INTRODUCTION TO RISC-V

RISC-V入门教程

ISA | 汇编指令 | 系统编程 | 组成原理 | 嵌入式应用

流水线处理器

主讲 邢建国

中国开放指令生态（RISC-V）联盟 | 浙江中心
China RICS-V Alliance | Zhejiang Center

浙江图灵算力研究院
ZHEJIANG TURING INSTITUTE

RISC-V®

流水线处理器

- 基本概念
- 流水线处理器
- 数据相关
- 控制相关
- 提高流水线性能

流水线基本概念

- CPU性能公式
  - CPU时间 = 指令数（IC） × 每指令平均时钟周期数（CPI） × 时钟周期（T）

•指令数（IC）

•程序执行所需的总指令条数，受算法、编译器优化及指令集（ISA）影响。

•CPI（Cycles Per Instruction）

•每条指令平均消耗的时钟周期，由CPU微架构决定（如流水线深度、缓存命中率）。

•时钟周期（T）

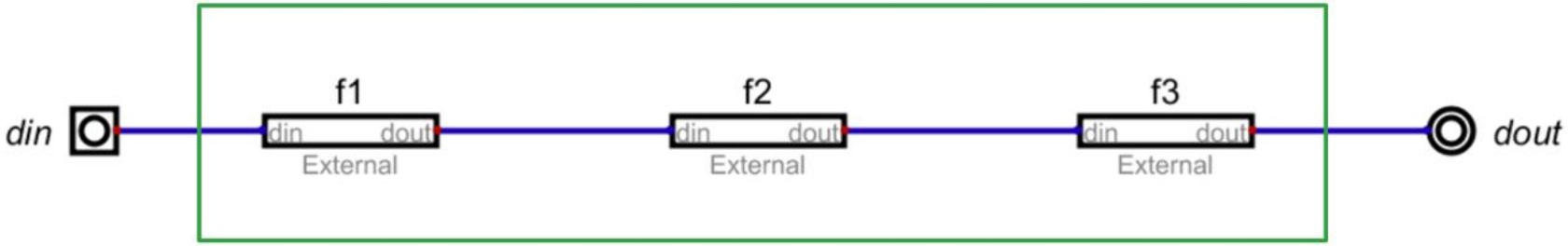
•单个时钟周期的持续时间（单位：秒），其倒数为时钟频率（F，单位：Hz）。

流水线基本概念

- CPU性能公式
  - CPU时间 = 指令数（IC） × 每指令平均时钟周期数（CPI） × 时钟周期（T）
- 提升处理器运行速度
  - 减少指令数
    - 复杂指令、算法优化
  - 提高主频
    - 减小时钟周期
  - 减少每条指令所需时钟周期数
    - 利用时间并行性，提高吞吐量（流水线）

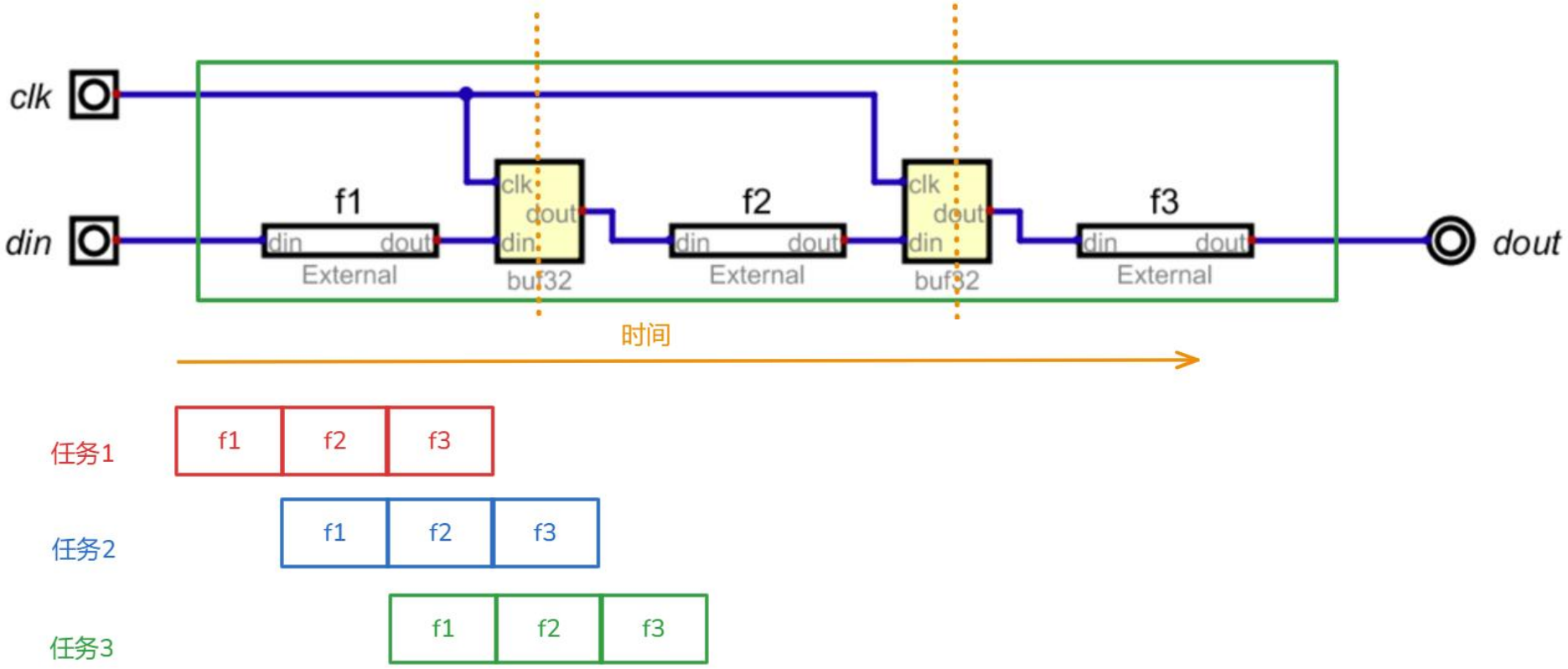
流水线基本概念

- 例子
  - 一个计算任务包括三个阶段：f1、f2、f3
  - 假定每一阶段所需均为T，则完成一次计算需要3T
  - 计算某一段时，其他段都是空闲的



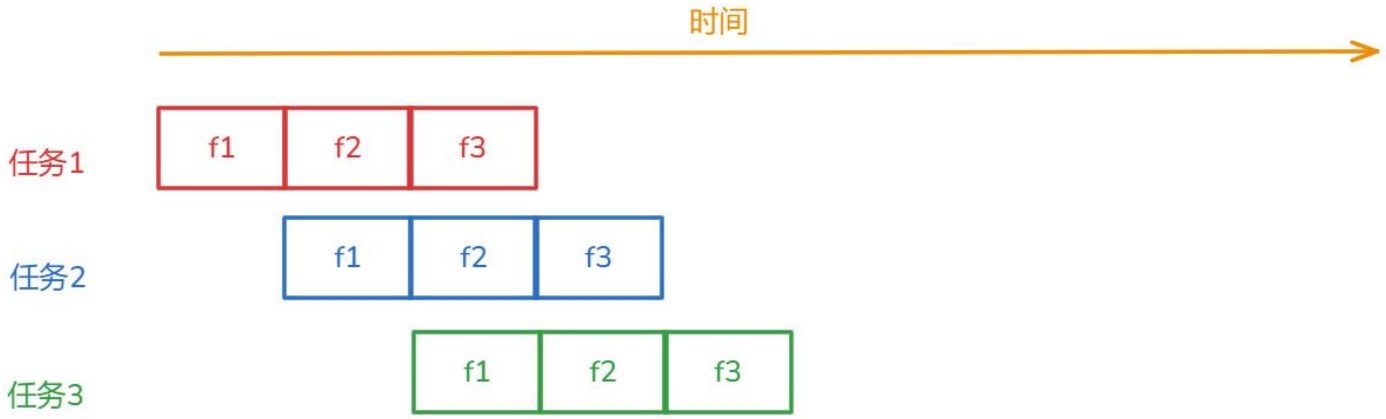
流水线基本概念

- 例子
  - 一个计算任务包括三个阶段：f1、f2、f3
  - 在各段之间增加寄存器
  - 三个计算任务并行，每一任务处于f1、f2、f3的某一段



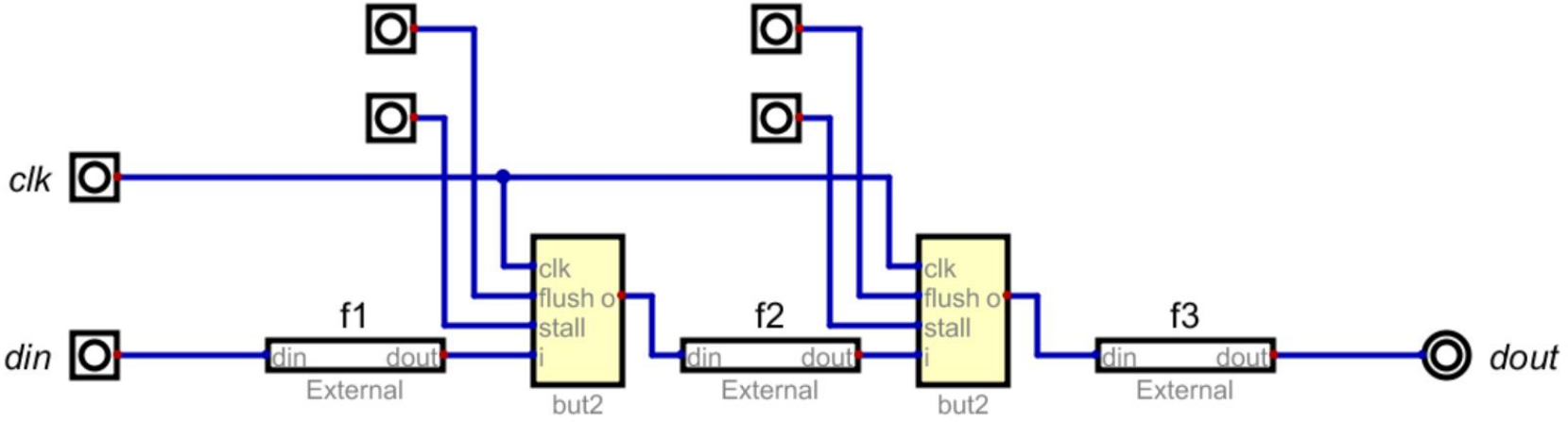
流水线基本概念

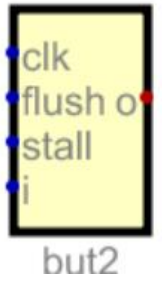

时间
任务1
f1
f2
f3
任务2
f1
f2
f3
任务3
f1
f2
f3


时间
任务1
f1
f2
f3
任务2
f1
f2
f3
任务3
f1
f2
f3

流水线的暂停(stall)和刷新(flush)

- 有的时候需要暂停流水线的某一段，或者作废流水线的某一段的数据



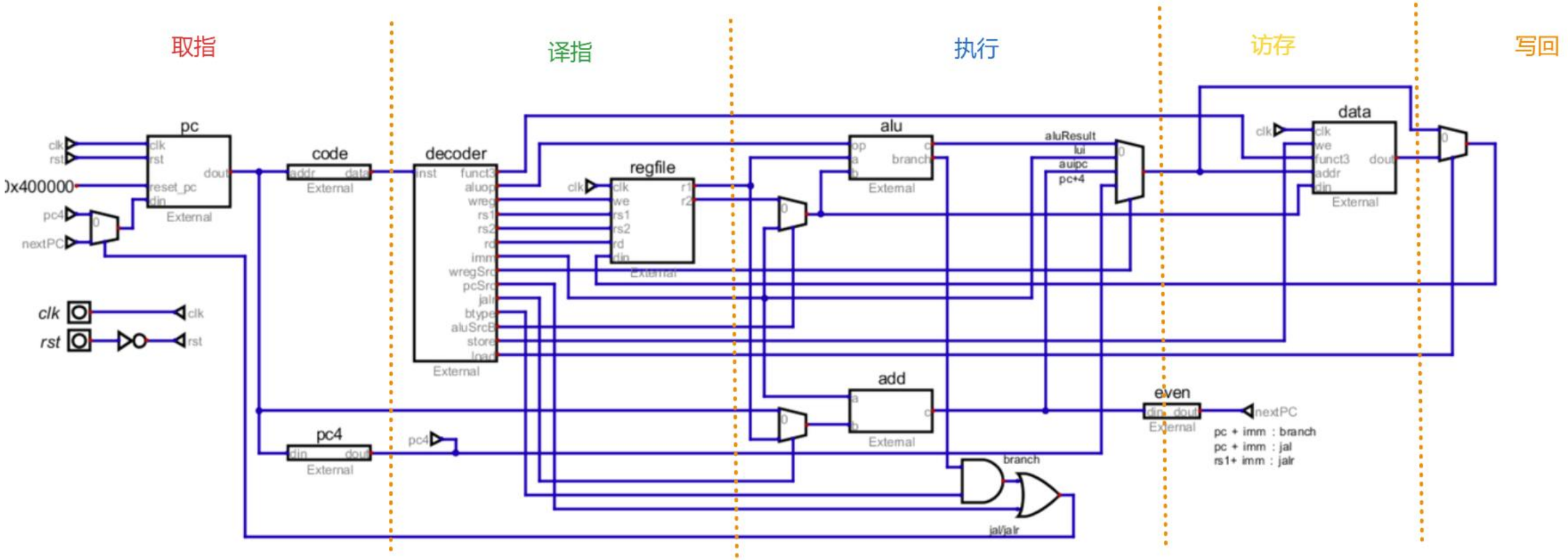



```
module buf32(
    input clk,
    input flush,
    input stall,
    input [31:0] din,
    output reg [31:0] dout
);
    always @(posedge clk)
    begin
        if(flush)
            dout = 0;
        else if(~stall)
            dout = din;
    end
endmodule
```

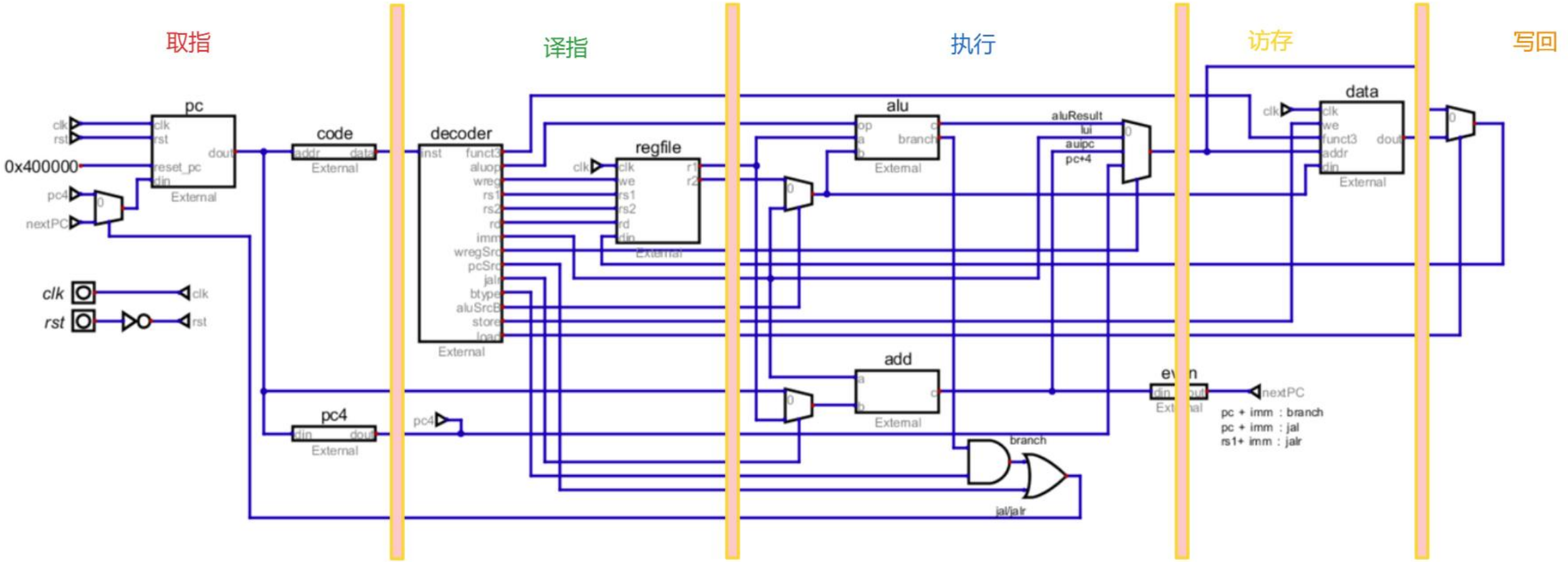
流水线处理器

- 将单周期处理器分为 5 个阶段：
  - 取指（fetch）
  - 译码（decode）
  - 执行（execute）
  - 访存（memory）
  - 写回（write back）
- 在阶段之间添加寄存器

流水线处理器

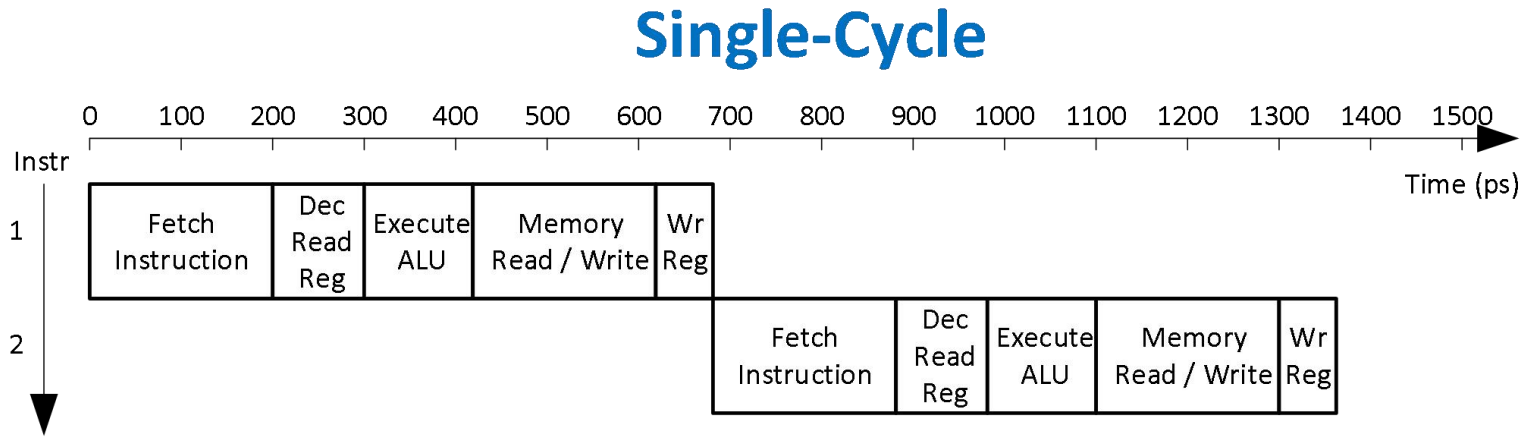

取指
译指
执行
访存
写回
pc
clk
rst
0x400000
pc4
nextPC
reset_pc
din
dout
External
code
addr
data
External
decoder
inst
funct3
aluop
wreg
rs1
rs2
rd
imm
wregSrc
pcSrc
jal
btype
aluSrcB
store
load
External
regfile
clk
we
rs1
rs2
rd
din
r1
r2
External
alu
op
a
b
c
branch
External
add
External
aluResult
lui
auipc
pc+4
data
clk
we
funct3
addr
din
dout
External
even
din
dout
External
nextPC
pc + imm : branch
pc + imm : jal
rs1+ imm : jalr
branch
jal/jalr
pc4
clk
rst

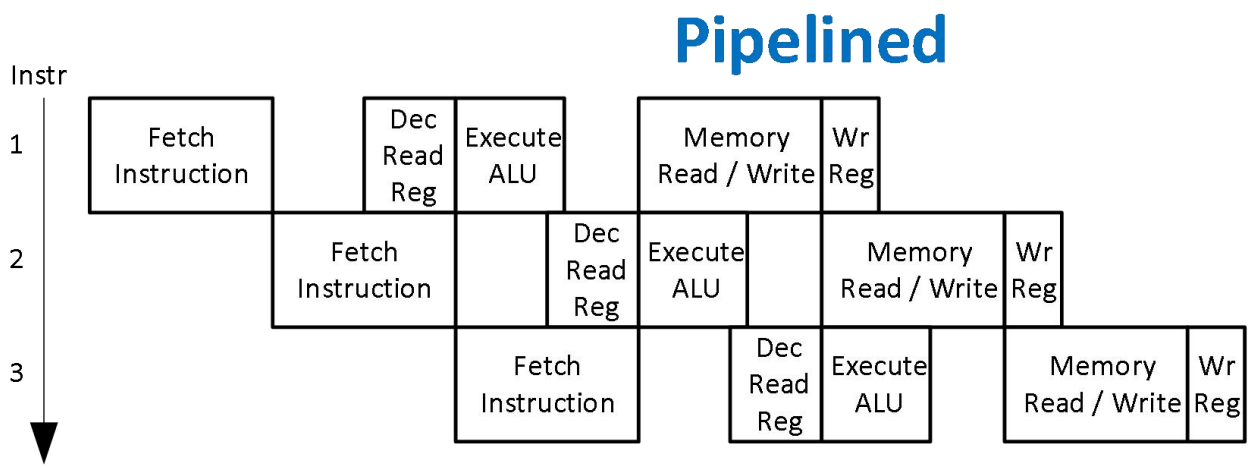
流水线处理器


取指
译指
执行
访存
写回
pc
clk
rst
0x400000
pc4
nextPC
reset_pc
dout
din
External
code
addr
data
External
decoder
inst
funct3
aluop
wreg
rs1
rs2
rd
imm
wregSrc
pcSrc
jal
btype
aluSrcB
store
load
External
regfile
clk
we
rs1
rs2
rd
din
r1
r2
External
alu
op
a
b
c
branch
External
add
External
aluResult
lui
auipc
pc+4
data
clk
we
funct3
addr
din
dout
External
branch
jal/jalr
nextPC
pc + imm : branch
pc + imm : jal
rs1+ imm : jalr

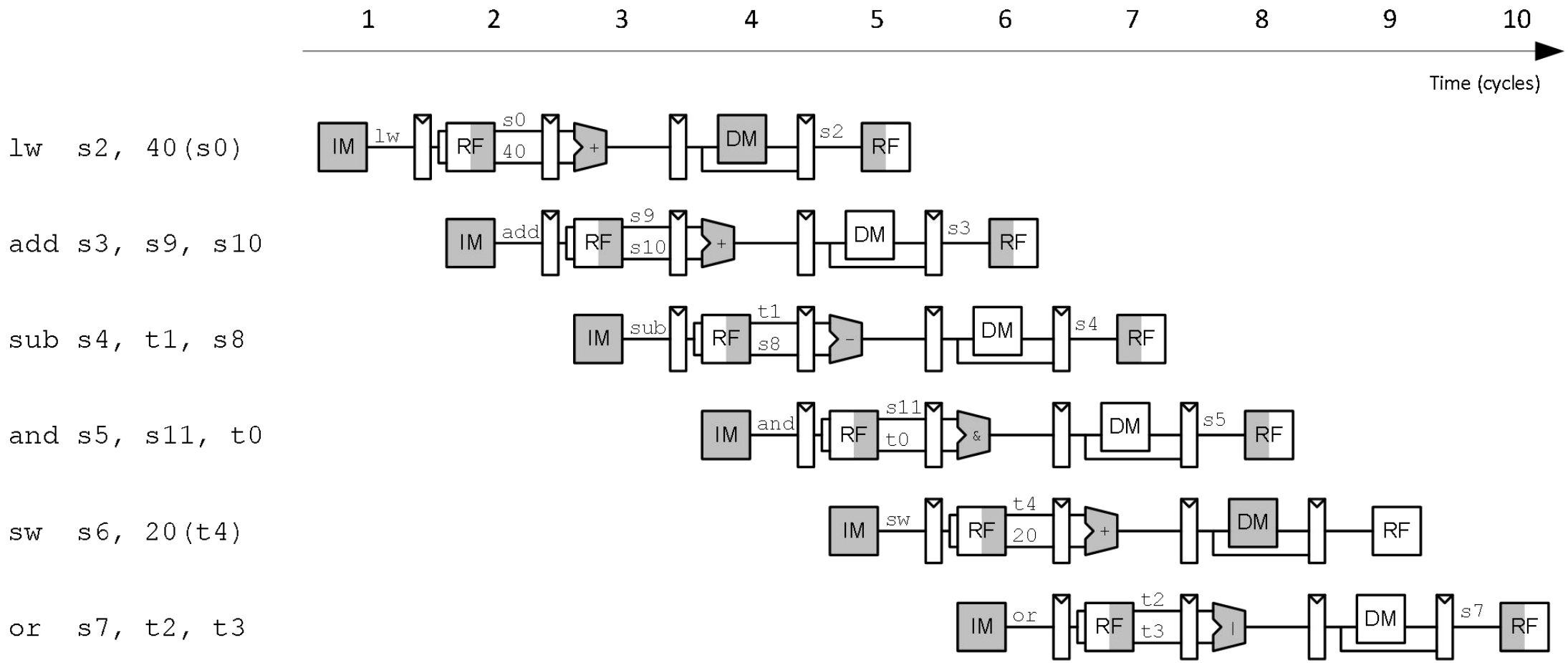
流水线处理器

- 将单周期处理器分为 5 个阶段：
  - 取指（fetch）
  - 译码（decode）
  - 执行（execute）
  - 访存（memory）
  - 写回（write back）
- 在阶段之间添加寄存器

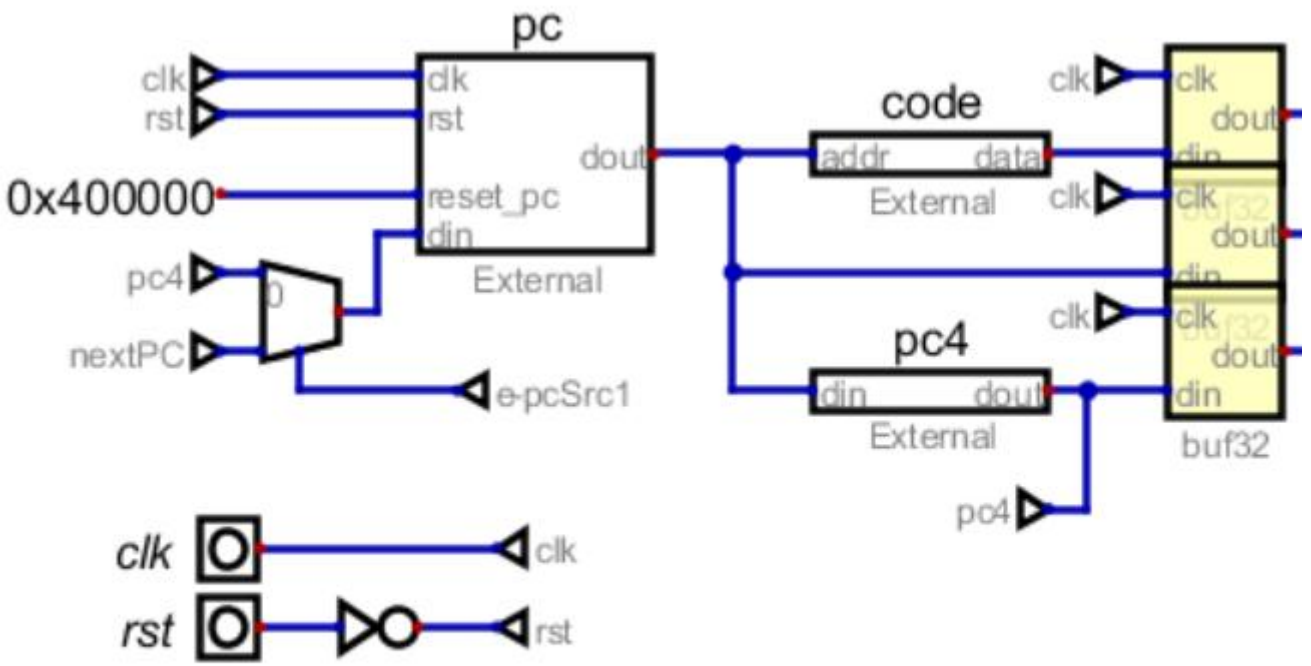




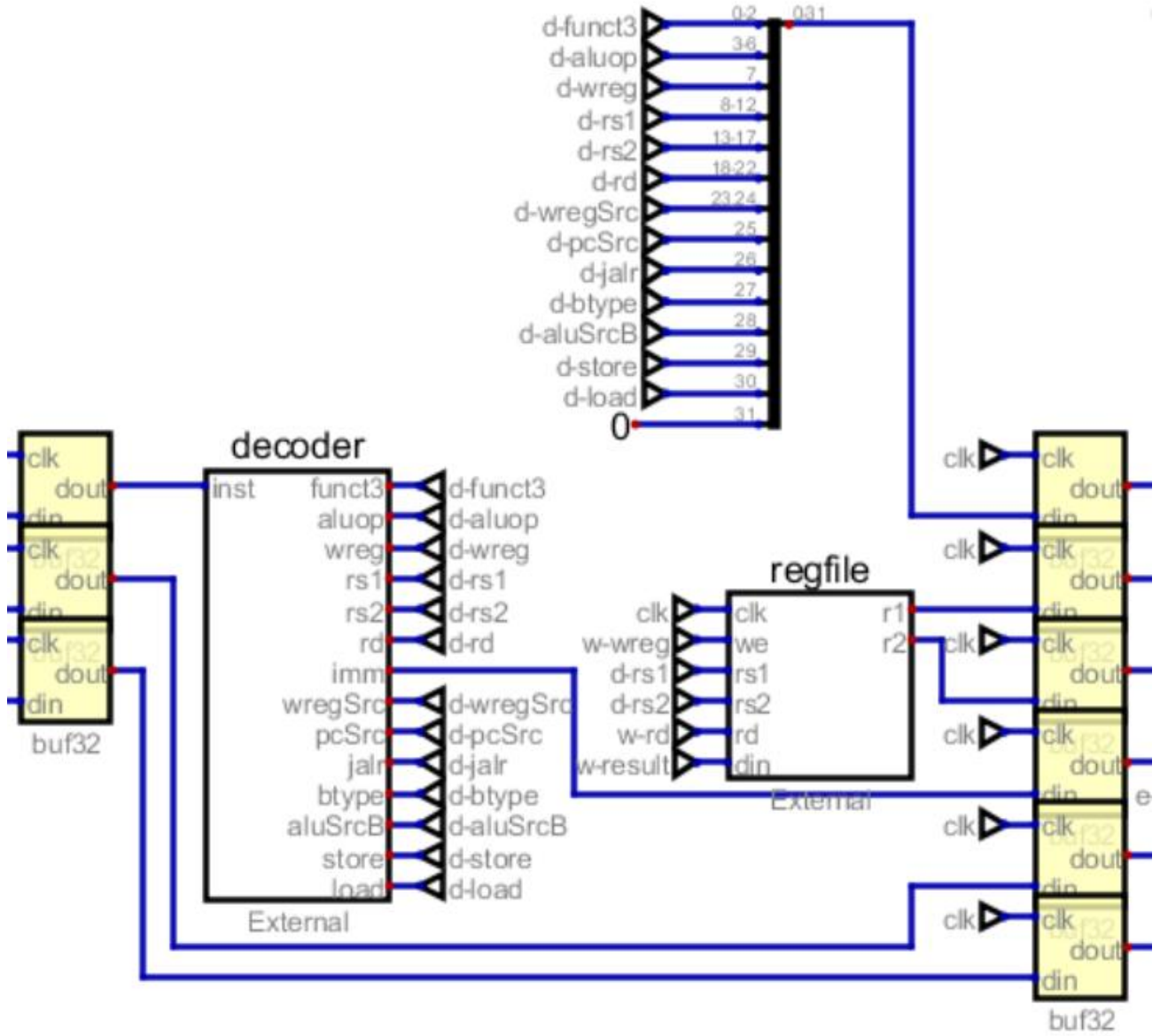
流水线处理器


1
2
3
4
5
6
7
8
9
10
Time (cycles)
lw s2, 40(s0)
add s3, s9, s10
sub s4, t1, s8
and s5, s11, t0
sw s6, 20(t4)
or s7, t2, t3
IM
RF
DM
lw
s0
40
s2
add
s9
s10
s3
sub
t1
s8
s4
and
s11
t0
s5
sw
t4
20
or
t2
t3
s7

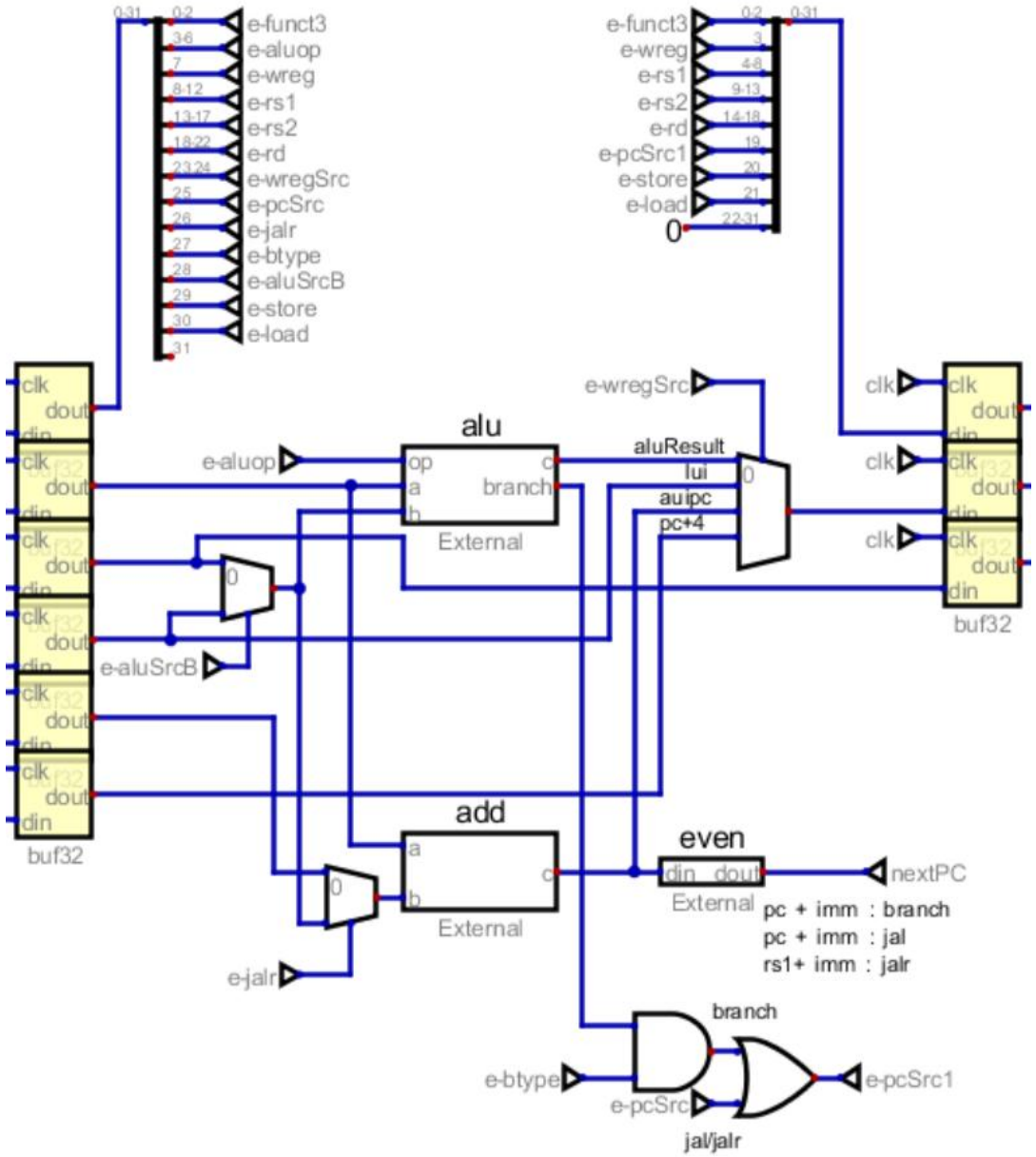
取指


pc
clk
rst
0x400000
pc4
nextPC
reset_pc
din
dout
External
e-pcSrc1
code
addr
data
External
pc4
din
dout
External
clk
rst
clk
rst
buf32
pc4

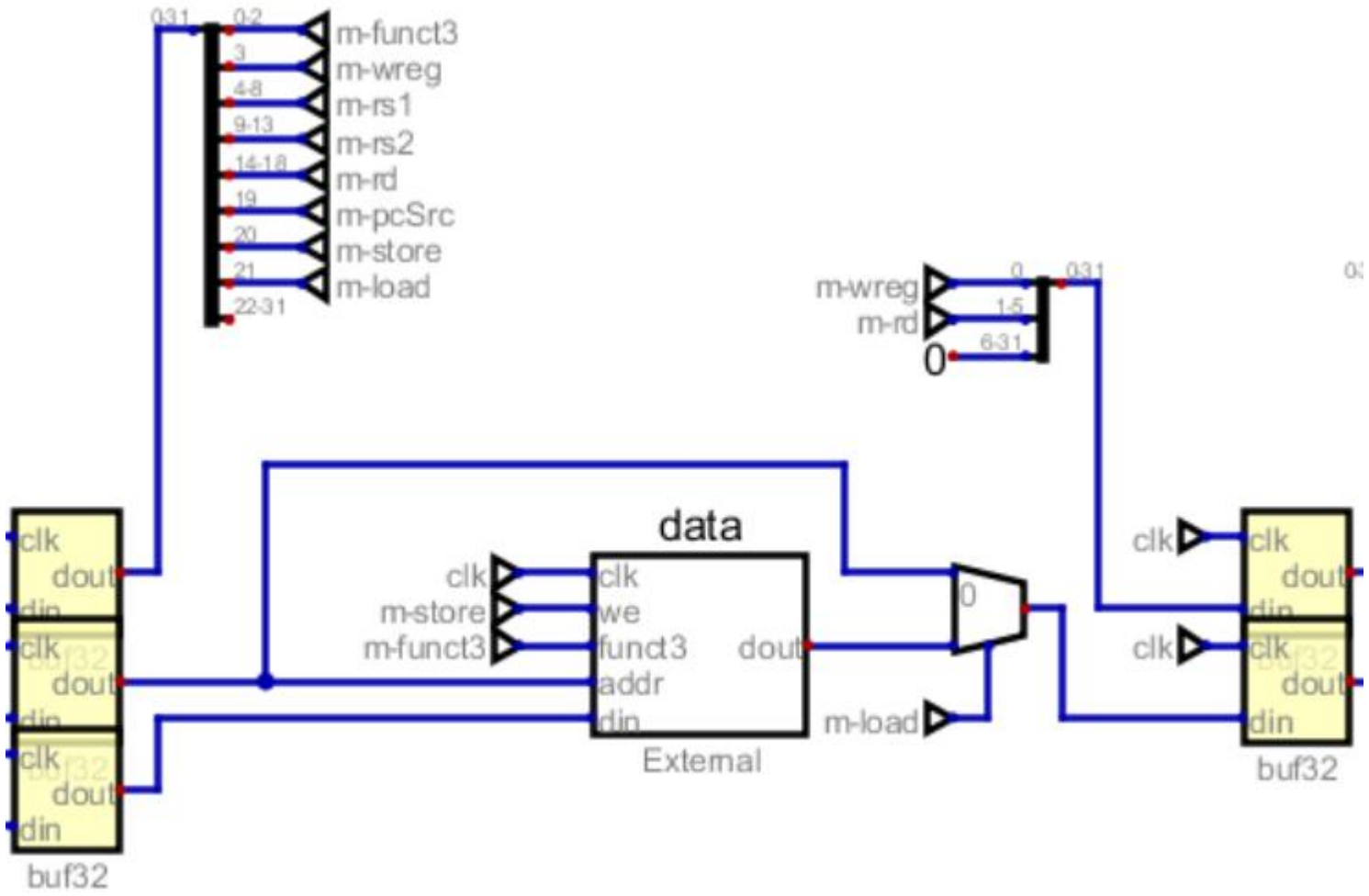
译指


d-funct3
d-aluop
d-wreg
d-rs1
d-rs2
d-rd
d-wregSrc
d-pcSrc
d-jalr
d-btype
d-aluSrcB
d-store
d-load
0
0-2
0-31
3-6
7
8-12
13-17
18-22
23-24
25
26
27
28
29
30
31
decoder
inst
funct3
aluop
wreg
rs1
rs2
rd
imm
wregSrc
pcSrc
jalr
btype
aluSrcB
store
load
External
regfile
clk
we
w-wreg
d-rs1
d-rs2
w-rd
w-result
rs1
rs2
rd
din
r1
r2
External
buf32
clk
dout
din
buf32

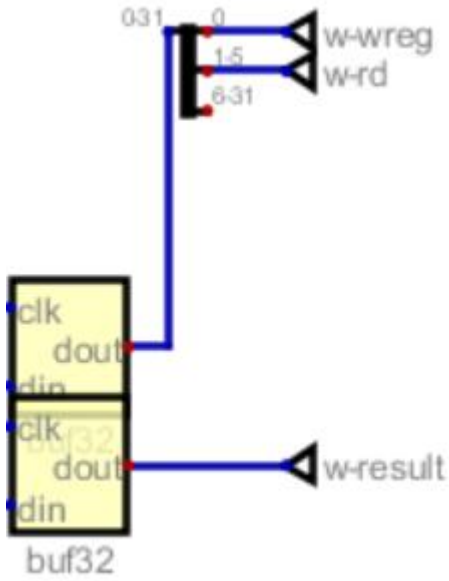
执行

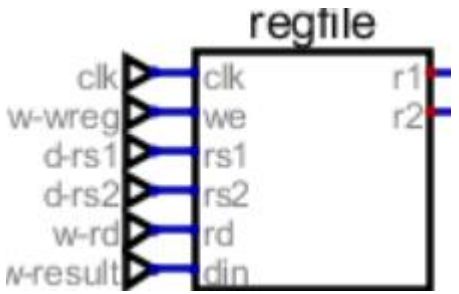

e-funct3
e-aluop
e-wreg
e-rs1
e-rs2
e-rd
e-wregSrc
e-pcSrc
e-jalr
e-btype
e-aluSrcB
e-store
e-load
e-funct3
e-wreg
e-rs1
e-rs2
e-rd
e-pcSrc1
e-store
e-load
0
clk
dout
din
buf32
e-wregSrc
alu
e-aluop
op
a
b
c
branch
External
aluResult
lui
auipc
pc+4
e-aluSrcB
add
e-jalr
even
din
dout
External
nextPC
pc + imm : branch
pc + imm : jal
rs1+ imm : jalr
branch
e-btype
e-pcSrc
e-pcSrc1
jal/jalr

访存


0-31
0-2
3
4-8
9-13
14-18
19
20
21
22-31
m-funct3
m-wreg
m-rs1
m-rs2
m-rd
m-pcSrc
m-store
m-load
m-wreg
m-rd
0
0
1-5
6-31
0-31
clk
dout
din
buf32
data
clk
m-store
m-funct3
clk
we
funct3
addr
din
dout
External
0
m-load
clk
clk
dout
din
buf32

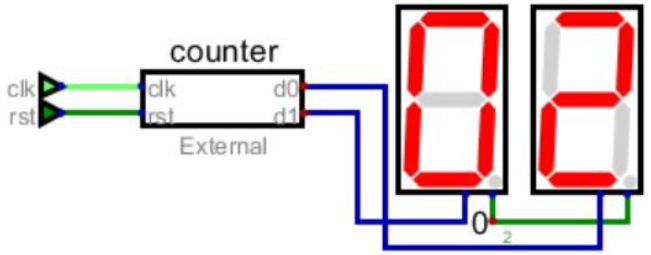
写回


031
0
1-5
6-31
w-wreg
w-rd
clk
dout
din
clk
dout
din
w-result
buf32


regfile
clk
w-wreg
d-rs1
d-rs2
w-rd
w-result
clk
we
rs1
rs2
rd
din
r1
r2

测试

```
.text
        li x1, 100
        li x2, 200
        li x3, 300
        li x4, 400
        li x5, 500
        li x6, 600
        add x7, x1, x2
        add x8, x2, x3
```



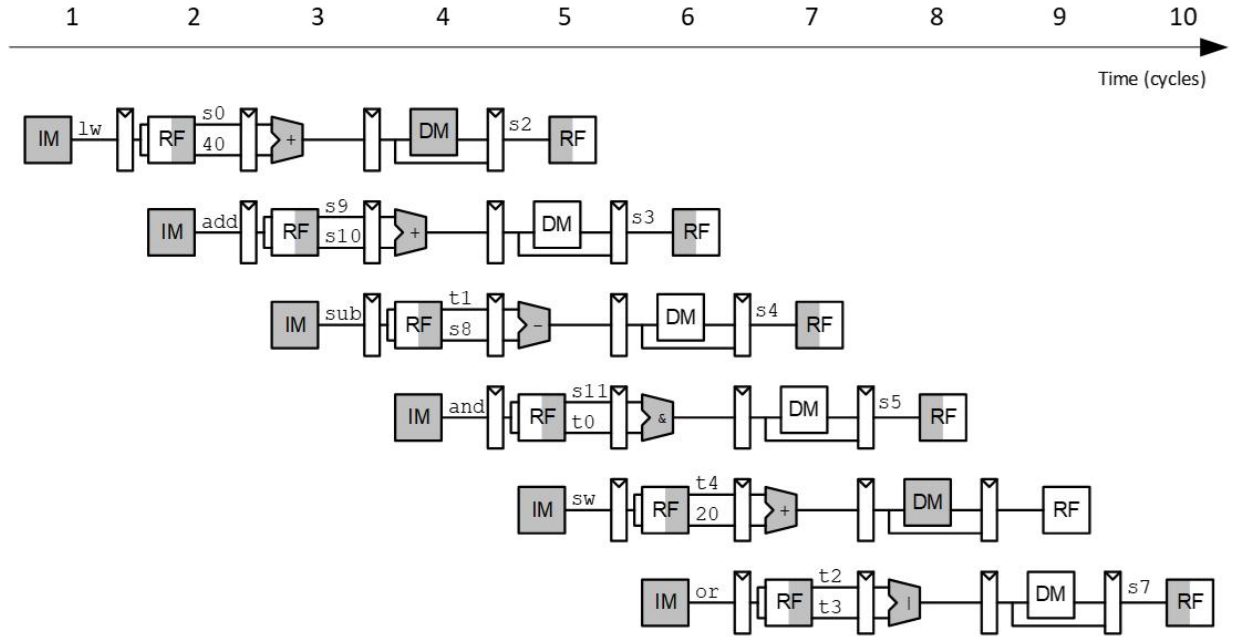
Address	Code	Basic	Source
0x00400000	0x06400093	addi x1, x0, 0x00000064	2: li x1, 100
0x00400004	0x0c800113	addi x2, x0, 0x000000c8	3: li x2, 200
0x00400008	0x12c00193	addi x3, x0, 0x0000012c	4: li x3, 300
0x0040000c	0x19000213	addi x4, x0, 0x00000190	5: li x4, 400
0x00400010	0x1f400293	addi x5, x0, 0x000001f4	6: li x5, 500
0x00400014	0x25800313	addi x6, x0, 0x00000258	7: li x6, 600
0x00400018	0x002083b3	add x7, x1, x2	8: add x7, x1, x2
0x0040001c	0x00310433	add x8, x2, x3	9: add x8, x2, x3

流水线冲突/相关（hazard）

- 流水线冲突/相关
  - 因指令间的依赖或资源冲突导致后续指令无法按预期执行的现象
- 结构相关
  - 硬件资源争用（如单端口存储器同时被IF和MEM阶段访问）
- 数据相关
  - 指令间的数据依赖导致后续指令需要等待前序指令的结果。
- 控制相关
  - 分支或跳转指令改变程序流，导致后续已取指令无效

流水线冲突/相关（hazard）

- 结构相关
  - 硬件资源争用（如单端口存储器同时被IF和MEM阶段访问）
  - 例如：
    - 在IF阶段取指令时，MEM阶段需要同时访问同一存储器加载数据。
    - 对某寄存器同时读写
  - 解决方法：
    - 增加硬件资源（如指令和数据存储器分离的哈佛架构）
    - 寄存器先写后读
    - 插入流水线停顿（Stall）




```
assign r1 = rs1 == 0? 0 :
            rs1 == rd ? din : regs[rs1];
assign r2 = rs2 == 0? 0 :
            rs2 == rd ? din : regs[rs2];
```

流水线冲突/相关（hazard）

```
add $s0, $t0, $t1   # $s0在EX阶段结束时才写入寄存器
sub $t2, $s0, $t3   # 需等待add指令的$s0值
```

- 数据相关
  - 指令间的数据依赖导致后续指令需要等待前序指令的结果。
  - RAW（Read After Write）：后指令需读取前指令未写入的数据（最常见）。
  - WAR（Write After Read）：后指令写入前指令需读取的寄存器（常见于乱序流水线）
  - WAW（Write After Write）：后指令覆盖前指令未完成的写入（常见于多周期指令）
- 解决方法
  - 前推（Forwarding / Bypassing）：将结果直接从EX或MEM阶段传递到后续指令的输入，避免等待WB阶段。
  - 插入Stall：当无法前推时（如Load后紧接使用数据的指令），插入“气泡”暂停流水线。

流水线冲突/相关（hazard）

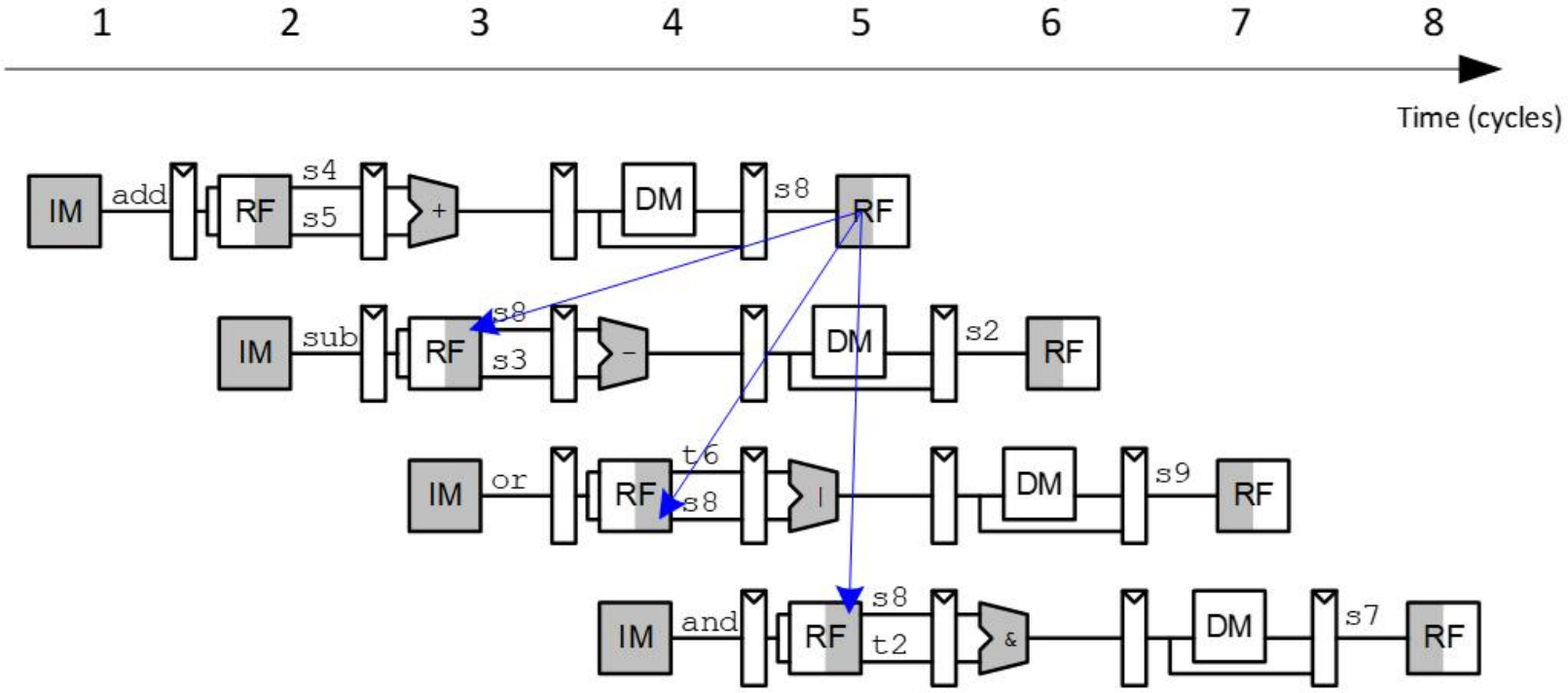
```
beq $s0, $s1, label   # 分支结果在EX阶段才能确定
add $t0, $t1, $t2     # 可能被错误取指并执行
```

- 控制相关
  - 分支或跳转指令改变程序流，导致后续已取指令无效
- 解决方法
  - 插入stall、flush
  - 静态分支预测：假设分支不成立，继续执行后续指令；若预测错误则清空流水线。
  - 动态分支预测：基于历史记录（如分支目标缓冲BTB）预测分支方向。
  - 延迟槽（Delay Slot）：MIPS架构中，分支指令后的1条指令总是执行（需编译器填充有效指令）。

数据相关（data hazard）

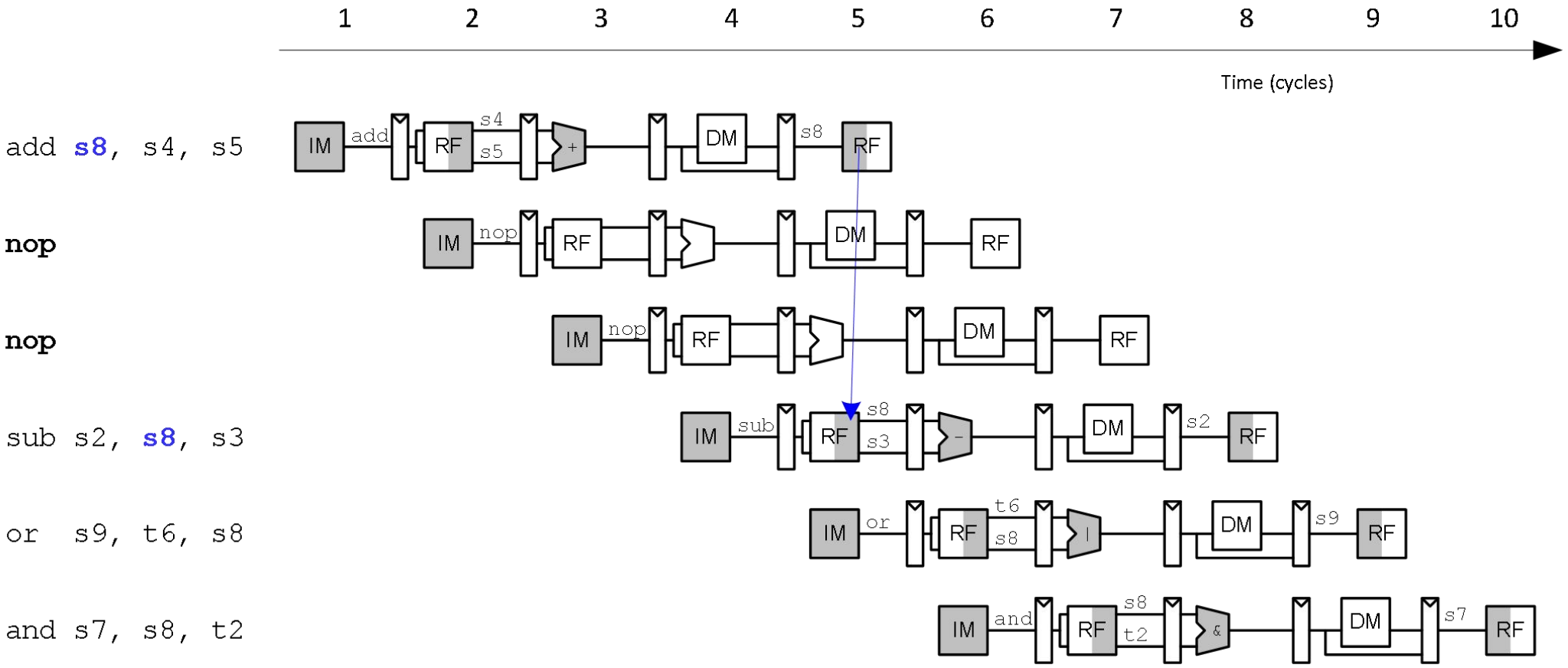
- RAW（Read After Write）：后指令需读取前指令未写入的数据

```
add s8, s4, s5
sub s2, s8, s3
or  s9, t6, s8
and s7, s8, t2
```



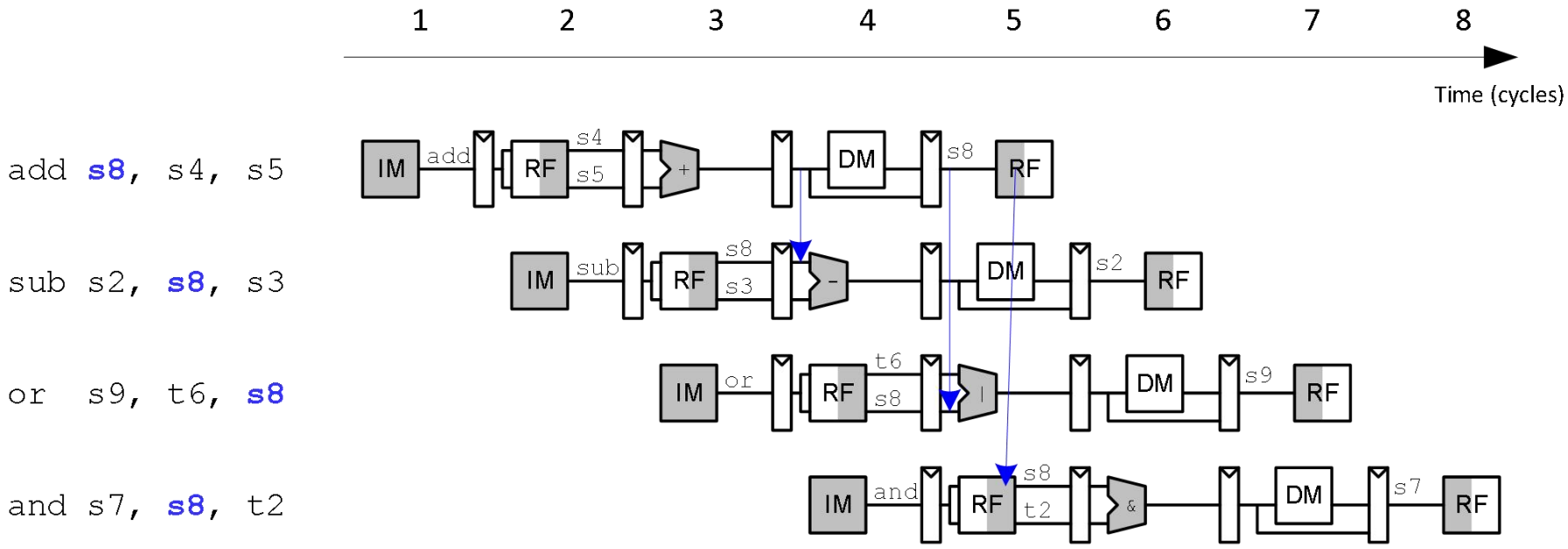
数据相关的解决：插入气泡

- 插入空指令（nop）直到依赖的结果可用



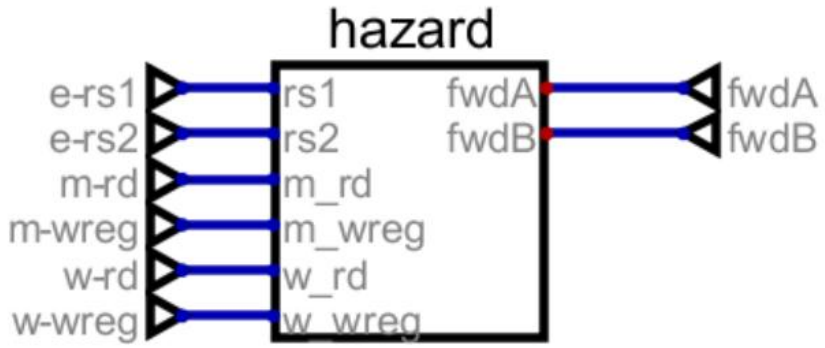
数据前递：Data forward

- 检查执行阶段的源寄存器是否与访存或写回阶段的指令目标寄存器匹配。
- 如果是，则转发结果。



■ 数据前递：Data forward

- 情况1：执行阶段Rs1或Rs2等于访存阶段Rd?
  - 是则从访存阶段的alu-result转发
- 情况2：执行阶段Rs1或Rs2等于写回阶段Rd?
  - 是则从写回阶段转发
- 情况3：否则使用从寄存器文件中读取的值
-

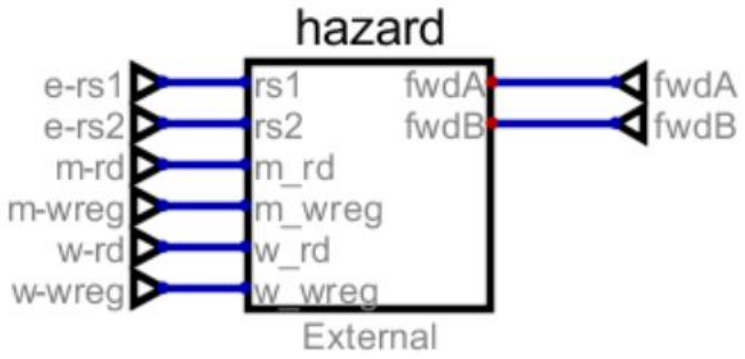


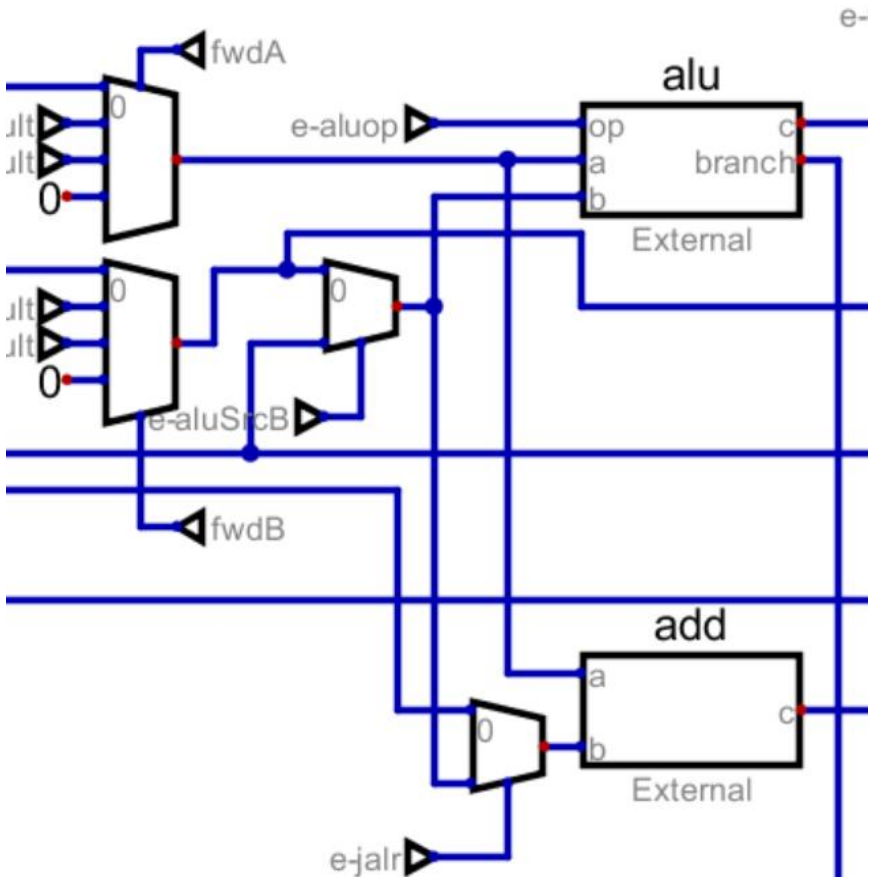

```
module hazard(
    input [4:0] rs1,
    input [4:0] rs2,
    input [4:0] m_rd,
    input m_wreg,
    input [4:0] w_rd,
    input w_wreg,
    output [1:0] fwdA,
    output [1:0] fwdB
);
    assign fwdA = (rs1 == m_rd && m_wreg && rs1 != 0) ? 1 :
                  (rs1 == w_rd && w_wreg && rs1 != 0) ? 2 : 0;

    assign fwdB = (rs2 == m_rd && w_wreg && rs1 != 0) ? 1 :
                  (rs2 == w_rd && w_wreg && rs2 != 0) ? 2 : 0;

endmodule
```

数据前递：Data forward





■ 数据前递：Data forward

- 测试 .text

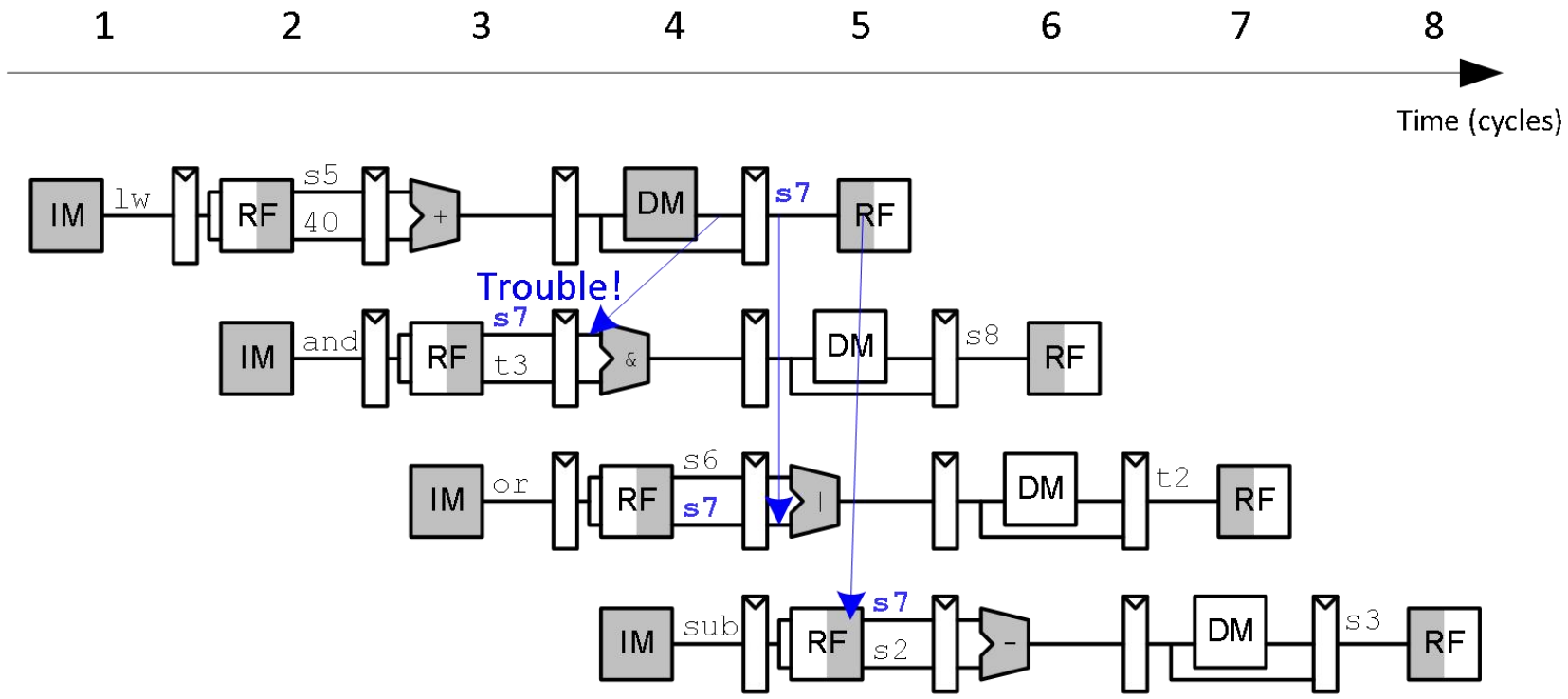
```
li x1, 100
addi x2, x1, 1
addi x3, x1, 2
addi x4, x1, 3
addi x5, x1, 4
```

Address	Code	Basic	Source
0x00400000	0x06400093	addi x1,x0,0x00000064	2: li x1, 100
0x00400004	0x00108113	addi x2,x1,1	3: addi x2, x1, 1
0x00400008	0x00208193	addi x3,x1,2	4: addi x3, x1, 2
0x0040000c	0x00308213	addi x4,x1,3	5: addi x4, x1, 3
0x00400010	0x00408293	addi x5,x1,4	6: addi x5, x1, 4

Load-use的数据相关

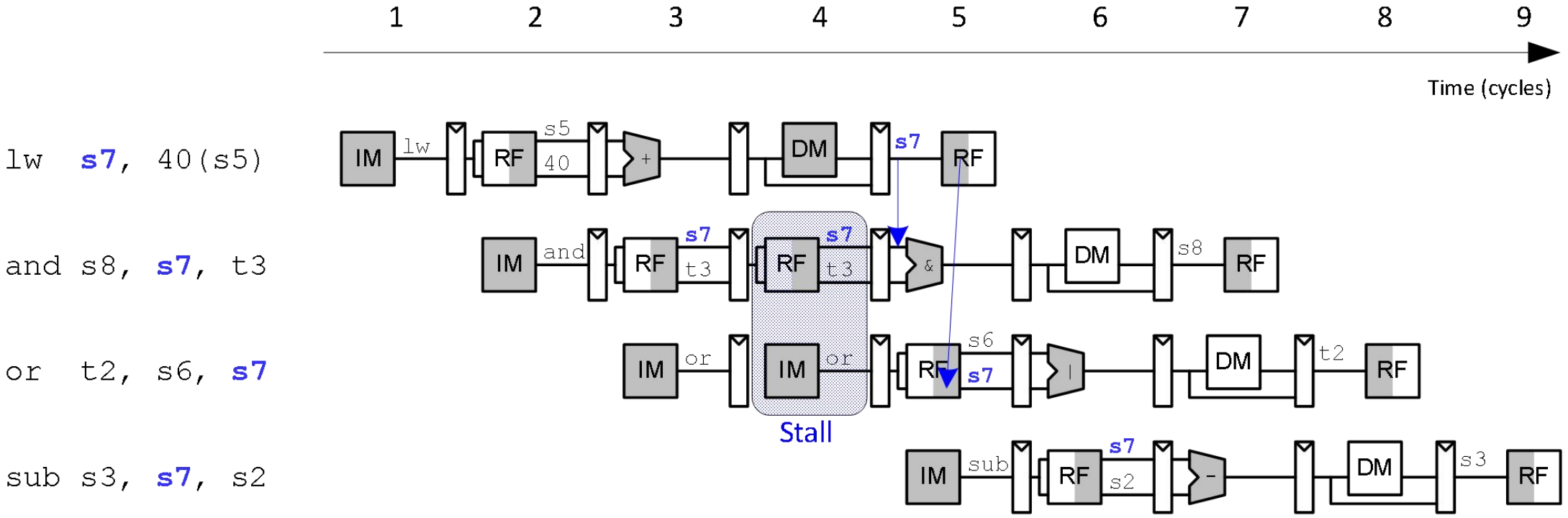
- Load指令得到数据在访存阶段末，因此要在写回阶段数据才有效

```
lw   s7, 40(s5)
and s8, s7, t3
or   t2, s6, s7
sub s3, s7, s2
```



Load-use的数据相关：插入stall

- 寄存器增加stall、flush信号
  - 当stall为真时，寄存器保持内容不变
  - 当flush为真时，寄存器值为0



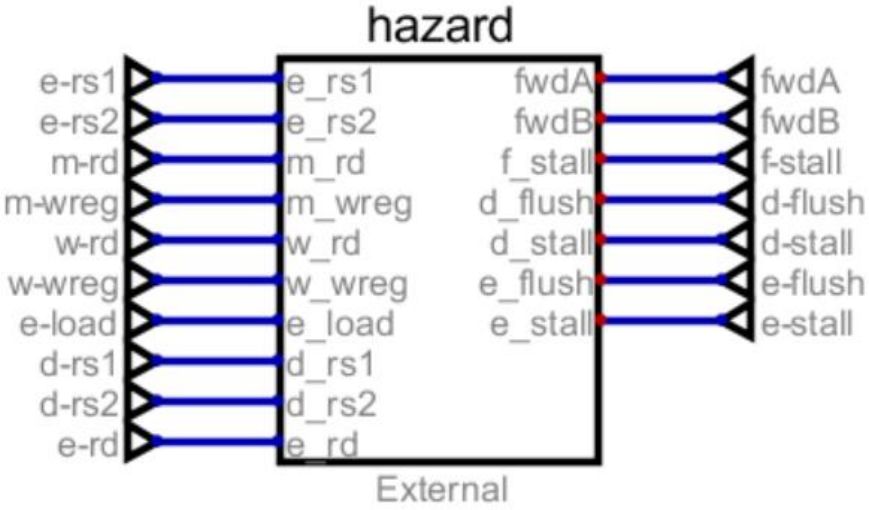
Load Stall逻辑

- 译指阶段的源寄存器是否与执行阶段的目标寄存器相同?
- 并且执行阶段的指令是lw吗?

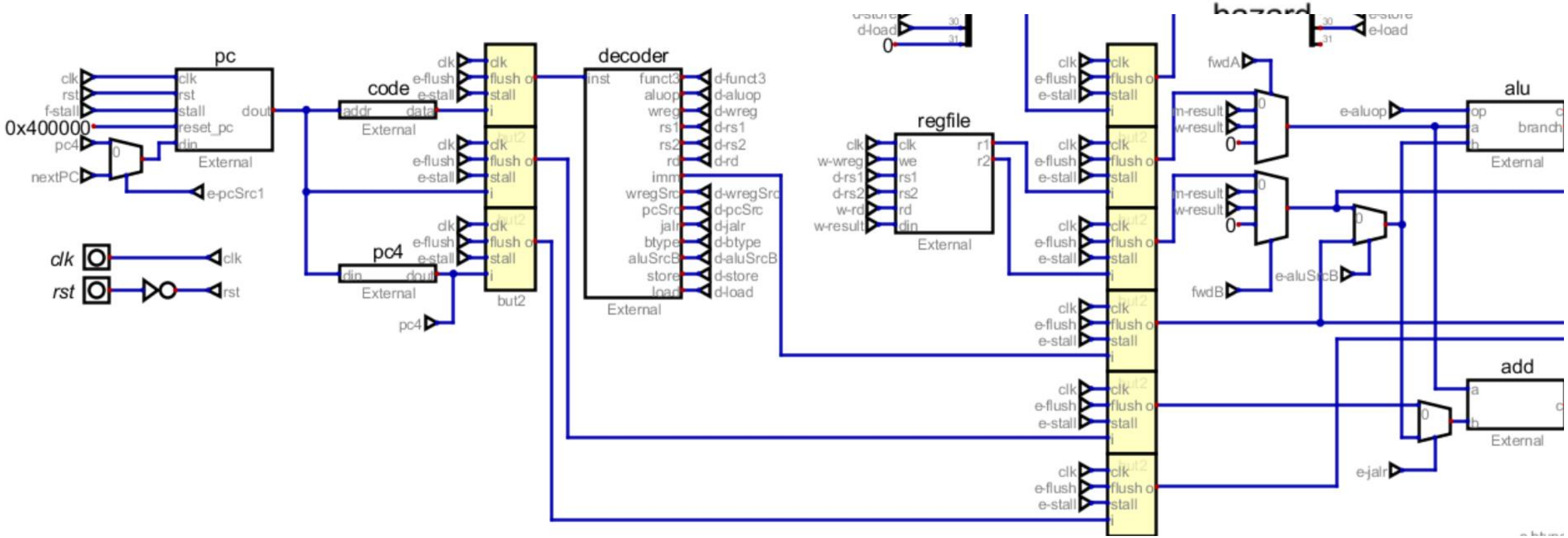
lwStall = ((d-rs1 == e-rd) OR (d-rs2 == e-rd)) AND e-load

f-stall = d-stall = e-flush = lwStall

（停止取指和译指，并刷新执行阶段。）



Load Stall逻辑



Load Stall逻辑

- 测试

```
.data
x:      .word 100
.text
        lw t0, x
        addi t1, t0, 1
        addi t2, t0, 2
        addi t3, t0, 3
        addi t4, t0, 4
```

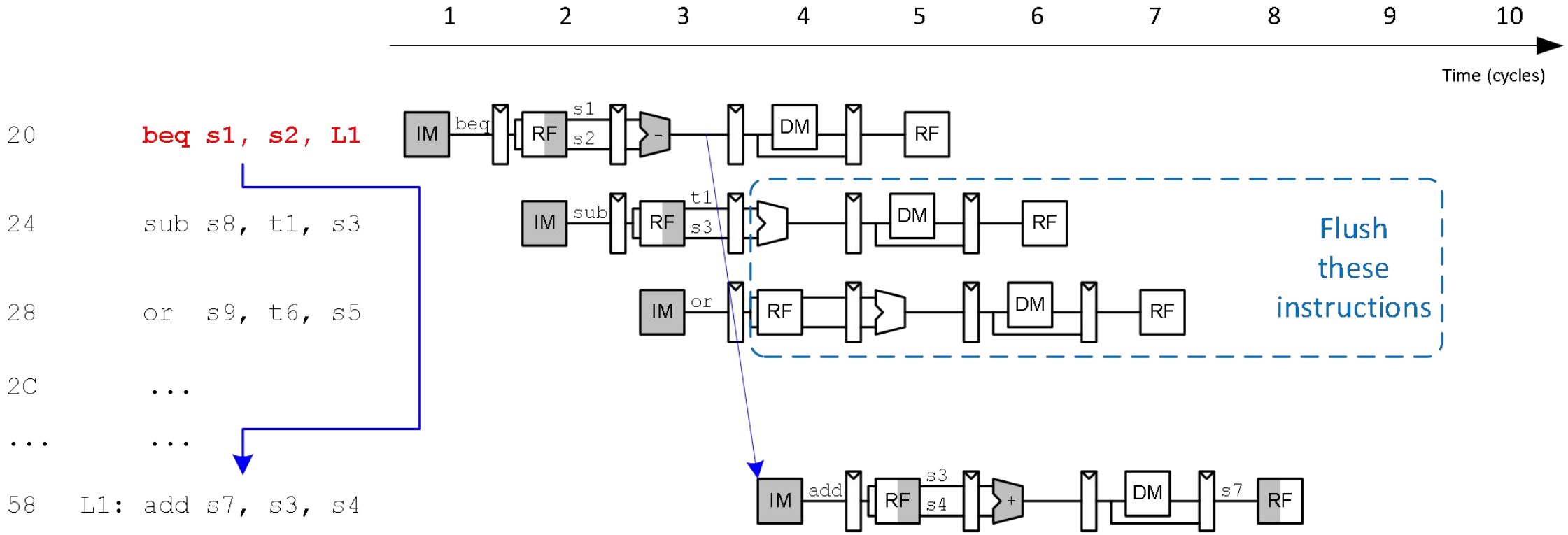
Address	Code	Basic	Source
0x00400000	0x0fc10297	auipc x5,0x0000fc10	4: lw t0, x
0x00400004	0x0002a283	lw x5,0(x5)	
0x00400008	0x00128313	addi x6,x5,1	5: addi t1, t0, 1
0x0040000c	0x00228393	addi x7,x5,2	6: addi t2, t0, 2
0x00400010	0x00328e13	addi x28,x5,3	7: addi t3, t0, 3
0x00400014	0x00428e93	addi x29,x5,4	8: addi t4, t0, 4

控制相关

- Branch指令
  - 条件转移指令采用静态预测，假定不转移
  - 直到执行阶段才确定转移地址
  - 条件转移指令到了执行阶段，后面又取了两条指令
    - 如果发生分支，这2条指令必须刷新（作废）
- jal、jalr
  - 无条件转移指令到了执行阶段才能得到转移地址，因此多取了两条指令
  - 这2条指令必须刷新（作废）

控制相关

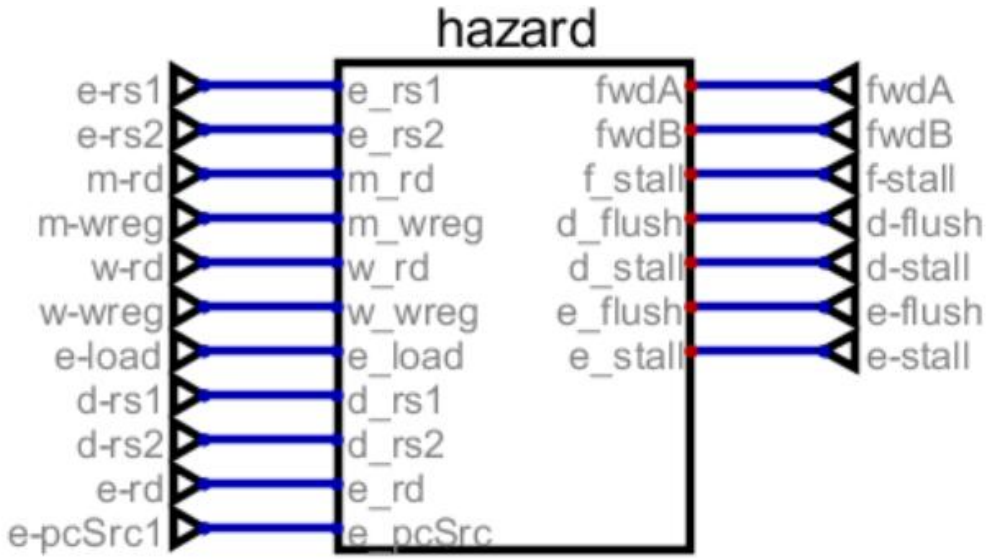
- 发生转移后必须要刷新后面多取的两条指令



控制相关

- 如果在执行阶段发生转移（e-pcSrc1），需要在Fetch和Decode阶段刷新指令
- 通过使用d-flushD和e-flush清除译指和执行阶段的寄存器

- d-flush = e-pcSrc1
- e-flush=lwStall或e-pcSrc1



■ 控制相关：

- 测试

```
.text
        li t0, 100
        li t1, 200
        bne t0, t1, not_equal
        li t2, 300
        j done
not_equal:
        li t3, 400
        li t4, 500
done:
```

Address	Code	Basic	Source
0x00400000	0x06400293	addi x5, x0, 0x00000064	2: li t0, 100
0x00400004	0x0c800313	addi x6, x0, 0x000000c8	3: li t1, 200
0x00400008	0x00629663	bne x5, x6, 0x0000000c	4: bne t0, t1, not_equal
0x0040000c	0x12c00393	addi x7, x0, 0x0000012c	5: li t2, 300
0x00400010	0x00c0006f	jal x0, 0x0000000c	6: j done
0x00400014	0x19000e13	addi x28, x0, 0x00000190	8: li t3, 400
0x00400018	0x1f400e93	addi x29, x0, 0x000001f4	9: li t4, 500

异常和中断

- 异常可能发生在各阶段，发生后，其后面的所有的指令应作废（精确异常）
  - 异常处理后的状态必须等价于所有异常前指令完成，异常后指令未执行。
  - 流水线中存在多条重叠执行的指令，需准确追踪异常来源的指令位置
  - 在某个时刻可能有多条指令触发异常
- 中断可能在任何时刻发生，需在指令边界或特定周期内响应，避免数据损坏
  - 中断应尽快能响应

提高流水线性能

- 增加流水线级数
  - 细分流水线：将流水线划分为更多级数，每级执行更简单的任务，从而提高时钟频率。
  - 平衡各级负载：确保各级负载均衡，避免瓶颈。
  - 数据相关和控制相关随着流水线级数的增加，会导致更多的性能损失
- 指令调度
  - 重排指令顺序，消除或减少指令之间的数据依赖及控制相关
    - 静态调度：编译器
    - 动态调度：记分牌算法、tamosulo算法
- 分支预测
  - 高级预测算法：使用如两级自适应预测器，提高预测精度。
  - 分支目标缓冲区（BTB）：存储分支目标地址，加快分支指令执行

提高流水线性能

- 推测执行
  - 控制推测：预测分支结果并提前执行后续指令，预测错误时回滚。
  - 数据推测：预测数据依赖关系，提前执行相关指令。
- 超标量架构
  - 多发射：每个时钟周期发射多条指令，提高并行度。
  - 多执行单元：增加ALU、FPU等执行单元，支持并行执行。

RISC-V®

流水线处理器

- 基本概念
- 流水线处理器
- 数据冲突
- 控制冲突
- 提高流水线性能