Northeastern University

Department of Electrical and Computer Engineering

EECE2323: **Digital Systems Design Lab**

Lecturer: **Dr. Emad Aboelela**

TAs:
Ke Chen
Linbin Chen

Lab # 4 - 5: **Adding Register File to ALU, Adding Data Memory to the Datapath**

Group # **14**:
Jin Hyeong Kim,
Timothy VanSlyke

Semester: Spring 2018
Date: March 9, 2018
Lab Session: Tuesday, 1:00PM
Lab Location: 9 Hayden Hall, Northeastern University, Boston, MA 02115

Contents

1 Introduction

In these experiments, we will investigate the problem of introducing stateful components to our existing computer system. Specifically, the system will gain persistent storage media in the form of registers and data memory (main memory/random-access memory). In order to support imperitive computation, it is necessary that a computer system provide methods of storing, accessing, and modifying persistent state. Modern CPU architectures typically provide a finite set of registers which may be used to store machine words across the execution of multiple instructions. Additionally, and while not typically considered to be part of the CPU itself, data memory is used to provide a conceptually infinite (but finite in practice) storage medium to the computer system.

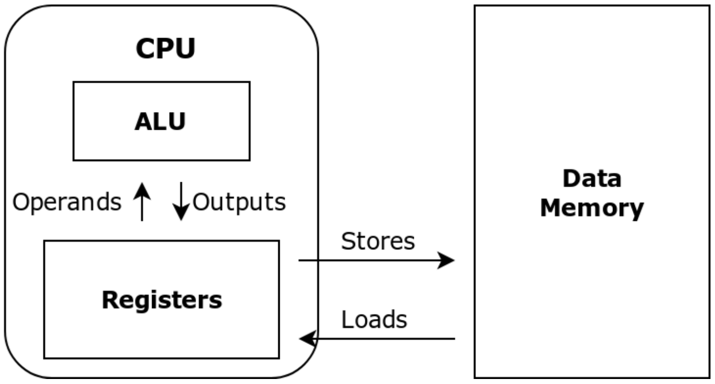


Figure 1: Simplified/minimalist diagram of a computer system with registers and data memory.

A basic memory heirarchy can be implemented by restricting the ALU's operands to be obtained only from registers, while also restricting loads and stores to data memory to only be accessible through registers. In our experiments, we implement this model with the small caveat that linear memory addressing is implmented by using the ALU's output as the address at which loads and stores from and to data memory are done.

2 Design Approach

3 Results and Analysis

3.1 Design Simulation

3.1.1 Testbench Simulation for the Register File

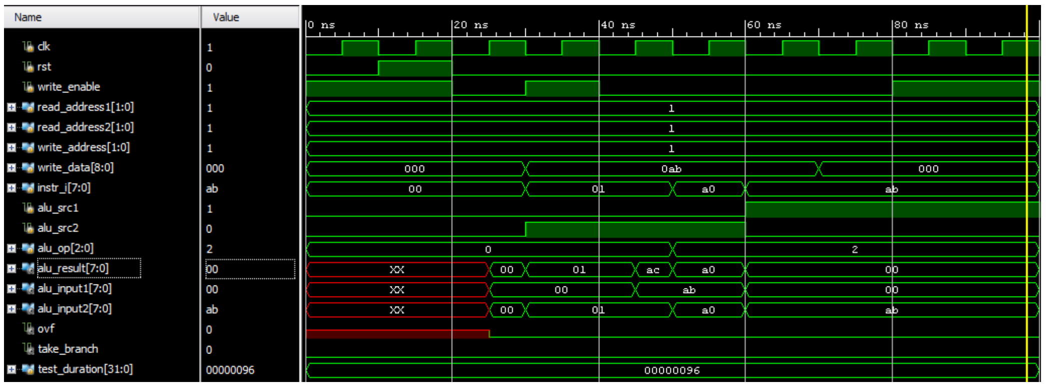
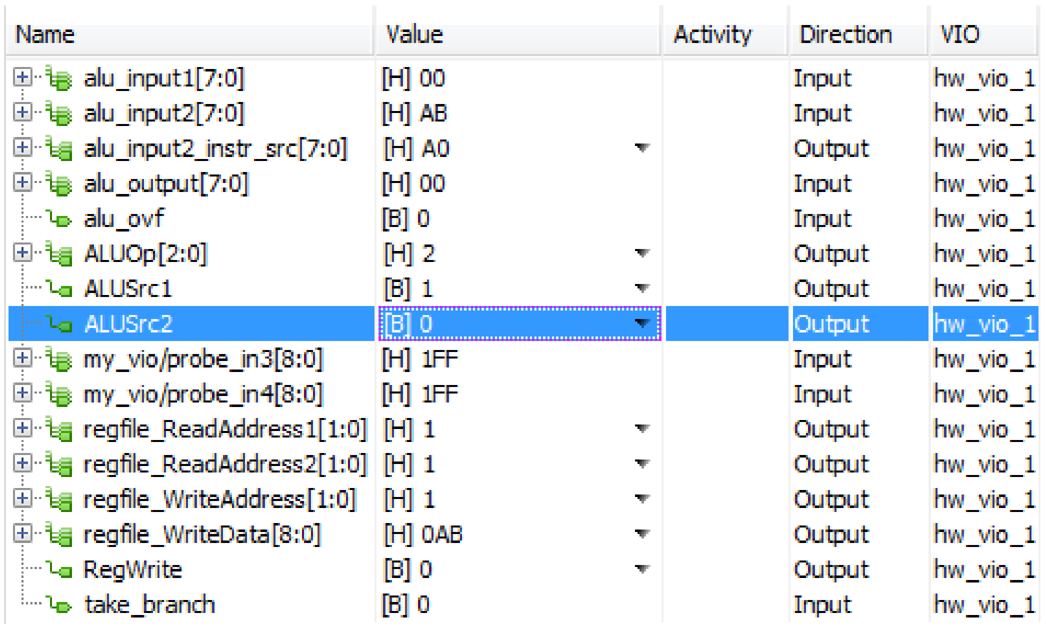


Figure 2: Simulation of the testbench for the register file implementation.

3.2 Hardware Testing

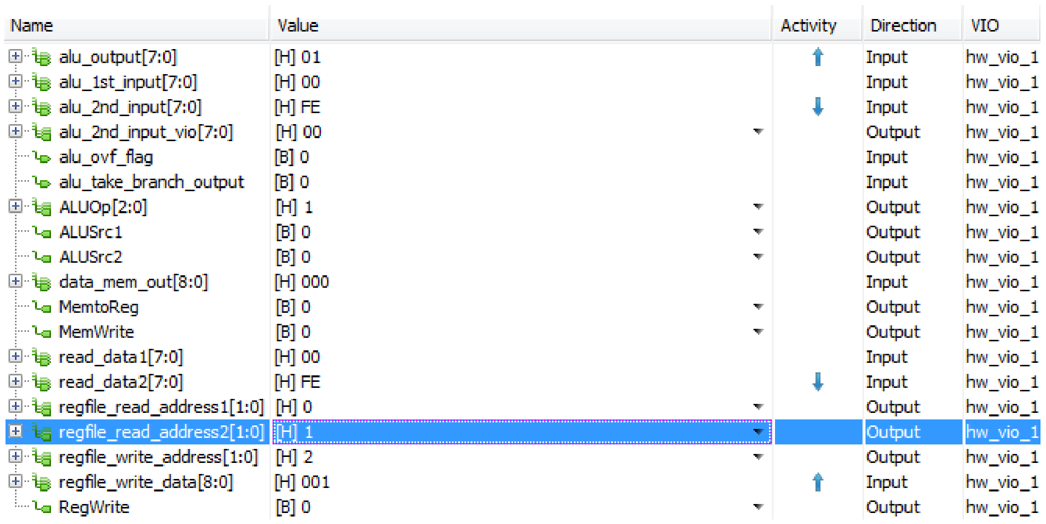
3.2.1 Hardware Test Results for the Register File



Name	Value	Activity	Direction	VIO
alu_input1[7:0]	[H] 00		Input	hw_vio_1
alu_input2[7:0]	[H] AB		Input	hw_vio_1
alu_input2_instr_src[7:0]	[H] A0		Output	hw_vio_1
alu_output[7:0]	[H] 00		Input	hw_vio_1
alu_ovf	[B] 0		Input	hw_vio_1
ALUOp[2:0]	[H] 2		Output	hw_vio_1
ALUSrc1	[B] 1		Output	hw_vio_1
ALUSrc2	[B] 0		Output	hw_vio_1
my_vio/probe_in3[8:0]	[H] 1FF		Input	hw_vio_1
my_vio/probe_in4[8:0]	[H] 1FF		Input	hw_vio_1
regfile_ReadAddress1[1:0]	[H] 1		Output	hw_vio_1
regfile_ReadAddress2[1:0]	[H] 1		Output	hw_vio_1
regfile_WriteAddress[1:0]	[H] 1		Output	hw_vio_1
regfile_WriteData[8:0]	[H] 0AB		Output	hw_vio_1
RegWrite	[B] 0		Output	hw_vio_1
take_branch	[B] 0		Input	hw_vio_1

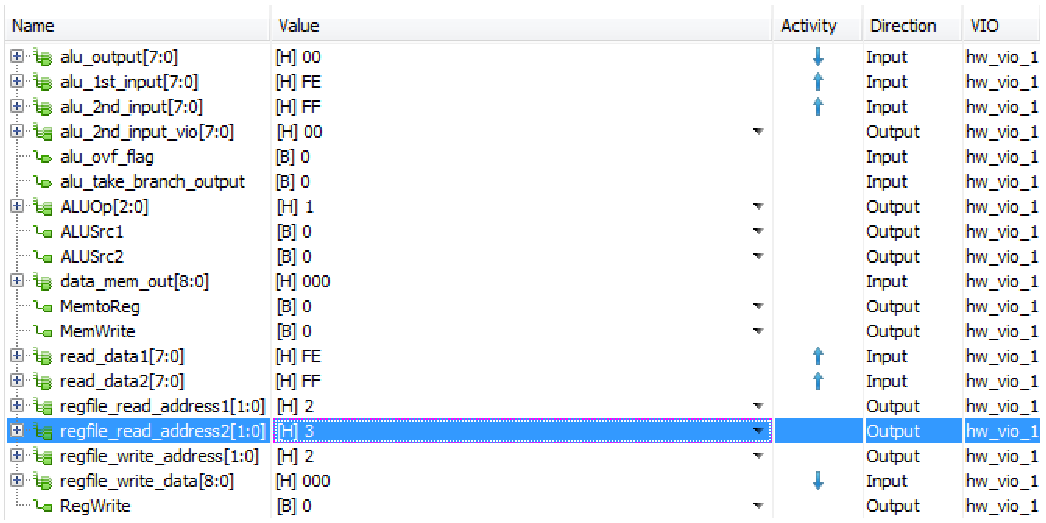
Figure 3: Hardware results for the register file implementation.

3.2.2 Hardware Test Results for the Data Memory Simulation



Name	Value	Activity	Direction	VIO
alu_output[7:0]	[H] 01	↑	Input	hw_vio_1
alu_1st_input[7:0]	[H] 00		Input	hw_vio_1
alu_2nd_input[7:0]	[H] FE	↓	Input	hw_vio_1
alu_2nd_input_vio[7:0]	[H] 00		Output	hw_vio_1
alu_ovf_flag	[B] 0		Input	hw_vio_1
alu_take_branch_output	[B] 0		Input	hw_vio_1
ALUOp[2:0]	[H] 1		Output	hw_vio_1
ALUSrc1	[B] 0		Output	hw_vio_1
ALUSrc2	[B] 0		Output	hw_vio_1
data_mem_out[8:0]	[H] 000		Input	hw_vio_1
MemtoReg	[B] 0		Output	hw_vio_1
MemWrite	[B] 0		Output	hw_vio_1
read_data1[7:0]	[H] 00		Input	hw_vio_1
read_data2[7:0]	[H] FE	↓	Input	hw_vio_1
regfile_read_address1[1:0]	[H] 0		Output	hw_vio_1
regfile_read_address2[1:0]	[H] 1		Output	hw_vio_1
regfile_write_address[1:0]	[H] 2		Output	hw_vio_1
regfile_write_data[8:0]	[H] 001	↑	Input	hw_vio_1
RegWrite	[B] 0		Output	hw_vio_1

Figure 4: Hardware results for the data memory implementation. This image shows the desired contents of the first two registers after running the test instruction sequence in the second experiment.



Name	Value	Activity	Direction	VIO
alu_output[7:0]	[H] 00	↓	Input	hw_vio_1
alu_1st_input[7:0]	[H] FE	↑	Input	hw_vio_1
alu_2nd_input[7:0]	[H] FF	↑	Input	hw_vio_1
alu_2nd_input_vio[7:0]	[H] 00		Output	hw_vio_1
alu_ovf_flag	[B] 0		Input	hw_vio_1
alu_take_branch_output	[B] 0		Input	hw_vio_1
ALUOp[2:0]	[H] 1		Output	hw_vio_1
ALUSrc1	[B] 0		Output	hw_vio_1
ALUSrc2	[B] 0		Output	hw_vio_1
data_mem_out[8:0]	[H] 000		Input	hw_vio_1
MemtoReg	[B] 0		Output	hw_vio_1
MemWrite	[B] 0		Output	hw_vio_1
read_data1[7:0]	[H] FE	↑	Input	hw_vio_1
read_data2[7:0]	[H] FF	↑	Input	hw_vio_1
regfile_read_address1[1:0]	[H] 2		Output	hw_vio_1
regfile_read_address2[1:0]	[H] 3		Output	hw_vio_1
regfile_write_address[1:0]	[H] 2		Output	hw_vio_1
regfile_write_data[8:0]	[H] 000	↓	Input	hw_vio_1
RegWrite	[B] 0		Output	hw_vio_1

Figure 5: Hardware results for the data memory implementation. This image shows the desired contents of the last two registers after running the test instruction sequence in the second experiment.

4 Conclusions

Appendices

A reg_file.v

```
‘timescale 1ns / 1ps

module reg_file #( parameter WIDTH = 9, DEPTH = 4 )(
  input rst,
  input clk,
  input wr_en,
  input [1:0] rd0_addr,
  input [1:0] rd1_addr,
  input [1:0] wr_addr,
  input [8:0] wr_data,
  output wire [8:0] rd0_data,
  output wire [8:0] rd1_data
);
  reg [WIDTH-1:0] storage [0:DEPTH-1];

  always @(posedge clk) begin
    if(rst) begin
      storage[0] <= 0;
      storage[1] <= 0;
      storage[2] <= 0;
      storage[3] <= 0;
    end
    else if(wr_en)
      storage[wr_addr] <= wr_data;
  end
  assign rd0_data = storage[rd0_addr];
  assign rd1_data = storage[rd1_addr];
endmodule
```

Figure 6: `reg_file.v` - 4×9 register file implementation in verilog.

B eightbit_alu.v

```
‘timescale 1ns / 1ps

module eightbit_alu(
    input signed [7:0] a,
    input signed [7:0] b,
    input [2:0] s,
    output reg [7:0] f,
    output reg ovf,
    output reg take_branch
  );
  always @(a, b, s) begin
    // operations on ’f’
    case(s)
      0: f = a + b;
      1: f = ~b;
      2: f = a & b;
      3: f = a | b;
      4: f = a >>> 1;
      5: f = a << 1;
      6: f = 0;
      7: f = 0;
    endcase
    // overflow check for addition
    ovf = (s == 0) && (f[7] != a[7]) && (f[7] != b[7]);
    // ’take_branch’ special cases
    take_branch = ((s == 6) && (a == b)) || ((s == 7) && (a != b));
  end
endmodule
```

Figure 7: eightbit_alu.v - 8-bit ALU in implementation in verilog.