It’s Alive!

Experiment #7

Froylan Aguirre

Hasham Ali

CPE 233 07

Objective

The modules of the RAT MCU are connected to form the RAT MCU which will interface with a Basys 3 board using the RAT wrapper. The interconnected modules will be controlled by the control unit, implemented here as a FSM. The instructions implemented in the control unit for now are IN, MOV (reg-immed), SUB (reg-reg), OUT, and BRN.

Procedure

The control unit was implemented first, then the modules interconnected, and finally the RAT MCU was placed in the RAT wrapper.

For now, the instructions implemented in the control unit are IN, MOV (reg-immed), SUB (reg-reg), OUT, and BRN. Before modeling, a table of instructions and their relevant values was created. In the control unit module, these signals are assigned the same values as in the table. This table is shown in Table.1.

Instruction	**Relevant Module Operations**	
	Module Operation Name	**Value**
BRN	PC_LD	1
	PC_MUX_SEL	00
IN (reg-immed)	RF_WR_SEL	11
	RF_WR	1
OUT (reg-immed)	RF_OE	1
	IO_STRB	1
SUB (reg-reg)	ALU_SEL	0010
	RF_OE	1
	ALU_OPY_SEL	0
	FLG_C_LD	1
	FLG_Z_LD	1
	RF_WR_SEL	00
	RF_WR	1
MOV (reg-immed)	ALU_SEL	1110
	ALU_OPY_SEL	1
	RF_WR	1
	RF_WR_SEL	00

Table.1. Instructions and their Control Unit signals.

Secondly, the RAT modules were interconnected according to the RAT MCU Architecture V3.00. The prog_rom module was created from the assembly of the program shown in Fig.1. However, the shadow flags and the interrupt hardware have not been fully implemented yet.

```
;------------------------------------------------------------------
;- I/O Port Constants
;-------------------------------------------------------------------
;------------------------------------------------------------------
.EQU SWITCH_PORT = 0x20 ; port for switch input
.EQU LED_PORT = 0x40 ; port for LED output
;------------------------------------------------------------------
.CSEG
.ORG 0x01
main: IN   r10,  SWITCH_PORT
      MOV  r11,  0x01
      SUB  r10,  r11
      OUT  r10,  LED_PORT
      BRN  main
```

Fig.1. RAT assembly program stored in prog_rom.

Lastly, the “complete” RAT MCU was placed in the RAT wrapper as an interface with the Basys 3 board. The RAT wrapper will allow the RAT MCU to properly handle I/O from the Basys 3 board such as I/O from the switches and LEDs. The RAT wrapper accomplishes this by selecting the proper I/O to accept or display based on the PORT_ID output of the RAT MCU. Before the RAT MCU was placed in the RAT wrapper, it was simulated in Vivado for correct functionality.

CONTROL UNIT VHDL CODE

```
----------------------------------------------------------------------------------
-- Froylan Aguirre & Hasham Ali
-- RAT MCU CONTROL UNIT for EXPERIMENT 7
----------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;


Entity CONTROL_UNIT is
    Port ( CLK           : in   STD_LOGIC;
           C             : in   STD_LOGIC;
           Z             : in   STD_LOGIC;
           INT           : in   STD_LOGIC;
           RESET         : in   STD_LOGIC;
           OPCODE_HI_5   : in   STD_LOGIC_VECTOR (4 downto 0);
           OPCODE_LO_2   : in   STD_LOGIC_VECTOR (1 downto 0);

           PC_LD         : out  STD_LOGIC;
           PC_INC        : out  STD_LOGIC;
           PC_MUX_SEL    : out  STD_LOGIC_VECTOR (1 downto 0);
           PC_OE         : out  STD_LOGIC;

           SP_LD         : out  STD_LOGIC;
           SP_INCR       : out  STD_LOGIC;
           SP_DECR       : out  STD_LOGIC;

           RF_WR         : out  STD_LOGIC;
           RF_WR_SEL     : out  STD_LOGIC_VECTOR (1 downto 0);
           RF_OE         : out  STD_LOGIC;

           ALU_OPY_SEL   : out  STD_LOGIC;
           ALU_SEL       : out  STD_LOGIC_VECTOR (3 downto 0);

           SCR_WR        : out  STD_LOGIC;
           SCR_ADDR_SEL  : out  STD_LOGIC_VECTOR (1 downto 0);
           SCR_OE        : out  STD_LOGIC;

           FLG_C_LD      : out  STD_LOGIC;
           FLG_C_SET     : out  STD_LOGIC;
           FLG_C_CLR     : out  STD_LOGIC;
           FLG_SHAD_LD   : out  STD_LOGIC;
           FLG_LD_SEL    : out  STD_LOGIC;
           FLG_Z_LD      : out  STD_LOGIC;

           I_SET    : out  STD_LOGIC;
           I_CLR    : out  STD_LOGIC;

           RST           : out  STD_LOGIC;
           IO_STRB       : out  STD_LOGIC);
end;

architecture Behavioral of CONTROL_UNIT is

   type state_type is (ST_init, ST_fet, ST_exec);
   signal PS,NS : state_type;
   signal sig_OPCODE_7: std_logic_vector (6 downto 0);
```

```
begin

   -- concatenate the all opcodes into a 7-bit complete opcode for
       -- easy instruction decoding.
   sig_OPCODE_7 <= OPCODE_HI_5 & OPCODE_LO_2;

   sync_p: process (CLK, NS, RESET)
   begin
      if (RESET = '1') then
          PS <= ST_init;
       elsif (rising_edge(CLK)) then
         PS <= NS;
       end if;
   end process sync_p;


   comb_p: process (sig_OPCODE_7, PS, NS, C, Z)
   begin

       -- schedule everything to known values -----------------------
      PC_LD      <= '0';
      PC_MUX_SEL <= "00";
      PC_OE      <= '0';
      PC_INC     <= '0';

      SP_LD   <= '0';
      SP_INCR <= '0';
      SP_DECR <= '0';

         RF_WR     <= '0';
      RF_WR_SEL <= "00";
      RF_OE     <= '0';

      ALU_OPY_SEL <= '0';
      ALU_SEL     <= "0000";

      SCR_WR       <= '0';
      SCR_OE       <= '0';
      SCR_ADDR_SEL <= "00";

      FLG_C_SET  <= '0';   FLG_C_CLR   <= '0';
      FLG_C_LD   <= '0';   FLG_Z_LD    <= '0';
      FLG_LD_SEL <= '0';   FLG_SHAD_LD <= '0';

         I_SET <= '0';
      I_CLR <= '0';

      IO_STRB <= '0';
      RST     <= '0';

   case PS is

      -- STATE: the init cycle ------------------------------------
       -- Initialize all control outputs to non-active states and
      --   Reset the PC and SP to all zeros.
       when ST_init =>
         RST <= '1';
          NS <= ST_fet;
      -- STATE: the fetch cycle -----------------------------------
      when ST_fet =>
         NS <= ST_exec;
         PC_INC <= '1';  -- increment PC


      -- STATE: the execute cycle ---------------------------------
      when ST_exec =>
           NS <= ST_fet;
           PC_INC <= '0';  -- don't increment PC

            case sig_OPCODE_7 is

               -- BRN -------------------
             when "0010000" =>
               PC_LD <= '1';
               PC_MUX_SEL <= "00";

               -- SUB reg-reg  --------
             when "0000110" =>
               ALU_SEL <= "0010";
               RF_OE <= '1';
               ALU_OPY_SEL <= '0';
               FLG_C_LD <= '1';
               FLG_Z_LD <= '1';
               RF_WR_SEL <= "00";
               RF_WR <= '1';


               -- IN reg-immed  ------
             when "1100100" | "1100101" | "1100110" | "1100111" =>
               RF_WR_SEL <= "11";
               RF_WR <= '1';

               -- OUT reg-immed  ------
             when "1101000" | "1101001" | "1101010" | "1101011" =>
               RF_OE <= '1';
               IO_STRB <= '1';

               -- MOV reg-immed  ------
             when "1101100" | "1101101" | "1101110" | "1101111" =>
               ALU_SEL <= "1110";
               ALU_OPY_SEL <= '0';
               RF_WR <= '1';
               RF_WR_SEL <= "00";

             when others =>            -- for inner case
                 NS <= ST_fet;

            end case; -- inner execute case statement

          when others =>    -- for outer case
               NS <= ST_fet;

           end case;  -- outer init/fetch/execute case

   end process comb_p;

end Behavioral;
```

Testing

The assembly program of Fig.1 simply subtracts one from the number on input port 0x20 and outputs the result on output port 0x40. To test the I/O, three numbers were inputed, 0x00, 0xFF, and 0x54 to check for correct subtraction. Fig. 2 to Fig. 4 shows the successful testing for these input values. Notice that when the result is outputted, PORT_ID value is 0x40 which is correct. Also, notice that the OUT instruction is executed when PC_COUNT is 0x005. This is correct as instructions are executed after PC_COUNT increments from the instruction's address. The address of the OUT instruction in prog_rom is 0x04 since the program's first instruction is stored at 0x01, therefore OUT will execute when PC_COUNT is 0x05.

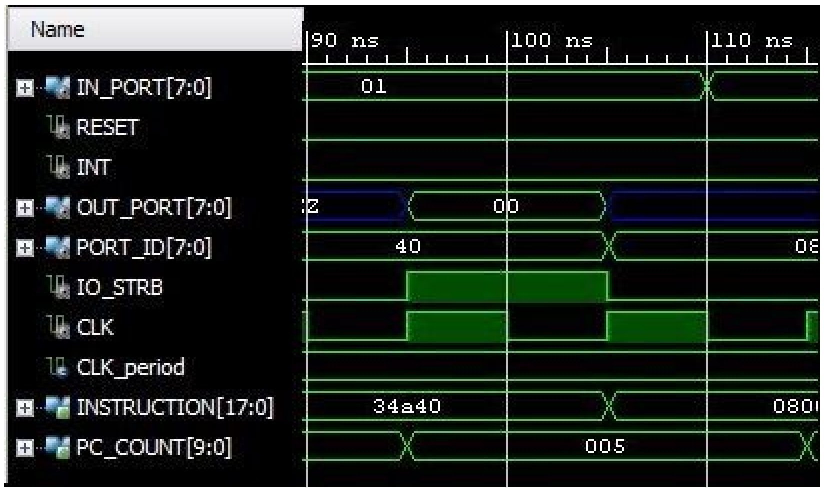


Fig.2. Input of 0x01 with correct output of 0x00.

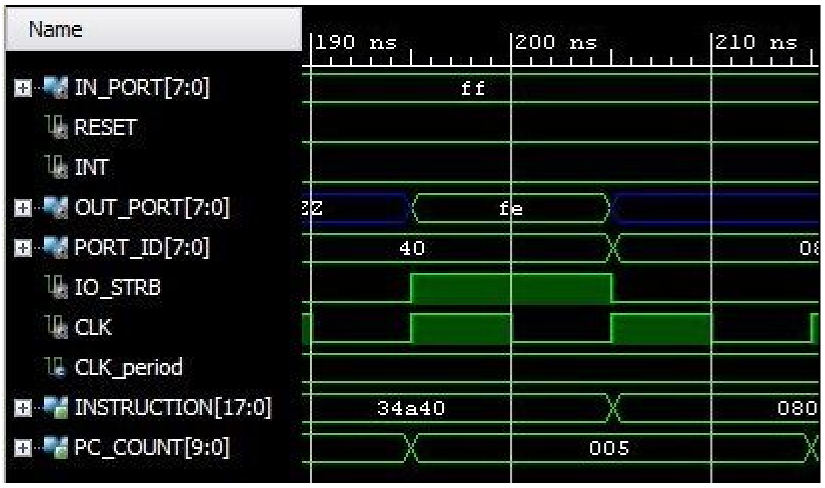


Fig. 3. Input of 0xFF with correct output of 0xFE.

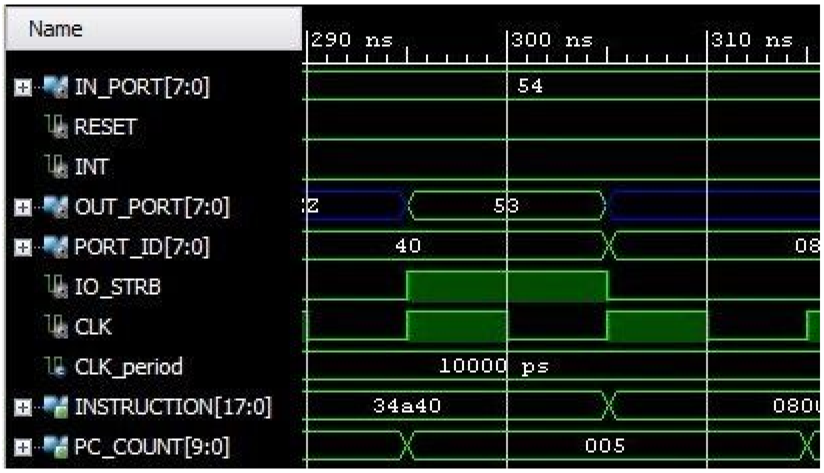


Fig. 4. Input of 0x54 with correct output of 0x53.

Fig. 5 shows the execution of one cycle of the assembly program. The input for this cycle is 0xFF. The program of Fig. 1 has five instructions stored in prog_rom beginning at address 0x01. Each address of PC_COUNT lasts for two clock cycles, except for address numbers 0x001 and 0x006. This is due to the cycles of fetching and execution, which takes a total of two clock cycles, one for fetching and another for execution. Table. 2 shows the instructions and their respective machine code representation. When BRN is executed, INSTRUCTION = 0x08008, the value of PC_COUNT jumps to 0x01. It isn't until the next clock cycle that the instruction stored at 0x001, 0x32A20, is executed.

The temporary INSTRUCTION value of 0x00000 is a result of when PC_COUNT is 0x006. Though there is no instruction stored at this address, the PC still reaches this value as the PC is incremented during the fetch state of the control unit, but since the instruction being executed is a BRN, the address changes before 0x00000 can be executed.

Instruction	**Machine Code**
main: IN r10,SWITCH_PORT	0x32A20
MOV r11,0x01	0x36B01
SUB r10,r11	0x02A5A
OUT r10,LED_PORT	0x34A40
BRN main	0x08008

Table. 2. Instructions stored in prog_rom and their machine code.

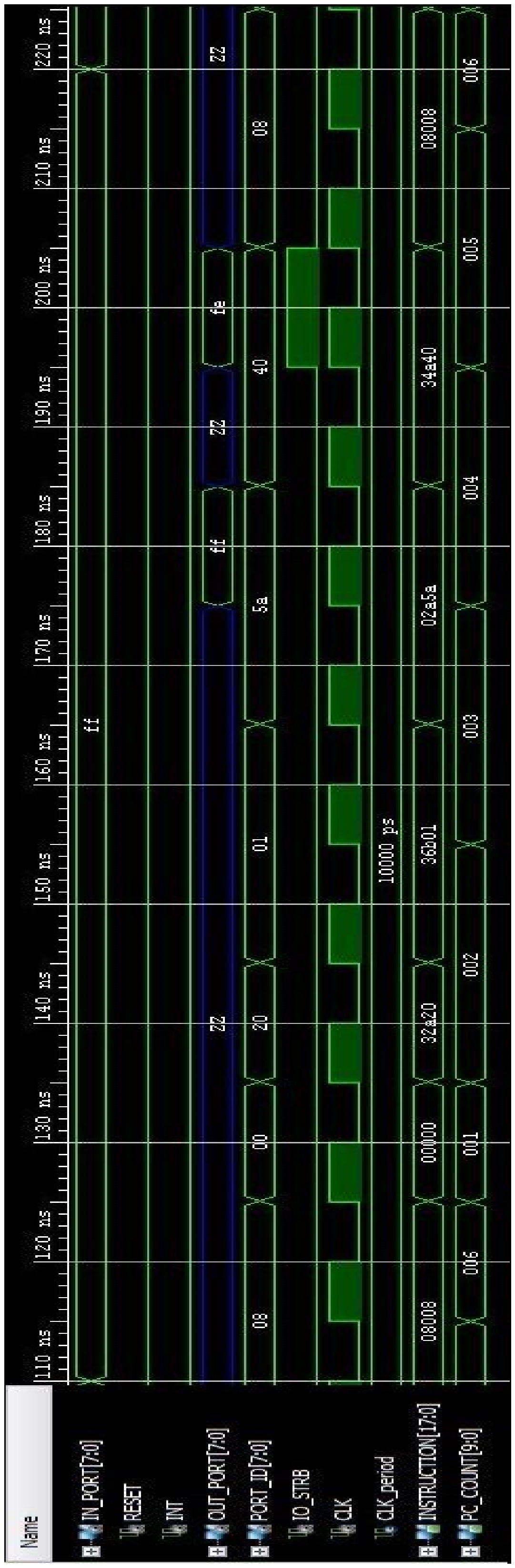


Fig. 5 One cycle of program execution.

Conclusion

Froylan

The basic functionality of the RAT MCU was implemented to execute a simple program by interconnecting modules and implementing the RAT MCUs control unit. The procedure and testing for this experiment will be applied to future experiments as more instructions are implemented in the control unit module. The control unit is an FSM that sends out control signals to the modules to accomplish a certain action (the instruction) in cycles of instruction fetching and execution. In addition, the RAT wrapper may be modified to support the I/O of more complex peripherals connected to the Basys 3 board.

Hasham

Experiment 7 was about putting together a handful of modules to create a partial RAT MCU. The only module used from the past experiments was the PC. The testing, using the given prog_rom, for the RAT MCU produced successful results by providing the outputs that were expected.

Questions

1. *How many different instruction formats does the program in this experiment use? Make a list of the instruction and their formats for this question.*

The program in the experiment uses three formats. The instructions and their formats are shown in the table below.

Instruction	**Format**
IN	reg-immed
MOV	reg-immed
SUB	reg-reg
OUT	reg-immed
BRN	immed

2. *In your own words, briefly but completely describe the purpose of the RAT "wrapper"*

The RAT wrapper allows the RAT MCU to properly interact with the outside world. For this experiment, the outside world is the Basys 3 development board's switches and LEDs. The RAT wrapper is the connection between outside hardware and the RAT MCU's I/O.

3. *Briefly but completely describe the relationship between the IN and OUT instructions and the port_IDs assigned in the RAT Wrapper VHDL model.*

The in_port input is an external input data that is connected to the RAT MCU. The IN instruction is responsible for controlling the MUX that selects the IN_PORT input from the RAT Wrapper. The same can be said for the OUT instructions relationship to the OUT_PORT. The OUT instruction directs the data from the multi-bus into the OUT_PORT output.

4. *Which of the modules you used in this experiment are combinatorial and which are sequential circuits? Be sure to include a complete list of modules when you answer this question.*

Sequential Modules	Combinatorial Modules
● RAT Control Unit ● C and Z Flip Flops ● PC ● FLAGS ● REG_FILE	● RAT Control Unit ● ALU ● Prog_ROM ● REG_FILE ● All the MUXs

5. *This experiment implements the C and Z flags as flip-flops. Briefly describe why the RAT MCU architecture requires that these devices have different control features.*

The C flag flip flops has more control features because the output is used as an input for the ALU. The C flag will need to be loaded, cleared, or set because of its role in the ALU. The Z flag flip flop will only need to be loaded since its value will either be zero or one depending on what the output output of the ALU is.

6. *The RAT MCU implementation in this experiment lacks a few important RAT MCU modules. Estimate which percentage of RAT MCU instructions you can fully implement with the RAT MCU hardware in this experiment.*

Instructions that can NOT be fully Implemented into the RAT MCU hardware
● Interrupt (State) ● Call ● CLI ● LD ● POP ● PUSH ● RET ● RETID ● RETIE ● SEI ● ST ● WSP

Since there is a total of 38 instructions and only 26 of these instructions can be fully implemented, then about 68% of the instructions can be used for the RAT MCU hardware for this experiment.

7. *Briefly explain in the context of the RAT MCU control unit why all FSM outputs were scheduled to be assigned values as the first step in the control unit's combinatorial process.*

The first step of implementing any FSM is to assign default values to the outputs. The outputs for the FSM are always combinatorial and need default values to the outputs to avoid any latches in synthesis and to prevent any problems when running the circuit.

8. *This experiment asked you to include only the control signals that a particular instruction used regardless of whether they were previously scheduled to be assigned to zero(s). Briefly state why this approach represents excellent VHDL coding style.*

Including only relevant control signals keeps the code concise and readable. If all control signals were listed for every instruction, the VHDL code would be unnecessarily long. All relevant signals are listed for each instruction, even the ones previously listed as zero, for easier debugging. This way, the control signals are together in one place for easy reference and possibly modification during testing.

9. *How much memory does the control unit use in this experiment contain? Also, state which signal represents that memory.*

For the control unit, the output signals act as the memory. For this experiment, the memory contained 16 bits total when calculating the signals. The signals that represent the memory were:

- PC_INC
- PC_LD
- PC_MUX_SEL (2 bits)
- ALU_SEL (4 bits)
- RF_OE
- ALU_OPY_SEL
- FLG_C_LD
- FLG_Z_LD
- RF_WR_SEL (2 bits)
- RF_WR
- IO_STRB

10. *In assembly language-land, we refer to instructions that do nothing as "nops" (pronounced "know ops"). In academia, we refer to "nops" as administrators. Many assembly language instructions actually have a dedicated nop instruction, but the RAT MCU does not. There are two approaches to faking a nop instruction in VHDL using the existing RAT MCU instruction sets. For this questions, list those two approaches. HINT: one of these approaches involves a branch instruction.*

- MOV rA, rA ; where rA is a register
- nops : BRN nops

11. Describe a situation where a NOP instruction or a NOP-type instruction would be useful.

NOP instructions are useful for timing constraints to avoid any jumps. The NOP instruction can be beneficial by placing a timing delay since this instruction will go through the FETCH and EXECUTE cycle. NOP instructions can also be useful to allow the previous instruction to finish its job.

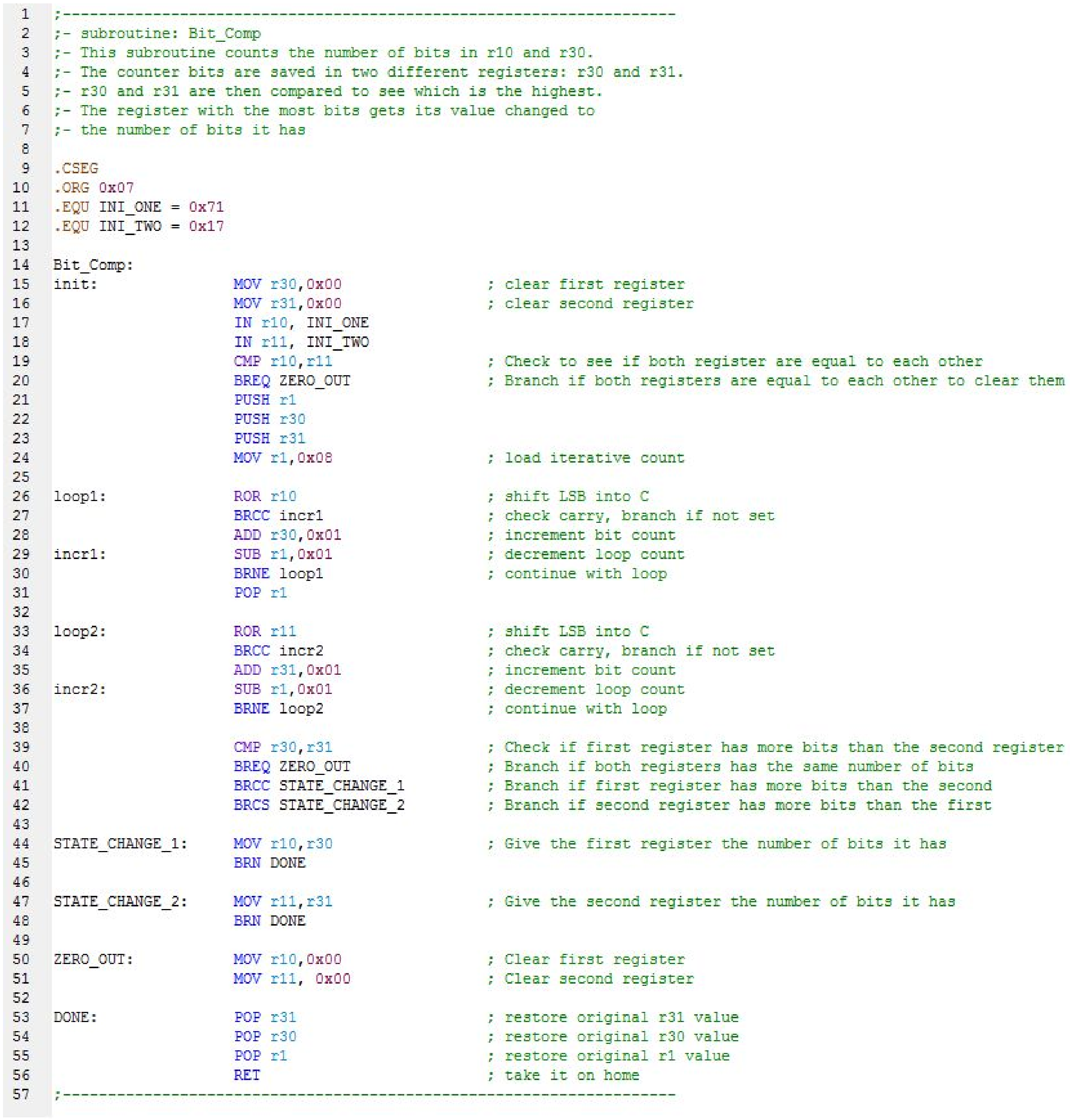
12. Briefly describe the difference in control signals between reg/reg and a reg/imm-type instruction for any single ALU-based instruction. Comment on whether these differences will expedite your work when you're required to implement a majority of the RAT instructions in a later next experiment.

Most of the instructions for the RAT will basically have two parts to it (reg/reg and a reg/imm-type). The both reg/reg and reg/imm-type will use the same ALU-based instructions since the ALU is only taking two data inputs, whether it is from a register or from the address. There will not be any changes made to the ALU, but in the control unit, there will be almost double the amount of instructions due to these different signal types. The reg/reg and reg/imm-type instructions will only be different by the signal that is sent to its corresponding MUX.

Hardware Design Assignment

a) A nibble swap operation would have to be added to the RAT ALU, and the appropriate control signals for the nibble swap added to the control unit. This nibble swap instruction would be 0xF for ALU_SEL.
b) In terms of the RAT assembler, the machine code translation for the mnemonic NBSWP would need to be programmed into the assembler.
c) There would be no modifications necessary to the RAT MCU memory.

Programming Assignment



```
;---------------------------------------------------------------------
;- subroutine: Bit_Comp
;- This subroutine counts the number of bits in r10 and r30.
;- The counter bits are saved in two different registers: r30 and r31.
;- r30 and r31 are then compared to see which is the highest.
;- The register with the most bits gets its value changed to
;- the number of bits it has

.CSEG
.ORG 0x07
.EQU INI_ONE = 0x71
.EQU INI_TWO = 0x17

Bit_Comp:
init:               MOV r30,0x00                ; clear first register
                    MOV r31,0x00                ; clear second register
                    IN r10, INI_ONE
                    IN r11, INI_TWO
                    CMP r10,r11                 ; Check to see if both register are equal to each other
                    BREQ ZERO_OUT               ; Branch if both registers are equal to each other to clear them
                    PUSH r1
                    PUSH r30
                    PUSH r31
                    MOV r1,0x08                 ; load iterative count

loop1:              ROR r10                     ; shift LSB into C
                    BRCC incr1                  ; check carry, branch if not set
                    ADD r30,0x01                ; increment bit count
incr1:              SUB r1,0x01                 ; decrement loop count
                    BRNE loop1                  ; continue with loop
                    POP r1

loop2:              ROR r11                     ; shift LSB into C
                    BRCC incr2                  ; check carry, branch if not set
                    ADD r31,0x01                ; increment bit count
incr2:              SUB r1,0x01                 ; decrement loop count
                    BRNE loop2                  ; continue with loop

                    CMP r30,r31                 ; Check if first register has more bits than the second register
                    BREQ ZERO_OUT               ; Branch if both registers has the same number of bits
                    BRCC STATE_CHANGE_1         ; Branch if first register has more bits than the second
                    BRCS STATE_CHANGE_2         ; Branch if second register has more bits than the first

STATE_CHANGE_1:     MOV r10,r30                 ; Give the first register the number of bits it has
                    BRN DONE

STATE_CHANGE_2:     MOV r11,r31                 ; Give the second register the number of bits it has
                    BRN DONE

ZERO_OUT:           MOV r10,0x00                ; Clear first register
                    MOV r11, 0x00               ; Clear second register

DONE:               POP r31                     ; restore original r31 value
                    POP r30                     ; restore original r30 value
                    POP r1                      ; restore original r1 value
                    RET                         ; take it on home
;---------------------------------------------------------------------
```