Fundamental Elements of VHDL

A VHDL program consists of a collection of **design units**, each of which is defined using three components

Library and Package Declaration

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

Libraries and packages are collections of commonly used items, such as data types, subprograms and components

The above two packages define *std_logic* and *std_logic_vector* data types, as well as *signed* and *unsigned*

I also find it extremely useful to create a file with my own data types and constants, that are then included declared below the *ieee* packages

```
library work;
use work.DataTypes_pkg.all;
```

Fundamental Elements of VHDL

Entity Declaration

```
entity entity_name is
  port(
      port_names: mode data_type;
      port_names: mode data_type;
      ...
      port_names: mode data_type;
  );
end entity_name;
```

port_names are the **formal** signal names of the design unit, which are used to connect this design unit to pins on an FPGA or to other design units

The *mode* component can be **in**, **out** or **inout** (for bi-directional port)

ALWAYS use *std_logic* and *std_logic_vector* as the data_type in **entity** declarations

Fundamental Elements of VHDL

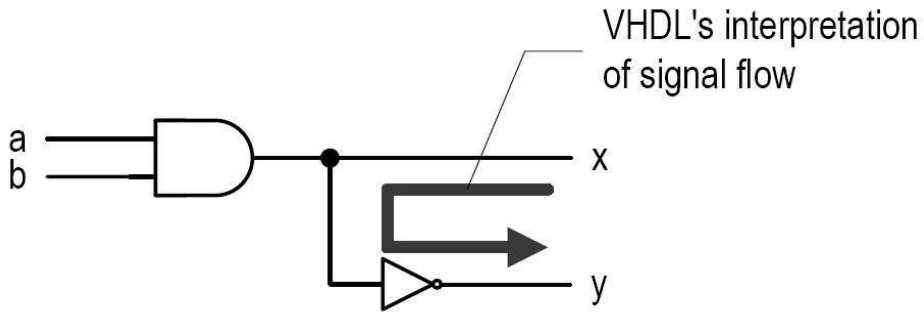
A common mistake with *mode* is to try to use a signal of mode **out** as an *input signal* within the architecture body

Consider:

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

architecture wrong_arch of mode_demo is
  begin
  x <= a and b;
  y <= not x;   -- ERROR!!!!
end wrong_arch;
```

Fundamental Elements of VHDL

Port signals defined to be *out* can NOT be read



This code reads and writes *x* so it must be defined as **inout** to avoid a syntax error

But *x* is really not a bi-directional signal in the true sense of the word

The solution you will be forced to adopt is to create an *internal* signal as follows

```
architecture ok_arch of mode_demo is
  signal ab: std_logic;
  begin
  ab <= a and b;
  x <= ab;
  y <= not ab;
end ok_arch;
```

Fundamental Elements of VHDL

Architecture Body

The architecture body specifies the logic functionality of the design unit

```
architecture arch_name of entity_name is
  declarations
  begin
  concurrent_stmt;
  concurrent_stmt;
end arch_name;
```

The *declaration* part is optional and can include **internal signal declarations** or **constant declarations**

There are several possibilities for *concurrent_stmts*, which we will cover soon

Fundamental Elements of VHDL

Comments start with two dashes, e.g.,

```
-- This is a comment in VHDL
```

An **identifier** can only contain alphabetic letters, decimal digits and underscore; the first character *must be a letter* and the last character **cannot** be an underscore

VHDL is case **IN**sensitive, i.e., the following identifiers are the same

nextstate, NextState, NEXTSTATE, nEXTsTATE

Smart convention: Use CAPITAL_LETTERs for constant names and the suffix *_n* to indicate active-low signals

Signal declaration

```
signal signal_name, signal_name, ... : data_type
```

Fundamental Elements of VHDL

The **std_logic_vector** is an array of elements with *std_logic* data type

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

The **downto** syntax puts the most significant bit (7) on the left, which is the natural representation for numbers (I rarely use the (0 **to** n) syntax)

std_logic constants are enclosed in single quotes: '1' and '0'
std_logic_vector constants are enclosed in double quotes: "00101"

Constant declaration

```
constant const_name, ... : data_type := value_expr;
```

Another smart convention:

```
constant BUS_WIDTH_LB: integer := 5;
constant BUS_WIDTH_NB: integer := 2**BUS_WIDTH_LB;
...
signal cnt: unsigned(BUS_WIDTH_LB-1 downto 0);
... if (cnt = BUS_WIDTH_NB - 1) then ...
```

Fundamental Elements of VHDL

operator	description	data type of operand a	data type of operand b	data type of result
a ** b	exponentiation	integer	integer	integer
abs a	absolute value	integer		integer
not a	negation	boolean, bit, bit_vector		boolean, bit, bit_vector
a * b	multiplication	integer	integer	integer
a / b	division			
a mod b	modulo			
a rem b	remainder			
+ a	identity	integer		integer
- a	negation			
a + b	addition	integer	integer	integer
a - b	subtraction			
a & b	concatenation	1-D array, element	1-D array, element	1-D array
a sll b	shift left logical	bit_vector	integer	bit_vector
a srl b	shift right logical			
a sla b	shift left arithmetic			
a sra b	shift right arithmetic			
a rol b	rotate left			
a ror b	rotate right			
a = b	equal to	any	same as a	boolean
a /= b	not equal to			
a < b	less than	scalar or 1-D array	same as a	boolean
a <= b	less than or equal to			
a > b	greater than			
a >= b	greater than or equal to			
a and b	and	boolean, bit, bit_vector	same as a	boolean, bit, bit_vector
a or b	or			
a xor b	xor			
a nand b	nand			
a nor b	nor			
a xnor b	xnor			

Not automatically synthesizable

Precedence	Operator
Highest	** abs not
	* / mod rem
	+ - (ident/neg)
	& + - (add/sub)
	sll srl sla sra rol ror
Lowest	and or nand nor xor xnor

Note: **and** and **or** have SAME precedence -- use parenthesis!

Fundamental Elements of VHDL

You will use *std_logic_vector* instead of *bit_vector* as defined in the table

Division by powers of 2 can be used in signal assignment stmts, e.g., *a/16*

This is implemented by the synthesis tool as a right shift operation

Division by other numbers requires a design unit that implements the division!

VHDL is a strongly-typed language, requiring frequent type casting and conversion

This is particularly evident with the *shift* operator

```
a <= resize(unsigned(b), 10) sll
        to_integer(unsigned(c));
```

Here, *a* is a *unsigned* of size 10 elements, and *b* and *c* are *std_logic_vector*

Bits or a range of bits can be referenced as

```
a(1)
a(7 downto 3)
```

Fundamental Elements of VHDL

VHDL relational operations, >, =, etc, must have operands of the same element type but their **widths may differ**

Avoid comparing operands of different widths, it's error prone

Concatenation operator (&) constructs and/or extends operands on the right

Also used to force a match between width of the operands on left and right

```
y <= "00" & a(7 downto 2);
y <= a(7) & a(7) & a(7 downto 2);
y <= a(1 downto 0) & a(7 downto 2);
```

Also useful when defining a *shift register* as we will see later

Array aggregate

```
a <= (7|5=>'1', 6|4|3|2|1|0=>'0');
a <= (7|5=>'1', others=>'0');
a <= (7 downto 3 => '0') & b(7 downto 5);
a <= (others=>'0');
```

Last assignment is very useful and works independent of the data type

Fundamental Elements of VHDL

IEEE numeric_std package

Standard VHDL and the *std_logic_1164* package support arithmetic operations only on *integer* data types

```
signal a, b, sum: integer;
. . .
sum <= a + b;
```

But this is inefficient in hardware because integer does NOT allow the range (number of bits) to be specified

We certainly don't want a 32-bit adder when an 8-bit adder would do

The *numeric_std* package allows an array of 0's and 1's to be interpreted as an *unsigned* or *signed* number, using these names as the data type

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

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

Fundamental Elements of VHDL

For *signed*, the array is interpreted in **2's-complement** format, with the MSB as the sign bit

Therefore "1100" represents 12 when interpreted as an unsigned number but -4 as a signed number

The *numeric_std* package supports arithmetic operations, including those involving integer constants

```
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, so BE VERY CAREFUL when choosing a size

Fundamental Elements of VHDL

overloaded operator	description	data type of operand a	data type of operand b	data type of result
abs a - a	absolute value negation	signed		signed
a * b a / b a **mod** b a **rem** b a + b a - b	arithmetic operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	unsigned unsigned signed signed
a = b a /= b a < b a <= b a > b a >= b	relational operation	unsigned unsigned, natural signed signed, integer	unsigned, natural unsigned signed, integer signed	boolean boolean boolean boolean

numeric_std
package definitions

function	description	data type of operand a	data type of operand b	data type of result
shift_left(a,b) shift_right(a,b) rotate_left(a,b) rotate_right(a,b)	shift left shift right rotate left rotate right	unsigned, signed	natural	same as a
resize(a,b) std_match(a,b)	resize array compare '-'	unsigned, signed unsigned, signed std_logic_vector, std_logic	natural same as a	same as a boolean
to_integer(a) to_unsigned(a,b) to_signed(a,b)	data type conversion	unsigned, signed natural integer	 natural natural	integer unsigned signed

Fundamental Elements of VHDL

There are three ***type conversion functions*** in *numeric_std* package

```
to_unsigned, to_signed and to_integer
```

Use *to_unsigned* and *to_signed* when assigning constants to *unsigned* and *signed* signals

```
a <= to_unsigned(2048, 13);
```

Assumes *a* is *unsigned* and of width 13

a is assigned the constant 2048

```
a <= resize(unsigned(b), 10) sll
        to_integer(unsigned(c));
```

Looked at this earlier -- *sll* operator requires an integer type as last operand

a must be *unsigned* of width 10

```
a(to_integer(b)) <= '1';
```

Indexing into *std_logic_vector* requires an integer data type

Here *b* must be *unsigned*

Fundamental Elements of VHDL

Type casting is also possible between 'closely related' data types

data type of a	to data type	conversion function / type casting	
unsigned, signed	std_logic_vector	std_logic_vector(a)	Type casting
signed, std_logic_vector	unsigned	unsigned(a)	
unsigned, signed	integer	to_integer(a)	Type conversion
natural	unsigned	to_unsigned(a, size)	
integer	signed	to_signed(a, size)	

```
signal u1, u2: unsigned(7 downto 0);
signal v1, v2, v3: std_logic_vector(7 downto 0);
signal sg: signed(7 downto 0);

u1 <= unsigned(v1);
v2 <= std_logic_vector(u2);
u2 <= unsigned(sg) + u1;
v3 <= std_logic_vector(unsigned(v1) + unsigned(v2));
```

Use *resize* to deal with width differences if they exist

Concurrent Signal Assignment Statements

From VHDL Essentials I, we discussed the **architecture** statement as one of the three components of a VHDL file

```
architecture arch_name of entity_name is
  declarations
  begin
  concurrent_stmt;
  concurrent_stmt;
end arch_name;
```

So far, we looked at only ***simple signal assignment*** statements as an example of a valid 'concurrent_stmt' within the architecture body

```
architecture beh of demo is
  signal ab: std_logic;
  begin
  ab <= a and b;
  x <= ab;
  y <= not ab;
end ok_arch;
```

Concurrent Signal Assignment Statements

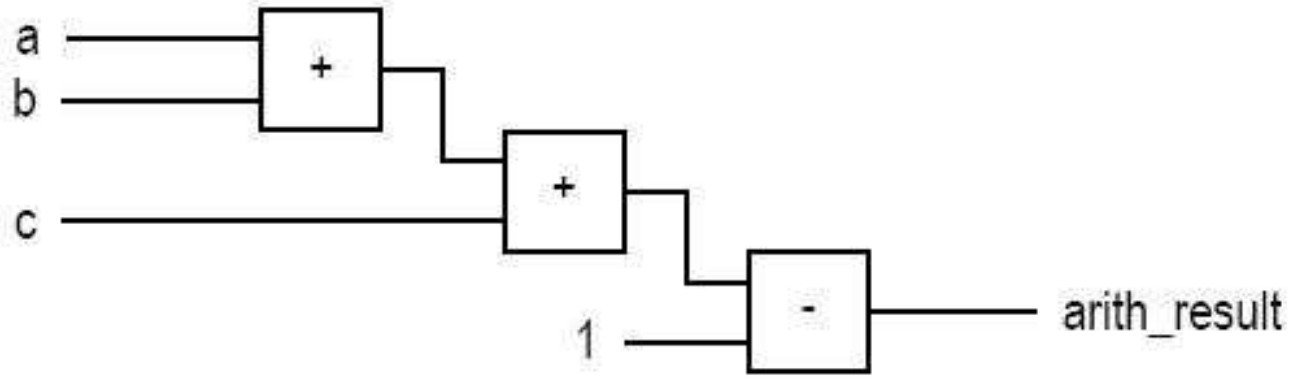
In this lecture, we consider two other types of 'concurrent_stmt', in particular, **conditional signal assignment** and **selected signal assignment**

Remember that all concurrent signal assignment statements describe hardware components that operate in **parallel**

This will be true when we discuss the **process** statement as well

For example, the following signal assignment is implemented as shown, and continuously re-computes *arith_out* as values on the wires labeled *a*, *b* and *c* change

```
arith_out <= a + b + c - 1;
```



Other signal assignments, if included, would operate **in parallel** with this circuit

Concurrent Signal Assignment Statements

One final note on *simple signal assignment*

Although it is syntactically correct to use a signal name on **both** sides of an assignment statement, NEVER do it!

```
q <=  (d and en) or ((not q) and (not en));
```

This describes a circuit which assigns to *q* the value of *d* when *en* is '1', otherwise it assigns the *inverse* of itself *not q*

This statement forms a *combinational feedback loop* and will not be synthesized and behave as you expect

You will encounter plenty of instances where you want to do this type of assignment, however, this is not the correct way to do it

We will discuss how to properly describe a circuit with this type of assignment later when we discuss D flip-flops

Selected Signal Assignment Statements

Selected signal assignment describes a circuit that implements a ***case statement*** in a programming language

```
with select_expression select
    signal_name <= value_expr_1 when choice_1,
                   value_expr_2 when choice_2,
                   value_expr_3 when choice_3,
                   ...
                   value_expr_n when others;
```

The *select_expression* is usually of type *std_logic* or *std_logic_vector*

choice_x are usually constants such as "00", "01", "10" and "11"

The *choices_x* must be **mutually exclusive** and **all inclusive**, i.e., always use **others** as the condition in the last clause to ensure this

The *selected signal assignment* statement is VERY COMMONLY used to describe a MUX-based circuit

Selected Signal Assignment Statements

```
architecture sel_arch of mux4 is
   begin
   with s select
     x <= a when "00",
          b when "01",
          c when "10",
          d when others;
end sel_arch;
```

Describing a **binary decoder** (n-to-2^n) is another common usage scenario

```
architecture sel_arch of decoder4 is
   begin
   with s select
     x <= "0001" when "00",
          "0010" when "01",
          "0100" when "10",
          "1000" when others;
end sel_arch;
```

Selected Signal Assignment Statements

Selected signal assignment can also be used to describe a circuit which uses a **truth table** as the source

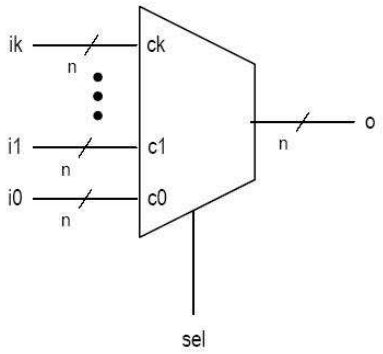
input a b	output y
0 0	0
0 1	1
1 0	1
1 1	1

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

Selected Signal Assignment Statements

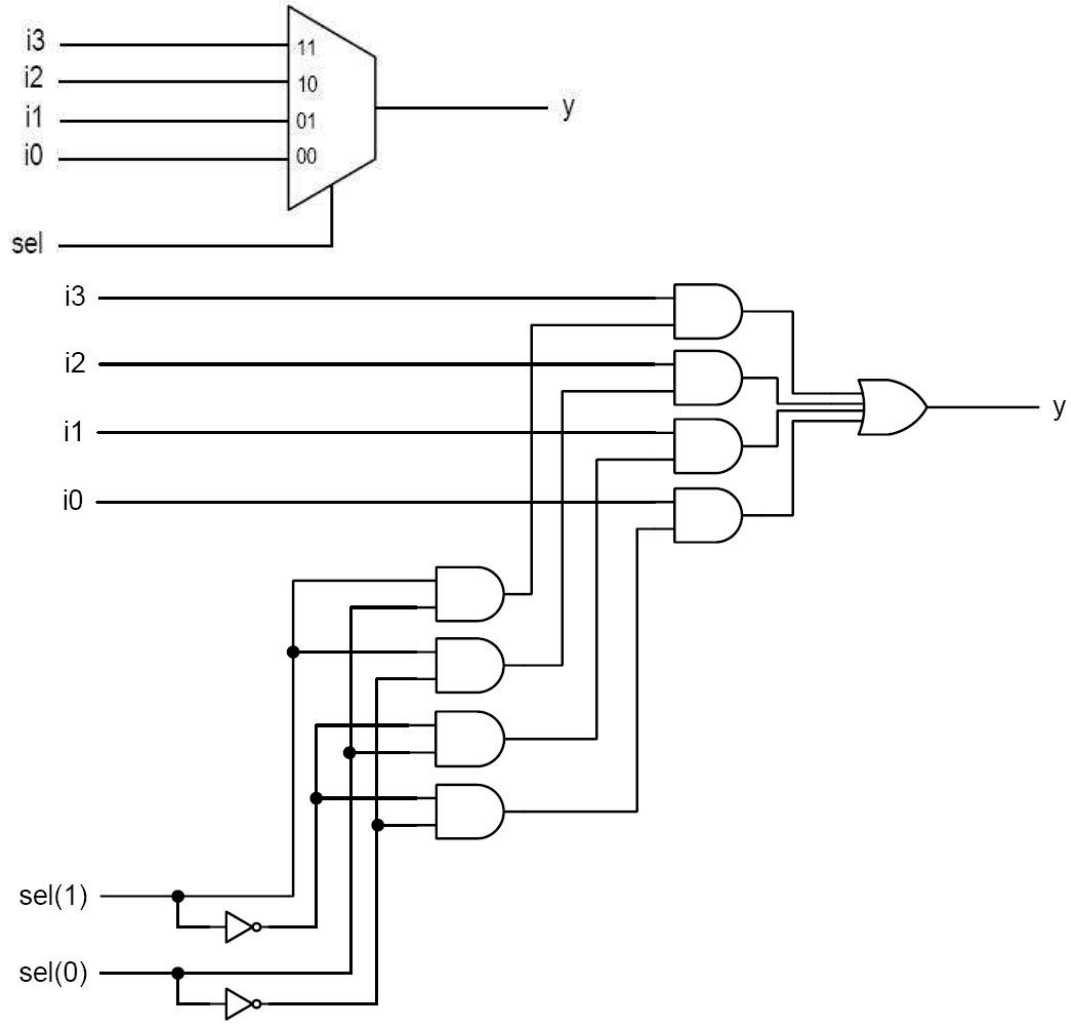
```
architecture beh of OR_gate_truth_table is
   signal tmp: std_logic_vector(1 downto 0);
   begin
   tmp <= a & b; -- concatenate a and b
   with tmp select
      y <= '0' when "00", -- rows of the truth table
           '1' when "01",
           '1' when "10",
           '1' when others;
end beh;
```

The conceptual implementation of a *selected signal assignment* is simply a **MUX**



Selected Signal Assignment Statements

When synthesized, a 4-to-1 MUX can be mapped into logic gates

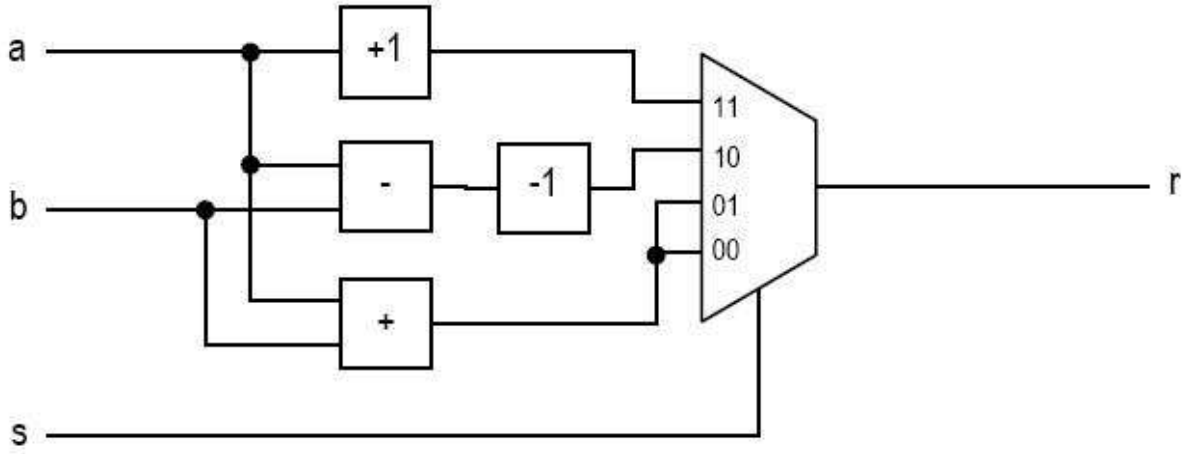


This circuit selects either *i0*, *i1*, *i2* or *i3* depending on the values of *sel(0)* and *sel(1)*

Selected Signal Assignment Statements

More complex MUX-based multi-bit arithmetic circuits can also be described

```
signal a, b, r: unsigned(7 downto 0);
signal s: std_logic_vector(1 downto 0);
...
with s select
   r <= a+1 when "11",
        a-b-1 when "10",
        a+b when others;
```



Conditional Signal Assignment Statements

Conditional signal assignment describes a circuit that implements an ***if stmt*** in a programming language

```
sig_name <=
      value_expr_1 when boolean_expr_1 else
      value_expr_2 when boolean_expr_2 else
      value_expr_3 when boolean_expr_3 else
      ...
      value_expr_n
```

Similar to *if stmts*, each of the *boolean_expr_x* generate **1** or **0** with the first one that generates a 1 causing the *value_expr_x* to be assigned to the *sig_name* signal

Only use *conditional signal assignment* when it is NOT possible to use *selected signal assignment*

The synthesis tool will generally use more logic gates to implement *conditional signal assignment* because of the **priority** that exists among the choices

boolean_expr_1 takes priority over *boolean_expr_2*

Conditional Signal Assignment Statements

A **priority encoder** is a good example of when you should use *conditional signal assignment*

A priority encoder checks the input requests and generates the code for the request of *highest priority*

input	output	
r	code	active
1 – – –	11	1
0 1 – –	10	1
0 0 1 –	01	1
0 0 0 1	00	1
0 0 0 0	00	0

There are four input requests, *r(3)*, ..., *r(0)*

The outputs include a 2-bit signal (*code*), which is the binary code of the highest priority request and a 1-bit signal *active* that indicates if there is an active request

Here, *r(3)* has the highest priority, i.e., when asserted, the other three requests are ignored and the *code* signal becomes "11"

When *r(3)* is **not** asserted, the second highest request, *r(2)* is given priority

The *active* signal is used to distinguish between the cases when *r(0)* is asserted and the case in which NO request is asserted

Conditional Signal Assignment Statements

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

entity prio_encoder42 is
  port(
     r: in std_logic_vector(3 downto 0);
     code: out std_logic_vector(1 downto 0);
     active: out std_logic);
end prio_encoder42;

architecture beh of prio_encoder42 is
   begin
   code <= "11" when (r(3)='1') else
           "10" when (r(2)='1') else
           "01" when (r(1)='1') else
           "00";
   active <= r(3) or r(2) or r(1) or r(0);
end beh;
```

Conditional Signal Assignment Statements

The *conditional signal assignment* statement implements a priority structure and requires additional logic to implement the *priority routing network*

Selected signal assignment requires the first two of the following circuits to be implemented while *conditional signal assignment* requires all three

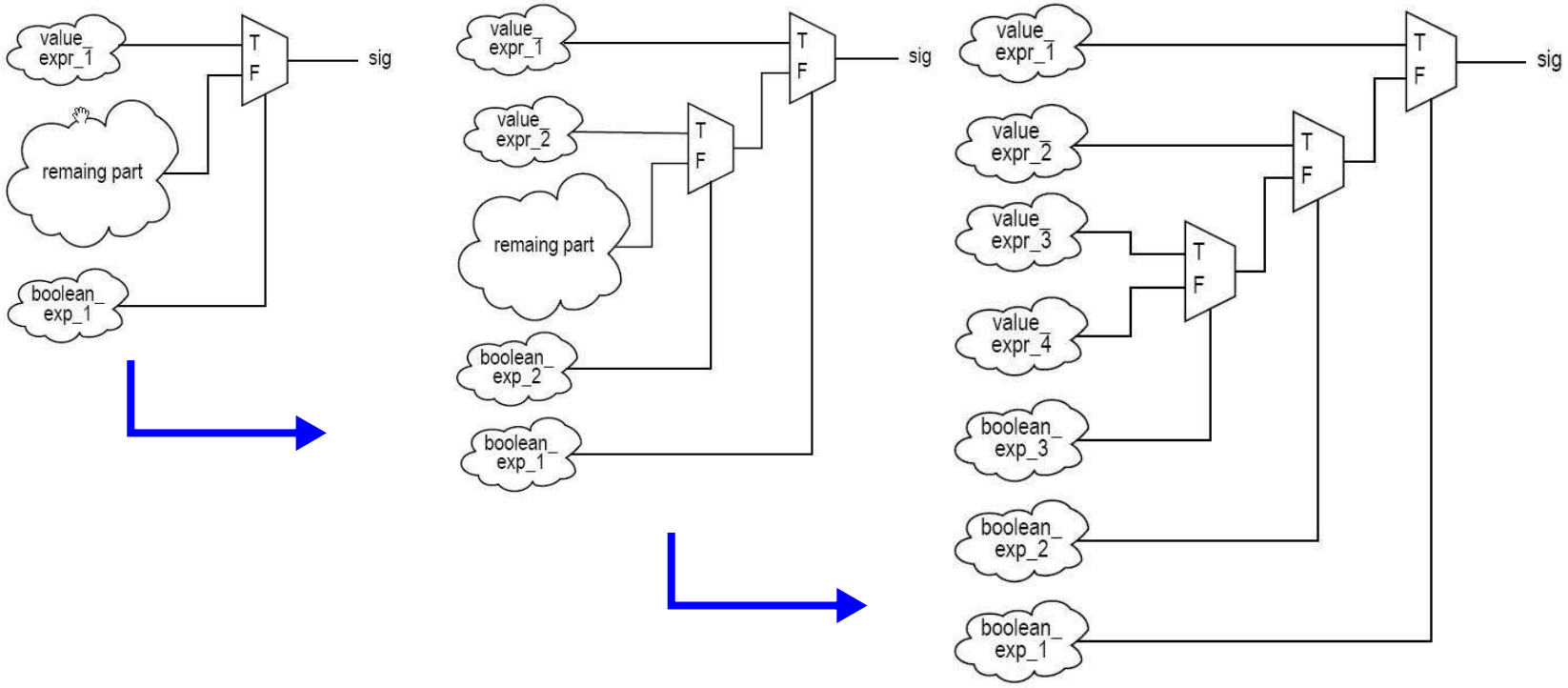
- Value expression circuits
- Boolean expression circuits
- Priority routing network

A *layered* sequence the MUXes are used to implement the *priority routing network* which determines which *value expression* circuit is connected to the output

The outputs of circuits that implement the *Boolean* expressions are used to drive the *select* inputs to the layered sequence of MUXes

Conditional Signal Assignment Statements

```
sig <= value_expr_1 when boolean_expr_1 else
       value_expr_2 when boolean_expr_2 else
       value_expr_3 when boolean_expr_3 else
       value_expr_4;
```



Adding **when** clauses increases the overall combinational delay of the circuit, so you will be limited in how many **when** clauses you can use

Process Blocks

The last of the 'concurrent_stmts' that we will consider is the **process** block

```
architecture arch_name of entity_name is
  declarations
  begin
  concurrent_stmt;
  concurrent_stmt;
end arch_name;
```

The entire contents of a **process** block operate ***in parallel*** with other *concurrent_stmt*

The **process** block itself is a concurrent statement and should be thought of as a **sub-circuit enclosed inside a black box**

You will 'read elsewhere' that the **process** block is a container for a set of *sequential statements*, which 'executes' from top to bottom

And you will immediately be tempted to cut-and-paste your C code into one

I give you fair warning, there is nothing *sequential* about a **process** block nor about it *executing* and any attempt to treat it as a 'container' for C code will fail hard!

Process Blocks

The **process** block provides semantics that allow you to describe a circuit in a *recipe-type* fashion

VHDL synthesis tools obey special semantics when interpreting statements within a **process** block

The majority of your descriptions of combinational logic and ALL of your descriptions of storage elements (DFFs) will be done inside a **process** block

VHDL allows a lot of different types of constructs to show up in a **process** block

We will restrict the components that go into a **process** block to three components, NO exceptions!

- *signal assignment*
- *if stmts*
- *case stmts*

There are also two forms accepted by VHDL for a **process** block

We will ONLY use the version with the *sensitivity list* (NO **wait** statements)

General Form of the Process Block

Our restricted form of the **process** block is given as follows

```
process(sensitivity_list)
    begin
    statement;
   ...
end process;
```

The *sensitivity_list* lists the **input** signals to the sub-circuit you are describing

The following shows a simple example of a **process** block using *signal assignment*

```
signal a, b, c, y: std_logic;
process(a, b, c)
    begin
    y <= a and b and c;
end process;
```

Although valid, you should consider *signal assignment* without the **process** block as an alternative, equivalent description of this 3-input AND gate

```
y <= a and b and c;
```

Golden Rules of Process Blocks

Before discussing the *if stmt* and the *case stmt*, let's cover my **golden rules**

NEVER VIOLATE THESE RULES

- **Rule 1:** All signals that are **read** with a **process** block must be included in the sensitivity list

  This includes signals that appear on the right side of *assignment* statements

  AND those that appear within Boolean expressions in *if* and *case* statements

- **Rule 2**: The **last assignment** to an output signal takes precedence over ALL previous assignments (listed earlier in the **process** block)

```
output_signal <= value_expression1;
...
if ( x = '1' ) then
   output_signal <= value_expression2;
end if;
```

If *(x = '1')* evaluates to be true, then *output_signal* is assigned *value_expression2*

Golden Rules of Process Blocks

NEVER VIOLATE THESE RULES

- **Rule 3**: All output signals MUST have an UNCONDITIONAL assignment

  You MUST include unconditional assignments to ALL output signals as the top-most set of assignment statements in your **process** block

- **Rule 4**: Never use a signal on both sides of an assignment statement (inside or outside of a **process** block)

```
tmp <= tmp or b;
```

  As was true of signal assignment outside the **process** block, this results in a combinational loop with the output connected to one of the inputs

- **Rule 5**: Never assign to an output signal both *inside* and *outside* of a **process** block

  If you use a *simple signal, selected signal,* or *conditional signal assignment* to assign a value to a signal, do NOT assign to it within a **process** block

  Turns out that multiple assignments to an output signal within a **process** block is allowed and is mandatory according to Rule 3 above!

Case Stmts Within a Process Block

The *case stmt* is equivalent to the *selected signal assignment* discussed earlier but is more general (similar in appearance to programming languages)

```
case case_expression is
   when choice_1 =>
      statements;
   ...
   when others =>
      statements;
end case;
```

As was true for *selected signal assignment*, *choice_x* terms must be **mutually exclusive** and **all inclusive**

```
x <= z;
case s is            -- Creates a MUX structure in the
   when "00" =>      -- hardware, similar to selected
      x <= a;        -- signal assignment
   when "01" =>
...
```

If Stmts Within a Process Block

The *if stmt* is equivalent to the *conditional signal assignment* discussed earlier but is more general (also similar in appearance to programming languages)

```
if boolean_expr_1 then
   statements;
elsif boolean_expr_2 then
   statements;
...
else
   statements;
end if;
```

You can use the *if stmt* to describe the 4-to-2 priority encoder discussed earlier

```
entity prio_encoder42 is
   port (
      r: in std_logic_vector(3 downto 0);
      code: out std_logic_vector(1 downto 0);
      active: out std_logic);
end prio_encoder42;
```

***If Stmts* Within a Process Block**

```
architecture if_arch of prio_encoder42 is
   begin
   process(r)
      begin
      code <= "00";
      if ( r(3)='1' ) then
         code <= "11";
      elsif ( r(2)='1' )then
         code <= "10";
      elsif ( r(1)='1' )then
         code <= "01";
      else
         code <= "00";
      end if;
   end process;
   active <= r(3) or r(2) or r(1) or r(0);
end if_arch;
```

If Stmts Within a Process Block

if stmts can be nested, as shown here when finding the max of *a*, *b* and *c*

```
process(a, b, c) -- NOTE: if stmts create a priority
   begin         -- network, i.e., a layered sequence
   max <= a;     -- of MUXes as shown earlier
   if (a > b) then
      if (a > c) then
         max <= a;
      else
         max <= c;
      end if;
   else
      if (b > c) then
         max <= b;
      else
         max <= c;
      end if;
   end if;
end process;
```

Consequences of Violating the Golden Rules of Process Blocks

- **Violating Rule 1**: Failing to include a signal that is read in the sensitivity list usually results in a 'WARNING' from the synthesis tool

But can (technically) result in the tool adding a memory element for the output signal

```
process(a)
    begin
    y <= a and b and c;
end process;
```

Signals *b* and *c* are not listed as inputs to this circuit, which means that *y* needs to maintains its value (FF) until *a* changes (*a* is treated as a clock that samples *b* and *c*)

- **Violating Rule 2**: There is no way to violate this rule regarding multiple assignments to an output signal within a **process** block

- **Violating Rule 3**: Failing to assign an output signal a *default* value causes **real problems** for students, with only an 'inferred xxx' warning from the tool

  A storage element is added to *preserve* the output signal value when **none** of the Boolean expressions are true (no assignment is made to the output signal)

Consequences of Violating the Golden Rules of Process Blocks

- **Violating Rule 3**:

The following 'instructs' the synthesis tool to add a storage element for *eq*

```
process(a, b)
    begin
    if (a = b) then
        eq <= '1';
    end if;
end process;
```

No assignment is made when *a* does not equal *b* -- the synthesis tool infers this as:

```
process(a, b)
    begin
    if (a = b) then
        eq <= '1';
    else
        eq <= eq;
    end if;
end process;
```

Consequences of Violating the Golden Rules of Process Blocks

- **Violating Rule 3**:

  The synthesis tool issues 'inferred xxx' warning, and happily creates a storage element to store *eq* until the Boolean condition ($a = b$) is true again!

  In fact, a variation of this syntax is used to describe valid FFs so it commonly used (as we will soon see)

THIS IS A COMMON BUG so beware!!!!

```
process(a, b) -- THIS VIOLATES GOLDEN RULE 3!
   begin      -- MUST INCLUDE DEFAULT ASSIGNMENTS HERE
   if (a > b) then
      gt <= '1';
   elsif (a = b) then
      eq <= '1';
   else
      lt <= '1';
   end if;
end process;
```

Consequences of Violating the Golden Rules of Process Blocks

- **Violating Rule 4**: Creating combinational loops generally cause the synthesis tool to generate a 'WARNING', but beware!

- **Violating Rule 5**: Assigning to an output signal more than once outside a **process** block, or both inside and outside a **process** block generates a syntax error

FFs and Registers

In this lecture, we show how the **process** block is used to create FFs and registers

Flip-flops (**FFs**) and **registers** are both derived using our standard data types, *std_logic*, *std_logic_vector*, *signed* and *unsigned*

Storage elements are critical to emulating *variables* in programming languages

They play a central role in allowing C programs to be converted into hardware implementations

VHDL (and verilog) allow complex hardware to be described in either **single-segment** style to **two-segment** style

Proponents of single-segment style argue that such descriptions are

- More efficient from a simulation perspective (the sensitivity list consists of *clk* only)
- More concise, i.e., requiring fewer VHDL statements to describe the circuit

Neither of these are compelling reasons, and neither offset the benefits of two-segment style (in my opinion)

One-Segment vs. Two-Segment Style

We will use **two-segment** style exclusively throughout the rest of this lecture series for several reasons:

- Two-segment provides a *conceptual advantage* by cleanly separating storage elements from the combinational logic portion of the design

  After many years of experience, the biggest challenge of writing VHDL is being able to **quickly** craft a description that has the fewest **bugs**

  The advantage afforded by partitioning the circuit into combinational and sequential components is difficult to over-state

- Two-segment style will provide opportunities to guide the synthesis tool to produce a more efficient hardware implementation (in my opinion)

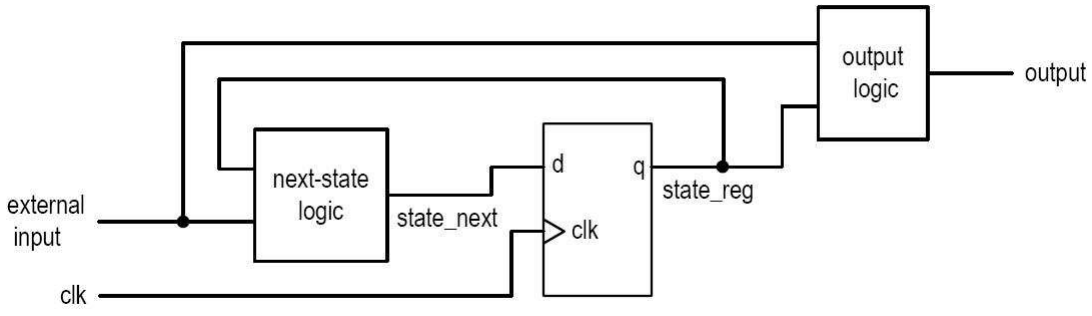
  Two-segment style provides **easy access** to **both** the inputs and outputs of FFs and registers

  Two-segment style will also allow the designer to easily specify **signals** (wires) in combinational circuit descriptions, without being forced to create FFs

FFs and Finite State Machines

We will focus on creating designs with a *single-clock domain* and a globally distributed *clk* signal (*globally synchronous*)

The end goal of our learning will be to create a **finite state machine** (**FSM**)



State registers (state_reg) represent the storage elements

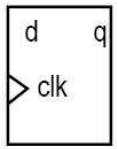
Next state logic represent the combinational circuit that determines *state_next*

From previous courses, you probably remember the following about FSM operation:

- At the rising edge of the clock, *state_next* is sampled and stored into the register (and becomes the new value of *state_reg*)
- The external *inputs* and *state_reg* signals propagate through **next-state** and **output** logic to determine the new values of the *state_next* and *output* signals
- This sequence of operations repeats indefinitely

FFs

DFFs are the workhorse of modern digital circuit design, and they play a central role in **FSM** and **datapath** implementations



clk	q*
0	q
1	q
⌐	d

(b) pos-edge triggered D FF

The truth table of a DFF specifies that the state of the FF remains *unchanged* until a **rising edge** of the clock arrives

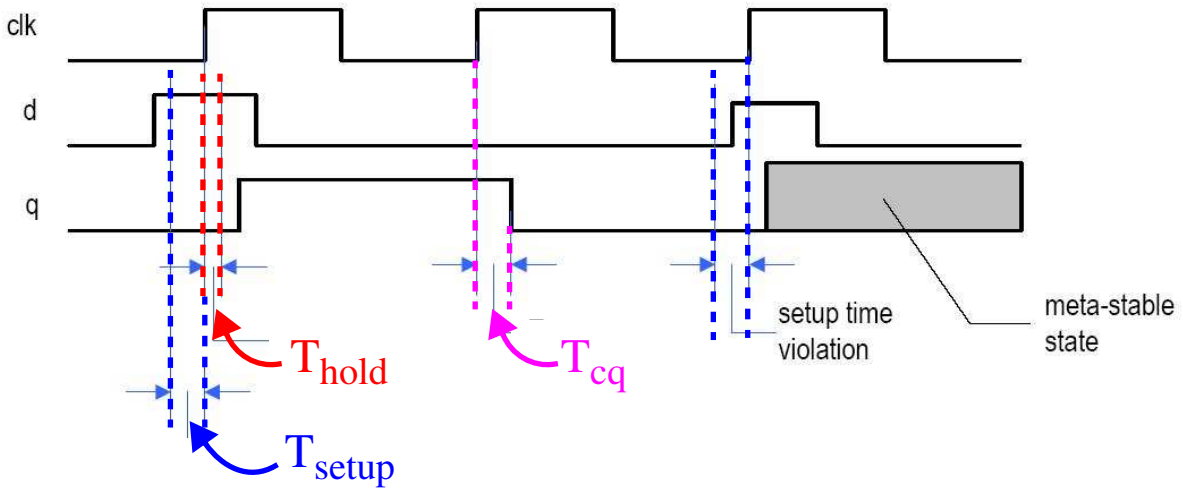
This type of FF is referred to a **rising-edge-triggered DFF**, or **FF** for short (we will NEVER use falling-edge-triggered FFs)

Most FFs that you will create will also have a *set* or *reset* signal

FFs

It is important to have a solid understanding of the timing diagram for a FF

Sooner or later, you will encounter timing violations in your design and will need to either 1) fix them or 2) slow the clock down



Every storage element has *setup* and *hold* time requirements

- Setup time is the amount of time a signal (driving the *d* input) needs to be stable **before** the rising edge of the *clk*
- Hold time is the amount of time this signal needs to be maintained on *d* **after** the rising edge of the *clk*

FFs

Timing violations are almost always *setup-time* violations

When this happens, the length of the path through the combinational circuit is TOO long

This can happen if you have too many **when-else** clauses in a conditional signal assignment, which creates a critical path that is longer than the *clk* cycle time

When you set the *clk* frequency during synthesis to, say, 50 MHz, ALL combinational paths in your design MUST be less than (20 ns **-** *setup-time)*

The synthesis tool will work very hard to create implementations from your VHDL descriptions that meet the timing requirements

If it fails, the onus is on you to fix the timing violations

We will discuss simple strategies that you can use to deal with timing violations when we get to FSM design

FFs

Use the following **process** block construct to create a FF with an **asynchronous reset**

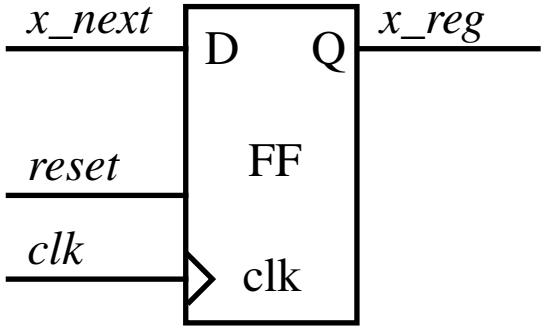
```
architecture beh of example_design is
   signal x_reg, x_next: std_logic;
   begin
   process(clk, reset)
      begin
      if ( reset = '1' ) then
         x_reg <= '0';
      elsif ( rising_edge(clk) ) then
         x_reg <= x_next;
      end if;
   end process;

   ...
end beh;
```

Memorize this syntax! You will use this same structure over-and-over again, in every VHDL module you create

FFs

The FF has two names, which specify the input, *x_next* and the output, *x_reg*

The synthesis tool creates a FF as follows from this **process** block description:



If you attempt to synthesize this by itself, the synthesis tool will delete the FF b/c the input and output are not connected to anything (they float)

In most design scenarios, you will want to 'control' updates to your FFs

In other words, your FFs will maintain their contents *most* of the time, and only occasionally will be updated with new values (on one of the rising *clk* edges)

There are two ways of doing this:

- Add a MUX before the input *x_next*
- Gate the clock, i.e., add an AND gate in series with the *clk* connection

FFs

Golden Rule of Digital Design: Never insert any logic in series with the *clk*

Doing so makes it difficult for the synthesis tool to do a proper timing analysis

Experienced circuit designers will violate this rule to reduce the switching frequency of the FFs and save energy

So unless you have a really good excuse, DON'T DO IT.

The alternative of using a MUX is by far the most common method

This is easily specified using a **with-select** or **when-else** statement

```
with en select
    x_next <= new_val when en ='1',
              x_reg when others;
```

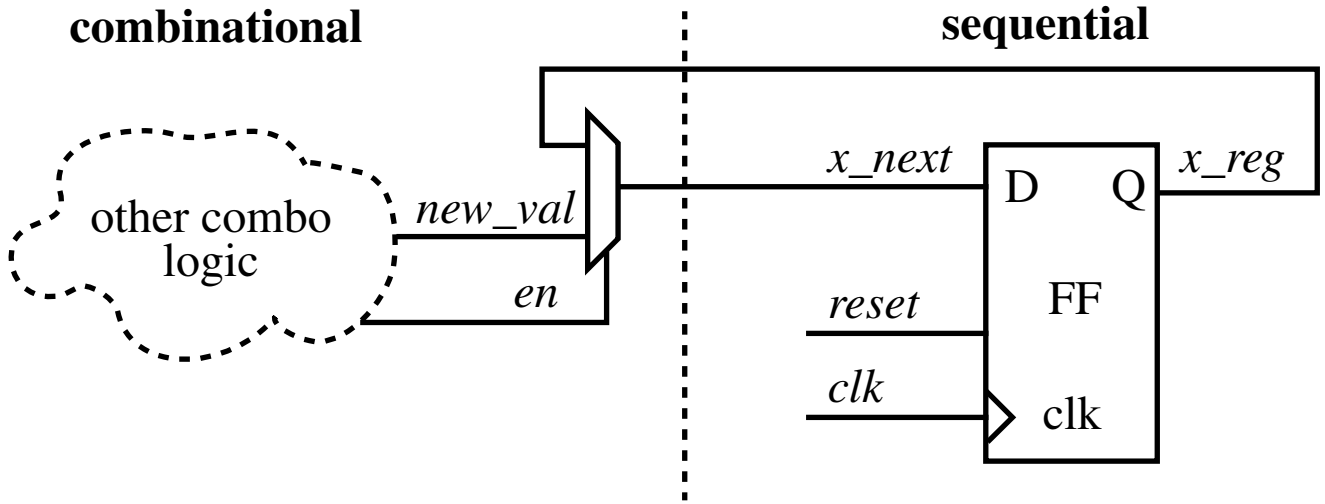
OR

```
x_next <= new_val when en ='1' else
          x_reg;
```

As we discussed, **with-select** is a better match since we are describing a MUX, but many times the Boolean expression may be more complex, with multiple signals

FFs

In either case, the following schematic is created by the synthesis tool



Note that I've separated the design into *combinational* and *sequential* components, and placed the MUX in the combinational component

As I mentioned, two-segment style creates this conceptual representation where:

- Combo: *_reg* signals are INPUTS and *_next* signals are OUTPUTS
- Seq: *_reg* signals are OUTPUTS and *_next* signals are INPUTS

The *other combo logic* portion must be specified, otherwise the synthesis tool will eliminate the FF and MUX -- more on this soon...

Golden Rules of FF Process Blocks

We will 'close the loop' and create a valid design soon, but let's first cover my **golden rules** on **process** blocks which describe FFs

NEVER VIOLATE THESE RULES

- **Rule 1:** Never include anything except *clk* and *reset* in the sensitivity list of FF **process** blocks

```
process(clk, reset)
   begin
   if ( reset = '1' ) then
      x_reg <= '0';
   elsif (rising_edge(clk)) then
      x_reg <= x_next;
   end if;
end process;
```

The synthesis tools looks for 'clues' in your VHDL code for constructs that describe FFs, and then *infers* them during synthesis

More importantly, YOU always want to be sure where these FFs are inferred!!!

Golden Rules of FF Process Blocks

- **Rule 2:** Always use this template: ***if*** *(reset = '1') ...* ***elsif*** *(rising_edge(clk)) ...*

```
process(clk, reset)
   begin
   if ( reset = '1' ) then
      x_reg <= '0';
   elsif ( rising_edge(clk) ) then
      x_reg <= x_next;
   end if;
end process;
```

The ONLY exception is when you want a **synchronous** reset, i.e., when reset of the FFs ONLY occurs on the rising edge of *clk* (NOTE: NEVER USE BOTH TYPES)

```
process(clk, reset)
   begin
   if ( rising_edge(clk) ) then
      if ( reset = '1' ) then
         x_reg <= '0';
      else
         x_reg <= x_next; ...
```

Golden Rules of FF Process Blocks

- **Rule 3:** Never include anything except **signal assignment** in the ***if-elsif*** statements

```
process(clk, reset)
   begin
   if ( reset = '1' ) then
      x_reg <= '0';
   elsif ( rising_edge(clk) ) then
      x_reg <= x_next;
   end if;
end process;
```

Define ALL combinational functions **OUTSIDE** the FF **process** block

VHDL allows a lot of flexibility w.r.t. describing circuits, and there are many, many ways that you can write code that will result in unexpected behavior

Although the labs will task you on writing VHDL and then inspecting the schematics produced by the synthesis tool, you will not be able to do this very often in practice

Even moderately complex designs make this type of inspection untenable

Consequences of Violating the Golden Rules of FF Process Blocks

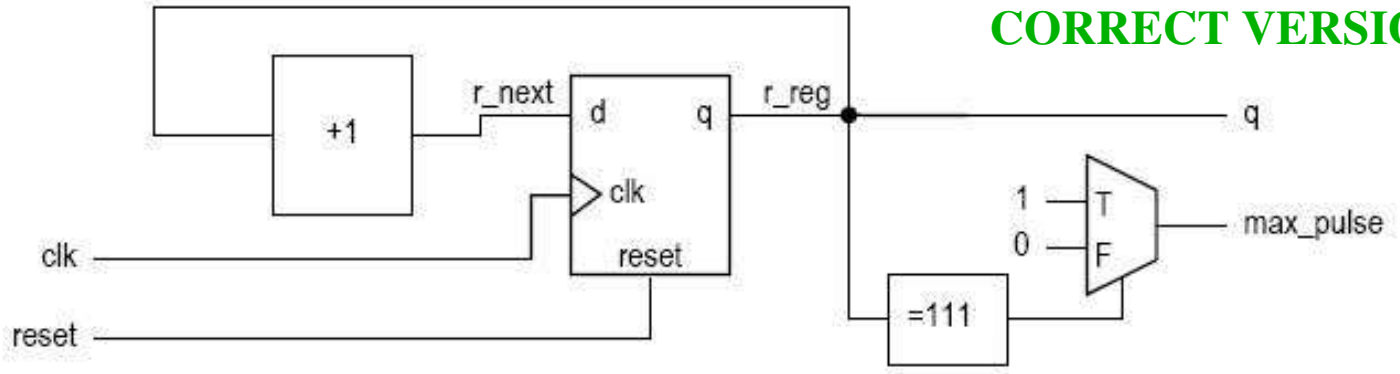
Violating **golden rule 3** is the most common

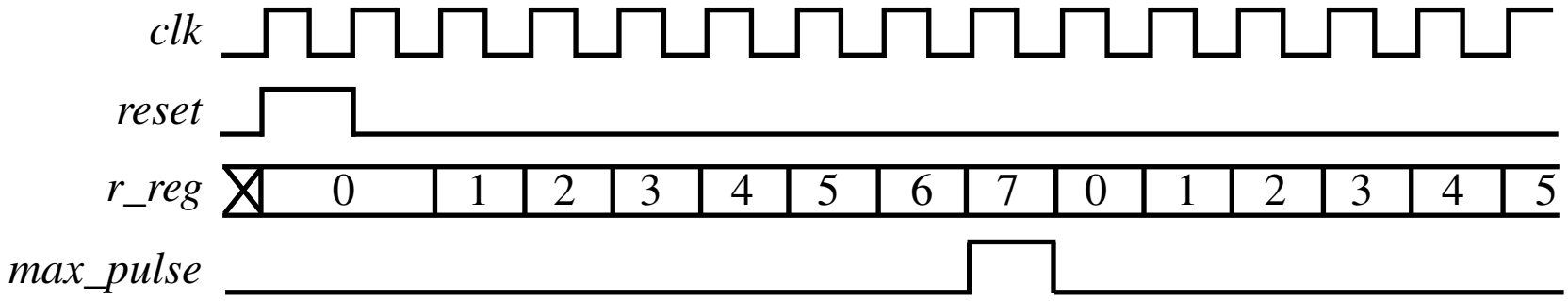
Here's the **correct** description of a circuit that generates a pulse every 8 clock cycles, which assumes *max_pulse* is defined in the **entity** as a std_logic **out** signal

```
architecture beh of pulse_cir is
   signal r_reg, r_next: unsigned(2 downto 0);
   begin
   process(clk, reset)
      begin
      if ( reset = '1' ) then
         r_reg <= (others=>'0');
      elsif ( rising_edge(clk) ) then
         r_reg <= r_next;
      end if;
   end process;
   r_next <= r_reg + 1;
   max_pulse <= '1' when r_reg = "111" else '0';
end beh;
```

Consequences of Violating the Golden Rules of FF Process Blocks

This synthesizes to the following correct version of the circuit





The synthesis tool predictably (and correctly) interprets both the sequential and combinational components of the design

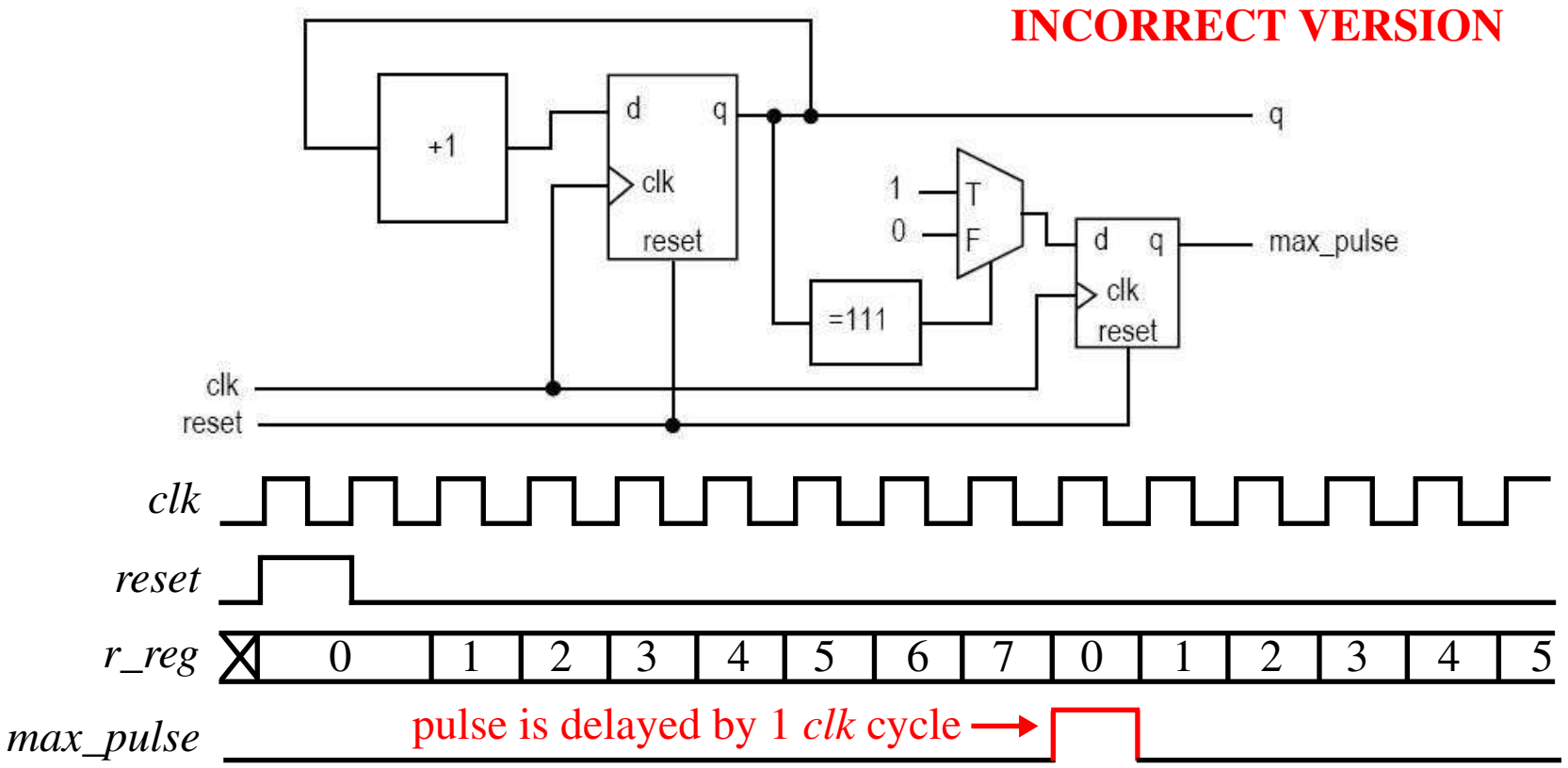
Consequences of Violating the Golden Rules of FF Process Blocks

If, on the other hand, you try to place the *max_pulse* code inside the **process** block:

```
architecture beh of pulse_cir_INCORRECT is
   signal r_reg, r_next: unsigned(2 downto 0);
   begin
   process(clk, reset)
      begin
      if ( reset = '1' ) then
         r_reg <= (others=>'0');
      elsif ( rising_edge(clk) ) then
         r_reg <= r_next;
         if (r_reg = "111") then
            max_pulse <= '1';
         else
            max_pulse <= '0';
         end if;
      end if;
   end process;
   r_next <= r_reg + 1; end pulse_cir_INCORRECT;
```

Consequences of Violating the Golden Rules of FF Process Blocks

The synthesis tool synthesizes this INCORRECT version of the circuit:



An additional FF for *max_pulse* is inferred because *max_pulse* is not assigned to **under all possible conditions** (it is in the *elsif* branch), and the pulse is delayed!

There are ways to fix this problem, but the best solution is 'DON'T DO THIS'

Register Transfer Methodology (RTL)

We typically use **algorithms** to accomplish complex tasks

Although it is common to execute algorithms on a GPU, a hardware implementation is sometimes needed because of power and performance constraints

RT methodology is a design process that describes system operation by a sequence of data transfers and manipulations among **registers**

This methodology supports sequential execution semantics used by microprocessors to execute a program

Consider an algorithm that computes the sum of 4 numbers, divides by 8 and rounds the result to the nearest integer

```
size = 4;
sum = 0;
for i in (0 to size-1) do
    { sum = sum + a(i); }
```

Register Transfer Methodology (RTL)

```
q = sum/8;
r = sum rem 8;
if (r > 3)
   { q = q + 1; }
outp = q;
```

Characteristics of an algorithm:

- Algorithms use **variables**, memory locations with a symbolic addresses, to store intermediate results
- Algorithms are executed sequentially and the order of the steps is important

One approach is to convert **sequential execution** into a **structural data flow**, where the sequence is embedded in the 'flow of data'

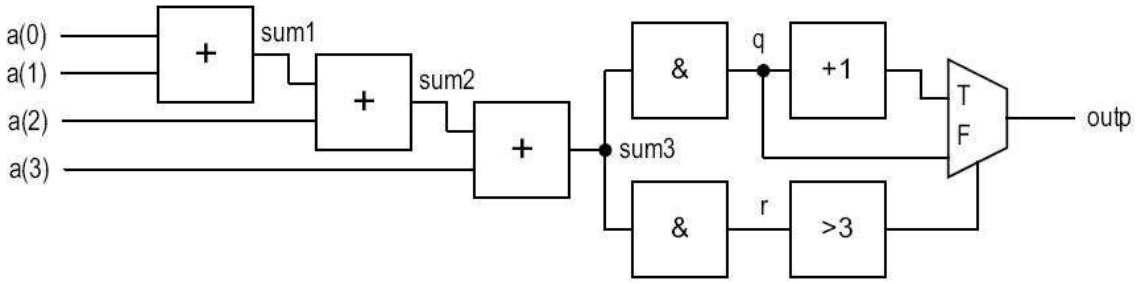
This is accomplished by mapping an algorithm into a system of *cascading hardware blocks*, where each block represents a statement in the algorithm

Register Transfer Methodology (RTL)

For example, the previous algorithm can be **unrolled** into a data flow diagram

```
sum <= 0;
sum0 <= a(0);
sum1 <= sum0 + a(1);
sum2 <= sum1 + a(2);
sum3 <= sum2 + a(3);
q <= "000" & sum3(8 downto 3);
r <= "00000" & sum3(2 downto 0);
outp <= q + 1 when (r > 3) else q;
```

Note that this is very different from the algorithm -- the circuit is strictly combinational with NO memory elements



The *structural data flow* model can only be applied to small tasks and is not flexible

Register Transfer Methodology (RTL)

Register Transfer Methodology introduces hardware that *matches* the variable and sequential execution model

- Registers are used to store intermediate data (model symbolic variables)
- A control path (FSM) is used to specify the order of register operations
- A data path is added to implement the operations (FSMD)

The basic action in RT methodology is the *register transfer operation*:

$$r_{\text{dest}} \leftarrow f(r_{\text{src1}}, r_{\text{src2}}, \ldots, r_{\text{src3}})$$

The function *f* uses the contents of the source registers, plus external inputs in some cases

Difference between an algorithm and an RT register is the implicit embedding of *clk*

- At the rising edge of the clock, the outputs of registers r_{src1}, r_{src2} become available
- The outputs drive the inputs of a combinational circuit that represents *f( )*
- At the **next rising edge** of the clock, the result is stored into r_{dest}

FSMD

The function *f()* can be any expression that is representable by a combinational circuit

$r \leftarrow 1$
$r \leftarrow r$
$r0 \leftarrow r1$
$n \leftarrow n - 1$
$y \leftarrow a \oplus b \oplus c \oplus d$
$s \leftarrow a^2 + b^2$

Note that we will continue to use the notation *_reg* and *_next* for the current output and next input of a register

The notation

$r_1 \leftarrow r_1 + r_2$

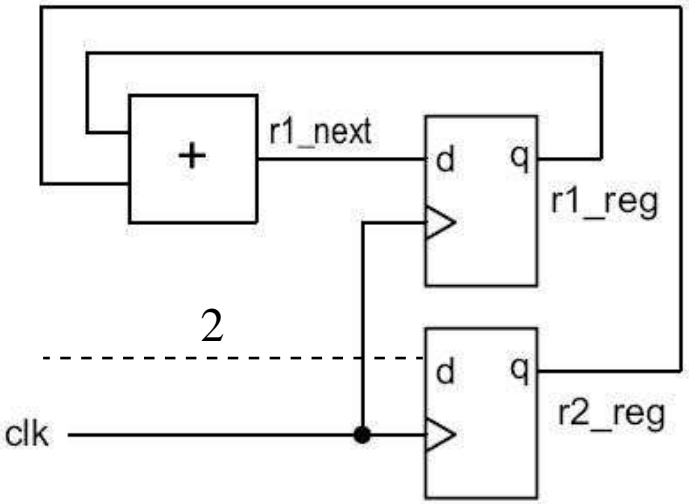
is translated as

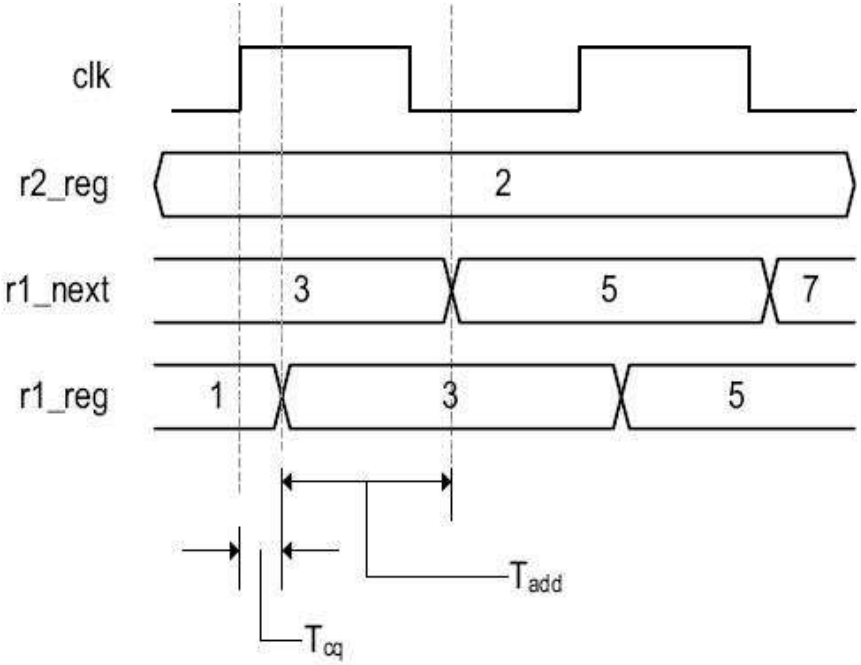
```
r1_next <= r1_reg + r2_reg;
r1_reg <= r1_next; -- on the next rising edge of clk
                   -- this inside the FF process block
```

FSMD

Be sure to study this carefully because it is heavily used in digital systems

$$r \leftarrow r1 + r2$$





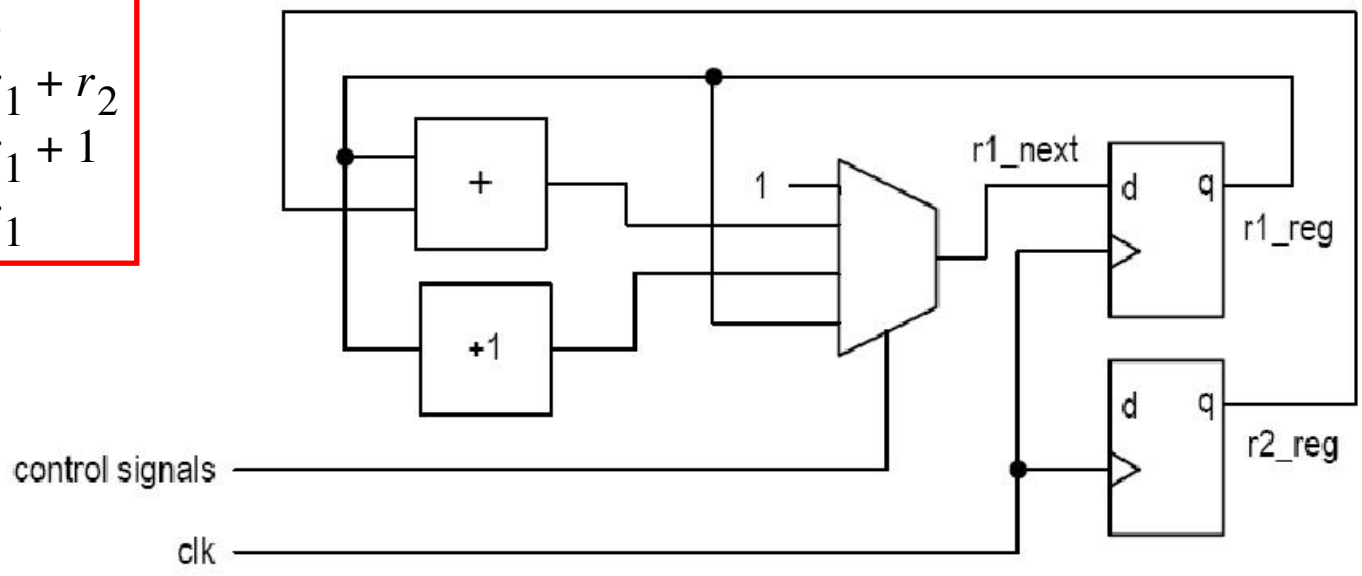
Multiple RT operations

An algorithm consists of many steps and a *destination* register my be loaded with different values over time

FSMD

Consider the following sequence of operations

$r_1 \leftarrow 1$
$r_1 \leftarrow r_1 + r_2$
$r_1 \leftarrow r_1 + 1$
$r_1 \leftarrow r_1$

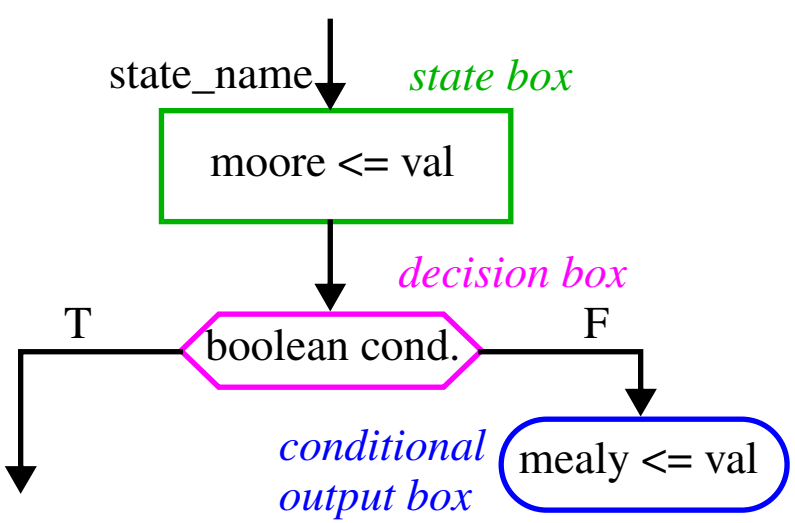


Since r_1 is the destination of multiple operations, we need a MUX to route the proper value to its input

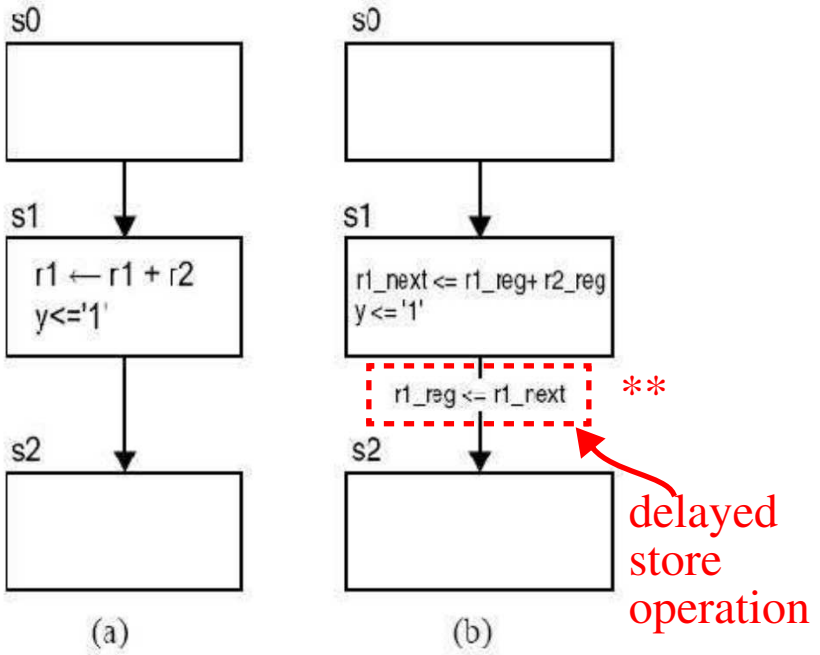
An FSM is used to drive the *control signals* so that the sequence of operations are carried out in the order given

FSMD and ASMD

An extended ASM chart known as **ASMD** (ASM with datapath) chart can be used to represent the FSMD



ASMD



State transitions and register updates occur at the same time on the rising edge of the clk

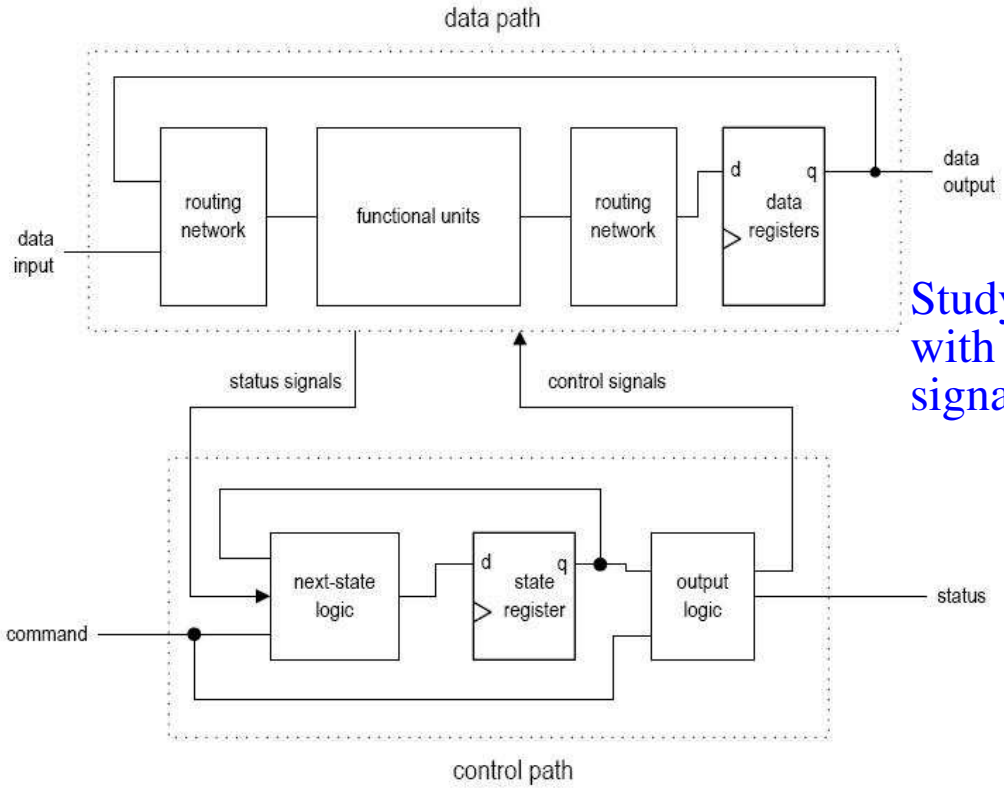
DELAYED STORE: The new value of r_1 is only available when the FSM enters the s_2 state

FSMD

NOTE: When a register is NOT being updated with a new value, it is assumed that it maintains its current value, i.e.,

$r_1 \leftarrow r_1$ These actions are NOT shown in the ASMD/state chart

Conceptual block diagram of an FSMD



Data Path

Regular sequential circuit

Study and become familiar with the input/output signals of both modules

Control Path

Random sequential circuit

FSMD Design Examples

Consider a repetitive addition multiplier

Basic algorithm: 7*5 = 7+7+7+7+7

```
if (a_in=0 or b_in=0) then
    { r = 0; }
else
    {
    a = a_in;
    n = b_in;
    r = 0;
    while (n != 0)
        {
        r = r + a;
        n = n - 1;
        }
    }
    return(r);
```

FSMD Design Examples

This code is a better match to an ASMD because ASMD does not have a **loop** construct

```
    if (a_in = 0 or b_in = 0) then
       { r = 0; }
    else
       {
       a = a_in;
       n = b_in;
       r = 0;
 op:   r = r + a;
       n = n - 1;
       if (n = 0) then
          { goto stop; }
       else
          { goto op; }
       }
stop: return(r);
```

FSMD Design Examples

To implement this in hardware, we must first define the I/O signals

- *a_in*, *b_in*: 8-bit unsigned input
- *clk*, *reset*: 1-bit input
- *start*: 1-bit command input
- *r*: 16-bit unsigned output
- *ready*: 1-bit status output -- asserted when unit has completed and is ready again

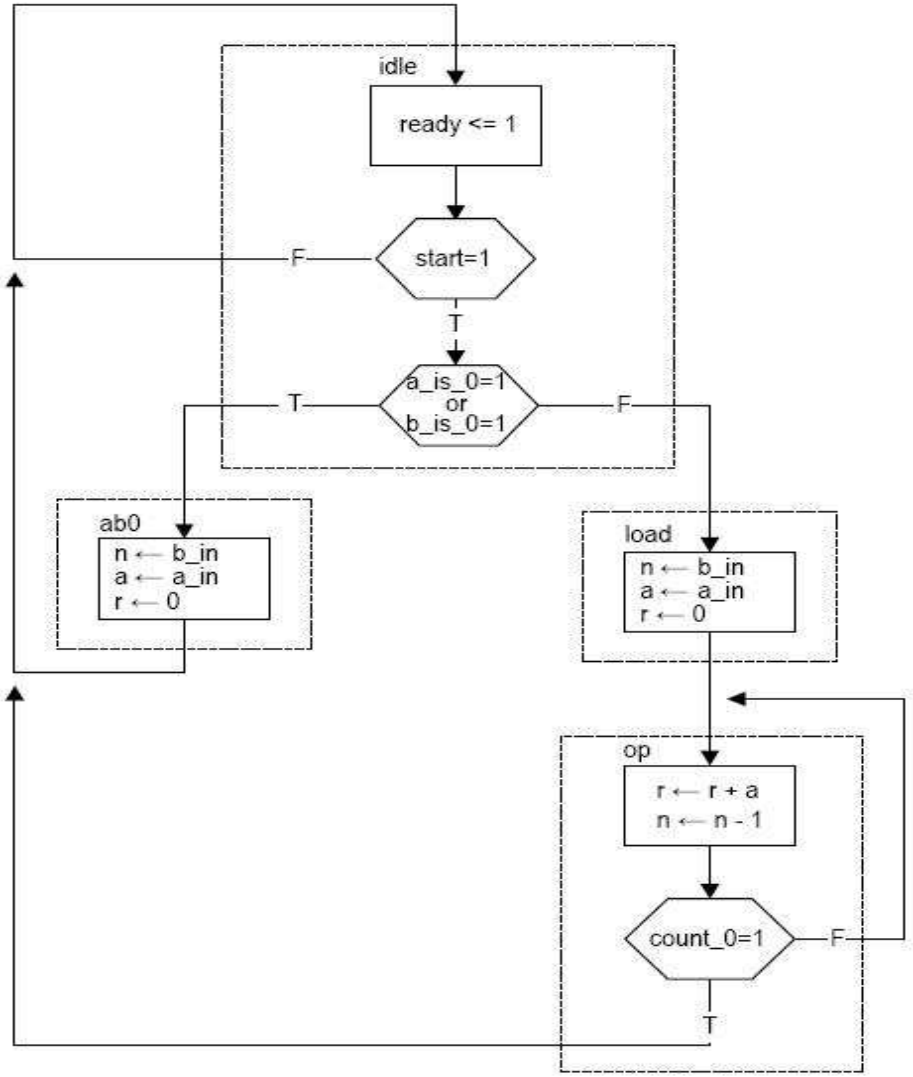
The *start* and *ready* signals are added to support sequential operation

When this unit is embedded in a larger design, and the main system wants to perform multiplication

- It checks *ready*
- If '1', it places inputs on *a_in* and *b_in* and asserts the *start* signal

FSMD Design Examples

The ASMD uses *a*, *n* and *r* data registers to emulate the three variables



FSMD Design Examples

With the ASMD chart available, we can refine the original block diagram

We first divide the system into a *data path* and a *control path*

For the control path, the input signals are *start*, *a_is_0*, *b_is_0* and *count_0* -- the first is an external signal, the latter three are status signals from the data path

These signals constitute the inputs to the FSM and are used in the *decision boxes*

The output of the control path are *ready* and control signals that specify the RT operations of the data path

In this example, we use the state register as the output control signals

Visualizing the data path can be accomplished by doing the following:

- List all RT operations
- Group RT operation according to the destination register
- Add combinational circuit/mux
- Add status circuits

FSMD Design Examples

For example

- RT operation with the *r* register

  $r \leftarrow r$ (in the idle state)
  $r \leftarrow 0$ (in the load and ab0 states)
  $r \leftarrow r + a$ (in the op state)

- RT operations with the *n* register

  $n \leftarrow n$ (in the idle state)
  $n \leftarrow$ b_in (in the load and ab0 state)
  $n \leftarrow n - 1$ (in the op state)

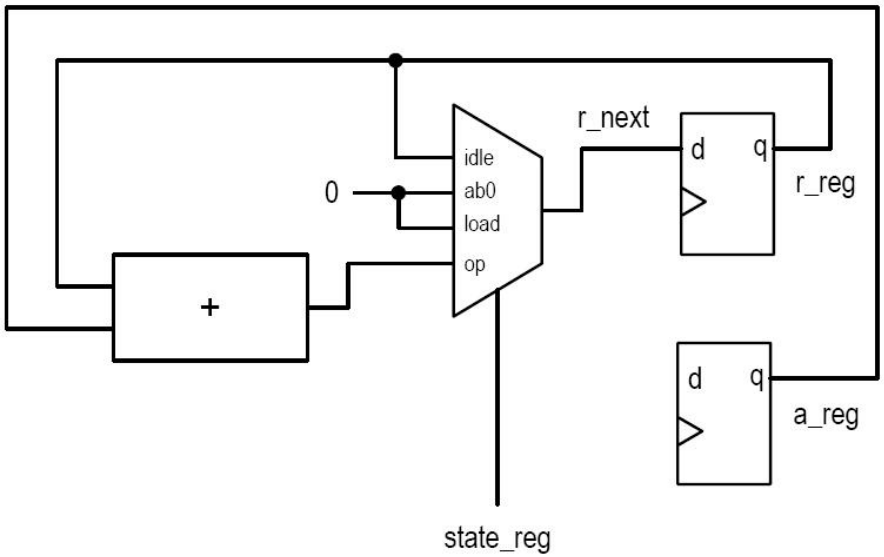
- RT operations with the *a* register

  $a \leftarrow a$ (in the idle and op states)
  $a \leftarrow$ a_in (in the load and ab0 states)

Note that the **default** operations MUST be included to build the proper data path

FSMD Design Examples

Let's consider the circuit associated with the *r* register



The three possible sources, 0, *r* and *r*+*a* are selected using a MUX

The select signals are labeled symbolically with the state names

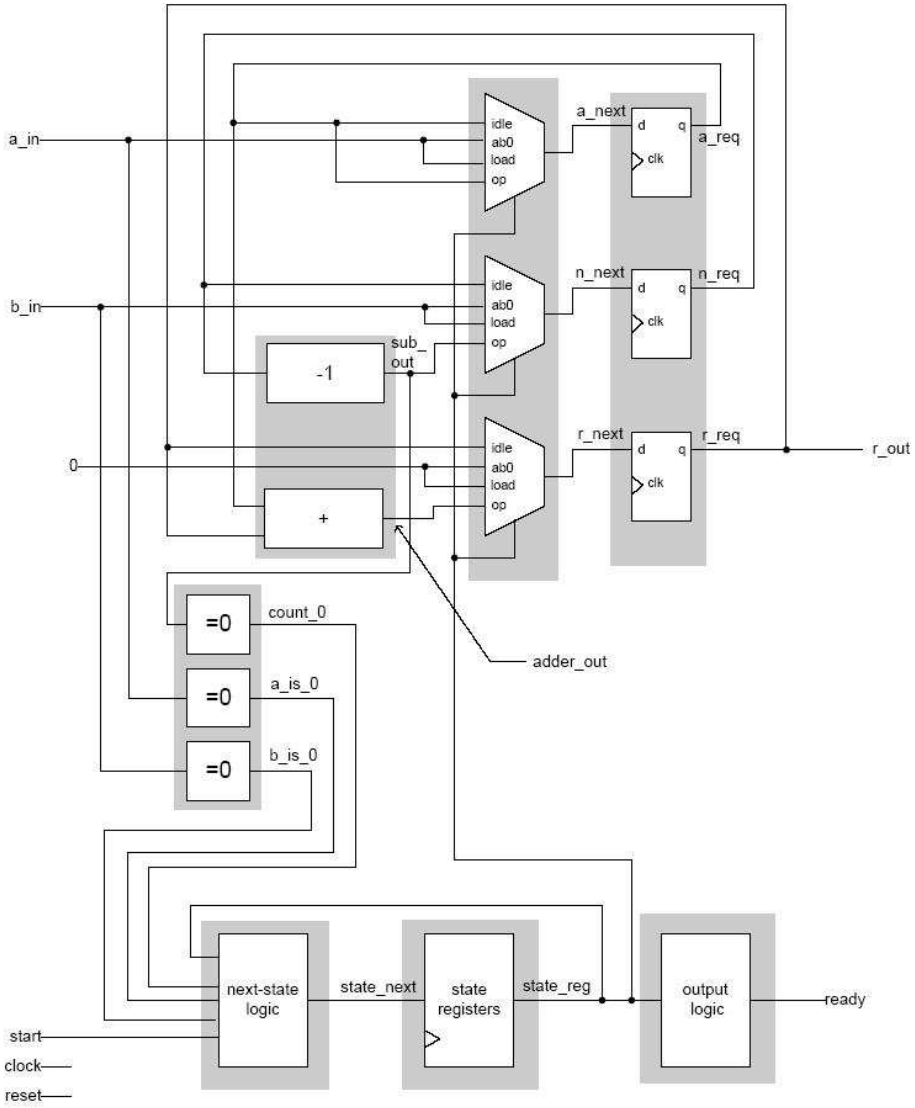
The routing is consistent with what is given on the previous slide

We can repeat this process for the other two registers and combine them

The status signals are implemented using three comparators

FSMD Design Examples

The entire (un-optimized) control and data path



FSMD Design Examples

```
architecture two_seg_arch of seq_mult is
   constant WIDTH: integer := 8;
   type state_type is (idle, ab0, load, op);
   signal state_reg, state_next: state_type;
   signal a_reg, a_next: unsigned(WIDTH-1 downto 0);
   signal n_reg, n_next: unsigned(WIDTH-1 downto 0);
   signal r_reg, r_next: unsigned(2*WIDTH-1 downto 0);
   begin

   -- state and data register
   process(clk, reset)
      begin
      if (reset = '1') then
         state_reg <= idle;
         a_reg <= (others => '0');
         n_reg <= (others => '0');
         r_reg <= (others => '0');
```

Two Segment VHDL Descriptions of FSMDs

```
        elsif (clk'event and clk = '1') then
            state_reg <= state_next;
            a_reg <= a_next;
            n_reg <= n_next;
            r_reg <= r_next;
        end if;
    end process;

    -- combinational circuit
    process(start, state_reg, a_reg, n_reg, r_reg, a_in,
        b_in, n_next)
        begin
        state_next <= state_reg;
        a_next <= a_reg;
        n_next <= n_reg;
        r_next <= r_reg;
        ready <='0';
```

Two Segment VHDL Descriptions of FSMDs

```
case state_reg is
   when idle =>
      if (start = '1') then
         if (a_in = "00000000" or
               b_in = "00000000") then
            state_next <= ab0;
         else
            state_next <= load;
         end if;
      end if;
      ready <= '1';

   when ab0 =>
      a_next <= unsigned(a_in);
      n_next <= unsigned(b_in);
      r_next <= (others => '0');
      state_next <= idle;
```

Two Segment VHDL Descriptions of FSMDs

```
            when load =>
               a_next <= unsigned(a_in);
               n_next <= unsigned(b_in);
               r_next <= (others => '0');
               state_next <= op;

            when op =>
               n_next <= n_reg - 1;
               r_next <= ("00000000" & a_reg) + r_reg;
               if (n_next = "00000000") then
                  state_next <= idle;
               end if;
         end case;
      end process;

      r <= std_logic_vector(r_reg);
   end two_seg_arch;
```