Digital Systems - Mini AVR 2

Pere Palà - Alexis López

iTIC http://itic.cat

April 2016

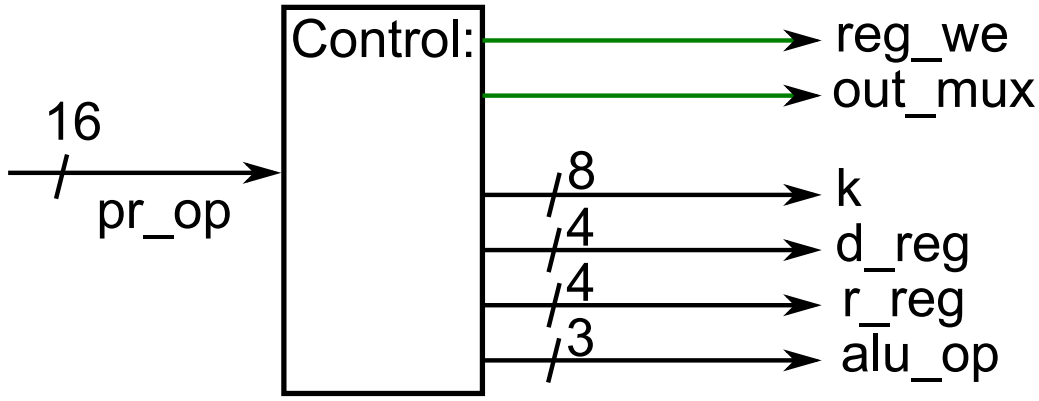
VHDL Implementation of the Status Register

```
type status_reg is
    record
      Z : std_logic;
      C : std_logic;
    end record;
```

```
signal pr_SR,
       nx_SR    : status_reg;
```

Control Unit

- Input: Present opcode
- Decode the instruction
- Generate signals:
  - Register write enable `reg_we`
  - Output multiplexer selection: decide if data written to register comes from:
    - The present opcode (such as LDI)
    - The ALU (such as ADC)
  - Generate d and r register addresses `d_reg`, `s_reg`
  - Indicate the desired operation to the ALU `alu_op`



VHDL Implementation of the Control Unit (1/2)

```
type    out_mux_type is (mux_alu,mux_lit);
signal  out_mux   : out_mux_type;
```

```
CONTROL : process(pr_op,pr_pc)
begin
    r_reg    <= (others => '-');     --Defaults
    d_reg    <= (others => '-');
    k        <= (others => '-');
    ALU_op   <= (others => '-');
    reg_we   <= '0';
    case pr_op(15 downto 12) is -- These bits are enough to
                                -- decide among our reduced
                                -- instruction set!
    when NOP => ------------------------NOP Instruction
      null;

    when LDI => ------------------------LDI Instruction
      d_reg    <= pr_op(7 downto 4);
      out_mux  <= mux_lit;
      k        <= pr_op(11 downto 8) & pr_op(3 downto 0);
      reg_we   <= '1';
```

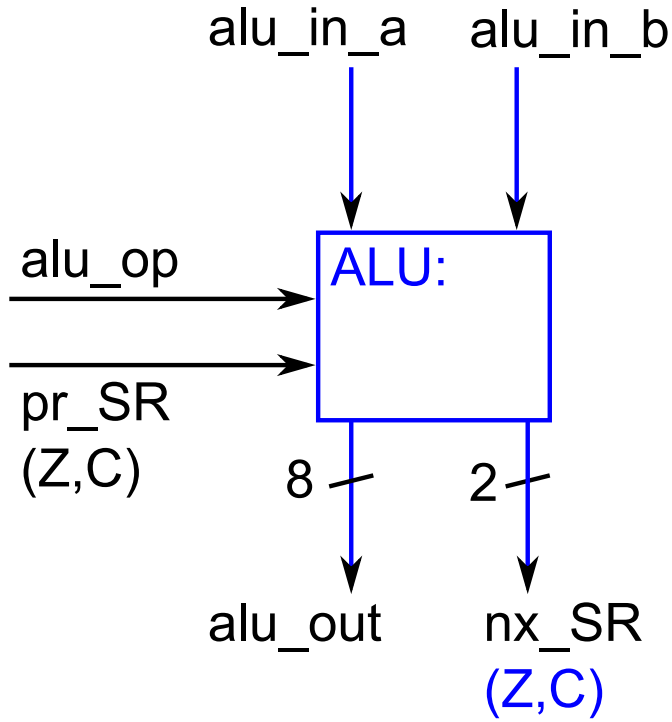
VHDL implementation of the Control Unit (2/2)

```
    when MOV => ---------------------------MOV Instruction
      r_reg   <= pr_op(3 downto 0);   --Source register
      d_reg   <= pr_op(7 downto 4);   --Destination register
      out_mux <= mux_alu;
      reg_we  <= '1';                 --Enable register write
      ALU_op  <= ALU_MOV;

    when ADC => ---------------------------ADC Instruction
      r_reg   <= pr_op(3 downto 0);
      d_reg   <= pr_op(7 downto 4);
      out_mux <= mux_alu;
      reg_we  <= '1';                 --Enable register write
      ALU_op  <= ALU_ADC;

    when others =>
      null;
    end case;
end process;
```

```
constant ALU_MOV : std_logic_vector(2 downto 0) := "010";
constant ALU_ADC : std_logic_vector(2 downto 0) := "001";
```

The ALU: Arithmetic and Logic Unit

- Inputs
  - Operands `alu_in_a`, `alu_in_b`
  - Operation code `alu_op`
  - Status register (for carry) `pr_SR`
- Outputs
  - Operation result `alu_out`
  - Updated status register `nx_SR`

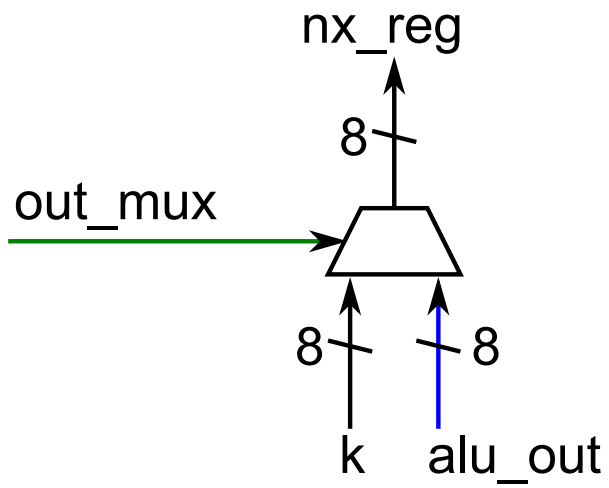


VHDL implementation of the ALU

```
ALU : process(alu_op,alu_in_a,alu_in_b,pr_SR,add_temp)
begin
  nx_SR <= pr_SR;     --by default, preserve status register
  add_temp <= (others => '-');
  case alu_op is
  when ALU_MOV  => ----------------------MOV: in_b --> out
    alu_out <= alu_in_b;
  when ALU_ADC  => ----------------------ADC: Carry in/out
    add_temp <= std_logic_vector(      --auxiliar SLV(9..0)
        unsigned('0' & alu_in_a & '1') +
        unsigned('0' & alu_in_b & pr_SR.C));  --Carry In
    alu_out  <= add_temp(8 downto 1);
    nx_SR.C  <= add_temp(9);                --Update Carry Flag
    if add_temp(8 downto 1) = x"00" then --Update Zero Flag
       nx_SR.Z <= '1';
    else
       nx_SR.Z <= '0';
    end if;
  when others => ---------------------------Should not happen
    alu_out <= (others => '-');      --Don't care
  end case;
end process;
```

Register Data-In Multiplexer

- Decide which data has to be stored in one of the registers
  - Data from the ALU, as in ADC or MOV instructions
  - Data from the opcode, as in the LDI instruction

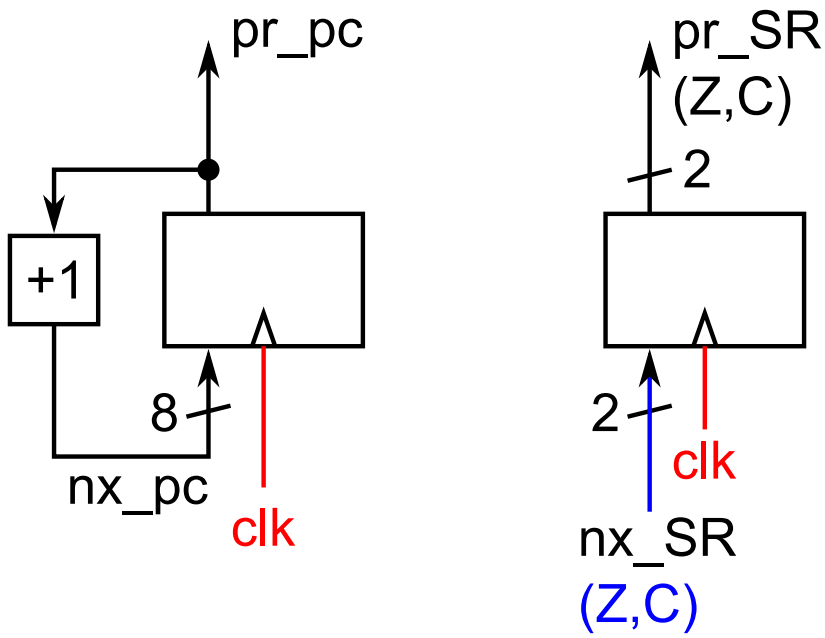


VHDL implementation of the Register Multiplexer

```
MUX : process(out_mux, alu_out, k)
begin
  case out_mux is
  when mux_alu  => nx_reg <= alu_out;
  when mux_lit  => nx_reg <= k;
  when others   => nx_reg <= (others => '-');
  end case;
end process;
```

Synchronous Elements

- Registered signals
  - Program counter `pr_pc`
  - Stauts register `pr_SR`



VHDL implementation of the Synchronous Elements

```
process(clk,reset) --Synchronous elements
begin
  if reset='1' then              --Initialize Processor
     pr_pc <= (others => '0');
     pr_SR.C <='0';
     pr_SR.Z <='0';
  elsif (rising_edge(clk)) then
     pr_pc      <= nx_pc;
     pr_SR      <= nx_SR;
  end if;
end process;

nx_pc    <= std_logic_vector(unsigned(pr_pc) + 1);
```

Whole Processor

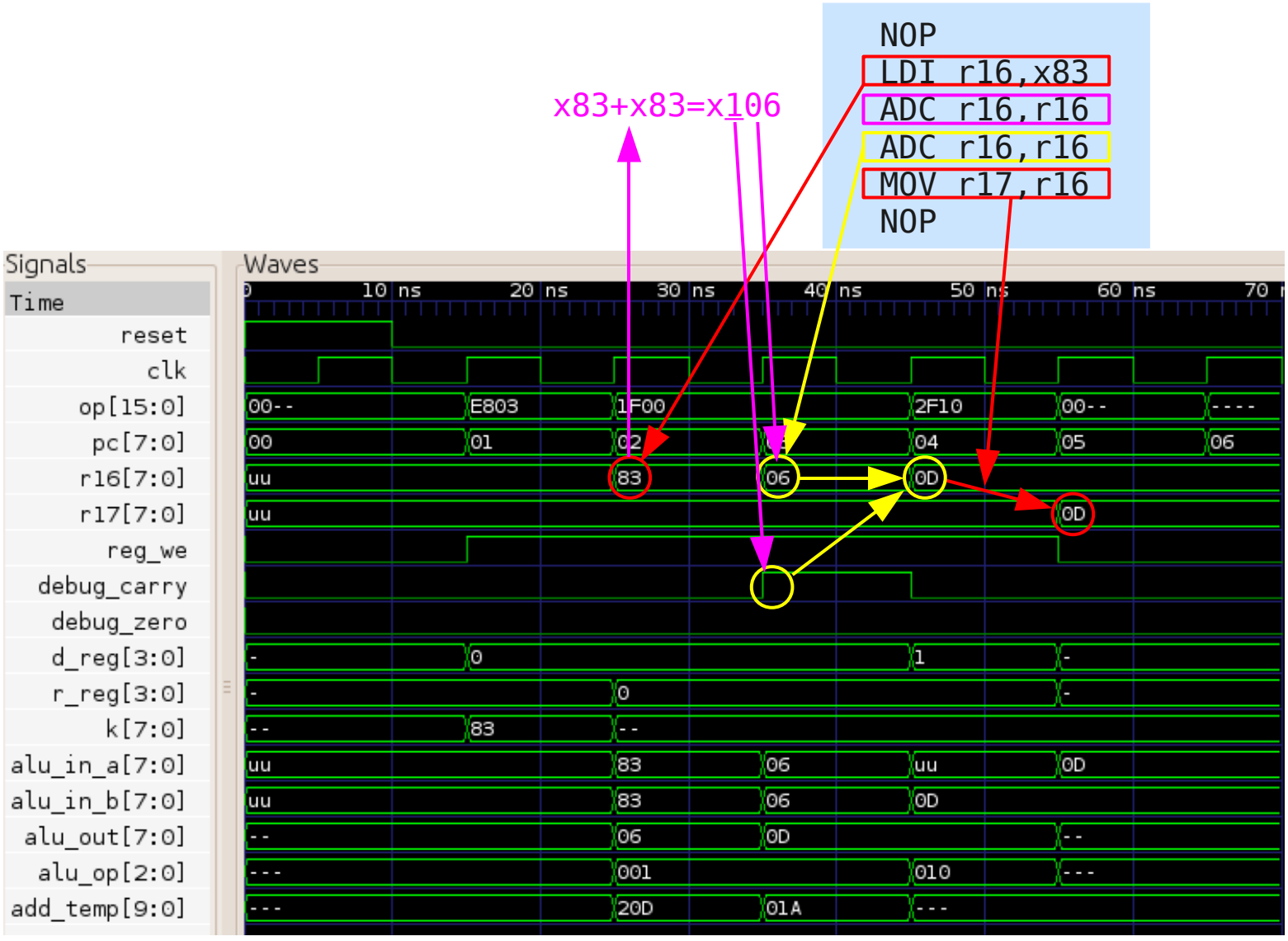
- Entity

```
entity mini_avr_01 is
   port (
      clk   : in  std_logic;
      reset : in  std_logic;
      -- Let's have some registers as outputs :
      r16   : out std_logic_vector( 7 downto 0);
      r17   : out std_logic_vector( 7 downto 0)
   );
end entity;
```

- Auxiliar signals for debugging

```
   r16 <=regs(0);
   r17 <=regs(1);
   debug_carry <= pr_SR.C;
   debug_zero  <= pr_SR.Z;
```

Simulation


NOP
LDI r16,x83
ADC r16,r16
ADC r16,r16
MOV r17,r16
NOP
x83+x83=x106
Signals
Waves
Time
reset
clk
op[15:0]
pc[7:0]
r16[7:0]
r17[7:0]
reg_we
debug_carry
debug_zero
d_reg[3:0]
r_reg[3:0]
k[7:0]
alu_in_a[7:0]
alu_in_b[7:0]
alu_out[7:0]
alu_op[2:0]
add_temp[9:0]
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns