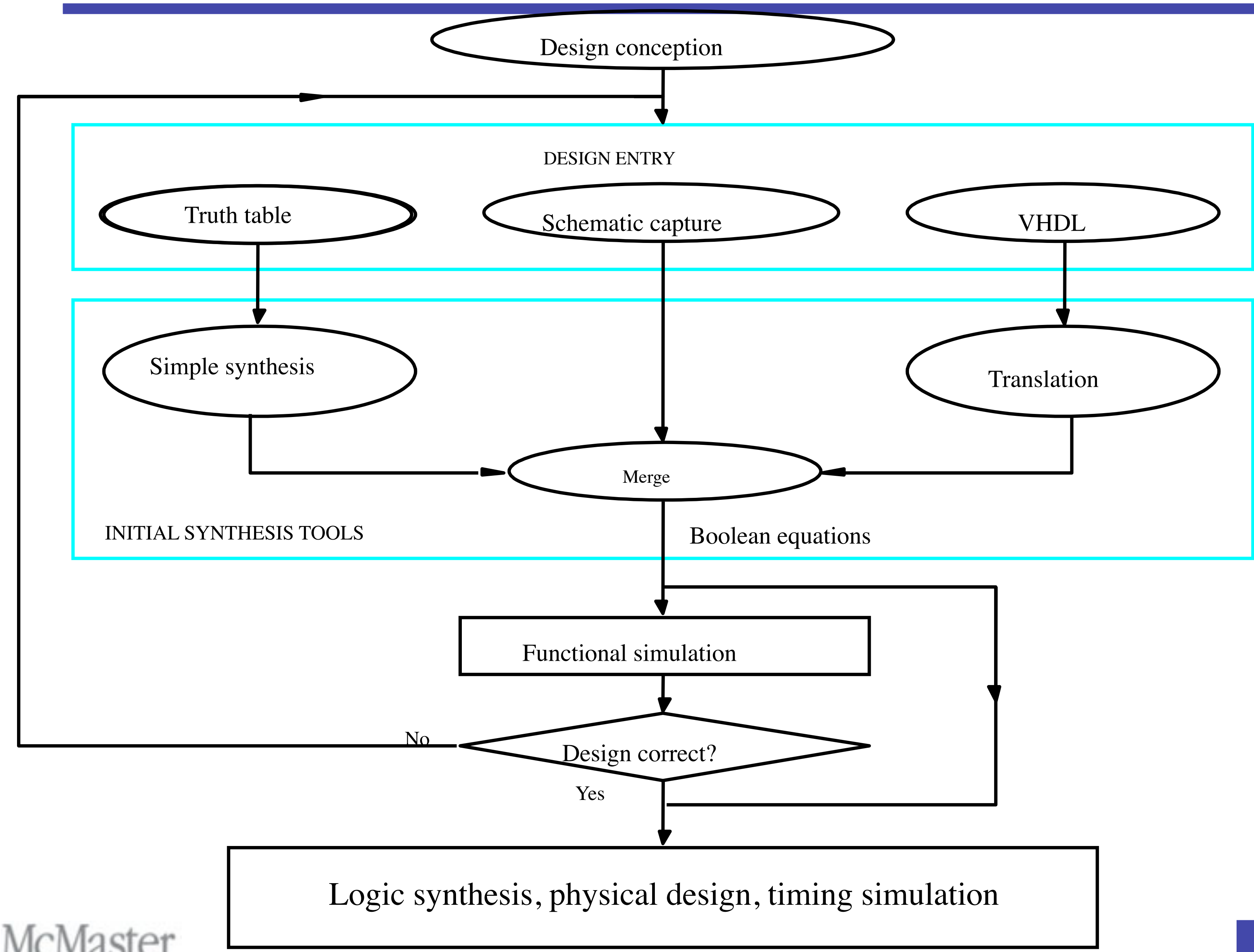
COE2DI4
VHDL

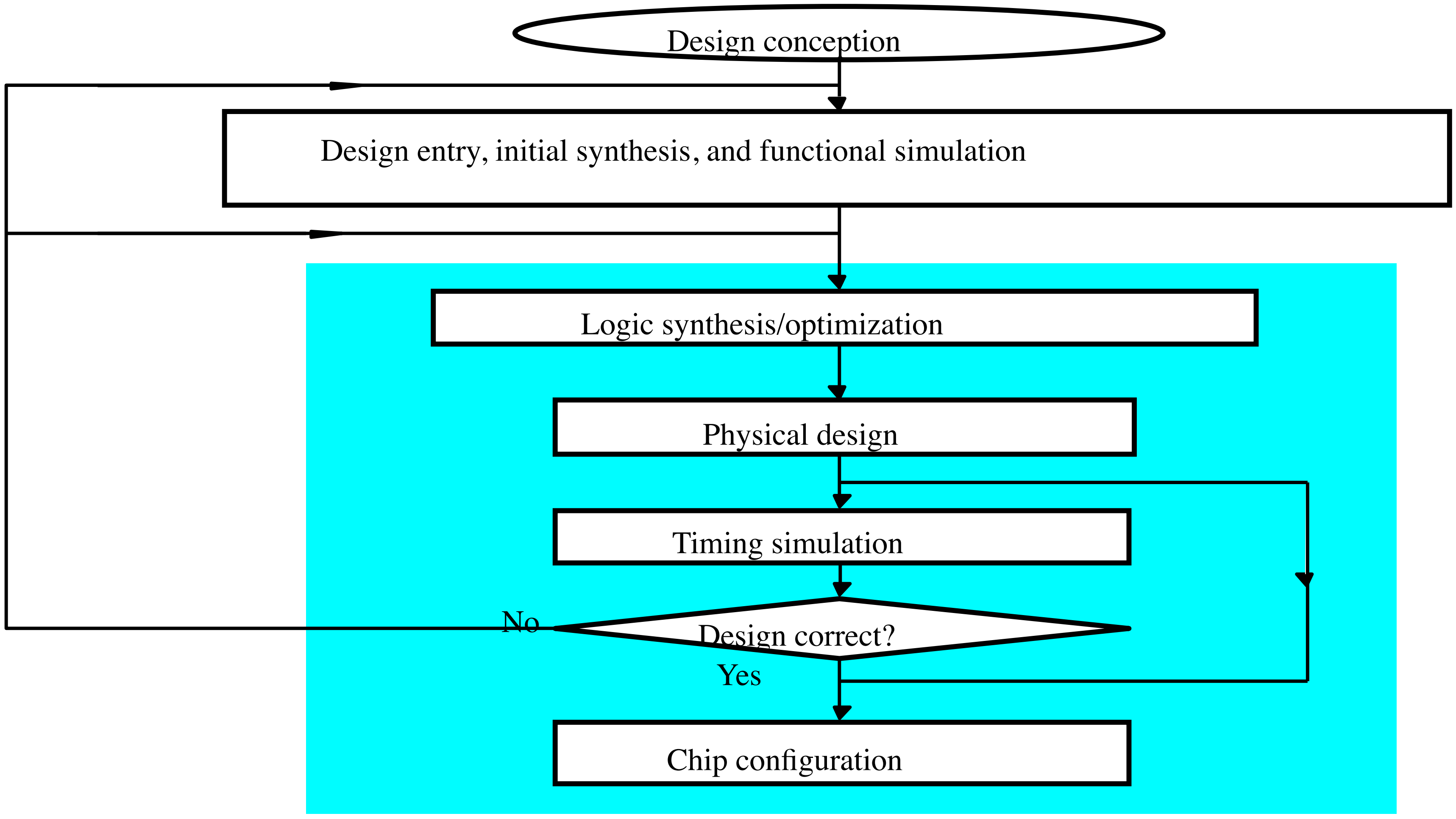
McMaster University

CAD tools

- A CAD system has tools for performing the following tasks:
  - Design entry
  - Initial synthesis
  - Functional simulation
  - Logic synthesis and optimization
  - Physical design
  - Timing simulation
  - Chip configuration



The first stages of a CAD system



A complete CAD system

CAD tools

- The starting point in the process of designing a digital circuit is the conception of what the circuit is supposed to do and the formulation of its general structure.
- This step is done manually. The rest is done by CAD tools.

CAD tools

- Design entry: a description of the circuit being designed should be entered into CAD system
- Different ways of doing this:
  - Truth tables
  - Schematic capture
  - Hardware description languages
- Initial synthesis: produces a network of logic gates

CAD tools

- Functional simulation: is used to verify the functionality of the circuit based on input provided by the designer
- This simulation is performed before any optimization and propagation delays are ignored.
- Goal: validate the basic operations of the circuit

CAD tools

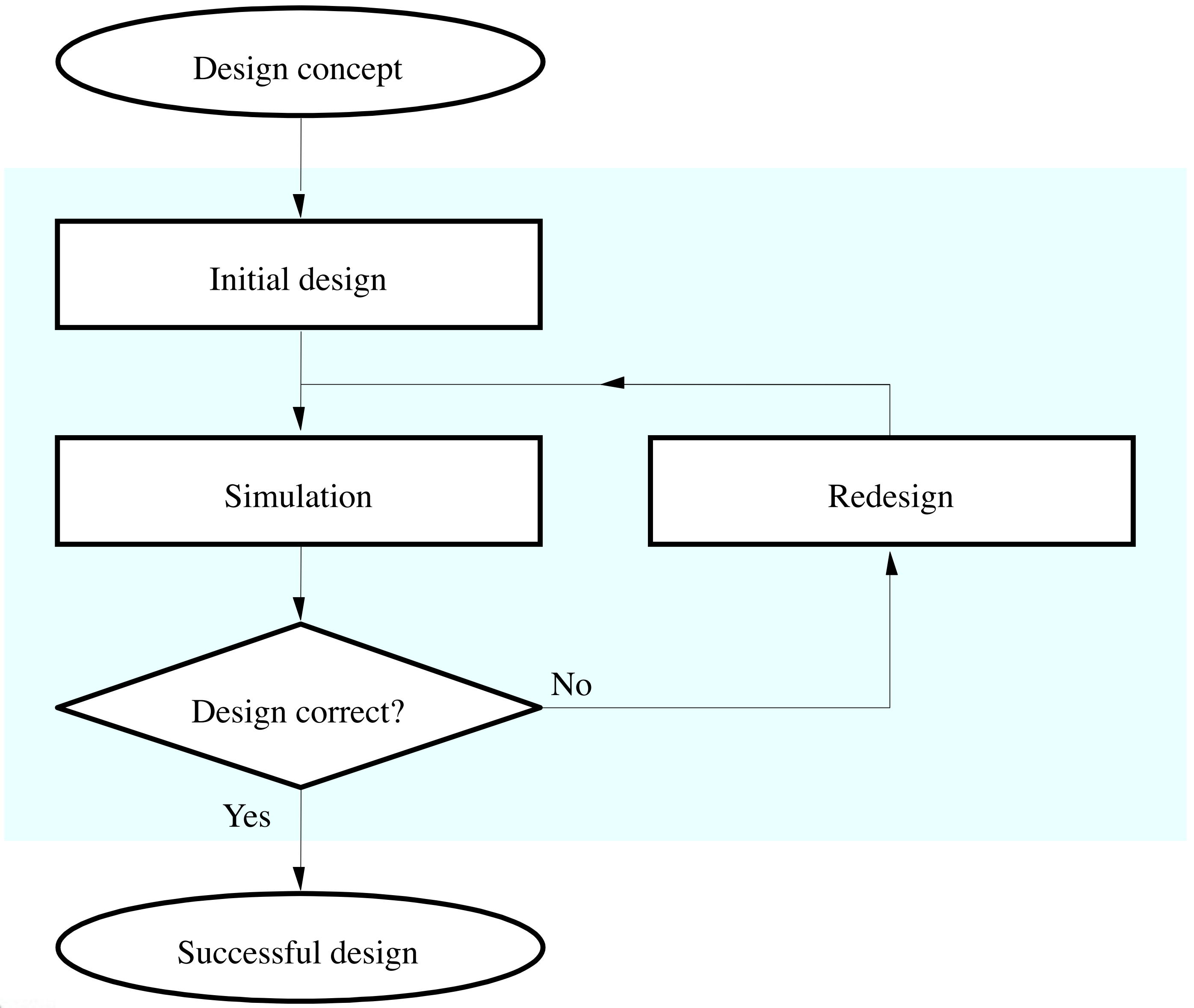
- Logic synthesis and optimization: produces an equivalent but better circuit
- The measure of what makes one circuit better depends on the needs of a design project and the technology chosen for implementation

CAD tools

- Physical design (layout synthesis): how to implement the circuit in the target technology
- This step consists of placement and routing
- Placement: where in the target device each logic function in the optimized circuit will be realized
- Routing: which wires in the chip are to be used to realize the required interconnections

CAD tools

- Timing simulation: determines the propagation delays that are expected in the implemented circuit
- Timing simulation: ensures that the implemented circuit meets the required performance
- Some of timing errors can be corrected by using the synthesis tool
- If the logic synthesis tool cannot resolve the timing problem, it is necessary to return to the beginning of the design flow to consider other alternatives
- Final step: configure the target chip to implement the circuit



The basic design loop

Introduction to VHDL

- VHDL is a language used to express complex digital systems concepts for documentation, simulation, verification and synthesis.
- VHDL is a language widely used to model and design digital hardware.
- Design tools translate a design described in VHDL into actual working system in various target technologies very fast and reliable.
- VHDL is supported by numerous CAD tools and programmable logic vendors.
- VHDL was first standardized in 1987 in IEEE 1076-1987
- An enhanced version was released in 1993

Introduction

- Advantages of using VHDL for design:

1. Shorter design time and reduced time to market
2. Reusability of already designed units
3. Fast exploration of design alternatives
4. Independence of the target implementation technology
5. Automated synthesis
6. Easy transportability to other design tools
7. Parallelization of the design process using a team work approach

Introduction

- VHDL consists of several parts organized as follows:

1. Actual VHDL language specified by IEEE
2. Some additional data type declarations in the standard package called IEEE standard 1164
3. A WORK library reserved for user's designs
4. Vendor packages with vendor libraries
5. User packages and libraries

VHDL design

- Two built-in libraries are WORK and STD
- VHDL source design units are complied into WORK library
- The ieee library is a storage place for IEEE standard design units
- User can create other libraries

VHDL design (Library)

- To use a library it should be declared (made accessible to the design):
  - Exp: library ieee
- WORK library is implicitly accessible in all designs and does not need to be declared
- Complied units in a library can be accessed via a use statement
- Syntax:
  - use library_name.package_name.item_name
  - use library_name.item_name
- Exp: use ieee.std_logic_1164.all

VHDL design (Package)

- Next level of hierarchy within a library is a package.
- Package is created to store common data types, constants and complied designs that will be used in more than one design (reusability)
- A package is used for:
  – Type and subtype declaration
  – Constant declaration
  – Function and procedure declaration
  – File declaration

Entity & Architecture

- A VHDL design is a paring of an entity declaration and an architecture body.
- Entity declaration: describes the design I/O and my include parameters used to customize an entity
- Architecture body: describes the function of a design
- Each I/O signal in an entity declaration is referred to as a port
- A port is a data object
- Like other data objects it can be assigned values and used in expressions

Entity & Architecture

- Each port must have a name, a direction (mode) and a data type.
- Mode: describes the direction in which data is transferred through a port
- Example: port (a, b : in bit_vector(3 downto 0);
  equals: out bit);
- Mode can be one of 4 values: in, out, inout, or buffer
- In: data flows only into the entity. The driver of the port is external (e.g., clock input)
- Out: data flows only from its source (inside the entity) to the port
- Note: out does not allow feedback

Entity & Architecture

- Buffer: for internal feedback (to use a port also as a driver within the architecture)
- Buffer is used for ports that must be readable inside the entity, such as the counter outputs (the present state of a counter must be used to determine its next stage
- Inout: allows data to flow into or out of the entity. It also allows for internal feedback
- Mode inout can replace any of the other modes

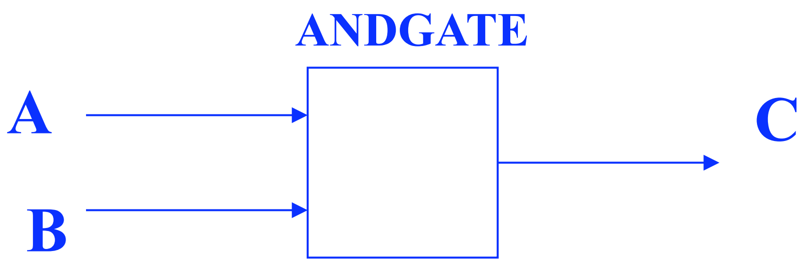
Entity & Architecture

- In addition to specifying modes for ports, you must declare data types for ports
- The most important data types in VHDL are Boolean, bit, bit_vector, and integer
- The most useful types provided by the IEEE std_logic_1164 package is std_logic and array of this type.
- For simulation and synthesis software to process these types, their declaration must be made visible to the entity by way of library and use clauses

```
entity eqcomp4 is
    port (a, b : in bit_vector(3 downto 0);
    equals: out bit);
end eqcomp4;
architecture dataflow of eqcomp4 is
begin
 equals <=‘1’ when (a=b) else ‘0’;
end dataflow;
```

VHDL design (Entity)

- Design entity defines a new component name, its input/output connections and describes parameterized values.
- Entity represents the I/O interface (external connections) of a component.



```
entity andgate is
    port (a,b : in bit;
    c: out bit);
end andgate
```

VHDL design (Entity)

- Syntax for an entity declaration:

```
entity entity_name is
[generic (list-of-generics-and-their-types);]
[port (list-of-interface-port-names-and-their-types);]
end [entity] entity-name;
```

- Generic list: allows additional information to pass into an entity
- Useful for parameterization of the design

VHDL design (Architecture)

- An architecture specifies the behavior, interconnections and components of an entity.
- Architecture defines the function of an entity
- It specifies the relationship between inputs and outputs.
- VHDL architectures are categorized in style as:
  1. Behavior
  2. Dataflow
  3. Structural
- A design can use any or all of these styles.

VHDL design (Architecture)

- Behavior: the behavior of the entity is expressed using sequentially executed procedural code (very similar to programming languages like C)
- Sometimes called high-level description
- Rather than specifying the structure of a circuit, you specify a set of statements that when executed in sequence model the behavior of the entity.
- Uses process statement and sequential statements (the ordering of statements inside process is important)

VHDL design (Architecture)

- Dataflow: specifies the functionality of the entity (the flow of information) without explicitly specifying its structure
- It specifies how data will be transferred from signal to signal and input to output without the use of sequential statements.
- No use of process or sequential statements
- Structural: an entity is modeled as a set of components connected by signals
- Components are instantiated and connected together

```
-- a four bit equality comparator
library ieee;
use iee.std_logic_1164.all;
entity eqcomp4 is
    port (a, b : in std_logic_vector(3 downto 0);
            equals: out std_logic);
end eqcomp4;
architecture behav of eqcomp4 is
begin
    comp: process (a, b);
    begin
            if a=b then
                    equals <= ‘1’;
            else
                    equals<=‘0’;
            end if;
    end process comp;
end behav;
```

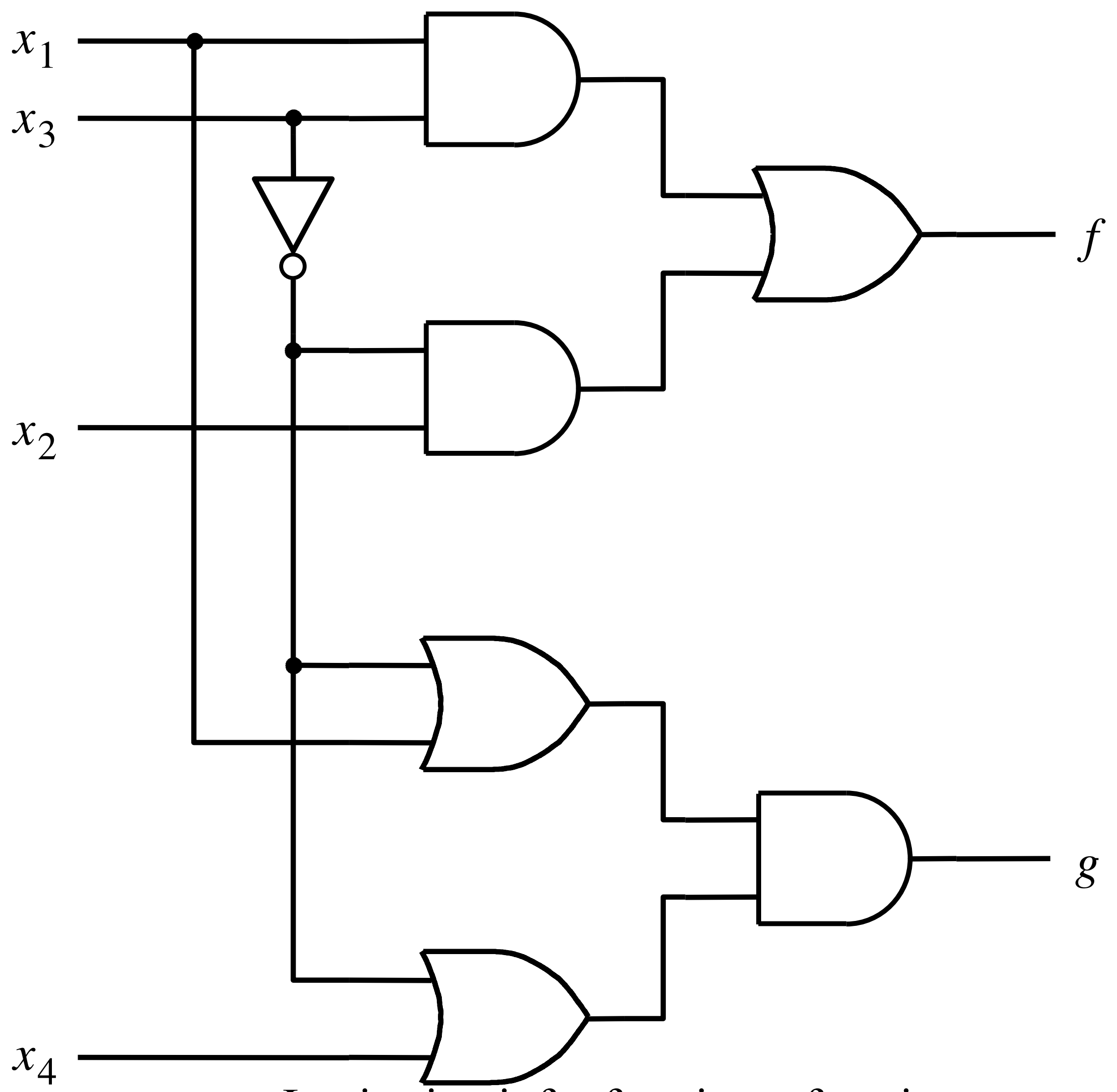
```
library ieee;
use iee.std_logic_1164.all;
entity eqcomp4 is
     port (a, b : in std_logic_vector(3 downto 0);
     equals: out std_logic);
end eqcomp4;

architecture dataflow of eqcomp4 is
begin
 equals <=‘1’ when (a=b) else ‘0’;
end dataflow;
```

```
library ieee;
use iee.std_logic_1164.all;
entity eqcomp4 is
     port (a, b : in std_logic_vector(3 downto 0);
     equals: out std_logic);
end eqcomp4;

architecture bool of eqcomp4 is
begin
 equals <=          not(a(0) xor b(0))
                    and not(a(1) xor b(1))
                    and not(a(2) xor b(2))
                    and not(a(3) xor b(3)) ;
end bool;
```

McMaster University

VHDL

- VHDL does not assume any precedence of operation and therefore parentheses are necessary in VHDL expressions.
- <= is the signal assignment operator in VHDL



Logic circuit for four-input function

```
ENTITY example2 IS
    PORT ( x1, x2, x3, x4  : IN   BIT ;
           f, g            : OUT  BIT ) ;
END example2 ;

ARCHITECTURE LogicFunc OF example2 IS
BEGIN
    f <= (x1 AND x3) OR (NOT x3 AND x2) ;
    g <= (NOT x3 OR x1) AND (NOT x3 OR x4) ;
END LogicFunc ;
```

Whenever there is an event on x1, x2, x3, the expression on the right side is evaluated and the value appears on f and/or g.

Example

- The signal assignments in the previous example are concurrent statements.
- Concurrent statements are order independent.

VHDL

- A very useful data type defined in VHDL: STD_LOGIC
- STD_LOGIC can have a number of legal values: 0, 1, z, -
- To use STD_LOGIC type the VHDL code must have these two lines:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

- The first line declares that the code will use ieee library

McMaster University

c_i	x_i	y_i	i	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Truth table

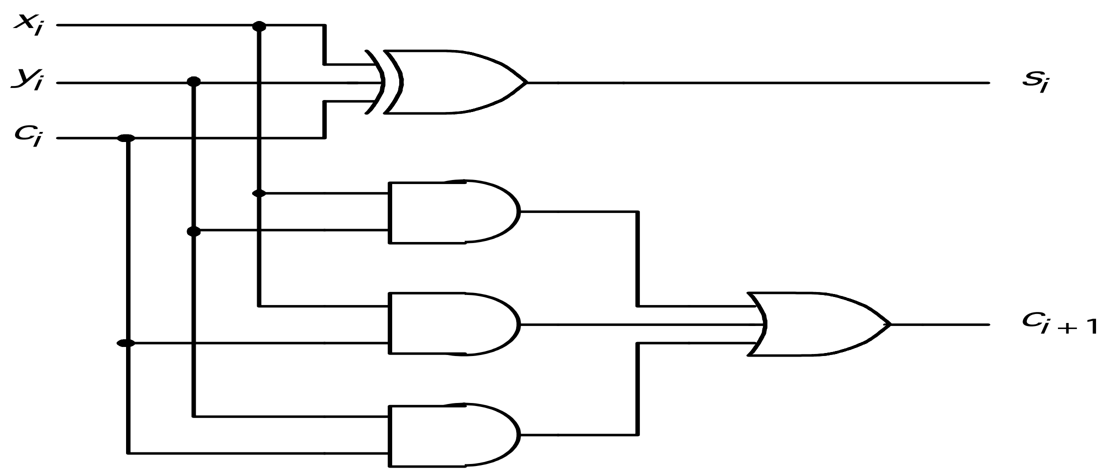
c_i \ x_iy_i	00	01	11	10
0		1		1
1	1		1	

$s_i = x_i$ Å y_i Å c_i

c_i \ x_iy_i	00	01	11	10
0			1	
1		1	1	1

$c_{i+1} = x_iy_i + x_ic_i + y_ic_i$

(b) Karnaugh maps



(c) Circuit

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY fulladd IS
	PORT (	Cin, x, y	: IN		STD_LOGIC ;
		s, Cout		: OUT		STD_LOGIC ) ;
END fulladd ;

ARCHITECTURE LogicFunc OF fulladd IS
BEGIN
	s <= x XOR y XOR Cin ;
	Cout <= (x AND y) OR (Cin AND x) OR (Cin AND y) ;
END LogicFunc ;
```

VHDL

- Now if we want to create a 4-bit adder we can use the 1-bit adder already designed as a sub-circuit.
- This is an important feature of VHDL which makes the reuse of entities possible.
- components: design entities used in other designs
- Before an entity can be used in another design it has to be declared
- A component declaration defines an interface for instantiating a component.

VHDL

- A component declaration may be
  1. in a package: the package is made accessible by use statement
  2. might be declared in an architecture declarative region using component statement
- Every time a component is used it has to be instantiated
- Every instantiation has a name.

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY adder4 IS
    PORT (  Cin             : IN    STD_LOGIC ;
            x3, x2, x1, x0  : IN    STD_LOGIC ;
            y3, y2, y1, y0  : IN    STD_LOGIC ;
            s3, s2, s1, s0  : OUT   STD_LOGIC ;
            Cout            : OUT   STD_LOGIC ) ;
END adder4 ;

ARCHITECTURE Structure OF adder4 IS
    SIGNAL c1, c2, c3 : STD_LOGIC ;
    COMPONENT fulladd
        PORT (  Cin, x, y   : IN    STD_LOGIC ;
                s, Cout     : OUT   STD_LOGIC ) ;
    END COMPONENT ;
BEGIN
    stage0: fulladd PORT MAP ( Cin, x0, y0, s0, c1 ) ;
    stage1: fulladd PORT MAP ( c1, x1, y1, s1, c2 ) ;
    stage2: fulladd PORT MAP ( c2, x2, y2, s2, c3 ) ;
    stage3: fulladd PORT MAP (
        Cin => c3, Cout => Cout, x => x3, y => y3, s => s3  ) ;
END Structure ;
```

Data types

- Now that we have multibit signals we are able to represents numbers.
- Are we able to perform arithmetic operations on the numbers in VHDL?
- Another package named std_logic_arith defines types for this
- Two predefined types in this package: SIGNED and UNSINGED
- SIGNED and UNSIGNED are the same as STD_LOGIC_VECTOR
- SIGNED represents signed integer data in two's complement form
- UNSIGNED represents unsigned integer data in the form of an array of std_logic

McMaster University

Arithmetic operations

$$\begin{tikzcd}
 & \text{std\_logic\_arith} \arrow[dl] \arrow[dr] & \\
\text{SIGNED} & & \text{UNSINGED}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{std\_logic\_signed} \arrow[d] \\
\text{STD\_LOGIC\_VECTOR}
\end{tikzcd}$$

$$\begin{tikzcd}
\text{std\_logic\_unsigned} \arrow[d] \\
\text{STD\_LOGIC\_VECTOR}
\end{tikzcd}$$

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_arith.all ;

ENTITY adder16 IS
    PORT (  Cin             : IN    STD_LOGIC ;
            X, Y            : IN    SIGNED(15 DOWNTO 0) ;
            S               : OUT   SIGNED(15 DOWNTO 0) ;
            Cout, Overflow  : OUT   STD_LOGIC ) ;
END adder16 ;

ARCHITECTURE Behavior OF adder16 IS
    SIGNAL Sum : SIGNED(16 DOWNTO 0) ;
BEGIN
    Sum <= ('0' & X) + Y + Cin ;
    S <= Sum(15 DOWNTO 0) ;
    Cout <= Sum(16) ;
    Overflow <= Sum(16) XOR X(15) XOR Y(15) XOR Sum(15) ;
END Behavior ;
```

Selective signal assignment

- Selected signal assignment is used to assign one of multiple values to a signal, based on some criteria.
- The WITH-SELECT-WHEN structure can be used for this purpose.
- Syntax: with selection_signal select

```
signal_name <= value_a when value1_of_selection_signal,
               value_b when value2_of_selection_signal,
               value_c when value3_of_selection_signal;
```

- All values of selection_signal must be listed in the when clause
- We can use the word OTHERS to cover some of the values

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
	PORT (	w0, w1, s	: IN	STD_LOGIC ;
		f		: OUT	STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
	WITH s SELECT
		f <= w0 WHEN '0',
		     w1 WHEN OTHERS ;
END Behavior ;
```

Conditional signal assignment

- Conditional signal assignment is used to assign one of multiple values to a signal, based on some criteria.
- The WHEN ELSE structure can be used for this purpose.
- Syntax:

```
signal_name <= value_a when condition1 else
               value_b when condition2 else
               value_c when condition3 else
               value_d;
```

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
USE ieee.std_logic_unsigned.all ;

ENTITY compare IS
    PORT (  A, B                : IN    STD_LOGIC_VECTOR(3 DOWNTO 0) ;
            AeqB, AgtB, AltB    : OUT   STD_LOGIC ) ;
END compare ;

ARCHITECTURE Behavior OF compare IS
BEGIN
    AeqB <= '1' WHEN A = B ELSE '0' ;
    AgtB <= '1' WHEN A > B ELSE '0' ;
    AltB <= '1' WHEN A < B ELSE '0' ;
END Behavior ;
```

Concurrent statements

- Assignment statements, selected assignment statements and conditional assignment statements are called concurrent statements because the order in which they appear in VHDL code does not affect the meaning of the code
- Concurrent statements are executed in parallel.
- Each concurrent statement is a different hardware element operating in parallel.

Sequential statements

- Sequential statements: ordering of statements may affect the meaning of the code
- Sequential statements should be placed inside process statement.

Process

```
[process_label:] process [(sensitivity_list)] [is]
begin
sequential_statements; these are
        wait_statement
        if_statement
        case_statement
end process [process_label]
```

Process

- Sensitivity list: signals to which the process is sensitive
- Each time an event occurs on any of the signals in sensitivity list, the sequential statements within the process are executed in the order they appear

If statement

- An if statement selects a sequence of statements for execution based on the value of a condition
- Syntax:

```
If condition1 then
      sequential_statements
[elsif condition2 then
      sequential_statements]
[else
      sequential_statements]
end if
```

McMaster University

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY mux2to1 IS
    PORT (  w0, w1, s       : IN      STD_LOGIC ;
            f               : OUT     STD_LOGIC ) ;
END mux2to1 ;

ARCHITECTURE Behavior OF mux2to1 IS
BEGIN
    PROCESS ( w0, w1, s )
    BEGIN
        IF s = '0' THEN
            f <= w0 ;
        ELSE
            f <= w1 ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

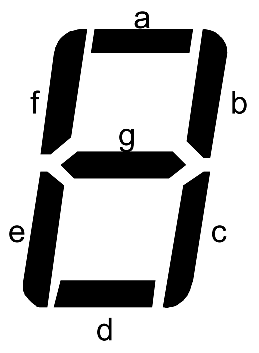
A 2-to-1 multiplexer specified using an if-then-else statement

Case

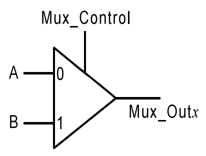
- Syntax:

```
case selection_signal is
      when value1_selection_signal => sequential_statements
      when value2_selection_signal => sequential_statements
      ..
      [when others => sequential_statements]
end case
```

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;
ENTITY seg7 IS
        PORT (    bcd      : IN         STD_LOGIC_VECTOR(3 DOWNTO 0) ;
                  leds     : OUT        STD_LOGIC_VECTOR(1 TO 7) ) ;
END seg7 ;
ARCHITECTURE Behavior OF seg7 IS
BEGIN
        PROCESS ( bcd )
        BEGIN
                CASE bcd IS                --              abcdefg
                        WHEN "0000"        => leds <=    "1111110" ;
                        WHEN "0001"        => leds <=    "0110000" ;
                        WHEN "0010"        => leds <=    "1101101" ;
                        WHEN "0011"        => leds <=    "1111001" ;
                        WHEN "0100"        => leds <=    "0110011" ;
                        WHEN "0101"        => leds <=    "1011011" ;
                        WHEN "0110"        => leds <=    "1011111" ;
                        WHEN "0111"        => leds <=    "1110000" ;
                        WHEN "1000"        => leds <=    "1111111" ;
                        WHEN "1001"        => leds <=    "1110011" ;
                        WHEN OTHERS        => leds <=    "-------" ;
                END CASE ;
        END PROCESS ;
END Behavior ;
```



```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;

ENTITY multiplexer IS                         -- Input Signals and Mux Control
        PORT(  A, B, Mux_Control        : IN    STD_LOGIC;
               Mux_Out1, Mux_Out2,
               Mux_Out3, Mux_Out4   : OUT   STD_LOGIC  );
END multiplexer;

ARCHITECTURE behavior OF multiplexer IS
BEGIN                                         -- selected signal assignment statement…

        Mux_Out1 <= A WHEN Mux_Control = '0' ELSE B;
                                              -- … with Select Statement
        WITH mux_control SELECT

        Mux_Out2 <=    A WHEN    '0',
                       B WHEN    '1',
                       A WHEN OTHERS;         -- OTHERS case required since STD_LOGIC
                                              --     has values other than "0" or "1"
        PROCESS ( A, B, Mux_Contro l)
        BEGIN                                 -- Statements inside a process
               IF Mux_Control = '0' THEN      --     execute sequentially.
                 Mux_Out3 <= A;
               ELSE
                 Mux_out3 <= B;
               END IF;

               CASE Mux_Control IS
                       WHEN '0' =>
                               Mux_Out4 <= A;
                       WHEN '1' =>
                               Mux_Out4 <= B;
                       WHEN OTHERS =>
                               Mux_Out4 <= A;
               END CASE;
        END PROCESS;
END behavior;
```



Combinational logic implementation

- Combinational logic circuits can be modeled in different ways:
  - Using signal assignment statements (which include expressions with logic, arithmetic and relational operators)
  - Using if and case statements

Logic operators

- Standard VHDL logical operators are defined for types bit, std_logic, Boolean and their arrays.

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity logic_operators_1 is
    Port(a, b, c, d, : in std_logic; y: out std_logic);
End logic_operators_1;
Architecture arch1 of logic_operators_1 is
Signal e:bit;
Begin
    y <=(a and b) or e;
    e <= c or d;
End arch1;
```

Conditional Logic

- Concurrent statements for creating conditional logic:
  - Conditional signal assignment
  - Selected signal assignment
- Sequential statements for creating conditional logic:
  - If statement
  - Case statement

Conditional Logic

```
Library ieee;
Use ieee.std_logic_1164.all;

Entity condit_stmt is Port(
                              sel, b, c: in boolean;
                               y:  out boolean);
End condit_stmt;

Architecture concurrent of condit_stmt is
Begin
    y <=b when sel else c;
End concurrent;
```

Conditional Logic

- The same function implemented using sequential statements

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity condit_stmt is Port(
                sel, b, c: in boolean;
                y: out boolean);
End condit_stmt;
Architecture sequential of condit_stmt is
begin
     Process(s,b,c)
      begin
         if sel then
            y <=b;
         else
           y <=c;
         end if;
      end process;
end sequential;
```

Three-state (tri-state) logic

- When data from multiple possible sources need to be directed to one or more destinations we usually use either multiplexers or three-state buffers.
- Output buffers are placed in a high impedance state so they do not drive a shared bus at the wrong time
- Bidirectional pins are placed in high impedance state so they are not driven by off-chip signals at the wrong time
- VHDL: Using the ‘Z’ (high impedance) which applies to the type std_logic

Three-state (tri-state) logic

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity tbuf4 is Port(
        enable: in std_logic;
        a: in std_logic_vector(0 to 3);
        y: out std_logic_vector(0 to 3));
End tbuf4;
Architecture arch1 of tbuf4 is
Begin
    Process(enable, a)
    begin
        if enable='1' then
          y<=a;
        else
         y<='Z';
        end if;
    end process
End arch1;
```

Three-state (tri-state) logic

The same function implemented using concurrent statements

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity tbuf4 is Port(
        enable: in std_logic;
        a: in std_logic_vector(0 to 3);
        y: out std_logic_vector(0 to 3));
End tbuf4;
Architecture arch2 of tbuf4 is
Begin
 y <=a when enable=‘1’ else ‘Z’;
End arch2;
```

Wait

- When a process has a sensitivity list it is always suspended after executing the last statement in the process
- Wait is an alternative way of suspending a process
- Syntax:

```
wait on sensitivity_list
wait until boolean_expression;
```

```
LIBRARY ieee ;
USE ieee.std_logic_1164.all ;

ENTITY flipflop IS
    PORT (  D, Clock  : IN     STD_LOGIC ;
            Q         : OUT    STD_LOGIC) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
    PROCESS ( Clock )
    BEGIN
        IF Clock'EVENT AND Clock = '1' THEN
            Q <= D ;
        END IF ;
    END PROCESS ;
END Behavior ;
```

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY flipflop IS
	PORT (	D, Clock	: IN	STD_LOGIC ;
		Q		: OUT	STD_LOGIC ) ;
END flipflop ;

ARCHITECTURE Behavior OF flipflop IS
BEGIN
	PROCESS
	BEGIN
		WAIT UNTIL Clock'EVENT AND Clock = '1' ;
		Q <= D ;
	END PROCESS ;
END Behavior ;
```

Sequential Logic Synthesis

- Sequential logic elements: latch, flip-flop, register, counter
- Behavior of sequential logic elements can be described using a process statement
- The sequential nature of process statements make them idea for the description of circuits that have memory and must save their state over time
- The design of sequential logic uses one or more of the following rules:
  1. A process that does not include all entity inputs in the sensitivity list (otherwise the combinational circuit will be inferred)
  2. Use incompletely specified if-then-elsif logic to imply that one or more signals must hold their values under certain conditions

McMaster University

Basic Sequential Logic

- Two most basic types of synchronous elements:
  1. D-type latch
  2. D-type flip-flop
- D-type latch: a level sensitive memory element that passes the input (D) to output (Q) when enabled (ENA=1) and hold the value of the output when disabled (ENA=0)
- D-type flip-flop: an edge-triggered memory element that transfers the input (D) to output (Q) when an active edge transition occurs on its clock. The output value is held until the next active clock edge
- Active clock edge: transition of clock from 0 to 1

Basic Sequential Logic

- Conditional specification is the most common method in describing behavior of basic memory elements
- This relies on an if statement and assigning a value in only one condition
- Example: a D latch

```
process(enable)
begin
   if enable='1' then
       q <=d;
  end if;
end process
```

- If we had assigned values in both conditions the behavior would be a multiplexer

Basic Sequential Logic

- Exp: Edge triggered flip-flop

```
process(clk)
begin
  if (clk and clk'event) then
     q <=d;
  end if;
End process
```

- The second method is to use a wait statement

```
wait until clk and clk'event
q <=d;
```

Basic Sequential Logic

- Latches can have additional inputs such as preset and clear.
- Preset and clear inputs to the latch are always asynchronous.
- Exp: a latch with active high preset and clear

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity latch is
    Port(enable, clear, preset, d : in std_logic;  q:out std_logic);
End latch;
Architecture arch2 of latch is
Begin
 process(enable, preset, clear)
   begin
   if (clear = ‘1’) then
      q <=‘0’;
   elsif (preset=‘1’)
      q <=‘1’;
   elsif (enable=‘1’)
      q <=d;
   end if;
  End process;
End arch2;
```

Basic Sequential Logic

- Registers can be implemented using if statement or wait statement
- They can have any combination of clear, preset and enable
- Exp: register with edge-triggered clock and asynchronous load

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity reg is
Port(load, clk, d, data : in std_logic; q:out std_logic);
End reg;
Architecture arch1 of reg is
Begin
 process(load, clk)
 begin
 if (load = ‘1’) then
   q <=data;
 elsif clk’event and clk=‘1‘
   q <=d;
 end if;
end process
End arch1;
```

Basic Sequential Logic

- Counters can be implemented with if and wait statements
- Exp: an 4-bit, synchronous load, up-down counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity counter is port (clk, clr, load, up, down: in std_logic; data: in std_logic_vector(3 downto 0); count: out
      std_logic_vector(3 downto 0));
end counter;

architecture count4 of counter is
      signal cnt: std_logic_vector(3 downto 0);
begin
      process (clr, clk)
      begin
              if clr='1' then cnt<='0000';
              elsif clk'event and clk='1' then
                          if load='1' then cnt<=data
                          elsif up='1' then
```

4-bit up-down counter

```
                if cnt='1111' then cnt <='0000';
                else cnt<=cnt+1;
                endif
        elsif down='1' then
                if cnt='0000'then cnt<='1111';
                else cnt<=cnt-1;
                end if
        else
          cnt<=cnt;
        end if;
   end if;
   count<=cnt;
   end process;
end count4;
```

Finite-State Machine

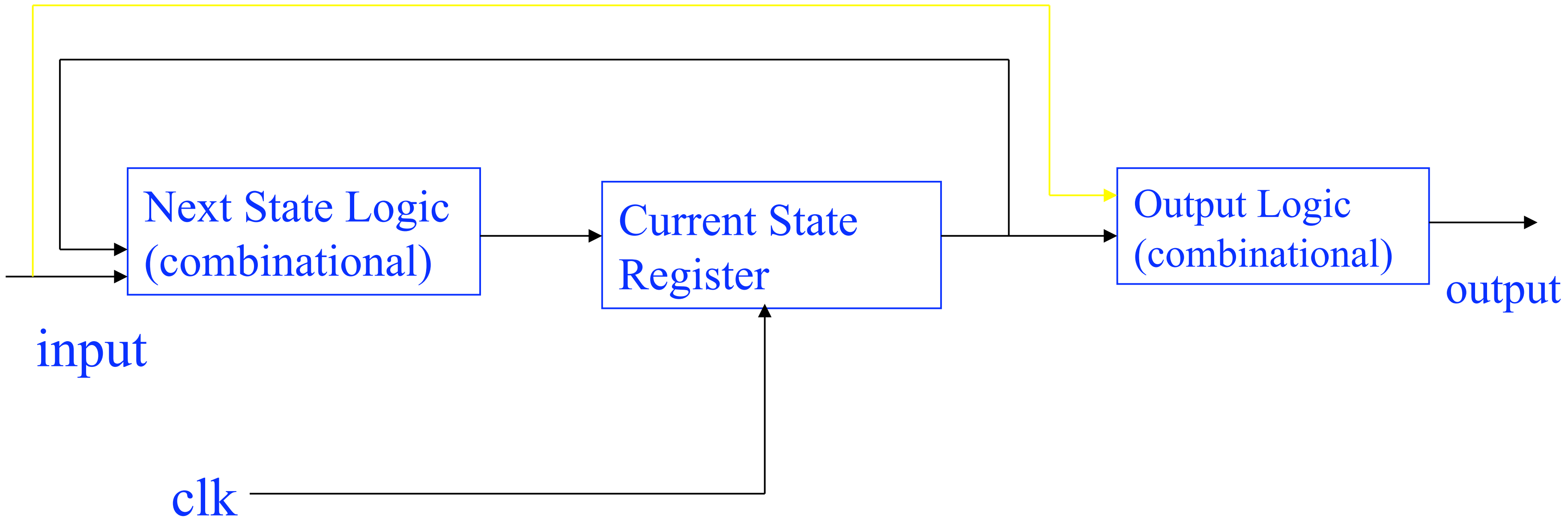
- Finite State Machines (FSMs) represent an important part of design of almost any complex digital system.
- FSMs are used to sequence specific operations, control other logic circuits, and provide synchronization of different parts of more complex circuits.
- FSM is a circuit that is designed to sequence through specific patterns of states in a predetermined manner.
- Sequence of states through which an FSM passes depends on the current state of the FSM and the input
- A state is represented by the binary value held on the current state register
- FSM is clocked from a free running clock source

Finite-State Machine

- FSM contains three main parts:

1. Current state register: holds the current state of FSM (the current state should be represented in binary form)
2. Next state logic: a combinational logic used to generate the transition to the next state from the current state
   - The next state is a function of current state and inputs to FSM
   - A feedback mechanism is necessary in FSM
3. Output logic: a combinational circuit used to generate output signals from the FSM.
   - Outputs are a function of the current state and possibly FSM's input

Finite-State Machine

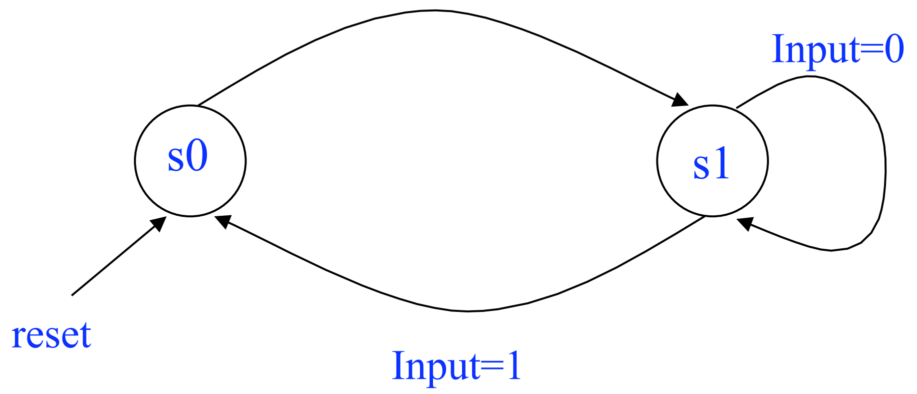

Next State Logic (combinational)
Current State Register
Output Logic (combinational)
input
output
clk

Finite-State Machine

- If the output is a function of only the current state: Moore FSM
- If the output depends on the input and state: Mealy FSM
- Behavior of an FSM is usually described either in the form of a state transition table or a state transition diagram.

Finite-State Machine

- To describe a FSM, an enumeration type for states, and a process statement for the state register and the next-state logic can be used.
- Example:



Finite-State Machine

```
Library ieee;
Use ieee.std_logic_1164.all;
Entity sm is
  Port(clk, reset, input : in std_logic; output: out std_logic);
End sm;
Architecture arch of state_machine is
  Type state_type is (s0, s1);
  Signal state: state_type;
Begin
  Process(clk, reset)
  Begin
    If reset=‘1’ then state <=s0;
    Elsif (clk’event and clk=‘1’) then
    Case state is
       When s0 =>
           state <= s1;
       When s1 =>
            If input=‘0’ then state <=s1;
            Else state <=s0;
            End if;
     End case;
    End if;
  End process;
Output <=‘1’ when state=s1 else ‘0’;
End arch;
```

Finite-State Machine

- An important issue when designing FSMs is state encoding: assignment of binary numbers to states.
- For small designs or those in which there are not too tight constraints in terms of resources, the common way is to let the synthesis tool encode the state automatically.
- For bigger designs a kind of manual intervention is necessary
- Sequential state encoding: increasing binary numbers are assigned to the states
  - Exp: s0=“00”, s1=“01”, s2=“10”,
- Gray code or Johnson state encoding are other options.
- One-hot encoding: each state is assigned its own flip-flop, in each state only one flip-flop can have value ‘1’.

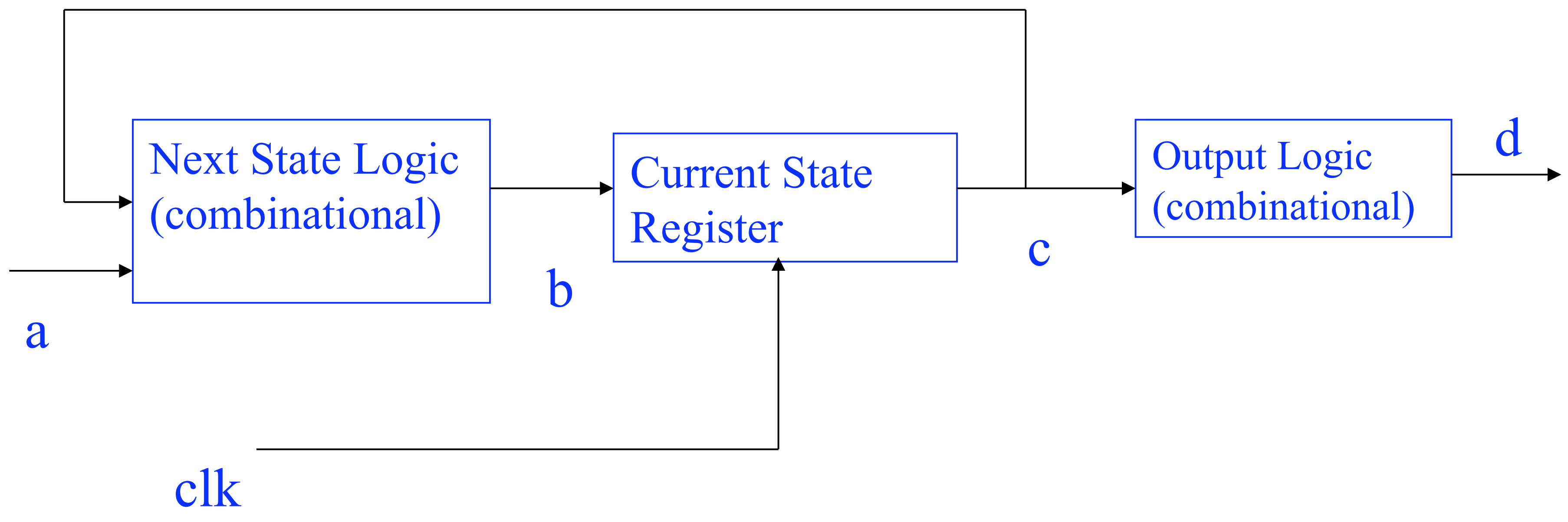
McMaster University

Finite-State Machine

- One-hot encoding is not optimal in terms of number of flip-flops, but used very often by FPLD synthesis tools.
- Reasons:
  - FPLDs have a high number of flip-flops available,
  - A large number of flip-flops used for state representation leads to a simpler next state logic.

Moore Machines

- Moore state machine: outputs are a function of current state only

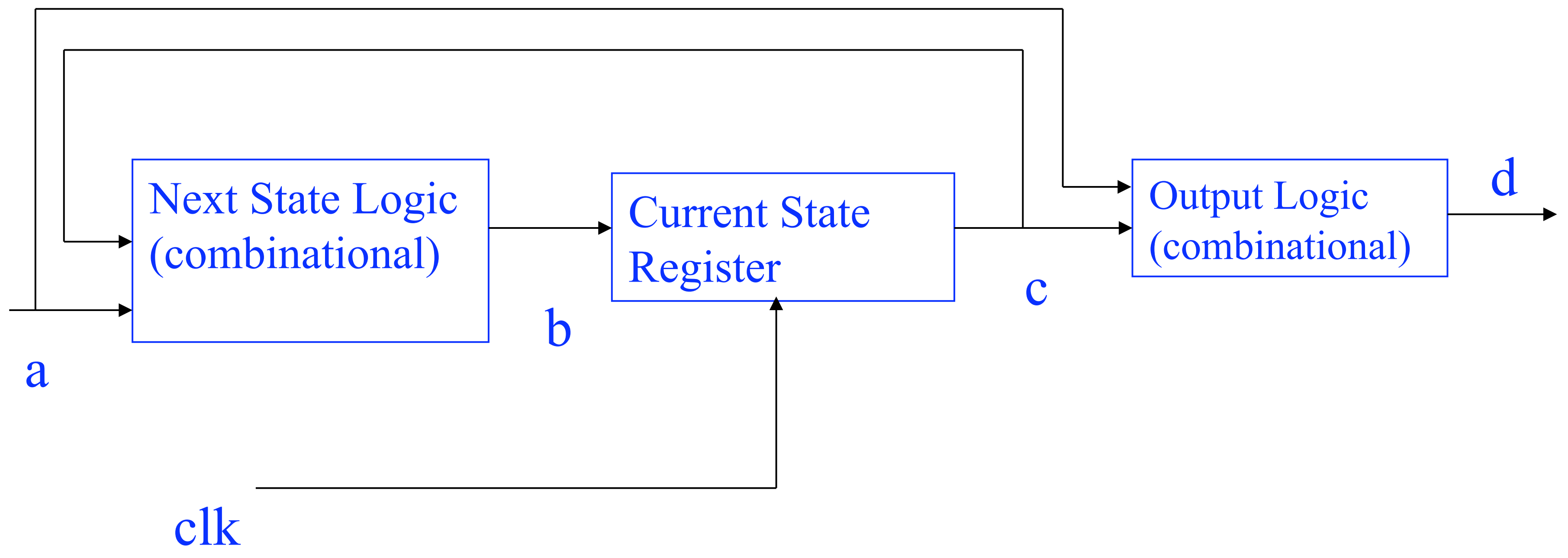

Next State Logic (combinational)
Current State Register
Output Logic (combinational)
a
b
c
d
clk

Moore Machines (pseudo code)

```
Entity system is
   Port (clk: std_logic; a: some_type; d: out some_type);
End system
Architecture moore1 of system is
  Signal b,c: some_type;
begin
Next_state: process (a,c)
   begin
     b <=next_state_logic(a,c);
   end process next_state;
State_reg: process(clk)
  begin
    if  (clk'event and clk='1') then
    c<=b;
  end process state_reg;
System_output: process(c)
  begin
     d<=output_logic(c)
  end process system_output
end moore1
```

Mealy Machines

- Mealy state machine: outputs are a function of current state and system inputs both


Next State Logic (combinational)
Current State Register
Output Logic (combinational)
a
b
c
d
clk

Mealy Machines (pseudo code)

```
Entity system is
  Port (clk: std_logic; a: some_type; d: out some_type);
End system
Architecture mealy1 of system is
  Signal b,c: some_type;
begin
next_state: process (a,c)
   begin
     b <=next_state_logic(a,c);
   end process next_state;
system_output: process(a,c)
  begin
     d<=output_logic(a,c)
  end process system_output
state_reg: process (clk)
  begin
    if  (clk'event and clk='1') then
    c<=b;
  end process state_reg;
end mealy1
```

Integer types

- Integer type: a type whose set of values fall within a specified integer range
- Exp:
  - Type index is integer range 0 to 15
  - Type word_length is range 31 downto 0;
- Values belonging to an integer type are called integer literals
- The underscore character can be used freely in writing integer literals and has no impact on the value of a literal
- Exp: 98_71_28 is the same as 987128

Type conversion

- to_stdlogicvector(bit_vector): converts a bit vector to a standard logic vector
- example: to_stdlogicvector(X"FFFF")
- conv_std_logic_vector(integer, bits): converts an integer to a standard logic vector
- example: conv_std_logic_vector(7,4) generates “0111”
- conv_integer(std_logic_vector): converts a standard logic vector to an integer
- example: conv_integer(“0111”) produces 7

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY memory IS
        PORT(  read_data        :       OUT  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
               read_address     :       IN   STD_LOGIC_VECTOR( 2 DOWNTO 0 );
               write_data       :       IN   STD_LOGIC_VECTOR( 7 DOWNTO 0 );
               write_address    :       IN   STD_LOGIC_VECTOR( 2 DOWNTO 0 );
               Memwrite         :       IN   STD_LOGIC;
               Clock            :       IN   STD_LOGIC  );
END memory;

ARCHITECTURE behavior OF memory IS
                                              -- define new data type for memory array
        TYPE memory_type IS ARRAY ( 0 TO 7 ) OF STD_LOGIC_VECTOR( 7 DOWNTO 0 );
        SIGNAL memory           : memory_type;
BEGIN
        -- Read Memory and  convert array index to an integer with CONV_INTEGER
        read_data <= memory( CONV_INTEGER( read_address( 2 DOWNTO 0 ) ) );

        PROCESS                         -- Write Memory?
        BEGIN
                WAIT UNTIL clock 'EVENT AND clock = '1';
                IF ( memwrite = '1' ) THEN
                                        -- convert array index to an integer with CONV_INTEGER
                    memory( CONV_INTEGER( write_address( 2 DOWNTO 0 ) ) ) <= write_data;
                END IF;
        END PROCESS;
END behavior;
```

Altera's VHDL (Quartus)

Package	Library	Content
maxplus2	altera	Quartus primitives and macrofunctions
std_logic_1164	ieee	Standard for VHDL modeling and the std_logic and std_logic_vector types
std_logic_arith	ieee	Singed and unsigned types, arithmetic and comparison functions for use with singed and unsigned types and the conversion functions conv_integer conv_singed, conv_unsigned
std_logic_signed	ieee	Functions to use std_logic_vector types as if they are singed types
std_logic_unsigned	ieee	Functions to use std_logic_vector types as if they are unsinged types
lpm_components	lpm	LPM megafunctions supported by VHDL.

Library of commonly used circuits

- Primitives: basic functional blocks used in circuit design
- Macrofunctions: collection of high-level building blocks that can be used in logic designs
- Macrofunction usual names have the prefix a_ due to the fact that VHDL does not support names that begin with digits.

Primitives

- Primitives:
  - Basic building blocks
  - Package: altera_primitives_components
  - Library: altera

Primitive	Primitive Name
Buffer	Carry, Cascade, Exp, Global, Lcell, Soft, Tri
Flip-Flop & latch	dff, dffe, jkff, KJFFF, SRFF, SRFFE, TFF, TFFE, latch
Input and outputs	INOUT, IN, OUT
Logic	AND, NOR, BAND, NOT, BNAND, OR, BNOR, XOR,…..

Macrofunctions

- Macrofunctions:
  - Collection of high-level building blocks that can be used in logic designs
  - All input ports have default signal values, so the designer can simply leave unused inputs unconnected
  - Macrofunction usual names have the prefix a_ due to the fact that VHDL does not support names that begin with digits.

Macrofunctions

Macrofunction	Name	Description
Adder	a_8fadd a_7480 a_74283	8 bit full adder Gated full adder 4 bit full adder with fast carry
Arithmetic logic unit	a_74181 a_74182	Arithmetic logic unit Look ahead carry generator
Application specific	ntsc	NTSC video control signal generator
Buffer	a_74240 a_74241	Octal inverting 3-state buffer Octal 3-state buffer
Comparator	a_8mcomp a_7485 a_74688	8-bit magnitude comparator 4-bit magnitude comparator 8-bit identity comparator
Converter	a_74184	BCD-to-binary converter

Macrofunctions

Macrofunction	Name	Description
Counter	Gray4 a_7468 a_7493 a_74191 a_74669	Gray code counter Dual decade counter 4-bit binary counter 4-bit up/down counter with asynch. load Synchr. 4-bit up/down counter
Decoder	a_16dmux a_7446 a_74138	4-to-16 decoder BCD-to-7 segment decoder 3-to-8 decoder
EDAC	a_74630	16-bit parallel error detection and correction circuit
Encoder	a_74148 a_74348	8-to-3 encoder 8-to-3 priority encoder with 3-state outputs
Frequency divider	a_7456	Frequency divider

Macrofunctions

Macrofunction	Name	Description
Latch	Inpltch a_7475 a_74259 a_74845	Input latch 4-bit bistable latch 8 bit addressable latch with clear 8 bit bus interface D latch with 3 state outputs
Multiplier	Mult4 a_74261	4-bit parallel multiplier 2-bit parallel binary multiplier
Multiplexer	a_21mux a_74151 a_74157 a_74356	2-to-1 multiplexer 8-to-1 multiplexer Quad 2-to-1 multiplexer 8-to-1 data selector/multiplexer/register with 3 state outputs
Parity generator/ checker	a_74180	9-bit odd/even parity generator/checker

Macrofunctions

Macrofunction	Name	Description
Register	a_7470 a_7473 a_74171 a_74173 a-74396	AND gated JK filp flop with preset and clear Dual JK flip-flop with clear Quad D flip-flop with clear 4-bit D register Octal storage register
Shift register	Barrelst a_7491 a_7495 a_74198 a_74674	8-bit barrel shifter Serial-in serial out shift register 4-bit parallel access shift register 8 bit bidirectional shift register 16-bit shift register
Storage register	a_7498	4-bit data selector/storage register

Macrofunctions

Macrofunction	Name	Description
SSI functions	Inhb a_7400 a_7421 a_7432 a_74386	Inhibit gate NAND2 gate AND4 gate OR2 gate Quadruple XOR gate
True/Complement I/O element	a_7487	4-bit true/complement I/O element Quadruple complementary output elements

Macrofunctions

```
Library ieee;
Use ieee_std_logic_1164.all;
Library altera;
Use altera.maxplus2.all;
Entity example is
    Port(data, clock, clearn, presetn: in std_logic;
          q_out: out std_logic;
          a, b, c, gn: in std_logic;
          d: in std_logic_vector(7 downto 0);
          y, wn: out std_logic);
End example
Architecture arch of example is
Begin
    dff1: dff port map (d=>data, q=>q_out clk=>clock, clrn=>clearn, prn=>presetn);
    mux: a_74151 port map(c,b,a,d, gn, y, wn);
End arch;
```

Library of Parameterized Modules

- Library of Parameterized Modules (lpm) is a library of macrofunctions that is included in Quartus II
- Each module is parameterized: there are parameters and the module can be used in different ways.
- Modules in the library are technology independent.
- The modules can be included in a schematic entry mode or in VHDL code.
- Package: lpm_components
- Library: lpm

lpm modules

	Name	Description
Gates	lpm_and	Multi-bit and gate
	lpm_inv	Multi-bit inverter
	lpm_bustri	Multi-bit three state buffer
	lpm_mux	Multi-input multi-bit multiplexer
	lpm_clshift	Combinatorial logic shifter and barrel shifter
	lpm_or	Multi-bit or gate
	lpm_constant	Constant generator
	lpm_xor	Multi-bit xor gate
	lpm_decode	Decoder
	mux	Single input multi-bit multiplexer
	busmux	Two-input multi-bit multiplexer
Arithmetic Components	lpm_compare	Two-input multi-bit comparator
	lpm_abs	Absolute value
	lpm_counter	Multi-bit counter with various control options
	lpm_add_sub	Multi-bit adder subtractor
	lpm_divide	Parameterized Divider
	lpm_mult	Multi-bit multiplier

McMaster University

lpm modules

	Name	Description
Memory	altdpram*	Parameterized Dual-Port RAM
	lpm_latch	Parameterized Latch
	csfifo	Cycle shared first-in first-out buffer
	lpm_shiftreg	Parameterized Shift Register
	dcfifo*	Parameterized Dual-Clock FIFO
	lpm_ram_dp	Parameterized Dual-Port RAM
	scfifo*	Parameterized Single-Clock FIFO
	lpm_ram_dq	Synchronous or Asynchronous RAM with a separate I/O ports
	csdpram	Cycle shared dual port RAM
	lpm_ram_io	Synchronous or Asynchronous RAM with a single I/O port
	lpm_ff	Parameterized flip flop
	lpm_rom	Synchronous or Asynchronous ROM
	lpm_fifo	Parameterized Single-Clock FIFO
	lpm_dff*	Parameterized D-Type flip flop and Shift Register
	lpm_fifo_dc	Parameterized Dual-Clock FIFO
	lpm_tff*	Parameterized T-Type flip flop
Other functions	clklock	Parameterized Phase-Locked Loop
	pll	Rising- and Falling-Edge Detector
	ntsc	NTSC Video Control Signal Generator

Parameterized Modules

- An instance of a parameterized function is created with a component instantiation statement and a generic map.
- Generic map assigns values to the parameters.

Parameterized Modules

```
Library ieee;
Use ieee_std_logic_1164.all;
Library lpm;
Use lpm.lpm_components.all;
Entity reg24lpm is
    Port(d: in std_logic_vector(23 downto 0); clk: in in std_logic;
           q: out std_logic_vector(23 downto 0));
End reg24lpm;
Architecture arch of reg24lpm is
Begin
    reg12a: lpm_ff
    generic map (lpm_width =>12)
    port map(data =>d(11 downto 0), clock => clk, q => q(11 downto 0));

    reg12b: lpm_ff
    generic map (lpm_width =>12)
    port map(data =>d(23 downto 12), clock => clk, q => q(23 downto 0));
end arch;
```

VHDL Synthesis of Multiply & Divide

- The lpm_mult function can be used to synthesize integer multiplication
- The function lpm_divide is also available for integer division.
- Syntax

```
COMPONENT lpm_mult
  GENERIC (LPM_WIDTHA: POSITIVE;
    LPM_WIDTHB: POSITIVE;
    LPM_WIDTHS: POSITIVE;
    LPM_WIDTHP: POSITIVE;
    LPM_REPRESENTATION: STRING := "UNSIGNED";
    LPM_PIPELINE: INTEGER := 0;
    LPM_TYPE: STRING := "L_MULT";
    LPM_HINT : STRING := "UNUSED");
  PORT (dataa: IN STD_LOGIC_VECTOR(LPM_WIDTHA-1 DOWNTO 0);
    datab: IN STD_LOGIC_VECTOR(LPM_WIDTHB-1 DOWNTO 0);
    aclr, clken, clock: IN STD_LOGIC := '0';
    sum: IN STD_LOGIC_VECTOR(LPM_WIDTHS-1 DOWNTO 0) := (OTHERS => '0');
    result: OUT STD_LOGIC_VECTOR(LPM_WIDTHP-1 DOWNTO 0));
END COMPONENT;
```

```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
USE  IEEE.STD_LOGIC_ARITH.ALL;
USE  IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY mult IS
        PORT(  A, B         : IN    STD_LOGIC_VECTOR(   7 DOWNTO 0 );
                  Product   : OUTSTD_LOGIC_VECTOR( 15 DOWNTO 0 ) );
END mult;

ARCHITECTURE a OF mult IS
BEGIN                                           -- LPM 8x8 multiply function P = A * B
        multiply: lpm_mult
        GENERIC MAP(  LPM_WIDTHA                   =>  8,
                      LPM_WIDTHB                   =>  8,
                      LPM_WIDTHS                   => 16,
                      LPM_WIDTHP                   => 16,
                      LPM_REPRESENTATION           =>   "UNSIGNED" )

        PORT MAP (    data   => A,
                      datab => B,
                      result => Product );
END a;
```

VHDL synthesis of memory

- The memory functions in LPM are lpm_ram_dq, lpm_ram_dp, lpm_ram_io, and lpm_rom.
- The memory can be set to an initial value using a file with extension .mif.
- lpm_ram_dq can implement asynchronous memory or memory with synchronous inputs and/or outputs.
- The lpm_ram_dq function uses EABs in FLEX 10K and Cyclone devices.
- The Quartus Compiler automatically implements suitable portions of this function in EABs.
- Small blocks of special purpose memory can be synthesized using registers.

VHDL synthesis of memory

VHDL Component Declaration:

```
COMPONENT lpm_ram_dq
  GENERIC (LPM_WIDTH: POSITIVE;
    LPM_TYPE: STRING := "L_RAM_DQ";
    LPM_WIDTHAD: POSITIVE;
    LPM_NUMWORDS: POSITIVE;
    LPM_FILE: STRING := "UNUSED";
    LPM_INDATA: STRING := "REGISTERED";
    LPM_ADDRESS_CONTROL: STRING := "REGISTERED";
    LPM_OUTDATA: STRING := "REGISTERED";
    LPM_HINT: STRING := "UNUSED");
  PORT (data: IN STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0);
    address: IN STD_LOGIC_VECTOR(LPM_WIDTHAD-1 DOWNTO 0);
    we: IN STD_LOGIC := '1';
    inclock: IN STD_LOGIC := '1';
    outclock: IN STD_LOGIC := '1';
    q: OUT STD_LOGIC_VECTOR(LPM_WIDTH-1 DOWNTO 0));
END COMPONENT;
```

VHDL synthesis of memory

LPM_WIDTH	Width of data[] and q[] ports.
LPM_WIDTHAD	Width of the address port.
LPM_NUMWORDS	Number of words stored in memory.
LPM_FILE	Name of the Memory Initialization File (.mif) or Hexadecimal (Intel-Format) File (.hex) containing ROM initialization data ("<filename>"), or "UNUSED". If omitted, contents default to all 0's.

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY memory IS
        PORT(  read_data        :       OUT  STD_LOGIC_VECTOR( 7 DOWNTO 0 );
                read_address    :       IN      STD_LOGIC_VECTOR( 2 DOWNTO 0 );
                write_data      :       IN      STD_LOGIC_VECTOR( 7 DOWNTO 0 );
                write_address   :       IN      STD_LOGIC_VECTOR( 2 DOWNTO 0 );
                Memwrite        :       IN      STD_LOGIC;
                Clock           :       IN      STD_LOGIC  );
END memory;

ARCHITECTURE behavior OF memory IS
                                        -- define new data type for memory array
        TYPE memory_type IS ARRAY ( 0 TO 7 ) OF STD_LOGIC_VECTOR( 7 DOWNTO 0 );
        SIGNAL memory           : memory_type;
BEGIN
        -- Read Memory and  convert array index to an integer with CONV_INTEGER
        read_data <= memory( CONV_INTEGER( read_address( 2 DOWNTO 0 ) ) );

        PROCESS                         -- Write Memory?
        BEGIN
                WAIT UNTIL clock 'EVENT AND clock = '1';
                IF ( memwrite = '1' ) THEN
                                -- convert array index to an integer with CONV_INTEGER
                    memory( CONV_INTEGER( write_address( 2 DOWNTO 0 ) ) ) <= write_data;
                END IF;
        END PROCESS;
END behavior;
```

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
LIBRARY lpm;
USE lpm.lpm_components.ALL;

ENTITY amemory IS
        PORT( read_data             :       OUT    STD_LOGIC_VECTOR( 7 DOWNTO 0 );
                  memory_address    :       IN     STD_LOGIC_VECTOR( 2 DOWNTO 0 );
                  write_data        :       IN     STD_LOGIC_VECTOR( 7 DOWNTO 0 );
                  Memwrite          :       IN     STD_LOGIC;
                  clock,reset       :       IN     STD_LOGIC );
END amemory;

ARCHITECTURE behavior OF amemory IS
BEGIN
        data_memory: lpm_ram_dq                -- LPM memory function

        GENERIC MAP ( lpm_widthad           => 3,
                      lpm_outdata           => "UNREGISTERED",
                      lpm_indata            => "REGISTERED",
                      lpm_address_control   => "UNREGISTERED",
                                              -- Reads in mif file for initial data values (optional)
                      lpm_file              => "memory.mif",
                      lpm_width             => 8 )

        PORT MAP ( data  => write_data, address  => memory_address( 2 DOWNTO 0 ),
                   We    => Memwrite, inclock    => clock, q => read_data );
END behavior;
```