Collin Clark
Zachary Irons
CPEG 324
April 30, 2018
Lab 3 Report

Abstract
Several designs were implemented for this single-cycle calculator including a register file comprised of 8-bit shift registers and an ALU comprised of an 8-bit full adder. All designs are implemented in gate-level RTL designs using synthesizable VHDL constructs such that they could be used to program FPGA chips with these designs. All of our designs ran conclusive test benches for corner cases and general case problems.

Division of Labor
Zach and Collin collaborated on the design of the single-cycle calculator. Most of the work for this lab involved combining existing components from previous labs into larger more complicated components. Then, test benches had to be written for those larger components. Collin and Zach each wrote VHDL code for several components and each wrote extensive test benches.

Detailed Strategy
All figures referenced in this section can be found in Appendix IV at the end of this report.
Zach and Collin began this project by designing the calculator. Once the component architecture was conceptualized the actual VHDL could be completed in a more organized manner. As seen in Figure 1 the design of the calculator involves several modules that interface with a larger structure. To start, the controller will be explained.

The controller, seen in Figure 2, is a module that generates all of the control signals for the calculator. All the selects for the multiplexers and all the enables for the modules come from the controller. Each of these signals will now be explained.

Enable_write is the enable for the register file. The registers can only be written to when it is on. It is the result of NANDing the I(7) and I(6) of the input signal. This means **enable_write** will be on for add, subtract, and load instructions only.

Mux_in is a multiplexer select for reading the registers. Due to the design of our ISA the location of the source registers address in the instruction differ depending on the instruction's opcode. When I(7) is on, **mux_in** will select the correct bits of the instruction to read the registers. **Mux_in** is on for load, compare, and display instructions, but off for add and subtract.

Add_sub is the control signal that changes the operation done by the ALU. **Add_sub** is on for add instructions and off for subtract instructions.

Enable_output is an enable that controls the module that prints things to the terminal. It is the result of ANDing I(7), I(6), and I(5). All three of these bits are on for only the display instruction.

Load is a multiplexer select signal that decides what is being loaded into the register file. It is only on when a load instruction is being executed. This is because load instructions load immediate values into the register instead of values from the ALU.

Offset is a multiplexer signal that chooses from two 4-bit numbers to be inputted into the skip module. On compare instructions the last bit, I(0), is the offset bit. When **offset** is 0 one instruction is skipped. Two instructions are skipped when **offset** is 1.

Compare is a control signal used to enable the skip module. **Compare** is on when the instruction's opcode is "110," the opcode for the compare instruction. The skip module receives the output from the ALU but will not skip any instructions unless **compare** is on.

The sign extend module is seen in Figure 3. It is a fairly simple module used only during load instructions. For load instructions, the last 4 bits represent an immediate value that is to be written to the destination register specified with the 5th and 6th bits. However, the registers for this calculator store 8-bit integers so the 4-bit immediate value must be sign extended. The module copies the significant bit of the immediate and propagates it to the four most significant bits of a new 8-bit integer signal called **sign_ext**. This value is multiplexed with the **load** signal to be input into the writeback value port of the register file.

The instruction skip module is seen in Figure 4. Before it can be discussed, several internal signals need to be addressed. When the compare instruction is executed the resulting bits from the ALU are NOR'd together. If all the bits are zero, the resulting value will be zero. If any one of the bits at all is a '1', the resulting value will be '1'. This value is AND'd with the **compare** signal to determine if the skip module should execute. This signal is then AND'd with **skip** signal generated by the skip module. This disables the skip module if a skip instruction is being skipped. The resulting signal is called **skip_enable**. The **offset** multiplexer signal also decides how many instructions to skip. The result of that multiplexer is the signal **skip_amount**.

The actual instruction skip module is based on the 4-bit shift register. The input value is the **skip_amount**, sel(0) is decided by **skip_enable**, sel(1) is always '1', and the I_SHIFT_IN is always '0.' The enable for the shift register is also always on. This means that when the **skip_enable** is on, the register is loading the **skip_amount**, and when it is off, the register is shifting it to the right. The bits output of the shift register are OR'd together. If any of the bits are '1' the **skip** signal is set to '1' and when all the bits are zero, so will **skip**. The 4-bit shift register always outputs the value of the register so when the **skip_amount** is being loaded it will also output. The two options for **shift_amount** are '0010' and '0001' representing skipping 2 instructions and skipping 1 instruction. When **skip_enable** is off the previous value of **skip_amount** is shifted right. When **skip_amount** is '0010' the signal **skip** that determines if instructions will be skipped is set to '1' for 2 instruction executions. The same is true for '0001'

except **skip** is ‘1’ for only one instruction. The inverse of the **skip** signal is AND’d with all the enables for all the important modules in the calculator, effectively disabling those components if **skip** is ‘1.’

The register file is the most important component of the calculator and it can be seen in Figure 5. It stores the values and allows the CPU to read and write to the registers it encompasses. It takes the address to two source registers, the address of the destination register, and an 8-bit writeback value. The register addresses are determined by the 8-bit instruction. Because certain instructions have different sized opcodes, the location of the source register addresses differs as well. The **mux_in** signal discussed in the controller section chooses which bits to slice off and feed to the register file. The destination register address is always in the same place. The writeback value must be determined from two possible signals. If the instruction is a load instruction, the writeback value is the output of the sign extend module. Otherwise, it is the result of the ALU. The **load** control signal decides this.

The internal components of the register file are four 8-bit shift registers whose inputs and outputs are determined by several multiplexers and demultiplexers. As seen in the schematic, the destination address is used as the demultiplexer control signal in deciding into which register the writeback value is loaded as well as determining the select value for the same register. The select values for all the registers are always set to hold whereas when a value is being written the select is set to load. This is done by making the input to the select demultiplexer equal to ‘11,’ the shift register’s value for load. The shift registers are always outputting the value they have stored. These outputs are the inputs to two multiplexers whose control signals are the addresses of the two source registers. When the CPU tries to read the value from these two registers, the value they contain is chosen as the output for the **v1** or **v2** signal respectively.

The enable for the register file is determined by the result of ANDing **enable_write** and the inverse of **skip**. This means that the register file can be read/written to only when an add, sub, or load instruction is being executed and instruction are not currently being skipped.

The ALU is simply a 8-bit full adder that takes the **add_sub** signal from the controller to determine whether addition or subtraction must be done. It outputs an 8-bit value that sent to be written into the register file. This output also affects the instruction skip module.

The print module is connected to the register file by the output **v1**. It takes that value as an input and outputs that same value when enable is on. It also displays this value to the terminal. The print module’s enable is determined by ANDing the **enable_output** signal and once again the inverse of **skip**. If the **enable_output** signal is off, the print module will assign output to zero.

The following section will discuss the various text benches used to verify the correctness of the VHDL components. Appendix V is the output in the terminal when the make command is executed. This section will explain why this proves the calculator works and is successful.

In the zip file of all the VHDL files is a folder called benchmarks. This contains all the instruction tests and results from Lab 1 in which this same calculator was implemented in Python. These tests are separated for each individual instruction type. The ".asi" files contain the actual 8-bit instructions, commented to show in common MIPS notation what these instructions do. The ".res" files show what the output of these files should be. Remember, values are only output when a display instruction is executed.

Test benches were also written for each individual instruction type to help organize the testing process. If the ".res" files are compared to the output of the test bench of the corresponding instruction type it will be noticed that the outputs are identical. This verifies the correctness of the VHDL implementation.

Results

All tests pass to expected values. The cumulative test bench is broken into four separate files, one to test each of the following: add, subtract, load, and compare instructions.

The load testbench loads several different types of immediate values, both positive and negative, to ensure sign extension works correctly, and prints the values to the console.

The add testbench sums positive and negative values, as well as summing to overflow and summing negative numbers to underflow. All correctly calculated values are returned by the calculator, including over and underflow values.

The subtract testbench follows the exact same pattern as the add testbench, but in reverse. All values are correct and can be audited from the benchmarks directory.

The compare testbench loads several register values and compares both successfully and unsuccessfully. Skipped instructions include altering register values and printing to console, which do not appear in the output, as expected. All logic is commented both in the benchmark file and the cpu testbench file for easy auditing.

All tests pass. Additional testbenches are included for each discrete module and can be run using the `make all` command in the associated makefile. All testing directions are included in the readme.md file bundled with the VHDL source code.

Conclusion

After several revision cycles, we are happy with the design described in this report and implemented in the attached VHDL code. Several revisions, both major and minor, took place between the initial design presentation and the final implementation. Among those are: modification of registers to include combinational reads, redesign of the output signal for testbench evaluation, separation of cpu testbench into several discrete benchmarks, and changing the on signal for our instruction skip module.

Due to the timing of our clock signal, registers were changed to combinational reads so that all signals would propagate to the correct ports by the time the clock hit a rising edge.

The output signal originally had a design error that would output the value from the ALU but did not display anything of significance on non-ALU functions. This was moved to the print module so the design could be programmatically evaluated at the same time to great success.

The cpu benchmark had become rather long and tedious to debug, and so was broken into several separate files for each instruction type. This approach is modeled after the design used in the first lab to ease test writing and concentrate errors on a specific area of the datapath.

The skip module would occasionally trigger during arithmetic instructions when the resulting value was zero. To avoid this, a new controller signal was added to ensure the instruction opcode matched the behavior of the skip module.

No obvious functional errors are present in the included design to the extent of our knowledge. All testbenches are working as described from lab 1.

Appendix I: Notebooks

Collin:

- Individual time spent: 6 hrs
- Group time spent: 13.5 hrs

Zach:

- Individual time spent: 2 hrs
- Group time spent: 13.5 hrs

Times are calculated using git commit timestamps for the duration of the project.

Appendix II: VHDL Files

adder_8_bit.vhdl
controller.vhdl
cpu.vhdl
demux_1_4.vhdl
dff.vhdl
full_adder.vhdl
instruction_skip.vhdl
mux_2_1.vhdl
mux_4_1.vhdl
print_module.vhdl
reg_file.vhdl

shift_reg.vhdl
shift_reg_8_bit.vhdl
sign_extend.vhdl

Appendix III: Testing
controller_tb.vhdl
cpu_add_tb.vhdl
cpu_comp_tb.vhdl
cpu_load_tb.vhdl
cpu_sub_tb.vhdl
demux_1_4_tb.vhdl
instruction_skip_tb.vhdl
print_module_tb.vhdl
reg_file_tb.vhdl
shift_reg_8_bit_tb.vhdl
shift_reg_tb.vhdl
sign_extend_tb.vhdl

Appendix IV: Figures
Figure 1: Schematic

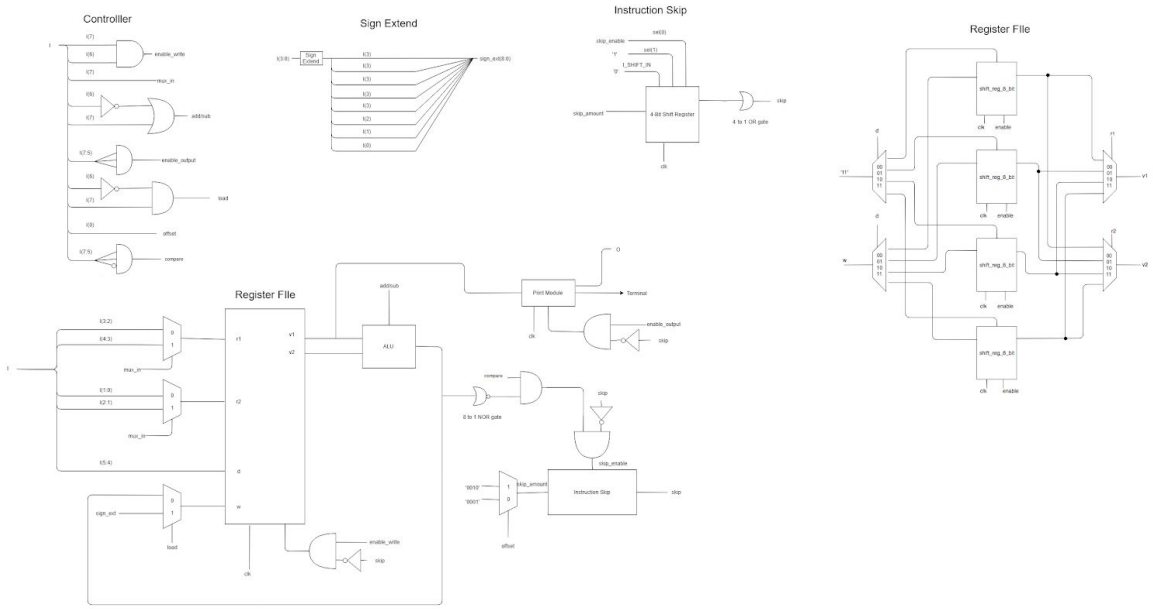


Figure 2: Controller

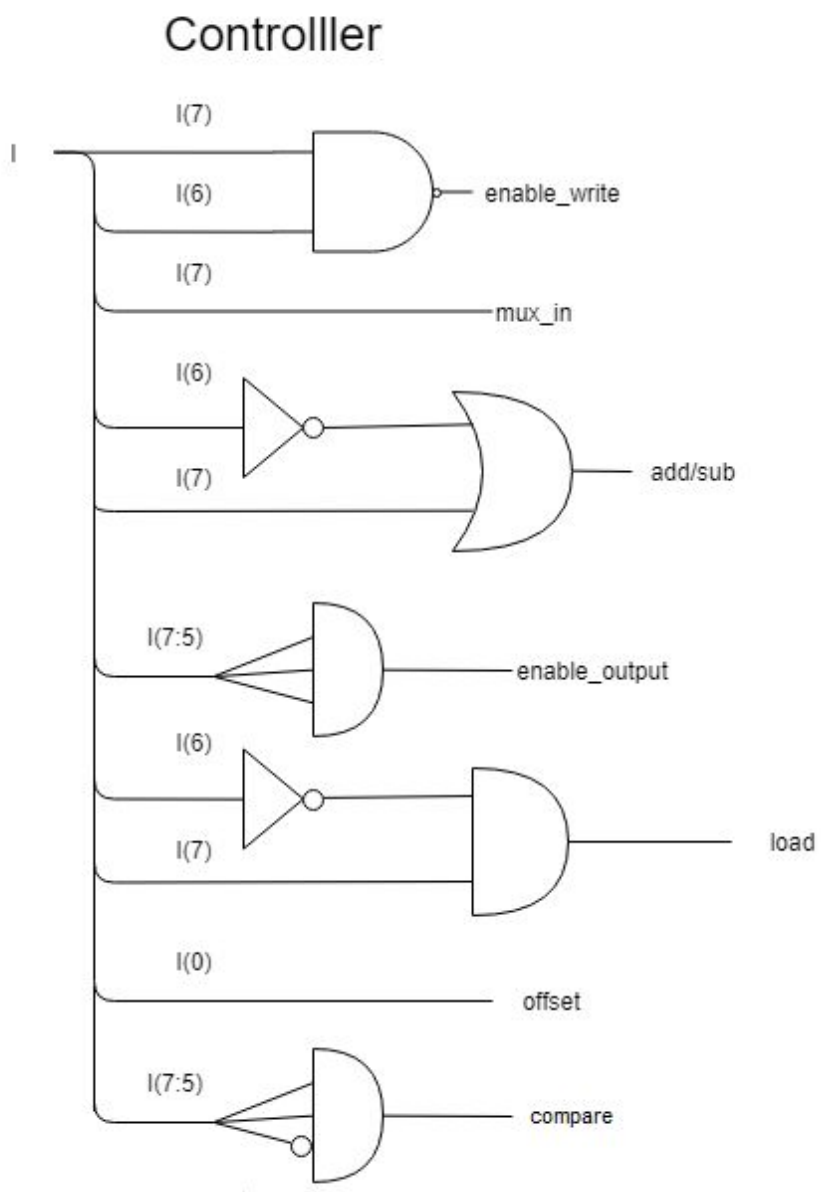


Figure 3: Sign Extend Module

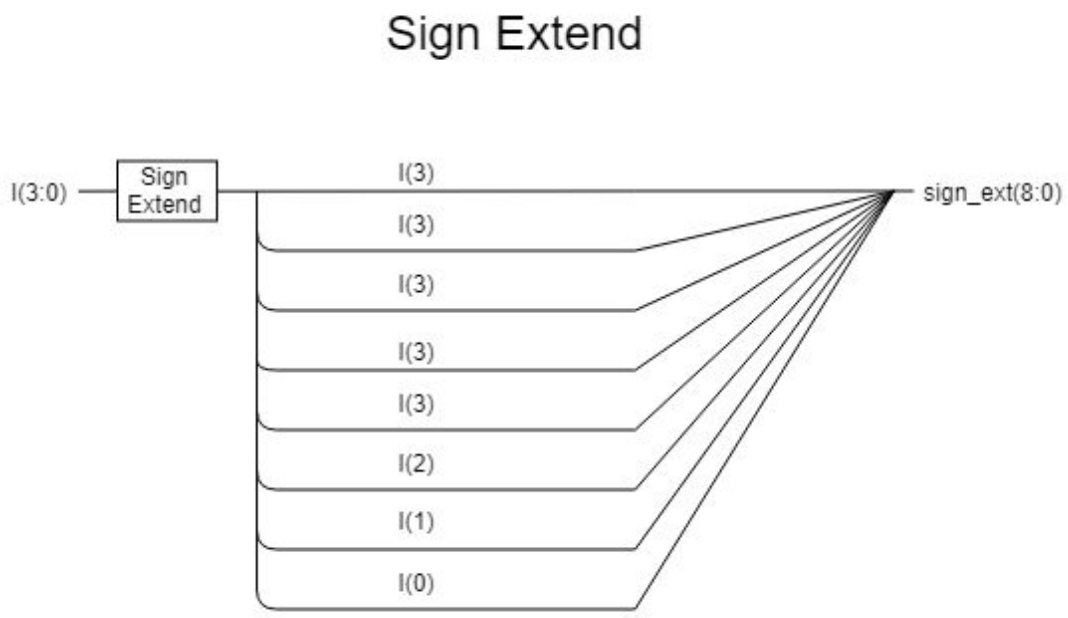


Figure 4: Instruction Skip Module

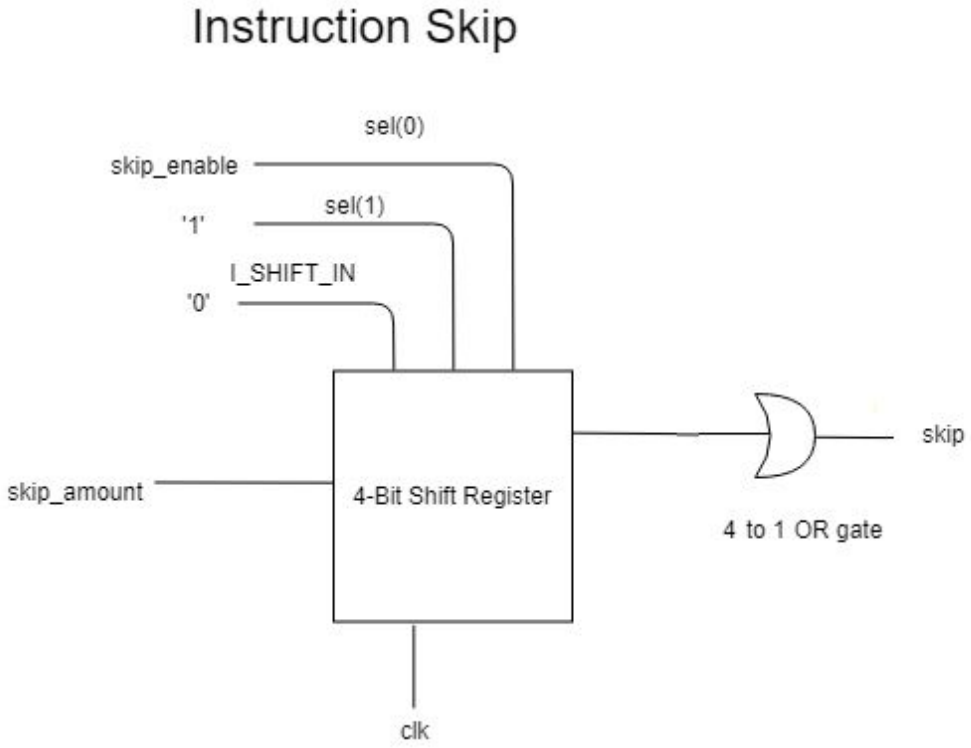


Figure 5: Register File

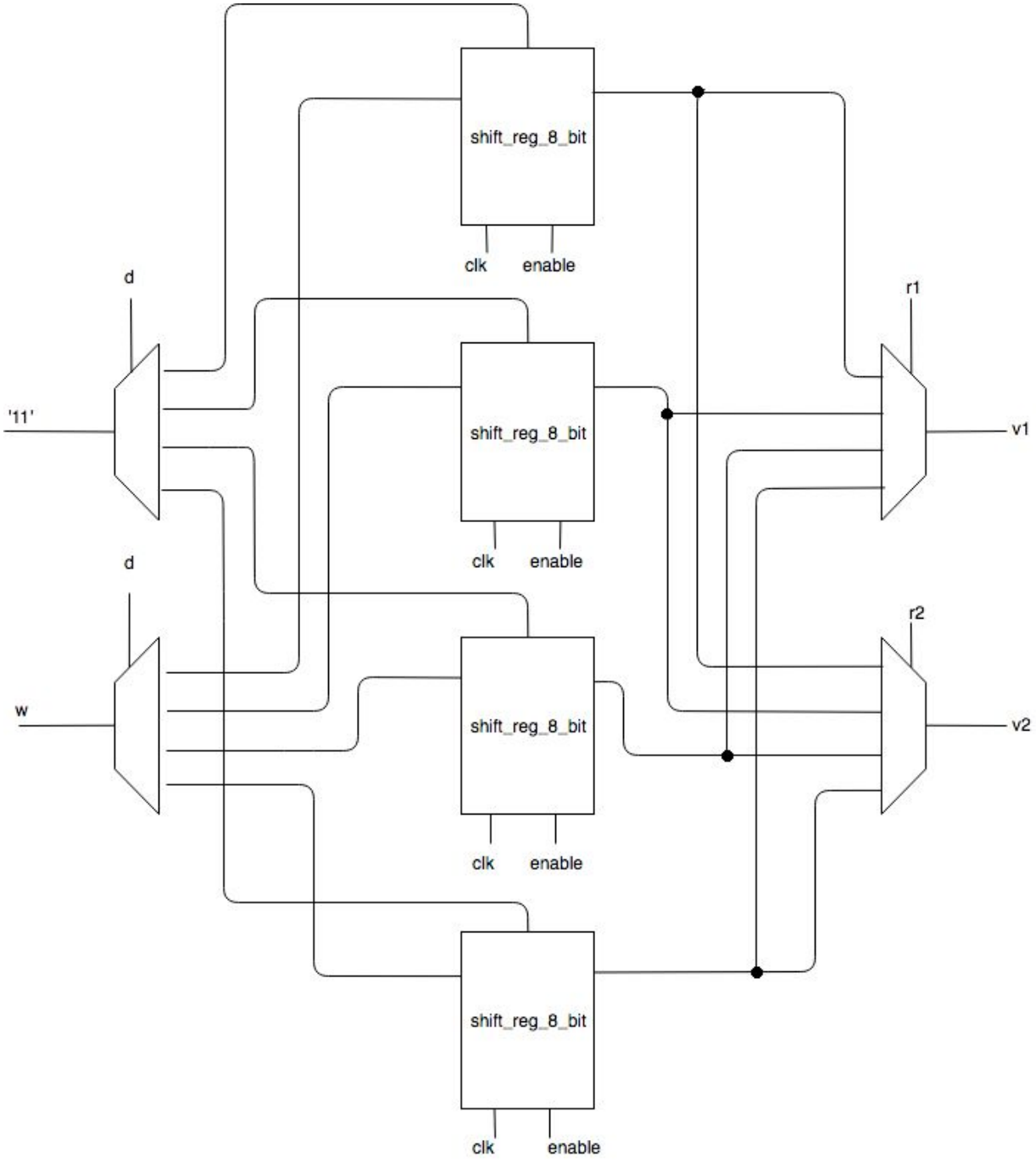
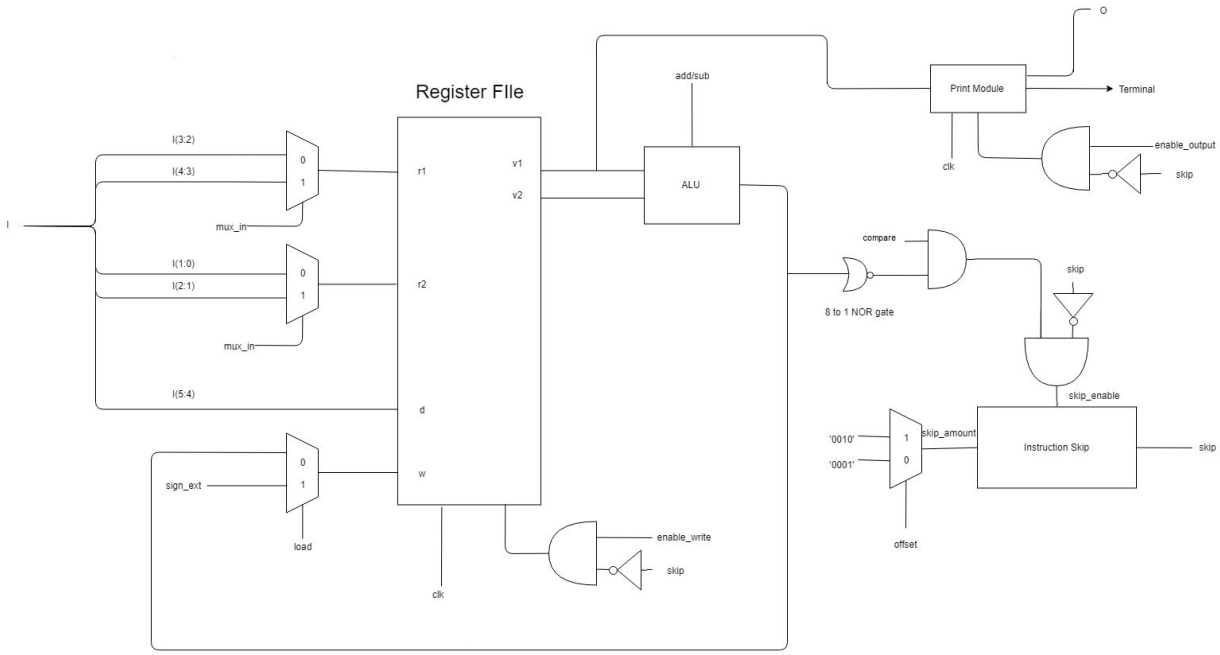


Figure 6: Datapath



Appendix V: Output

```
ghdl -a reg_file.vhdl
ghdl -a shift_reg_8_bit.vhdl
ghdl -a shift_reg.vhdl
ghdl -a dff.vhdl
ghdl -a mux_4_1.vhdl
ghdl -a demux_1_4.vhdl
ghdl -a instruction_skip.vhdl
ghdl -a cpu.vhdl
ghdl -a controller.vhdl
ghdl -a sign_extend.vhdl
ghdl -a print_module.vhdl
ghdl -a mux_2_1.vhdl
ghdl -a adder_8_bit.vhdl
ghdl -a full_adder.vhdl
ghdl -a cpu_load_tb.vhdl
ghdl -e cpu_load_tb
ghdl -r cpu_load_tb --vcd=cpu_load_tb.vcd
../../src/ieee/numeric_std-body.v93:2124:7:@0ms:(assertion warning):
NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
print_module.vhdl:55:9:@1ns:(report note): value: 0
print_module.vhdl:55:9:@3ns:(report note): value: 0
print_module.vhdl:55:9:@5ns:(report note): value: 0
print_module.vhdl:55:9:@7ns:(report note): value: 0
print_module.vhdl:55:9:@17ns:(report note): value: -1
print_module.vhdl:55:9:@19ns:(report note): value: 1
print_module.vhdl:55:9:@21ns:(report note): value: 7
```

```
print_module.vhdl:55:9:@23ns:(report note): value: -8
cpu_load_tb.vhdl:76:5:@24ns:(assertion note): end of test
ghdl -a cpu_sub_tb.vhdl
ghdl -e cpu_sub_tb
ghdl -r cpu_sub_tb --vcd=cpu_sub_tb.vcd
../../src/ieee/numeric_std-body.v93:2124:7:@0ms:(assertion warning):
NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
print_module.vhdl:55:9:@1ns:(report note): value: 0
print_module.vhdl:55:9:@3ns:(report note): value: 0
print_module.vhdl:55:9:@5ns:(report note): value: 0
print_module.vhdl:55:9:@7ns:(report note): value: 0
print_module.vhdl:55:9:@17ns:(report note): value: 1
print_module.vhdl:55:9:@19ns:(report note): value: -1
print_module.vhdl:55:9:@21ns:(report note): value: 7
print_module.vhdl:55:9:@23ns:(report note): value: -8
print_module.vhdl:55:9:@27ns:(report note): value: 6
print_module.vhdl:55:9:@31ns:(report note): value: -5
print_module.vhdl:55:9:@35ns:(report note): value: -7
print_module.vhdl:55:9:@39ns:(report note): value: 6
print_module.vhdl:55:9:@43ns:(report note): value: 7
print_module.vhdl:55:9:@47ns:(report note): value: -15
print_module.vhdl:55:9:@51ns:(report note): value: 16
print_module.vhdl:55:9:@55ns:(report note): value: -31
print_module.vhdl:55:9:@59ns:(report note): value: 47
print_module.vhdl:55:9:@63ns:(report note): value: -78
print_module.vhdl:55:9:@67ns:(report note): value: 125
print_module.vhdl:55:9:@71ns:(report note): value: 53
print_module.vhdl:55:9:@75ns:(report note): value: -53
cpu_sub_tb.vhdl:128:5:@76ns:(assertion note): end of test
ghdl -a cpu_add_tb.vhdl
ghdl -e cpu_add_tb
ghdl -r cpu_add_tb --vcd=cpu_add_tb.vcd
../../src/ieee/numeric_std-body.v93:2124:7:@0ms:(assertion warning):
NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
print_module.vhdl:55:9:@1ns:(report note): value: 0
print_module.vhdl:55:9:@3ns:(report note): value: 0
print_module.vhdl:55:9:@5ns:(report note): value: 0
print_module.vhdl:55:9:@7ns:(report note): value: 0
print_module.vhdl:55:9:@17ns:(report note): value: 1
print_module.vhdl:55:9:@19ns:(report note): value: -1
print_module.vhdl:55:9:@21ns:(report note): value: 7
print_module.vhdl:55:9:@23ns:(report note): value: -8
print_module.vhdl:55:9:@27ns:(report note): value: 2
print_module.vhdl:55:9:@31ns:(report note): value: 9
print_module.vhdl:55:9:@35ns:(report note): value: 16
print_module.vhdl:55:9:@39ns:(report note): value: 25
print_module.vhdl:55:9:@43ns:(report note): value: 41
print_module.vhdl:55:9:@47ns:(report note): value: 66
```

```
print_module.vhdl:55:9:@51ns:(report note): value: 107
print_module.vhdl:55:9:@55ns:(report note): value: -83
print_module.vhdl:55:9:@59ns:(report note): value: -91
print_module.vhdl:55:9:@63ns:(report note): value: 82
print_module.vhdl:55:9:@67ns:(report note): value: -9
print_module.vhdl:55:9:@71ns:(report note): value: 98
cpu_add_tb.vhdl:124:5:@72ns:(assertion note): end of test
ghdl -a cpu_comp_tb.vhdl
ghdl -e cpu_comp_tb
ghdl -r cpu_comp_tb --vcd=cpu_comp_tb.vcd
../../src/ieee/numeric_std-body.v93:2124:7:@0ms:(assertion warning):
NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
print_module.vhdl:55:9:@21ns:(report note): value: -1
print_module.vhdl:55:9:@25ns:(report note): value: -1
print_module.vhdl:55:9:@27ns:(report note): value: -1
cpu_comp_tb.vhdl:84:5:@32ns:(assertion note): end of test
rm controller.o full_adder.o sign_extend.o mux_2_1.o cpu_load_tb.o cpu_sub_tb.o
cpu_comp_tb.o print_module.o cpu.o adder_8_bit.o cpu_add_tb.o
```