COE 501: Computer Architecture

Problem Set 4: Pipelining Basic and Intermediate Concepts

1) (15 pts) Use the following code fragment:

```
I1:     LD      R1, 0(R2)          ; Load R1 = Memory(R2)
I2:     DADDI   R1, R1, 1          ; R1 = R1 + 1
I3:     SD      R1, 0(R2)          ; Store Memory(R2) = R1
I4:     DADDI   R2, R2, 8          ; R2 = R2 + 8
I5:     DADDI   R4, R4, -1         ; R4 = R4 - 1
I6:     BNE     R4, R0, I1         ; Branch if R4 != 0
```

Assume that the initial value of R4 is 100.

a) (2 pts) List all the true data dependences in the code above within one loop iteration. Record the register, source instruction, and destination instruction.

Data Dependences (within one loop iteration):

Register R1: I1 (LD) → I2 (DADDI)
Register R1: I2 (DADDI) → I3 (SD)
Register R4: I5 (DADDI) → I6 (BNE)

b) (4 pts) Show the timing of the above instruction sequence for the 5-stage MIPS pipeline without any forwarding hardware. Use a pipeline timing chart to show all stall cycles. Assume that the branch is handled by predicting it as NOT taken. If the branch outcome is TAKEN, it kills the next two instructions in the pipeline. How many cycles does this loop take to execute? What is the average CPI?

No forwarding hardware. Taken branch kills next two instructions.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
LD	IF	ID	EX	M	WB																	
DADDI		IF	s	s	s	ID	EX	M	WB													
SD						IF	s	s	s	ID	EX	M										
DADDI										IF	ID	EX	M	WB								
DADDI											IF	ID	EX	M	WB							
BNE												IF	s	s	s	ID	EX					
next1																						
next2																						
LD																		IF	ID	EX	M	WB
	Average of 17 cycles per iteration																	Start of next iteration				

Total cycles = 17 × 100 = 1700 cycles
(Last iterate does not branch and we don't kill next 2 instructions but this ignored for simplicity)
Average CPI = 17 cycles / 6 instructions = 2.83

c) (5 pts) Assuming delayed branching, rewrite the above code to take advantage of the branch delay slot. Show the timing of the above instruction sequence for the 5-stage MIPS pipeline with full forwarding hardware. How many cycles does this loop take to execute? What is the average CPI?

Delayed Branching + Forwarding hardware.
Code can be rewritten as follows to take advantage of the branch delay slot:

```
I1:     LD      R1, 0(R2)          ; Load R1 = Memory(R2)
I2:     DADDI   R1, R1, 1          ; R1 = R1 + 1
I3:     SD      R1, 0(R2)          ; Store Memory(R2) = R1
I5:     DADDI   R4, R4, -1         ; R4 = R4 - 1
I6:     BNE     R4, R0, I1         ; Branch if R4 != 0
I4:     DADDI   R2, R2, 8          ; R2 = R2 + 8
```

	1	2	3	4	5	6	7	8	9	10	11	12	13
LD	IF	ID	EX	M	WB								
DADDI		IF	s	ID	EX	M	WB						
SD				IF	ID	EX	M						
DADDI					IF	ID	EX	M	WB				
BNE						IF	ID	EX					
DADDI							IF	ID	EX	M	WB		
next													
LD									IF	ID	EX	M	WB
	8 cycles per iteration								Next iterate				

Total cycles = 8 × 100 = 800 cycles
Average CPI = 8 cycles / 6 instructions = 1.33

d) (4 pts) Cache memory stages sometimes take longer to access than other pipeline stages. Consider a 7-stage pipeline: IF1, IF2, ID, EX, MEM1, MEM2, WB, where instruction fetch is split into two stages: IF1 and IF2, and the data memory is also split into two stages: MEM1 and MEM2. Show the timing of the above instruction sequence for the 7-stage pipeline will full forwarding hardware. Assume that the branch is handled by predicting it as always TAKEN with zero delay in the IF1 stage. How many cycles does this loop take to execute? What is the average CPI?

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
LD	IF1	IF2	ID	EX	M1	M2	WB										
DADDI		IF1	IF2	s	s	ID	EX	M1	M2	WB							
SD			IF1	s	s	IF2	ID	EX	M1	M2							
DADDI						IF1	IF2	ID	EX	M1	M2	WB					
DADDI							IF1	IF2	ID	EX	M1	M2	WB				
BNE								IF1	IF2	ID	EX						
LD									IF1	IF2	ID	EX	M1	M2	WB		
	8 cycles per iteration								Next Iterate								

Total cycles = 8 × 100 = 800 cycles
Average CPI = 8 cycles / 6 instructions = 1.33

2) (5 pts) Consider the following branch and jump frequencies. Assume there is NO branch target buffer (BTB) in the first stage and that branches and jumps are not resolved until later stages in the pipeline.

Conditional branches = 20%
Unconditional Jumps and Calls = 3%
70% of conditional branches are taken

a) (2 pts) We are examining a 5-stage processor pipeline where the unconditional jump and call instructions are resolved at the end of the second stage, and the conditional branches are resolved at the end of the third stage. Ignoring other pipeline stalls, how much faster would the processor pipeline be without any control hazards?

Unconditional Jump and Call Delay = 1 cycle (Kills next instruction)
Taken Conditional Branch Delay = 2 cycles (Kills next 2 instructions)
Untaken Conditional Branch kills 0 instructions

CPI with control hazards = $1 + 0.03 \times 1 + 0.2 \times 0.7 \times 2 = 1.31$
CPI without control hazards = 1
Speedup = 1.31 / 1 = 1.31

b) (3 pts) Now assume a 10-stage deep pipeline, where unconditional jumps and calls are resolved at the end of the fourth stage and conditional branches are resolved at the end of the seventh stage. Ignoring other pipeline stalls, how much faster would the processor pipeline be without any control hazards?

Unconditional Jump and Call Delay = 3 cycles (Kills next 3 instructions)
Taken Conditional Branch Delay = 6 cycles (Kills next 6 instructions)
Untaken Conditional Branch kills 0 instructions

CPI with control hazards = $1 + 0.03 \times 3 + 0.2 \times 0.7 \times 6 = 1.93$
CPI without control hazards = 1
Speedup = 1.93 / 1 = 1.93

3) (7 pts) In this problem, we will explore how a deep processor pipeline affects performance in two ways: faster clock cycle and increased stalls due to data and control hazards. Assume that the original processor is a 5-stage pipeline with a 1 ns clock cycle. The second processor is a 12-stage pipeline with a 0.5 ns clock cycle. The 5-stage pipeline experiences one stall cycle due to a data hazard every 5 instructions, whereas the 12-stage pipeline experiences 3 stall cycles every 8 instructions. In addition, branches constitute 20% of the instruction count, and the misprediction rate for both pipelines is 5%.

a) (3 pts) What is the speedup of the 12-stage pipeline over the 5-stage pipeline, taking into account only data hazards?

Average CPI (5-stage pipeline) = 1 + 1/5 = 6/5 (Data hazard stalls only)
Average CPI (12-stage pipeline) = 1 + 3/8 = 11/8 (Data hazard stalls only)
Speedup = (6/5 × 1 ns) / (11/8 × 0.5 ns) = 1.745 (Data hazards only)

b) (4 pts) If the branch misprediction penalty is 2 cycles for the 5-stage pipeline, but 6 cycles for the 12-stage pipeline, what are the CPIs of each, taking into account the stalls of the data hazards and branch hazards?

Average CPI (5-stage pipeline) = 1 + 1/5 + 0.2 × 0.05 × 2 = 1.22 (Data + Branch Hazards)
Average CPI (12-stage pipeline) = 1 + 3/8 + 0.2 × 0.05 × 6 = 1.435 (Data + Branch Hazards)
Speedup = (1.22 × 1 ns) / (1.435 × 0.5 ns) = 1.70 (Data + Branch Hazards)

4) (13 pts) We will now add support for register-memory ALU operations to the classic five-stage MIPS pipeline. To simplify the problem, all memory addressing will be restricted to register indirect. All addresses are simply a value held in a register. No displacement may be added to the register value. For example, **ADD R4, R5, (R8)** means **R4 = R5 + Memory(R8)**. Only one memory operand can be read, but not written. To write memory, the store instruction should be used instead. Register-register ALU operations are unchanged. For example, the instruction **ADD R4, R5, R8** means **R4 = R5 + R8**.

a) (2 pts) List a rearranged order of the five traditional stages of the MIPS pipeline that will support register-memory operations implemented exclusively by register indirect addressing.

IF = Instruction Fetch (as before)
ID = Instruction Decode (as before)
MEM = Memory Stage (comes before the Execute stage)
EX = Execute (comes after the Memory stage)
WB = Write Back stage (as before)

The memory stage should come before the execute stage to allow a memory operand to be read from memory before execution.

b) (5 pts) Describe what forwarding paths are needed for the rearranged pipeline by stating the source stage, destination stage, and information transferred on each needed new path. Give an instruction sequence showing each data hazard that can be resolved by forwarding data between stages. Draw a timing diagram showing the forwarding between stages.

Forwarding from MEM back to MEM stage:

```
LD      R7, (R6)      ; Load R7 = Memory(R6)
LD      R8, (R7)      ; Load R8 = Memory(R7)
```

Value of R7 should be forwarded from output of MEM back to address input of MEM.

```
LD      R8, (R6)      ; Load R8 = Memory(R6)
SD      R8, (R7)      ; Store Memory(R7) = R8
```

Value of R8 should be forwarded from output of MEM back to data input of MEM.

Forwarding from WB and EX stages back to the EX stage:

```
ADD     R4, R5, (R6)  ; R4 = R5 + Memory(R6)
SUB     R7, R5, (R8)  ; R7 = R5 – Memory (R8)
AND     R9, R4, R7    ; R9 = R4 & R7
```

Values of R4 and R7 should be forwarded from WB and EX stages back to the EX stage.

Forwarding from WB and EX stages back to the MEM stage:

DADD R4, R5, (R6) ; R4 = R5 + Memory(R6)
DSUB R7, R5, (R8) ; R7 = R5 – Memory (R8)
SD R4, (R9) ; Memory(R9) = R4
AND R3, R3, (R4) ; R3 = R3 & Memory(R4)

Value of R4 should be forwarded from the output of the EX stage back data input of the MEM stage (needed by SD). In addition, value of R4 should be forwarded from the WB stage back to the address input of the MEM stage (needed by AND).

	1	2	3	4	5	6	7	8	9	10
LD R7, (R6)	IF	ID	MEM	EX	WB					
LD R8, (R7)		IF	ID	MEM	EX	WB				
LD R8, (R6)	IF	ID	MEM	EX	WB					
SD R8, (R7)		IF	ID	MEM						
ADD R4, R5, (R6)	IF	ID	MEM	EX	WB					
SUB R7, R5, (R8)		IF	ID	MEM	EX	WB				
AND R9, R4, R7			IF	ID	MEM	EX	WB			
DADD R4, R5, (R6)	IF	ID	MEM	EX	WB					
DSUB R7, R5, (R8)		IF	ID	MEM	EX	WB				
SD R4, (R9)			IF	ID	MEM					
AND R3, R7, (R4)				IF	ID	MEM	EX	WB		

c) (3 pts) For the reordered stages of the pipeline, what data hazards cannot be forwarded and cause stall cycles? Give an instruction sequence showing each data hazard that causes stall cycles. Draw a timing diagram showing the stall cycles caused by each data hazard.

Because the EX stage is rearranged after the MEM stage, some RAW data hazards cause stall cycles in the new pipeline.

DADD R4, R5, (R6) ; R4 = R5 + Memory(R6)
SD R4, (R9) ; Memory(R9) = R4

Stall 1 cycle until the value of R4 is computed in the EX stage. The MEM stage is waiting for data to be computed in the EX stage.

DADDI R7, R7, 16 ; R7 = R7 + 16
DSUB R8, R8, (R7) ; R8 = R8 – Memory(R7)

Stall 1 cycle until the value of R7 is computed in the EX stage. The MEM stage is waiting for the address to be computed in the EX stage.

	1	2	3	4	5	6	7	8	9	10
DADD R4, R5, (R6)	IF	ID	MEM	EX	WB					
SD R4, (R9)		IF	ID	**stall**	MEM					
DADDI R7, R7, 16	IF	ID	MEM	EX	WB					
DSUB R8, R8, (R7)		IF	ID	**stall**	MEM	EX	WB			

d) (1 pts) What is the penalty of the branch instruction in the new pipeline?

Because the EX stage is now the fourth stage in the pipeline, the penalty of the branch instruction has increased from 2 cycles to 3 cycles.

e) (2 pts) List all of the ways that the new pipeline with register-memory ALU operations can have a different instruction count for a given program than the original pipeline (that supports register-register ALU operations only). Give specific instruction sequences, one for the original pipeline and one for the rearranged pipeline, to illustrate each way.

Because register-memory operations are supported, the number of load instructions can be reduced. For example, to translate A = B + C requires 4 instructions in the original MIPS architecture, while 3 instructions only if register-memory operations are supported.

A = B + C (No register-memory ALU operations) R4, R5, R6 contain addresses of A, B, and C:

```
LD      R10, (R5)          ; Load R10 = Memory(R5) = B
LD      R11, (R6)          ; Load R11 = Memory(R6) = C
DADD    R12, R10, R11      ; R12 = B + C
SD      R12, (R4)          ; Store A = Memory(R4) = R12
```

A = B + C (Register-memory ALU operations are supported):

```
LD      R10, (R5)          ; Load R10 = Memory(R5) = B
DADD    R12, R10, (R6)     ; R12 = B + C
SD      R12, (R4)          ; Store A = Memory(R4) = R12
```

Because Register Indirect addressing can be used only and there is no displacement addressing, additional ALU instructions are required to calculate memory addresses. For example, the following instruction sequence:

```
LD      R10, 8(R4)
LD      R11, 16(R4)
LD      R12, 24(R4)
```

Should be rewritten as follows if only register-indirect addressing is supported, causing an increase in the instruction count:

```
DADDI   R5, R4, 8
DADDI   R6, R4, 16
DADDI   R7, R4, 24
LD      R10, (R5)
LD      R11, (R6)
LD      R12, (R7)
```