18-747 Lecture 3: Pipeline Interlock

James C. Hoe
Dept of ECE, CMU
September 5, 2001

Reading Assignments: *S&L Ch 2, pp 34-51*

Announcements: My Office hours: MW, 4:30-5:30 PM HH D201
*** Aaron, TT 4:30-5:30, undergraduate lounge ***
Recitation starts next week: F 2:30-3:20 PH1112 (this room)

Handouts: Handout #3 Problem Set 1

Hazard Analysis Procedure

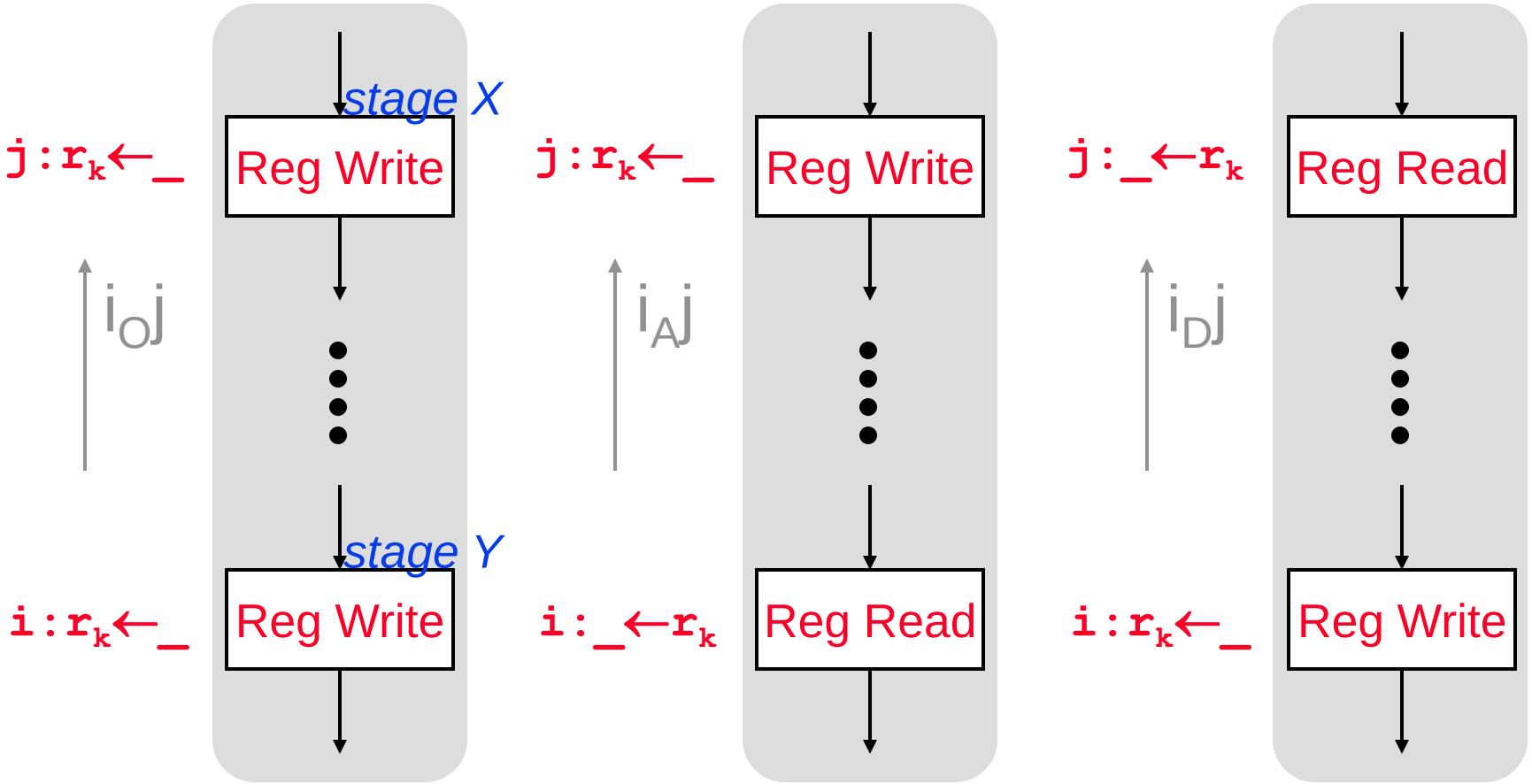
- Memory Data Dependences
  - Output Dependence (WAW)
  - Anti Dependence (WAR)
  - True Data Dependence (RAW)
- Register Data Dependences
  - Output Dependence (WAW)
  - Anti Dependence (WAR)
  - True Data Dependence (RAW)
- Control Dependences

Terminology

- Pipeline Hazards:
  - Potential violations of program dependences
  - Must ensure program dependences are not violated
- Hazard Resolution:
  - Static Method: Performed at compiled time in software
  - Dynamic Method: Performed at run time using hardware

    Stall, Flush or Forward
- Pipeline Interlock:
  - Hardware mechanisms for dynamic hazard resolution
  - Must detect and enforce dependences at run time

Necessary Conditions for Data Hazards

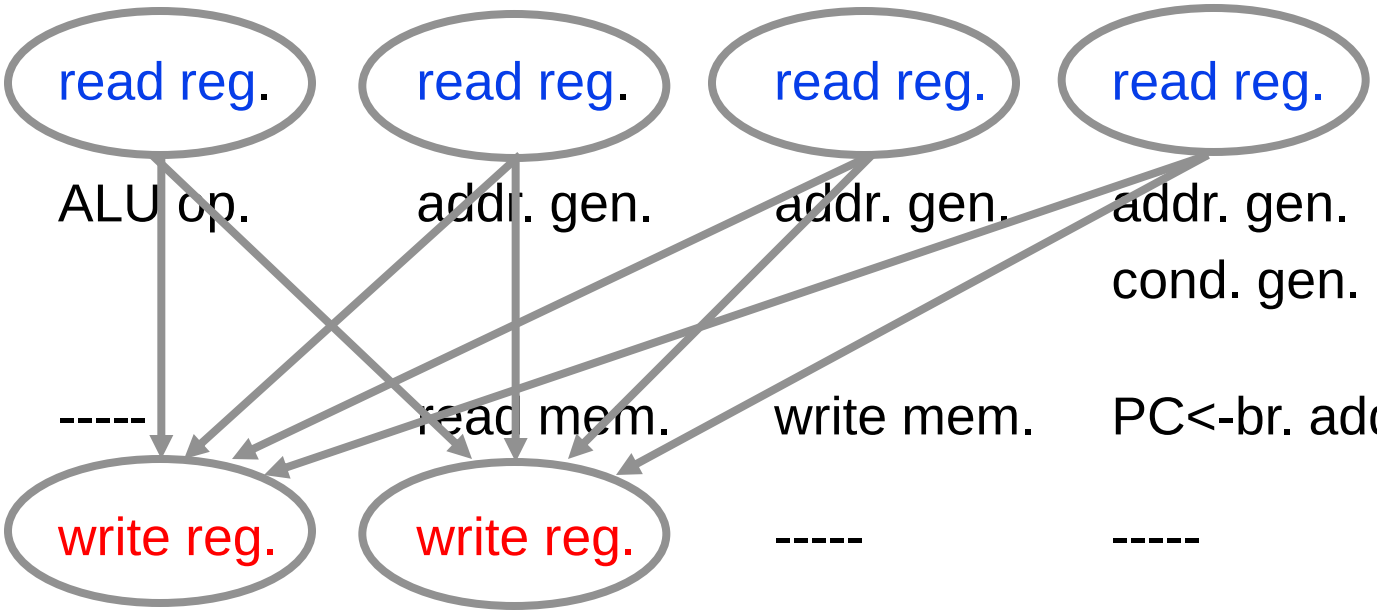


$dist(i,j) \leq dist(X,Y) \Rightarrow$ Hazard!!

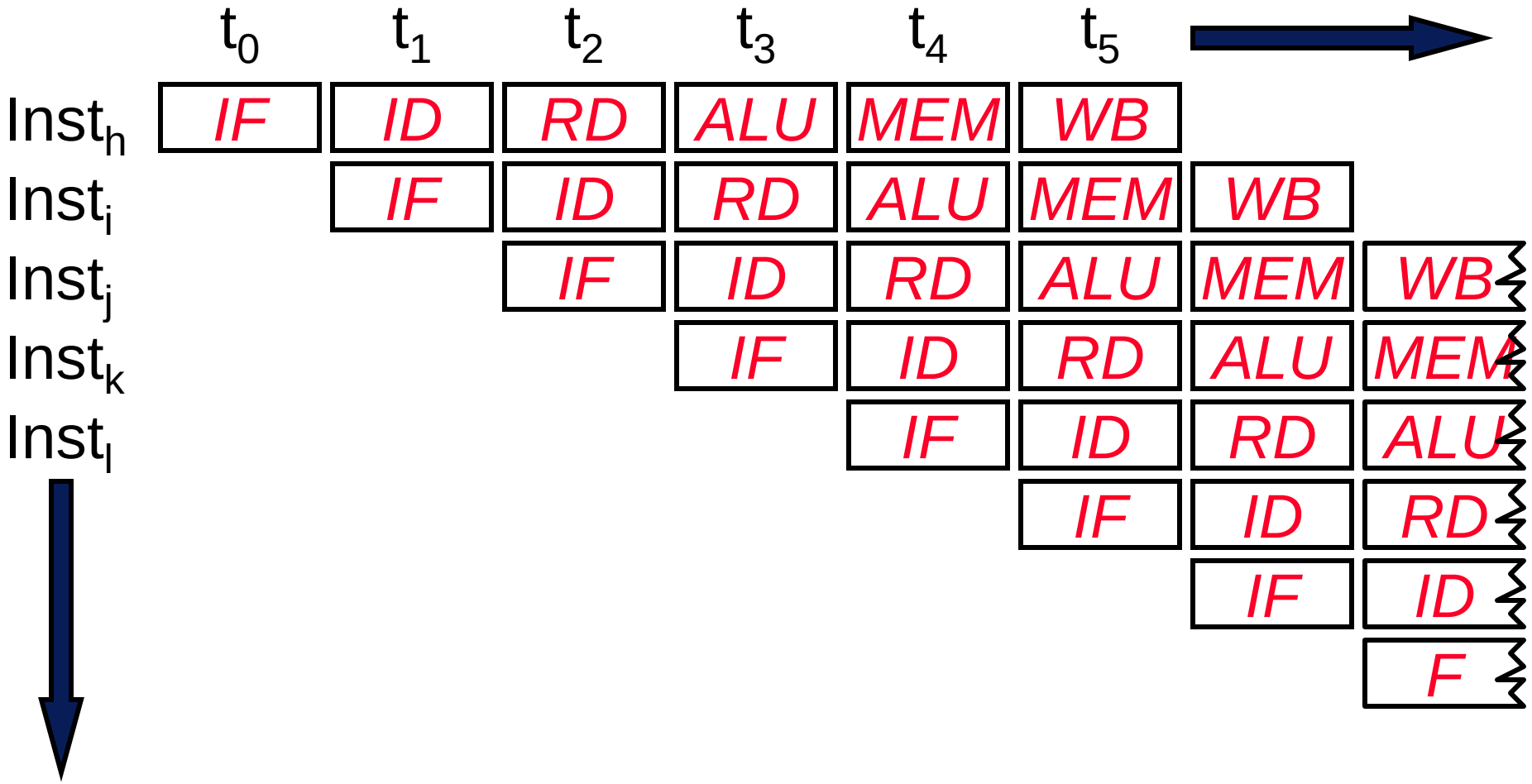
$dist(i,j) > dist(X,Y) \Rightarrow$ Safe

Inter-instruction Data Hazards

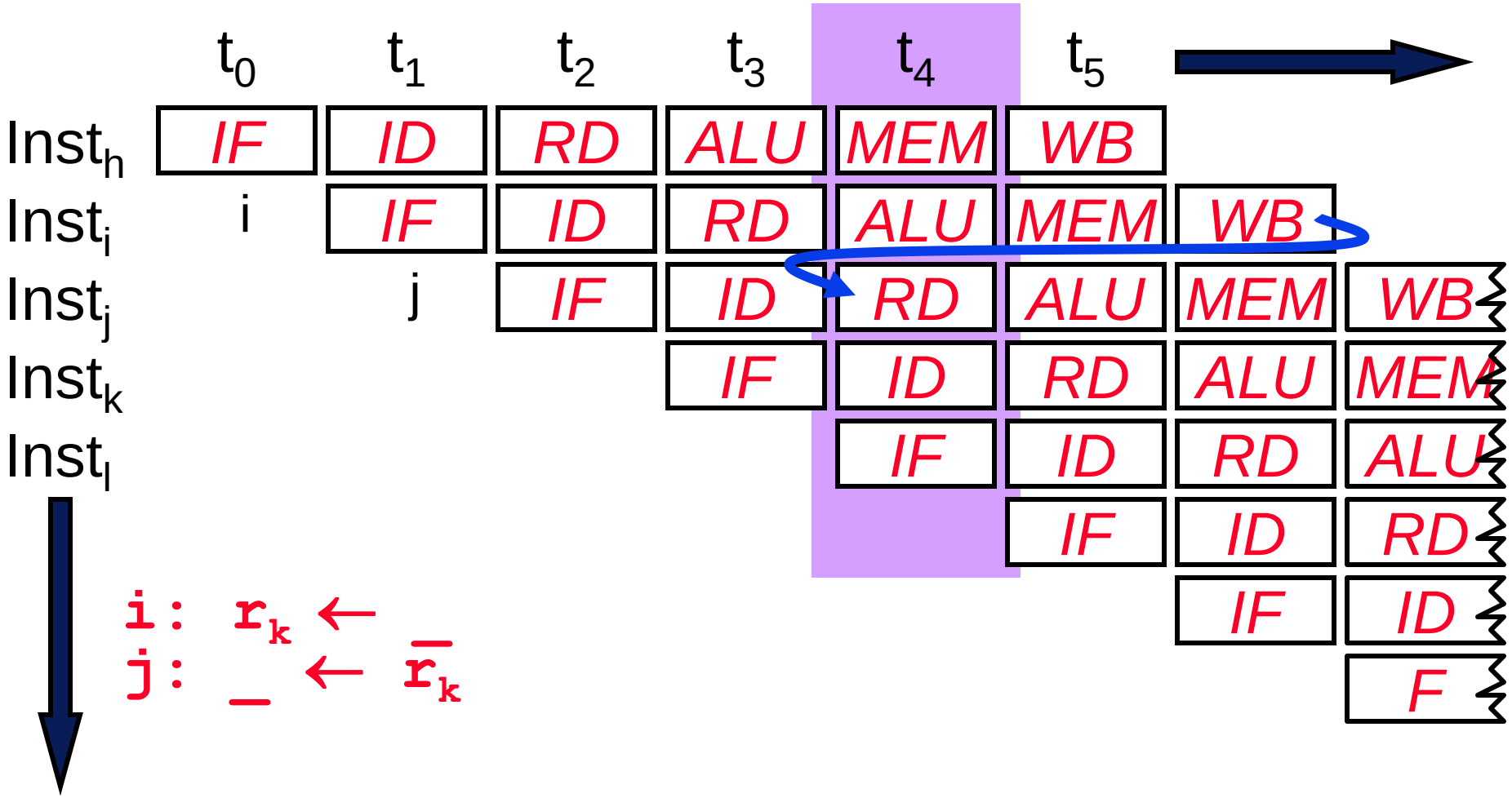
Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	ALU op.	addr. gen.	addr. gen.	addr. gen. cond. gen.
5. MEM	-----	read mem.	write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----



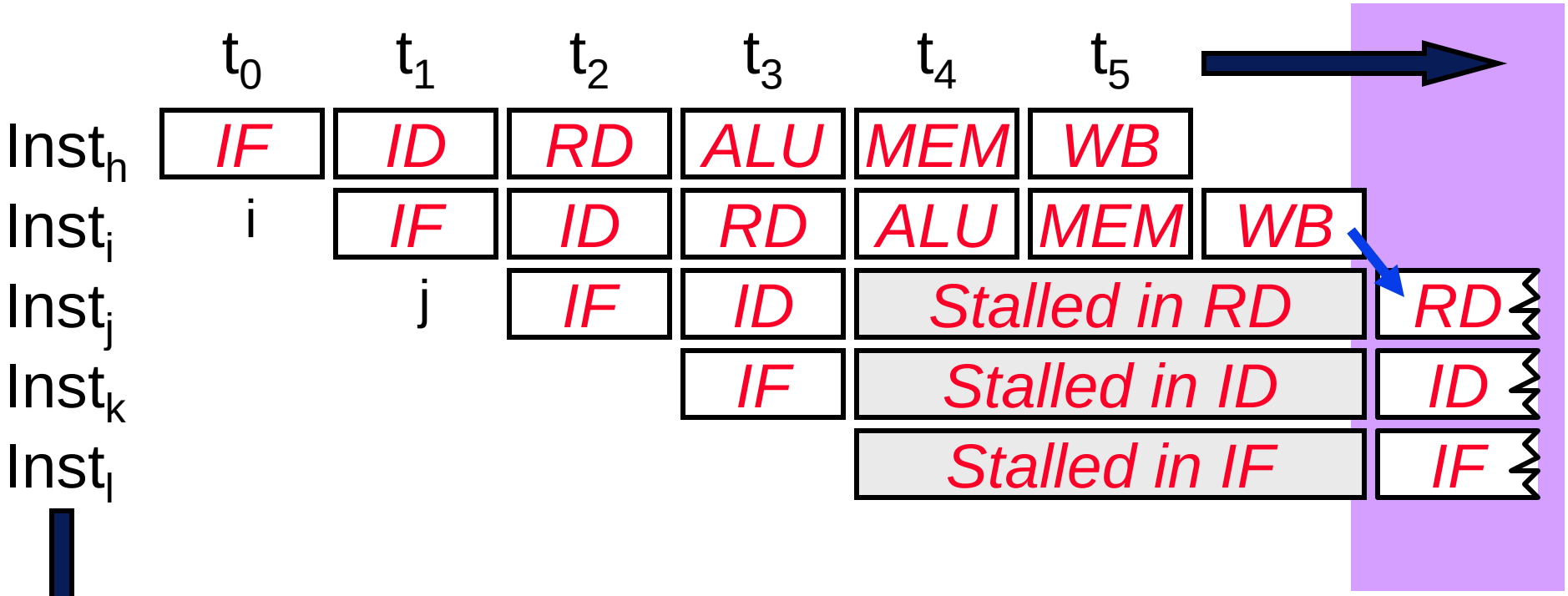
Pipelining: Steady State



Pipelining: Data Hazards



Pipelining: Stall on Data Hazard




```
i: rk ← _
bubble
bubble
bubble
j: _ ← rk
```

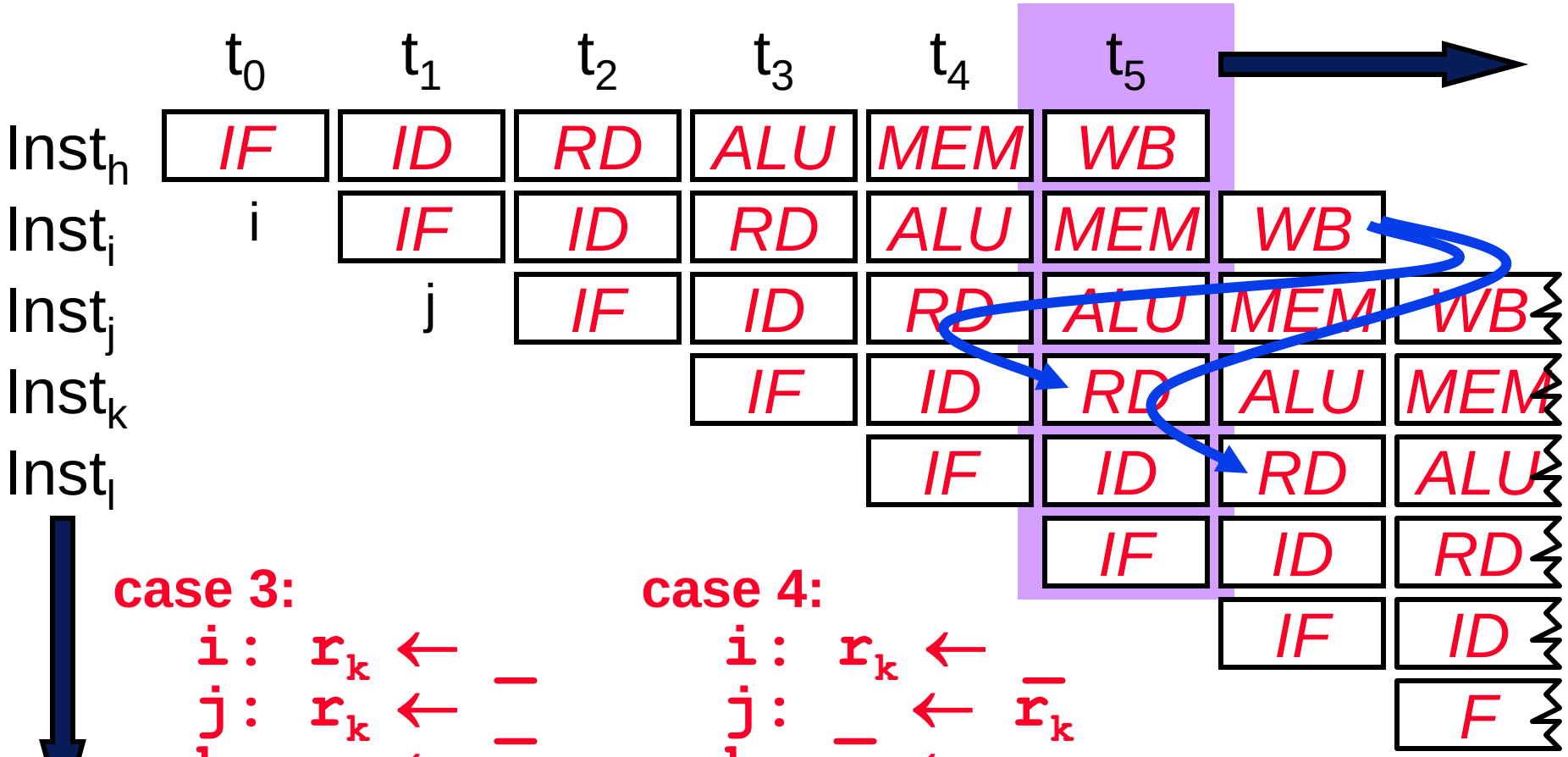
Stall==make the younger instruction wait until the hazard has passed

1. *stop all up-stream stages*
2. *drain all down-stream stages*

Pipeline: Stall on Data Hazard

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_i	I_j	I_k	I_l *stall*	I_l *stall*	I_l *stall*	I_l				
ID	I_h	I_i	I_j	I_k *stall*	I_k *stall*	I_k *stall*	I_k	I_l			
RD		I_h	I_i	I_j *stall*	I_j *stall*	I_j *stall*	I_j	I_k	I_l		
ALU			I_h	I_i	nop	nop	nop	I_j	I_k	I_l	
MEM				I_h	I_i	nop	nop	nop	I_j	I_k	I_l
WB					I_h	I_i	nop	nop	nop	I_j	I_k

What should these cases look like?



case 3:

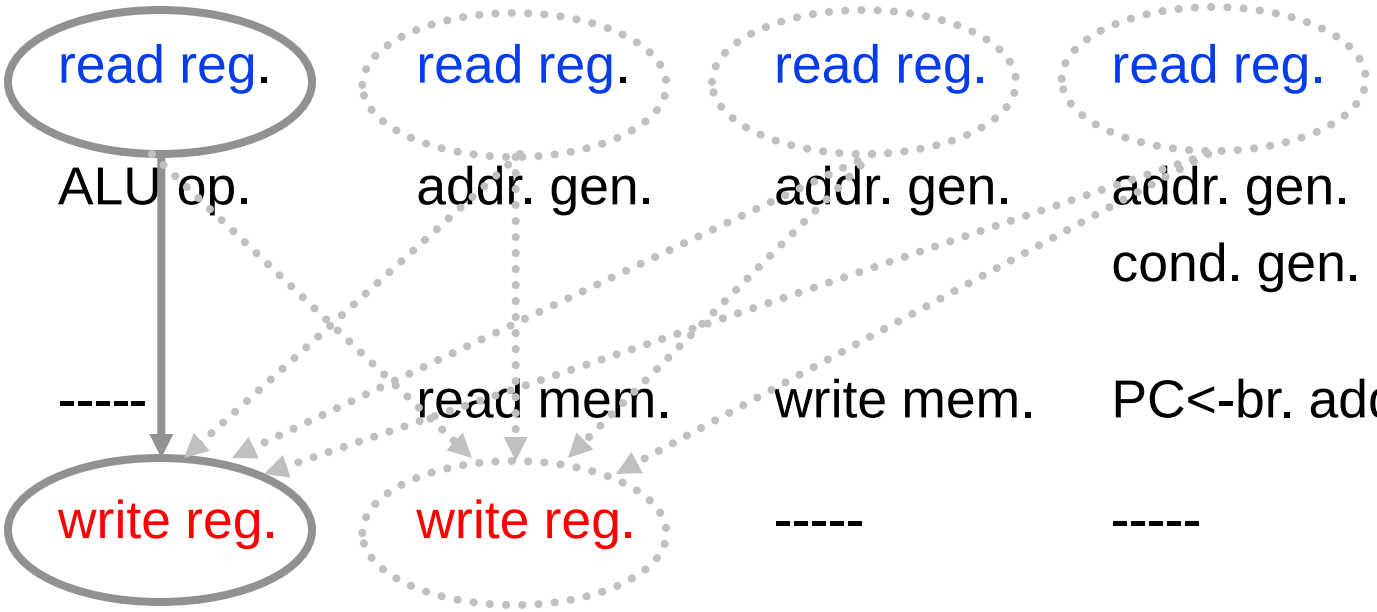
```
i: rk ← _
j: rk ← _
k: rk ← _
l: _  ← rk
```

case 4:

```
i: rk ←
j: _  ← rk
k: _  ← rk
l: _  ← rk
```

Stall Conditions

Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	ALU op.	addr. gen.	addr. gen.	addr. gen. cond. gen.
5. MEM	-----	read mem.	write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----



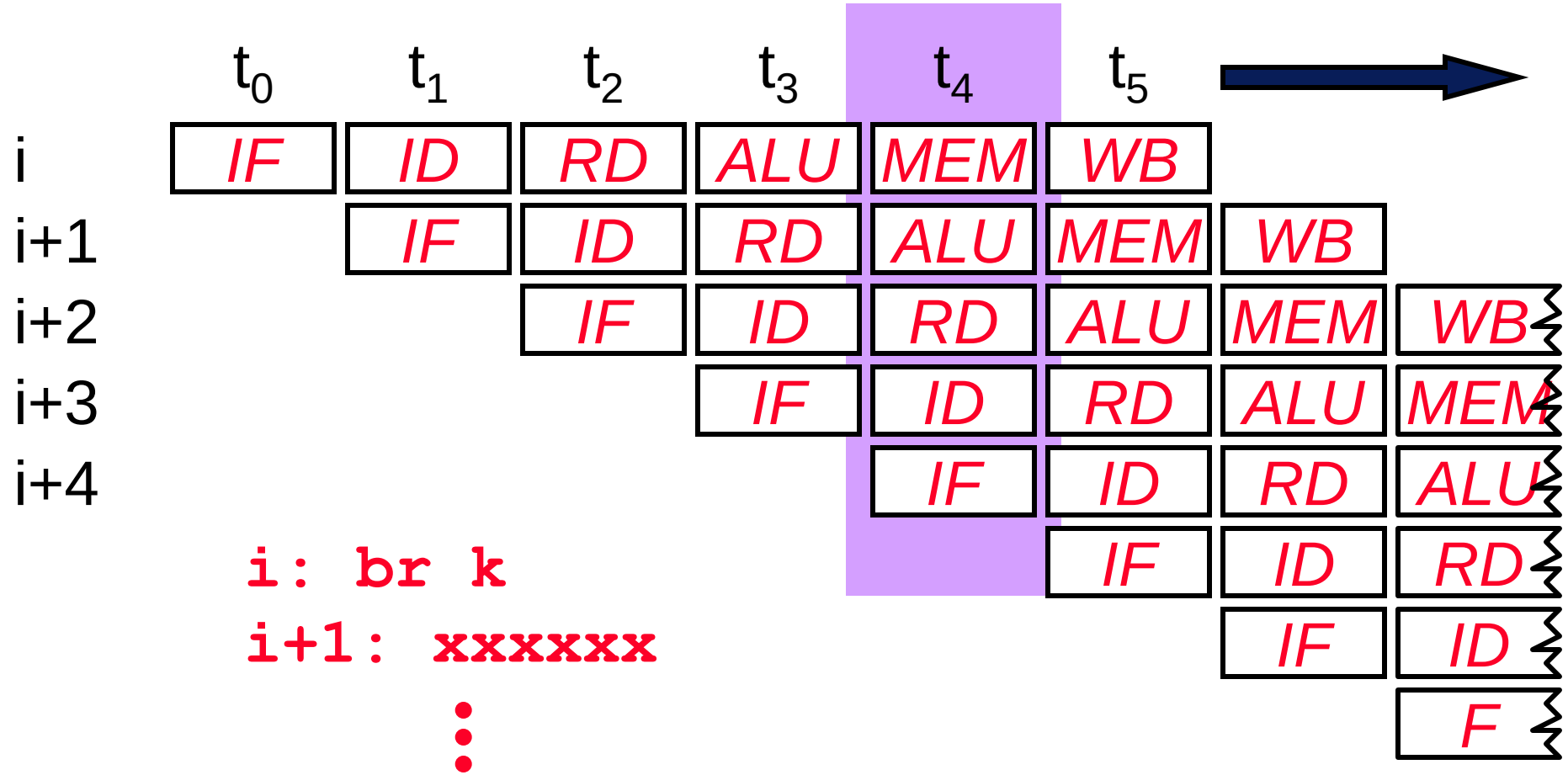
How many scenarios could trigger a particular hazard?

Inter-instruction Control Hazards

Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	ALU op.	addr. gen.	addr. gen.	addr. gen. cond. gen.
5. MEM	-----	read mem.	write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----

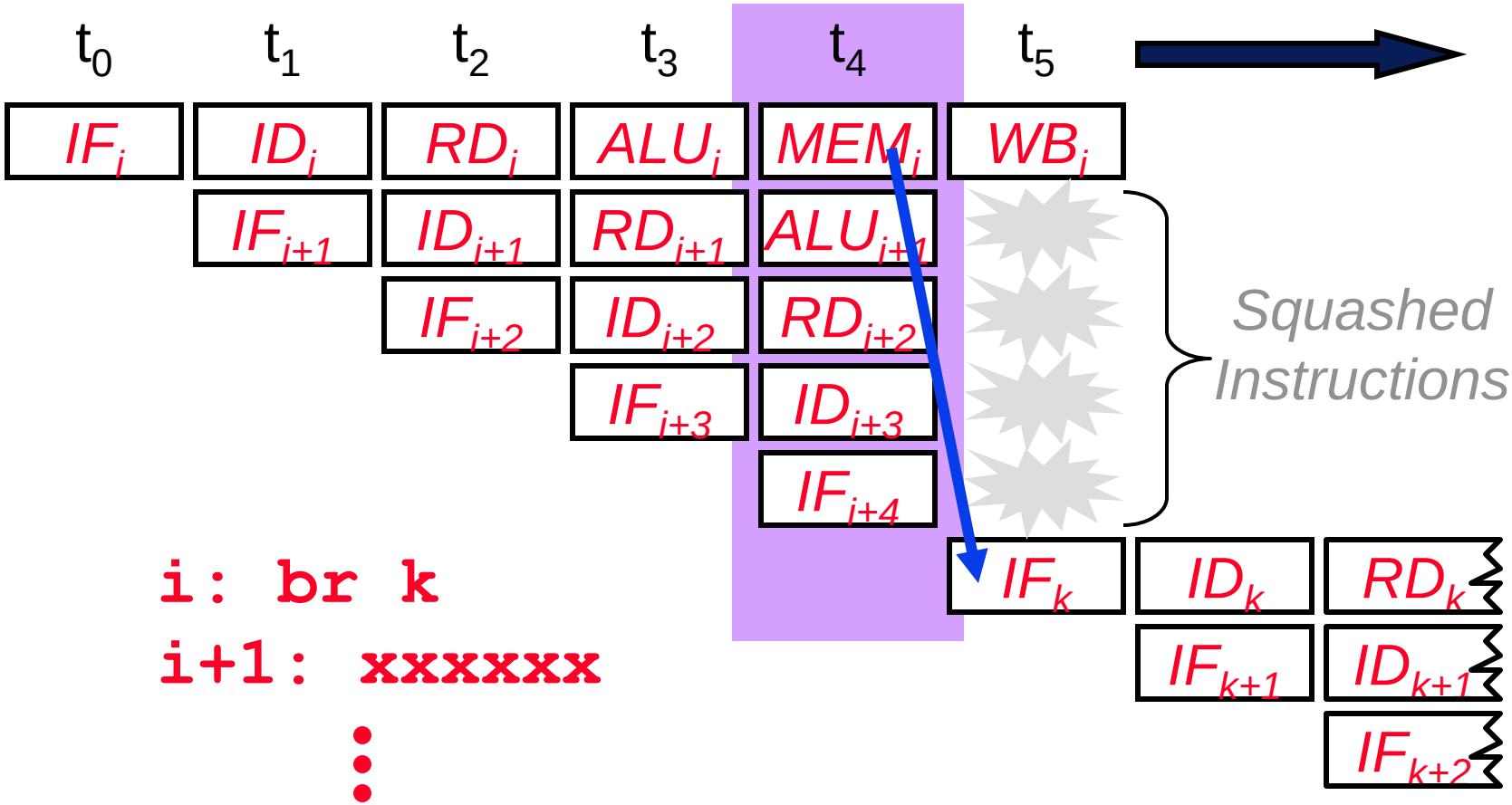
Can we use pipeline stall to resolve control hazards?

Pipeline: Control Hazard




```
i:   br k
i+1: xxxxxx
       :
k:   yyyyyy
```

Pipeline: Control Hazard

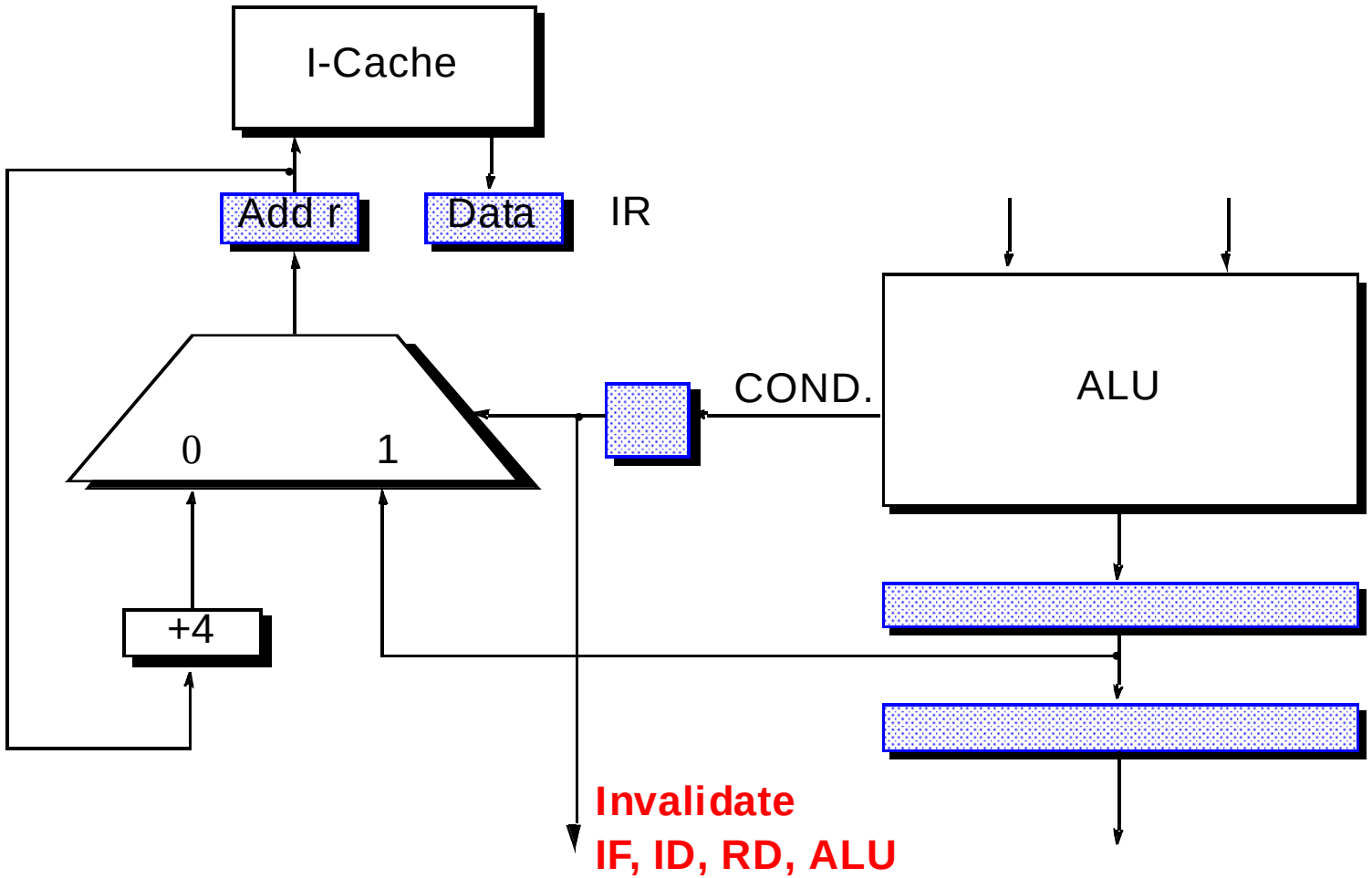



```
i: br k
i+1: xxxxxx
     ⋮
k: yyyyyy
```

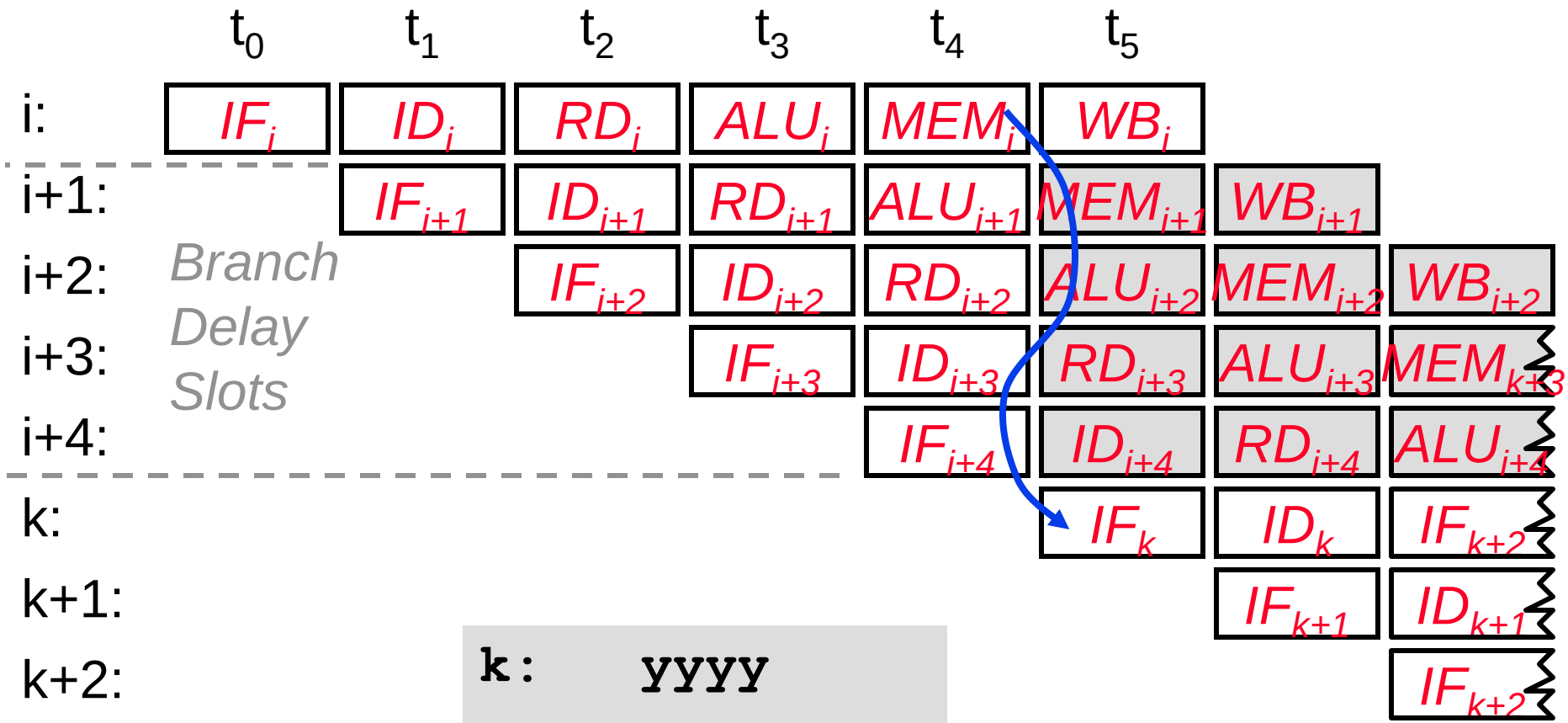
Pipeline Flush on Control Hazards

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_i	I_{i+1}	I_{i+2}	I_{i+3}	I_{i+4}	I_k	I_{k+1}	I_{k+2}	I_{k+3}	I_{k+4}	I_{k+5}
ID		I_i	I_{i+1}	I_{i+2}	I_{i+3}	nop	I_k	I_{k+1}	I_{k+2}	I_{k+3}	I_{k+4}
RD			I_i	I_{i+1}	I_{i+2}	nop	nop	I_k	I_{k+1}	I_{k+2}	I_{k+3}
ALU				I_i	I_{i+1}	nop	nop	nop	I_k	I_{k+1}	I_{k+2}
MEM					I_i	nop	nop	nop	nop	I_k	I_{k+1}
WB						I_i	nop	nop	nop	nop	I_k

Branch Instruction Interlock:



Delayed Branches

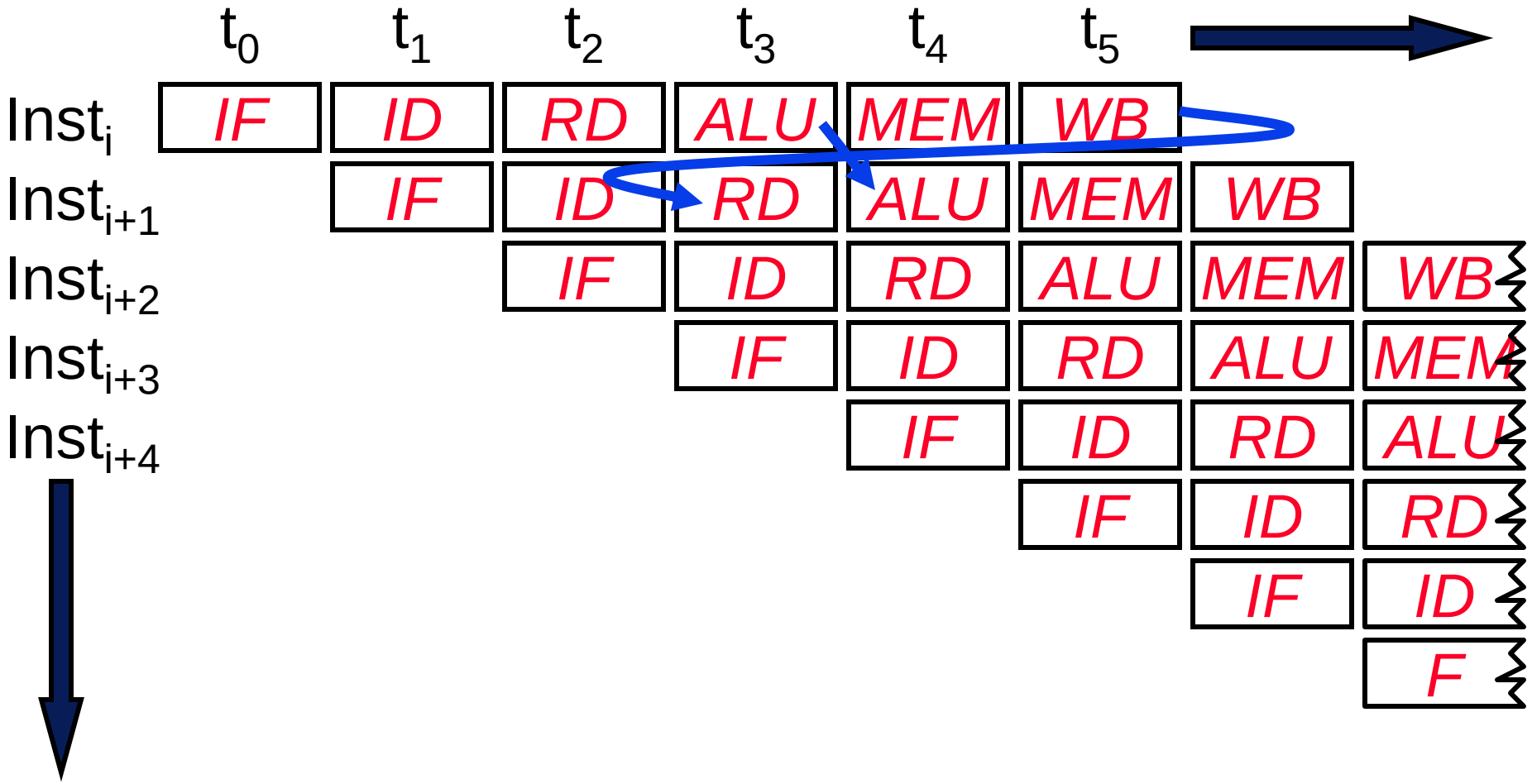



```
k:    yyyy
      . . . .
i:    Br. to k:
i+1:  xxxx
      . . . .
```

Penalties Due to Stalling for RAW

Leading $Inst_i$	ALU	Load	Branch
Trailing $Inst_j$	ALU, L/S, Br.	ALU, L/S, Br.	ALU, L/S, Br.
Hazard register	Int. Reg. (Ri)	Int. Reg. (Ri)	PC
Register WRITE stage (i)	WB (stage 6)	WB (stage 6)	MEM (stage 5)
Register READ stage (j)	RD (stage 3)	RD (stage 3)	IF (stage 1)
RAW distance or penalty:	3 cycles	3 cycles	4 cycles

Forwarding Path Analysis



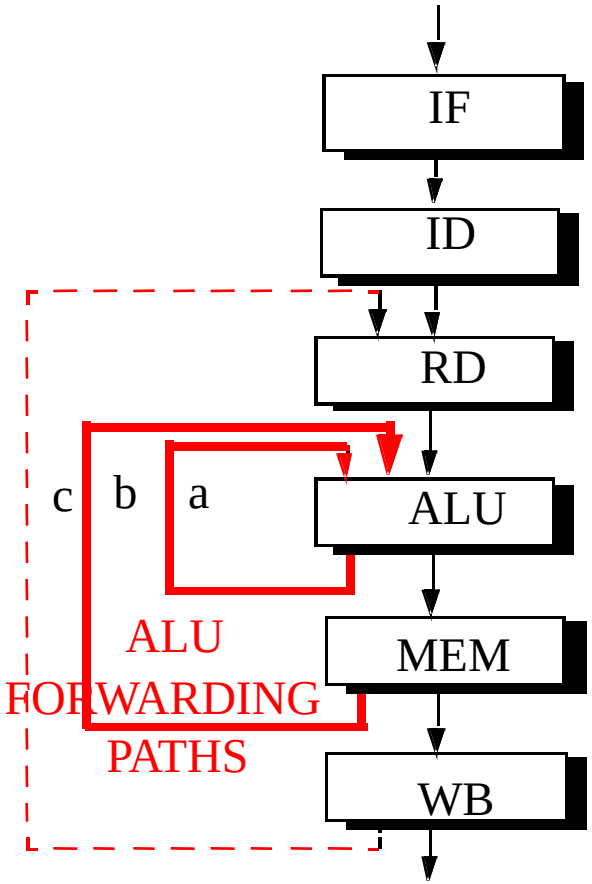
Critical Forwarding Paths

Pipe Stage	ALU Inst.	Load inst.	Store inst.	Branch inst.
1. IF	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4	I-cache PC<PC+4
2. ID	decode	decode	decode	decode
3. RD	read reg.	read reg.	read reg.	read reg.
4. ALU	◆ ALU op. ◆	◆ addr. gen.	◆ addr. gen.	◆ addr. gen. cond. gen.
5. MEM	-----	◆ read mem.	◆ write mem.	PC<-br. addr.
6. WB	write reg.	write reg.	-----	-----

Penalties with Forwarding Paths

Leading $Inst_i$ (producer)	ALU	Load	Branch
Trailing $Inst_j$ (consumer)	ALU, L/S, Br.	ALU, L/S, Br.	ALU, L/S, Br.
Hazard register	Int. Reg. (Ri)	Int. Reg. (Ri)	PC
Value Produced stage (i)	ALU (stage 4)	MEM (stage 5)	MEM (stage 5)
Value Consumed stage (j)	ALU (stage 4)	ALU (stage 4)	IF (stage 1)
Forward from outputs of:	ALU,MEM,WB	MEM,WB	MEM
Forward to input of:	ALU	ALU	IF
RAW distance or penalty:	0 cycles	1 cycles	4 cycles

Implementation of Pipeline Interlock: ALU



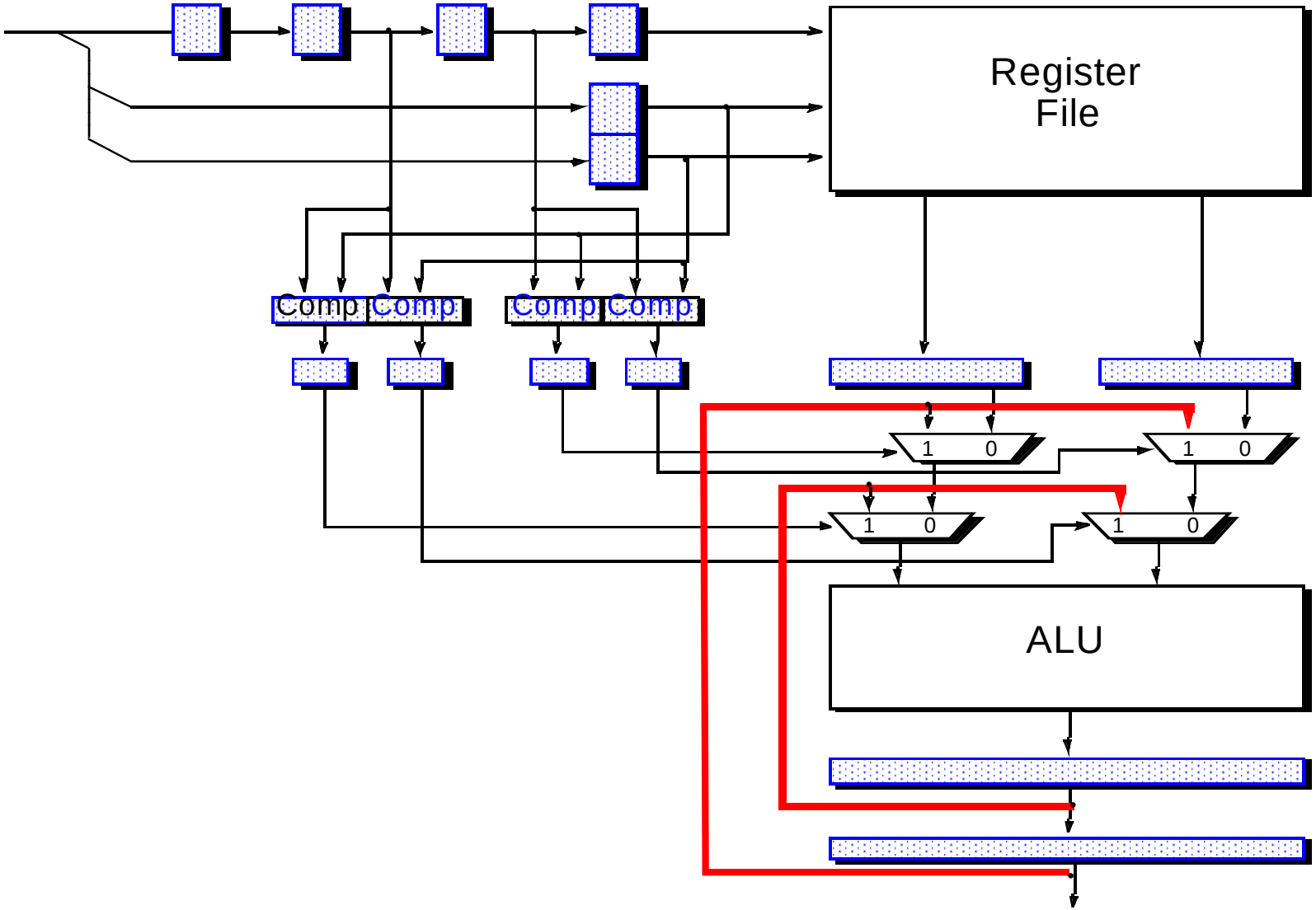
dist=1	dist=2	dist=3
i+1: _ ← R_X	i+2: _ ←R_X	i+3: _ ←R_X
i: R_X ← _	i+1: R_y ← _	i+2: R_z ← _
	i: R_X ← _	i+1: R_Y ← _
		i: R_X ← _

(i → i+1) Forwarding via Path a

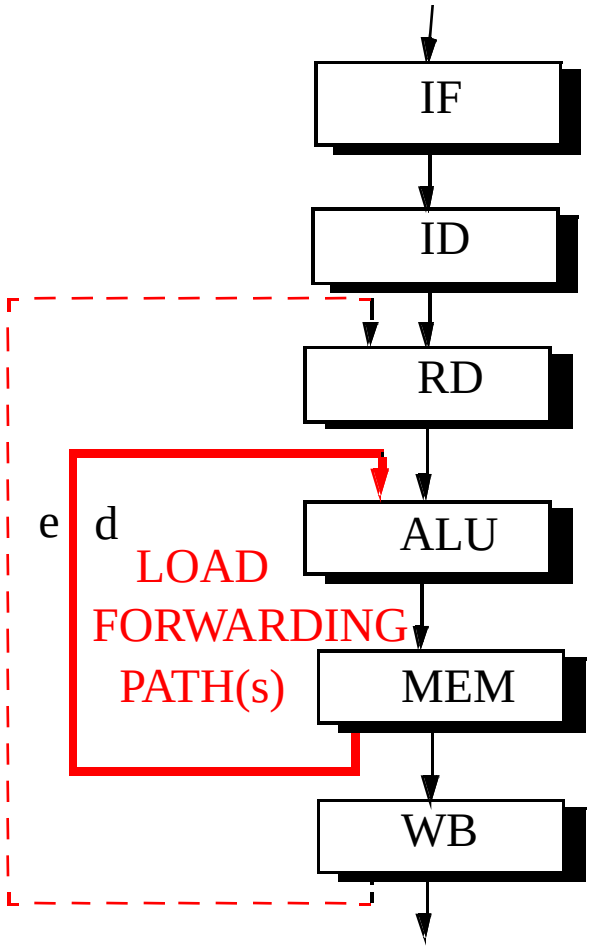
(i → i+2) Forwarding via Path b

(i → i+3) i writes R1 before i+3 reads R1

ALU Forwarding Paths



Forwarding Path(s) for Load Instructions:



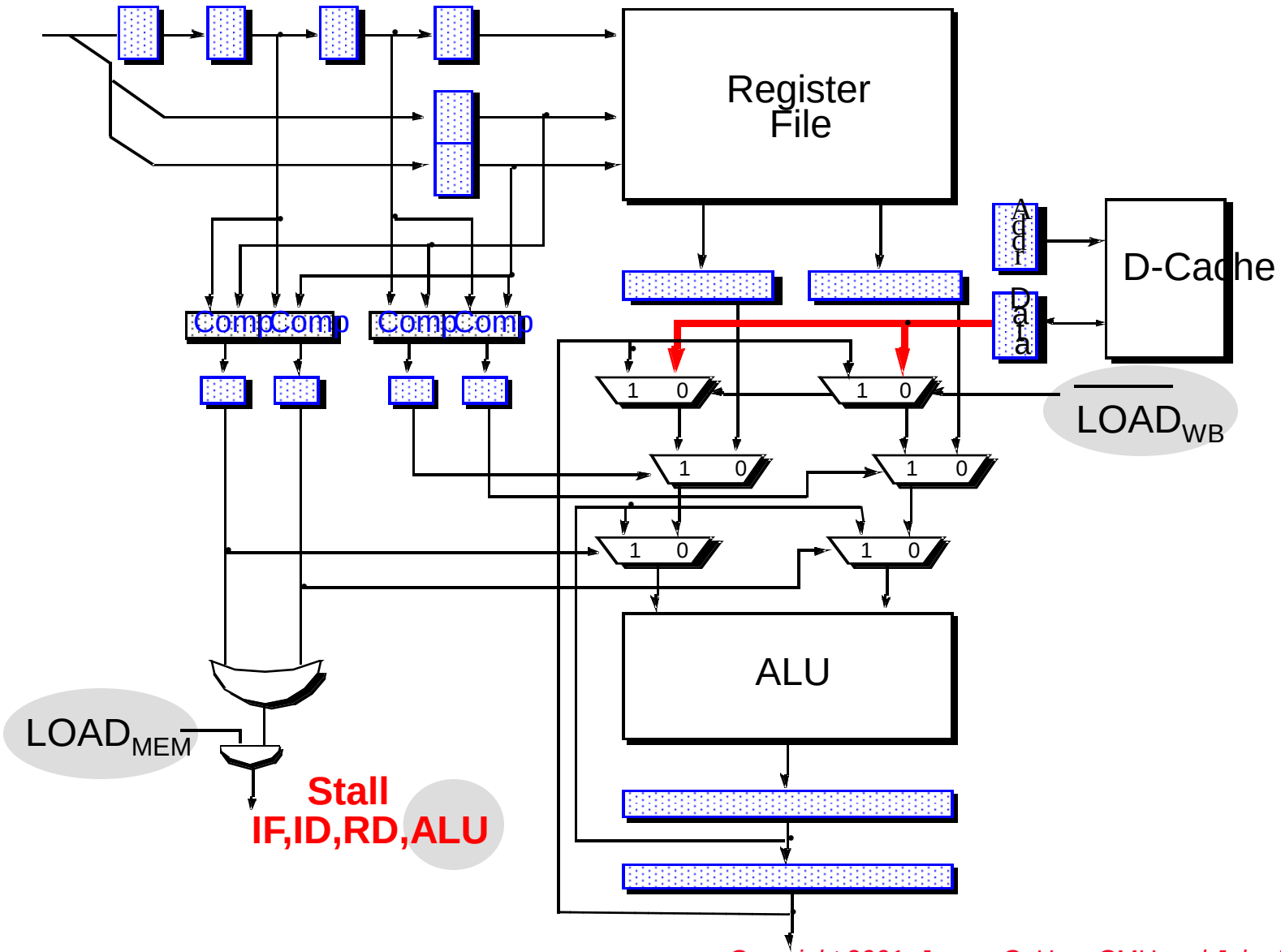
dist=1	dist=2	dist=3
i+1: _ ← R_X	i+2: _ ←R_X	i+3: _ ←R_X
i: R_X ← mem[ ]	i+1: R_y ← _	i+2: R_z ← _
	i: R_X ← mem[ ]	i+1: R_Y ← _
		i: R_X ← mem[ ]

(i → i+1)
Stall i+1

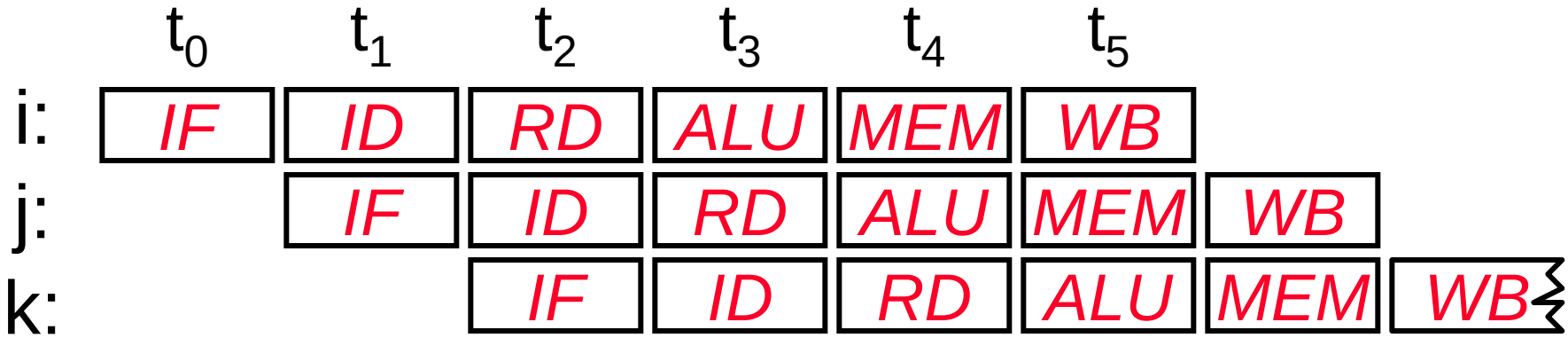
(i → i+1)
Forwarding via Path d

(i → i+2)
i writes R1 before i+2 reads R1

Load Forwarding Path



Load Delay Slot *(MIPS R2000)*



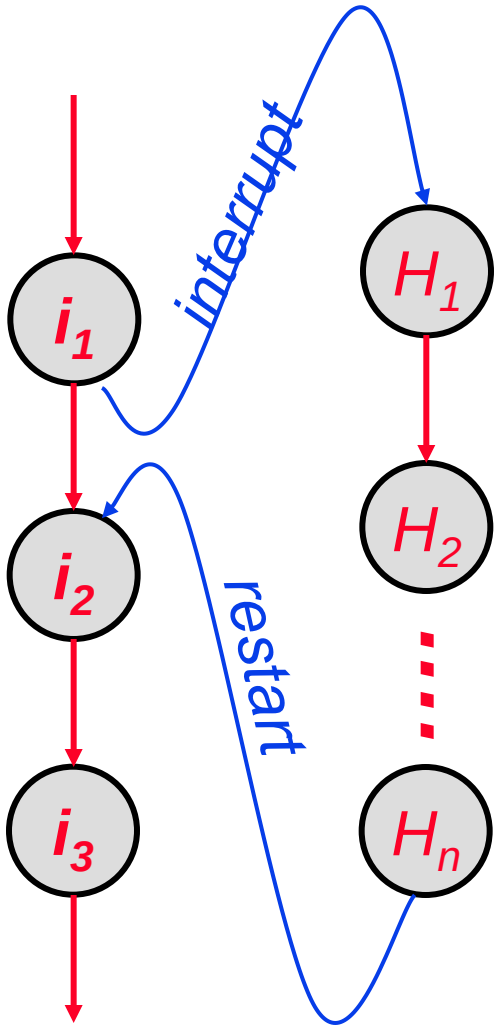

```
h:    Rk ← --
      ......
i:    Rk ← MEM[ - ]
j:    -- ← Rk
k:    -- ← Rk
```

- The effect of a "delayed" Load is not visible to the instructions in its delay slots.

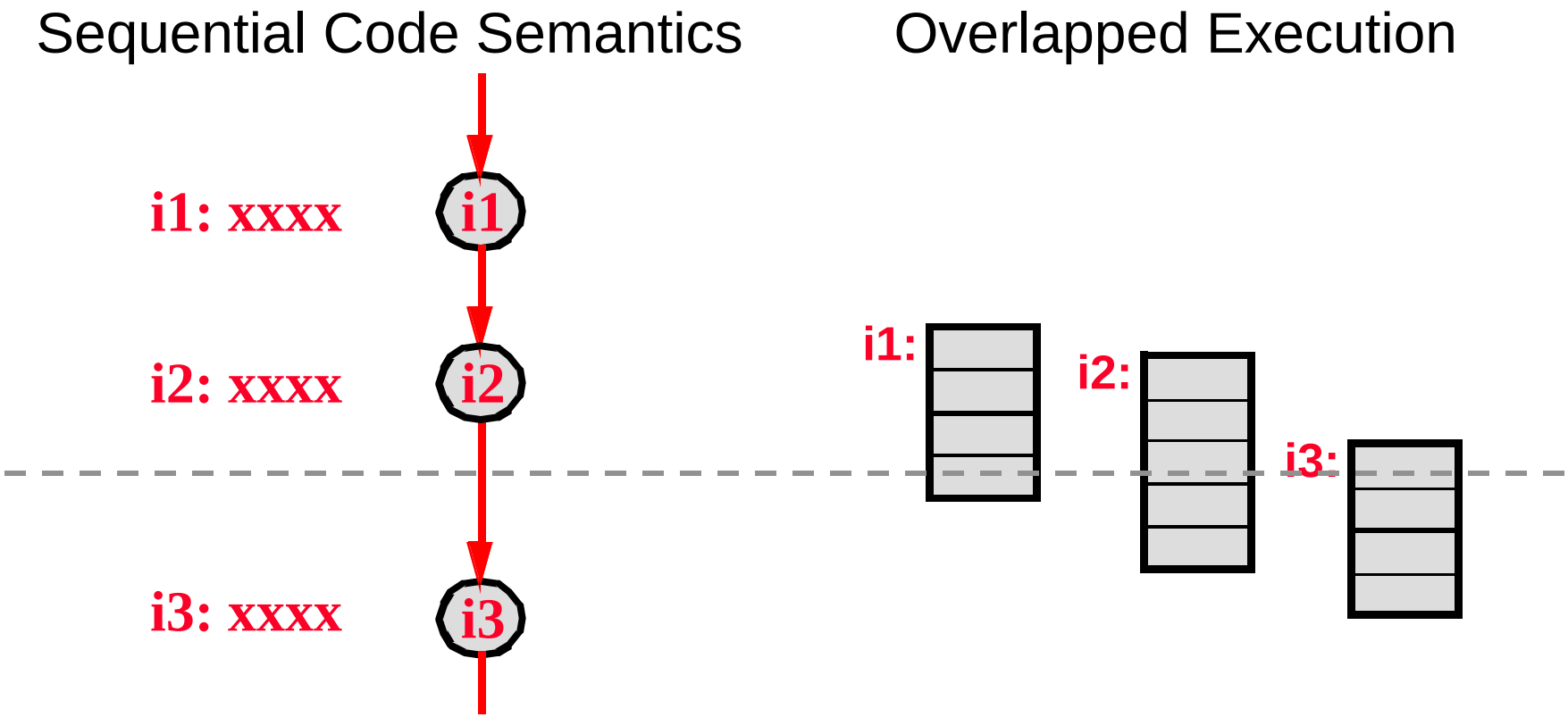
Which (R_k) do we really mean?

Interrupts

- An unexpected transfer of control flow
  - Control is given to a system program and later given back *(restartable)*
  - Transparent to the interrupted program
- Asynchronous Interrupt (e.g. I/O) can be taken at some convenient point
- Synchronous Interrupt (e.g. divide by 0) is associated with a particular instruction



Precise Interrupts



A precise interrupt appears (to the interrupt handler) to take place exactly between two instructions

What to Do on an Interrupt

- Find the point of interrupt in the instruction stream
- Allow instructions prior to the interrupt point to complete
- Save enough state to allow restarting at the same point
  - Program Counter, status registers, etc.

    (registers that are clobbered during transition)
  - Why not general purpose register file??
- Start executing from a pre-specified interrupt handler address
  - Almost always involves a change in “protection mode”
  - Must be careful to not leave any loophole such that a draining user instruction can act like a “privileged” instruction during transition
- “Return-from-Interrupt” Instruction: Jump to the saved PC and also restore clobbered states from saved copies

Stopping and Restarting a Pipeline

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
IF	I_1	I_2	I_3	I_4	I_5	I_6	nop	I_{IH}	I_{IH+1}	I_{IH+2}	I_{IH+3}
ID		I_1	I_2	I_3	I_4	I_5	nop	nop	I_{IH}	I_{IH+1}	I_{IH+2}
RD			I_1	I_2	I_3	I_4	nop	nop	nop	I_{IH}	I_{IH+1}
ALU				I_1	I_2	I_3	nop	nop	nop	nop	I_{IH}
MEM					I_1	I_2	nop	nop	nop	nop	nop
WB						I_1	I_2	nop	nop	nop	nop

Real Pipelined Processor Example: MIPS R2000

Stage name	Phase	Function performed
1. IF	ϕ1	Translate virtual instr. addr. using TLB
	ϕ2	Access I-cache using physical address
2. RD	ϕ1	Return instr. from I-cache, check tags & parity
	ϕ2	Read reg. file; if a branch, generate target addr.
3. ALU	ϕ1	Start ALU op.; if a branch, check br. Condition
	ϕ2	Finish ALU op.; if a load/store, translate virtual addr.
4. MEM	ϕ1	Access D-cache
	ϕ2	Return data from D-cache, check tags & parity
5. WB	ϕ1	Write register file
	ϕ2	---

Intel i486 5-Stage "CISC" Pipeline

Stage name	**Function performed**
1. Instruction Fetch	Fetch instruction from the 32-byte prefetch queue (prefetch unit fills and flushes prefetch queue)
2. Instruction Decode-1	Translate instr. into control signals or microcode addr. Initiate addr. generation and memory access
3. Instruction Decode-2	Access microcode memory Outputs microinstruction to execution unit
4. Execute	Execute ALU and memory accessing operations
5. Register Write-back	Write back results to register

IBM's Experience on Pipelined Processors

[Agerwala and Cocke 1987]

Attributes and Assumptions:

- Memory Bandwidth
  - at least one word/cycle to fetch 1 instruction/cycle from I-cache
  - 40% of instructions are load/store, require access to D-cache

- Code Characteristics (dynamic)
  - loads - 25%
  - stores - 15%
  - ALU/RR - 40%
  - branches - 20% 1/3 unconditional (always taken);
    1/3 conditional taken;
    1/3 conditional not taken

More Statistics and Assumptions

- Cache Performance
  - hit ratio of 100% is assumed in the experiments
  - cache latency: I-cache = i; D-cache = d; default: i=d=1 cycle
- Load and Branch Scheduling
  - loads:
    - 25% cannot be scheduled
    - 75% can be moved back 1 instruction
  - branches:
    - unconditional - 100% schedulable
    - conditional - 50% schedulable

CPI Calculations I

- No cache bypass of reg. file, no scheduling of loads or branches
  - Load Penalty: 2 cycles
  - Branch Penalty: 2 cycles
  - Total CPI: 1 + 0.5 + 0.27 = 1.77 CPI
- Bypass, no scheduling of loads or branches
  - Load Penalty: 1 cycle
  - Total CPI: 1 + 0.25 + 0.27 = 1.52 CPI

CPI Calculations II

- Bypass, scheduling of loads and branches
  - Load Penalty:

    75% can be moved back 1 => no penalty

    remaining 25% => 1 cycle penalty

  - Branch Penalty:

    1/3 Uncond. 100% schedulable => 1 cycle

    1/3 Cond. Not Taken, if biased for NT => no penalty

    1/3 Cond. Taken

    50% schedulable => 1 cycle

    50% unschedulable => 2 cycles

  - Total CPI: 1 + 0.063 + 0.167 = 1.23 CPI

CPI Calculations III

- Parallel target address generation
  - 90% of branches can be coded as PC relative
    i.e. target address can be computed without register access
  - A separate adder can compute (PC+offset) in the decode stage
  - Branch Penalty:

PC-relative addressing	**Schedulable**	**Branch penalty**
YES (90%)	YES (50%)	0 cycle
YES (90%)	NO (50%)	1 cycle
NO (10%)	YES (50%)	1 cycle
NO (10%)	NO (50%)	2 cycles

  - Conditional: Unconditional:
  - Total CPI: 1 + 0.063 + 0.087 = 1.15 CPI = 0.87 IPC

Pipelined Depth

$$\text{Processor Performance} = \frac{\text{Time}}{\text{Program}}$$

$$= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}}$$

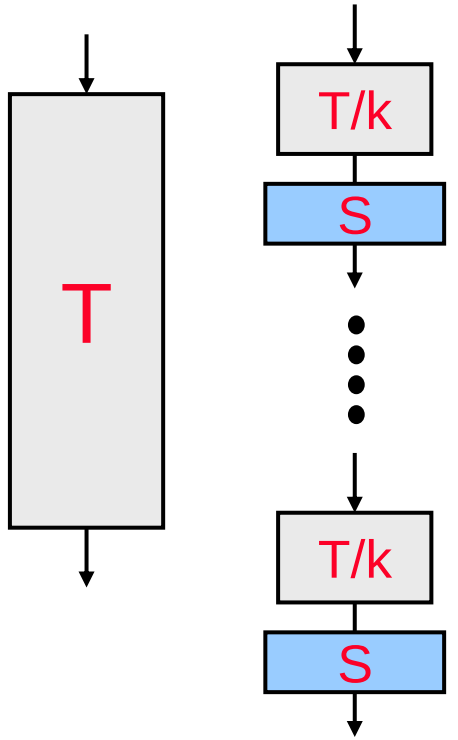
(code size) (CPI) (cycle time)

?

(T/ k +S)

unpipelined

k-stage pipelined



Limitations of Scalar Pipelines

- Upper Bound on Scalar Pipeline Throughtput

  Limited by IPC = 1

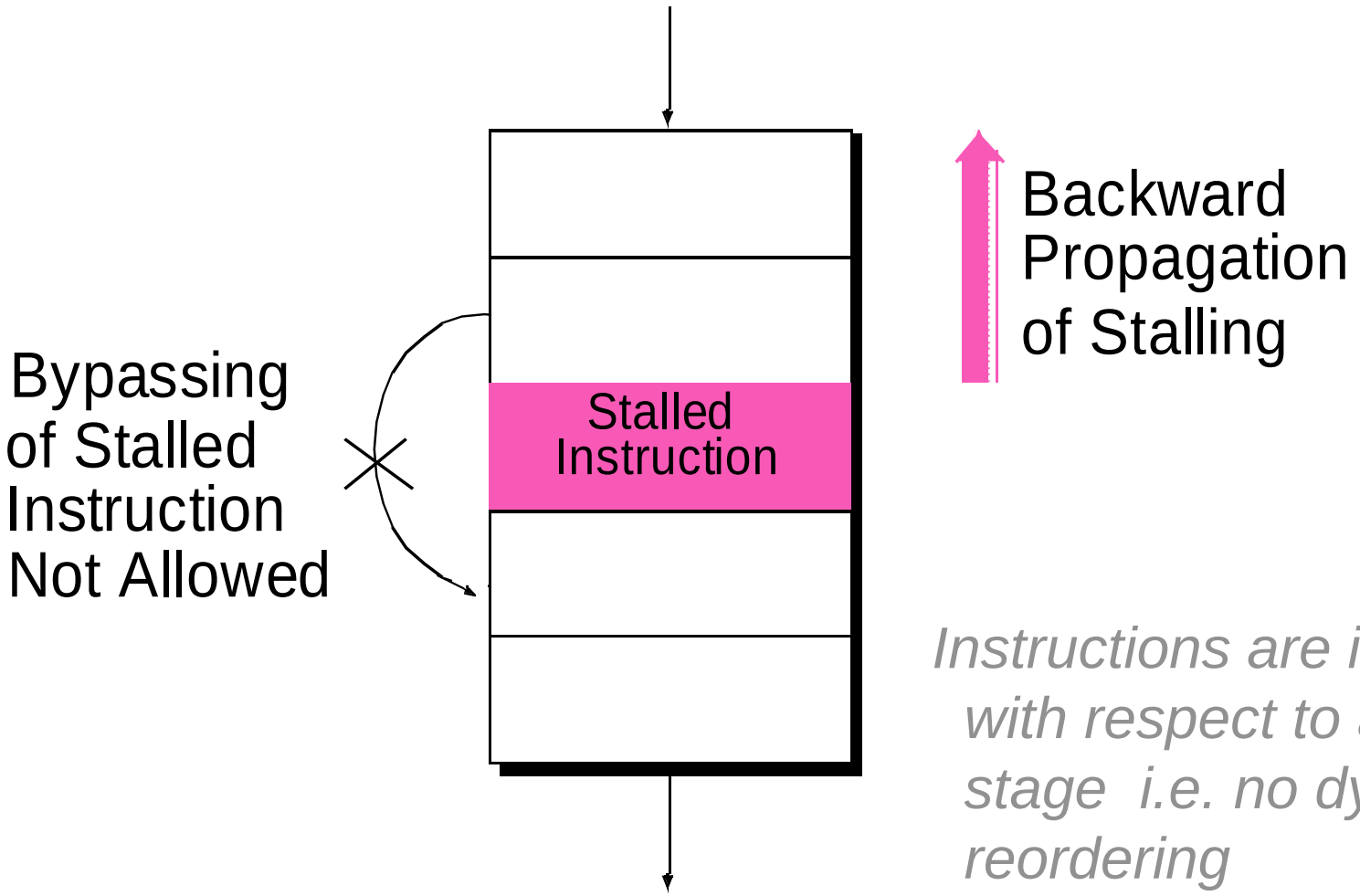
- Inefficient Unification Into Single Pipeline

  Long latency for each instruction

- Performance Lost Due to Rigid Pipeline

  Unnecessary stalls

Stalls in an Inorder Scalar Pipeline



Instructions are in order with respect to any one stage i.e. no dynamic reordering