COMP4300

Lab Exercise Four

Objective

This lab builds the MIPS controller and gets a few instructions running.

Instructions

Develop VHDL for the MIPS controller and instruction decoder entities. You should develop an architecture for each of the entities given below. A template for the architecture of the controller is also given, just so you won't get way off track by starting off wrong. THIS BUILDS ON THE LAB THREE RESULTS, SO IF YOU DON'T HAVE THOSE WORKING, THAT IS YOUR FIRST TASK. The entity declarations for the controller and decoder are at the ftp site in the file `lab4_control_v3.vhd`.

First, you will have to develop some remaining components not asked for in LAB 3. These include a one-bit register (to store the jump condition code), a five-bit mux (to handle write-back register index). The entity declarations for these components are in the file `lab4_datapath_v4.vhd` on Canvas.

I am giving you the entities for the memories IM and DM in the file `lab4_datapath_v4.vhd` on Blackboard. The IM and DM memories bear some mention. Your processor should start loading instructions from address 0 of the instruction memory (IM), then load subsequent instructions from address 4, 8, etc. We are not handling jumps yet. To make things easier, the IM entity returns you a full DLX word from the address you ask for. It is just a 1023 element array of `dlx_word` . The only way I have implemented for entering your program is for you to pre-load the IM with some hard-coded instructions. I've preloaded one for you at address zero:

```
 LW R1,4092(R0)
```

That is the first instruction you should get working. You should develop test programs for all the instructions you are expected to implement (see below).

The global wiring for the MIPS is found in a file on Blackboard `lab4_mips_v5.vhd` . This defines the top-level entity `mips` and connects the components.

You should test the `mips` entity by developing simulation files for the entity. You will of course have to do unit testing on the instruction decoder and dlx controller, but I don't want to see those results unless you have problems and have a question.

The only input to the MIPS itself is an external clock. This clock will have a 200ns cycle time (i.e. it is zero from time 0 to time 100, 1 from time 101 to time 200, 0 from 201 to 300, etc.). This is long enough to let all the 5 ns delays run out before the clock cycle ends.

You should use the `dlx_types` package in the datapath file to for the various pieces of a dlx word.

Propagation delays through the decoder and controller should be 5 ns, just like all the other functional units.

Instruction Decoder This is just another piece of combinational logic that takes an instruction as input and chops it into the appropriate component pieces. If a field is not used for a particular type of instruction, set it to zero (won't matter, because it won't be used).

MIPS Controller This is a 5-state machine that handles the clock inputs to all the latches, sets the control inputs to the multiplexors, and converts the function code into a 4-bit code and signed bit for the ALU. It should implement the MIPS architecture exactly as described in textbook sections A.3 (but no pipeline). Each instruction will take 5 cycles to execute, and instruction execution will not overlap.

Instruction Format

There is only one instruction format for MIPS instructions, and all instructions are 32 bits (4 bytes) long. An instruction must begin in an address divisible by 4. Jump target addresses must be valid addresses for the start of instructions (you don't need to test for that unless you want to implement the control flow instructions for extra credit). The format looks like this:

Instructions with an immediate operand, and loads and stores

Opcode	Source reg	Dest Reg	Immediate
Bits 31-26	Bits 25-21	Bits 20-16	Bits 15-0

Instructions with two register operands and a register destination

Opcode	Source 1 reg num	Source 2 reg num	Dest reg num	Shift amount	Funct code
Bits 31-26	Bits 25-21	Bits 20-16	Bits 15-11	Bits 10-6	Bits 0-5

Instructions with a jump target address (not required)

Opcode	Offset applied to PC
Bits 31-26	Bits 25-0

Opcodes and function codes Here are the opcodes and function codes you will need to run MIPS instructions. Ignore the 5-bit "shamt" filed in instructions. All the function codes we will use fit in the 6-bit "funct" field (Textbook figure 2.27). These are the ONLY operations I want you to spend time on, until you get these all running. If you do get them all, and want some more, come and see me for extra credit.

```
ADD   opcode = 0x0, function = 0x20, alu_oper = 0x0, signed = 1
ADDU  opcode = 0x0, function =0x21,  alu_oper = 0x0, signed = 0
ADDI  opcode = 0x8,                  alu_oper = 0x0, signed = 1
ADDUI opcode = 0x9                   alu_oper = 0x0, signed = 0
SUB   opcode = 0x0, function = 0x22  alu_oper = 0x1, signed = 1
SUBI  opcode = 0xa                   alu_oper = 0x1, signed = 1
SUBU  opcode = 0x0, function = 0x23  alu_oper = 0x1, signed = 0
MUL  opcode = 0x0, function = 0xe    alu_oper = 0xe, signed = 1
MULU opcode = 0x0, function = 0x16   alu_oper = 0xe, signed = 0
AND   opcode = 0x0, function = 0x24  alu_oper = 0x2
ANDI  opcode = 0xc                   alu_oper = 0x2
OR    opcode = 0x0, function = 0x25  alu_oper = 0x3
SLT   opcode = 0x0, function = 0x2a  alu_oper = 0xb, signed = 1
SLTU  opcode = 0x0, function = 0x2b  alu_oper = 0x??, signed = 0

LW    opcode = 0x23
SW    opcode = 0x2b
```

If you finish all this and want the extra credit introduce conditional and unconditional branches.

Deliverables

Just as in the previous lab, please turn in the following things for this lab:

- Your VHDL code.
- Transcripts of tests running your simulations
- There's no need for a .do file because all you have to do to set the simulation running is drive the clock with a long enough period that everything settles down before it transitions again (200nS period should be fine)