VHDL: A Tutorial!

Mani B. Srivastava

UCLA - EE

OUTLINE

- Introduction to the language

  - *simple examples*

- VHDL's model of a system

  - *its computation model: processes, signals and time*

- Language features

- VHDL for logic and queue simulation

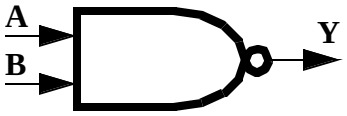
WHAT IS VHDL?

Programming Language + Hardware Modelling Language

It has all of the following:

- *Sequential Procedural language: PASCAL and ADA like*
- *Concurrency: statically allocated network of processes*
- *Timing constructs*
- *Discrete-event simulation semantics*
- *Object-oriented goodies: libraries, packages, polymorphism*

A NAND Gate Example




```
-- black-box definition (interface)
entity NAND is
        generic ( Tpd : time := 0 ns );
        port ( A, B : in bit; Y : out bit );
end entity;
```

```
-- an implementation (contents)
architecture BEHAVIOR_1 of NAND is
begin
        Y <= A nand B after Tpd;
end BEHAVIOR_1;
```

Important Concepts

entity

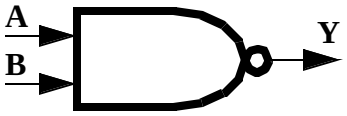
architecture

generic

port

waveform assignment

Another Implementation of NAND




```
-- there can be multiple implementations
architecture BEHAVIOR_2 of NAND is
        signal X : bit;
begin
        -- concurrent statements
        Y <= X after Tpd;
        X <= '1' when A='0' or B='0' else
                '0';
end BEHAVIOR_2;
```

Important Concepts

multiple architectures

signal

concurrent statements

Yet More NAND Gates!!!

```
entity NAND_N is
        generic ( N : integer := 4; Tpd : time);
        port ( A, B : in bit_vector(1 to N);
                Y : out bit_vector(1 to N));
end NAND_N;

architecture BEHAVIOR_1 of NAND_N is
begin
        process
                variable X : bit_vector(1 to N);
        begin
                X := A nand B;
                Y <= X after Td;
                wait on A, B;
        end process;
end BEHAVIOR_1;
```

Important Concepts

process

variable

wait

sequential statements

events

The *process* Statement

[label:] **process** [(sensitivity_list)]
    [declarations]
begin
    {sequential_statement}
end process [label];

• It defines an independent sequential process which repeatedly executes its body.

• Following are equivalent:

```
process (A,B)
begin
    C <= A or B;
end;
```

```
process
begin
    C <= A or B;
    wait on A, B;
end;
```

•No *wait* statements allowed in the body if there is a sensitivity_list.

The *wait* Statement

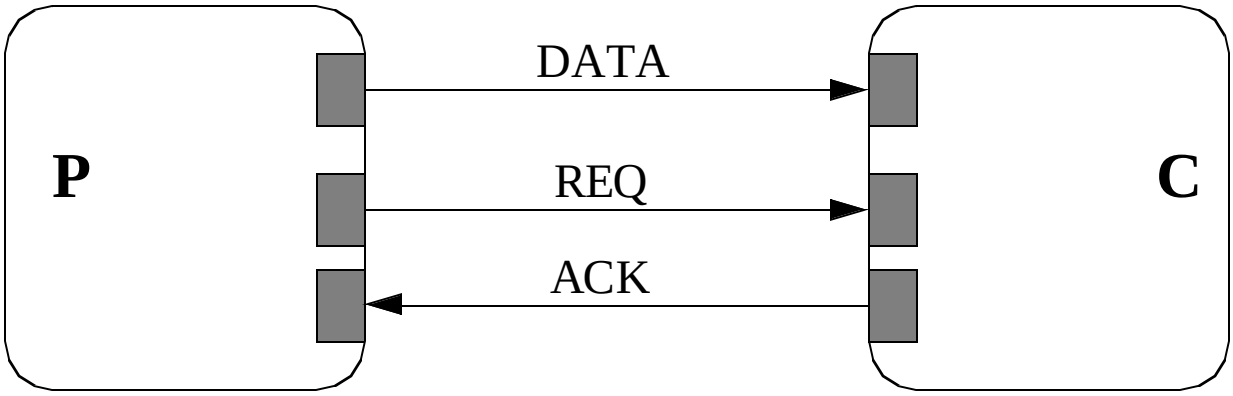
wait [**on** *list_of_signals*]
[**until** *boolean_expression*]
[**for** *time_expression*] ;

This is the ***ONLY*** sequential statement during which time advances!

examples:

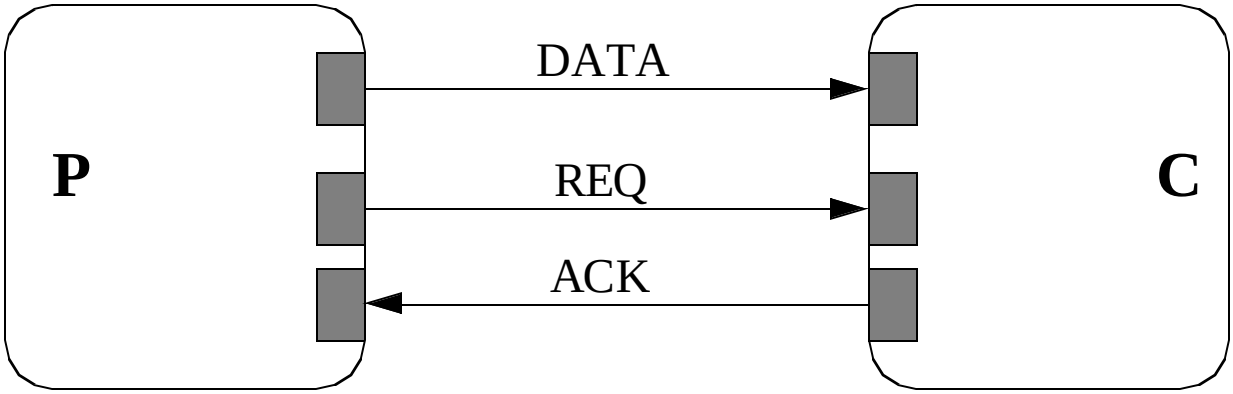
```
-- wait for a rising or falling edge on CLK
wait on CLK;
wait until CLK'EVENT; -- this is equivalent to the above

-- wait for rising edge of CLK
wait on CLK until CLK='1';
wait until CLK='1'; -- this is equivalent to the above

-- wait for 10 ns
wait until 10 ns;

-- wait for ever (the process effectively dies!)
wait;
```

A Simple Producer-Consumer Example



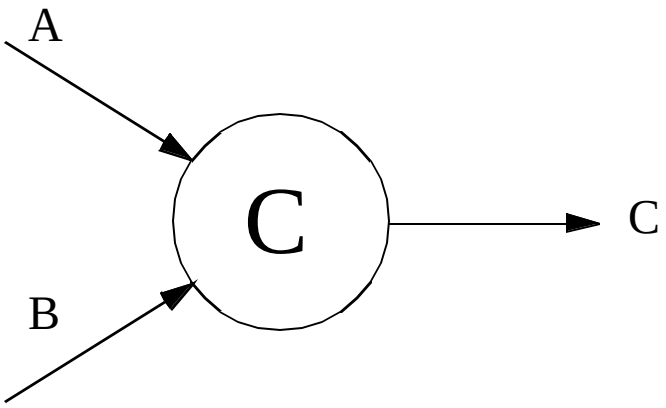

```
entity producer_consumer is
end producer_comsumer;

architecture two_phase of producer_consumer is
        signal REQ, ACK : bit;
        signal DATA : integer;
begin
        P: process begin
                DATA <= produce();
                REQ <= not REQ;
                wait on ACK;
        end P;
        C: process begin
                wait on REQ;
                consume(DATA);
                ACK <= not ACK;
        end C;
end two_phase;
```

Producer-Consumer *contd.* : 4-φ case




```
architecture four_phase of producer_consumer is
    signal REQ, ACK : bit := '0';
    signal DATA : integer;
begin
    P: process begin
        DATA <= produce();
        REQ <= '1';
        wait until ACK='1';
        REQ <= '0';
        wait until ACK='0';
    end P;
    C: process begin
        wait until REQ='1';
        consume(DATA);
        ACK <= '1';
        wait until REQ='0';
        ACK <= '0';
    end C;
end four_phase;
```

Muller C-Element




```
entity MULLER_C_ELEMENT is
      port (A,B : in bit; C : out bit);
end MULLER_C_ELEMENT;

architecture BEHAVIOR is
begin
      process begin
            wait until A='1' and B='1';
            C <= '1';
            wait until A='0' and B='0';
            C <= '0';
      end process;
end BEHAVIOR;
```

Could have written:

```
wait until A=B;
C <= A;
```

An Edge-Triggered D Flip-Flop

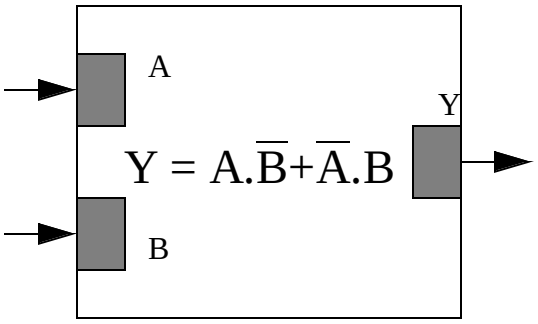
```
entity DFF is
        generic (T_setup, T_hold, T_delay : time:=0 ns);
        port (D, CLK: in bit; Q : out bit);
begin
        -- check setup time
        assert not (CLK'EVENT and CLK='1' and
           D'LAST_EVENT < T_setup)
              report " Setup violation"
              severity WARNING;
        -- check hold time
        assert not (CLK'DELAYED(T_hold)'EVENT and
           CLK'DELAYED(Thold)='1' and
           D'LAST_EVENT < T_hold)
              report " Hold violation"
              severity WARNING;
end DFF;
architecture BEHAVIOR of DFF is
begin
        process begin
              wait on CLK until CLK='1';
              Q <= D after T_delay;
        end process;
end BEHAVIOR;
```

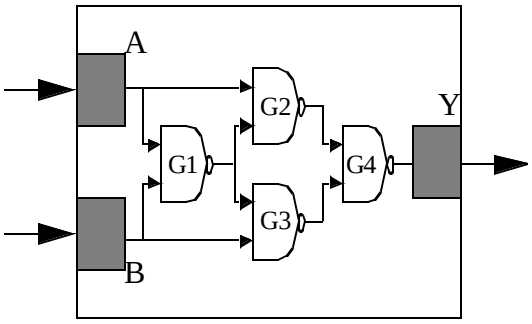
Try writing this in THOR!

Behavior *vs* Structure Description

An entity can be described by its *behavior* or by its *structure*, or in a mixed fashion.

example: a 2-input XOR gate






```
entity XOR is
        port ( A,B : in bit; Y : out bit);
end XOR;

architecture BEHAVIOR of XOR is
begin
        Y <= (A and not B) or (not A and B);
end BEHAVIOR;


architecture MIXED of XOR is
        component NAND
                port ( A, B : in bit; Y : out bit);
        end component;
        signal C, D, E : bit;
begin
        D <= A nand C;
        E <= C nand B;
        G1 : NAND port map (A, B, C);
        G4 : NAND port map (D, E, Y);
end MIXED;
```

```
architecture STRUCTURE of XOR is
        component NAND
                port ( A, B : in bit; Y : out bit);
        end component;
        signal C, D, E : bit;
begin
        G1 : NAND port map (A, B, C);
        G2 : NAND port map
                (A => A, B => C, Y => D);
        G3 : NAND port map
                (C, B => B, Y => E);
        G4 : NAND port map (D, E, Y);
end STRUCTURE;
```

Component Instantiation is just another Concurrent Statement!

The *Generate* Statement

Used to generate *iteratively* or *conditionally* a set of concurrent statements.

example: a ripple-carry adder

```
entity RIPPLE_ADDER is
        port (A, B : in bit_vector; CIN : in bit;
                SUM : out bit_vector; COUT : out bit);
begin
        assert A'LENGTH=B'LENGTH and
                A'LENGTH=SUM'LENGTH
                report "Bad port connections"
                severity ERROR;
end;
architecture STRUCTURE of RIPPLE_ADDER is
        alias IN1 : bit_vector(0 to A'LENGTH-1) is A;
        alias IN2 : bit_vector(0 to A'LENGTH-1) is B;
        alias S : bit_vector(0 to A'LENGTH-1) is SUM;
        signal C : bit_vector(IN1'RANGE);
        component FULL_ADDER port (A,B,CIN: in bit; S, COUT: out bit);
        end component;
begin
        L1: for I in S'RANGE generate
                L2: if I=0 generate
                        FA1: FULL_ADDER
                                port map (IN1(0),IN2(0),CIN,S(0),C(0));
                end generate;
                L3: if I>0 generate
                        FA2: FULL_ADDER
                                port map (IN1(I),IN2(I),C(I-1),S(I),C(I));
                end generate;
        end generate;
        COUT <= C(C'HIGH);
end STRUCTURE;
```

Concurrent *vs* Sequential Statements

- **Concurrent Statements**

Process	independent sequential process
Block	groups concurrent statements
Concurrent Procedure	convenient syntax for commonly occurring form of processes
Concurrent Assertion	
Concurrent Signal Assignment	
Component Instantiation	structure decomposition
Generate Statement	regular description

Order of execution is not defined!

- Sequential Statements

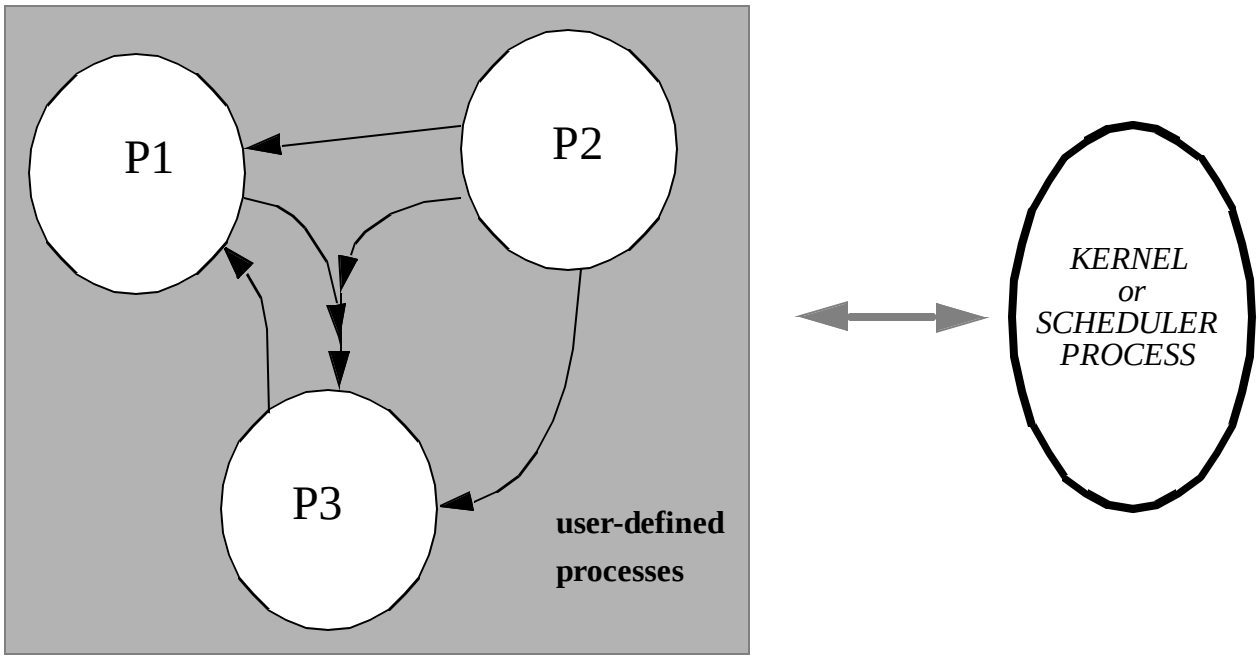
Wait	synchronization of processes
Assertion	
Signal Assignment	
Variable Assignment	
Procedure Call	
If	
Case	
Loop (for, while)	
Next	
Exit	
Return	
Null	

VHDL's Model of a System

• Static network of concurrent *processes* communicating using *signals*.

• A process has *drivers* for certain signals.

• A signal may be driven by multiple processes.



Reminds one of a *multi-tasking* OS!

And, most (all?) VHDL simulators are indeed very similar in philosophy … a *kernel process* coordinates the activity of user-defined processes during simulation.

Simplified Anatomy of the VHDL Kernel Process

```
vhdl_simulator_kernel()
{
    /* initialization phase */
    time = 0 ns;
    for (each process P) {
        run P until it suspends;
    }

    while TRUE do {
        /* this is one simulation cycle ... */
        if (no driver is active) {
            time = next time at which a driver is active
                or a process resumes;
            if (time = TIME'HIGH) break;
        }
        update_signals(); /* events may occur */
        for (each process P) {
            if (P is sensitive to signal S and an event has
            occurred on S in this cycle) {
                resume P;    /* put it on a list ... */
            }
        }
        for (each process P that has just resumed) {
            run P until it suspends;
        }
    }
}
```

Signals versus *Variables*

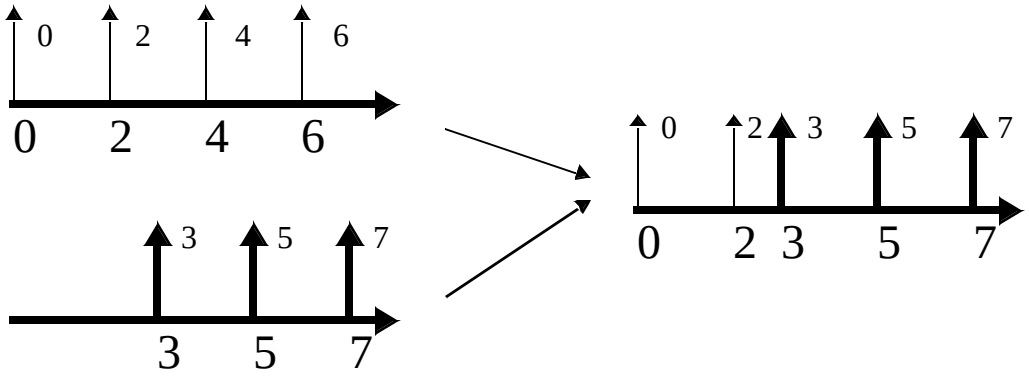
```
architecture DUMMY_1 of JUNK is
        signal Y : bit := '0' ;
begin
        process
                variable X : bit := ' 0' ;
        begin
                wait for 10 ns;
                X := '1';
                Y <= X;
                wait for 10 ns;
                -- What is Y at this point ? Answer: '1'
                ...
        end process;
end DUMMY_1;

architecture DUMMY_2 of JUNK is
        signal X, Y : bit := '0';
begin
        process
        begin
                wait for 10 ns;
                X <= '1';
                Y <= X;
                wait for 10 ns;
                -- What is Y at this point ? Answer: '0'
                ...
        end process;
end DUMMY_2;
```

Signal assignments with 0 delay take effect only after a *delta* delay. *i.e.*, in the next simulation cycle.

TRANSACTION SCHEDULING MODEL
TRANSPORT *vs* INERTIAL DELAY

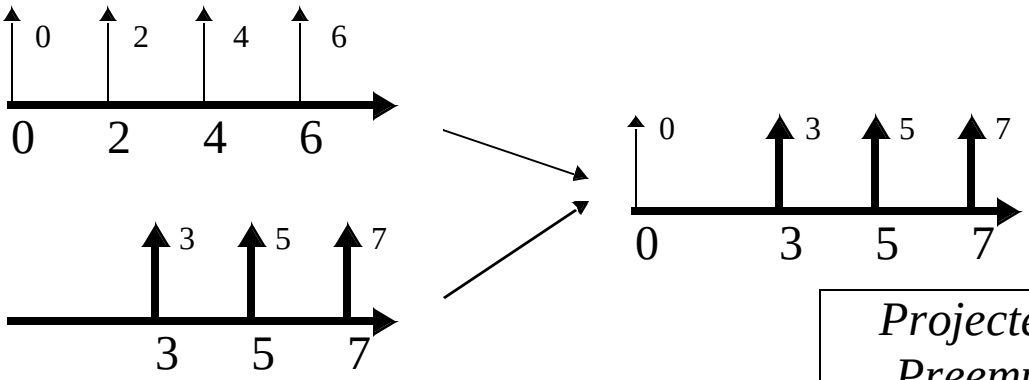
Case 1: transport delay model

Y <= 0 **after** 0 ns, 2 **after** 2 ns, 4 **after** 4 ns, 6 **after** 6 ns;
wait for 1 ns;
Y <= **transport** 3 **after** 2 ns, 5 **after** 4 ns, 7 **after** 6 ns;



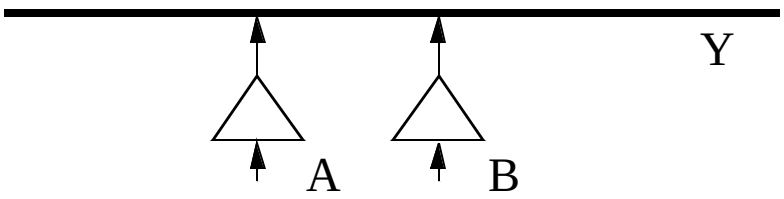
Case 2: inertial delay model

Y <= 0 **after** 0 ns, 2 **after** 2 ns, 4 **after** 4 ns, 6 **after** 6 ns;
wait for 1 ns;
Y <= 3 **after** 2 ns, 5 **after** 4 ns, 7 **after** 6 ns;



Projected waveform
Preemptive timing
Transport delay
Inertial delay

Signals with Multiple Drivers



Y <= A; -- in process1
and, Y <= B; -- in process2

What is the value of the signal in such a case?

VHDL uses the concept of a *Resolution Function* that is attached to a *signal* or a *type*, and is called every time the value of signal needs to be determined -- that is every time a driver changes value.

example: wire-anding (open-collector)

```
package RESOLVED is
    function wired_and (V:bit_vector) return bit;
    subtype rbit is wired_and bit;
end RESOLVED;
package body RESOLVED is
    function wired_and(V:bit_vector) return bit is
    begin
        for I in V'RANGE loop
            if V(I)='0' then return '0'; end if;
        end loop;
        return '1';
    end wired_and;
end RESOLVED;
```

Guarded Signals - *register* and *bus*

Guarded signals are those whose drivers can be *turned off*.

What happens when all drivers of a *guarded* signal are off?

Case 1: retain the last driven value

signal X : bit **register;**

useful for modelling charge storage nodes

Case 2: float to a user defined default value

signal Y : bit **bus**;

useful for modelling busses

Two ways to turn off the drivers:

```
-- null waveform in sequential signal assignment
signal_name <= null after time_expression;

-- guarded concurrent signal assignment
block (data_bus_enable='1')
begin
      data_bus <= guarded "0011";
end block;
```

How do VHDL and THOR differ?

• VHDL allows more or less arbitrary data types, parameterized models, and many other language goodies!

• VHDL can mix structure and behavior in a module.

• THOR has only *wired-X* resolution.

• VHDL is *process-oriented*, THOR is *event-oriented*.

THOR to VHDL conversion is easy ...

```
THOR_PROCESS: process
begin
        thor_init_section();
        while TRUE loop
                wait on list_of_input_and_biput_signals;
                thor_body_section();
        end loop;
end process THOR_PROCESS;
```

VHDL to THOR conversion is not!

THOR models are written as state machines.
In VHDL processes, the state is implicit.

• THOR has very poor delay modelling capabilities.

Using VHDL like C!

Normal *sequential* procedural programs can be written in VHDL without ever utilizing the *event scheduler* or the concurrent concepts.

example:

```
entity HelloWorld is end;

architecture C_LIKE of HelloWorld is
    use std.textio.all;
begin
    main: process
        variable buf : line;
    begin
        write(buf, ”Hello World!”);
        writeln(output, buf);
        wait;   -- needed to terminate the program
    end process main;
end C_LIKE;
```

Language Features: TYPES

TYPE = Set of Values + Set of Operations

VHDL TYPES:

- SCALAR
  - ENUMERATION *e.g.* character, bit, boolean
  - INTEGER *e.g.* integer
  - FLOATING *e.g.* real
  - PHYSICAL *e.g.* time
- COMPOSITE
  - ARRAY *e.g.* bit_vector, string
  - RECORD
- ACCESS
- FILE

examples:

```
type bit is ('0', '1');
type thor_bit is ('U', '0', '1', 'Z');
type memory_address is range 0 to 2**32-1;
type small_float is range 0.0 to 1.0;
type weight is range 0 to 1E8
    units
        Gm; -- base unit
        Kg = 1000 Gm; -- kilogram
        Tonne = 1000 Kg; -- tonne
    end units;
```

Language Features: SUBTYPES

SUBTYPE = TYPE + constraints on values

- *TYPE is the base-type of SUBTYPE*
- *SUBTYPE inherits all the operators of TYPE*
- *SUBTYPE can be more or less used interchangeably with TYPE*

examples:

```
subtype natural is integer range 0 to integer'HIGH;
subtype good_thor_bit is thor_bit range '0' to '1';
subtype small_float is real range 0.0 to 1.0;
```

examples of Array and Record types:

```
-- unconstrained array (defines an array type)
type bit_vector is array (natural range <>) of bit;
-- constrained array (define an array type and subtype)
type word is array (0 to 31) of bit;
-- another unconstrained array
type memory is array (natural range <>) of word;
-- following is illegal!
type memory is array (natural range <>) of bit_vector;
-- an example record
type PERSON is
        record
                name    : string(1 to 20);
                age     : integer range 0 to 150;
        end record;
```

Language Features: OVERLOADING

•Pre-defined operators (*e.g.*, +, -, and, nand *etc.*) can be *overloaded* to call functions

example:

```
function "and" (L,R : thor_bit) return thor_bit is
begin
    if L='0' or R='0' then
        return '0';
    elsif L='1' and R='1' then
        return '1';
    else
        return 'U';
    end if;
end "and";

-- now one can say
C <= A and B; -- where A, B and C are of type thor_bit
```

• Two subprograms (functions or procedures) can have the same name, *i.e.*, the names can be *overloaded*. They are distinguished by parameter types. *e.g.*,

```
function MAX(A,B:integer) return integer;
function MAX(A,B:real) return real;
```

Language Features: CONFIGURATIONS

• *Component* declarations really define a template for a design *entity*.

• The *binding* of an *entity* to this template is done through a *configuration* declaration.

```
entity data_path is
        ...
end data_path;
architecture INCOMPLETE of data_path is
        component alu
                port(function : in alu_function;
                        op1, op2 : in bit_vector_32;
                        result : out bit_vector_32);
        end component;
begin
        ...
end INCOMPLETE;
configuration DEMO_CONFIG of data_path is
        for INCOMPLETE
                for all:alu
                        use entity work.alu_cell(BEHAVIOR)
                        port map (function_code => function,
                                operand1 => op1, operand2 => op2,
                                result => result, flags => open);
                end for;
        end for;
end DEMO_CONFIG;
```

Language Features: PACKAGES

• A *package* is a collection of reusable declarations (constants, types, functions, procedures, signals *etc.*)

A package has a
- *declaration* (interface), and a
- *body* (contents) [optional]

example:

```
package SIMPLE_THOR is
    type thor_bit is ('U', '0','1','Z');
    function "and" (L,R: thor_bit) return thor_bit;
    function "or" (L,R:thor_bit) return thor_bit;
    ...
end SIMPLE_THOR;

package body SIMPLE_THOR is
    function "and" (L,R: thor_bit) return thor_bit is
    begin
    ...
    end "and";
    ...
end SIMPLE_THOR;

-- and then it can be used after saying
library my_lib; use my_lib.SIMPLE_THOR.all;
```

Language Features: DESIGN UNITS and LIBRARIES

• VHDL constructs are written in a *design file* and the compiler puts them into a *design library*.

• *Libraries* are made up of *design units.*

- *primary design units*
  - entity declarations
  - package declarations
  - configuration declarations
- *secondary design units*
  - architecture bodies
  - package bodies

• *Libraries* have a *logical* name and the OS maps the *logical* name to a *physical* name.

- for example, directories on UNIX

• Two special *libraries:*

work*:* the working design library

std*:* contains packages *standard* and *textio*

• To declare libraries that are referenced in a design unit:

library *library_name*;

• To make certain library units directly visible:

use *library_name.unit_name*;

use also defines *dependency* among *design units.*

Logic Simulation In VHDL

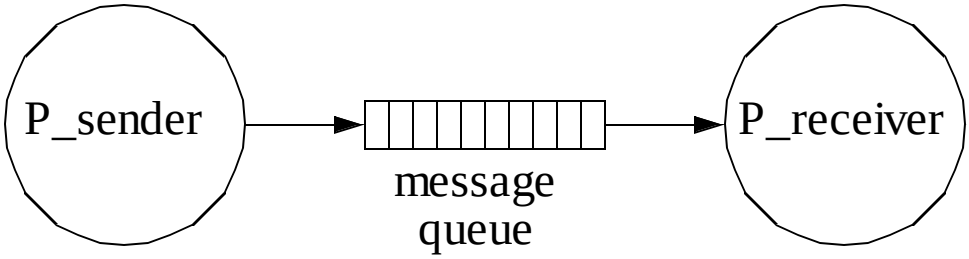
- The *2-state bit* data type is insufficient for low-level simulation.

- Multi-Valued types can easily be built in VHDL
  - several packages available
  - but no standard yet ...

- THOR uses a 4-value (‘U’,’0’,’1’,’Z’) system
  - but no notion of *strength*
  - only wired-X resolution

- Multi-State/Strength systems with interval logic

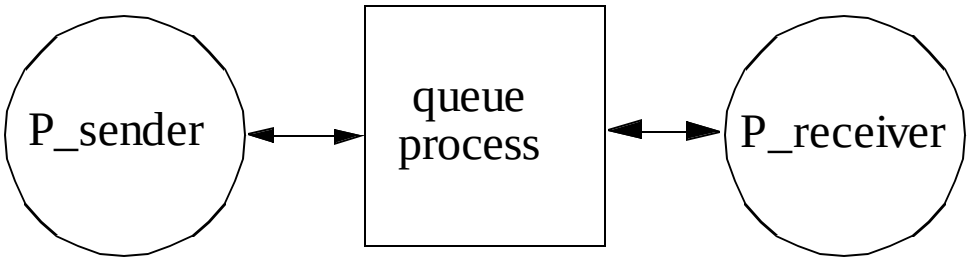
High-Level Message Queue Simulation



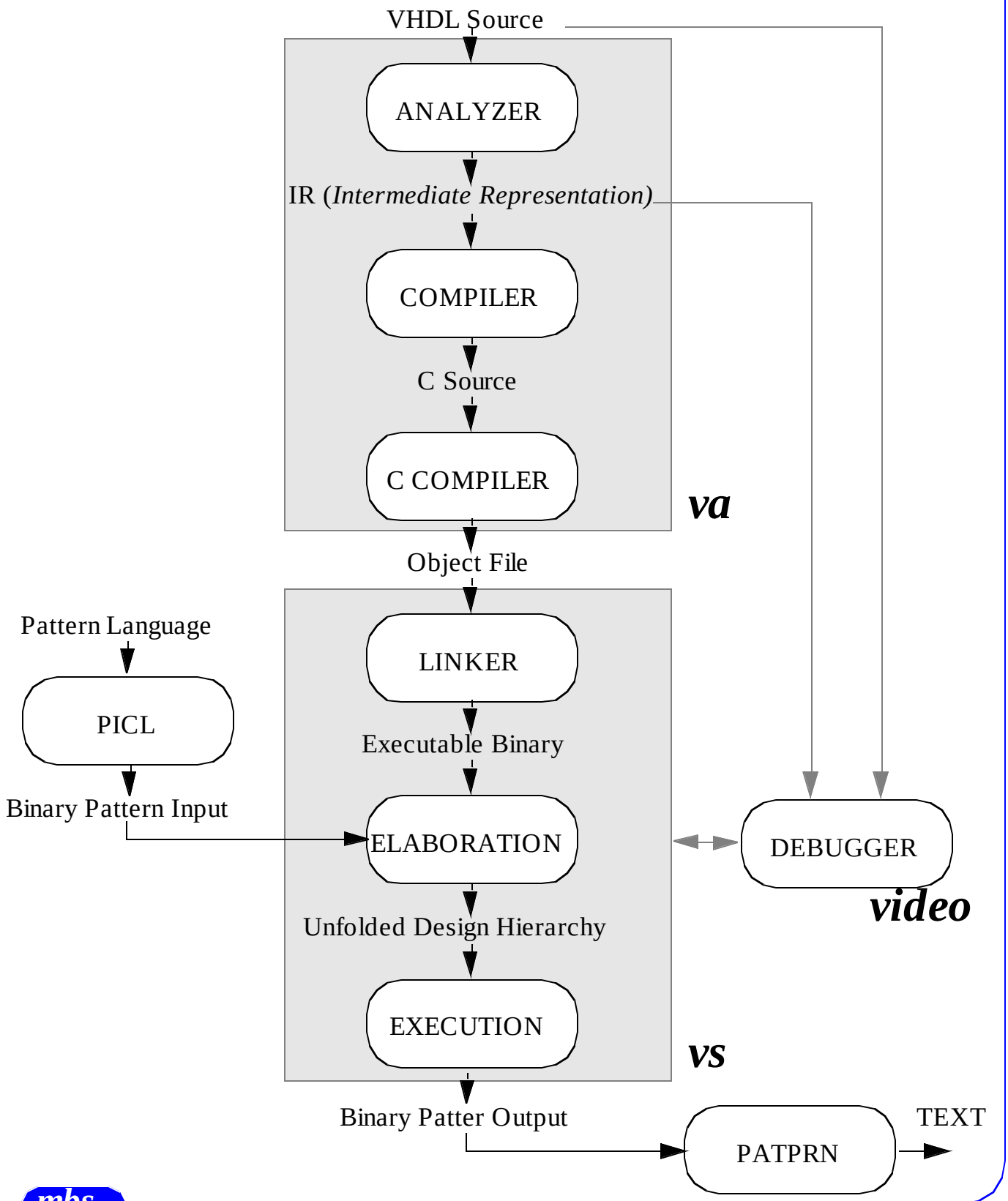
• VHDL *signal* is not a very good inter-process communication primitive for high-level simulation

- unbuffered and non-interlocked
- cannot support queues directly

• Queue package to support message queues with
- single reader/writer
- synchronous (unbuffered) or asynchronous (buffered - finite depth and infinite depth)
- write with blocking/overwrite/timeout
- read with blocking/previous/timeout



MCC VHDL Simulator



Problems in VHDL

- No *generic packages*
- No *function pointers*
- File I/O is pretty clumsy ...
- No math library yet
  - can use C-interface
- No standard package for low level simulation
- No support for high level simulation with message queues
- Arbitrary data types make user-interface a problem
- Just too complex!