Department of Electrical and Computer Engineering
Queen's University

# ELEC 374 Digital Systems Engineering
# Laboratory Project

Winter 2024

## Designing a Simple RISC Computer: Phase 1

## 1. Objectives

The objective of this project is to design, simulate, implement, and verify a simple RISC Computer (Mini SRC), consisting of a simple RISC processor, memory, and I/O. Phase 1 of this project consists of the design and Functional Simulation of a part of the Mini SRC datapath. In this Phase, you will design the necessary logic and simulate the *add, sub, mul, div, and, or, shr, shra, shl, ror, rol, neg,* and *not* instructions. The "Select and Encode" logic, "CON FF" Logic, "Input/Output" ports, "Memory Subsystem" and load/store instructions, branch and jump instructions, as well as *addi, andi*, and *ori* instructions will be designed and simulated in Phase 2. The complete datapath is shown in Figure 1.

Design input can be done entirely using Verilog or VHDL. For your design, consult the Lecture Slides on Verilog or VHDL that have various examples including multiplexers, encoders, registers, memory units, clock generation, state machines, etc. You are also advised to refer to Section 6, "A Simple Datapath Design in Verilog", of the Introduction to Intel Quartus Prime Design Software, ModelSim-Intel FPGA Starter Edition, and DE0-CV Development Board tutorial document as a starting point for designing your registers, bus, data transfer over the bus, testbenches, etc.

You may use components available in the Quartus Prime Library. However, you must design your own advanced multiplication circuitry (using the 32x32 Booth algorithm with bit-pair recoding of multiplier, or using Carry-Save Adders for the addition of the summands, whichever you prefer) for the multiplication instruction. Note that you are NOT allowed to use simple arithmetic operators in an HDL language for the *Arithmetic Logic Unit* (ALU) implementation of this project, such as the adder or subtractor circuitry, among others. However, you are allowed to use + or – arithmetic operators in the implementation of your multiplication or division circuitry, as the intention here is on the implementation of a particular algorithm that we discussed in class for the multiplication/division operation.

## 2. Preliminaries

**2.1 DataPath:** Figure 1 illustrates a simplified single-bus Datapath for the Mini SRC (see Figure 4.2 on page 143 and Figure 4.3 on page 148 of the Lab Reader). As shown in Figure 1, the datapath consists of a 32-bit bus, BUS. The bus is responsible for transferring the information among different components of the system. There can be only one transaction at a time on the bus. There are sixteen 32-bit registers *R0* through *R15* in the Mini SRC, as discussed in the CPU specification document. There are also two dedicated 32-bit registers *HI* and *LO* for holding the result of a multiplication or a division operation. The 32-bit *Instruction Register*, *IR*, holds the current instruction. The 32-bit *Program Counter*, *PC*, points to the address of the next instruction after the execution of the current instruction. *PC* is incremented by 1 during the instruction fetch, using *IncPC* control signal in the ALU. You may opt for a hardware incrementer for PC outside ALU. In general, you are welcome to come up with your own ideas and design decisions during the entire CPU design project.

The ALU has two inputs, A and B, and an output, C. Because the bus can support only one transaction at a time, one of the inputs (A input) to the ALU needs to be stored in the 32-bit temporary register, *Y*. As

discussed in the CPU Specification document, the ALU supports the addition, subtraction, multiplication, division, shift right, shift right arithmetic, shift left, rotate right, rotate left, logical AND, Logical OR, negate, and NOT operations. The control signals to the ALU (generated by the Control Unit in Phase 3, as shown in Figure 5) will enforce the required operation. These control signals include *ADD, SUB, MUL, DIV, SHR, SHRA, SHL, ROR, ROL, AND, OR, NEG,* and *NOT* control signals, among others. Note that in Phase 1 and Phase 2, you simulate such control signals. The *Z* register holds the result of the operation in ALU and will be able to drive the Bus in the next clock cycle when the bus is free. The *Z* register is 64-bit long to hold the result of a multiplication (product) or a division (remainder in the higher byte, and quotient in the lower byte) operation temporarily before loading the *HI* and *LO* registers. You may need a multiplexer between the Y register and the A input of the ALU for any other potential input. You may also need to include a simple circuitry (such as a multiplexer) between the ALU output C and the *Z* register. Depending on the current instruction in the Instruction Register, *IR,* this logic selects the output of one of the ALU units to drive the *Z* register.

The *Memory Address register* (*MAR*) holds the address of a memory location. The *Memory Data Register* (*MDR*) holds either the data read from memory, or the data to be written into the memory. The *Select and Encode Logic* allows transfer of the contents of a register onto the bus, as well as loading the registers with the contents of the bus. The *CON FF Logic* is used to determine if the condition is met to allow branching to take place. The *Input* (In.Port) and *Output* (*Out.Port*) registers are 32-bit registers each, and are used to connect the CPU to the outside world.

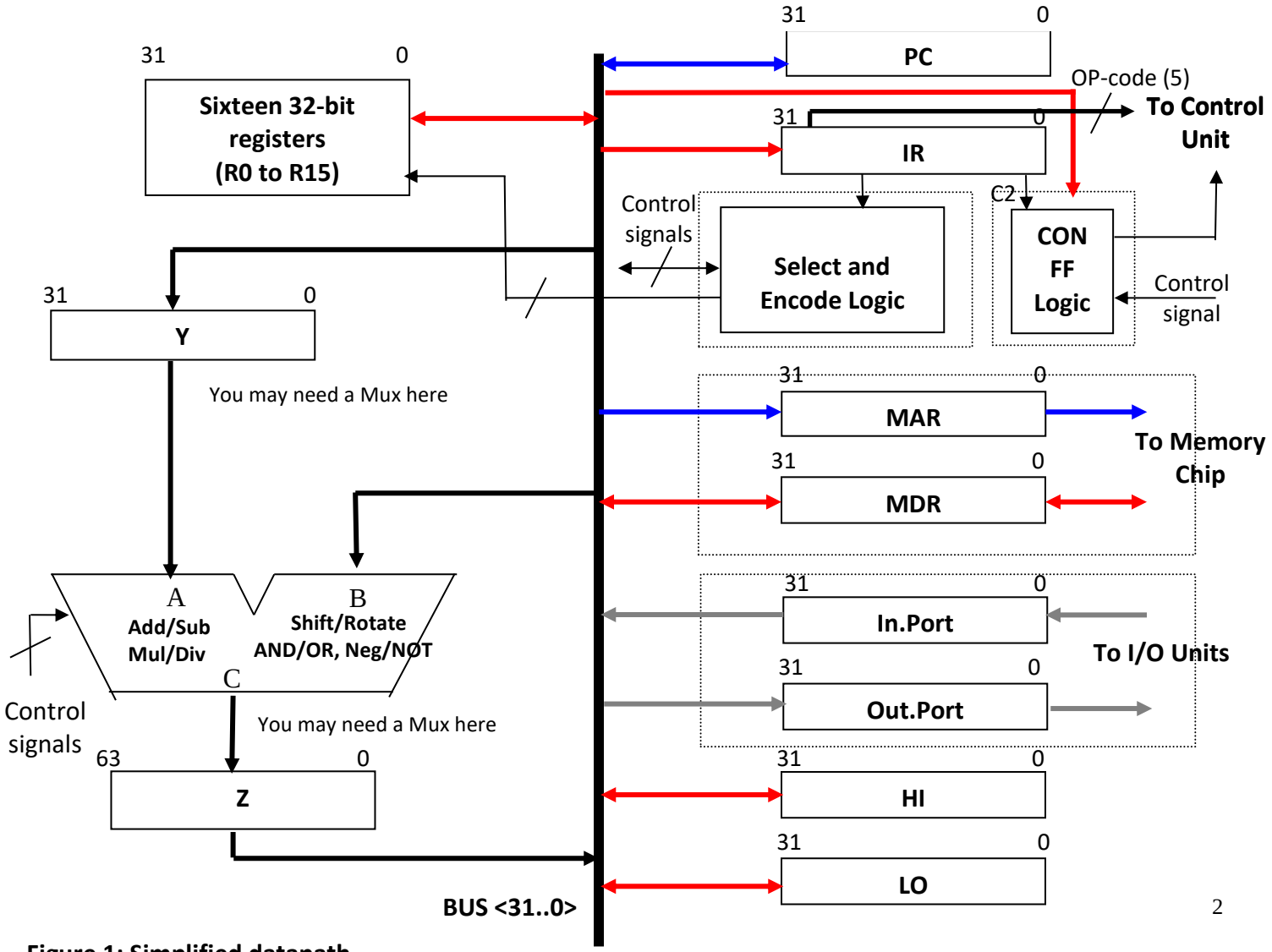


**Figure 1: Simplified datapath**

As a reminder, the "Select and Encode logic", "CON FF Logic", "Input/Output ports", "Memory Subsystem" and load/store instructions, branch and jump instructions, as well as *addi, andi*, and *ori* instructions will be tackled in Phase 2. The information provided here is for the sake of the completeness in describing the datapath. More information about these units will be provided in Phase 2.

**A Typical Register:** Figure 2 shows the block diagram for a typical register, such as *R1* (there will be a minor revision to *R0* circuitry that we will discuss in Phase 2 when we design the "Select and Encode Logic"). The input to the register, **BusMuxOut**, is coming directly from the bus. The contents of the bus is saved onto the register using the synchronous *Clock* signal and the *R1in* signal. The *R1in* signal is the control signal that allows the data from the bus to be written into the register *R1*. The *R1out* signal is the control signal that allows the contents of *R1* to be placed on the bus, through the **BusMuxIn-R1** input (see Figure 3). The *Clear* signal is used to reset the registers to a known state.

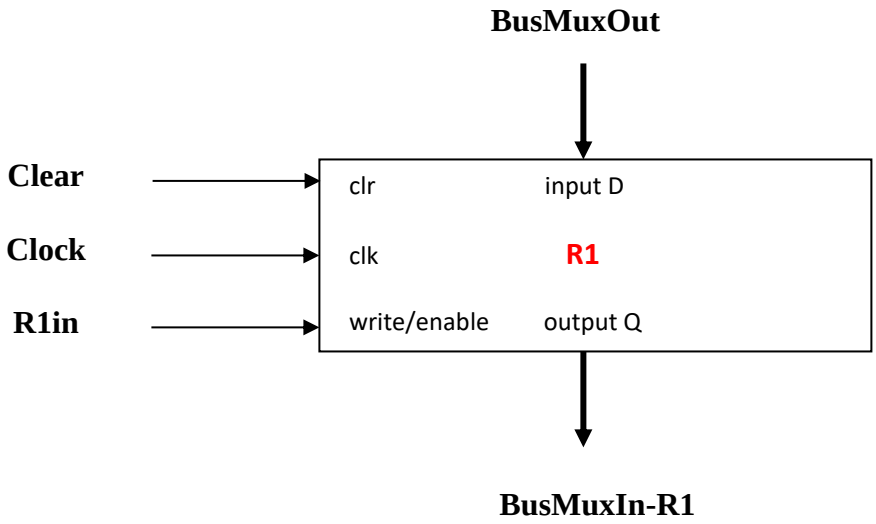


**Figure 2: A typical register**

**Bus design:** One of the important aspects of the Mini SRC datapath is its bus. The Bus may be implemented by a multiplexer and an encoder, tri-state buffers, or other techniques. Figure 3 shows a typical bus design using the multiplexer/encoder approach. The Mini SRC Bus is implemented using a 32:1 Multiplexer, **BusMux**, with five select input signals coming from a 32-to-5 encoder. The idea is to choose only one of the registers *R0 to R15, HI, LO,* $Z_{high}$, $Z_{low}$, *PC, MDR, In.Port,* or the sign-extended version of the constant C, as the source of the bus. The output of the BusMux is **BusMuxOut**. The inputs to the 32-to-5 encoder, which could select any of the above registers, are the control signals generated by the Control Unit (in Phase 3) or by the "Select and Encode Logic" (in Phase 2). However, in Phase 1, we just simulate these signals.

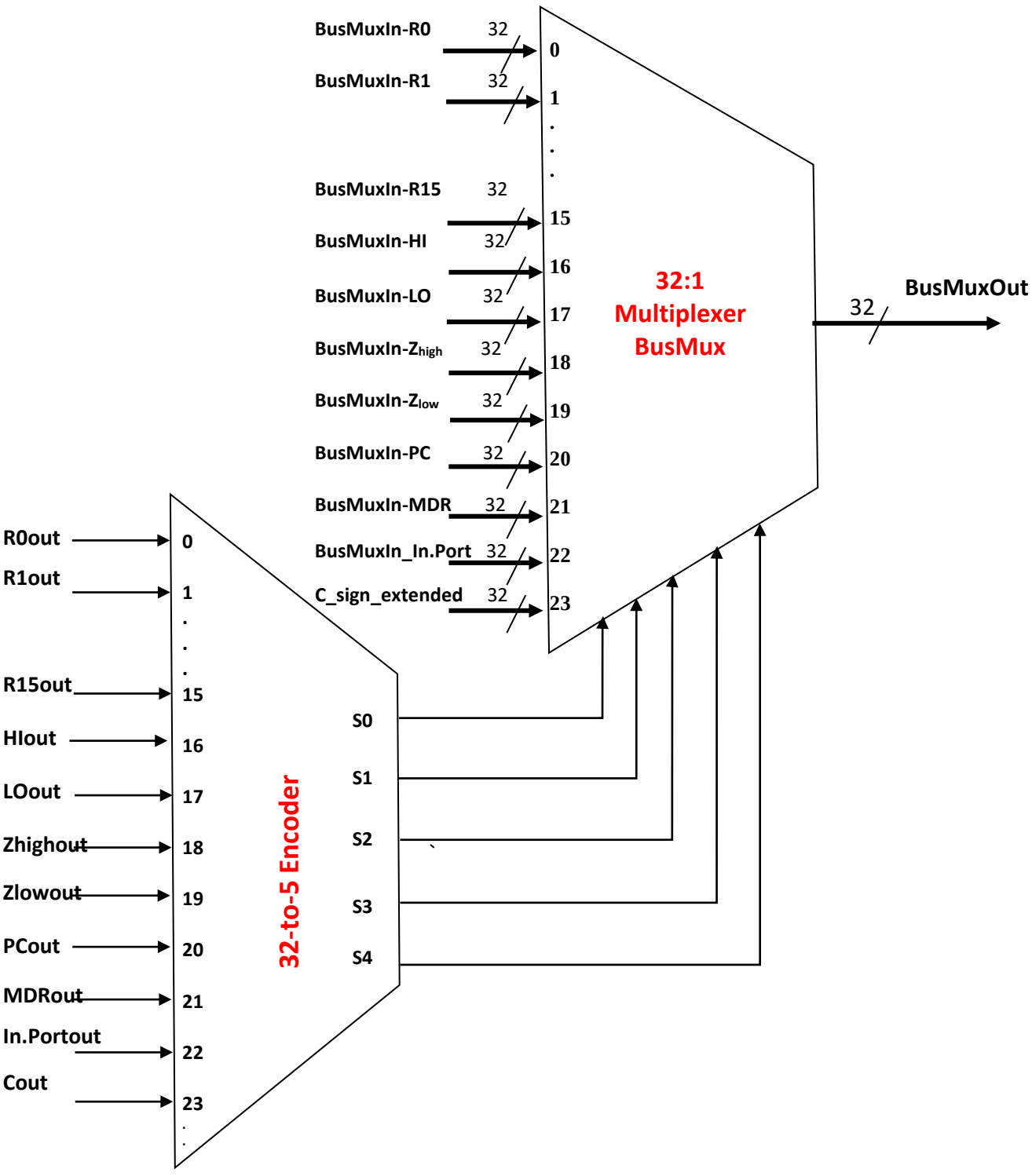


**Figure 3: A typical Bus**

**Memory Data Register:** The *Memory Data Register (MDR)* is different from the other registers in the sense that it has two input sources and two output sinks. Figure 4 presents how *MDR* is connected to the memory bus, and to the internal bus. The inputs to the *MDR* comes from the memory unit (**Mdatain**) or from the Bus (**BusMuxOut**). Data is stored in the *MDR* using the synchronous *Clock* signal and the *MDRin* control signal. The *MDR* contents can be written into the memory chip or drive the Bus.

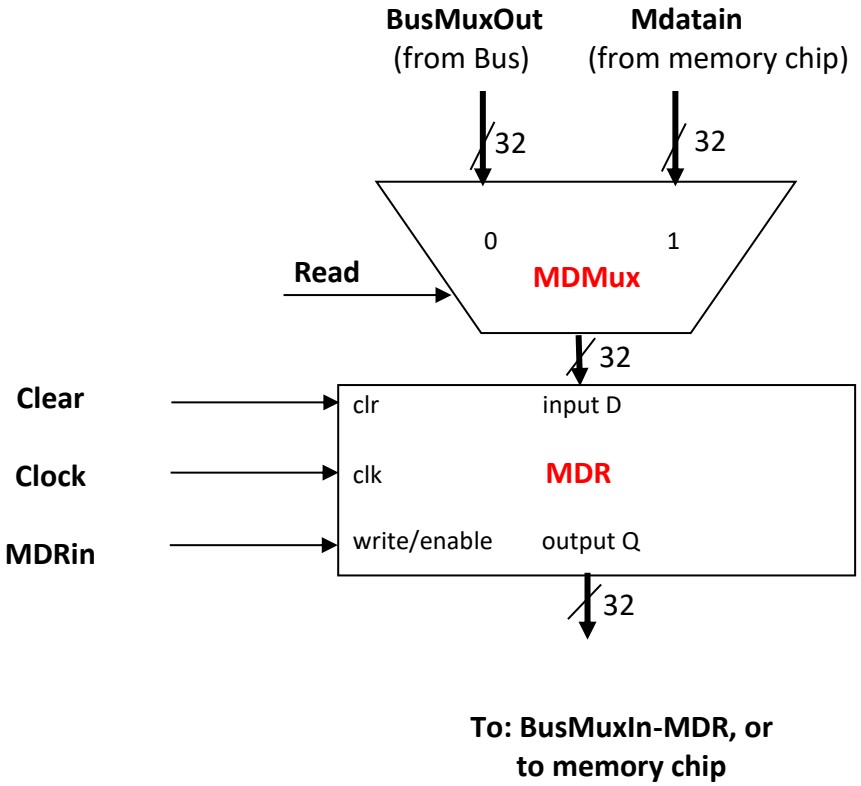


**Figure 4: The MDR unit**

**2.2 Control Unit:** The Control unit is to be designed in Phase 3. However, a block diagram is provided in Figure 5 for a better understanding of the Datapath. The Control Unit is at the heart of the processor. It accepts as inputs those signals that are needed to operate the processor and provides as outputs all the control signals necessary to execute the instructions. The outputs from the Control Unit are the control signals that we use to generate Control Sequences for the instructions of the Mini SRC.

Please note that you should not be concerned about instruction decoding in Phase 1 or Phase 2 of this project. Instruction decoding will be done in Phase3 using Verilog or VHDL. The details of the Control Unit will be discussed in Phase 3.

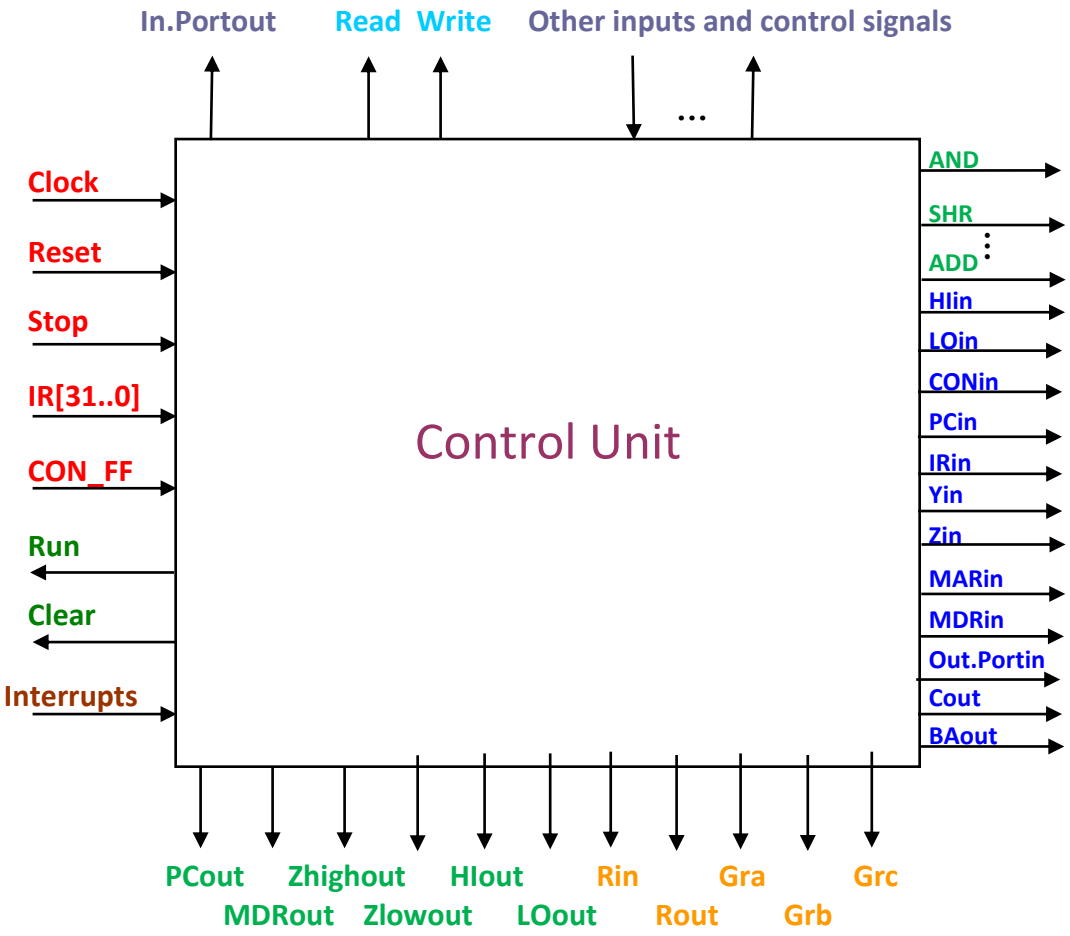


**Figure 5: Block diagram of the Control Unit**

## 3. Lab Procedure

Design the datapath shown in Figure 1 (except for the "Select and Encode Logic", "CON FF Logic", "Memory chip in the Memory Subsystem", and "Input/Output ports" units) using an all HDL (Verilog or VHDL) in Intel Quartus Prime design software. You are highly advised to follow the general steps below, and those mentioned in the CPU Design Project Tutorial, when designing and implementing your datapath. In each step of the way, functionally simulate your design to check if it works properly.

1. Design the registers R0 to R15, PC, IR, Y, Z, MAR, HI, and LO (see Figures 1 and 2).
2. Design the bidirectional Bus (Figure 3) and connect (a few of) the registers that you designed in Step 1 to the Bus. Functionally simulate your design and check if data transfer is done properly between the registers over the bus.
3. Design the MDR unit (Figure 4) and connect it to the BUS. Leave the rest of the Memory Subsystem (connection to the memory chip) for Phase 2.
4. Design your ALU. Start with simple ALU operations such as logical AND, OR, etc. Then, design the more involved operations such as ADD/SUB, MUL, and DIV circuitry. Finally, design the rest of the ALU operations.

- For the multiplication unit, you are to design your own advanced multiplication circuitry (using the 32x32 Booth algorithm with bit-pair recoding of multiplier, or using Carry-Save Adders for the addition of the summands, whichever you prefer) for the multiplication operation.
- You are welcome to design and simulate any other advanced design techniques that you have learned in class for a bonus mark.

You are highly advised to functionally simulate your design as you go through the different design steps. In order to test the Datapath by Functional Simulation, the following control and output signals may be required. In Phase 3, these control signals will be generated by the Control Unit.

Control Signals: **R0in, R0out; R1in, R1out; …; R15in, R15out; HIin; HIout; LOin; LOout; PCin, PCout; IRin; Zin; Zhighout, Zlowout; Yin; MARin; MDRin, MDRout; Read; Mdatain[31..0]**

Outputs: **R0, R1, …, R15, HI, LO, IR, BusMuxOut, Z** (minimum required output signals for demo to TA in lab), and any other outputs you may wish to observe or show in your simulation.

Using the following control sequences, test your design for *and, or, add, sub, mul, div, shr, shra, shl, ror, rol, neg,* and *not* instructions.

**3.1** In the lab, demonstrate that your Logical AND circuitry and datapath work correctly by simulating the Control Sequence for the logical ***and R1, R2, R3*** instruction (similar to Table 4.7 on page 155 of the Lab Reader for the "add" instruction), modified for the Datapath in isolation, as follows:

**Control Sequence:**

| Step | Control Sequence |
|---|---|
| T0 | PCout, MARin, IncPC, Zin |
| T1 | Zlowout, PCin, Read, Mdatain[31..0], MDRin |
| T2 | MDRout, IRin |
| T3 | R2out, Yin |
| T4 | R3out, AND, Zin |
| T5 | Zlowout, R1in |

Notes: T0 , T1, and T2 steps are used for the instruction fetch.

In T1, Mdatain[31..0] should be set to the 32-bit pattern for the *and R1, R2, R3* instruction. Its pattern can be determined by referring to the specification of the Mini SRC.
In Phase 2, the 32-bit pattern will directly come from the memory chip.
Do NOT consider instruction decoding in Phase 1 and Phase 2. It will be done in Phase 3.

As demonstrated in the tutorial on Intel Quartus Prime and ModelSim-Intel, to simulate your design using ModelSim, you will need to write a testbench program in VHDL or Verilog. Here is a sample testbench

template in VHDL for the logical AND instruction, ***and R1, R2, R3***, which you may need to verify and revise for your AND circuitry, and for other instructions.

```
-- and datapath_tb.vhd file: <This is the filename>
LIBRARY ieee;
USE ieee.std_logic_1164.all;

-- entity declaration only; no definition here
ENTITY datapath_tb IS
END ENTITY datapath_tb;

-- Architecture of the testbench with the signal names
ARCHITECTURE datapath_tb_arch OF datapath_tb IS     -- Add any other signals to see in your simulation
   SIGNAL        PCout_tb, Zlowout_tb, MDRout_tb, R2out_tb, R3out_tb:          std_logic;
   SIGNAL        MARin_tb, Zin_tb, PCin_tb, MDRin_tb, IRin_tb, Yin_tb:         std_logic;
   SIGNAL        IncPC_tb, Read_tb, AND_tb, R1in_tb, R2in_tb, R3in_tb:         std_logic;
   SIGNAL        Clock_tb:                                                     std_logic;
   SIGNAL        Mdatain_tb              :                                     std_logic_vector (31 downto 0);

    TYPE         State IS (default, Reg_load1a, Reg_load1b, Reg_load2a, Reg_load2b, Reg_load3a, Reg_load3b, T0, T1,
                           T2, T3, T4, T5);
    SIGNAL       Present_state:  State := default;

      -- component instantiation of the datapath
 COMPONENT datapath
    PORT (
                 PCout, Zlowout, MDRout, R2out, R3out:                 in      std_logic;
                 MARin, Zin, PCin, MDRin, IRin, Yin:                   in      std_logic;
                 IncPC, Read, AND, R1in, R2in, R3in:                   in      std_logic;
                 Clock:                                                in      Std_logic;
                 Mdatain:                                              in      std_logic_vector (31 downto 0);
END COMPONENT datapath;

 BEGIN
 DUT  : datapath
--port mapping: between the dut and the testbench signals
  PORT MAP (
        PCout          =>      PCout_tb,
        Zlowout        =>      Zlowout_tb,
        MDRout         =>      MDRout_tb,
        R2out          =>      R2out_tb,
        R3out          =>      R3out_tb,
        MARin          =>      MARin_tb,
        Zin            =>      Zin_tb,
        PCin           =>      PCin_tb,
        MDRin          =>      MDRin_tb,
        IRin           =>      IRin_tb,
        Yin            =>      Yin_tb,
```

```
        IncPC           =>      IncPC_tb,
        Read            =>      Read_tb,
        AND             =>      AND_tb,

        R1in            =>      R1in_tb,
        R2in            =>      R2in_tb,
        R3in            =>      R3in_tb,
        Clock           =>      Clock_tb,
        Mdatain         =>      Mdatain_tb);

--add test logic here
        Clock_process: PROCESS IS
        BEGIN
                Clock_tb <= '1', '0' after 10 ns;
                Wait for 20 ns;
        END PROCESS Clock_process;

        PROCESS (Clock_tb)  IS          -- finite state machine
        BEGIN
                IF (rising_edge (Clock_tb)) THEN                -- if clock rising-edge
                        CASE Present_state IS
                                WHEN Default =>
                                        Present_state <= Reg_load1a;
                                WHEN Reg_load1a =>
                                        Present_state <= Reg_load1b;
                                WHEN Reg_load1b =>
                                        Present_state <= Reg_load2a;
                                WHEN Reg_load2a =>
                                        Present_state <= Reg_load2b;
                                WHEN Reg_load2b =>
                                        Present_state <= Reg_load3a;
                                WHEN Reg_load3a =>
                                        Present_state <= Reg_load3b;
                                WHEN Reg_load3b =>
                                        Present_state <= T0;
                                WHEN T0 =>
                                        Present_state <= T1;
                                WHEN T1 =>
                                        Present_state <= T2;
                                WHEN T2 =>
                                        Present_state <= T3;
                                WHEN T3 =>
                                        Present_state <= T4;
                                WHEN T4 =>
                                        Present_state <= T5;
                                WHEN OTHERS =>
                        END CASE;
                END IF;
```

```
END PROCESS;

PROCESS (Present_state) IS       -- do the required job in each state
BEGIN
        CASE Present_state IS      -- assert the required signals in each clock cycle
                WHEN Default =>
                    PCout_tb <= '0';  Zlowout_tb <= '0';  MDRout_tb <= '0';          -- initialize the signals
                    R2out_tb <= '0';  R3out_tb <= '0';  MARin_tb <= '0';  Zin_tb <= '0';
                    PCin_tb <='0';  MDRin_tb <= '0';  IRin_tb  <= '0';  Yin_tb <= '0';
                    IncPC_tb <= '0';  Read_tb <= '0';  AND_tb <= '0';
                    R1in_tb <= '0';  R2in_tb <= '0';  R3in_tb <= '0'; Mdatain_tb <= x"00000000";

                WHEN Reg_load1a =>
                     Mdatain_tb <= x"00000012";
                     Read_tb <= '0', '1' after 10 ns, '0' after 25 ns;  -- the first zero is there for completeness
                     MDRin_tb <= '0', '1' after 10 ns, '0' after 25 ns;
                WHEN Reg_load1b =>
                     MDRout_tb <= '1' after 10 ns, '0' after 25 ns;
                     R2in_tb <= '1' after 10 ns, '0' after 25 ns;       --  initialize R2 with the value $12
                WHEN Reg_load2a =>
                     Mdatain_tb <= x"00000014";
                     Read_tb <= '1' after 10 ns, '0' after 25 ns;
                     MDRin_tb <= '1' after 10 ns, '0' after 25 ns;
                WHEN Reg_load2b =>
                     MDRout_tb <= '1' after 10 ns, '0' after 25 ns;
                     R3in_tb <= '1' after 10 ns, '0' after 25 ns;       --  initialize R3 with the value $14
                WHEN Reg_load3a =>
                     Mdatain_tb <= x"00000018";
                     Read_tb <= '1' after 10 ns, '0' after 25 ns;
                     MDRin_tb <= '1' after 10 ns, '0' after 25 ns;
                WHEN Reg_load3b =>
                     MDRout_tb <= '1' after 10 ns, '0' after 25 ns;
                     R1in_tb <= '1' after 10 ns, '0' after 25 ns;       --  initialize R1 with the value $18

                WHEN T0 =>                                  -- see if you need to de-assert these signals
                     PCout_tb <= '1';  MARin_tb <= '1';  IncPC_tb <= '1';  Zin_tb <= '1';
                WHEN T1 =>
                     Zlowout_tb <= '1';  PCin_tb <= '1';  Read_tb <= '1';  MDRin_tb <= '1';
                     Mdatain_tb <= x"28918000";                     -- opcode for "and R1, R2, R3"
                WHEN T2 =>
                     MDRout_tb <= '1';  IRin_tb <= '1';
                WHEN T3 =>
                     R2out_tb <= '1';  Yin_tb <= '1';
                WHEN T4 =>
                     R3out_tb <= '1';  AND_tb <= '1';  Zin_tb <= '1';
                WHEN T5 =>
                     Zlowout_tb <= '1';  R1in_tb <= '1';
                WHEN OTHERS =>
```

```
                END CASE;
        END PROCESS;
END ARCHITECTURE datapath_tb_arch;
```

Here is a sample testbench template in Verilog for the logical AND instruction, ***and R1, R2, R3***, which you may need to verify and revise for your AND circuitry, and for other instructions.

```
// and datapath_tb.v file: <This is the filename>
`timescale 1ns/10ps
module datapath_tb;
   reg    PCout, Zlowout, MDRout, R2out, R3out;        // add any other signals to see in your simulation
   reg    MARin, Zin, PCin, MDRin, IRin, Yin;
   reg    IncPC, Read, AND, R1in, R2in, R3in;
   reg    Clock;
   reg    [31:0] Mdatain;

    parameter    Default = 4'b0000, Reg_load1a = 4'b0001, Reg_load1b = 4'b0010, Reg_load2a = 4'b0011,
                 Reg_load2b = 4'b0100, Reg_load3a = 4'b0101, Reg_load3b = 4'b0110, T0 = 4'b0111,
                 T1 = 4'b1000, T2 = 4'b1001, T3 = 4'b1010, T4 = 4'b1011, T5 = 4'b1100;
   reg    [3:0] Present_state = Default;

Datapath DUT(PCout, Zlowout, MDRout, R2out, R3out, MARin, Zin, PCin, MDRin, IRin, Yin, IncPC, Read, AND, R1in,
R2in, R3in, Clock, Mdatain);

// add test logic here
initial
   begin
     Clock = 0;
     forever #10 Clock = ~ Clock;
end

always @(posedge Clock)          // finite state machine; if clock rising-edge
  begin
    case (Present_state)
          Default          :     Present_state = Reg_load1a;
          Reg_load1a       :     Present_state = Reg_load1b;
          Reg_load1b       :     Present_state = Reg_load2a;
          Reg_load2a       :     Present_state = Reg_load2b;
          Reg_load2b       :     Present_state = Reg_load3a;
          Reg_load3a       :     Present_state = Reg_load3b;
          Reg_load3b       :     Present_state = T0;
          T0               :     Present_state = T1;
          T1               :     Present_state = T2;
          T2               :     Present_state = T3;
          T3               :     Present_state = T4;
          T4               :     Present_state = T5;
```

```
    endcase
  end

always @(Present_state)          // do the required job in each state
  begin
    case (Present_state)          // assert the required signals in each clock cycle
        Default: begin
                        PCout <= 0;   Zlowout <= 0;   MDRout <= 0;          // initialize the signals
                        R2out <= 0;   R3out <= 0;   MARin <= 0;   Zin <= 0;
                        PCin <=0;   MDRin <= 0;   IRin  <= 0;   Yin <= 0;
                        IncPC <= 0;   Read <= 0;   AND <= 0;
                        R1in <= 0; R2in <= 0; R3in <= 0; Mdatain <= 32'h00000000;
        end
        Reg_load1a: begin
                        Mdatain <= 32'h00000012;
                        Read = 0; MDRin = 0;              // the first zero is there for completeness
                        #10 Read <= 1; MDRin <= 1;
                        #15 Read <= 0; MDRin <= 0;
        end
        Reg_load1b: begin
                        #10 MDRout <= 1; R2in <= 1;
                        #15 MDRout <= 0; R2in <= 0;    // initialize R2 with the value $12
        end
        Reg_load2a: begin
                        Mdatain <= 32'h00000014;
                        #10 Read <= 1; MDRin <= 1;
                        #15 Read <= 0; MDRin <= 0;
        end
        Reg_load2b: begin
                        #10 MDRout <= 1; R3in <= 1;
                        #15 MDRout <= 0; R3in <= 0;      // initialize R3 with the value $14
        end
        Reg_load3a: begin
                        Mdatain <= 32'h00000018;
                        #10 Read <= 1; MDRin <= 1;
                        #15 Read <= 0; MDRin <= 0;
        end
        Reg_load3b: begin
                        #10 MDRout <= 1; R1in <= 1;
                        #15 MDRout <= 0; R1in <= 0;      // initialize R1 with the value $18
        end

        T0: begin                                                // see if you need to de-assert these signals
                        PCout <= 1; MARin <= 1; IncPC <= 1; Zin <= 1;
        end
        T1: begin
                        Zlowout <= 1; PCin <= 1; Read <= 1; MDRin <= 1;
                        Mdatain <= 32'h28918000;                 // opcode for "and R1, R2, R3"
```

```
        end
        T2: begin
                        MDRout <= 1; IRin <= 1;
        end
        T3: begin
                        R2out <= 1; Yin <= 1;
        end
        T4: begin
                        R3out <= 1; AND <= 1; Zin <= 1;
        end
        T5: begin
                        Zlowout <= 1; R1in <= 1;
        end
      endcase
  end
endmodule
```

**3.2** Demonstrate that your Logical OR design works fine by simulating the Control Sequence for the ***or R1, R2, R3*** instruction. The Control Sequence is the same as the one for the *and* instruction except for using the OR control signal in T4 instead of the AND signal.

**3.3** Demonstrate that your Adder works correctly by simulating the Control Sequence for the ***add R1, R2, R3*** instruction, modified for the Datapath in isolation, as follows:

**Control Sequence:**

| **Step** | **Control Sequence** |
|---|---|
| T0 | PCout, MARin, IncPC, Zin |
| T1 | Zlowout, PCin, Read, Mdatain[31..0], MDRin |
| T2 | MDRout, IRin |
| T3 | R2out, Yin |
| T4 | R3out, ADD, Zin |
| T5 | Zlowout, R1in |

**3.4** Demonstrate that your Subtractor circuitry works fine by simulating the Control Sequence for the ***sub R1, R2, R3*** instruction. The Control Sequence is the same as the one used for the *add* instruction except for using the SUB control signal in T4 instead of the ADD signal.

**3.5** Demonstrate that your Multiplication circuitry works correctly by simulating the Control Sequence for the ***mul R4, R5*** instruction.

**Control Sequence:**

| Step | Control Sequence |
|---|---|
| T0 | PCout, MARin, IncPC, Zin |
| T1 | Zlowout, PCin, Read, Mdatain[31..0], MDRin |
| T2 | MDRout, IRin |
| T3 | R4out, Yin |
| T4 | R5out, MUL, Zin |
| T5 | Zlowout, LOin |
| T6 | Zhighout, HIin |

You may need to use control signals to wait for the completion of the multiplication operation.

**3.6** Demonstrate that your Division circuitry works fine by simulating the Control Sequence for the ***div R4, R5*** instruction. The Control Sequence is similar to the *mul* instruction except for using the DIV control signal in T4 instead of the MUL signal. Be careful where the quotient and remainder are loaded inside the *Z* register, and change T5 and T6 control signals accordingly.

**3.7** Demonstrate that your Shift Right circuitry works correctly by simulating the Control Sequence for the ***shr R1, R2, R3*** instruction. The following Control Sequence is for a one-time shift right operation. Revise it accordingly for the count in R3.

**Control Sequence:**

| Step | Control Sequence |
|---|---|
| T0 | PCout, MARin, IncPC, Zin |
| T1 | Zlowout, PCin, Read, Mdatain[31..0], MDRin |
| T2 | MDRout, IRin |
| T3 | R2out, Yin |
| T4 | SHR, Zin |
| T5 | Zlowout, R1in |

**3.8** Demonstrate that your Shift Right Arithmetic circuitry works fine by simulating the Control Sequence for the ***shra R1, R2, R3*** instruction. The Control Sequence is the same as the *shr* instruction except for using the SHRA control signal in T4 instead of SHR.

**3.9** Demonstrate that your Shift Left circuitry works fine by simulating the Control Sequence for the ***shl R1, R2, R3*** instruction. The Control Sequence is the same as the *shr* instruction except for using the SHL control signal in T4 instead of SHR.

**3.10** Demonstrate that your Rotate Right circuitry works fine by simulating the Control Sequence for the ***ror R1, R2, R3*** instruction. The following Control Sequence is for a one-time rotate right operation. Revise it accordingly for the count in R3.

**Control Sequence:**

| Step | Control Sequence |
|---|---|
| T0 | PCout, MARin, IncPC, Zin |
| T1 | Zlowout, PCin, Read, Mdatain[31..0], MDRin |
| T2 | MDRout, IRin |
| T3 | R2out, Yin |
| T4 | ROR, Zin |
| T5 | Zlowout, R1in |

**3.11** Demonstrate that your Rotate left circuitry works correctly by simulating the Control Sequence for the ***rol R1, R2, R3*** instruction. The Control Sequence is the same as the *ror* instruction except for using the ROL control signal in T4 instead of ROR.

**3.12** Demonstrate that your Negate circuitry works correctly by simulating the Control Sequence for the ***neg R6, R7*** instruction.

**Control Sequence:**

| Step | Control Sequence |
|---|---|
| T0 | PCout, MARin, IncPC, Zin |
| T1 | Zlowout, PCin, Read, Mdatain[31..0], MDRin |
| T2 | MDRout, IRin |
| T3 | R7out, NEG, Zin |
| T4 | Zlowout, R6in |

**3.13** Demonstrate that the Not circuitry works correctly by simulating the Control Sequence for the ***not R6, R7*** instruction. The Control Sequence is the same as the *neg* instruction except for using the NOT control signal in T3 instead of NEG.

**4. Report:** Upload your Phase 1 Lab report (one per group) in PDF format to onQ by 11:59pm on the day of your demo. Your report should include:

- Your Verilog/VHDL code (and schematic, if any)
- Your testbenches (if they are similar, just include one testbench and discuss the differences for the other cases).
- Functional simulation runs for all the tests (Sections 3.1 to 3.13)