北京大学
PEKING UNIVERSITY

人工智能的硬件基石
从物理器件到计算架构

第五讲：指令集与流水线架构-2

主讲：陶耀宇

2025年春季

注意事项

北京大学 PEKING UNIVERSITY

- **课程作业情况**

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

指令集架构一般需要哪些指令?

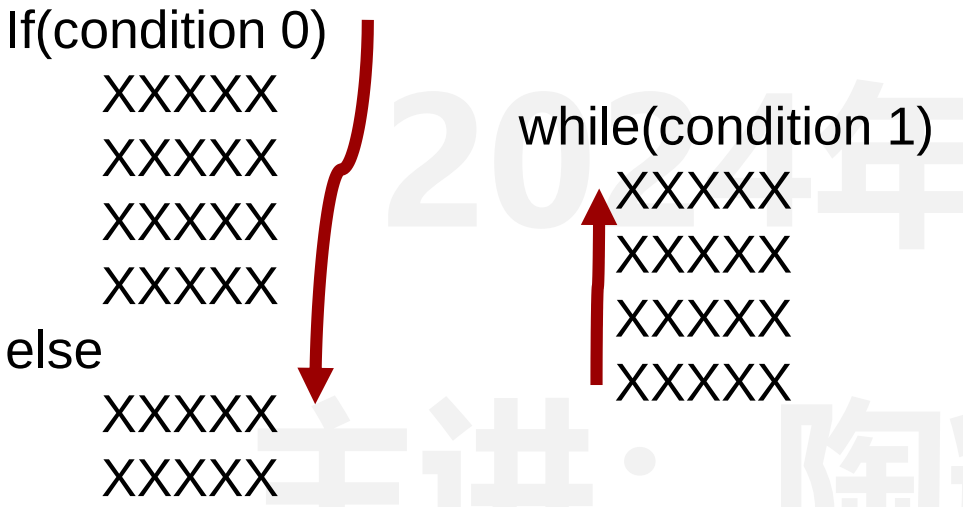
- **四大类：传输指令、运算指令、控制指令、系统指令**
  - **数据传输（mov指令）**
    - 在处理器和存储器之间传输数据（加载load/保存save变量）
    - 其中一个操作数必须是处理器的寄存器（不能在两个存储器位置之间传输数据）
    - 通过指令的后缀来指定具体大小（movb，movw，movl，movq）
  - **算术逻辑单元ALU操作**
    - 其中一个操作数必须是处理器的寄存器
    - 通过指令来指定大小和操作（addl，orq，andb，subw）
  - 控制指令
    - 无条件/有条件跳转（cmpq，jmp，je，jne，jl，jge）
    - 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.)

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

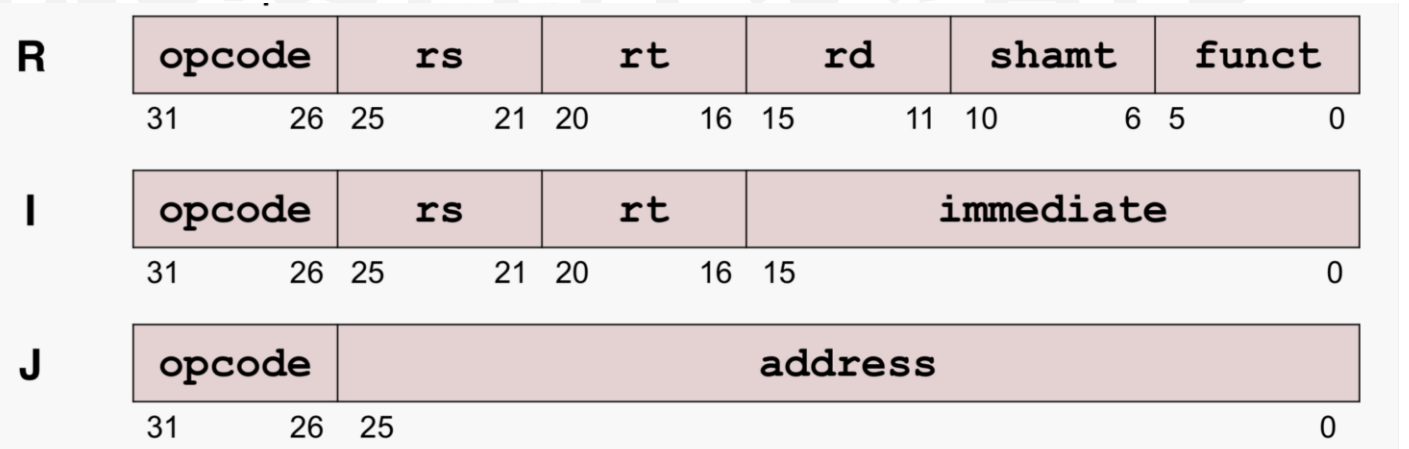
代表性指令集： MIPS一种典型的RISC指令

- 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

•6bit 操作数

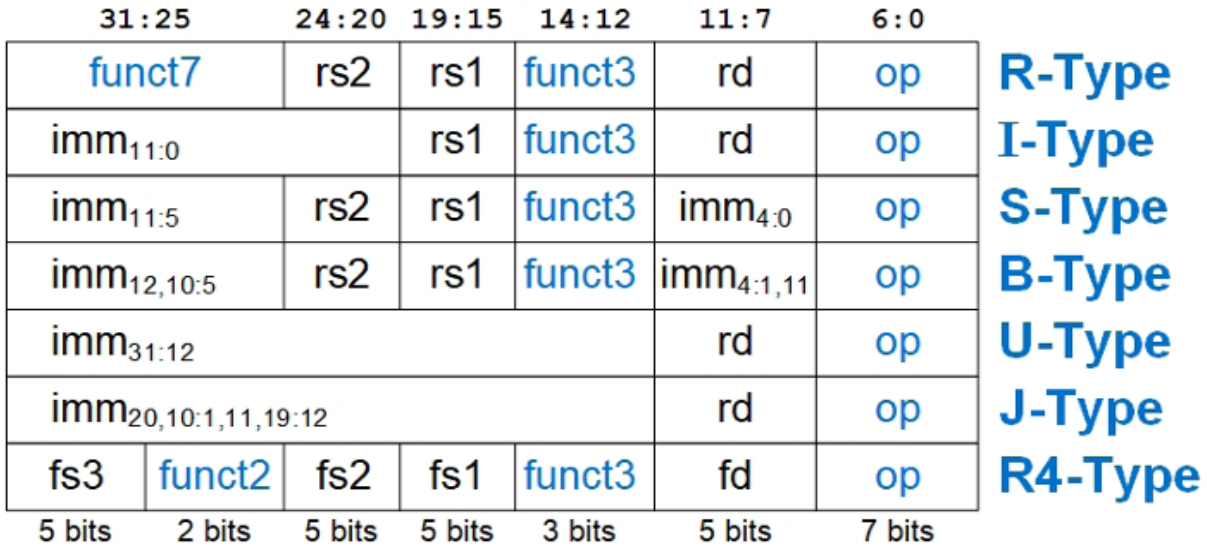
•有三种不同类别的寄存器



- RD - 5-bit destination register
- RS - 5-bit source register
- RT - 5-bit target register

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



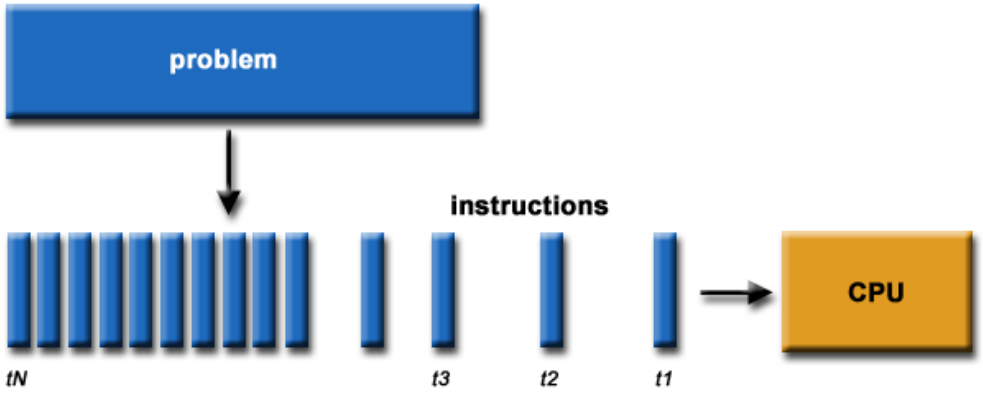
31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	R-Type
$imm_{11:0}$		rs1	funct3	rd	op	I-Type
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	S-Type
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	B-Type
$imm_{31:12}$				rd	op	U-Type
$imm_{20,10:1,11,19:12}$				rd	op	J-Type
fs3	funct2	fs2	fs1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

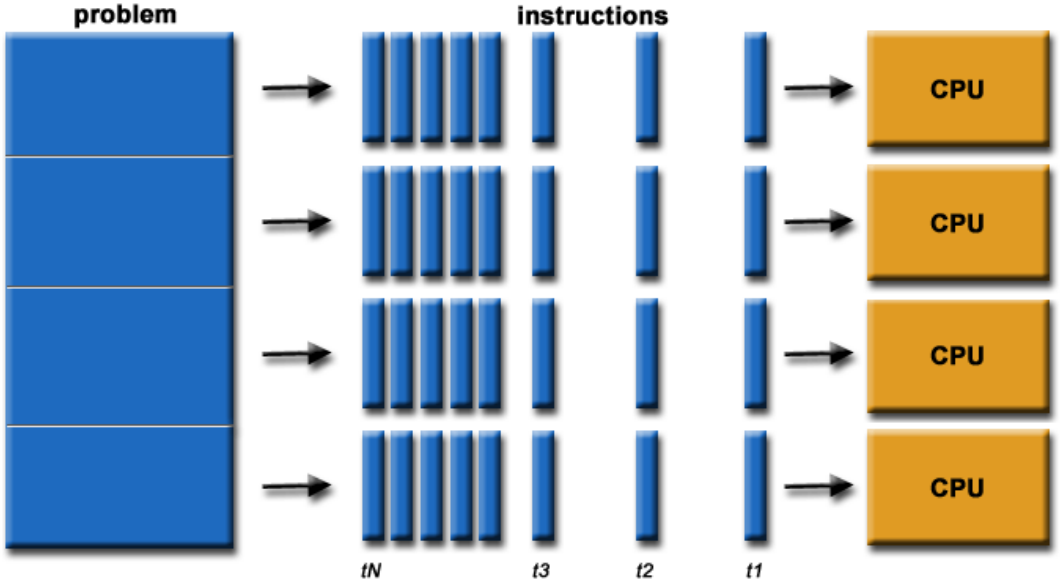
- imm : 12b位宽有符号立即数,如imm11:0
- uimm: 5b位宽无符号立即数,如imm4:0
- upimm: 20b位宽立即数，位于指令的高位，如imm31:12
- Address: rs1 + SignExt(imm11:0)
- [Address]: 在Address处的数据
- BTA: branch target address，即PC+ SignExt({imm12:1，1`b0})
- JTA: jump target address,即PC+ SignExt({imm20:1，1`b0})
- Label: 文本表明指令地址
- SignExt: 高位补符号位，将值拓展到32b
- ZeroExt: 高位补0，将值拓展到32b
- Csr: 控制与状态寄存器

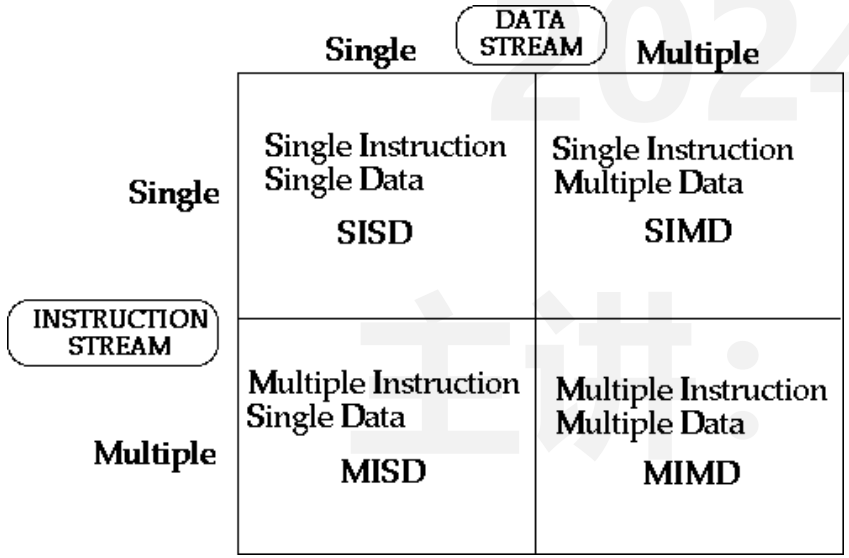
代表性指令集： GPU的CUDA指令集

- SISD、SIMD、MISD、MIMD的扩展

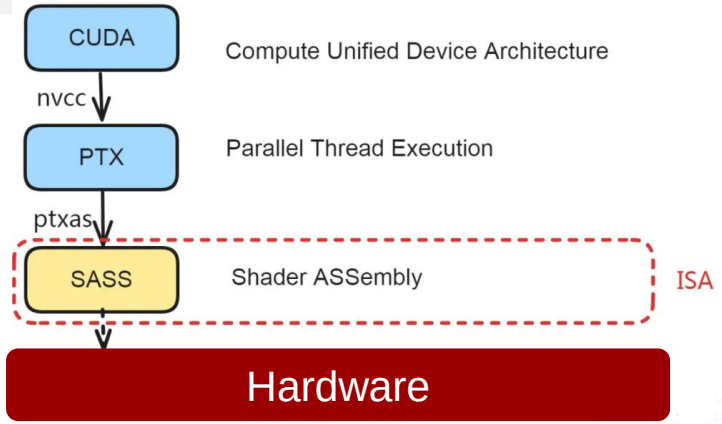


并行计算



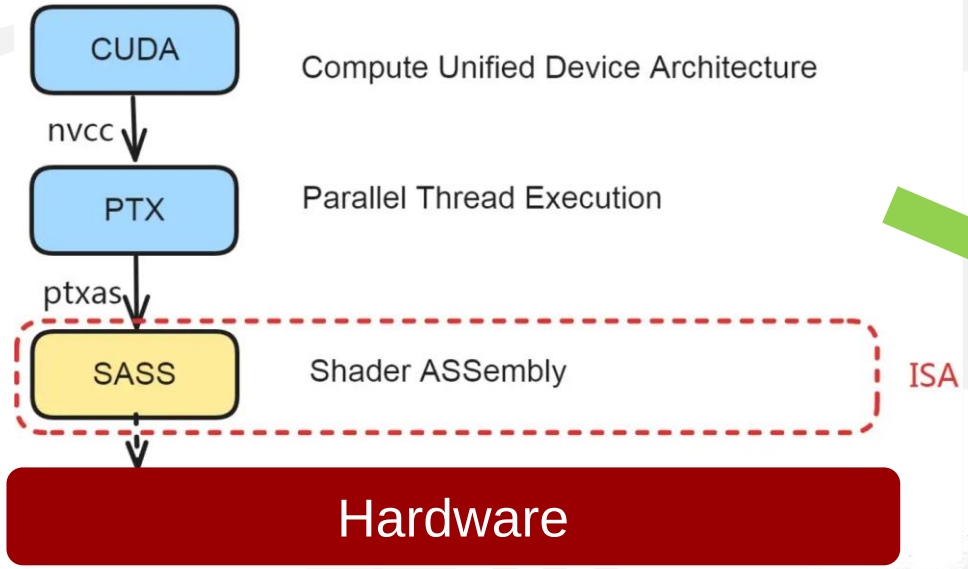


CUDA指令集的层次



代表性指令集： GPU的CUDA指令集

- PTX和SASS



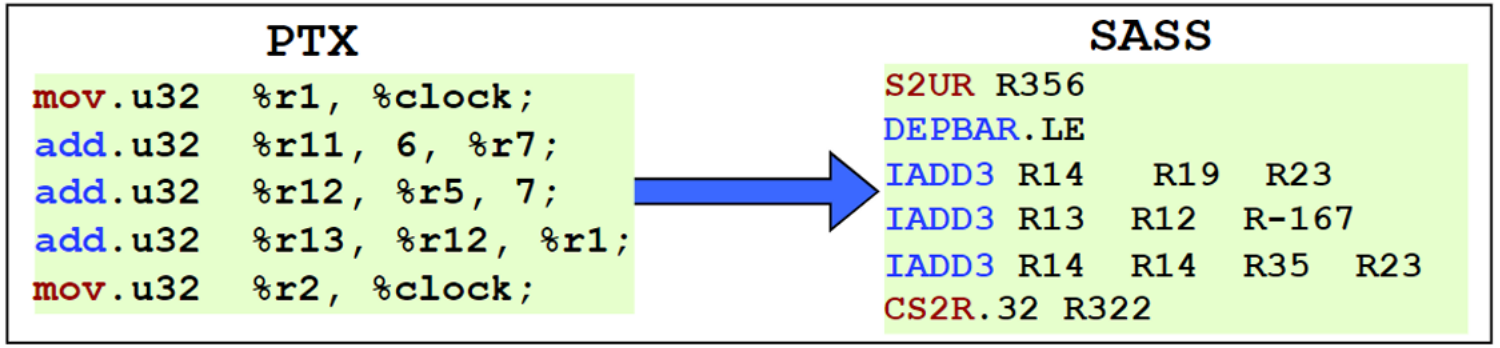
CUDA C/C++程序编译后，一般NVCC会同时生成PTX和SASS，用户也可以指定只生成其中一种。SASS是机器码的硬件指令集，编译的SM版本与当前GPU的SM版本不对应的话是不能运行的

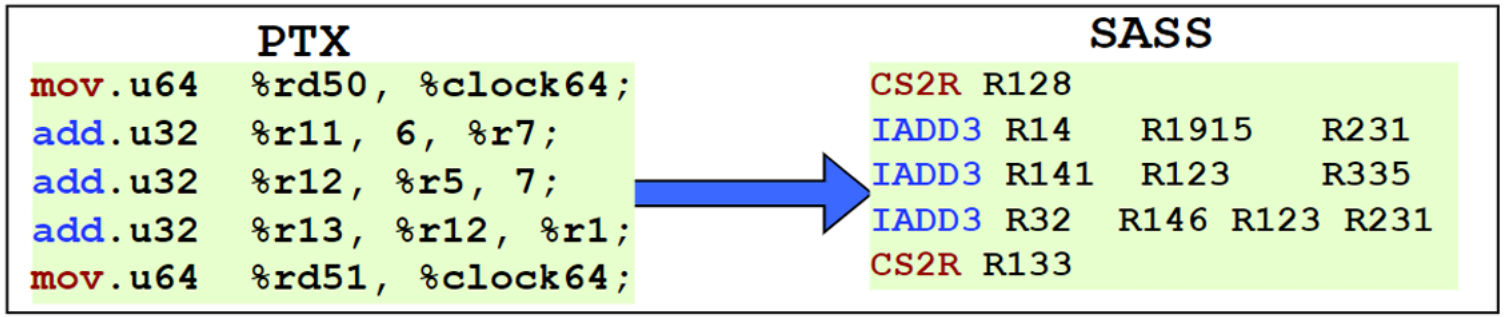
从SASS上抽象出来的一种更上层的软件编程模型，介于CUDA C/C++和SASS之间

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

北京大学 PEKING UNIVERSITY

- PTX和SASS

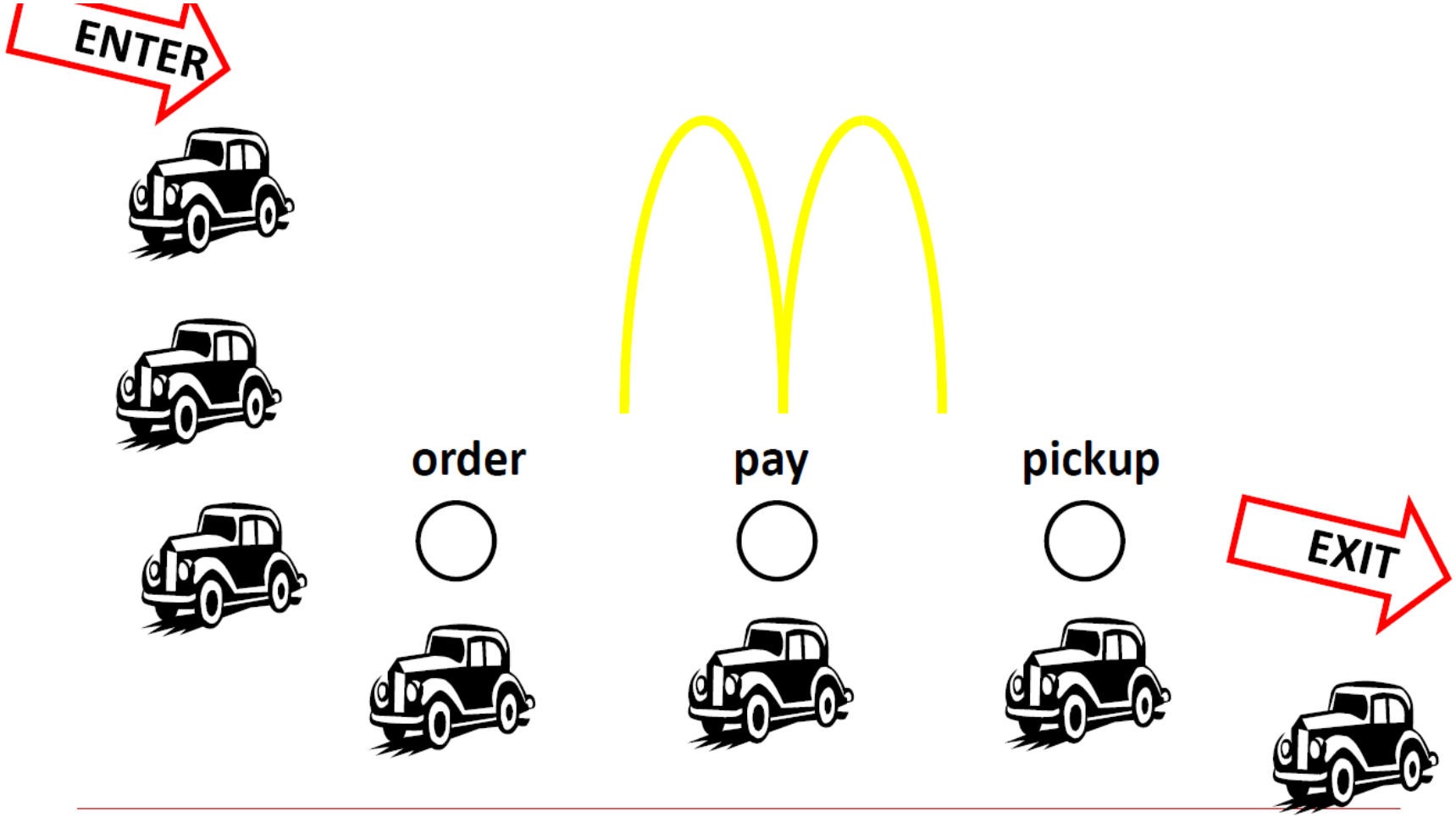
Div / Rem Instruction			Pop Instruction		
rem/div.u16/s16	multiple instructions	290	popc.b32S	POPC	6
rem/div.s32/u32	multiple instructions	66	popc.b64	2*UPOPC + UIADD3	7
rem/div.u64/s64	multiple instructions	420	**Clz Instruction**		
div.rn.f32	multiple instructions	525	clz.b32	FLO.U32 + IADD	7
div.rn.f64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

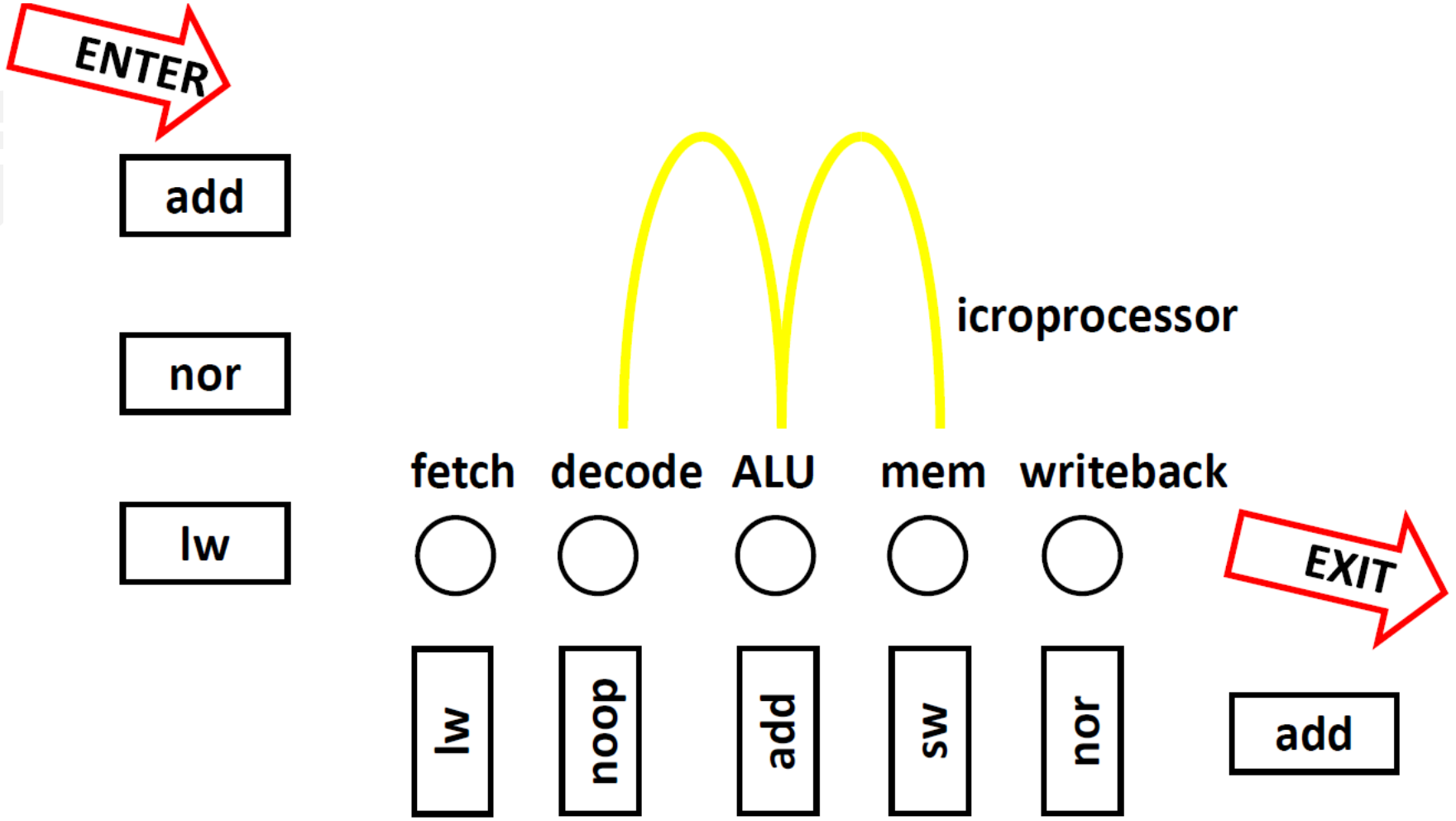
回顾：什么是流水线架构

- 流水线式运行方式 - 提高吞吐率的有效手段



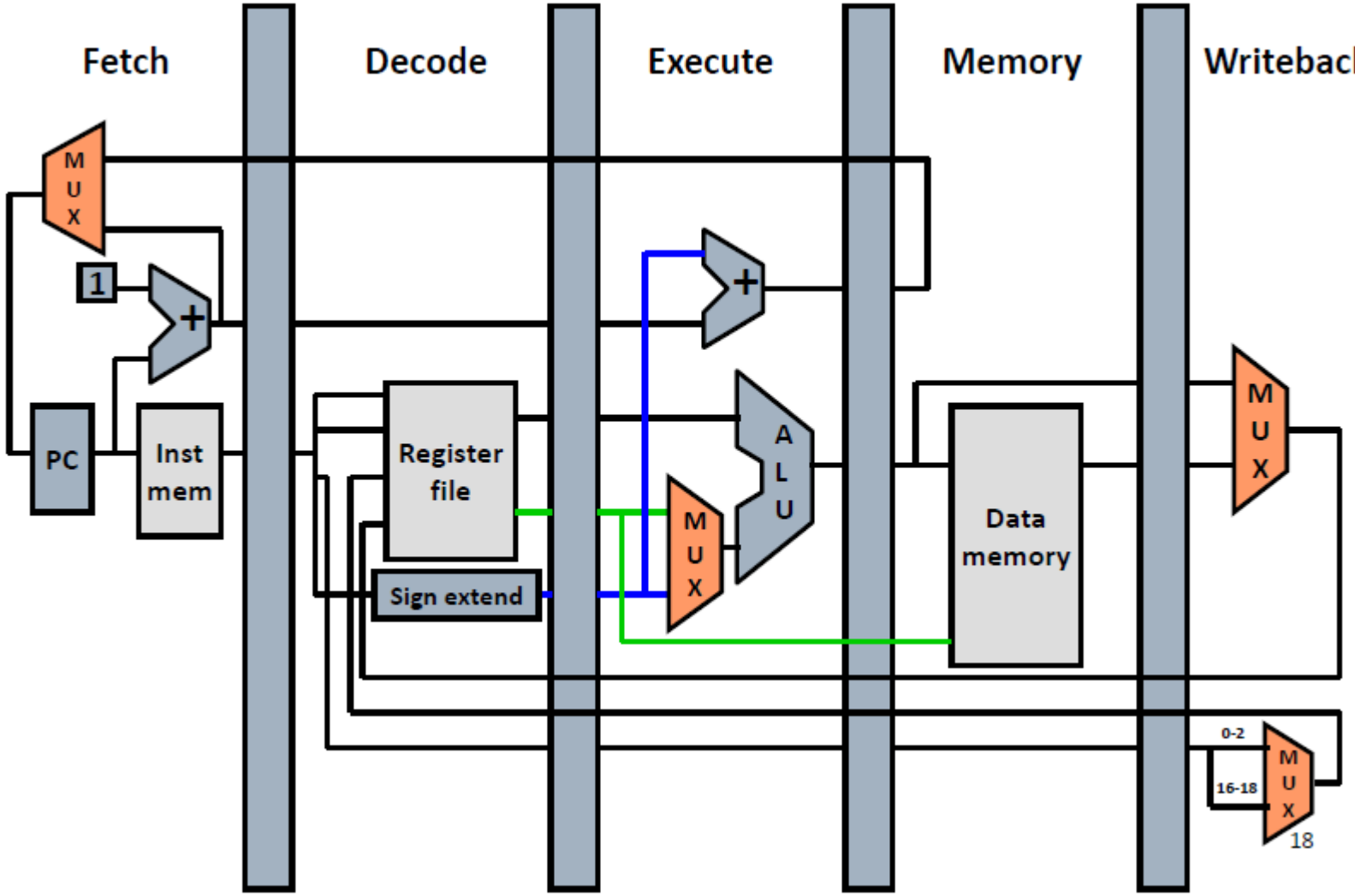
回顾：什么是流水线架构

- 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

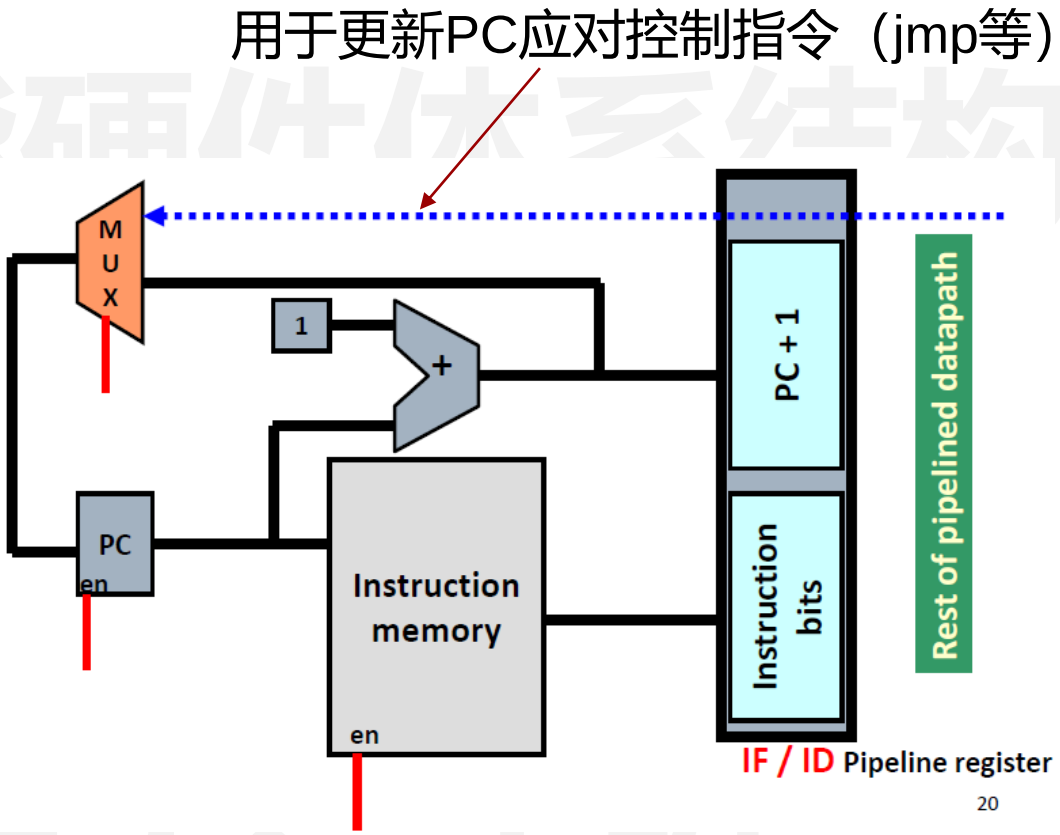
- 5级流水线设计：Fetch、Decode、Execute、Memory、Writeback



第一级：Fetch指令

- PC控制从指令Memory里读取的地址

- 设计一个可以每个周期都从存储器获取一个指令的数据路径
  - 使用PC来索引存储器，从而读取指令
  - 假设没有跳转，每周期递增PC
- 把后面几级执行命令所需的所有数据写入IF/ID寄存器中
  - 下一级将读取这个流水线寄存器
  - 注意流水线寄存器必须是边沿触发

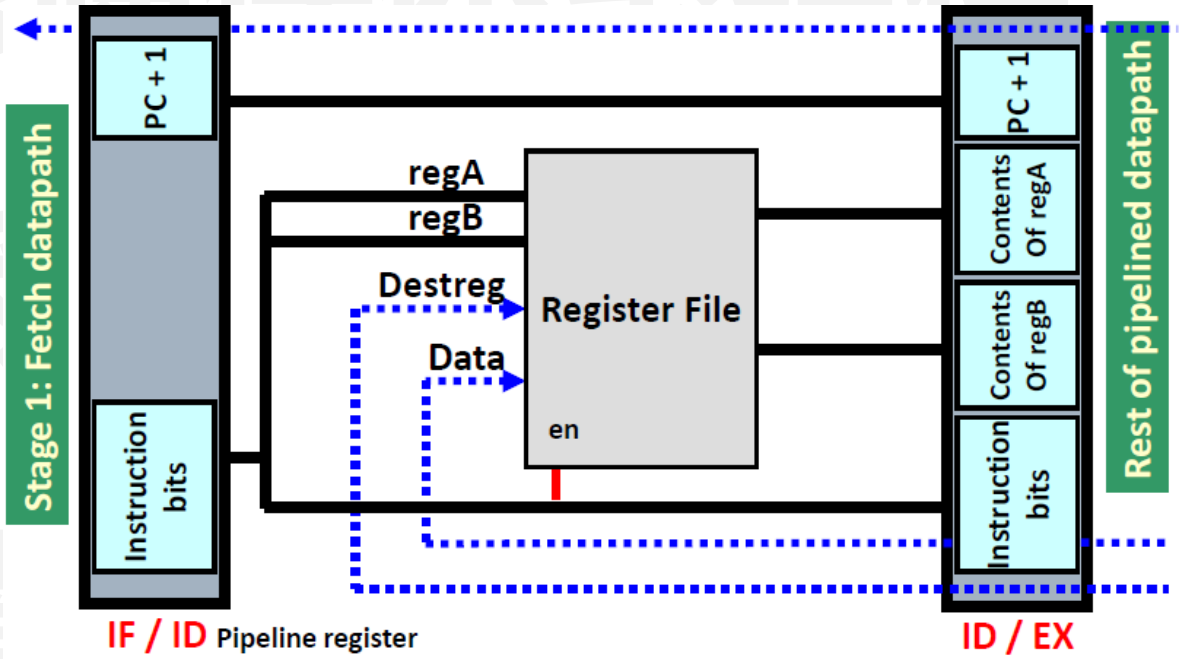


第二级：Decode指令

- 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

- 设计一个读取IF/ID流水线寄存器，解码指令，读取寄存器集群（由指令中的regA/B指定）的数据通路
  - 解码是容易的，只需要传递opcode，让后几级找到面对此指令各自的控制信号即可
- 把后面几级执行命令所需的所有数据写入ID/EX寄存器中
  - 传递偏移量和目的寄存器索引（或者简单起见，可以传递整个指令）
  - 还包括了PC+1的指令地址，即便此指令没有开始解码



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

• 设计一个正确执行指令中ALU操作的数据通路，其中ALU的输入值来自ID/EX寄存器

•其输入一个为regA的内容，另一个为regB的内容或者指令中的偏置量

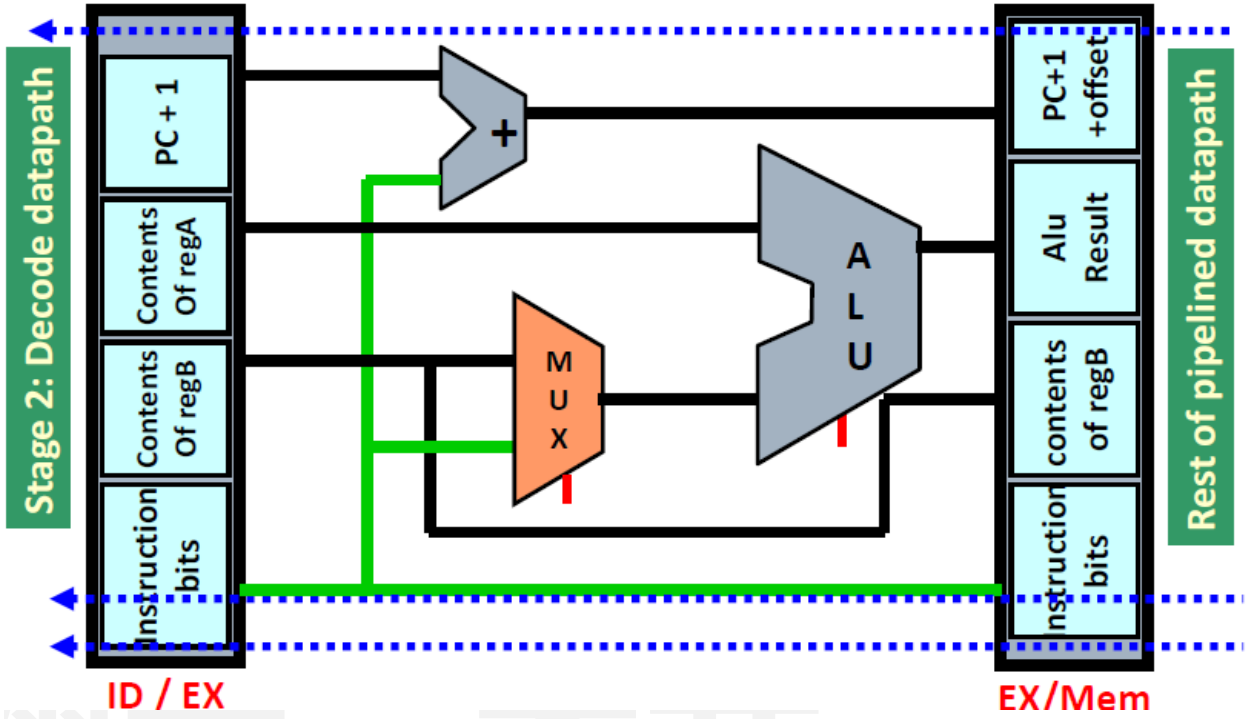
•可计算PC+1+偏置量，用于检验是否要分支跳转

•把后面几级执行命令所需的所有数据写入EX/Mem寄存器中

•ALU结果，regB内容以及PC+1+offset

•表明opcode和目的寄存器位置的指令bit

•regA和regB内容的比较结果



第四级：Memory操作

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

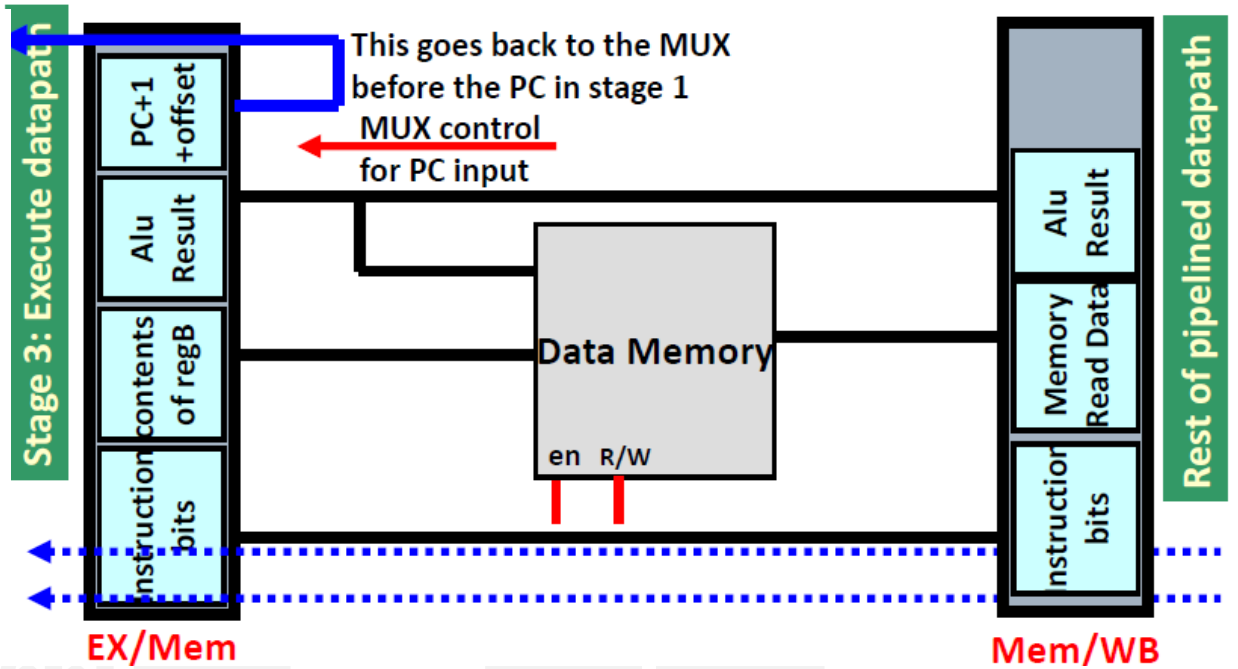
• 设计一个正确执行指令中存储器操作的数据通路，其输入来自于EX/Mem寄存器

•在ld和st指令中，ALU结果包含地址

•把后面一级执行命令所需的所有数据写入Mem/WB寄存器中

•ALU结果和MemData

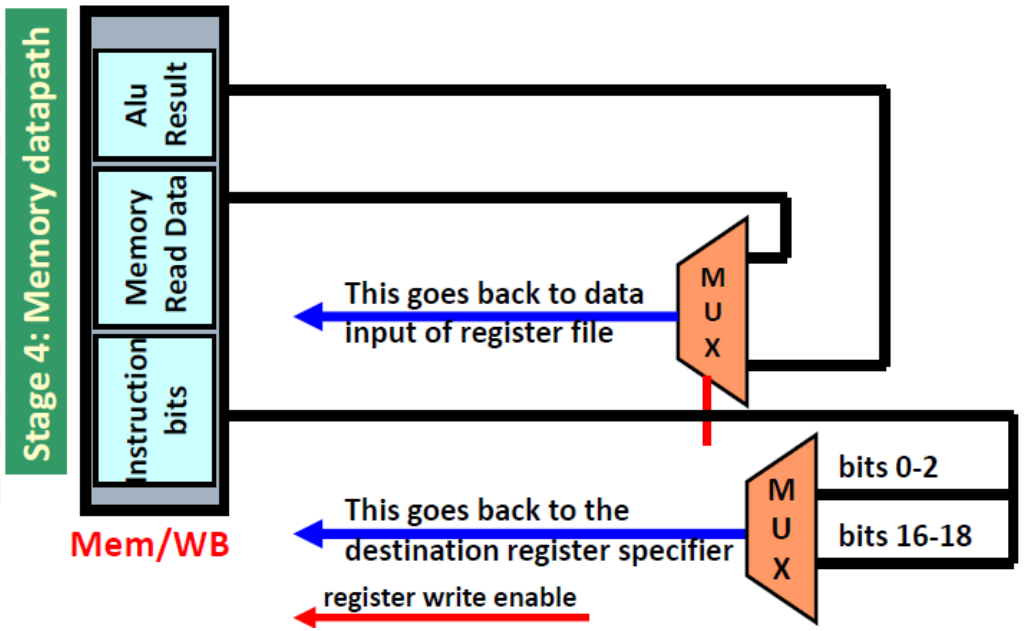
•表明opcode和目的寄存器位置的指令bit



第五级：Writeback操作

- ALU计算结果或Data Memory读取的结果写回destReg

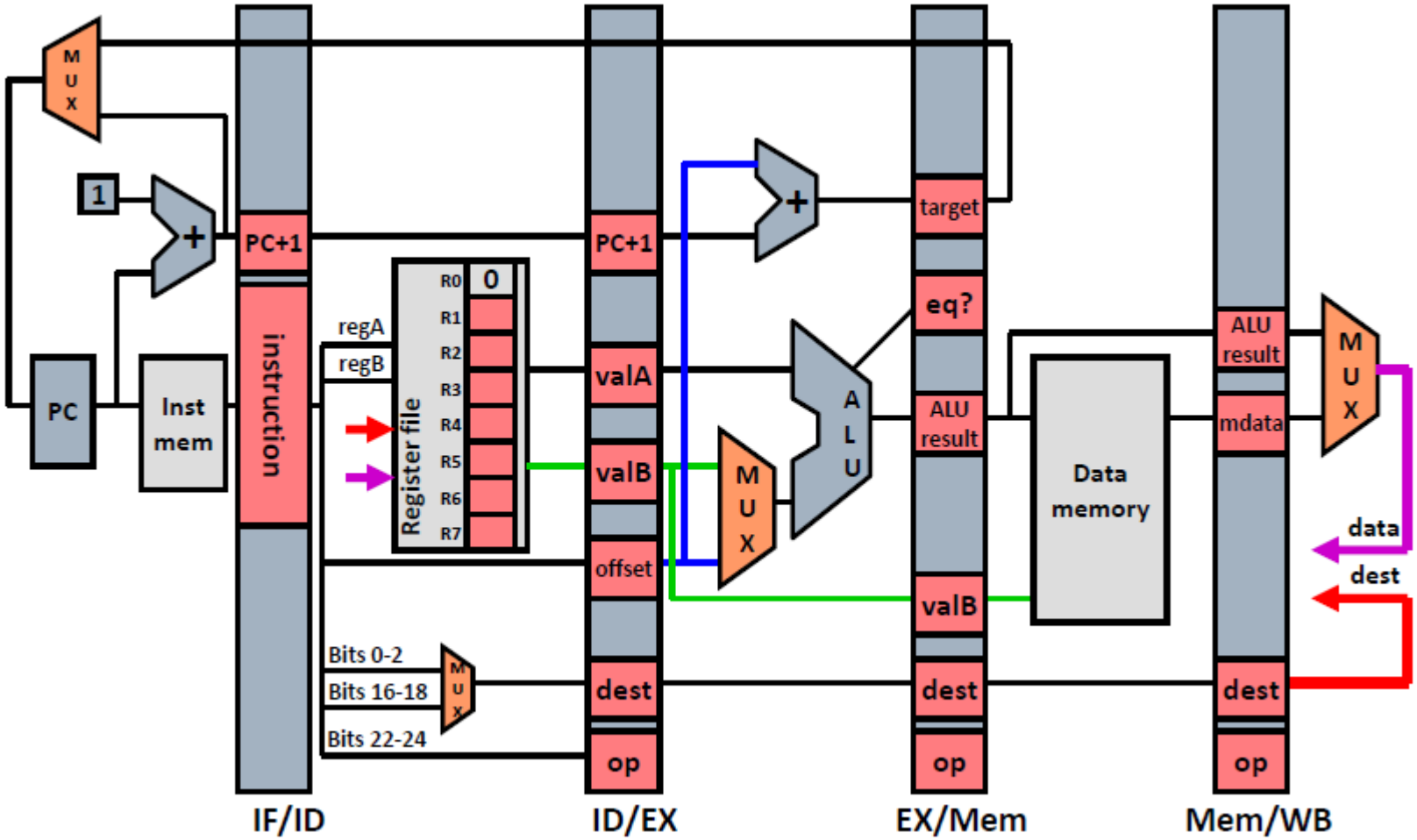
- 设计一个完成指令执行的数据通路，如果需要的话将结果写入到寄存器集群中
  - Ld指令中，将MemData写入到目的寄存器中
  - 将ALU结果写入到目的寄存器中
  - Opcode可以控制寄存器写使能信号



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

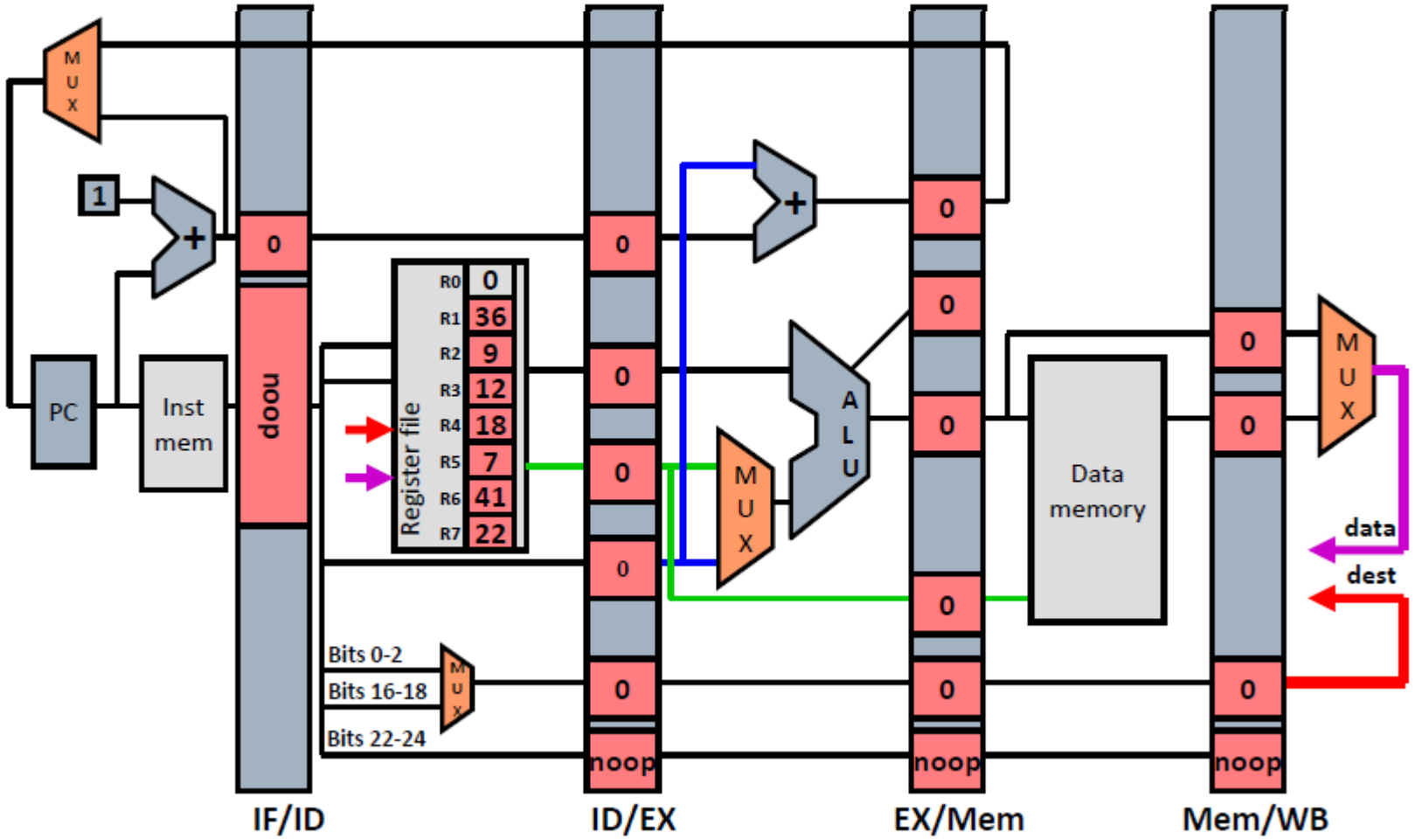
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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
```

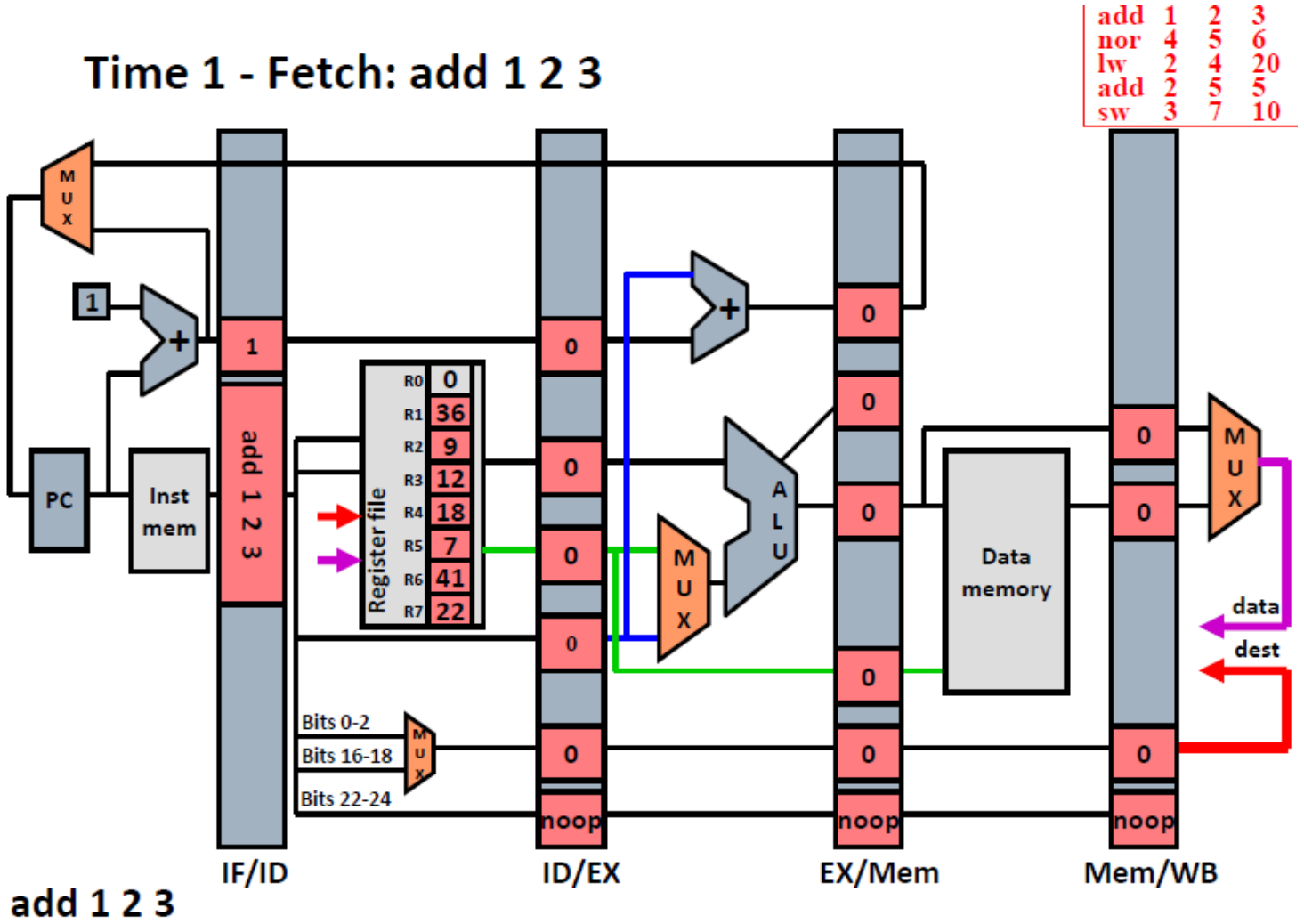


初始状态：t0

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

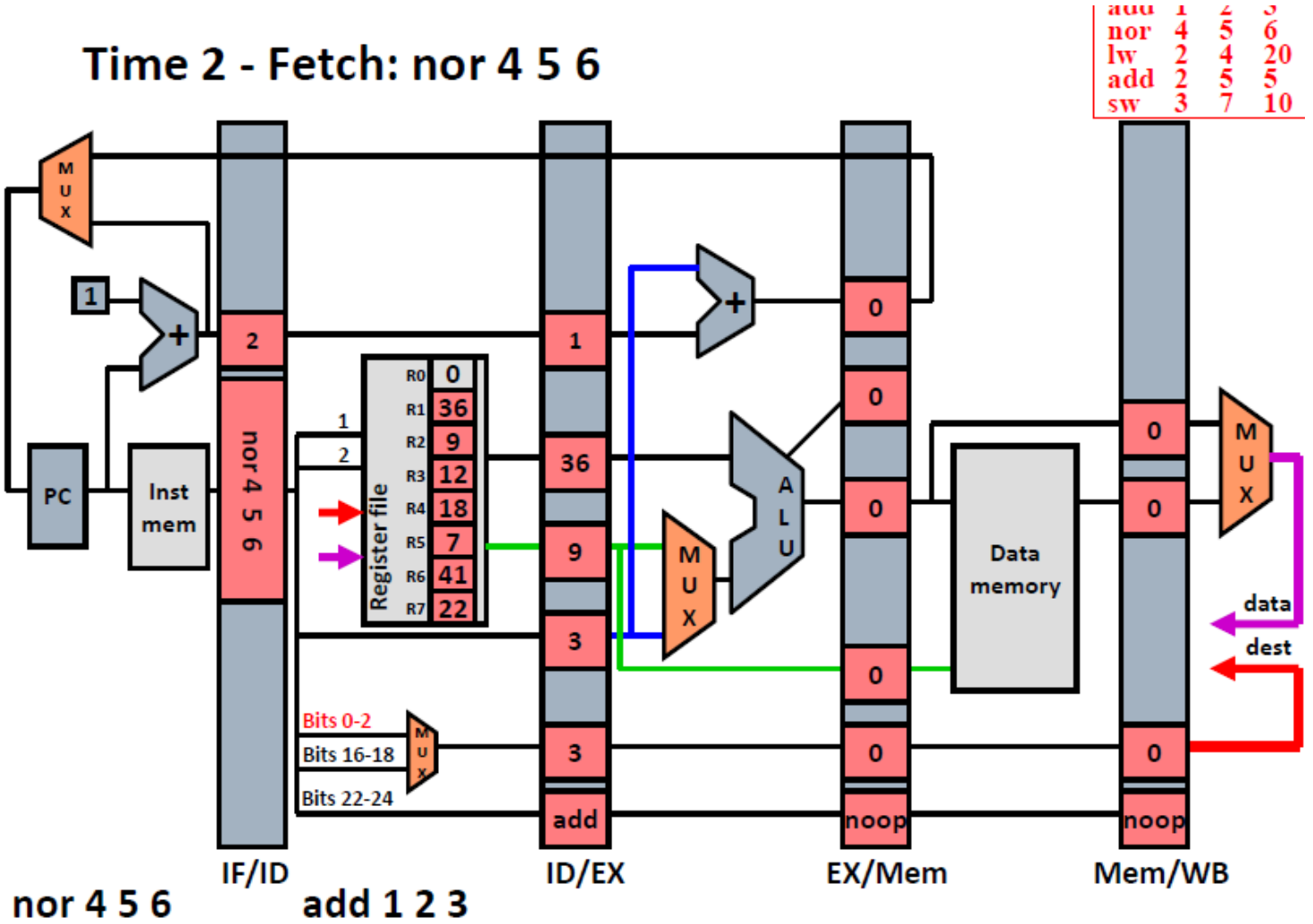
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

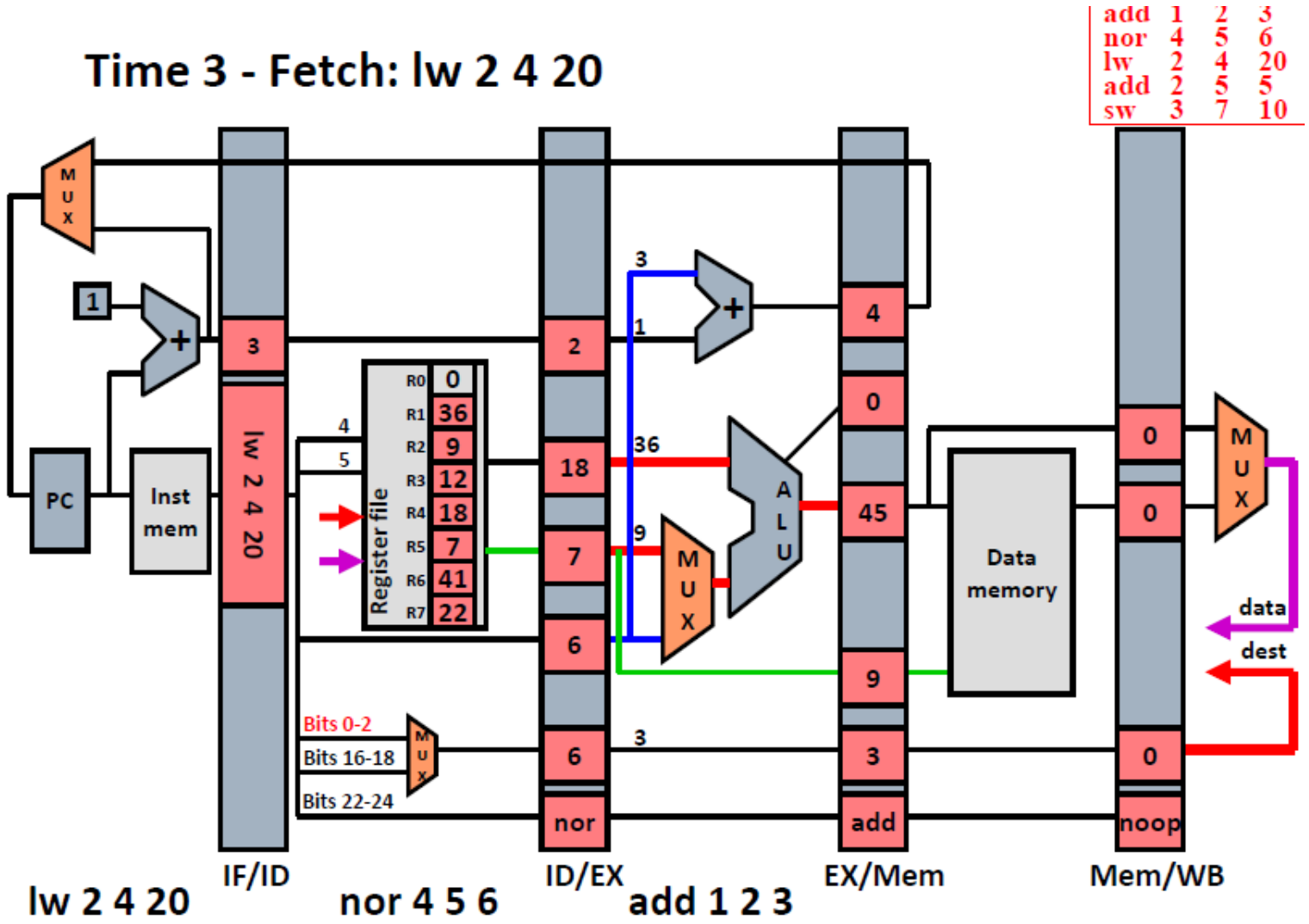
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

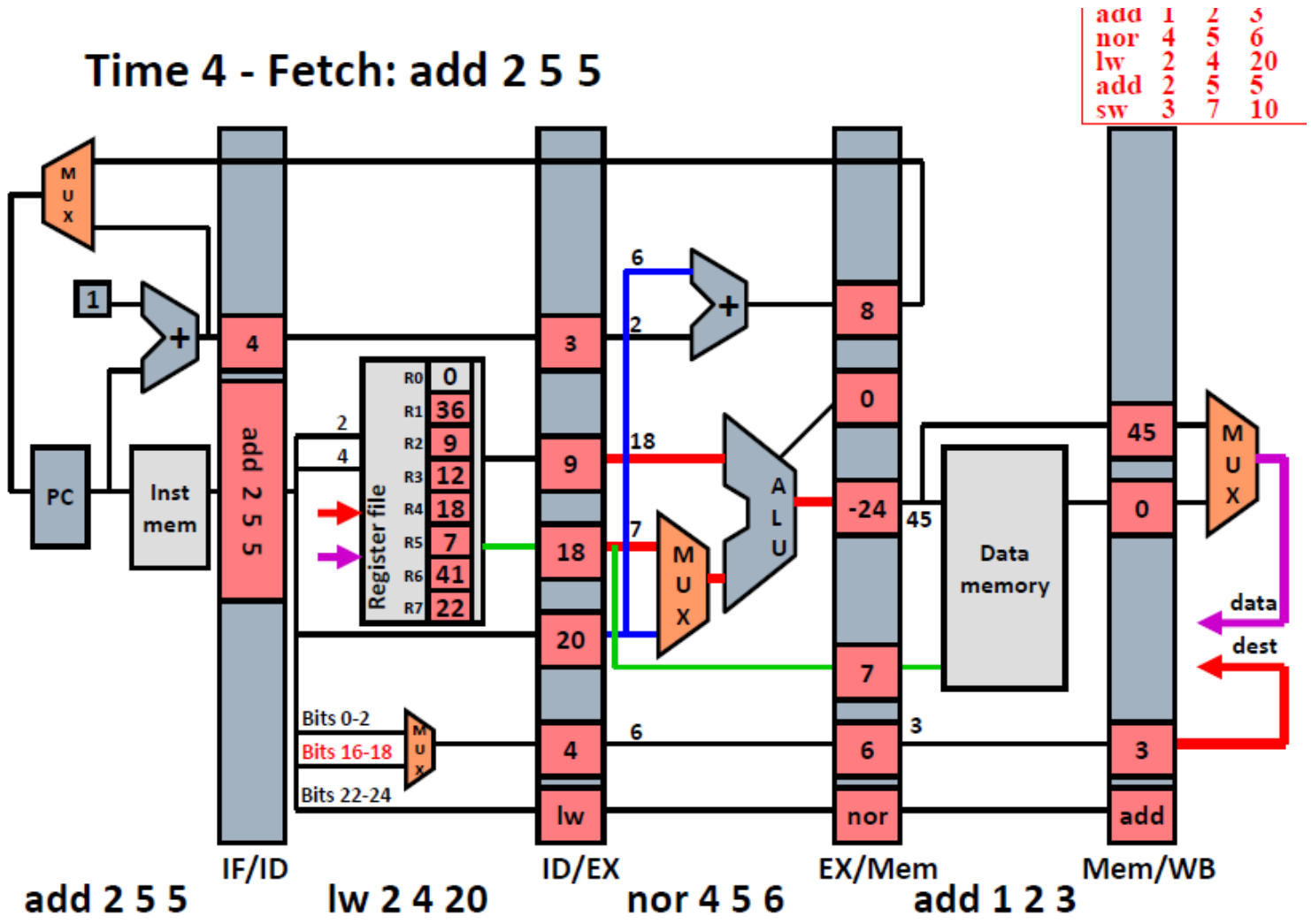
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

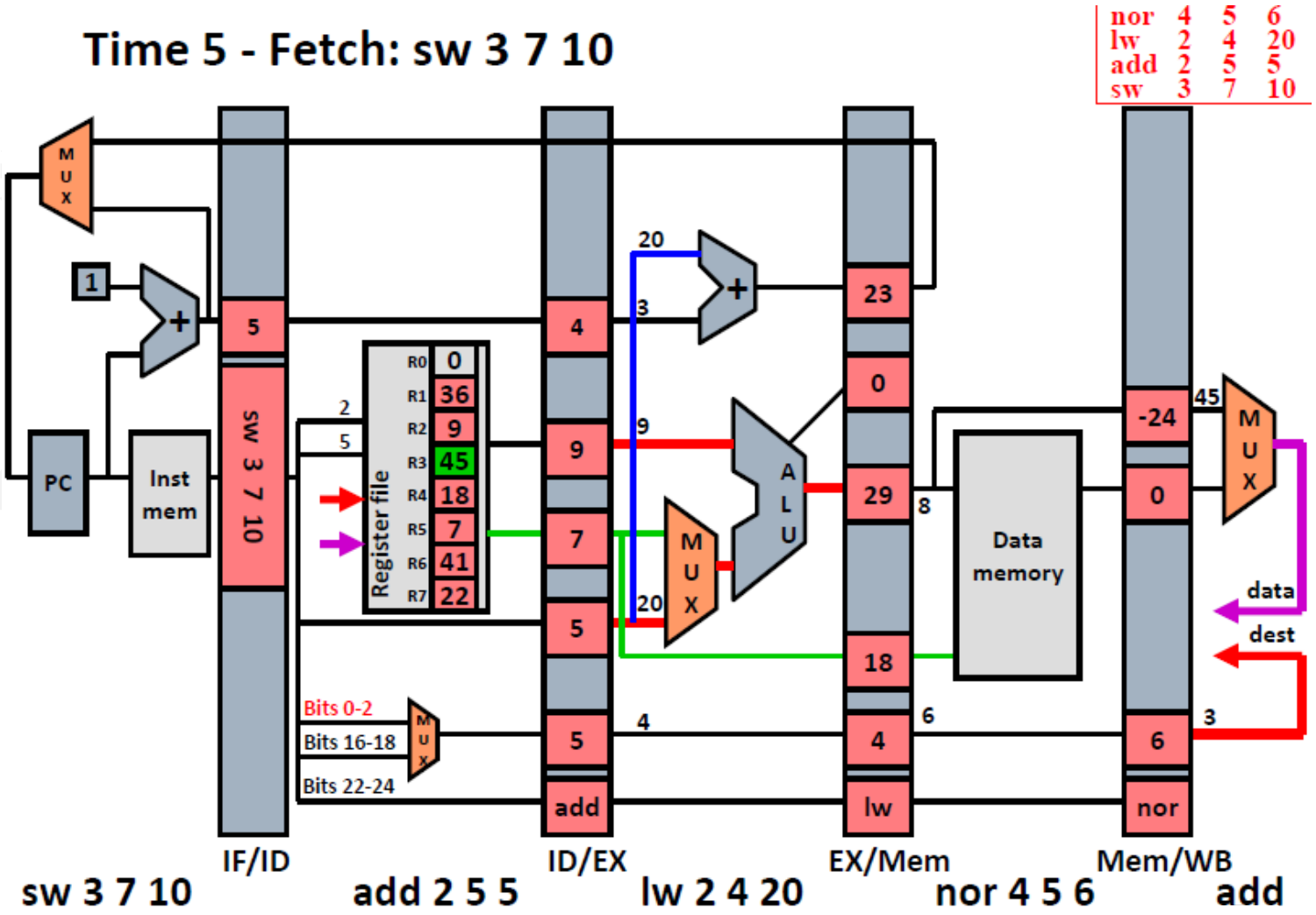
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

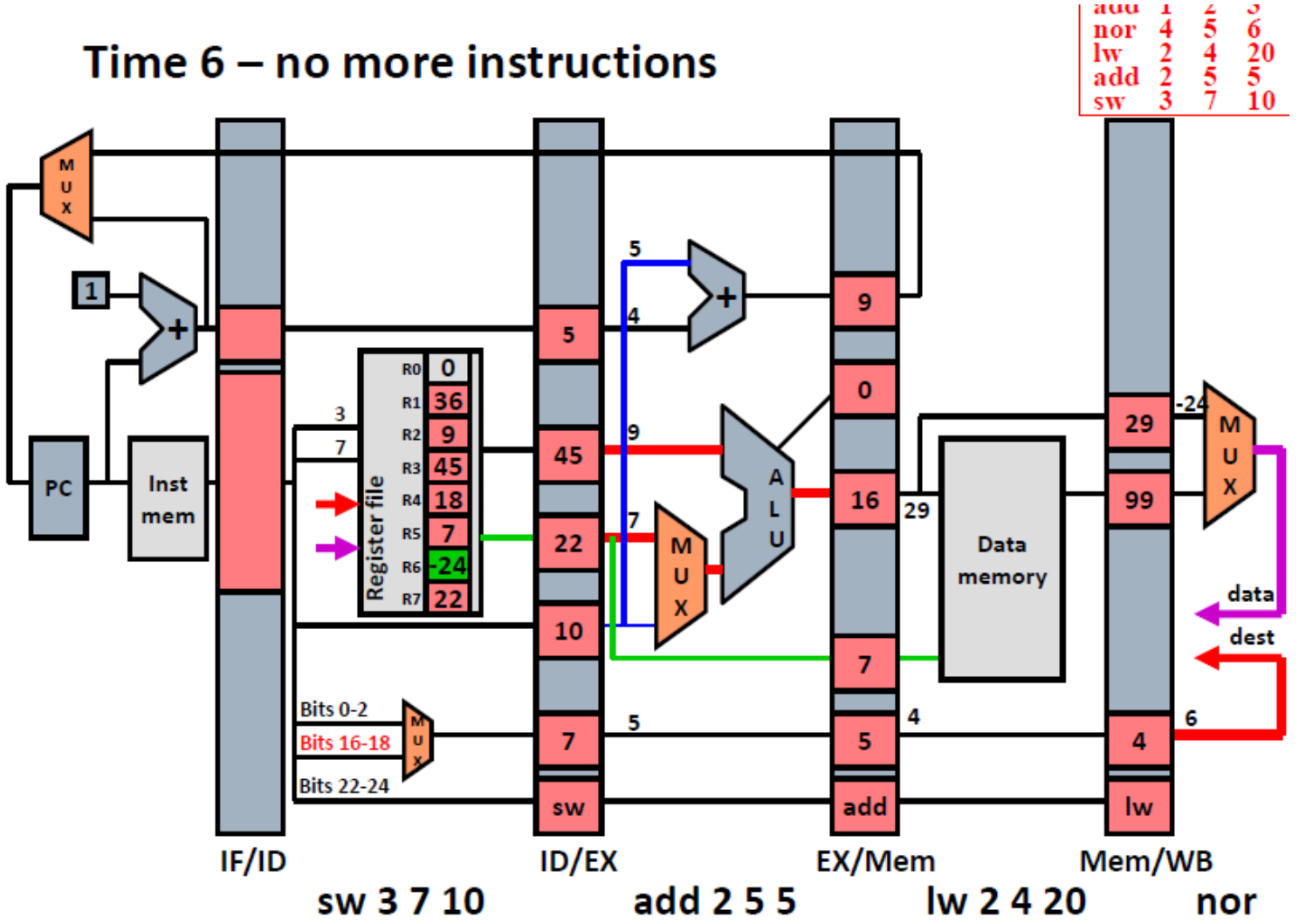
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

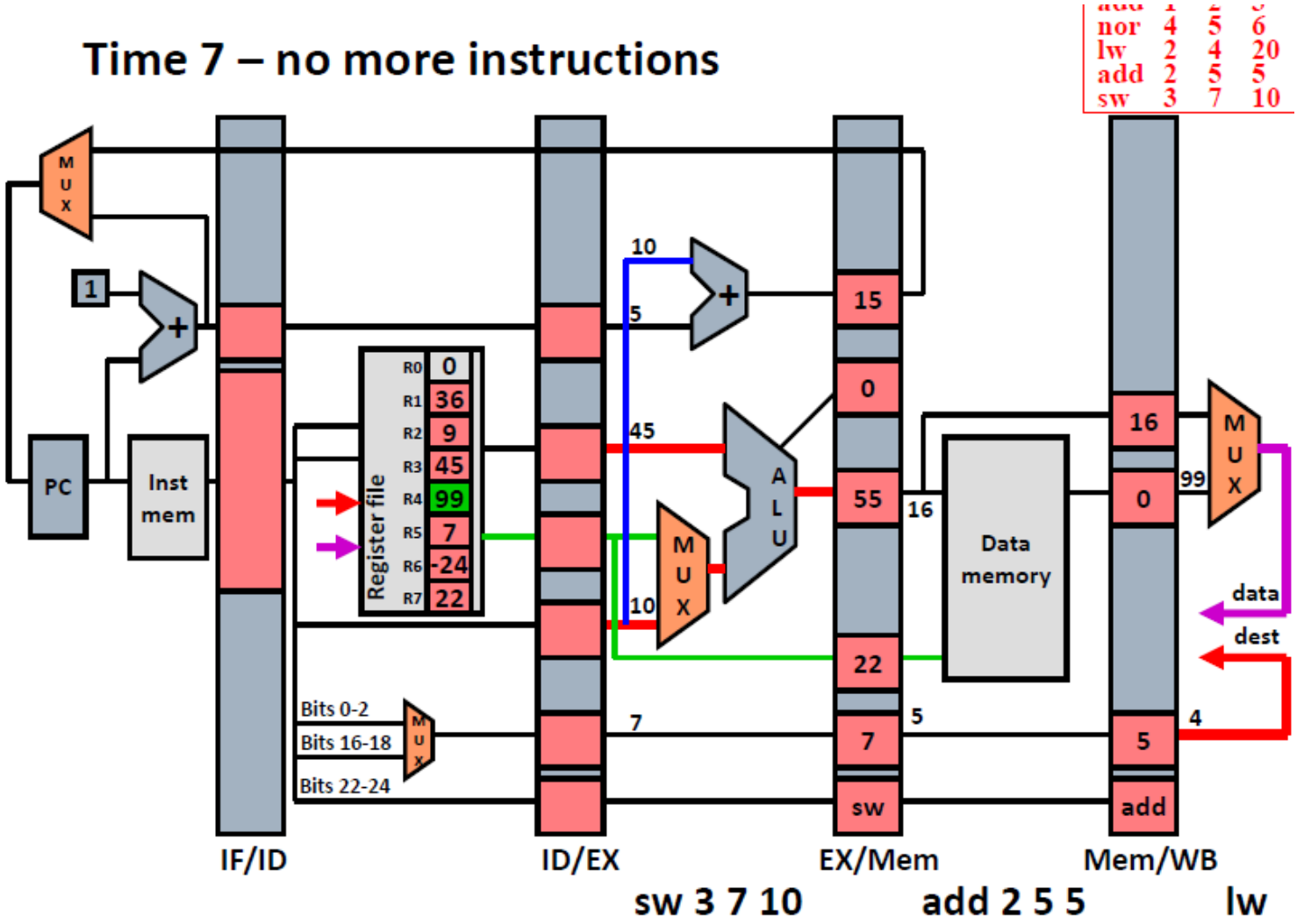
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



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

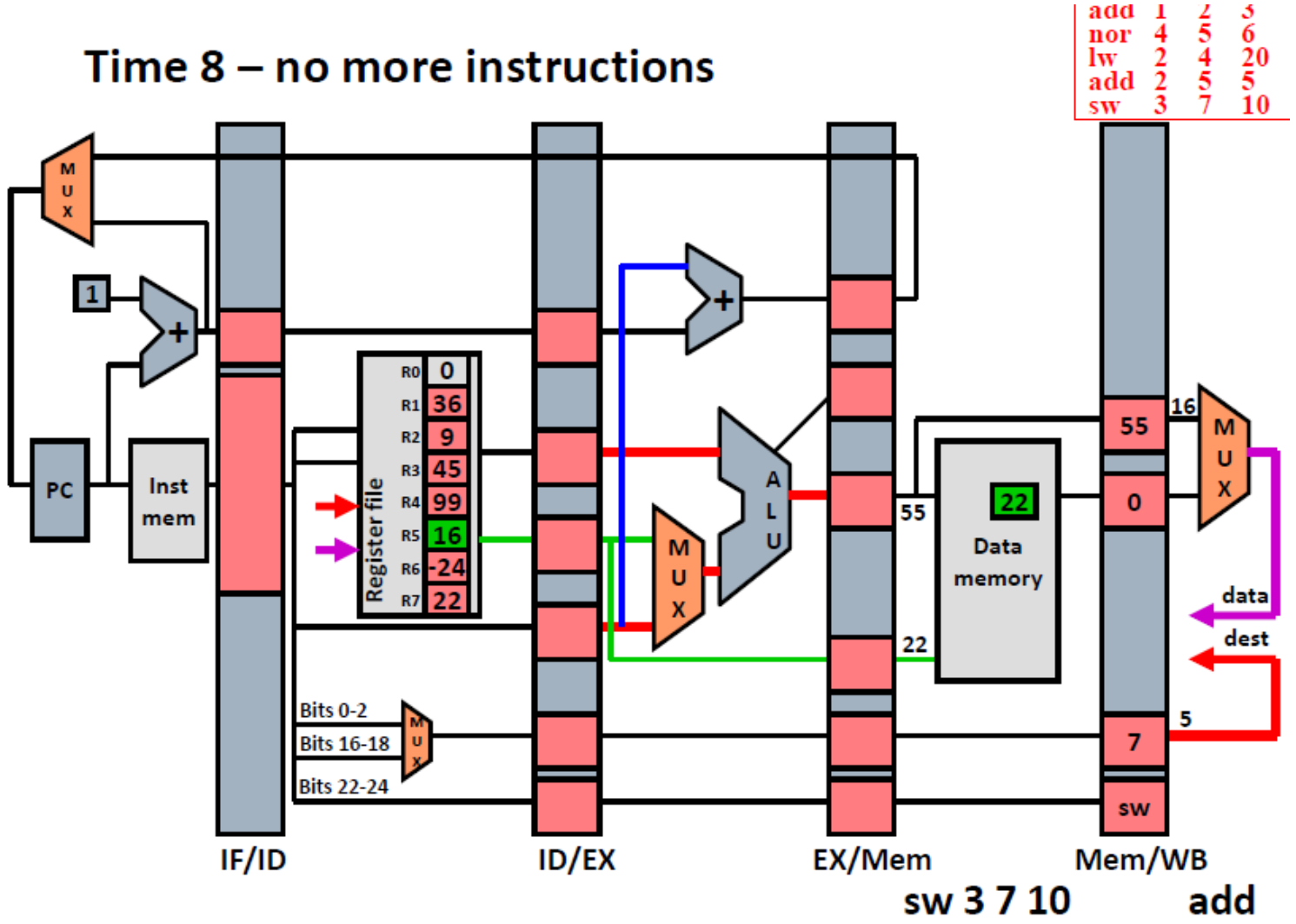
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



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

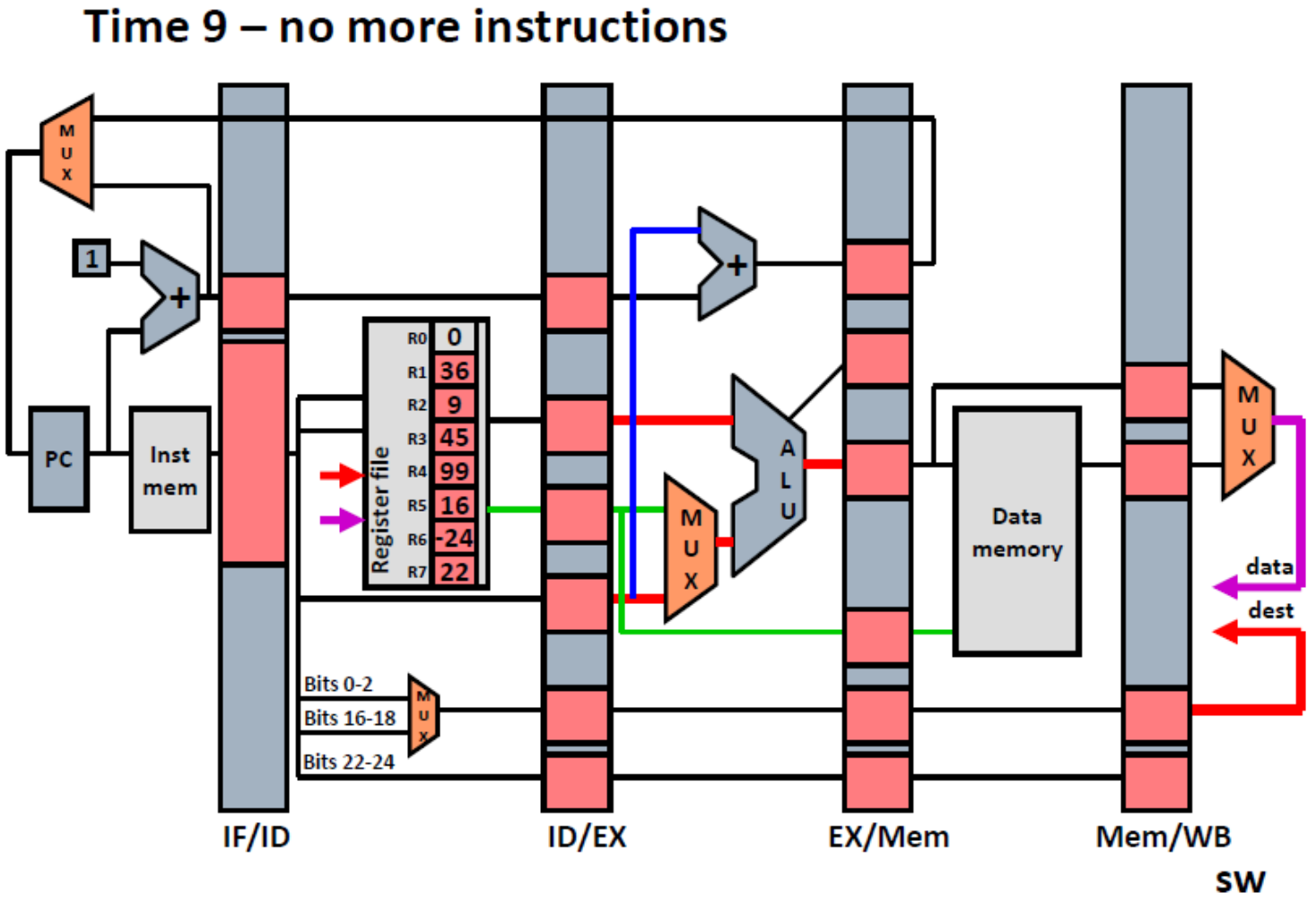
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

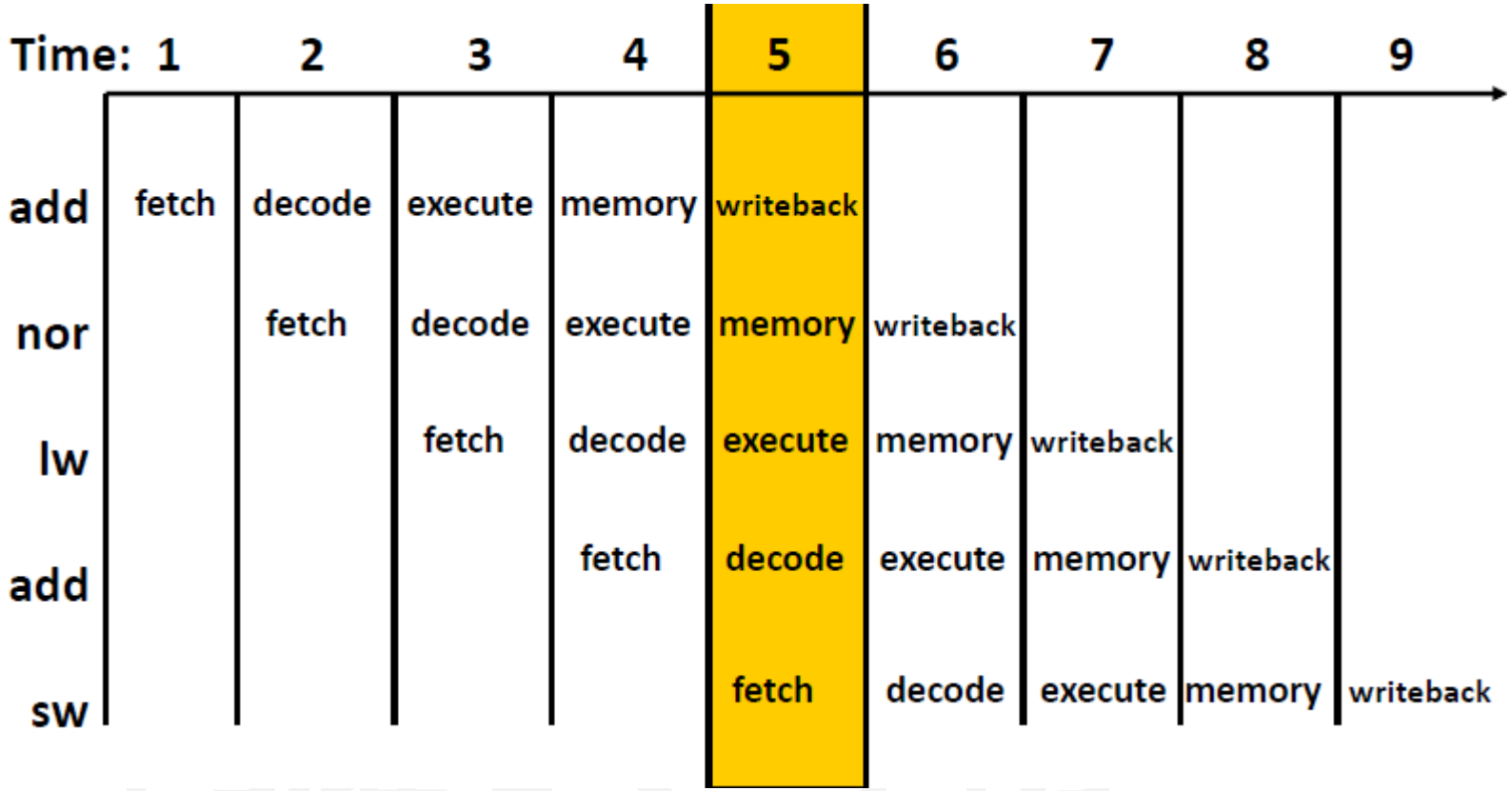
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

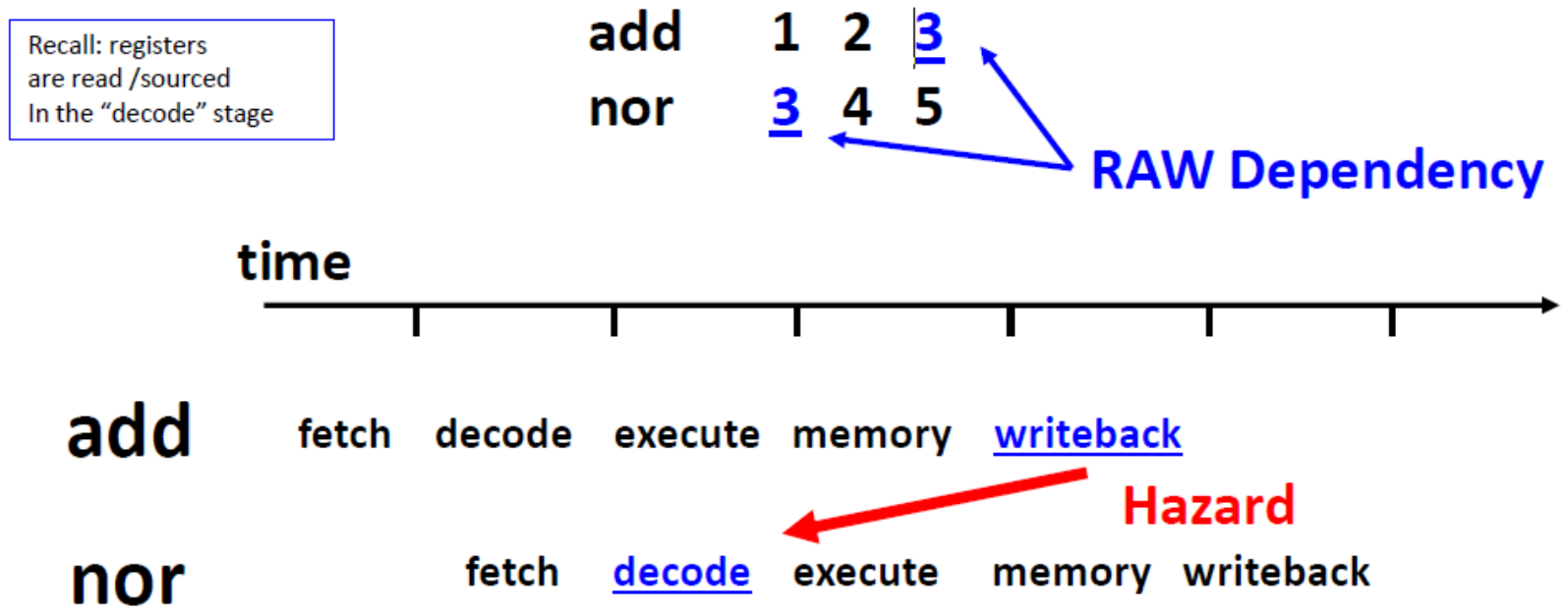
- Data hazards数据冒险：由于流水线中，在第2级读取寄存器、第5级写入寄存器，因而有可能在寄存器值还未写入的时候读取到错误值

- Control hazards控制冒险：当branch指令运行到第4级时，我们才知道此指令是否会改变PC值。在那之前应该取什么指令地址?

- 例外：有时候我们需要暂停指令运行，转向其他任务（可能是OS），然后再恢复指令运行。应当如何确保在正确位置恢复指令运行?

问题1：Data Hazards

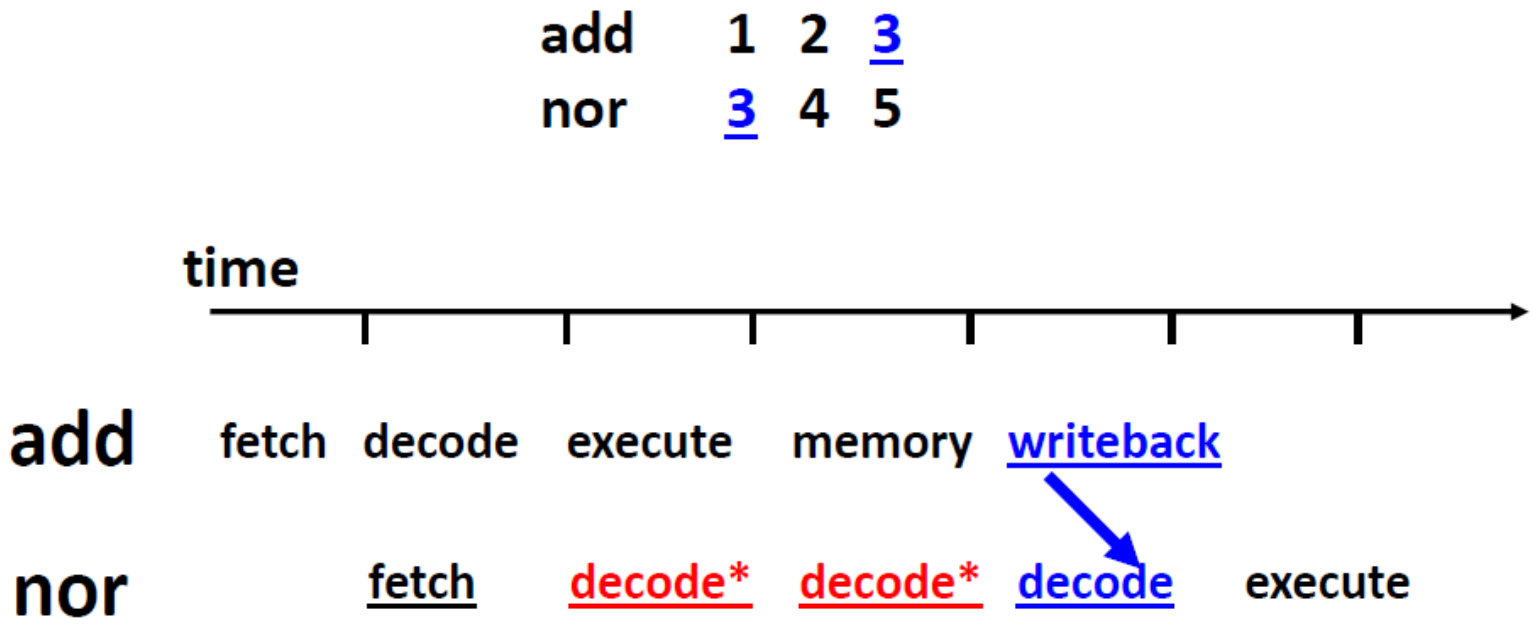
- RAW问题：Read After Write数据冲突



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

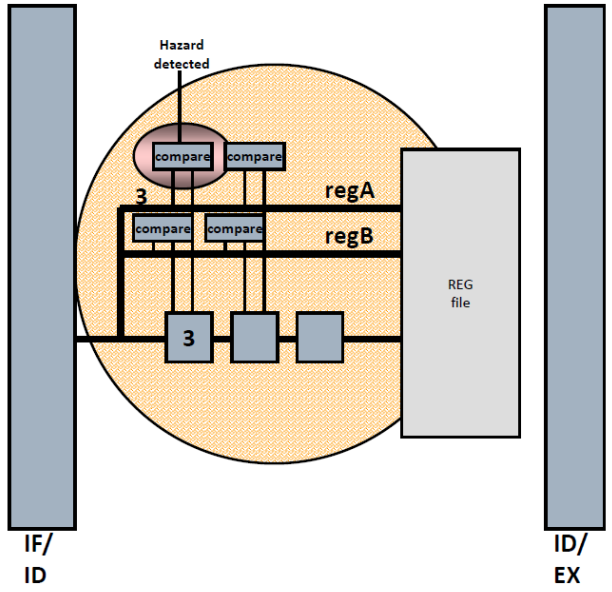


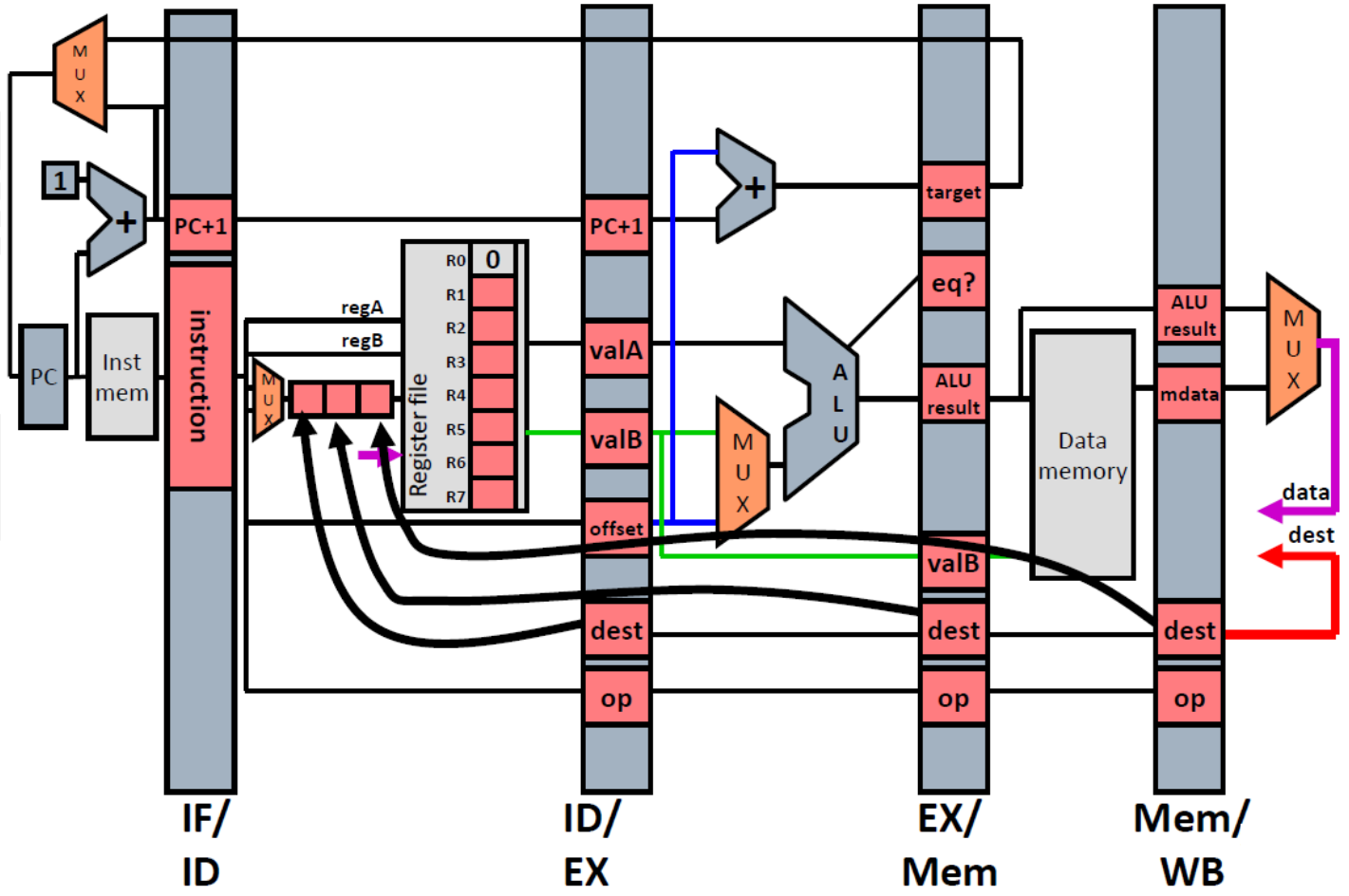
简单解决办法：流水线停顿（Pipeline Stall）

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

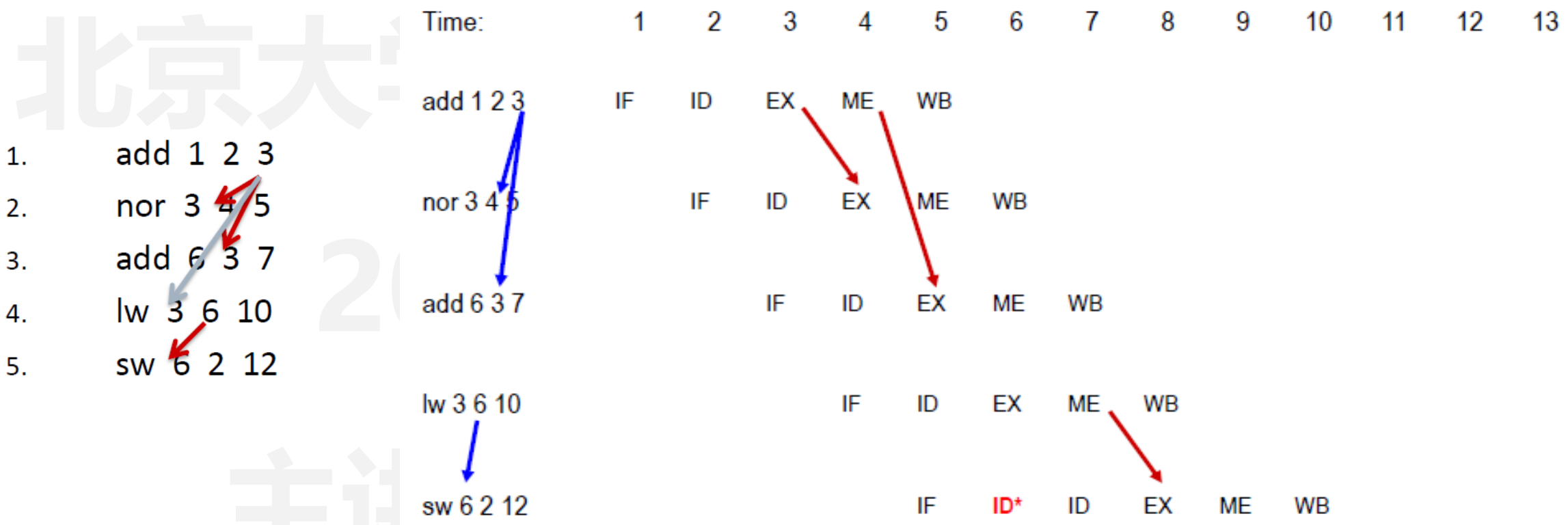




进阶解决办法：Detect and Forward

问题1：Data Hazards

- RAW问题：Read After Write数据冲突



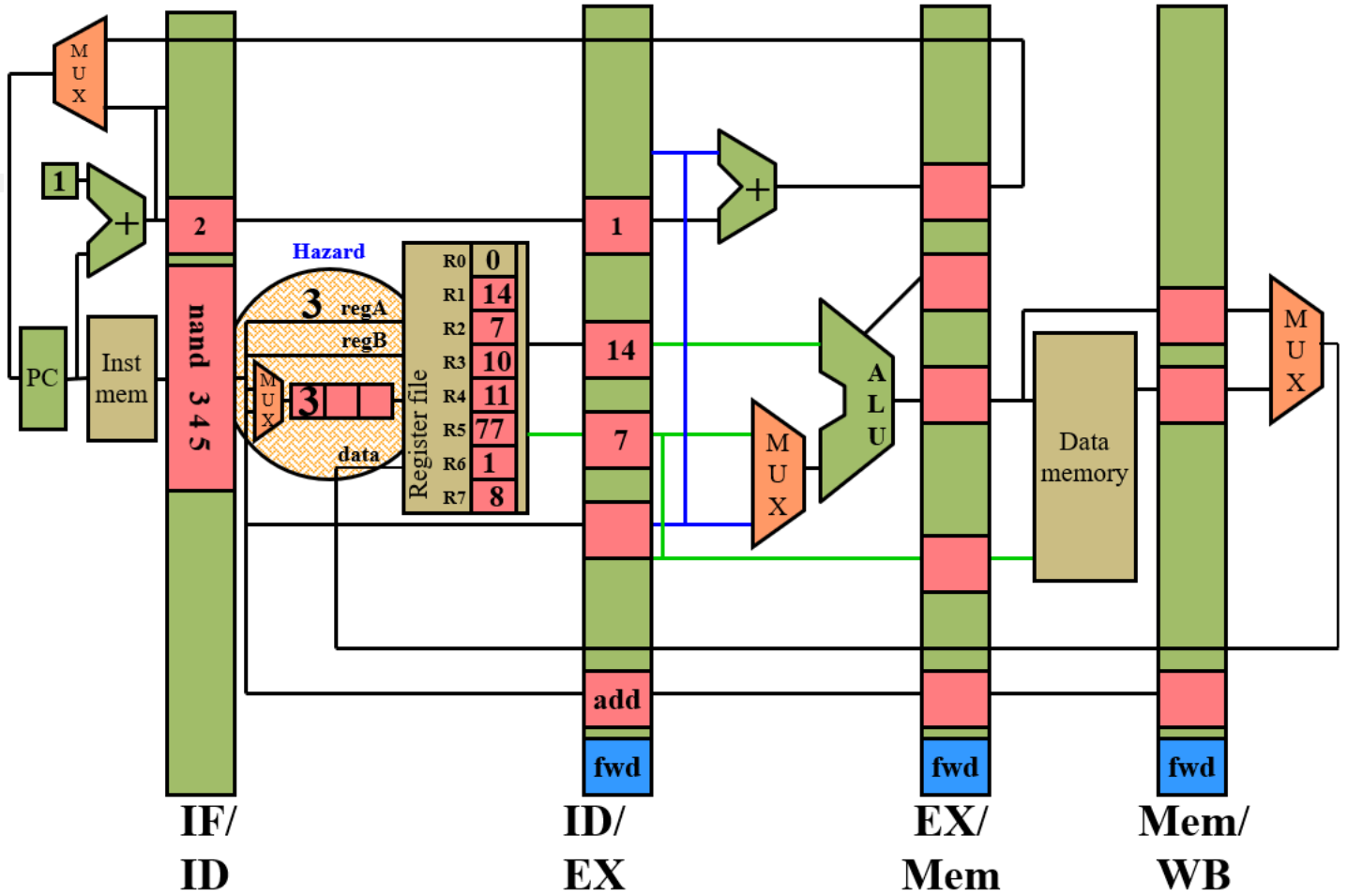
进阶解决办法：Detect and Forward

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

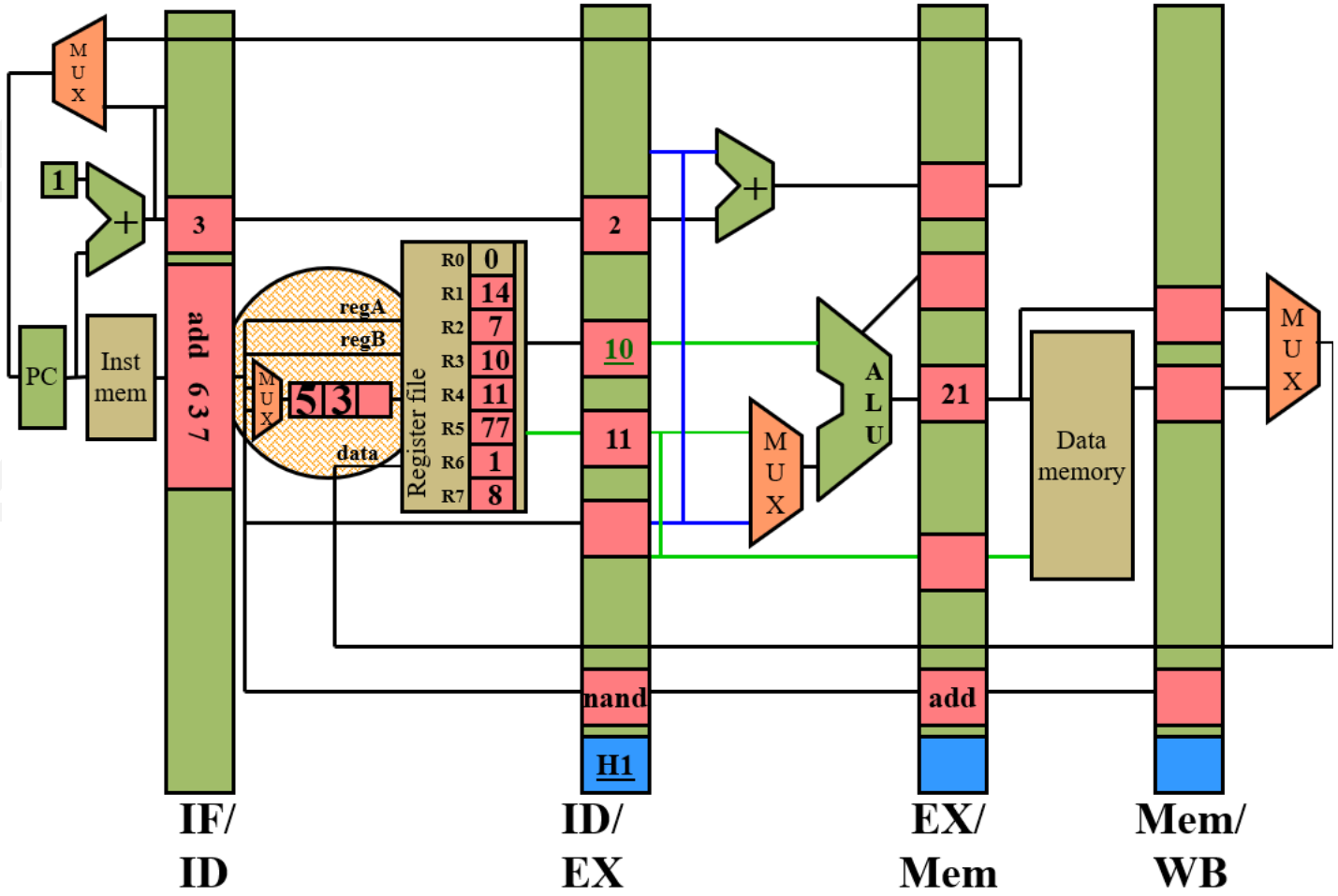


问题1：Data Hazards

- **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

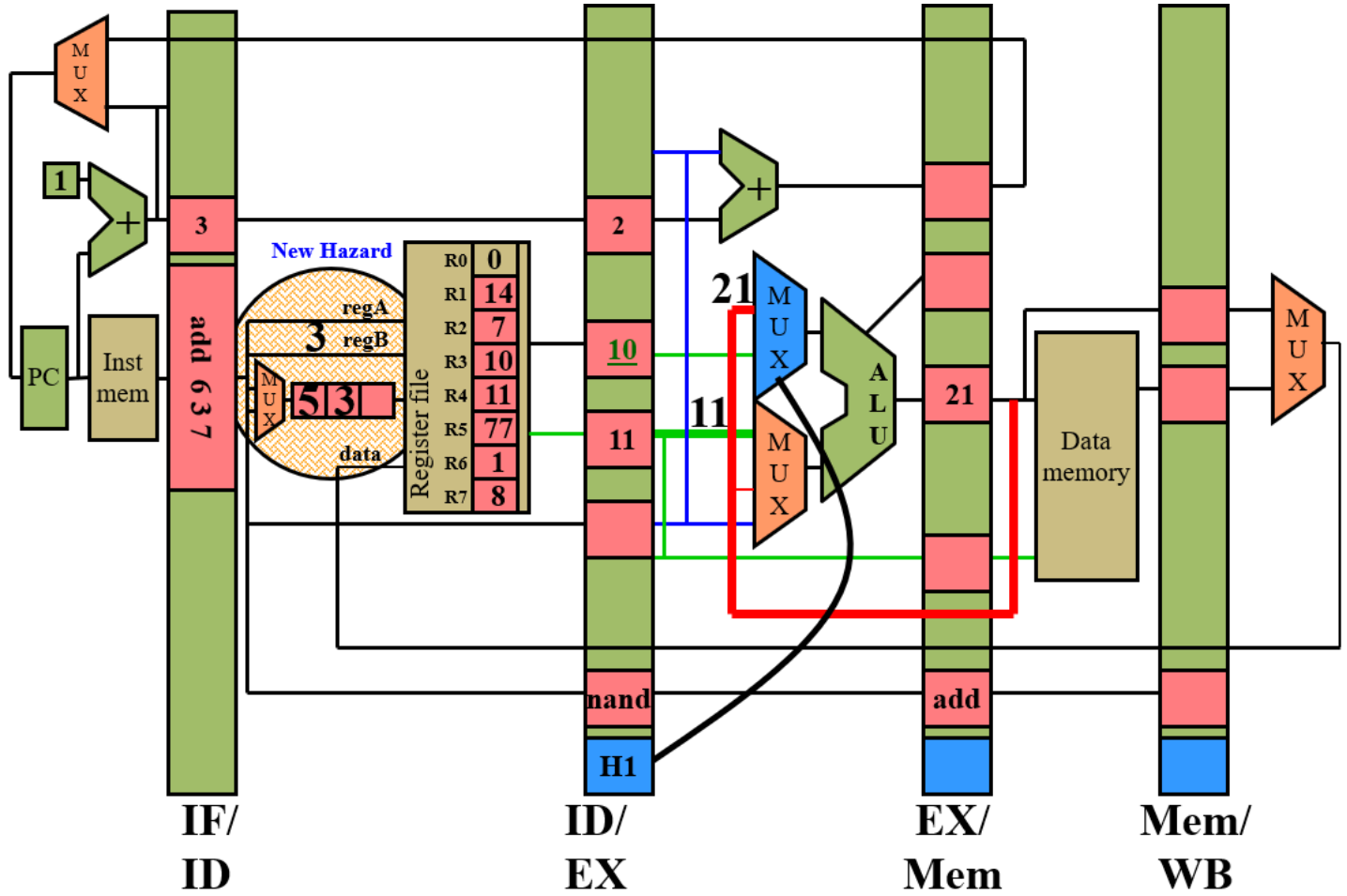


问题1：Data Hazards

- **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）

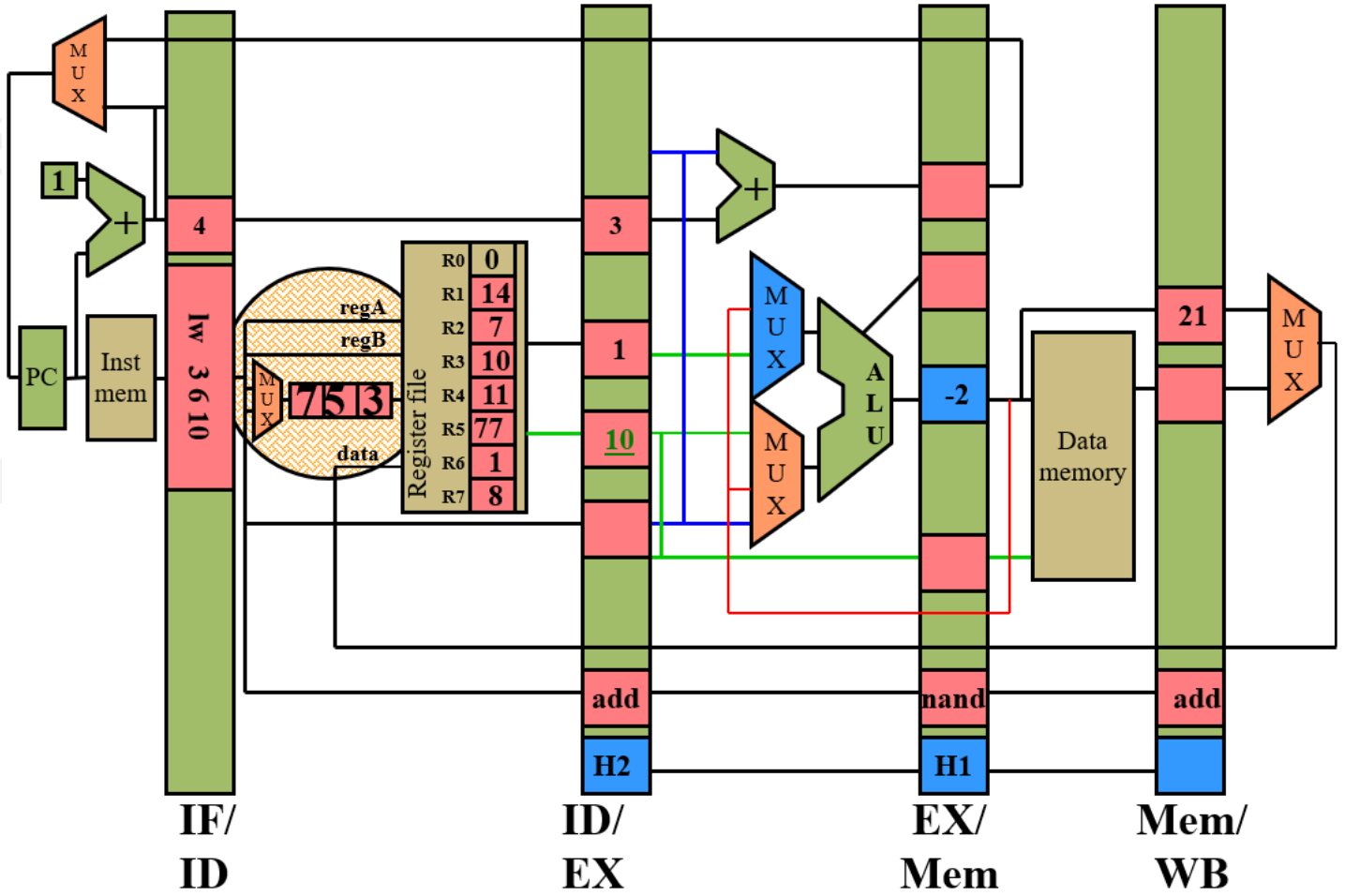


问题1：Data Hazards

- **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

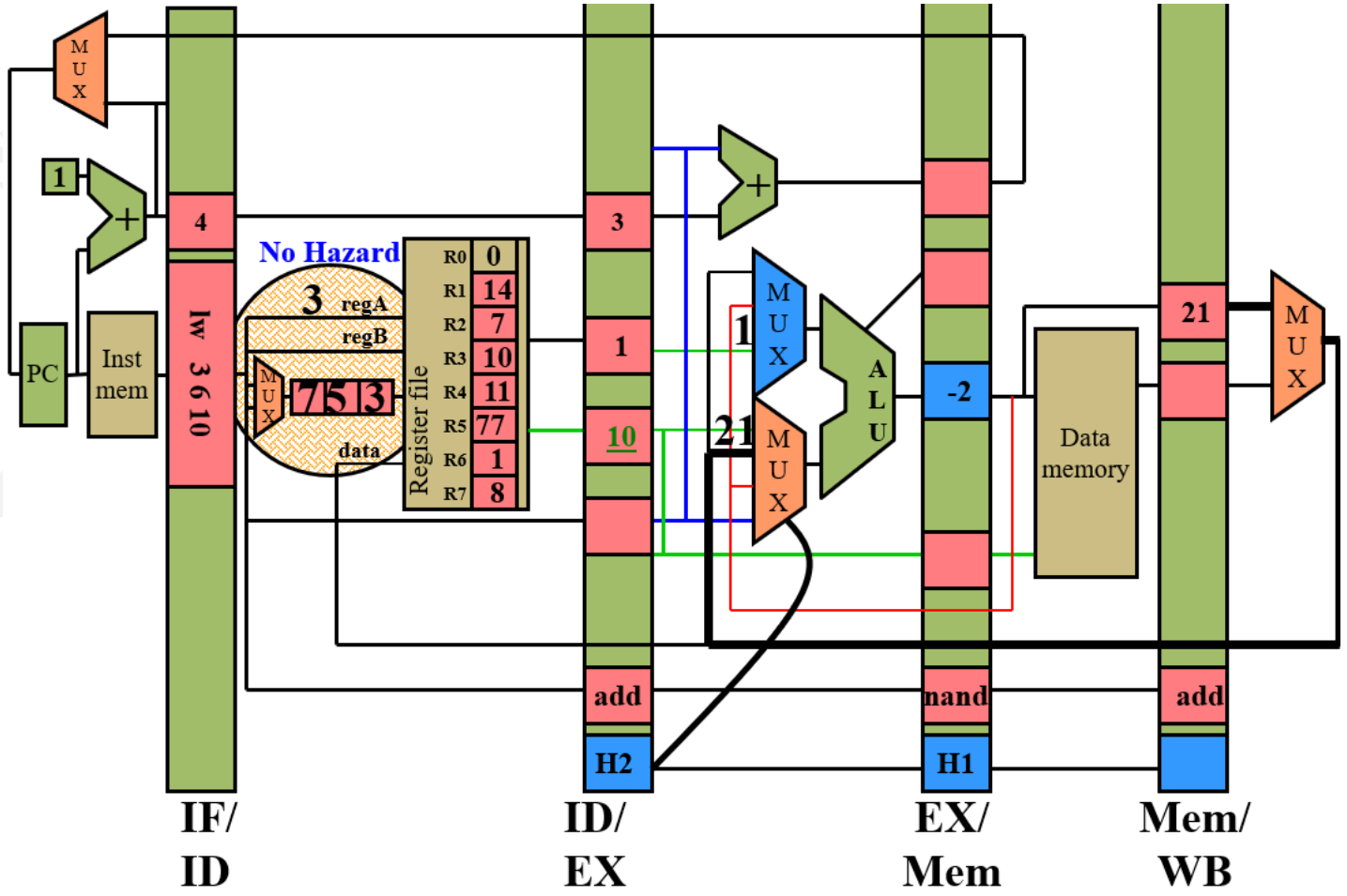


问题1：Data Hazards

- **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

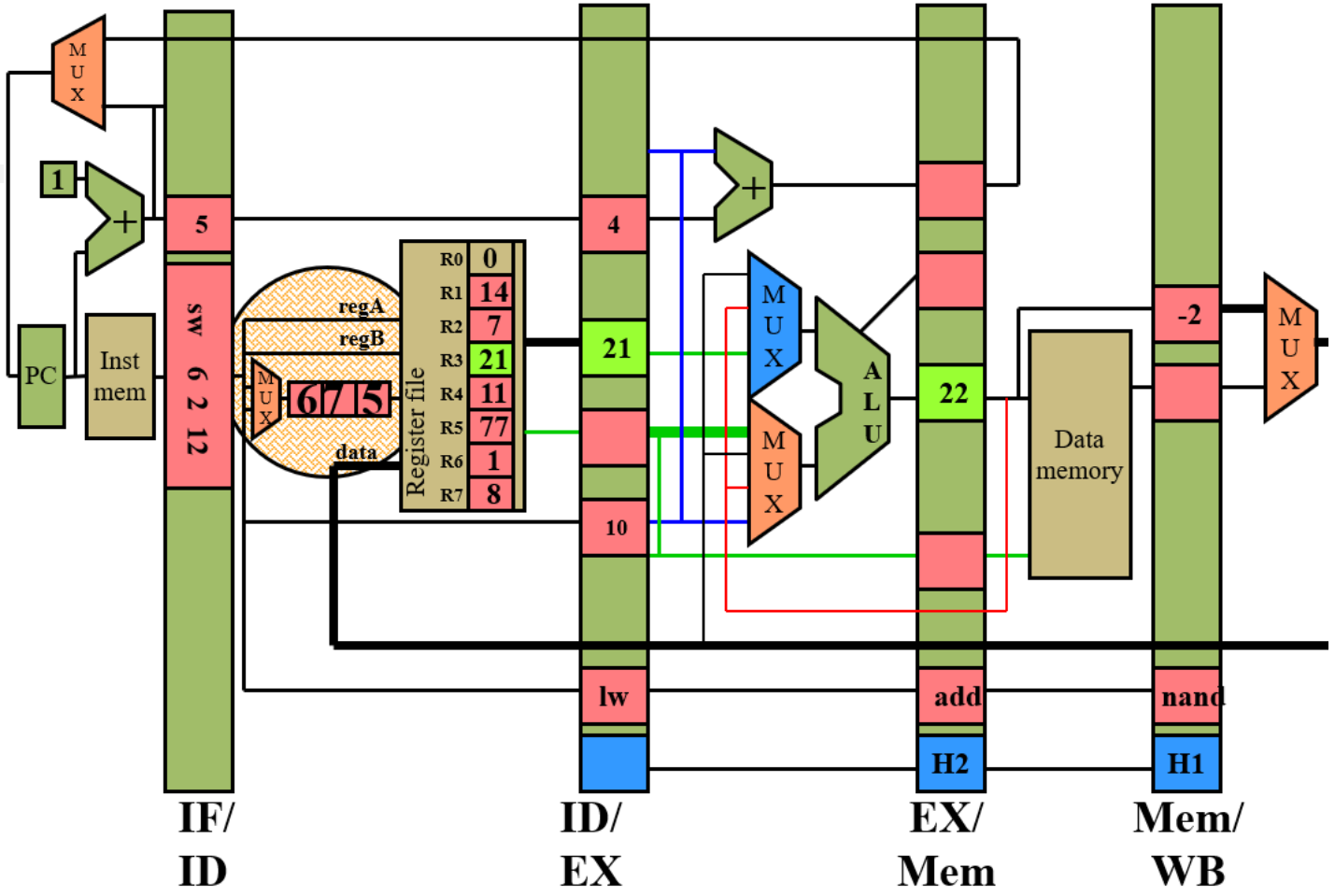


问题1：Data Hazards

- **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

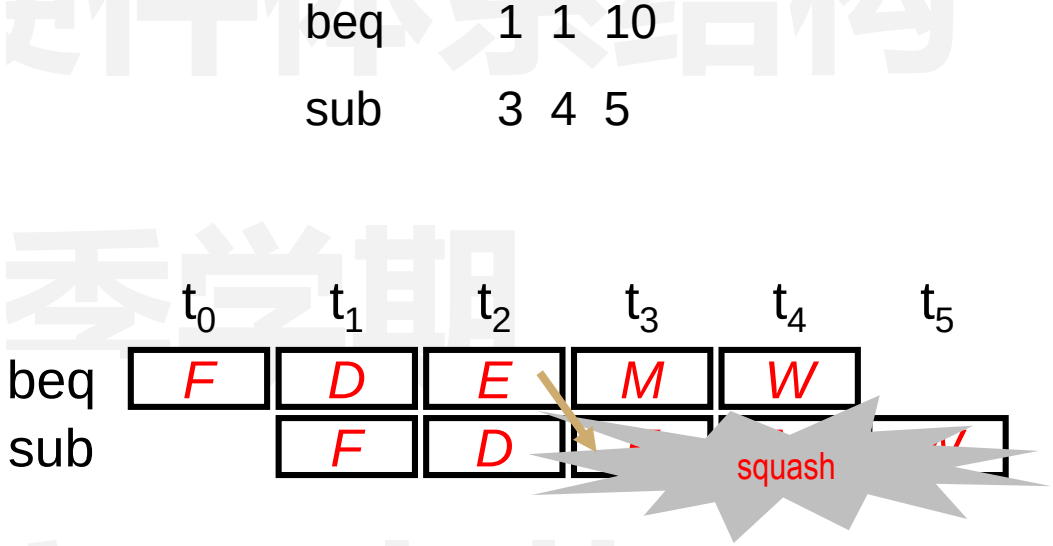
    R1=R4+R5

- 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: 从存储器读取指令
- Decode: 从寄存器读取源操作数
- Execute: 计算目标地址并检测是否符合条件
- Memory: 在符合条件的情况下，**将目标地址送给PC**
- Writeback: 无操作



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)

在分支解决之前，停止获取新的指令

Speculate and squash (dynamic)

- 继续执行分支指令后面的指令，如果出错则丢弃这些指令

问题2：Control Hazards

- **Detection and Stall：检测-停顿机制**

Detection

- 在decode时，检查操作数来判断是否为分支跳转指令

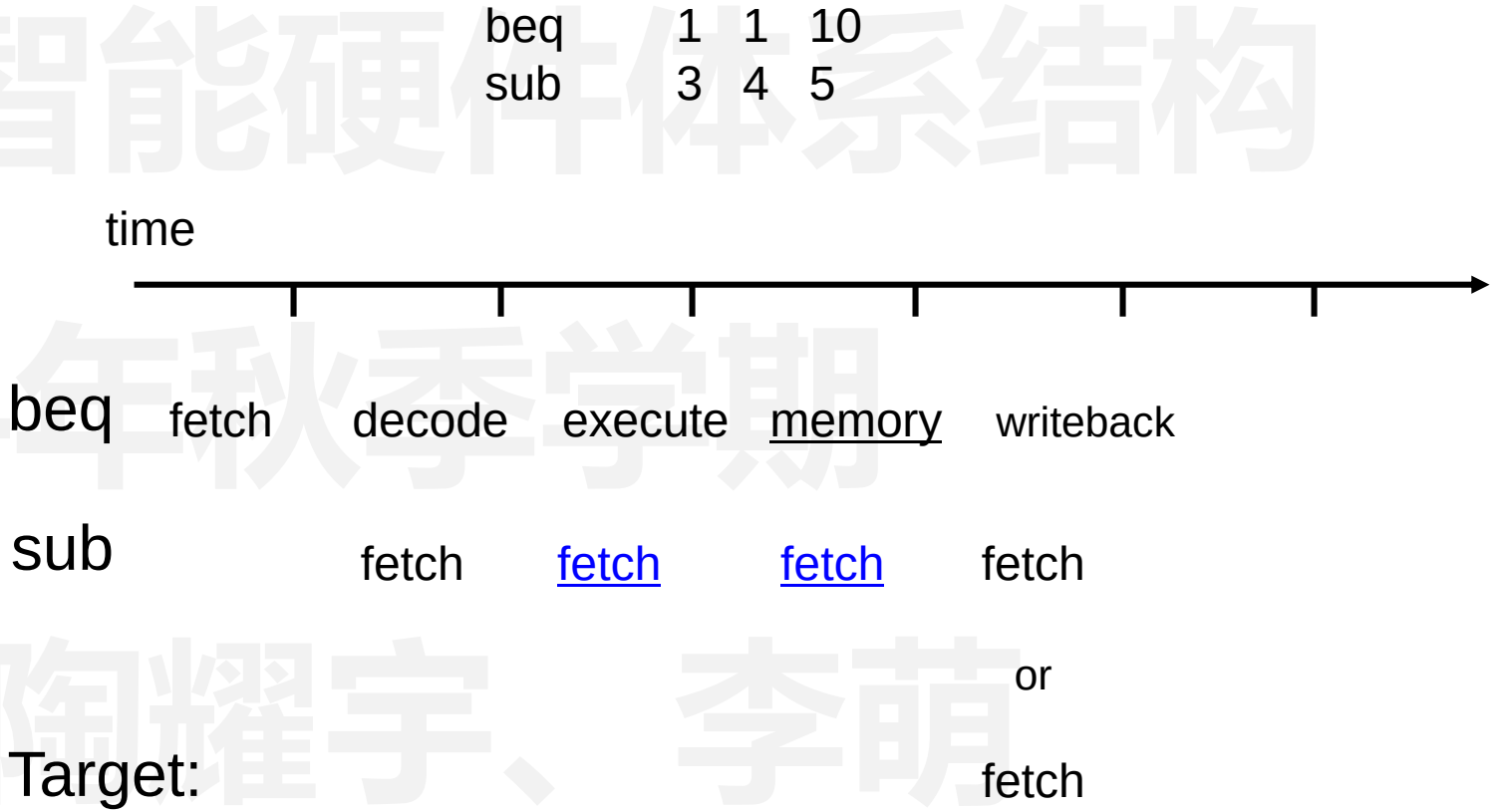
Stall

- 将下一个指令保持在Fetch中
- 将noop 传递给Decode

分支指令会使CPI增加

这些stalls并不全是必要的

- 假设branch没有跳转，完全可以把branch指令当作一个noop空指令，连续读取其后面的指令
- 但上述方法需要保证错误的指令都不会被完整执行



问题2：Control Hazards

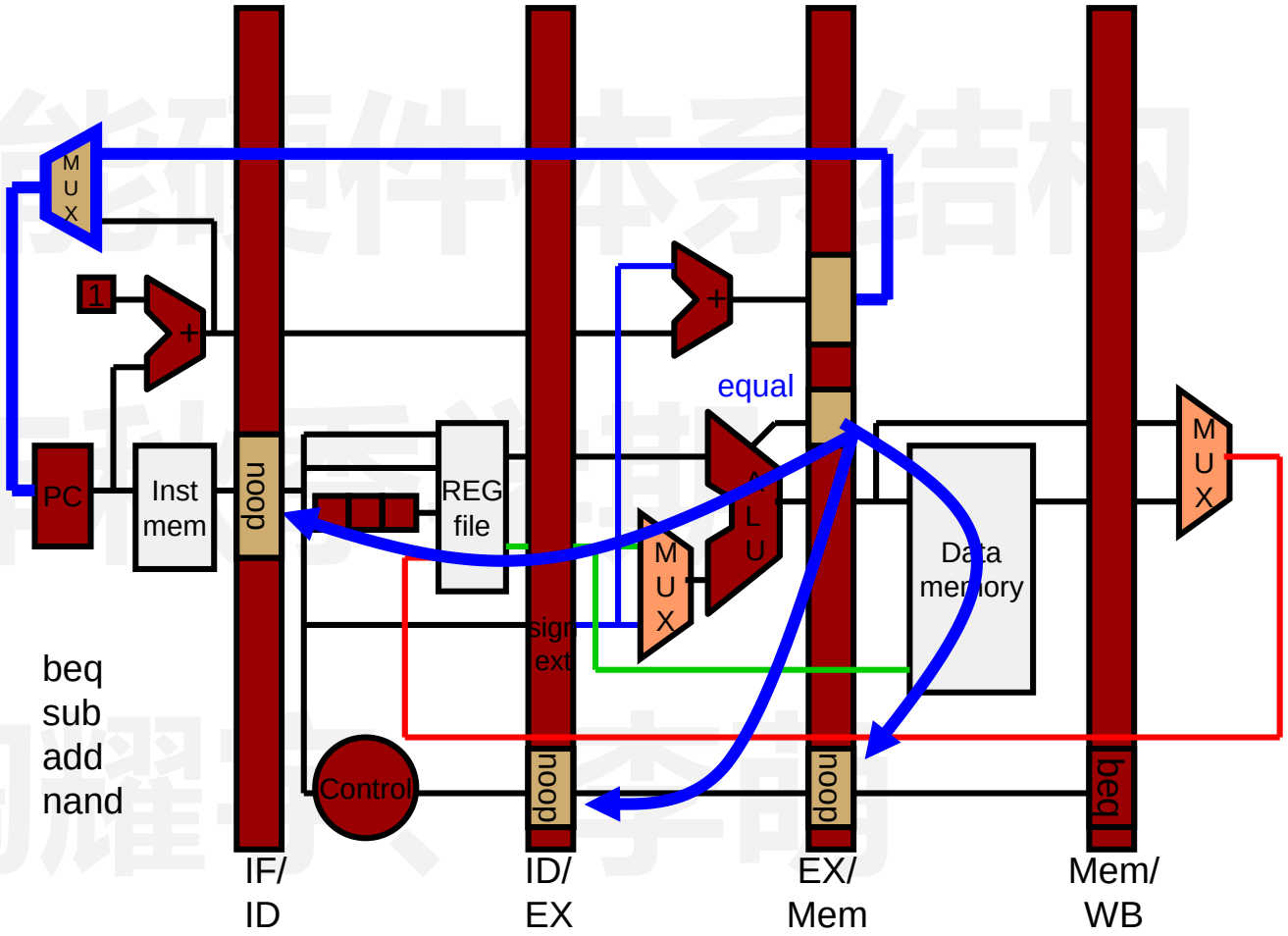
- **Speculate and Squash：投机-制止机制**

Speculate “Not-Taken”

- 假设“分支跳转未执行”

Squash

- 将Fetch，Decode，Execute中的操作数都用noop覆盖
- 将目标指令地址传递给Fetch



问题2：Control Hazards

- **Speculate and Squash：投机-制止机制的问题**

总是假设分支跳转未执行，是否有更策略

- 同时预测分支跳转的方向与目标
- 利用程序中的重复与循环来完成预测

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:

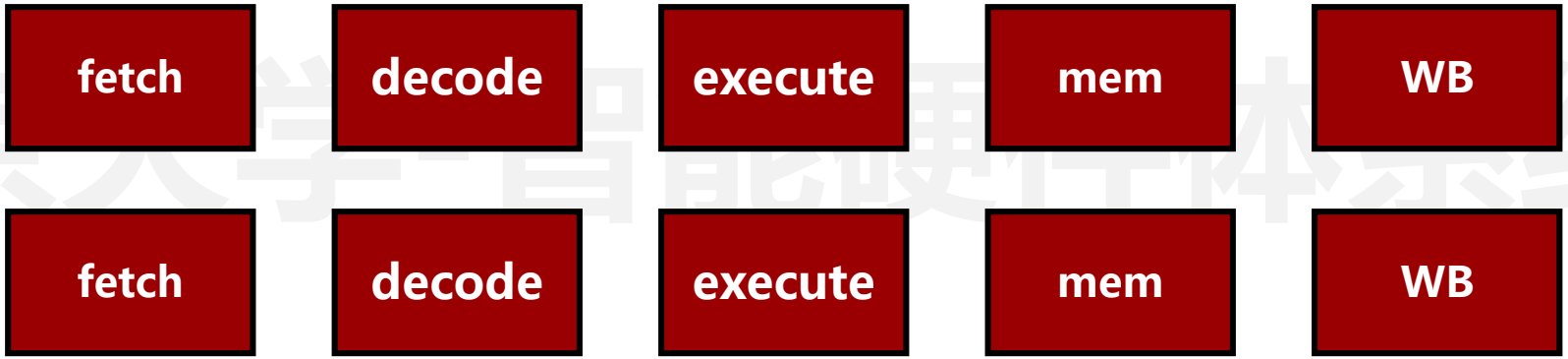
1、标量流水吞吐带来的上界

2、严格顺序流水执行带来的性能损失

Unnecessary stalls

并行度的来源

- 简单并行流水线



更加复杂的冒险检测

- 2X 流水线寄存器用于前馈
- 2X 指令检查
- 2X more destinations (MUXes)
- 需要考虑同一级的多个指令是否独立的问题

Superscalar：超标量的概念

- Instruction-level parallelism

指令并行度

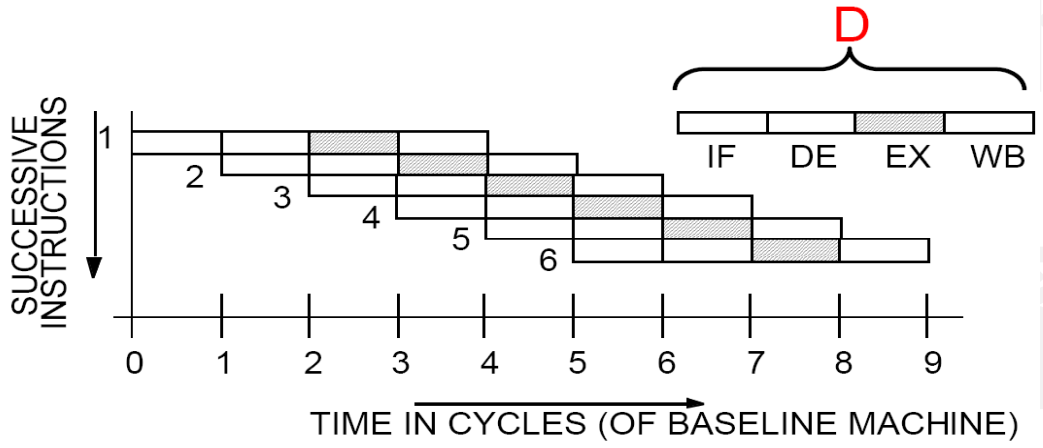
同时执行的指令个数

Scalar Pipeline (baseline)

Instruction Parallelism = D

Operation Latency = 1

Peak IPC = 1



Peak IPC

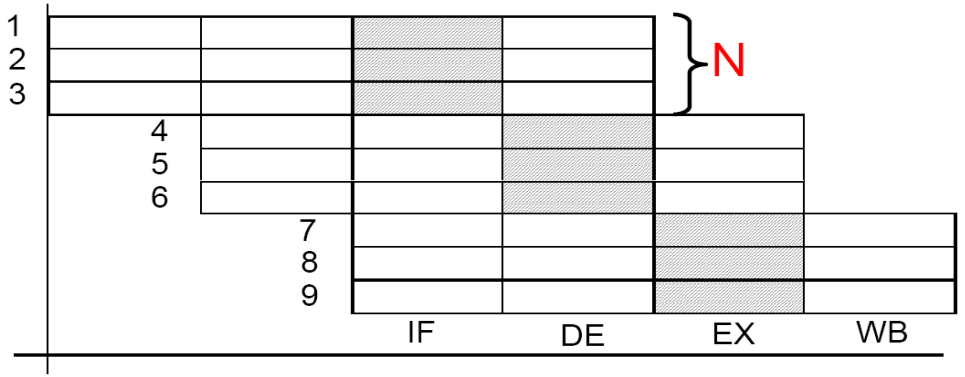
可以在一个时钟周期内运行的指令的最大可持续数量

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

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
`addf f0,f1,f2`	F	D	E+	E+	E+	W										
`mulf f2,f3,f2`		F	D	d*	d*	E*	E*	E*	E*	E*	W					
`subf f0,f1,f4`			F	p*	p*	D	E+	E+	E+	W						

在cycle 4出现的问题

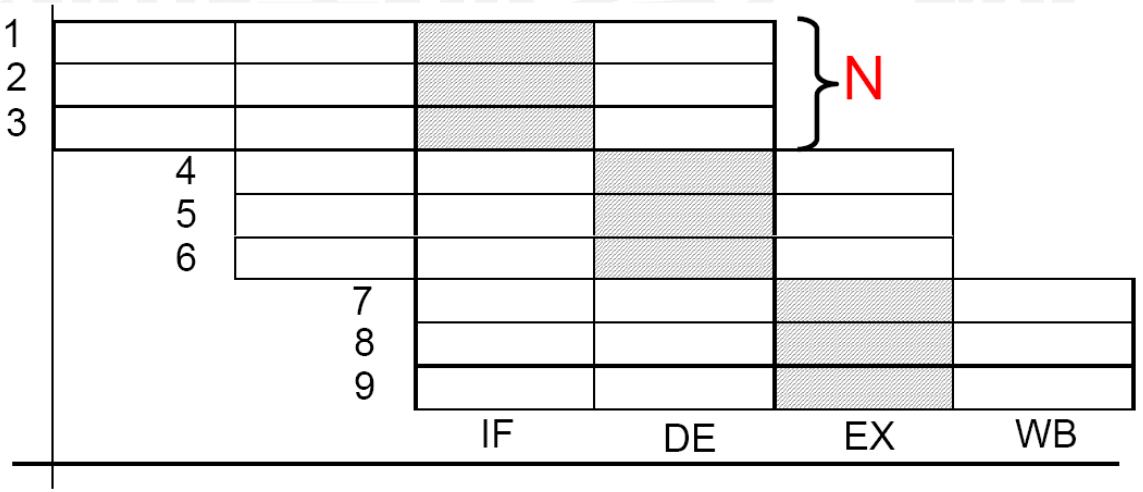
- `mulf` 因为**RAW hazard**而需要stall
  - 这里的stall是必要的
- `subf` 因**pipeline hazard**而需要stall
  - `subf` 不能进入Decode， 因为`mulf`在那里`stall`
  - subf的stall是不必要的

所以为什么不让subf在cycle4就进入Decode呢?

Out-Of-Order：乱序执行的概念

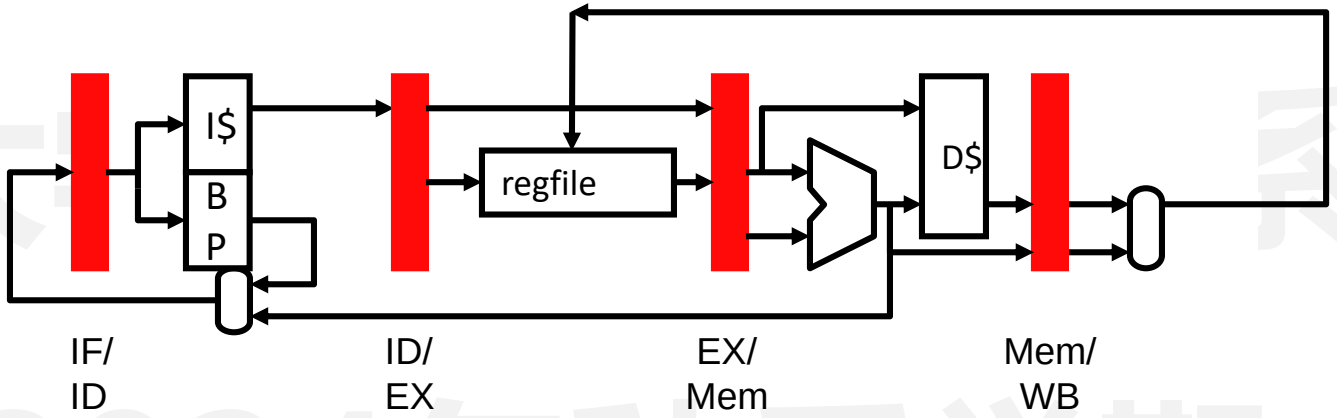
- **Instruction-level parallelism**

- 如果并行度超过一定的阈值，顺序执行流水线的CPI将会严重下降
  - *当相关指令的平均距接近为NxM*
- 前馈不再有效
  - *因为频繁的stall，流水线将会一直是空闲的*



Out-Of-Order：乱序执行的概念

- The Problem With In-Order Pipelines



- **In-order pipeline**
  - **Structural hazard**: 流水线每级只有一个指令寄存器
    - 每周期每级只有一个指令（除非流水线复制多份）
    - 后级指令不能越过前级指令 without “clobbering” it
- **Out-of-order pipeline**
  - 通过去除**structural hazard**的方法来实现指令的跳跃执行

Out-Of-Order：乱序执行的概念

- 乱序执行完全在硬件实现

- **动态调度**
  - **完全在硬件中**
  - **也称为“乱序执行”（OoO）**
- **将许多指令提取到指令窗口中**
  - **使用分支预测推测过去（多个）分支**
  - **在分支错误预测时刷新流水线**
- **重命名以避免错误的依赖关系（WAW 和 WAR）**
- **尽可能快的执行指令**
  - **寄存器依赖项是已知的**
  - **处理内存依赖关系更棘手（稍后会详细介绍）**
- **按顺序提交指令**
  - **任何奇怪的事情都会在 commit 之前发生，只需 flush pipeline**
- **当前机器：100+ 指令调度窗口**

乱序执行

以非顺序执行指令…

减少 RAW 停顿

提高流水线和功能单元 （FU） 利用率

最初的动机是提高 FP 单位的利用率

为并行执行行 （ILP） 提供更多机会

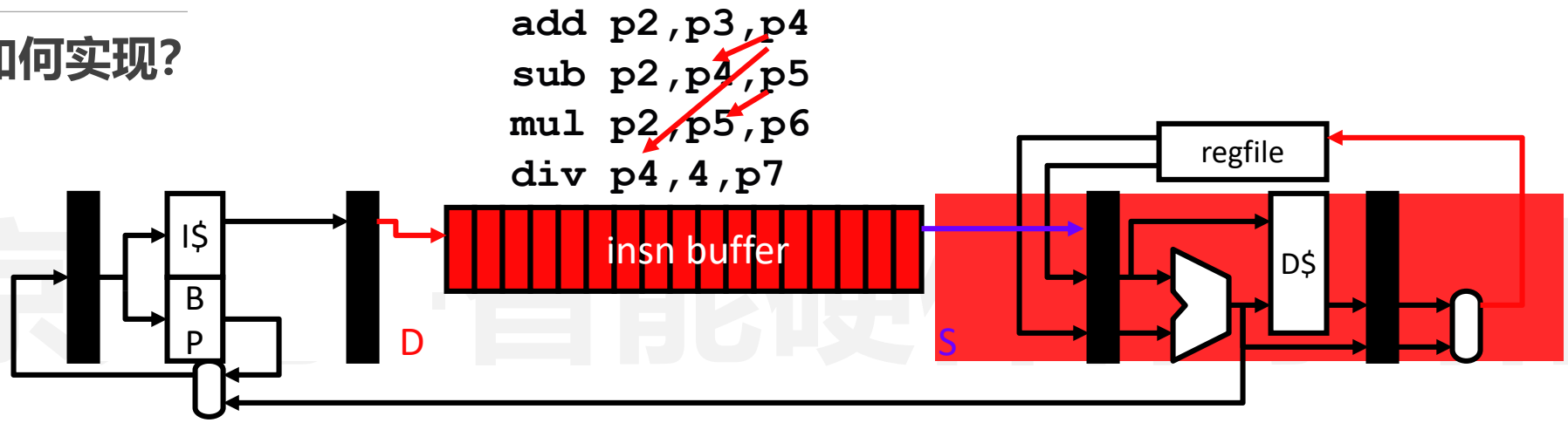
不按顺序-可以并行

…但要让它看起来像顺序执行

重要但困难，接下来会讲到

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
```

- 指令获取/解码/重命名后进入*Instruction Buffer*
  - Also called “instruction window” or “instruction scheduler”
- 指令每周期检查是否就绪
  - 就绪后执行

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**

  - 依赖项独立于硬件而存在
    - 如果 Inst #1000 需要 Inst #1 的结果，则存在依赖关系
    - *只有当硬件必须处理它时，它才是一种危险*
      - 因此，在我们的流水线机器中，我们只担心依赖指令之间没有两条指令的“buffer”.

数据依赖

- True/False Data Dependencies

- 真数据依赖
- RAW – Read after Write

R1=R2+R3

R4=R1+R5

- True dependencies prevent reordering
  - (Mostly) unavoidable

- 假数据依赖
  - WAW – Write after Write

R1=R2+R3

R1=R4+R5

  - WAR – Write after Read

R2=R1+R3

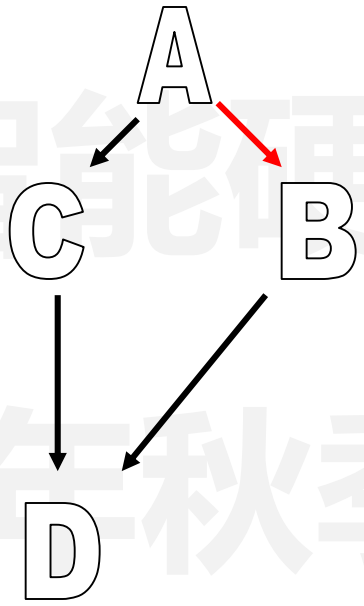
R1=R4+R5

- False dependencies prevent reordering
  - 通过renaming可以被消除

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
```

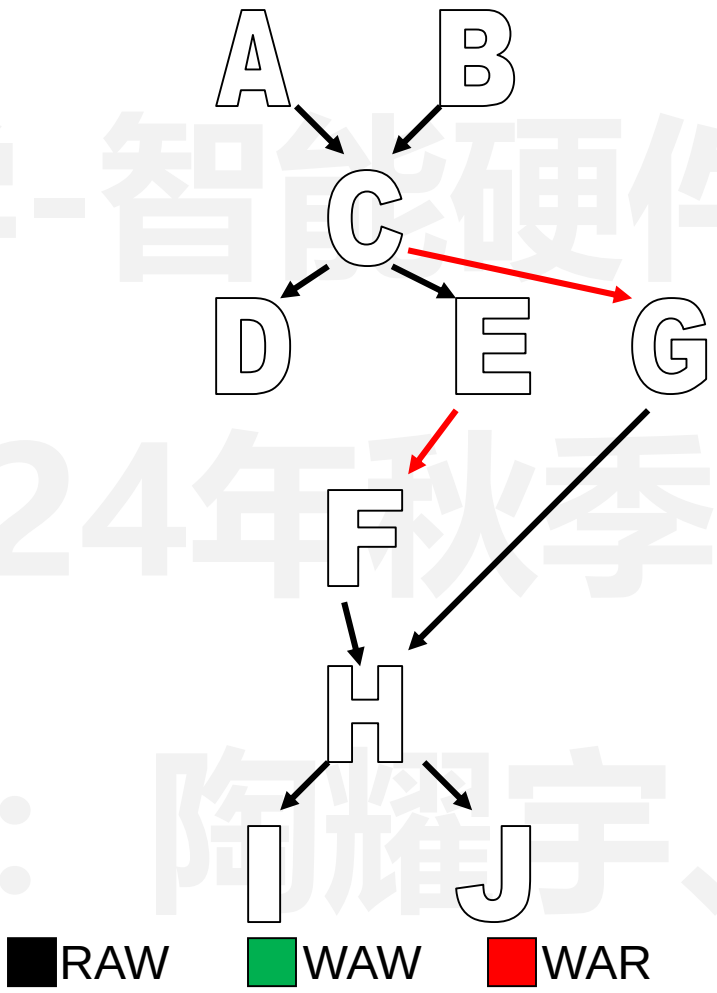


RAW WAW WAR

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
```



- 从逻辑上讲，F-J 没有理由依赖于 A-E. So…..
  - ABFG
  - CH
  - DEIJ
  - Should be possible.
- 但这会导致 C 或 H 具有错误的 reg 输入
- 如何解决?
  - Remember, the dependency is really on the *name* of the register
  - **So… 改变寄存器名称即可!**

触发器重命名

- 触发器Register重命名概念

  - 寄存器名称是任意的
  - 寄存器名称只需要在写入之间保持一致

R1= .....

.... = R1 ....

....= ... R1

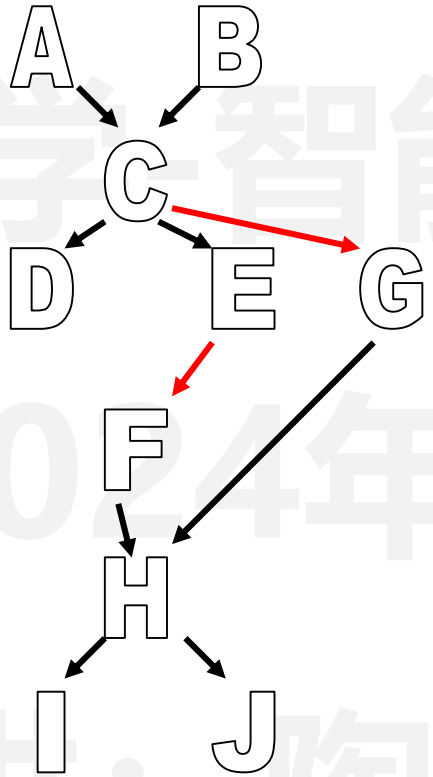
R1= ....

R1中的值是有效的，在写入到最后一次读取期间

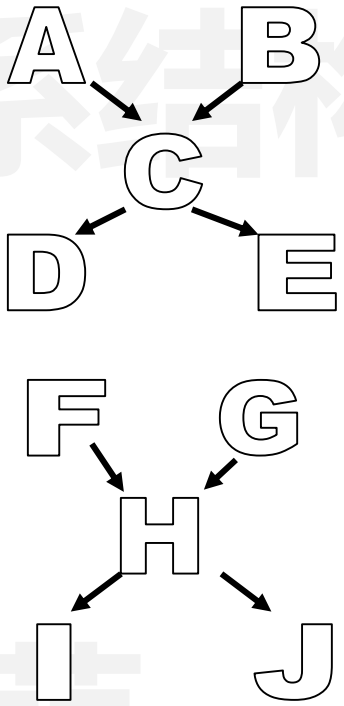
触发器重命名

- 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

- 触发器Register重命名机制

- 每次写入架构寄存器时，我们都会将其分配给物理寄存器
  - 在再次写入架构寄存器之前，我们将继续将其转换为物理寄存器号保持 RAW 依赖项不变
- 很简单，让我们看一个例子:
  - 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
```