PIPELINING: BASIC AND INTERMEDIATE CONCEPTS

CHAPTER # 03

WHAT IS PIPELINING?

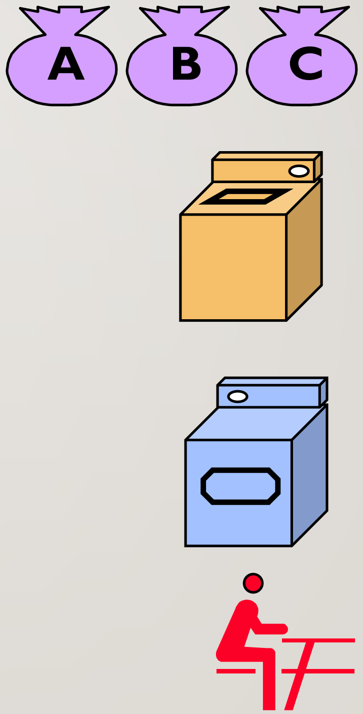
- Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction.
- Today, pipelining is the key implementation technique used to make fast processors, and even processors that cost less than a dollar are pipelined.
- A pipeline is like an assembly line. In an automobile assembly line, there are many steps, each contributing something to the construction of the car.

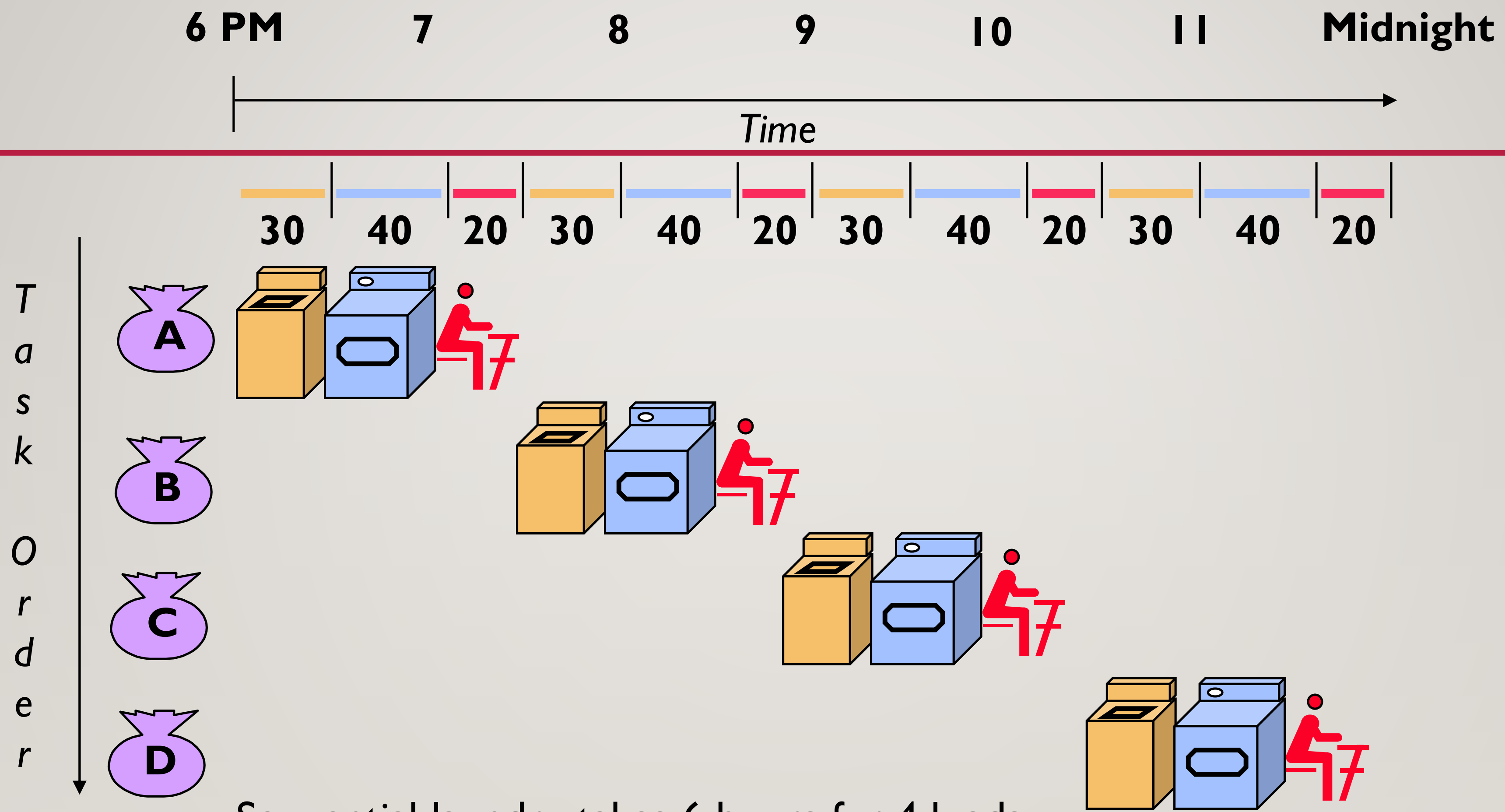
WHAT IS PIPELINING?

- Each step operates in parallel with the other steps, although on a different car.
- In a computer pipeline, each step in the pipeline completes a part of an instruction. Like the assembly line, different steps are completing different parts of different instructions in parallel. Each of these steps is called a pipe stage or a pipe segments.
- The stages are connected one to the next to form a pipe—instructions enter at one end, progress through the stages, and exit at the other end, just as cars would in an assembly line.

WHAT IS PIPELINING

- Laundry Example
- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, and fold
- Washer takes 30 minutes
- Dryer takes 40 minutes
- "Folder" takes 20 minutes

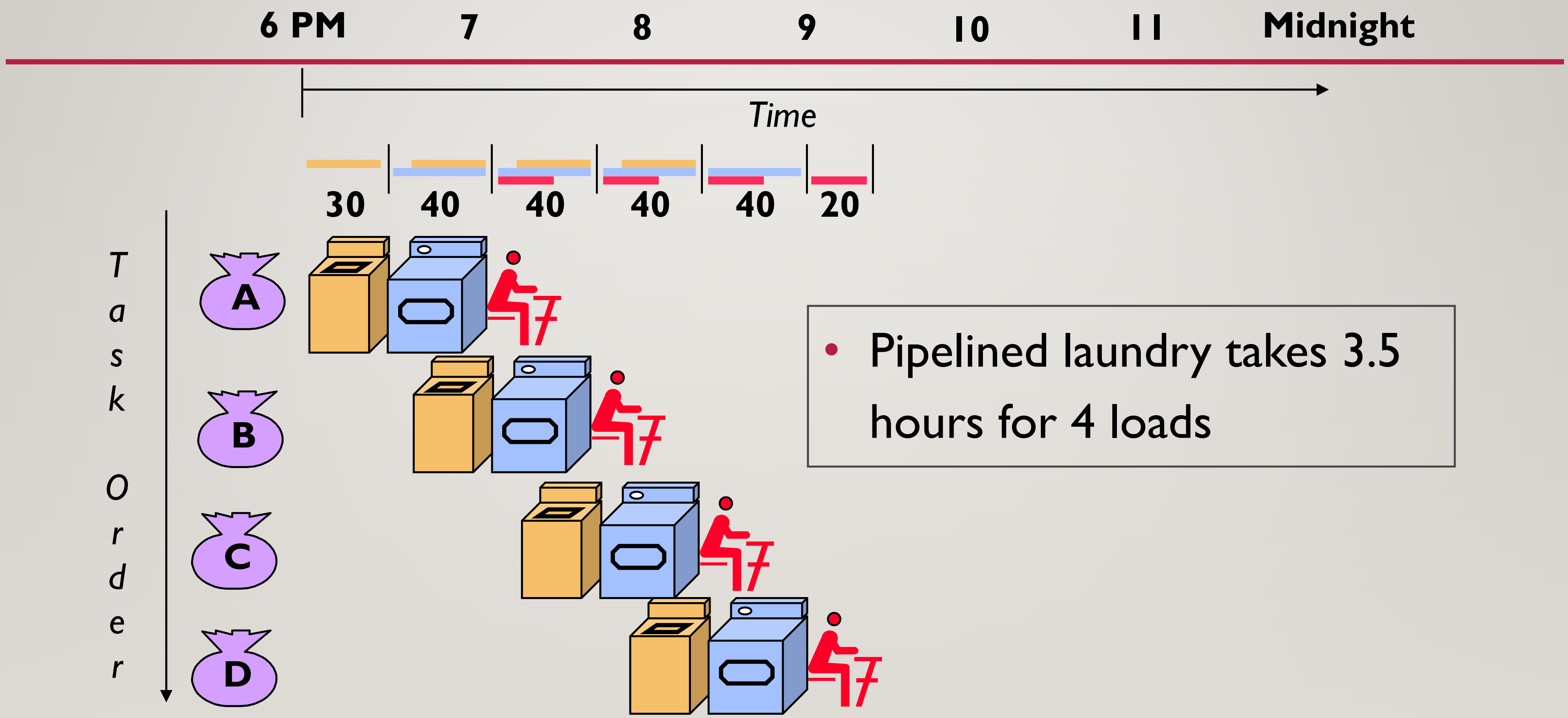




Sequential laundry takes 6 hours for 4 loads

If they learned pipelining, how long would laundry take?

WHAT IS PIPELINING
START WORK ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

WHAT IS PIPELINING

- In an automobile assembly line, throughput is defined as the number of cars per hour and is determined by how often a completed car exits the assembly line.
- Likewise, the throughput of an instruction pipeline is determined by how often an instruction exits the pipeline.
- Because the pipe stages are hooked together, all the stages must be ready to proceed at the same time, just as we would require in an assembly line.

WHAT IS PIPELINING

- The time required between moving an instruction one step down the pipeline is a processor cycle.
- Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage, just as in an auto assembly line the longest step would determine the time between advancing cars in the line.
- In a computer, this processor cycle is almost always 1 clock cycle.

WHAT IS PIPELINING

- The pipeline designer's goal is to balance the length of each pipeline stage, just as the designer of the assembly line tries to balance the time for each step in the process. If the stages are perfectly balanced, then the time per instruction on the pipelined processor—assuming ideal conditions—is equal to:

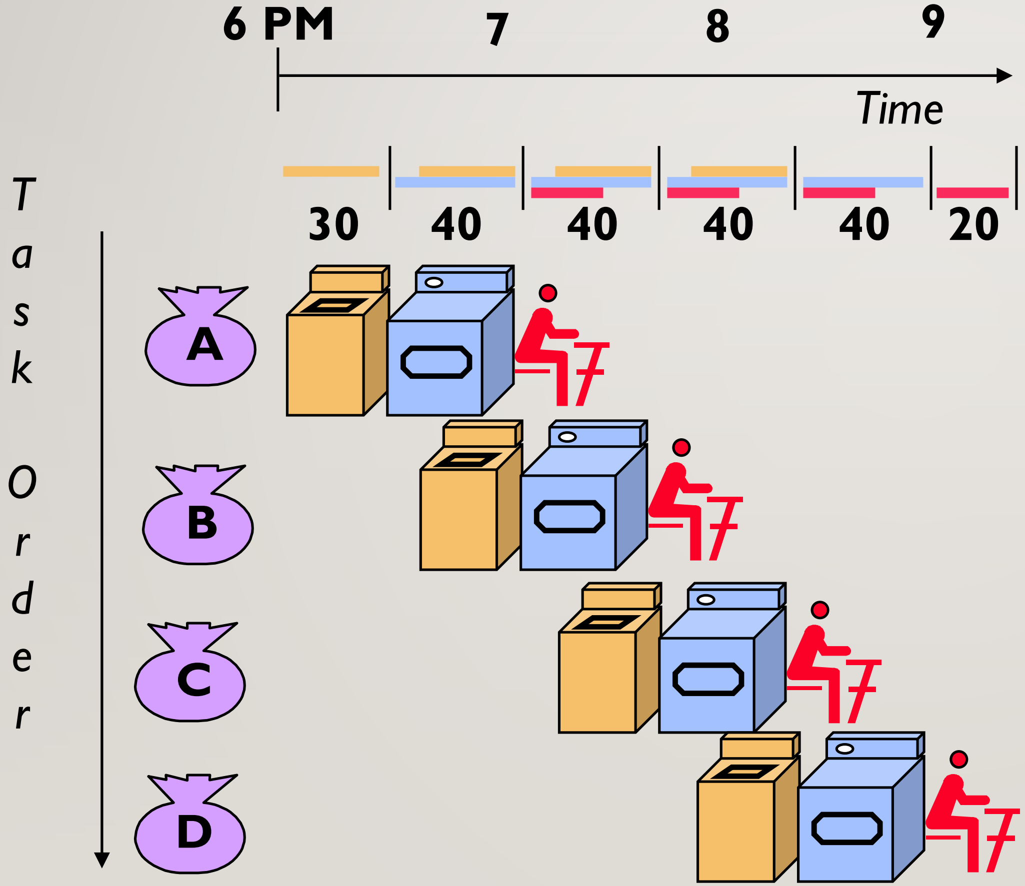
$$\frac{\text{Time per instruction on unpipelined machine}}{\text{Number of pipe stages}}$$

WHAT IS PIPELINING

- Under these conditions, the speedup from pipelining equals the number of pipe stages, just as an assembly line with n stages can ideally produce cars n times as fast. Usually, however, the stages will not be perfectly balanced; furthermore, pipelining does involve some overhead.
- Thus, the time per instruction on the pipelined processor will not have its minimum possible value, yet it can be close.
- Pipelining yields a reduction in the average execution time per instruction. If the starting point is a processor that takes multiple clock cycles per instruction, then pipelining reduces the CPI. This is the primary view we will take.

What Is Pipelining

PIPELINING LESSONS



- Pipelining doesn't help latency of single task, it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Multiple tasks operating simultaneously
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup

A SIMPLE IMPLEMENTATION OF A RISC INSTRUCTION SET

- Every instruction in this RISC subset can be implemented in, at most, 5 clock cycles. The 5 clock cycles are as follows.
- **Instruction fetch cycle (IF):**
- Send the program counter (PC) to memory and fetch the current instruction from memory. Update the PC to the next sequential instruction by adding 4 (because each instruction is 4 bytes) to the PC.

A SIMPLE IMPLEMENTATION OF A RISC INSTRUCTION SET

- **Instruction decode/register fetch cycle (ID):**
- Decode the instruction and read the registers corresponding to register source specifiers from the register file.
- Do the equality test on the registers as they are read, for a possible branch. Sign-extend the offset field of the instruction in case it is needed.
- Compute the possible branch target address by adding the sign-extended offset to the incremented PC.

A SIMPLE IMPLEMENTATION OF A RISC INSTRUCTION SET

- **Execution/effective address cycle (EX):**
- The ALU operates on the operands prepared in the prior cycle, performing one of three functions, depending on the instruction type.
- ■ Memory reference—The ALU adds the base register and the offset to form the effective address.
- ■ Register-Register ALU instruction—The ALU performs the operation specified by the ALU opcode on the values read from the register file.
- ■ Register-Immediate ALU instruction—The ALU performs the operation specified by the ALU opcode on the first value read from the register file and the sign-extended immediate.
- ■ Conditional branch—Determine whether the condition is true.

A SIMPLE IMPLEMENTATION OF A RISC INSTRUCTION SET

- **Memory access (MEM):**
- If the instruction is a load, the memory does a read using the effective address computed in the previous cycle. If it is a store, then the memory writes the data from the second register read from the register file using the effective address.
- **Write-back cycle (WB):**
- Write the result into the register file, whether it comes from the memory system (for a load) or from the ALU (for an ALU instruction).

A SIMPLE IMPLEMENTATION OF A RISC INSTRUCTION SET

ALU Instructions: `op $x,$y,$z`

Instr. Fetch &. PC Increm.	Read of Source Regs. `$y` and `$z`	ALU OP (`$y op $z`)	Write Back of Destinat. Reg. `$x`

Load Instructions: `lw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y`	ALU Op. (`$y+offset`)	Read Mem. `M($y+offset)`	Write Back of Destinat. Reg. `$x`

Store Instructions: `sw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y` & Source `$x`	ALU Op. (`$y+offset`)	Write Mem. `M($y+offset)`

Conditional Branch: `beq $x,$y,offset`

Instr. Fetch & PC Increm.	Read of Source Regs. `$x` and `$y`	ALU Op. (`$x-$y`) & (`PC+4+offset`)

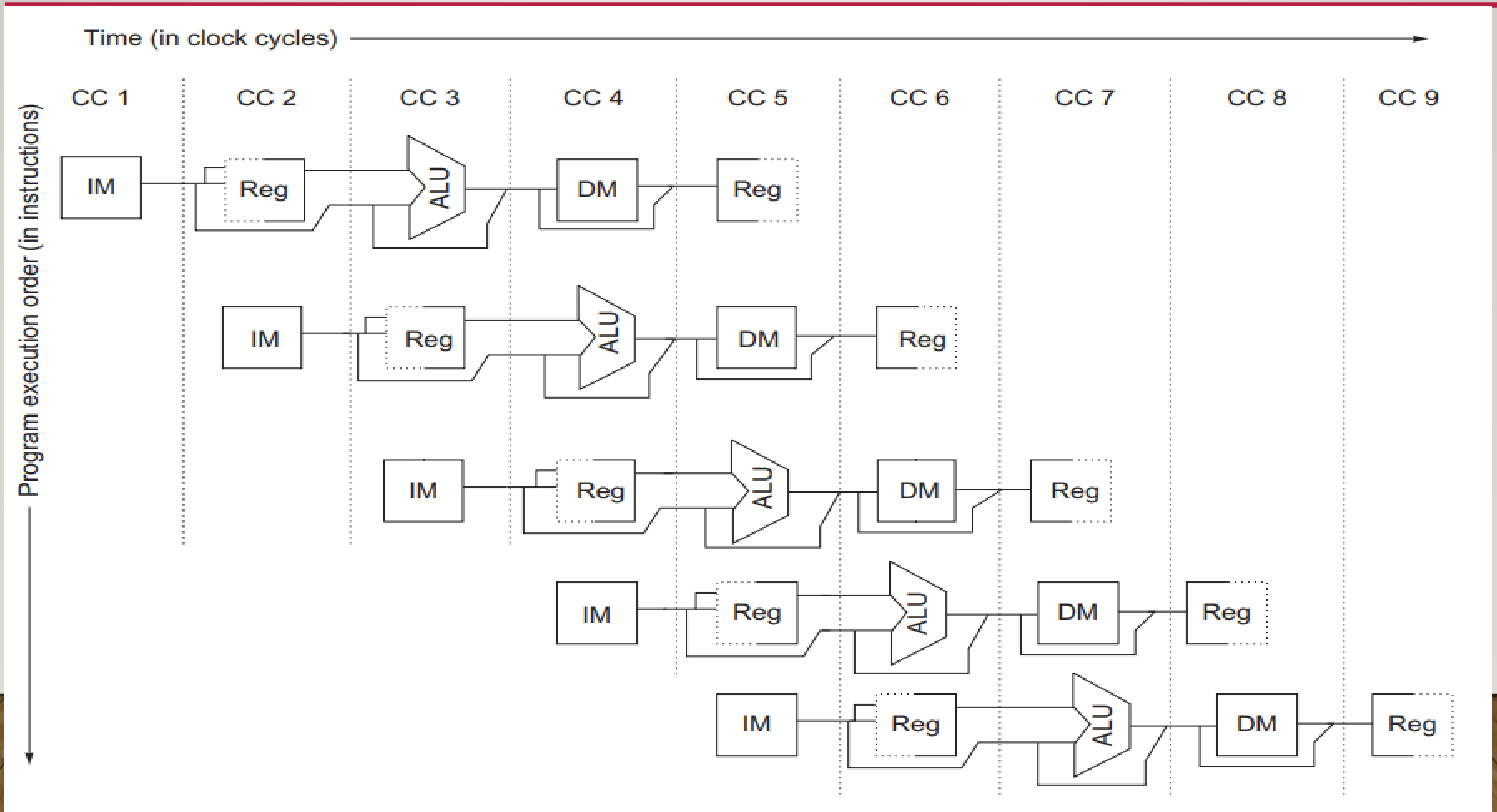
THE CLASSIC FIVE-STAGE PIPELINE FOR A RISC PROCESSOR

- We can pipeline the execution described in the previous section with almost no changes by simply starting a new instruction on each clock cycle.
- Each of the clock cycles from the previous section becomes a pipe stage—a cycle in the pipeline. This results in the execution pattern shown in Figure, which is the typical way a pipeline structure is drawn.
- Although each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions.

THE CLASSIC FIVE-STAGE PIPELINE FOR A RISC PROCESSOR

	Clock number								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction i	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

THE CLASSIC FIVE-STAGE PIPELINE FOR A RISC PROCESSOR



BASIC PERFORMANCE ISSUES IN PIPELINING

- Pipelining increases the processor instruction throughput—the number of instructions completed per unit of time—but it does not reduce the execution time of an individual instruction. In fact, it usually slightly increases the execution time of each instruction due to overhead in the control of the pipeline.
- In addition to limitations arising from pipeline latency, limits arise from imbalance among the pipe stages and from pipelining overhead. Imbalance among the pipe stages reduces performance because the clock can run no faster than the time needed for the slowest pipeline stage. Pipeline overhead arises from the combination of pipeline register delay and clock skew.

EXAMPLE

- Consider the un-pipelined processor in the previous section. Assume that it has a 4 GHz clock (or a 0.5 ns clock cycle) and that it uses four cycles for ALU operations and branches and five cycles for memory operations. Assume that the relative frequencies of these operations are 40%, 20%, and 40%, respectively. Suppose that due to clock skew and setup, pipelining the processor adds 0.1 ns of overhead to the clock. Ignoring any latency impact, *how much speedup in the instruction execution rate will we gain from a pipeline*?

SOLUTION

- The average instruction execution time on the un-pipelined processor is:

$$\text{CPU time} = \left(\sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i\right) \times \text{Clock cycle time}$$

$$= 0.5\text{ ns} \times [(40\% + 20\%) \times 4 + 40\% \times 5]$$

$$= 0.5\text{ ns} \times 4.4$$

$$= 2.2\text{ ns}$$

In the pipelined implementation, the clock must run at the speed of the slowest stage plus overhead, which will be 0.5 + 0.1 or 0.6 ns; this is the average instruction execution time. Thus, the speedup from pipelining is

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

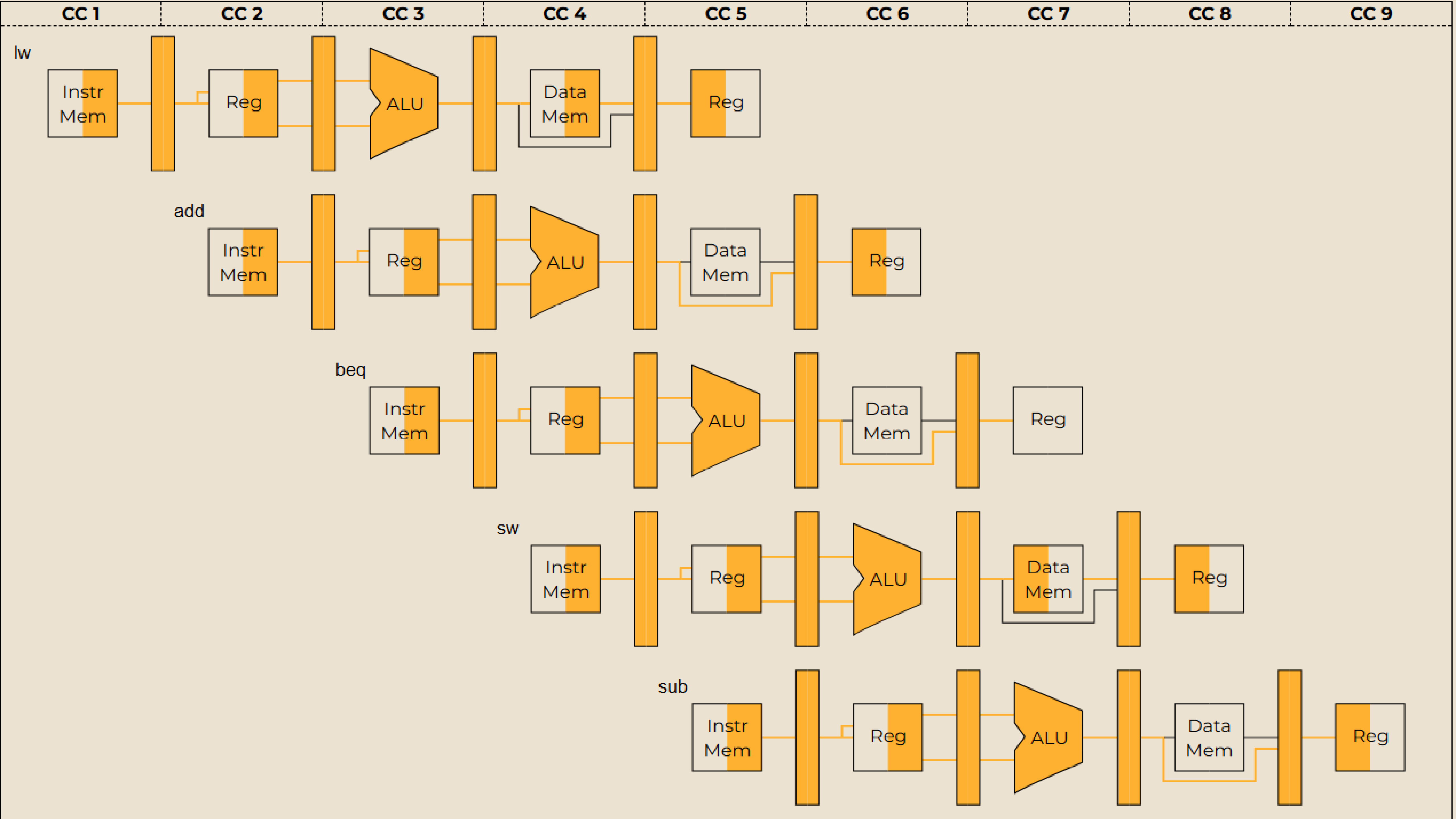
$$= \frac{2.2\text{ ns}}{0.6\text{ ns}} = 3.7\text{ times}$$

THE MAJOR HURDLE OF PIPELINING—PIPELINE HAZARDS

- There are situations, called hazards, that prevent the next instruction in the instruction stream from executing during its designated clock cycle.
- Hazards reduce the performance from the ideal speedup gained by pipelining.
- There are three classes of hazards:
- 1. Structural hazards
- 2. Data hazards
- 3. Control hazards

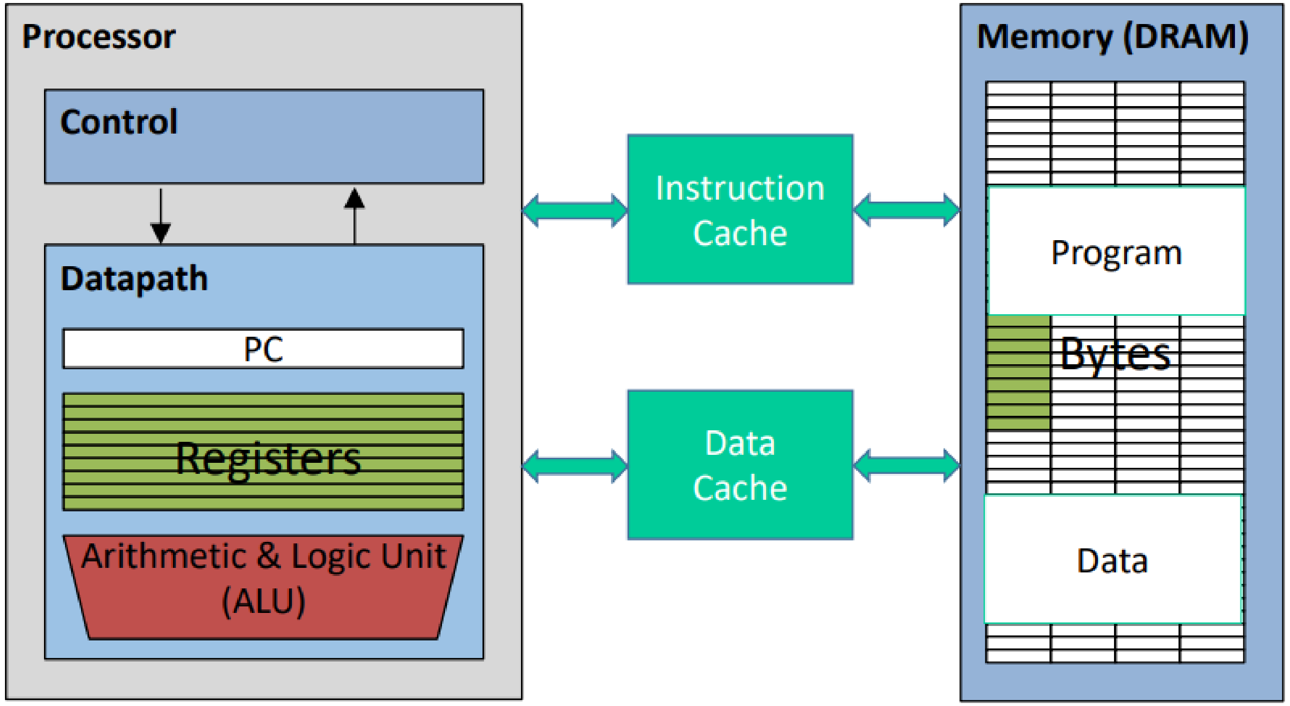
THE MAJOR HURDLE OF PIPELINING—PIPELINE HAZARDS

- **Structural hazards:** HW cannot support this combination of instructions (single person to fold and put clothes away).
- **Data hazards:** Instruction depends on result of prior instruction still in the pipeline (missing sock)
- **Control hazards:** Pipelining of branches & other instructions that change the PC
- Common solution is to **stall** the pipeline until the hazard is resolved, inserting one or more "**bubbles**" in the pipeline

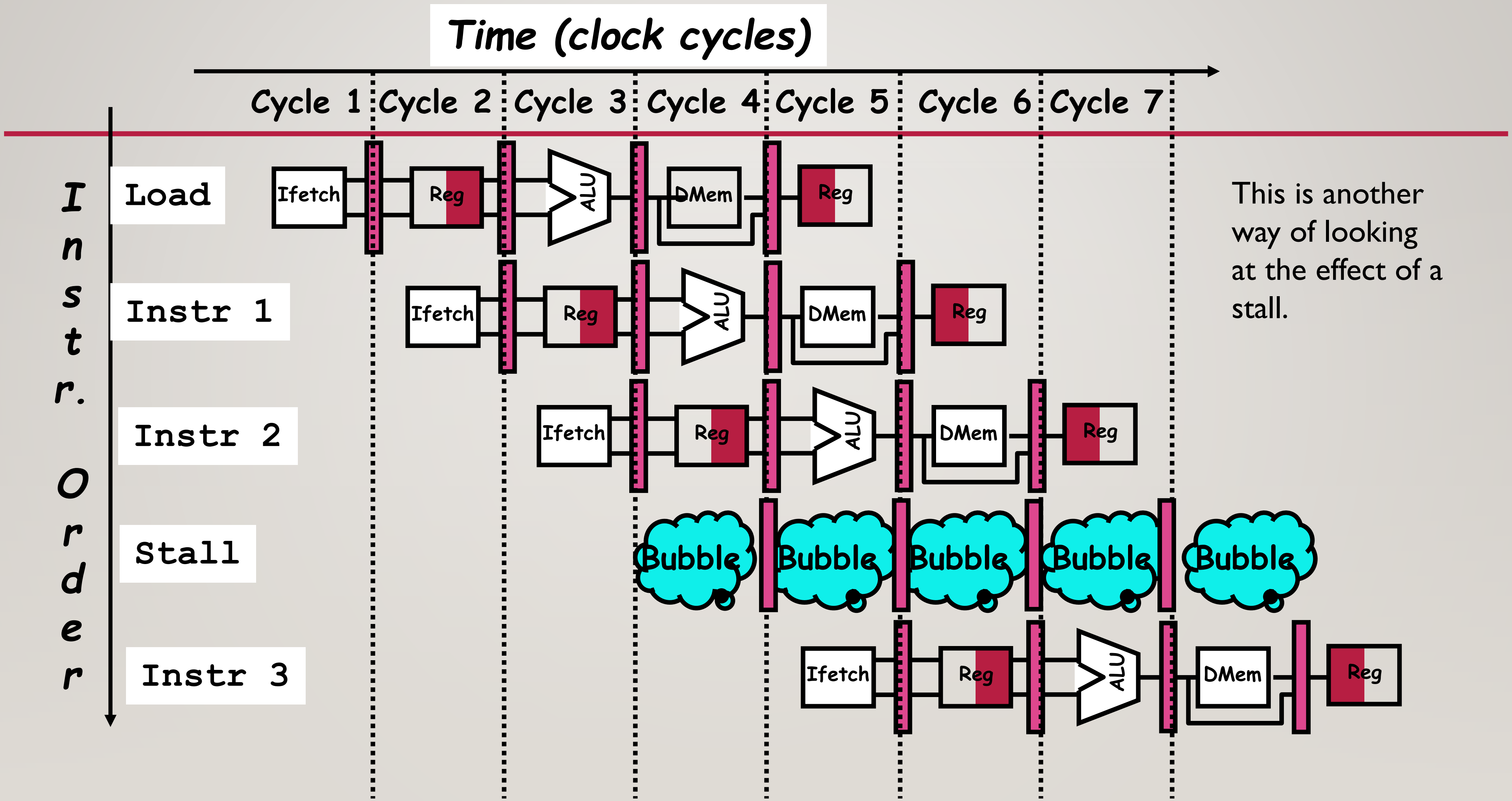


CC 1
CC 2
CC 3
CC 4
CC 5
CC 6
CC 7
CC 8
CC 9
lw
Instr Mem
Reg
ALU
Data Mem
Reg
add
Instr Mem
Reg
ALU
Data Mem
Reg
beq
Instr Mem
Reg
ALU
Data Mem
Reg
sw
Instr Mem
Reg
ALU
Data Mem
Reg
sub
Instr Mem
Reg
ALU
Data Mem
Reg

STRUCTURAL HAZARDS:



Caches: small and fast “buffer” memories



This is another way of looking at the effect of a stall.

STALLS IN PIPELINE

Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

DATA HAZARDS

- A data hazard occurs when the pipeline execution must be stalled because one step must wait for another one to complete. This comes up when a planned instruction can not execute in the planned clock cycle because the data needed is not yet available.
- Read After Write (RAW) Instr_j tries to read operand before Instr_I writes it.

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

DATA HAZARDS

- Write After Read (WAR) $Instr_J$ tries to write operand *before* $Instr_I$ reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Write After Write (WAW) $Instr_J$ tries to write operand *before* $Instr_I$ writes it

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

EXAMPLE

- Consider the pipelined execution of these instructions:
- add x1,x2,x3
- sub x4,x1,x5
- and x6,x1,x7
- or x8,x1,x9
- xor x10,x1,x11

Time (clock cycles)

IF ID/RF EX MEM WB

I n s t r. O r d e r

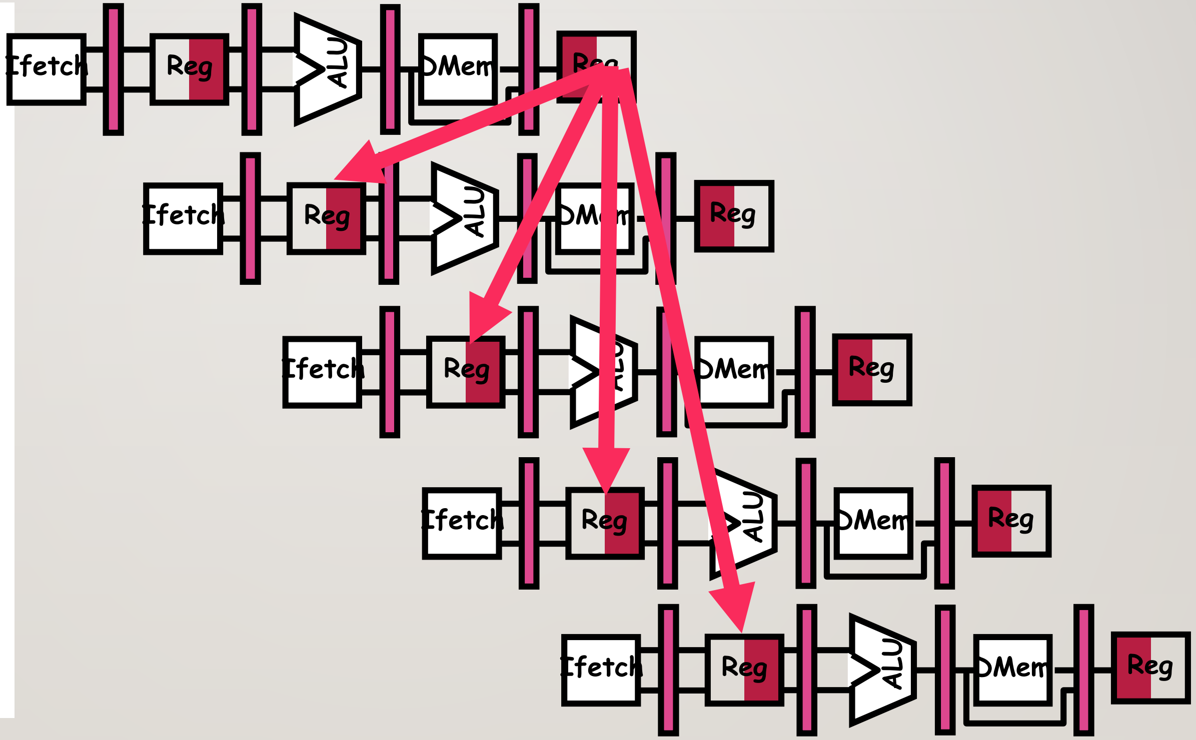
add r1,r2,r3

sub r4,r1,r3

and r6,r1,r7

or r8,r1,r9

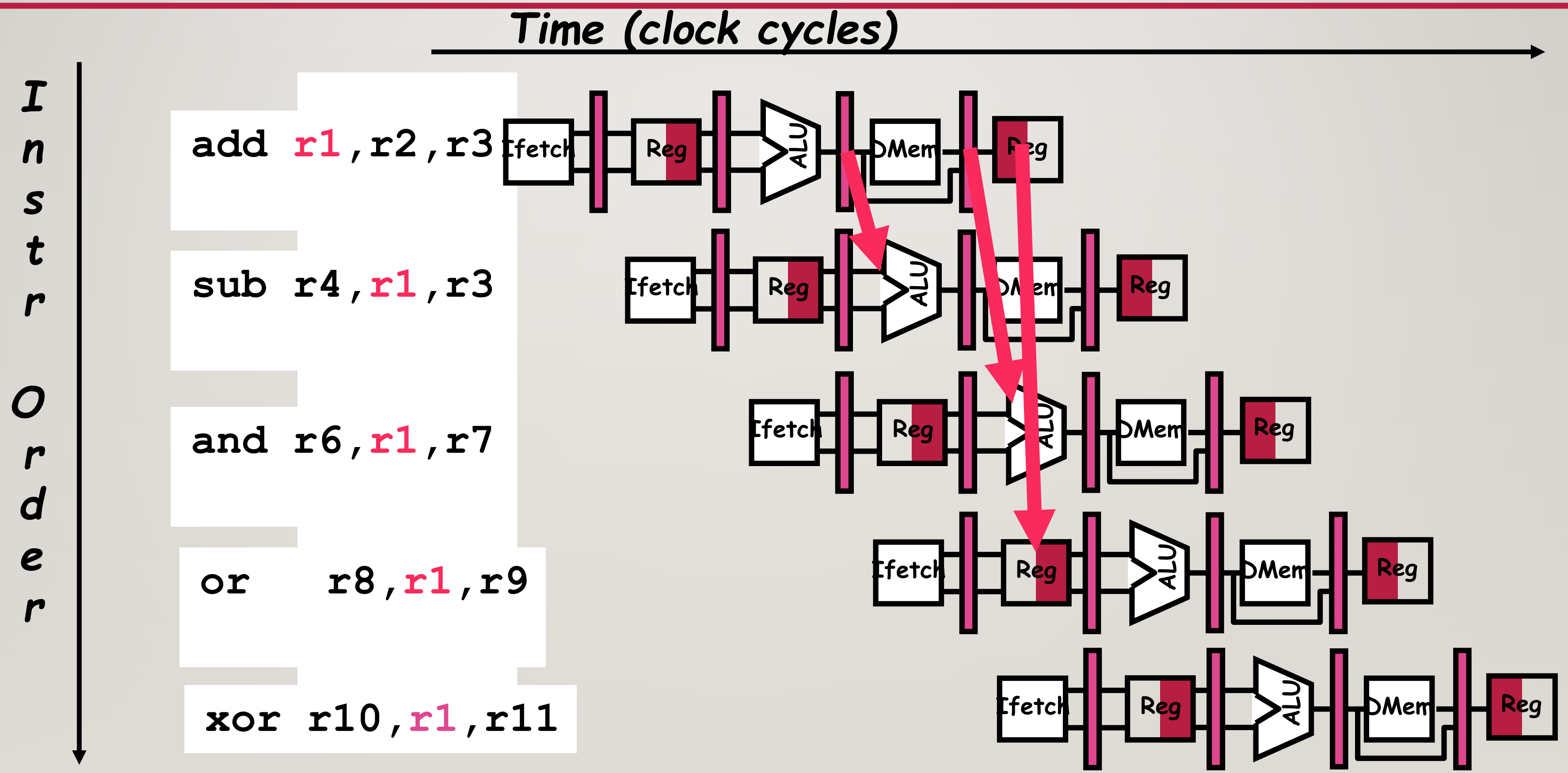
xor r10,r1,r11



The use of the result of the ADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

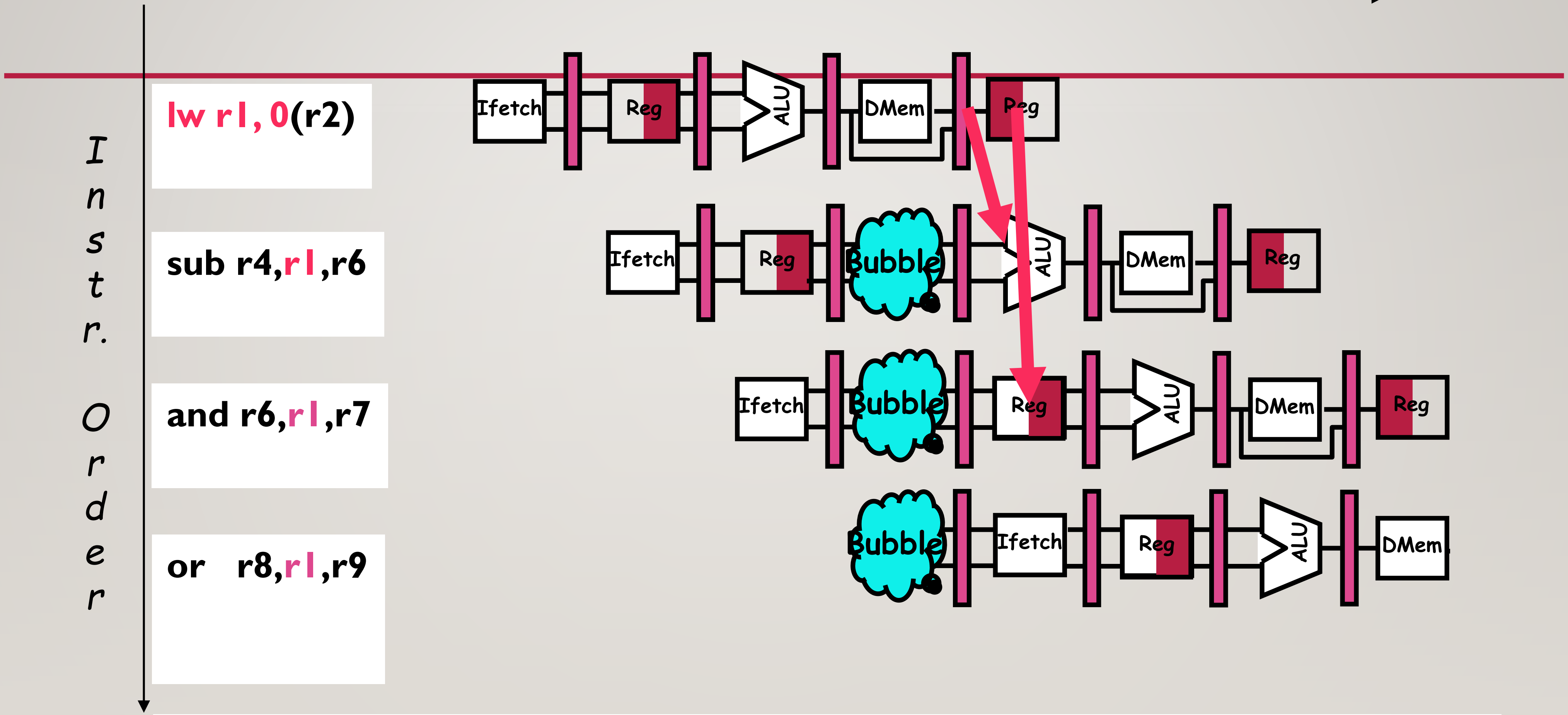
Forwarding To Avoid Data Hazard

Forwarding is the concept of making data available to the input of the ALU for subsequent instructions, even though the generating instruction hasn't gotten to WB in order to write the memory or registers.



Time (clock cycles)

The stall is necessary as shown here.



There are some instances where hazards occur, even with forwarding.

This is another representation of the stall.

LW R1, 0(R2)	IF	ID	EX	MEM	WB			
SUB R4, R1, R5		IF	ID	EX	MEM	WB		
AND R6, R1, R7			IF	ID	EX	MEM	WB	
OR R8, R1, R9				IF	ID	EX	MEM	WB

LW R1, 0(R2)	IF	ID	EX	MEM	WB				
SUB R4, R1, R5		IF	ID	stall	EX	MEM	WB		
AND R6, R1, R7			IF	stall	ID	EX	MEM	WB	
OR R8, R1, R9				stall	IF	ID	EX	MEM	WB

Pipeline Scheduling

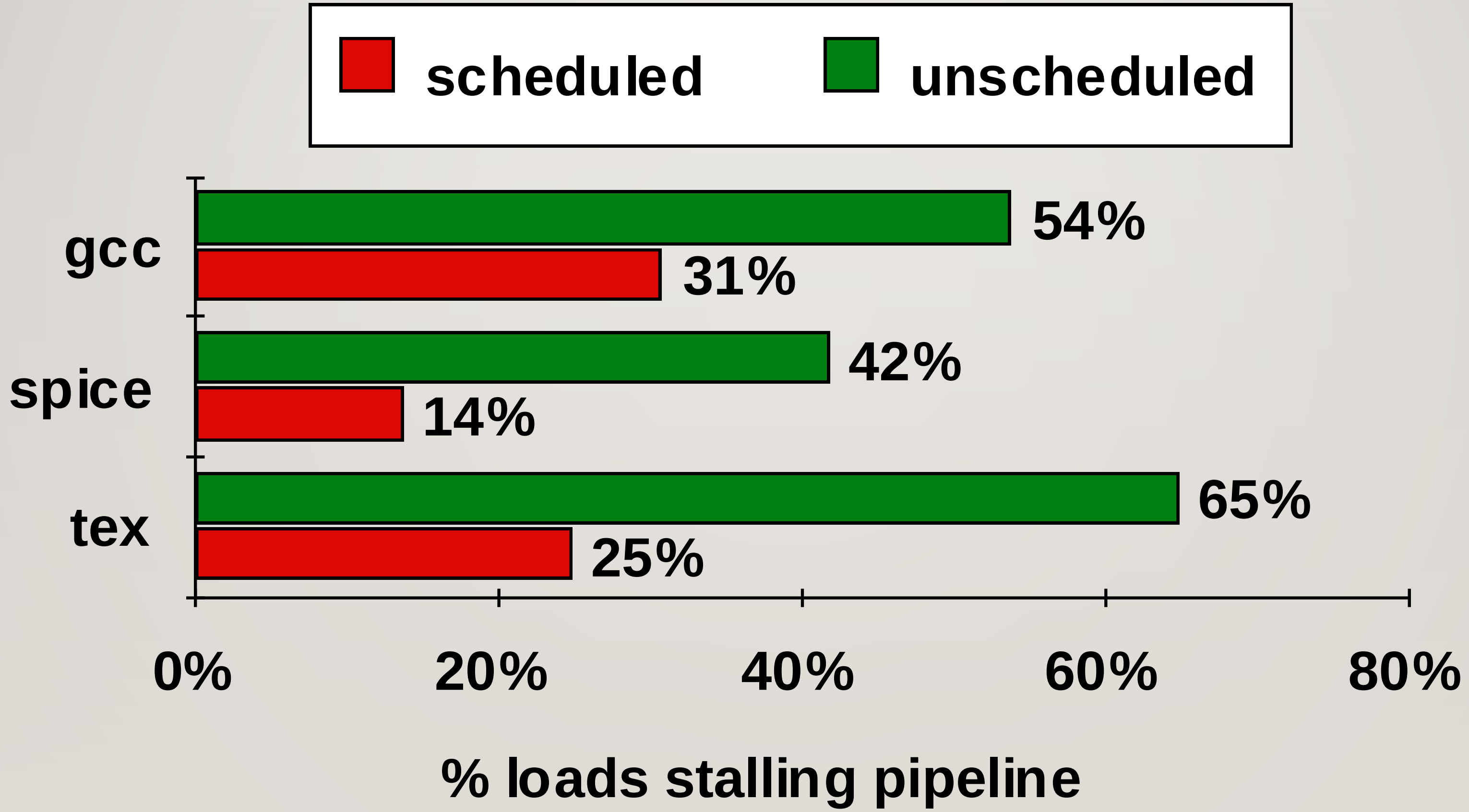
Instruction scheduled by compiler - move instruction in order to reduce stall.

lw Rb, b	code sequence for **a = b+c** before scheduling
lw Rc, c	
Add Ra, Rb, Rc	stall
sw a, Ra	
lw Re, e	code sequence for **d = e+f** before scheduling
lw Rf, f	
sub Rd, Re, Rf	stall
sw d, Rd	

Arrangement of code after scheduling.

lw Rb, b
lw Rc, c
lw Re, e
Add Ra, Rb, Rc
lw Rf, f
sw a, Ra
sub Rd, Re, Rf
sw d, Rd

Pipeline Scheduling



CONTROL HAZARDS

- A control hazard is when we need to find the destination of a branch, and can't fetch any new instructions until we know that destination.
- Control hazards can cause a greater performance loss for our RISC V pipeline than do data hazards. When a branch is executed, it may or may not change the PC to something other than its current value plus 4. Recall that if a branch changes the PC to its target address, it is a taken branch; if it falls through, it is not taken, or untaken. If instruction *i* is a taken branch, then the PC is usually not changed until the end of ID, after the completion of the address calculation and comparison.

CONTROL HAZARD ON BRANCHES THREE STAGE STALL

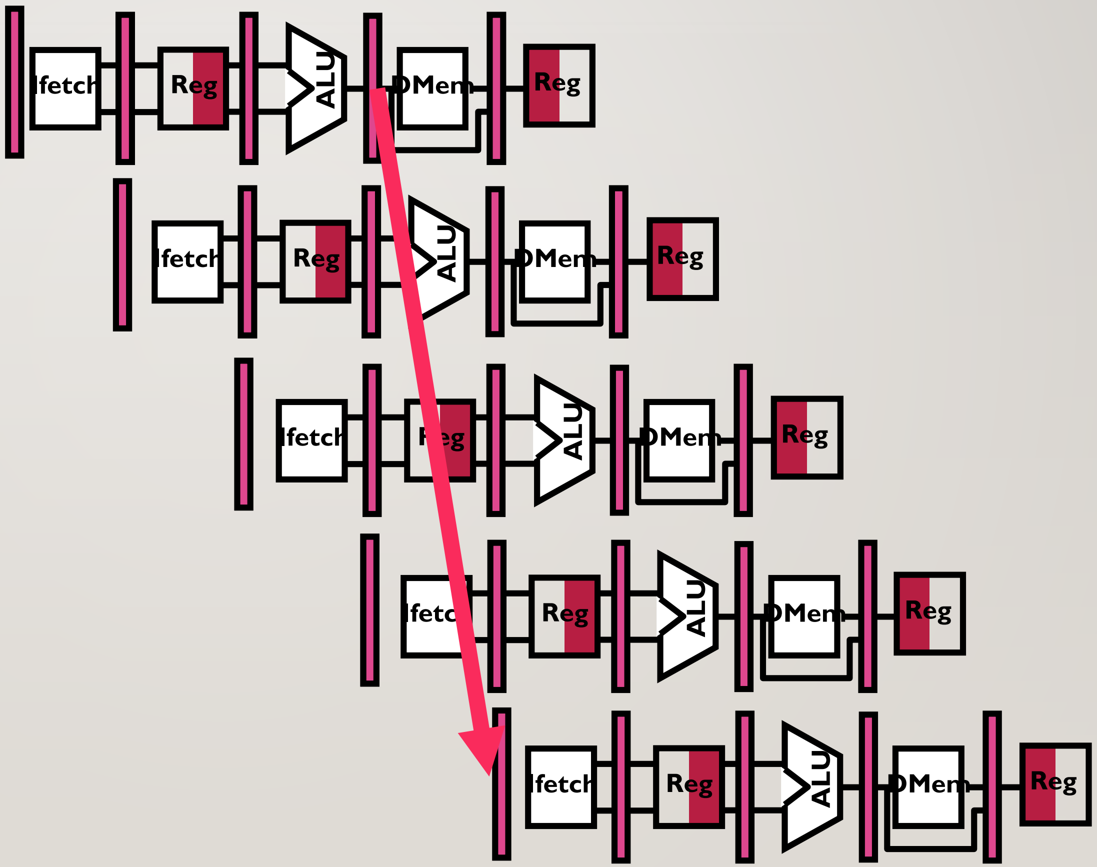
10: beq r1,r3,36

14: and r2,r3,r5

18: or r6,r1,r7

22: add r8,r1,r9

36: xor r10,r1,r11



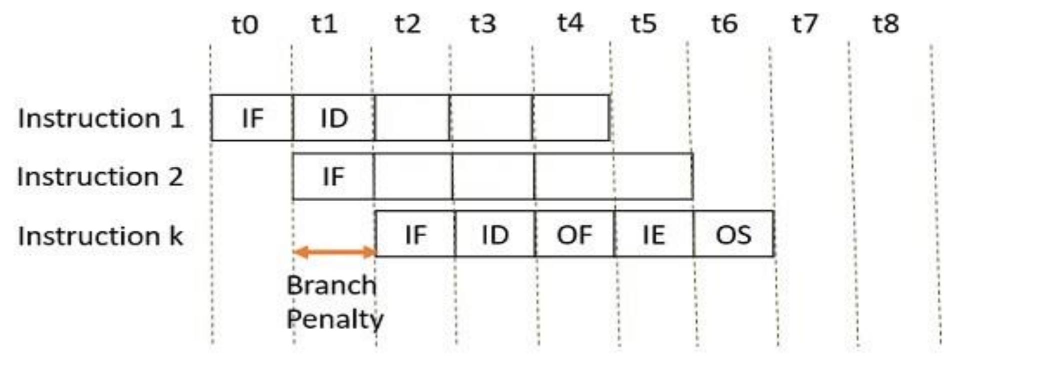
REDUCING PIPELINE BRANCH PENALTIES

- *1. Freeze or flush the pipeline: the simplest scheme*
- Hold or delete any instructions after the branch until the branch destination is known .
- This is the solution shown in Figure
- The branch penalty is fixed and cannot be reduced by software

Branch instruction	IF	ID	EX	MEM	WB		
Branch successor		IF	IF	ID	EX	MEM	WB
Branch successor + 1				IF	ID	EX	MEM
Branch successor + 2					IF	ID	EX

REDUCING PIPELINE BRANCH PENALTIES

- *2.Treat every branch as not taken.*
- Continue to fetch instructions as if there were no branch.
- Restart fetch at target address (and turn previously fetched instruction into a NOP) if the branch is taken.



Branch Delay when Target Branch is Determined at Decode stage

REDUCING PIPELINE BRANCH PENALTIES

- *3. Treat every branch as taken*
- As soon as the branch is decoded and the target address is computed, we assume the branch to be taken and begin fetching and executing at the target..
- This buys us a one-cycle improvement when the branch is actually taken, because we know the target address at the end of ID, one cycle before we know whether the branch condition is satisfied in the ALU stage. In either a predicted-taken or predicted-not-taken scheme, the compiler can improve performance by organizing the code so that the most frequent path matches the hardware's choice.

REDUCING PIPELINE BRANCH PENALTIES

Untaken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB
Taken branch instruction	IF	ID	EX	MEM	WB				
Instruction $i+1$		IF	**idle**	**idle**	**idle**	**idle**			
Branch target			IF	ID	EX	MEM	WB		
Branch target $+1$				IF	ID	EX	MEM	WB	
Branch target $+2$					IF	ID	EX	MEM	WB

REDUCING PIPELINE BRANCH PENALTIES

- A fourth scheme, which was heavily used in early RISC processors is called delayed branch. In a delayed branch, the execution cycle with a branch delay of one is:

```
branch instruction
sequential successor1
branch target if taken
```

- Although it is possible to have a branch delay longer than one, in practice almost all processors with delayed branch have a single instruction delay; other techniques are used if the pipeline has a longer potential branch penalty. The job of the compiler is to make the successor instructions valid and useful.

REDUCING PIPELINE BRANCH PENALTIES

Untaken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB			
Instruction $i+2$			IF	ID	EX	MEM	WB		
Instruction $i+3$				IF	ID	EX	MEM	WB	
Instruction $i+4$					IF	ID	EX	MEM	WB

Taken branch instruction	IF	ID	EX	MEM	WB				
Branch delay instruction ($i+1$)		IF	ID	EX	MEM	WB			
Branch target			IF	ID	EX	MEM	WB		
Branch target + 1				IF	ID	EX	MEM	WB	
Branch target + 2					IF	ID	EX	MEM	WB

PERFORMANCE OF PIPELINES WITH STALLS

$$\text{Speedup from pipelining} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined} \times \text{Clock cycle unpipelined}}{\text{CPI pipelined} \times \text{Clock cycle pipelined}}$$

$$\text{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$

$$= 1 + \text{Pipelines stall clock cycles per instruction}$$

$$\text{Speedup} = \frac{\text{CPI unpiplined}}{1 + \text{Pipeline stall cycles per instruction}}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles per instruction}}$$

PERFORMANCE OF BRANCH SCHEMES

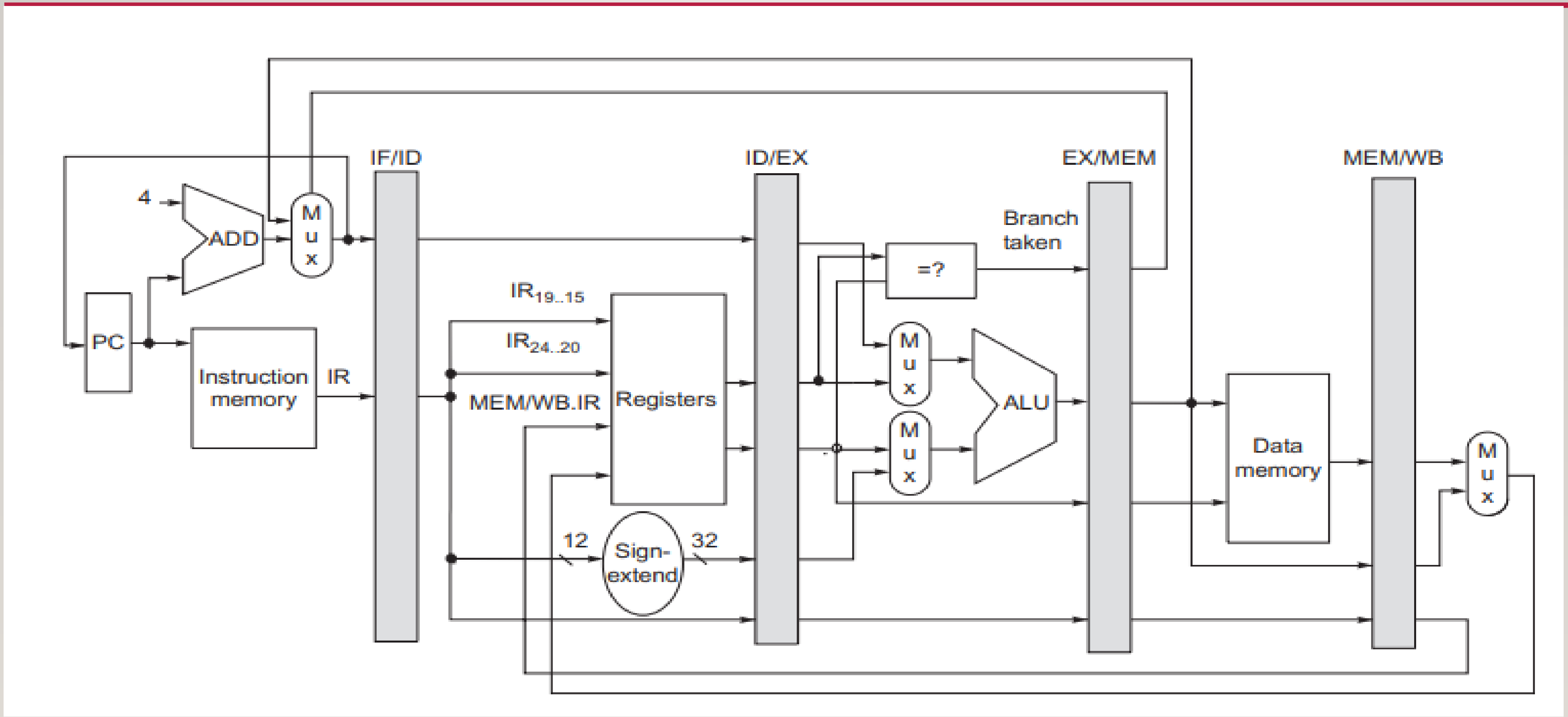
- What is the effective performance of each of these schemes?
- The effective pipeline speedup with branch penalties, assuming an ideal CPI of 1, is

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall cycles from branches}}$$

Pipeline stall cycles from branches = Branch frequency x Branch penalty

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

A BASIC PIPELINE FOR RISCV



EXCEPTION IN PIPELINE

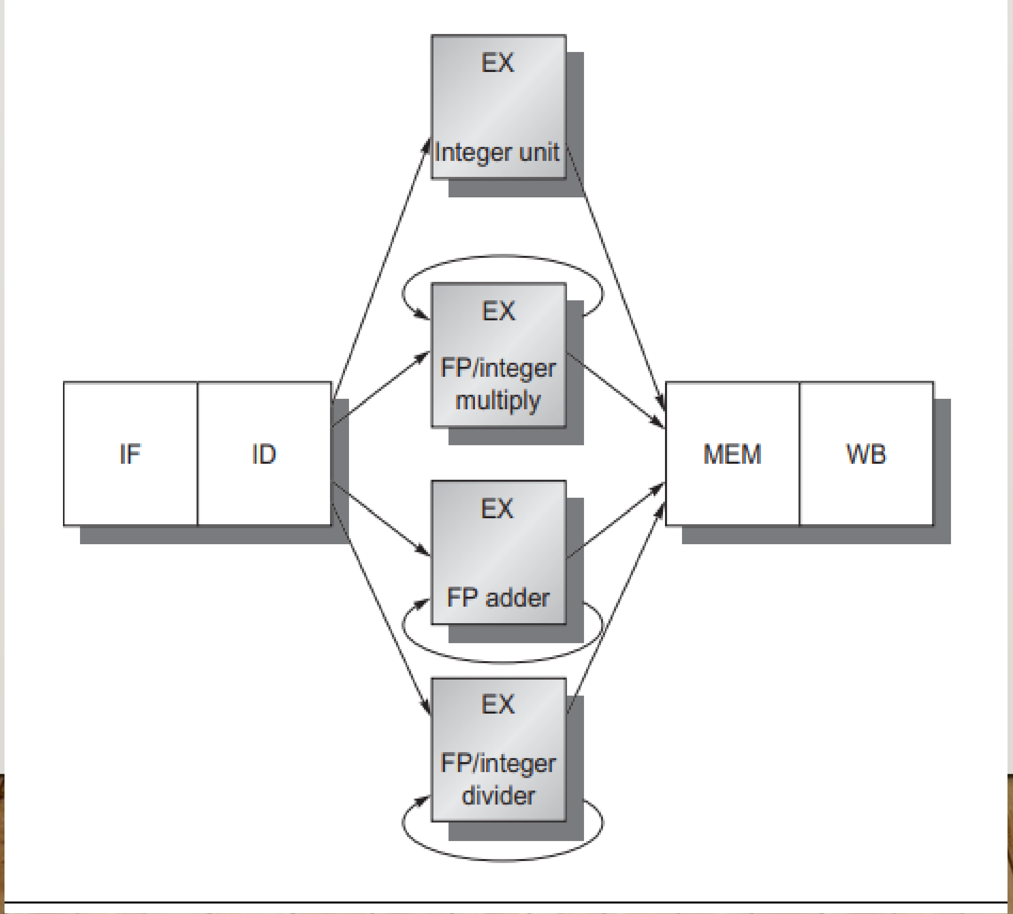
- The terminology used to describe exceptional situations where the normal execution order of instruction is changed varies among processors. The terms interrupt, fault, and exception are used, although not in a consistent fashion. We use the term exception to cover all these mechanisms,
- ■ I/O device request ■ Invoking an operating system service from a user program.
- ■ Tracing instruction execution ■ Breakpoint (programmer-requested interrupt)
- ■ Integer arithmetic overflow■ FP arithmetic anomaly ■ Page fault (not in main memory) ■ Misaligned memory accesses (if alignment is required) ■ Memory protection violation ■ Using an undefined or unimplemented instruction ■ Hardware malfunctions ■ Power failure

EXCEPTION IN PIPELINE

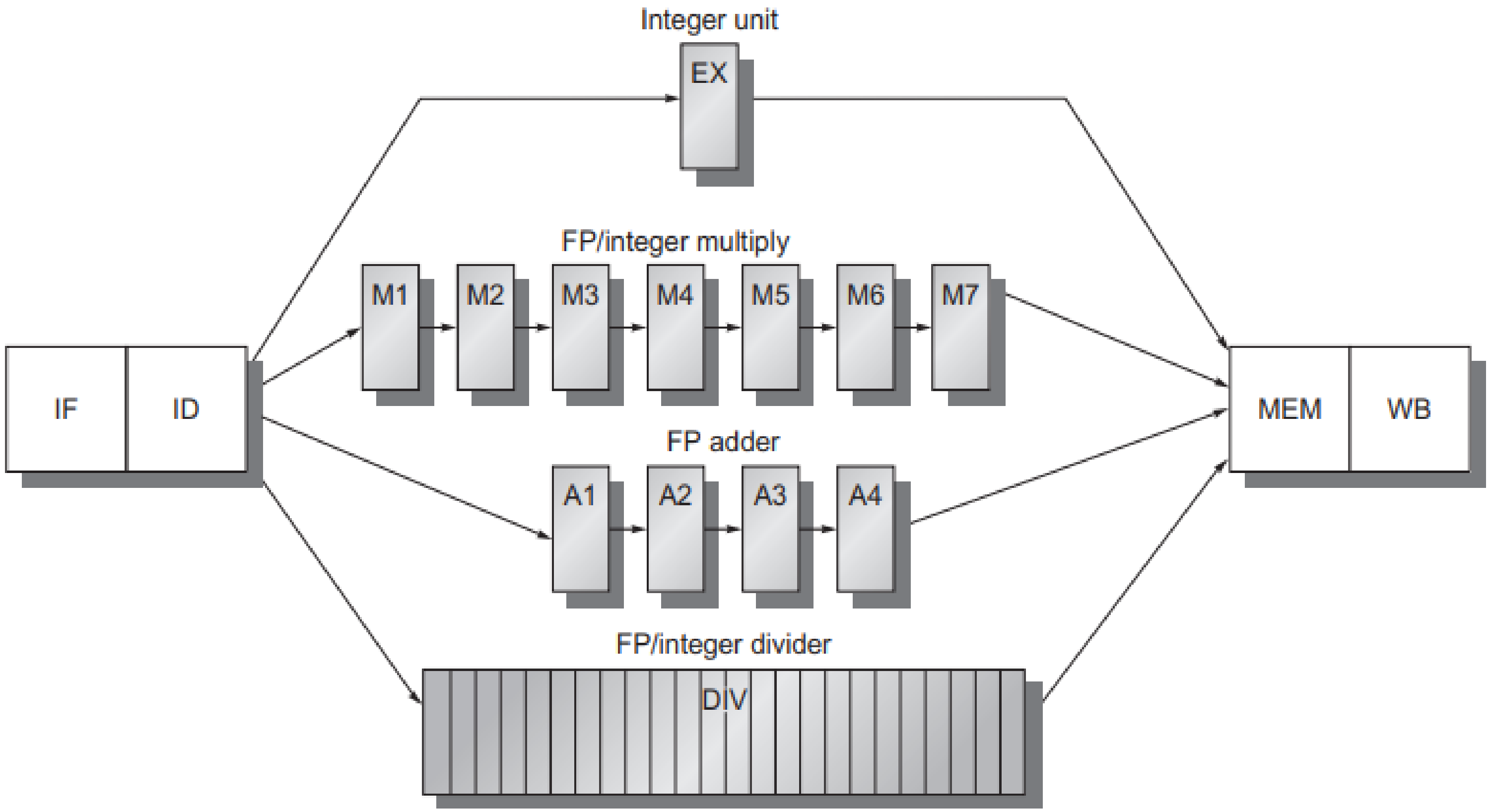
- Synchronous versus asynchronous
- User requested versus forced
- User maskable versus user nonmaskable
- Within versus between instructions
- Resume versus terminate

EXTENDING THE RISC V INTEGER PIPELINE TO HANDLE MULTICYCLE OPERATIONS

- We now want to explore how our RISC V pipeline can be extended to handle floating-point operations.

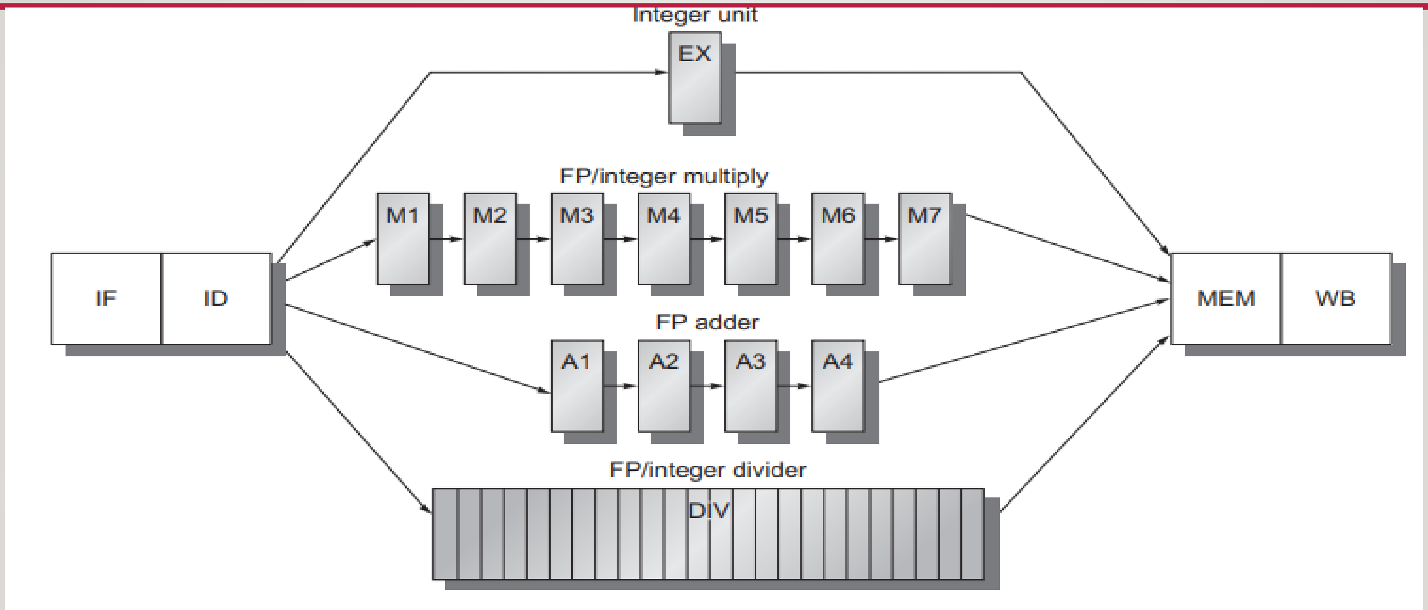


EXTENDING THE RISC V INTEGER PIPELINE TO HANDLE MULTICYCLE OPERATIONS



Functional unit	Latency	Initiation interval
Integer ALU	0	1
Data memory (integer and FP loads)	1	1
FP add	3	1
FP multiply (also integer multiply)	6	1
FP divide (also integer divide)	24	25

EXTENDING THE RISC V INTEGER PIPELINE TO HANDLE MULTICYCLE OPERATIONS



fmul.d	IF	ID	*M1*	M2	M3	M4	M5	M6	**M7**	MEM	WB
fadd.d		IF	ID	*A1*	A2	A3	**A4**	**MEM**	WB		
fld			IF	ID	*EX*	**MEM**	WB				
fsd				IF	ID	*EX*	**MEM**	WB			

HAZARDS AND FORWARDING IN LONGER LATENCY PIPELINES

- Because the divide unit is not fully pipelined, structural hazards can occur. These will need to be detected and issuing instructions will need to be stalled.
- Because the instructions have varying running times, the number of register writes required in a cycle can be larger than 1.
- Write after write (WAW) hazards are possible.
- Instructions can complete in a different order than they were issued, causing problems with exceptions ; ***Imprecise exception.***
- Because of longer latency of operations, stalls for RAW hazards will be more frequent.

HAZARDS AND FORWARDING IN LONGER LATENCY PIPELINES

	Clock cycle number																
Instruction	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**
`fld f4,0(x2)`	IF	ID	EX	MEM	WB												
`fmul.d f0,f4,f6`		IF	ID	Stall	M1	M2	M3	M4	M5	M6	M7	MEM	WB				
`fadd.d f2,f0,f8`			IF	Stall	ID	Stall	Stall	Stall	Stall	Stall	Stall	A1	A2	A3	A4	MEM	WB
`fsd f2,0(x2)`					IF	Stall	Stall	Stall	Stall	Stall	Stall	ID	EX	Stall	Stall	Stall	MEM

HAZARDS AND FORWARDING IN LONGER LATENCY PIPELINES

Instruction	Clock cycle number										
	1	2	3	4	5	6	7	8	9	10	11
`fmul.d f0,f4,f6`	IF	ID	M1	M2	M3	M4	M5	M6	M7	MEM	WB
...		IF	ID	EX	MEM	WB					
...			IF	ID	EX	MEM	WB				
`fadd.d f2,f4,f6`				IF	ID	A1	A2	A3	A4	MEM	WB
...					IF	ID	EX	MEM	WB		
...						IF	ID	EX	MEM	WB	
`fld f2,0(x2)`							IF	ID	EX	MEM	WB

CLASS ACTIVITY-I

- For the code sequence below, choose the statement that best describes requirements for correctness:

```
lw  t0,0(t0)
add t1,t0,t0
```

A No stalls as is
B No stalls with forwarding
C Must stall

CLASS ACTIVITY-II

- For the code sequence below, choose the statement that best describes requirements for correctness

```
add  t1, t0, t0
addi t2,t0,5
addi t4,t1,5
```

A No stalls as is

B No stalls with forwarding

C Must stall

CLASS ACTIVITY-III

- For the code sequence below, choose the statement that best describes requirements for correctness

```
addi t1,t0,1
addi t2,t0,2
addi t3,t0,2
addi t3,t0,4
addi t5,t1,5
```

A No stalls as is
B No stalls with forwarding
C Must stall

EXERCISE

```
sub   $t2, $t1, $t3
and   $t7, $t2, $t5
or    $t8, $t6, $t2
add   $t9, $t2, $t2
sw    $t5, 12($t2)
```

- If any dependencies exist where are they and what type are they?
- How many cycles does it take to execute the code fragment?

SOLUTION

	Clock								
	1	2	3	4	5	6	7	8	9
sub	IF	ID	EX	ME	WB				
and		IF	ID	EX	ME	WB			
or			IF	ID	EX	ME	WB		
and				IF	ID	EX	ME	WB	
sw					IF	ID	EX	ME	WB

EXERCISE

```
1. lw   x5, 0(x10)    # Load word from memory into x5
2. add  x6, x5, x7    # Add x5 and x7, result stored in x6
3. sw   x6, 4(x8)     # Store word from x6 into memory at address 4(x8)
```

- If any dependencies exist where are they and what type are they?
- How many cycles does it take to execute the code fragment?

EXERCISE

```
mul.d f0, f2, f4
addi  r1, r1, 1
add.d f6, f8, f10
```

How many cycles does it take to execute the code fragment? Draw Pipeline Diagram to support your answer

NUMERICAL

- The time delay of various segments in a 5 stage pipeline are t1=35 ns , t2= 30ns , t3= 40ns ,t4= 45 ns and t5= 35 ns . The interface register delay time is t= 5ns. How long would it take to complete 150 instructions in the pipeline? (Assuming all instructions are independent.

NUMERICAL

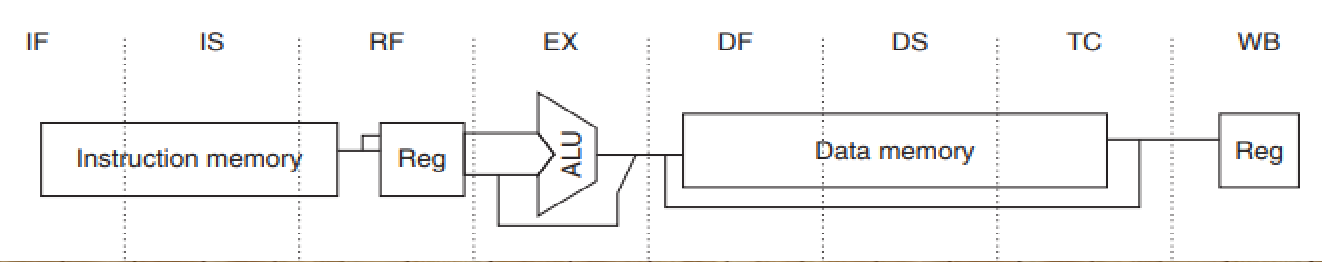
- Given a non-pipelined architecture , running at 1 GHz, that takes 5 cycles to complete an instruction . It was later converted to a 5 stage pipeline operating at 800 MHz. A stall of 70 cycles happens in 2% of memory instructions and a stall of 2 cycles happens in 20% of the branch instructions. 30 % instructions are of memory and 20% are of branches.
- What is actual speedup obtained by pipelining ?

THE MIPS R4000 PIPELINE

- The MIPS architecture and RISC V are very similar, differing only in a few instructions, including a delayed branch in the MIPS ISA.
- The R4000 is a 64 bit instruction set .
- However, it uses an 8-stage integer pipeline as opposed to the 5-stage pipeline.
- The extra stages are incorporated in to the instruction fetch and memory access stages.
- The strategy of using a deeper pipeline for speeding up memory access is often called **super pipelining.**
- Instruction and data memory are fully pipelined, so a new instruction can start on every clock cycle.

THE MIPS R4000 PIPELINE

- **The Pipeline Stages**
- IF : First half of instruction fetch; PC selection actually happens here, together with initiation of instruction cache access
- IS : Second half of instruction fetch, complete instruction cache access.
- RF : Instruction decode and register fetch, hazard checking, and also instruction cache hit detection
- EX : Execution, which includes effective address calculation, ALU operation, and branch target completion of data cache access
- DF : Data fetch, first half of data cache access
- DS : Second half of data fetch, completion of data cache access
- TC : Tag check, determine whether the data cache access hit
- WB : Write back for loads and register-register operation.



THE MIPS R4000 PIPELINE

- **Load Delays:**

Instruction	1	2	3	4	5	6	7	8	9	10	11
LW R1,(R2)	IF	IS	RF	EX	DF	DS	TC	WB			
ADD R3,R4,R1		IF	IS	RF	stall	stall	EX	DF	DS	TC	WB

	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
ld x1,...	IF	IS	RF	EX	DF	DS	TC	WB	
add x2,x1,...		IF	IS	RF	Stall	Stall	EX	DF	DS
sub x3,x1,...			IF	IS	Stall	Stall	RF	EX	DF
or x4,x1,...				IF	Stall	Stall	IS	RF	EX

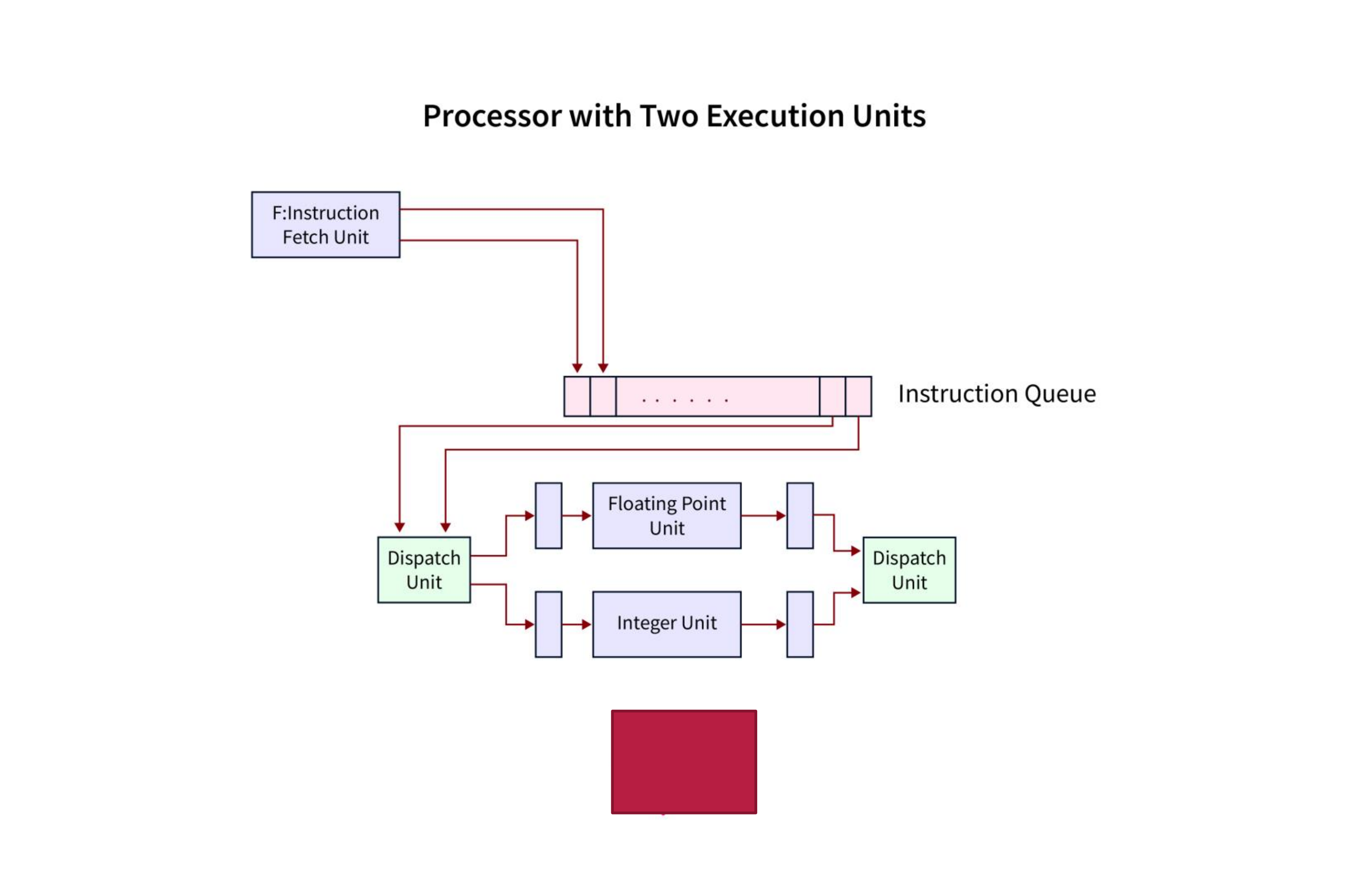
WHAT IS SUPERSCALAR PROCESSOR?

- A type of microprocessor that is used to implement a type of parallelism known as instruction-level parallelism in a single processor to execute more than one instruction during a CLK cycle by dispatching simultaneously various instructions to special execution units on the processor. A **scalar processor** executes single instruction for each clock cycle; a superscalar processor can execute more than one instruction during a clock cycle.

WHAT IS SUPERSCALAR PROCESSOR?

- Superscalar processing is the ability to initiate multiple instructions during the same clock cycle.
- A typical Superscalar processor fetches and decodes the incoming instruction stream several instructions at a time.
- Superscalar architecture exploit the potential of ILP(Instruction Level Parallelism).

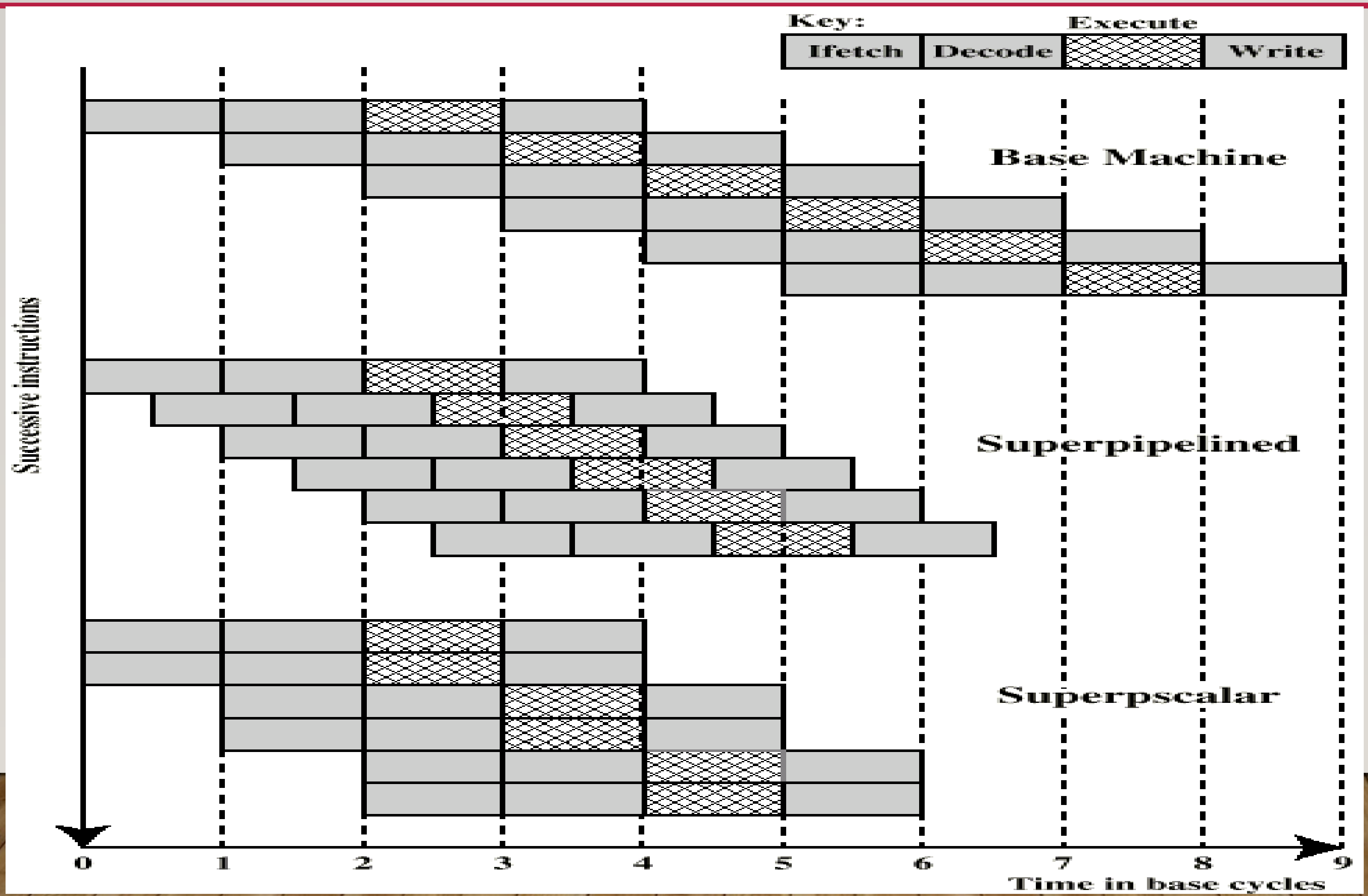
WHAT IS SUPERSCALAR PROCESSOR?



WHAT IS SUPERSCALAR PROCESSOR?

IF	ID	EX	MEM	WB				
IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB
				IF	ID	EX	MEM	WB

SUPER PIPELINE VS SUPER SCALAR



WHAT IS GOOD WITH SUPERSCALARS?

- **The hardware solves everything**
- Hardware detects potential parallelism between instructions.
- Hardware tries to issue as many instructions as possible in parallel.
- Hardware solves register renaming.

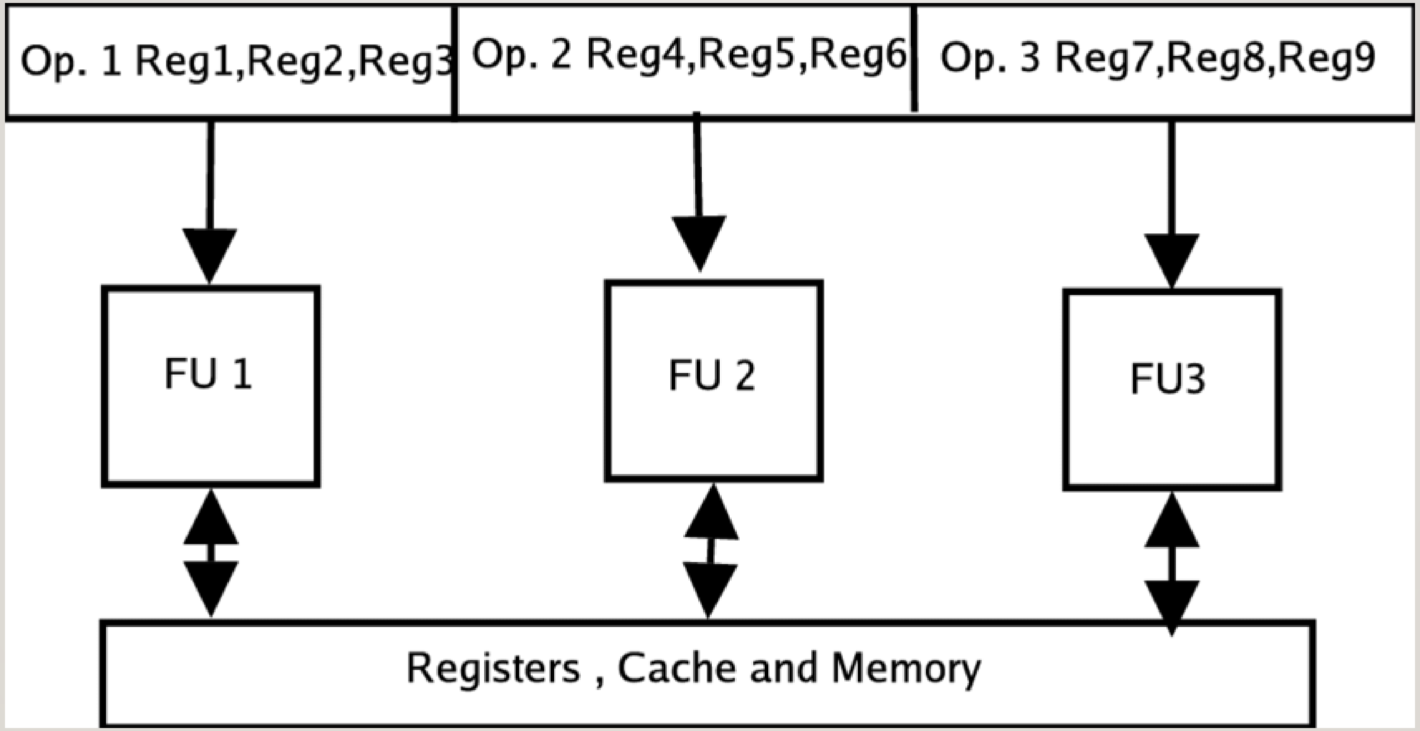
WHAT IS BAD WITH SUPERSCALARS?

- **Very complex**
- Much hardware is needed for run-time detection. There is a limit in how far we can go with this technique.
- Power consumption can be very large!
- **The instruction window is limited → this limits the capacity to detect potentially parallel instructions.**

VLIW (VERY LONG INSTRUCTION WORD) PROCESSORS

- In this style of architectures, the compiler formats a fixed number of operations as one big instruction (called a **bundle**) and schedules them.
- With few numbers of instructions, say 3, it is usually called **LIW** (Long Instruction Word).
- There is a change in the instruction set architecture, i.e., 1 program counter points to 1 bundle (not 1 operation).
- The operations in a bundle are issued in parallel.
- The bundles follow a fixed format and so the decode operations are done in parallel.

VLIW (VERY LONG INSTRUCTION WORD) PROCESSORS



RECALL FROM PIPELINING REVIEW

- **Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls**
- **Ideal pipeline CPI**: measure of the maximum performance attainable by the implementation
- **Structural hazards**: HW cannot support this combination of instructions
- **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
- **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

INSTRUCTION LEVEL PARALLELISM

- **Instruction-Level Parallelism (ILP):** overlap the execution of instructions to improve performance
- 2 approaches to exploit ILP:
- 1) Rely on hardware to help discover and exploit the parallelism **dynamically** (e.g., Pentium 4, AMD Opteron, IBM Power) , and
- 2) Rely on software technology to find parallelism, **statically** at compile-time (e.g., Itanium 2)

HAZARD VS DEPENDENCE

- **Dependence**: fixed property of instruction stream (i.e., program)
- **Hazard:** property of program and processor organization
- Definition: a hazard is created whenever there is a dependence between instructions, and they are close enough that the overlap during execution would change the order of access to the operand involved in the dependence.
- – implies potential for executing things in wrong order .potential only exists if instructions can be simultaneously "in-flight" (i.e. in the pipeline simultaneously)
- For example, can have RAW dependence with or without hazard – When distance between RAW instructions is larger than the pipeline depth

ASSUMPTION OF FP LATENCY

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

FP ALU1	IF	ID	F1	F2	F3	F4	MEM	WB			
FP ALU2		IF	ID	stall	stall	stall	F1	F2	F3	F4	MEM

FP ALU	IF	ID	F1	F2	F3	F4	MEM	WB
Store		IF	ID	EXE	stall	stall	MEM	WB

BASIC PIPELINE SCHEDULING AND LOOP UNROLLING

```
for (i=999; i>=0; i=i−1)
      x[i] = x[i] + s;
```

```
Loop:  fld     f0,0(x1)      //f0=array element
       fadd.d  f4,f0,f2      //add scalar in f2
       fsd     f4,0(x1)      //store result
       addi    x1,x1,−8      //decrement pointer
                             //8 bytes (per DW)
       bne     x1,x2,Loop    //branch x1≠x2
```

```
                                 Clock cycle issued
Loop:  fld     f0,0(x1)          1
       stall                     2
       fadd.d  f4,f0,f2          3
       stall                     4
       stall                     5
       fsd     f4,0(x1)          6
       addi    x1,x1,−8          7
       bne     x1,x2,Loop        8
```

```
Loop:  fld     f0,0(x1)
       addi    x1,x1,−8
       fadd.d  f4,f0,f2
       stall
       stall
       fsd     f4,8(x1)
       bne     x1,x2,Loop
```

LOOP UNROLLING

- Loop overhead (instructions that do book-keeping for the loop): **2**
- Actual work (the ld, add.d, and s.d): 3 instructions
- ***Can we somehow get execution time to be 3 cycles per iteration?***
- A simple scheme to increase the number of instructions relative to the branch overhead instructions.
- Replicates the loop body multiple times, adjusting the loop termination code
-

LOOP UNROLLING (STRAIGHTFORWARD WAY)

```
Loop:  fld     f0,0(x1)
       fadd.d  f4,f0,f2
       fsd     f4,0(x1)      //drop addi & bne
       fld     f6,-8(x1)
       fadd.d  f8,f6,f2
       fsd     f8,-8(x1)     //drop addi & bne
       fld     f0,-16(x1)
       fadd.d  f12,f0,f2
       fsd     f12,-16(x1)   //drop addi & bne
       fld     f14,-24(x1)
       fadd.d  f16,f14,f2
       fsd     f16,-24(x1)
       addi    x1,x1,-32
       bne     x1,x2,Loop
```

Eliminates 3 branches

Eliminates 3 decrements of X1

1 cycle stall for FLD
2 cycles stall for FADD

LOOP UNROLLING (SCHEDULING THAT MINIMIZES STALLS)

```
Loop:   fld     f0,0(x1)
        fld     f6,–8(x1)
        fld     f0,–16(x1)
        fld     f14,–24(x1)
        fadd.d  f4,f0,f2
        fadd.d  f8,f6,f2
        fadd.d  f12,f0,f2
        fadd.d  f16,f14,f2
        fsd     f4,0(x1)
        fsd     f8,–8(x1)
        fsd     f12,16(x1)
        fsd     f16,8(x1)
        addi    x1,x1,–32
        bne     x1,x2,Loop
```