北京大学
PEKING UNIVERSITY

智能硬件体系结构

第六讲：流水线架构

主讲：陶耀宇、李萌

2025年秋季

注意事项

北京大学 PEKING UNIVERSITY

• 课程作业情况

- **第1次作业已于10.17日11:59发布，截止日期：11月7号晚11:59**
  - 3次作业可以使用总计7个Late day
  - Late Day耗尽后，每晚交1天扣除20%当次作业分数
- **第1次lab时间：10月17日11:59发布 - 11月10日晚11:59**
  - **3个子任务（60%+30%+10%）**
- 第2次lab时间：11月10日-12月7日

指令集的分类2 – 计算指令

北京大学 PEKING UNIVERSITY

- 利用ALU来完成实际计算任务

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

指令集的分类2 – 计算指令

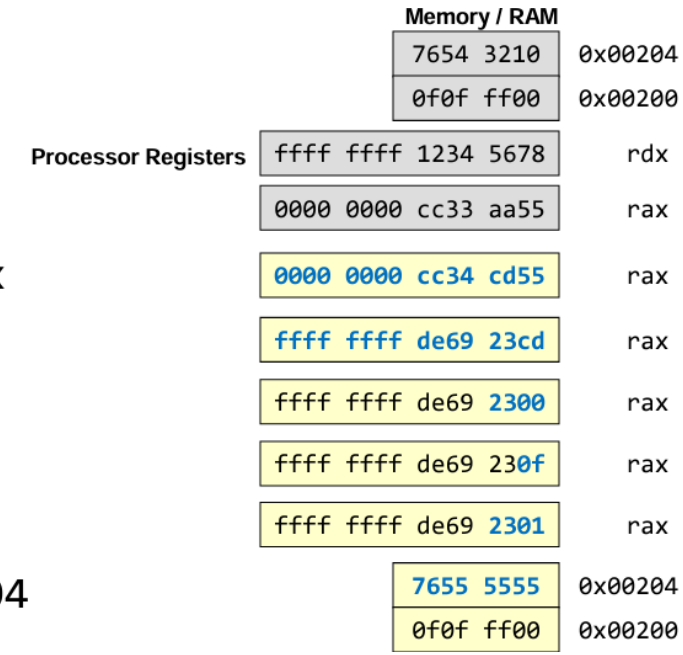
- 利用ALU来完成实际计算任务

- 基于给定数据尺寸执行算术/逻辑操作
- 限制：两个操作数都不能是存储器

- Format
  - add[b,w,l,q] src2, src1/dst
  - Example 1: addq %rbx, %rax (%rax += %rbx)
  - Example 2: subq %rbx, %rax (%rax -= %rbx)

Work from right->left->right

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



指令集的分类2 – 计算指令：实际案例

- 计算指令配合传输指令完成一个代码的编译过程

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

Original Code

```
f1:

    movl    (%esi), %eax
    addl    (%edi,%edx,4), %eax


    ret
```

Compiler Output

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

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

Original Code

```
f1:

    addb    $1, (%edi)
    subl    %esi, 4(%edi)


    ret
```

Compiler Output

指令集的分类3 – 控制指令

- 控制指令地址跳跃

适用于if、case判断语句以及for、while等循环语句等等

将会在后续分支预测等内容深入讲解


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

;Pretend there is a lot of code inbetween here.

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

Jump to address 497000

then continue the code.

指令集的分类4 – 系统指令

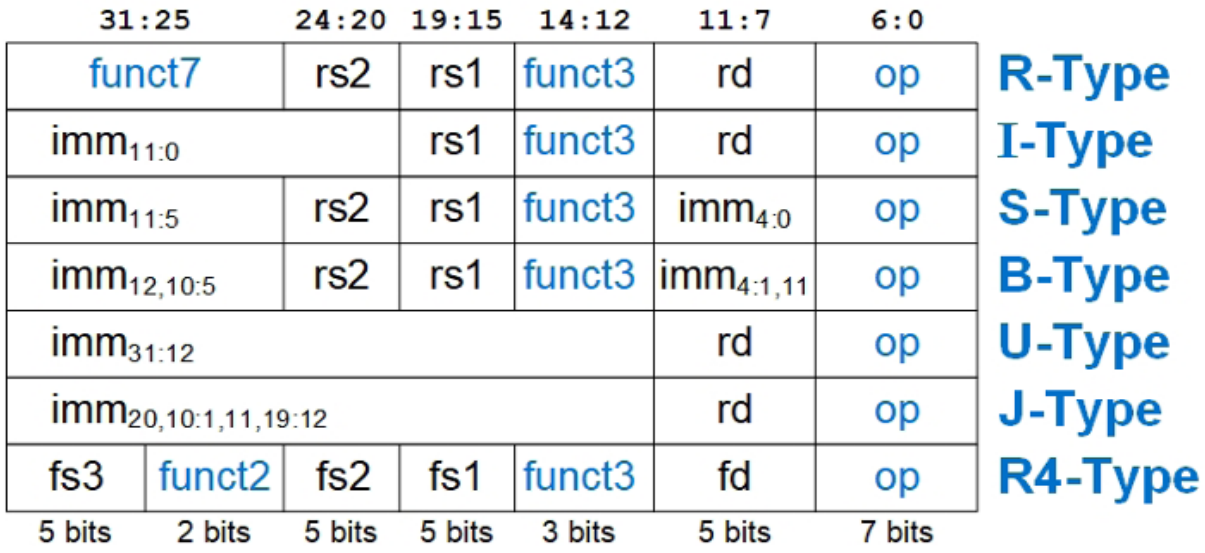
- 与上层操作系统OS对接，例如OS可更改register内容等

编译器设计内容

主讲：陶耀宇、李萌

代表性指令集： RISC-V一种典型的RISC指令

- 完全开源，扩展性较好，指令种类多



31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	R-Type
$imm_{11:0}$		rs1	funct3	rd	op	I-Type
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	S-Type
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	B-Type
$imm_{31:12}$				rd	op	U-Type
$imm_{20,10:1,11,19:12}$				rd	op	J-Type
fs3	funct2	fs2	fs1	funct3	fd	op	R4-Type
5 bits	2 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

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

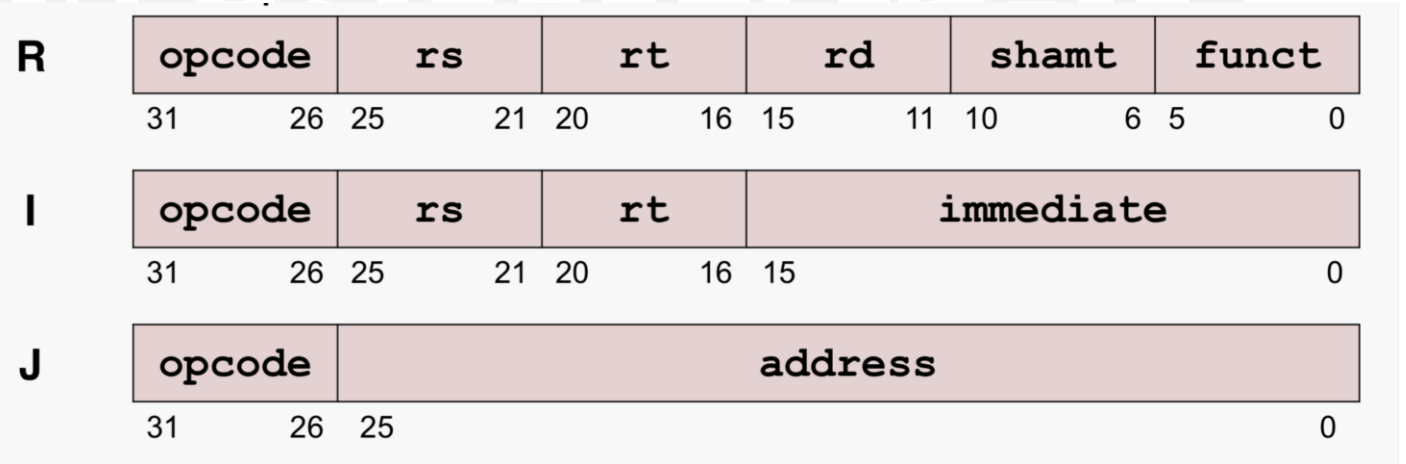
- `imm`: signed immediate in $imm_{11:0}$
- `uimm`: 5-bit unsigned immediate in $imm_{4:0}$
- `upimm`: 20 upper bits of a 32-bit immediate, in $imm_{31:12}$
- `Address`: memory address: rs1 + SignExt($imm_{11:0}$)
- `[Address]`: data at memory location Address
- `BTA`: branch target address: PC + SignExt({$imm_{12:1}$,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

代表性指令集： MIPS一种典型的RISC指令

- 指令长度固定，相对简单（单周期指令）
  - 3种CPU指令， 都是32比特对齐words
  - I-type (Instruction)
  - J-type (Jump)
  - R-type (Register)

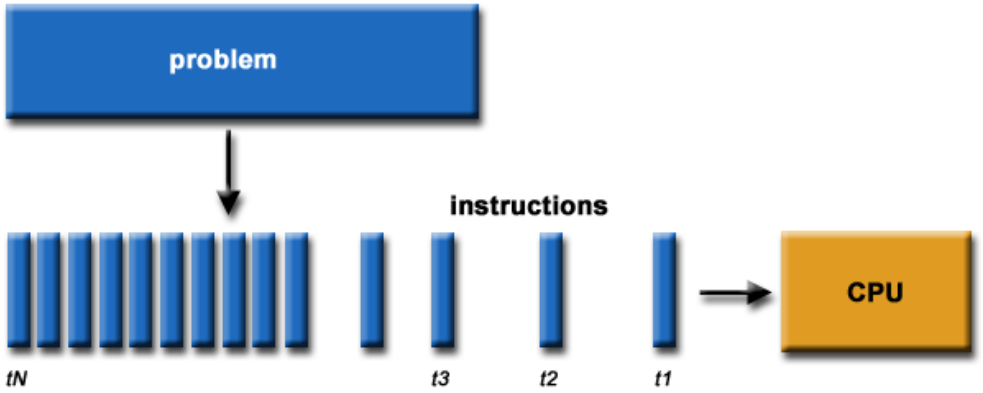
•Opcode

- 6-bit operation code
- There are 3 different register specifiers:
  - RD - 5-bit destination register
  - RS - 5-bit source register
  - RT - 5-bit target register

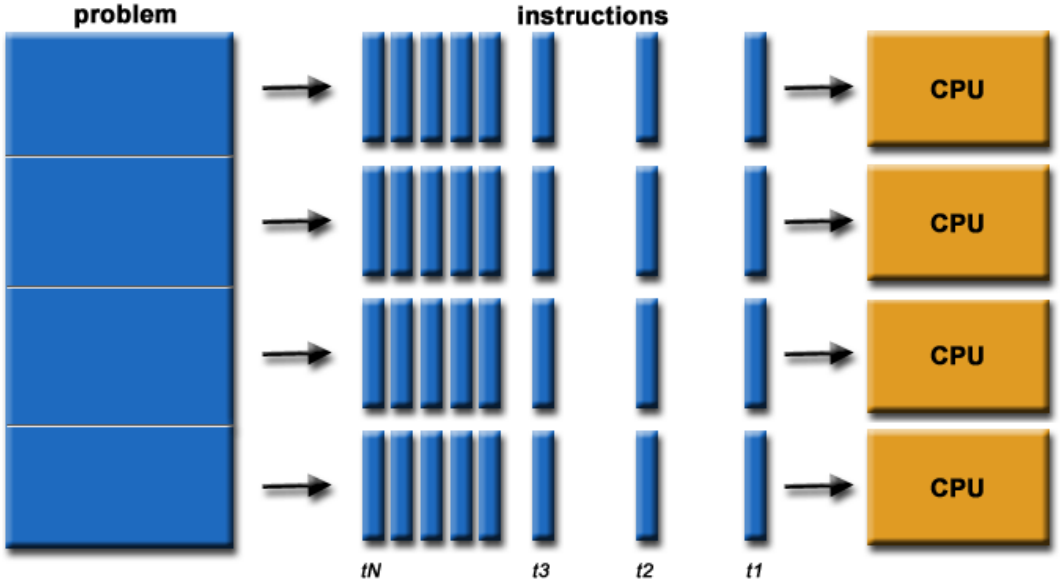


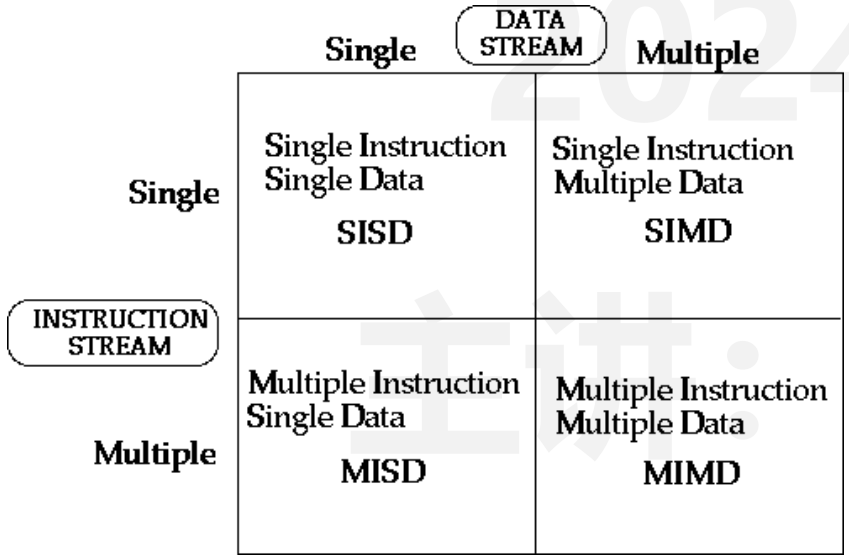
代表性指令集： GPU的CUDA指令集

- SISD、SIMD、MISD、MIMD的扩展

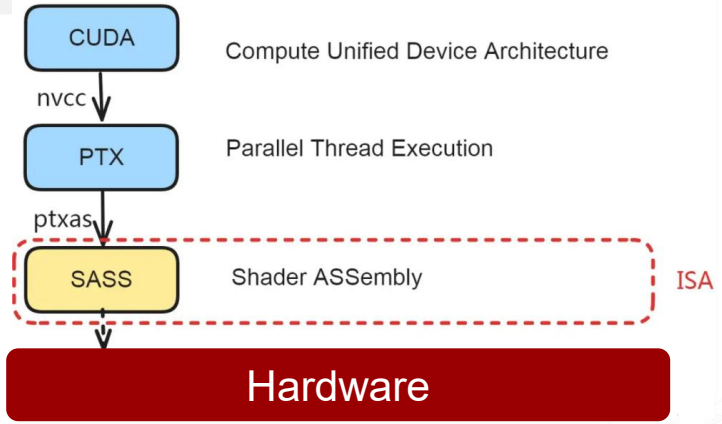


并行计算



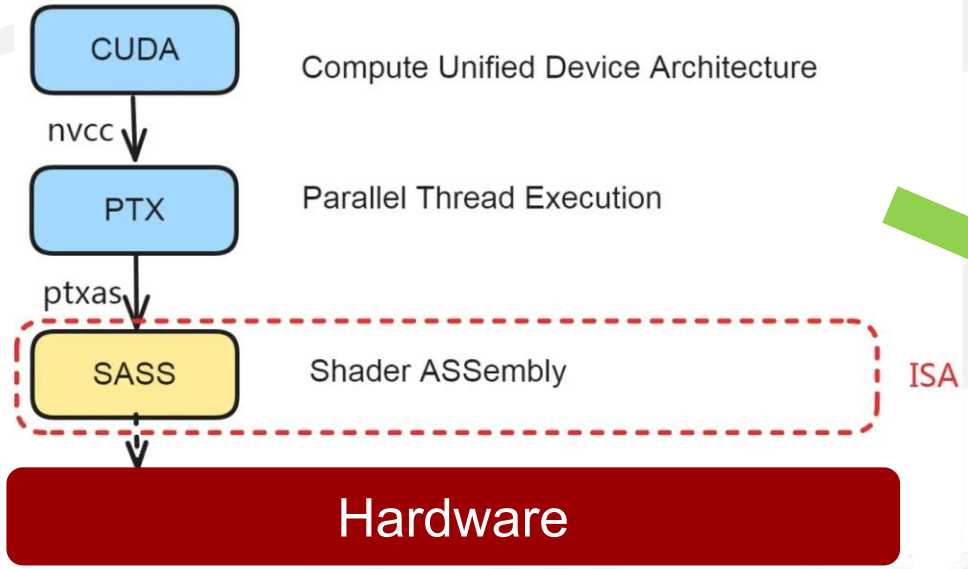


CUDA指令集的层次



代表性指令集： GPU的CUDA指令集

- PTX和SASS



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

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

SASS指令集与GPU的SM架构有直接对应关系，一旦硬件架构设计完成就不再改变

代表性指令集： GPU的CUDA指令集

- PTX和SASS

PTX

```
mov.u32   %r1, %clock;
add.u32   %r11, 6, %r7;
add.u32   %r12, %r5, 7;
add.u32   %r13, %r12, %r1;
mov.u32   %r2, %clock;
```

SASS

```
S2UR R356
DEPBAR.LE
IADD3 R14   R19  R23
IADD3 R13  R12  R-167
IADD3 R14  R14  R35  R23
CS2R.32 R322
```

PTX

```
mov.u64   %rd50, %clock64;
add.u32   %r11, 6, %r7;
add.u32   %r12, %r5, 7;
add.u32   %r13, %r12, %r1;
mov.u64   %rd51, %clock64;
```

SASS

```
CS2R R128
IADD3 R14   R1915   R231
IADD3 R141  R123    R335
IADD3 R32  R146 R123 R231
CS2R R133
```

Mapping of PTX to SASS

代表性指令集： GPU的CUDA指令集

北京大学 PEKING UNIVERSITY

- PTX和SASS

PTX	SASS	cycles	PTX	SASS	cycles
Add / sub instruction			**Min/Max instructions**		
add.u16	UIADD3	2	Min.u16	ULOP3.LUT+UISETP.LT.U32.AND+USEL	8
addc.u32	IADD3.X	2	min.u32	IMNMX.U32	2
add.u32	IADD	2	min.u64	UISETP.LT.U32.AND+2*USEL	8
add.u64	UIADD3.x+ UIADD3	4	min.s16	PRMT+IMNMX	4
add.s64	UIADD3.x+UIADD3	4	min.s32	IMNMX	2
add.f16	HADD	2	Min.s64	UISETP.LT.U32.AND+UISETP.LT.AND.EX+2*USEL	8
add.f32	FADD	2	min.f16	HMNMX2+PRMT	4
add.f64	DADD	4	min.f32	FMNMX	2
Mul instruction			min.f364	DSETP.MIN.AND+IMAD.MOV.U32+UMOV+FSEL	10
mul.wide.u16	LOP3.LUT+IMAD	4	**Neg instruction**		
mul.wide.u32	IMAD	4	neg.s16	UIADD3+UPRMT	5
mul.lo.u16	LOP3.LUT+IMAD	4	neg.s32	IADD3	2
mul.lo.u32	IMAD	2	neg.s64	IMAD.MOV.U32+HFMA2.MMA+MOV+UIADD3	10
mul.lo.u64	IMAD	2	neg.f32	FADD or IMAD.MOV.U32 *	2
mul24.lo.u32	PRMT + IMAD	3	neg.f64	DADD+(UMOV)	4
mul24.hi.u32	UPRMT+USHF.R.U32.HI+IMAD.U32+PRMT	9	**FMA instruction**		
mul.rn.f16	HMUL2	2	fma.rn.f16	HFMA2	2
mul.rn.f32	FMUL	2	fma.rn.f32	FFMA	2
mul.rn.f64	DMUL	4	fma.rn.f64	DFMA	4
MAD Instruction			**Sqrt Instruction**		
mad.lo.u16	LOP3.LUT+IMAD	4	sqrt.rn.f32	[multiple instrs including MUFU.RSQ]	190-235
mad.lo.u32	FFMA	2	sqrt.approx.f32	[multiple instrs including MUFU.SQRT]	2-18
mad.lo.u64	IMAD	2	sqrt.rn.f64	[multiple insts including MUFU.RSQ64]	260-340
mad24.lo.u32	SGXT.U32 + IMAD	4	**Rsqrt Instruction**		
mad24.hi.u32	USHF.R.U32.HI+UIMAD.WIDE.U32+2*UPRMT+IADD3	11	rsqrt.approx.f32	[multiple insts including MUFU.RSQ]	2-18
mad.rn.f32	FFMA	2	rsqrt.approx.f64	MUFU.RSQ64H	8-11
mad.rn.f64	DFMA	4	**Rcp Instruction**		
Sad Instruction			rcp.rn.f32	[multiple insts including MUFU.RCP]	198
sad.u16/s16	(2*LOP3) +ULOP3+ VABSDIFF	6	rcp.approx.f32	[multiple insts including MUFU.RCP]	23
sad.u32/s32	VABSDIFF +IMAD (1 IMAD + 1 Umov for 3 instrs)	3	rcp.rn.f64	[multiple insts including MUFU.RCP64H]	244
sad.u64/s64	UISETP.GE.U32.AND+UIADD+IADD	10	ex2.approx.f32	FSTEP + FMUL + MUFU.EX2 + FMUL	14

代表性指令集： GPU的CUDA指令集

- PTX和SASS

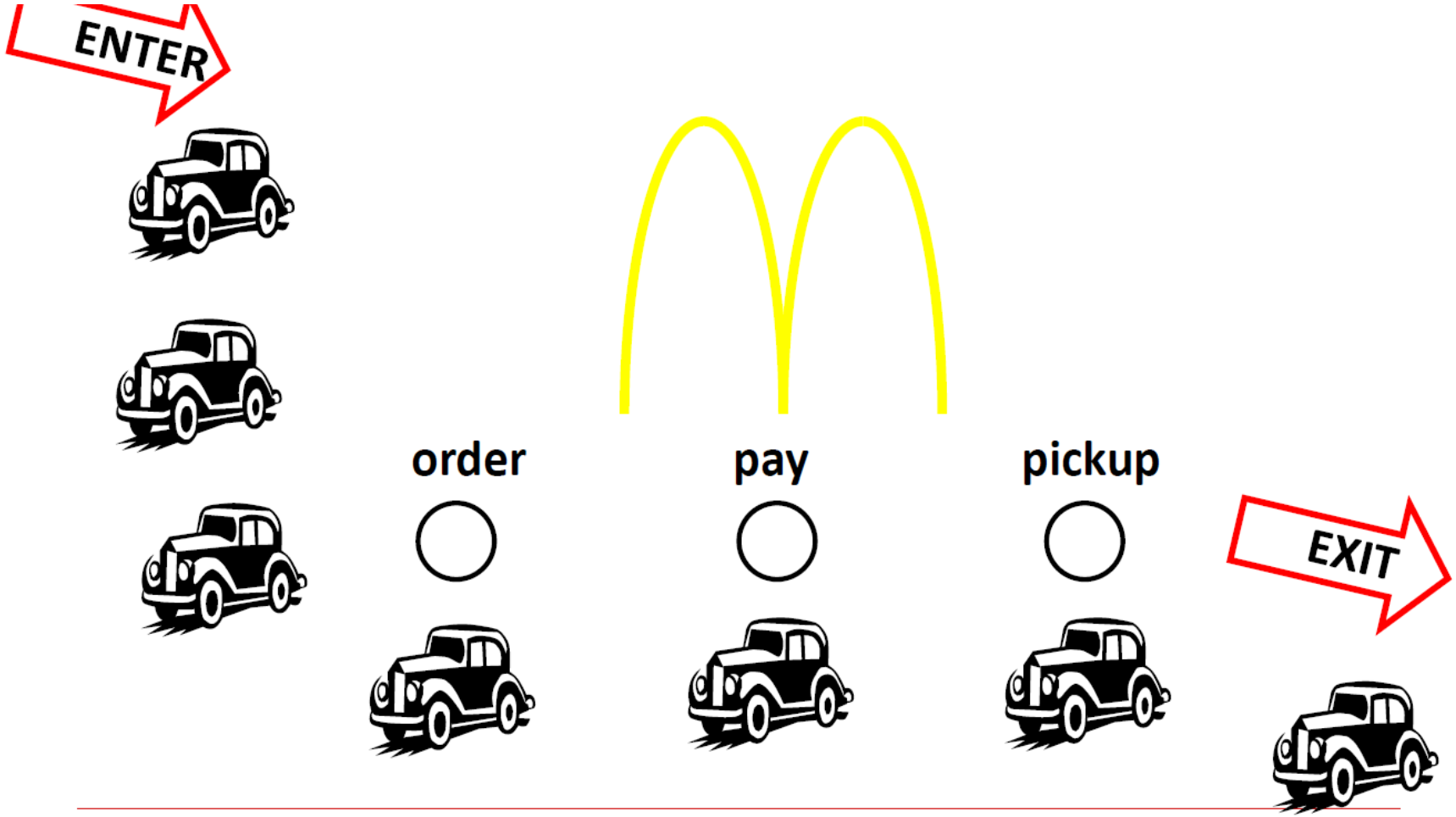
Div / Rem Instruction			Pop Instruction		
rem/div.u16/s16	multiple instructions	290	popc.b32S	POPC	6
rem/div.s32/u32	multiple instructions	66	popc.b64	2*UPOPC + UIADD3	7
rem/div.u64/s64	multiple instructions	420	**Clz Instruction**		
div.rn.f32	multiple instructions	525	clz.b32	FLO.U32 + IADD	7
div.rn.f64	multiple instructions	426	clz.b64	UISETP.NE.U32.AND+USEL+UFLO.U32+2*UIADD3	13
Abs Instruction			**Bfind Instruction**		
abs.s16	PRMT+IABS+PRMT	4	bfind.u32	FLO.U32	6
abs.s32	IABS	2	bfind.u64	FLO.U32+ISETP.NE.U32.AND+IADD3+BRA	164
abs.s64	UISETP.LT.AND+UIADD3.X +UIADD3+2*USEL	11	bfind.s32	FLO	6
abs.f16	PRMT	1	bfind.s64	multiple instructions	195
abs.ftz.f32	FADD.FTZ	2	**testp Instruction**		
abs.f64	DADD or (DADD+UMOV)	4	testp.normal.f32	IMAD.MOV.U32+2*ISETP.GE.U32.AND	0 or 6
Brev Instruction			testp.subnor.f32	ISETP.LT.U32.AND	0 or 6
brev.b32	BREV + SGXT.U32	2	testp.normal.f64	2*UISETP.LE.U32.AND+2*UISETP.GE.U32.AND	13
brev.b64	2*UBREV+MOV	6	testp.subnor.f64	UISETP.LT.U32.AND+2*UISETP.GE.U32.AND.EX	8
copysign Instruction			**Other Instruction**		
copysign.f32	2*LOP3.LUT or 1.5*LOP3.LUT	4	sin.approx.f32	FMUL + MUFU.SIN	8
copysign.f64	2*ULOP3.LUT+IMAD.U32+*MOV	6	cos.approx.f32	FMUL.RZ+MUFU.COS	8
and/or/xor Instruction			lg2.approx.f32	FSETP.GEU.AND+FMUL+MUFU.LG2+FADD	18
and.b16	LOP3.LUT or 1.5*LOP3.LUT	2	ex2.approx.f32	FSETP.GEU.AND+2*FMUL+MUFU.EX2	18
and.b32	LOP3.LUT	2	ex2.approx.f16	MUFU.EX2.F16	6
and.b64	ULOP3.LUT	2-3	tanh.approx.f32	MUFU.TANH	6
Not Instruction			tanh.approx.f16	MUFU.TANH.F16	6
not.b16	LOP3.LUT	2	bar.warp.sync;	NOP	changes
not.b32	LOP3.LUT	2	fns.b32	multiple instructions	79
not.b64	2*ULOP3.LUT	4	cvt.rzi.s32.f32	F2I.TRUNC.NTZ	6
lop3 Instruction			setp.ne.s32	ISETP.NE.AND	10
lop3.b32	IMAD.MOV.U32+LOP3.LUT	4	mov.u32 clock	CS2R.32	2
cnot Instruction			**Bfi Instruction**		
cnot.b16	ULOP3.LUT+ISETP.EQ.U32.AND+SEL	5	bfi.b32	3*PRMT+2*IMAD.MOV+SHF.L.U32+BMSK+LOP3.LUT	11
cnot.b32	UISETP.EQ.U32.AND+USEL	4	bfi.b64	UMOV+USHF.L.U32+(UIADD3+ULOP3.LUT)*	5
cnot.b64	multiple instructions	11	**dp4a.u32/s32 Instruction**		
bfe Instruction			dp4a.u32.u32	IMAD.MOV.U32+IDP.4A.U8.U8	135-170
bfe.s32/.u32	3*PRMT+2*IMAD.MOV+SHF.R.U32.HI+SGXT/.U32	11	**dp2a.u32/s32 Instruction**		
bfe.u64	UMOV+USHF.L.U32+(UIADD3+ULOP3.LUT)*	5	dp2a.lo.u32.u32	IMAD.MOV.U32+IDP.2A.LO.U16.U8	135-170
bfe.s64	multiple instructions	14			

目 录

CONTENTS

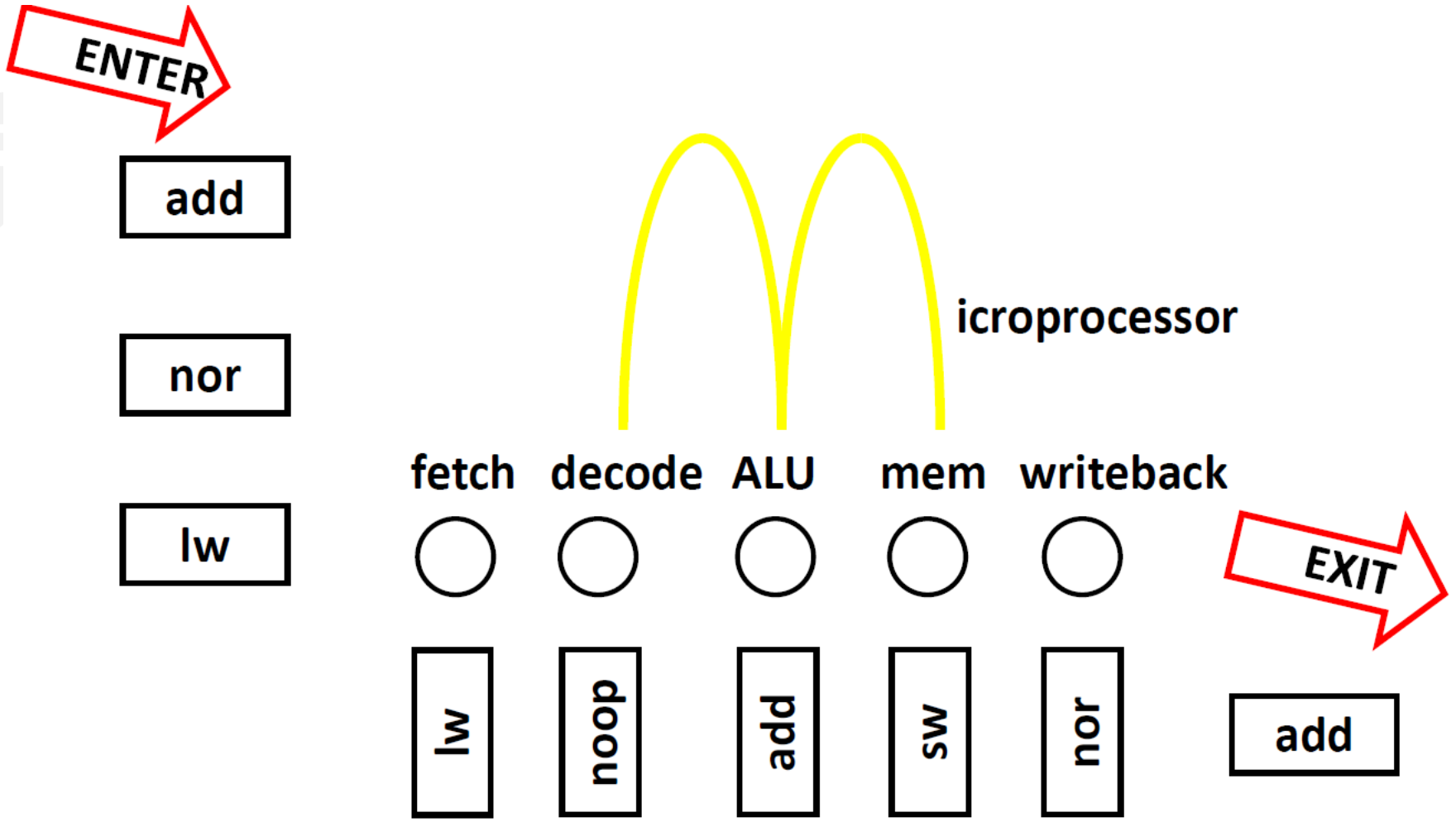
回顾：什么是流水线架构

- 流水线式运行方式 - 提高吞吐率的有效手段



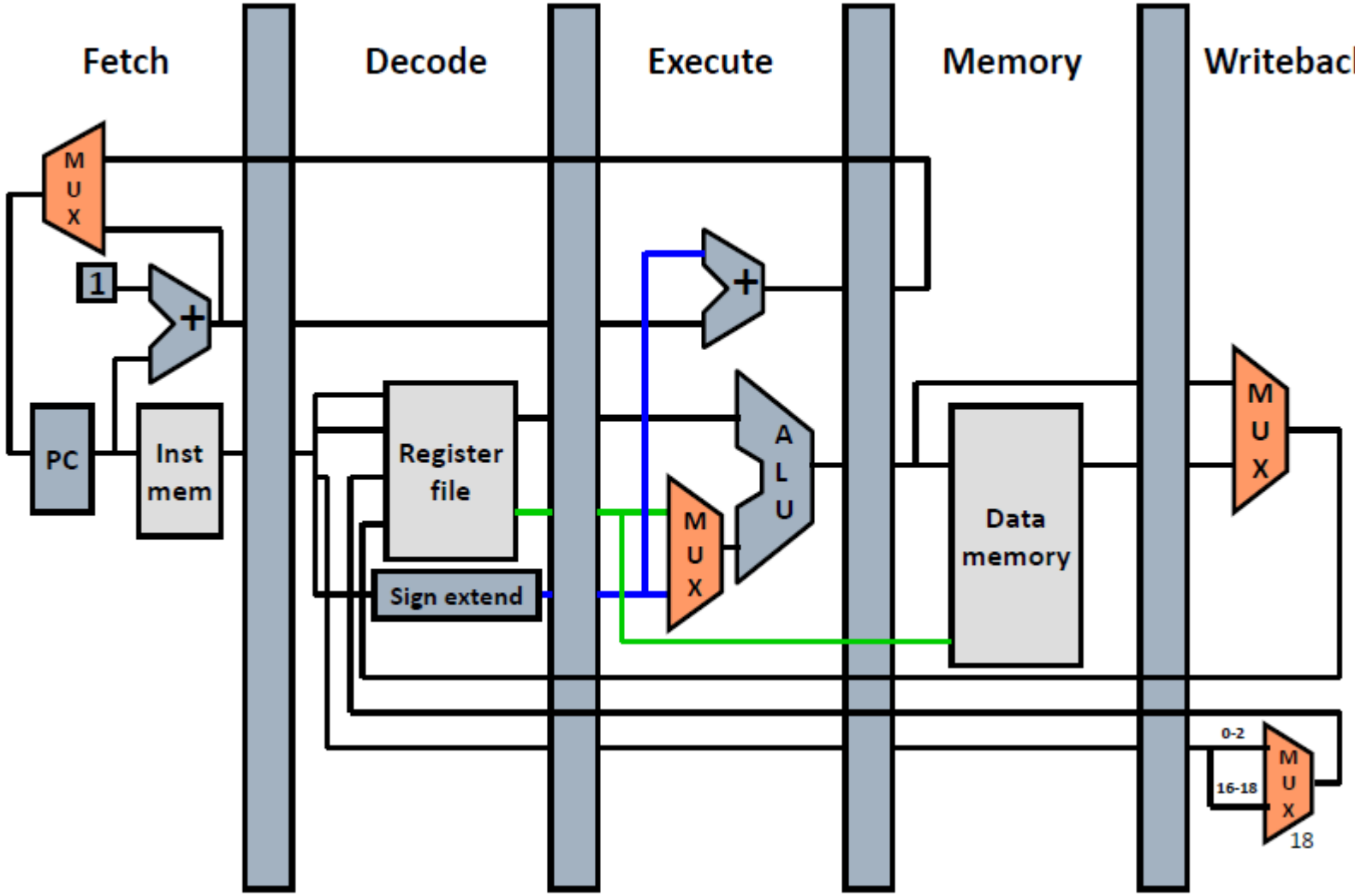
回顾：什么是流水线架构

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



最基本的单条流水线设计示意图

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

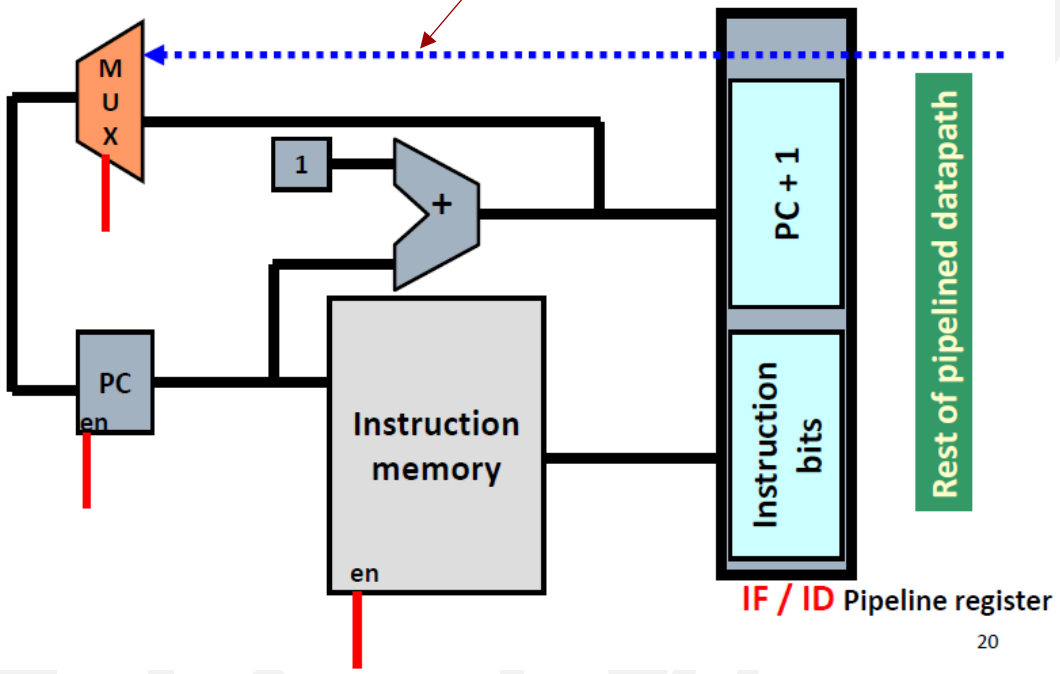


第一级：Fetch指令

- PC控制从指令Memory里读取的地址

用于更新PC应对控制指令（jmp等）

- 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

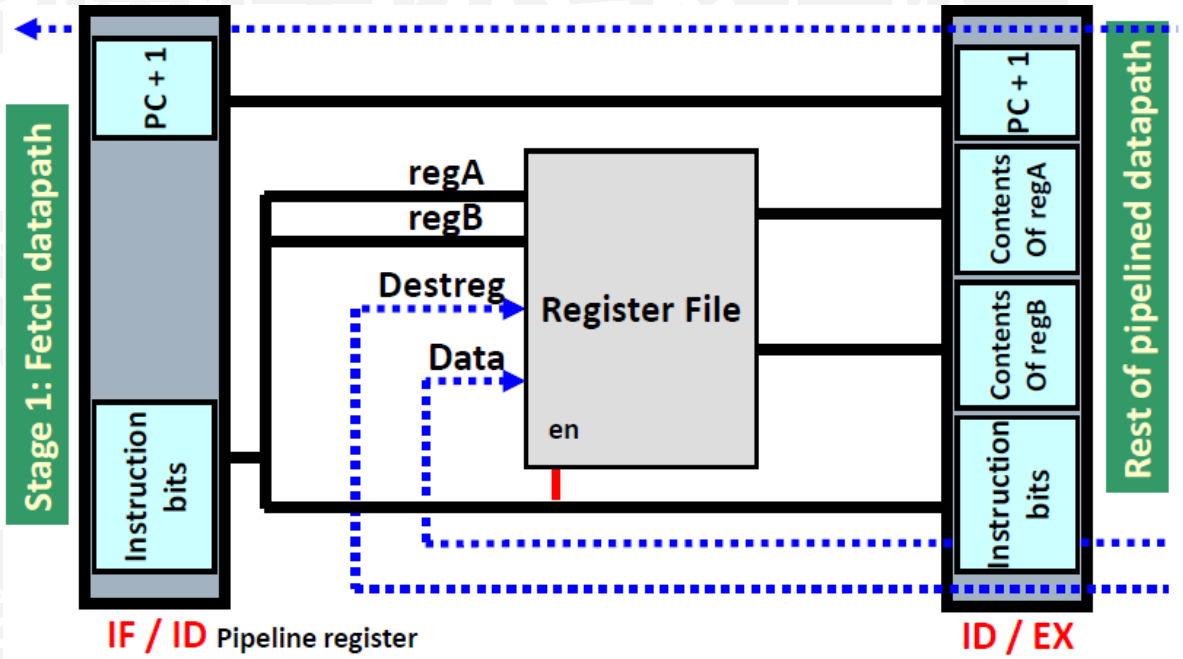


第二级：Decode指令

- 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

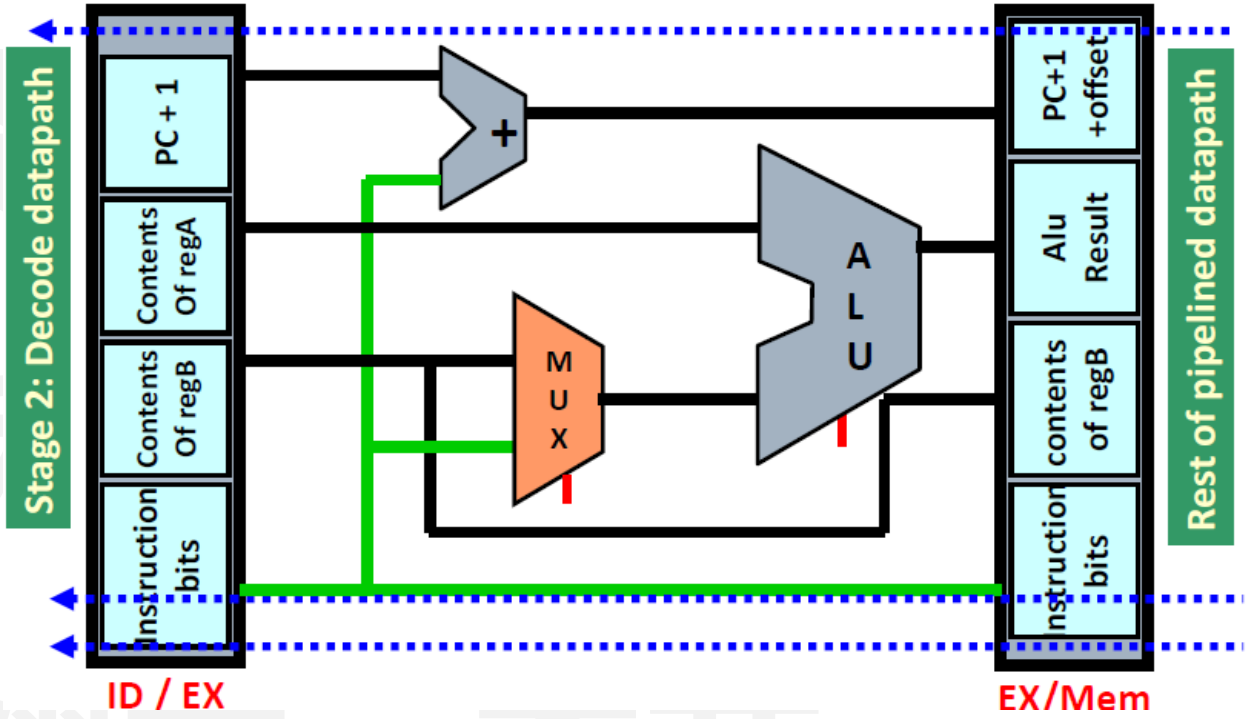
- Design a datapath that reads the IF/ID pipeline register, decodes instruction and reads register file (specified by regA and regB of instruction bits).
  - Decode is easy, just pass on the opcode and let later stages figure out their own control signals for the instruction.
- Write everything needed 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

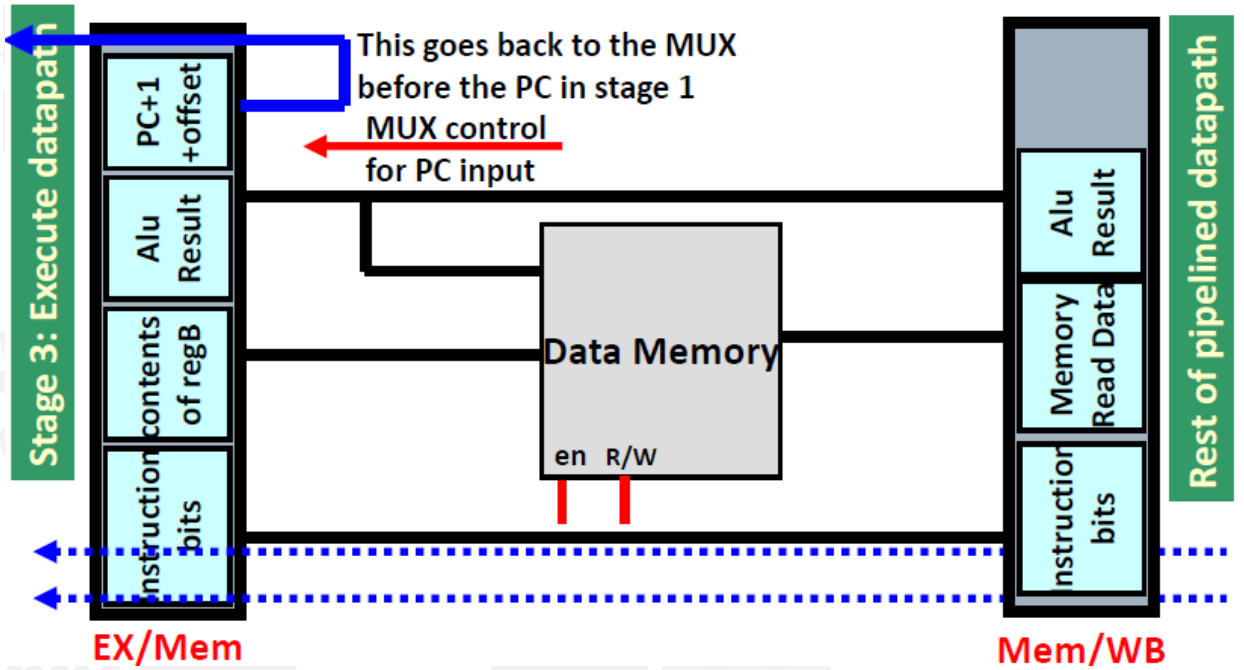
- Design a datapath that performs the proper ALU operation for the instruction specified and the values present in the ID/EX pipeline register.
  - The inputs are the contents of regA and either the contents of regB or the offset field on the instruction.
  - Also, calculate PC+1+offset in case this 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



第四级：Memory操作

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

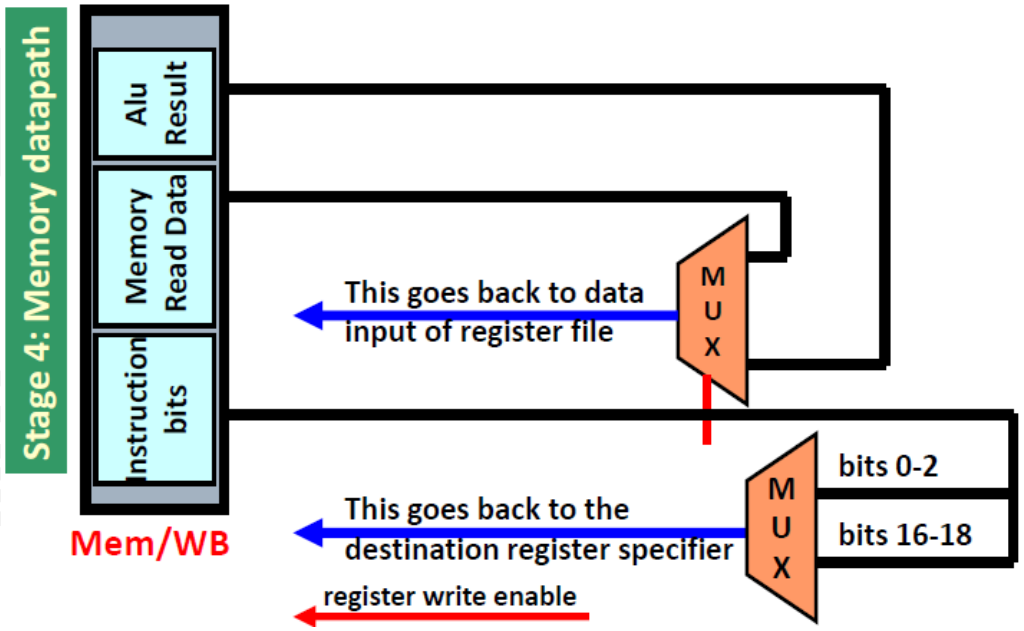
- Design a datapath that performs the proper memory operation for the instruction specified and the values present in the EX/Mem pipeline register.
  - ALU result contains address for ld and st instructions.
  - Opcode bits control memory R/W and enable signals.
- Write everything needed to complete 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

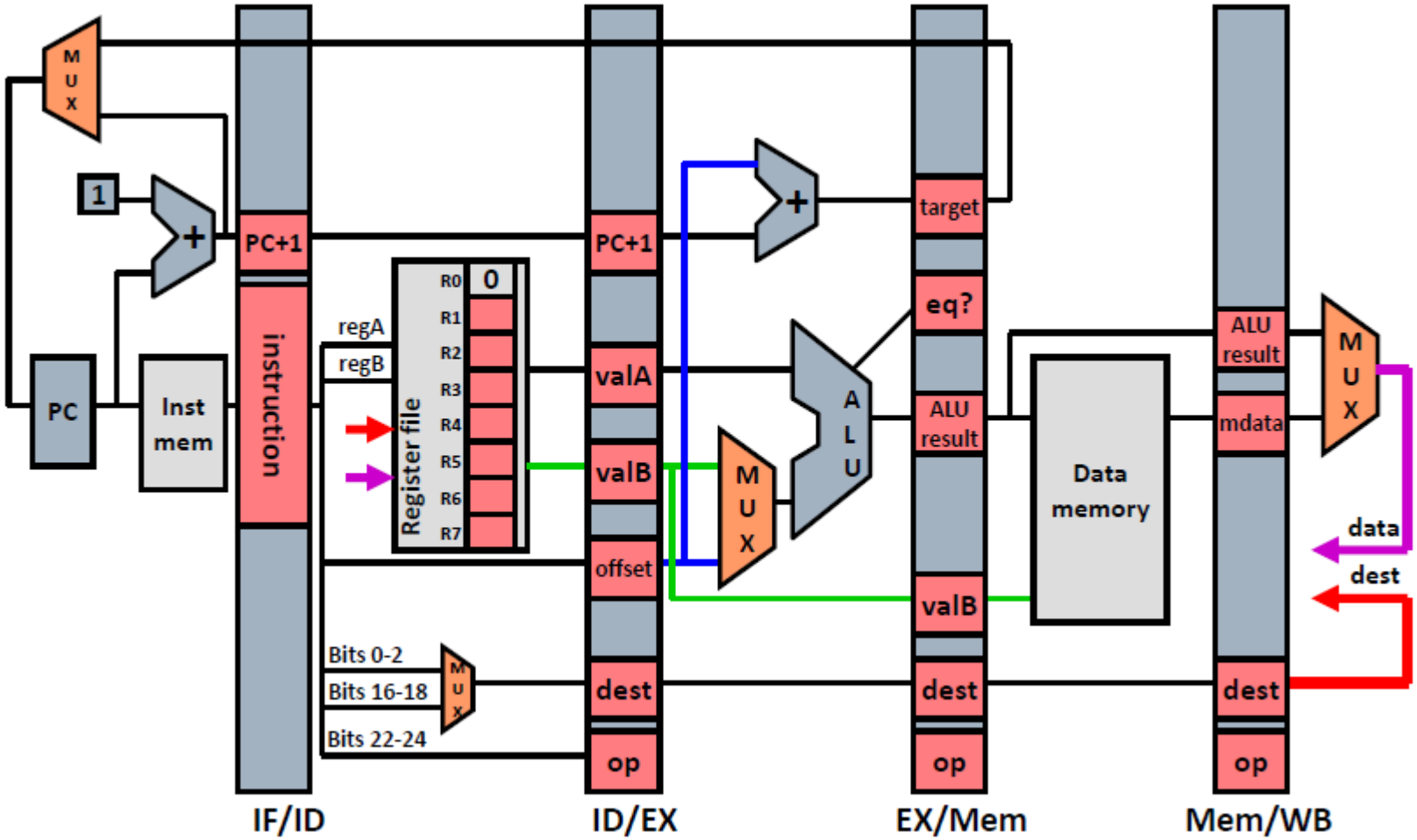
- Design a datapath that completes the execution of this instruction, writing to the register file if required.
  - Write MemData to destReg for ld instruction
  - Write ALU result to destReg for add or nand instructions.
  - Opcode bits also control register write enable signal.



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

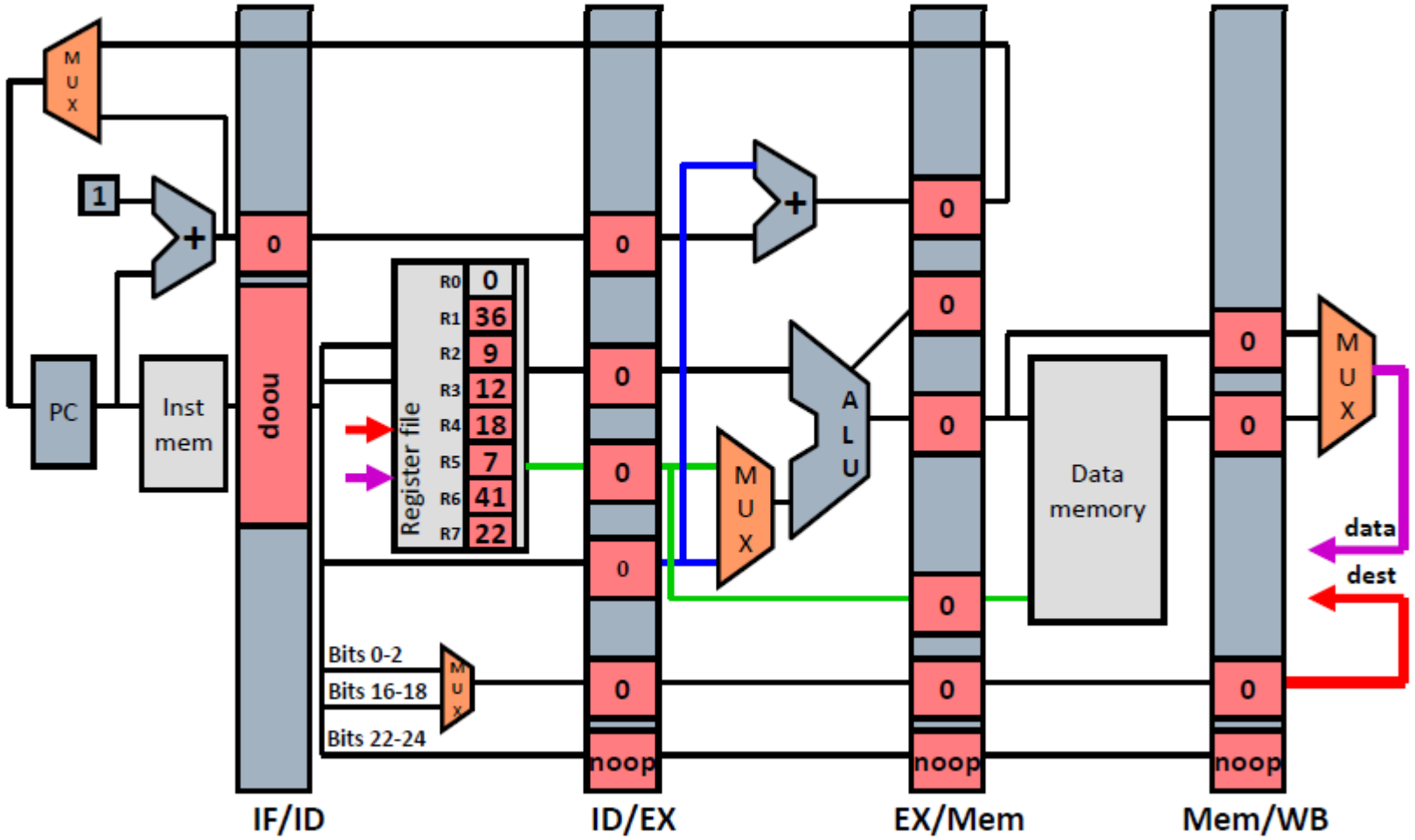
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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
```

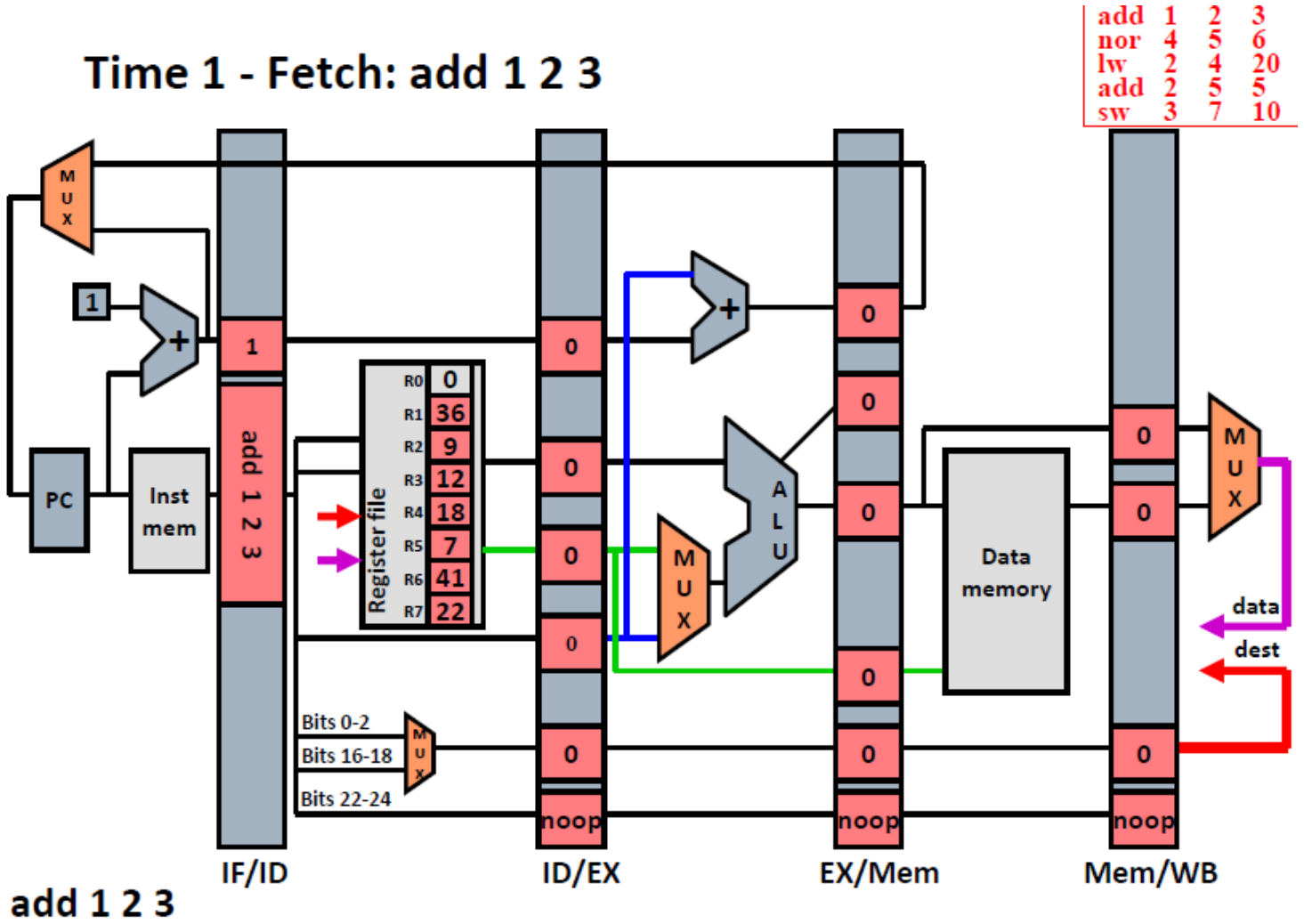


初始状态：t0

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

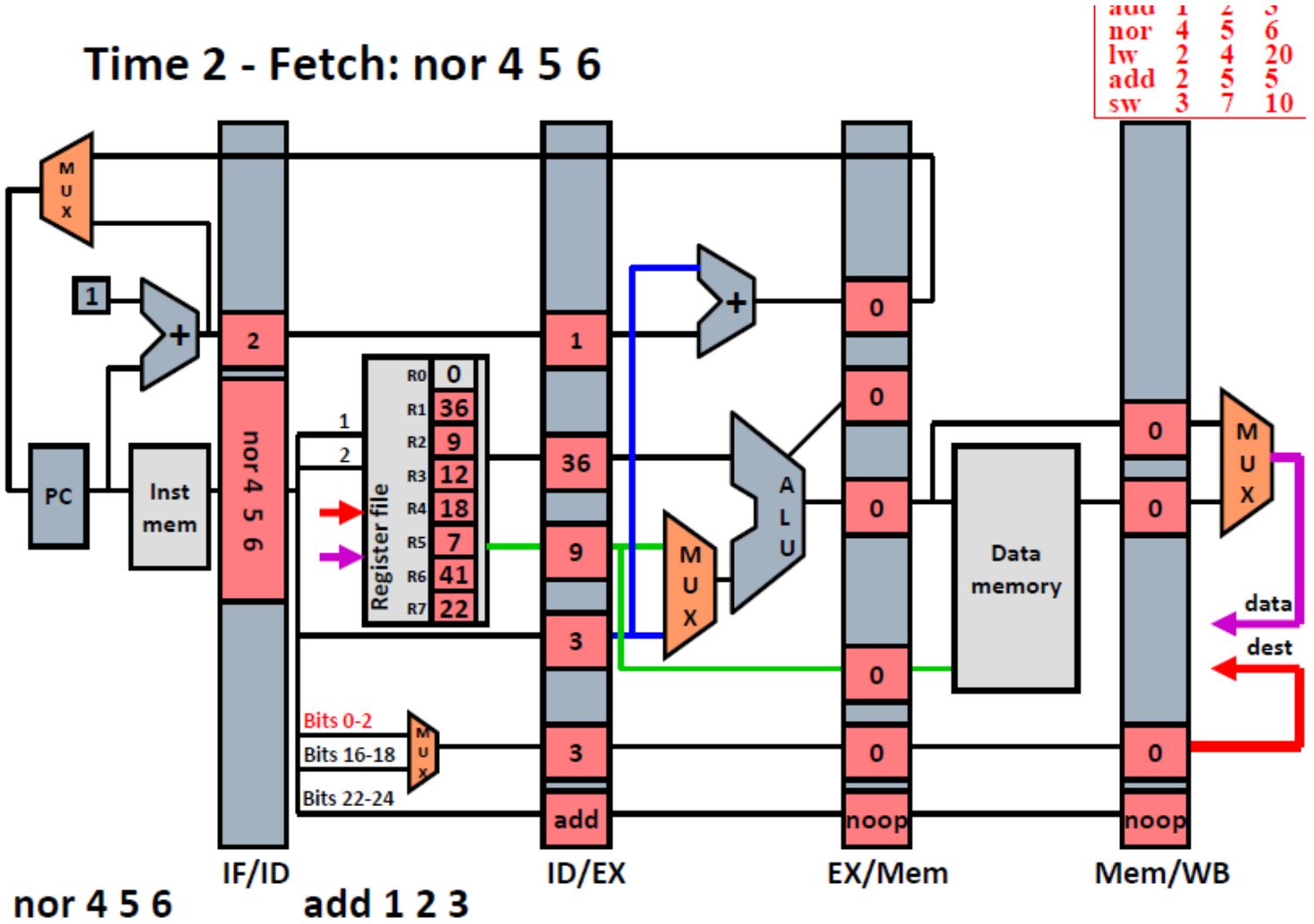
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

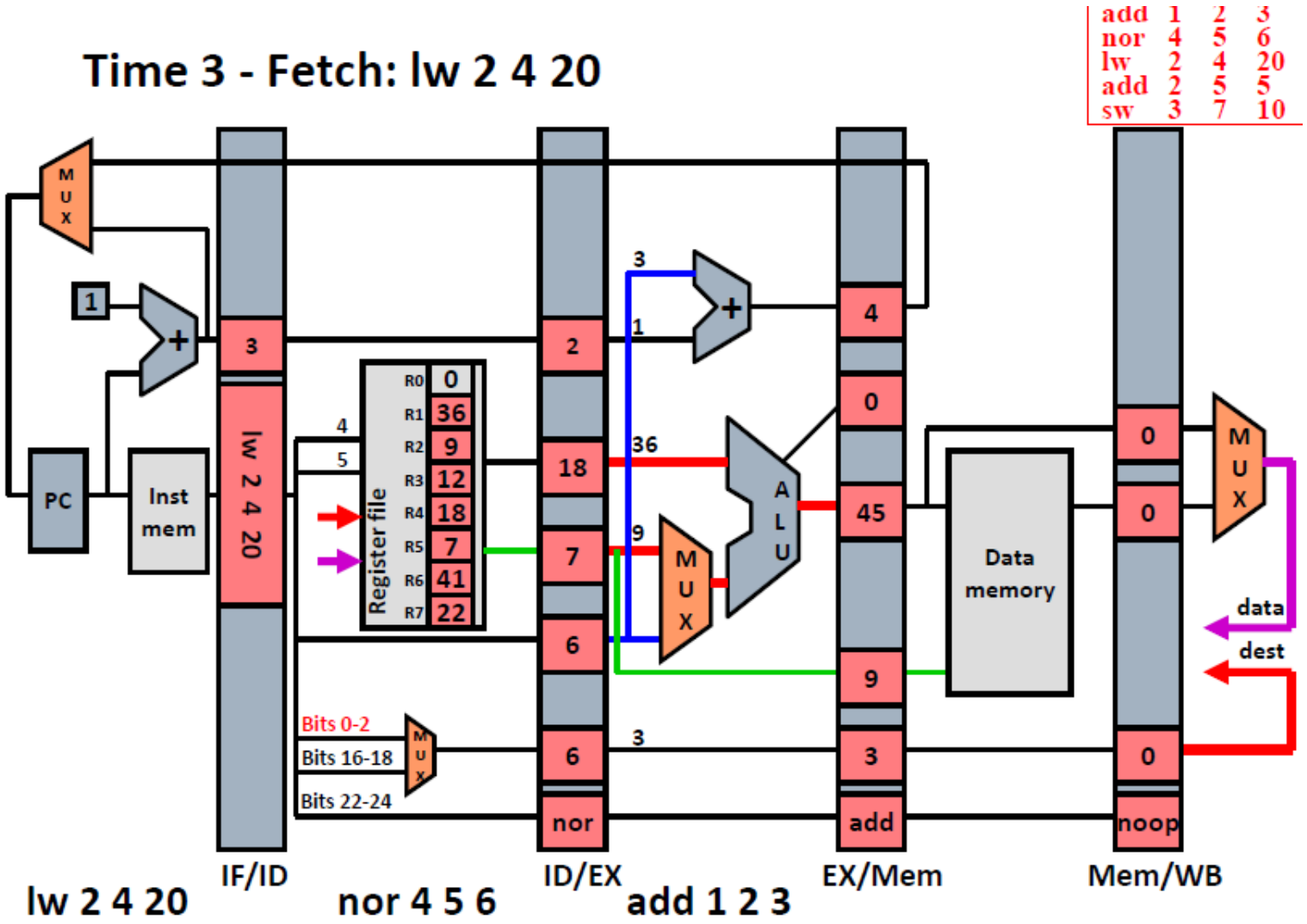
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

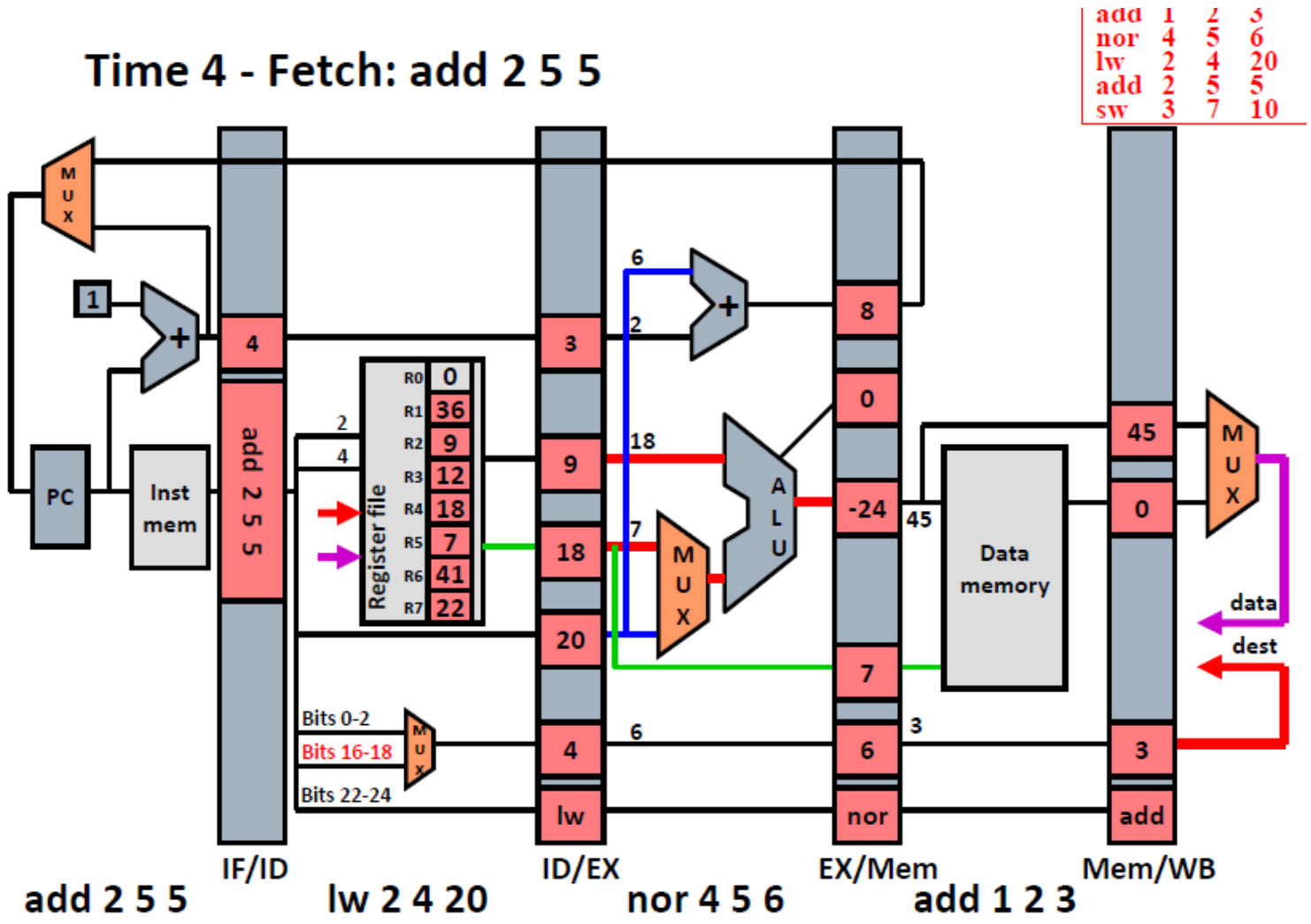
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

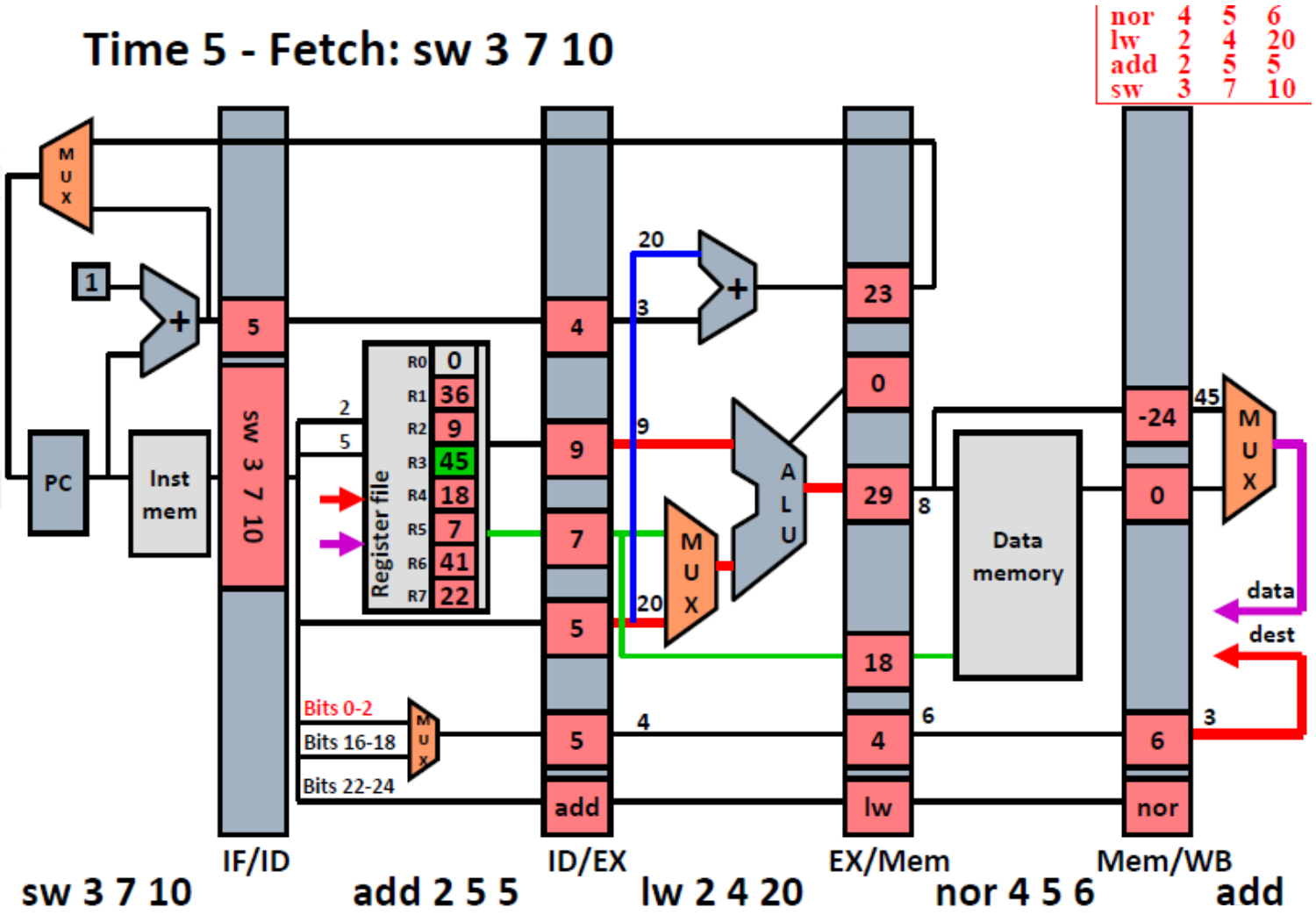
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

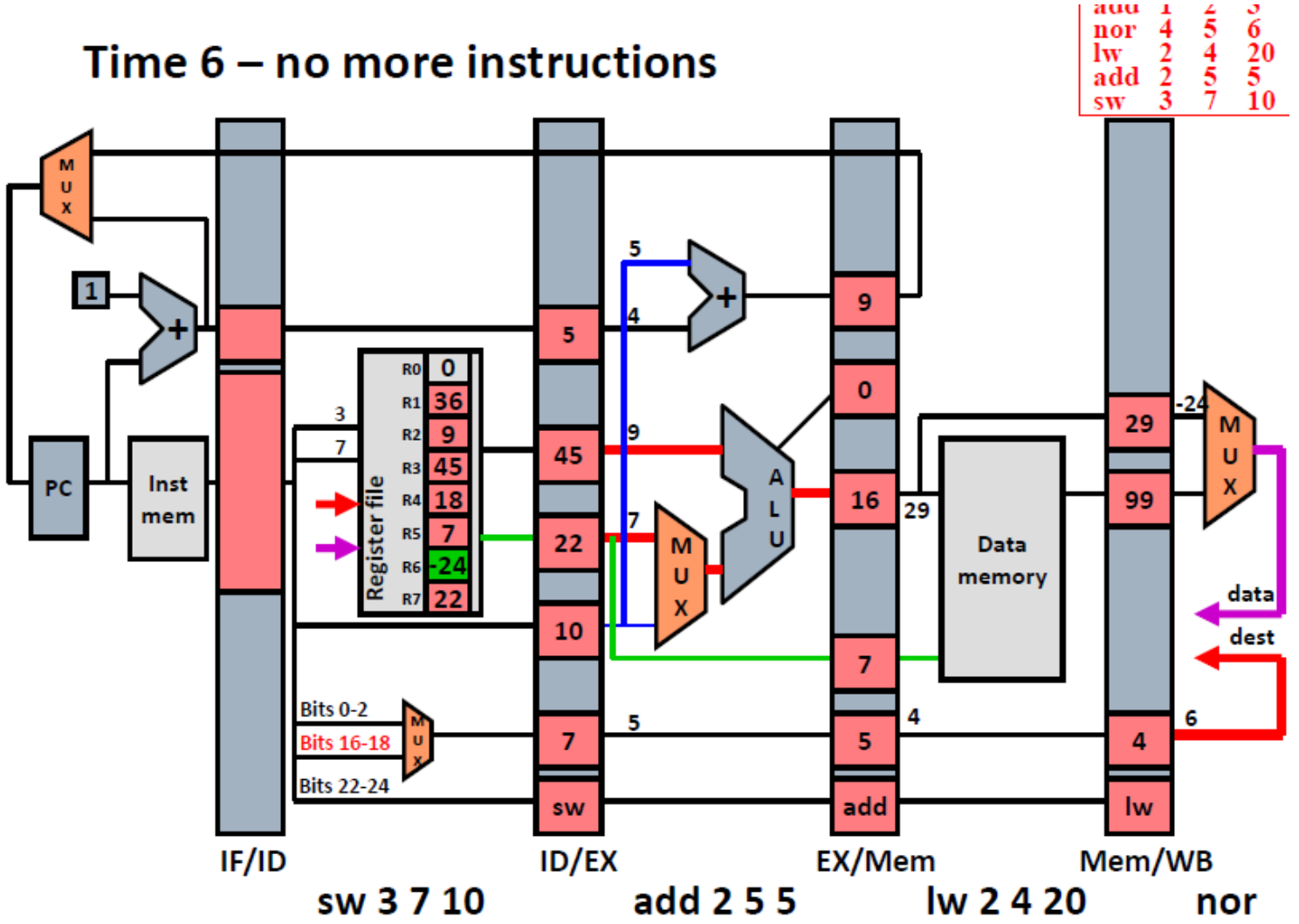
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

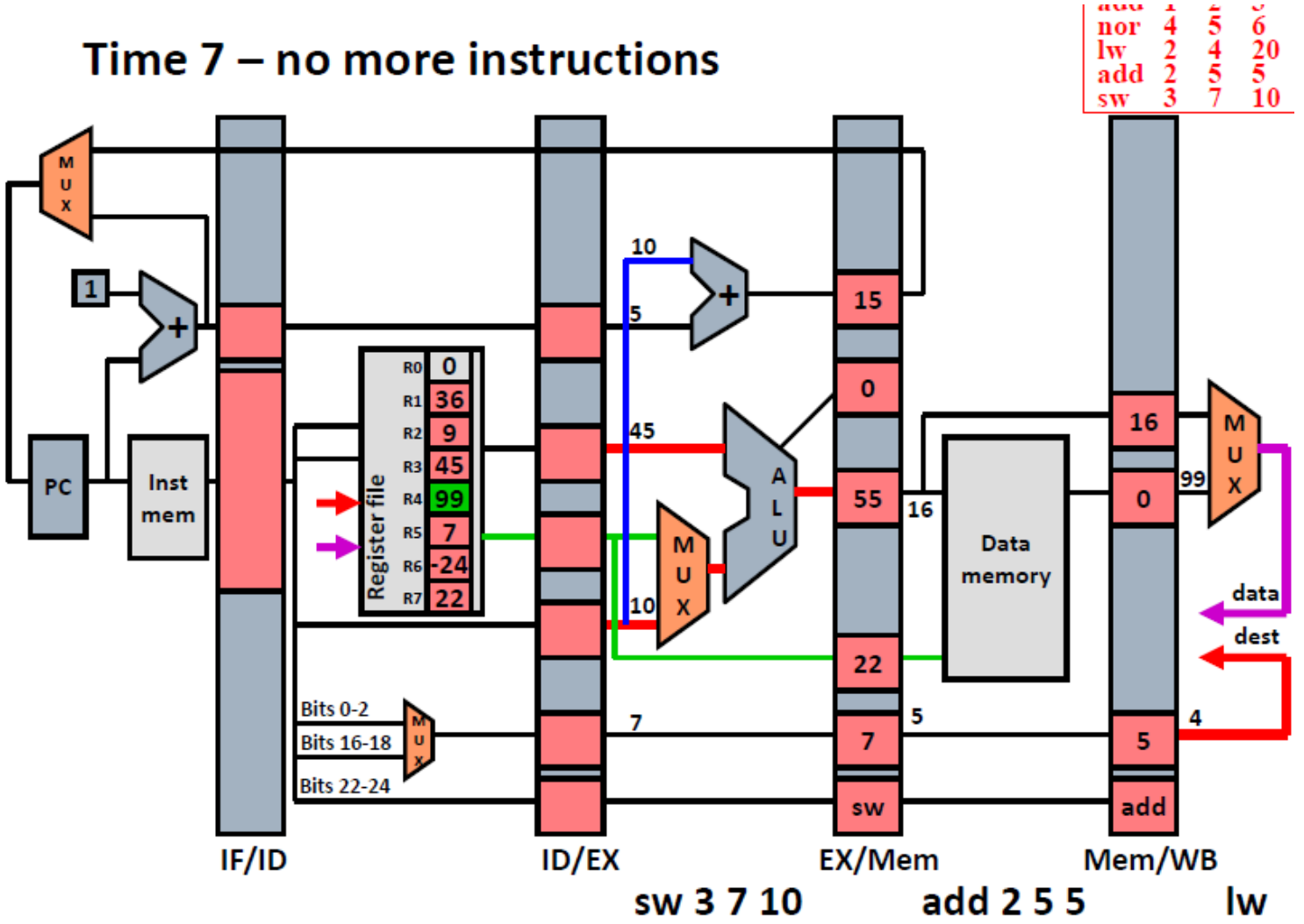
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

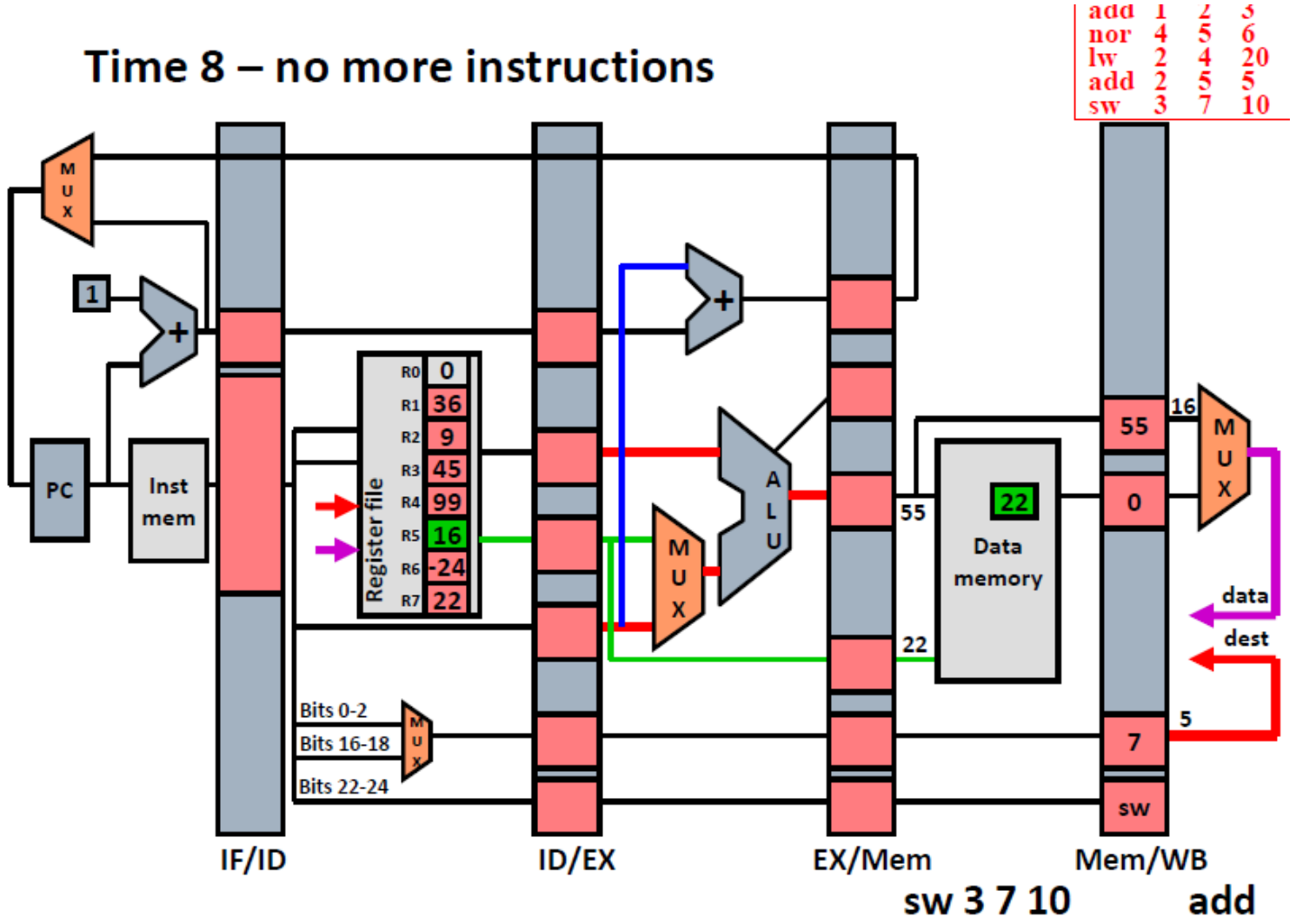
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10] =reg 7



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

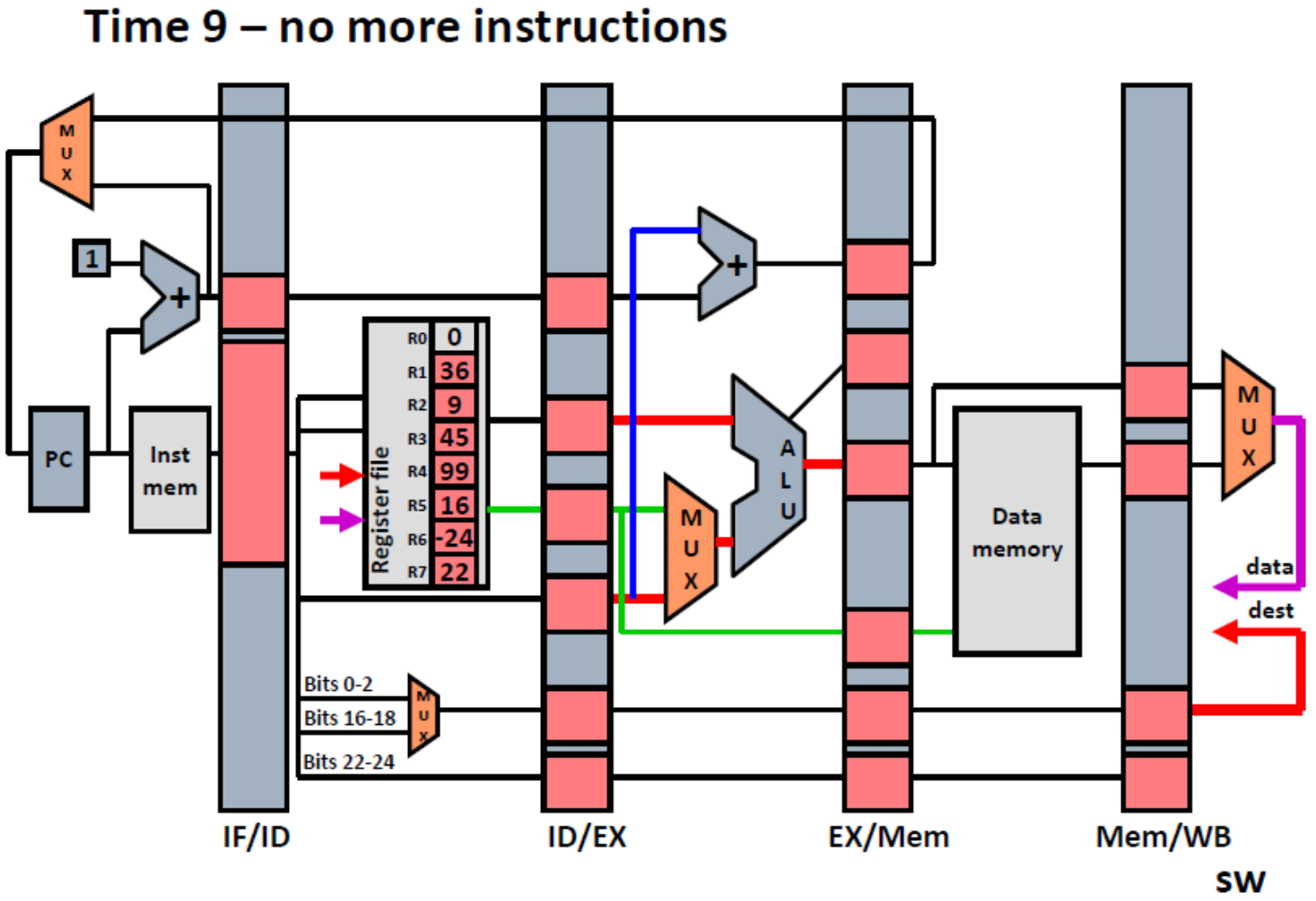
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

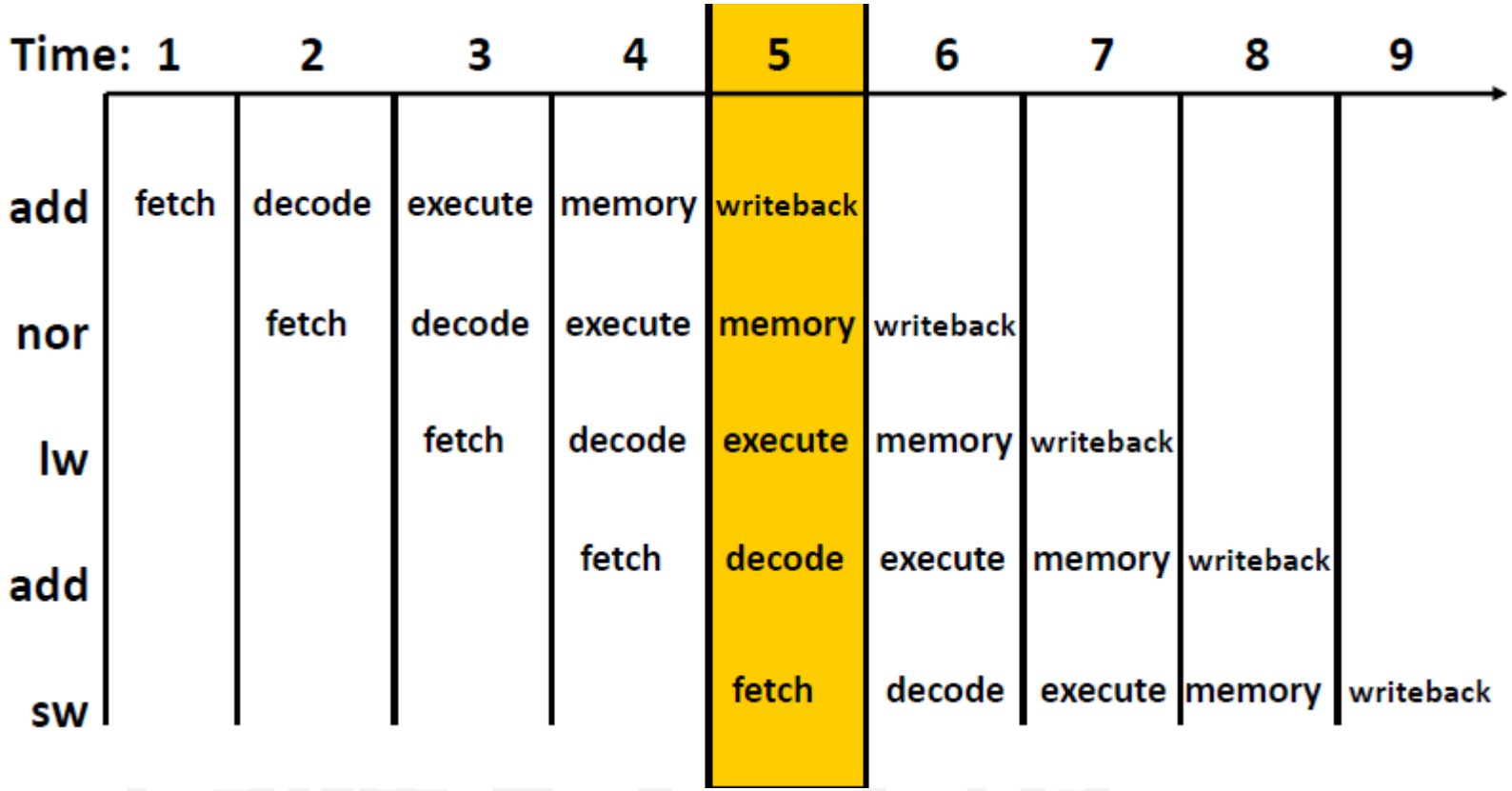
```
add   1  2  3    ; reg 3 = reg 1 + reg 2
nor   4  5  6    ; reg 6 = reg 4 nor reg 5
lw    2  4  20   ; reg 4 =  Mem[reg2+20]
add   2  5  5    ; reg 5 = reg 2 + reg 5
sw    3  7  10   ; Mem[reg3+10] =reg 7
```



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

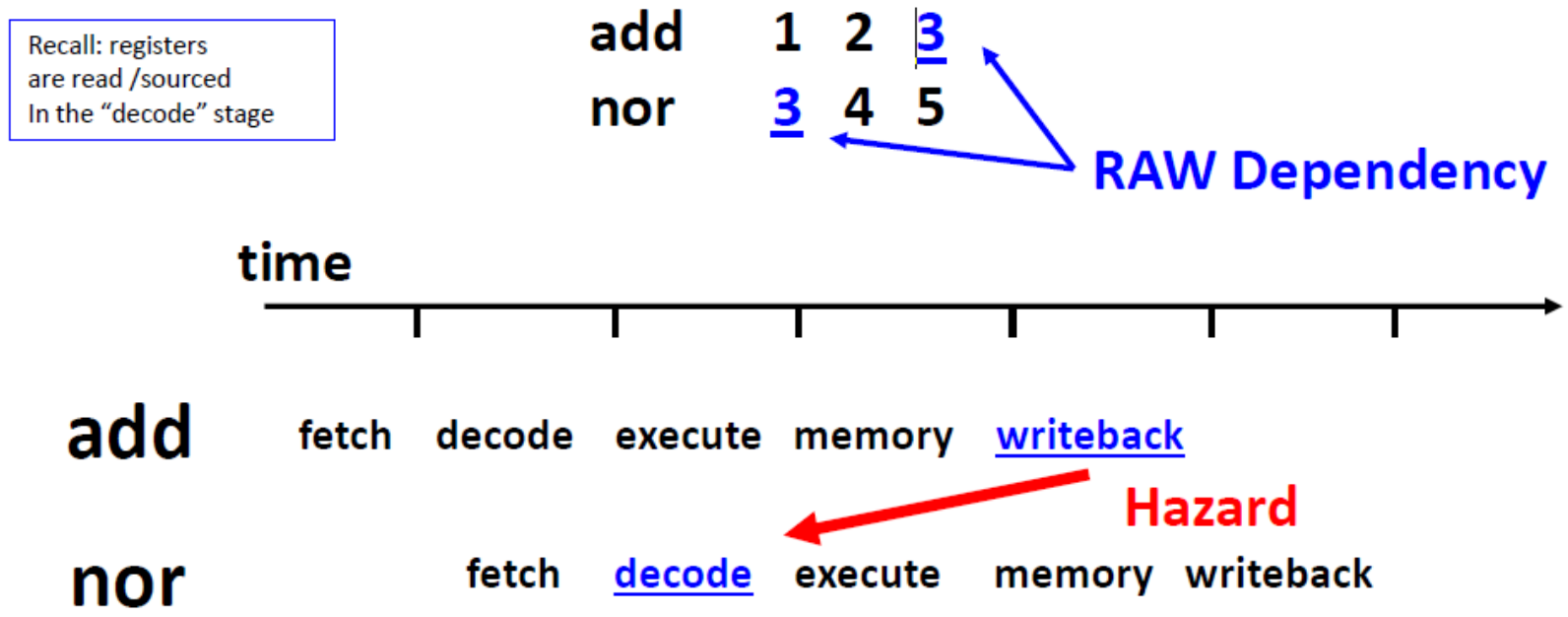
简单5级流水线可能存在什么问题?

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

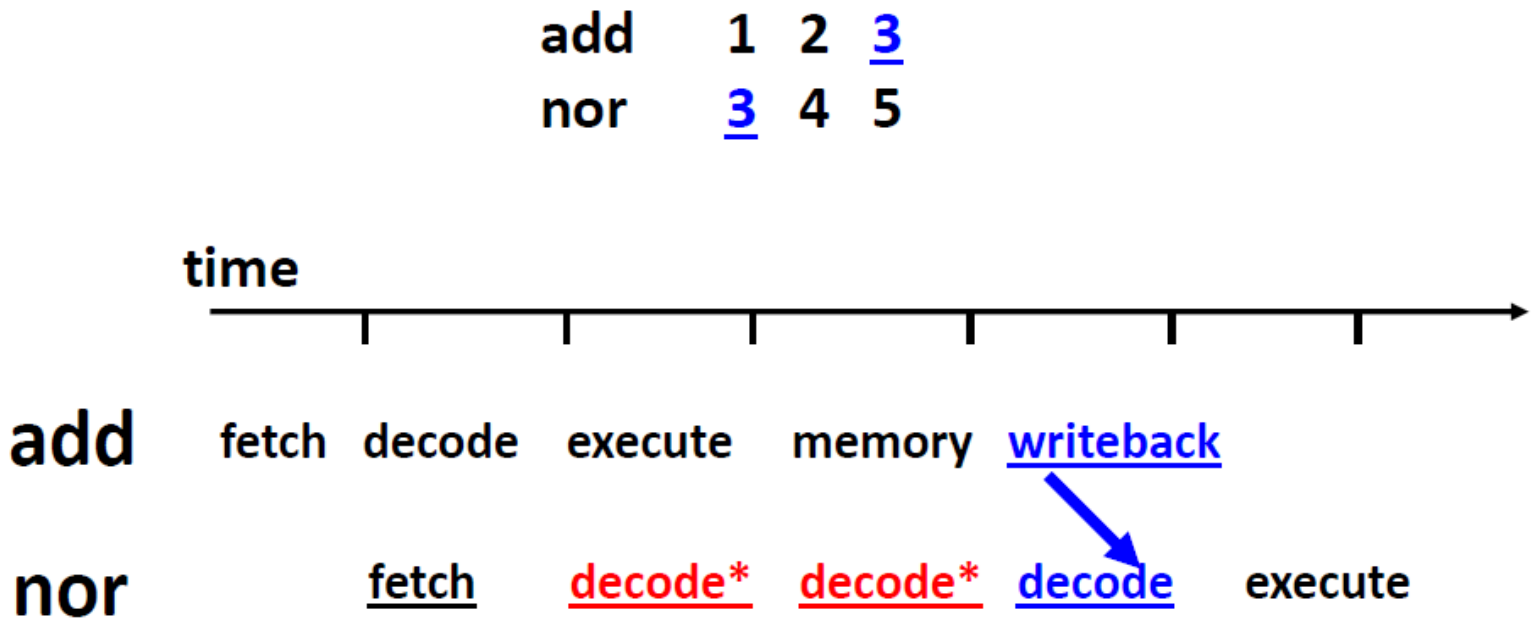
- RAW问题：Read After Write数据冲突



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

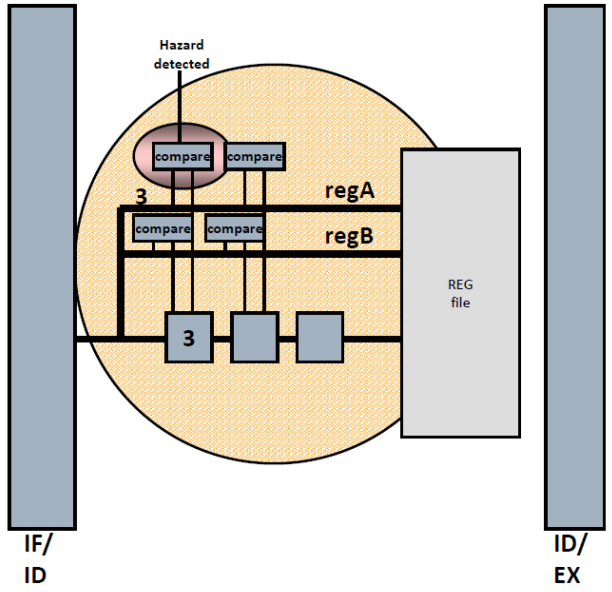


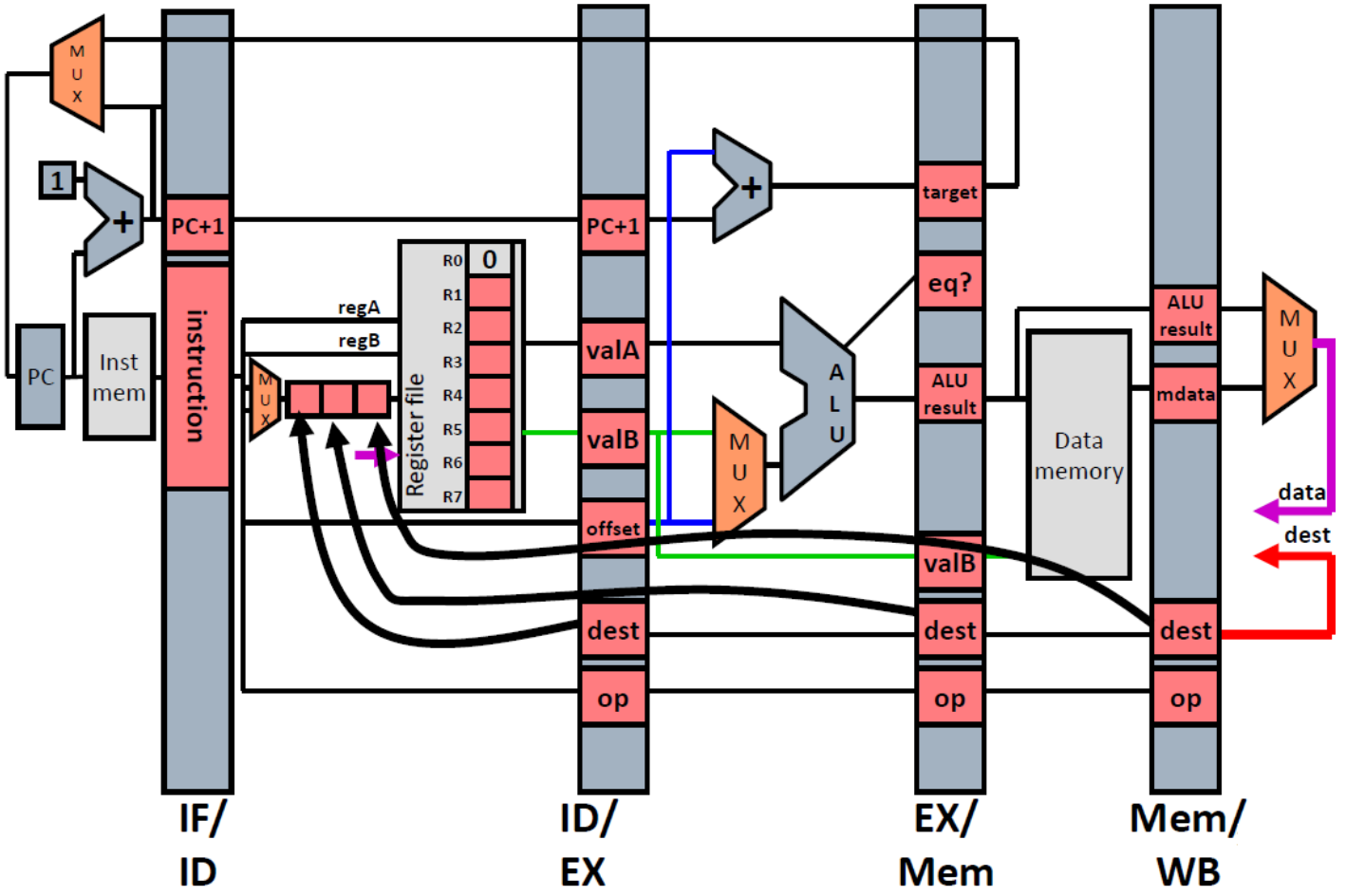
简单解决办法：流水线停顿（Pipeline Stall）

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
```

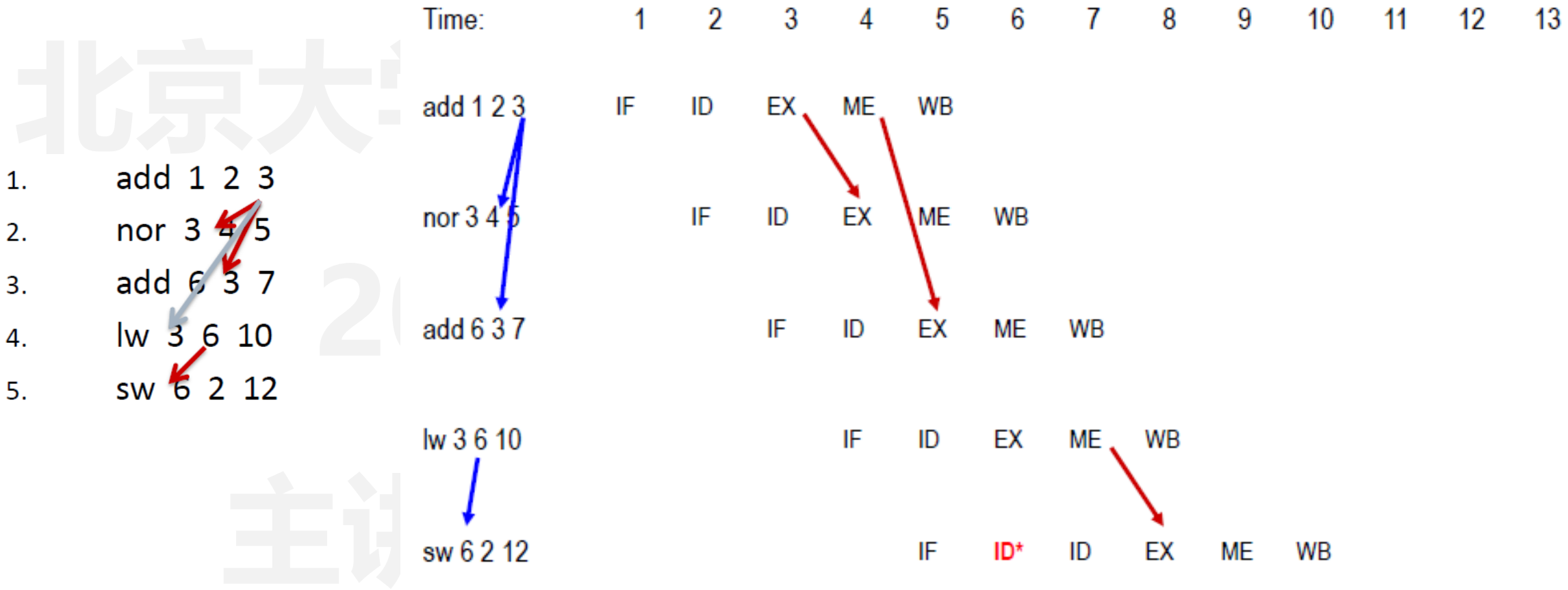




进阶解决办法：Detect and Forward

问题1：Data Hazards

- RAW问题：Read After Write数据冲突



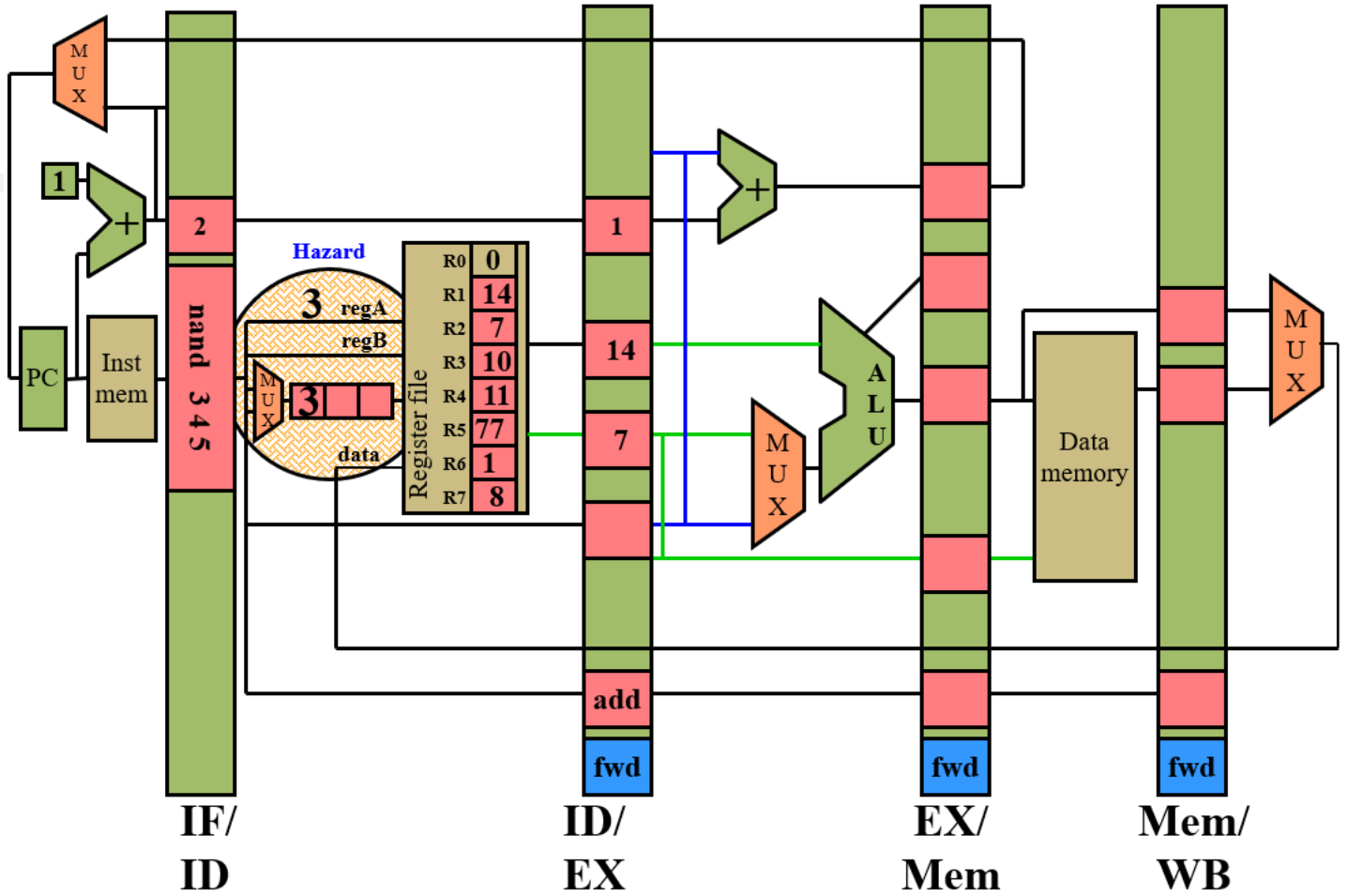
进阶解决办法：Detect and Forward

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

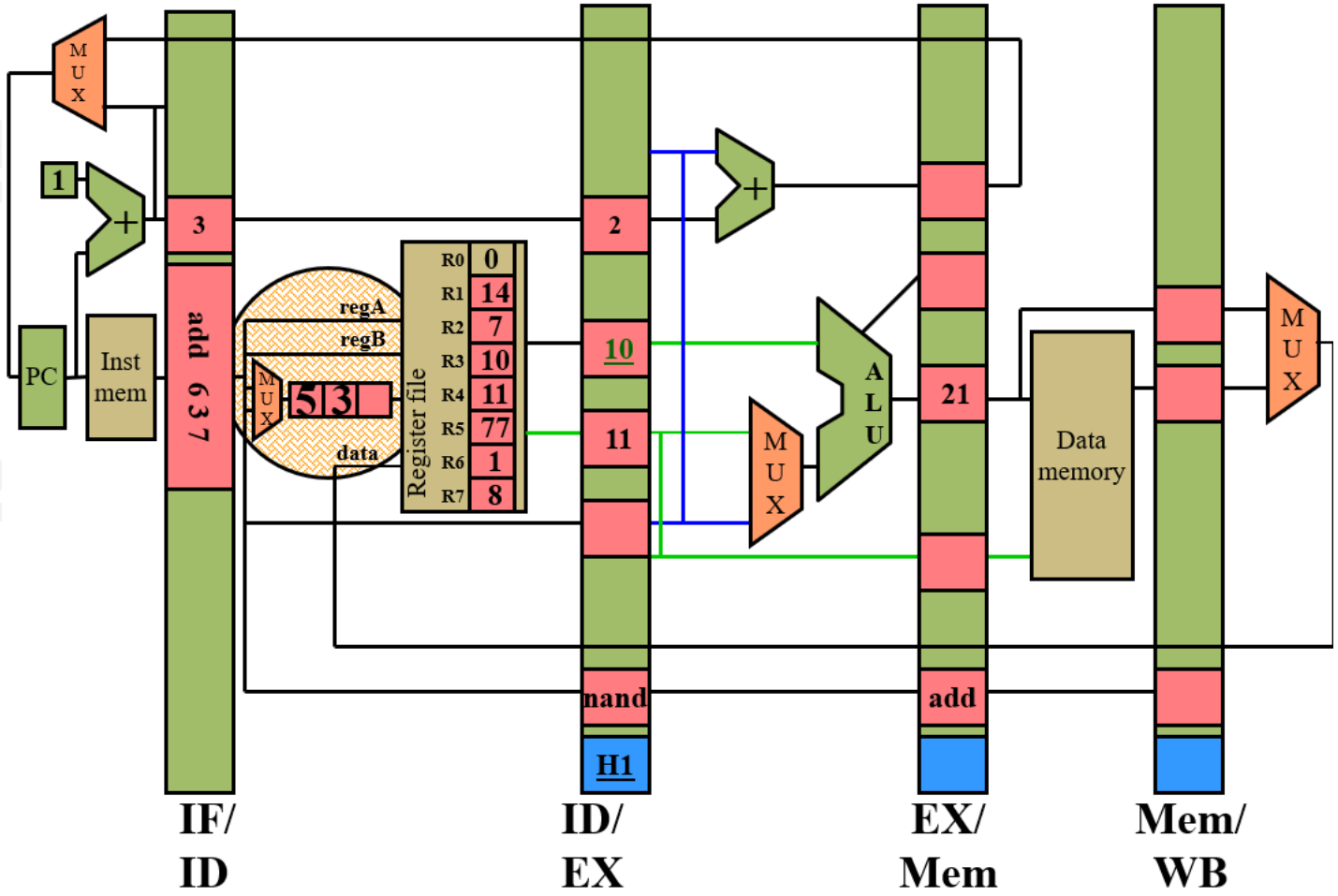


问题1：Data Hazards

- **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

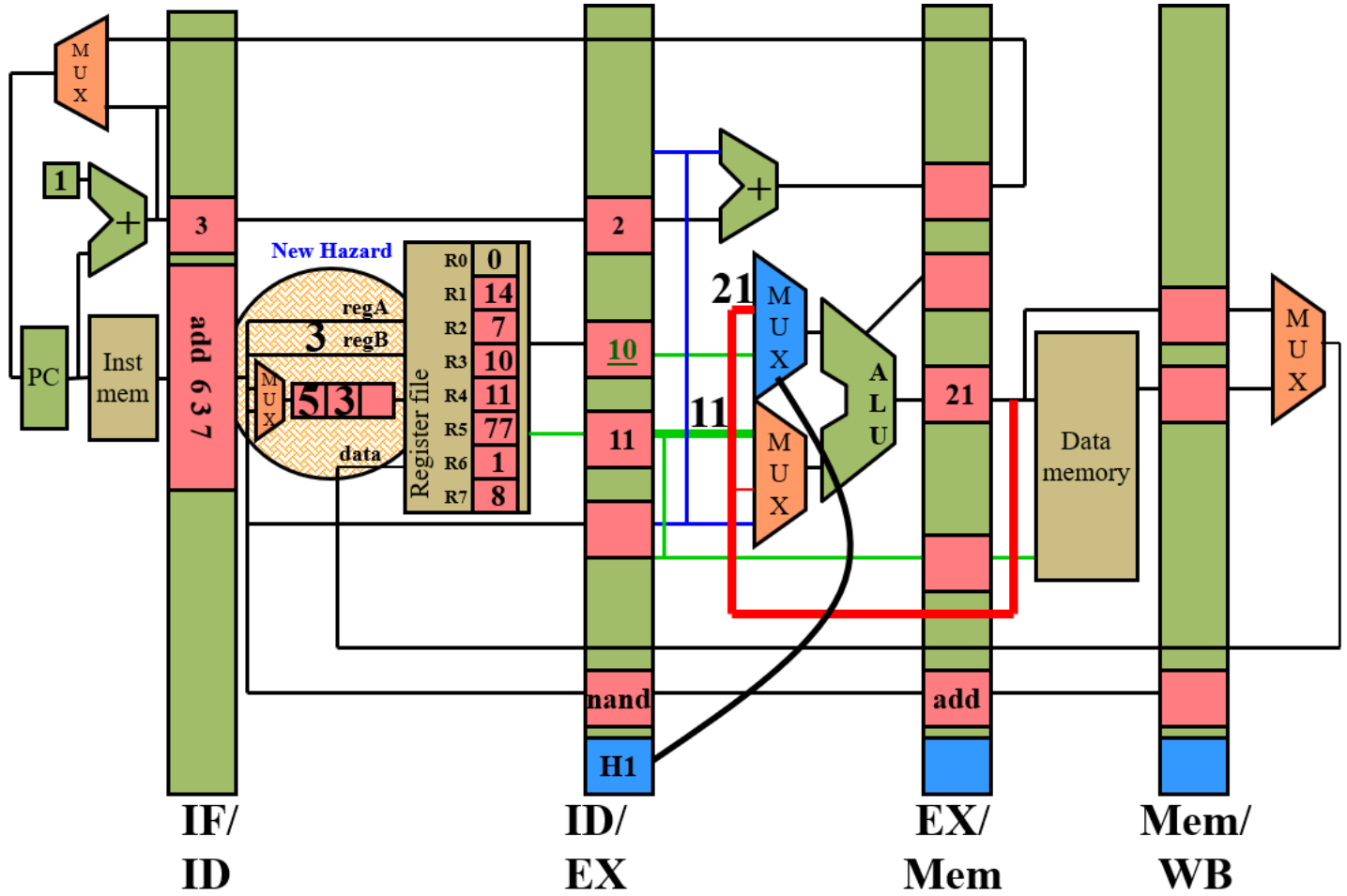


问题1：Data Hazards

- **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）

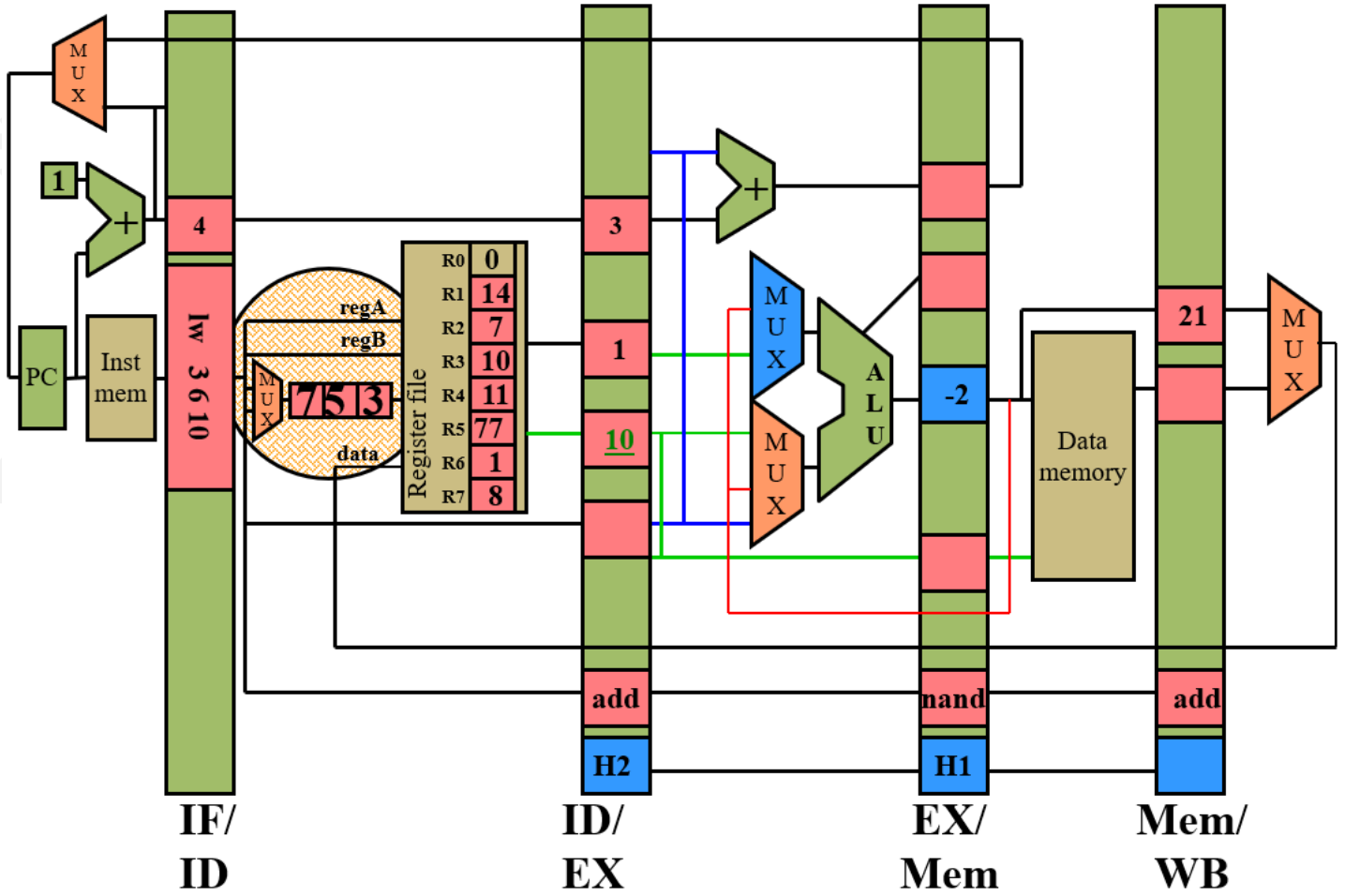


问题1：Data Hazards

- **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

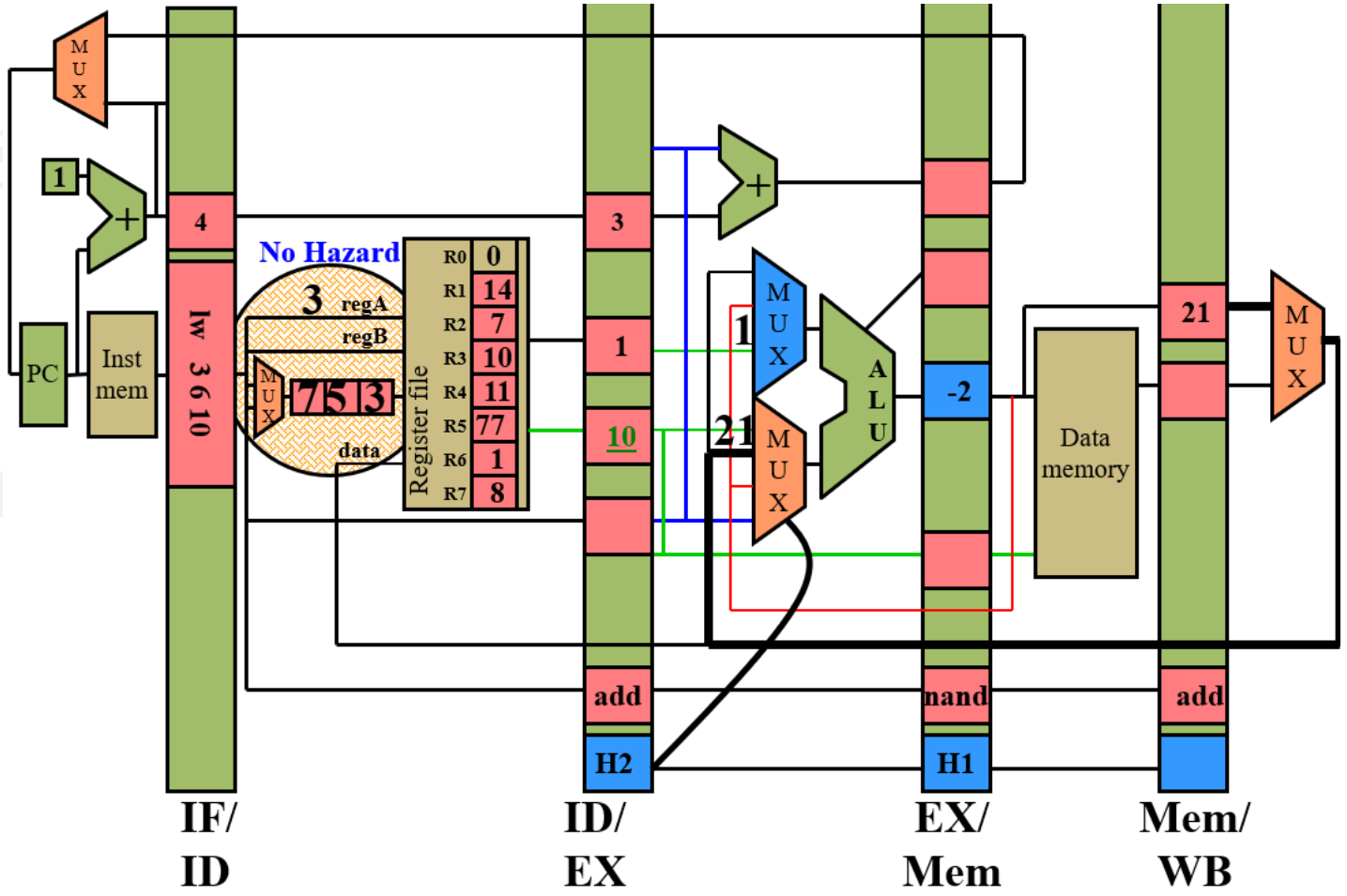


问题1：Data Hazards

- 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

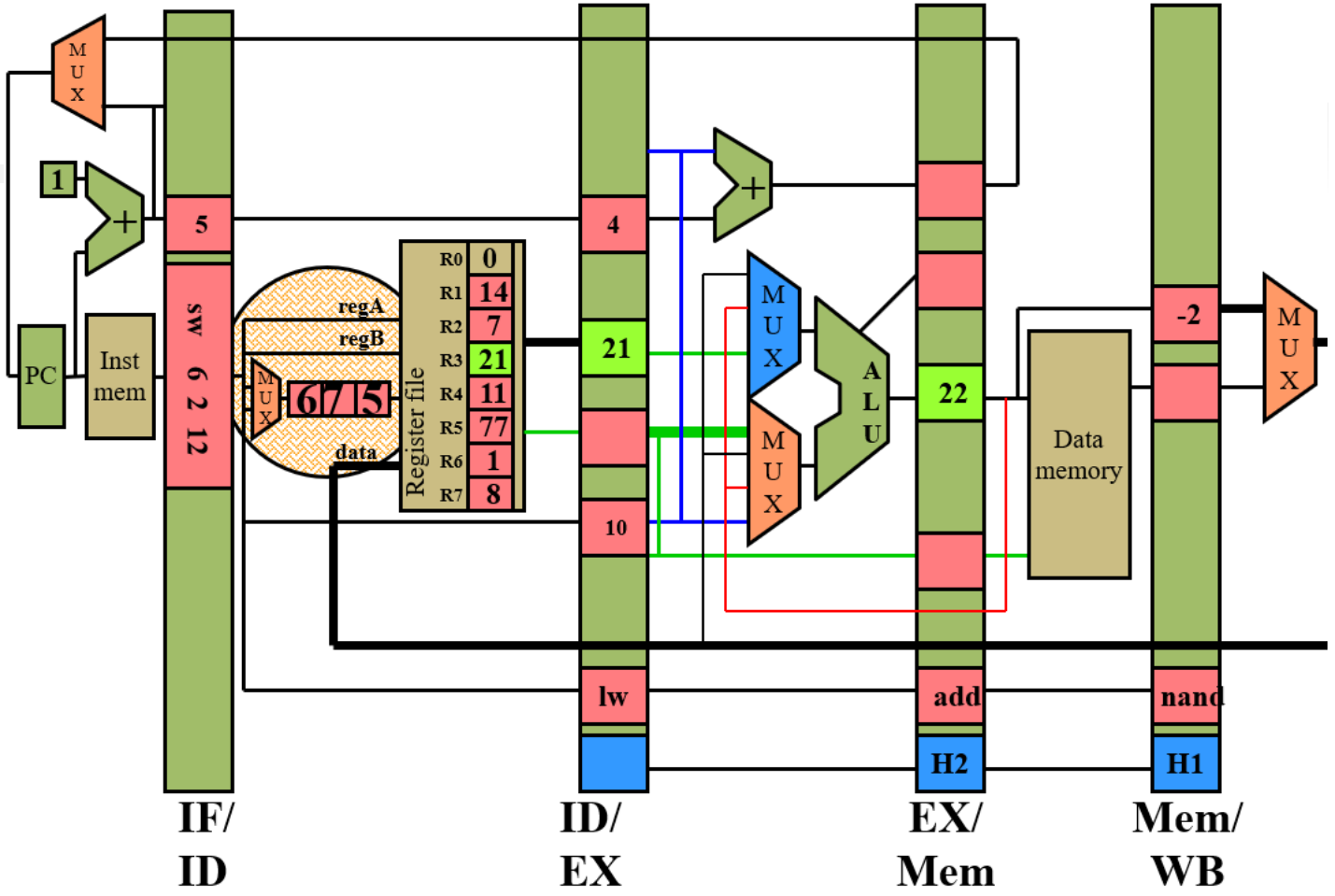


问题1：Data Hazards

- 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

北京大学 PEKING UNIVERSITY

- **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

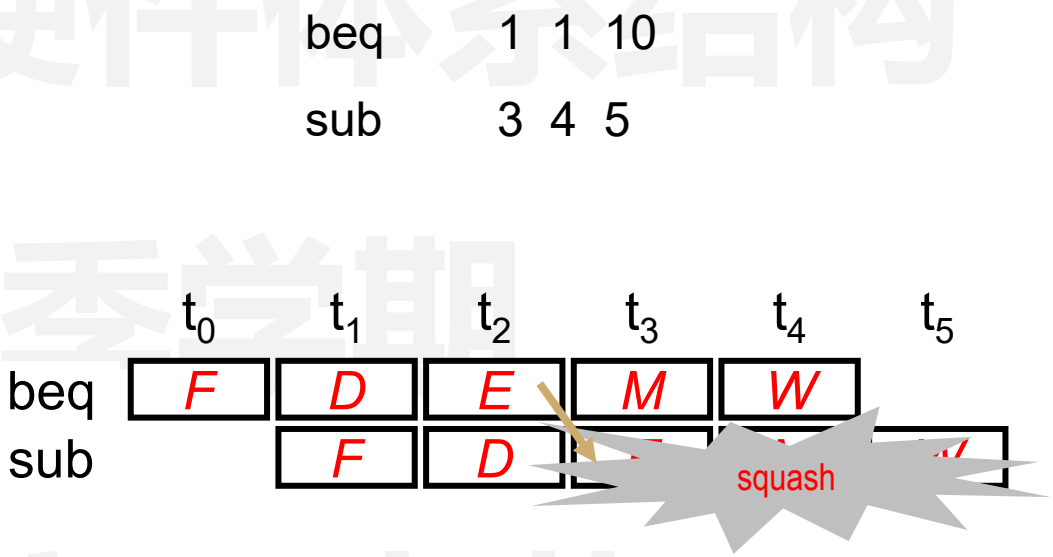
    R1=R4+R5

- 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**

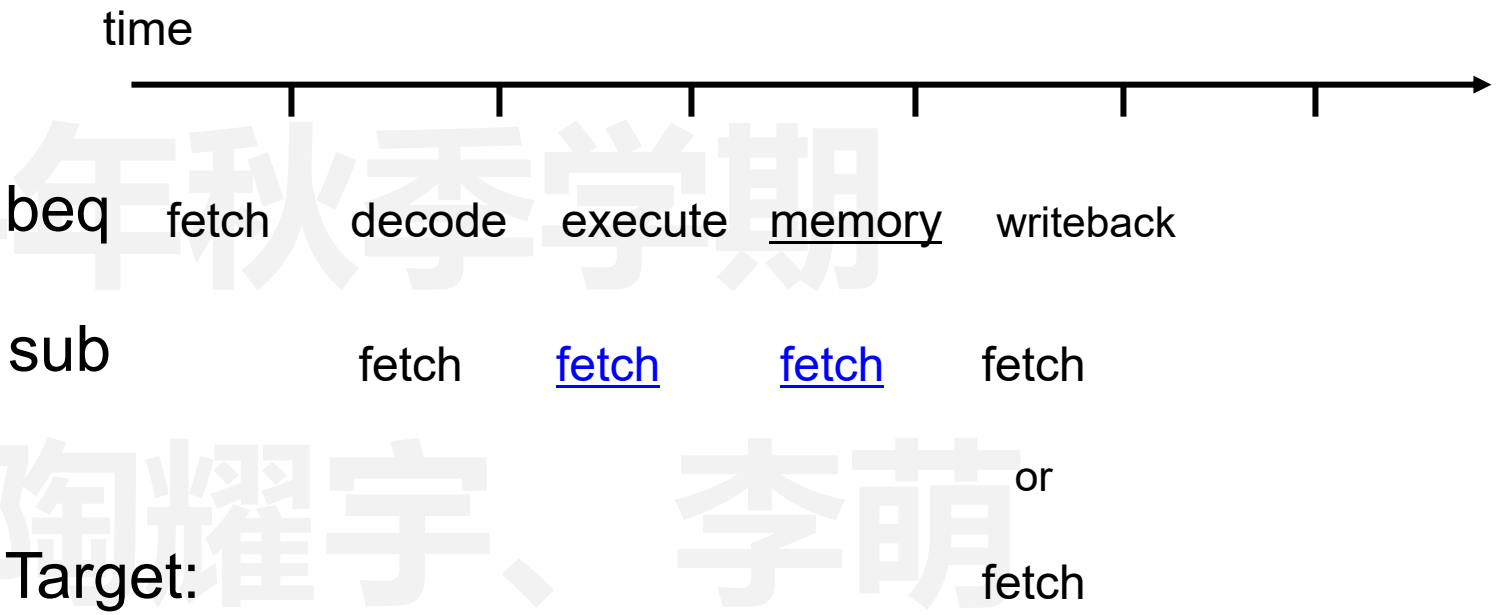
Detection

- 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

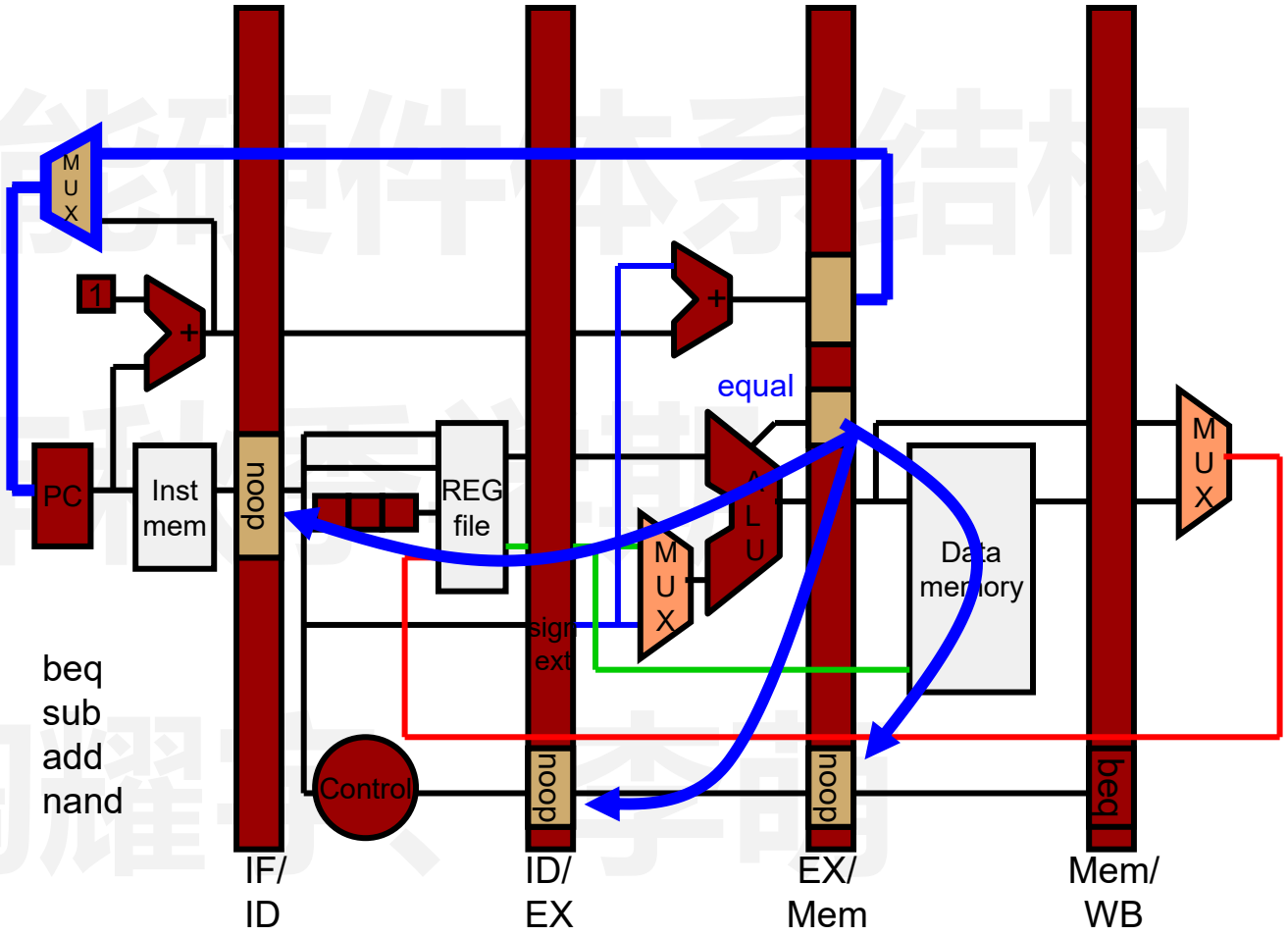
- **Speculate and Squash：投机-制止机制**

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:

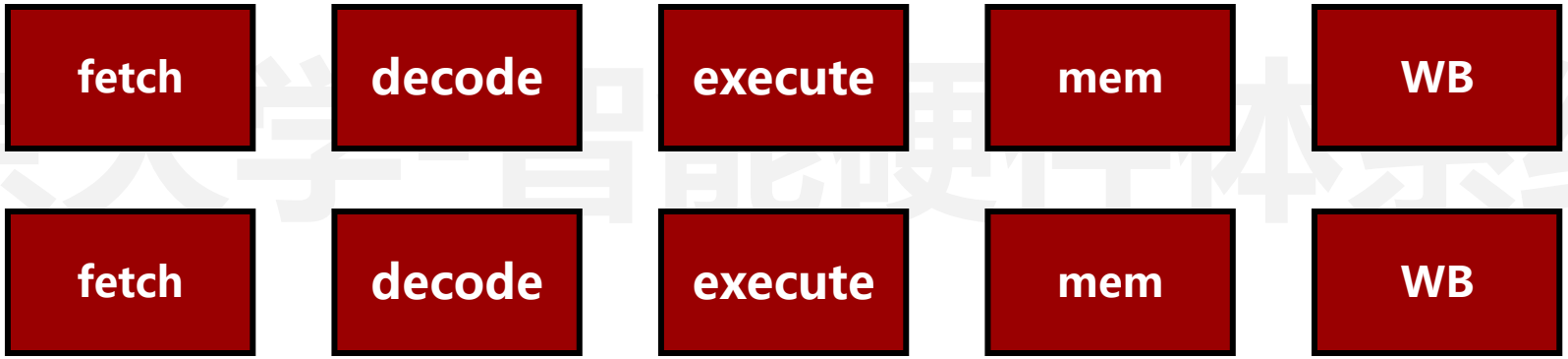
1、Upper Bound on Scalar Pipeline Throughput

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

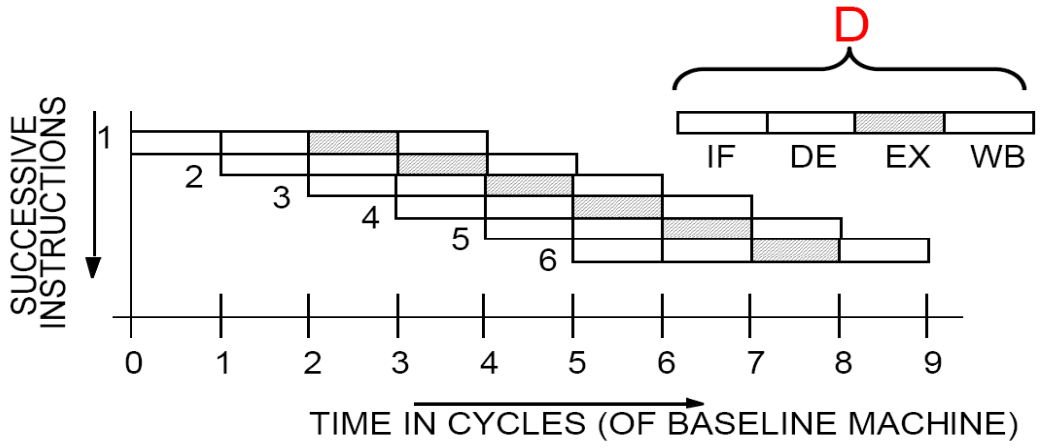
Number of instructions being worked on

Scalar Pipeline (baseline)

Instruction Parallelism = D

Operation Latency = 1

Peak IPC = 1



Peak IPC

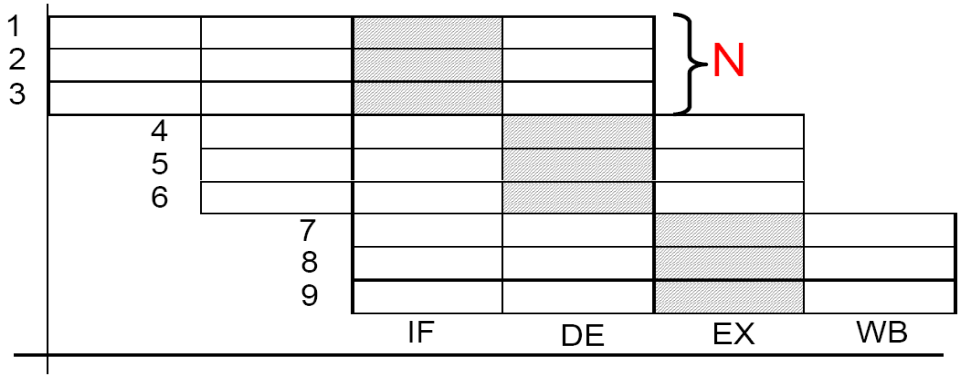
The maximum sustainable number of instructions that can be executed per clock.

Superscalar (Pipelined) Execution

IP = *DxN*

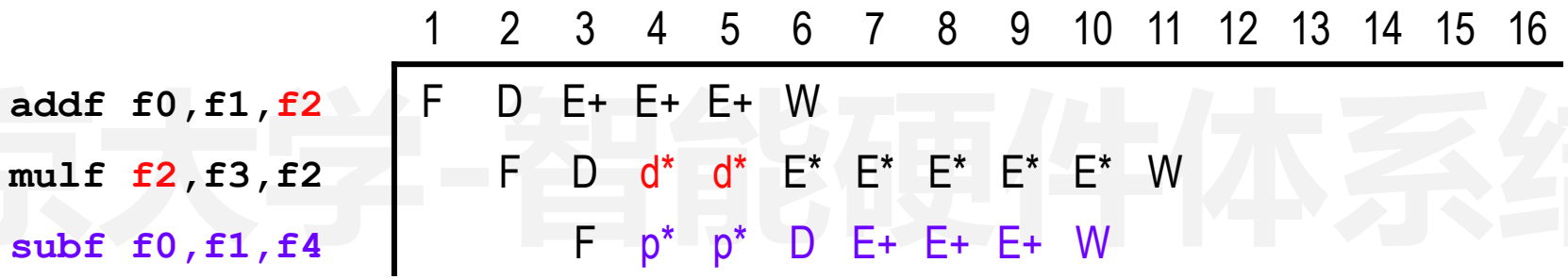
OL = *1 baseline cycles*

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?

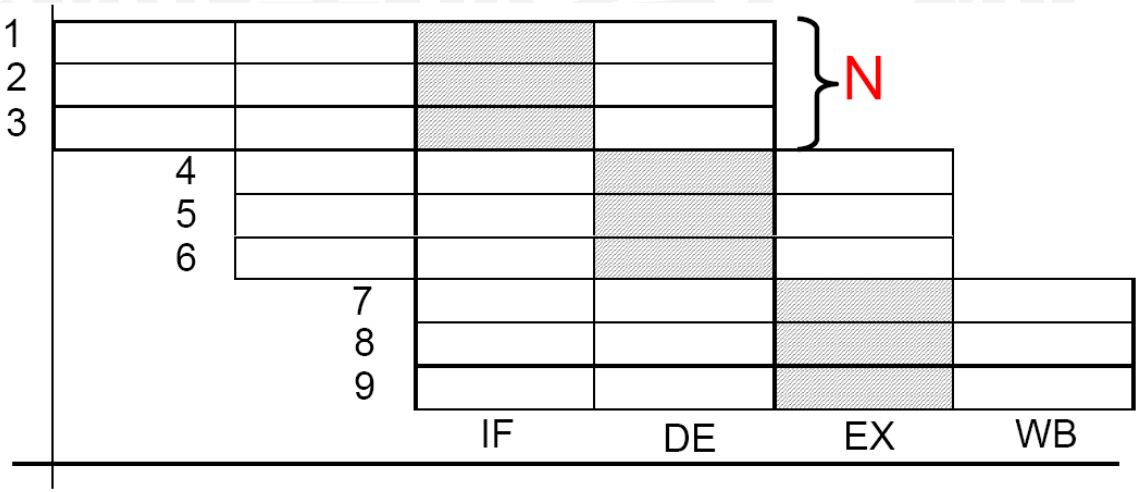
- **`mulf`** stalls due to **RAW hazard**
  - OK, this is a fundamental problem
- **`subf`** stalls due to **pipeline hazard**
  - Why? **`subf`** can’t proceed into D because **`mulf`** is there
  - That is the only reason, and it isn’t a fundamental one

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

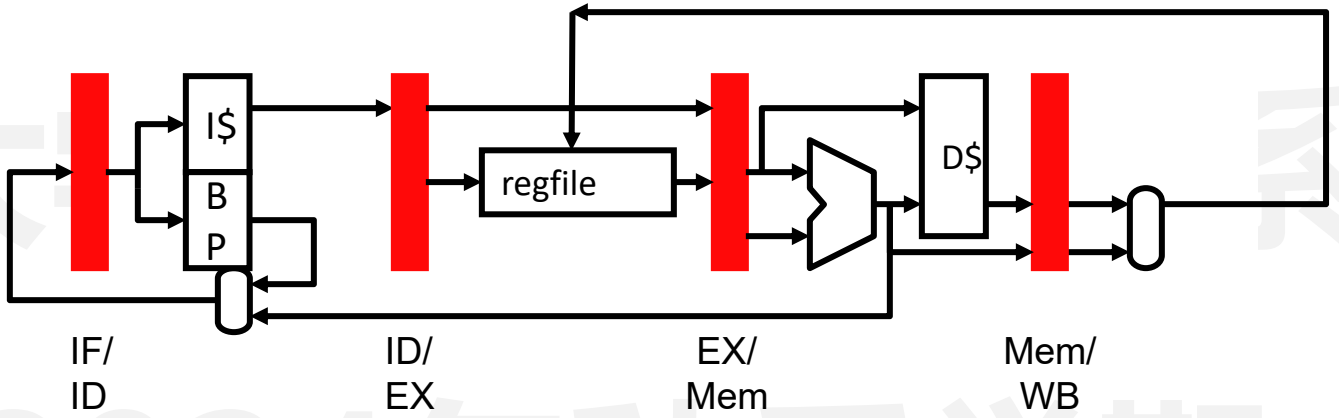
- **Instruction-level parallelism**

- 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

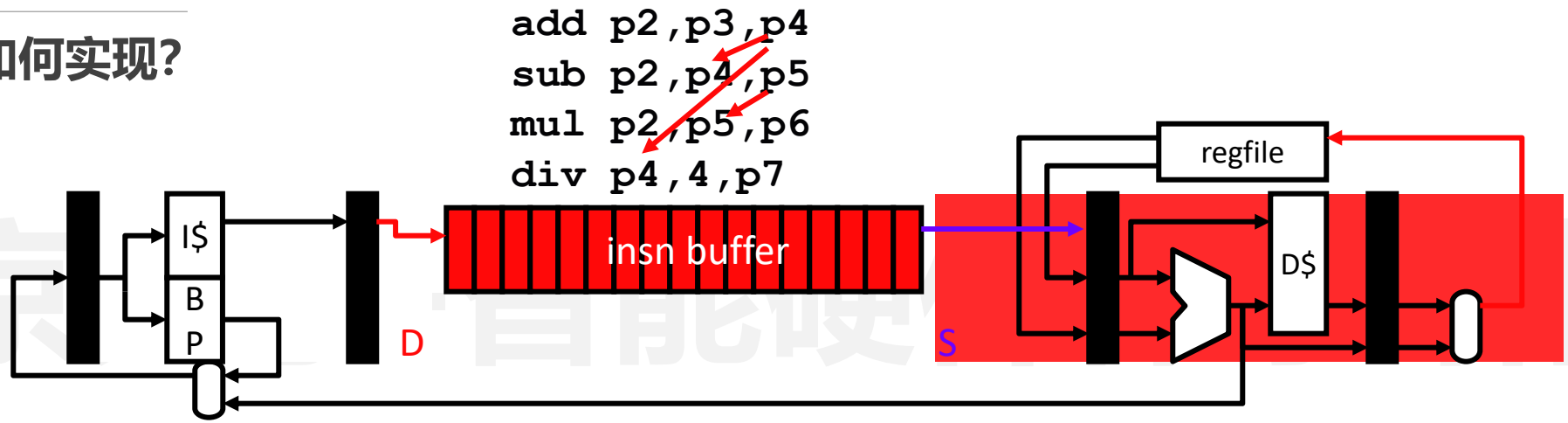
Important

–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

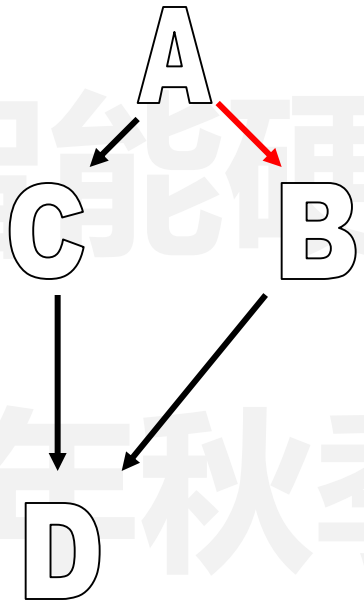
R1=R4+R5

- 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
```

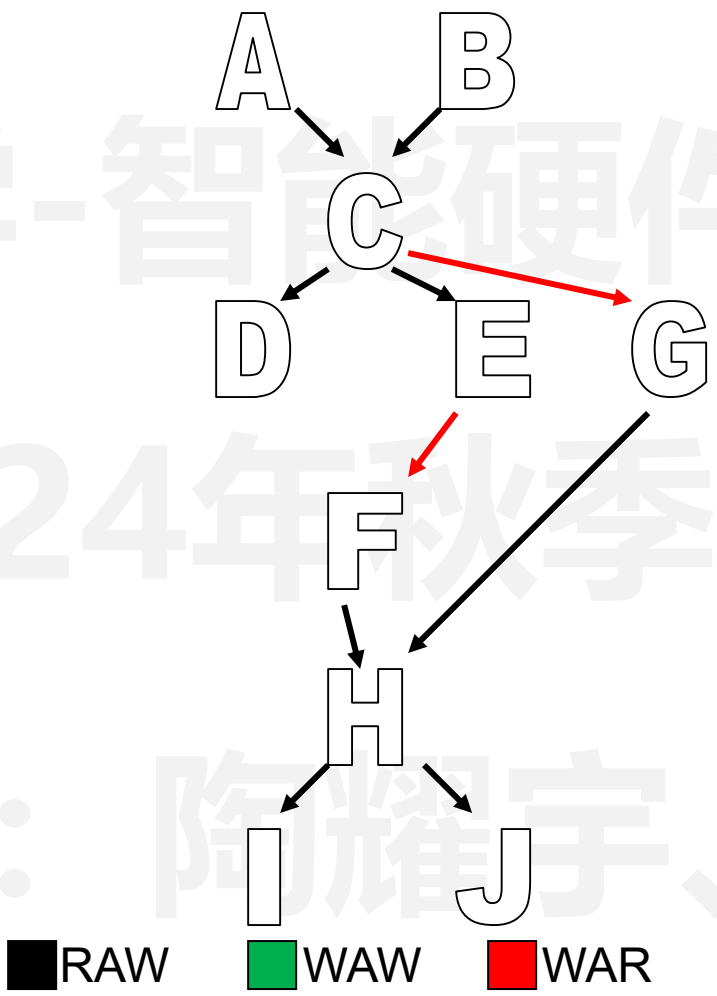


RAW WAW WAR

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
```

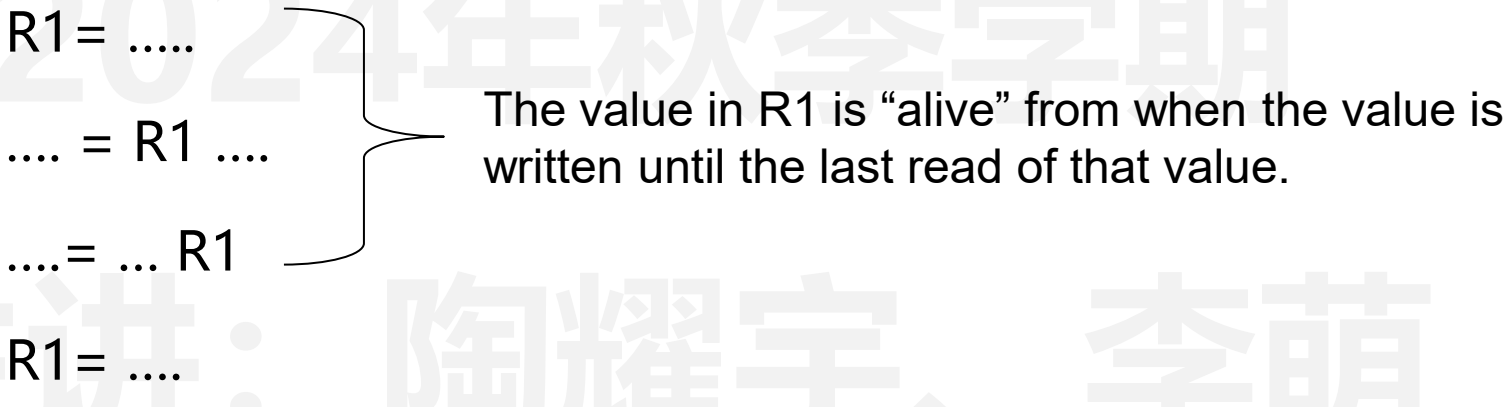


- Well, logically there is no reason for F-J to be dependent on A-E. So…..
  - ABFG
  - CH
  - DEIJ
  - Should be possible.
- But that would cause either C or H to have the wrong reg inputs
- How do we fix this?
  - Remember, the dependency is really on the *name* of the register
  - **So… change the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

寄存器重命名

- 寄存器Register重命名概念

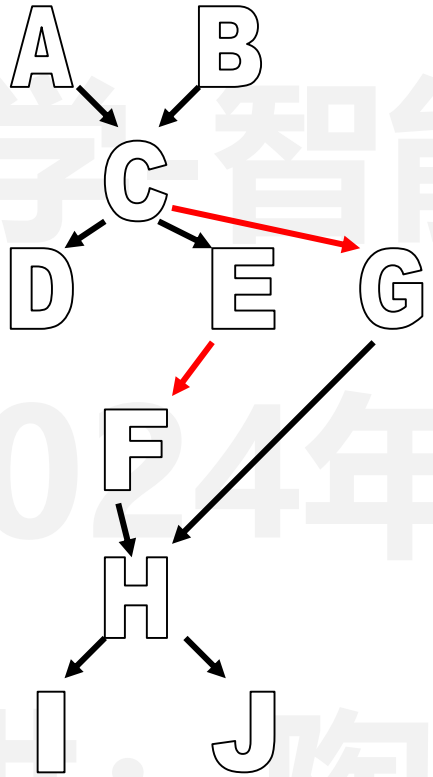
- The register names are arbitrary
- The register name only needs to be consistent between writes.



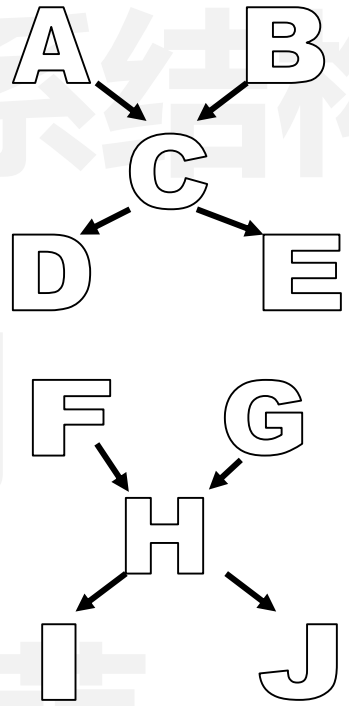
寄存器重命名

- 寄存器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

寄存器重命名

- 寄存器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
虚拟的架构寄存器

Physical register
实际的电路寄存器

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
```