"I will neither give nor receive unauthorized assistance on this TEST. I will use only one computing device to perform this TEST. I will not use cell while performing this test."

- Anthony Ramos

CSC342/43

Midterm Lab

Spring 2021

Anthony Ramos

Contents

Assignment 1

Objective: Utilize Intel Tutorial on RAM design using LPMs to design and simulate a Data memory module, instruction memory module, and a dual ported register file.

Data Memory Module (W = 32, D =16)

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Ramos_DataMemory is
   generic (K:integer:=32; -- Number of bits per word
        A:integer:=5); -- Number of address bits
   port(
      clock: in std_logic;
      wren : in std_logic :='0'; -- Set to 0 to initially read from memory :: Set 1 to initially write to memory
           data: in std_logic_vector(K-1 downto 0);
      address: in std_logic_vector(A-1 downto 0);
      q : out std_logic_vector(K-1 downto 0));
end Ramos_DataMemory;

architecture arch of Ramos_DataMemory is
   type mem is array(0 to 15) of std_logic_vector(K-1 downto 0);
   signal mem_array: mem;

   begin
   process(clock)
   begin
      if(clock'event and clock = '1') then
    if (wren = '1') then -- Write TO memory
       mem_array(to_integer(unsigned(address))) <= data;
    elsif (wren = '0') then -- Read FROM memory
       mem_array <= ( 0 => "00001110000001000000100011100000",
                      1 => "00010000000000001100000100010011",
                      2 => "00100000000011000000000000000010",
                      3 => "00000100000000000000010000000011",
                      4 => "00000000000011100000000000010000100",
                      5 => "00000011000100000001000010000100",
                      6 => "00110010000100000000000010000111",
                      7 => "00100000011111101110001111000111",
                      8 => "00111000000011000011111111110010",
                      9 => "01111111111111111111111111111111",
                 others =>"00000000000000000000000000000000");
       q <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
    end if;
       end if;
       q <= mem_array(to_integer(unsigned(address)));
    end process;

end arch;
```

Figure 1a: Data Memory Module VHDL Code

The data memory module as shown above {}. Changes in the output ‘q’ occur only at the rising edge of the clock (CLK) signal. In terms of reading and writing memory, if the *Write-Enable* WREN = 0, we will read some value from memory based on a 5-bit address. Above, shows a list of 10 memory locations which could alternatively been defined separately in a *Memory Initialization File (MIF)*. However, if WREN = ‘1’, we will write some value (stored in data) to a memory address.

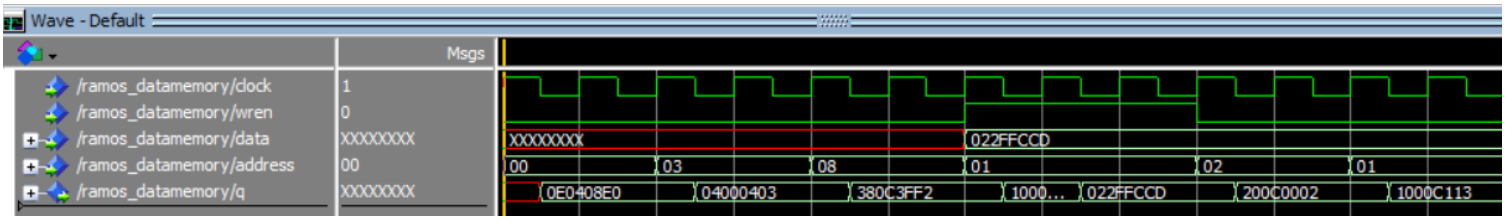

Figure 1b: Data Memory Model ModelSim Simulation

In the simulation above, we initialized WREN to 0 which tells the memory module that we will be reading memory values. First, we read the memory value stored at address 0x00, this data value shown at the output q. We read a new memory value every two clock cycles. It should be noted that initially, no data is being defined although we could have. At some point of the simulation however, we define some data with value 0x022FFCCD. We wish to write this value at memory address 0x01 and so we must set WREN = 1. The output q reflects this change at the next available clock cycle and so q now holds the value 0x022FFCCD at address 0x01. We later set WREN = 0 again and continue reading memory values. Notice that although the value 0x022FFCCD is still in the data input, it is not being written anywhere so long as WREN is 0. The memory values themselves are of no real significance, but rather are used to demonstrate the reading and writing of them to and from memory.

Instruction Memory Module

The instruction memory module is very similar to the module previously implemented. However, the values used do in fact have significance. They represent 32-bit MIPS instructions. Figure 2 below shows Text Segment window of a MIPS assembly program and MIPS instructions along with their corresponding machine code in hexadecimal.

Text Segment

Bkpt	Address	Code	Basic	
	0x00400000	0x3c011001	lui $1,0x00001001	8: lw $s0, a
	0x00400004	0x8c300000	lw $16,0x00000000($1)	
	0x00400008	0x3c011001	lui $1,0x00001001	9: lw $s1, b
	0x0040000c	0x8c310004	lw $17,0x00000004($1)	
	0x00400010	0x3c011001	lui $1,0x00001001	10: lw $s2, c
	0x00400014	0x8c320008	lw $18,0x00000008($1)	
	0x00400018	0x3c011001	lui $1,0x00001001	11: lw $s3, d
	0x0040001c	0x8c33000c	lw $19,0x0000000c($1)	
	0x00400020	0x16130002	bne $16,$19,0x00000002	12: bne $s0, $s3, Else
	0x00400024	0x02328020	add $16,$17,$18	14: add $s0, $s1, $s2
	0x00400028	0x0810000c	j 0x00400030	15: j Exit
	0x0040002c	0x02328022	sub $16,$17,$18	18: sub $s0, $s1, $s2
	0x00400030	0x3c011001	lui $1,0x00001001	20: sw $s0, a
	0x00400034	0xac300000	sw $16,0x00000000($1)	

Figure 2: Text Segment window showing MIPS instruction and machine code

The following VHDL file simply replaces the memory values with the 5 MIPS instructions values (in binary). Also, note that the *DEPTH* of the module is changed to 31. For convenience, the chosen MIPS instructions are given in table 1 below.

MIPS Instruction	32-Bit Representation
lui	0x3C011001
j	0x0810000C
add	0x02328020
sub	0x02328022
bne	0x16130002

Table 1: MIPS instructions with corresponding machine code

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Ramos_InstructionMemory is
   generic (K:integer:=32; -- Number of bits per word
       A:integer:=5); -- Number of address bits
   port(
      clock: in std_logic;
      wren : in std_logic :='0'; -- Set to 0 to initially read from memory :: Set 1 to initially write to memory
           data: in std_logic_vector(K-1 downto 0);
      address: in std_logic_vector(A-1 downto 0);
      q : out std_logic_vector(K-1 downto 0));
end Ramos_InstructionMemory;

architecture arch of Ramos_InstructionMemory is
   type mem is array(0 to 31) of std_logic_vector(K-1 downto 0);
   signal mem_array: mem;

   begin
   process(clock)
   begin
      if(clock'event and clock = '1') then
    if (wren = '1') then -- Write TO memory
       mem_array(to_integer(unsigned(address))) <= data;
    elsif (wren = '0') then -- Read FROM memory
       mem_array <= ( 0 => "00111100000000010001000000000001", -- lui MIPS Instruction
                      1 => "00001000000100000000000000001100", -- j MIPS Instruction
                      2 => "00000010001100101000000000100000", -- add MIPS Instruction
                      3 => "00000010001100101000000000100010", -- sub MIPS Instruction
                      4 => "00010110000100110000000000000010", -- bne MIPS Instruction
                 others =>"00000000000000000000000000000000");
       q <= "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX";
    end if;
       end if;
       q <= mem_array(to_integer(unsigned(address)));
    end process;


end arch;
```

Figure 3a: Instruction Memory Module VHDL Code

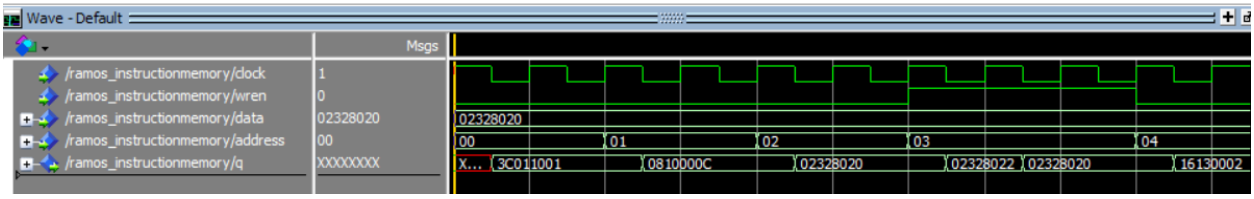

Figure 1b: Data Memory Model ModelSim Simulation

The simulation above, is very similar to the first simulation. We initialized WREN to 0 which tells the memory module that we will be reading memory values. We read a new memory value about every two clock cycles First, we read the memory value stored at address 0x00, which is the MIPS instruction lui according to our table. We then read the memory values at address 0x01 and 0x02 which read the MIPS instructions j and add respectively and output them to q. We initially defined some data 0x02328020 (add instruction) but have not written it anywhere since WREN = 0. Now, we wish to write this value at memory address 0x03 and so we must set

WREN = 1. The output q reflects this change at the next available clock cycle. We set WREN = 0 again and continue reading the last memory value at address 0x4 (the MIPS sub instruction).

Dual Ported Register File

The dual ported register file consists of one write address, two individual read addresses, and two outputs q1 and q2. The functionality depends on three parameters: CLOCK, RST, and WREN. To read data, both WREN and RST must be set to 0. Unlike the previous design, we have two addresses in which we can read data from. We could choose to read two different memory values simultaneously or even choose to read the same memory value and display it at both outputs. However, if we wish to write some data, then RST = WREN = 1 with a HIGH clock signal. The VHDL code for this device is given below.

```
Library IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;

ENTITY Ramos_RegisterFile is
   generic (K:integer:=32; -- Number of bits per word
        A:integer:=5); -- Number of address bits
   port(
      clock: in std_logic;
      wren : in std_logic;
      rst  : in std_logic;
      data : in std_logic_vector(K-1 downto 0);
      waddress  : in std_logic_vector(A-1 downto 0);
      raddress1 : in std_logic_vector(A-1 downto 0);
      raddress2 : in std_logic_vector(A-1 downto 0);
      q1 : out std_logic_vector(K-1 downto 0);
      q2 : out std_logic_vector(K-1 downto 0));
end Ramos_RegisterFile;

architecture arch of Ramos_RegisterFile is
   type reg is array(0 to 31) of std_logic_vector(K-1 downto 0);
   signal reg_array: reg := ( --Initialize some data
      0 => "00111100000000010001000000000001",
      1 => "00000110001111111000010000100011",
      2 => "11111111111111111110000000000000",
      others => "00000000000000000000000000000000");

   begin

   process(rst, clock)
   begin
    if (rst = '1') then
      reg_array <= (others => (others => '0'));
    elsif (clock'event and clock = '1') then
      if (wren = '1') then
         reg_array(to_integer(unsigned(waddress))) <= data;
      end if;
    end if;
       q1 <= reg_array(to_integer(unsigned(raddress1)));
       q2 <= reg_array(to_integer(unsigned(raddress2)));
    end process;

end arch;
```

Figure 4a: Register File VHDL Code

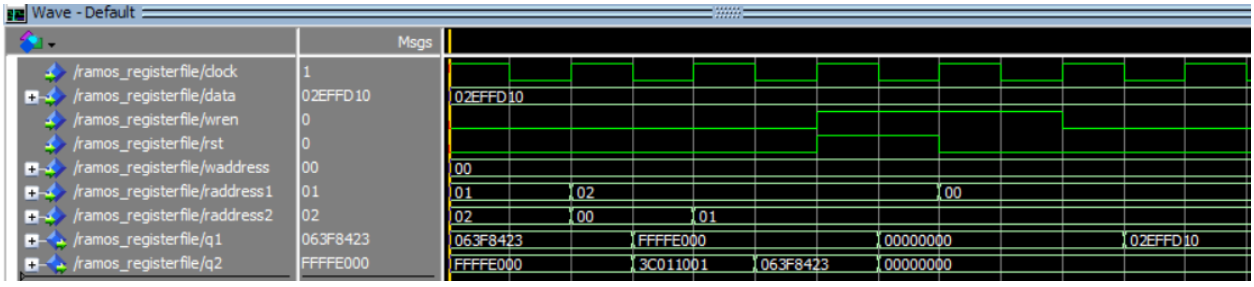

Figure 4b: Register File ModelSim Simulation

In the simulation above, we begin by defining some arbitrary data 0x02EFFD10 and setting the write address to 0x00. However, because RST = WREN = 0, we cannot write this data yet. For

now, we shall read the memory values stored at addresses 0x01 and 0x02 for the first and second read addresses respectively. We will then display those values at outputs q1 and q2. We then switch the read addresses such that q1 now contains the previous value of q2 and q2 now contains the previous value of q1. We read some additional memory values until we decide to set RST = WREN = 1 which now allows us to write to the desired memory address (i.e. 0x00). Note that, while RST = WREN = 1, the memory values at all addresses are set to 0 rather than having them be undefined. Upon setting RST = WREN = 0, we can now see that the value 0x02EFFD10 has been written at address 0x00 on output q1.

Assignment 2

Objective: Design a 32-Bit ADD/SUB unit both from starch and using LPM modules as described in figures 5a and 5b utilizing “Using Library Modules in VHDL Design” guide by Intel.

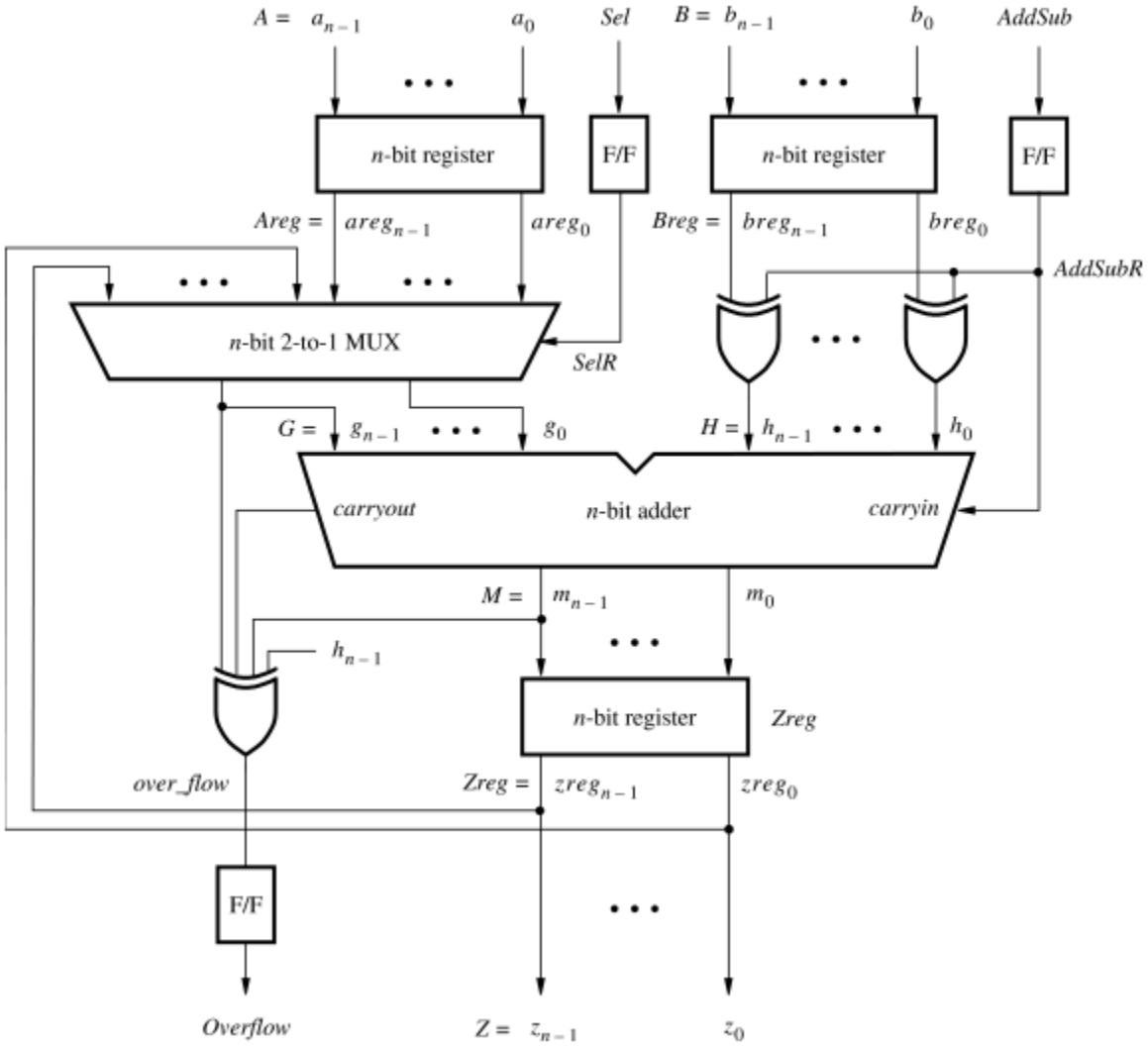

Figure 5a: Adder Subtractor Unit (from scratch)

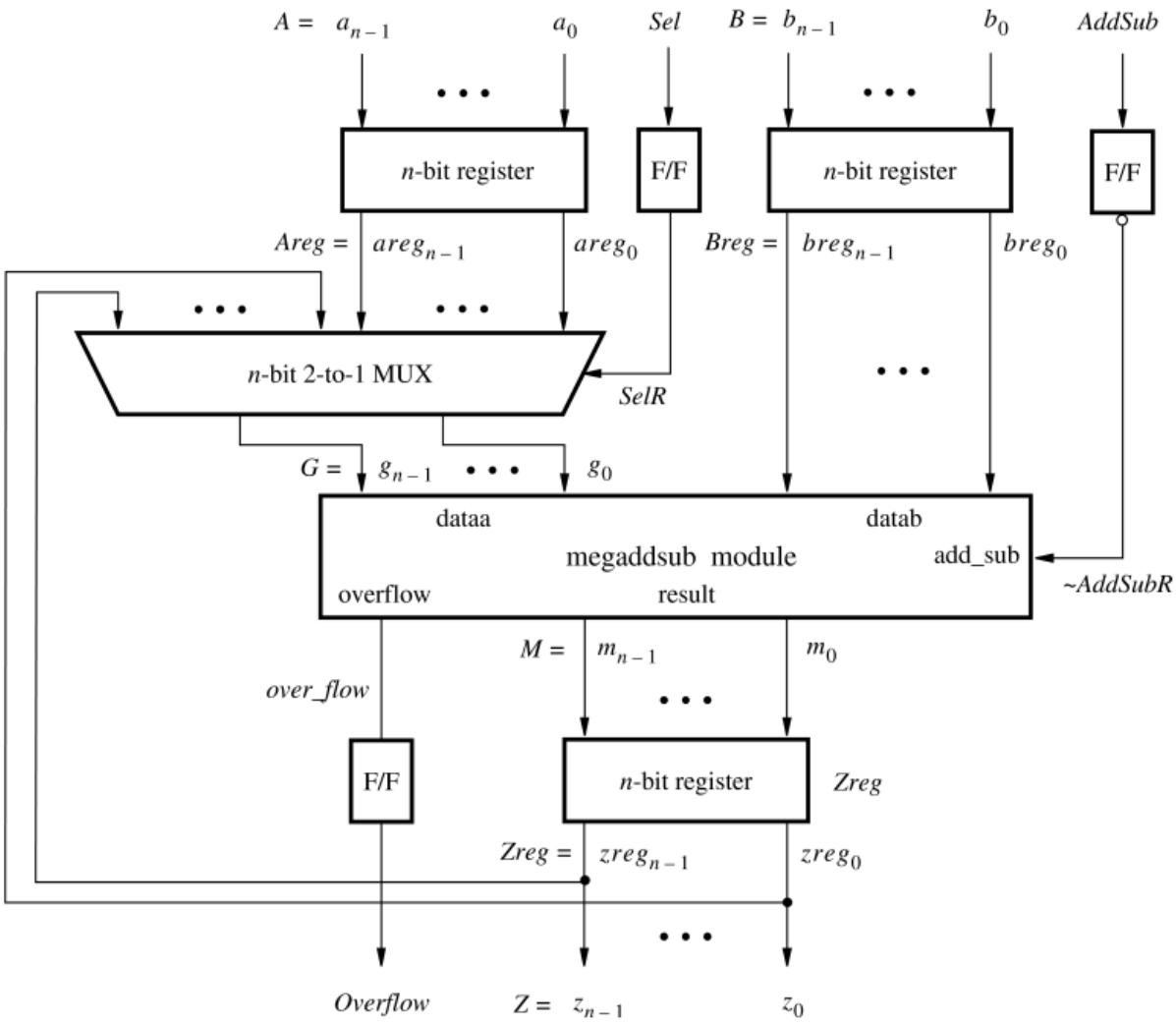

Figure 5b: Adder Subtractor Unit (using LPMs)

32-Bit Adder (from scratch)
Components needed:

- 3x 32-Bit Registers (Incorporated in adder/subtractor unit VHDL file)
- 3x F/F (Incorporated in adder/subtractor unit VHDL file)
- 3x XOR gates (Incorporated in adder/subtractor unit VHDL file)
- 1x 32-Bit 2-to-1 Mux
- 1x 32 Bit Adder/Subtractor

```
-- 32-bit 2-to-1 multiplexer
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY Ramos_mux2to1 IS
    GENERIC ( k : INTEGER := 32 ) ;
    PORT ( V, W : IN STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
           Selm : IN STD_LOGIC ;
              F : OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0) ) ;
    END Ramos_mux2to1 ;

ARCHITECTURE Behavior OF Ramos_mux2to1 IS
BEGIN
    PROCESS ( V, W, Selm )
    BEGIN
       IF Selm = '0' THEN
          F <= V ;
       ELSE
          F <= W ;
       END IF ;
    END PROCESS ;
END Behavior ;
```

Figure: 32-Bit 2to1 Multiplexer VHDL code

```
--32-Bit Adder
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_signed.all ;

ENTITY Ramos_adderk IS
    GENERIC ( k : INTEGER := 32 ) ;
    PORT ( carryin : IN STD_LOGIC ;
                X, Y : IN STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
                   S : OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
           carryout : OUT STD_LOGIC ) ;
    END Ramos_adderk ;

ARCHITECTURE Behavior OF Ramos_adderk IS
    SIGNAL Sum : STD_LOGIC_VECTOR(k DOWNTO 0) ;
BEGIN
    Sum <= ('0'& X) + ('0'& Y) + carryin ;
    S <= Sum(k-1 DOWNTO 0) ;
    carryout <= Sum(k) ;
END Behavior ;
```

Figure: 32-Bit Adder VHDL code

```
ENTITY Ramos_addersubtractor IS
   GENERIC ( n : INTEGER := 32 ) ;
   PORT  (A, B : IN STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
         Clock, Reset, Sel, AddSub : IN STD_LOGIC ;
         Z : BUFFER STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
         Overflow : OUT STD_LOGIC ) ;
END Ramos_addersubtractor ;

ARCHITECTURE Behavior OF Ramos_addersubtractor IS
   SIGNAL G, H, M, Areg, Breg, Zreg, AddSubR_n : STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
   SIGNAL SelR, AddSubR, carryout, over_flow : STD_LOGIC ;
   COMPONENT Ramos_mux2to1
      GENERIC ( k : INTEGER := 32 ) ;
      PORT ( V, W : IN STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
             Selm : IN STD_LOGIC ;
                F : OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0) ) ;
      END COMPONENT ;

      COMPONENT Ramos_adderk
         GENERIC ( k : INTEGER := 32 ) ;
         PORT (carryin : IN STD_LOGIC ;
               X, Y : IN STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
               S : OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
               carryout : OUT STD_LOGIC );
      END COMPONENT ;
BEGIN
   PROCESS ( Reset, Clock )
   BEGIN
      IF Reset = '1' THEN
      Areg <= (OTHERS => '0'); Breg <= (OTHERS => '0');
      Zreg <= (OTHERS => '0'); SelR <= '0'; AddSubR <= '0'; Overflow <= '0';
   ELSIF Clock'EVENT AND Clock = '1' THEN
      Areg <= A; Breg <= B; Zreg <= M;
      SelR <= Sel; AddSubR <= AddSub; Overflow <= over_flow;
   END IF ;
   END PROCESS ;
   nbit_adder: Ramos_adderk
      GENERIC MAP ( k => n )
      PORT MAP ( AddSubR, G, H, M, carryout ) ;
   multiplexer: Ramos_mux2to1
      GENERIC MAP ( k => n )
      PORT MAP ( Areg, Z, SelR, G ) ;
   AddSubR_n <= (OTHERS => AddSubR) ;
   H <= Breg XOR AddSubR_n ;
   over_flow <= carryout XOR G(n-1) XOR H(n-1) XOR M(n-1) ;
   Z <= Zreg ;
END Behavior
```

Figure: Full Add/Sub unit VHDL code

32-Bit Adder (using LPMs)

- 3x 32-Bit Registers (Incorporated in adder/subtractor unit VHDL file)
- 3x F/F (Incorporated in adder/subtractor unit VHDL file)
- 1x 32-Bit 2-to-1 Mux (Previously implemented above)
- 1x 32 Bit Adder/Subtractor

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY lpm;
USE lpm.all;
ENTITY Ramos_megaddsub IS
   PORT ( add_sub  : IN STD_LOGIC ;
          dataa    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
          datab    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
          overflow : OUT STD_LOGIC;
          result   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0) );
END Ramos_megaddsub;

ARCHITECTURE SYN OF Ramos_megaddsub IS
   SIGNAL sub_wire0 : STD_LOGIC ;
   SIGNAL sub_wire1 : STD_LOGIC_VECTOR (31 DOWNTO 0);
   COMPONENT lpm_add_sub
   GENERIC ( lpm_direction : STRING;
             lpm_hint : STRING;
             lpm_representation : STRING;
             lpm_type : STRING;
             lpm_width : NATURAL );
   PORT (
      dataa : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      add_sub : IN STD_LOGIC ;
      datab : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      overflow : OUT STD_LOGIC ;
      result : OUT STD_LOGIC_VECTOR (31 DOWNTO 0) );
   END COMPONENT;

BEGIN
   overflow <= sub_wire0;
   result <= sub_wire1(31 DOWNTO 0);
   lpm_add_sub_component : lpm_add_sub
   GENERIC MAP ( lpm_direction => "UNUSED",
      lpm_hint => "ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO",
      lpm_representation => "SIGNED",
      lpm_type => "LPM_ADD_SUB",
      lpm_width => 32 )
   PORT MAP ( dataa => dataa,
      add_sub => add_sub,
      datab => datab,
      overflow => sub_wire0,
      result => sub_wire1 );
END SYN;
```

Figure: megaddsub component VHDL code

```vhdl
-- Top-level entity
ENTITY Ramos_addersubtractor2 IS
    GENERIC ( n : INTEGER := 32 ) ;
    PORT (A, B : IN STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
          Clock, Reset, Sel, AddSub : IN STD_LOGIC ;
          Z : BUFFER STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
          Overflow : OUT STD_LOGIC ) ;
END Ramos_addersubtractor2 ;

ARCHITECTURE Behavior OF Ramos_addersubtractor2 IS
    SIGNAL G, M, Areg, Breg, Zreg : STD_LOGIC_VECTOR(n-1 DOWNTO 0) ;
    SIGNAL SelR, AddSubR, over_flow : STD_LOGIC ;
    COMPONENT mux2to1
       GENERIC ( k : INTEGER := 32 ) ;
       PORT ( V, W : IN STD_LOGIC_VECTOR(k-1 DOWNTO 0) ;
              Selm : IN STD_LOGIC ;
              F : OUT STD_LOGIC_VECTOR(k-1 DOWNTO 0) ) ;
    END COMPONENT ;

    COMPONENT megaddsub
       PORT (add_sub : IN STD_LOGIC ;
             dataa, datab : IN STD_LOGIC_VECTOR(31 DOWNTO 0) ;
             result : OUT STD_LOGIC_VECTOR(31 DOWNTO 0) ;
             overflow : OUT STD_LOGIC ) ;
       END COMPONENT ;
BEGIN
-- Define flip-flops and registers
    PROCESS ( Reset, Clock )
    BEGIN
       IF Reset = '1' THEN
          Areg <= (OTHERS => '0'); Breg <= (OTHERS => '0');
          Zreg <= (OTHERS => '0'); SelR <= '0'; AddSubR <= '0'; Overflow <= '0';
       ELSIF Clock'EVENT AND Clock = '1' THEN
          Areg <= A; Breg <= B; Zreg <= M;
          SelR <= Sel; AddSubR <= NOT AddSub; Overflow <= over_flow;
       END IF ;
    END PROCESS ;

-- Define combinational circuit
    nbit_addsub: Ramos_megaddsub
       PORT MAP ( AddSubR, G, Breg, M, over_flow ) ;
    multiplexer: Ramos_mux2to1
       GENERIC MAP ( k => n )
       PORT MAP ( Areg, Z, SelR, G ) ;
    Z <= Zreg ;
END Behavior;
```

Figure: Full adder/subtractor unit VHDL code