Digital Systems - Mini AVR 1

Implementing NOP, LDI, ADC, MOV

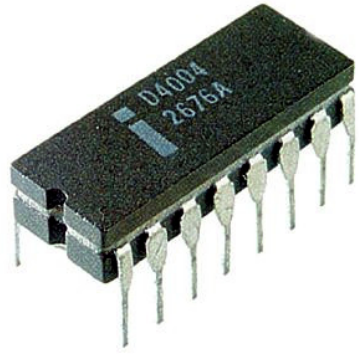
Pere Palà Schönwälder

iTIC http://itic.cat

April 2025

Processor Basics

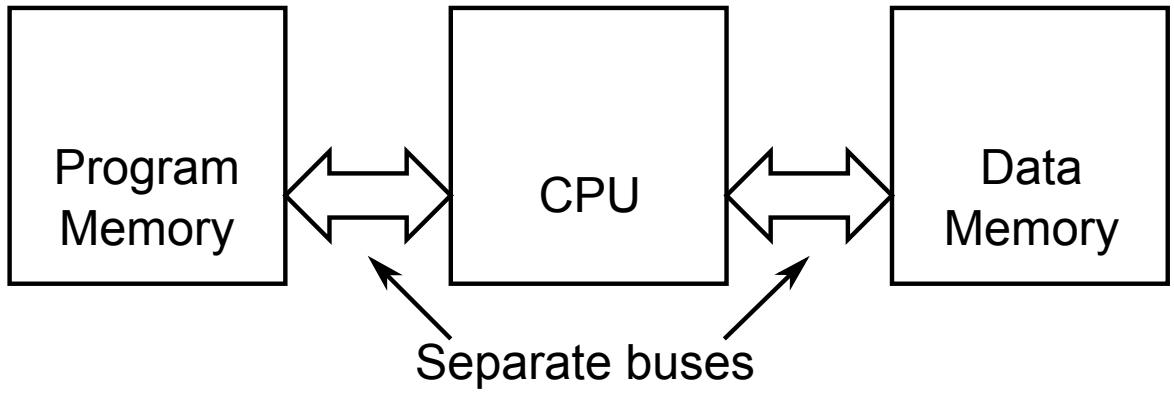
- Processor: A specific (and interesting) (and complex) kind of digital system
- A processor inside a digital system
  - Specialized tasks
  - Multiple small processors
  - Other digital circuitry complements the system
- Processor basics
  - A list of instructions
  - Space to store data
  - Core that acts according to instructions
  - Input and output ports to interact with the real world
- Architectures
  - Von Neumann
  - Harvard



Intel 4004

Harvard Architecture

- Separate memories
  - Program (instruction) memory
  - Data memory

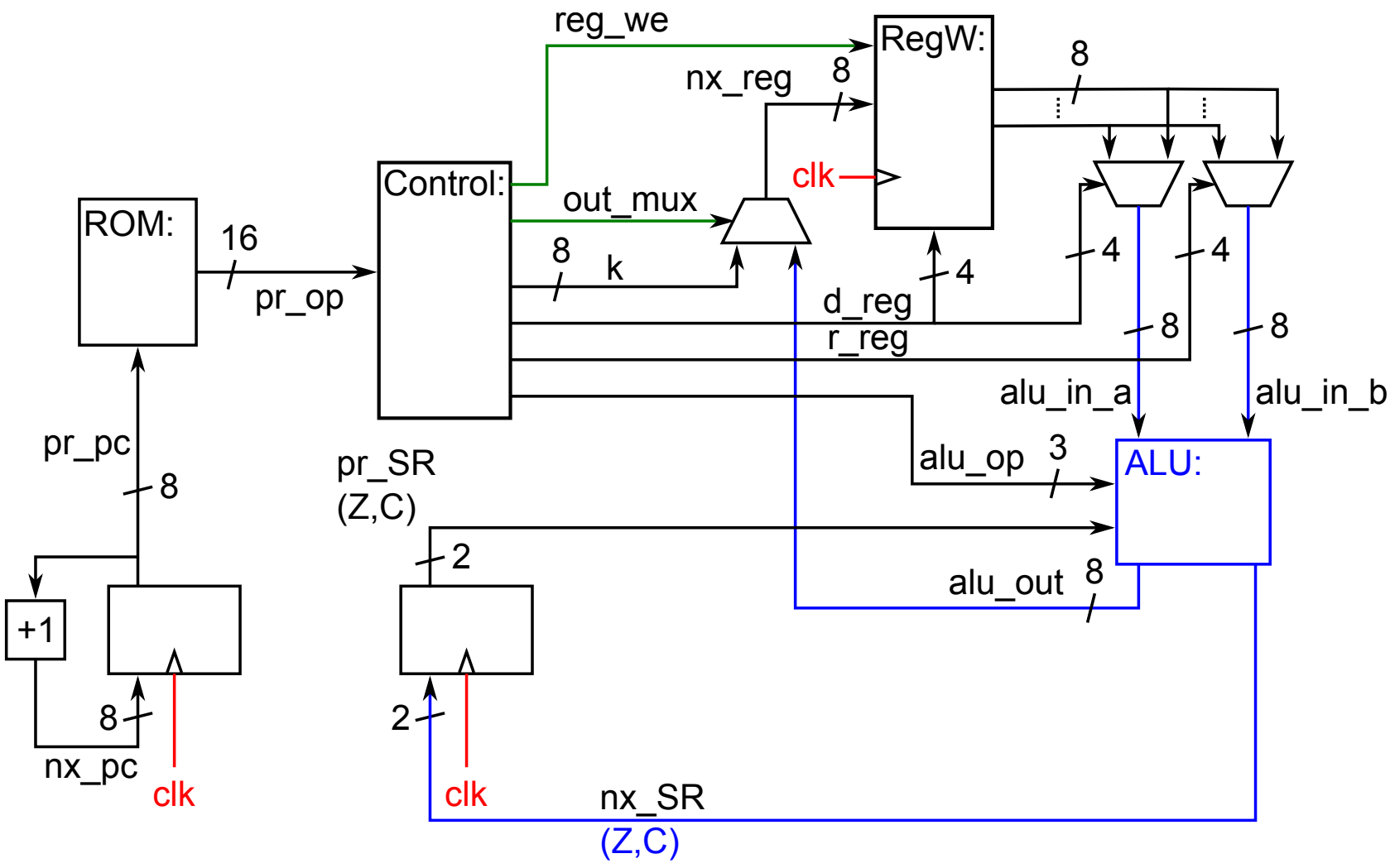


- Allows accessing instructions and data simultaneously
- Eliminates bottleneck of common bus
- Common in small micro-controllers, such as the Atmel AVR

The Mini AVR

- Objective
  - To build a small AVR-compatible micro-controller
  - With a reduced number of instructions
- Why?
  - Just for fun!
  - To be able to port an AVR program to our micro-controller
- Instructions
  - NOP : No operation
  - LDI : Load immediate
  - ADC : Add with carry
  - MOV : Move
  - AND, OR, EOR : And, or, exclusive-or operations
  - RJMP : Relative jump
  - BREQ : Branch if equal
- Some flags of the Status Register (SREG)
  - C: Carry Flag
  - Z: Zero Flag

Overview (don't panic yet!)



Documentation of AVR instructions : NOP

- NOP – No Operation
- Description
  - This instruction performs a single cycle No Operation.
- Operation: No
- Syntax: NOP
- Operands: None
- Program Counter: PC $\leftarrow$ PC + 1
- 16-bit Opcode: `0000 0000 0000 0000`
- Status Register (SREG) and Boolean Formula:

```
I T H S V N Z C
- - - - - - - -
```

Documentation of AVR instructions : LDI

- LDI – Load Immediate
- Description:
  - Loads an 8 bit constant directly to register 16 to 31.
- Operation: Rd $\leftarrow$ K
- Syntax: LDI Rd,K
- Operands: $16 \leq d \leq 31$, $0 \leq K \leq 255$
- Program Counter: PC $\leftarrow$ PC + 1
- 16-bit Opcode: `1110 KKKK dddd KKKK`
- Status Register (SREG) and Boolean Formula:

```
I T H S V N Z C
- - - - - - - -
```

Documentation of AVR instructions : ADC

- ADC – Add with Carry
- Description:
  - Description: Adds two registers and the contents of the C Flag and places the result in the destination register Rd.
- Operation: Rd $\leftarrow$ Rd + Rr + C
- Syntax: ADC Rd,Rr
- Operands: $0 \leq d \leq 31$, $0 \leq r \leq 31$
- Program Counter: PC $\leftarrow$ PC + 1
- 16-bit Opcode: `0001 11rd dddd rrrr`
- Status Register (SREG) and Boolean Formula:

  `I T H S V N Z C`

  \- - $\Leftrightarrow\Leftrightarrow\Leftrightarrow\Leftrightarrow\Leftrightarrow\Leftrightarrow$
- Z: Set if the result is x"00"; cleared otherwise.
- C: Set if there was carry from the MSB of the result; cleared otherwise

Documentation of AVR instructions : MOV

- MOV – Copy Register
- Description:
  - This instruction makes a copy of one register into another. The source register Rr is left unchanged, while the destination register Rd is loaded with a copy of Rr.
- Operation: Rd $\leftarrow$ Rr
- Syntax: MOV Rd,Rr
- Operands: $0 \leq d \leq 31$, $0 \leq r \leq 31$
- Program Counter: PC $\leftarrow$ PC + 1
- 16-bit Opcode: `0010 11rd dddd rrrr`
- Status Register (SREG) and Boolean Formula:

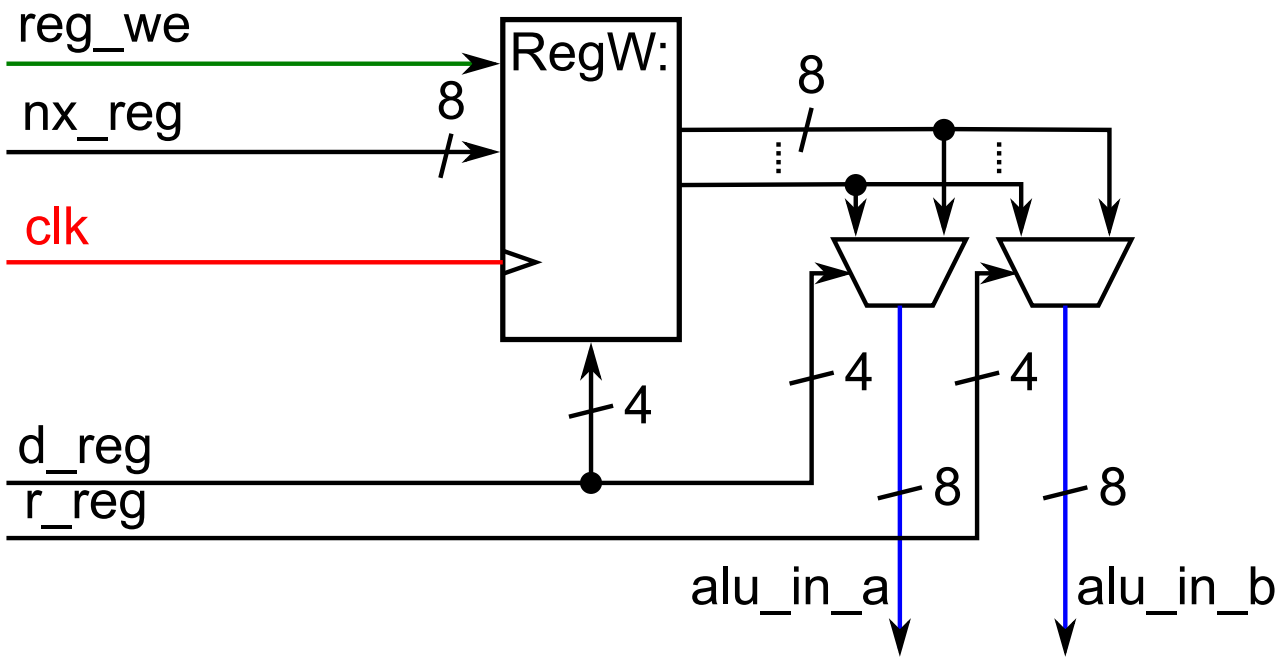
```
I T H S V N Z C
- - - - - - - -
```

References

- Further information:
- 8-bit AVR instruction set manual.

Registers

- Implement only registers from 16 to 31
- ADC 16-bit Opcode
  - `0001 11rd dddd rrrr`
  - `0001 1111 dddd rrrr`
- No stack, stack pointer, etc



VHDL implementation of registers

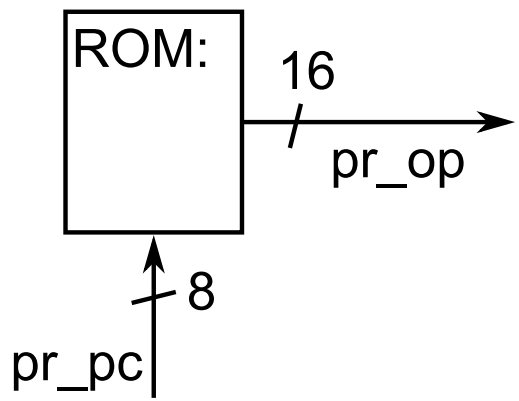
```
type register_bank  is
   array (15 downto 0) of std_logic_vector(7 downto 0);
signal regs : register_bank;              --Registers r16..r31
```

```
   RegW : process(clk) --Register write
   begin
      if rising_edge(clk) then
         if reg_we = '1' then -- write
            regs(to_integer(unsigned(d_reg))) <= nx_reg;
         end if;
      end if;
   end process RegW;
```

```
   --Register read: ALU inputs
   alu_in_a <= regs(to_integer(unsigned(d_reg)));
   alu_in_b <= regs(to_integer(unsigned(r_reg)));
```

Program ROM

- 256 positions (arbitrary limit)
  - Program Counter has to be 8 bits long
- 16-bit opcodes
- Not clocked: Purely combinational ROM
- Given `pr_pc`, the present program counter, the system outputs `pr_op`, the present opcode



Program listing

- A simple program code:

```
NOP             ; do nothing
LDI r16,x83     ; r16 ← x83
ADC r16,r16     ; x83 + x83 = x106 : r16 ← 06 and C is set!
ADC r16,r16     ; x06 + x06 + x01 = x0D and C is cleared
MOV r17,r16     ; r17 ← x0D
NOP             : do nothing
```

- NOP : `0000 0000 0000 0000`
- LDI : `1110 kkkk dddd kkkk`
- ADC : `0001 1111 dddd rrrr` (`0001 11rd dddd rrrr`)
- MOV : `0010 1111 dddd rrrr` (`0010 11rd dddd rrrr`)
- The first 4 bits are enough to determine the instruction
- VHDL definition:

```
constant NOP : std_logic_vector(3 downto 0) := "0000";
constant LDI : std_logic_vector(3 downto 0) := "1110";
constant ADC : std_logic_vector(3 downto 0) := "0001";
constant MOV : std_logic_vector(3 downto 0) := "0010";
```

VHDL implementation of the program memory

```
ROM : process(pr_pc) --Program ROM
begin
   case pr_pc is
   when X"00" =>
      pr_op <= NOP & "0000" & "--------";      -- NOP
   when X"01" =>
      pr_op <= LDI & "1000" & "0000" & "0011";-- LDI r16,x83
   when X"02" =>
      pr_op <= ADC & "1111" & "0000" & "0000";-- ADC r16,r16
   when X"03" =>
      pr_op <= ADC & "1111" & "0000" & "0000";-- ADC r16,r16
   when X"04" =>
      pr_op <= MOV & "1111" & "0001" & "0000";-- MOV r17,r16
   when X"05" =>
      pr_op <= NOP & NOP    & "--------";      -- NOP
   when others => pr_op <= (others => '-');
 end case;
end process;
```

- This works as a first try! Later we will learn to describe a ROM in a way that allows the synthesis tools to choose (much) better implementations.

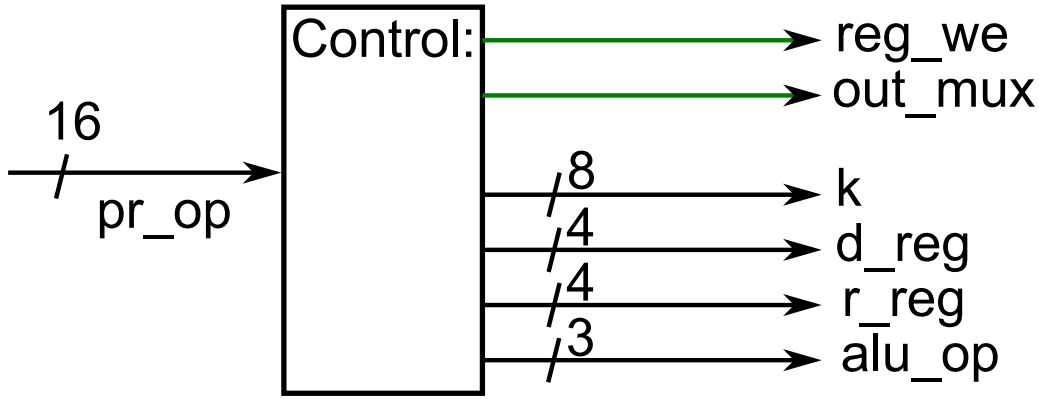
VHDL Implementation of the Status Register

```
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)
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';
```

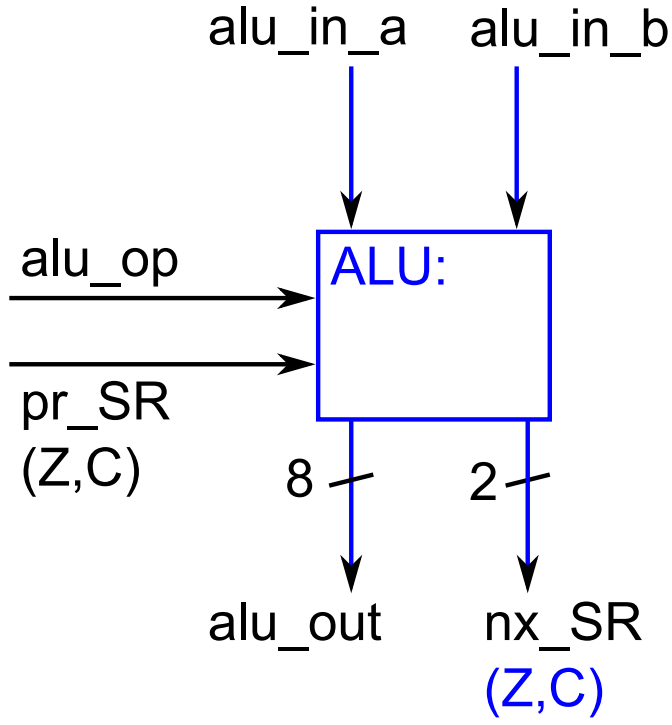
VHDL implementation of the Control Unit (2/2)

```
    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`

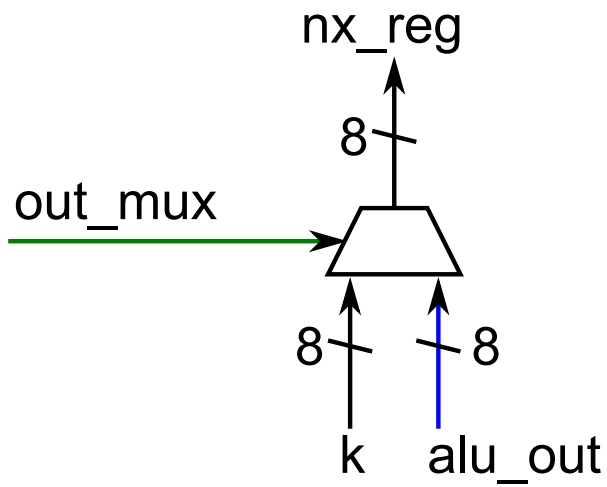


VHDL implementation of the ALU

```
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

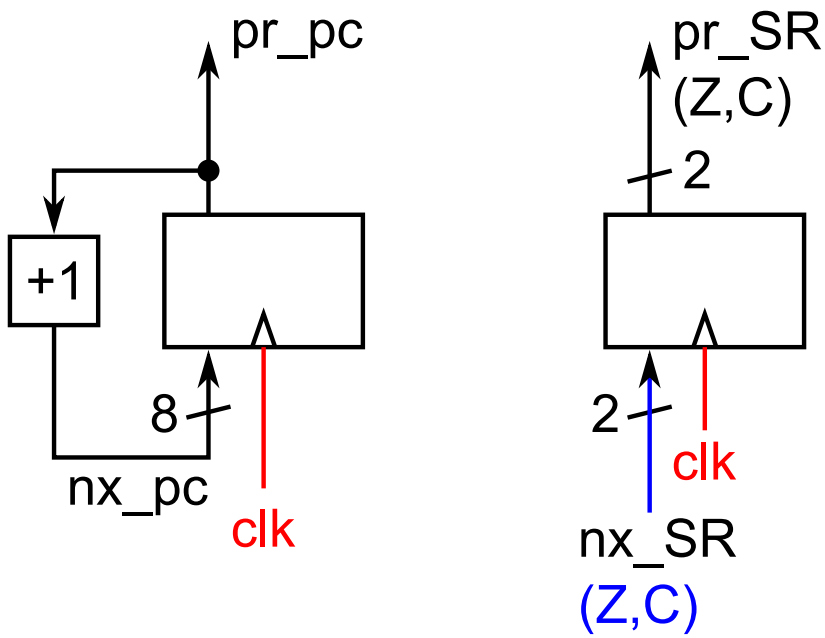


VHDL implementation of the Register Multiplexer

```
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

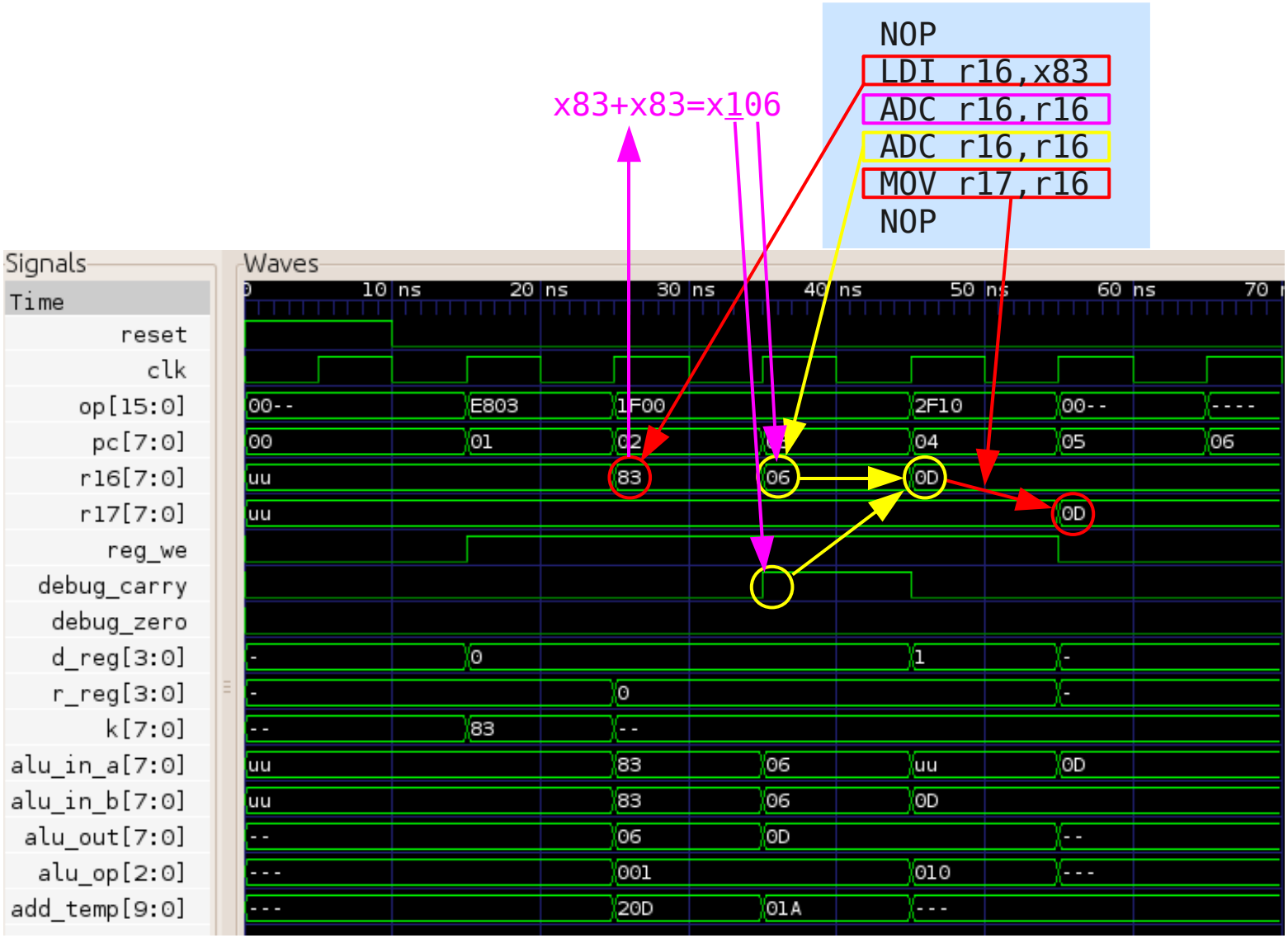
- Entity

```
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
00--
E803
1F00
2F10
00--

00
01
02
04
05
06
uu
83
06
0D
0D
uu
-
0
1
-
-
0
-
--
83
--
uu
83
06
uu
0D
uu
83
06
0D
--
06
0D
--

001
010

20D
01A
