Pipelined Processor Implementation

You are a part of the processor design team at the TLA corporation. This design team is developing a pipelined implementation of a *variant* of the MIPS instruction set. The instruction set contains the following instructions: R-type (register-register), `LW`, `SW` and `BEQ`. The processor does not have delay slots in load and branch instructions.
The pipeline is a *four stage pipeline* with the stages

IF Instruction Fetch stage.

ID Instruction Decode, Register Fetch stage.

EXMEM Execute and Memory operation stage.

WB Register File writeback stage.

Each pipeline stage takes one clock cycle. For the most part, this pipeline is very similar to the MIPS pipeline. Some *important differences* between the MIPS pipeline and this pipeline are listed below:

- The **EXMEM** stage does memory access for data in case of a LW or SW instruction and, unlike the original MIPS pipeline, there is no separate **MEM** pipeline stage.
- With the help of extra hardware, the branches have been optimized to take fewer cycles.
- In the **ID** stage after the register values are obtained, the memory address is calculated and stored in MAdr. In this pipeline stage the branch target address (BTA) is also computed.

The pipeline has the following *hardware resources*:

ALU This is only used to perform the operations specified in R-type instructions.

Adders $+_{pc}$, $+_{imm}$ and $+_{bta}$. The pipeline table on the next page shows where these adders are used.

1. $+_{pc}$ increments the PC value by 4 in the IF stage.
2. $+_{imm}$ is used to compute the memory address in the ID stage.
3. $+_{bta}$ is used to compute the branch target address (BTA) in the ID stage (the pipeline description given to you shows this).

Register File This has three ports, two read and one write. In a single cycle two read operations and a write operation can be done and the write operation precedes the read operation.

Cache Unified instruction and data cache from which the processor fetches instructions and does data access. A memory read or write operation takes one cycle. The memory system can support only *one* operation per cycle.

The table below describes the pipeline. Some of the notation used in the table is given below.

IR ← Memory[PC] Set the instruction register (IR) to the word at the memory address in the program counter (PC).

PC ← PC $+_{pc}$ 4 Increment the program counter by 4. Use the $+_{pc}$ adder for this operation.

Rs Register[IR[25:21]]

Rt Register[IR[20:16]]

Rd Register[IR[15:11]]

Imm <IR[15]^ 17 # IR[14:0]> = (sign extended immediate field)

ImmS2 <IR[15]^ 15 # IR[14:0] # 00> = (sign extended immediate field shifted left by 2)

STAGE	R-Type Instruction	LOAD Instruction	STORE Instruction	BEQ Instruction
IF	IR ← Memory[PC] PC ← PC $+_{pc}$ 4	IR ← Memory[PC] PC ← PC $+_{pc}$ 4	IR ← Memory[PC] PC ← PC $+_{pc}$ 4	IR ← Memory[PC] PC ← PC $+_{pc}$ 4
ID	A ← Rs B ← Rt	A ← Rs B ← Rt MAdr ← A $+_{imm}$ Imm	A ← Rs B ← Rt MAdr ← A $+_{imm}$ Imm	A ← Rs B ← Rt BTA ← PC $+_{bta}$ ImmS2 if (A==B) PC ← BTA
EXMEM	ALUout ← A op B	MemData ← Memory[MAdr]	Memory[MAdr] ← B	
WB	Rd ← ALUout	Rt ← MemData		

The pipeline above has *hazards* which prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

Structural These arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.

Data These arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

Control These arise from the pipelining of branches and other instructions that change the PC.

A number of hardware techniques can be used to minimize the performance loss due to pipeline hazards.

Problem 1. (5 points):

Your responsibility as a part of the design team is to identify structural, data and control hazards in the pipeline and suggest solutions to them. The solution you suggest should minimize the performance loss due to these hazards.

A. There is one structural hazard in the design which prevents an instruction fetch on every cycle. Identify it and propose a solution to eliminate it.

B. Upon encountering a branch the processor continues fetching instructions assuming that the branch is not taken. If the branch is indeed taken, it flushes the instructions in the pipe which have been fetched assuming a not taken branch.

What is the branch penalty for a branch when it is taken ? When it is not taken ?

If an application contains 10% branches and 60% of the branches are taken, and all non-branch instructions take one cycle, what is the processor CPI for this application?

C. For this problem consider the 5-stage pipeline as described in Fig 6.48 of your book and the 4-stage pipeline described above. Compute the CPI of the two processors for an application with 20% loads. In this application there is a probability of 0.1 that the value read by the load instruction is used by the instruction following the load. Assume all other instructions require one cycle.

5-stage CPI:

4-stage CPI:

Problem 2. (14 points):

There are a number of data hazards in the design which will cause the pipeline to operate incorrectly for some instruction sequences. You must identify these data hazards and a solution to fix the hazard by filling in the table entries below: For full credit, each row of a column must be correctly and completely filled. In case a hazard does not exist, you may strike out columns 3 to 9. For fixing the hazard, you must indicate a solution which minimizes the speed penalty.

In column 7 indicate the latest pipe stage at the beginning of which the operand must be available.

MIPS CODE	Hazard Exists? (Y/N)	Hazard Type					Hazard Fix	
		Value	Where Generated?		Where Needed?		Stall	Forward
		Register Number	Instr Num.	Pipe Stage	Instr Num	Pipe Stage	(Y/N)	(Y/N)
1. add $1,$2,$3 2. add $3,$1,$2								
1. add $1,$2,$3 2. lw $5,0xf0($1)								
1. add $1,$2,$3 2. sw $1,0xf0($5)								
1. add $1,$2,$3 2. add $3,$5,$6 3. add $5,$1,$2								
1. lw $1,0x40($2) 2. add $3,$1,$6								
1. lw $1,0x40($2) 2. sw $1,0x44($3)								
1. lw $1,0x40($2) 2. add $0,$0,$0 3. sw $1,0x44($3)								