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
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
Basic opcode structure

<table>
<thead>
<tr>
<th>Opcode (6 bits)</th>
<th>Additional parameters (26 bits)</th>
</tr>
</thead>
</table>

Arithmetic/Logic Register-type opcodes

<table>
<thead>
<tr>
<th>Opcode (6)</th>
<th>Rs (5)</th>
<th>Rt (5)</th>
<th>Rd (5)</th>
<th>p (11)</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Opcode(6)</th>
<th>Rs(5)</th>
<th>Rt(5)</th>
<th>Immediate(16)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;001000&quot;</td>
<td></td>
<td></td>
<td>ADD Rs + Immediate → Rt</td>
</tr>
<tr>
<td>&quot;001001&quot;</td>
<td></td>
<td></td>
<td>SUB Rs - Immediate → Rt</td>
</tr>
<tr>
<td>&quot;000100&quot;</td>
<td></td>
<td></td>
<td>BEQ, Jump to “Immediate” address if Rs = Rt</td>
</tr>
<tr>
<td>&quot;000101&quot;</td>
<td></td>
<td></td>
<td>BNE, Jump to “Immediate” address if Rs not = Rt</td>
</tr>
<tr>
<td>&quot;100011&quot;</td>
<td></td>
<td></td>
<td>LW, load word from location Rs + “Immediate” and store it into Rt</td>
</tr>
<tr>
<td>&quot;100011&quot;</td>
<td></td>
<td></td>
<td>SW, store word from Rt into location Rs + “Immediate”</td>
</tr>
</tbody>
</table>

**Example Instructions:**

- **ADD** I R1, #5, R2  
  001000 00001 00010 0000000000000101

- **SUB** I R0, #12, R7  
  001001 00000 00111 0000000000001100
### Instruction Set

#### Other instructions

<table>
<thead>
<tr>
<th>Opcode (6 bits)</th>
<th>Parameter (26 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;000000&quot;</td>
<td>NOP</td>
</tr>
<tr>
<td>&quot;010010&quot;</td>
<td>JUMP, absolute jump to “parameter” address</td>
</tr>
<tr>
<td>&quot;111111&quot;</td>
<td>HALT</td>
</tr>
</tbody>
</table>
Part II

Hardware Components
- **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
- **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.

Corrado Santoro  
A VHDL SoC
Tri-state Outputs

- Logic "1"
- Logic "0"
- Floating, High Impedence Logic "Z"
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 impedance
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 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
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;
MemWr: in std_logic;
MemRd: in std_logic;
Rst: 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 process;
end architecture;
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 hardware 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 impedance
  - 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 impedance 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;
```
... 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;
...

Corrado Santoro  A VHDL SoC
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;
    ...
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;
    end if;
end case;
... 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:

<table>
<thead>
<tr>
<th>Opcode(6)</th>
<th>Rs(5)</th>
<th>Rt(5)</th>
<th>Rd(5)</th>
<th>p(11)</th>
</tr>
</thead>
</table>

- **"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;
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 & "100000000000000000000000000000";
  when "..01" => out_byte <= SW & R0 & R1 & "100000000000000000000000000000";
  when "..10" => out_byte <= JUMP & "00000000000000000000000000000000";
  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

<table>
<thead>
<tr>
<th>Address</th>
<th>Opcode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000 0000</td>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>0000 0001</td>
<td>ADD_I R0, R1, #1</td>
<td>R1 &lt;- R0 + 1, equiv to R1 &lt;- 1</td>
</tr>
<tr>
<td>0000 0002</td>
<td>SW R0, R1, #0x8000</td>
<td>MEM(R0 + 0x8000) &lt;- R1, equiv to MEM(0x8000) &lt;- R1</td>
</tr>
<tr>
<td>0000 0003</td>
<td>LW R0, R2, #0x8000</td>
<td>R2 &lt;- MEM(R0 + 0x8000), equiv to R2 &lt;- MEM(0x8000)</td>
</tr>
<tr>
<td>0000 0004</td>
<td>AND_I R2, R2, #1</td>
<td>R2 &lt;- R2 and 1</td>
</tr>
<tr>
<td>0000 0005</td>
<td>BEQ R0, R2, #8</td>
<td>if R2 = 0 go to #8</td>
</tr>
<tr>
<td>0000 0006</td>
<td>ADD_I R1, R1, #1</td>
<td>R1 &lt;- R1 + 1</td>
</tr>
<tr>
<td>0000 0007</td>
<td>JUMP #2</td>
<td>go to address #2</td>
</tr>
<tr>
<td>0000 0008</td>
<td>SUB_I R1, R1, #1</td>
<td>R1 &lt;- R1 - 1</td>
</tr>
<tr>
<td>0000 0009</td>
<td>JUMP #2</td>
<td>go to address #2</td>
</tr>
</tbody>
</table>

---

```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 & "000000000000000100";
        when "..0110" => out_byte <= ADD_I & R1 & R1 & "00000000000000001";
        when "..0111" => out_byte <= JUMP & "00000000000000000000000010";
        when "..1000" => out_byte <= SUB_I & R1 & R1 & "000000000000000001";
        when "..1001" => out_byte <= JUMP & "0000000000000000000000000010";
        when others => (others => 'Z');
        end case;
      end if
      end if
  end process
  data_bus_out <= out_byte;
end architecture
```
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