**Lab 5: PCIMID (or PCIMControl)**

**Introduction:** Let us build the three blocks on the left in Fig. 4.17, p.277 - Program Counter (PC), the Instruction Memory (IM) and the blue oval called *Control* in the text we call it as Instruction Decoder (ID). These blocks together are called the “Control Unit” in some texts. Refer to the very first picture of the computer in our text (p.17) where the processor is divided into Control and Data Path or say in the beginning of chapter 4 (p.255).

The Control Unit’s role is to issue the ***control signals*** to the different parts of the Data Path and to orchestrate the processing of the instruction. We will build each one of the three modules, test it and then integrate and test the bigger module and so on.

**Part 5A: PC**

Adder

4

clock

Adder out

To IM

Fetch & Execute

PC+4

Fetch & Execute

PC+8

PC=0

PC

64

PCOut

PCIn

PC is a register that gets written into with positive edge of the clock and so we can use a modified version of the Register File we created (note: there is only one register!). Many signals of Register File can be omitted. Initially PC will be zero and so the 0th instruction will be fetched from the Instruction memory. That will be decoded and executed and then the clock’s next positive edge will occur so that the next instruction is fetched (it will get executed and the next positive edge …). See Figure on the right. In this picture, the clock is shown with unequal on period and off period. Practical clocks will be similar to this. The ‘duty cycle’ – the ratio of on period to the total period can be as small as 0.1 (10%). Since our circuits use *edge driven* sub systems, we need to apply the edge and then *wait* for the circuit to respond to that edge and change. Then it will be ok to apply the next edge (we need to let the dust settle down).

Depending on the number of clock cycles you need to simulate, you can in the ‘apply stimulus’ section, before the final end, add

“#10 $finish;” // for say, assuming 10 cycles of 1 ns clock // period

//Some versions of Xilinx may not accept this line

The adder module can be implemented by declaring PCIn as input and Adder out as wire and by a statement “assign Adder out = PCOut + 4;”

Simulate and check whether PCOut increments by 4 every time (you may change the radix to decimal).

**Lab 5B: PC+IM**

**Part B1: IM**

64

32

IM

PCOut

InstructionOut

Instruction Memory is Read Only and therefore the module can be simplified compared to the Data memory. We can omit the writing part. *We will have to initialize or populate the memory with instructions.* The task we plan to do later is the following: First we will load registers 5 and 10 from data memory locations 40-47 and 80-87. Then we will be performing the 4 ALU instructions on them (as in a previous Lab) and write back the results in Registers R1, R2, R3 and R4. Then we will store those results of the calculation in data memory locations 1 onward (8-15, 16-23 etc.)

We need to know the formats and opcodes for these instructions (Refer to Text Fig.4.14 (p.274) to fill the instruction memory.

We will use 0 for the base address when we refer to the data memory. Therefore Rn will be 0 (5’b 00000). (In the Legv8, actually R31 is hardwired permanently as zero) We will simulate Rn by **initializing** Rn as all 0s in the RF module.). The instruction format for D type (since we will load R5 and R10 first) is as below.

5

5

Rt

Rn

00

Opcode

Disp

9

2

11

Rn is the base register and Rt is the destination for load. Displacement is signed 9 bit and since we again use a 256 Byte (32 double words and not the possible 16 Exa Bytes - remember our PC is 64 bits), we will assume only positive values above the base and so msb in displacement is 0 and the bits are 00 to FF in hex. For the first instruction of loading register 5, Rt will be 5’b 00101. Memory address is 40 and so the 9 bit displacement is 0 0010 1000. Referring to Figure 4.12 and 4.13, the 4 bit ALU operation must be 0010 for adding the base and displacement. Refer to Lab 2. ALUOp will come from the Instruction Decoder (not yet implemented) and so we need to supply (from the sky) ‘00’. The 11 bit Opcode field for LDUR, according to Figure 4. 12 and 13 pp. 272-273 can be anything (Xs). But in Figure 2.20, p.122 (or the green card stuck at the beginning of text) must be 11111000010. So the 32 bit Instruction in the 0th location in the Instruction Memory will be

32’b 11111000010\_0 0010 1000\_00\_00000\_ 00101// Underscore is harmless, helps us to //recognize the different fields.

Initializing memory is achieved by saying

initial

begin

IM[0] = 32’b 11111000010\_0 0010 1000\_00\_00000\_ 00101; //first load //instruction LDUR X5, [X0, #40] – X0 is base register which has 0, 40 is //offset and destination is register X5

// second load instruction (loading R10 with DM[80 ...] follows.

3rd to 6th instructions will be two arithmetic and two logic operations The instruction format for R type is as below.

shamt

Rm

Opcode

Rt

Rn

11

6

5

5

5

See examples below:

IM[2] = 32’b 10001011000\_01010\_000000\_00101\_ 00001;;// ADD X1, X5, X10

//…

IM[6] = 32’b 11111000000\_0 0000 1000\_00\_00000\_ 00001;

// STUR X1, [X0, #8]

//Three more store follow

end

Did you spot anything wrong? Memories are always byte thin and they are addressed with byte numbers. We are using word numbers above when we are referring to the Instruction Memory locations, not byte numbers as we should. We are saying IM[0]; IM[2], IM[6] etc. Just as we did for Data memory, each 32 bit word of instruction must be sliced into 8 bit bytes and stored (lab 4). Address for the instruction comes from Program Counter as 64 bits but we need to access **four** consecutive bytes, one at a time, starting from that address (as we did eight consecutive bytes in the case of Data Memory). For Data Memory, ALU will give the effective address after adding the base register content and the sign extended value as 64 bits but we access eight consecutive bytes, starting with that number.

Therefore we will **initialize** IM (simulating non-volatility) byte by byte just as we did the DM.

initial

begin

IM[0] = 8’hF8; //remember “big endian” convention, most significant occupies low address

IM[1] = 8’h42;

…

**Part B2: PC+IM**

After populating the Instruction Memory with the required 2 + 4 + 4 instructions, we need to combine the module with the PC module and then let the clock run for 10 or more cycles to see whether the Instruction Memory spits out the instructions in order.

Adder out

PCOut

PCIn

Adder

4

clock

PC

IM

InstructionOut

**Part 5C: PC + IM + ID**

**Part C1: ID:** The Instruction Decoder is responsible to convert the opcode 11 bits and give out the 9 control signals.

9

11

ID

See the table in Fig. 4.18 in p.278 which lists the 9 control signals (7 control signals + the 2 bit ALUOp). If you remember, ALUOp will go along with the 11 bit Opcode to the ALU Control of Lab 2B to give the 4 bit ALU operation - this was given in Figure 4.13, p.273).

To implement the Decoder, we will use the same way we implemented the ALU Control in Lab 2 B - we will have case statements. After declaring 11 bit Opcode as input, we need to declare each of the seven control signals as 1 bit reg and ALUOp as 2 bit reg.

Using Table in Figure 4.18 in p.278, example for LDUR is

always case (Opcode)

begin

11'b 11111000010: begin

Reg2Loc <= 0; ALUSrc <= 1; MemtoReg <= 1; RegWrite <= 1;

MemRead <= 1; MemWrite <= 0; Branch <=0; ALUOp <= 00;

end

…….. // other opcodes and the control signals they need to produce

end

We will omit CBZ instruction. So we will have four R Type Opcodes and one STUR Opcode. We can get the 11 bit Opcode for different cases from Figure 2.20, p.122 or the green card in the front of the book and the seven control signals and ALUOp from Figure 4.18 (p.278).

**Part C2: PCIMID**

Verify that as clock is activated, the correct seven control signals + 2 bit ALUOp come out of the Instruction Decoder for the ten instructions in order.

PCOut

PCIn

Adder

4

clock

11

ID

IM

PC

9

Control Signals

InstructionOut

Looking ahead, in Lab 6, we will combine the ‘Control Unit’ of Lab 5 with the Data Path of Lab 4 and test the whole processor. In Lab 7, we may add the branch facility and test it. I know you just can’t wait to do that.

mc, s17