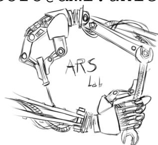
Implementing a System-on-Chip using VHDL

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it



S.D.R. Course

Our SoC is made of ...

- **A clock generator**, we use the built-in 50 MHz clock provided in the Altera DE0 board
- **A CPU**, we will use a 32-bit RISC architecture (not pipelined)
- **A ROM**, where we will place (statically) our code
- **A Parallel I/O Port**, 32-bit, connected to the 4-digit hex display and to switches and pushbuttons

CPU Architecture

Basic CPU Architecture

- A **32-bits** MIPS-like processor:
  - DATA BUS and ADDRESS BUS are 32-bits wide
  - Memory is organised in *words* 32-bits wide
- Also CPU instructions are 32-bits wide
- Registers:
  - 32 general purpose registers R0-R31
  - R0 is read-only and contains always “0” (like the MIPS processor)
  - Program Counter (PC), contains the memory address of the next instruction
  - Instruction Register (IR), contains the current instruction to be executed

Instruction Set

Basic opcode structure

Arithmetic/Logic Register-type opcodes

- **"010001"**, MOV Rs $\rightarrow$ Rt (Rd is not used)
- **"010010"**, ADD Rs + Rt $\rightarrow$ Rd
- **"010011"**, SUB Rs - Rt $\rightarrow$ Rd

```
MOV R1, R2        010001 00001 00010 00000 00000000000
ADD R1, R4, R2    010010 00001 00100 00010 00000000000
SUB R0, R7, R3    010011 00000 00111 00011 00000000000
```

Immediate-type opcodes

- **"001000"**, ADD_I Rs + Immediate → Rt
- **"001001"**, SUB_I Rs - Immediate → Rt
- **"000100"**, BEQ, Jump to "Immediate" address if Rs = Rt
- **"000101"**, BNE, Jump to "Immediate" address if Rs not = Rt
- **"100011"**, LW, load word from location Rs + "Immediate" and store it into Rt
- **"100011"**, SW, store word from Rt into location Rs + "Immediate"

```
ADD_I R1, #5, R2     001000 00001 00010 0000000000000101
SUB_I R0, #12, R7    001001 00000 00111 0000000000001100
```

Other instructions

Opcode(6 bits)	parameter(26 bits)

- **”000000”**, NOP
- **”010010”**, JUMP, absolute jump to “parameter” address
- **”111111”**, HALT

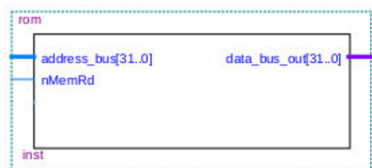
Part II

Hardware Components

CPU

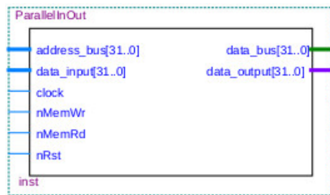
- **clock**, input, the main CPU clock
- **nRst**, input, hardware RESET, active low
- **address_bus**, 32-bit, output
- **data_bus**, 32-bit, bidirectional
- **control_bus**:
  - **nMemWr**, output, active low when a memory write is executed
  - **nMemRd**, output, active low when a memory read is executed
  - **nHalt**, output, active low when the CPU is in the HALT state

ROM



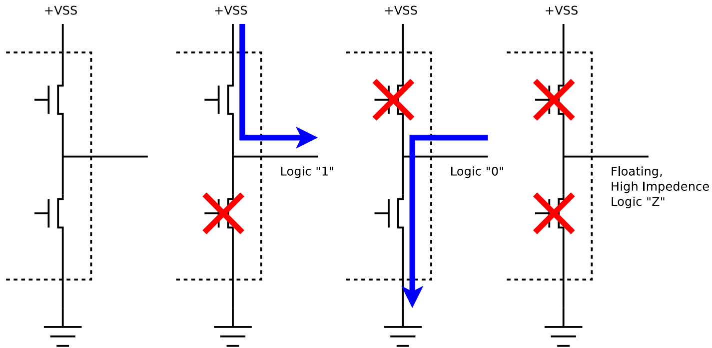
- **address_bus**, 32-bit, input, address to be read
- **data_bus**, 32-bit, output, data read
- **nMemRd**, input, active low when a memory read is executed
- **The ROM is designed to contain the software to be executed which is hard-coded in the VHDL file.**

Digital Output Port

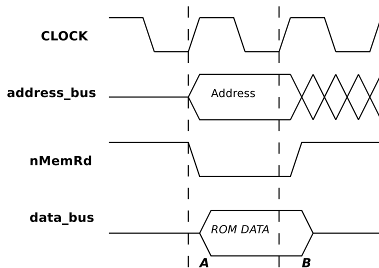


- **clock**, input, the main clock
- **nRst**, input, RESET, active low
- **address_bus**, 32-bit, input, address to be written
- **data_bus**, 32-bit, bidirectional, data bus
- **nMemWr**, input, active low when a memory write is executed
- **nMemRd**, input, active low when a memory read is executed
- **data_output**, 32-bit, output, the data that has been written to the port
- **data_input**, 32-bit, input, the data that has been read from the port
- **The module reacts to memory address 0x8000**
- **When a “store” instruction to address 0x8000 is executed, the written data appears on data_output pins.**
- **When a “load” instruction from address 0x8000 is executed, the data present on data_input pins is read.**

Tri-state Outputs



ROM/CPU Timings



- The CPU is synchronised on **rising edge** of the clock
  - The address to be read is set-up (by the CPU) on the **address bus** and the **nMemRd** signal is asserted (event **"A"**)
  - The ROM recognises the nMemRd signal and outputs, on the **data bus**, the word addressed
  - At the next rising edge, the CPU reads word from the data bus, the **nMemRd** signal is de-asserted (event **"B"**)
  - The ROM recognises that nMemRd is no more active and puts the **data bus** to high impedence

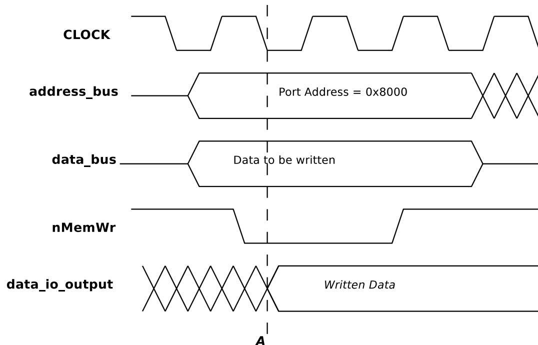
The ROM in VHDL

```
library ieee;
...

entity rom is
  port( data_bus_out: out std_logic_vector(31 downto 0);
        address_bus:  in std_logic_vector(31 downto 0);
        nMemRd: in std_logic);
end rom;

architecture rom_arch of rom is
  signal out_byte: std_logic_vector(31 downto 0);
begin
  process(nMemRd)
  begin
      if nMemRd = '0' then
        case address_bus is
          when "00000000000000000000000000000000" => out_byte <= NOP;
          when "00000000000000000000000000000001" => out_byte <= ADD_I & R0 & R1 & ...;
...
          when others =>  out_byte <= (others => 'Z');
        end case;
      else
        out_byte <= (others => 'Z');
      end if;
  end process;
  data_bus_out <= out_byte;
end architecture;
```

Parallel I/O Timings



- Parallel Output Port synchronises on **falling edge** of the clock
  - First the CPU sets-up the address of the port (0x8000), on the **address bus**, and the data to be written, on the **data bus**
  - Then the **nMemWr** signal is asserted (by the CPU)
  - At the next falling edge clock, the Port recognises the nMemWr signal and copies data from the **data bus** to the **output port** (event **"A"**)
  - After some clock cycles the CPU de-asserts **nMemWr** signal

The Parallel Output Port in VHDL

```
library ieee;
...
entity ParallelInOut is
port(address_bus: in std_logic_vector(31 downto 0);
     data_bus:  inout std_logic_vector(31 downto 0);
     data_output : out std_logic_vector(31 downto 0);
     data_input : in std_logic_vector(31 downto 0);
     clock: in std_logic;
     nMemWr:  in std_logic;
     nMemRd:  in std_logic;
     nRst: in std_logic );
     end ParallelInOut;

architecture pInOut of ParallelInOut is
begin
  process(clock,nRst)
  begin
    if nRst = '0' then
      data_output  <= (others => '1');
      data_bus <= (others => 'Z');
    elsif (clock'event and clock = '0') then
      if address_bus = x"00008000" then -- address is 0x8000
        if nMemWr = '0' then data_output  <= data_bus;
        elsif nMemRd = '0' then data_bus <= data_input;
        else data_bus <= (others => 'Z');
        end if;
      end if;
    end if;
  end process;
end architecture;
```

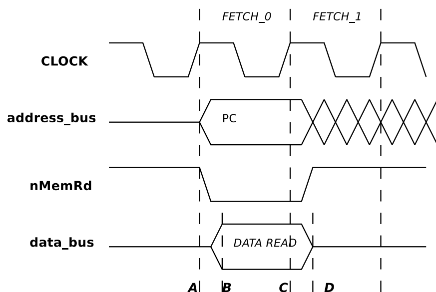
Part III

The CPU in VHDL

The CPU

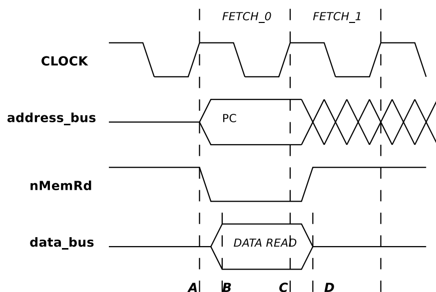
- It is implemented as a **finite-state machine** triggered by the clock signal
- While all peripherals react to falling edge of the clock, the CPU reacts to **rising edge**
- This is required to meet harware reaction times:
  - At the rising edge the CPU prepares the signals on address/data/control bus to interact with a peripheral/memory
  - At the (next) falling edge the peripheral/memory executes the action on the basis of such prepared signals
- **States:**
  - RESET: entered when nRst is '0'
  - FETCH_0, FETCH_1: fetch phase, executed in two clock cycles
  - EXEC: execute phase, with several sub-states on the basis of the instruction of be executed
  - HALT: halt phase, the CPU is stopped

The Fetch Phase



- FETCH_0: The CPU (event "A")
  - outputs the Program Counter to the Address Bus
  - puts the Data Bus to high impedence
  - asserts the nMemRd signal
  - sets the next state to FETCH_1
- The ROM (event "B")
  - recognises the nMemRd signal
  - outputs the word data to the data bus

The Fetch Phase



- FETCH_1: The CPU (event "C")
  - reads the word from the Data Bus and stores it into the Instruction Register
  - de-asserts the nMemRd signal
  - sets the next state to EXEC
- The ROM (event "D")
  - recognises de-activation of the nMemRd signal
  - puts the Data Bus in the high impedence state

The CPU in VHDL (I)

```
library ieee;
...
entity CPU is
port(clock: in std_logic;
     nRst: in std_logic;
     data_bus: inout std_logic_vector(31 downto 0);
     address_bus: out std_logic_vector(31 downto 0);
     nMemRd: out std_logic;
     nMemWr: out std_logic;
     nHalt: out std_logic);
end CPU;

architecture my_CPU of CPU is
  type state_type is (st_reset, st_fetch_0, st_fetch_1, st_exec, st_halt);
  type sub_state_type is (exec_0, exec_1);
  type register_array is array(0 to 7) of std_logic_vector(31 downto 0);
  signal PC: std_logic_vector(31 downto 0) := (others => '0');
  signal IR: std_logic_vector(31 downto 0) := (others => '0');
  signal REGS: register_array;
  signal state: state_type := st_halt;
  signal sub_state: sub_state_type := exec_0;
begin
  process(clock,nRst)
...
```

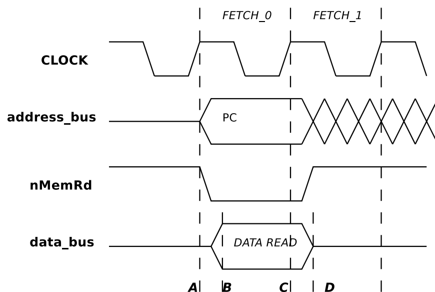
The CPU in VHDL (II)

```
...
  process(clock,nRst)
  variable op_code : std_logic_vector(5 downto 0);
  variable rd, rs, rt : integer;
  variable func : std_logic_vector(5 downto 0);
  variable immediate_val : std_logic_vector(15 downto 0);
  variable NextPC : std_logic_vector(31 downto 0);
  begin
    if (nRst = '0') then
      state <= st_reset;
    elsif (clock'event and clock = '1') then -- rising edge
      NextPC := PC + 1;
      case state is
        when st_reset => ... ;
        when st_fetch_0 => ...;
        when st_fetch_1 => ...;
        when st_exec => ...;
        when st_halt => ...;
      end case;
      if (state = st_exec and sub_state = exec_0) then
        PC <= NextPC;
      end if;
    end if;
  end process;
end architecture;
```

The CPU in VHDL (III)

```
...
case state is
  when st_reset =>
    PC <= (others => '0');
    REGS(0) <= (others => '0');
    REGS(1) <= (others => '0');
    REGS(2) <= (others => '0');
    REGS(3) <= (others => '0');
    REGS(4) <= (others => '0');
    REGS(5) <= (others => '0');
    REGS(6) <= (others => '0');
    REGS(7) <= (others => '0');
    nMemRd <= '1';
    nMemWr <= '1';
    data_bus <= (others => 'Z');
    nHalt <= '1';
    state <= st_fetch_0;
...
```

The CPU in VHDL (IV)




```
...
case state is
  ...
  when st_fetch_0 =>
    address_bus <= PC;
    nMemRd <= '0';
    state <= st_fetch_1;

  when st_fetch_1 =>
    IR <= data_bus;
    nMemRd <= '1';
    sub_state <= exec_0;
    state <= st_exec;
...
```

The CPU in VHDL (V)

```
...
case state is
  when st_exec =>
    if IR = NOP then
      state <= st_fetch_0;
    else
      op_code := IR(31 downto 26);
      rs := conv_integer(IR(25 downto 21));
      rt := conv_integer(IR(20 downto 16));
      rd := conv_integer(IR(15 downto 11));
      immediate_val := IR(15 downto 0);
      func := IR(5 downto 0);
      case op_code is
        when MOV => ...;
        when ADD => ...;
        ...
      end case;
    end if;
...
```

Basic structure: | Opcode(6 bits) | Additional parameters(26 bits) |

Arithmetic/Logic Register-type: | Opcode(6) | Rs(5) | Rt(5) | Rd(5) | p(11) |

Immediate-type: | Opcode(6) | Rs(5) | Rt(5) | Immediate(16) |

The CPU in VHDL (VI)

```
...
case op_code is
  when MOV =>
    if rt /= 0 then -- do not write into R0
      REGS(rt) <= REGS(rs);   -- move from rs to rt
    end if;
    state <= st_fetch_0;
  when ADD =>
    if rd /= 0 then -- do not write into R0
      REGS(rd) <= REGS(rs) + REGS(rt); -- add rs + rt into rd
    end if;
    state <= st_fetch_0;
  when SUBT =>
    if rd /= 0 then -- do not write into R0
      REGS(rd) <= REGS(rs) - REGS(rt); -- sub rs - rt into rd
    end if;
    state <= st_fetch_0;
  ...
end case;
...
```

Arithmetic/Logic Register-type:

Opcode(6)	Rs(5)	Rt(5)	Rd(5)	p(11)

- **"010001"**, MOV Rs → Rt (Rd is not used)
- **"010010"**, ADD Rs + Rt → Rd
- **"010011"**, SUB Rs - Rt → Rd

The CPU in VHDL (VII)

```
...
op_code := IR(31 downto 26);
rs := conv_integer(IR(25 downto 21));
rt := conv_integer(IR(20 downto 16));
rd := conv_integer(IR(15 downto 11));
immediate_val := IR(15 downto 0);
func := IR(5 downto 0);
case op_code is
  ...
  when ADD_I =>  -- Add Immediate
    if rt /= 0 then
      REGS(rt) <= REGS(rs) + immediate_val;
    end if;
    state <= st_fetch_0;
  when JUMP =>  -- jump
    NextPC := "000000" & IR(25 downto 0);
    state <= st_fetch_0;
  ...
end case;
...
```

Basic structure: | Opcode(6 bits) | Additional parameters(26 bits) |

Immediate-type: | Opcode(6) | Rs(5) | Rt(5) | Immediate(16) |

- **"001000"**, ADD_I Rs + Immediate → Rt
- **"010010"**, JUMP, absolute jump to "parameter" address

The CPU in VHDL (VIII)

```
...
case op_code is
  ...
  when SW =>   -- store word
    case sub_state is

      when exec_0 => address_bus <= immediate_val + REGS(rs);
                     data_bus <= REGS(rt);
                     nMemWr <= '0';
                     state <= st_exec;
                     sub_state <= exec_1;

      when exec_1 => address_bus <= (others => 'Z');
                     data_bus <= (others => 'Z');
                     nMemWr <= '1';
                     state <= st_fetch_0;

    end case;
...
```

Immediate-type: | Opcode(6) | Rs(5) | Rt(5) | Immediate(16) |

- **"100011"**, SW, store word from Rt into location Rs + "Immediate"

A simple "counter" program

```
0000 0000: NOP
0000 0001: ADD_I R0, R1, #1      ; R1 <- R0 + 1, eqv to R1 <- 1
0000 0002: SW R0, R1, #0x8000    ; MEM(R0 + 0x8000) <- R1, eqv to MEM(0x8000) <- R1
0000 0003: ADD_I R1, R1, #1      ; R1 <- R1 + 1
0000 0004: JUMP #2               ; go to address #2
```

```
architecture rom_arch of rom is
signal out_byte: std_logic_vector(31 downto 0);
begin
  process(clock)
  begin
    if (clock'event and clock = '0') then -- falling edge
      if nMemRd = '0' then
        case address_bus is
          when "..0000" => out_byte <= NOP;
          when "..0001" => out_byte <= ADD_I & R0 & R1 & "0000000000000001";
          when "..0010" => out_byte <= SW & R0 & R1 & "1000000000000000";
          when "..0011" => out_byte <= ADD_I & R1 & R1 & "0000000000000001";
          when "..0100" => out_byte <= JUMP & "00000000000000000000000010";
          when others => out_byte <= (others => 'Z');
        end case;
      else
        out_byte <= (others => 'Z');
      end if;
    end if;
  end process;
  data_bus_out <= out_byte;
end architecture;
```

Reading Input and display data

```
0000 0000: LW R0, R1, #0x8000   ; R1 <- MEM(R0 + 0x8000), eqv to R1 <- MEM(0x8000)
0000 0001: SW R0, R1, #0x8000   ; MEM(R0 + 0x8000) <- R1, eqv to MEM(0x8000) <- R1
0000 0004: JUMP #0              ; go to address #0
```

```
architecture rom_arch of rom is
signal out_byte: std_logic_vector(31 downto 0);
begin
  process(clock)
  begin
    if (clock'event and clock = '0') then -- falling edge
      if nMemRd = '0' then
        case address_bus is
          when "..00" => out_byte <= LW & R0 & R1 & "1000000000000000";
          when "..01" => out_byte <= SW & R0 & R1 & "1000000000000000";
          when "..10" => out_byte <= JUMP & "00000000000000000000000000";
          when others => out_byte <= (others => 'Z');
        end case;
      else
        out_byte <= (others => 'Z');
      end if;
    end if;
  end process;
  data_bus_out <= out_byte;
end architecture;
```

An “up/down counter” program

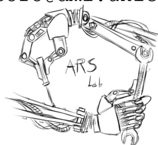
```
0000 0000: NOP
0000 0001: ADD_I R0, R1, #1      ; R1 <- R0 + 1, eqv to R1 <- 1
0000 0002: SW R0, R1, #0x8000    ; MEM(R0 + 0x8000) <- R1, eqv to MEM(0x8000) <- R1
0000 0003: LW R0, R2, #0x8000    ; R2 <- MEM(R0 + 0x8000), eqv to R2 <- MEM(0x8000)
0000 0004: AND_I R2, R2, #1      ; R2 <- R2 and 1
0000 0005: BEQ R0, R2, #8        ; if R2 = 0 go to #8
0000 0006: ADD_I R1, R1, #1      ; R1 <- R1 + 1
0000 0007: JUMP #2               ; go to address #2
0000 0008: SUB_I R1, R1, #1      ; R1 <- R1 - 1
0000 0009: JUMP #2               ; go to address #2
```

```vhdl
architecture rom_arch of rom is
signal out_byte: std_logic_vector(31 downto 0);
begin
  process(clock)
  begin
    if (clock'event and clock = '0') then -- falling edge
      if nMemRd = '0' then
        case address_bus is
          when "..0000" => out_byte <= NOP;
          when "..0001" => out_byte <= ADD_I & R0 & R1 & "0000000000000001";
          when "..0010" => out_byte <= SW & R0 & R1 & "1000000000000000";
          when "..0011" => out_byte <= LW & R0 & R2 & "1000000000000000";
          when "..0100" => out_byte <= AND_I & R2 & R2 & "0000000000000001";
          when "..0101" => out_byte <= BEQ & R0 & R2 & "0000000000001000";
          when "..0110" => out_byte <= ADD_I & R1 & R1 & "0000000000000001";
          when "..0111" => out_byte <= JUMP & "00000000000000000000000010";
          when "..1000" => out_byte <= SUB_I & R1 & R1 & "0000000000000001";
          when "..1001" => out_byte <= JUMP & "00000000000000000000000010";
          when others => (others => 'Z');
        end case;
```

Implementing a System-on-Chip using VHDL

Corrado Santoro

ARSLAB - Autonomous and Robotic Systems Laboratory
Dipartimento di Matematica e Informatica - Università di Catania, Italy
santoro@dmi.unict.it



S.D.R. Course