

17 Introduction to Pipelining

CSC 230

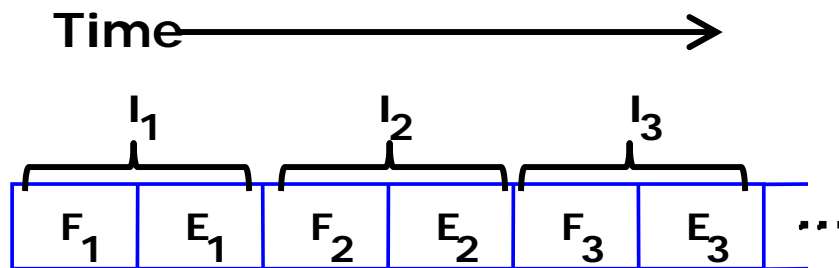
Department of Computer Science
University of Victoria

M&H: 6.7

Stallings: 14.4; 14.6, 15.5

Increasing performance

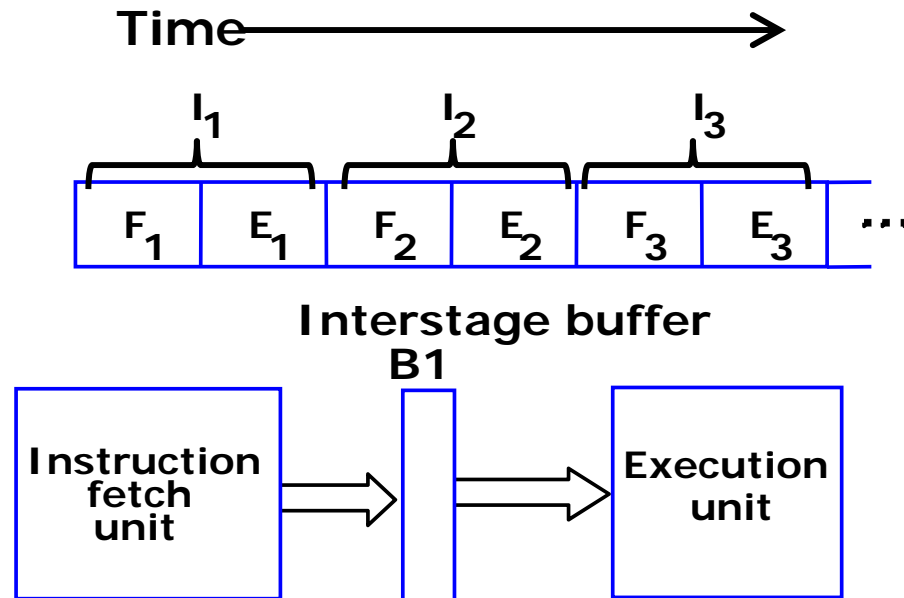
1. break the work in small units
2. “push” it through a pipeline of “workers”
3. each “worker” performs a function on one piece of work



(a) Sequential execution

F = Fetch cycle
E = Execute cycle

Increasing performance → Pipelining



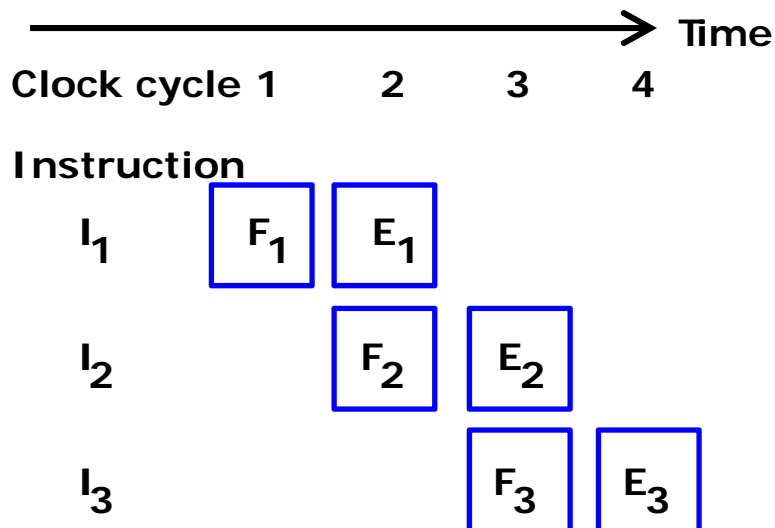
(a) Sequential execution

F = Fetch cycle

E = Execute cycle

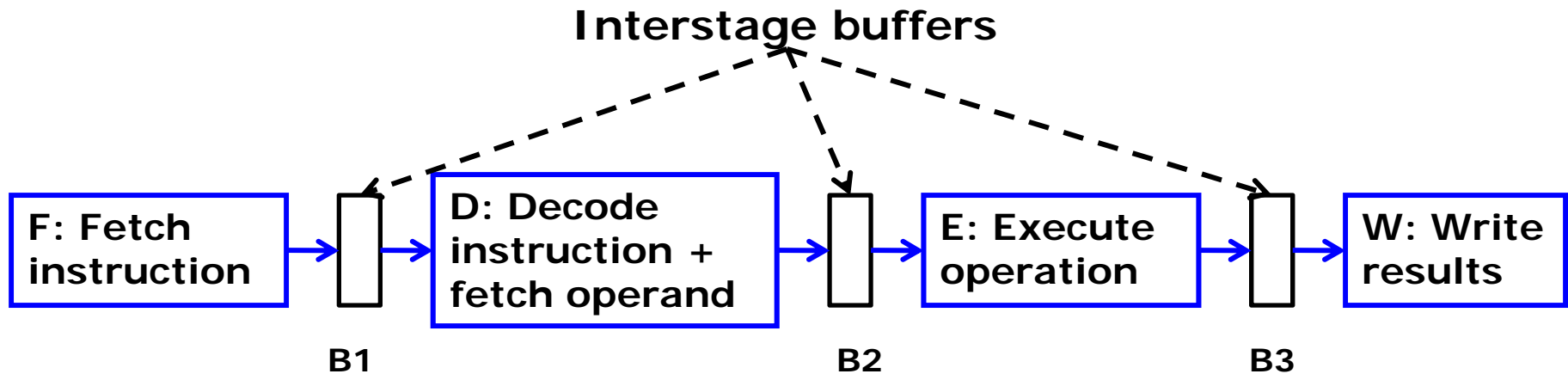
(b) Hardware organization

→ split control unit



(c) Pipelined execution

Hardware Organization to support the pipeline

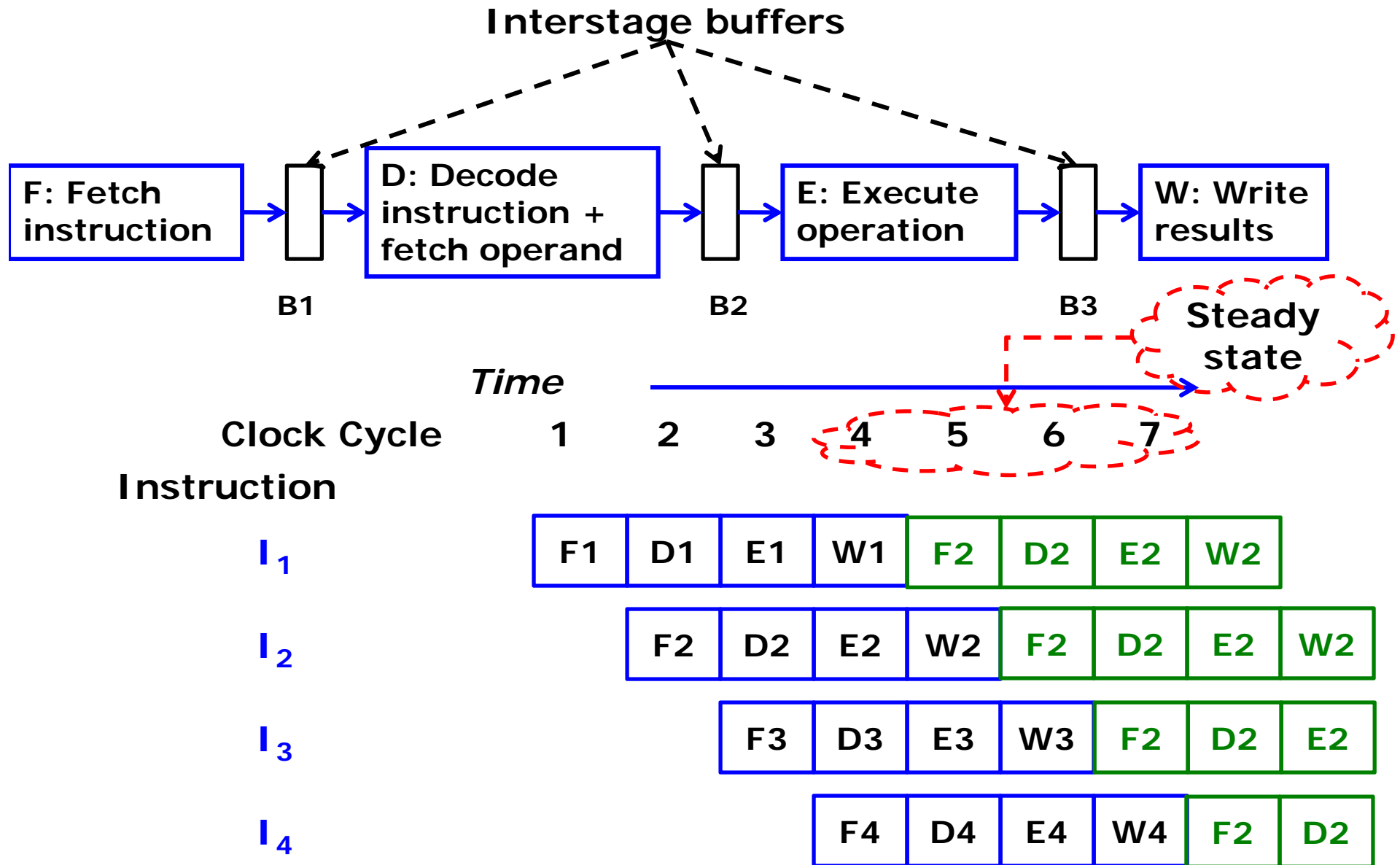


- F Fetch: read instruction from memory
- D Decode: decode instruction and fetch source operands
- E Execute: performed specified operation
- W Write: store result in destination location

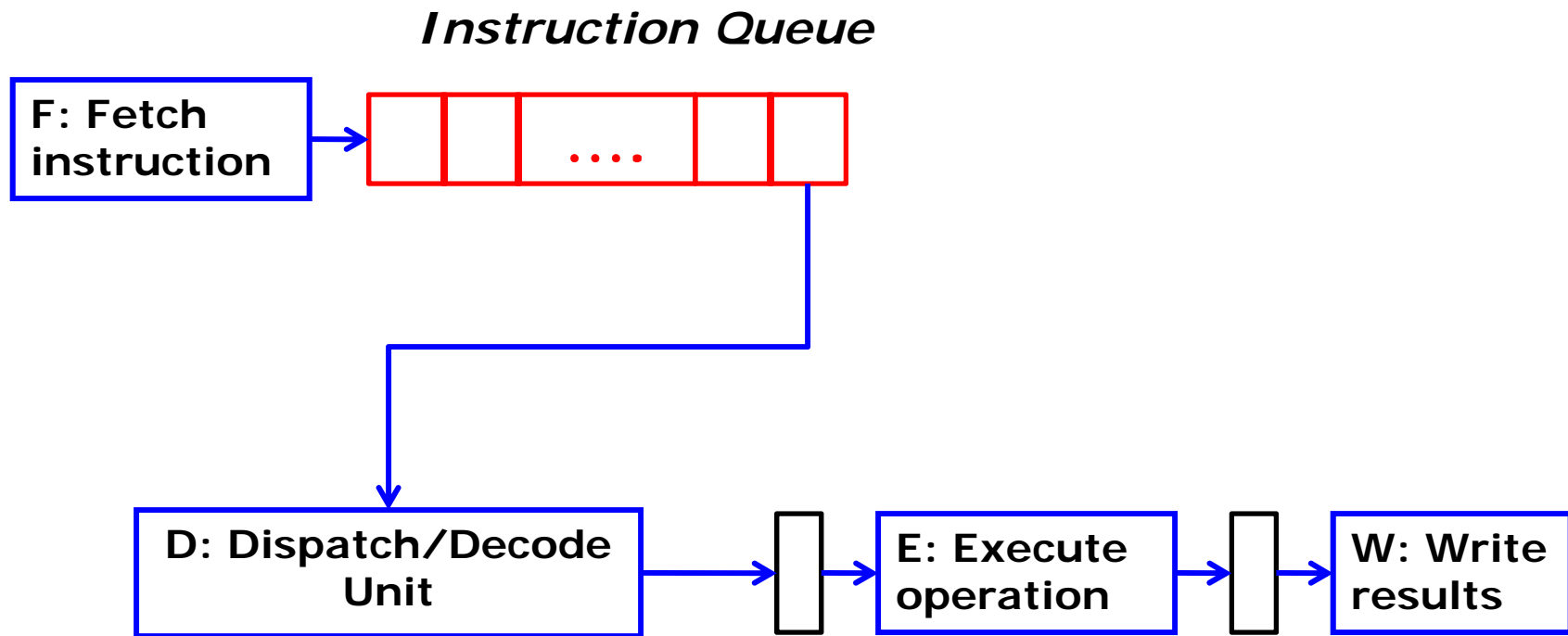
Why 4 here?

it reflects the 4 sub-phases of a clock cycle

Instruction execution divided into four steps

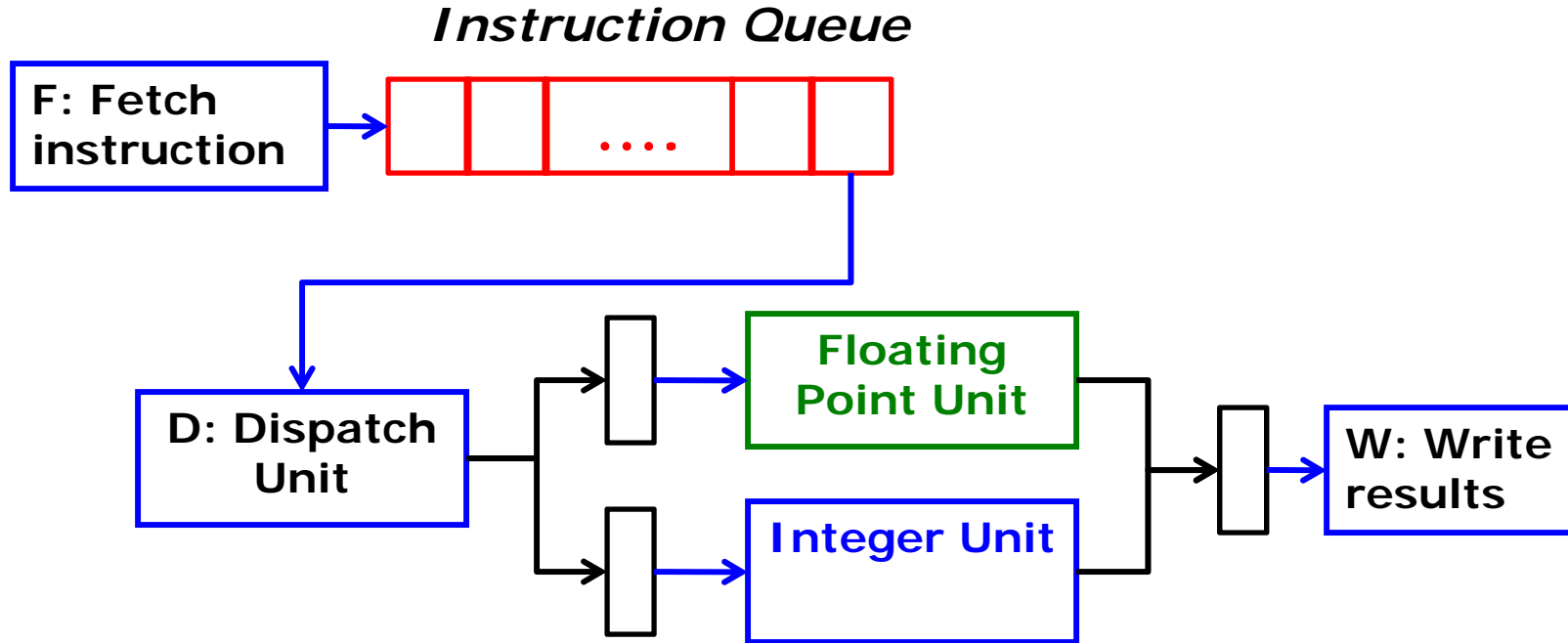


Use of an instruction queue



Superscalar Operation: > 1 execution unit

Example: A processor with two execution units



- ❑ Maximum throughput of a pipelined processor is 1 instruction per clock cycle
- ❑ Add multiple processing units for several instructions in parallel in each stage → **superscalar processor**

Note: this is not the same as a dual core processor!

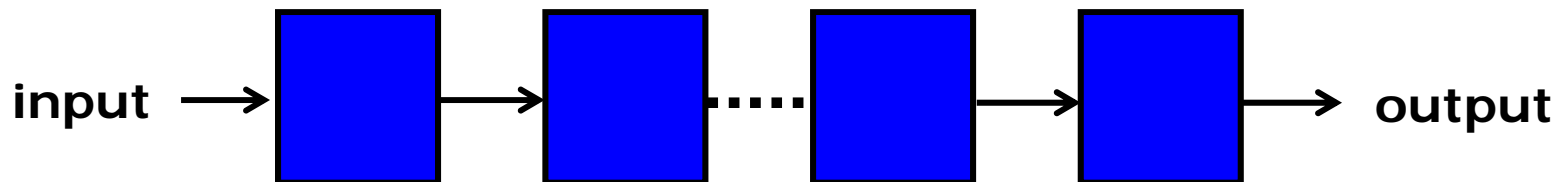
Performance issues

- ❑ Need to execute N processes
- ❑ Each process requires m cycles

The N processes require in general:

$$T_{\text{serial}} = N \times m \text{ cycles}$$

Build a pipeline with m stages



- The first item will emerge after m cycles
- The second item will emerge after $m+1$ cycles
- It will take another $N-1$ cycles for the last item to emerge

for a total of $T_{\text{pipeline}} = m + N - 1$ cycles

Consider the possible speedup

$$speedup = \frac{T_{serial}}{T_{pipeline}} = \frac{m \times N}{m + N - 1}$$

If $N \gg m$, then

$$speedup = \frac{m \times N}{m + N - 1} \approx m$$

Consider $m=4$ stages versus $N = \text{millions of processing steps}$

Speedup

→ Thus an m stage pipeline can potentially/theoretically speed-up the processing by a factor of m

Provided that there are no disruptions → no branching!

→ Interrupting a pipeline is expensive

The state must be restored, and the cause of the interruption dealt with, before it is allowed to continue

A lot of ingenuity and hardware are devoted in modern processors exactly to ensure that such interruptions do not happen

UltraSPARC II – 9 stage pipeline

Pentium Pro – 12 stage pipeline

Pentium 4 – 20 stage pipeline (2 pipeline stages in each clock cycle)

ARM: 3 stage pipeline (up to 13 in optimized units)

General performance definitions from earlier

$$T = \frac{N \times S}{R}$$

T = performance (time)

N = actual number of executed instructions

S = number of 1-clock steps needed for 1 instruction

R = clock rate (cycles per second)

$$P_s = \frac{R}{S}$$

Throughput:

that is, the number of instructions
executed per second

Questions now:

- 1. How much can a pipeline increase P_s ?*
- 2. How many stages is optimal?*

Another view of performance (see also 14.4)

n = number of instruction executed

k = number of stages in the pipeline

P_i = time delay in i th stage

$T_{k,n}$ = time for a pipeline with k stages to execute n instructions

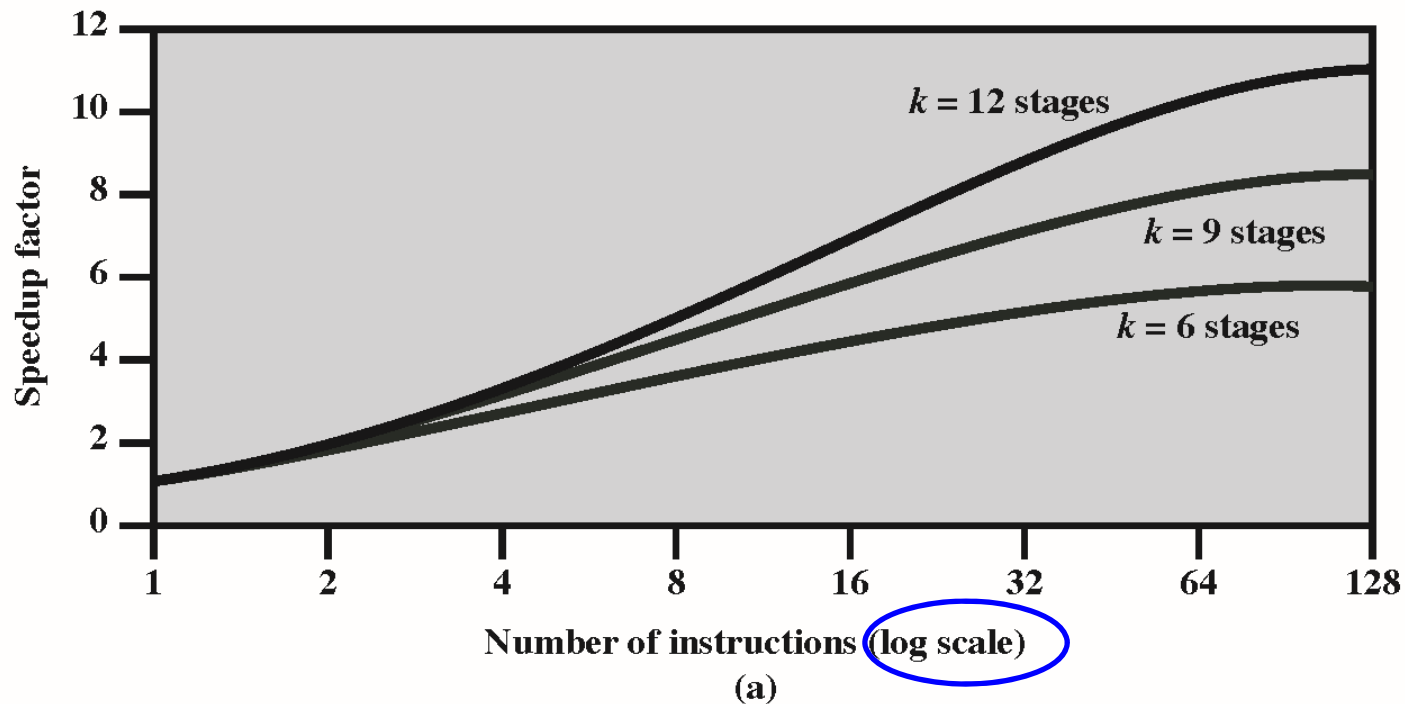
$$T_{k,n} = [k + (n - 1)] P_i$$

$T_{1,n}$ = time without a pipeline (1 stage) to execute n instructions

→ Calculating potential speedup:

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{n P_i}{[k + (n - 1)] P_i} = \frac{n}{k + (n - 1)}$$

Speedup as a function of the number of executed instructions (no branching)



As $(n \rightarrow \infty)$, speedup approaches k

Single cycle versus pipelined performance: example (not ARM)

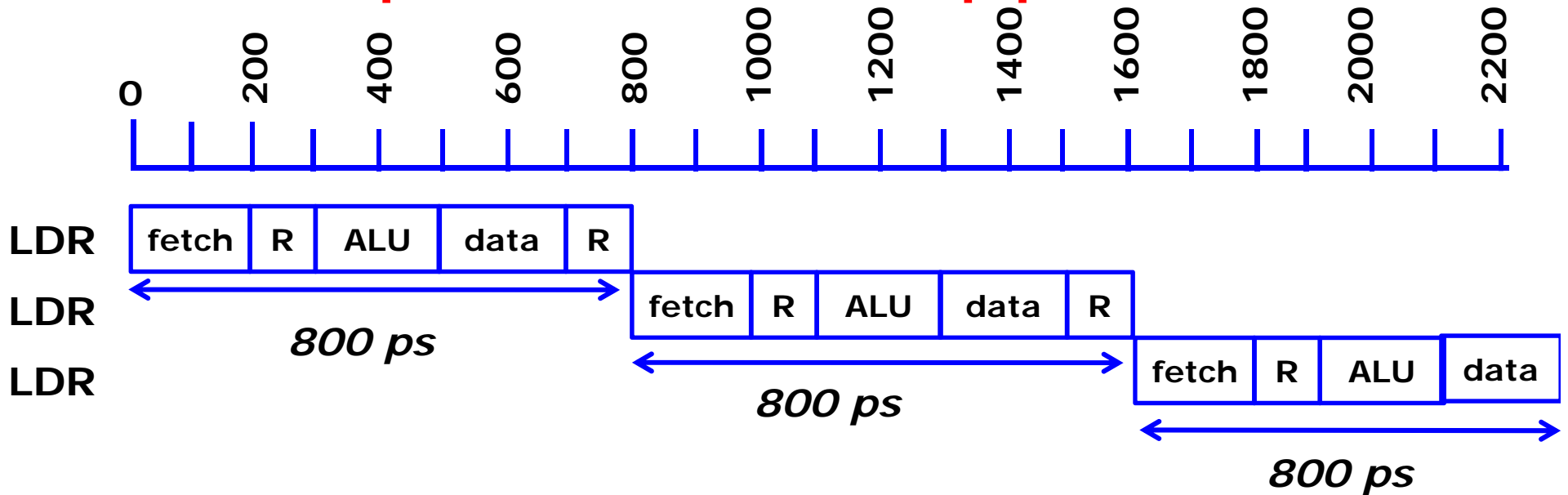
Instruction class	Instruction Fetch	Register access	ALU Operation	Data access	Register access	Total Time
Load word (LDR)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (STR)	200 ps	100 ps	200 ps	200 ps		700 ps
Arithmetic (ADD, SUB, AND, ORR)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (Bcc)	200 ps	100 ps	200 ps			500 ps

Total time for each instruction calculated from the time of each component

Goal: every instruction must take 1 clock cycle

- The clock cycle must accommodate the slowest instruction
- Here it must be no faster than 800 ps

Pipelined versus non-pipelined



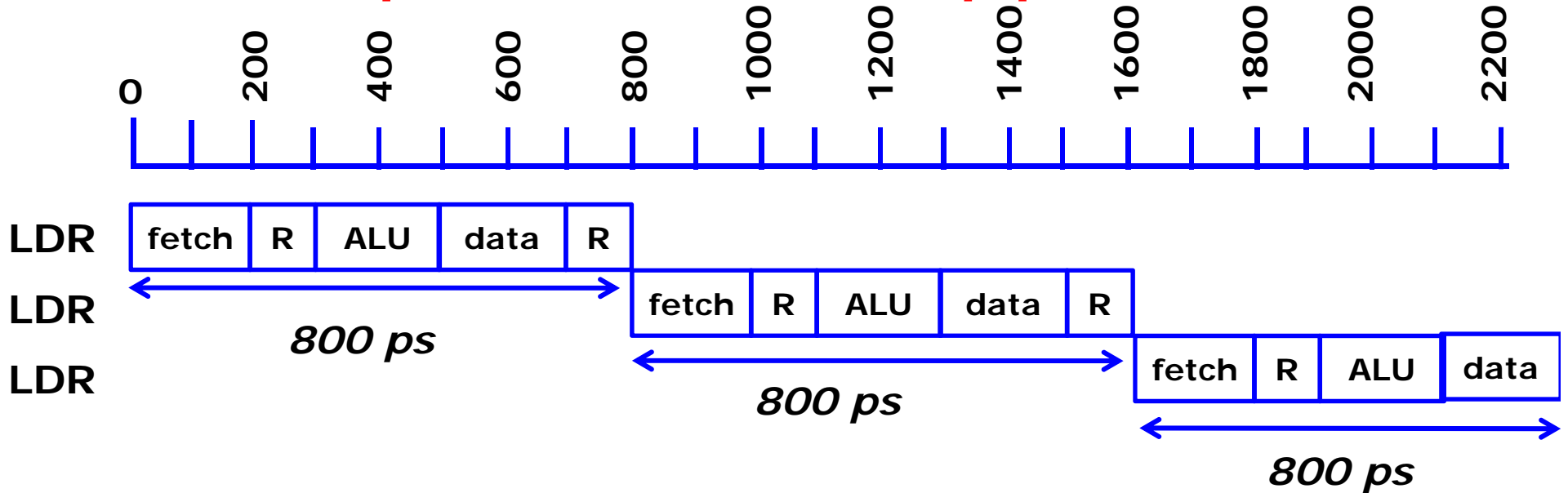
Total Time for 3 LDR = 2400 ps

Clock cycle must be 800 ps

Average time between instruction = 800 ps

What is the clock rate?

Pipelined versus non-pipelined



Total Time for 3 LDR = 2400 ps

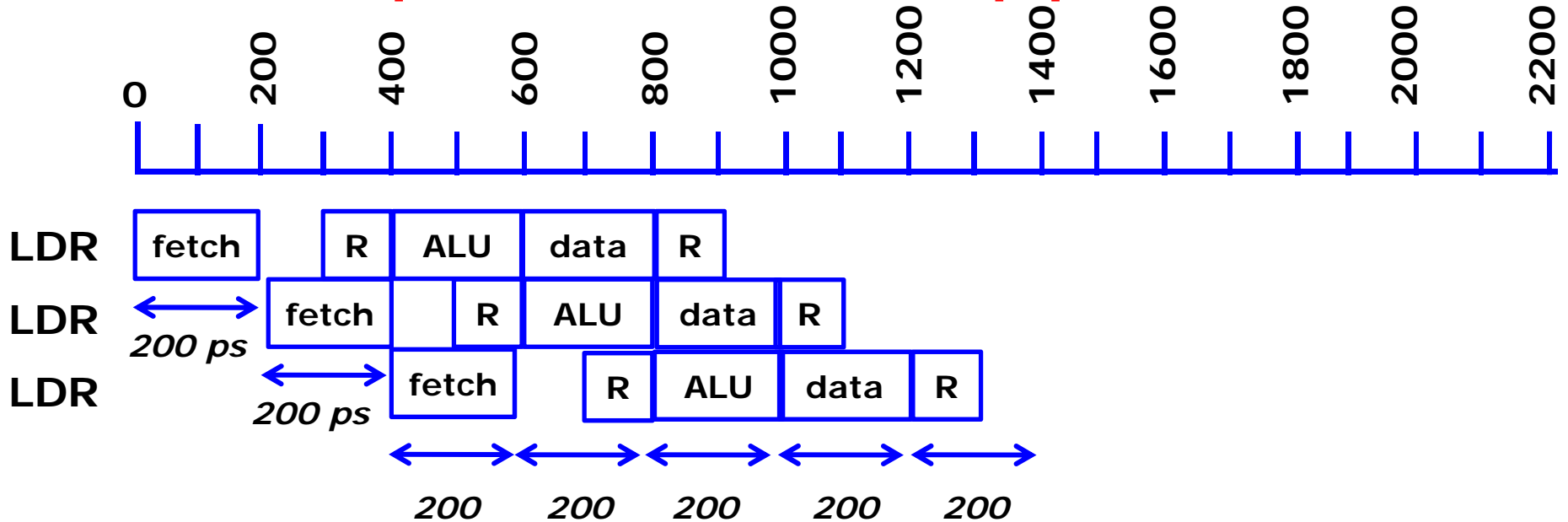
Clock cycle must be 800 ps

Average time between instruction = 800 ps

What is the clock rate?

$R = 1/P \rightarrow 1/800 \text{ ps} = 0.00125 \rightarrow \text{multiply by } 10^{12} \text{ for seconds}$
 $\rightarrow 1,250,000,000 \text{ Hz} = 1.25 \text{ GHz}$

Pipelined versus non-pipelined



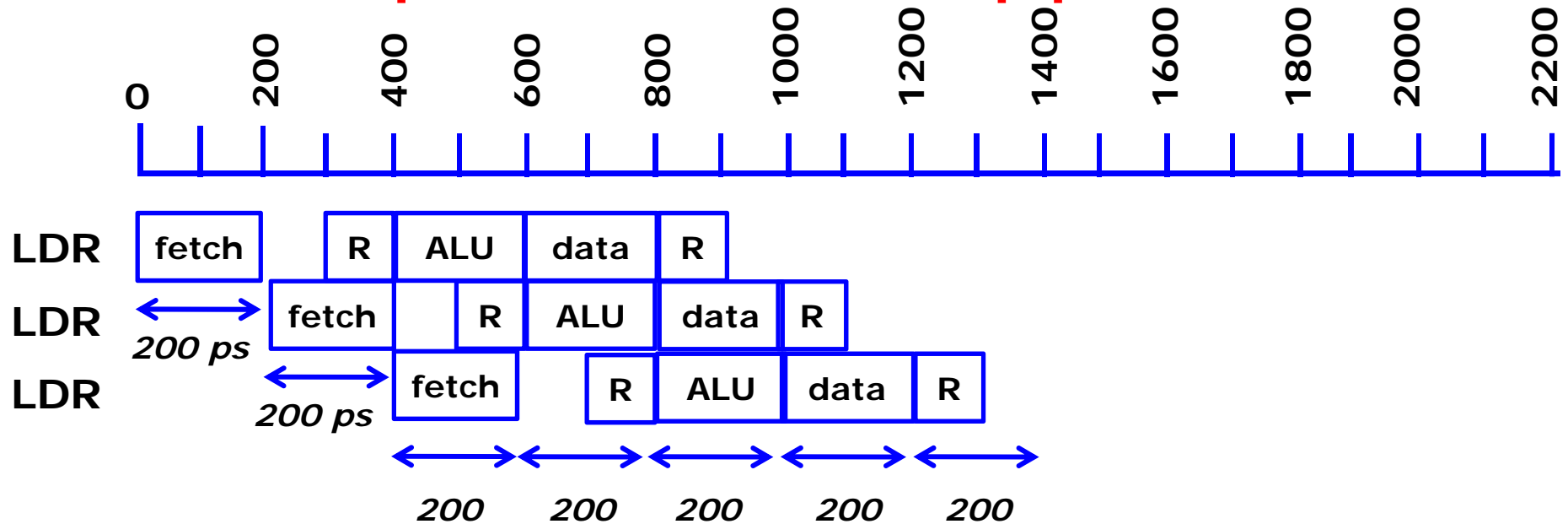
Total Time for 3 LDR = 1400 ps

Clock cycle must be 200 ps

Average time between instruction = 200 ps

What is the clock rate?

Pipelined versus non-pipelined



Total Time for 3 LDR = 1400 ps

Clock cycle must be 200 ps

Average time between instruction = 200 ps

What is the clock rate?

$R = 1/P \rightarrow 1/200 \text{ ps} = 0.005 \rightarrow \text{multiply by } 10^{12} \text{ for seconds}$
 $\rightarrow 5,000,000,000 \text{ MHz} = 5 \text{ GHz}$

Speedup in this example?

There are 4 stages in pipeline

→ Expect a speedup of ~ 4

Actual Speedup: $2400/1400 = 1.71$

→ Not 4!

→ too few instructions for evaluation?

Try adding 1,000,000 more instructions (total of 1,000,003)

- each extra instruction adds an extra 200 ps
thus $1400 + (1,000,000 \times 200) = 200,001,400$ ps

- non-pipelined:
 $2400 + (1,000,000 \times 800) = 800,002,400$

Final speedup:

$$800,002,400 / 200,001,400 = 3.999984$$

Pipelining and performance: the Philosophy

Pipelining increases performance by:

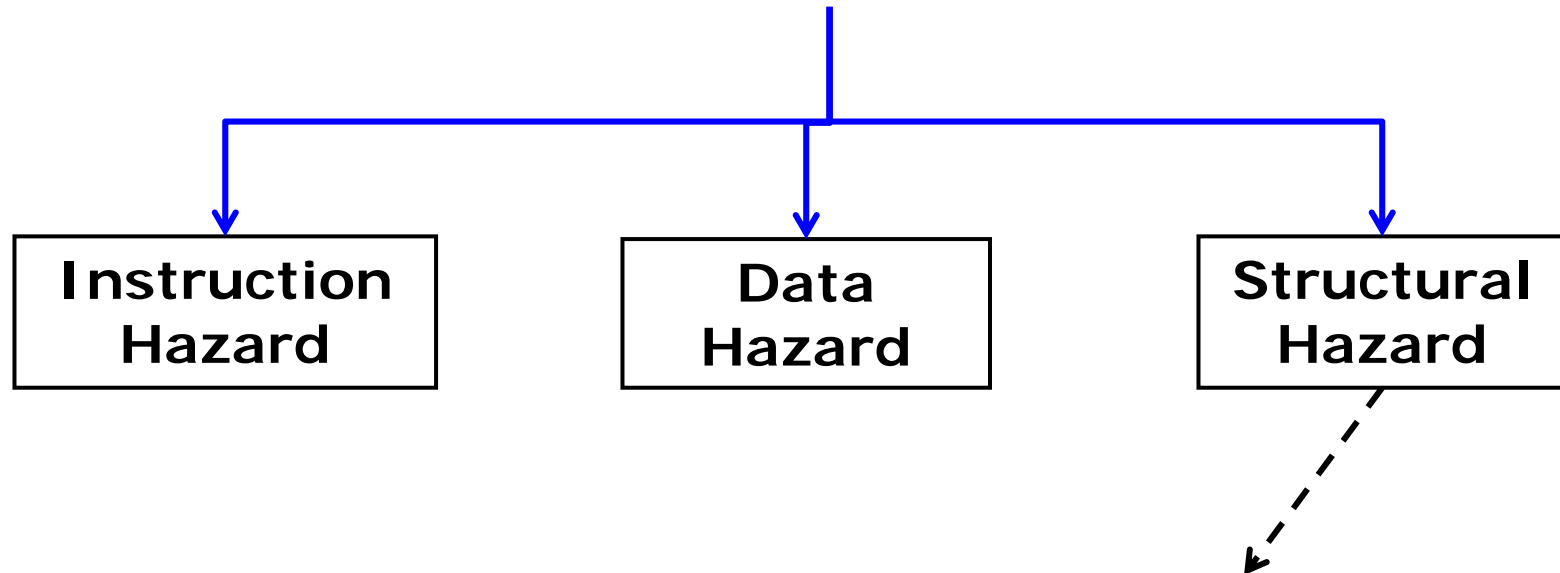
→ increasing instruction throughput

As opposed to decreasing the execution time of each individual instruction

Throughput may be considered the more important metric given the size of application programs (billions of instructions executed)

Performance and Hazards

Hazard = when pipeline STALLS → one stage needs to be stretched over time



when two instructions need access to same hardware resource

e.g. one instruction being fetched from memory while another needs data from memory

Data Hazard

condition when one of the operands is not yet available

usually caused by DATA DEPENDENCIES between instructions

MUL **r2**, r5, r4 here **DESTINATION** operand in MUL is **R2**
SUB r3, **r2**, #1 **R2** is **SOURCE** operand in SUB

(1) Hardware solution: operand forwarding by control
 hardware with extra path/buffer
 (better)

(1) Software solution: delay introduced by compiler
 e.g. *reorder instructions*

e.g. *add delay*

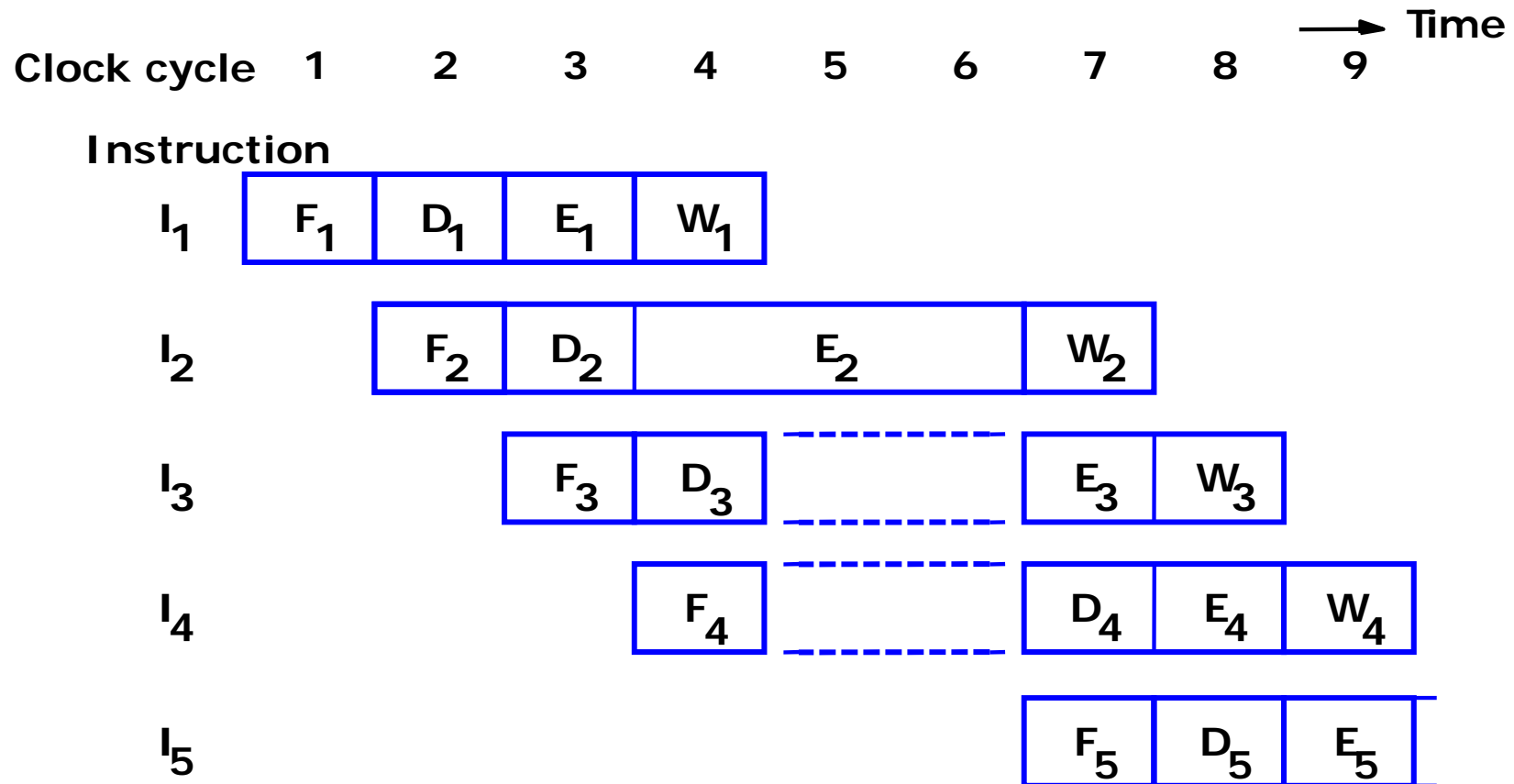
MUL **r2**, r5, r4

NOP *(or other)*

SUB r3, **r2**, #1

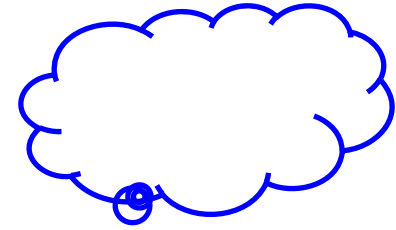
Data Hazard

condition when one of the operands is not yet available



Effect of an execution operation
taking more than one clock cycle

Pipeline stalls = Bubbles



A = B + E;
C = B + F;

```
LDR R1,=B
LDR R1,[R1] @R1=B
LDR R2,=E
LDR R2,[R2] @R2=E
ADD R3,R1,R2 @R3=A
LDR R4,=A
STR R3,[R4] @store A
LDR R5,=F
LDR R5,[R5] @R5=F
ADD R6,R5,R1 @R6=C
LDR R7,=C
STR R6,[R7] @store C
```

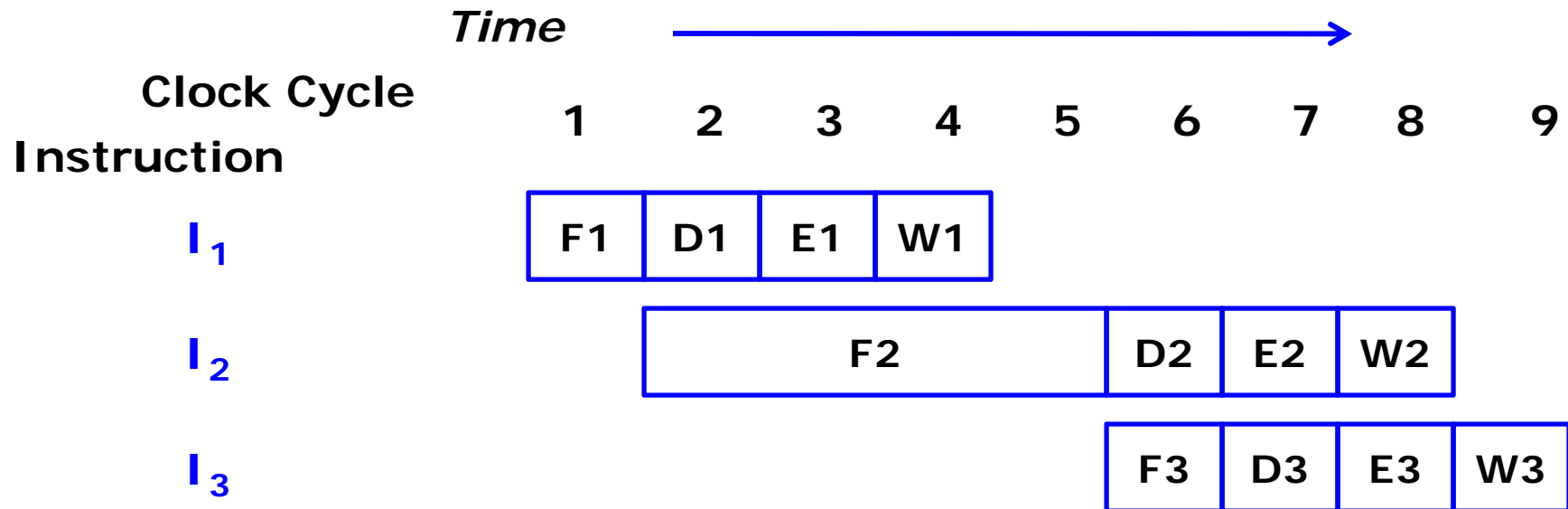
```
LDR R1,=B
LDR R2,=E
LDR R5,=F
LDR R1,[R1] @R1=B
LDR R2,[R2] @R2=E
LDR R5,[R5] @R5=F
LDR R4,=A
LDR R7,=C
ADD R3,R1,R2 @R3=A
ADD R6,R5,R1 @R6=C
STR R3,[R4] @store A
STR R6,[R7] @store C
```

Re-ordering instructions

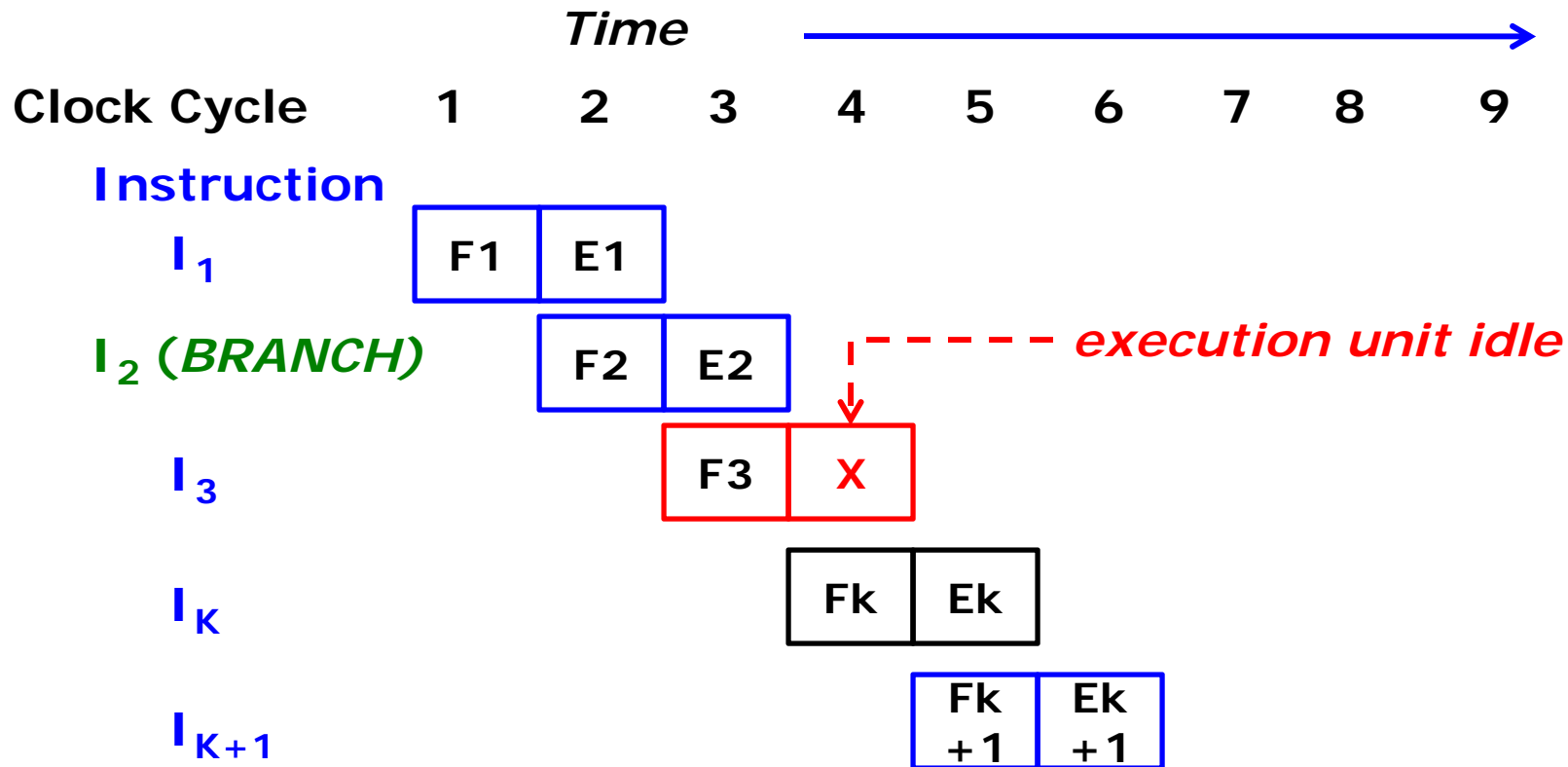
Instruction Hazard

condition when one of the instructions is not yet available

e.g. cache miss or branch instr.



Branch instructions: the problem



Idle cycles caused by a branch instructions = branch penalty:

- clock 3 \rightarrow fetch for I_3
- clock 4 \rightarrow discard I_3 , fetch I_k \rightarrow execution unit idle

Two examples: impact of branches on performance

Example:

- Pipeline with 5 stages
 - If there is a branch instruction \rightarrow *penalty $b = 4$ cycles*
 - $P_b = 0.25 \rightarrow$ probability that instruction is branch
 - $P_t = 0.5 \rightarrow$ probability that a branch is taken
- a) Calculate CPI_{avg} = average number of cycles per instruction
b) Calculate execution efficiency

a) Let $CPI_{NoBranch} = 1$ *no branches*

$$CPI_{avg} = (1 - P_b) (CPI_{NoBranch}) +$$

If a branch \rightarrow $P_b [P_t (1 + b) + (1 - P_t) (CPI_{NoBranch})]$

$= 1 + b P_b P_t$

branch taken with penalty *branch not taken*

Example (continued)

$$CPI_{avg} = 1 + b P_b P_t$$

$$\text{Thus: } CPI_{avg} = 1 + (4) (0.25) (0.5) = 1.5$$

➔ Thus it takes 1.5 cycles per instruction on average with this pipeline with the particular constraints

Now calculate the execution efficiency

$$\text{Execution Efficiency} = (CPI_{NoBranch}) / (CPI_{avg}) = 1 / 1.5 = 67\%$$

This means that the processor runs at 67% of potential top efficiency when there are branches

HOWEVER, remember that without a pipeline it would take 5 cycles per instruction (i.e. $CPI_{avg} = 5$)

Based on Example:

Assume that time per cycle is 20 ns

Then $\text{CPI}_{\text{avg}} = 1.5 \times 20\text{ns} = 30 \text{ ns}$

Increasing the number of cycles per instruction (CPI) may improve the execution efficiency of a pipeline.

Now a question from your employer:

- pipeline goes from 5 to 6 depth
- cycle time goes from 20ns to 10 ns

Calculate the new (improved?) execution efficiency

Based on Example:

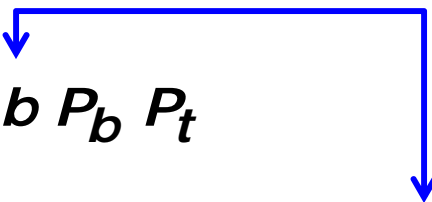
Assume that time per cycle is 20 ns

Then $CPI_{avg} = 1.5 \times 20ns = 30 ns$

Now a question from your employer:

- pipeline goes from 5 to 6 depth
- cycle time goes from 20ns to 10 ns

Calculate the new (improved?) execution efficiency

$$CPI_{avg} = 1 + b P_b P_t$$


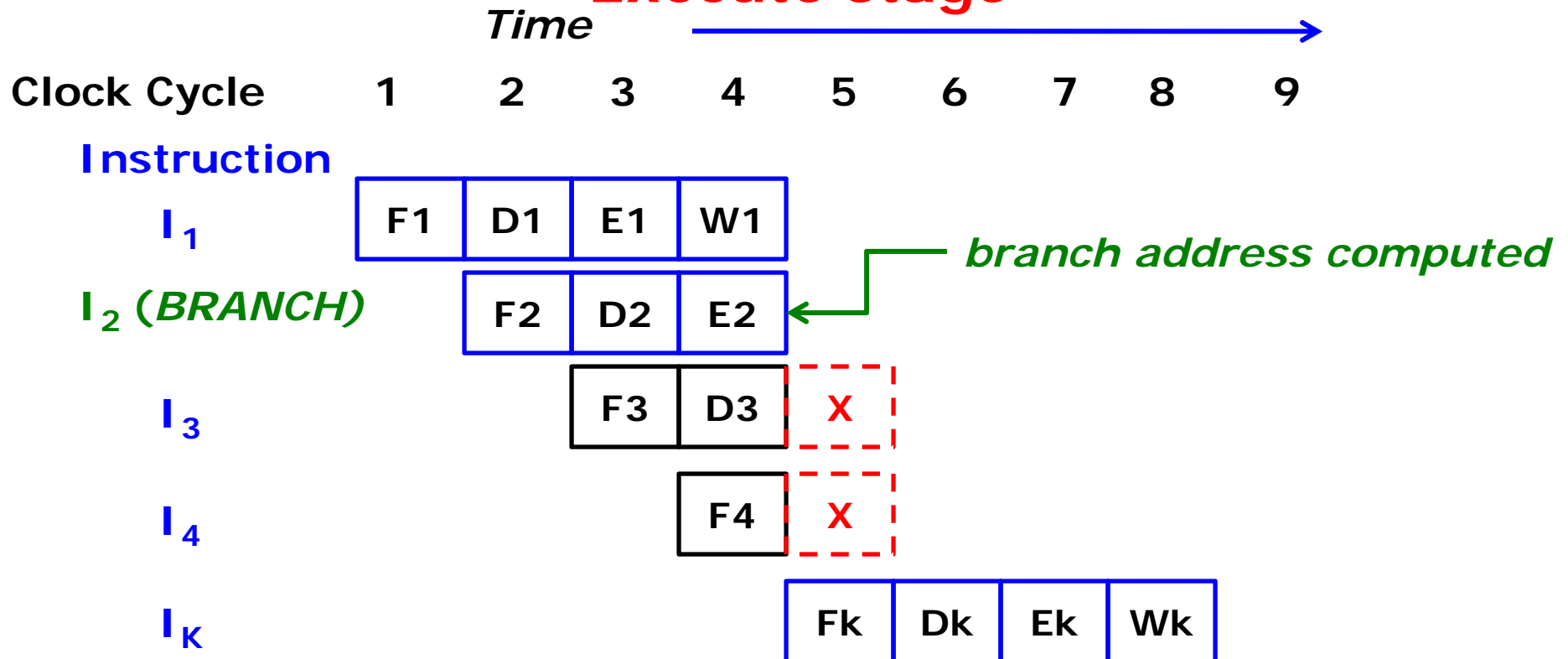
$$\text{Thus: } CPI_{avg} = 1 + (5) (0.25) (0.5) = 1.625$$

At 10ns $\rightarrow CPI_{avg} = 1.625 \times 10ns = 16.25 ns$

$$\text{Execution Efficiency} = 1 / 1.625 = 62\%$$

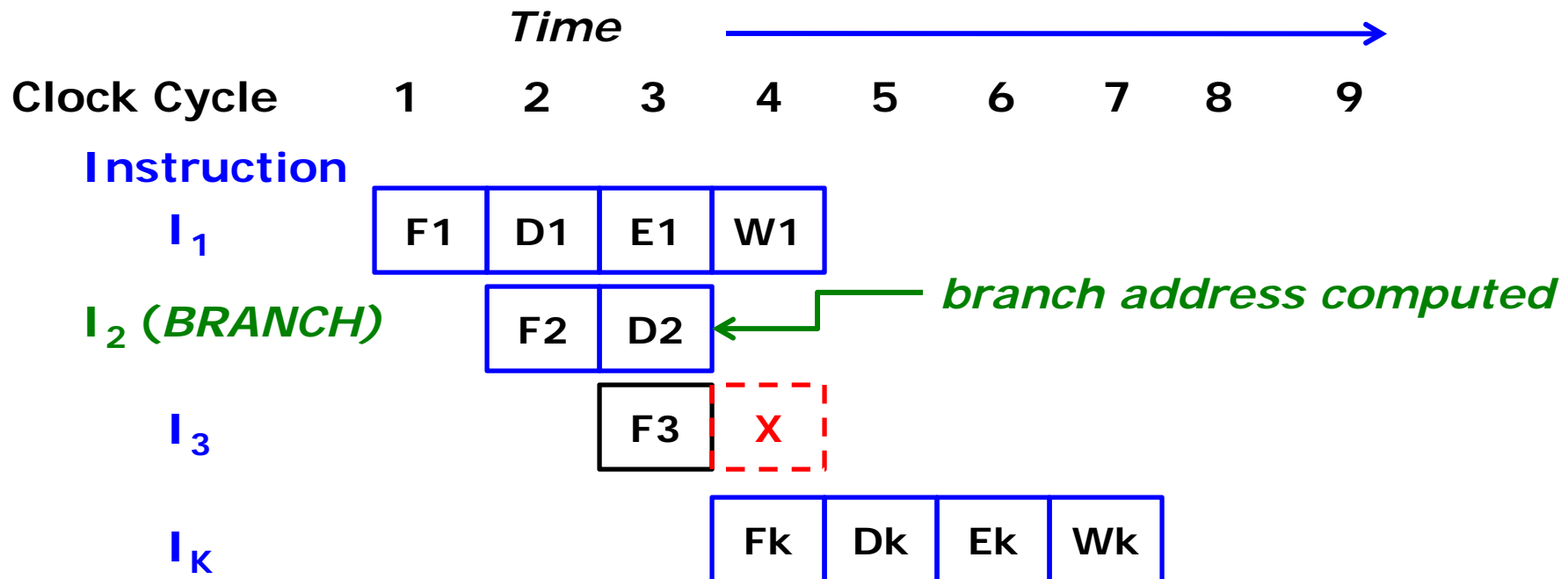
Better?
Worse?

Scenario 1: Branch address computed in Execute stage



- branch address computed at E2
 - must throw away I_3 and I_4
 - I_k fetched in clock cycle 5
- 2 clock cycles penalty

Scenario 2: Branch address computed in Decode stage – additional hardware needed



- branch address computed at D2
- must throw away only I_3
- I_k fetched in clock cycle 4
- ➔ 1 clock cycles penalty

General solutions: Dealing with branches (1)

Multiple Streams: Have two pipelines

- Prefetch each branch into a separate pipeline

- Use appropriate pipeline

- Leads to bus & register contention

- Multiple branches lead to further pipelines being needed

Prefetch Branch Target

- Target of branch is prefetched in addition to instructions following branch

- Keep target until branch is executed

- Used by IBM 360/91

General solutions: Dealing with branches (2)

Loop Buffer

Very fast memory maintained by fetch stage of pipeline

Check buffer before fetching from memory

Very good for small loops or jumps

Used by CRAY-1

Branch Prediction: *Predict when a branch is never taken or when a branch is always taken*

a) Assume that jump will not happen

Always fetch next instruction

Used by 68020 & VAX 11/780

VAX will not prefetch after branch if a page fault would result (O/S v CPU design)

b) Assume that jump will happen

Always fetch target instruction

General solutions: Dealing with branches (3)

Predict by Opcode

Some instructions are more likely to result in a jump

Can get up to 75% success

Taken/Not taken switch

Based on previous history

Good for loops

Delayed Branch

Do not take jump until you have to

Rearrange instructions

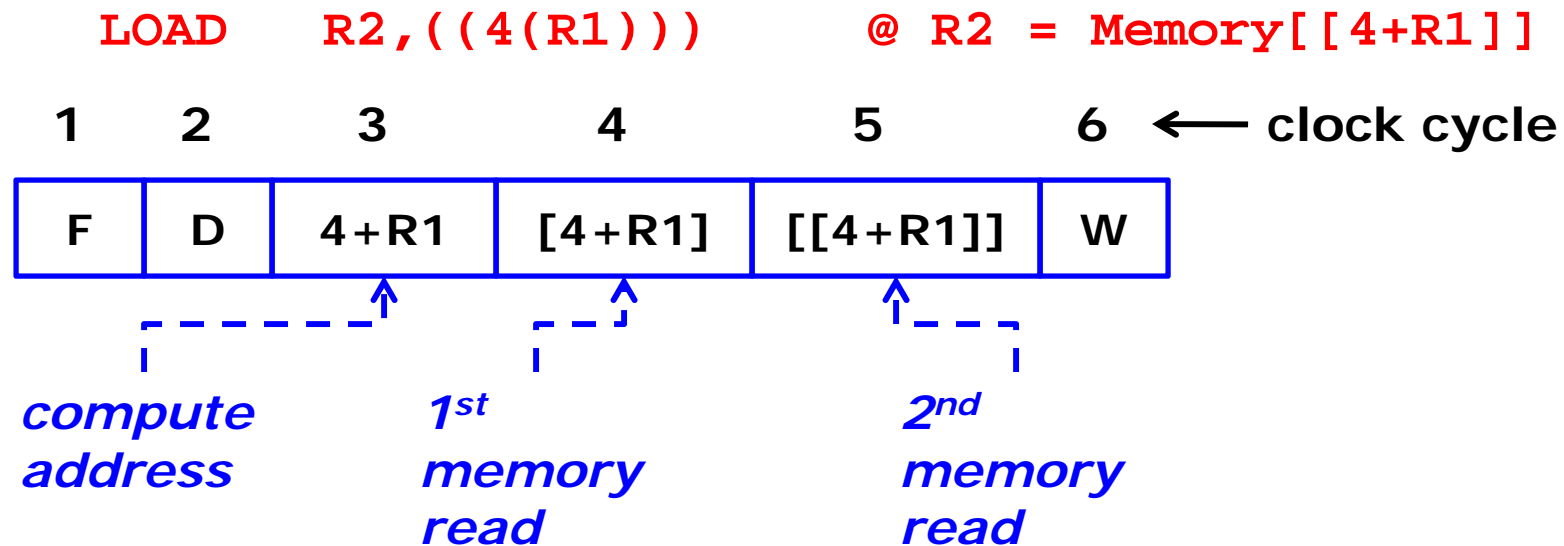
Pipelining and Instruction Sets Design

When deciding on addressing modes to be incorporated in instruction set design, one must be aware of effects on the pipeline

To get maximum advantage from a pipeline:

- access to an operand in only 1 memory access
- only load and store to access memory operands
- no other side effects

Pipelining and Instruction Sets Design: example



If R2 is needed in the next instruction, there would be a 3 cycle stall (data hazard)

Pipelining and Instruction Sets Design: example

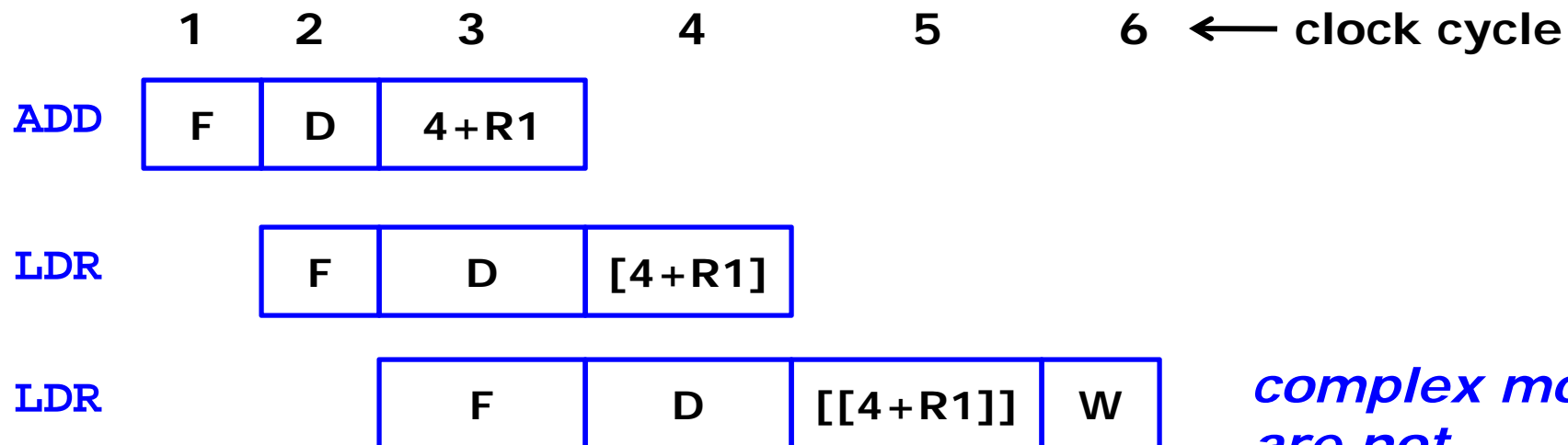
LOAD R2 , ((4(R1))) @ R2 = Memory[[4+R1]]

Implement it in RISC:

ADD R1,R1,#4

LDR R2,[R1]

LDR R2,[R2]



Same number of clock cycles!

*complex modes
are not
necessarily
faster – just
less code*

Performance and Memory and Cache and Virtual Memory

- All units must work at the same speed
- The clock is determined by the slowest of the units
- Important to be able to quantify the impact of a pipeline in processing a sequence of data
- If one unit is faster, it waits idle

Crucial step: FETCH → it deals with memory access

Cache: can make Fetch step as quick as other pipeline stages

What are the biggest advantages of pipelining?

- ❑ Pipelining improves performance by increasing instruction throughput
 - as opposed to decreasing the execution time of an individual instruction
- ❑ Pipelining is a technique that exploits parallelism among the instructions in a sequential instruction stream
 - invisible to the programmer (unlike programming a multiprocessor system)

UltraSPARC II

