Katherine Perry
Cpeg 324
Lab 4 report

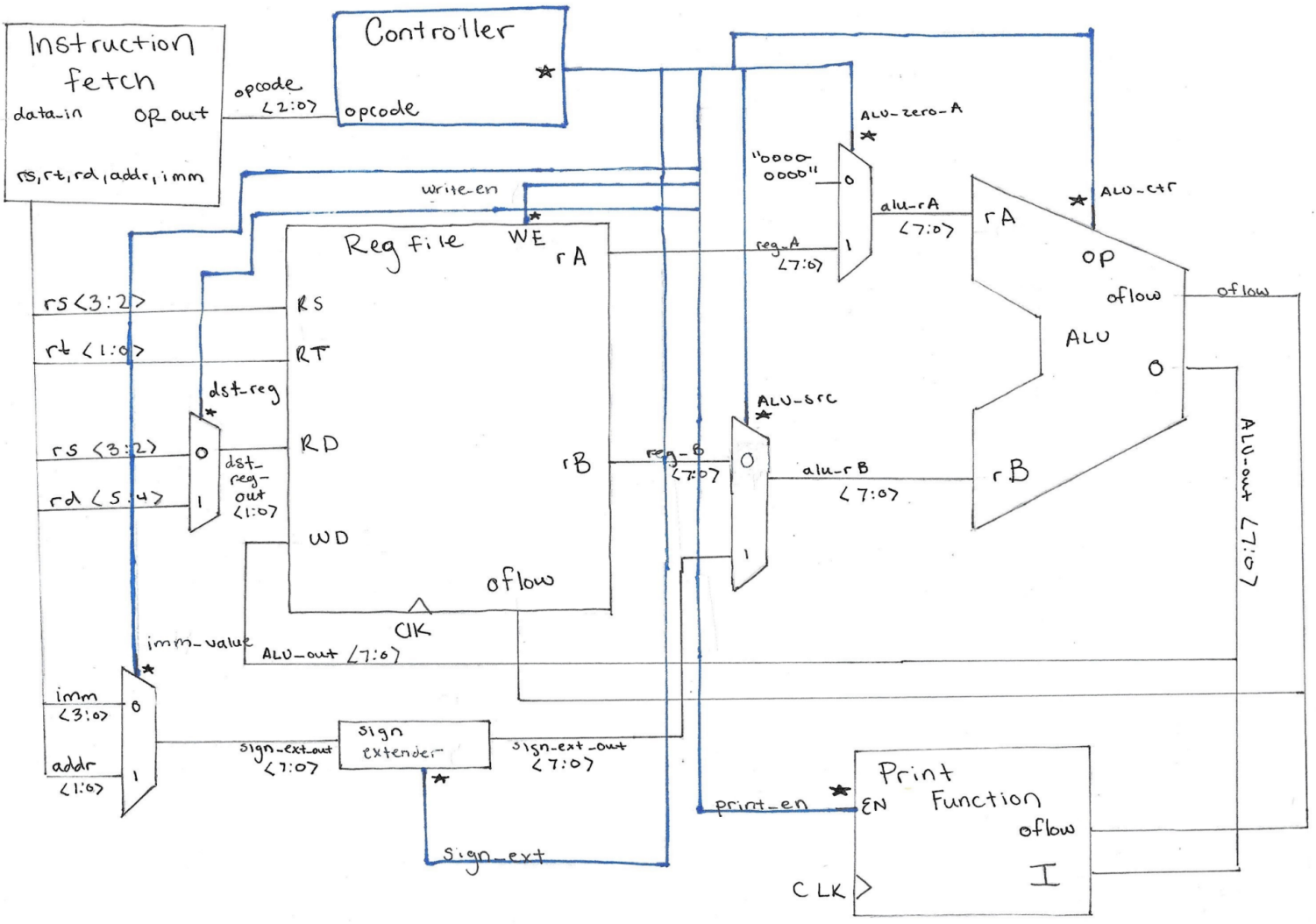
A Single-Cycle Calculator

Circuit Design

The design for the 8-bit calculator circuit consists of 5 components, 8 control signals and a clock signal. The five components are the instruction fetch unit, the controller unit, the register file, the ALU and the print function. This single-cycle system implements five different instructions: load address, load immediate, ADD, SUB and display. Below is the RTL of the calculator modeled in VHDL, displaying the connections between components and control signals. The design intention and datapath for each instruction will be further explained in the following sections.

- Instruction Fetch
- Controller
- RegFile
- ALU
- Print Function

- Instructions:
  - Load Address
  - Load Immediate
  - ADD
  - SUB
  - Display



Instruction Fetch: inst_fet.vhdl

The instruction fetch component reads the 8-bit ISA instruction (data_in) and designates each bit to one of the follow:

- **Op_out:**

  A 3-bit output port that is sent to the controller unit and determines the control signals and switches for the rest of the circuit. In order to keep the ISA to 8-bits the op codes have been compressed and the instruction fetch must decode either a 2-bit or 3-bit portion of the ISA instruction to decide which command is being called. This is done by looking at the first two MSBs. If they are "00", then the instruction fetch uses the next MSB to determine the command. If the first two MSBs are not "00", then the MSB of op_out is '0' and op_out(1 downto 0) is equal to the two MSBs of the 8-bit instruction input. Below is a table that shows the 8-bit instruction to op_out translation.

Command:	ISA 8-bit representation:	Op_out value:
Load Address	"000xxxxx"	"000"
Load Immediate	"01xxxxxx"	"001"
ADD	"10xxxxxx"	"010"
SUB	"11xxxxxx"	"011"
DISPLAY	"001xxxxx"	"100"

See appendix A for VHDL code corresponding to this procedure

- **RS:**

  A 2-bit output that denotes the bits of the first register to read from. This output is forwarded to the register file for read functions. Instruction fetch sets RS to the third and second bit (3 downto 2) of the 8-bit ISA instruction input.

- **RT:**

  A 2-bit output that denotes the bits of the second register to read from. This output is forwarded to the register file for read functions. Instruction fetch sets RT to the first and zeroth bit (1 downto 0) of the 8-bit ISA instruction input.

- **RD:**

  A 2-bit output that denotes the bits of the register to write to. This output is forwarded to the register file for write functions. Instruction fetch sets RD to the fifth and fourth bit (5 downto 4) of the 8-bit ISA instruction input.

- **Addr:**

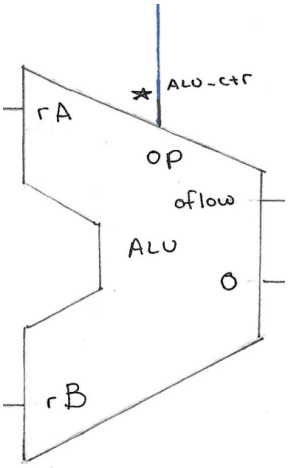
  A 2-bit output that holds a label. This label will be used to create an address that eventually is loaded into a register after passing through the ALU. This output port is forwarded to the IMM_VALUE MUX.

- **Imm:**

  A 4-bit output that holds an immediate value. This value will be loaded into a register after passing through the ALU. This output port is forwarded to the IMM_VALUE MUX.

Controller : control.vhdl

This component inputs the decoded opcodes from the instruction fetch and loads values to the 8 control signals based on mapping shown below. These control signal values are then directed to different MUXs and components throughout the design.



Alu_ctr

A 2-bit signal that controls the operation of the ALU. There are three operations: ADD (“00”), SUB(“01”), and print (“10”).

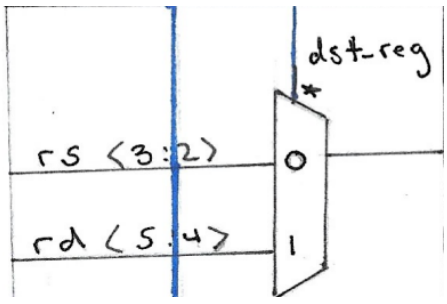
Alu_ctr(1) <= opcode(2)
Alu_ ctr(0) <= opcode(1) and opcode(0)

Mapping table:

Command:	opcode(2)	opcode(1)	opcode(0)	Alu_ctr(1)	Alu_ctr(0)
Load Addr	0	0	0	0	0
Load Imm	0	0	1	0	0
ADD	0	1	0	0	0
SUB	0	1	1	0	1
Display	1	0	0	1	0

Dst_reg

A 1-bit signal that decides whether the write register on regfile is either the designated RD or RS bits from the instruction fetch component. This signal controls a MUX that connects to the write register input port on the regfile component.



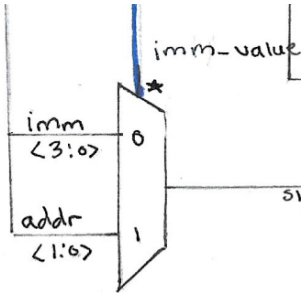
‘0’ => RS
‘1’ => RD
Dst_reg <= opcode(1) or opcode(0)

Mapping table:

Command:	opcode(2)	opcode(1)	opcode(0)	Dst_reg
Load Addr	0	0	0	Rs (0)
Load Imm	0	0	1	Rd (1)
ADD	0	1	0	Rd (1)
SUB	0	1	1	Rd (1)
Display	1	0	0	X

Imm_value

A 1-bit signal that decides which value addr or imm from instruction fetch is forwarded to the next port. This signal controls a MUX that connects to the sign_extender.



'0' => imm
'1' => addr
Imm_value <= not opcode(0)

Mapping table:

Command:	opcode(2)	opcode(1)	opcode(0)	Imm_value
Load Addr	0	0	0	1
Load Imm	0	0	1	0
ADD	0	1	0	X
SUB	0	1	1	X
Display	1	0	0	X

Write_en

A 1-bit signal that enables the write function of regfile. This signal is forwarded to the input port WE on regfile component.

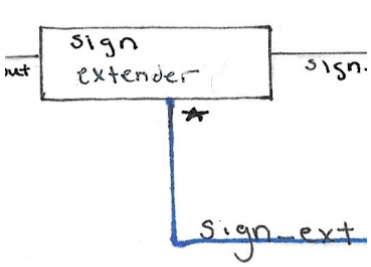
'0' => write disable
'1' => write enabled
Write_en <= not opcode(2)

Mapping table:

Command:	opcode(2)	opcode(1)	opcode(0)	Write_en
Load Addr	0	0	0	1
Load Imm	0	0	1	1
ADD	0	1	0	1
SUB	0	1	1	1
Display	1	0	0	0

Sign_ext

A 1-bit signal that controls a MUX which decides how to extend a 4-bit signal into an 8-bit signal. It will either sign extend it or zero extend the signal. This signal is then forwarded to the ALU_SRC MUX.



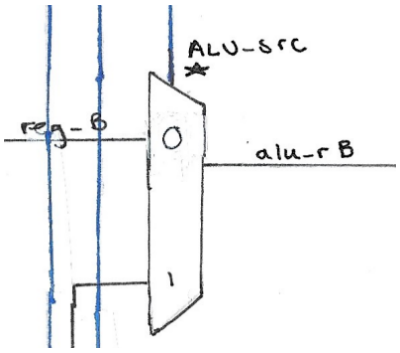
'0' => zero extend
'1' => sign extend
Sign_ext <= opcode(0)

Mapping table:

Command:	opcode(2)	opcode(1)	opcode(0)	Sign_ext
Load Addr	0	0	0	0
Load Imm	0	0	1	1
ADD	0	1	0	X
SUB	0	1	1	X
Display	1	0	0	X

Alu_src

A 1-bit signal that determines the signal that goes to the second input of the ALU either the output of sign_ext or the second read output of regfile. This signal controls the ALU_SRC MUX and connects to the second input of the ALU.



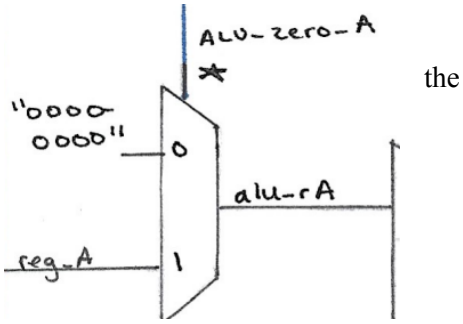
'0' => regfile (reg_B)
'1' => sign_ext
Alu_src <= not opcode(1)

Mapping table:

Command:	opcode(2)	opcode(1)	opcode(0)	Alu_src
Load Addr	0	0	0	1
Load Imm	0	0	1	1
ADD	0	1	0	0
SUB	0	1	1	0
Display	1	0	0	X

Alu_zero_A

A 1-bit signal that decides the first input of the ALU. This signal controls a MUX with two options of "00000000" or the first output of regfile. This MUX forwards its value to the first input of the ALU.



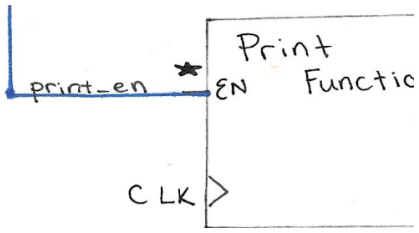
'0' => "00000000"
'1' => regfile (reg_ A)
Alu_zero_A <= (not opcode(0)) or opcode(1)

Mapping table:

Command:	opcode(2)	opcode(1)	opcode(0)	Alu_zero_A
Load Addr	0	0	0	1
Load Imm	0	0	1	0
ADD	0	1	0	1
SUB	0	1	1	1
Display	1	0	0	1

Print_en

A 1-bit signal that enables the print function to print.



'0' => printing disabled
'1' => printing enabled
Print_en <= opcode(2)

Mapping table:

Command:	opcode(2)	opcode(1)	opcode(0)	Print_en
Load Addr	0	0	0	0
Load Imm	0	0	1	0
ADD	0	1	0	0
SUB	0	1	1	0
Display	1	0	0	1

A table of all the control signals and their values for each instruction is shown in appendix B.

Register File : regfile.vhdl

This component inputs ports from the instruction fetch (RS,RT, and RD), controller (write_en), and ALU (ALU_out and oflow). The regfile is connected to four 8-bit registers with addresses “00”, “01”, “10”, and “11”. The writing function is determined by the clock, write_en and oflow. When the clock is at rising edge, enable = ‘1’ and oflow = ‘0’, the regfile will write ALU_out to one of its four registers as denoted by RD. If overflow (oflow) is detected by the ALU, no writing will occur even if the write enable is ‘1’. The value at RD will remain the same. The read function is combinational and will always output two register values stored by RS and RT. These two registers values are forwarded to signals that eventually connect to the ALU.

ALU : alu.vhdl

This component inputs ports from the controller(alu_ctr) and ALU_ZERO_A MUX and ALU_SRC MUX. The outputs, ALU_out and oflow, of this component go to the print function and register file. The ALU has three operations: ADD, SUB, and Print which are determined by alu_ctr. Before doing any operations , the ALU sign extends the two inputs to be able to detect overflow, these two inputs are ext_A and ext_B. To limit the need of conversion all operations are done using two's complements. This means that regardless of whether or not the value is positive or negative, the method to add or subtract will not change. This also makes determining overflow easier, because overflow can only occur when adding the two negative or two positive numbers.

ADD

The add operation uses VHDL operator ‘+’ to add the two inputs from ext_A and ext_B. To determine overflow, first the ALU will compare the MSBs of the two inputs. If they are the same, then overflow will need to be calculated; if not, oflow = ’0’. After adding the two inputs, the result will be stored in a 9-bit signal. If both inputs are positive or negative, overflow will occur if the sign of the result differs from the inputs.

SUB

The sub operation uses VHDL operator '+' to subtract the two inputs ext_A and ext_B. This is because A - B = A + (-B). By converting ext_B to its two's complement (not(ext_B) +1), the operation becomes A + (-B). By using this method, no sign conversion needs to be done.

To determine overflow, first the ALU will compare the MSBs of the two inputs. If they are opposite then overflow will need to be calculated; if not oflow = '0'. This is because after two's complement conversion, inputs of opposite signs become the same sign. Overflow only occurs when adding two values of the same sign. After subtracting the two inputs, the result will be stored in a 9-bit signal. If both inputs are positive or negative after two's complement, overflow will occur if the sign of the result differs from the inputs.

Overflow explained:

ALU_out is an 8-bit output. So the 9th bit of result (result(8)) is used to determine overflow when adding two positive numbers in cases when result(7) = '0'. The first bit of any signal is used to determine the sign. If that bit has changed, then the sign of the value has changed.

Sign	Overflow occurs when	Explanation
Both positive	If result(8) or result(7) = '1'	If result(7) = 1 and result(8) = 0 the sign has changed If result(7) = 0 and result(8) = 1 the sign has changed If both result(7) and result(8) = 1 the sign has changed
Both negative	Not result(7)	If result(7) = 0 the sign has change regardless of what result(8) is, overflow has occurred

Print

This operation will occur when alu_ctr = "10" or "11" by default. This will assign the output (ALU_out) to the first input (ext_A(7 downto 0)) and assign oflow = '0' since no change is to occur to the data values in this operation. This will then be forwarded to the print function.

Print Function : print.vhdl

This component inputs the outputs of ALU (ALU_out and oflow), controller (Print_en) and clock and will print to monitor under specific conditions. This component uses Textio package's features such as write() aad writeline() to display. The ALU_out is converted to its signed decimal value, and then displayed with textio package if a rising edge has occurred and print_en = '1'. This component will also print when a rising edge has occurred and oflow = '1'. However, it will print an overflow error, alerting the user that the write data was not stored in RD.

Instructions and Datapath

Load Address:

ISA instruction explained

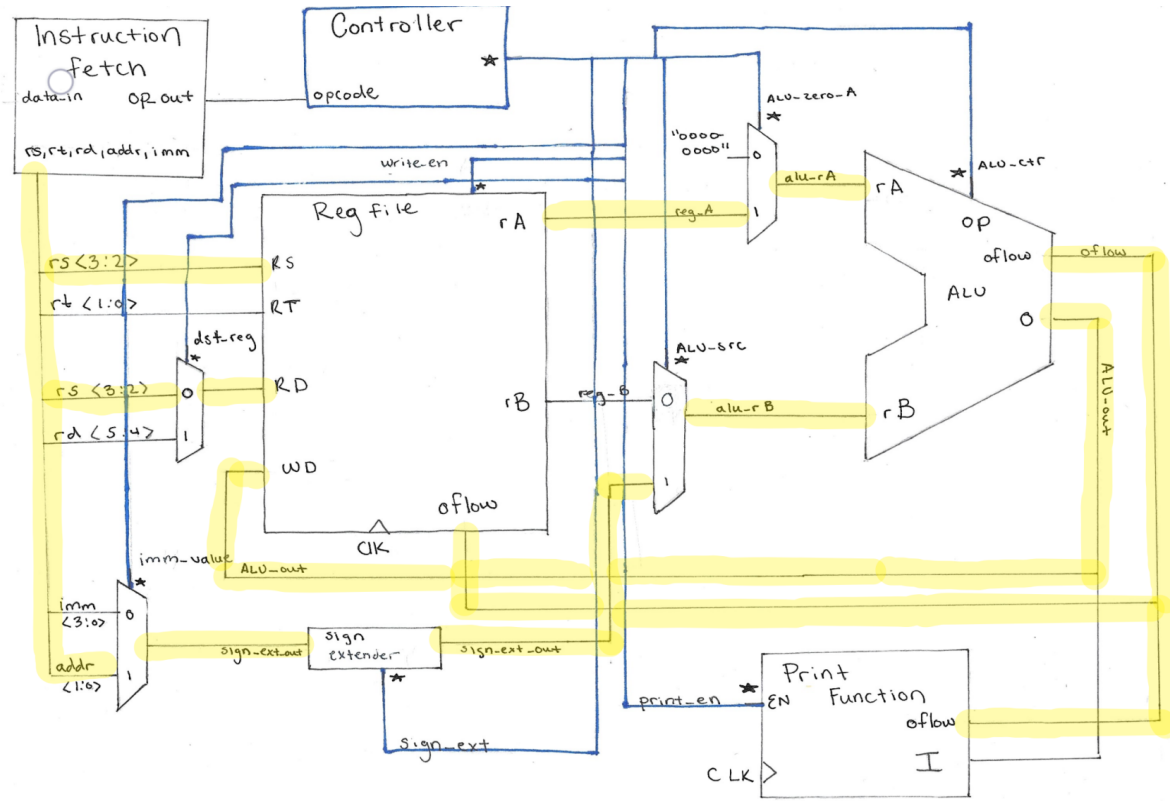
Bits	7	6	5	4	3	2	1	0
	0	0	0	X	R	S	i	i

DataPath:

The instruction fetch values that will matter will be RS, RD, and addr. RS will be used to compute the address, RD will be the register that the data is written to and addr is the data input from the 8-bit instruction. Dst_reg will select RS as RD. Imm_value will select addr as value. Sign extender will use zero extend, since the label isn't signed. Alu_src will select sign_ext_out as input for rB for the ALU. RS will determine the rA value, and alu_zero_A will select reg_A as the input for rA in the ALU. ALU will be set to ADD, and add RS and addr to create the label of the address. This value will be ALU_out and written to RD. If overflow were to occur, the print function will alert the user.

Pseudocode for address : RD < = Address = [RS + LABEL]

Dst_reg	Imm_value	Sign_ext	write_en	Alu_src	alu_zero_A	Alu_ctr	print_en
0	1	0	1	1	1	ADD	0



Load Immediate

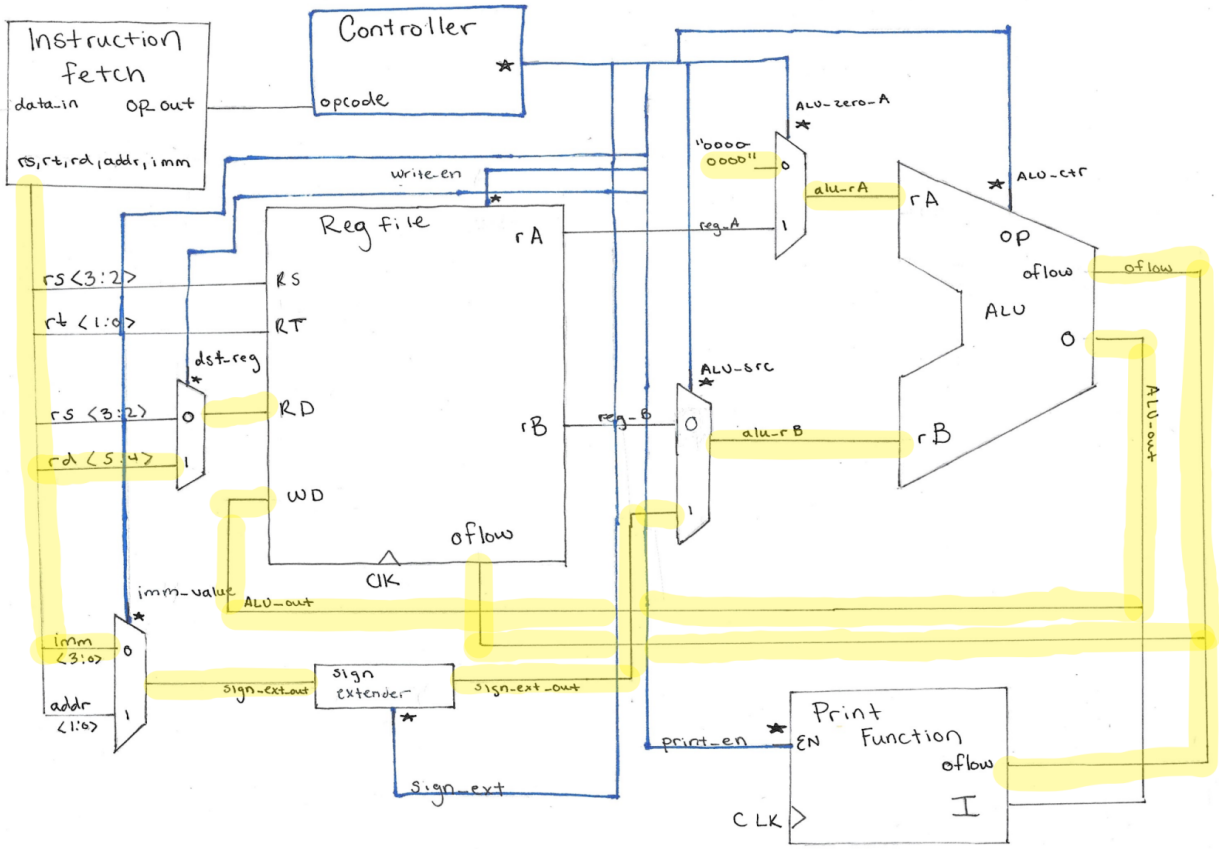
ISA instruction explained

Bits	7	6	5	4	3	2	1	0
	0	1	R	D	i	i	i	i

DataPath:

The instruction fetch values that will matter will be RD and imm. RD will be the register that the data is written to and imm is the data input from the 8-bit instruction. Dst_reg will select RD as RD. Imm_value will select imm as value. Sign extender will use sign-extend and alu_src will select sign_ext_out as input for rB for the ALU. Alu_zero_A will select “00000000” as the input for rA in the ALU. This will prevent the data in rB from changing. ALU will be set to ADD, and add rA (zeros) and imm. ALU_out which is equal to imm will be written to RD.

Dst_reg	Imm_value	Sign_ext	write_en	Alu_src	alu_zero_A	Alu_ctr	print_en
1	0	1	1	1	0	ADD	0



ADD

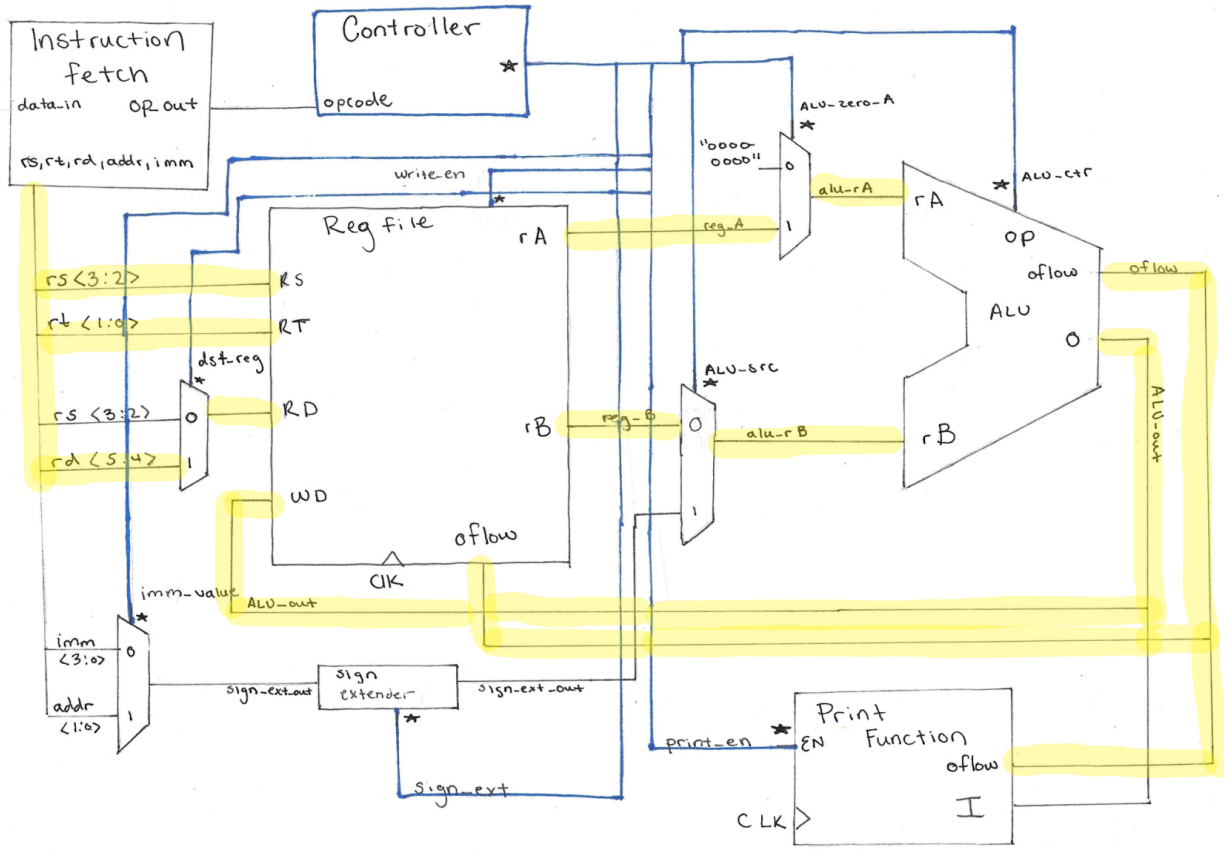
ISA instruction explained

Bits	7	6	5	4	3	2	1	0
	1	0	R	D	R	S	R	T

DataPath:

The instruction fetch values that will matter will be RS, RT, and RD. RS and RT will be read from in regfile as values to be added in the ALU and RD will be the register that the data is written to. Dst_reg will select RD as RD. Alu_src will select reg_B as input for rB for the ALU. Since reg_B is selected then imm_value and sign_ext doesn't matter. RS will determine the rA value, and alu_zero_A will select reg_A as the input for rA in the ALU. ALU will be set to ADD, and add RS and RT. This value will be ALU_out and written to RD. If overflow were to occur, the print function will alert the user and RD will not change.

Dst_reg	Imm_value	Sign_ext	write_en	Alu_src	alu_zero_A	Alu_ctr	print_en
1	X	X	1	0	1	ADD	0



SUB

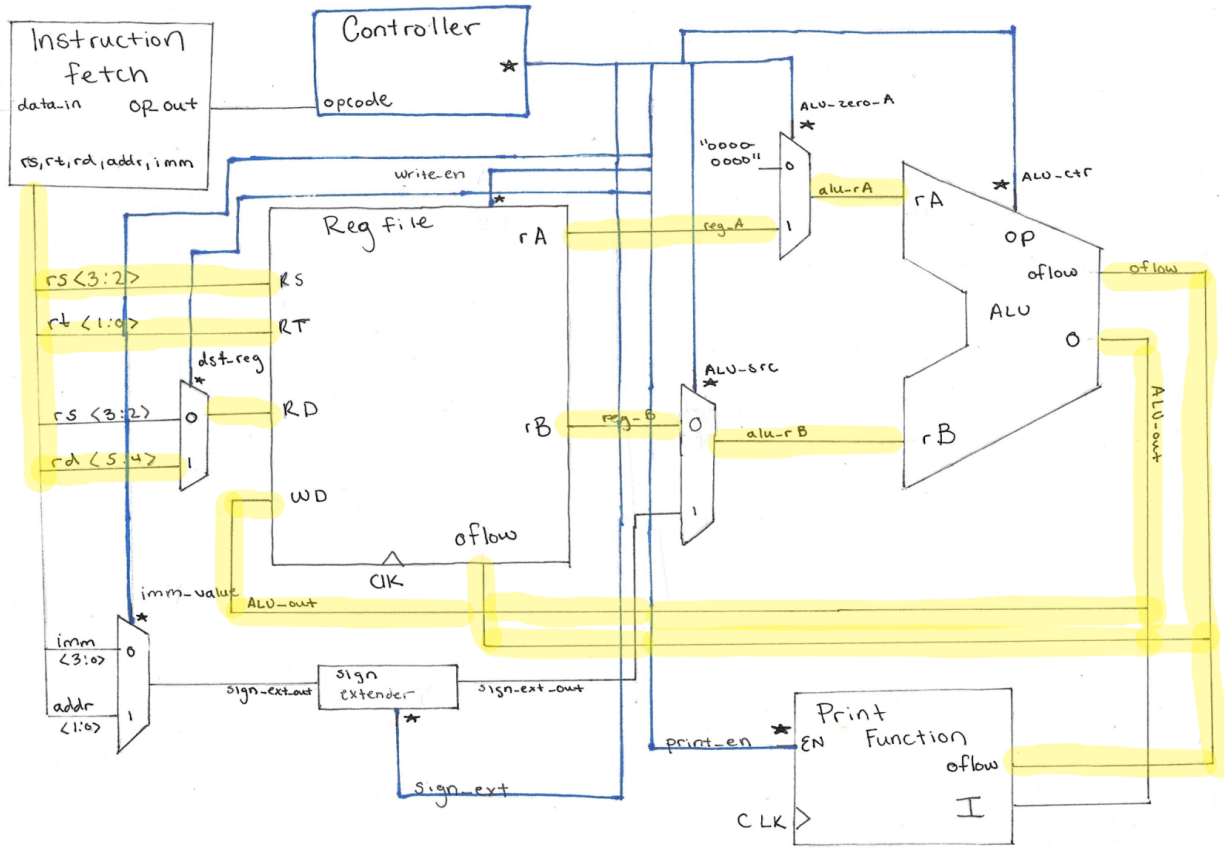
ISA instruction

Bits	7	6	5	4	3	2	1	0
	1	1	R	D	R	S	R	T

DataPath:

The instruction fetch values that will matter will be RS, RT, and RD. RS and RT will be read from in regfile as values to be subtracted in the ALU and RD will be the register that the data is written to. Dst_reg will select RD as RD. Alu_src will select reg_B as input for rB for the ALU. Since reg_B is selected then imm_value and sign_ext doesn't matter. RS will determine the rA value, and alu_zero_A will select reg_A as the input for rA in the ALU. ALU will be set to SUB, and subtract RT from RS. This value will be ALU_out and written to RD. If overflow were to occur, the print function will alert the user and RD will not change.

Dst_reg	Imm_value	Sign_ext	write_en	Alu_src	alu_zero_A	Alu_ctr	print_en
1	X	X	1	0	1	SUB	0



Display

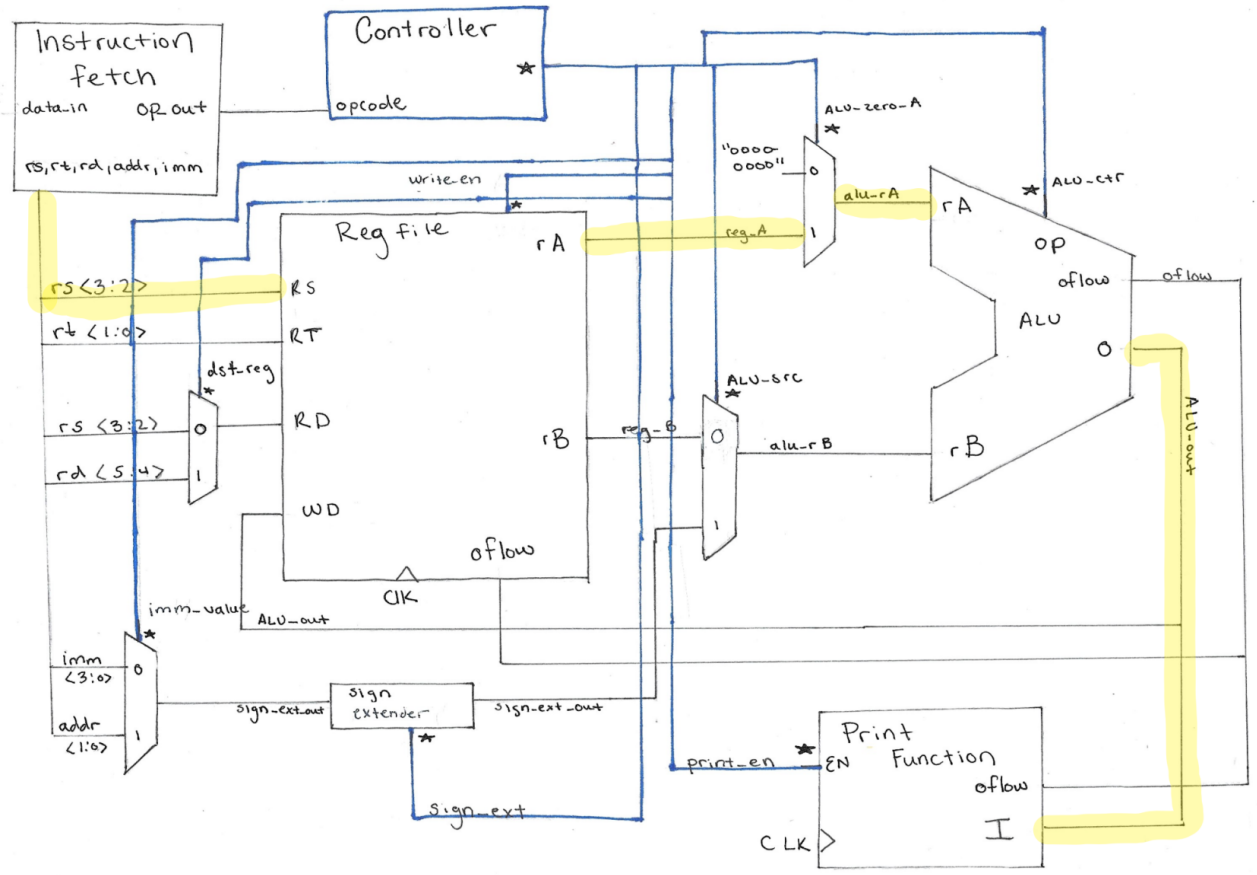
ISA instruction

Bits	7	6	5	4	3	2	1	0
	0	0	1	X	R	S	X	X

DataPath:

The instruction fetch values that will matter will be RS. RS is the register with the data that will be displayed. RS will determine the rA value, and alu_zero_A will select reg_A as the input for rA in the ALU. ALU will be set to Print, and set ALU_out as rA. Since rB will not be used in this ALU operation, the values of imm_value,sign_ext, and alu_src do not matter. Since no data is being written to (write_en = 0) the value of RD does not matter. ALU_out will go to the print function and display value of RS to monitor.

Dst_reg	Imm_value	Sign_ext	write_en	Alu_src	alu_zero_A	Alu_ctr	print_en
X	X	X	0	X	1	Print	1



EDITED ISA

Changes made to ISA will be highlighted

1. **ISA Design**

-Registers are r0, r1, r2, or r3

Instruction	Functionality Description	Example	Binary Representation
La Opcode = 000	Loads address of a label or integer constant into a register	la reg1, addr Example: La $8, 0x0001	000 X rs aa Opcode, register, address
Load Opcode = 001	Load 4-bit immediate number into register	load reg, imNum Example: load $2,(4)	01 rs iiii Compressed opcode, register, immediate

Add
 	
Opcode = 010
	Adds the values of the integers stored in the register given and stores the value in a third register	Add des, reg1, reg2
		
		Example:
		3=2+1
		Add $3, $2, $1
		($3=3)
			10 rd rs rt
			
			Compressed opcode, destination, register1, register 2

Sub
 	
Opcode = 011
	Subtracts the values of the integers stored in the register given and stores the value in a third register	sub des, reg1, reg2
		
		Example:
		1=2-1
		Sub $3, $2, $1
		($3=1)
			11 rd rs rt
			
			Compressed opcode, destination, register1, register 2

Disp Opcode = 100	Display value in a register to the console	disp reg Example: disp $3 -r3 =2 -2 would display	001 X rs XX Coded opcode, register

Changes:

- In order to keep instruction to 8-bits opcode was either compressed to 2-bit or rearranged. Instruction fetch components will decode them properly for the controller.
- Skip and comp were removed
- X's were added to show where meaningless bits will be positioned.

2. **Benchmarks (in binary format)**

Binary Representation :

Instruction		**Binary Representation**
la	Addresses $1 with given address(0000011)	00000111
Load	Register $1 has immediate number 3 loaded into it	01010011
add	Adds two registers, $3 and $1 (1 and 3) and stores the value in a third one $2 (4)	10101101
sub	Subtracts two registers, $2 and $3(4 and 1) and stores the value in a third one $2 (3)	11101011
disp	Displays value which is being stored in register $1	00100100

Changes:

- Binary representation was changed to match the binary representation in the ISA Design table.
- Skip and Comp were removed.

3. **Explanation of why/how these benchmarks test implementation of ISA**

La: The benchmark test will call La and then call the disp instruction, printing the same register loaded in La. If the display instruction prints the expected output, then La was successfully executed.

Load: The benchmark test will call load and then call the disp instruction to print the same register used in load. If the display instruction prints the expected output, then Load was successfully executed.

Add: The benchmark test will call add and then call the disp instruction to print the register denoted as RD in the add instruction. If the display instruction prints the expected output then Add was successfully executed.

Sub: The benchmark test will call sub and then call the disp instruction to print the register denoted as RD in the sub instruction. If the display instruction prints the expected output then Sub was successfully executed.

Disp: The benchmark will display all four registers first. The registers are initialized to zero. If all four instructions print zero, the display instruction is working correctly.

Changes:

- Skip and comp were removed.
- Since skip and comp were removed, the way of testing success was to use display to see if the expected output matched what was displayed. If there is no math, test was not successful.

Testbench

Note: Each test instruction will also have a clock input. This means each instruction will repeat like such:

(“00100000”,’0’) -- won’t print due to clock

(“00100000”,’1’) -- will print 0 with rising edge

This is to create a rising edge. Clock will not be included below, but will be in the testbench file.

Expected outputs of entries in calc_tb.vhdl :

Following entries test all possible combinations for all 5 instructions

Test Display

00100000	Display reg 0: **0**
00100100	Display reg 1: **0**
00100100	Display reg 2: **0**
00101100	Display reg 3: **0**

Test Load Address

--reg 0

00000011	Load address of 3 into reg 0 : “0000000”+ “00000011” = “00000011”
00100000	Display reg 0: **3**
00000001	Load address of 1 into reg 0 : “00000011”+ “00000001” = “00000100”
00100000	Display reg 0: **4**

--reg 1

00000110	Load address of 2 into reg 1 : “00000000” + “00000010” = “00000010”
00100100	Display reg 1: **2**
00000101	Load address of 1 into reg 1 : “00000010”+ “00000001” = “00000011”
00100100	Display reg 1: **3**

--reg 2

00001001	Load address of 1 into reg 2 : “00000000” + “00000001” = “00000001”
00101000	Display reg 2: **1**
00001001	Load address of 1 into reg 2 : “00000001”+ “00000001” = “00000010”
00101000	Display reg2: **2**

--reg 3

00001111	Load address of 3 into reg 3 : “00000000” + “00000011” = “00000011”
00101100	Display reg 3: **3**
00001101	Load address of 1 into reg 3 : “00000011”+ “00000001” = “00000100”
00101100	Display reg3: **4**

Test Load Immediate

--reg 0

01000001	Load 1 into reg 0 : reg 0 = “00000001”
00100000	Display reg 0: **1**
01000100	Load 4 into reg 0 : reg 0 = “00000100”
00100000	Display reg 0: **4**

--reg 1

01011111	Load -1 into reg 1 : reg 1 = “11111111”
00100100	Display reg 1: **-1**
01011101	Load -3 into reg 1 : reg 1 = “11111101”
00100100	Display reg 1: **-3**

--reg 2

01100110	Load 6 into reg 2 : reg 2 = “00000110”
00101000	Display reg 2: **6**
01101001	Load -7 into reg 2 : reg 2 = “11111001”
00101000	Display reg 2: **-7**

--reg 3

01110111	Load 7 into reg 3 : reg 3 = “00000111”
00101100	Display reg 3: **7**
01110010	Load 7 into reg 3 : reg 3 = ”00000010”
00101100	Display reg 3: **2**

Test ADD

-- reg 0 as RD

--reg 0 as RS and RT = other registers

10000000	ADD reg 0 and reg 0 and load into reg 0 : 4 + 4
00100000	Display reg 0: **8**
10000001	ADD reg 0 and reg 1 and load into reg 0 : 8 + -3
00100000	Display reg 0: **5**
10000010	ADD reg 0 and reg 2 and load into reg 0 : 5 + -7
00100000	Display reg 0: **-2**
10000011	ADD reg 0 and reg 3 and load into reg 0 : -2 + 2
00100000	Display reg 0: **0**

--reg 1 as RS and RT = other registers

10000100	ADD reg 1 and reg 0 and load into reg 0 : -3 + 0
00100000	Display reg 0: **-3**
10000101	ADD reg 1 and reg 1 and load into reg 0 : -3 + -3
00100000	Display reg 0: **-6**
10000110	ADD reg 1 and reg 2 and load into reg 0 : -3 + -7
00100000	Display reg 0: **-10**
10000111	ADD reg 1 and reg 3 and load into reg 0 : -3 + 2
00100000	Display reg 0: **-1**

--reg 2 as RS and RT = other registers

10001000	ADD reg 2 and reg 0 and load into reg 0 : -7 + -1
00100000	Display reg 0: **-8**
10001001	ADD reg 2 and reg 1 and load into reg 0 : -7 + -3
00100000	Display reg 0: **-10**
10001010	ADD reg 2 and reg 2 and load into reg 0 : -7 + -7
00100000	Display reg 0: **-14**

10001011	ADD reg 2 and reg 3 and load into reg 0 : -7 + 2
00100000	Display reg 0: **-5**

--reg 3 as RS and RT = other registers

10001100	ADD reg 3 and reg 0 and load into reg 0 : 2+ -5
00100000	Display reg 0: **-3**
10001101	ADD reg 3 and reg 1 and load into reg 0 : 2 + -3
00100000	Display reg 0: **-1**
10001110	ADD reg 3 and reg 2 and load into reg 0 : 2 + -7
00100000	Display reg 0: **-5**
10001111	ADD reg 3 and reg 3 and load into reg 0 : 2 + 2
00100000	Display reg 0: **4**

-- reg 1 as RD

--reg 0 as RS and RT = other registers

10010000	ADD reg 0 and reg 0 and load into reg 1 : 4+4
00100100	Display reg 1: **8**
10010001	ADD reg 0 and reg 1 and load into reg 1 : 4 + 8
00100100	Display reg 1: **12**
10010010	ADD reg 0 and reg 2 and load into reg 1 : 4 + -7
00100100	Display reg 1: **-3**
10010011	ADD reg 0 and reg 3 and load into reg 1 : 4 + 2
00100100	Display reg 1: **6**

--reg 1 as RS and rt = other registers

10010100	ADD reg 1 and reg 0 and load into reg 1 : 6 + 4
00100100	Display reg 1: **10**
10010101	ADD reg 1 and reg 1 and load into reg 1 : 10 + 10
00100100	Display reg 1: **20**

10010110	ADD reg 1 and reg 2 and load into reg 1 : 20 + -7
00100100	Display reg 1: **13**
10010111	ADD reg 1 and reg 3 and load into reg 1 : 13 + 2
00100100	Display reg 1: **15**

--reg 2 as RS and rt = other registers

10011000	ADD reg 2 and reg 0 and load into reg 1 : -7 + 4
00100100	Display reg 1: **-3**
10011001	ADD reg 2 and reg 1 and load into reg 1 : -7 + -3
00100100	Display reg 1: **-10**
10011010	ADD reg 2 and reg 2 and load into reg 1 : -7 + -7
00100100	Display reg 1: **-14**
10011011	ADD reg 2 and reg 3 and load into reg 1 : -7 + 2
00100100	Display reg 1: **-5**

--reg 3 as RS and rt = other registers

10011100	ADD reg 3 and reg 0 and load into reg 1 : 2 + 4
00100100	Display reg 1: **6**
10011101	ADD reg 3 and reg 1 and load into reg 1 : 2 + 6
00100100	Display reg 1: **8**
10011110	ADD reg 3 and reg 2 and load into reg 1 : 2 + -7
00100100	Display reg 1: **-5**
10011111	ADD reg 3 and reg 3 and load into reg 1 : 2 + 2
00100100	Display reg 1: **4**

-- reg 2 as RD

--reg 0 as RS and RT = other registers

10100000	ADD reg 0 and reg 0 and load into reg 2 : 4 + 4
00101000	Display reg 2: **8**

10100001	ADD reg 0 and reg 1 and load into reg 2 : 4 + 4
00101000	Display reg 2: **8**
10100010	ADD reg 0 and reg 2 and load into reg 2 : 4 + 8
00101000	Display reg 2: **12**
10100011	ADD reg 0 and reg 3 and load into reg 2 : 4 + 2
00101000	Display reg 2: **6**

--reg 1 as RS and RT = other registers

10100100	ADD reg 1 and reg 0 and load into reg 2 : 4 + 4
00101000	Display reg 2: **8**
10100101	ADD reg 1 and reg 1 and load into reg 2 : 4 + 4
00101000	Display reg 2: **8**
10100110	ADD reg 1 and reg 2 and load into reg 2 : 4 + 8
00101000	Display reg 2: **12**
10100111	ADD reg 1 and reg 3 and load into reg 2 : 4 + 2
00101000	Display reg 2: **6**

--reg 2 as RS and RT = other registers

10101000	ADD reg 2 and reg 0 and load into reg 2 : 6 + 4
00101000	Display reg 2: **10**
10101001	ADD reg 2 and reg 1 and load into reg 2 : 10 +4
00101000	Display reg 2: **14**
10101010	ADD reg 2 and reg 2 and load into reg 2 : 14 + 14
00101000	Display reg 2: **28**
10101011	ADD reg 2 and reg 3 and load into reg 2 : 28 + 2
00101000	Display reg 2: **30**

--reg 3 as RS and RT = other registers

10101100	ADD reg 3 and reg 0 and load into reg 2 : 2 + 4

00101000	Display reg 2: **6**
10101101	ADD reg 3 and reg 1 and load into reg 2 : 2 + 4
00101000	Display reg 2: **6**
10101110	ADD reg 3 and reg 2 and load into reg 2 : 2 + 6
00101000	Display reg 2: **8**
10101111	ADD reg 3 and reg 3 and load into reg 2 : 2 + 2
00101000	Display reg 2: **4**

-- reg 3 as RD

--reg 0 as RS and RT = other registers

10110000	ADD reg 0 and reg 0 and load into reg 3 : 4 + 4
00101100	Display reg 3: **8**
10110001	ADD reg 0 and reg 1 and load into reg 3 : 4 + 4
00101100	Display reg 3: **8**
10110010	ADD reg 0 and reg 2 and load into reg 3 : 4 + 4
00101100	Display reg 3: **8**
10110011	ADD reg 0 and reg 3 and load into reg 3 : 4 + 8
00101100	Display reg 3: **12**

--reg 1 as RS and RT = other registers

10110100	ADD reg 1 and reg 0 and load into reg 3 : 4 + 4
00101100	Display reg 3: **8**
10110101	ADD reg 1 and reg 1 and load into reg 3 : 4 + 4
00101100	Display reg 3: **8**
10110110	ADD reg 1 and reg 2 and load into reg 3 : 4 + 4
00101100	Display reg 3: **8**
10110111	ADD reg 1 and reg 3 and load into reg 3 : 4 + 8
00101100	Display reg 3: **12**

--reg 2 as RS and RT = other registers

10111000	ADD reg 2 and reg 0 and load into reg 3 : 4 + 4
00101100	Display reg 3: **8**
10111001	ADD reg 2 and reg 1 and load into reg 3 : 4 + 4
00101100	Display reg 3: **8**
10111010	ADD reg 2 and reg 2 and load into reg 3 : 4 + 4
00101100	Display reg 3: **8**
10111011	ADD reg 2 and reg 3 and load into reg 3 : 4 + 8
00101100	Display reg 3: **12**

--reg 3 as RS and RT = other registers

10111100	ADD reg 3 and reg 0 and load into reg 3 : 12 + 4
00101100	Display reg 3: **16**
10111101	ADD reg 3 and reg 1 and load into reg 3 : 16 + 4
00101100	Display reg 3: **20**
10111110	ADD reg 3 and reg 2 and load into reg 3 : 20 + 4
00101100	Display reg 3: **24**
10111111	ADD reg 3 and reg 3 and load into reg 3 : 24 + 24
00101100	Display reg 3: **48**

Test Overflow with ADD instruction

10111111	ADD reg 3 and reg 3 and load into reg 3 : 48 + 48
10111111	ADD reg 3 and reg 3 and load into reg 3 : 96 + 96 Displays **"ERROR: OVERFLOW!"**
00101100	Display reg 3: **96 (**overflow prevents writing from occurring)

Test SUB

-- reg 0 as RD

--reg 0 as RS and RT = other registers

11000000	SUB: reg 0 - reg 0 and load into reg 0 : 4 - 4
00100000	Display reg 0: **0**
11000001	SUB: reg 0 - reg 1 and load into reg 0 : 0 - 4
00100000	Display reg 0: **-4**
11000010	SUB: reg 0 - reg 2 and load into reg 0 : -4 - 4
00100000	Display reg 0: **-8**
11000011	SUB: reg 0 - reg 3 and load into reg 0 : -8 - 96
00100000	Display reg 0: **-104**

--reg 1 as RS and RT = other registers

11000100	SUB: reg 1 - reg 0 and load into reg 0 : 4 - (-104)
00100000	Display reg 0: **108**
11000101	SUB: reg 1 - reg 1 and load into reg 0 : 4 - 4
00100000	Display reg 0: **0**
11000110	SUB: reg 1 - reg 2 and load into reg 0 : 4 - 4
00100000	Display reg 0: **0**
11000111	SUB: reg 1 - reg 3 and load into reg 0 : 4 - 96
00100000	Display reg 0: **-92**

--reg 2 as RS and RT = other registers

11001000	SUB: reg 2 - reg 0 and load into reg 0 : 4 - (-92)
00100000	Display reg 0: **96**
11001001	SUB: reg 2 - reg 1 and load into reg 0 : 4 - 4
00100000	Display reg 0: **0**
11001010	SUB: reg 2 - reg 2 and load into reg 0 : 4 - 4
00100000	Display reg 0: **0**
11001011	SUB: reg 2 - reg 3 and load into reg 0 : 4 - 96
00100000	Display reg 0: **-92**

--reg 3 as RS and RT = other registers

11001100	SUB: reg 3 - reg 0 and load into reg 0 : 96 - (-92) ***Displays overflow error***
00100000	Display reg 0: **-92**
11001101	SUB: reg 3 - reg 1 and load into reg 0 : 96 - 4
00100000	Display reg 0: **92**
11001110	SUB: reg 3 - reg 2 and load into reg 0 : 96 - 4
00100000	Display reg 0: **92**
11001111	SUB: reg 3 - reg 3 and load into reg 0 : 96 - 96
00100000	Display reg 0: **0**

-- reg 1 as RD

--reg 0 as RS and RT = other registers

11010000	SUB: reg 0 - reg 0 and load into reg 1 : 0 - 0
00100100	Display reg 1: **0**
11010001	SUB: reg 0 - reg 1 and load into reg 1 : 0 - 0
00100100	Display reg 1: **0**
11010010	SUB: reg 0 - reg 2 and load into reg 1 : 0 - 4
00100100	Display reg 1: **-4**
11010011	SUB: reg 0 - reg 3 and load into reg 1 : 0 - 96
00100100	Display reg 1: **-96**

--reg 1 as RS and rt = other registers

1110100	SUB: reg 1 - reg 0 and load into reg 1 : -96 - 0
00100100	Display reg 1: **-96**
11010101	SUB: reg 1 - reg 1 and load into reg 1 : -96 - (-96)
00100100	Display reg 1: **0**
11010110	SUB: reg 1 and reg 2 and load into reg 1 : 0 - 4
00100100	Display reg 1: **-4**

11010111	SUB: reg 1 and reg 3 and load into reg 1 : -4 - 96
00100100	Display reg 1: **-100**

--reg 2 as RS and rt = other registers

11011000	SUB: reg 2 - reg 0 and load into reg 1 : 4 - 0
00100100	Display reg 1: **4**
11011001	SUB: reg 2 - reg 1 and load into reg 1 : 4 - 4
00100100	Display reg 1: **0**
11011010	SUB: reg 2 - reg 2 and load into reg 1 : 4 - 4
00100100	Display reg 1: **0**
11011011	SUB: reg 2 - reg 3 and load into reg 1 : 4 - 96
00100100	Display reg 1: **-92**

--reg 3 as RS and rt = other registers

11011100	SUB: reg 3 - reg 0 and load into reg 1 : 96 - 0
00100100	Display reg 1: **96**
11011101	SUB: reg 3 - reg 1 and load into reg 1 : 96 - 96
00100100	Display reg 1: **0**
11011110	SUB: reg 3 - reg 2 and load into reg 1 : 96 - 4
00100100	Display reg 1: **92**
11011111	SUB: reg 3 - reg 3 and load into reg 1 : 96 - 96
00100100	Display reg 1: **0**

-- reg 2 as RD

--reg 0 as RS and RT = other registers

11100000	SUB: reg 0 - reg 0 and load into reg 2 : 0 - 0
00101000	Display reg 2: **0**
11100001	SUB: reg 0 - reg 1 and load into reg 2 : 0 - 0
00101000	Display reg 2: **0**

11100010	SUB: reg 0 - reg 2 and load into reg 2 : 0 - 0
00101000	Display reg 2: **0**
11100011	SUB: reg 0 - reg 3 and load into reg 2 : 0 - 96
00101000	Display reg 2: **-96**

--reg 1 as RS and RT = other registers

11100100	SUB: reg 1 - reg 0 and load into reg 2 : 0 - 0
00101000	Display reg 2: **0**
11100101	SUB: reg 1 - reg 1 and load into reg 2 : 0 - 0
00101000	Display reg 2: **0**
11100110	SUB: reg 1 - reg 2 and load into reg 2 : 0 - 0
00101000	Display reg 2: **0**
11100111	SUB: reg 1 - reg 3 and load into reg 2 : 0 - 96
00101000	Display reg 2: **-96**

--reg 2 as RS and RT = other registers

11101000	SUB: reg 2 - reg 0 and load into reg 2 : -96 - 0
00101000	Display reg 2: **-96**
11101001	SUB: reg 2 - reg 1 and load into reg 2 : -96 - 0
00101000	Display reg 2: **-96**
11101010	SUB: reg 2 - reg 2 and load into reg 2 : -96 - (-96)
00101000	Display reg 2: **0**
11101011	SUB: reg 2 - reg 3 and load into reg 2 : 0 - 96
00101000	Display reg 2: **-96**

--reg 3 as RS and RT = other registers

11101100	SUB: reg 3 - reg 0 and load into reg 2 : 96 - 0
00101000	Display reg 2: **96**
11101101	SUB: reg 3 - reg 1 and load into reg 2 : 96 - 0

00101000	Display reg 2: **96**
11101110	SUB: reg 3 - reg 2 and load into reg 2 : 96 - 96
00101000	Display reg 2: **0**
11101111	SUB: reg 3 - reg 3 and load into reg 2 : 96 - 96
00101000	Display reg 2: **0**

---------- test overflow with SUB instruction----------

11000011	SUB: reg 0 - reg 3 and load into reg 0 : 0 - 96 = -96
11000011	SUB: reg 0 - reg 3 and load into reg 0 : -96 - 96 ***Displays overflow error***
00100000	Display reg 0: **-96**

-- reg 3 as RD

--reg 0 as RS and RT = other registers

11110000	SUB: reg 0 - reg 0 and load into reg 3 : -96 - (-96)
00101100	Display reg 3: **0**
11110001	SUB: reg 0 - reg 1 and load into reg 3 : -96 - 0
00101100	Display reg 3: **-96**
11110010	SUB: reg 0 - reg 2 and load into reg 3 : -96 - 0
00101100	Display reg 3: **-96**
11110011	SUB: reg 0 - reg 3 and load into reg 3 : - 96 - (-96)
00101100	Display reg 3: **0**

--reg 1 as RS and RT = other registers

11110100	SUB: reg 1 - reg 0 and load into reg 3 : 0 - (-96)
00101100	Display reg 3: **96**
11110101	SUB: reg 1 - reg 1 and load into reg 3 : 96 - 96
00101100	Display reg 3: **0**
11110110	SUB: reg 1 - reg 2 and load into reg 3 : 0 - 0

11110100	SUB: reg 1 - reg 0 and load into reg 3 : 0 - (-96)
00101100	Display reg 3: **96**
11110101	SUB: reg 1 - reg 1 and load into reg 3 : 96 - 96
00101100	Display reg 3: **0**
11110111	SUB: reg 1 - reg 3 and load into reg 3 : 0 - 0
00101100	Display reg 3: **0**

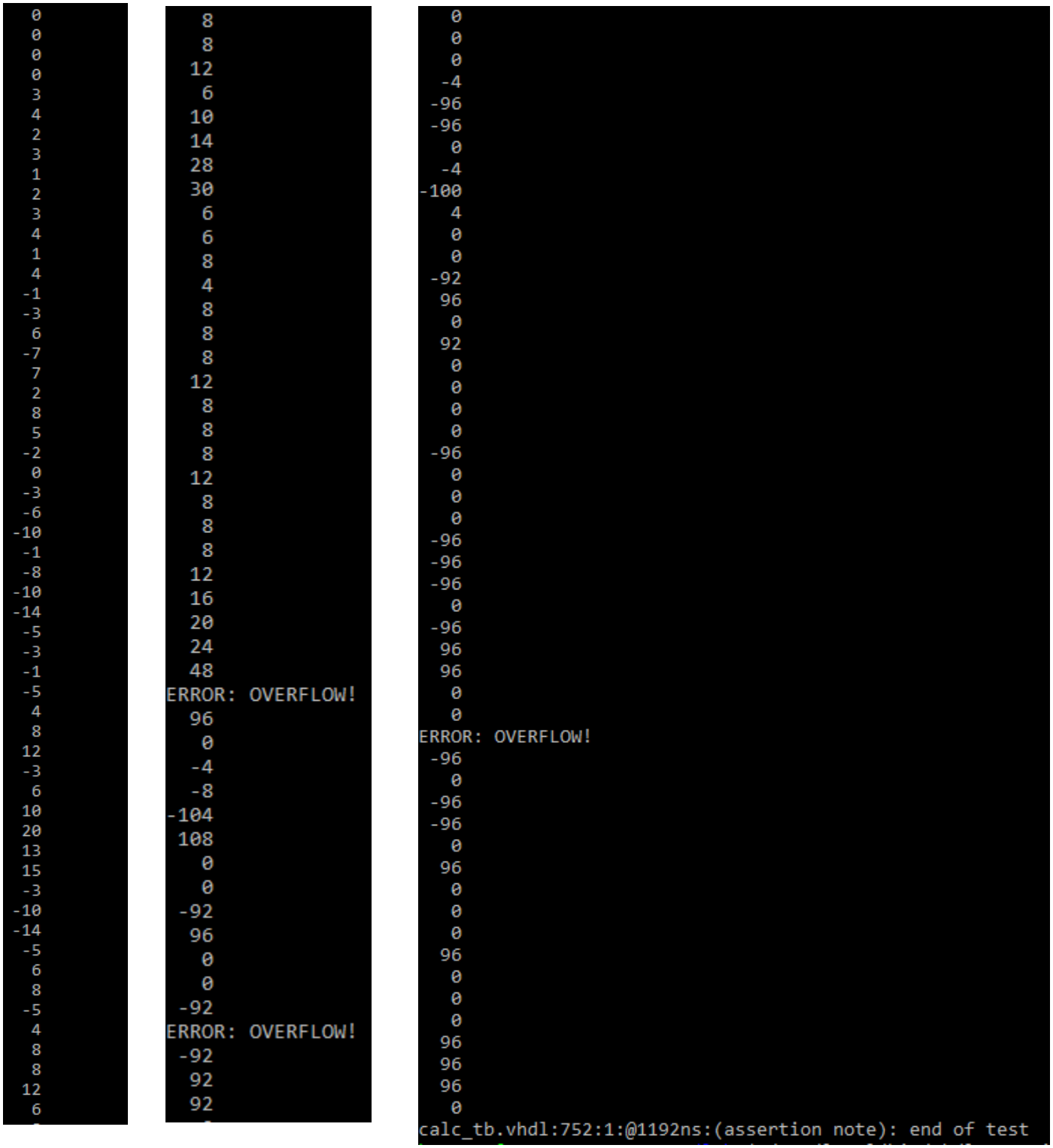
--reg 2 as RS and RT = other registers

11111000	SUB: reg 2 - reg 0 and load into reg 3 : 0 - (-96)
00101100	Display reg 3: **96**
11111001	SUB: reg 2 - reg 1 and load into reg 3 : 0 - 0
00101100	Display reg 3: **0**
11111010	SUB: reg 2 - reg 2 and load into reg 3 : 0 - 0
00101100	Display reg 3: **0**
11111011	SUB: reg 2 - reg 3 and load into reg 3 : 0 - 0
00101100	Display reg 3: **0**

--reg 3 as RS and RT = other registers

11111100	SUB: reg 3 - reg 0 and load into reg 3 : 0 - (-96)
00101100	Display reg 3: **96**
11111101	SUB: reg 3 - reg 1 and load into reg 3 : 96 - 0
00101100	Display reg 3: **96**
11111110	SUB: reg 3 - reg 2 and load into reg 3 : 96 - 0
00101100	Display reg 3: **96**
11111111	SUB: reg 3 - reg 3 and load into reg 3 : 96 - 96
00101100	Display reg 3: **0**

Output from calc_tb.vhdl:



The expected outputs match the output from the testbench file. Therefore, all instructions have executed successfully. Testbench of individual components can be found in the appendix.

Appendix

A. Inst_fet.vhdl code for Op_out

```
process(data_in)
begin


                if (data_in(7 downto 6) = "00") then   -- "00" can be two different instruction (Load Addr or display)

                        if (data_in(5) = '0') then  -- data_in(5) decides MSB for op_out  when "00"
                                op_out <= "000"; -- instruction code for load address
                        else
                                op_out <= "100"; -- instruction code for display
                        end if;
                else                                -- instrutions codes for load immediate, add, and sub
                        op_out(2) <= '0';
                        op_out(1 downto 0) <= data_in(7 downto 6);

                end if;
```

B. Table of Control signals and Instructions Summarized

Instruction	alu_ctr	dst_reg	imm_value	write_en	sign_ext	alu_src	alu_zero_A	print_en
Load Addr	ADD (00)	0	1	1	0	1	1	0
Load Imm	ADD (00)	1	0	1	1	1	0	0
ADD	ADD (00)	1	**X**	1	**X**	0	1	0
SUB	SUB (01)	1	**X**	1	**X**	0	1	0
Display	Print (10)	**X**	**X**	0	**X**	**X**	1	1

C. Testbench of Instruction Fetch and output

Entries:
(data_in(input) , op_out , RS , RT , RD , Addr, Imm)
("00000000", "000", "00","00","00","00", "0000"), --000 opcode output
("00101010", "100", "10","10","10","10", "1010"), -- 100 opcode output
("01001100", "001", "11","00","00","00", "1100"), --001 opcode output
("11111111", "011", "11","11","11","11", "1111"), --011 opcode output
("10000000", "010", "00","00","00","00", "0000") --010 opcode output

Output from inst_tb.vhdl:

```
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -a --ieee=synopsys inst_fet.vhdl
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -a --ieee=synopsys inst_tb.vhdl
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -e --ieee=synopsys inst_tb
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -r --ieee=synopsys inst_tb --vcd=inst_tb.vcd
inst_tb.vhdl:64:1:@5ns:(assertion note): end of test
```

Testbench was successful as shown above by no assertion errors.

D. Testbench of Controller and Output

Entries:

Opcode (input)	Alu_ctr (output)	Dst_reg (output)	Imm_value (output)	Write_en (output)	Sign_ext (output)	Alu_src (output)	alu_zero_A (output)	Print_en (output)
“000”	“00”	‘0’	‘1’	‘1’	‘0’	‘1’	‘1’	‘0’
“001”	“00”	‘1’	‘0’	‘1’	‘1’	‘1’	‘0’	‘0’
“010”	“00”	‘1’	‘1’	‘1’	‘0’	‘0’	‘1’	‘0’
“011”	“01”	‘1’	‘0’	‘1’	‘1’	‘0’	‘1’	‘0’
“100”	“10”	‘0’	‘1’	‘0’	‘0’	‘1’	‘1’	‘1’

Output from control_tb.vhdl:

```
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -a --ieee=synopsys control.vhdl
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -a --ieee=synopsys control_tb.vhdl
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -e --ieee=synopsys control_tb
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -r --ieee=synopsys control_tb --vcd=control_tb.vcd
control_tb.vhdl:77:1:@5ns:(assertion note): end of test
```

Testbench was successful as shown above by no assertion errors.

E. Testbench of reg_tb.vhdl and Output

Reg_tb.vhdl is the testbench of reg_8.vhdl that regfile.vhdl uses as register behavioral code.

Entries:

(In _data, clock , oflow , Out_data(output))

-- overflow isnt detected

("00000010", '1','0', "00000000"), -- write to register with no rising edge

("00000010", '1','0', "00000010"), -- write to register at rising edge

("00000110", '0','0', "00000010"), -- write to register with no rising edge and enable = 0

("00000110", '0','0', "00000010"), -- write to register at rising edge and enable = 0

("00000011", '1','0', "00000010"), -- write to register with no rising edge
("00000011", '1','0', "00000011"), -- write to register at rising edge
("00000111", '0','0', "00000011"), -- write to register with no rising edge and enable = 0
("00000111", '0','0', "00000011"), -- write to register at rising edge and enable = 0

--overflow is detected : no writing should occur
("00000010", '1','1', "00000011"), -- try to write to register with no rising edge
("00000010", '1','1', "00000011"), -- try to write to register at rising edge
("00000110", '0','1', "00000011"), -- try to write to register with no rising edge and enable = 0
("00000110", '0','1', "00000011"), -- try to write to register at rising edge and enable = 0
("00000011", '1','1', "00000011"), -- try to write to register with no rising edge
("00000011", '1','1', "00000011"), -- try to write to register at rising edge
("00000111", '0','1', "00000011"), -- try to write to register with no rising edge and enable = 0
("00000111", '0','1', "00000011"), -- try to write to register at rising edge and enable = 0

Output from reg_tb.vhdl:

```
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -e --ieee=synopsys reg_tb
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -r --ieee=synopsys reg_tb --vcd=reg_tb.vcd --stop-time=32ns
reg_tb.vhdl:87:1:@32ns:(assertion note): end of test
/usr/local/bin/ghdl:info: simulation stopped by --stop-time @32ns
```

Testbench was successful as shown above by no assertion errors.

F. Testbench of Register File and Output

Entries:

(In _data, RS, RT, RD, WE, oflow , reg 1 value(output), reg 2 value(output))

-- test intialization
-- reads are combo so clock should not interfere
-- no writing should occur
-- rs and rt read reg 0
("11111111", "00","01","00",'0','0', "00000000","00000000"), -- no rising edge and en = 0
("11111111", "00","01","00",'0','0', "00000000","00000000"), -- rising edge and en = 0
--rs and rt read reg 1
("11111111", "01","00","01",'0', '0', "00000000","00000000"), -- no rising edge and en = 0
("11111111", "01","00","01",'0', '0', "00000000","00000000"), -- rising edge and en = 0
-- rs and rt read reg 2
("11111111", "10","11","10",'0','0', "00000000","00000000"), -- no rising edge and en = 0
("11111111", "10","11","10",'0','0', "00000000","00000000"), -- rising edge and en = 0
-- rs and rt read reg 3
("11111111", "11","10","11",'0','0', "00000000","00000000"), -- no rising edge and en = 0
("11111111", "11","10","11",'0','0', "00000000","00000000"), -- rising edge and en = 0

--test writing with data set 1
-- writing should occur when rising edge and en = 1

--write to reg 0
("11111111", "00","01","00",'1', '0', "00000000","00000000"), -- no rising edge and en = 1
("11111111", "00","01","00",'1', '0', "11111111","00000000"), -- rising edge and en = 1
-- if there is overflow no writing occurs
("11000111", "00","01","00",'1', '1', "11111111","00000000"), -- no rising edge and en = 1 and overflow detected
("11000111", "00","01","00",'1', '1', "11111111","00000000"), -- rising edge and en = 1 and overflow detected

-- write to reg 1
("11111110", "01","00","01",'1','0', "00000000","11111111"), -- no rising edge and en = 1
("11111110", "01","00","01",'1','0', "11111110","11111111"), -- rising edge and en = 1
-- if there is overflow no writing occurs
("00001110", "01","00","01",'1','1', "11111110","11111111"), -- no rising edge and en = 1 and overflow detected
("00001110", "01","00","01",'1','1', "11111110","11111111"), -- rising edge and en = 1 and overflow detected

-- write to reg 2
("11111100", "10","11","10",'1','0', "00000000","00000000"), -- no rising edge and en = 1
("11111100", "10","11","10",'1','0', "11111100","00000000"), -- rising edge and en = 1
--if there is overflow no writing occurs
("10011100", "10","11","10",'1','1', "11111100","00000000"), -- no rising edge and en = 1 and overflow detected
("10011100", "10","11","10",'1','1', "11111100","00000000"), -- rising edge and en = 1 and overflow detected

-- write to reg 3
("11111000", "11","10","11",'1','0', "00000000","11111100"), -- no rising edge and en = 1
("11111000", "11","10","11",'1','0', "11111000","11111100"), -- rising edge and en = 1
-- if there is overflow no writing occurs
("00000000", "11","10","11",'1','1', "11111000","11111100"), -- no rising edge and en = 1 and overflow detected
("00000000", "11","10","11",'1','1', "11111000","11111100"), -- rising edge and en = 1 and oveflow detected

-- test writing with data set 2
-- writing should occur when rising edge and en = 1

--write to reg 0
("00001111", "00","01","00",'1','0', "11111111","11111110"), -- no rising edge and en = 1
("00001111", "00","01","00",'1','0', "00001111","11111110"), -- rising edge and en = 1
-- if there is overflow no writing occurs

("11111111", "00","01","00",'1','1', "00001111","11111110"), -- no rising edge and en = 1 and overflow detected
("11111111", "00","01","00",'1','1', "00001111","11111110"), -- rising edge and en = 1 and overflow detected

--write to reg 1
("00011111", "01","00","01",'1','0', "11111110","00001111"), -- no rising edge and en = 1
("00011111", "01","00","01",'1','0', "00011111","00001111"), -- rising edge and en = 1
-- if there is overflow no writing occurs
("11111111", "01","00","01",'1','1', "00011111","00001111"), -- no rising edge and en = 1 and overflow detected
("11111111", "01","00","01",'1','1', "00011111","00001111"), -- rising edge and en = 1 and overflow detected

-- write to reg 2
("00111111", "10","11","10",'1','0', "11111100","11111000"), -- no rising edge and en = 1
("00111111", "10","11","10",'1','0', "00111111","11111000"), -- rising edge and en = 1
-- if there is overflow no writing occurs
("11111111", "10","11","10",'1','1', "00111111","11111000"), -- no rising edge and en = 1 and overflow detected
("11111111", "10","11","10",'1','1', "00111111","11111000"), -- rising edge and en = 1 and overflow detected

-- write to reg 3
("01111111", "11","10","11",'1','0', "11111000","00111111"), -- no rising edge and en = 1
("01111111", "11","10","11",'1','0', "01111111","00111111"), -- rising edge and en = 1
-- if there is overflow no writing occurs
("11111111", "11","10","11",'1','1', "01111111","00111111"), -- no rising edge and en = 1 and overflow detected
("11111111", "11","10","11",'1','1', "01111111","00111111") -- rising edge and en = 1 and overflow detected

Output from regfile_tb.vhdl:

```
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -a --ieee=synopsys regfile.vhdl
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -a --ieee=synopsys regfile_tb.vhdl
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -e --ieee=synopsys regfile_tb
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -r --ieee=synopsys regfile_tb --vcd=regfile_tb.vcd --stop-time=81ns
regfile_tb.vhdl:155:1:@80ns:(assertion note): end of test
/usr/local/bin/ghdl:info: simulation stopped by --stop-time @80ns
```

Testbench was successful as shown above by no assertion errors.

G. Testbench of ALU and Output

Entries:

(reg 1 value, reg 2 value , OP select, , oflow (output), ALU_out (output))

--Test add
("00000000", "00000001", "00", '0', "00000001"), -- 0 + 1 no overflow
("11111111", "11111111", "00", '0', "11111110"), -- -1 + -1 no overflow

--Test sub
("00000001", "00000000", "01", '0', "00000001"), -- 1 - 0 no overflow
("00011110", "01100100", "01", '0', "10111010"), -- 30 - 100 no overflow

--test overflow
("10000001", "00001010", "01", '1', "01110111"), -- -127 - (10) overflow should occur
("10010100", "01100100", "01", '1', "00110000"), -- -108 - (100) overflow should occur
("01111111", "00000001", "00", '1', "10000000"), -- -127 + 1 overflow should occur
("01100100", "01100100", "00", '1', "11001000"), -- 100 + 100 overflow should occur

--Test print
("00000001", "00000000", "10", '0', "00000001"), -- output should be equal to rA
("11111111", "00000001", "11", '0', "11111111") -- output should be equal to rA

Output from alu_tb.vhdl:

```
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -a --ieee=synopsys alu.vhdl
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -a --ieee=synopsys alu_tb.vhdl
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -e --ieee=synopsys alu_tb
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -r --ieee=synopsys alu_tb --vcd=alu_tb.vcd
alu_tb.vhdl:84:1:@10ns:(assertion note): end of test
```

Testbench was successful as shown above by no assertion errors.

H. Testbench of Print Function and Output

Entries:

(In_data, clock, oflow) -- outputs are printed

("00000001", '0', '0'), -- no print should occur
("00000001", '0', '0'), -- no print should occur en = 0
("00000001", '1', '0'), -- no print should occur
("00000001", '1', '0'), -- print " 1"
("11111111", '0', '0'), -- no printing should occur
("11111111", '0', '0'), -- no printing should occur en = 0
("11111111", '1', '0'), -- no printing should occur
("11111111", '1', '0'), -- print " -1"
("00000011", '0', '1'), -- no printing should occur
("00000011", '0', '1') -- print overflow error

-- note: oflow and print_en will never be '1' at the same time due to ALU operations
-- so it is not included in testbench

Output from print_tb.vhdl:

```
kpperry@DESKTOP-O3E7411:~/lab3$ /usr/local/bin/ghdl -r --ieee=synopsys print_tb --vcd=print_tb.vcd --stop-time=20ns
../../src/ieee/v93/numeric_std-body.vhdl:2124:7:@0ms:(assertion warning): NUMERIC_STD.TO_INTEGER: metavalue detected, returning 0
   1
  -1
ERROR: OVERFLOW!
print_tb.vhdl:77:1:@20ns:(assertion note): end of test
/usr/local/bin/ghdl:info: simulation stopped by --stop-time @20ns
```

Testbench was successful as shown above by expected output matching what was printed.