Computer Architecture-I

Assignment 2 Solution

Assumptions:

a) We assume that accesses to data memory and code memory can happen in parallel due to dedicated instruction and data bus.
b) The `STORE` instruction does not write the value to memory in the *MEM* stage but waits till *WB* stage.
c) The discussion assumes this is an integer pipeline and all stages require one clock cycle.
d) There are 16 general purpose registers and the PC in the register file.

1. Structural hazards in the given pipeline due to adders can happen in the following parts:

a. The *IF* stage needs an incrementer to set the PC to point to the next instruction to be fetched (PC + 4).
b. The *ALU1* stage uses and adder to compute the effective address for load/store and branch instructions.
c. The *ALU2* stage needs an adder for its arithmetic operations.

Being pipelined, all these adders will be required by their respective stages in parallel to avoid a stall. This is elucidated by the following sequence of instructions:

	CLK1	CLK2	CLK3	CLK4	CLK5	CLK6
ADD	IF	RF	**ALU1**	MEM	**ALU2**	WB
OR		IF	RF	**ALU1**	MEM	**ALU2**
STORE			**IF**	RF	**ALU1**	MEM
ADD				**IF**	RF	**ALU1**
ADD					**IF**	RF

During `CLK3 – CLK6`, we can see that the adder resource is required by different instructions at the same time (stages are marked in bold). In particular, `CLK5` shows the requirement of all the suggested adders. Thus, we require *three* adders to avoid structural hazards in this pipeline.

2. Register file read-write ports are accessed from various stages of the pipeline:

a. The PC is read by the *IF* stage in every clock cycle and also written to after incrementing by 4. These constitute a read and write to the register file in every cycle.
b. The *RF* stage fetches the registers in source operands required by the instruction.

c. *WB* stage writes to register file for all the instructions.

Consider the following sequence of instructions:

```
LOAD   R4, 0(R8)
AND    R1, R5, R2
OR     R1, R1, R3
OR     R2, R2, R7
ADD    R3, R2, R1
STORE  R3, 0(R8)
```

The `LOAD` instruction reads the value from memory location in R8 and writes it to the register R4 in the *WB* stage. At this time, instruction `ADD` is in the *RF* stage and wants to read two operands (R2 and R1). Thus, in steady state execution of the pipeline, there are three reads and two write to the register file in parallel. In order to avoid stalls due to structural hazards, it is necessary to design the register file with *two* write ports and *three* read ports.

3. It is assumed that the ALUs are on different hardware. Following examples show sequence of instructions which expose potential data hazards in the given pipeline and can be countered using forwarding to the ALUs.

a. (ALU2/WB - ALU2) and (WB – ALU2)

	CLK1	CLK2	CLK3	CLK4	CLK5	CLK6	CLK7
LOAD **R4,**8(R7)	IF	RF	ALU1	MEM	**ALU2**	**WB**	
AND R2,R2,R4		IF	RF	ALU1	MEM	**ALU2**	WB
BEQ R1,**R4,**L1			IF	RF	ALU1	MEM	**ALU2**

As per the given instruction set architecture description, the branch statement is testing for equality of the two source register operands and jumps conditionally to the location L1 (a PC-relative offset).

b. MEM/ALU2 – ALU1

	CLK1	CLK2	CLK3	CLK4	CLK5	CLK6	CLK7
LOAD **R4,**4(R2)	IF	RF	ALU1	**MEM**	ALU2	WB	
AND R1,R1,R3		IF	RF	ALU1	MEM	ALU2	WB
STORE R6,R1,4(**R4**)			IF	RF	**ALU1**	MEM	ALU2

c. ALU2/WB – ALU1

	CLK1	CLK2	CLK3	CLK4	CLK5	CLK6	CLK7
ADD R4,R4,R2	IF	RF	ALU1	MEM	**ALU2**	WB	
SUB R6,R6,R8		IF	RF	ALU1	MEM	ALU2	WB
AND R1,R1,R3			IF	RF	ALU1	MEM	ALU2
LOAD R1,0(R4)				IF	RF	**ALU1**	MEM

4. Now, we look at data hazards involving other pipeline stages and the forwarding done to circumvent them. We assume that the STORE instruction does not write the value to memory in the *MEM* stage but waits till *WB* stage.

a. MEM/ALU2 – MEM

	CLK1	CLK2	CLK3	CLK4	CLK5	CLK6	CLK7
LOAD **R6,**4(R3)	IF	RF	ALU1	**MEM**	ALU2	WB	
STORE R4,**R6,**4(**R2**)		IF	RF	ALU1	**MEM**	ALU2	WB

b. ALU2/WB – MEM

	CLK1	CLK2	CLK3	CLK4	CLK5	CLK6	CLK7
ADD **R4**,R7,R7	IF	RF	ALU1	MEM	**ALU2**	WB	
OR R1,R1,R2		IF	RF	ALU1	MEM	ALU2	WB
STORE R6,**R4**,8(R5)			IF	RF	ALU1	**MEM**	ALU2

5. There are a few other data and control hazards which cannot be taken care of, in spite of the forwarding shown in the questions above. We assume that the STORE instruction does not write the value to memory in the *MEM* stage but waits till *WB* stage.

a. ALU2/WB – ALU1

	CLK1	CLK2	CLK3	CLK4	CLK5	CLK6	CLK7
ADD **R4**,R3,R2	IF	RF	ALU1	MEM	ALU2	WB	
STORE R6,R1,4(**R4**)		IF	RF	**Stall**	**Stall**	ALU1	MEM

b. MEM/ALU2 – ALU1

	CLK1	CLK2	CLK3	CLK4	CLK5	CLK6	CLK7
STORE R6,R4,4(**R2**)	IF	RF	ALU1	MEM	ALU2	WB	
LOAD R1,4(**R2**)		IF	RF	ALU1	**Stall**	**Stall**	MEM

c. ALU2/WB – MEM

	CLK1	CLK2	CLK3	CLK4	CLK5	CLK6	CLK7
SUB **R1**,R1,R2	IF	RF	ALU1	MEM	ALU2	WB	
STORE R3,**R1**,4(R4)		IF	RF	ALU1	**Stall**	MEM	WB

d. Conditional Branch – Control Hazard

We assume that the branch prediction policy is to statically predict *not-taken*. In the given pipeline, the condition for branching is evaluated in ALU2 stage. Thus, if there is a mis-prediction, all the instructions in the pipeline prior to ALU2 stage are to be flushed (converted to NOPs) and a new instruction is fetched from the target address on the next clock cycle. Thus, there is a stall of *four* clock cycles.

The data hazards which can be avoided by forwarding to the ALUs and other stages have been given above in (Q3, Q4). There still exist hazards in scenarios where the data produced by one stage is required by the previous stage in the same clock cycle. We also have cases where the data produced by one stage is required by another stage in a clock cycle before the current one. This is impossible to achieve as we cannot pass data backwards in time. These cases have been illustrated above and shown that a *Stall* is inevitable. Thus, we can say that we have explored all the possible hazards for this pipeline.

6. Three schemes below can be used to ensure that the pipeline has precise exceptions:

a. Buffering with History File:

In this scheme, we buffer the original contents of registers and memory in a *history* file before changing them in the target locations. In this way, once an exception occurs, a trap is injected, the exception is handled and pipeline will resume from the instruction after the faulting instruction. It will roll back to the previous checkpoint by recovering data to target locations from the *history* file. Upon restarting the pipeline, we will execute the following instructions again.

b. Verify and Execute:

In this scheme, we issue an instruction from the RF stage only after verifying that all the instructions (issued) ahead in the pipeline will not cause an exception. This is a conservative approach wherein, instructions are tested and flagged as potential for exceptions. If

flagged, further instructions will not be issued. Instead, a trap routine will handle the problematic operation. This makes the logic for restarting very simple and always tractable. This method works well for certain arithmetic exceptions like overflow, divide by zero, etc. which happen within an instruction execution.

c. Exception Status Vector:

In this scheme, we maintain a status vector for exceptions caused by an instruction. The moment we detect an exception, the corresponding flag is marked along with the instruction that caused it. We also disable memory and register writes for this instruction. Since, in our pipeline, the instructions are not committed till the very last stage (WB), this method is most suited. The instructions ahead of the fault-causing instruction complete their operation without any delay and those behind, are also allowed to proceed normally. When the fault-causing instruction comes to the WB stage, we check the exception status vector and resolve all the exceptions as per the order in the original un-pipelined instruction sequence.

The first scheme does not give much advantage for the given pipeline as the stages are already designed in a way such that commit happens only in the last stage. Thus, there are no writes to be saved in the history file. This holds true because we have assumed in-order execution and completion. The 'Verify and Execute' scheme involves a significant hardware overhead to detect potential exceptions. Furthermore, the pipeline will experience frequent stalling which can affect performance. The status vector method is most suited for the given pipeline. When multiple exceptions occur, this method guarantees an in-order handling of all exceptions. It also has lesser hardware involved.

7. Let us assume that the branch prediction for the given pipeline is statically predicted as *not-taken*.
In the ideal case, the CPI for the pipeline should be 1.
But, practically, there will be stalls because of branch mis-prediction. The extent to which they affect the performance will be decided by the penalty incurred and the branch frequency.

Type of Branch	Branch Frequency	Branch Penalty (cycles)
Unconditional	0.05	2
Cond. - Taken	0.1	4
Cond. - Not Taken	0.15	0

Thus, the total stalls caused because of branches are,
(0.05 x 2) + (0.1 x 4) + (0.15 x 0) = 0.50
Therefore, CPI = 1 + 0.50 = 1.50
Thus, the required ratio = 1.50/1 = *1.50*

The penalty of a mis-prediction is quite high (4 cycles) for the given pipeline when compared to the classic 5 stage pipeline penalty of 2 cycles (branch decision is done in RF/ID stage). This causes the CPI to be adversely affected for the given pipeline.

8. Let us assume that the MIPS register-register architecture is the classic 5-stage pipeline as compared to the 6 stages in the given pipeline. If there were no stalls and hazards, our register-memory architecture has a better performance of 20%.

Optimistic Assessment:

The table gives the instruction mix for various benchmarks. Let us consider the integer average numbers for our calculation. We can see that 26% are `LOAD`, 10% are `STORE` instructions and the arithmetic operations (`ADD, SUB, OR, AND`) constitute 35%. If we consider the MIPS register-register architecture, it could very well be that most of the arithmetic instructions in the benchmarks are to be supported by a preceding `LOAD` to bring in the data from some memory location.

Register-Register (MIPS)	Register-Memory
`LOAD R4, 4(R2)` `ADD R1, R1, R4`	`ADD R1, R1, 4(R2)`

Now, if we were to generate the code for our register-memory instruction set, each of these `LOAD` instructions would be avoided. Thus, the code size will be lesser by 26%.

The other consequence of this would be lesser RAW hazards to be handled in the pipeline because of `LOAD` due to a limited set of registers. As seen in the code snippet above, the MIPS code will need to stall the `ADD` instruction in ID/RF stage till the `LOAD` instruction retrieves the updated value of R4 at the end of MEM stage. It can be seen that there is no stall in the corresponding code in the register-memory architecture.

Pessimistic Assessment:

We saw previously how compact code can be written using the register-memory architecture. This might seem like an advantage at first, but there can be a difference in performance despite code-size as shown below.

Register-Register (MIPS)	Register-Memory

LOAD R4, 4(R2)
ADD R1, R1, R4
OR R7, R4, R5
SUB R3, R3, R4
STORE R6, R7, 4(R2)
	ADD R1, R1, 4(R2)
	OR R7,R5,4(R2)
	SUB R3,R3,4(R2)
	STORE R6,R7,4(R2)

The register-memory code uses a memory reference each time it needs the value at location `4(R2)` (all three instructions, in this case). Each memory reference is a slow operation (when compared to a general purpose register access) and is increasing traffic on the memory bus (more contention). The register-register code however, loads the value once, performs computation on it by accessing the register where the value is loaded and finally writes back the result when done.

Branch instructions have a smaller penalty (2 cycles) in the register-register architecture assuming the decision for branching is made in the ID/RF stage. In our pipeline, the decision is made in the ALU2 stage. Thus, as seen before, the branch penalty can be as high as 4 cycles.

With the given data, the total numbers of branches (conditional branch, jump, call, return) constitute 15% of the instructions. Assuming that the number of branch statements remains the same for the register-memory architecture, we can compare the effect on CPI in both cases.

$CPI_{register\text{-}register} = 1 + (0.15 \times 2) = 1.3$
$CPI_{register\text{-}memory} = 1 + (0.12 \times 4 + 0.03 \times 2) = 1.54$

This calculation assumes worst case of mis-prediction (i.e. All 12% of conditional branches are mis-predicted).

As we have seen before, there are situations where stalls become inevitable to avoid data hazards. In such cases, the register-memory architecture will perform worse than register-register since the access to data may be from memory and not a general purpose register. Moreover, the access time may not be uniform for different parts of memory, causing CPI to vary according to operand location. So, even if instruction count is reduced, the complexity of addressing modes increases decreasing the average CPI. Ultimately, this hints at the fact that register-memory pipeline cannot be run at high clock speeds like register-register architectures.

Another observation here is that, the encoded instruction length for the register-memory architecture will be greater than the MIPS, since we have three operand instructions with memory references.