ELEC 374: MiniSRC CPU Project

Final Report

Group 4

Jacob Chisholm (20335775)

Hendrix Gryspeerdt (21hgg3)

Luke Strickland (21laps1)

April 6, 2025

“We do hereby verify that this written lab report is our own work and contains our own original ideas, concepts, and designs. No portion of this report has been copied in whole or in part from another source, with the possible exception of properly referenced material".

Abstract

This report presents the design, implementation, and evaluation of the MiniSRC CPU, a 32-bit RISC-based processor architecture supporting integer arithmetic, logic operations, conditional branching, memory and port-mapped I/O, and a VGA interface. The design comprises a detailed instruction set architecture, data path components, and control logic optimized for efficient decoding and minimal propagation delay. Implementation was done on an FPGA (Cyclone II DE2 development board), achieving a maximum operating frequency of 50 MHz with approximately 7000 logic elements utilized. Although fully operational individually, pipelining was not fully integrated, providing room for future work on pipeline registers, hazard detection, and forwarding logic. Overall, the project delivered a working and scalable foundation suitable for further expansion into a fully pipelined processor.

Table of Contents

Project Specification

Processor State

PC<31..0>: 32-bit Program Counter (PC)

IR<31..0>: 32-bit Instruction Register (IR)

R[0..15]<31..0>: 16 32-bit registers, named R[0] through R[15]

- R[0]<31..0>: 1 Constant zero register
- R[1..7]<31..0>: 7 General-Purpose Registers
- R[8]<31..0>: Return Address Register (RA)
- R[9]<31..0>: Stack Pointer (SP)
- R[10..13]<31..0>: Four Argument Registers
- R[14..15]<31..0]: Two Return Value Registers

RASH<31..0>: (Register ALU Storage Hi) 32-bit Register dedicated to keep the high-order word of a Multiplication product, or the Remainder of a Division operation

RASL<31..0>: (Register ALU Storage Low) 32-bit Register dedicated to keep the low-order word of a Multiplication product, or the Quotient of a Division operation

I/O State

Mem.In<31..0>: 32-bit memory data input port

MemData.Out<31..0>: 32-bit memory data output port

MemAddress.Out<31..0>: 32-bit memory address output port

MemRead.Out: indicate that the processor is requesting data from MemAddress.Out.

MemWrite.Out: indicate that the processor is requesting data to be written at location MemAddress.Out.

MemReady.In: Indicate whether assertion of MemRead.out or MemWrite.out signals have completed their function in the memory unit.

In.Port<31..0>: 32-bit Input Port

Out.Port<31..0>: 32-bit Output Port

Run.Out: Run/halt Indicator

Reset.In: Reset signal

Our processor assumes a byte-addressable address space, hence the memory address output of MemAddress.Out and the logical address space of the processor is 2^{32} 8-bit bytes or 2^{32} 32-bit words. This has implications for the encoding of the branch instruction and program counter where the branch offset is stored as an offset in words since each instruction is a 32-bit word in memory and the program counter is incremented by multiples of 4.

Additionally, for simplicity, our processor does not output a Stop or Halt signal when in the halted state. The halted state can be tested for by observing that MemAddress.Out remains unchanged between the number of cycles it takes to complete an instruction.

Instruction Set Specification

The Arithmetic Logic Unit (ALU) performs 13 operations: addition, subtraction, multiplication, division, shift right, shift right arithmetic, shift left, rotate right, rotate left, logical AND, logical OR, Negate (2's complement), and NOT (1's complement). The instructions in Mini SRC are one-word (32-bit) long each. They can be categorized as Load and Store instructions, Arithmetic and Logical instructions, Conditional Branch and Jump instructions, Input/Output instructions, and miscellaneous instructions. The following six addressing modes are supported: Direct, Indexed, Register, Register Indirect, Immediate, and Relative.

Instruction Formats

There are five instruction formats, as shown in the table below.

Name	Fields					Comments
Field size	31..27 5 bits	26..23 4 bits	22..19 4 bits	18..15 4 bits	14..0 15 bits	All instructions are 32-bit long
R-Format	OP-code	Ra	Rb	Rc	Unused	3 Register ALU instruction
I-Format	OP-code	Ra	Rb	Constant C / Unused		ALU Immediate instruction; Load/Store
B-Format	OP-code	Ra	C2	Constant C		Branch

J-Format	OP-code	Ra	Unused	Jump
M-Format	OP-code	Unused		Misc.

Op-code: specifies the type of operation to be performed.

Ra, Rb, Rc: specify addresses of registers in the register file, 0000=>R0, 0001=>R1, …, 1111=>R15.

C: 19-bit constant value.

C2: 4-bit value representing the branch condition type where only the lower 2 bits are considered.

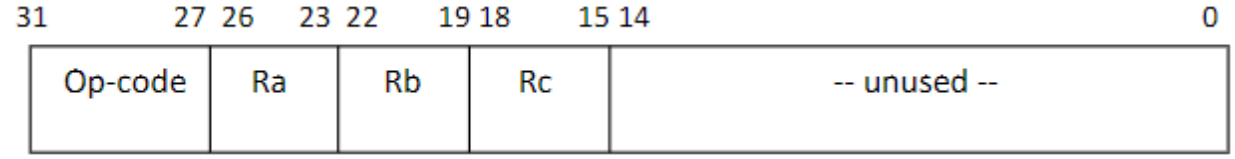
Load and Store instructions

operands in memory can be accessed only through load/store instructions.

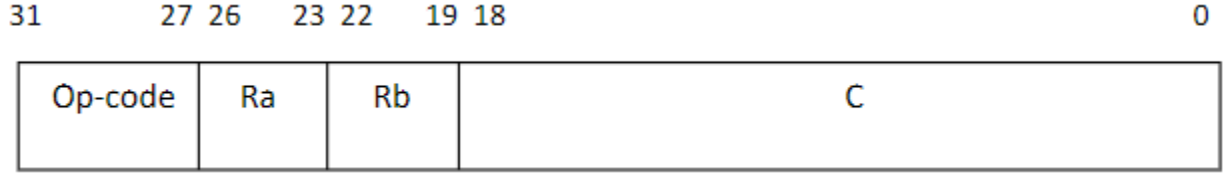
(a) ld, ldi, st I-Format

Arithmetic and Logical instructions

(a) add, sub, and, or, shr, shra, shl, ror, rol R-Format



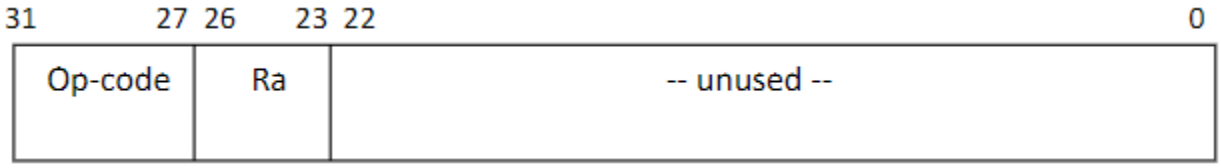
(b) addi, andi, ori I-Format



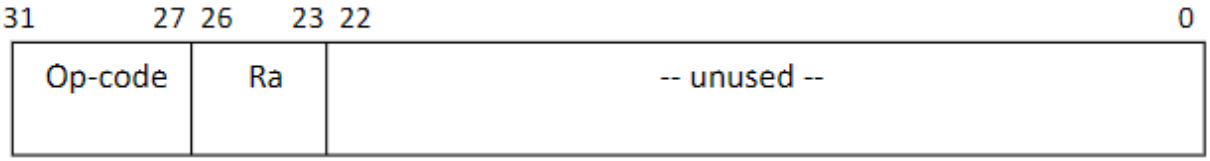
(c) mul, div, neg, not I-Format

Branch instructions: brzr, brnz, brmi, brpl B-Format

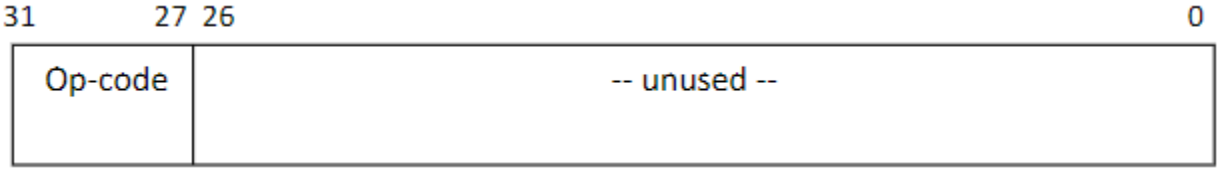
Jump instructions: jr, jal J-Format



Input/Output and MFHI/MFLO instructions: in, out, mfhi, mflo J-Format



Miscellaneous instructions: nop, halt M-Format



Explicit Instruction Encodings

Load and Store Instructions

ld: Load Direct (00000xxxx0000xxxxxxxxxxxxxxxxxxxx)	R[Ra] ← M[C (sign-extended)] Direct addressing, Rb = R0	ld	Ra, C

ld: Load Indexed/Register Indirect (00000xxxxxxxxxxxxxxxxxxxxxxxxxxx)	R[Ra] ← M[R[Rb] + C (sign-extended)] Indexed addressing, Rb ≠ R0 If C = 0 → Register Indirect addressing	ld	Ra, C(Rb)
ldi: Load Immediate (00001xxxx0000xxxxxxxxxxxxxxxxxxxx)	R[Ra] ← C (sign-extended) Immediate addressing, Rb = R0	ldi	Ra, C
(00001xxxxxxxxxxxxxxxxxxxxxxxxxxxx)	R[Ra] ← R[Rb] + C (sign-extended) Immediate addressing, Rb ≠ R0 If C = 0 → instruction acts like a simple register transfer If C ≠ 0 and Ra = Rb → Increment/decrement instruction	ldi	Ra, C(Rb)

To note, the Load Immediate instruction above in the format “ldi Ra, C” is identical in functionality to the Add Immediate instruction (detailed in the next section) with the first argument register being R0 (addi Ra, R0, C).

st: Store Direct (00010xxxx0000xxxxxxxxxxxxxxxxxxxx)	M[C (sign-extended)] ← R[Ra] Direct addressing, Rb = R0	st	C, Ra
st: Store Indexed/Register Indirect (00010xxxxxxxxxxxxxxxxxxxxxxxxxxx)	M[R[Rb] + C (sign-extended)] ← R[Ra] Indexed addressing, Rb ≠ R0 If C = 0 → Register Indirect addressing	st	C(Rb), Ra

Arithmetic and Logical Instructions

Three-register instructions:

add: Add (00011xxxxxxxxxxx----------)	R[Ra] ← R[Rb] + R[Rc]	add	Ra, Rb, Rc
sub: Sub (00100xxxxxxxxxxx----------)	R[Ra] ← R[Rb] - R[Rc]	sub	Ra, Rb, Rc
and: AND (00101xxxxxxxxxxx----------)	R[Ra] ← R[Rb] ^ R[Rc]	and	Ra, Rb, Rc
or: OR (00110xxxxxxxxxxx----------)	R[Ra] ← R[Rb] $_{v}$ R[Rc]	or	Ra, Rb, Rc

ror: Rotate Right — Rotate right R[Rb] into R[Ra] by count in R[Rc] — ror Ra, Rb, Rc
(00111xxxxxxxxxxx----------)

rol: Rotate Left — Rotate left R[Rb] into R[Ra] by count in R[Rc] — rol Ra, Rb, Rc
(01000xxxxxxxxxxx----------)

shr: Shift Right — Shift right R[Rb] into R[Ra] by count in R[Rc] — shr Ra, Rb, Rc
(01001xxxxxxxxxxx----------)

shra: Shift Right Arithmetic — Shift right arithmetic R[Rb] into R[Ra] by count in R[Rc] — shra Ra, Rb, Rc
(01010xxxxxxxxxxx----------)

shl: Shift Left — Shift left R[Rb] into R[Ra] by count in R[Rc] — shl Ra, Rb, Rc
(01011xxxxxxxxxxx----------)

Two-register with immediate value instructions:

addi: Add Immediate — R[Ra] ← R[Rb] + C (sign-extended) — addi Ra, Rb, C
(01100xxxxxxxxxxxxxxxxxxxxxxxxxx)
Immediate addressing
If C = 0 → instruction acts like a simple register transfer
If C ≠ 0 and Ra = Rb → Increment/decrement instruction
Similar to ldi, however Rb can be any register

andi: AND Immediate — R[Ra] ← R[Rb] ∧ C (sign-extended) — andi Ra, Rb, C
(01101xxxxxxxxxxxxxxxxxxxxxxxxxx)
Immediate addressing

ori: OR Immediate — R[Ra] ← R[Rb] $_{\vee}$ C (sign-extended) — ori Ra, Rb, C
(01110xxxxxxxxxxxxxxxxxxxxxxxxxx)
Immediate addressing

Two-register without immediate value instructions:

div: Divide — HI, LO ← R[Ra] ÷ R[Rb] — div Ra, Rb
(01111xxxxxxxx----------)

mul: Multiply — HI, LO ← R[Ra] × R[Rb] — mul Ra, Rb
(10000xxxxxxxx----------)

neg: Negate — R[Ra] ← - R[Rb] — neg Ra, Rb
(10001xxxxxxxx----------)

not: NOT — R[Ra] ← $\overline{R[Rb]}$ — not Ra, Rb
(10010xxxxxxxx----------)

Conditional Branch Instructions

brzr, brnz, brmi, brpl

Branch (10011xxxx--xxxxxxxxxxxxxxxxxxxx)

PC ← PC + 1 + C (sign-extended) if R[Ra] meets the condition

Condition: --00: branch if zero
--01: branch if nonzero
--10: branch if positive
--11: branch if negative

brzr	Ra, C
brnz	Ra, C
brpl	Ra, C
brmi	Ra, C

Jump Instructions

jal: jump and link (10100xxxx----------)

R[15] ← PC + 1
PC ← R[Ra]

jal Ra

jr: return from procedure (10101xxxx----------)

PC ← R[Ra]
If Ra = R15, it is for procedure return

jr Ra

Input/Output and MFHI/MFLO (Move From HI, Move From LO) Instructions

in: Input (10110xxxx----------)	R[Ra] ← In.Port	in Ra
out: Output (10111xxxx----------)	Out.Port ← R[Ra]	out Ra
mflo: Move from LO (11000xxxx----------)	R[Ra] ← LO	mflo Ra
mfhi: Move from HI (11001xxxx----------)	R[Ra] ← HI	mfhi Ra

Miscellaneous Instructions

nop: No-operation (11010----------)	Do nothing	nop
halt: Halt (11011----------)	Halt the control stepping process	halt

Project Design and Implementation

The designed processor features integer arithmetic instructions including addition, subtraction, multiplication, and division.

The complete design can be broken up into the processor, consisting of the Control Unit and Data Path, as well as the interfaces.

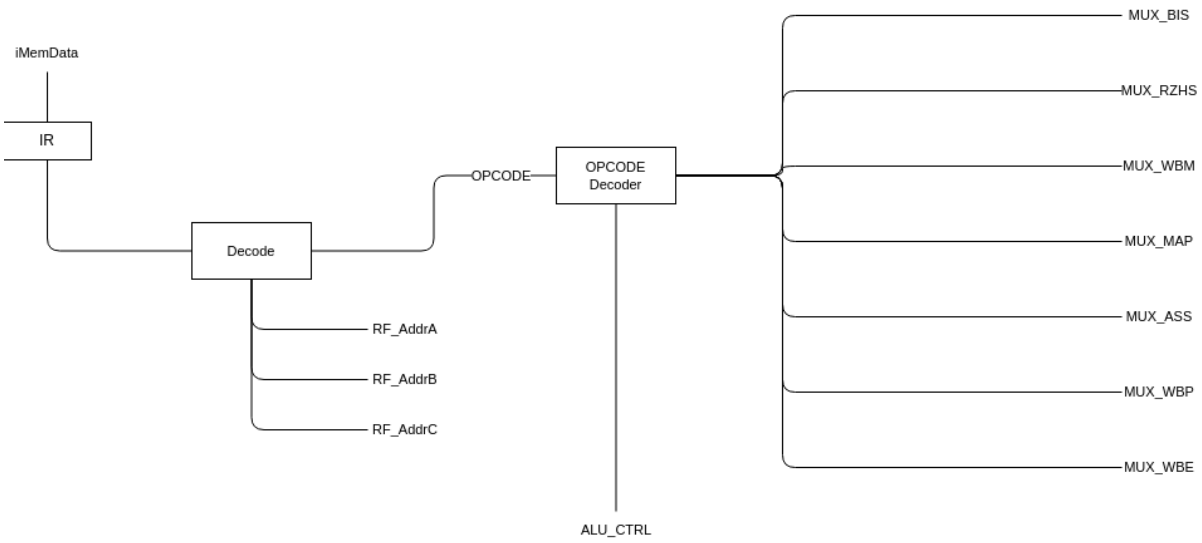
Control Unit

The control unit handles instruction decoding and issues control signals to the data path.

Every cycle of every instruction starts with the control unit, therefore, it must be as optimised as possible to reduce delay within the processor.

To minimise cycle times, the control unit is broken up into two stages: Decode and issue.

An illustration of the control unit is shown below.



Instruction Decode

The processor starts in cycle 1, fetching the next instruction. After fetching the instruction, it is loaded into the instruction register (IR). Then, a series of comparisons is made against the fields in the IR to create one hot signals corresponding to each instruction. The add instruction decode has been shown below for reference.

```Verilog
assign OP_ADD = (ID_OpCode == `ISA_ADD);
```

For code readability, all instructions were defined inside the `ISA.vh` file. In the above example, `ISA_ADD` is defined as `5'b00011`. To support a wide variety of instructions with fan-in constraints, instructions are further reduced into their types. This reduction is shown below for R-Type instructions.

```
// Opcode Format Wire (Useful for data path MUX Assignments)
assign OPF_R  = (OP_ADD || OP_SUB || OP_AND || OP_OR || OP_ROR || OP_ROL ||
OP_SRL || OP_SRA || OP_SLL);
```

Through reducing all instructions to one hot signals, control signal assignment can be done through a series of gates with minimal fan-in.

Data Path Control Signals

In addition to decoding instructions, the control unit must also issue control signals to the data path. Data path control signals consist of multiplexer select signals in addition to load enable signals for pipeline registers. Each signal must be asserted depending on the current cycle of

the processor and the type of instruction in the IR. The current cycle of the processor is tracked through a one-hot wraparound counter. As shown below, the counter will only increment if the processor is not halted and an external ready signal has been set to high.

```
// Assign Cycle
always @(posedge iClk or negedge nRst)
begin
    if(!nRst)
        Cycle = 5'b00001;
    else begin
        if(iRdy & ~OP_HLT) Cycle = {Cycle[4:1], Cycle[5]};
    end
end
```

In simulation, this signal was set to high as there are no memory delays. However, when the processor is connected to an external memory chip, the memory management unit (MMU) can assert this signal low while it waits for the memory data to become available.

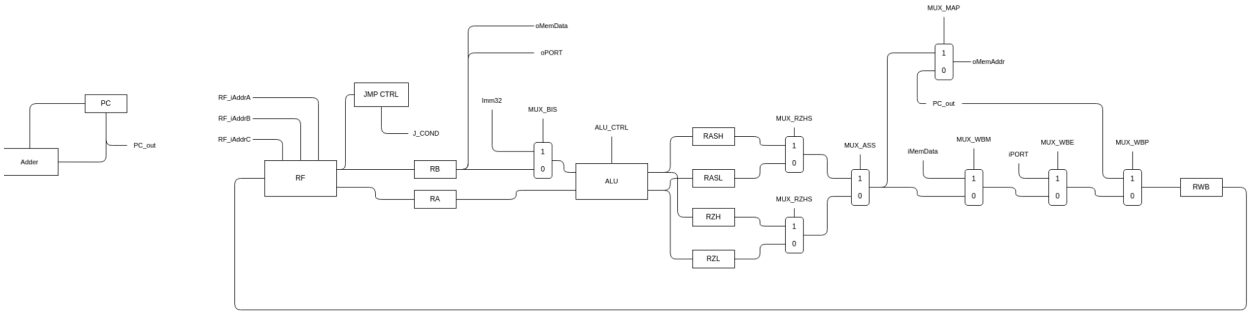
With the exception of the ALU (Arithmetic and Logic Unit) control signals, all control signals are a combination of one hot signals. This allows for a maximum delay of 8 gates between the instruction register and any control signal output. The control signal assignment for the program counter load enable is shown below.

```
// PC Load Enable
assign oPC_en = Cycle[1] || (Cycle[3] && (BR_TRUE || OP_JAL || OP_JFR));
```

The full control unit code can be found in the appendix.

Data Path

The data path contains the bulk of the processor. Housed within the data path is the Program Counter (PC), Register File (RF), and ALU. To illustrate the connections between these components, a diagram is shown below.



To shorten critical paths and to support future pipelining, registers are placed in between the functional units. Two input multiplexers are used to select the data supplied to and taken from these registers. Two input multiplexers were used to both reduce gate delay and to simplify the control signals. As seen in the following diagram, every multiplexer used in the data path can select between the previous selection or an external data source. This allows the use of one hot signals to select the data that is written back to the register files, simplifying the control unit.

Program Counter

In addition to housing the Program Counter (PC) register, the PC module also contains the PC adder and multiplexers used for branching. The PC adder is shown below. The adder is hardwired to always accept the current PC value as one of its inputs. The other input can be selected as either the default increment value or a branch offset.

```
// PC Adder Selection
assign add_in = iOffsetEn ? iOffset : `PC_INCREMENT;

CLA pc_adder(
    .iX(add_in),
    .iY(pc_out),
    .iCarry(1'b0),
    .oS(add_out),
    // Intentionally leave these ports unconnected
    .oCarry(),
    .oOverflow(),
    .oZero(),
    .oNegative()
);
```

To support function calls and jumping, the output of the adder is sent into another multiplexer before being loaded into the PC. The multiplexer, shown below, can be used to load the PC with either the output from the adder or an address from a register.

```
// PC Input Selection
assign pc_in  = iLoadEn ? iLoad : add_out;

REG32 #(.RESET(StartAddr)) pc(
    .iClk(iClk),
    .nRst(nRst),
    .iEn(iEn),
    .iD(pc_in),
    .oQ(pc_out)
);
```

Register File

As per the ISA, the register file has 16 registers, including R0, the zero register. The register file is made up of 15 register units. Each unit has its own data output and write signal.

```
REG32 r1(  .iClk(iClk), .nRst(nRst), .iEn(r1_write),  .iD(iRegC), .oQ(r1_out)  );
```

Two multiplexers are used to select the two source registers. Based on the RA and RB address inputs, the register data is provided onto the RA and RB outputs. The output assignment for RA is shown below.

```
// Output Register A assignment
assign oRegA =  (iAddrA == 4'b0001) ? r1_out  :
                (iAddrA == 4'b0010) ? r2_out  :
                (iAddrA == 4'b0011) ? r3_out  :
                (iAddrA == 4'b0100) ? r4_out  :
                (iAddrA == 4'b0101) ? r5_out  :
                (iAddrA == 4'b0110) ? r6_out  :
                (iAddrA == 4'b0111) ? r7_out  :
                (iAddrA == 4'b1000) ? r8_out  :
                (iAddrA == 4'b1001) ? r9_out  :
                (iAddrA == 4'b1010) ? r10_out :
                (iAddrA == 4'b1011) ? r11_out :
                (iAddrA == 4'b1100) ? r12_out :
                (iAddrA == 4'b1101) ? r13_out :
                (iAddrA == 4'b1110) ? r14_out :
                (iAddrA == 4'b1111) ? r15_out :
                32'd0;
```

As shown above, the multiplexer returns zero if no register is selected. This effectively implements the zero register, R0.

The write-back data is supplied to all registers, however, only the destination registers load enable (LE) signal (iWrite) is asserted in cycle 5.

```
// Write Signal Assert
assign r1_write  = (iAddrC == 4'b0001) && iWrite;
assign r2_write  = (iAddrC == 4'b0010) && iWrite;
assign r3_write  = (iAddrC == 4'b0011) && iWrite;
assign r4_write  = (iAddrC == 4'b0100) && iWrite;
assign r5_write  = (iAddrC == 4'b0101) && iWrite;
...
```

Note that all of the load enable signals depend on the iWrite input. This is to allow the register file to not write to any register, a useful feature for instructions like store that do not return any result.

ALU

The ALU, or Arithmetic and Logic Unit, houses all of the integer arithmetic units including an adder, divider, multiplier and several bit manipulation units. The ALU produces a 64-bit output dependent on which ALU operation signals are asserted by the control unit. A large multiplexer is used to select this output.

Adder

The adder consists of four 8-bit Carry Look ahead Adders (CLAs). Each CLA is connected through a ripple carry mechanism. The Adder can also be used for the subtraction and negate instructions.

Subtraction is performed through a twos complement of the second input and by setting the carry bit before performing the addition. The negation instruction was implemented through a subtraction of 0-RA. The multiplexers used to support this behaviour are shown below:

```
assign cla_iA = (iCtrl == `CTRL_ALU_NEG) ? 32'd0 : iA;

// XOR input B for subtraction and set carry to 1
assign cla_iB = (iCtrl == `CTRL_ALU_SUB) ? 32'hFFFFFFFF ^ iB :
                (iCtrl == `CTRL_ALU_NEG) ? 32'hFFFFFFFF ^ iA : iB;

assign cla_iCarry = (iCtrl == `CTRL_ALU_SUB || iCtrl == `CTRL_ALU_NEG);
```

Divider

The divider performs the non-restoring division algorithm using a 32-bit array divider. The divider was constructed using a generate for the first 31 stages. The final stage of the divider was implemented separately to perform the final restore needed in the non-restoring division algorithm. See full divider code in the appendix.

Multiplier

The Multiplier is capable of performing multiplication of 2, 32-bit signed 2's-complement numbers. The algorithm employs a 2-bit booth encoding to reduce the number of summands from 32 to 16. The summands are then left shifted accordingly and sign extended to align to a width of 64 bits. The 16 summands are then added together using 3 layers of 4-to-2 carry-save adders (https://www.geoffknagge.com/fyp/carrysave.shtml). The final 2 summands output from the final 4-to-2 carry-save adder are added using a carry-propagate adder to produce the final product.

The multiplier could be optimised to use fewer gates (but not less propagation delay) by narrowing the width of the 4-to-2 carry-save adders by only sign extending enough to meet the widths of the 4 inputs to each individual 4-to-2 reducer and directing the less significant bits of the partial sums directly to the product. It could be even further optimised at a system-level in combination with the ALU by utilising the same adder to add the final 2 summands.

The multiplier was implemented with MUL32 as the top level module, which was composed of an instance of BoothEncode_2bit_Nbit module and multiple Reducer4to2_Nbit modules through the use of generate blocks. The BoothEncode_2bit_Nbit module is simply a composition of (N+1)/2 (floor division) BoothEncode_2bit modules applied to a bit vector of length N, and similarly for the Reducer4to2_Nbit module.

Full multiplier code is shown in the appendix.

Bit Manipulation Units

In addition to the above units, the ALU also contains several bit manipulation units to perform bit-wise operations such as and, or, xor, shifting, and rotate. The and, or, and xor units were created using Verilog primitives. Shifting and rotation were implemented using a series of multiplexers. Each multiplexer can select between the result of the last multiplexer or the result but shifted. This implementation, shown below, allows for simple and fast bit manipulation with minimal gate usage.

```
module SHIFT_RIGHT_ARITH(
    iD, iShamt,
    oD
);

input wire [31:0] iD;
input wire [4:0]  iShamt;
output wire [31:0] oD;

wire [31:0] S1, S2, S3, S4, S5;

assign S1 = iShamt[0] ? {{1{iD[31]}}, iD[31:1]}   : iD;
assign S2 = iShamt[1] ? {{2{S1[31]}}, S1[31:2]}   : S1;
assign S3 = iShamt[2] ? {{4{S2[31]}}, S2[31:4]}   : S2;
assign S4 = iShamt[3] ? {{8{S3[31]}}, S3[31:8]}   : S3;
assign S5 = iShamt[4] ? {{16{S4[31]}}, S4[31:16]} : S4;

assign oD = S5;

endmodule
```

The above code snippet showcases the Shift Right Arithmetic operation. All other shifting modules follow the same format but with different bit manipulation logic in the concatenation portion.

Interfaces

Interfaces are external inputs and outputs that are readable and/or controllable by the processor. The designed processor contains a memory-mapped VGA interface and two port-mapped interfaces. The two port-mapped interfaces are the eight seven-segment displays and the seventeen switches available on the DE2 development board.

VGA Interface

To reduce the complexity of the VGA interface, it is set to a fixed resolution and only controlled by four registers. Each register can be set by the processor to set the colour of its quadrant on the display. The VGA interface then displays the four colours at a 640 by 480 resolution. The four registers used by VGA can be written to by setting the 30th bit of the memory address. The lowest two bits of the address are then used to select the VGA register to write to. Memory decoding and interface selection are done through the following code:

```
// Memory Select Wires
wire mems_dimem, mems_vga;

assign mems_dimem = proc_mem_addr[31:30] == 2'b00;
assign mems_vga = proc_mem_addr[31:30] == 2'b01;

memory mem(
    .iClk(Clk),
    .iRead(proc_mem_read & mems_dimem),
    .iWrite(proc_mem_write & mems_dimem),
    .iData(proc_mem_out),
    .iAddr(proc_mem_addr),
    .oData(proc_mem_in)
);

vga_interface vga(
    .iClk_50(iClk_50),
    .nRst(nRst),
    .iCR(proc_mem_out),
    .iAddr(proc_mem_addr),
    .iWrite(proc_mem_write & mems_vga),
    .oVGA_R(oVGA_R),
    .oVGA_G(oVGA_G),
    .oVGA_B(oVGA_B),
    .oVGA_Clk(oVGA_Clk),
    .oVGA_Blank(oVGA_Blank),
    .oVGA_HSync(oVGA_HSync),
    .oVGA_VSync(oVGA_VSync),
    .oVGA_Sync(oVGA_Sync)
);
```

Port Mapped Interfaces

Using the “in” and “out” instructions from the MiniSRC ISA, the switch states can be loaded, and the seven-segment displays can be written to. The “in” instruction loads the value from the switches. As shown below, the lower 18 bits contain 1's or 0's depending on the state of each switch. The top 14 bits are always zero.

```
assign proc_port_in = {14'd0, iSW};
assign proc_port_in = {14'd0, iSW};
```

The out instruction writes to the 8 seven-segment displays. Each display can show hex 0 through F. With 8 displays, the full 32-bit number written to the out port can be displayed in hex.

As shown below, each segment displays four bits of information.

```
SevenSeg s1(
    .iNum(iNum[3:0]),
    .oSeg(oSeg1)
);
```

Then, the four-bit number is mapped onto the display with a case statement.

Evaluation Results

After loading the design onto the DE2 FPGA development board, it was tested to find the maximum clock frequency and logic element usage. Maximum clock frequency was determined by running an integer computation program with multiplication and division. If the program returned the correct computational result, then the frequency was increased. The design was tested at 100Hz, 25kHz, 25MHz, and then at 50MHz. With all clock frequencies, the program returned the correct result. Therefore, the maximum operating frequency of the CPU is greater than or equal to 50MHz. Frequencies greater than 50MHz were not tested as 50MHz is the maximum available clock frequency on the DE2 development board. To accomplish a CPI (Clocks Per Instruction) of 5 at 50MHz, the design uses approximately 7000 logic elements, or 20% of the Cyclone II's number of total logic elements.

Discussion, Conclusions, and Future Work

The initial goal of this project was to design and implement a fully functional five-stage pipelined processor consisting of Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory (MEM), and Write Back (WB) stages. While each of these stages has been successfully implemented independently, the pipeline integration remains incomplete. Currently, the project provides a robust foundation upon which full pipelining can be developed and finalized. Preliminary pipeline stage registers have been established to facilitate future pipelining; however, one critical component still missing is the implementation of pipeline registers for control signals.

To achieve fully pipelined operation, several steps are recommended for future development. First, it will be necessary to incorporate hazard detection, either through compilation techniques or hardware-based detection. Hazard detection logic will introduce stalls or insert NOP (no operation) instructions whenever dependencies are encountered that cannot be resolved through forwarding.

Secondly, the addition of multiple forwarding units will be essential, specifically at the EX and MEM stages, and potentially at the ID stage. Forwarding from the WB stage can be managed

directly through the register file. Forwarding at the ID stage may become important depending on whether branch decisions remain in the EX stage or are relocated earlier to the ID stage. Moving branch decisions to the ID stage would reduce wasted clock cycles caused by pipeline flushes due to incorrect branch predictions.

In the initial implementation of branch instructions, no subsequent instructions will be issued until branch targets have been determined, thereby eliminating the need for pipeline flushing. Once the pipeline is verified to be fully operational and stable, advanced features such as branch prediction and associated flush logic should be introduced to enhance efficiency and reduce pipeline stalls further.

Although the pipelined processor was not fully completed, this project can still be considered a success. The processor design and individual pipeline stages have been fully developed and validated, creating a robust and reliable base upon which future enhancements can be implemented. The groundwork laid in this project greatly simplifies further efforts towards achieving a fully operational 5-stage pipeline. By clearly identifying areas needing further attention, such as hazard detection, forwarding units, and control signal pipeline registers, future development is well-guided. Ultimately, the current state of the processor represents significant progress and provides an excellent starting point for continued improvement.

Appendices

Appendix 1: Verilog HDL Code

To browse and download the code for the project including test benches, see the github repository: https://github.com/Jchisholm204/MiniSRC. All the code for the processor is included in the sections below.

ALU code

ALU module

```
`include "../Control/ALU.vh"
module ALU(
    // 32 bit Inputs
    iA, iB,
    // Control Signal
    iCtrl,
    // 64 bit output
    oC_hi, oC_lo,
    // Zero Output / Negative Output
    oZero, oNeg
);

input wire [31:0] iA, iB;
input wire [3:0] iCtrl;
output wire [31:0] oC_hi, oC_lo;
output wire oZero, oNeg;

// Module output
wire [31:0] out_hi, out_lo;

// Adder/Subtract/Not IO
wire [31:0] cla_out, cla_iB, cla_iA;
wire cla_iCarry, cla_oCarry, cla_overflow, cla_zero, cla_neg;

// OR/XOR/AND IO
wire [31:0] or_out, xor_out, and_out;

// Shifter IO
wire [31:0] sft_data, sft_out;
wire [4:0] sft_shamt;
wire sft_arith, sft_left;

// Multiplier IO
wire [63:0] mul_out;
wire mul_neg;

// Divider IO
wire [31:0] div_q, div_r, div_dividend, div_divisor;
wire [31:0] div_rmdr, div_qtnt;
wire div_iNegA, div_iNegB, div_neg;

// ROL / ROR IO
wire [31:0] ROR_out, ROL_out;
```

```
45  // NOT
46  wire [31:0] NOT_out;
47
48  // Adder/Subtract
49
50  // XOR input B for subtraction and set carry to 1
51  assign cla_iA = (iCtrl == `CTRL_ALU_NEG) ? 32'd0 : iA;
52  // assign cla_iA = (iCtrl == `CTRL_ALU_NEG) ? 32'd0 : iA;
53
54  assign cla_iB = (iCtrl == `CTRL_ALU_SUB) ? 32'hFFFFFFFF ^ iB :
55                  (iCtrl == `CTRL_ALU_NEG) ? 32'hFFFFFFFF ^ iA : iB;
56
57  assign cla_iCarry = (iCtrl == `CTRL_ALU_SUB || iCtrl == `CTRL_ALU_NEG);
58
59  CLA cla(
60      .iX(cla_iA),
61      .iY(cla_iB),
62      .iCarry(cla_iCarry),
63      .oS(cla_out),
64      .oCarry(cla_oCarry),
65      .oOverflow(cla_overflow),
66      .oZero(cla_zero),
67      .oNegative(cla_neg)
68  );
69
70  // OR
71  ALU_OR bor(
72      .iA(iA),
73      .iB(iB),
74      .oC(or_out)
75  );
76
77  // XOR
78  ALU_XOR bxor(
79      .iA(iA),
80      .iB(iB),
81      .oC(xor_out)
82  );
83
84  // AND
85  ALU_AND band(
86      .iA(iA),
87      .iB(iB),
88      .oC(and_out)
89  );
```

```
91   // Bit Shifter
92   assign sft_data = iA;
93   assign sft_shamt = iB[4:0];
94   // Negate Arithmetic shift (logic low)
95   assign sft_arith = ~(iCtrl == `CTRL_ALU_SRA);
96   assign sft_left  = ~(iCtrl == `CTRL_ALU_SLL);
97
98   SHIFT sft(
99       .iD(sft_data),
100      .iShamt(sft_shamt),
101      .nArith(sft_arith),
102      .nLeft(sft_left),
103      .oD(sft_out)
104  );
105
106  // Multiplier
107
108  MUL32 mul(
109      .iA(iA),
110      .iB(iB),
111      .oP(mul_out)
112  );
113
114  // assign mul_out = iA * iB;
115  assign mul_neg = mul_out[63];
116
117  // Divider
118  assign div_iNegA = iA[31];
119  assign div_iNegB = iB[31];
120
121  // If either is negative, the quotient must be negative
122  assign div_neg = div_iNegA ^ div_iNegB;
123
124  // If the divisor or dividend is negative, make it positive
125  assign div_dividend = div_iNegA ? 32'hFFFFFFFF ^ (iA - 1) : iA;
126  assign div_divisor = div_iNegB ? 32'hFFFFFFFF ^ (iB - 1) : iB;
```

```
128  DIV32 div(
129      .iQ(div_divisor),
130      .iD(div_dividend),
131      .oQ(div_q),
132      .oR(div_r)
133  );
134
135  // Quotient will be negative if A or B is negative but not both
136  assign div_qtnt = div_neg ? (32'hFFFFFFFF ^ div_q) + 1 : div_q;
137  // Remainder carries the same sign as the dividend
138  assign div_rmdr = div_iNegB ? (32'hFFFFFFFF ^ div_r) + 1 : div_r;
139
140  // ROR
141  ROR ror(
142      .iD(iA),
143      .iShamt(iB[4:0]),
144      .oD(ROR_out)
145  );
146
147  // ROL
148  ROL rol(
149      .iD(iA),
150      .iShamt(iB[4:0]),
151      .oD(ROL_out)
152  );
153
154  // NOT
155  generate
156      genvar i;
157      for(i = 0; i < 32; i = i + 1) begin : alu_not_gen
158          not (NOT_out[i], iA[i]);
159      end
160  endgenerate
```

```
// Module Outputs
// Set low output register
assign out_lo = (iCtrl == `CTRL_ALU_ADD) ? cla_out :
                (iCtrl == `CTRL_ALU_SUB) ? cla_out :
                (iCtrl == `CTRL_ALU_OR)  ? or_out  :
                (iCtrl == `CTRL_ALU_XOR) ? xor_out :
                (iCtrl == `CTRL_ALU_AND) ? and_out :
                (iCtrl == `CTRL_ALU_MUL) ? mul_out[31:0] :
                (iCtrl == `CTRL_ALU_DIV) ? div_qtnt :
                (iCtrl == `CTRL_ALU_SLL) ? sft_out :
                (iCtrl == `CTRL_ALU_SRL) ? sft_out :
                (iCtrl == `CTRL_ALU_SRA) ? sft_out :
                (iCtrl == `CTRL_ALU_ROR) ? ROR_out :
                (iCtrl == `CTRL_ALU_ROL) ? ROL_out :
                (iCtrl == `CTRL_ALU_NOT) ? NOT_out :
                (iCtrl == `CTRL_ALU_NEG) ? cla_out :
                32'h00000000;

// Set high output register (Zero on anything not needing 64 bits)
assign out_hi = (iCtrl == `CTRL_ALU_MUL) ? mul_out[63:32] :
                (iCtrl == `CTRL_ALU_DIV) ? div_rmdr :
                32'h00000000;

// Output register
assign oC_lo = out_lo;
assign oC_hi = out_hi;

// Assign the negative outputs based on the control inputs
assign oNeg =   (iCtrl == `CTRL_ALU_ADD) ? cla_neg :
                (iCtrl == `CTRL_ALU_SUB) ? cla_neg :
                (iCtrl == `CTRL_ALU_MUL) ? mul_neg :
                (iCtrl == `CTRL_ALU_DIV) ? div_neg :
                out_lo[31];

// Assign zero if both the high and low registers are zero
assign oZero = ({out_hi, out_lo} == 64'd0);


endmodule
```

AND module

```
module ALU_AND(
    iA, iB, oC
);
input wire [31:0] iA, iB;
output wire [31:0] oC;

generate
    genvar i;
    for(i = 0; i < 32; i = i + 1) begin : alu_and_gen
        and (oC[i], iA[i], iB[i]);
    end
endgenerate

endmodule
```

CLA module

```
// Carry Lookahead Adder
module CLA(
    iX,
    iY,
    iCarry,
    oS,
    oCarry,
    oOverflow,
    oZero,
    oNegative
);

input wire iCarry;
input wire [31:0] iX, iY;
output wire [31:0] oS;
output wire oCarry, oOverflow, oZero, oNegative;

wire [3:0] C, O;

assign C[0] = iCarry;

CLA8 add1(
    .iX(iX[7:0]),
    .iY(iY[7:0]),
    .iCarry(C[0]),
    .oCarry(C[1]),
    .oS(oS[7:0]),
    .oOverflow()
);

CLA8 add2(
    .iX(iX[15:8]),
    .iY(iY[15:8]),
    .iCarry(C[1]),
    .oCarry(C[2]),
    .oS(oS[15:8]),
    .oOverflow()
);
```

```
CLA8 add3(
    .iX(iX[23:16]),
    .iY(iY[23:16]),
    .iCarry(C[2]),
    .oCarry(C[3]),
    .oS(oS[23:16]),
    .oOverflow()
);

CLA8 add4(
    .iX(iX[31:24]),
    .iY(iY[31:24]),
    .iCarry(C[3]),
    .oCarry(oCarry),
    .oS(oS[31:24]),
    .oOverflow(oOverflow)
);

assign oZero = ~|oS;
assign oNegative = oS[31];
assign oOverflow = O[3];

endmodule

module CLA8(iX, iY, iCarry, oCarry, oS, oOverflow);
input wire [7:0] iX, iY;
input wire iCarry;
output wire oCarry, oOverflow;
output wire [7:0] oS;

wire [7:0] G, P;
wire [8:0] C;

// Generate the Generate and Propagate Signals
assign G = iX & iY;
assign P = iX | iY;
```

DIV32 module

```
// ONLY Accepts positive numbers
module DIV32(
    iQ, iD,
    oQ, oR
);

input wire [31:0] iQ, iD;
output wire [31:0] oQ, oR;

wire [31:0] Q;
wire [31:0] A[31:0];

// Initialize the first partial remainder as 0
assign A[0] = 32'd0;

// Create 31 divisor levels
generate
    genvar i;
    for(i = 0; i < 31; i = i + 1) begin : gen_div_lvl
        DIV_LEVEL divlvl (
            .iA(A[i]),
            .iQ(iQ[31-i]),
            .iD(iD),
            .oA(A[i+1]),
            .oQ(Q[31-i])
        );
    end
endgenerate

// Final divisor Level
wire [31:0] shift, X;

assign shift = {A[31][30:0], iQ[0]};
assign X = shift[31] ? shift + iD : shift - iD;
assign Q[0] = ~X[31];
```

```
// Final divisor Level
wire [31:0] shift, X;

assign shift = {A[31][30:0], iQ[0]};
assign X = shift[31] ? shift + iD : shift - iD;
assign Q[0] = ~X[31];

// Remainder and output assignments
assign oR = X[31] ? X + iD : X;
assign oQ = Q;

endmodule

module DIV_LEVEL(
    iA, iQ, iD,
    oA, oQ
);

input wire [31:0] iA, iD;
input wire iQ;
output wire [31:0] oA;
output wire oQ;

wire [31:0] shift;

// Shift in the next bit of the quotient
assign shift = {iA[30:0], iQ};
// Add or subtract the divisor
assign oA = shift[31] ? shift + iD : shift - iD;
// Output 1 to result if the add/sub result was positive
assign oQ = ~oA[31];

endmodule
```

MUL32 module

```
// TODO: Implement the ability to do this with unsigned numbers also.
// TODO: Combine all the carry-save adder generate blocks into one generate block.
// EXTRA: index the carry-save adder structure so this multiplier works for N being any power of 2.

// This assumes that the inputs are 32-bit 2's-complement signed integers.
// Multiply (multiplicand) A by (multiplier) B to get (product) P.
module MUL32 (input signed [31:0] iA, input signed [31:0] iB, output signed [63:0] oP);

localparam N = 32;

// The number of booth encoded values.
localparam BTH = (N+1)/2; // = 16
// For an ideal cascade of 4-to-2 reducers to align with the number of inputs, we need N to be a power of 2.
// Number of levels of Carry-Save Addition (CSA) using 4-to-2 reducers.
// localparam CSA_levels = $clog2(BTH); // = 4
// # of 4-to-2 reducers in each level of CSA.
// localparam reducers_in_CSA_level1 = BTH/4; // = 4
// localparam reducers_in_CSA_level2 = CSA_level1*2/4; // = 2
// localparam reducers_in_CSA_level3 = CSA_level2*2/4; // = 1
// Final Carry-Propagate Addition (CPA) is done using the final 2 outputs from the last level of CSA.

// Extend by 1 bit to handle negation
wire signed [N+1:0] A, negA;
wire signed [N+1:0] A2, negA2;
wire [N+1:0] notA;
wire [BTH-1:0] booth_sign;
wire [1:0] booth_magnitude [BTH-1:0];
wire [2*BTH-1:0] flat_booth_magnitude;

assign A = {{2{iA[N-1]}}, iA};
assign negA = notA + 1'b1;
assign A2 = {iA[N-1], iA, 1'b0};
assign negA2 = {negA[N:0], 1'b0};

// Compute the Booth Encoding of the multiplier (B).
genvar i;
generate
for (i = 0; i < N+2; i = i + 1) begin : gen_notA
    assign notA[i] = ~A[i];
end
for (i = 0; i < BTH; i = i + 1) begin : map_booth_magnitude
    assign booth_magnitude[i] = {flat_booth_magnitude[i+BTH], flat_booth_magnitude[i]};
end
endgenerate
```

```
BoothEncode_2bit_Nbit #(N) be2bit(iB, booth_sign, flat_booth_magnitude[2*BTH-1:BTH], flat_booth_magnitude[BTH-1:0]);

wire signed [N+1:0] initialValue[BTH-1:0];

generate
    for (i = 0; i < BTH; i = i + 1) begin : gen_initialValue
        assign initialValue[i] = booth_magnitude[i][1] ? (booth_sign[i] ? negA2 : A2)
            : (booth_magnitude[i][0] ? (booth_sign[i] ? negA : A)
            : {(N+1){1'b0}});
    end
endgenerate
/*
the s's in this diagram represent the sign bit of the initial values.
the z's in this diagram represent the padded zero bit of the initial values.
        00000000
ssssss0000000000
ssss0000000000zz
ss0000000000zzzz
0000000000zzzzzz
*/
wire signed [2*N-1:0] shiftedInitialValue[BTH-1:0];
generate
    for (i = 0; i < BTH; i = i + 1) begin : gen_shiftedInitialValue
        assign shiftedInitialValue[i] = {{(N-2*i){initialValue[i][N+1]}}, initialValue[i], {(2*i){1'b0}}};
    end
endgenerate
// CSA Layer 1: 16 numbers (34-bit each+shifts) -> 4*4-to-2 reducers. -> 8 numbers (64-bit each).
// Carry-Save Addition using 4-to-2 reducers.
/* same as this but with 32-bit numbers.
0 0000 0000 0000 0000
------00000000000000000 i = 0, j = 0
----00000000000000000--        j = 1
--00000000000000000----        j = 2
00000000000000000------        j = 3
..................... i = 1, j = 0
*/
// need to make sure that sign extension is done properly.
// The lowest 2 bits are piped directly to the output.
// We can ignore the upper bit from each reducer because it is place value 2*N, and the final result (oP) is only 2*N bits (not 2*N+1).
wire signed [2*N-1:2] iCSA1[3:0][3:0];
wire signed [2*N:2] oCSA1[3:0][1:0];
genvar j;
```

```
generate
    // i is the index of each reducer in the first layer.
    for (i = 0; i < 4; i = i + 1) begin : gen_CSA1
        for (j = 0; j < 4; j = j + 1) begin : gen_CSA1_shift
            assign iCSA1[i][j] = shiftedInitialValue[4*i+j][2*N-1:2];
        end
        Reducer4to2_Nbit #(2*N-2) CSA1_i(
            .iW(iCSA1[i][3]), .iX(iCSA1[i][2]), .iY(iCSA1[i][1]), .iZ(iCSA1[i][0]), .iCarry(1'b0),
            .oSum1(oCSA1[i][1][2*N:3]), .oSum0(oCSA1[i][0][2*N:2]));
        assign oCSA1[i][1][2] = 1'b0;
    end
endgenerate

// CSA Layer 2: 8 numbers -> 2*4-to-2 reducers. -> 4 numbers
wire [2*N-1:2] iCSA2[1:0][3:0];
wire [2*N:2] oCSA2[1:0][1:0];
generate
    for (i = 0; i < 2; i = i + 1) begin : gen_CSA2
        for (j = 0; j < 4; j = j + 1) begin : gen_CSA2_shift
            assign iCSA2[i][j] = oCSA1[2*i+j/2][j%2][2*N-1:2];
        end
        Reducer4to2_Nbit #(2*N-2) CSA2_i(
            .iW(iCSA2[i][3]), .iX(iCSA2[i][2]), .iY(iCSA2[i][1]), .iZ(iCSA2[i][0]), .iCarry(1'b0),
            .oSum1(oCSA2[i][1][2*N:3]), .oSum0(oCSA2[i][0][2*N:2]));
        assign oCSA2[i][1][2] = 1'b0;
    end
endgenerate

// CSA Layer 3: 4 numbers -> 1*4-to-2 reducer -> 2 numbers
wire [2*N-1:2] iCSA3[3:0];
wire [2*N:2] oCSA3[1:0];
generate
    for (i = 0; i < 4; i = i + 1) begin : gen_CAS3
        assign iCSA3[i] = oCSA2[i/2][i%2][2*N-1:2];
    end
endgenerate
Reducer4to2_Nbit #(2*N-2) CSA3(
    .iW(iCSA3[3]), .iX(iCSA3[2]), .iY(iCSA3[1]), .iZ(iCSA3[0]), .iCarry(1'b0),
    .oSum1(oCSA3[1][2*N:3]), .oSum0(oCSA3[0][2*N:2]));
assign oCSA3[1][2] = 1'b0;

// Carry-Propagate Addition using final 2 outputs from carry-save adders.
assign oP = {oCSA3[1][2*N-1:2] + oCSA3[0][2*N-1:2], initialValue[0][1:0]};
```

```
module BoothEncode_2bit_Nbit #(parameter N = 32) (input signed [N-1:0] iA, output [(N+1)/2-1:0] oSign,
output [(N+1)/2-1:0] oMagnitude1, output [(N+1)/2-1:0] oMagnitude0);

wire [N+1:0] A2;
assign A2 = {iA[N-1], iA, 1'b0};


genvar i;
generate
    for (i = 1; i+1 <= N+1; i = i + 2) begin : gen
        BoothEncode_2bit be2bit(A2[i+1:i-1], oSign[i/2], {oMagnitude1[i/2], oMagnitude0[i/2]});
    end
endgenerate
endmodule


// 0, 1, 2, -2, -1,
// 1 bit for sign: 0 = positive/zero, 1 = negative
// 2 bits for magnitude: 0 = 00, 1 = 01, 2 = 10
// table
//     // iA  : oSign  oMagnitude
//     3'b000 : 0      2'b00;
//     3'b001 : 0      2'b01;
//     3'b010 : 0      2'b01;
//     3'b011 : 0      2'b10;
//     3'b100 : 1      2'b10;
//     3'b101 : 1      2'b01;
//     3'b110 : 1      2'b01;
//     3'b111 : 0      2'b00; // making sure that there is no sign for zero.
// endtable
module BoothEncode_2bit(input [2:0] iA, output oSign, output [1:0] oMagnitude);

assign oSign = iA[2] & (~iA[1] | ~iA[0]);
assign oMagnitude[0] = iA[1] ^ iA[0];
assign oMagnitude[1] = (iA[2] && !(iA[1] || iA[0])) || (!iA[2] && (iA[1] && iA[0]));

endmodule

module Reducer4to2_Nbit #(parameter N = 32) (input [N-1:0] iW, input [N-1:0] iX, input [N-1:0] iY, input [N-1:0] iZ,
input iCarry, output [N-1:0] oSum1, output [N:0] oSum0);

wire carry [N:0];

assign carry[0] = iCarry;

genvar i;
generate
for (i = 0; i < N; i = i + 1) begin : gen
    Reducer4to2 r4to2({iW[i], iX[i], iY[i], iZ[i]}, carry[i], {oSum1[i], oSum0[i]}, carry[i+1]);
end
endgenerate

assign oSum0[N] = carry[N];

endmodule

// oCarry is of the same place value of the most significant bit of the sum. https://www.geoffknagge.com/fyp/carrysave.shtml
module Reducer4to2 (input [3:0] iA, input iCarry, output [1:0] oSum, output oCarry);
wire w, x, y, z;
assign {w, x, y, z} = iA;

// used Karnaugh Maps to simplify the equations (https://www.charlie-coleman.com/experiments/kmap/)

assign oSum[1] = (w & x & y & z) | (iCarry & (w ^ x ^ y ^ z));

assign oSum[0] = iCarry ^ w ^ x ^ y ^ z;

assign oCarry = (w | x | y) & (w | x | z) & (w | y | z) & (x | y | z);

endmodule
```

OR module

```
module ALU_OR(
    iA, iB, oC
);
input wire [31:0] iA, iB;
output wire [31:0] oC;

generate
    genvar i;
    for(i = 0; i < 32; i = i + 1) begin : alu_or_gen
        or (oC[i], iA[i], iB[i]);
    end
endgenerate

endmodule
```

ROL and ROR module

```
module ROL(
    iD, iShamt,
    oD
);

input wire [31:0] iD;
input wire [4:0]  iShamt;
output wire [31:0] oD;

wire [31:0] S1, S2, S3, S4, S5;

assign S1 = iShamt[0] ? {iD[30:0],  iD[31]}    : iD;
assign S2 = iShamt[1] ? {S1[29:0],  S1[31:30]} : S1;
assign S3 = iShamt[2] ? {S2[27:0],  S2[31:28]} : S2;
assign S4 = iShamt[3] ? {S3[23:0],  S3[31:24]} : S3;
assign S5 = iShamt[4] ? {S4[15:0],  S4[31:16]} : S4;

assign oD = S5;

endmodule

module ROR(
    iD, iShamt,
    oD
);

input wire [31:0] iD;
input wire [4:0]  iShamt;
output wire [31:0] oD;

wire [31:0] S1, S2, S3, S4, S5;

assign S1 = iShamt[0] ? {iD[0],     iD[31:1]}  : iD;
assign S2 = iShamt[1] ? {S1[1:0],   S1[31:2]}  : S1;
assign S3 = iShamt[2] ? {S2[3:0],   S2[31:4]}  : S2;
assign S4 = iShamt[3] ? {S3[7:0],   S3[31:8]}  : S3;
assign S5 = iShamt[4] ? {S4[15:0],  S4[31:16]} : S4;

assign oD = S5;

endmodule
```

SHIFT module

```v
module SHIFT(
    iD, iShamt,
    nArith, nLeft,
    oD
);

input wire [31:0] iD;
input wire [4:0]  iShamt;
input wire nArith, nLeft;
output wire [31:0] oD;

wire [31:0] left, right_arith, right_logic, right;

SHIFT_LEFT leftshift(
    .iD(iD),
    .iShamt(iShamt),
    .oD(left)
);

SHIFT_RIGHT_ARITH rightarith(
    .iD(iD),
    .iShamt(iShamt),
    .oD(right_arith)
);

SHIFT_RIGHT_LOGIC rightlogic(
    .iD(iD),
    .iShamt(iShamt),
    .oD(right_logic)
);

assign right = nArith ? right_logic : right_arith;
assign oD = nLeft ? right : left;

endmodule

module SHIFT_LEFT(
    iD, iShamt,
    oD
);

input wire [31:0] iD;
input wire [4:0]  iShamt;
output wire [31:0] oD;
wire [31:0] S1, S2, S3, S4, S5;

assign S1 = iShamt[0] ? {iD[30:0],  1'd0} : iD;
assign S2 = iShamt[1] ? {S1[29:0],  2'd0} : S1;
assign S3 = iShamt[2] ? {S2[27:0],  4'd0} : S2;
assign S4 = iShamt[3] ? {S3[23:0],  8'd0} : S3;
assign S5 = iShamt[4] ? {S4[15:0], 16'd0} : S4;
assign oD = S5;

endmodule

module SHIFT_RIGHT_ARITH(
    iD, iShamt,
    oD
);

input wire [31:0] iD;
input wire [4:0]  iShamt;
output wire [31:0] oD;
wire [31:0] S1, S2, S3, S4, S5;
assign S1 = iShamt[0] ? {{1{iD[31]}}, iD[31:1]}   : iD;
assign S2 = iShamt[1] ? {{2{S1[31]}}, S1[31:2]}   : S1;
assign S3 = iShamt[2] ? {{4{S2[31]}}, S2[31:4]}   : S2;
assign S4 = iShamt[3] ? {{8{S3[31]}}, S3[31:8]}   : S3;
assign S5 = iShamt[4] ? {{16{S4[31]}}, S4[31:16]} : S4;
assign oD = S5;

endmodule

module SHIFT_RIGHT_LOGIC(
    iD, iShamt,
    oD
);

input wire [31:0] iD;
input wire [4:0]  iShamt;
output wire [31:0] oD;
wire [31:0] S1, S2, S3, S4, S5;
assign S1 = iShamt[0] ? {1'd0,  iD[31:1]}  : iD;
assign S2 = iShamt[1] ? {2'd0,  S1[31:2]}  : S1;
assign S3 = iShamt[2] ? {4'd0,  S2[31:4]}  : S2;
assign S4 = iShamt[3] ? {8'd0,  S3[31:8]}  : S3;
assign S5 = iShamt[4] ? {16'd0, S4[31:8]}  : S4;
assign oD = S5;
endmodule
```

XOR module

```v
module ALU_XOR(
    iA, iB, oC
);
input wire [31:0] iA, iB;
output wire [31:0] oC;

generate
    genvar i;
    for(i = 0; i < 32; i = i + 1) begin : alu_xor_gen
        xor (oC[i], iA[i], iB[i]);
    end
endgenerate

endmodule
```

Control Code

Control Module

```
`include "ISA.vh"
`include "ALU.vh"


module Control (
    // Clock, reset and ready signals
    // Ready is an active high that allows the next step to continue
    iClk, nRst, iRdy,
    // Memory Signals/Control
    iMemData, oMemRead, oMemWrite,
    // Pipe Control
    oPipe_nRst,
    // Program Counter Control
    oPC_nRst, oPC_en, oPC_tmpEn, oPC_load, oPC_offset,
    // Register File Control
    oRF_Write,
    oRF_AddrA, oRF_AddrB, oRF_AddrC,
    // Write Back Register Control
    oRWB_en,
    // ALU Control
    oALU_Ctrl, oRA_en, oRB_en,
    oRZH_en, oRZL_en, oRAS_en,
    // Jump Feedback
    iJ_zero, iJ_nZero, iJ_pos, iJ_neg,
    // Port Register enable
    oREP_en,
    // Multiplexers
    oMUX_BIS, oMUX_RZHS, oMUX_WBM, oMUX_MAP, oMUX_ASS, oMUX_WBP, oMUX_WBE,
    // Imm32 Output
    oImm32
);

// Clock, reset and ready signals
// Ready is an active high that allows the next step to continue
input wire iClk, nRst, iRdy;
// Memory Signals/Control
input wire [31:0] iMemData;
output wire oMemRead, oMemWrite;
// Pipe Control
output wire oPipe_nRst;
// Program Counter Control
output wire oPC_nRst, oPC_en, oPC_tmpEn, oPC_load, oPC_offset;
// Register File Control
output wire oRF_Write;
output wire [3:0] oRF_AddrA, oRF_AddrB, oRF_AddrC;
```

```
46  // Write Back Register Control
47  output wire oRWB_en;
48  // ALU Control
49  output wire [3:0] oALU_Ctrl;
50  output wire oRA_en, oRB_en;
51  output wire oRZH_en, oRZL_en, oRAS_en;
52  // Jump Feedback
53  input wire iJ_zero, iJ_nZero, iJ_pos, iJ_neg;
54  // External Port Enable
55  output wire oREP_en;
56  // Multiplexers
57  output wire oMUX_BIS, oMUX_RZHS, oMUX_WBM, oMUX_MAP, oMUX_ASS, oMUX_WBP, oMUX_WBE;
58  // Imm32 Output
59  output wire [31:0] oImm32;
60
61  // Step Counter
62  reg [5:1] Cycle;
63
64  // IR
65  wire IR_en;
66  wire [31:0] IR_out;
67
68  // Decoder IO
69  wire [3:0] ID_RA, ID_RB, ID_RC;
70  wire [4:0] ID_OpCode;
71  wire [31:0] ID_imm32, ID_BRD;
72  wire [1:0] ID_BRC;
73
74  // OpCode R-Format Wires
75  wire OP_LD, OP_LI,  OP_ST,  OP_ADD, OP_SUB, OP_AND,
76       OP_OR, OP_ROR, OP_ROL, OP_SRL, OP_SRA, OP_SLL;
77  // OpCode I-Format Wires
78  wire OP_ADDI, OP_ANDI, OP_ORI, OP_DIV, OP_MUL, OP_NEG, OP_NOT;
79  // OpCode B-Format Wires
80  wire OP_BRx;
81  // OpCode J-Format Wires
82  wire OP_JAL, OP_JFR, OP_IN, OP_OUT, OP_MFL, OP_MFH;
83  // OpCode M-Format Wires
84  wire OP_NOP, OP_HLT;
85  // OpCode Format Wires
86  // (Useful for data path MUX Assignments)
87  wire OPF_R, OPF_I, OPF_B, OPF_J, OPF_M;
88  // Branch Conditional Wires
89  wire BR_ZERO, BR_NZRO, BR_POS, BR_NEG;
90  wire BR_TRUE;
```

```
92  // Assign Cycle
93  always @(posedge iClk or negedge nRst)
94  begin
95      if(!nRst)
96          Cycle = 5'b00001;
97      else begin
98          if(iRdy & ~OP_HLT) Cycle = {Cycle[4:1], Cycle[5]};
99      end
100 end
101
102 // Instruction Register
103 assign IR_en = Cycle[1];
104 REG32 IR(.iClk(iClk), .nRst(nRst), .iEn(IR_en), .iD(iMemData), .oQ(IR_out));
105
106 // Decoder
107 Decode decoder(
108     .iINS(IR_out),
109     .oImm32(ID_imm32),
110     .oRa(ID_RA),
111     .oRb(ID_RB),
112     .oRc(ID_RC),
113     .oCode(ID_OpCode),
114     // Branch Distance
115     .oBRD(ID_BRD),
116     // Branch Code
117     .oBRC(ID_BRC)
118 );
119
120 // Assign OP-Code Types
121
122 // Assign R-Format Wires
123 assign OP_ADD = (ID_OpCode == `ISA_ADD);
124 assign OP_SUB = (ID_OpCode == `ISA_SUB);
125 assign OP_AND = (ID_OpCode == `ISA_AND);
126 assign OP_OR  = (ID_OpCode == `ISA_OR);
127 assign OP_ROR = (ID_OpCode == `ISA_ROR);
128 assign OP_ROL = (ID_OpCode == `ISA_ROL);
129 assign OP_SRL = (ID_OpCode == `ISA_SRL);
130 assign OP_SRA = (ID_OpCode == `ISA_SRA);
131 assign OP_SLL = (ID_OpCode == `ISA_SLL);
132 // Opcode Format Wire (Useful for data path MUX Assignments)
133 assign OPF_R  = (OP_ADD || OP_SUB || OP_AND || OP_OR || OP_ROR || OP_ROL || OP_SRL || OP_SRA || OP_SLL);
134 // Assign I-Format Wires
135 assign OP_LD  = (ID_OpCode == `ISA_LD);
136 assign OP_LI  = (ID_OpCode == `ISA_LI);
```

```
137   assign OP_ST  = (ID_OpCode == `ISA_ST);
138   assign OP_ADDI = (ID_OpCode == `ISA_ADDI) || OP_LI;
139   assign OP_ANDI = (ID_OpCode == `ISA_ANDI);
140   assign OP_ORI  = (ID_OpCode == `ISA_ORI);
141   assign OP_DIV  = (ID_OpCode == `ISA_DIV);
142   assign OP_MUL  = (ID_OpCode == `ISA_MUL);
143   assign OP_NEG  = (ID_OpCode == `ISA_NEG);
144   assign OP_NOT  = (ID_OpCode == `ISA_NOT);
145   // Opcode Format Wire (Useful for data path MUX Assignments)
146   assign OPF_I   = (OP_LD || OP_ST || OP_ADDI || OP_ANDI || OP_ORI || OP_DIV || OP_MUL || OP_NEG || OP_NOT);
147   // Assign B-Format Wires
148   assign OP_BRx = (ID_OpCode == `ISA_BRx);
149   // Opcode Format Wire (Useful for data path MUX Assignments)
150   assign OPF_B = OP_BRx;
151   // Assign J-Format Wires
152   assign OP_JAL = (ID_OpCode == `ISA_JAL);
153   assign OP_JFR = (ID_OpCode == `ISA_JFR);
154   assign OP_IN  = (ID_OpCode == `ISA_IN);
155   assign OP_OUT = (ID_OpCode == `ISA_OUT);
156   assign OP_MFL = (ID_OpCode == `ISA_MFL);
157   assign OP_MFH = (ID_OpCode == `ISA_MFH);
158   // Opcode Format Wire (Useful for data path MUX Assignments)
159   assign OPF_J  = (OP_JAL || OP_JFR || OP_MFL || OP_MFH || OP_IN || OP_OUT);
160   // Assign M-Format Wires
161   assign OP_NOP = (ID_OpCode == `ISA_NOP);
162   assign OP_HLT = (ID_OpCode == `ISA_HLT);
163   // Opcode Format Wire (Useful for data path MUX Assignments)
164   assign OPF_M  =  (OP_NOP || OP_HLT);
165
166   // Assign Branch Wires
167   // iJ_xxx based on RF_RB in data path
168   assign BR_ZERO = (ID_BRC == `ISA_BR_ZERO) && iJ_zero;
169   assign BR_NZRO = (ID_BRC == `ISA_BR_NZRO) && iJ_nZero;
170   assign BR_POS = (ID_BRC == `ISA_BR_POSI) &&  iJ_pos;
171   assign BR_NEG = (ID_BRC == `ISA_BR_NEGA) &&  iJ_neg;
172   assign BR_TRUE = (BR_ZERO || BR_NZRO || BR_POS || BR_NEG) && OP_BRx;
173
174   // Assign Control outputs based on Codes and Cycle
175
176   // Pipe Reset Signal
177   assign oPipe_nRst = nRst;
178
179   // Program Counter Control Signals
180   // PC Reset (Should only be reset on CPU reset)
181   assign oPC_nRst = nRst;
```

```
// PC Load Enable
assign oPC_en = Cycle[1] || (Cycle[3] && (BR_TRUE || OP_JAL || OP_JFR));
assign oPC_tmpEn = Cycle[1];
// PC Jump Enable
assign oPC_offset = Cycle[3] && BR_TRUE;
assign oPC_load = Cycle[3] && (OP_JFR || OP_JAL);

// Register File Control Signals
assign oRF_Write = Cycle[5] && (OPF_R || (OPF_I && ~OP_DIV && ~OP_MUL && ~OP_ST) || OP_MFH || OP_MFL || OP_JAL || OP_IN);
// Note: Most ISA's use RC as the write back address, MiniSRC uses RA
// RA is dependent on ISA type, use R0 if RA is not specified
// RA is used to load PC on JMP/JAL
assign oRF_AddrA =  (OPF_R | OPF_I) ? ID_RB :
                    (OPF_J) ? ID_RA : 4'h0;
// RB is dependent on ISA type, use R0 if RB is not specified
assign oRF_AddrB =  (OPF_I | OPF_B | OPF_J) ? ID_RA :
                    (OPF_R) ? ID_RC : 4'h0;
// Store is always RA
// ISA Specification states to store PC in r15 on JAL (Jump and Link)
assign oRF_AddrC = (OP_JAL) ? 4'h8 : ID_RA;

// Register File Write Back Register Load Enable
assign oRWB_en = 1'b1;

// ALU Control Signals Also, this should be renamed to "Ctrl" like the key on the keyboard.
assign oALU_Ctrl =  (OP_ADD || OP_ADDI) ? `CTRL_ALU_ADD :
                    (OP_SUB)            ? `CTRL_ALU_SUB :
                    (OP_OR  || OP_ORI)  ? `CTRL_ALU_OR  :
                    (OP_AND || OP_ANDI) ? `CTRL_ALU_AND :
                    (OP_MUL)            ? `CTRL_ALU_MUL :
                    (OP_DIV)            ? `CTRL_ALU_DIV :
                    (OP_SLL)            ? `CTRL_ALU_SLL :
                    (OP_SRL)            ? `CTRL_ALU_SRL :
                    (OP_SRA)            ? `CTRL_ALU_SRA :
                    (OP_ROR)            ? `CTRL_ALU_ROR :
                    (OP_ROL)            ? `CTRL_ALU_ROL :
                    (OP_NOT)            ? `CTRL_ALU_NOT :
                    (OP_NEG)            ? `CTRL_ALU_NEG :
                    // ALU Add is default for most instructions - so why not remove the (OP_ADD || OP_ADDI) ?
                    `CTRL_ALU_ADD;
// ALU Input A Register Load Enable
assign oRA_en = 1'b1;
// ALU Input B Register Load Enable
assign oRB_en = 1'b1;
```

```
227   // ALU Result High Load EN
228   assign oRZH_en = 1'b1;
229   // ALU Result Low Load EN
230   assign oRZL_en = 1'b1;
231   // ALU Result Save EN
232   assign oRAS_en = (OP_DIV || OP_MUL);
233
234   // External Port Register Enable
235   assign oREP_en = OP_OUT && Cycle[4];
236
237   // ALU B Input Select (Selects Imm)
238   assign oMUX_BIS = OPF_I && ~(OP_DIV || OP_MUL);
239   // ALU Result High Select
240   assign oMUX_RZHS = (OP_MFH);
241   // RF Write Back Select
242   assign oMUX_WBM = (OP_LD);
243   // Memory Address Output Select
244   // assign oMUX_MA = Cycle[1];
245   assign oMUX_MAP = ~((OP_LD || OP_ST) && Cycle[4]);
246   // ALU Storage Select
247   assign oMUX_ASS = (OP_MFL || OP_MFH);
248   // Write Back Program Counter Select
249   assign oMUX_WBP = OP_JAL;
250   // Write Back External Port Select
251   assign oMUX_WBE = OP_IN;
252
253   // Immediate value output
254   // Assign Imm32 branch distance if the branch is true
255   assign oImm32 = OP_BRx ? ID_BRD : ID_imm32;
256
257   // Memory Read/Write Signals
258   assign oMemRead = Cycle[1] || (Cycle[4] && OP_LD);
259   assign oMemWrite = Cycle[4] && OP_ST;
260
261   endmodule
```

Decode Module

```
1  module Decode (
2      // Input Instruction
3      iINS,
4      // Immediate Value (Sign Extended)
5      oImm32,
6      // Reg A, Reg B, Reg C addresses
7      oRa, oRb, oRc,
8      // OP Code
9      oCode,
10     // Branch Distance
11     oBRD,
12     // Branch Code
13     oBRC
14 );
15
16 // Taken from instruction formats: section 2.1 of the Processor specifications
17
18 input wire [31:0] iINS;
19 output wire [31:0] oImm32;
20 output wire [3:0] oRa, oRb, oRc;
21 output wire [4:0] oCode;
22 output wire [31:0] oBRD;
23 output wire [1:0] oBRC;
24
25 assign oCode  = iINS[31:27];
26 assign oRa    = iINS[26:23];
27 assign oRb    = iINS[22:19];
28 assign oRc    = iINS[18:15];
29 assign oImm32 = {{13{iINS[18]}}, iINS[18:0]};
30 assign oBRD   = {{11{iINS[18]}}, iINS[18:0], 2'b00};
31 assign oBRC   = iINS[20:19];
32
33 endmodule
```

PC Module

```
module PC #(
    parameter StartAddr = 32'h00000000
) (
    iClk,
    iEn, iTmpEn, nRst,
    iLoadEn, iOffsetEn,
    iLoad, iOffset,
    oPC,
    oPC_tmp
);
`include "../constants.vh"

input wire iClk, iEn, iTmpEn, nRst;
input wire iLoadEn, iOffsetEn;
input wire [31:0] iLoad, iOffset;
output wire [31:0] oPC, oPC_tmp;

wire [31:0] pc_out, pc_tmp_out, add_in, add_out, pc_in;

// PC Adder Selection
assign add_in = iOffsetEn ? iOffset : `PC_INCREMENT;

CLA pc_adder(
    .iX(add_in),
    .iY(pc_out),
    .iCarry(1'b0),
    .oS(add_out),
    // Intentionally leave these ports unconnected
    .oCarry(),
    .oOverflow(),
    .oZero(),
    .oNegative()
);

// PC Input Selection
assign pc_in  = iLoadEn ? iLoad : add_out;

REG32 #(.RESET(StartAddr)) pc(
    .iClk(iClk),
    .nRst(nRst),
    .iEn(iEn),
    .iD(pc_in),
    .oQ(pc_out)
);
```

```
REG32 pc_tmp(
    .iClk(iClk),
    .nRst(nRst),
    .iEn(iTmpEn),
    .iD(add_out),
    .oQ(pc_tmp_out)
);

assign oPC = pc_out;
assign oPC_tmp = pc_tmp_out;

endmodule
```

Instruction and ALU encoding

```
// ALU Verilog Header File

// Control Parameters
`define CTRL_ALU_ADD 4'h0
`define CTRL_ALU_SUB 4'h1
`define CTRL_ALU_OR  4'h2
`define CTRL_ALU_XOR 4'h3
`define CTRL_ALU_AND 4'h4
`define CTRL_ALU_MUL 4'h5
`define CTRL_ALU_DIV 4'h6
`define CTRL_ALU_SLL 4'h7
`define CTRL_ALU_SRL 4'h8
`define CTRL_ALU_SRA 4'h9
`define CTRL_ALU_ROR 4'hA
`define CTRL_ALU_ROL 4'hB
`define CTRL_ALU_NOT 4'hC
`define CTRL_ALU_NEG 4'hD
```

```
// Opcode Signatures
// R Format Instructions
`define ISA_ADD  5'b00011
`define ISA_SUB  5'b00100
`define ISA_AND  5'b00101
`define ISA_OR   5'b00110
`define ISA_ROR  5'b00111
`define ISA_ROL  5'b01000
`define ISA_SRL  5'b01001
`define ISA_SRA  5'b01010
`define ISA_SLL  5'b01011
// I Format Instructions
`define ISA_LD   5'b00000
`define ISA_LI   5'b00001
`define ISA_ST   5'b00010
`define ISA_ADDI 5'b01100
`define ISA_ANDI 5'b01101
`define ISA_ORI  5'b01110
`define ISA_DIV  5'b01111
`define ISA_MUL  5'b10000
`define ISA_NEG  5'b10001
`define ISA_NOT  5'b10010
// B Format Instructions
`define ISA_BRx  5'b10011
// J Format Instructions
`define ISA_JAL  5'b10100
`define ISA_JFR  5'b10101
`define ISA_IN   5'b10110
`define ISA_OUT  5'b10111
`define ISA_MFL  5'b11000
`define ISA_MFH  5'b11001
// M Format Instructions
`define ISA_NOP  5'b11010
`define ISA_HLT  5'b11011

// Branch Codes
// Branch if Zero
`define ISA_BR_ZERO 2'b00
// Branch if NonZero
`define ISA_BR_NZRO 2'b01
// Branch if Positive
`define ISA_BR_POSI 2'b10
// Branch if negative
`define ISA_BR_NEGA 2'b11
```

Datapath module

```
module Datapath(
    // Clock and reset signals (reset is active low)
    iClk, nRst,
    // Memory Signals
    iMemData,
    oMemAddr, oMemData,
    // Port IO
    iPORT, oPORT,
    // Program Counter Control
    iPC_nRst, iPC_en, iPC_tmpEn, iPC_load, iPC_offset,
    // Register File Control
    iRF_Write,
    iRF_AddrA, iRF_AddrB, iRF_AddrC,
    // Write Back Register Control
    iRWB_en,
    // ALU Control
    iALU_Ctrl, iRA_en, iRB_en,
    iRZH_en, iRZL_en, iRAS_en,
    // Jump Feedback
    oJ_zero, oJ_nZero, oJ_pos, oJ_neg,
    // ALU Results
    oALU_neg, oALU_zero,
    // Memory Control
    // iRMA_en, iRMD_en,
    // Multiplexers
    iMUX_BIS, // ALU B Input/Immediate Select
    iMUX_RZHS, // ALU Result High Select
    iMUX_WBM, // Write back in Memory Select
    iMUX_MAP, // Memory Address out PC Select
    iMUX_ASS, // ALU Storage Select
    iMUX_WBP, // Write back Program Counter Select
    iMUX_WBE, // Write back in External Port Select
    // Imm32 Output
    iImm32
);

`include "constants.vh"

input wire iClk, nRst;
// Memory Signals
input wire [31:0] iMemData;
output wire [31:0] oMemData, oMemAddr;
// Port IO
input wire [31:0] iPORT;
output wire [31:0] oPORT;
// Program Counter Control
input wire iPC_nRst, iPC_en, iPC_tmpEn, iPC_load, iPC_offset;
// Register File Control
input wire iRF_Write;
input wire [3:0] iRF_AddrA, iRF_AddrB, iRF_AddrC;
// Write Back Register Control
input wire iRWB_en;
// ALU Control
input wire [3:0] iALU_Ctrl;
input wire iRA_en, iRB_en;
input wire iRZH_en, iRZL_en, iRAS_en;
output wire oALU_neg, oALU_zero;
// Jump Feedback
output wire oJ_zero, oJ_nZero, oJ_pos, oJ_neg;
// Multiplexers
input wire iMUX_BIS, iMUX_RZHS, iMUX_WBM, iMUX_MAP, iMUX_ASS, iMUX_WBP, iMUX_WBE;
// Imm32 Output
input wire [31:0] iImm32;

// Internal Clock Signal
wire Clk;
assign Clk = iClk & nRst;

// Program Counter Signals
wire [31:0] PC_out, PC_tOut;

// Register File IO
wire [31:0] RF_oRegA, RF_oRegB, RF_iRegC;
wire [31:0] RWB_in;

// ALU IO
wire [31:0] ALU_iA, ALU_iB, ALU_oC_hi, ALU_oC_lo;

// ALU Immediate Registers
wire [31:0] RA_out, RB_out;
wire [31:0] RZH_out, RZL_out, RZ_out;
// ALU Storage Registers
wire [31:0] RASH_out, RASL_out, RAS_out;
// ALU Output
wire [31:0] RZX_out;


// Program Counter
PC #(.StartAddr(`START_PC_ADDRESS)) pc(
    .iClk(Clk),
```

```
    .iEn(iPC_en),
    .iTmpEn(iPC_tmpEn),
    .nRst(iPC_nRst),
    .iLoadEn(iPC_load),
    .iOffsetEn(iPC_offset),
    .iLoad(RF_oRegA),
    .iOffset(iImm32),
    .oPC(PC_out),
    .oPC_tmp(PC_tOut)
);


// Register File
RegFile RF(
    .iClk(Clk),
    .nRst(nRst),
    .iWrite(iRF_Write),
    .iAddrA(iRF_AddrA),
    .iAddrB(iRF_AddrB),
    .iAddrC(iRF_AddrC),
    .oRegA(RF_oRegA),
    .oRegB(RF_oRegB),
    .iRegC(RF_iRegC)
);

// Jump Outputs
// RB is linked to RA in the ISA
assign oJ_zero = (RF_oRegB == 32'd0);
assign oJ_nZero = |RF_oRegB;
assign oJ_pos   = ~RF_oRegB[31] && oJ_nZero;
assign oJ_neg   = RF_oRegB[31] && oJ_nZero;

// RF stationary/buffer registers
REG32 RA(.iClk(Clk), .nRst(nRst), .iEn(iRA_en), .iD(RF_oRegA), .oQ(RA_out));
REG32 RB(.iClk(Clk), .nRst(nRst), .iEn(iRB_en), .iD(RF_oRegB), .oQ(RB_out));

// ALU Input Multiplexers

assign ALU_iA = RA_out;
assign ALU_iB = iMUX_BIS ? iImm32 : RB_out;

// ALU
ALU alu(
    .iA(ALU_iA),
```

```
    .iB(ALU_iB),
    .iCtrl(iALU_Ctrl),
    .oC_hi(ALU_oC_hi),
    .oC_lo(ALU_oC_lo),
    .oZero(oALU_zero),
    .oNeg(oALU_neg)
);

// ALU Result Registers
REG32 RZH(.iClk(Clk), .nRst(nRst), .iEn(iRZH_en), .iD(ALU_oC_hi), .oQ(RZH_out));
REG32 RZL(.iClk(Clk), .nRst(nRst), .iEn(iRZL_en), .iD(ALU_oC_lo), .oQ(RZL_out));

// ALU Storage Registers - Persist data until reset or next H/L transaction
REG32 RASH(.iClk(Clk), .nRst(nRst), .iEn(iRAS_en), .iD(ALU_oC_hi), .oQ(RASH_out));
REG32 RASL(.iClk(Clk), .nRst(nRst), .iEn(iRAS_en), .iD(ALU_oC_lo), .oQ(RASL_out));

// 32 bit ALU result selection
assign RAS_out = iMUX_RZHS ? RASH_out : RASL_out;
assign RZ_out  = iMUX_RZHS ? RZH_out : RZL_out;
// Select between storage or current registers
assign RZX_out = iMUX_ASS ? RAS_out : RZ_out;

// Memory
assign oMemAddr = iMUX_MAP ? PC_out : RZX_out ;
assign oMemData = RB_out;
    // iRF_AddrA, iRF_AddrB, iRF_AddrC,
// assign oPORT = {RF_oRegA[15:0], RF_oRegB[15:0]};
assign oPORT = RB_out;

// Write Back
// Select Memory input on WBM, Select PC for JAL, otherwise use ALU result
assign RWB_in = iMUX_WBM ? iMemData :
                iMUX_WBE ? iPORT    :
                iMUX_WBP ? PC_tOut  : RZX_out;

// Write back buffer register
REG32 RWB(.iClk(iClk), .nRst(nRst), .iEn(iRWB_en), .iD(RWB_in), .oQ(RF_iRegC));

endmodule
```

Processor module

```
module Processor(
    iClk, nRst,
    oMemAddr, oMemData,
    iMemData, iMemRdy,
    oMemRead, oMemWrite,
    iPORT, oPORT
);

`include "constants.vh"

input wire iClk, nRst, iMemRdy;
output wire oMemRead, oMemWrite;
input wire [31:0] iMemData;
output wire [31:0] oMemData, oMemAddr;
input wire [31:0] iPORT;
output wire [31:0] oPORT;

// Program Counter Signals
wire PC_nRst, PC_en, PC_tmpEn, PC_load, PC_offset;

// Register File IO
wire RF_iWrite;
wire [3:0] RF_iAddrA, RF_iAddrB, RF_iAddrC;
wire RWB_en;

// ALU IO
wire [3:0]  ALU_iCtrl;
wire ALU_oZero, ALU_oNeg;

// ALU Immediate Registers
wire RA_en, RB_en;
wire RZH_en, RZL_en;
// ALU Storage Registers
wire RAS_en;

// Jump/Branch Signals
wire J_zero, J_nZero, J_pos, J_neg;

// External Port Signals
wire REP_en;
wire [31:0] REP_in;

// Multiplexer Signals
wire MUX_BIS, MUX_RZHS, MUX_WBM, MUX_MAP, MUX_ASS, MUX_WBP, MUX_WBE;
```

```
// Control Signals
wire [31:0] CT_imm32;

// Control Unit
Control Ctrl(
    // Clock, reset and ready signals
    // Ready is an active high that allows the next step to continue
    .iClk(iClk),
    .nRst(nRst),
    .iRdy(iMemRdy),
    // Memory Signals/Control
    .iMemData(iMemData),
    .oMemRead(oMemRead),
    .oMemWrite(oMemWrite),
    // Pipe Control
    .oPipe_nRst(pipe_rst),
    // Program Counter Control
    .oPC_nRst(PC_nRst),
    .oPC_en(PC_en),
    .oPC_tmpEn(PC_tmpEn),
    .oPC_load(PC_load),
    .oPC_offset(PC_offset),
    // Register File Control
    .oRF_Write(RF_iWrite),
    .oRF_AddrA(RF_iAddrA),
    .oRF_AddrB(RF_iAddrB),
    .oRF_AddrC(RF_iAddrC),
    .oRWB_en(RWB_en),
    // ALU Control
    .oALU_Ctrl(ALU_iCtrl),
    .oRA_en(RA_en),
    .oRB_en(RB_en),
    .oRZH_en(RZH_en),
    .oRZL_en(RZL_en),
    .oRAS_en(RAS_en),
    // Jump Feedback
    .iJ_zero(J_zero),
    .iJ_nZero(J_nZero),
    .iJ_pos(J_pos),
    .iJ_neg(J_neg),
    // External Port Register Enable
    .oREP_en(REP_en),
    // Multiplexers
    .oMUX_BIS(MUX_BIS),
    .oMUX_RZHS(MUX_RZHS),
```

```
    .oMUX_WBM(MUX_WBM),
    .oMUX_MAP(MUX_MAP),
    .oMUX_ASS(MUX_ASS),
    .oMUX_WBP(MUX_WBP),
    .oMUX_WBE(MUX_WBE),
    // Imm32 Output
    .oImm32(CT_imm32)
);

Datapath pipe(
    // Clock and reset signals (reset is active low)
    .iClk(iClk),
    .nRst(pipe_rst),
    // Memory Signals
    .iMemData(iMemData),
    .oMemAddr(oMemAddr),
    .oMemData(oMemData),
    // Port Signals
    .iPORT(iPORT),
    .oPORT(REP_in),
    // Program Counter Control
    .iPC_nRst(PC_nRst),
    .iPC_en(PC_en),
    .iPC_tmpEn(PC_tmpEn),
    .iPC_load(PC_load),
    .iPC_offset(PC_offset),
    // Register File Control
    .iRF_Write(RF_iWrite),
    .iRF_AddrA(RF_iAddrA),
    .iRF_AddrB(RF_iAddrB),
    .iRF_AddrC(RF_iAddrC),
    // Write Back Register Control
    .iRWB_en(RWB_en),
    // ALU Control
    .iALU_Ctrl(ALU_iCtrl),
    .iRA_en(RA_en),
    .iRB_en(RB_en),
    .iRZH_en(RZH_en),
    .iRZL_en(RZL_en),
    .iRAS_en(RAS_en),
    // Jump Feedback
    .oJ_zero(J_zero),
    .oJ_nZero(J_nZero),
    .oJ_pos(J_pos),
    .oJ_neg(J_neg),
    // ALU Results
    .oALU_neg(ALU_oNeg),
    .oALU_zero(ALU_oZero),
    // Multiplexers
    .iMUX_BIS(MUX_BIS), // ALU B Input/Immediate Select
    .iMUX_RZHS(MUX_RZHS), // ALU Result High Select
    .iMUX_WBM(MUX_WBM), // Write back in Memory Select
    .iMUX_MAP(MUX_MAP), // Memory Address out PC Select
    .iMUX_ASS(MUX_ASS), // ALU Storage Select
    .iMUX_WBP(MUX_WBP),
    .iMUX_WBE(MUX_WBE),
    // Imm32 Output
    .iImm32(CT_imm32)
);

REG32 REP(.iClk(iClk), .nRst(nRst), .iEn(REP_en), .iD(REP_in), .oQ(oPORT));
// assign oPORT = REP_in;

endmodule
```

Register module

```
module REG32 #(parameter RESET = 32'd0)(
    iClk,
    nRst, iEn,
    iD, oQ
);

input wire iClk, nRst, iEn;
input wire [31:0] iD;
output reg [31:0] oQ;

always@(posedge iClk or negedge nRst)
begin
    if(!nRst)
        oQ <= RESET;
    else begin
        if(iEn) oQ <= iD;
    end
end

endmodule
```

Register file module

```
module RegFile(
    iClk, nRst, iWrite,
    iAddrA, iAddrB, iAddrC,
    oRegA, oRegB, iRegC
);

// Module IO
input wire iClk, nRst, iWrite;
input wire [3:0] iAddrA, iAddrB, iAddrC;
output wire [31:0] oRegA, oRegB;
input wire [31:0] iRegC;


// Register IO
wire r1_write, r2_write, r3_write, r5_write, r6_write,
     r7_write, r8_write, r9_write, r10_write, r11_write,
     r12_write, r13_write, r14_write, r15_write;

wire [31:0] r1_out, r2_out, r3_out, r4_out, r5_out,
            r6_out, r7_out, r8_out, r9_out, r10_out,
            r11_out, r12_out, r13_out, r14_out, r15_out;

// Write Signal Assert
assign r1_write  = (iAddrC == 4'b0001) && iWrite;
assign r2_write  = (iAddrC == 4'b0010) && iWrite;
assign r3_write  = (iAddrC == 4'b0011) && iWrite;
assign r4_write  = (iAddrC == 4'b0100) && iWrite;
assign r5_write  = (iAddrC == 4'b0101) && iWrite;
assign r6_write  = (iAddrC == 4'b0110) && iWrite;
assign r7_write  = (iAddrC == 4'b0111) && iWrite;
assign r8_write  = (iAddrC == 4'b1000) && iWrite;
assign r9_write  = (iAddrC == 4'b1001) && iWrite;
assign r10_write = (iAddrC == 4'b1010) && iWrite;
assign r11_write = (iAddrC == 4'b1011) && iWrite;
assign r12_write = (iAddrC == 4'b1100) && iWrite;
assign r13_write = (iAddrC == 4'b1101) && iWrite;
assign r14_write = (iAddrC == 4'b1110) && iWrite;
assign r15_write = (iAddrC == 4'b1111) && iWrite;


// Output Register A assignment
assign oRegA =  (iAddrA == 4'b0001) ? r1_out  :
                (iAddrA == 4'b0010) ? r2_out  :
                (iAddrA == 4'b0011) ? r3_out  :
                (iAddrA == 4'b0100) ? r4_out  :
                (iAddrA == 4'b0101) ? r5_out  :
                (iAddrA == 4'b0110) ? r6_out  :
                (iAddrA == 4'b0111) ? r7_out  :
                (iAddrA == 4'b1000) ? r8_out  :
                (iAddrA == 4'b1001) ? r9_out  :
                (iAddrA == 4'b1010) ? r10_out :
                (iAddrA == 4'b1011) ? r11_out :
                (iAddrA == 4'b1100) ? r12_out :
                (iAddrA == 4'b1101) ? r13_out :
                (iAddrA == 4'b1110) ? r14_out :
                (iAddrA == 4'b1111) ? r15_out :
                32'd0;

// Output Register A assignment
assign oRegB =  (iAddrB == 4'b0001) ? r1_out  :
                (iAddrB == 4'b0010) ? r2_out  :
                (iAddrB == 4'b0011) ? r3_out  :
                (iAddrB == 4'b0100) ? r4_out  :
                (iAddrB == 4'b0101) ? r5_out  :
                (iAddrB == 4'b0110) ? r6_out  :
                (iAddrB == 4'b0111) ? r7_out  :
                (iAddrB == 4'b1000) ? r8_out  :
                (iAddrB == 4'b1001) ? r9_out  :
                (iAddrB == 4'b1010) ? r10_out :
                (iAddrB == 4'b1011) ? r11_out :
                (iAddrB == 4'b1100) ? r12_out :
                (iAddrB == 4'b1101) ? r13_out :
                (iAddrB == 4'b1110) ? r14_out :
                (iAddrB == 4'b1111) ? r15_out :
                32'd0;

// Registers
REG32 r1(  .iClk(iClk), .nRst(nRst), .iEn(r1_write),  .iD(iRegC), .oQ(r1_out)  );
REG32 r2(  .iClk(iClk), .nRst(nRst), .iEn(r2_write),  .iD(iRegC), .oQ(r2_out)  );
REG32 r3(  .iClk(iClk), .nRst(nRst), .iEn(r3_write),  .iD(iRegC), .oQ(r3_out)  );
REG32 r4(  .iClk(iClk), .nRst(nRst), .iEn(r4_write),  .iD(iRegC), .oQ(r4_out)  );
REG32 r5(  .iClk(iClk), .nRst(nRst), .iEn(r5_write),  .iD(iRegC), .oQ(r5_out)  );
REG32 r6(  .iClk(iClk), .nRst(nRst), .iEn(r6_write),  .iD(iRegC), .oQ(r6_out)  );
REG32 r7(  .iClk(iClk), .nRst(nRst), .iEn(r7_write),  .iD(iRegC), .oQ(r7_out)  );
REG32 r8(  .iClk(iClk), .nRst(nRst), .iEn(r8_write),  .iD(iRegC), .oQ(r8_out)  );
REG32 r9(  .iClk(iClk), .nRst(nRst), .iEn(r9_write),  .iD(iRegC), .oQ(r9_out)  );
REG32 r10( .iClk(iClk), .nRst(nRst), .iEn(r10_write), .iD(iRegC), .oQ(r10_out) );
REG32 r11( .iClk(iClk), .nRst(nRst), .iEn(r11_write), .iD(iRegC), .oQ(r11_out) );
REG32 r12( .iClk(iClk), .nRst(nRst), .iEn(r12_write), .iD(iRegC), .oQ(r12_out) );
REG32 r13( .iClk(iClk), .nRst(nRst), .iEn(r13_write), .iD(iRegC), .oQ(r13_out) );
REG32 r14( .iClk(iClk), .nRst(nRst), .iEn(r14_write), .iD(iRegC), .oQ(r14_out) );
REG32 r15( .iClk(iClk), .nRst(nRst), .iEn(r15_write), .iD(iRegC), .oQ(r15_out) );


endmodule
```

Seven Segment Code

Sev_segs module

```
module SevSegs(
    iNum,
    oSeg1,
    oSeg2,
    oSeg3,
    oSeg4,
    oSeg5,
    oSeg6,
    oSeg7,
    oSeg8
);
input wire [31:0] iNum;
output wire [6:0] oSeg1, oSeg2, oSeg3, oSeg4, oSeg5, oSeg6, oSeg7, oSeg8;
SevenSeg s1(
    .iNum(iNum[3:0]),
    .oSeg(oSeg1)
);
SevenSeg s2(
    .iNum(iNum[7:4]),
    .oSeg(oSeg2)
);
SevenSeg s3(
    .iNum(iNum[11:8]),
    .oSeg(oSeg3)
);
SevenSeg s4(
    .iNum(iNum[15:12]),
    .oSeg(oSeg4)
);
SevenSeg s5(
    .iNum(iNum[19:16]),
    .oSeg(oSeg5)
);
SevenSeg s6(
    .iNum(iNum[23:20]),
    .oSeg(oSeg6)
);
SevenSeg s7(
    .iNum(iNum[27:24]),
    .oSeg(oSeg7)
);
SevenSeg s8(
    .iNum(iNum[31:28]),
    .oSeg(oSeg8)
);
```

Seven_seg module

```
module SevenSeg(
    iNum,
    oSeg
);
input wire [3:0] iNum;
output reg [6:0] oSeg;

always @(iNum) begin
    case(iNum)
        4'h0: oSeg = 7'b1000000;
        4'h1: oSeg = 7'b1111001;
        4'h2: oSeg = 7'b0100100;
        4'h3: oSeg = 7'b0110000;
        4'h4: oSeg = 7'b0011001;
        4'h5: oSeg = 7'b0010010;
        4'h6: oSeg = 7'b0000010;
        4'h7: oSeg = 7'b1111000;
        4'h8: oSeg = 7'b0000000;
        4'h9: oSeg = 7'b0010000;
        4'hA: oSeg = 7'b0001000;
        4'hB: oSeg = 7'b0000011;
        4'hC: oSeg = 7'b1000110;
        4'hD: oSeg = 7'b0100001;
        4'hE: oSeg = 7'b0000110;
        4'hF: oSeg = 7'b0001110;
    endcase
end

endmodule
```

VGA Code

VGA controller code

```
// Simple VGA Controller
// 640x480 Resolution

module vga_controller(
    iClk_50,
    nRst,
    iVGA_colorData,
    oVGA_colorAddress,
    oVGA_R,
    oVGA_G,
    oVGA_B,
    oVGA_Clk,
    oVGA_Blank,
    oVGA_HSync,
    oVGA_VSync,
    oVGA_Sync,
);

input wire iClk_50, nRst;
input wire [29:0]  iVGA_colorData;
output wire [31:0] oVGA_colorAddress;
output wire [9:0] oVGA_R, oVGA_G, oVGA_B;
output wire oVGA_Clk, oVGA_Blank, oVGA_HSync, oVGA_VSync, oVGA_Sync;

parameter WIDTH = 16'd639;
parameter HEIGHT = 16'd479;

// Signal States
// Active Video
parameter SS_AV = 3'd0;
// Front Porch
parameter SS_FP = 3'd1;
// Sync Pulse
parameter SS_SP = 3'd2;
// Back Porch
parameter SS_BP = 3'd3;
// Blank
parameter SS_BL = 3'd4;

// Video States
// Active Video
parameter VS_AV = 2'd0;
// Horizontal Sync
parameter VS_HS = 2'd1;
// Vertical Sync
```

```
C:\Users\lukea\Documents\github\MiniSRC\VGA

// Video State
reg [1:0] videoState, videoState_next;
// Horizontal and Vertical Signal States
reg [2:0] hState, vState;
reg [2:0] hState_next, vState_next;

// Signal Clock Counters
reg [15:0] s_width, s_height;

// In/Out Color Data
assign oVGA_colorAddress = {s_height, s_width};
wire [9:0] cR, cG, cB;
assign cR = iVGA_colorData[9:0];
assign cG = iVGA_colorData[19:10];
assign cB = iVGA_colorData[29:20];

// Clock Divider
reg Clk_25;
always @(posedge iClk_50, negedge nRst) begin
    if(~nRst) Clk_25 = 1'b0;
    else Clk_25 = ~Clk_25;
end

always @(posedge Clk_25, negedge nRst) begin
    if(~nRst) begin
        videoState = VS_AV;
        hState = SS_AV;
        vState = SS_AV;
    end
    else begin
        videoState = videoState_next;
        vState = vState_next;
        hState = hState_next;
        case(videoState)
            // Active Video
            VS_AV: begin
                oVGA_R = cR;
                oVGA_G = cG;
                oVGA_B = cB;
                if (s_width == WIDTH) begin
                    s_width = 16'd0;
                    if(s_height == HEIGHT) begin
                        s_height = 16'd0;
```

```
                    videoState_next = VS_VS;
                    vState_next = SS_FP;
                    hState_next = SS_BL;
                end else begin
                    s_height = s_height + 16'd1;
                    videoState_next = VS_HS;
                    vState_next = SS_AV;
                    hState_next = SS_FP;
                end
            end else begin
                s_width = s_width + 16'd1;
                videoState_next = VS_AV;
                vState_next = SS_AV;
                hState_next = SS_AV;
            end
        end
        // Vertical Sync
        VS_VS: begin
            oVGA_R = 10'd0;
            oVGA_G = 10'd0;
            oVGA_B = 10'd0;
            case(vState)
                SS_FP:;
                SS_SP:;
                SS_BP:;
                default: begin
                    videoState_next = VS_AV;
                    vState_next = SS_AV;
                    hState_next = SS_AV;
                end
            endcase
        end
        // Horizontal Sync
        VS_HS: begin
        end
        default: begin
            videoState_next = VS_AV;
            vState_next = SS_AV;
            hState_next = SS_AV;
        end
```

```
        endcase
    end
end

endmodule

```

Memory code

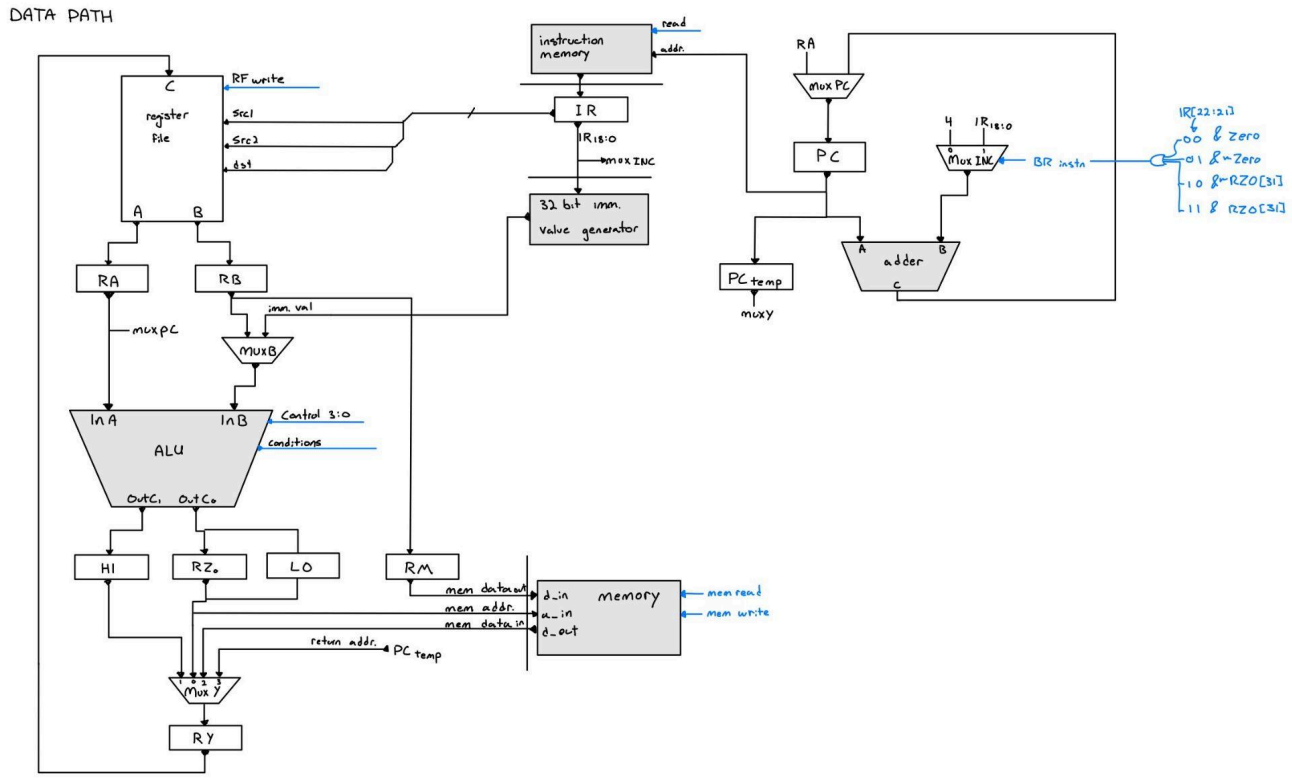
Memory module

```
`include "../Processor/Control/ISA.vh"
`include "../Processor/tb/sim_ISA.vh"
module memory(
    iClk, iRead, iWrite,
    iData, iAddr,
    oData
);

input wire iClk, iRead, iWrite;
input wire [31:0] iData, iAddr;
output reg [31:0] oData;

reg [31:0] mem[0:1023];

wire [31:2] Addri;
assign Addri = iAddr[31:2];

// assign oData = iRead ? mem[Addri] : 32'd0;

always @(posedge iClk) begin
    if(iWrite)
        mem[Addri] <= iData;
    oData <= mem[Addri];
end

initial $readmemh("program.hex", mem);

// initial begin
//     // Set the output port to the value in R1
//     mem[0] = `INS_J(`ISA_OUT, 4'h1);
//     // Increase R1 by 1
//     mem[1] = `INS_I(`ISA_ADDI, 4'h1, 4'h1, 19'd1);
//     // Load the counter value from the in port into R2
//     mem[2] = `INS_J(`ISA_IN, 4'h2);
//     // Loop for the duration of R2
//     mem[3] = `INS_I(`ISA_ADDI, 4'h3, 4'h3, 19'd1);
//     mem[4] = `INS_R(`ISA_SUB, 4'h4, 4'h2, 4'h3);
//     mem[5] = `INS_B(`ISA_BRx, 4'h4, `ISA_BR_POSI, -19'd3);
//     mem[6] = `INS_I(`ISA_ADDI, 4'h3, 4'h0, 19'd0);
//     // Go back to the beginning of the code
//     mem[7] = `INS_J(`ISA_JFR, 4'h0);
// end

endmodule
```

MMU module

```
module MMU(
    iClk, iRead, iWrite,
    iData, iAddr,
    oData
);

input wire iClk, iRead, iWrite;
input wire [31:0] iData, iAddr;
output wire [31:0] oData;


memory mem(
    .iClk(iClk),
    .iRead(iRead),
    .iWrite(iWrite),
    .iData(iData),
    .iAddr(iAddr),
    .oData(oData)
);

endmodule
```

Appendix 2: Processor Schematic



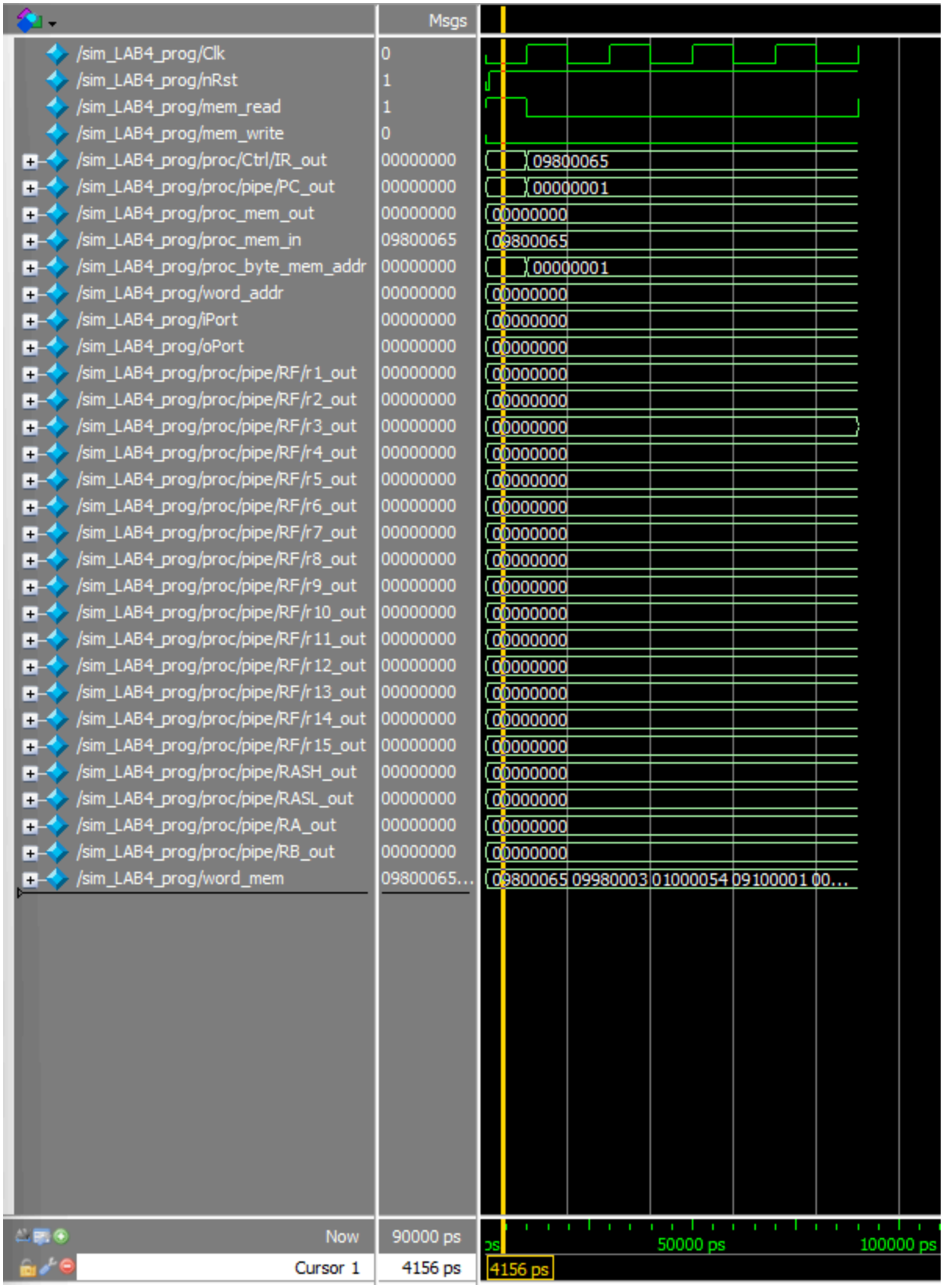
Appendix 3: Functional Simulation results for Phase 4

Simulation test bench code:

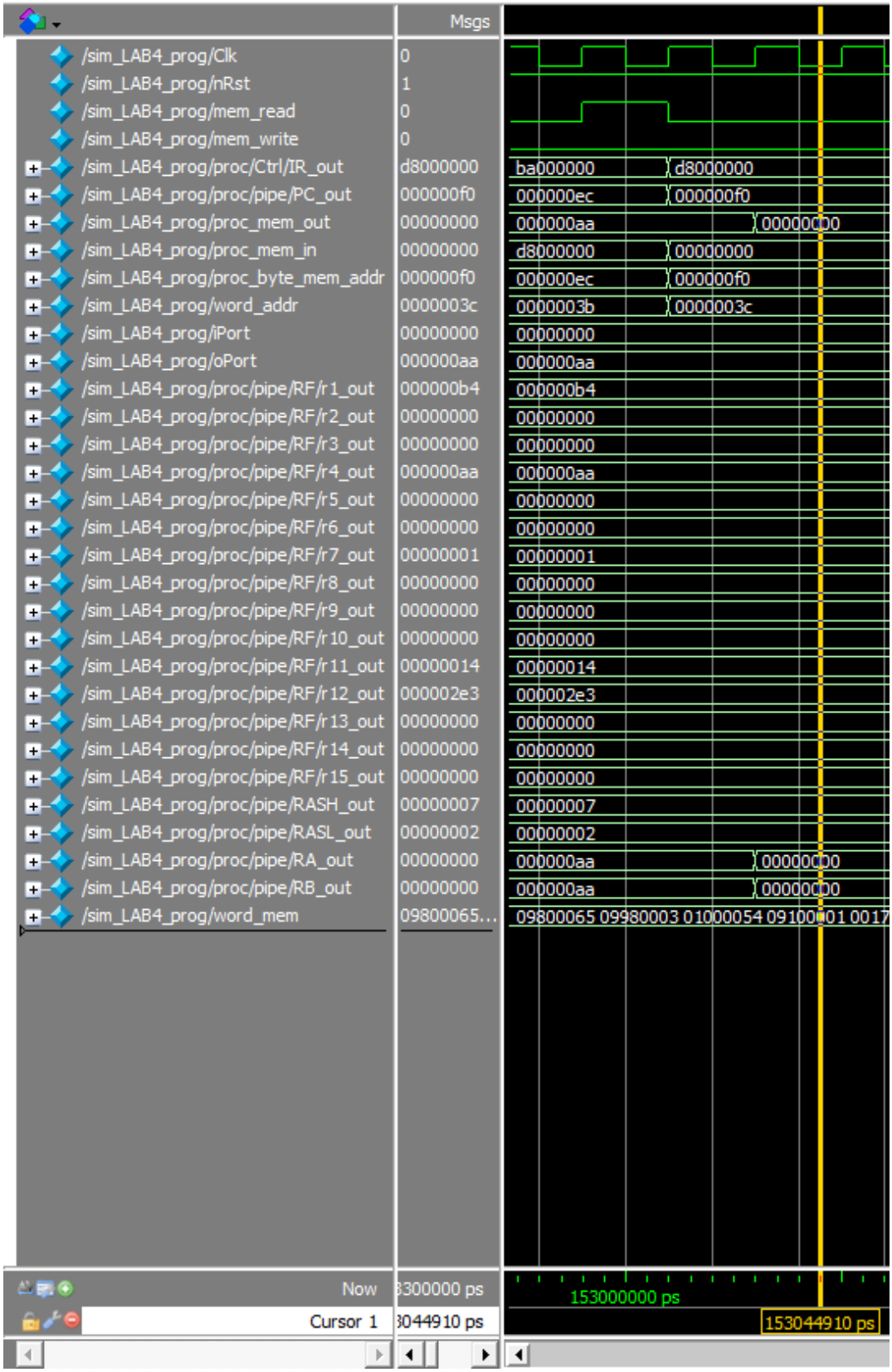
```
1   `timescale 1ns/1ps
2   `include "../../Control/ISA.vh"
3   `include "../../constants.vh"
4   `include "../sim_ISA.vh"
5
6   // Complete simulation in 153100ns with iPort = 0x00000000.
7
8   module sim_LAB4_prog();
9
10  parameter SA = `START_PC_ADDRESS;
11
12  wire Clk;
13  reg nRst = 1'b0;
14
15  ClockGenerator cg(
16      .nRst(nRst),
17      .oClk(Clk)
18  );
19
20  wire mem_read, mem_write;
21  wire [31:0] proc_mem_out, proc_byte_mem_addr;
22  reg [31:0] proc_mem_in;
23  wire [31:0] oPort;
24  wire [31:0] iPort = 32'h00000000;
25
26  wire [31:2] word_addr;
27  assign word_addr = proc_byte_mem_addr[31:2];
28
29  `define MEM_MAX (511)
30  reg [31:0] word_mem[0:`MEM_MAX];
```

```
integer i;
initial begin
    for (i = 0; i <= `MEM_MAX; i = i + 1) begin
        word_mem[i] = 32'h00000000; // Initialize memory to zero
    end
    // initialize memory with program
    $readmemh("Processor/tb/lab4demo/program.hex", word_mem);
    #1;
    nRst = 1'b1; // Release reset after 1ns
end

always @(proc_mem_out, proc_mem_in, word_addr) begin
    if(mem_write)
        word_mem[word_addr] <= proc_mem_out;
    proc_mem_in <= word_mem[word_addr];
end

Processor proc(
    .iClk(Clk),
    .nRst(nRst),
    .oMemAddr(proc_byte_mem_addr),
    .oMemData(proc_mem_out),
    .iMemData(proc_mem_in),
    .iMemRdy(1'b1),
    .oMemRead(mem_read),
    .oMemWrite(mem_write)
    ,.iPORT(iPort)
    ,.oPORT(oPort)
);

endmodule
```

Processor State before execution:



Processor State after execution:



For memory state before and after execution, see the following Appendix 4.

Appendix 4: Printout of the contents of memory for Phase 4

Before Execution:

```
 @0 00001001100000000000000001100101
 @1 00001001100110000000000000000011
 @2 00000001000000000000000001010100
 @3 00001001000100000000000000000001
 @4 00000000000101111111111111111010
 @5 00001000100000000000000000000011
 @6 00001001100000000000000001010111
 @7 10011001100110000000000000000011
 @8 00001001100110000000000000000011
 @9 00000010000111111111111111111010
 @a 11010000000000000000000000000000
 @b 10011010000100000000000000000010
 @c 00001011000110000000000000000111
 @d 00001010101101111111111111111100
 @e 00011001100110001000000000000000
 @f 01100010001000000000000000000010
@10 10001010001000000000000000000000
@11 10010010001000000000000000000000
@12 01101010001000000000000000001111
@13 00111001000000001000000000000000
@14 01110010000100000000000000000111
@15 01010001001000001000000000000000
@16 01001001100110001000000000000000
@17 00010001100000000000000010010010
@18 01000001100000001000000000000000
@19 00110010100010000000000000000000
@1a 00101001000110000000000000000000
@1b 00010010100100000000000001010100
@1c 00100000000110101000000000000000
@1d 01011001000110001000000000000000
@1e 00001010100000000000000000001000
@1f 00001011000000000000000000010111
@20 10000011001010000000000000000000
@21 11001010000000000000000000000000
@22 11000011100000000000000000000000
@23 01111011001010000000000000000000
```

```
@24 00001101001010000000000000000001
@25 00001101101101111111111111111101
@26 00001110001110000000001000101011
@27 00001110101000000000000000000100
@28 10110010000000000000000000000000
@29 00010010000000000000000101010100
@2a 00001000100000000000000010110100
@2b 00001011100000000000000000000001
@2c 00001010100000000000000000101000
@2d 10111010000000000000000000000000
@2e 00001010101011111111111111111111
@2f 10011010100000000000000000001001
@30 01100011000000000000000000001010
@31 11010000000000000000000000000000
@32 00001011001101111111111111111111
@33 10011011000101111111111111111101
@34 01001010001000111000000000000000
@35 10011010000101111111111111110111
@36 00000010000000000000000101010100
@37 10101000100000000000000000000000
@38 11010000000000000000000000000000
@39 00001010000000000000000010101010
@3a 10111010000000000000000000000000
@3b 11011000000000000000000000000000
@3c 00000000000000000000000000000000
@3d 00000000000000000000000000000000
@3e 00000000000000000000000000000000
@3f 00000000000000000000000000000000
@40 00000000000000000000000000000000
@41 00000000000000000000000000000000
@42 00000000000000000000000000000000
@43 00000000000000000000000000000000
@44 00000000000000000000000000000000
@45 00000000000000000000000000000000
@46 00000000000000000000000000000000
@47 00000000000000000000000000000000
@48 00000000000000000000000000000000
@49 00000000000000000000000000000000
```

```
@4a 00000000000000000000000000000000
@4b 00000000000000000000000000000000
@4c 00000000000000000000000000000000
@4d 00000000000000000000000000000000
@4e 00000000000000000000000000000000
@4f 00000000000000000000000000000000
@50 00000000000000000000000000000000
@51 00000000000000000000000000000000
@52 00000000000000000000000000000000
@53 00000000000000000000000000000000
@54 00000000000000000000000010010111
@55 00000000000000000000000000000000
@56 00000000000000000000000000000000
@57 00000000000000000000000000000000
@58 00000000000000000000000000000000
@59 00000000000000000000000000000000
@5a 00000000000000000000000000000000
@5b 00000000000000000000000000000000
@5c 00000000000000000000000000000000
@5d 00000000000000000000000000000000
@5e 00000000000000000000000000000000
@5f 00000000000000000000000000000000
@60 00000000000000000000000000000000
@61 00000000000000000000000000000000
@62 00000000000000000000000000000000
@63 00000000000000000000000000000000
@64 00000000000000000000000000000000
@65 00000000000000000000000000000000
@66 00000000000000000000000000000000
@67 00000000000000000000000000000000
@68 00000000000000000000000000000000
@69 00000000000000000000000000000000
@6a 00000000000000000000000000000000
@6b 00000000000000000000000000000000
@6c 00000000000000000000000000000000
@6d 00000000000000000000000000000000
@6e 00000000000000000000000000000000
@6f 00000000000000000000000000000000
```

```
@70 00000000000000000000000000000000
@71 00000000000000000000000000000000
@72 00000000000000000000000000000000
@73 00000000000000000000000000000000
@74 00000000000000000000000000000000
@75 00000000000000000000000000000000
@76 00000000000000000000000000000000
@77 00000000000000000000000000000000
@78 00000000000000000000000000000000
@79 00000000000000000000000000000000
@7a 00000000000000000000000000000000
@7b 00000000000000000000000000000000
@7c 00000000000000000000000000000000
@7d 00000000000000000000000000000000
@7e 00000000000000000000000000000000
@7f 00000000000000000000000000000000
@80 00000000000000000000000000000000
@81 00000000000000000000000000000000
@82 00000000000000000000000000000000
@83 00000000000000000000000000000000
@84 00000000000000000000000000000000
@85 00000000000000000000000000000000
@86 00000000000000000000000000000000
@87 00000000000000000000000000000000
@88 00000000000000000000000000000000
@89 00000000000000000000000000000000
@8a 00000000000000000000000000000000
@8b 00000000000000000000000000000000
@8c 00000000000000000000000000000000
@8d 00000000000000000000000000000000
@8e 00000000000000000000000000000000
@8f 00000000000000000000000000000000
@90 00000000000000000000000000000000
@91 00000000000000000000000000000000
@92 00000000000000000000000001000110
@93 00000000000000000000000000000000
@94 00000000000000000000000000000000
@95 00000000000000000000000000000000
```

```
@96 00000000000000000000000000000000
@97 00000000000000000000000000000000
@98 00000000000000000000000000000000
@99 00000000000000000000000000000000
@9a 00000000000000000000000000000000
@9b 00000000000000000000000000000000
@9c 00000000000000000000000000000000
@9d 00000000000000000000000000000000
@9e 00000000000000000000000000000000
@9f 00000000000000000000000000000000
@a0 00000000000000000000000000000000
@a1 00000000000000000000000000000000
@a2 00000000000000000000000000000000
@a3 00000000000000000000000000000000
@a4 00000000000000000000000000000000
@a5 00000000000000000000000000000000
@a6 00000000000000000000000000000000
@a7 00000000000000000000000000000000
@a8 00000000000000000000000000000000
@a9 00000000000000000000000000000000
@aa 00000000000000000000000000000000
@ab 00000000000000000000000000000000
@ac 00000000000000000000000000000000
@ad 00000000000000000000000000000000
@ae 00000000000000000000000000000000
@af 00000000000000000000000000000000
@b0 00000000000000000000000000000000
@b1 00000000000000000000000000000000
@b2 00000000000000000000000000000000
@b3 00000000000000000000000000000000
@b4 00000000000000000000000000000000
@b5 00000000000000000000000000000000
@b6 00000000000000000000000000000000
@b7 00000000000000000000000000000000
@b8 00000000000000000000000000000000
@b9 00011100000011110000000000000000
@ba 00011111110101100000000000000000
@bb 00100111010111101000000000000000
```

```
@bc 00100111111111110000000000000000
@bd 10101100000000000000000000000000
@be 00000000000000000000000000000000
@bf 00000000000000000000000000000000
@c0 00000000000000000000000000000000
@c1 00000000000000000000000000000000
@c2 00000000000000000000000000000000
@c3 00000000000000000000000000000000
@c4 00000000000000000000000000000000
@c5 00000000000000000000000000000000
@c6 00000000000000000000000000000000
@c7 00000000000000000000000000000000
@c8 00000000000000000000000000000000
@c9 00000000000000000000000000000000
@ca 00000000000000000000000000000000
@cb 00000000000000000000000000000000
@cc 00000000000000000000000000000000
@cd 00000000000000000000000000000000
@ce 00000000000000000000000000000000
@cf 00000000000000000000000000000000
@d0 00000000000000000000000000000000
@d1 00000000000000000000000000000000
@d2 00000000000000000000000000000000
@d3 00000000000000000000000000000000
@d4 00000000000000000000000000000000
@d5 00000000000000000000000000000000
@d6 00000000000000000000000000000000
@d7 00000000000000000000000000000000
@d8 00000000000000000000000000000000
@d9 00000000000000000000000000000000
@da 00000000000000000000000000000000
@db 00000000000000000000000000000000
@dc 00000000000000000000000000000000
@dd 00000000000000000000000000000000
@de 00000000000000000000000000000000
@df 00000000000000000000000000000000
@e0 00000000000000000000000000000000
@e1 00000000000000000000000000000000
```

```
@e2 00000000000000000000000000000000
@e3 00000000000000000000000000000000
@e4 00000000000000000000000000000000
@e5 00000000000000000000000000000000
@e6 00000000000000000000000000000000
@e7 00000000000000000000000000000000
@e8 00000000000000000000000000000000
@e9 00000000000000000000000000000000
@ea 00000000000000000000000000000000
@eb 00000000000000000000000000000000
@ec 00000000000000000000000000000000
@ed 00000000000000000000000000000000
@ee 00000000000000000000000000000000
@ef 00000000000000000000000000000000
@f0 00000000000000001111111111111111
@f1 00000000000000000000000000000000
@f2 00000000000000000000000000000000
@f3 00000000000000000000000000000000
@f4 00000000000000000000000000000000
@f5 00000000000000000000000000000000
@f6 00000000000000000000000000000000
@f7 00000000000000000000000000000000
@f8 00000000000000000000000000000000
@f9 00000000000000000000000000000000
@fa 00000000000000000000000000000000
@fb 00000000000000000000000000000000
@fc 00000000000000000000000000000000
@fd 00000000000000000000000000000000
@fe 00000000000000000000000000000000
@ff 00000000000000000000000000000000
@100 00000000000000000000000000000000
@101 00000000000000000000000000000000
@102 00000000000000000000000000000000
@103 00000000000000000000000000000000
@104 00000000000000000000000000000000
@105 00000000000000000000000000000000
@106 00000000000000000000000000000000
@107 00000000000000000000000000000000
```

```
@108 00000000000000000000000000000000
@109 00000000000000000000000000000000
@10a 00000000000000000000000000000000
@10b 00000000000000000000000000000000
@10c 00000000000000000000000000000000
@10d 00000000000000000000000000000000
@10e 00000000000000000000000000000000
@10f 00000000000000000000000000000000
@110 00000000000000000000000000000000
@111 00000000000000000000000000000000
@112 00000000000000000000000000000000
@113 00000000000000000000000000000000
@114 00000000000000000000000000000000
@115 00000000000000000000000000000000
@116 00000000000000000000000000000000
@117 00000000000000000000000000000000
@118 00000000000000000000000000000000
@119 00000000000000000000000000000000
@11a 00000000000000000000000000000000
@11b 00000000000000000000000000000000
@11c 00000000000000000000000000000000
@11d 00000000000000000000000000000000
@11e 00000000000000000000000000000000
@11f 00000000000000000000000000000000
@120 00000000000000000000000000000000
@121 00000000000000000000000000000000
@122 00000000000000000000000000000000
@123 00000000000000000000000000000000
@124 00000000000000000000000000000000
@125 00000000000000000000000000000000
@126 00000000000000000000000000000000
@127 00000000000000000000000000000000
@128 00000000000000000000000000000000
@129 00000000000000000000000000000000
@12a 00000000000000000000000000000000
@12b 00000000000000000000000000000000
@12c 00000000000000000000000000000000
@12d 00000000000000000000000000000000
```

```
@12e 00000000000000000000000000000000
@12f 00000000000000000000000000000000
@130 00000000000000000000000000000000
@131 00000000000000000000000000000000
@132 00000000000000000000000000000000
@133 00000000000000000000000000000000
@134 00000000000000000000000000000000
@135 00000000000000000000000000000000
@136 00000000000000000000000000000000
@137 00000000000000000000000000000000
@138 00000000000000000000000000000000
@139 00000000000000000000000000000000
@13a 00000000000000000000000000000000
@13b 00000000000000000000000000000000
@13c 00000000000000000000000000000000
@13d 00000000000000000000000000000000
@13e 00000000000000000000000000000000
@13f 00000000000000000000000000000000
@140 00000000000000000000000000000000
@141 00000000000000000000000000000000
@142 00000000000000000000000000000000
@143 00000000000000000000000000000000
@144 00000000000000000000000000000000
@145 00000000000000000000000000000000
@146 00000000000000000000000000000000
@147 00000000000000000000000000000000
@148 00000000000000000000000000000000
@149 00000000000000000000000000000000
@14a 00000000000000000000000000000000
@14b 00000000000000000000000000000000
@14c 00000000000000000000000000000000
@14d 00000000000000000000000000000000
@14e 00000000000000000000000000000000
@14f 00000000000000000000000000000000
@150 00000000000000000000000000000000
@151 00000000000000000000000000000000
@152 00000000000000000000000000000000
@153 00000000000000000000000000000000
@154 00000000000000000000000000000000
@155 00000000000000000000000000000000
@156 00000000000000000000000000000000
@157 00000000000000000000000000000000
@158 00000000000000000000000000000000
@159 00000000000000000000000000000000
@15a 00000000000000000000000000000000
@15b 00000000000000000000000000000000
@15c 00000000000000000000000000000000
@15d 00000000000000000000000000000000
@15e 00000000000000000000000000000000
@15f 00000000000000000000000000000000
@160 00000000000000000000000000000000
@161 00000000000000000000000000000000
@162 00000000000000000000000000000000
@163 00000000000000000000000000000000
@164 00000000000000000000000000000000
@165 00000000000000000000000000000000
@166 00000000000000000000000000000000
@167 00000000000000000000000000000000
@168 00000000000000000000000000000000
@169 00000000000000000000000000000000
@16a 00000000000000000000000000000000
@16b 00000000000000000000000000000000
@16c 00000000000000000000000000000000
@16d 00000000000000000000000000000000
@16e 00000000000000000000000000000000
@16f 00000000000000000000000000000000
@170 00000000000000000000000000000000
@171 00000000000000000000000000000000
@172 00000000000000000000000000000000
@173 00000000000000000000000000000000
@174 00000000000000000000000000000000
@175 00000000000000000000000000000000
@176 00000000000000000000000000000000
@177 00000000000000000000000000000000
@178 00000000000000000000000000000000
@179 00000000000000000000000000000000
```

```
@17a 00000000000000000000000000000000
@17b 00000000000000000000000000000000
@17c 00000000000000000000000000000000
@17d 00000000000000000000000000000000
@17e 00000000000000000000000000000000
@17f 00000000000000000000000000000000
@180 00000000000000000000000000000000
@181 00000000000000000000000000000000
@182 00000000000000000000000000000000
@183 00000000000000000000000000000000
@184 00000000000000000000000000000000
@185 00000000000000000000000000000000
@186 00000000000000000000000000000000
@187 00000000000000000000000000000000
@188 00000000000000000000000000000000
@189 00000000000000000000000000000000
@18a 00000000000000000000000000000000
@18b 00000000000000000000000000000000
@18c 00000000000000000000000000000000
@18d 00000000000000000000000000000000
@18e 00000000000000000000000000000000
@18f 00000000000000000000000000000000
@190 00000000000000000000000000000000
@191 00000000000000000000000000000000
@192 00000000000000000000000000000000
@193 00000000000000000000000000000000
@194 00000000000000000000000000000000
@195 00000000000000000000000000000000
@196 00000000000000000000000000000000
@197 00000000000000000000000000000000
@198 00000000000000000000000000000000
@199 00000000000000000000000000000000
@19a 00000000000000000000000000000000
@19b 00000000000000000000000000000000
@19c 00000000000000000000000000000000
@19d 00000000000000000000000000000000
@19e 00000000000000000000000000000000
@19f 00000000000000000000000000000000
```

```
@1a0 00000000000000000000000000000000
@1a1 00000000000000000000000000000000
@1a2 00000000000000000000000000000000
@1a3 00000000000000000000000000000000
@1a4 00000000000000000000000000000000
@1a5 00000000000000000000000000000000
@1a6 00000000000000000000000000000000
@1a7 00000000000000000000000000000000
@1a8 00000000000000000000000000000000
@1a9 00000000000000000000000000000000
@1aa 00000000000000000000000000000000
@1ab 00000000000000000000000000000000
@1ac 00000000000000000000000000000000
@1ad 00000000000000000000000000000000
@1ae 00000000000000000000000000000000
@1af 00000000000000000000000000000000
@1b0 00000000000000000000000000000000
@1b1 00000000000000000000000000000000
@1b2 00000000000000000000000000000000
@1b3 00000000000000000000000000000000
@1b4 00000000000000000000000000000000
@1b5 00000000000000000000000000000000
@1b6 00000000000000000000000000000000
@1b7 00000000000000000000000000000000
@1b8 00000000000000000000000000000000
@1b9 00000000000000000000000000000000
@1ba 00000000000000000000000000000000
@1bb 00000000000000000000000000000000
@1bc 00000000000000000000000000000000
@1bd 00000000000000000000000000000000
@1be 00000000000000000000000000000000
@1bf 00000000000000000000000000000000
@1c0 00000000000000000000000000000000
@1c1 00000000000000000000000000000000
@1c2 00000000000000000000000000000000
@1c3 00000000000000000000000000000000
@1c4 00000000000000000000000000000000
@1c5 00000000000000000000000000000000
```

```
@1c6 00000000000000000000000000000000
@1c7 00000000000000000000000000000000
@1c8 00000000000000000000000000000000
@1c9 00000000000000000000000000000000
@1ca 00000000000000000000000000000000
@1cb 00000000000000000000000000000000
@1cc 00000000000000000000000000000000
@1cd 00000000000000000000000000000000
@1ce 00000000000000000000000000000000
@1cf 00000000000000000000000000000000
@1d0 00000000000000000000000000000000
@1d1 00000000000000000000000000000000
@1d2 00000000000000000000000000000000
@1d3 00000000000000000000000000000000
@1d4 00000000000000000000000000000000
@1d5 00000000000000000000000000000000
@1d6 00000000000000000000000000000000
@1d7 00000000000000000000000000000000
@1d8 00000000000000000000000000000000
@1d9 00000000000000000000000000000000
@1da 00000000000000000000000000000000
@1db 00000000000000000000000000000000
@1dc 00000000000000000000000000000000
@1dd 00000000000000000000000000000000
@1de 00000000000000000000000000000000
@1df 00000000000000000000000000000000
@1e0 00000000000000000000000000000000
@1e1 00000000000000000000000000000000
@1e2 00000000000000000000000000000000
@1e3 00000000000000000000000000000000
@1e4 00000000000000000000000000000000
@1e5 00000000000000000000000000000000
@1e6 00000000000000000000000000000000
@1e7 00000000000000000000000000000000
@1e8 00000000000000000000000000000000
@1e9 00000000000000000000000000000000
@1ea 00000000000000000000000000000000
@1eb 00000000000000000000000000000000
```

```
@1ec 00000000000000000000000000000000
@1ed 00000000000000000000000000000000
@1ee 00000000000000000000000000000000
@1ef 00000000000000000000000000000000
@1f0 00000000000000000000000000000000
@1f1 00000000000000000000000000000000
@1f2 00000000000000000000000000000000
@1f3 00000000000000000000000000000000
@1f4 00000000000000000000000000000000
@1f5 00000000000000000000000000000000
@1f6 00000000000000000000000000000000
@1f7 00000000000000000000000000000000
@1f8 00000000000000000000000000000000
@1f9 00000000000000000000000000000000
@1fa 00000000000000000000000000000000
@1fb 00000000000000000000000000000000
@1fc 00000000000000000000000000000000
@1fd 00000000000000000000000000000000
@1fe 00000000000000000000000000000000
@1ff 00000000000000000000000000000000
```

After Execution:

```
 @0 00001001100000000000000001100101
 @1 0000100110011000000000000000011
 @2 00000001000000000000000001010100
 @3 00001001000100000000000000000001
 @4 00000000000101111111111111111010
 @5 00001000100000000000000000000011
 @6 00001001100000000000000001010111
 @7 10011001100110000000000000000011
 @8 00001001100110000000000000000011
 @9 00000010000111111111111111111010
 @a 11010000000000000000000000000000
 @b 10011010000100000000000000000010
 @c 00001011000110000000000000000111
 @d 00001010101101111111111111111100
 @e 00011001100110001000000000000000
 @f 01100010001000000000000000000010
@10 10001010001000000000000000000000
@11 10010010001000000000000000000000
@12 01101010001000000000000000001111
@13 00111001000000010000000000000000
@14 01110010000100000000000000000111
@15 00000000000000000000000000000011
@16 01001001100110001000000000000000
@17 00010001100000000000000010010010
@18 01000001100000001000000000000000
@19 00110010100010000000000000000000
@1a 00101001000110000000000000000000
@1b 00010010100100000000000001010100
@1c 00100000000110101000000000000000
@1d 01011001000110001000000000000000
@1e 00001010100000000000000000001000
@1f 00001011000000000000000000010111
@20 10000011001010000000000000000000
@21 11001010000000000000000000000000
@22 11000011100000000000000000000000
@23 01111011001010000000000000000000
@24 00000000000000000000000000001011
@25 00001101101101111111111111111101
```

```
@26 00001110001110000000001000101011
@27 00000000000000000000000000001011
@28 10110010000000000000000000000000
@29 00010010000000000000000101010100
@2a 00001000100000000000000010110100
@2b 00001011100000000000000000000001
@2c 00001010100000000000000000101000
@2d 10111010000000000000000000000000
@2e 00001010101011111111111111111111
@2f 10011010100000000000000000001001
@30 01100011000000000000000000001010
@31 11010000000000000000000000000000
@32 00001011001101111111111111111111
@33 10011011000101111111111111111101
@34 01001010001000111000000000000000
@35 10011010000101111111111111110111
@36 00000010000000000000000101010100
@37 10101000100000000000000000000000
@38 11010000000000000000000000000000
@39 00001010000000000000000010101010
@3a 10111010000000000000000000000000
@3b 11011000000000000000000000000000
@3c 00000000000000000000000000000000
@3d 00000000000000000000000000000000
@3e 00000000000000000000000000000000
@3f 00000000000000000000000000000000
@40 00000000000000000000000000000000
@41 00000000000000000000000000000000
@42 00000000000000000000000000000000
@43 00000000000000000000000000000000
@44 00000000000000000000000000000000
@45 00000000000000000000000000000000
@46 00000000000000000000000000000000
@47 00000000000000000000000000000000
@48 00000000000000000000000000000000
@49 00000000000000000000000000000000
@4a 00000000000000000000000000000000
@4b 00000000000000000000000000000000
```

```
@4c 00000000000000000000000000000000
@4d 00000000000000000000000000000000
@4e 00000000000000000000000000000000
@4f 00000000000000000000000000000000
@50 00000000000000000000000000000000
@51 00000000000000000000000000000000
@52 00000000000000000000000000000000
@53 00000000000000000000000000000000
@54 00000000000000000000000010010111
@55 00000000000000000000000000000000
@56 00000000000000000000000000000000
@57 00000000000000000000000000000000
@58 00000000000000000000000000000000
@59 00000000000000000000000000000000
@5a 00000000000000000000000000000000
@5b 00000000000000000000000000000000
@5c 00000000000000000000000000000000
@5d 00000000000000000000000000000000
@5e 00000000000000000000000000000000
@5f 00000000000000000000000000000000
@60 00000000000000000000000000000000
@61 00000000000000000000000000000000
@62 00000000000000000000000000000000
@63 00000000000000000000000000000000
@64 00000000000000000000000000000000
@65 00000000000000000000000000000000
@66 00000000000000000000000000000000
@67 00000000000000000000000000000000
@68 00000000000000000000000000000000
@69 00000000000000000000000000000000
@6a 00000000000000000000000000000000
@6b 00000000000000000000000000000000
@6c 00000000000000000000000000000000
@6d 00000000000000000000000000000000
@6e 00000000000000000000000000000000
@6f 00000000000000000000000000000000
@70 00000000000000000000000000000000
@71 00000000000000000000000000000000
```

```
@72 00000000000000000000000000000000
@73 00000000000000000000000000000000
@74 00000000000000000000000000000000
@75 00000000000000000000000000000000
@76 00000000000000000000000000000000
@77 00000000000000000000000000000000
@78 00000000000000000000000000000000
@79 00000000000000000000000000000000
@7a 00000000000000000000000000000000
@7b 00000000000000000000000000000000
@7c 00000000000000000000000000000000
@7d 00000000000000000000000000000000
@7e 00000000000000000000000000000000
@7f 00000000000000000000000000000000
@80 00000000000000000000000000000000
@81 00000000000000000000000000000000
@82 00000000000000000000000000000000
@83 00000000000000000000000000000000
@84 00000000000000000000000000000000
@85 00000000000000000000000000000000
@86 00000000000000000000000000000000
@87 00000000000000000000000000000000
@88 00000000000000000000000000000000
@89 00000000000000000000000000000000
@8a 00000000000000000000000000000000
@8b 00000000000000000000000000000000
@8c 00000000000000000000000000000000
@8d 00000000000000000000000000000000
@8e 00000000000000000000000000000000
@8f 00000000000000000000000000000000
@90 00000000000000000000000000000000
@91 00000000000000000000000000000000
@92 00000000000000000000000001000110
@93 00000000000000000000000000000000
@94 00000000000000000000000000000000
@95 00000000000000000000000000000000
@96 00000000000000000000000000000000
@97 00000000000000000000000000000000
```

```
@98 00000000000000000000000000000000
@99 00000000000000000000000000000000
@9a 00000000000000000000000000000000
@9b 00000000000000000000000000000000
@9c 00000000000000000000000000000000
@9d 00000000000000000000000000000000
@9e 00000000000000000000000000000000
@9f 00000000000000000000000000000000
@a0 00000000000000000000000000000000
@a1 00000000000000000000000000000000
@a2 00000000000000000000000000000000
@a3 00000000000000000000000000000000
@a4 00000000000000000000000000000000
@a5 00000000000000000000000000000000
@a6 00000000000000000000000000000000
@a7 00000000000000000000000000000000
@a8 00000000000000000000000000000000
@a9 00000000000000000000000000000000
@aa 00000000000000000000000000000000
@ab 00000000000000000000000000000000
@ac 00000000000000000000000000000000
@ad 00000000000000000000000000000000
@ae 00000000000000000000000000000000
@af 00000000000000000000000000000000
@b0 00000000000000000000000000000000
@b1 00000000000000000000000000000000
@b2 00000000000000000000000000000000
@b3 00000000000000000000000000000000
@b4 00000000000000000000000000000000
@b5 00000000000000000000000000000000
@b6 00000000000000000000000000000000
@b7 00000000000000000000000000000000
@b8 00000000000000000000000000000000
@b9 00011100000001111000000000000000
@ba 00011111110101100000000000000000
@bb 00100111010111101000000000000000
@bc 00100111111111110000000000000000
@bd 10101100000000000000000000000000
```

```
@be 00000000000000000000000000000000
@bf 00000000000000000000000000000000
@c0 00000000000000000000000000000000
@c1 00000000000000000000000000000000
@c2 00000000000000000000000000000000
@c3 00000000000000000000000000000000
@c4 00000000000000000000000000000000
@c5 00000000000000000000000000000000
@c6 00000000000000000000000000000000
@c7 00000000000000000000000000000000
@c8 00000000000000000000000000000000
@c9 00000000000000000000000000000000
@ca 00000000000000000000000000000000
@cb 00000000000000000000000000000000
@cc 00000000000000000000000000000000
@cd 00000000000000000000000000000000
@ce 00000000000000000000000000000000
@cf 00000000000000000000000000000000
@d0 00000000000000000000000000000000
@d1 00000000000000000000000000000000
@d2 00000000000000000000000000000000
@d3 00000000000000000000000000000000
@d4 00000000000000000000000000000000
@d5 00000000000000000000000000000000
@d6 00000000000000000000000000000000
@d7 00000000000000000000000000000000
@d8 00000000000000000000000000000000
@d9 00000000000000000000000000000000
@da 00000000000000000000000000000000
@db 00000000000000000000000000000000
@dc 00000000000000000000000000000000
@dd 00000000000000000000000000000000
@de 00000000000000000000000000000000
@df 00000000000000000000000000000000
@e0 00000000000000000000000000000000
@e1 00000000000000000000000000000000
@e2 00000000000000000000000000000000
@e3 00000000000000000000000000000000
```

```
@e4 00000000000000000000000000000000
@e5 00000000000000000000000000000000
@e6 00000000000000000000000000000000
@e7 00000000000000000000000000000000
@e8 00000000000000000000000000000000
@e9 00000000000000000000000000000000
@ea 00000000000000000000000000000000
@eb 00000000000000000000000000000000
@ec 00000000000000000000000000000000
@ed 00000000000000000000000000000000
@ee 00000000000000000000000000000000
@ef 00000000000000000000000000000000
@f0 00000000000000001111111111111111
@f1 00000000000000000000000000000000
@f2 00000000000000000000000000000000
@f3 00000000000000000000000000000000
@f4 00000000000000000000000000000000
@f5 00000000000000000000000000000000
@f6 00000000000000000000000000000000
@f7 00000000000000000000000000000000
@f8 00000000000000000000000000000000
@f9 00000000000000000000000000000000
@fa 00000000000000000000000000000000
@fb 00000000000000000000000000000000
@fc 00000000000000000000000000000000
@fd 00000000000000000000000000000000
@fe 00000000000000000000000000000000
@ff 00000000000000000000000000000000
@100 00000000000000000000000000000000
@101 00000000000000000000000000000000
@102 00000000000000000000000000000000
@103 00000000000000000000000000000000
@104 00000000000000000000000000000000
@105 00000000000000000000000000000000
@106 00000000000000000000000000000000
@107 00000000000000000000000000000000
@108 00000000000000000000000000000000
@109 00000000000000000000000000000000
```

```
@10a 00000000000000000000000000000000
@10b 00000000000000000000000000000000
@10c 00000000000000000000000000000000
@10d 00000000000000000000000000000000
@10e 00000000000000000000000000000000
@10f 00000000000000000000000000000000
@110 00000000000000000000000000000000
@111 00000000000000000000000000000000
@112 00000000000000000000000000000000
@113 00000000000000000000000000000000
@114 00000000000000000000000000000000
@115 00000000000000000000000000000000
@116 00000000000000000000000000000000
@117 00000000000000000000000000000000
@118 00000000000000000000000000000000
@119 00000000000000000000000000000000
@11a 00000000000000000000000000000000
@11b 00000000000000000000000000000000
@11c 00000000000000000000000000000000
@11d 00000000000000000000000000000000
@11e 00000000000000000000000000000000
@11f 00000000000000000000000000000000
@120 00000000000000000000000000000000
@121 00000000000000000000000000000000
@122 00000000000000000000000000000000
@123 00000000000000000000000000000000
@124 00000000000000000000000000000000
@125 00000000000000000000000000000000
@126 00000000000000000000000000000000
@127 00000000000000000000000000000000
@128 00000000000000000000000000000000
@129 00000000000000000000000000000000
@12a 00000000000000000000000000000000
@12b 00000000000000000000000000000000
@12c 00000000000000000000000000000000
@12d 00000000000000000000000000000000
@12e 00000000000000000000000000000000
@12f 00000000000000000000000000000000
@130 00000000000000000000000000000000
```

```
@131 00000000000000000000000000000000
@132 00000000000000000000000000000000
@133 00000000000000000000000000000000
@134 00000000000000000000000000000000
@135 00000000000000000000000000000000
@136 00000000000000000000000000000000
@137 00000000000000000000000000000000
@138 00000000000000000000000000000000
@139 00000000000000000000000000000000
@13a 00000000000000000000000000000000
@13b 00000000000000000000000000000000
@13c 00000000000000000000000000000000
@13d 00000000000000000000000000000000
@13e 00000000000000000000000000000000
@13f 00000000000000000000000000000000
@140 00000000000000000000000000000000
@141 00000000000000000000000000000000
@142 00000000000000000000000000000000
@143 00000000000000000000000000000000
@144 00000000000000000000000000000000
@145 00000000000000000000000000000000
@146 00000000000000000000000000000000
@147 00000000000000000000000000000000
@148 00000000000000000000000000000000
@149 00000000000000000000000000000000
@14a 00000000000000000000000000000000
@14b 00000000000000000000000000000000
@14c 00000000000000000000000000000000
@14d 00000000000000000000000000000000
@14e 00000000000000000000000000000000
@14f 00000000000000000000000000000000
@150 00000000000000000000000000000000
@151 00000000000000000000000000000000
@152 00000000000000000000000000000000
@153 00000000000000000000000000000000
@154 00000000000000000000000000000000
@155 00000000000000000000000000000000
@156 00000000000000000000000000000000
@157 00000000000000000000000000000000
```

```
@158 00000000000000000000000000000000
@159 00000000000000000000000000000000
@15a 00000000000000000000000000000000
@15b 00000000000000000000000000000000
@15c 00000000000000000000000000000000
@15d 00000000000000000000000000000000
@15e 00000000000000000000000000000000
@15f 00000000000000000000000000000000
@160 00000000000000000000000000000000
@161 00000000000000000000000000000000
@162 00000000000000000000000000000000
@163 00000000000000000000000000000000
@164 00000000000000000000000000000000
@165 00000000000000000000000000000000
@166 00000000000000000000000000000000
@167 00000000000000000000000000000000
@168 00000000000000000000000000000000
@169 00000000000000000000000000000000
@16a 00000000000000000000000000000000
@16b 00000000000000000000000000000000
@16c 00000000000000000000000000000000
@16d 00000000000000000000000000000000
@16e 00000000000000000000000000000000
@16f 00000000000000000000000000000000
@170 00000000000000000000000000000000
@171 00000000000000000000000000000000
@172 00000000000000000000000000000000
@173 00000000000000000000000000000000
@174 00000000000000000000000000000000
@175 00000000000000000000000000000000
@176 00000000000000000000000000000000
@177 00000000000000000000000000000000
@178 00000000000000000000000000000000
@179 00000000000000000000000000000000
@17a 00000000000000000000000000000000
@17b 00000000000000000000000000000000
@17c 00000000000000000000000000000000
@17d 00000000000000000000000000000000
@17e 00000000000000000000000000000000
```

```
@17f 00000000000000000000000000000000
@180 00000000000000000000000000000000
@181 00000000000000000000000000000000
@182 00000000000000000000000000000000
@183 00000000000000000000000000000000
@184 00000000000000000000000000000000
@185 00000000000000000000000000000000
@186 00000000000000000000000000000000
@187 00000000000000000000000000000000
@188 00000000000000000000000000000000
@189 00000000000000000000000000000000
@18a 00000000000000000000000000000000
@18b 00000000000000000000000000000000
@18c 00000000000000000000000000000000
@18d 00000000000000000000000000000000
@18e 00000000000000000000000000000000
@18f 00000000000000000000000000000000
@190 00000000000000000000000000000000
@191 00000000000000000000000000000000
@192 00000000000000000000000000000000
@193 00000000000000000000000000000000
@194 00000000000000000000000000000000
@195 00000000000000000000000000000000
@196 00000000000000000000000000000000
@197 00000000000000000000000000000000
@198 00000000000000000000000000000000
@199 00000000000000000000000000000000
@19a 00000000000000000000000000000000
@19b 00000000000000000000000000000000
@19c 00000000000000000000000000000000
@19d 00000000000000000000000000000000
@19e 00000000000000000000000000000000
@19f 00000000000000000000000000000000
@1a0 00000000000000000000000000000000
@1a1 00000000000000000000000000000000
@1a2 00000000000000000000000000000000
@1a3 00000000000000000000000000000000
@1a4 00000000000000000000000000000000
@1a5 00000000000000000000000000000000
```

```
@1a6 00000000000000000000000000000000
@1a7 00000000000000000000000000000000
@1a8 00000000000000000000000000000000
@1a9 00000000000000000000000000000000
@1aa 00000000000000000000000000000000
@1ab 00000000000000000000000000000000
@1ac 00000000000000000000000000000000
@1ad 00000000000000000000000000000000
@1ae 00000000000000000000000000000000
@1af 00000000000000000000000000000000
@1b0 00000000000000000000000000000000
@1b1 00000000000000000000000000000000
@1b2 00000000000000000000000000000000
@1b3 00000000000000000000000000000000
@1b4 00000000000000000000000000000000
@1b5 00000000000000000000000000000000
@1b6 00000000000000000000000000000000
@1b7 00000000000000000000000000000000
@1b8 00000000000000000000000000000000
@1b9 00000000000000000000000000000000
@1ba 00000000000000000000000000000000
@1bb 00000000000000000000000000000000
@1bc 00000000000000000000000000000000
@1bd 00000000000000000000000000000000
@1be 00000000000000000000000000000000
@1bf 00000000000000000000000000000000
@1c0 00000000000000000000000000000000
@1c1 00000000000000000000000000000000
@1c2 00000000000000000000000000000000
@1c3 00000000000000000000000000000000
@1c4 00000000000000000000000000000000
@1c5 00000000000000000000000000000000
@1c6 00000000000000000000000000000000
@1c7 00000000000000000000000000000000
@1c8 00000000000000000000000000000000
@1c9 00000000000000000000000000000000
@1ca 00000000000000000000000000000000
@1cb 00000000000000000000000000000000
@1cc 00000000000000000000000000000000
```

```
@1cd 00000000000000000000000000000000
@1ce 00000000000000000000000000000000
@1cf 00000000000000000000000000000000
@1d0 00000000000000000000000000000000
@1d1 00000000000000000000000000000000
@1d2 00000000000000000000000000000000
@1d3 00000000000000000000000000000000
@1d4 00000000000000000000000000000000
@1d5 00000000000000000000000000000000
@1d6 00000000000000000000000000000000
@1d7 00000000000000000000000000000000
@1d8 00000000000000000000000000000000
@1d9 00000000000000000000000000000000
@1da 00000000000000000000000000000000
@1db 00000000000000000000000000000000
@1dc 00000000000000000000000000000000
@1dd 00000000000000000000000000000000
@1de 00000000000000000000000000000000
@1df 00000000000000000000000000000000
@1e0 00000000000000000000000000000000
@1e1 00000000000000000000000000000000
@1e2 00000000000000000000000000000000
@1e3 00000000000000000000000000000000
@1e4 00000000000000000000000000000000
@1e5 00000000000000000000000000000000
@1e6 00000000000000000000000000000000
@1e7 00000000000000000000000000000000
@1e8 00000000000000000000000000000000
@1e9 00000000000000000000000000000000
@1ea 00000000000000000000000000000000
@1eb 00000000000000000000000000000000
@1ec 00000000000000000000000000000000
@1ed 00000000000000000000000000000000
@1ee 00000000000000000000000000000000
@1ef 00000000000000000000000000000000
@1f0 00000000000000000000000000000000
@1f1 00000000000000000000000000000000
@1f2 00000000000000000000000000000000
```

```
@1f3 00000000000000000000000000000000
@1f4 00000000000000000000000000000000
@1f5 00000000000000000000000000000000
@1f6 00000000000000000000000000000000
@1f7 00000000000000000000000000000000
@1f8 00000000000000000000000000000000
@1f9 00000000000000000000000000000000
@1fa 00000000000000000000000000000000
@1fb 00000000000000000000000000000000
@1fc 00000000000000000000000000000000
@1fd 00000000000000000000000000000000
@1fe 00000000000000000000000000000000
@1ff 00000000000000000000000000000000
```

Appendix 5: Image of the FPGA board during operation

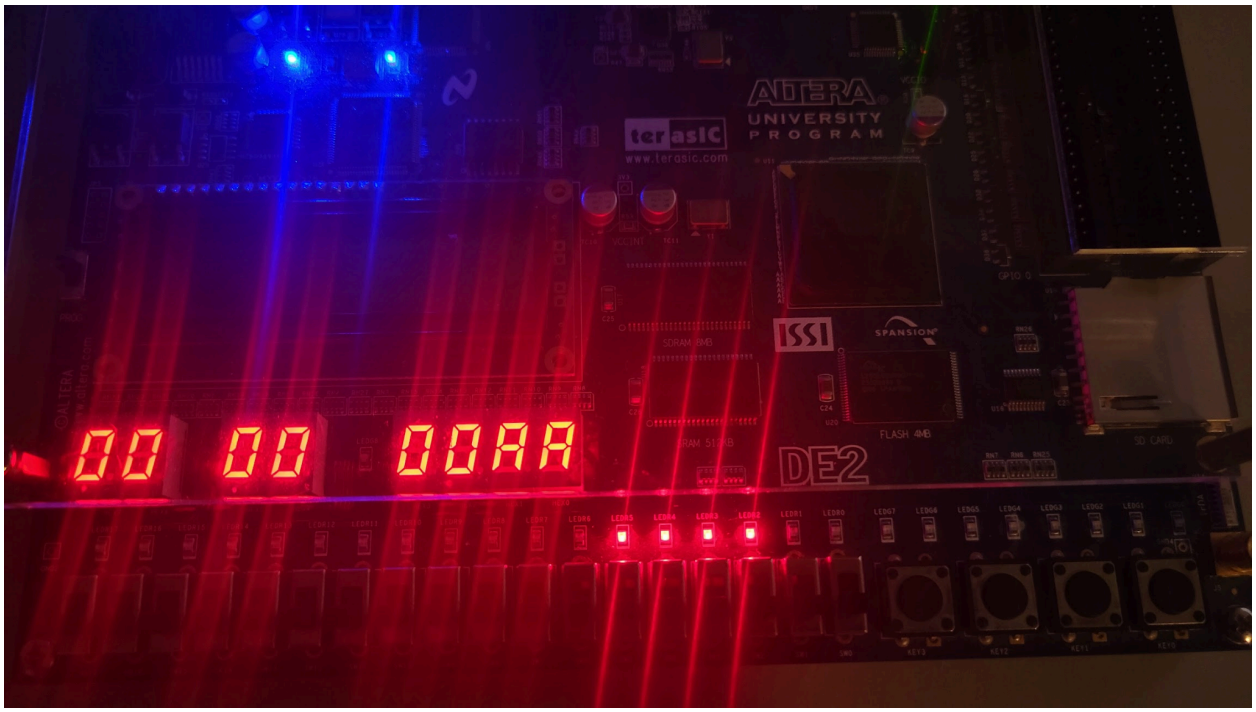


Image of the Cyclone II DE2 FPGA Board upon completion of the phase 4 program.

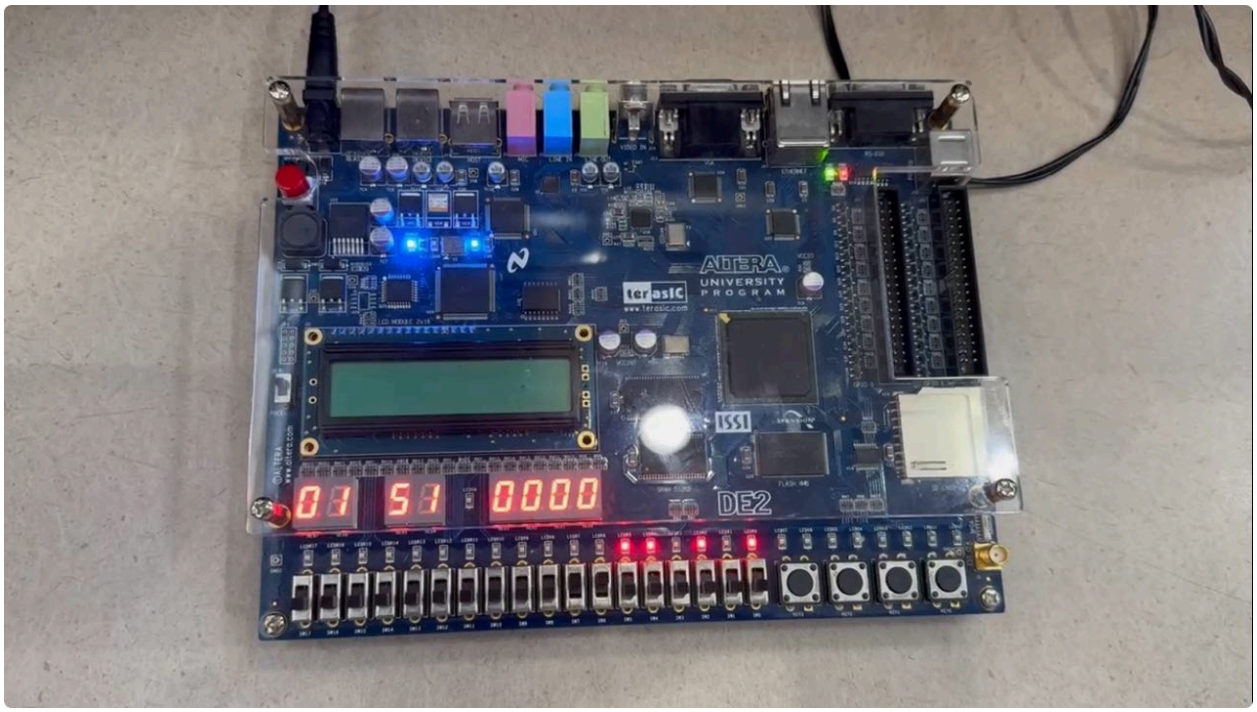


Image of the Cyclone II DE2 FPGA Board mid execution of the phase 4 program.