



北京大学
PEKING UNIVERSITY

智能硬件体系结构

第四讲：指令集与流水线架构



主讲：陶耀宇、李萌

2024年秋季

注意事项

• 课程作业情况

- 第1次作业提交截止日期是**10月15日**

老师Office Hour: 资源西楼2208b, 周五、周一下午可预约时间

助教Office Hour: 周五下午, 具体时间助教会放出

- 第2次作业将在**10月16日**放出 (简化版)
- 第1次编程作业 (简化版, 延迟一周放出) **10月16号~11月16号**

目录

CONTENTS



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

为什么需要指令集?

- 指令集可以看做链接软件和硬件的一个协议



ISA (instruction set architecture)

A well-defined hardware/software interface

The “**contract**” between software and hardware

- Functional definition** of operations, modes, and storage locations supported by hardware
- Precise description** of how to invoke, and access them

No guarantees regarding

- How operations are implemented
- Which operations are fast and which are slow and when
- Which operations take more power and which take less

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
- Programmer-visible states
 - Program counter, general purpose registers, memory, control registers
- Programmer-visible behaviors (state transitions)
 - What to do, when to do it
- A binary encoding

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]
```

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

...be careful what goes in

指令集的分类

- RISC和CISC两种指令集
- Recall “Iron” law:
 - $(\text{instructions}/\text{program}) * (\text{cycles}/\text{instruction}) * (\text{seconds}/\text{cycle})$
- CISC (Complex Instruction Set Computing) 例如X86等商用指令集
 - Improve “instructions/program” with “complex” instructions
 - Easy for assembly-level programmers, good code density
- RISC (Reduced Instruction Set Computing) 例如MIPS/ARM/RISC-V
 - Improve “cycles/instruction” with many single-cycle instructions
 - Increases “instruction/program”, but hopefully not as much
 - Help from smart compiler
 - Perhaps improve clock cycle time (seconds/cycle)
 - via aggressive implementation allowed by simpler instructions

指令集设计思路

- 兼顾软件可编程性、硬件可实现性和兼容性

• Programmability

- Easy to express programs efficiently?

• Implementability

- Easy to design high-performance implementations?

- More recently

- Easy to design low-power implementations?

- Easy to design high-reliability implementations?

- Easy to design low-cost implementations?

• Compatibility

- Easy to maintain programmability (implementability) as languages and programs evolves?

- 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

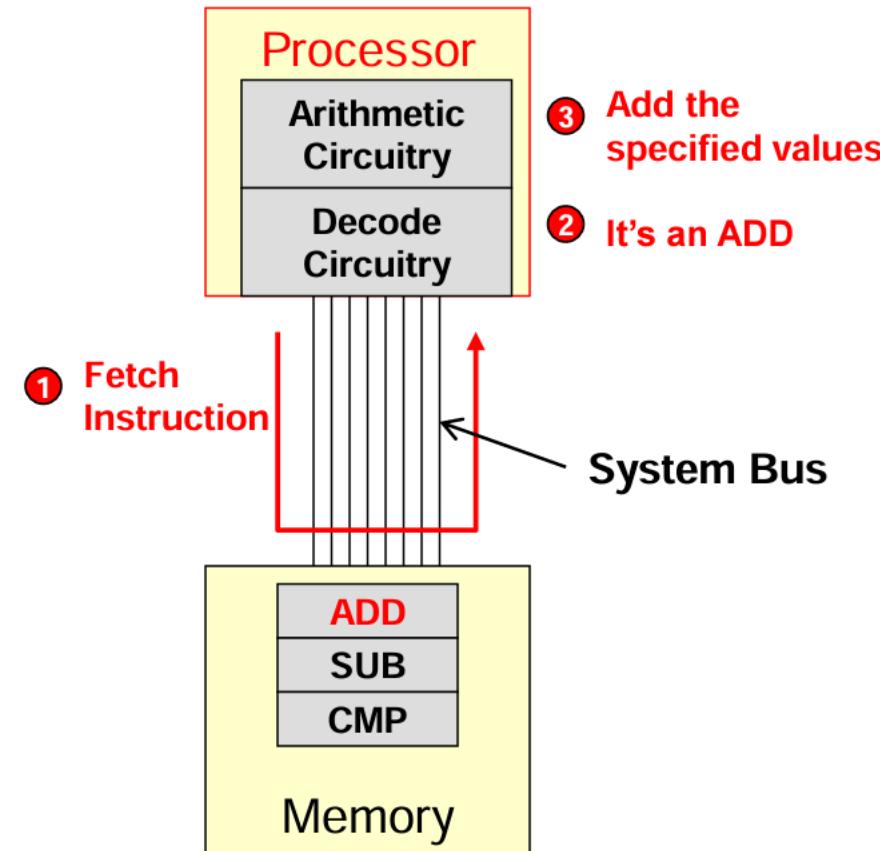
```
Disassembly of section .text:
0000000000000000 <_Z3absiPi>:
 0: 85 ff  test   %edi,%edi
 2: 79 05  jns    9 <_Z3absiPi+0x9>
 4: f7 df  neg    %edi
 6: 89 3e  mov    %edi,(%rsi)
 8: c3    retq
 9: 89 3e  mov    %edi,(%rsi)
 b:  c3    retq
```

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

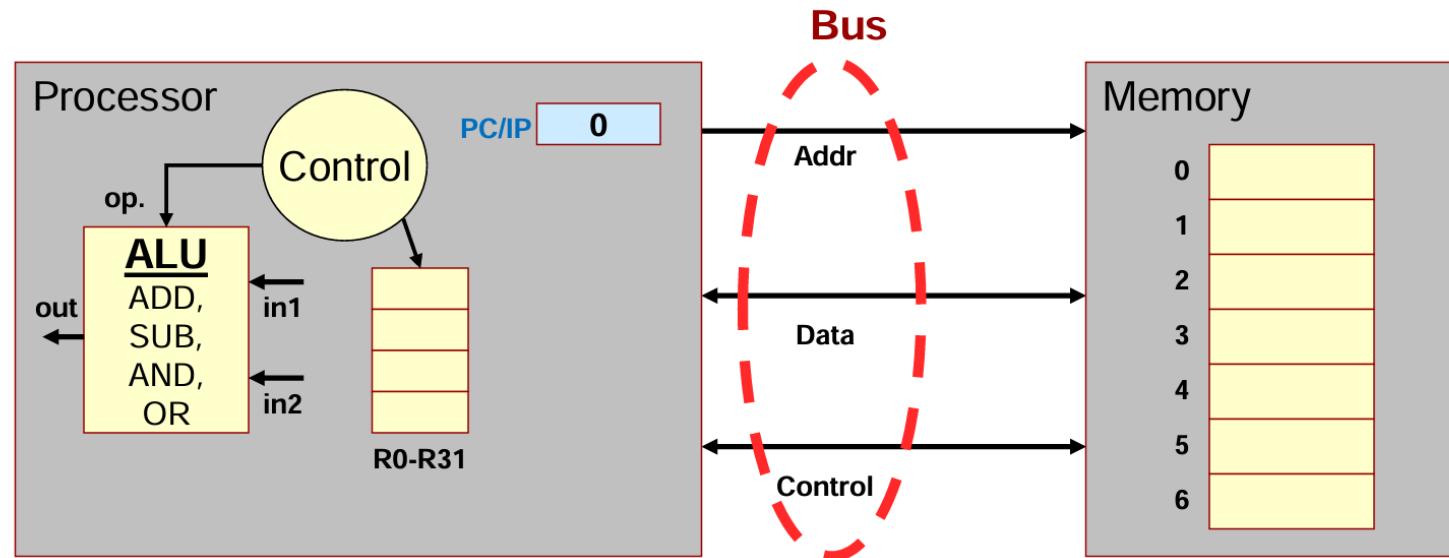
- Performs the same 3-step process over and over again
 - Fetch an instruction from memory
 - Decode the instruction
 - Is it an ADD, SUB, etc.?
 - Execute the instruction
 - Perform the specified operation
- This process is known as the **Instruction Cycle**



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

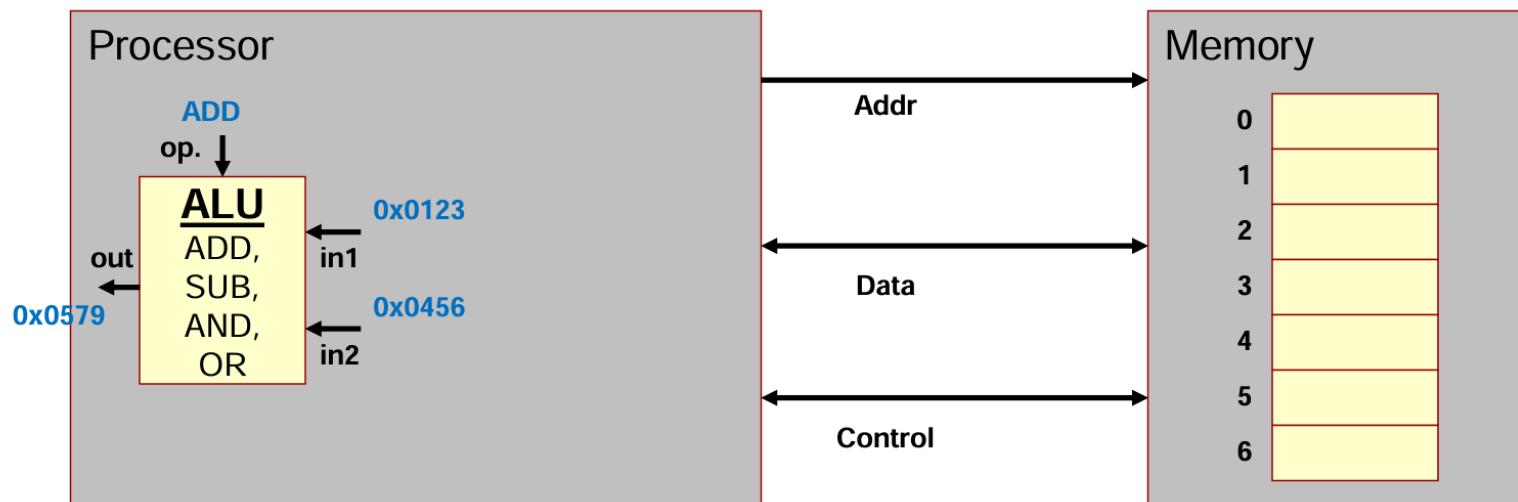
- 3 Primary Components inside a processor
 - ALU
 - Registers
 - Control Circuitry
- Connects to memory and I/O via **address**, **data**, and **control** buses (**bus** = group of wires)



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

- Digital circuit that performs arithmetic operations like addition and subtraction along with logical operations (AND, OR, etc.)

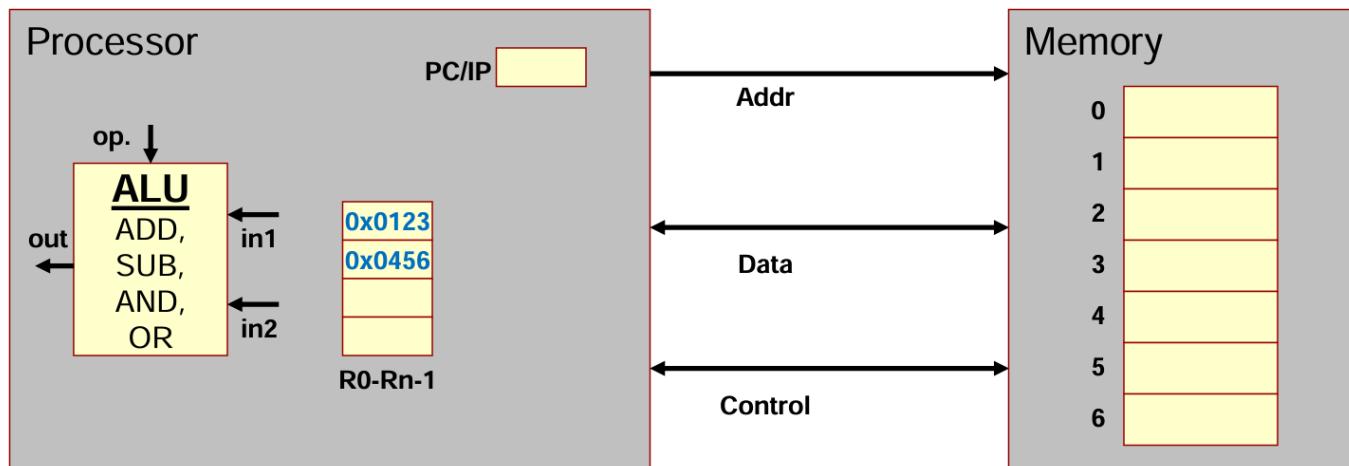


传统存算分离的指令集架构 – 核心部件1：Register

- Register负责将ALU运算结果暂存在靠近ALU的地方

- Recall memory is **SLOW** compared to a processor
- Registers provide **fast, temporary** storage locations within the processor

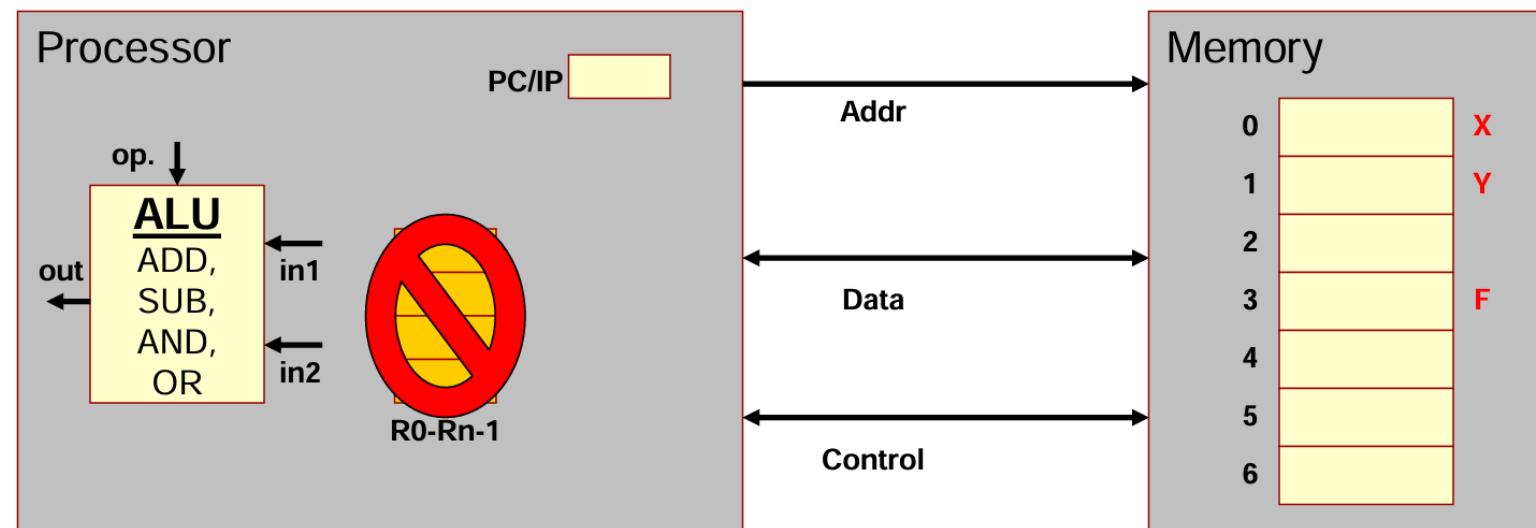
- Registers available to software instructions for use by the programmer/compiler
- Programmer/compiler is in charge of using these registers **as inputs (source locations) and outputs (destination locations)**



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

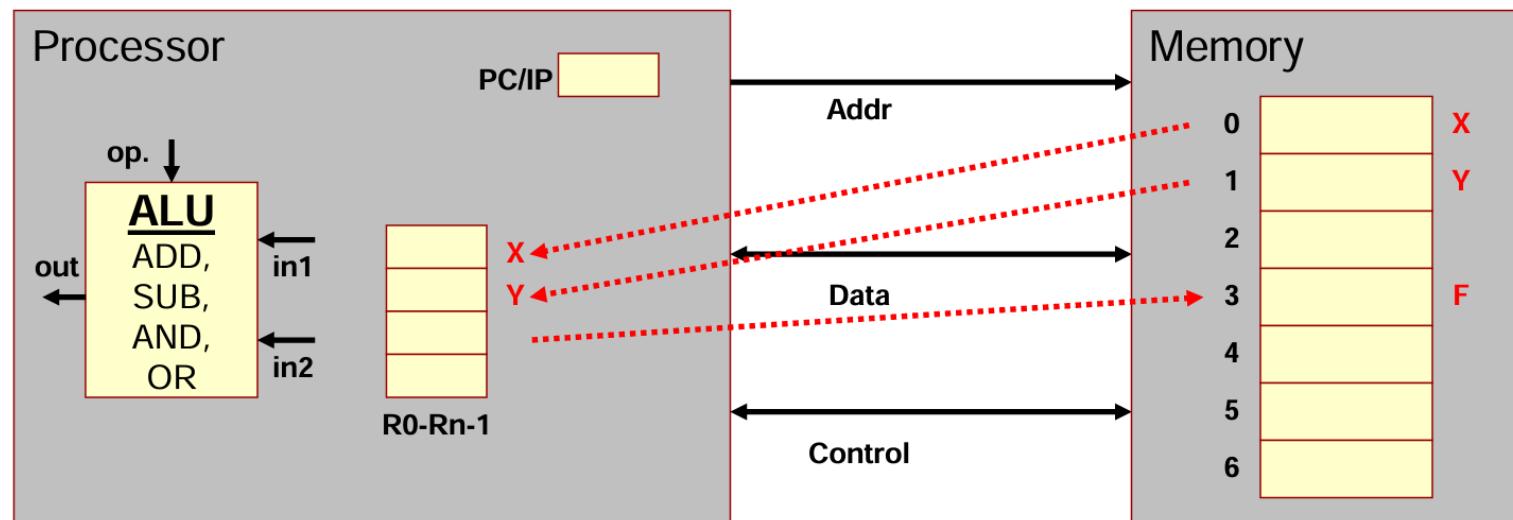
- Example w/o registers: $F = (X+Y) - (X*Y)$
 - Requires an ADD instruction, MULtiply instruction, and SUBtract Instruction
 - w/o registers
 - ADD: Load X and Y from memory, store result to memory
 - MUL: Load X and Y again from mem., store result to memory
 - SUB: Load results from ADD and MUL and store result to memory
 - 9 memory accesses



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

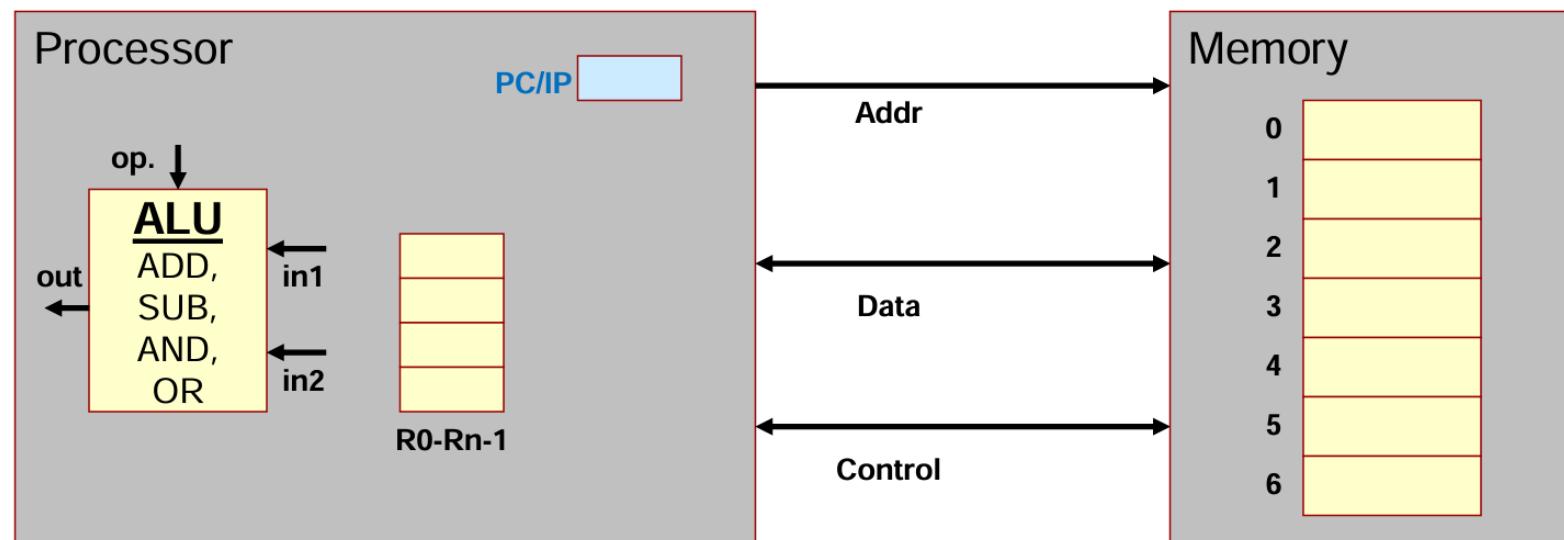
- Example w/ registers: $F = (X+Y) - (X*Y)$
 - Load X and Y into registers
 - ADD: R0 + R1 and store result in R2
 - MUL: R0 * R1 and store result in R3
 - SUB: R2 – R3 and store result in R4
 - Store R4 back to memory
 - 3 total memory access



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

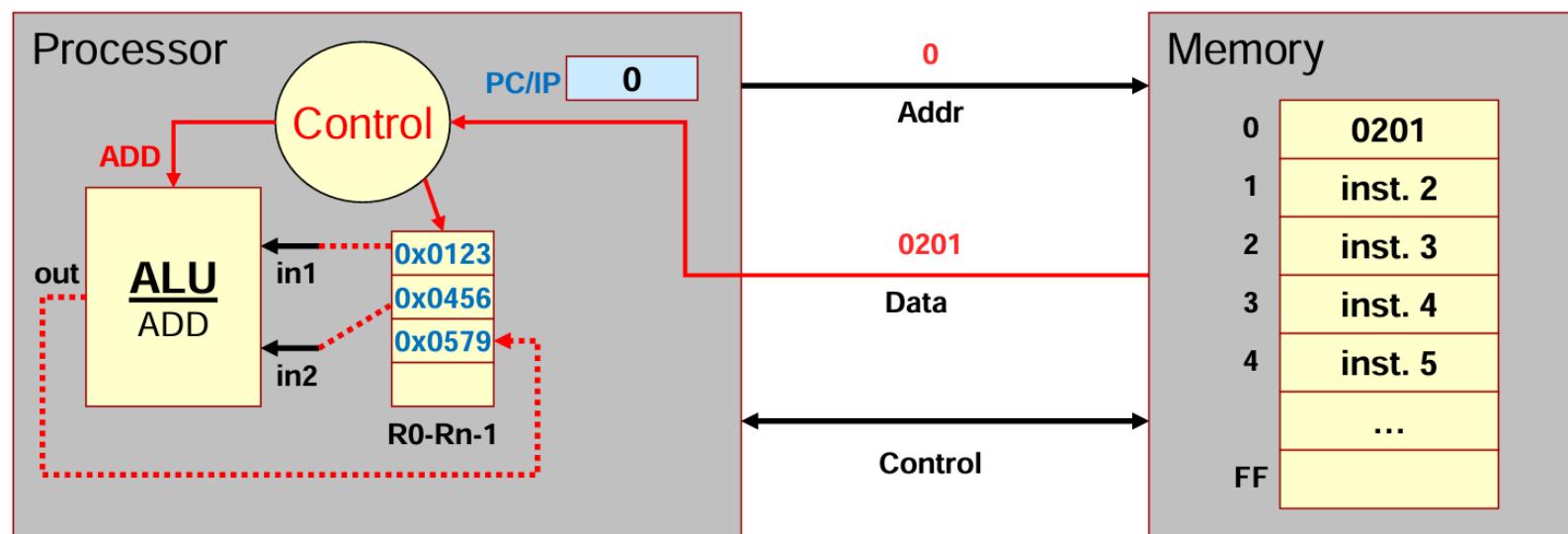
- Some bookkeeping information is needed to make the processor operate correctly
- Example: Program Counter/Instruction Pointer (PC/IP) Reg.
 - Recall that the processor must fetch instructions from memory before decoding and executing them
 - PC/IP register holds the address of the next instruction to fetch



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）

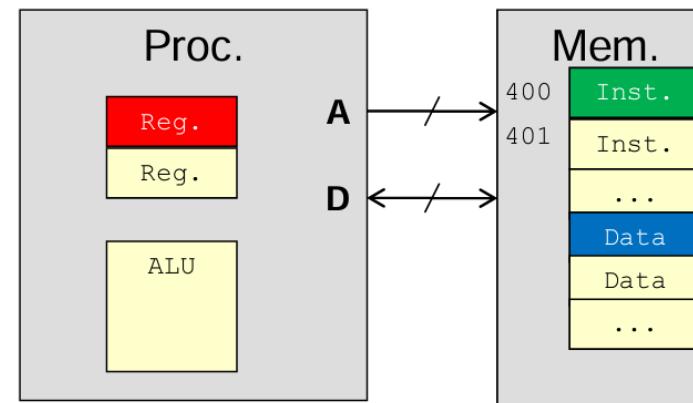
- Assume 0x0201 is machine code for an ADD instruction of R2
 $= R0 + R1$
- Control Logic will...
 - select the registers (R0 and R1)
 - tell the ALU to add
 - select the destination register (R2)



指令集数据的位置

- 数据可以存储在register、主存memory或指令内部

- Source operands must be in one of the following 3 locations:
 - A register value (e.g. %rax)
 - A value in a memory location (e.g. value at address 0x0200e8)
 - A constant stored in the instruction itself (known as an ‘immediate’ value)
[e.g. ADDI \$1,D0]
 - The \$ indicates the constant/immediate
- Destination operands must be
 - A register
 - A memory location (specified by its address)



目录

CONTENTS



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

主流指令集都有哪些？

- X86、MIPS、ARM、RISC-V

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

指令集架构一般需要哪些指令？

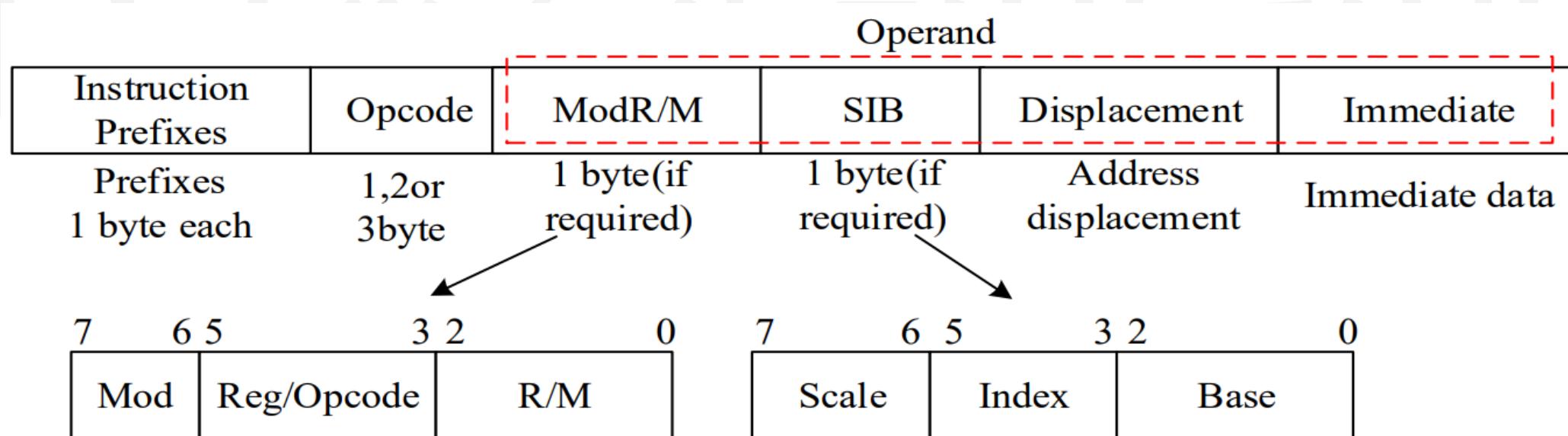
• 四大类：传输指令、运算指令、控制指令、系统指令

- Data Transfer (`mov` instruction)
 - Moves data between processor & memory (loads and saves variables between processor and memory)
 - One operand must be a processor register (can't move data from one memory location to another)
 - Specifies size via a suffix on the instruction (`movb`, `movw`, `movl`, `movq`)
- ALU Operations
 - One operand must be a processor register
 - Size and operation specified by instruction (`addl`, `orq`, `andb`, `subw`)
- Control / Program Flow
 - Unconditional/Conditional Branch (`cmpq`, `jmp`, `je`, `jne`, `jl`, `jge`)
 - Subroutine Calls (`call`, `ret`)
- Privileged / System Instructions
 - Instructions that can only be used by OS or other “supervisor” software (e.g. `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）



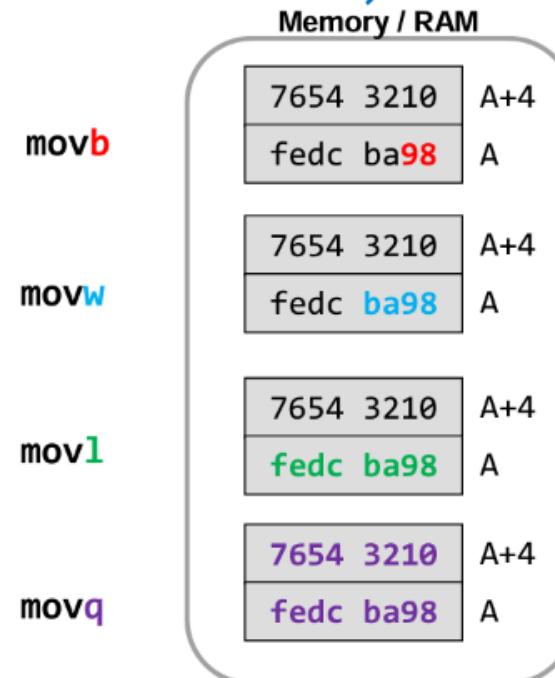
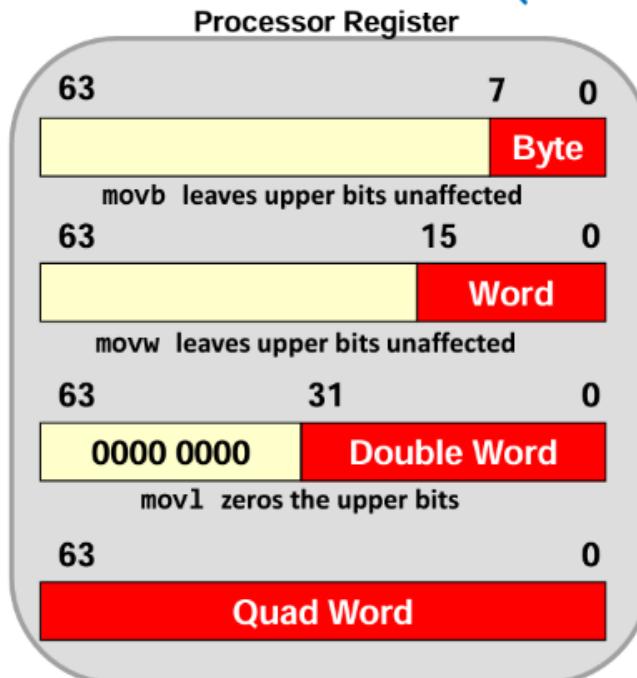
指令集的分类1 – 传输指令

- 指令集可以看做链接软件和硬件的一个协议

- Moves data between memory and processor register**

- Size is explicitly defined by the instruction suffix ('mov[bwlq]') used
- Recall: Start address **should** be divisible by size of access

(Assume start address = A)



Byte operations only access the **1-byte** at the specified address

Word operations access the **2-bytes starting** at the specified address

Word operations access the **4-bytes starting** at the specified address

Word operations access the **8-bytes starting** at the specified address

指令集的分类1 – 传输指令：指令的地址模式

- 指令集一般包含多种地址模式 – 以广泛商用的X86或MIPS为案例

x64 assembly code uses sixteen 64-bit registers. Additionally, the lower bytes of some of these registers may be accessed independently as 32-, 16- or 8-bit registers. The register names are as follows:

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]

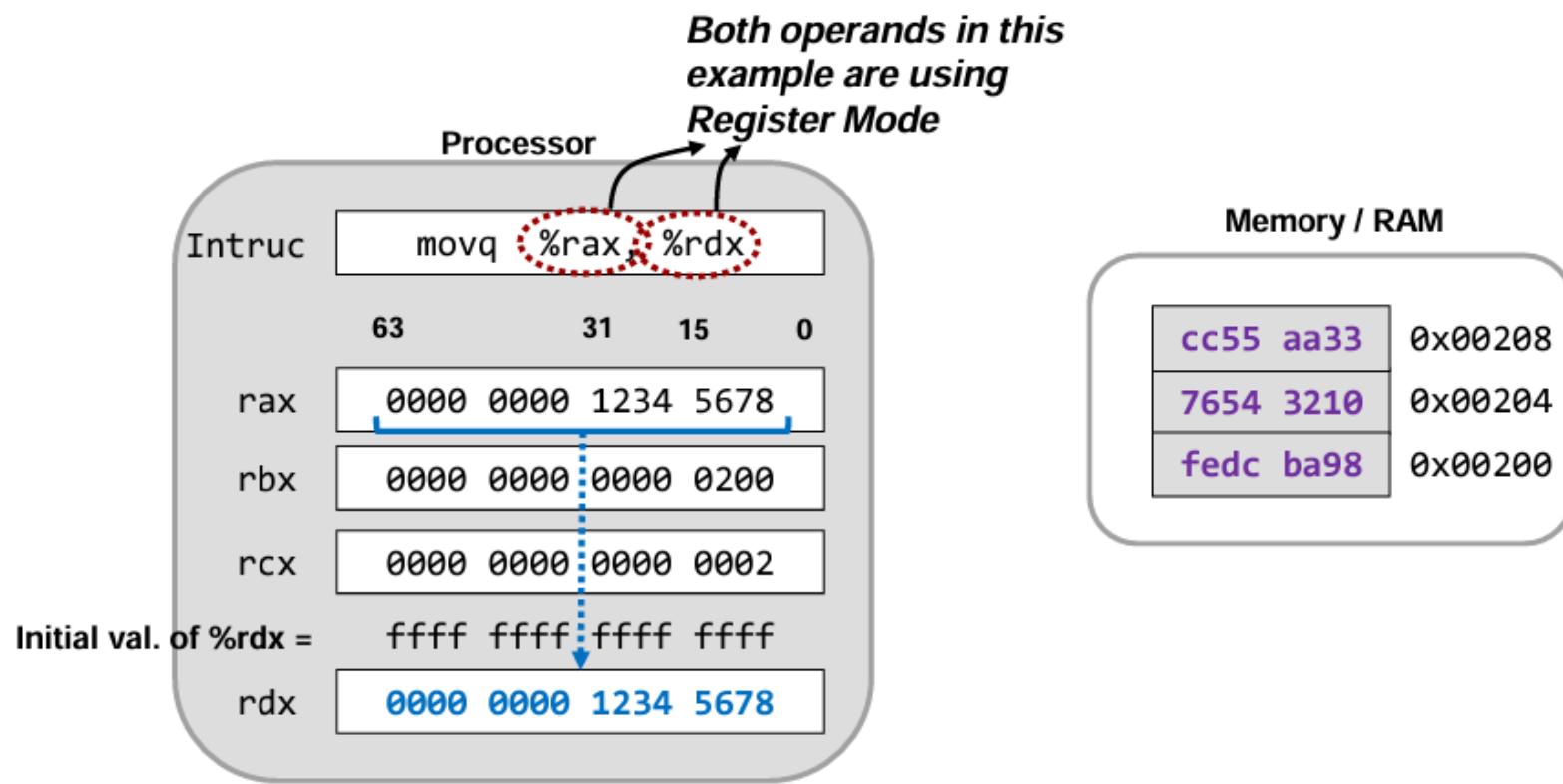
†Known as the scale factor and can be {1,2,4, or 8}

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

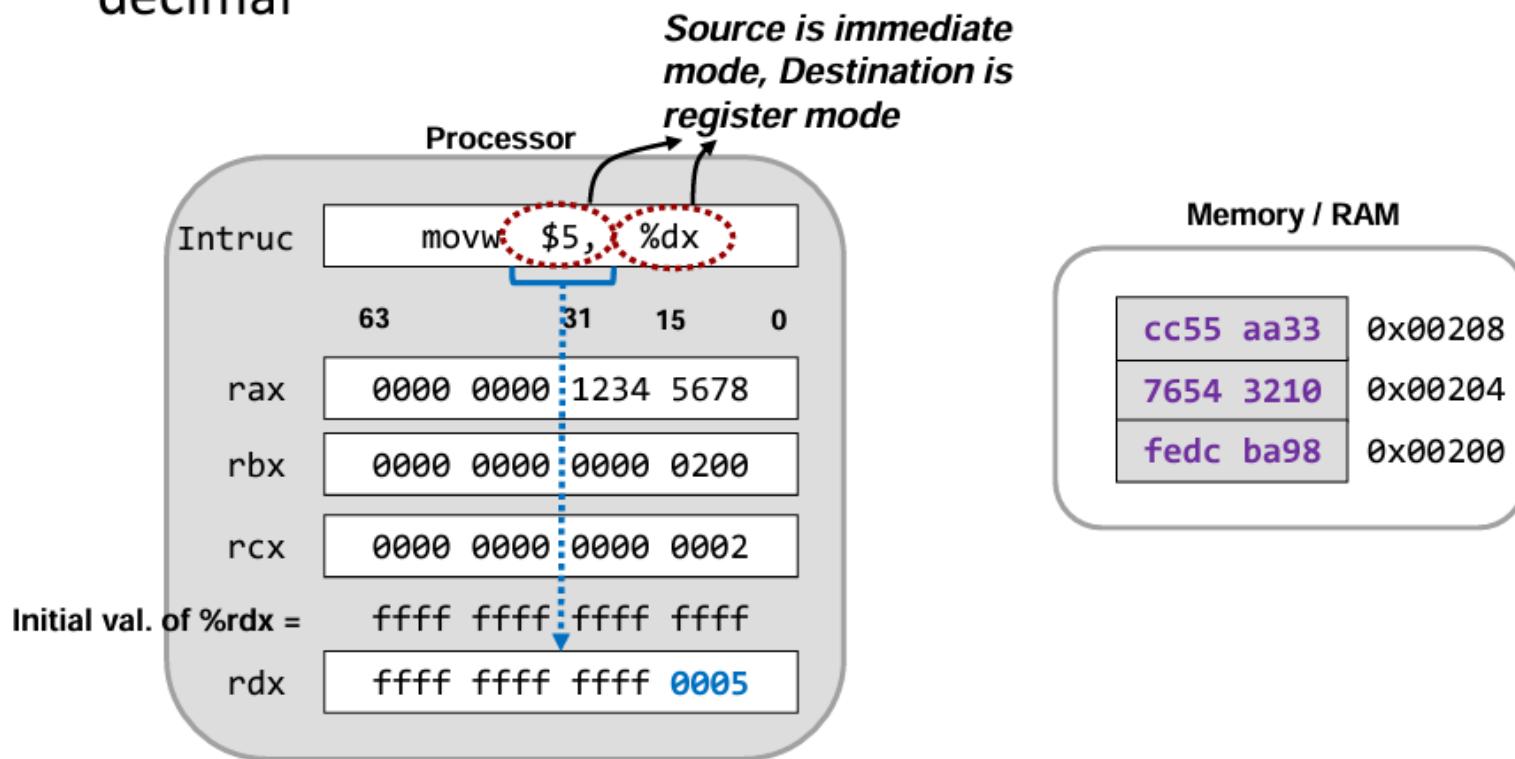
- Specifies the contents of a register as the operand



指令集的分类1 – 传输指令： Immediate Mode

- Specifies the a constant stored in the instruction as the operand

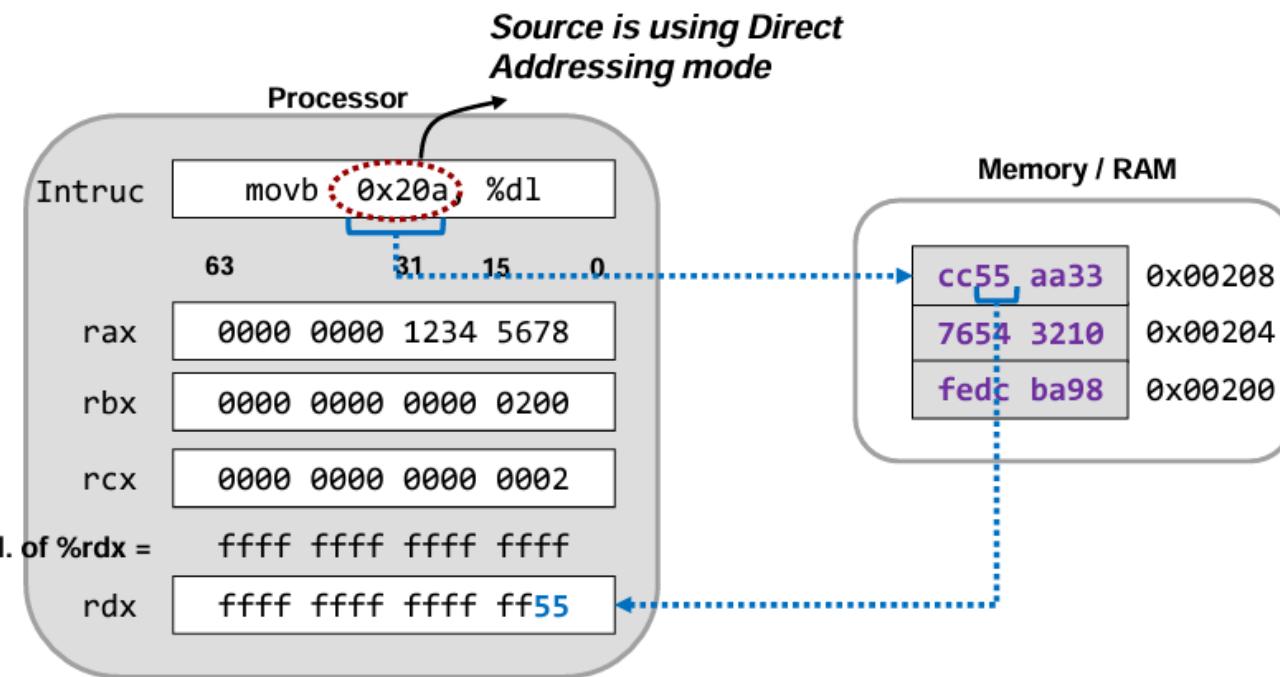
- Immediate is indicated with '\$' and can be specified in hex or decimal



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

- Specifies a constant memory address where the true operand is located

- Address can be specified in decimal or hex

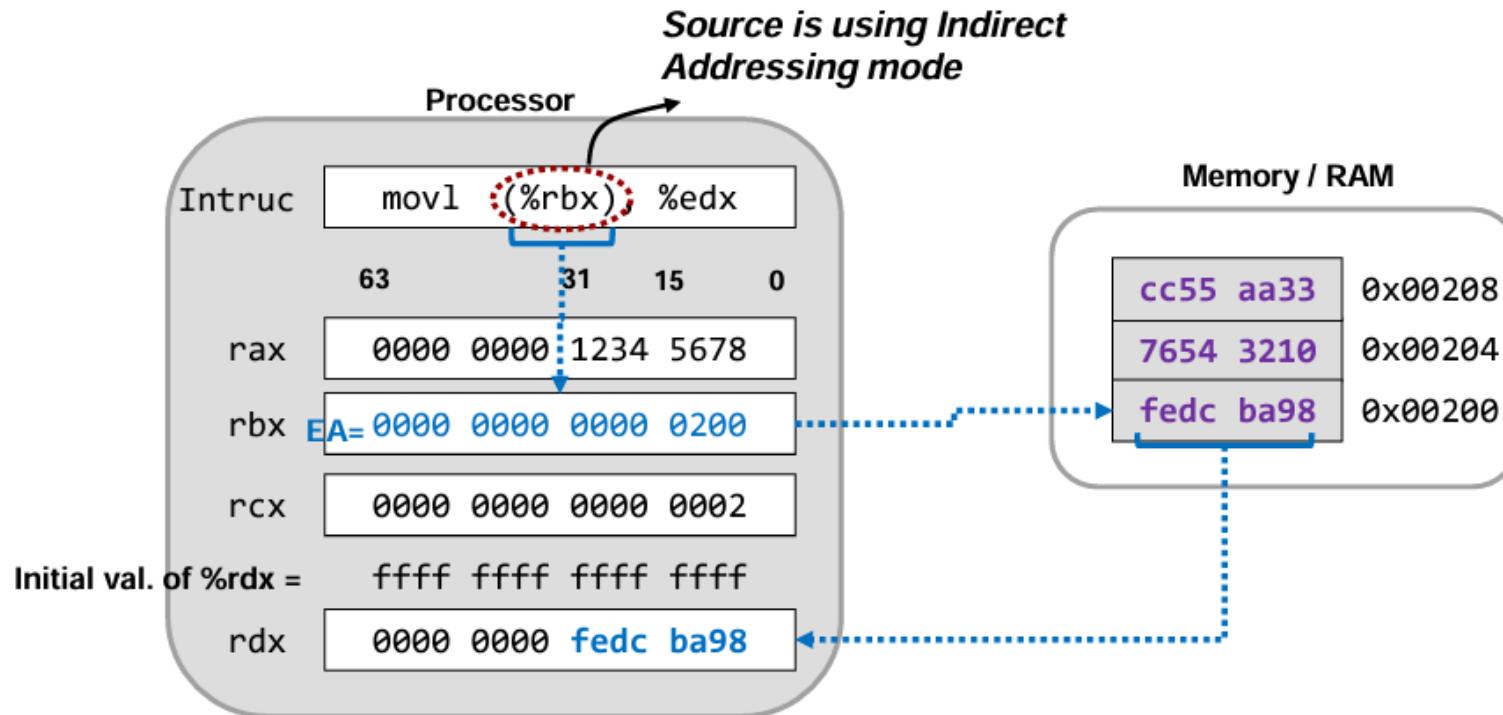


指令集的分类1 – 传输指令： Indirect Addressing Mode

- Specifies a register whose value will be used as the effective address in memory where the true operand is located

类似于指针

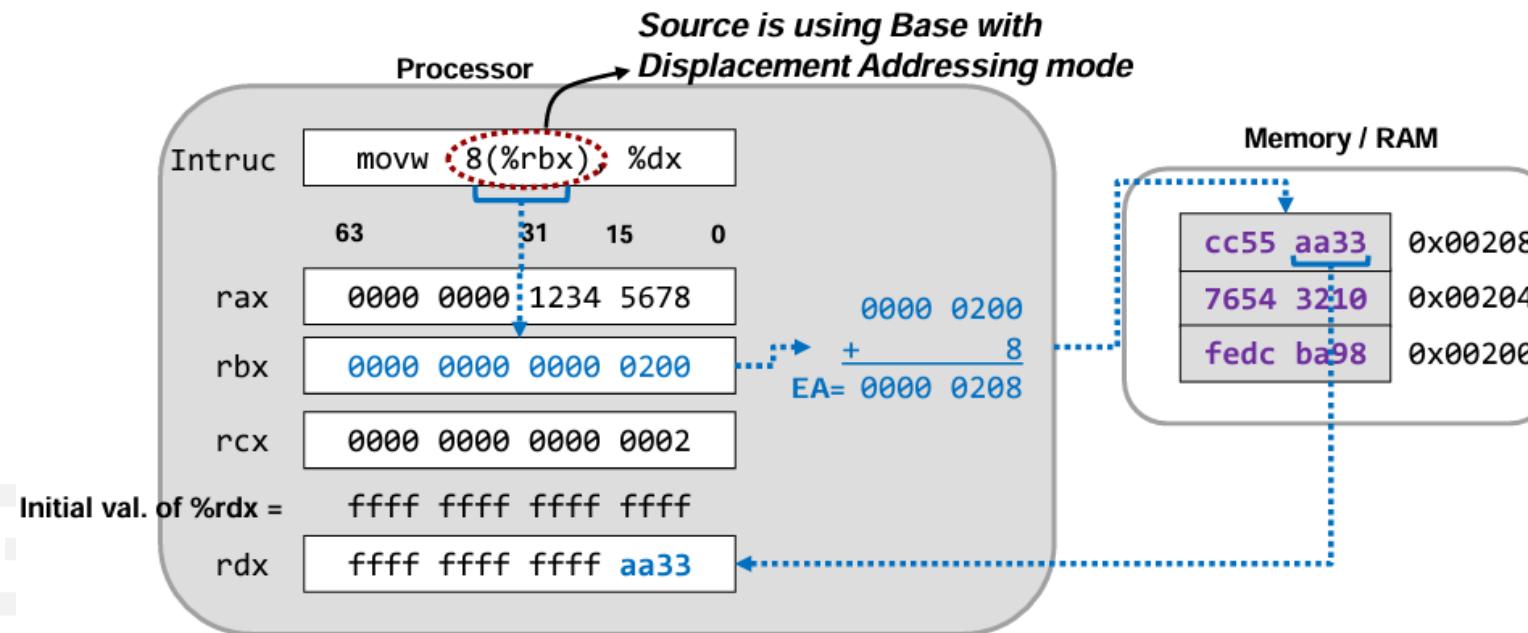
- Parentheses indicate indirect addressing mode



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

- Adds a constant displacement to the value in a register and uses the sum as the effective address of the actual operand in memory



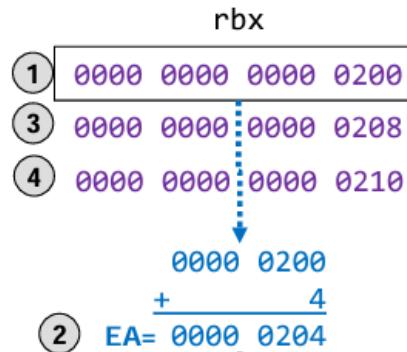
指令集的分类1 – 传输指令： Base/Indirect with Displacement Addressing Mode

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

- Useful for access members of a struct or object

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

C Code



Memory / RAM

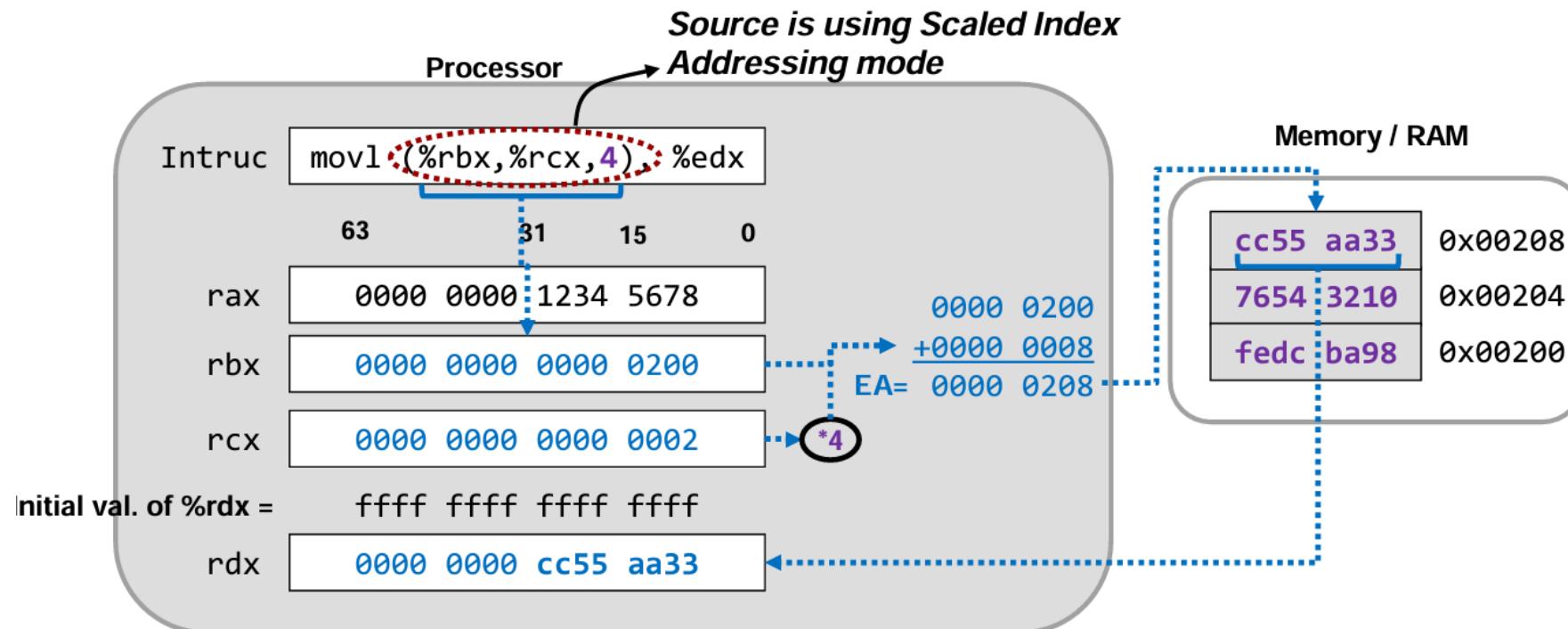
data[2].y	0000 0002	0x00214
data[2].x	0000 0001	0x00210
data[1].y	0000 0002	0x0020c
data[1].x	0000 0001	0x00208
data[0].y	0000 0002	0x00204
data[0].x	0000 0001	0x00200

① ③
② ④
Assembly

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

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

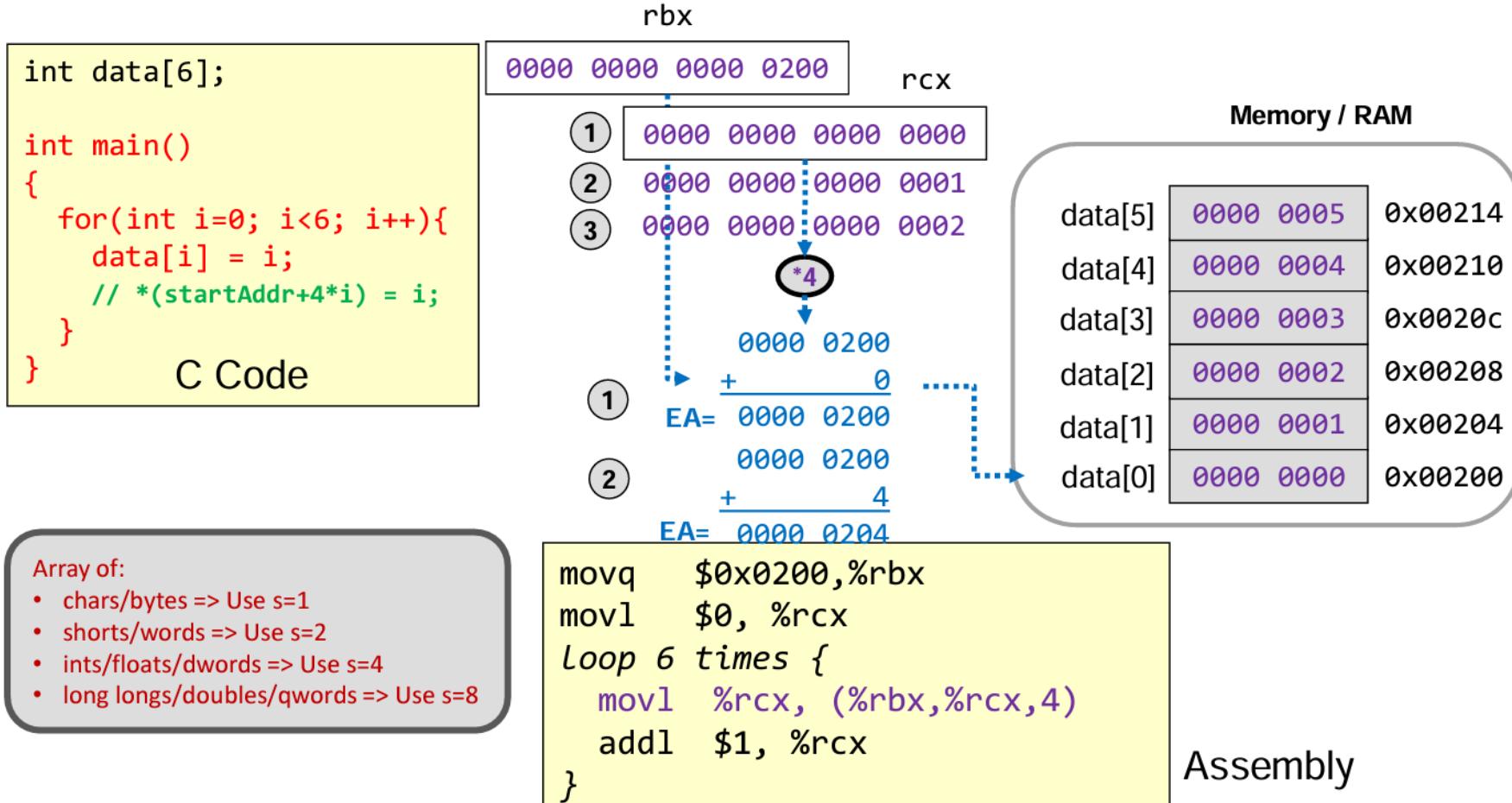
- 地址格式: Form: (%reg1,%reg2,s) [s = 1, 2, 4, or 8]
 - Uses the result of %reg1 + %reg2*s as the effective address of the actual operand in memory



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

- 为什么需要Scaled Index Addressing Mode实际案例

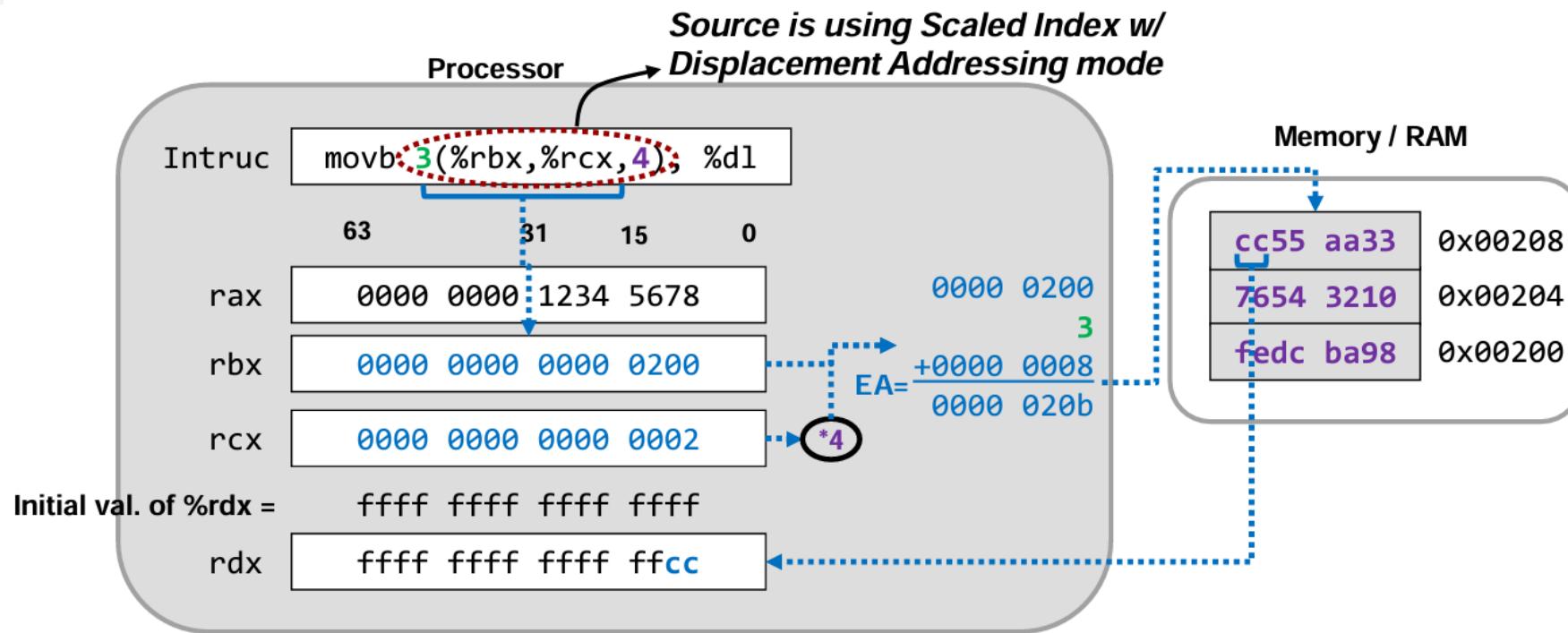
- Useful for accessing array elements



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

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

- Uses the result of $d + \%reg1 + \%reg2*s$ as the effective address of the actual operand in memory



指令集的分类1 – 传输指令：Addressing Mode案例

- 实际程序中可能由多种Addressing Mode共同组成

Processor Registers	Memory / RAM
0000 0000 0000 0200	rbx
0000 0000 0000 0003	rcx
	Memory / RAM
	cdef 89ab 0x00204
	7654 3210 0x00200
	f00d face 0x001fc
	dead beef 0x001f8
– movq (%rbx), %rax	cdef 89ab 7654 3210 rax
– movl -4(%rbx), %eax	0000 0000 f00d face rax
– movb (%rbx,%rcx), %al	0000 0000 f00d fa76 rax
– movw (%rbx,%rcx,2), %ax	0000 0000 f00d cdef rax
– movsb -16(%rbx,%rcx,4), %eax	0000 0000 ffff ffce rax
– movw %cx, 0xe0(%rbx,%rcx,2)	0000 0000 0x002e8 0003 0000 0x002e4

指令集的分类2 – 计算指令

- 利用ALU来完成实际计算任务

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

指令集的分类2 – 计算指令

- 利用ALU来完成实际计算任务

- Performs arithmetic/logic operation on the given size of data
- Restriction: Both operands cannot be memory
- Format
 - `add[b,w,l,q] [src2, src1/dst]` Work from right->left->right
 - Example 1: `addq %rbx, %rax` (`%rax += %rbx`)
 - Example 2: `subq %rbx, %rax` (`%rax -= %rbx`)

- Initial Conditions

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

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

Processor Registers

主讲：陶耀宇、李萌

指令集的分类2 – 计算指令：实际案例

- 计算指令配合传输指令完成一个代码的编译过程

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

Original Code

```
f1:
    movl (%esi), %eax
    addl (%edi,%edx,4), %eax
    ret
```

Compiler Output

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

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

Original Code

```
f1:
    addb $1, (%edi)
    subl %esi, 4(%edi)
    ret
```

Compiler Output

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

If(condition 0)

XXXXXX
XXXXXX
XXXXXX
XXXXXX

else

XXXXXX
XXXXXX

while(condition 1)

XXXXXX
XXXXXX
XXXXXX
XXXXXX

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

;Pretend there is a lot of code inbetween here.

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

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

北京大学-智能硬件体系结构

编译器设计内容

2024年秋季学期

主讲：陶耀宇、李萌

代表性指令集：RISC-V一种典型的RISC指令

- 完全开源，扩展性较好，指令种类多

31:25	24:20	19:15	14:12	11:7	6:0
funct7	rs2	rs1	funct3	rd	op
imm _{11:0}		rs1	funct3	rd	op
imm _{11:5}	rs2	rs1	funct3	imm _{4:0}	op
imm _{12,10:5}	rs2	rs1	funct3	imm _{4:1,11}	op
imm _{31:12}				rd	op
imm _{20,10:1,11,19:12}				rd	op
fs3	funct2	fs2	fs1	funct3	fd
5 bits	2 bits	5 bits	5 bits	3 bits	7 bits

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

- imm:
- uimm:
- upimm:
- Address:
- [Address]:
- BTA:
- JTA:
- label:
- SignExt:
- ZeroExt:
- csr:

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

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

主讲：陆雅宇、李萌

代表性指令集：MIPS一种典型的RISC指令

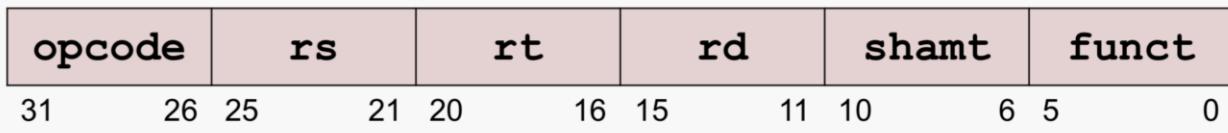
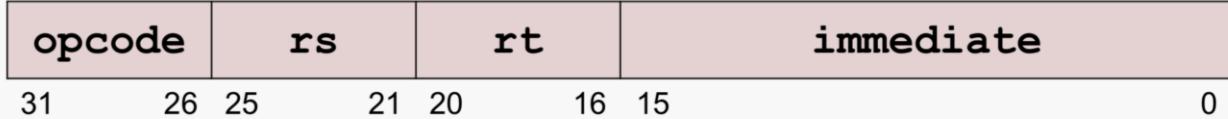
- 指令长度固定，相对简单（单周期指令）

• 3 types of CPU instructions each of which are 32-bit aligned 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

R	
I	
J	

目录

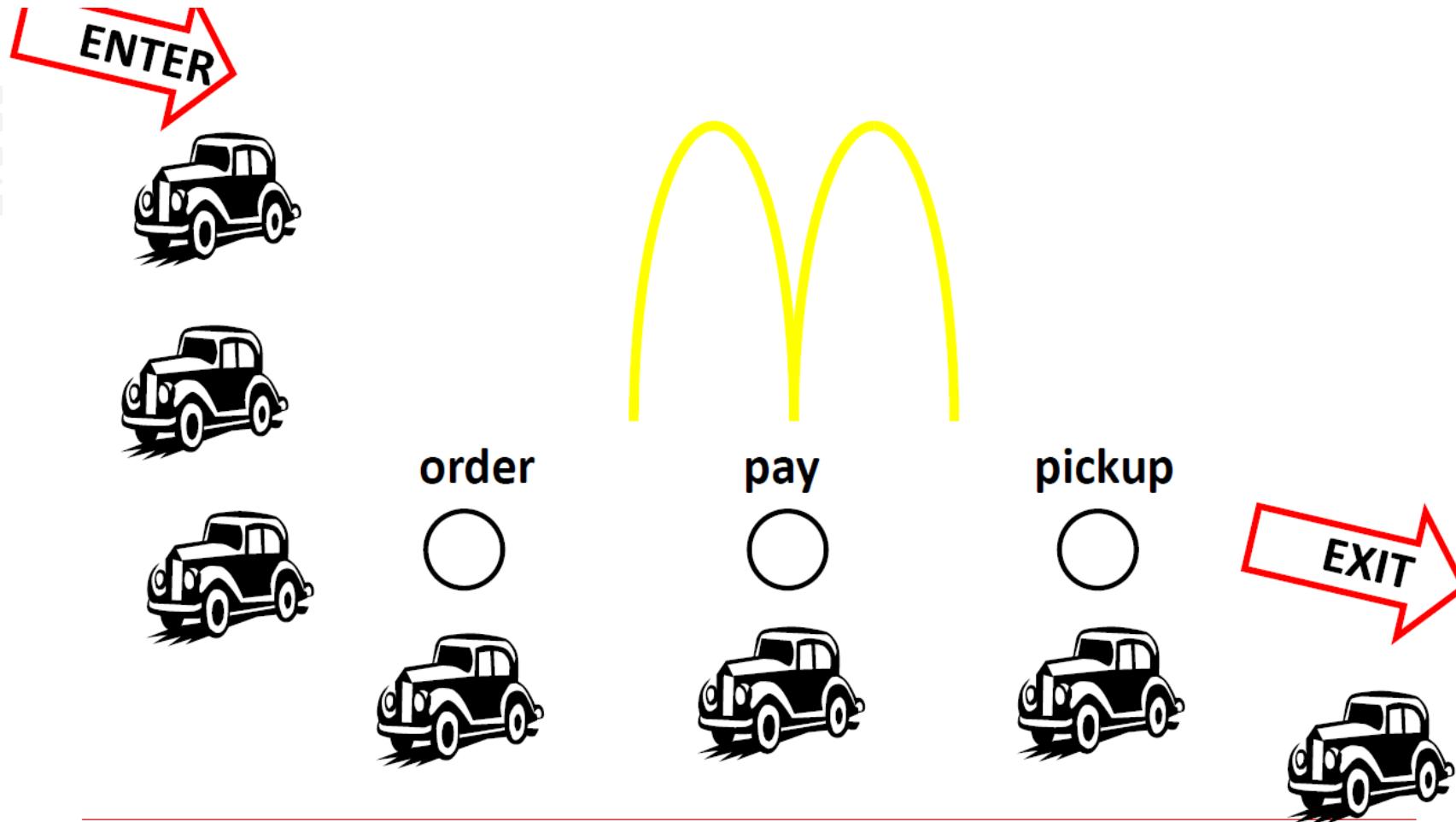
CONTENTS



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

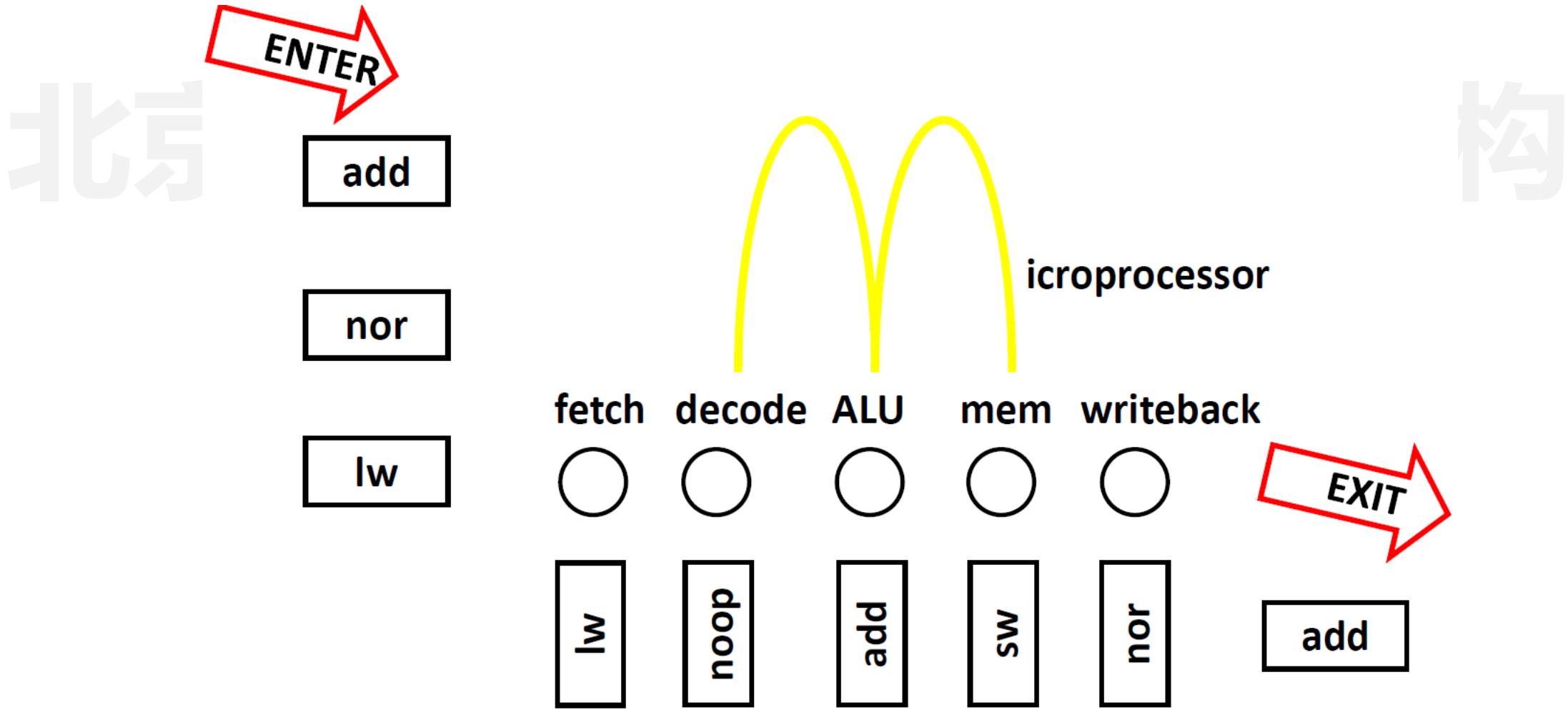
回顾：什么是流水线架构

- 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)



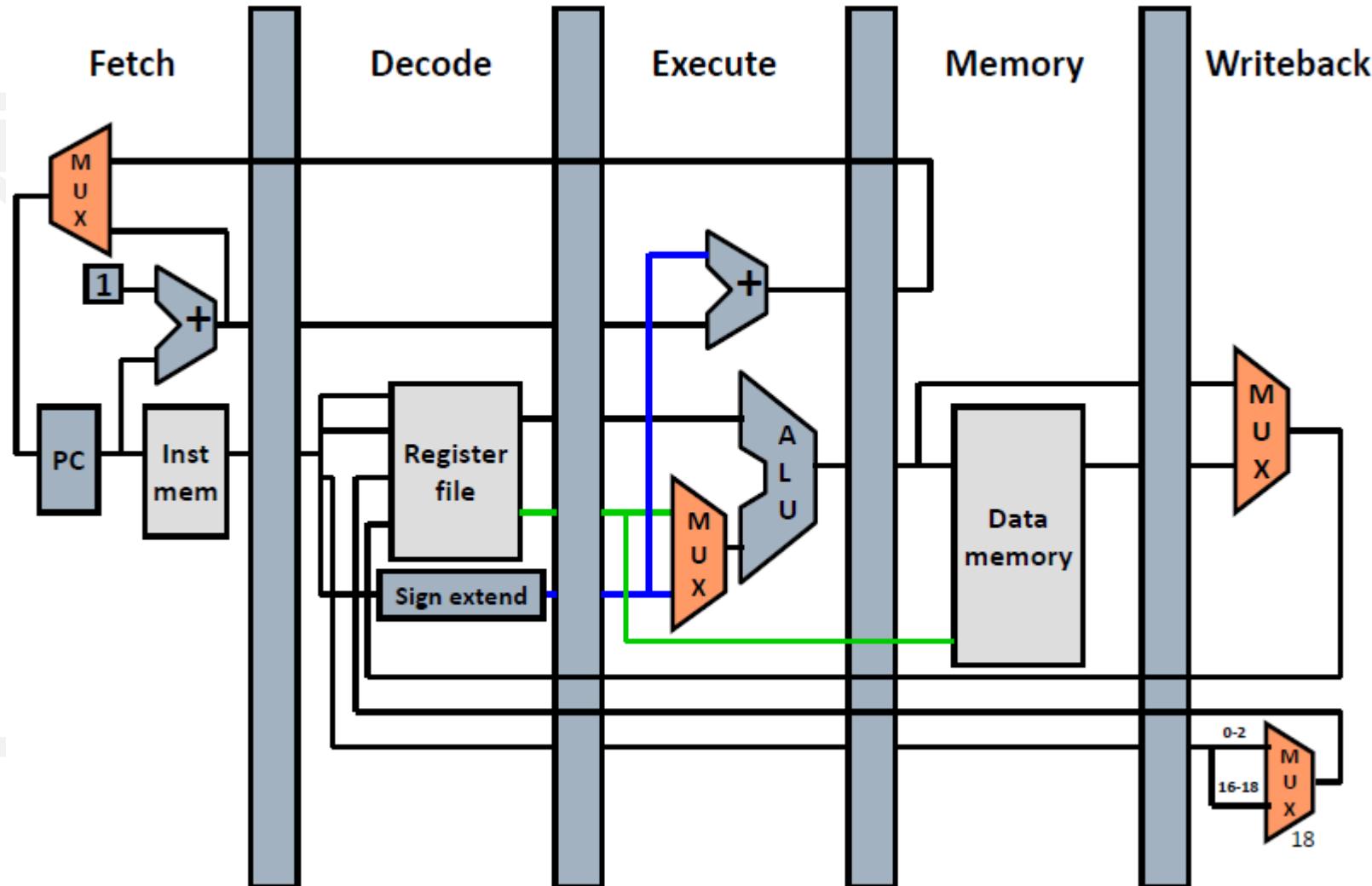
计算机CPU的流水线架构演进

• 多条、深度流水线设计

- Execute as many instructions at the same time as possible.
 - Pipelining: 12-20+ cycles
 - Multiple pipelines
- Pentium:
 - 2 pipelines, 5 cycles each (10 instructions “in flight”)
- Pentium Pro/II/III
 - 3 pipelines (kinda), 12 cycles each (kinda)
 - Instructions can execute out of their original program order
- Pentium IV
 - 4 pipelines, 20 cycles deep
 - Prescott: 4 pipelines, 31 cycles deep (could be clocked up to 8 GHz with special cooling)
- Core i7 (Nehalem)
 - 4 pipelines, 16 cycles deep

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

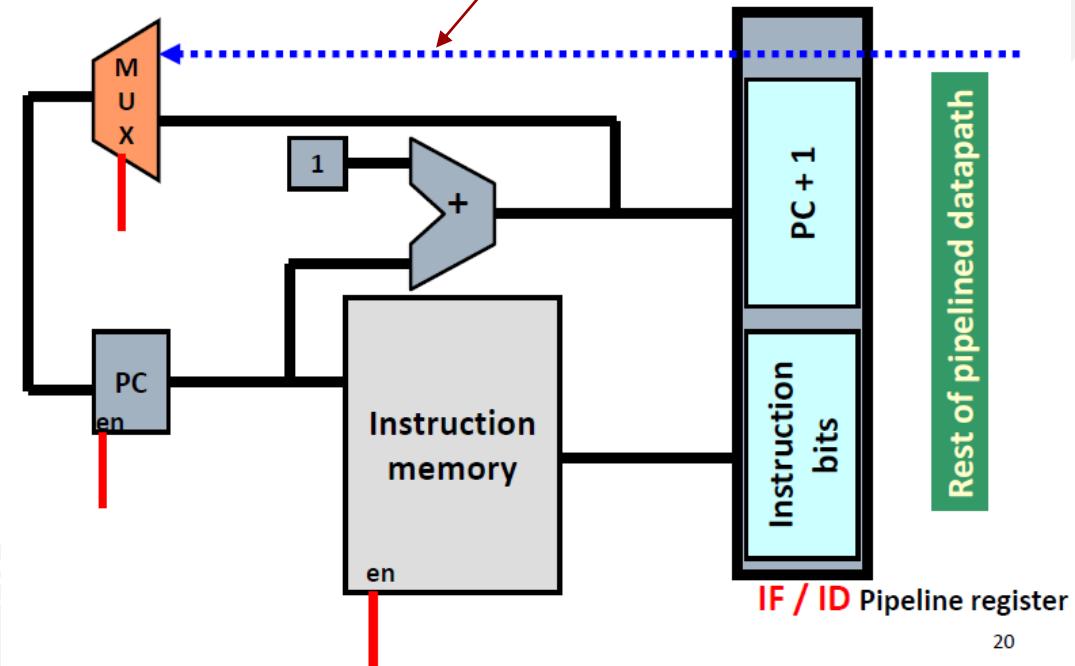


第一级：Fetch指令

- PC控制从指令Memory里读取的地址

- Design a datapath that can fetch an instruction from memory every cycle.
 - Use PC to index memory to read instruction
 - Increment the PC (assume no branches for now)
- Write everything needed to complete execution to the pipeline register (IF/ID)
 - The next stage will read this pipeline register.
 - Note that pipeline register must be edge-triggered

用于更新PC应对控制指令 (jmp等)

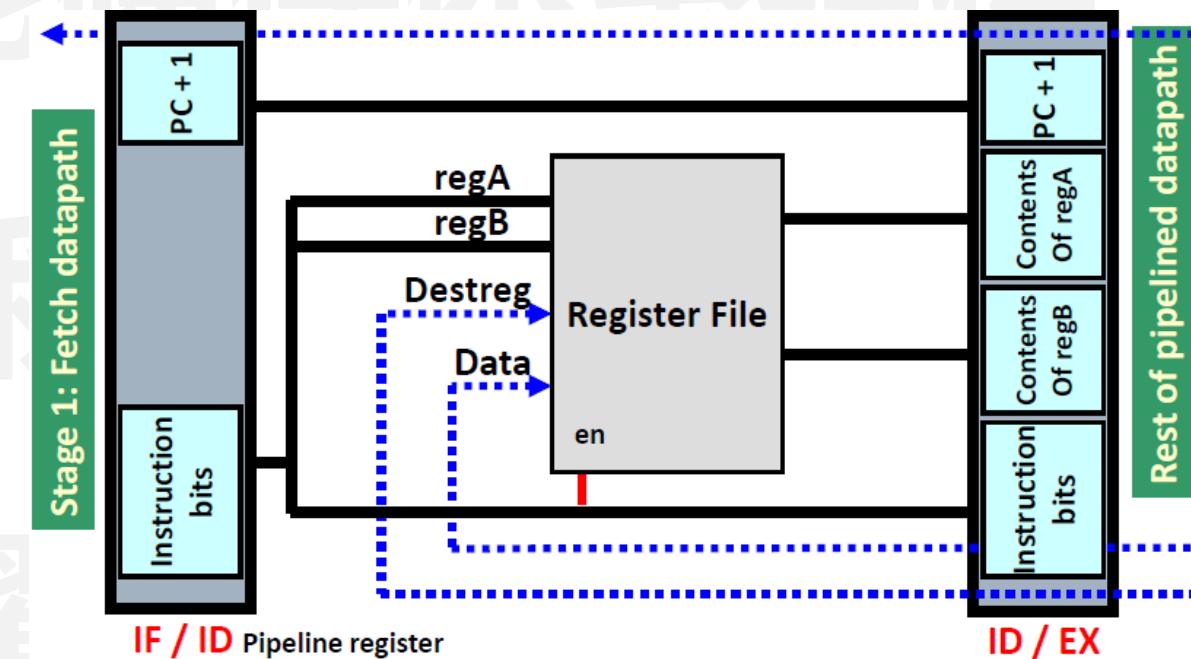


第二级：Decode指令

- 根据指令的register从register集群中读取运算所需的值

假设最简单的指令格式：opcode regA/Data regB/DestReg

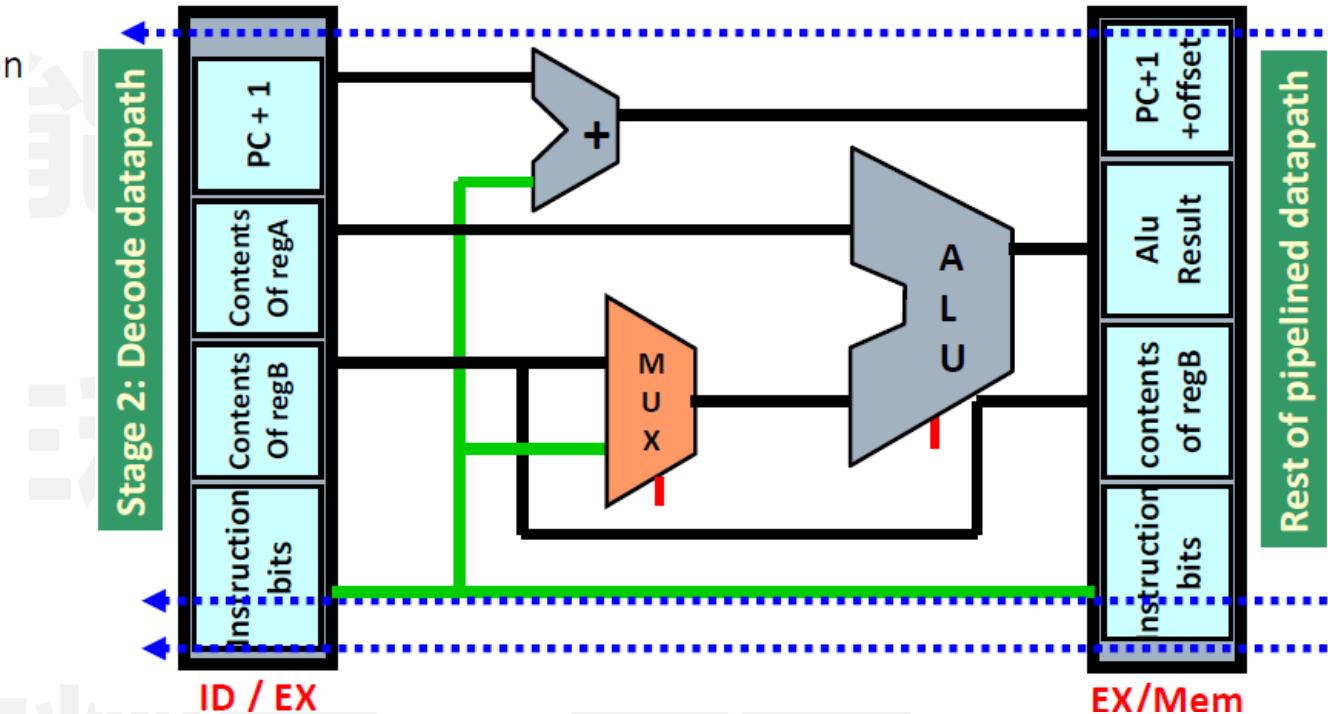
- Design a datapath that reads the IF/ID pipeline register, decodes instruction and reads register file (specified by regA and regB of instruction bits).
 - Decode is easy, just pass on the opcode and let later stages figure out their own control signals for the instruction.
- Write everything needed to complete execution to the pipeline register (ID/EX)
 - Pass on the offset field and both destination register specifiers (or simply pass on the whole instruction!).
 - Including PC+1 even though decode didn't use it.



第三级：Execute指令

- 利用ALU和加法器计算运算结果并更新下一个指令的PC值

- Design a datapath that performs the proper ALU operation for the instruction specified and the values present in the ID/EX pipeline register.
 - The inputs are the contents of regA and either the contents of regB or the offset field on the instruction.
 - Also, calculate PC+1+offset in case this is a branch.
- Write everything needed to complete execution to the pipeline register (EX/Mem)
 - ALU result, contents of regB and PC+1+offset
 - Instruction bits for opcode and destReg specifiers
 - Result from comparison of regA and regB contents

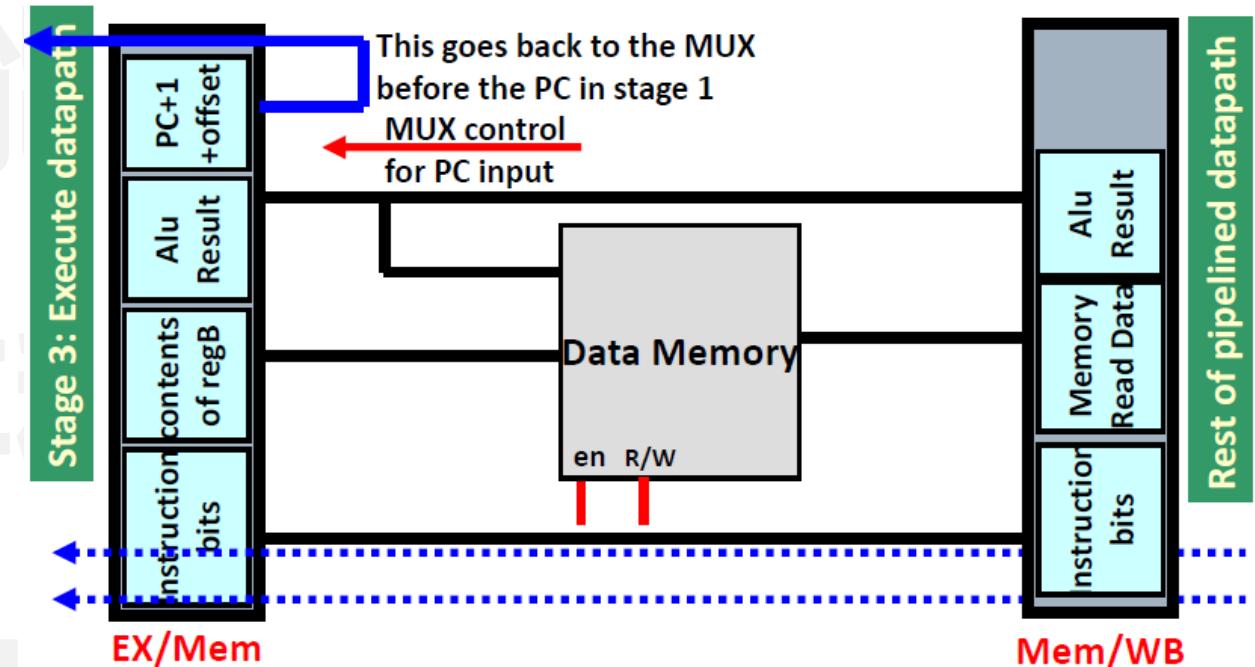


主讲：陶耀宇、李萌

第四级：Memory操作

- 将ALU结果或register读取结果存入Memory，或从Memory读出

- Design a datapath that performs the proper memory operation for the instruction specified and the values present in the EX/Mem pipeline register.
 - ALU result contains address for **Id** and **st** instructions.
 - Opcode bits control memory R/W and enable signals.
- Write everything needed to complete execution to the **pipeline register (Mem/WB)**
 - ALU result and MemData
 - Instruction bits for opcode and destReg specifiers

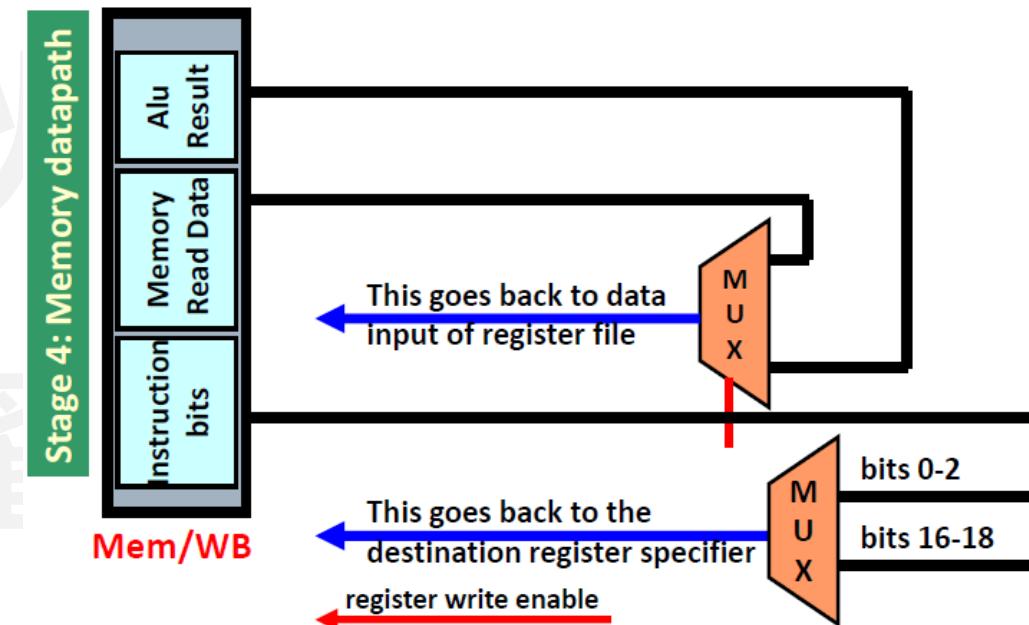


第五级：Writeback操作

- ALU计算结果或Data Memory读取的结果写回destReg

- Design a datapath that completes the execution of this instruction, writing to the register file if required.
 - Write MemData to destReg for Id instruction
 - Write ALU result to destReg for add or nand instructions.
 - Opcode bits also control register write enable signal.

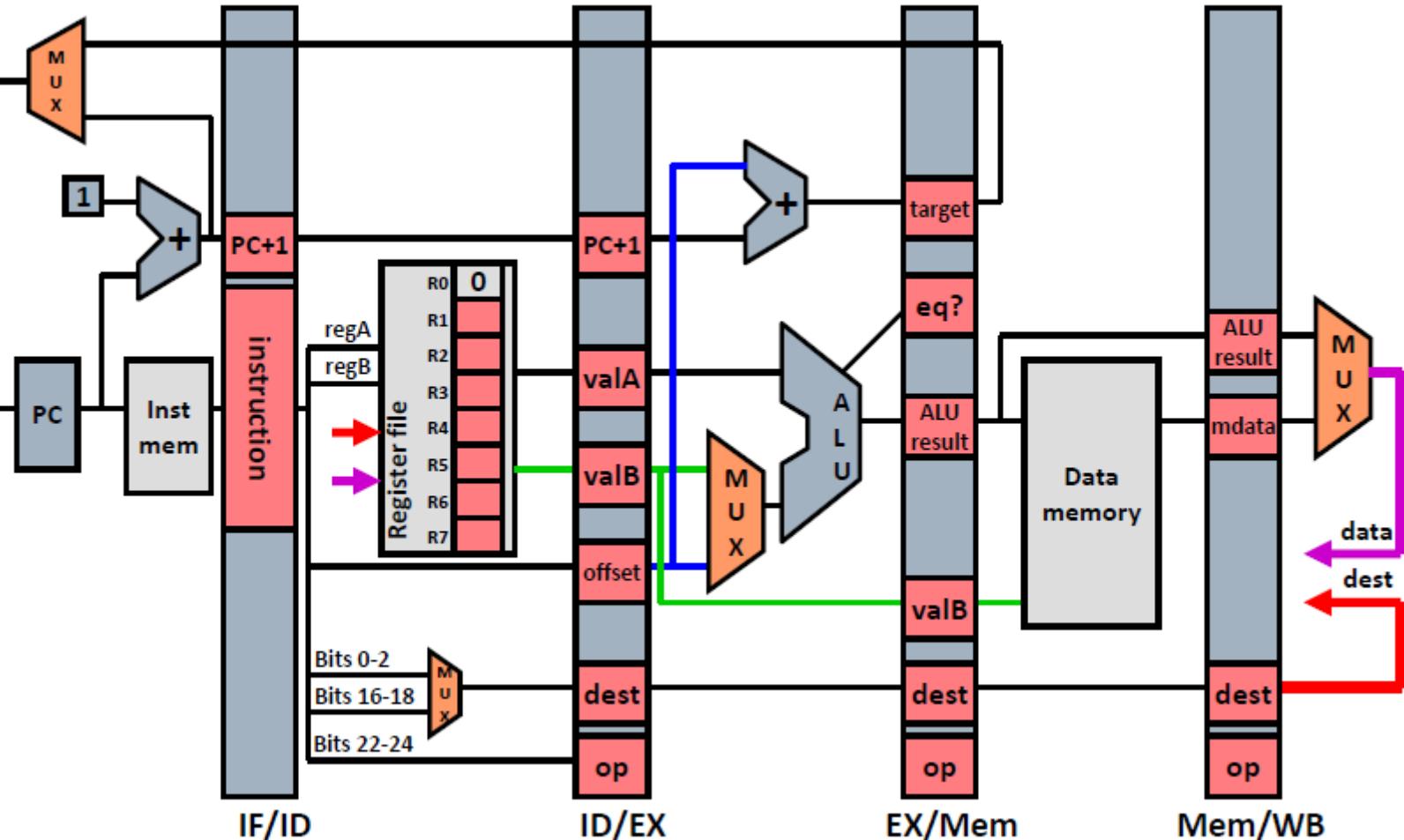
部件体系结构



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

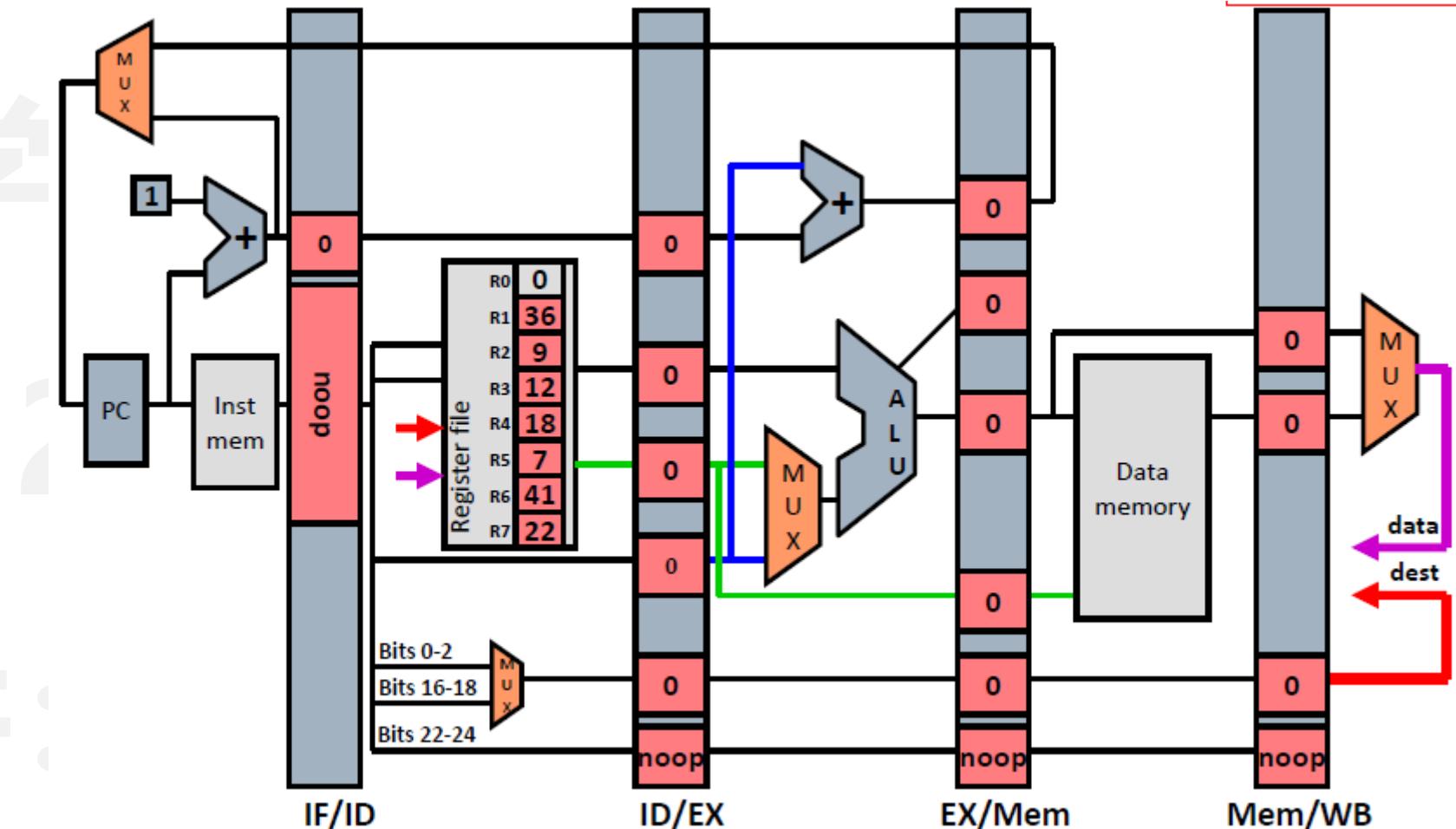
• add 1 2 3 ; reg 3 = reg 1 + reg 2
 • nor 4 5 6 ; reg 6 = reg 4 nor reg 5
 • lw 2 4 20 ; reg 4 = Mem[reg2+20]
 • add 2 5 5 ; reg 5 = reg 2 + reg 5
 • sw 3 7 10 ; Mem[reg3+10] = reg 7



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

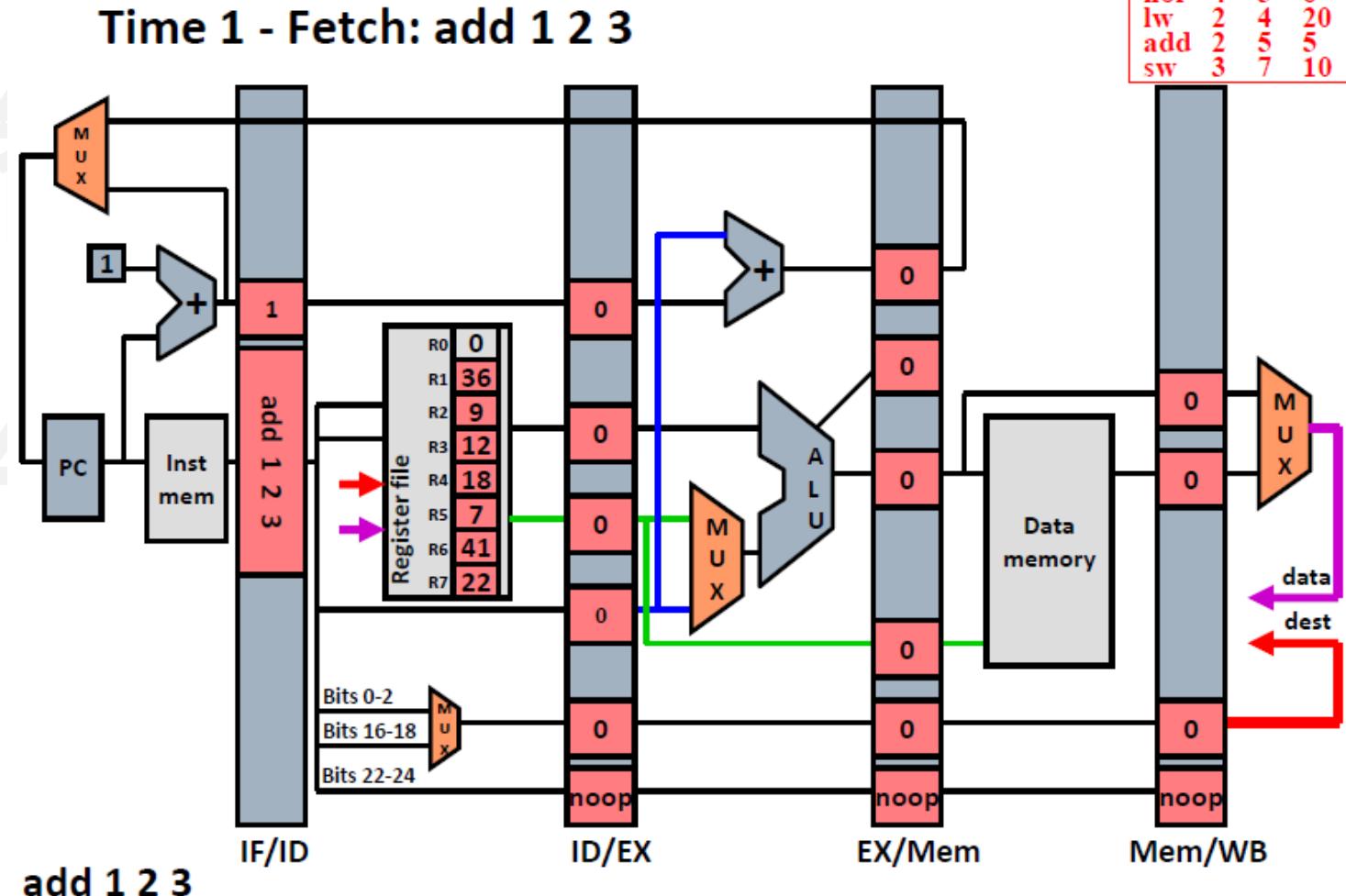
• add 1 2 3 ; reg 3 = reg 1 + reg 2
 • nor 4 5 6 ; reg 6 = reg 4 nor reg 5
 • lw 2 4 20 ; reg 4 = Mem[reg2+20]
 • add 2 5 5 ; reg 5 = reg 2 + reg 5
 • sw 3 7 10 ; Mem[reg3+10] = reg 7



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

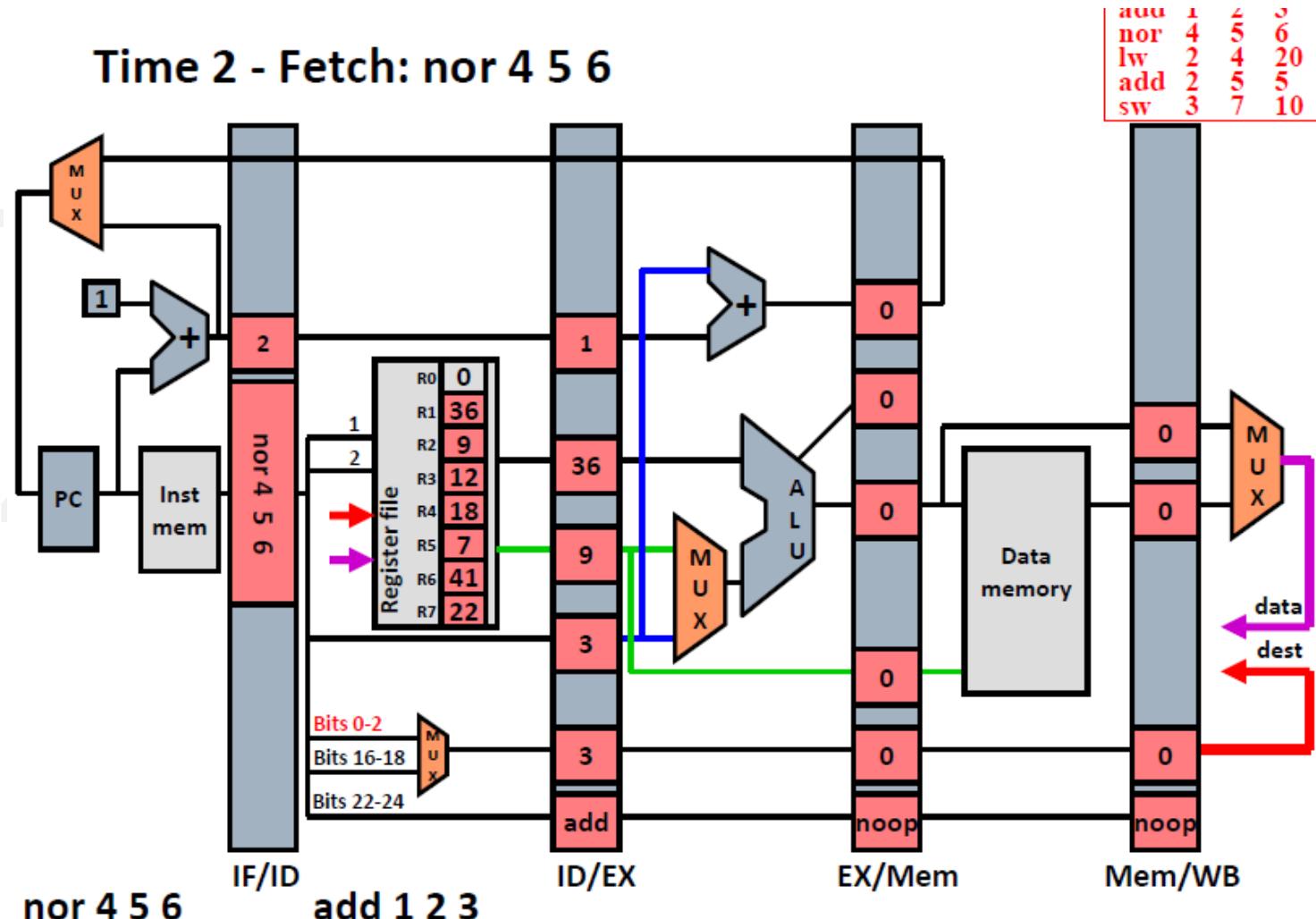
• add 1 2 3 ; reg 3 = reg 1 + reg 2
 • nor 4 5 6 ; reg 6 = reg 4 nor reg 5
 • lw 2 4 20 ; reg 4 = Mem[reg2+20]
 • add 2 5 5 ; reg 5 = reg 2 + reg 5
 • sw 3 7 10 ; Mem[reg3+10] = reg 7



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

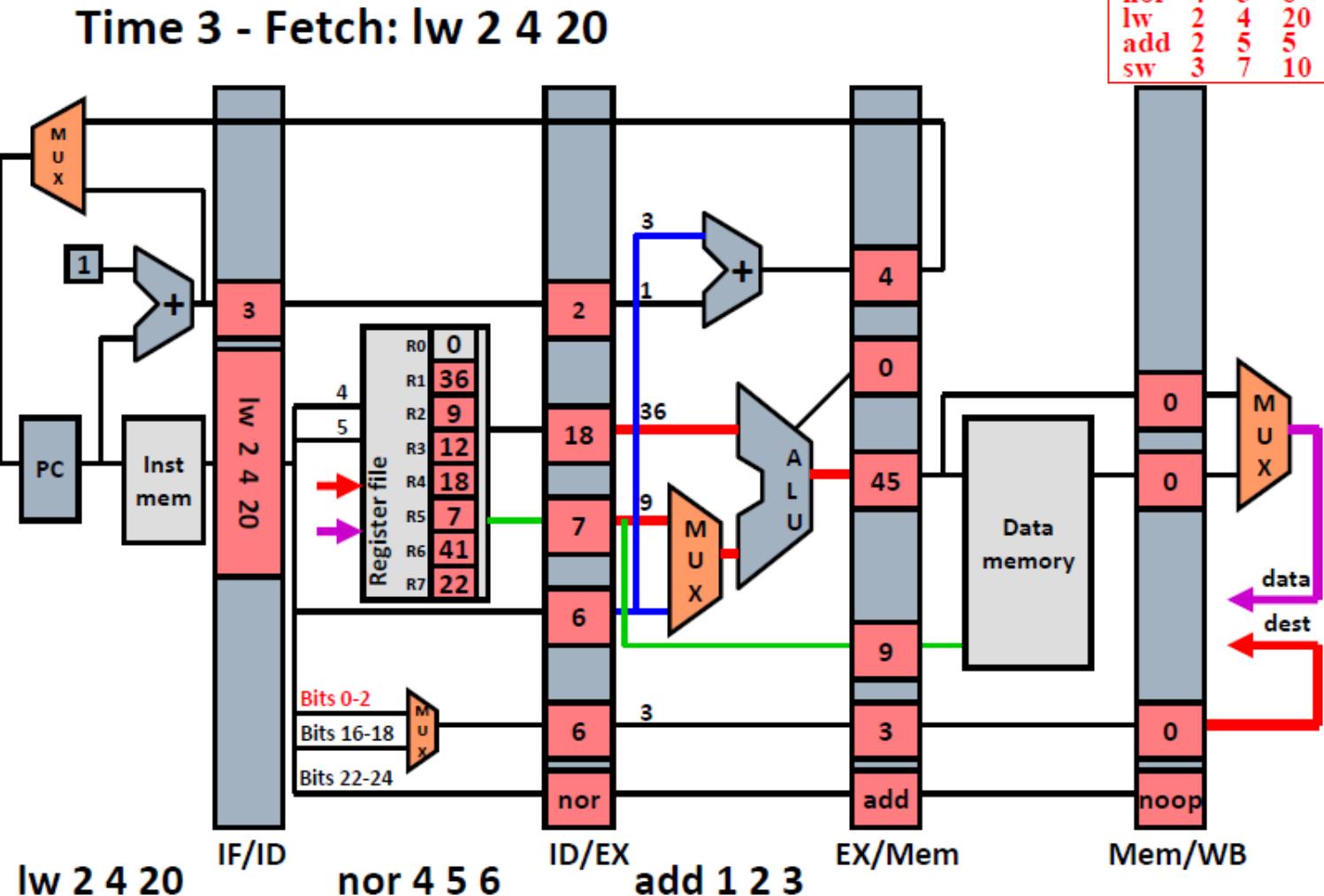
• add 1 2 3 ; reg 3 = reg 1 + reg 2
 • nor 4 5 6 ; reg 6 = reg 4 nor reg 5
 • lw 2 4 20 ; reg 4 = Mem[reg2+20]
 • add 2 5 5 ; reg 5 = reg 2 + reg 5
 • sw 3 7 10 ; Mem[reg3+10] = reg 7



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

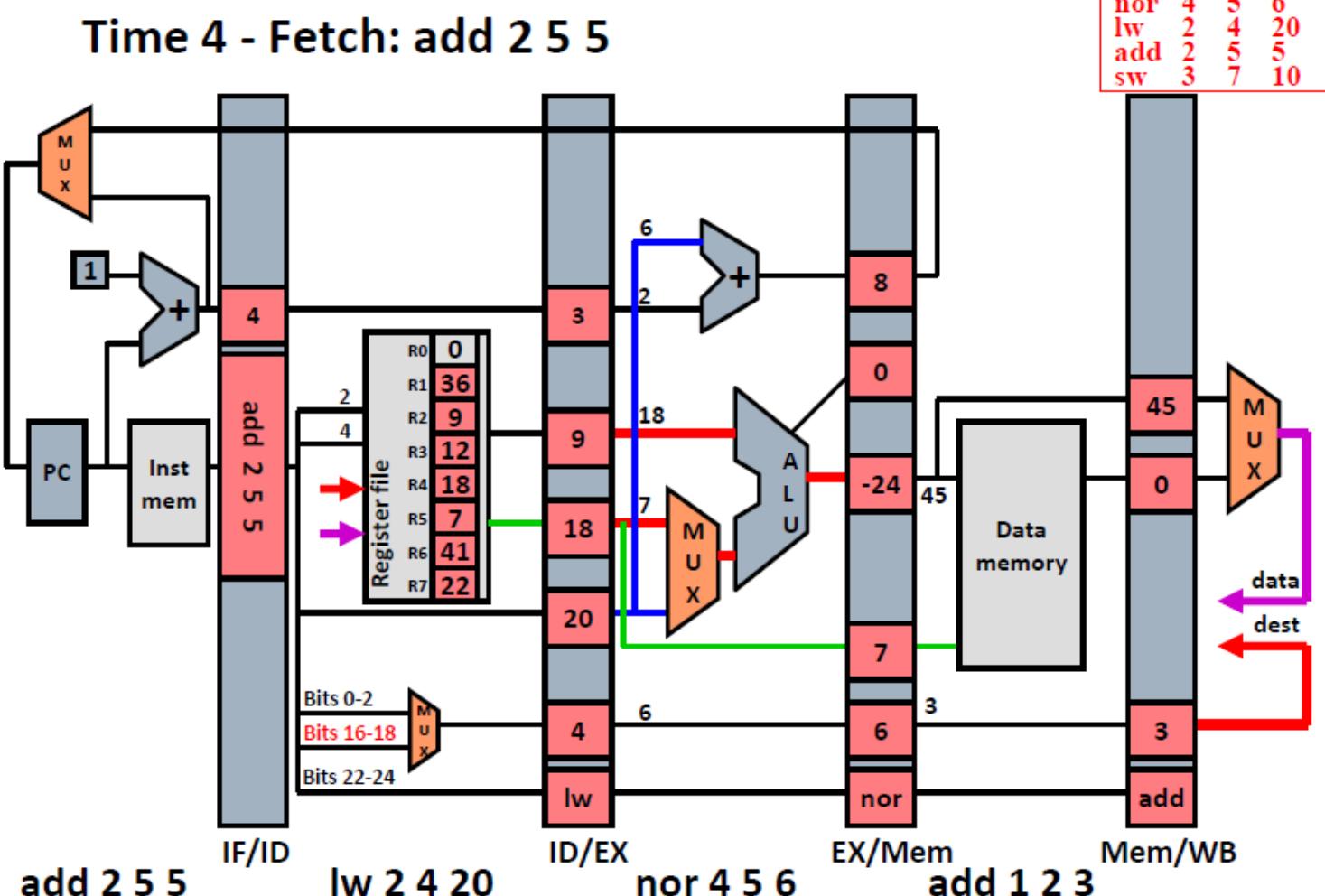
• add 1 2 3 ; reg 3 = reg 1 + reg 2
 • nor 4 5 6 ; reg 6 = reg 4 nor reg 5
 • lw 2 4 20 ; reg 4 = Mem[reg2+20]
 • add 2 5 5 ; reg 5 = reg 2 + reg 5
 • sw 3 7 10 ; Mem[reg3+10] = reg 7



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

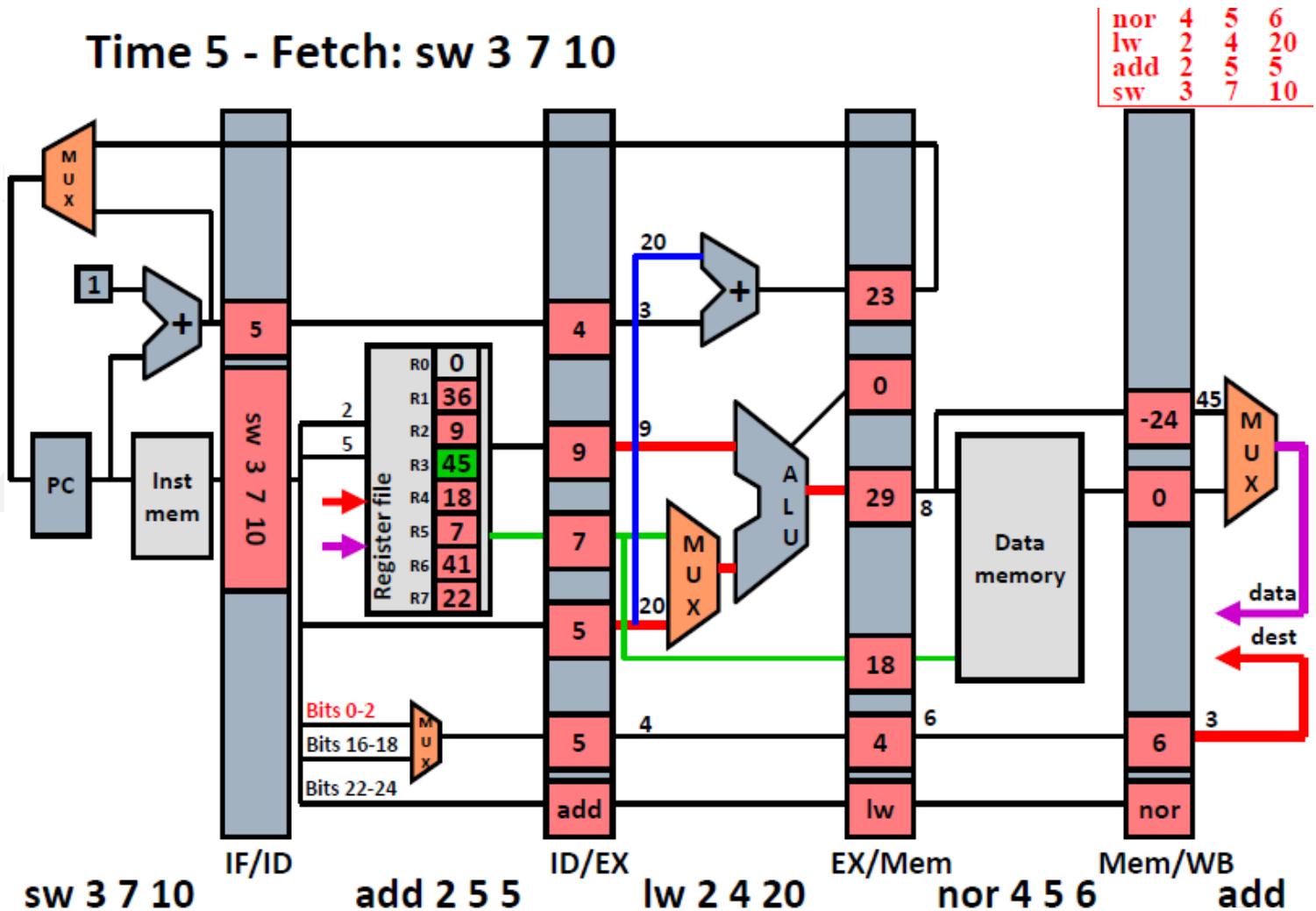
• add 1 2 3 ; reg 3 = reg 1 + reg 2
 • nor 4 5 6 ; reg 6 = reg 4 nor reg 5
 • lw 2 4 20 ; reg 4 = Mem[reg2+20]
 • add 2 5 5 ; reg 5 = reg 2 + reg 5
 • sw 3 7 10 ; Mem[reg3+10] = reg 7



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

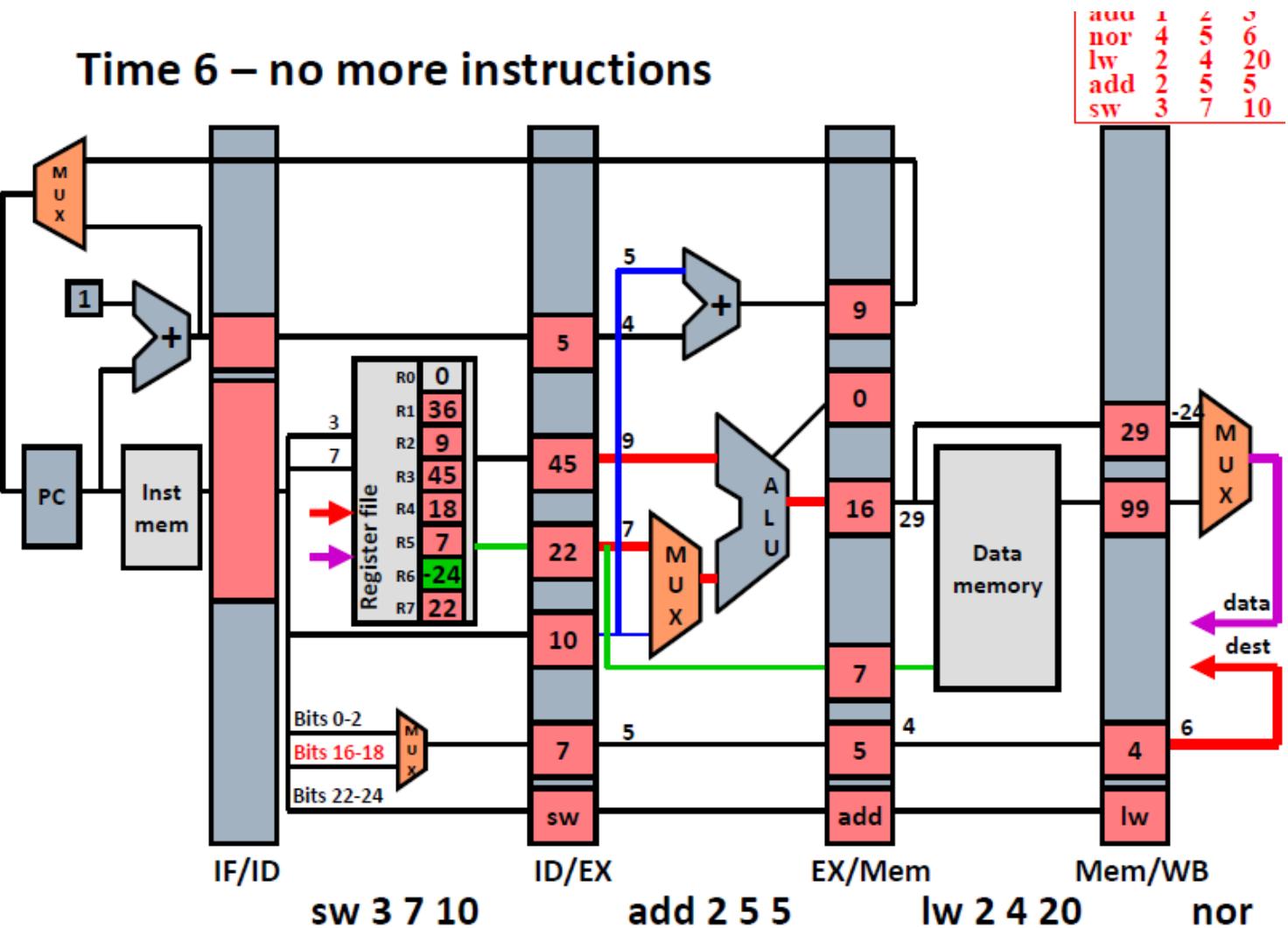


5级流水线的实际案例

- 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

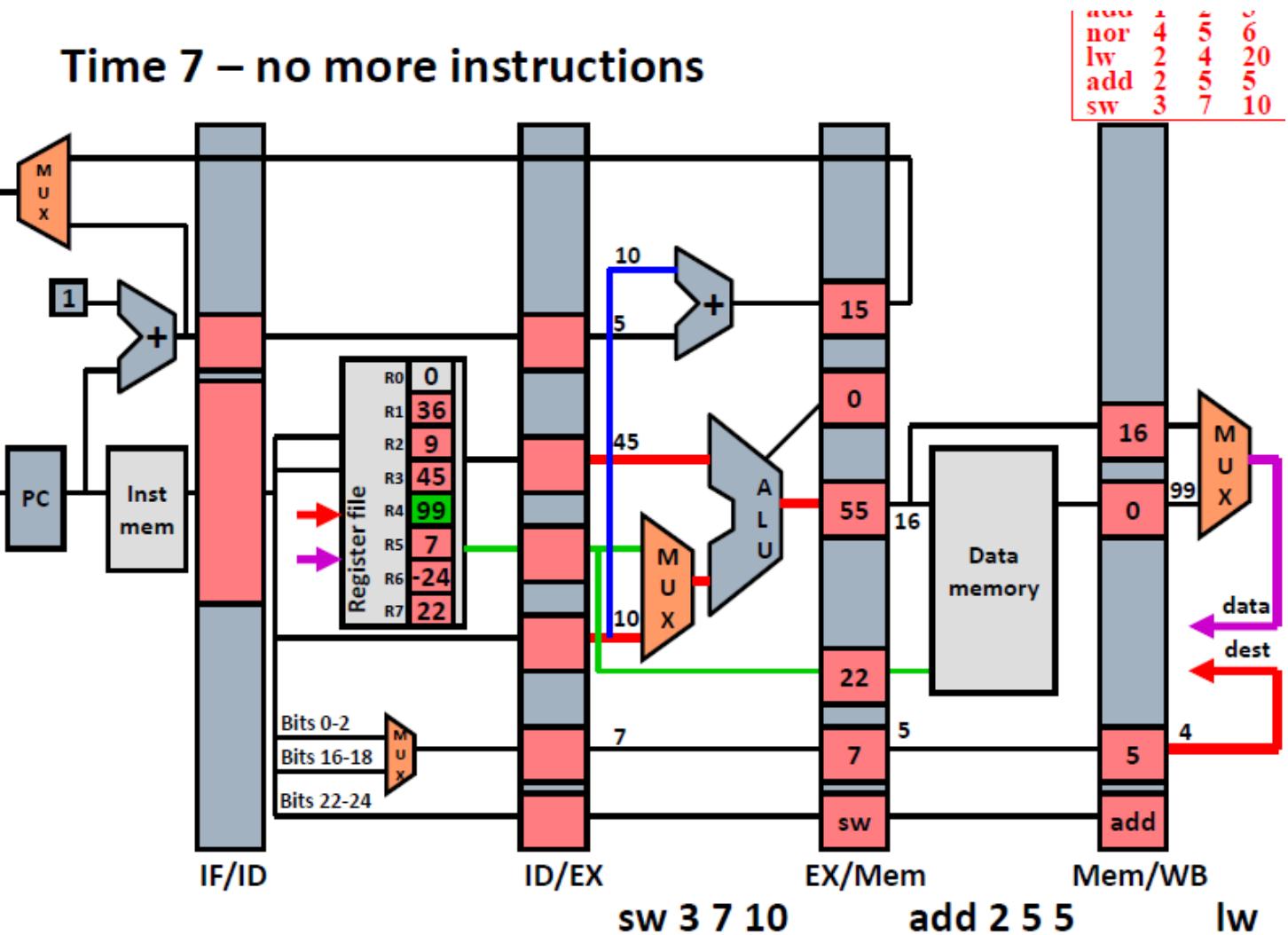
Time 6 – no more instructions



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

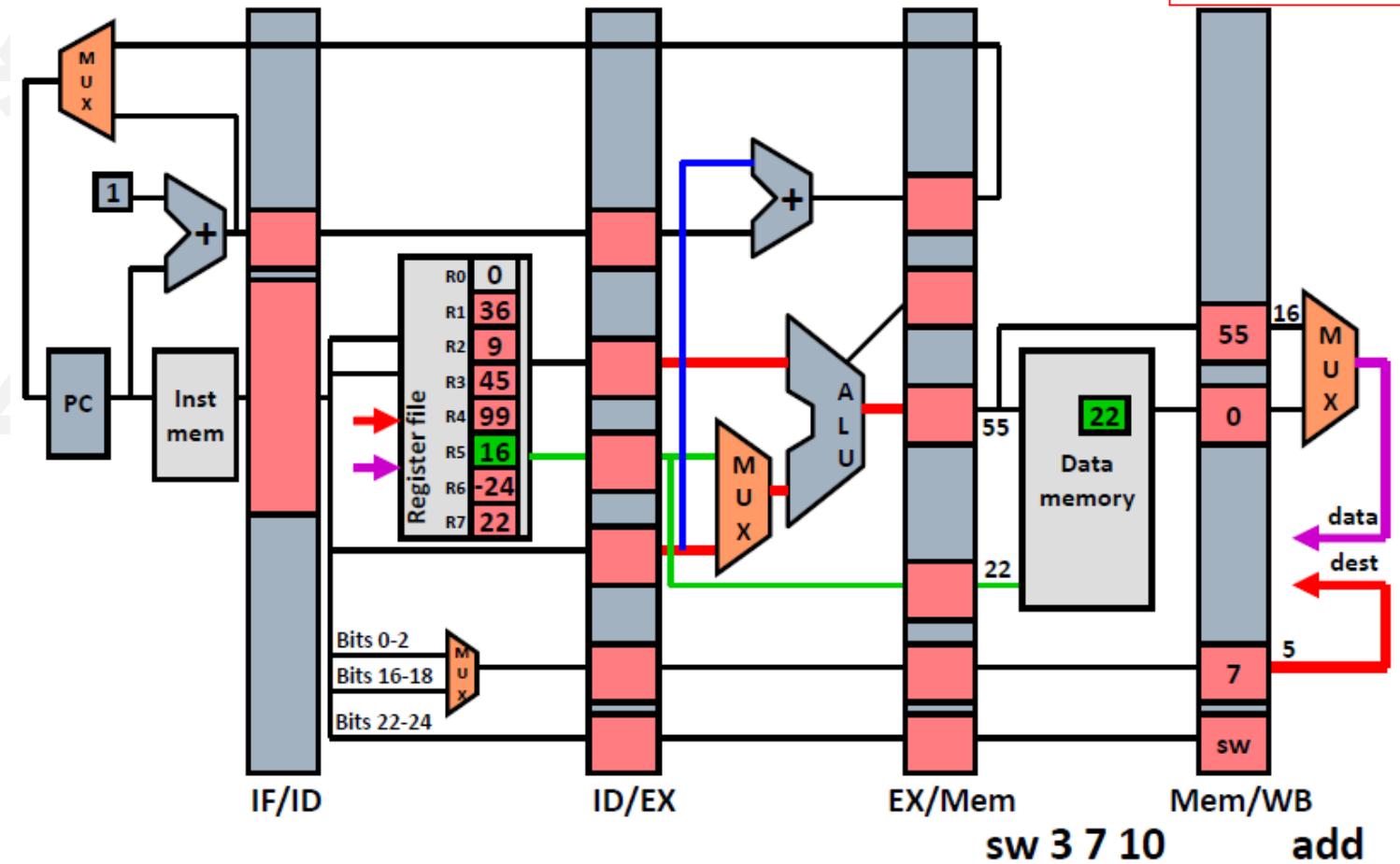


5级流水线的实际案例

- 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

Time 8 – no more instructions



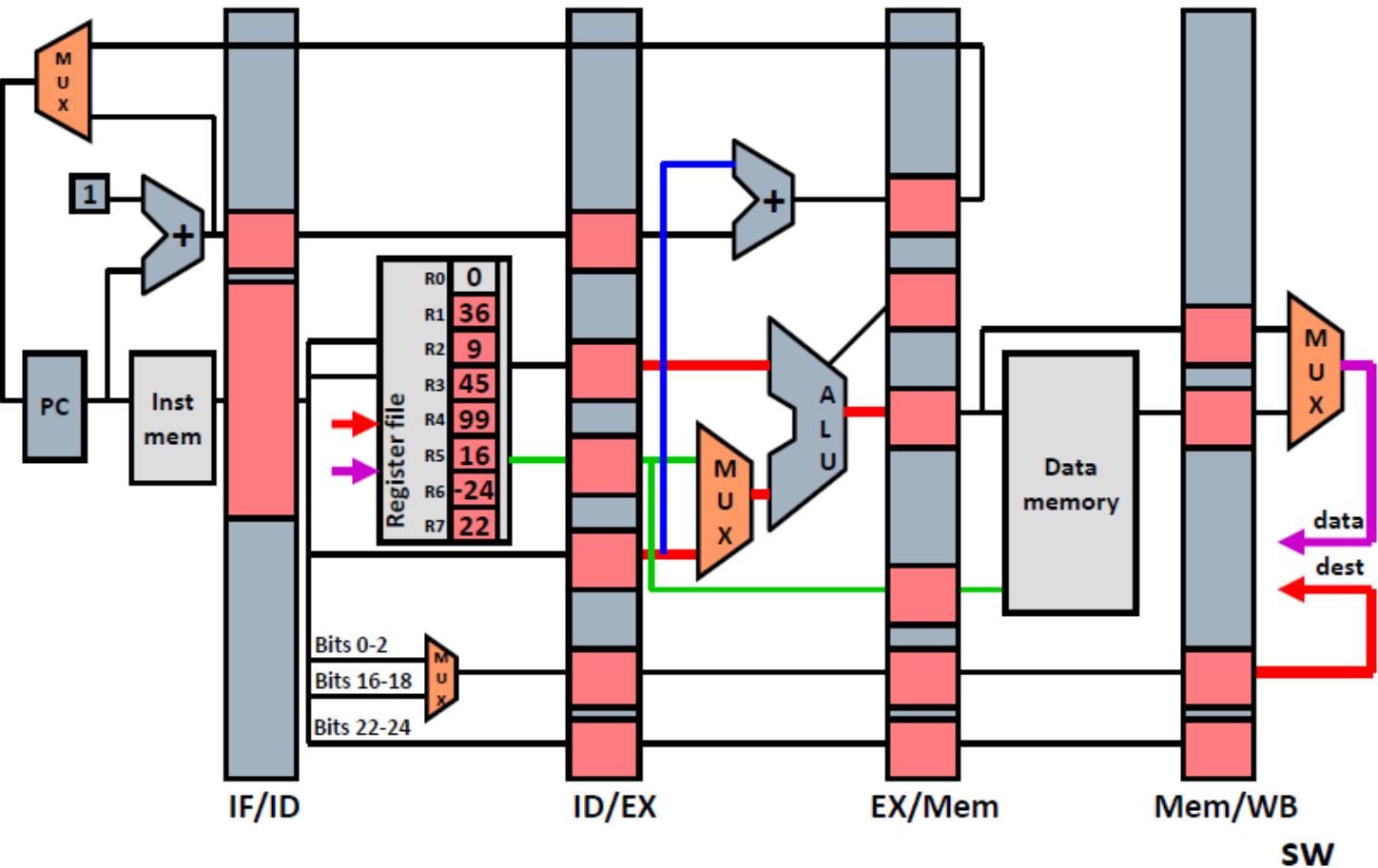
add	1	2	5
nor	4	5	6
lw	2	4	20
add	2	5	5
sw	3	7	10

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

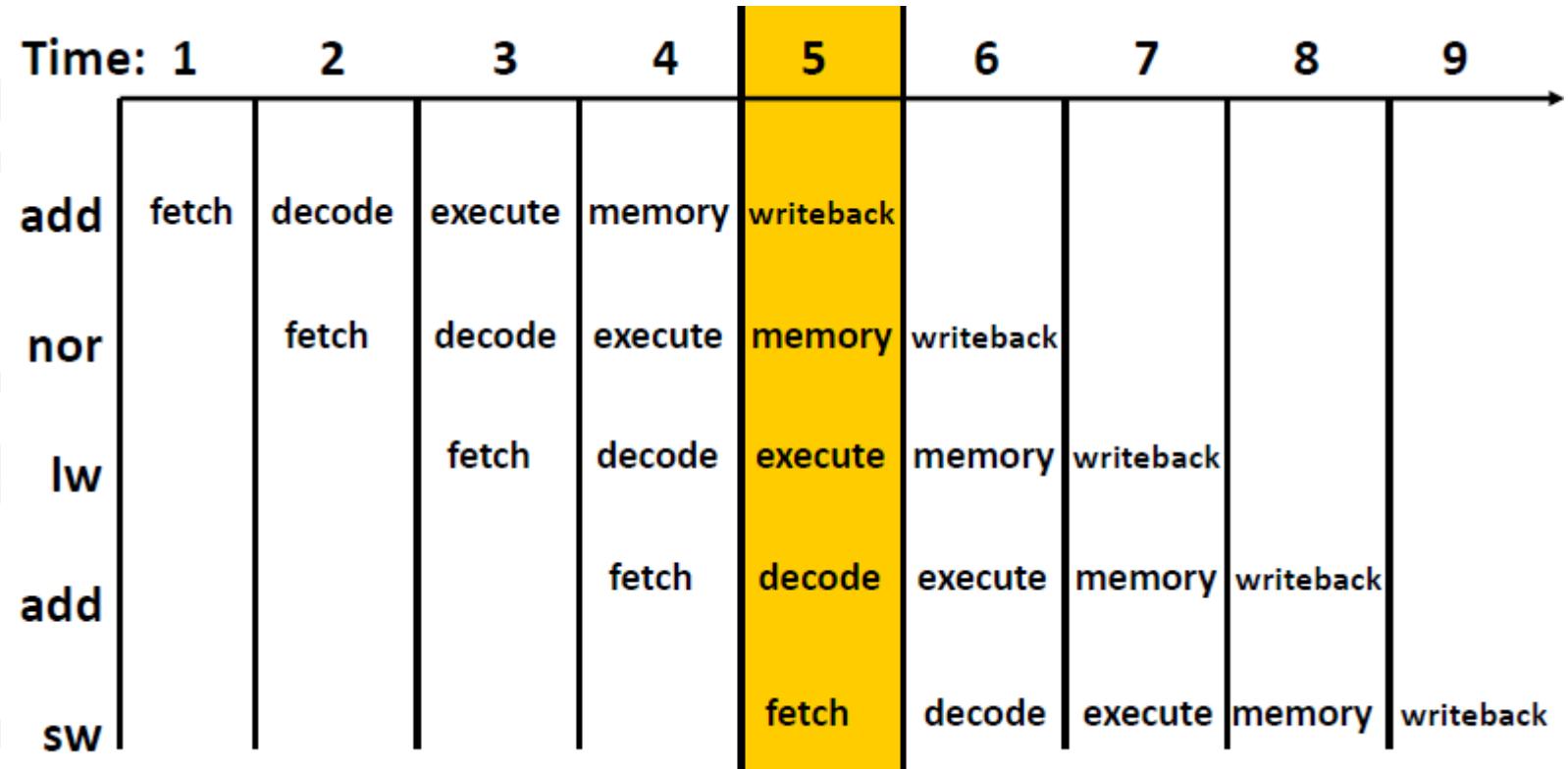
• add 1 2 3 ; reg 3 = reg 1 + reg 2
 • nor 4 5 6 ; reg 6 = reg 4 nor reg 5
 • lw 2 4 20 ; reg 4 = Mem[reg2+20]
 • add 2 5 5 ; reg 5 = reg 2 + reg 5
 • sw 3 7 10 ; Mem[reg3+10] = reg 7

Time 9 – no more instructions



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



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

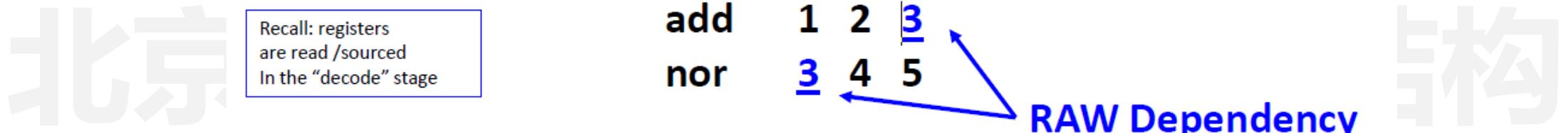
简单5级流水线可能存在什么问题？

- **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

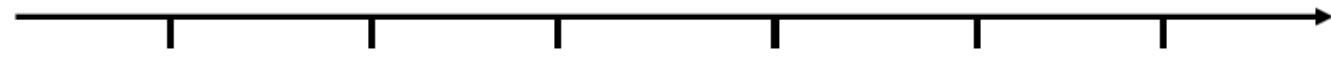
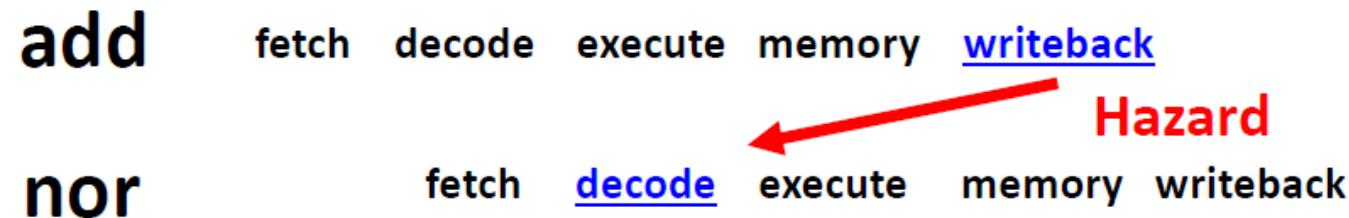
主讲：陶耀宇、李萌

问题1：Data Hazards

- RAW问题：Read After Write数据冲突



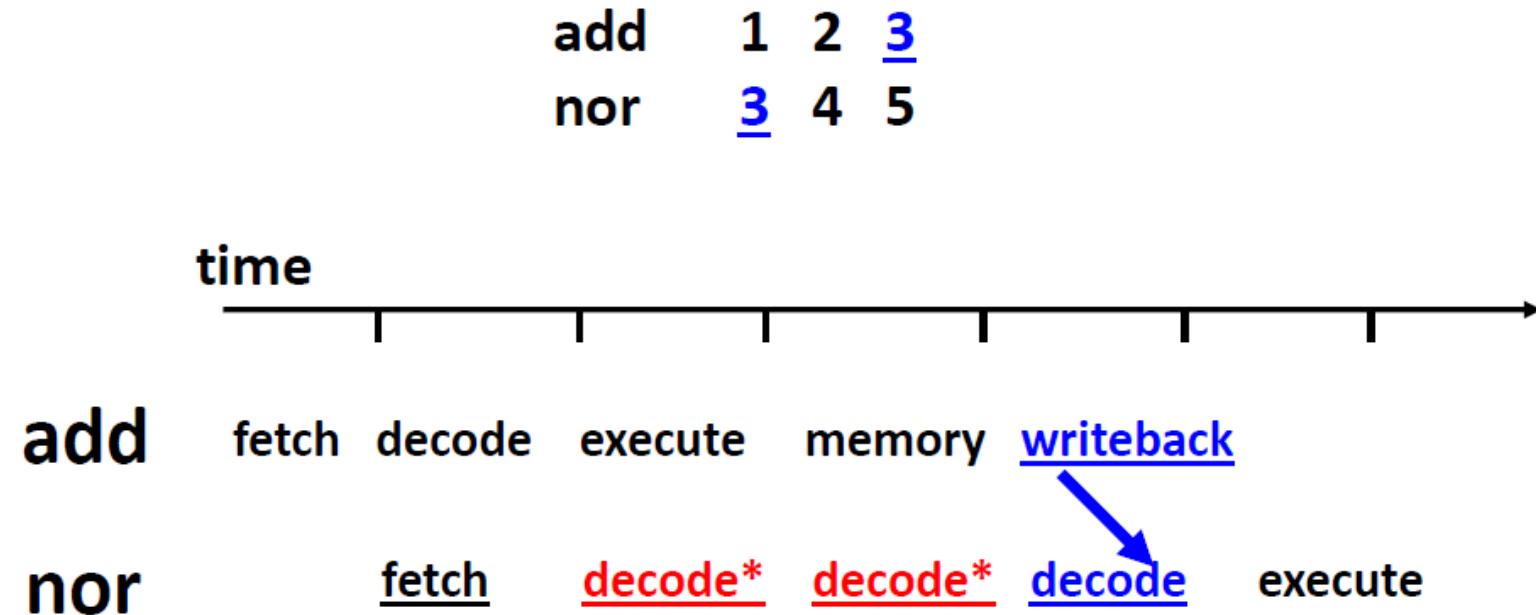
time

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

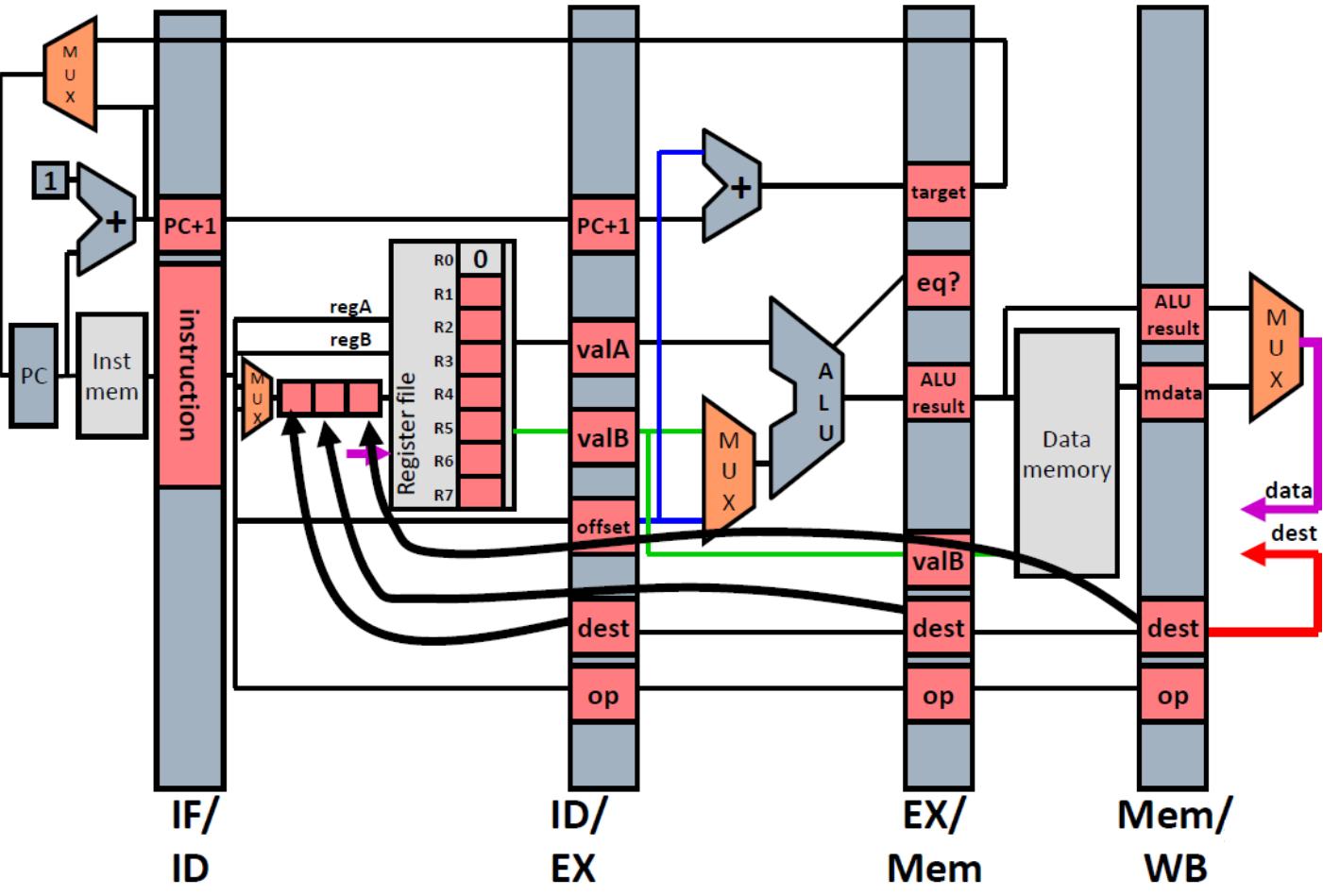
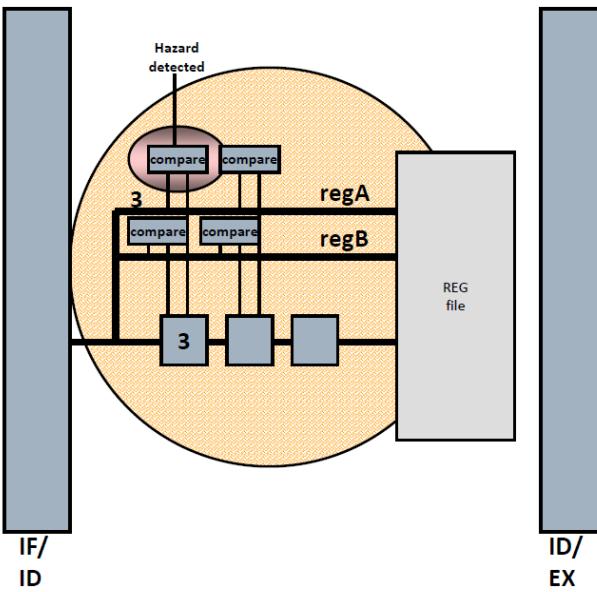


简单解决办法：流水线停顿 (Pipeline Stall)

问题1：Data Hazards

- RAW问题：Read After Write数据冲突

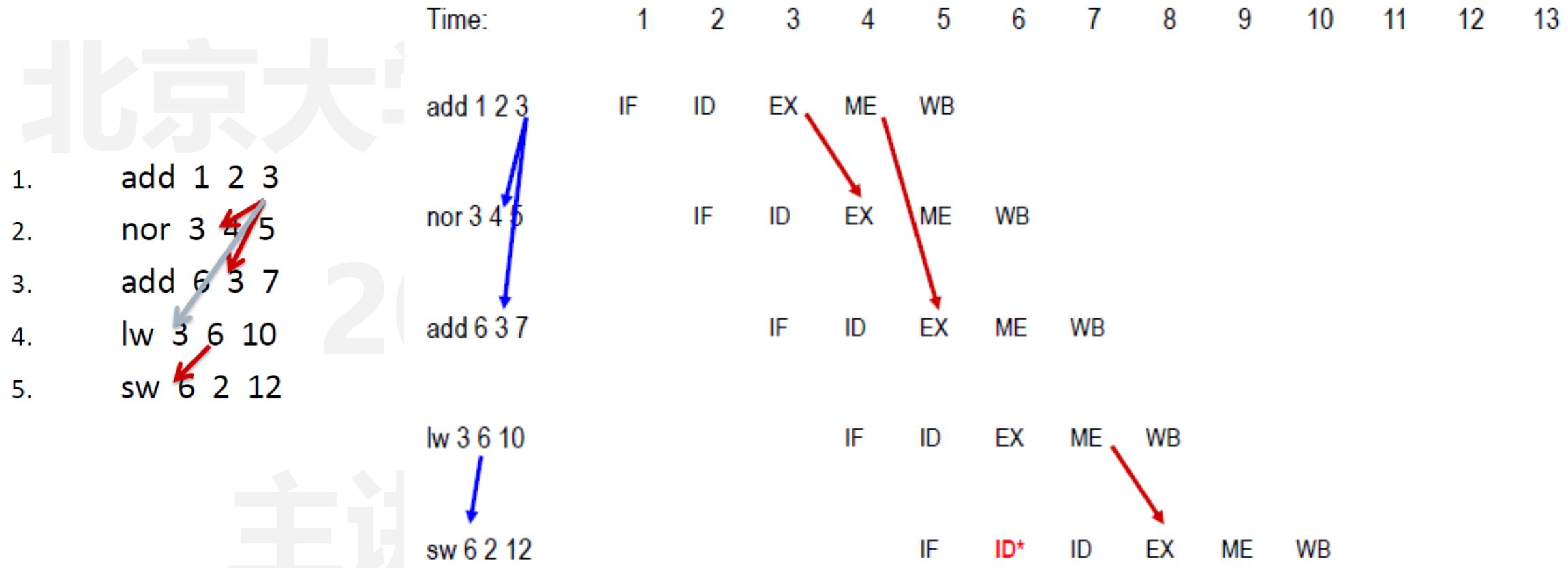
1. add 1 2 3
 2. nor 3 4 5
 3. add 6 3 7
 4. lw 3 6 10
 5. sw 6 2 12



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

- 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

- 下一讲内容继续.....

北京大学-智能硬件体系结构

2024年秋季学期

主讲：陶耀宇、李萌