CS 2340 - Homework 4

Diego Rodrigues

April 20, 2024

Question 1

(a)
- RegWrite = 1
- ALUSrc = 0
- MemRead = 0
- MemWrite = 0
- MemtoReg = 0
- ALUOp = 00

(b)
- Instruction memory
- Registers
- ALU
- Control unit

Question 2

(a) 25% (from load) + 10% (from store) = 35%

(b) 100%

(c) 28% (from I-type) + 25% (from load) = 53%

(d) The sign extend unit does not have 'off' cycles - it's always producing output whenever it has input, and it is up to the control logic to determine whether or not that output is utilized.

Question 3

(a) This would affect load instruction because it expect to write data from memory into a register.

(b) The ALU will always use the value from the second register for computations and never an immediate value. This would affect immediate instructions (e.g., addi, andi, ori, xori, lw, sw)

Question 4

(a) From the diagram and the description, it seems that most of the necessary components for supporting I-type instructions are already in place. The sign-extend unit is there and so is the ALU input multiplexor controlled by the ALUSrc signal.

(b)
- RegDst = 0, since the destination register is specified in the rt field of the instruction, and RegDst is irrelevant as we do not choose between rd and rt for I-type instructions.
- MemToReg = 0, since we want to write the ALU result to the register, not data from memory.
- MemRead = 0, as memory is not read during addi.
- MemWrite = 0, since we are not writing to memory.
- Branch = 0, as this is not a branch instruction.

Question 5

(a) R-format Instruction

(a) Register read 30ps
(b) Instruction Mem 250ps
(c) Mux 25ps
(d) Register Setup 30ps
(e) Register File 150ps
(f) Mux 25ps
(g) ALU 200ps
(h) Mux 25ps
(i) Register Setup 30ps (for writing back)
(j) Register File 150ps
(k) total = 965ps

(b) lw Instruction

(a) Register read 30ps
(b) Instruction Mem 250ps
(c) Mux 25ps
(d) Register Setup 30ps
(e) Register File 150ps
(f) ALU 200ps
(g) Memory 250ps

(h) Mux 25ps
(i) Register Setup 30ps (for writing back)
(j) Register File 150ps
(k) total = 1140ps

(c) sw Instruction

(a) Register read 30ps
(b) Instruction Mem 250ps
(c) Mux 25ps
(d) Register Setup 30ps
(e) Register File 150ps
(f) Mux 25ps
(g) ALU 200ps
(h) Memory 250ps
(i) total = 960ps

(d) beq Instruction

(a) Register read 30ps
(b) Instruction Mem 250ps
(c) Register Setup 30ps
(d) Register File 150ps
(e) Mux 25ps
(f) ALU 200ps
(g) Single Gate 5ps
(h) Mux 25ps
(i) Register read 30ps (PC)
(j) total = 745ps

(e) i-format Instruction

(a) Register read 30ps
(b) Instruction Mem 250ps
(c) Register Setup 30ps
(d) Register File 150ps
(e) Mux 25ps
(f) ALU 200ps
(g) Mux 25ps

(h) Register Setup 30ps (for writing back)

(i) Register File 150ps

(j) total = 890ps

(f) max of all = 1140ps

Question 6

(a) For a non-pipelined processor, the clock cycle time is the sum of all the stage latencies since each instruction must complete before the next one begins: $250 + 350 + 150 + 300 + 200 = 1250$ ps.

For a pipelined processor, the clock cycle time is determined by the longest pipeline stage, which is the Instruction Decode (ID) stage: 350 ps.

(b) In a non-pipelined processor, the total latency for a single instruction is the same as the clock cycle time: 1250 ps.

In a pipelined processor, the latency for an instruction is the sum of the latencies of all stages it must pass through. However, because instructions are overlapped, we consider only the longest path for the clock cycle time. The total latency for the lw instruction in a pipeline would be the clock cycle time multiplied by the number of stages since each stage is one clock cycle: $350p \times s5stages = 1750$.

(c) You would split the longest stage to improve the overall clock cycle time. The longest stage is the ID stage at 350 ps. If we split it into two stages of 175 ps each, the new longest stage (if no other changes are made) would be the MEM stage at 300 ps.

(d) The new clock cycle time would then be 300 ps.

(e) Utilization is calculated by considering how often the resource is used. For the data memory (MEM), which is used by lw and sw instructions, which account for 20% and 15% of the instructions respectively, the total utilization would be $20\% + 15\% = 35$.

(f) The write-register port is used by any instruction that writes back to a register. This includes all R-type instructions (alu), lw instructions, and the result of any I-type instructions that are not stores. In the given instruction mix, this includes 45% alu, 20% lw, and does not include sw or beq (since they don't write to a register), so the total utilization is $45\% + 20\% = 65\%$.

Question 7 Assuming that registers $\$s0$ and $\$s1$ are initialized to 11 and 22 respectively, and the pipeline does not handle data hazards, we have the following instructions:

```
addi $s0, $s1, 5
add  $s2, $s0, $s1
addi $s3, $s0, 15
```

The final values of the registers $\$s2$ and $\$s3$ will be computed as follows:

(a) The instruction `addi $s0, $s1, 5` will attempt to add 5 to the value of $\$s1$ (22) and store the result in $\$s0$. However, because the pipeline does not handle data hazards, the original value of $\$s0$ (11) is used in the subsequent instruction.

(b) The instruction `add $s2, $s0, $s1` will add the value of $\$s0$ (11) and $\$s1$ (22), resulting in 33, which will be stored in $\$s2$.

(c) The instruction `addi $s3, $s0, 15` will again use the original value of $\$s0$ (11) and add 15 to it, resulting in 26, which will be stored in $\$s3$.

So, the final values will be:

$$\$s2 = 33$$

$$\$s3 = 26$$

Question 8

- The `lw` instruction can use forwarding to resolve the dependency on the `add` instruction for register $12.
- The `sub` instruction has to stall because the loaded value in $15 is not available until after the MEM stage of the `lw` instruction.
- The second `add` instruction can use forwarding for the value in register $13 from the `sub` instruction.

The pipeline diagram with stalls:

Cycle	IF	ID	EX	MEM	WB
1	add				
2	lw	add			
3	stall	lw	add		
4	stall	stall	lw	add	
5	sub	stall	stall	lw	add
6	add	sub	stall	stall	lw
7		add	sub	stall	stall
8			add	sub	
9				add	sub

Question 9 To minimize the performance by introducing the maximum number of stalls:

```
lw  $3, 0($5)
add $7, $7, $3   // Stall: $3 is not ready
lw  $4, 4($5)    // Stall: previous add is still executing
add $8, $8, $4   // Stall: $4 is not ready
add $10, $7, $8  // Stall: both adds are just executed
sw  $6, 0($5)    // Stall: add to $10 is still executing
beq $10, $11, loop // Stall: sw just executed
```

The reordered sequence introduces data hazards between consecutive instructions that cannot be resolved without stalling, due to the absence of forwarding hardware.

Question 10 (BONUS)

```
Instruction      IF    ID    EX    MEM   WB
sw $s5, 12($s3)  1     2     3     4     5
lw $s5, 8($s3)   stall 6     7     8     9
sub $s4, $s2, $s1      stall 10    11    12
bez $s4, label               stall stall 13
add $s2, $s0, $s1                  stall 14
sub $s2, $s6, $s1                        15
```

In general, it is not possible to completely avoid stalls resulting from a structural hazard just by reordering code, since the hazard arises from hardware limitations rather than instruction order. However, certain reordering can minimize the impact by aligning memory access instructions with those that do not require memory access, thus optimizing the pipeline flow as much as possible.