Computer Organization and Design

Homework 5

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

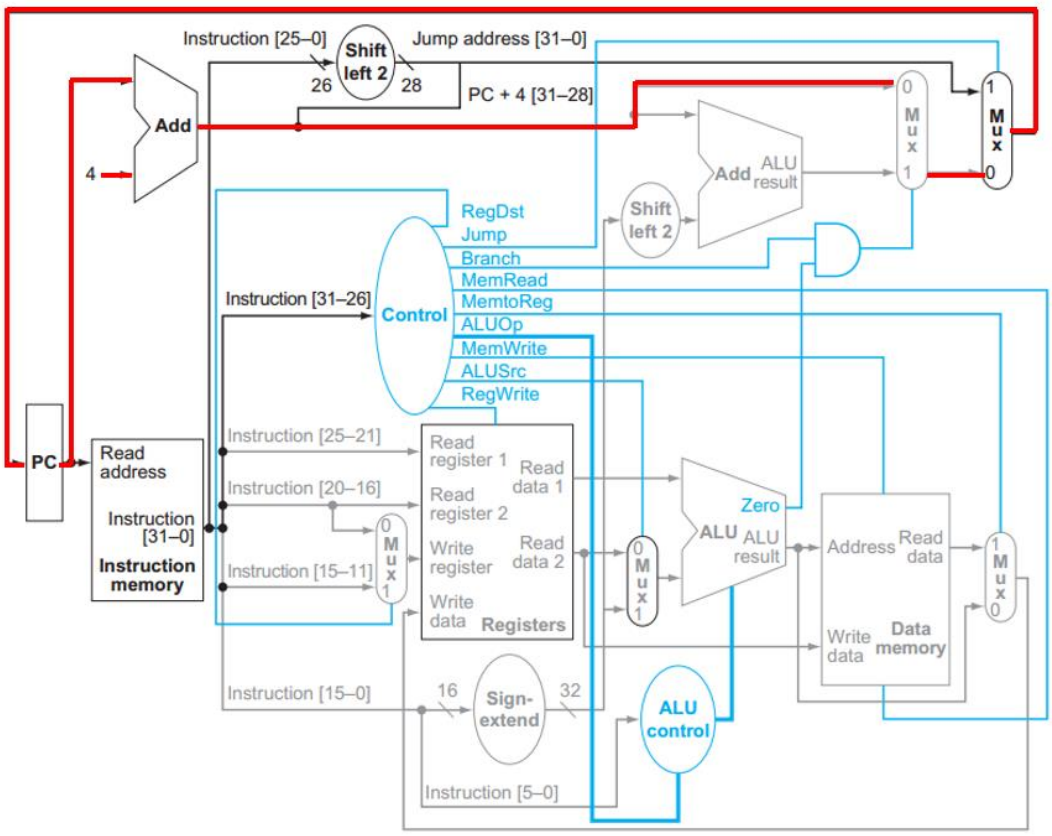
4.7.1 What are the outputs of the sign-extend and the jump "Shift left 2" unit (near the top of Figure 4.24) for this instruction word?
4.7.2 What are the values of the ALU control unit's inputs for this instruction?
4.7.3 What is the new PC address after this instruction is executed? Highlight the path through which this value is determined.
4.7.4 For each Mux, show the values of its data output during the execution of this instruction and these register values.
4.7.5 For the ALU and the two add units, what are their data input values?
4.7.6 What are the values of all inputs for the "Registers" unit?
Solution:

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

4.7.1 The outputs of the sign-extend:
00000000000000000000000000010100
The outputs of the jump "Shift left 2" unit:
00011000100000000000001010000

4.7.2
ALUOp = 00;
Instruction = 010100

4.7.3 new PC address = PC+4;
Highlight the path:



4.7.4

(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

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

4.7.6

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?

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?

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?

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.

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.
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.

Solution:
4.10.1 In the pipelined execution shown below, *** represents a stall when an instruction cannot be fetched because a load or store instruction is using the memory in that cycle. Cycles are represented from left to right, and for each instruction we show the pipeline stage it is in during that cycle:

Instruction	Pipeline Stage	Cycles

SW R16,12(R6)
LW R16,8(R6)
BEQ R5,R4,Lbl
ADD R5,R1,R4
SLT R5,R15,R4
	IF ID EX MEM WB
	IF ED EX MEM WB
	IF ID EX MEM WB
	*** *** IF ID EX MEM WB
	IF ID EX MEM WB
		11

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 This change only saves one cycle in an entire execution without data hazards (such as the one given). Tis cycle is saved because the last instruction finishes one cycle earlier (one less stage to go through). If there were data hazards from loads to other instructions, the change would help eliminate some stall cycles.

Instructions Executed	Cycles with 5 stages	Cycles with 4 stages	Speedup
5	4 + 5 = 9	3 + 5 = 8	9/8 = 1.13

4.10.3 Stall-on-branch delays the fetch of the next instruction until the branch is executed. When branches execute in the EXE stage, each branch causes two stall cycles. When branches execute in the ID stage, each branch only causes one stall cycle. Without branch stalls (e.g., with perfect branch prediction) there are no stalls, and the execution time is 4 plus the number of executed instructions. We have

Instructions Executed	Branches Executed	Cycles with branch in EXE	Cycles with branch in ID	Speedup
5	1	4 + 5 + 1*2 = 11	4 + 5 + 1*1 = 10	11/10 = 1.10

4.10.4 The number of cycles for the (normal) 5-stage and the (combined EX/MEM) 4-stage pipeline is already computed in 4.10.2. The clock cycle time is equal to the latency of the longest-latency stage. Combining EX and MEM stages affects clock time only if

the combined EX/MEM stage becomes the longest-latency stage:

Cycle time with 5 stages	Cycle time with 4 stages	Speedup
200 ps (IF)	210 ps (MEM + 20 ps)	(9*200)/(8*210) = 1.07

4.10.5

New ID latency	New EX latency	New cycle time	Old cycle time	Speedup
180 ps	140 ps	200 ps (IF)	200 ps (IF)	(11*200)/(10*200) = 1.10

4.10.6

The cycle time remains unchanged: a 20 ps reduction in EX latency has no effect on clock cycle time because EX is not the longest-latency stage. The change does affect execution time because it adds one additional stall cycle to each branch. Because the clock cycle time does not improve but the number of cycles increases, the speedup from this change will be below 1 (a slowdown). In 4.10.3 we already computed the number of cycles when branch is in EX stage. We have:

	Cycles with branch in EX	Execution time (branch in EX)	Cycles with branch in MEM	Execution time (branch in MEM)	Speedup
a.	4 + 5 + 1*2 = 11	11*200 ps = 2200 ps	4 + 5 + 1*3 = 12	12*200 ps = 2400 ps	0.92