

Picoblaze IRQ Example

- Modify the top-level design to hook up the interrupt signal and “ack”

```
pblaze : kcpsm3 port map
(
  address => address,
  instruction => instruction,
  port_id => port_id,
  write_strobe => write_strobe,
  out_port => out_port,
  read_strobe => read_strobe,
  in_port => in_port_reg,
  interrupt => pico_interrupt,
  interrupt_ack => open,
  reset => reset,
  clk => clk
);
```

ACK is not necessarily needed – **if** we guarantee the interrupt stays high till the Pico will see it.

Picoblaze IRQ Example

- Generate the Interrupt

```
pico_interrupt <= sec2_enable or sec2_enable_d1 when rising_edge(clk);  
sec2_enable_d1 <= sec2_enable when rising_edge(clk);
```

```
makeirqsig : clkdivider generic map (divideby => 50000000*4)  
  port map (clk=>clk, reset=>reset, pulseout => sec4_enable);
```

OR

```
process(clk,reset)  
begin  
  if reset='1' then pico_interrupt <= '0';  
  elsif rising_edge(clk) then  
    if sec2_enable = '1' then  
      pico_interrupt <= '1';  
    elsif irq_ack = '1' then  
      pico_interrupt <= '0';  
    end if;  
  end if;  
end process;  
  
makeirqsig : clkdivider generic map (divideby => 50000000*8)  
  port map (clk=>clk, reset=>reset, pulseout => sec8_enable);
```

Picoblaze IRQ Example

- Handle the Interrupt

MYISR:

```
LOAD s3,5a  
LOAD s4,5a  
RETURNI ENABLE
```

ADDRESS 3FF

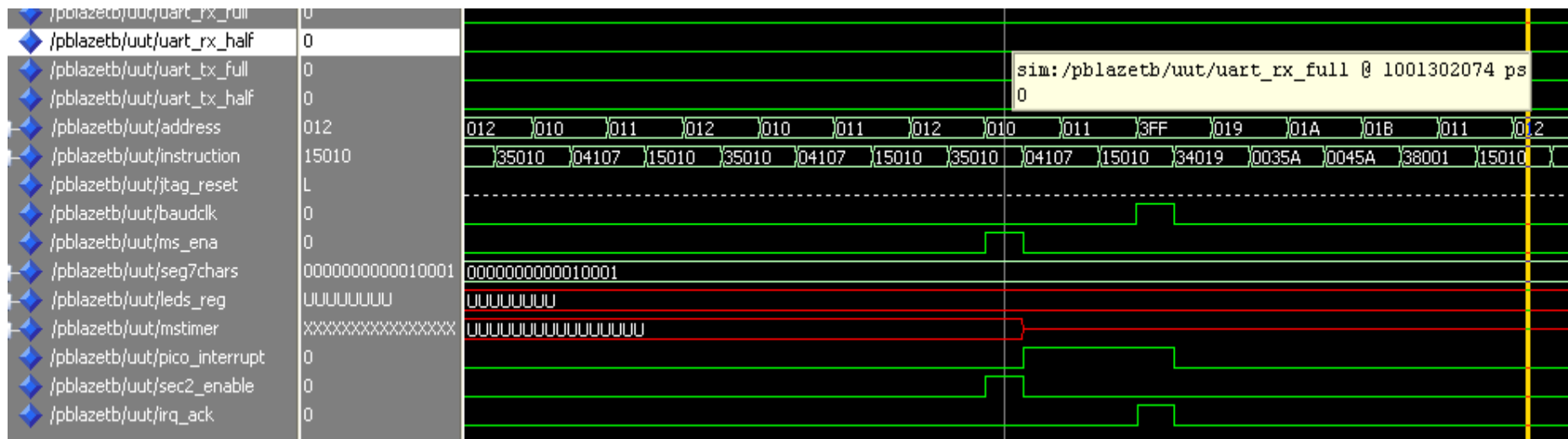
```
JUMP MYISR
```

Re-enable interrupts when returning



Picoblaze sets PC to this value when IRQ happens

Picoblaze IRQ Example



Packages

- Packages are used to collect together a set of closely related **sub-programs, parts, or types**.
- For example, `std_logic_1164` is a package that defines four types (`std_ulogic`, `std_logic`, `std_ulogic_vector`, `std_logic_vector`) **as well as the operations that can be performed on them**.
- A package is comprised of a package header and a package body

```
package test_parts is
    constant cycleW : integer := 3;
    constant addrW  : integer := 22;
    constant dataW  : integer := 16;

    procedure generate_clock (Tperiod,
                              Tpulse,
                              Tphase : in time;
                              signal clk : out std_logic);
end package test_parts;

package body test_parts is

    procedure generate_clock ( Tperiod,
                              Tpulse,
                              Tphase : in time;
                              signal clk : out std_logic ) is

    begin
        wait for Tphase;
        loop
            clk <= '0', '1' after Tpulse;
            wait for Tperiod;
        end loop;
    end procedure generate_clock;

end package body test_parts;
```



```
use work.my_components.all;

architecture behavior of simplecircuit is
    signal andout1, andout2 : std_logic;
begin
    U6: and2 port map (a1,b1,andout1);
    U7: and2 port map (a2,b2,andout2);
    sigout <= andout1 or andout2;
end behavior;
```

```
library ieee;
use ieee.std_logic_1164.all;

use work.test_parts.all;

.
.
.
```

Packages and Operator Overloading

- It is generally the case that the operations that you want to perform on your newly defined types (say `std_logic_vector`) are the same as the VHDL `std` operators.
- VHDL supports “operator overloading”
 - multiple versions of the operators can be available, with the same name
 - which version is “called” depends on the parameters and their types

```
-----  
-- overloaded logical operators  
-----  
FUNCTION "and" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;  
FUNCTION "nand" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;  
FUNCTION "or" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;  
FUNCTION "nor" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;  
FUNCTION "xor" ( l : std_ulogic; r : std_ulogic ) RETURN UX01;  
-- function "xnor" ( l : std_ulogic; r : std_ulogic ) return ux01;  
function xnor ( l : std_ulogic; r : std_ulogic ) return ux01;  
FUNCTION "not" ( l : std_ulogic ) RETURN UX01;  
  
-----  
-- vectorized overloaded logical operators  
-----  
FUNCTION "and" ( l, r : std_logic_vector ) RETURN std_logic_vector;  
FUNCTION "and" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;  
  
FUNCTION "nand" ( l, r : std_logic_vector ) RETURN std_logic_vector;  
FUNCTION "nand" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;  
  
FUNCTION "or" ( l, r : std_logic_vector ) RETURN std_logic_vector;  
FUNCTION "or" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;  
  
FUNCTION "nor" ( l, r : std_logic_vector ) RETURN std_logic_vector;  
FUNCTION "nor" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;  
  
FUNCTION "xor" ( l, r : std_logic_vector ) RETURN std_logic_vector;  
FUNCTION "xor" ( l, r : std_ulogic_vector ) RETURN std_ulogic_vector;
```

from `std_logic_1164` package header

bitwise boolean operators are overloaded for `std_logic`, so their behavior is now governed by the package body

Example Package from Lab 6

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

package myparts is

component kcpsm3 is
    Port (
        address : out std_logic_vector(9 downto 0);
        instruction : in std_logic_vector(17 downto 0);
        port_id : out std_logic_vector(7 downto 0);
        write_strobe : out std_logic;
        out_port : out std_logic_vector(7 downto 0);
        read_strobe : out std_logic;
        in_port : in std_logic_vector(7 downto 0);
        interrupt : in std_logic;
        interrupt_ack : out std_logic;
        reset : in std_logic;
        clk : in std_logic);
    end component;

... and everything else
... no package body required
```

In toplevel.vhd : “use work.myparts.all”