Sheet 4 Solution

Exercise 4-1:

For the following instruction sequences:

a.
```
lw   $1, 40($2)
add  $2, $3, $3
add  $1, $1, $2
sw   $1, 20($2)
```

b.
```
add  $1, $2, $3
sw   $2, 0($1)
lw   $1, 4($2)
add  $2, $2, $1
```

Assume that before any of the instructions is executed, all values in data memory are 0s and that registers $0 through $3 have the initial values shown in the table below. Which value is the first one to be forwarded and what is the value it overrides? Assume a hazard detection unit that assumes forwarding is implemented, but the forwarding mechanism itself is not, what will be the final register values after this instruction sequence? Add nops to this instruction sequence to ensure correct execution in spite of the missing support for forwarding.

	$0	**$1**	**$2**	**$3**
For instruction sequence a	0	1	31	1000
For instruction sequence b	0	-2	63	2500

Solution:

The following values will be forwarding:

	Instruction sequence	**First values to be forwarded**
a.	I1: lw $1, 40($2) I2: add $2, $3, $3 I3: add $1, $1, $2 I4: sw $1, 20($2)	($1) I1 to I3 (0 overrides 1) ($2) I2 to I3 (2000 overrides 31)
b.	I1: add $1, $2, $3 I2: sw $2, 0($1) I3: lw $1, 4($2) I4: add $2, $2, $1	($1) I1 to I2 (2563 overrides -2)

A register modification becomes “visible” to the EX stage of the following instructions only two cycles after the instruction that produces the register value leaves the EX stage. The forwarding-assuming hazard detection unit only adds a one-cycle stall if the instruction that immediately follows a load is dependent on the load. Thus, the final register values will be as follows:

	Instruction sequence with forwarding stalls	Execution without forwarding	Values after execution
a.	I1: lw $1, 40($2) I2: add $2, $3, $3 I3: add $1, $1, $2 I4: sw $1, 20($2)	$1=0 (Visible at I4 and after) $2=2000 (after I4) $1=32 (after I4)	$0=0 $1=32 $2=2000 $3=1000
b.	I1: add $1, $2, $3 I2: sw $2, 0($1) I3: lw $1, 4($2) Stall I4: add $2, $2, $1	$1=2563 (Visible at Stall and after) $1=0 (after I4) $2=2626 (after I4)	$0=0 $1=0 $2=2626 $3=2500

The Instruction execution with NOPs should be as follows:

	Instruction sequence with forwarding stalls	Correct execution Sequence with NOPs

a.	I1: lw $1, 40($2)
	I2: add $2, $3, $3
	I3: add $1, $1, $2
	I4: sw $1, 20($2)
		lw $1, 40($2)
		add $2, $3, $3
		nop
		nop
		add $1, $1, $2
		nop
		nop
		sw $1, 20($2)

b.	I1: add $1, $2, $3
	I2: sw $2, 0($1)
	I3: lw $1, 4($2)
	Stall
	I4: add $2, $2, $1
		add $1, $2, $3
		nop
		nop
		sw $2, 0($1)
		lw $1, 4($2)
		nop
		nop
		add $2, $2, $1

Exercise 4-2:

For the following sequences of instructions:

1.
```
        lw   $1, 40($6)
        beq  $2, $0, Label      Assume $2 == $0
        sw   $6, 50($2)
Label:  add  $2, $3, $4
        sw   $3, 50($4)
```

2.
```
        lw   $5, -16($5)
        sw   $4, -16($4)
        lw   $3, -20($4)
        beq  $2, $0, Label      Assume $2 != $0
        add  $5, $1, $4
```

Assuming the following latencies for the individual pipeline stages:

	IF	ID	EX	MEM	WB
For sequence 1	100ps	120ps	90ps	130ps	60ps
For sequence 2	180ps	100ps	170ps	220ps	60ps

a. Assume that all branches are perfectly predicted (eliminating control hazards). If we have only 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 five-stage pipeline that only has one memory? Data hazards can be eliminated by adding ***nop***s to the code. Can structural hazard be eliminated in the same way? Why?
b. Assume that all branches are perfectly predicted (eliminating control hazards). 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 four stages. Change this code to accommodate this changed ISA. Assuming this change doesn't affect clock cycle time, what speed-up is achieved for this instruction sequence?
c. Assuming stall-on-branch, what speed-up 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?
d. Repeat the speed-up calculation of part b, but take into account the possible change in clock cycle time and the provided pipeline stage latencies. When EX and MEM are done in a single stage, most of their work can be done in parallel. As a result, EX/MEM stage has a latency that is larger of the original two plus 20ps needed for the work that couldn't be done in parallel.
e. Repeat the speed-up calculation of part c, but take into account the possible change in clock cycle time and the provided pipeline stage latencies. Assume the latency ID stage increases by 50% and the latency of the EX stage decreases by 10ps when branch outcome resolution is moved to ID.
f. Assume stall-on-branch, 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 speed-up in this case? Assume that the latency of the EX stage is reduced by 20ps and the latency of the MEM stage remains unchanged.

Solution:

a. In the pipelined execution, *** represents a stall when an instruction can't be fetched because a load or store instruction is using the memory in that cycle. We can't add ***nop***s to eliminate structural hazards as ***nop***s need to be fetched just like any other instructions, so this hazard must be addressed with a hardware hazard detection unit in the processor.

	Instructions	Pipeline stage	Cycles

1	lw $1, 40($6)
	beq $2, $0, Label
	add $2, $3, $4
	sw $3, 50($4)
		IF ID EX MEM WB
		IF ID EX MEM WB
		IF ID EX MEM WB
		*** IF ID EX MEM WB
			9

2	lw $5, -16($5)
	sw $4, -16($4)
	lw $3, -20($4)
	beq $2, $0, Label
	add $5, $1, $4
		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
			12

b. This change only saves one cycle in an entire execution without data hazards. If there were data hazards from loads to other instructions, the change would help eliminate some stall cycles.

	Instruction executed	Cycles with 5 stages	Cycles with 4 stages	Speed-up
1	4	4+4 = 8	3+4 = 7	8/7 = 1.14
2	5	4+5 = 9	3+5 = 8	9/8 = 1.13

c. Stall-on-branch delays the fetch of the next instruction until the branch is executed. When branches execute in the ID stage, each branch cause one stall only. Without the branch stalls; perfect branch prediction, there are no stalls, and the execution time is 4 plus the number of executed instructions.

	Instruction executed	Branches executed	Cycles with branch in EX	Cycles with branch in ID	Speed-up
1	4	1	4+4+1*2 = 10	4+4+1*1=9	10/9 = 1.11
2	5	1	4+5+1*2 = 11	4+5+1*1=10	11/10 = 1.1

d. 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.

	Cycles time with 5 stages	Cycles time with 4 stages	Speed-up
1	130ps (MEM)	150ps (MEM +20ps)	(8*130)/(7*150) = 0.99 slowdown
2	220ps (MEM)	240ps (MEM +20ps)	(9*220)/(8*240) = 1.03

e.

	New ID latency	NEW EX latency	New cycle time	Old cycle time	Speed-up
1	180ps	80ps	180ps (ID)	130ps (MEM)	(10*130)/(9*180) = 0.8 slowdown
2	150ps	160ps	220ps (MEM)	220ps (MEM)	(11*220)/(10*220) = 1.1

f. The cycle time remains unchanged; a 20ps reduction in EX latency has no effect on clock cycle time because EX is not the longest-latency stage. The change affects the execution time because it adds one additional stall cycle to each branch, because the clock cycle time doesn't improve but the number of cycles increases.

	Cycles with branch in EX	Execution time (branch in EX)	Cycles with branch in MEM	Execution time (branch in MEM)	Speed-up
1	4+4+1*2 = 10	10*130 = 1300ps	4+4+1*3 = 11	11*130 = 1430ps	0.91
2	4+5+1*2 = 11	11*220 = 2420ps	4+5+1*3 = 12	12*220 = 2640ps	0.92

Exercise 4-3:

Assuming a static branch-not-taken prediction technique, and that a conditional branch outcome is actually determined in the MEM phase, how many cycles does a beq instruction has to stall if the instruction immediately preceding it affects one of the register it compares. Consider each of the following cases:

Forwarding Available	Preceding Instruction
Yes	lw
Yes	R-format
No	lw
No	R-format

What is branch penalty if the branch turns out to be taken?

Solution:

Forwarding Available	Preceding Instruction	Number of stall cycles
Yes	lw	1
Yes	R-format	0
No	lw	2
No	R-format	2

The branch penalty if the branch turns out to be taken is 3 clock cycles.

Exercise 4-4:

Repeat Exercise 4-3, assuming the branch outcome is determined in the EX phase. Repeat once more if the outcome is determined in the ID phase

Solution:

If branch is determined in the EX phase (same as if the branch is determined in MEM phase)

Forwarding Available	Preceding Instruction	Number of stall cycles
Yes	lw	1
Yes	R-format	0
No	lw	2
No	R-format	2

The branch penalty if the branch turns out to be taken is 2 clock cycles.

If branch is determined in the ID phase

Forwarding Available	Preceding Instruction	Number of stall cycles
Yes	lw	2
Yes	R-format	1
No	lw	2
No	R-format	2

The branch penalty if the branch turns out to be taken is 1 clock cycle only.

Exercise 4-5:

What does the IF.Flush control signal do? Give an example when it should be set to 1.

Solution:

The IF.Flush control signal clears the instruction stored in the IF/ID register (replacing it with 32 zeros). This is equivalent to the following instruction sll $0,$0,0 (which is the actual encoding of a nop instruction). The IF.Flush signal should be set to one to cancel an instruction that has just been fetched if it was fetched due to an incorrect branch prediction (as in the case of a static branch-not-taken prediction when the actual branch outcome turns out to be taken. In this case we set the IF.Flush signal to 1 to cancel the instruction immediately after the conditional branch instruction that has been incorrectly fetched).

Exercise 4-6:

Consider the following MIPS code where $s0 contains the value 500, which is the starting address of an array containing the values from 1 to 12. If a 1-bit branch predictor is used to predict the branch outcome of both conditional branch instructions, what is the misprediction percentage? Assume that both bne instructions are initially predicted as taken. What if a 2-bit branch predictor is used? In this case assume that both bne instructions are initially considered to be in the weakly-taken state (the weakly-taken state is the taken state that is directly connected to a not taken state).

```
    addi  $t0, $zero, 548
    addi  $t1, $zero, 10
L1: add   $t2, $s0, $zero
```

```
L2: lw   $t3, 0($t2)
    addi $t3, $t3, 1
    sw   $t3, 0($t2)
    addi $t2, $t2, 4
    bne  $t0, $t2, L2
    addi $t1, $t1, -1
    bne  $t1, $zero, L1
```

Solution:

The bne instruction in line 8 is executed 120 times.
The bne instruction in line 10 is executed 10 times.

If 1-bit branch prediction is used,
Line 8 bne is misspredicted = 1+2*9 = 19 times
Line 10 bne is misspredicted 1 time
Total number of misspredictions = 20 times
Misspreduction percentage = 20/130 = 15.38 %

If 2-bit branch prediction is used,
Line 8 bne is misspredicted 10 times
Line 10 bne is misspredicted 1 time
Total number of misspredictions = 11 times
Misspreduction percentage = 11/130 = 8.46%

Exercise 4-7:

In which of the pipeline stages does each of the following exceptions get detected?

a. Undefined Opcode
b. Overflow
c. Invalid Data Address (in case of lw/sw instructions).
d. Invalid Instruction Address (in case of beq instructions).

Solution:

a. Undefined Opcode: Detected in the ID phase
b. Overflow: Detected in the EX phase
c. Invalid Data Address (in case of lw/sw instructions): Detected in the EX phase
d. Invalid Instruction Address (in case of beq instructions): Detected in the ID, EX, or MEM phase depending on where the branch outcome gets decided.

Exercise 4-8:

Assume that the lw instruction below causes an invalid data address exception and that the first instruction is fetched in clock cycle 1. In which cycle does the exception occur? List the instructions in each of the pipeline phases during this clock cycle and during the clock cycle that follows. What control signals are affected by the exception? In which clock cycles are these control signals set and what are their values set to?

```
    add   $t0, $t1, $t2
    or    $t1, $t2, $t3
    lw    $s0, 0($s1)
    sw    $s3, 0($s2)
    beq   $t5, $t4, label
```

Solution:

The exception occurs in the EX phase of the `lw` instruction which is cycle 5 (since the lw instruction is fetched in cycle 3). In clock cycle 5, the `add` instruction is in the WB phase, the `or` instruction is in the MEM phase, the `lw` instruction is in the EX phase, the `sw` instruction is in the ID phase, and the `beq` is in the IF phase. Because of the exception detected in clock cycle5, the state of the pipeline in clock cycle 6 will be as follows: The `or` instruction is in the WB instruction, the first instruction of the exception handler is in the IF phase, while the ID, EX, and MEM phases will have bubbles corresponding to the flushed `beq`, `sw`, and `lw` instructions respectively.

The control signals affected by the exception are the IF.Flush, ID.Flush, EX.Flush, and the selection lines of the multiplexer controlling the next PC value. These signals will be set in clock cycle 5 (when the exception is detected). The IF.Flush, ID.Flush, EX.Flush will all be set to 1, while the selection lines of the PC source multiplexer will be adjusted to load the PC with the exception handler address.

Exercise 4-9:

Consider the following C code and the corresponding MIPS instructions. Assuming the loop is executed for a huge number of iterations and assuming perfect branch direction and target prediction, what is the speedup (in terms of IPC) achieved when going from a 1-issue processor to a 2-issue statically scheduled (in-order superscalar) processor with the restriction that for each instruction packet one instruction has to be ALU/branch and the other has to be a memory access instruction. Assume that the compiler is free to rearrange instructions in case of the 2-issue processor. What would the IPC speedup become if the compiler also unrolls the loop with an unrolling factor of 2? Show the compiler schedule you assumed in both cases (with and without unrolling).

```
for(i=0; i!=j; i++)
       a[i] = b[i];
```

Equivalent MIPS instructions:

```
    add   $t0, $0, $0
    beq   $t0, $t1, ex
lp: sll   $t2, $t0, 2
    addi  $t3, $t2, $s1
    lw    $t4, 0($t3)
    addi  $t5, $t2, $s0
    sw    $t4, 0($t5)
    addi  $t0, $t0, 1
    bne   $t0, $t1, lp
ex:
```

Solution:

Since the loop is executed a large number of times, we can ignore the effect of the instructions outside the loop on the IPC. In case of a 1-issue processor the IPC would be 1 (since the loop body would require 7 clock cycles to execute 7 instructions). In case of a 2-issue processor, the compiler could schedule the loop instructions as follows:

ALU/Branch	Load Store
`sll $t2, $t0, 2`	nop
`addi $t3, $t2, $s1`	nop
`addi $t5, $t2, $s0`	`lw $t4, 0($t3)`
`addi $t0, $t0, 1`	nop
`bne $t0, $t1, lp`	`sw $t4, 0($t5)`

In this case the IPC would be: 7/5 = 1.4. This indicates in a speedup of 1.4.

In case of unrolling the loop by a factor of 2, the schedule might become:

ALU/Branch	Load Store
`sll $t2, $t0, 2`	nop
`addi $t3, $t2, $s1`	nop
`addi $t5, $t2, $s0`	`lw $t4, 0($t3)`
`addi $t0, $t0, 2`	`lw $t6, 4($t3)`
nop	`sw $t4, 0($t5)`
`bne $t0, $t1, lp`	`sw $t6, 4($t5)`

In this case the IPC would be: 9/6 = 1.5 and the speedup would become 1.5.