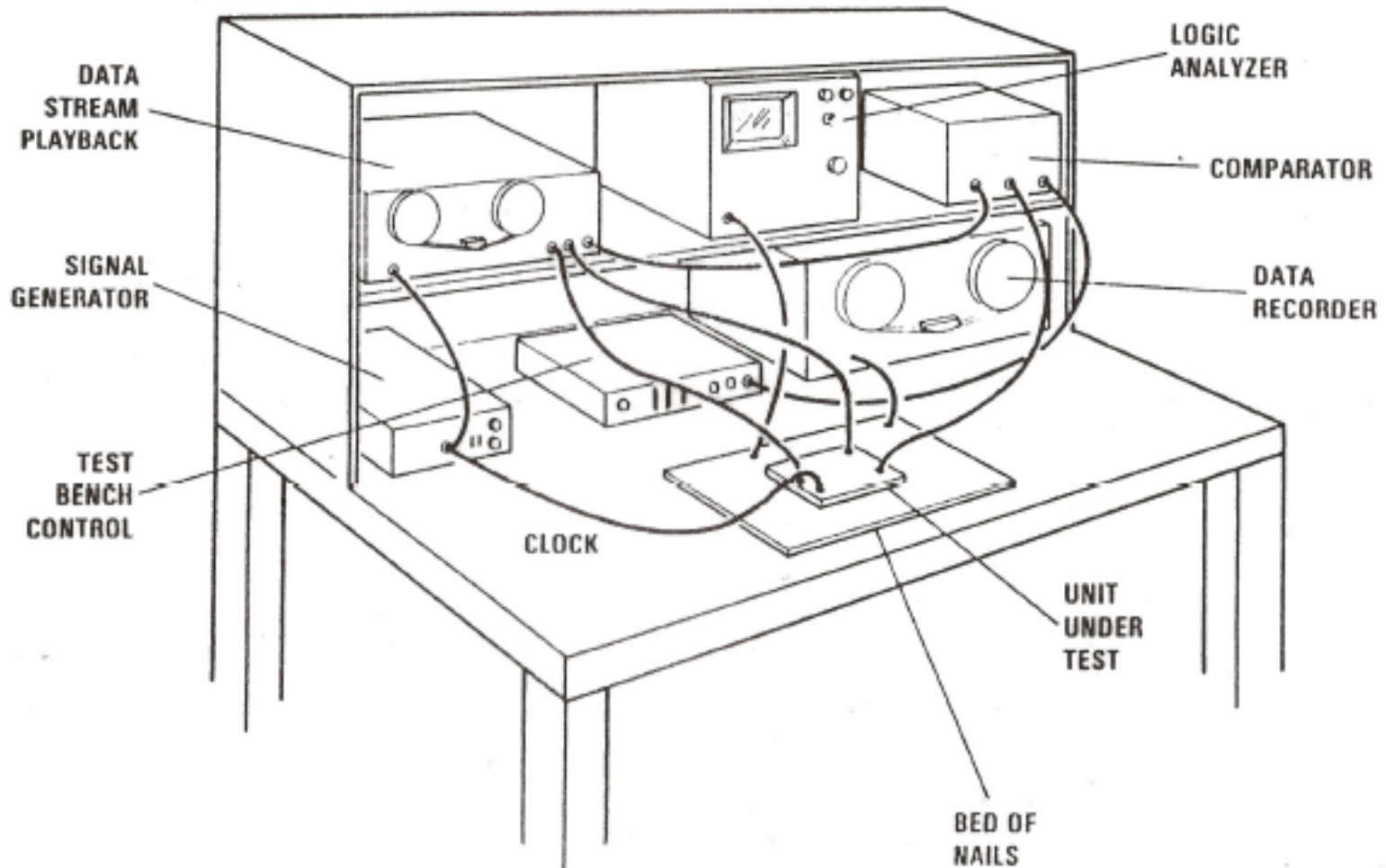


VHDL Testbench Design

Textbook chapters 2.19, 4.10-4.12, 9.5

The Test Bench Concept



Elements of a VHDL/Verilog testbench

- ▶ **Unit Under Test (UUT) – or Device Under Test (DUT)**
 - ▶ instantiate one or more UUT's
- ▶ **Stimulus of UUT inputs**
 - ▶ algorithmic
 - ▶ from arrays
 - ▶ from files
- ▶ **Checking of UUT outputs**
 - ▶ assertions
 - ▶ write to files



Instantiating the UUT

-- 8 bit adder testbench

entity adder_bench is -- no top-level I/O ports

end adder_bench;

architecture test of adder_bench is

 component adder is -- declare the adder component

 port (

 X,Y: in std_logic_vector(7 downto 0);

 Z: out std_logic_vector(7 downto 0)

);

 signal A,B,Sum: std_logic_vector(7 downto 0); --internal signals

begin

 UUT: adder port map (A,B,Sum); --instantiate adder as UUT



Algorithmic generation of stimulus

-- Generate test values for an 8-bit adder inputs A & B

process begin

for m in 0 to 255 loop -- 256 addend values

A <= std_logic_vector(to_UNSIGNED(m,8)); -- apply m to A

for n in 0 to 255 loop -- 256 augend values

B <= std_logic_vector(to_UNSIGNED(n,8)); -- apply n to B

wait for T ns; -- allow time for addition

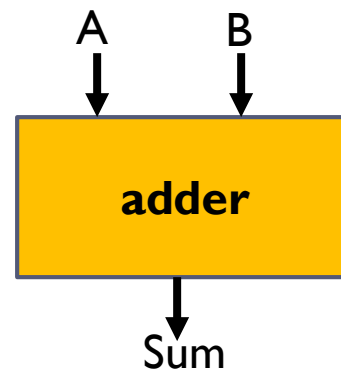
assert (to_integer(UNSIGNED(Sum)) = (m + n)) -- expected sum

report "Incorrect sum"

severity NOTE;

end loop; end loop;

end process;



Check results with “assertions”

-- Assert statement checks for expected condition

assert (A = (B + C)) -- expect A = B+C (any boolean condition)

report “Error message”

severity NOTE;

- ▶ Match data types for A, B, C
- ▶ Print “Error message” if assert condition FALSE
(condition is not what we expected)
- ▶ Specify one of four severity levels:
NOTE, WARNING, ERROR, FAILURE
- ▶ Simulator allows selection of severity level to halt simulation
 - ▶ ERROR generally should stop simulation
 - ▶ NOTE generally should not stop simulation



Stimulating clock inputs

-- Simple 50% duty cycle clock

clk <= not clk after T ns; -- T is constant or defined earlier

-- Clock process, using “wait” to suspend for T1/T2

process begin

clk <= '1'; wait for T1 ns; -- clk high for T1 ns

clk <= '0'; wait for T2 ns; -- clk low for T2 ns

end process;



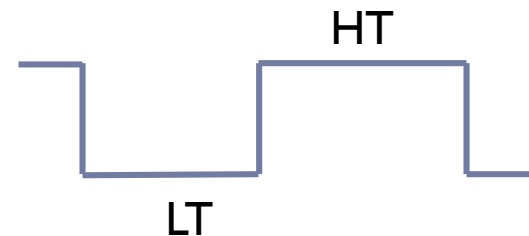
-- Alternate format for clock waveform

process begin

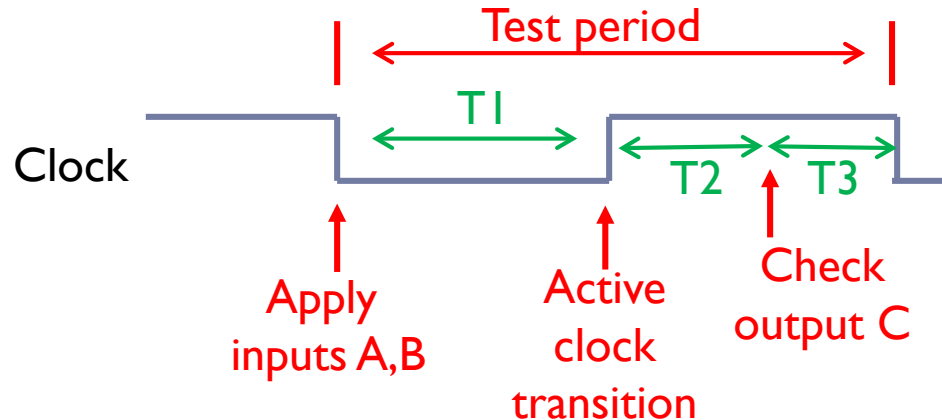
clk <= '1' after LT, '0' after LT + HT;

wait for LT + HT;

end process;



Sync patterns with clock transitions



```
A <= '0';           -- schedule pattern to be applied to input A
B <= '1';           -- schedule pattern to be applied to input B
wait for T1;        -- time for A & B to propagate to flip flop inputs
Clock <= '1';       -- activate the flip-flop clock
wait for T2;        -- time for output C to settle
assert C = '0'     -- verify that output C is the expected value
    report "Error in output C"
    severity ERROR;
wait for T3;        -- wait until time for next test period
```



Sync patterns with various signals

-- Test 4x4 bit multiplier algorithm

```
process begin
```

```
  for m in 0 to 15 loop;
```

```
    A <= std_logic_vector(to_UNSIGNED(m,4)); -- apply multiplier
```

```
    for n in 0 to 15 loop;
```

```
      B <= std_logic_vector(to_UNSIGNED(n,4)); -- apply multiplicand
```

```
      wait until CLK'EVENT and CLK = '1'; -- clock in A & B
```

```
      wait for 1 ns; -- move next change past clock edge
```

```
      Start <= '1','0' after 20 ns; -- pulse Start signal
```

```
      wait until Done = '1'; -- wait for Done to signal end of multiply
```

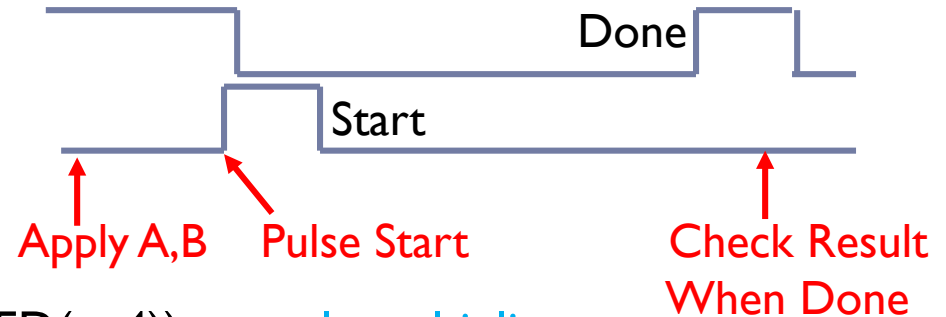
```
      wait until CLK'EVENT and CLK = '1'; -- finish last clock
```

```
      assert P = (A * B) report "Error" severity WARNING; -- check product
```

```
    end loop;
```

```
  end loop;
```

```
end process;
```



Checking setup/hold time constraints

-- Setup time T_{su} for flip flop D input before rising clock edge is 2ns

```
assert not (CK'stable and (CK = '1') and not D'stable(2ns))
  report "Setup violation: D not stable for 2ns before CK";
```

-- DeMorgan equivalent

```
assert CK'stable or (CK = '0') or D'stable(2ns)
  report "Setup violation: D not stable for 2ns before CK";
```

-- Figure 8-6 in the Roth textbook

check: process

begin

```
wait until (clk'event and CLK = '1');
```

```
assert (D'stable(setup_time))
```

```
  report "Setup time violation"
```

```
  severity ERROR;
```

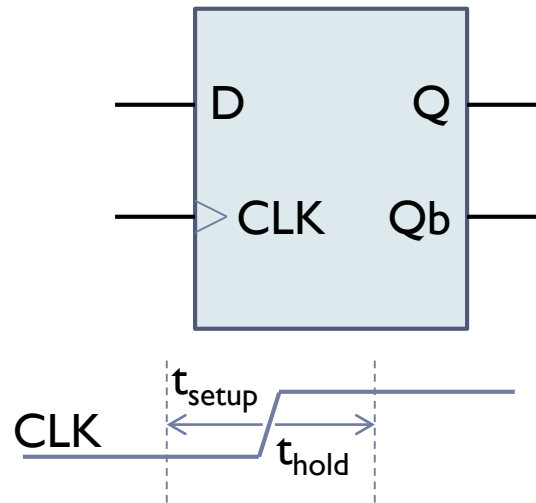
```
wait for hold_time;
```

```
assert (D'stable(hold_time))
```

```
  report "Hold time violation"
```

```
  severity ERROR;
```

```
end process check;
```



D should be “stable” for t_{setup} prior to the clock edge and remain stable until t_{hold} following the clock edge.

Testbench for a modulo-7 counter

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;

ENTITY modulo7_bench is end modulo7_bench;

ARCHITECTURE test of modulo7_bench is
    component modulo7
        PORT (reset,count,load,clk: in std_logic;
             I: in std_logic_vector(2 downto 0);
             Q: out std_logic_vector(2 downto 0));
    end component;
    for all: modulo7 use entity work.modulo7(Behave);
    signal clk : STD_LOGIC := '0';
    signal res, cnt, ld: STD_LOGIC;
    signal din, qout: std_logic_vector(2 downto 0);

begin
    -- instantiate the component to be tested
    UUT: modulo7 port map(res,cnt,ld,clk,din,qout);
```

Alternative
to “do” file

Continue on
next slide



Testbench: *modulo7_bench.vhd*

qint = expected outputs of UUT

```
clk <= not clk after 10 ns;
```

```

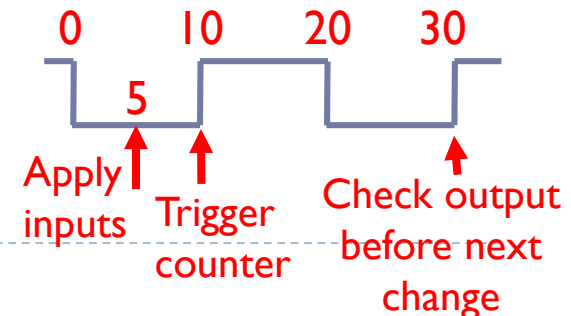
P1: process
  variable qint: UNSIGNED(2 downto 0);
  variable i: integer;
begin
  qint := "000";
  din <= "101"; res <= '1';
  cnt <= '0'; ld <= '0';
  wait for 10 ns;
  res <= '0';    --activate reset for 10ns
  wait for 10 ns;
  assert UNSIGNED(qout) = qint
    report "ERROR Q not 000"
    severity WARNING;
  res <= '1';    --deactivate reset
  wait for 5 ns; --hold after reset
  ld <= '1';     --enable load
  wait until clk'event and clk = '1';

```

```

  qint := UNSIGNED(din); --loaded value
  wait for 5 ns;         --hold after load
  ld <= '0';             --disable load
  cnt <= '1';           --enable count
  for i in 0 to 20 loop
    wait until clk'event and clk = '1';
    assert UNSIGNED(qout) = qint
      report "ERROR Q not Q+1"
      severity WARNING;
    if (qint = "110") then
      qint := "000";    --roll over
    else
      qint := qint + "001"; --increment
    end if;
  end loop;
end process;

```



Print message if incorrect result

Test vectors from an array

-- Can be used if vector generation is not “algorithmic”

```
type vectors is array (1 to N) of std_logic_vector(7 downto 0);  
signal V: vectors := -- initialize vector array
```

```
(  
    "00001100", -- pattern 1  
    "00001001", -- pattern 2  
    "00110100", -- pattern 3  
    ..  
    "00111100" -- pattern N  
);
```

*Also use to initialize
“memory” contents.*

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

```
begin  
UUT: somemodule port map (in1 => A, .....
```

```
process  
begin  
    for i in 0 to N loop  
        A <= V(i); -- apply ith vector to A
```



Reading test vectors from files

```
use std.textio.all;           -- Contains file/text support
architecture m1 of bench is begin
    signal Vec: std_logic_vector(7 downto 0); -- test vector
process
    file P: text open read_mode is "testvecs"; -- test vector file
    variable LN: line;           -- temp variable for file read
    variable LB: bit_vector(31 downto 0); -- for read function
begin
    while not endfile(P) loop    -- Read vectors from data file
        readline(P, LN);        -- Read one line of the file (type "line")
        read(LN, LB);           -- Get bit_vector from line
        Vec <= to_stdlogicvector(LB); -- Vec is std_logic_vector
    end loop;
end process;
```

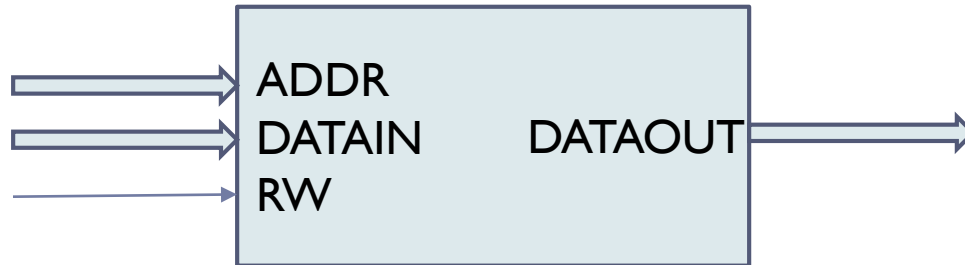


Memory testbench design

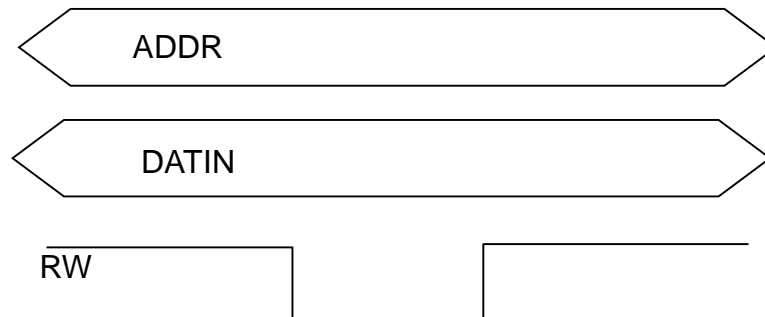
- ▶ **Basic testbench operation:**
 - ▶ **Step 1:** Write data patterns to each address in the memory
 - ▶ **Step 2:** Read each memory address and verify that the data read from the memory matches what was written in Step 1.
 - ▶ **Step 3:** Repeat Steps 1 and 2 for different sets of data patterns.



Memory read and write timing

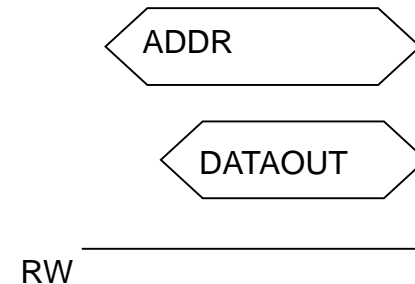


Write Operation



1. Apply patterns to ADDR and DATAIN
2. After a short delay, pulse RW (low)
3. Data captured in memory on rising edge of RW – should also be on DATAOUT

Read Operation



1. Apply patterns to ADDR
2. Leave RW high (for read)
3. DATAOUT from memory after a short delay



Memory testbench process general format

process begin

RW <= '1'; -- default level for RW

-- Write data to all N memory locations (k = # address bits)

for A in 0 to N loop

ADDR <= std_logic_vector(to_unsigned(A,k)); -- convert A to ADDR type

DATAIN <= next_data; -- data to be written to address A

RW <= '0' after T1 ns, '1' after T2 ns; -- pulse RW from 1-0-1

wait for T3 ns; -- wait until after RW returns to 1

end loop;

-- Read data from all N memory locations and verify that data matches what was written

for A in 0 to N loop

ADDR <= std_logic_vector(to_unsigned(A,k)); -- convert A to ADDR type

wait for T4 ns; -- allow memory time to read and provide data

assert DATAOUT = expected_data -- did we read expected data?

report "Unexpected data"

severity WARNING;

end loop;

end process;

We need some method for determining data patterns to be written.



Memory testbench input/output files

We can provide a sequences of operations, addresses, and data from a text file, and write testbench results to another text file, using the VHDL textio package.

Input file format:

```
w 0 10000000
w 1 00100001
w 2 00000000
w 3 00000000
r 0
r 1
r 2
r 3
e 0
```

Operation Address Data

Operations are write (w), read (r), and end (e).

Output file format:

```
w 0 10000000 10000000
w 1 00100001 00100001
w 2 00000000 00000000
w 3 00000000 00000000
r 0 00000001
r 1 00100001
r 2 10100100
r 3 00000110
```

Data read on DOUT

Black: Command from input file

Green: Data read on DOUT



```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use STD.TEXTIO.all;           -- package with routines for reading/writing files

entity TEST is
end entity;

architecture RTL of TEST is
    signal RW:          std_logic;           -- read/write control to MUT
    signal ADD:        std_logic_vector(1 downto 0); -- address to MUT
    signal DIN,DOUT:   std_logic_vector(7 downto 0); -- data to/from MUT
    signal STOP:       std_logic := '0';    -- stop reading vector file at end
    component Memry is
        port ( RW:      in          std_logic;
              ADDR:    in          std_logic_vector(1 downto 0);
              DATIN:   in          std_logic_vector(7 downto 0);
              DATO:    out         std_logic_vector(7 downto 0));
    end component;
begin
    MUT: Memry port map (RW, ADD, DIN, DOUT); -- instantiate memory component

```



-- main process for test bench to read/write files

process

```
file SCRIPT: TEXT is in "mut.vec";           -- "file pointer" to input vector file
file RESULT: TEXT is out "mut.out";         -- "file pointer" to output results file
variable L: line;                           -- variable to store contents of line to/from files
variable OP: character;                     -- operation variable (read/write/end)
variable AD: integer;                       -- address variable
variable DAT: bit_vector(7 downto 0);      -- variable for data transfer to/from files
```

begin

if (STOP = '0') then

```
RW <= '1';                                 -- set RW to read
READLINE(SCRIPT,L);                        -- read a line from the input file
READ(L,OP);                                -- read the operation from the line
READ(L,AD);                                -- read the address from the line
ADD <= std_logic_vector(to_unsigned(AD,2)); -- apply address to memory
```

(next slides for read and write operations)



-- Memory write operation

if (OP = 'w') then

 READ(L,DAT); -- read data from the input line

 DIN <= to_std_logic_vector(DAT);

 RW <= '1', '0' after 10 ns, '1' after 20 ns; -- pulse RW 0 for 10 ns

 wait for 30 ns;

 WRITE(L,OP); -- write operation to output line

 WRITE(L,' '); -- write a space to output line

 WRITE(L,AD); -- write address to output line

 WRITE(L,' '); -- write a space to output line

 WRITE(L,DAT); -- writes input data to output line

 DAT := to_bitvector(DOUT); -- DOUT should match DAT written

 WRITE(L,' '); -- write a space to output line

 WRITE(L,DAT); -- write DAT to output line

 WRITELINE(RESULT,L); -- write output line to output file



```

-- Memory read operation
elsif (OP = 'r') then
    wait for 10 ns;           -- wait for 10 ns to read
    DAT := to_bitvector(DOUT); -- convert DOUT to BIT_VECTOR
    WRITE(L,OP);             -- write operation to output line
    WRITE(L,' ');           -- write a space to output line
    WRITE(L,AD);            -- write address to output line
    WRITE(L,' ');           -- write a space to output line
    WRITE(L,DAT);           -- write DAT to output line
    WRITELINE(RESET,L);     -- write output line to output file
-- Stop operation
else
    STOP <= '1';           -- stop read/write of files when 'e' encountered
    wait for 10 ns;       -- wait for 10 ns to read
end if;
end if;
end process;
end architecture;

```

