**Computer Architecture Lab 3 Report**

**Single-Cycle CPU**

정건화 Geonhwa Jeong, CiTE, 20130392

김준영 Junyoung Kim, CiTE, 20140263

1. ***Introduction***

The main goal of this lab is to implement Single-Cycle CPU using Verilog with the datapath and the control unit. Since MIPS is too complicated for beginners, we used TSC ISA which is 16-bit RISC ISA. Since typical CPUs comprise of two components, we decided to follow the typical design using datapath and control unit.

In the design part, we will show which instructions we should be able to support and how to implement briefly. We build various sub-modules so that the full CPU can be understood easily and programmed efficiently. In the implementation part, we will show how we actually implemented and illustrates several issues we confronted.

By finishing this lab, our team was able to fully understand how the Single-Cycle CPU works. By actually implementing them using Verilog, we could figure out how to implement datapath and control unit. Also, we were able to know why dividing into submodules is important when implementing complex program like this.

1. ***Design***

For this assignment, we needed to build CPU which can operate 16 instruction types: 8 for ALU operations, ADI, ORI, LHI, LWD, SWD, BNE, BEQ, and JMP.

ADI $rt, $rs, imm - Save the result of $rs + imm(sign extended to 16 bits) to $rt

ORI $rt, $rs, imm - Save the result of $rs | imm(zero extended to 16 bits) to $rt

LHI $rt, imm - Save the result of immediate value concatenated to 8 bits zeros to $rt

{ imm[7:0], {8{0}} }

LWD $rt, $rs, offset - Load word from memory address $rs + offset(sign extended to 16 bits)

SWD $rt, $rs, offset - Store word to memory address $rs + offset(signed extended to 16 bits)

BNE $rs, $rt, offset - If the contents of $rs and $rt are not equal, pc is changed to target address which is the sum of address of the instruction after branch instruction and the offset (signed extended to 16 bits).

BEQ $rs, $rt, offset - If the contents of $rs and $rt are equal, pc is changed to target address which is the sum of address of the instruction after branch instruction and the offset (signed extended to 16 bits).

JMP offset - A target address is computed by concatenating the four high-order bits of pc with 12-bit offset. pc is written to the target address.

We decided to use six modules to implement the CPU.

The first module is the CPU itself. The CPU connects the control\_unit and data\_path. It also fetches the next instruction from memory according to the current PC value.

The control\_unit is a combinational circuit which takes the instruction as the input and returns a 12-bit output which serves as the microcode for the datapath.

The data\_path connects the three modules register\_file, data\_memory, and the ALU and feeds the needed control signals to them. The data\_path also calculates the next value of PC by checking the instruction and outputs from the modules inside it.

The register\_file contains 4 registers, and updates or reads from them depending on the input.

The data\_memory module writes and reads from memory when needed. It is used mainly for the load and store instructions as fetching instructions is already done by the CPU.

The ALU handles basic operations on 2 inputs such as addition, subtraction, 2’s complementation, etc.

A key point of our design is that we split the clock in half and read the next instruction at the positive edge and read/wrote from/to the memory other than the instruction space at the negative edge. That way, fetching instructions does not interfere with instructions that access memory such as load and store and vice versa.

1. ***Implementation***

First of all, in cpu.v, we implemented cpu with following inputs and outputs.

input ackOuput, inputReady, reset\_n, clk

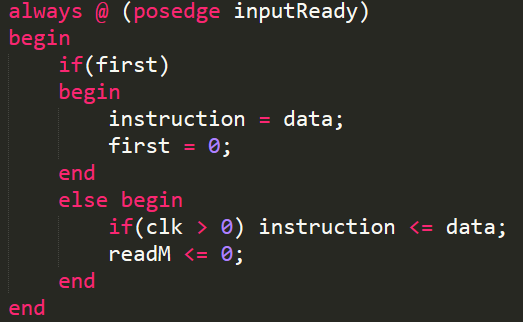
output reg readM, wire writeM, reg [`WORD\_SIZE-1:0] address

inout wire [`WORDSIZE-1:0] data

As designed earlier, we used control\_unit module and data\_path module inside cpu module. In cpu module, PC is updated to nextPC and readM is set by 1 so that it can read an instruction from memory on every positive edge of clk. data is assigned by

assign data =(!clk) ? outputData : `WORD\_SIZE`bz

where outputData is the data from data\_path. instruction is set by data on every positive edge of inputReady. Since the instruction should only be updated on positive clock cycle, we check whether clk is positive when setting data into instruction.



Next, control\_unit module is defined in control\_unit.v file. control\_unit takes [`WORD\_SIZE-1:0] instruction as the input and reg [11:0] signal as output. This module is straightforward. This implements the combinational logic for calculating proper control signals by checking the instruction.

The another module that is used by cpu module is data\_path module which is defined in data\_path.v file. The inputs and outputs of this module are specified in the following.

input wire [`WORD\_SIZE-1:0] inputData, ackOutput, inputReady,

[`WORD\_SIZE-1:0] instruction, [7:0] PC, clk, [11:0] conSignal, reset\_n

output reg readM, writeM, [`WORD\_SIZE-1:0] address, [`WORD\_SIZE-1:0] outputData,

[7:0] nextPC

The control signals that we need can be accessed by conSignal. Signal order is RegDst, Jump, Branch, MemRead, MemtoReg, MemWrite, ALUSrc, RegWrite, ALUOp(4 bits). data\_path module is composed of a register\_file, ALU, data\_memory and some registers. Also this assigns nextPC as the following code.

assign nextPC = Jump ? jumpTarget : ((bcond && Branch) ? PC + 1 + extended : PC + 1);

By the value of RegDst, rd value is chosen between instruction[7:6] and instruction [9:8]. Similarly, the second input for ALU B is chosen by ALUSrc.

The next module that will be explained is data\_memory module which is defined in data\_memory.v file. Inputs and outputs for this module are shown at the following.

input [`WORD\_SIZE-1:0] calc\_address, [`WORD\_SIZE-1:0] writeData,

[`WORD\_SIZE-1:0] inputData, ackOutput, inputReady, MemWrite, MemRead, clk

output reg readM, reg writeM, [`WORD\_SIZE-1:0] address,

[`WORD\_SIZE-1:0] outputData, [`WORD\_SIZE-1:0] readData

In this module, we deal with data memory read and write which is different from instruction memory read and write. We designed to handle data memory read/write on every negative edge of clock. On every negative edge of clock, if MemRead, then set readM as 1 and else if MemWrite, set readM as 0 and writeM as 1. Also, set writeM as 0 on every positive edge of ackOutput so that it doesn’t overwrite wrong value.

The last module to be explained is alu module. We modified this module from the one that we implemented on Lab 1. Followings are inputs and outputs for this ALU module.

input [15:0] A, [15:0] B,

output [15:0] C, reg bcond

We modified slightly so that we can handle some instructions like LHI and TCP. We implemented LHI when OP == 4`b1111. LHI instruction concatenates 8 zeros after the value B. TCP instruction is implemented when OP == 4`b1100. TCP instruction simply find 2’s complement of the given value by calculating ~(A) + 1.

Two files, opcodes.v and cpu\_evaluation\_tb.v files were given. We didn’t change anything, so there is nothing to explain for those files.

1. ***Discussion***

We will briefly explain the difficulties that we met during the assignment and how we handled those problems.

1. Separating instruction memory read/write and data memory read/write

The problem that took that most time was making sure instruction reads did not interfere with memory read/write. To solve this we split the clock into two and read instruction only at the positive edge and made data read/writes only at the negative edge.

To read instructions we wrote the following code

1. ?
2. Using high impedance

A high impedance output is

1. ***Conclusion***

From this assignment, we were able to implement single-cycle CPU with Verilog. We had many challenges during this lab since we were not familiar enough about using Verilog and designing a program in HW Designer’s perspective. From those challenges, we are pretty sure that we learned a lot and ready for next assignments.