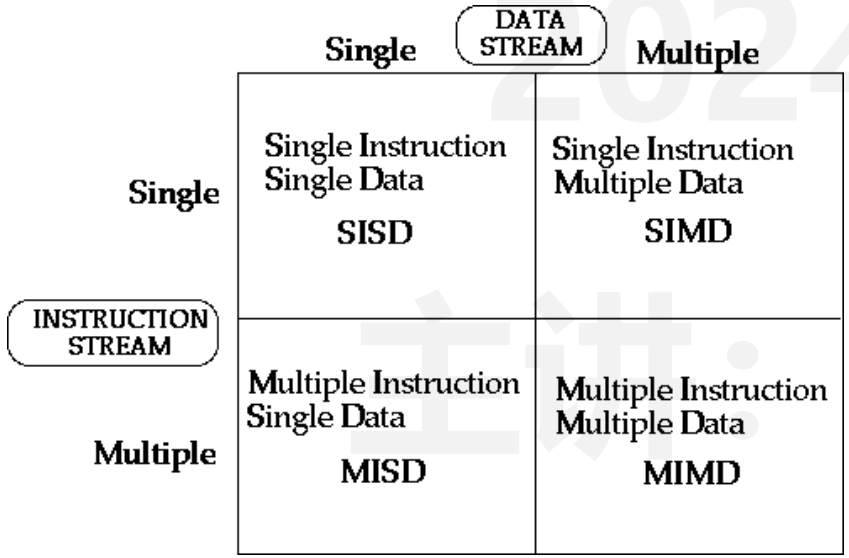
代表性指令集： GPU的CUDA指令集

- SISD、SIMD、MISD、MIMD的扩展

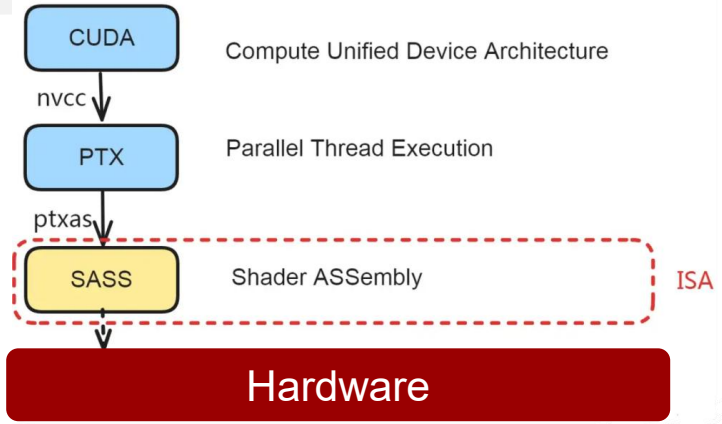


并行计算



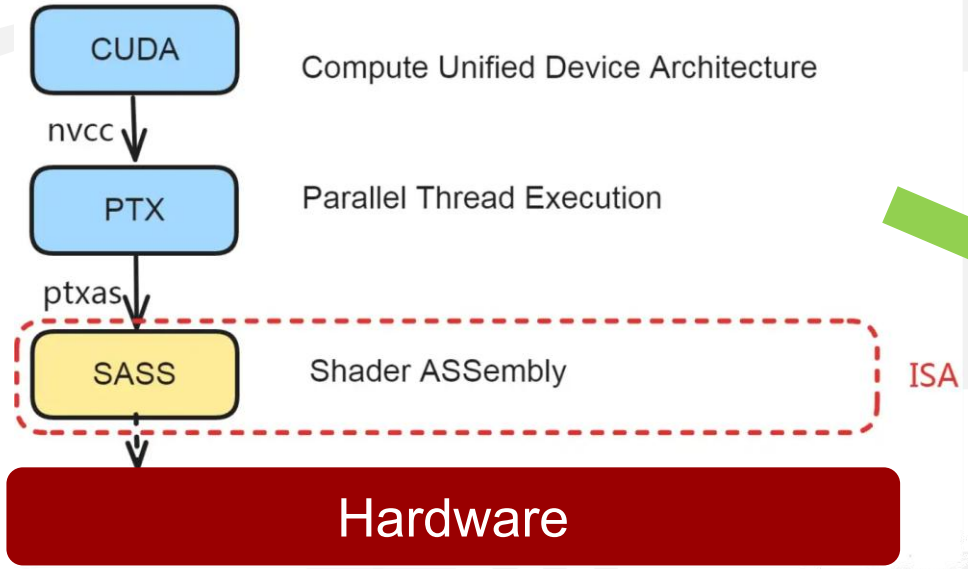


CUDA指令集的层次



代表性指令集： GPU的CUDA指令集

- PTX和SASS



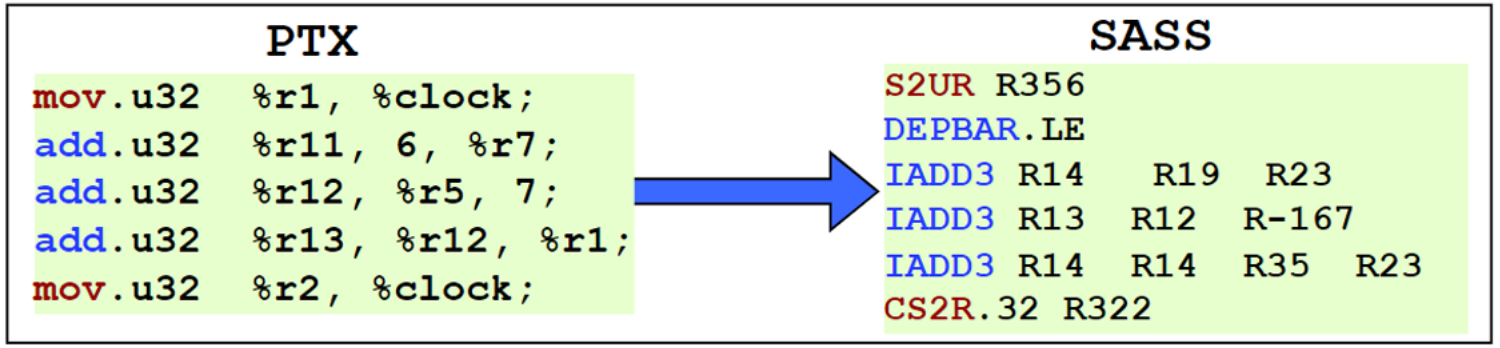
CUDA C/C++程序编译后，一般NVCC会同时生成PTX和SASS，用户也可以指定只生成其中一种。SASS是机器码的硬件指令集，编译的SM版本与当前GPU的SM版本不对应的话是不能运行的

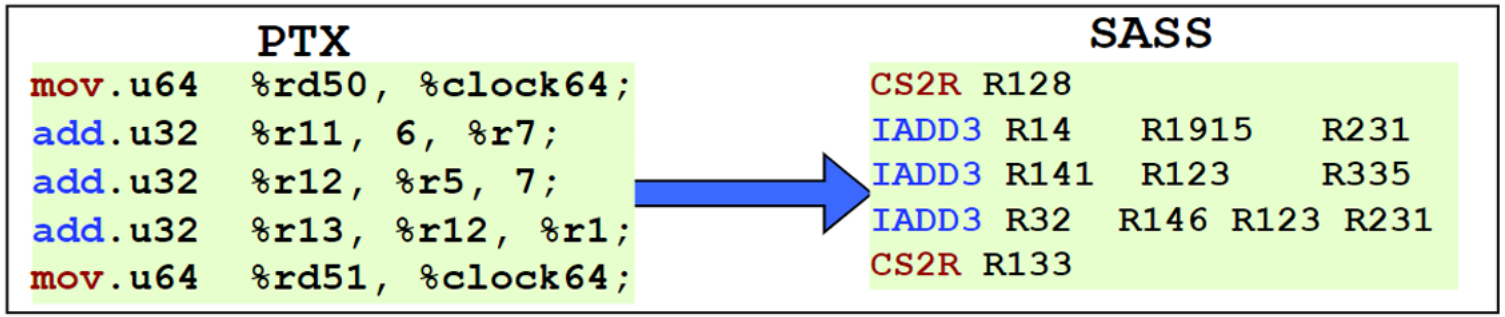
从SASS上抽象出来的一种更上层的软件编程模型，介于CUDA C/C++和SASS之间

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





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

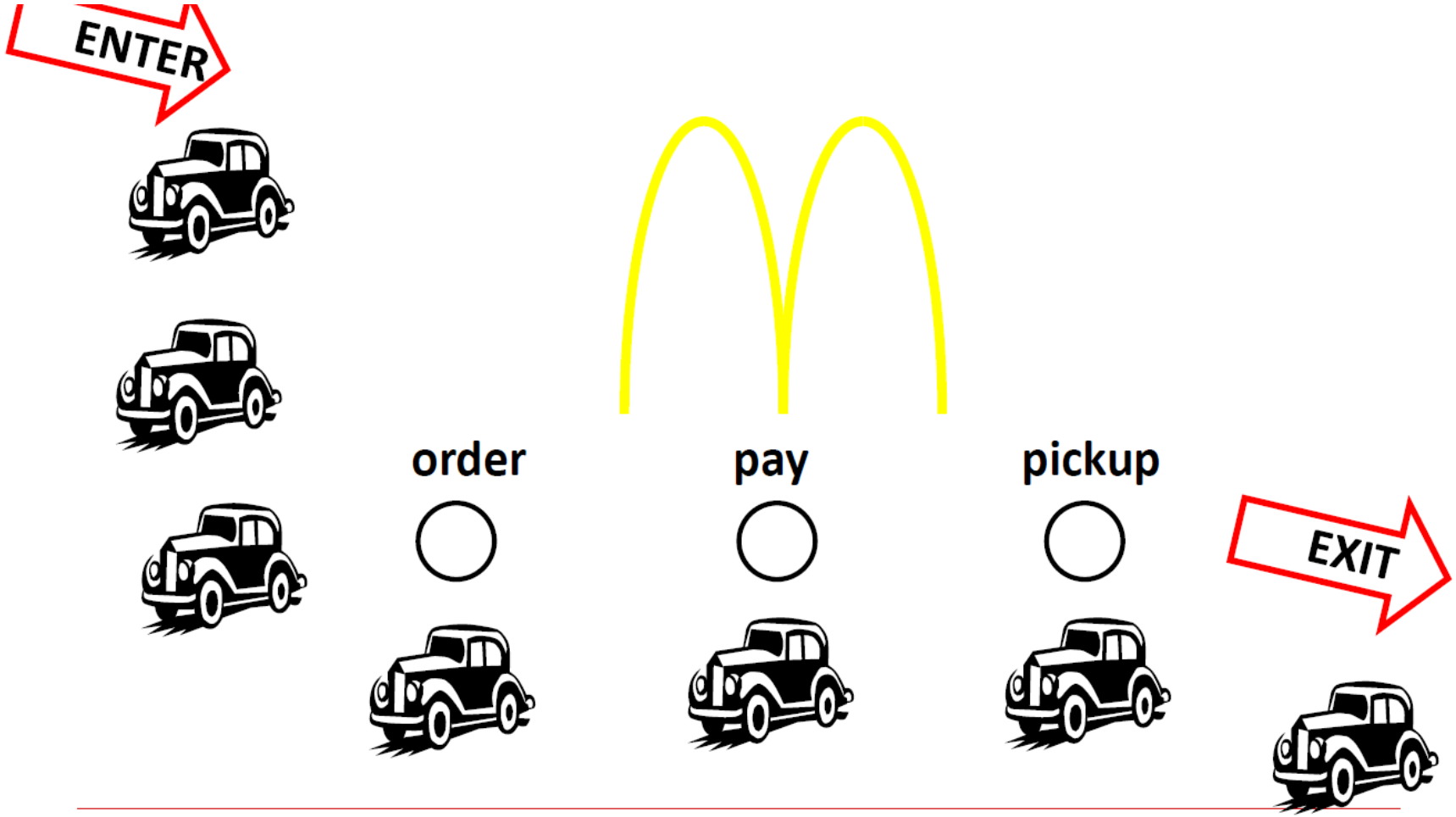
Div / Rem Instruction			Pop Instruction		
rem/div.u16/s16	multiple instructions	290	popc.b32S	POPC	6
rem/div.s32/u32	multiple instructions	66	popc.b64	2*UPOPC + UIADD3	7
rem/div.u64/s64	multiple instructions	420	**Clz Instruction**		
div.rn.f32	multiple instructions	525	clz.b32	FLO.U32 + IADD	7
div.rn.f64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

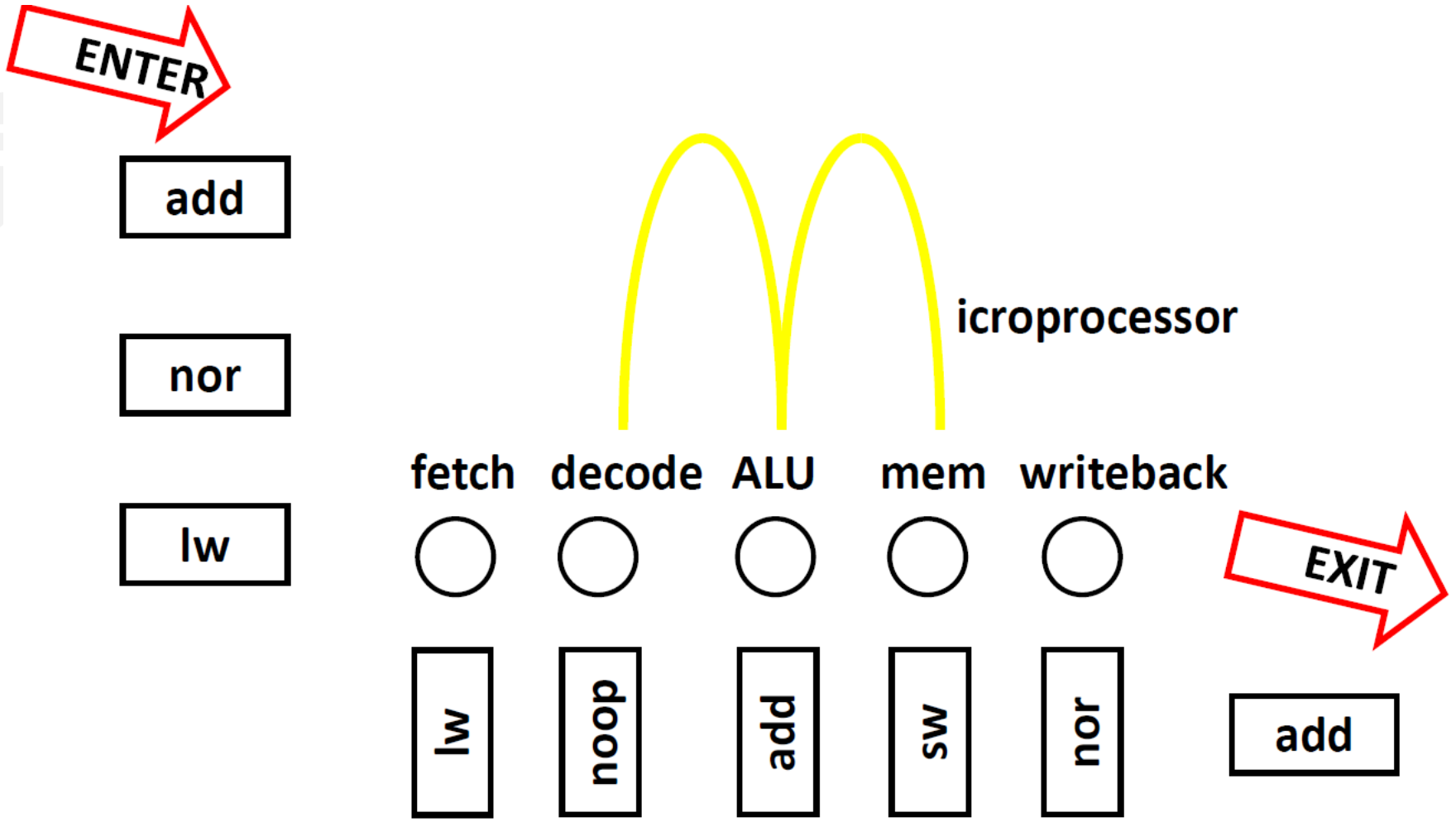
回顾：什么是流水线架构

- 流水线式运行方式 - 提高吞吐率的有效手段



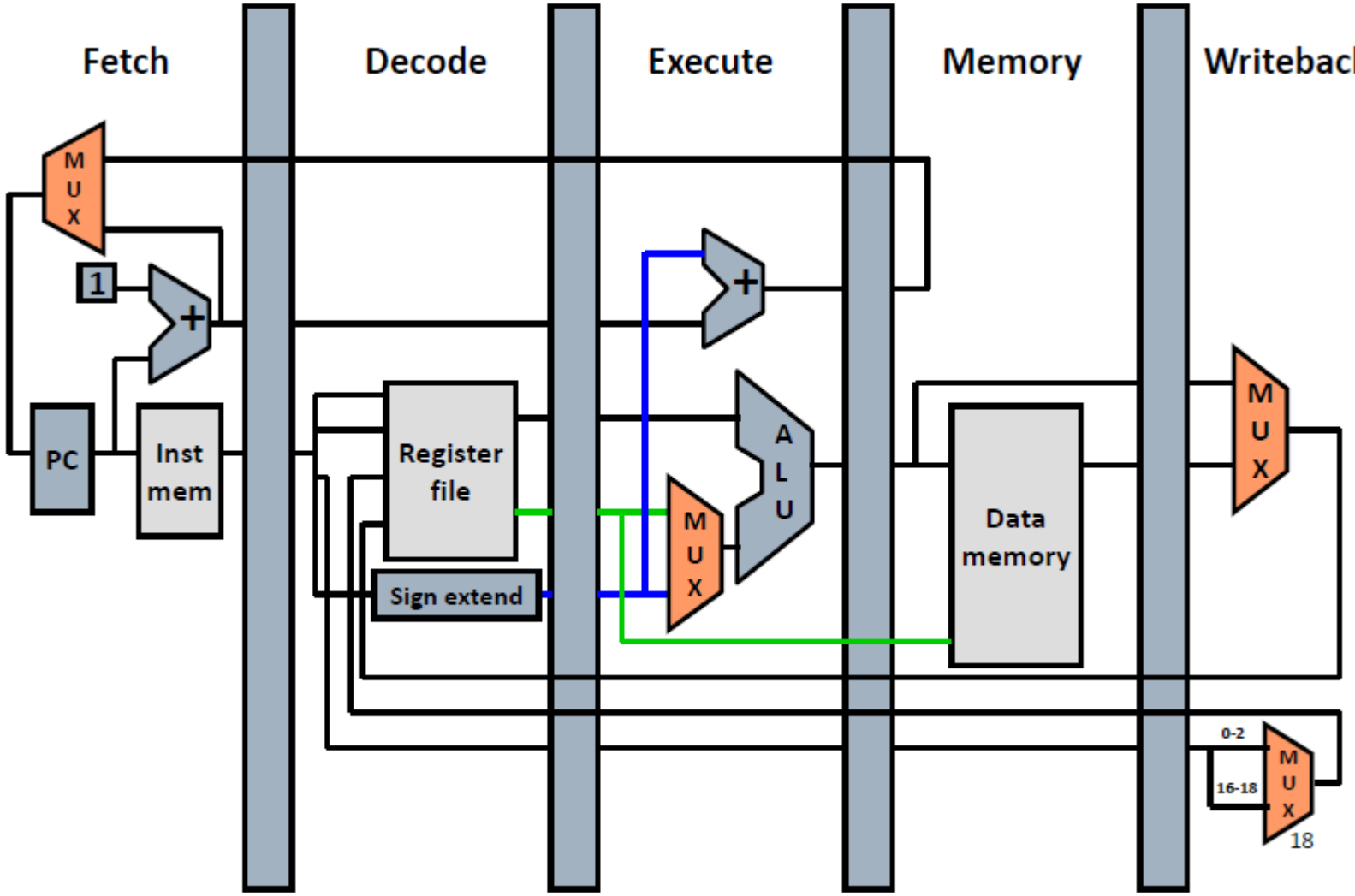
回顾：什么是流水线架构

- 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

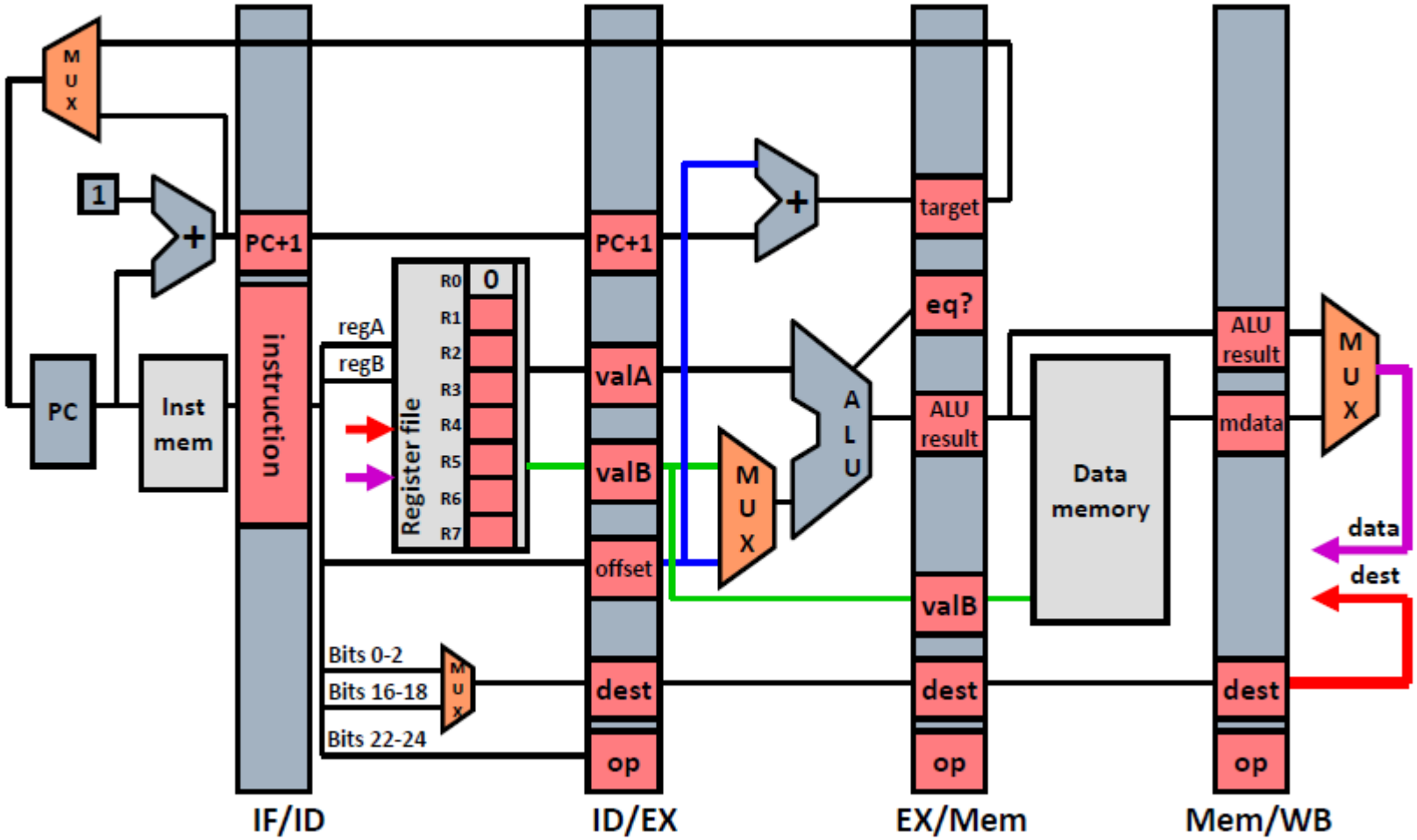
- 5级流水线设计：Fetch、Decode、Execute、Memory、Writeback



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

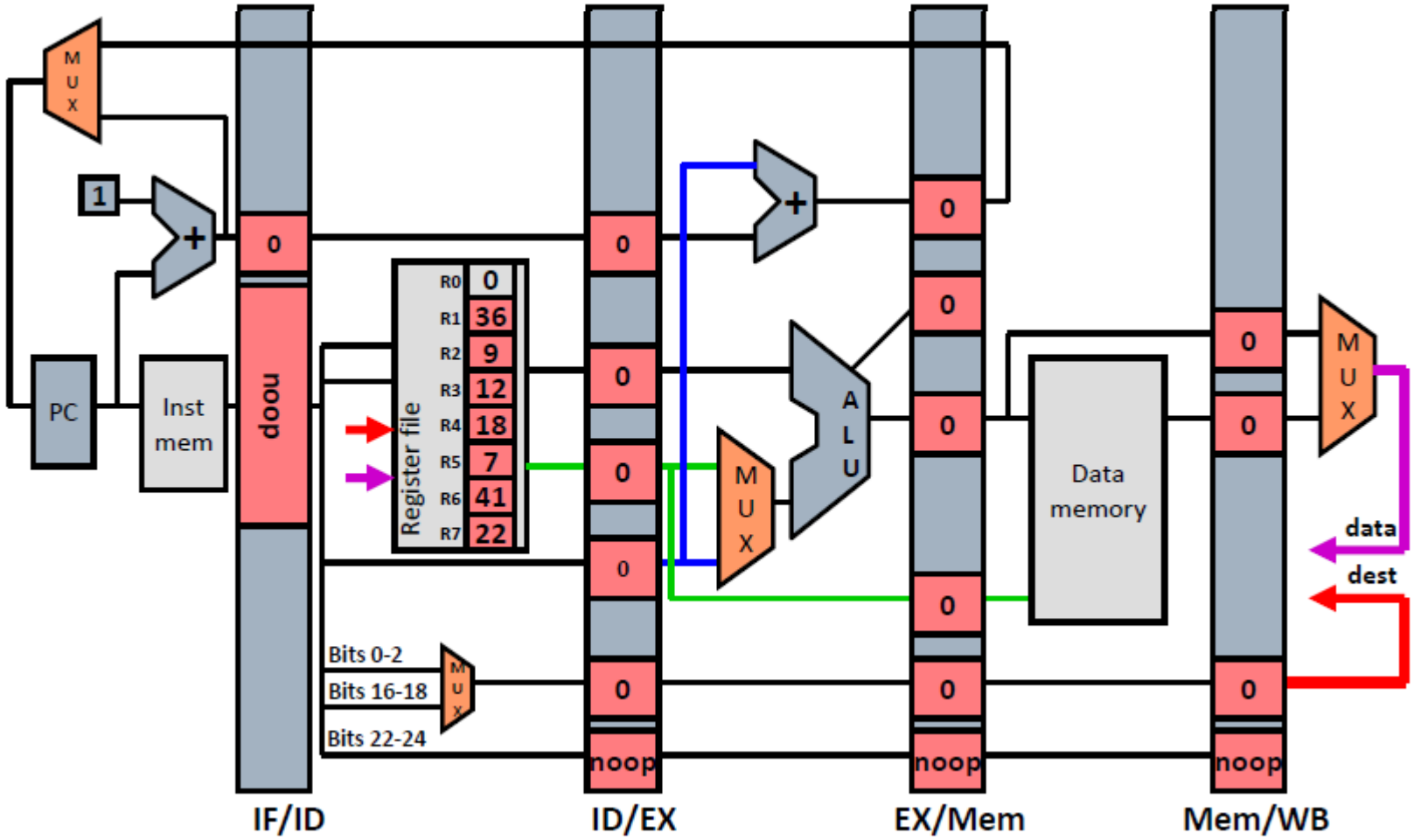
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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
```

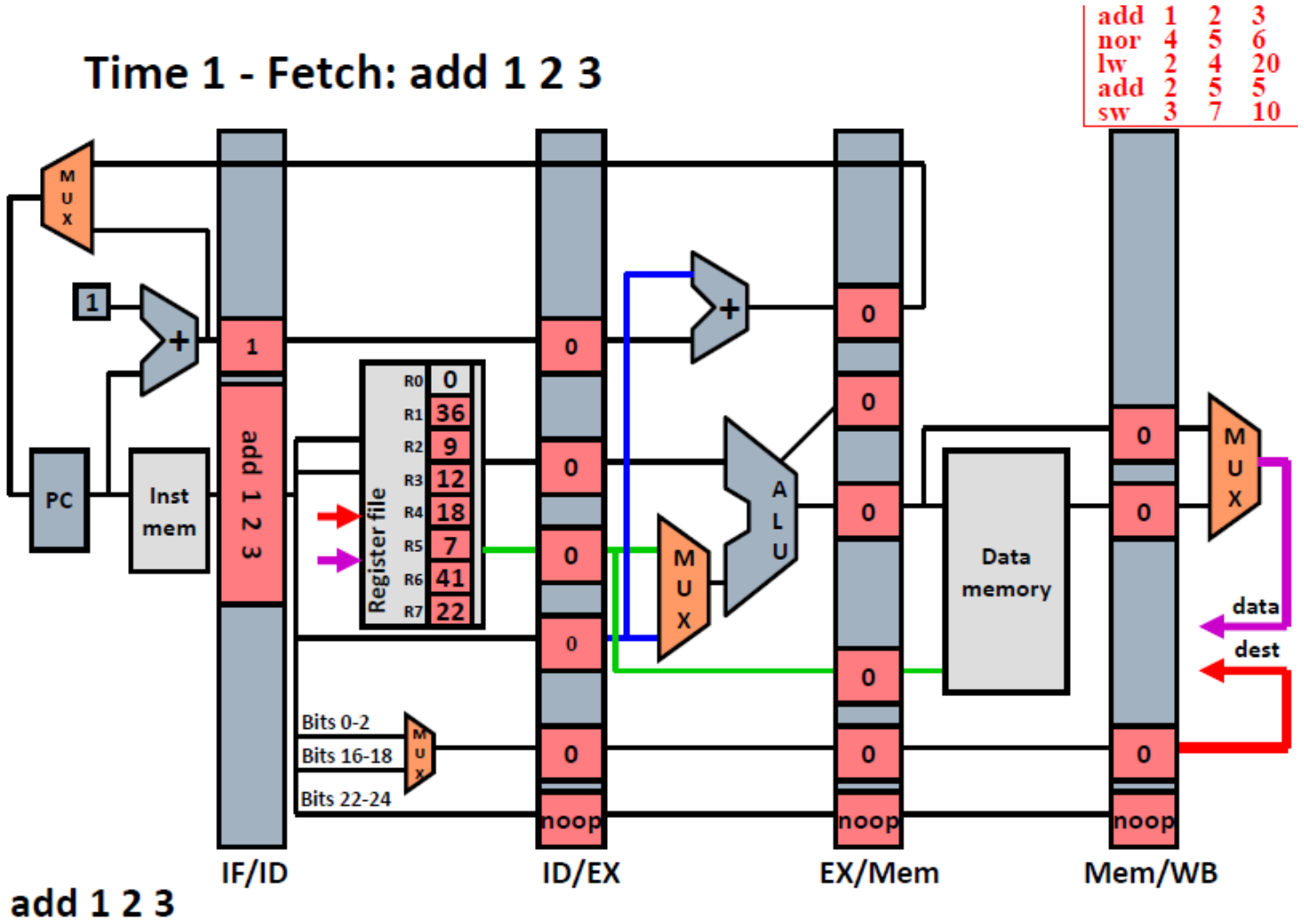


初始状态：t0

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

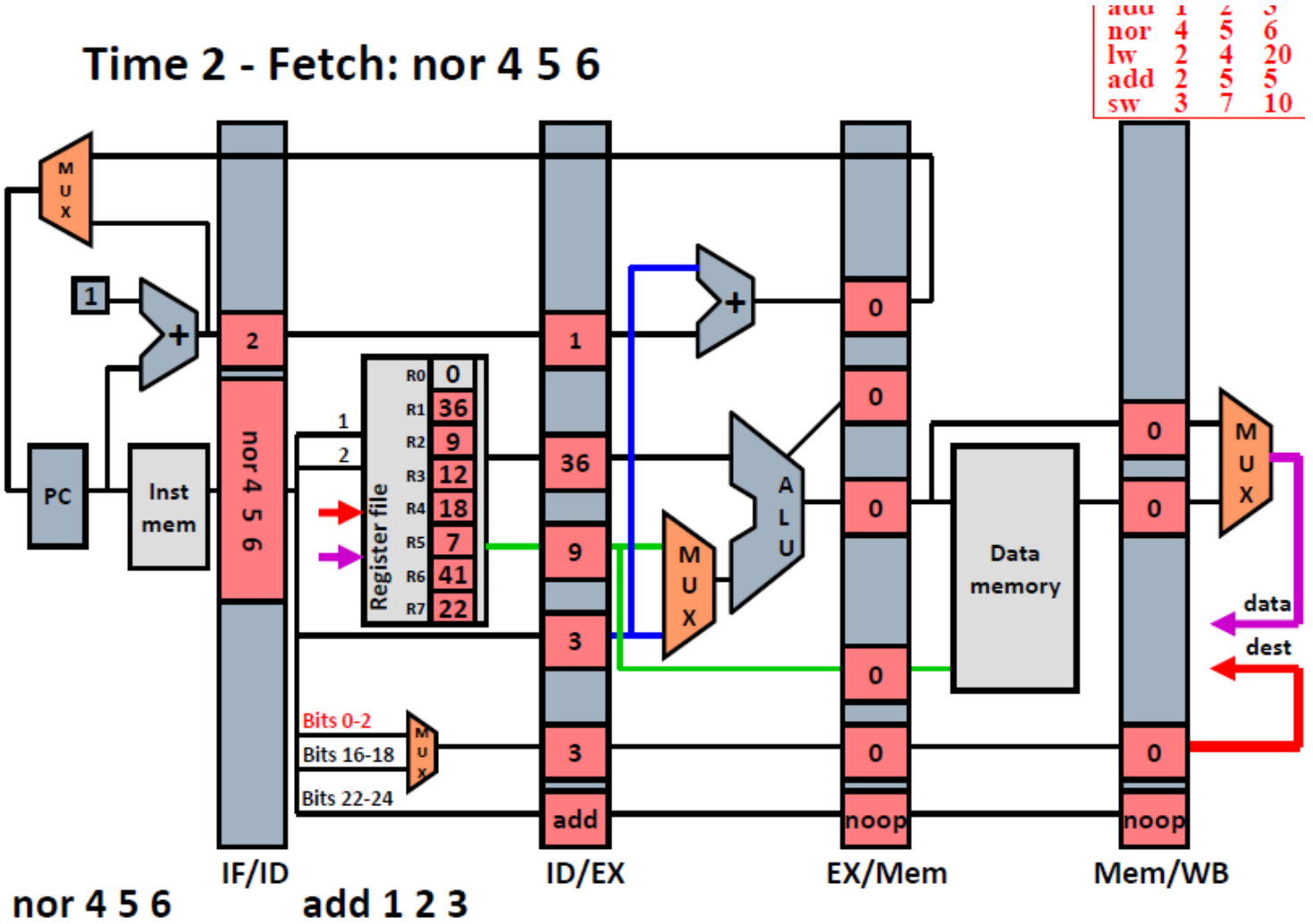
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

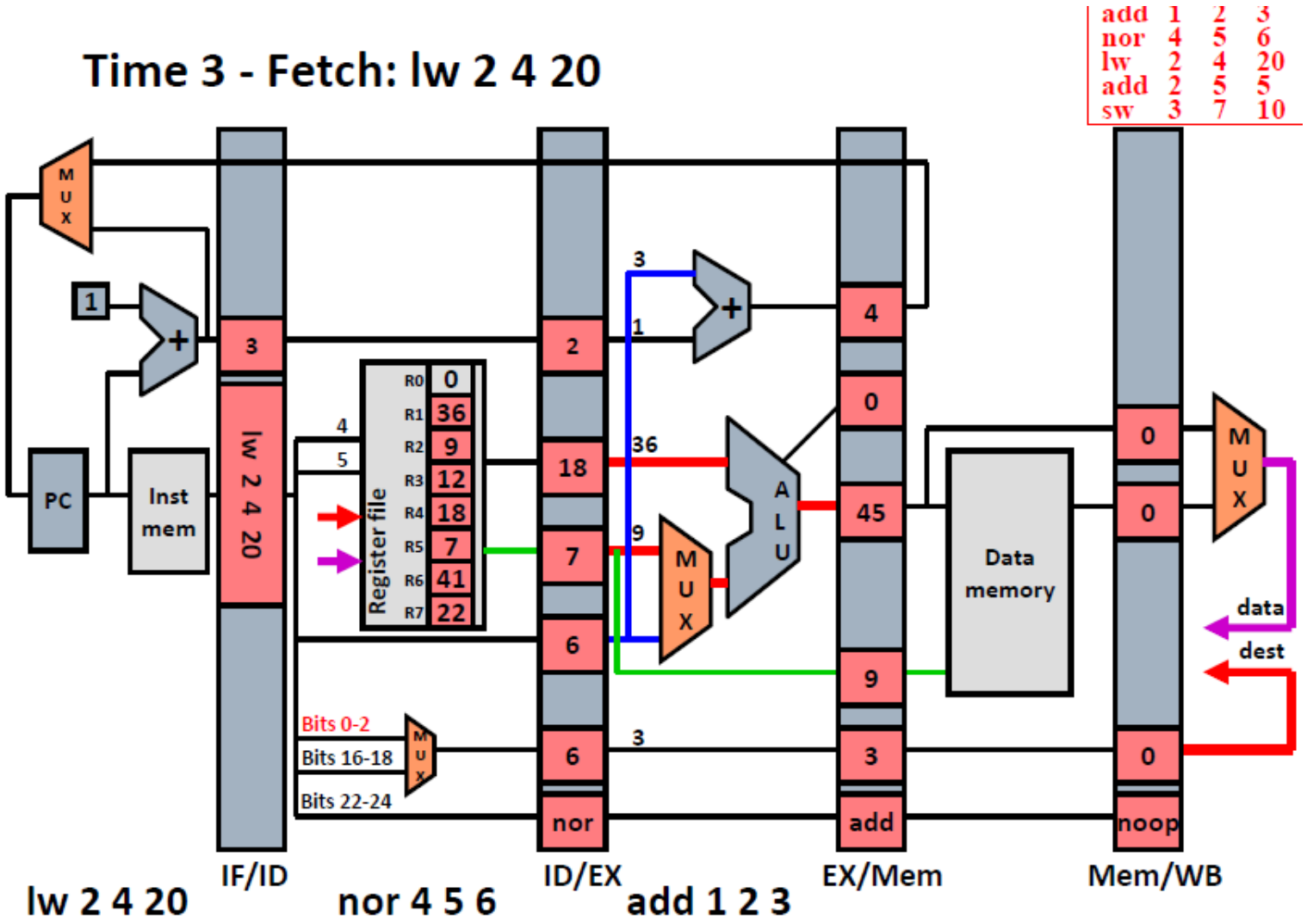
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

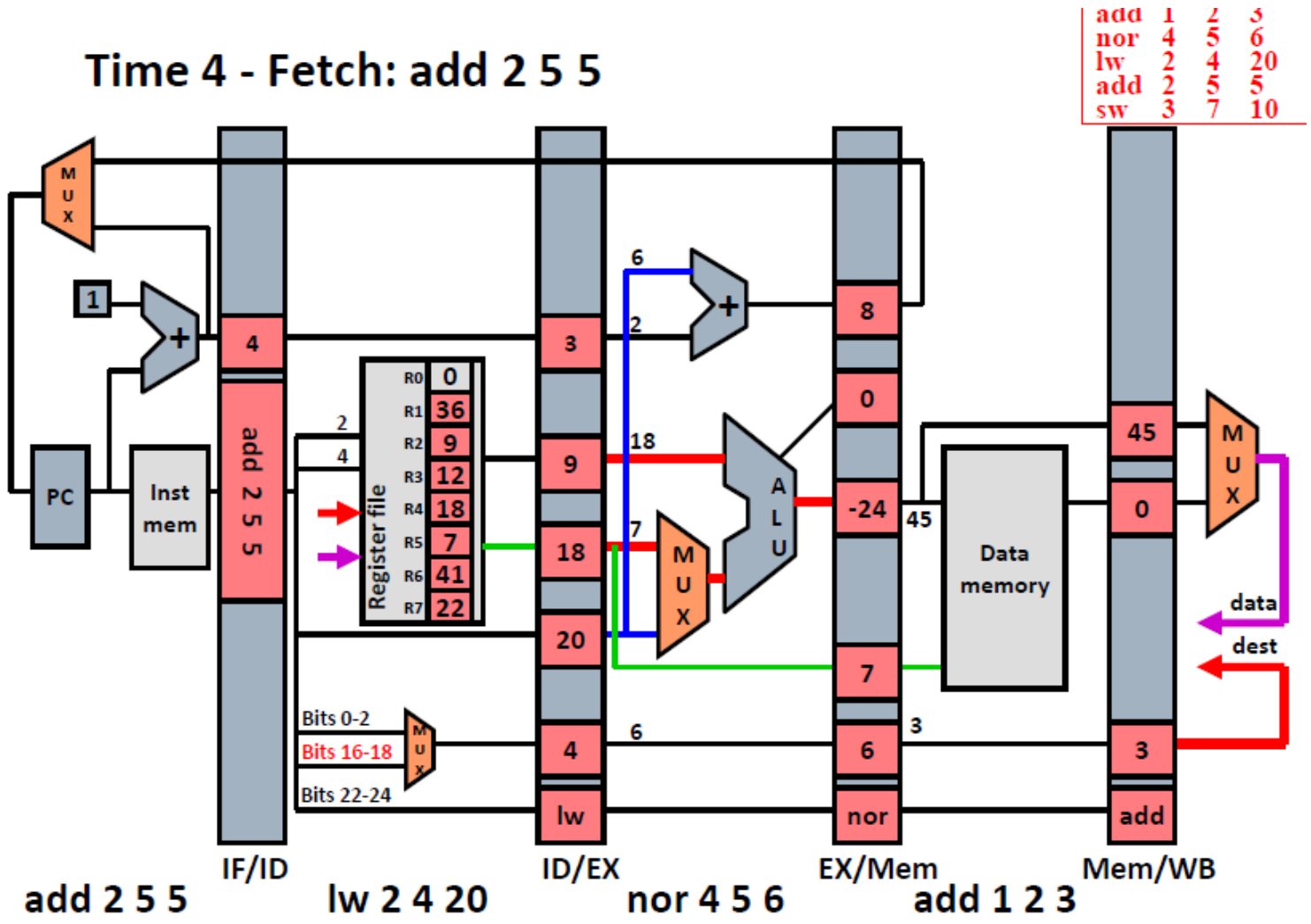
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

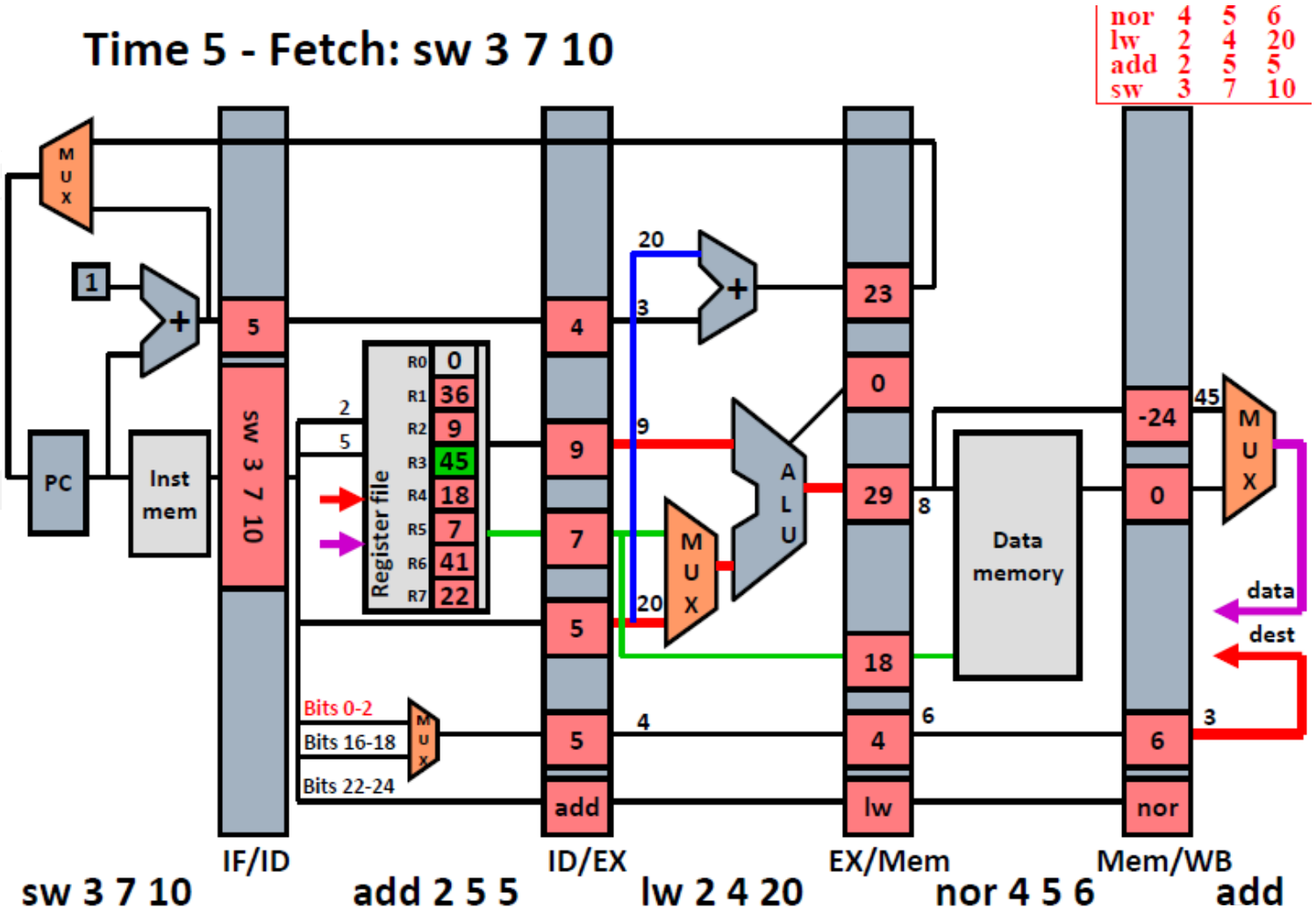
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

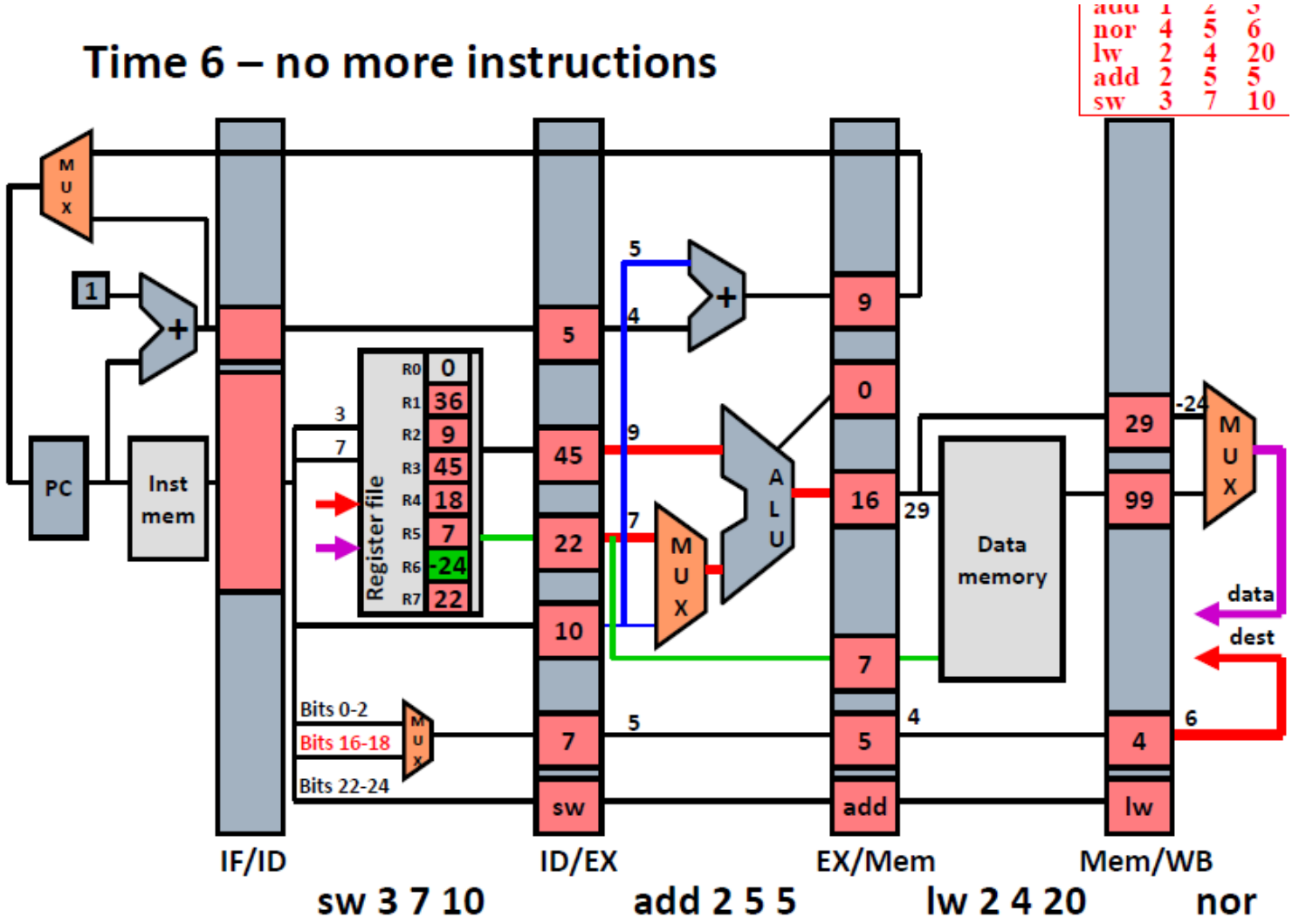
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

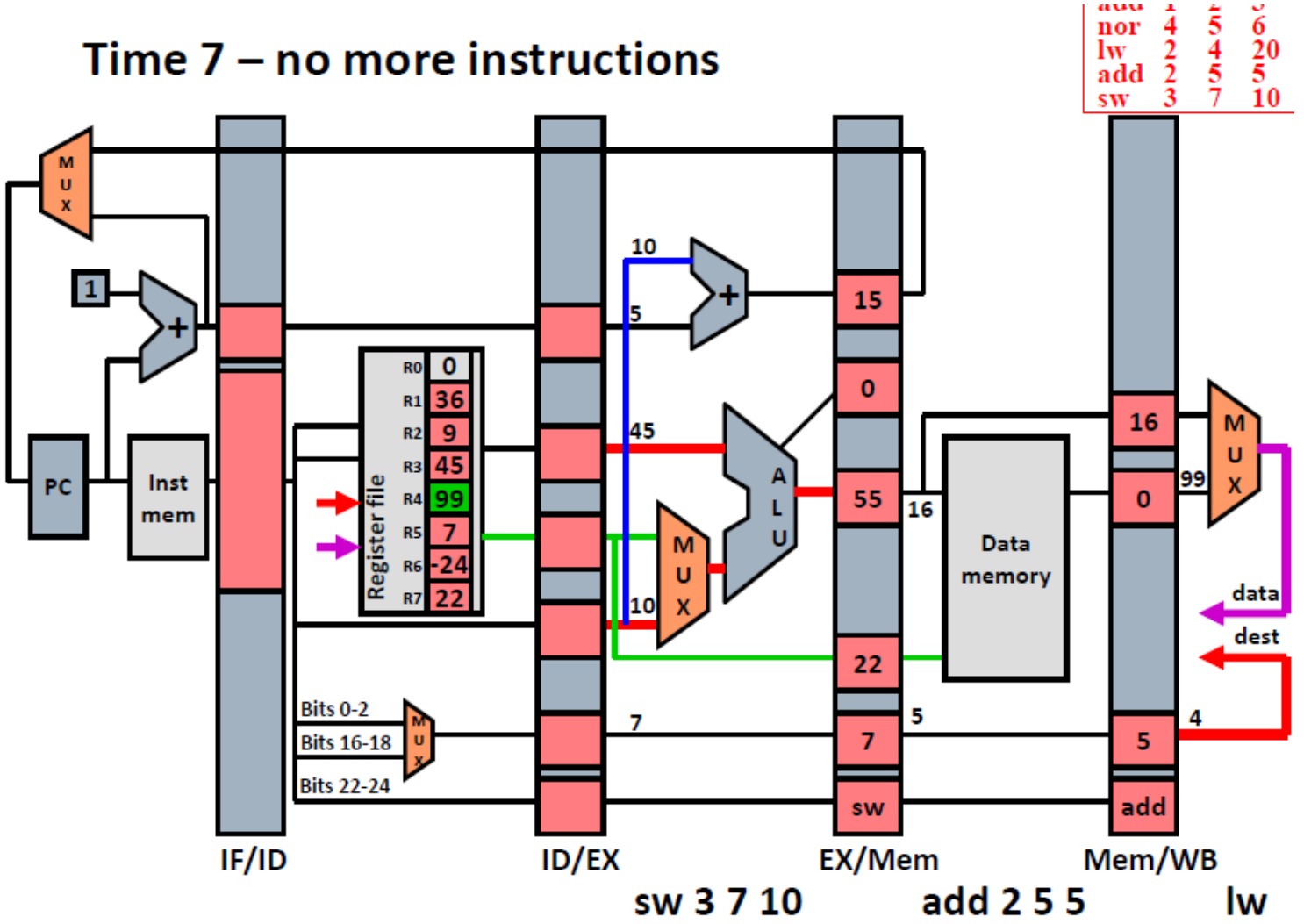
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

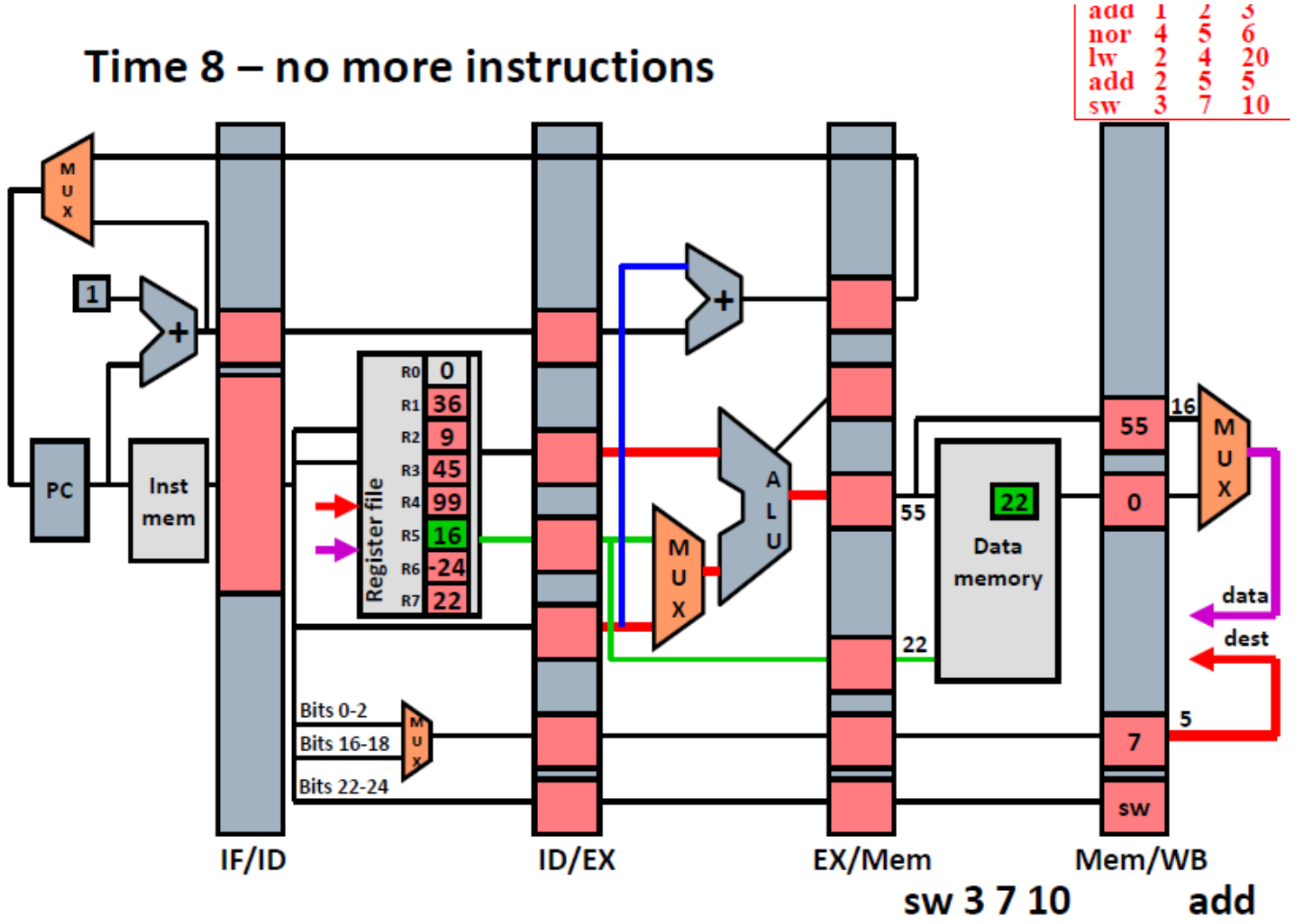
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



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

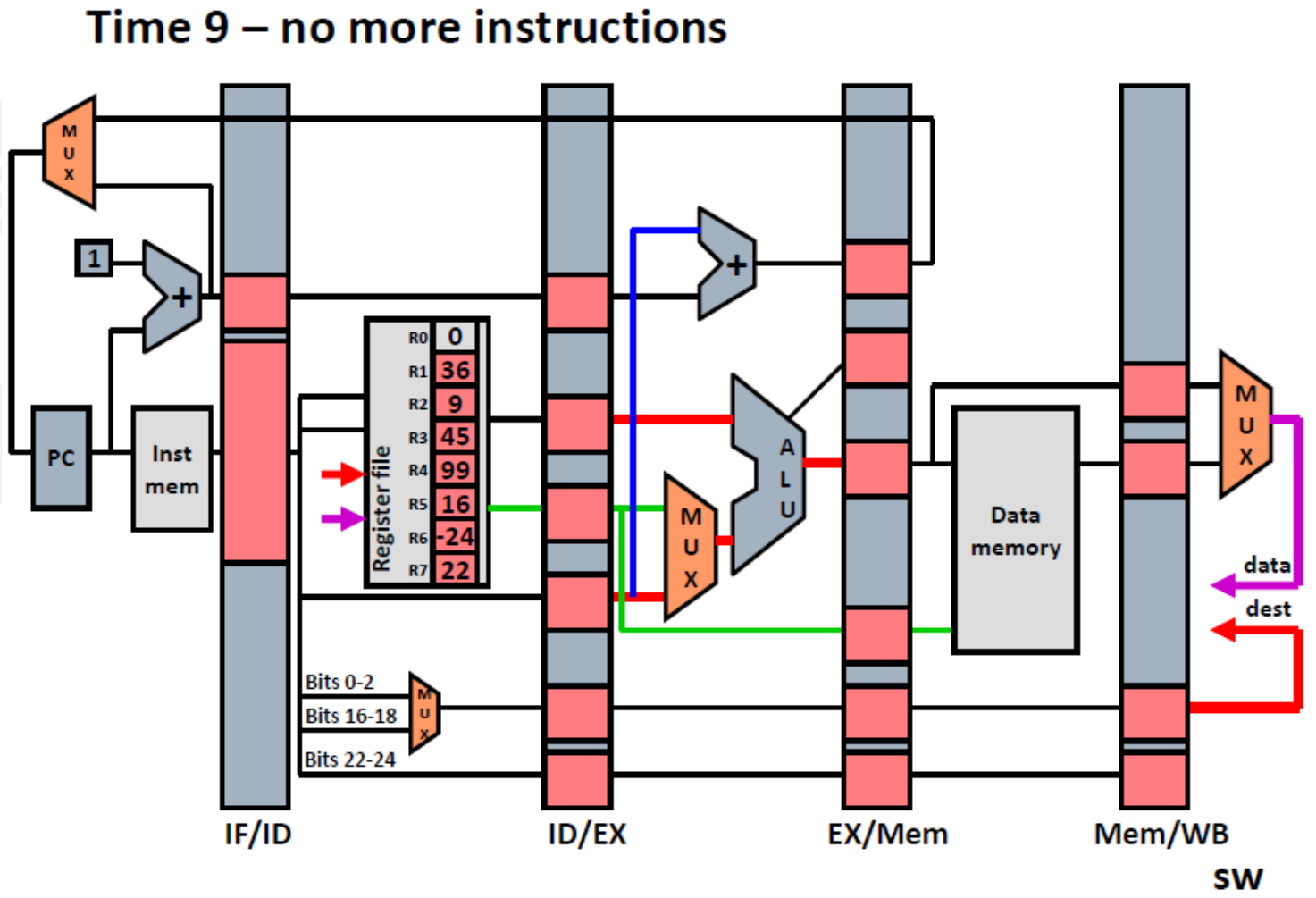
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

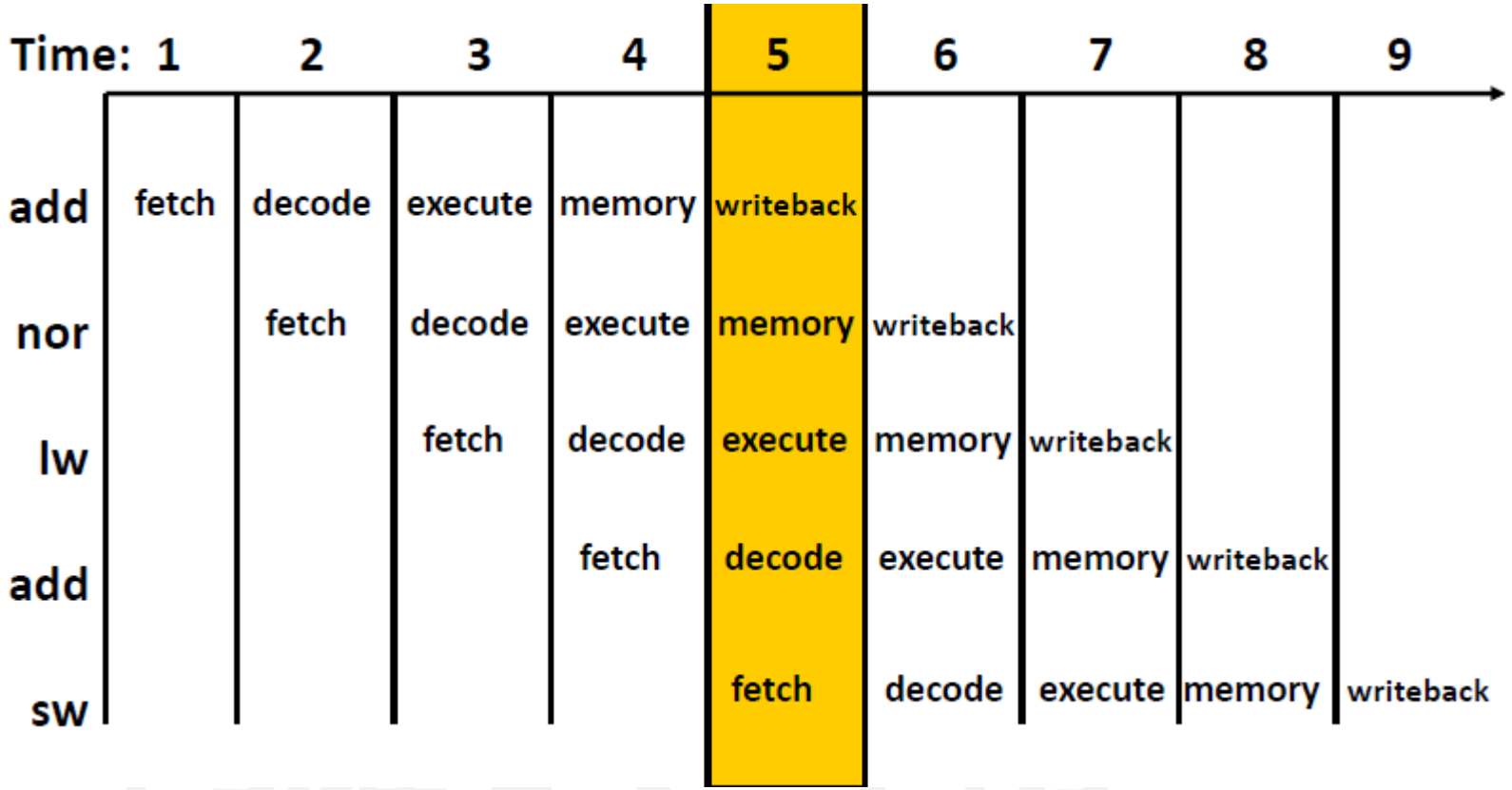
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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
```



目 录

CONTENTS

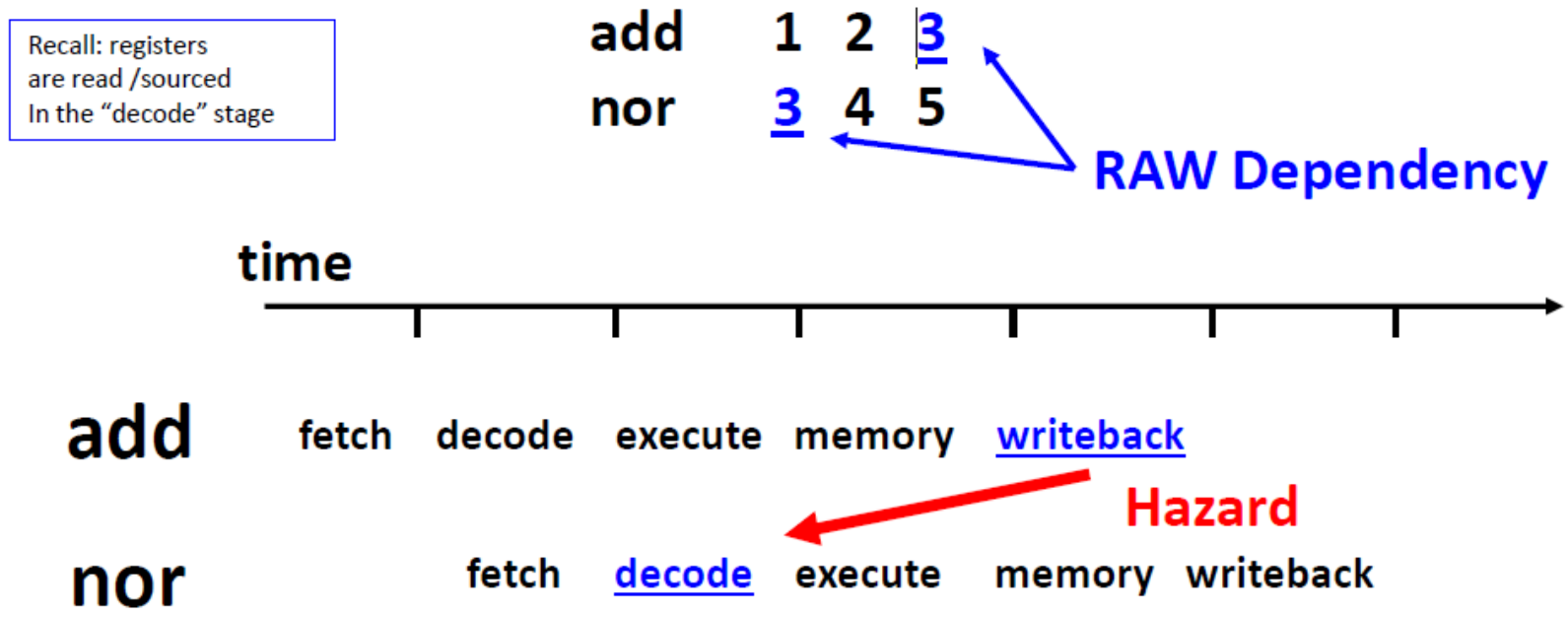
简单5级流水线可能存在什么问题?

北京大学 PEKING UNIVERSITY

- **Data hazards :** since register reads occur in stage 2 and register writes occur in stage 5 it is possible to read the wrong value if it is about to be written.
- **Control hazards :** A branch instruction may change the PC, but not until stage 4. What do we fetch before that?
- **Exceptions:** Sometimes we need to pause execution, switch to another task (maybe the OS), and then resume execution… how to we make sure we resume at the right spot

问题1：Data Hazards

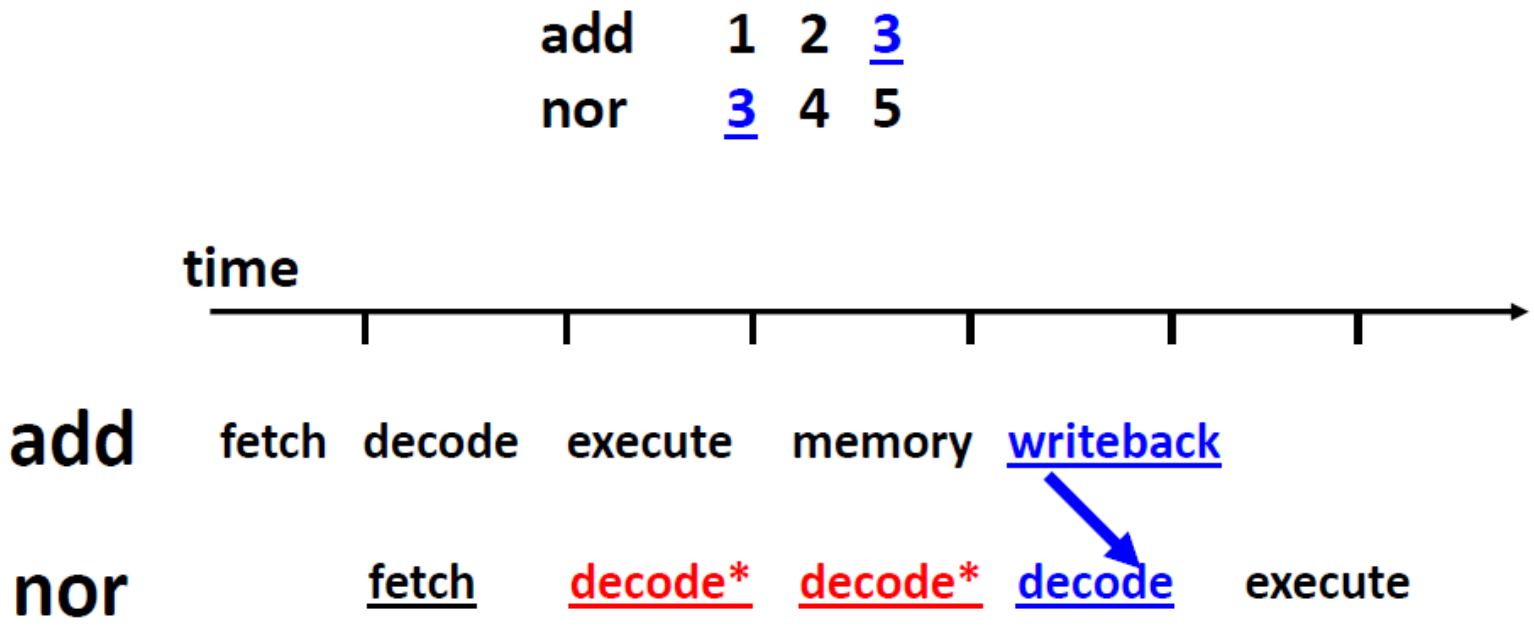
- RAW问题：Read After Write数据冲突



If not careful, nor will read a stale value of register 3

问题1：Data Hazards

- RAW问题：Read After Write数据冲突

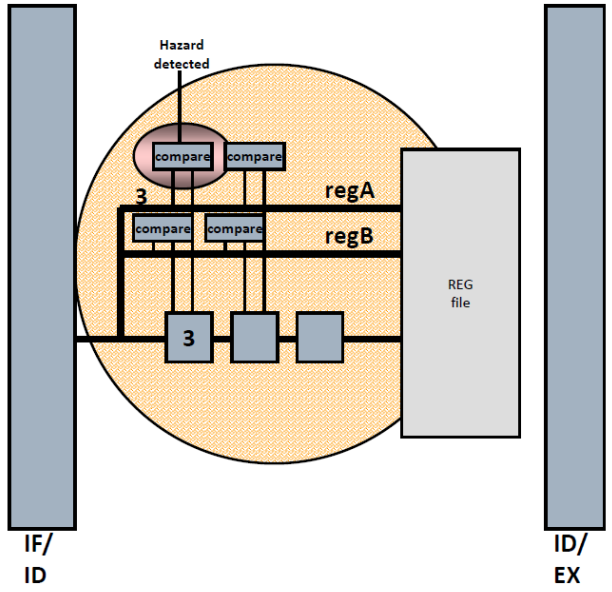


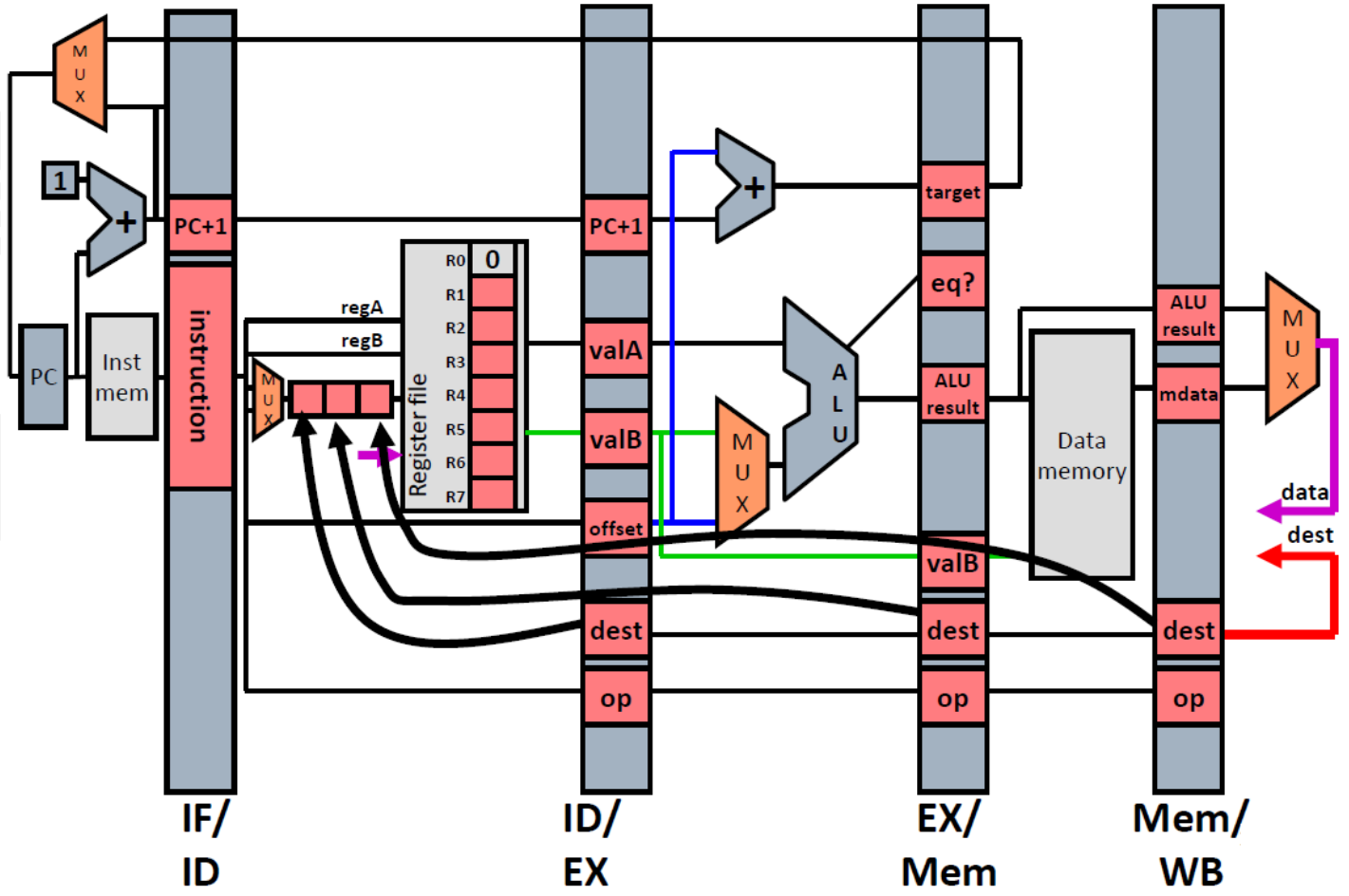
简单解决办法：流水线停顿（Pipeline Stall）

问题1：Data Hazards

- RAW问题：Read After Write数据冲突

```
1.  add 1 2 3
2.  nor 3 4 5
3.  add 6 3 7
4.  lw  3 6 10
5.  sw  6 2 12
```

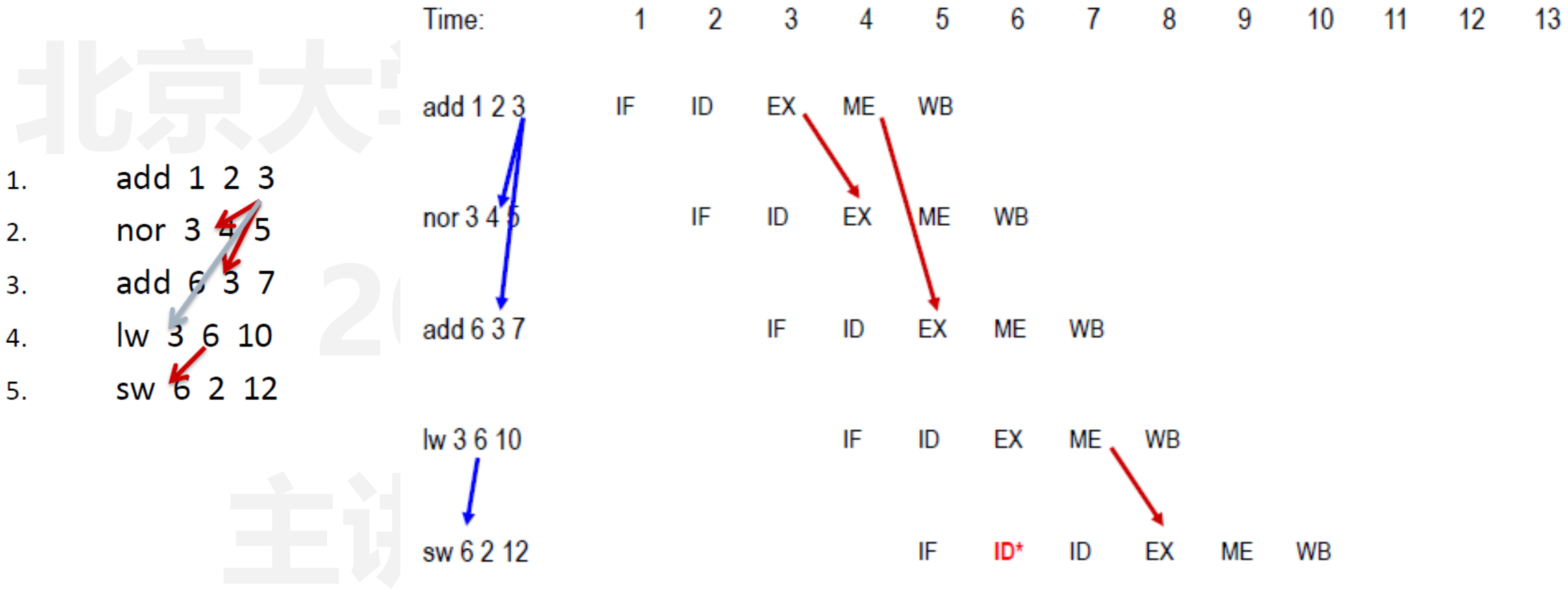




进阶解决办法：Detect and Forward

问题1：Data Hazards

- RAW问题：Read After Write数据冲突



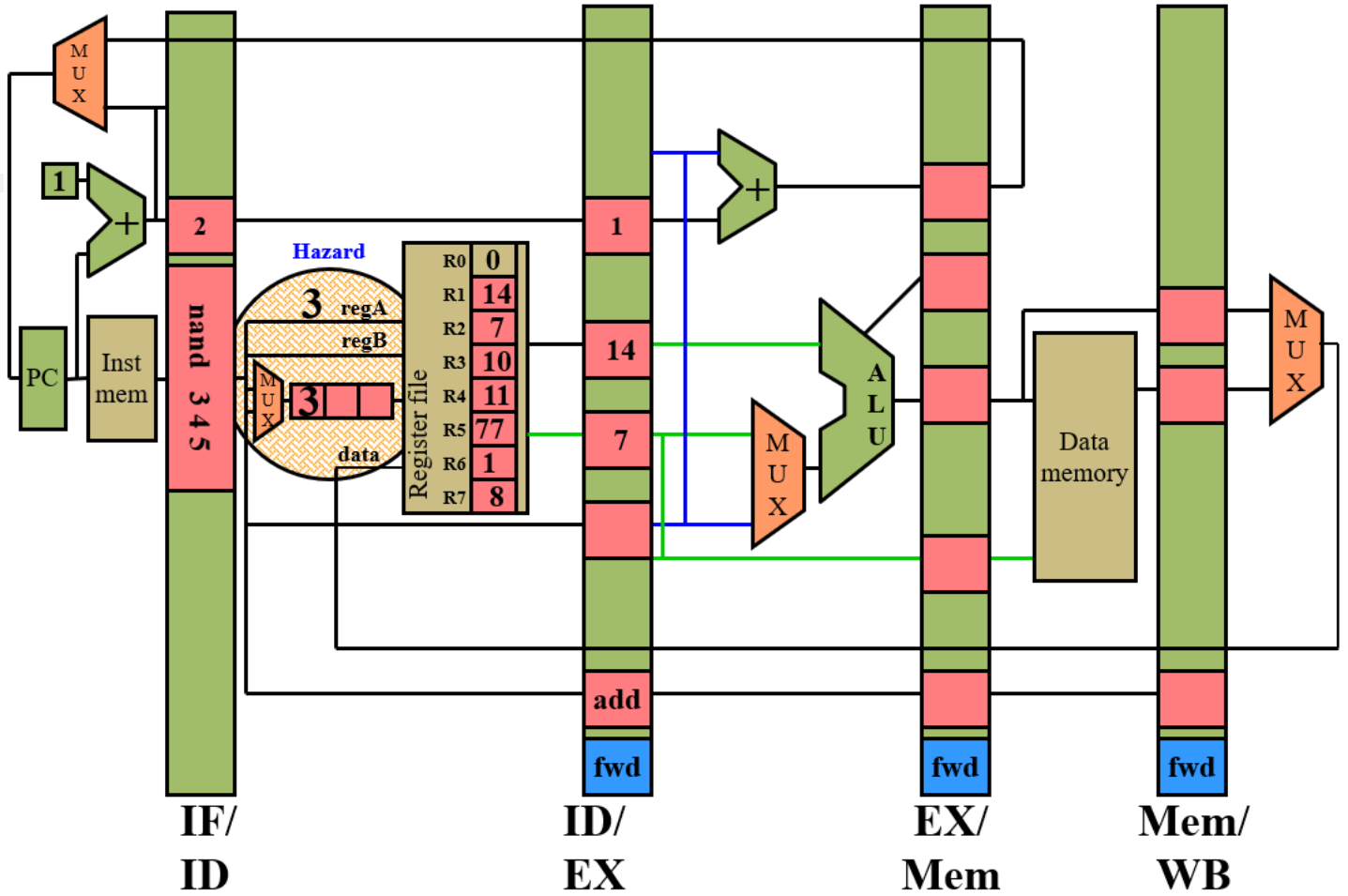
进阶解决办法：Detect and Forward

问题1：Data Hazards

- **Detect and Forward案例**

```
add 1 2 3     // r3 = r1 + r2
nand 3 4 5    // r5 = r3 NAND r4
add 6 3 7     // r7 = r3 + r6
lw 3 6 10     // r6 = MEM[r3+10]
sw 6 2 12     // MEM[r6+12]=r2
```

Cycle 3前半段

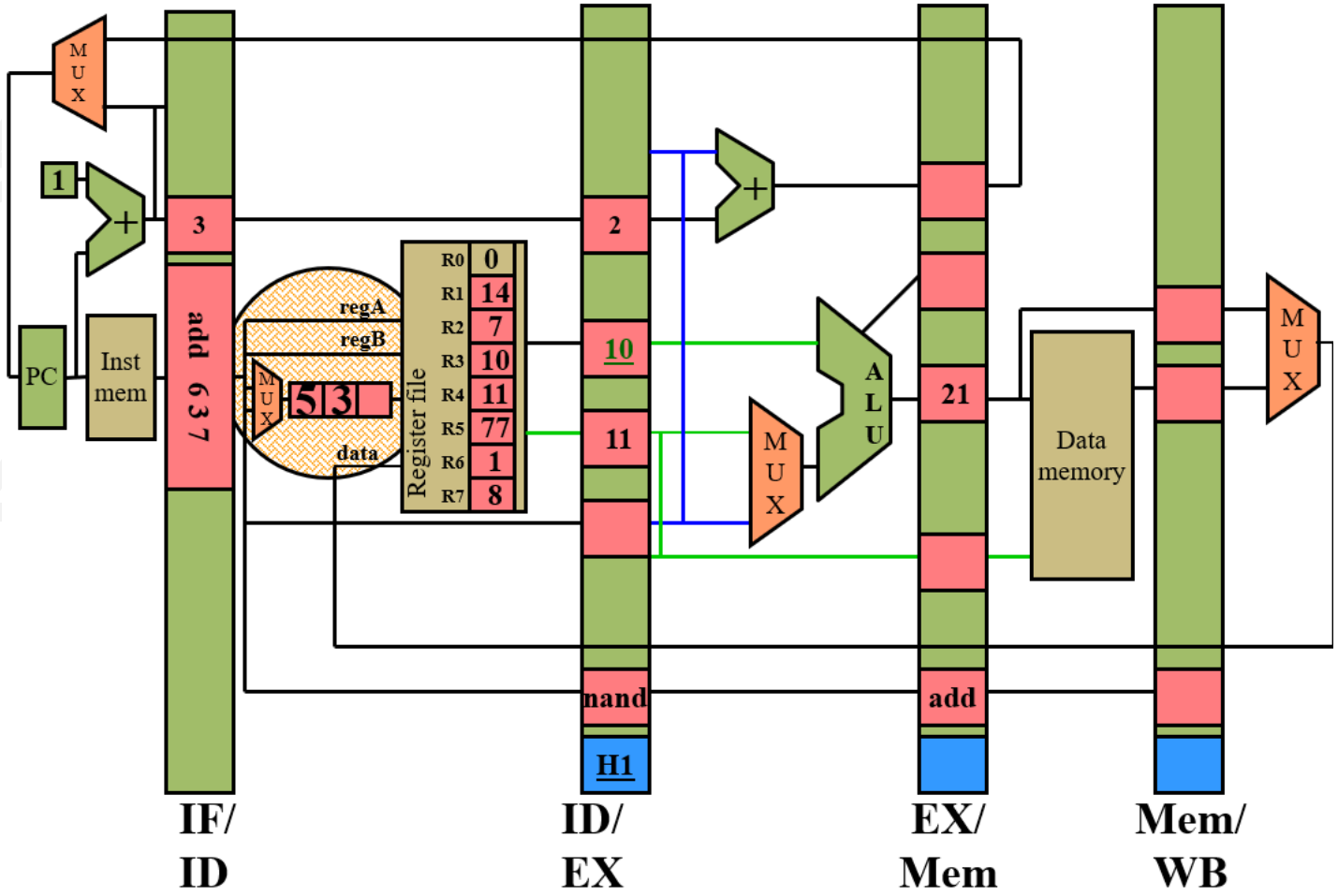


问题1：Data Hazards

- **Detect and Forward案例**

```
add 1 2 3     // r3 = r1 + r2
nand 3 4 5    // r5 = r3 NAND r4
add 6 3 7     // r7 = r3 + r6
lw 3 6 10     // r6 = MEM[r3+10]
sw 6 2 12     // MEM[r6+12]=r2
```

Cycle 3后半段

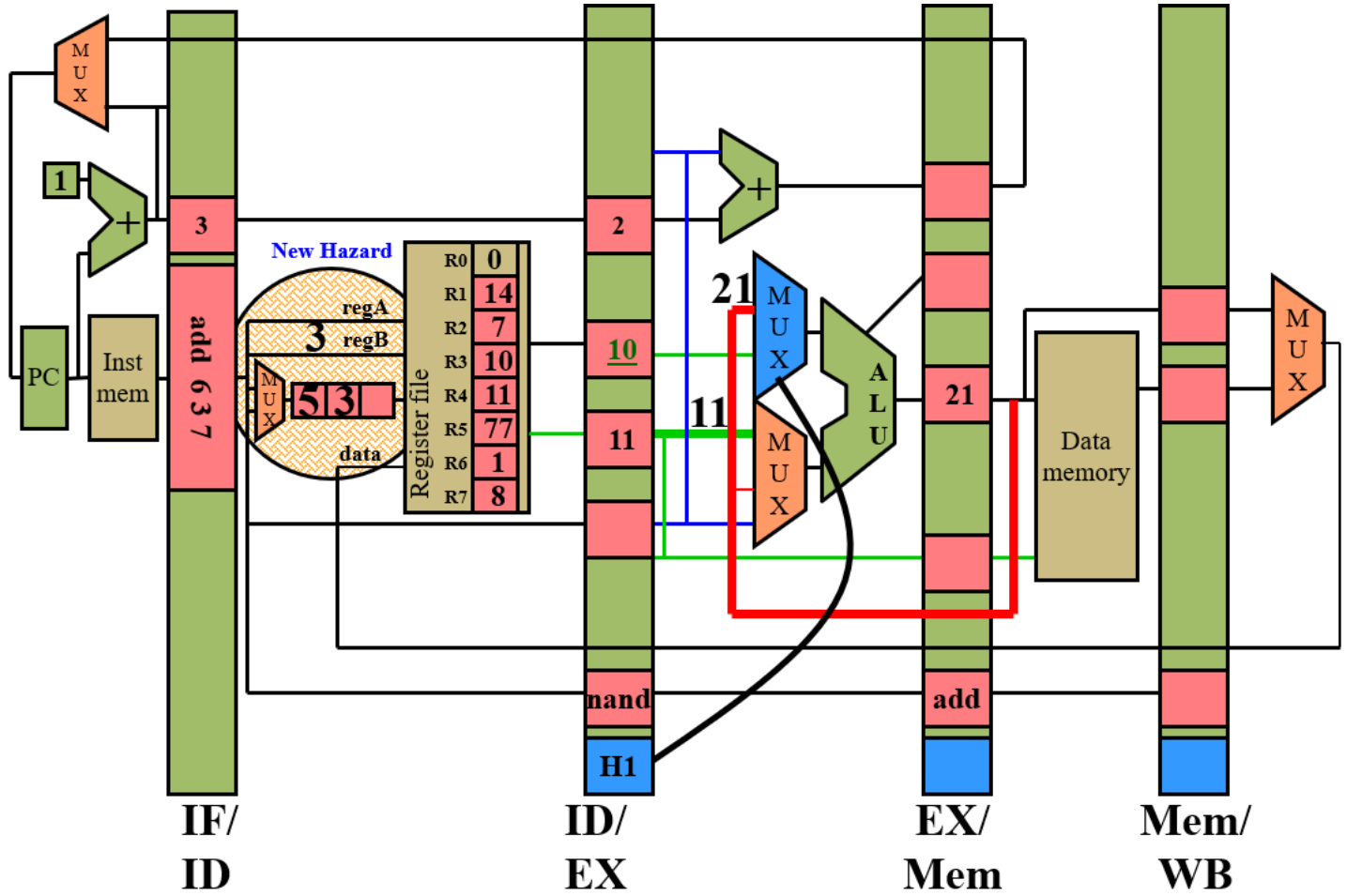


问题1：Data Hazards

- Detect and Forward案例

```
add 1 2 3     // r3 = r1 + r2
nand 3 4 5    // r5 = r3 NAND r4
add 6 3 7     // r7 = r3 + r6
lw 3 6 10     // r6 = MEM[r3+10]
sw 6 2 12     // MEM[r6+12]=r2
```

Cycle 4前半段（forwarding）

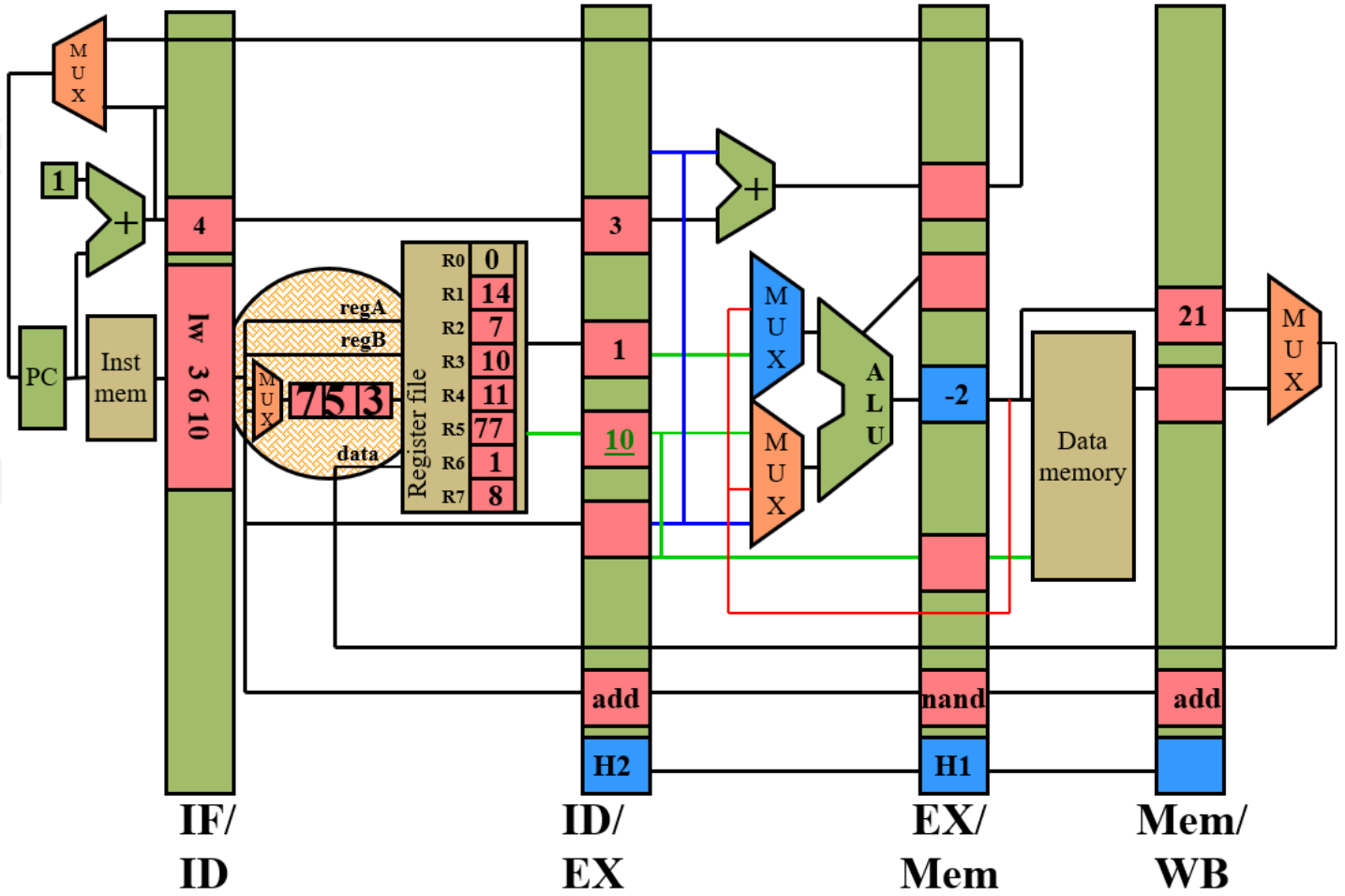


问题1：Data Hazards

- **Detect and Forward案例**

```
add 1 2 3     // r3 = r1 + r2 = 21
nand 3 4 5    // r5 = r3 NAND r4
add 6 3 7     // r7 = r3 + r6 = 22
lw 3 6 10     // r6 = MEM[r3+10]
sw 6 2 12     // MEM[r6+12]=r2
```

Cycle 4后半段

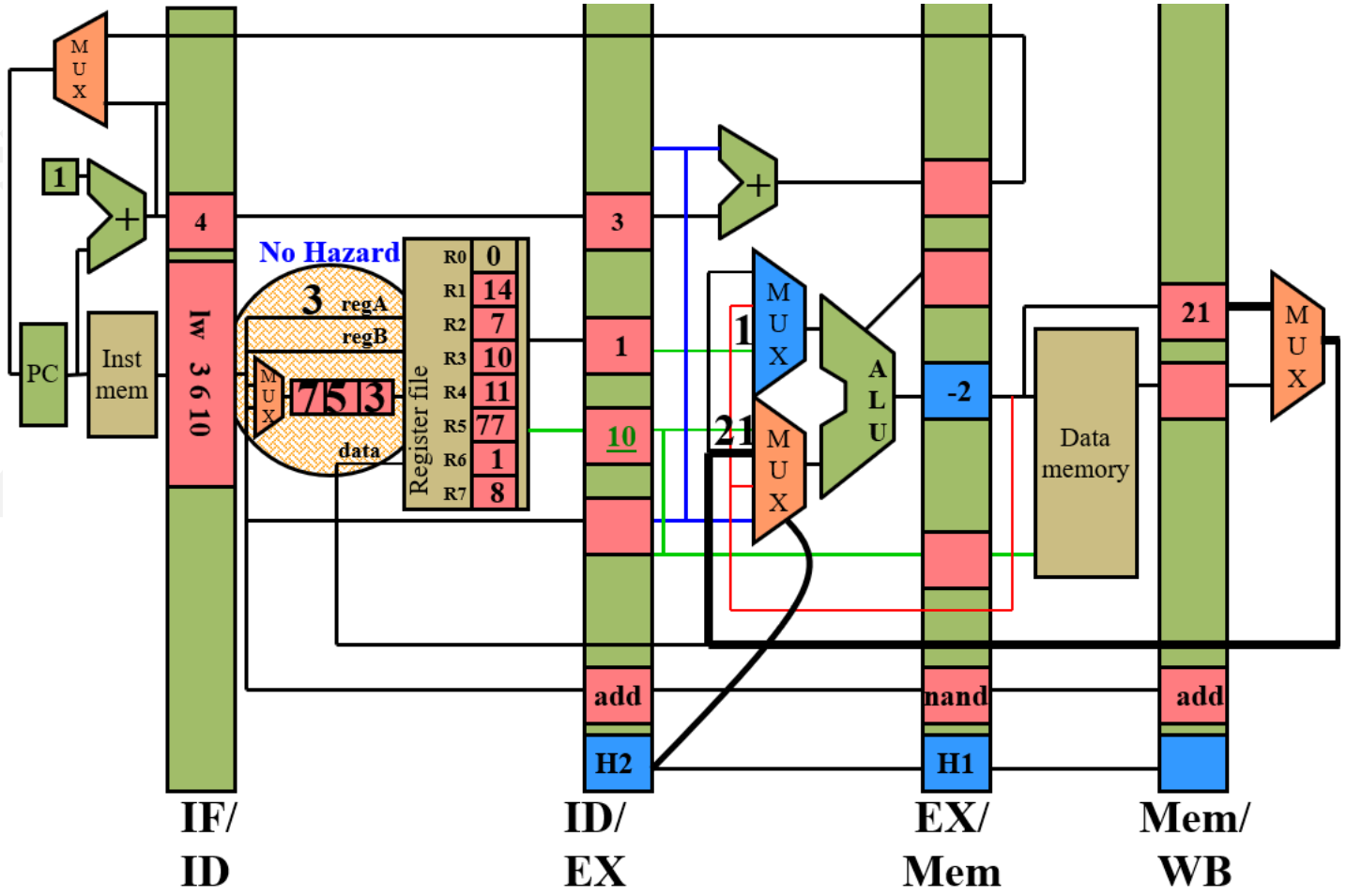


问题1：Data Hazards

- Detect and Forward案例

```
add 1 2 3     // r3 = r1 + r2
nand 3 4 5    // r5 = r3 NAND r4
add 6 3 7     // r7 = r3 + r6
lw 3 6 10     // r6 = MEM[r3+10]
sw 6 2 12     // MEM[r6+12]=r2
```

Cycle 5前半段

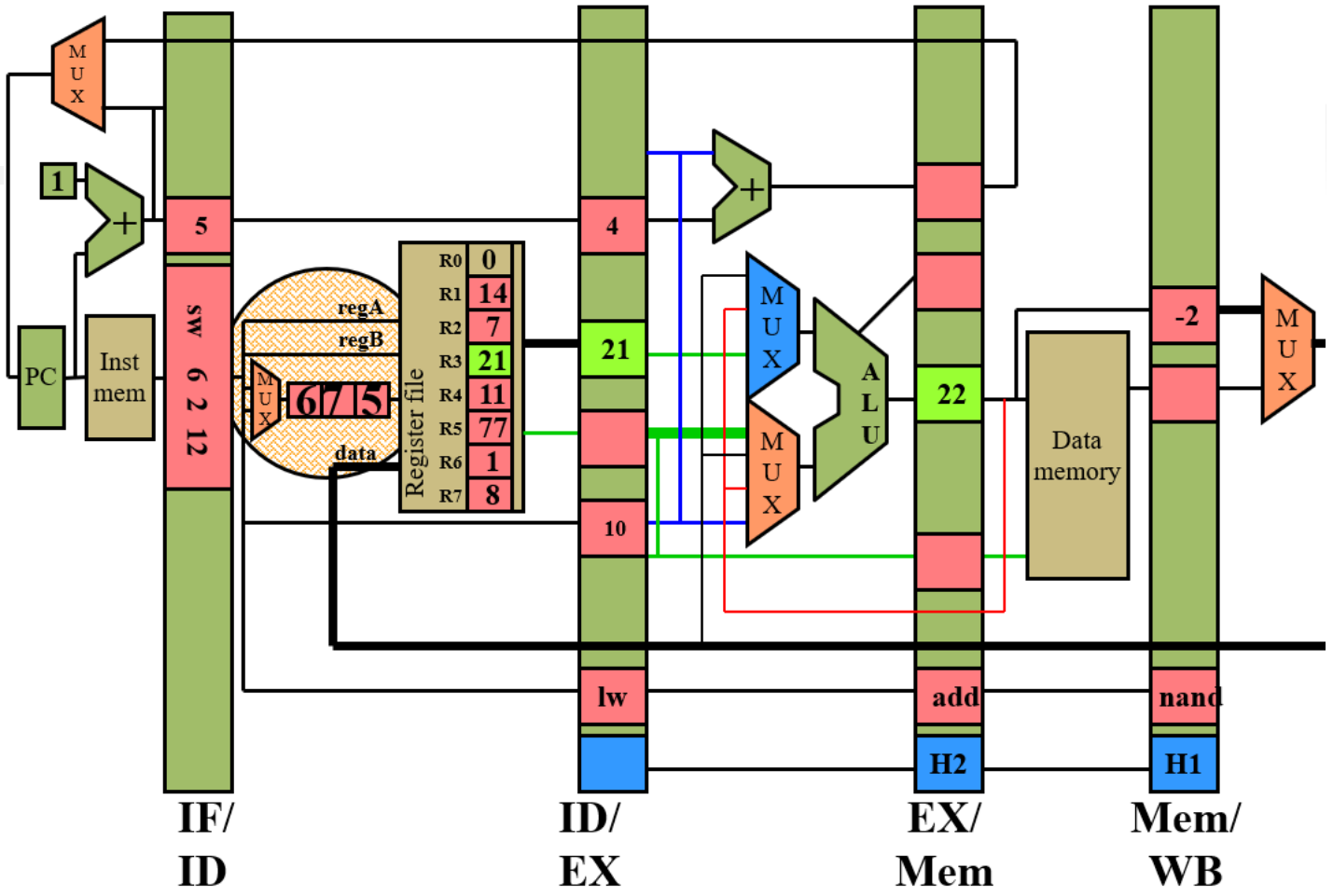


问题1：Data Hazards

- Detect and Forward案例

```
add 1 2 3     // r3 = r1 + r2
nand 3 4 5    // r5 = r3 NAND r4
add 6 3 7     // r7 = r3 + r6
lw 3 6 10     // r6 = MEM[r3+10]
sw 6 2 12     // MEM[r6+12]=r2
```

Cycle 5后半段



问题1：Data Hazards

- **WAW和WAR问题：Write After Write和Write After Read**

- False or Name dependencies
  - WAW – Write after Write

    R1=R2+R3

    R1=R4+R5
  - WAR – Write after Read

    R2=R1+R3

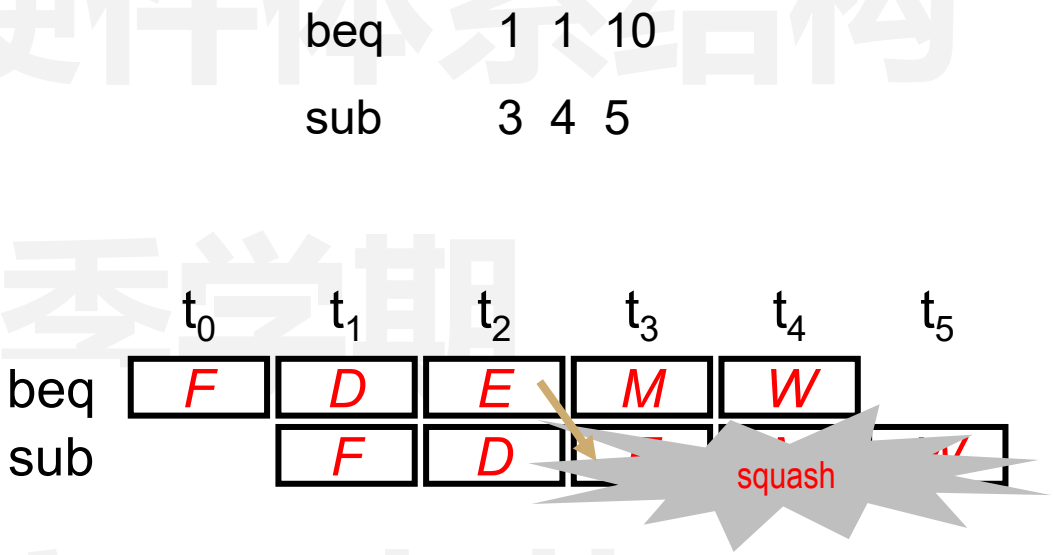
    R1=R4+R5

- 在顺序的单条5级流水线上不会出现问题
- **指令乱序执行则会出现问题，可利用Register重命名解决（后续乱序执行深入讲解）**

问题2：Control Hazards

- **Branch类指令**

- Fetch: read instruction from memory
- Decode: read source operands from reg
- Execute: calculate target address and
  test for equality
- Memory: **Send target to PC** if test is equal
- Writeback: Nothing left to do



问题2：Control Hazards

- 如何解决Control Hazards

Avoidance (static)

- No branches?
- Convert branches to predication
  - Control dependence becomes data dependence

Detect and Stall (dynamic)

- Stop fetch until branch resolves

Speculate and squash (dynamic)

- Keep going past branch, throw away instructions if wrong

问题2：Control Hazards

- **Detection and Stall：检测-停顿机制**

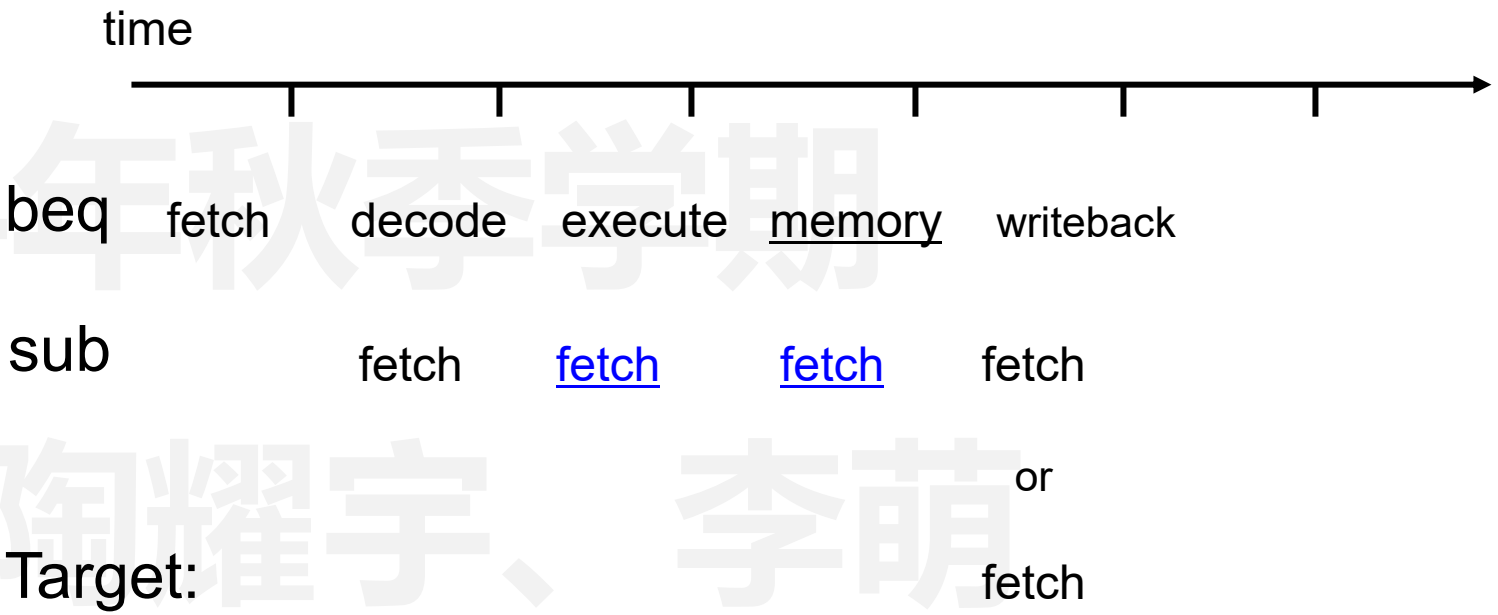
Detection

- In decode, check if opcode is branch or jump

Stall

- Hold next instruction in Fetch
- Pass noop to Decode

```
beq     1   1   10
sub     3   4   5
```



- CPI increases on every branch
- Are these stalls necessary? Not always!
  - Assume branch is NOT taken
    - Keep fetching, treat branch as noop
    - If wrong, make sure bad instructions don’ t complete

问题2：Control Hazards

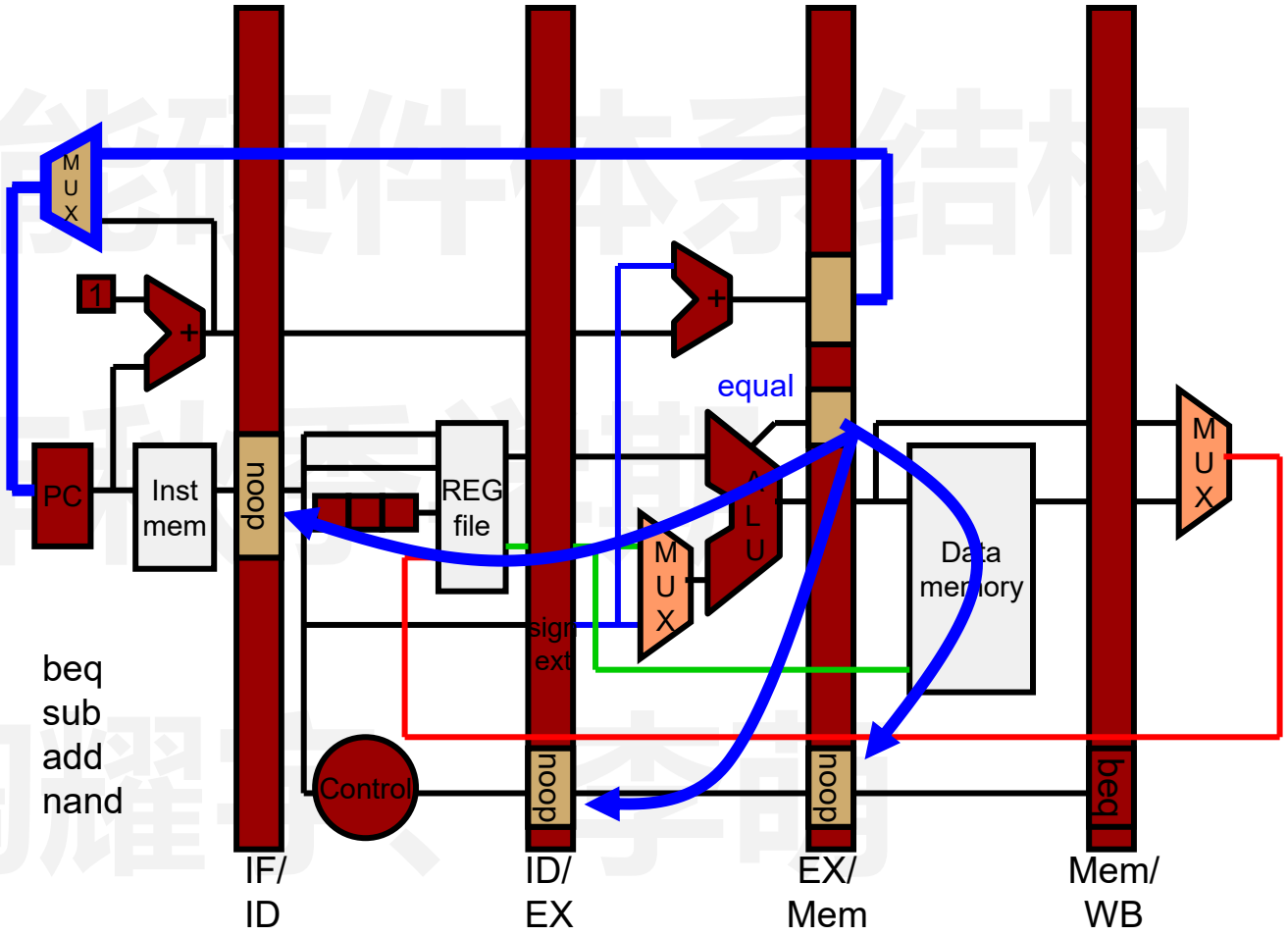
- **Speculate and Squash：投机-制止机制**

Speculate “Not-Taken”

- Assume branch is not taken

Squash

- Overwrite opcodes in Fetch, Decode, Execute with noop
- Pass target to Fetch



问题2：Control Hazards

- **Speculate and Squash：投机-制止机制的问题**

Always assumes branch is not taken

Can we do better? Yes.

- **Predict branch direction and target!**
- Why possible? Program behavior repeats.

More on branch prediction to come...

如何提高指令运行的并行度?

- Instruction-level parallelism

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

(code size) (CPI) (cycle time)

算法程序指标 体系结构指标 体系结构/电路

Architecture --> Implementation --> Realization

Compiler Designer Processor Designer Chip Designer

两大限制:

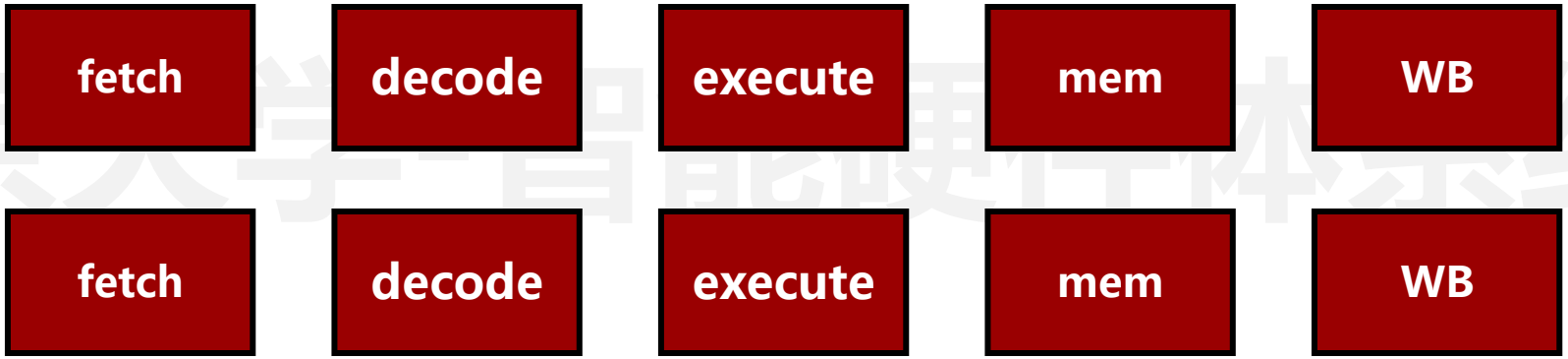
1、Upper Bound on Scalar Pipeline Throughput

2、Performance Lost Due to Rigid In-order Pipeline

Unnecessary stalls

并行度的来源

- 简单并行流水线



More complex hazard detection

- 2X pipeline registers to forward from
- 2X more instructions to check
- 2X more destinations (MUXes)
- Need to worry about dependent instructions in the same stage

Superscalar：超标量的概念

- Instruction-level parallelism

Instruction parallelism

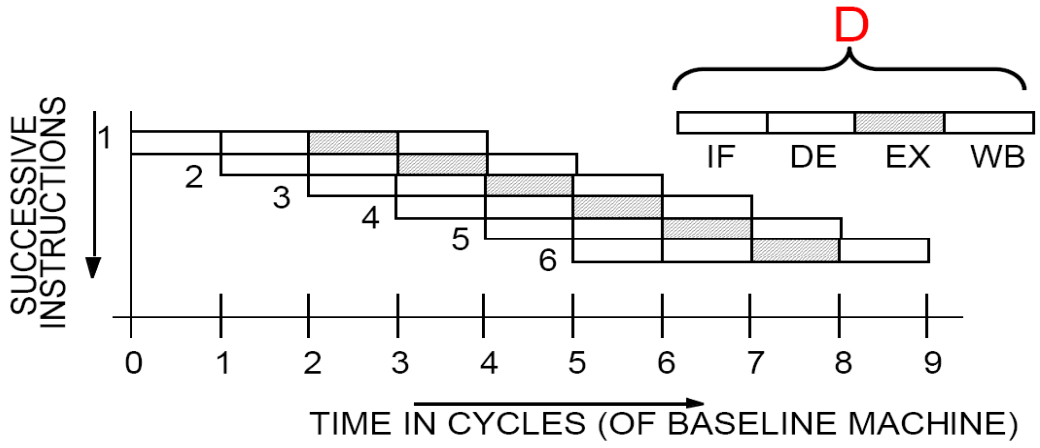
Number of instructions being worked on

Scalar Pipeline (baseline)

Instruction Parallelism = D

Operation Latency = 1

Peak IPC = 1



Peak IPC

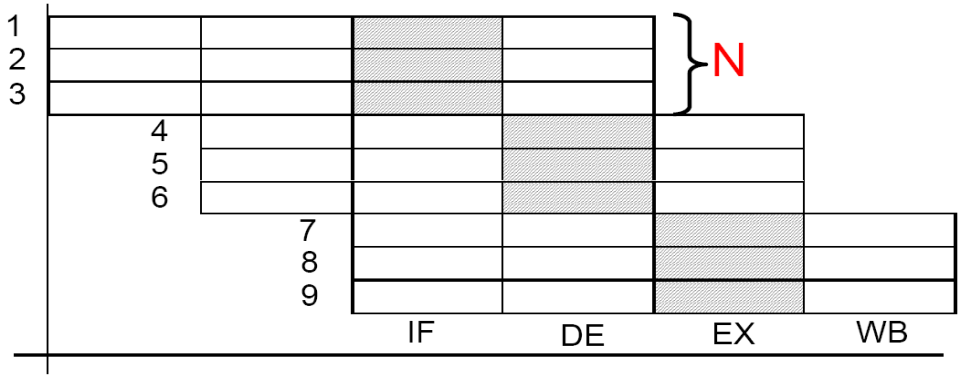
The maximum sustainable number of instructions that can be executed per clock.

Superscalar (Pipelined) Execution

IP = *DxN*

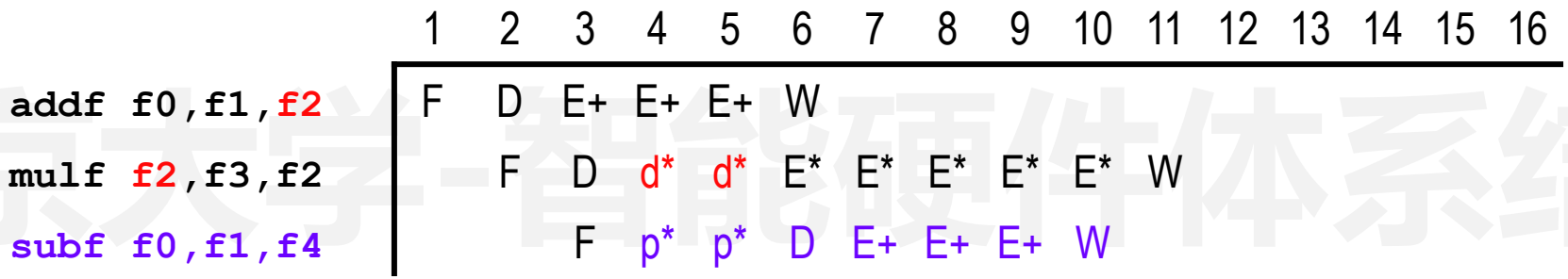
OL = *1 baseline cycles*

Peak IPC = *N per baseline cycle*



Out-Of-Order：乱序执行的概念

- **Missed Speedup in In-Order Pipelines**



What’s happening in cycle 4?

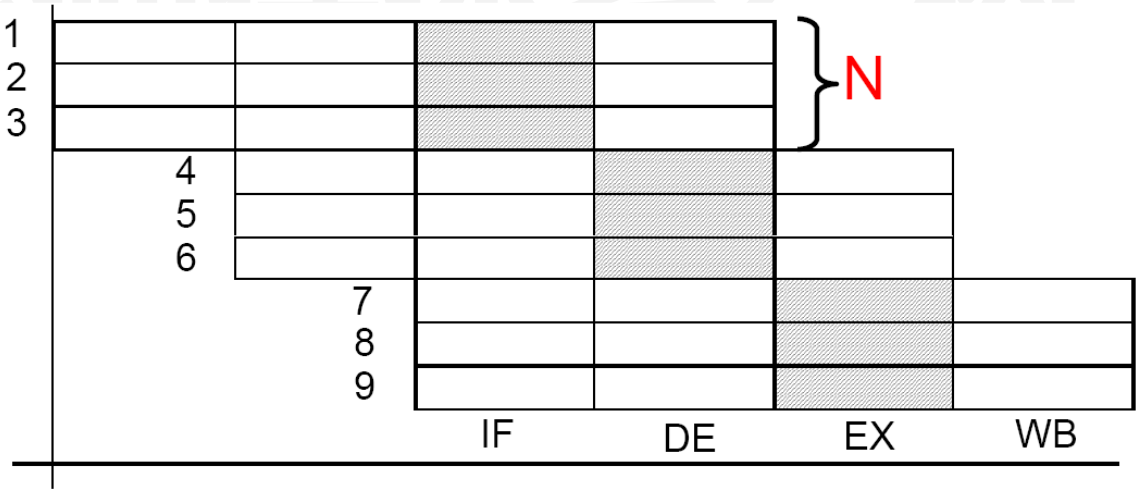
- **`mulf`** stalls due to **RAW hazard**
  - OK, this is a fundamental problem
- **`subf`** stalls due to **pipeline hazard**
  - Why? **`subf`** can’t proceed into D because **`mulf`** is there
  - That is the only reason, and it isn’t a fundamental one

Why can’t **`subf`** go into D in cycle 4 and E+ in cycle 5?

Out-Of-Order：乱序执行的概念

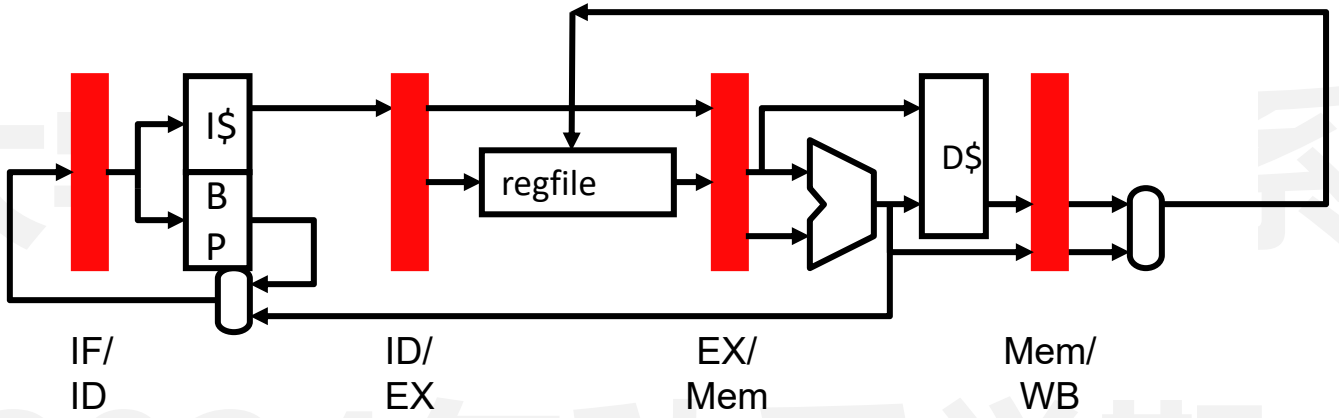
- **Instruction-level parallelism**

- CPI of in-order pipelines degrades sharply if the machine parallelism is increased beyond a certain point.
  - *when NxM approaches average distance between dependent instructions*
- Forwarding is no longer effective
  - *Pipeline may never be full due to frequent dependency stalls!*



Out-Of-Order：乱序执行的概念

- The Problem With In-Order Pipelines



- **In-order pipeline**
  - **Structural hazard**: 1 insn register (latch) per stage
    - 1 instruction per stage per cycle (unless pipeline is replicated)
    - Younger instr. can't "pass" older instr. without "clobbering" it
- **Out-of-order pipeline**
  - Implement "passing" functionality by removing **structural hazard**

Out-Of-Order：乱序执行的概念

北京大学 PEKING UNIVERSITY

- **乱序执行完全在硬件实现**

- Dynamic scheduling
  - Totally in the hardware
  - Also called “out-of-order execution” (OoO)
- Fetch many instructions into instruction window
  - Use branch prediction to speculate past (multiple) branches
  - Flush pipeline on branch misprediction
- Rename to avoid false dependencies (WAW and WAR)
- Execute instructions as soon as possible
  - Register dependencies are known
  - Handling memory dependencies more tricky (much more later)
- Commit instructions in order
  - Any strange happens before commit, just flush the pipeline
- Current machines: 100+ instruction scheduling window

Out-of-order execution

Execute instructions in non-sequential order…

+Reduce RAW stalls

+Increase pipeline and functional unit (FU) utilization

Original motivation was to increase FP unit utilization

+Expose more opportunities for parallel issue (ILP)

Not in-order → can be in parallel

…but make it appear like sequential execution

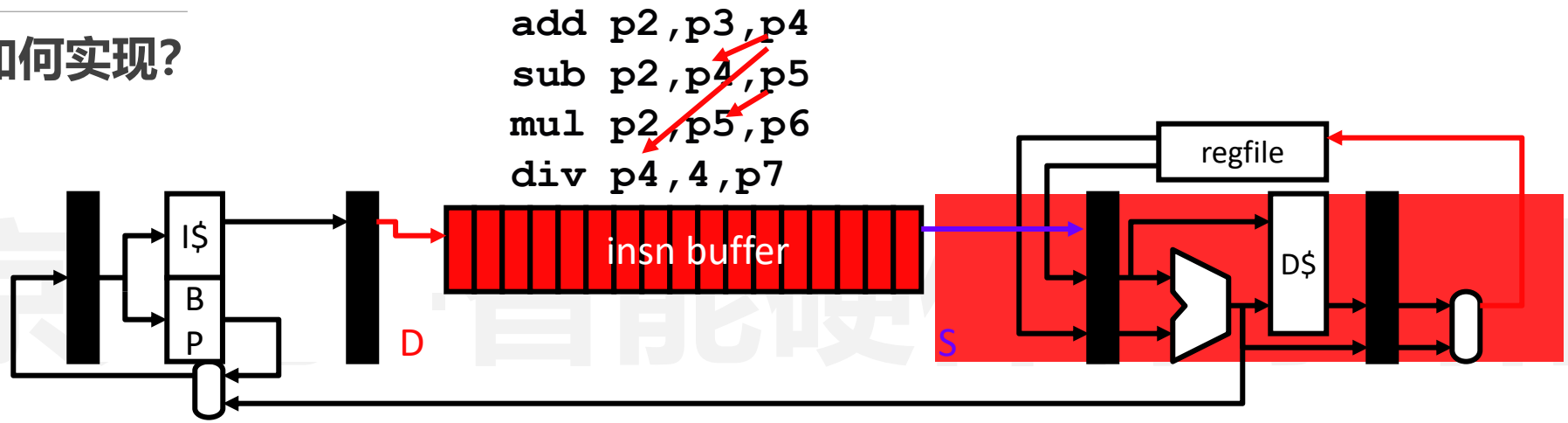
Important

–But difficult

Next few lectures

Out-Of-Order：乱序执行的概念

- 乱序执行如何实现?



Ready Table

P2	P3	P4	P5	P6	P7
Yes	Yes				
Yes	Yes	Yes			
Yes	Yes	Yes	Yes		Yes
Yes	Yes	Yes	Yes	Yes	Yes

t

```
add p2,p3,p4
sub p2,p4,p5  and div p4,4,p7
mul p2,p5,p6
```

- Instructions fetch/decoded/renamed into *Instruction Buffer*
  - Also called "instruction window" or "instruction scheduler"
- Instructions (conceptually) check ready bits every cycle
  - Execute when ready

数据依赖与数据冲突

- 数据依赖存在于原始任务逻辑，与硬件体系结构如何设计无关

- A dependency exists *independent* of the hardware.
  - So if Inst #1′ s result is needed for Inst #1000 there is a dependency
  - It is only a *hazard* if the hardware has to deal with it.
    - So in our pipelined machine we only worried if there wasn′ t a “buffer” of two instructions between the dependent instructions.

数据依赖

- True/False Data Dependencies

- True data dependency
  - RAW – Read after Write

    R1 = R2 + R3

    R4 = R1 + R5

- True dependencies prevent reordering
  - (Mostly) unavoidable

- False or Name dependencies
  - WAW – Write after Write

    R1 = R2 + R3

    ↓

    R1 = R4 + R5
  - WAR – Write after Read

    R2 = R1 + R3

    ↓

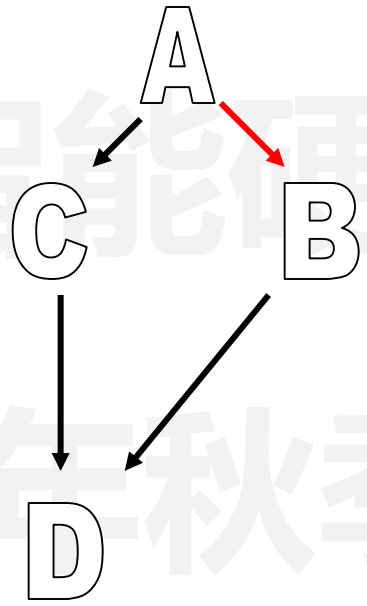
    R1 = R4 + R5

- False dependencies prevent reordering
  - Can they be eliminated? (Yes, with renaming!)

数据依赖图

- True/False Data Dependencies

```
R1=MEM[R2+0]    // A
R2=R2+4         // B
R3=R1+R4        // C
MEM[R2+0]=R3    // D
```

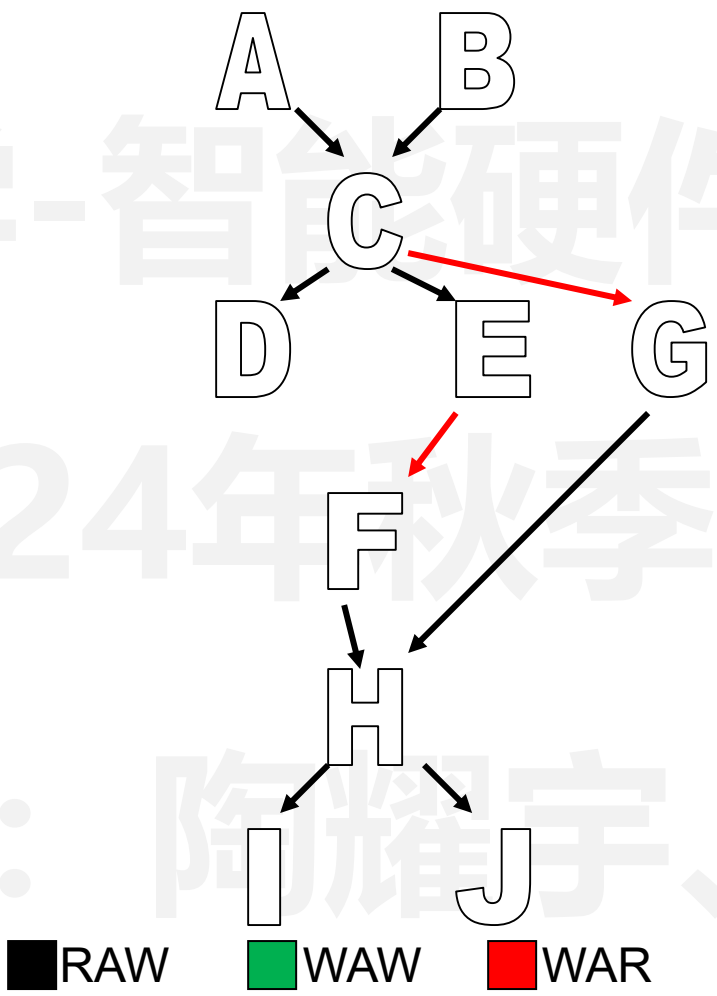


RAW WAW WAR

数据依赖图

- True/False Data Dependencies

```
R1=MEM[R3+4]     // A
R2=MEM[R3+8]     // B
R1=R1*R2         // C
MEM[R3+4]=R1     // D
MEM[R3+8]=R1     // E
R1=MEM[R3+12]    // F
R2=MEM[R3+16]    // G
R1=R1*R2         // H
MEM[R3+12]=R1    // I
MEM[R3+16]=R1    // J
```

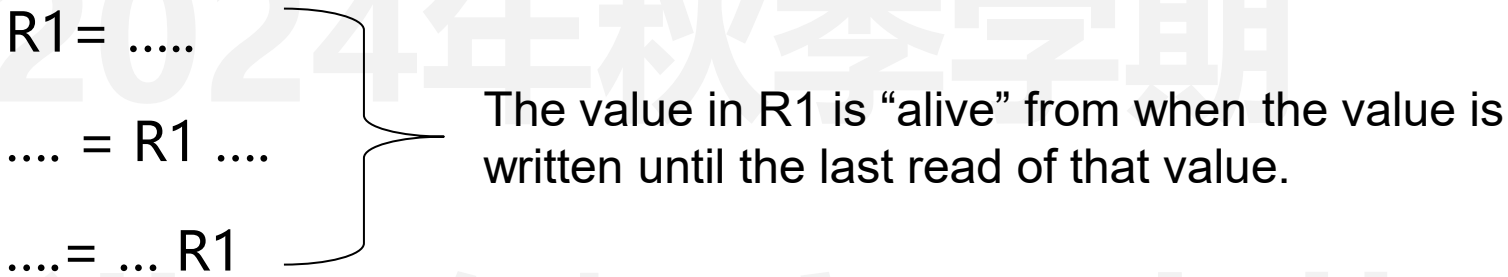


- Well, logically there is no reason for F-J to be dependent on A-E. So…..
  - ABFG
  - CH
  - DEIJ
  - Should be possible.
- But that would cause either C or H to have the wrong reg inputs
- How do we fix this?
  - Remember, the dependency is really on the *name* of the register
  - **So… change the register names!**

触发器重命名

- 触发器Register重命名概念

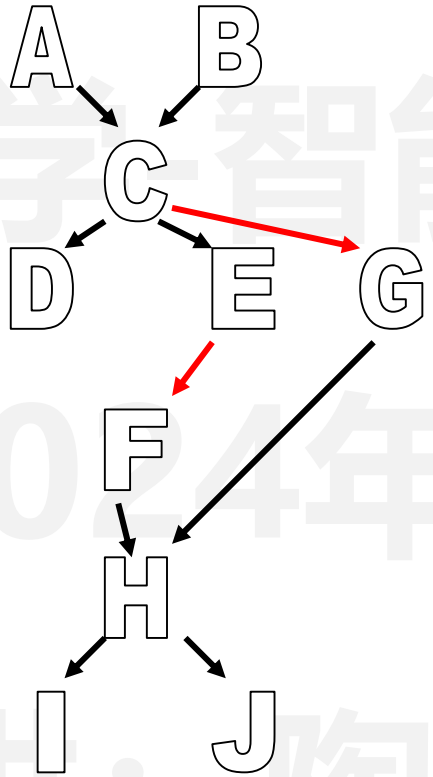
- The register names are arbitrary
- The register name only needs to be consistent between writes.



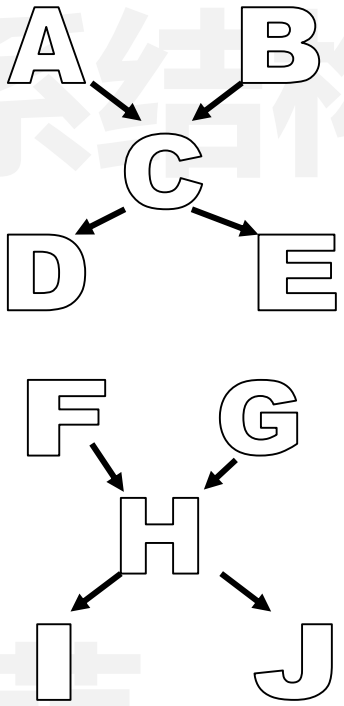
触发器重命名

- 触发器Register重命名的效果

```
R1=MEM[R3+4]     // A
R2=MEM[R3+8]     // B
R1=R1*R2         // C
MEM[R3+4]=R1     // D
MEM[R3+8]=R1     // E
R1=MEM[R3+12]    // F
R2=MEM[R3+16]    // G
R1=R1*R2         // H
MEM[R3+12]=R1    // I
MEM[R3+16]=R1    // J
```




```
P1=MEM[R3+4]     //A
P2=MEM[R3+8]     //B
P3=P1*P2         //C
MEM[R3+4]=P3     //D
MEM[R3+8]=P3     //E
P4=MEM[R3+12]    //F
P5=MEM[R3+16]    //G
P6=P4*P5         //H
MEM[R3+12]=P6    //I
MEM[R3+16]=P6    //J
```



RAW WAW WAR

触发器重命名

北京大学 PEKING UNIVERSITY

- **触发器Register重命名机制**

  - Every time an architecture register is written we assign it to a physical register
    - Until the architected register is written again, we continue to translate it to the physical register number
    - Leaves **RAW** dependencies intact
  - It is really simple, let's look at an example:
    - Names: `r1,r2,r3`
    - Locations: `p1,p2,p3,p4,p5,p6,p7`
    - Original mapping: `r1→p1`, `r2→p2`, `r3→p3`, `p4–p7` are "free"

Architecture register
虚拟的架构触发器

Physical register
实际的电路触发器

MapTable

r1	r2	r3
p1	p2	p3
p4	p2	p3
p4	p2	p5
p4	p2	p6

FreeList

p4,p5,p6,p7
p5,p6,p7
p6,p7
p7

Orig. insns

```
add r2,r3,r1
sub r2,r1,r3
mul r2,r3,r3
div r1,4,r1
```

Renamed insns

```
add p2,p3,p4
sub p2,p4,p5
mul p2,p5,p6
div p4,4,p7
```