Patrick Hoey
Advanced Digital Systems
EECE.5500
Final Project Report

CPE CPU – A RISC-V Soft Core

System Overview

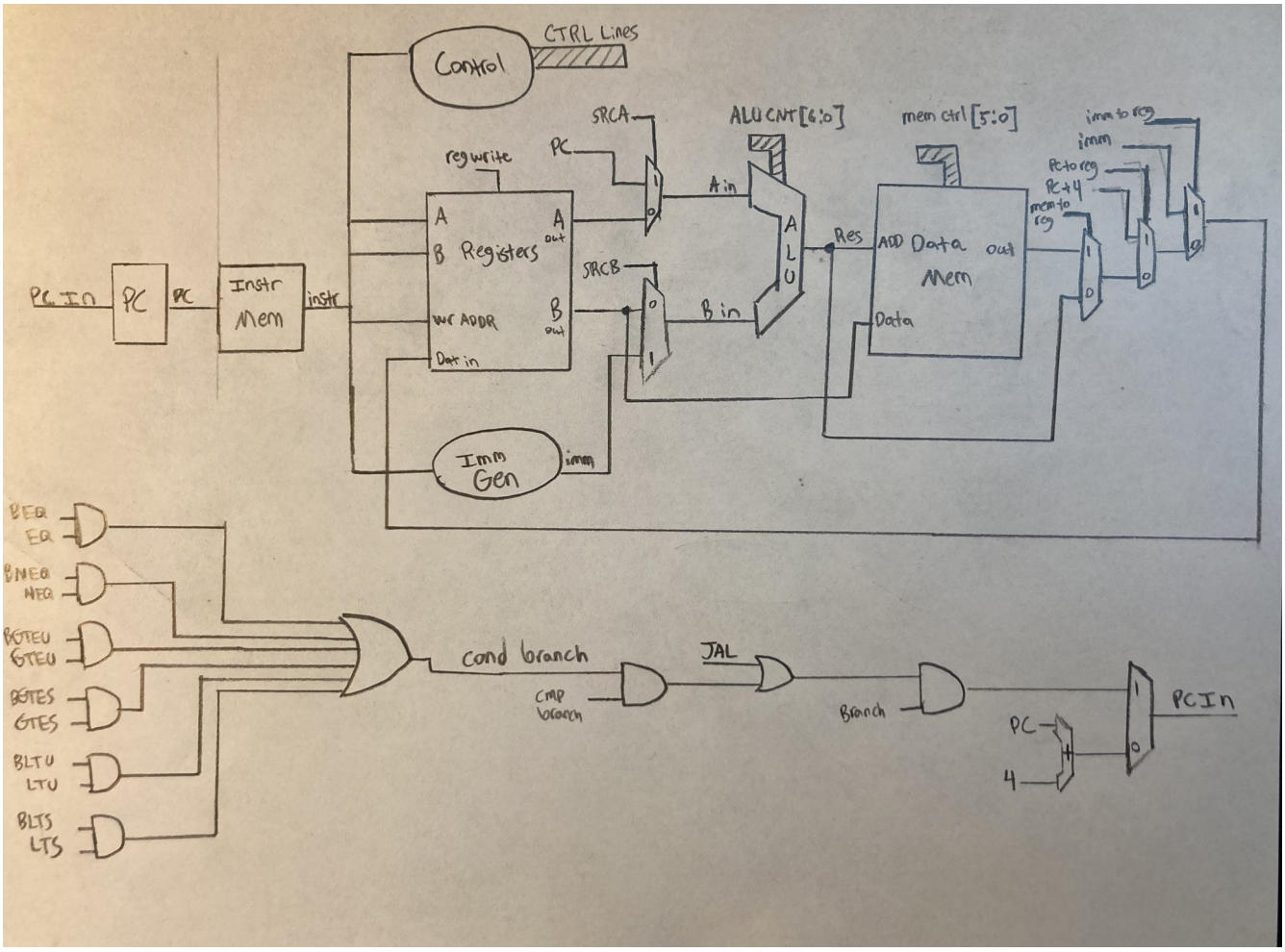
The CPE CPU is a soft core CPU implementing the RISC-V RV32I instruction set. The RV32I ISA is a fully open source instruction set that is considered the baseline for all RISC-V implementations, and compilation for this ISA is fully supported by GNU's gcc and the associated linking and assembly tools.

It was designed completely hierarchically with a high focus on readability, and it's main purpose was for me to develop a better understanding of Computer Architecture. The processor is a single cycle design and it does not implement any sort of pipe-lining. The memory interface for both the instruction and data memory is a simple 32-bit interface with some control lines to select the read and write method (byte, half word, full word). The micro architecture was implemented based on the design within Patterson and Hennessy's Computer Architecture and Design with some modifications to support the full RV32I ISA.

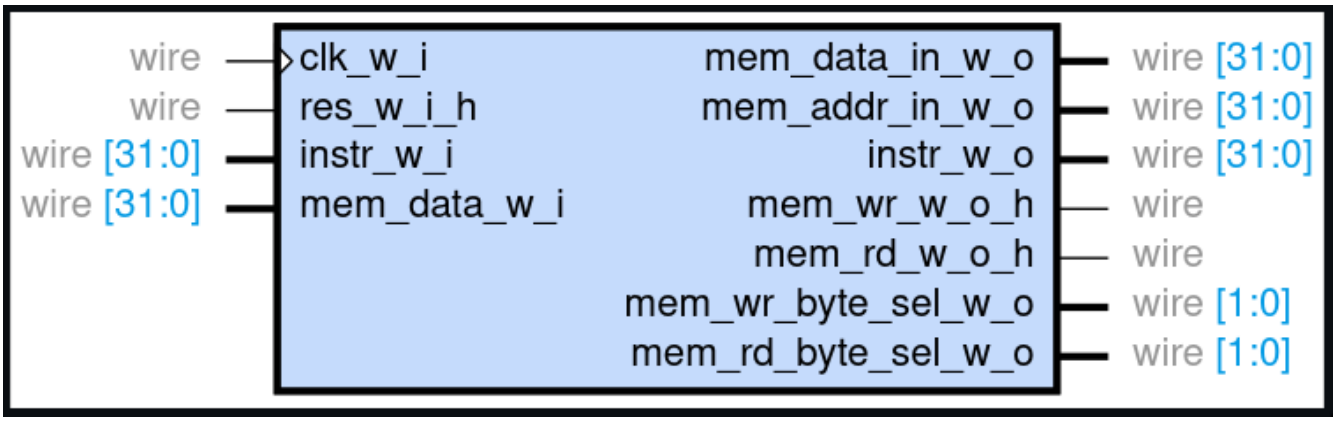
The RV32I ISA is fully specified within this document: RISC-V Specification

This site offers valuable examples of using the ISA alongside the instruction format for each instruction: Examples and Instruction Explanations

CPE CPU Micro Architecture



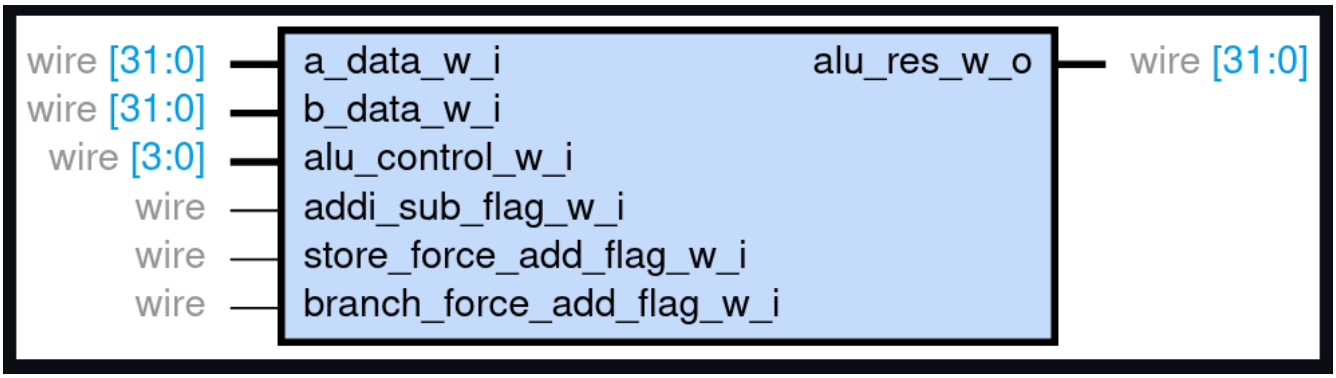
CPE CPU



cpe_cpu.v

The focus here was by no means to be efficient. This is a single cycle processor, it's far from efficient. The goal was simplicity. The interface wires are relatively straight forward. The reset is asynchronous high. The outputs are responsible for selecting the proper instruction from the memory, for determining whether a read or write should occur to the data memory, and determine what the method of the read or write should be as well (Byte, Half Word, Word). The only other signal that may need clarification is instr_w_o, which is the program counters output (IE: what memory address to read from the instruction memory).

Sub Module – ALU

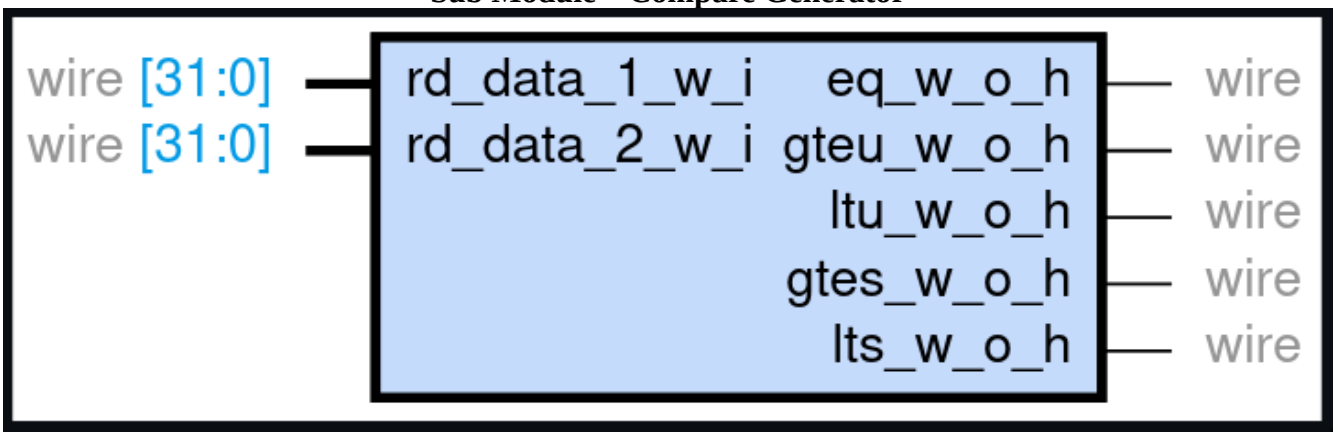


alu.v

The ALU is capable of performing all of the basic arithmetic operations on the systems. The A input can come from either register output 1, or from the program counter, and the B input can either come from register output 2 or the immediate generator.

addi_sub_flag_w_i forces an add even when were adding an immediate (ADDI allowed negative control bits on the standard ALU control bits). store_force_add_flag_w_i forces the ALU to add so that it can calculate the proper memory address. Again due to competing control lines. branch_force_add_flag_w_i forces the ALU to, you guessed it, add again. I need to calculate the memory and these all have different control signals that contradict one another based upon the different instruction types.

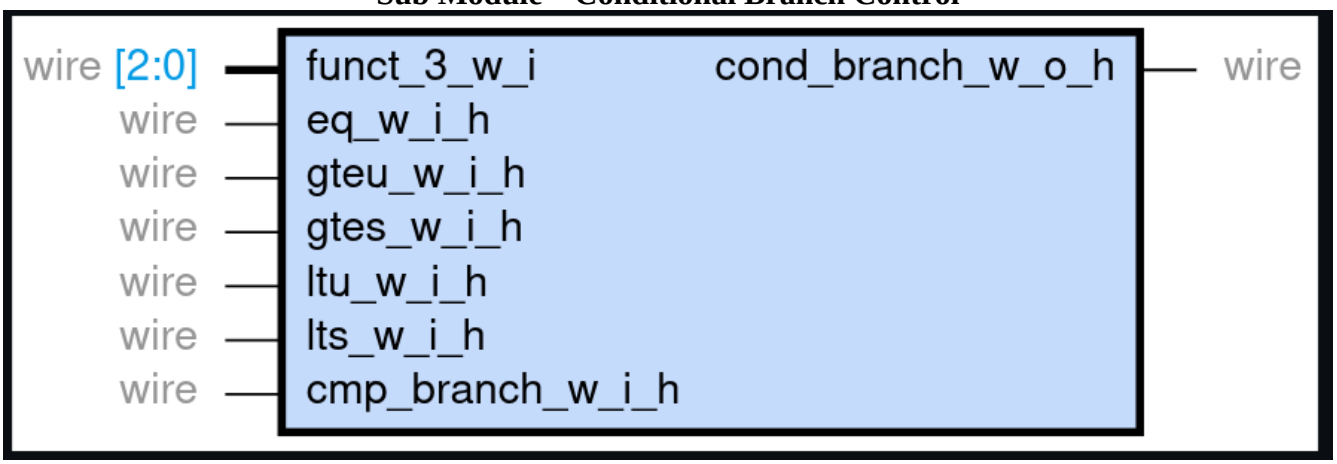
Sub Module – Compare Generator



cmp_gen.v

This module simply compares the outputs of register out 1 and register out 2 and gives you some comparison flags based upon them. These flags control branch instructions later.

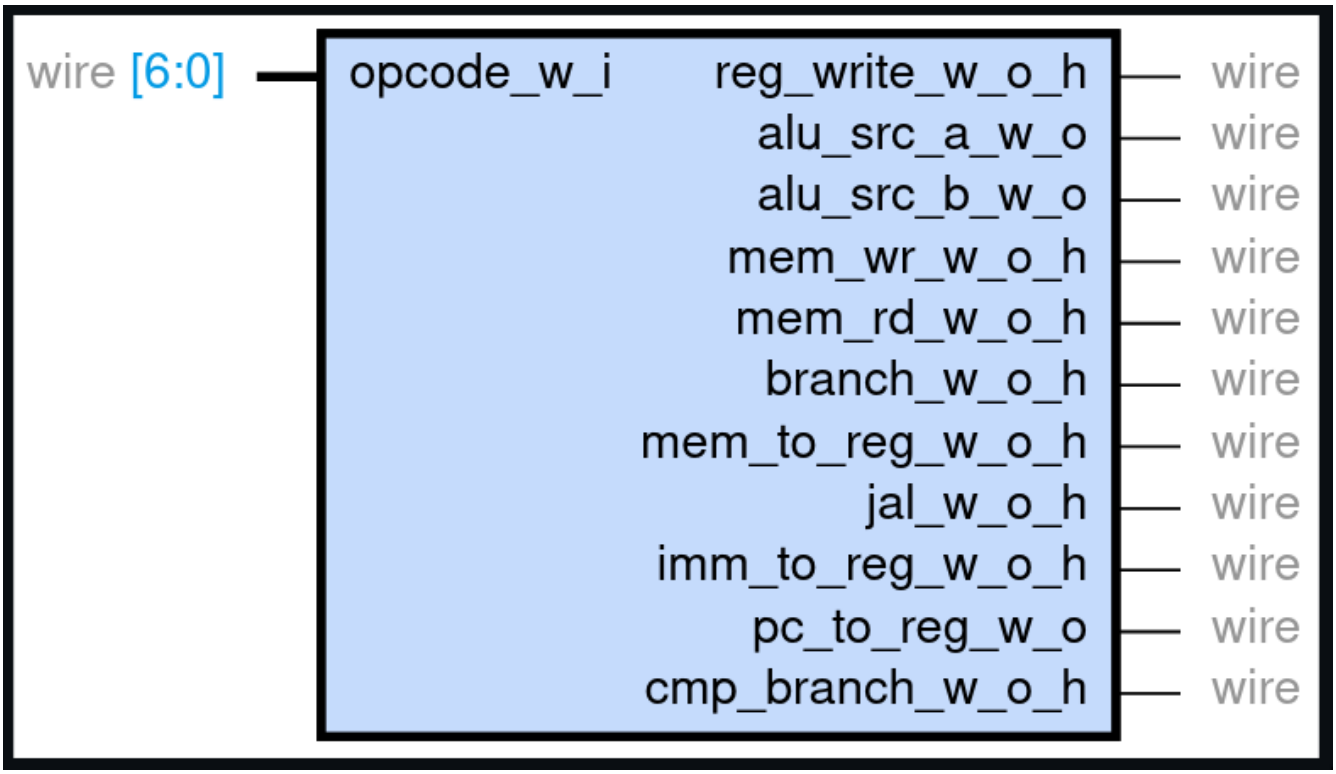
Sub Module – Conditional Branch Control



cond_branch_control.v

This module determines based upon the instruction in and the compare flags from cmp_gen whether or not a branch should be taken. This alongside the control registers that MUX of the PC input determine the branch result.

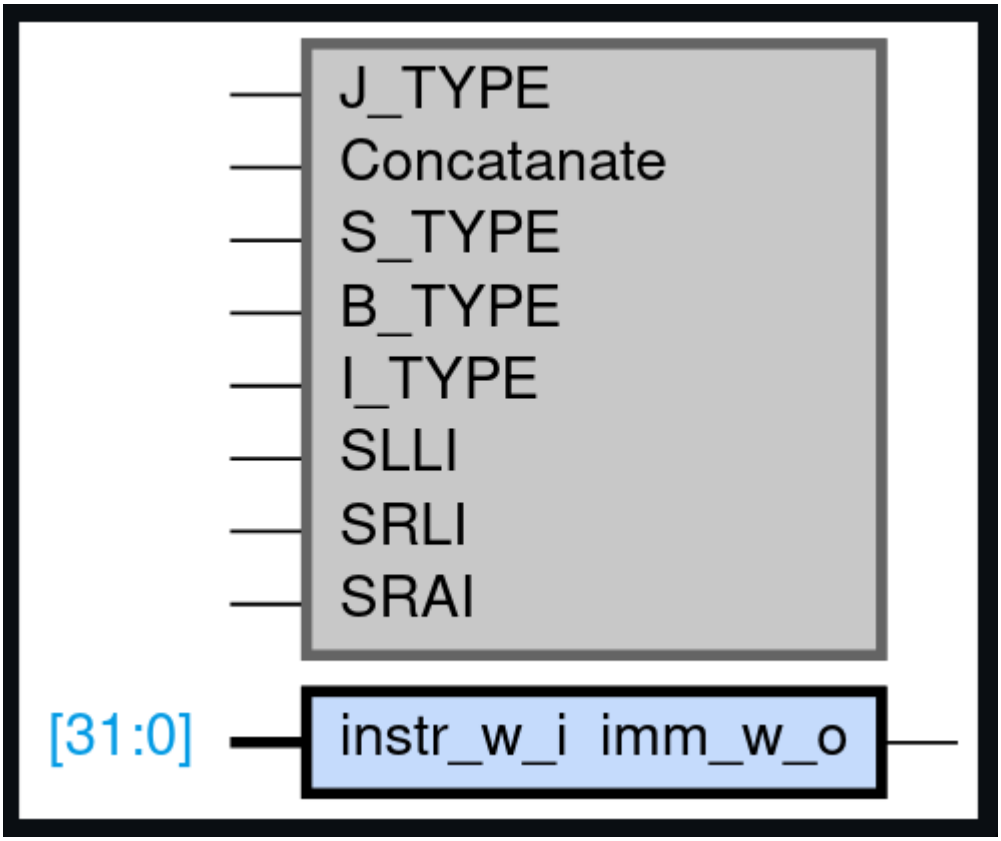
Sub Module – Control



control.v

This module takes in the op code and sets the relative control signals to the expected values. Alongside the conditional branch control this determines what path the signals take throughout the chip. It controls things like the mux lines, the memory control signals, and the memory word selects (for Byte, Half Word, and Word reads and writes). It also enables or disables the branch at the top level based on the opcode.

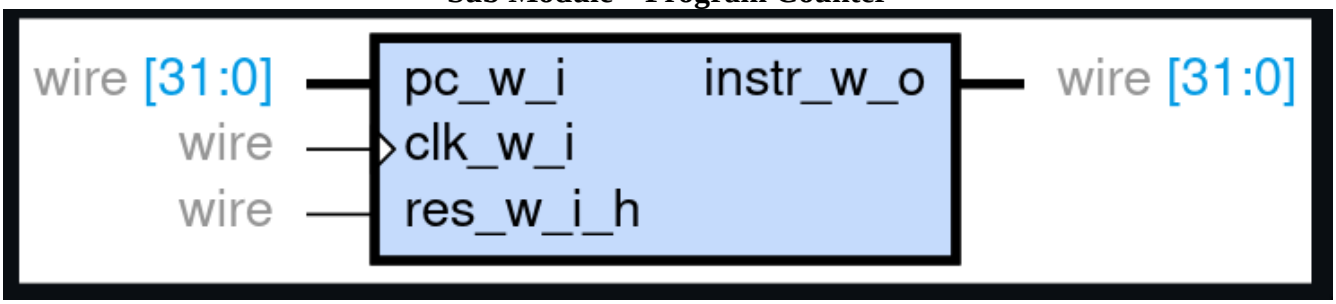
Sub Module – Immediate Generator



imm_gen.v

This module takes the instruction input and based upon the opp code and function values it will generate the appropriate immediate. Note the gray block is simply the parameter used as internal defines.

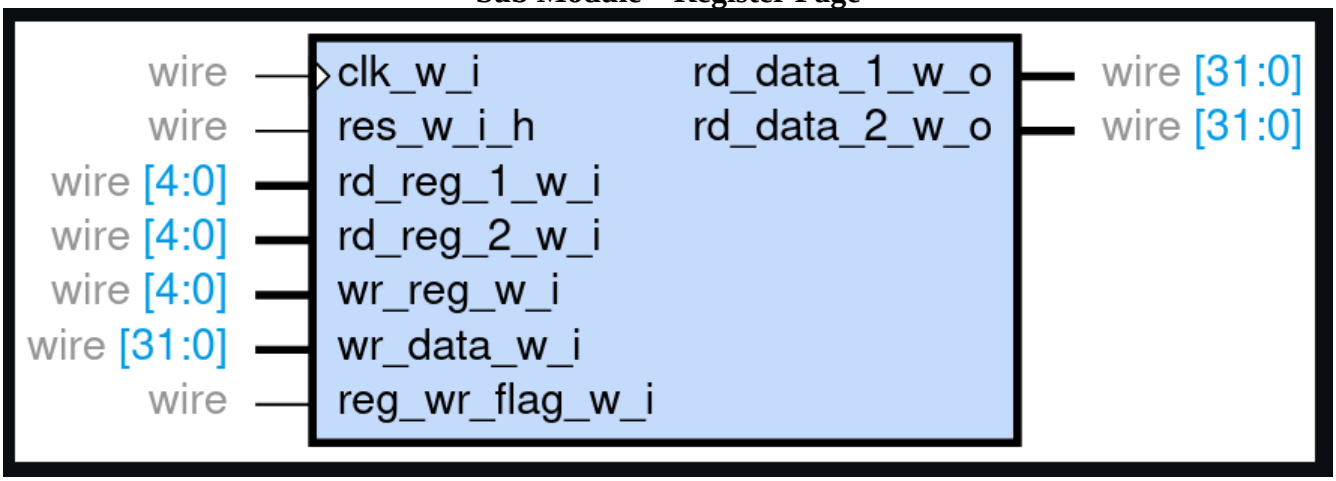
Sub Module – Program Counter



pc.v

The program counter simply counts based upon the selected input. This will just be PC + 4 if you're not branching, and if you are branching the result will be dependent upon the instruction.

Sub Module – Register Page



registers.v

The register page updates the registers based upon the write value and write address every clock cycle. It will output the selected outputs 1 and 2 as soon as the internal combinatorial logic allows as well. The register page contains 32, 32-bit registers. Register 0 will always be zero, even if written to.

Testing

Test development was done on Linux using Icarus Verilog alongside Make. GTKWave was used for the waveform viewer. Sub modules are self checking, but at the top level I model the memory and check results by hand. The instruction list can be seen within the ISA Breakdown spreadsheet within the notes folder.

Testing - Required Packages

```
sudo apt install iverilog gtkwave

# This tool draws the pretty block diagrams :)
pip3 install --upgrade symbolator
```

Testing - Running Tests

```
cd CPE_CPU/sim/rtl_sim/
# Target names listed below
make TARGET_NAME
# If you want to run the whole suit regressively and search for errors run this
make all|grep errors
# Or this to store to a file
make all >> test_res.dat
```

Testing - Test Targets

- all – runs all tests repressively
- cpe_cpu – runs the top level module and must be checked by hand
- alu
- cmp_gen
- control
- imm_gen
- pc
- registers
- cond_branch_control

It should be noted that all of the above targets can be appended with _wave to view the .vcd output of the test benches with the exception of the target all.

Testing - Sub Module Test Results

```
patrick@patrick-desktop ~/ws/school/advanced_digital_systems/CPE-CPU/sim/rtl_sim (master) $ make all
iverilog -o alu.tb alu_tb.v ../../rtl/verilog/alu.v
vvp alu.tb
VCD info: dumpfile alu.vcd opened for output.
              310002 ns: finished with           0 errors over       10000 trials

iverilog -o cmp_gen.tb cmp_gen_tb.v ../../rtl/verilog/cmp_gen.v
vvp cmp_gen.tb
VCD info: dumpfile cmp_gen.vcd opened for output.
               60002 ns: finished with           0 errors over       10000 trials

iverilog -o control.tb control_tb.v ../../rtl/verilog/control.v
vvp control.tb
VCD info: dumpfile control.vcd opened for output.
                  11 ns: finished with           0 errors

iverilog -o imm_gen.tb imm_gen_tb.v ../../rtl/verilog/imm_gen.v
vvp imm_gen.tb
VCD info: dumpfile imm_gen.vcd opened for output.
              160009 ns: finished with           0 errors over       10000 trials

iverilog -o pc.tb pc_tb.v ../../rtl/verilog/pc.v
vvp pc.tb
VCD info: dumpfile pc.vcd opened for output.
               20029 ns: finished with           0 errors over       10000 trials

iverilog -o registers.tb registers_tb.v ../../rtl/verilog/registers.v
vvp registers.tb
VCD info: dumpfile registers.vcd opened for output.
               60030 ns: finished with           0 errors over       10000 trials

iverilog -o cond_branch_control.tb cond_branch_control_tb.v ../../rtl/verilog/cond_branch_control.v
vvp cond_branch_control.tb
VCD info: dumpfile cond_branch_control.vcd opened for output.
                  32 ns: finished with           0 errors
```

All sub modules pass without error.

CPE CPU Test Vectors and Expected Results

I – Non Load Tests	Cycle	Expected	Pass/Fail	Result
ADDI 0x1, 0x0, 1	1	R1 = 1	P	1
ADDI 0x2, 0x1, 1	2	R2 = 2	P	2
ADDI 0x3, 0x2, 2	3	R3 = 4	P	4
ADDI 0x4, 0x3, -5	4	R4 = -1	P	-1
SLTI 0x5, 0x1, 2	5	R5 = 1	P	1
SLTI 0x5, 0x1, -1	6	R6 = 0	P	0
SLTIU 0x7, 0x1, 0	7	R7 = 0	P	0
SLTIU 0x8, 0x1, -1	8	R8 = 1	P	1
XORI 0x9, 0x1, 2	9	R9 = 3	P	3
XORI 0xA, 0x1, 1	10	R10 = 0	P	0
ORI 0xB, 0x1, 1	11	R11 = 1	P	1
OR 0xC, 0x1, 6	12	R12 = 7	P	7
SLLI 0xD, 0x1, 1	13	R13 = 2	P	2
SRLI 0xE, 0x1, 1	14	R14 = 0	P	0
SRLI 0xF, 0x2, 1	15	R15 = 1	P	1
SRAI 0x10, 0x1, 1	16	R16 = 0	P	0
SRAI 0x11, 0x2, 1	17	R17 = 1	P	1
SRAI 0x12, 0x4, 1	18	R18 = -1	P	-1
ANDI 0x13, 0x1, 1	19	R19 = 1	P	1
ANDI 0x14, 0x1, 0	20	R20 = 0	P	0
ANDI 0x15, 0x1, 0xFFF	21	R21 = 1	P	1
R Type Tests	Cycle	Expected	Pass Pass/Fail	Result
ADD 22, 1, 0	22	R22 = 1	P	1
ADD 23, 4, 0	23	R23 = -1	P	-1
SUB 24, 1, 0	24	R24 = 1	P	1
SUB 25, 0, 1	25	R25 = -1	P	-1
SUB 26, 3, 1	26	R26 = 3	P	3
SLL 27, 1, 1	27	R27 = 2	P	2
SLL 28, 1, 0	28	R28 = 1	P	1
SLL 29, 2, 3	29	R29 = 32	P	32
SLT 30, 1, 0	30	R30 = 0	P	0
SLT 31, 0, 1	31	R31 = 1	P	1
SLT 22, 4, 0	32	R22 = 1	P	1
SLT 23, 4, 2	33	R23 = 1	P	1
SLTU 24, 1, 0	34	R24 = 0	P	0
SLTU 25, 0, 1	35	R25 =1	P	1
SLTU 26, 4, 0	36	R26 = 0	P	0
XOR 27, 0, 3	37	R27 = 4	P	4
XOR 28, 1, 3	38	R28 = 5	P	5
SRL 29, 1, 0	39	R29 = 1	P	1
SRL 30, 2, 1	40	R30 = 1	P	1
SRL 31, 4, 1	41	R31 = 0x7FFF_FFFF	P	2147483647
SRA 22, 1, 0	42	R22 = 1	P	1
SRA 23, 2, 1	43	R23 = 1	P	1
SRA 24, 4, 1	44	R24 = -1	P	-1
OR 25, 1, 0	45	R25 = 1	P	1
OR 26, 2, 1	46	R26 = 3	P	3
OR 27, 1, 1	47	R27 = 1	P	1
AND 28, 1, 1	48	R28 = 1	P	1
AND 29, 1, 0	49	R29 = 0	P	0
AND 30, 1, 9	50	R30 = 1	P	1
			Pass	

S Type Tests	Cycle	Expected	Pass/Fail	Result			
SB 1, 0(rs0)	51	WR_SEL 00, DATA = 1, ADDR = 0	P	Cycle:	51	Wr_Sel:00 Data: 1 Addr:	0
SB , 1(rs0)	52	WR_SEL 00, DATA = 1, ADDR = 1	P	Cycle:	52	Wr_Sel:00 Data: 1 Addr:	1
SB 1, 2(rs0)	53	WR_SEL 00, DATA = 1, ADDR = 2	P	Cycle:	53	Wr_Sel:00 Data: 0 Addr:	2
SB 1, 3(rs0)	54	WR_SEL 00, DATA = 1, ADDR = 3	P	Cycle:	54	Wr_Sel:00 Data: 0 Addr:	3
SH 1, 0(rs0)	55	WR_SEL=01, DATA = 1, ADDR = 0	P	Cycle:	55	Wr_Sel:01 Data: 1 Addr:	0
SH 1,1(rs0)	56	WR_SEL=01, DATA = 1, ADDR = 2	P	Cycle:	56	Wr_Sel:01 Data: 1 Addr:	2
SH 1,0(rs0)	57	WR_SEL 10, DATA = 1, ADDR = 0	P	Cycle:	57	Wr_Sel:10 Data: 1 Addr:	0
			Pass				
I – Load Tests	Cycle	Expected	Pass/Fail	Result			
LB 26, 0(rs0)	58	WR_SEL 00, R26 = 1	P	Cycle:	58	Wr_Sel:00 Data: 1 Addr:	0
LB 27, 1(rs0)	59	WR_SEL 00, R27 = 1	P	Cycle:	59	Wr_Sel:00 Data: 1 Addr:	1
LB 28, 2(rs0)	60	WR_SEL 00, R28 = 1	P	Cycle:	60	Wr_Sel:00 Data: 1 Addr:	2
LB 29, 3(rs0)	61	WR_SEL 00, R29 = 1	P	Cycle:	61	Wr_Sel:00 Data: 1 Addr:	3
LH 30, 0(rx0)	62	WR_SEL 01, R30 = 1	P	Cycle:	62	Wr_Sel:01 Data: 1 Addr:	0
LH 31, 2(rx0)	63	WR_SEL 01, R31 = 1	P	Cycle:	63	Wr_Sel:01 Data: 1 Addr:	2
LW 22, 0(rx0)	64	WR_SEL 10, R30 = 1	P	Cycle:	64	Wr_Sel:10 Data: 1 Addr:	0
			Pass				
B Type Tests	Cycle	Expected	Pass/Fail	Result			
BEQ 0, 0, 8	65	PC + 8	P	8			
BEQ 1, 0, 8	66	PC + 4	P	4			
BNE 0, 0, 8	67	PC + 4	P	4			
BNE 0, 1, 8	68	PC + 8	P	8			
BLT 0, 0, 8	69	PC + 4	P	4			
BLT 0, 1, 8	70	PC + 8	P	8			
BLT 4, 0, 8	71	PC + 8	P	8			
BGE 0, 0, 8	72	PC + 8	P	8			
BGE 1, 0, 8	73	PC + 8	P	8			
BGE 4, 0, 8	74	PC + 4	P	4			
BLTU 0, 0, 8	75	PC + 4	P	4			
BLTU 0, 1, 8	76	PC + 8	P	8			
BLTU 4, 0, 8	77	PC + 4	P	4			
BGTEU 0, 0, 8	78	PC + 8	P	8			
BGTEU 1, 0, 8	79	PC + 8	P	8			
BGTEU 4, 0, 8	80	PC + 8	P	8			
BGTEU 0, 1, 8	81	PC + 4	P	4			
			Pass				
U Type Tests	Cycle	Expected	Pass/Fail	Result			
LUI 22, (1 << 19)	82	R22 = (1<<19)	P	4294963200			
AUIPC 23, (1 << 19)	83	PC + (1 << 19)	P	4294963568			
			Pass				
J Type Tests	Cycle	Expected	Pass/Fail	Result			
JALR 24, 3, 376	84	R24 = PC + 4 PC = x[r0] + OFFSET	P	376 NEW PC, R24 = 376			
JAL 25, 8	85	R25 = PC +4, PC = PC + 8	F – Only R25	R25 = 380, NEW PC = 384			
			Pass				

Testing - CPE CPU Results

```
iverilog -o cpe_cpu.tb cpe_cpu_tb.v ../../rtl/verilog/cpe_cpu.v ../../rtl/
log/cond_branch_control.v ../../rtl/verilog/cmp_gen.v
vvp cpe_cpu.tb
VCD info: dumpfile cpe_cpu.vcd opened for output.
Setting reset_done
                640 ns: Cycle:          1 Addr: 1 Res:          1
                645 ns: Cycle:          2 Addr: 2 Res:          2
                655 ns: Cycle:          3 Addr: 3 Res:          4
                665 ns: Cycle:          4 Addr: 4 Res:         -1
                675 ns: Cycle:          5 Addr: 5 Res:          1
                685 ns: Cycle:          6 Addr: 6 Res:          0
                695 ns: Cycle:          7 Addr: 7 Res:          0
                705 ns: Cycle:          8 Addr: 8 Res:          1
                715 ns: Cycle:          9 Addr: 9 Res:          3
                725 ns: Cycle:         10 Addr:10 Res:          0
                735 ns: Cycle:         11 Addr:11 Res:          1
                745 ns: Cycle:         12 Addr:12 Res:          7
                755 ns: Cycle:         13 Addr:13 Res:          2
                765 ns: Cycle:         14 Addr:14 Res:          0
                775 ns: Cycle:         15 Addr:15 Res:          1
                785 ns: Cycle:         16 Addr:16 Res:          0
                795 ns: Cycle:         17 Addr:17 Res:          1
                805 ns: Cycle:         18 Addr:18 Res:         -1
                815 ns: Cycle:         19 Addr:19 Res:          1
                825 ns: Cycle:         20 Addr:20 Res:          0
                835 ns: Cycle:         21 Addr:21 Res:          1
                835 ns: I Type Complete

                845 ns: Cycle:         22 Addr:22 Res:          1
                855 ns: Cycle:         23 Addr:23 Res:         -1
                865 ns: Cycle:         24 Addr:24 Res:          1
                875 ns: Cycle:         25 Addr:25 Res:         -1
                885 ns: Cycle:         26 Addr:26 Res:          3
                895 ns: Cycle:         27 Addr:27 Res:          2
                905 ns: Cycle:         28 Addr:28 Res:          1
                915 ns: Cycle:         29 Addr:29 Res:         32
                925 ns: Cycle:         30 Addr:30 Res:          0
                935 ns: Cycle:         31 Addr:31 Res:          1
                945 ns: Cycle:         32 Addr:22 Res:          1
                955 ns: Cycle:         33 Addr:23 Res:          1
                965 ns: Cycle:         34 Addr:24 Res:          0
                975 ns: Cycle:         35 Addr:25 Res:          1
                985 ns: Cycle:         36 Addr:26 Res:          0
                995 ns: Cycle:         37 Addr:27 Res:          4
               1005 ns: Cycle:         38 Addr:28 Res:          5
               1015 ns: Cycle:         39 Addr:29 Res:          1
               1025 ns: Cycle:         40 Addr:30 Res:          1
               1035 ns: Cycle:         41 Addr:31 Res: 2147483647
```

```
1035 ns: Cycle:         41 Addr:31 Res: 2147483647
1045 ns: Cycle:         42 Addr:22 Res:          1
1055 ns: Cycle:         43 Addr:23 Res:          1
1065 ns: Cycle:         44 Addr:24 Res:         -1
1075 ns: Cycle:         45 Addr:25 Res:          1
1085 ns: Cycle:         46 Addr:26 Res:          3
1095 ns: Cycle:         47 Addr:27 Res:          1
1105 ns: Cycle:         48 Addr:28 Res:          1
1115 ns: Cycle:         49 Addr:29 Res:          0
1125 ns: Cycle:         50 Addr:30 Res:          1
1125 ns: R Type Complete

1135 ns: Cycle:         51 Wr_Sel:00 Data:         1 Addr:         0
1145 ns: Cycle:         52 Wr_Sel:00 Data:         1 Addr:         1
1155 ns: Cycle:         53 Wr_Sel:00 Data:         1 Addr:         2
1165 ns: Cycle:         54 Wr_Sel:00 Data:         1 Addr:         3
1175 ns: Cycle:         55 Wr_Sel:01 Data:         1 Addr:         0
1185 ns: Cycle:         56 Wr_Sel:01 Data:         1 Addr:         2
1195 ns: Cycle:         57 Wr_Sel:10 Data:         1 Addr:         0
1195 ns: S Type Complete

1205 ns: Cycle:         58 Wr_Sel:00 Data:         1 Addr:         0
1215 ns: Cycle:         59 Wr_Sel:00 Data:         1 Addr:         1
1225 ns: Cycle:         60 Wr_Sel:00 Data:         1 Addr:         2
1235 ns: Cycle:         61 Wr_Sel:00 Data:         1 Addr:         3
1245 ns: Cycle:         62 Wr_Sel:01 Data:         1 Addr:         0
1255 ns: Cycle:         63 Wr_Sel:01 Data:         1 Addr:         2
1265 ns: Cycle:         64 Wr_Sel:10 Data:         1 Addr:         0
1265 ns: S Type Complete

1275 ns: Cycle:         65 New PC:       264 Old PC:       256 Difference:          8
1285 ns: Cycle:         66 New PC:       268 Old PC:       264 Difference:          4
1295 ns: Cycle:         67 New PC:       272 Old PC:       268 Difference:          4
1305 ns: Cycle:         68 New PC:       280 Old PC:       272 Difference:          8
1315 ns: Cycle:         69 New PC:       284 Old PC:       280 Difference:          4
1325 ns: Cycle:         70 New PC:       292 Old PC:       284 Difference:          8
1335 ns: Cycle:         71 New PC:       300 Old PC:       292 Difference:          8
1345 ns: Cycle:         72 New PC:       308 Old PC:       300 Difference:          8
1355 ns: Cycle:         73 New PC:       316 Old PC:       308 Difference:          8
1365 ns: Cycle:         74 New PC:       320 Old PC:       316 Difference:          4
1375 ns: Cycle:         75 New PC:       324 Old PC:       320 Difference:          4
1385 ns: Cycle:         76 New PC:       332 Old PC:       324 Difference:          8
1395 ns: Cycle:         77 New PC:       336 Old PC:       332 Difference:          4
1405 ns: Cycle:         78 New PC:       344 Old PC:       336 Difference:          8
1415 ns: Cycle:         79 New PC:       352 Old PC:       344 Difference:          8
1425 ns: Cycle:         80 New PC:       360 Old PC:       352 Difference:          8
1435 ns: Cycle:         81 New PC:       364 Old PC:       360 Difference:          4
1435 ns: B Type Complete

1445 ns: Cycle:         82 Res:4294963200 Old PC:       364 Input + Old PC2147479916
1455 ns: Cycle:         83 Res:4294963568 Old PC:       368 Input + Old PC2147480288
1455 ns: U Type Complete

1465 ns: Cycle:         84 Res:       376 Old PC:       372 New PC:       376
1475 ns: Cycle:         85 Res:       380 Old PC:       376 New PC:       384
1475 ns: JAL/JALR Type Complete

2130 ns: finished ERRORS must be counted by hand
```

Testing - Conclusions

As you can see, all of the test results were combined into the test vector data sheet and compared against the expected results. After a lot of bug fixing, all tests are passing and the ISA is fully functional. The only exceptions to this are the ECALL, EBREAK, and FENCE instructions. These instructions are used to organize memory accesses between different processor cores, and for the GNU debugger to take execution of the processor over live. Neither of these features was relevant to me within this design. This design can be compiled for with GCC assuming the memory architecture you design has an appropriate linker script.