A VERILOG Model of the Pipelined MIPS Processor

PROBLEM

You will be working on a pipelined version of the MIPS (in Verilog). It basically follows the model of Appendix A and has five stages:

1. IF - instruction fetch
2. ID - instruction decode
3. EX - execute or calculate effective address
4. MEM - memory access
5. WB - write back

The model has no structural hazards (since it uses separate data and program memory), but I have not made any of the improvements to handle data hazards or control hazards. That is, there is no data forwarding, and I have not made the modifications we discussed to shorten the control hazard problem. Although the design has the capability to be stalled, the current implementation does not make use of this. Therefore the only way to handle these problems in the design is to insert NOP's liberally whenever the compiler (your brain) detects a data or control hazard.

I have included a piece of object code (called code.v) along with the commented version of the same (codeCommented.v). This code executes a simple loop, and uses NOPs to solve the hazard problems. Your job is to do the following:

Modify the design so that no NOPs are necessary in the code, subject to the following expectations/constraints. Modify the decode stage so that it stalls the pipeline for the appropriate amount of time whenever it detects a data or control hazard. NOTE: You may NOT simply modify the decoder so it stalls three cycles after each instruction.

THE SAMPLE DESIGN (may have bugs, so beware)

The basic outline of the MIPS consists of the register file and the five stages of the pipeline, separated by pipeline registers. The clock signal is a 20 MHz (50 ns) square wave. Each functional unit performs the specified function and produces its results on the negative edge of the clock. The pipeline registers forward inputs to outputs on the rising edge of the clock. In my model data is stored in the reg file on the rising edge of the clock.

Appendix A in your book shows what must happen in each of the pipeline stages for each type of instruction. The discussion below describes the implementation of the interface to each stage.

Instruction Fetch (IF)

This stage is responsible for maintaining the PC and fetching new instructions from program memory. The code memory must be called CODE_MEM[2048:0] and be local to this module. The interface is as follows:

NAME	DIRECTION	WIDTH	DESCRIPTION
PC_IN	input	32	new PC for control transfer
PC_PULSE	input	1	high pulse reloads PC (with PC_IN)
STALL	input	1	don't fetch a new instruction
CLOCK	input	1	global clock for the processor
IR_OUT	output	32	new instruction
PC_OUT	output	32	PC associated with IR_OUT

Instruction Decode (ID)
This stage unpacks the instruction and determines if the pipeline must stall to circumvent a pipeline hazard. Not that A_REG_ADD and B_REG_ADD are applied to the register file so that the registers contents are stored in the pipeline register. The interface is:

NAME	DIRECTION	WIDTH	DESCRIPTION
IR	input	32	instruction register
PC_IN	input	32	new PC for control transfer
CLOCK	input	1	global clock for the processor
PC_OUT	output	32	corresponding PC
OP	output	6	opcode output
FC	output	6	function code output
IMMED	output	32	sign extended immediate
A_REG_ADD	output	5	register specification for ALU_INPUT_A
B_REG_ADD	output	5	register specification for ALU_INPUT_B
D_REG_ADD	output	5	register specification for DESTINATION
STALL_IF	output	1	stall instruction fetch unit
STALL_PIPE	output	1	stall pipeline

Execute Stage (EX)
This section performs a register to register ALU operation, calculates a memory address, or generates a PC branch address. The interface is:

NAME	DIRECTION	WIDTH	DESCRIPTION
PC	input	32	PC for this instruction
OP_IN	input	6	opcode input
FC_IN	input	6	function code input
IMMED	input	32	sign extended immediate
ALU_IN_A	input	32	ALU input A
ALU_IN_B	input	32	ALU input B
DREG_IN	input	5	destination register specification
STALL_IN	input	1	don't execute this
CLOCK	input	1	global clock for the processor
ALU_OUT	output	32	ALU result (data or mem add or PC add)
MEM_DATA	output	32	data to store in memory (if a store)
COND	output	1	result of branch/jmp check (1 = take branch)
OP_OUT	output	6	opcode output
FC_OUT	output	6	function code output
STALL_OUT	output	1	stall pipeline
DREG_OUT	output	5	destination register specification

Memory Stage (MEM)
This section loads data from memory, stores data to memory, or updates the PC. The data memory must be called DATA_MEM[2048:0] and be local to this module. The interface is:

NAME	DIRECTION	WIDTH	DESCRIPTION
ALU_DATA_IN	input	32	ALU result from previous stage
MDR_IN	input	32	data to be stored in memory (if store)
COND	input	1	result of branch/jmp check (1 = take branch)
OP	input	6	opcode input
FC	input	6	function code input
DREG_IN	input	5	destination register specification
STALL_IN	input	1	don't execute this
CLOCK	input	1	global clock for the processor
MDR_OUT	output	32	data to be stored in register
STALL_OUT	output	1	propagate stall or no write (on branch/jmp)
DREG_OUT	output	5	destination register specification
PC	output	32	updated PC
PC_PULSE	output	1	pulse high to reload PC

Write Stage (WB)
This stage transfers the result of a reg to reg operation or a load instruction into the register file. The interface is:

NAME	DIRECTION	WIDTH	DESCRIPTION
DATA_IN	input	32	data to be stored in register
DREG_IN	input	5	destination register specification
STALL_IN	input	1	don't change register
CLOCK	input	1	global clock for the processor
REG_ADD	output	5	address of register to be written
REG_DATA	output	32	data to be written in reg file
REG_WR	output	1	write register

Getting Started
In folder MIPS Design you will find all of the files necessary to construct and run my version of the processor. The files are:

ifetch.v	does the fetch portion
decode.v	does the decode portion
ex.v	does the execute portion
mem.v	does the mem portion
wb.v	does the wb portion
regfil.v	the register file
clkgen	the clock generator
reg_if_dec.v	the register between the IF and DEC stages
reg_dec_ex.v	the register between the DEC and EX stages
reg_ex_mem.v	the register between the EX and MEM stages
reg_mem_wb.v	the register between the MEM and WB stages
cpu.v	the module that connects them all together
code.v	the code portion of our original program, with NOPS inserted
data.v	the data portion of our original program

files.v	the file that lists all these files so you can just type verilog -f files.v and watch it go.

You will also find another module called codeCommented.v which is the same as code.v but with the code commented.

The good news is that not only do these files exist and compile, but the code actually runs and more-or-less works (more on this below), and even prints out some debugging information.

The bad news is that

a. The documentation in the modules is limited,

b. The decode module has no stall detection built in (although as you will see there is provision for passing the stall signals around).

c. There are some small bugs you may have to deal with as you work.

You should start by running the code, looking at the output, and staring at listings of the Verilog models I have given you until you get a clear understanding of what is going on. Then decide what architectural improvements you wish to make. You can work on this project in teams but no more than 2 per team.

You will be graded based on how many NOPs you can remove from the program (code.v) – assuming that your modifications are legitimate, and how long your design will take to run the program. You will not receive zero credit from the project if one of the following applies:

a. Your design does not compile.
b. None of the NOPs from the program can be removed.
c. Your design stalls every instruction (unnecessary stalls will cause loss of points as well).

Enjoy this assignment, and make sure you save your files. It makes a pretty impressive statement of your knowledge of architecture and of simulation using hardware description languages (HDL).