Pipeline and parallelism

1. Pipeline

(1) Five steps:

The same principles apply to processors where we pipeline instruction-execution.

MIPS instructions classically take five steps:

Fetch instruction from memory.(**IF**)

Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.(**ID**)

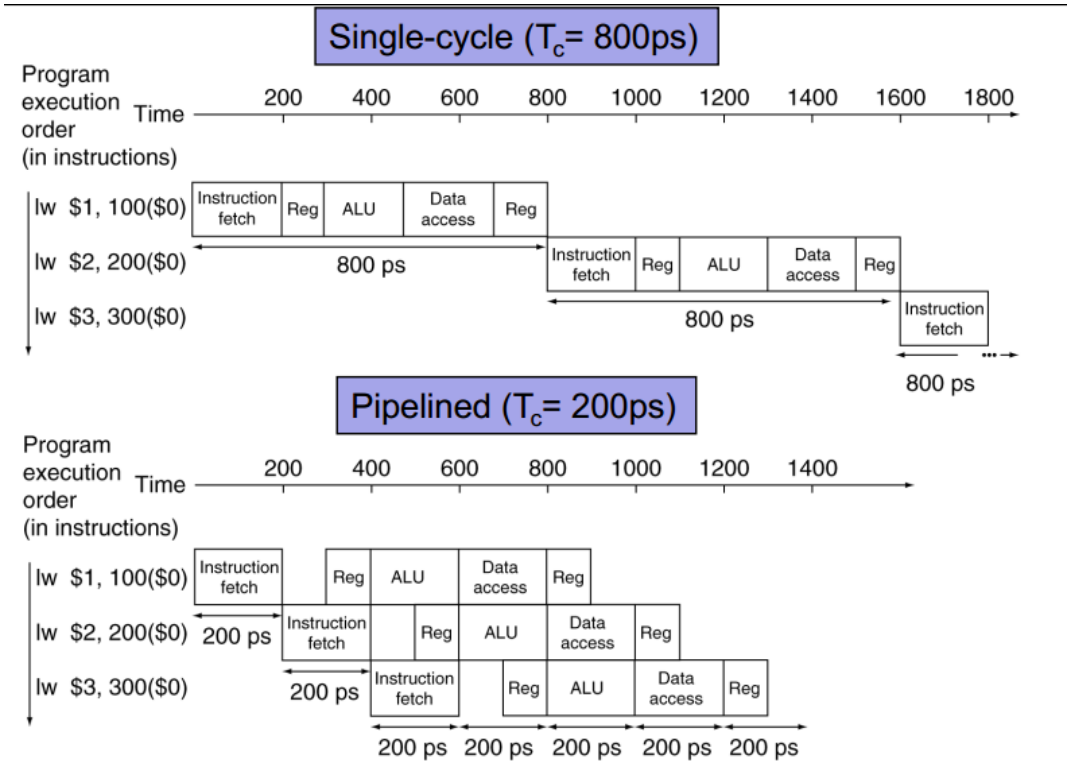
Execute the operation or calculate an address.(**EX**)

Access an operand in data memory.(**MEM**)

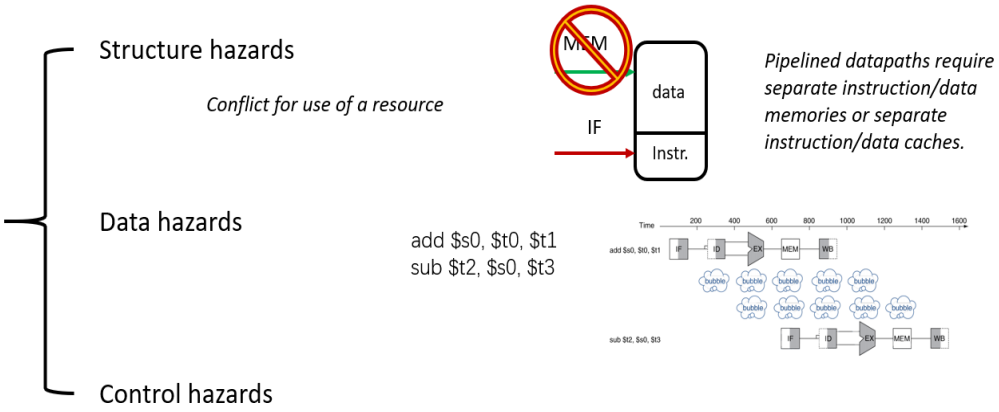
Write the result into a register.(**WB**)

(2) Longest delay determines clock period

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instruction}_{\text{nonpipelined}}}{\text{Number of pipe stages}}$$



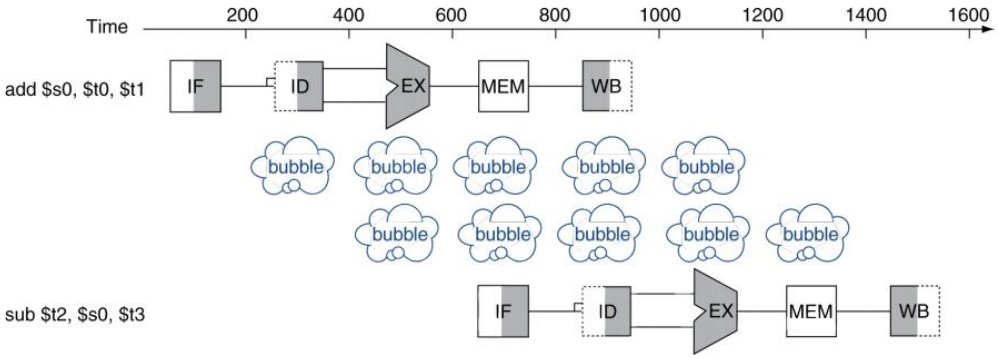
(3) Hazards



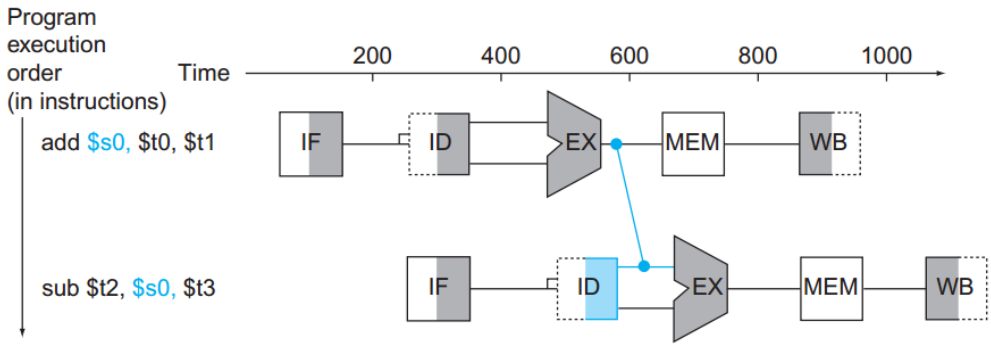
(4) Forwarding

Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.

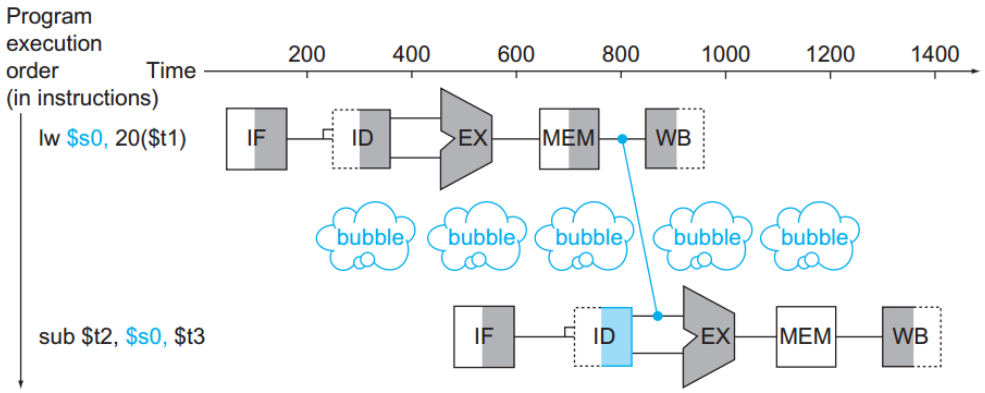
Without forwarding:



With forwarding:



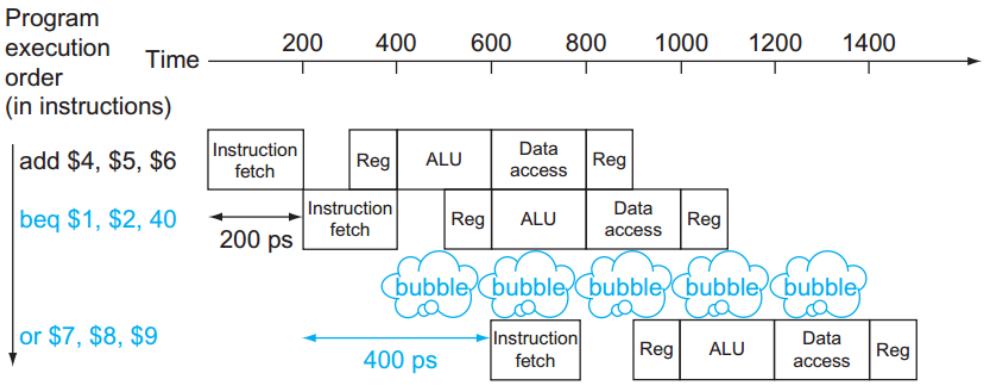
Notice: We need a stall even with forwarding when an R-format instruction following a load tries to use the data.



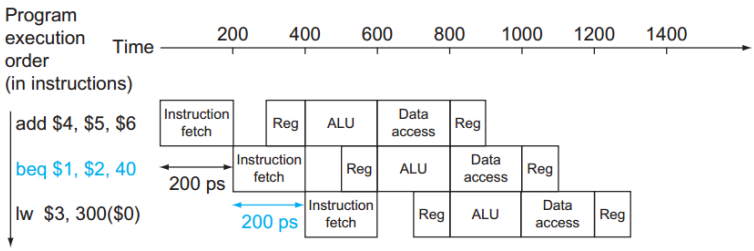
(5) Branch prediction

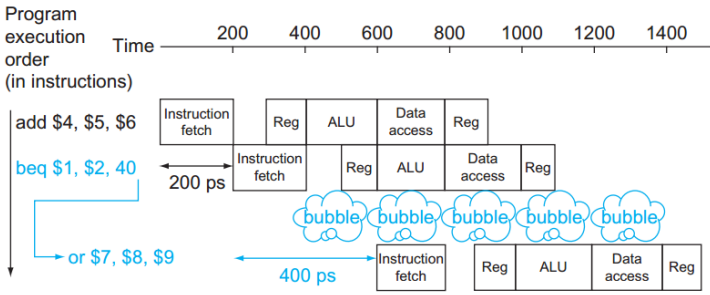
An instruction must be fetched at every clock cycle to sustain the pipeline, yet in our design the decision about whether to branch doesn't occur until the **MEM** pipeline stage.

Moving the branch execution to the **ID** stage is an improvement, because it reduces the penalty of a branch to only one instruction if the branch is taken, namely, the one currently being fetched. The exercises explore the details of implementing the forwarding path and detecting the hazard.



Predict-Not taken example:





Static branch prediction

Based on typical branch behavior

Dynamic branch prediction

Hardware measures actual branch behavior
• e.g., record recent history of each branch

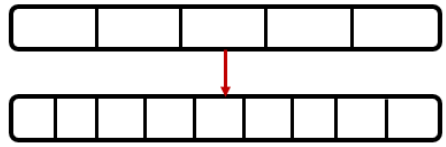
Assume future behavior will continue the trend
• When wrong, stall while re-fetching, and update history

2. Parallelism

Pipelining exploits the potential parallelism among instructions. This parallelism is called instruction-level parallelism (ILP). [指令级并行]

To increase ILP:

Deeper pipeline



Multiple issue (多发射)

(1) Multiple issue

Static multiple issue

Compiler *groups instructions into "issue packets"*

Determined by pipeline resources required

Packages them into "issue slots"

Compiler detects and avoids hazards

Dynamic multiple issue

CPU *examines instruction stream, chooses instructions to issue each cycle, and resolves hazards using advanced techniques at runtime*

Compiler can help by reordering instructions

(2) Static multiple issue

Instruction type	Pipe stages							
ALU or branch instruction	IF	ID	EX	MEM	WB			
Load or store instruction	IF	ID	EX	MEM	WB			
ALU or branch instruction		IF	ID	EX	MEM	WB		
Load or store instruction		IF	ID	EX	MEM	WB		
ALU or branch instruction			IF	ID	EX	MEM	WB	
Load or store instruction			IF	ID	EX	MEM	WB	
ALU or branch instruction				IF	ID	EX	MEM	WB
Load or store instruction				IF	ID	EX	MEM	WB

Loop unrolling example:

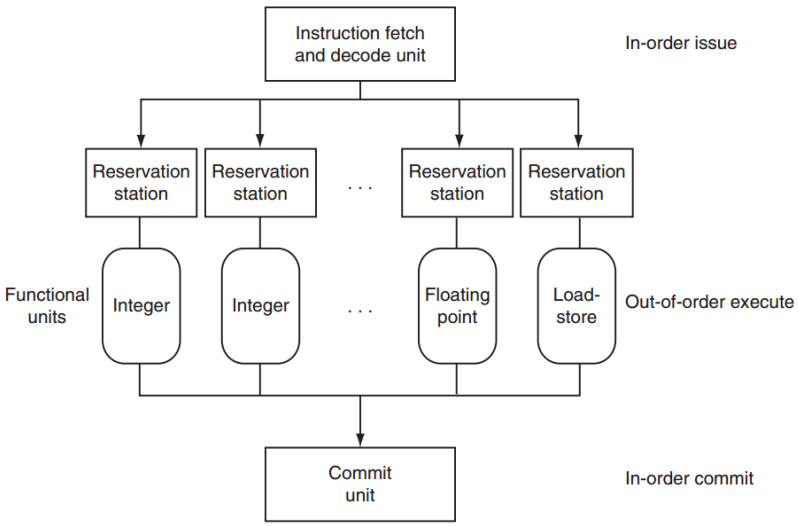
```
Loop: lw   $t0, 0($s1)      # $t0=array element
      addu $t0, $t0, $s2    # add scalar in $s2
      sw   $t0, 0($s1)      # store result
      addi $s1, $s1,-4      # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,-16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t3, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

Replicate loop body to expose more parallelism.

Use different registers per replication.

(3) Dynamic multiple issue



When an instruction issues, it is copied to a **reservation station** for the appropriate functional unit. Any **operands** that are **available** in the register file or reorder buffer are also immediately copied into the reservation station. The instruction is buffered in the reservation station **until all the operands and the functional unit are available**. For the issuing instruction, the register copy of the operand is no longer required, and if a write to that register occurred, the value could be overwritten.

If **an operand is not in the register file or reorder buffer**, **it must be waiting to be produced by a functional unit**. The name of the functional unit that will produce the result is tracked. When that unit eventually produces the result, it is copied directly into the waiting reservation station from the functional unit bypassing the registers.

3. Speculation

Speculation is an approach that allows the compiler or the processor to "guess" about the properties of an instruction, so as to enable execution to begin for other instructions that may depend on the speculated instruction.

Examples

- Speculate on branch outcome
  - Roll back if path taken is different
- Speculate on load
  - Roll back if location is updated

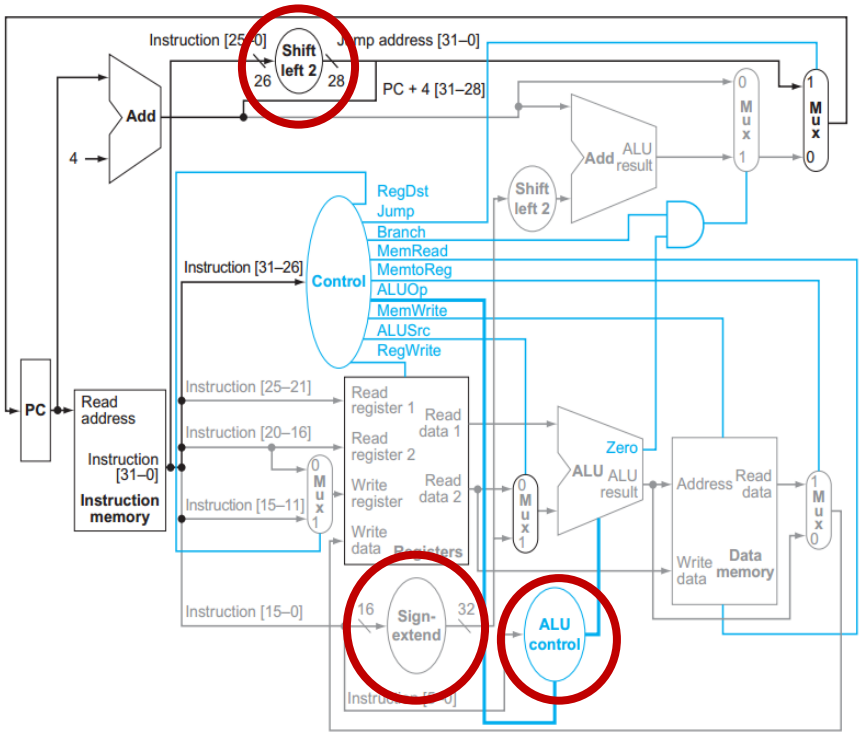
Function:

- Compiler can reorder instructions
  - e.g., move load before branch
  - Can include "fix-up" instructions to recover from incorrect guess
- Hardware can look ahead for instructions to execute
  - Buffer results until it determines they are actually needed
  - Flush buffers on incorrect speculation

4.7 In this exercise we examine in detail how an instruction is executed in a single-cycle datapath. Problems in this exercise refer to a clock cycle in which the processor fetches the following instruction word:
10101100011000100000000000010100
Assume that data memory is all zeros and that the processor's registers have the following values at the beginning of the cycle in which the above instruction word is fetched:

r0	r1	r2	r3	r4	r5	r6	r8	r12	r31
0	–1	2	–3	–4	10	6	8	2	–16

$$101011_{binary} = 43 = 2b_{hex} \rightarrow sw$$
$$00011_{binary} = 3 \rightarrow rs = \$v1$$
$$00010_{binary} = 2 \rightarrow rt = \$v0$$
$$0000000000010100_{binary} = 20$$
$$sw\ \$v0, 20(\$v1)$$



4.7.1 What are the outputs of the sign-extend and the jump "Shift left 2" unit (near the top of Figure 4.24) for this instruction word?

The outputs of the sign-extend:
00000000000000000000000000010100

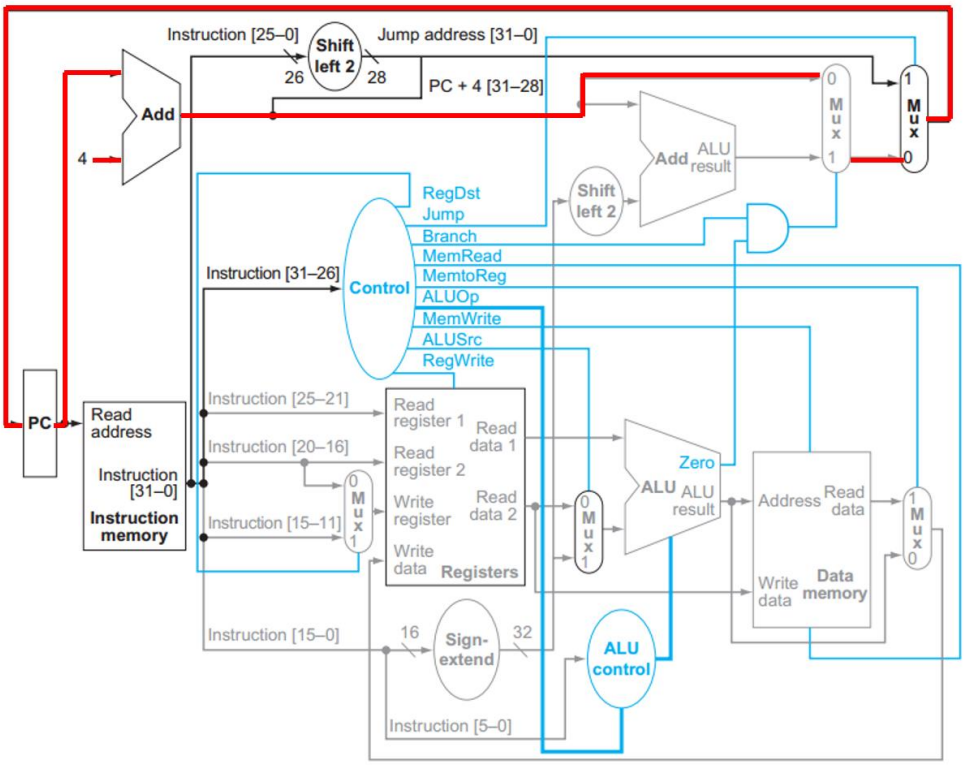
The outputs of the jump "Shift left 2" unit:
00011000100000000000001010000

4.7.2 What are the values of the ALU control unit's inputs for this instruction?

ALUOp = 00; Instruction = 010100

4.7.3 What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.

new PC address = PC+4;

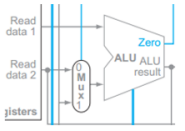


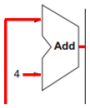
4.7.4 For each Mux, show the values of its data output during the execution of this instruction and these register values.

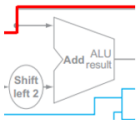
(1) Write Register MUX would have control of DC because we are storing into memory so not touching register file. Therefore, we don't care what comes out of the MUX because we won't be using the write register.
(2) ALU MUX would have control of 1 because we want to use the immediate, not the second register for the ALU. Output of sign-extend [: 0000000000000000 (16) 0000000000010100 [15-0] This would be our output for the ALU MUX -> 20
(3) The ALU/Mem MUX would have control of DC because we won't be writing anything to the register file, so we don't care what goes into the write data port. Therefore, we don't care what comes out of the MUX.
(4) From the previous parts, we said that the Branch and Jump MUX will both output PC+4.

Write Reg Mux	ALU Mux	ALU/Mem Mux	Branch Mux	Jump
2 or 0	20	X	PC+4	PC+4

4.7.5 For the ALU and the two add units, what are their data input values?

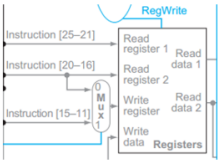






ALU	Add(PC+4)	Add (Branch)
-3 and 20	PC and 4	PC+4 and 80

4.7.6 What are the values of all inputs for the “Registers” unit?



Read Register1	Read Regster2	Write Register	Write Data	RegWrite
3	2	2 or 0 (X)	X	0

4.10 In this exercise, we examine how resource hazards, control hazards, and Instruction Set Architecture (ISA) design can affect pipelined execution. Problems in this exercise refer to the following fragment of MIPS code:

```
sw r16,12(r6)
lw r16,8(r6)
beq r5,r4,Label   # Assume r5!=r4
add r5,r1,r4
slt r5,r15,r4
```

Assume that individual pipeline stages have the following latencies:

IF	ID	EX	MEM	WB
200ps	120ps	150ps	190ps	100ps

4.10.1 For this problem, assume that all branches are perfectly predicted (this eliminates all control hazards) and that no delay slots are used. If we only have one memory (for both instructions and data), there is a structural hazard every time we need to fetch an instruction in the same cycle in which another instruction accesses data. To guarantee forward progress, this hazard must always be resolved in favor of the instruction that accesses data. What is the total execution time of this instruction sequence in the 5-stage pipeline that only has one memory? We have seen that data hazards can be eliminated by adding nops to the code. Can you do the same with this structural hazard? Why?

Instruction	Pipeline Stage											Cycles
SW R16, 12(R6)	IF	ID	EX	MEM	WB							
LW R16 8(R6)		IF	ID	EX	MEM	WB						
BEQ R5,R4,Lb1			IF	ID	EX	MEM	WB					11
ADD R5,R1,R4				***	***	IF	ID	EX	MEM	WB		
SLT R5,R15,R4							IF	ID	EX	MEM	WB	

$clock\ cycle\ time = 200\ ps$ ⟶ $Total\ Execution\ Time = 11 \times 200\ ps = 2200\ ps$

We cannot add NOPs to the code to eliminate this hazard – NOPs need to be fetched just like any other instructions, so this hazard must be addressed with a hardware hazard detection unit in the processor.

4.10.2 For this problem, assume that all branches are perfectly predicted (this eliminates all control hazards) and that no delay slots are used. If we change load/store instructions to use a register (without an offset) as the address, these instructions no longer need to use the ALU. As a result, MEM and EX stages can be overlapped and the pipeline has only 4 stages. Change this code to accommodate this changed ISA. Assuming this change does not affect clock cycle time, what speedup is achieved in this instruction sequence?

Instruction	Pipeline Stage									Cycles
SW R16, 12(R6)	IF	ID	EX	MEM	WB					
LW R16 8(R6)		IF	ID	EX	MEM	WB				
BEQ R5,R4,Lb1			IF	ID	EX	MEM	WB			9
ADD R5,R1,R4				IF	ID	EX	MEM	WB		
SLT R5,R15,R4					IF	ID	EX	MEM	WB	

Instruction	Pipeline Stage								Cycles
SW R16, 12(R6)	IF	ID	E/M	WB					
LW R16 8(R6)		IF	ID	E/M	WB				
BEQ R5,R4,Lb1			IF	ID	E/M	WB			8
ADD R5,R1,R4				IF	ID	E/M	WB		
SLT R5,R15,R4					IF	ID	E/M	WB	

$Speedup = 9/8 = 1.125$

4.10.3 Assuming stall-on-branch and no delay slots, what speedup is achieved on this code if branch outcomes are determined in the ID stage, relative to the execution where branch outcomes are determined in the EX stage?

Instruction	Pipeline Stage											Cycles
SW R16, 12(R6)	IF	ID	EX	MEM	WB							
LW R16 8(R6)		IF	ID	EX	MEM	WB						
BEQ R5,R4,Lb1			IF	ID	EX	MEM	WB					11
ADD R5,R1,R4				***	***	IF	ID	EX	MEM	WB		
SLT R5,R15,R4							IF	ID	EX	MEM	WB	

Instruction	Pipeline Stage									Cycles
SW R16, 12(R6)	IF	ID	EX	MEM	WB					
LW R16 8(R6)		IF	ID	EX	MEM	WB				
BEQ R5,R4,Lb1			IF	ID	EX	MEM	WB			10
ADD R5,R1,R4				***	IF	ID	EX	MEM	WB	
SLT R5,R15,R4						IF	ID	EX	MEM	WB

$$Speedup = 11/10 = 1.10$$

4.10.4 Given these pipeline stage latencies, repeat the speedup calculation from 4.10.2, but take into account the (possible) change in clock cycle time. When EX and MEM are done in a single stage, most of their work can be done in parallel. As a result, the resulting EX/MEM stage has a latency that is the larger of the original two, plus 20 ps needed for the work that could not be done in parallel.

Instruction	Pipeline Stage									Cycles
SW R16, 12(R6)	IF	ID	EX	MEM	WB					
LW R16 8(R6)		IF	ID	EX	MEM	WB				
BEQ R5,R4,Lb1			IF	ID	EX	MEM	WB			9
ADD R5,R1,R4				IF	ID	EX	MEM	WB		
SLT R5,R15,R4					IF	ID	EX	MEM	WB	

$$Cycle\ time\ with\ 5\ stages: 200\ ps(IF)$$

Instruction	Pipeline Stage								Cycles
SW R16, 12(R6)	IF	ID	E/M	WB					
LW R16 8(R6)		IF	ID	E/M	WB				
BEQ R5,R4,Lb1			IF	ID	E/M	WB			8
ADD R5,R1,R4				IF	ID	E/M	WB		
SLT R5,R15,R4					IF	ID	E/M	WB	

$$Cycle\ time\ with\ 4\ stages: 210\ ps(MEM + 20\ ps)$$

$$Speedup = (9 \times 200)/(8 \times 210) = 1.07$$

4.10.5 Given these pipeline stage latencies, repeat the speedup calculation from 4.10.3, taking into account the (possible) change in clock cycle time. Assume that the latency ID stage increases by 50% and the latency of the EX stage decreases by 10ps when branch outcome resolution is moved from EX to ID.

Instruction	Pipeline Stage											Cycles
SW R16, 12(R6)	IF	ID	EX	MEM	WB							
LW R16 8(R6)		IF	ID	EX	MEM	WB						
BEQ R5,R4,Lb1			IF	ID	EX	MEM	WB					11
ADD R5,R1,R4				***	***	IF	ID	EX	MEM	WB		
SLT R5,R15,R4							IF	ID	EX	MEM	WB	

$$Cycle\ time\ with\ branch\ in\ EX: 200\ ps(IF)$$

Instruction	Pipeline Stage									Cycles
SW R16, 12(R6)	IF	ID	EX	MEM	WB					
LW R16 8(R6)		IF	ID	EX	MEM	WB				
BEQ R5,R4,Lb1			IF	ID	EX	MEM	WB			10
ADD R5,R1,R4				***	IF	ID	EX	MEM	WB	
SLT R5,R15,R4						IF	ID	EX	MEM	WB

$$New\ EX\ latency: 140\ ps \qquad New\ ID\ latency: 180\ ps \qquad Cycle\ time\ with\ branch\ in\ ID: 200\ ps(IF)$$

$$Speedup = (11 \times 200)/(10 \times 200) = 1.10$$

4.10.6 Assuming stall-on-branch and no delay slots, what is the new clock cycle time and execution time of this instruction sequence if beq address computation is moved to the MEM stage? What is the speedup from this change? Assume that the latency of the EX stage is reduced by 20 ps and the latency of the MEM stage is unchanged when branch outcome resolution is moved from EX to MEM.

Instruction	Pipeline Stage											Cycles
SW R16, 12(R6)	IF	ID	EX	MEM	WB							
LW R16 8(R6)		IF	ID	EX	MEM	WB						
BEQ R5,R4,Lb1			IF	ID	EX	MEM	WB					11
ADD R5,R1,R4				***	***	IF	ID	EX	MEM	WB		
SLT R5,R15,R4							IF	ID	EX	MEM	WB	

Cycle time with branch in EX: 200 $ps(IF)$

Instruction	Pipeline Stage											Cycles
SW R16, 12(R6)	IF	ID	EX	MEM	WB							
LW R16 8(R6)		IF	ID	EX	MEM	WB						
BEQ R5,R4,Lb1			IF	ID	EX	MEM	WB					12
ADD R5,R1,R4				***	***	***	IF	ID	EX	MEM	WB	
SLT R5,R15,R4								IF	ID	EX	MEM	WB

New EX latency: 130 ps *Cycle time with branch in MEM*: 200 $ps(IF)$

New clock cycle time = 200 ps *New execution time* = $12 \times 200\ ps = 2400\ ps$

Speedup = $(11 \times 200)/(12 \times 200) = 0.92$