---

-- wine_example.vhd
---

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
### Std.Logic Data Type

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘U’</td>
<td>Uninitialized</td>
</tr>
<tr>
<td>‘X’</td>
<td>Forcing unknown</td>
</tr>
<tr>
<td>‘0’</td>
<td>Forcing 0</td>
</tr>
<tr>
<td>‘1’</td>
<td>Forcing 1</td>
</tr>
<tr>
<td>‘Z’</td>
<td>High impedance</td>
</tr>
<tr>
<td>‘W’</td>
<td>Weak unknown</td>
</tr>
<tr>
<td>‘L’</td>
<td>Weak 0</td>
</tr>
<tr>
<td>‘H’</td>
<td>Weak 1</td>
</tr>
<tr>
<td>‘-’</td>
<td>Don’t care</td>
</tr>
</tbody>
</table>

- A `std_logic_vector` is simply an array of `std_logic` bits.
- `std_logic_1164` package defines these data types and provides conversion functions.
- `std_logic_arith` package provides arithmetic functions.
- `std_logic_unsigned` package indicates vectors are unsigned.
- `std_logic_signed` package indicates vectors are signed.
Nondeterminism Package

- `selection(num)` function - returns a random value between 1 and `num`.
- `selection(num, size)` function - returns random value between 1 and `num` encoded as a `std_logic_vector` with `size` bits.
- `delay(l, u)` function - returns a random time between `l` and `u` ns.
package nondeterminism is
    shared variable s1: integer := 844396720;
    shared variable s2: integer := 821616997;
    -- Returns a number between 1 and num.
    impure function selection(constant num: in integer) return integer;
    -- Returns a std_logic_vector of size bits between 1 and num.
    impure function selection(constant num: in integer;
                                constant size: in integer) return std_logic_vector;
    -- Returns random delay between lower and upper.
    impure function delay(constant l: in integer; constant u: in integer) return time;
end nondeterminism;
**Communication Channels**

- *Channel* is used as a point-to-point means of communication between two concurrently operating processes.

- Channel package (see channel.vhd in Appendix A.2) includes:
  - *channel* data type
  - *send* procedures
  - *receive* procedures
  - *init_channel* function
  - *active* function
  - *passive* function
  - *probe* function
package channel is
  constant MAX_BIT_WIDTH:natural:=32;
  subtype datatype is std_logic_vector((MAX_BIT_WIDTH-1) downto 0);
  constant dataZ:datatype:=datatype'(others => 'Z');
  constant data0:datatype:=datatype'(others => '0');
  constant dataACK:datatype:=datatype'(others => '1');
  type channel is record
    dataright, dataleft:datatype;
    pending_send, pending_recv, sync:std_logic;
  end record;
  type bools is array (natural range <>) of boolean;
procedure send(signal c1: inout channel);
procedure send(signal c1: inout channel; signal d1: inout std_logic);
procedure send(signal c1: inout channel; signal d1: inout std_logic;
    signal c2: inout channel; signal d2: inout std_logic);
procedure send(signal c1: inout channel; signal d1: inout std_logic;
    signal c2: inout channel; signal d2: inout std_logic;
    signal c3: inout channel; signal d3: inout std_logic);
procedure send(signal c1: inout channel; signal d1: inout std_logic_vector);
procedure send(signal c1: inout channel; signal d1: inout std_logic_vector;
    signal c2: inout channel; signal d2: inout std_logic_vector);
procedure send(signal c1: inout channel; signal d1: inout std_logic_vector;
    signal c2: inout channel; signal d2: inout std_logic_vector;
    signal c3: inout channel; signal d3: inout std_logic_vector);
procedure receive(signal c1:inout channel);
procedure receive(signal c1:inout channel; signal d1:inout std_logic);
procedure receive(signal c1:inout channel; signal d1:inout std_logic;
signal c2:inout channel; signal d2:inout std_logic);
procedure receive(signal c1:inout channel; signal d1:inout std_logic;
signal c2:inout channel; signal d2:inout std_logic;
signal c3:inout channel; signal d3:inout std_logic);
procedure receive(signal c1:inout channel; signal d1:inout std_logic_vector);
procedure receive(signal c1:inout channel; signal d1:inout std_logic_vector;
signal c2:inout channel; signal d2:inout std_logic_vector);
procedure receive(signal c1:inout channel; signal d1:inout std_logic_vector;
signal c2:inout channel; signal d2:inout std_logic_vector;
signal c3:inout channel; signal d3:inout std_logic_vector);
function init_channel return channel;
function active return channel;
function passive return channel;
function probe(signal chan:in channel) return boolean;
end channel;
entity wine_example is
end wine_example;
architecture behavior of wine_example is
-- declarations
begin
-- concurrent statements
end behavior;
type wine_list is (cabernet, merlot, zinfandel, chardonnay, sauvignon_blanc, pinot_noir, riesling, bubbly);
signal wine_drunk:wine_list;
signal WineryShop:channel:=init_channel;
signal ShopPatron:channel:=init_channel;
signal bottle:std_logic_vector(2 downto 0):="000";
signal shelf:std_logic_vector(2 downto 0);
signal bag:std_logic_vector(2 downto 0);
Concurrent Processes: winery

```plaintext
winery: process
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  send(WineryShop,bottle);
end process winery;
```
Concurrent Processes: shop

```
shop: process
begin
    receive(WineryShop, shelf);
    send(ShopPatron, shelf);
end process shop;
```
Concurrent Processes: patron

```vhdl
patron: process
begin
    receive(ShopPatron, bag);
    wine_drunk <= wine_list'val(conv_integer(bag));
end process patron;
```
Block Diagram for *wine_shop*

- **Winery**
- **WineryShop**
- **Shop**
- **ShopPatron**
- **Patron**
entity shop is
  port (wine_delivery: inout channel:=init_channel;
        wine_selling: inout channel:=init_channel);
end shop;
arithmetic behavior of shop is
  signal shelf: std_logic_vector(2 downto 0);
begin
  shop: process
  begin
    receive (wine_delivery, shelf);
    send (wine_selling, shelf);
  end process shop;
end behavior;
entity winery is
  port (wine_shipping: inout channel:=init_channel);
end winery;
architecture behavior of winery is
  signal bottle: std_logic_vector(2 downto 0):="000";
begin
  winery: process
  begin
    bottle <= selection(8,3);
    wait for delay(5,10);
    send(wine_shipping,bottle);
  end process winery;
end behavior;
entity patron is
  port (wine_buying: inout channel := init_channel);
end patron;

architecture behavior of patron is
  type wine_list is (cabernet, merlot, zinfandel, chardonnay, sauvignon_blanc, pinot_noir, riesling, bubbly);
  signal wine_drunk: wine_list;
  signal bag: std_logic_vector(2 downto 0);
begin
  patron: process
  begin
    receive (wine_buying, bag);
    wine_drunk <= wine_list'val (conv_integer (bag));
  end process patron;
end behavior;
-- wine_example2.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.channel.all;
entity wine_example is
end wine_example;
architecture structure of wine_example is
-- component and signal declarations
begin
-- component instantiations
end structure;
component winery
   port (wine_shipping: inout channel);
end component;
component shop
   port (wine_delivery: inout channel;
       wine_selling: inout channel);
end component;
component patron
   port (wine_buying: inout channel);
end component;
signal WineryShop:channel:=init_channel;
signal ShopPatron:channel:=init_channel;
begin
    THE_WINERY: winery
        port map (wine_shipping => WineryShop);
    THE_SHOP: shop
        port map (wine_delivery => WineryShop,
                  wine_selling => ShopPatron);
    THE_PATRON: patron
        port map (wineBuying => Shop Patron);
end structure;
architecture new_structure of wine_example is
-- component declarations
-- channel declarations
begin
-- winery
  OLD_SHOP:shop
    port map (wine_delivery => WineryShop,
              wine_selling  => ShopNewShop);
  NEW_SHOP:shop
    port map (wine_delivery => ShopNewShop,
              wine_selling  => NewShopPatron);
-- patron
end new_structure;
Deterministic Selection: if-then-else

```vhdl
winery2: process begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  if (wine_list'val(conv_integer(bottle)) = merlot) then
    send(WineryNewShop,bottle);
  else
    send(WineryOldShop,bottle);
  end if;
end process winery2;
```
Deterministic Selection: case

```vhdl
winery3: process
begin
    bottle <= selection(8,3);
    wait for delay(5,10);
    case (wine_list'val(conv_integer(bottle))) is
        when merlot =>
            send(WineryNewShop,bottle);
        when others =>
            send(WineryOldShop,bottle);
    end case;
end process winery3;
```
```vhdl
winery4: process
variable z: integer;
begin
    bottle <= selection(8, 3);
    wait for delay(5, 10);
    z := selection(2);
    if (z = 1) then
        send(WineryNewShop, bottle);
    else
        send(WineryOldShop, bottle);
    end if;
end process winery4;
```
Non-deterministic Selection: case

```vhdl
winery5: process
variable z: integer;
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  z:=selection(2);
  case z is
    when 1 =>
      send(WineryNewShop,bottle);
    when others =>
      send(WineryOldShop,bottle);
  end case;
end process winery5;
```

Repetition: for loops

```vhdl
winery6: process
begin
  for i in 1 to 4 loop
    bottle <= selection(8,3);
    wait for delay(5,10);
    send(WineryOldShop,bottle);
  end loop;
  for i in 1 to 3 loop
    bottle <= selection(8,3);
    wait for delay(5,10);
    send(WineryNewShop,bottle);
  end loop;
end process winery6;
```
```vhdl
winery7: process
begin
    while (wine_list'val(conv_integer(bottle)) /= merlot)
        loop
            bottle <= selection(8,3);
            wait for delay(5,10);
            send(WineryOldShop,bottle);
        end loop;
    bottle <= selection(8,3);
    wait for delay(5,10);
    send(WineryNewShop,bottle);
end process winery7;
```
Repetition: infinite loops

winery8: process
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  send(WineryOldShop, bottle);
  loop
    bottle <= selection(8,3);
    wait for delay(5,10);
    send(WineryNewShop, bottle);
  end loop;
end process winery8;
producer: process
begin
    send(X,x);
    send(Y,y);
end process producer;

consumer: process
begin
    receive(Y,a);
    receive(X,b);
end process consumer;
patron2: process begin
  if (probe(OldShopPatron)) then
    receive(OldShopPatron,bag);
    wine_drunk <= wine_list'val(conv_integer(bag));
  elsif (probe(NewShopPatron)) then
    receive(NewShopPatron,bag);
    wine_drunk <= wine_list'val(conv_integer(bag));
  end if;
  wait for delay(5,10);
end process patron2;
winery9: process
begin
    bottle1 <= selection(8,3);
    bottle2 <= selection(8,3);
    wait for delay(5,10);
    send(WineryOldShop,bottle1,WineryNewShop,bottle2);
end process winery9;
patron3 : process
begin
   receive (OldShopPatron, bag1, NewShopPatron, bag2);
   wine_drunk1 <= wine_list'val(conv_integer(bag1));
   wine_drunk2 <= wine_list'val(conv_integer(bag2));
end process patron3;
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>rd := rs + rt</td>
<td>add r1, r2, r3</td>
</tr>
<tr>
<td>sub</td>
<td>rd := rs − rt</td>
<td>sub r1, r2, r3</td>
</tr>
<tr>
<td>and</td>
<td>rd := rs &amp; rt</td>
<td>and r1, r2, r3</td>
</tr>
<tr>
<td>or</td>
<td>rd := rs</td>
<td>rt</td>
</tr>
<tr>
<td>lw</td>
<td>rt := mem[rs + offset]</td>
<td>lw r1, (32)r2</td>
</tr>
<tr>
<td>sw</td>
<td>mem[rs + offset] := rt</td>
<td>sw r1, (32)r2</td>
</tr>
<tr>
<td>beq</td>
<td>if (rs==rt) then</td>
<td>beq r1, r2, Loop</td>
</tr>
<tr>
<td></td>
<td>PC := PC + offset</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>PC := address</td>
<td>j Loop</td>
</tr>
</tbody>
</table>
## MiniMIPS: ISA

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Opcode</th>
<th>Func</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>sub</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>and</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>or</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>lw</td>
<td>35</td>
<td>n/a</td>
</tr>
<tr>
<td>sw</td>
<td>43</td>
<td>n/a</td>
</tr>
<tr>
<td>beq</td>
<td>4</td>
<td>n/a</td>
</tr>
<tr>
<td>j</td>
<td>2</td>
<td>n/a</td>
</tr>
</tbody>
</table>
## MiniMIPS: Instruction Formats

### Register instructions

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>func</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x6</td>
<td>0x5</td>
<td>0x5</td>
<td>0x5</td>
<td>0x5</td>
<td>0x6</td>
</tr>
</tbody>
</table>

### Load/store/branch instructions

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x6</td>
<td>0x5</td>
<td>0x5</td>
<td>0x16</td>
</tr>
</tbody>
</table>

### Jump instructions

<table>
<thead>
<tr>
<th>opcode</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x6</td>
<td>0x26</td>
</tr>
</tbody>
</table>
-- Entity/architecture declarations
-- ieee stuff
use work.channel.all;
entity minimips is
end minimips;
architecture structure of minimips is
-- Component declarations
-- Signal declarations
begin
-- Component instantiations
end structure;
MiniMIPS: imem.vhd

-- ieee stuff
use work.nondeterminism.all;
use work.channel.all;
entity imem is
  port(address:inout channel:=init_channel;
       data:inout channel:=init_channel);
end imem;
architecture behavior of imem is
  type memory is array (0 to 7) of
    std_logic_vector(31 downto 0);
  signal addr:std_logic_vector(31 downto 0);
  signal instr:std_logic_vector(31 downto 0);
begin
process
  variable imem:memory:=(
    X"8c220000", -- L: lw r2,0(r1)
    ...
  );
begin
  receive(address,addr);
  instr <= imem(conv_integer(addr(2 downto 0)));
  wait for delay(5,10);
  send(data,instr);
end process;
end behavior;
entity fetch is
  port (imem_address: inout channel:=init_channel;
        imem_data: inout channel:=init_channel;
        decode_instr: inout channel:=init_channel;
        branch_decision: inout channel:=init_channel);
end fetch;
architecture behavior of fetch is
  signal PC: std_logic_vector(31 downto 0) :=
      (others => '0');
  signal instr: std_logic_vector(31 downto 0);
  signal bd: std_logic;
  alias opcode: std_logic_vector(5 downto 0) is
      instr(31 downto 26);
  alias offset: std_logic_vector(15 downto 0) is
      instr(15 downto 0);
  alias address: std_logic_vector(25 downto 0) is
      instr(25 downto 0);
begin
process
  variable branch_offset: std_logic_vector (31 downto 0);
begin
  send (imem_address, PC);
  receive (imem_data, instr);
  PC <= PC + 1;
  wait for delay (5, 10);
  case opcode is
  when "000110" => -- j
    PC <= (PC(31 downto 26) & address);
  wait for delay (5, 10);
when "000100" => -- beq
    send(decode_instr,instr);
    receive(branch_decision,bd);
    if (bd = '1') then
        branch_offset(31 downto 16) :=
            (others=>instr(15));
        branch_offset(15 downto 0) := offset;
        PC <= PC + branch_offset;
        wait for delay(5,10);
    end if;
    when others =>
        send(decode_instr,instr);
    end case;
end process;
end behavior;
entity decode is
    port (decode_instr: inout channel:=init_channel;
          execute_op: inout channel:=init_channel;
          execute_rs: inout channel:=init_channel;
          execute_rt: inout channel:=init_channel;
          execute_rd: inout channel:=init_channel;
          execute_func: inout channel:=init_channel;
          execute_offset: inout channel:=init_channel;
          dmem_datain: inout channel:=init_channel;
          dmem_dataout: inout channel:=init_channel);
end decode;
type reg_array is array (0 to 7) of std_logic_vector(31 downto 0);
signal instr:std_logic_vector(31 downto 0);
alias op:std_logic_vector(5 downto 0) is instr(31 downto 26);
alias rs:std_logic_vector(2 downto 0) is instr(23 downto 21);
alias rt:std_logic_vector(2 downto 0) is instr(18 downto 16);
alias rd:std_logic_vector(2 downto 0) is instr(13 downto 11);
alias func:std_logic_vector(5 downto 0) is instr(5 downto 0);
alias offset:std_logic_vector(15 downto 0) is instr(15 downto 0);
signal registers:reg_array := (X"00000000",...);
signal reg_rs:std_logic_vector(31 downto 0);
signal reg_rt:std_logic_vector(31 downto 0);
signal reg_rd:std_logic_vector(31 downto 0);
process
begin
receive(decode_instr,instr);
reg_rs <= reg(conv_integer(rs));
reg_rt <= reg(conv_integer(rt));
wait for delay(5,10);
send(execute_op,op);
case op is
when "000000" => -- ALU op
  send(execute_func,func,execute_rs,reg_rs,
       execute_rt,reg_rt);
  receive(execute_rd,reg_rd);
  reg(conv_integer(rd)) <= reg_rd;
  wait for delay(5,10);
when "000100" => -- beq  
  send(execute_rs,reg_rs,execute_rt,reg_rt);
when "100011" => -- lw  
  send(execute_rs,reg_rs,execute_offset,offset);
  receive(dmem_dataout,reg_rt);
  reg(conv_integer(rt)) <= reg_rt;
wait for delay(5,10);
when "101011" => -- sw  
  send(execute_rs,reg_rs,execute_offset,offset, 
       dmem_datain,reg_rt);
when others => -- undefined
    assert false
    report "Illegal instruction"
    severity error;
end case;
end process;
end behavior;
entity execute is
  port (execute_op: inout channel:=init_channel;
          execute_rs: inout channel:=init_channel;
          execute_rt: inout channel:=init_channel;
          execute_rd: inout channel:=init_channel;
          execute_func: inout channel:=init_channel;
          execute_offset: inout channel:=init_channel;
          dmem_addr: inout channel:=init_channel;
          dmem_rw: inout channel:=init_channel;
          branch_decision: inout channel:=init_channel);
end execute;
architecture behavior of execute is
  signal rs:std_logic_vector(31 downto 0);
  signal rt:std_logic_vector(31 downto 0);
  signal rd:std_logic_vector(31 downto 0);
  signal op:std_logic_vector(5 downto 0);
  signal func:std_logic_vector(5 downto 0);
  signal offset:std_logic_vector(15 downto 0);
  signal rw:std_logic;
  signal bd:std_logic;
begin
process
  variable addr_offset: std_logic_vector(31 downto 0);
begin
  receive(execute_op, op);
  case op is
    when "000100" => -- beq
      receive(execute_rs, rs, execute_rt, rt);
      if (rs = rt) then bd <= '1';
      else bd <= '0';
    end if;
    wait for delay(5, 10);
  send(branch_decision, bd);
when "000000" => -- ALU op
    receive(execute_func, func, execute_rs, rs, execute_rt, rt);
    case func is
        when "100000" => -- add
            rd <= rs + rt;
        when "100010" => -- sub
            rd <= rs - rt;
        when "100100" => -- and
            rd <= rs and rt;
        when "100101" => -- or
            rd <= rs or rt;
        when others =>
            rd <= (others => 'X'); -- undefined
    end case;
    wait for delay(5,10);
    send(execute_rd, rd);
when "100011" => -- lw
  receive(execute_rs,rs,execute_offset,offset);
  addr_offset(31 downto 16) :=
       (others => offset(15));
  addr_offset(15 downto 0) := offset;
  rd <= rs + addr_offset;
  rw <= '1';
  wait for delay(5,10);
  send(dmem_addr,rd);
  send(dmem_rw,rw);
when "101011" => -- sw
  receive(execute_rs, rs, execute_offset, offset);
  addr_offset(31 downto 16) :=
      (others => offset(15));
  addr_offset(15 downto 0) := offset;
  rd <= rs + addr_offset;
  rw <= '0';
  wait for delay(5,10);
  send(dmem_addr, rd);
  send(dmem_rw, rw);
when others => -- undefined
  assert false
      report "Illegal instruction" severity error;
end case;
end process;
end behavior;
entity dmem is
  port (address: inout channel:=init_channel;
         data_in: inout channel:=init_channel;
         data_out: inout channel:=init_channel;
         read_write: inout channel:=init_channel);
end dmem;
architecture behavior of dmem is
  type memory is array (0 to 7) of
    std_logic_vector(31 downto 0);
  signal addr: std_logic_vector(31 downto 0);
  signal d: std_logic_vector(31 downto 0);
  signal rw: std_logic;
  signal dmem: memory:=(X"00000000",...);
begin
receive(address, addr);
receive(read_write, rw);
case rw is
when '1' =>
    d <= dmem(conv_integer(addr(2 downto 0)));
    wait for delay(5,10);
    send(data_out, d);
when '0' =>
    receive(data_in, d);
    dmem(conv_integer(addr(2 downto 0))) <= d;
    wait for delay(5,10);
when others =>
    wait for delay(5,10);
end case;
- $r1$ contains 1.
- $r2$ contains 2.

\begin{align*}
\text{add } & r1, r2, r2 \\
\text{add } & r4, r1, r1
\end{align*}
signal reg_locks:booleans(0 to 7):=
  (others => false);
signal decode_to_wb:channel:=init_channel;
signal wb_instr:std_logic_vector(31 downto 0);
alias wb_op:std_logic_vector(5 downto 0) is
  wb_instr(31 downto 26);
alias wb_rt:std_logic_vector(2 downto 0) is
  wb_instr(18 downto 16);
alias wb_rd:std_logic_vector(2 downto 0) is
  wb_instr(13 downto 11);
signal lock:channel:=init_channel;
begin
receive(decode_instr,instr);
if ((reg_locks(conv_integer(rs))) or
    (reg_locks(conv_integer(rt)))) then
    wait until ((not reg_locks(conv_integer(rs))) and
                (not reg_locks(conv_integer(rt))));
end if;
reg_rs <= reg(conv_integer(rs));
reg_rt <= reg(conv_integer(rt));
send(execute_op,op);
wait for delay(5,10);
when "000000" => -- ALU op
    send(execute_func,func,execute_rs,reg_rs,
            execute_rt,reg_rt);
    send(decode_to_wb,instr);
    receive(lock);
when "100011" => -- lw
    send(execute_rs,reg_rs,execute_offset,offset);
    send(decode_to_wb,instr);
    receive(lock);
writeback: process
begin
  receive (decode_to_wb, wb_instr);
  case wb_op is
  when "000000" => -- ALU op
    reg_locks (conv_integer (wb_rd)) <= true;
    wait for 1 ns;
    send (lock);
  receive (execute_rd, reg_rd);
  reg (conv_integer (wb_rd)) <= reg_rd;
  wait for delay (5,10);
  reg_locks (conv_integer (wb_rd)) <= false;
  wait for delay (5,10);
when "100011" => -- lw
  reg_locks(conv_integer(wb_rt)) <= true;
  wait for 1 ns;
  send(lock);
  receive(dmem_dataout,reg_rd);
  reg(conv_integer(wb_rt)) <= reg_rd;
  wait for delay(5,10);
  reg_locks(conv_integer(wb_rt)) <= false;
  wait for delay(5,10);
  when others => -- undefined
    wait for delay(5,10);
  end case;
end process;
end behavior;
Summary

- *std_logic* data type
- Nondeterminism package
- Channel package
- Send and receive procedures
- Structural modeling
- Selection and repetition
- The probe
- Parallel composition
- MiniMIPS