PART I: SERIAL LOGIC ELEMENTS

CHAPTER 1: SERIAL ARITHMETIC UNIT

1.0 Serial Arithmetic Unit Summary

The design of circuits for performing various operations on serial data is carried out in this project. The project involves combinational logic design and the use of shift registers.

1.1 Arithmetic and Logic Units

An Arithmetic and Logic Unit (ALU) is a circuit that performs various arithmetic and logic operations on two input operands. Typical ALUs perform binary operations such as add, subtract, AND, OR, XOR; unary operations such as increment, decrement, transfer input directly to output; and constant operations such as output constant 0 or 1. The 74181, 74281, and 74381 are examples of MSI ALUs that operate on 4 bit parallel data. It is also possible to design serial ALUs for which the two input operands and the output are stored in shift registers and processed one bit at a time. The advantage of a serial ALU is that only one copy of the circuitry for one ALU bit is needed to process n-bit data, rather than n copies for a parallel ALU; the disadvantage is that n clock cycles are required to process the data.

1.1.1 The Serial ALU, Part I

The assignment is to design a 6-input, 2-output combinational ALU circuit that processes two 1-bit operands. The circuit should have two data inputs A and B, a carry input CI, three function select inputs F1, F2, F3, a data output S, and a carry output CO. The circuit should perform eight functions as shown in table 1. For each operation, the value of S should be computed as shown; for the arithmetic operations (increment, negate, add,

subtract) the value of CO should be computed as the carry from the indicated addition.

F1 F2 F3	Operation	Name
0 0 0	S <- 0	Clear
0 0 1	S <- A'	Complement
0 1 0	S <- A plus CI	Increment
0 1 1	S <- A' plus CI	Negate
1 0 0	S <- B	Transfer B
1 0 1	S <- A XOR B	Exclusive OR
1 1 0	S <- A plus B plus CI	Add
1 1 1	S <- A plus B' plus CI	Subtract

Table 1. ALU Functions

1.1.2 The Assignment, Part II

The ALU of part I, can be used to serially process operands of any length. Complete the design of the system in Figure 1, which uses two shift registers to hold 4 bit operands A and B. The system should have the following inputs and Outputs:

F1, F2, F3: These three signals shall set the desired function

B-In: These four signals determine the values of the B register set by a parallel load.

Load: This signal is used to load B-In into the B register and also should set the carry flip-flop with the proper initial carry for arithmetic operations as determined by F1, F2, F3.

Clock: This signal controls the shifting of A and B and loading of the carry flip-flop for performing operations. The result should be

contained in A, and B should be restored to its loaded value, after four clock pulses.

A-REG: These four signals should show the contents of the A register.

B-REG: These four signals should show the contents of the B register.

C: This signal should show the contents of the carry flip-flop.

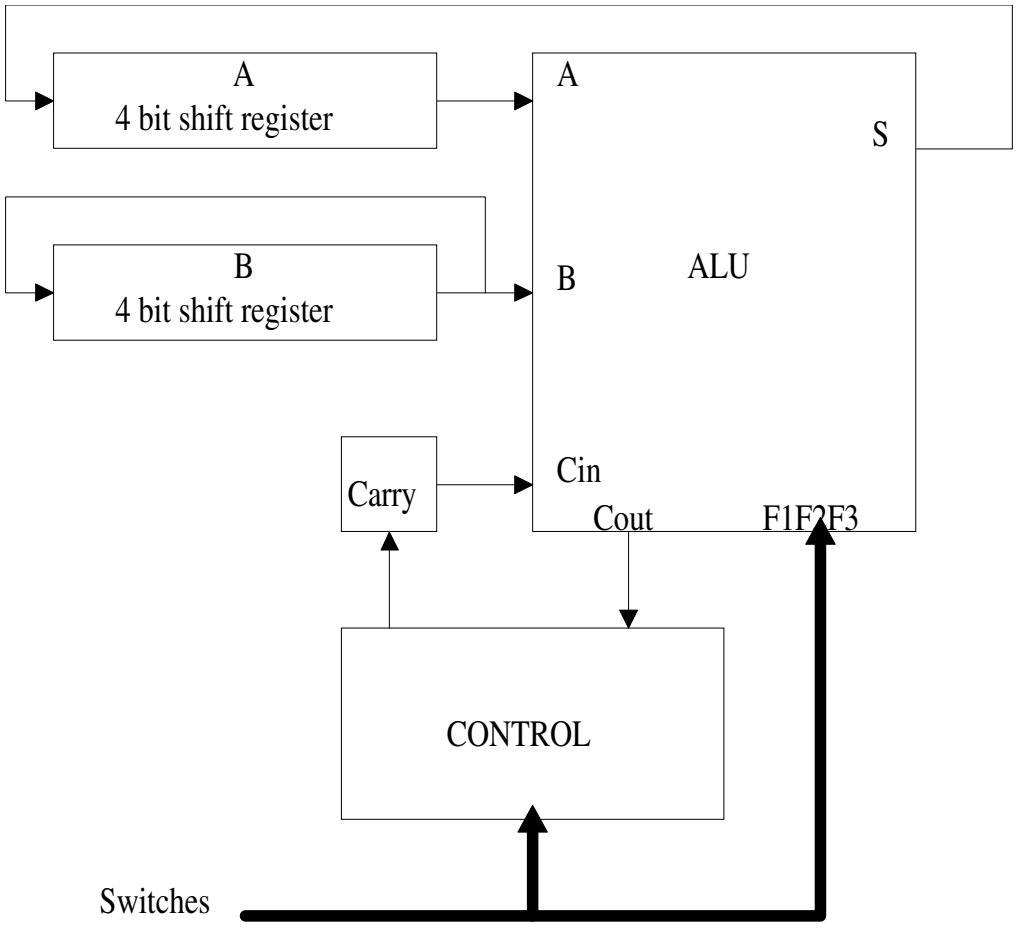

Figure 1. ALU Block Diagram

1.2 The VHDL Solution, Part I

Representing an Arithmetic and Logic Unit in VHDL is a straight forward task. The code is a straight forward implementation of the ALU. The functions are implemented by using a case statement based upon the function select inputs.

1.2.1 The ALU Part I VHDL Entity

The ALU's interface is implemented as defined above. The following code is the interface:

```
ENTITY alu_1 IS
  PORT(opcode       : IN    bit_vector(2 DOWNTO 0);
       a            : IN    bit;
       b            : IN    bit;
       cin          : IN    bit;
       clock        : IN    bit;
       s            : OUT   bit;
       cout         : OUT   bit);
END alu_1;
```

Listing 1. ALU Entity Statement

1.2.2 The ALU Part I VHDL Architecture

The Following code is the VHDL architecture solution for the 1-bit ALU:

```
ARCHITECTURE behave_alu_1 OF alu_1 IS

BEGIN  --  behave
  PROCESS
    BEGIN
      WAIT UNTIL clock'EVENT and clock='1';
      CASE opcode IS
        WHEN "000" =>                    --  clear, s <= 0
          s <= '0';
        WHEN "001" =>                    --  compliment, s <= a'
          s <= NOT(a);
        WHEN "010" =>                    --  increment, s <= a+cin
          s <= a XOR cin;
          cout <= a AND cin;
        WHEN "011" =>                    --  negate, s <= a' + cin
```

```
          s <= NOT(a XOR cin);
          cout <= (NOT(a)) AND cin;
        WHEN "100" =>                       --  transfer b, s <= b
          s <= b;
        WHEN "101" =>                       --  XOR, a XOR b
          s <= a XOR b;
        WHEN "110" =>                       --  Add, a + b + cin
          s <= (a XOR b) XOR cin;
          cout <= (a AND cin) OR (b AND (a XOR cin));
        WHEN "111" =>                       --  Subtract, a + b' + cin
          s <= a XOR (NOT(b)) XOR cin;
          cout <= (a AND cin) OR ((NOT(b)) AND (a XOR cin));
        WHEN OTHERS =>
      END CASE;                             --  opcode
  END PROCESS;
END behave_alu_1;
```

Listing 2. ALU Architecture

The ALU architecture is implemented as a VHDL case statement sensitive to the function selects. This interface is not purely combinational, as the entire process is sensitive to the clock. This code is going to be used in the next section to implement a four bit ALU. This code has been synthesized into LSI Logic 100K ASIC primitives using the Synopsys VHDL Synthesis tool. This code appears in Appendix G.

1.2.3 The ALU Part I Test bench

To test the first ALU, we can easily step through all the sixteen cases. This is shown below in listing 3.

```
--------------------------------------------------------------------------
-------
--  EE126 Project #1, part I Testbench
--  by: Frank Bruno
--------------------------------------------------------------------------
-------

ENTITY test_alu_1 IS

END test_alu_1;

ARCHITECTURE b_alu_1 OF test_alu_1 IS
  COMPONENT alu_1
  PORT(opcode       : IN    bit_vector(2 DOWNTO 0);
       a            : IN    bit;
```

```
        b            : IN    bit;
        cin          : IN    bit;
        clock        : IN    bit;
        s            : OUT   bit;
        cout         : OUT   bit);
  END COMPONENT;

  SIGNAL opcode : bit_vector(2 DOWNTO 0);
  SIGNAL a : bit;
  SIGNAL b : bit;
  SIGNAL cin : bit;
  SIGNAL s : bit;
  SIGNAL cout : bit;
  SIGNAL clock : bit;
BEGIN  --  b_alu_1
  u1: alu_1
  PORT MAP(opcode => opcode,
           a => a,
           b => b,
           cin => cin,
           clock => clock,
           s => s,
           cout => cout);

clock_gen:
PROCESS
  BEGIN
    clock <= '0' AFTER 40 ns;
    WAIT FOR 40 ns;
    clock <= '1' AFTER 40 ns;
    WAIT FOR 40 ns;
END PROCESS;

PROCESS
  VARIABLE state : integer := 0;
  VARIABLE jump : integer;
  BEGIN
    WAIT UNTIL clock'EVENT AND clock ='1';
    CASE state IS
      WHEN 0 =>
        a <= '0';
        b <= '0';
        cin <= '0';
        jump := 1;
        state := 1;
      WHEN 1 =>
        opcode <= "000";
        state := 2;
      WHEN 2 =>
        opcode <= "001";
        state := 3;
      WHEN 3 =>
        opcode <= "010";
        state := 4;
      WHEN 4 =>
        opcode <= "011";
        state := 5;
      WHEN 5 =>
        opcode <= "100";
        state := 6;
```

```
      WHEN 6 =>
        opcode <= "101";
        state := 7;
      WHEN 7 =>
        opcode <= "110";
        state := 8;
      WHEN 8 =>
        opcode <= "111";
        state := (8 + jump);
      WHEN 9 =>
        a <= '0';
        b <= '0';
        cin  <= '1';
        jump := 2;
        state := 1;
      WHEN 10 =>
        a <= '0';
        b <= '1';
        cin <= '0';
        jump := 3;
        state := 1;
      WHEN 11 =>
        cin <= '1';
        jump := 4;
        state := 1;
      WHEN 12 =>
        a <= '1';
        b <= '0';
        cin <= '0';
        jump := 5;
        state := 1;
      WHEN 13 =>
        cin <= '1';
        jump := 6;
        state := 1;
      WHEN 15 =>
        b <= '1';
        cin <= '0';
        jump := 7;
        state := 1;
      WHEN 16 =>
        cin <= '1';
        jump := 8;
        state := 1;

      WHEN OTHERS =>
    END CASE;
END PROCESS;
END b_alu_1;

CONFIGURATION config1 OF test_alu_1 IS
FOR b_alu_1
  FOR u1: alu_1
    USE ENTITY work.alu_1;
  END FOR;
END FOR;
END config1;
```

Listing 3. ALU I Test bench

1.3 The VHDL Solution to Part II

Using the ALU derived in part I, it is possible to create a variable length ALU by using two shift registers and control logic. The VHDL implementation follows.

1.3.1 The ALU Part II VHDL Entity

The VHDL entity statement for the 4 bit serial ALU follows:

```
ENTITY alu_2 IS
  PORT(op code       : IN    bit_vector(2 DOWNTO 0);
       bin           : IN    bit_vector(3 DOWNTO 0);
       load          : IN    bit;
       clock         : IN    bit;
       areg          : BUFFER   bit_vector(3 DOWNTO 0);
       breg          : BUFFER   bit_vector(3 DOWNTO 0);
       carry         : BUFFER   bit);
END alu_2;
```

Listing 4. ALU II Entity Statement

1.3.2 The ALU Part II VHDL Architecture

The VHDL architecture is based upon the ALU part I above. Below is the actual VHDL code:

```
ARCHITECTURE behave_alu_2 OF alu_2 IS

BEGIN  --  behave_alu_2

  PROCESS
    VARIABLE a : bit;
    VARIABLE b : bit;
    VARIABLE cin : bit;
    VARIABLE s : bit;
    VARIABLE cout : bit;

    BEGIN  --  PROCESS
    WAIT UNTIL clock'EVENT and clock = '1';
    IF (load = '1') THEN
      breg <= bin;
      areg <= "0000";
    ELSE
      a := areg(0);
      areg(0) <= areg(1);
      areg(1) <= areg(2);
      areg(2) <= areg(3);
```

```
      b := breg(0);
      breg(0) <= breg(1);
      breg(1) <= breg(2);
      breg(2) <= breg(3);
      breg(3) <= b;
      cin := carry;
    CASE opcode IS
      WHEN "000" =>                     --  clear, s <= 0
        s := '0';
      WHEN "001" =>                     --  compliment, s <= a'
        s := NOT(a);
      WHEN "010" =>                     --  increment, s <= a+cin
        s := a XOR cin;
        cout := a AND cin;
      WHEN "011" =>                     --  negate, s <= a' + cin
        s := NOT(a XOR cin);
        cout := (NOT(a)) AND cin;
      WHEN "100" =>                     --  transfer b, s <= b
        s := b;
      WHEN "101" =>                     --  XOR, a XOR b
        s := a XOR b;
      WHEN "110" =>                     --  Add, a + b + cin
        s := (a XOR b) XOR cin;
        cout := (a AND cin) OR (b AND (a XOR cin));
      WHEN "111" =>                     --  Subtract, a + b' + cin
        s := a XOR (NOT(b)) XOR cin;
        cout := (a AND cin) OR ((NOT(b)) AND (a XOR cin));
      WHEN OTHERS =>
    END CASE;                           --  opcode
    areg(3) <= s;
    carry <= cout;
    END IF;
  END PROCESS;

END behave_alu_2;
```

Listing 5. ALU II Architecture

The above code implements the ALU and is based upon the ALU from part I. It was decided that in a simple case such as this that I would implement the ALU as a flat model, no structure to it.

1.3.3 The ALU Part II VHDL Test bench

It was shown in part I that the basic ALU worked properly. Therefore, to test this ALU, we will try a couple of operands.

```
-------------------------------------------------------------------------
-------
--  EE126 Project #1, part II Testbench
--  by: Frank Bruno
-------------------------------------------------------------------------
-------

ENTITY test_alu_2 IS

END test_alu_2;

ARCHITECTURE b_alu_2 OF test_alu_2 IS
  COMPONENT alu_2
    PORT(opcode       : IN    bit_vector(2 DOWNTO 0);
         bin          : IN    bit_vector(3 DOWNTO 0);
         load         : IN    bit;
         clock        : IN    bit;
         areg         : BUFFER   bit_vector(3 DOWNTO 0);
         breg         : BUFFER   bit_vector(3 DOWNTO 0);
         carry        : BUFFER   bit);
  END COMPONENT;
  SIGNAL opcode : bit_vector(2 DOWNTO 0);
  SIGNAL bin : bit_vector(3 DOWNTO 0);
  SIGNAL load : bit;
  SIGNAL areg : bit_vector(3 DOWNTO 0);
  SIGNAL breg : bit_vector(3 DOWNTO 0);
  SIGNAL clock : bit;
  SIGNAL carry : bit;
BEGIN  --  b_alu_2
  u1: alu_2
  PORT MAP(opcode => opcode,
           bin => bin,
           load => load,
           clock => clock,
           areg => areg,
           breg => breg,
          carry => carry);

clock_gen:
PROCESS
  BEGIN
    clock <= '0' AFTER 40 ns;
    WAIT FOR 40 ns;
    clock <= '1' AFTER 40 ns;
    WAIT FOR 40 ns;
END PROCESS;

PROCESS
  VARIABLE state : integer := 0;
  BEGIN
    WAIT UNTIL clock'EVENT AND clock ='1';
    CASE state IS
      WHEN 0 =>
      load <= '1';
      bin <= "1010";
        opcode <= "000";
        state := 1;
      WHEN 1 =>
      load <= '0';
        state := 2;
```

```
      WHEN 2 =>
        state := 3;
      WHEN 3 =>
        state := 4;
      WHEN 4 =>
        state := 5;
      WHEN 5 =>
        opcode <= "110";
        state := 6;
      WHEN OTHERS =>
    END CASE;
END PROCESS;

END b_alu_2;

CONFIGURATION config2 OF test_alu_2 IS
FOR b_alu_2
  FOR u1: alu_2
    USE ENTITY work.alu_2;
  END FOR;
END FOR;
END config2;
```

Listing 6. ALU II Testbench

CHAPTER 2. SERIAL BINARY MULTIPLIER

2.0 Serial Binary Multiplier Summary

This project requires the design of a four bit multiplier using serial techniques. It provides exposure to shift registers and control logic design.

2.1 The Serial Binary Multiplier Assignment

This project is to design a serial multiplier to form the eight bit product of two four bit operands. Use an "add-and-shift" algorithm in which individual four bit additions are performed serially. The circuit should have the following inputs and outputs:

MULTIPLIER -	These four signals are used to set the value of the multiplier.
MULTIPLICAND-	These four signals are used to set the value of the multiplicand.
INIT-	This signal initializes the multiplier for a new multiplication. Setting this signal will clear the product register, load the operands, and initialize the control circuitry.
CLOCK-	This signal is used to actually perform the serial multiplication. A worst case multiplication should require about 20 clock pulses.
DONE-	This signal, reset by INIT, should be high when the multiplication is complete.
PRODUCT-	These signals display the product.

Circuitry should be included that will shorten the execution time when there are zeros in the multiplier. Briefly indicate how the circuit could be extended to multiply 16 bit operands.

2.2 VHDL solution to the Serial Binary Multiplier

The VHDL solution to the problem that was chosen is one which uses the Booth's algorithm for serial multiplication. This allows all the criteria to be met. Booth's algorithm is very easy to implement using software. Since VHDL software turns into hardware, it seemed to be the ideal choice.

2.2.1 The Serial Binary Multiplier VHDL Entity

The Multipliers interface is defined above, and the ENTITY follows the definition as written. The following code defines the interface:

```
ENTITY mult IS
  PORT(multiplier      : IN     bit_vector(3 DOWNTO 0);
       multiplicand    : IN     bit_vector(3 DOWNTO 0);
       clock           : IN     bit;
       init            : IN     bit;
       ac              : BUFFER bit_vector(3 DOWNTO 0);
       qout            : BUFFER bit_vector(3 DOWNTO 0);
       sc              : BUFFER integer range 0 to 4;
       qr              : BUFFER bit_vector(4 DOWNTO 0);
       br              : BUFFER bit_vector(3 DOWNTO 0);
       done            : OUT    bit);
END mult;
```

Listing 7. Serial Binary Multiplier Entity

2.2.2 The Serial Binary Multiplier VHDL Architecture

The Architecture for the serial multiplier is based upon the case statement. The CASE statement is ideal for this, as booths algorithm is essentially a sequential algorithm with logic to steer the direction of the case. Below is the ARCHITECTURE with explanations:

```
ARCHITECTURE behave_mult OF mult IS

BEGIN  --  behave

  multiply:
  PROCESS
    VARIABLE state : integer range 0 to 3;
    VARIABLE status : bit_vector(3 DOWNTO 0);
```

```
  VARIABLE result : bit_vector(3 DOWNTO 0);
  VARIABLE as : bit;
  VARIABLE temp : bit_vector(3 DOWNTO 0);

  BEGIN
    WAIT UNTIL clock'EVENT and clock='1';
    IF (init = '0') THEN
      ac <= "0000";
      br <= multiplicand;
      qr(4 DOWNTO 1) <= multiplier;
      ac <= "0000";
      sc <= 4;
      qr(0) <= '0';
      done <= '0';
      state := 0;
    ELSE
      CASE state IS
        WHEN 0 =>
          CASE qr(1 DOWNTO 0) IS
            WHEN "01" =>
              ac <= ac + br;
            WHEN "10" =>
              ac <= ac - br;
            WHEN OTHERS =>
          END CASE;                    --  qr(1 DOWNTO 0)
          state := 1;
        WHEN 1 =>
          qr(0) <= qr(1);
          qr(1) <= qr(2);
          qr(2) <= qr(3);
          qr(3) <= qr(4);
          qr(4) <= ac(0);
          ac(0) <= ac(1);
          ac(1) <= ac(2);
          ac(2) <= ac(3);
          sc <= sc - 1;
          state := 2;
        WHEN 2 =>
          IF (sc = 0) THEN
            state := 3;
          ELSE
            state := 0;
          END IF;
        WHEN 3 =>
          done <= '1';
          qout <= qr(4 DOWNTO 1);
          state := 0;
        WHEN others =>

      END CASE;                        --  state
    END IF;
END PROCESS;
```

Listing 8. Serial Binary Multiplier Architecture

2.2.3 Initialization Section

The initialization section is needed to ensure that the registers are cleared before we begin the multiplication operation. This is very important, for if this is not done, we will have old values in the registers which will give us incorrect results. Below is the code for initialization:

```
IF (init = '0') THEN
  ac <= "0000";
  br <= multiplicand;
  qr(4 DOWNTO 1) <= multiplier;
  ac <= "0000";
  sc <= 4;
  qr(0) <= '0';
  done <= '0';
  state := 0;
ELSE
```

Listing 9. Serial Binary Multiplier Initialization Code

2.2.2 Booth's Algorithm

We can see the remainder of the process is a representation of Booth's algorithm. Briefly, Case 0 determines the value which is loaded into the AC register. Case 1 performs the actual shifting to the values of AC and QR registers. Case 2 determines whether we have completed the algorithm. If we have, we go to case 3, which will put us back into state 0 to start over again.

2.3 The Serial Binary Multiplier Test bench

A simple test bench to test the multiplication of two numbers appears below:

```
-------------------------------------------------------------------------
-------
--  EE126 Project #1, part I Testbench
--  by: Frank Bruno
-------------------------------------------------------------------------
-------

ENTITY test_mult IS
```

```
END test_mult;

ARCHITECTURE b_mult OF test_mult IS
  COMPONENT mult
  PORT(multiplier      : IN     bit_vector(3 DOWNTO 0);
       multiplicand    : IN     bit_vector(3 DOWNTO 0);
       clock           : IN     bit;
       init            : IN     bit;
       ac              : BUFFER bit_vector(3 DOWNTO 0);
       qout            : BUFFER bit_vector(3 DOWNTO 0);
       sc              : BUFFER integer range 0 to 4;
       qr              : BUFFER bit_vector(4 DOWNTO 0);
       br              : BUFFER bit_vector(3 DOWNTO 0);
       done            : OUT    bit);
  END COMPONENT;
  SIGNAL multiplier : bit_vector(3 DOWNTO 0);
  SIGNAL multiplicand : bit_vector(3 DOWNTO 0);
  SIGNAL clock : bit;
  SIGNAL init : bit;
  SIGNAL ac : bit_vector(3 DOWNTO 0);
  SIGNAL qout : bit_vector(3 DOWNTO 0);
  SIGNAL done : bit;
  SIGNAL sc : integer range 0 to 4;
  SIGNAL qr : bit_vector(4 DOWNTO 0);
  SIGNAL br : bit_vector(3 DOWNTO 0);

BEGIN  --  b_mult
  u1: mult
  PORT MAP(multiplier => multiplier,
           multiplicand  => multiplicand,
           clock => clock,
           init => init,
           ac => ac,
           qout => qout,
           sc => sc,
           qr => qr,
           br => br,
           done => done);

clock_gen:
PROCESS
  BEGIN
    clock <= '0' AFTER 40 ns;
    WAIT FOR 40 ns;
    clock <= '1' AFTER 40 ns;
    WAIT FOR 40 ns;
END PROCESS;

PROCESS
  VARIABLE state : integer := 0;
  BEGIN
    WAIT UNTIL clock'EVENT AND clock ='1';
    CASE state IS
      WHEN 0 =>
      init <= '0';
      multiplier <= "1011";
        multiplicand <= "1001";
        state := 1;
      WHEN 1 =>
        init <= '1';
```

```
        IF (done = '1') THEN
          init  <= '0';
          multiplier <= "0111";
          multiplicand <= "0101";
          state := 2;
        END IF;
      WHEN 2 =>
        init  <= '1';
        state := 3;
      WHEN 3 =>
        IF (done = '1') THEN
          init  <= '0';
          multiplier <= "1011";
          multiplicand  <= "0110";
          state := 4;
        END IF;
      WHEN 4 =>
        init  <= '1';
        state := 5;
      WHEN OTHERS =>
    END CASE;
END PROCESS;

END b_mult;

CONFIGURATION c_mult OF test_mult IS
FOR b_mult
  FOR u1: mult
    USE ENTITY work.mult;
  END FOR;
END FOR;
END c_mult;
```

Listing 10. Serial Binary Multiplier Testbench

2.4 The Serial Binary Multiplier Synthesized Results

The synthesized code for the Serial multiplier appears in Appendix L.

PART II COMPUTING ELEMENTS

CHAPTER 3. A SIMPLE PROCESSOR

3.0 A Simple Processor Summary

A simple processor for four bit operands is designed in this project. The processor has four 4-bit registers, a number of 8-bit instructions, and memory for a 16 instruction program. A sample program is a multiply routine.

3.1 Simple Processor Functional Description

This section gives the description of a small processor. Figure 2 is a block diagram of the processor as it would appear to a programmer. The processor has all the rudimentary elements of a computer, but it has insufficient features to serve in any practical capacity. The machine has three classes of instructions: register instructions, branch instructions, and halt-I/O instructions.

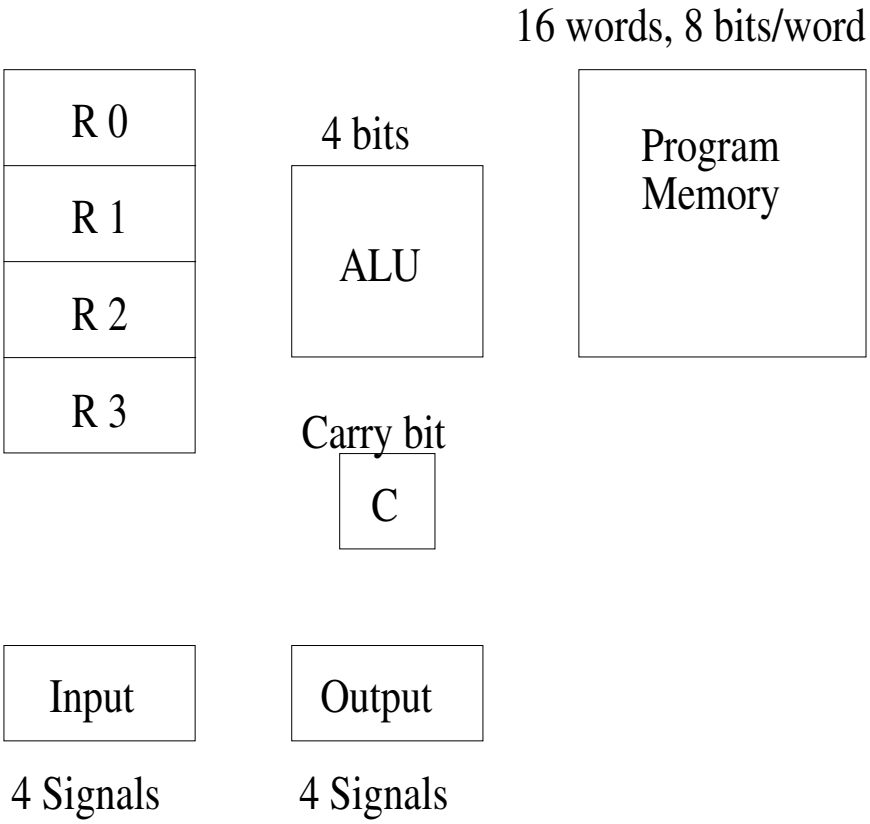

Figure 2. Simple Processor Structure

3.1.1 Register Instructions

Register instructions perform an operation on the two registers specified by S and D. The notation (S) denotes the contents of the register specified by S. A function specified by OP and CC is performed on (S) and (D) and the result stored in (D). OP specifies one of the operations and CC controls the carry-in bit used in the operation. The meanings of OP and CC are detailed in Tables 2 and 3. The values of C_I for each operation is determined by CC as shown. C is the contents of the carry flip-flop shown in the block diagram.

OP	Function
00	(S) + C_I -> (D)
01	(S) + (D) + C_I -> (D)
10	(D) - (S) - C_I' -> (D)

Table 2. Basic Register Instructions

CC	C_I
00	0
01	1
10	C
11	C'

Table 3. Carry Bit

Register instructions also cause the C flip-flop to be loaded. C equals 1 whenever an addition overflows or when a subtraction does not produce a borrow. No other commands affect the C flip-flop. Table 5 gives some examples of register instructions.

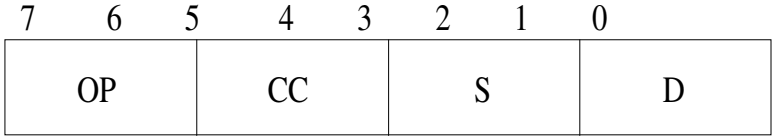

Table 4. Register Bit Fields

MNEMONIC			OPERATION	CODE
CLEAR		R2	R2 - R2 -> R2	10 01 10 10
INCREMENT		R3	R3 + 1 -> R3	00 01 11 11
ROTATE		R1	R1 + R1 -> R1	01 00 01 01
ADD	R1,	R0	R1 + R0 -> R0	01 00 01 00

SUBTRACT	R2, R0	R0 - R2 -> R0	10 01 10 00
MOVE	R3, R2	R3 -> R2	00 00 11 10

Table 5. Extended Register Instructions

3.1.2 Branch Instructions

Branch Instructions cause a branch to the specified address in instruction memory if C_I is equal to 1. Again, C_I is defined by CC, unless CC = 0, which indicates another instruction class. This gives us three branch instructions: unconditional, conditional on C and, and conditional on C'.

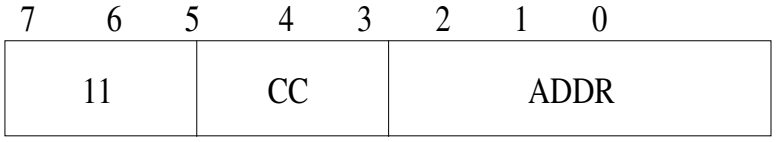

Table 6. Branch Instruction Bit Fields

3.1.3 Halt and I/O Instructions

Halt and Input/Output instructions are obtained when OP=11 and CC=00, as shown in Table 7. If the L bit is set, then the instruction loads (inputs) the data from the switch register into the register specified by D. The H bit specifies that the machine is to halt after execution of the instruction. When the machine halts, the register specified by D is displayed in the output signal. Note that both L and H may be equal to one, which causes the register to be loaded and then displayed in the lights.

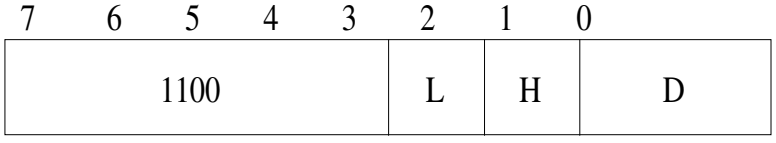

Table 7. Halt and I/O Instructions

3.1.4 Simple Computer I/O

The Following IO is required for the simple computer:

RESET	Initializes control and clear PC
START	Starts program execution at PC
INCPC	Increments the PC when the machine is halted
INSTR	Eight signals to specify an instruction
LOAD	Loads MEMORY(PC) with INSTR
SW	Four signals which act as toggle switches
DATA OUT	Displays RD at all times
PC OUT	Displays PC at all times

3.2 A VHDL implementation of the EE126 Simple Computer

The above definition of a simple digital computer is to be carried over into a VHDL program. We will include a few extra signals in the ENTITY statement to allow monitoring of internal conditions. For synthesis, we divide the simple computer into a processor and a memory segment. This allows us to only compile the actual processing section, without the memory.

3.2.1 Simple Computer Memory

The memory section of the simple computer is defined below. It allows us to define simple arrays to hold the instructions.

```
-------------------------------------------------------------------------
-------
--  Simple processor from EE126, memory segment
--  By: Frank Bruno
--  for Professor Chang
-------------------------------------------------------------------------
-------
```

```
USE work.alu_pkg.ALL;
USE work.bv_arithmetic.ALL;

ENTITY simple_memory IS
  PORT(instr         : IN bit_vector(7 DOWNTO 0);
       pc            : IN integer range 0 to 15;
       load          : IN bit;
       clock         : IN bit;
       current_mem   : OUT bit_vector(7 DOWNTO 0)
       );
END simple_memory;

ARCHITECTURE behave_simple_mem OF simple_memory IS
  SIGNAL memory : bit_8(15 DOWNTO 0);

BEGIN  --  behave_simple_mem

  PROCESS(pc, load)

    BEGIN

-------------------------------------------------------------------------
-------
--  on a load event, load the memory
-------------------------------------------------------------------------
-------
      IF (load = '1') THEN
        memory(pc) <= instr;
      END IF;

-------------------------------------------------------------------------
-------
--  let the program know the current memory pointed to by PC
-------------------------------------------------------------------------
-------
      current_mem <= memory(pc);

  END PROCESS;

END behave_simple_mem;
```

Listing 11. Simple Computer Memory Section

3.2.2 Simple Computer VHDL Entity

The simple computer ENTITY defines the I/O to the system. It appears in full below:

```
ENTITY processor IS
  PORT(carry           : BUFFER bit;
       out_reg         : BUFFER bit_vector(3 DOWNTO 0);
       pc_out          : BUFFER bit_vector(3 DOWNTO 0);
       input           : IN bit_vector(3 DOWNTO 0);
       clock           : IN bit;
```

```
       reset           : IN bit;
       inc_pc          : IN bit;
       instr           : IN bit_vector(7 DOWNTO 0);
       load            : IN bit;
       start           : IN bit
       );
END processor;
```

Listing 12. Simple Computer ENTITY

3.2.3 Simple Computer VHDL Architecture

The simple computer architecture is a complex CASE statement which handles all of the simple computer instructions. It handles decoding and actual execution. The architecture appears below:

```
ARCHITECTURE process_behave OF processor IS
  COMPONENT simple_memory
    PORT(instr         : IN bit_vector(7 DOWNTO 0);
         pc            : IN integer range 0 to 15;
         load          : IN bit;
         clock         : IN bit;
         current_mem   : OUT bit_vector( 7 DOWNTO 0)
         );
  END COMPONENT;
  SIGNAL pc : integer range 0 to 15;
  SIGNAL rs : bit_vector(3 DOWNTO 0);
  SIGNAL rd : bit_vector(3 DOWNTO 0);
  SIGNAL r : bit_4_array(3 DOWNTO 0);
  SIGNAL halt : bit := '1';
  SIGNAL current_mem : bit_vector(7 DOWNTO 0);

BEGIN  --  process_behave

  u1: simple_memory
  PORT MAP(instr => instr,
           pc => pc,
           load => load,
           clock => clock,
           current_mem => current_mem);

  output_pc:
  PROCESS(pc)
    BEGIN
    pc_out <= itobv(pc, 4);
  END PROCESS;

  main:
  PROCESS
    VARIABLE state : integer range 0 to 6 := 0;
    VARIABLE reg_num : integer range 0 to 3;
    VARIABLE ci : bit;
    VARIABLE reg1 : integer range 0 to 3;
    VARIABLE reg2 : integer range 0 to 3;
```

```
VARIABLE temp_ci : integer range 0 to 1;
VARIABLE temp_i1 : integer;
VARIABLE temp_i2 : integer;
VARIABLE temp_out : integer;
VARIABLE status : bit_vector(3 DOWNTO 0);
VARIABLE output : bit_vector(3 DOWNTO 0);
VARIABLE final : bit_vector(3 DOWNTO 0);
VARIABLE v : bit;
VARIABLE v2 : bit;
VARIABLE inter : integer;

BEGIN
  WAIT UNTIL clock'EVENT and clock = '1';
  IF (inc_pc = '1') THEN
    pc <= pc + 1;
  END IF;

  IF (start = '1') THEN
    halt <= '0';
  END IF;

  IF (reset = '1') THEN
    pc <= 0;
    halt <= '1';
  END IF;

  out_reg <= rd;
  IF (halt = '0') THEN
  CASE state IS
    WHEN 0 =>                          --  decode instruction
      IF (current_mem(7 DOWNTO 4) = "1100") THEN
        state := 1;
      ELSIF (current_mem(7 DOWNTO 6) = "11") THEN
        state := 3;
        CASE current_mem(5 DOWNTO 4) IS
        WHEN "00" =>
          ci := '0';
        WHEN "01" =>
          ci := '1';
        WHEN "10" =>
          ci := carry;
        WHEN "11" =>
          ci := NOT(carry);
      END CASE;                        --  current_mem(5 DOWNTO 4)
      ELSE
        state := 4;
        CASE current_mem(5 DOWNTO 4) IS
        WHEN "00" =>
          ci := '0';
        WHEN "01" =>
          ci := '1';
        WHEN "10" =>
          ci := carry;
        WHEN "11" =>
          ci := NOT(carry);
      END CASE;                        --  current_mem(5 DOWNTO 4)
      END IF;
    WHEN 1 =>                          --  halt & i/o
      IF (current_mem(3) = '1') THEN
        rd <= input;
```

```
  END IF;
  state := 2;
WHEN 2 =>
  IF (current_mem(3) = '1') THEN
    reg_num := bvtoi(current_mem(1 DOWNTO 0));
    r(reg_num) <= rd;
  END IF;
  IF (current_mem(2) = '1') THEN
    rd <= r(bvtoi(current_mem(1 DOWNTO 0)));
    halt <= '1';
  END IF;
  IF (pc = 15) THEN
    pc  <= 0;
  ELSE
    pc <= pc + 1;
  END IF;
  state := 0;
WHEN 3 =>
  IF (ci = '1') THEN
    pc <= bvtoi(current_mem(3 DOWNTO 0));
  ELSE
    IF (pc = 15) THEN
      pc <= 0;
    ELSE
      pc <= pc + 1;
    END IF;
  END IF;
  state := 6;
WHEN 4 =>
  reg1 := bvtoi(current_mem(3 DOWNTO 2));--  (s)
  reg2 := bvtoi(current_mem(1 DOWNTO 0));--  (d)
  rs <= r(reg1);
  rd <= r(reg2);
  state := 5;
WHEN 5 =>
  IF (ci = '0') THEN
    temp_ci := 0;
  ELSE
    temp_ci := 1;
  END IF;
  temp_i1 := sbvtoi(rs);
  CASE current_mem(7 DOWNTO 6) IS
    WHEN "00" =>
      temp_out := temp_i1 + temp_ci;
      IF (temp_out < -16 OR temp_out > 15) THEN
        carry <= '1';
      ELSIF ((temp_out >= 0) AND (temp_i1 < temp_ci)) THEN
        carry <= '1';
      ELSE
        carry <= '0';
      END IF;
    WHEN "01" =>
      temp_i2 := sbvtoi(rd);
      temp_out := (temp_i1 + temp_i2) + temp_ci;
      IF (temp_out < -16 OR temp_out > 15) THEN
        carry <= '1';
      ELSIF ((temp_out >= 0) AND ((temp_i1 < temp_i2) OR
        (temp_i1 < temp_ci) OR temp_i2 < temp_ci)) THEN
        carry <= '1';
      ELSE
```

```
              carry <= '0';
            END IF;
          WHEN "10" =>
            IF (temp_ci = 0) THEN
              temp_ci := 1;
            ELSE
              temp_ci := 0;
            END IF;
            temp_i2 := sbvtoi(rd);
            inter := temp_i2 - temp_i1;
            temp_out := inter - temp_ci;
            IF ((temp_i2 < temp_i1) OR (inter < temp_ci)) THEN
              carry <= '1';
            ELSE
              carry <= '0';
            END IF;
          WHEN OTHERS =>
        END CASE;                        --  current_mem(7 DOWNTO 6)
        r(reg2) <= itobv(temp_out, 4);
        IF (pc = 15) THEN
          pc  <= 0;
        ELSE
          pc <= pc + 1;
        END IF;
        state := 0;
      WHEN 6 =>
        state := 0;
    END CASE;                            --  state
    END IF;
  END PROCESS;
END process_behave;
```

Listing 13. Simple Computer Architecture

3.2.4 Simple Computer Configuration

The simple computer configuration binds the memory structure to the architecture and entity of the simple computer.

```
CONFIGURATION config_processor OF processor IS
  FOR process_behave
    FOR u1: simple_memory
      USE ENTITY work.simple_memory;
    END FOR;
  END FOR;
END config_processor;
```

Listing 14. Simple Computer Configuration

3.2.5 Simple Computer Test bench

The Simple computer assignment defines a sample program to multiply two unsigned four bit numbers. The program is included below, and the results are included in appendix O.

```
-------------------------------------------------------------------------
-------
--  Simple processor from EE126
--  By: Frank Bruno
--  for Professor Chang
-------------------------------------------------------------------------
-------

USE work.alu_pkg.ALL;
USE work.bv_arithmetic.ALL;

ENTITY processor_test IS

END processor_test;

ARCHITECTURE process_test OF processor_test IS
  COMPONENT processor
  PORT(carry           : BUFFER bit;
       out_reg         : BUFFER bit_vector(3 DOWNTO 0);
       pc_out          : BUFFER bit_vector(3 DOWNTO 0);
       input           : IN bit_vector(3 DOWNTO 0);
       clock           : IN bit;
       reset           : IN bit;
       inc_pc          : IN bit;
       instr           : IN bit_vector(7 DOWNTO 0);
       load            : IN bit;
       start           : IN bit
       );
  END COMPONENT;
  SIGNAL carry : bit;
  SIGNAL out_reg : bit_vector(3 DOWNTO 0);
  SIGNAL pc_out : bit_vector(3 DOWNTO 0);
  SIGNAL input : bit_vector(3 DOWNTO 0);
  SIGNAL clock : bit;
  SIGNAL reset : bit;
  SIGNAL inc_pc : bit;
  SIGNAL instr : bit_vector(7 DOWNTO 0);
  SIGNAL load : bit;
  SIGNAL start : bit;

BEGIN  --  process_test
  u1: processor
  PORT MAP(carry => carry,
           out_reg => out_reg,
           pc_out => pc_out,
           input => input,
           clock => clock,
           reset => reset,
```

```
            inc_pc => inc_pc,
            instr => instr,
            load => load,
            start => start);

clock_gen:
PROCESS
  BEGIN
    clock <= '0' AFTER 40 ns;
    WAIT FOR 40 ns;
    clock <= '1' AFTER 40 ns;
    WAIT FOR 40 ns;
END PROCESS;

  PROCESS
    VARIABLE t_state : integer := 0;

    BEGIN
      WAIT until clock'event and clock = '1';
      CASE t_state IS
        WHEN 0 =>                          --  set up for a load
          reset <= '1';
          inc_pc <= '0';
          load <= '0';
          start <= '0';
        WHEN 1 =>                          --  start loading
          reset <= '0';
          load <= '1';
          inc_pc <= '1';
          instr <= "11001101";
        WHEN 2 =>
          instr <= "10010000";
        WHEN 3 =>
          instr <= "10000100";
        WHEN 4 =>
          instr <= "11001001";
        WHEN 5 =>
          instr <= "10011010";
        WHEN 6 =>
          instr <= "10011111";
        WHEN 7 =>
          instr <= "00010000";
        WHEN 8 =>
          instr <= "11101011";
        WHEN 9 =>
          instr <= "01000110";
        WHEN 10 =>
          instr <= "00101111";
        WHEN 11 =>
          instr <= "11010110";
        WHEN 12 =>
          instr <= "11000111";
        WHEN 13 =>
          instr <= "11000110";
        WHEN 14 =>
          instr <= "11010000";
        WHEN 15 =>
          load <= '0';
          inc_pc <= '0';
          reset <= '1';
```

```
        WHEN 16 =>
          reset <= '0';
        WHEN 17 =>                          --  begin
          start <= '1';
          input <= "0010";
        WHEN 18 =>
          start <= '0';
        WHEN 23 =>
          start  <= '1';
          input <= "0111";
        WHEN 24 =>
          start <= '0';
        WHEN 220 =>
          start <= '1';
        WHEN others =>
          start <= '0';
      END CASE;                             --  state
      t_state := t_state + 1;
  END PROCESS;
END process_test;

CONFIGURATION process_test_config OF processor_test IS
  FOR process_test
    FOR u1: processor
      USE ENTITY work.processor;
    END FOR;                               --
  END FOR;                                 --
END process_test_config;
```

Listing 15. Simple Computer Multiplication Testbench

CHAPTER 4. THE BASIC COMPUTER

4.0 The Basic Computer Definition

The Basic Computer is the first real example of a general purpose digital system. The basic computer is capable of executing programs stored in its memory. The basic computer has 4k of 16 bit memory, so quite complex programs can be developed. The following sections give a brief overview of the basic computer. For more detailed information, please refer to chapter 5 in Computer System Architecture, M. Morris Mano, Prentice Hall.

4.1 Basic Computer Instructions

The basic computer instruction set is made up of three types of instructions: memory reference instructions, register reference instructions, and input/output instructions.

4.1.1 Memory Reference Instructions

A memory reference instruction uses 1 bit to determine whether the instruction is direct or indirect addressing mode, 3 bits to determine the operation, and 12 bits to determine the address. The format for a memory reference instruction appears in Table 8.

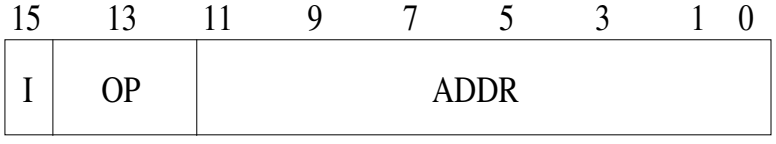

Table 8. Memory Reference Instruction Format

The seven memory reference instructions are listed below in Tables 9 and 10.

Symbol	Binary Code(15:12)	Description
AND	0000	AND memory word to AC
ADD	0001	ADD memory word to AC
LDA	0010	Load AC from memory
STA	0011	Store AC into memory
BUN	0100	Branch unconditionally
BSA	0101	Branch and save return address
ISZ	0110	Increment and skip if zero

Table 9. Direct Memory Reference Instructions

Symbol	Binary Code(15:12)	Description
AND	1000	AND memory word to AC
ADD	1001	ADD memory word to AC
LDA	1010	Load AC from memory
STA	1011	Store AC into memory
BUN	1100	Branch unconditionally
BSA	1101	Branch and save return address
ISZ	1110	Increment and skip if zero

Table 10. Indirect Memory Reference Instructions

4.1.2 Register Reference Instructions

Register reference instructions use all 16 bits to define an operation. Bits 15 DOWNTO 12 are always set to 0111. Table 11 lists the bit fields for the register reference instructions and Table 12 lists the actual instructions.

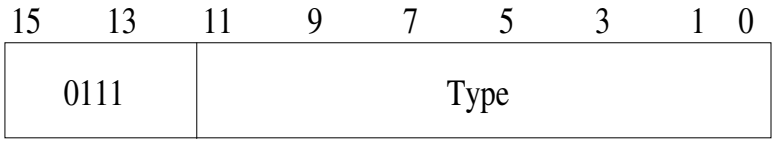

Table 11. Register Reference Instruction Format

Symbol	Binary Code(11:00)	Description
CLA	100000000000	Clear AC
CLE	010000000000	Clear E
CMA	001000000000	Compliment AC
CME	000100000000	Compliment E
CIR		