



UiO : **Department of Informatics**
University of Oslo

IN 3160, IN4160

Combinational building blocks (and their VHDL)

Yngve Hafting



In this course you will learn about the design of advanced digital systems. This includes programmable logic circuits, a hardware design language and system-on-chip design (processor, memory and logic on a chip). Lab assignments provide practical experience in how real design can be made.

After completion of the course you will:

- understand important principles for design and testing of digital systems
- **understand the relationship between behaviour and different construction criteria**
- **be able to describe advanced digital systems at different levels of detail**
- **be able to perform simulation and synthesis of digital systems.**

Goals for this lesson:

- Know the basic structure and function of widely used combinational structures.
 - Multiplexers
 - Encoders
 - Decoders
 - Arbiter
 - Comparator
 - Shifters
 - ROM

Today: Building blocks

- About dataflow representations
- Encoders vs Decoders
- Decoder
- Multiplexer
- Encoders
- Arbiters
- Shifters
- Comparators
 - VHDL: dataflow vs RTL examples
- ROM
- RAM

Next lecture:

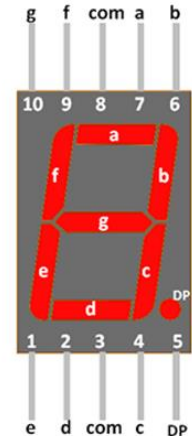
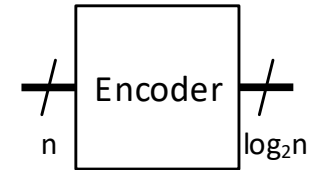
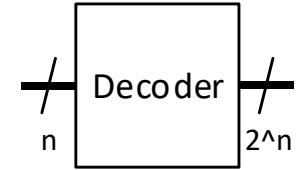
- Subroutines
- Packages & Libraries
- Clocked statements.

Data flow representations

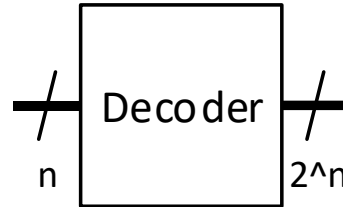
- Dataflow
 - Matches port/gate schematics
 - Use *when this is the only way* to achieve desired function
 - Tweaking (speed / area / power).
- *To show how building blocks are made,* this presentation uses low level representations
 - *Normally we want our code to be at a higher level*
 - *easier to read,*
 - *easier to maintain*
 - *lets the synthesis tool decide primitives*

Encoders and decoders

- ... convert signals from one type to another
- Encoder = inverse decoder
- Several types:
 - One hot decoder " $n \rightarrow 2^n$ "
 - Typical use: in multiplexers, memory arrays (RAM, ROM)
 - Binary encoder " $n \rightarrow \log_2(n)$ "
 - Priority encoder
 - Arbiter
 - Seven segment decoder
 - 4-5 bit Binary / BCD (binary coded decimal) to 8 bit...
- Conditional statements generally creates decoders as needed...



N to 2^N Decoder



- Ex: generic N to 2^N decoder
 - (binary to *one hot* converter)
- Demonstrates strict type check in VHDL
 - numeric_std is required for 'unsigned' and 'integer' conversions

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

```
entity decoder is  
  generic (n : positive := 4);  
  port (  
    a : in std_logic_vector (n-1 downto 0);  
    z : out std_logic_vector (2**n-1 downto 0)  
  );  
end entity decoder;
```

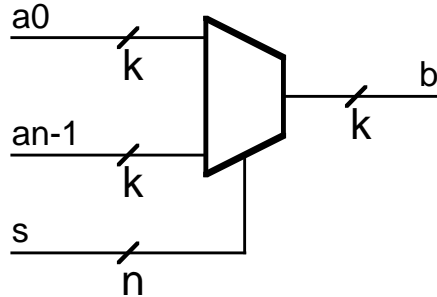
```
architecture rotate of decoder is  
  constant one_vector : unsigned (z'range) := to_unsigned(1, z'high+1);  
begin  
  z <= std_logic_vector (one_vector sll to_integer(unsigned(a)));  
end architecture rotate;
```

```
-- signal shift: integer;  
-- shift <= to_integer(unsigned(a));  
-- z <= std_logic_vector(one sll shift);
```

Total amount of bits
integer 1

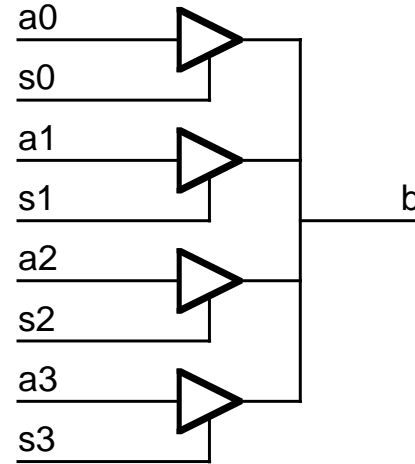
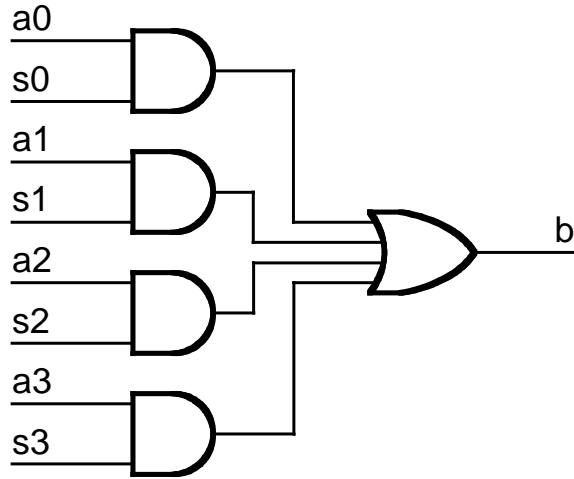
Multiplexer

- k-bit Binary-Select Multiplexer:
 - n k-bit inputs
 - n-bit one-hot select signal s
 - Multiplexers are commonly used as *data selectors*

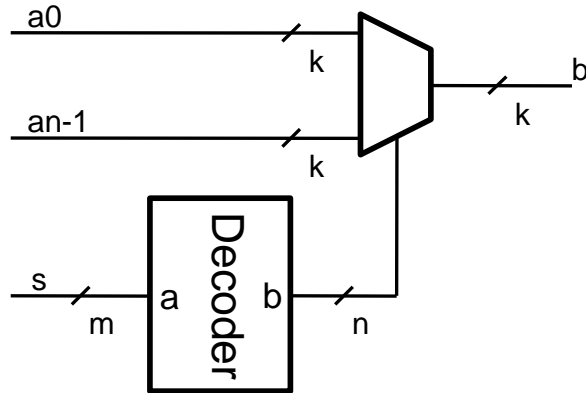


Selects one of n k-bit inputs
s must be one-hot
 $b = a[i] \text{ if } s[i]=1$

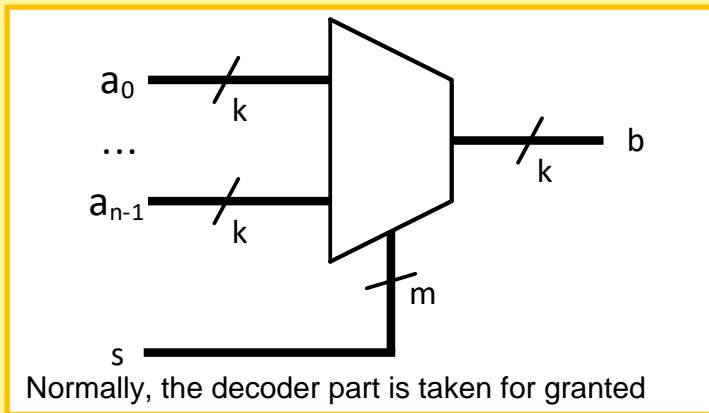
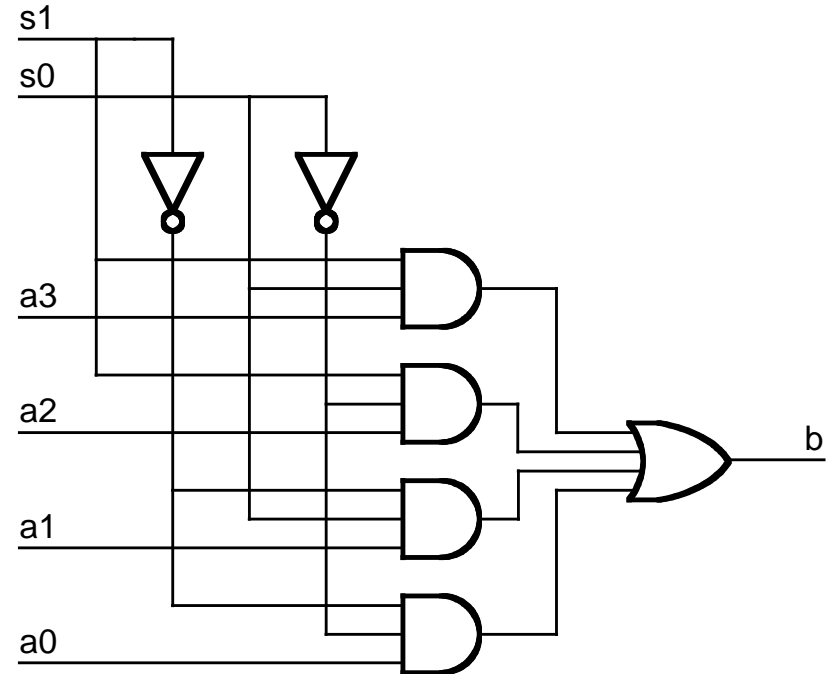
Multiplexer Implementation



k-bit Binary-Select Multiplexer (Cont)



Ex: select 1 out of 4 bit (4 to 1 bit mux)
 $k=1, m=2, n=4$



```
-- three input mux with one-hot select (arbitrary width)
```

```
library ieee;
```

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

```
entity Mux3a is
```

```
  generic( k : integer := 1 );
```

```
  port( a2, a1, a0 : in std_logic_vector( k-1 downto 0 ); -- inputs
```

```
        s : in std_logic_vector( 2 downto 0 ); -- one-hot select
```

```
        b : out std_logic_vector( k-1 downto 0 ) );
```

```
end Mux3a;
```

```
architecture case_impl of Mux3a is
```

```
begin
```

```
  process(all) begin
```

```
    case s is
```

```
      when "001" => b <= a0;
```

```
      when "010" => b <= a1;
```

```
      when "100" => b <= a2;
```

```
      when others => b <= (others => '-');
```

```
    end case;
```

```
  end process;
```

```
end case_impl;
```

```
architecture select_impl of Mux3a is
```

```
begin
```

```
  with s select b <=
```

```
    a0 when "001",
```

```
    a1 when "010",
```

```
    a2 when "100",
```

```
    (others => '-') when others;
```

```
end select_impl;
```

- Can this be implemented using 'select' statement?
 - Single input vector
 - Single output vector...
 - QED...

Structural Implementation of k-bit Binary-Select Multiplexer

```
-- 3:1 multiplexer with binary select (arbitrary width)
```

```
library ieee;
```

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

```
use work.ch8.all; -- D&H Library with generic decoder
```

```
entity Muxb3 is
```

```
  generic(k : integer:= 1);
```

```
  port(
```

```
    a2, a1, a0 : in std_logic_vector(k-1 downto 0); -- 3 k-bit inputs
```

```
    sb         : in std_logic_vector(1 downto 0);   -- binary select
```

```
    b         : out std_logic_vector(k-1 downto 0)
```

```
  );
```

```
end Muxb3;
```

```
architecture struct_impl of Muxb3 is
```

```
  signal s: std_logic_vector(2 downto 0);
```

```
begin
```

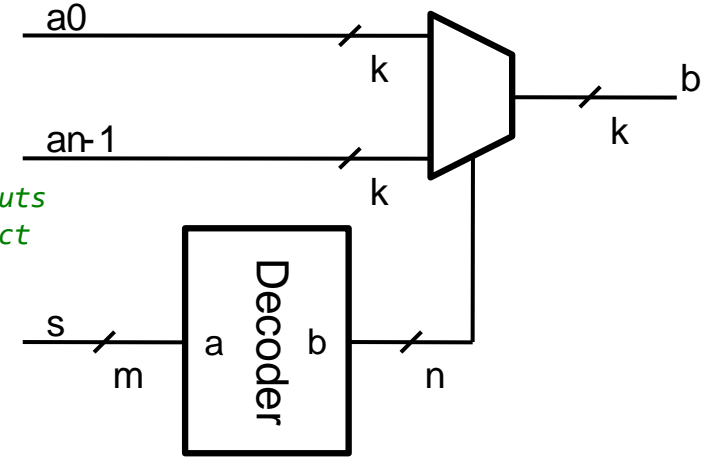
```
  -- decoder converts binary to one-hot
```

```
  d: Dec generic map(2,3) port map(sb,s);
```

```
  -- multiplexer selects input
```

```
  mx: Mux3 generic map(k) port map(a2,a1,a0,s,b);
```

```
end struct_impl;
```



Binary Encoder *(output is the number of most significant bit set)*

Don't cares vs ordered priority:

```

architecture dont_care of priority is
begin
  with a select y <=
    "00" when "0001",
    "01" when "001-",
    "10" when "01--",
    "11" when "1---",
    "00" when others;

  with a select valid <=
    '1' when "1---" | "01--" | "001-" | "0001",
    '0' when others;
end architecture dont_care;

```

```

architecture ordered of priority is
begin
  y <=
    "11" when a(3) else
    "10" when a(2) else
    "01" when a(1) else
    "00" when a(0) else
    "00";

  valid <= or a;

  --valid <='0' when a="0000" else '1';
end architecture ordered;

```

A3	A2	A1	A0	Y1	Y0	Valid
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	-	0	1	1
0	1	-	-	1	0	1
1	-	-	-	1	1	1

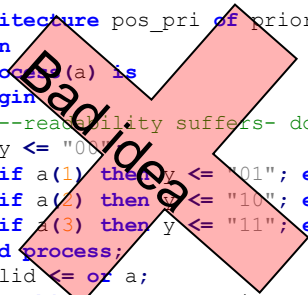
Don't cares vs sequential ordered priority

```
architecture mcase of priority is
begin
  process(a) is
  begin
    --default values
    y <= "00";
    valid <= '1';
    case? a is
      when "0001" => y <= "00";
      when "001-" => y <= "01";
      when "01--" => y <= "10";
      when "1---" => y <= "11";
      when others => valid <= '0';
    end case?;
  end process;
end architecture mcase;
```

```
architecture default_if of priority is
begin
  process(a) is
  begin
    --default values
    y <= "00";
    valid <= '1';
    if a(3) then y <= "11";
    elsif a(2) then y <= "10";
    elsif a(1) then y <= "01";
    elsif a(0) then y <= "00";
    else valid <= '0';
    end if;
  end process;
end architecture default_if;
```

```
architecture non_default of priority is
begin
  process(a) is
  begin
    --no default values
    if a(3) then
      y <= "11";
      valid <= '1';
    elsif a(2) then
      y <= "10";
      valid <= '1';
    elsif a(1) then
      y <= "01";
      valid <= '1';
    elsif a(0) then
      y <= "00";
      valid <= '1';
    else
      y <= "00";
      valid <= '0';
    end if;
  end process;
end architecture non_default;
```

```
architecture pos_pri of priority is
begin
  process(a) is
  begin
    --readability suffers- don't do
    y <= "00";
    if a(1) then y <= "01"; end if;
    if a(2) then y <= "10"; end if;
    if a(3) then y <= "11"; end if;
  end process;
  valid <= or a;
end architecture pos_pri;
```



Sequential priority (if):

- Easy to forget specifying all outputs for all input options
- Readability suffers as complexity grows

Case/ matching case:

- Test on input (a) is written once

A3	A2	A1	A0	Y1	Y0	Valid
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	-	0	1	1
0	1	-	-	1	0	1
1	-	-	-	1	1	1

Case- or selected- statement is preferred

Generic priority encoder

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

entity priority is
generic ( n : positive := 2);
port(
a : in  std_logic_vector(2**n-1 downto 0);
y : out std_logic_vector(n-1 downto 0);
valid : out std_logic
);
end entity priority;
```

```
architecture range_iterative of priority is
begin
process(a) is
begin
-- default values
valid <= '0';
y <= (others => '0');

-- iterate i from 2**n - 1 down to 0
-- set y to i when the highest bit is true
for i in a'range loop
if a(i) = '1' then
y <= std_logic_vector(to_unsigned(i,n));
valid <= '1';
exit; -- without exit (loop exit),
end if; -- lower bits would be prioritized
end loop;
end process;
```

These architectures are equivalent = Does the same

- Note how the priority changes due to the **exit** statement in the for-loop.
- Changing the direction of a range would also change priority...
 - (ie (0 to 2**n-1))



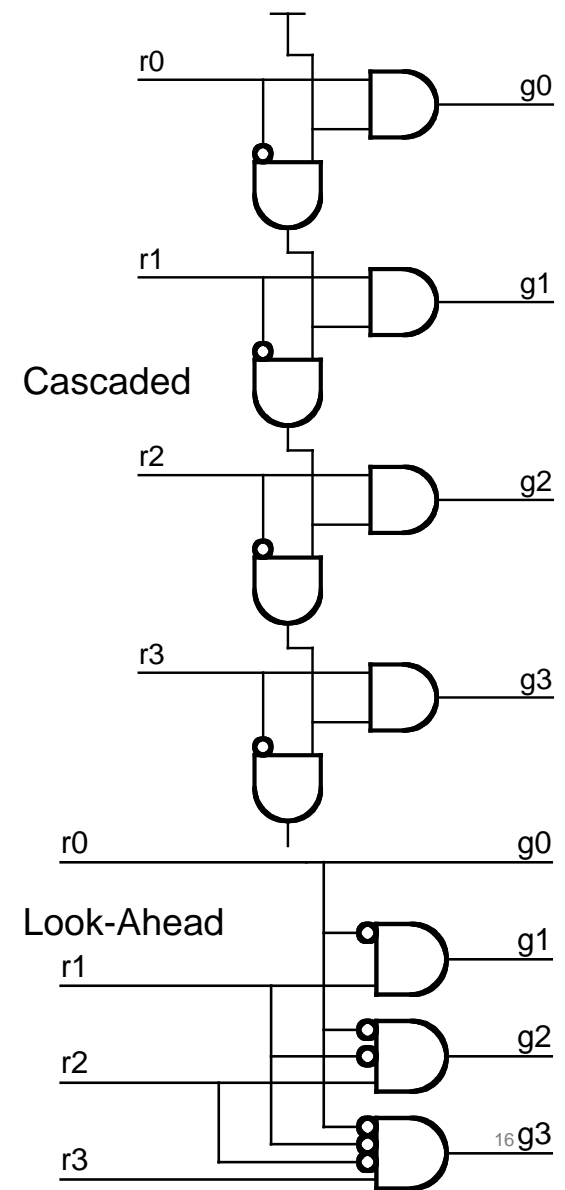
```
architecture reverse_range_iterative of priority is
begin
process(a) is
begin
-- default values
valid <= '0';
y <= (others => '0');

-- iterate from 0 to 2**n-1
for i in a'reverse_range loop
if a(i) = '1' then
y <= std_logic_vector(to_unsigned(i,n));
valid <= '1';
end if; -- no exit, means the last iteration (i=2**n-1)
end loop; -- will be prioritized for setting y
end process;
```

NOTE: All iterations create HW
how we iterate determine priority
There is no "conditional execution"

Arbiter

- Arbiters is used to sort requests for resources
 - interrupt handling in a cpu or microprocessor
 - Finds the least (or most significant) one-bit
 - cascaded vs look-ahead principle
 - VHDL = priority encoder (previous page).
 - Normally we let synthesis tool decide
 - FPGA => mostly LUT based
 - Structural code may bind a solution
 - Is it a critical feature?
 - Does not synthesis provide desired result?



Priority encoder test bench

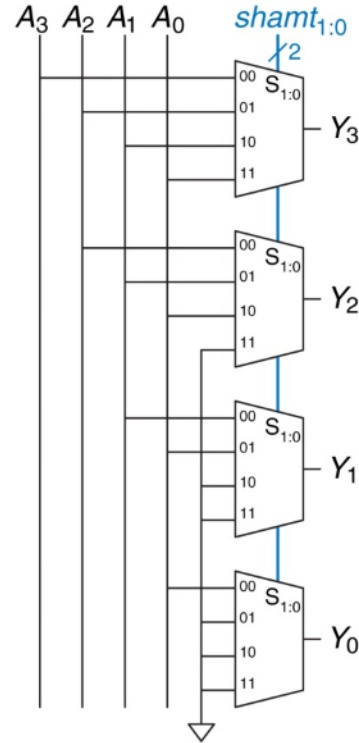
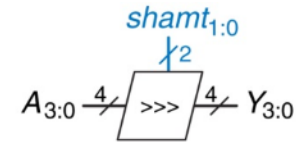
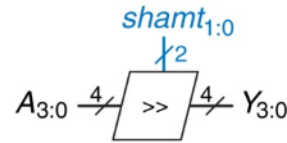
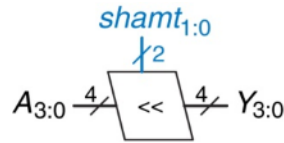
- The example makes stimuli to a combinational function independent of number of bits
- The attribute x'high gives the highest bit number to the vector x and x'low the lowest bit number
- Python equivalent tb code:

```
STIMULI :  
process  
    variable ain : integer := 0;  
begin  
    loop  
        for ain in 0 to 2**('a'high-'a'low+1)-1 loop  
            a <= std_logic_vector  
                (TO_UNSIGNED(ain, 'a'high-'a'low+1));  
            wait for 50 ns;  
        end loop;  
    end loop;  
end process;
```

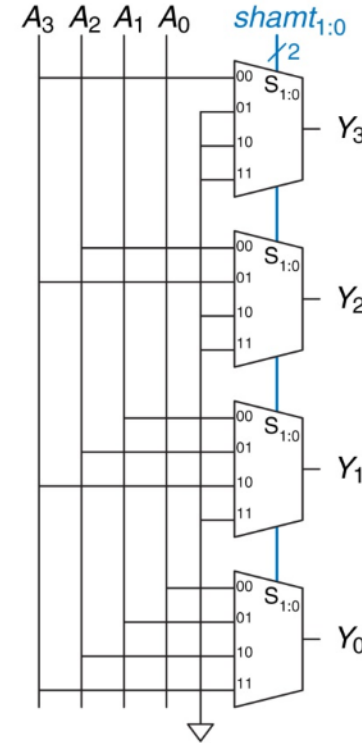
```
async def stimuli_generator(dut):  
    ''' Generates all data for this tesbench'''  
    for i in range( 2**len(dut.a)):  
        dut.a.value = i  
        await Timer(1, units= 'ns')
```


Shifters

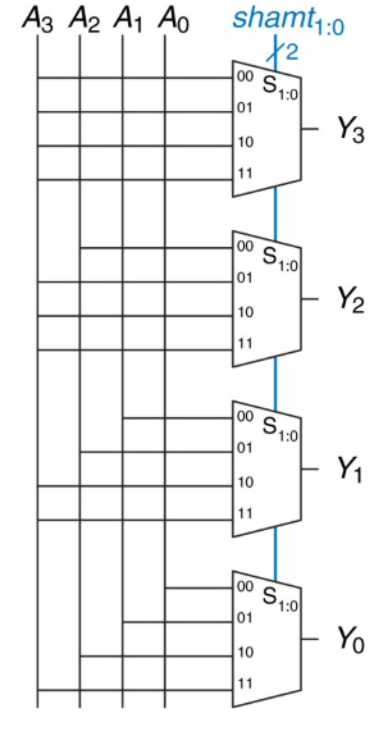
- Ex, 4 bit :
 - a) SLL
 - b) SRL
 - c) SRA



(a)



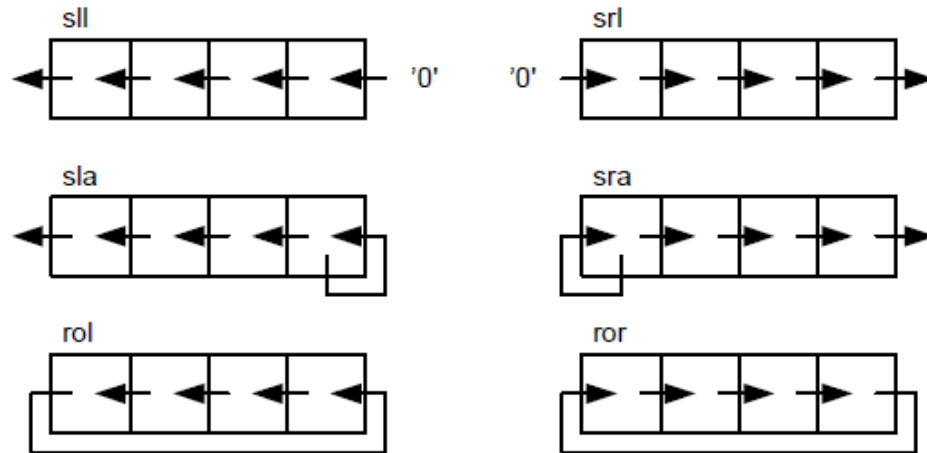
(b)



(c)

Shift operators in VHDL

- The shift operators are defined for **bit_vector** (originally)
 - and **unsigned** and **signed** in **numeric_std**
- If you are defining shift operators for other types, you have to make so called “overload”-operators
- By overload we mean that there are an already existing operator with the same name, but is written for another data type



```
-- simple shift left operation in VHDL
variable n : positive := 5;
...
a(31 downto n) <= a(31-n downto 0); -- a'high is 31,
a(n-1 downto 0) <= (others => '0'); -- a'low is 0.
```

```
-- using sll (requires numeric_std library)
a <= std_logic_vector( to_unsigned(a) sll(n) );
```

Shift operators

- The standard libraries does not define shift operators for **std_logic_vector**
- The standard synthesis library **numeric_std** defines two data types which are sub types of **std_logic**:
 - **unsigned**
 - **signed**
 - For these two it exists shift operators (overload)
- Use type casting to go between **std_logic_vector** and **signed / unsigned**

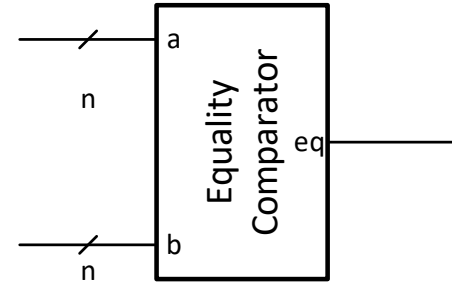
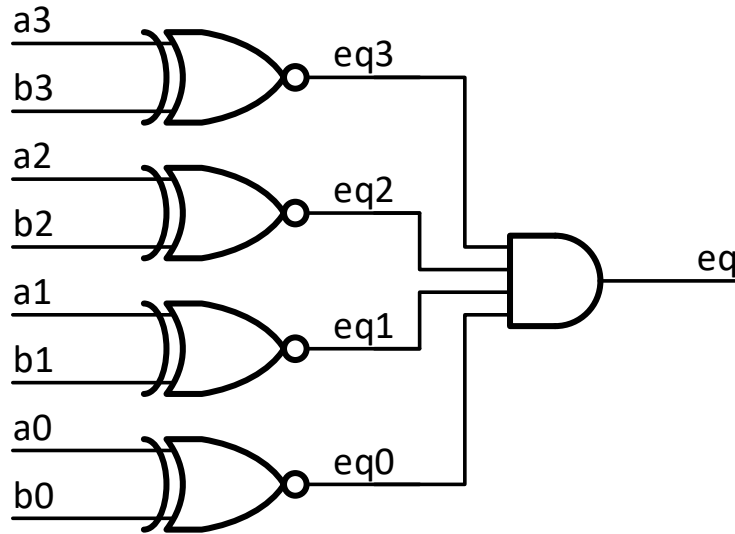
```
-- a is std_logic_vector.  
a <= std_logic_vector( to_unsigned(a) sll(n) );
```

Comparators

- Equality '='
- Magnitude '<', '>'

Equality Comparator

Is true when both input vectors are equal



```
-- high level comparator usage, IF (inside process)
if (a = b) then
  p <= q;
else
  p <= (others => '0');
end if;

-- high level comparator usage, WHEN ... ELSE
p <= q when (a = b) else (others => '0');
```

```
eq <= '1' when (a = b) else '0';
```

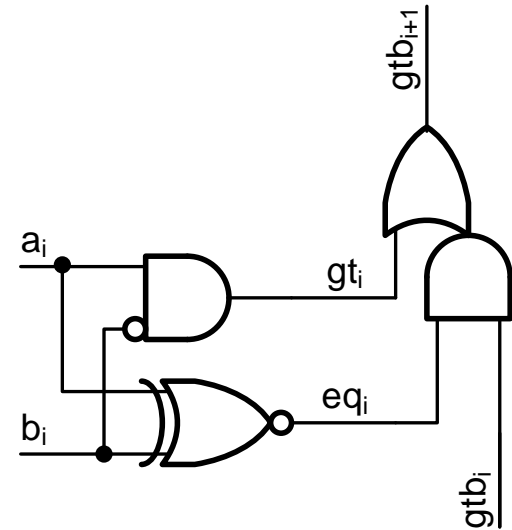
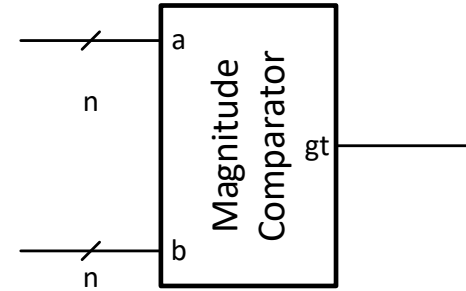
```
-- Dataflow: eq <= and (a xnor b);
```

Magnitude Comparator

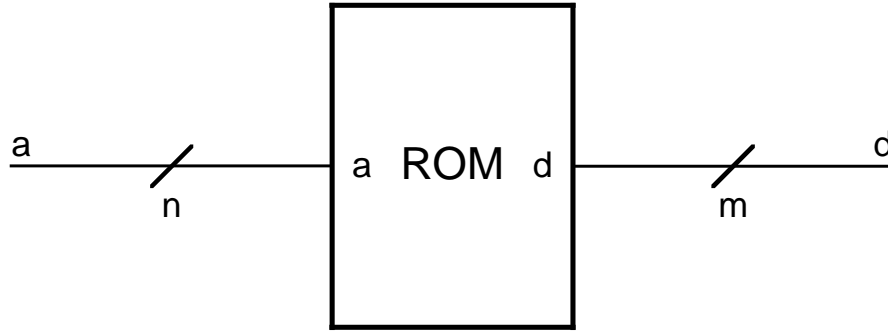
- 'if (a>b) then ...'
will infer what you need most of the time

- Dataflow example.

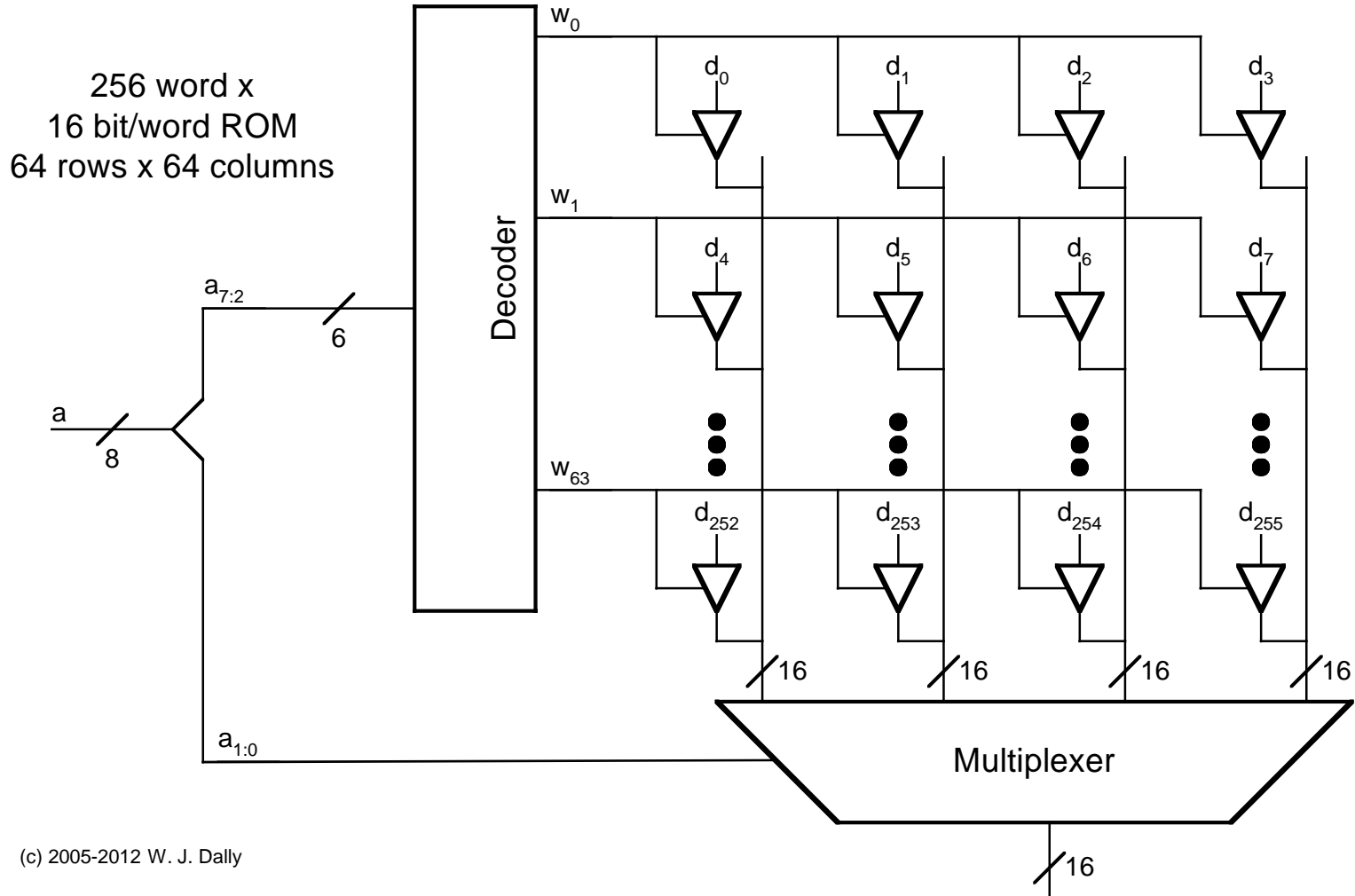
```
library ieee;  
use ieee.std_logic_1164.all;  
  
entity MagComp is  
  generic( k: integer := 8 );  
  port( a, b: in std_logic_vector(k-1 downto 0);  
        gt: out std_logic );  
end MagComp;  
  
architecture impl of MagComp is  
  signal eqi, gti : std_logic_vector(k-1 downto 0);  
  signal gtb: std_logic_vector(k downto 0);  
begin  
  eqi <= a xnor b;  
  gti <= a and not b;  
  gtb <= (gti or (eqi and gtb(k-1 downto 0))) & '0';  
  gt <= gtb(k);  
end impl;
```



Read-only memory (ROM)



2-D array implementation



ROM using VHDL

- ROM can be implemented using
 - **select**'ed statement
 - **case**
 - D&H demonstrates this.
 - **constant**'s
 - Example next slide
- File IO *can* be used to store ROM values.
 - Tools may be picky about implementations.
 - *We will look into that later.*

Example: ROM

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;

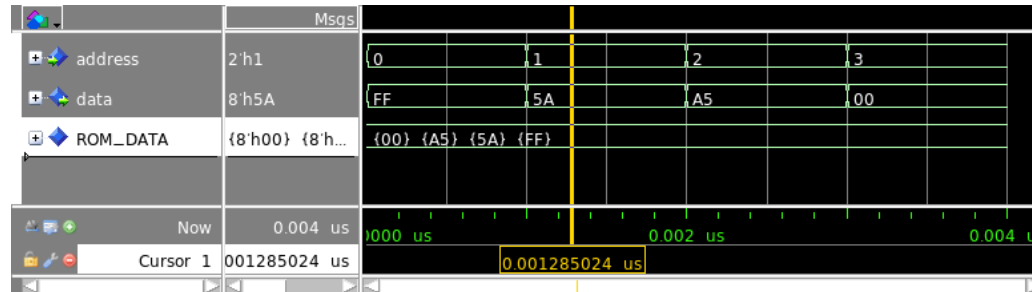
entity ROM is
  generic(
    data_width: natural := 8;
    addr_width: natural := 2);
  port(
    address: in std_logic_vector(addr_width-1 downto 0);
    data: out std_logic_vector(data_width-1 downto 0));
end entity;

architecture synth of ROM is
  type memory_array is array(2**addr_width-1 downto 0) of
    std_logic_vector(data_width-1 downto 0);

  constant ROM_DATA: memory_array := (
    8x"00", -- address 3 (from 'left to 'right)
    8x"A5", -- address 2
    8x"5A", -- address 1
    8x"FF"  -- address 0
  );

begin
  data <= ROM_DATA(to_integer(unsigned(address)));
end architecture synth;
```

- 4 byte ROM example
 - 8 bit data
 - 2 bit address
- We can define array types in VHDL
- Constants are set using :=
- Array data is listed in the sequence given by the type (array) definition
 - Here: (2**addr_width-1 downto 0) => 3, 2, 1, 0
- Indexing requires conversion to integer



RAM using VHDL

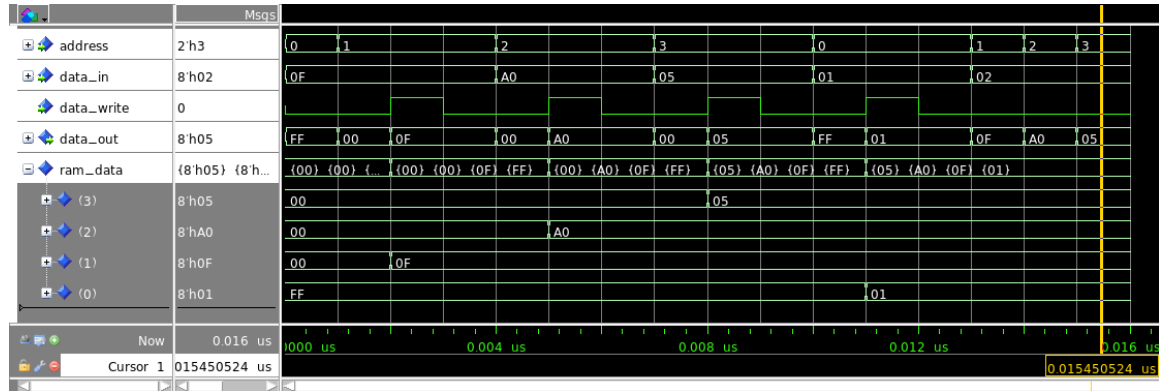
```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.numeric_std.all;

entity RAM is
generic(
    data_width: natural := 8;
    addr_width: natural := 2
);
port(
    address: in std_logic_vector(addr_width-1 downto 0);
    data_in: in std_logic_vector(data_width-1 downto 0);
    data_write: in std_logic;
    data_out: out std_logic_vector(data_width-1 downto 0)
);
end entity RAM;
```

```
architecture synth of RAM is
type memory_array is array(2**addr_width-1 downto 0) of
    std_logic_vector(data_width-1 downto 0);

signal ram_data: memory_array :=
    (8x"00", 8x"00", 8x"00", 8x"FF");
begin
data_out <=
    ram_data(to_integer(unsigned(address)));

ram_data(to_integer(unsigned(address))) <=
    data_in when data_write; -- else latched
end architecture synth;
```



- 4 Byte RAM example
- Mostly like the ROM example
 - Added write and data_in
 - Data is a **signal**, not **constant**
- Signals *may* have default values in synthesis (RAM based FPGAs)
- Writing is latched
 - *not combinational*

Suggested reading

- D&H 8.1- 8.9 p157-192
- *(8.10 PLA -> Architecture)*