CE3001 Lab3. Simple Control and Datapath

Lu Shengliang

SLU001

School of Computer Engineering

Nanyang Technological University

SLU001@e.ntu.edu.sg

October 21, 2013

Abstract

Simple Control and **Datapath** are designed to implement indirectly feeding data inputs to ALU or RF. It is required to perform a sequence of operations by employing an control unit.

1 Introduction

1.1 Control Unit & Datapath Specification

The ADD, SUB, AND and OR instructions have a three address format(Table 1). opCode R_d, R_s, R_t (Execution is $R_d \Rightarrow R_s$ (OP) R_t). R_d is the destination register, and R_s and R_t are the source registers for operand 1 and 2, respectively. The bit-level format for the three-address format is:

	opCode	R_d	R_s	R_t
Index	15-12	11-8	7-4	3-0

Table 1: Instruction Format 1

	opCode	R_d	R_s	**imm**
Index	15-12	11-8	7-4	3-0

Table 2: Instruction Format 2

The SLL, SRL, SRA, and RL instructions have a two address and one immediate format. opcode R_d, R_s, imm (Execution is $R_d \Leftarrow R_s$ (OP) imm). R_d is the destination register, R_s is the source register and imm is used as the shift amount for ALU. The bit-level format for the two address plus immediate format is:

1.2 Design structure & Port list

A block diagram of the expected design is shown in Figure 1.

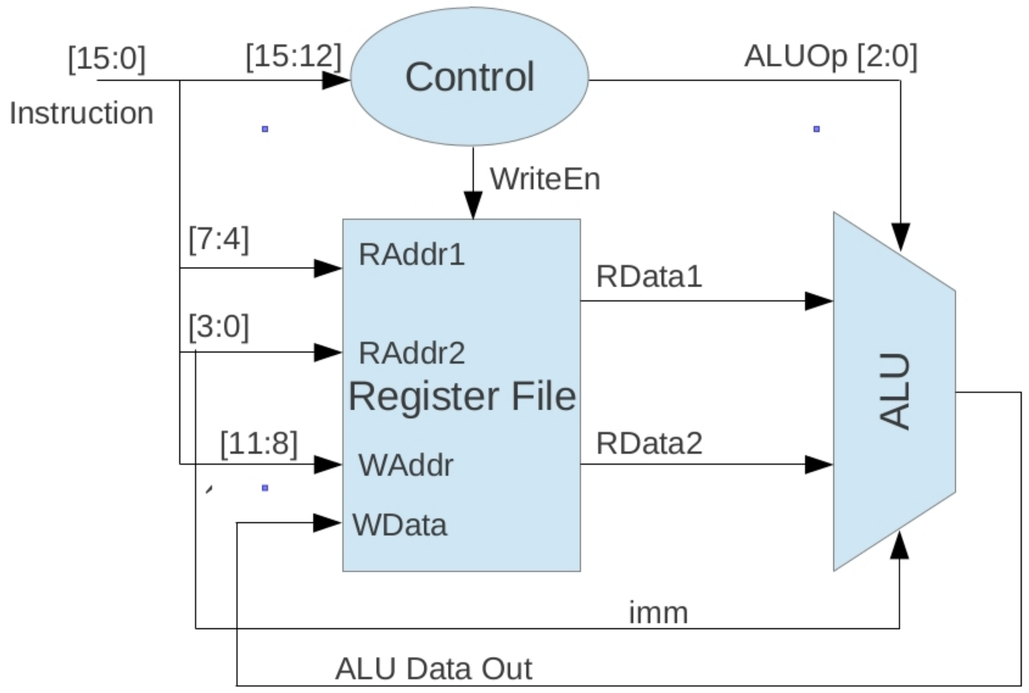

Figure 1: Control Unit and Datapath

And the port name are listed below.

Port Name	Direction	Size	Description
Instruction	Input	16 bit	Instruction word
DataInit	Input	16 bit	Initialization Data
InitSel	Input	1 bit	Select bit from Initialization data
AlUOut	Output	16 bit	Data Output from the ALU

Table 3: Port List Specification

1.3 Structure of the rest of the paper

The rest of the paper first describes the Verilog implementation of **Control Unit** and **Datapath** in *Section 2*. *Section 3* presents the experimental results using *test-bench*, which valid the functionality of our RF. *Section 4* presents our conclusions and discussions.

2 Implementation

2.1 Verilog Code Control.v

```
module Control (input [3:0] ControlInput, output WriteEn, ALUop);
  assign WriteEn = ControlInput[3];
  assign ALUop = ControlInput[2:0];
endmodule
```

Our implementing strategies are using concatenation, which are basic assignments.
The related testbench implementation results will be listed in *Section 3*.

2.2 Verilog Code datapath.v

```
module datapath(input [15:0] Instruction, DataInit,
                input InitSel, input clk, reset,
                output [15:0] ALUOut);

  wire [15:0] WData;
  wire [15:0] RData1, RData2;
  wire [3:0] RAddr1, RAddr2, WAddr;
  wire [2:0] ALUop;
  wire Wen;

  assign  RAddr1 = Instruction[7:4];
  assign  RAddr2 = Instruction[3:0];
  assign  WAddr = Instruction[11:8];
  assign  WData = InitSel ? ALUOut : DataInit;

  Control Con(.ControlInput(Instruction[15:12]),
              .WriteEn(Wen), .ALUop(ALUop));
  Reg_File Reg(.RAddr1(RAddr1), .RAddr2(RAddr2),
               .WAddr(WAddr), .WData(WData),
               .Wen(Wen), .Clock(clk), .Reset(reset),
               .RData1(RData1), .RData2(RData2));
  alu a0(.A(RData1), .B(RData2), .op(ALUop),
         .out(ALUOut), .imm(RAddr2));
endmodule
```

The datapath module is used to manage the *Datapath* for ALU, RF and control Unit. It takes the instruction input and separate it to different ports.

3 Evaluation

3.1 Testbench Code Control_tb.v

```
module Control_tb();
  reg [3:0] control_input;
  wire WriteEn;
  wire [2:0] ALUOp;

  Control C0(
             control_input,
             WriteEn,
             ALUOp
             );

  initial
    begin
      control_input = 0;
      #10 control_input = 4'b1010;
      #10 control_input = 4'b1100;
      repeat (10) begin
       #10 control_input = $random;
      end
      #10 $finish;
    end
endmodule // Control_tb
```

A testbench has been designed to test the *Control.v* file. A simulation function is provided by *ModelSim* software.

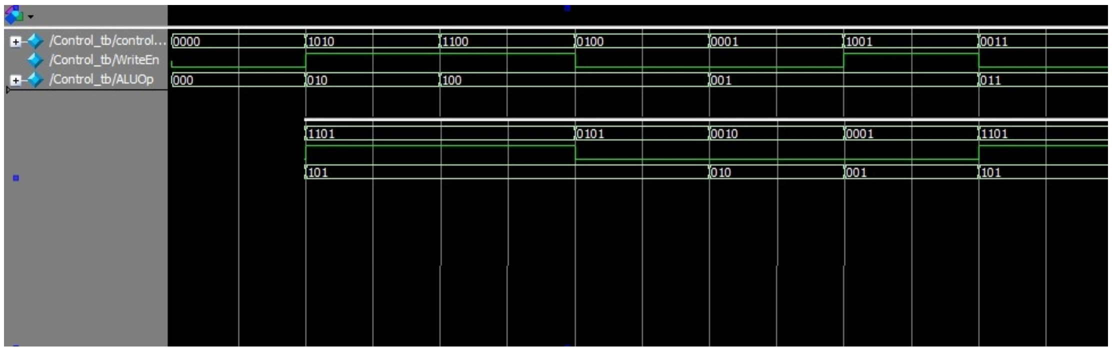

Figure 2: Control_tb results

3.2 First Testbench Code datapath_tb.v

```
module dataPath_tb();
  reg [15:0] Instruction, DataInit;
  reg InitSel, clk, reset;
  wire [15:0] ALUOut;
  datapath dp1(.Instruction(Instruction),
                     .DataInit(DataInit),
                     .InitSel(InitSel),
                     .clk(clk),
                     .reset(reset),
                     .ALUOut(ALUOut)
                     );
  always #5 clk = ~clk;
  initial begin
    reset = 0;
    clk = 0;
    #20 reset = 1;
    repeat (10000) begin
      #10 InitSel = 0;
      {DataInit, Instruction} = $random;
    end
    repeat (15) begin
      {Instruction, DataInit} = $random;
      InitSel = $random;
      #10 $display("Instruction␣=␣%b,␣ALUOut␣=␣%b",
                    Instruction, ALUOut);
    end
    $finish;
  end // initial begin
endmodule // dataPath_tb
```

The testbench firstly ran 10000 times randomly write initial data in to RF(we assumed 10000 should be enough. and actually it did), to make sure RF are full of data rather than *xxxx xxxx*.
Then there were 15 times randomly testing cases, in order to test ALUOut connection with RF, Control and datapath modules.
There is one more testbench provided by lecturer, which will be given below.

3.3 Second Testbench Code datapath_tb_file_io.v

```
`include "datapath.v"
`timescale 1ns / 10ps
`define EOF 32'hFFFF_FFFF
`define NULL 0
`define MAX_LINE_LENGTH 1000
`define ISIZE 16
`define DSIZE 16
module datapath_tb_fileio;
  reg                         clk_half;
  reg                         clk;
  reg                         rst;
  reg                         InitSel;
  reg [`ISIZE-1:0]            Instruction;
  reg [`DSIZE-1:0]            DataInit;
  integer                     file_input, file_output;
  integer                     file_gold, c, r;
  reg [15:0]                  exp;
  reg [8*`MAX_LINE_LENGTH:0] line;
  wire [`DSIZE-1:0]           ALUOut;
  datapath datapath_inst (
                           .clk(clk),
                           .reset(rst),
                           .Instruction(Instruction),
                           .InitSel(InitSel),
                           .DataInit(DataInit),
                           .ALUOut(ALUOut)
                           );
  always #5 clk = ~clk;
  always@(posedge clk)
    clk_half <= ~clk_half;
  initial
    begin
      file_input  = $fopen("input.txt","r");
      file_output = $fopen("output.txt","w");
      file_gold   = $fopen("gold.txt","r");
      clk = 0;
      clk_half =0;
      rst = 1;
      #5 rst = 0;
      #10 rst = 1;
      InitSel = 0;
      while (!$feof(file_input))
        begin
          c = $fgetc(file_input);
          if (c == "/" | c == "#" | c == "%")
            r = $fgets(line, file_input);
          else
            begin
              r = $ungetc(c, file_input);
              r = $fscanf(file_input, "%h %h %b",
                          Instruction, DataInit, InitSel);
            end
          #20; // 20ns for each iteration
        end // while (!$feof(file_input))
      $fclose(file_input);
      $fclose(file_gold);
      $fclose(file_output);
      #100 $finish;
    end // end of initial
  always@(posedge clk_half)
    if (InitSel)
      begin
        $fwrite(file_output, "%h\n", ALUOut);
        r = $fscanf(file_gold, "%h\n", exp);
        if (ALUOut != exp)
          begin
            $fdisplay(file_output, "Error:␣expected:␣%h\n", exp);
          end
        else
          $fdisplay(file_output, "Matched:␣%h", ALUOut);
      end
endmodule // datapath_tb_fileio
```

This testbench reads instruction and data from a file named *input.txt*. And then, it generates the value accordingly and compares it with given correct results which given from file *gold.txt*.

```
//===========
// input.txt|
//===========
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
// file_input.txt
// format:
// Instruction(hex)      DataInit(hex)    InitSel(bin)
// First, initialize the register file
8000    0010    0
8100    0011    0
8200    0012    0
8300    0013    0
8400    0014    0
8500    0015    0
8600    0016    0
8700    0017    0
8800    0018    0
8900    0019    0
8a00    001a    0
8b00    001b    0
8c00    001c    0
8d00    001d    0
8e00    001e    0
8f00    001f    0

// verify alu
8321    xxxx    1
9421    xxxx    1
a521    xxxx    1
b621    xxxx    1
c721    xxxx    1
d821    xxxx    1
e921    xxxx    1
fa21    xxxx    1

//===========
//  gold.txt|
//===========
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
0023
0001
0010
0013
0024
0009
0009
0024

//===========
//output.txt|
//===========
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
0023
Matched: 0023
0001
Matched: 0001
0010
Matched: 0010
0013
Matched: 0013
0024
Matched: 0024
0009
Matched: 0009
0009
Matched: 0009
0024
Matched: 0024
0024
Matched: 0024
```

The testbench is used to generate output and compare it with *gold.txt* file. If everything is matched, then the *datapath.v* itself should be correct.

4 Conclusions and Future Work

The datapath design works properly based on two sets testing results.
For this report, the testbench given has small mistake, which are supposed to be fixed.
The connection of different modules are complicated. It is better if we can draw the diagram out first.