VHDL
Complex Sequential Systems

EL 310
Erkay Savaş
Sabancı University
Any synchronous sequential system could be described by an ASM chart. It does not make much sense to design the whole system using a single ASM chart. What makes the design process manageable is to partition the design in some way. The most straightforward way to partition a design is to split it in two:
- **datapath**: consists of components that store and manipulate data
- **control**: can be designed as a state machine that controls functioning of those datapath components.
Our Old Example

- Traffic light controller with a timer:

  - Assume that the clock frequency is such that the timed interval is completed in 256 clock cycles.
  - There is an abundance of states.
  - States from s0 to s255 map to a simple controller.
  - We can just separate the traffic light controller from the timer.
Linked ASM Charts

Hardware subroutine model
Alternative Linked ASM Chart

A conventional counter
Datapath/Controller Partitioning

• In our previous example, the ASM chart in the right hand side is a simple 8-bit counter.
• It would be more efficient to use an existing 8-bit controller instead of designing it using our ASM chart.
  - Existing counter may correspond to an existing hardware or a VHDL model.
• Then it is convenient to think of a sequential system in terms of
  - the **datapath** whose components that have been previously designed (or can be adapted)
  - the **controller** which is a design-specific state machine
Datapath/Controller Partitioning

Controller

Datapath

Control Signals

Status Signals

clock, reset

System Inputs

System Outputs
Register Transfer Operations

- The datapath would normally contain registers.
- The functionality of the system can be described in terms of register transfer operations.
- We can express register transfer operations in the extended ASM chart notation.

\[
Z \leftarrow 1
\]

- The notation means that \( Z \) takes the value 1 at the end of the state indicated, and holds that value until it is set to another value or reset.
If $Z$ is reset to 0 and it is only set to 1 in the present state, the registered output would be implemented as a flip-flop and a multiplexor.
A More Complex Example

\[
\begin{align*}
B_0 & \leftarrow X_0 \\
B_1 & \leftarrow X_1 \\
B_2 & \leftarrow X_2
\end{align*}
\]

\[
B_0 \leftarrow 0 \\
B_0 \leftarrow B_0 \oplus B_1 \\
B_1 \leftarrow B_1 \oplus B_2 \\
B_2 \leftarrow B_2 \cdot B_1 \cdot B_0 \\
\]

\[
B_0 \leftarrow B_1 \\
B_1 \leftarrow B_2 \\
B_2 \leftarrow 0
\]
Implementation of Controller

• Next state equations for the controller

\[ S_0^+ = S_0 \cdot S_1 \cdot (\overline{A + X_0}) \]

\[ S_1^+ = S_0 \cdot S_1 \cdot A \]
Implementation of Datapath

\[ X_2 \rightarrow S_{1S_0} \rightarrow 0 \rightarrow D \rightarrow B_2 \]

\[ X_1 \rightarrow S_{1S_0} \rightarrow 0 \rightarrow D \rightarrow B_1 \]

\[ X_0 \rightarrow S_{1S_0} \rightarrow 0 \rightarrow D \rightarrow B_0 \]

D Q

& \quad \oplus

\text{clock}
A Simple Microprocessor Design

- High level language expressions
- Assembler instructions
- Machine instructions.
  - 8, 16, 32-bit numbers
- Example: 8-bit processor (opcode three bits, operands are five bits)
  - \( a = b + c; \)
  translated into
  - LOAD b 00000001
  - ADD c 01000010
  - STORE a 00100011
A Simple Microprocessor Design

- **Example**
  - **opcode**
    - LOAD 000
    - ADD 010
    - STORE 001
  - **addresses**
    - a 00001
    - b 00010
    - c 00011

- **Datapath/controller partition**
- **controller** (a.k.a. sequencer) is a state machine.
- **Bits of the opcode are input to the controller**
  - the opcode may be decoded to generate a larger set of inputs to the controller (micro-coding approach)
A Simple Microprocessor Design

- In our instructions,
  - first three bits are the opcode
  - the remaining five bits are the address
  - **direct** addressing mode.
- Other addressing mode (**immediate**)  
  - `a = b + 5;`
  - `LOAD b`
  - `ADD #5`
  - `STORE a`
  - The opcodes for **direct** add and **immediate** add will be different.
  - Controller will go through different states when executing these two instructions
Designing the Simple Microprocessor

- Design approach
  - Partition the design into datapath and controller
- 8-bit microprocessor
  - datapath components and bus are all 8-bit wide.
- Features
  - A single bus
  - Each component uses a three-state buffers so that only one component is attempting to drive the bus.
  - Fully synchronous with a single clock
  - a synchronous reset
Block Diagram of the Processor

PC

IR

Controller

opcode

RAM

MDR

MAR

Flags

ACC

Control Signals
Inside the CPU

• Memory
  - Main memory is commonly implemented as asynchronous SRAM.
  - Synchronous registers act as buffers between the main memory and the CPU.
    • Memory Address Register (MAR)
    • Memory Data Register (MDR)

• Arithmetic Logic Unit
  - performs operation such as add, sub, and, or, etc.
  - combinational
  - One input is the ACC and the other is the bus.
Inside the CPU

• ALU (cont.)
  - It usually has other outputs called flags (for testing overflow, zero result, etc).
  - Flags act as inputs to the controller (status signals)

• ACC (accumulator)
  - a synchronous register
  - It always acts as one of the source and destination registers.

• PC (program counter)
  - Instructions in a program are held sequentially in memory
  - The PC register holds the address of the current instruction to be executed
• **PC (cont.)**
  - The content of PC is automatically updated.
  - It must also be possible to load a new address in case of branch and jump instructions.

• **IR (instruction register)**
  - The instruction to be executed is read from the memory
  - IR holds the current instruction during the execution of it.
  - bits of opcode (operation code) field of the instruction are input to the controller.
## Controller (Sequencer)

<table>
<thead>
<tr>
<th>Control signal</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC_bus</td>
<td>drive bus with contents of ACC (enable three-state output)</td>
</tr>
<tr>
<td>load_ACC</td>
<td>load ACC from bus</td>
</tr>
<tr>
<td>PC_bus</td>
<td>drive bus with contents of PC</td>
</tr>
<tr>
<td>load_IR</td>
<td>load IR from bus</td>
</tr>
<tr>
<td>load_MAR</td>
<td>load MAR from bus</td>
</tr>
<tr>
<td>MDR_bus</td>
<td>drive bus with contents of MDR</td>
</tr>
<tr>
<td>load_MDR</td>
<td>load MDR from bus</td>
</tr>
<tr>
<td>ALU_ACC</td>
<td>Load ACC with result from ALU</td>
</tr>
<tr>
<td>INC_PC</td>
<td>Increment PC and save the result in PC</td>
</tr>
<tr>
<td>Addr_bus</td>
<td>drive bus with operand part of instruction held in IR</td>
</tr>
<tr>
<td>CS</td>
<td>Chip Select. Use contents of MAR to set up memory address</td>
</tr>
<tr>
<td>R_NW</td>
<td>Read, Not Write. When false, contents of MDR are stored in memory</td>
</tr>
<tr>
<td>ALU_add</td>
<td>Perform an add operation in the ALU</td>
</tr>
<tr>
<td>ALU_sub</td>
<td>Perform a subtract operation in the ALU</td>
</tr>
</tbody>
</table>
Reminder: Block Diagram of the Processor

PC → IR → Controller

Controller → Control Signals

Controller → MDR → MDR

Controller → RAM

PC → ACC

ACC → Flags
ASM Chart for the Controller

s0
MAR ← PC
PC ← PC + 1

s1
CS
R_NW

s2
IR ← MDR

s3
MAR ← Addr

s4
MDR ← ACC

s5
CS

s6
CS
R_NW

s7
ACC ← MDR

s8

op=store
op=load
op=add

1
0

1
0

1
0

1
0

ACC ← MDR + ACC
ACC ← MDR - ACC

Fetch
Execute
ASM Chart with Branching

s6
CS
R_NW

1

op=load

0

op=bne

s7
ACC ← MDR

s8

0

ACC ← MDR + ACC

ACC ← MDR - ACC

s9
z_flag

1

PC ← MDR

0

1
Reminder: Block Diagram of the Processor

**PC** → **IR** → **Controller**

**opcode** → **Control Signals**

**ACC**

**Flags**

**MDR**

**RAM**
Methods to Synthesize Bus

- Recall that tri-state buffers are used for constructing busses.

- The following method can be used with Xilinx FPGA; but not with Actel FPGA since this technology does not support three-state logic.

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;

entity multiplexor is
    port(a, b, c, d, e: in std_ulogic;
         s: in std_ulogic_vector(4 downto 0);
         y: out std_ulogic);
end entity multiplexor;
```
Methods to Synthesize Bus

architecture version1 of multiplexor is
begin
  p0: process(s, a, b, c, d, e) is
  begin
    case s is
      when "00001" => y <= a;
      when "00010" => y <= b;
      when "00100" => y <= c;
      when "01000" => y <= d;
      when others => y <= e;
    end case;
  end process p0;
end architecture version1;

architecture version2 of multiplexor is
begin
  y <= a when s(0) = '1' else 'Z';
  y <= b when s(1) = '1' else 'Z';
  y <= c when s(2) = '1' else 'Z';
  y <= d when s(3) = '1' else 'Z';
  y <= e when s(4) = '1' else 'Z';
end architecture version2;
Synthesis Results with SimplifyPro

- Version 1 Architecture
Synthesis Results with SimplifyPro

- Version 2 Architecture

three-state buffers
Synthesis Results with Synopsys
library ieee;
use ieee.std_logic_1164.all;

package cpu_defs is

    type opcode is (load, store, add, sub, bne);
    constant word_w: natural := 8; -- # of bits for bus
    constant op_w: natural := 3; -- # of bits for opcode
    constant rfill: std_ulogic_vector(op_w-1 downto 0):=
        (others => '0'); -- padding for address

    -- two functions to do conversion between opcode bits of
    -- std_ulogic_vector and opcode type
    function slv2op (slv: in std_ulogic_vector) return opcode;
    function op2slv (op: in opcode) return std_ulogic_vector;

end package cpu_defs;

package body cpu_defs is ...

...  

package body cpu_defs is  

  type optable is array(opcode) of  
    std_ulogic_vector(op_w downto 0);  

  constant trans_table: optable  
    := ("000", "001", "010", "011", "100");  

  function op2slv (op: in opcode) return std_ulogic_vector is  
  begin  
    return trans_table(op);  
  end function op2slv;  

  function slv2op (slv: in std_ulogic_vector) return opcode is  
  begin  
    ...  
  end function slv2op;  

end package body cpu_defs;
package body cpu_defs is

function slv2op (slv: in std_ulogic_vector) return opcode is
    variable transop: opcode;
begin
    -- this is how it should be done, but some synthesis
    -- tools may not support it.
    for i in opcode loop
        if slv = trans_table(i) then
            transop := i;
        end if;
    end loop;
end function slv2op;
end package body cpu_defs;
package body cpu_defs is

function slv2op (slv: in std_ulogic_vector) return opcode is
    variable transop: opcode;
    begin
        -- this is a less elegant method;
        -- if the definitions of the opcode and/or trans_table are
        -- changed, then this code also has to be changed.
        -- There is therefore potential for inconsistency
        case slv is
            when "000" => transop := load;
            when "001" => transop := store;
            when "010" => transop := add;
            when "011" => transop := sub;
            when "100" => transop := bne;
            end case;
        return transop;
    end function slv2op;
end package body cpu_defs;
Controller: Entity Declaration

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use work.cpu_defs.all;
entity controller is
  port(clock, reset: in std_logic;
       op: in opcode;
       z_flag: in std_ulogic; -- status signal
       ACC_bus, load_ACC, PC_bus, load_PC,
       load_IR, load_MAR, MDR_bus, load_MDR,
       ALU_ACC, ALU_add, ALU_sub, INC_PC,
       Addr_bus, CS, R_NW: out std_ulogic);
end entity controller;

architecture rtl of controller is
  ...
end architecture rtl;
```
Controller: Sequential Part

architecture rtl of controller is
  type state is (s0, s1, s2, s3, s4, s5, s6, s7, s8, s9);
  signal present_state, next_state: state;
begin
  seq: process(clock, reset) is
  begin
    if reset = '1' then
      present_state <= s0;
    elsif rising_edge(clock) then
      present_state <= next_state;
    end if;
  end process seq;
  ...
end architecture rtl;
Controller: Combinational Part

... 

comm: process(present_state, op, z_flag) is 
begin
  -- reset all the control signals to a default
  ACC_bus <= '0';
  load_ACC <= '0';
  PC_bus <= '0';
  load_PC <= '0';
  load_IR <= '0';
  load_MAR <= '0';
  MDR_bus <= '0';
  load_MDR <= '0';
  ALU_ACC <= '0';
  ALU_add <= '0';
  ALU_sub <= '0';
  INC_PC <= '0';
  Addr_bus <= '0';
  CS <= '0';
  R_NW <= '0';
...
end process com;
end architecture rtl;
Controller: Combinational Part

... comm: process(present_state, op, z_flag) is begin ...

  case present_state is
    when s0 => -- current inst address to be loaded in
                -- MAR, PC incremented, etc.
      PC_bus <= '1';
      load_MAR <= '1';
      INC_PC <= '1';
      load_PC <= '1';
      next_state <= s1;
    ...
  end case;
end process com;
end architecture rtl;
Controller: Combinational Part

... com: process(present_state, op, z_flag) is
begin
... case present_state is
... when s1 => -- memory is read, MDR contains the inst.
CS <= '1';
R_NW <= '1';
next_state <= s2;
when s2 => -- inst. transferred from MDR to IR
MDR_bus <= '1';
load_IR <= '1';
next_state <= s3;
... end case;
end process com;
end architecture rtl;
Controller: Combinational Part

...  
comm: process (present_state, op, z_flag) is
begin
    ...
    case present_state is
    ...
        when s3 =>  -- direct address to be loaded in MAR
            Addr_bus <= '1';
            load_MAR <= '1';
            if op = store then
                next_state <= s4;
            else
                next_state <= s6;
            end if;
    ...
    end case;
end process com;
end architecture rtl;
Controller: Combinational Part

...  
comm: process(present_state, op, z_flag) is
begin

    ...  
    case present_state is
    ...
    when s4 => -- store (memory to be written)
        ACC_bus <= '1';
        load_MDR <= '1';
        next_state <= s5;
    when s5 => -- select memory to write it with MDR
        CS <= '1';
        next_state <= s0;
    ...
    end case;
end process com;
end architecture rtl;
Controller: Combinational Part

...

comm: process(present_state, op, z_flag) is
begin
...
case present_state is
...
when s6 =>
    CS <= '1'; R_NW <= '1'; -- read the memory just in case
    if op = load then next_state <= s7;
    else
        if op = bne then
            if z_flag = '1' then next_state <= s9; -- taken
            else next_state <= s0; -- not taken
        end if;
        else next_state <= s8; -- not a branch
    end if;
end if;
...
end case;
end process com;
end architecture rtl;
Controller: Combinational Part

... comm: process(present_state, op, z_flag) is
begin

... case present_state is

... when s7 => -- load instruction
  MDR_bus <= '1'; -- ACC gets content of MDR
  load_ACC <= '1';
  next_state <= s0;

... end case;
end process com;
end architecture rtl;
Controller: Combinational Part

... comm: process(present_state, op, z_flag) is begin ...

case present_state is ...

when s8 =>    -- ALU operation
    MDR_bus <= '1';    -- one operand from MDR
    ALU_ACC <= '1';    -- the other from ACC
    if op = add then
        ALU_add <= '1';
    elsif op = sub then
        ALU_sub <= '1';
    end if;
    next_state <= s0;
...
end case;
end process com;
end architecture rtl;
Controller: Combinational Part

...  
  comm: process(present_state, op, z_flag) is  
  begin  
    ...  
    case present_state is  
    ...  
      when s9 => -- completing the branch instruction  
        MDR_bus <= '1';  
        load_PC <= '1';  
        next_state <= s0;  
      end case;  
    end process com;  
  end process  
end architecture rtl;  

IEEE.NUMERIC_STD Package

-- Library : This package shall be compiled into a library symbolically
-- : named IEEE.
-- :
-- Developers : IEEE DASC Synthesis Working Group, PAR 1076.3
-- :
-- Purpose : This package defines numeric types and arithmetic functions
-- : for use with synthesis tools. Two numeric types are defined:
-- :
-- : --> UNSIGNED : represents unsigned number in vector form
-- :
-- : --> SIGNED : represents a signed number in vector form
-- :
-- : The base element type is type STD_LOGIC.
-- :
-- : The leftmost bit is treated as the most significant bit.
-- :
-- : Signed numbers are represented in two's complement form.
-- :
-- : This package contains overloaded arithmetic operators on
-- :
-- : the SIGNED and UNSIGNED types. The package also contains
-- :
-- : useful type conversions functions.
-- :
-- : If any argument to a function is a null array, a null array is
-- : returned (exceptions, if any, are noted individually).
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.cpu_defs.all;

entity ALU is
   port (clock, reset: in std_ulogic;
         ACC_bus, load_ACC, ALU_ACC, ALU_add, ALU_sub: in std_ulogic;
         system_bus: inout std_ulogic_vector(word_w-1 downto 0);
         z_flag: out std_ulogic);
end entity ALU;
architecture rtl of ALU is
...
end architecture rtl;
Datapath Design: ALU

architecture rtl of ALU is
  signal acc: unsigned(word_w-1 downto 0);
  constant zero: unsigned(word_w-1 downto 0) := (others => '0');
begin
  system_bus <= std_ulogic_vector(acc) when ACC_bus = '1'
                   else (others => 'Z');
  z_flag <= '1' when acc = zero else '0';
process(clock, reset) is
begin
  ...
  end process;
end architecture rtl;
architecture rtl of ALU is

... process (clock, reset) is
begin
  if reset = '1' then
    acc <= (others => '0');
  elsif rising_edge(clock) then
    if ALU_ACC = '1' then
      if ALU_add = '1' then
        acc <= acc + unsigned(system_bus);
      elsif ALU_sub = '1' then
        acc <= acc - unsigned(system_bus);
      end if;
    elsif load_ACC = '1' then
      acc <= unsigned(system_bus);
    end if;
  end if;
end process
end architecture rtl;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.cpu_defs.all;

entity PC is
  port (clock, reset: in std_ulogic;
        PC_bus, load_PC, INC_PC: in std_ulogic;
        system_bus: inout std_ulogic_vector(word_w-1 downto 0));
end entity PC;

architecture rtl of PC is
...
end architecture rtl;
Datapath Design: PC

architecture rtl of PC is
  signal count: unsigned (word_w – op_w – 1 downto 0);
begin
  system_bus <= refill & std_ulogic_vector(count)
    when PC_bus = '1' else (others => 'Z');
  process (clock, reset) is
  begin
    if reset = '1' then count <= (others => '0');
    elsif rising_edge(clock) then
      if load_PC = '1' then
        if INC_PC = '1' then
          count <= count + 1;
        else
          count <= unsigned(system_bus(word_w-op_w-1 downto 0));
        end if;
      end if;
    end if;
  end process;
end architecture rtl;
library ieee;
use ieee.std_logic_1164.all;
use work.cpu_defs.all;

entity IR is
  port (clock, reset: in std_ulogic;
        Addr_bus, load_IR: in std_ulogic;
        system_bus: inout std_ulogic_vector(word_w-1 downto 0);
        op: out opcode);
end entity IR;

architecture rtl of IR is
  ...
end architecture rtl;
architecture rtl of IR is

    signal instr_reg: std_ulogic_vector(word_w - 1 downto 0);

begin

    system_bus <= refill & instr_reg(word_w-op_w-1 downto 0)
        when Addr_bus = '1' else (others => 'Z');
    op <= slv2op(instr_reg(word_w - 1 downto word_w - op_w));

process (clock, reset)
begin
    if reset = '1' then
        instr_reg <= (others => '0');
    elsif rising_edge(clock) then
        if load_IR = '1' then
            instr_reg <= system_bus;
        end if;
    end if;
end process;

end architecture rtl;
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.cpu defs.all;
entity RAM is
  port (clock, reset: in std_ulogic;
        MDR_bus, load_MDR, load_MAR, CS, R_NW: in std_ulogic;
        system_bus: inout std_ulogic_vector(word_w-1 downto 0));
end entity RAM;
architecture rtl of RAM is
  ...
end architecture rtl;
architecture rtl of RAM is
  signal mdr: std_ulogic_vector(word_w-1 downto 0);
  signal mar: std_ulogic_vector(word_w - op_w -1 downto 0);
begin
  system_bus <= mdr when MDR_bus = '1' else (others => 'Z');
  process (clock, reset) is
    type mem_array is array(0 to 2**(word_w-op_w-1) of
      std_ulogic_vector(word_w-1 downto 0);
    variable mem: mem_array;
    ...
    end process;
end architecture rtl;
Datapath Design: RAM

...  

process (clock, reset) is  
  type mem_array is array (0 to 2**(word_w-op_w-1)) of  
    std_ulogic_vector (word_w-1 downto 0);  
  variable mem: mem_array;  
  constant prog: mem_array := (  
    0 => op2slv(load) &  
      std_ulogic_vector (to_unsigned (4, word_w - op_w)),  
    1 => op2slv(add) &  
      std_ulogic_vector (to_unsigned (5, word_w - op_w)),  
    2 => op2slv(store) &  
      std_ulogic_vector (to_unsigned (6, word_w-op_w)),  
    3 => op2slv(bne) &  
      std_ulogic_vector (to_unsigned (7, word_w-op_w)),  
    4 => std_ulogic_vector (to_unsigned (2, word_w)),  
    5 => std_ulogic_vector (to_unsigned (3, word_w)),  
    others => (others => ‘0’));  
...  
  end process;  
end architecture rtl;
Datapath Design: RAM

...  

process (clock, reset) is
begin
  if reset = '1' then
    mdr <= (others => '0');
    mar <= (others => '0');
    mem <= prog;
  elsif rising_edge(clock) then
    ...
  end if;

  end process;
end architecture rtl;
Datapath Design: RAM

process (clock, reset) is
begin
  ...
  elsif rising_edge(clock) then
    if load_MAR = '1' then
      mar <= unsigned(system_bus(word_w-op_w-1 downto 0));
    elsif load_MDR = '1' then
      mdr <= system_bus;
    elsif CS = '1' then
      if R_NW then mdr <= mem(to_integer(mar));
      else mem(to_integer(mar)) := mdr;
      end if;
    end if;
  end if;
  end if;
end process;
end architecture rtl;
• **Structural Modeling**
  - Components can be put together by instantiating them.

• **Issues**
  - system bus must be resolved at this level
  - it must be declared with the type `std_logic_vector`.
  - system bus is chosen to be treated as a port; it could have been a signal
  - Type conversion during instantiation
  - if the port is an input (mode `in`)  
    
    ```
    x => to_bit(y)  
    ```
  
    where `x` is of type `bit` while `y` is of type `std_ulogic`.
Putting Everything Together

- Type conversion during instantiation
  - if the port is an output (mode `out`)
    \[\text{to\_std\_ulogic}(x) \Rightarrow y\]
    simply means that \(x\) is converted to \text{std\_ulogic} before being assigned to \(y\).
  - if the port is bidirectional (mode `inout`)
    \[\text{to\_std\_ulogic}(x) \Rightarrow \text{to\_bit}(y)\]
library ieee;
use ieee.std_logic_1164.all;
use work.cpu_defs.all;

entity CPU is
  port (clock, reset: in std_ulogic;
        system_bus: inout std_logic_vector(word_w-1 downto 0));
end entity CPU;

architecture top of CPU is
...
end architecture CPU;
architecture top of CPU is

component controller is
port (clock, reset: in std_logic;
    op: in opcode;
    z_flag: in std_ulogic;
    ACC_bus, load_ACC, PC_bus, load_PC,
    load_IR, load_MAR, MDR_bus, load_MDR,
    ALU_ACC, ALU_add, ALU_sub, INC_PC,
    Addr_bus, CS, R_NW: out std_ulogic);
end component controller;

component IR is
port (clock, reset: in std_ulogic;
    Addr_bus, load_IR: in std_ulogic;
    system_bus: inout std_ulogic_vector(word_w-1 downto 0);
    op: out opcode);
end component IR;
...
end architecture CPU;
architecture top of CPU is

...  

component PC is
    port (clock, reset: in std_ulogic;
          PC_bus, load_PC, INC_PC: in std_ulogic;
          system_bus: inout
                    std_ulogic_vector(word_w-1 downto 0));
    end component PC;

component RAM is
    port (clock, reset: in std_ulogic;
          MDR_bus, load_MDR, load_MAR, CS, R_NW: in std_ulogic;
          system_bus: inout
                    std_ulogic_vector(word_w-1 downto 0));
    end component RAM;

    ...  

end architecture CPU;
architecture top of CPU is

    component ALU is
        port (clock, reset: in std_ulogic;
            ACC_bus, load_ACC, ALU_ACC, ALU_add, ALU_sub: in
            std_ulogic;
            system_bus: inout
                std_ulogic_vector(word_w-1 downto 0);
            z_flag: out std_ulogic);
    end component ALU;

    signal ACC_bus, load_ACC, PC_bus, load_PC,
        load_IR, load_MAR, MDR, bus, load_MDR,
        ALU_ACC, ALU_add, ALU_sub, INC_PC,
        Addr_bus, CS, R_NW: std_ulogic;

    signal z_flag: std_ulogic;
    signal op: opcode;

    begin
        ...
    end architecture CPU;
CPU: Architecture

architecture top of CPU is
  ...
begin
  CNTRL1: controller port map(
    clock => clock, reset => reset, op => op,
    z_flag => z_flag, ACC_bus => ACC_bus,
    load_ACC => load_ACC, PC_bus => PC_bus,
    load_PC => load_PC, load_IR => load_IR,
    load_MAR => load_MAR, MDR_bus => MDR_bus,
    load_MDR => load_MDR, ALU_ACC => ALU_ACC,
    ALU_add => ALU_add, ALU_sub => ALU_sub,
    INC_PC => INC_PC, Addr_bus => Addr_bus,
    CS => CS, R_NW => R_NW);
  ...
end architecture CPU;
CPU: Architecture

architecture top of CPU is
    ...
begin
    I1: IR port map(
        clock => clock, reset => reset, Addr_bus => Addr_bus,
        load_IR => load_IR, op => op,
        std_logic_vector(system_bus) =>
            std_ulogic_vector(system_bus));
    P1: PC port map(
        clock => clock, reset => reset, PC_bus => PC_bus,
        load_PC => load_PC, INC_PC => INC_PC,
        std_logic_vector(system_bus) =>
            std_ulogic_vector(system_bus));
    ...
end architecture CPU;
architecture top of CPU is
...
begin
A1: ALU port map(
    clock => clock, reset => reset, ACC_bus => ACC_bus,
    load_ACC => load_ACC, ALU_add => ALU_Add,
    ALU_sub => ALU_sub,
    std_logic_vector(system_bus) =>
        std_ulogic_vector(system_bus),
    z_flag => z_flag);
R1: RAM port map(
    clock => clock, reset => reset, MDR_bus => MDR_bus,
    load_MDR => load_MDR, load_MAR => load_MAR,
    CS => CS, R_NW => R_NW,
    std_logic_vector(system_bus) =>
        std_ulogic_vector(system_bus));
end architecture CPU;
library ieee;
use ieee.std_logic_1164.all;
use work.cpu_defs.all;

entity test_cpu is
end entity cpu;

architecture tb of testcpu is
  component CPU is
    port (clock, reset: in std_ulogic;
          system_bus: inout
                      std_logic_vector(word_w-1 downto 0));
  end component CPU;
  ...
end architecture tb;
architecture tb of testcpu is

    component CPU is
        port (clock, reset: in std_ulogic;
              system_bus: inout
                std_logic_vector(word_w-1 downto 0));
    end component CPU;

    signal clock, reset: std_ulogic := '0';
    signal system_bus: std_logic_vector(word_w – 1 downto 0);

begin
    C1: CPU port map (clock, reset, system_bus);
    reset <= '1' after 1 ns, '0' after 2 ns;
    clock <= not clock after 10 ns;
end architecture tb;
configuration mapping of testcpu is
  for tb
    for C1: CPU use entity work.cpu(top);
      for top
        for CNTRL1: Controller use entity work.controller(rtl);
          end for;
        end for;
      end for;
    end for;
  end for;
end configuration mapping;