Department of Electrical and Computer Engineering
Queen's University

ELEC 374 Digital Systems Engineering
Laboratory Project

Winter 2023

Designing a Simple RISC Computer: Phase 3

---

1. Objectives

The purpose 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 VHDL or Verilog. 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 the previous phases to generate control sequences for the instructions of the Mini SRC.

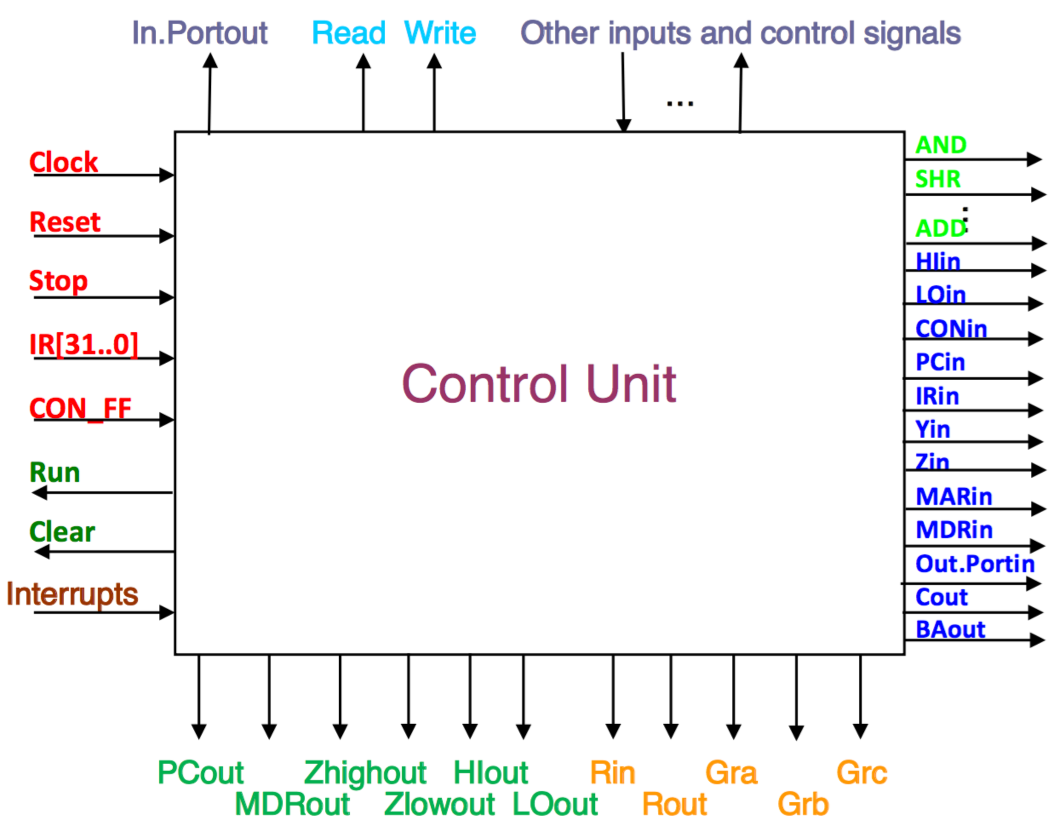

Figure 1: Block diagram of the Control Unit

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. Of course, you are free to come up with your own design, if you wish.

Method 1: It is possible to write the VHDL/Verilog 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, it may generate more hardware. The following sample VHDL and Verilog code are provided as a starting point for this method, which you may need to verify and revise for your Control Unit:

```
-- 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
        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
        begin
                if (Reset = '1') then           -- reset the processor
```

```
                present_state <= reset_state;
        elsif (Clock'event and Clock = '1')) then        -- if clock rising-edge
                case present_state is
                        when reset_state =>
                                present_state <= fetch0;
                        when fetch0 =>
                                present_state <= fetch1;
                        when fetch1 =>
                                present_state <= fetch2;
                        when fetch2 =>      -- instruction decoding based on the opcode to set the next state
                                case IR(31 downto 27) is
                                        when "00011" =>        -- this is the add instruction
                                                present_state <= add3;
                                          ⋮
                                        when others =>
                                end case;
                        when add3 =>
                                present_state <= add4;
                        when add4 =>
                                present_state <= add5;
                          ⋮
                        when others =>
                end case;
        end if;
    end process;

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

Method 2: In this approach, you will need to derive all control signal setting conditions for all instructions. For this, you must examine all of the control sequences of the machine. The logic for each control signal is generated by going through the control sequences looking for every occurrence of that control signal and writing the Boolean equation for the signal. For example, the Zlowout signal occurs at T1 in all instructions, at T5 in *and*, *or, add*, *sub*, *mul, div, shr,* shra, *shl, ror, rol, ld,* and *ldi* instructions, and in some T states for other instructions. Therefore,

Zlowout = T1 + T5 . (AND + OR + ADD + SUB + MUL + DIV + SHR + SHRA + SHL + ROR + ROL + …) + …

The problem with this approach is that the logic for each control signal may change with the addition of new instructions, therefore this approach is provided here merely for the sake of completeness. You are advised to use Method 1, especially if you intend to extend the scope of the project by adding new instructions, etc.

The following sample VHDL and Verilog code are provided as a starting point for Method 2, which you may need to verify and revise for your Control Unit:

```
-- this is the VHDL sample code for Method 2 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
        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, Zlowout, …, 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
        signal T0, T1, T2, T3, T4, T5, …, :          std_logic;
        signal ADD_s, SUB_s, AND_s, OR_s, …, :  std_logic;
        type state is (reset_state, S0, S1, …, );
        signal present_state:     state;
begin
        process (Clock, Reset, …)       -- finite state machine
        begin
                if (Reset = '1') then           -- reset the processor
                        present_state <= reset_state;
                elsif (Clock'event and Clock = '1')) then          -- if clock rising-edge
                        T0 <= '0'; T1 <= '0'; T2 <= '0'; T3 <= '0'; T4 <= '0'; T5 <= '0';  …
                        case present_state is
                                when reset_state =>
                                        present_state <= S0;
                                        T0 <= '1';
                                when S0 =>
                                        present_state <= S1;
                                        T1 <= '1';
                                   ⋮
                                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 "00101" =>
                    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;
----------------------------------------------------

// this is the Verilog sample code for Method 2 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
                  Yin, Zin, PCout, IncPC, Zlowout, ..., MARin,
                  Read, Write, ..., Clear,
                  ADD, AND, ..., SHR,
   input          [31:0] IR,
   input          Clock, Reset, Stop, ..., Con_FF);

parameter      reset_state = 4'b0000, S0 = 4'b0001, S1 = 4'b0010, ...;
   reg            [3:0] present_state = reset_state;        // adjust the bit pattern based on the number of states
   reg            T0, T1, T2, T3, T4, T5, ..., ;
   reg            ADD_s, SUB_s, AND_s, OR_s, ..., :

always @(posedge Clock, posedge Reset, ...)          // finite state machine; if clock or reset rising-edge
  begin
    if (Reset == 1'b1) present_state =  reset_state;     // reset the processor
    else begin
        T0 <= 0; T1 <= 0; T2 <= 0; T3 <= 0; T4 <= 0; T5 <= 0;
        case (present_state)
           reset_state:  begin
                present_state = S0;
                T0 <= 1;
           end
```

```
            S0:  begin
                    present_state = S1;
                    T1 <= 1;
                 end
                   ⋮
          endcase
    end
  end

always @(IR)
 begin
   ADD_s <= 0; AND_s <= 0; …
  case (IR[31:27])                          // inst. decoding based on the opcode
         5'b00011:      ADD_s <= 1;      // this is the add instruction
         5'b00101:      AND_s <= 1;      // this is the and instruction
           ⋮
  endcase
 end

always @(Clock, T0, T1, …)
 begin
   ADD <= ADD_s && T4;                     // control signal assignment
   Zlowout <= T1 || (T5 && (AND_s || OR_s || ADD_s || SUB_s || …)) || …;
           ⋮
 end
endmodule
```

3. Procedure

3.1) Use one of the above Methods (or come up with your own design style) and write your VHDL/Verilog code to implement the Control Unit. Add the Control Unit 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 for the sake of testing the control unit and the 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 the memory locations $68 and $52 with the 32-bit hexadecimal values $55 and $26, 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 that your design works fine.

```
ORG     0
ldi     R1, 2           ; R1 = 2
ldi     R0, 0(R1)       ; R0 = 2
ld      R2, $68         ; R2 = ($68) = $55
ldi     R2, -4(R2)      ; R2 = $51
ld      R1, 1(R2)       ; R1 = ($52) = $26
```

```
            ldi     R3, $69         ; R3 = $69
            brmi    R3, 4           ; continue with the next instruction (will not branch)
            ldi     R3, 2(R3)       ; R3 = $6B
            ld      R7, -3(R3)      ; R7 = ($6B - 3) = $55
            nop
            brpl    R7, 2           ; continue with the instruction at "target" (will branch)
            ldi     R2, 5(R0)       ; this instruction will not execute
            ldi     R3, 2(R1)       ; this instruction will not execute
target:     add     R3, R2, R3      ; R3 = $BC
            addi    R7, R7, 2       ; R7 = $57
            neg     R7, R7          ; R7 = $FFFFFFA9
            not     R7, R7          ; R7 = $56
            andi    R7, R7, $0F     ; R7 = 6
            ror     R1, R1, R0      ; R1 = $80000009
            ori     R7, R1, $1C     ; R7 = $8000001D
            shra    R7, R7, R0      ; R7 = $E0000007
            shr     R2, R3, R0      ; R2 = $2F
            st      $52, R2         ; ($52) = $2F   new value in memory with address $52
            rol     R2, R2, R0      ; R2 = $BC
            or      R2, R3, R0      ; R2 = $BE
            and     R1, R2, R1      ; R1 = $8
            st      $60(R1), R3     ; ($68) = $BC   new value in memory with address $68
            sub     R3, R2, R3      ; R3 = 2
            shl     R1, R2, R0      ; R1 = $2F8
            ldi     R4, 6           ; R4 = 6
            ldi     R5, $32         ; R5 = $32
            mul     R5, R4          ; HI = 0; LO = $12C
            mfhi    R7              ; R7 = 0
            mflo    R6              ; R6 = $12C
            div     R5, R4          ; HI = 2, LO = 8
            ldi     R8, -1(R4)      ; R8 = 5          setting up argument registers
            ldi     R9, -19(R5)     ; R9 = $1F            R8, R9, R10, and R11
            ldi     R10, 0(R6)      ; R10 = $12C
            ldi     R11, 0(R7)      ; R11 = 0
            jal     R10             ; address of subroutine subA in R10 - return address in R15
            halt                    ; upon return, the program halts

subA:       ORG     $12C            ; procedure subA
            add     R13, R8, R10    ; R12 and R13 are return value registers
            sub     R12, R9, R11    ; R13 = $131, R12 = $1F
            sub     R13, R13, R12   ; R13 = $112
            jr      R15             ; return from procedure
```

4. Report

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

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