CPE/EE 422/522
Advanced Logic Design
L15

Electrical and Computer Engineering
University of Alabama in Huntsville

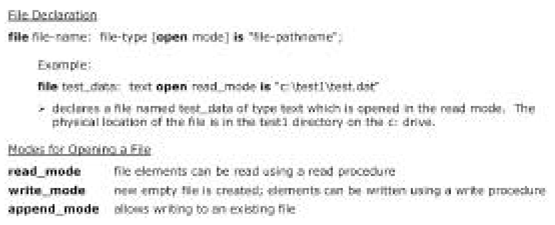
Outline

- VHDL
  - What we know (additional topics)
    - Attributes
    - Transport and Inertial Delays
    - Operator Overloading
    - Multivalued Logic and Signal Resolution
    - IEEE 1164 Standard Logic
    - Generics
    - Generate Statements
    - Synthesis of VHDL Code
    - Synthesis Examples
  - What we don't know
    - Files and Text IO
  - Networks for Arithmetic Operations
  - SM Charts

Files

- File input/output in VHDL
- Used in test benches
  - Source of test data
  - Storage for test results
- VHDL provides a standard TEXTIO package
  - read/write lines of text

Files



File Declaration

file file-name: file-type [**open** mode] **is** "file-pathname";

Example:

file test_data: text **open** read_mode **is** "c:\test1\test.dat"

➢ declares a file named test_data of type text which is opened in the read mode. The physical location of the file is in the test1 directory on the c: drive.

Modes for Opening a File

read_mode	file elements can be read using a read procedure
write_mode	new empty file is created; elements can be written using a write procedure
append_mode	allows writing to an existing file

Standard TEXTIO Package

- Contains declarations and procedures for working with files composed of lines of text
- Defines a file type named text:

```
      type text is file of string;
```

- Contains procedures for reading lines of text from a file of type text and for writing lines of text to a file

Reading TEXTIO file

- Readline reads a line of text and places it in a buffer with an associated pointer
- Pointer to the buffer must be of type line, which is declared in the textio package as:

```
      type line is access string;
```

- When a variable of type line is declared, it creates a pointer to a string
- Code

```
variable buff: line;
...
readline (test_data, buff);
```

  - reads a line of text from test_data and places it in a buffer which is pointed to by buff

Extracting Data from the Line Buffer

- To extract data from the line buffer, call a read procedure one or more times
- For example, if bv4 is a bit_vector of length four, the call

```
read(buff, bv4)
```

  - extracts a 4-bit vector from the buffer, sets bv4 equal to this vector, and adjusts the pointer buff to point to the next character in the buffer. Another call to read will then extract the next data object from the line buffer.

Extracting Data from the Line Buffer (cont'd)

- TEXTIO provides overloaded *read* procedures to read data of types bit, bit_vector, boolean, character, integer, real, string, and time from buffer
- Read forms

```
read(pointer, value)
read(pointer, value, good)
```

  - good is boolean that returns TRUE if the read is successful and FALSE if it is not
  - type and size of value determines which of the read procedures is called
  - character, strings, and bit_vectors within files of type text are not delimited by quotes

Writing to TEXTIO files

- Call one or more write procedures to write data to a line buffer and then call writeline to write the line to a file

```
variable buffw : line;
variable int1 : integer;
variable bv8 : bit_vector(7 downto 0);
...
write(buffw, int1, right, 6); --right just., 6 ch. wide
write(buffw, bv8, right, 10);
writeln(buffw, output_file);
```

- Write parameters: 1) buffer pointer of type line, 2) a value of any acceptable type, 3) justification (left or right), and 4) field width (number of characters)

An Example

- Procedure to read data from a file and store the data in a memory array
- Format of the data in the file
  - `address N comments`
    `byte1 byte2 ... byteN comments`
    - address – 4 hex digits
    - N – indicates the number of bytes of code
    - bytei - 2 hex digits
    - each byte is separated by one space
    - the last byte must be followed by a space
    - anything following the last state will not be read and will be treated as a comment

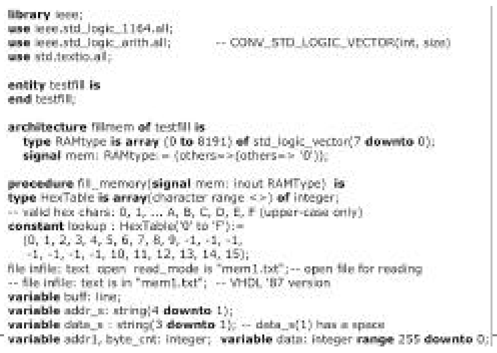
An Example (cont'd)

- Code sequence: an example

```
– 12AC 7 (7 hex bytes follow)
  AE 03 B6 91 C7 00 0C (LDX imm, LDA dir, STA ext)
  005B 2 (2 bytes follow)
  01 FC_
```

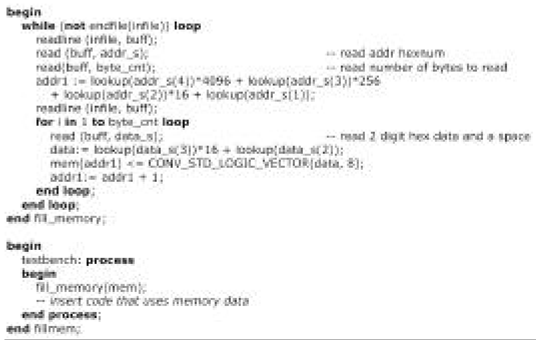
- TEXTIO does not include read procedure for hex numbers
  - we will read each hex value as a string of characters and then convert the string to an integer
- How to implement conversion?
  - table lookup – constant named lookup is an array of integers indexed by characters in the range '0' to 'F'
  - this range includes the 23 ASCII characters: '0', '1', ... '9', ':', ';', '<', '=', '>', '?', '@', 'A', ... 'F'
  - corresponding values: 0, 1, ... 9, -1, -1, -1, -1, -1, -1, -1, 10, 11, 12, 13, 14, 15

VHDL Code to Fill Memory Array



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;          -- CONV_STD_LOGIC_VECTOR(int, size)
use std.textio.all;

entity testfill is
end testfill;

architecture fillmem of testfill is
   type RAMtype is array (0 to 8191) of std_logic_vector(7 downto 0);
   signal mem: RAMtype:= (others=>(others=> '0'));

procedure fill_memory(signal mem: inout RAMType)  is
type HexTable is array(character range <>) of integer;
-- valid hex chars: 0, 1, ... A, B, C, D, E, F (upper-case only)
constant lookup : HexTable('0' to 'F'):=
   (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -1, -1, -1,
    -1, -1, -1, -1, 10, 11, 12, 13, 14, 15);
file infile: text  open  read_mode is "mem1.txt";-- open file for reading
-- file infile: text is in "mem1.txt";  -- VHDL '87 version
variable buff: line;
variable addr_s: string(4 downto 1);
variable data_s : string(3 downto 1); -- data_s(1) has a space
variable addr1, byte_cnt: integer;  variable data: integer range 255 downto 0;
```

VHDL Code to Fill Memory Array (cont'd)



```
begin
   while (not endfile(infile)) loop
      readline (infile, buff);
      read (buff, addr_s);                                   -- read addr hexnum
      read(buff, byte_cnt);                                  -- read number of bytes to read
      addr1 := lookup(addr_s(4))*4096 + lookup(addr_s(3))*256
         + lookup(addr_s(2))*16 + lookup(addr_s(1));
      readline (infile, buff);
      for i in 1 to byte_cnt loop
         read (buff, data_s);                                -- read 2 digit hex data and a space
         data:= lookup(data_s(3))*16 + lookup(data_s(2));
         mem(addr1) <= CONV_STD_LOGIC_VECTOR(data, 8);
         addr1:= addr1 + 1;
      end loop;
   end loop;
end fill_memory;

begin
   testbench: process
   begin
      fill_memory(mem);
      -- insert code that uses memory data
   end process;
end fillmem;
```

Things to Remember

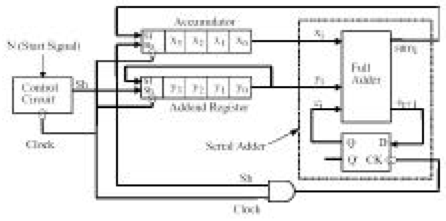
- Attributes associated to signals
  - allow checking for setup, hold times, and other timing specifications
- Attributes associated to arrays
  - allow us to write procedures that do not depend on the manner in which arrays are indexed
- Inertial and transport delays
  - allow modeling of different delay types that occur in real systems
- Operator overloading
  - allow us to extend the definition of VHDL operators so that they can be used with different types of operands

Things to Remember (cont'd)

- Multivalued logic and the associated resolution functions
  - allow us to model tri -state buses, and systems where a signal is driven by more than one source
- Generics
  - allow us to specify parameter values for a component when the component is instantiated
- Generate statements
  - efficient way to describe systems with iterative structure
- TEXTIO
  - convenient way for file input/output

Networks for Arithmetic Operations

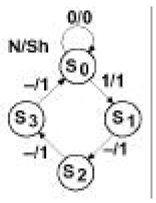
Case Study: Serial Adder with Accumulator



Networks for Arithmetic Operations

Serial Adder with Accumulator

	X	Y	c_i	sum	c_{i+1}
t_0	0101	0111	0	0	1
t_1	0010	1011	1	0	1
t_2	0001	1101	1	1	1
t_3	1000	1110	1	1	0
t_4	1100	0111	0	(1)	(0)



Present State	Next State		Present Output (Sh)	
	N=0	N=1	N=0	N=1
S_0	S_0	S_1	0	1
S_1	S_2	S_2	1	1
S_2	S_3	S_3	1	1
S_3	S_0	S_0	1	1

State Graphs for Control Networks

- Use variable names instead of 0s and 1s
  - E.g., XiXj/ZpZq
    - if Xi and Xj inputs are 1, the outputs Zp and Zq are 1 (all other outputs are 0s)
  - E.g., X = X1X2X3X4, Z = Z1Z2Z3Z4
    - X1X4'/Z2Z3 == 1 - - 0 / 0 1 1 0

Constraints on Input Labels

- Assume: I – input expression => we traverse the arc when I=1

1. If I_i and I_j are any pair of input labels on arcs exiting state S_k, then $I_iI_j = 0$ if $i \neq j$.

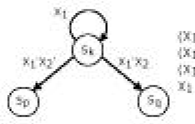
Assures that at most one input label can be 1 at any given time

2. If n arcs exit state S_k and the n arcs have input labels $I_1, I_2, \dots, I_n$, respectively, then $I_1 + I_2 + \dots + I_n = 1$.

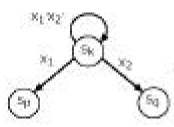
Assures that at least one input label will be 1 at any given time

1 + 2: Exactly one label will be 1 =>
the next state will be uniquely defined for every input combination

Constraints on Input Labels (cont'd)



$(X_1)(X_1'X_2') = 0$
$(X_1)(X_1'X_2) = 0$
$(X_1'X_2')(X_1'X_2) = 0$
$X_1 + X_1'X_2' + X_1'X_2 = 1$

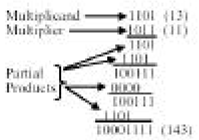


Inputs are X_1 X_2 X_3
($X_1 = X_2 = 1$ not allowed)

	000	001	010	011	100	101	110	111
S_k	S_k	S_k	S_q	S_q	S_p	S_p	–	–

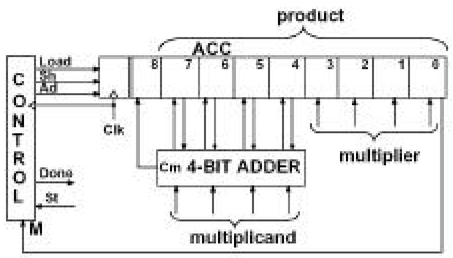
Networks for Arithmetic Operations

Case Study: Serial Parallel Multiplier



Note: we use unsigned binary numbers

Block Diagram of a Binary Multiplier

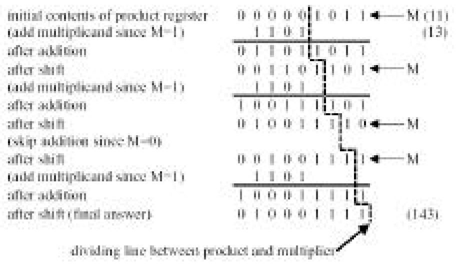


Ad – add signal // adder outputs are stored into the ACC

Sh – shift signal // shift all 9 bits to right

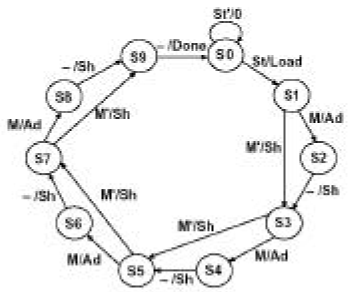
Ld – load signal // load multiplier into the 4 lower bits of the ACC and clear the upper 5 bits

Multiplication Example

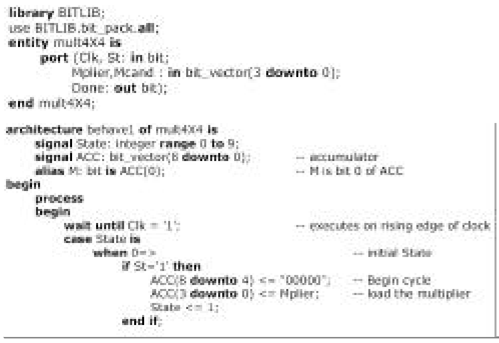


```
initial contents of product register   0 0 0 0 0|1 0 1 1 ←— M (11)
(add multiplicand since M=1)             1 1 0 1              (13)
                                       ___________________
after addition                         0 1 1 0 1|1 0 1 1
after shift                            0 0 1 1 0 1|1 0 1 ←— M
(add multiplicand since M=1)             1 1 0 1
                                       ___________________
after addition                         1 0 0 1 1 1|1 0 1
after shift                            0 1 0 0 1 1 1|1 0 ←— M
(skip addition since M=0)
after shift                            0 0 1 0 0 1 1 1|1 ←— M
(add multiplicand since M=1)             1 1 0 1
                                       ___________________
after addition                         1 0 0 0 1 1 1 1|1
after shift (final answer)             0 1 0 0 0 1 1 1 1|     (143)

        dividing line between product and multiplier
```

State Graph for Binary Multiplier

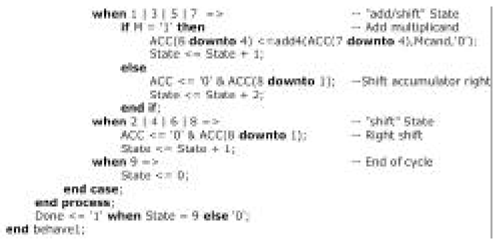


Behavioral VHDL Model



```
library BITLIB;
use BITLIB.bit_pack.all;
entity mult4X4 is
      port (Clk, St: in bit;
            Mplier,Mcand : in bit_vector(3 downto 0);
            Done: out bit);
end mult4X4;

architecture behave1 of mult4X4 is
     signal State: integer range 0 to 9;
     signal ACC: bit_vector(8 downto 0);           -- accumulator
     alias M: bit is ACC(0);                       -- M is bit 0 of ACC
begin
     process
     begin
          wait until Clk = '1';                    -- executes on rising edge of clock
          case State is
               when 0=>                                   -- initial State
                    if St='1' then
                         ACC(8 downto 4) <= "00000";       -- Begin cycle
                         ACC(3 downto 0) <= Mplier;        -- load the multiplier
                         State <= 1;
                    end if;
```

Behavioral VHDL Model (cont'd)

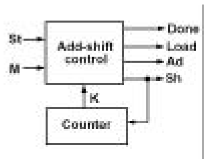


```
               when 1 | 3 | 5 | 7  =>                            -- "add/shift" State
                    if M = '1' then                              -- Add multiplicand
                         ACC(8 downto 4) <=add4(ACC(7 downto 4),Mcand,'0');
                         State <= State + 1;
                    else
                         ACC <= '0' & ACC(8 downto 1);           --Shift accumulator right
                         State <= State + 2;
                    end if;
               when 2 | 4 | 6 | 8 =>                             -- "shift" State
                    ACC <= '0' & ACC(8 downto 1);                -- Right shift
                    State <= State + 1;
               when 9 =>                                         -- End of cycle
                    State <= 0;
          end case;
     end process;
     Done <= '1' when State = 9 else '0';
end behave1;
```

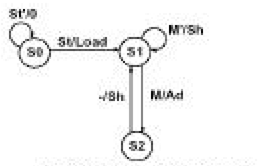
Multiplier Control with Counter

- Current design: control part generates the control signals (shift/add) and counts the number of steps
- If the number of bits is large (e.g., 64), the control network can be divided into a counter and a shift/add control

Multiplier Control with Counter (cont'd)



(a) Multiplier control

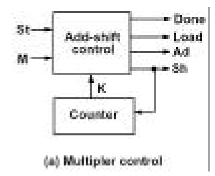


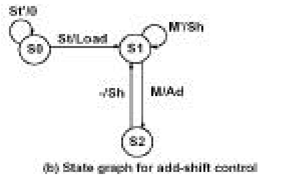
(b) State graph for add-shift control

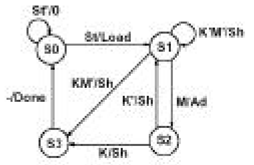
Add-shifts control: tests St and M and generates the proper sequence of add and shift signals

Counter control: counter generates a completion signal K that stops the multiplier after the proper number of shifts have been completed

Multiplier Control with Counter (cont'd)



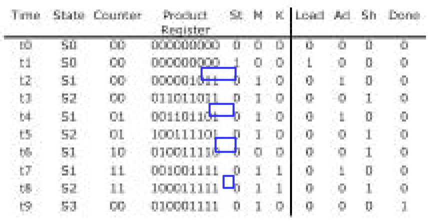




• Increment counter each time a shift signal is generated

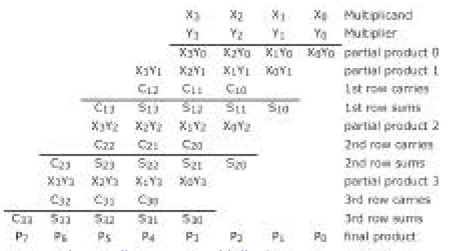
• Generate K after n-1 shifts occured

Operation of a Multiplier Using Counter



Time	State	Counter	Product Register	St	M	K	Load	Ad	Sh	Done
t0	S0	00	000000000	0	0	0	0	0	0	0
t1	S0	00	000000000	1	0	0	1	0	0	0
t2	S1	00	000001011	0	1	0	0	1	0	0
t3	S2	00	011011011	0	1	0	0	0	1	0
t4	S1	01	001101101	0	1	0	0	1	0	0
t5	S2	01	100111101	0	1	0	0	0	1	0
t6	S1	10	010011110	0	0	0	0	0	1	0
t7	S1	11	001001111	0	1	1	0	1	0	0
t8	S2	11	100011111	0	1	1	0	0	1	0
t9	S3	00	010001111	0	1	0	0	0	0	1

Array Multiplier



				X_3	X_2	X_1	X_0	Multiplicand
				Y_3	Y_2	Y_1	Y_0	Multiplier
				X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0	partial product 0
			X_3Y_1	X_2Y_1	X_1Y_1	X_0Y_1		partial product 1
			C_{12}	C_{11}	C_{10}			1st row carries
		C_{13}	S_{13}	S_{12}	S_{11}	S_{10}		1st row sums
		X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2			partial product 2
		C_{22}	C_{21}	C_{20}				2nd row carries
	C_{23}	S_{23}	S_{22}	S_{21}	S_{20}			2nd row sums
	X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3				partial product 3
	C_{32}	C_{31}	C_{30}					3rd row carries
C_{33}	S_{33}	S_{32}	S_{31}	S_{30}				3rd row sums
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0	final product

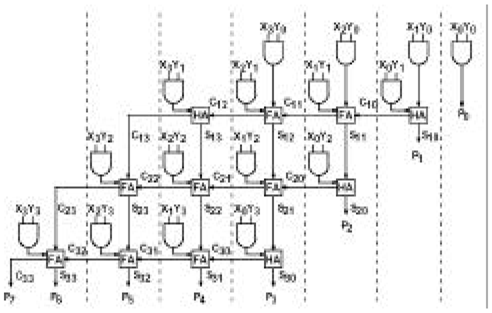
• What do we need to realize Array Multiplier?

• AND gates = ?

• FA = ?

• HA = ?

Array Multiplier (cont'd)



Array Multiplier (cont'd)

- Complexity of the N-bit array multiplier
  - number of AND gates = ?
  - number of HA = ?
  - number of FA = ?
- Delay
  - tg – longest AND gate delay
  - tad – longest possible delay through an adder

Multiplication of Signed Binary Numbers

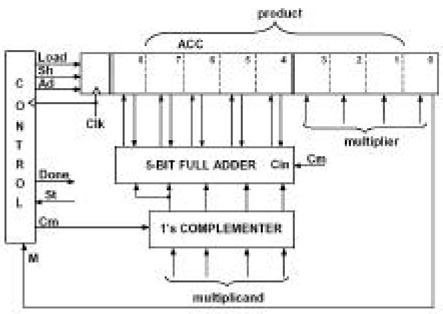
- How to multiply signed binary numbers?
- Procedure
  - Complement the multiplier if negative
  - Complement the multiplicand if negative
  - Multiply two positive binary numbers
  - Complement the product if it should be negative
- Simple but requires more hardware and time than other available methods

Multiplication of Signed Binary Numbers

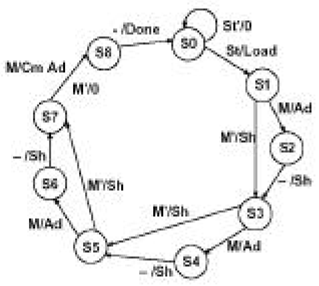
- Four cases
  - Multiplicand is positive, multiplier is positive
  - Multiplicand is negative, multiplier is positive
  - Multiplicand is positive, multiplier is negative
  - Multiplier is negative, multiplicand is negative
- Examples
  - 0111 x 0101 = ?
  - 1101 x 0101 = ?
  - 0101 x 1101 = ?
  - 1011 x 1101 = ?

- Preserve the sign of the partial product at each step
- If multiplier is negative, complement the multiplicand before adding it in at the last step

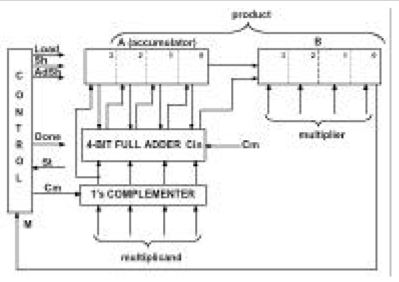
2's Complement Multiplier



State Graph for 2’s Complement Multiplier

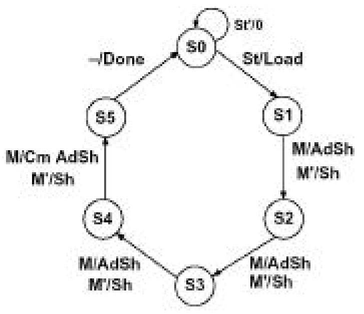


Faster Multiplier

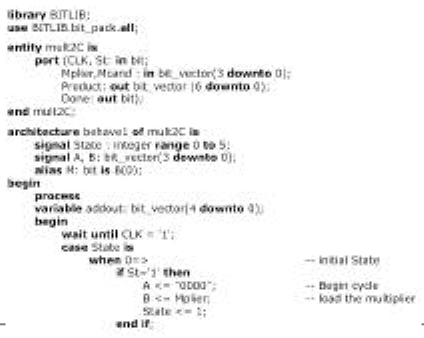


- Move wires from the adder outputs one position to the right => add and shift can occur at the same clock cycle

State Graph for Faster Multiplier

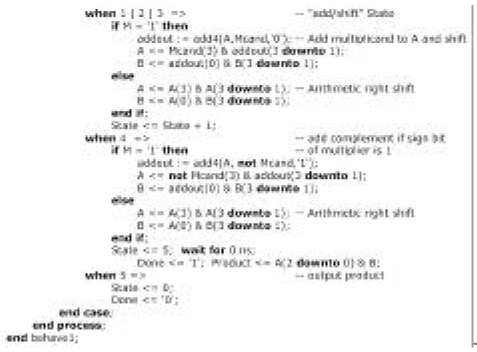


Behavioral Model for Faster Multiplier



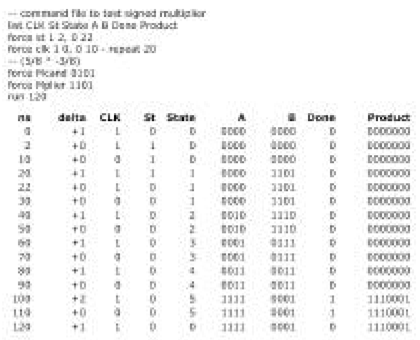
```
library BITLIB;
use BITLIB.bit_pack.all;

entity mult2C is
      port (CLK, St: in bit;
            Mplier,Mcand : in bit_vector(3 downto 0);
            Product: out bit_vector (6 downto 0);
            Done: out bit);
end mult2C;

architecture behave1 of mult2C is
      signal State : integer range 0 to 5;
      signal A, B: bit_vector(3 downto 0);
      alias M: bit is B(0);
begin
      process
      variable addout: bit_vector(4 downto 0);
      begin
            wait until CLK = '1';
            case State is
                  when 0=>                                    -- initial State
                        if St='1' then
                              A <= "0000";                    -- Begin cycle
                              B <= Mplier;                    -- load the multiplier
                              State <= 1;
                        end if;
```

Behavioral Model for Faster Multiplier



```
            when 1 | 2 | 3 =>                         -- "add/shift" State
                if M = '1' then
                    addout := add4(A,Mcand,'0'); -- Add multiplicand to A and shift
                    A <= Mcand(3) & addout(3 downto 1);
                    B <= addout(0) & B(3 downto 1);
                else
                    A <= A(3) & A(3 downto 1); -- Arithmetic right shift
                    B <= A(0) & B(3 downto 1);
                end if;
                State <= State + 1;
            when 4 =>                                 -- add complement if sign bit
                if M = '1' then                       -- of multiplier is 1
                    addout := add4(A, not Mcand,'1');
                    A <= not Mcand(3) & addout(3 downto 1);
                    B <= addout(0) & B(3 downto 1);
                else
                    A <= A(3) & A(3 downto 1); -- Arithmetic right shift
                    B <= A(0) & B(3 downto 1);
                end if;
                State <= 5; wait for 0 ns;
                    Done <= '1'; Product <= A(2 downto 0) & B;
            when 5 =>                                 -- output product
                State <= 0;
                Done <= '0';
        end case;
    end process;
end behave1;
```

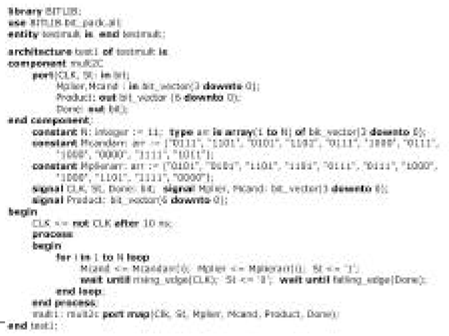
Command File and Simulation



```
-- command file to test signed multiplier
list CLK St State A B Done Product
force st 1 2, 0 22
force clk 1 0, 0 10 - repeat 20
-- (5/8 * -3/8)
force Mcand 0101
force Mplier 1101
run 120
```

```
 ns    delta  CLK   St  State     A      B   Done   Product
  0     +1     1    0     0     0000   0000   0   0000000
  2     +0     1    1     0     0000   0000   0   0000000
 10     +0     0    1     0     0000   0000   0   0000000
 20     +1     1    1     1     0000   1101   0   0000000
 22     +0     1    0     1     0000   1101   0   0000000
 30     +0     0    0     1     0000   1101   0   0000000
 40     +1     1    0     2     0010   1110   0   0000000
 50     +0     0    0     2     0010   1110   0   0000000
 60     +1     1    0     3     0001   0111   0   0000000
 70     +0     0    0     3     0001   0111   0   0000000
 80     +1     1    0     4     0011   0011   0   0000000
 90     +0     0    0     4     0011   0011   0   0000000
100     +2     1    0     5     1111   0001   1   1110001
110     +0     0    0     5     1111   0001   1   1110001
120     +1     1    0     0     1111   0001   0   1110001
```

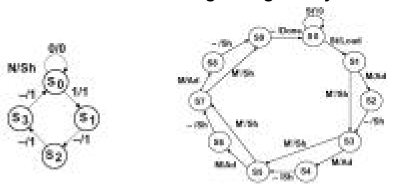
Test Bench for Signed Multiplier



```
library BITLIB;
use BITLIB.bit_pack.all;
entity testmult is end testmult;

architecture test1 of testmult is
component mult2C
    port(CLK, St: in bit;
        Mplier,Mcand : in bit_vector(3 downto 0);
        Product: out bit_vector (6 downto 0);
        Done: out bit);
end component;
    constant N: integer := 11; type arr is array(1 to N) of bit_vector(3 downto 0);
    constant Mcandarr: arr := ("0111", "1101", "0101", "1101", "0111", "1000", "0111",
        "1000", "0000", "1111", "1011");
    constant Mplierarr: arr := ("0101", "0101", "1101", "1101", "0111", "0111", "1000",
        "1000", "1101", "1111", "0000");
    signal CLK, St, Done: bit; signal Mplier, Mcand: bit_vector(3 downto 0);
    signal Product: bit_vector(6 downto 0);
begin
    CLK <= not CLK after 10 ns;
    process
    begin
        for i in 1 to N loop
            Mcand <= Mcandarr(i); Mplier <= Mplierarr(i); St <= '1';
            wait until rising_edge(CLK); St <= '0'; wait until falling_edge(Done);
        end loop;
    end process;
    mult1: mult2c port map(Clk, St, Mplier, Mcand, Product, Done);
end test1;
```

Digital design with SM Charts

- State graphs used to describe
  state machines controlling a digital system

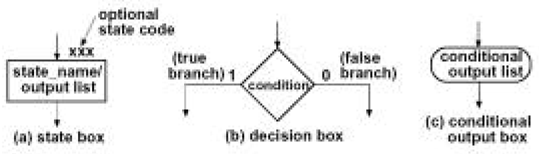


- Alternative: use state machine flowchart

State Machine Charts

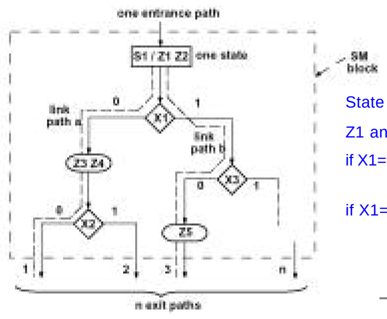
- SM chart or ASM (*Algorithmic State Machine*) chart
- Easier to understand the operation of digital system by examining of the SM chart instead of equivalent state graph
- SM chart leads directly to hardware realization

Components of SM charts



SM Blocks

SM chart is constructed from SM blocks



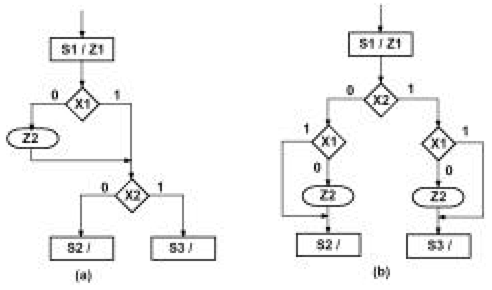
State S1 is entered =>

Z1 and Z2 become 1

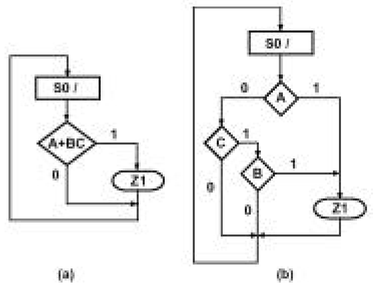
if X1=0
Z3 and Z4 become 1

if X1=1 and X3=0
Z5 become 1

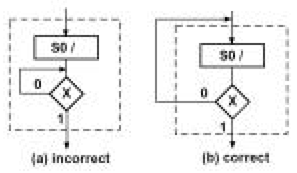
Equivalent SM Blocks



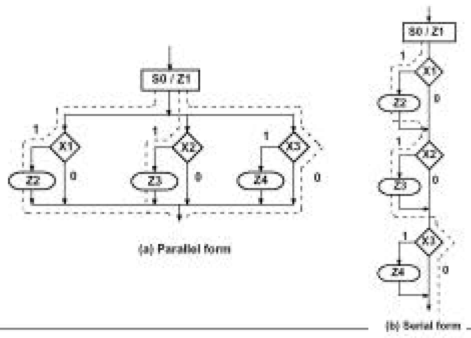
Equivalent SM Charts for Comb Networks



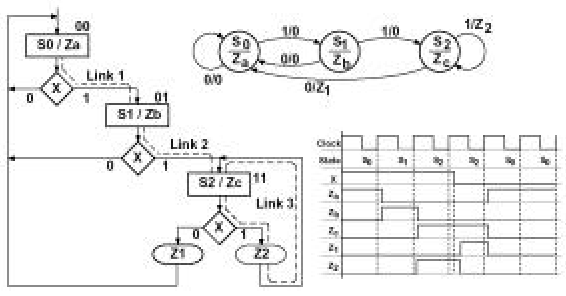
Block with Feedback



Equivalent SM Blocks



Converting a State Graph to an SM Chart


00
S0 / Za
Link 1
X
0
1
01
S1 / Zb
Link 2
X
0
1
11
S2 / Zc
Link 3
X
0
1
Z1
Z2
1/0
1/0
1/Z2
0/0
0/0
0/Z1
Clock
State
X
Za
Zb
Zc
Z1
Z2