Department of Electrical and Computer Engineering
Queen's University

# ELEC 374 Digital Systems Engineering
# Laboratory Project

Winter 2024

## Designing a Simple RISC Computer: Phase 3

---

## 1. Objectives

The objective of this project is to design, simulate, implement, and verify a simple RISC Computer (Mini SRC). So far, you have designed and functionally simulated the Datapath portion of the Mini SRC (except for *nop* and *halt* instructions that you will test in this phase). Phase 3 of this project consists of adding and testing the Control Unit in Mini SRC. You are to design the Control Unit in Verilog or VHDL. Testing will be done by Functional Simulation.

## 2. Preliminaries

### 2.1 Control Unit

A block diagram of the Control Unit for Mini SRC is shown in Figure 1. The Control Unit is at the heart of the processor. It accepts as input those signals that are needed to operate the processor and provides as output all control signals necessary to execute the instructions. The outputs from the Control Unit are the control signals that we have been using in previous phases to generate control sequences for the instructions of the Mini SRC.

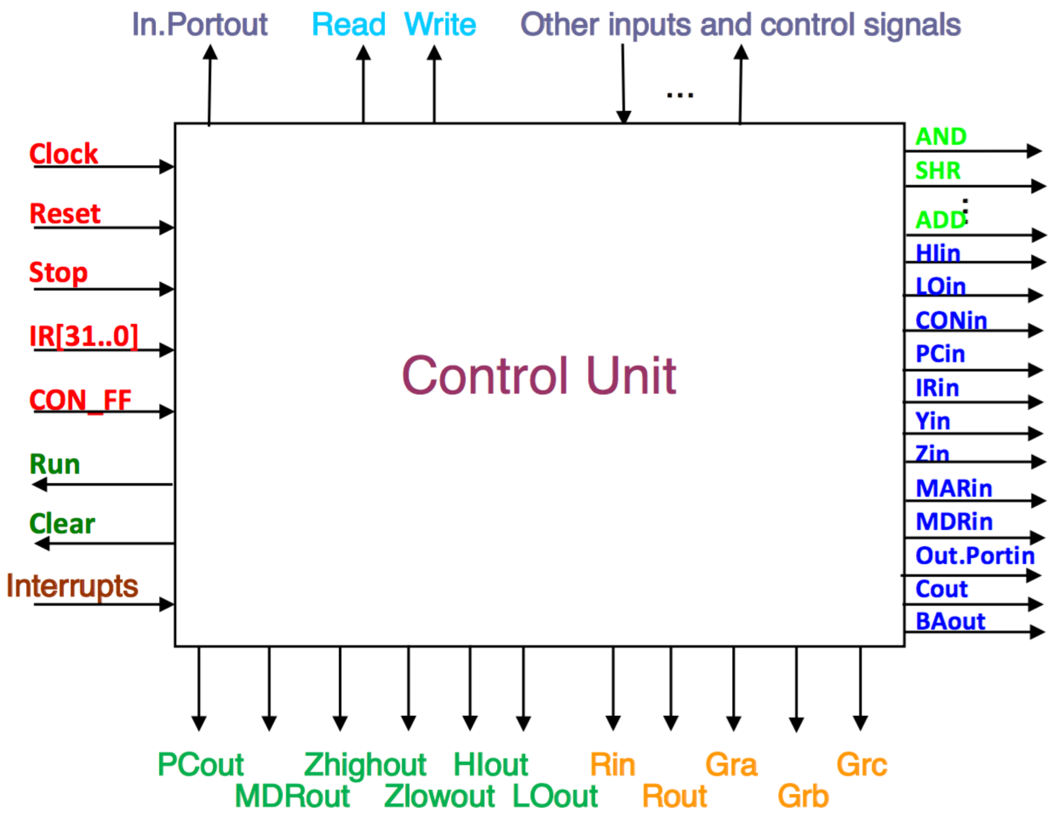

**Figure 1: Block diagram of the Control Unit**

The Control Unit generates the control signals from four principal sources:

(a) The op-code fields of the IR
(b) Signals from the Datapath such as CON FF, and the condition code registers (if any)
(c) Control step information such as signals T0, T1, …
(d) External inputs such as Stop, Reset, Done (if memory is slow), and other signals such as interrupts (if any)

The external *Reset* input should get the Mini SRC to an initial state, where all registers including PC in the Datapath are set to 0, the *Run* indicator is set to 1, and the processor starts at Step T0. As the clock continues to run, instructions should be fetched and executed one after the other until a *halt* instruction is encountered, at which point the control stepping process should be halted and the *Run* indicator is set to 0. Note that, the external *Stop* input signal works the same way as the *halt* instruction.
During T0, T1, and T2, the control signals that are asserted to implement the "instruction fetch" sequence are independent of the bits in the Instruction Register. For instance, in Step T0, the control signals PCout, MARin, IncPC and Zin are set to 1. In Step T1, the control signals Zlowout, PCin, Read, and MDRin are set to 1. In Step T2, the control signals MDRout and IRin are set to 1. However, from Step T3 onward, until the current instruction is completed, the control signals that are asserted are a function of both Step Ti and the op-code bits in the IR register.

In the following, you will see two different methods to design your Control Unit. There is a trade-off between the two methods. Method 1 is clearly the easier method, but it may generate more hardware. You are, of course, welcome to come up with your own design.

**Method 1:** It is possible to write the Verilog/VHDL code without worrying about the combinational logic expressions for each control signal. Therefore, the code will come clean, and the instructions will be executed in the most efficient manner. However, this approach may generate more hardware. The following sample Verilog/VHDL code is provided as a starting point for this method, which you may need to verify and revise for your Control Unit:

```
// this is the Verilog sample code for Method 1 for the Control Unit
`timescale 1ns/10ps
module control_unit (
   output reg     Gra, Grb, Grc, Rin, …, Rout,           // define the inputs and outputs to your Control Unit here
                  Yin, Zin, PCout, IncPC, …, MARin,
                  Read, Write, …, Clear,
                  ADD, AND, …, SHR,
   input          [31:0] IR,
   input          Clock, Reset, Stop, …, Con_FF);
   parameter      reset_state = 4'b0000, fetch0 = 4'b0001, fetch1 = 4'b0010, fetch2 = 4'b0011,
                  add3 = 4'b0100, add4 = 4'b0101, add5 = 4'b0110, …;
   reg            [3:0] present_state = reset_state;   // adjust the bit pattern based on the number of states

always @(posedge Clock, posedge Reset)               // finite state machine; if clock or reset rising-edge
  begin
    if (Reset == 1'b1) present_state =  reset_state;
    else case (present_state)
         reset_state:     present_state = fetch0;
         fetch0:          present_state = fetch1;
         fetch1:          present_state = fetch2;
         fetch2:          begin
```

```
                                        case (IR[31:27])       // inst. decoding based on the opcode to set the next state
                                            5'b00011:   present_state = add3;    // this is the add instruction
                                              ⋮
                                        endcase
                                end
            add3:               present_state = add4;
            add4:               present_state = add5;
             ⋮
          endcase
     end

always @(present_state)            // do the job for each state
 begin
     case (present_state)             // assert the required signals in each state
            reset_state: begin
                    Gra <= 0;   Grb <= 0;   Grc <= 0;  Yin <= 0;               // initialize the signals
                     ⋮
            end
            fetch0: begin
                    PCout <= 1;        // see if you need to de-assert these signals
                    MARin <= 1;
                    IncPC <= 1;
                    Zin <= 1;
            end
            add3: begin
                    Grb <= 1;  Rout <= 1;
                    Yin <= 1;
            end
             ⋮
      endcase
 end
endmodule

-----------------------------------------------------
-- this is the VHDL sample code for Method 1 for the Control Unit
library ieee;
use ieee.std_logic_1164.all;

entity control_unit is               -- define the inputs and outputs to your Control Unit here
            port (Clock, Reset, Stop, …, CON_FF:              in       std_logic;
                  IR:                                        in       std_logic _vector(31 downto 0);
                  Gra, Grb, Grc, Rin, …, Rout:               out      std_logic;
                  Yin, Zin, PCout, IncPC, …, MARin:          out      std_logic;
                  Read, Write, …, Clear:                     out      std_logic;
                  ADD, AND, …, SHR:                          out      std_logic);
end control_unit;

architecture behavior of control_unit is
type state is (reset_state, fetch0, fetch1, fetch2, add3, add4, add5, …);
signal present_state:     state;
begin
            process (Clock, Reset)                  -- finite state machine
```

```
                        when others =>
                    end case;
                end if;
        end process;

        process (IR)
        begin
                ADD_s <= '0'; AND_s <= '0'; …
                case IR(31 downto 27)  is     -- inst. decoding based on the opcode
                    when "00011" =>
                            ADD_s <= '1';    -- this is the add instruction
                    when "01010" =>
                            AND_s <= '1';    -- this is the and instruction
                     ⋮
                    when others =>
                end case;
        end process;

        process (Clock, T0, T1, …)
        begin
                ADD <= ADD_s AND T4;              -- control signal assignment
                Zlowout <= T1 OR (T5 AND (AND_s OR OR_s OR ADD_s OR SUB_s OR …)) OR …;
                ⋮
        end process;
end behavior;
```

## 3. Procedure

**3.1)** Use one of the above methods, or come up with your own design style, and write your Verilog/VHDL code to implement the Control Unit and add to your Datapath.

**3.2)** Run a functional simulation of the following program on Mini SRC and demonstrate it to one of the TAs. This program is provided to test the control unit and instructions in Mini SRC, except for brnz/brzr Branch and input/output instructions that will be included in the test code for Phase 4.

Encode your program in the memory with the starting address zero. Initialize registers R0 - R15 and the PC to 0 with the *Reset* input signal. Initialize memory locations 0x47 and 0x8E with the 32-bit hexadecimal values 0x94 and 0x34, respectively.

Minimum outputs are IR, PC, MDR, MAR, R0 – R15, HI, and LO. Add any other signals you would like to observe to convince yourself and the TA that your design works fine.

```
Address      ORG    0
   0         ldi    R2, 0x69        ; R2 = 0x69
             ldi    R2, 2(R2)       ; R2 = 0x6B
             ld     R1, 0x47        ; R1 = (0x47) = 0x94
             ldi    R1, 1(R1)       ; R1 = 0x95
             ld     R0, -7(R1)      ; R0 = (0x8E) = 0x34
             ldi    R3, 3           ; R3 = 3
             ldi    R2, 0x43        ; R2 = 0x43
```

```
            brmi   R2, 3          ; continue with the next instruction (will not branch)
            ldi    R2, 6(R2)      ; R2 = 0x49
            ld     R7, -2(R2)     ; R7 = (0x49 - 2) = 0x94
            nop
            brpl   R7, 2          ; continue with the instruction at "target" (will branch)
            ldi    R5, 4(R2)      ; this instruction will not execute
            ldi    R4, -3(R5)     ; this instruction will not execute
target:     add    R2, R2, R3     ; R2 = 0x4C
            addi   R7, R7, 3      ; R7 = 0x97
            neg    R7, R7         ; R7 = 0xFFFFFF69
            not    R7, R7         ; R7 = 0x96
            andi   R7, R7, 0xF    ; R7 = 6
            ror    R1, R0, R3     ; R1 = 0x80000006
            ori    R7, R1, 9      ; R7 = 0x8000000F
            shra   R1, R7, R3     ; R1 = 0xF0000001
            shr    R2, R2, R3     ; R2 = 9
            st     0x8E, R2       ; (0x8E) = 9    new value in memory with address 0x8E
            rol    R2, R0, R3     ; R2 = 0x1A0
            or     R4, R3, R0     ; R4 = 0x37
            and    R1, R2, R0     ; R1 = 0x20
            st     0x27(R1), R4   ; (0x47) = 0x37    new value in memory with address 0x47
            sub    R0, R2, R4     ; R0 = 0x169
            shl    R1, R2, R3     ; R1 = 0xD00
            ldi    R4, 6          ; R4 = 6
            ldi    R5, 0x1B       ; R5 = 0x1B
            mul    R5, R4         ; HI = 0; LO = 0xA2
            mfhi   R7             ; R7 = 0
            mflo   R6             ; R6 = 0xA2
            div    R5, R4         ; HI = 3, LO = 4
            ldi    R10, 1(R4)     ; R10 = 7          setting up argument registers
            ldi    R11, -2(R5)    ; R11 = 0x19           R8, R9, R10, and R11
            ldi    R12, 0(R6)     ; R12 = 0xA2
            ldi    R13, 3(R7)     ; R13 = 3
            jal    R12            ; address of subroutine subA in R12 - return address in R15
0x29        halt                  ; upon return, the program halts

subA:       ORG    0xA2           ; procedure subA
0xA2        add    R9, R10, R12   ; R8 and R9 are return value registers
            sub    R8, R11, R13   ; R9 = 0xA9, R8 = 0x16
            sub    R9, R9, R8     ; R9 = 0x93
            jr     R15            ; return from procedure
```

## 4. Report

Upload your Phase 3 report (one per group) in PDF format to onQ by 11:59pm on the day of your Phase 3 demo. Phase 3 report consists of:

- Your Verilog/VHDL code (and schematic, if any)
- Functional simulation run of the program
- Printouts of the contents of memory before and after the program run