

합성과 시뮬레이션을 위한

# HDL 의 기초



## 목표



- 하드웨어 기술 언어 (**HDL-Hardware Description Languages**) 에 의한 디지털 회로 설계에 대한 고찰
- **HDL**에 의한 디지털 회로의 시뮬레이션 및 합성의 기본 개념 이해
- **VHDL** 와 **Verilog** 등 **HDL**의 기본적인 형태
- 예제를 통한 **HDL**에 의한 디지털 회로 기술



## HDL이란 무엇인가?



### ■ HDL의 종류

◆ VHDL (~93%), Verilog(~35%)

### ■ 전자회로의 **Modeling**

■ 전자회로 입력(**Design Entry**)언어

■ 회로 테스트용 언어

■ 회로의 네트리스트 기술 언어

■ 표준화 및 이식성(**portable**) 우수



## VHDL의 개략적인 역사



■ **1981:** 미 국방부 (**DoD**)의 지원아래 HDL에 대한 표준화 착수

■ **1983-1985:** VHDL 개발 (**Intermetrics, IBM, TI**)

■ **1987:** IEEE Standard 1076-1987 제정

■ **1991:** IEEE Standard 1164 (standard nine-valued logic) 제정



## VHDL의 역사 (계속)



- **1993: IEEE Standard 1076-1993**
- **1995: IEEE Standards 1076.3,1076.4**
- **IEEE 1076 VHDL Working Group**
  - ◆ **IEEE 1076.1 : VHDL-A Analog extension**
  - ◆ **IEEE 1076.2 : Math Package**
  - ◆ **IEEE 1076.3 : Synthesis Package (std\_logic\_1164)**
  - ◆ **IEEE 1076.4 : Timing Methodology (VITAL)**
  - ◆ **IEEE 1076.5 : Utility**
  - ◆ **IEEE P1165 : EDIF**



## Verilog의 개략적인 역사



- **1985: Verilog language and simulator developed by Gateway Automation**
- **1985-1989: Increasing use of Verilog as a Golden simulator for ASIC companies.**
- **1989: Verilog merges with Cadence**
- **1990: Open Verilog International (OVI) formed**
- **1995: IEEE standard 1364 adopted**



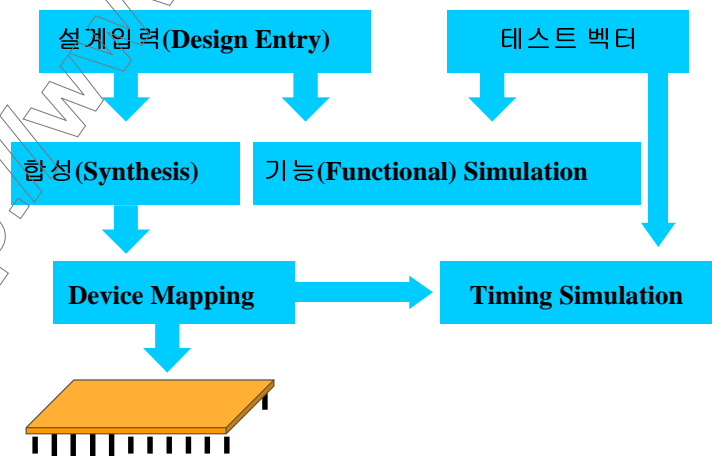
## HDL을 어디에 쓰는가?



- 다양한 추상화 수준에서 설계사양 기술
  - ◆ **Transistor** 수준에서 시스템 수준까지..
- 실제 설계 입력
  - ◆ 합성가능한 코딩
- 기능 및 타이밍 시뮬레이션
- 설계의 **Documentation**
- 다양한 **EDA** 툴 사이의 **Interface**



## 자동화된 설계 과정





## 왜 HDL을 사용 하는가?



- 설계의 생산성 향상
- Time to Market
- Design Re-Use
- 적은 인력으로 복잡도 높은 설계 (3~5명)
- 향후 Device/Tool/Tech 의 변경에 대처
- EDA 표준 (Portability, 설계 Tool -Synthesis, Test/Verify, P&R-사이의 Interface)



## 왜 HDL사용을 기피하는가?



- 배우기 까다롭다?
  - ◆ HDL만 관련한 것은 아니다. 예제들이 많이 있으니...
- tool 값이 너무 비싸다?
  - ◆ 맞는 말이긴 하지만 상황이 바뀌고 있다
  - ◆ PC-Based, \$10,000 이하
- HDL은 ASIC에 치중되어 있어서 PLD나 FPGA에 맞지 않는다?
  - ◆ HDL은 범용 언어이다
- PLD/FPGA 응용 전문가가 드물다?
  - ◆ Device vendor FAE들이 만족스럽지 않을지 모르지만 누구나 전문가가 될 수 있다



## Verilog 로 할까? VHDL로 할까?



- Verilog 가 시뮬레이션 모델에 유용?
  - ◆ VHDL에는 VITAL이...
- VHDL은 다양한 수준의 기술이 가능하며 설계관리 용이
- VHDL, Verilog 모두 배우긴 쉽지만 완전히 터득하기 어렵긴 마찬가지
  - ◆ 경험이 중요!
- 장단점에 대한 논의는 끝이 없다
  - ◆ 언어를 말하기 전에 사용할 수 있는 틀은?



## PLD 전용 언어들은?



- 범용 HDL(VHDL, Verilog)에 비해 배우기 쉽고 간단하다.
- PLD 전용이다.
  - ◆ built-in device macros, pin and node numbers, 등등...
  - ◆ re-targetting 불가
- 실제 회로의 합성용일 뿐이다.
  - ◆ 시뮬레이션과 Documentation용은 아니다
- PLD 의 크기, 핀 할당등에 적절하다.
- 툴 가격이 저렴하거나 무료이다.



## 추상화(Abstraction) 수준



Behavior

Performance Specifications

Test Benches

Sequential Descriptions

State Machines

Dataflow

Register Transfers

Selected Assignments

Arithmetic Operations

Structure

Boolean Equations

Hierarchy

Physical Information

Level of Abstraction

Synthesizable



## 추상화 수준(계속)



PLD  
전용언어



Verilog



VHDL

Level of Abstraction



## 추상화 수준 (Behavior Style)



### ■ Behavior:

- ◆ Sequential statements
- ◆ implied registers (like a software programming language)

*/\* Programming Language \*/*

```
int a, b, c;
a = b;
b = c;
```

*-- VHDL*  
*-- testbench clock generator*  
 clock: process  
 constant PERIOD := 100 ns;  
 variable c: std\_ulogic := '0';  
 begin  
 wait for PERIOD / 2;  
 c := not c;  
 Clk <= c;  
 end process;

*-- VHDL*  
*-- Register inference*  
 clock: process(Clk)  
 begin  
 if (Clk='1' AND Clk'EVENT) then  
 q <= d;  
 end if;  
 end process;



## 추상화 수준 (Dataflow)



### ■ Dataflow:

- ◆ Concurrent assignments
- ◆ Explicit registers (like a PLD language)

```
signal a,b,c,d,q;
begin
  c <= a and b;
  b <= c and d;
  q <= d when (clk='1' and clk'event);
end;
```





## 추상화 수준 (Structure)



### ■ Structure:

#### ◆ Connected components (like a netlist)

```
component nand2
  port ( i0, i1 : std_logic; o : out std_logic);
end component;
signal a, b, c, d;
begin
  u1 : nand2 port map (i0 => a, i1 => b, o => c);
  u2 : nand2 port map (i0 => c, i1 => d, o => a);
end;
```

## Lab. Exercise



## Switch-Level Simulation

# VHDL 개요



**VHDL Design Units**  
**VHDL Types**  
**VHDL Objects**  
**VHDL Statements**

# VHDL Design Units



**Entity**  
**Architecture**  
**Configuration**  
**Package**



## VHDL Design Units



Package

```
package my_common is
  -- Common declarations
end my_common;
```

Entity

```
entity my_design is
  port(...);
end my_design;
```

Architecture

```
architecture my_arch of my_design is
  begin
  .....
end my_arch;
```

Configuration

```
configuration this_build of my_design is
  -- Configuration (binding) information
end this_build;
```



## Simple Design



Standard Libraries

Package

Package

Entity/Architecture

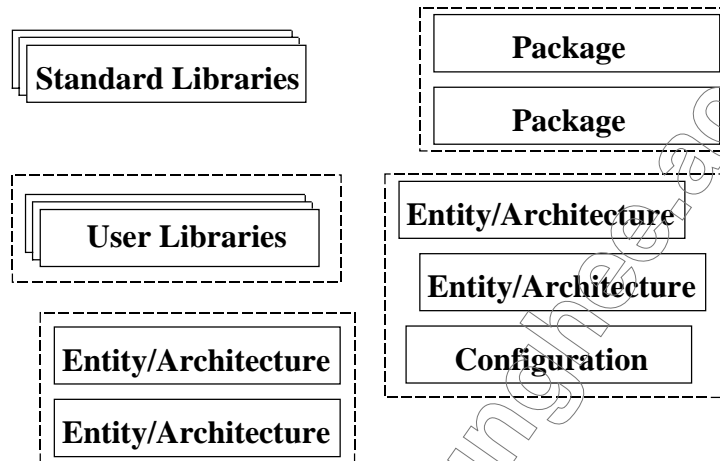
Entity/Architecture

Entity/Architecture

VHDL Source File



## More Complex Design



## Entity and Architecture



- **Basic Building Blocks**
- **Entity : “Black Box”**
  - ◆ **Boundary of Logic Block**
  - ◆ **Declare Port and Generics**
  - ◆ **System Interface**
- **Architecture**
  - ◆ **Contents of Logic Block**
  - ◆ **Structural, Dataflow, Behavioral**

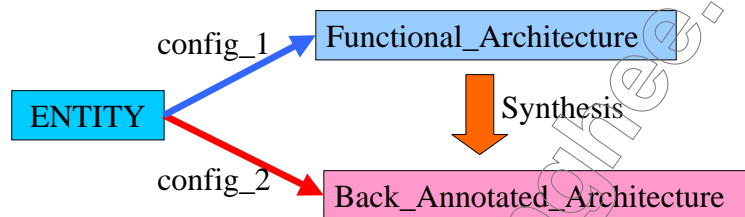


## Configuration



### ■ Configuration

- ◆ Configure each component instantiation with Different Entity or Architecture



## Entity Example



```
entity compare is
  port( A, B: in bit_vector(0 to 7);
        EQ: out bit);
end compare;
```

- Specifies entity name : compare
- Specifies type and direction (mode) for all ports : in, out, inout, buffer
- Provides a complete specification for the interface : type and bit width



## Architecture Example



```
architecture compare1 of compare is  
begin
```

```
    EQ <= '1' when (A = B) else '0';
```

```
end compare1;
```

- ARCHITECTURE is bound to an entity by name
- ARCHITECTURE defines the entity contents
- ARCHITECTURE may contain references to lower-level entities (hierarchy)

## Lab. Exercise



### 8-Bit Compare

# VHDL Data Types



**Scalar Types**  
**Composite Types**



## Data Types (VHDL 자료형)



- All signals have a type
- Type checking is strict !
- Each type has a defined set of valid operations
- Each type has a defined set of values
- Types are user-extendable



## VHDL Data Types (분류)



- **Scalar Types**
  - ◆ Enumeration Type
  - ◆ Integer Type
  - ◆ Floating-Point Types
  - ◆ Physical Types
- **Composite Types**
  - ◆ Array Types
  - ◆ Record Types
- **Access Types**
- **File Types**



## Common Types



- **Bit (or std\_logic, defined by IEEE 1164)**
- **Bit\_vector (or std\_logic\_vector)**
- **Boolean**
- **Integer**
- **Character**
- **String**
- **Enumerated type**





## More Types



- Real (floating point)
- Physical (time, current, etc.)
- Arrays (single- and multi-dimensional, constrained or unconstrained)
- Records
- Access types (pointers)
- File types



## Enumerated Types



- Many common VHDL types are actually enumerated types:
  - ◆ Boolean (TRUE, FALSE);
  - ◆ Bit (1,0)
  - ◆ char
- User extendable:

```
type Boolean is ( FALSE, TRUE);  
type bit is ('0', '1');  
type std_ulogic is ('U','X','0','1','Z','W','L','H','-');  
type states is ( IDLE, RECEIVE, SEND);
```



## Enumerated Type Encoding



- One Hot Encoding (State Machine)
- Binary Encoding

	bit2	bit1	bit0
IDLE	-	-	1
RECEIVE	-	1	-
SEND	1	-	-

**OneHot**

	bit1	bit0
IDLE	0	0
RECEIVE	0	1
SEND	1	0

**Binary**



## Integer and Floating-Point Type



- Integer Type (Synthesizable)

```

type integer is range -2147483647 to 2147483647; -- VHDL Predefined
-- 32-bit signed integer

type my_integer is range 0 to 15; -- 4-bits
signal count : my_integer;
signal count : integer range 0 to 15; -- count(0) : LSB, count(3) : MSB

```

- Floating-Point Type (NOT Synthesizable)

```

type real is range -1.0E38 to 1.0E38; -- VHDL Predefined
type my_real is range 0.0 to 1.0; -- VHDL Predefined

```



## Physical types



### ■ Represent relations between quantities

- ◆ NOT Synthesizable

### ■ Example : “time”

```
type time is range -2147483647 to 2147483647
units
  fs;
  ps = 1000 fs;
  ns = 1000 ps;
  us = 1000 ns;
  ms = 1000 us;
  sec = 1000 ms;
  min = 60 sec;
  hr = 60 min;
end units;
```



## Array Types



### ■ Synthesizable

### ■ Array Example (Unconstraint)

```
type std_ulogic_vector is array ( NATURAL RANGE <> ) OF std_ulogic;
type UNSIGNED is array (NATURAL range <> ) of BIT;
type SIGNED is array (NATURAL range <> ) of BIT;
type matrix is array (natural range <>, natural range <>) of bit;
```

### ■ Array Example (Constraint)

```
type byte is array (7 downto 0) of bit;
```



## Record Type



### ■ Example

```
type month_name is (jan,feb,mar,apr,may,jun,jul,aug,sep,oct,nov,dev);
type date is
record
  day : integer range 1 to 31;
  month : month_name;
  year : integer range 0 to 4000;
end record;
```

```
constant my_birthday : date := (21, nov, 1963);
```

```
SIGNAL my_birthday : date;

my_birthday.year <= 1963;
my_birthday.month <= nov;
my_birthday.day <= 21; -- 5-bit
-- my_birthday(0) : LSB
-- my_birthday(4) : MSB
```



## Subtypes and Type Conversion



### ■ Type checking is strict !

### ■ Type with constraint

```
subtype <subtype_name> is <base_type> [<constraint>];
```

### ■ Type Conversions

```
<type>(<expressions>)
```



## VHDL type checking is strict !



```
type big_integer is range 0 to 1000;  
type small_integer is range 0 to 7;  
  
signal intermediate : small_integer;  
signal final : big_integer;  
  
final <= intermediate * 5; -- type mismatch error
```



or



```
type big_integer is range 0 to 1000;  
subtype small_integer is big_integer range 0 to 7;  
  
signal intermediate : small_integer;  
signal final : big_integer;  
  
final <= intermediate * 5; -- Same Base type
```

```
type big_integer is range 0 to 1000;  
type small_integer is big_integer range 0 to 7;  
  
signal intermediate : small_integer;  
signal final : big_integer;  
  
final <= big_integer(intermediate * 5); -- Type Conversion
```



## IEEE 1076 Predefined Types



```
type bit is ('0', '1');  
type bit_vector is array (integer range <>) of bit;  
type integer is range MININT to MAXINT;  
subtype positive is integer range 1 to MAXINT;  
subtype natural is integer range 0 to MAXINT;  
type boolean is (TRUE, FALSE);
```



## IEEE 1164 Predefined Types



- `std_logic`, `std_logic_vector` (resolved)
- `std_ulogic`, `std_ulogic_vector` (unresolved)
- Nine valued enumerated type
- Used as replacements for `bit`, `bit_vector`
- Used as the standard data types for system interfaces



## IEEE 1164 Predefined Types



```
TYPE std_ulogic IS ('U', -- Uninitialized
                  'X', -- Forcing Unknown
                  '0', -- Forcing 0
                  '1', -- Forcing 1
                  'Z', -- High Impedance
                  'W', -- Weak Unknown
                  'L', -- Weak 0
                  'H', -- Weak 1
                  '-' -- Don't care
                  );
type std_ulogic_vector is array (natural range <>) of std_ulogic;
subtype std_logic is resolution_func std_ulogic;
type std_logic_vector is (natural range <>) of std_logic;
subtype X01Z is resolution_func std_ulogic range 'X' to 'Z';
```



## Type Conversion Example



```
package my_type is
  type big_integer is range 0 to 1023;
  subtype small_integer is big_integer range 0 to 7;
end package;

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.my_type.all;

entity subtype_test is
  port ( a : in small_integer;
        b : in small_integer;
        c : out std_logic_vector(9 downto 0) );
end subtype_test;
architecture behave of subtype_test is
  signal t_int : big_integer;
begin
  t_int <= a + b; -- for subtype, type casting not needed !
  c <= std_logic_vector(to_unsigned(natural(t_int), 10)); -- type casting type conversion
end;
```



## Type Conversion/Type Casting



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

type big_integer is range 0 to 1023; -- 10-bits
signal t_int : big_integer;
signal c : out std_logic_vector(9 downto 0);
```

```
c <= std_logic_vector( to_unsigned( natural( t_int ) , 10 ) );
```

Type casting  
"unsigned" to "std\_logic\_vector"

Type conversion  
"natural" to "unsigned"

Type casting  
"integer" to "natural"

big\_integer

# VHDL Objects



**SIGNAL**  
**CONSTANT**  
**VARIABLE**  
**PORT**  
**Loop Variable**  
**Generic**



## SIGNAL



- Wire in a Logic Circuit
- CANNOT be declared in Sequential Blocks; “Function”, “Procedure” and “Process”
- Range restriction
- Initial value (but ignored by synthesis)
- Signal has Attribute; 'EVENT
- Assignment ( $\leftarrow$ ) is scheduled after delta delay





## VARIABLE



- CANNOT be declared in the dataflow(concurrent block) or Package
- Declared in Process, Function, and Procedure
- Initial value (but ignored by synthesis)
- Assignment (:=) is immediate



## CONSTANT



- CANNOT be assigned after declaration
- Initialization is required

```
constant ZEE_8 : std_logic_vector (0 to 7) := "ZZZZZZZZ";
```



## PORT



- **Interface terminal of Entity**
- **Normal signal with Direction Mode restriction:**
  - ◆ **IN, OUT : unidirectional**  
`s_out <= s_in1 + s_in0;`
  - ◆ **INOUT : bidirectional**  
`s_inout <= s_in1 + s_in0;`  
`s_out <= s_inout + s_in1;`
  - ◆ **BUFFER : bidirectional**  
`buff <= buff + s_in;`
- **CAN have initial value**



## GENERIC



- **Property of Entity**
- **Definition size of interface and Delay,etc.**

```
entity counter is
  generic ( size : integer := 8;
            delay : time := 10 ns; )
  port ( clk, reset : IN std_logic;
         count : OUT std_logic_vector( size-1 downto 0 ) );
end counter;
```

```
inst_1 : counter
  generic map ( size => 16, delay => 5 ns )
  port map ( clk=>clk, reset=>reset, count=>count);
```



## Loop Variable



- NOT declared
- Get its type and value from specified range in the iteration scheme

```
for i in 0 to ( b'LENGTH-1) loop  
  a(i) <= b(i) and ena;  
end loop;
```

## VHDL Statements



**Conditional Statements**  
**Selection Statements**  
**Loop and Generate Statements**  
**Assignment Statements**



## Conditional Statements



### ■ Dataflow

```
output_signal <=
  x when a=1 else
  y when a=2 else
  z when a=3 else
  (others=>"0");
```

### ■ Behavior

```
if a=1 then
  output_signal <= x;
elsif a=2 then
  output_signal <= y;
elsif a=3 then
  output_signal <= z;
else
  output_signal <=
  (others=>"0");
```



## Selection Statements



### ■ Dataflow

```
with sel select
  output_signal <=
    x when "0010",
    y when "0100",
    z when "1000",
    (others=>"0")
  when others;
```

### ■ Behavior

```
case sel is
  when "0010" =>
    output_signal <= x;
  when "0100" =>
    output_signal <= y;
  when "1000" =>
    output_signal <= z;
  when others =>
    output_signal <=
      (others=>"0");
end case;
```



## for / while loop statements



- Iterative/Repeative operation
- Loop statement can be only used in sequential block (behavior)

```
-- for loop : Synthesizable
signal result, input_sig : bit_vector( 0 to 5);
signal ena;
.....
for i in 0 to 5 loop
  result(i) <= input_sig(i) and ena;
end loop;
```

```
-- while loop : partially synthesizable
variable i : integer;
.....
i := 0;
while ( i<6 ) loop
  result(i) <= input_sig(i) and ena;
  i := i + 1;
end loop;
```



## for / while loop statements



- Sequential environment allow EXIT/NEXT statement in the Loop
- while loop is NOT synthesizable, if loop condition is evaluate at run-time

How many Loop ?

Synthesizer

How many bits of Hardware?

```
i := -1;
while (TRUE) loop
  i := i + 1;
  exit if ( i > 5 );
  if ( input_sig(i) = '0' ) then
    result(i) <= '0';
    next;
  end if;
  result(i) <= ena;
end loop;
```

*Synthesizable  
But NOT RECOMENDED style*



## for - generate loop



- Generate block execute concurrently
- NOT allow NEXT/EXIT statement
- Synthesizable

```
-- for-generate : Synthesizable
signal result, input_sig : bit_vector( 0 to 5);
signal ena;
.....
for i in 0 to 5 generate
    result(i) <= input_sig(i) and ena;
end generate;
```

## Lab. Exercise



### PREP #1 “for loop” example

# VHDL Assignment



## SIGNAL 과 VARIABLE



## VHDL에서 값의 할당 (assignment)



### ■ SIGNAL

- ◆ 병렬구문 (concurrent statement)

### ■ VARIABLE

- ◆ 순차구문 (sequential statement) : PROCESS

### ■ CONSTANT

- ◆ C언어의 #define, Assembly언어에서의 EQU
- ◆ 컴파일시 값이 결정 된다



## SIGNAL 과 VARIABLE의 차이



### ■ VARIABLE :

- ◆ “:=”
- ◆ 순차적인 즉시 할당(**instaneous assignment**)
- ◆ **PROCESS BLOCK**에서 **sequential statement**
- ◆ **Example:**
  - **AV := X\*Y;**
  - **BV := AV+Z;**



## SIGNAL 과 VARIABLE의 차이



### ■ SIGNAL :

- ◆ “<=”
- ◆ 지연 할당(**delayed assignment**)
- ◆ **Concurrent statement**
- ◆ **Attributes**
- ◆ **Example:**
  - **AS <= X\*Y after 2 ns;**
  - **BS <= AS + Z after 2 ns;**





# Assignment Execution



AS <= X \* Y after 2 ns;  
BS <= AS + Z after 2 ns;

	Initial	T1	T1+2	T1+4	T1+6
X	1	4	5	5	3
Y	2	2	2	3	2
AS	2	2	8	10	15
Z	0	3	2	2	2
BS	2	2	5	10	12

*Delayed Concurrent Assignment*

AV := X \* Y;  
BV := AV + Z;

	Initial	T1	T1+2	T1+4	T1+6
X	1	4	5	5	3
Y	2	2	2	3	2
AV	2	8	10	15	6
Z	0	3	2	2	2
BV	2	11	12	17	8

*Instantaneous Sequential Assignment*



# SIGNAL 과 VARIABLE 합성



```

ENTITY sigvar1 IS
PORT(
  clk : IN std_logic;
  din : IN std_logic_vector(2 DOWNTO 0);
  dout : OUT std_logic_vector(2 DOWNTO 0) );
END sigvar1;

```



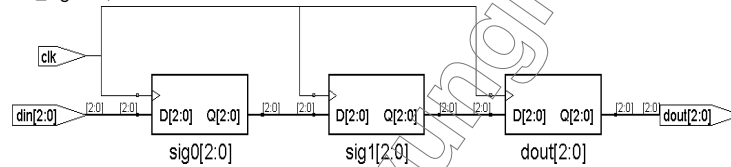
## SIGNAL 과 VARIABLE 합성 (1)



```

ARCHITECTURE a_sigvar1 OF sigvar1 IS
  SIGNAL sig0, sig1 : std_logic_vector(2 DOWNTO 0);
BEGIN
  PROCESS (clk)
  BEGIN
    IF clk='1' AND clk'EVENT THEN
      sig0 <= din;
      sig1 <= sig0;
      dout <= sig1;
    END IF;
  END PROCESS;
END a_sigvar1;

```



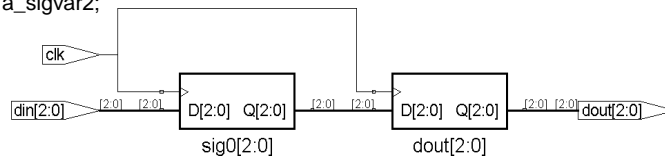
## SIGNAL 과 VARIABLE 합성 (2)



```

ARCHITECTURE a_sigvar2 OF sigvar2 IS
  SIGNAL sig0, sig1 : std_logic_vector(2 DOWNTO 0);
BEGIN
  PROCESS (clk)
  BEGIN
    IF clk='1' AND clk'EVENT THEN
      sig0 <= din; -- This statement infer F/F
      sig1 <= sig0; -- This statement infer another F/F
    END IF;
  END PROCESS;
  dout <= sig1; -- "sig1" DOES NOTHING. Just Connected to "dout"
END a_sigvar2;

```





## SIGNAL 과 VARIABLE 합성 (2-1)



```
ARCHITECTURE a_sigvar2_1 OF sigvar2_1 IS
  SIGNAL sig0, sig1 : std_logic_vector(2 DOWNTO 0);
BEGIN
  PROCESS (clk)
  BEGIN
    IF clk='1' AND clk'EVENT THEN
      sig0 <= din; -- This Statement infers F/F
    END IF;
  END PROCESS;

  sig1 <= sig0; -- This is Concurrent assignment
  dout <= sig1; -- "sig0" and "sig1" is does NOTHING! Just Connected

END a_sigvar2_1;
```



## SIGNAL 과 VARIABLE 합성 (Err)



```
ARCHITECTURE a_sigvar2err OF sigvar2err IS
BEGIN
  PROCESS (clk)
  VARIABLE sig0, sig1 : std_logic_vector(2 DOWNTO 0);
  BEGIN
    IF clk='1' AND clk'EVENT THEN
      sig0 <= din;
      sig1 <= sig0; -- This statement causes ERROR "sig1 must be SIGNAL"
      dout <= sig1; -- Variable assignment, ":= " expected...
    END IF;
  END PROCESS;
END sigvar2err;
```



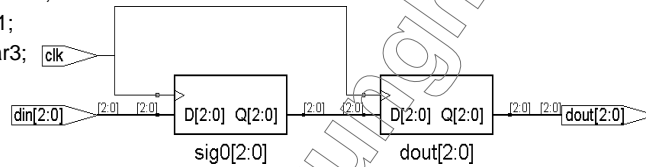
## SIGNAL 과 VARIABLE 합성 (3)



```

ARCHITECTURE a_sigvar3 OF sigvar3 IS
  SIGNAL sig0, sig1 : std_logic_vector(2 DOWNTO 0);
BEGIN
  PROCESS (clk)
  BEGIN
    IF clk='1' AND clk'EVENT THEN
      sig1 <= sig0; -- Concurrent assignment in the Process Block does
      sig0 <= din; -- not cares it's sequence. Compare Example #2
    END IF;
  END PROCESS;
  dout <= sig1;
END a_sigvar3;

```



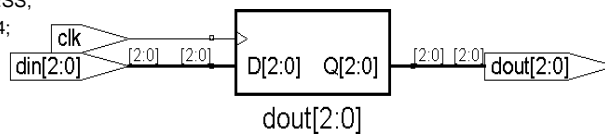
## SIGNAL 과 VARIABLE 합성 (4)



```

ARCHITECTURE a_sigvar4 OF sigvar4 IS
BEGIN
  PROCESS (clk)
  VARIABLE var0, var1 : std_logic_vector(2 DOWNTO 0);
  BEGIN
    IF clk='1' AND clk'EVENT THEN
      var0 := din; -- These two sequential assignment DOES NOTHING!
      var1 := var0; -- Compare with Concurrent assignment; Example #2 and #3
      dout <= var1; -- Just this statement infers F/F between "din" and "dout"
    END IF;
  END PROCESS;
END a_sigvar4;

```





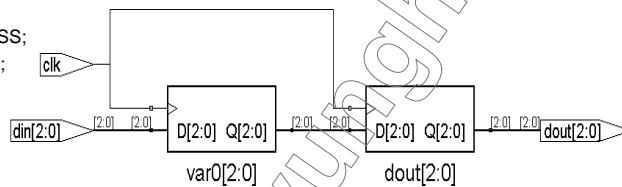
## SIGNAL 과 VARIABLE 합성 (5)



```

ARCHITECTURE a_sigvar5 OF sigvar5 IS
BEGIN
PROCESS (clk)
VARIABLE var0, var1 : std_logic_vector(2 DOWNTO 0);
BEGIN
IF clk='1' AND clk'EVENT THEN
var1 := var0; -- These Two sequential statement infer TWO stage F/F
var0 := din; -- The order of Sequential statements are very important!
-- Compare with Example #4
dout <= var1; -- F/F inferred between "var0" and "dout"
-- Another F/F is inferred between "din" and "var0"
END IF;
END PROCESS;
END a_sigvar5;

```



## SIGNAL 과 VARIABLE 합성 (6)



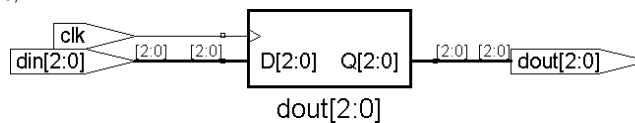
```

ARCHITECTURE a_sigvar5 OF sigvar5 IS
BEGIN
PROCESS (clk)
VARIABLE var0, var1 : std_logic_vector(2 DOWNTO 0);
BEGIN
IF clk='1' AND clk'EVENT THEN
var0 := din; -- Synthesized as ONE stage F/F
var1 := var0; -- between "din" and "var1"
END IF;
dout <= var1; -- But "var1" does nothing!, connected to "dout"
-- F/F inferred between "din" and "dout"
END PROCESS;
END a_sigvar5;

```

VARIABLEs are "local"

Pass local VARIABLE to global SIGNAL





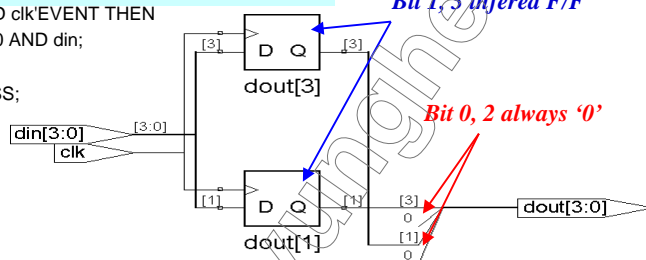
## SIGNAL 과 VARIABLE 합성 (7)



ARCHITECTURE a\_sigvar7 OF sigvar7 IS  
BEGIN

```
PROCESS (clk)
  VARIABLE var0 : std_logic_vector(3 DOWNTO 0);
  BEGIN
    var0 := "0000"; -- These Three sequential statement
    var0(3) := '1'; -- builds Bit map "1010"
    var0(1) := '1';
    IF clk='1' AND clk'EVENT THEN
      dout <= var0 AND din;
    END IF;
  END PROCESS;
END a_sigvar7;
```

Sequential assignment  
has NO multiple drive



## SIGNAL 과 VARIABLE 합성 (8)



ARCHITECTURE a\_sigvar7 OF sigvar7 IS  
SIGNAL sig0 : std\_logic\_vector(3 DOWNTO 0);  
BEGIN

```
PROCESS (clk)
  BEGIN
    sig0 <= "0000"; -- The concurrent statement in the PROCESS block
    sig0(3) <= '1'; -- works sequentially...
    sig0(1) <= '1'; -- Compare with Example #7
    IF clk='1' AND clk'EVENT THEN
      dout <= sig0 AND din;
    END IF;
  END PROCESS;
```

Sequential Block  
NO multiple  
drive problem

Concurrent NOT  
allow multiple drive!

```
-- Following concurrent Statements cause MULTIPLE Drive ERROR!  
-- This region is CONCURRENT!!!!!!!!!!!!!!  
-- sig0 <= "0000";  
-- sig0(3) <= '1';  
-- sig0(1) <= '1';  
END a_sigvar7;
```

## Lab. Exercise



## Signal and Variable



## Tools



### ■ VHDL Simulator

#### ◆ ModelTech's V-System VHDL

<http://www.model.com>

#### ◆ Aldec's Active VHDL

<http://www.aldec.com>

### ■ Read Tool's Tutorials



## Ex.1 sigvar1



- **Compile**
  - ◆ \vhdl\_src\sigvar1.vhd
- **Create Testbench**
- **Simulate and check result (waveform)**
  - ◆ Check Three-Stage Latch ?



## Create Testbench (Active VHDL)



- **Use Testbench Generation Tool**
- **Insert user's stimulus (Clock Signal)**

```
signal CLK : STD_LOGIC := '0';
signal DIN : .....
```

← Give Initial Value of "clk"

→ User can put stimulus here

```
clktoggle : process (clk)
begin
  clk <= not clk after 50 ns;
end process;

stimuli: process(clk)
begin
  if (clk'event and clk='0') then
    din <= "111";
  end if;
end process;
```





## Create Macro File (.do)



- Use Testbench Generation Tool
- Insert Simulation Command

```
vcom "sigvar1.vhd"  
vcom "sigvar1_TB.vhd"  
vsim TESTBENCH_FOR_sigvar1  
# The following lines can be used for timing simulation  
# vcom <backannotated_vhdl_file_name>  
# vcom "sigvar1_TB_tim_cfg.vhd"  
# vsim TIMING_FOR_sigvar1  
view wave  
wave \uut\clk  
wave \uut\din  
wave \uut\dout  
wave \uut\sig0  
wave \uut\sig1  
run 500 ns
```



## Simulation



The screenshot shows the Vivado IDE interface during a simulation. The top window displays a timing diagram with signals: CLK (clock), BOUT, B01, B02, and SIG. The bottom window shows the console output with the following text:

```
Design: 2017.01.14. 2017.01.14 10:04:30 AM 최오영  
Design: Opening Design: ?:\home\user\designer\Project\hdl_soc\sig_var\sig_var.vhd  
ELABEAD: Elaboration process...  
ELABEAD: Elaboration time 0.1 [s].
```



## Ex.2 sigvar2



- **Compile**
  - ◆ \vhdl\_src\sigvar2.vhd
- **Create Testbench**
- **Simulate and check result (waveform)**
  - ◆ Check Two-Stage Latch ?
  - ◆ “sig1” and “dout” are same



## Ex.3 sigvar2-1



- **Compile**
  - ◆ \vhdl\_src\sigvar2-1.vhd
- **Create Testbench**
- **Simulate and check result (waveform)**
  - ◆ Check One-Stage Latch ?
  - ◆ “sig0”, “sig1” and “dout” are same



## Ex.4 sigvar3



- **Compile**
  - ◆ \vhdl\_src\sigvar3.vhd
- **Create Testbench**
- **Simulate and check result (waveform)**
  - ◆ Check Two-Stage Latch ?
  - ◆ “sig1” and “dout” are same



## Ex.5 sigvar4



- **Compile**
  - ◆ \vhdl\_src\sigvar4.vhd
- **Create Testbench**
- **Simulate and check result (waveform)**
  - ◆ Check One-Stage Latch ?
  - ◆ All cascade variable are reduced



## Ex.6 sigvar5



- **Compile**
  - ◆ \vhdl\_src\sigvar5.vhd
- **Create Testbench**
- **Simulate and check result (waveform)**
  - ◆ Check Two-Stage Latch ?
  - ◆ Order of assignment statements is important in the behavior (sequential) environment



## Ex.7 sigvar6



- **Compile**
  - ◆ \vhdl\_src\sigvar6.vhd
- **Create Testbench**
- **Simulate and check result (waveform)**
  - ◆ Check One-Stage Latch ?
  - ◆ Pass “local” variable to output port or SIGNAL



## Ex.8 sigvar7



- **Compile**
  - ◆ \vhdl\_src\sigvar7.vhd
- **Create Testbench**
- **Simulate and check result (waveform)**
  - ◆ Check One-Stage Latch ?
  - ◆ Compare Input value and Latched output (masked by variable)



## Ex.9 sigvar8



- **Compile**
  - ◆ \vhdl\_src\sigvar8.vhd
- **Create Testbench**
- **Simulate and check result (waveform)**
  - ◆ Check One-Stage Latch ?
  - ◆ Compare Input value and Latched output (masked by signal in the process-sequential behavior)



## Ex.10 sigvar8err



### ■ Compile

◆ \vhdl\_src\sigvar8err.vhd

### ■ Create Testbench

### ■ Simulate and check result (waveform)

◆ By using resolved type “std\_logic”, multiple drive does not cause error when compiling.

◆ Compare Input value and Latched output (‘0’ and ‘1’ resolved as ERROR ‘X’)

## VHDL Operator



**IEEE 1076 Predefined  
IEEE 1164 Predefined  
Synthesizable Arithmetics  
Operator Overloading**



## VHDL Operators (IEEE 1076)



- **Binary operators:**
  - ◆ AND, OR, NAND, NOR, XOR, NOT
  - ◆ predefined for BIT and BOOLEAN
- **Relational operators:**
  - ◆ =, /=, <, >, <=, >=
  - ◆ predefined for all types
- **Arithmetic operators:**
  - ◆ +, -, \*, /, \*\*, abs, mod, rem
  - ◆ predefined for integer types
- **Concatenation operators:**
  - ◆ &



## Synthesizable Arithmetic Operators



- **IEEE 1076.3 Arithmetic Package**
  - ◆ **std\_logic\_1164 Package**
    - New Logic types : std\_logic, std\_ulogic
    - Logical Operators
  - ◆ **numeric\_bit**
    - +, -, \*,abs between SIGNED, UNSIGNED of bit and bit\_vector
  - ◆ **numeric\_std**
    - +, -, \*,abs between SIGNED, UNSIGNED of std\_logic and std\_logic\_vector
- **Synopsys Arithmetic Package**
  - ◆ std\_logic\_arith
  - ◆ std\_logic\_signed
  - ◆ std\_logic\_unsigned



## AND Operator (IEEE 1164)

```

-- truth table for 'and' function
CONSTANT and_table : std_logic_table := (
-----
-- | U X 0 1 Z W L H - | |
-----
( 'U', 'U', '0', 'U', 'U', 'U', '0', 'U', 'U' ), -- | U |
( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ), -- | X |
( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),
( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ), -- and
( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),
( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' ),
( '0', '0', '0', '0', '0', '0', '0', '0', '0' ),
( 'U', 'X', '0', '1', 'X', 'X', '0', '1', 'X' ),
( 'U', 'X', '0', 'X', 'X', 'X', '0', 'X', 'X' )
);
FUNCTION "and" ( l, r : std_logic_vector ) RETURN std_logic_vector IS
    ALIAS lv : std_logic_vector ( 1 TO l'LENGTH ) IS l;
    ALIAS rv : std_logic_vector ( 1 TO r'LENGTH ) IS r;
    VARIABLE result : std_logic_vector ( 1 TO l'LENGTH );
BEGIN
    IF ( l'LENGTH /= r'LENGTH ) THEN
        ASSERT FALSE
            REPORT "arguments of overloaded 'and' operator are not of the same length"
            SEVERITY FAILURE;
    ELSE
        FOR i IN result'RANGE LOOP
            result(i) := and_table (lv(i), rv(i));
        END LOOP;
    END IF;
    RETURN result;
END "and";

```



## Synthesizable Arithmetic Operators



### IEEE 1076.3

- ◆ Accepted types : “SIGNED”, “UNSIGNED”, NATURAL, INTEGER
- ◆ Use Type Conversion for STD\_LOGIC
- ◆ Types defined (NUMERIC\_STD)
  - type UNSIGNED is array (NATURAL range <>) of STD\_LOGIC;
  - type SIGNED is array (NATURAL range <>) of STD\_LOGIC;
- ◆ Types defined (NUMERIC\_BIT)
  - type UNSIGNED is array (NATURAL range <>) of BIT;
  - type SIGNED is array (NATURAL range <>) of BIT;
- ◆ Validation Suit





## Synthesizable Arithmetic Operators



### ■ Arithmetic Operators

- ◆ Built-in Subprogram
- ◆ In-Fix
  - +, -, /, \*, /=, <, >, <=, >=, rem, mod
  - Synthesizer NOT support “/,”rem”,”mod”
- ◆ Unary
  - abs, -

### ■ Edge Detection (NUMERIC\_BIT)

- ◆ RISING\_EDGE
- ◆ FALLING\_EDGE



## Synthesizable Arithmetic Operators



### ■ SHIFT/ROTATE

- ◆ SHIFT\_RIGHT, SHIFT\_LEFT
- ◆ ROTATE\_RIGHT, ROTATE\_LEFT

### ■ Type Conversion

- ◆ RESIZE
- ◆ TO\_INTEGER, TO\_UNSIGNED, TO\_SIGNED

### ■ Logical Operations

- ◆ NOT, OR, NOR, AND, NAND, XNOR



## Synthesis Example (Synopsys Library)



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;      -- Synopsys Arithmetic Package
use ieee.std_logic_unsigned.all;  -- Synopsys Arithmetic Package

architecture behave of prep5 is
    signal q_i, adder_output, multiply_output: std_logic_vector (7 downto 0);
begin
    multiply_output <= a * b; -- STD_LOGIC_VECTOR Type Multiply/Addition
    adder_output <= (multiply_output + q_i) when mac = '1' else
        multiply_output;
    .....
end behave;
```



## Synthesis Example (IEEE 1076.3)



```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all; -- IEEE 1076.3 Synthesis Library

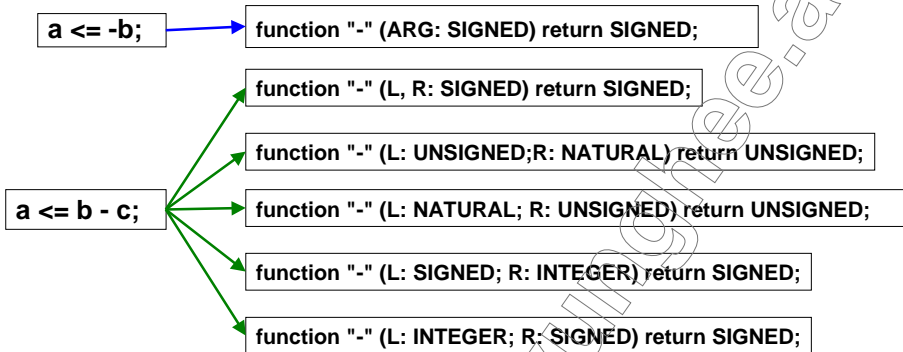
architecture behave of prep5 is
    signal q_i, adder_output, multiply_output: unsigned (7 downto 0);
    -- Unsigned SIGNAL
begin
    multiply_output <= signed(a) * signed(b); -- Convert to STD_LOGIC
    -- to UNSIGNED
    adder_output <= (multiply_output + q_i) when mac = '1' else
        multiply_output;
    q <= std_logic_vector(adder_output); -- Interface to STD_LOGIC
end behave;
```



# Operator Overloading



- VHDL is Object-Oriented
- Type check strictly, but Overloading



## Lab. Exercise



### PREP #5 Arithmetic Circuits

# VHDL Attributes



- Predefined Attributes**
- Vendor-Defined Attributes**
- Tool-Defined Attributes**



## Attributes



### ■ Predefined attributes

- ◆ Return information about data types, signals, blocks, architectures, events, etc.
- ◆ Examples: 'LENGTH, 'LEFT, 'EVENT

### ■ Vendor defined attributes

- ◆ Often used in synthesis
- ◆ Example: enum\_encoding, pinnum

### ■ User defined attributes

- ◆ Not often used



## VHDL Predefined Attributes



- Defined for SIGNAL
- Edge of SIGNAL
  - ◆ EVENT
- Value Returning
  - ◆ LEFT
  - ◆ RIGHT
  - ◆ HIGH
  - ◆ LOW
  - ◆ LENGTH
  - ◆ RANGE
  - ◆ REVERSE\_RANGE



## Predefined Attribute Example



```
signal vector_up : bit_vector( 4 to 9);
signal vector_dn : bit_vector( 25 to 0);

vector_up'LEFT -- returns integer 4
vector_dn'LEFT -- returns integer 25
vector_up'RIGHT -- returns integer 9
vector_dn'RIGHT -- returns integer 0
vector_up'HIGH -- returns integer 9
vector_dn'HIGH -- returns integer 25
vector_up'LOW -- returns integer 4
vector_dn'LOW -- returns integer 0
vector_up'LENGTH -- returns integer 6
vector_dn'LENGTH -- returns integer 26
vector_up'RANGE -- returns integer range 4 to 9
vector_dn'RANGE -- returns integer range 25 to 0
vector_up'REVERSE_RANGE -- returns integer range 9 to 4
vector_dn'REVERSE_RANGE -- returns integer range 0 to 25
```



## Vendor-Defined Attributes



### ■ Metamor Synthesizer

#### ◆ Pin Number Attribute

ENTITY test IS

PORT ( clk, start : IN bit;  
count0, count1 : out bit );

attribute pinnum : string; *-- define pinnum attribute*

attribute pinnum of clk : signal is "13";

attribute pinnum of start : signal is "8";

attribute pinnum of seg40 : signal is "19";

attribute pinnum of seg41 : signal is "20";

END test;



## User Defined Attributes



### ■ Why?

### ■ Example

signal my\_vector : bit\_vector (0 to 4);

attribute MIDDLE : integer;

attribute MIDDLE of my\_vector : signal is my\_vector'LENGTH/2;

.....

my\_vector'MIDDLE *-- returns integer 2*



## Usage of Attributes (Example)



```
Entity masked_parity is
  generic ( size: integer :=5);
  port ( source : in bit_vector(0 to size);
        mask : in bit_vector(source'RANGE); -- RANGE (0 to size)
        result : out bit );
end masked_parity;
```

architecture behave of masked\_parity is

```
begin
  process (source, mask)
    variable tmp : bit;
    variable masked_source : bit_vector (source'RANGE); -- RANGE (0 to size)
  begin
    masked_source := source and mask;
    tmp := masked_source(source'LEFT); -- LEFT
    for I in source'LEFT+1 to source'RIGHT loop -- RIGHT
      tmp := tmp xor masked_source(I);
    end loop;
    result <= tmp;
  end process;
end behave;
```



## Usage of Attribute (Example)



```
function TO_INTEGER (ARG: UNSIGNED) return NATURAL is
  constant ARG_LEFT: INTEGER := ARG'LENGTH-1; -- LENGTH
  alias XARG: UNSIGNED(ARG_LEFT downto 0) is ARG;
  variable RESULT: NATURAL := 0;
begin
  if (ARG'LENGTH < 1) then
    assert NO_WARNING
    report "NUMERIC_BIT.TO_INTEGER: null detected, returning 0"
    severity WARNING;
    return 0;
  end if;
  for I in XARG'RANGE loop -- RANGE
    RESULT := RESULT+RESULT;
    if XARG(I) = '1' then
      RESULT := RESULT + 1;
    end if;
  end loop;
  return RESULT;
end TO_INTEGER;
```

# VHDL Design Partition



**Processes**  
**Blocks**  
**Functions**  
**Procedures**



## Processes and Blocks



### ■ Process

- ◆ Sequential Env.
- ◆ Local VARIABLE
- ◆ NOT Nested Process
- ◆ Sensitivity list or WAIT statement
- ◆ communicate with SIGNAL

### ■ Block

- ◆ Concurrent Env.
- ◆ Local SIGNAL
- ◆ Nested Blocks
- ◆ GUARDED expression





## Concurrent vs. Sequential



**Concurrent**

**Sequential**

architecture sample of my\_proj is  
signal Qreg: bit\_vector(15 downto 0);  
begin

P1: process(Rst, Clk1)  
begin  
...  
end process;

**Sequential**

P2: process(Rst, Clk2)  
begin  
...  
end process;

Q <= Qreg;  
DataAddr <= BaseAddr & Q(7 downto 0);

end sample;



## Concurrent vs. Sequential



**BEGIN**

Statement

Statement

Statement

**END**

**BEGIN**

Statement

Statement

Statement

**END**



## Process Example



Entity experiment is  
 port ( source : in bit\_vector(0 to 3);  
 ce : in bit;  
 wrclk : in bit;  
 selector : in bit\_vector(0 to 1);  
 result : out bit );  
 end experiment;

architecture behave od experiment is  
 signal intreg : bit\_vector(0 to 3);  
 begin -- dataflow(concurrent) environment

```

writer : process -- process statement
-- variable declarative region (empty here)
begin
-- sequential (clocked) statement
wait until wrclk'event and wrclk='1';
if (ce='1') then
  intreg <= source;
end if;
end process;

```

```

reader : process (intreg, selector) -- process
-- with sensitivity list
-- variable declarative region (empty here)
begin
-- sequential (not-clocked) statements
case selector is
when "00" =>
  result <= intreg(0);
when "01" =>
  result <= intreg(1);
when "10" =>
  result <= intreg(2);
when "11" =>
  result <= intreg(3);
end case;
end process;
end behave;

```



## Block Example



Architecture dataflow of example is  
 signal global\_sig, g1, g2, c, bit;  
 begin

```

B1 : block -- Block declarative region
  signal local_sig : bit; -- Local SIGNAL
begin -- Block concurrent statements
  local_sig <= global_sig; -- Communicate with global signal directly

```

```

B2 : block (c='1') -- Block in a Block
  port ( o1, o2 : out bit) -- Block has "GUARD" expression
  port map (o1=>g1, o2=>g2); -- Block port declarations
begin -- Block port mapping to global signal
  o1 <= guarded local_sig; -- Latch inference
  o2 <= global_sig;
end block;

```

```

end block;
end dataflow;

```



## Subprograms : Functions & Procedures



- Execute Sequentially
- Local variable
- Can be called from the dataflow (concurrent) or any sequential environment
- Can be overloaded (Synthesizer resolve this)
- Functions
  - ◆ all input argument, return single value
- Procedure
  - ◆ in, out, inout argument
  - ◆ Side effect



## Subprogram Example



```
■ Procedure
procedure increment ( vect : inout bit_vector; ena : in bit := '1' ) is -- in, inout argument
begin
  if (ena='1') then
    vect := vector_adder (x=>vect, "000001" ); -- Cause side effect
  end if ;
end increment ;
```

```
■ Functions
function vector_adder (x : bit_vector; y : bit_vector) -- input argument
return bit_vector is -- return type
  variable carry : bit := '0';
  variable result : bit_vector(x'RANGE) ;
begin
  for i in x'RANGE loop
    result (i) := x(i) XOR y(i) XOR carry ;
    carry := carry AND (x(i) OR y(i)) OR x(i) AND y(i) ;
  end loop ;
  return result ; -- one return value
end vector_adder ;
```

# Resolution Functions



## Multiple Drive Resolution Function and Type Resolve Multiple Drive Error



## Multiple Drive



- On Concurrent Assignment, multiple source to one signal must be resolved
- Multiple Drive Problem is common when modeling the buses with three-state drivers
- `std_logic_1164` provides resolved type and resolution function “resolved”.

```
subtype std_logic is resolved std_ulogic;
```



# Resolution Function (IEEE 1164)

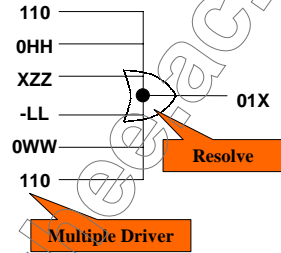


```

-- resolution function
CONSTANT resolution_table : std_logic_table := (
  -- | U X 0 1 Z W L H - | |
  ( 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U', 'U' ), -- U |
  ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ), -- X |
  ( 'U', 'X', '0', 'X', '0', '0', '0', '0', 'X', 'X' ), -- 0 |
  ( 'U', 'X', 'X', '1', '1', '1', '1', '1', 'X', 'X' ), -- 1 |
  ( 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', 'X', 'X' ), -- Z |
  ( 'U', 'X', '0', '1', 'W', 'W', 'W', 'W', 'X', 'X' ), -- W |
  ( 'U', 'X', '0', '1', 'L', 'W', 'L', 'W', 'X', 'X' ), -- L |
  ( 'U', 'X', '0', '1', 'H', 'W', 'W', 'H', 'X', 'X' ), -- H |
  ( 'U', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X' ) -- - |
);

FUNCTION resolved ( s : std_ulogic_vector ) RETURN std_ulogic IS
  VARIABLE result : std_ulogic := 'Z';
BEGIN
  IF (s'LENGTH = 1) THEN RETURN s('LOW');
  ELSE
    FOR i IN s'RANGE LOOP
      result := resolution_table(result, s(i));
    END LOOP;
  END IF;
  RETURN result;
END resolved;

```



# Resolve Multiple Drive Error



```

entity test_resolver is
  port ( a, b : bit ;
         o : out bit );
end test_resolver;
architecture dataflow of test_resolver is
  signal tmp : bit;
begin
  tmp <= a ;
  tmp <= b ;
  o <= tmp ;
end dataflow ;

```

write resolution function

Use resolved type

```

library ieee;
use ieee.std_logic_1164.all;
entity test_resolver is
  port ( a, b : std_logic ;
         o : out std_logic );
end test_resolver;
architecture dataflow of test_resolver is
  signal tmp : std_logic ;
begin
  tmp <= a ;
  tmp <= b ;
  o <= tmp ;
end dataflow ;

```

```

entity test_resolver is
  port ( a, b : bit ;
         o : out bit );
end test_resolver;

architecture dataflow of test_resolver is

  -- Write the resolution function that ANDs the elements:
  function my_and_resolved ( a : bit_vector ) return bit is
    variable result : bit := '1';
  begin
    for i in a'range loop
      result := result AND a(i);
    end loop;
    return result;
  end my_and_resolved;

  -- Declare the signal and attach the resolution function to it:
  signal tmp : my_and_resolved bit;
begin
  tmp <= a ;
  tmp <= b ;
  o <= tmp ;
end dataflow ;

```

# BUS and REGISTER



# BUS and REGISTER



## ■ VHDL Entity Class

### ◆ BUS

- make GUARDED signal (three-state)

### ◆ REGISTER

- make Latch inference



## BUS Example



```
-- include the IEEE 1164 package to use type std_logic.
library ieee ;
use ieee.std_logic_1164.all ;

entity test_bus is
  port ( c,d : std_logic ;
        o : out std_logic BUS) ; -- An entity with a BUS entity-class signal
end test_bus ;

architecture behave of test_bus is
begin
  process (c,d)
  begin
    if (c = '1') then
      o <= d ;
    else
      o <= NULL ; -- Switch-Off (Three-State)
    end if ;
  end process ;
end behave ;
```



## REGISTER Example



```
-- include the IEEE 1164 package to use type std_logic.
library ieee ;
use ieee.std_logic_1164.all ;

entity test_reg is
  port ( c,d : std_logic ;
        o : out std_logic) ;
end test_reg ;

architecture behave of test_reg is
  signal s_o : std_logic REGISTER ; -- REGISTER entity-class signal
begin
  process (c,d)
  begin
    if (c = '1') then
      s_o <= d ;
    else
      s_o <= NULL ; -- DO NOTHING, Latch inference
    end if ;
  end process ;
  o <= s_o ;
end behave ;
```

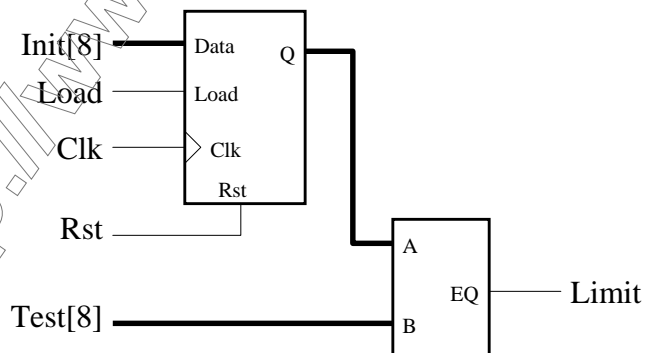
## Sample Circuit



## Shifter-Comparator



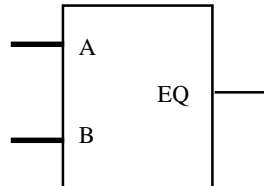
## Sample Circuit







## Comparator (Dataflow)



-----  
-- Eight-bit comparator

--  
entity compare is  
port ( A, B: in bit\_vector(0 to 7);  
EQ: out bit);  
end compare;

architecture compare1 of compare is  
begin

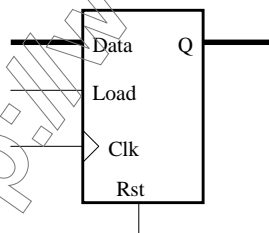
-- **Concurrent Statements**

EQ <= '1' when (A = B) else '0';

end compare1;



## 8-bit shifter



-----  
-- Eight-bit barrel shifter

--  
entity rotate is  
port( Clk, Rst, Load: in bit;  
Data: bit\_vector(0 to 7);  
Q: out bit\_vector(0 to 7));  
end rotate;



## 8-bit Shifter (Behavior)



Registers are implied for all assignments dependent on the clock.

```

architecture rotate1 of rotate is
  signal Qreg: bit_vector(0 to 7);
begin
  -- Process Block : Sequential Statements
  reg: process(Rst, Clk)
  begin
    if Rst = '1' then -- Async reset
      Qreg <= '00000000';
    elsif (Clk = '1' and Clk'event) then
      -- Register Implied
      if (Load = '1') then -- Load
        Qreg <= Data;
      else -- Shift
        Qreg <= Qreg(1 to 7) & Qreg(0);
      end if;
    end if;
  end process;
  Q <= Qreg;
end rotate1;

```



## 8-bit Shifter (Dataflow)



Describing a circuit using dataflow may result in a simpler design description.

```

architecture rotate2 of rotate is
  signal D, Qreg: bit_vector(0 to 7);
begin
  -- Shift Logic
  D <= Data when (Load = '1') else
    Qreg(1 to 7) & Qreg(0);

  dff(Rst, Clk, D, Qreg); -- D-F/F Procedure

  Q <= Qreg;

end rotate2;

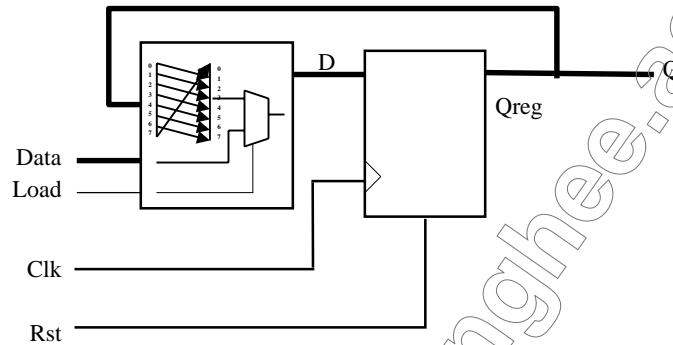
```



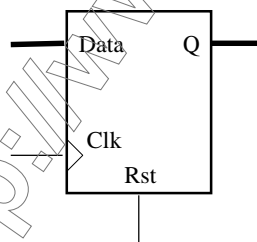
## 8-bit Shifter (Dataflow)



Inputs                      Shift Logic                      Registers                      Outputs



## D Flip-Flop Procedure



```

procedure dff ( Rst, Clk : in bit;
               D : in bit_vector(0 to 7);
               Q : out bit_vector(0 to 7)) is
begin
  if Rst = '1' then
    Q <= '00000000';
  elsif Clk = '1' and Clk'event then
    Q <= D;
  end if;
end dff;

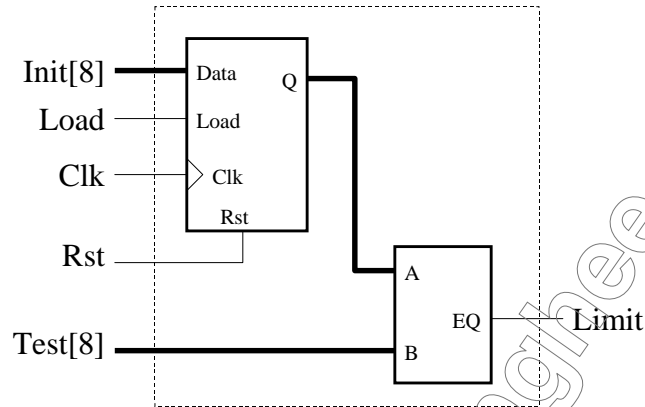
```

**Procedures can help to describe common behavior.**

*Note: a component could also be used for this purpose.*



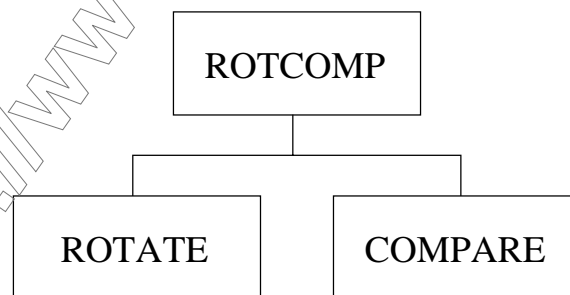
## Structural VHDL



*The structural features of VHDL can be used to create the equivalent of a schematic netlist.*



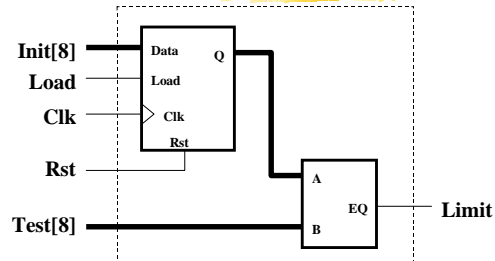
## Design Hierarchy



*You can think of your hierarchical design as a tree structure.*



## Entity



```
entity rotcomp is
  port( Clk, Rst, Load: in bit;
        Init : in bit_vector(0 to 7);
        Test : in bit_vector(0 to 7);
        Limit : out bit );
end rotcomp;
```



## Architecture



architecture structure of rotcomp is

-- **Component Declare "Compare"**

component compare

port(A, B: in bit\_vector(0 to 7); EQ: out bit);

end component;

-- **Component Declare "Rotate"**

component rotate

port(Clk, Rst, Load: in bit;

Data: in bit\_vector(0 to 7);

Q: out bit\_vector(0 to 7));

end component;

-- **Signals for netlist**

signal Q: bit\_vector(0 to 7);

begin

-- **Component Instantiate and Netlist**

COMP1: compare port map ( EQ=>Limit, A=>Q, B=>Test);

ROT1: rotate port map (Clk, Rst, Load, Init, Q);

end structure;



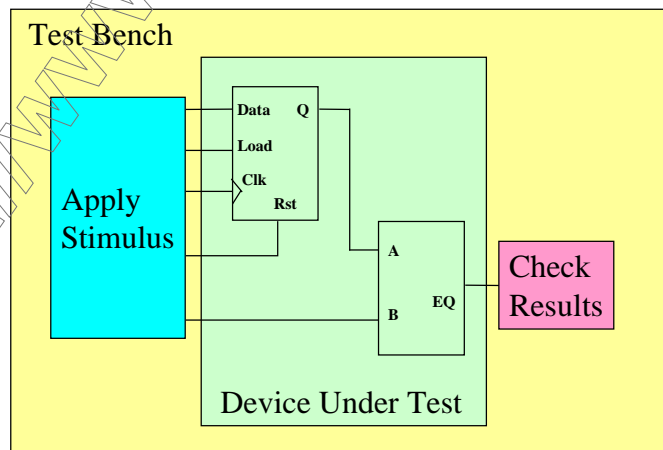
## Test Benches



- Analogous to a hardware test fixture
- Written using behavioral (sequential) VHDL statements
- Use structure (hierarchy) to include the design
- Can be used to create automated tests

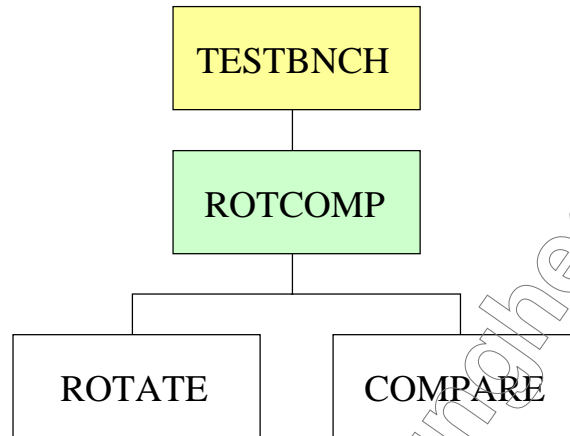


## Test Benches: Virtual Circuits





## Design Hierarchy



## Sample Test Bench



entity testbnch is  
 -- Testbench has NO Port  
 end testbnch;

architecture behavior of testbnch is

-- Declare DUT Component  
 component rotcomp is  
 port(Clk, Rst, Load: bit;  
 Init: bit\_vector(0 to 7);  
 Test: bit\_vector(0 to 7);  
 Limit: out bit;  
 end component;

-- SIGNALs used in testbench  
 signal Clk, Rst, Load: bit;  
 signal Init: bit\_vector(0 to 7);  
 signal Test: bit\_vector(0 to 7);  
 signal Limit: bit;

begin  
 DUT: rotcomp -- Instantiate DUT  
 port map (Clk, Rst, Load, Init, Test,  
 Limit);

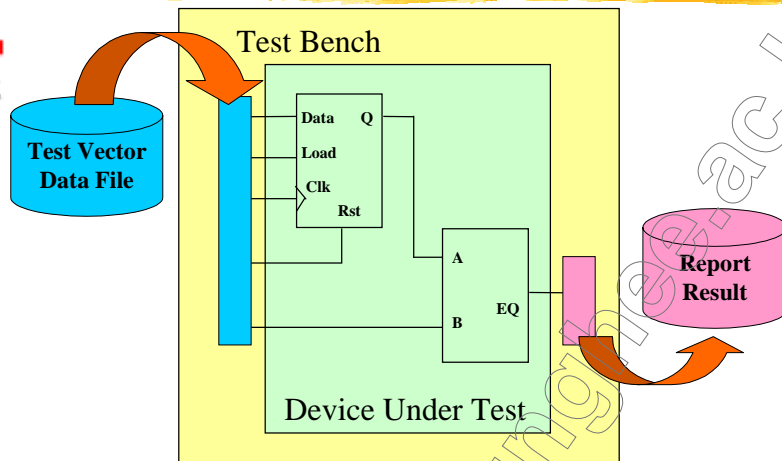
clock: process -- Clock Generator  
 variable clktmp: bit := '0';  
 begin  
 clktmp := not clktmp;  
 Clk <= clktmp;  
 wait for PERIOD / 2;  
 end process;

stimulus: process -- Apply Stimulus  
 begin  
 Rst <= '0';  
 Load <= '1';  
 Init <= "00001111";  
 Test <= "11110000";  
 wait for PERIOD;  
 Load <= '0';  
 wait for PERIOD \* 4;  
 end process;

end behavior;



## Reading Test Vectors



## Reading Test Vectors



```
stimulus: process
file vecFile : text is in "test4.vec"; -- Data File
variable vecLine : line; -- Text IO Buffer
variable vecString : string;
begin
while not endfile(vecFile) loop
readline(vecFile, vecLine); -- Read Line from File
read(vecLine, r, good => good_number); -- Copy to IO Buffer
read (file_line,vecString);
-- Convert the input string to stimulus data
-- Apply the stimulus, wait for some time, etc.
end loop;
assert false report "Test complete";
wait;
end process;
```





## What We're Learned So Far



- Basic language features
- Dataflow, behavior and structure
- Combinational and registered circuits
- Hierarchy
- Test bench basics
- Lots of VHDL jargon!

## Lab. Exercise



### Testbench

# Verilog 개요



**Data flow and behavioral descriptions**  
**Concurrent and sequential functionality**  
**Numbers and data types.**



## Modules



- **Basic Building Block**
  - ◆ **Boundaries and Contents of logic**
- **Module name and Port List**
- **Port's Direction mode**

```
module small_block (a, b, c, o1, o2);  
  input a, b, c;  
  output o1, o2;  
  
  wire s;  
  
  assign o1 = s || c;  
  assign s = a && b;  
  assign o2 = s ^ c;  
  
endmodule
```



## Numbers



- Bitwidth and Prefix
- Default : 32-bit width decimal

Name	Prefix	Legal Characters
binary	'b	01xXzZ_?
octal	'o	0~7xXzZ_?
decimal	'd	0~9xXzZ_?
hexadecimal	'h	0~9a~fA~FxXzZ_?

"\_" a separator to improve readability, 'x','X' Unknown, 'z','Z','?' Tri-State Value

Example)

8'b01010101	8-bit Binary number
20'hff_ff	20-bit Hexadecimal number
352	32-bit Decimal number
10'bZ	10-bit all tri-state



## Data Types in Verilog



- Only two fundamental types: nets and registers.
- A net is analogous to a wire.
- A Reg is a wire that has memory (can store a value over time).
- Verilog lacks the Data richness of VHDL, but also eliminates the type checking and conversion overhead.



## Data Types



### ■ Data Types (default scalar)

#### ◆ Nets

- wire
- tri
- supply0
- supply1
- wand & wor

#### ◆ register

#### ◆ parameter

### ■ Optional Range Specification

[<MSB>:<LSB>]



## Net Data Types



### ■ NOT store value

#### ■ wire Nets

- ◆ driven by single gates

#### ■ tri Nets

- ◆ driven by multiple gates

#### ■ supply Nets

- ◆ Power or GND supplies in the circuits

#### ■ wand and wor Net Types

- ◆ wired logic



## Register Data Type



- Store value
- Use keyword **reg**
- Register assigned in the **always** block with clock edge event (**posedge** or **negedge**)
- Flip-Flop Synthesized only in **always** block **reg**



## Parameter Data Type



- **parameter**
  - ◆ a Default value
  - ◆ Overriden when instantiated
- Declaration Local to **begin~end** block
- Array of **reg** and **integer** Declaration

```
input [10:0] data;
always @ (data)
begin: named_block
integer i;
parity = 0;
for (i = 0; i < 11; i = i + 1)
parity = parity ^ data[i];
end //named_block
```

```
input [0:3] address;
input [0:7] date_in;
output [0:7] data_out;
reg [0:7] data_out, mem [0:15];

always @ (address or date_in or we)
if (we) mem [address] = date_in;
else data_out = mem [address];
```



## Continuous Assignments



- Assign to nets and ports in the dataflow env.

```
module tri_asgn (source, ce, wrclk, selector, result) ;
input [0:3]source, ce, wrclk ;
input [0:1]selector ;
output result ;
reg [0:3]intreg ;
reg result ;
wire [0:1]sel = selector ; // net declaration assignment
tri result_int ;
// continuous assignment statement
assign
result_int = (sel == 2'b00)? intreg[0] : 1'bZ ,
result_int = (sel == 2'b01)? intreg[1] : 1'bZ ,
result_int = (sel == 2'b10)? intreg[2] : 1'bZ ,
result_int = (sel == 2'b11)? intreg[3] : 1'bZ ;
always @(posedge wrclk)
begin
if (ce)
begin
intreg = source;
result = result_int ;
end
end
endmodule
```



## Procedural Assignments



- Assign register variables
- Blocking assignment
  - ◆ “=”
  - ◆ sequential block
  - ◆ assign immediately
- Non-Blocking Assignment
  - ◆ “<=”
  - ◆ Scheduled assignment



## Always Block



- sequential execution block
- sensitivity list
- communicate via reg variable

```

always @(condition)
begin
    sequential statement;
    sequential statement;
    sequential statement;
    .
    .
end

```



## Concurrent vs. Sequential



```

module my_proj (...)
    declarations
        concurrent assignments, etc.
    Sequential {
        always @(condition)
        begin
            ...
        end
    }
    Concurrent {
        Sequential {
            always @(condition)
            begin
                ...
            end
        }
        concurrent assignments, etc.
    }
endmodule

```



## Always block Example



```
module mux_case (source, ce, wrclk, selector, result);
input [0:3]source;
input ce, wrclk;
input [0:1]selector;
output result;
reg [0:3]intreg;
reg result, result_int;

always @(posedge wrclk) // Clocked Flip-Flop
begin
if (ce)
intreg = source;
result = result_int;
end

always @(intreg or selector) // Combinational Block
case (selector)
2b'00: result_int = intreg[0];
2b'01: result_int = intreg[1];
2b'10: result_int = intreg[2];
2b'11: result_int = intreg[3];
endcase
endmodule
```



## Module Instantiation



- Design hierachy
- Parameter override during instantiation of Module

```
module top (a, b);
input [0:3] a;
output [0:3] b;
do_assign #(4) name (a, b);
endmodule

module do_assign (a, b);
parameter n = 2;
input [0:n-1] a;
output [0:n-1] b;
assign b = a;
endmodule
```

- defparam statement

```
do_assign name (a, b);
defparam name.n = 4;
```





## Module Instanciation Example



```
module scanner (reset, stop, load, clk, load_value, data) ;
input reset, stop, load, clk;
input [3:0]load_value;
output [3:0]data;
reg [4:0] addr;

// Instantiate and connect 4 32x1-bit rams
RAM_32x1 U0 (.a(addr), .d(load_value[0]), .we(load), .o(data[0]));
RAM_32x1 U1 (.a(addr), .d(load_value[1]), .we(load), .o(data[1]));
RAM_32x1 U2 (.a(addr), .d(load_value[2]), .we(load), .o(data[2]));
RAM_32x1 U3 (.a(addr), .d(load_value[3]), .we(load), .o(data[3]));

// Generate the address for the rams
always @(posedge clk or posedge reset)
begin
if (reset)
addr = 5'b0 ;
else if (~stop)
addr = addr + 5'b1 ;
end
endmodule

module RAM_32x1 ( a, we, d, o);
input [4:0] a;
input we, d ;
output o;
endmodule
```



## Operators



- Arithmetic
  - ◆ + - \* /
  - ◆ divider support constant and poer of 2
- Relational
  - ◆ < > <= >=
- Logical
  - ◆ == (EQ) != (NEQ) !(NOT) && (AND) || (OR)
- Bitwise
  - ◆ ~ (NOT) & (AND) |(OR) ^ (XOR) ~^ (EQ) ^~ (EQ)
- Reduction (unary)
  - ◆ & (AND) |(OR) ^ (XOR) ~& (NAND) ~|(NOR) ~^ (NOT-XOR)
- Conditional
  - ◆ ? :
- Concatenation
  - ◆ { }



## Example : === and !==



```
module triple_eq_neq (in1, in2, O);
output [0:10] O;
input [0:2] in1, in2;
assign
O[0] = 3'b0x0 === 3'b0x0, // output is 1
O[1] = 3'b0x0 !== 3'b0x0, // output is 0
O[2] = 3'b0x0 === 3'b1x0, // output is 0
O[3] = 3'b0x0 !== 3'b1x0, // output is 1
O[4] = in1===3'b0x0,      // LHS is non constant so this
                          // produces warning that comparison
                          // metalogical character is
                          // with zero. output is 0
O[5] = in1 !== 3'b0x0,    // LHS is non constant so this
                          // produces warning that comparison
                          // with metalogical character is
                          // zero.output is 1,because it
                          // checks for not equality
O[6] = in1 === 3'b010,    // normal comparison
O[7] = in1 !== 3'b010,    // normal comparison
O[8] = in1 === in2,      // normal comparison
O[9] = in1 !== in2,      // normal comparison
O[10] = 3'b00x === 1'bx; // output is 1
endmodule
```



## Statements



### ■ if~else statement

```
if (condition)
  expressions;
else if (condition)
  expressions;
end;
```

### ■ case/casex/casez statement

```
case (var)
  const0 : begin
    ...
  end;
  const0 : begin
    ...
  end;
  ...
  default : begin
    ...
  end;
endcase;
```



## Statements



- **disable statement**
- **for statements**
  - ◆ **for (init\_value; loop\_condition; incrment)**
- **forever statement**
- **repeat statement**
- **while statement**



## Subprograms



- **Functions**
  - ◆ **input params and return value**
  - ◆ **combinational**
- **Tasks**
  - ◆ **input and output params**
  - ◆ **combinational and sequential**
  - ◆ **System Tasks**



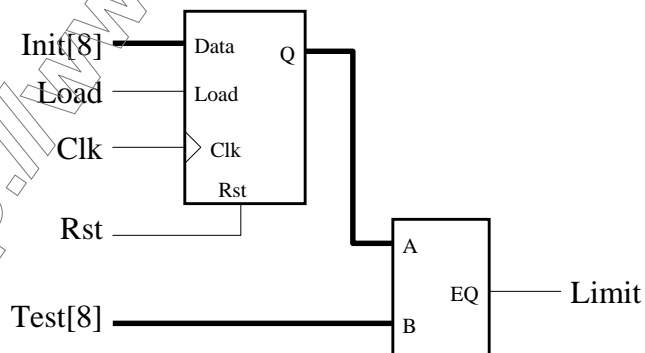
## Compiler directives



- ``define`
- ``ifdef`
- ``else`
- ``endif`
- ``include`
- ``signed`
- ``unsigned`
- ``unconnected_drive`
- ``nounconnected_drive`

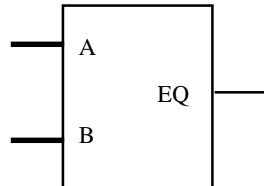


## Sample Circuit





## Sample : Comparator



```

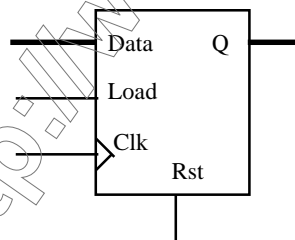
/*****
 * Eight-bit comparator
 */
module compare (A,B,EQ);
  input [0:7] A;
  input [0:7] B;
  output EQ;

  assign EQ = (A == B);

endmodule
  
```



## Sample : Shifter



```

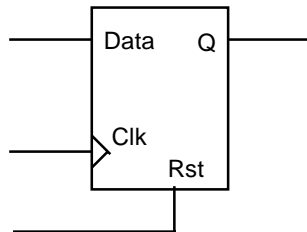
/*****
 * Shifter
 */
module rotate (Clk,Rst,Load,Data,Q);
  input Clk,Rst,Load;
  input [0:7] Data;
  output [0:7] Q;

  reg [0:7] Q;

  always @(Rst) Q = 0;
  always @(posedge Clk)
  begin
    if (Load)
      Q = Data;
    else
      Q = {Q[1:7], Q[0]};
    end;
  endmodule
  
```



## D Flip-Flop



```

module dff (Rst,Clk,D,Q);
  input Rst,Clk,D;
  output Q;
  reg Q;

```

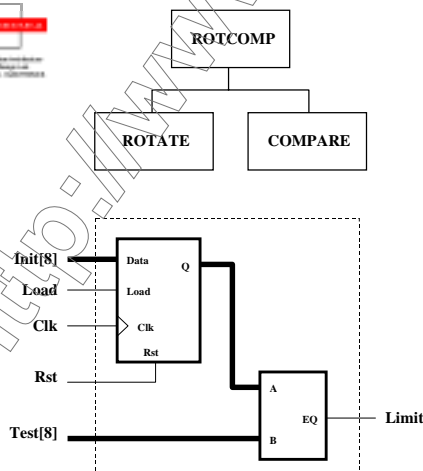
```

  always @(posedge Rst or posedge Clk)
    if Rst
      Q = 1;
    else
      Q = D;
endmodule ;

```



## Structural Verilog



```

module rotcomp (Clk,Rst,Load,Init,Test,Limit);
  input Clk,Rst,Load;
  input [0:7] Init;
  input [0:7] Test;
  output Match;

```

```

  wire [0:7] Q;

```

```

  rotate ROT1 (Clk,Rst,Load,Init,Q);
  compare COMP1 (Q,Test,Limit);

```

```

endmodule

```

## HDL과 PLD 전용 언어의 비교



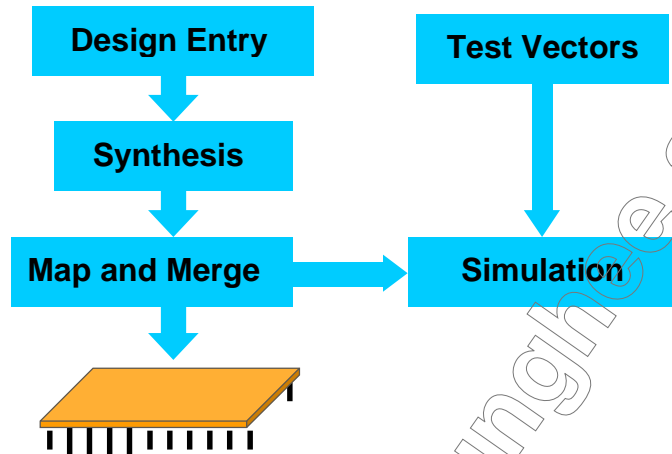
### 근본적인 차이



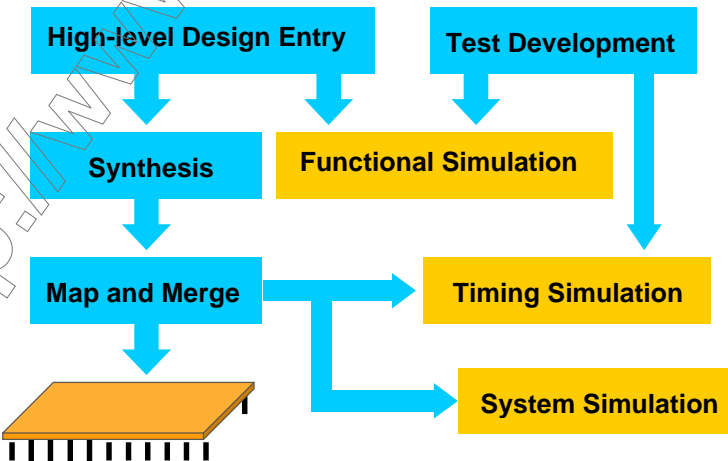
- **VHDL, Verilog** 과 같은 **HDL**은 합성과 시뮬레이션을 위한 언어이다.
- **HDL**은 순차구문(**sequential**)을 제공하여 언어적인 성격을 가진다.
- **HDL**은 고수준의 자료형(**integer, real...**)을 다루며 제어구문 (**if~else, case~ when**)을 지원한다.
- **HDL**은 세부적인 타이밍정보를 지정할수 있다.



## PLD Language Flow



## HDL Language Flow







## More Than Design Entry...



- HDLs allow the overall design specification (not just the grubby details) to be described.
- Standard languages opens up tremendous opportunities for system simulation and interface to other design tools.



## Similarities Between HDLs and PLD Languages



- VHDL, Verilog and PLD Languages provide similar features for control flow.
- Hierarchy features are provided in most PLD languages.
- Describing combinational and registered logic is similar, but...



## Some Important Differences



- VHDL and Verilog do not have built-in registers (Registers inference) or other device features.
- PLD languages do not have high-level data types.
- PLD languages have powerful truth table and state diagram languages.
- PLD languages are generally non-portable, and are poor test languages.



## ABEL and VHDL Compare



### ■ VHDL

```
entity counter is
  port(Clk,Rst,Load: bit;
        Data: in integer range 0 to 255;
        Count: out integer range 0 to 255);
end counter;

architecture count2 of counter is
  signal D,Q: integer range 0 to 255;
begin
  D <= Data when Load = '1' else
    0 when Q = 255 else
    Q + 1;

  dff(Rst, Clk, D, Q);

  Count <= Q;
end count2;
```

### ■ ABEL

```
module counter
  Clk,Rst pin 1,2;
  D7..D0 pin 4..11;
  Data = [D7..D0];
  Q7..Q0 pin 14..21 istype 'reg';
  Count = [Q7..Q0];

  equations
    Count.clk = Clk;
    Count.clr = Rst;
    Count := Data & Load
      # (Count.fb + 1) & !Load;

end counter;
```

**PLD languages have built-in features for PLD and FPGA design.**



## State Machines (Altera AHDL)



### Special data type (my\_machine):

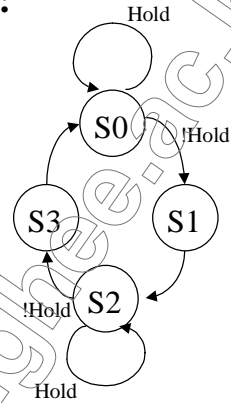
```

subdesign sm
( Clk, Rst, Hold: input;
  O: output;)

variable
  my_machine: machine of bits q1,q0
              with states (S0,S1,S2,S3);

begin
  ...

```



## Simple State Machine (AHDL)



```

variable
  my_machine: machine of bits q1,q0
              with states (S0,S1,S2,S2);

begin
  my_machine.clk = Clk;
  my_machine.reset = Rst;

  table
    my_machine, Hold => my_machine, O;
    S0      , 0 => S0, 1;
    S0      , 1 => S1, 0;
    S1      , X => S2, 0;
    S2      , X => S3, 1;
    S3      , 0 => S3, 1;
    S3      , 1 => S0, 1;
  end table;
end;

```

*Neither VHDL nor Verilog have such a concise state machine language.*



## Summary



- For simple circuits, PLD languages such as ABEL and AHDL are more concise.
- VHDL and Verilog do not assume anything about your registers; you need to be explicit about their behavior.
- Follow an RTL strategy (know where your registers are, and know which signals are registered) when using HDL synthesis.

## VHDL Synthesis



**Flip-Flops**  
**Three-,Bi-Dir Buffers**  
**State Machines**  
**Arithmetic and Related Logics**  
**Resource Sharing and Optimization**  
**Mux and Selectors**  
**ROM, PLA, Decoders**



## Goals of This Section



- More closely examine synthesis conventions
- Find out what is and is not synthesizable in today's tools
- See how HDL statements relate to actual hardware
- Examine the most common synthesis pitfalls

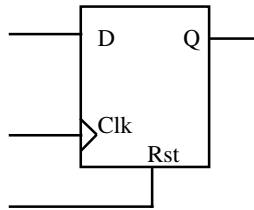
## Flip-Flops



**Sync/Async Rest or Reset**  
**Latch**  
**Chip Enable**



## Flip-Flop Convention

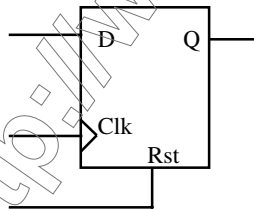


architecture behavior of dff is

```
begin
  process(Rst,Clk)
  begin
    if Rst = '1' then -- Async reset
      Q <= (others=>'0');
    elsif (Clk = '1' and Clk'event) then
      Q <= D;
    end if;
  end process;
end behavior;
```



## Synchronous Reset Convention

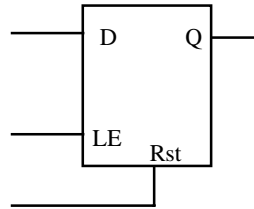


architecture behavior of dff is

```
begin
  process(Clk)
  begin
    if (Clk = '1' and Clk'event) then
      if Rst = '1' then -- Sync reset
        Q <= (others=>'0');
      else
        Q <= D;
      end if;
    end if;
  end process;
end behavior;
```



## Latch Convention

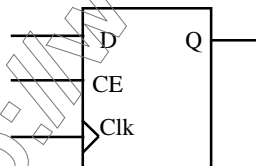


architecture behavior of latch is

```
begin
process(LE, D)
begin
  if LE = '1' then
    Q <= D;
  end if;
end process;
end behavior;
```



## Clock Enable



architecture behavior of dff is

```
begin
process(Clk)
begin
  if Clk = '1' and Clk'event then
    if CE = '1' then
      Q <= D;
    end if;
  end if;
end process;
end behavior;
```



## Flip-Flop Convention (block)



-- Level Sensitive

```
B1 : block (ena='1')
begin
  Q <= guarded D;
end block;
```

-- Edge-Sensitive Sync reset

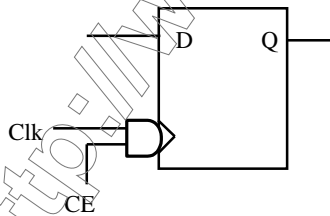
```
B2 : block (clk'event and clk='1')
begin
  Q <= guarded 0 when Rst='1' else D;
end block;
```

-- Clock Enable

```
B2 : block (clk'event and clk='1')
begin
  Q <= guarded D when CE='1' else Q;
end block;
```



## Gated Clock



architecture behavior of dff is

```
begin
  process(Clk)
  begin
    if CE = '1' and
      Clk = '1' and Clk'event then
      Q <= D;
    end if;
  end process;
end behavior;
```





## Flip-Flop (Wait Statement)



architecture behavior of dff is

```
begin
  process ← NO Sensitivity List
  begin
    wait until clk'event and clk='1'; ←
    Q <= D;
  end process;
end behavior;
```



## Suggested Register Conventions



- Use a **process** statement (or procedure) to describe a flip-flop or latch
- Use **'event** attribute or **rising\_edge** function to detect clock edge
- Use **if-then-else** style for reset and clock logic
- **Avoid Gated Flip-Flop**
- **Avoid accidental Latch**
- **Verilog conventions are similar**

## Lab. Exercise



### **Flip-Flops**

- Sync Set/Reset**
- Async Set/Reset**
- Level Sensitive Latch**
- Edge Triggered Latch**

## Buffers



- Three-State Buffers**
- Bi-Directional Buffers**
- Buses**



## Three-state Buffers



```
entity three-state is
  port ( input_1, input_2 : in std_logic ;
        ena_1, ena_2 : in std_logic ;
        output : out std_logic ) ;
end three-state ;
```

```
architecture dataflow of three-state is
begin
```

*-- Dataflow*

```
output <= input_1 when ena_1 = '1' else 'Z';
output <= input_2 when ena_2 = '1' else 'Z';
```

```
end dataflow ;
```

*-- Behavior*

```
driver1 : process (ena_1, input_1)
begin
  if (ena_1='1') then
    output <= input_1 ;
  else
    output <= 'Z';
  end if ;
end process ;
```

```
driver2 : process (ena_2, input_2)
begin
  if (ena_2='1') then
    output <= input_2 ;
  else
    output <= 'Z';
  end if ;
end process ;
```

*Multiple Drive!*

*Synthesizable, but "ena\_1" and "ena\_2"  
Never be '1' simultaneously!*



## Bi-Directional Buffers



```
entity bidir_function is
  port ( bidir_port : inout std_logic ;
        ena       : in std_logic ;
        ...
  );
end bidir_function ;
```

```
architecture a_bidir of bidir_function is
  signal internal_signal, internal_input : std_logic ;
begin
```

```
bidir_port <= internal_signal when ena = '1' else 'Z';
internal_input <= bidir_port ;
```

```
-- use internal_input
..... <= internal_input;
```

```
-- generate internal_signal
internal_signal <= .....
```

```
end a_bidir ;
```



## Buses



entity three-state is

```
port ( input_signal_1, input_signal_2 : in  std_logic_vector (0 to 7) ;  
       ena_1, ena_2                 : in  std_logic ;  
       output_signal                 : out std_logic_vector(0 to 7)  
     );  
end three-state ;
```

architecture exemplar of three-state is  
begin

```
output_signal <= input_signal_1 when ena_1 = '1' else 'ZZZZZZZZ';  
output_signal <= input_signal_2 when ena_2 = '1' else 'ZZZZZZZZ';
```

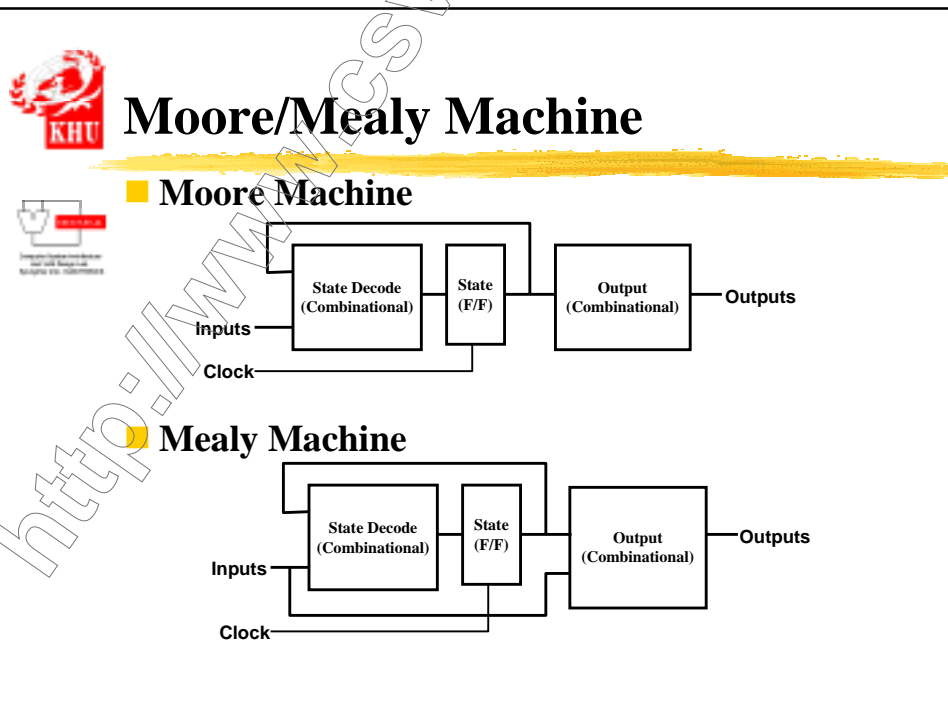
end exemplar ;

## Lab. Exercise



### Tristate Buffer Bidirectional Bus

# State Machine





## State Machine Coding Style



- Use Enumeration Type to represent “State”
- Supply Power-Up and Reset Condition
- Use **case** statement for state transition
- **DO NOT** use **OTHER** entry in the **case**
- Use **if~else** for input condition
- Always assign state output under every condition



## State Machine Example



### ■ DRAM Refresh

```
entity ras_cas is
  port ( clk, cs, refresh, reset : in bit ;
        ras, cas, ready       : out bit ) ;
end ras_cas ;
```

```
architecture a_ras_cas of ras_cas is
```

```
  -- Define the possible states of the state machine
```

```
  type state_type is (s0, s1, s2, s3, s4) ;
```

```
  signal present_state, next_state : state_type ;
```

```
begin
```

```
  registers : process (clk, reset)
```

```
  begin
```

```
    -- process to update the present state
```

```
    if (reset='1') then
```

```
      present_state <= s0 ;
```

```
    elsif clk'event and clk = '1' then
```

```
      present_state <= next_state ;
```

```
    end if ;
```

```
  end process ;
```

Enumerate Type

State F/F



## State Machine Example (cont.)



transitions :

```
process (present_state, refresh, cs)
```

```
begin
```

```
-- process to calculate the next state
```

```
-- and the outputs
```

```
case present_state is
```

```
when s0 =>
```

```
  ras <= '1'; cas <= '1'; ready <= '1';
```

```
  if (refresh = '1') then
```

```
    next_state <= s3 ;
```

```
  elsif (cs = '1') then
```

```
    next_state <= s1 ;
```

```
  else
```

```
    next_state <= s0 ;
```

```
  end if ;
```

```
when s1 =>
```

```
  ras <= '0'; cas <= '1'; ready <= '1';
```

```
  next_state <= s2 ;
```

```
when s2 =>
```

```
  ras <= '0'; cas <= '0'; ready <= '0';
```

```
  if (cs = '0') then
```

```
    next_state <= s0 ;
```

```
  else
```

```
    next_state <= s2 ;
```

```
  end if ;
```

```
when s3 =>
```

```
  ras <= '1'; cas <= '0'; ready <= '0';
```

```
  next_state <= s4 ;
```

```
when s4 =>
```

```
  ras <= '0'; cas <= '0'; ready <= '0';
```

```
  next_state <= s0 ;
```

```
end case ;
```

```
end process ;
```

```
end a_ras_cas ;
```

## Lab. Exercise



### PREP #3

### Small Statemachine

## Lab. Exercise



### **PREP #4** **State machine**

## Arithmetic and Relational Logic







# Overview



## VHDL Synthesizer Generate Arithmetic Logic

- ◆ Technology Specific Macros
- ◆ Use Optimized Library of Parametized Module
- ◆ Synthesizable operators : "+", "-", "\*\*", "abs"
- ◆ "-" apply same as "+"
- ◆ "=", "/=", "<", ">", "<=", ">=" generate comparator

## Avoid explosion of combinational logic

- ◆ Constant operand
- ◆ Division("/") by power of two (Shifter)
- ◆ Synthesizable exponentiation("\*\*"), if both constant operands

## Integer Operation

- ◆ Bit-width of generated arithmetic circuit is depend on operand's range defined
- ◆ 2's complement implementation if integer range extends below 0
- ◆ unsigned implementation, when positive range

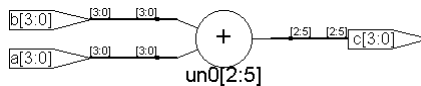


# Arithmetic Logic (RTL View)



```
entity add_int8 is
  port(
    a, b : in integer range 0 to 15;
    c : out integer range 0 to 15
  );
end add_int8;
```

```
architecture behave of add_int8 is
begin
  c <= a + b;
end behave;
```



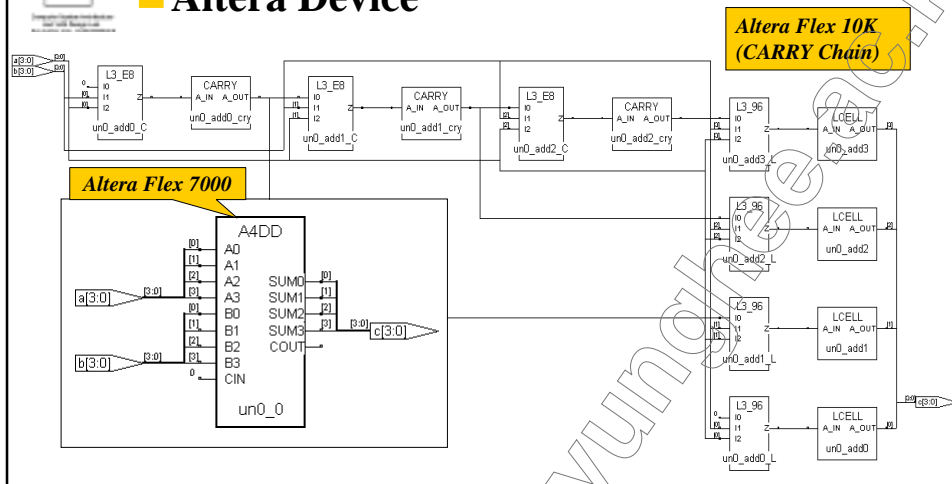
Synthesis RTL View



# Arithmetic Logic (Tech. View)



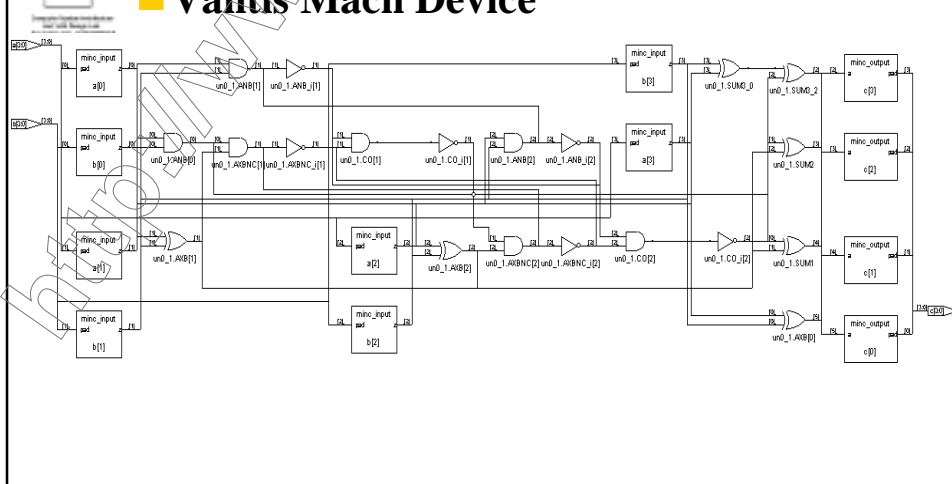
## Altera Device



# Arithmetic Logic (Tech. View)



## Vantis Mach Device





## Integer Division Example (1)



```

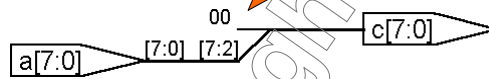
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity div_posint is
  port(a : in integer range 0 to 255;
        c : out integer range 0 to 255 );
end div_posint;

architecture behave of div_posint is
begin
  c <= a / 4;
end behave;

```

Hardwired Shift



## Integer Division Example (2)



```

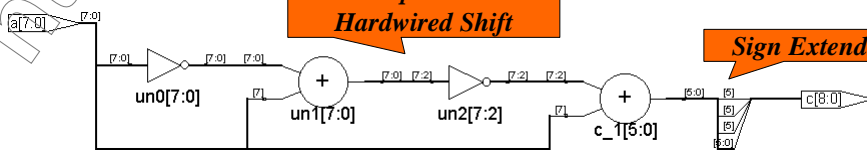
entity div_cmpl is
  port(a : in integer range -128 to 127;
        c : out integer range -256 to 255 );
end div_cmpl;

architecture behave of div_cmpl is
begin
  c <= a / 4;
end behave;

```

2's Complement and Hardwired Shift

Sign Extend



## Lab. Exercise



### Arithmetics

## Improve Synthesis



**Module Generation**  
**Resource Sharing**  
**Ranged Integer**  
**Design Optimization**



## Module Generation



- **Optimized generic modules (Parametized Library Modules)**
  - ◆ Memory Module : ROM, RAM
  - ◆ Counters
  - ◆ Arithmetic and Relational logic
- **Technology Specific Macros**
- **Mega/Macro Functions**
- **Most synthesizer tools support Technology specific macros and/or Module Generation options**



## LPM Package Example (Altera)



```
component LPM_ADD_SUB
generic (
  LPM_WIDTH: positive;
  LPM_REPRESENTATION: string := SIGNED;
  LPM_DIRECTION: string := UNUSED;
  LPM_HINT : string := UNUSED;
  LPM_PIPELINE : integer := 0;
  LPM_TYPE: string := L_ADD_SUB );
port (
  DATAA: in std_logic_vector(LPM_WIDTH-1 downto 0);
  DATAB: in std_logic_vector(LPM_WIDTH-1 downto 0);
  ACLR : in std_logic := '0';
  CLOCK : in std_logic := '1';
  CIN: in std_logic := '0';
  ADD_SUB: in std_logic := '1';
  RESULT: out std_logic_vector(LPM_WIDTH-1 downto 0);
  COUT: out std_logic;
  OVERFLOW: out std_logic );
end component;
```



## Device Specific Synthesis



- Specifying place-and-route constraints, etc. can be tricky in an HDL.
- Examples: pin assignments, critical nets, clock buffers, etc.
- Hierarchy can be used to reference non-VHDL elements in some tools.
- No standards have been established; follow your synthesis documentation!



## Pin Assignments



```
Entity CONTROL Is
Port (Reset, Clk, Mode, VS, ENDFR: in std_ulogic;
      RAMWE, RAMOE: out std_ulogic;
      ADOE, INCAD: out std_ulogic);
attribute pinnum: string;
attribute pinnum of Clk: signal is "1"; -- Device Specific !
attribute pinnum of Mode: signal is "2";
attribute pinnum of Data: signal is "37,23,22,21,20,19,18,17";
attribute pinnum of Addr: signal is "29,28,27,26,16,25,24,12";
attribute pinnum of RAMWE: signal is "32";
attribute pinnum of RAMOE: signal is "33";
attribute pinnum of ADOE: signal is "34";
End CONTROL;
```



## Device-specific Macros



architecture structure of lca\_macro is

*-- Refer to vendor supplied Libraries*

```
component RAM16x1
  port (A0, A1, A2, A3, WE, D: in STD_LOGIC;
        O: out STD_LOGIC);
end component;
begin
  ...
  RAM1: RAM16x1 port map (A(0),A(1),A(2),A(3),WE,
                          DATAIN, DATAOUT);
  ...
end structure;
```



## Resource Share (use Temp signal)



```
process(a,b,c,d)
begin
  if ( a+b=c ) then
    e <= a;
  elsif ( a+b=d ) then
    e <= b;
  else
    e <= c;
  end if;
end process;
```

```
process(a,b,c,d)
variable temp : integer range 0 to 15;
begin
  temp := a + b;
  if ( temp=c ) then
    e <= a;
  elsif ( temp=d ) then
    e <= b;
  else
    e <= c;
  end if;
end process;
```



## Resource Share (use parenthesis)



```

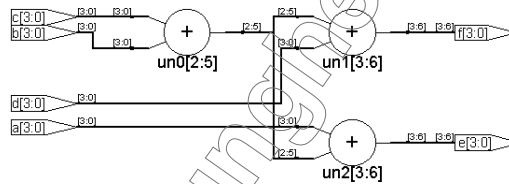
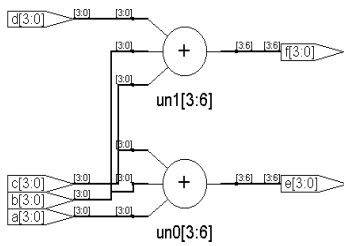
process(a,b,c,d)
begin
  e <= a + b + c;
  f <= b + c + d;
end process;

```

```

process(a,b,c,d)
begin
  e <= a + (b + c);
  f <= (b + c) + d;
end process;

```



## Resource Share (mutual exclusive)



```

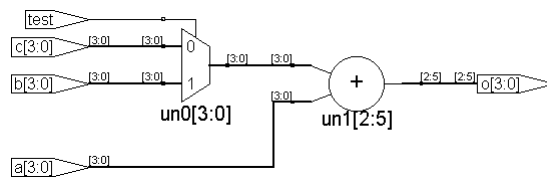
process(a,b,c,test)
begin
  if (test=TRUE) then
    o <= a + b;
  else
    o <= a + c;
  end if;
end process;

```

```

process(a,b,c,test)
variable temp : integer range 0 to 15;
begin
  if (test=TRUE) then
    temp := b;
  else
    temp := c;
  end if;
  o <= a + temp;
end process;

```







## Ranged Integer



- Integer type : 32-bit default

```
variable a, b, c : integer;  
c := a + b; -- 32-bit adder
```

- Ranged integer

```
variable a, b, c : integer range 0 to 255;  
c := a + b; -- 8-bit adder
```



## Advanced Design Optimization



- Change functionality without violating design spec.
- Understand VHDL and Circuit generated by Synthesizer
- Example : Loadable counter

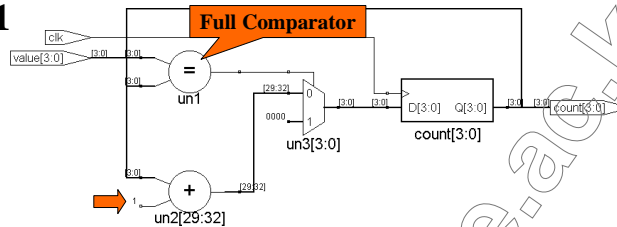
```
L_count1 :  
process  
begin  
wait until clk'event and clk='1';  
if (count = value) then  
count <= 0;  
count_done <= TRUE;  
else  
count <= count + 1;  
count_done <= FALSE;  
end if;  
end process;
```

```
L_count2 :  
process  
constant ZERO : integer range 0 to 15 := 0;  
begin  
wait until clk'event and clk='1';  
if (count = ZERO) then  
count <= value;  
count_done <= TRUE;  
else  
count <= count - 1;  
count_done <= FALSE;  
end if;  
end process;
```

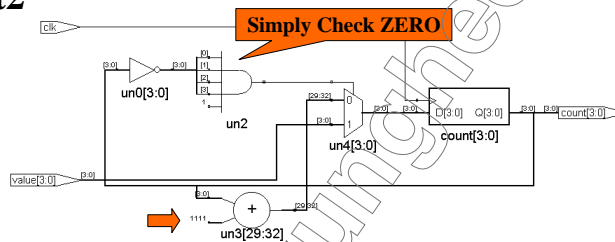


## Loadable counter

### L\_count1



### L\_count2



## Multiplexer and selectors

### Use case statement

```

case test_vector is
  when "000" => o <= bus(0) ;
  when "001" | "010" | "100" => o <= bus(1) ;
  when "011" | "101" | "110" => o <= bus(2) ;
  when "111" => o <= bus(3) ;
end case;

```

### Use variable indexed array

```

signal vec : std_logic_vector (0 to 15) ;
signal o : std_logic ;
signal i : integer range 0 to 15 ;
...
o <= vec(i) ;

```

*Equiv.*

```

case i is
  when 0 => o <= vec(0) ;
  when 1 => o <= vec(1) ;
  when 2 => o <= vec(2) ;
  when 3 => o <= vec(3) ;
  ...
end case ;

```

## Lab. Exercise



## Resource Sharing



## ROMs, PLAs and Decoders

### ■ ROM

- ◆ Address : case condition
- ◆ Data : case statement

```
case address is
  when "0000" => data <= "0111111";
  when "0001" => data <= "0011000";
  when "0010" => data <= "1101101";
  when "0011" => data <= "1111100";
  when "0100" => data <= "1011010";
  when "0101" => data <= "1110110";
  when "0110" => data <= "1110110";
  when "0111" => data <= "0011100";
  when "1000" => data <= "1111111";
  when "1001" => data <= "1111110";
  when "1010" => data <= "1011111";
  when "1011" => data <= "1110011";
  when "1100" => data <= "0100111";
  when "1101" => data <= "1111001";
  when "1110" => data <= "1100111";
  when "1111" => data <= "1000111";
end case;
```



## Seven-Segment Decoder



```

type seven_segment is array (6 downto 0) ;
type rom_type is array (natural range <=>) of seven_segment ;
constant hex_to_7 : rom_type (0 to 15) :=
  ("0111111", -- 0
   "0011000", -- 1
   "1101101", -- 2
   "1111100", -- 3
   "1011010", -- 4
   "1110110", -- 5
   "1110111", -- 6
   "0011100", -- 7
   "1111111", -- 8
   "1111110", -- 9
   "1011111", -- A
   "1110011", -- B
   "0100111", -- C
   "1111001", -- D
   "1100111", -- E
   "1000111"); -- F

```

Display segment index numbers :

```

      2
     1 3
      6
     0 4
      5

```

*-- Now, the ROM field can be accessed via a integer index*  
 display\_bus <= hex\_to\_7 (i) ;



## PLA and ROM



### ■ ROM

- ◆ fixed input decoder Plane-as address
- ◆ all cased case statement

### ■ PLA

- ◆ Input allows don't care condition "-","x"
- ◆ Cannot use case statement

### ■ Use Modeled ROM/PLA for large size design



## 2-Dimensional PLA



```
type std_logic_pla is array (natural range <>, natural range <>) of std_logic;
```

```
.....
```

```
procedure pla_table (
```

```
    constant invec : in std_logic_vector;
```

```
    signal outvec : out std_logic_vector;
```

```
    constant table : in std_logic_pla
```

```
) is
```

```
variable x : std_logic_vector (table'range(1)) ; -- product lines
```

```
variable y : std_logic_vector (outvec'range) ; -- outputs
```

```
variable b : std_logic ;
```

```
begin
```

```
    assert (invec'length + outvec'length = table'length(2))
```

```
    report "Size of Inputs and Outputs do not match table size"
```

```
    severity ERROR ;
```



## 2-Dimensional PLA (cont'ed)



```
-- Calculate the AND plane
```

```
x := (others => '1');
```

```
for i in table'range(1) loop -- PLA Table ROWs
```

```
    for j in invec'range loop
```

```
        b := table (i, table'left(2)-invec'left+j) ;
```

```
        if (b='1') then
```

```
            x(i) := x(i) AND invec (j) ;
```

```
        elsif (b='0') then
```

```
            x(i) := x(i) AND NOT invec(j) ;
```

```
        end if ;
```

```
-- If b is not '0' or '1' (e.g. '-') product line is insensitive to invec(j)
```

```
    end loop ;
```

```
end loop ;
```



## 2-Dimensional PLA (cont'ed)



```
-- Calculate the OR plane
y := (others=>'0') ;
for i in table'range(1) loop
  for j in outvec'range loop
    b := table(i,table'right(2) - outvec'right+j) ;
    if (b='1') then
      y(j) := y(j) OR x(i);
    end if ;
  end loop ;
end loop ;
outvec <= y ;

end pla_table ;
```



## 2-Dimensional PLA (cont'ed)



```
constant pos_of_fist_one : std_logic_pla (4 downto 0, 6 downto 0) :=
  ("1- - - 000", -- first '1' is at position 0
   "01- - 001", -- first '1' is at position 1
   "001- 010", -- first '1' is at position 2
   "0001011", -- first '1' is at position 3
   "0000111" ); -- There is no '1' in the input
signal test_vector : std_logic_vector (3 downto 0) ;
signal result_vector : std_logic_vector (2 downto 0) ;
...
-- Now use the pla table procedure with PLA pos_of_fist_one
-- test_vector is the input of the PLA, result_vector the output.
...
pla_table ( test_vector, result_vector, pos_of_fist_one) ;
```

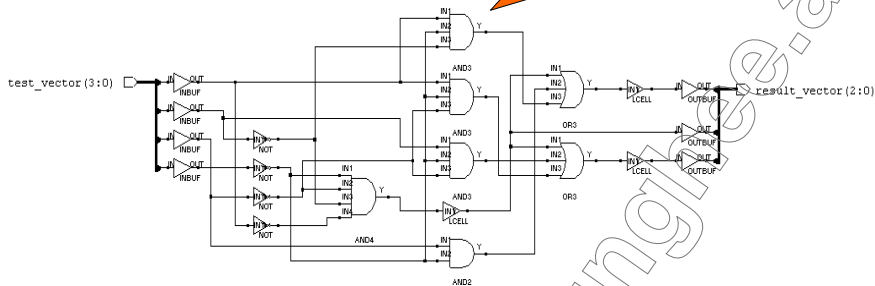


## 2-Dimensional PLA Synthesis (1)



■ Altera 7000

Synthesis as glue logic

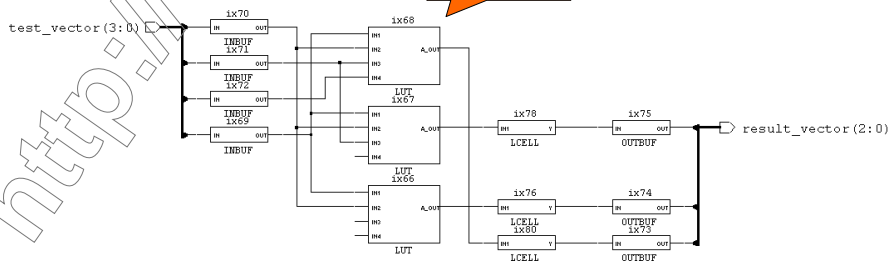


## 2-Dimensional PLA Synthesis (2)



■ Altera 10k

LUTs



## Lab. Exercise



## 2-Dimensional PLA

## Synthesis pitfalls





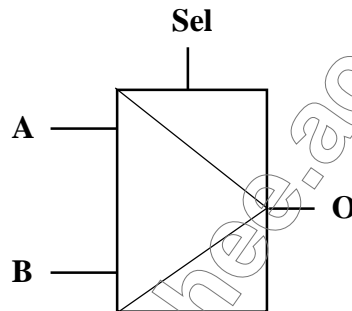


## Combinational Logic



### IF-ELSE Statement:

```
process (A,B,Sel)
begin
  if (Sel = '1') then
    O <= A;
  else
    O <= B;
  end if;
end process;
```



**No problem here, but...**

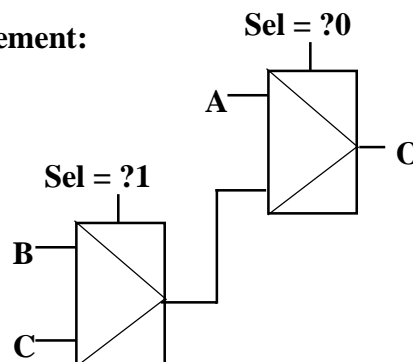


## Combinational Logic (cont)



### IF-ELSIF-ELSE Statement:

```
process (A,B,C,Sel)
begin
  if (Sel = '00') then
    O <= A;
  elsif (Sel = '01') then
    O <= B;
  else
    O <= C;
  end if;
end process;
```



**Be careful with priorities!**

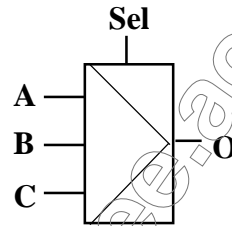


## Combinational Logic (cont)



### ■ CASE Statement:

```
process (A,B,C,Sel)
begin
  case Sel is
    when '00' =>
      O <= A;
    when '01' =>
      O <= B;
    when others =>
      O <= C;
  end case;
end process;
```



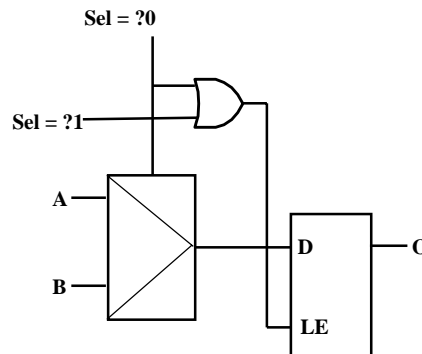
**When in doubt, use a case statement!**



## Accidental Latches



```
process (A,B,Sel)
begin
  if (Sel = '00') then
    O <= A;
  elsif (Sel = '10') then
    O <= B;
  end if;
end process;
```



**Don't leave out any conditions or you may get latches!**



## Accidental Latches (cont)



architecture behavior of dff is

```
begin
  process(Rst, Clk)
  begin
    if Rst = '1' then
      Q1 <= (others=>'0');
    elsif rising_edge(Clk) then
      Q1 <= D1;
      Q2 <= D2; -- Note: should generate error
    end if;
  end process;
end behavior;
```

*Hint: what happens to Q2 when Rst is held high on a rising edge of Clk?*



## Accidental Latches (cont)



```
process (current_state)
begin
  case current_state is
    when S0 =>
      O1 <= ?; O2 <= ?;
    when S1 =>
      O1 <= ?; O2 <= ?;
    when S2 =>
      O1 <= ?; -- O2 doesn't matter in state S2 or S3
    when S3 =>
      O1 <= ?;
  end case;
end process;
```



## Accidental Latches (cont)



```

process (current_state)
begin
  O2 <= ?;      -- Be safe! Set default values here.
  case current_state is
    when S0 =>
      O1 <= ?; O2 <= ?;
    when S1 =>
      O1 <= ?; O2 <= ?;
    when S2 =>
      O1 <= ?;      -- O2 doesn't matter in state S2 or S3
    when S3 =>
      O1 <= ?;
  end case;
end process;

```



## Q & Qbar



```

ENTITY qqbar IS
  PORT ( data, clk : IN  std_logic;
         q, qbar  : OUT std_logic );
END qqbar;

```

```

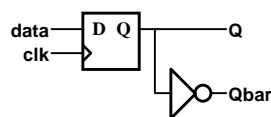
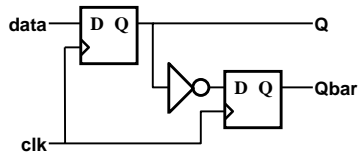
ARCHITECTURE a_qqbar OF qqbar IS
BEGIN
  PROCESS (data, clk)
  BEGIN
    IF clk='1' AND clk'EVENT THEN
      q <= data;
      qbar <= NOT q;
    END IF;
  END PROCESS;
END a_qqbar;

```

```

ARCHITECTURE a_qqbar_1 OF qqbar IS
BEGIN
  PROCESS (data, clk)
  BEGIN
    IF clk='1' AND clk'EVENT THEN
      q <= data;
    END IF;
  END PROCESS;
  qbar <= NOT q;
END a_qqbar_1;

```





## Unsynthesizable Functions



```
function Vec_To_Int (INVEC : bit_vector) return integer is
  variable result: integer := 0;
begin
  for i in INVEC'reverse_range loop
    result := result * 2;
    if (INVEC(i) = '1') then
      result <= result + 1;
    end if;
  end loop;
  return result;
end Vec_To_Int;
```

*Theoretically synthesizable (check, it's just a wire) but you'll never get it through the synthesis tool!*



## UnSynthesizable VHDL



- After clauses (delay specifications)
- Initialization values
- Signals that have multiple clocks
- Unconstrained loops
- Text I/O (File types)
- Access types
- Real types (floating point numbers)



## UnSynthesizable Verilog



- User-defined-primitives (UDPs)
- Initial statements
- Multiple clocks
- Pull-downs, pull-ups, etc.
- Forever statements and other infinite loops
- Force, release, etc.
- Join and fork parallel blocks
- Specify blocks, etc.



## Initialization Values



```
type state_type is (S0,S1,S2,S3,S4) ;  
signal current_state: state_type := S0 ;
```

- Simulation will start in state S0
- Synthesis will ignore initialization, so you get the device powerup for the initial state
- Summary: always provide a reset!



## Multiple Clocks



```
Process(clk0, clk1)
begin
  if clk0'EVENT and clk0='1' then
    q <= d0;
  elsif clk1'EVENT and clk1='1' then
    q <= d1;
  end if;
end process;
```

```
Process(clk0)
begin
  if clk0'EVENT and clk0='1' then
    q <= d0;
  end if;
end process;

Process(clk1)
begin
  if clk1'EVENT and clk1='1' then
    q <= d1;
  end if;
end process;
```

- Not synthesizable
- Rewrite with multiple registers



## What We're Learned



- Seen how VHDL and Verilog fit in the overall design process
- Covered the basics of the languages
- Examined some common synthesis-related conventions
- Learned about some common synthesis-related mistakes



## Suggested Reading



- IEEE, *IEEE Standard 1076-1993 VHDL Language Reference Manual*
- IEEE, *IEEE Standard 1364 Verilog Language Reference Manual*
- Pellerin, David and Douglas Taylor, *VHDL Made Easy*, Prentice Hall, 1996
- Armstrong and Gray, *Structured logic Design with VHDL*, Prentice-Hall, 1996
- Internet: [comp.lang.vhdl](http://comp.lang.vhdl), [comp.lang.verilog](http://comp.lang.verilog), <http://www.vhdl.org>

<http://www.csvlsi.kyungheec.ac.kr>