Department of Electrical and Computer Engineering
Queen's University

ELEC-374 Digital Systems Engineering
Laboratory Project

Winter 2017

Designing a Simple RISC Computer: Phase 1

Objectives: The purpose 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. The complete datapath is shown in Section 1. In this Phase, you will design the logic and simulate *add, sub, mul, div, and, or, shr, shl, ror, rol, neg*, and *not* instructions. Note that the *"Select and Encode"* logic, *"CON FF"* Logic, *"Input/Output"* ports, the "Memory Subsystem", branch and jump instructions as well as *addi, andi,* and *ori* instructions will be designed and simulated in Phase 2.

Design input can be done using Schematic Capture, all HDL, or a mixed schematic/HDL approach. You may use components such as gates, registers, buffers, multiplexers, and encoders available in the Quartus II Library, which can be configured using the Wizard facility. On top of this, in phase 1 you are also required to design two of the following four components: 1) 16x16 hierarchical two-level *Carry Look Ahead adder;* 2) 32x32 Regular *Booth multiplier; 3) 32x32 bit-pair* Booth multiplier; and 4) 8x8 array multiplier. If you design more than two of these components, it will be considered as a bonus for you. Note that in the two-level Carry Look Ahead Adder design, Carry Look Ahead Adder must be used in both levels. Note that the main objective is to get the minimum requirements done in time, then go after an advanced design for different parts of your CPU.

1. Preliminaries:

(a) 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 Notes).

As shown in Figure 1, 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. 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 *Program Counter*, *PC*, holds the address of the current instruction. The 32-bit *Instruction Register*, *IR*, holds the current instruction. *PC* should be incremented by 1, using *IncPC* control signal in ALU, to point to the next instruction. You may instead opt for a hardware incrementer outside ALU. In general, you are welcome to come up with your own ideas and design decisions during the entire project.

The *Arithmetic Logic Unit* (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 pre-saved in a 32-bit register, *Y*. The ALU supports the addition, subtraction, multiplication, division, shift right, shift left, rotate right, rotate left, logical AND, Logical OR, as well as negate and NOT operations. The control signals to the

ALU (generated by the Control Unit, shown in Figure 5) will choose the required operation. These control signals include *ADD, SUB, MUL, DIV, SHR, SHL, ROR, ROL, AND, OR, NEG,* and *NOT* control signals, among others. 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. 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.

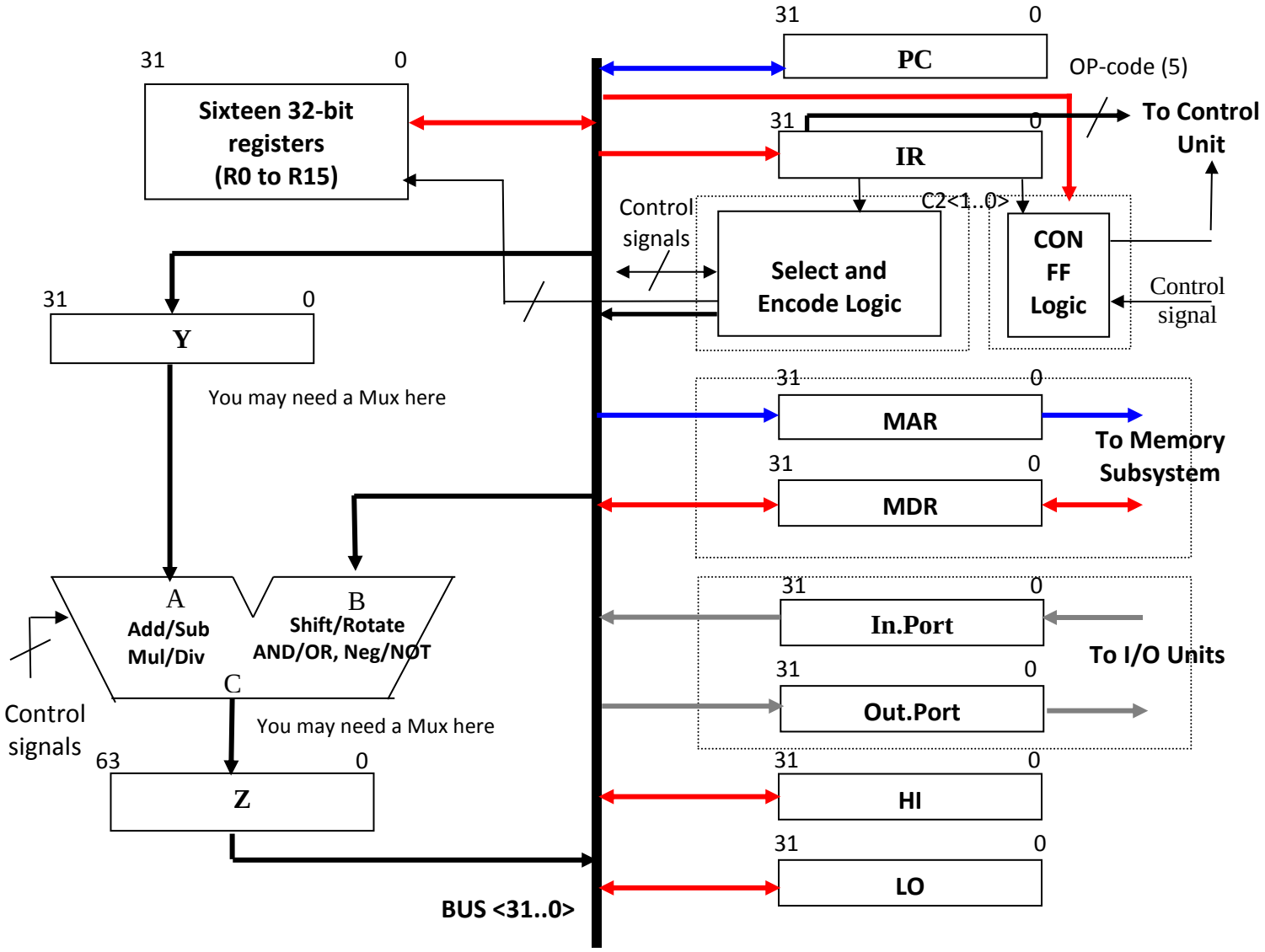

Figure 1: Simplified datapath

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* signals allow transfer of a register's contents 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 has been 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.

Note that the "Select and Encode logic", "CON FF Logic", "Input/Output ports", the "Memory Subsystem" 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 datapath description. 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 it in Phase 2 when we design the "Select and Encode Logic"). The input to the register, *BusMuxOut* is coming directly from the bus. The content 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 onto the register *R1*. As mentioned earlier, the *R1out* signal is the control signal that allows *R1* to drive the bus. The *Clear* signal is used to reset the registers to a known state.

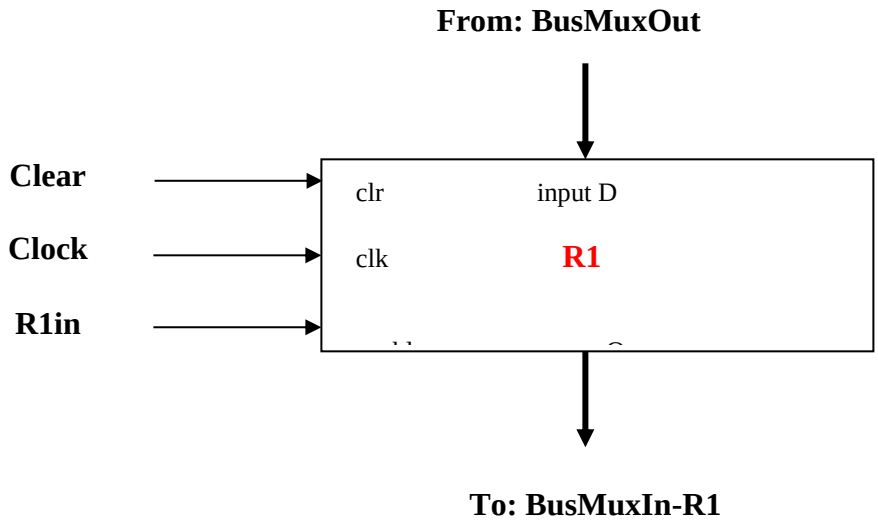

Figure 2: A typical register

Bus design: One of the important aspects of the datapath is its bus. The Bus may be implemented by tri-state buffers, or by a multiplexer and an encoder. 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 inputs coming from a 32-to-5 encoder. The idea is to choose only one of the registers *R0 to R15, HI, LO, PC, MDR, In.Port,* Z_{high} *or* Z_{low} as the source of the bus. The output of the BusMux is the Bus itself. 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 will just simulate these signals.

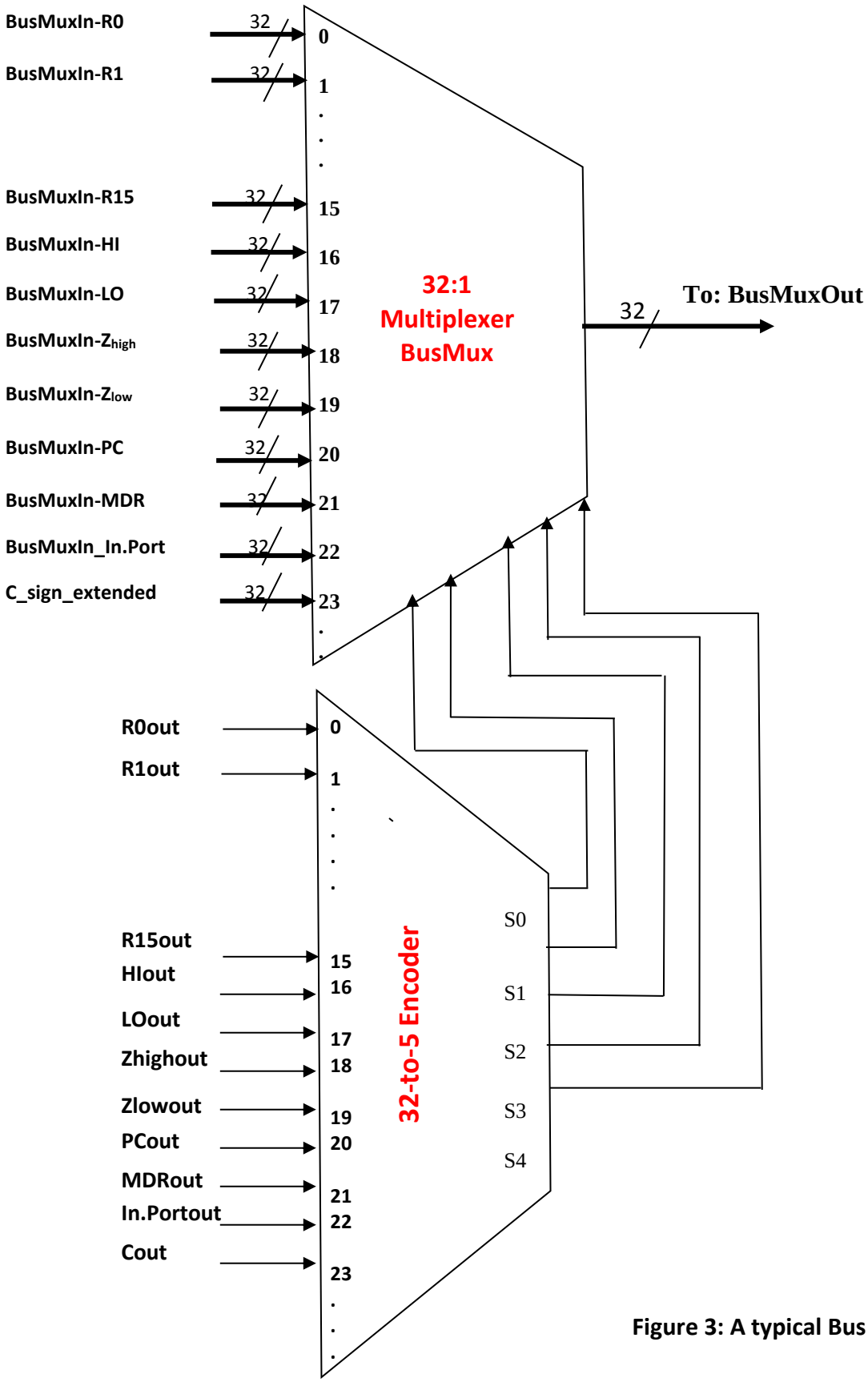

Figure 3: A typical Bus

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 sources. Figure 4 presents how *MDR* is connected to the memory bus, and to the internal bus. The inputs to the *MDR* comes form the memory unit (*Mdatain*) or from the Bus (*BusMuxOut*). Data is stored in the *MDR* using the synchronous *Clock* signal and the *MDRin* signal. The *MDR* contents can be written onto the memory, or drive the Bus.

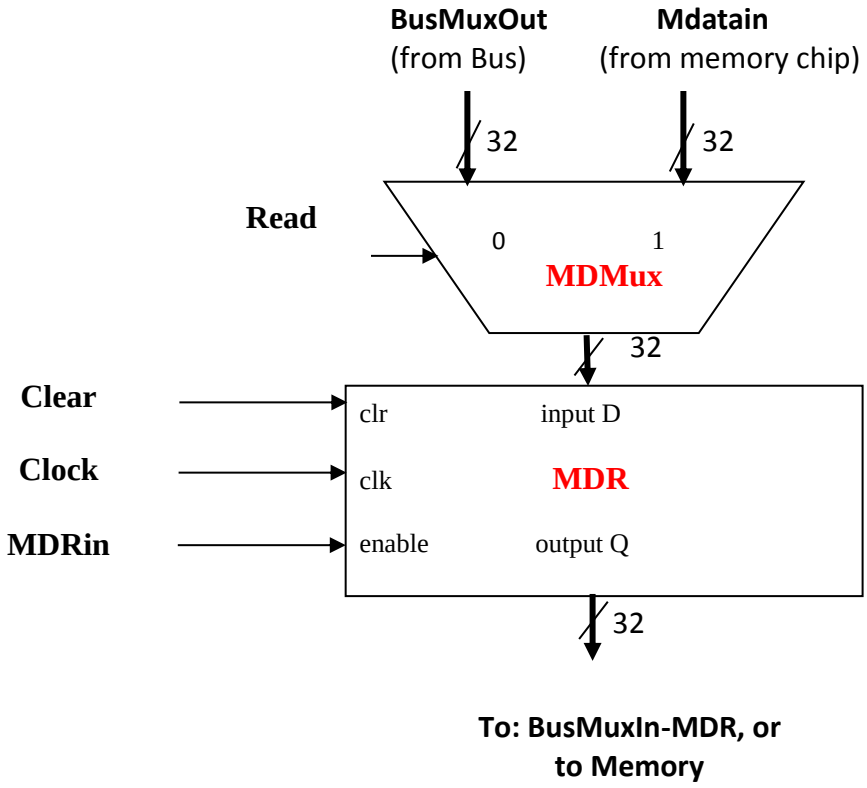

Figure 4: The MDR unit

(b) Control Unit: The Control unit is to be designed in Phase 3. However, a block diagram is provided here for a better understanding of the Datapath, as shown in Figure 5. 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 the instruction decoding in Phase 1 and Phase 2 of this project. Instruction decoding will be done in Phase3 with VHDL or Verilog. We will talk about the Control Unit in detail in Phase 3.

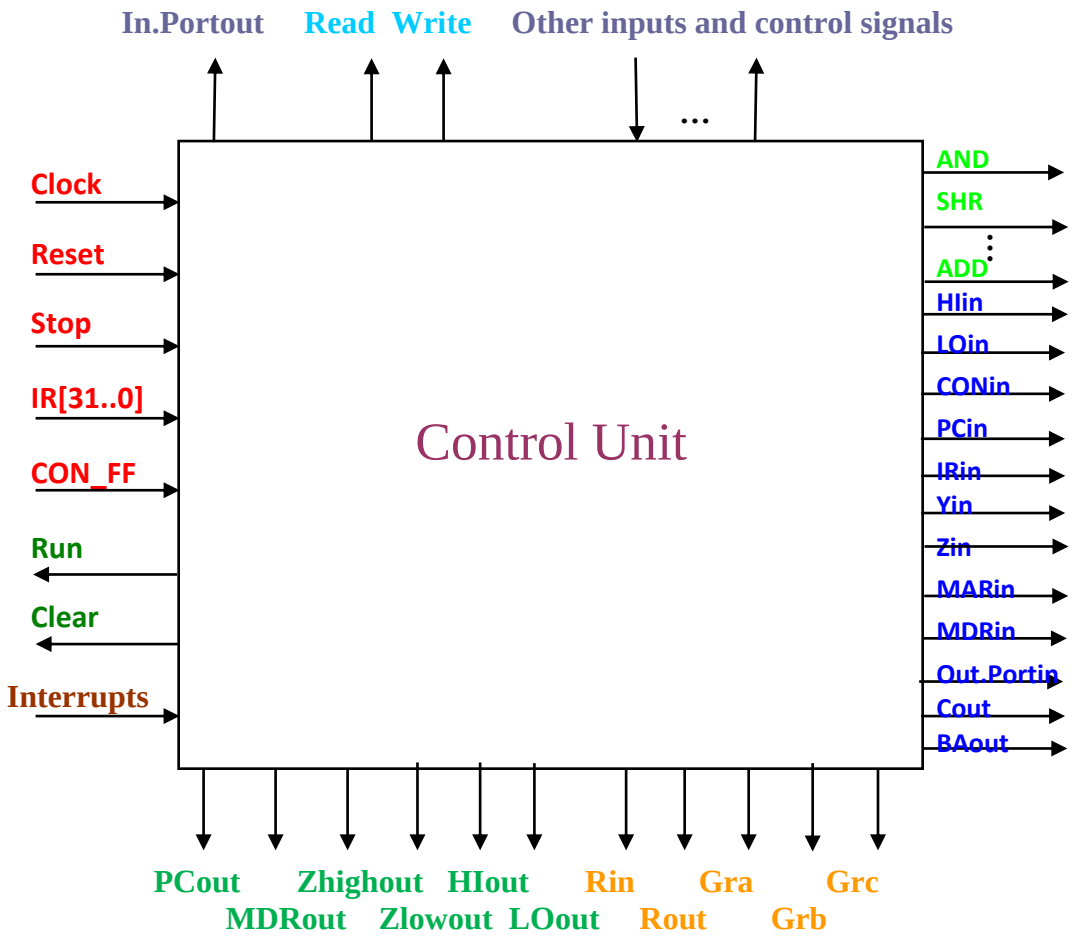

Figure 5: Block diagram of the Control Unit

2. Lab Procedure: Design the datapath shown in Figure 1 (except for the "Select and Encode Logic", "CON FF Logic", "Memory Subsystem", and "Input/Output ports" units) using Schematic Capture, HDL (VHDL or Verilog), or a mixed schematic/HDL in Quartus II design software. You are advised to follow the steps below when designing your datapath:

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 the registers that you designed in Step 1 to the Bus.
3. Design the MDR register (Figure 4) and connect it to the BUS. Leave the rest of the Memory Subsystem (connection to the memory module) for Phase 2.
4. Design your ALU. Start with simple ALU operations such as AND, OR, and NOT. Then, design the more involved operations such as ADD/SUB, MUL, and DIV circuitry. Finally, design the rest of the ALU operations. Using the following control sequences, test your design for *add, sub, mul, div, and, or, shr, shl, ror, rol, neg,* and *not* instructions.
    - You also need to design, two of the following components 8x8 Array Multiplier, 32x32 Booth algorithm with bit-pair recoding, 32x32 Regular Booth multiplier, and 16x16 2-level Carry Look Ahead Adder.

- During the term, you may design and implement any other advanced design techniques that you learned in class for the bonus marks.

For Phase 1, you require to generate testbench and demonstrate the simulation of at least two of the following components (8x8 Array Multiplier, 32x32 Booth algorithm with bit-pair recoding, 32x32 Regular Booth multiplier, and 16x16 2-level Carry Look Ahead Adder). However, it is optional to use these components in the next phases of your CPU design project.

In order to design and test the Datapath by Functional Simulation, the following control signals are required. In Phase 3, these control signals will be generated by the Control Unit.

Control Signals: **R0in, R0out; R1in, R1out; …, R15in, R15out; HIout; HIin; LOout; LOin; 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 for demo to TA in lab) and any others you may wish to observe as outputs for simulation.

2.a) In the lab, demonstrate that your Adder works correctly by simulating the Control Sequence for the ***add R1, R2, R3*** instruction (see table 4.7 on page 155 of the Lab Notes), 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

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 *add R1, R2, R3* instruction. Its pattern can be determined by referring to the specification for Mini SRC. In Phase 2, the opcode directly will come from the memory unit, but note that you may need to insert a control signal to make sure when your memory data becomes available.

Instruction decoding will be done in Phase 3.

As demonstrated in the Quartus II tutorial, to simulate your design using Modelsim, you will need to write a testbench program in VHDL or Verilog. Here is a sample testbench in VHDL for the add instruction which

you may need to verify and revise it for other instructions. You may also use the university program for your simulation.

```
-- adder_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;
-- Architecture of the testbench with the signal names
ARCHITECTURE datapath_tb_arch OF datapath_tb IS     -- Add any other signal you would like 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, ADD_tb, R1in_tb, R2in_tb, R3in_tb:       std_logic;
   SIGNAL      Mdatain_tb, BusMuxOut_tb:                                   std_logic_vector (31 downto 0);
   SIGNAL      Clock_tb:       Std_logic;

    TYPE       State IS (default, Reg_load1, Reg_load2, Reg_load3, T0, T1, T2, T3, T4, T5);
    SIGNAL     Present_state:  State := default;

     -- component instantiation of the datapath
 COMPONENT datapath
    PORT (
                PCout, Zlowout, MDRout, R2out, R3out:                out     std_logic;
                MARin, Zin, PCin, MDRin, IRin, Yin, R1in, R2in, R3in:  out   std_logic;
                IncPC, Read, ADD:                                    out     std_logic;
                Mdatain, BusMuxOut:                                  out     std_logic_vector (31 downto 0);
                Clock:                                               in      Std_logic;
 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,
        R1in           =>     R1in_tb,
        R2in           =>     R2in_tb,
```

```
        R3in            =>      R3in_tb,
        IncPC           =>      IncPC_tb,
        Read            =>      Read_tb,
        ADD             =>      ADD_tb,
        Mdatain         =>      Mdatain_tb,
        BusMuxOut       =>      BusMuxOut_tb,
        Clock           =>      Clock_tb);

--add test logic here
        Clock_process: PROCESS
        BEGIN
                Clock_tb <= '1', '0' after 10 ns;
                Wait for 20 ns;
        END PROCESS Clock_process
        PROCESS (Clock_tb)          -- finite state machine
        BEGIN
                IF (Clock_tb'EVENT AND Clock_tb = '1') THEN          -- if clock rising-edge
                        CASE Present_state IS
                                WHEN Default =>
                                        Present_state <= Reg_load1;
                                WHEN Reg_load1 =>
                                        Present_state <= Reg_load2;
                                WHEN Reg_load2 =>
                                        Present_state <= Reg_load3;
                                WHEN Reg_load3 =>
                                        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)         -- 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';
```

```
                    R1in_tb <= '0';   IncPC_tb <= '0';   Read_tb <= '0';   ADD_tb <= '0';


                WHEN Reg_load1 =>
                    Mdatain_tb <= x"00000012";
                    Read_tb <= '0', '1' after 10 ns, '0' after 10 ns;
                    MDRin_tb <= '0', '1' after 10 ns, '0' after 10 ns;
                    MDRout_tb <= '0', '1' after 10 ns, '0' after 10 ns;
                    R1in_tb <= '0', '1' after 10 ns;

                WHEN Reg_load2 =>
                    Mdatain_tb <= x"00000014" after 10 ns;
                    Read_tb <= '0', '1' after 10 ns, '0' after 10 ns;
                    MDRin_tb <= '0', '1' after 10 ns, '0' after 10 ns;
                    MDRout_tb <= '0', '1' after 10 ns, '0' after 10 ns;
                    R2in_tb <= '0', '1' after 10 ns;

                WHEN Reg_load3 =>
                    Mdatain_tb <= x"00000016" after 10 ns;
                    Read_tb <= '0', '1' after 10 ns, '0' after 10 ns;
                    MDRin_tb <= '0', '1' after 10 ns, '0' after 10 ns;
                    MDRout_tb <= '0', '1' after 10 ns, '0' after 10 ns;
                    R3in_tb <= '0', '1' after 10 ns'

                WHEN T0 =>
                    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[31..0] <= x"294c0000";                    -- opcode for add R1, R2, R3
                WHEN T2 =>
                    MDRout_tb <= '1';   IRin_tb <= '1'
                WHEN T3 =>
                    R2out_tb <= '1';   Yin_tb <= '1';
                WHEN T4 =>
                    R3out_tb <= '1';   ADD_tb <= '1';   Zin_tb <= '1';
                WHEN T5 =>
                    Zlowout_tb <= '1';   R1in_tb <= '1';
                WHEN OTHERS =>
            END CASE;
        END PROCESS;
END ARCHITECTURE datapath_tb_arch;
```

2.b) Demonstrate that your Subtract circuitry works fine by simulating the Control Sequence for the ***sub R0, R4, R5*** instruction. The Control Sequence is the same as the one used for *add* instruction except for using the SUB control signal in T4 instead of the ADD signal.

2.c) Demonstrate that your Multiplication circuitry works correctly by simulating the Control Sequence for the ***mul R5, R7*** instruction.

Control Sequence:

Step	Control Sequence
T0	PCout, MARin, IncPC, Zin
T1	Zlowout, PCin, Read, Mdatain[31..0], MDRin
T2	MDRout, IRin
T3	R5out, Yin
T4	R7out, MUL, Zin
T5	Zlowout, LOin
T6	Zhighout, HIin

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

2.d) Demonstrate that your Division circuitry works fine by simulating the Control Sequence for the ***div R3, R1*** 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.

2.e) Demonstrate your logical AND works correctly by simulating the Control Sequence for the logical ***and R2, R3, R6*** instruction, as follows:

Control Sequence:

Step	Control Sequence
T0	PCout, MARin, IncPC, Zin
T1	Zlowout, PCin, Read, Mdatain[31..0], MDRin
T2	MDRout, IRin
T3	R3out, Yin
T4	R6out, AND, Zin
T5	Zlowout, R2in

2.f) Demonstrate that your logical OR design works fine by simulating the Control Sequence for the ***or R0, R1, R7*** instruction. The Control Sequence is the same as the *and* instruction except for using the OR control signal in T4 instead of the AND signal.

2.g) Demonstrate that your Shift right circuitry works correctly by simulating the Control Sequence for the ***shr R2, R1, 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	R1out, Yin
T4	SHR, Zin
T5	Zlowout, R2in

2.h) Demonstrate that your Shift left circuitry works fine by simulating the Control Sequence for the ***shl R3, R0, R5*** instruction. The Control Sequence is the same as the *shr* instruction except for using the SHL control signal in T4 instead of SHR.

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

Control Sequence:

Step	Control Sequence
T0	PCout, MARin, IncPC, Zin
T1	Zlowout, PCin, Read, Mdatain[31..0], MDRin
T2	MDRout, IRin
T3	R1out, Yin
T4	ROR, Zin
T5	Zlowout, R1in

2.j) Demonstrate that your Rotate left circuitry works correctly by simulating the Control Sequence for the ***rol R0, R0, R4*** instruction. The Control Sequence is the similar to the *ror* instruction except for using the ROL control signal in T4 instead of ROR.

2.k) Demonstrate that your negate circuitry works correctly by simulating the Control Sequence for the ***neg R1, R2*** instruction.

Control Sequence:

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

2.l) Demonstrate that your *not* circuitry works correctly by simulating the Control Sequence for the ***not R1, R2*** instruction. The Control Sequence is the same as the *neg* instruction except for using the NOT control signal in T3 instead of NEG.

3. Report:

The phase 1 report (one per group) include:

Printouts of your Schematic, HDL codes
Printout of your testbenches (if used)
Functional simulation runs for all the tests