

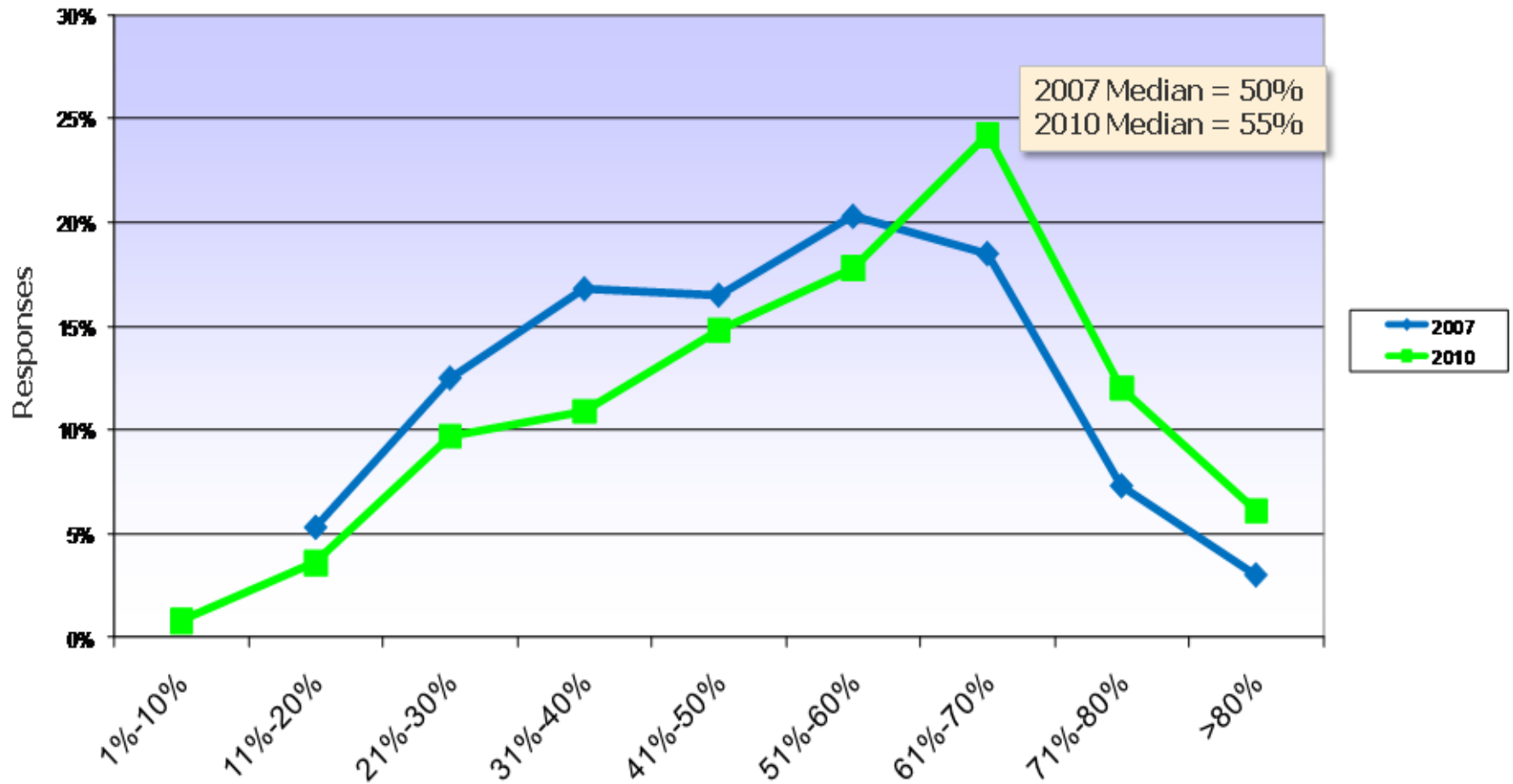
CDA 4253/CIS 6930 FPGA System Design

VHDL Testbench Development

Hao Zheng
Comp. Sci & Eng
University of South Florida

Effort Spent On Verification

Trend in the percentage of total project time spent in verification

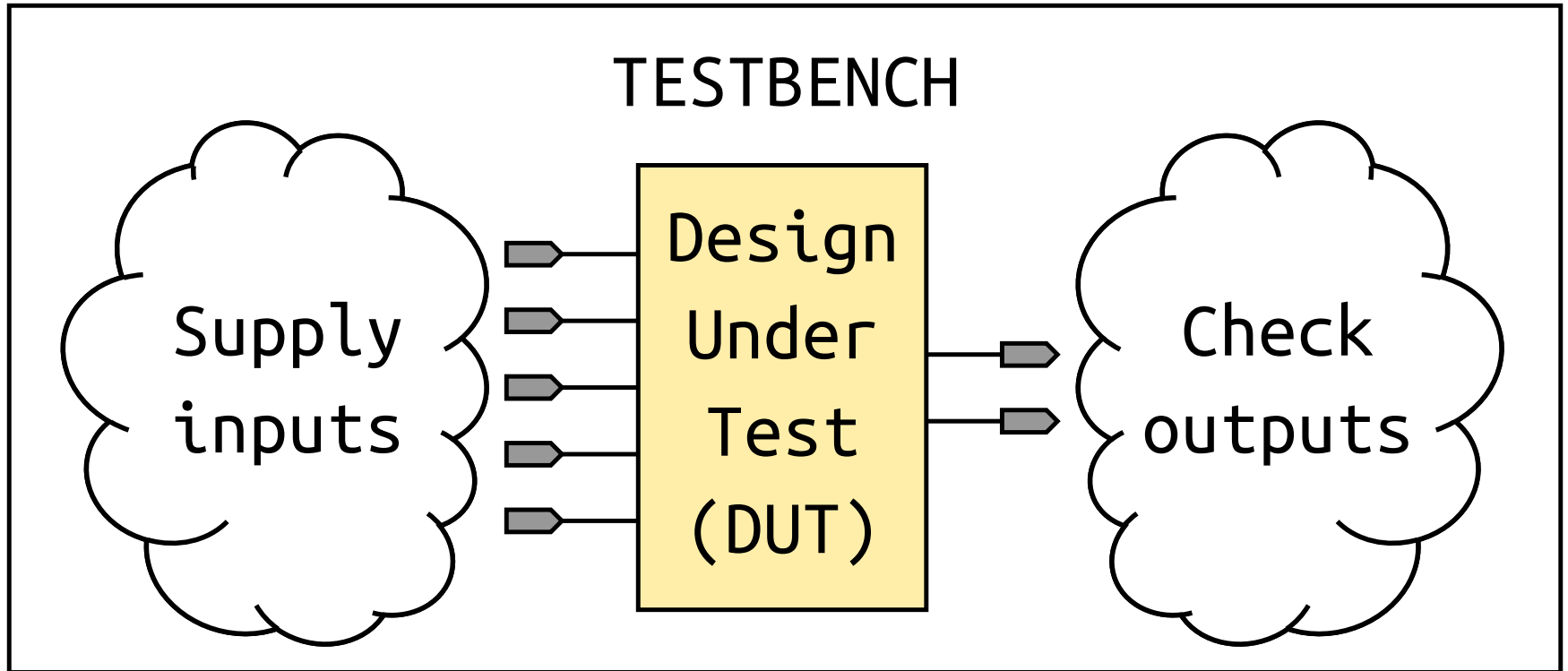


> 70% projects spent > 40% time in verification

Validation, Verification, and Testing

- **Validation**: Does the product meet customers' wishes?
 - Am I building the right product?
- **Verification**: Does the product meet the specification?
 - Am I building the product right?
 - Debugging begins when error is detected
- **Testing**: Is chip fabricated as meant to?
 - No short-circuits, open connects, slow transistors etc.
 - Post-manufacturing tests at the silicon fab
 - Accept/Reject
- Often these are used interchangeably
 - E.g., both terms are used: DUT (design under test) and DUV (design under verification)

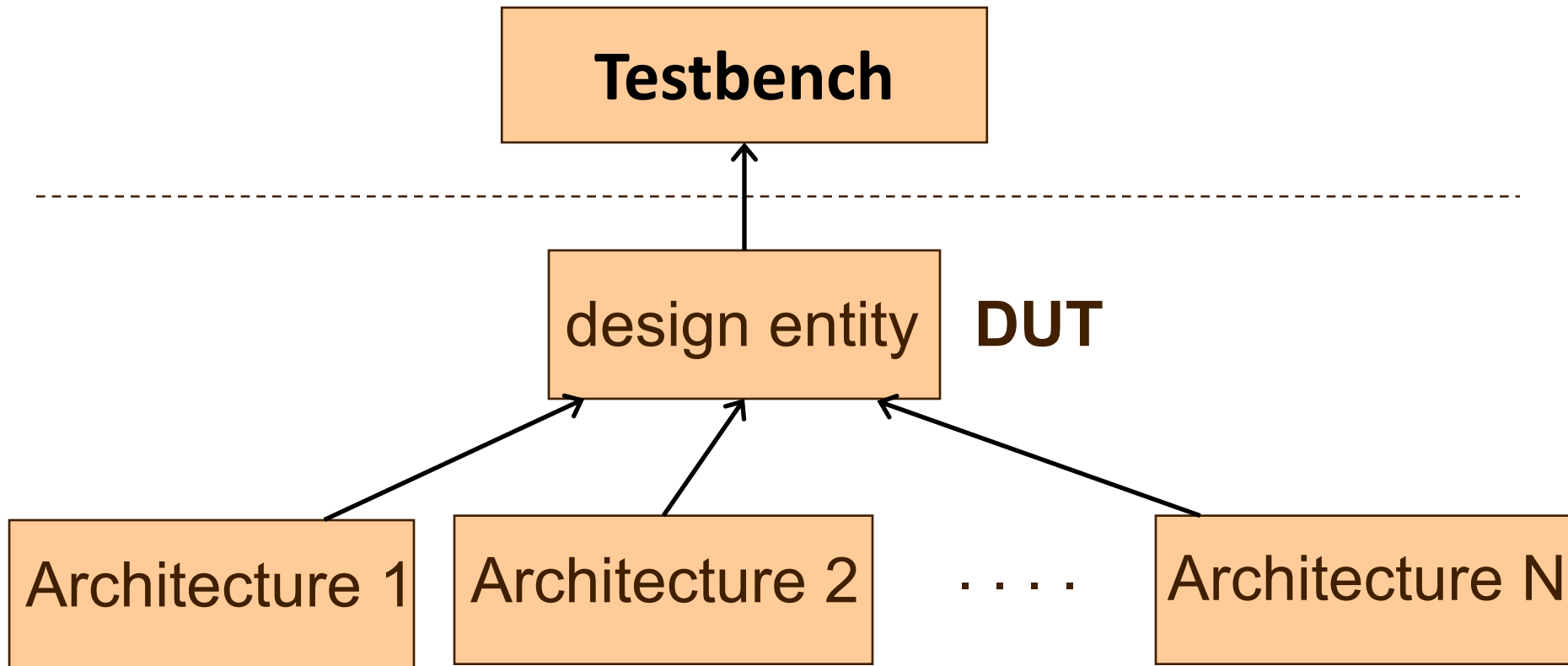
Basic Testbench Architecture



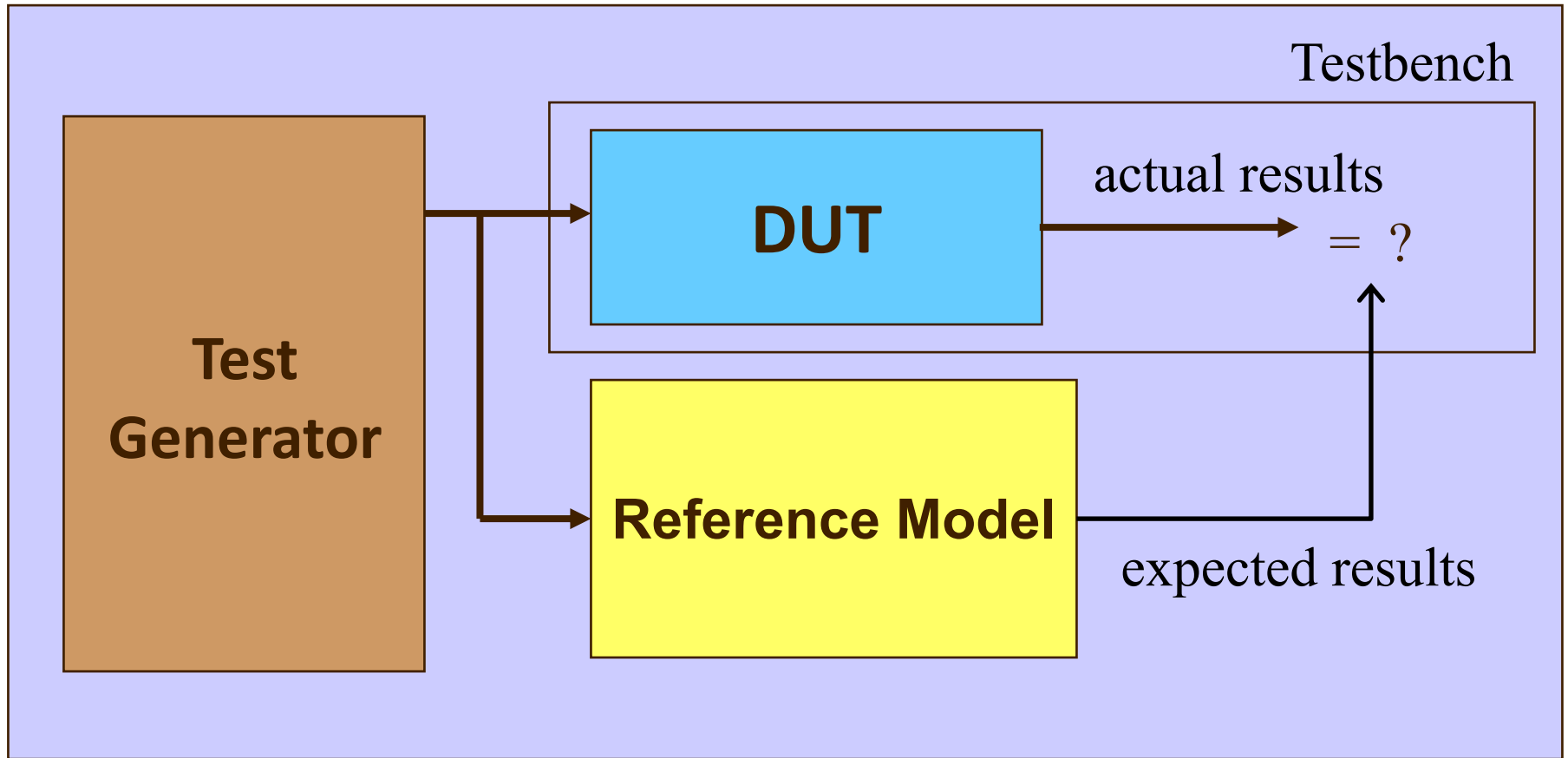
Testbench Defined

- *Testbench* = VHDL entity that applies stimuli (drives the inputs) to the Design Under Test (DUT) and (optionally) verifies expected outputs.
- The results can be viewed in a waveform window or written to a file.
- Since *Testbench* is written in VHDL, it is not restricted to a single simulation tool (portability).
- The same *Testbench* can be easily adapted to test different implementations (i.e. different *architectures*) of the same design.

The same testbench can be used to test multiple implementations of the same circuit (multiple architectures)



Possible sources of expected results used for comparison



Testbench Anatomy

```
ENTITY my_entity_tb IS
    --TB entity has no ports
END my_entity_tb;

ARCHITECTURE behavioral OF tb IS
    --Local signals and constants
BEGIN
    DUT:entity work.TestComp PORT MAP( -- Instantiations of DUTs
        );

    test_vector: PROCESS } -- Input stimuli
    END PROCESS;

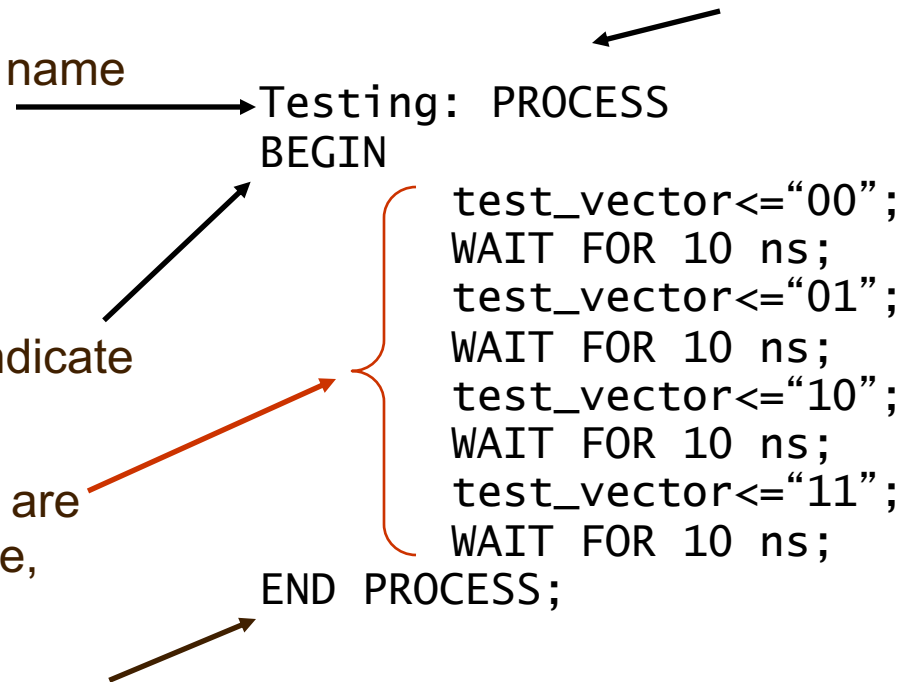
    monitor: process
    -- monitor and check the outputs from DUT
    end process;
END behavioral;
```


Process without Sensitivity List and its use in Testbenches

What is a PROCESS?

→ A process is a sequence of instructions referred to as sequential statements.

- A process can be given a unique name using an optional LABEL
- This is followed by the keyword PROCESS
- The keyword BEGIN is used to indicate the start of the process
- All statements within the process are executed **SEQUENTIALLY**. Hence, order of statements is important.



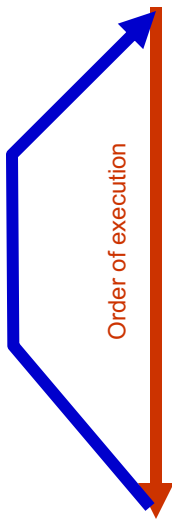
A process cannot have a sensitivity list and use wait statements

Execution of statements in a PROCESS

- The execution of statements continues sequentially till the last statement in the process.
- After execution of the last statement, the control is again passed to the beginning of the process.

```
Testing: PROCESS  
BEGIN
```

```
test_vector<="00";  
WAIT FOR 10 ns;  
test_vector<="01";  
WAIT FOR 10 ns;  
test_vector<="10";  
WAIT FOR 10 ns;  
test_vector<="11";  
WAIT FOR 10 ns;  
END PROCESS;
```



Order of execution

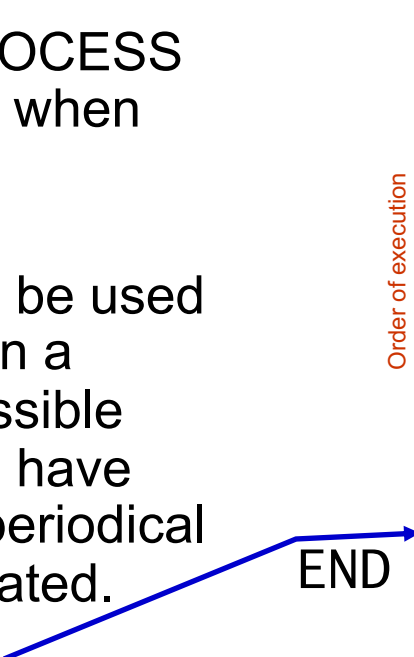
Program control is passed to the first statement after BEGIN

PROCESS with a WAIT Statement

- The last statement in the PROCESS is a **WAIT** instead of WAIT FOR 10 ns.
- This will cause the PROCESS to **suspend indefinitely** when the WAIT statement is executed.
- This form of WAIT can be used in a process included in a testbench when all possible combinations of inputs have been tested or a non-periodical signal has to be generated.

Testing: PROCESS
BEGIN

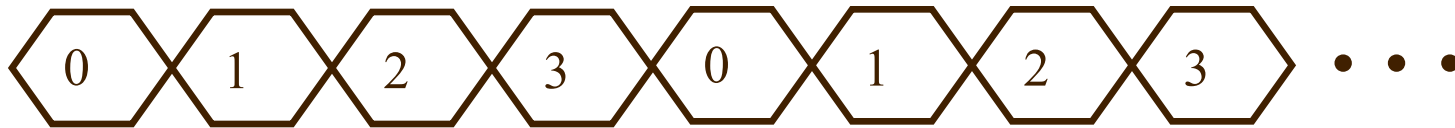
```
test_vector<="00";  
WAIT FOR 10 ns;  
test_vector<="01";  
WAIT FOR 10 ns;  
test_vector<="10";  
WAIT FOR 10 ns;  
test_vector<="11";  
WAIT;  
END PROCESS;
```



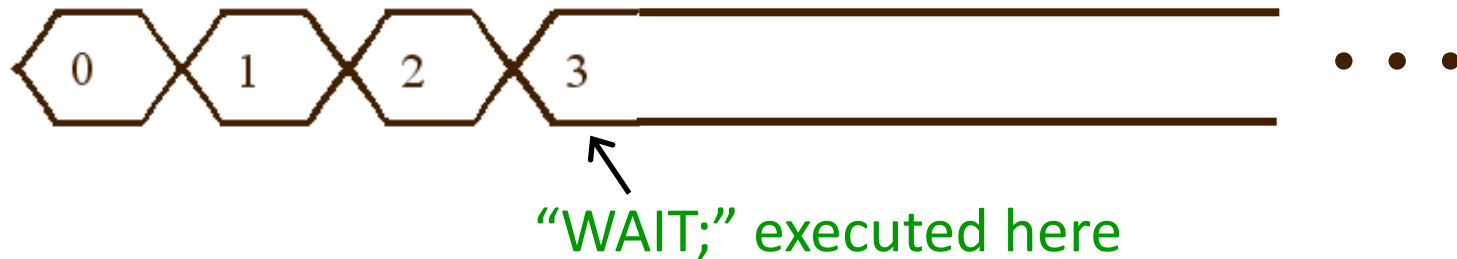
Program execution stops here

WAIT FOR vs. WAIT

WAIT FOR 10ns : waveform will keep repeating itself forever

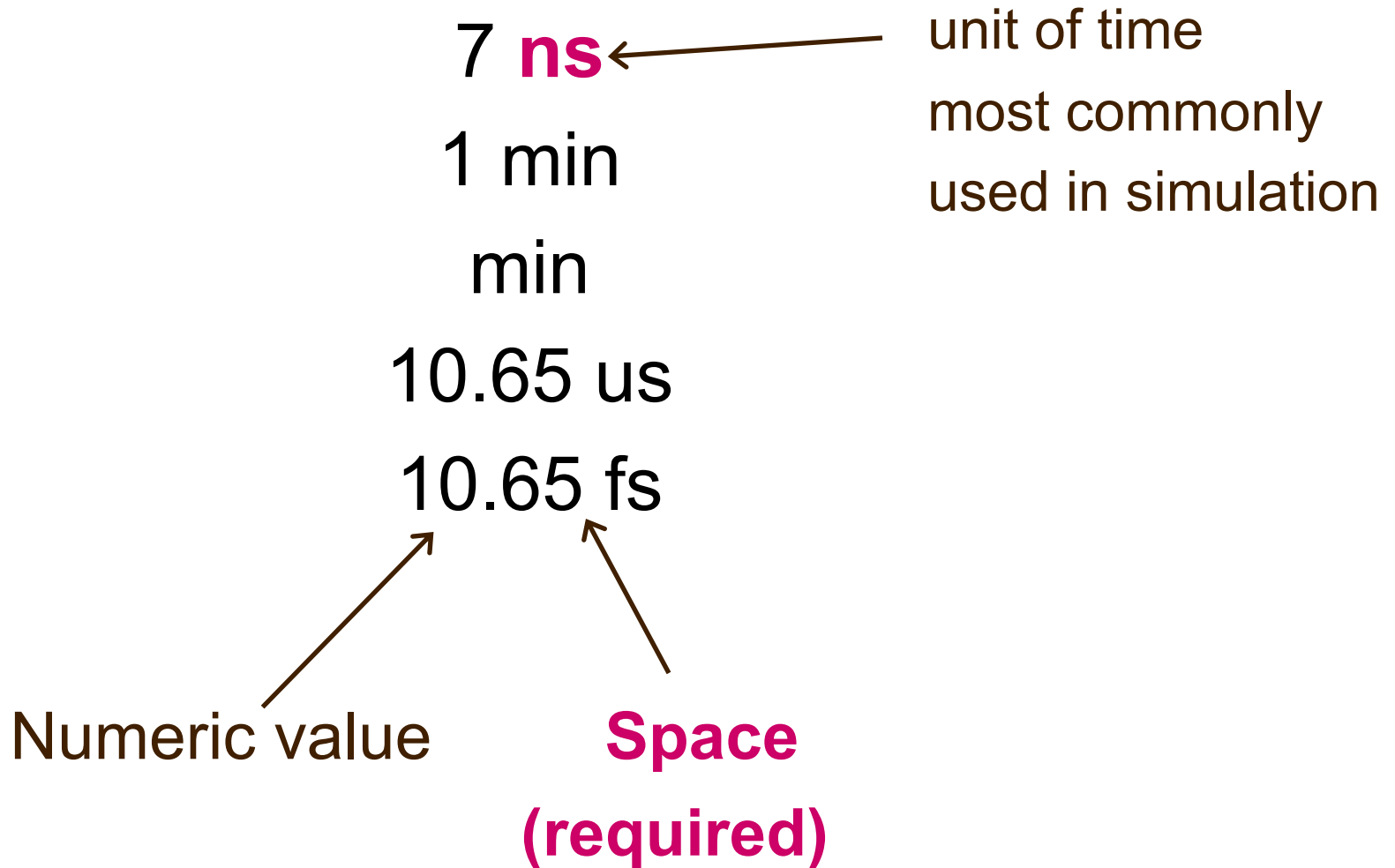


WAIT : waveform will keep its state after the last wait instruction.



Specifying time in VHDL

Time Values – Examples



Units of time

Unit

Definition

Base Unit

fs femtoseconds (10^{-15} seconds)

Derived Units

ps picoseconds (10^{-12} seconds)

ns nanoseconds (10^{-9} seconds)

us microseconds (10^{-6} seconds)

ms milliseconds (10^{-3} seconds)

sec seconds

min minutes (60 seconds)

hr hours (3600 seconds)

Simple Testbenches

Generating Clock Signal

```
CONSTANT clk1_period : TIME := 20 ns;  
CONSTANT clk2_period : TIME := 200 ns;  
SIGNAL clk1 : STD_LOGIC;  
SIGNAL clk2 : STD_LOGIC := '0';
```

```
begin
```

```
    clk1_generator: PROCESS
```

```
    begin
```

```
        clk1 <= '0';
```

```
        WAIT FOR clk1_period/2;
```

```
        clk1 <= '1';
```

```
        WAIT FOR clk1_period/2;
```

```
    END PROCESS;
```

```
    clk2 <= not clk2 after clk2_period/2;
```

```
    .....
```

```
END behavioral;
```

Generate One-Time Signals – Reset

Architecture behavioral

```
CONSTANT reset1_width : TIME := 100 ns;
```

```
CONSTANT reset2_width : TIME := 150 ns;
```

```
SIGNAL reset1 : STD_LOGIC;
```

```
SIGNAL reset2 : STD_LOGIC := '1';
```

```
BEGIN
```

```
reset1_generator: process  
begin
```

```
    reset1 <= '1';
```

```
    WAIT FOR reset1_width;
```

```
    reset1 <= '0';
```

```
    WAIT;
```

```
end process;
```

```
END behavioral;
```

```
reset2_generator: process  
begin
```

```
    WAIT FOR reset2_width;
```

```
    reset2 <= '0';
```

```
    WAIT;
```

```
end process;
```

Test Vectors

Set of pairs: {Input Values i , Expected Outputs Values i }

Input Values 1, Expected Output Values 1

Input Values 2, Expected Output Values 2

.....

Input Values N , Expected Output Values N

Test vectors can cover either:

- all combinations of inputs (for very simple circuits only)
- selected representative combinations of inputs
(most realistic circuits)

Generating selected values of one input

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

BEGIN
    .....
    testing: PROCESS
    BEGIN
        test_vector <= "000";
        WAIT FOR 10 ns;
        test_vector <= "001";
        WAIT FOR 10 ns;
        test_vector <= "010";
        WAIT FOR 10 ns;
        test_vector <= "011";
        WAIT FOR 10 ns;
        test_vector <= "100";
        WAIT FOR 10 ns;
    END PROCESS;
    .....
END behavioral;
```

Generating all values of one input

```
USE ieee.std_logic_unsigned.all;
```

```
.....
```

```
SIGNAL test_vector : STD_LOGIC_VECTOR(2 downto 0) := "000";
```

```
BEGIN
```

```
.....
```

```
testing: PROCESS
```

```
BEGIN
```

```
    WAIT FOR 10 ns;
```

```
    test_vector <= test_vector + 1;
```

```
end process TESTING;
```

```
.....
```

```
END behavioral;
```

Generating all possible values of two inputs

```
USE ieee.std_logic_unsigned.all;
...
SIGNAL test_ab : STD_LOGIC_VECTOR(2 downto 0);
SIGNAL test_sel : STD_LOGIC_VECTOR(1 downto 0);
BEGIN
    .....
    double_loop: PROCESS
    BEGIN
        test_ab <="00";
        test_sel <="00";
        for I in 0 to 3 loop
            for J in 0 to 3 loop
                wait for 10 ns;
                test_ab <= test_ab + 1;
            end loop;
            test_sel <= test_sel + 1;
        end loop;
    END PROCESS;
    .....
END behavioral;
```

Checking Outputs

```
test_generator: PROCESS
  begin
    -- apply a test vector to inputs
    wait until rising_edge(clk1);
  END PROCESS;
```

```
Monitor: PROCESS
  begin
    wait until rising_edge(clk1);
    -- check the design output
  END PROCESS;
```

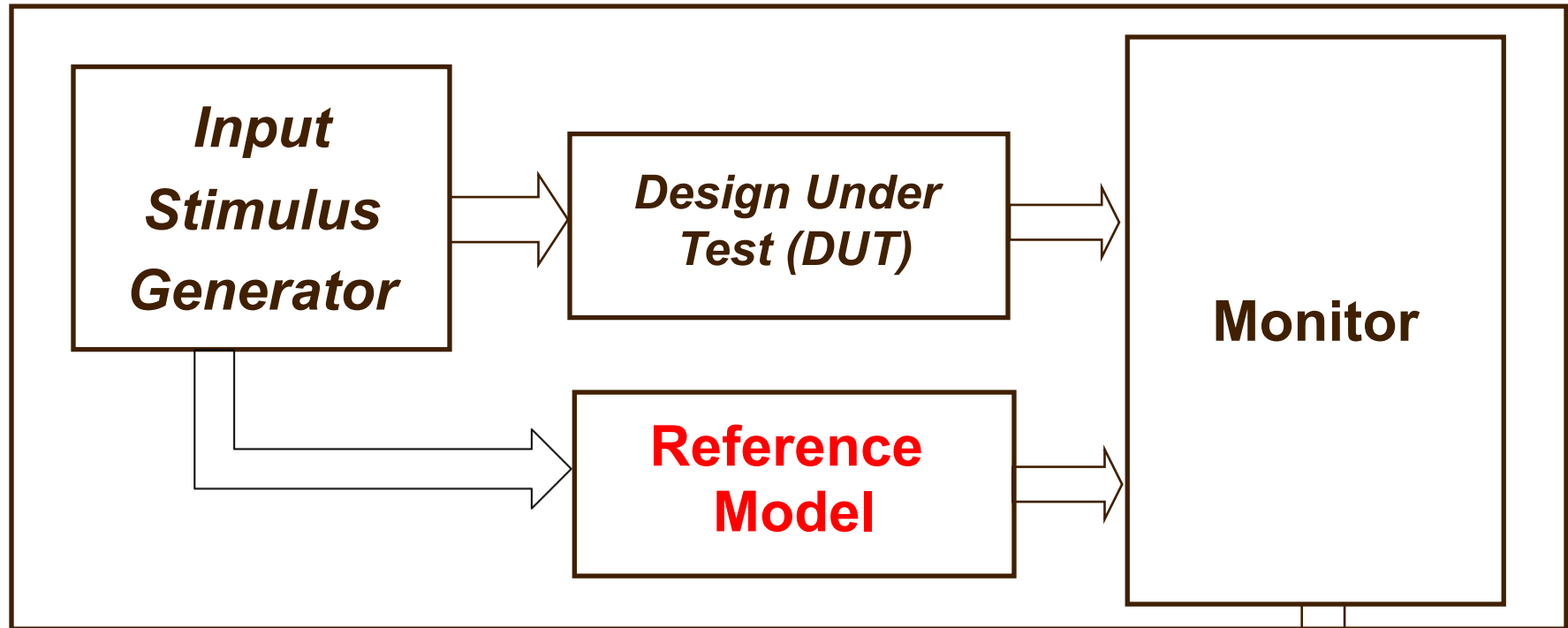

Example: Arbiter

```
test_generator: PROCESS
    variable req : unsigned(1 to 3) := 0;
begin
    r <= std_logic_vector(req);
    req := req + "001";
    wait until rising_edge(clk1);
END PROCESS;
```

```
Monitor: PROCESS
begin
    wait until rising_edge(clk1);
    -- check that at most one g is '1'
END PROCESS;
```

More Advanced Testbenches

More Advanced Testbenches



The reference model can be

- *A C program*
- *in VHDL*

Design
Correct/Incorrect

Test Generation – Records

TYPE test_vector **IS RECORD**

operation : STD_LOGIC_VECTOR(1 DOWNT0 0);

a : STD_LOGIC;

b : STD_LOGIC;

y : STD_LOGIC;

END RECORD;

CONSTANT num_vectors : INTEGER := 16;

TYPE test_vectors **IS ARRAY** (0 TO num_vectors-1) **OF** test_vector;

CONSTANT and_op : STD_LOGIC_VECTOR(1 DOWNT0 0) := "00";

CONSTANT or_op : STD_LOGIC_VECTOR(1 DOWNT0 0) := "01";

CONSTANT xor_op : STD_LOGIC_VECTOR(1 DOWNT0 0) := "10";

CONSTANT xnor_op : STD_LOGIC_VECTOR(1 DOWNT0 0) := "11";

Test Generation – Records

```
CONSTANT test_vector_table: test_vectors :=(  
  (operation => AND_OP,  a=>'0', b=>'0', y=>'0'),  
  (operation => AND_OP,  a=>'0', b=>'1', y=>'0'),  
  (operation => AND_OP,  a=>'1', b=>'0', y=>'0'),  
  (operation => AND_OP,  a=>'1', b=>'1', y=>'1'),  
  (operation => OR_OP,   a=>'0', b=>'0', y=>'0'),  
  (operation => OR_OP,   a=>'0', b=>'1', y=>'1'),  
  (operation => OR_OP,   a=>'1', b=>'0', y=>'1'),  
  (operation => OR_OP,   a=>'1', b=>'1', y=>'1'),  
  (operation => XOR_OP,  a=>'0', b=>'0', y=>'0'),  
  (operation => XOR_OP,  a=>'0', b=>'1', y=>'1'),  
  (operation => XOR_OP,  a=>'1', b=>'0', y=>'1'),  
  (operation => XOR_OP,  a=>'1', b=>'1', y=>'0'),  
  (operation => XNOR_OP, a=>'0', b=>'0', y=>'1'),  
  (operation => XNOR_OP, a=>'0', b=>'1', y=>'0'),  
  (operation => XNOR_OP, a=>'1', b=>'0', y=>'0'),  
  (operation => XNOR_OP, a=>'1', b=>'1', y=>'1')  
);
```

Test data can be generated externally, and read into TB at runtime.

Random Number Generator

- Impossible to enumerate all inputs.
- Need to simulate environment inputs
 - Their value and timing hard to define precisely.
- Use function **UNIFORM** to simulate randomness.

```
use ieee.math_real.all
```

```
UNIFORM(seed1, seed2, x)
```

```
-- returns a pseudo-random number x with uniform distribution
```

```
-- in (0.0, 1.0)
```

```
-- seed1 and seed2 are seed values in [1, 2147483562] and
```

```
-- [1, 2147483398], respectively.
```

Random Reset

```
library ieee;
use ieee.math_real.all;
...
architecture behavior of testbench is
begin
process
    constant delay_range : time := 10000 ns;
    variable rand_delay : time := 1 ns;
    variable seed1, seed2: positive; -- seed values for random generator
    variable rand: real;    -- random real-number value in range 0 to 1.0
begin
    reset <= '0';
    uniform(seed1, seed2, rand);    -- generate random number
    rand_delay := rand * delay_range;
    wait for rand_delay;
    reset <= '1'
    wait for 10 ns;
    reset <= '0';
end process;
end behavior;
```

Random Data

```
library ieee;
use ieee.math_real.all;
...
architecture behavior of testbench is
    signal d1, d2 : std_logic_vector(7 downto 0);
begin
    process
        constant data_range : integer := 255;
        variable seed1, seed2: positive;
        variable rand: real;
    begin
        wait until rising_edge(clk) and done = '1';
        uniform(seed1, seed2, rand);
        d1 <= rand * data_range;
        uniform(seed1, seed2, rand);
        d2 <= rand * data_range;
        start <= '1';
    end process;
end behavior;
```


Assert – Monitoring and Checking

- Used to create self-checking monitors
- Assert is a **non-synthesizable** statement whose purpose is to write out messages on the screen when problems are found during simulation.
- Depending on the **severity of the problem**, the simulator is instructed to continue simulation or halt.

Assert – Syntax

```
ASSERT condition -- must hold during entire simulation  
[REPORT "message"]  
[SEVERITY severity_level ];
```

The message is written when the condition is FALSE.

Severity_level can be:

Note, Warning, Error (default), or Failure.

Assert – Examples (1)

```
assert initial_value <= max_value  
    report "initial value too large"  
    severity error;
```

```
assert packet_length /= 0  
    report "empty network packet received"  
    severity warning;
```

```
assert reset = false  
    report "Initialization complete"  
    severity note;
```

Assert Example – Check Bin Div Results

```
Process(valid, dividend, divisor, ready, q, r)
  variable dvdend, dvsor : ...
Begin
  wait until valid = '1';
  dvdend := dividend;
  dvsor := divisor;
  wait until ready = '1';
  assert q = dvdend / dvsor and
         r = dvdend rem dvsor;
         report "division results are not correct"
         severity error;
end process;
```

Summary

- HW debug is difficult
 - Simulation offers full observability
- Test generation is key
 - Your design is as good as how it is tested
 - Use randomness to exercise design for high coverage
- Monitor/Checker allows automatic observation and checking
 - Help pinpoint sources of bugs temporally and spatially
 - Reference model captures correct behavior
 - Assertions define properties of correct behavior

Backup

Developing Effective Testbenches

Report – Syntax

```
REPORT "message"  
[SEVERITY severity_level ];
```

The message is always written.

Severity_level can be:

Note (default), Warning, Error, or Failure.

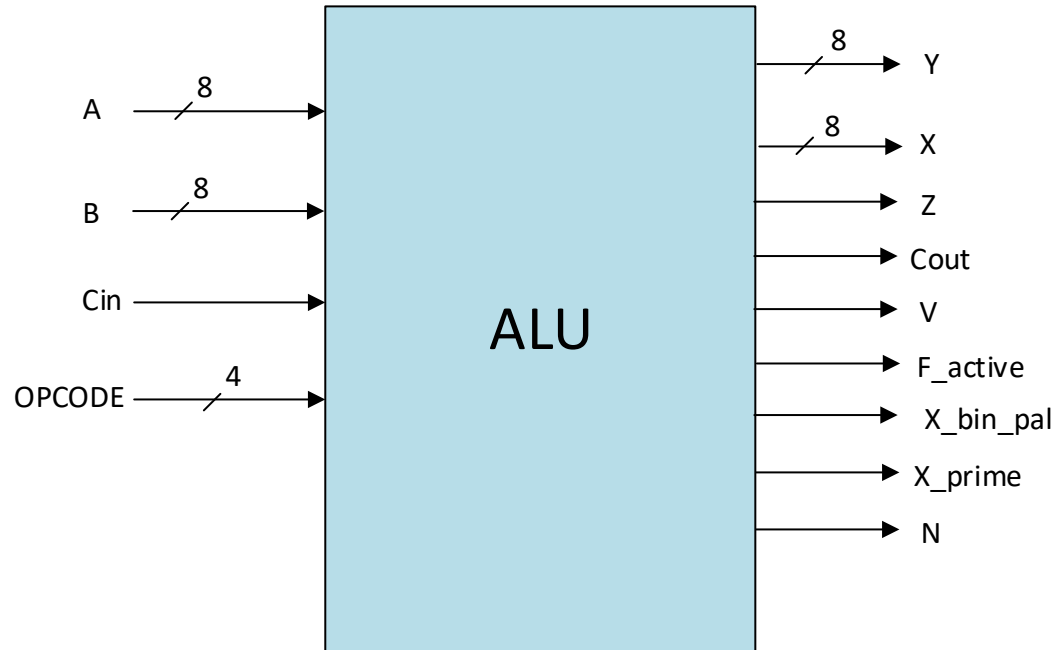
Report - Examples

```
report "Initialization complete";
```

```
report "Current time = " & time'image(now);
```

```
report "Incorrect branch" severity error;
```


Interface : Combinational Logic



Interface of an 8-bit ALU

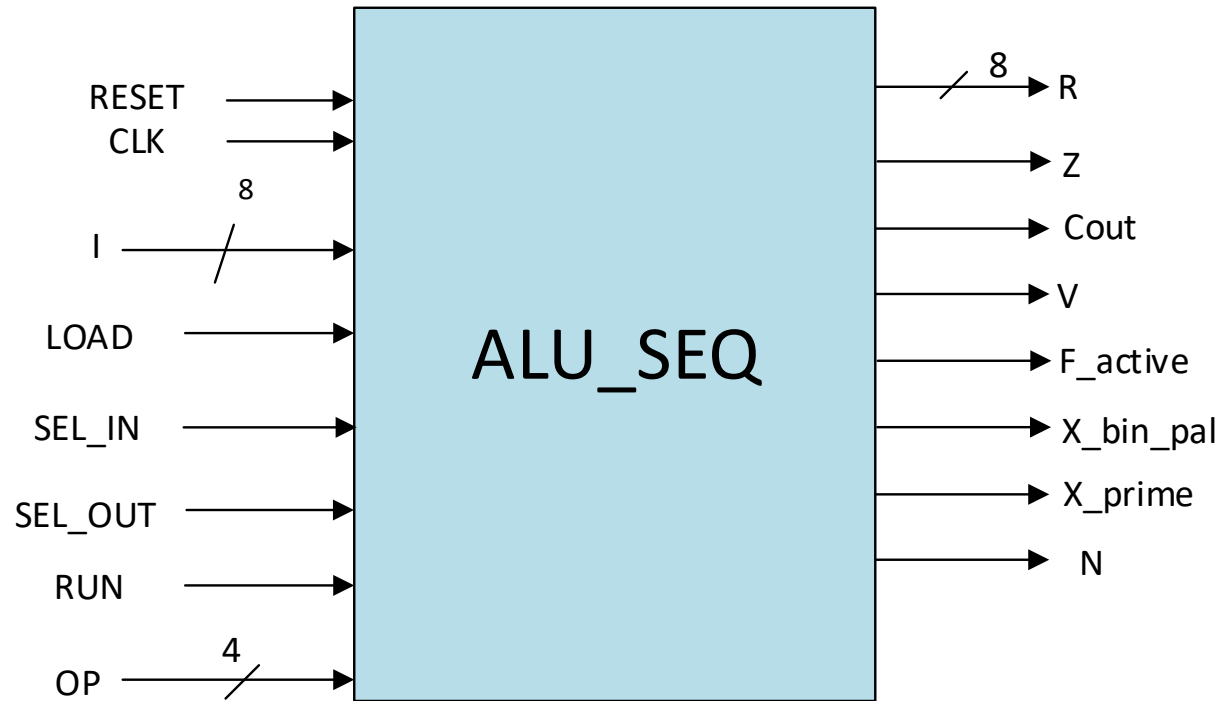
Ports

Name	Mode	Width	Meaning
A	IN	8	Input A
B	IN	8	Input B
Cin	IN	1	Carry In
OPCODE	IN	4	Operation Code
X	OUT	8	Output or Least Significant Byte of Output
Y	OUT	8	Most Significant Byte of Output or Zero
Z	OUT	1	Zero Flag
Cout	OUT	1	Carry out Flag
V	OUT	1	Overflow Flag
F_active	OUT	1	Logical OR of Z, Cout and V
X_bin_pal	OUT	1	Flag set to high when the output X is a binary palindromic number (numbers that remain the same when binary digits are reversed)
X_prime	OUT	1	Flag set to high when the output X is a prime number
N	OUT	1	Flag set to high when the output X is a negative number

Instruction Set

OPCODE	OPERATION	FORMULA	Z	Cout	V	X_bin_pal	X_prime	N
0000	AND	$X = A \text{ AND } B$	↕	–	–	↕	↕	0
0001	OR	$X = A \text{ OR } B$	↕	–	–	↕	↕	0
0010	XOR	$X = A \text{ XOR } B$	↕	–	–	↕	↕	0
0011	XNOR	$X = A \text{ XNOR } B$	↕	–	–	↕	↕	0
0100	Unsigned Addition	$(\text{Cout}:X) = A + B$	↕	↕	–	↕	↕	0
0101	Signed Addition	$X = A + B$	↕	–	↕	↕	↕	↕
0110	Unsigned Addition with Carry	$(\text{Cout}:X) = A + B + \text{Cin}$	↕	↕	–	↕	↕	0
0111	Signed Multiplication	$(Y:X) = A * B$	↕	–	–	↕	↕	↕
1000	Unsigned Multiplication	$(Y:X) = A * B$	↕	–	–	↕	↕	0
1001	Unsigned Subtraction	$X = A - B$	↕	↕	–	↕	↕	0
1010	Rotation Left	$X = A \lll 1$	↕	–	–	↕	↕	0
1011	Rotation Left with Carry	$(\text{Cout}:X) = (\text{Cin}:A) \lll 1$	↕	↕	–	↕	↕	0
1100	Logic Shift Right	$X = A \gg 1$	↕	↕	–	↕	↕	0
1101	Arithmetic Shift Right	$X = A \gg 1$	↕	↕	–	↕	↕	↕
1110	Logic Shift Left	$X = A \ll 1$	↕	↕	↕	↕	↕	↕
1111	BCD to Binary Conversion	$(Y:X) = \text{BCD2BIN}(B:A)$	↕	↕	–	↕	↕	–

Interface : Sequential Logic



Interface of an ALU_SEQ

Ports

Name	Mode	Width	Meaning
CLK	IN	1	System clock
RESET	IN	1	Reset active high
I	IN	8	Value of an operand A or B
LOAD	IN	1	Loading value at input I to one of the internal registers holding A or B (control signal active for one clock period; the action takes place at the rising edge of the clock)
SEL_IN	IN	1	0: loading register A 1: loading register B
OP	IN	4	Operation mode
RUN	IN	1	Writing the result to registers holding X0, X1, Y0, and Y1 (control signal active for one clock period; the action takes place at the rising edge of the clock)

Name	Mode	Width	Meaning
SEL_OUT	IN	1	0: R = X 1: R = Y
R	OUT	8	Digit of a result
Z	OUT	1	Zero flag.
Cout	OUT	1	Carry out flag.
V	OUT	1	Overflow flag.
F_active	OUT	1	Logical OR of Z, Cout and V.
X_bin_pal	OUT	1	Flag set to high when the output X is a binary palindromic number (numbers that remain the same when binary digits are reversed)
X_prime	OUT	1	Flag set to high when the output X is a prime number
N	OUT	1	Flag set to high when the output X is a negative number