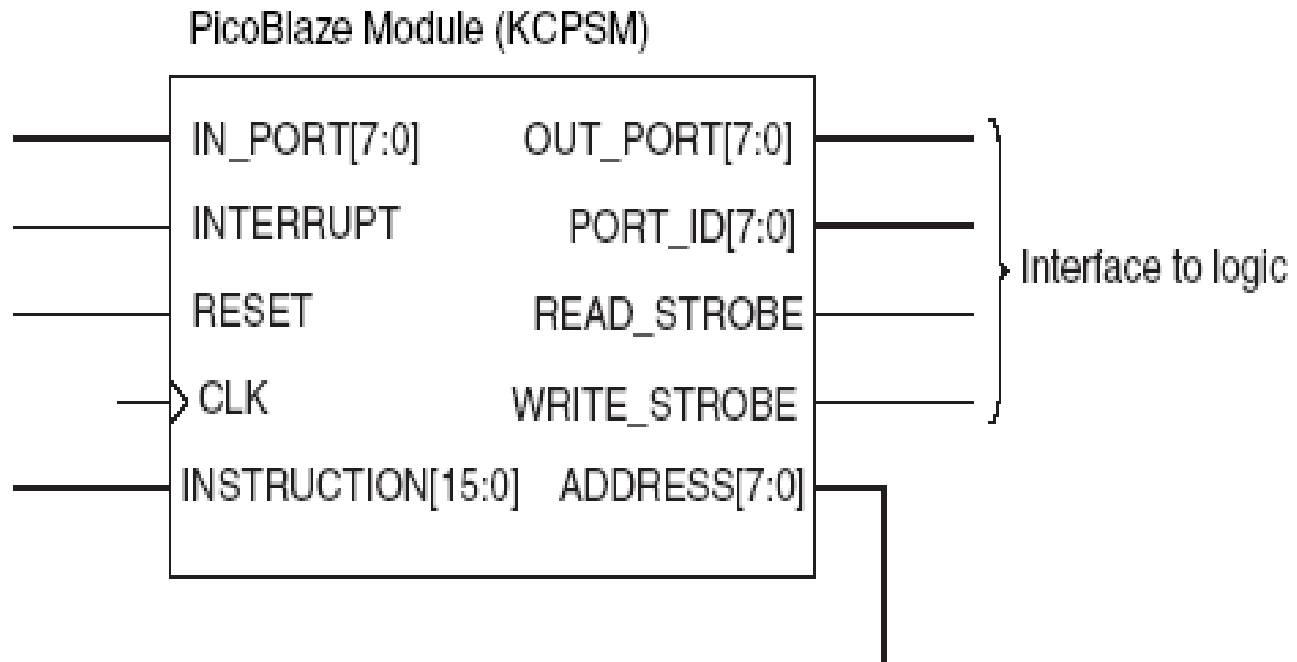


# Simulator Operation Advanced Testbenching

Simulator Operation  
File I/O, Automated Testing

# Brief Aside – Picoblaze Interrupts

# Interrupts



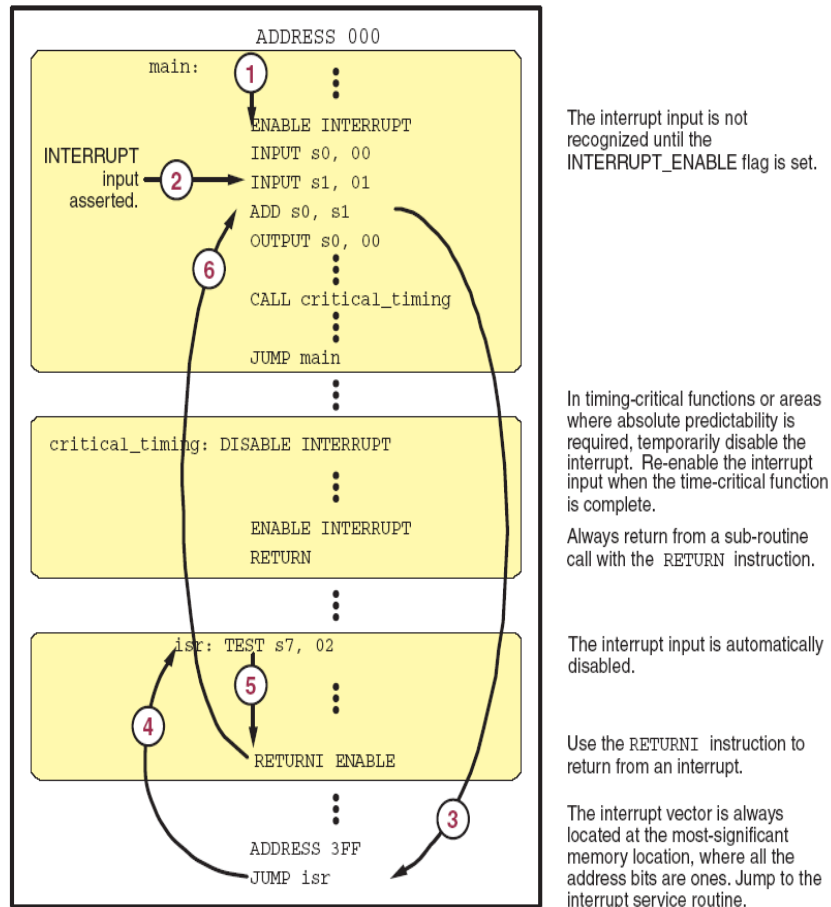
1 External Interrupt Source. Active high, level triggered.  
Interrupts can be enabled and disabled on the processor. By default on power-up  
Or reset, the interrupts are disabled (ignored)

# Interrupt Flow

- Forces the processor to jump to address 0xFF
  - Saves the return address
  - Saves the z,c flags
  - Disables further interrupts
- Presumably at address 0xff you have located a jump instruction which jumps to an **interrupt service routine**: i.e. what will the processor do to handle this event

# Interrupt Timing

## Example Interrupt Flow



The interrupt input is not recognized until the `INTERRUPT_ENABLE` flag is set.

In timing-critical functions or areas where absolute predictability is required, temporarily disable the interrupt. Re-enable the interrupt input when the time-critical function is complete.

Always return from a sub-routine call with the `RETURN` instruction.

The interrupt input is automatically disabled.

Use the `RETURNI` instruction to return from an interrupt.

The interrupt vector is always located at the most-significant memory location, where all the address bits are ones. Jump to the interrupt service routine.

UG129\_c4\_02\_051404

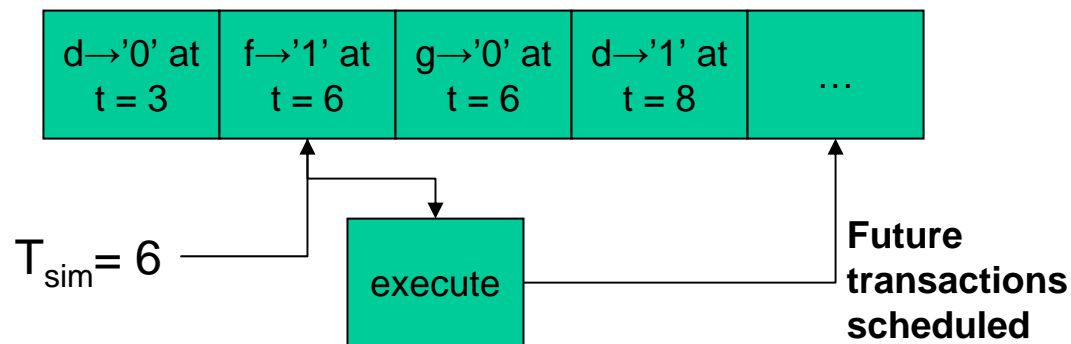
Figure 4-2: Example Interrupt Flow

# RETURNI

- At the end of the interrupt service routine, code must have a RETURNI instruction so that flags are restored.
  - RETURNI Enable
    - Enables interrupts
  - RETURNI Disable
    - Leaves them off
- Note that interrupts should never be allowed to happen while servicing an interrupt (i.e. there is only a “1-deep stack” for saving the flags)
- Note also that registers are not preserved, so you are responsible for your own run-time model. E.g. if an ISR destroys register contents, the rest of the program must not use those registers.

# Event Driven Simulation

- The VHDL simulator is an **event driven simulator** (very common for digital simulation)
  - The simulator has a queue of transactions scheduled for the future, which are processed in chronological order
  - Each transaction can schedule future transactions, which are added to the queue



# VHDL Simulation Time

- There is one simulation time for the entire simulated system (one master clock).
- Simulation is indexed by an integer  $T_c$ .
- Units are in seconds
  - A default resolution is specified when the simulator is invoked – often 1 ns for functional simulation and 100 ps for timing simulation (depending on the resolution of the models used).
- Simulation time begins at 0 s
- Simulation time is advanced by the execution of the *simulation cycle* to the time of the next queued event.
- A transaction can be scheduled “immediately” using *delta time*.

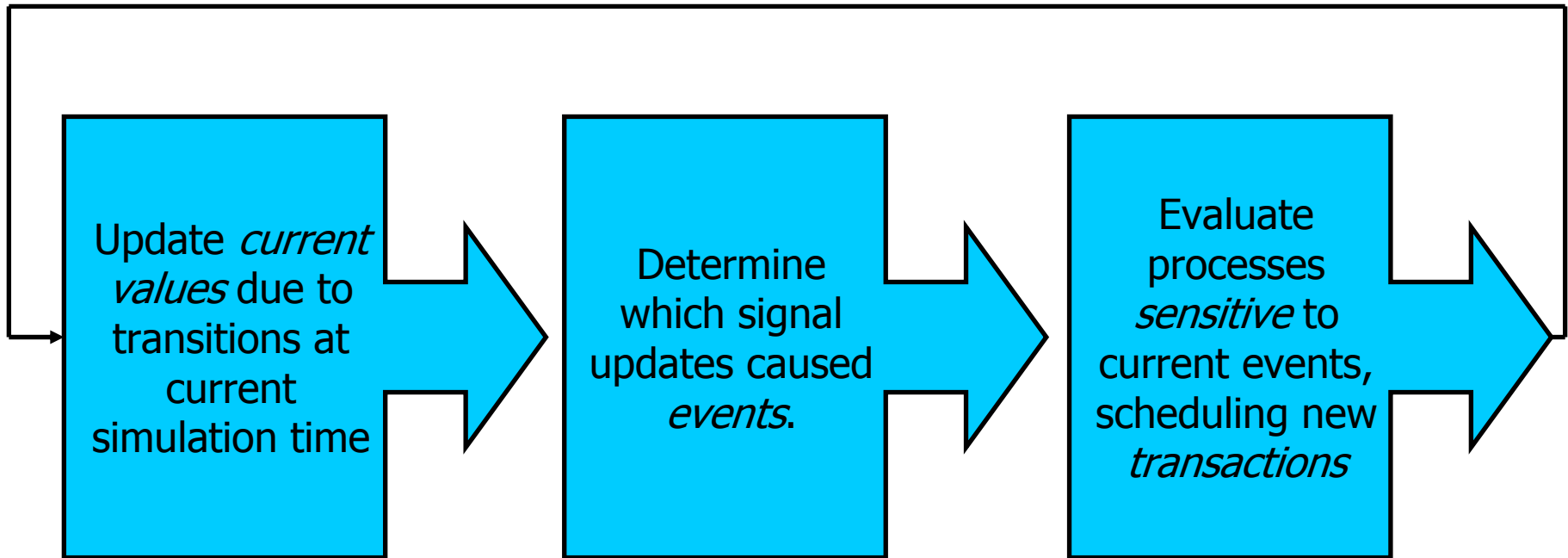
# VHDL Simulation Terminology

- All signals have a *current value* associated with them at all times during a simulation
- A *transaction* is an update of the current value of a signal
- During a simulation cycle in which a signal experiences a transaction, the signal is said to be *active*
- An *event* is a transaction that results in a change of value
- Processes can schedule transactions to take place on their output signals, both at future times or “immediately”.

# VHDL Simulation Cycle

- The simulation cycle governs execution of the simulation.
- The simulation cycle is repeated until simulation terminates.
- Steps in the simulation cycle.
  1. The current time  $T_C$  is assigned the next schedule simulation time  $T_N$ .
  2. Each active signal is updated with its new value.
  3. Processes that are sensitive to signals that have just experienced *events* are marked to resume during the current simulation cycle, as well as processes scheduled to resume at the current time.
  4. Each process that is marked to resume is executed (in no defined order of processes) until (if) it suspends.
  5. The next simulation time  $T_N$  is calculated according to the next time a signal is schedule to become active or a process is scheduled to resume.
- If  $T_N = T_C$  then the next simulation cycle is called a *delta cycle*.

# VHDL Simulation Cycle, Simplified

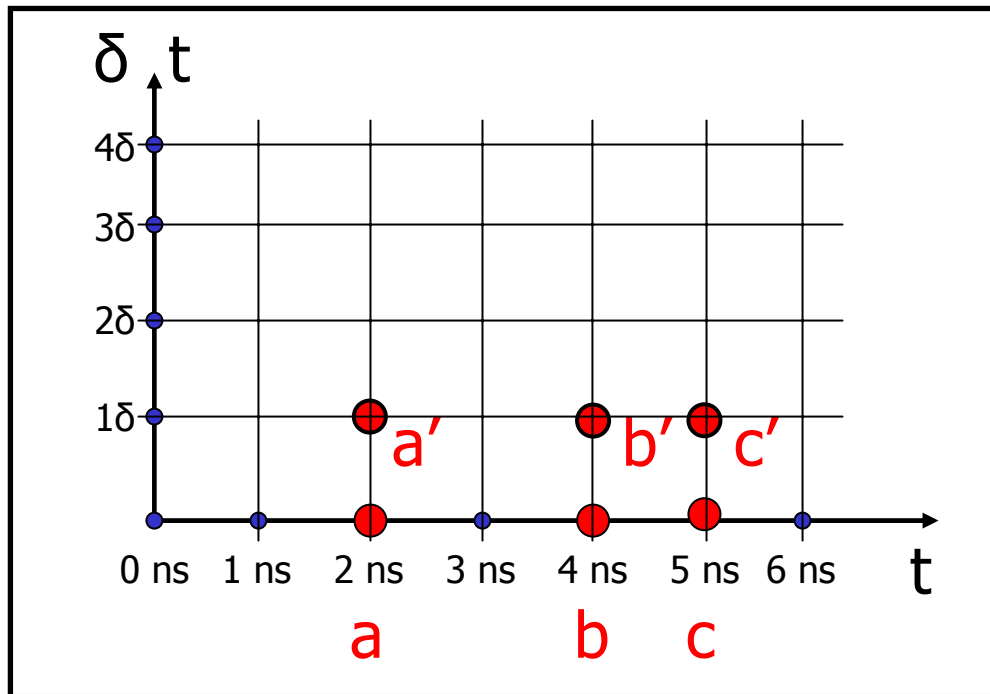


# Delta Time

- Infinitesimally small ( $\delta$ ) advance in time
- The current time  $T_C$  does not advance
- An infinite number of delta time steps can occur between tags in the current time  $T_C$
- “Which delta cycle” is not accessible in the language; “do something immediately” means do it during the next cycle, which will be a delta cycle
- Provides a means of ordering an event and the events that result from it

# More Delta Time

- 
- **a** **b** **c**
- $x \leq '1'$  after 2 ns, '0' after 4ns, '1' after 5 ns
- $y \leq x$  after 0 ns
- 
- 



Scheduled Transactions

● y

● x

# Determinancy in VHDL

- Delta time provides additional ordering information
- The order of execution of processes that resume simultaneously is unspecified – potential indeterminacy
- The separation of events and their resultant events by delta time means that processes can only influence resumptions of their processes during future cycles

Consider if both P1 and P2 were to execute simultaneously:

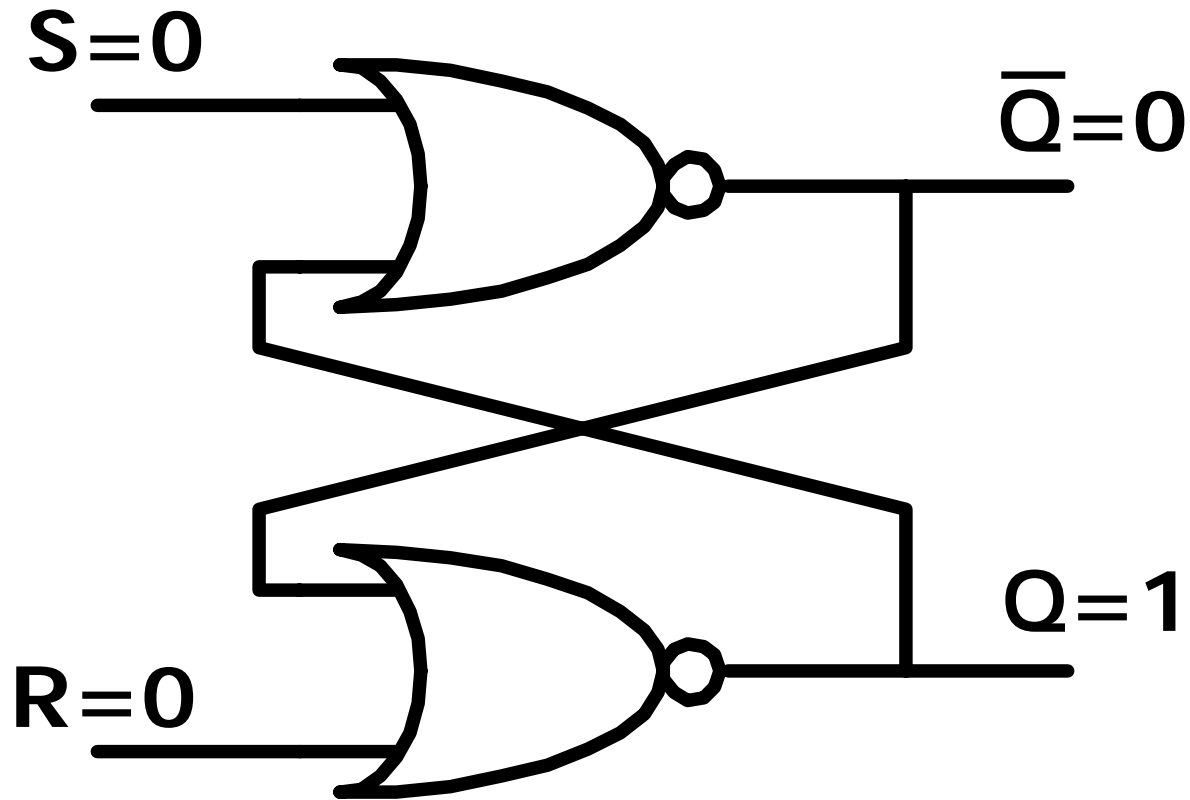
```
P1: ...  
begin  
  
    if rising_edge...  
        c <= '0';  
  
    ...  
End process;
```

```
P2: ...  
begin  
    if rising_edge...  
        if c = '0' then  
            ...  
        end if;  
  
    ...  
end process;
```

# Initial Values

- All signals have an initial value when simulation begins
  - They are defined at elaboration (when the design is “filled out” and prepared for simulation)
  - Note that all processes are run during elaboration as well
- Initialized by leftmost value of type
  - Type `std_ulogic` is (`'U'`, `'X'`, `'0'`, `'1'`, `'Z'`, `'W'`, `'L'`, `'H'`, `'-'`);
- All initial values are ignored by the synthesizer
  - An opportunity for post-place-and-route mismatch!

# VHDL Simulation Example



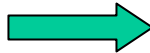
P1:  $Q \leq R \text{ nor } Q\text{bar};$

P2:  $Q\text{bar} \leq S \text{ nor } Q;$

# CSAs Are Processes

- CSAs are a shorthand for writing a process
- A CSA is a process where all signals are on the sensitivity list

```
x <= a or b or c;
```



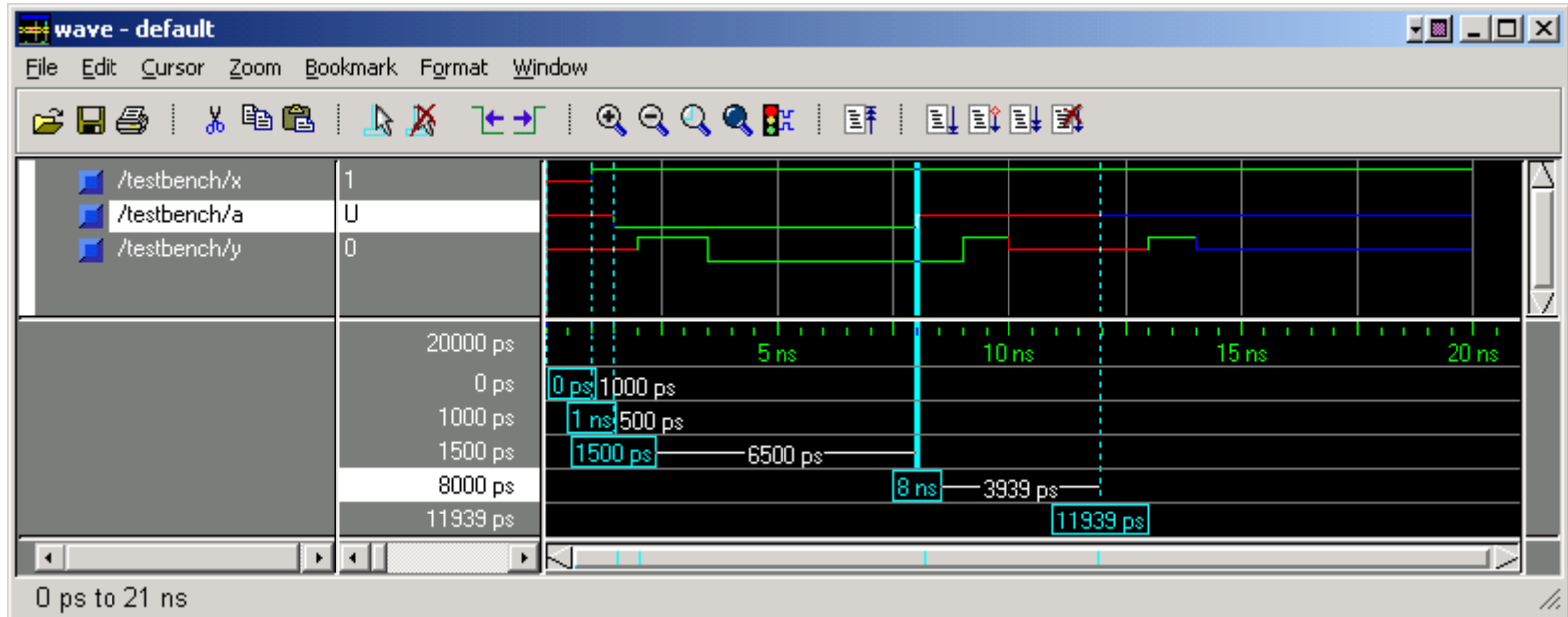
```
genx : process (a, b, c) is  
begin  
    x <= a or b or c;  
end process;
```

```
y <= x after 1 ns,  
    a after 2 ns;
```



```
geny : process (x, a) is  
begin  
    y <= x after 1 ns, a after 2 ns;  
end process;
```

# CSAs Are Processes (II)



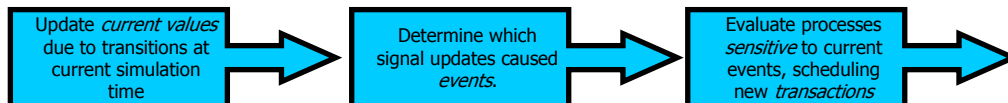
```
y <= x after 1 ns, a after 2 ns;
```

```
x <= '1' after 1 ns;
```

```
a <= '0' after 1.5 ns, 'U' after 8 ns, 'Z' after 12 ns;
```

# VHDL Simulation Example (2)

Event Processing	$t$	$t + \Delta_t$	$t + 2 \times \Delta_t$	$t + 3 \times \Delta_t$
S	0	0	0	0
Q	1	<b>1→0</b>	0	<b>0→0</b>
R	<b>0→1</b>	1	1	1
Qbar	0	0	<b>0→1</b>	1
transaction queue	R→1	Q→0	Qbar→1	Q→0
events	R (P1)	Q (P2)	Qbar (P1)	No more events, $t$ incremented to next scheduled item in queue



P1:  $Q \leq R \text{ nor } Qbar$ ;  
 P2:  $Qbar \leq S \text{ nor } Q$ ;

# Agenda

- Introduce some tools to enable more advanced testbenches
  - Attributes: Many different attributes of a signal are stored in the simulator and can be accessed by knowing the appropriate attribute name.
  - File Input / Output: The VHDL testbench can read and write text files by using builtin and standard packages.
  - Active Testbench (not really a tool, rather just an idea). Making a testbench actually behave like a real device is not as hard if we don't need to make it synthesizable.
- Encourage the use of these tools to make more useful test-benches (i.e. not just a “.do” file replacement)

# Attributes for Testbenching

- **$S'$ delayed(T)**: A signal that takes on the same values as S but is delayed by time T (note that this get the simulation history for S).
- **$S'$ stable(T)**: A Boolean signal that is true if there has been no event on S in the time interval T up to the current time, otherwise false.
- **$S'$ quiet(T)**: A Boolean signal that is true if there is no transaction on S in the time interval T up to the current time, otherwise false.
- **$S'$ transaction**: A signal of type bit that changes value from '0' to '1' or vice versa each time there is a a transaction on S.

# Attributes for Testbenching (II)

- **S'event**: True if there is an event on S in the current simulation cycle, false otherwise
- **S'active**: True if there is a transaction on S in the current simulation cycle, false otherwise.
- **S'last\_event**: The time interval since the last event on S.
- **S'last\_active**: The time interval since the last transaction on S.
- **S'last\_value**: The value of S just before the last event on S.

# Attributes for Testbenching, Example

Check setup time for a DFF:

```
.  
.br/>if clk'event and (clk='1' or clk='H')  
    and (clk'last_value='0' or clk'last_value='L') then  
    assert d'last_event >= Tsu -- setup time  
    report "Timing error: d changed within setup time of clk";  
end if;  
.br/.
```

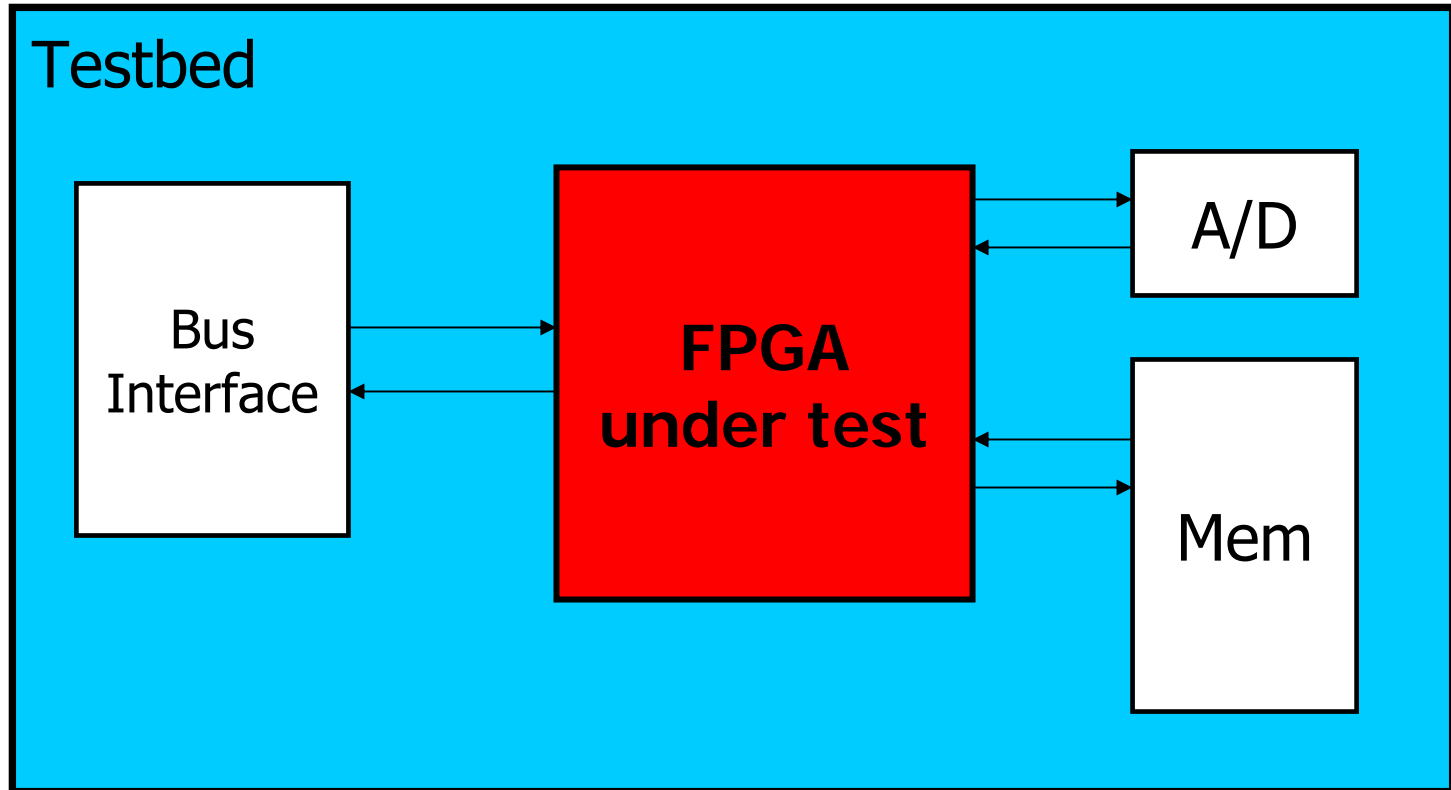
Check pulse width of a clock:

```
.br/>.br/>assert (not clk'event) or clk'delayed'last_event >= Tpw_clk  
    report "Clock frequency too high.";  
.br/.
```

# Active Testbench

- Design is frequently in a system which is quite difficult to test by manual forcing of signals.
  - If your device has two-way communication with other devices, a testbench that actually responds to signals from your device is of great benefit

# Example of System for Testbench



Instead of testing by stimulating all FPGA inputs with test vectors, use VHDL to create models of the A/D, memory, and bus interface, so that the FPGA can be observed in a replica of the actual system.

# File I/O: Modeling Memory

- When memory is part of a system, we'd like a way to get some contents into our memory model in a fast easy way.
  - The contents of memory are often created by some other tool, and thus already exist in electronic form. (e.g. the PIC assembler)
  - FILE I/O is a perfect tool for this

# File I/O: Automated Testing

- Instead of creating a testbench which explicitly tries all possible alternatives, a testbench can use FILE I/O to read a “vector file” which contains a list of all the stimulus in tabular form
- The VHDL testbench can also contain statements that make sure your design is working automatically, and thus doesn't count on you observing the output of the wave window for verification.

# File I/O: VHDL as Test Generator

- A VHDL model can export transitions that occur on all signals to a file in any format for use by automated testing equipment
  - Example: Generate test vectors to program an IC tester for circuit verification, check for stuck nodes etc.
- The system level model can also be used to create stimulus and test assertions for another design tool altogether.
  - Example: A chip designed in MAGIC, uses a program called IRSIM to simulate. VHDL model of system can be used to automatically generate test file in IRSIM format.

# File I/O: Miscellaneous

- Testbench can record informative messages as events happen

# Package std.TextIO

- Built in File I/O facility of VHDL is VERY rudimentary, essentially just enough to read or write a file
- VHDL '87 and '93 do it in two different ways, with '93 being slightly more advanced – we'll discuss '87 first

## In Declarative Section of Architecture

```
file cmdfile : text is in "flashfile.txt";
```

↑            ↑            ↑            ↑  
file name    type        mode        filename on disk

Note: no explicit opening / closing of file... this is done automatically

# TextIO subprograms

```
procedure readline (f : in text; l : out line);  
-- reads a line of text from the file and puts it in l.
```

```
procedure read (l: inout line;.... lots of these....);  
-- used to get elements off of a line of text
```

```
function endfile (f : in text) return boolean;  
-- returns true if the end of the file was reached
```

basic flow is to open a file (with the declaration) and consecutively :

- read a line of text

- use **read**, or its derivatives to extract vhdl elements until the end of the file

# read procedures in std.textio

- `read(l: inout line; value: out bit; good : out bit)`
- `read(l: inout line; value: out bit)`

Variants include this same pair for value types of:

`bit_vector`

`boolean`

`character`

`integer`

`real`

`string`

`time`

Note : no `std_logic_vector`..etc. so you can't have `111Z00Z1`

# write procedures in std.textio

- `procedure write (l : inout line; value : in bit;  
justified: in side := right; field : in width := 0);`

Variants include this same pair for value types of:

bit\_vector

boolean

character

integer

real

string

time

all read and writes operate on a "line", and readline and writeline are used to write that line to a file.

# std\_logic\_textio

- This package extends std.textio by adding the read and write functions for:
  - std\_ulogic
  - std\_ulogic\_vector
  - std\_logic\_vector

# std\_logic\_textio

Using these text I/O packages, we can now read and write files that have various types in them.

For example, an initialization file for a ROM model:

```
111000111111  
001001110110  
110111011101  
110110110110
```

We can make a model that will read through these things in a text file to load its internal data. Why would we do this?

- Simple and easy to change
- No recompiling our design to change the program

Would be better if we could do HEX, since that is more compact, and closely related to the assembler listing file

# Octal and Hex I/O

```
procedure HREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR);
```

```
procedure HREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR;  
GOOD: out BOOLEAN);
```

```
procedure HWRITE(L:inout LINE; VALUE:in STD_ULOGIC_VECTOR;  
JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
```

```
procedure OREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR);
```

```
procedure OREAD(L:inout LINE; VALUE:out STD_ULOGIC_VECTOR;  
GOOD: out BOOLEAN);
```

```
procedure OWRITE(L:inout LINE; VALUE:in STD_ULOGIC_VECTOR;  
JUSTIFIED:in SIDE := RIGHT; FIELD:in WIDTH := 0);
```

This set exists for value type : `std_ulogic_vector`, `std_logic_vector`

# A simple program memory

Assume an adapted listing file romcontents.txt:

```
001  
002  
003  
004  
005  
006  
007  
008  
009  
00a  
00b  
00c  
00d  
00e  
00f  
200  
210  
220  
230  
240  
250  
260  
270
```

No more than the length of our program memory

# program memory

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use std.textio.all;
use ieee.std_logic_textio.all;

entity program is
    Port ( adr : in std_logic_vector(7 downto 0);
          data : out std_logic_vector(11 downto 0);
          clk : in std_logic);
end program;

architecture Behavioral of program is
    file datafile: text is "romcontents.txt";
    type memorytype is array(0 to 255) of std_logic_vector(11
downto 0);
    signal rom : memorytype;
    signal integer_address: integer range 0 to 255;
```

# program memory

```
begin
  clkedge : process(clk)
  begin
    if rising_edge(clk) then
      integer_address <= to_integer(adr);
    end if;
  end process clkedge;

  data <= rom(integer_address);
  .
  .
  .
```

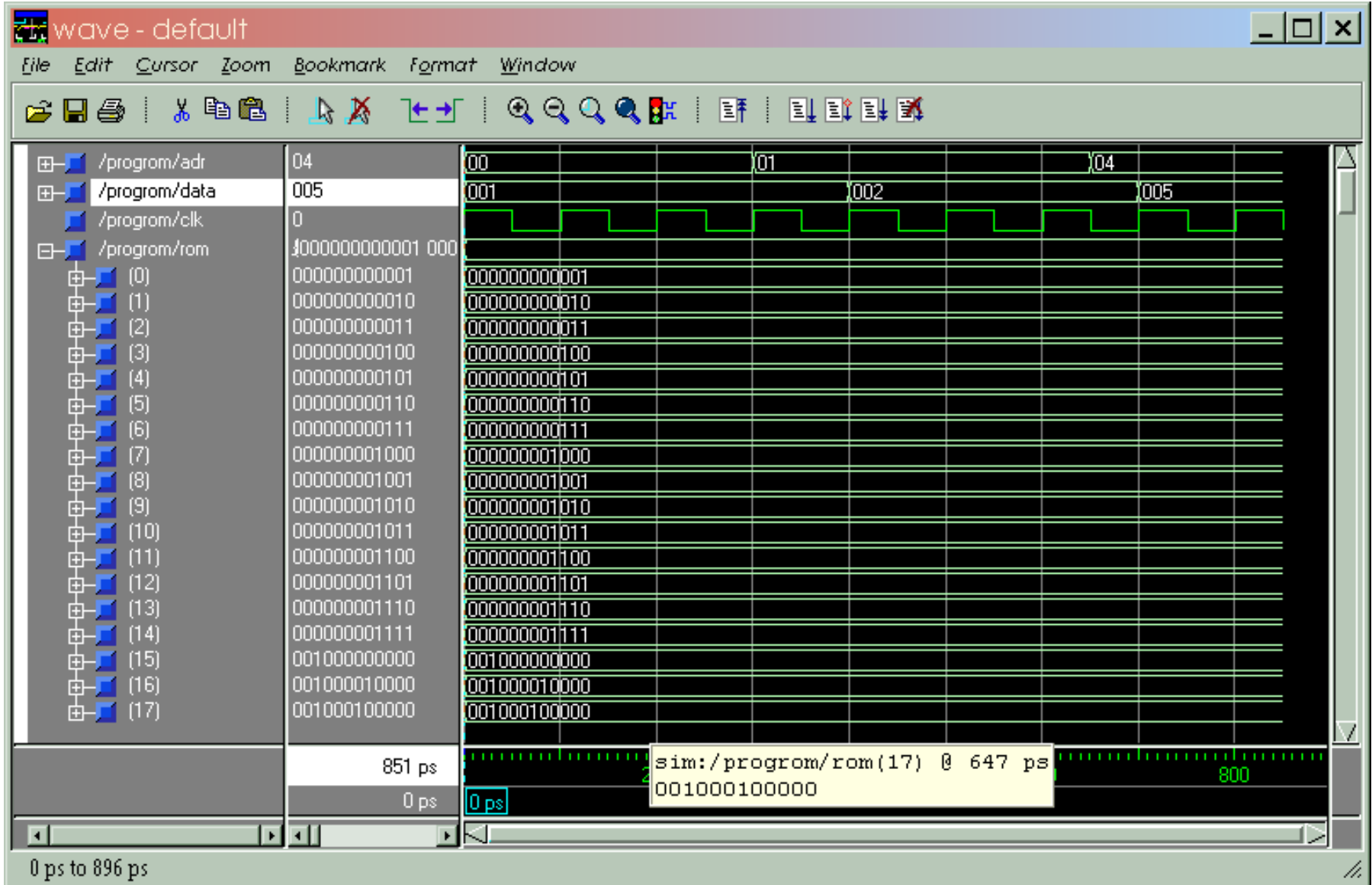
So far, nothing new. We wrote the rest of our model presuming that rom(...) has been filled in some way

# Filling the ROM

```
fillrom : process
    variable line_in : line;
    variable ctr : integer;
    variable readfromfile : std_logic_vector(11
                                                downto 0);
begin
    ctr := 0;
    while not endfile (datafile) loop
        readline(datafile,line_in);
        hread(line_in,readfromfile);
        rom(ctr) <= readfromfile;
        ctr := ctr + 1;
    end loop;
    wait;
end process fillrom;

end Behavioral;
```

# In Action



# VHDL '93 File Handling

'93 can explicitly open and close files using the `file_open` and `file_close` procedures...

Note the usefulness of this, since VHDL has no file "seek" (go to a certain position in the file)

- If you want to traverse a file multiple times
- Read through a file in random order as needed...
- A signal to the module can be used to re-load contents from a file which may have changed...etc.

# VHDL '93 File Handling

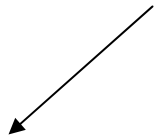
```
process
  file data : text is in "mux.dat"; ←————— 87
begin
  while not endfile (data) loop
    ...
  end loop;
  wait;
end process;
```

These do exactly the same thing

```
process
  file data : text open read_mode is "mux.dat"; ←————— 93
begin
  while not endfile (data) loop
    ...
  end loop;
  wait;
end process;
```

# Using '93 explicit File Handling

```
process                                read_mode, write_mode, append_mode
  file data : text;
begin
  for I in 1 to 2 loop
    file_open(data,"mux.dat",read_mode);
    while not endfile (data) loop
      ...
    end loop;
    file_close(data);
  end loop;
  wait;
end process;
```



# Generating Files for other tools

```
process (Data,Adr,Fast_Clock,rd,wr,cs,reset)
file outfile: TEXT is out "setramfile.txt";
variable L : LINE;
variable databitvector : Bit_Vector(0 to 15);
begin
  if (Reset'Event) then
    if (Reset = '1') then
      WRITE(L, string("h reset"));
    else
      WRITE(L, string("l reset"));
    end if;
    WRITELINE(outfile,L);
  end if;
  if (Adr'Event) then
    WRITE(L,string("set address "));
    WRITE(L,adr);
    WRITELINE(outfile,L);
if (cs'Event) and (cs = '1') then
  WRITE(L,string("h chip_Select"));
```

This process monitors all the signals going into a particular entity, and creates a file for the simulator used to test that entity.

We could make a procedure that took an array of signals and produced a vector for every time a signal changed.

# Automated Testing

- combination of **file I/O** and **asserts** are used to:
  - read a file of test inputs, and desired outputs
  - stimulate the circuit with the inputs
  - compare the outputs with the desired outputs from the file using the assert.
  - print out errors and/or stop on assertion failure.

## Multiplexer Testing File :

```
-- order = a,b,sel result
100 1
110 1
001 0
101 0
111 1
```

# Automated Testing Ex

```
entity mux_test is
end;

use std.textio.all;
architecture test_bench of mux_test is
    component mux
        port(in0,in1,sel : in bit; z: out bit);
    end component;
    signal in0,in1,sel,z : bit;
begin
    UUT : mux port map (in0,in1,sel,z);

    ... test process goes here ...
end test_bench;
```

# Automated Testing Ex

```
process
  file data : text is in "mux.dat";
  variable sample : line;
  variable in0_var, in1_var, sel_var, z_var : bit
begin
  while not endfile(data) loop
    readline(data, sample);
    read(sample, in0_var);
    read(sample, in1_var);
    read(sample, sel_var);
    read(sample, z_var);
    in0 <= in0_var;
    in1 <= in1_var;
    sel <= sel_var;
    wait for 10 ns;
    assert z = z_var report "z incorrect" severity error;
  end loop;
  wait;
end;
```

# Assertions

Used to check to make sure a condition is true, if not, a **report** is generated by the simulator.

```
assert alu_result = "11001100"  
    report "alu has made an error!"  
    severity error;
```



Severity can be: [note, warning, error, failure, fatal]  
Simulator can be told to ignore or break on all but fatal (which always breaks)

We show the string as a constant string, but it can be any string. Functions can be written to convert values to strings and print them out