EE 435 Verilog Digital Systems Modeling
Final Project

Anthony Donaldson
Matthew Erhardt

May 2nd 2018

Contents

Design Overview

We designed a β RISC processor using behavioral Verilog. We split up the design into a controller FSM and a data path that controls the ALU and memory. The controller FSM can be seen in Figure 1 and the data path table can be seen in Figure 2.

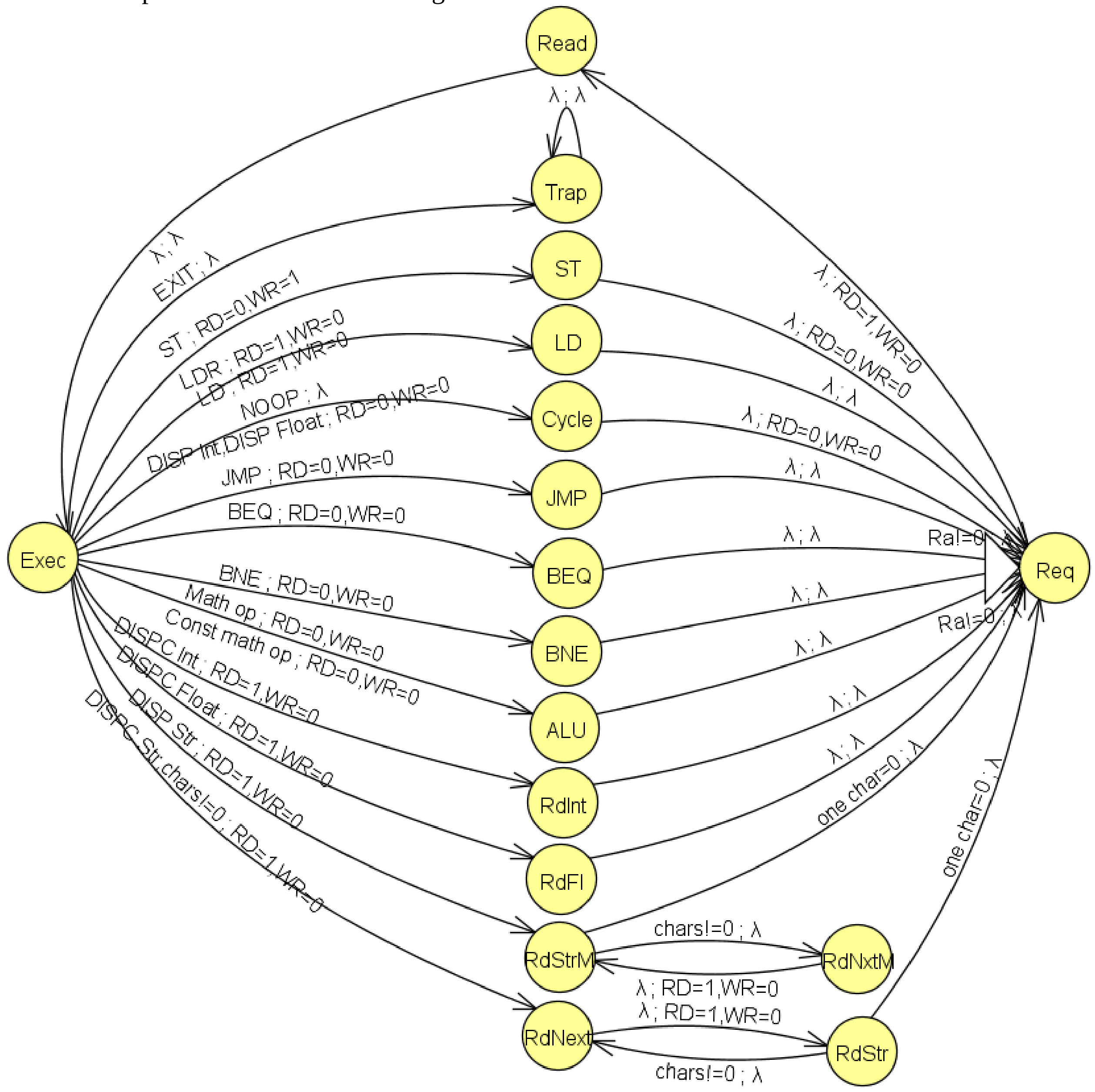

Figure 1. The Mealy machine representation of the control path.

State	Input	Output
Request Instruction		MAR=PC
Read Instruction		IR=data
Exec Instruction	IR=LD	MAR=Ra+lit

	IR=ST	MDR=Rc
		MAR=Ra+lit
		PC+=4

	IR=DISP,type=int	txbuf+=data
		PC+=4

	IR=DISP,type=float	txbuf+=data
		PC+=4

	IR=DISP,type=String	MAR=PC+4+4*lit
	IR=JMP	Rc=PC+4
	IR=BNE	Rc=PC+4
	IR=DISPC,type=int	MAR=PC+4
	IR=DISPC,type=float	MAR=PC+4
	IR=DISPC,type=String,chars!=0	txbuf+=chars

	IR=DISPC,type=String,else	txbuf+=chars
		PC+=4

	LDR	MAR=PC+4+4*lit

	IR starts with 10	Ara=Ra
		Arb=Rb
		PC+=4

	IR starts with 11	Ara=Ra
		Arb=SIGNEXT(lit)
		PC+=4

Read String		MAR+=4
Read String Memory		MAR +=4
Handle BNE	Ra=0	PC+=4
	Ra!=0	PC+=4+(4*lit)
Handle BEQ	Ra=0	PC+=4+(4*lit)
	Ra!=0	PC+=4
Handle JMP		PC=Ra

Handle LD		Rc=data
		PC+=4

Handle ALU		Rc=data
		PC+=4

Read String	chars!=0	txbuf+=chars

	else	txbuf+=chars
		PC=MAR+4

Read String Memory		txbuf+=chars

Read Int		txbuf+=data
		PC+=4

Read Float		txbuf+=data
		PC+=4

Figure 2. A table of the data path's combinational logic

We split up the FSM into a separate module that takes in the instruction register and a few other flags, determines the next state, and outputs flags to the data path. The data flow diagram can be seen below.

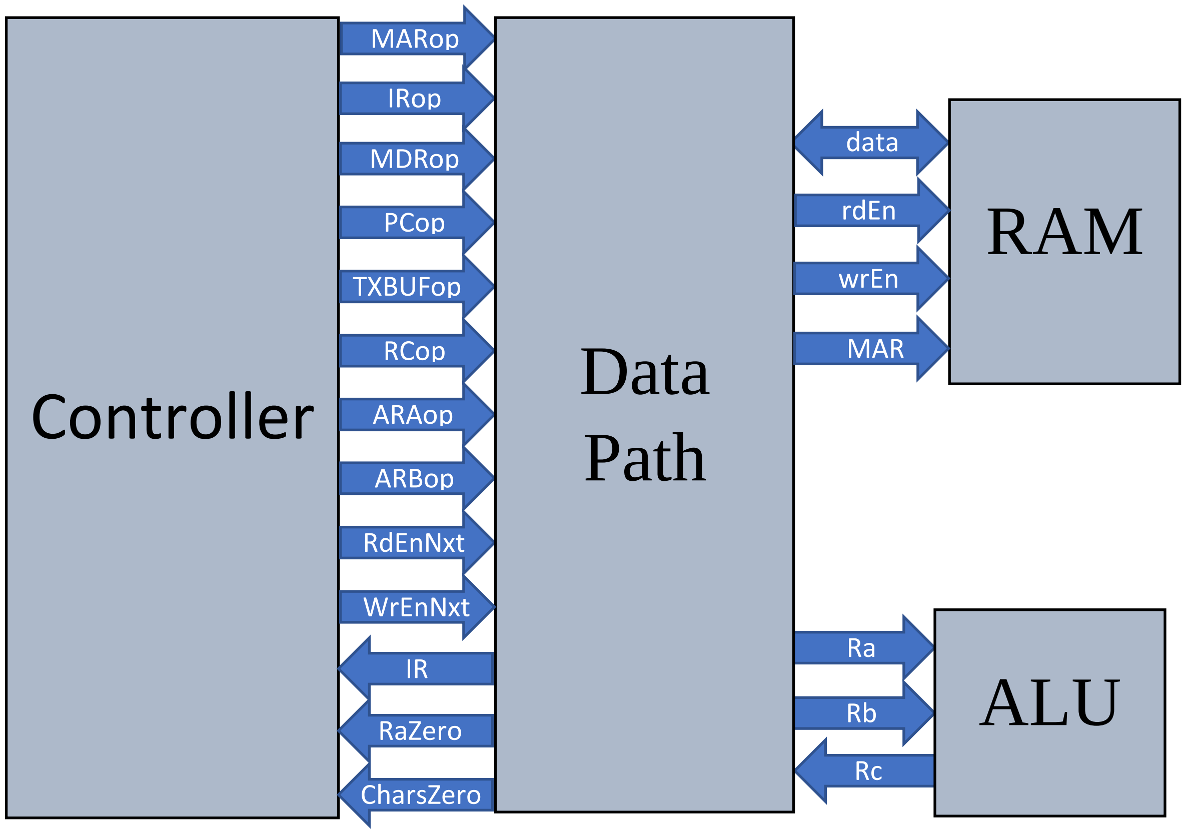

Figure 3. A diagram of the data flow within the CPU

Design Details

The ALU is written in behavioral Verilog. It acts as a combinational circuit. It reads the instruction opcode and selects which output to write to the output register based on that opcode. It only supports fixed point operations since the β instruction set only has fixed point operations. However, it could be extended to allow floating point operations if they were designed in Verilog. Verilog does not support floating point operations natively.

The RAM module simply holds 1024 32-bit words or 4KB of memory. It operates on negative edge of the clock to decrease the time required for memory operations. It loads a program on restart which the CPU begins executing immediately. The test program currently takes up 70% of the available RAM. More RAM modules could be easily added by using the Memory Controller module and modifying the memory tasks.

The controller FSM determines what state the CPU is in and sets flags that tell the data path what to do. The controller FSM needs to know if Ra is zero for branch commands and that the characters it reads from memory aren't zero to know when a null terminator was reached when printing a string.

In designing the FSM, we were careful not to modify any variables that could introduce race conditions. For instance, if the RAM module is turned off while the data path is reading, the value of that register would be unknown. States were also combined if they accomplished the same tasks. Splitting the design into a control and data path helped simplify the design since the data path has been reduced to multiplexers in essence and the controller is a simple finite state machine. Another benefit of this design is that it makes it more manageable when the number of data operations is limited.

The processor meets the standards of a RISC β microprocessor in that it implements the full instruction set and completes an instruction every four clock cycles. However, we added two instructions that do not follow these standards. We added two instructions for displaying text to the terminal, DISP and DISPC. DISP reads a number or string from memory and prints it while DISPC reads the number or string from the instruction. DISPC uses two words when printing a 32-bit integer and a variable length when printing a string. Two additional instructions were added that do follow the RISC architecture. We added NOOP and TRAP instruction. The NOOP doesn't do anything and the TRAP instruction stops the processor. We used these new instructions to help test the processor.

To test the processor, a normal testbench with a stimulus couldn't be used. The processor does not have any data inputs other than RAM, so the test had to be loaded into RAM. Since Verilog does not accept binary files and the β instruction set is not standard in any popular assembly language, a new assembler had to be made for this project. It was written in Python and based off an assembler for DCPU-16, an assembly language for a game with a working CPU emulator. We simplified the assembler into two passes: one pass to read instructions and calculate addresses, and another to translate instructions into binary. This binary file can then be loaded by the RAM module for execution.

This is the output from ModelSim's Transcript window. Each test involving data has an expected value on the left and the actual result on the right. For branching instructions, a pass or fail message was printed depending on where execution went. All the tests passed after debugging the assembler, cpu, and assembly program. The test would print an error message and exit if the test failed.

There are several extensions that could be made to this CPU. For one, floating point operations would be nice to have. Another extension would be to reimplement the ALU with structural Verilog instead of behavioral Verilog. We would have to create optimized modules for operations such as addition and multiplication.

```
# Hello world!
# Display string passed
# R0 1=1
# R1 2=2
# R2 3=3
# R3 4=4
# R4 5=5
# R5 6=6
# R6 7=7
# R7 8=8
# R8 9=9
# R9 10=10
# R10 11=11
# R11 12=12
# R12 13=13
# R13 14=14
# R14 15=15
# R15 16=16
# R16 17=17
# R17 18=18
# R18 19=19
# R19 20=20
# R20 21=21
# R21 22=22
# R22 23=23
# R23 24=24
# R24 25=25
# R25 26=26
# R26 27=27
# R27 28=28
# R28 29=29
# R29 30=30
# R30 31=31
# R31 0=0
# Passed BEQ branch
# Passed BEQ non branch
# Passed BNE branch
# Passed BNE non branch
# ST -15=-15
# Passed LD/ST
# ADD 17=17
# SUB 13=13
# MUL 30=30
# DIV 7=7
# CMPEQ 0=0
# CMPEQ 1=1
# CMPLT 0=0
# CMPLT 1=1
# CMPLE 0=0
# CMPLE 1=1
# CMPLE 1=1
# AND 2=2
# OR 15=15
# XOR 13=13
# XNOR -14=-14
# SHL 60=60
# SHR 3=3
# SRA 3=3
# SRA -4=-4
# ANDC 2=2
# ORC 15=15
# XORC 13=13
# XNORC -14=-14
# SHLC 60=60
# SHRC 3=3
# SRAC 3=3
# SRAC -4=-4
# Passed JMP
# Passed all tests
```

This is the code for this project. It can also be found at:
https://github.com/donaldsa18/BMicroprocessor

params.svh

```
parameter DWIDTH = 32;
parameter MEMDEPTH = 1024;
parameter AWIDTH = $clog2(MEMDEPTH);
parameter CPUAWIDTH = 16;
parameter NUM_REGS = 32;
```

instructionstruct.sv

```
package InstructionStruct;
`include "params.svh"

typedef enum bit [5:0] {
    LD     = 6'b011000,
    ST     = 6'b011001,
    DISP   = 6'b011110, //a new instruction for displaying text
    JMP    = 6'b011011,
    BEQ    = 6'b011100,
    BNE    = 6'b011101,
    DISPC  = 6'b011010, //a new instruction for displaying constant text
    LDR    = 6'b011111,
    ADD    = 6'b100000,
    SUB    = 6'b100001,
    MUL    = 6'b100010,
    DIV    = 6'b100011,
    CMPEQ  = 6'b100100,
    CMPLT  = 6'b100101,
    CMPLE  = 6'b100110,
    AND    = 6'b101000,
    OR     = 6'b101001,
    XOR    = 6'b101010,
    XNOR   = 6'b101011,
    SHL    = 6'b101100,
    SHR    = 6'b101101,
    SRA    = 6'b101110,
    ADDC   = 6'b110000,
    SUBC   = 6'b110001,
    MULC   = 6'b110010,
    DIVC   = 6'b110011,
    CMPEQC = 6'b110100,
    CMPLTC = 6'b110101,
    CMPLEC = 6'b110110,
    ANDC   = 6'b111000,
    ORC    = 6'b111001,
    XORC   = 6'b111010,
    XNORC  = 6'b111011,
    SHLC   = 6'b111100,
    SHRC   = 6'b111101,
    SRAC   = 6'b111110,
    EXIT   = 6'b000001,
    NOOP    = 6'b000000
} opcode_t;

typedef enum {
    request_instruction,
    read_instruction,
    exec_instruction,
    handle_ld,
    handle_ldr,
    handle_alu,
    trap,
    empty_cycle,
    read_string,
    read_int,
    read_float,
    read_string_mem,
    read_next_str_mem,
    handle_jmp,
    handle_beq,
    handle_bne,
    read_next_str
} cpu_state_t;

typedef enum bit [2:0] {
    mar_noop = 3'd0,
    mar_pc = 3'd1,
    mar_pc_incr = 3'd2,
    mar_ra = 3'd3,
    mar_jmp = 3'd4,
    mar_incr = 3'd5
} MAR_OP_t;

typedef enum bit {
    ir_noop = 1'b0,
    ir_data = 1'b1
} IR_OP_t;

typedef enum bit {
    mdr_noop = 1'b0,
    mdr_rc = 1'b1
} MDR_OP_t;

typedef enum bit [2:0] {
    pc_noop = 3'd0,
    pc_incr = 3'd1,
    pc_jmp = 3'd2,
    pc_ra = 3'd3,
    pc_mar = 3'd4,
    pc_incr2 = 3'd5
} PC_OP_t;

typedef enum bit [3:0] {
    txbuf_noop = 4'd0,
    txbuf_data_1 = 4'd1,
    txbuf_data_2 = 4'd2,
    txbuf_data_3 = 4'd3,
    txbuf_data_4 = 4'd4,
    txbuf_chars_1 = 4'd5,
    txbuf_chars_2 = 4'd6,
    txbuf_chars_3 = 4'd7,
    txbuf_err = 4'd8,
    txbuf_int = 4'd9,
    txbuf_float = 4'd10,
    txbuf_int_ra = 4'd11,
    txbuf_float_ra = 4'd12
} TXBUF_OP_t;

typedef enum bit [1:0] {
    rc_noop = 2'd0,
    rc_pc_incr = 2'd1,
    rc_data = 2'd2,
    rc_arc = 2'd3
} RC_OP_t;

typedef enum bit {
    ara_noop = 1'b0,
    ara_ra = 1'b1
} ARA_OP_t;

typedef enum bit[1:0] {
    arb_noop = 2'd0,
    arb_rb = 2'd1,
    arb_lit = 2'd2
} ARB_OP_t;

typedef enum bit {
    read_on = 1'b1,
    read_off = 1'b0
} read_t;

typedef enum bit {
    write_on = 1'b1,
    write_off = 1'b0
} write_t;

typedef struct packed {
    opcode_t opcode;
    bit [4:0] Rc;
    bit [4:0] Ra;
    bit [4:0] Rb;
    bit [10:0] unused;
} instruction_reg_t;

typedef struct packed {
    opcode_t opcode;
    bit [4:0] Rc;
    bit [4:0] Ra;
    bit [15:0] lit;
} instruction_lit_t;

typedef struct packed {
    opcode_t opcode;
    bit [1:0] datatype;
    bit [6:0] charA;
    bit [6:0] charB;
    bit [6:0] charC;
    bit [2:0] unused;
} instruction_str_t;

typedef union packed {
    instruction_reg_t regular;
    instruction_lit_t literal;
    instruction_str_t str;
    bit [31:0] bits;
} instruction_t;

endpackage : InstructionStruct
```

alu.sv

```
/*
*
* 32-bit ALU
*
* Authors: Anthony Donaldson, Matthew Erhardt
*
*/
module alu(Rc,Ra,Rb,opcode);
    import InstructionStruct::*;
    output [DWIDTH-1:0] Rc;
    reg [DWIDTH-1:0] Rc;
    input signed [DWIDTH-1:0] Ra,Rb;
    input [5:0] opcode;

    initial
        Rc = 0;

    always @(*) begin
        case(opcode)
            ADD: Rc <= Ra + Rb;
            SUB: Rc <= Ra - Rb;
            MUL: Rc <= Ra * Rb;
            DIV: Rc <= (Rb != 0) ? (Ra / Rb) : 0;
            CMPEQ: Rc <= (Ra == Rb);
            CMPLT: Rc <= (Ra < Rb);
            CMPLE: Rc <= (Ra <= Rb);
            AND: Rc <= Ra & Rb;
            OR: Rc <= Ra | Rb;
            XOR: Rc <= Ra ^ Rb;
            XNOR: Rc <= Ra ~^ Rb;
            SHL: Rc <= Ra << Rb;
            SHR: Rc <= Ra >> Rb;
            SRA: Rc <= Ra >>> Rb;
            ANDC: Rc <= Ra & Rb;
            ORC: Rc <= Ra | Rb;
            XORC: Rc <= Ra ^ Rb;
            XNORC: Rc <= Ra ~^ Rb;
            SHLC: Rc <= Ra << Rb;
            SHRC: Rc <= Ra >> Rb;
            SRAC: Rc <= Ra >>> Rb;
            CMPEQC: Rc <= (Ra == Rb);
            CMPLTC: Rc <= (Ra < Rb);
            CMPLEC: Rc <= (Ra <= Rb);
            default: Rc <= {32{1'bz}};
        endcase
    end
endmodule
```

ram.sv

```
/*
*
* RAM Module
*
* Authors: Anthony Donaldson, Matthew Erhardt
*
*/
module ram(data,addr,rdEn,wrEn,reset,clk);
import InstructionStruct::*;
inout [DWIDTH-1:0] data;
input [AWIDTH-1:0] addr;
input rdEn,wrEn,reset,clk;

tri [DWIDTH-1:0] data;

reg [DWIDTH-1:0] mem [MEMDEPTH-1:0];
assign data = (rdEn) ? mem[addr] : {DWIDTH{1'bz}};

integer i;

task reset_mem;
    for(i = 0; i < MEMDEPTH; i++)
        mem[i] = i;
    $readmemb("program.bin", mem);//, 0, MEMDEPTH-1);
endtask

initial
    reset_mem;

always @(posedge reset)
    reset_mem;

always @(negedge clk) begin
    if(wrEn && !rdEn)
        mem[addr] <= data;
end

endmodule
```

MemControl.sv

```
/*
*
* Memory Controller
*
* Authors: Anthony Donaldson, Matthew Erhardt
*
*/

module MemControl(data,addr,clk,reset,rw,valid);
import InstructionStruct::*;
inout [DWIDTH-1:0] data;
input [CPUAWIDTH-1:0] addr;
input clk,reset,rw,valid;

localparam N = 2^(CPUAWIDTH-AWIDTH);

//Translate high 4 bits into enables for each module
reg [N-1:0] rdEn,wrEn;
assign rdEn = (valid && rw) ? (1 << addr[CPUAWIDTH-1:AWIDTH+3]) : 0;
assign wrEn = (valid && !rw) ? (1 << addr[CPUAWIDTH-1:AWIDTH+3]) : 0;

//Generate RAM modules
generate
    genvar i;
    for(i = 0; i < N; i++) begin: gen_ram
        ram mod(data,addr[AWIDTH+2:2],rdEn[i],wrEn[i],reset,clk);
    end
endgenerate

endmodule
```

cpu_data.sv

```
/*
*
* B-Processor
*
* Authors: Anthony Donaldson, Matthew Erhardt
*
*/
`timescale 1ns / 1ns

module cpu_data(tx,clk,reset);

import InstructionStruct::*;

//Can print characters using this output
output [6:0] tx;
reg [6:0] tx;

//Serial output buffer
string txbuf = "";
string txbufnxt = "";

reg tx_status;
reg tx_first;

//External clock and active high reset
input clk,reset;

//registers
reg signed [DWIDTH-1:0] R [NUM_REGS-1:0];
reg signed [DWIDTH-1:0] Rnxt;

//data bus to memory controller
tri signed [DWIDTH-1:0] data;

//memory address register
reg [CPUAWIDTH-1:0] MAR;
reg [CPUAWIDTH-1:0] MARnxt;

//memory data register
reg [DWIDTH-1:0] MDR;
reg [DWIDTH-1:0] MDRnxt;

//instruction register
instruction_t IR;
instruction_t IRnxt;

//control signals for memory controller
//reg rw,valid;
//program counter
reg [CPUAWIDTH-1:0] PC;
reg [CPUAWIDTH-1:0] PCnxt;

//ALU registers
reg [DWIDTH-1:0] ARa,ARb;
reg [DWIDTH-1:0] ARanxt,ARbnxt;
wire [DWIDTH-1:0] ARc;

//for loops
integer i;

//Flags for controller
reg RaZero;
reg [3:0] charsZero;

//Flags from controller
wire [2:0] MARop;
wire IRop;
wire MDRop;
wire [2:0] PCop;
wire [3:0] TXBUFop;
wire [1:0] RCop;
wire ARAop;
wire [1:0] ARBop;
wire rdEnnxt;
wire wrEnnxt;
reg rdEn;
reg wrEn;

reg clk_del;

//Submodules - memory controller has a memory module
//ALU has registers for Ra&Rb input and wire Rc for ouput
//MemControl mc(data,MAR,clk,reset,rw,valid);
ram mod(data,MAR[AWIDTH+1:2],rdEn,wrEn,reset,clk);
alu math(ARc,ARa,ARb,IR.bits[DWIDTH-1:DWIDTH-6]);
cpu_controller
control(MARop,IRop,MDRop,PCop,TXBUFop,RCop,ARAop,ARBop,rdEnnxt,wrEnnxt,IR.bit
s,RaZero,charsZero,clk,reset);

task reset_cpu;
    MAR = 0;
    MDR = 0;
    PC = 0;
    ARa = 0;
    ARb = 0;
    IR = 0;
    MARnxt = 0;
    MDRnxt = 0;
    PCnxt = 0;
    ARanxt = 0;
    ARbnxt = 0;
    IRnxt = 0;
    tx = 0;
    tx_status = 1;
    tx_first = 0;
    for(i = 0; i < NUM_REGS; i++)
        R[i] = 0;
endtask

assign data = (wrEn == 1'b1 && rdEn == 1'b0) ? MDR : {DWIDTH{1'bz}};
```

```
initial
    reset_cpu;

always @(clk)
    clk_del <= #1 clk;

//Print each character in the buffer one by one until it is empty
//Follows UART protocol
always @(negedge clk) begin
    if(txbuf.len() > 0) begin
        //Start each message with a 0
        if(tx_status) begin
            tx = 0;
            tx_status = 0;
            tx_first = 1;
        end
        else begin
            tx = txbuf[0];
            txbuf = txbuf.substr(1,txbuf.len()-1);
        end
    end
    //End each message with all 1's
    else begin
        tx = 7'b1111111;
        tx_status = 1;
    end
end

always @(posedge reset) reset_cpu;

//Flags for control path
assign RaZero = (R[IR.regular.Ra] === 0);
assign charsZero = {(data[31:25] !== 0),(data[24:18] !== 0),(data[17:11] !==
0),(data[10:4] !== 0)};

always @(posedge clk_del) begin
    #1 rdEn = rdEnnxt;
end

always @(posedge clk_del) begin
    #1 wrEn = wrEnnxt;
end


//Data path
//Sets registers based flags
//always @(MARop,IRop,MDRop,PCop,TXBUFop,RCop,ARAop,ARBop) begin
always @(posedge clk_del) begin
    case(MARop)
        mar_noop: MARnxt = MAR;
        mar_pc: MARnxt = PC;
        mar_pc_incr: MARnxt = PC + 4;
        mar_ra: MARnxt = R[IR.literal.Ra] + IR.literal.lit;
        mar_jmp: MARnxt = PC + 4 + (4*IR.literal.lit);
        mar_incr: MARnxt = MAR + 4;
        //default: txbuf <= $sformatf("%sInvalid MARop\n",txbuf);
    endcase
    MAR = MARnxt;
end

always @(posedge clk_del) begin
    case(IRop)
        ir_noop: IRnxt = IR;
        ir_data: IRnxt = data;
        //default: txbuf <= $sformatf("%sInvalid IRop\n",txbuf);
    endcase
    IR = IRnxt;
end

always @(posedge clk_del) begin
    case(MDRop)
        mdr_noop: MDRnxt = MDR;
        mdr_rc: MDRnxt = R[IR.literal.Rc];
        //default: txbuf <= $sformatf("%sInvalid MDRop\n",txbuf);
    endcase
    MDR = MDRnxt;
end

always @(posedge clk_del) begin
    case(PCop)
        pc_noop: PCnxt = PC;
        pc_incr: PCnxt = PC + 4;
        pc_jmp: PCnxt = PC + 4 + (4*IR.literal.lit);
        pc_ra: PCnxt = R[IR.literal.Ra];
        pc_mar: PCnxt = MAR + 4;
        pc_incr2: PCnxt = PC + 8;
        //default: txbuf <= $sformatf("%sInvalid PCop\n",txbuf);
    endcase
    PC = PCnxt;
end

always @(posedge clk_del) begin
    case(TXBUFop)
        txbuf_noop: txbufnxt = txbuf;
        txbuf_data_1: txbufnxt = $sformatf("%s%c",txbuf,{1'b0,data[31:25]});
        txbuf_data_2: txbufnxt =
$sformatf("%s%c%c",txbuf,{1'b0,data[31:25]},{1'b0,data[24:18]});
        txbuf_data_3: txbufnxt =
$sformatf("%s%c%c%c",txbuf,{1'b0,data[31:25]},{1'b0,data[24:18]},{1'b0,data[1
7:11]});
        txbuf_data_4: txbufnxt =
$sformatf("%s%c%c%c%c",txbuf,{1'b0,data[31:25]},{1'b0,data[24:18]},{1'b0,data
[17:11]},{1'b0,data[10:4]});
        txbuf_chars_1: txbufnxt = $sformatf("%s%c",txbuf,IR.str.charA);
        txbuf_chars_2: txbufnxt =
$sformatf("%s%c%c",txbuf,IR.str.charA,IR.str.charB);
        txbuf_chars_3: txbufnxt =
$sformatf("%s%c%c%c",txbuf,IR.str.charA,IR.str.charB,IR.str.charC);
        txbuf_err: txbufnxt = $sformatf("%sInvalid instruction opcode=%h
PC=%d\n",txbuf,IR.regular.opcode,PC);
        txbuf_int: txbufnxt = $sformatf("%s%0d",txbuf,data);
        txbuf_float: txbufnxt = $sformatf("%s%0f",txbuf,data);
        txbuf_int_ra: txbufnxt = $sformatf("%s%0d",txbuf,R[IR.literal.Ra]);
        txbuf_float_ra: txbufnxt = $sformatf("%s%0f",txbuf,R[IR.literal.Ra]);
        //default: txbuf <= $sformatf("%sInvalid TXBUFop\n",txbuf);
    endcase
    txbuf = txbufnxt;
end

always @(posedge clk_del) begin
    case(RCop)
        rc_noop: Rnxt = R[IR.literal.Rc];
        rc_pc_incr: Rnxt = PC + 4;
        rc_data: Rnxt = data;
        rc_arc: Rnxt = (IR.regular.Rc != 5'd31) ? ARc : 0;
        //default: txbuf <= $sformatf("%sInvalid RCop\n",txbuf);
    endcase
    R[IR.literal.Rc] = Rnxt;
end

always @(posedge clk_del) begin
    case(ARAop)
        ara_noop: ARanxt = ARa;
        ara_ra: ARanxt = R[IR.regular.Ra];
        //default: txbuf <= $sformatf("%sInvalid ARAop\n",txbuf);
    endcase
    ARa = ARanxt;
end

always @(posedge clk_del) begin
    case(ARBop)
        arb_noop: ARbnxt = ARb;
        arb_rb: ARbnxt = R[IR.regular.Rb];
        arb_lit: ARbnxt = {{8{IR.literal.lit[15]}},IR.literal.lit };
    endcase
    ARb = ARbnxt;
end

endmodule
```

```
/*
*
* B-Processor Testbench
*
* Authors: Anthony Donaldson, Matthew Erhardt
*
*/
`timescale 1ns / 1ns

module cpu_data_tb;

reg clk, reset;
wire [6:0] tx;
reg startTx;
string txbuf = "";
cpu_data c(tx,clk,reset);

initial begin
    clk = 1'b0;
    forever #5 clk = !clk;
end

initial begin
    startTx = 0;
    reset = 0;
    #1 reset = 1;
    #1 reset = 0;
    #16500 $stop;
end

always @(posedge clk) begin
    if(startTx && tx != 7'b1111111 && tx != 0) begin
        $write("%c", tx);
        txbuf <= $sformatf("%s%c",txbuf,tx);
    end
    if(tx == 0)
        startTx = 1;
    else if(tx == 7'b1111111)
        startTx = 0;
end
endmodule
```

MemControl_tb.sv

```
/*
*
* Memory Controller Testbench
*
* Authors: Anthony Donaldson, Matthew Erhardt
*
*/
module MemControl_tb;

import InstructionStruct::*;

tri [DWIDTH-1:0] data;
reg [DWIDTH-1:0] datareg;

reg clk,reset,rw,valid;
reg [CPUAWIDTH-1:0] addr;

MemControl mc(data,addr,clk,reset,rw,valid);

initial begin
    clk = 1;
    forever
        #5 clk = ~clk;
end

assign data = (!rw && valid) ? datareg : {DWIDTH{1'bz}};

initial begin
    rw = 1;
    valid = 0;
    addr = 0;
    datareg = 0;
    reset = 1;
    #2 reset = 0;
    #5 valid = 1;
    #20 addr = 4;
    #20 addr = 8;
    #20 rw = 0;
        addr = 0;
        datareg = $random;
    #20 addr = 4;
        datareg = $random;
    #20 addr = 8;
        datareg = $random;
    #20 rw = 1;
        addr = 0;
    #20 rw = 1;
        addr = 4;
    #20 rw = 1;
        addr = 8;
    #20 $stop;
end

endmodule
```

ram_tb.sv

```
/*
*
* RAM Module Testbench
*
* Authors: Anthony Donaldson, Matthew Erhardt
*
*/
module ram_tb;
import InstructionStruct::*;
tri [DWIDTH-1:0] data;
reg [DWIDTH-1:0] datareg;
reg rdEn,wrEn,reset,clk;
reg [AWIDTH-1:0] addr;
ram r0(data,addr,rdEn,wrEn,reset,clk);

initial begin
    clk = 1;
    forever
        #5 clk = ~clk;
end

assign data = (wrEn) ? datareg : {DWIDTH{1'bz}};

initial begin
    rdEn = 0;
    wrEn = 0;
    reset = 1;
    addr = 0;
    #2 reset = 0;
    #5 rdEn = 1;
    #10 addr = 1;
    #10 addr = 2;
    #10 rdEn = 0;
        wrEn = 1;
        datareg = $random;
        addr = 0;
    #10 datareg = $random;
        addr = 1;
    #10 datareg = $random;
        addr = 2;
    #10 wrEn = 0;
        rdEn = 1;
        addr = 0;
    #10 addr = 1;
    #10 addr = 2;
    #10 $stop;

end

endmodule
```

program.asm

```
BEQ R1,start_label,R2;
DISPC "\nPassed JMP\nPassed all tests\n";
trap;

start_label:

DISPC "Hello world!";

; Test string printing

DISP label_str,STR;

; Test all registers
ORC R31,1,R0;
DISPC "\nR0 1=";
DISP R0,int;
CMPEQC R0,1,R1;
BEQ R1,failedReg,R3;

ORC R31,2,R1;
DISPC "\nR1 2=";
DISP R1,int;
CMPEQC R1,2,R0;
BEQ R0,failedReg,R3;

ORC R31,3,R2;
DISPC "\nR2 3=";
DISP R2,int;
CMPEQC R2,3,R0;
BEQ R0,failedReg,R3;

ORC R31,4,R3;
DISPC "\nR3 4=";
DISP R3,int;
CMPEQC R3,4,R0;
BEQ R0,failedReg,R3;

ORC R31,5,R4;
DISPC "\nR4 5=";
DISP R4,int;
CMPEQC R4,5,R0;
BEQ R0,failedReg,R3;

ORC R31,6,R5;
DISPC "\nR5 6=";
DISP R5,int;
CMPEQC R5,6,R0;
BEQ R0,failedReg,R3;

ORC R31,7,R6;
DISPC "\nR6 7=";
DISP R6,int;
CMPEQC R6,7,R0;
BEQ R0,failedReg,R3;
```

```
ORC R31,8,R7;
DISPC "\nR7 8=";
DISP R7,int;
CMPEQC R7,8,R0;
BEQ R0,failedReg,R3;

ORC R31,9,R8;
DISPC "\nR8 9=";
DISP R8,int;
CMPEQC R8,9,R0;
BEQ R0,failedReg,R3;

ORC R31,10,R9;
DISPC "\nR9 10=";
DISP R9,int;
CMPEQC R9,10,R0;
BEQ R0,failedReg,R3;

ORC R31,11,R10;
DISPC "\nR10 11=";
DISP R10,int;
CMPEQC R10,11,R0;
BEQ R0,failedReg,R3;

ORC R31,12,R11;
DISPC "\nR11 12=";
DISP R11,int;
CMPEQC R11,12,R0;
BEQ R0,failedReg,R3;

ORC R31,13,R12;
DISPC "\nR12 13=";
DISP R12,int;
CMPEQC R12,13,R0;
BEQ R0,failedReg,R3;

ORC R31,14,R13;
DISPC "\nR13 14=";
DISP R13,int;
CMPEQC R13,14,R0;
BEQ R0,failedReg,R3;

ORC R31,15,R14;
DISPC "\nR14 15=";
DISP R14,int;
CMPEQC R14,15,R0;
BEQ R0,failedReg,R3;

ORC R31,16,R15;
DISPC "\nR15 16=";
DISP R15,int;
CMPEQC R15,16,R0;
BEQ R0,failedReg,R3;

ORC R31,17,R16;
DISPC "\nR16 17=";
```

```
DISP R16,int;
CMPEQC R16,17,R0;
BEQ R0,failedReg,R3;

ORC R31,18,R17;
DISPC "\nR17 18=";
DISP R17,int;
CMPEQC R17,18,R0;
BEQ R0,failedReg,R3;

ORC R31,19,R18;
DISPC "\nR18 19=";
DISP R18,int;
CMPEQC R18,19,R0;
BEQ R0,failedReg,R3;

ORC R31,20,R19;
DISPC "\nR19 20=";
DISP R19,int;
CMPEQC R19,20,R0;
BEQ R0,failedReg,R3;

ORC R31,21,R20;
DISPC "\nR20 21=";
DISP R20,int;
CMPEQC R20,21,R0;
BEQ R0,failedReg,R3;

ORC R31,22,R21;
DISPC "\nR21 22=";
DISP R21,int;
CMPEQC R21,22,R0;
BEQ R0,failedReg,R3;

ORC R31,23,R22;
DISPC "\nR22 23=";
DISP R22,int;
CMPEQC R22,23,R0;
BEQ R0,failedReg,R3;

ORC R31,24,R23;
DISPC "\nR23 24=";
DISP R23,int;
CMPEQC R23,24,R0;
BEQ R0,failedReg,R3;

ORC R31,25,R24;
DISPC "\nR24 25=";
DISP R24,int;
CMPEQC R24,25,R0;
BEQ R0,failedReg,R3;

ORC R31,26,R25;
DISPC "\nR25 26=";
DISP R25,int;
CMPEQC R25,26,R0;
```

```
BEQ R0,failedReg,R3;

ORC R31,27,R26;
DISPC "\nR26 27=";
DISP R26,int;
CMPEQC R26,27,R0;
BEQ R0,failedReg,R3;

ORC R31,28,R27;
DISPC "\nR27 28=";
DISP R27,int;
CMPEQC R27,28,R0;
BEQ R0,failedReg,R3;

ORC R31,29,R28;
DISPC "\nR28 29=";
DISP R28,int;
CMPEQC R28,29,R0;
BEQ R0,failedReg,R3;

ORC R31,30,R29;
DISPC "\nR29 30=";
DISP R29,int;
CMPEQC R29,30,R0;
BEQ R0,failedReg,R3;

ORC R31,31,R30;
DISPC "\nR30 31=";
DISP R30,int;
CMPEQC R30,31,R0;
BEQ R0,failedReg,R3;

ORC R31,32,R31;
DISPC "\nR31 0=";
DISP R31,int;
BNE R31,failedReg,R3;

; Test BEQ/BNE

BEQ R31,continue_beq,R3;
dispc "\nFailed BEQ";
trap;

continue_beq:
dispc "\nPassed BEQ branch";
BEQ R0,failedBEQ,R3;
dispc "\nPassed BEQ non branch";
BNE R0,continue_bne,R3;
dispc "\nFailed BNE";
trap;

continue_bne:
dispc "\nPassed BNE branch";
BNE R31,failedBNE,R3;
dispc "\nPassed BNE non branch";
```

```
LDR label_neg,R0;

; Test storing/loading a word into memory

DISPC "\nST -15=";
ST R0,4000,R31;
LD R31,4000,R1;
DISP R1,int;
CMPEQ R0,R1,R4;
BEQ R4,failedST,R3;
dispc "\nPassed LD/ST";

; Test all math operations
LDR label_a,R0;
LDR label_b,R1;

DISPC "\nADD 17=";
ADD R0,R1,R2;
DISP R2,int;
LDR label_add,R3;
CMPEQ R2,R3,R4
BEQ R4,failedADD,R3;

DISPC "\nSUB 13=";
SUB R0,R1,R2;
DISP R2,int;
LDR label_sub,R3;
CMPEQ R2,R3,R4
BEQ R4,failedSUB,R3;

DISPC "\nMUL 30=";
MUL R0,R1,R2;
DISP R2,int;
LDR label_mul,R3;
CMPEQ R2,R3,R4
BEQ R4,failedMUL,R3;

DISPC "\nDIV 7=";
DIV R0,R1,R2;
DISP R2,int;
LDR label_div,R3;
CMPEQ R2,R3,R4
BEQ R4,failedDIV,R3;

DISPC "\nCMPEQ 0=";
CMPEQ R0,R1,R2;
DISP R2,int;
LDR label_0,R3;
CMPEQ R2,R3,R4
BEQ R4,failedCMPEQ,R3;

DISPC "\nCMPEQ 1=";
CMPEQ R1,R1,R2;
DISP R2,int;
LDR label_1,R3;
CMPEQ R2,R3,R4
```

```
BEQ R4,failedCMPEQ,R3;

DISPC "\nCMPLT 0=";
CMPLT R0,R1,R2;
DISP R2,int;
LDR label_0,R3;
CMPEQ R2,R3,R4
BEQ R4,failedCMPLT,R3;

DISPC "\nCMPLT 1=";
CMPLT R1,R0,R2;
DISP R2,int;
LDR label_1,R3;
CMPEQ R2,R3,R4
BEQ R4,failedCMPLT,R3;

DISPC "\nCMPLE 0=";
CMPLE R0,R1,R2;
DISP R2,int;
LDR label_0,R3;
CMPEQ R2,R3,R4
BEQ R4,failedCMPLE,R3;

DISPC "\nCMPLE 1=";
CMPLE R0,R0,R2;
DISP R2,int;
LDR label_1,R3;
CMPEQ R2,R3,R4
BEQ R4,failedCMPLE,R3;

DISPC "\nCMPLE 1=";
CMPLE R1,R0,R2;
DISP R2,int;
LDR label_1,R3;
CMPEQ R2,R3,R4
BEQ R4,failedCMPLE,R3;

DISPC "\nAND 2=";
AND R0,R1,R2;
DISP R2,int;
LDR label_and,R3;
CMPEQ R2,R3,R4
BEQ R4,failedAND,R3;

DISPC "\nOR 15=";
OR R0,R1,R2;
DISP R2,int;
LDR label_or,R3;
CMPEQ R2,R3,R4
BEQ R4,failedOR,R3;

DISPC "\nXOR 13=";
XOR R0,R1,R2;
DISP R2,int;
LDR label_xor,R3;
CMPEQ R2,R3,R4
```

```
BEQ R4,failedXOR,R3;

DISPC "\nXNOR -14=";
XNOR R0,R1,R2;
DISP R2,int;
LDR label_xnor,R3;
CMPEQ R2,R3,R4
BEQ R4,failedXNOR,R3;

DISPC "\nSHL 60=";
SHL R0,R1,R2;
DISP R2,int;
LDR label_shl,R3;
CMPEQ R2,R3,R4
BEQ R4,failedSHL,R3;

DISPC "\nSHR 3=";
SHR R0,R1,R2;
DISP R2,int;
LDR label_shr,R3;
CMPEQ R2,R3,R4
BEQ R4,failedSHR,R3;

DISPC "\nSRA 3=";
SRA R0,R1,R2;
DISP R2,int;
LDR label_shr,R3;
CMPEQ R2,R3,R4
BEQ R4,failedSRA,R3;

LDR label_neg,R0;

DISPC "\nSRA -4=";
SRA R0,R1,R2;
DISP R2,int;
LDR label_sra_neg,R3;
CMPEQ R2,R3,R4
BEQ R4,failedSRA,R3;

LDR label_a,R0;

;Test all constant math operations

DISPC "\nANDC 2=";
ANDC R0,2,R2
DISP R2,int;
LDR label_and,R3;
CMPEQ R2,R3,R4
BEQ R4,failedANDC,R3;

DISPC "\nORC 15=";
ORC R0,2,R2
DISP R2,int;
LDR label_or,R3;
CMPEQ R2,R3,R4
BEQ R4,failedORC,R3;
```

```
DISPC "\nXORC 13=";
XORC R0,2,R2;
DISP R2,int;
LDR label_xor,R3;
CMPEQ R2,R3,R4
BEQ R4,failedXORC,R3;

DISPC "\nXNORC -14=";
XNORC R0,2,R2;
DISP R2,int;
LDR label_xnor,R3;
CMPEQ R2,R3,R4
BEQ R4,failedXNORC,R3;

DISPC "\nSHLC 60=";
SHLC R0,2,R2;
DISP R2,int;
LDR label_shl,R3;
CMPEQ R2,R3,R4
BEQ R4,failedSHLC,R3;

DISPC "\nSHRC 3=";
SHRC R0,2,R2;
DISP R2,int;
LDR label_shr,R3;
CMPEQ R2,R3,R4
BEQ R4,failedSHRC,R3;

LDR label_a,R0;

DISPC "\nSRAC 3=";
SRAC R0,2,R2;
DISP R2,int;
LDR label_sra,R3;
CMPEQ R2,R3,R4
BEQ R4,failedSRAC,R3;

LDR label_neg,R0;

DISPC "\nSRAC -4=";
SRAC R0,2,R2;
DISP R2,int;
LDR label_sra_neg,R3;
CMPEQ R2,R3,R4;
BEQ R4,failedSRAC,R3;

; Test JMP

LDR label_1,R1;
JMP R31,R0;

trap;

failedReg: DISPC "\nFailed Register\n";
trap;
```

```
failedLD: DISPC "\nFailed LD\n";
trap;
failedST: DISPC "\nFailed ST\n";
trap;
failedJMP: DISPC "\nFailed JMP\n";
trap;
faileDBEQ: DISPC "\nFailed BEQ\n";
trap;
faileDBNE: DISPC "\nFailed BNE\n";
trap;
failedLDR: DISPC "\nFailed LDR\n";
trap;
failedADD: DISPC "\nFailed ADD\n";
trap;
failedSUB: DISPC "\nFailed SUB\n";
trap;
failedMUL: DISPC "\nFailed MUL\n";
trap;
failedDIV: DISPC "\nFailed DIV\n";
trap;
failedCMPEQ: DISPC "\nFailed CMPEQ\n";
trap;
failedCMPLT: DISPC "\nFailed CMPLT\n";
trap;
failedCMPLE: DISPC "\nFailed CMPLE\n";
trap;
failedAND: DISPC "\nFailed AND\n";
trap;
failedOR: DISPC "\nFailed OR\n";
trap;
failedXOR: DISPC "\nFailed XOR\n";
trap;
failedXNOR: DISPC "\nFailed XNOR\n";
trap;
failedSHL: DISPC "\nFailed SHL\n";
trap;
failedSHR: DISPC "\nFailed SHR\n";
trap;
failedSRA: DISPC "\nFailed SRA\n";
trap;
failedADDC: DISPC "\nFailed ADDC\n";
trap;
failedSUBC: DISPC "\nFailed SUBC\n";
trap;
failedMULC: DISPC "\nFailed MULC\n";
trap;
failedDIVC: DISPC "\nFailed DIVC\n";
trap;
failedCMPEQC: DISPC "\nFailed CMPEQC\n";
trap;
failedCMPLTC: DISPC "\nFailed CMPLTC\n";
trap;
failedCMPLEC: DISPC "\nFailed CMPLEC\n";
trap;
failedANDC: DISPC "\nFailed ANDC\n";
trap;
```

```
failedORC: DISPC "\nFailed ORC\n";
trap;
failedXORC: DISPC "\nFailed XORC\n";
trap;
failedXNORC: DISPC "\nFailed XNORC\n";
trap;
failedSHLC: DISPC "\nFailed SHLC\n";
trap;
failedSHRC: DISPC "\nFailed SHRC\n";
trap;
failedSRAC: DISPC "\nFailed SRAC\n";
trap;
failedST: DISPC "\nFailed ST\n";
trap;
failedBNE: DISPC "\nFailed BNE\n";
trap;
failedBEQ: DISPC "\nFailed BEQ\n";
trap;

label_a: DB 15;
label_b: DB 2;
label_neg: DB -15;
label_or: DB 15;
label_and: DB 2;
label_add: DB 17;
label_sub: DB 13;
label_mul: DB 30;
label_div: DB 7;
label_xor: DB 13;
label_shl: DB 60;
label_shr: DB 3;
label_sra: DB 3;
label_sra_neg: DB -4;
label_xnor: DB -14;
label_0: DB 0;
label_1: DB 1;
label_str: DB "\nDisplay string passed"
```

program.bin

```
011100_00010_00001_0000000000001001 //BEQ R1,start_label,R2 line: 0
011010_11_0001010_1010000_1100001_000 //START: DISPC "\nPassed JMP\nPassed
all tests\n"
1110011_1110011_1100101_1100100_0000
0100000_1001010_1001101_1010000_0000
0001010_1010000_1100001_1110011_0000
1110011_1100101_1100100_0100000_0000
1100001_1101100_1101100_0100000_0000
1110100_1100101_1110011_1110100_0000
1110011_0001010_0000000__00000000000 //END: DISPC "\nPassed JMP\nPassed all
tests\n"
000001_00000_00000_0000000000000000 //trap line: 9
011010_11_1001000_1100101_1101100_000 //START: DISPC "Hello world!"
1101100_1101111_0100000_1110111_0000
1101111_1110010_1101100_1100100_0000
0100001_0000000__000000000000000000 //END: DISPC "Hello world!"
011110_00011_00000_0000001010111100 //DISP label_str,STR line: 14
111001_00000_11111_0000000000000001 //ORC R31,1,R0 line: 15
011010_11_0001010_1010010_0110000_000 //START: DISPC "\nR0 1="
0100000_0110001_0111101_0000000_0000 //END: DISPC "\nR0 1="
011110_00001_00000_0000000000000000 //DISP R0,int line: 18
110100_00001_00000_0000000000000001 //CMPEQC R0,1,R1 line: 19
011100_00011_00001_0000000111100011 //BEQ R1,failedReg,R3 line: 20
111001_00001_11111_0000000000000010 //ORC R31,2,R1 line: 21
011010_11_0001010_1010010_0110001_000 //START: DISPC "\nR1 2="
0100000_0110010_0111101_0000000_0000 //END: DISPC "\nR1 2="
011110_00001_00001_0000000000000000 //DISP R1,int line: 24
110100_00000_00001_0000000000000010 //CMPEQC R1,2,R0 line: 25
011100_00011_00000_0000000111011101 //BEQ R0,failedReg,R3 line: 26
111001_00010_11111_0000000000000011 //ORC R31,3,R2 line: 27
011010_11_0001010_1010010_0110010_000 //START: DISPC "\nR2 3="
0100000_0110011_0111101_0000000_0000 //END: DISPC "\nR2 3="
011110_00001_00010_0000000000000000 //DISP R2,int line: 30
110100_00000_00010_0000000000000011 //CMPEQC R2,3,R0 line: 31
011100_00011_00000_0000000111010111 //BEQ R0,failedReg,R3 line: 32
111001_00011_11111_0000000000000100 //ORC R31,4,R3 line: 33
011010_11_0001010_1010010_0110011_000 //START: DISPC "\nR3 4="
0100000_0110100_0111101_0000000_0000 //END: DISPC "\nR3 4="
011110_00001_00011_0000000000000000 //DISP R3,int line: 36
110100_00000_00011_0000000000000100 //CMPEQC R3,4,R0 line: 37
011100_00011_00000_0000000111010001 //BEQ R0,failedReg,R3 line: 38
111001_00100_11111_0000000000000101 //ORC R31,5,R4 line: 39
011010_11_0001010_1010010_0110100_000 //START: DISPC "\nR4 5="
0100000_0110101_0111101_0000000_0000 //END: DISPC "\nR4 5="
011110_00001_00100_0000000000000000 //DISP R4,int line: 42
110100_00000_00100_0000000000000101 //CMPEQC R4,5,R0 line: 43
011100_00011_00000_0000000111001011 //BEQ R0,failedReg,R3 line: 44
111001_00101_11111_0000000000000110 //ORC R31,6,R5 line: 45
011010_11_0001010_1010010_0110101_000 //START: DISPC "\nR5 6="
0100000_0110110_0111101_0000000_0000 //END: DISPC "\nR5 6="
011110_00001_00101_0000000000000000 //DISP R5,int line: 48
110100_00000_00101_0000000000000110 //CMPEQC R5,6,R0 line: 49
011100_00011_00000_0000000111000101 //BEQ R0,failedReg,R3 line: 50
111001_00110_11111_0000000000000111 //ORC R31,7,R6 line: 51
011010_11_0001010_1010010_0110110_000 //START: DISPC "\nR6 7="
```

```
0100000_0110111_0111101_0000000_0000 //END: DISPC "\nR6 7="
011110_00001_00110_0000000000000000 //DISP R6,int line: 54
110100_00000_00110_0000000000000111 //CMPEQC R6,7,R0 line: 55
011100_00011_00000_0000000110111111 //BEQ R0,failedReg,R3 line: 56
111001_00111_11111_0000000000001000 //ORC R31,8,R7 line: 57
011010_11_0001010_1010010_0110111_000 //START: DISPC "\nR7 8="
0100000_0111000_0111101_0000000_0000 //END: DISPC "\nR7 8="
011110_00001_00111_0000000000000000 //DISP R7,int line: 60
110100_00000_00111_0000000000001000 //CMPEQC R7,8,R0 line: 61
011100_00011_00000_0000000110111001 //BEQ R0,failedReg,R3 line: 62
111001_01000_11111_0000000000001001 //ORC R31,9,R8 line: 63
011010_11_0001010_1010010_0111000_000 //START: DISPC "\nR8 9="
0100000_0111001_0111101_0000000_0000 //END: DISPC "\nR8 9="
011110_00001_01000_0000000000000000 //DISP R8,int line: 66
110100_00000_01000_0000000000001001 //CMPEQC R8,9,R0 line: 67
011100_00011_00000_0000000110110011 //BEQ R0,failedReg,R3 line: 68
111001_01001_11111_0000000000001010 //ORC R31,10,R9 line: 69
011010_11_0001010_1010010_0111001_000 //START: DISPC "\nR9 10="
0100000_0110001_0110000_0111101_0000
0000000__0000000000000000000000000 //END: DISPC "\nR9 10="
011110_00001_01001_0000000000000000 //DISP R9,int line: 73
110100_00000_01001_0000000000001010 //CMPEQC R9,10,R0 line: 74
011100_00011_00000_0000000110101100 //BEQ R0,failedReg,R3 line: 75
111001_01010_11111_0000000000001011 //ORC R31,11,R10 line: 76
011010_11_0001010_1010010_0110001_000 //START: DISPC "\nR10 11="
0110000_0100000_0110001_0110001_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR10 11="
011110_00001_01010_0000000000000000 //DISP R10,int line: 80
110100_00000_01010_0000000000001011 //CMPEQC R10,11,R0 line: 81
011100_00011_00000_0000000110100101 //BEQ R0,failedReg,R3 line: 82
111001_01011_11111_0000000000001100 //ORC R31,12,R11 line: 83
011010_11_0001010_1010010_0110001_000 //START: DISPC "\nR11 12="
0110001_0100000_0110001_0110010_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR11 12="
011110_00001_01011_0000000000000000 //DISP R11,int line: 87
110100_00000_01011_0000000000001100 //CMPEQC R11,12,R0 line: 88
011100_00011_00000_0000000110011110 //BEQ R0,failedReg,R3 line: 89
111001_01100_11111_0000000000001101 //ORC R31,13,R12 line: 90
011010_11_0001010_1010010_0110001_000 //START: DISPC "\nR12 13="
0110010_0100000_0110001_0110011_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR12 13="
011110_00001_01100_0000000000000000 //DISP R12,int line: 94
110100_00000_01100_0000000000001101 //CMPEQC R12,13,R0 line: 95
011100_00011_00000_0000000110010111 //BEQ R0,failedReg,R3 line: 96
111001_01101_11111_0000000000001110 //ORC R31,14,R13 line: 97
011010_11_0001010_1010010_0110001_000 //START: DISPC "\nR13 14="
0110011_0100000_0110001_0110100_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR13 14="
011110_00001_01101_0000000000000000 //DISP R13,int line: 101
110100_00000_01101_0000000000001110 //CMPEQC R13,14,R0 line: 102
011100_00011_00000_0000000110010000 //BEQ R0,failedReg,R3 line: 103
111001_01110_11111_0000000000001111 //ORC R31,15,R14 line: 104
011010_11_0001010_1010010_0110001_000 //START: DISPC "\nR14 15="
0110100_0100000_0110001_0110101_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR14 15="
011110_00001_01110_0000000000000000 //DISP R14,int line: 108
```

```
110100_00000_01110_0000000000001111 //CMPEQC R14,15,R0 line: 109
011100_00011_00000_0000000110001001 //BEQ R0,failedReg,R3 line: 110
111001_01111_11111_0000000000010000 //ORC R31,16,R15 line: 111
011010_11_0001010_1010010_0110001_000 //START: DISPC "\nR15 16="
0110101_0100000_0110001_0110110_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR15 16="
011110_00001_01111_0000000000000000 //DISP R15,int line: 115
110100_00000_01111_0000000000010000 //CMPEQC R15,16,R0 line: 116
011100_00011_00000_0000000110000010 //BEQ R0,failedReg,R3 line: 117
111001_10000_11111_0000000000010001 //ORC R31,17,R16 line: 118
011010_11_0001010_1010010_0110001_000 //START: DISPC "\nR16 17="
0110110_0100000_0110001_0110111_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR16 17="
011110_00001_10000_0000000000000000 //DISP R16,int line: 122
110100_00000_10000_0000000000010001 //CMPEQC R16,17,R0 line: 123
011100_00011_00000_0000000101111011 //BEQ R0,failedReg,R3 line: 124
111001_10001_11111_0000000000010010 //ORC R31,18,R17 line: 125
011010_11_0001010_1010010_0110001_000 //START: DISPC "\nR17 18="
0110111_0100000_0110001_0111000_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR17 18="
011110_00001_10001_0000000000000000 //DISP R17,int line: 129
110100_00000_10001_0000000000010010 //CMPEQC R17,18,R0 line: 130
011100_00011_00000_0000000101110100 //BEQ R0,failedReg,R3 line: 131
111001_10010_11111_0000000000010011 //ORC R31,19,R18 line: 132
011010_11_0001010_1010010_0110001_000 //START: DISPC "\nR18 19="
0111000_0100000_0110001_0111001_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR18 19="
011110_00001_10010_0000000000000000 //DISP R18,int line: 136
110100_00000_10010_0000000000010011 //CMPEQC R18,19,R0 line: 137
011100_00011_00000_0000000101101101 //BEQ R0,failedReg,R3 line: 138
111001_10011_11111_0000000000010100 //ORC R31,20,R19 line: 139
011010_11_0001010_1010010_0110001_000 //START: DISPC "\nR19 20="
0111001_0100000_0110010_0110000_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR19 20="
011110_00001_10011_0000000000000000 //DISP R19,int line: 143
110100_00000_10011_0000000000010100 //CMPEQC R19,20,R0 line: 144
011100_00011_00000_0000000101100110 //BEQ R0,failedReg,R3 line: 145
111001_10100_11111_0000000000010101 //ORC R31,21,R20 line: 146
011010_11_0001010_1010010_0110010_000 //START: DISPC "\nR20 21="
0110000_0100000_0110010_0110001_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR20 21="
011110_00001_10100_0000000000000000 //DISP R20,int line: 150
110100_00000_10100_0000000000010101 //CMPEQC R20,21,R0 line: 151
011100_00011_00000_0000000101011111 //BEQ R0,failedReg,R3 line: 152
111001_10101_11111_0000000000010110 //ORC R31,22,R21 line: 153
011010_11_0001010_1010010_0110010_000 //START: DISPC "\nR21 22="
0110001_0100000_0110010_0110010_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR21 22="
011110_00001_10101_0000000000000000 //DISP R21,int line: 157
110100_00000_10101_0000000000010110 //CMPEQC R21,22,R0 line: 158
011100_00011_00000_0000000101011000 //BEQ R0,failedReg,R3 line: 159
111001_10110_11111_0000000000010111 //ORC R31,23,R22 line: 160
011010_11_0001010_1010010_0110010_000 //START: DISPC "\nR22 23="
0110010_0100000_0110010_0110011_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR22 23="
011110_00001_10110_0000000000000000 //DISP R22,int line: 164
```

```
110100_00000_10110_0000000000010111 //CMPEQC R22,23,R0 line: 165
011100_00011_00000_0000000101010001 //BEQ R0,failedReg,R3 line: 166
111001_10111_11111_0000000000011000 //ORC R31,24,R23 line: 167
011010_11_0001010_1010010_0110010_000 //START: DISPC "\nR23 24="
0110011_0100000_0110010_0110100_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR23 24="
011110_00001_10111_0000000000000000 //DISP R23,int line: 171
110100_00000_10111_0000000000011000 //CMPEQC R23,24,R0 line: 172
011100_00011_00000_0000000101001010 //BEQ R0,failedReg,R3 line: 173
111001_11000_11111_0000000000011001 //ORC R31,25,R24 line: 174
011010_11_0001010_1010010_0110010_000 //START: DISPC "\nR24 25="
0110100_0100000_0110010_0110101_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR24 25="
011110_00001_11000_0000000000000000 //DISP R24,int line: 178
110100_00000_11000_0000000000011001 //CMPEQC R24,25,R0 line: 179
011100_00011_00000_0000000101000011 //BEQ R0,failedReg,R3 line: 180
111001_11001_11111_0000000000011010 //ORC R31,26,R25 line: 181
011010_11_0001010_1010010_0110010_000 //START: DISPC "\nR25 26="
0110101_0100000_0110010_0110110_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR25 26="
011110_00001_11001_0000000000000000 //DISP R25,int line: 185
110100_00000_11001_0000000000011010 //CMPEQC R25,26,R0 line: 186
011100_00011_00000_0000000100111100 //BEQ R0,failedReg,R3 line: 187
111001_11010_11111_0000000000011011 //ORC R31,27,R26 line: 188
011010_11_0001010_1010010_0110010_000 //START: DISPC "\nR26 27="
0110110_0100000_0110010_0110111_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR26 27="
011110_00001_11010_0000000000000000 //DISP R26,int line: 192
110100_00000_11010_0000000000011011 //CMPEQC R26,27,R0 line: 193
011100_00011_00000_0000000100110101 //BEQ R0,failedReg,R3 line: 194
111001_11011_11111_0000000000011100 //ORC R31,28,R27 line: 195
011010_11_0001010_1010010_0110010_000 //START: DISPC "\nR27 28="
0110111_0100000_0110010_0111000_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR27 28="
011110_00001_11011_0000000000000000 //DISP R27,int line: 199
110100_00000_11011_0000000000011100 //CMPEQC R27,28,R0 line: 200
011100_00011_00000_0000000100101110 //BEQ R0,failedReg,R3 line: 201
111001_11100_11111_0000000000011101 //ORC R31,29,R28 line: 202
011010_11_0001010_1010010_0110010_000 //START: DISPC "\nR28 29="
0111000_0100000_0110010_0111001_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR28 29="
011110_00001_11100_0000000000000000 //DISP R28,int line: 206
110100_00000_11100_0000000000011101 //CMPEQC R28,29,R0 line: 207
011100_00011_00000_0000000100100111 //BEQ R0,failedReg,R3 line: 208
111001_11101_11111_0000000000011110 //ORC R31,30,R29 line: 209
011010_11_0001010_1010010_0110010_000 //START: DISPC "\nR29 30="
0111001_0100000_0110011_0110000_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR29 30="
011110_00001_11101_0000000000000000 //DISP R29,int line: 213
110100_00000_11101_0000000000011110 //CMPEQC R29,30,R0 line: 214
011100_00011_00000_0000000100100000 //BEQ R0,failedReg,R3 line: 215
111001_11110_11111_0000000000011111 //ORC R31,31,R30 line: 216
011010_11_0001010_1010010_0110011_000 //START: DISPC "\nR30 31="
0110000_0100000_0110011_0110001_0000
0111101_0000000__000000000000000000 //END: DISPC "\nR30 31="
011110_00001_11110_0000000000000000 //DISP R30,int line: 220
```

```
110100_00000_11110_0000000000011111 //CMPEQC R30,31,R0 line: 221
011100_00011_00000_0000000100011001 //BEQ R0,failedReg,R3 line: 222
111001_11111_11111_0000000000100000 //ORC R31,32,R31 line: 223
011010_11_0001010_1010010_0110011_000 //START: DISPC "\nR31 0="
0110001_0100000_0110000_0111101_0000
0000000__0000000000000000000000000 //END: DISPC "\nR31 0="
011110_00001_11111_0000000000000000 //DISP R31,int line: 227
011101_00011_11111_0000000100010011 //BNE R31,failedReg,R3 line: 228
011100_00011_11111_0000000000000101 //BEQ R31,continue_beq,R3 line: 229
011010_11_0001010_1000110_1100001_000 //START: dispc "\nFailed BEQ"
1101001_1101100_1100101_1100100_0000
0100000_1000010_1000101_1010001_0000
0000000__0000000000000000000000000 //END: dispc "\nFailed BEQ"
000001_00000_00000_0000000000000000 //trap line: 234
011010_11_0001010_1010000_1100001_000 //START: dispc "\nPassed BEQ branch"
1110011_1110011_1100101_1100100_0000
0100000_1000010_1000101_1010001_0000
0100000_1100010_1110010_1100001_0000
1101110_1100011_1101000_0000000_0000 //END: dispc "\nPassed BEQ branch"
011100_00011_00000_0000000111000100 //BEQ R0,failedBEQ,R3 line: 240
011010_11_0001010_1010000_1100001_000 //START: dispc "\nPassed BEQ non
branch"
1110011_1110011_1100101_1100100_0000
0100000_1000010_1000101_1010001_0000
0100000_1101110_1101111_1101110_0000
0100000_1100010_1110010_1100001_0000
1101110_1100011_1101000_0000000_0000 //END: dispc "\nPassed BEQ non branch"
011101_00011_00000_0000000000000101 //BNE R0,continue_bne,R3 line: 247
011010_11_0001010_1000110_1100001_000 //START: dispc "\nFailed BNE"
1101001_1101100_1100101_1100100_0000
0100000_1000010_1001110_1000101_0000
0000000__0000000000000000000000000 //END: dispc "\nFailed BNE"
000001_00000_00000_0000000000000000 //trap line: 252
011010_11_0001010_1010000_1100001_000 //START: dispc "\nPassed BNE branch"
1110011_1110011_1100101_1100100_0000
0100000_1000010_1001110_1000101_0000
0100000_1100010_1110010_1100001_0000
1101110_1100011_1101000_0000000_0000 //END: dispc "\nPassed BNE branch"
011101_00011_11111_0000000110101101 //BNE R31,failedBNE,R3 line: 258
011010_11_0001010_1010000_1100001_000 //START: dispc "\nPassed BNE non
branch"
1110011_1110011_1100101_1100100_0000
0100000_1000010_1001110_1000101_0000
0100000_1101110_1101111_1101110_0000
0100000_1100010_1110010_1100001_0000
1101110_1100011_1101000_0000000_0000 //END: dispc "\nPassed BNE non branch"
011111_00000_00000_0000000110110010 //LDR label_neg,R0 line: 265
011010_11_0001010_1010011_1010100_000 //START: DISPC "\nST -15="
0100000_0101101_0110001_0110101_0000
0111101_0000000__000000000000000000 //END: DISPC "\nST -15="
011001_00000_11111_0000111110100000 //ST R0,4000,R31 line: 269
011000_00001_11111_0000111110100000 //LD R31,4000,R1 line: 270
011110_00001_00001_0000000000000000 //DISP R1,int line: 271
100100_00100_00000_00001_00000000000 //CMPEQ R0,R1,R4 line: 272
011100_00011_00100_0000000110011001 //BEQ R4,failedST,R3 line: 273
011010_11_0001010_1010000_1100001_000 //START: dispc "\nPassed LD/ST"
```

```
1110011_1110011_1100101_1100100_0000
0100000_1001100_1000100_0101111_0000
1010011_1010100_0000000__00000000000 //END: dispc "\nPassed LD/ST"
011111_00000_00000_0000000110100011 //LDR label_a,R0 line: 278
011111_00001_00000_0000000110100011 //LDR label_b,R1 line: 279
011010_11_0001010_1000001_1000100_000 //START: DISPC "\nADD 17="
1000100_0100000_0110001_0110111_0000
0111101_0000000__000000000000000000 //END: DISPC "\nADD 17="
100000_00010_00000_00001_00000000000 //ADD R0,R1,R2 line: 283
011110_00001_00010_0000000000000000 //DISP R2,int line: 284
011111_00011_00000_0000000110100001 //LDR label_add,R3 line: 285
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 286
011100_00011_00100_0000000011111100 //BEQ R4,failedADD,R3 line: 287
011010_11_0001010_1010011_1010101_000 //START: DISPC "\nSUB 13="
1000010_0100000_0110001_0110011_0000
0111101_0000000__000000000000000000 //END: DISPC "\nSUB 13="
100001_00010_00000_00001_00000000000 //SUB R0,R1,R2 line: 291
011110_00001_00010_0000000000000000 //DISP R2,int line: 292
011111_00011_00000_0000000110011010 //LDR label_sub,R3 line: 293
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 294
011100_00011_00100_0000000011111001 //BEQ R4,failedSUB,R3 line: 295
011010_11_0001010_1001101_1010101_000 //START: DISPC "\nMUL 30="
1001100_0100000_0110011_0110000_0000
0111101_0000000__000000000000000000 //END: DISPC "\nMUL 30="
100010_00010_00000_00001_00000000000 //MUL R0,R1,R2 line: 299
011110_00001_00010_0000000000000000 //DISP R2,int line: 300
011111_00011_00000_0000000110010011 //LDR label_mul,R3 line: 301
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 302
011100_00011_00100_0000000011110110 //BEQ R4,failedMUL,R3 line: 303
011010_11_0001010_1000100_1001001_000 //START: DISPC "\nDIV 7="
1010110_0100000_0110111_0111101_0000
0000000__0000000000000000000000000 //END: DISPC "\nDIV 7="
100011_00010_00000_00001_00000000000 //DIV R0,R1,R2 line: 307
011110_00001_00010_0000000000000000 //DISP R2,int line: 308
011111_00011_00000_0000000110001100 //LDR label_div,R3 line: 309
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 310
011100_00011_00100_0000000011110011 //BEQ R4,failedDIV,R3 line: 311
011010_11_0001010_1000011_1001101_000 //START: DISPC "\nCMPEQ 0="
1010000_1000101_1010001_0100000_0000
0110000_0111101_0000000__00000000000 //END: DISPC "\nCMPEQ 0="
100100_00010_00000_00001_00000000000 //CMPEQ R0,R1,R2 line: 315
011110_00001_00010_0000000000000000 //DISP R2,int line: 316
011111_00011_00000_0000000110001011 //LDR label_0,R3 line: 317
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 318
011100_00011_00100_0000000011110000 //BEQ R4,failedCMPEQ,R3 line: 319
011010_11_0001010_1000011_1001101_000 //START: DISPC "\nCMPEQ 1="
1010000_1000101_1010001_0100000_0000
0110001_0111101_0000000__00000000000 //END: DISPC "\nCMPEQ 1="
100100_00010_00001_00001_00000000000 //CMPEQ R1,R1,R2 line: 323
011110_00001_00010_0000000000000000 //DISP R2,int line: 324
011111_00011_00000_0000000110000100 //LDR label_1,R3 line: 325
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 326
011100_00011_00100_0000000011101000 //BEQ R4,failedCMPEQ,R3 line: 327
011010_11_0001010_1000011_1001101_000 //START: DISPC "\nCMPLT 0="
1010000_1001100_1010100_0100000_0000
0110000_0111101_0000000__00000000000 //END: DISPC "\nCMPLT 0="
```

```
100101_00010_00000_00001_00000000000 //CMPLT R0,R1,R2 line: 331
011110_00001_00010_0000000000000000 //DISP R2,int line: 332
011111_00011_00000_0000000101111011 //LDR label_0,R3 line: 333
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 334
011100_00011_00100_0000000011100101 //BEQ R4,failedCMPLT,R3 line: 335
011010_11_0001010_1000011_1001101_000 //START: DISPC "\nCMPLT 1="
1010000_1001100_1010100_0100000_0000
0110001_0111101_0000000__00000000000 //END: DISPC "\nCMPLT 1="
100101_00010_00001_00000_00000000000 //CMPLT R1,R0,R2 line: 339
011110_00001_00010_0000000000000000 //DISP R2,int line: 340
011111_00011_00000_0000000101110100 //LDR label_1,R3 line: 341
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 342
011100_00011_00100_0000000011011101 //BEQ R4,failedCMPLT,R3 line: 343
011010_11_0001010_1000011_1001101_000 //START: DISPC "\nCMPLE 0="
1010000_1001100_1000101_0100000_0000
0110000_0111101_0000000__00000000000 //END: DISPC "\nCMPLE 0="
100110_00010_00000_00001_00000000000 //CMPLE R0,R1,R2 line: 347
011110_00001_00010_0000000000000000 //DISP R2,int line: 348
011111_00011_00000_0000000101101011 //LDR label_0,R3 line: 349
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 350
011100_00011_00100_0000000011011010 //BEQ R4,failedCMPLE,R3 line: 351
011010_11_0001010_1000011_1001101_000 //START: DISPC "\nCMPLE 1="
1010000_1001100_1000101_0100000_0000
0110001_0111101_0000000__00000000000 //END: DISPC "\nCMPLE 1="
100110_00010_00000_00000_00000000000 //CMPLE R0,R0,R2 line: 355
011110_00001_00010_0000000000000000 //DISP R2,int line: 356
011111_00011_00000_0000000101100100 //LDR label_1,R3 line: 357
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 358
011100_00011_00100_0000000011010010 //BEQ R4,failedCMPLE,R3 line: 359
011010_11_0001010_1000011_1001101_000 //START: DISPC "\nCMPLE 1="
1010000_1001100_1000101_0100000_0000
0110001_0111101_0000000__00000000000 //END: DISPC "\nCMPLE 1="
100110_00010_00001_00000_00000000000 //CMPLE R1,R0,R2 line: 363
011110_00001_00010_0000000000000000 //DISP R2,int line: 364
011111_00011_00000_0000000101011100 //LDR label_1,R3 line: 365
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 366
011100_00011_00100_0000000011001010 //BEQ R4,failedCMPLE,R3 line: 367
011010_11_0001010_1000001_1001110_000 //START: DISPC "\nAND 2="
1000100_0100000_0110010_0111101_0000
0000000__0000000000000000000000000 //END: DISPC "\nAND 2="
101000_00010_00000_00001_00000000000 //AND R0,R1,R2 line: 371
011110_00001_00010_0000000000000000 //DISP R2,int line: 372
011111_00011_00000_0000000101001000 //LDR label_and,R3 line: 373
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 374
011100_00011_00100_0000000011000111 //BEQ R4,failedAND,R3 line: 375
011010_11_0001010_1001111_1010010_000 //START: DISPC "\nOR 15="
0100000_0110001_0110101_0111101_0000
0000000__0000000000000000000000000 //END: DISPC "\nOR 15="
101001_00010_00000_00001_00000000000 //OR R0,R1,R2 line: 379
011110_00001_00010_0000000000000000 //DISP R2,int line: 380
011111_00011_00000_0000000100111111 //LDR label_or,R3 line: 381
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 382
011100_00011_00100_0000000011000100 //BEQ R4,failedOR,R3 line: 383
011010_11_0001010_1011000_1001111_000 //START: DISPC "\nXOR 13="
1010010_0100000_0110001_0110011_0000
0111101_0000000__000000000000000000 //END: DISPC "\nXOR 13="
```

```
101010_00010_00000_00001_00000000000 //XOR R0,R1,R2 line: 387
011110_00001_00010_0000000000000000 //DISP R2,int line: 388
011111_00011_00000_0000000100111101 //LDR label_xor,R3 line: 389
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 390
011100_00011_00100_0000000011000001 //BEQ R4,failedXOR,R3 line: 391
011010_11_0001010_1011000_1001110_000 //START: DISPC "\nXNOR -14="
1001111_1010010_0100000_0101101_0000
0110001_0110100_0111101_0000000_0000 //END: DISPC "\nXNOR -14="
101011_00010_00000_00001_00000000000 //XNOR R0,R1,R2 line: 395
011110_00001_00010_0000000000000000 //DISP R2,int line: 396
011111_00011_00000_0000000100111010 //LDR label_xnor,R3 line: 397
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 398
011100_00011_00100_0000000010111110 //BEQ R4,failedXNOR,R3 line: 399
011010_11_0001010_1010011_1001000_000 //START: DISPC "\nSHL 60="
1001100_0100000_0110110_0110000_0000
0111101_0000000__000000000000000000 //END: DISPC "\nSHL 60="
101100_00010_00000_00001_00000000000 //SHL R0,R1,R2 line: 403
011110_00001_00010_0000000000000000 //DISP R2,int line: 404
011111_00011_00000_0000000100101110 //LDR label_shl,R3 line: 405
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 406
011100_00011_00100_0000000010111011 //BEQ R4,failedSHL,R3 line: 407
011010_11_0001010_1010011_1001000_000 //START: DISPC "\nSHR 3="
1010010_0100000_0110011_0111101_0000
0000000__0000000000000000000000000 //END: DISPC "\nSHR 3="
101101_00010_00000_00001_00000000000 //SHR R0,R1,R2 line: 411
011110_00001_00010_0000000000000000 //DISP R2,int line: 412
011111_00011_00000_0000000100100111 //LDR label_shr,R3 line: 413
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 414
011100_00011_00100_0000000010111000 //BEQ R4,failedSHR,R3 line: 415
011010_11_0001010_1010011_1010010_000 //START: DISPC "\nSRA 3="
1000001_0100000_0110011_0111101_0000
0000000__0000000000000000000000000 //END: DISPC "\nSRA 3="
101110_00010_00000_00001_00000000000 //SRA R0,R1,R2 line: 419
011110_00001_00010_0000000000000000 //DISP R2,int line: 420
011111_00011_00000_0000000100011111 //LDR label_shr,R3 line: 421
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 422
011100_00011_00100_0000000010110101 //BEQ R4,failedSRA,R3 line: 423
011111_00000_00000_0000000100010011 //LDR label_neg,R0 line: 424
011010_11_0001010_1010011_1010010_000 //START: DISPC "\nSRA -4="
1000001_0100000_0101101_0110100_0000
0111101_0000000__000000000000000000 //END: DISPC "\nSRA -4="
101110_00010_00000_00001_00000000000 //SRA R0,R1,R2 line: 428
011110_00001_00010_0000000000000000 //DISP R2,int line: 429
011111_00011_00000_0000000100011000 //LDR label_sra_neg,R3 line: 430
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 431
011100_00011_00100_0000000010101100 //BEQ R4,failedSRA,R3 line: 432
011111_00000_00000_0000000100001000 //LDR label_a,R0 line: 433
011010_11_0001010_1000001_1001110_000 //START: DISPC "\nANDC 2="
1000100_1000011_0100000_0110010_0000
0111101_0000000__000000000000000000 //END: DISPC "\nANDC 2="
111000_00010_00000_0000000000000010 //ANDC R0,2,R2 line: 437
011110_00001_00010_0000000000000000 //DISP R2,int line: 438
011111_00011_00000_0000000100000110 //LDR label_and,R3 line: 439
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 440
011100_00011_00100_0000000011001110 //BEQ R4,failedANDC,R3 line: 441
011010_11_0001010_1001111_1010010_000 //START: DISPC "\nORC 15="
```

```
1000011_0100000_0110001_0110101_0000
0111101_0000000__000000000000000000 //END: DISPC "\nORC 15="
111001_00010_00000_0000000000000010 //ORC R0,2,R2 line: 445
011110_00001_00010_0000000000000000 //DISP R2,int line: 446
011111_00011_00000_0000000011111101 //LDR label_or,R3 line: 447
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 448
011100_00011_00100_0000000011001011 //BEQ R4,failedORC,R3 line: 449
011010_11_0001010_1011000_1001111_000 //START: DISPC "\nXORC 13="
1010010_1000011_0100000_0110001_0000
0110011_0111101_0000000__00000000000 //END: DISPC "\nXORC 13="
111010_00010_00000_0000000000000010 //XORC R0,2,R2 line: 453
011110_00001_00010_0000000000000000 //DISP R2,int line: 454
011111_00011_00000_0000000011111011 //LDR label_xor,R3 line: 455
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 456
011100_00011_00100_0000000011001000 //BEQ R4,failedXORC,R3 line: 457
011010_11_0001010_1011000_1001110_000 //START: DISPC "\nXNORC -14="
1001111_1010010_1000011_0100000_0000
0101101_0110001_0110100_0111101_0000
0000000__000000000000000000000000 //END: DISPC "\nXNORC -14="
111011_00010_00000_0000000000000010 //XNORC R0,2,R2 line: 462
011110_00001_00010_0000000000000000 //DISP R2,int line: 463
011111_00011_00000_0000000011110111 //LDR label_xnor,R3 line: 464
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 465
011100_00011_00100_0000000011000100 //BEQ R4,failedXNORC,R3 line: 466
011010_11_0001010_1010011_1001000_000 //START: DISPC "\nSHLC 60="
1001100_1000011_0100000_0110110_0000
0110000_0111101_0000000__00000000000 //END: DISPC "\nSHLC 60="
111100_00010_00000_0000000000000010 //SHLC R0,2,R2 line: 470
011110_00001_00010_0000000000000000 //DISP R2,int line: 471
011111_00011_00000_0000000011101011 //LDR label_shl,R3 line: 472
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 473
011100_00011_00100_0000000011000001 //BEQ R4,failedSHLC,R3 line: 474
011010_11_0001010_1010011_1001000_000 //START: DISPC "\nSHRC 3="
1010010_1000011_0100000_0110011_0000
0111101_0000000__000000000000000000 //END: DISPC "\nSHRC 3="
111101_00010_00000_0000000000000010 //SHRC R0,2,R2 line: 478
011110_00001_00010_0000000000000000 //DISP R2,int line: 479
011111_00011_00000_0000000011100100 //LDR label_shr,R3 line: 480
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 481
011100_00011_00100_0000000010111110 //BEQ R4,failedSHRC,R3 line: 482
011111_00000_00000_0000000011010110 //LDR label_a,R0 line: 483
011010_11_0001010_1010011_1010010_000 //START: DISPC "\nSRAC 3="
1000001_1000011_0100000_0110011_0000
0111101_0000000__000000000000000000 //END: DISPC "\nSRAC 3="
111110_00010_00000_0000000000000010 //SRAC R0,2,R2 line: 487
011110_00001_00010_0000000000000000 //DISP R2,int line: 488
011111_00011_00000_0000000011011100 //LDR label_sra,R3 line: 489
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 490
011100_00011_00100_0000000010111010 //BEQ R4,failedSRAC,R3 line: 491
011111_00000_00000_0000000011001111 //LDR label_neg,R0 line: 492
011010_11_0001010_1010011_1010010_000 //START: DISPC "\nSRAC -4="
1000001_1000011_0100000_0101101_0000
0110100_0111101_0000000__00000000000 //END: DISPC "\nSRAC -4="
111110_00010_00000_0000000000000010 //SRAC R0,2,R2 line: 496
011110_00001_00010_0000000000000000 //DISP R2,int line: 497
011111_00011_00000_0000000011010100 //LDR label_sra_neg,R3 line: 498
```

```
100100_00100_00010_00011_00000000000 //CMPEQ R2,R3,R4 line: 499
011100_00011_00100_0000000010110001 //BEQ R4,failedSRAC,R3 line: 500
011111_00001_00000_0000000011010100 //LDR label_1,R1 line: 501
011011_00000_11111_0000000000000000 //JMP R31,R0 line: 502
000001_00000_00000_0000000000000000 //trap line: 503
011010_11_0001010_1000110_1100001_000 //START: failedReg: DISPC "\nFailed
Register\n"
1101001_1101100_1100101_1100100_0000
0100000_1010010_1100101_1100111_0000
1101001_1110011_1110100_1100101_0000
1110010_0001010_0000000__00000000000 //END: failedReg: DISPC "\nFailed
Register\n"
000001_00000_00000_0000000000000000 //trap line: 509
011010_11_0001010_1000110_1100001_000 //START: failedLD: DISPC "\nFailed
LD\n"
1101001_1101100_1100101_1100100_0000
0100000_1001100_1000100_0001010_0000
0000000__0000000000000000000000000 //END: failedLD: DISPC "\nFailed LD\n"
000001_00000_00000_0000000000000000 //trap line: 514
011010_11_0001010_1000110_1100001_000 //START: failedST: DISPC "\nFailed
ST\n"
1101001_1101100_1100101_1100100_0000
0100000_1010011_1010100_0001010_0000
0000000__0000000000000000000000000 //END: failedST: DISPC "\nFailed ST\n"
000001_00000_00000_0000000000000000 //trap line: 519
011010_11_0001010_1000110_1100001_000 //START: failedJMP: DISPC "\nFailed
JMP\n"
1101001_1101100_1100101_1100100_0000
0100000_1001010_1001101_1010000_0000
0001010_0000000__000000000000000000 //END: failedJMP: DISPC "\nFailed JMP\n"
000001_00000_00000_0000000000000000 //trap line: 524
011010_11_0001010_1000110_1100001_000 //START: faileDBEQ: DISPC "\nFailed
BEQ\n"
1101001_1101100_1100101_1100100_0000
0100000_1000010_1000101_1010001_0000
0001010_0000000__000000000000000000 //END: faileDBEQ: DISPC "\nFailed BEQ\n"
000001_00000_00000_0000000000000000 //trap line: 529
011010_11_0001010_1000110_1100001_000 //START: faileDBNE: DISPC "\nFailed
BNE\n"
1101001_1101100_1100101_1100100_0000
0100000_1000010_1001110_1000101_0000
0001010_0000000__000000000000000000 //END: faileDBNE: DISPC "\nFailed BNE\n"
000001_00000_00000_0000000000000000 //trap line: 534
011010_11_0001010_1000110_1100001_000 //START: failedLDR: DISPC "\nFailed
LDR\n"
1101001_1101100_1100101_1100100_0000
0100000_1001100_1000100_1010010_0000
0001010_0000000__000000000000000000 //END: failedLDR: DISPC "\nFailed LDR\n"
000001_00000_00000_0000000000000000 //trap line: 539
011010_11_0001010_1000110_1100001_000 //START: failedADD: DISPC "\nFailed
ADD\n"
1101001_1101100_1100101_1100100_0000
0100000_1000001_1000100_1000100_0000
0001010_0000000__000000000000000000 //END: failedADD: DISPC "\nFailed ADD\n"
000001_00000_00000_0000000000000000 //trap line: 544
```

```
011010_11_0001010_1000110_1100001_000 //START: failedSUB: DISPC "\nFailed
SUB\n"
1101001_1101100_1100101_1100100_0000
0100000_1010011_1010101_1000010_0000
0001010_0000000__000000000000000000 //END: failedSUB: DISPC "\nFailed SUB\n"
000001_00000_00000_0000000000000000 //trap line: 549
011010_11_0001010_1000110_1100001_000 //START: failedMUL: DISPC "\nFailed
MUL\n"
1101001_1101100_1100101_1100100_0000
0100000_1001101_1010101_1001100_0000
0001010_0000000__000000000000000000 //END: failedMUL: DISPC "\nFailed MUL\n"
000001_00000_00000_0000000000000000 //trap line: 554
011010_11_0001010_1000110_1100001_000 //START: failedDIV: DISPC "\nFailed
DIV\n"
1101001_1101100_1100101_1100100_0000
0100000_1000100_1001001_1010110_0000
0001010_0000000__000000000000000000 //END: failedDIV: DISPC "\nFailed DIV\n"
000001_00000_00000_0000000000000000 //trap line: 559
011010_11_0001010_1000110_1100001_000 //START: failedCMPEQ: DISPC "\nFailed
CMPEQ\n"
1101001_1101100_1100101_1100100_0000
0100000_1000011_1001101_1010000_0000
1000101_1010001_0001010_0000000_0000 //END: failedCMPEQ: DISPC "\nFailed
CMPEQ\n"
000001_00000_00000_0000000000000000 //trap line: 564
011010_11_0001010_1000110_1100001_000 //START: failedCMPLT: DISPC "\nFailed
CMPLT\n"
1101001_1101100_1100101_1100100_0000
0100000_1000011_1001101_1010000_0000
1001100_1010100_0001010_0000000_0000 //END: failedCMPLT: DISPC "\nFailed
CMPLT\n"
000001_00000_00000_0000000000000000 //trap line: 569
011010_11_0001010_1000110_1100001_000 //START: failedCMPLE: DISPC "\nFailed
CMPLE\n"
1101001_1101100_1100101_1100100_0000
0100000_1000011_1001101_1010000_0000
1001100_1000101_0001010_0000000_0000 //END: failedCMPLE: DISPC "\nFailed
CMPLE\n"
000001_00000_00000_0000000000000000 //trap line: 574
011010_11_0001010_1000110_1100001_000 //START: failedAND: DISPC "\nFailed
AND\n"
1101001_1101100_1100101_1100100_0000
0100000_1000001_1001110_1000100_0000
0001010_0000000__000000000000000000 //END: failedAND: DISPC "\nFailed AND\n"
000001_00000_00000_0000000000000000 //trap line: 579
011010_11_0001010_1000110_1100001_000 //START: failedOR: DISPC "\nFailed
OR\n"
1101001_1101100_1100101_1100100_0000
0100000_1001111_1010010_0001010_0000
0000000__0000000000000000000000000 //END: failedOR: DISPC "\nFailed OR\n"
000001_00000_00000_0000000000000000 //trap line: 584
011010_11_0001010_1000110_1100001_000 //START: failedXOR: DISPC "\nFailed
XOR\n"
1101001_1101100_1100101_1100100_0000
0100000_1011000_1001111_1010010_0000
0001010_0000000__000000000000000000 //END: failedXOR: DISPC "\nFailed XOR\n"
```

```
000001_00000_00000_0000000000000000 //trap line: 589
011010_11_0001010_1000110_1100001_000 //START: failedXNOR: DISPC "\nFailed
XNOR\n"
1101001_1101100_1100101_1100100_0000
0100000_1011000_1001110_1001111_0000
1010010_0001010_0000000__00000000000 //END: failedXNOR: DISPC "\nFailed
XNOR\n"
000001_00000_00000_0000000000000000 //trap line: 594
011010_11_0001010_1000110_1100001_000 //START: failedSHL: DISPC "\nFailed
SHL\n"
1101001_1101100_1100101_1100100_0000
0100000_1010011_1001000_1001100_0000
0001010_0000000__000000000000000000 //END: failedSHL: DISPC "\nFailed SHL\n"
000001_00000_00000_0000000000000000 //trap line: 599
011010_11_0001010_1000110_1100001_000 //START: failedSHR: DISPC "\nFailed
SHR\n"
1101001_1101100_1100101_1100100_0000
0100000_1010011_1001000_1010010_0000
0001010_0000000__000000000000000000 //END: failedSHR: DISPC "\nFailed SHR\n"
000001_00000_00000_0000000000000000 //trap line: 604
011010_11_0001010_1000110_1100001_000 //START: failedSRA: DISPC "\nFailed
SRA\n"
1101001_1101100_1100101_1100100_0000
0100000_1010011_1010010_1000001_0000
0001010_0000000__000000000000000000 //END: failedSRA: DISPC "\nFailed SRA\n"
000001_00000_00000_0000000000000000 //trap line: 609
011010_11_0001010_1000110_1100001_000 //START: failedADDC: DISPC "\nFailed
ADDC\n"
1101001_1101100_1100101_1100100_0000
0100000_1000001_1000100_1000100_0000
1000011_0001010_0000000__00000000000 //END: failedADDC: DISPC "\nFailed
ADDC\n"
000001_00000_00000_0000000000000000 //trap line: 614
011010_11_0001010_1000110_1100001_000 //START: failedSUBC: DISPC "\nFailed
SUBC\n"
1101001_1101100_1100101_1100100_0000
0100000_1010011_1010101_1000010_0000
1000011_0001010_0000000__00000000000 //END: failedSUBC: DISPC "\nFailed
SUBC\n"
000001_00000_00000_0000000000000000 //trap line: 619
011010_11_0001010_1000110_1100001_000 //START: failedMULC: DISPC "\nFailed
MULC\n"
1101001_1101100_1100101_1100100_0000
0100000_1001101_1010101_1001100_0000
1000011_0001010_0000000__00000000000 //END: failedMULC: DISPC "\nFailed
MULC\n"
000001_00000_00000_0000000000000000 //trap line: 624
011010_11_0001010_1000110_1100001_000 //START: failedDIVC: DISPC "\nFailed
DIVC\n"
1101001_1101100_1100101_1100100_0000
0100000_1000100_1001001_1010110_0000
1000011_0001010_0000000__00000000000 //END: failedDIVC: DISPC "\nFailed
DIVC\n"
000001_00000_00000_0000000000000000 //trap line: 629
011010_11_0001010_1000110_1100001_000 //START: failedCMPEQC: DISPC "\nFailed
CMPEQC\n"
```

```
1101001_1101100_1100101_1100100_0000
0100000_1000011_1001101_1010000_0000
1000101_1010001_1000011_0001010_0000
0000000__0000000000000000000000000 //END: failedCMPEQC: DISPC "\nFailed
CMPEQC\n"
000001_00000_00000_0000000000000000 //trap line: 635
011010_11_0001010_1000110_1100001_000 //START: failedCMPLTC: DISPC "\nFailed
CMPLTC\n"
1101001_1101100_1100101_1100100_0000
0100000_1000011_1001101_1010000_0000
1001100_1010100_1000011_0001010_0000
0000000__0000000000000000000000000 //END: failedCMPLTC: DISPC "\nFailed
CMPLTC\n"
000001_00000_00000_0000000000000000 //trap line: 641
011010_11_0001010_1000110_1100001_000 //START: failedCMPLEC: DISPC "\nFailed
CMPLEC\n"
1101001_1101100_1100101_1100100_0000
0100000_1000011_1001101_1010000_0000
1001100_1000101_1000011_0001010_0000
0000000__0000000000000000000000000 //END: failedCMPLEC: DISPC "\nFailed
CMPLEC\n"
000001_00000_00000_0000000000000000 //trap line: 647
011010_11_0001010_1000110_1100001_000 //START: failedANDC: DISPC "\nFailed
ANDC\n"
1101001_1101100_1100101_1100100_0000
0100000_1000001_1001110_1000100_0000
1000011_0001010_0000000__00000000000 //END: failedANDC: DISPC "\nFailed
ANDC\n"
000001_00000_00000_0000000000000000 //trap line: 652
011010_11_0001010_1000110_1100001_000 //START: failedORC: DISPC "\nFailed
ORC\n"
1101001_1101100_1100101_1100100_0000
0100000_1001111_1010010_1000011_0000
0001010_0000000__000000000000000000 //END: failedORC: DISPC "\nFailed ORC\n"
000001_00000_00000_0000000000000000 //trap line: 657
011010_11_0001010_1000110_1100001_000 //START: failedXORC: DISPC "\nFailed
XORC\n"
1101001_1101100_1100101_1100100_0000
0100000_1011000_1001111_1010010_0000
1000011_0001010_0000000__00000000000 //END: failedXORC: DISPC "\nFailed
XORC\n"
000001_00000_00000_0000000000000000 //trap line: 662
011010_11_0001010_1000110_1100001_000 //START: failedXNORC: DISPC "\nFailed
XNORC\n"
1101001_1101100_1100101_1100100_0000
0100000_1011000_1001110_1001111_0000
1010010_1000011_0001010_0000000_0000 //END: failedXNORC: DISPC "\nFailed
XNORC\n"
000001_00000_00000_0000000000000000 //trap line: 667
011010_11_0001010_1000110_1100001_000 //START: failedSHLC: DISPC "\nFailed
SHLC\n"
1101001_1101100_1100101_1100100_0000
0100000_1010011_1001000_1001100_0000
1000011_0001010_0000000__00000000000 //END: failedSHLC: DISPC "\nFailed
SHLC\n"
000001_00000_00000_0000000000000000 //trap line: 672
```

```
011010_11_0001010_1000110_1100001_000 //START: failedSHRC: DISPC "\nFailed
SHRC\n"
1101001_1101100_1100101_1100100_0000
0100000_1010011_1001000_1010010_0000
1000011_0001010_0000000__00000000000 //END: failedSHRC: DISPC "\nFailed
SHRC\n"
000001_00000_00000_0000000000000000 //trap line: 677
011010_11_0001010_1000110_1100001_000 //START: failedSRAC: DISPC "\nFailed
SRAC\n"
1101001_1101100_1100101_1100100_0000
0100000_1010011_1010010_1000001_0000
1000011_0001010_0000000__00000000000 //END: failedSRAC: DISPC "\nFailed
SRAC\n"
000001_00000_00000_0000000000000000 //trap line: 682
011010_11_0001010_1000110_1100001_000 //START: failedST: DISPC "\nFailed
ST\n"
1101001_1101100_1100101_1100100_0000
0100000_1010011_1010100_0001010_0000
0000000__0000000000000000000000000 //END: failedST: DISPC "\nFailed ST\n"
000001_00000_00000_0000000000000000 //trap line: 687
011010_11_0001010_1000110_1100001_000 //START: failedBNE: DISPC "\nFailed
BNE\n"
1101001_1101100_1100101_1100100_0000
0100000_1000010_1001110_1000101_0000
0001010_0000000__000000000000000000 //END: failedBNE: DISPC "\nFailed BNE\n"
000001_00000_00000_0000000000000000 //trap line: 692
011010_11_0001010_1000110_1100001_000 //START: failedBEQ: DISPC "\nFailed
BEQ\n"
1101001_1101100_1100101_1100100_0000
0100000_1000010_1000101_1010001_0000
0001010_0000000__000000000000000000 //END: failedBEQ: DISPC "\nFailed BEQ\n"
000001_00000_00000_0000000000000000 //trap line: 697
00000000000000000000000000001111 //label_a: DB 15 line: 698
00000000000000000000000000000010 //label_b: DB 2 line: 699
11111111111111111111111111110001 //label_neg: DB -15 line: 700
00000000000000000000000000001111 //label_or: DB 15 line: 701
00000000000000000000000000000010 //label_and: DB 2 line: 702
00000000000000000000000000010001 //label_add: DB 17 line: 703
00000000000000000000000000001101 //label_sub: DB 13 line: 704
00000000000000000000000000011110 //label_mul: DB 30 line: 705
00000000000000000000000000000111 //label_div: DB 7 line: 706
00000000000000000000000000001101 //label_xor: DB 13 line: 707
00000000000000000000000000111100 //label_shl: DB 60 line: 708
00000000000000000000000000000011 //label_shr: DB 3 line: 709
00000000000000000000000000000011 //label_sra: DB 3 line: 710
11111111111111111111111111111100 //label_sra_neg: DB -4 line: 711
11111111111111111111111111110010 //label_xnor: DB -14 line: 712
00000000000000000000000000000000 //label_0: DB 0 line: 713
00000000000000000000000000000001 //label_1: DB 1 line: 714
0001010_1000100_1101001_1110011_0000 //START: label_str: DB "\nDisplay string
passed"
1110000_1101100_1100001_1111001_0000
0100000_1110011_1110100_1110010_0000
1101001_1101110_1100111_0100000_0000
1110000_1100001_1110011_1110011_0000
```

```
1100101_1100100_0000000_00000000000 //END: label_str: DB "\nDisplay string
passed"
```

compile.py

```
import sys
import os
import re
import binascii
import codecs
import ctypes
import math

# B Microprocessor Assembler
# Usage: python compile.py program.asm
# Outputs a .bin file of the same name filled with 32-bit words
# of the assembly program written in binary formatted for Verilog
# Based off of https://gist.github.com/TerrorBite/2464756

class SourceLine:
    def __init__(self, line_num, inum, line, opcode, arg1, arg2, arg3):
        self.line_num = line_num
        self.inum = inum
        self.line = line
        self.opcode = opcode
        self.arg1 = arg1
        self.arg2 = arg2
        self.arg3 = arg3


def twos_comp(val, bits):
    """compute the 2's complement of int value val"""
    if (val & (1 << (bits - 1))) != 0:  # if sign bit is set e.g., 8bit: 128-255
        val = val + (1 << bits)   # compute negative value
    return val  # return positive value as is


def is_int(val):
    try:
        int(val)
        return True
    except ValueError:
        return False


def is_float(val):
    try:
        float(val)
        return True
    except ValueError:
        return False


def str_to_int_arr(val):
    # print("Parsing '{}'".format(val))
    if val is None:
        return [0]
    parsed = codecs.getdecoder("unicode_escape")(val[1:-1])[0]
    code_list = [int(ord(c)) for c in parsed]
    code_list.append(0)
    return code_list


class BCompiler:
    def __init__(self):
        self.re_line = re.compile(
            r'^([^:]+:\s+)?(?P<instr>[A-Za-
z]{2,6})\s+(?P<argc>[^,]+)?(,\s*(?P<arga>[^,]+))?(,(?P<argb>[^,;]+))?')
        self.re_label = re.compile(r'^(?P<label>[^:]+):')
        self.re_char = re.compile(r"'(?P<escape>\\)?(?P<char>[^'\\])'")
        self.re_disp_str = re.compile(r'^([^:]+:\s+)?(?P<instr>[A-Za-
z]{2,6})\s+(?P<arg>\"(?:[^\"]|\\\")*\")')
        self.instructions = {
            "LD": 0b011000,
            "ST": 0b011001,
            "DISP": 0b011110,
            "JMP": 0b011011,
            "BEQ": 0b011100,
            "BNE": 0b011101,
            "DISPC": 0b011010,
            "LDR": 0b011111,
            "ADD": 0b100000,
            "SUB": 0b100001,
            "MUL": 0b100010,
            "DIV": 0b100011,
            "CMPEQ": 0b100100,
            "CMPLT": 0b100101,
            "CMPLE": 0b100110,
            "AND": 0b101000,
            "OR": 0b101001,
            "XOR": 0b101010,
            "XNOR": 0b101011,
            "SHL": 0b101100,
            "SHR": 0b101101,
            "SRA": 0b101110,
            "ADDC": 0b110000,
            "SUBC": 0b110001,
            "MULC": 0b110010,
            "DIVC": 0b110011,
            "CMPEQC": 0b110100,
            "CMPLTC": 0b110101,
            "CMPLEC": 0b110110,
            "ANDC": 0b111000,
            "ORC": 0b111001,
            "XORC": 0b111010,
            "XNORC": 0b111011,
            "SHLC": 0b111100,
            "SHRC": 0b111101,
            "SRAC": 0b111110,
            "TRAP": 0b000001,
            "DB": None
        }
        self.regs = {
            "R0": 0,
            "R1": 1,
            "R2": 2,
            "R3": 3,
            "R4": 4,
            "R5": 5,
            "R6": 6,
            "R7": 7,
            "R8": 8,
            "R9": 9,
            "R10": 10,
            "R11": 11,
            "R12": 12,
            "R13": 13,
            "R14": 14,
            "R15": 15,
            "R16": 16,
            "R17": 17,
            "R18": 18,
            "R19": 19,
            "R20": 20,
            "R21": 21,
            "R22": 22,
            "R23": 23,
            "R24": 24,
            "R25": 25,
            "R26": 26,
            "R27": 27,
            "R28": 28,
            "R29": 29,
            "R30": 30,
            "R31": 31,
            "XP": 30,
            "SP": 29,
            "LP": 28,
            "BP": 27
        }

        self.types = {
            "INT": 1,
            "INTEGER": 1,
            "FLOAT": 2,
            "DOUBLE": 2,
            "STR": 3,
            "STRING": 3
        }

    def char_to_int(self, char):
        if char is None:
            return 0

        m = self.re_char.match(char)

        # Not a character, so return
        if m is None:
            return char

        esc = m.group('escape')
        c = m.group('char')

        # Not an escape character, so return ascii code
        if esc != '\\':
            return ord(c)

        # Can't craft these, so handle each escape character
        if c == 'a':
            return ord('\a')
        elif c == 'b':
            return ord('\b')
        elif c == 'f':
            return ord('\f')
        elif c == 'n':
            return ord('\n')
        elif c == 'r':
            return ord('\r')
        elif c == 't':
            return ord('\t')
        elif c == 'v':
            return ord('\v')
        elif len(c) > 1:
            if c[0] == 'x':
                return int(c[1:], 16)
            elif c[0] == 'o':
                return int(c[1:], 8)
        else:
            return ord(c)

    def compile(self):
        filepath = None
        if len(sys.argv) < 2:
            filepath = "program.asm"
        else:
            filepath = sys.argv[1]

        if not os.path.isfile(filepath):
            print("File path {} does not exist. Exiting...".format(filepath))

            sys.exit()

        labels = {}
        icount = 0
        sourcelines = []
        with open("program.bin", "w") as fout:
            # Pass 1
            with open(filepath) as fin:
                linecount = 0
                for line in fin:
                    linecount += 1
                    line = line.strip().split(';')[0]
                    m = self.re_label.match(line)
                    if m is not None:
                        labels[m.group('label')] = icount + 1
                        print("Line {}: Found label {}".format(linecount,
m.group('label')))
                    m = self.re_line.match(line)
                    if m is not None:
                        arga = self.char_to_int(m.group('arga'))
                        argb = self.char_to_int(m.group('argb'))
                        argc = self.char_to_int(m.group('argc'))
                        # Have to count the size of a dispc instruction since it has
data after it
                        if m.group('instr').upper() == "DISPC":
                            mstr = self.re_disp_str.match(line)
                            if is_int(argc):
                                sourcelines.append(
                                    SourceLine(linecount, icount, line,
m.group('instr'), int(argc), None, None))
                                icount += 2
                                print("Line {}: Found int {}".format(linecount,
int(argc)))
                            elif is_float(argc):
                                sourcelines.append(
                                    SourceLine(linecount, icount, line,
m.group('instr'), float(argc), None, None))
                                icount += 2
                                print("Line {}: Found float {}".format(linecount,
float(argc)))
                            elif mstr is not None:
                                str_data = str_to_int_arr(mstr.group('arg'))
                                sourcelines.append(
                                    SourceLine(linecount, icount, line,
m.group('instr'), str_data, None, None))
```

```
                                icount += 1 + int(len(str_data) / 4)
                                print("Line {}: Found string {}".format(linecount,
mstr.group('arg')))
                            else:
                                print(
                                "Line {}: DB is not a string, int, or float, it is a
{}".format(linecount, type(argc)))
                        elif m.group('instr').upper() == "DB":
                            mstr = self.re_disp_str.match(line)
                            if is_int(argc):
                                sourcelines.append(
                                    SourceLine(linecount, icount, line,
m.group('instr'), int(argc), None, None))
                                icount += 1
                                print("Line {}: Found int {}".format(linecount,
int(argc)))
                            elif is_float(argc):
                                sourcelines.append(
                                    SourceLine(linecount, icount, line,
m.group('instr'), float(argc), None, None))
                                icount += 1
                                print("Line {}: Found float {}".format(linecount,
float(argc)))
                            elif mstr is not None:
                                str_data = str_to_int_arr(mstr.group('arg'))
                                sourcelines.append(
                                    SourceLine(linecount, icount, line,
m.group('instr'), str_data, None, None))
                                icount += math.ceil(len(str_data) / 4.0)
                                print("Line {}: Found string {}".format(linecount,
mstr.group('arg')))
                            else:
                                print(
                                    "Line {}: DB is not a string, int, or float, it is
a {}".format(linecount,

type(argc)))
                        else:
                            sourcelines.append(SourceLine(linecount, icount, line,
m.group('instr'), argc, arga, argb))
                            print(
                            "Line {}: Found instruction
{}({},{},{})".format(linecount, m.group('instr'), argc, arga,
                                                                            argb))
                            icount += 1
                    elif "TRAP" == line.upper():
                        sourcelines.append(SourceLine(linecount, icount, line, "trap",
None, None, None))
                        print("Line {}: Found instruction trap".format(linecount))
                        icount += 1
            # Pass 2
            for line in sourcelines:
                if line.opcode.upper() in self.instructions:
                    upper_opcode = line.opcode.upper()
                    inst = ""
                    if upper_opcode == "DB":
                        if type(line.arg1) is int:
                            print("Line {}: processing db {}".format(line.line_num,
line.arg1))
                            inst += "{:032b}".format(twos_comp(line.arg1, 32))
                        elif type(line.arg1) is float:
                            inst +=
"{:032b}".format(bin(ctypes.c_uint.from_buffer(ctypes.c_float(line.arg1)).value))
```

```
                        elif type(line.arg1) is list:
                            num_bits = 0
                            for i in range(0, len(line.arg1)):
                                if (i + 1) % 4 == 0:
                                    inst += "{:07b}".format(line.arg1[i])
                                    num_bits += 7
                                    # Pad string ending
                                    if num_bits < 32:
                                        inst += "_" + ("0" * (32 - num_bits))
                                    if i == 3:
                                        inst += " //START: " + line.line

                                    # Don't print new line if this is the last
character
                                    if i != len(line.arg1):
                                        inst += "\n"
                                    num_bits = 0
                                else:
                                    inst += "{:07b}_".format(line.arg1[i])
                                    num_bits += 7
                            # Pad the last bits
                            if 0 < num_bits < 32:
                                inst += ("0" * (32 - num_bits))
                        else:
                            print("Line {}: Invalid db {}".format(line.line_num,
type(line.arg1)))
                    else:
                        opcode = self.instructions[upper_opcode]
                        inst += "{:06b}_".format(opcode)
                        args_stripped = [str(line.arg1).replace("$", "").upper(),
                                         str(line.arg2).replace("$", "").upper(),
                                         str(line.arg3).replace("$", "").upper()]
                        # print(args_stripped)
                        opcode_cat = opcode >> 3
                        # Special instruction
                        if upper_opcode == "TRAP":
                            inst += "00000_00000_0000000000000000"
                        elif opcode_cat == 0b011:
                            if upper_opcode == "DISP":
                                if args_stripped[1] in args_stripped[1]:
                                    if args_stripped[1] in ["STR", "STRING"]:
                                        inst +=
"{:05b}_{:05b}_{:016b}".format(self.types[args_stripped[1]], 0,

labels[line.arg1] - line.inum - 2)
                                    else:
                                        inst +=
"{:05b}_{:05b}_{:016b}".format(self.types[args_stripped[1]],

self.regs[args_stripped[0]], 0)
                                else:
                                    print(
                                    "Error: this shouldn't happen. DISP type is a {},
not an int, float, or str".format(
                                        type(line.arg2)))
                            elif upper_opcode == "LD":
                                if args_stripped[2] in self.regs:
                                    if args_stripped[0] in self.regs:
                                        inst +=
"{:05b}_{:05b}_{:016b}".format(self.regs[args_stripped[2]],

self.regs[args_stripped[0]],
twos_comp(int(line.arg2), 16))
                                    else:
                                        print("Line {}: {} is not a
register".format(line.line_num, args_stripped[0]))
                                else:
                                    print("Line {}: {} is not a
register".format(line.line_num, args_stripped[2]))
                            elif upper_opcode == "ST":
                                if args_stripped[2] in self.regs:
                                    if args_stripped[0] in self.regs:
                                        inst +=
"{:05b}_{:05b}_{:016b}".format(self.regs[args_stripped[0]],

self.regs[args_stripped[2]],

twos_comp(int(line.arg2), 16))
                                    else:
                                        print("Line {}: {} is not a
register".format(line.line_num, args_stripped[0]))
                                else:
                                    print("Line {}: {} is not a
register".format(line.line_num, args_stripped[2]))
                            elif upper_opcode == "JMP":
                                if args_stripped[1] in self.regs:
                                    if args_stripped[0] in self.regs:
                                        inst +=
"{:05b}_{:05b}_{:016b}".format(self.regs[args_stripped[1]],

self.regs[args_stripped[0]], 0)
                                    else:
                                        print("Line {}: {} is not a
register".format(line.line_num, args_stripped[0]))
                                else:
                                    print("Line {}: {} is not a
register".format(line.line_num, args_stripped[1]))
                            elif upper_opcode in ["BEQ", "BNE"]:
                                if args_stripped[2] in self.regs:
                                    if args_stripped[0] in self.regs:
                                        inst +=
"{:05b}_{:05b}_{:016b}".format(self.regs[args_stripped[2]],

self.regs[args_stripped[0]],

twos_comp(labels[line.arg2] - line.inum - 2, 16))
                                    else:
                                        print("Line {}: {} is not a
register".format(line.line_num, args_stripped[0]))
                                else:
                                    print("Line {}: {} is not a
register".format(line.line_num, args_stripped[2]))
                            elif upper_opcode == "LDR":
                                inst +=
"{:05b}_{:05b}_{:016b}".format(self.regs[args_stripped[1]], 0,

twos_comp(labels[line.arg1] - line.inum - 2, 16))
                            elif upper_opcode == "DISPC":
                                if type(line.arg1) is int:
                                    inst += "{:05b}_{:05b}_{:016b}\n{:031b}".format(1,
0, 0, twos_comp(line.arg1, 32))
                                elif type(line.arg1) is float:
                                    inst += "{:05b}_{:05b}_{:016b}\n{:031b}".format(2,
0, 0, bin(
ctypes.c_uint.from_buffer(ctypes.c_float(line.arg1)).value))
                                elif type(line.arg1) is list:
                                    inst += "{:02b}_".format(3)
                                    num_bits = 0
                                    for i in range(0, 3):
                                        if len(line.arg1) > i == 2:
                                            inst += "{:07b}_000 //START:
{}\n".format(line.arg1[i], line.line)
                                        elif len(line.arg1) > i:
                                            inst += "{:07b}_".format(line.arg1[i])
                                        else:
                                            inst += "{:07b}".format(0)
                                    if len(line.arg1) > 3:
                                        for i in range(3, len(line.arg1)):
                                            if (i - 2) % 4 == 0:
                                                inst += "{:07b}".format(line.arg1[i])
                                                num_bits += 7
                                                # Pad string ending
                                                if num_bits < 32:
                                                    inst += "_" + ("0" * (32 -
num_bits))

                                                # Don't print new line if this is the
last character
                                                if i != len(line.arg1):
                                                    inst += "\n"
                                                num_bits = 0
                                            else:
                                                #
print("charcode={}".format(line.arg1[i]))
                                                inst += "{:07b}_".format(line.arg1[i])
                                                num_bits += 7
                                    # Pad the last bits
                                    if 0 < num_bits < 32:
                                        inst += "_" + ("0" * (32 - num_bits))
                                else:
                                    print("Error: this shouldn't happen. DISPC arg is
a {}".format(type(line.arg1)))
                            else:
                                print("Error: this shouldn't happen.
Opcode='{}'".format(upper_opcode))
                        # Check if constant math operation
                        elif opcode_cat in [0b110, 0b111]:
                            if args_stripped[0] in self.regs:
                                if args_stripped[2] in self.regs:
                                    if line.arg2.isdigit():
                                        inst +=
"{:05b}_{:05b}_{:016b}".format(self.regs[args_stripped[2]],

self.regs[args_stripped[0]],

twos_comp(int(line.arg2), 16))
                                    else:
                                        print("Line {}: Literal {} is not a
number".format(line.line_num, line.arg3))
                                else:
                                    print("Line {}: {} is not a register (const)
opcode={}".format(line.line_num,

args_stripped[2],

opcode))
                            else:
                                print("Line {}: {} is not a register (const)
opcode={}".format(line.line_num,

args_stripped[0],

opcode))

                        # Check if math operation
                        elif opcode_cat in [0b100, 0b101]:
                            error = False
                            for arg in args_stripped:
                                if arg not in self.regs:
                                    print("Line {}: {} is not a register
(math)".format(line.line_num, arg))
                                    error = True
                            if not error:
                                inst +=
"{:05b}_{:05b}_{:05b}_{:011}".format(self.regs[args_stripped[2]],

self.regs[args_stripped[0]],

self.regs[args_stripped[1]], 0)
                    if inst[-1] == '\n':
                        inst = inst[:-1]
                    if '\n' in inst:
                        fout.write('{} //END: {}\n'.format(inst, line.line))
                    else:
                        fout.write('{} //{} line: {}\n'.format(inst, line.line,
line.inum))

                else:
                    print("Line %d: invalid instruction %s".format(line.line_num,
line.opcode))
        print("Done!")


if __name__ == '__main__':
    bc = BCompiler()
    bc.compile()
```