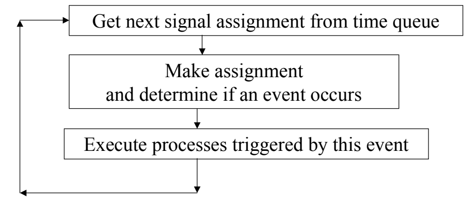
Event Driven Simulation

- A VHDL Simulator is an *event-driven* simulator
- *Events* occur on signals
- An *event* is a change in signal value at a particular time.
- The *time queue* is an ordered list of signal assignments for all signals in the simulator
- Signal assignments are ordered in ascending order (increasing time) on the time queue
- If a signal assignment causes a change in signal value, this is an event
  - Executing an event triggers a process which may generate more signal assignments to be placed on the time queue.

Simulation Loop



Time queue: A<= '1' at 2 ns, B <= '0' at 4 ns, C <= '1' at 10 ns
Actually represented by ordered pairs (value, time) on time queue

A: ('1', 2 ns), B: ('0', 4 ns), C: ('1', 10 ns)

Review: Concurrent vs. Sequential statements

- Recall the VHDL statements are divided into two classes 'concurrent' statements and 'sequential' statements.
- Examples of concurrent statements:

```
y <= A when (S = '1') else B;
s <= A after 10 ns;
```

Concurrent statements execute whenever events on signals used by the concurrent statements trigger them.

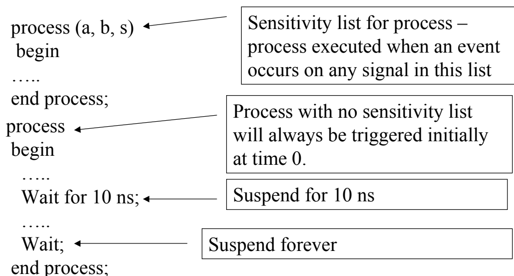
Processes

- A 'process' is a concurrent statement. Sequential statements can only be used within a process.
  - Statements are executed 'sequentially' within a process until the process is suspended either via a 'wait' statement or until there are no more statements to be executed.
- An example of a sequential statement in a process is:

```
if (clk'event and clk = '1') then
    y <= A when (S = '1') else B;
end if;
```


Process Triggering

A process can be triggered by resumption after a wait statement or by an event on a signal in its sensitivity list:



Some Rules for Processes

If a process has a sensitivity list, then it cannot contain a 'wait' statement.

A process with a sensitivity list is always triggered at time 0 because all signals always have an initial event placed on them at time 0.

A process without a sensitivity list is always triggered at time 0 initially.

If a process without a sensitivity list 'falls out the bottom' then it immediately loops back to the top until it hits a wait statement.

An Infinite Loop

```
process
 begin

   A <= '1';

 end process;
```

This process generates an infinite loop because it will 'fall out the bottom', loop back, and never encounter a wait statement.

You will get a compiler warning about this – if you execute it, the modelsim simulator may hang.

A Common Problem

A common problem is to not include a needed signal on a sensitivity list:

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

This is implementing a 2/1 mux. Signal 'B' has been left off the sensitivity list by mistake – if 'S=0' and a change occurs on 'B', this change will not be propagated to the Y output! This can be hard to debug – be careful with sensitivity lists!

Delta Time

Signal Assignments are used to place ordered pairs on time queue

```
A <=  transport '1' after 10 ns;
```

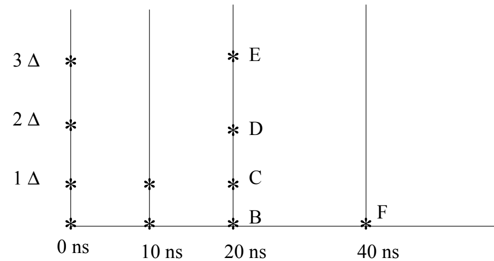
will add the assignment A: ('1', *NOW* + 10 ns) where *NOW* is the current simulation time.

What about :

```
A <=  transport '1';
```

will add the assignment A: ('1', NOW + 1 *delta*) . A delta is an infinitesimal interval used by the simulator to maintain assignment ordering for assignments that take place at the *same simulation time*.

Two-Dimensional Time



Maximum number of delta units within one time unit is simulator dependent (for *Modelsim*, is configurable in *modelsim.ini* via '*Iterationlimit*' parameter – default is 5000).

From " VHDL Techniques, Experiments, and Caveats", Joe Pick

A Delta-Time Infinite Loop

The following process will cause the *modelsim* simulator to exceed its delta time iteration limit

```
signal  a : std_logic := '0';

process (a)
begin
      a <= not (a);
end process;
```

Signal 'A' changes value for each signal assignment. This causes the process to be triggered again. Time advances by 1 delta each time the process is triggered and the simulator will halt after the iteration limit is reached.

Signal Assignment Rules

A signal assignment within a process will add an ordered pair to the time queue. There may already be assignments to this signal on the time queue.

1. If the new assignment time is AFTER the other assignment times, then the new assignment pair is added to the end of the list.

2. Any ordered pairs for this signal on the assignment list that have times LATER than the new assignment are removed from the time queue.

A signal assignment pair can only be executed by the simulator after the process suspends.

Example 1

```
signal ta, tb, tc, td, te: std_logic := '0';
 begin
  pa:process
    begin
   -- schedule of 'ta' assignment is put immediately
   -- on timequeue. Because 'Z' assignment happens in TIME after
   -- '1' assignment, then both assignments are valid.
       ta <= transport '1' after 3 ns;
       ta <= transport 'Z' after 4 ns;
       wait;
    end process pa;
```


Example 2

```
pb:process
    begin
        -- Because '1' assignment happens in TIME before
        -- 'Z' assignment, then 'Z' assignment is cancelled.
       tb <= transport 'Z' after 4 ns;
       tb <= transport '1' after 3 ns;
       wait;
    end process pb;
```


Example 3

```
 pc:process
    begin
        -- Because 'Z' assignment happens in TIME after
        -- '1' assignment, then both assignments are valid.
      tc <= transport '1' after 3 ns,  'Z' after 4 ns;
      wait;
    end process pc;
```

Note that multiple assignments can be made with one assignment statement. The assignments must be in increasing time.

Example 4

```
pd:process
    variable i: integer := 0;
    begin
      -- result of assignments is same as in process 'a'
       if (i = 0) then
             -- first assignment
           td <= transport '1' after 3 ns;
            i := i + 1;
       else
           td <= transport 'Z' after 4 ns;
           wait;
        end if;
    end process pd;
```


Example 5

```
 pe:process
    variable i: integer := 0;
    begin
      -- result of assignments is same as in process 'b'
       if (i = 0) then
             -- first assignment
           te <= transport 'Z' after 4 ns;
            i := i + 1;
       else
           te <= transport '1' after 3 ns;
           wait;
        end if;
    end process pe;
```


Variables versus Signals

- Examples 4 & 5 used a *local variable* (will discuss global variables later).
  - Can only be declared within processes
  - Are not visible outside of a process
  - Variables retain their values between process executions
  - Variable assignment takes place immediately
  - Variable assignment uses " := " operator
- What about signals?
  - Signals must be declared outside of processes
  - Signals are visible to all processes and are used to carry information between processes
  - Signals have a waveform history and also retain their values between process invocations
  - Signal assingment within a process only places that assignment on the time queue – it cannot be acted on until the process suspends
  - Signals use the " <= " operator

Two processes…..

```
signal  a,b,d: integer:=  0;

one:process  (a)
begin
 a <=  1;
 if (a = 1 ) then  b <= 1;  end if;
 a <= 0;
end process one;

two:process (a)
 variable c : integer := 0;
 begin
  c := 1;
  if (c = 1) then d <= 1; end if;
 end process two;
```

This assignment does not happen because the ‘a<=1’ assignment does not take place unless process is suspended! The later assignment of ‘a<=0’ replaces that value on the queue.

This assignment occurs because the variable assignment c:=1 occurs immediately.

VHDL Delay Models

The assignment statement:

A <= transport ‘1’ after 10 ns;

uses the *transport* delay model. This delay model will always place this assignment on the time queue.

The assignment

A <= ‘1’ after 10 ns;

uses the default delay model which is called the *inertial* delay model. This delay model will reject any signal changes that occur within 10 ns of each other.

Inertial versus Transport Delay

dest_1 <= source after 10 ns; -- inertial delay model
dest_2 <= transport source after 10 ns; -- transport delay

A Problem with Inertial Delay

Inertial delay is intended to model physical device that have inertia and reject spikes. Unfortunately, the model assumes that the propagation delay and inertia delay are the same.

y <= A after 4 ns;

Inertial model can be used to reject glitch, but it also defines prop delay. Output has prop delay of 4 ns, rejects glitches < 4 ns.

y <= transport A after 4 ns;

Output delayed from input by 4 ns. No glitch rejection.

Wanted: Spike rejection time ≠ Prop delay.

Y = *reject* 2 ns *inertial* A after 4 ns;

‘reject’ statement added in VHDL ’93 standard.

Pre-defined Signal Attributes

There are many pre-defined attributes for signals. Some examples are:

A’*event* returns a boolean that is true if an event occurred on signal *A* during the current simulation cycle.

A’*last_event* returns a time value that is the amount of absolute time that has elapsed since *A* had an event. Warning: it is NOT the time of the last event, but rather the amount of time that has passed since the last event.

A’*last_active* returns the amount of absolute times that has elapsed since *A* last had a transaction. A transaction is any update to a signal (an update may or may not cause an event).

Signal Attributes that return Signals

Some signal attributes actually return new signals.

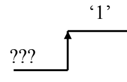
A'*delayed (T)* returns a signal as the same type as *A* but delayed by *T*. If T is not specified, then *T* = 1 delay time unit.

A'*stable(T)* returns a boolean signal that is true if *A* has not had an event for the length of time T.

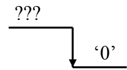
A'*quiet(T)* returns a boolean signal that is true if *A* has not had a transaction for the length of time T.

A'*transaction* returns a BIT signal that toggles for each transaction on *A*.

Detecting Edges



if (A'event and A = '1') then ....
-- rising edge



if (A'event and A = '0') then ....
--- falling edge

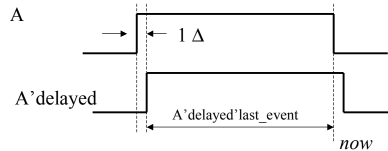
Measuring Pulse Width

```
process (A)
variable  pw :time;
begin
  if (A = '0') then
     pw :=  A'last_event;
  ..........
```

Returns 0!!!!

'last_event returns elapsed time from last event. The last event triggered this process!!!

Measuring Pulse Width (cont)

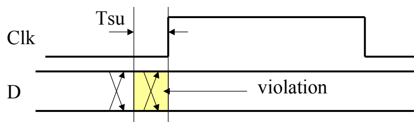



```
process (A)
variable  pw :time;
begin
  if (A = '0') then
     pw :=  A'delayed'last_event;
  ..........
```

Returns pulse width.
'*delayed* returns *A* delayed by 1 delta.

'*last_event* returns elapsed time between now and last event.

Measuring Setup Time



Does *D* change before the setup time?

```
if (clk'event and clk = '1') then
   if ( D'last_event < Tsu) then
       ----- setup time violation
```