



北京大学
PEKING UNIVERSITY

智能硬件体系结构

第五讲：指令集与流水线架构-2



主讲：陶耀宇、李萌

2024年秋季

注意事项

• 课程作业情况

- 第2次作业将在**10月17日**放出
- 第1次编程作业（简化中）预计**10月20号~11月20号**

2024年秋季学期

主讲：陶耀宇、李萌

目 录

CONTENTS



01. 指令集架构基础

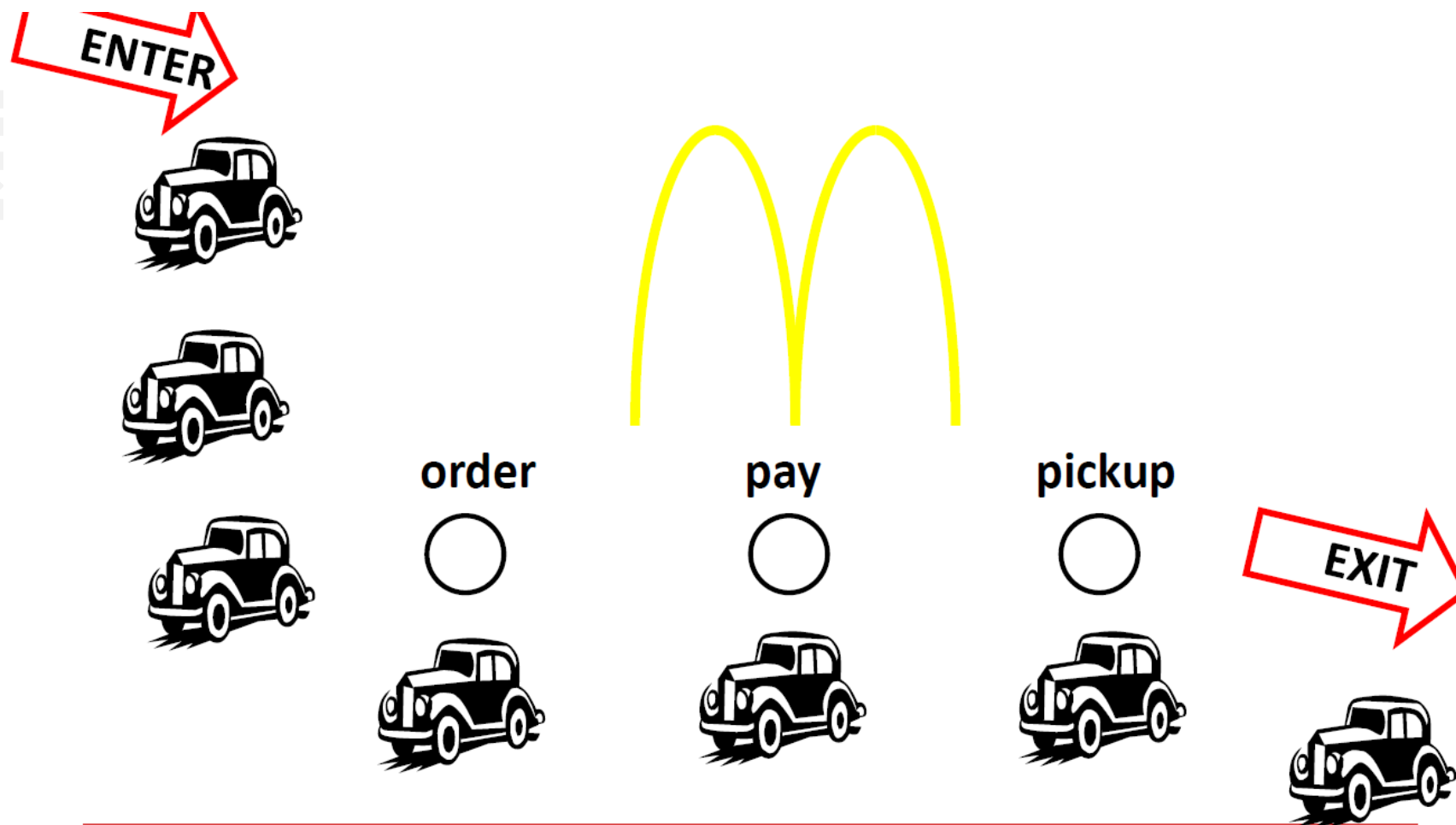
02. 指令集设计基础

03. 流水线架构基础

04. 流水线架构优化

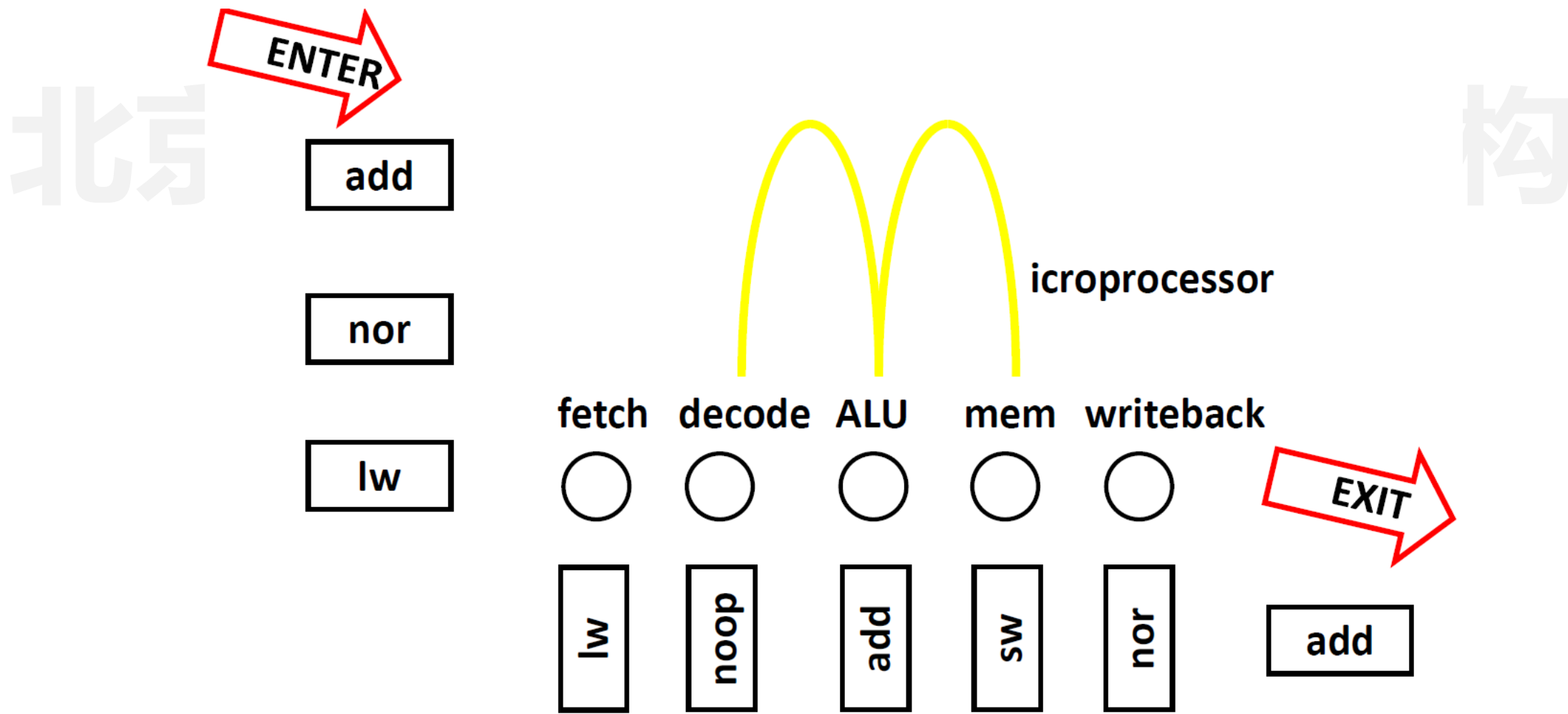
回顾：什么是流水线架构

- ### • 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)



- 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

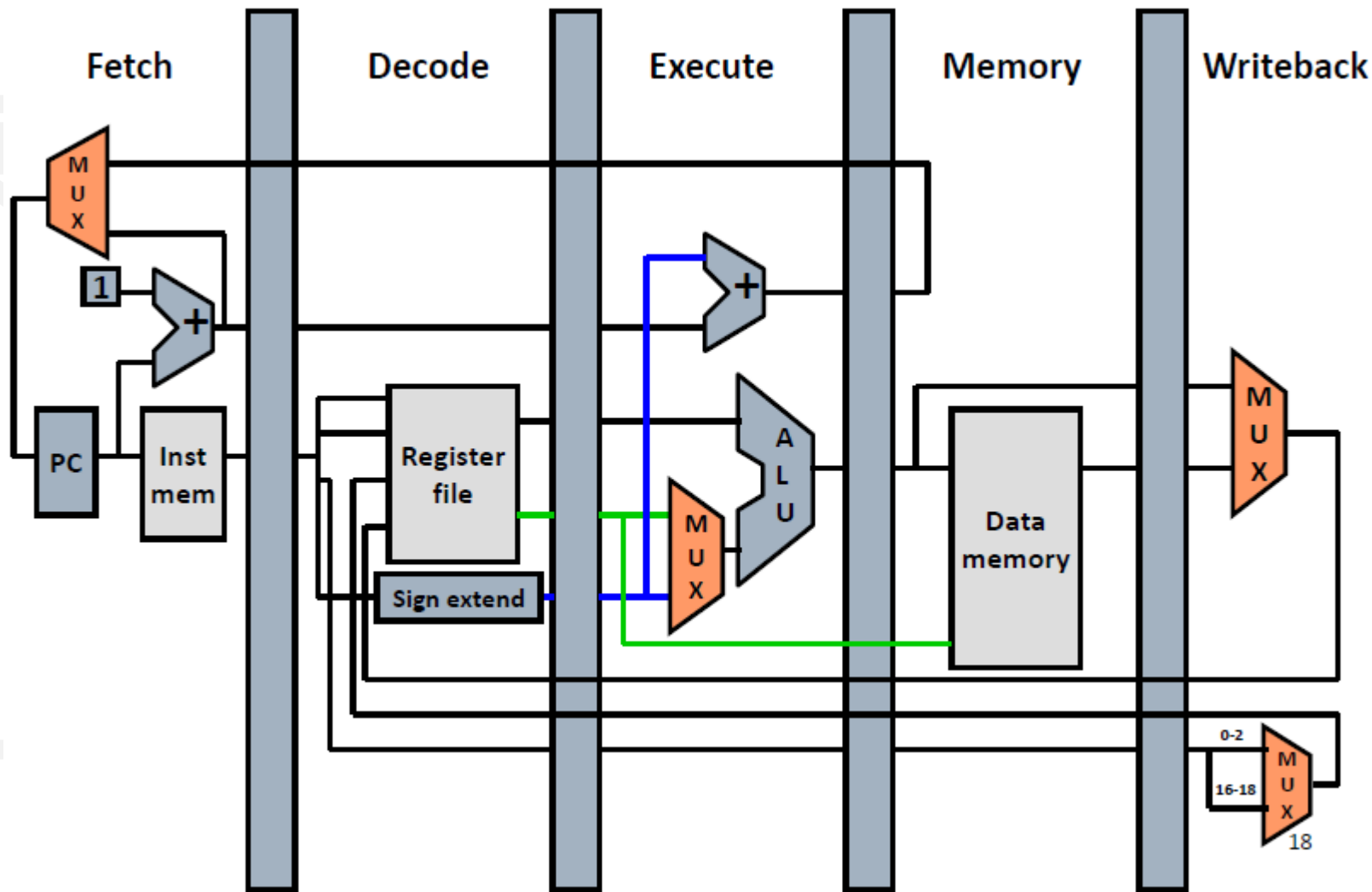
- 4 pipelines, 20 cycles deep
- Prescott: 4 pipelines, 31 cycles deep (could be clocked up to 8 GHz with special cooling)

- ☐ Core i7 (Nehalem)

- 4 pipelines, 16 cycles deep

最基本的单条流水线设计示意图

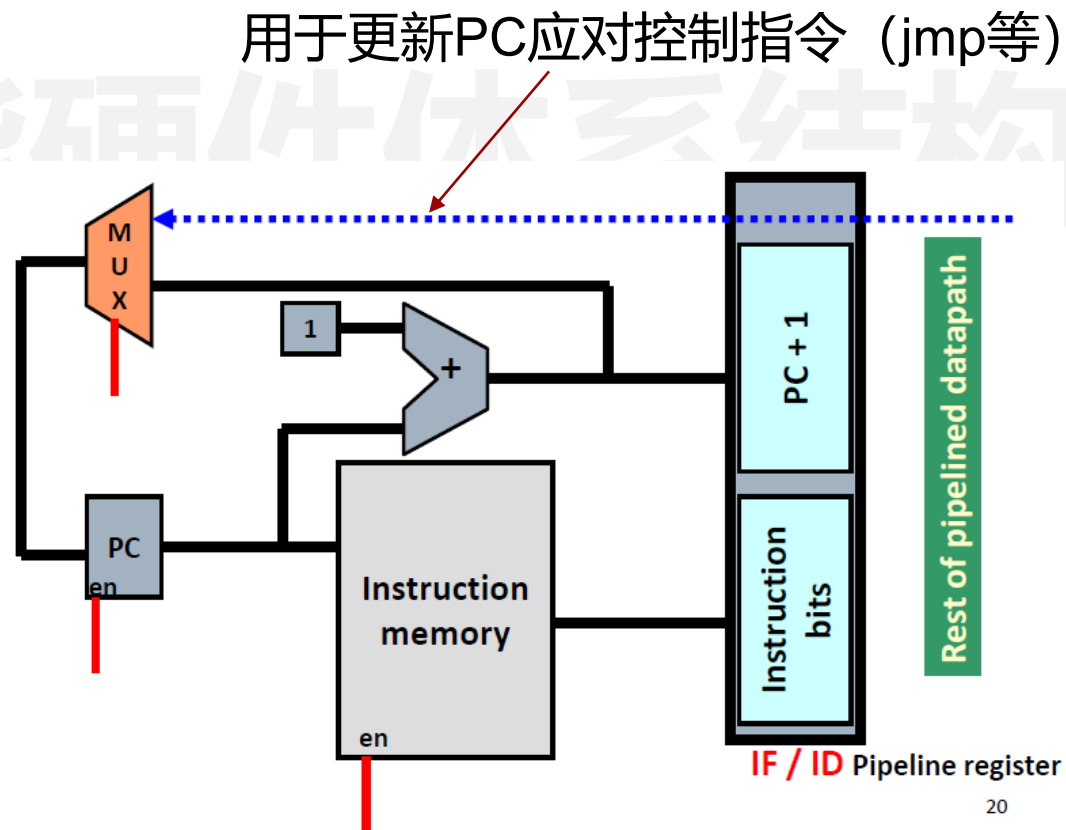
- 5级流水线设计: Fetch、Decode、Execute、Memory、Writeback



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

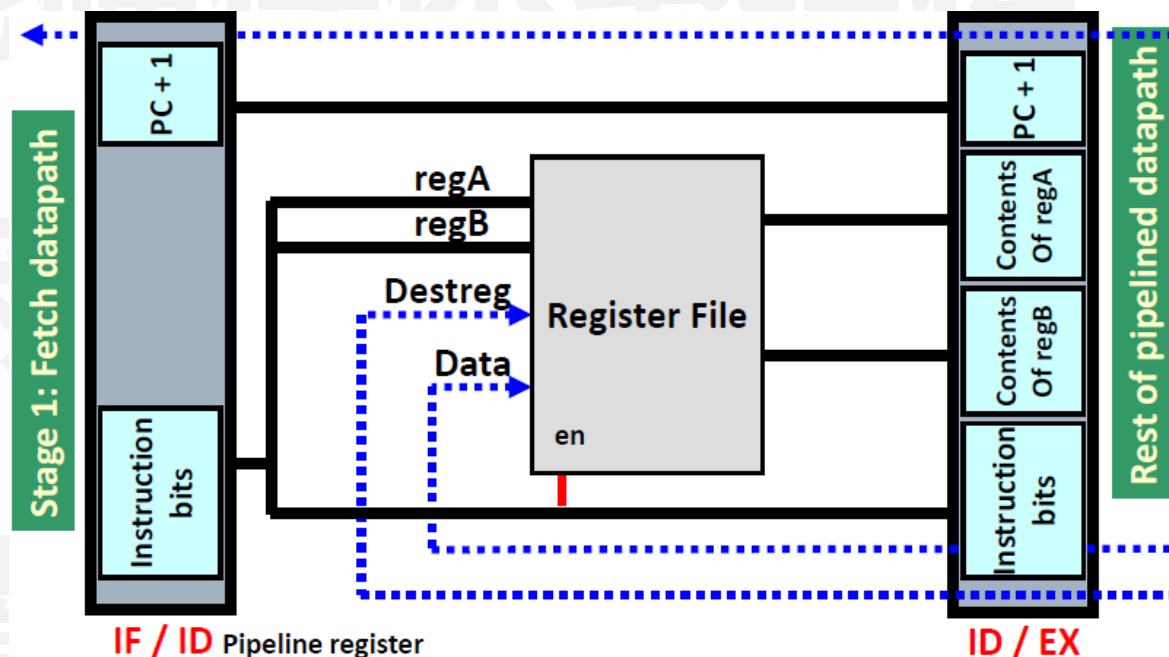


第二级：Decode指令

- 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

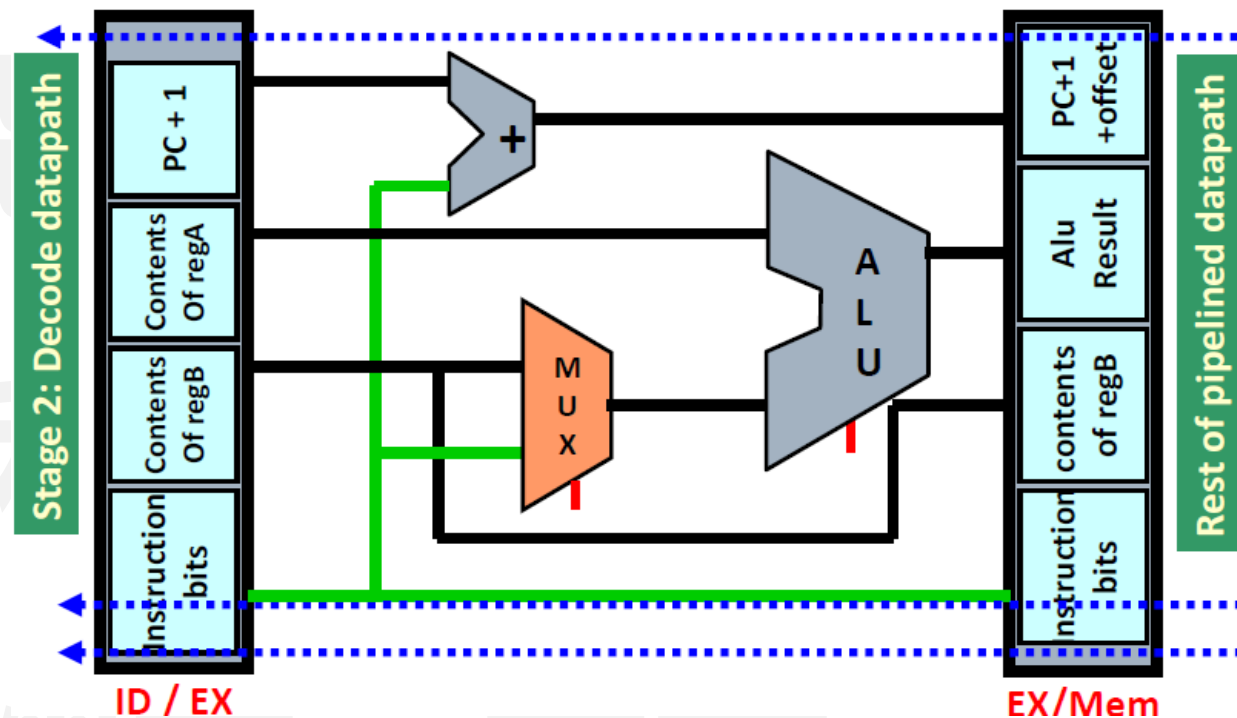
- Design a datapath that reads the IF/ID pipeline register, decodes instruction and reads register file (specified by regA and regB of instruction bits).
 - Decode is easy, just pass on the opcode and let later stages figure out their own control signals for the instruction.
- Write everything needed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

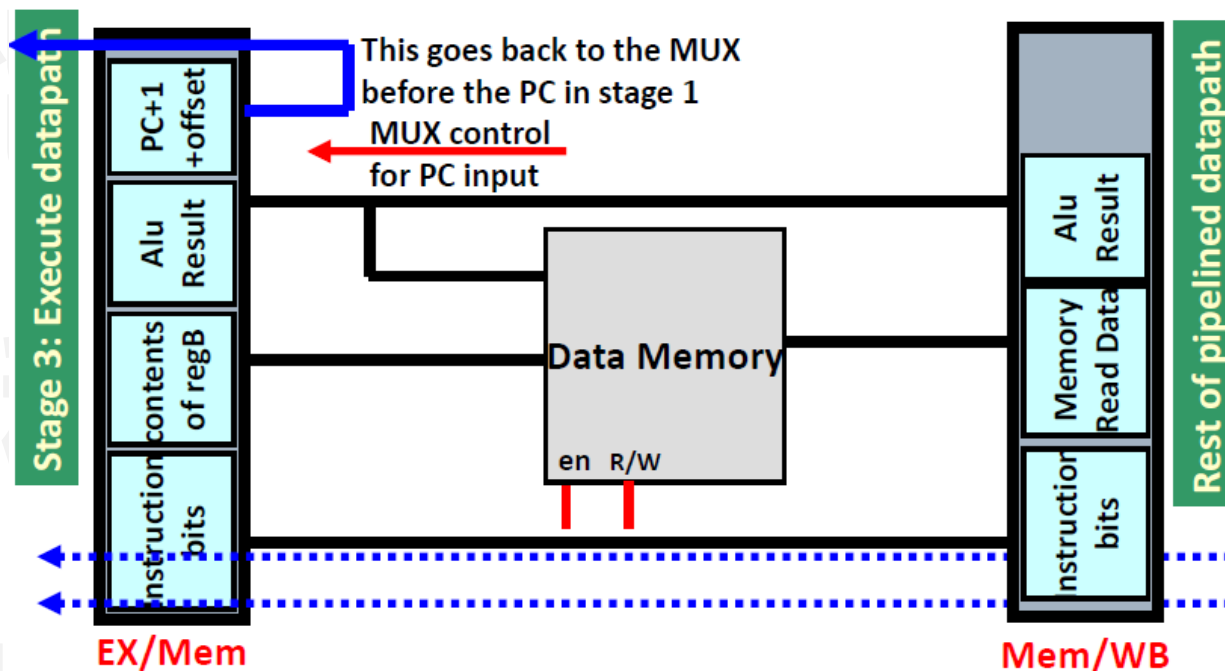


主讲：陶耀宇、李明

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

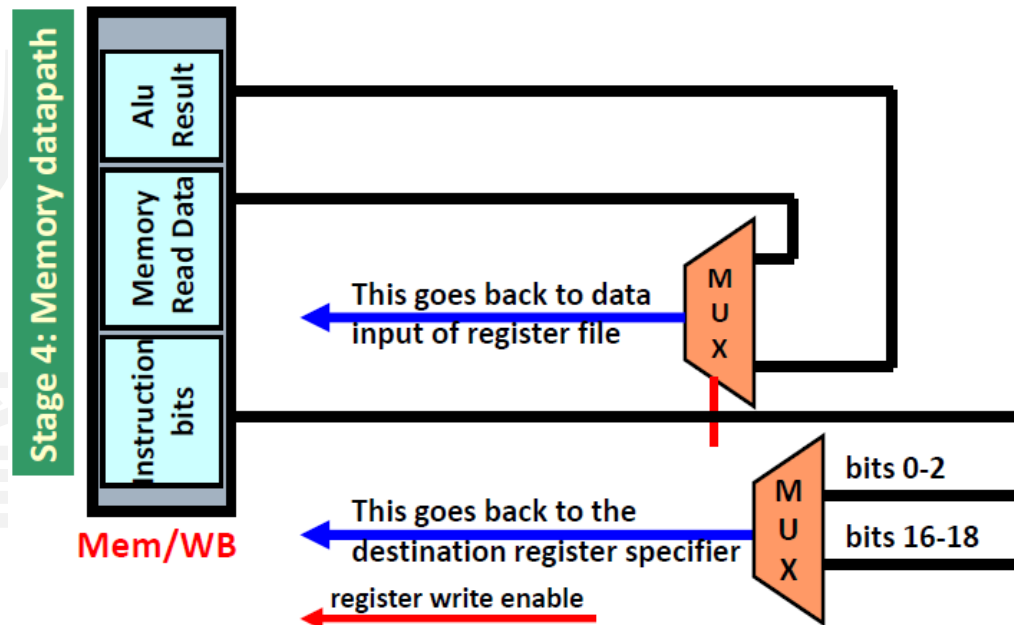


主讲：陶耀宇、李明

第五级：Writeback操作

- ALU计算结果或Data Memory读取的结果写回destReg

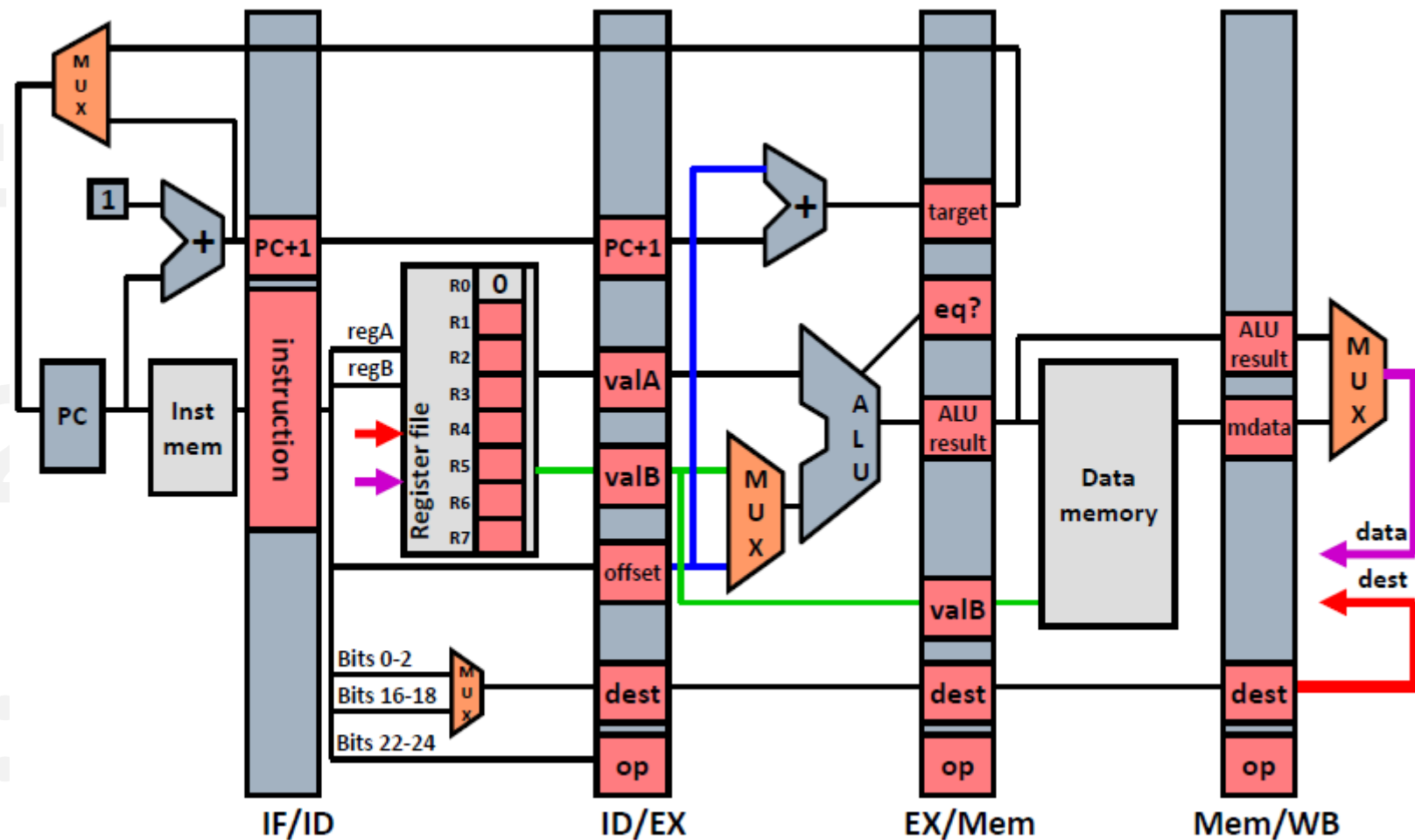
- Design a datapath that completes the execution of this instruction, writing to the register file if required.
 - Write MemData to destReg for ld instruction
 - Write ALU result to destReg for add or nand instructions.
 - Opcode bits also control register write enable signal.



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

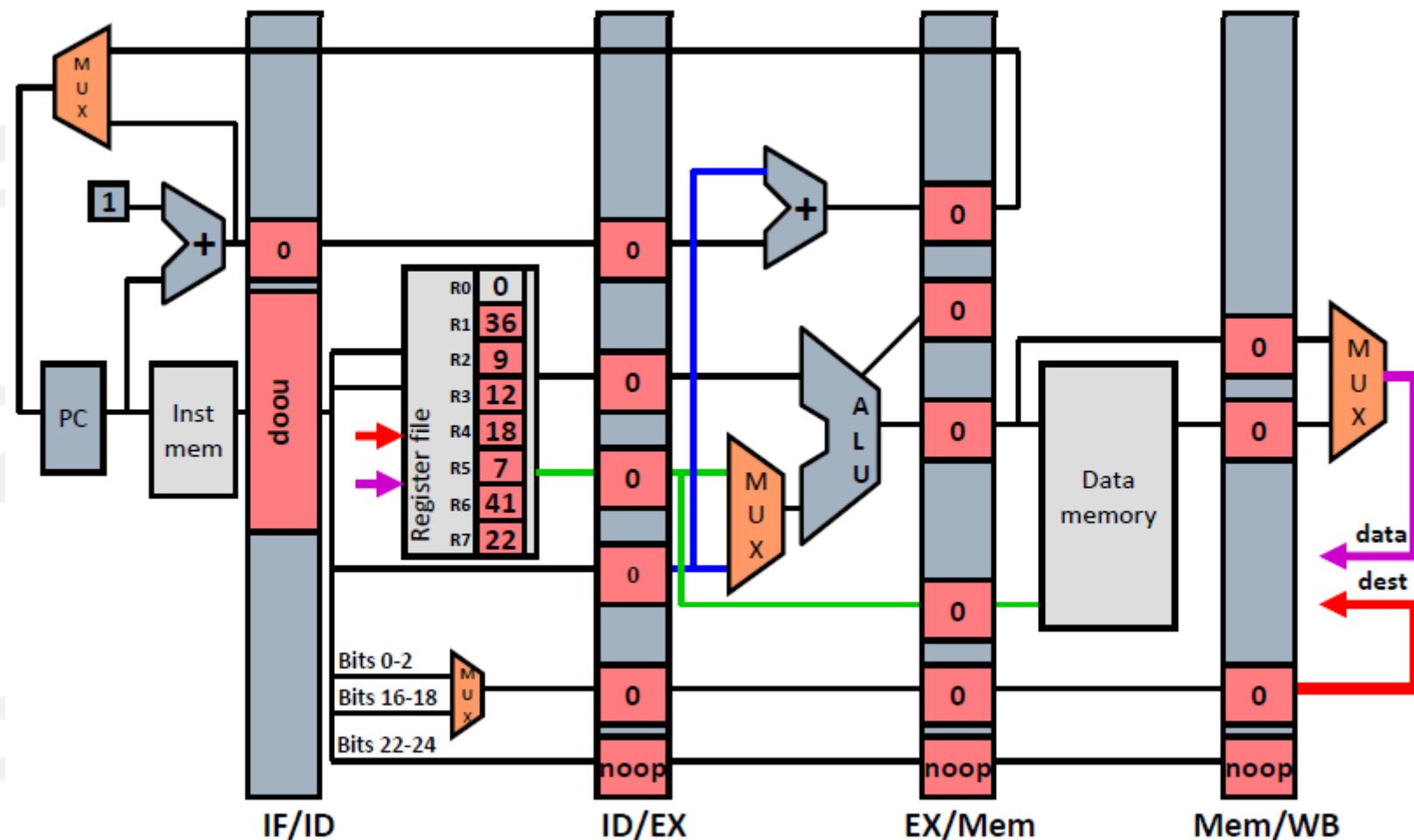
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10]=reg 7



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

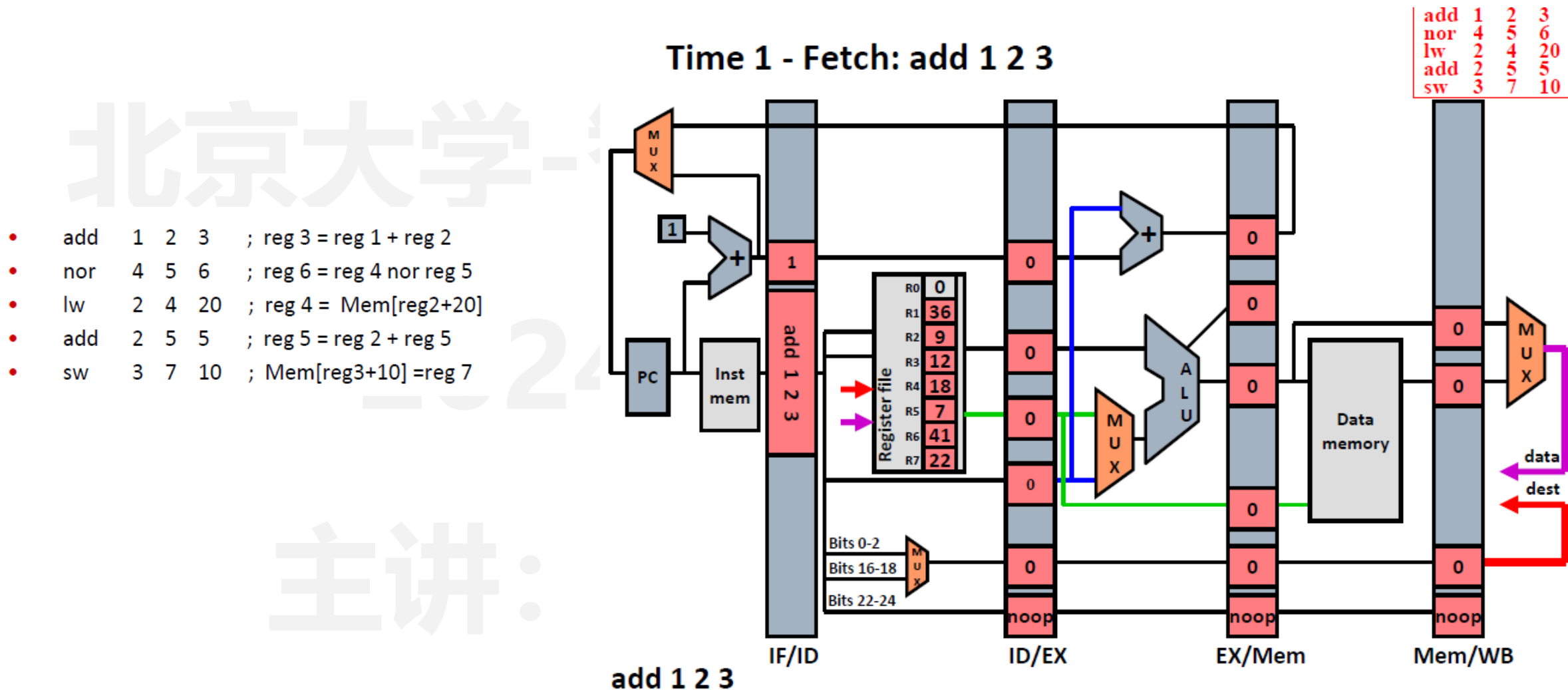
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10]=reg 7



初始状态: t_0

5级流水线的实际案例

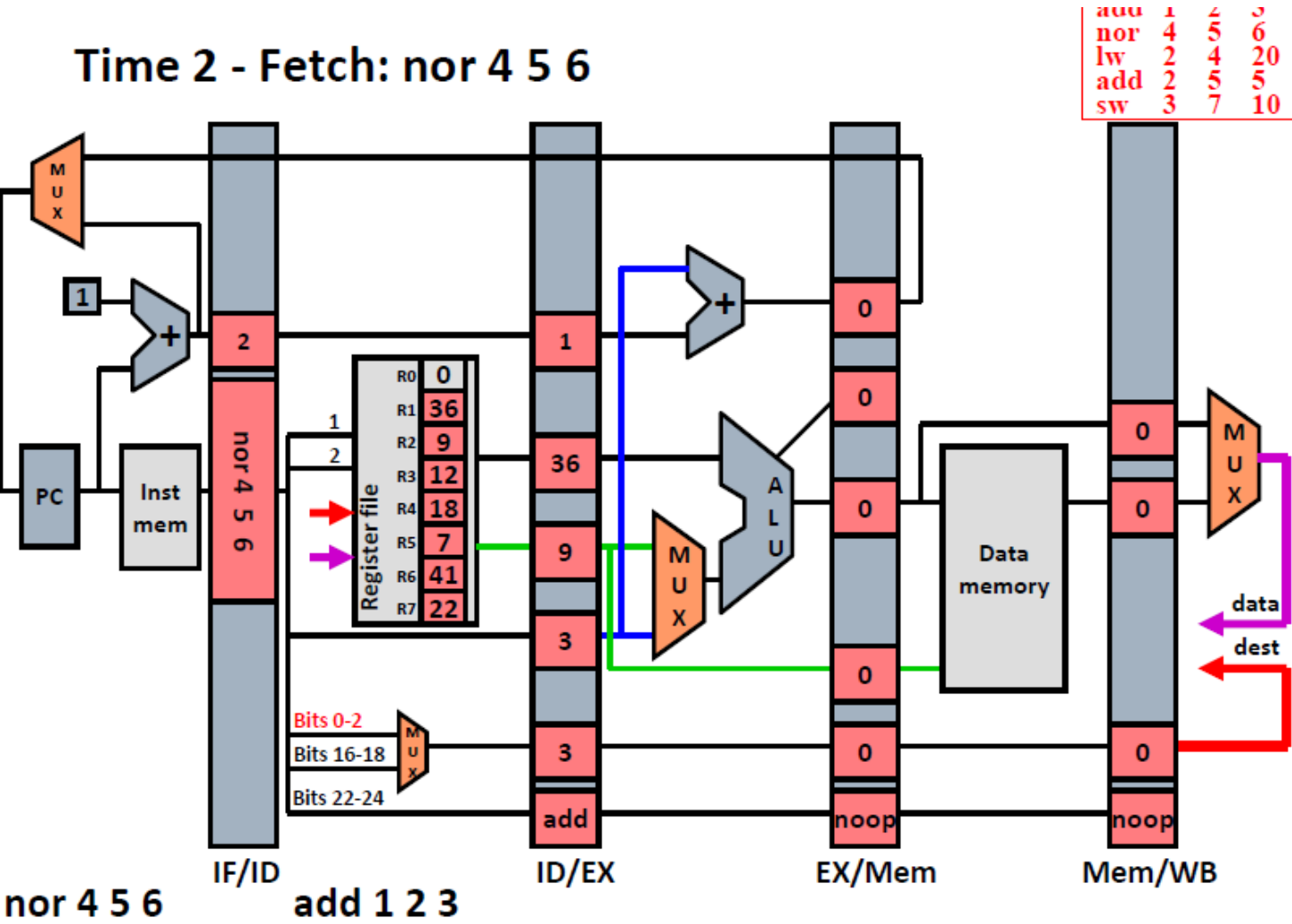
- 假设运行以下指令在5级流水线上



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

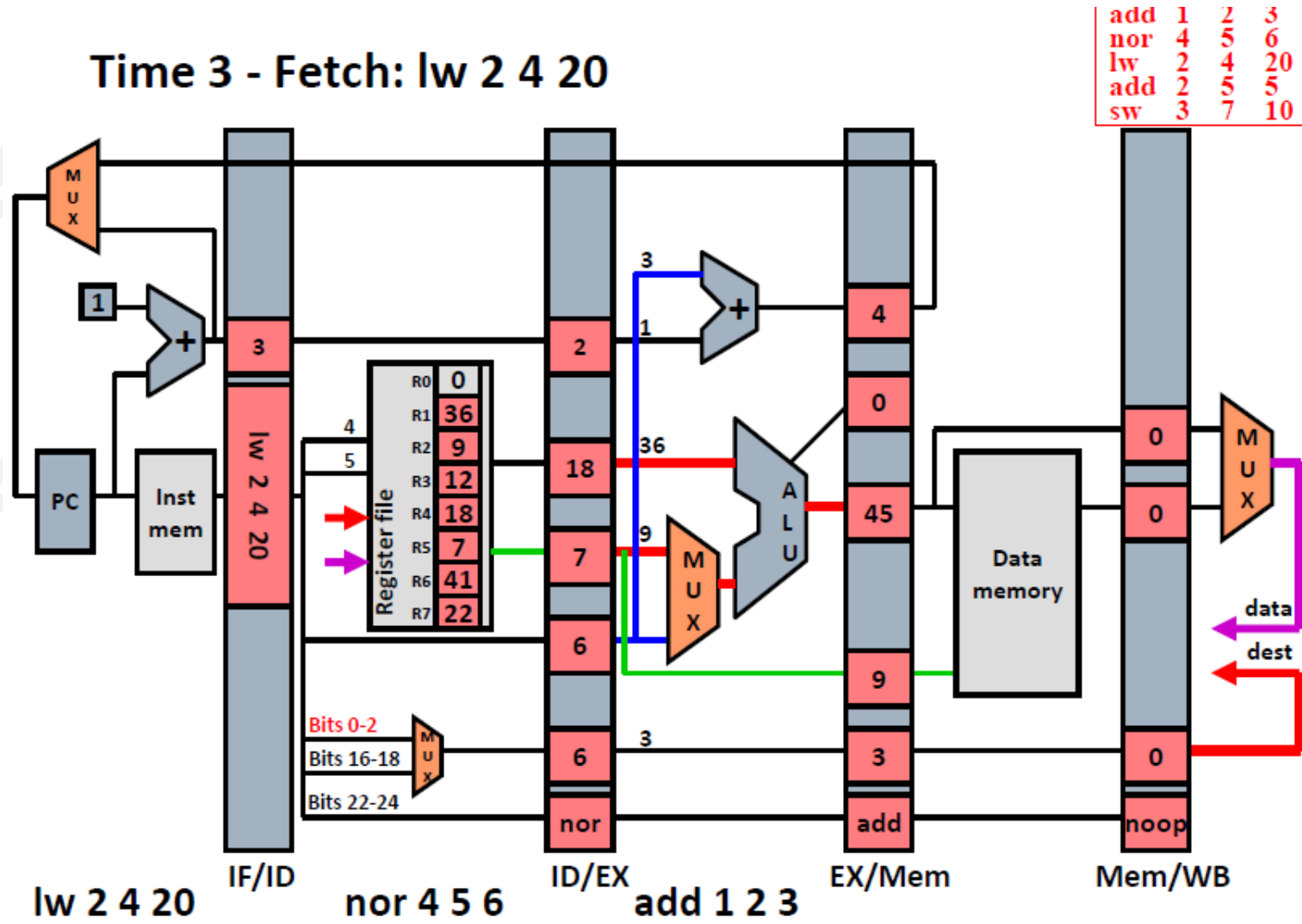
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10]=reg 7



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

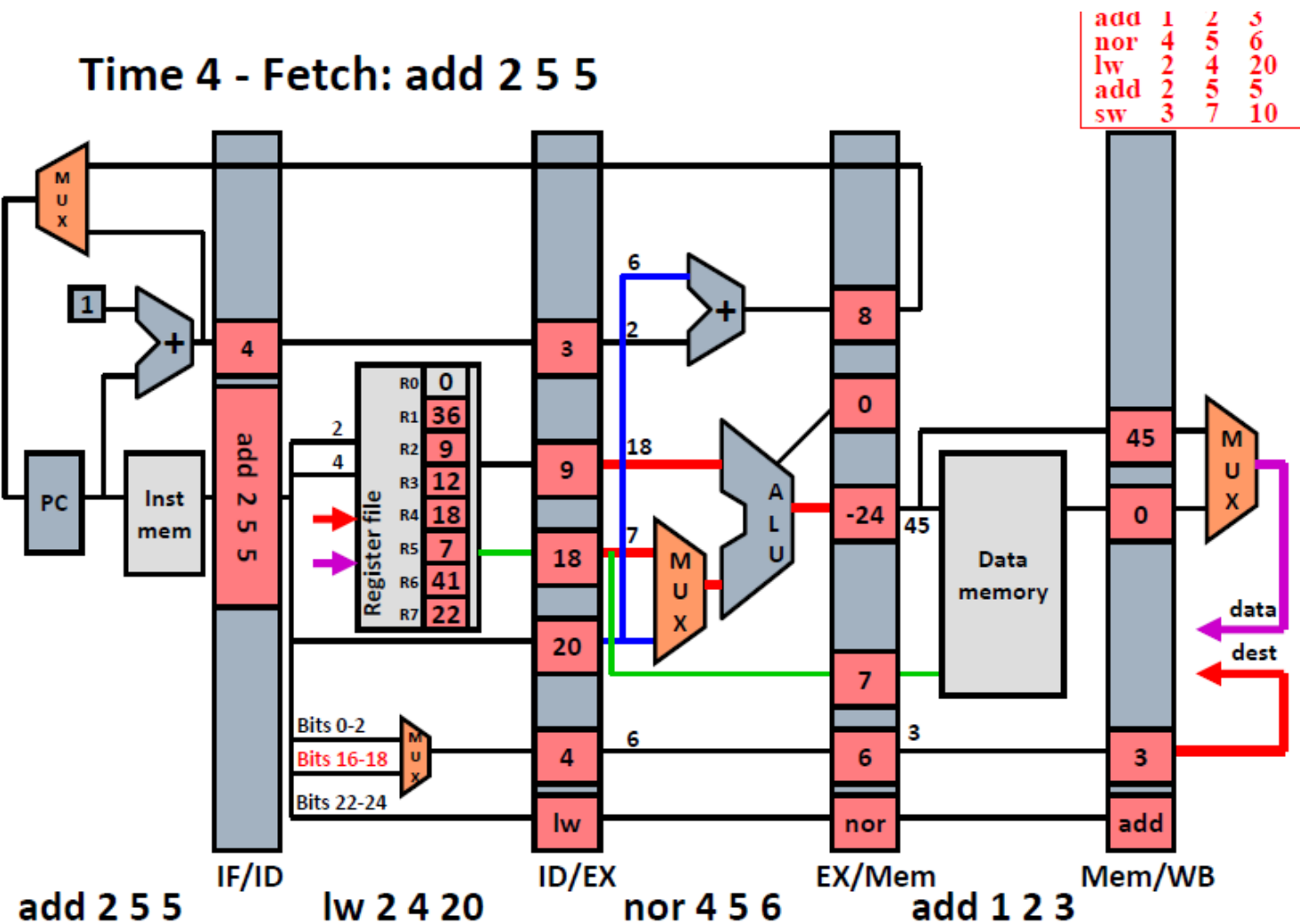
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10]=reg 7



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

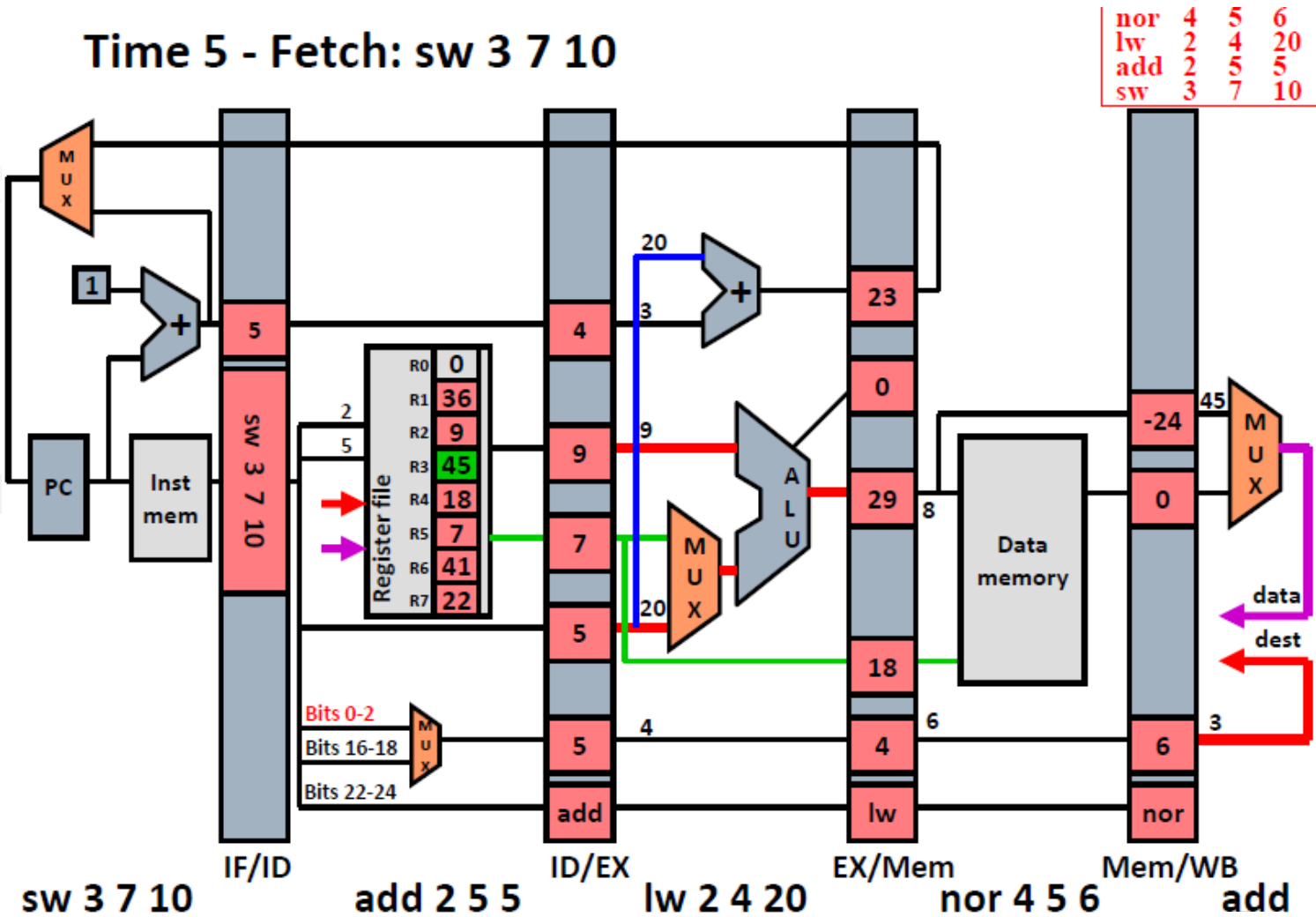
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10]=reg 7



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

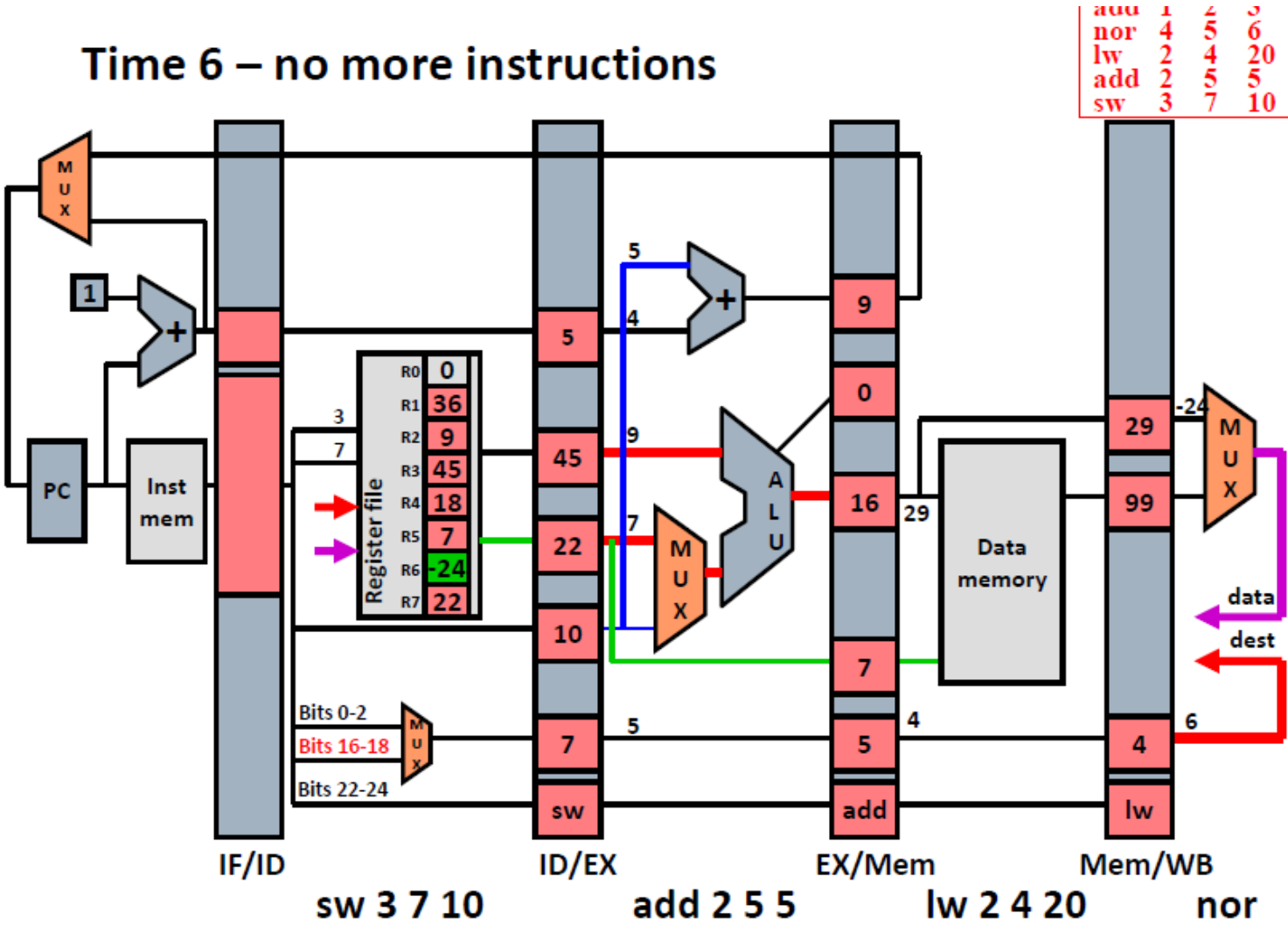


5级流水线的实际案例

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

主讲:



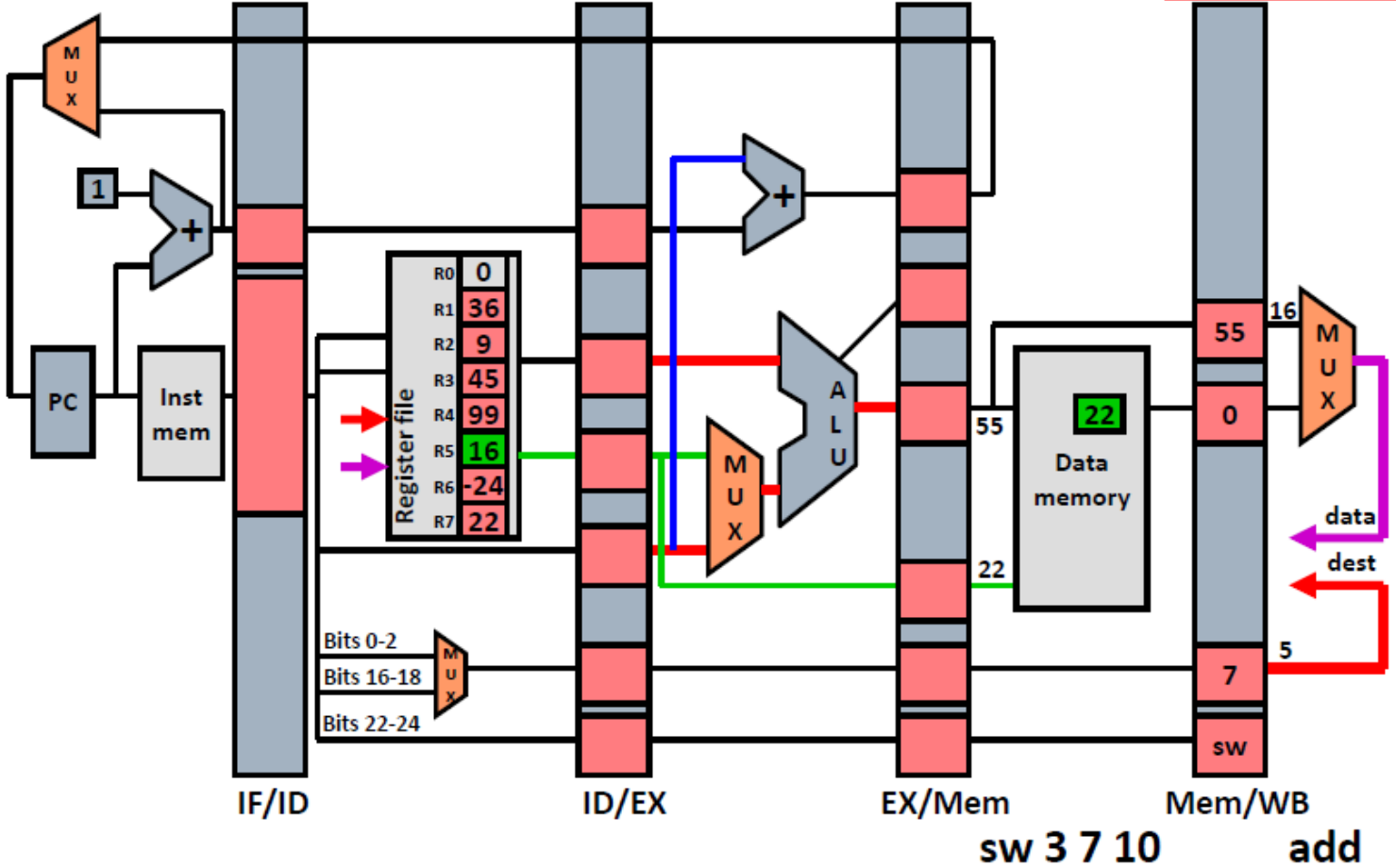
- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10]=reg 7

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

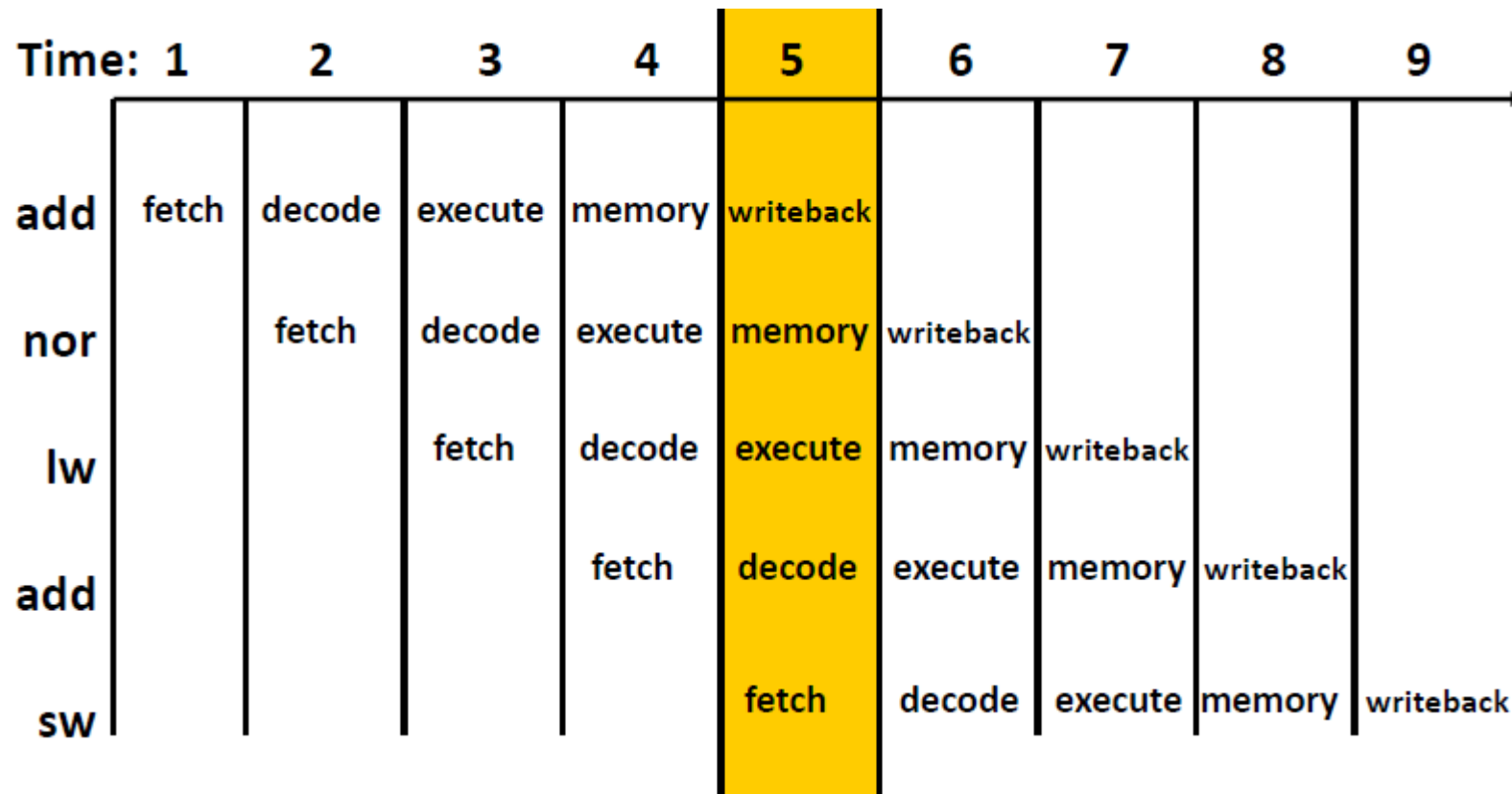


- add 1 2 3 ; reg 3 = reg 1 + reg 2
- nor 4 5 6 ; reg 6 = reg 4 nor reg 5
- lw 2 4 20 ; reg 4 = Mem[reg2+20]
- add 2 5 5 ; reg 5 = reg 2 + reg 5
- sw 3 7 10 ; Mem[reg3+10]=reg 7

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



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级流水线可能存在的问题？

- **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

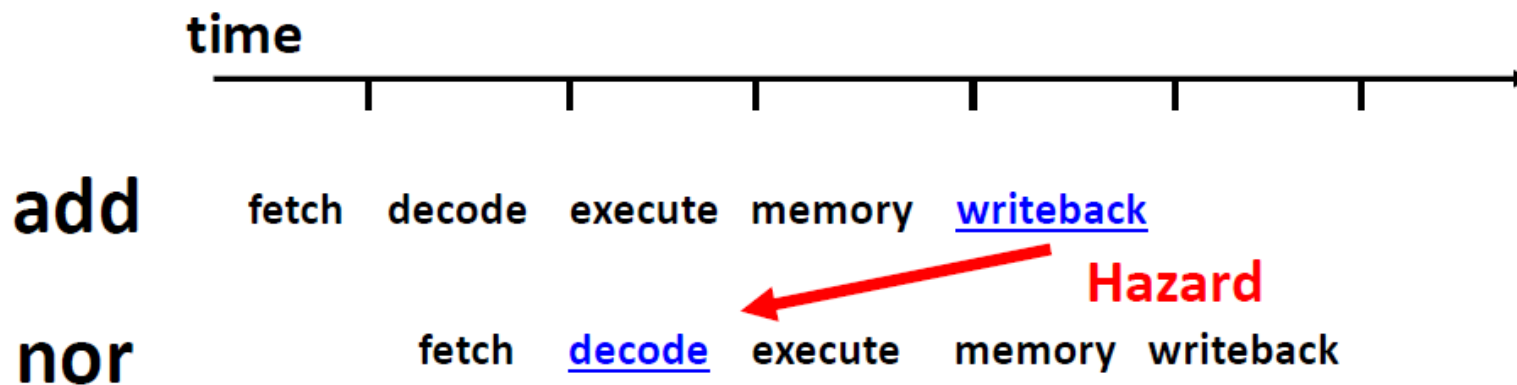
问题1: Data Hazards

- RAW问题: Read After Write数据冲突

Recall: registers
are read /sourced
In the "decode" stage

add 1 2 3
nor 3 4 5

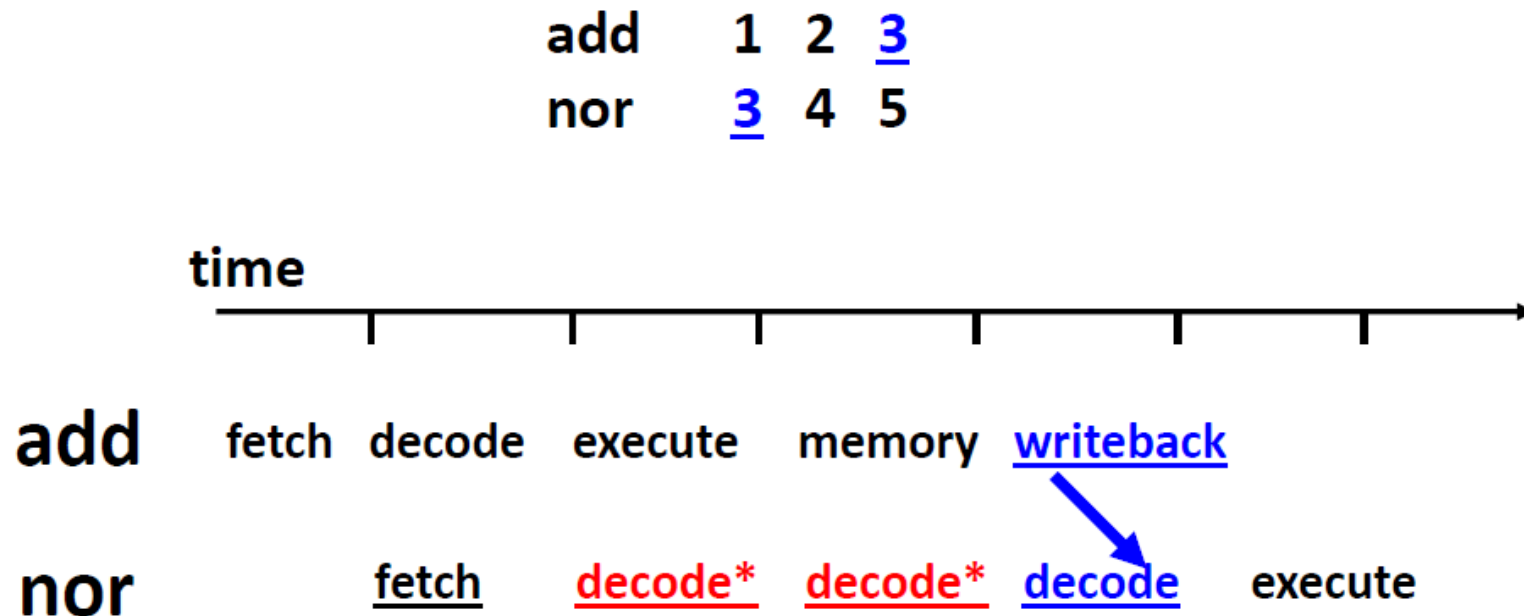
RAW Dependency



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

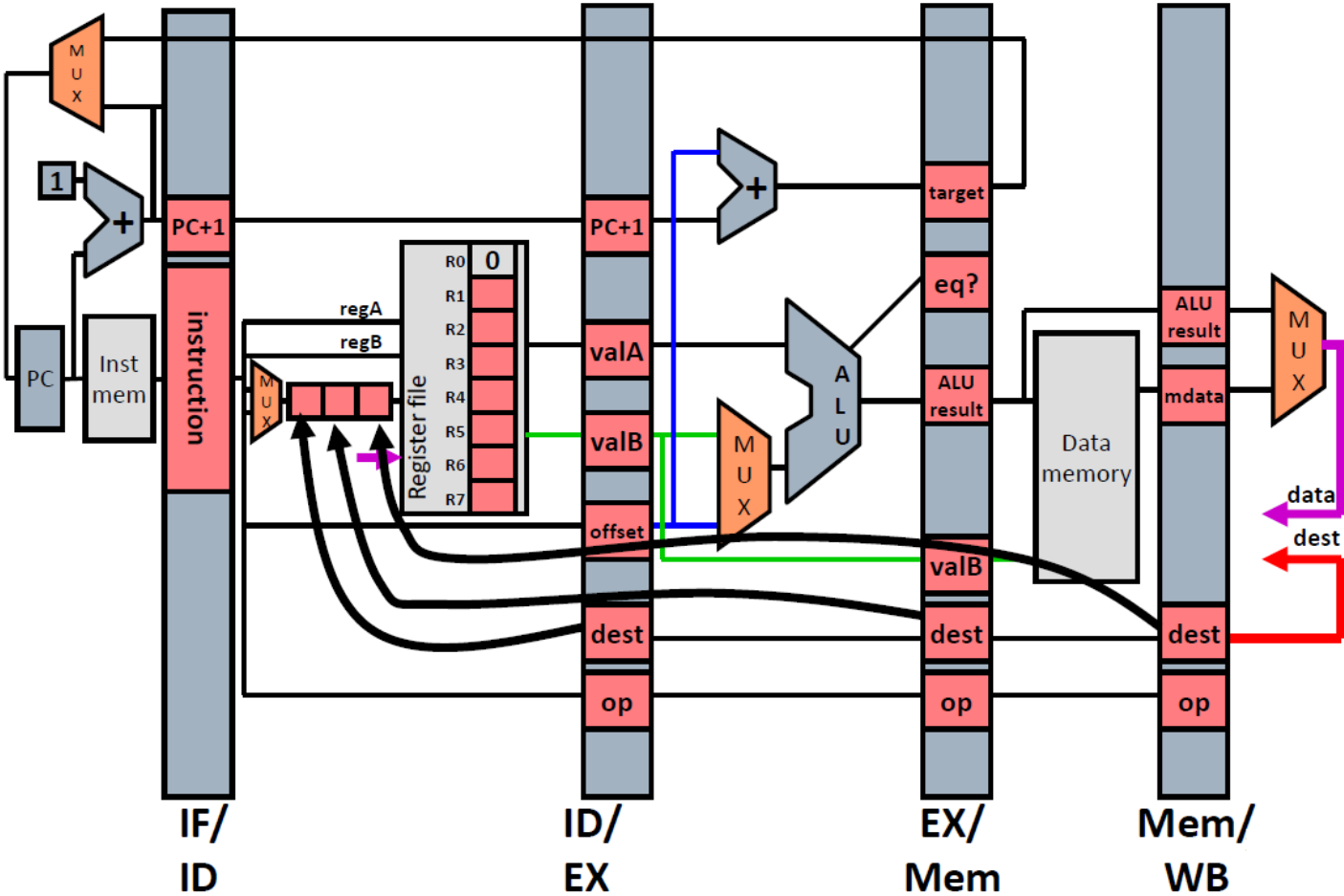
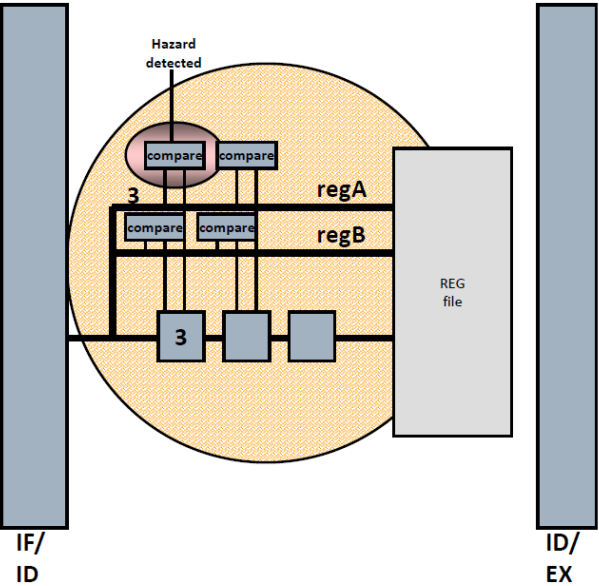


简单解决办法: 流水线停顿 (Pipeline Stall)

问题1: Data Hazards

- RAW问题: Read After Write数据冲突

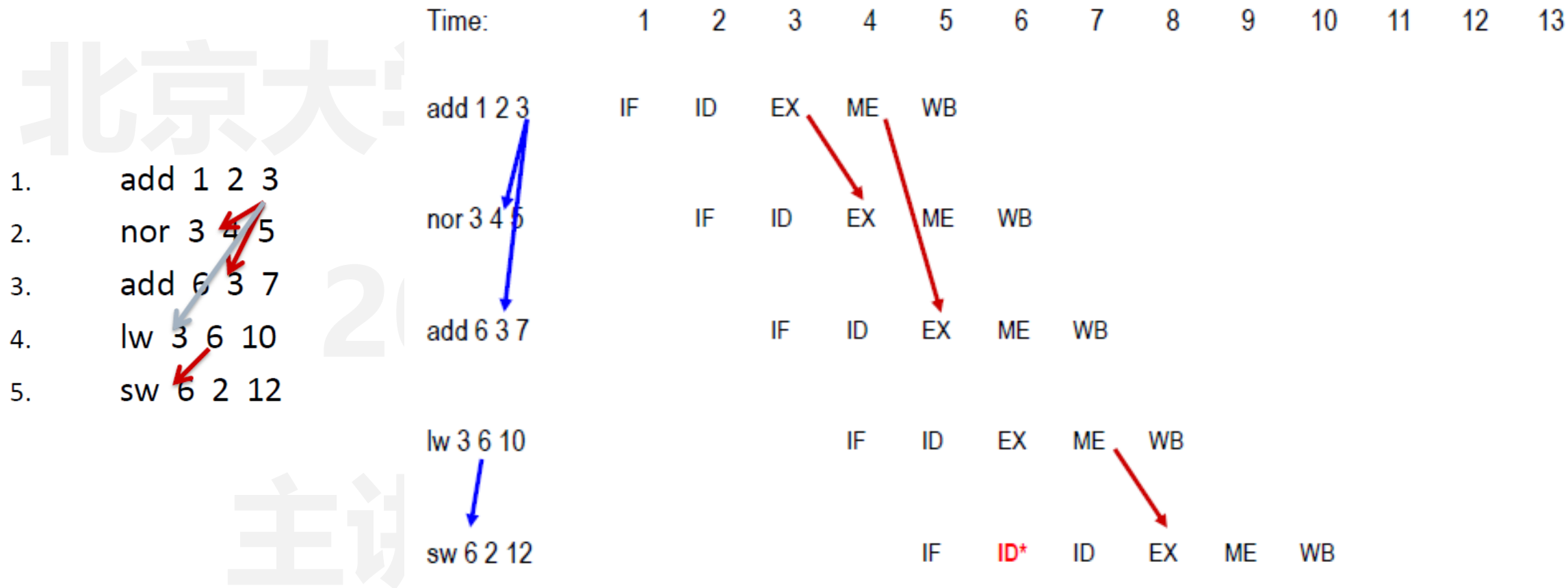
```
1. add 1 2 3
2. nor 3 4 5
3. add 6 3 7
4. lw 3 6 10
5. sw 6 2 12
```



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 // $r3 = r1 + r2$

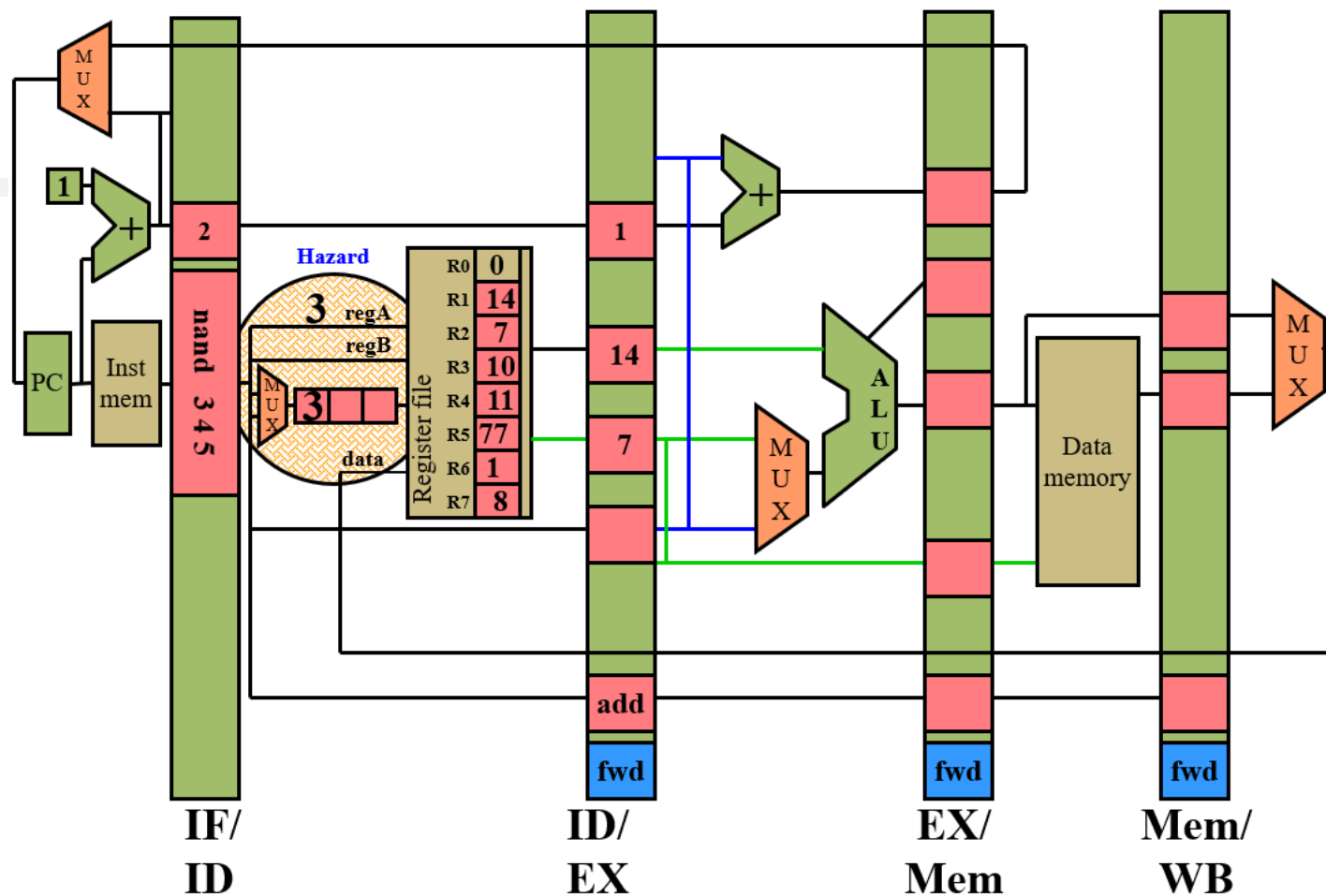
nand 3 4 5 // $r5 = r3 \text{ NAND } r4$

add 6 3 7 // $r7 = r3 + r6$

lw 3 6 10 // $r6 = \text{MEM}[r3+10]$

sw 6 2 12 // $\text{MEM}[r6+12]=r2$

Cycle 3前半段



问题1: Data Hazards

- **Detect and Forward**案例

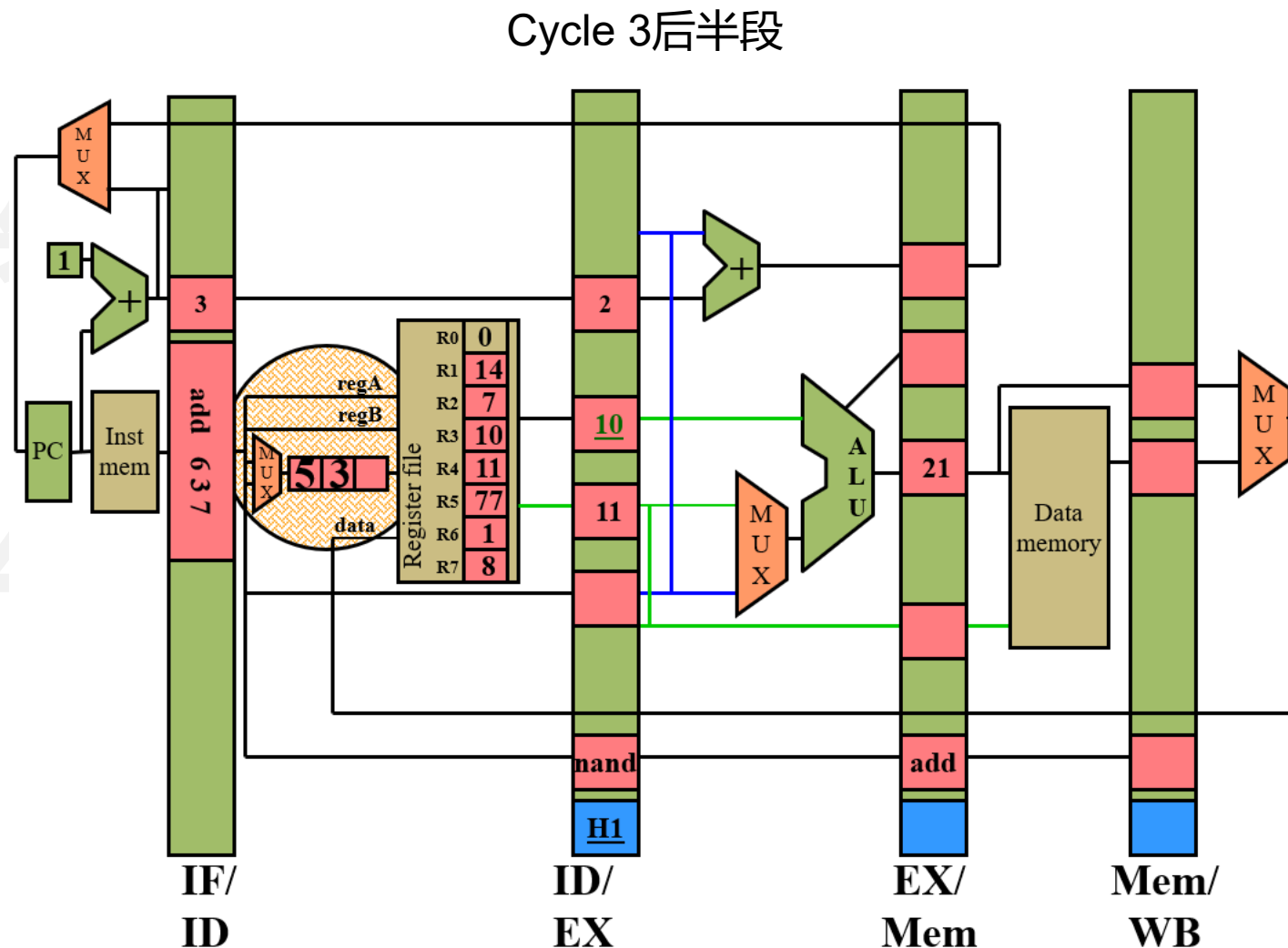
```
add 1 2 3 // r3 = r1 + r2
```

```
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
```



问题1: Data Hazards

• Detect and Forward案例

Cycle 4前半段 (forwarding)

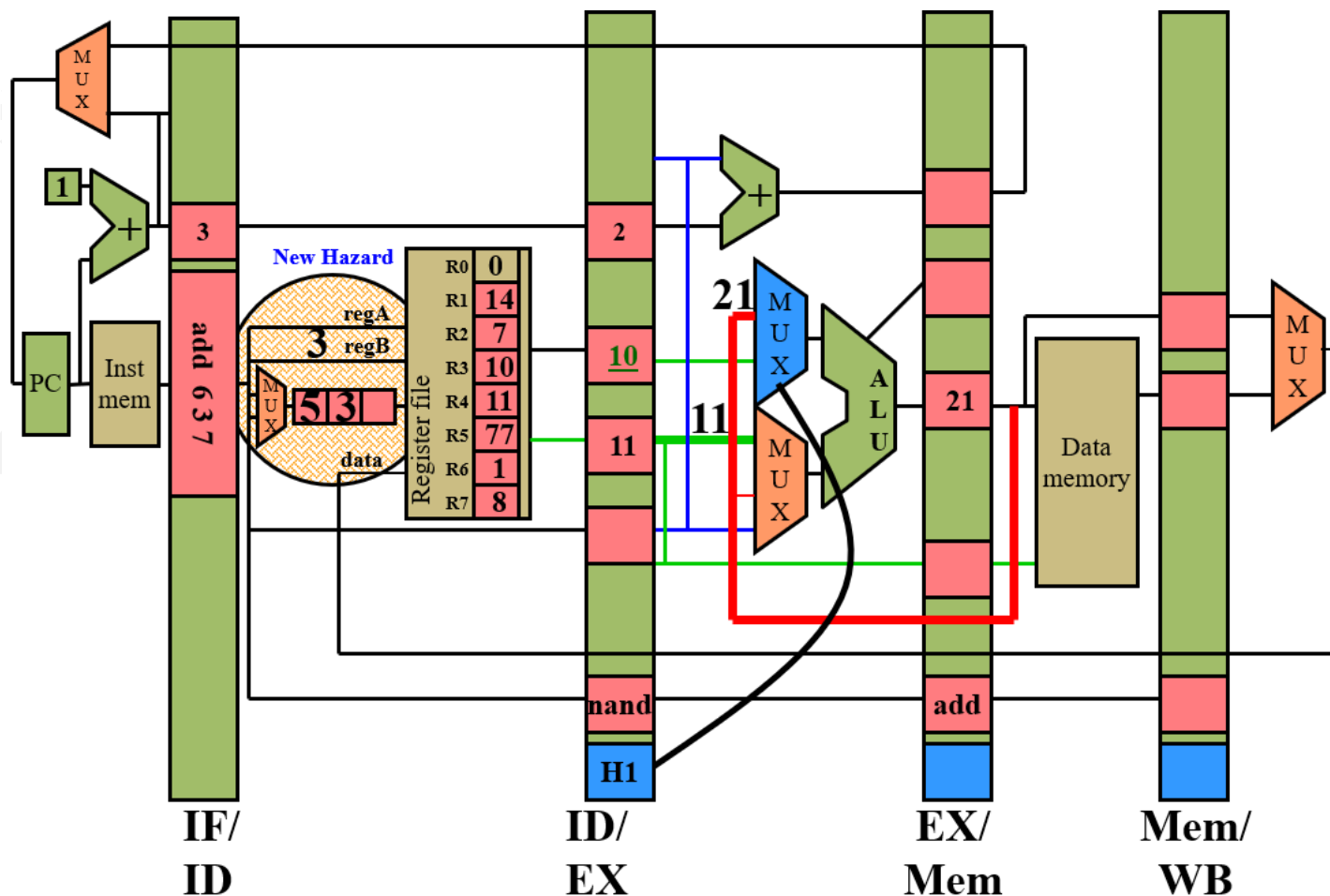
add 1 2 3 // r3 = r1 + r2

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

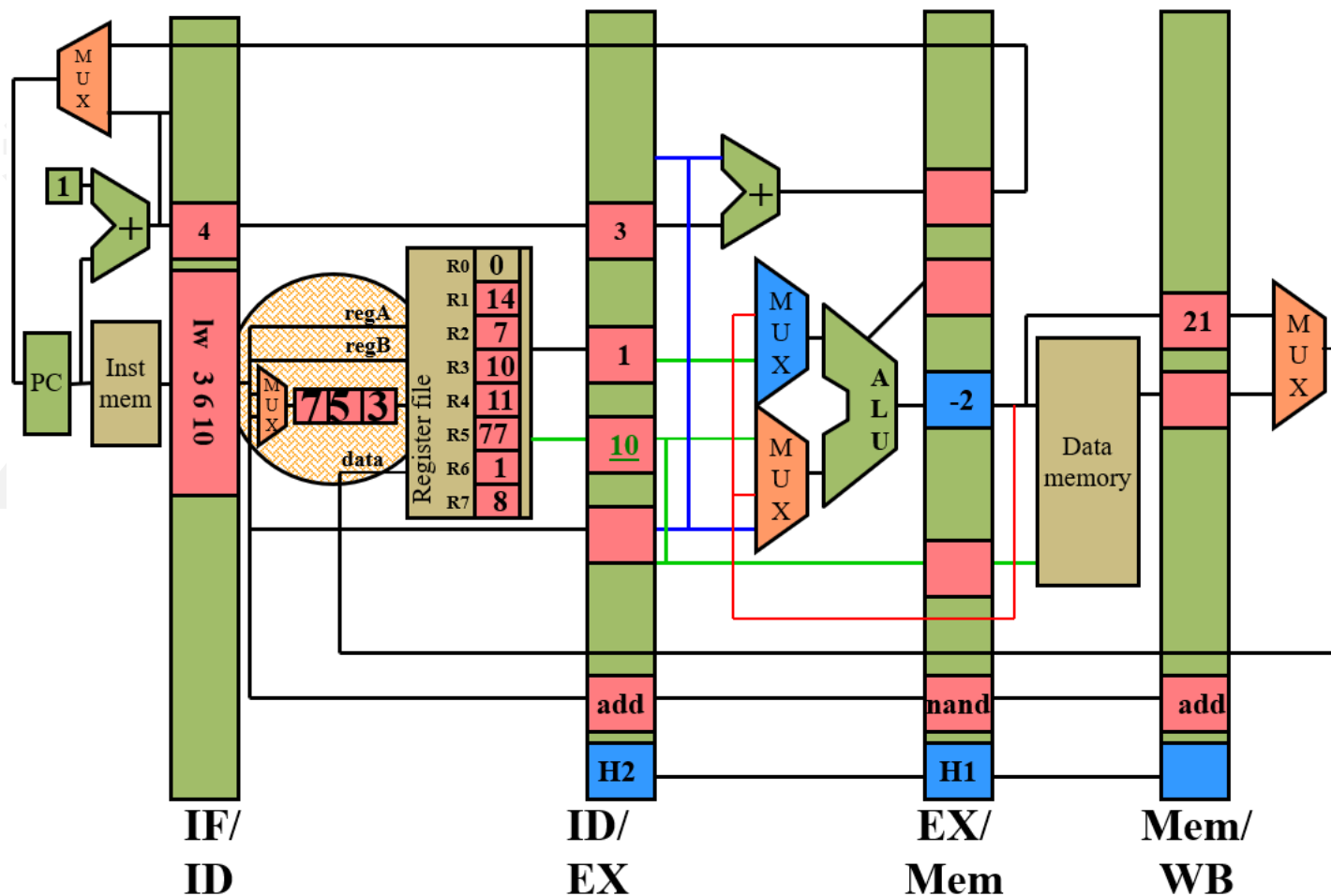


问题1: Data Hazards

• Detect and Forward案例

Cycle 4后半段

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



问题1: Data Hazards

• Detect and Forward案例

add 1 2 3 // r3 = r1 + r2

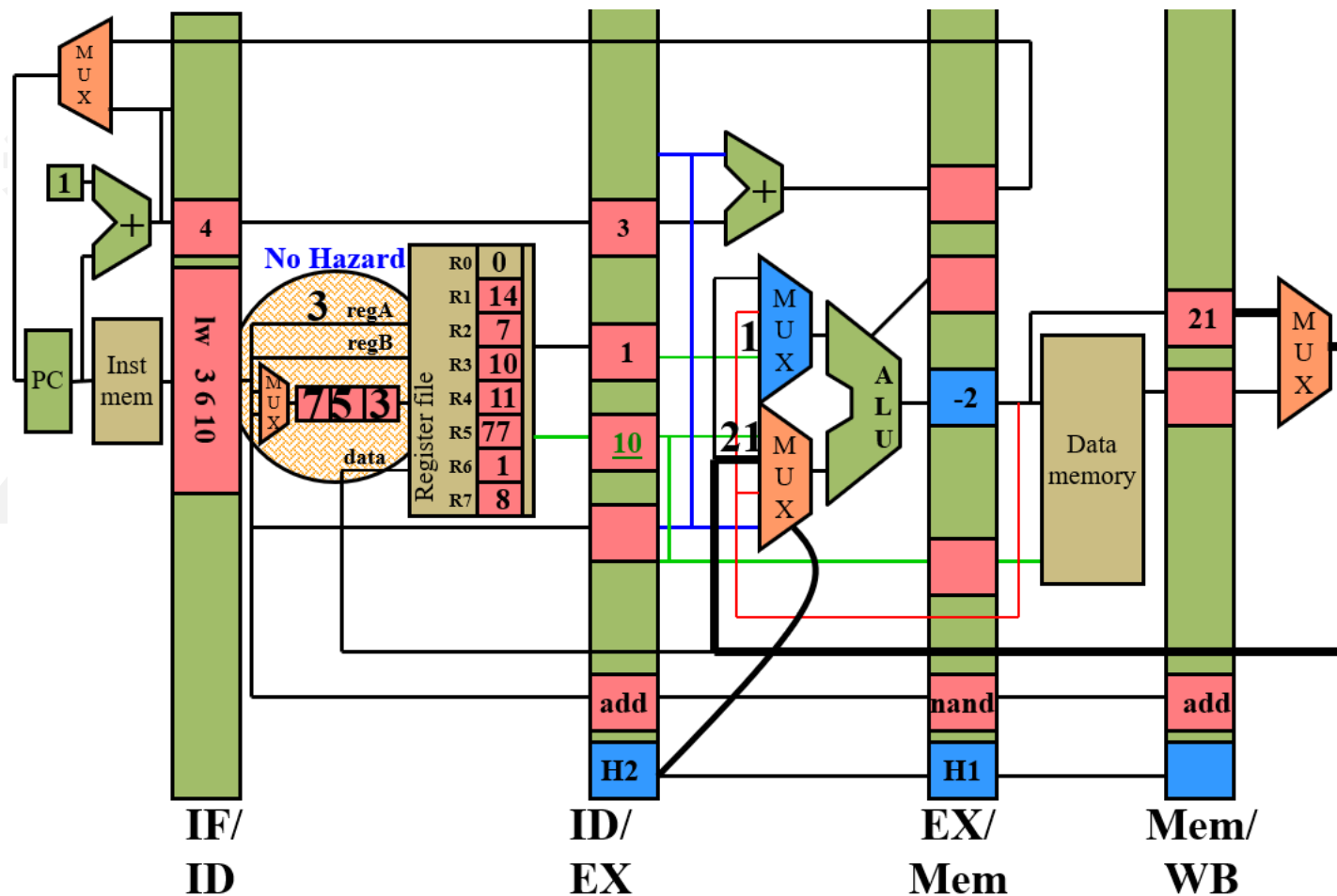
nand 3 4 5 // r5 = r3 NAND r4

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

Cycle 5后半段

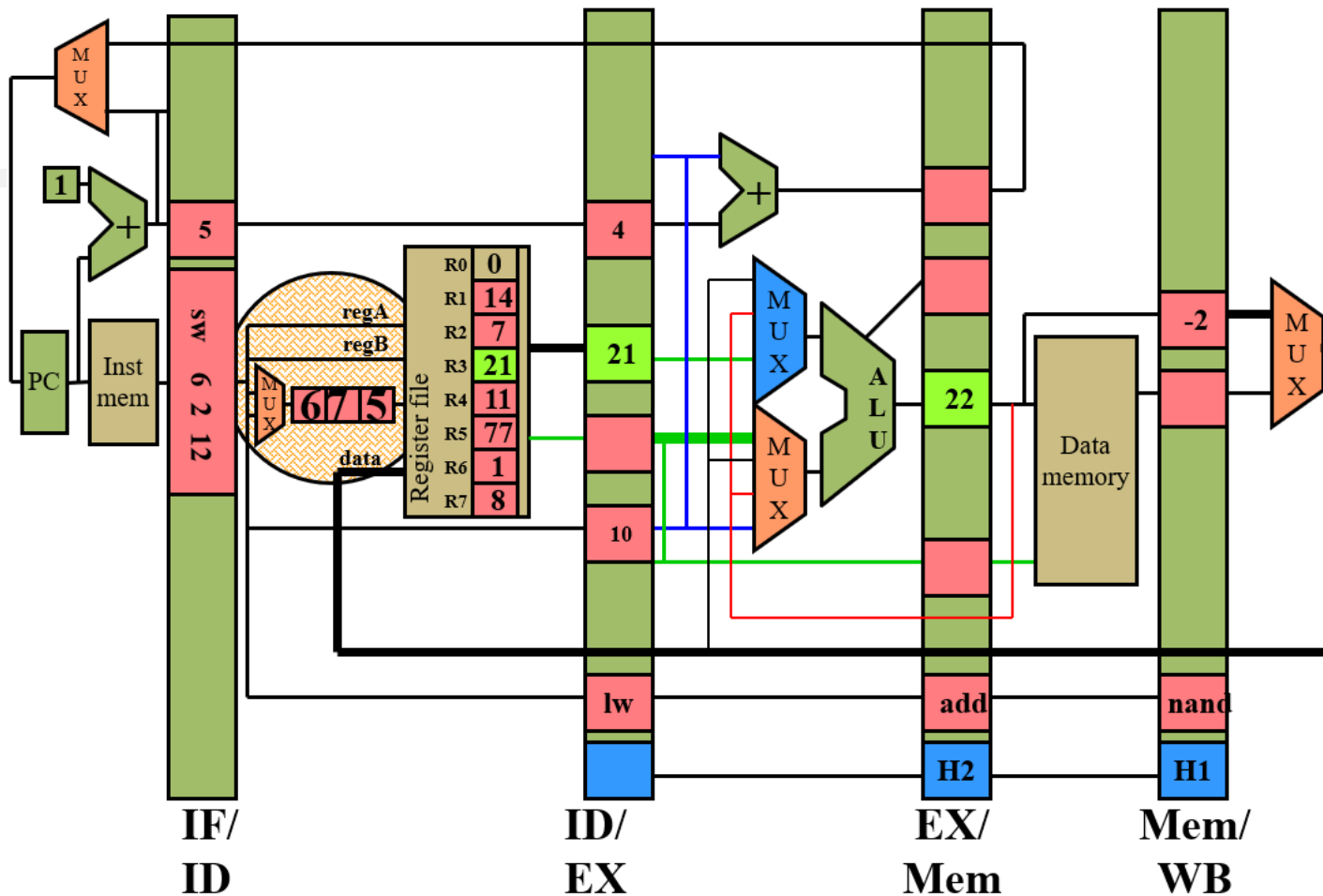
add 1 2 3 // r3 = r1 + r2

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



问题1: Data Hazards

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

- False or Name dependencies

- WAW – Write after Write

$$R1=R2+R3$$

$$R1=R4+R5$$

- WAR – Write after Read

$$R2=R1+R3$$

$$R1=R4+R5$$

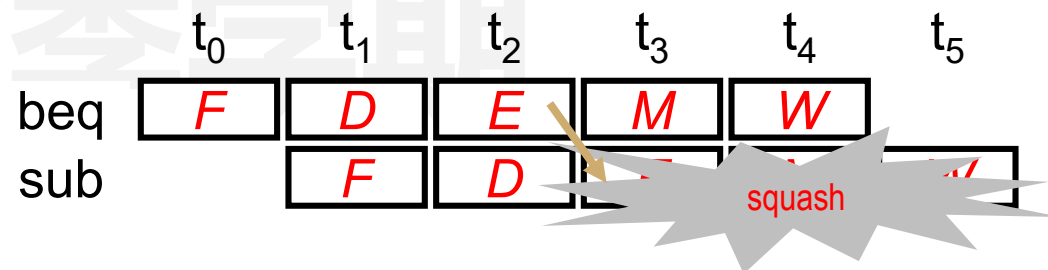
- 在顺序的单条5级流水线上不会出现问题
- 指令乱序执行则会出现问题, 可利用Register重命名解决 (后续乱序执行深入讲解)

问题2: Control Hazards

- Branch类指令

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

beq 1 1 10
sub 3 4 5



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

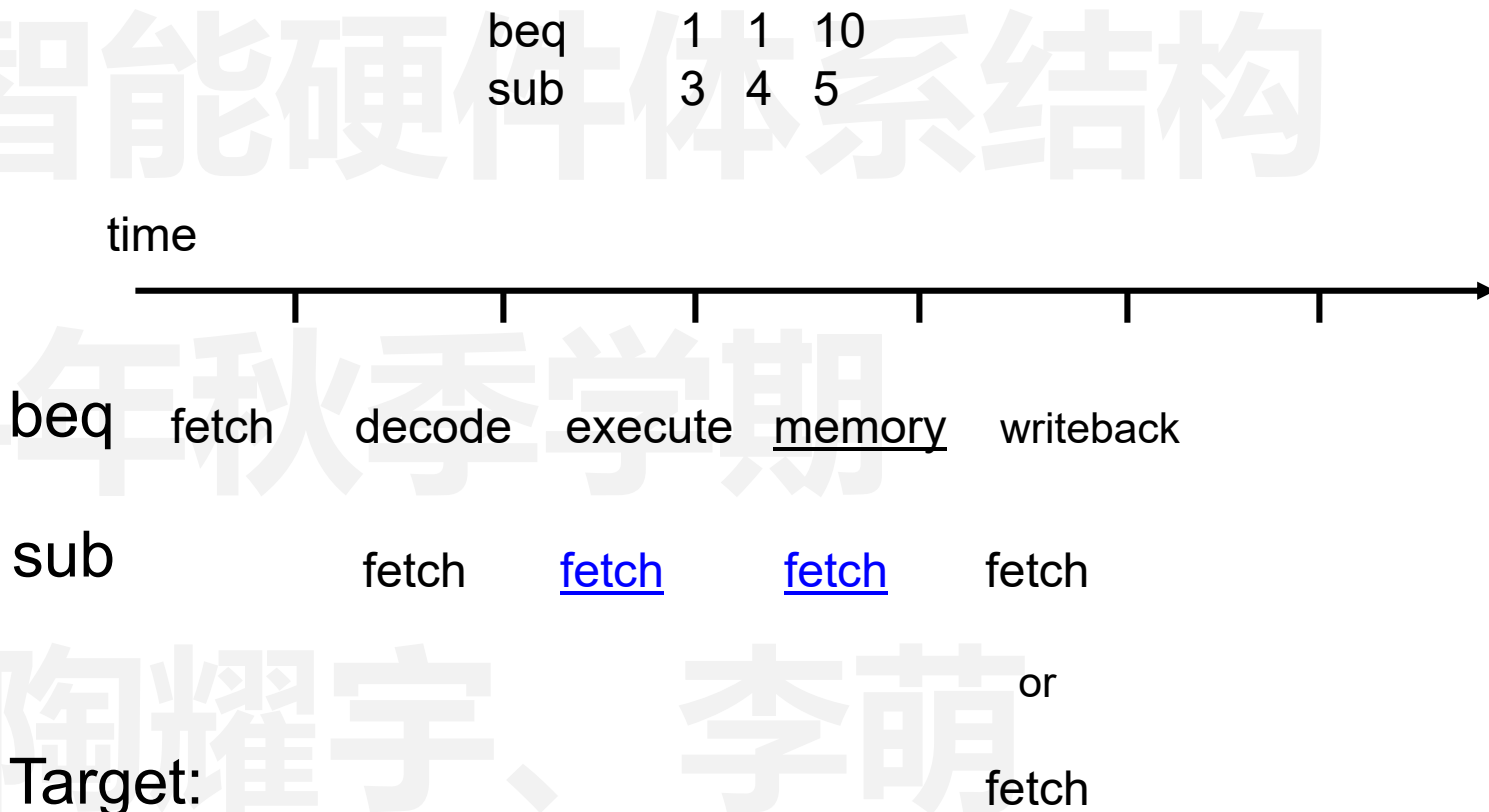
Detection

- In decode, check if opcode is branch or jump

Stall

- Hold next instruction in Fetch
- Pass noop to Decode

- 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

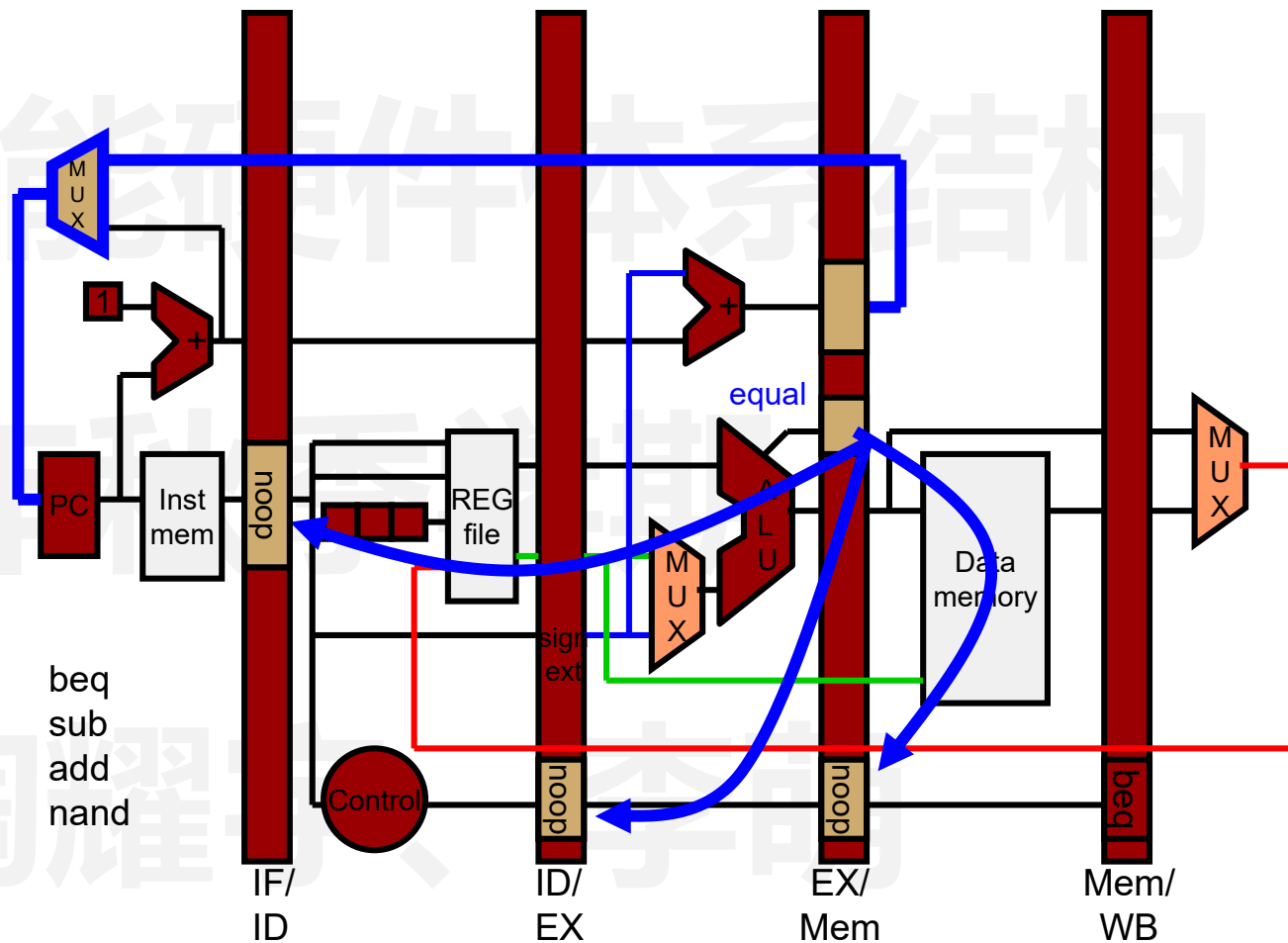
• Speculate and Squash: 投机-制止机制

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

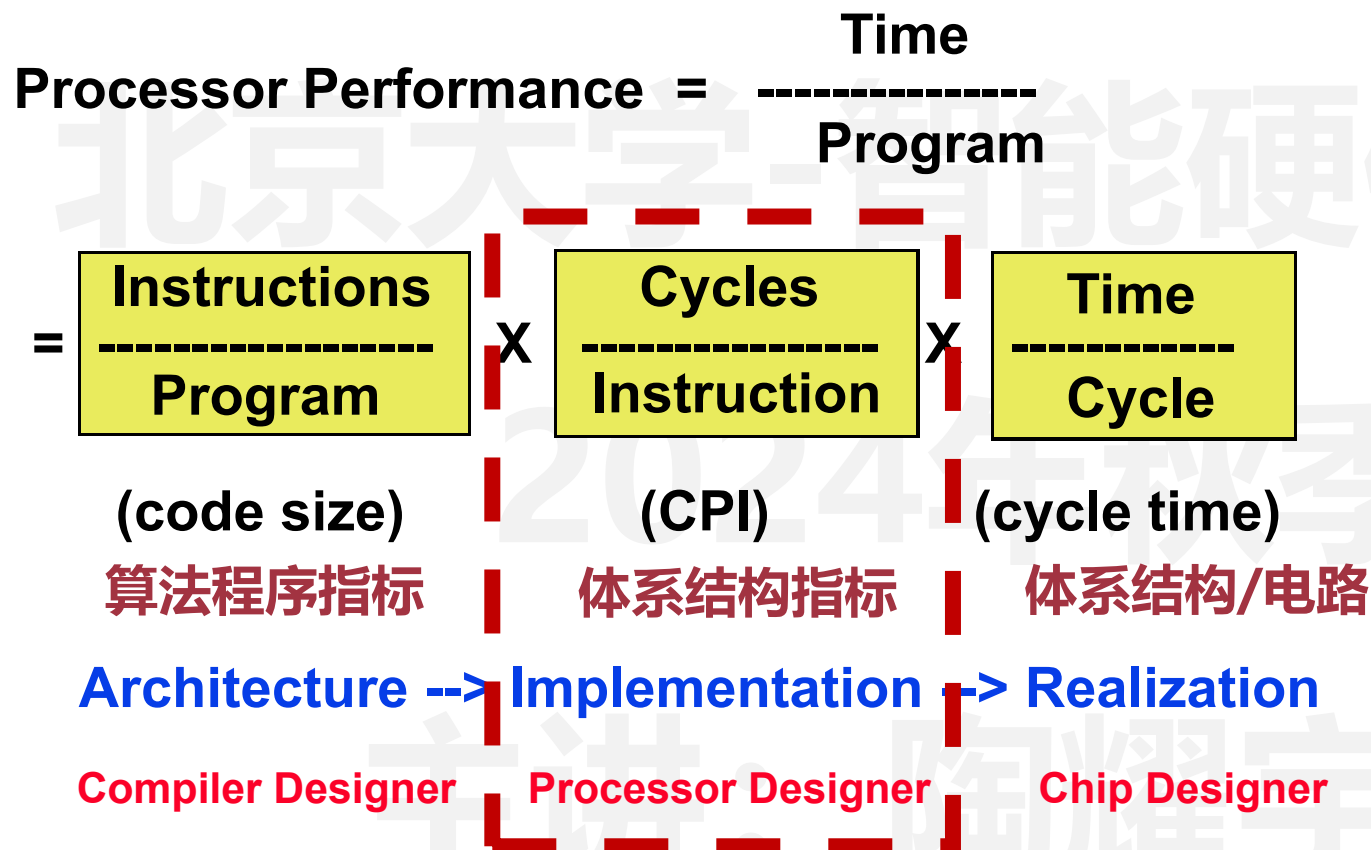
Can we do better? Yes.

- **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

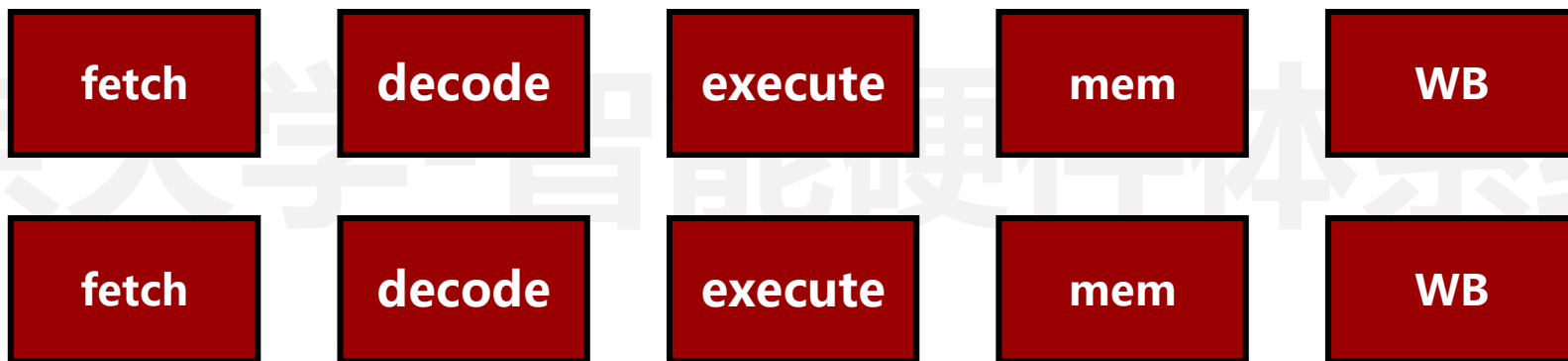


两大限制：

- 1、Upper Bound on Scalar Pipeline Throughput
- 2、Performance Lost Due to Rigid In-order Pipeline

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

Peak IPC

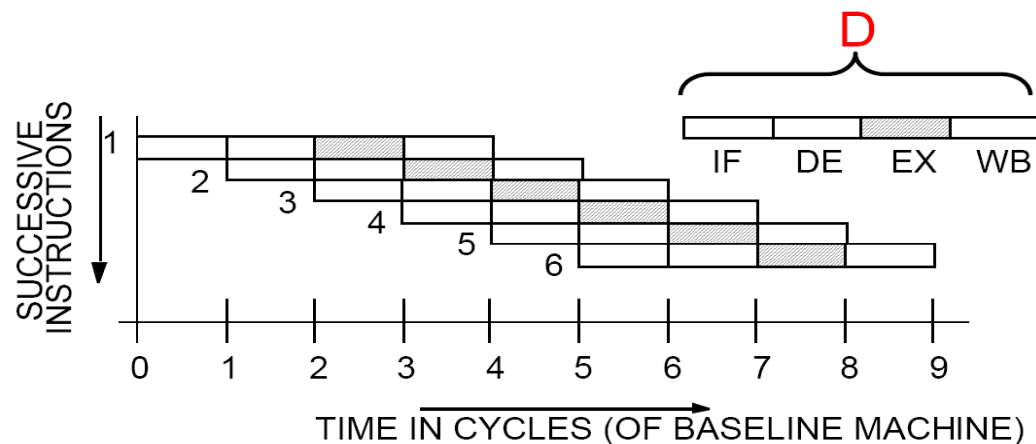
The maximum sustainable number of instructions that can be executed per clock.

Scalar Pipeline (baseline)

Instruction Parallelism = D

Operation Latency = 1

Peak IPC = 1

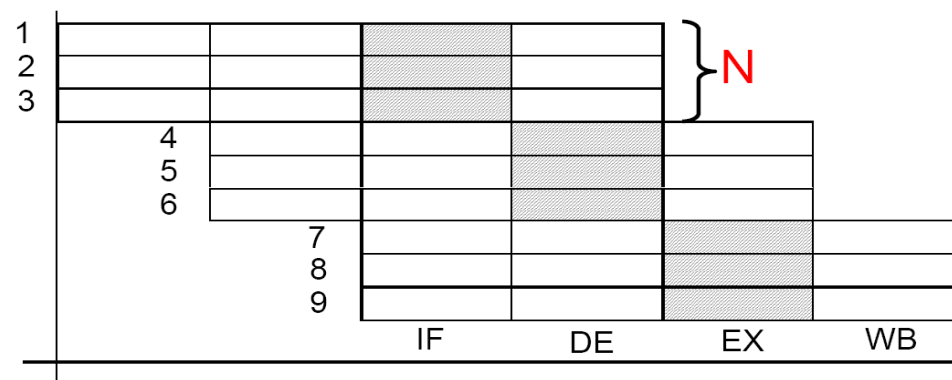


Superscalar (Pipelined) Execution

IP = $D \times N$

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
<code>addf f0, f1, f2</code>	F	D	E+	E+	E+	W										
<code>mulf f2, f3, f2</code>		F	D	d*	d*	E*	E*	E*	E*	E*	W					
<code>subf f0, f1, f4</code>			F	p*	p*	D	E+	E+	E+	W						

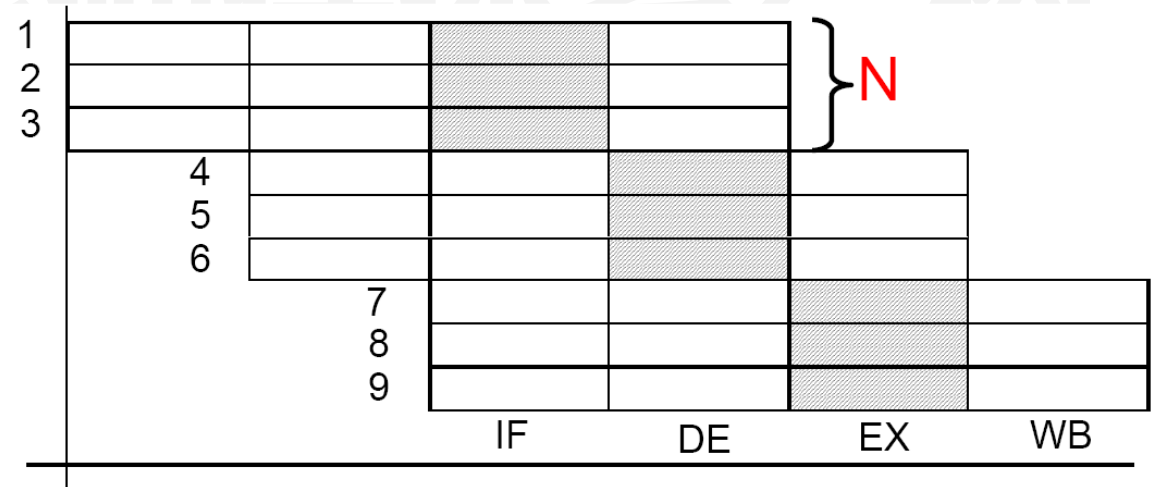
What' s happening in cycle 4?

- `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?

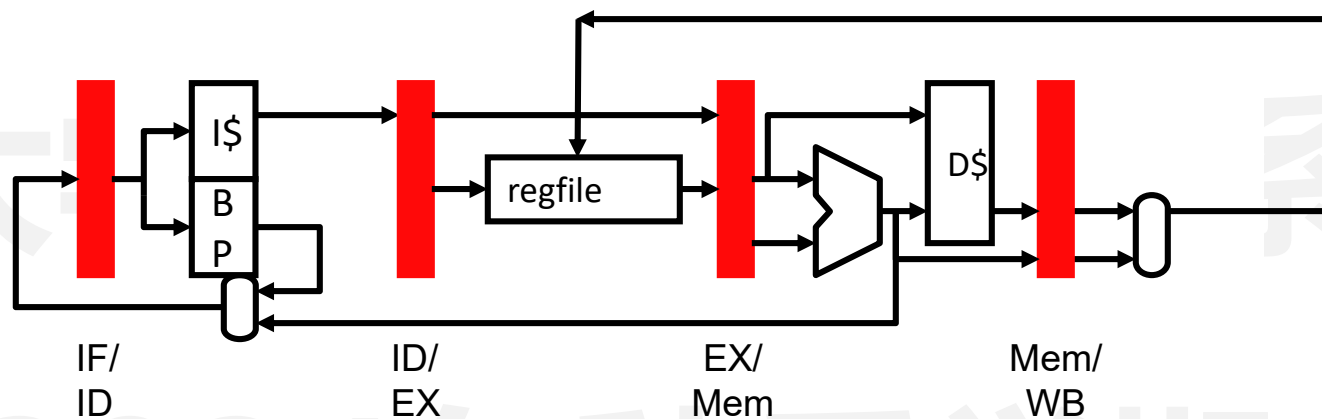
Out-Of-Order: 乱序执行的概念

- Instruction-level parallelism
- CPI of in-order pipelines degrades sharply if the machine parallelism is increased beyond a certain point.
 - *when $N \times M$ 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

• 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

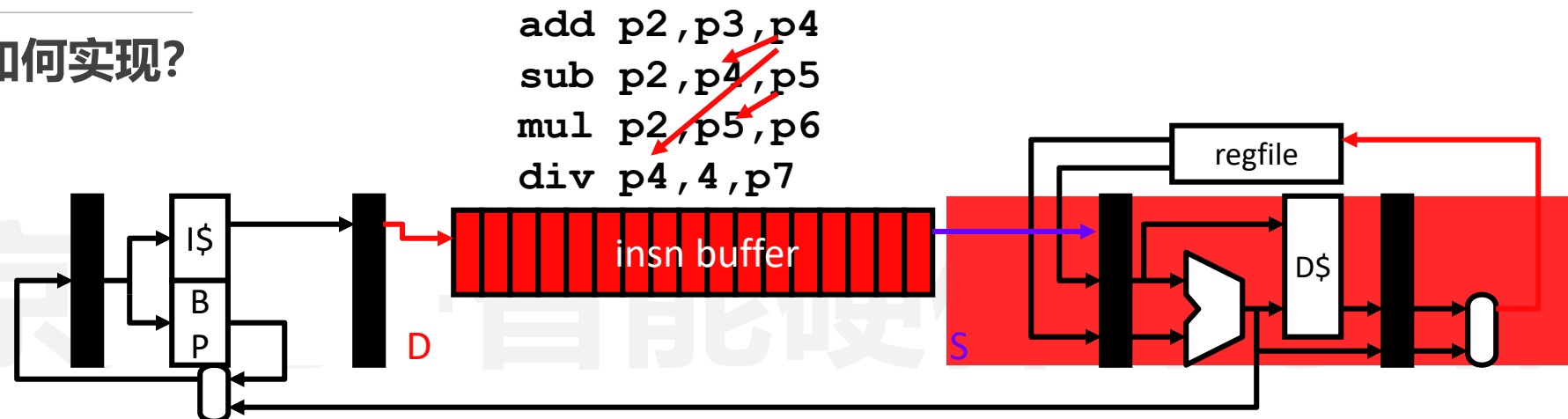
Important

–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
t	Yes	Yes				
	Yes	Yes	Yes			
	Yes	Yes	Yes	Yes		Yes
	Yes	Yes	Yes	Yes	Yes	Yes

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.

- 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!)

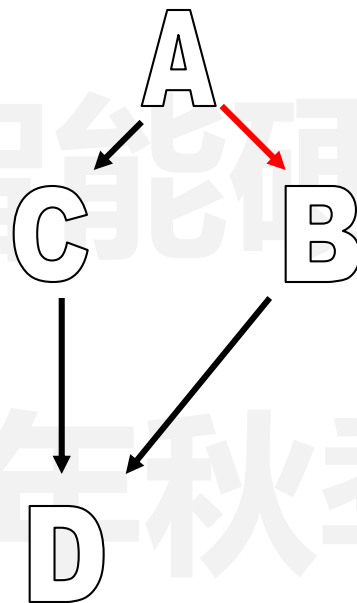
- True/False Data Dependencies

$R1 = \text{MEM}[R2 + 0]$ // A

$R2 = R2 + 4$ // B

$R3 = R1 + R4$ // C

$\text{MEM}[R2 + 0] = R3$ // D



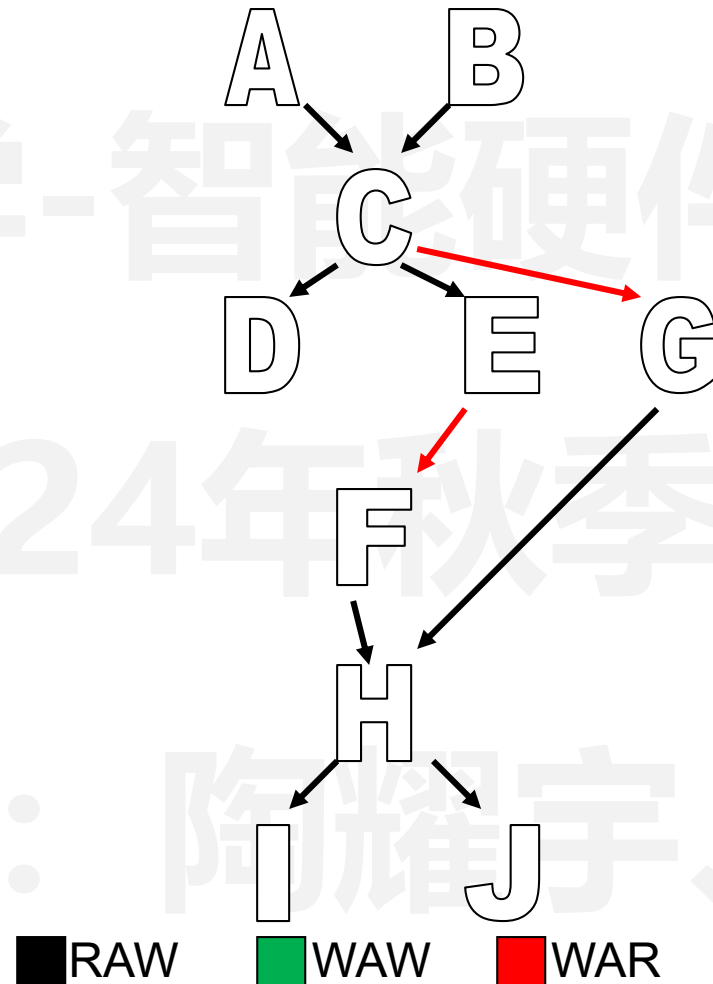
■ RAW

■ WAW

■ WAR

• True/False Data Dependencies

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



- Well, logically there is no reason for F-J to be dependent on A-E. So.....

- ABFG
- CH
- DEIJ
- Should be possible.
- But that would cause either C or H to have the wrong reg inputs
- How do we fix this?
 - Remember, the dependency is really on the *name* of the register
 - **So... change the register names!**

- 触发器Register重命名概念

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

R1 =

.... = R1

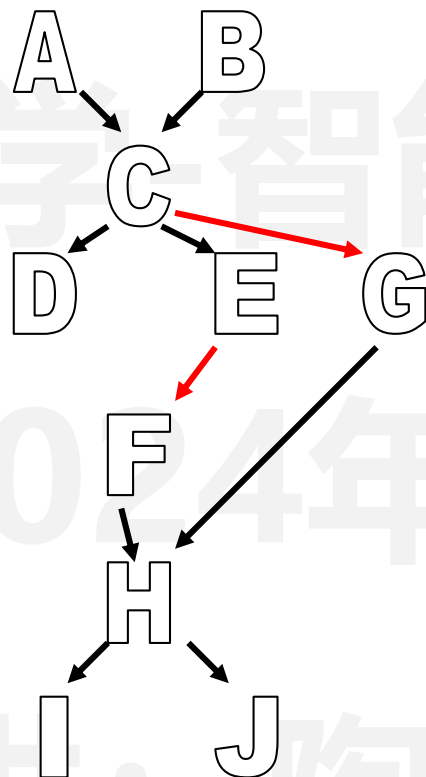
.... = ... R1

R1 =

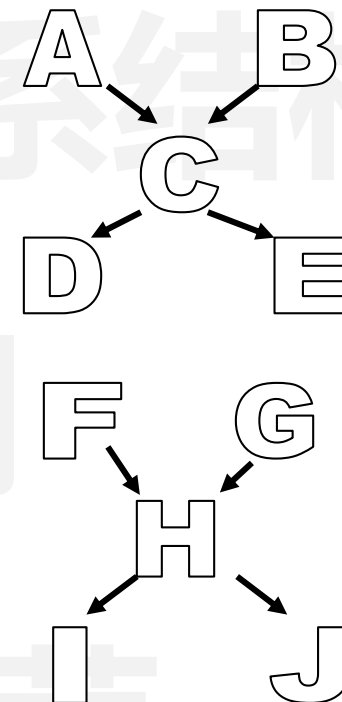
The value in R1 is “alive” from when the value is written until the last read of that value.

• 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Register重命名机制

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

Architecture register
虚拟的架构触发器

Physical register
实际的电路触发器

MapTable

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

FreeList

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

Orig. insns

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

Renamed insns

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