VHDL

Programmable Logic Devices

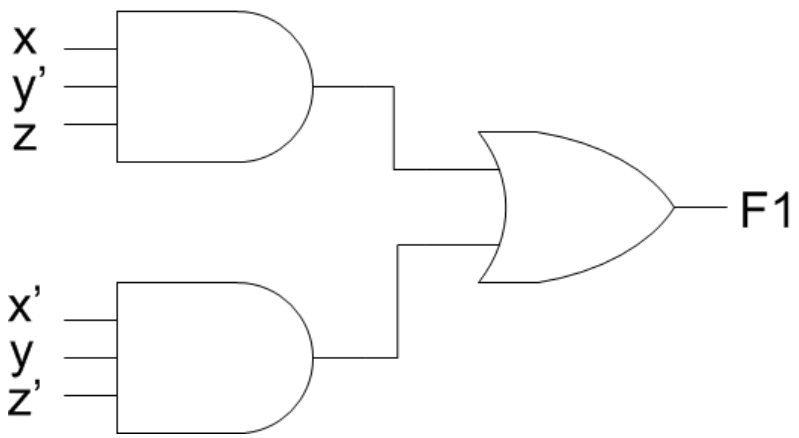
- EEPROM (Electrical Erasable Programmable ROM)
  - Memory devices
- PLA (programmable logic array)
  - Can be used to directly implement any two-level SOP logic equations
  - Array of AND gates and OR gates
- PAL (programmable AND-array logic device)
  - Sum terms is fixed, program only the product terms

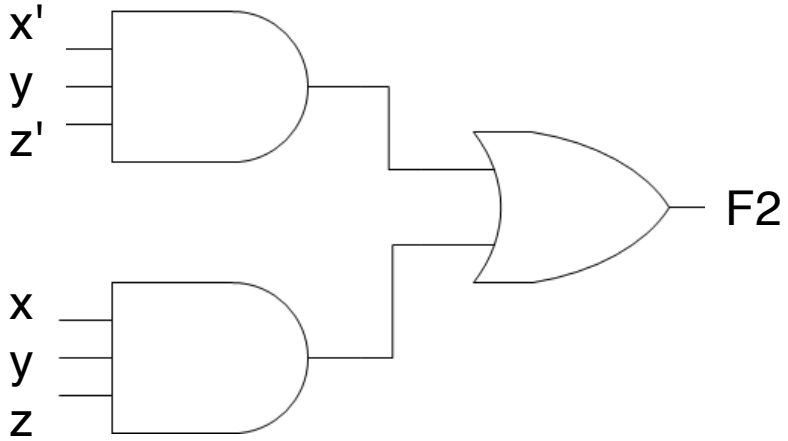
Programmable Logic Devices

- GAL (Generic Array Logic Device)
  - Similar to PAL with added functionality
- Gate Array
  - Identical cells laid out on a rectangular matrix with routing channel between adjacent rows and columns
  - FPGA (Field Programmable Gate Array)
  - CPLD (Complex Programmable Logic Device)
- Application-Specific Integrated Circuit (ASIC)
  - Highest level of complexity

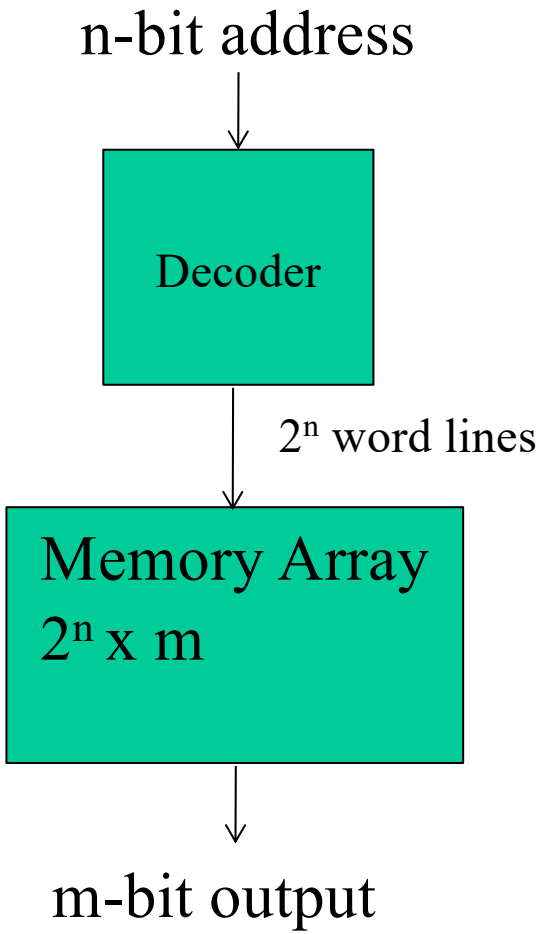
Boolean Function

- F1(x, y, z) = xy'z + x'y z'
- F2(x, y, z) = x'yz' + xyz





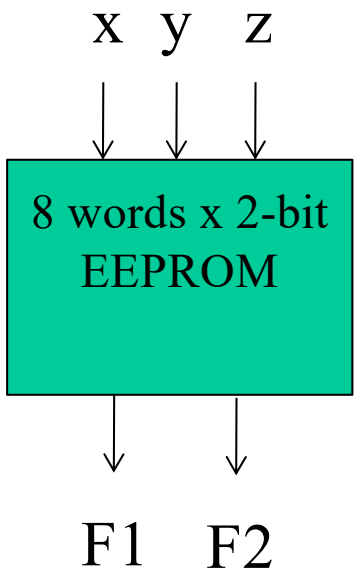
EEPROM



$F1(x, y, z) = xy'z + x'yz'$

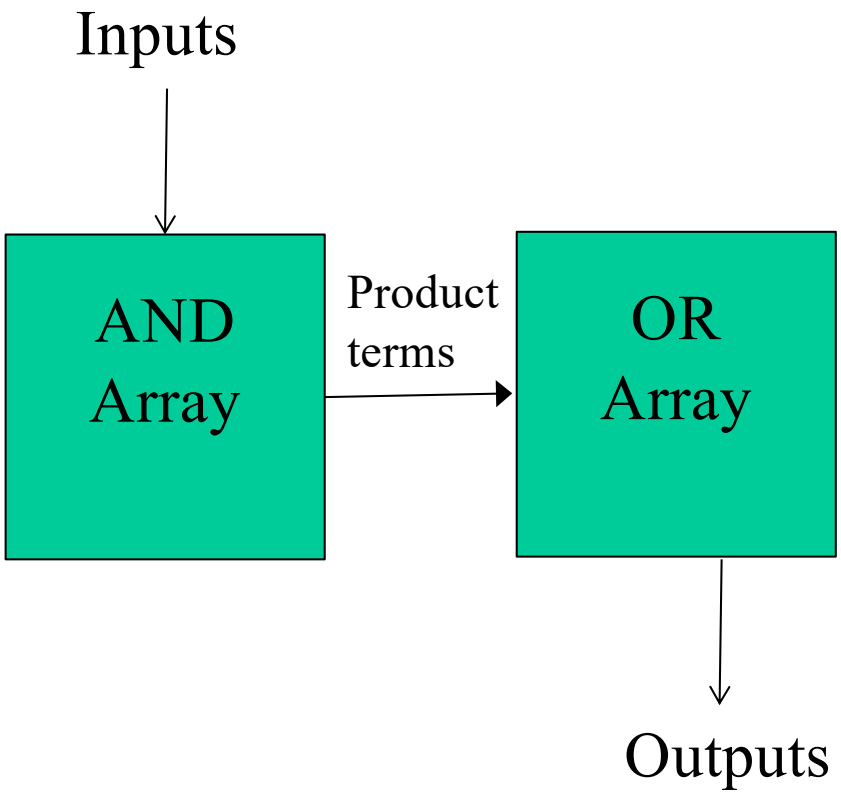
$F2(x, y, z) = x'yz' + xyz$

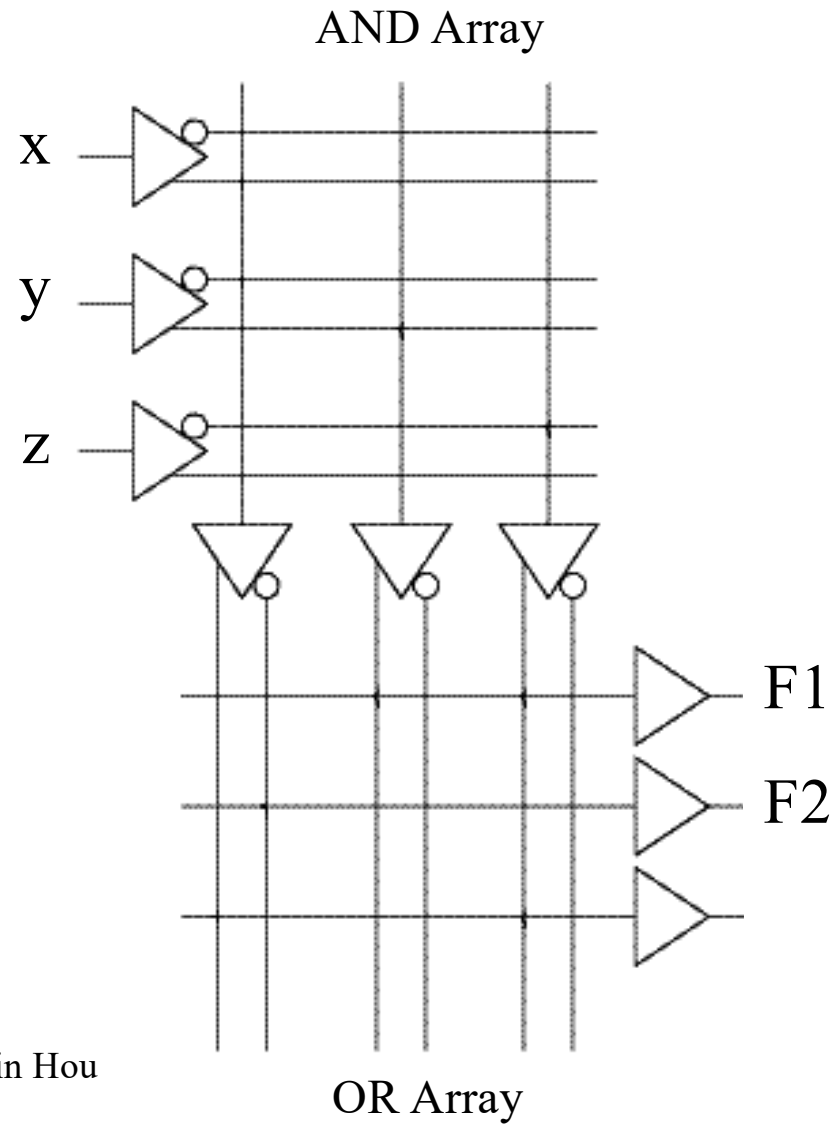
x	y	z	F1	F2
0	0	0	0	0
0	0	1	0	0
0	1	0	1	1
0	1	1	0	0
1	0	0	0	0
1	0	1	1	0
1	1	0	0	0
1	1	1	0	1



PLA

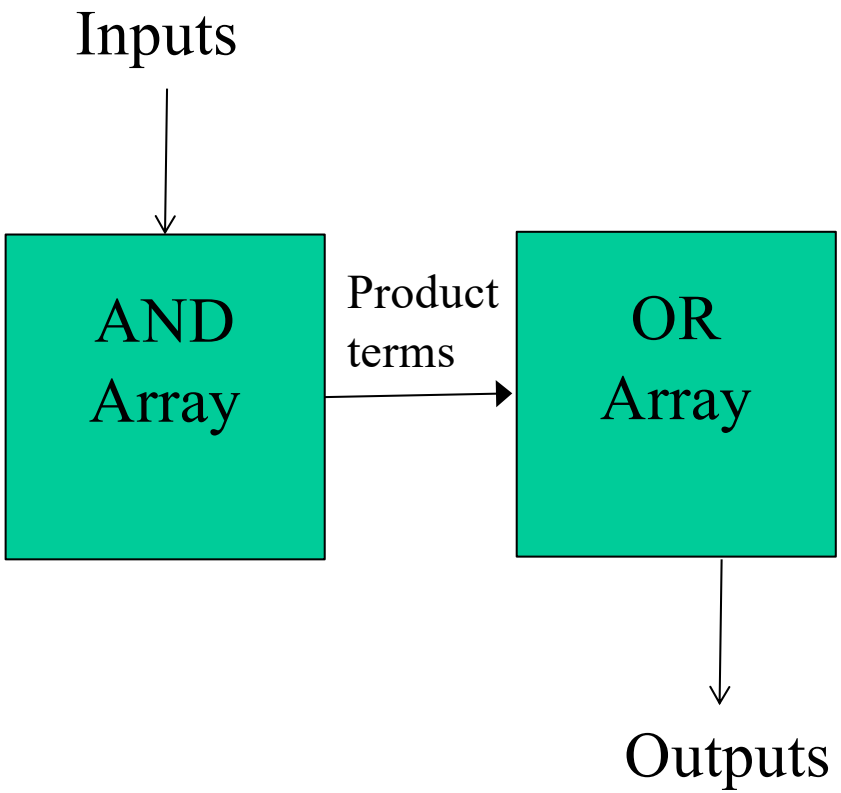
F1(x, y, z) = xy'z + x'yz'
F2(x, y, z) = x'yz' + xyz

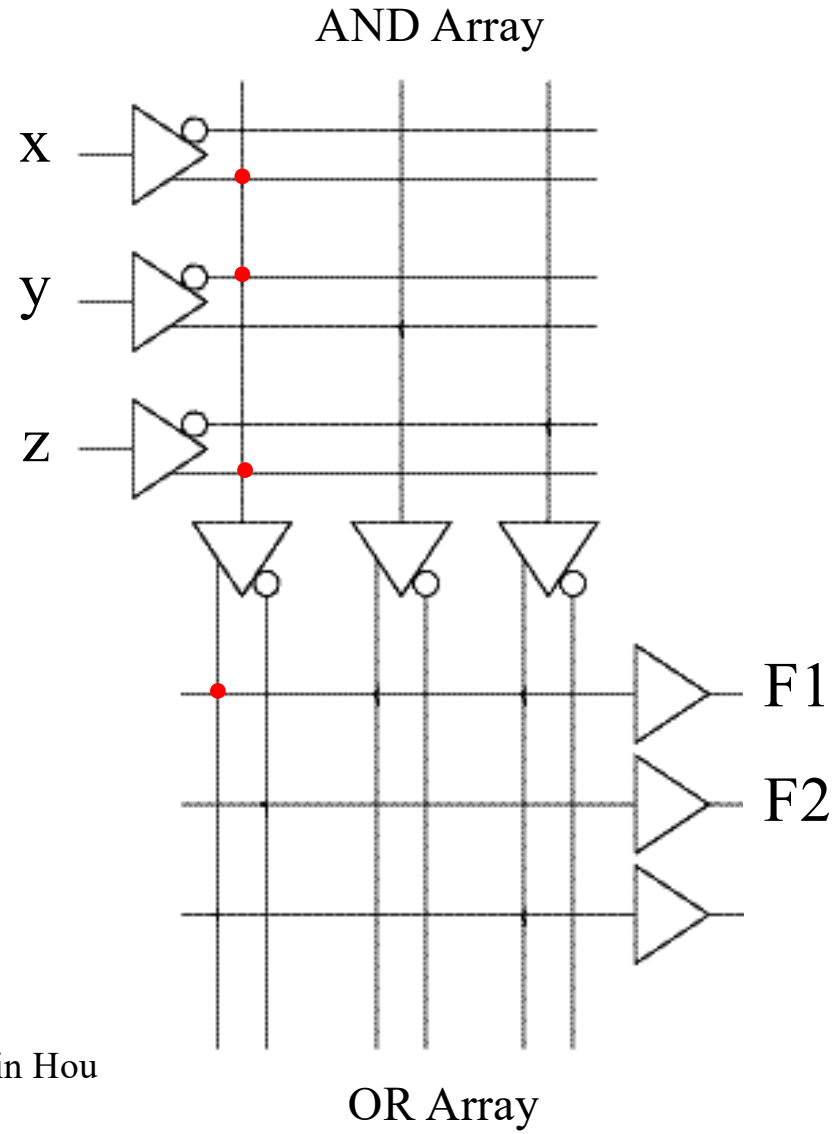




PLA

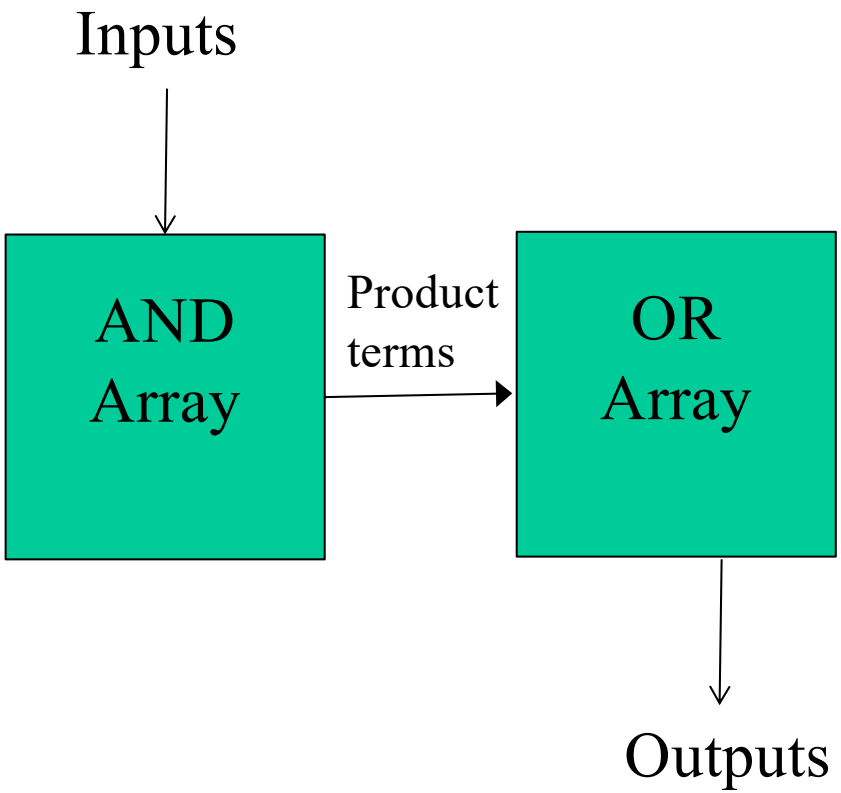
F1(x, y, z) = xy'z + x'yz'
F2(x, y, z) = x'yz' + xyz

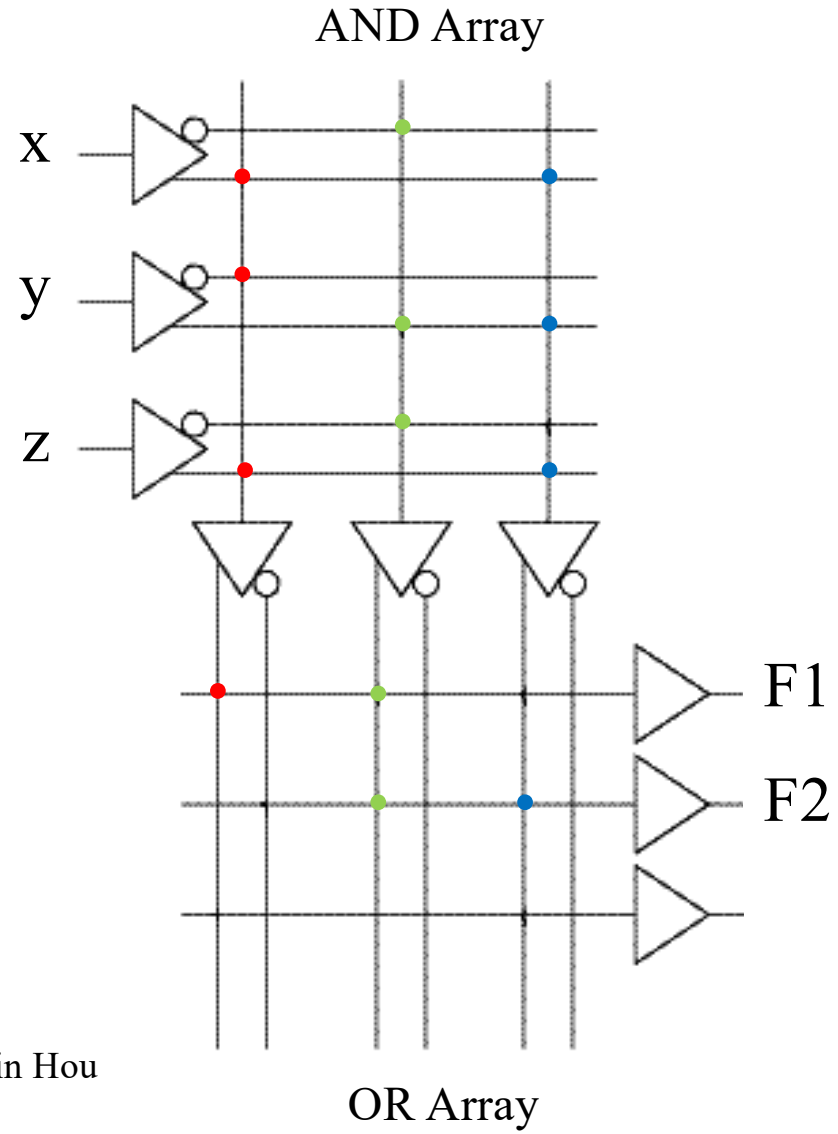




PLA

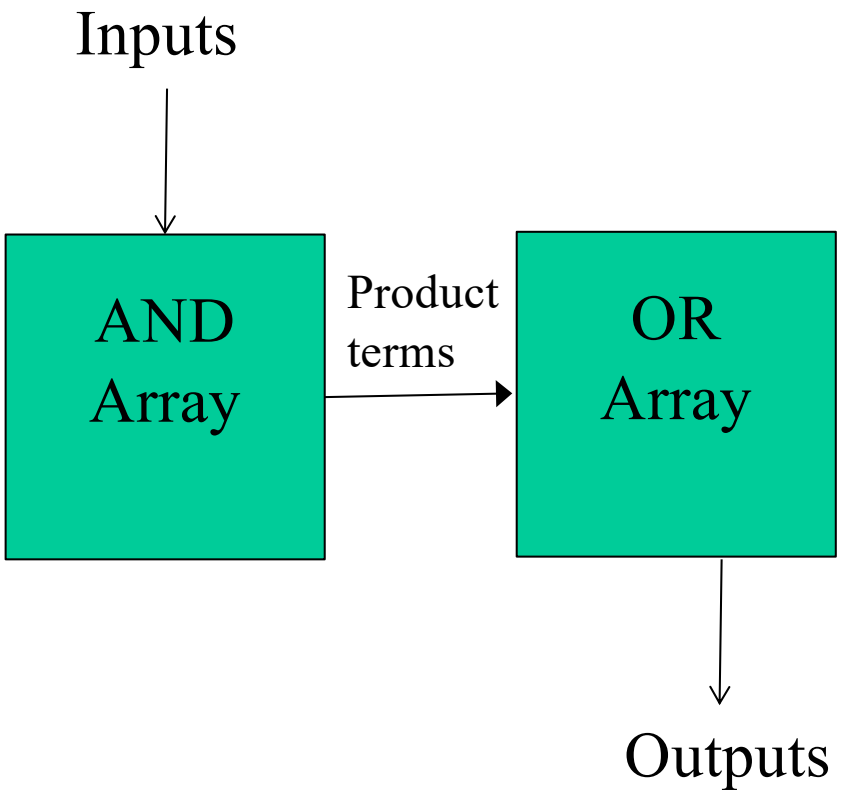
F1(x, y, z) = xy'z + x'yz'
F2(x, y, z) = x'yz' + xyz

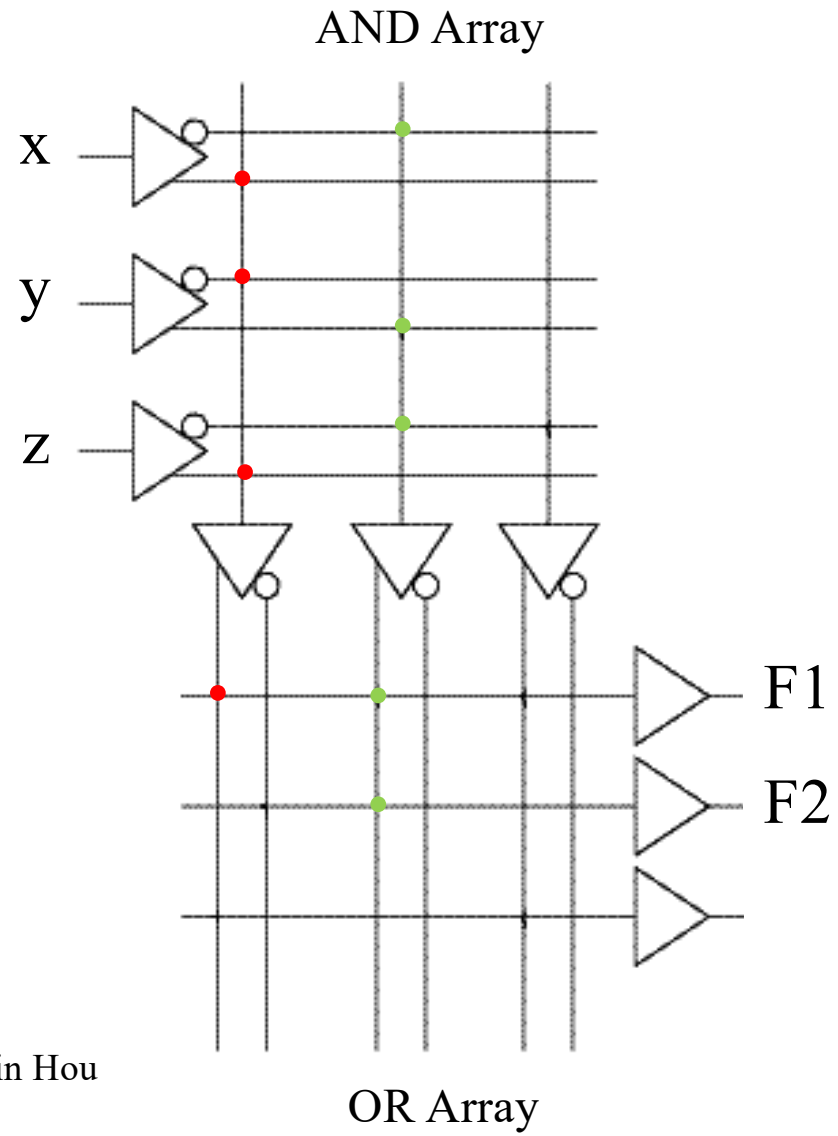




PLA

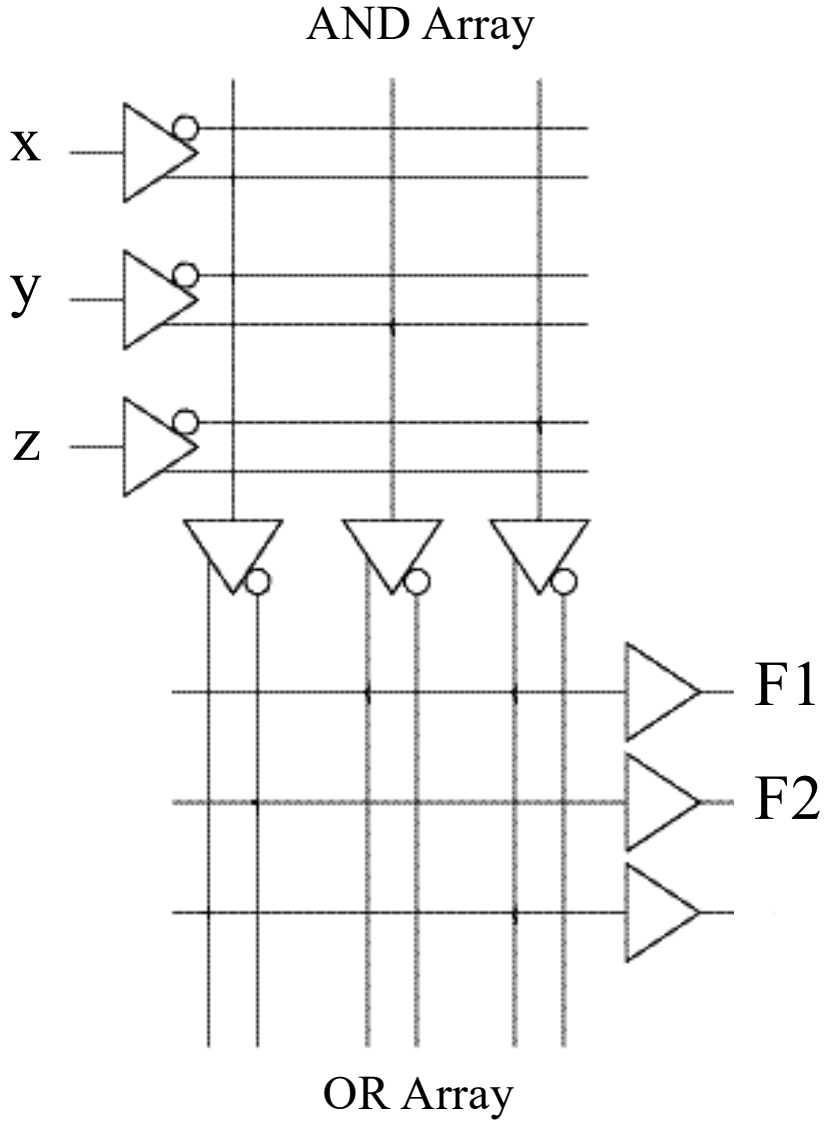
F1(x, y, z) = xy'z + x'yz'
F2(x, y, z) = x'yz' + xyz





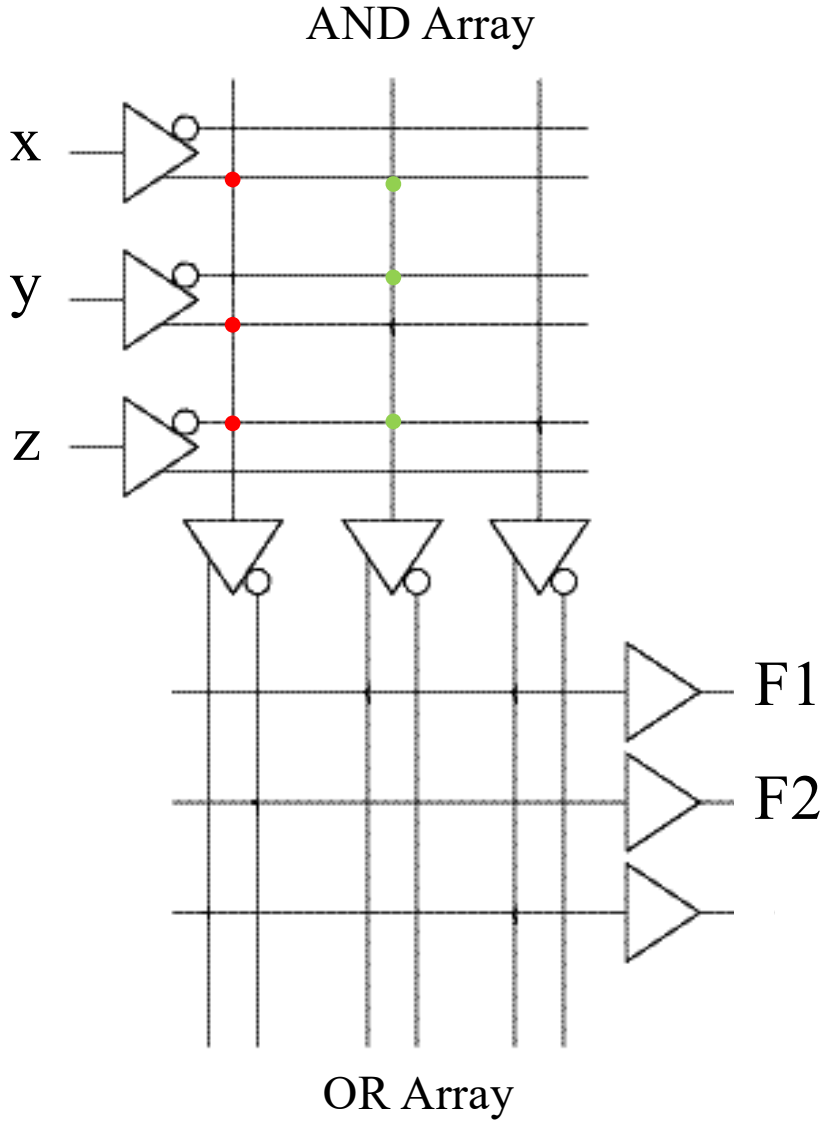
In class exercise

$$F1(x, y, z) = xyz' + xy'z'$$
$$F2(x, y, z) = x'yz' + xyz'$$

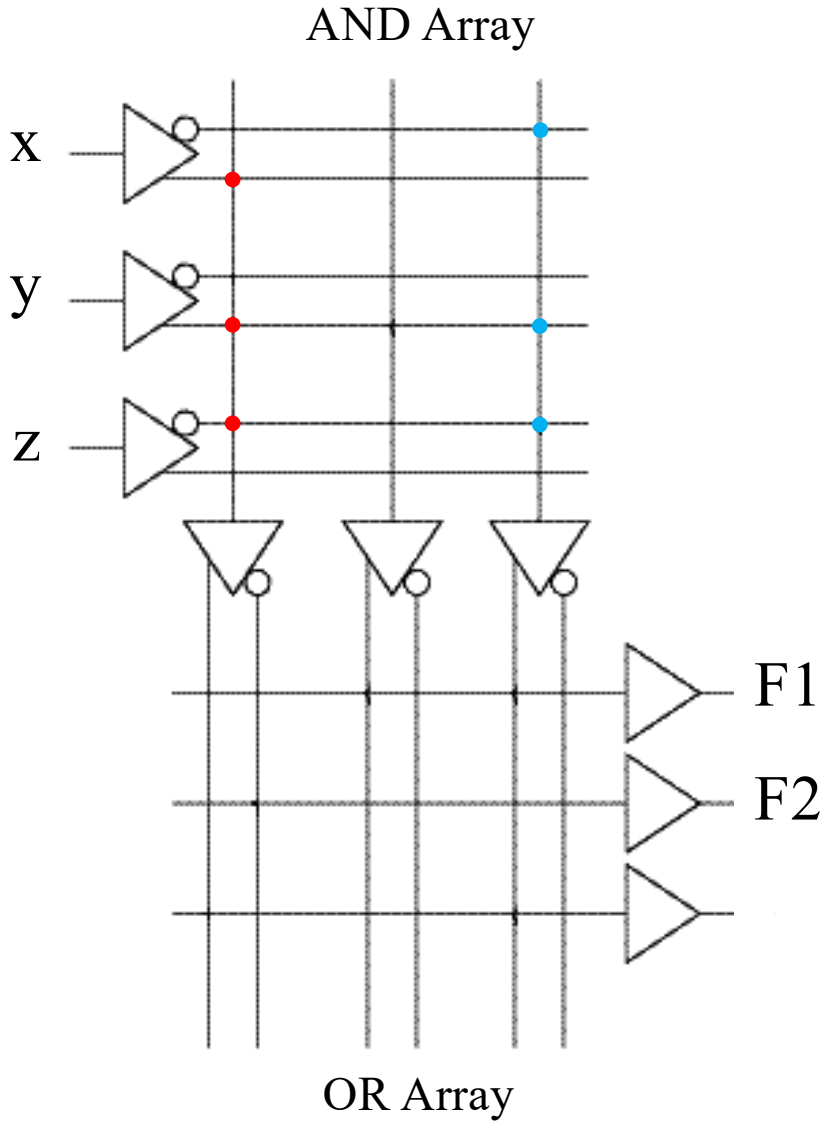


F1(x, y, z) = xyz' + xy'z'

F2(x, y, z) = x'yz' + xyz'



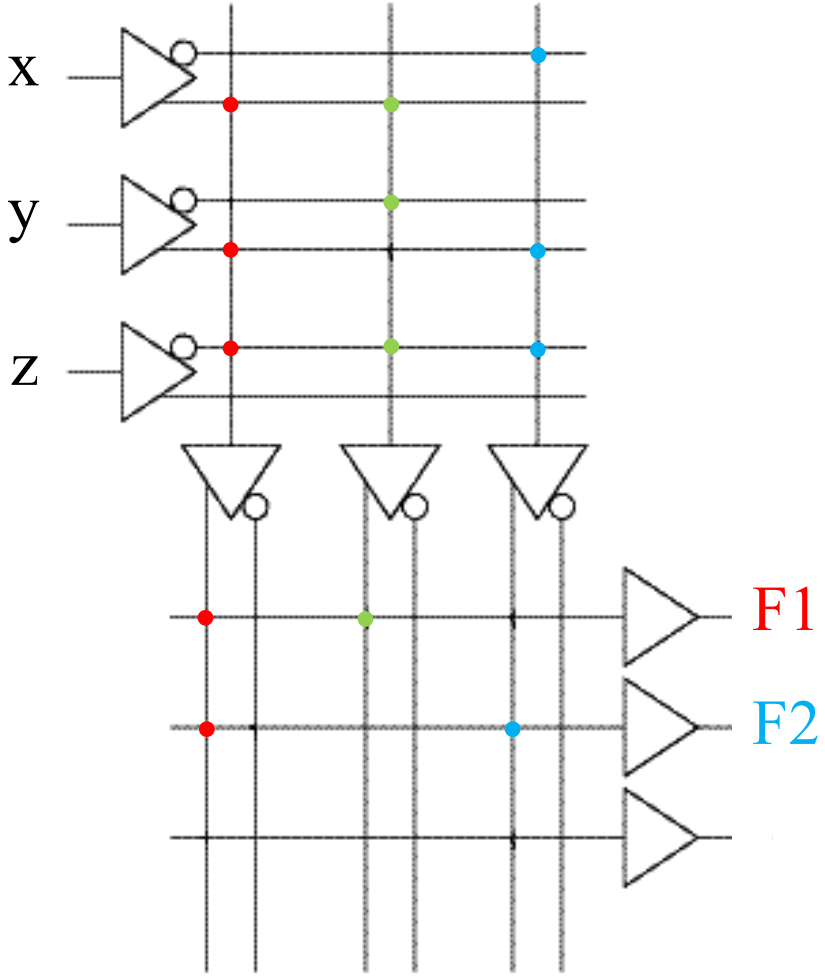
$$F1(x, y, z) = xyz' + xy'z'$$
$$F2(x, y, z) = x'yz' + xyz'$$



F1(x, y, z) = xyz' + xy'z'

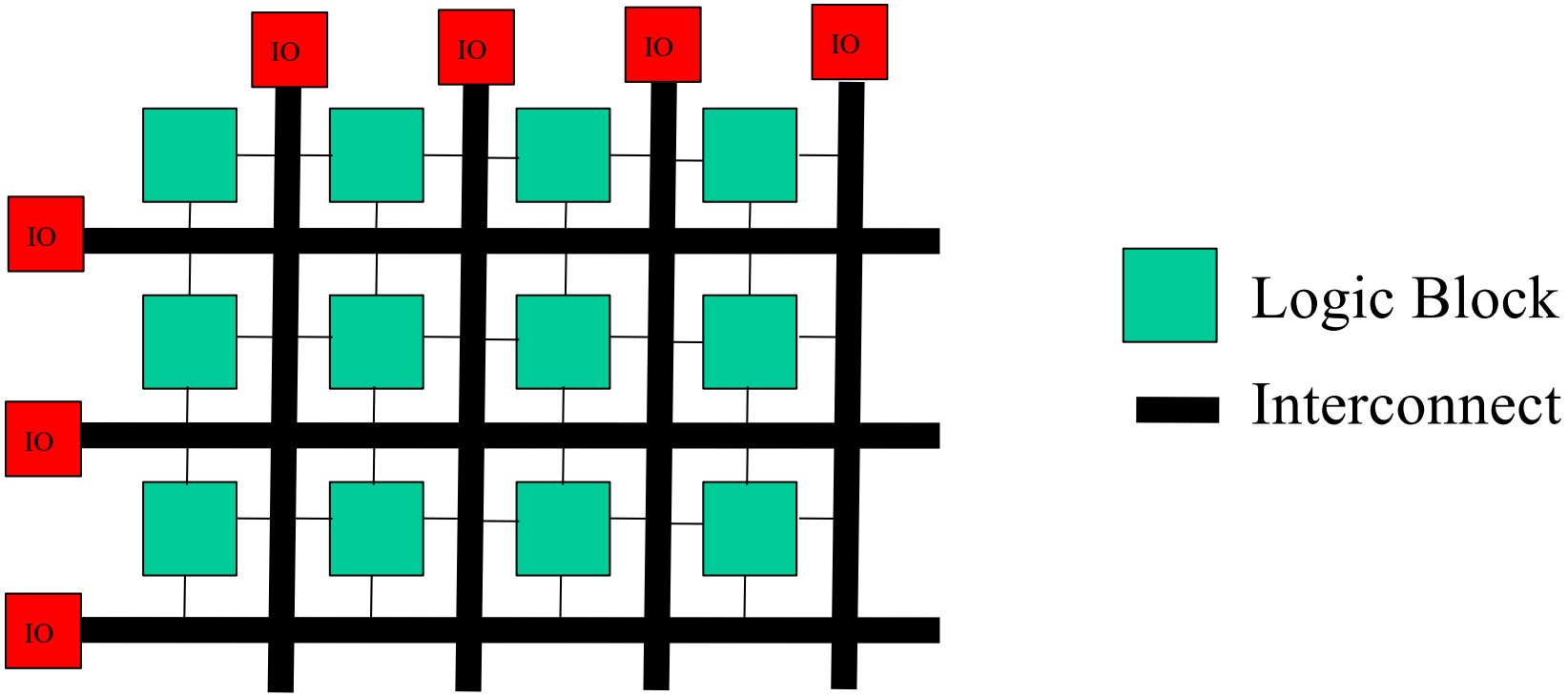
F2(x, y, z) = x'yz' + xyz'

AND Array

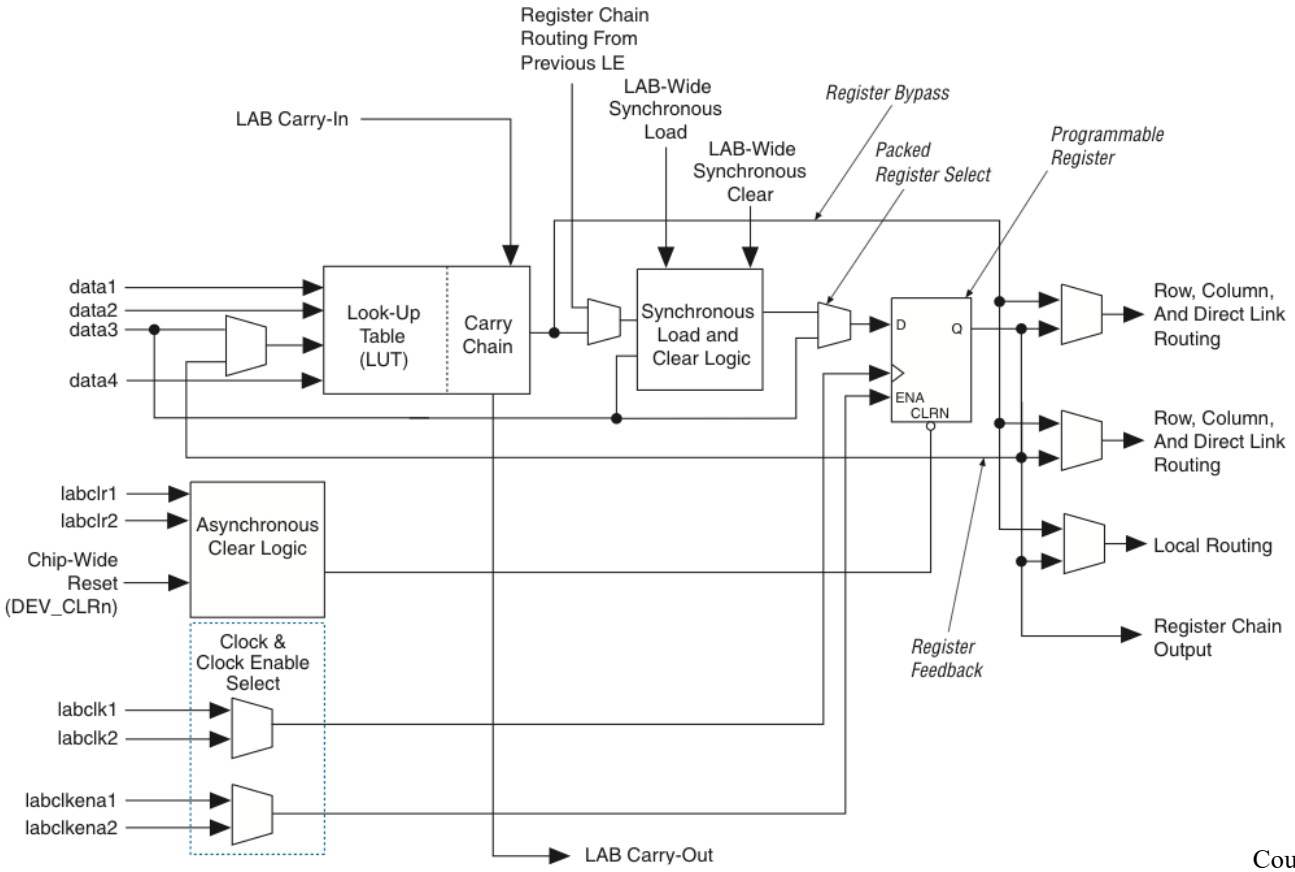


OR Array

FPGA



CPLD - Cyclone II LE



Courtesy of Altera Corp.

Hardware Description Language

- HDL
- Describe and simulate logic circuits
- Verilog HDL and VHDL

VHDL

- VHSIC Hardware Description Language.
- Very High Speed Integrated Circuits.
- Hardware description language used to model digital system from gate-level to algorithm.
- Integration of different programming language features – sequential programming languages + concurrent programming languages.

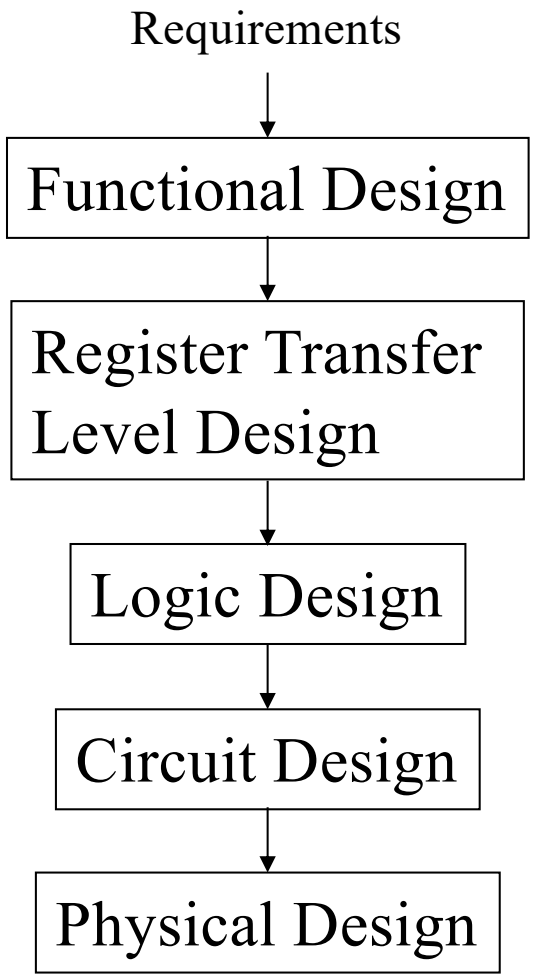
Capabilities

- Express concurrent or sequential behavior of digital systems.
- Model interconnection of components.
- Comprehensive description of digital system in a single model.

Brief History of VHDL

- 1981 DoD VHSIC program generated language requirements.
- 1985 team from IBM, TI, Intermetrics releases version 7.2 of VHDL.
- 1987 become industry standard, IEEE Std 1076-1987.
- 1993 upgraded, IEEE Std 1076-1993.
- IEEE 1164 VHDL Multivalue Logic (std_logic_1164) Packages

Top-Down Digital Design



Behavior Simulation

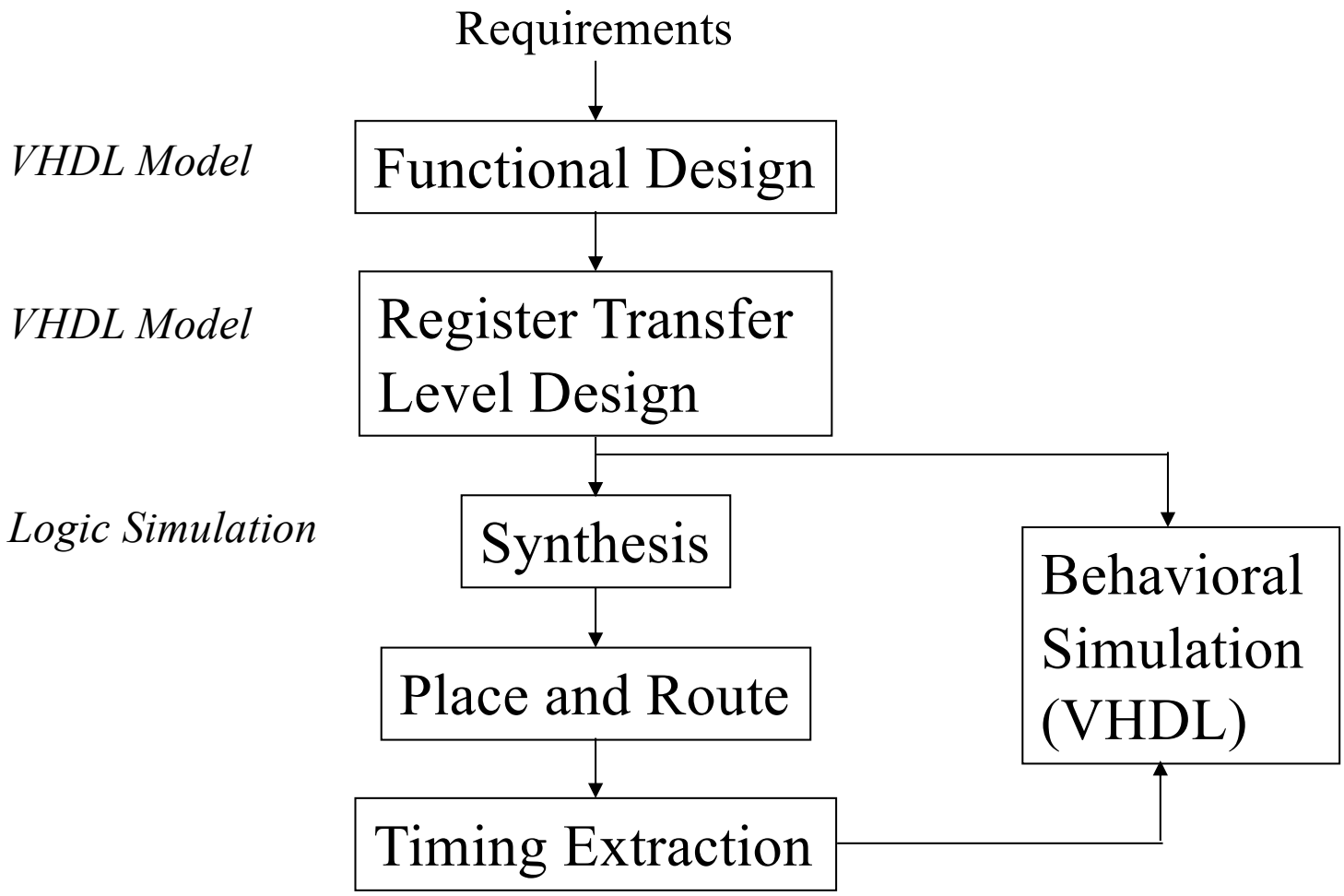
RTL Simulation Validation

Logic Simulation, Verification, Fault Simulation

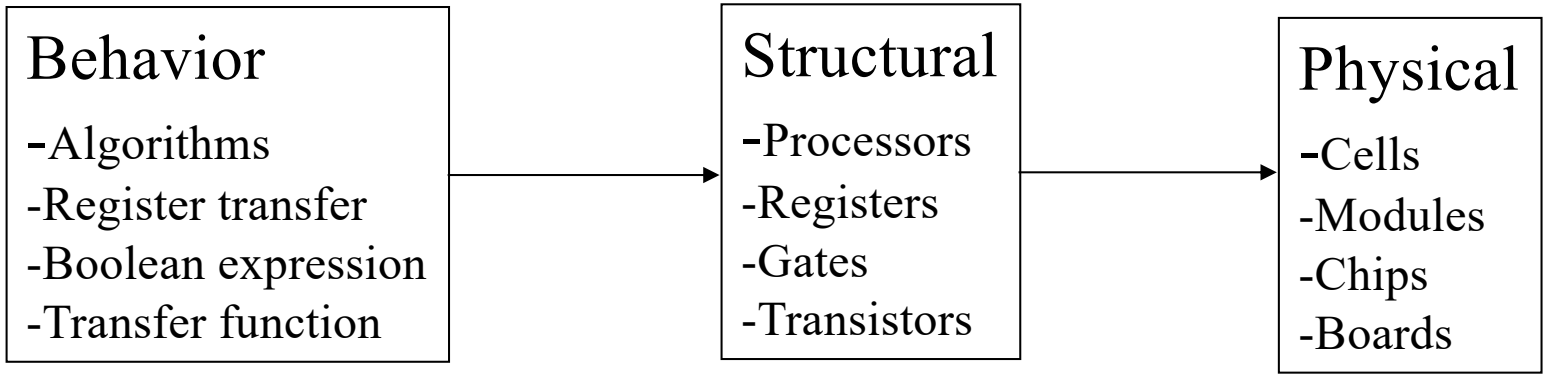
Timing Simulation
Circuit Analysis

Design Rule Checking

Synthesis Flow for PLD



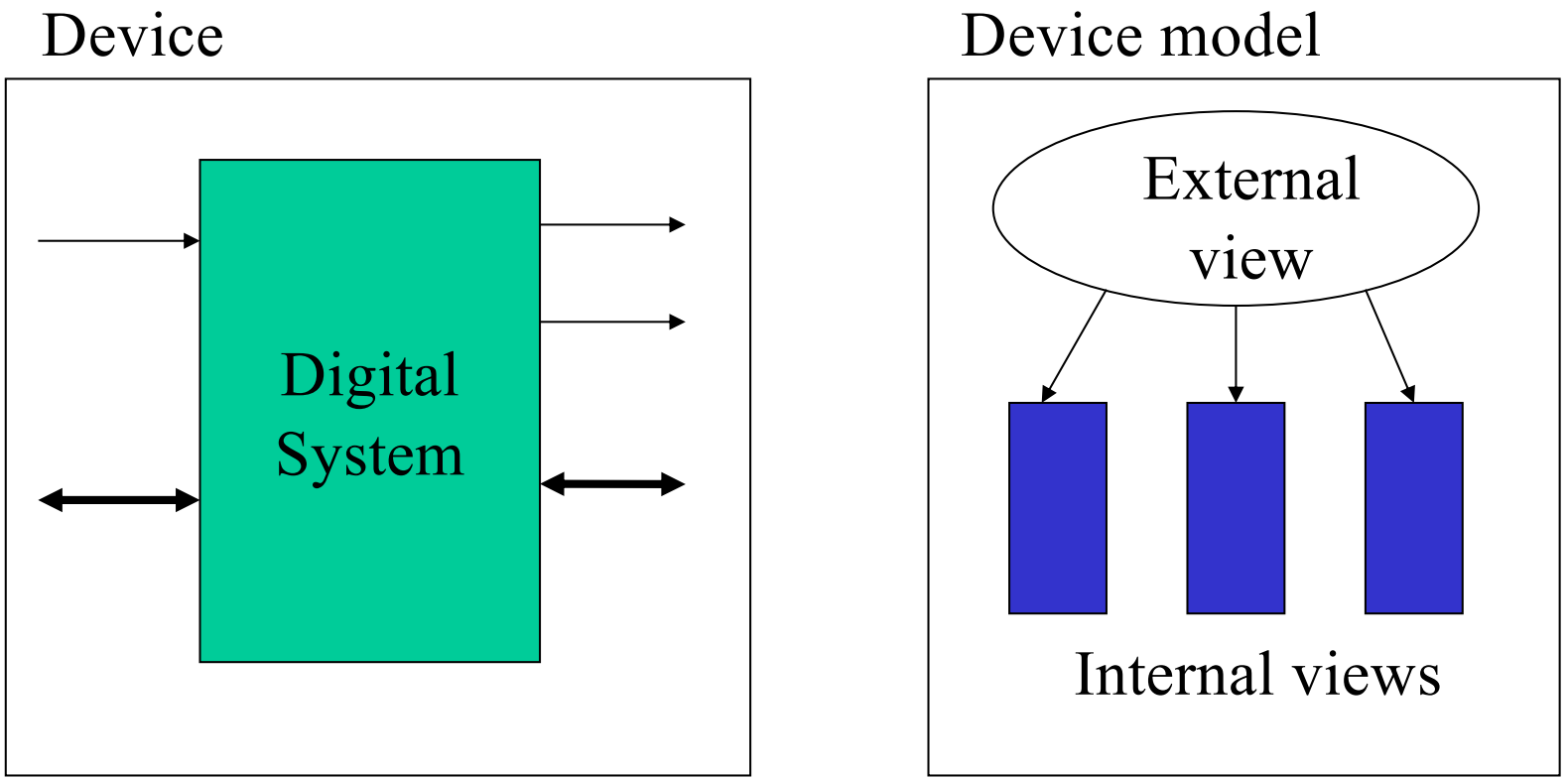
Different Design Views



Hardware Abstraction

- Model digital hardware devices.
- External view of device + one or more internal views.
- Internal view specifies the functionality or structure.
- External view specifies the interface.

Device versus Device Model

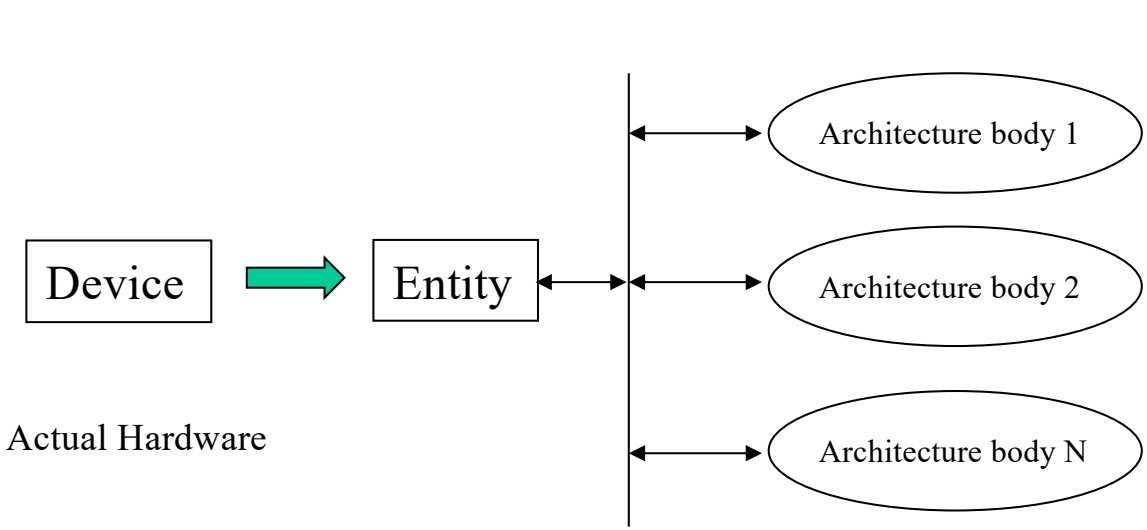


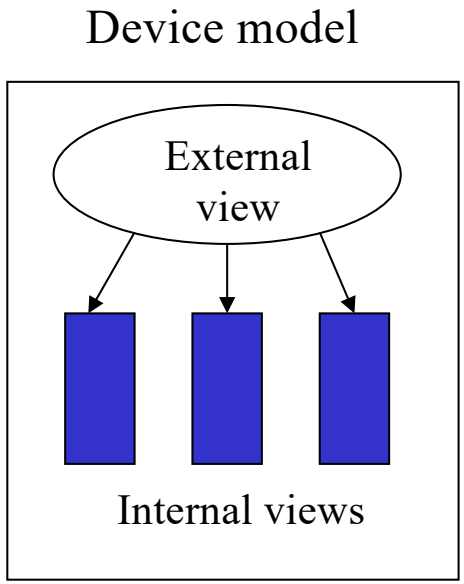
Basic Terminology

- Design unit
  - Entity declaration.
  - Architecture body.
  - Configuration declaration.
  - Package declaration.
  - Package body.

Design Unit

- Entity declaration + at least one Architecture body.





Design Unit

- Entity declaration = external view of the design unit, such as input and output signal names.
- Architecture body = internal description of the design unit
  - interconnected components represents the structure of the design.
  - a set of concurrent or sequential statements that represents the behavior of the design.

Identifiers

- A-Z, a-z, 0-9, _
- First character must be a letter
- Last character cannot be _
- Case insensitive

COUNT, count, COuNT refer to the same variable

DRIVE_BUS, CONST32_59, RAM_Address

Data Objects

- Holds a value of a specified type
- Four classes of data objects:
  - Constant: holds a single value of a given type, cannot be changed.
  - Variable: holds a single value of a given type, can be changed
  - Signal: holds a list of values, current and a set of possible future values
  - File: holds a sequence of values. Values can be read or written into the file.

Constants and Variables

- Analogous to those in programming languages
- Variables are programming objects

constant BUS_WIDTH: **integer** := 8;

variable done: **boolean**;
variable sum: **integer range** 0 **to** 100 := 10;
variable a: **bit** := '1';
variable control_status: **bit_vector** (10 **downto** 0);

Signals

- Signal has time value.
- The sequence of values assigned to a signal forms the waveform of the signal.
- Signal can declared to be of a specific type.
- Signals correspond to hardware objects.
- Signals are synthesized into wires or storage devices

Signals

- **signal** clock: **bit**;
- **signal** data_bus: **bit_vector**(0 **to** 7);
- **signal** init_a: **bit** := '0';
- **signal** init_a: **bit_vector**(3 **downto** 0) := "0001";
- **signal** init_b: **bit_vector**(0 **to** 7) := (0 => '1', **others** => '0');

Files

- File declaration

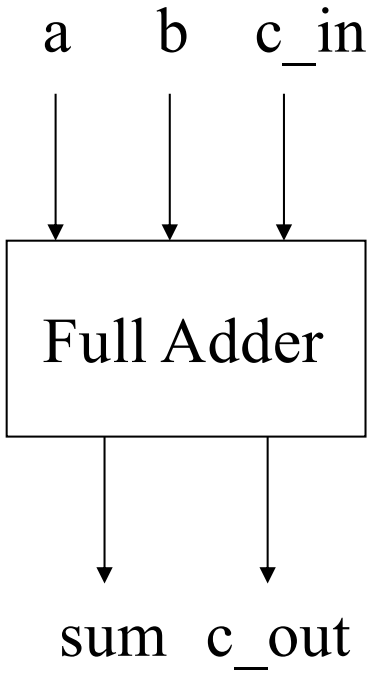
file file_name : file-type-name [[**open** mode] **is** string-expression];

file Stimulus: text **open** READ_MODE **is** "/usr/home/sim.sti";

Entity Declaration

- External view of the entity.
- Input and output signal names.

Full-adder




```
entity full_adder is
port (a, b, c_in : in bit;
      sum, c_out : out bit);
end full_adder;
```

type of signal

bit - basic signal type, can take on value 0 or 1

mode of signal

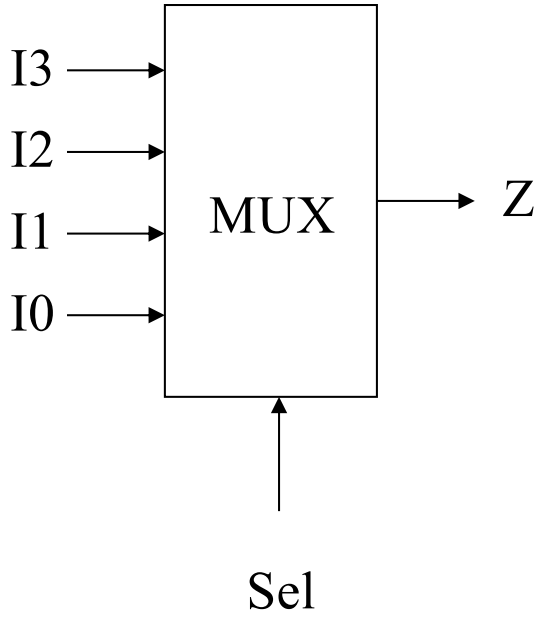
in - input

out – output

inout - input/output

buffer – output with read capability

4-to-1 Multiplexor




```
entity mux is
port (I0, I1, I2, I3 : in bit_vector (7 downto 0);
      Sel    : in bit_vector (1 downto 0);
      Z      : out bit_vector (7 downto 0));
end mux;
```

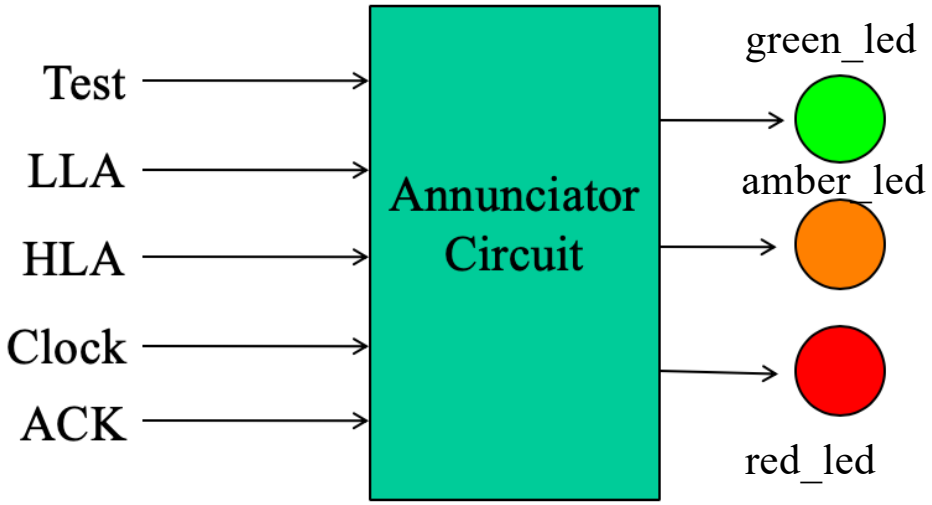
I0 - 8 bit vector, bit 7 is MSB, bit 0 is LSB

type of signal

bit_vector - vector of signals, each of type bit

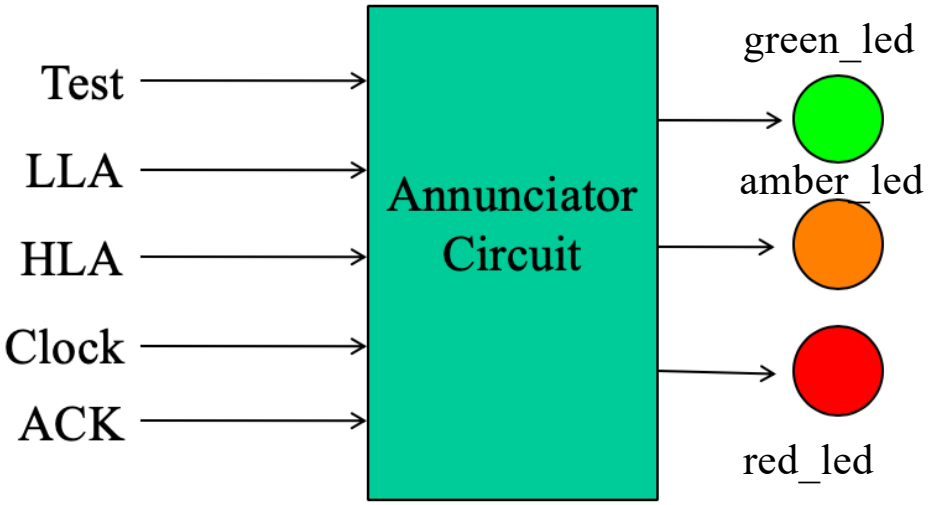
bit_vector(7 **downto** 0) - MSB = bit 7, LSB = bit 0
bit_vector(0 **to** 7) - MSB = bit 0, LSB = bit 7

Annuciator - Entity




```
VHDL CODE:
```

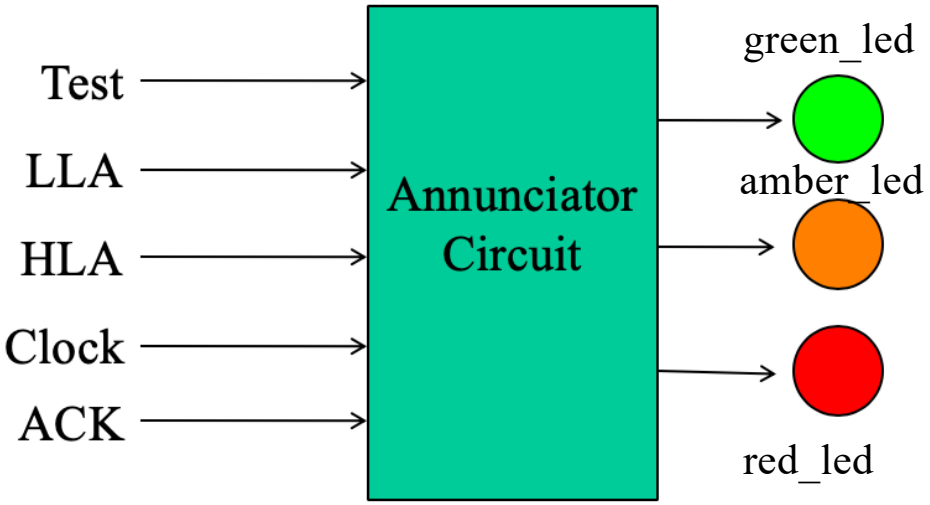
Annuciator - Entity



VHDL CODE:

```
entity annuciator is

end annuciator;
```

Annuciator - Entity



VHDL CODE:

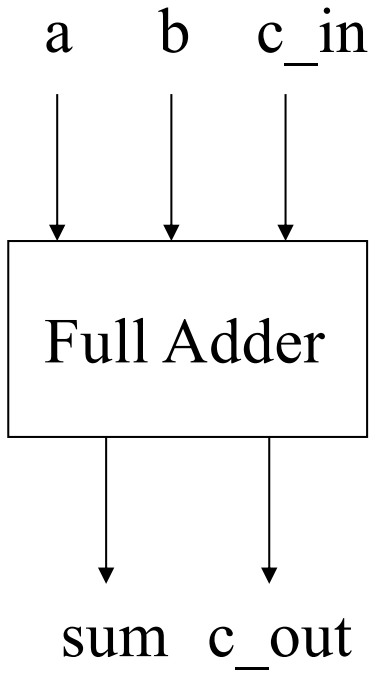
```
entity annuciator is
port (Test, LLA, HLA, Clock, ACK: in bit;
      green_led : out bit;
      amber_led : out bit;
      red_led : out bit);
end annuciator;
```

std_logic Signals

- Nine-valued logic system

Value	Meaning
U	Uninitialized
X	Forcing unknown
0	Forcing zero
1	Forcing one
Z	High impedance
W	Weak unknown
L	Weak zero
H	Weak one
-	Don't care

Full-adder

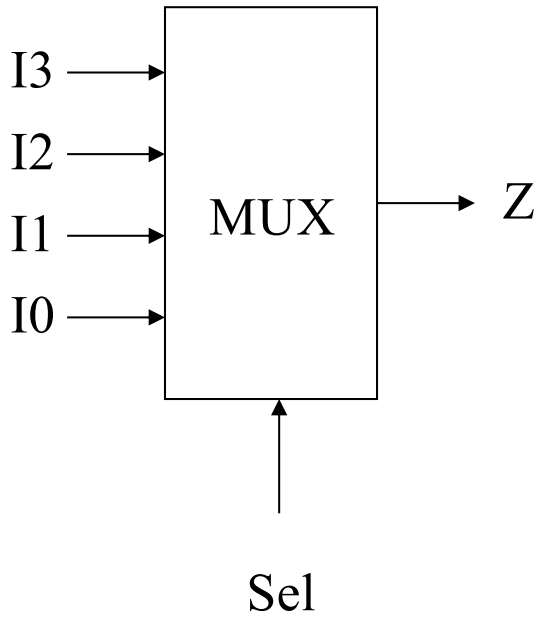



```
library IEEE;
use IEEE.std_logic_1164.all;

entity full_adder is
port (a, b, c_in : in std_logic;
      sum, c_out : out std_logic);
end full_adder;
```

IEEE 1164 standard

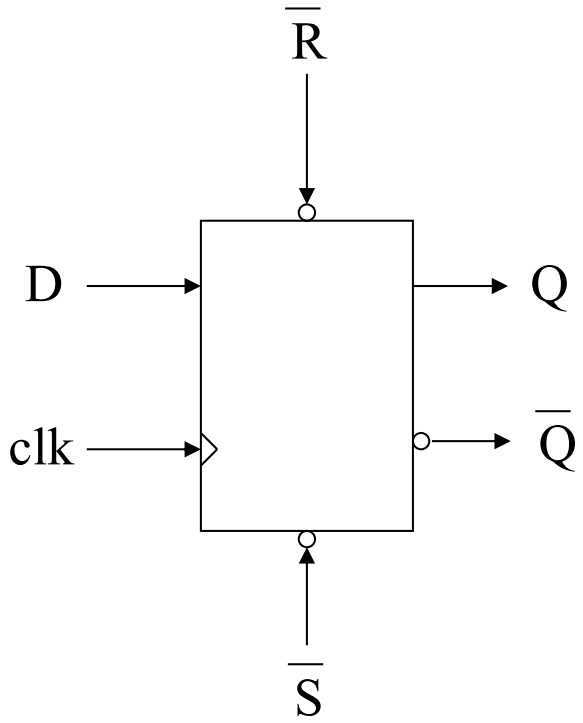
std_logic, std_logic_vector

4-to-1 Multiplexor



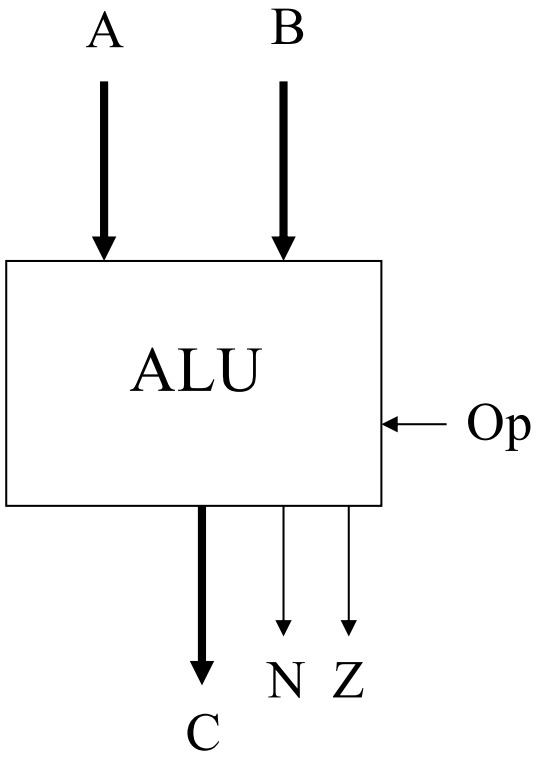

```
library IEEE;
use IEEE.std_logic_1164.all;

entity mux is
port (I0, I1 : in std_logic_vector (7 downto 0);
      I2, I3 : in std_logic_vector (7 downto 0);
      Sel    : in std_logic_vector (1 downto 0);
      Z      : out std_logic_vector (7 downto 0));
end mux;
```

D Flip-Flop




```
library IEEE;
use IEEE.std_logic_1164.all;

entity D_ff is
port (D, clk, Rbar, Sbar : in std_logic;
      Q, Qbar : out std_logic);
end D_ff;
```

64-bit ALU




```
library IEEE;
use IEEE.std_logic_1164.all;

entity ALU64 is
port (A, B : in std_logic_vector (63 downto 0);
      C    : out std_logic_vector (63 downto 0);
      Op   : in std_logic_vector (5 downto 0);
      N, Z : out std_logic);
end ALU64;
```

Behavior of Design Entity

- Declaration of the module name behavioral
- Description of the behavior of the full_adder

```
architecture behavioral of full_adder is
begin
  -- description of behavior
end behavioral;


-- comments
```

Signal Assignment

- Signal is assigned a specific value at a specific time.

```
signal_name <= expression1
```

- <= signal assignment operator.
- right side is evaluated and assigned to the signal.
- For example, a <= b **and** c;

Concurrent Signal Assignment

- Digital system operations are inherently concurrent.

```
a <= b and c;
d <= b or c;
```

- Order of statements does not imply order of execution.
- Concurrency is a natural part of the systems in VHDL

Concurrency

```
-- b = '0', c = '1'
c <= b;
d <= c;
```

- What is the value of d?

Review

VHDL is a _____ language

Entity declaration describe _____

Architecture body describe _____

There can be only ___ entity declaration, but ___ architecture body

Every line of VHDL code ends with a _____

The last statement in the port(…) does not have a ____

Signals in VHDL correspond to _____ in the circuit

You need to declare _____ library to use the std_logic type

Statements in the architecture body are executed _____

The operator for signal assignment is _____

Review

VHDL is a Hardware Description language

Entity declaration describe external view

Architecture body describe internal view

There can be only 1 entity declaration, but many architecture body

Every line of VHDL code ends with a ; (semicolon)

The last statement in the port(…) does not have a ; (semicolon)

Signals in VHDL correspond to wires in the circuit

You need to declare ieee library to use the std_logic type

Statements in the architecture body are executed concurrently

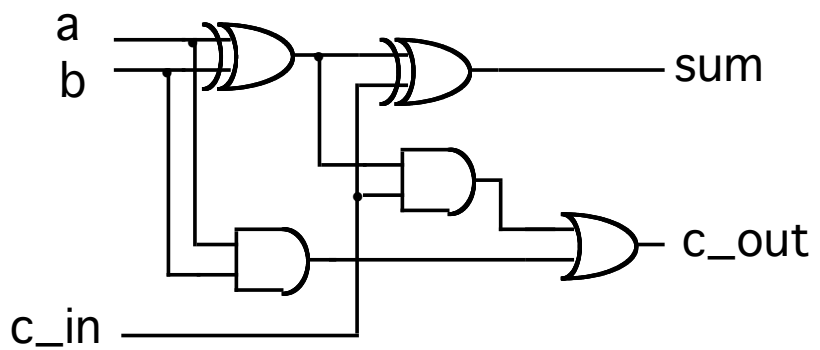
The operator for signal assignment is <=

Full Adder

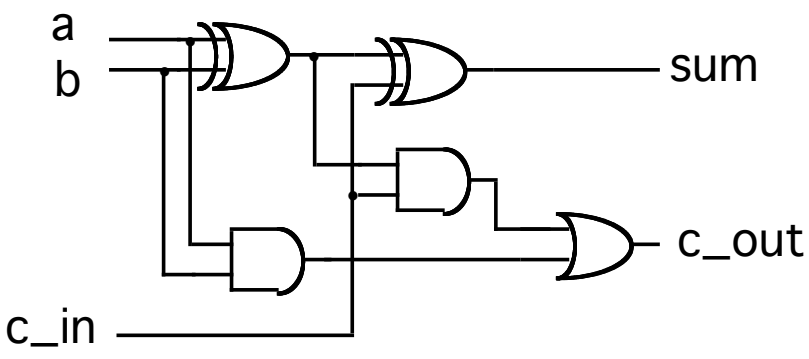
a	b	c_in	c_out	sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$sum = a \oplus b \oplus c\_in$$

$$c\_out = (a \oplus b)c\_in + ab$$



VHDL model of a full adder

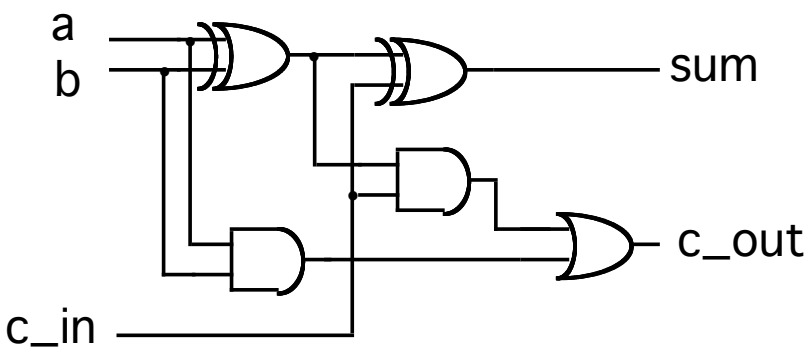


$$sum = a \oplus b \oplus c\_in$$

$$c\_out = (a \oplus b)c\_in + ab$$

```
library IEEE;
use IEEE.std_logic_1164.all;
entity full_adder is
port (a, b, c_in: in std_logic;
      sum, c_out: out std_logic);
end full_adder;

architecture dataflow of full_adder is
begin
L1: sum <= a xor b xor c_in;
L2: c_out <= ((a xor b) and c_in) or
                  (a and b);
end dataflow;
```

VHDL model of a full adder



$$sum = a \oplus b \oplus c\_in$$

$$c\_out = (a \oplus b)c\_in + ab$$

```
library IEEE;
use IEEE.std_logic_1164.all;
entity full_adder is
port (a, b, c_in: in std_logic;
      sum, c_out: out std_logic);
end full_adder;

architecture dataflow of full_adder is
begin
L1: sum <= a xor b xor c_in;
L2: c_out <= ((a xor b) and c_in) or
                (a and b);
end dataflow;
```

Label - optional

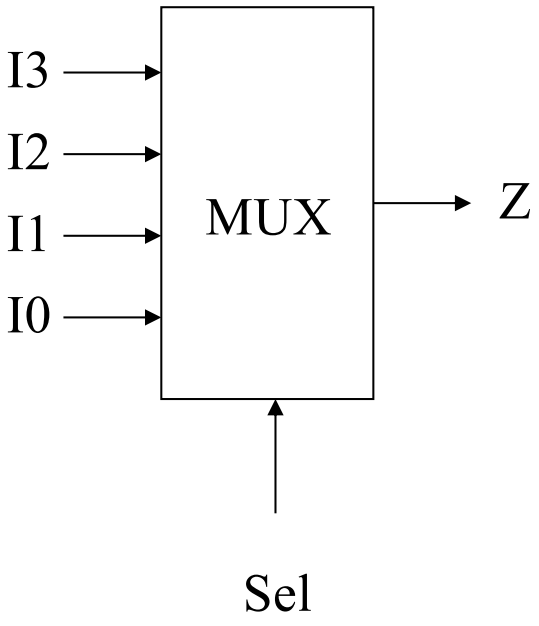
Conditional Signal Assignment

- Assign if condition is true

```
signal_name <= expression1 when condition1 else
               expression2 when condition2 else
               expression3;
```

- Conditions may overlap, assign first "true" condition
- Must have an "else" part

VHDL model of a 4-to-1 MUX




```
library IEEE;
use IEEE.std_logic_1164.all;
entity mux4 is
port (I0, I1: in std_logic_vector (7 downto 0);
      I2, I3: in std_logic_vector (7 downto 0);
      Sel   : in std_logic_vector (1 downto 0);
      Z     : out std_logic_vector (7 downto 0));
end mux4;

architecture dataflow of mux4 is
begin
Z <= I0 when Sel(0) = '0' and Sel(1) = '0' else
     I1 when Sel(0) = '1' and Sel(1) = '0' else
     I2 when Sel(0) = '0' and Sel(1) = '1' else
     I3 when Sel = "11" else "00000000";
end dataflow;
```

Signal Declaration

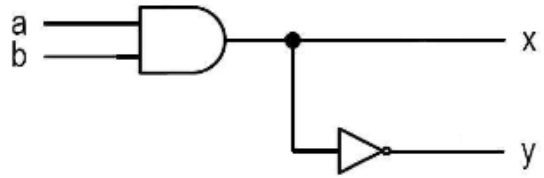
- Declared in architecture body

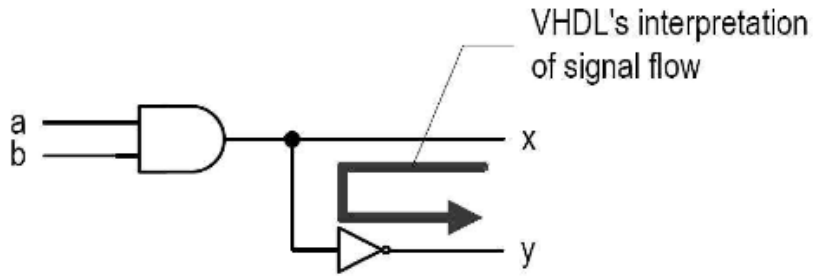
```
architecture dataflow of example is
  -- signal declarations
begin
    .
    .
end dataflow;
```

```
architecture dataflow of example is
  signal s1, s2, s3: bit;
  signal s4: bit_vector(7 downto 0);
  signal s5, s6: bit_vector(3 downto 0);
begin
   s1 <= s2;
   s3 <= s4(0);
   s5 <= s4(3 downto 0);
   s6 <= s4(1 to 4);
end dataflow;
```

Common Mistakes

```
library ieee;
use ieee.std_logic_1164.all;
entity demo is
  port( a, b: in std_logic;
        x, y: out std_logic);
end demo;

architecture wrong of demo is
begin
  x <= a and b;
  y <= not x;
end wrong;
```

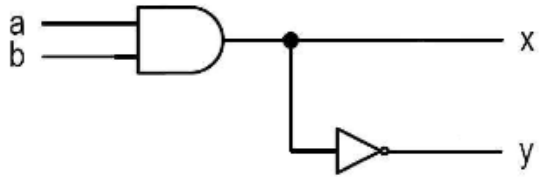




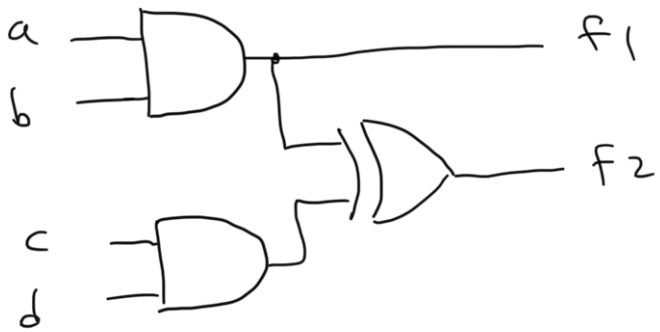
Error – since x is declared as an **out** signal

Correct

```
library ieee;
use ieee.std_logic_1164.all;
entity demo is
  port( a, b: in std_logic;
        x, y: out std_logic);
end demo;

architecture correct of demo is
begin
  x <= a and b;
  y <= not (a and b);
end correct;
```

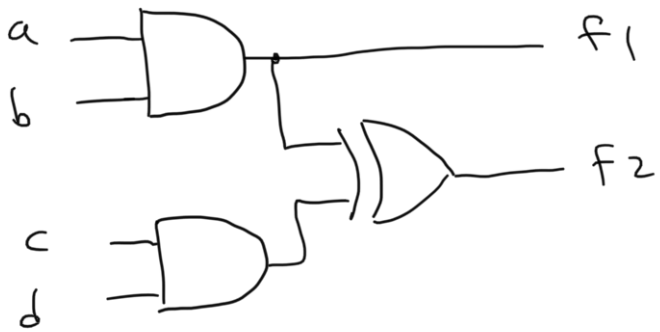


```
library ieee;
use ieee.std_logic_1164.all;
```



$f1 = a \cdot b$

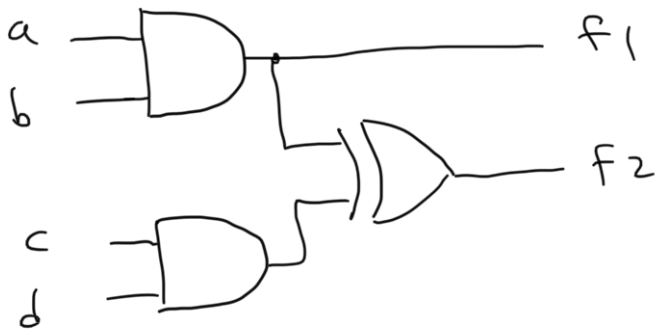
$f2 = (a \cdot b) \oplus (c \cdot d)$



$f1 = a \cdot b$

$f2 = (a \cdot b) \oplus (c \cdot d)$

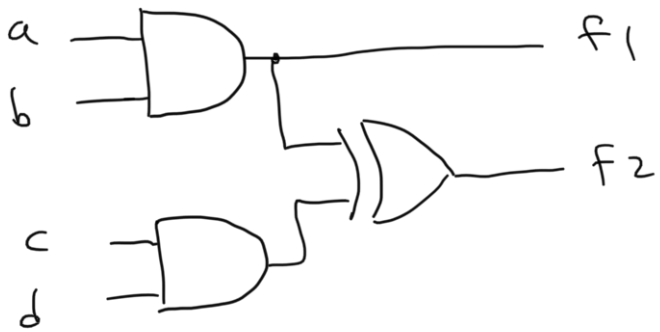
```
library ieee;
use ieee.std_logic_1164.all;

entity circuit is


end circuit;
```



$f1 = a \cdot b$

$f2 = (a \cdot b) \oplus (c \cdot d)$

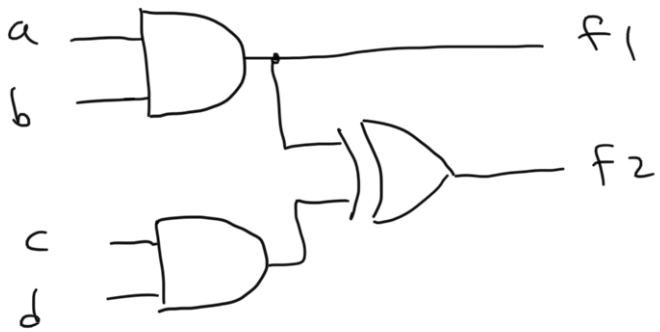
```
library ieee;
use ieee.std_logic_1164.all;

entity circuit is
  port (a, b, c, d: in std_logic;
        f1, f2: out std_logic);
end circuit;
```



$f1 = a \cdot b$

$f2 = (a \cdot b) \oplus (c \cdot d)$

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

entity circuit is
  port (a, b, c, d: in std_logic;
        f1, f2: out std_logic);
end circuit;

architecture dataflow of circuit is
begin



end dataflow;
```



$f1 = a \cdot b$

$f2 = (a \cdot b) \oplus (c \cdot d)$

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

entity circuit is
  port (a, b, c, d: in std_logic;
        f1, f2: out std_logic);
end circuit;

architecture dataflow of circuit is
begin
  f1 <= a and b;
  f2 <= (a and b) xor (c and d);

end dataflow;
```

Data Types

- A name associated with a set of values and a set of operators
- Predefined types: integer, boolean, bit, ...

Subtypes

- A type with constraints

```
subtype my_integer is integer range 48 to 100;
subtype your_integer is my_integer;

signal a: my_integer;
signal b: your_integer;
```

Enumeration

- A type with a set of user-defined values

```
type micro_op is (load, store, add, sub, mult);
subtype arith_op is micro_op range add to mult;
```

```
signal alu_control: arith_op;
```

Integer Type

```
integer range is -2147483648 to 2147483647;

type index is integer range 0 to 20;

subtype natural is integer range 0 to 2147483647;
subtype positive is integer range 1 to 2147483647;


signal count : integer range 0 to 100 := 0;
```

Floating Point Type

- 16.26, 0.02, 62.2E-3

real range is -1.79769E308 to 1.79769E308;

```
type TTL_voltage is real range -5.5 to -1.4;
variable a: TTL_voltage;
```

```
For modeling and simulation, synthesis tools generally cannot
directly translate REAL type operations into hardware
implementations. You need pre-designed IP cores for floating-
point arithmetic.
```

Physical Type

- Predefined type – time
  - supports femtoseconds (fs), picoseconds (ps), nanoseconds (ns), microseconds (us), milliseconds (ms), seconds (sec), minutes (min), and hours (hr).

```
signal_a <= signal_b after 10ns;
```

Time type is not synthesizable and is for simulation only

Array Type

- Contains elements of the same type

```
type ADDRESS is array (0 to 15) of bit;
type ROM is array (0 to 15, 0 to 15) of bit;
```

- Unconstrained array type (index range is open)

```
type STACK is array (integer range <>) of ADDRESS;
```

String Literals

- One dimensional array of characters
- "abc", "a""bc""d"

```
variable error_msg: string(1 to 20) := "Error";
```

- Hexdecimal, octal, binary bit string literals

```
X"FC10", O"237", B"00011011"

a_bit_vector <= X"F3";    a_bit_vector <= "11110011";
```

Record Types

- Grouping of different data types into a single structured object.
- Composed of elements of same or different data types

```
type CHIP_TYPE is (DIP, SOP, BGA);

type MODULE is record
   kind: CHIP_TYPE;
   n_inputs: positive;
   n_outputs: positive;
end record;
```

```
type CHIP_TYPE is (DIP, SOP, BGA);

type MODULE is
  record
   kind: CHIP_TYPE;
   n_inputs: positive;
   n_outputs: positive;
  end record;

variable NAND_GATE : MODULE;

NAND_GATE := (kind=>DIP, n_inputs=>4, n_outputs=>2);
NAND_GATE := (DIP, 4, 2);
NAND_GATE.n_inputs := 4;
```

Operators

- Logical operators
- Relational operators
- Shift operators
- Adding operators
- Multiplying operators
- Misc. operators

Logical Operators

- and, or, nand, nor, xor, xnor
- not
- use parentheses for ambiguous precedence

Relational Operators

- =, /=, <, <=, >, >=
- can be used for array to compare one element at a time
- "001" < "100" is true
- "abc" < "abd" is true
- "VHDL" < "VHDL97" is true

Shift/Rotate Operators

- sll, srl, sla, sra, rol, ror

```
signal a: bit_vector(6 downto 0) := "1001010";
signal b: bit_vector(6 downto 0);

1. b <= a sll 2; -- b = "0101000"
2. b <= a sra 2; -- b = "1110010"
3. b <= a rol 2; -- b = "0101010"
4. b <= a ror -2; -- b = "0101010"

rol 1 is the same as ror -1
```

Shifted in bits are shown in red

Adding Operators

- + (addition), - (subtraction), & (concatenation)
- '0' & '1' is "01"
- "VH" & "DL" is "VHDL"

```
signal a, b, c: bit;
signal bus: bit_vector(0 to 2);

bus <= a & b & c;

-- Instead of bus(0) <= a; bus(1) <= b; bus(2) <= c;
```

Multiplying Operators

- * (multiply)
- / (divide)
- mod (modulus) – same sign as divisor
- rem (remainder) – same sign as dividend
- A rem B = A - (A/B) * B
- A mod B = A - B * floor(A/B)

Misc. Operators

- abs (absolute)
- ** (exponential)

Process statement

- All statements are executed sequentially within the process block.
- Allow sequential statements similar to programming language such as C.
- Used to describe behavior that needs sequential control (like if-else, case, loops)

Process Statements

```
[process-label :] process [(sensitivity-list)] [is]
  [process-item-declarations]
begin
   wait statements
   variable assignment statements
   signal assignment statements
   if statements
   case statements
   loop statements
   null statements
   exit statements
   next statements
   assertion statements
   report statements
   procedure call statements
   return statements
end process [process-label];
```

- Sensitivity list

```
process (A, B, C, D)
```

  - list of signals that the process is sensitive.
  - whenever an event occurs on any of the signals in the sensitive list, the sequential statements within the process will be executed.
  - The sensitivity list of the process statement should contain all the signals which are being 'read' in that process.

```
process(X,Y)
begin
   Z <= X and Y;
end process;
```

- Variable assignment statement
  - Variables can only be used inside a process
  - Variables update immediately inside a process

```
variable-object := expression;
A := A + 1;
```

- Signal assignment statement
  - Signals update after the process suspends

```
signal-object <= expression;
S <= SA and SB;
```

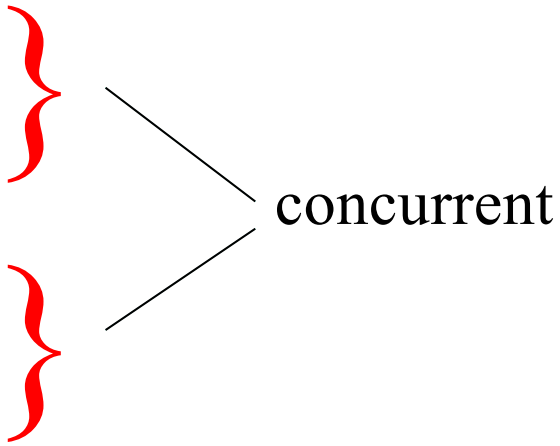
Sequential Statements

```
architecture sequential of circuit is
  signal Z: bit;
begin
  process (A, B, C, D)
    variable temp1, temp2: bit;
  begin
    temp1 := A and B;
    temp2 := C and D;
    Z <= not (temp1 or temp2);
  end process;
end sequential;
```

} sequential inside process

Concurrent

```
architecture sequential of circuit is
begin

  process (A, B, C, D)
    variable temp1, temp2: bit;
  begin
    ...
  end process;

  process (A, B, C, D)
    variable temp1, temp2: bit;
  begin
    ...
  end process;

end sequential;
```



Sequential Statements

```
architecture sequential of circuit is
  signal Z: bit
begin
  process (A, B, C, D)
    variable temp1, temp2: bit;
  begin
    temp1 := A and B;          -- Not synthesized
    temp1 := A xor B;          -- Only this is synthesized
    temp2 := C and D;
    Z <= not (temp1 or temp2);
  end process;
end sequential;
```

```
architecture behavior of test1 is
begin
  P1: process(A, B)
  begin
    z <= A;
    z <= B;
  end process P1;
end behavior;
```

OK

```
architecture behavior of test2 is
begin
  P1: process(A, B)
  begin
    z <= A;
  end process P1;
  z <= B;
end behavior;
```

Error!

```
architecture behavior of test1 is
begin
  P1: process(A, B)
  begin
    z <= A;
    z <= B;
  end process P1;
end behavior;
```

OK

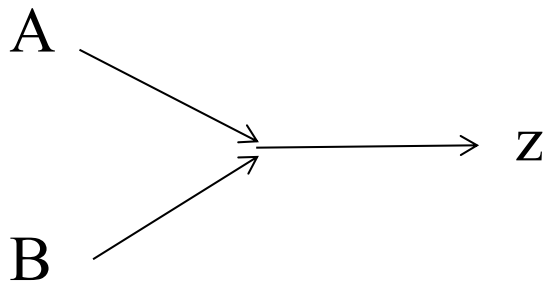
Statements are sequential inside process

```
architecture behavior of test2 is
begin
  P1: process(A)
  begin
    z <= A;
  end process P1;

  z <= B;
end behavior;
```

Concurrent

Error!

Process and the signal assignment are concurrent, same as applying two signals to z at the same time



- Combinational process - models combinational circuit. It must contain a sensitivity list with all the signals that are inputs to the process.

- Sequential process – model sequential circuit. If a process contains a wait statement, the process will be interpreted as a sequential process.

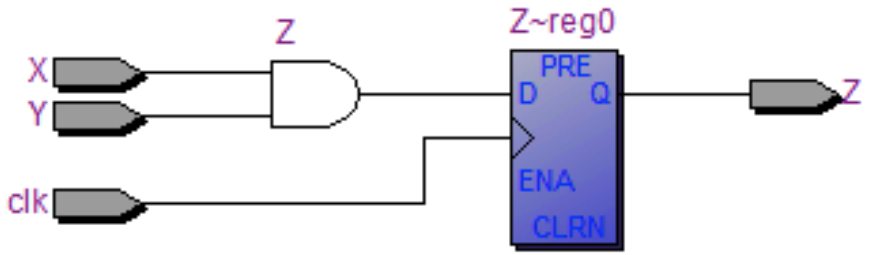
- Wait statement
  - execution suspended until the wait for event occurs

```
wait on sensitivity-list;
wait until boolean-expression;
wait for time-expression;
wait on sensitivity-list until boolean-expression
                for time-expression;

wait on A, B, C;
wait until A = B;
wait for 10 ns;
```

Process must have either a sensitivity list or a wait statement

Synthesis

```
entity example is
port (A, B, clk: in bit;
      Z      : out bit);
end example;

architecture proc of example
begin
   process
   begin
     wait until clk = '1';
     Z <= X and Y;
   end process;
end proc;
```



Flip-flops are added to the outputs

Process Statements

```
[process-label :] process [(sensitivity-list)] [is]
  [process-item-declarations]
begin
    wait statements
    variable assignment statements
    signal assignment statements
    if statements
    case statements
    loop statements
    null statements
    exit statements
    next statements
    assertion statements
    report statements
    procedure call statements
    return statements
end process [process-label];
```

- If statement

```
if boolean-expression then
    sequential-statements
[ elsif boolean-expression then
    sequential-statements ]
[ else
    sequential statemtents ]
end if;
```

```
if b <= 10 then
    b := b + 1;
end if;
```

```
if okay then
    in_a <= '0';
else
    in_a <= '1';
end if;
```

```
if b <= 10 then
    b := b + 1;
elsif b > 20 then
    b := b - 1;
else
    b := b - 2;
end if;
```

```
if b <= 10 then
    b := b + 1;
else if b > 20 then
        b := b - 1;
    else
        b := b - 2;
    end if;
end if;
```

- If statement

```
if boolean-expression then
    sequential-statements
[ elsif boolean-expression then
    sequential-statements ]
[ else
    sequential statemtents ]
end if;
```

```
if b <= 10 then
    b := b + 1;
end if;
```

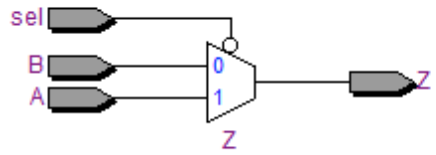
```
if okay then
    in_a <= '0';
end if;
```

```
if b <= 10 then
    b := b + 1;
elsif b > 20 then
    b := b - 1;
else
    b := b - 2;
end if;
```

```
if b <= 10 then
    b := b + 1;
else if b > 20 then
        b := b - 1;
    else
        b := b - 2;
    end if;
end if;
```

If-then-else Synthesis

```
entity example is
port (A, B, sel: in bit;
      Z      : out bit);
end example;

architecture proc of example
begin
    process(A, B, sel)
    begin

      if sel = '0' then
          Z <= a;
      else
          Z <= b;
      end if;

    end process;

end proc;
```



- Case statement

```
case expression is
  when choices => sequential-statements
[ when others => sequential-statements
    sequential-statements ]
end case;
```

Case statement must cover all possible values of expression

4-to-1 Multiplexer

```
library IEEE;
use IEEE.std_logic_1164.all;
entity MUX is
port (A, B, C, D: in std_logic; Sel: in std_logic_vector(0 to 1);
      Z: out std_logic);
end MUX;
architecture behavior of MUX is
begin
  P1: process(A, B, C, D, Sel)
  begin
    case Sel is
      when "00" => z <= A;
      when "01" => z <= B;
      when "10" => z <= C;
      when "11" => z <= D;
      when others => z <= 'X';
    end case;
  end process P1;
end behavior;
```

- Think of the if and case statement as routing structures rather than as sequential control constructs.
- An "if statement" infers a priority routing structure
- A "case statement" infers a multiplexing structure

Process Statements

```
[process-label :] process [(sensitivity-list)] [is]
  [process-item-declarations]
begin
    wait statements
    variable assignment statements
    signal assignment statements
    if statements
    case statements
    loop statements
    null statements
    exit statements
    next statements
    assertion statements
    report statements
    procedure call statements
    return statements
end process [process-label];
```

- Null statement

```
null;
```

causes no action.

```
if A = 0 then
  null;
else
 A := A + 1;
end if;
```

- Loop statement

```
[loop-label:] iteration-scheme loop
  sequential-statements
end loop;
```

```
iteration scheme 1:
```

```
for identifier in range loop
```

```
sum := 0;
for i in 1 to N loop
  sum := sum + i;
end loop;
```

- Think of a for loop statement as a mechanism to describe replicated structure

iteration scheme 2:

```
while boolean-expression loop
```

```
sum := 0;
i := 1;
while i <= N loop
  sum := sum + i;
  i := i + 1;
end loop;
```

iteration scheme 3:

```
loop

sum := 0;
i := 1;
loop
  sum := sum + i;
  i := i + 1;
  if i > N then
    exit;
  end if;
end loop;
```

Equivalent

```
process(A, B)
begin

   for i in 0 to 3 loop
      sum(i) <= A(i) xor B(i);
   end loop;

end process;
```

```
process(A, B)
begin

   sum(0) <= A(0) xor B(0);
   sum(1) <= A(1) xor B(1);
   sum(2) <= A(2) xor B(2);
   sum(3) <= A(3) xor B(3);

end process;
```

- Exit statement

```
exit [loop-label] [when condition];
```

can only be used inside a loop, exit the loop.

```
loop
  exit when A = B;
end loop;
```

```
L1: loop
  L2: loop
    exit L1 when A = B;
  end loop;
end loop;
```

- Next statement

```
next [loop-label] [when condition];
```

can only be used inside a loop, skips the remaining statements in the current iteration of the specified loop.

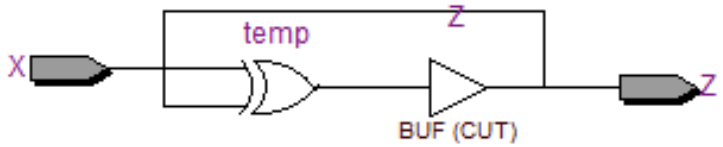
```
for i in 1 to N loop
  next when i = 3;
  sum := sum + i;
end loop;
```

Feedback

- Reading of a signal that is also assigned to in the same process
- In a combinational process, the previous value is an output of the combinational logic – the feedback is asynchronous.
- In a sequential process, the previous value is the value stored in a latch or register, -the feedback is synchronous.

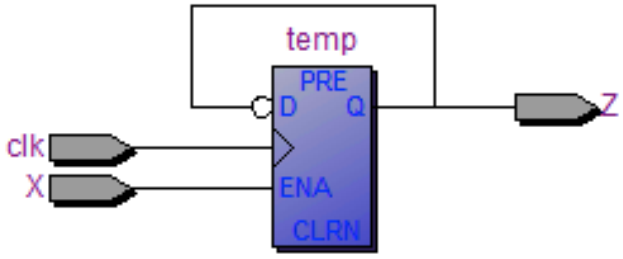
Asynchronous

```
entity example is
port (X: in bit;
      Z: out bit);
end example;

architecture proc of example
  signal temp: bit;
begin
    process(X)
    begin

      temp <= temp xor X;

    end process;

    Z <= temp;

end proc;
```



Synchronous

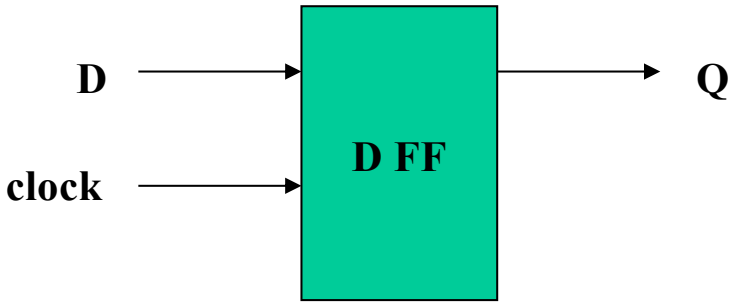
```
entity example is
port (X, clk: in bit;
      Z: out bit);
end example;

architecture proc of example
  signal temp: bit;
begin
    process
    begin

      wait until clk = '1';
      temp <= temp xor X;

    end process;

    Z <= temp;

end proc;
```



Clk	X	Temp	Z
0	*	0	0
0	*	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Example - D Flip-Flop

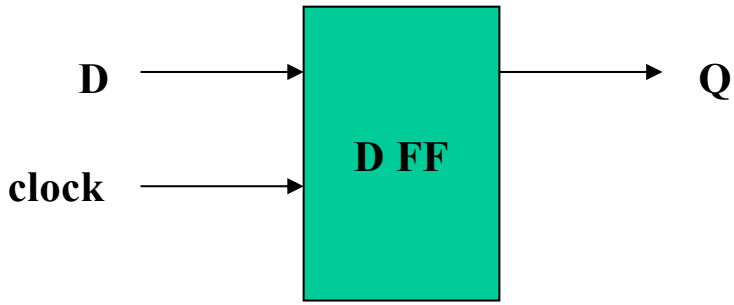
- Modeling behavior of a edge triggered D flip-flop
- Inputs: D, clock
- Output: Q



clock	D	Q
↑	0	0
↑	1	1

Entity

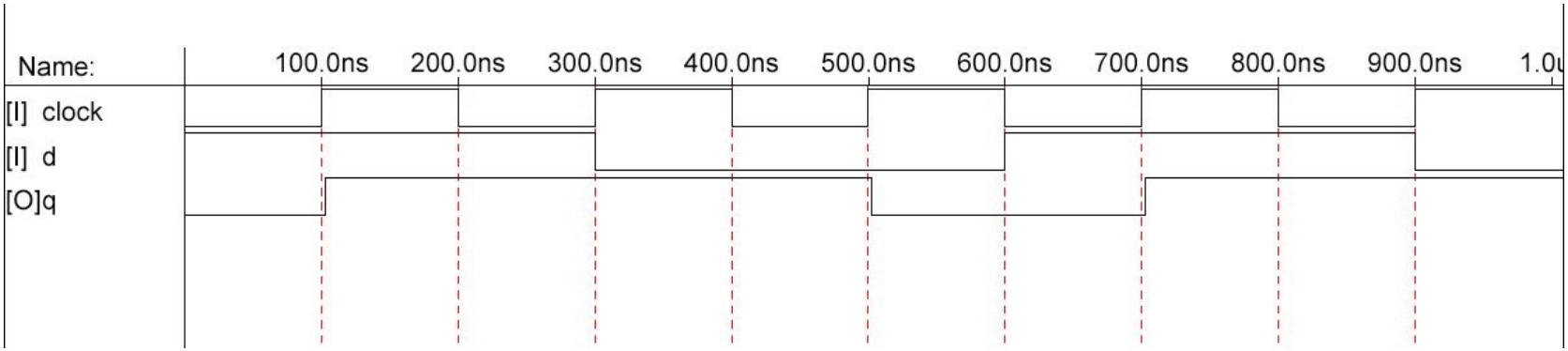
```
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;
```



D FF (Behavior)

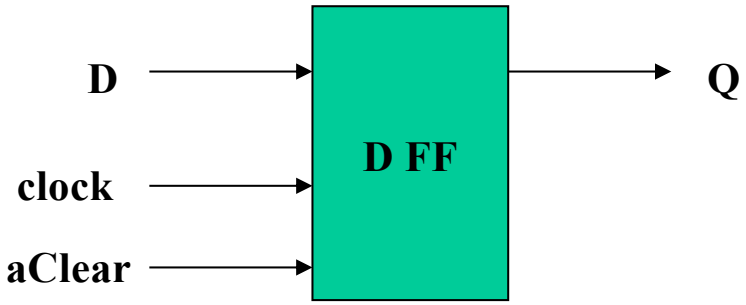
```
architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```

D FF Output


Name:
100.0ns
200.0ns
300.0ns
400.0ns
500.0ns
600.0ns
700.0ns
800.0ns
900.0ns
[I] clock
[I] d
[O]q

Example - D FF with Asynchronous Clear

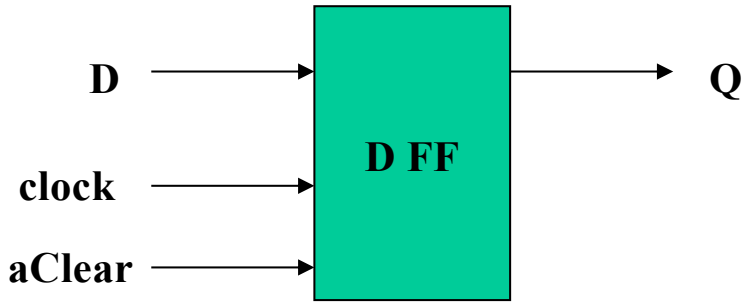
- Modeling behavior of a edge triggered D flip-flop with asynchronous clear
- Inputs: D, clock, aClear
- Output: Q



clock	aClear	D	Q
	1	X	0
↑	0	0	0
↑	0	1	1

Entity

```
library ieee;
use ieee.std_logic_1164.all;
entity d_ff_aclear is
  port (clock, d, aClear: in std_logic;
        q: out std_logic);
end d_ff_aclear;
```



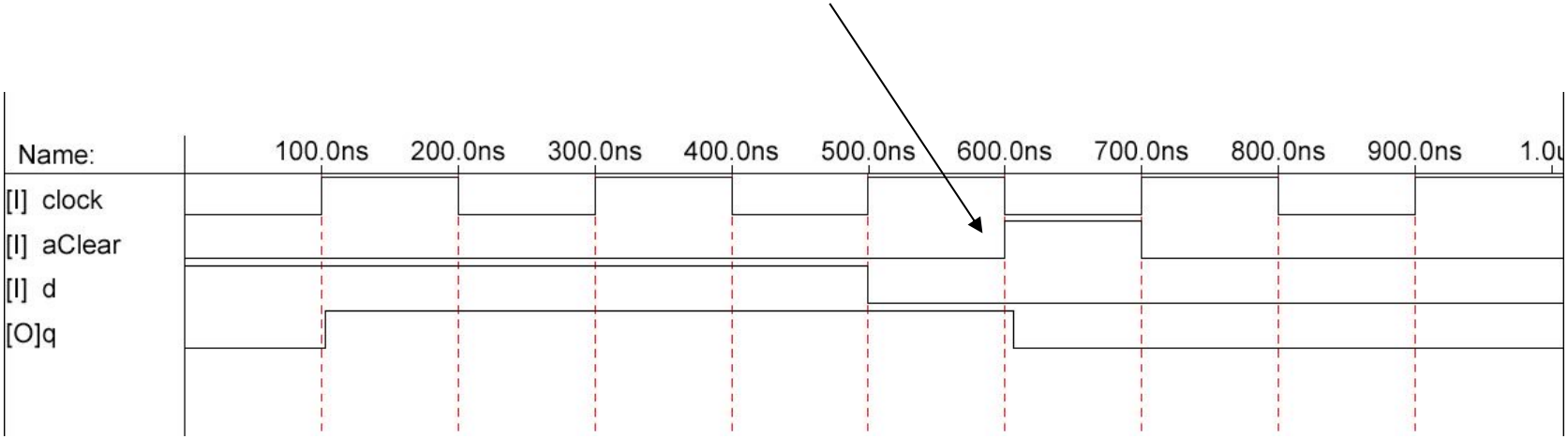
D FF with Asynchronous Clear

```
architecture behavior of d_ff_aclear is
begin
  process(aClear, clock)
  begin
    if aClear = '1' then
     q <= '0';
    elsif rising_edge(clock) then
      q <= d;
    end if;
  end process;
end behavior;
```

D FF with Asynchronous Clear

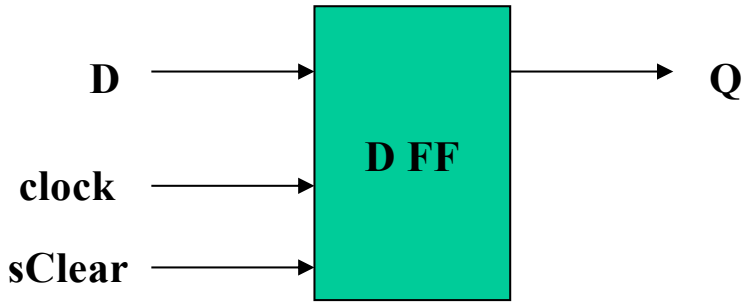
```
architecture behavior of d_ff_aclear is
begin
  process(aClear, clock)
  begin
    if aClear = '1' then
     q <= '0';
    elsif clock'EVENT and clock = '1' then
      q <= d;
    end if;
  end process;
end behavior;
```

D FF with Asynchronous Clear Output



Example - D FF with Synchronous Clear

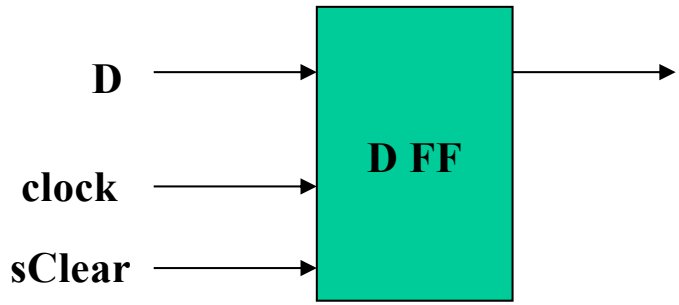
- Modeling behavior of a edge triggered D flip-flop with synchronous clear
- Inputs: D, clock, sClear
- Output: Q



clock	sClear	D	Q
↑	1	X	0
↑	0	0	0
↑	0	1	1

Entity

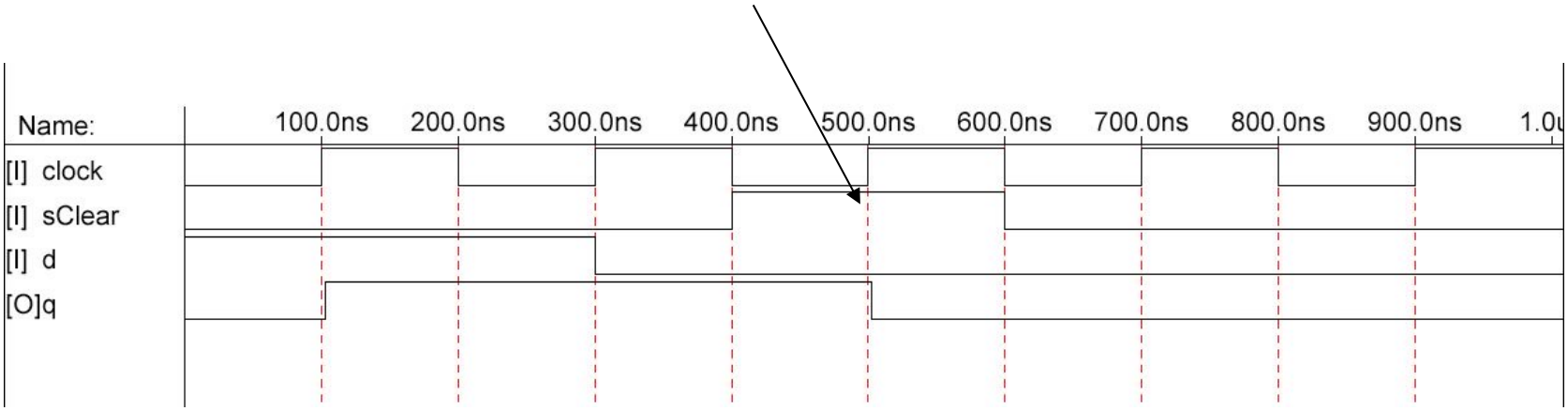
```
library ieee;
use ieee.std_logic_1164.all;
entity d_ff_sclear is
  port (clock, d, sClear: in std_logic;
        q: out std_logic);
end d_ff_sclear;
```



D FF with Synchronous Clear

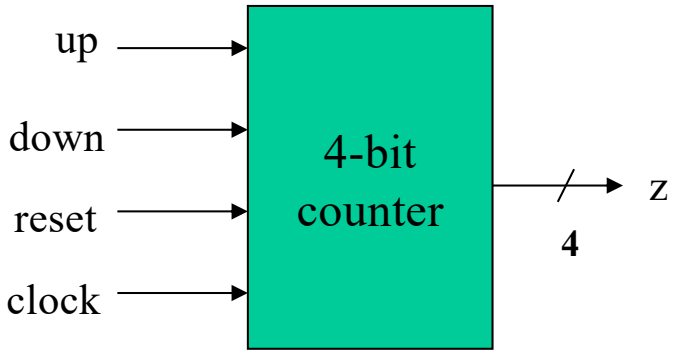
```
architecture behavior of d_ff_sclear is
begin
  process
  begin
    wait until(rising_edge(clock));
    if sClear = '1' then
     q <= '0';
    else
      q <= d;
    end if;
  end process;
end behavior;
```

D FF with Synchronous Clear Output



Example - 4-bit Counter

- 4-bit up/down counter
- Inputs: up, down, reset, clock
- Outputs: 4-bit count value

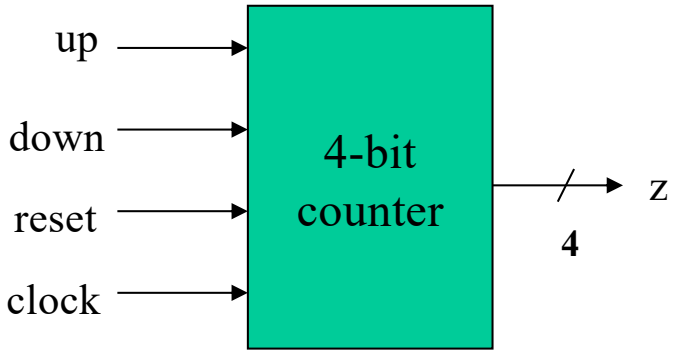


Functions

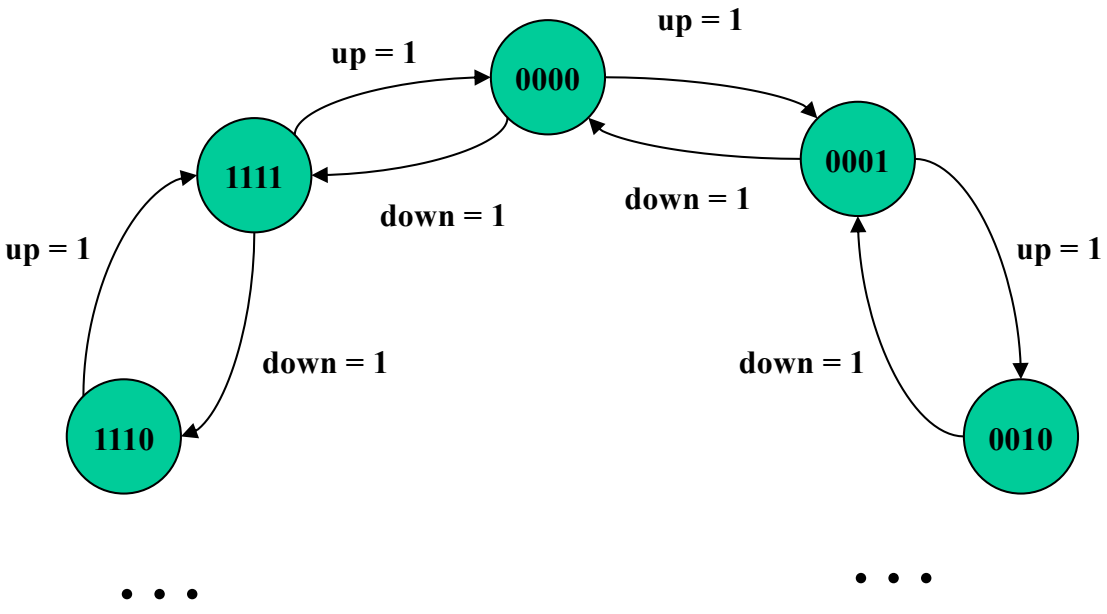
- Count on rising edge of clock
- Increment counter if up = 1 and down = 0
- decrement counter if up = 0 and down = 1

Entity

```
library ieee;
use ieee.std_logic_1164.all;
entity counter4 is
  port (reset, clock, up, down: in std_logic;
        z: out std_logic_vector(3 downto 0));
end counter4;
```



State Diagram



4-bit Counter (State Machine)

```
architecture state_machine of counter4 is
signal state: std_logic_vector(3 downto 0) := "0000";
signal nextState: std_logic_vector(3 downto 0);
begin
   clock_process: process
   begin
        wait until(rising_edge(clock));
        if reset = '1' then
                state <= "0000";
        else
                state <= nextState;
        end if;
   end process clock _process;

   z <= state;
```

4-bit Counter (State Machine)

```
ns_process: process(up, down, reset)
  begin
    if reset = '0' then
      case state is
        when "0000" =>
          if up = '1' and down = '0' then
            nextState <= "0001";
          elsif up = '0'and down = '1' then
            nextState <= "1111";
          else nextState <= "0000";
          end if;
        when "0001" =>
          if up = '1' and down = '0' then
            nextState <= "0010";
          elsif up = '0'and down = '1' then
            nextState <= "0000";
          else nextState <= "0001";
          end if;
        when "0010" => . . .

      end case;
    end if;
  end process ns_process;
end state_machine;
```

Counting Up

Present State				Next State			
Q_3	Q_2	Q_1	Q_0	NQ_3	NQ_2	NQ_1	NQ_0
0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0
0	0	1	0	0	0	1	1
0	0	1	1	0	1	0	0
0	1	0	0	0	1	0	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	1	1
0	1	1	1	1	0	0	0
1	0	0	0	1	0	0	1
1	0	0	1	1	0	1	0
1	0	1	0	1	0	1	1
1	0	1	1	1	1	0	0
1	1	0	0	1	1	0	1
1	1	0	1	1	1	1	0
1	1	1	0	1	1	1	1
1	1	1	1	0	0	0	0

$$NQ_3 = \overline{Q}_3 Q_2 Q_1 Q_0 + Q_3(\overline{Q}_2 + \overline{Q}_1 + \overline{Q}_0)$$

$$NQ_2 = Q_2\overline{Q}_1 + Q_2\overline{Q}_0 + \overline{Q}_2 Q_1 Q_0$$

$$NQ_1 = Q_1\overline{Q}_0 + \overline{Q}_1 Q_0$$

$$NQ_0 = \overline{Q}_0$$

Present State				Next State			
Q_3	Q_2	Q_1	Q_0	NQ_3	NQ_2	NQ_1	NQ_0
0	0	0	0	1	1	1	1
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	1
0	0	1	1	0	0	1	0
0	1	0	0	0	0	1	1
0	1	0	1	0	1	0	0
0	1	1	0	0	1	0	1
0	1	1	1	0	1	1	0
1	0	0	0	0	1	1	1
1	0	0	1	1	0	0	0
1	0	1	0	1	0	0	1
1	0	1	1	1	0	1	0
1	1	0	0	1	0	1	1
1	1	0	1	1	1	0	0
1	1	1	0	1	1	0	1
1	1	1	1	1	1	1	0

Counting Down

$$NQ_3 = \overline{Q}_3\overline{Q}_2\overline{Q}_1\overline{Q}_0 + Q_3(Q_2 + Q_1 + Q_0)$$

$$NQ_2 = Q_2Q_1 + Q_2Q_0 + \overline{Q}_2\overline{Q}_1\overline{Q}_0$$

$$NQ_1 = Q_1Q_0 + \overline{Q}_1\overline{Q}_0$$

$$NQ_0 = \overline{Q}_0$$

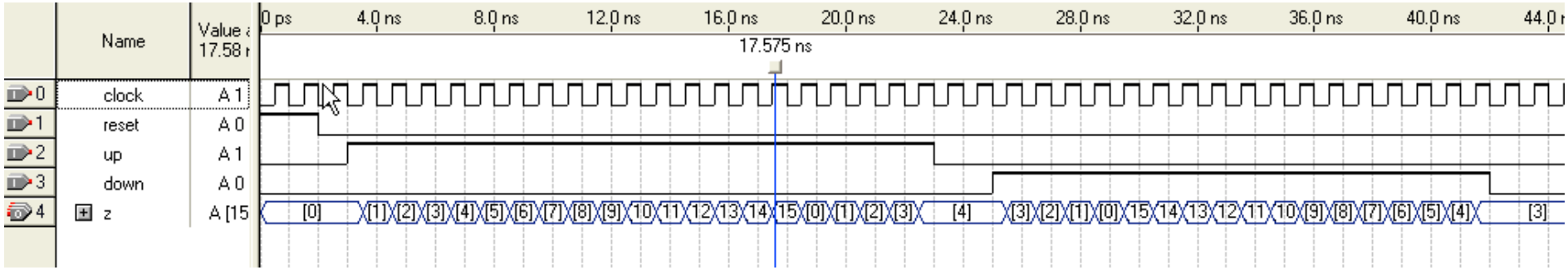
Combined Equations

$$NQ_3 = up\ \overline{down}(\overline{Q}_3Q_2Q_1Q_0 + Q_3(\overline{Q}_2 + \overline{Q}_1 + \overline{Q}_0))$$
$$+ \overline{up}\ down(\overline{Q}_3\overline{Q}_2\overline{Q}_1\overline{Q}_0 + Q_3(Q_2 + Q_1 + Q_0))$$
$$+ \overline{up}\ \overline{down}\ Q_3 + up\ down\ Q_3$$
$$NQ_2 = up(Q_2\overline{Q}_1 + Q_2\overline{Q}_0 + \overline{Q}_2Q_1Q_0)$$
$$+ down(Q_2Q_1 + Q_2Q_0 + \overline{Q}_2\overline{Q}_1\overline{Q}_0)$$
$$+ \overline{up}\ \overline{down}\ Q_2 + up\ down\ Q_2$$
$$NQ_1 = up(Q_1\overline{Q}_0 + \overline{Q}_1Q_0) + down(Q_1Q_0 + \overline{Q}_1\overline{Q}_0)$$
$$+ \overline{up}\ \overline{down}\ Q_1 + up\ down\ Q_1$$
$$NQ_0 = (up\ \overline{down} + \overline{up}\ down)\overline{Q}_0 + \overline{up}\ \overline{down}\ Q_0 + up\ down\ Q_0$$

4-bit Counter (Equation)

```
architecture equation of counter4 is
begin
 clock_process: process
  begin
    wait until(rising_edge(clock));
    if (reset = '1') then
      q <= "0000";
    else
      q <= nq;
    end if;
  end process clock_process;
 z <= q;
  nq(3) <= (up and down and q(3)) or (not up and not down and q(3)) or (up and not
    down and ((not q(3) and q(2) and q(1) and q(0)) or (q(3) and (not q(2) or not
    q(1) or not q(0))))or (down and not up and ((not q(3) and not q(2) and not
    q(1) and not q(0)) or (q(3) and (q(2) or q(1) or q(0)))));
  nq(2) <= (up and down and q(2)) or (not up and not down and q(2)) or (up and
    ((q(2) and not q(1)) or (q(2) and not q(0)) or (not q(2) and q(1) and q(0))))
    or (down and ((q(2) and q(1)) or (q(2) and q(0)) or (not q(2) and not q(1) and
    not q(0))));
  nq(1) <= (up and down and q(1)) or (not up and not down and q(1)) or (up and
    (q(1) xor q(0))) or (down and not (q(1) xor q(0)));
  nq(0) <= (up and down and q(0)) or (not up and not down and q(0)) or (((up and
    not down) or (not up and down)) and not q(0));
end equation;
```

4-bit Counter (Equation)



Behavior

- Use a variable `count` to keep track of count value
- `count` is an integer with values between 0 and 15

Counting Process

```
count_process: process
  variable count: integer range 0 to 15;
  begin
    wait until(rising_edge(clock));
    if reset = '1' then
       count := 0;
    elsif up = '1' and down = '0' then
      count := count + 1;
    elsif up = '0' and down = '1' then
      count := count - 1;
    end if;
     -- output the count value, how?
  end process count_process;
```

Output

- `count` is a variable not a signal
- Need to convert into a 4-bit std_logic_vector
- Use function
  - conv_std_logic_vector(parameter, size)
  - parameter: can be of type integer, unsigned, signed, std_logic
  - size: is the number of bits
- Function is in std_logic_arith package

Counting Process

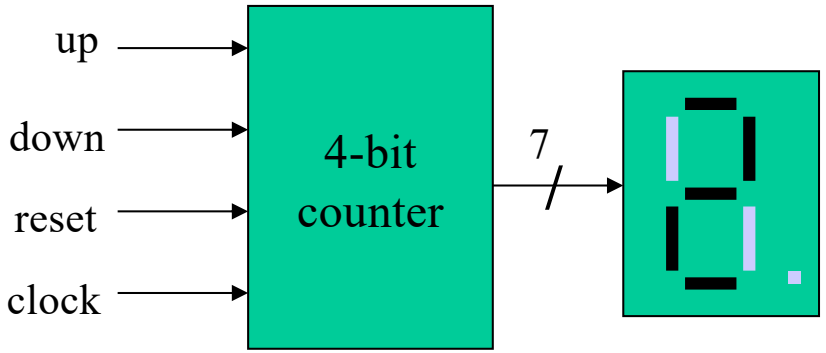
```
count_process: process
  variable count: integer range 0 to 15;
  begin
    wait until(rising_edge(clock));
    if reset = '1' then
       count := 0;
    elsif up = '1' and down = '0' then
      count := count + 1;
    elsif up = '0' and down = '1' then
      count := count - 1;
    end if;
     z <= conv_std_logic_vector(count, 4);
  end process count_process;
```

4-bit Counter (Behavior)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity counter4 is
port (reset, clock, up, down: in std_logic;
      z: out std_logic_vector(3 downto 0));
end counter4;

architecture behavior of counter4 is
begin
count_process: process
  variable count: integer range 0 to 15;
  begin
    wait until(rising_edge(clock));
    if reset = '1' then
          count := 0;
    elsif up = '1' and down = '0' then
      count := count + 1;
    elsif up = '0' and down = '1' then
      count := count - 1;
    end if;
      z <= conv_std_logic_vector(count, 4);
  end process count_process;
end behavior;
```

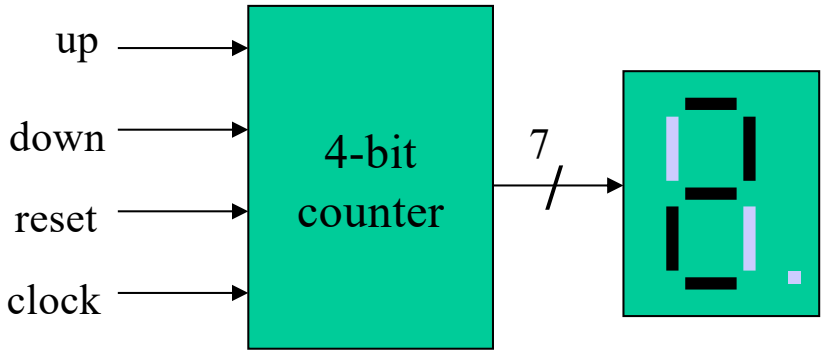
Seven Segment Display

- Display the counter value on a seven segment display
- Change `output` to a 7-bit vector (0-6)
- Active low



Entity

```
entity counter4display is
  port (reset, clk, up, down: in std_logic;
        output: out std_logic_vector(0 to 6));
end counter4display;
```



4-bit Counter With Display

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity counter4 is
port (reset, clock, up, down: in std_logic;
      z: out std_logic_vector(0 to 6));
end counter4;

architecture behavior of counter4 is
begin
count_process: process
    variable count: integer range 0 to 15;
  begin
    wait until(rising_edge(clock));
    if reset = '1' then
          count := 0;
    elsif up = '1' and down = '0' then
      count := count + 1;
    elsif up = '0' and down = '1' then
      count := count - 1;
    end if;
    -- output to seven segment display
  end process count_process;
end behavior;
```

4-bit Counter With Display

```
-- output to seven segment display
case count is   -- 0123456
  when  0 => z <= "0000001";
  when  1 => z <= "1001111";
  when  2 => z <= "0010010";
  when  3 => z <= "0000110";
  when  4 => z <= "1001100";
  when  5 => z <= "0100100";
  when  6 => z <= "0100000";
  when  7 => z <= "0001111";
  when  8 => z <= "0000000";
  when  9 => z <= "0000100";
  when 10 => z <= "0001000";
  when 11 => z <= "1100000";
  when 12 => z <= "0110001";
  when 13 => z <= "1000010";
  when 14 => z <= "0110000";
  when 15 => z <= "0111000";
  when others => z <= "1111111";
end case;
```

Data Types and Operators

The *std_logic_1164* package supports conversions between *bit* and *std_logic* as well as between *bit_vector* and *std_logic_vector*.

Function	Data type of a	Data type of result
`to_bit(a)`	`std_logic`	`bit`
`to_stdulogic(a)`	`bit`	`std_logic`
`to_bitvector(a)`	`std_logic_vector`	`bit_vector`
`to_stdlogicvector(a)`	`bit_vector`	`std_logic_vector`

Example

```
signal s1, s2, s3: std_logic_vector(7 downto 0);
signal b1, b2: bit_vector(7 downto 0);

s1 <= b1;
-- bit_vector assigned to std_logic_vector

b2 <= s1 and s2;
-- std_logic_vector assigned to bit_vector

s3 <= b1 and s2;
-- and is undefined for bit_vector and std_logic_vector

We can use the conversion functions to correct these problems
s1 <= to_stdlogicvector(b1);
b2 <= to_bitvector(s1 and s2);
s3 <= to_stdlogicvector(b1) and s2;
```

Numerical Data

- IEEE numeric_std package
  - Allows array of bits to be interpreted as an unsigned or signed number
- Two new data types
  - **unsigned** and **signed**
  - both are defined as an array of elements with std_logic data type
  - Signed data are interpreted as a number in 2's complement representation

Example

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

signal x : signed(15 downto 0);
Signal y : unsigned(15 downto 0);
```

Operators

- *numeric_std* package supports the arithmetic operations, **abs**, *, /, **mod**, **rem**, + and -
- These operators can operands with data types
  - *unsigned* and *unsigned*
  - *unsigned* and *natural*
  - *signed* and *signed*
  - *signed* and *integer*
- Correct usage

  signal a, b, c, d, e: unsigned(7 **downto** 0);

  a <= b + c;

  d <= b + 1;

  e <= (5 + a + b) - c;
- Note that the sum "wraps around" when overflow occurs as in physical adder

Data Type Conversion

- Conversion can be carried out by a type conversion function or by type casting
- There are three *type conversion functions* in *numeric_std* package
  - to_unsigned(), to_signed() and to_integer()
- The function *to_integer* converts from data types *unsigned* or *signed*
- The functions *to_unsigned* and *to_signed* convert from an integer data type to a specific number of bits (second parameter)
- *Type casting* is also possible between 'closely related' data types.

From data type	To data type	Conversion function
unsigned, signed,	std_logic_vector	std_logic_vector(a)
signed, std_logic_vector	unsigned	unsigned(a)
unsigned, signed	integer	to_integer(a)
natural	unsigned	to_unsigned(a, size)
integer	signed	to_signed(a, size)

Example

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

signal u1, u2: unsigned(7 downto 0);
signal v1, v2: std_logic_vector(7 downto 0);

u1 <= unsigned(v1);
v2 <= std_logic_vector(u2);
```

Example

```
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6: unsigned(3 downto 0);
signal s7: signed(3 downto 0);

u3 <= u2 + u1;
u4 <= u2 + 1;
```

Correct ?

```
u5 <= s7;
u6 <= 5;
```

Correct ?

Correction

```
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6: unsigned(3 downto 0);
signal s7: signed(3 downto 0);

u5 <= unsigned(s7);
u6 <= to_unsigned(5, 4);
```

Example

```
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6: unsigned(3 downto 0);
signal s7: signed(3 downto 0);
```

```
u7 <= s7 + u1;
```

Correct?

```
s3 <= u3;
s4 <= 5;
```

Correct?

Example

```
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6: unsigned(3 downto 0);
signal s7: signed(3 downto 0);
```

Wrong

```
u7 <= s7 + u1;   -- + undefined over the types
```

Correct

```
u7 <= unsigned(s7) + u1;
```

Wrong

```
s3 <= u3; -- type mismatch
s4 <= 5; -- type mismatch
```

Correct

```
s3 <= std_logic_vector(u3); -- type casting
s4 <= std_logic_vector(to_unsigned(5,4));
```

Example

```
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6: unsigned(3 downto 0);
signal s7: signed(3 downto 0);
```

```
s5 <= s1 + s2;
s6 <= s2 + 1;
```

Correct?

Example

```
signal s1, s2, s3, s4, s5, s6: std_logic_vector(3 downto 0);
signal u1, u2, u3, u4, u5, u6: unsigned(3 downto 0);
signal s7: signed(3 downto 0);
```

Wrong

```
        s5 <= s1 + s2; -- '+' undefined over std_logic_vector
        s6 <= s2 + 1; -- '+' undefined
```

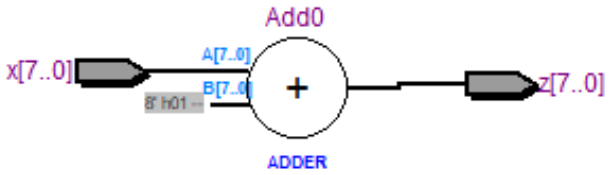
Correct

```
        s5 <= std_logic_vector(unsigned(s1) + unsigned(s2));
        s6 <= std_logic_vector(unsigned(s2) + 1);
```

Using unsigned type

```
library ieee;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;
entity adder is
port (x: in unsigned(7 downto 0);
      z: out unsigned(7 downto 0)
     );
end adder;

architecture behavior of adder is
begin
        z <= x + 1;
end behavior;
```

Error if you use std_logic_vector type



Other Counters

- Looked at binary counters
- Lab. looked at BCD counters
- Other counters:
  - Ring counter
  - Linear feedback shift register

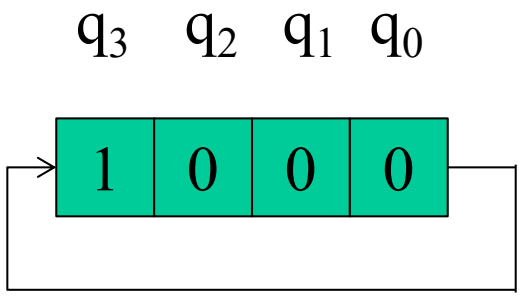
Ring Counter

- Circulates a single '1' around the ring
- For example, 4-bit ring counter

State	Q3	Q2	Q1	Q0
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	1	0	0	0

4-bit Ring Counter

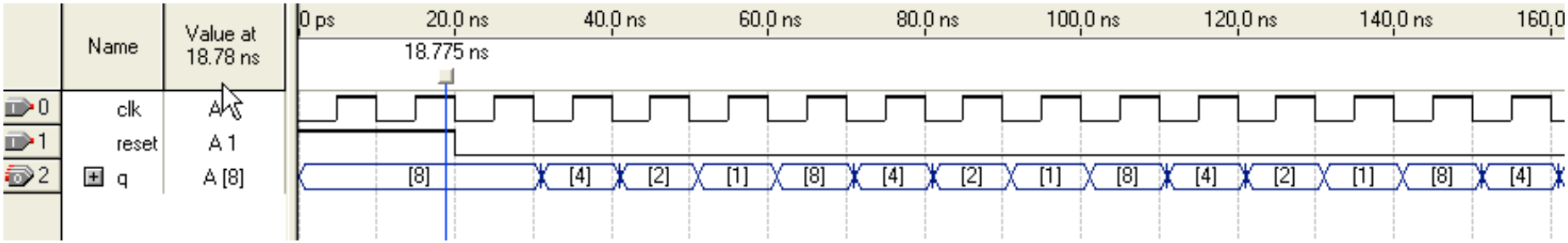
```
library ieee;
use ieee.std_logic_1164.all;

entity RingCounter4Bit is
  port (reset, clk: in std_logic;
        q: out std_logic_vector(3 downto 0));
end RingCounter4Bit ;

architecture behavior of RingCounter4Bit is
   signal ps, ns: std_logic_vector(3 downto 0);
begin
   process(reset, clk)
   begin
          if reset = '1' then
              ps <= (3 => '1', others => '0');
          elsif clk'event and clk = '1' then
              ps <= ns;
          end if;
   end process;

  ns <= ps(0) & ps(3 downto 1);
  q <= ps;
end behavior;
```



"1000"

Ring Counter Output

- q = "1000", "0100", "0010", "0001"



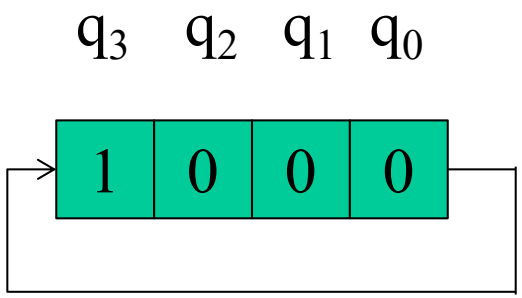
generic declaration

- Add parameters to the model by using generic declaration
- Declared inside the entity declaration
- generic (*identifier*: *type* := *default_value*);
- generic (n: integer := 4);

4-bit Ring Counter

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

entity RingCounter4Bit is
  port (reset, clk: in std_logic;
        q: out std_logic_vector(3 downto 0));
end RingCounter4Bit ;

architecture behavior of RingCounter4Bit is
   signal ps, ns: std_logic_vector(3 downto 0);
begin
   process(reset, clk)
   begin
          if reset = '1' then
              ps <= (3 => '1', others => '0');
          elsif clk'event and clk = '1' then
              ps <= ns;
          end if;
   end process;

  ns <= ps(0) & ps(3 downto 1);
  q <= ps;
end behavior;
```

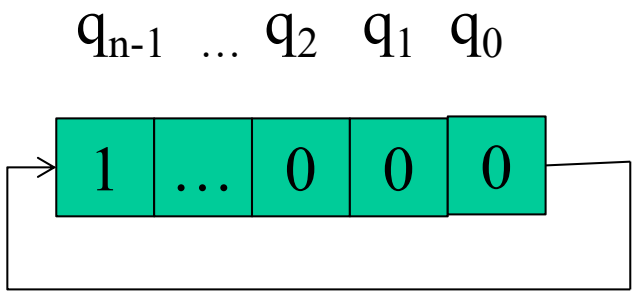
Add generic parameter n



Replace with n-1

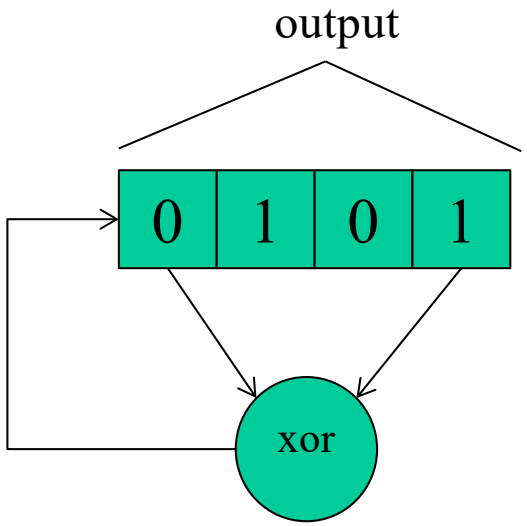
4-bit Ring Counter

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

entity RingCounter4Bit is
  generic (n: integer := 4);
  port (reset, clk: in std_logic;
        q: out std_logic_vector(n-1 downto 0));
end RingCounter4Bit ;

architecture behavior of RingCounter4Bit is
   signal ps, ns: std_logic_vector(n-1 downto 0);
begin
   process(reset, clk)
   begin
          if reset = '1' then
              ps <= (n-1 => '1', others => '0');
          elsif clk'event and clk = '1' then
              ps <= ns;
          end if;
   end process;

  ns <= ps(0) & ps(n-1 downto 1);
  q <= ps;
end behavior;
```



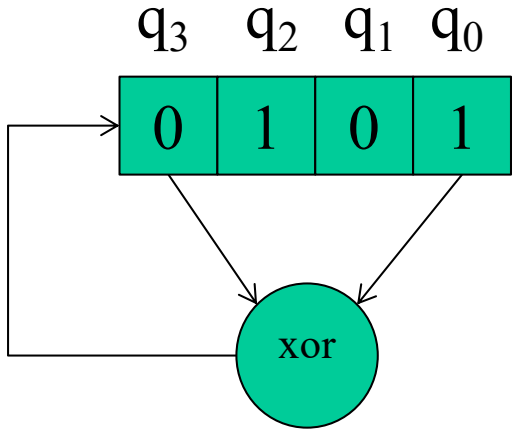
Linear Feedback Shift Register

- A linear feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous state
- The input bit of the shift register is driven by the exclusive-or (XOR) of some bits of the overall shift register value.
- It can used be as a pseudo random number generator



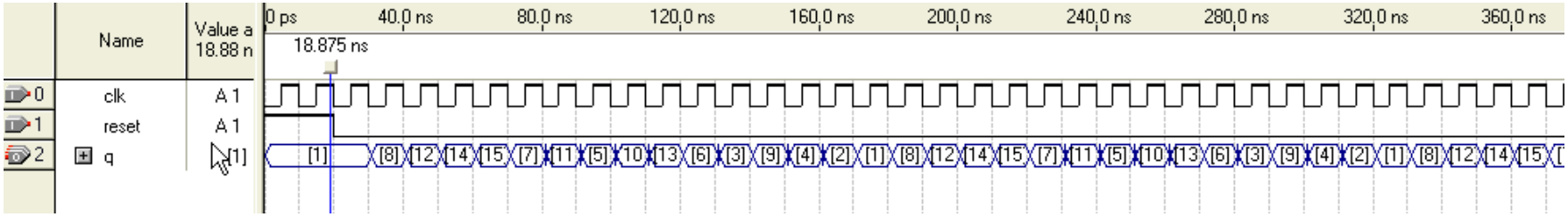
4-bit LFSR

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

entity LFSR4Bit is
  port (reset, clk: in std_logic;
        q: out std_logic_vector(3 downto 0));
end LFSR4Bit ;

architecture behavior of LFSR4Bit is
  signal ps, ns: std_logic_vector(3 downto 0);
  constant seed: std_logic_vector(3 downto 0) := "0001";
  signal fout: std_logic;
begin
  process (clk, reset)
  begin
          if reset = '1' then
              ps <= seed;
          elsif clk'event and clk = '1' then
              ps <= ns;
          end if;
   end process;
  fout <= ps(0) xor ps(3);
  ns <= fout & ps(3 downto 1);
  q <= ps;
end behavior;
```

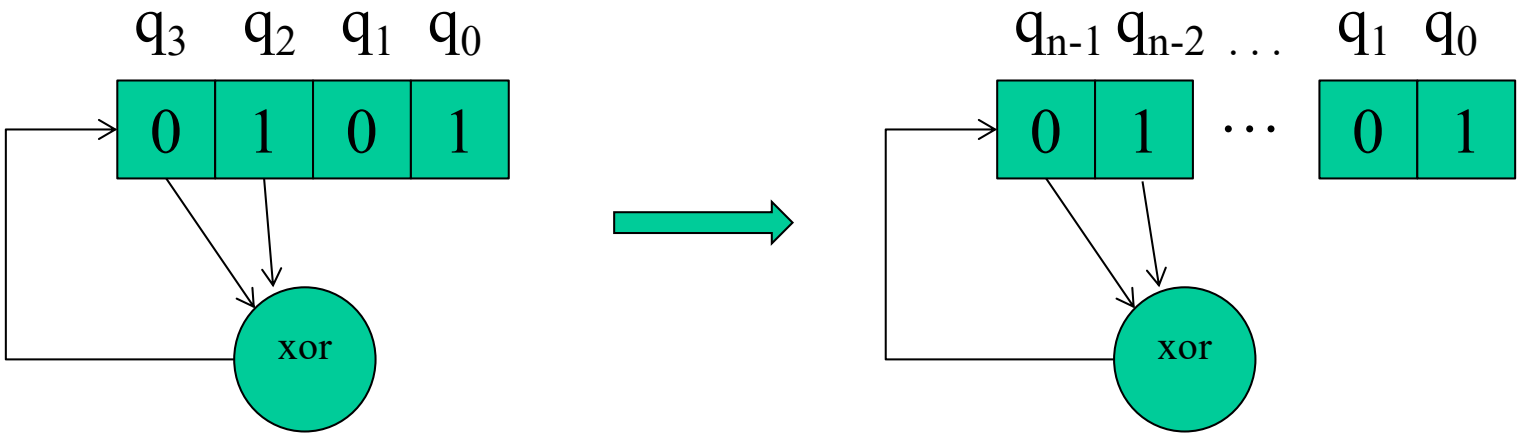


LFSR Outputs

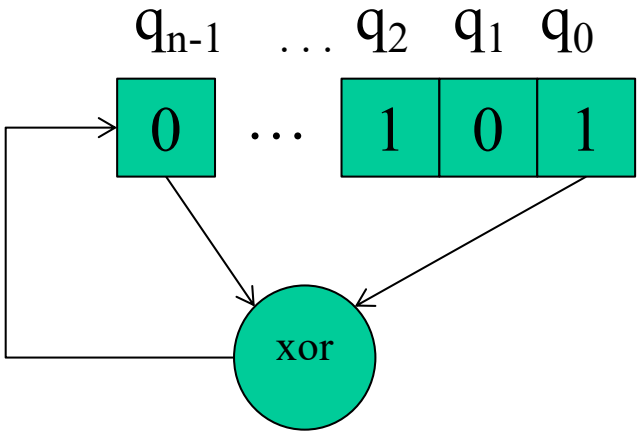
- q = 1, 8, 12, 14, 15, 7, 11, 5, 10, 13, 6, 3, 9, 4, 2



Worksheet

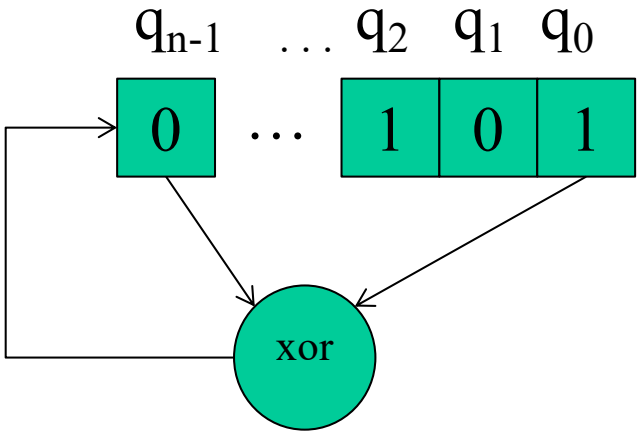


n-bit LFSR




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

entity LFSR_NBit is

  port (reset, clk: in std_logic;
        q: out std_logic_vector(3 downto 0));

end LFSR_NBit ;
```

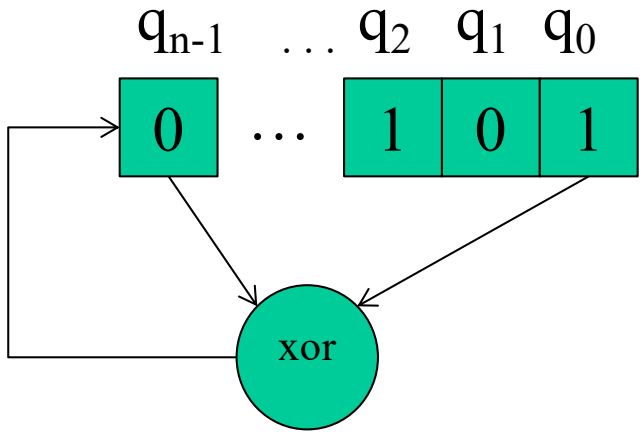
n-bit LFSR




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

entity LFSR_NBit is
  generic (n: integer := 8);
  port (reset, clk: in std_logic;
        q: out std_logic_vector(n-1 downto 0));

end LFSR_NBit ;
```

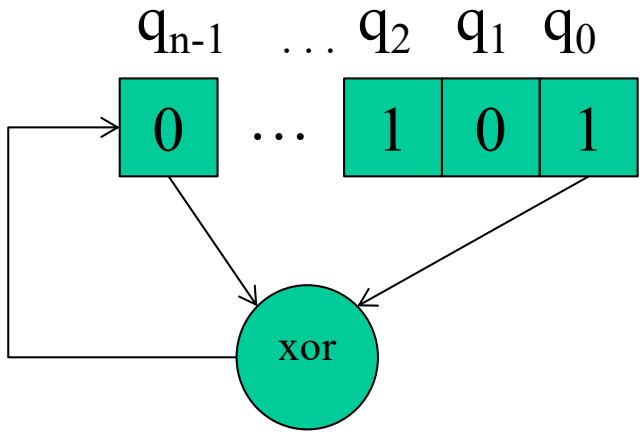
n-bit LFSR

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

entity LFSR_NBit is
  port (reset, clk: in std_logic;
        q: out std_logic_vector(3 downto 0));
end LFSR_NBit ;

architecture behavior of LFSR_NBit is
  signal ps, ns: std_logic_vector(3 downto 0);
  signal fout: std_logic;
begin
  process (clk, reset)
  begin
          if reset = '1' then
              ps <= "0001";
          elsif clk'event and clk = '1' then
              ps <= ns;
          end if;
   end process;
  fout <= ps(0) xor ps(3);
  ns <= fout & ps(3 downto 1);
  q <= ps;
end behavior;
```



n-bit LFSR

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

entity LFSR_NBit is
  generic (n: integer := 8);
  port (reset, clk: in std_logic;
        q: out std_logic_vector(n-1 downto 0));
end LFSR_NBit ;

architecture behavior of LFSR_NBit is
  signal ps, ns: std_logic_vector(n-1 downto 0);
  signal fout: std_logic;
begin
  process (clk, reset)
  begin
          if reset = '1' then
              ps <= (0 => '1', others => '0');
          elsif clk'event and clk = '1' then
              ps <= ns;
          end if;
   end process;
  fout <= ps(0) xor ps(n-1);
  ns <= fout & ps(n-1 downto 1);
  q <= ps;
end behavior;
```



Parity Generator

- A parity bit is a bit added to a string of binary code so that the total number of 1-bits in the string is even or odd.

Binary Code	Odd Parity	Even Parity
0 1 1 1 0 1 1	0 1 1 1 0 1 1 0	0 1 1 1 0 1 1 1
0 1 0 1 1 1 0	0 1 0 1 1 1 0 1	0 1 0 1 1 1 0 0

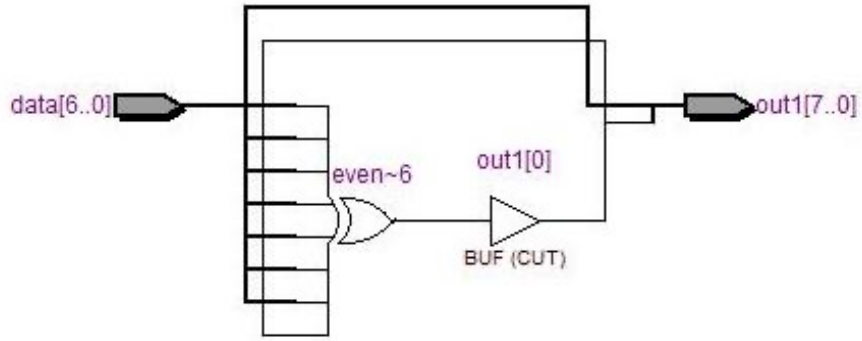
A	B	A xor B
0	0	0
0	1	1
1	0	1
1	1	0

1 xor 0 = 1
1 xor 1 = 0
0 xor 1 = 1
1 xor 1 = 0
0 xor 0 = 0
0 xor 1 = 1
1 xor 1 = 0

1 xor 0 = 1
1 xor 1 = 0
0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
0 xor 1 = 1
1 xor 0 = 1

0 xor 0 = 0
0 xor 1 = 1
1 xor 1 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0
0 xor 1 = 1

0 xor 0 = 0
0 xor 1 = 1
1 xor 0 = 1
1 xor 1 = 0
0 xor 1= 1
1 xor 1 = 0
0 xor 0 = 0




```
-- Even parity bit generator

library ieee;
use ieee.std_logic_1164.all;


entity even_parity is
  port (data: in std_logic_vector(6 downto 0);
        out1: out std_logic_vector(7 downto 0) );
end even_parity;

architecture behavior of even_parity is

begin
  process (data)
    variable even: std_logic := '0';
  begin
    for i in 6 downto 0 loop
      even := even xor data(i);
    end loop;

    out1 <= data & even;
  end process;

end behavior;
```

even : = data(6) xor data (5) … xor data(0);

Function

- Functions are used to compute a value based on the values of the input parameters.

```
function name (parameter list) return type is
 -- declarations
begin
    -- sequential statements
end name;
```

- Function may have local variables
- A function may contain any sequential statements except signal assignment and wait statement.
- A function may be called in either sequential or concurrent statements.

Function Example

```
architecture fun of example is

    function b2sl(x: boolean) return std_logic is
    begin
       if x then
        return '1';
       else
        return '0';
       end if;
    end b2sl;

begin
   a <= b2sl(true);
   b <= b2sl(false);
end fun;
```

4-bit Counter With Display

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;

entity counter4 is
port (reset, clock, up, down: in std_logic;
      z: out std_logic_vector(0 to 6));
end counter4;

architecture behavior of counter4 is
  function seven(x: integer) return std_logic_vector is
  begin
     case x is          -- 0123456
       when  0 => return "0000001";
       when  1 => return "1001111";
       when  2 => return "0010010";
       when  3 => return "0000110";
       when  4 => return "1001100";
       when  5 => return "0100100";
       when  6 => return "0100000";
       when  7 => return "0001111";
       when  8 => return "0000000";
       when  9 => return "0000100";
       when 10 => return "0001000";
       when 11 => return "1100000";
       when 12 => return "0110001";
       when 13 => return "1000010";
       when 14 => return "0110000";
       when 15 => return "0111000";
       when others => return "1111111";
     end case;
  end seven;
```

```
begin
count_process: process
     variable count: integer range 0 to 15;
  begin
    wait until(rising_edge(clock));
    if reset = '1' then
              count := 0;
    elsif up = '1' and down = '0' then
      count := count + 1;
    elsif up = '0' and down = '1' then
      count := count - 1;
    end if;

    z <= seven(count);
  end process count_process;

end behavior;
```

Procedure

- Procedure are subprograms that can modify one or more of the input parameters.

```
procedure name (parameter_list) is
 -- declarations
begin
 -- sequential statements
end name;
```

- Formal parameters are separated by semicolons.
- Each parameter must be declared with type and may have mode
- Procedure may be called sequentially or concurrently.

Procedure Example

```
architecture proc of example is
    signal zout: std_logic;
    shared variable xvar: boolean;
    procedure b2sl_proc(variable x: boolean;
                        signal y: out std_logic) is
    begin
       if x then
        y <= '1';
       else
        y <= '0';
       end if;
    end b2sl_proc;

begin
   b2sl_proc(xvar, zout);
end proc;
```

Shared Variables

- Variables are declared inside the process statement, therefore it is local to the process and cannot be access by other process.
- Shared variables allow inter-process communication.

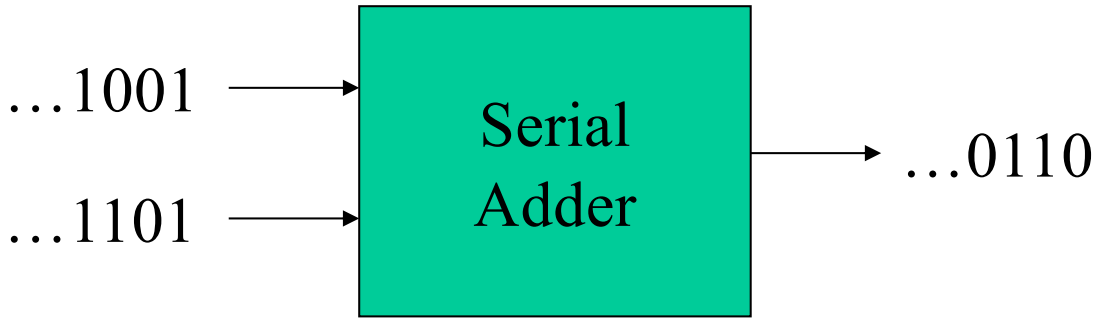
```
shared variable x: integer;
```

- Shared variables are declared in the architecture body.
- Only one process may access the variable in a single simulation cycle.

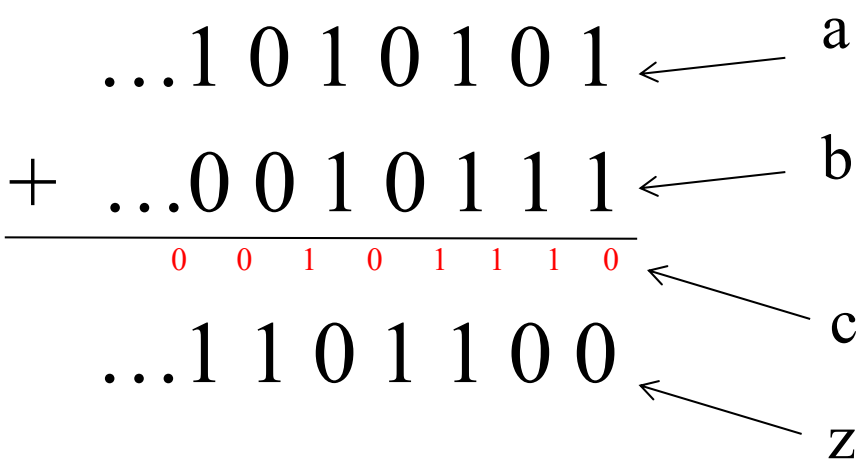
```
entity example is
  . . .
end example;

architect behavior of example is
   shared variable imshared: integer;  -- declare shared

begin
   A: process
     variable x: integer;  -- local
   begin
      . . .
      x := imshared;    -- read shared variable
   end process;

   B: process
     variable y: integer;  -- local
   begin
      . . .
      imshared := y     -- write shared variable (only 1 process should write)
   end process;

   C: process
     variable z: integer;  -- local
   begin
      . . .
      z := imshared;    -- read shared variable
   end process;

end behavior;
```

Serial Adder

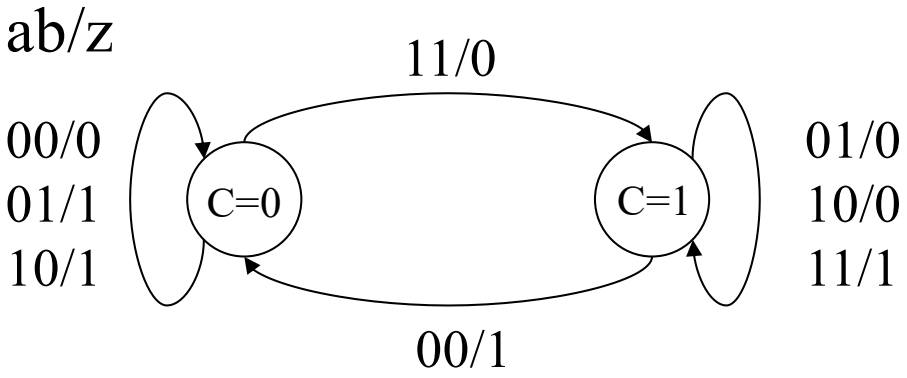
- Add two serial stream of binary numbers
- Implemented as a two states FSM, using the value of the carry as the state



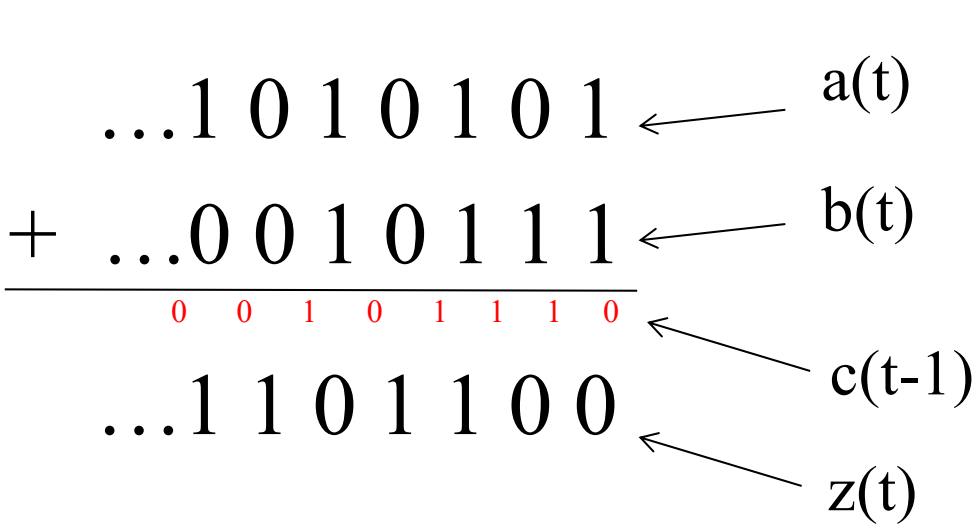
Serial Addition

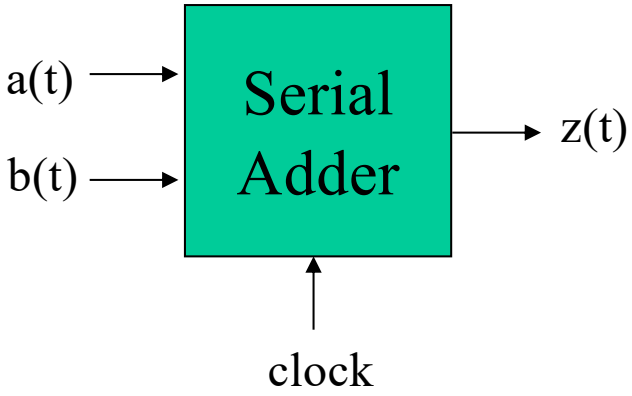


State machine approach



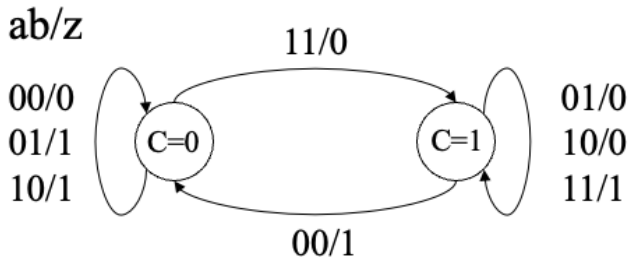
Serial Adder



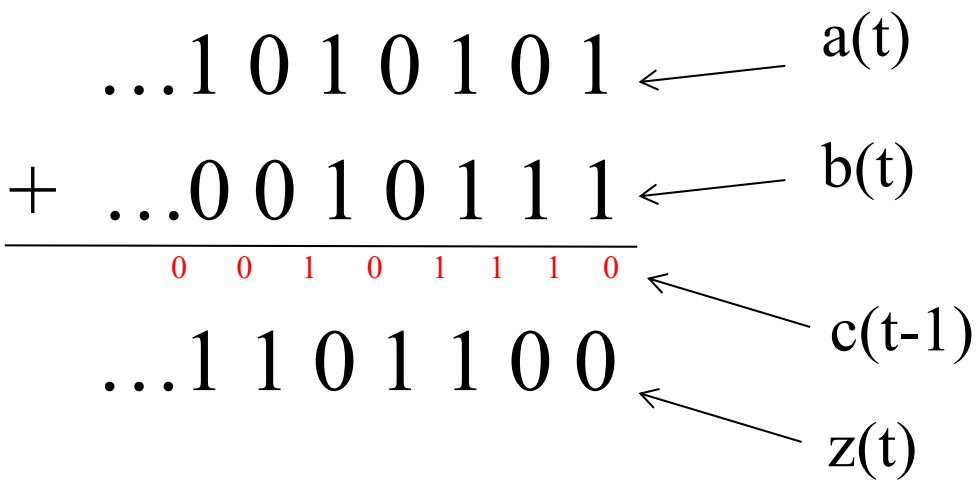


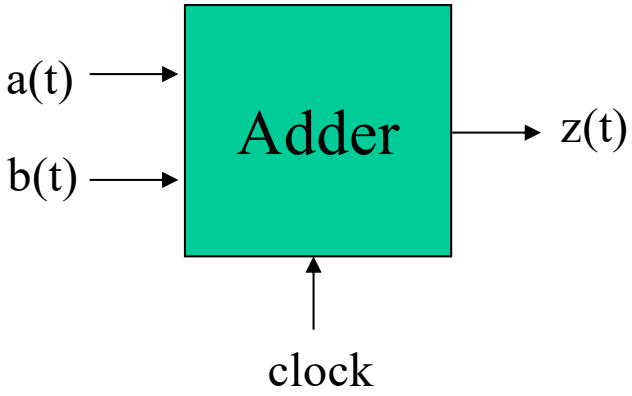

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity serial_adder is
  port (a, b, clock: in std_logic;
        z: out std_logic);
end serial_adder;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity serial_adder is
  port (a, b, clk: in std_logic;
        z: out std_logic);
end serial_adder;

architecture fsm of serial_adder is
  signal c: std_logic := '0';
  signal nc: std_logic := '0';
  signal tz: std_logic;
begin
  ns_process: process (a, b)
  begin
    case c is
      when '0' =>
        if a = '1' and b = '1' then
          nc <= '1'; tz <= '0';
        elsif a = '0' and b = '0' then
          nc <= '0'; tz <= '0';
        else
          nc <= '0'; tz <= '1';
        end if;
      when others =>
        if a = '0' and b = '0' then
          nc <= '0'; tz <= '1';
        elsif a = '1' and b = '1' then
          nc <= '1'; tz <= '1';
        else
          nc <= '1'; tz <= '0';
        end if;
    end case;
  end process ns_process;

  clk_process: process
  begin
    wait until(rising_edge(clk));
    c <= nc;
    z <= tz;
  end process clk_process;
end fsm;
```



Serial Adder



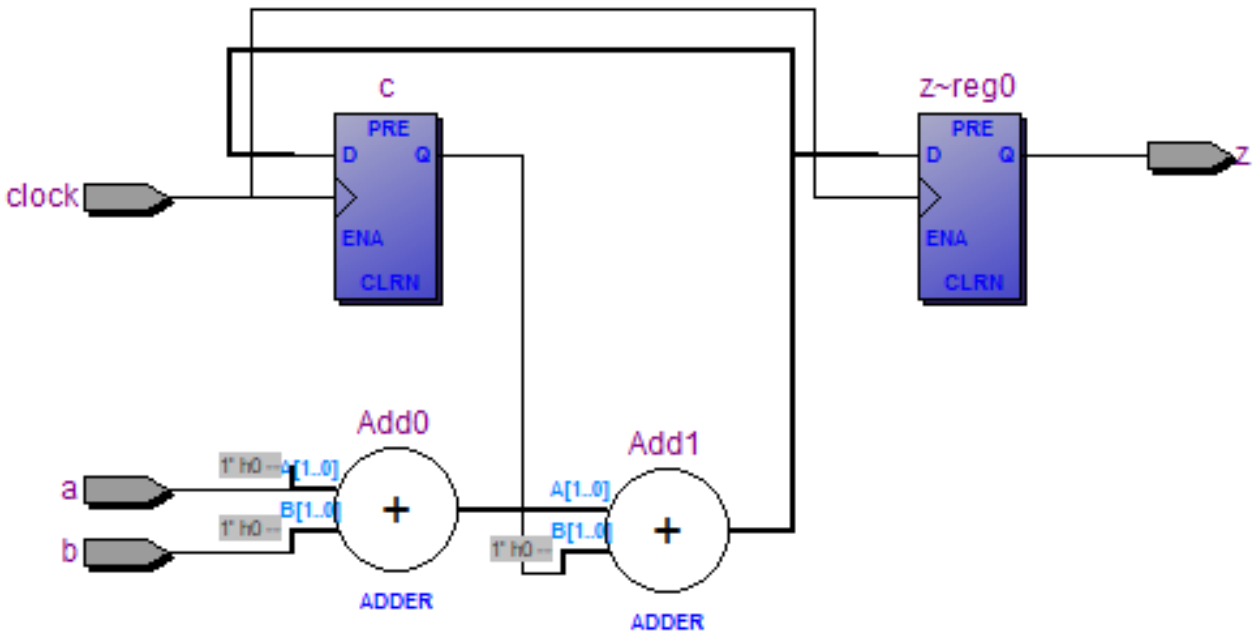



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
entity serial_adder is
  port (a, b, clock: in std_logic;
        z: out std_logic);
end serial_adder;
```

addition operator

$$c(t), z(t) = a(t) + b(t) + c(t\text{-}1)$$

```
architecture behavior of serial_adder is
  signal c: std_logic := '0';
begin
  process
    variable x, y, carry, s: unsigned(1 downto 0);
  begin
    wait until(rising_edge(clock));
    x := '0' & a;
    y := '0' & b;
    carry := '0' & c;
    s := x + y + carry;
    z <= s(0);
    c <= s(1);
  end process;
end behavior;
```



Structural Modeling

- Behavioral descriptions of a circuit may be specified using concurrent signal assignment statements
- For more complex circuit, we can use one or more processes and sequential statements to model the circuit
- The circuit may also be modeled in an alternate way using structural model
- Structural model using VHDL statements to describe how to form the circuit by connecting the components of the circuit

Components

- You can think of a VHDL model as a standalone circuit or it can be used as part of a more complex circuit
- You can have a library of circuit components, such as register, MUX, counter, state machines, etc.
- Connect the components to form the model of the circuit
- It is like wiring the circuit on a breadboard

Component Declaration

- Components must be declared in the architecture body where they are used unless they are part of a library
- Similar to the entity declaration, use the keyword **component** instead of **entity**
- Declare the interface to the component
- Like declaring subroutine/function and parameters in a program

Structural Model Template

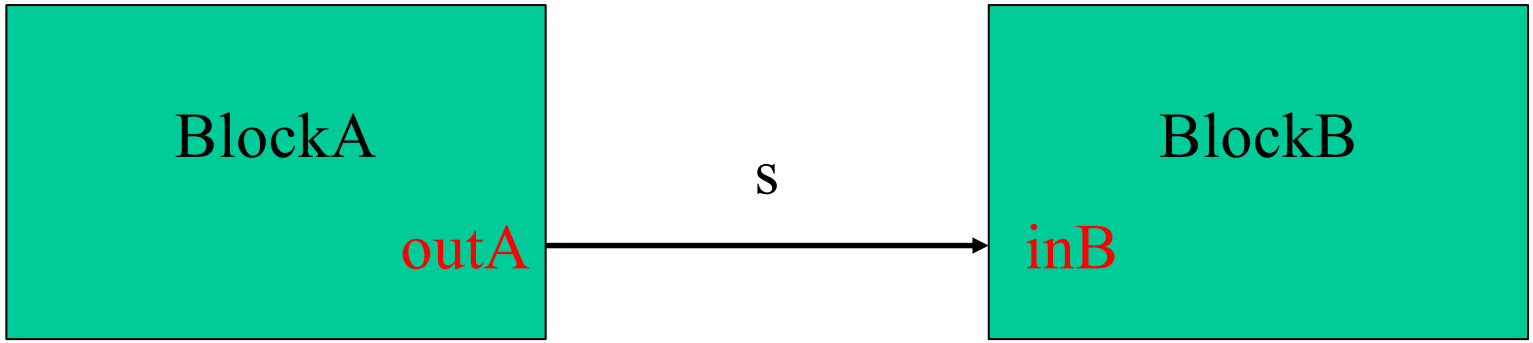
```
entity entity_name is
  port (input signals: in type;
        output signals: out type);
end entity_name;

architecture arch_name of entity_name is

component component1_name
port (input signals: in type;
      output signals: out type);
end component;

component component2_name
port (input signals: in type;
      output signals: out type);
end component;

signal internal signals : type := initialization;

begin
  Label1: component1_name port map(port=>signal, ...);
  Label2: component2_name port map(port=>signal, ...);
end arch_name;
```

Connecting Components

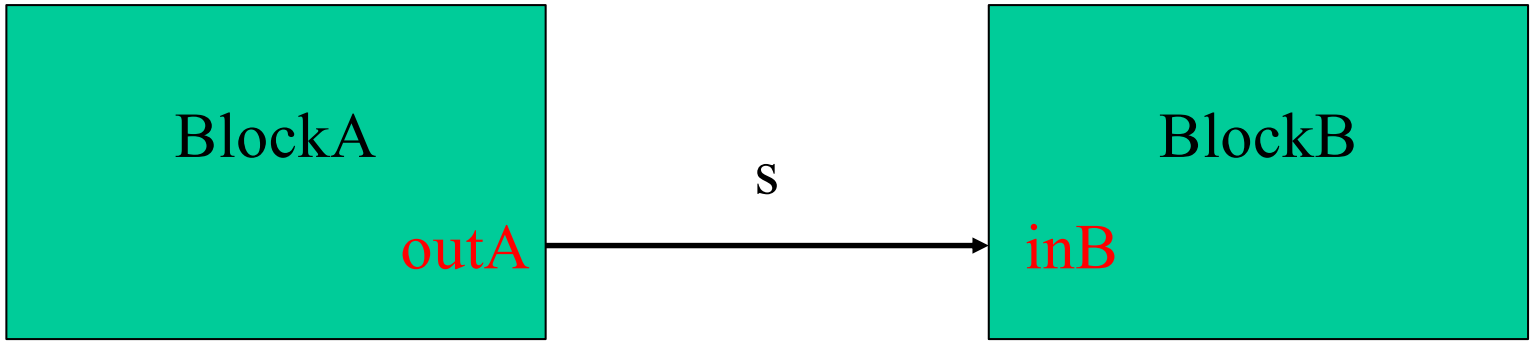

```
entity blockA is
  port (outA: out bit);
end blockA;
```

```
entity blockB is
  port (inB: in bit);
end blockB;
```

Connecting Components



Connecting Components




```
signal s: bit;

Label1: BlockA port map(outA => s);
Label2: BlockB port map(inB => s);
```

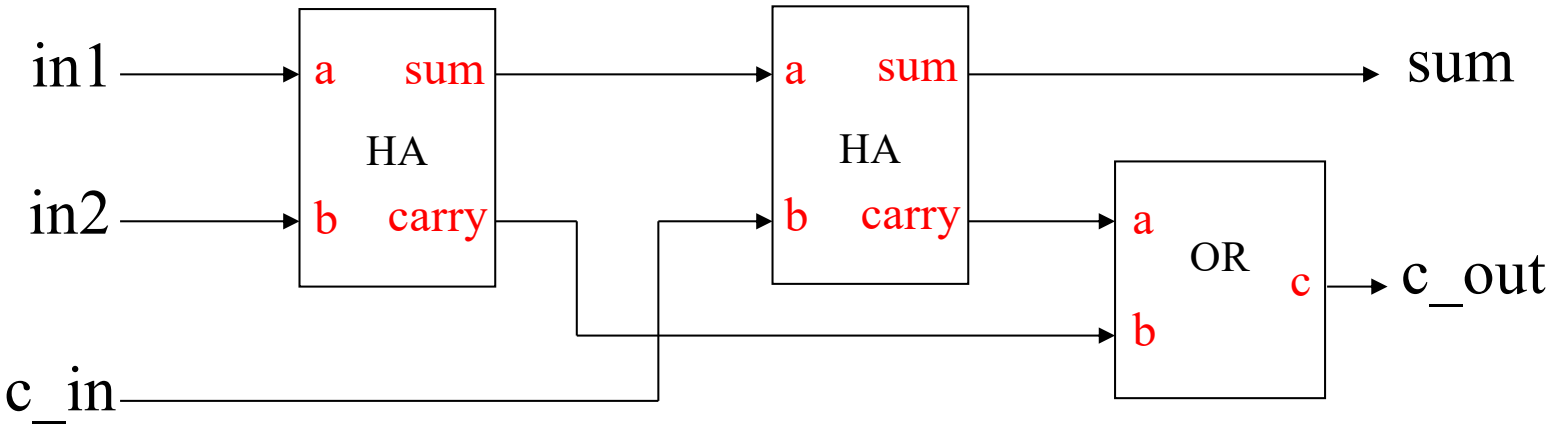
Using the Components

- Use **port map**( ) construct to connect the input and output ports of the component to signals
- Label: name **port map**(port => signal, . . .)
- A label must be present for each port map statement, why?
- Each port map statement creates an instance of the component

Component Files

- Each component is itself a complete model, that is you can compile and program it into hardware
- Components can be used in other components.
- The VHDL file of the component must be added into the Quartus II project.

Full-Adder Circuit



Half-adder

$$\text{sum} = a \oplus b$$

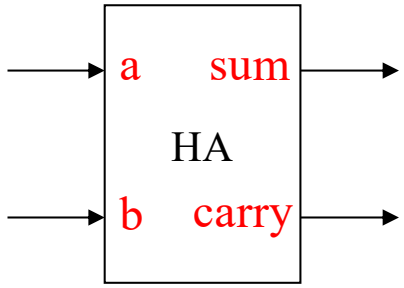
$$\text{carry} = a \cdot b$$

Full-adder

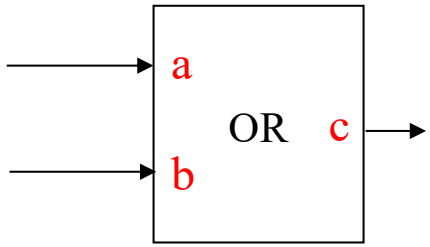
$$\text{sum} = (\text{in1} \oplus \text{in2}) \oplus \text{c\_in}$$

$$\text{c\_out} = (\text{in1} \oplus \text{in2}) \cdot \text{c\_in} + (\text{in1} \cdot \text{in2})$$

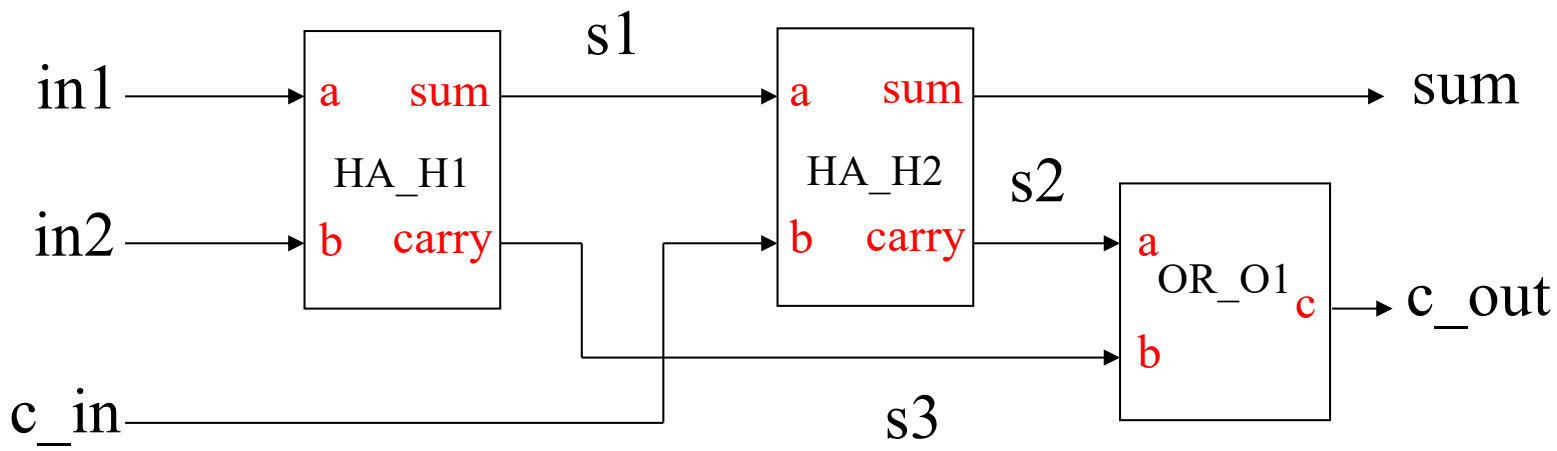
```
-- file: half_adder.vhd
library ieee;
use ieee.std_logic_1164.all;
entity half_adder is
  port (a, b: in std_logic;
        sum, carry: out std_logic);
end half_adder;

architecture dataflow of half_adder is
begin
    sum <= a xor b;
    carry <= a and b;
end dataflow;
```




```
-- file: or_2.vhd
library ieee;
use ieee.std_logic_1164.all;
entity or_2 is
  port (a, b: in std_logic;
        c: out std_logic);
end or_2;

architecture dataflow of or_2 is
begin
    c <= a or b;
end dataflow;
```



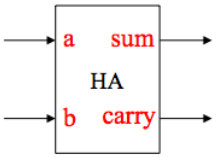
Full-Adder Circuit

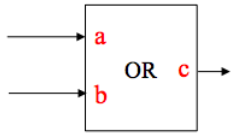


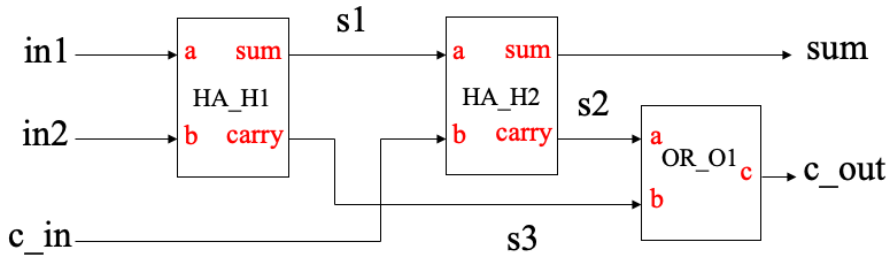
identifiers in red are port names of the component
identifiers in black are either inputs/outputs or internal signals

```vhdl
-- file: full_adder.vhd
library ieee;
use ieee.std_logic_1164.all;
entity full_adder is
  port (in1, in2, c_in: in std_logic;
        sum, c_out: out std_logic);
end full_adder;

architecture structural of full_adder is

component half_adder
port (a, b: in std_logic;
      sum, carry: out std_logic);
end component;

component or_2
port (a, b: in std_logic;
      c: out std_logic);
end component;

signal s1, s2, s3: std_logic;
begin
   HA_H1: half_adder port map(a=>in1, b=>in2, sum=>s1, carry=>s3);
   HA_H2: half_adder port map(a=>s1, b=>c_in, sum=>sum, carry=>s2);
   OR_O1: or_2 port map(a=>s2, b=>s3, c=>c_out);
end structural;
```

```vhdl
-- file: half_adder.vhd
library ieee;
use ieee.std_logic_1164.all;
entity half_adder is
  port (a, b: in std_logic;
        sum, carry: out std_logic);
end half_adder;

architecture dataflow of half_adder is
begin
    sum <= a xor b;
    carry <= a and b;
end dataflow;

-- file: or_2.vhd
library ieee;
use ieee.std_logic_1164.all;
entity or_2 is
  port (a, b: in std_logic;
        c: out std_logic);
end or_2;

architecture dataflow of or_2 is
begin
    c <= a or b;
end dataflow;
```

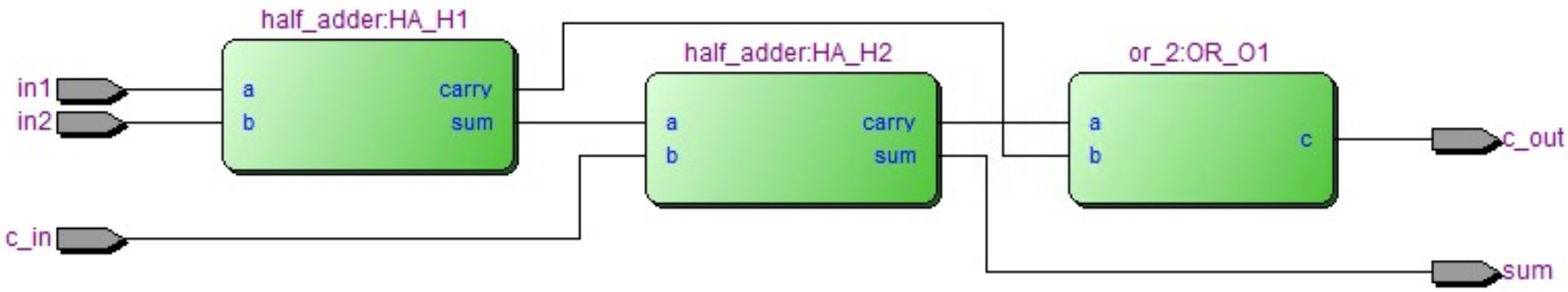






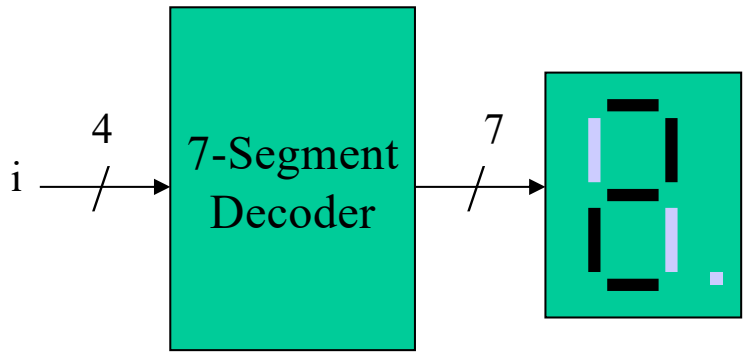
Full-Adder Structural Model

- The two half_adder port map statements created two instances of the half-adder, HA_H1 and HA_H2
- The OR_O1 port map statement created one instance of the two-input or



BCD to Seven Segment Decoder

- Convert binary-code-decimal value to seven segment display

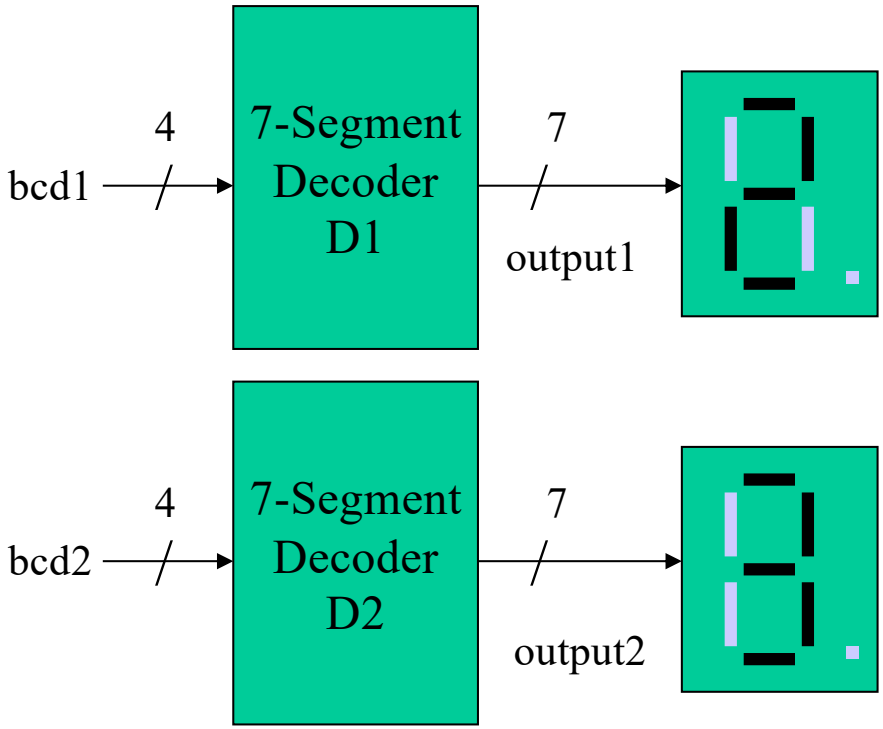


BCD to 7-Segment Model

```
-- Binary Code Decimal to 7-segment display
-- file: BCD_7Seq.vhd
library ieee;
use ieee.std_logic_1164.all;

entity BCD_7Seg is
  port (i : in std_logic_vector(3 downto 0);
        segments: out std_logic_vector(0 to 6));
end BCD_7Seg;

architecture behavior of BCD_7Seg is
begin           --segment
  with i select --0123456
    segments <=  "0000001" when "0000",
                 "1001111" when "0001",
                 "0010010" when "0010",
                 "0000110" when "0011",
                 "1001100" when "0100",
                 "0100100" when "0101",
                 "0100000" when "0110",
                 "0001111" when "0111",
                 "0000000" when "1000",
                 "0001100" when "1001",
                 "1111111" when others;
end behavior;
```

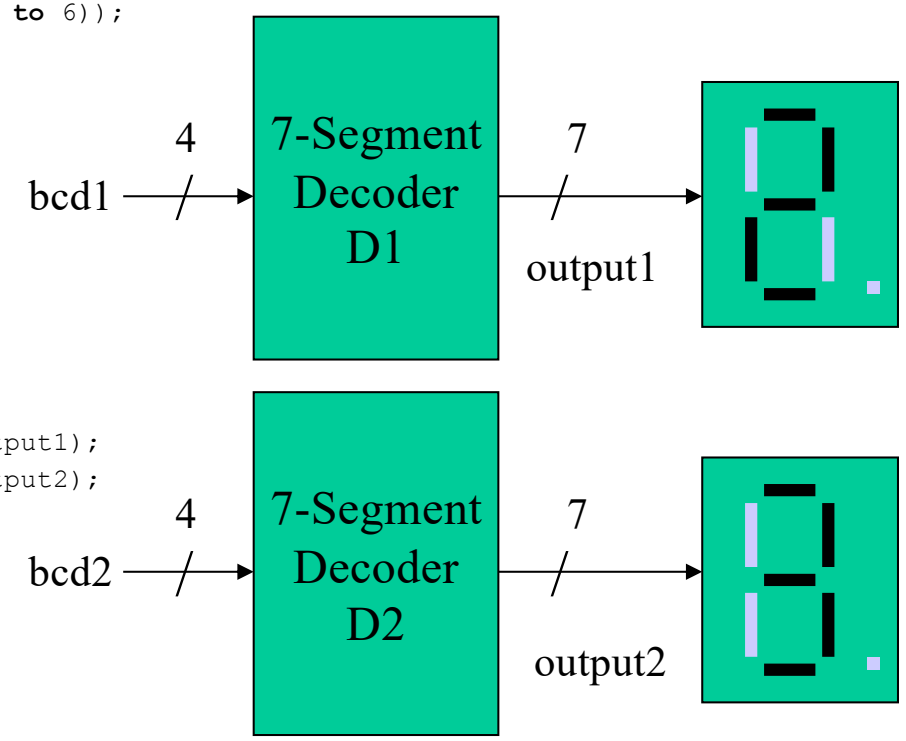
Two-Digit Display

- Display two digits
- Use two BCD to seven segment decoders



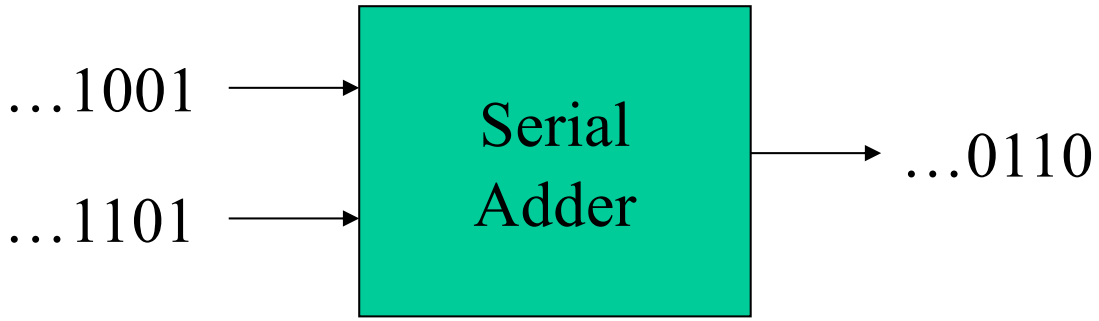
2-Digit Display

```
-- file: two_digits.vhd
library ieee;
use ieee.std_logic_1164.all;
entity two_digits is
  port (bcd1, bcd2: in std_logic_vector(3 downto 0);
        output1, output2: out std_logic_vector(0 to 6));
end two_digits;

architecture structural of two_digits is

component BCD_7Seg
  port (i : in std_logic_vector(3 downto 0);
        segments: out std_logic_vector(0 to 6));
end component;

begin
     D1: BCD_7Seg port map(i=>bcd1, segments=>output1);
     D2: BCD_7Seg port map(i=>bcd2, segments=>output2);
end structural;
```

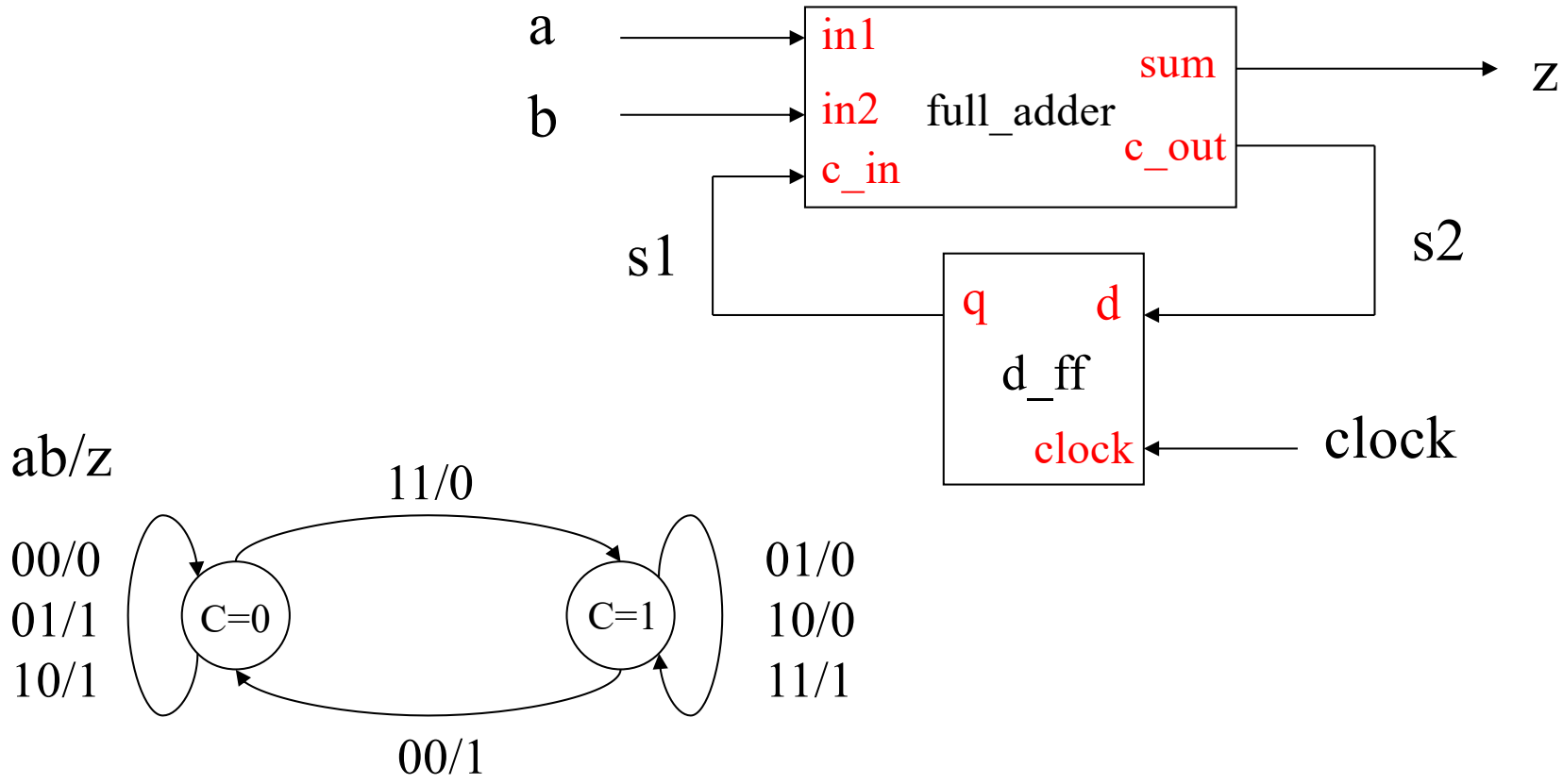


Serial Adder

- Add two serial stream of binary numbers
- Implemented as a two states FSM, using the value of the carry as the state

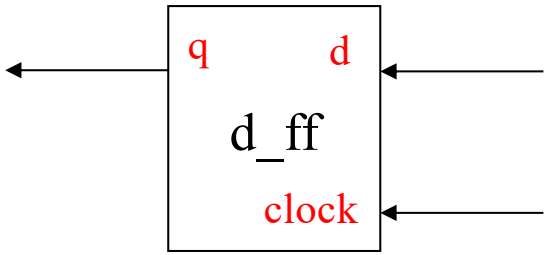


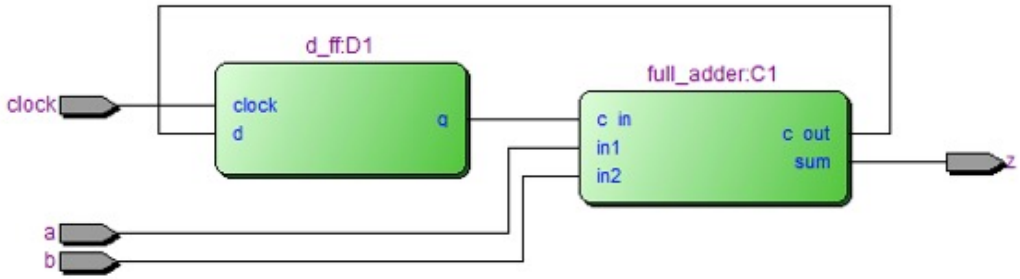
Serial Adder



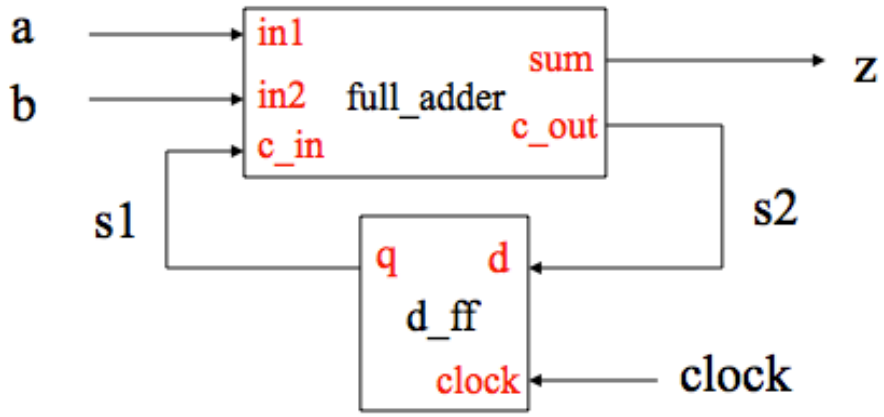
D_FF

```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```



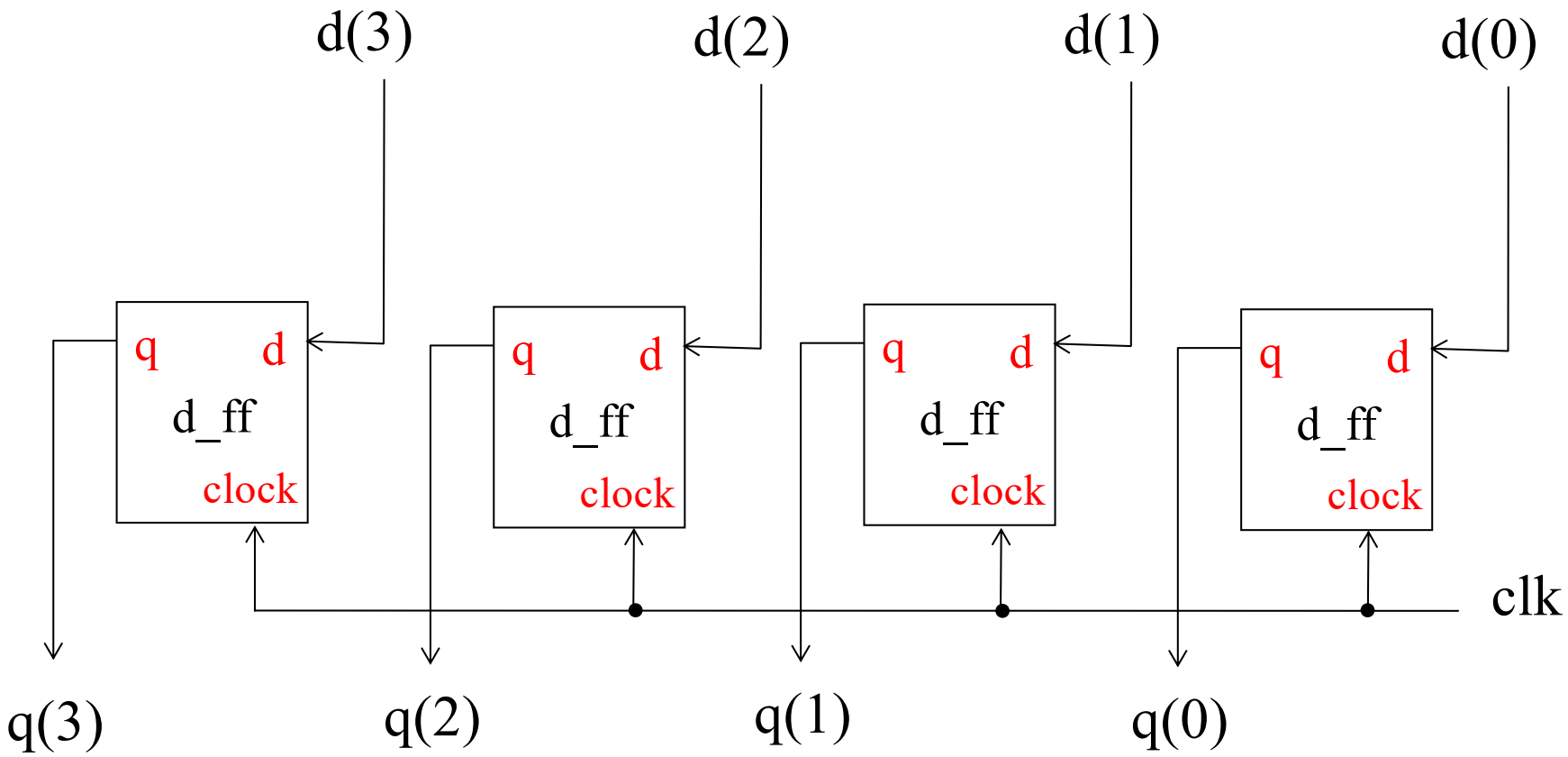



```
-- file: serial_adder.vhd
library ieee;
use ieee.std_logic_1164.all;
entity serial_adder is
  port (a, b, clock: in std_logic;
        z: out std_logic);
end serial_adder;

architecture structural of serial_adder is

component full_adder
  port (in1, in2, c_in: in std_logic;
        sum, c_out: out std_logic);
end component;

component d_ff
  port (clock, d: in std_logic;
        q: out std_logic);
end component;
```



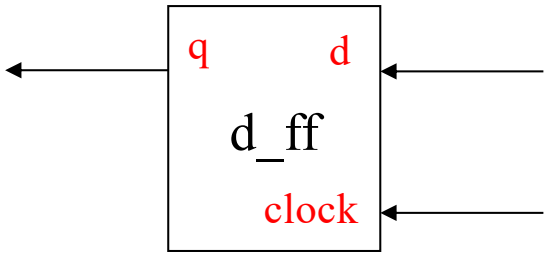

```
signal s1, s2: std_logic;
begin
  C1: full_adder port map(in1=>a, in2=>b, c_in=>s1, sum=>z, c_out=>s2);
  D1: d_ff port map(clock=>clock, d=>s2, q=>s1);
end structural;
```

4-bit Register using D FF



D_FF

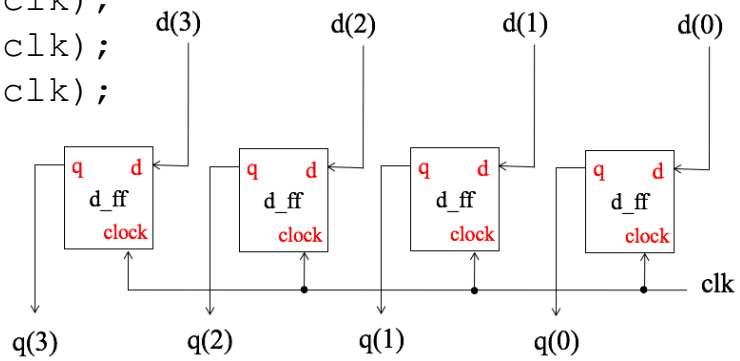
```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```



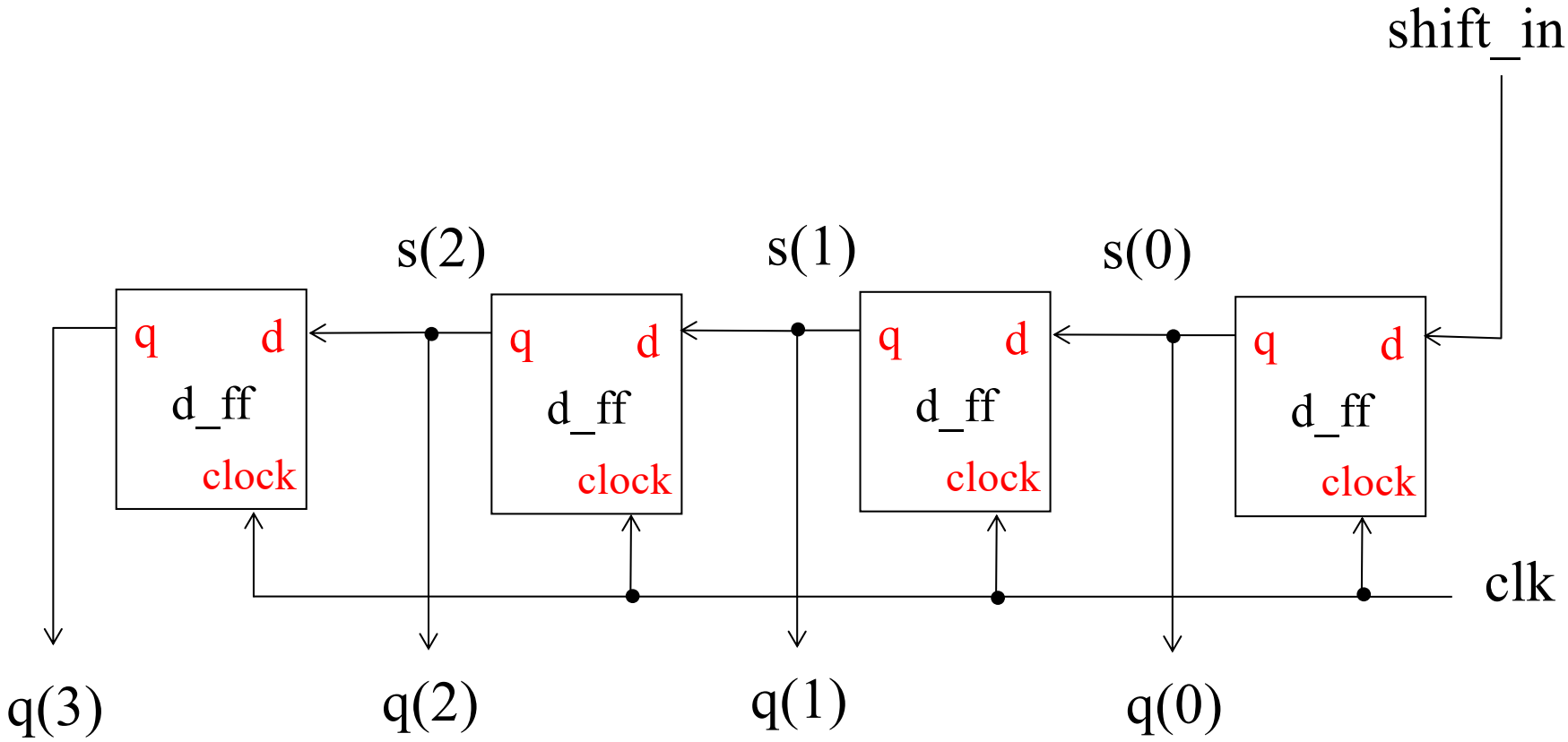
4-bit Register - Component

```
library ieee;
use ieee.std_logic_1164.all;
entity d_register is
  port (d: in std_logic_vector(3 downto 0);
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end d_register;

architecture structural of d_register is

component d_ff
  port(d, clock: in std_logic;
       q: out std_logic);
end component;

begin
  bit0: d_ff port map (d=>d(0), q=>q(0), clock=>clk);
  bit1: d_ff port map (d=>d(1), q=>q(1), clock=>clk);
  bit2: d_ff port map (d=>d(2), q=>q(2), clock=>clk);
  bit3: d_ff port map (d=>d(3), q=>q(3), clock=>clk);
end structural;
```

```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```



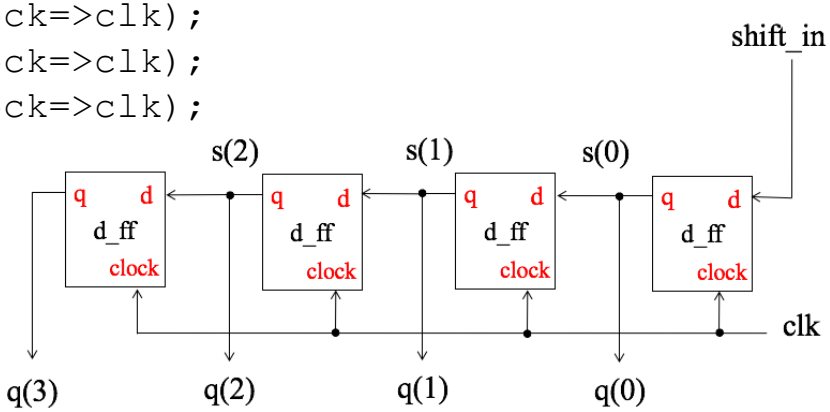
4-bit Shift Register



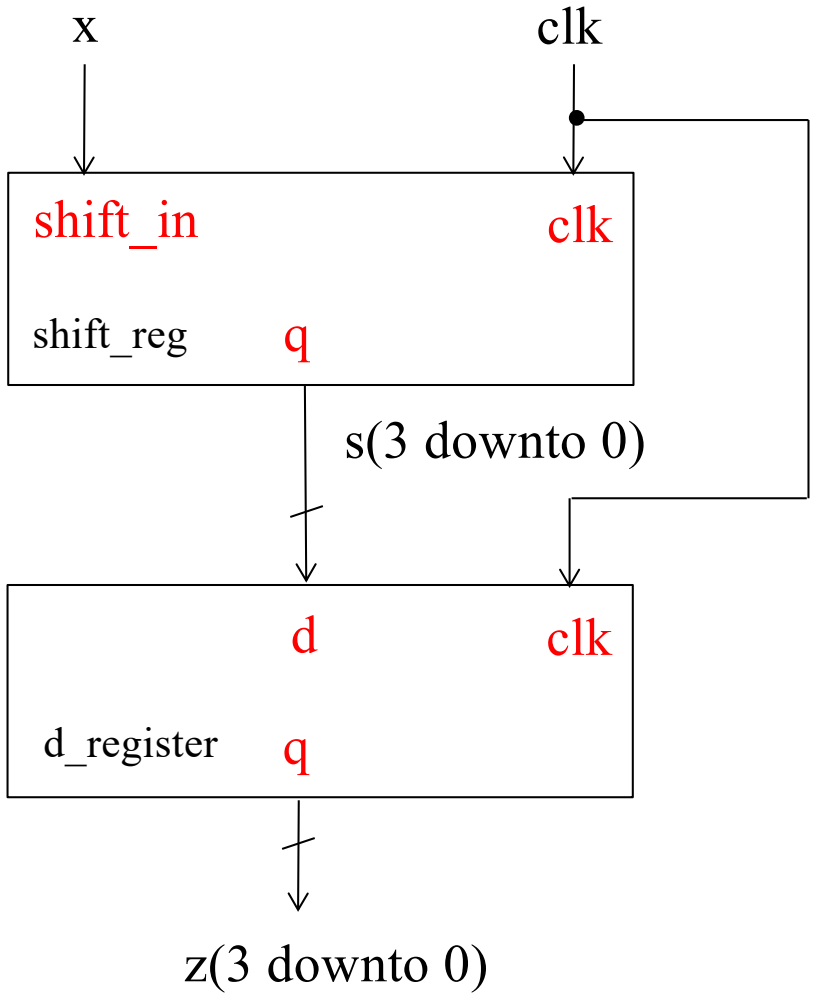
4-bit Shift Register - Component

```
library ieee;
use ieee.std_logic_1164.all;
entity shift_reg is
  port (shift_in: in std_logic;
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end shift_reg;

architecture structural of shift_reg is
component d_ff
  port(d, clock: in std_logic;
       q: out std_logic);
end component;
signal s: std_logic_vector(2 downto 0);
begin
  bit0: d_ff port map (d=>shift_in, q=>s(0), clock=>clk);
  bit1: d_ff port map (d=>s(0), q=>s(1), clock=>clk);
  bit2: d_ff port map (d=>s(1), q=>s(2), clock=>clk);
  bit3: d_ff port map (d=>s(2), q=>q(3), clock=>clk);
  q(2) <= s(2); q(1) <= s(1); q(0) <= s(0);
end structural;
```

```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```

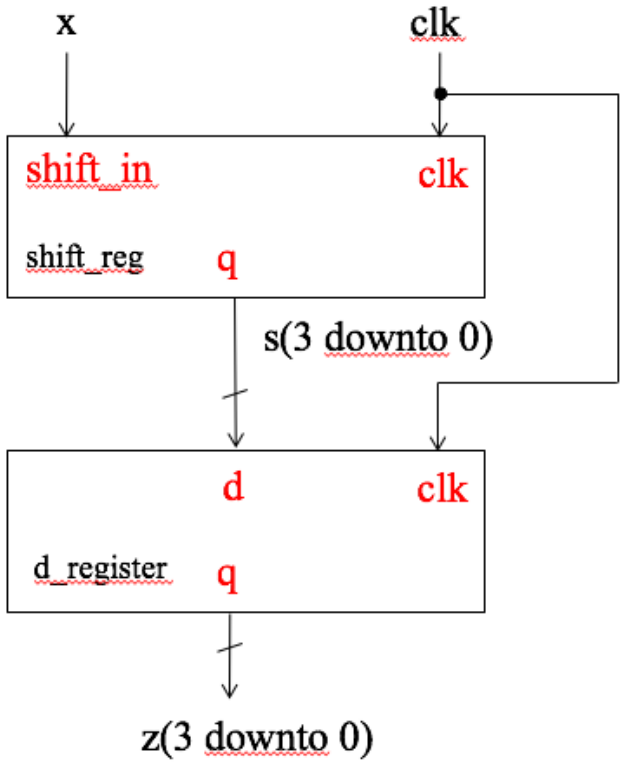


4-bit Shift Register + 4-bit Register



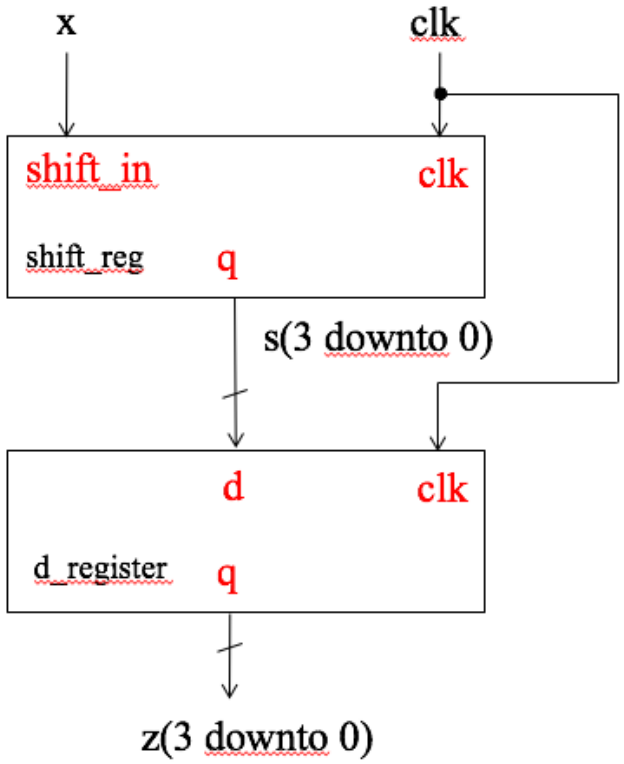
4-bit Shift Register + 4-bit Register

```
library ieee;
use ieee.std_logic_1164.all;
entity circuit is
  port (x, clk: in std_logic;
        z: out std_logic_vector(3 downto 0));
end circuit;

architecture structural of circuit is
component shift_reg
  port (shift_in: in std_logic;
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end component;

component d_register
  port (d: in std_logic_vector(3 downto 0);
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end component;

signal s: std_logic_vector(3 downto 0);
begin


end structural;
```



4-bit Shift Register + 4-bit Register

```
library ieee;
use ieee.std_logic_1164.all;
entity circuit is
  port (x, clk: in std_logic;
        z: out std_logic_vector(3 downto 0));
end circuit;

architecture structural of circuit is
component shift_reg
  port (shift_in: in std_logic;
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end component;

component dregister
  port (d: in std_logic_vector(3 downto 0);
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end component;

signal s: std_logic_vector(3 downto 0);
begin
  block1: shift_reg port map (shift_in=>x, q=>s, clk=>clk);
  block2: d_register port map (d=>s, q=>z, clk=>clk);
end structural;
```



Generate

- Allow well patterned structures to be created easily.
- Any VHDL concurrent statement can be included in a GENERATE statement, including another GENERATE statement.
- Reduce the number of statements

For … Generate

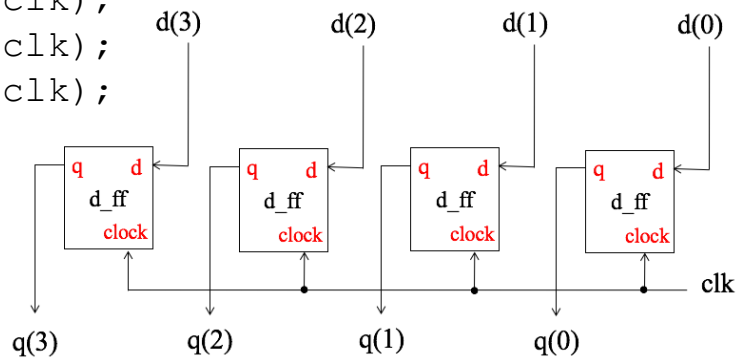
- Instantiates identical components

```
label: for index in range generate
        statements
end generate;
```

4-bit Register - Component

```
library ieee;
use ieee.std_logic_1164.all;
entity d_register is
  port (d: in std_logic_vector(3 downto 0);
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end d_register;

architecture structural of d_register is

component d_ff
  port(d, clock: in std_logic;
       q: out std_logic);
end component;

begin
  bit0: d_ff port map (d=>d(0), q=>q(0), clock=>clk);
  bit1: d_ff port map (d=>d(1), q=>q(1), clock=>clk);
  bit2: d_ff port map (d=>d(2), q=>q(2), clock=>clk);
  bit3: d_ff port map (d=>d(3), q=>q(3), clock=>clk);
end structural;
```

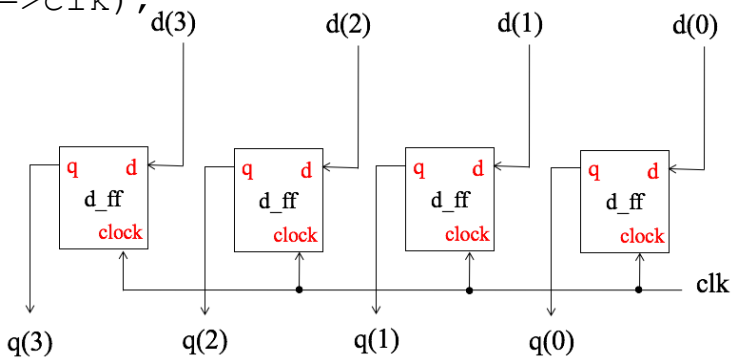
```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```



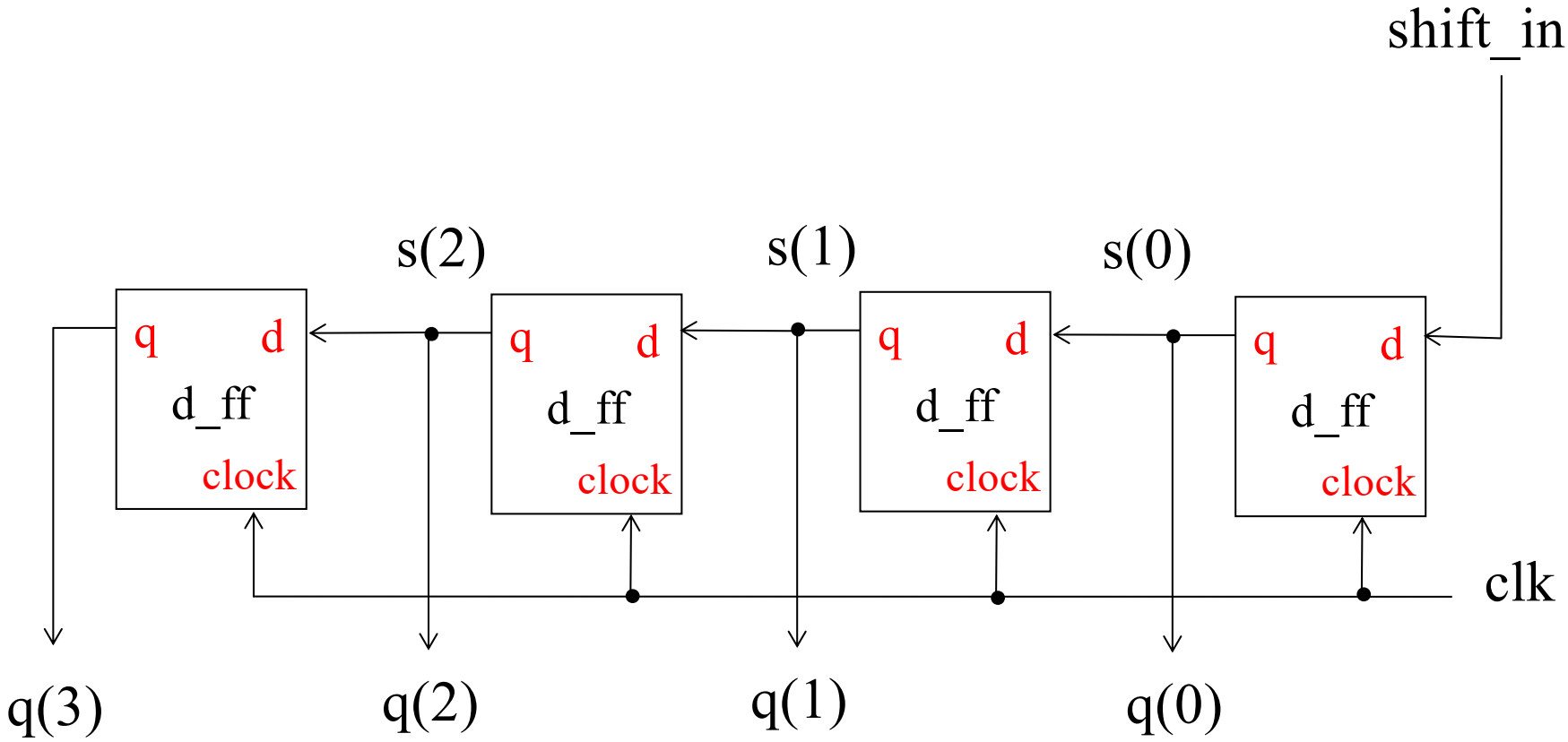
4-bit Register - Component

```
library ieee;
use ieee.std_logic_1164.all;
entity d_register is
  port (d: in std_logic_vector(3 downto 0);
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end d_register;

architecture structural of d_register is

component d_ff
  port(d, clock: in std_logic;
       q: out std_logic);
end component;

begin
  for i in 0 to 3 generate
    biti: d_ff port map (d=>d(i), q=>q(i), clock=>clk);
  end generate;
end structural;
```

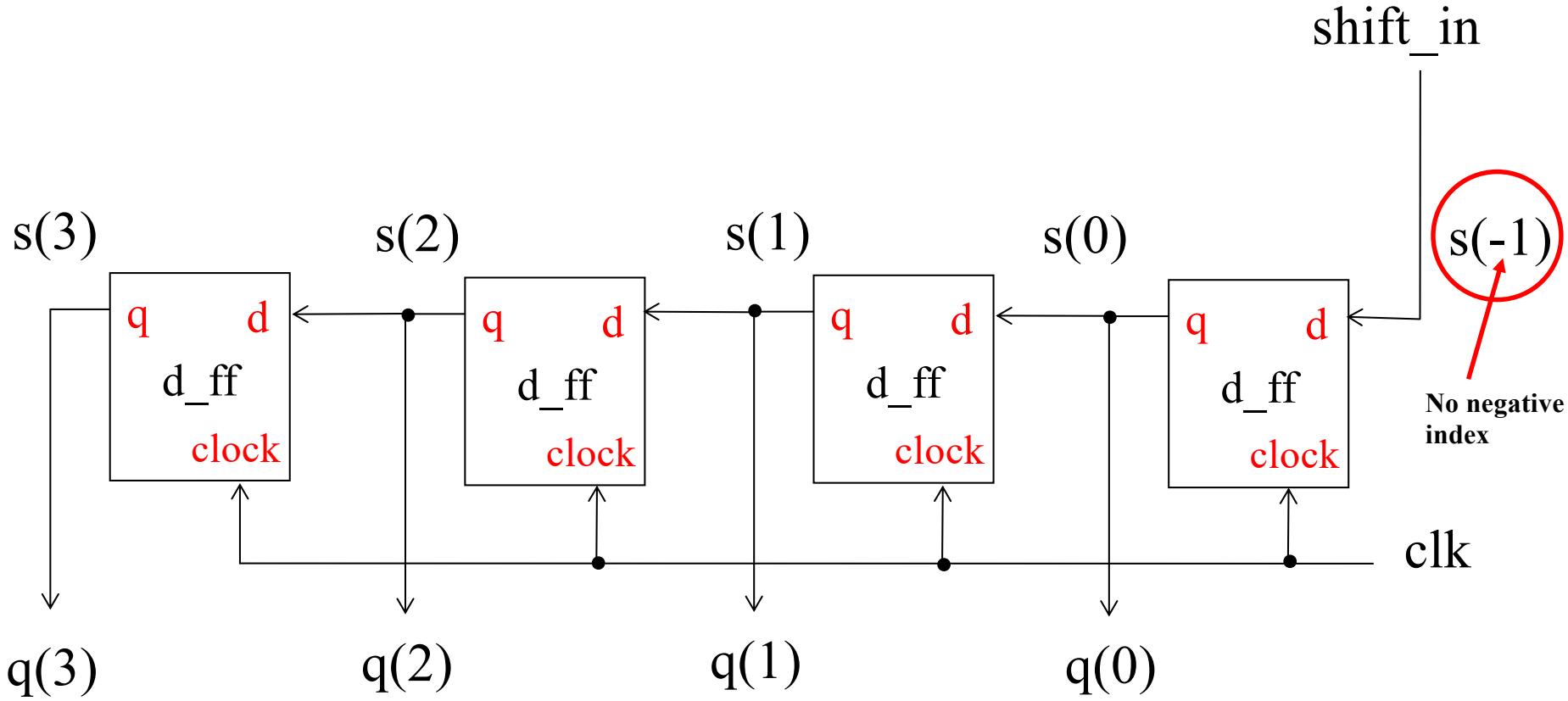
```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```



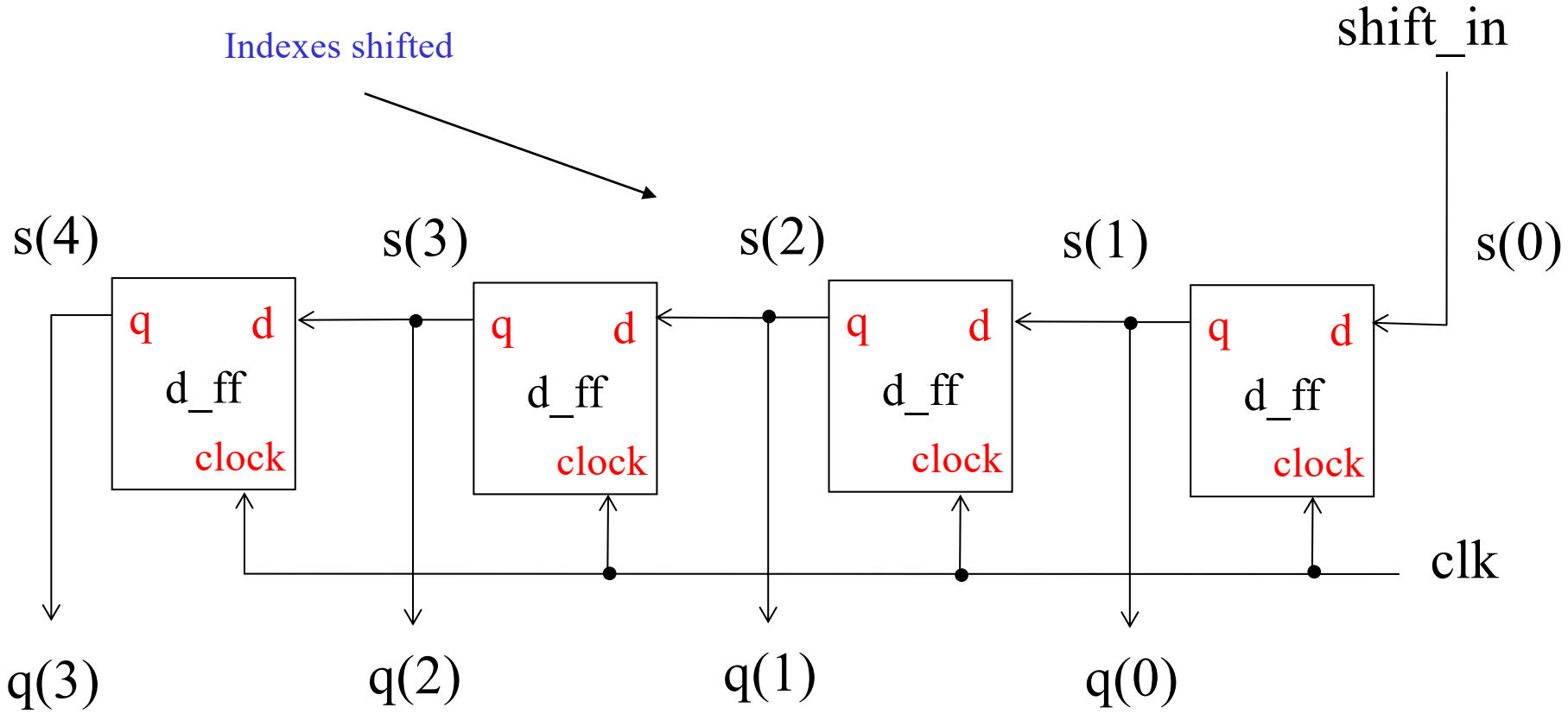
4-bit Shift Register



4-bit Shift Register



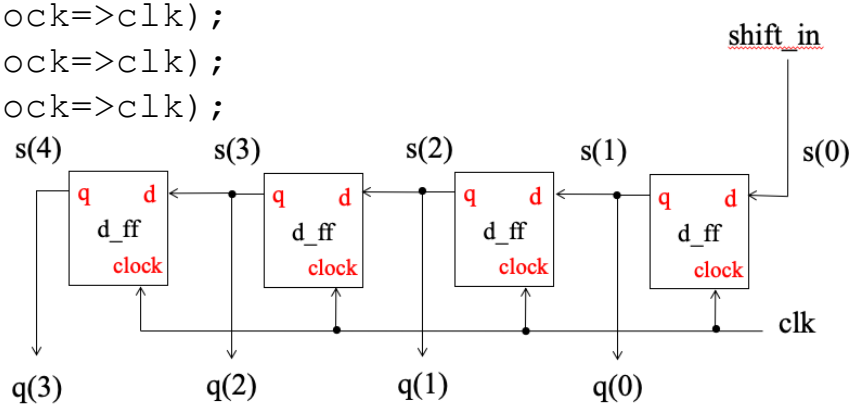
4-bit Shift Register



4-bit Shift Register - Component

```
library ieee;
use ieee.std_logic_1164.all;
entity shift_reg is
  port (shift_in: in std_logic;
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end shift_reg;

architecture structural of shift_reg is
component d_ff
  port(d, clock: in std_logic;
       q: out std_logic);
end component;
signal s: std_logic_vector(4 downto 0);
begin
  bit0: d_ff port map (d=>s(0), q=>s(1), clock=>clk);
  bit1: d_ff port map (d=>s(1), q=>s(2), clock=>clk);
  bit2: d_ff port map (d=>s(2), q=>s(3), clock=>clk);
  bit3: d_ff port map (d=>s(3), q=>s(4), clock=>clk);
  q(3) <= s(4); q(2) <= s(3);
  q(1) <= s(2); q(0) <= s(1);
  s(0) <= shift_in;
end structural;
```

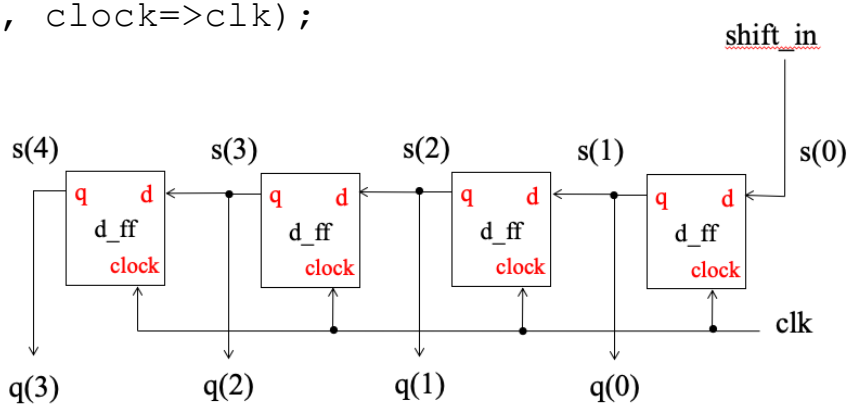
```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```



4-bit Shift Register - Component

```
library ieee;
use ieee.std_logic_1164.all;
entity shift_reg is
  port (shift_in: in std_logic;
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end shift_reg;

architecture structural of shift_reg is
component d_ff
  port(d, clock: in std_logic;
       q: out std_logic);
end component;
signal s: std_logic_vector(4 downto 0);
begin
  for i in 0 to 3 generate
    biti: d_ff port map (d=>s(i), q=>s(i+1), clock=>clk);
    q(i) <= s(i+1);
  end generate;
  s(0) <= shift_in;
end structural;
```

```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```



Generic declaration

- Declared inside entity declaration

  generic(parameter: type := default value);

  generic(n: integer := 4; m: integer := 3);

- Used to declare a fixed value for parameters
- You can actually assign the parameter value when you use the port map statement

Parameterize Models

- Allow parameters in describing the models
- Make design units more general purpose
- Allow information (such as propagation delay, size of component, etc.) to be passed into the design

Why Parameterized Model?

- Consider the two-input OR gate (or_2) we implemented
- What if you need a 3-input OR gate? Create or_3 model? …

```
-- file: or_2.vhd
library ieee;
use ieee.std_logic_1164.all;
entity or_2 is
  port (a, b: in std_logic;
        c: out std_logic);
end or_2;

architecture dataflow of or_2 is
begin
     c <= a or b;
end dataflow;
```

```
-- file: or_3.vhd
library ieee;
use ieee.std_logic_1164.all;
entity or_3 is
  port (a, b, c: in std_logic;
        d: out std_logic);
end or_3;

architecture dataflow of or_3 is
begin
     d <= a or b or c;
end dataflow;
```

VHDL Generics

- The parameterized model's behavior is determined by the values of the generic parameters.
- Declaring a parameter, added in component declaration
  - generic (parameter: type)
- Assigning a value to the parameter, added to the port map construct
  - generic map (parameter => value)

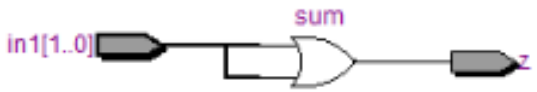
Parameterized Model

- Instead of having a model for a 2-input or gate, 3-input or gate, ….
- Model an OR gate with a parameter *n* (number of inputs)
- Problem:
  - Need to declare the input in terms of *n*
  - Need an algorithm to find the OR of *n* inputs

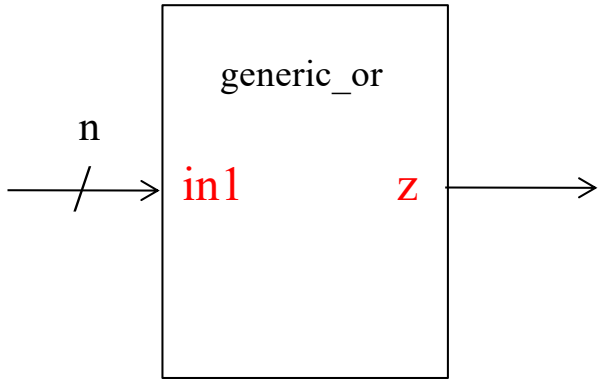
N-input OR

- Number of inputs is the parameter *n*
- Need a way to declare the *n* inputs
  - Use vector parameterized by *n*
- The OR function must be implemented based on the parameter *n*
  - Use a loop to OR each input

N-input OR

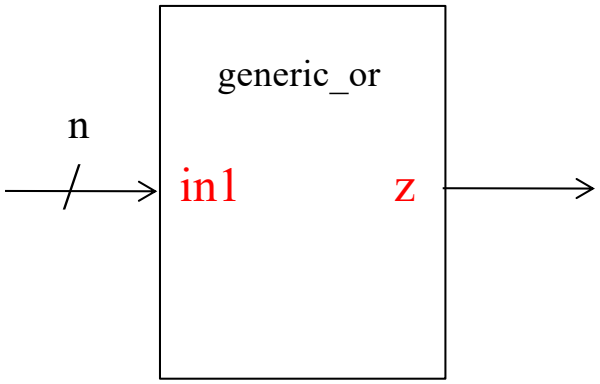



```
-- n input or
-- file: generic_or.vhd
library ieee;
use ieee.std_logic_1164.all;

entity generic_or is
  generic(n: positive := 2);  -- positive means n>0, default n=2
  port (in1: in std_logic_vector((n-1) downto 0);
        z: out std_logic);
end generic_or;

architecture behavioral of generic_or is
begin
  process(in1)
  variable sum: std_logic;
  begin
    sum := '0';
    for i in 0 to (n-1) loop
      sum := sum or in1(i);
    end loop;
    z <= sum;
  end process;
end behavioral;
```

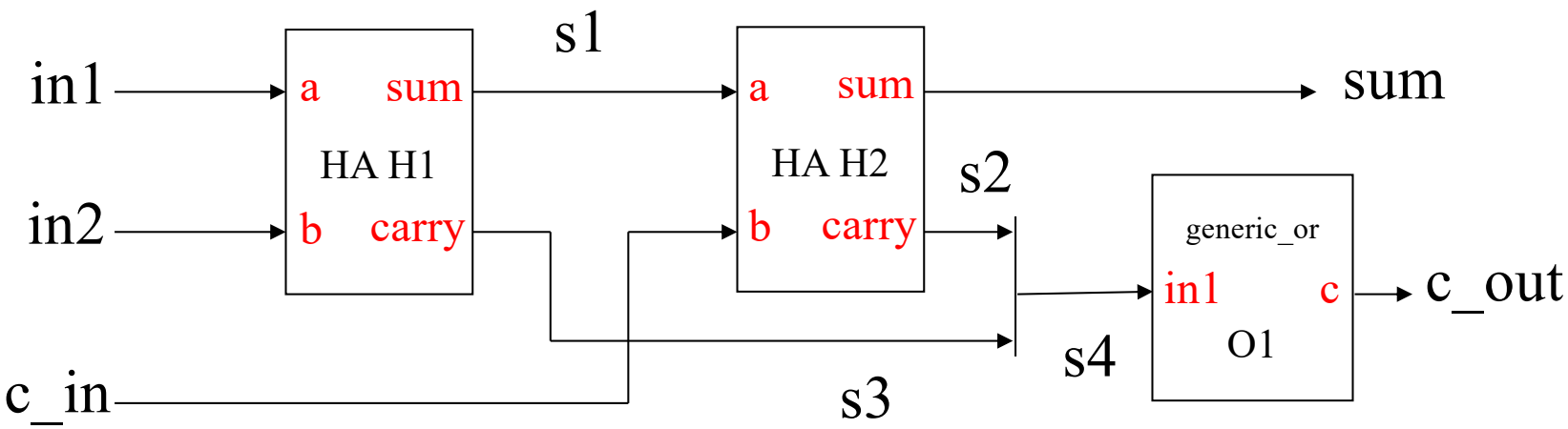


Component Declaration

```
component generic_or
  generic(n: positive);
  port (in1: in std_logic_vector((n-1) downto 0);
        z: out std_logic);
end component;
```

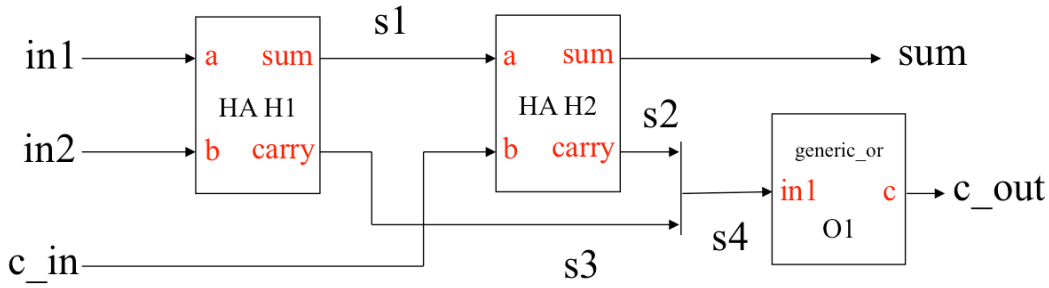


Full-Adder Circuit



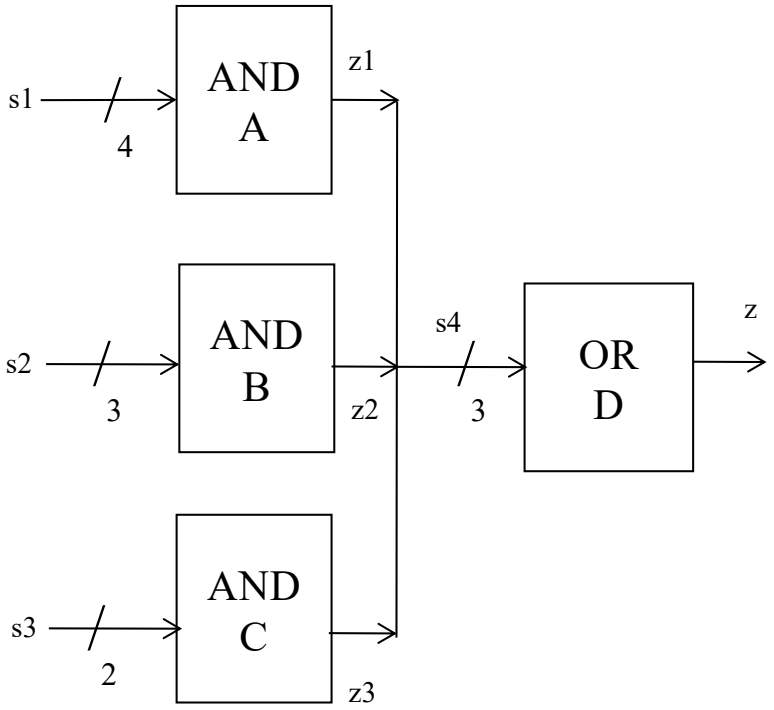
Full-Adder

```
library ieee;
use ieee.std_logic_1164.all;
entity full_adder is
  port (in1, in2, c_in: in std_logic;
        sum, c_out: out std_logic);
end full_adder;

architecture structural of full_adder is

component half_adder
  port (a, b: in std_logic;
        sum, carry: out std_logic);
end component;

component generic_or
generic (n: positive);
  port (in1: in std_logic_vector((n-1) downto 0);
        z: out std_logic);
end component;

signal s1, s2, s3: std_logic;
signal s4: std_logic_vector(1 downto 0);
begin
  HA_H1: half_adder port map(a=>in1, b=>in2, sum=>s1, carry=>s3);
  HA_H2: half_adder port map(a=>s1, b=>c_in, sum=>sum, carry=>s2);
  s4 <= s2 & s3;
  O1: generic_or generic map(n=>2) port map(in1=>s4, z=>c_out);
end structural;
```



Example

```
library ieee;
use ieee.std_logic_1164.all;
entity example is
  port (s1: in std_logic_vector(3 downto 0);
        s2: in std_logic_vector(2 downto 0);
        s3: in std_logic_vector(1 downto 0);
        z: out std_logic);
end example;

architecture structural of example is

component generic_and
  generic (n: positive);
  port (in1: in std_logic_vector((n-1) downto 0);
        z: out std_logic);
end component;

component generic_or
  generic (n: positive);
  port (in1: in std_logic_vector((n-1) downto 0);
        z: out std_logic);
end component;

signal s4: std_logic_vector(2 downto 0);
signal z1, z2, z3: std_logic;
begin
  A: generic_and generic map(n=>4) port map(in1=>s1, z=>z1);
  B: generic_and generic map(n=>3) port map(in1=>s2, z=>z2);
  C: generic_and generic map(n=>2) port map(in1=>s3, z=>z3);
  s4 <= z1 & z2 & z3;
  D: generic_or generic map(n=>3) port map(in1=>s4, z=>z);
end structural;
```



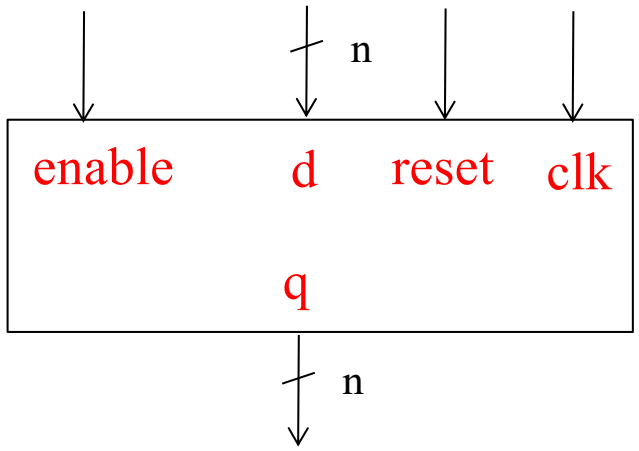
4-bit Register

- Inputs – d, bits 3 down to 0
- Outputs – q, bits 3 down to 0
- Controls – enable, reset
- Use behavioral modeling

clk	reset	enable	d	q
X	1	X	X	0
↑	0	1		d
X	0	0	X	q

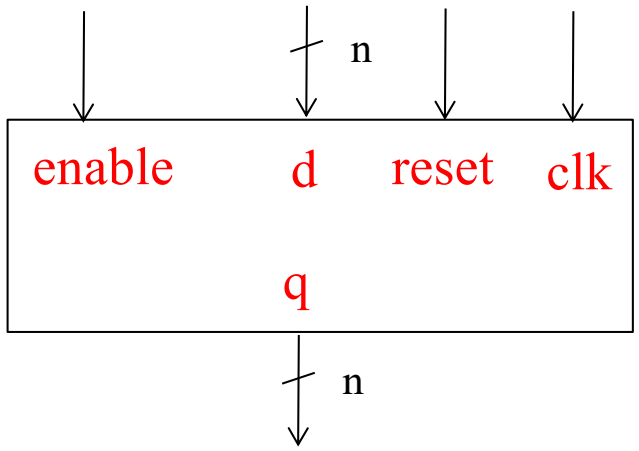
4-bit Register - Behavioral

```
library ieee;
use ieee.std_logic_1164.all;
entity four_bit_reg is
port (clk, reset, enable: in std_logic;
       d: in std_logic_vector(3 downto 0);
       q: out std_logic_vector(3 downto 0));
end four_bit_reg;

architecture behavioral of four_bit_reg is
begin
  reg_process: process (clk, reset)
  begin
    if reset = '1' then
      q <= "0000"
      elsif (rising_edge(clk)) then
        if enable = '1' then
          q <= d;
        end if;
    end if;
  end process reg_process;
end behavioral;
```



4-bit Register - Behavioral

```
library ieee;
use ieee.std_logic_1164.all;
entity four_bit_reg is
port (clk, reset, enable: in std_logic;
       d: in std_logic_vector(3 downto 0);
       q: out std_logic_vector(3 downto 0));
end four_bit_reg;

architecture behavioral of four_bit_reg is
begin
  reg_process: process (clk, reset)
  begin
    if reset = '1' then
      q <= "0000"
    elsif (rising_edge(clk)) then
      if enable = '1' then
        q <= d;
      end if;
    end if;
  end process reg_process;
end behavioral;
```



n-bit Register

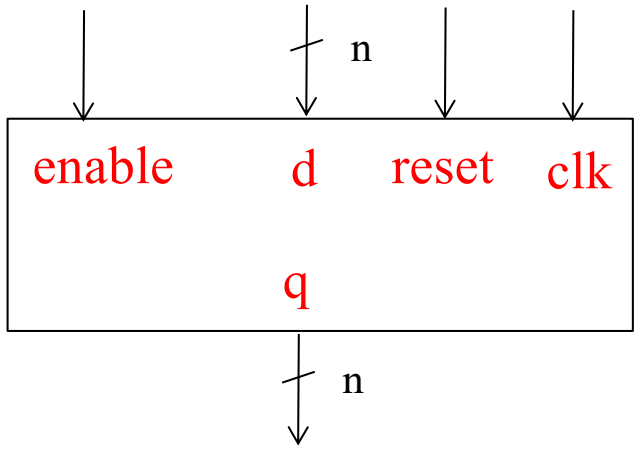
- Parameterize number of bits, n
- Inputs – d, bits n-1 down to 0
- Outputs – q, bits n-1 down to 0
- Controls – enable, reset
- Use behavioral modeling

clk	reset	enable	d	q
X	1	X	X	0
↑	0	1		d
X	0	0	X	q

N-bit Register - Behavioral

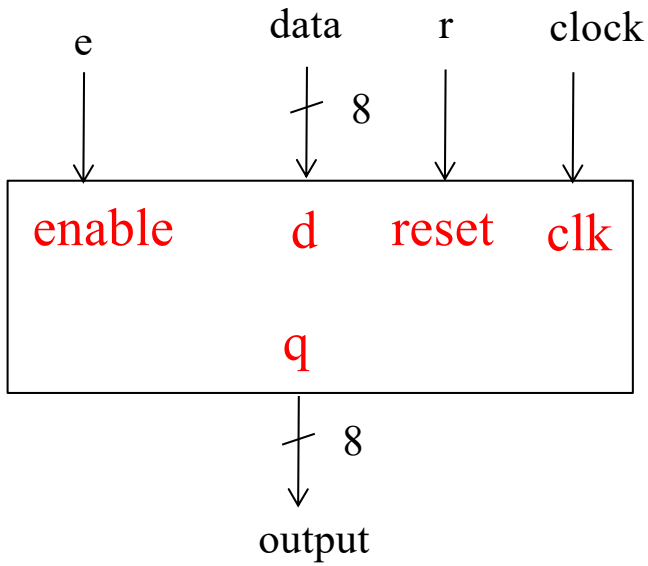
```
library ieee;
use ieee.std_logic_1164.all;
entity generic_reg is
 generic (n: positive);
 port (clk, reset, enable: in std_logic;
       d: in std_logic_vector(n-1 downto 0);
       q: out std_logic_vector(n-1 downto 0));
end generic_reg;

architecture behavioral of generic_reg is
begin
  reg_process: process (clk, reset)
  begin
    if reset= '1' then
      q <= (others=>'0');
      elsif (rising_edge(clk)) then
        if enable = '1' then
          q <= d;
        end if;
    end if;
  end process reg_process;
end behavioral;
```

"000 … 0"



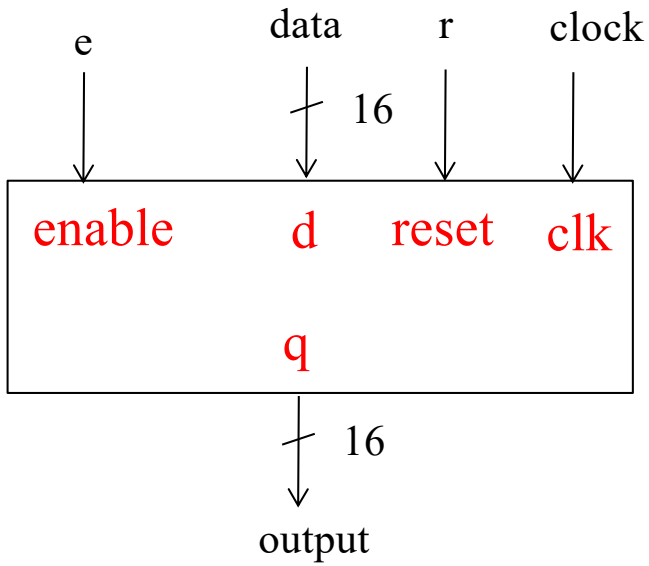
8-bit Register

```
signal clock, r, e: std_logic;
signal data, out: std_logic_vector(7 downto 0);

. . .

label1: generic_reg
    generic map(n => 8);
    port map(clk=>clock;, reset=>r, enable=>e,
             d=>data, q=> output);
```



16-bit Register

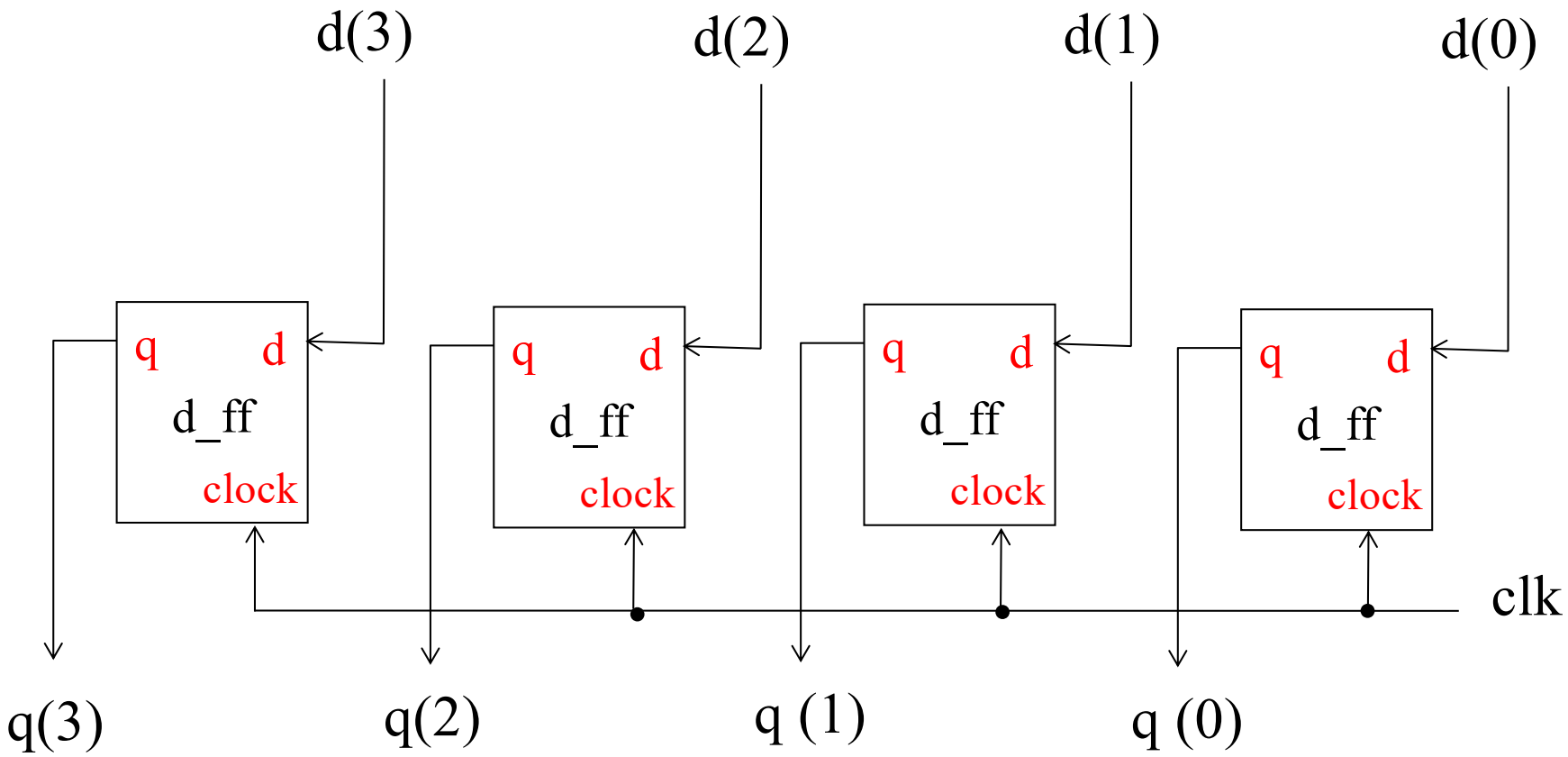
```
signal clock, r, e: std_logic;
signal data, output: std_logic_vector(15 downto 0);

. . .

label1: generic_reg
    generic map(n => 16);
    port map(clk=>clock, reset=>r, enable=>e,
             d=>data, q=> output);
```



Parameterized Model for Structural Modeling

- What if you want to parameterize model created with structural modeling?
- How to specific n port map statements?

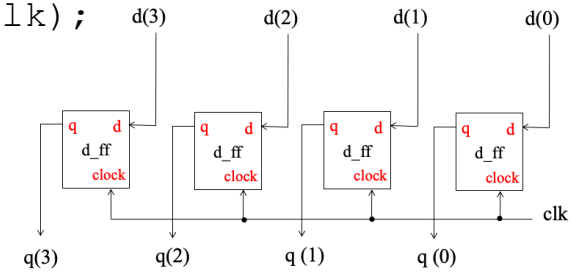
4-bit Register using D FF



4-bit Register - Component

```
library ieee;
use ieee.std_logic_1164.all;
entity dregister is
  port (d: in std_logic_vector(3 downto 0);
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end dregister;

architecture structural of dregister is
component d_ff
  port(d, clock: in std_logic;
       q: out std_logic);
end component;

begin
  bit0: d_ff port map (d=>d(0), q=>q(0), clock=>clk);
  bit1: d_ff port map (d=>d(1), q=>q(1), clock=>clk);
  bit2: d_ff port map (d=>d(2), q=>q(2), clock=>clk);
  bit3: d_ff port map (d=>d(3), q=>q(3), clock=>clk);
end structural;
```

```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```



n-bit Register - Component

```
library ieee;
use ieee.std_logic_1164.all;
entity dregister is
  generic (n: positive);
  port (d: in std_logic_vector(n-1 downto 0);
        q: out std_logic_vector(n-1 downto 0);
        clk: in std_logic);
end dregister;

architecture structural of dregister is
component d_ff
  port(d, clock: in std_logic;
       q: out std_logic);
end component;

begin

        PORT MAP statements

end structural;
```

```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```

n-bit Register - Component

```
library ieee;
use ieee.std_logic_1164.all;
entity dregister is
  generic (n: positive);
  port (d: in std_logic_vector(n-1 downto 0);
        q: out std_logic_vector(n-1 downto 0);
        clk: in std_logic);
end dregister;

architecture structural of dregister is
component d_ff
  port(d, clock: in std_logic;
       q: out std_logic);
end component;

begin
  bit0: d_ff port map (d=>d(0), q=>q(0), clock=>clk);
  bit1: d_ff port map (d=>d(1), q=>q(1), clock=>clk);
  bit2: d_ff port map (d=>d(2), q=>q(2), clock=>clk);
  bit3: d_ff port map (d=>d(3), q=>q(3), clock=>clk);
  . . .
end structural;
```

```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```

Need n port map statements

Generate

- Allow well- patterned structures to be created easily.
- Any VHDL concurrent statement can be included in a GENERATE statement, including another GENERATE statement.
- Reduce the number of statements

For … Generate

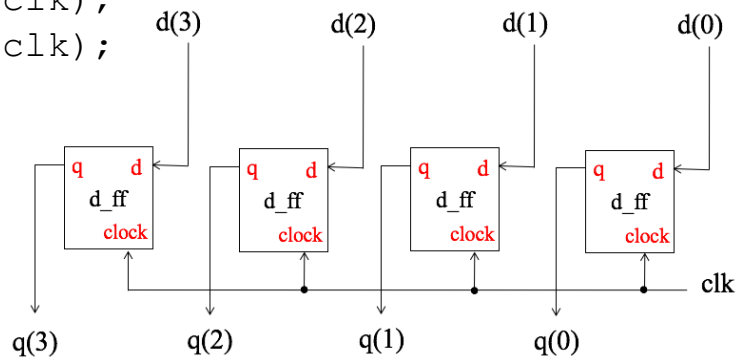
- Instantiates identical components

```
label: for index in range generate
        statements
end generate;
```

4-bit Register - Component

```
library ieee;
use ieee.std_logic_1164.all;
entity d_register is
  port (d: in std_logic_vector(3 downto 0);
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end d_register;

architecture structural of d_register is
component d_ff
  port(d, clock: in std_logic;
       q: out std_logic);
end component;

begin
  bit0: d_ff port map (d=>d(0), q=>q(0), clock=>clk);
  bit1: d_ff port map (d=>d(1), q=>q(1), clock=>clk);
  bit2: d_ff port map (d=>d(2), q=>q(2), clock=>clk);
  bit3: d_ff port map (d=>d(3), q=>q(3), clock=>clk);
end structural;
```

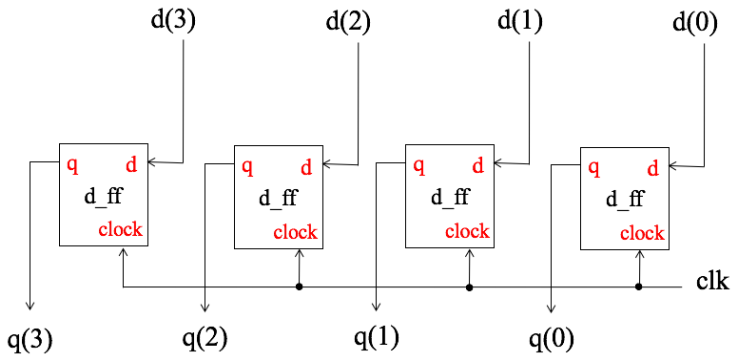
```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```



4-bit Register - Component

```
library ieee;
use ieee.std_logic_1164.all;
entity d_register is
  port (d: in std_logic_vector(3 downto 0);
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end d_register;

architecture structural of d_register is
component d_ff
  port(d, clock: in std_logic;
       q: out std_logic);
end component;

begin
  L: for i in 0 to 3 generate
  bit1: d_ff port map (d=>d(i), q=>q(i), clock=>clk);
  end generate;
end structural;
```

```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```



n-bit Register - Component

```
library ieee;
use ieee.std_logic_1164.all;
entity dregister is
  generic (n: positive);
  port (d: in std_logic_vector(n-1 downto 0);
        q: out std_logic_vector(n-1 downto 0);
        clk: in std_logic);
end dregister;

architecture structural of dregister is
component d_ff
  port(d, clock: in std_logic;
       q: out std_logic);
end component;

begin
  L: for i in 0 to n-1 generate
     bit1: d_ff port map (d=>d(i), q=>q(i), clock=>clk);
  end generate;
end structural;
```

```
-- file: d_ff.vhd
library ieee;
use ieee.std_logic_1164.all;
entity d_ff is
  port (clock, d: in std_logic;
        q: out std_logic);
end d_ff;

architecture behavior of d_ff is
begin
  process
  begin
    wait until(rising_edge(clock));
    q <= d;
  end process;
end behavior;
```

Using Generate – case 1

```
-- Assign c(0)<=a(0), c(1)<=b(0), c(2)<=a(1), c(3)<=b(1), …
--

entity gen1 is
  port (a, b: in bit(3 downto 0);
        c: out bit(7 downto 0));
end gen1;

architecture try1 of gen1 is

begin
 c(0) <= a(0); c(1) <= b(0);
 c(2) <= a(1); c(3) <= b(1);
 c(4) <= a(2); c(5) <= b(2);
 c(6) <= a(3); c(7) <= b(3);
end try1;
```

Using Generate – case 2

```
-- Assign c(0)<=a(0), c(1)<=b(0), c(2)<=a(1), c(3)<=b(1), …
--

entity gen1 is
  port (a, b: in bit(3 downto 0);
        c: out bit(7 downto 0));
end gen1;

architecture try2 of gen1 is

begin

  label1: for i in 0 to 3 generate
     c(2*i) <= a(i);
  end generate;

  label2: for i in 0 to 3 generate
     c(2*i+1) <= b(i);
  end generate;

end try2;
```

If *condition* Generate

- Use inside for – generate to take care of boundary cases.

```
label: if condition generate
                    statements
               end generate;
```

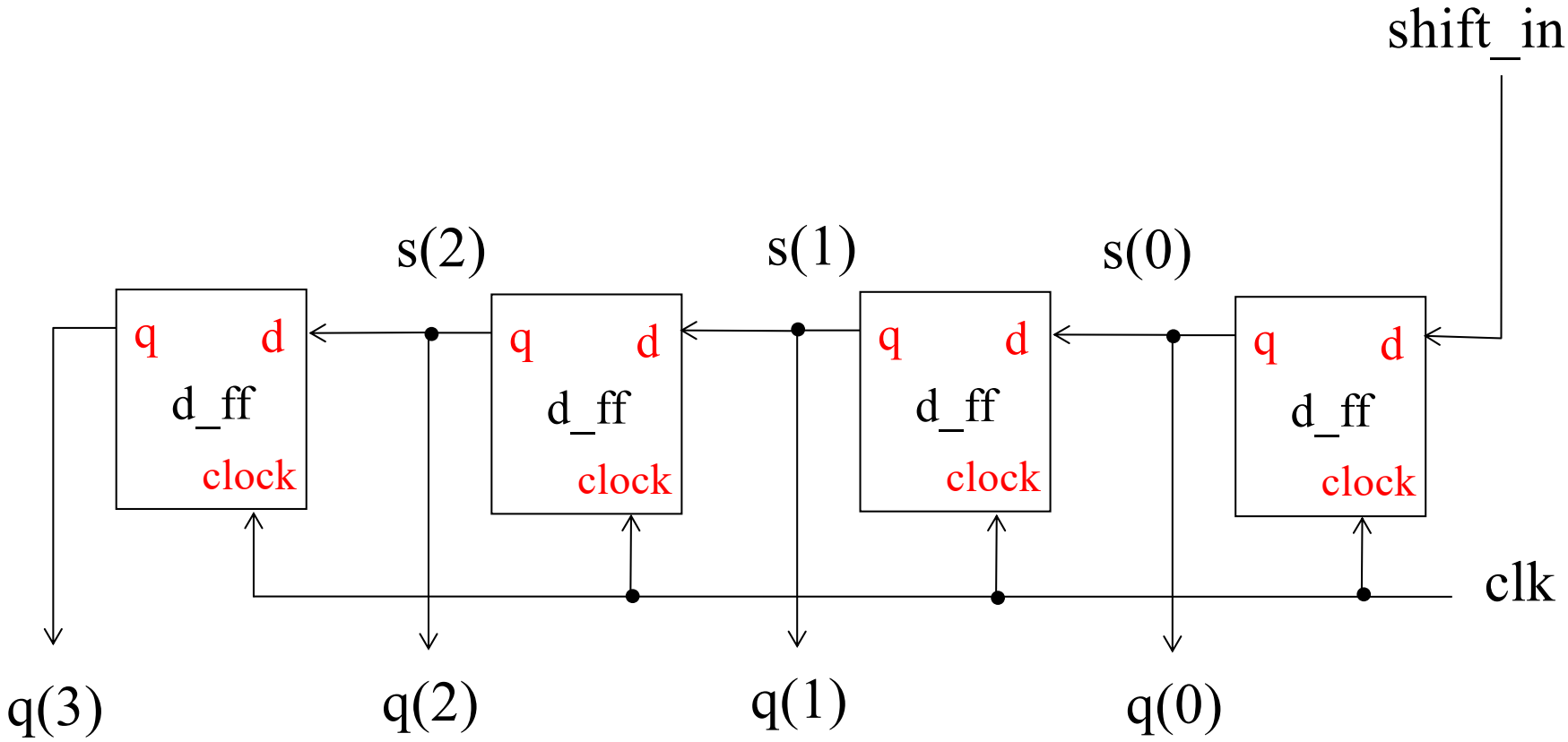
Using Generate – case 3

```
-- Assign c(0)<=a(0), c(1)<=b(0), c(2)<=a(1), c(3)<=b(1), …
--

entity gen1 is
  port (a, b: in bit(3 downto 0);
        c: out bit(7 downto 0));
end gen1;

architecture try3 of gen1 is

begin

   label0: for i in 0 to 7 generate
     label1: if i = 2*(i/2) generate
         c(i) <= a(i/2);
     end generate;

     label2: if i = 2*(i/2)+1 generate
         c(i) <= b(i/2);
     end generate;
  end generate;

end try3;
```

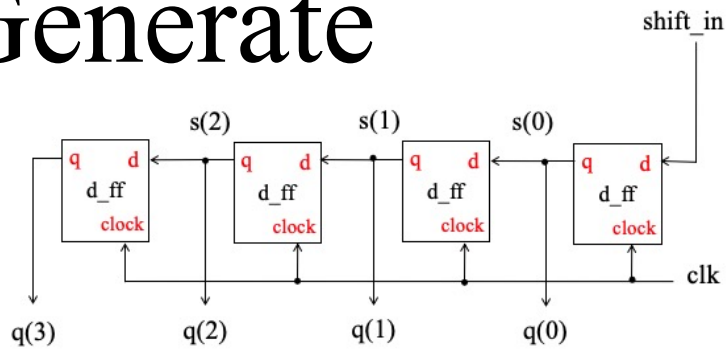
Guidelines For Generate

- Structure the array of components to be indexed
- Declare signal vectors local to the architecture to interconnect the components
- Determine the range of components that can be encapsulated within a generate statement
- Write one component instantiation statement

4-bit Shift Register

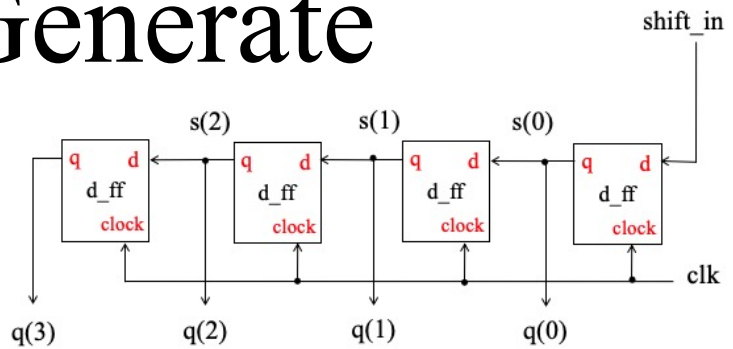


N-bit Register - Generate

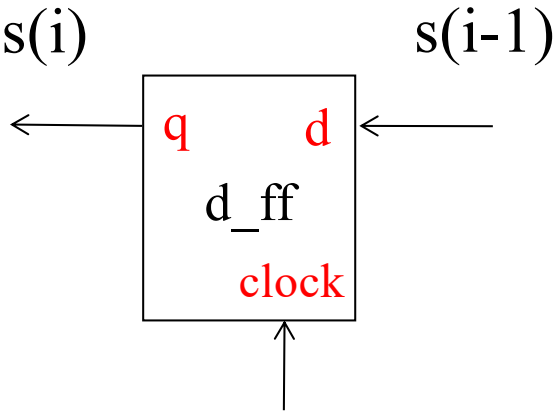



```
library ieee;
use ieee.std_logic_1164.all;
entity shift_reg is
  port (shift_in: in std_logic;
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end shift_reg;

architecture structural of shift_reg is
  component d_ff
    port(d, clock: in std_logic;
         q: out std_logic);
  end component;

  signal s: std_logic_vector(0 to 3);
begin
  reg: for i in 3 downto 0 generate


  end generate;
end structural;
```

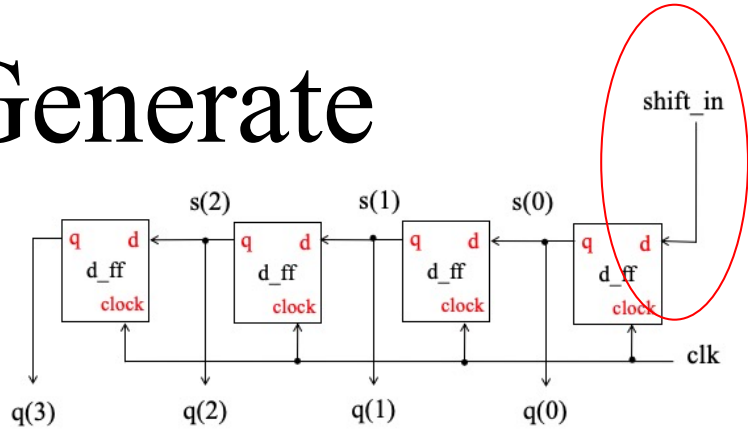
N-bit Register - Generate



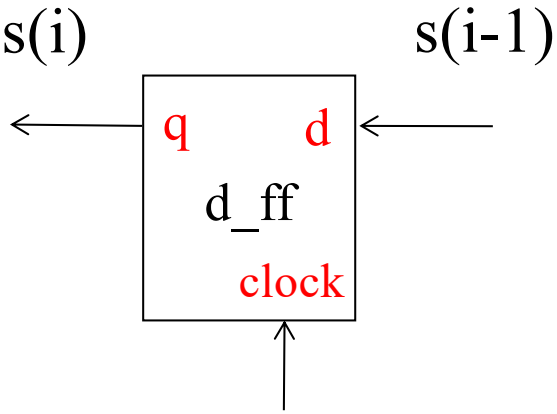

```
library ieee;
use ieee.std_logic_1164.all;
entity shift_reg is
  port (shift_in: in std_logic;
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end shift_reg;

architecture structural of shift_reg is
  component d_ff
    port(d, clock: in std_logic;
         q: out std_logic);
  end component;

  signal s: std_logic_vector(0 to 3);
begin
  reg: for i in 3 downto 0 generate
    dbit: d_ff port map(d=>s(i-1), q=>s(i), clock=>clk);
    q(i) <= s(i);
  end generate;
end structural;
```



N-bit Register - Generate




```
library ieee;
use ieee.std_logic_1164.all;
entity shift_reg is
  port (shift_in: in std_logic;
        q: out std_logic_vector(3 downto 0);
        clk: in std_logic);
end shift_reg;

architecture structural of shift_reg is
  component d_ff
    port(d, clock: in std_logic;
         q: out std_logic);
  end component;

  signal s: std_logic_vector(0 to 3);
begin
  reg: for i in 3 downto 0 generate
    dbit: d_ff port map(d=>s(i-1), q=>s(i), clock=>clk);
    q(i) <= s(i);
  end generate;
end structural;
```



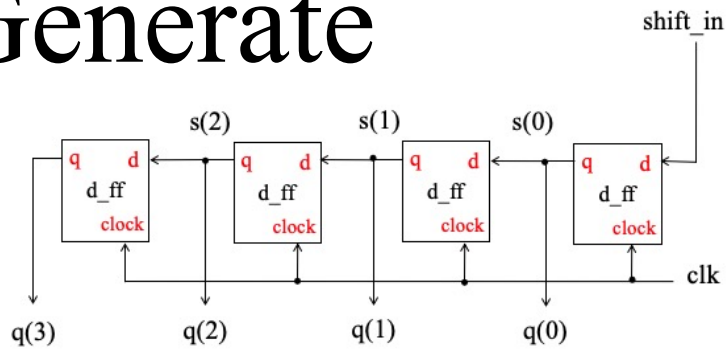
i=0, d => shift_in

If *condition* Generate

- Use inside for – generate to take care of boundary cases.

```
label: if condition generate
           begin
                statements
           end generate;
```

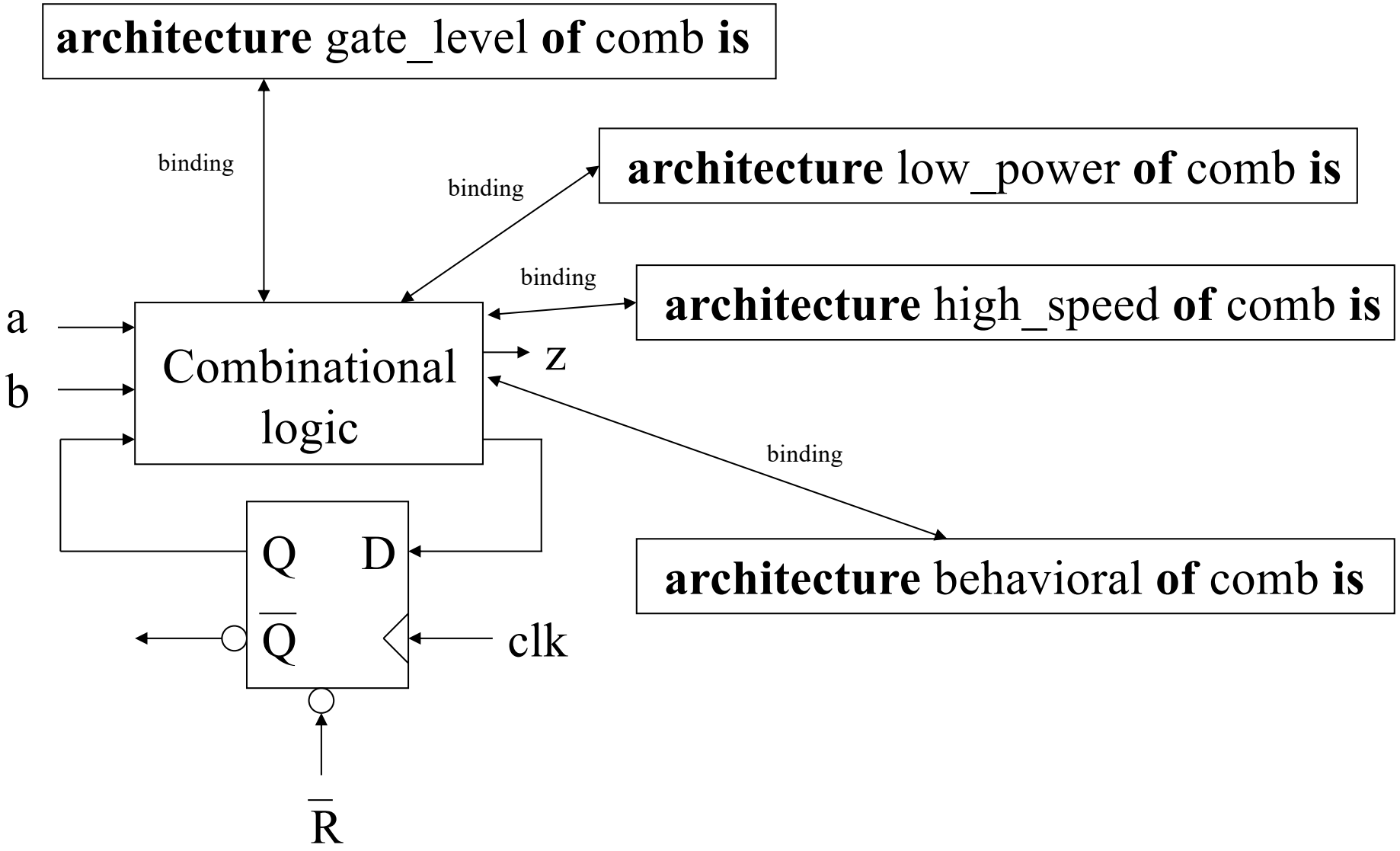
N-bit Register - Generate



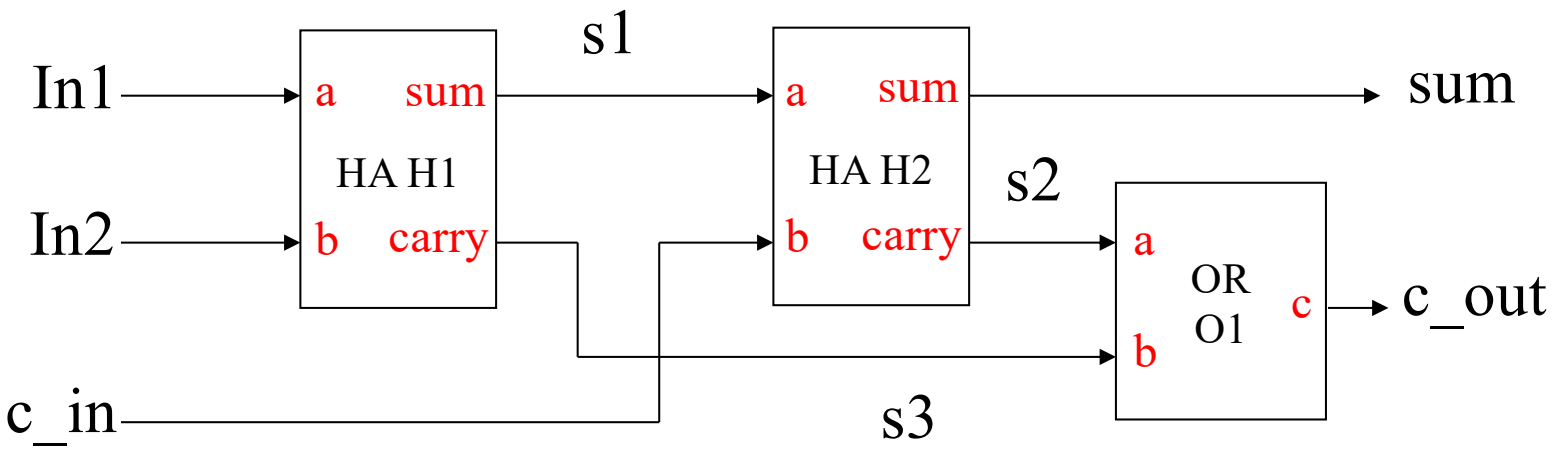

```
  signal s: std_logic_vector(0 to 3);
begin
  reg: for i in 3 downto 0 generate

        L1: if (i=0) generate
           begin
            dbit: d_ff port map(d=>shift_in, q=>s(0), clock=>clk);
           end generate;

        L2: if (i>0) generate
           begin
            dbit: d_ff port map(d=>s(i-1), q=>s(i), clock=>clk);
           end generate;

        q(i) <= s(i);

      end generate;
end structural;
```

VHDL Configuration

- Multiple architecture body to describe different implementation
- How to specify which architecture is to be used?
- Configuration statement

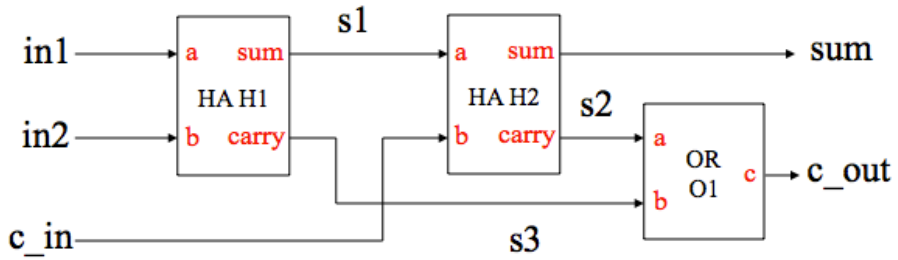

architecture gate_level of comb is
binding
architecture low_power of comb is
binding
architecture high_speed of comb is
binding
a
b
Combinational logic
z
binding
architecture behavioral of comb is
Q
D
Q
clk
R

Full-Adder Circuit

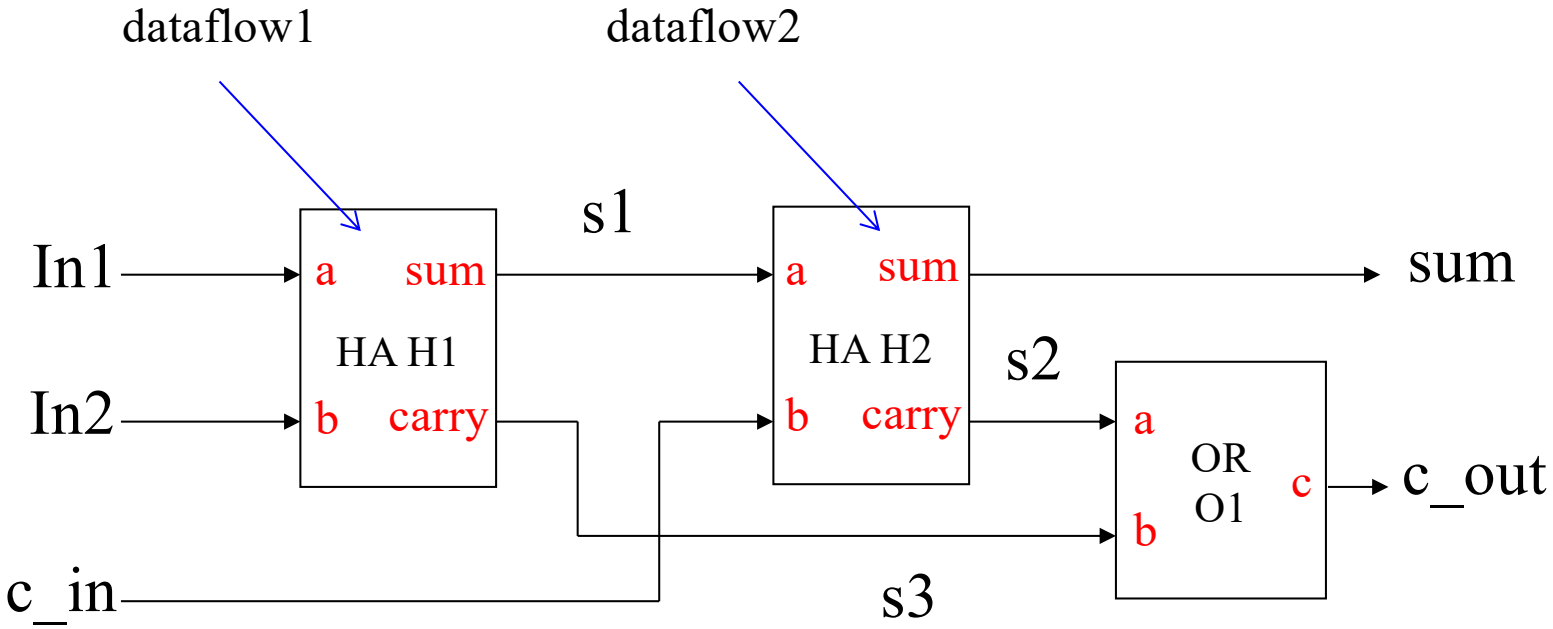


```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity full_adder is
  port (in1, in2, c_in: in std_logic;
        sum, c_out: out std_logic);
end full_adder;

architecture structural of full_adder is

component half_adder
  port (a, b: in std_logic;
        sum, carry: out std_logic);
end component;

component or_2
  port (a, b: in std_logic;
        c: out std_logic);
end component;

signal s1, s2, s3: std_logic;
begin
   HA_H1: half_adder port map(a=>in1, b=>in2, sum=>s1, carry=>s3);
   HA_H2: half_adder port map(a=>s1, b=>c_in, sum=>sum, carry=>s2);
   OR_O1: or_2 port map(a=>s2, b=>s3, c=>c_out);
end structural;
```

```vhdl
-- file: half_adder.vhd
library ieee;
use ieee.std_logic_1164.all;
entity half_adder is
  port (a, b: in std_logic;
        sum, carry: out std_logic);
end half_adder;

architecture dataflow of half_adder is
begin
    sum <= a xor b;
    carry <= a and b;
end dataflow;

-- file: or_2.vhd
library ieee;
use ieee.std_logic_1164.all;
entity or_2 is
  port (a, b: in std_logic;
        c: out std_logic);
end or_2;

architecture dataflow of or_2 is
begin
    c <= a or b;
end dataflow;
```

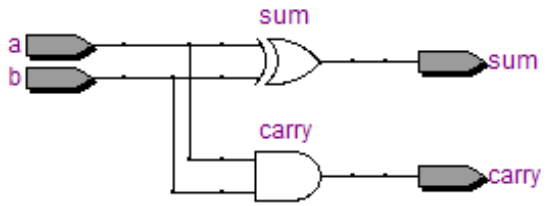


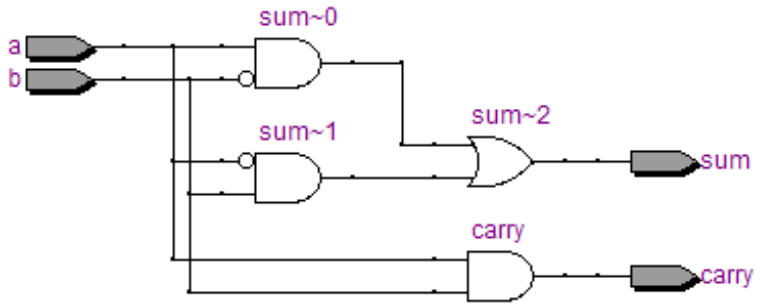
Full-Adder Circuit



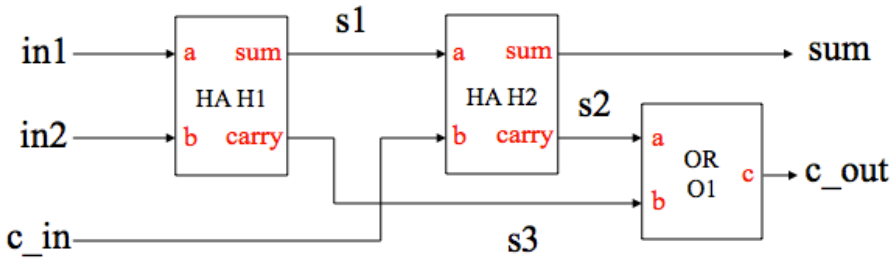
Half-Adder Circuit

```
library ieee;
use ieee.std_logic_1164.all;
entity half_adder is
  port (a, b: in std_logic;
        sum, carry: out std_logic);
end half_adder;

architecture dataflow1 of half_adder is
begin
   sum <= a xor b;
   carry = a and b;
end dataflow1;

architecture dataflow2 of half_adder is
begin
   sum <= ((not a) and b) or (a and (not b));
   carry = a and b;
end dataflow2;
```





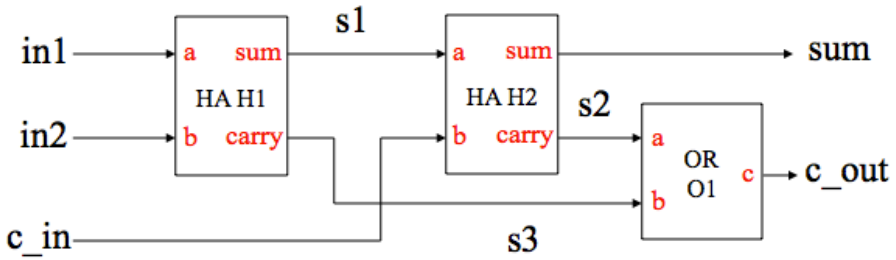
```
library ieee;
use ieee.std_logic_1164.all;
entity full_adder is
  port (in1, in2, c_in: in std_logic;
        sum, c_out: out std_logic);
end full_adder;

architecture structural of full_adder is

component half_adder
  port (a, b: in std_logic;
        sum, carry: out std_logic);
end component;

component or_2
  port (a, b: in std_logic;
        c: out std_logic);
end component;

signal s1, s2, s3: std_logic;

begin
  H1: half_adder port map(a=>in1, b=>in2, sum=>s1, carry=>s3);
  H2: half_adder port map(a=>s1, b=>c_in, sum=>sum, carry=>s2);
  O1: or_2 port map(a=>s2, b=>s3, c=>c_out);
end structural;
```

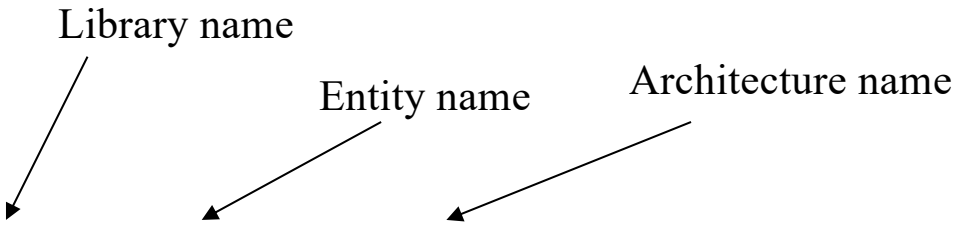


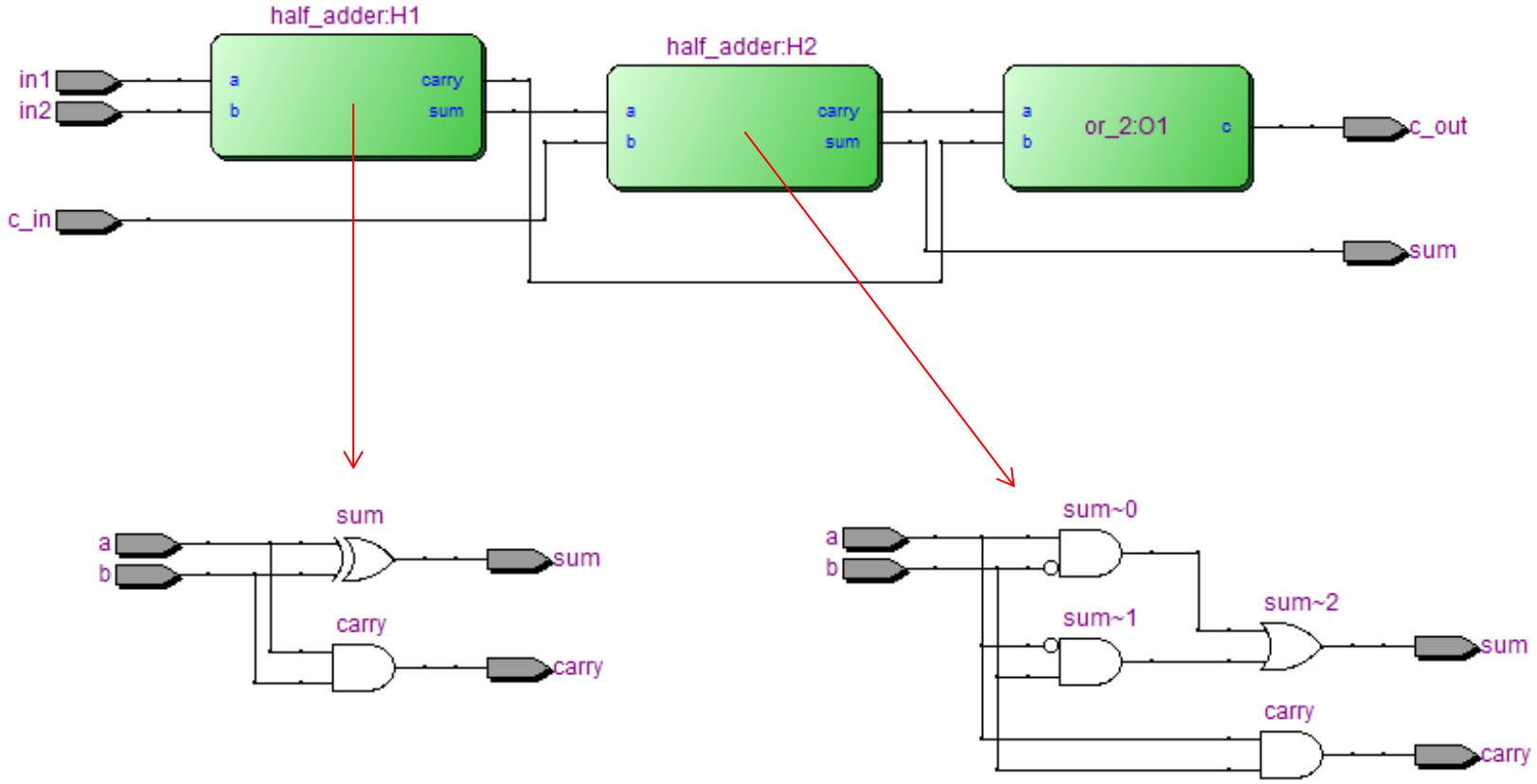

```
library ieee;
use ieee.std_logic_1164.all;
entity half_adder is
  port (a, b: in std_logic;
        sum, carry: out std_logic);
end half_adder;

architecture dataflow1 of half_adder is
begin
   sum <= a xor b;
   carry = a and b;
end dataflow1;

architecture dataflow2 of half_adder is
begin
   sum <= ((not a) and b) or (a and (not b));
   carry = a and b;
end dataflow2;
```

Use the last architecture body

```
library ieee;
use ieee.std_logic_1164.all;
entity full_adder is
  port (in1, in2, c_in: in std_logic;
        sum, c_out: out std_logic);
end full_adder;

architecture structural of full_adder is

component half_adder
  port (a, b: in std_logic;
        sum, carry: out std_logic);
end component;

component or_2
  port (a, b: in std_logic;
        c: out std_logic);
end component;

for H1: half_adder use entity WORK.half_adder(dataflow1);
for H2: half_adder use entity WORK.half_adder(dataflow2);

signal s1, s2, s3: std_logic;

begin
  H1: half_adder port map(a=>in1, b=>in2, sum=>s1, carry=>s3);
  H2: half_adder port map(a=>s1, b=>c_in, sum=>sum, carry=>s2);
  O1: or_2 port map(a=>s2, b=>s3, c=>c_out);
end structural;
```






half_adder:H1
in1
in2
a
b
carry
sum
half_adder:H2
c_in
a
b
carry
sum
a
b
or_2:O1
c
c_out
sum
sum
a
b
sum
carry
sum
carry
sum~0
a
b
sum~1
sum~2
sum
carry
carry

Configuration

- If all the half adder uses the same components

```
for all: half_adder use entity WORK.half_adder(behavioral);
```

Configuration Declaration

- Configuration declaration can be placed in a separate file

```
configuration config_a of full_adder is
  for structural
    for H1: half_adder use entity WORK.half_adder(behavioral);
    end for;

    for H2: half_adder use entity WORK.half_adder(structural);
    end for;
  end for;
end config_a;
```

Library of Parameterized Modules (lpm)

- Technology-independent library of logic functions
- Parameterized functions and scalable
  - lpm_mux - multiplexer
  - lpm_counter - counter
  - lpm_ff - flip-flops
  - lpm_rom - ROM
  - and more

lpm Library

- Declare library

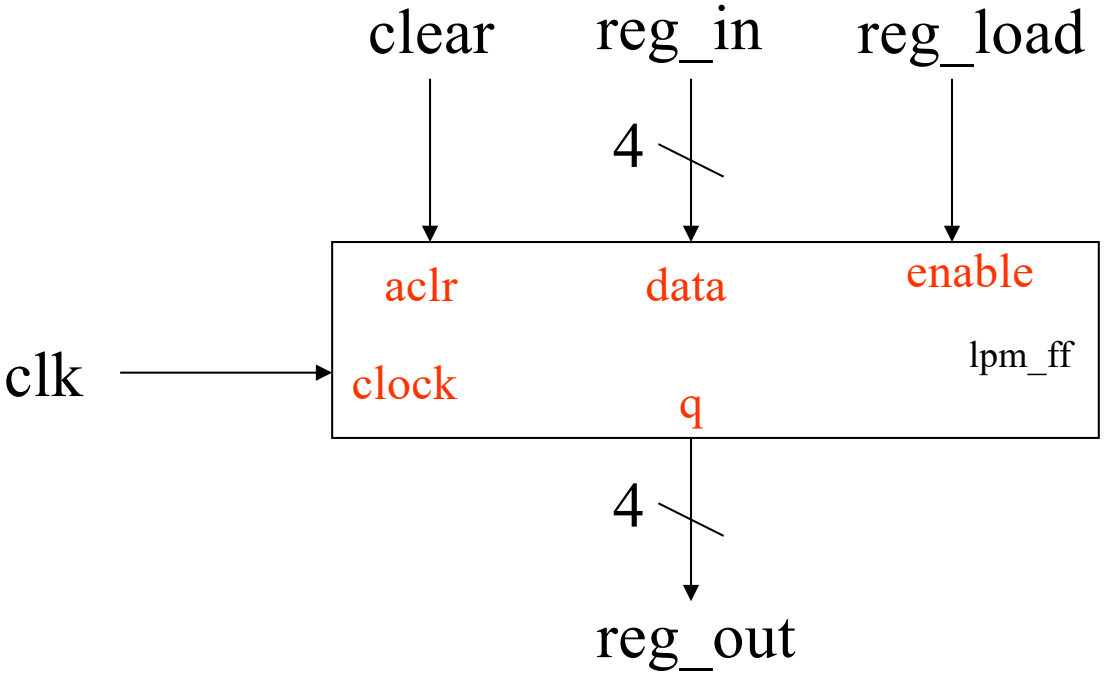
```
library lpm;
use lpm.lpm_components.all;
```

lpm_ff

- Generic
  - lpm_width = number of flip flops
- Ports
  - data[] = input
  - q[] = output
  - aclr = asynchronous clear
  - clock = clock
  - enable = clock enable

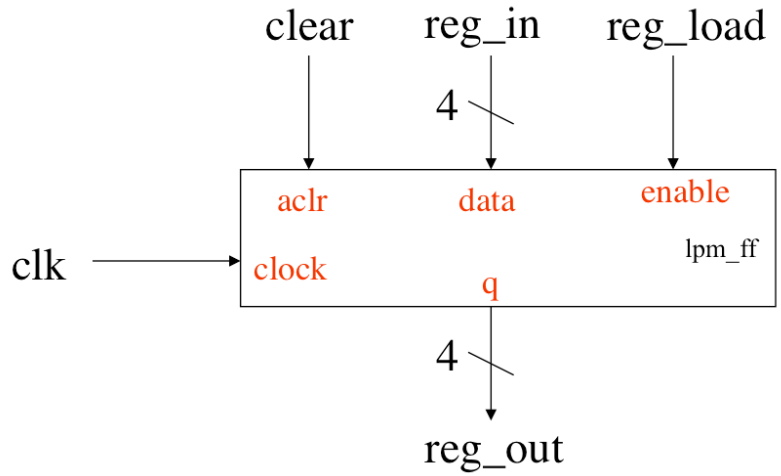
Default is d flip-flop, load on rising clock edge

4-bit Register



4-bit Register

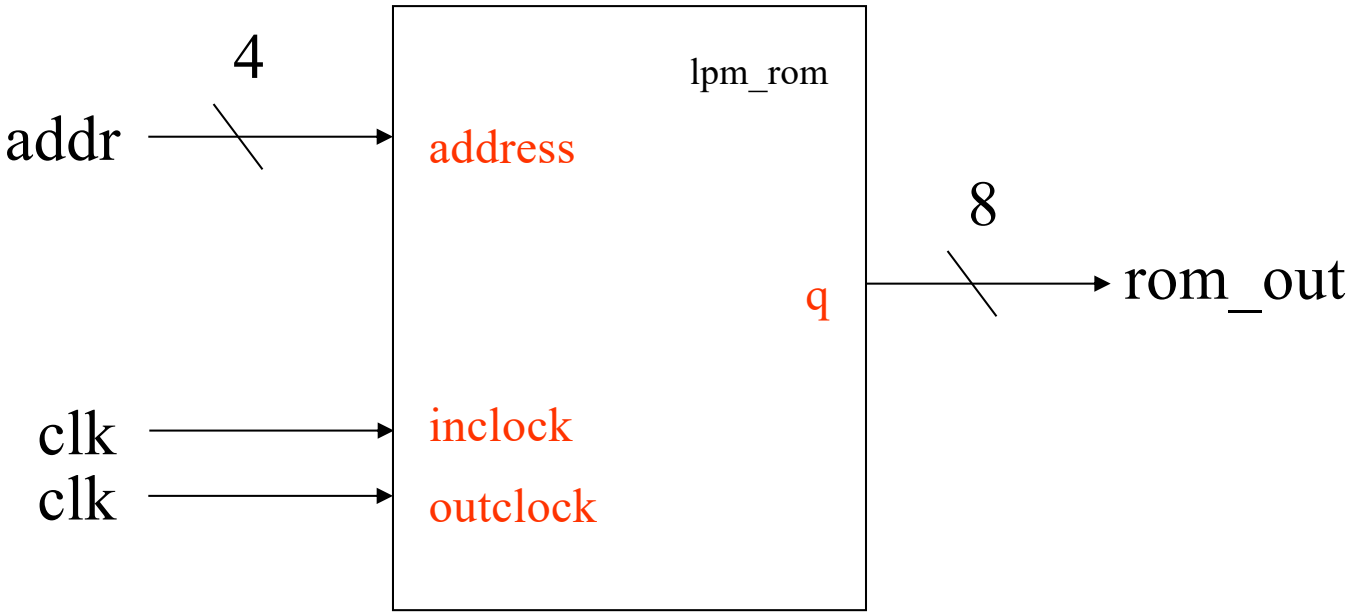
```
signal reg_in, reg_out: std_logic_vector(3 downto 0);
signal clk, reg_load, clear: std_logic;

reg1: lpm_ff
    generic map(lpm_width=>4)
    port map(data=>reg_in, q=>reg_out, clock=>clk,
             enable=>reg_load, aclr=>clear);
```



lpm_rom

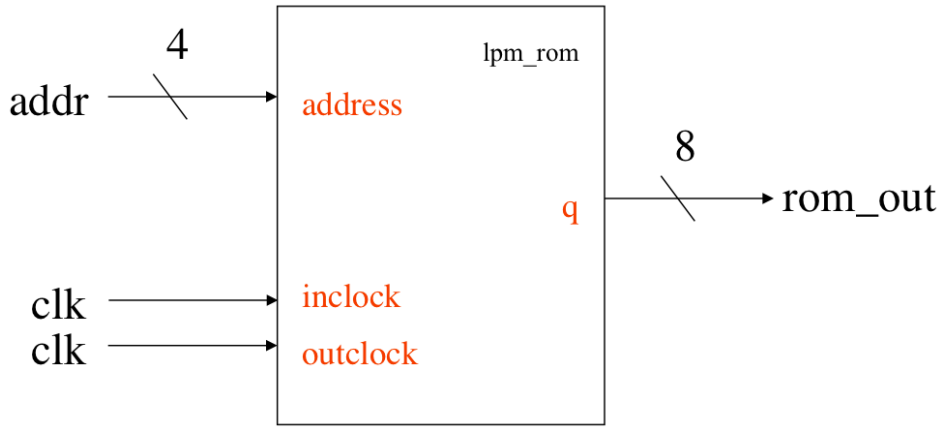
- Generics
  - lpm_widthad = width of address
  - lpm_width = width of data
  - lpm_file = data file
- Ports
  - address[] = address
  - q[] = output
  - inclock = synchronous address
  - outclock = synchronous output

16x8 ROM

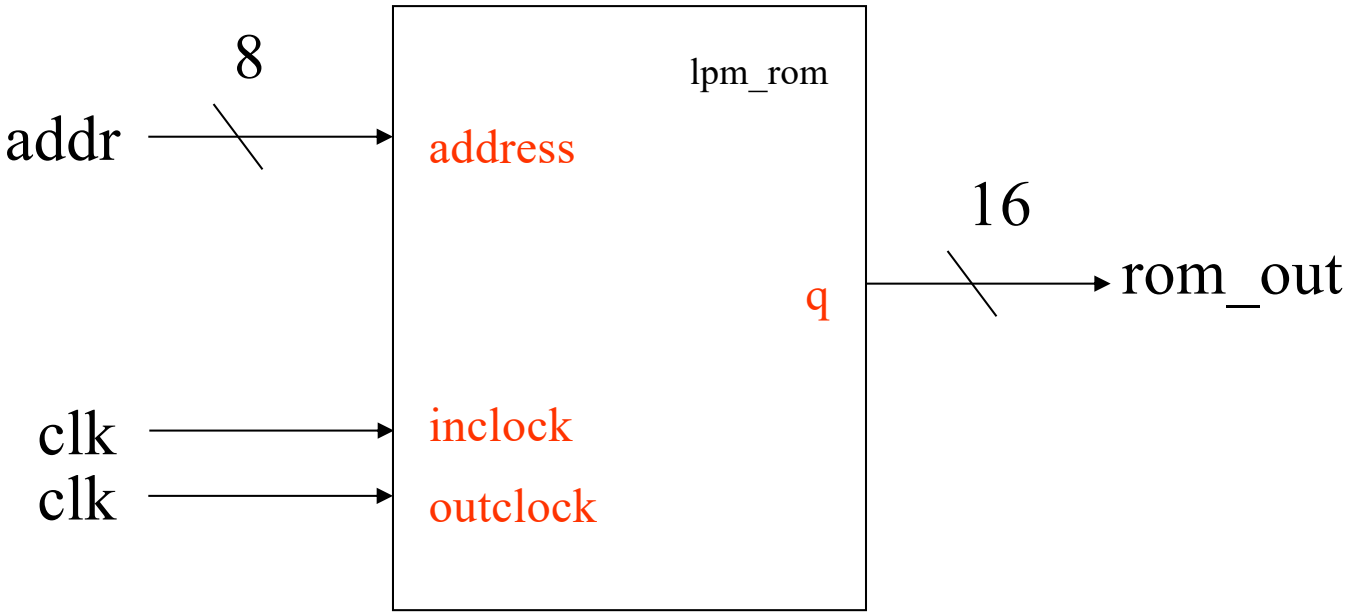


16x8 ROM

```
signal addr: std_logic_vector(3 downto 0);
signal rom_out: std_logic_vector(7 downto 0);

ROM1: lpm_rom
  generic map(lpm_widthad=>4, lpm_width=>8, lpm_file=>"r.mif")
  port map(address=>addr, q=>rom_out, inclock=>clk,
                outclock=>clk);
```

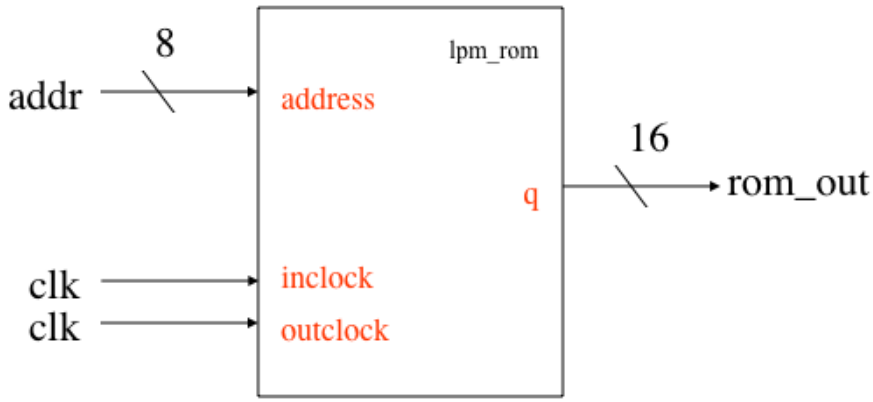


256x16 ROM



256x16 ROM

```
signal addr: std_logic_vector(7 downto 0);
signal rom_out: std_logic_vector(15 downto 0);

ROM2: lpm_rom
  generic map(lpm_widthad=>8, lpm_width=>16, lpm_file=>"r.mif")
  port map(address=>addr, q=>rom_out, inclock=>clk,
        outclock=>clk);
```



ROM data file

- Memory Initialization File - .mif extension

depth = number of ROM entries

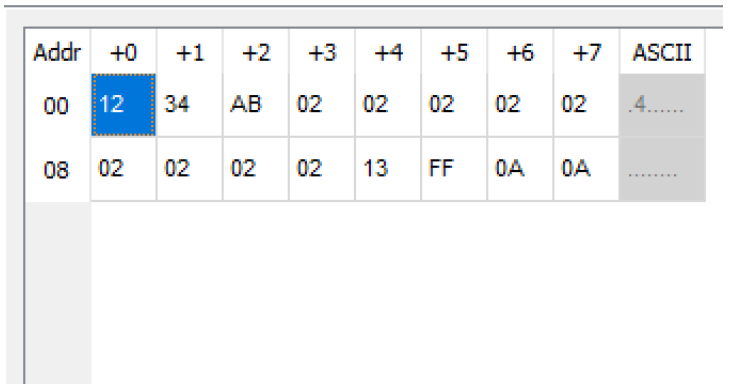
width = size of data

address_radix = radix used for address

data_radix = radix used for data

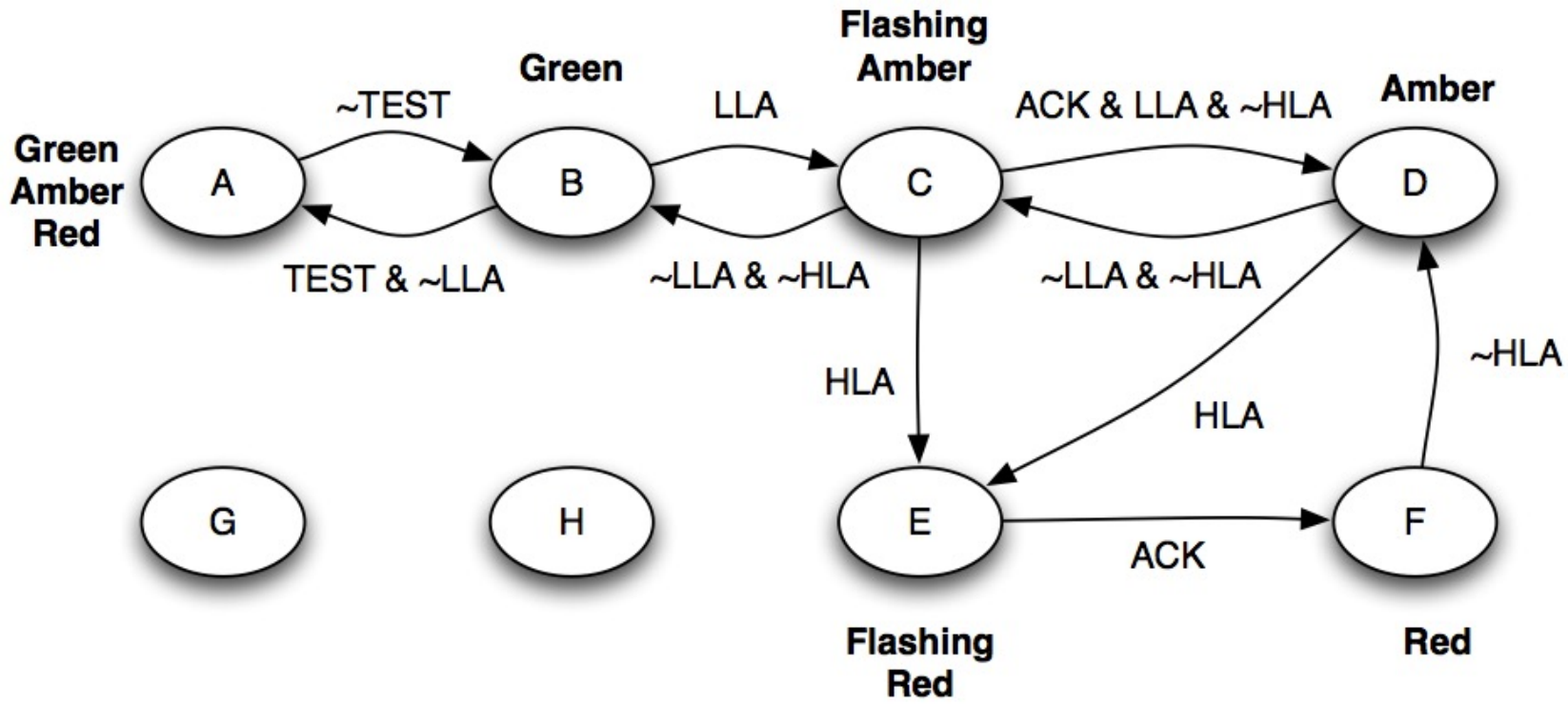
16x8 ROM mif

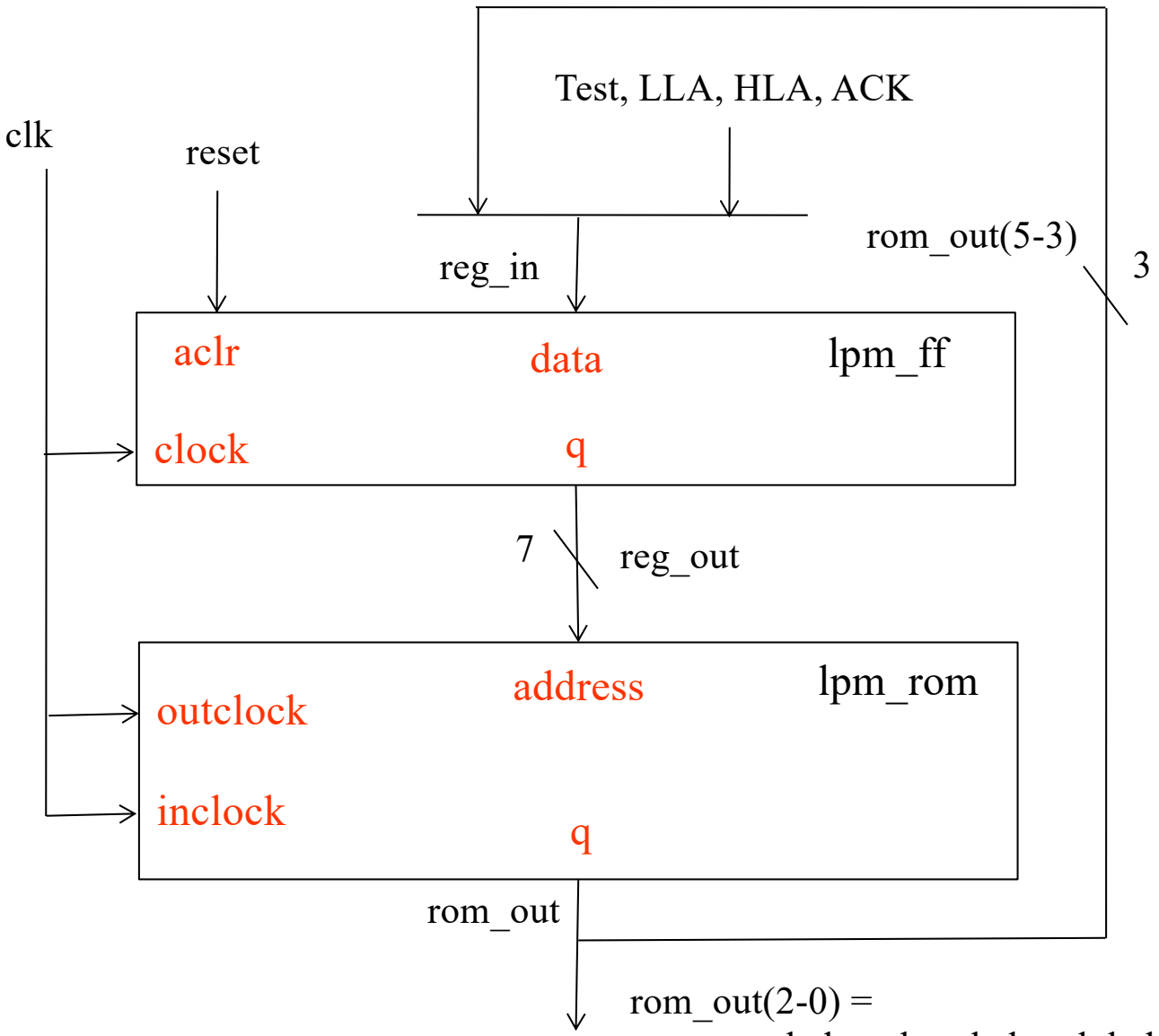
```
-- ROM data file
--
depth = 16; -- memory depth and width are in decimal
width = 8;

address_radix = hex;  -- can be bin, oct, dec, hex,
data_radix = hex;     -- default is hex

content begin
  0: 12 34 AB;   -- M[0]=12, M[1]=34, M[2]=AB
  [3..B]: 2;     -- M[3]=M[4]= … = M[B] = 2
  D: FF;         -- M[D]=FF;
  C: 13;         -- M[C]=13;
  [E..F]: 0A;    -- M[E]=M[F]=A
end;
```



Addr	+0	+1	+2	+3	+4	+5	+6	+7	ASCII
00	12	34	AB	02	02	02	02	02	.4.....
08	02	02	02	02	13	FF	0A	0A	.....

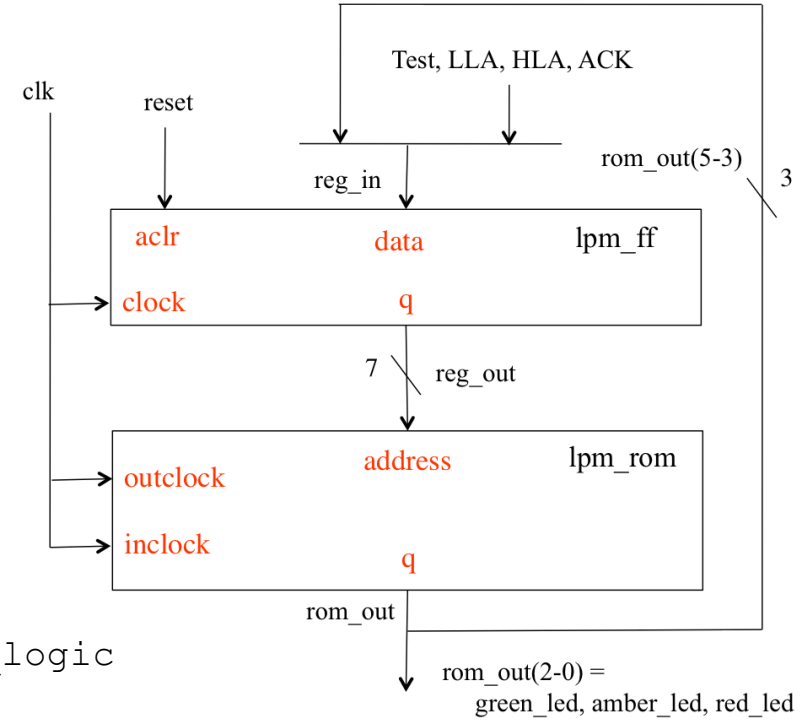
Event Driven Circuit




Test, LLA, HLA, ACK
clk
reset
rom_out(5-3)
3
reg_in
aclr
data
lpm_ff
clock
q
7
reg_out
address
lpm_rom
outclock
inclock
q
rom_out
rom_out(2-0) =
green_led, amber_led, red_led

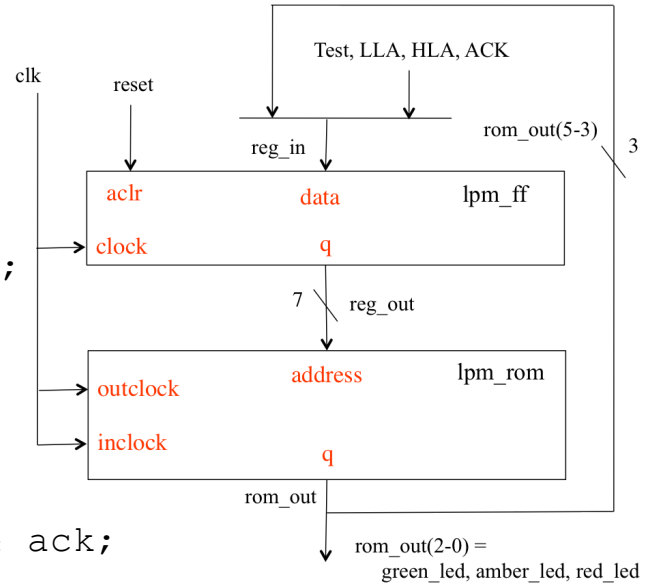
Lab1LPM Entity

```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity lab1lpm is
port (
    clk: in std_logic;
    reset: in std_logic;
    test, lla, hla, ack: in std_logic;
    green_led, amber_led, red_led: out std_logic
);
end lab1lpm;
```

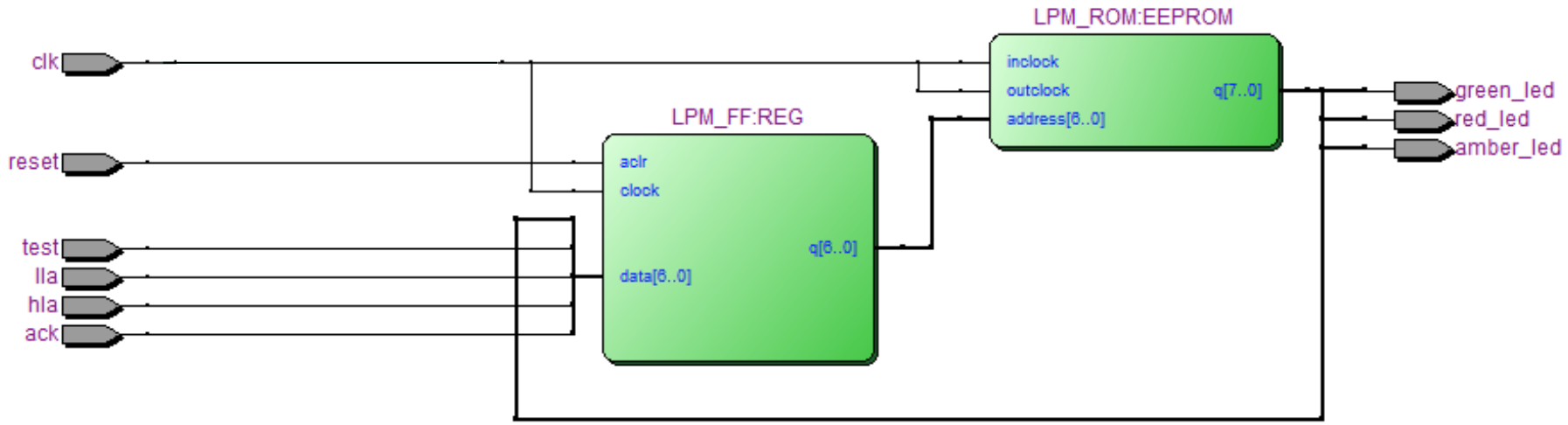


Lab1LPM Architecture

```
architecture structural of lab1lpm is
 signal rom_out: std_logic_vector(5 downto 0);
 signal reg_in, reg_out: std_logic_vector(6 downto 0);
begin
    green_led <= rom_out(2);
    amber_led <= rom_out(1);
    red_led <= rom_out(0);
    reg_in <= rom_out(5 downto 3) & test & lla & hla & ack;
    Latch: lpm_ff
          generic map(lpm_width=>7)
          port map(data=>reg_in, q=>reg_out, clock=>clk, aclr=>reset);
    ROM: lpm_rom
          generic map(lpm_widthad=>7, lpm_width=>6, lpm_file=>"lab1lpm.mif")
          port map(address=>reg_out, q=>rom_out, inclock=>clk, outclock=>clk);
end structural;
```



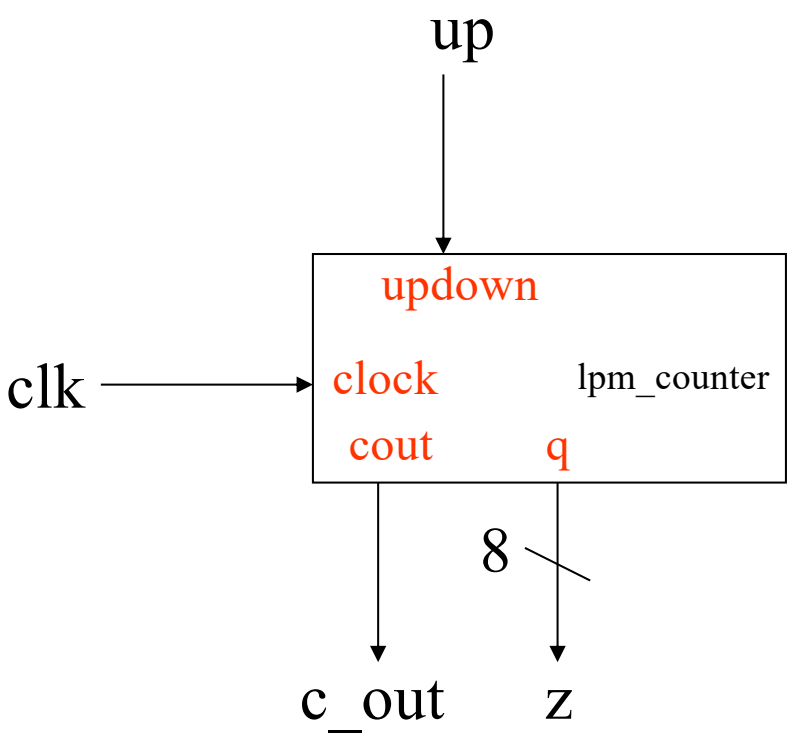
RTL View - Lab1LPM



lpm_counter

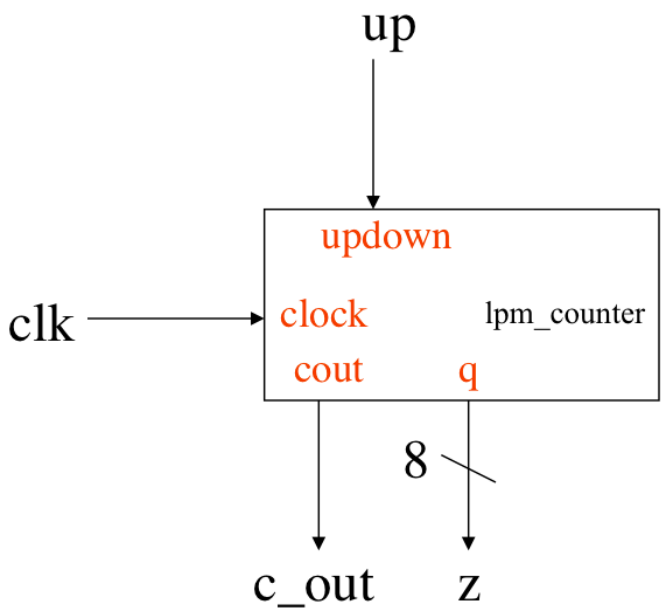
- Generics
  - lpm_width = number of bits
- Ports
  - clock = clock
  - q[] = counter value
  - updown = up/down (1/0)
  - cnt_en = counter enable
  - cout = carry_out
  - data[] = input
  - sload = synchronous load

8-bit Counter



8-bit Counter

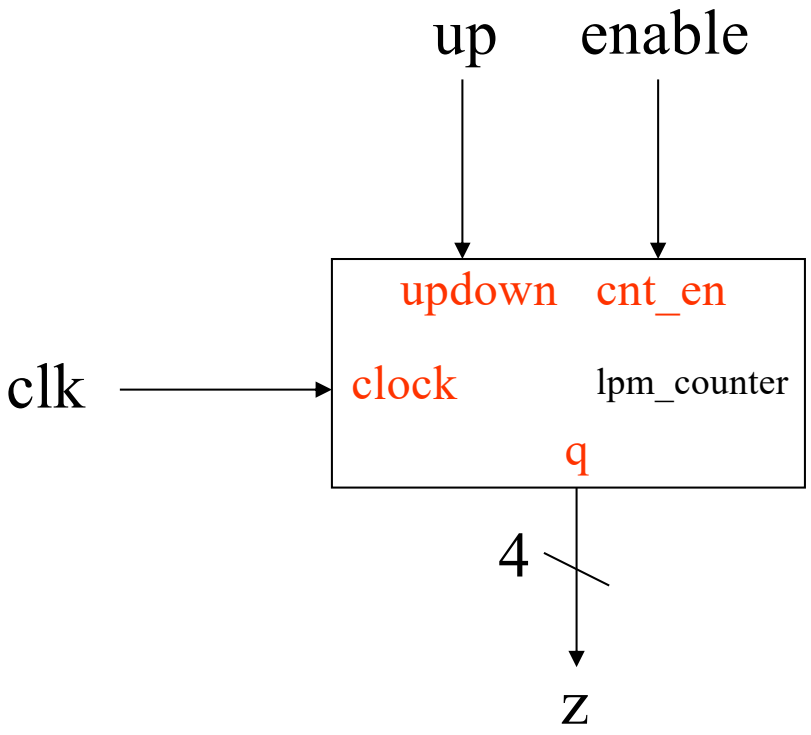
```
signal z: std_logic_vector(7 downto 0);
signal clk, c_out, up: std_logic;

up <= '1';
Counter: lpm_counter
    generic map(lpm_width=>8)
    port map(q=>z, clock=>clk, cout=>c_out, updown=>up);
```



Up/Down Signal

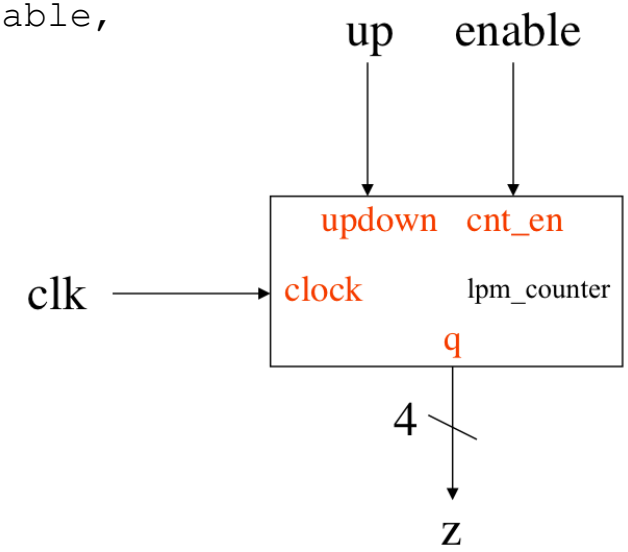
- Separate input for up and down
- Count up if up = 1 and down = 0
- Count down if up = 0 and down = 1
- Use enable to control counter
  - updown = up
  - cnt_en = (up and not down) or (not up and down)

4-bit Up/Down Counter



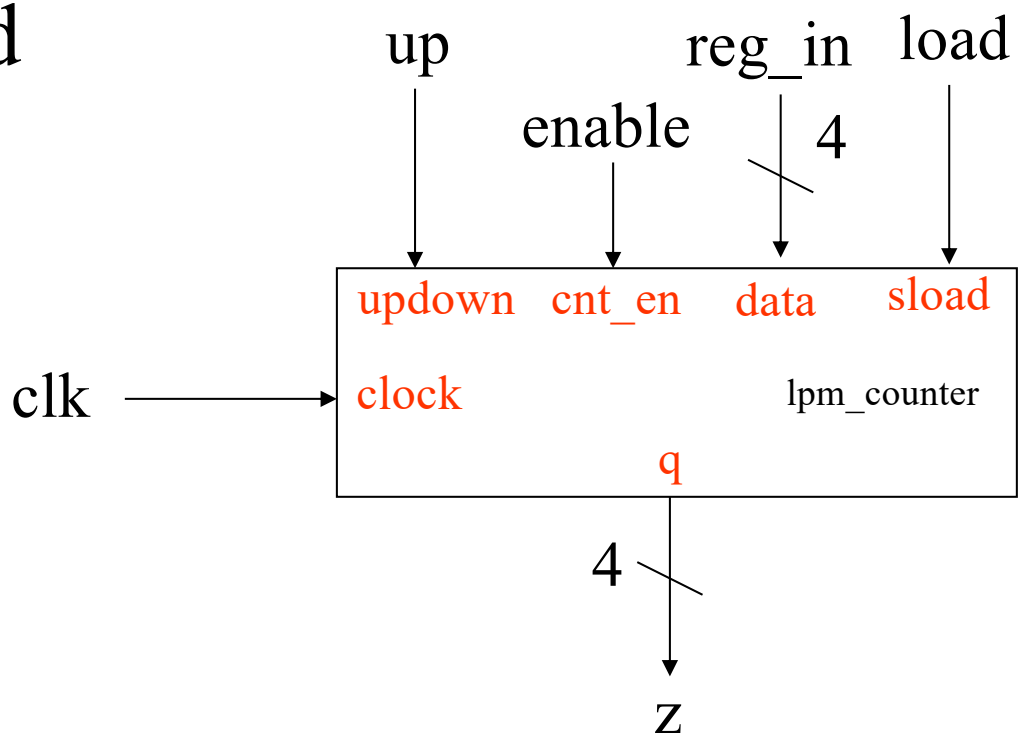
4-bit Up/Down Counter

```
constant size: integer := 4;
signal enable, up, reset, clock: std_logic;
signal z: std_logic_vector(3 downto 0);

begin
  enable <= (up and not down) or (down and not up);
  up_down <= up;
  Counter: lpm_counter
           generic map (lpm_width=>size)
           port map (clock=>clk, q=>z, cnt_en=>enable,
                     updown=>up);
```



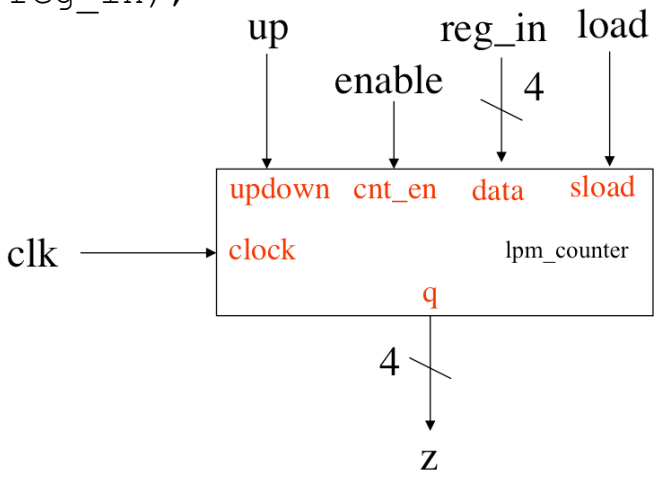
4-bit Counter/Register

- Use as both register and counter
- Synchronous load



4-bit Counter/Register

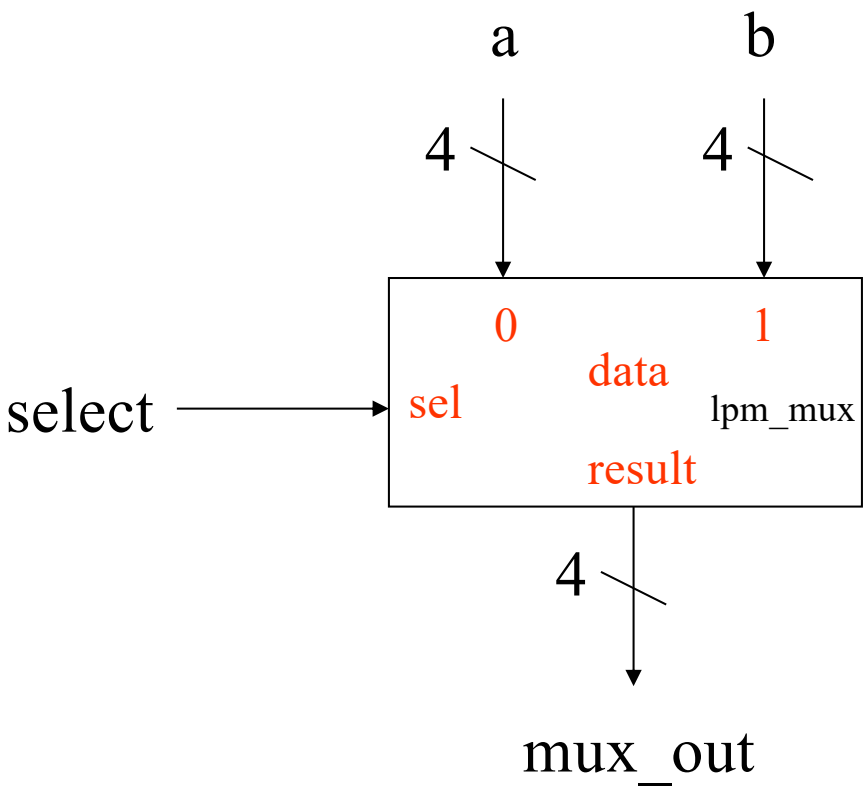
```
constant size: integer := 4;
signal enable, up, reset, clock: std_logic;
signal reg_in, z: std_logic_vector(3 downto 0);

begin
  enable <= (up and not down) or (down and not up);
  up_down <= up;
  counter: lpm_counter
           generic map (lpm_width=>size)
           port map (clock=>clk, q=>z, cnt_en=>enable, updown=>up,
                     sload=>load, data => reg_in);
```



lpm_mux

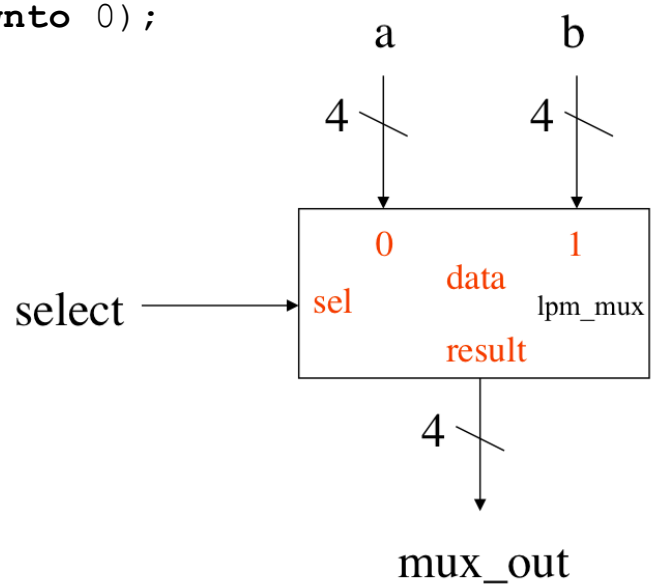
- Generics
  - lpm_width = data width
  - lpm_size = number of input data
  - lpm_widths = selection width
- Ports
  - sel[] = selection input
  - data[][] = data input
  - result[] = output

2x1 MUX, 4-bit Wide

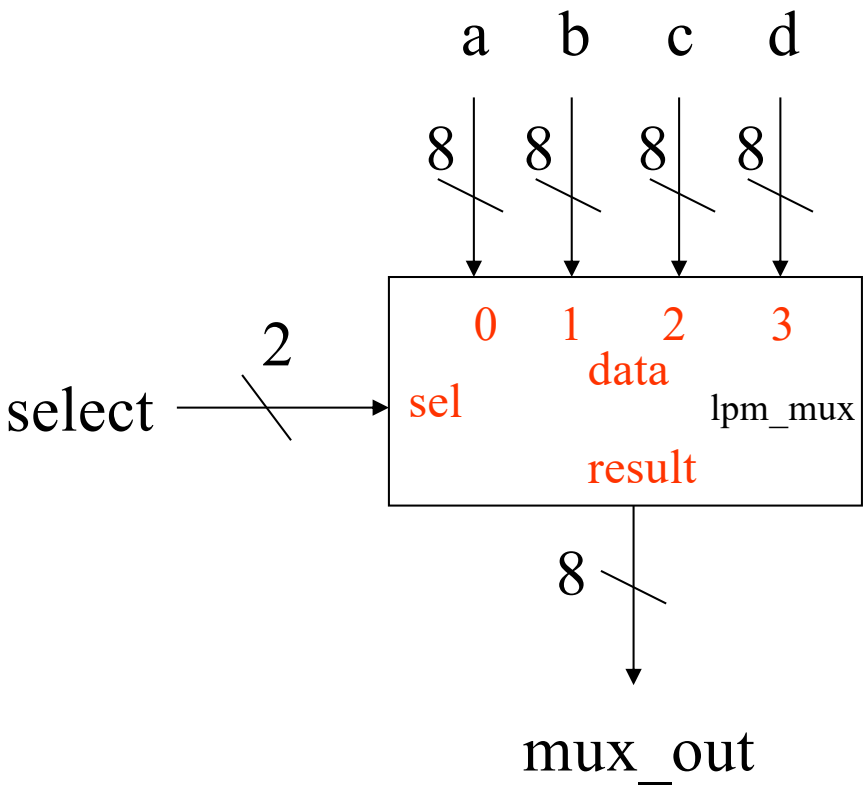


2x1 MUX, 4-bit Wide

```
signal mux_out: std_logic_vector(3 downto 0);
signal mux_data: std_logic_2D(1 downto 0, 3 downto 0);
signal mux_select: std_logic_vector(0 to 0);

begin
  MuxG: for i in 0 to 3 generate
    mux_data(0, i) <= a(i);
    mux_data(1, i) <= b(i);
  end generate;

  mux_select(0) <= select;

  Mux1: lpm_mux
    generic map(lpm_width=>4, lpm_size=>2, lpm_widths=>1)
    port map(result=>mux_out, data=>mux_data, sel=>mux_select);
```

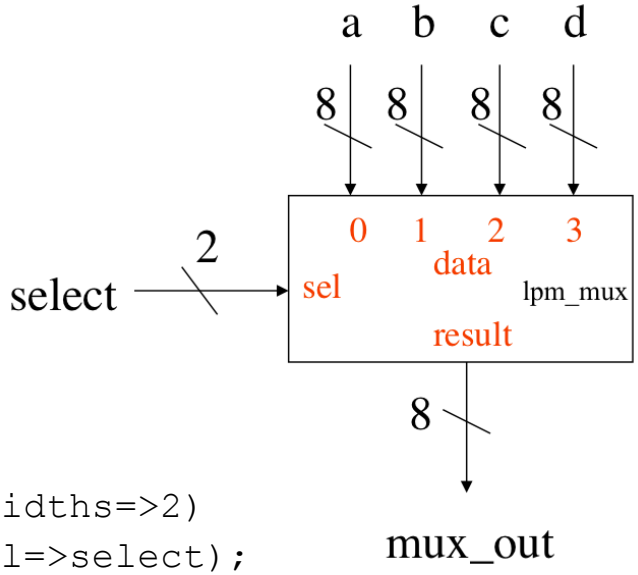


4x1 MUX, 8-bit Wide



4x1 MUX, 8-bit Wide

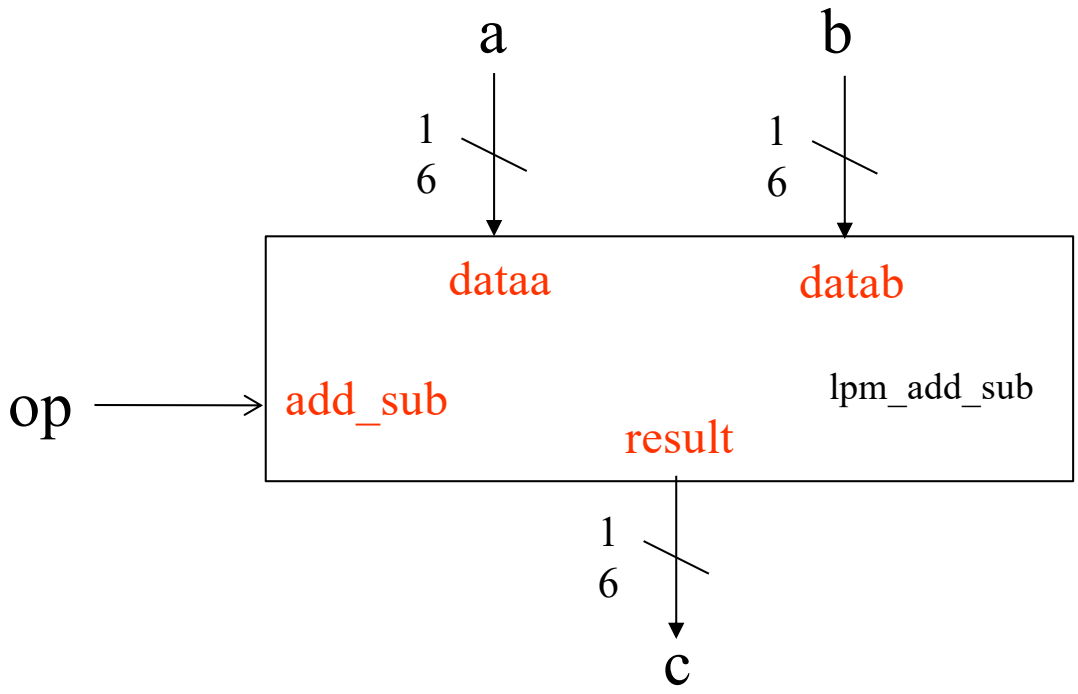
```
signal mux_out: std_logic_vector(7 downto 0);
signal mux_data: std_logic_2D(3 downto 0, 7 downto 0);

begin
  MuxG2: for i in 0 to 7 generate
    mux_data(0, i) <= a(i);
    mux_data(1, i) <= b(i);
    mux_data(2, i) <= c(i);
    mux_data(3, i) <= d(i);
  end generate;

  Mux2: lpm_mux
    generic map(lpm_width=>8, lpm_size=>4, lpm_widths=>2)
    port map(result=>mux_out, data=>mux_data, sel=>select);
```



lpm_add_sub

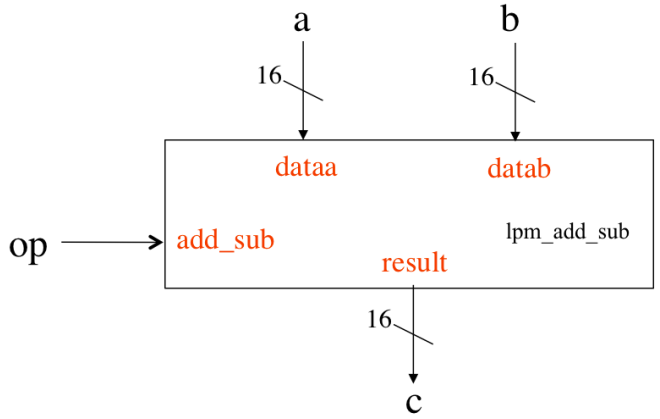
- Generic
  - lpm_width = size of data
- Ports
  - dataa[] = input
  - datab[] = input
  - result[] = output
  - add_sub = 1/0 – add/subtract

ALU



16-bit add/subtract

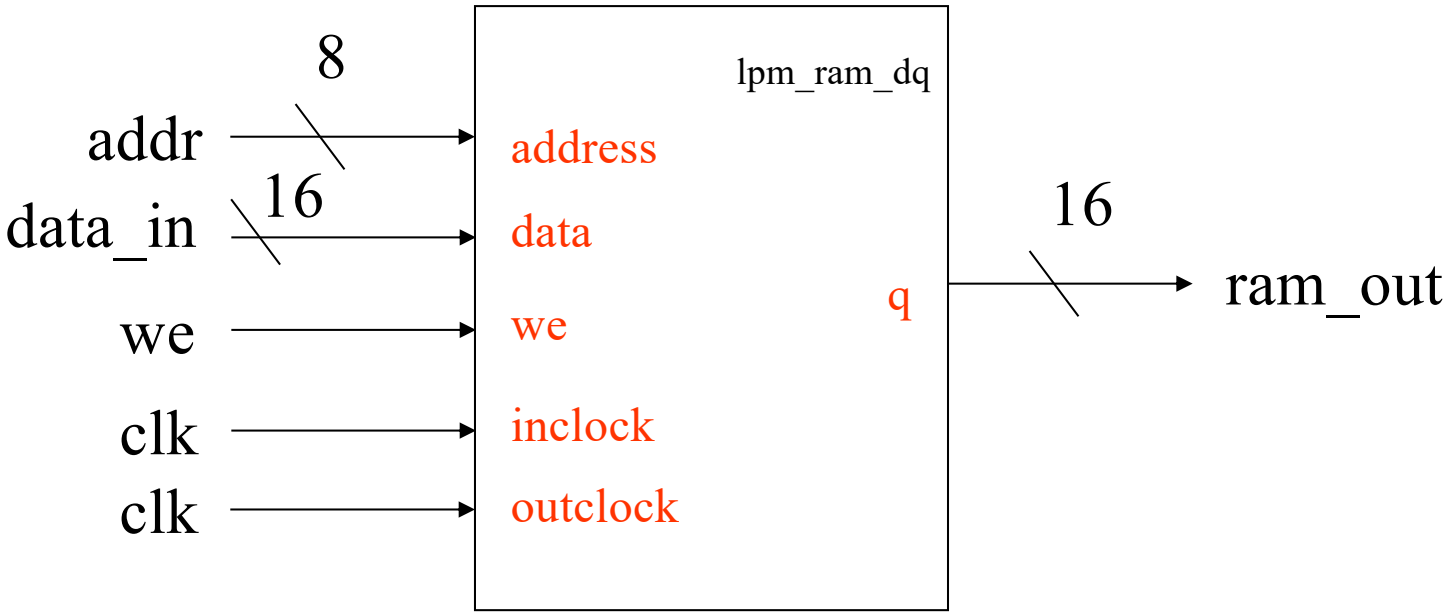
```
signal a, b, c: std_logic_vector(15 downto 0);
signal op: std_logic;

ALU: lpm_add_sub
   generic map (lpm_width=>16)
   port map (dataa=>a, datab=>b, result=>c, add_sub=>op);
```



lpm_ram_dq

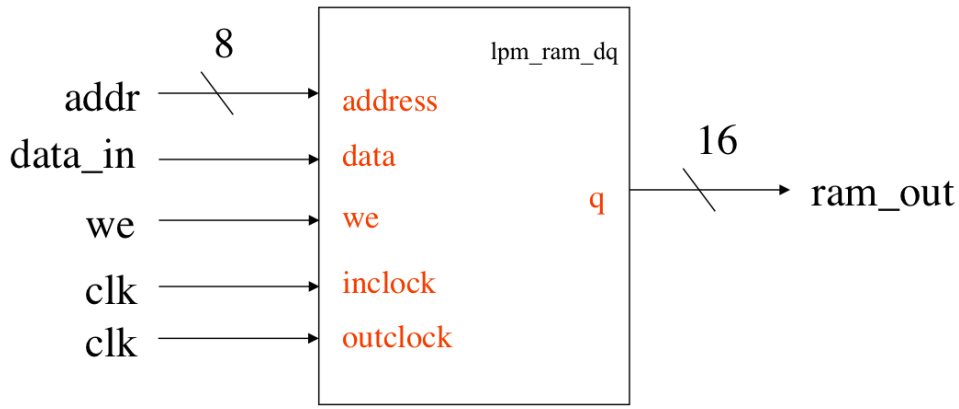
- Generics
  - lpm_widthad = width of address
  - lpm_width = width of data
  - lpm_file = data file
- Ports
  - address[] = address
  - q[] = output
  - inclock = synchronous address
  - outclock = synchronous output
  - we = write enable
  - data[] = data input

256x16 RAM


8
addr
16
data_in
we
clk
clk
lpm_ram_dq
address
data
we
inclock
outclock
q
16
ram_out

256x16 RAM

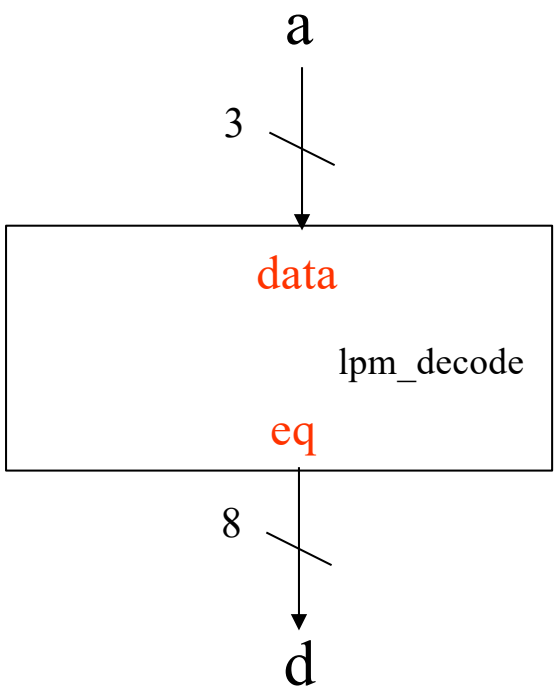
```
signal addr: std_logic_vector(7 downto 0);
signal data_in, ram_out: std_logic_vector(15 downto 0);
signal inclock, outclock, we: std_logic;

RAM: lpm_ram_dq
  generic map(lpm_widthad=>8, lpm_width=>16, lpm_file=>"r.mif")
  port map(address=>addr, q=>ram_out, inclock=>clk,
                data=>data_in, outclock=>clk, we=>we);
```



lpm_decode

- Generics
  - lpm_width = data width
  - lpm_decodes = number of decoder outputs
    (lpm_decodes <= 2^lpm_width)
- Ports
  - data[] = inputs (size = lpm_width)
  - eq[] = outputs (size = lpm_decodes)

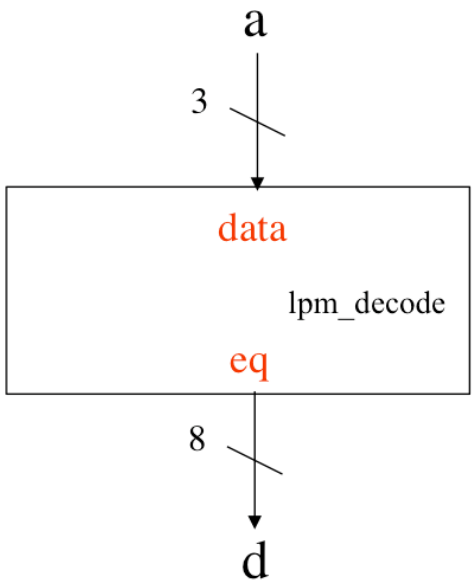
3 to 8 Decoder



a	d(7)	d(6)	d(5)	d(4)	d(3)	d(2)	d(1)	d(0)
000	0	0	0	0	0	0	0	1
001	0	0	0	0	0	0	1	0
010	0	0	0	0	0	1	0	0
011	0	0	0	0	1	0	0	0
100	0	0	0	1	0	0	0	0
101	0	0	1	0	0	0	0	0
110	0	1	0	0	0	0	0	0
111	1	0	0	0	0	0	0	0

3 to 8 Decoder

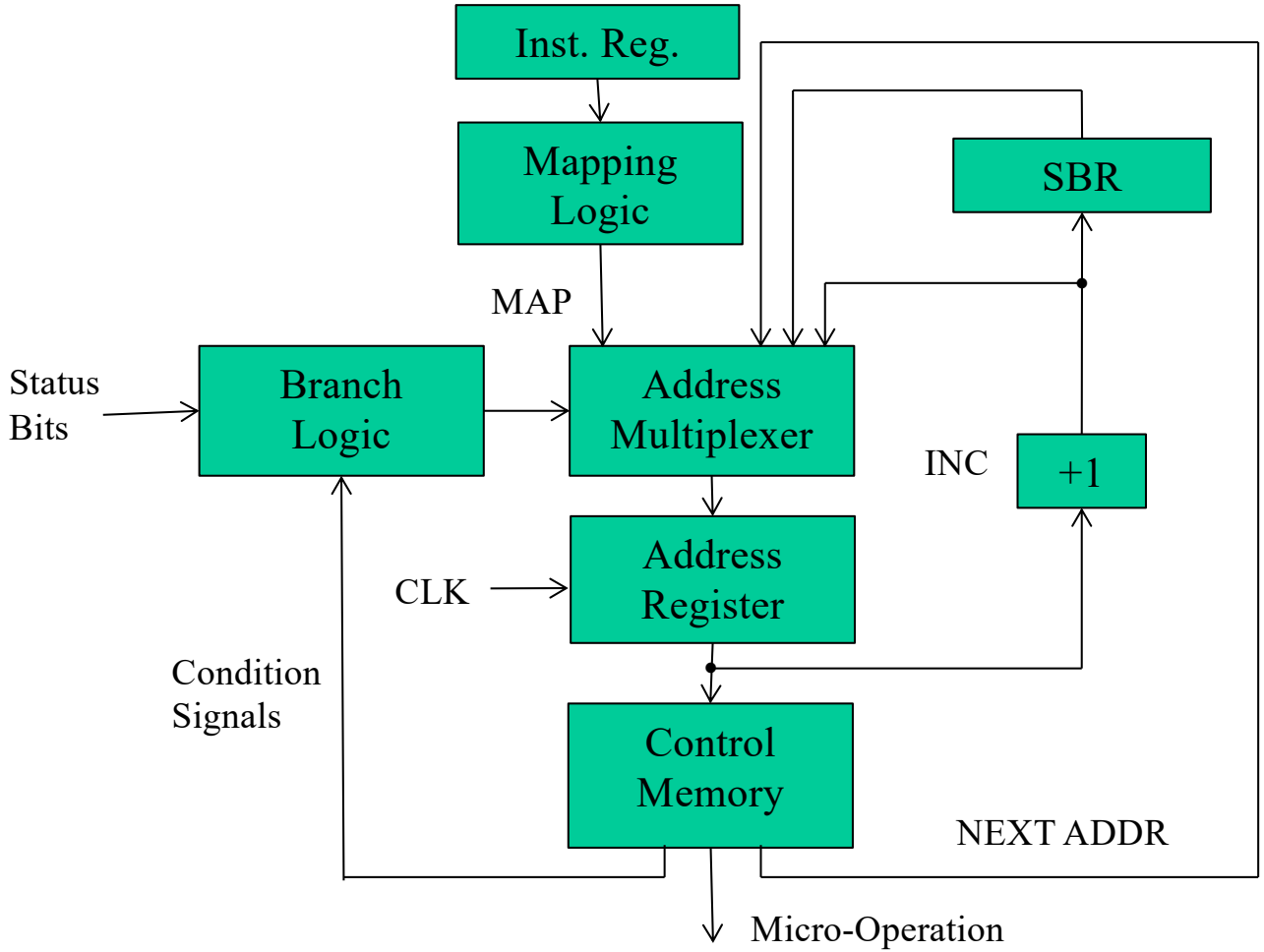
```
signal a: std_logic_vector(2 downto 0);
signal d: std_logic_vector(7 downto 0);

decoder: lpm_decode
   generic map (lpm_width=>3, lpm_decodes=>8)
   port map (data=>a, eq=>d);
```



Micro-sequencer

- Control unit implementation
- Instruction execution is divided into a sequence of micro-instruction
- Each micro-instruction contains
  - the micro-operations (control signals)
  - location of the next micro-instruction
  - others

Generic Micro-sequencer



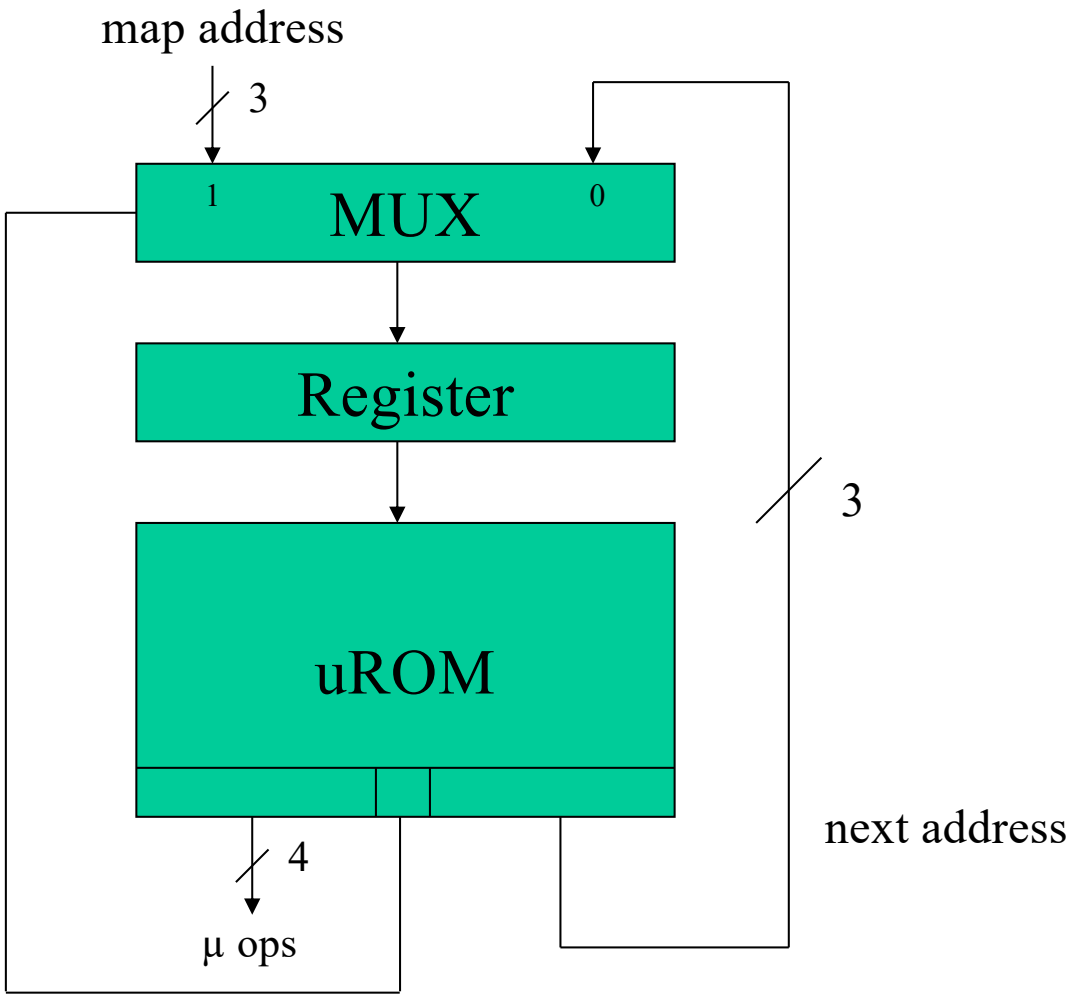
Micro-instruction Format

- 8-bit

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Z_1	Z_2	Z_3	Z_4	MUX	A_2	A_1	A_0

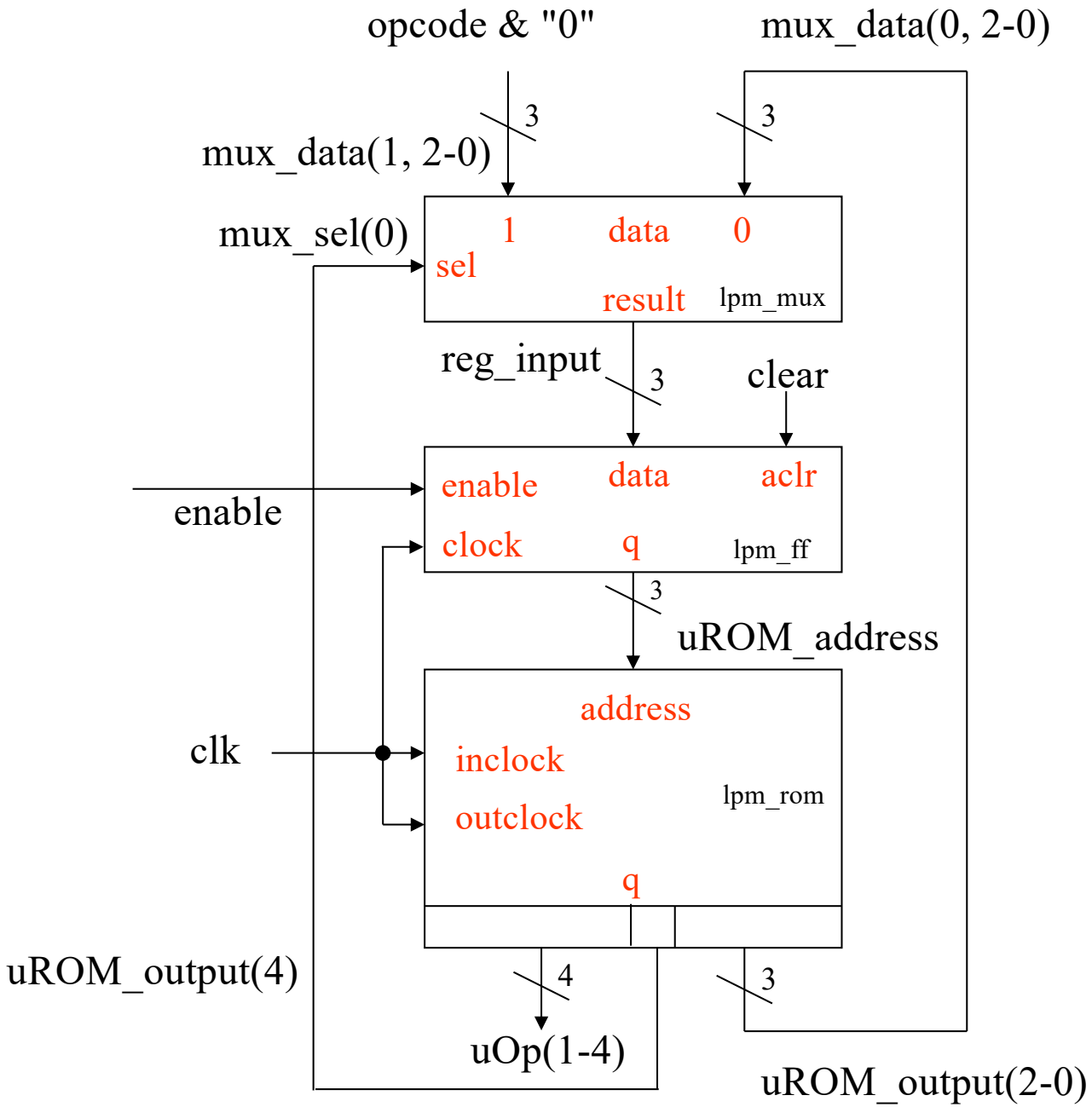
- Address MUX control (MUX)
  - 0: From micro ROM
  - 1: From external
- A_2-A_0: Next address
- Z_1, Z_2, Z_3, Z_4: micro-operations

Simple Design



Simple design

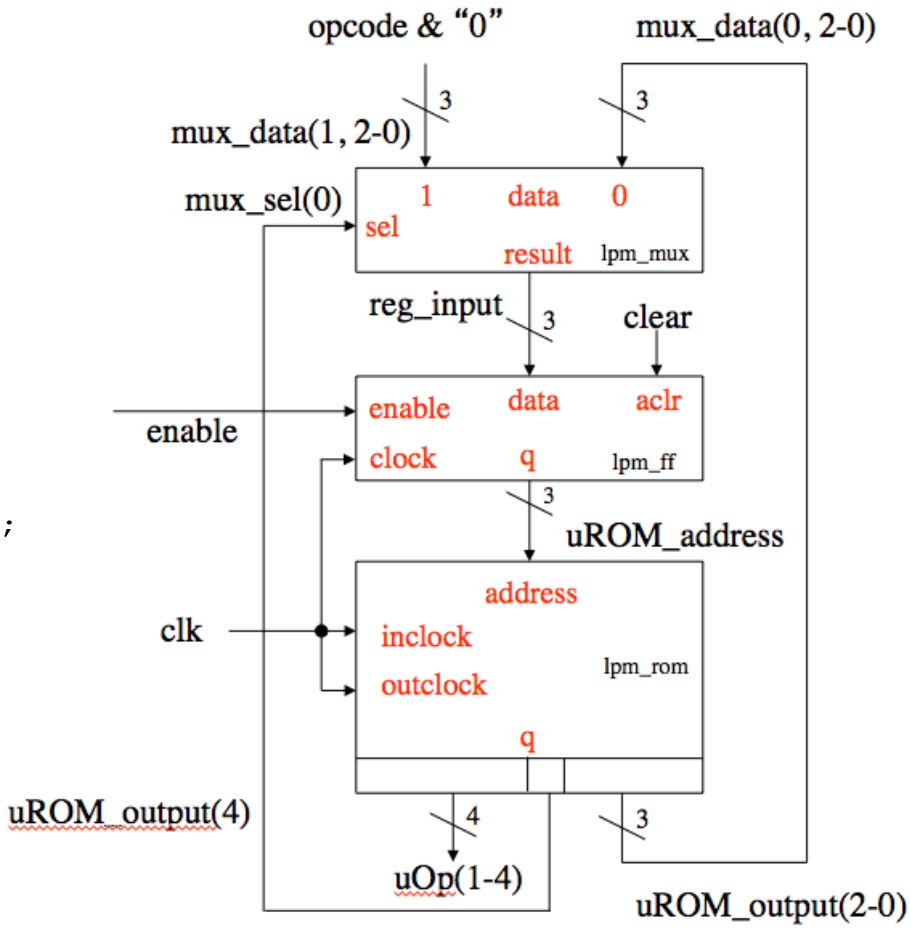
- Next micro-address selection – 2x1 MUX
- Buffer register – 3-bit register
- Micro-ROM – 8x8 ROM

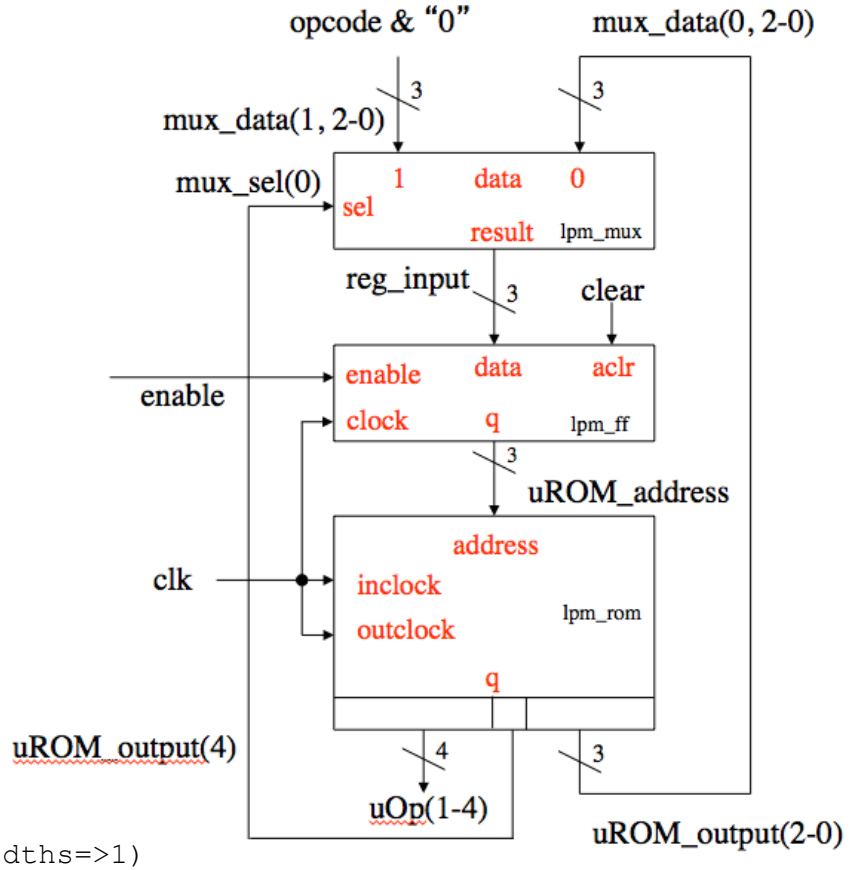

opcode & "0"
mux_data(0, 2-0)
3
3
mux_data(1, 2-0)
mux_sel(0)
1
data
0
sel
result
lpm_mux
reg_input
3
clear
enable
data
aclr
enable
clock
q
lpm_ff
3
uROM_address
address
clk
inclock
lpm_rom
outclock
q
uROM_output(4)
4
3
uOp(1-4)
uROM_output(2-0)

Mircosequencer

- Implement with VHDL
- lpm_mux
- lpm_ff
- lpm_rom

```
-- ECE 495 - simple micro sequencer
-- Dr. Hou
--

library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity uSeq is
  port (
    opcode: in std_logic_vector(1 downto 0);
    uop : out std_logic_vector(1 to 4);
    enable, clear: in std_logic;
    clk: in std_logic
  );
end uSeq;
```






```
architecture structural of uSeq is

signal uROM_address: std_logic_vector (2 downto 0);
signal uROM_output: std_logic_vector (7 downto 0);
signal mux_data: std_logic_2D(1 downto 0, 2 downto 0);
signal mux_sel: std_logic_vector(0 to 0);
signal reg_input, temp: std_logic_vector(2 downto 0);

begin
  temp <= opcode & "0";
  for_label: for i in 0 to 2 generate
       mux_data(0, i) <= uROM_output(i);
       mux_data(1, i) <= temp(i);
  end generate;
  mux_sel(0) <= uROM_output(4);
  A1: lpm_mux
        generic map (lpm_width=>3, lpm_size=>2, lpm_widths=>1)
        port map (result=>reg_input, data=>mux_data, sel=>mux_sel);
  A2: lpm_ff
        generic map (lpm_width=>3)
        port map (enable=>enable, data=>reg_input, q=>uROM_address, aclr=>clear, clock=>clk);
  A3: lpm_rom
        generic map (lpm_widthad=>3, lpm_width=>8, lpm_file =>"urom.mif")
        port map (address=>uROM_address, q=>uROM_output, inclock=>clk, outclock=>clk);
  uop <= ROM_output(7 downto 4);
end structural;
```

Nano CPU

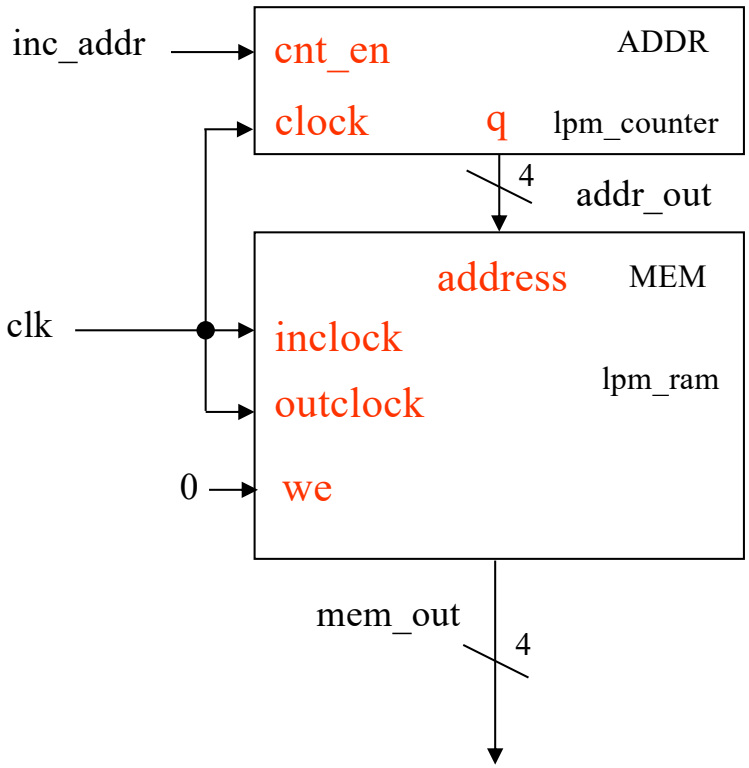
- 4-bit Address register – ADDR
- 16x4 RAM
- Two 4-bit registers: REG_A, REG_B
- Two instructions: I_1 and I_2 (two bit opcode)

Register Transfer Operations

- 4-bit Address register – ADDR, can be incremented
- 16x4 RAM – MEM[ADDR]
- Two 4-bit registers: REG_A, REG_B
- Micro-operations
  - ADDR ← ADDR + 1
  - REG_A ← MEM[ADDR]
  - REG_B ← MEM[ADDR]
  - REG_B ← REG_A

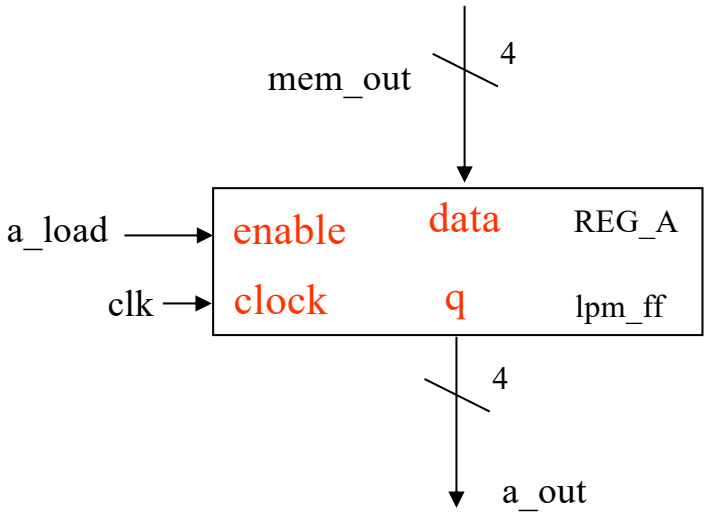
Datapath and Control Signals

- ADDR ⟵ ADDR + 1
- MEM[ADDR]
  - Use counter to implement ADDR
  - Control signal – inc_addr
  - Output of counter connected to address bus of MEM



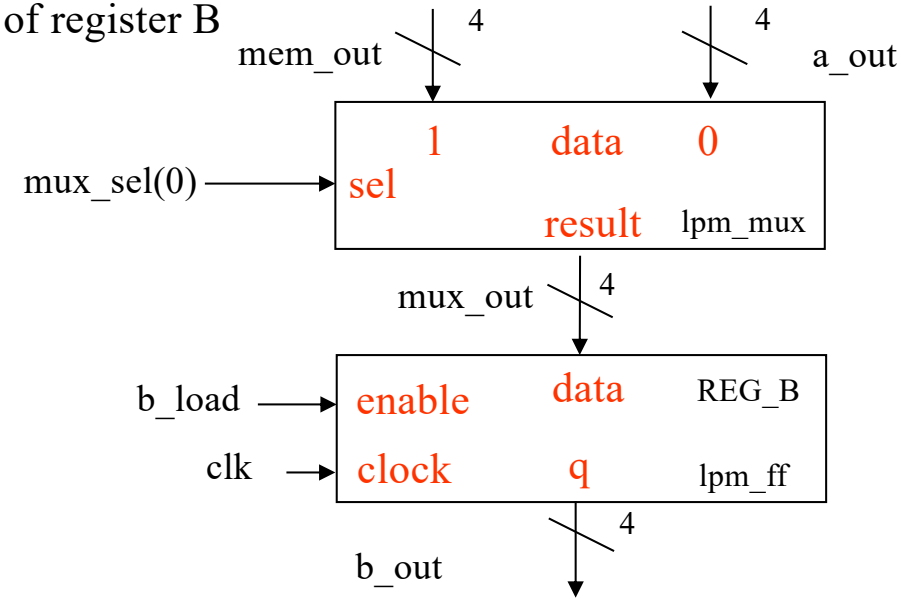
Datapath and Control Signals

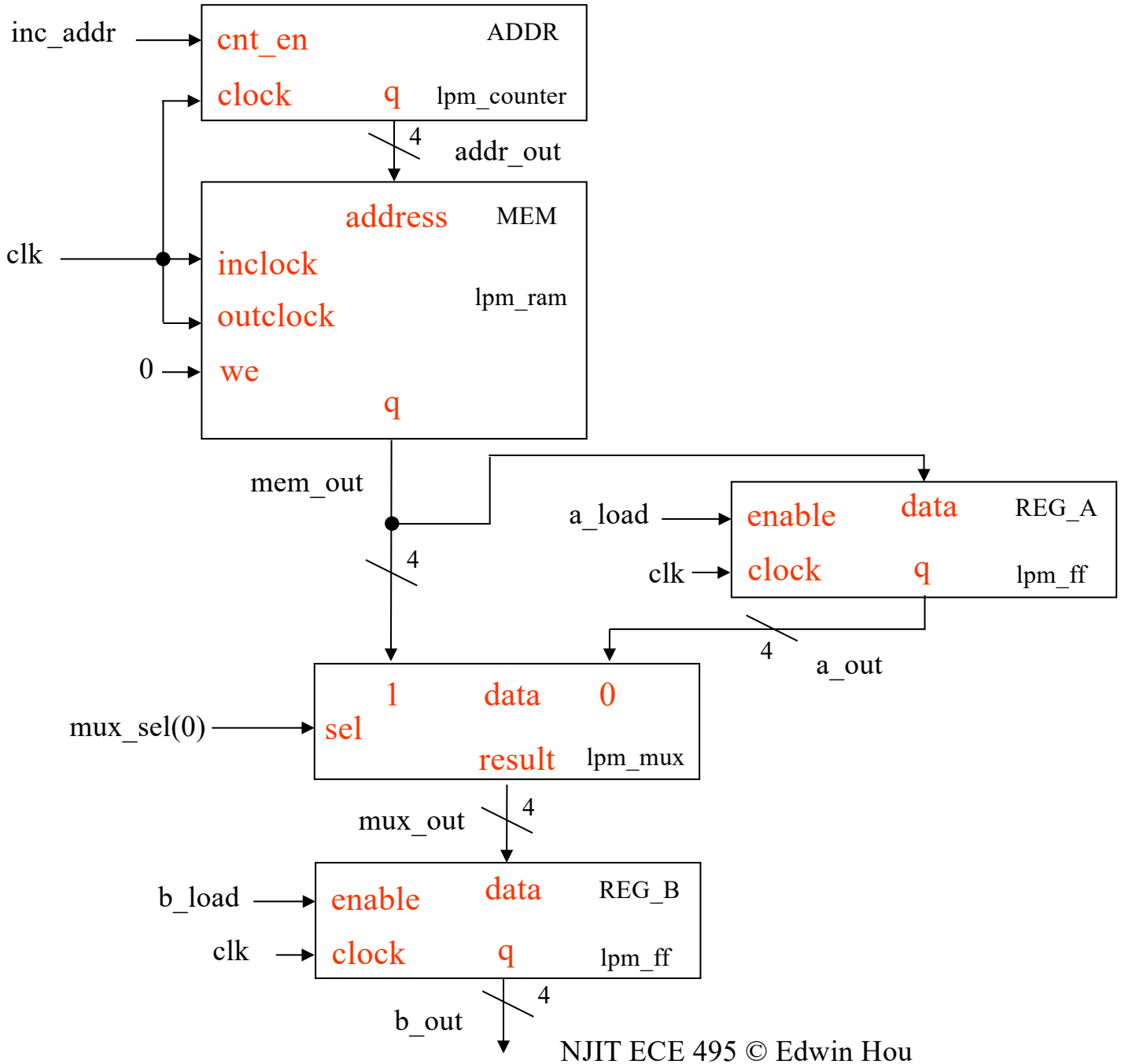
- REG_A ⟵ MEM[ADDR]
  - Control signal – a_load
  - Output of MEM connected to input of register A



Datapath and Control Signals

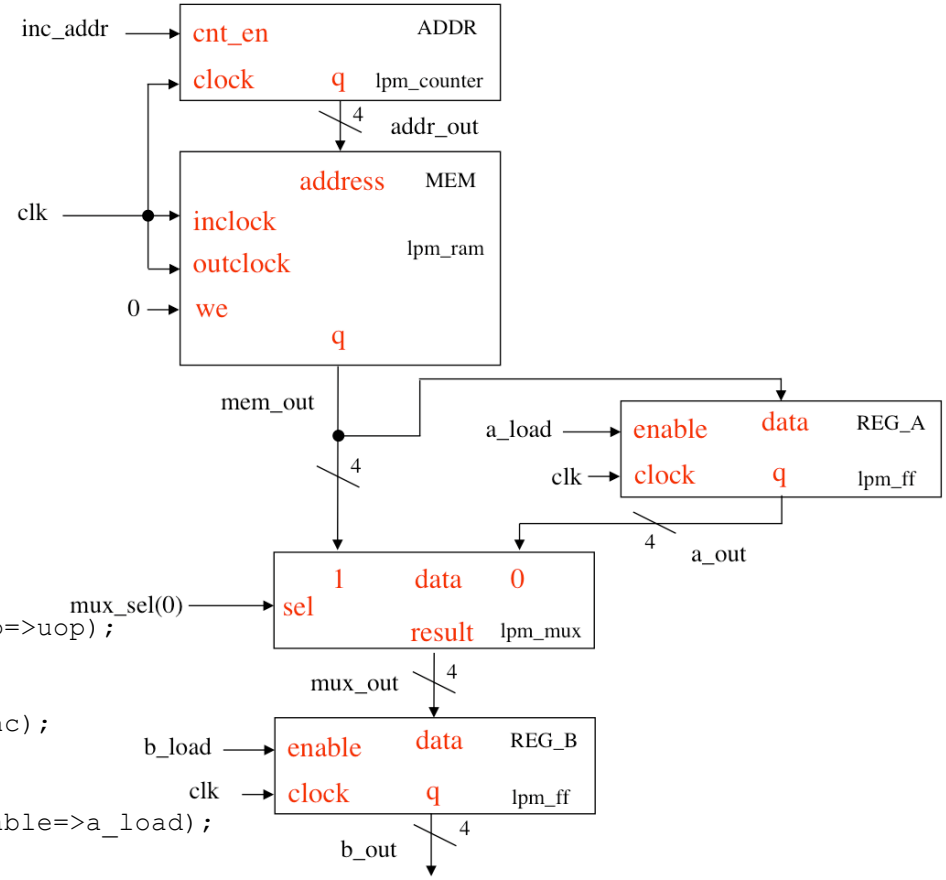
- REG_B ⟵ MEM[ADDR]
- REG_B ⟵ REG_A
  - Need a multiplexer to select between the two sources
  - Control signals – b_load, mux_select
  - Output of mux connected to input of register B





```
library ieee;
use ieee.std_logic_1164.all;
library lpm;
use lpm.lpm_components.all;

entity nano_cpu is
port (clk: in std_logic;
      opcode: in std_logic_vector(1 downto 0);
      seg_a, seg_b, seg_addr: out std_logic_vector(0 to 6);
      seg_useq: out std_logic_vector(1 to 4);
      pause, clear: in std_logic);
end mini_cpu;

architecture structural of mini_cpu is

component mini_uSeq
port (opcode: in std_logic_vector(1 downto 0);
      uop: out std_logic_vector(1 to 4);
      enable, clear: in std_logic;
      clock: in std_logic);
end component;

component hex_7Seg
port (i: in std_logic_vector(3 downto 0);
      segments: out std_logic_vector(0 to 6));
end component;

signal mux_data: std_logic_2d(1 downto 0, 3 downto 0);
signal mux_out: std_logic_vector(3 downto 0);
signal mux_select: std_logic_vector(0 to 0);
signal addr_inc, b_load, a_load: std_logic;
signal addr_out, a_out, b_out, mem_out: std_logic_vector(3 downto 0);
signal a_in: std_logic_vector(3 downto 0);
signal uPC_enable, cout: std_logic;
signal uop: std_logic_vector(1 to 4);
```

```
begin
    Delay: lpm_counter
            generic map(lpm_width=>26)
            port map(clock=>clk, cout=>cout);
    upc_enable <= cout and not pause;
    addr_inc <= uop(1) and upc_enable;
    a_load <= uop(2) and upc_enable;
    b_load <= uop(3) and upc_enable;
    mux_select(0) <= uop(4) and upc_enable;
    seg_useq <= uop;

    MUX_C: for i in 0 to 3 generate
            mux_data(0, i) <= a_out(i);
            mux_data(1, i) <= mem_out(i);
    end generate;

    useq: mini_uSeq
            port map(clock=>clk, opcode=>opcode,
                        enable=>upc_enable, clear=>clear, uop=>uop);
    Addr: lpm_counter
            generic map(lpm_width=>4)
            port map(q=>addr_out, clock=>clk, cnt_en=>addr_inc);
    RegA: lpm_ff
            generic map(lpm_width=>4)
            port map(data=>mem_out, q=>a_out, clock=>clk, enable=>a_load);
    RegB: lpm_ff
            generic map(lpm_width=>4)
            port map(data=>mux_out, q=>b_out, clock=>clk, enable=>b_load);
    MUX: lpm_mux
            generic map(lpm_width=>4, lpm_size=>2, lpm_widths=>1)
            port map(result=>mux_out, data=>mux_data, sel=>mux_select);
    RAM: lpm_ram_dq
            generic map(lpm_widthad=>4, lpm_width=>4, lpm_file=>"Data.mif")
            port map(data=>"0000", address=>addr_out, q=>mem_out,
                        inclock=>clk, outclock=>clk, we=>'0');

    displayAddr: hex_7Seg port map(i=>addr_out, segments=>seg_addr);
    displayA: hex_7Seg port map(i=>a_out, segments=>seg_a);
    displayB: hex_7Seg port map(i=>b_out, segments=>seg_b);
end structural;
```



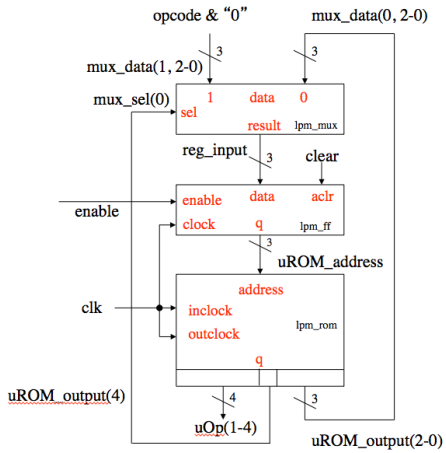
Control Signals

Micro-operation	inc_addr	a_load	b_load	mux_select
ADDR ⟵ ADDR + 1	1	0	0	x
REG_A ⟵ MEM[ADDR]	0	1	0	x
REG_B ⟵ MEM[ADDR]	0	0	1	1
REG_B ⟵ REG_A	0	0	1	0

Macro Operations

- Instruction cycle: Instruction fetch, Instruction execution
- Instruction fetch: 2 cycles
  - addr ⟵ addr +1
  - NOP, map
- Instruction execution: I_1, 2 cycles, opcode – 01.
  - REG_A ⟵ MEM[addr]
  - addr ⟵ addr +1, goto fetch
- Instruction execution: I_2, 4 cycles, opcode – 10
  - REG_B ⟵ REG_A
  - REG_A ⟵ MEM[addr]
  - addr ⟵ addr +1,
  - REG_B ⟵ MEM[addr], goto fetch

Micro ROM Content



I_1: 0-1-2-3 I_2: 0-1-4-5-6-7

I_1 opcode & "0" = 010 = 2
I_2 opcode & "0" = 100 = 4

	inc_ addr	a_load	b_load	mux_ select	Map?	Next address			
Address	D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	Operation
0	1	0	0	0	0	0	0	1	addr ⟵ addr+1
1	0	0	0	0	1	0	0	0	NOP, map
2	0	1	0	0	0	0	1	1	REG_A ⟵ MEM[addr]
3	1	0	0	0	0	0	0	0	addr ⟵ addr+1, goto 0
4	0	0	1	0	0	1	0	1	REG_B ⟵ REG_A
5	0	1	0	0	0	1	1	0	REG_A ⟵ MEM[addr]
6	1	0	0	0	0	1	1	1	addr ⟵ addr+1
7	0	0	1	1	0	0	0	0	REG_B ⟵ MEM[addr] goto 0

RAM Content

Address	Content	Address	Content
0	7	8	E
1	6	9	D
2	5	A	C
3	4	B	B
4	3	C	A
5	2	D	9
6	1	E	8
7	F	F	0

I_1 Results

addr	REG_A
0	0

addr	REG_A

addr	REG_A

RAM Content			
Address	Content	Address	Content
0	7	8	E
1	6	9	D
2	5	A	C
3	4	B	B
4	3	C	A
5	2	D	9
6	1	E	8
7	F	F	0

Instruction fetch: 2 cycles

addr ⟵ addr +1

NOP, map

Instruction execution: I_1, 2 cycles

REG_A ⟵ MEM[addr]

addr ⟵ addr +1, goto fetch

I_1 Results

addr	REG_A
0	0
1	0
1	6
2	6
3	6
3	4
4	4
5	4

addr	REG_A
5	2
6	2
7	2
7	F
8	F
9	F
9	D
A	D

addr	REG_A
B	D
B	B
C	B
D	B
D	9
E	9
F	9
F	0

RAM Content			
Address	Content	Address	Content
0	7	8	E
1	6	9	D
2	5	A	C
3	4	B	B
4	3	C	A
5	2	D	9
6	1	E	8
7	F	F	0

Instruction fetch: 2 cycles

addr ⟵ addr +1

NOP, map

Instruction execution: I_1, 2 cycles

REG_A ⟵ MEM[addr]

addr ⟵ addr +1, goto fetch

I_2 Result

addr	REG_A	REG_B
0	0	0

addr	REG_A	REG_B

addr	REG_A	REG_B

RAM Content			
Address	Content	Address	Content
0	7	8	E
1	6	9	D
2	5	A	C
3	4	B	B
4	3	C	A
5	2	D	9
6	1	E	8
7	F	F	0

Instruction fetch: 2 cycles

addr ⟵ addr +1

NOP, map

Instruction execution: I_2, 4 cycles

REG_B ⟵ REG_A

REG_A ⟵ MEM[addr]

addr ⟵ addr +1,

REG_B ⟵ MEM[addr], goto fetch

I_2 Result

addr	REG_A	REG_B
0	0	0
1	0	0
1	0	0
1	6	0
2	6	0
2	6	5
3	6	5
3	6	6
3	4	6
4	4	6
4	4	3
5	4	3
5	4	4
5	2	4

addr	REG_A	REG_B
6	2	4
6	2	1
7	2	1
7	2	2
7	F	2
8	F	2
8	F	E
9	F	E
9	F	F
9	D	F
A	D	F
A	D	C
B	D	C
B	D	D

addr	REG_A	REG_B
B	B	D
C	B	D
C	B	A
D	B	A
D	B	B
D	9	B
E	9	B
E	9	8
F	9	8
F	9	9
F	0	9
0	0	9
0	0	7

RAM Content			
Address	Content	Address	Content
0	7	8	E
1	6	9	D
2	5	A	C
3	4	B	B
4	3	C	A
5	2	D	9
6	1	E	8
7	F	F	0

Instruction fetch: 2 cycles

addr ⟵ addr +1

NOP, map

Instruction execution: I_2, 4 cycles

REG_B ⟵ REG_A

REG_A ⟵ MEM[addr]

addr ⟵ addr +1,

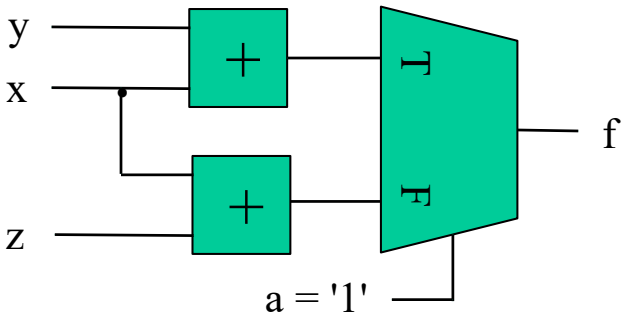
REG_B ⟵ MEM[addr], goto fetch

Resource Sharing

- When a VHDL program is synthesized, all statements and language constructs of the program will be mapped to the hardware
- Reduce the overall size of the synthesized hardware by identifying resources that can be use by different operations

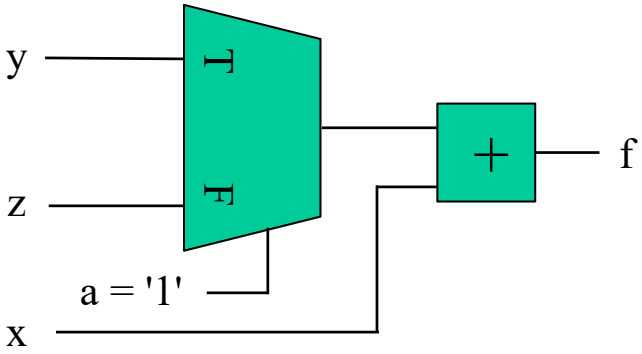
Example

```
f <= x + y when a = '1'
        else x + z;
```



- Uses two adders and one multiplexor

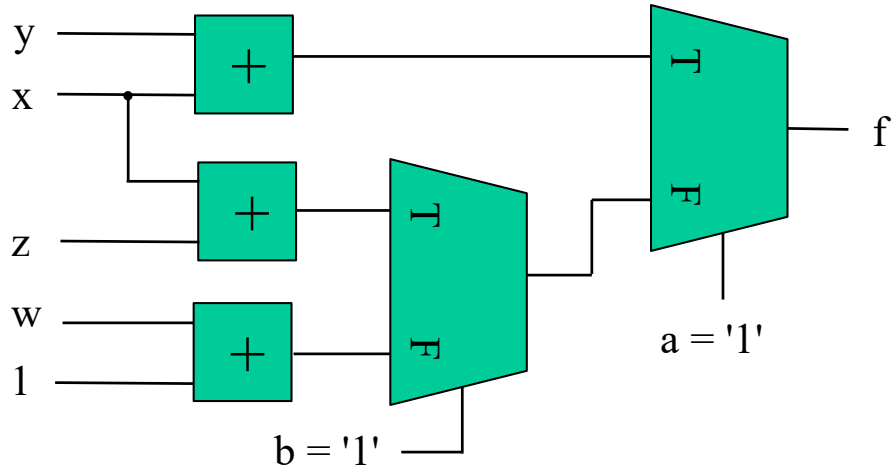
```
tmp <= y when a = '1' else z;
f <= x + tmp;
```



- Uses one adder and one multiplexor

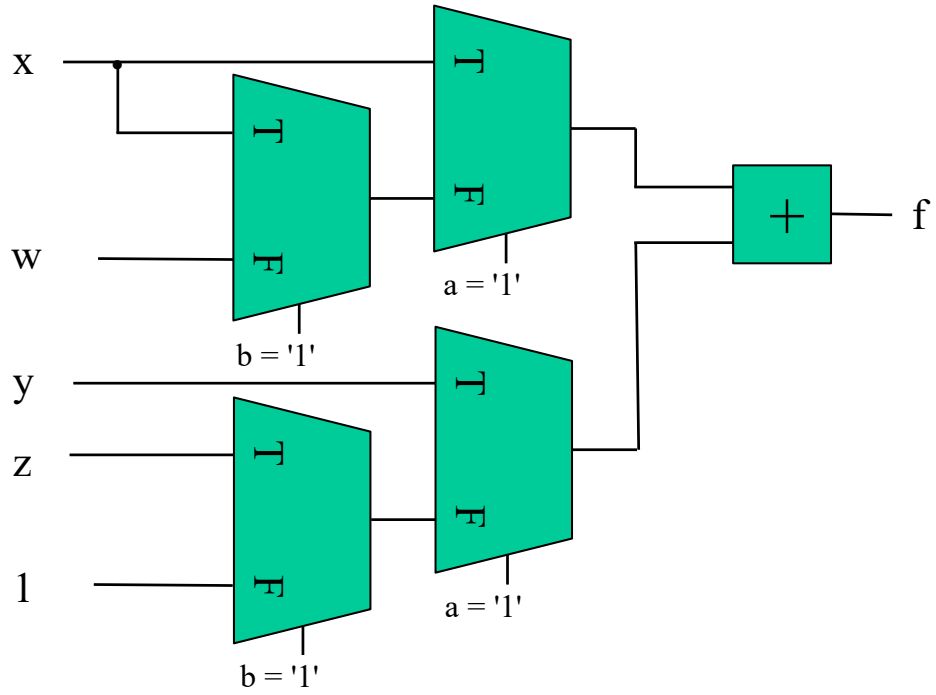
Example

```
process (w, x, y, z, a, b)
begin
  if a = '1' then
     f <= x + y;
  elsif b = '1' then
     f <= x + z;
  else
     f <= w + 1;
  end if;
end process;
```



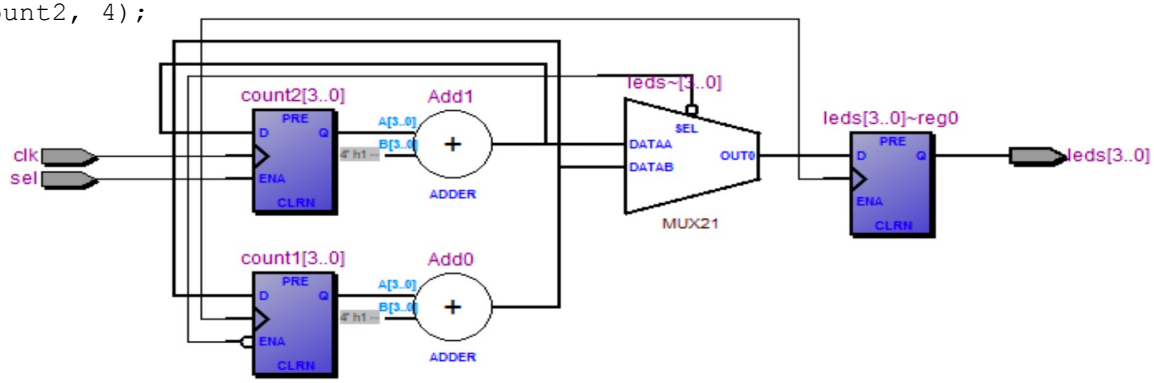
Example

```
process (w, x, y, z, a, b)
begin
  if a = '1' then
     t1 <= x;
     t2 <= y
  elsif b = '1' then
     t1 <= x;
     t2 <= z
  else
     t1 <= w;
     t2 <= 1;
  end if;
end process;
f <= t1 + t2;
```



Extra Hardware

```
architecture behavior1 of test is

begin
  process
    variable count1, count2: integer range 0 to 15 := 0;
  begin
    wait until(rising_edge(clk));
    if sel = '0' then
      count1 := count1 + 1;
      leds <= conv_std_logic_vector(count1, 4);
    else
      count2 := count2 + 1;
      leds <= conv_std_logic_vector(count2, 4);
    end if;
  end process;
end behavior1;
```



Optimized

```
architecture behavior2 of test is
begin
  process
    variable count1, count2, count: integer range 0 to 15;
  begin
    wait until(rising_edge(clk));
    if sel = '0' then
      count := count1;
    else
      count := count2;
    end if;
    count := count + 1;
    if sel = '0' then
      count1 := count;
    else
      count2 := count;
    end if;
    leds <= conv_std_logic_vector(count, 4);
 end process;
end behavior2;
```