LC-3++, a Modified LC-3 Machine

Date: 12/07/2016

Name: Kexuan Zou kzou3@illinois.edu

Dawei Wang dwang56@illinois.edu

Table of Contents

Introduction

The intel FPGA board has a resizable SRAM that can be easily modified, this gives the chance to build and simulate a list of machines based on the interpretation of pre-initialized memory content, such as LC-3, MIPS, and x86. There are, however, key difference between the above three assembly machines: LC-3 takes its data width as 16-bit binary strings, whereas MIPS and x86 takes 32-bit strings (and x86_64 expand each instruction to 64-bits). 16-bit instructions have many drawbacks and weaknesses. The first is its incompetence to house a lengthy instruction set. For example, LC-3 only takes the 4 most significant bits as its opcode, yet MIPS has 6 bits of opcode plus 6 more bits of funct code. Another disadvantage is that 16-bit as static data (that can be used by the program) can only represent 2's complement integers. On the other hand, MIPS can take advantage of the 32-bit data width and add IEEE 754 standard floating numbers to its collection.

Therefore, to add floating number and other functionalities to LC-3, we expanded its data-width to 32-bit and built up a Von Neumann machine capable of doing both integer and floating point operations. As this machine is based more on LC-3 rather than MIPS or x86 (with no pipelining or branch prediction available), we named ours "LC-3++".

The LC-3++ assembly language

Format

The input file that the compiler takes is in the form of assembly code. The format is similar to the structure of MIPS or x86 assembly language and can be easily translated to 32-bit wide binary streams. Like any assembly language, the grammatical structure of this language is rigid so that in the process of assembling, there is no need to construct a parsing tree. The assembly language is divided into 2 main parts: a data section, where all global variables are declared before the start of the program, and a function section, where the machine starts to read the instructions line by line and process. A commented sample piece of code is provided below.

```
@text                //start of memory initialization
@data                //start of global data section
int: x #4            //declare global variable with value
```

```
int: y #13
int: z #7
float: a #4.25
@func                //start of function section
and $i0, $i0, #0     //i0 <= i0 & 0
and $i1, $i1, #0
and $i2, $i2, #0
ld $i0, x            //load x := 4 into i0
ld $i1, y
add $i2, $i0, $i1    //i2 := i0 + i1;
ld $i0, z
add $i2, $i2, $i0
ld $i0, a
add $i2, $i2, $i0
rnd $i2, $f1
pause                //halt the program
@end                 //end of memory initialization
```

Register layout

The microprocessor uses a total of 20 registers varying in length from 2 to 32 bits. Unlike the LC-3 and MIPS assembly language, only 3 general-purpose registers are used for each of the three data types because generally each instruction can be completed with the help of no more than 3 registers. In hardware layout the registers are grouped into sections: a functional register section contains $ze, $se, $co and $re, a general purpose section contains the 9 registers, and a LOHI register section contains $lo and $hi. Below is a chart and a diagram of the registers used

Name	Number	Size (bits)	Explanation
$ze	$0	32	Zero register storing 0x00000000
$se	$1	32	Sentinel register storing 0x80000000
$i0 - $i2	$2 - $4	32	Integer registers
$f0 - $f2	$5 - $7	32	Floating point registers
$t0 - $t2	$8 - $10	32	Temp registers that are only used to store / load data

$lo	$11	32	Function register
$hi	$12	32	Function register
$co	$13	2	Comparison register
$re	$14	32	Return address

Instruction set

Instructions are grouped by their structures into three types: R, D and J. Each R (Register) –type instruction starts with a 6-bit opcode, ends with a 5-bit data type code and 6 funct code, and specifies 2 source registers ($sr) and a destination register($dr). Each D (Data) –type instruction specifies two registers and a 16-bit constant value or offset from the current PC; J (Jump) –type instructions have a 6-bit opcode and a 16-bit offset. Below is the detailed layout of each instruction.

	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
add $dr, $sr1, $sr2	000000	dr	sr1	sr2	00000	000000
	[31:26]	[25:21]	[20:16]	[15:0]		
add $dr, $sr, c	000001	dr	sr	c		
	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
sub $dr, $sr1, $sr2	000010	dr	sr1	sr2	00000	000001
	[31:26]	[25:21]	[20:16]	[15:0]		
sub $dr, $sr, c	000011	dr	sr	c		

	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
mul $dr, $sr1, $sr2	000100	dr	sr1	sr2	00000	010000
	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
div $sr1, $sr2	000101	00000	sr1	sr2	00000	100001
	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
addf $dr, $sr1, $sr2	000110	dr	sr1	sr2	00001	000000
	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
subf $dr, $sr1, $sr2	000111	dr	sr1	sr2	00001	000001
	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
mulf $dr, $sr1, $sr2	001000	dr	sr1	sr2	00001	000010
	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
divf $sr1, $sr2	001001	00000	sr1	sr2	00001	000011
	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
and $dr, $sr1, $sr2	001010	dr	sr1	sr2	00000	000110
	[31:26]	[25:21]	[20:16]	[15:0]		
and $dr, $sr, c	001011	dr	sr	c		
	[31:26]	[25:21]	[20:16]	[15:0]		
ld $dr, LABEL	001100	dr	00000	offset[15:0]		
	[31:26]	[25:21]	[20:16]	[15:0]		
li $dr, c	001101	dr	00000	c		
	[31:26]	[25:21]	[20:16]	[15:0]		
lr $dr, $sr, c	001110	dr	sr	c		

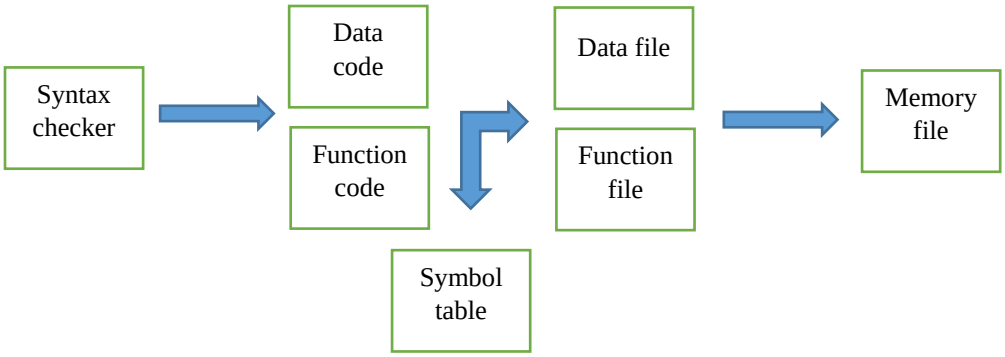
	[31:26]	[25:21]	[20:16]	[15:0]		
ldf $dr, c	001111	dr	00000	c		
	[31:26]	[25:21]	[20:16]	[15:0]		
lrf $dr, $sr, c	010000	dr	sr	c		
	[31:26]	[25:21]	[20:16]	[15:0]		
sa $sr, $br, c	010001	sr	br	c		
	[31:26]	[25:21]	[20:16]	[15:0]		
saf $sr, $br, c	010010	sr	br	c		
	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
mv $dr, $sr	010011	dr	sr			
	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
mvhi $dr	010100	dr	hi			
	[31:26]	[25:21]	[20:16]	[15:11]	[10:6]	[5:0]
mvlo $dr	010101	dr	lo			
	[31:26]	[25:21]	[20:16]	[15:0]		
sftl $dr, $sr, c	010110	dr	sr	c		
	[31:26]	[25:21]	[20:16]	[15:0]		
sftr $dr, $sr, c	010111	dr	sr	c		
	[31:26]	[25:21]	[20:16]	[15:0]		
jsr c	011000	re	0	c		
	[31:26]	[25:21]	[20:0]			
ret	011001	re	0			

	[31:26]	[25:23]	[22:21]	[20:16]	[15:0]	
ifp c / ifz c / ifn c	011010	000	cond	0	c	
	[31:26]	[25:21]	[20:16]	[15:0]		
cmp $sr1, $sr2	011011	sr1	sr2	0		
	[31:26]	[25:21]	[20:16]	[15:0]		
cmpf $sr1, $sr2	011100	sr1	sr2	0		
	[31:26]	[25:21]	[20:16]			
rnd	011101	dr	sr			
	[31:26]	[25:16]		[15:0]		
pse	111111	0		xffff		
	[31:26]	[25:21]	[20:16]	[15:0]		
out	011110	0	sr	0		

Assembler (C)

A brief description

After the .asm file is created, an assembler is required to “translate” the human readable language into a machine friendly, 32-bit wide binary strings to fit into the SRAM so that at time of execution, memory has already been initialized. This requires a I/O to sort the file into several sections (in this case data section and function section), analyze the files line by line to check for syntax errors and generate a symbol table (at the first cycle of compilation). Then the assembler translates the code into binary strings and write to the .mif file, which will be used to initialize the memory block with the desired data. A flowchart is shown below to further articulate the process.



Functions

1. Remove space

   Signature: `void removeSpace(char* source)`

   Parameters: pointer to a char array.

   Return: none.

   Description: This function removes spaces and newline characters in a given char array. Note that if one uses windows system to write the source file, '\r' may occur in the buffer rather than '\n'.

2. Decimal to binary converter

   Signature: `void dectoBin(char* buffer, int digits)`

   Parameters: pointer to the char array (initialized to contain the decimal string), length of the binary string after conversion.

   Return: none.

   Description: This function changes the decimal passed into the function as parameter to a binary string and a length to specify the length of the output. The string buffer initially contains the decimal string to be converted, i.e. "16" as in ASCII 31 and 36, and after conversion the output contains the corresponding binary string. This string can be later used to concatenate with other data. First the decimal string is converted into its corresponding integer and later cast as a hexadecimal string, then by using the look-up table the binary string is generated.

3. Integer to hexadecimal converter

   Signature: `void inttoHex(char* buffer, int val, int line)`

   Parameters: pointer to the char array, line title in integer, total line count.

   Return: none.

Description: This function convert a integer val to hexadecimal and is used to generate the line title used by the .mif file. The integer val is fed into the function by a for loop.

4. Line counter

   Signature: `int countlines(char *fname)`

   Parameters: file name pointer

   Return: total line count in the file.

   Description: Counts the total lines of code in the file specified by the file name.

5. Data.asm generator

   Signature: `void getData(char *fname)`

   Parameters: file name pointer

   Return: none.

   Description: This function reads from the fname and extracts the global data section (from "@data" to "@func") into the data.asm file.

6. Function parser

   Signature: `void parseFunc(char buffer[], int off, int line)`

   Parameters: string buffer, the offset generated from the symbol table, line number

   Return: none

   Description: This function translates each line into a 32-bit binary string according to the instruction set. Since each line starts with a opcode, it is easy to sort the instructions into several categories and translate them. If it's a load instruction, the offset is translated into binary string, and loaded into the correct spot.

7. Func.asm generator

   Signature: `void getFunc(char *fname)`

   Parameters: file name pointer

   Return: none.

   Description: This function reads from the fname and extracts the function section (from "@func" to "@end") into the func.asm file.

8. Data parser

   Signature: `void parseData(char buffer[], int line)`

Parameters: string buffer, line number

Return: none

Description: This function reads from string buffer and writes the data as a binary string. The function should distinguish between data types, i.e. int is stored as 32-bit 2's complement and float is stored as 32-bit IEEE 754.

9. Data writer

   Signature: `void writeData(char *fname, int n)`

   Parameters: file name, total line count

   Return: none

   Description: This function iteratively calls data parser to parse the data.asm line by line and send the data to std::out. After the execution the binary string is written to data.dat.

10. Function writer

    Signature: `void writeFunc(char *fname, int table[], int n, int offset)`

    Parameters: file name, symbol table, line count, offset to the data lines.

    Return: none

    Description: This function iteratively calls function parser to parse the func.asm line by line and send the data to std::out. After the execution the binary string is written to func.dat.

11. Symbol table generator

    Signature: `void getTable(int table[], int dataLine, int funcLine)`

    Parameters: An empty symbol table, data line count, function line count.

    Return: none

    Description: This function generates the symbol table from func.asm and data.asm. The symbol table is declared inside the main function. For a detailed implementation of the function, refer to *algorithms and techniques* section.

12. Syntax checker

    Signature: `void checkError(char* filename)`

    Parameters: file name

    Return: none.

    Description: This function calls a series of functions to check for syntax errors and compile errors. A counter is set in the function so that the function can check multiple errors at a time and

returns the total error count. Since the assembly language is rigidly structured, various instructions can be categorized under the same error-checking helper function. Error prompt contains line number and an instance of the error. For example, below is a sample prompt:

```
Error: In line 8: add $i2, $i1, $i5
Invalid use of register
Error: In line 12: ld $i2, $i1
Invalid use of variable.
2 errors generated.
```

13. Merge files

    Signature: `void mergeFile(char* fname1, char* fname2, char* fname3)`

    Parameters: 2 source file names and a destination file name.

    Return: none.

    Description: Merges 2 files into a third file and remove the 2 source files.

14. .mif getter

    Signature: `void getMif(char* source, char* dest, int size)`

    Parameters: source file name, destination file name, line count of the file.

    Return: none

    Description: Generate the .mif file given the binary string file memory.dat. For a description of the .mif file, refer to *memory initialization file* section. After all is done, a line is displayed on the terminal:

    ```
    memory.sv generated.  Line count: 10. Memory usage: 40 Bytes.
    ```

15. Main function

    Description: Calls the helper functions. First remove conflicting executable files (if they exist). Next, print to std::out to ask the user to enter source code name. Then reads the source file and calls helper function to generate the .mif file.

Memory initialization file

Memory initialization file (.mif) is an ASCII file that specifies the initial content of the SRAM block. This file loads the binary string into SRAM at compile time so that by the time the

program starts to execute, the instructions are already loaded into the memory. Below is a .mif file generated by the sample .asm file displayed above with comments:

```
DEPTH = 19;               //the size of memory in addressability
WIDTH = 32;               //the size of data in bits
ADDRESS_RADIX = HEX;      //the radix for address
DATA_RADIX = BIN;         //the radix for data
CONTENT
BEGIN
00 : 10001000000000000000000000000000;
01 : 10001000001000010000000000000000;
02 : 10001000010000100000000000000000;
03 : 00000100000000000000000000001100;
04 : 00000100001000000000000000001100;
05 : 00000000010000000000100000000000;
06 : 00000100000000000000000000001011;
07 : 00000000010000100000000000000000;
08 : 00000100000000000000000000001010;
09 : 00000000010000100000000000000000;
0A : 10000000010000100000000000000001;
0B : 10001000000000000000000000000000;
0C : 10001000001000010000000000000000;
0D : 10001000010000100000000000000000;
0E : 11111111111111111111111111111111;
0F : 00000000000000000000000000000100;
10 : 00000000000000000000000000001101;
11 : 00000000000000000000000000000111;
12 : 11111111111111111111111111100111;

END;
```

Test memory as initialized arrays

Memory initialization files are a helpful way to initialize the SRAM chip on FPGA with values. However, there is a major issue: .mif files are not compatible with ModelSim RTL simulation program and SRAM sometimes suffer from stability issues. Therefore, an easier method is to generate a test memory file that, with the help of corresponding functions, initialize 32-bit wide arrays that contain the above values. The above .asm file generate the following .sv file, which can be later added to the top level:

```
`include "SLC3PP_2.sv"
import SLC3PP_2::*;

module test_memory ( input                  clk,
                            input          reset,
                     inout  [31:0]  I_O,
                     input  [31:0]  A,
                     input          CE,
                                    OE,
                                    WE );

   parameter size = 256;

   logic [31:0] mem_array [size-1:0];
   logic [31:0] mem_out;
   logic [31:0] I_O_wire;

   assign mem_out = mem_array[A[7:0]];

   always_comb
   begin
      I_O_wire = 32'bZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ;

      if (~CE && ~OE && WE) begin
            I_O_wire[31:0] = mem_out[31:0];
        end
```

```
    end

    always_ff @ (posedge clk or posedge reset) begin
         if(reset) begin
mem_array[0] <= opANDi(i0,i0,16'b0000000000000000);
mem_array[1] <= opANDi(i1,i1,16'b0000000000000000);
mem_array[2] <= opANDi(i2,i2,16'b0000000000000000);
mem_array[3] <= opLD(i0,16'b0000000000000110);
mem_array[4] <= opLD(i1,16'b0000000000000110);
mem_array[5] <= opADD(i2,i0,i1);
mem_array[6] <= opLD(i0,16'b0000000000000110);
mem_array[7] <= opADD(i2,i2,i0);
mem_array[8] <= opRND(i2,f1);
mem_array[9] <= opPSE(16'hffff);
mem_array[10] <= 32'b00000000000000000000000000000100;
mem_array[11] <= 32'b00000000000000000000000000001101;
mem_array[12] <= 32'b00000000000000000000000000000111;
mem_array[13] <= 32'b01000000100010000000000000000000;
         for (integer i = 14; i <= size - 1; i = i + 1)
             begin
             mem_array[i] <= 32'h0;
             end
         end
         else if (~CE && ~WE && A[15:8]==8'b00000000) begin
             mem_array[A[7:0]][31:0] <= I_O[31:0];
         end

    end

    assign I_O = I_O_wire;

endmodule
```

Both method is viable (at least to simulate the machine itself) and provided in two versions of the assembler.

Algorithms and techniques

1. Look-up tables

   Since there are many binary strings that are otherwise hard to implement, several look-up tables are initialized at the beginning as static arrays. This way mapping obvious keys to hideous binary counterparts made easy. For example, a static array regi is declared so that function parser is easier to implement. Register $i0 is just regi[0], where the array is as follows:

   ```
   const char regi[3][6] = {"00000", "00001", "00010"}; //as in .mif
   const char regi[3][3] = {"i0", "i1", "i2"}; //as in .sv
   ```

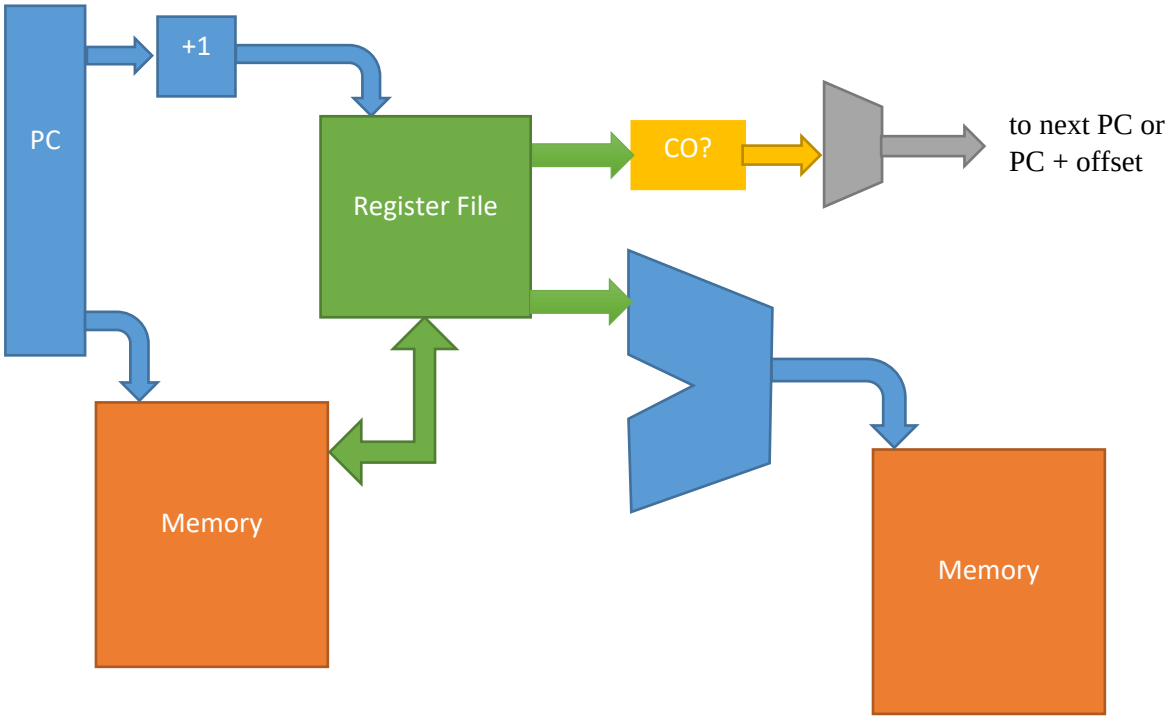
2. Symbol table

   Since symbol table is only responsible for instruction ld, a (int, int) hash table can be generated so that each index is the line number of the function, where table[index] is the offset value. An sample symbol table is as follows:

0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	0	4	5	0	0

Modules (SystemVerilog)

Overall layout



Integer arithmetic unit

1. Integer multiplier

   Inputs: clk, reset, execute, [15:0] val1, val2,

   Outputs: [31:0] out, Ready

   Time complexity: 33 cycles.

   Purpose: this module takes 2 16-bits inputs of 2's complement binary numbers and multiply them together.

   Description: We have done the multiplier module is our lab5, and during that module the method of shifting bits and add numbers is used. In this module, we apply the same algorithm but in a more efficient way of implementation. Though we still use the state machine, but we does not use it to control the adding and shifting process. We implement our integer multiplier module by always_ff signal and we use some temporary data inside the module and let the module do calculation itself during each cycle. And what the state machine does is only to sent the module an execute signal to let it start working and after the module finish calculation, it will send the ready signal to the state machine to let it goes to the next states. That is, our state machine

contains only 2 states for this module, the first one is to load input data in to module's temporary data register, the second state takes 32 clock cycle to compute the value (each cycle can do add/shift for one time and we have totally 16 bits to process which in the worst case needs 16 adds and 16 shifts = 32 cycle).

Algorithm:

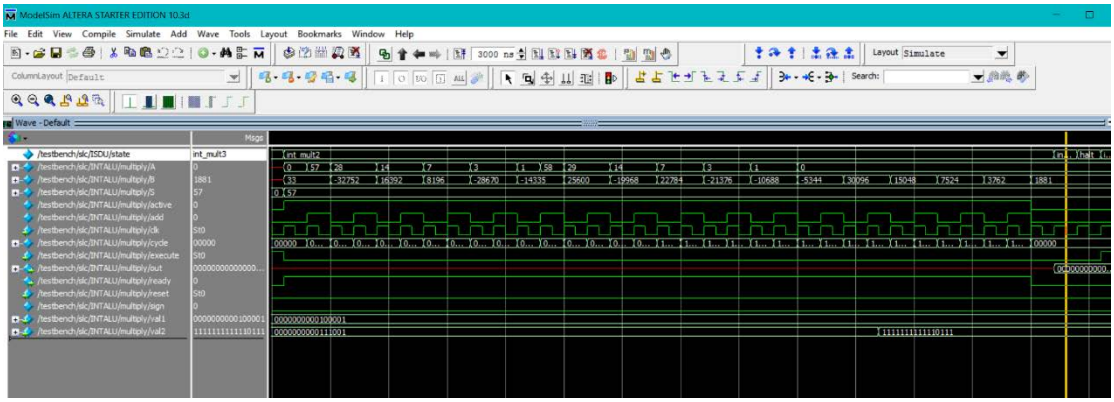
```
if (reset)
  set all the signals to zero.
Else if (execute)
  cycle <= 5'b0;
sign <= val1[15] ^ val2[15];          // select the correct sign
A <= 16'b0;   // initialize value
B <= abs (val1);
S <= abs (val2);
active <= 1'b1;
add <= 1'b0;        // determine next operation is add or shift
Else if (active)
if (add == 1'b0)
  if (B[0] == 1'b1)
    A <= A+S; // add signal
     add <= 1'b1;    // update the add signal
else
  B <= {A[0], B[15:1]};   // shift A and B
  A <= {1'b0, A[15:1]};
if (cycle == 5'b11111) active <= 0; // if complete, halt the module
  add <= 1'b0;      // update add signal
  cycle <= cycle+5'b00001;      // increase counter
else // after calculation, output the value according to the sign
  if (sign == 1'b0) out <= {A, B};
  else out <= ~{A, B}+32'b1;
```

Example:

Consider 33 * 57 = 1881

For the simulation below, we can see that the integer multiplication takes 3 states, int_div1, int_div2, int_div3. The input value (A, B, S) at the beginning are A = 0, B = 33, C = 57. Then at the end of the int_div3, the output which is the combination of A and B. A is still 0, B changes to 1881, so the final result is 1881 which matches the result.

Simulation:



2. Arithmetic and logical unit

   Inputs: clk, reset, INT_EXE, [31:0] A, B, [3:0] INTALUMUX, EXE, [15:0] offset,

   Outputs: INTMULT_READY, INTDIV_READY, [31:0] out1, out2

   Purpose: this module can do all the integer calculation needed for our lc3++ program including add, subtract, multiply, divide, and, not, right shift, left shift or compare.

   Description: In this module, for the simple operation like not, and, we just use the operator provided. However, for other complex operation like multiply and divide, there are some specific modules to do these stuff and included in this INTALU module. And finally, a big mux is used to select the signal needed for out1. Out2 is only used by the divider and the quotient is connected directly to the out2.

3. Floating point unit

   Inputs: clk, reset, [31:0] A, B, [2:0] FPALUMUX,

   Output: zero, [31:0] out

   Purpose: This module does basic calculation for floating point numbers include add, subtract, multiply and divide.

Description: This module includes all the calculation modules for floating point and use a mux to select the signal needed.

4. Integer divisor

Inputs: clk, reset, execute, dividend [31:0], divisor [31:0].

Outputs: quotient [31:0], remainder [31:0], ready.

Time complexity: 65 cycles

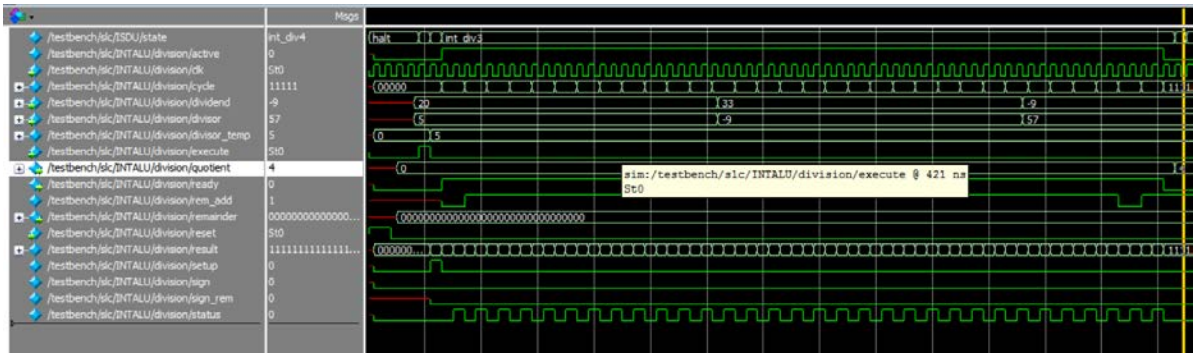
Purpose: This module divides two 2's complement number s and returns a 32-bit quotient and 32-bit remainder. Ready signal is fed into state machine.

Description: This module takes a 32-bit dividend and a 32-bit divisor. First determine the sign of remainder (which has the same sign as divisor), and the sign of quotient. Then Subtract dividend from the shifted divisor until 32 rounds are reached.

Algorithm:

```
procedure int_div(dividend, divisor)
  quotient = 0;
  cycle = 32;
  while (cycle != 0) {
      divisor = divisor >> 1; //00
      if (divisor >= dividend) //01
          quotient = {quotient[30, 0], 0}; //10
      else {
          dividend = dividend - divisor;
          quotient = {quotient[30, 0], 1};
      }
      cycle--;
  }
end procedure
```

Simulation:



Floating point unit

1. Floating point comparator

   Inputs: clk, reset, execute, [31:0] A, [31:0] B.

   Outputs: [1:0] out, read.

   Time complexity: 6 cycles.

   Purpose: This module compares two IEEE-754 floating numbers and returns a 2-bit comparison result. If A = B, out := 00; if A < B, out := 11; if A > B, out := 01.

   Description: The module takes two arguments A and B, both 32-bit wide. It first checks sign bit and determine (partially) what the output would be. Note that “1s” (aka. hidden bit) are appended in front of the two fraction parts. Then it normalizes the exponents and align the fraction part (by shifting the fraction part with the smaller exponent right by the offset). After that, it compares the fraction part and together with the sign bits corresponding to each number and outputs the 2-bit signal.

   Algorithm:

```
procedure float_cmp(A, B)
  exp_diff := |A_exp – B_exp|;
  A_frac := {1, A_frac};
  B_frac := {1, B_frac};
  if (A_exp < B_exp)
    A_frac >> exp_diff;
  else B_frac >> exp_diff;
  if (A_frac = B_frac) return 00;
  if (A_frac > B_frac) return sign? 01 : 11;
  else return sign? 11:01;
```

```
end procedure
```

Example: Consider two numbers -1.75 and -0.25.

1 01111111 (1)10000000000000000000000 = -1.75

0

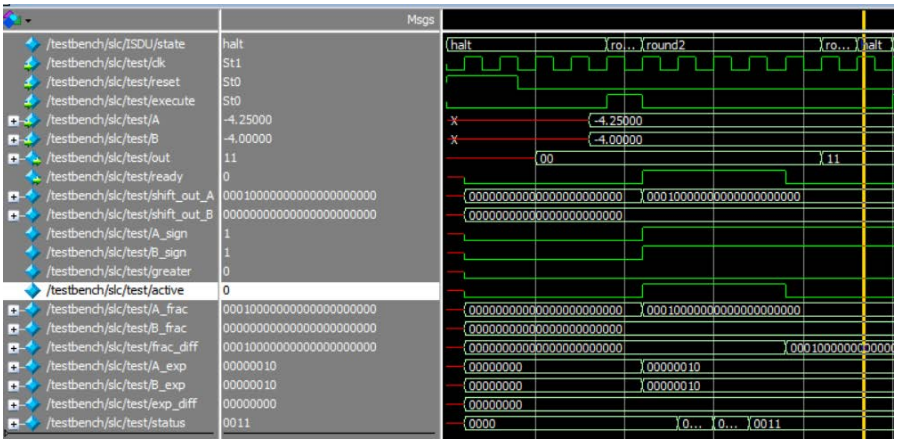
1 01111101 (1)00000000000000000000000 = -0.25

-2

00100000000000000000000 (after shift)

Since fraction part of -1.75 is greater than that of -0.25, but sign is flipped, -1.75 is smaller than -0.25.

Simulation:



2. Floating point rounder

Inputs: clk, reset, execute, in [31:0].

Outputs: out [31:0], ready.

Time complexity: 3 cycles.

Purpose: This module rounds a IEEE 754 floating point to its nearest integer (in 2's complement form) and rounds toward 0. Output is in 2's complement form.

Description: Since 2's complement can only take care of values (in integers) from -2^{31} to $2^{31}-1$, if the exponent of the input is greater than 30 or smaller than -30, the output is automatically set to infinity. If exponent is smaller than -2, the output is set to 0. After checking for edge cases, fraction (together with its hidden 1) is shifted left or right determined by the sign of exponents,

and so now the fraction part is the unsigned integer. After sign extension this value is send to the output.

Algorithm:

```
procedure round(num)
  sign = num[31];
  exp = num[30:23] - 127;
  if (exp > 31)
    exception = 1;
    num := -sign * infty;
    return;
  if (exp = -1)
    out := -sign * 1;
    return;
  else if (exp <= -2)
    out := 0;
    return;
  reg[54:0] := {31'b0, 1, num[22:0]};
  frac << exp;
  if (reg[22] = 0) out := reg[54:23];
  else out := reg[54:23] + 1;
end procedure
```

Example: Consider 4.25:

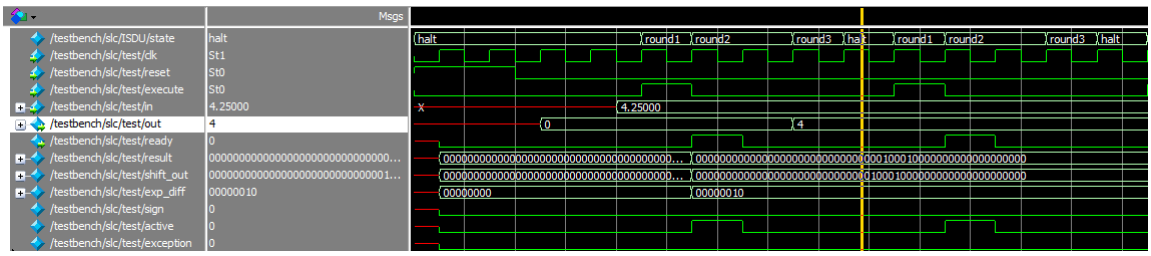
0 10000001 (1)00010000000000000000000 = 4.25

2 100.01 = 4.25

After shift the fraction part becomes:

100 01000000000000000000, which then rounds to b100 = #4.

Simulation:



3. Floating point adder/subtractor

   Inputs: clk, reset, select, execute, A [31:0], B [31:0].

   Outputs: out [31:0], ready.

   Time complexity:

   Purpose: This module adds / subtracts 2 floating point numbers controlled by the select bit (0 is addition and 1 is subtraction). It outputs a 32-bit floating point result and a ready signal to the state machine.

   Description: Similar to the rounding module, the key step in this module is to denormalize and add / subtract the modified fraction part of the two numbers. Even though special cases like infinity and 0 have been taken care of by the exception checker, this module still need to check for overflow, i.e. 4.75 + 4.50 = 9.25, with exponent being incremented by 1. Therefore, the fraction part in this module not only contains a hidden 1, but also takes into account the overflown bit (initialized with a single 0 bit at the front). If this bit becomes 1 after the operation, then exponent is incremented by one. Since subtraction is just adding the same number with sign bit flopped, in the below algorithm only addition is shown.

   Algorithm:

```
procedure add(A, B)
  A_frac := {0, 1, A[22:0]};
  B_frac := {0, 1, B[22:0]};
  exp_diff := |A_exp – B_exp|;
  if (A_exp < B_exp)
    A_frac >> exp_diff;
  else B_frac >> exp_diff;
  frac_diff := A_frac – B_frac;
  if (A > 0 and B > 0)
    result := A_frac + B_frac;
    sign := 0;
  else if (A > 0 and B < 0)
    if (frac_diff < 0)
      result := B_frac – A_frac;
      sign := 1;
```

```
        else
          result := A_frac – B_frac;
          sign := 0;
      else if (A < 0 and B > 0)
        if (frac_diff < 0)
          result := B_frac – A_frac;
          sign := 0;
        else
          result := A_frac – B_frac;
          sign := 1;
      else
        result := A_frac + B_frac;
        sign := 1;
      if (result < 0)
        exp++;
        result := result >> 1;
      out := {sign, exp, result[22:0]};
    end procedure
```

Example: Consider -1.75 + (-0.25) = -2.0:

1 01111111 11000000000000000000000 = -1.75

1 01111101 00000000000000000000000 = -0.25

A_frac: (01)11000000000000000000000

B_frac: (01)00000000000000000000000

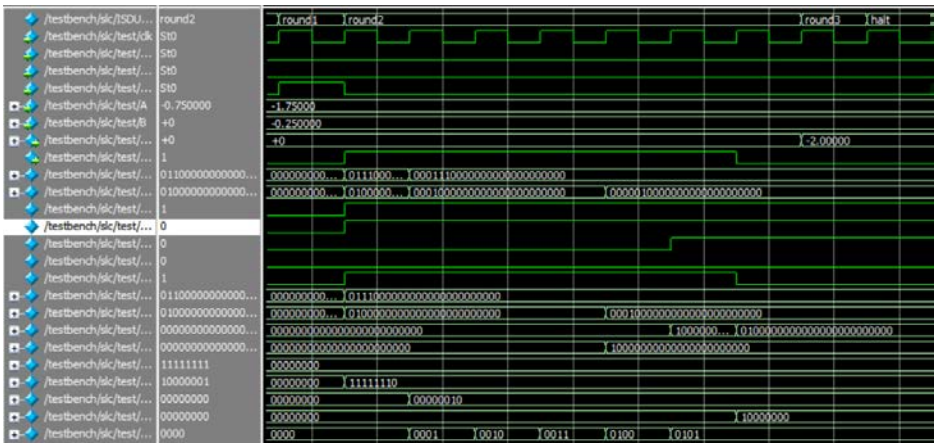
Since A_frac > B_frac, shift B right by 2, which gives:

B_frac: (00)01000000000000000000000

Adding the two parts gives 10000000000000000000000, and since the MSB is one indicating there is an overflow, add 1 to the exponent, which gives:

1 10000000 00000000000000000000000 = -2.

Simulation:



4. Floating point multiplier

   Inputs: [31:0] val1, val2,

   Outputs: [31:0] out

   Time complexity: 1 cycle

   Purpose: this module can do16 bit floating point multiplication and return the result.

   Description: the multiplication of floating point contains 4 main step. First, the sign of the result should be determined. Second, add the 8-bits exponents to get the product's exponent. Third, multiply the mantissa to get the product's mantissa. Fourth, normalize the result. Since for the multiply part, it only needs to multiply the unsigned binary number, standard ' * ' is applied to make calculation easier.

   Algorithm:

```
Out[31] = A[31] ^ B[31]; // XOR inputs' first bit to get the sign
Out[30:23] = EXP(A) + EXP(B) – 127; // add up the exponent
Out[22: 0] = 1.M(A) * 1.M(B); // multiply the mantissa.
normalze (out);
return out;
```

   Example:

   125.34*3.9=488.826

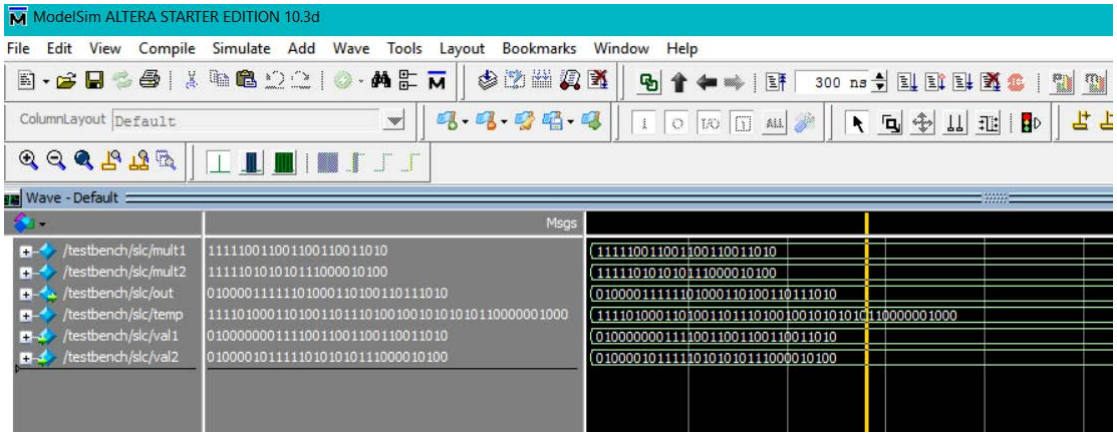
   0 10000000 11110011001100110011010 = 3.9

   0 10000101 11110101010111000010100 = 125.34

   0 10000111 11101000110100110111010 = 488.826

From the simulation result below, we can see that the input mult1, mult2 are 125.34 and 3.9 then the output out is exactly 488.826 which matches the result.

Simulation:



5. Floating point divider

Inputs: [31:0] val1, val2,

Outputs: [31:0] val1, val2

Time complexity: 1 cycle

Purpose: this module takes 2 32 bits of input and divide them. The quotient will be rounded to a floating point value and there is no remainder.

Description: this module is similar to the one of floating point integer and it still takes 4 main part to compute it. First, the signed is determined by the first bits of inputs. Second, the exponential is subtracted to get the real exponent. Third, mantissa is divided as well to get the correct quotient. Fourth, normalize the value.

Unlike the multiply module, the result of dividing two mantissa has some little errors which leads to the final result is not totally the same as the real result. The precision various sometimes but will not larger than 5 % of the real value.

Algorithm:

```
Out[31] = A[31] ^ B[31];  // XOR inputs' first bit to get the sign
Out[30:23] = EXP(A) - EXP(B) + 127;   // subtract the exponent
Out[22: 0] = 1.M(A) / 1.M(B); // leads to some errors
normalize(out);
return out;
```

Example:

488.826 / 3.9 = 125.34

0 10000111 11101000110100110111010 = 488.826

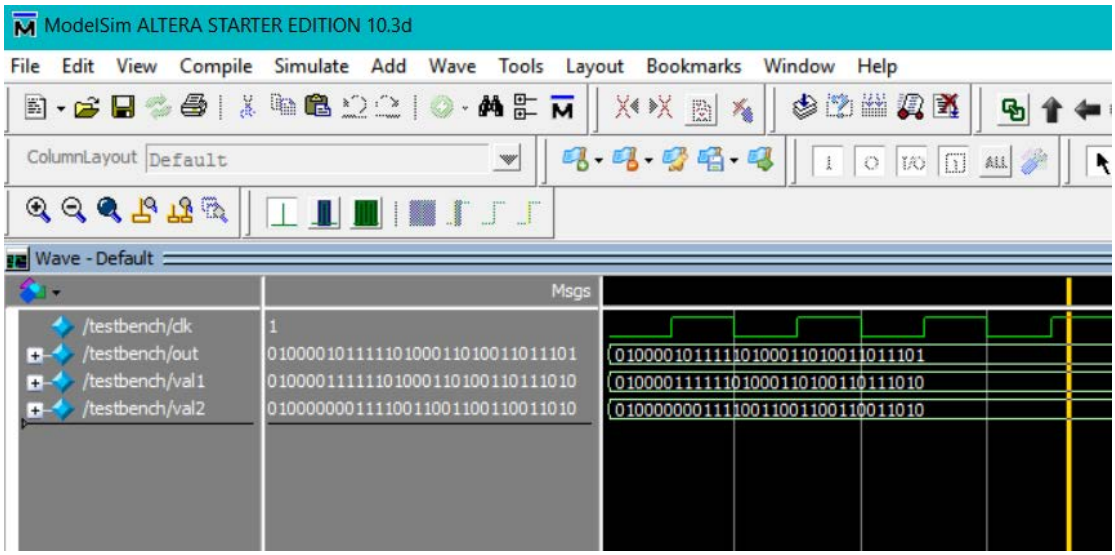
0 10000000 11110011001100110011010 = 3.9

0 10000101 11110101010111000010100 = 125.34

0 10000101 11110100011010011011101 = 125.10325

From the photo below, the input val1, val2 are 488.826 and 3.9, the output value out is 125.10325. This shows that our division of floating point value has some errors but the result is close to the real result.

Simulation:



Data management

1. FP exception detector

   Inputs: [31:0] A, B,

   Outputs: zero, [31:0] add_zero, sub_zero, mult_zero, div_zero

   Purpose: This module is only used for floating point ALU. Since for floating point calculation, 0 is a special value and is hard to calculate, so this special module is used to handle case which the input value is zero.

   Description: In this module, it will process with 2 cases.

   When input A is zero, add result = B, subtract result = -B, multiply result = 0, divide result = 0;

   When input B is zero, add result = A, subtract result = A, multiply result = 0, divide result = $\pm\infty$.

2. Branch taker

   Inputs: clk, reset, N, Z, P, LD_BEN,

   [1:0] C,

   Outputs: BEN.

   Purpose: This module generates the jmp signal, which is used to determine if the if instruction should jump to other address.

   Description: This module sends a signal, jmp, to the controller to determine whether to take the branch or not. Users should first write the compare instruction to compare two numbers. Then the result will be stored in CO (C register). Then if the user executes the if instruction, value from CO will be popped to this module and to compare with the expectation of the compare result. Which means that the compare and if combined together will be a complete logic to compare condition.

   The CO register has these values:

   C = 2'b10 means the compare result before is first number smaller than second number

   C = 2'b01 means the compare result before is second number smaller than the first number.

   C = 2'b00 means the compare result before is first number equal to second number.

   C value is generated by the compare instruction from the ALU.

3. Register file

   Inputs: clk, reset, LD_REG, [4:0] DR, SR1, SR2, [31:0] data_in, data_in2,

   Outputs: [31:0] sr1_out, sr2_out

   Purpose: this is the register in our lc3++ system. When the system is processing some data, the inputs and outputs will be pick up from and save back to these registers.

   Description: In our system, we have designed several registers.

   1 zero register which always holds value 0

   1 sentinel register which always hold value 8'h80000000. It is used for string implementation, which needs a sentinel node at the end of the consecutive memory block so that the PC knows when to stop reading.

   3 floating point registers which provide data for floating point calculation.

   3 temporary registers for users to save temporary data and each one can hold a 32-bit value.

   1 low register and 1 high register for division, multiplication and floating point rounding. After division, the quotient and remainder will be send to these two registers.

Algorithm: To implement this module, I first declare all the registers and use an always comb signal together with the input SR, DR value to control the data in and out through registers.

4. Global bus

   Inputs: [31:0] pc, mdr, adder, alu1, alu2, g_pc, g_mdr, g_adder, g_alu,

   Outputs: [31:0] g_bus1, g_bus2

   Purpose: This global bus is used to transfer 32-bit data between each section of the system.

   Description: this module takes the value from PC, MDR, adder and ALU and transfer this data to mar, register file, memory and other parts which needs this data. In most of the cases, the data only pass through the g_bus1, and g_bus2 is specially used for the division. During division, the g_bus1 will hold the quotient and g_bus2 will hold the reminder.

5. Sign extention

   Inputs: [15:0] in,

   Outputs: [31:0] out

   Purpose: Tthis module extends input value to 32 bit (if they are not already 32-bit wide).

6. Reg_32

   Inputs: clk, reset, load, [31:0] data_in,

   Outputs: [31:0] data_out

   Purpose: A standard 32 bit register.

7. Muxes

   Inputs: [31:0] inA, inB, inC … Control,

   Outputs: [31:0] out

   Purpose:Sstandard mux with different number of inputs and selection bit(s).

8. Ram.v (Memory file)

   Inputs: [7:0] address,

   Clock,

[31:0] data,

Wren,

Outputs: [31:0] q

Purpose: This module is a wizard-generated file. It is our lc3++ system's memory which store the instructions and the temporary data.

Description: This module can make connection between c program and system Verilog files. Users will first write assembling language and then the compiler will generate a .mif file. System Verilog can take this .mif file and generate this memory file. This file is truly the memory which stores the instructions of our assembling language. The lc3++ will begin process at the first line and then line by line until the end of the program.

9. Controller / state machine

Inputs: clk, reset, run, intmult_ready, intdiv_ready, fpas_ready, cmpf_ready, rnd_ready, jmp, [5:0] opcode, func

Outputs: exe, [3:0] INTALUMUX, [2:0] FPALUMUX,[1:0] PCMUX, ADDR2MUX, SR2MUX, ADDR1MUX, ALUMUX, GATE_PC, GATE_MDR, GATE_ADDER, GATE_ALU, LD_IR, LD_PC, LD_REG, LD_MAR, LD_MDR, LD_DISR, MIO_EN, WR_EN, CE, OE, WE

Purpose: This is the central finite state machine of our program. It controls the overall running of the program.

Description: "Three-always" structure is used to implement the finite state machine. The first always_ff is to let the program process one state during a clock cycle. The second always_comb declares the relationship between each states, and build the link between states. The third always declares during each state, which signal should be send to control the gates, mux and other stuffs in the system. The ISDU also has some particular signal which is used by the ALU. The input signal ready and output exe and zero signal is the exchange signal between ISDU and ALU which can make ALU run multiply clock cycles of time during one state in ISDU. This can make ALU has enough time to do complicated operations such like multiplication and division which takes more than 30 clock cycles.

Algorithm:

```
Always_ff     state <= next_state
Always_comb   state to go
Alyways_comb    signals inside this state
```

Top level

Inputs: clk, reset_in, run_in, Continue_in,

Outputs: VGA_CLK, VGA_SYNC_N, VGA_BLANK_N, VGA_VS, VGA_HS,
[7:0] VGA_R, VGA_G, VGA_B

Purpose: this is the top level of the program which can connect all the modules together to build the real program structure.

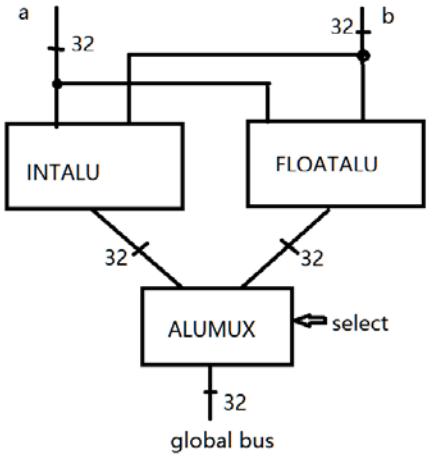
Description: This module contains several modules to build the basic program structure:

1. ISDU: this module controls the running of the program.
2. INTALU/FPALU: This module does most of the calculations of the program. It can calculate both integers and floating points.
3. Global bus: This module transfer data between modules.
4. Memory: The program's system memory. It stores the instructions and values saved by the program.
5. Registers:
   IR register: store the current instruction
   PC register: store the processing address, it can indicate which line is the program currently processing.
   MAR register: store the data address
   MDR register: store the value from memory or the global bus
   DISR register: store the value used to display on VGA.
6. Mux
   PC mux: select signals for PC
   SR2 mux: select signal for the second value of ALU
   Adder1 mux: select signal for the address adder
   Adder2 mux: select signal for the address adder, together with adder1 can implement the ld, st and other instructions which needs to jump some address in the memory.
   MDR mux: select signal for the MDR
7. Sign extend: extend some signals to 32 bits or other bits which the system needed.
8. VGA modules: There are 2 modules used for VGA. The VGA controller is used to control the writing signal to VGA and the blitter module can take a 32 bit signal and let the VGA controller to display it on the screen.

ALU structure

In our project, the most difficult part is the ALU since we not only implement the integer multiplication and division but also applied the floating point add, subtract, multiplication and division. So this makes our ALU's structure more complicated.

Our ALU is divided to two main parts. The first part is the INTALU which can handle integer calculation including not only basic decimal operation but also some binary operation including left and right shift, and, not. The second part is the FPALU which contains the basic operation of the floating points. The two number we want to calculated will be directly pass into these two ALUs and two signals from the ISDU will be send to ALUs to tell them which operation should



they do. Then the result of two ALUs will be send to an ALU select mux. This mux is controlled by the ISDU as well, selecting the signals between INTALU and FPALU.

Inside INTALU, there is just some modules for calculation and a mux to select outputs. However, in FPALU there is a special module called zero detect. This module is used to handle the case when one of the input A or B is zero. During this module, it will calculate the output value of 4 operations according to the case of A or B equals to zero and send the 4 signals out. Also, it will send a flag signal zero which indicate that if the module detected the zero from input A and B. In FPALU, according to the zero signal, it will select weather to use the result from operation module or the result from the zero detection module. What's more, the zero signal will be send to the ISDU, as soon as it detected the signal, it will take action of it that is, instead of waiting more than 30 cycle for ALU's operation module to calculate, it will go directly to the next state.

Standard output

In order to communicate with users, every language has a means of standard input and output which can read input from user with a prompt and can output the data to the console. Since LC-3++ only deals with integers and floating point numbers, and all information must be known at compile time, it is impossible to read from standard input (keyboard). In correspondent with the out $sr instruction, which outputs the value of a given register to the VGA monitor, several extra modules are needed. font_rom.sv specifies the sprites of all ASCII characters, one at a time. blitter.sv specifies the places where the font shows on the 640*480 VGA panel, takes care of refreshing the screen, and outputs the RGB signal to VGA_controller.sv. Each letter has its ROM and their size are fixed. The width of the letter is 8 and height of the letter is 16. Through particular algorithm, we can determine that draw x and draw y is inside which of the letter. Because of this, we can extract particular letter's address and its data.VGA_controller.sv is a clocked module that outputs the corresponding sprites to the monitor. `out $sr` often comes before pause so that the output signal can stay on the screen for a while before the continue button on the FPGA board is pressed. Below is an example: ($f0 initially contains #4.25)

```
out $f0

pause
```

And the monitor will show:

OUT

0x40880000