CSE3017 PROJECT – CPU

1 The Design

The design for the CPU we have come with includes the modules:

- a 16-bit ALU which has 16 operations,
- a Register File consisting of 16 registers (Just like the Block RAM we did on lab 5),
- a 16-bit Comparator,
- a 12-bit Program Counter,
- Status/Comparison registers,
- Multiplexers,
- Demultiplexers

ALU operations:

- ADD, (addition)
- SUB, (subtraction)
- NEG, (negation)
- INC, (increment)
- DEC, (decrement)
- MOV, (move to register)
- AND, (logical and)
- OR, (logical or)
- XOR, (logical xor)
- NOT, (logical not)
- ASR, (arithmetic shift right)
- ASL, (arithmetic shift left)
- LSR, (logical shift right)
- LSL, (logical shift left)
- CSR, (circular shift right)
- CSL, (circular shift left)

Result of ALU is connected to Register File as feedback from the operation done.

This creates the possibility for doing operations on read registers and written back to them.

The Register File works simple:

If we look at all the assembly instructions there is at most 3 registers. Lets say these registers are called “Rz, Rx, Ry” in order and lets say for example instruction is “ADD”. So we have received a instruction:

ADD Rz, Rx, Ry

This will be interpreted on Register File after CPU decodes the instruction. Register file gets z, x and y as inputs so the module can decide which register to read: Rx and Ry. Also which register to read the result into: Rz.

Also we already know that operations write only some registers and some operations, outside of ALU operations, do not even write registers for example:

STR (store to RAM/BUS), NOP (No Operation)

So we should also give register file an input so it can decide whether to write or not. This is done by the input: regWR (register write)

We also need clock since we deal with registers. And we need reset signal so we can clean registers if CPU is reseted via outside input or the operation: RES.

We also have an enable input as CPU can be disabled if enable input is not given to CPU.

Finally z, x, y inputs will be assigned to needed decoded registere indexes. For example ADD:

ADD Rz, Rx, Ry

But for STR:

STR Rx, Ry

or

STR Rx, (Immediate Value)

All operations have specific syntaxes and the design will be done according to their own syntax by using multiplexers at inputs of Register File deciding the registers to read/write.

We can decide the Rx and Ry read registers by 16x16 multiplexers (16 bit data 16 bit select bits) and we can decide which register to read into by 1x16 (1 bit data 16 bit select bits). The choosen read register is enabled so that clock's are effective and data is written into others remain same.

Finally we have to decide which data to read into selected read register. There is 3 different inputs according to instruction.

For example for immediate operations:

MOV R1, 10

We have to somehow pull the "10" immediate value from the decoded instruction and put to Rz input of Register File.

For example for normal operations:

MOV R1, R2

We already discussed this type of operations result can be pulled from the ALU output and read into register file.

For example for load operation:

LD R1, 0x20

The value at address 0x20 should be loaded from the RAM/BUS.

This all can be done by two 16x16 multiplexers by selecting the input according to if the instruction is immediate or not and if the instruction is a load instruction or not.

Next we have to design a 16-bit Comparator so we can actually execute comparison (CMP) and conditional jump/branching instructions.

In our design we included a CMP instruction and 16 different conditional jumps. These jumps are:

- JG, (jump greater)
- JGE, (jump greater than or equal)
- JE, (jump equal)
- JNE, (jump not equal)
- JLE, (jump less than or equal)
- JL, (jump less than)
- JC, (jump if carry, carry ALU flag)
- JZ, (jump if zero, zero ALU flag)
- JN, (jump if negative, negative ALU flag)
- JO, (jump if overflow, overflow ALU flag)
- JP, (jump if odd parity, odd parity ALU flag)
- JNC, (jump if not carry)
- JNZ, (jump if not zero)
- JNN, (jump if not negative, positive)
- JNO, (jump if not overflow)
- JNP (jump if not odd parity, even parity)

So we need 2 register to store these conditions so when a jump instruction comes we decide upon whether to jump or not by reading these registers.

First register is called CPSR (Current Program Status Register):

This register holds status flags from the ALU if the alu is enabled otherwise should remain the same so the conditions won’t be overwritten in other non-ALU instructions such as CMP.

Second register is called Status register:

This register holds comparison results and only enabled when the CMP instruction is being executed.

Finally we can decide upon whether to jump or not according to corresponding condition by decoding the condition from the instruction bits. This can be done using a 16x1 multiplexer with the CPSR and Status register outputs.

We can give these conditions as a signal to program counter to signify a incoming jump and a jump address so the program counter can load the address to jump. Program counter is a simple 12 bit counter unless a jump is received then the new address is loaded. Program counter output is connected to ROM so we fetch needed instruction one by one sequentially each clock.

Finally we have to have a RAM load/store unit. At early designs we implemented a single port RAM type structure because we were designing a SDRAM controller which we have a SDRAM on board. We designed a SDRAM module but it did not work at all and we didn’t even got any output as result (0’s always). We tried analysing the signals using the Quartus program with SignalTap logic analyser but we weren’t able to understand it either so we changed the idea to design a Block RAM. Only thing changed is now we have much less memory but it is still enough.

So we have 2 different designs for the CPU first we are gonna show the codes for the early designs with SDRAM in mind. Next we will be showing last designs we did.

Now this out of the way we should add a table for the instruction set structure:

Command Table

ALU Instruction	B1, B0	B3, B2	B5, B4	B9…B6	B13...B10	B17...B14	B21...B18	B25...B22	B29…B26	B30
ADD Rz, Rx, Ry	00	00	00	z	x	y	XXXX	XXXX	XXXX	1
SUB Rz, Rx, Ry	00	01	00	z	x	y	XXXX	XXXX	XXXX	1
NEG Rz, Rx, Ry	00	10	00	z	x	y	XXXX	XXXX	XXXX	1
INC Rz, Rx	00	11	00	z	x	XXXX	XXXX	XXXX	XXXX	1
DEC Rz, Rx	00	00	01	z	x	XXXX	XXXX	XXXX	XXXX	1
AND Rz, Rx, Ry	00	10	01	z	x	y	XXXX	XXXX	XXXX	1
OR Rz, Rx, Ry	00	11	01	z	x	y	XXXX	XXXX	XXXX	1
XOR Rz, Rx, Ry	00	00	10	z	x	y	XXXX	XXXX	XXXX	1
NOT Rz, Rx	00	01	10	z	x	XXXX	XXXX	XXXX	XXXX	1
ASR Rz, Rx, Ry	00	10	10	z	x	y	XXXX	XXXX	XXXX	1
ASL Rz, Rx, Ry	00	11	10	z	x	y	XXXX	XXXX	XXXX	1
LSR Rz, Rx, Ry	00	00	11	z	x	y	XXXX	XXXX	XXXX	1

LSL Rz, Rx, Ry	00	01	11	z	x	y	XXXX	XXXX	XXXX	1
CSR Rz, Rx, Ry	00	10	11	z	x	y	XXXX	XXXX	XXXX	1
CSL Rz, Rx, Ry	00	11	11	z	x	y	XXXX	XXXX	XXXX	1

MOV Instruction	**B1, B0**	**B3, B2**	**B5, B4**	**B9…B6**	**B13...B10**	**B17...B14**	**B21...B18**	**B25...B22**	**B29…B26**	**B30**
MOV Rz, Rx	00	01	01	z	x	XXXX	XXXX	XXXX	XXXX	1
MOV Rz, #Imm	00	01	01	z	16-Bit Immediate Value				XXXX	1

LD/STR Ins.	**B1, B0**	**B3, B2**	**B5, B4**	**B9…B6**	**B13...B10**	**B17...B14**	**B21...B18**	**B25...B22**	**B29…B26**	**B30**
LD Rz, Ry	01	XX	XX	z	XXXX	y	XXXX	XXXX	XXXX	1
STR Rx, Ry	10	XX	XX	XXXX	x	y	XXXX	XXXX	XXXX	0

Immediates

LD/STR Ins.	**B1, B0**	**B3, B2**	**B5, B4**	**B9…B6**	**B13...B10**	**B17...B14**	**B21...B18**	**B25...B22**	**B29…B26**	**B30**
LD Rz, #Imm	01	XX	XX	z	12-Bit Address			XXXX	XXXX	1
STR Rx, #Imm	10	XX	XX	x	12-Bit Address			XXXX	XXXX	0

Other Ins.	B1, B0	B3, B2	B5, B4	B9…B6	B13...B10	B17...B14	B21...B18	B25...B22	B29…B26	B30
NOP	11	11	XX	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	0
CMP Rx, Ry	11	01	XX	XXXX	x	y	XXXX	XXXX	XXXX	0
RES	11	00	XX	XXXX	XXXX	XXXX	XXXX	XXXX	XXXX	0

JUMP Instruction	B1, B0	B3, B2	B5, B4	B9…B6	B13...B10	B17...B14	B21...B18	B25...B22	B29…B26	B30
JG Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	0000	0
JGE Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	0001	0
JE Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	0010	0
JNE Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	0011	0
JLE Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	0100	0
JL Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	0101	0
JC Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	0110	0
JZ Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	0111	0
JN Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	1000	0
JO Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	1001	0
JP Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	1010	0

JNC Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	1011	0
JNZ Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	1100	0
JNN Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	1101	0
JNO Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	1110	0
JNP Ry	11	10	XX	XXXX	XXXX	y	XXXX	XXXX	1111	0

Immediates

JUMP Instruction	**B1, B0**	**B3, B2**	**B5, B4**	**B9…B6**	**B13...B10**	**B17...B14**	**B21...B18**	**B25...B22**	**B29…B26**	**B30**
JG #Imm	11	10	XX	XXXX		12-Bit Address		XXXX	0000	0
JGE #Imm	11	10	XX	XXXX		12-Bit Address		XXXX	0001	0
JE #Imm	11	10	XX	XXXX		12-Bit Address		XXXX	0010	0
JNE #Imm	11	10	XX	XXXX		12-Bit Address		XXXX	0011	0
JLE #Imm	11	10	XX	XXXX		12-Bit Address		XXXX	0100	0
JL #Imm	11	10	XX	XXXX		12-Bit Address		XXXX	0101	0
JC #Imm	11	10	XX	XXXX		12-Bit Address		XXXX	0110	0
JZ #Imm	11	10	XX	XXXX		12-Bit Address		XXXX	0111	0
JN #Imm	11	10	XX	XXXX		12-Bit Address		XXXX	1000	0

JO #Imm	11	10	XX	XXXX	12-Bit Address	XXXX	1001	0
JP #Imm	11	10	XX	XXXX	12-Bit Address	XXXX	1010	0
JNC #Imm	11	10	XX	XXXX	12-Bit Address	XXXX	1011	0
JNZ #Imm	11	10	XX	XXXX	12-Bit Address	XXXX	1100	0
JNN #Imm	11	10	XX	XXXX	12-Bit Address	XXXX	1101	0
JNO #Imm	11	10	XX	XXXX	12-Bit Address	XXXX	1110	0
JNP #Imm	11	10	XX	XXXX	12-Bit Address	XXXX	1111	0

In our implementation we decided to change don't care values with zeros.

And we designed a assembler so we can directly turn these assembly codes for instruction into machine code for our CPU. The java code for our assembler:

```
package Assembler;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Scanner;

import java.io.File;
import java.io.FileWriter;

public class Assembler {

private static ArrayList<String> instructions = new ArrayList<String>(Arrays.asList(new
String[]{
// ALU Operations
"ADD", "SUB", "NEG", "INC",
"DEC", "AND", "OR", "XOR" ,
"NOT", "ASR", "ASL", "LSR",
"LSL", "CSR", "CSL",
// MOV Operation
"MOV",
// RAM Operations
"LD", "STR",

// Other Instructions
"NOP", "CMP", "RES",
```

```
// JUMP Instructions (Comparator)
"JG", "JGE", "JE", "JNE",
"JLE", "JL",
// Jump Instructions (CPSR)
"JC", "JZ", "JN", "JO",
"JP", "JNC", "JNZ", "JNN",
"JNO", "JNP"
}));

private static ArrayList<String> codes = new ArrayList<String>(Arrays.asList(new String[]
{
// ALU Operations
"000000", "000100", "001000", "001100",
"010000", "011000", "011100", "100000" ,
"100100", "101000", "101100", "110000",
"110100", "111000", "111100",
// MOV Operation
"010100",
// RAM Operations
"000001", "000010",

// Other Instructions
"001111", "000111", "000011",

// JUMP Instructions (Comparator)
"001011", "001011", "001011", "001011",
"001011", "001011",
// Jump Instructions (CPSR)
"001011", "001011", "001011", "001011",
"001011", "001011", "001011", "001011",
"001011", "001011"
}));

public static void main(String[] args) {
StringBuilder executable = new StringBuilder();
try {
File inputFile = new File(args[0]);
if (inputFile.exists()) {
Scanner sc = new Scanner(inputFile);
int lineNumber = 1;
while(sc.hasNextLine()) {
String line = sc.nextLine();
line = line.toUpperCase();
if(line.trim().length() != 0){
if(line.trim().charAt(0) != ';') {
String parsed = parseLine(line, lineNumber);

executable.append(toHex(parsed).toUpperCase() + "\n");
lineNumber++;
}
```

```
}
}
sc.close();
} else {
System.out.println("Input file does not exist...");
System.exit(-1);
}
File outputFile = new File(args[1]);
if(!outputFile.exists()) {
System.out.println("Output file does not exist...");
System.out.println("Creating output file: " + args[1]);
outputFile.createNewFile();
}
FileWriter fw = new FileWriter(outputFile, false);
/* Not needed
String fileFormat =
"DEPTH = 4096; -- The size of memory in words\n" +
"WIDTH = 32; -- The size of data in bits\n" +
"ADDRESS_RADIX = HEX; -- The radix for address values\n" +
"DATA_RADIX = HEX; -- The radix for data values\n" +
"CONTENT -- start of (address : data pairs)\n" +
"BEGIN\n" +
"-- memory address : data\n" +
"%s" +
"END;\n";


String lines[] = executable.toString().split("\n");
StringBuilder content = new StringBuilder();

for(int i = 0; i < lines.length; i++) {
content.append("" + toHex(toBinary(i, 12)) + " : " + lines[i] + ";\n");
}

fw.write(String.format(fileFormat, content));
*/

fw.write(executable.toString());

fw.close();
} catch (Exception e) {
e.printStackTrace();
}
}

private static String toHex(String bin) {
String hex = "";

for(int i = bin.length() - 1; i > 0; i-=4) {
int hexdigit = 0;
for(int j = i; j > i-4 && j >= 0; j--) {
```

```
hexdigit += (bin.charAt(j) == '1' ? 1: 0) * Math.pow(2, i-j);
}
hex = Integer.toString(hexdigit, 16) + hex;
}

return hex;
}
private static String toBinary(int number, int length) {
String bin = Integer.toBinaryString((0x1 << (length)) | number).substring(1);
return bin;
}
private static String parseLine(String line, int lineNumber) {
String binaryLine = "";

String words[] = line.replaceAll(",", "").split(" ");

String instruction = words[0];

boolean immediate = false;

for(int i = 1; i < words.length; i++) {
if(words[i].charAt(0) != 'R') {
if(words[i].length() >= 4) {
if(!words[i].substring(0, 3).equals("ACC")) {
immediate = true;
break;
}
} else {
immediate = true;
break;
}
}
}

binaryLine += immediate ? "1": "0"; // Decide for the Immediate bit

int indexOfInstruction = instructions.indexOf(instruction);

if(indexOfInstruction == -1) {
System.err.println("Instruction: \"" + instruction + "\" at line " + lineNumber + " is not
supported.");
System.exit(-1);
}

binaryLine += indexOfInstruction < 17 ? "1" : "0"; // Decide for the RegWr bit

binaryLine += indexOfInstruction > 20 ? toBinary(indexOfInstruction - 21, 4) : "0000";

String operands[] = new String[words.length - 1];
```

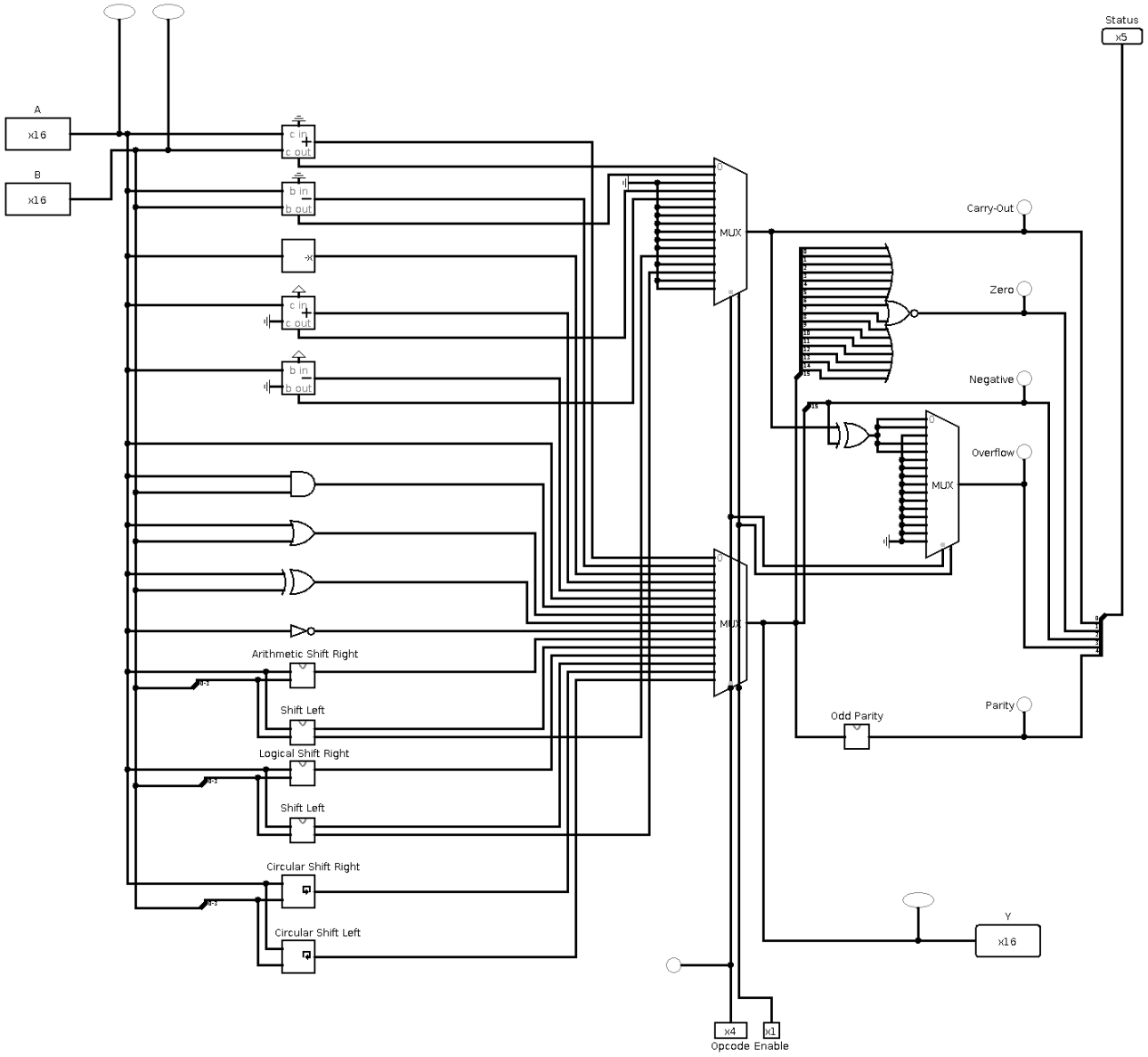
```
for(int i = 1; i < words.length; i++) {
operands[i-1] = words[i];
}

binaryLine += parseOperands(indexOfInstruction, operands, immediate); // Put the bits
related to instruction operands.

binaryLine += codes.get(indexOfInstruction);

return binaryLine;
}

private static String parseOperands(int insIndex, String[] operands, boolean immediate) {
String binary = "";
String instruction = instructions.get(insIndex);

for(int i = 0; i < operands.length; i++) {
if(operands[i].startsWith("R")) {
operands[i] = operands[i].replaceAll("R", "");
} else if (operands[i].startsWith("ACC")) {
operands[i] = Integer.toString(Integer.parseInt(operands[i].replaceAll("ACC", "")) + 12);
} else if(operands[i].startsWith("0X")) {
operands[i] = Integer.toString(Integer.parseInt(operands[i].replaceAll("0X", ""), 16));
} else if(operands[i].startsWith("0B")) {
operands[i] = Integer.toString(Integer.parseInt(operands[i].replaceAll("0B", ""), 2));
}
}

if(immediate) {
if(instruction.equals("LD") || instruction.equals("STR")) {
binary += "0000";

int address = Integer.parseInt(operands[1]);

binary += toBinary(address, 12);

int registerIndex = Integer.parseInt(operands[0]);

binary += toBinary(registerIndex, 4);

} else if(instruction.equals("MOV")){
int value = Integer.parseInt(operands[1]);

binary += toBinary(value, 16);

int registerIndex = Integer.parseInt(operands[0]);

binary += toBinary(registerIndex, 4);
} else {
binary += "0000";
```

```
int value = Integer.parseInt(operands[0]);

binary += toBinary(value, 12);

binary += "0000";
}
} else {
binary += "00000000";
if(instruction.equals("INC") || instruction.equals("DEC") || instruction.equals("NOT") ||
instruction.equals("MOV")) {
binary += "0000";

int xRegisterIndex = Integer.parseInt(operands[1]);

binary += toBinary(xRegisterIndex, 4);

int zRegisterIndex = Integer.parseInt(operands[0]);

binary += toBinary(zRegisterIndex, 4);
} else if(instruction.equals("LD")) {
int yRegisterIndex = Integer.parseInt(operands[1]);

binary += toBinary(yRegisterIndex, 4);
binary += "0000";

int zRegisterIndex = Integer.parseInt(operands[0]);

binary += toBinary(zRegisterIndex, 4);
} else if(instruction.equals("STR") || instruction.equals("CMP")) {
int yRegisterIndex = Integer.parseInt(operands[1]);

binary += toBinary(yRegisterIndex, 4);

int xRegisterIndex = Integer.parseInt(operands[0]);

binary += toBinary(xRegisterIndex, 4);

binary += "0000";
} else if(instruction.equals("NOP") || instruction.equals("RES")) {
binary += "000000000000";
} else if(insIndex < 15) {
int yRegisterIndex = Integer.parseInt(operands[2]);

binary += toBinary(yRegisterIndex, 4);
int xRegisterIndex = Integer.parseInt(operands[1]);

binary += toBinary(xRegisterIndex, 4);

int zRegisterIndex = Integer.parseInt(operands[0]);
```

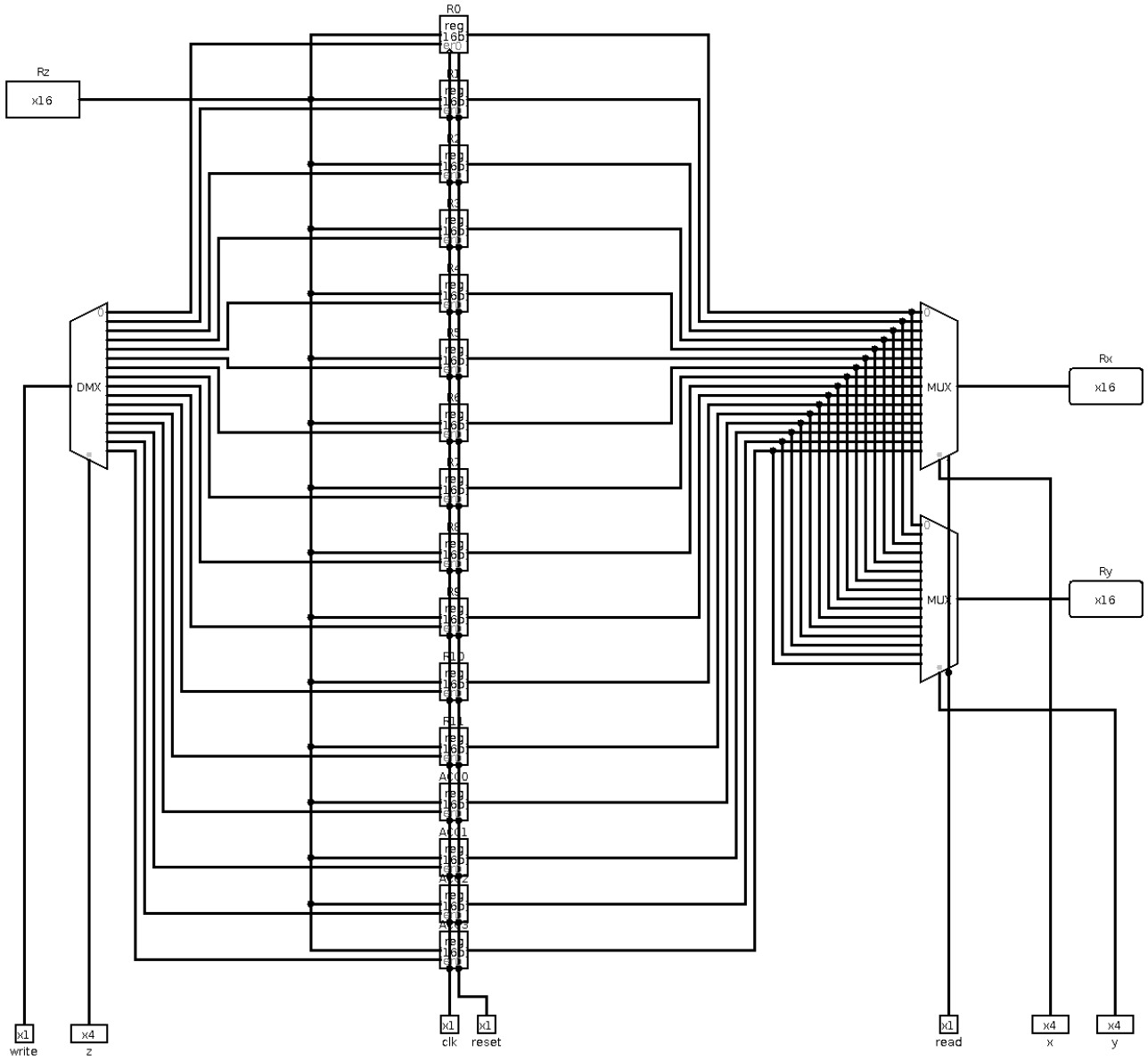
```
binary += toBinary(zRegisterIndex, 4);
} else {
int yRegisterIndex = Integer.parseInt(operands[0]);

binary += toBinary(yRegisterIndex, 4);

binary += "00000000";
}
}

return binary;
}
}
```

Logic circuits for our first designs:

First the ALU:

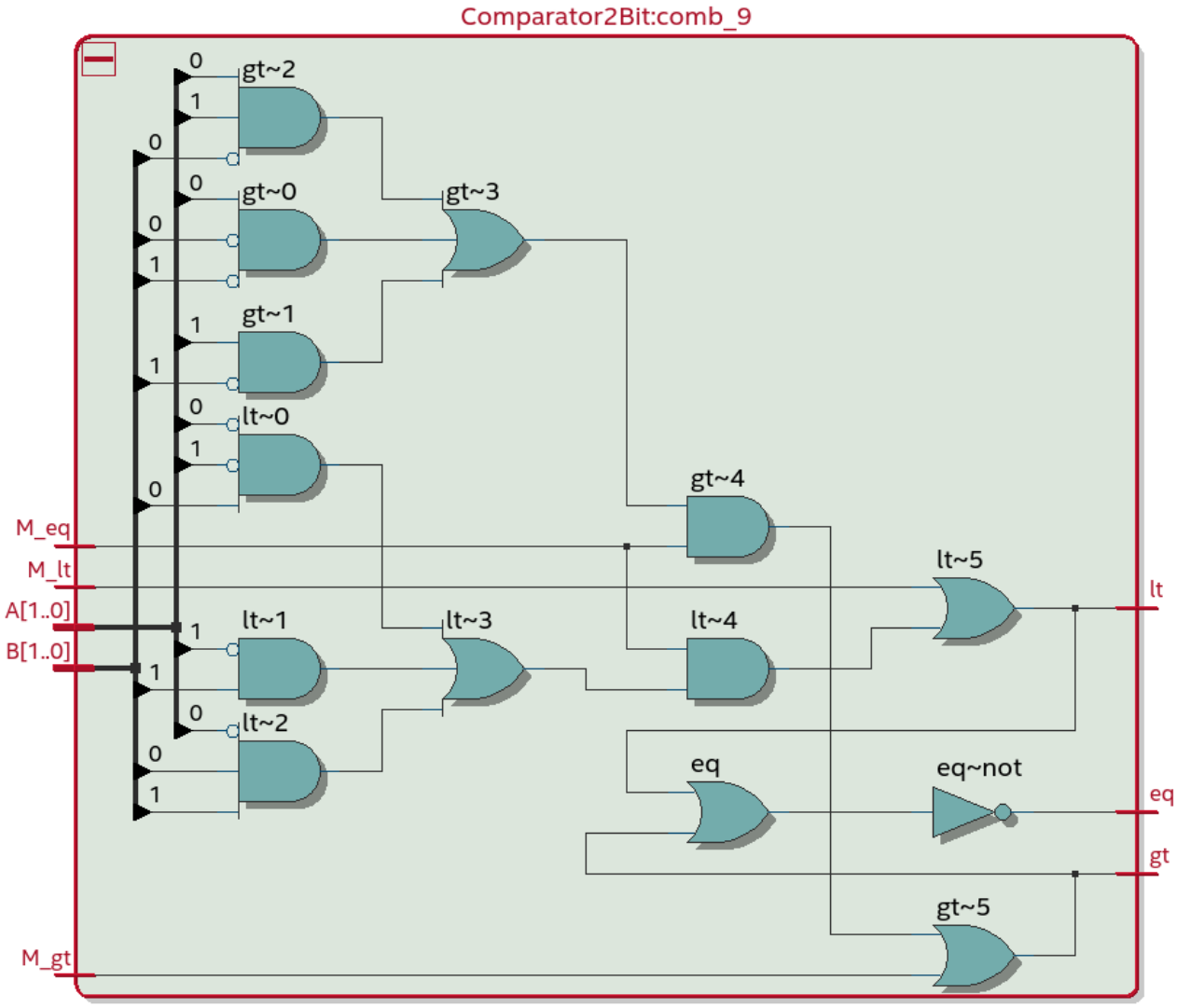


Next Register File:

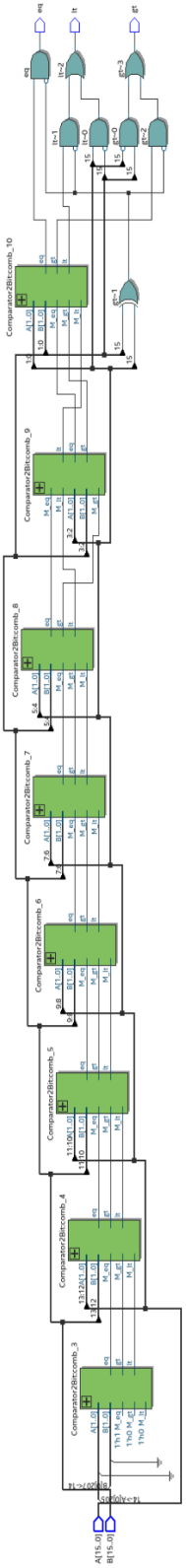


And to create a 16-bit Comparator:

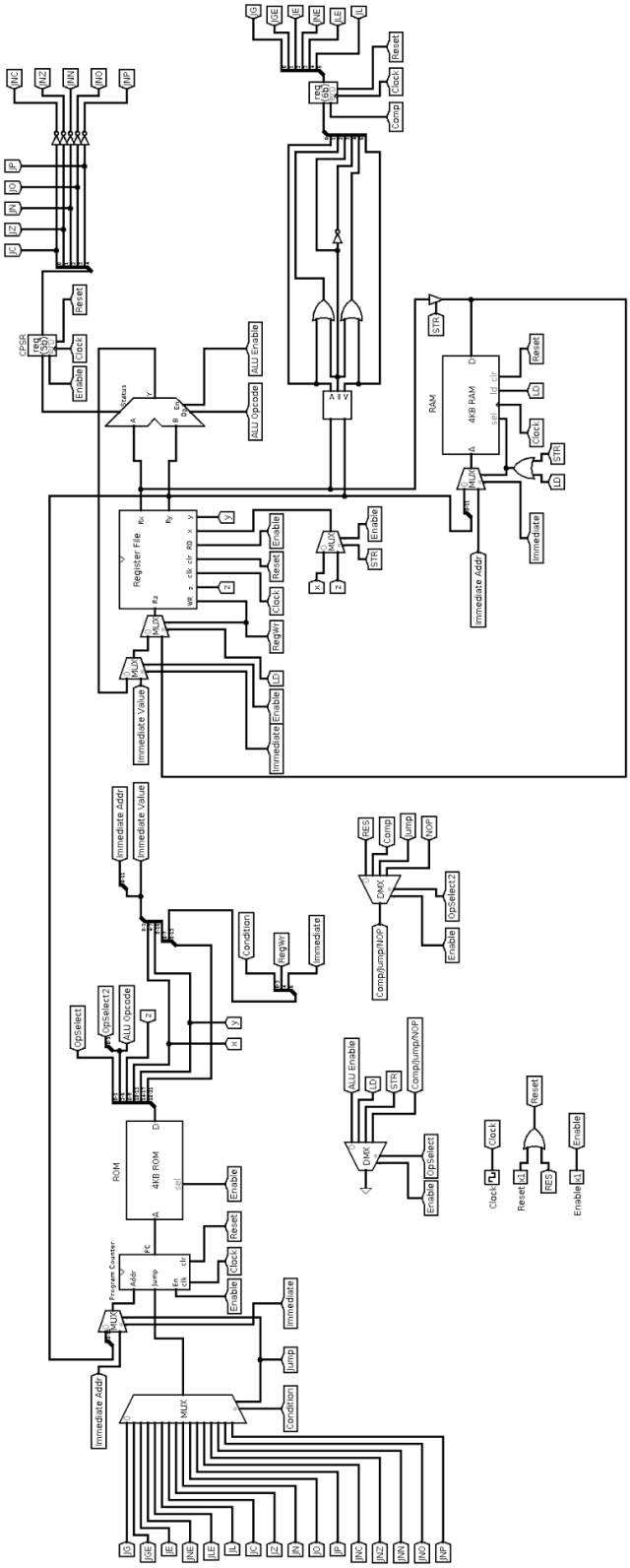
First 2-bit comparator:



16-Bit comparator:



Finally the main design:



2 Codes

Main design first then submodules:

First design for our CPU:

```
module CPU(
input clk,
input res,
input enable,
// Instruction fetch pins (ROM)
output [11:0] instructionAddr,
input [31:0] instruction,
output selInstruction,
// Data load/str pins (RAM)
output [11:0] dataAddr,
input [15:0] dataIn,
output [15:0] dataOut,
output selData,
output ldData,
output clrData
);

// DECODE THE RECEIVED INSTRUCTION:
wire [1:0] opSelect1;
wire [1:0] opSelect2;
wire [3:0] ALUOpcode;
wire [3:0] z;
wire [3:0] x;
wire [3:0] y;
wire [11:0] immediateAddr;
wire [15:0] immediateValue;
wire [3:0] condition;
wire regWr;
wire immediate;
assign opSelect1 = instruction[1:0];
assign opSelect2 = instruction[3:2];
assign ALUOpcode = instruction[5:2];
assign z = instruction[9:6];
assign x = instruction[13:10];
assign y = instruction[17:14];
assign immediateAddr = instruction[21:10];
assign immediateValue = instruction[25:10];
assign condition = instruction[29:26];
assign regWr = instruction[30];
assign immediate = instruction[31];
// Create CPU operation group selectors:
wire ALUEnable;
wire LD;
wire STR;
wire nextGroup;
```

```
DEMUX1x4(
1'b1,
opSelect1,
enable,
{nextGroup, STR, LD, ALUEnable}
);
wire RES;
wire COMP;
wire JUMP;
wire NOP;
DEMUX1x4(
nextGroup,
opSelect2,
enable,
{NOP, JUMP, COMP, RES}
);
// Create inner CPU reset signal:
wire reset;
assign reset = res | RES;
// ALU result will be connected to this wire:
wire [15:0] ALUResult;
// Setting up Register File
wire [15:0] Rz0;
MUX2x16(ALUResult, immediateValue, immediate, enable, Rz0);
wire [15:0] Rz; // Read register value
MUX2x16(Rz0, dataIn, LD, regWr, Rz);
wire [3:0] xSelected;
MUX2x4(x, z, STR, enable, xSelected);
wire [15:0] Rx; // A input for ALU/Comparator
wire [15:0] Ry; // B input for ALU/Comparator
RegisterFile(
regWr,
enable,
~clk,
reset,
z,
xSelected,
y,
Rz,
Rx,
Ry
);
// Setting up ALU
wire [4:0] CPSRnow;
ALU16Bit(
Rx,
Ry,
ALUOpcode,
ALUEnable,
ALUResult,
CPSRnow
```

```
);
// Setting up Comparator:
wire gt;
wire eq;
wire lt;
Comparator16Bit(
Rx,
Ry,
gt,
eq,
lt
);
// Setting up CPSR:
wire [4:0] CPSR;
Register5Bit(
CPSRnow,
~clk,
enable,
reset,
CPSR
);
// Setting up Comparison registers:
wire [5:0] regCOMP;
Register6Bit(
{lt, lt | eq, ~eq, eq, eq | gt, gt},
~clk,
COMP,
reset,
regCOMP
);
// Setting up data (RAM) connections:
assign selData = LD | STR;
MUX2x12(Ry[11:0], immediateAddr, immediate, selData, dataAddr);
assign ldData = LD;
assign clrData = reset;
assign dataOut = Rx;
// Selecting jump conditions:
wire jumpPC;
MUX16x1(
{
~CPSR[4], // JNP
~CPSR[3], // JNO
~CPSR[2], // JNN
~CPSR[1], // JNZ
~CPSR[0], // JNC
CPSR[4], // JP
CPSR[3], // JO
CPSR[2], // JN
CPSR[1], // JZ
CPSR[0], // JC
regCOMP[5], // JL
```

```
regCOMP[4], // JLE
regCOMP[3], // JNE
regCOMP[2], // JE
regCOMP[1], // JGE
regCOMP[0] // JG
},
condition,
JUMP,
jumpPC
);
// Setting up Program Counter:
wire [11:0] addrPC;
MUX2x12(Ry[11:0], immediateAddr, immediate, JUMP, addrPC);
ProgramCounter(
addrPC,
jumpPC,
enable,
clk,
reset,
instructionAddr
);
// Setting up instruction (ROM) connections:
assign selInstruction = enable;

endmodule
```

Second design (Program Counter changed etc.):

```
module CPUv2(
input clk,
input resIn,
input enable,
// Instruction fetch pins (ROM)
output reg [11:0] instructionAddr,
input [31:0] instructionNow,
output selInstruction,
// Data load/str pins (RAM)
output [11:0] dataAddr,
input [15:0] dataIn,
output [15:0] dataOut,
output selData,
output ldData,
output clrData
);

// Fetched Instruction:
reg [31:0] instruction;

// DECODE THE RECEIVED INSTRUCTION:
wire [1:0] opSelect1;
```

```
wire [1:0] opSelect2;
wire [3:0] ALUOpcode;
wire [3:0] z;
wire [3:0] x;
wire [3:0] y;
wire [11:0] immediateAddr;
wire [15:0] immediateValue;
wire [3:0] condition;
wire regWr;
wire immediate;
assign opSelect1 = instruction[1:0];
assign opSelect2 = instruction[3:2];
assign ALUOpcode = instruction[5:2];
assign z = instruction[9:6];
assign x = instruction[13:10];
assign y = instruction[17:14];
assign immediateAddr = instruction[21:10];
assign immediateValue = instruction[25:10];
assign condition = instruction[29:26];
assign regWr = instruction[30];
assign immediate = instruction[31];
// Create CPU operation group selectors:
wire ALUEnable;
wire LD;
wire STR;
wire nextGroup;
DEMUX1x4(
1'b1,
opSelect1,
enable,
{nextGroup, STR, LD, ALUEnable}
);
wire RES;
wire COMP;
wire JUMP;
wire NOP;
DEMUX1x4(
nextGroup,
opSelect2,
enable,
{NOP, JUMP, COMP, RES}
);
// Create inner CPU reset signal:
wire reset;
assign reset = resIn | RES;
// ALU result will be connected to this wire:
wire [15:0] ALUResult;
// Setting up Register File
wire [15:0] Rz0;
MUX2x16(ALUResult, immediateValue, immediate, enable, Rz0);
wire [15:0] Rz; // Read register value
```

```
MUX2x16(Rz0, dataIn, LD, regWr, Rz);
wire [3:0] xSelected;
MUX2x4(x, z, STR, enable, xSelected);
wire [15:0] Rx; // A input for ALU/Comparator
wire [15:0] Ry; // B input for ALU/Comparator
RegisterFile(
regWr,
enable,
~clk,
reset,
z,
xSelected,
y,
Rz,
Rx,
Ry
);
// Setting up ALU
wire [4:0] CPSRnow;
ALU16Bit(
Rx,
Ry,
ALUOpcode,
ALUEnable,
ALUResult,
CPSRnow
);
// Setting up Comparator:
wire gt;
wire eq;
wire lt;
Comparator16Bit(
Rx,
Ry,
gt,
eq,
lt
);
// Setting up data (RAM) connections:
assign selData = LD | STR;
MUX2x12(Ry[11:0], immediateAddr, immediate, selData, dataAddr);
assign ldData = LD;
assign clrData = reset;
assign dataOut = Rx;
// Status/Comparison Registers:
reg [4:0] CPSR;
reg [5:0] regCOMP;
// Selecting jump conditions:
wire jumpPC;
MUX16x1(
{
```

```
~CPSR[4], // JNP
~CPSR[3], // JNO
~CPSR[2], // JNN
~CPSR[1], // JNZ
~CPSR[0], // JNC
CPSR[4], // JP
CPSR[3], // JO
CPSR[2], // JN
CPSR[1], // JZ
CPSR[0], // JC
regCOMP[5], // JL
regCOMP[4], // JLE
regCOMP[3], // JNE
regCOMP[2], // JE
regCOMP[1], // JGE
regCOMP[0] // JG
},
condition,
JUMP,
jumpPC
);
// Setting up Program Counter:
wire [11:0] addrPC;
MUX2x12(Ry[11:0], immediateAddr, immediate, JUMP, addrPC);
/*
ProgramCounter(
addrPC,
jumpPC,
enable,
clk,
reset,
instructionAddr
);
*/
// Setting up instruction (ROM) connections:
assign selInstruction = enable;
// Current Program Status Register
initial begin
CPSR <= 5'b000000;
end
always @(posedge ALUEnable or posedge reset)
begin
if(reset)
begin
CPSR <= 5'b00000;
end
else
begin
CPSR <= enable ? CPSRnow : 5'b00000;
end
end
```

```
// Comparison Registers
initial begin
regCOMP <= 6'b00000;
end
always @(posedge COMP or posedge reset)
begin
if(reset)
begin
regCOMP <= 6'b000000;
end
else
begin
regCOMP <= enable ? {lt, lt | eq, ~eq, eq, eq | gt, gt} : 6'b000000;
end
end
// Program Counter (Instruction Fetching Mechanism)
reg resetFlag;
reg jumpFlag;
initial begin
instruction <= 32'h00000000;
instructionAddr <= 12'h000;
resetFlag <= 1'b0;
jumpFlag <= 1'b0;
end
always @(posedge clk or posedge reset or posedge jumpPC)
begin
instruction <= instructionNow; // Fetch current instruction

if(reset)
begin
instructionAddr <= 12'h000;
resetFlag <= 1'b1;
end
else if(jumpPC)
begin
instructionAddr <= enable ? addrPC : instructionAddr;
jumpFlag <= 1'b1;
end
else
begin
if(resetFlag)
begin
resetFlag <= 1'b0; // Wait one cycle to load first instruction
end
else if(jumpFlag)
begin
jumpFlag <= 1'b0; // Wait one cycle to load next instruction
end
else
begin
if(instructionAddr == 12'hFFF)
```

```
begin
instructionAddr <= 12'h000;
end
else
begin
instructionAddr <= enable ? instructionAddr + 12'h001 : instructionAddr;
end
end
end
end

endmodule
```

Now submodules:

ALU:

```
module ALU16Bit(
input [15:0] A,
input [15:0] B,
input [3:0] opcode,
input enable,
output [15:0] Y,
output [4:0] status
);

wire [15:0] opResult [15:0];
wire [15:0] opCarry;
// ADD - A, B, Cin, Result, Cout
Adder16Bit(A, B, 1'b0, opResult[0], opCarry[0]);
// SUB - A, B, Bin, Result, Bout
Subtractor16Bit(A, B, 1'b0, opResult[1], opCarry[1]);
// NEG - A, Result
Negator16Bit(A, opResult[2]);
assign opCarry[2] = 1'b0;
// INC - A, 1, Cin, Result, Cout
Adder16Bit(A, 1'b1, 1'b0, opResult[3], opCarry[3]);
// DEC - A, 1, Bin, Result, Bout
Subtractor16Bit(A, 1'b1, 1'b0, opResult[4], opCarry[4]);
// MOV (Pass-through) - A, Result
assign opResult[5] = A;
assign opCarry[5] = 1'b0;
// AND - A, B, Result
assign opResult[6] = A & B;
assign opCarry[6] = 1'b0;
// OR - A, B, Result
assign opResult[7] = A | B;
assign opCarry[7] = 1'b0;
// XOR - A, B, Result
assign opResult[8] = A ^ B;
```

```
assign opCarry[8] = 1'b0;
// NOT - A, Result
assign opResult[9] = ~A;
assign opCarry[9] = 1'b0;
// ASR - A, B, Result
ArithmeticShiftRight16Bit(A, B[3:0], opResult[10]);
assign opCarry[10] = 1'b0;
// ASL - A, B, Result, Cout
ShiftLeft16Bit(A, B[3:0], opResult[11], opCarry[11]);
// LSR - A, B, Result
LogicalShiftRight16Bit(A, B[3:0], opResult[12]);
assign opCarry[12] = 1'b0;
// LSL - A, B, Result, Cout
ShiftLeft16Bit(A, B[3:0], opResult[13], opCarry[13]);
// CSR - A, B, Result
CircularShiftRight16Bit(A, B[3:0], opResult[14]);
assign opCarry[14] = 1'b0;

// CSL - A, B, Result
CircularShiftLeft16Bit(A, B[3:0], opResult[15]);
assign opCarry[15] = 1'b0;
// Decide the operation
wire [15:0] result;
MUX16x16(opResult[0], opResult[1], opResult[2], opResult[3], opResult[4], opResult[5],
opResult[6], opResult[7], opResult[8], opResult[9], opResult[10], opResult[11],
opResult[12], opResult[13], opResult[14], opResult[15], opcode, 1'b1, result);
// Decide the flags
wire [4:0] flags;
MUX16x1(opCarry, opcode, 1'b1, flags[0]); // Carry-Out
assign flags[1] = ~(result[0] | result[1] | result[2] | result[3] | result[4] | result[5] | result[6]
| result[7] | result[8] | result[9] | result[10] | result[11] | result[12] | result[13] | result[14] |
result[15]); // Zero
assign flags[2] = result[15]; // Negative
wire OV;
assign OV = flags[0] ^ flags[2];
MUX16x1({11'b000000000, OV, OV, 1'b0, OV, OV}, opcode, 1'b1, flags[3]); // Overflow
OddParity16Bit(result, flags[4]); // Parity
// Add tri-state buffer at end
assign Y = enable ? result : 16'b0000000000000000;
assign status = enable ? flags : 5'b00000;
endmodule
```

Operations related to ALU:

Adder: (ADD)

```
module Adder16Bit(
input [15:0] A,
input [15:0] B,
input Cin,
output [15:0] Result,
output Cout
);
wire [14:0] w;
FullAdder(A[0], B[0], Cin, Result[0], w[0]);
FullAdder(A[1], B[1], w[0], Result[1], w[1]);
FullAdder(A[2], B[2], w[1], Result[2], w[2]);
FullAdder(A[3], B[3], w[2], Result[3], w[3]);
FullAdder(A[4], B[4], w[3], Result[4], w[4]);
FullAdder(A[5], B[5], w[4], Result[5], w[5]);
FullAdder(A[6], B[6], w[5], Result[6], w[6]);
FullAdder(A[7], B[7], w[6], Result[7], w[7]);
FullAdder(A[8], B[8], w[7], Result[8], w[8]);
FullAdder(A[9], B[9], w[8], Result[9], w[9]);
FullAdder(A[10], B[10], w[9], Result[10], w[10]);
FullAdder(A[11], B[11], w[10], Result[11], w[11]);
FullAdder(A[12], B[12], w[11], Result[12], w[12]);
FullAdder(A[13], B[13], w[12], Result[13], w[13]);
FullAdder(A[14], B[14], w[13], Result[14], w[14]);
FullAdder(A[15], B[15], w[14], Result[15], Cout);

endmodule
```

Subtractor: (SUB)

```
module Subtractor16Bit(
input [15:0] A,
input [15:0] B,
input Bin, // Borrow in
output [15:0] Result,
output Bout // Borrow out
);

wire Cout;
Adder16Bit(A, ~B, ~Bin, Result, Cout);
assign Bout = ~Cout;

endmodule
```

Negator: (NEG)

```
module Negator16Bit(
input [15:0] X,
output [15:0] Y
);

wire w;
Adder16Bit(~X, 16'h0001, 1'b0, Y, w);

endmodule
```

Arithmetic Shift Right: (ASR)

```
module ArithmeticShiftRight16Bit(
input [15:0] A,
input [3:0] B,
output [15:0] C
);

wire [15:0] w [15:0];
assign w[0] = A;
assign w[1] = {A[15], A[15:1]};
assign w[2] = {A[15], 1'b0, A[15:2]};
assign w[3] = {A[15], 2'b00, A[15:3]};
assign w[4] = {A[15], 3'b000, A[15:4]};
assign w[5] = {A[15], 4'b0000, A[15:5]};
assign w[6] = {A[15], 5'b00000, A[15:6]};
assign w[7] = {A[15], 6'b000000, A[15:7]};
assign w[8] = {A[15], 7'b00000000, A[15:8]};
assign w[9] = {A[15], 8'b000000000, A[15:9]};
assign w[10] = {A[15], 9'b000000000, A[15:10]};
assign w[11] = {A[15], 10'b0000000000, A[15:11]};
assign w[12] = {A[15], 11'b00000000000, A[15:12]};
assign w[13] = {A[15], 12'b000000000000, A[15:13]};
assign w[14] = {A[15], 13'b0000000000000, A[15:14]};
assign w[15] = {A[15], 14'b00000000000000, A[15]};

MUX16x16(w[0], w[1], w[2], w[3], w[4], w[5], w[6], w[7], w[8], w[9], w[10], w[11], w[12],
w[13], w[14], w[15], B, 1'b1, C);

endmodule
```

Shift Left: (LSL and ASL)

```
module ShiftLeft16Bit(
input [15:0] A,
input [3:0] B,
output [15:0] C,
output Cout
);

wire [15:0] w [15:0];
assign w[0] = A;
assign w[1] = {A[14:0], 1'b0};
assign w[2] = {A[13:0], 2'b00};
assign w[3] = {A[12:0], 3'b000};
assign w[4] = {A[11:0], 4'b0000};
assign w[5] = {A[10:0], 5'b00000};
assign w[6] = {A[9:0], 6'b000000};
assign w[7] = {A[8:0], 7'b0000000};
assign w[8] = {A[7:0], 8'b00000000};
assign w[9] = {A[6:0], 9'b000000000};
assign w[10] = {A[5:0], 10'b0000000000};
assign w[11] = {A[4:0], 11'b00000000000};
assign w[12] = {A[3:0], 12'b000000000000};
assign w[13] = {A[2:0], 13'b0000000000000};
assign w[14] = {A[1:0], 14'b00000000000000};
assign w[15] = {A[0], 15'b000000000000000};
wire [15:0] carry;
assign carry[0] = 1'b0;
assign carry[1] = A[15];
assign carry[2] = A[15] | A[14];
assign carry[3] = A[15] | A[14] | A[13];
assign carry[4] = A[15] | A[14] | A[13] | A[12];
assign carry[5] = A[15] | A[14] | A[13] | A[12] | A[11];
assign carry[6] = A[15] | A[14] | A[13] | A[12] | A[11] | A[10];
assign carry[7] = A[15] | A[14] | A[13] | A[12] | A[11] | A[10] | A[9];
assign carry[8] = A[15] | A[14] | A[13] | A[12] | A[11] | A[10] | A[9] | A[8];
assign carry[9] = A[15] | A[14] | A[13] | A[12] | A[11] | A[10] | A[9] | A[8] | A[7];
assign carry[10] = A[15] | A[14] | A[13] | A[12] | A[11] | A[10] | A[9] | A[8] | A[7] | A[6];
assign carry[11] = A[15] | A[14] | A[13] | A[12] | A[11] | A[10] | A[9] | A[8] | A[7] | A[6] |
A[5];
assign carry[12] = A[15] | A[14] | A[13] | A[12] | A[11] | A[10] | A[9] | A[8] | A[7] | A[6] |
A[5] | A[4];
assign carry[13] = A[15] | A[14] | A[13] | A[12] | A[11] | A[10] | A[9] | A[8] | A[7] | A[6] |
A[5] | A[4] | A[3];
assign carry[14] = A[15] | A[14] | A[13] | A[12] | A[11] | A[10] | A[9] | A[8] | A[7] | A[6] |
A[5] | A[4] | A[3] | A[2];
assign carry[15] = A[15] | A[14] | A[13] | A[12] | A[11] | A[10] | A[9] | A[8] | A[7] | A[6] |
A[5] | A[4] | A[3] | A[2] | A[1];

MUX16x16(w[0], w[1], w[2], w[3], w[4], w[5], w[6], w[7], w[8], w[9], w[10], w[11], w[12],
w[13], w[14], w[15], B, 1'b1, C);
```

```
MUX16x1(carry, B, 1'b1, Cout);

endmodule
```

Logical Shift Right: (LSR)

```
module LogicalShiftRight16Bit(
input [15:0] A,
input [3:0] B,
output [15:0] C
);

wire [15:0] w [15:0];
assign w[0] = A;
assign w[1] = {1'b0, A[15:1]};
assign w[2] = {2'b00, A[15:2]};
assign w[3] = {3'b000, A[15:3]};
assign w[4] = {4'b0000, A[15:4]};
assign w[5] = {5'b00000, A[15:5]};
assign w[6] = {6'b000000, A[15:6]};
assign w[7] = {7'b0000000, A[15:7]};
assign w[8] = {8'b00000000, A[15:8]};
assign w[9] = {9'b000000000, A[15:9]};
assign w[10] = {10'b0000000000, A[15:10]};
assign w[11] = {11'b00000000000, A[15:11]};
assign w[12] = {12'b000000000000, A[15:12]};
assign w[13] = {13'b0000000000000, A[15:13]};
assign w[14] = {14'b00000000000000, A[15:14]};
assign w[15] = {15'b000000000000000, A[15]};

MUX16x16(w[0], w[1], w[2], w[3], w[4], w[5], w[6], w[7], w[8], w[9], w[10], w[11], w[12],
w[13], w[14], w[15], B, 1'b1, C);

endmodule
```

Circular Shift Right: (CSR)

```
module CircularShiftRight16Bit(
input [15:0] A,
input [3:0] B,
output [15:0] C
);

wire [15:0] w [15:0];
assign w[0] = A;
assign w[1] = {A[0], A[15:1]};
assign w[2] = {A[1:0], A[15:2]};
assign w[3] = {A[2:0], A[15:3]};
```

```
assign w[4] = {A[3:0], A[15:4]};
assign w[5] = {A[4:0], A[15:5]};
assign w[6] = {A[5:0], A[15:6]};
assign w[7] = {A[6:0], A[15:7]};
assign w[8] = {A[7:0], A[15:8]};
assign w[9] = {A[8:0], A[15:9]};
assign w[10] = {A[9:0], A[15:10]};
assign w[11] = {A[10:0], A[15:11]};
assign w[12] = {A[11:0], A[15:12]};
assign w[13] = {A[12:0], A[15:13]};
assign w[14] = {A[13:0], A[15:14]};
assign w[15] = {A[14:0], A[15]};

MUX16x16(w[0], w[1], w[2], w[3], w[4], w[5], w[6], w[7], w[8], w[9], w[10], w[11], w[12],
w[13], w[14], w[15], B, 1'b1, C);

endmodule
```

Circular Shift Left: (CSL)

```
module CircularShiftLeft16Bit(
input [15:0] A,
input [3:0] B,
output [15:0] C
);

wire [15:0] w [15:0];
assign w[0] = A;
assign w[1] = {A[14:0], A[15]};
assign w[2] = {A[13:0], A[15:14]};
assign w[3] = {A[12:0], A[15:13]};
assign w[4] = {A[11:0], A[15:12]};
assign w[5] = {A[10:0], A[15:11]};
assign w[6] = {A[9:0], A[15:10]};
assign w[7] = {A[8:0], A[15:9]};
assign w[8] = {A[7:0], A[15:8]};
assign w[9] = {A[6:0], A[15:7]};
assign w[10] = {A[5:0], A[15:6]};
assign w[11] = {A[4:0], A[15:5]};
assign w[12] = {A[3:0], A[15:4]};
assign w[13] = {A[2:0], A[15:3]};
assign w[14] = {A[1:0], A[15:2]};
assign w[15] = {A[0], A[15:1]};

MUX16x16(w[0], w[1], w[2], w[3], w[4], w[5], w[6], w[7], w[8], w[9], w[10], w[11], w[12],
w[13], w[14], w[15], B, 1'b1, C);

endmodule
```

Multiplexers:

2-bit select, 1-bit data:

```
module MUX2x1(
input [1:0] data,
input sel,
input enable,
output out
);
wire selected;
assign selected = data[0] & ~sel | data[1] & sel;
assign out = enable ? selected : 1'b0;

endmodule
```

2-bit select, 4-bit data:

```
module MUX2x4(
input [3:0] data0,
input [3:0] data1,
input sel,
input enable,
output [3:0] out
);
MUX2x1({data1[0], data0[0]}, sel, enable, out[0]);
MUX2x1({data1[1], data0[1]}, sel, enable, out[1]);
MUX2x1({data1[2], data0[2]}, sel, enable, out[2]);
MUX2x1({data1[3], data0[3]}, sel, enable, out[3]);
endmodule
```

2-bit select, 12-bit data:

```
module MUX2x12(
input [11:0] data0,
input [11:0] data1,
input sel,
input enable,
output [11:0] out
);
MUX2x1({data1[0], data0[0]}, sel, enable, out[0]);
MUX2x1({data1[1], data0[1]}, sel, enable, out[1]);
MUX2x1({data1[2], data0[2]}, sel, enable, out[2]);
MUX2x1({data1[3], data0[3]}, sel, enable, out[3]);
MUX2x1({data1[4], data0[4]}, sel, enable, out[4]);
MUX2x1({data1[5], data0[5]}, sel, enable, out[5]);
MUX2x1({data1[6], data0[6]}, sel, enable, out[6]);
MUX2x1({data1[7], data0[7]}, sel, enable, out[7]);
MUX2x1({data1[8], data0[8]}, sel, enable, out[8]);
MUX2x1({data1[9], data0[9]}, sel, enable, out[9]);
```

```
MUX2x1({data1[10], data0[10]}, sel, enable, out[10]);
MUX2x1({data1[11], data0[11]}, sel, enable, out[11]);
endmodule
```

2-bit select, 16-bit data:

```
module MUX2x16(
input [15:0] data0,
input [15:0] data1,
input sel,
input enable,
output [15:0] out
);
MUX2x1({data1[0], data0[0]}, sel, enable, out[0]);
MUX2x1({data1[1], data0[1]}, sel, enable, out[1]);
MUX2x1({data1[2], data0[2]}, sel, enable, out[2]);
MUX2x1({data1[3], data0[3]}, sel, enable, out[3]);
MUX2x1({data1[4], data0[4]}, sel, enable, out[4]);
MUX2x1({data1[5], data0[5]}, sel, enable, out[5]);
MUX2x1({data1[6], data0[6]}, sel, enable, out[6]);
MUX2x1({data1[7], data0[7]}, sel, enable, out[7]);
MUX2x1({data1[8], data0[8]}, sel, enable, out[8]);
MUX2x1({data1[9], data0[9]}, sel, enable, out[9]);
MUX2x1({data1[10], data0[10]}, sel, enable, out[10]);
MUX2x1({data1[11], data0[11]}, sel, enable, out[11]);
MUX2x1({data1[12], data0[12]}, sel, enable, out[12]);
MUX2x1({data1[13], data0[13]}, sel, enable, out[13]);
MUX2x1({data1[14], data0[14]}, sel, enable, out[14]);
MUX2x1({data1[15], data0[15]}, sel, enable, out[15]);
endmodule
```

4-bit select, 1-bit data:

```
module MUX4x1(
input [3:0] data,
input [1:0] sel,
input enable,
output out
);
wire [1:0] w;
MUX2x1({data[2], data[0]}, sel[1], 1'b1, w[0]);
MUX2x1({data[3], data[1]}, sel[1], 1'b1, w[1]);
MUX2x1(w, sel[0], enable, out);

endmodule
```

8-bit select, 1-bit data:

```
module MUX8x1(
input [7:0] data,
input [2:0] sel,
input enable,
output out
);
wire [1:0] w;
MUX4x1({data[6], data[4], data[2], data[0]}, sel[2:1], 1'b1, w[0]);
MUX4x1({data[7], data[5], data[3], data[1]}, sel[2:1], 1'b1, w[1]);
MUX2x1(w, sel[0], enable, out);

endmodule
```

16-bit select, 1-bit data:

```
module MUX16x1(
input [15:0] data,
input [3:0] sel,
input enable,
output out
);
wire [1:0] w;
MUX8x1({data[14], data[12], data[10], data[8], data[6], data[4], data[2], data[0]},
sel[3:1], 1'b1, w[0]);
MUX8x1({data[15], data[13], data[11], data[9], data[7], data[5], data[3], data[1]},
sel[3:1], 1'b1, w[1]);
MUX2x1(w, sel[0], enable, out);

endmodule
```

16-bit select, 16-bit data:

```
module MUX16x16(
input [15:0] data0,
input [15:0] data1,
input [15:0] data2,
input [15:0] data3,
input [15:0] data4,
input [15:0] data5,
input [15:0] data6,
input [15:0] data7,
input [15:0] data8,
input [15:0] data9,
input [15:0] data10,
input [15:0] data11,
input [15:0] data12,
input [15:0] data13,
input [15:0] data14,
```

```
input [15:0] data15,
input [3:0] sel,
input enable,
output [15:0] out
);
wire [15:0] data_transpose [15:0];
assign data_transpose[0] = {data15[0], data14[0], data13[0], data12[0], data11[0],
data10[0], data9[0], data8[0], data7[0], data6[0], data5[0], data4[0], data3[0], data2[0],
data1[0], data0[0]};
assign data_transpose[1] = {data15[1], data14[1], data13[1], data12[1], data11[1],
data10[1], data9[1], data8[1], data7[1], data6[1], data5[1], data4[1], data3[1], data2[1],
data1[1], data0[1]};
assign data_transpose[2] = {data15[2], data14[2], data13[2], data12[2], data11[2],
data10[2], data9[2], data8[2], data7[2], data6[2], data5[2], data4[2], data3[2], data2[2],
data1[2], data0[2]};
assign data_transpose[3] = {data15[3], data14[3], data13[3], data12[3], data11[3],
data10[3], data9[3], data8[3], data7[3], data6[3], data5[3], data4[3], data3[3], data2[3],
data1[3], data0[3]};
assign data_transpose[4] = {data15[4], data14[4], data13[4], data12[4], data11[4],
data10[4], data9[4], data8[4], data7[4], data6[4], data5[4], data4[4], data3[4], data2[4],
data1[4], data0[4]};
assign data_transpose[5] = {data15[5], data14[5], data13[5], data12[5], data11[5],
data10[5], data9[5], data8[5], data7[5], data6[5], data5[5], data4[5], data3[5], data2[5],
data1[5], data0[5]};
assign data_transpose[6] = {data15[6], data14[6], data13[6], data12[6], data11[6],
data10[6], data9[6], data8[6], data7[6], data6[6], data5[6], data4[6], data3[6], data2[6],
data1[6], data0[6]};
assign data_transpose[7] = {data15[7], data14[7], data13[7], data12[7], data11[7],
data10[7], data9[7], data8[7], data7[7], data6[7], data5[7], data4[7], data3[7], data2[7],
data1[7], data0[7]};
assign data_transpose[8] = {data15[8], data14[8], data13[8], data12[8], data11[8],
data10[8], data9[8], data8[8], data7[8], data6[8], data5[8], data4[8], data3[8], data2[8],
data1[8], data0[8]};
assign data_transpose[9] = {data15[9], data14[9], data13[9], data12[9], data11[9],
data10[9], data9[9], data8[9], data7[9], data6[9], data5[9], data4[9], data3[9], data2[9],
data1[9], data0[9]};
assign data_transpose[10] = {data15[10], data14[10], data13[10], data12[10],
data11[10], data10[10], data9[10], data8[10], data7[10], data6[10], data5[10], data4[10],
data3[10], data2[10], data1[10], data0[10]};
assign data_transpose[11] = {data15[11], data14[11], data13[11], data12[11],
data11[11], data10[11], data9[11], data8[11], data7[11], data6[11], data5[11], data4[11],
data3[11], data2[11], data1[11], data0[11]};
assign data_transpose[12] = {data15[12], data14[12], data13[12], data12[12],
data11[12], data10[12], data9[12], data8[12], data7[12], data6[12], data5[12], data4[12],
data3[12], data2[12], data1[12], data0[12]};
assign data_transpose[13] = {data15[13], data14[13], data13[13], data12[13],
data11[13], data10[13], data9[13], data8[13], data7[13], data6[13], data5[13], data4[13],
data3[13], data2[13], data1[13], data0[13]};
assign data_transpose[14] = {data15[14], data14[14], data13[14], data12[14],
data11[14], data10[14], data9[14], data8[14], data7[14], data6[14], data5[14], data4[14],
data3[14], data2[14], data1[14], data0[14]};
```

```
assign data_transpose[15] = {data15[15], data14[15], data13[15], data12[15],
data11[15], data10[15], data9[15], data8[15], data7[15], data6[15], data5[15], data4[15],
data3[15], data2[15], data1[15], data0[15]};
MUX16x1(data_transpose[0], sel, enable, out[0]);
MUX16x1(data_transpose[1], sel, enable, out[1]);
MUX16x1(data_transpose[2], sel, enable, out[2]);
MUX16x1(data_transpose[3], sel, enable, out[3]);
MUX16x1(data_transpose[4], sel, enable, out[4]);
MUX16x1(data_transpose[5], sel, enable, out[5]);
MUX16x1(data_transpose[6], sel, enable, out[6]);
MUX16x1(data_transpose[7], sel, enable, out[7]);
MUX16x1(data_transpose[8], sel, enable, out[8]);
MUX16x1(data_transpose[9], sel, enable, out[9]);
MUX16x1(data_transpose[10], sel, enable, out[10]);
MUX16x1(data_transpose[11], sel, enable, out[11]);
MUX16x1(data_transpose[12], sel, enable, out[12]);
MUX16x1(data_transpose[13], sel, enable, out[13]);
MUX16x1(data_transpose[14], sel, enable, out[14]);
MUX16x1(data_transpose[15], sel, enable, out[15]);
endmodule
```

Odd Parity module for the ALU result flag:

```
module OddParity16Bit(
input [15:0] A,
output odd
);

assign odd =
A[0]^A[1]^A[2]^A[3]^A[4]^A[5]^A[6]^A[7]^A[8]^A[9]^A[10]^A[11]^A[12]^A[13]^A[1
4]^A[15];

endmodule
```

Register File:

```
module RegisterFile(
input WR,
input RD,
input clk,
input reset,
input [3:0] z,
input [3:0] x,
input [3:0] y,
input [15:0] Rz,
output [15:0] Rx,
output [15:0] Ry
);
```

```
// Select which register to enable (Select Rz)
wire [15:0] read;
DEMUX1x16(WR, z, 1'b1, read);
// Define the registers
wire [15:0] regNet [15:0];
// R0
Register16Bit(Rz, clk, read[0], reset, regNet[0]);
// R1
Register16Bit(Rz, clk, read[1], reset, regNet[1]);
// R2
Register16Bit(Rz, clk, read[2], reset, regNet[2]);

// R3
Register16Bit(Rz, clk, read[3], reset, regNet[3]);
// R4
Register16Bit(Rz, clk, read[4], reset, regNet[4]);
// R5
Register16Bit(Rz, clk, read[5], reset, regNet[5]);
// R6
Register16Bit(Rz, clk, read[6], reset, regNet[6]);
// R7
Register16Bit(Rz, clk, read[7], reset, regNet[7]);
// R8
Register16Bit(Rz, clk, read[8], reset, regNet[8]);
// R9
Register16Bit(Rz, clk, read[9], reset, regNet[9]);
// R10
Register16Bit(Rz, clk, read[10], reset, regNet[10]);
// R11
Register16Bit(Rz, clk, read[11], reset, regNet[11]);
// ACC0
Register16Bit(Rz, clk, read[12], reset, regNet[12]);
// ACC1
Register16Bit(Rz, clk, read[13], reset, regNet[13]);
// ACC2
Register16Bit(Rz, clk, read[14], reset, regNet[14]);
// ACC3
Register16Bit(Rz, clk, read[15], reset, regNet[15]);
// Select which registers to read (Select Rx and Ry)
// Rx
MUX16x16(regNet[0], regNet[1], regNet[2], regNet[3],
regNet[4], regNet[5], regNet[6], regNet[7],
regNet[8], regNet[9], regNet[10], regNet[11],
regNet[12], regNet[13], regNet[14], regNet[15],
x, RD, Rx
);
// Ry
MUX16x16(regNet[0], regNet[1], regNet[2], regNet[3],
regNet[4], regNet[5], regNet[6], regNet[7],
regNet[8], regNet[9], regNet[10], regNet[11],
regNet[12], regNet[13], regNet[14], regNet[15],
```

```
y, RD, Ry
);
endmodule
```

Register modules:

5-bit register:

```
module Register5Bit(
input [4:0] D,
input clk,
input en,
input clr,
output reg [4:0] Q
);
initial begin
Q <= 5'h000;
end
always @(posedge clk or posedge clr)
begin
if(clr) begin
Q <= 5'h000;
end
else begin
Q <= en ? D: Q;
end
end

endmodule
```

6-bit register:

```
module Register6Bit(
input [5:0] D,
input clk,
input en,
input clr,
output reg [5:0] Q
);
initial begin
Q <= 6'h000;
end
always @(posedge clk or posedge clr)
begin
if(clr) begin
Q <= 6'h000;
end
else begin
Q <= en ? D: Q;
end
```

```
end

endmodule
```

12-bit register:

```
module Register12Bit(
input [11:0] D,
input clk,
input en,
input clr,
output reg [11:0] Q
);
initial begin
Q <= 12'h000;
end
always @(posedge clk or posedge clr)
begin
if(clr) begin
Q <= 12'h000;
end
else begin
Q <= en ? D: Q;
end
end

endmodule
```

16-bit register:

```
module Register16Bit(
input [15:0] D,
input clk,
input en,
input clr,
output reg [15:0] Q
);
initial begin
Q <= 16'h0000;
end
always @(posedge clk or posedge clr)
begin
if(clr) begin
Q <= 16'h0000;
end
else begin
Q <= en ? D: Q;
end
end
```

```
endmodule
```

Old program counter module:

```
module ProgramCounter(
input [11:0] addr,
input jump,
input en,
input clk,
input reset,
output [11:0] PC
);

Counter12Bit(
addr,
jump,
en,
clk,
reset,
PC
);

endmodule
```

12-bit Counter:

```
module Counter12Bit(
input [11:0] D,
input load,
input en,
input clk,
input reset,
output reg [11:0] Q
);

initial begin
Q <= 12'h000;
end
always @(posedge clk or posedge load or posedge reset)
begin
if(reset)
begin
Q <= 12'h000;
end
else if(load)
begin
Q <= en ? D : Q;
end
else
```

```
begin
Q <= en ? Q + 12'h001 : Q;
end
end

endmodule
```

Demultiplexers:

1-bit data, 2 outputs:

```
module DEMUX1x2(
input data,
input sel,
input en,
output [1:0] out
);

wire [1:0] w;
assign w[0] = ~sel & data;
assign w[1] = sel & data;
assign out = en ? w : 2'b00;

endmodule
```

1-bit data, 4 outputs:

```
module DEMUX1x4(
input data,
input [1:0] sel,
input en,
output [3:0] out
);

wire [1:0] w0;
wire [3:0] w1;
DEMUX1x2(data, sel[1], 1'b1, w0);
DEMUX1x2(w0[0], sel[0], 1'b1, w1[1:0]);
DEMUX1x2(w0[1], sel[0], 1'b1, w1[3:2]);
assign out = en ? w1 : 4'b0000;

endmodule
```

1-bit data, 8 outputs:

```
module DEMUX1x8(
input data,
input [2:0] sel,
input en,
```

```
output [7:0] out
);

wire [1:0] w0;
wire [7:0] w1;
DEMUX1x2(data, sel[2], 1'b1, w0);
DEMUX1x4(w0[0], sel[1:0], 1'b1, w1[3:0]);
DEMUX1x4(w0[1], sel[1:0], 1'b1, w1[7:4]);
assign out = en ? w1 : 8'b00000000;

endmodule
```

1-bit data, 16 outputs:

```
module DEMUX1x16(
input data,
input [3:0] sel,
input en,
output [15:0] out
);

wire [1:0] w0;
wire [15:0] w1;
DEMUX1x2(data, sel[3], 1'b1, w0);
DEMUX1x8(w0[0], sel[2:0], 1'b1, w1[7:0]);
DEMUX1x8(w0[1], sel[2:0], 1'b1, w1[15:8]);
assign out = en ? w1 : 16'b0000000000000000;

endmodule
```

FullAdder module:

```
module FullAdder(
input A,
input B,
input Cin,
output Sum,
output Cout
);

assign Sum = (A & ~B | ~A & B) & ~Cin | ~(A & ~B | ~A & B) & Cin;
assign Cout = A & B | A & Cin | B & Cin | A & B & Cin;

endmodule
```

Finally we created a tester so we can actually see the CPU work on FPGA by connecting ROM, RAM and digital tube binded to a spesific RAM address.

CPU Tester Module:

```
module CPUTester(
input clk,
input clkBtn,
input res,
input enable,
input [1:0] btn,
output reg [3:0] dig,
output [7:0] seg
);

// Block ROM:
wire [11:0] ROMaddr;
reg [31:0] ROMdata;
wire ROMsel; // cs
// Block RAM:
wire [11:0] RAMaddr;
wire [15:0] RAMdataIn;
reg [15:0] RAMdataOut;
wire RAMsel; // cs
wire RAMld; // ~we
wire RAMclr;
// Define CPU connections:
CPUv2(
~clk,
~res,
enable,
// Instruction fetch pins (ROM)
ROMaddr,
ROMdata,
ROMsel,
// Data load/str pins (RAM)
RAMaddr,
RAMdataOut,
RAMdataIn,
RAMsel,
RAMld,
RAMclr
);
// Define ROM functionality:
reg [31:0] ROM [63:0];
initial begin
$readmemh("ROM.hex", ROM);
end
always @(negedge clk)
begin
ROMdata <= ROMsel ? ROM[ROMaddr[7:0]] : 32'h00000000;
```

```
end
// Digital Tube value register:
reg [15:0] hex4;
// Define RAM functionality:
reg [15:0] RAM [63:0];
integer i;
always @(negedge clk or posedge RAMclr)
begin
if(RAMclr == 1'b1)
begin
for(i = 0; i < 64; i = i + 1)
begin
RAM[i] <= 16'h0000;
end
hex4 <= 16'h0000;
end
else
begin
if(RAMsel)
begin
if(RAMld)
begin
if(RAMaddr == 12'd111)
begin
// Bind buttons to RAM:
RAMdataOut <= {14'b00000000000000, ~btn};
end
else
begin
RAMdataOut <= RAM[RAMaddr];
end
end
else
begin
if(RAMaddr == 12'd110)
begin
// Bind RAM[1] as digital tube:
hex4 <= RAMdataIn;
end
else
begin
RAM[RAMaddr] <= RAMdataIn;
end
end
end
else
begin
RAMdataOut <= 16'h0000;
end
end
end
```

```
reg clock1KHz;
reg [15:0] pulseCount;
reg [2:0] counter;
reg [15:0] hexSel;
reg [6:0] hexEncoding;
initial begin
clock1KHz <= 1'b0;
pulseCount <= 16'd0;
dig <= 4'b0111;
counter <= 3'b000;
hexSel <= 16'h0000;
end
always @(negedge clk)
begin
if(~res)
begin
clock1KHz <= 1'b0;
pulseCount <= 16'd0;
end
// 50000 clock cycles are needed to count for 1KHz clock (50000 division):
if(pulseCount == 16'd50000)
begin
pulseCount <= 16'd0;
clock1KHz <= 1'b0;
end
else
begin
if(pulseCount == 16'd25000)
begin
clock1KHz <= 1'b1;
end
pulseCount <= pulseCount + 16'd1;
end
end
always @(posedge clock1KHz)
begin
if(~res)
begin
dig <= 4'b0111;
counter <= 3'b000;
hexSel <= 16'h0000;
end
if(counter == 3'b100)
begin
dig <= 4'b0111;
hexSel <= hex4;
counter <= 3'b000;
end
else
begin
dig <= {dig[2:0], dig[3]};
```

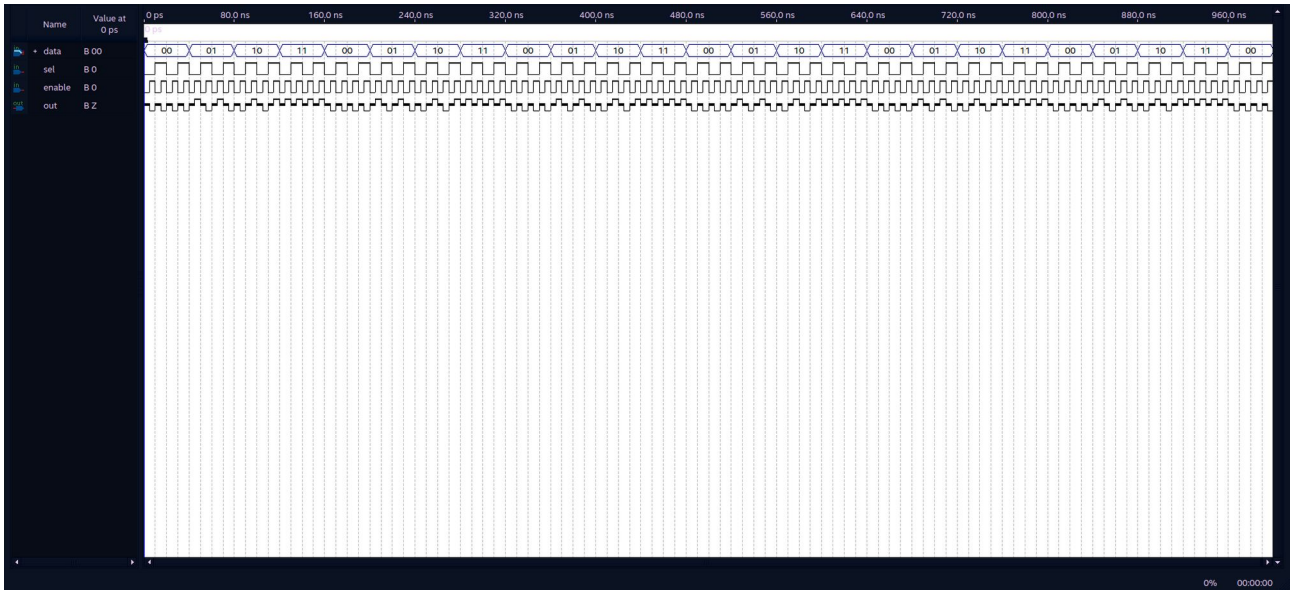
```
// Set digital tube to show recieved code:
case (hexSel[3:0])
4'b0000 : hexEncoding <= 7'h3f;
4'b0001 : hexEncoding <= 7'h06;
4'b0010 : hexEncoding <= 7'h5b;
4'b0011 : hexEncoding <= 7'h4f;
4'b0100 : hexEncoding <= 7'h66;
4'b0101 : hexEncoding <= 7'h6d;
4'b0110 : hexEncoding <= 7'h7d;
4'b0111 : hexEncoding <= 7'h07;
4'b1000 : hexEncoding <= 7'h7f;
4'b1001 : hexEncoding <= 7'h6f;
4'b1010 : hexEncoding <= 7'h77;
4'b1011 : hexEncoding <= 7'h7c;
4'b1100 : hexEncoding <= 7'h39;
4'b1101 : hexEncoding <= 7'h5e;
4'b1110 : hexEncoding <= 7'h79;
4'b1111 : hexEncoding <= 7'h71;
endcase
hexSel <= {4'b0000, hexSel[15:4]};
counter <= counter + 3'b001;
end
end
assign seg[7] = 1'b1;
assign seg[6] = ~hexEncoding[6];
assign seg[5] = ~hexEncoding[5];
assign seg[4] = ~hexEncoding[4];
assign seg[3] = ~hexEncoding[3];
assign seg[2] = ~hexEncoding[2];
assign seg[1] = ~hexEncoding[1];
assign seg[0] = ~hexEncoding[0];
endmodule
```

3 Simulations

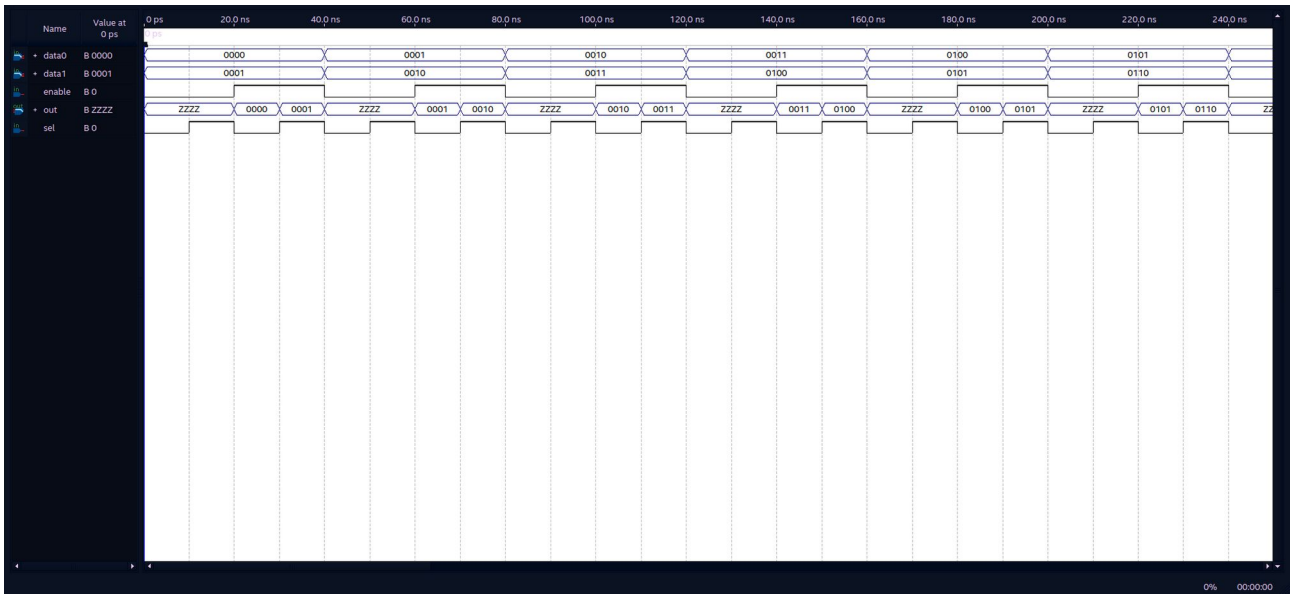
We simulated all of this modules using University Waveform Program as debugger.

Lets start from the bottom level modules upto top CPU module. (CPU cannot be simulated using waveform so we will show the CPU using CPUTester module on FPGA)

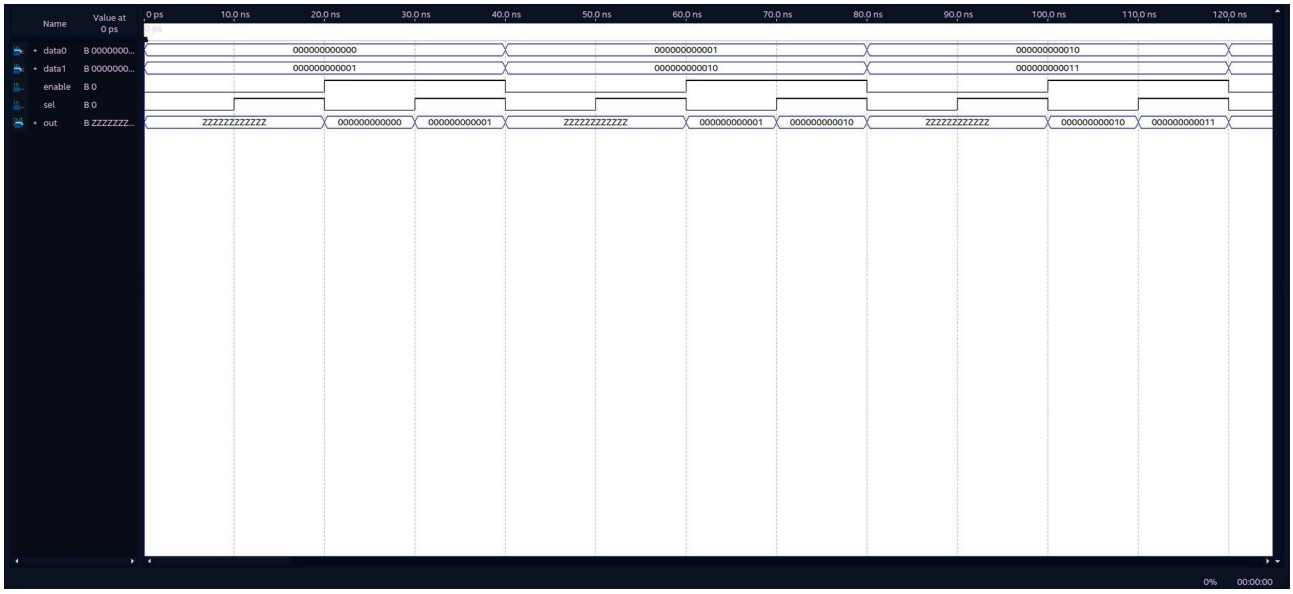
MUX2x1:



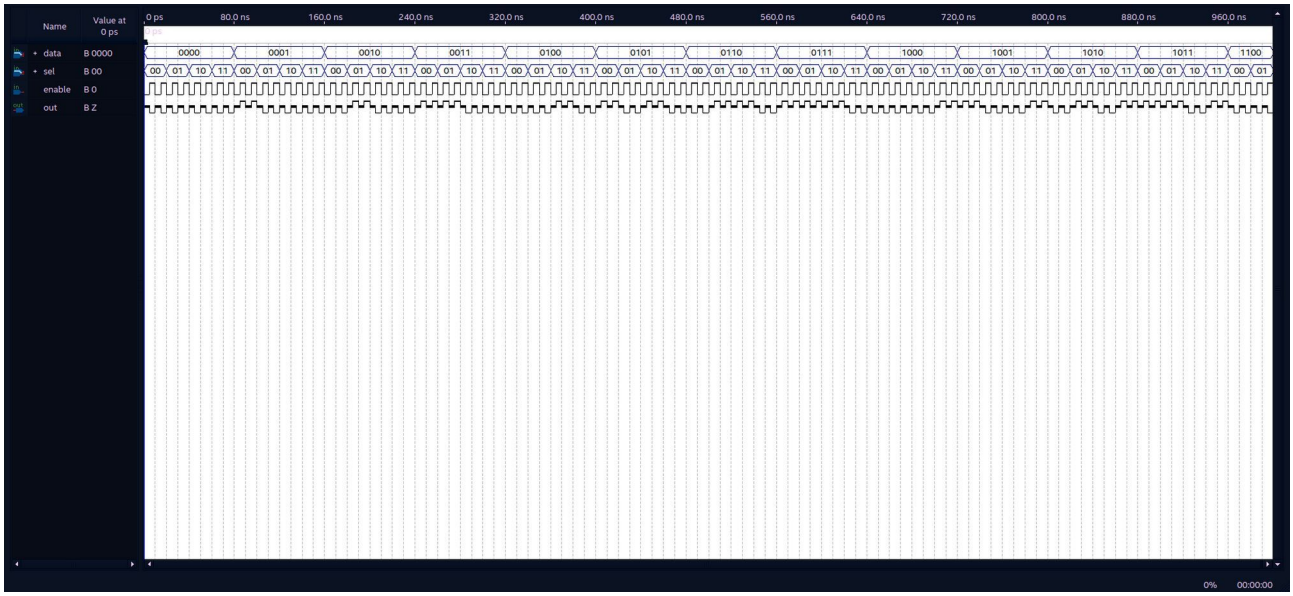
MUX2x4:



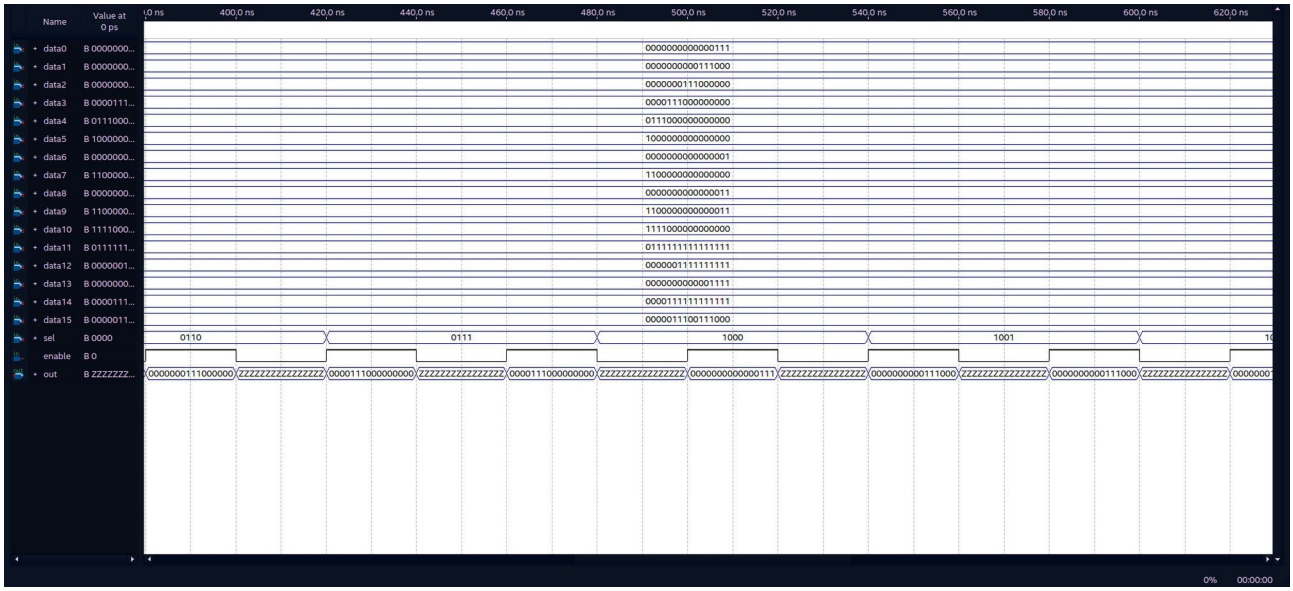
MUX2x12:



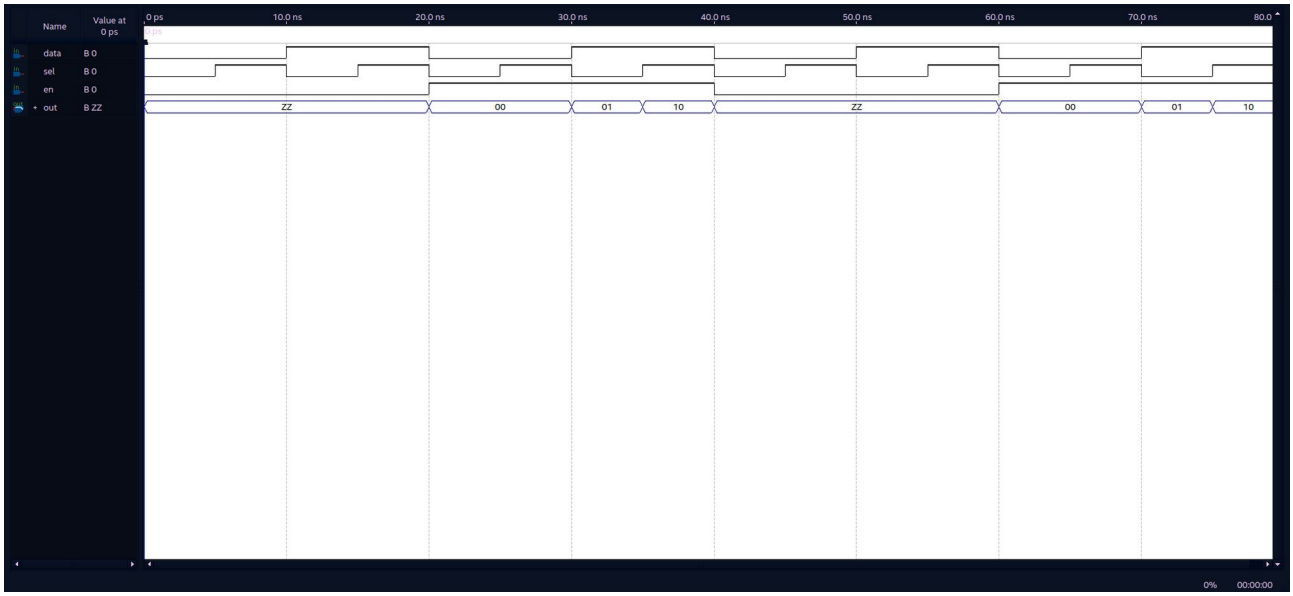
MUX4x1:



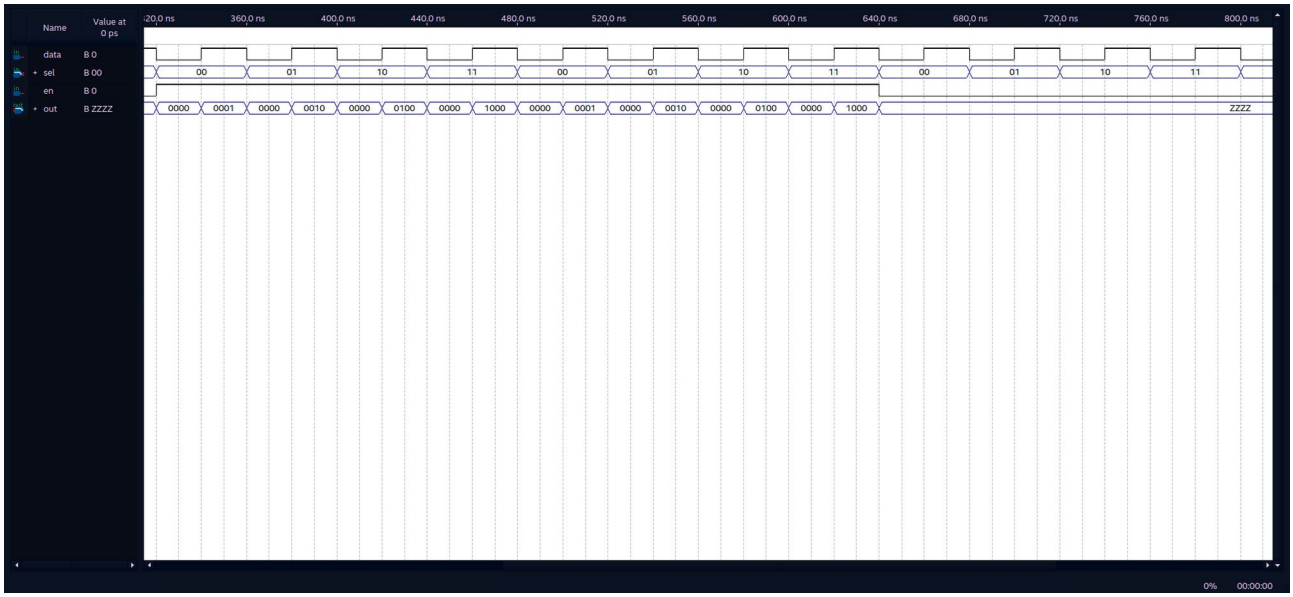
MUX16x16:



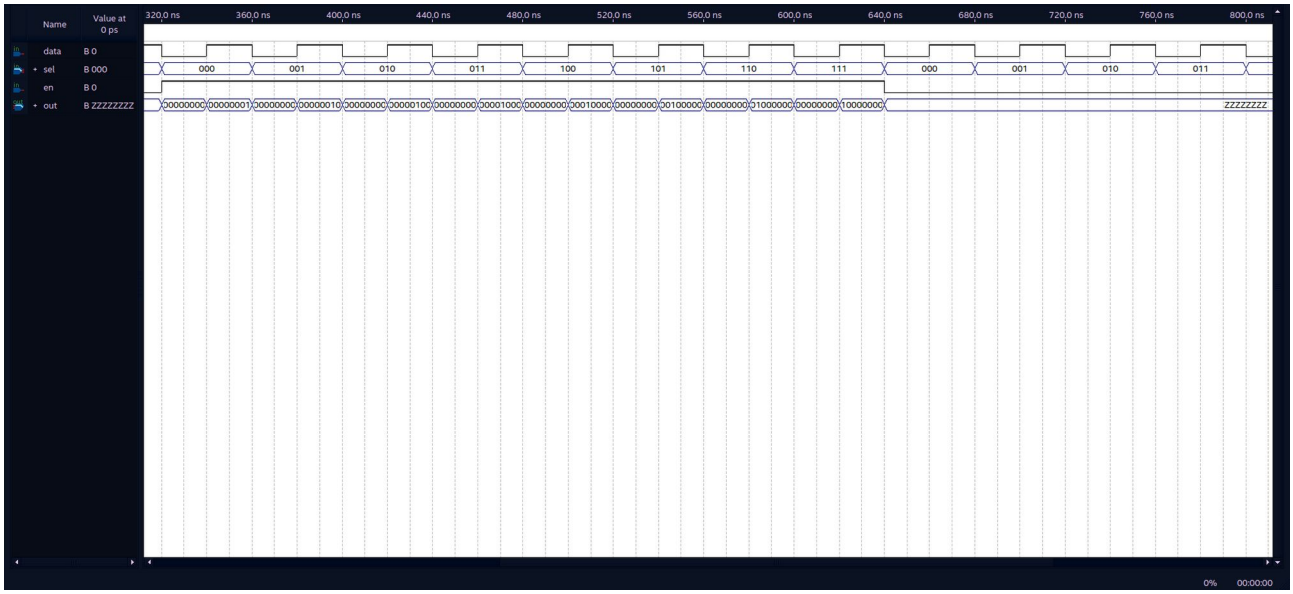
DEMUX1x2:



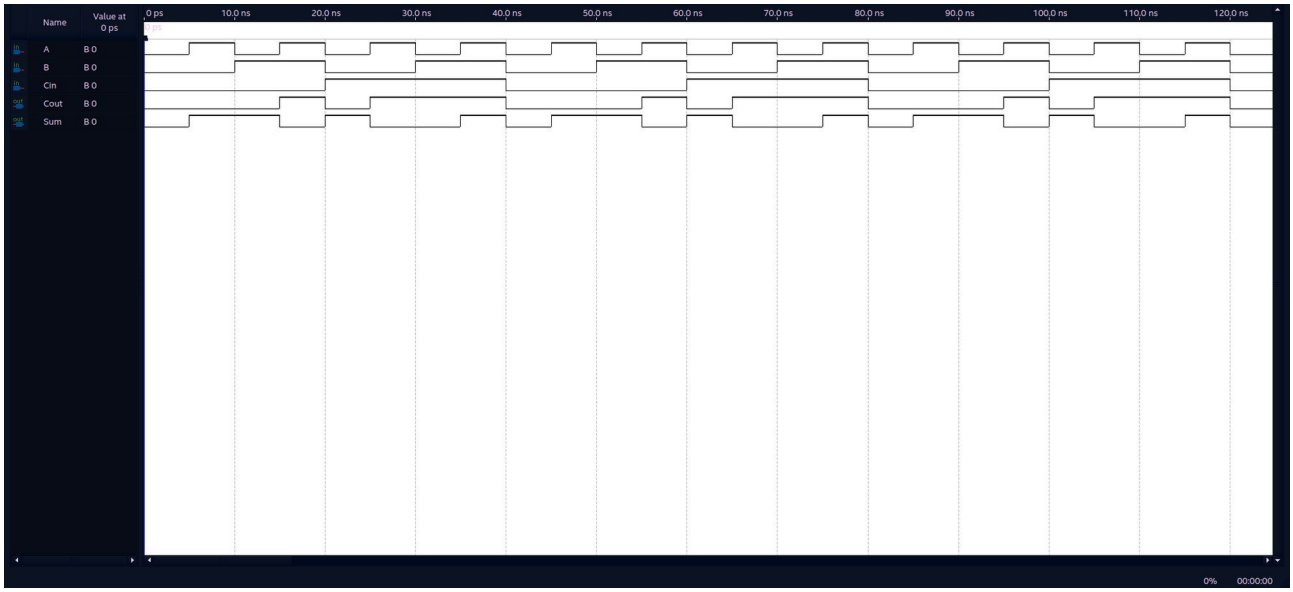
DEMUX1x4:



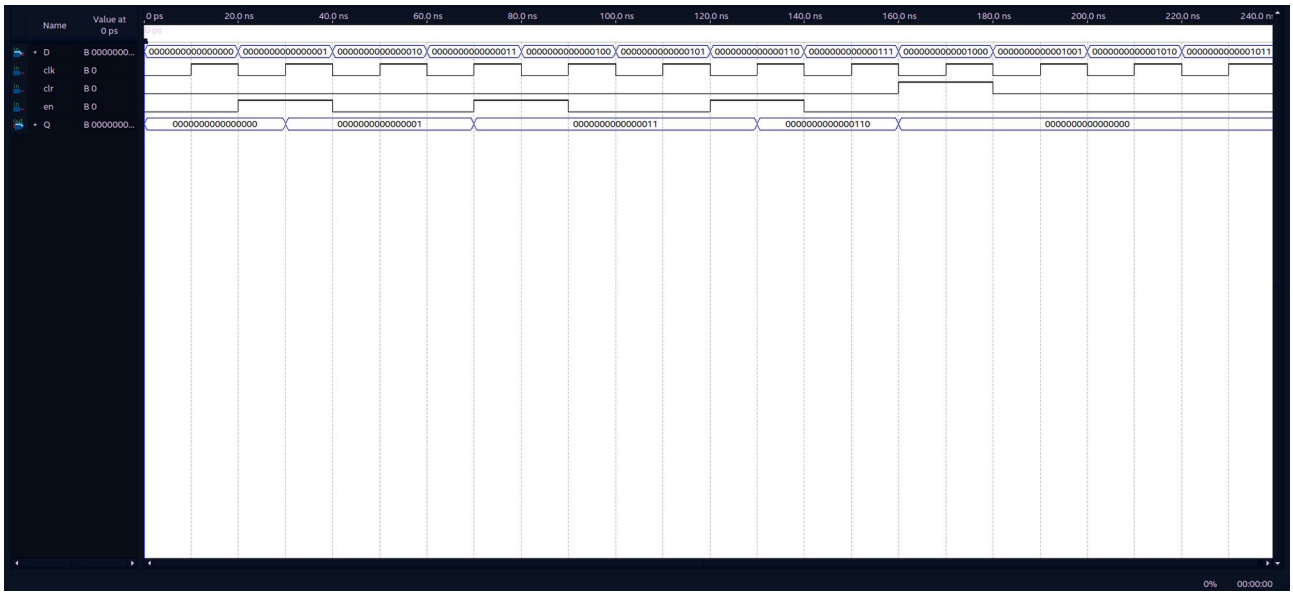
DEMUX1x8:



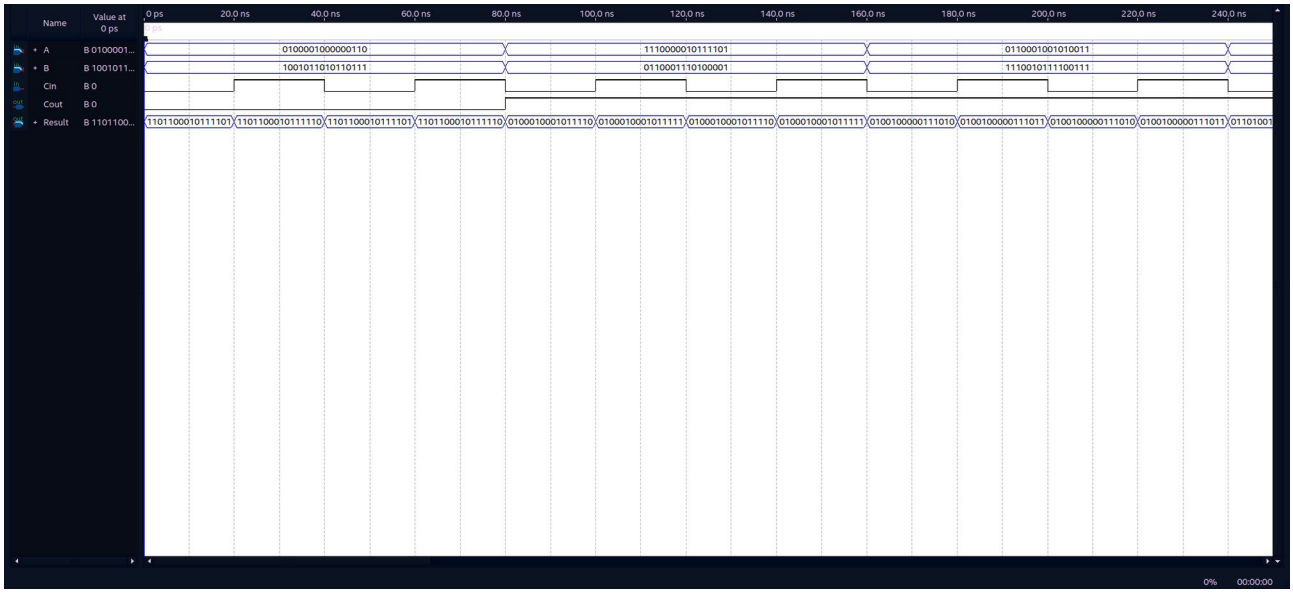
FullAdder:



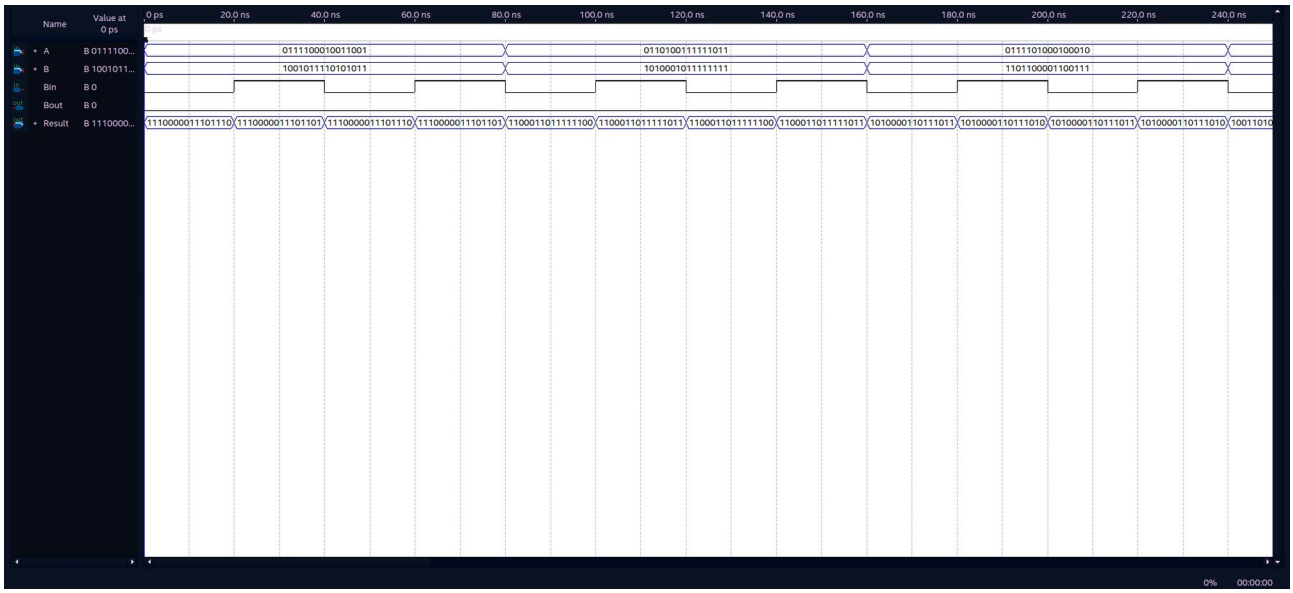
Register16Bit:



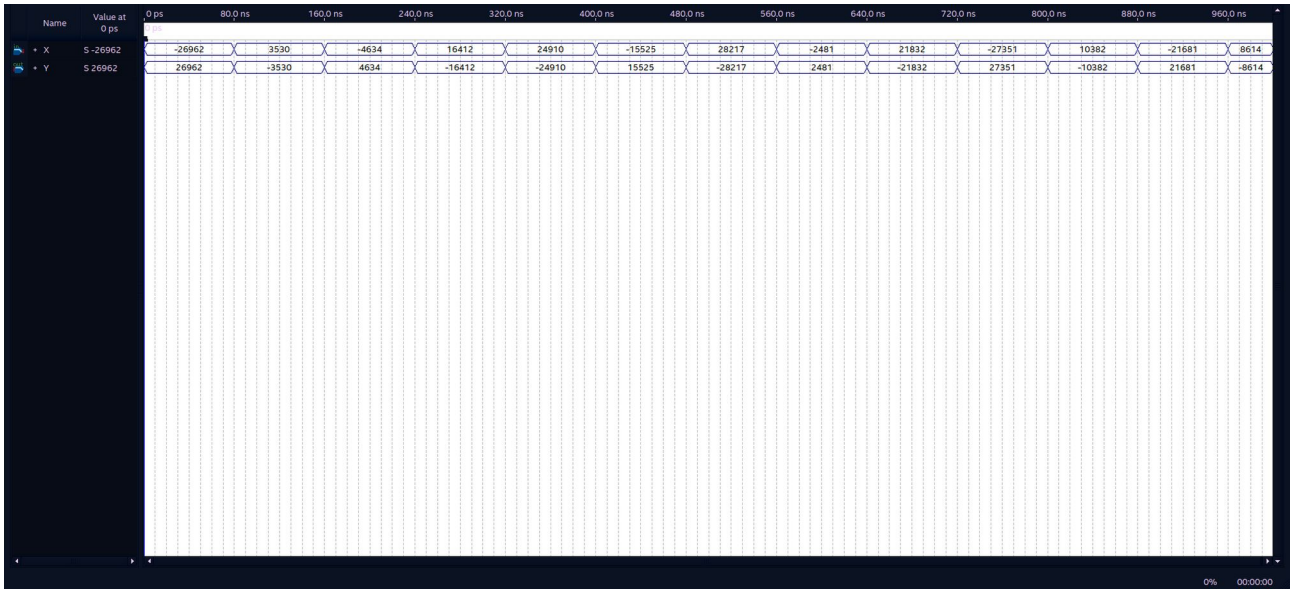
Adder16Bit:



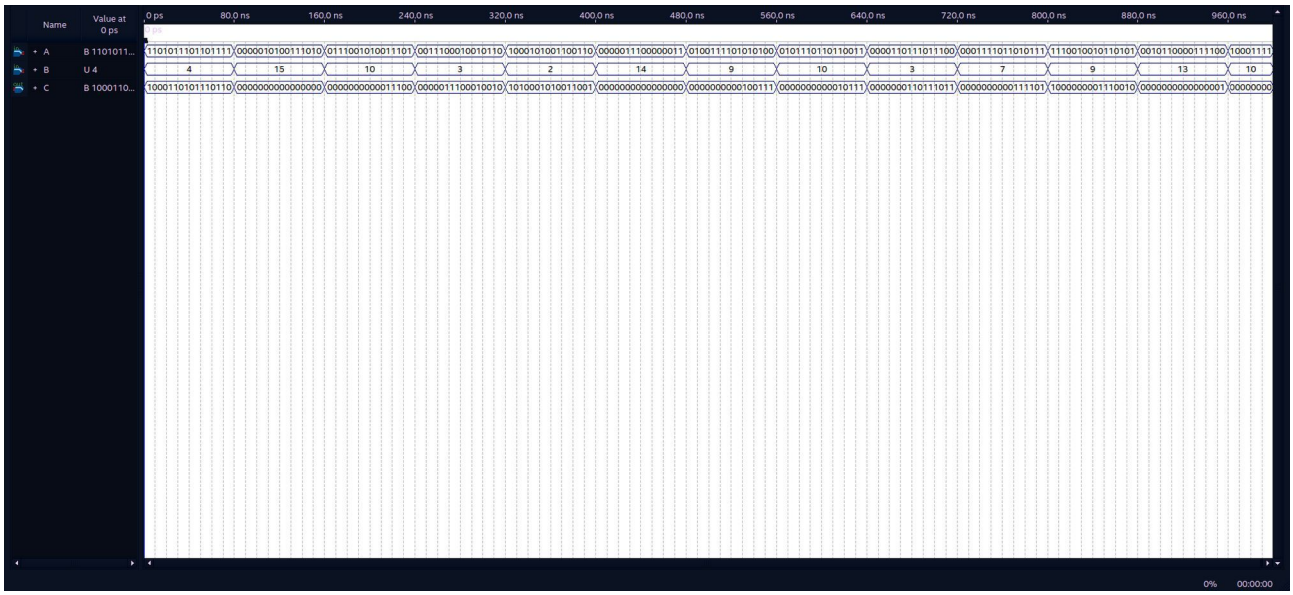
Subtractor16Bit:



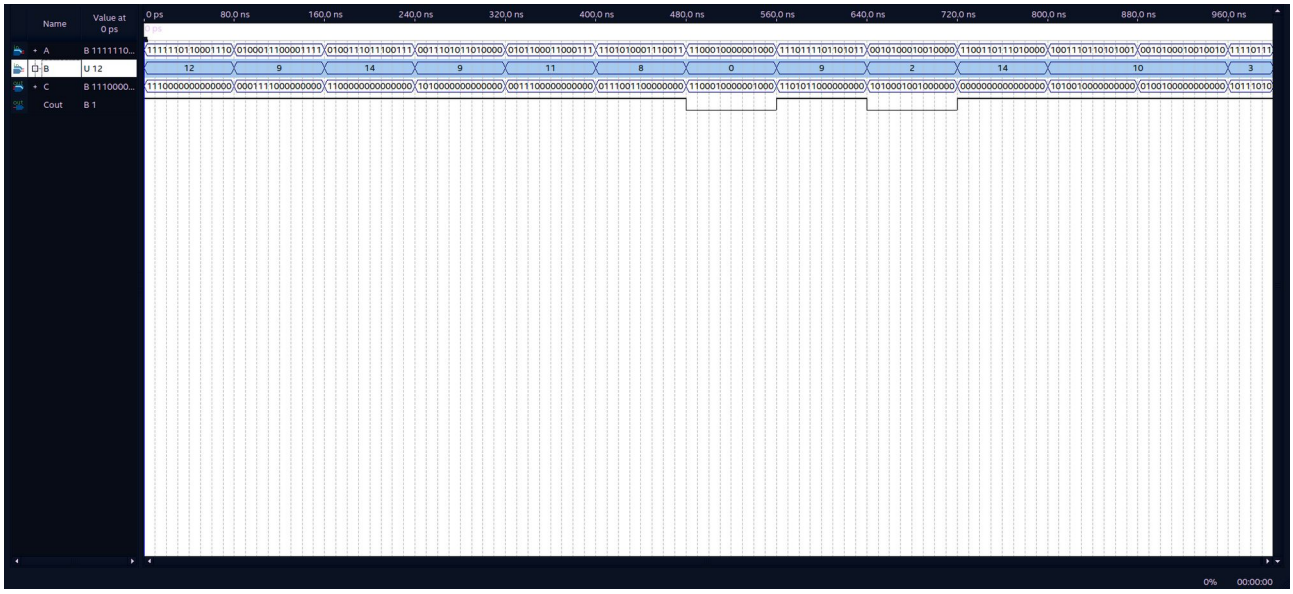
Negator16Bit:



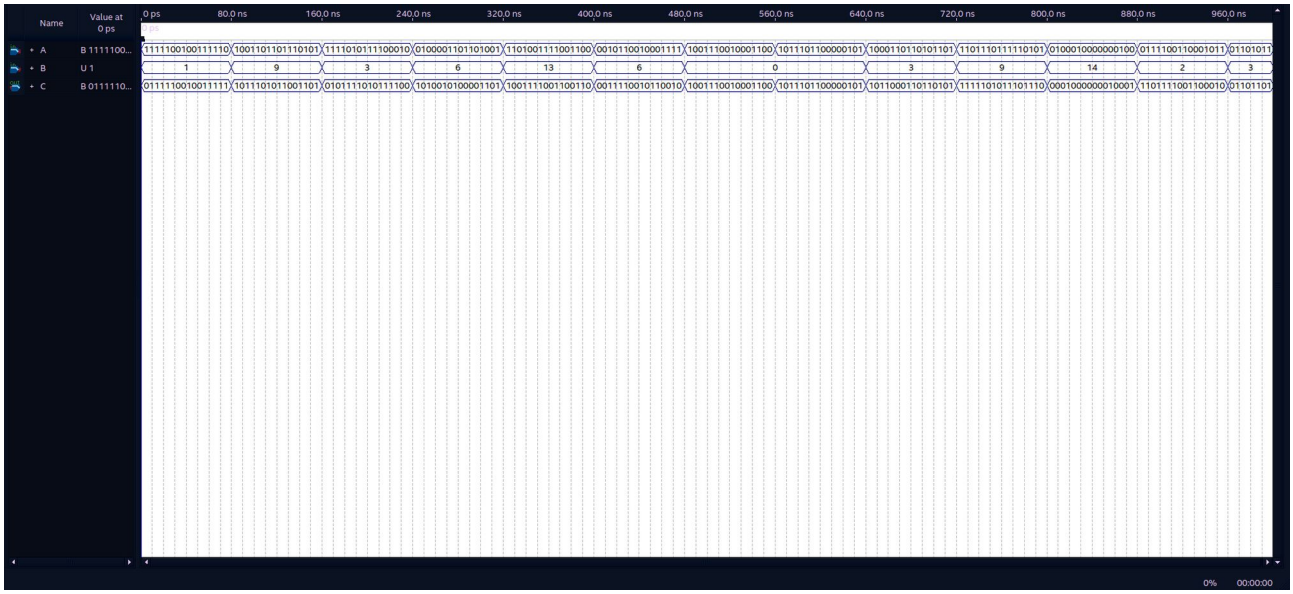
ArithmeticShiftRight16Bit:



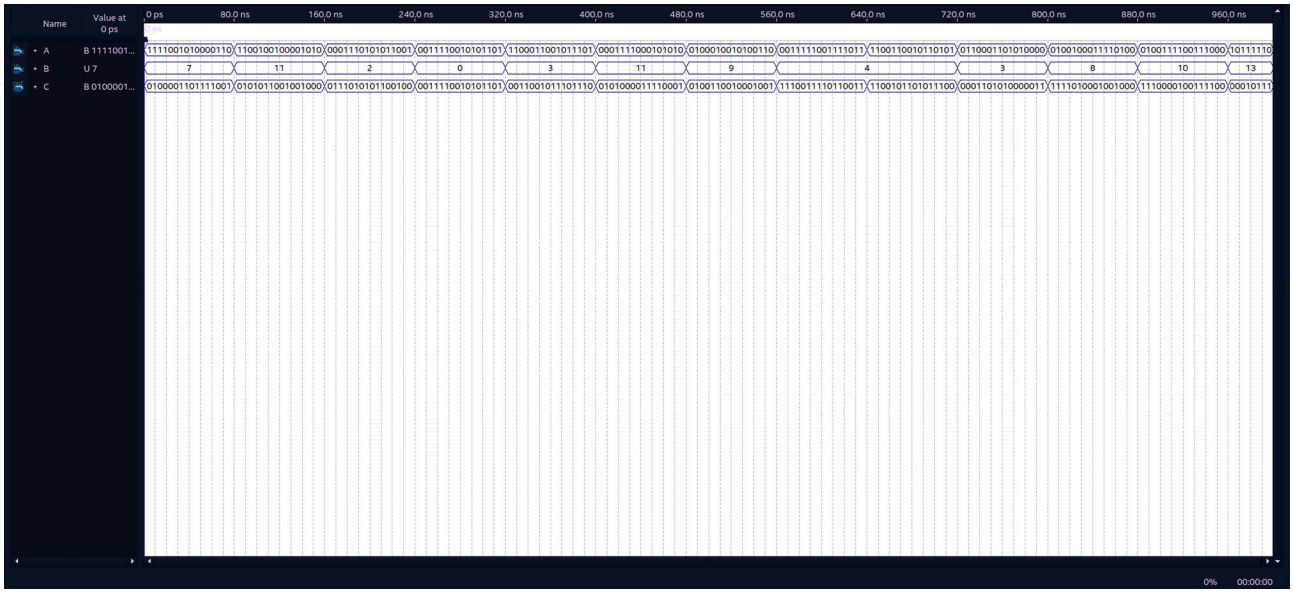
ShiftLeft16Bit:



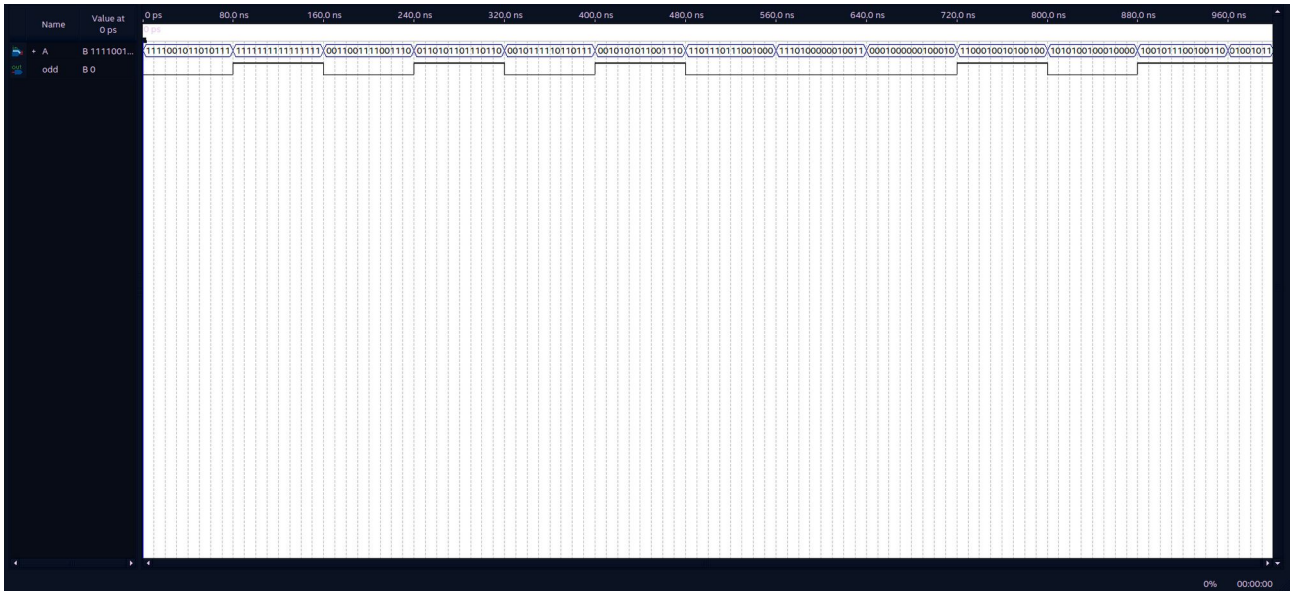
CircularShiftRight16Bit:



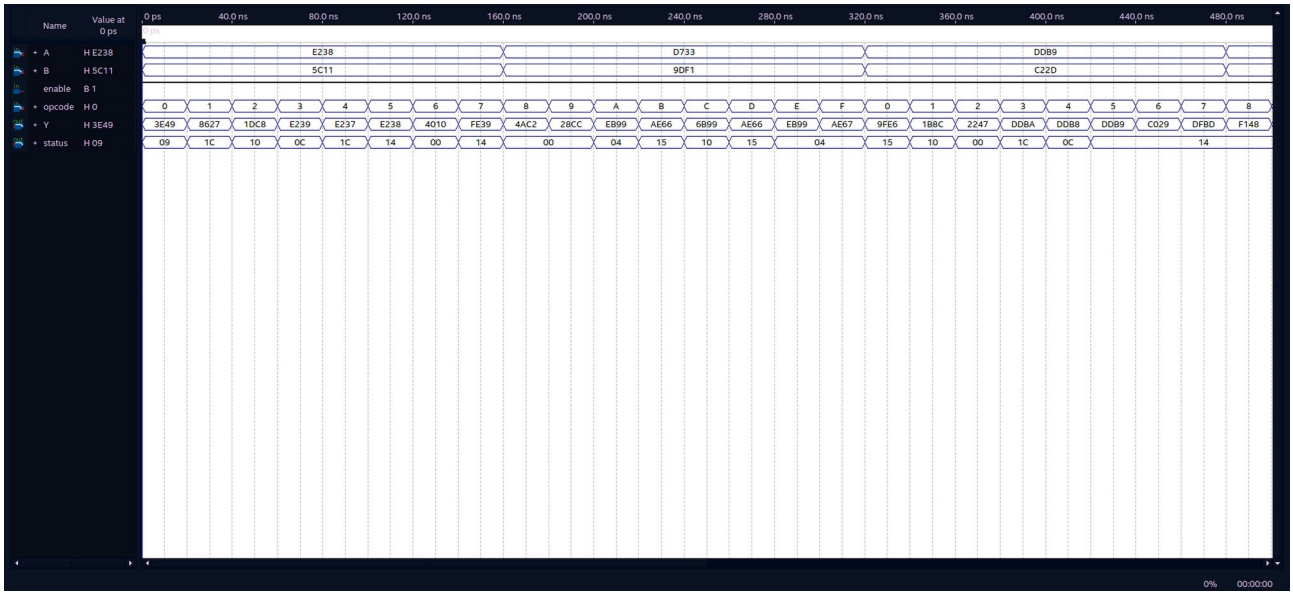
CircularShiftLeft16Bit:



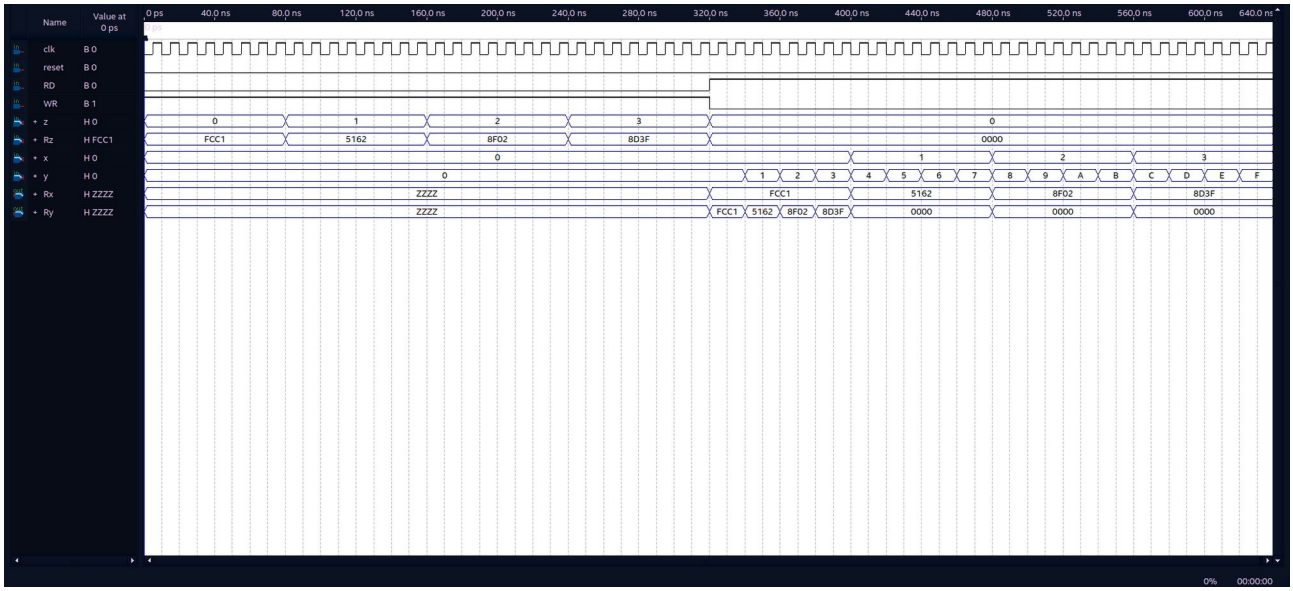
OddParity16Bit:



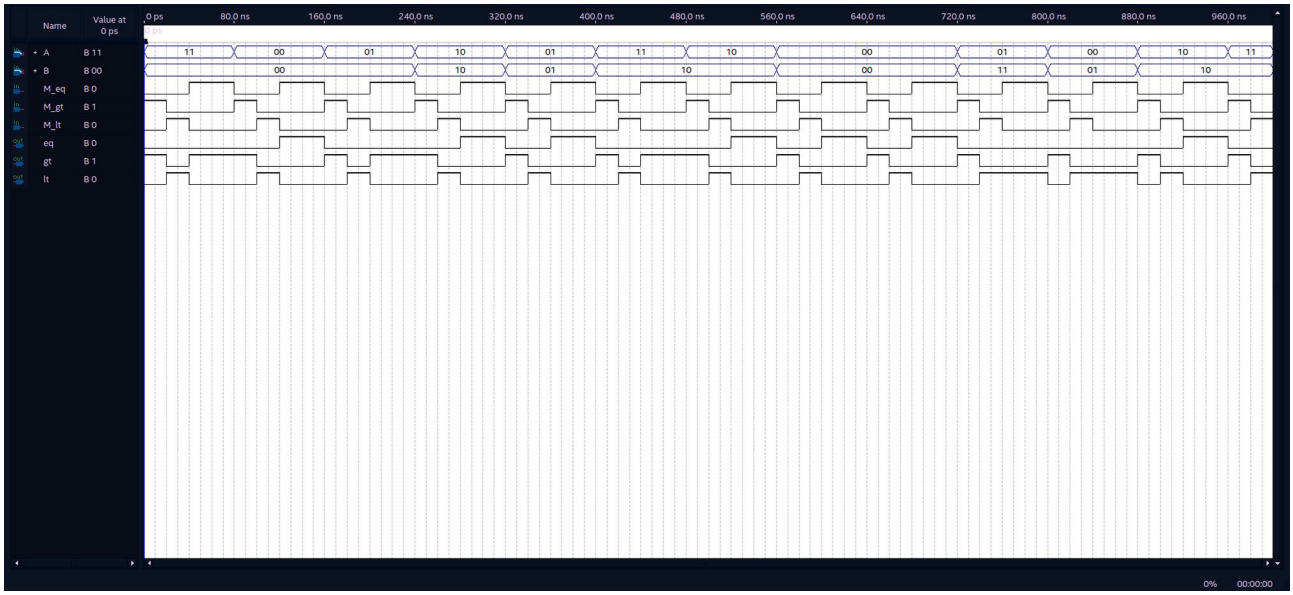
ALU16Bit:



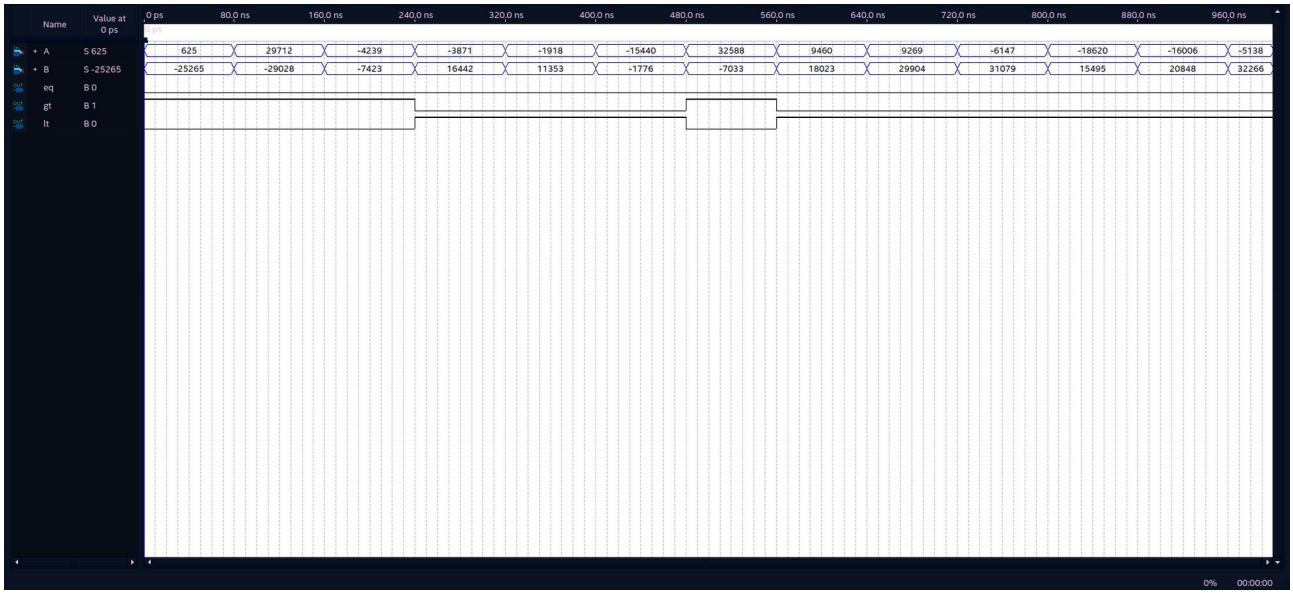
RegisterFile:



Comparator2Bit:



Comparator16Bit:



4 Practical Testing