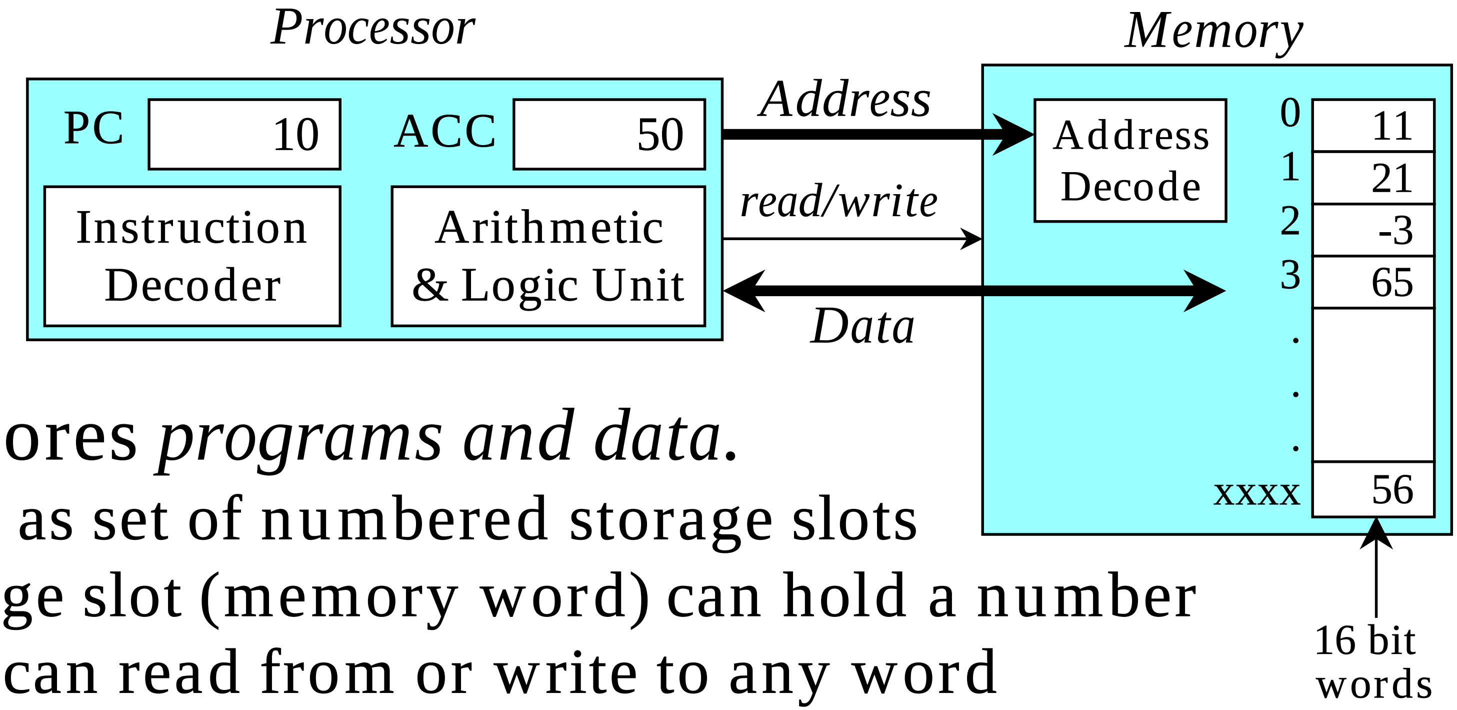
Introduction to Computer Design

- Review of simple processor and memory
- Fetch and execute cycle
- Processor organization
- Executing instructions
- Processor implementation

Read MK 339-376

Basic Processor & Memory



- Memory stores *programs and data.*
  - » organized as set of numbered storage slots
  - » each storage slot (memory word) can hold a number
  - » processor can read from or write to any word
- Fetch & execute cycle
  - » read word whose address is in *Program Counter* (PC) and increment PC
  - » interpret stored value as *instruction* (decoding)
  - » perform instruction using *Accumulator* (ACC) and *Arithmetic & Logic Unit* (ALU)

Simple Instruction Set

0xxx load *ACC* with value stored in memory word **xxx**

1xxx store value in *ACC* into memory word **xxx**

2xxx add value in memory word **xxx** to value in *ACC*

3000 negate the value in *ACC*

3001 halt

4xxx change the value of *PC* to **xxx**

5xxx if the value of *ACC* is zero, change *PC* value to **xxx**

6xxx load *ACC* with value whose address is stored in word **xxx**

7xxx store *ACC* value into word whose address is in word **xxx**

8xxx change *ACC* value to **xxx**

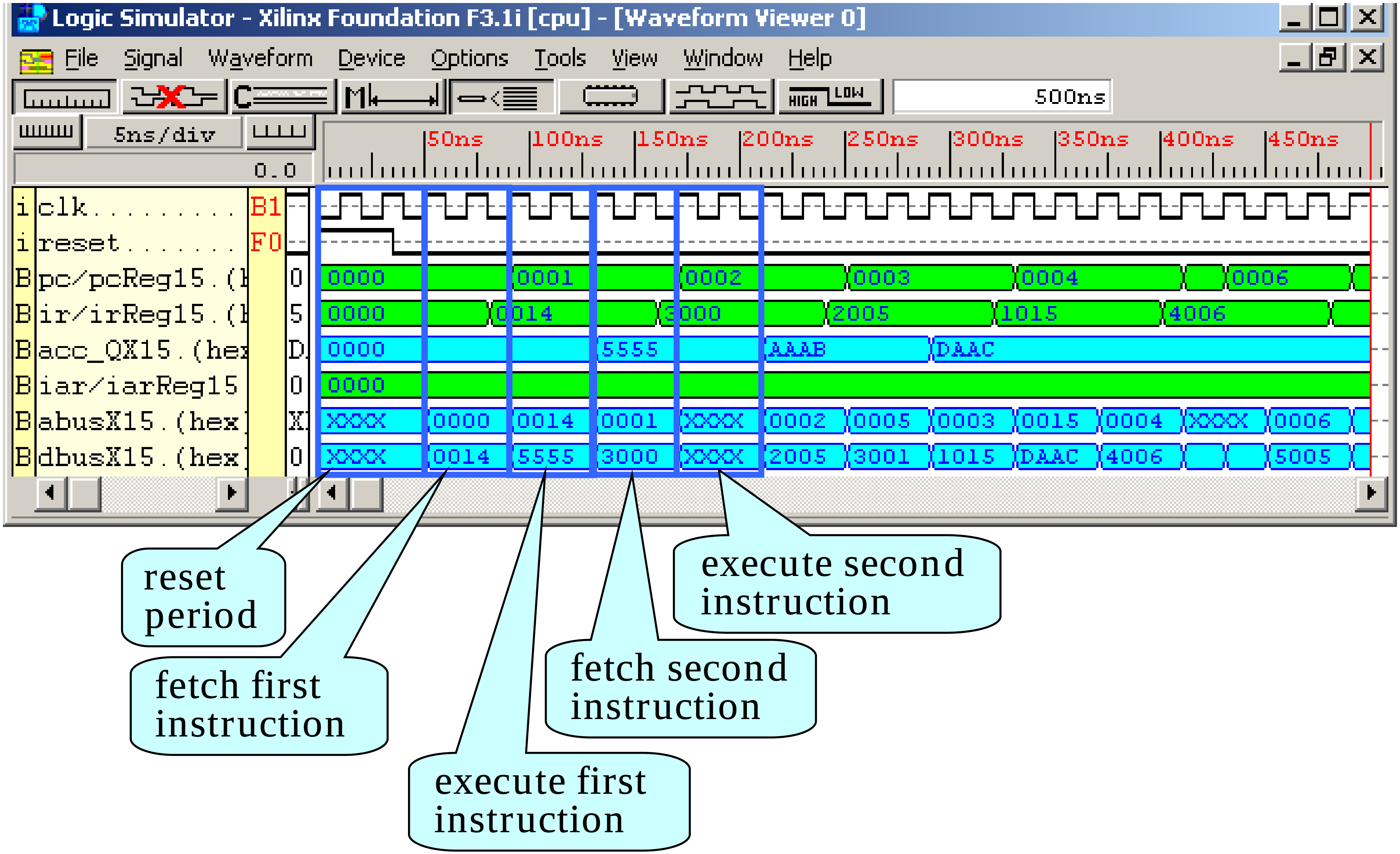
9xxx add **xxx** to the value in *ACC*

Simple Program

- Add the values in locations 0-9 and write sum in 10.

```
Address     Instruction       Comment
0010                          Store sum here
0011                          Pointer to “next” value here
0012 (start)8000 (load “00”)  initialize sum
0013        1010 (store 10)
0014        1011 (store 11)   initialize pointer
0015 (loop) 8010 (load “10”)  if pointer=10, then quit
0016        3000 (negate)
0017        2011 (add 11)
0018        5020 (if 0 goto 20)
0019        6011 (load *11)   sum = sum + *pointer
001a        2010 (add 10)
001b        1010 (store 10)
001c        8001 (load “1”)   pointer = pointer + 1
001d        2011 (add 11)
001e        1011 (store 11)
001f        4015 (goto 15)
0020  (end) 3001 (halt)
```

Execution of a Computer Program


Logic Simulator - Xilinx Foundation F3.1i [cpu] - [Waveform Viewer 0]
File Signal Waveform Device Options Tools View Window Help
500ns
5ns/div
0.0
50ns 100ns 150ns 200ns 250ns 300ns 350ns 400ns 450ns
i clk.......... B1
i reset........ F0
B pc/pcReg15.(h 0 0000 0001 0002 0003 0004 0006
B ir/irReg15.(h 5 0000 0014 3000 2005 1015 4006
B acc_QX15.(hex D 0000 5555 AAAB DAAC
B iar/iarReg15 0 0000
B abusX15.(hex) X XXXX 0000 0014 0001 XXXX 0002 0005 0003 0015 0004 XXXX 0006
B dbusX15.(hex) 0 XXXX 0014 5555 3000 XXXX 2005 3001 1015 DAAC 4006 5005

reset period

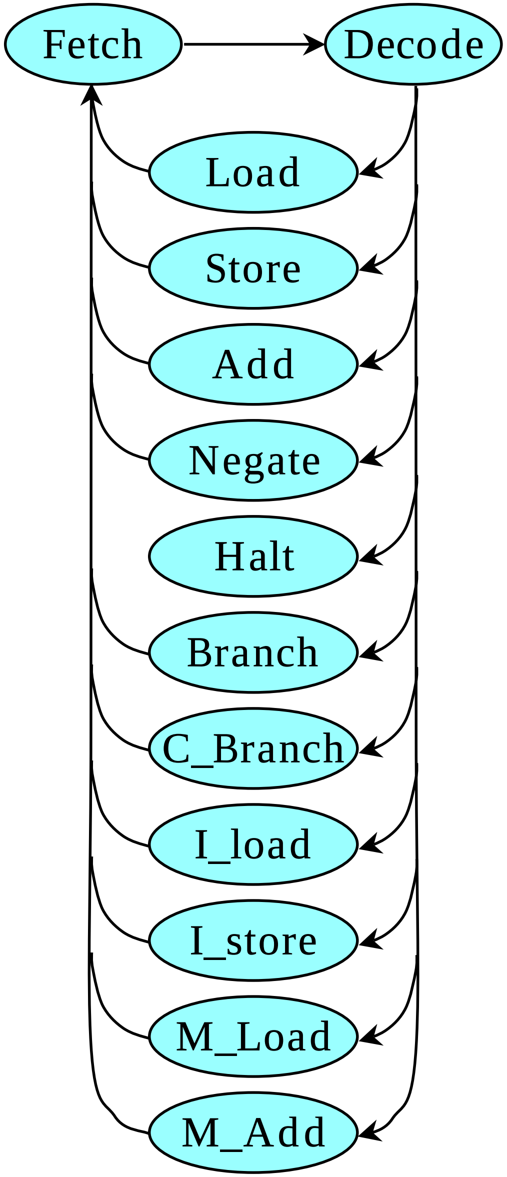
fetch first instruction

execute first instruction

fetch second instruction

execute second instruction

Processing Cycle

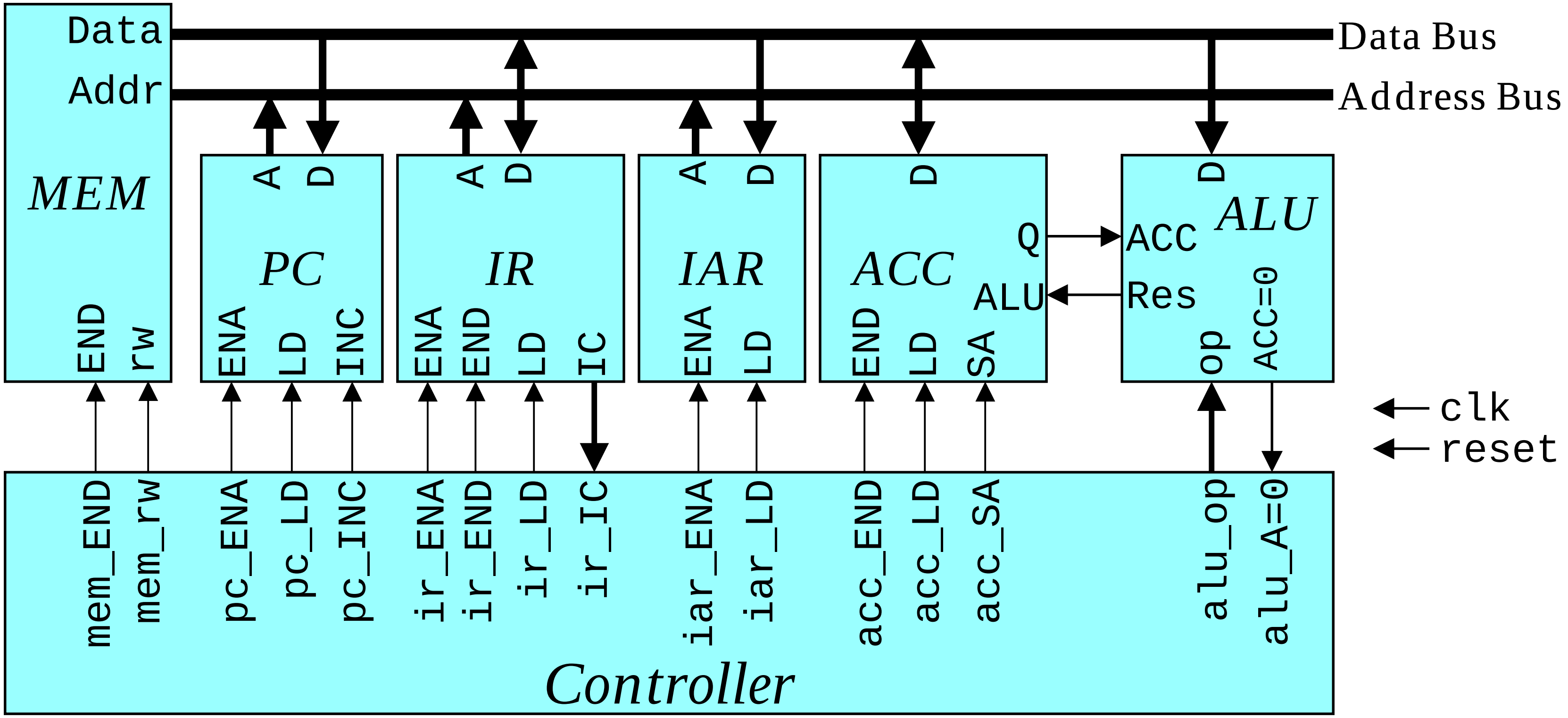


- Instruction fetch
  - » *PC* used to read word from memory
  - » *PC* is incremented
- Instruction decode
  - » first 4 bits of retrieved instruction are decoded to determine what to do
  - » appropriate circuitry activated
- Instruction execution
  - » retrieve additional memory words
  - » write to memory
  - » modify *PC* or *ACC* contents
  - » may take different amounts of time to complete

Instruction Execution

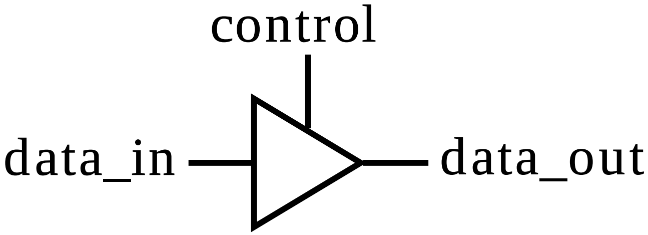
- Load
  - » transfer data from memory to *ACC*, using low 12 bits of instruction word as memory address
  - » requires asserting of memory signals and *ACC* load signal
- Conditional branch
  - » determine if *ACC*=0
  - » if so, transfer low 12 bits of instruction word to *PC*
  - » requires assertion of *PC* load signal
- Indirect store
  - » transfer data from memory to Indirect Address Register (*IAR*) using low 12 bits of instruction word as memory address
  - » transfer data from *ACC* to memory, using *IAR* contents as address
  - » requires assertion of *IAR* load and memory write signals

Processor Block Diagram

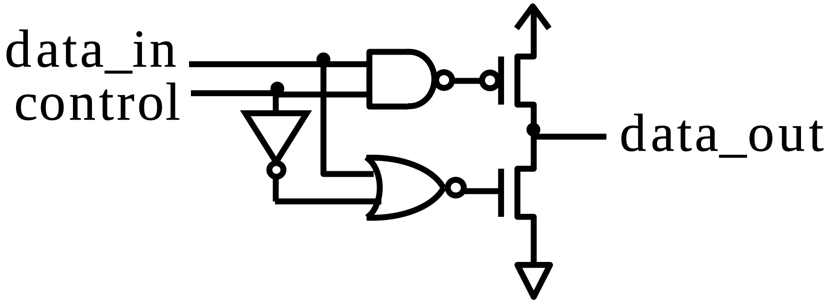


- Processor consists of set of registers, *ALU* & controller.
- Controller contains sequential circuit that asserts control signals needed to fetch & execute insructions.

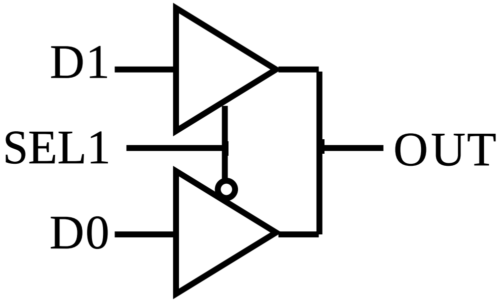
Tri-State Buffers



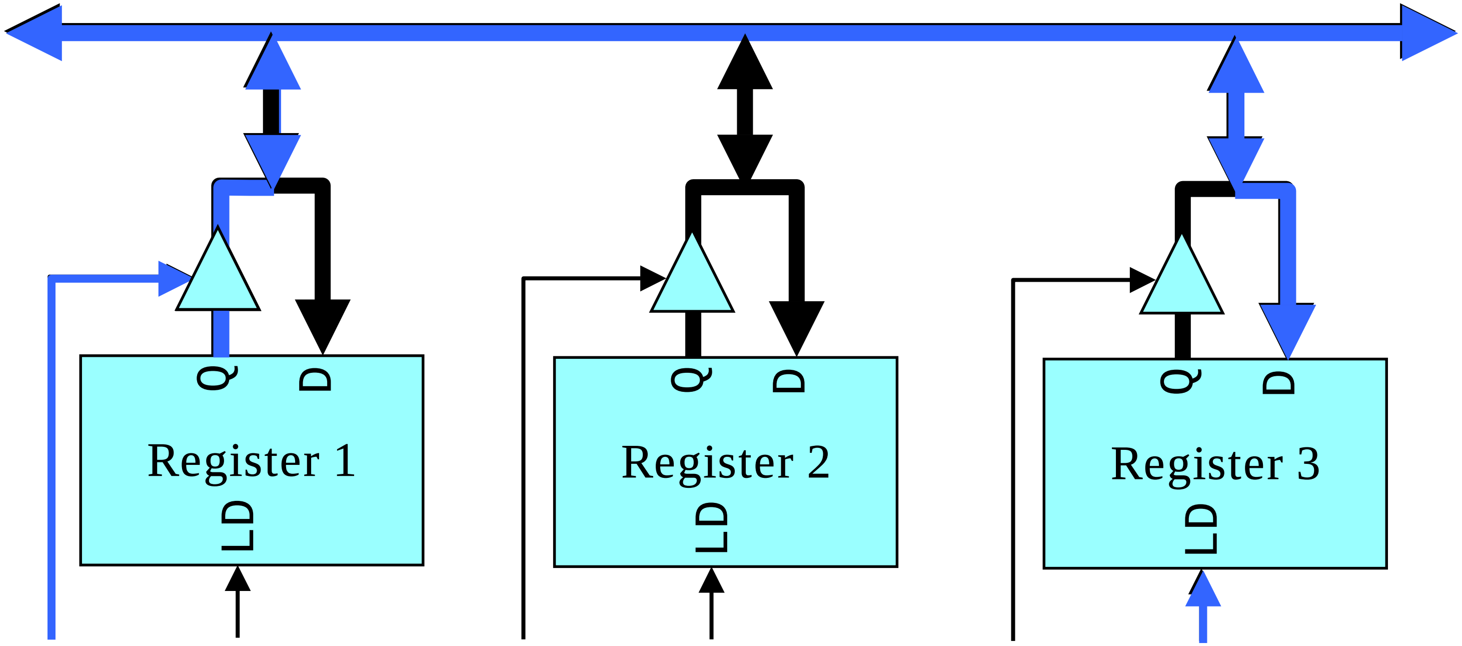
CMOS implementation



- A *tri-state buffer* has a data input and a control input.
  - » when control input is asserted, output equals input
  - » when control input is not asserted, output is disconnected from input - called *high impedance state*
- Tri-state buffers, can be used to build "distributed"multiplexors.
- Shared outputs are called *buses*.
- Also allows single wire to be used as data input and output.

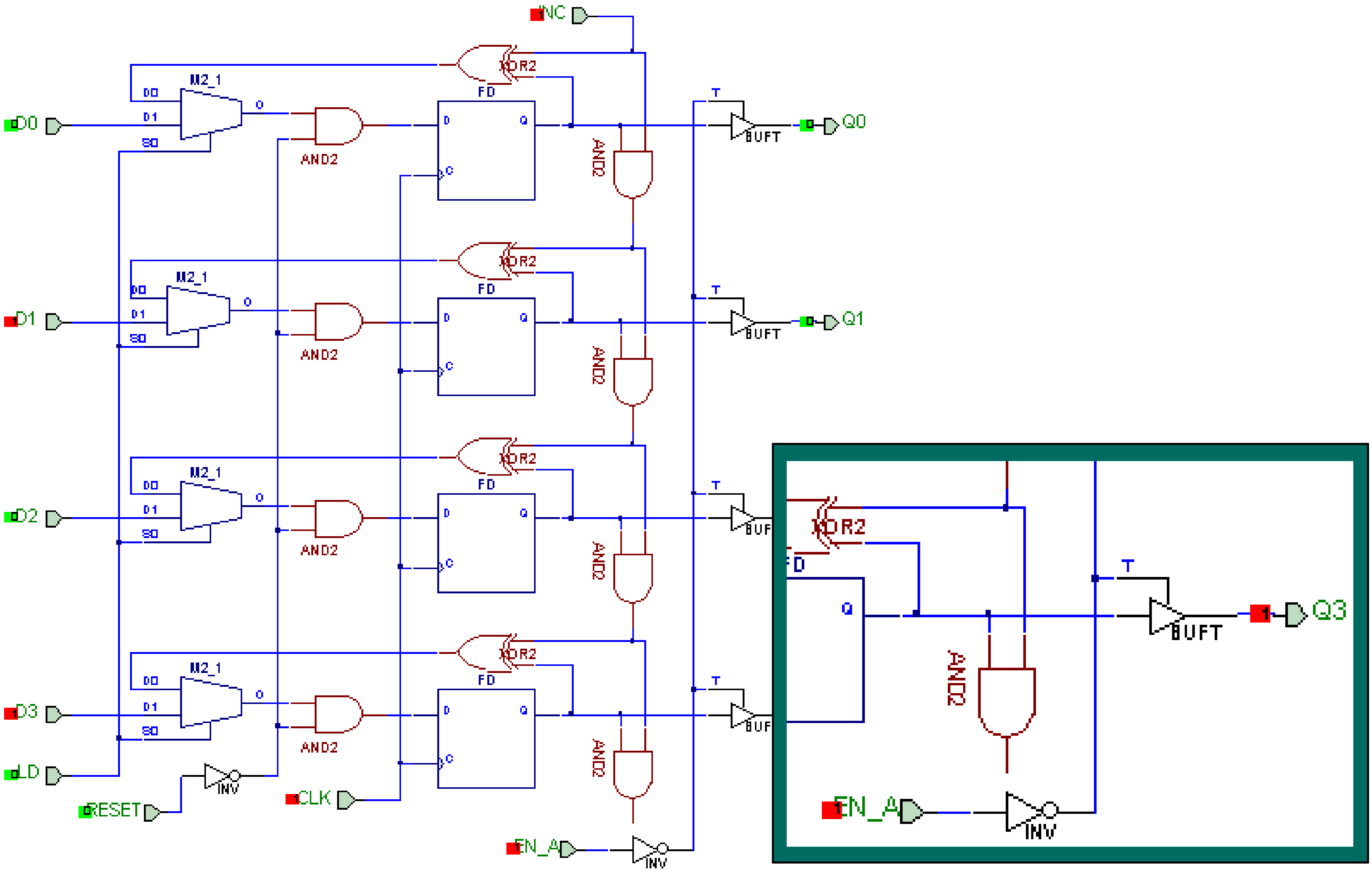


Data Transfer Using Buses

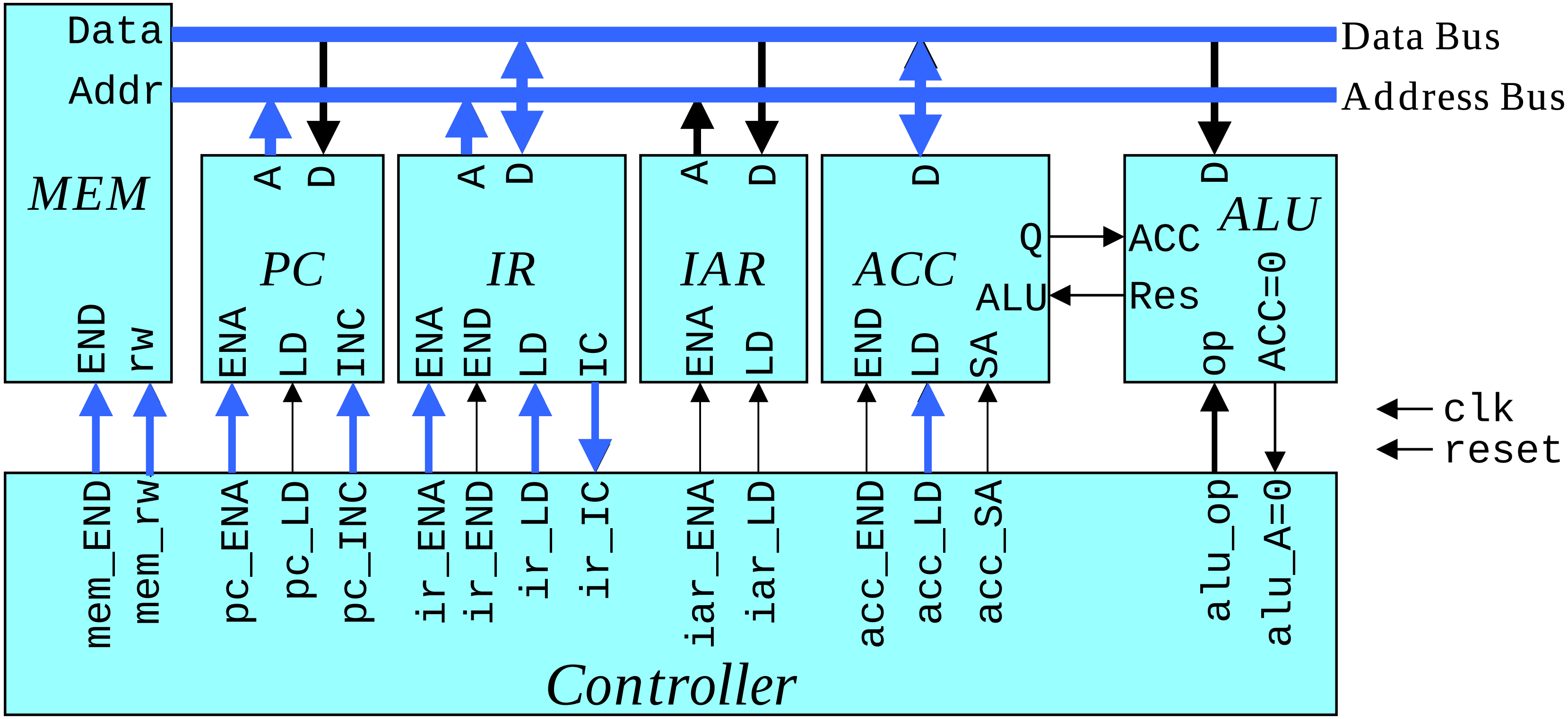


- A *bus* is a shared set of wires used to transfer data among any of several sources / destinations.
- Data transfers involve:
  - » enabling source to place data on the bus
  - » loading data into destination

Program Counter Schematic (4 bit)

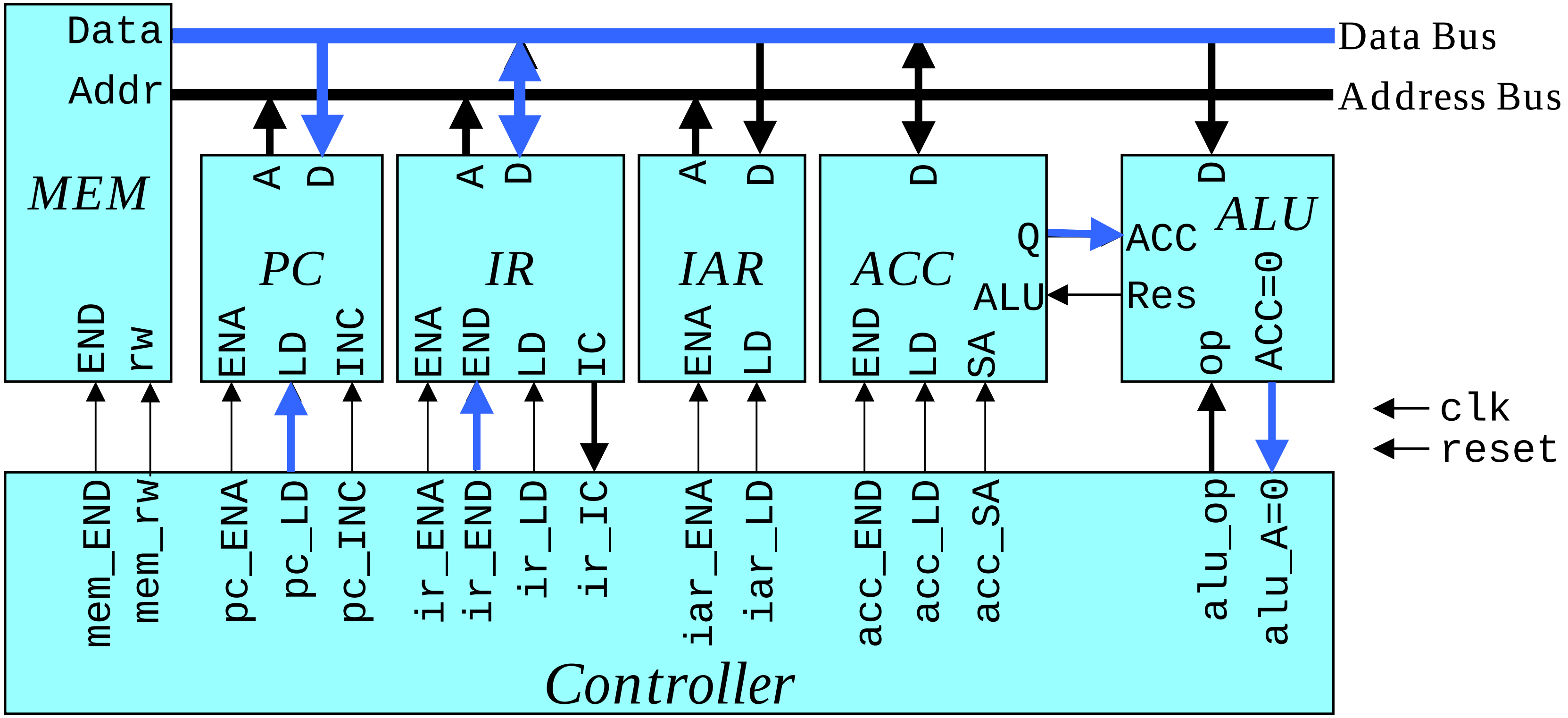


Load Instruction



- Fetch instruction and increment *PC*.
- Transfer data from memory to *ACC* using low 12 bits of instruction as address

Conditional Branch

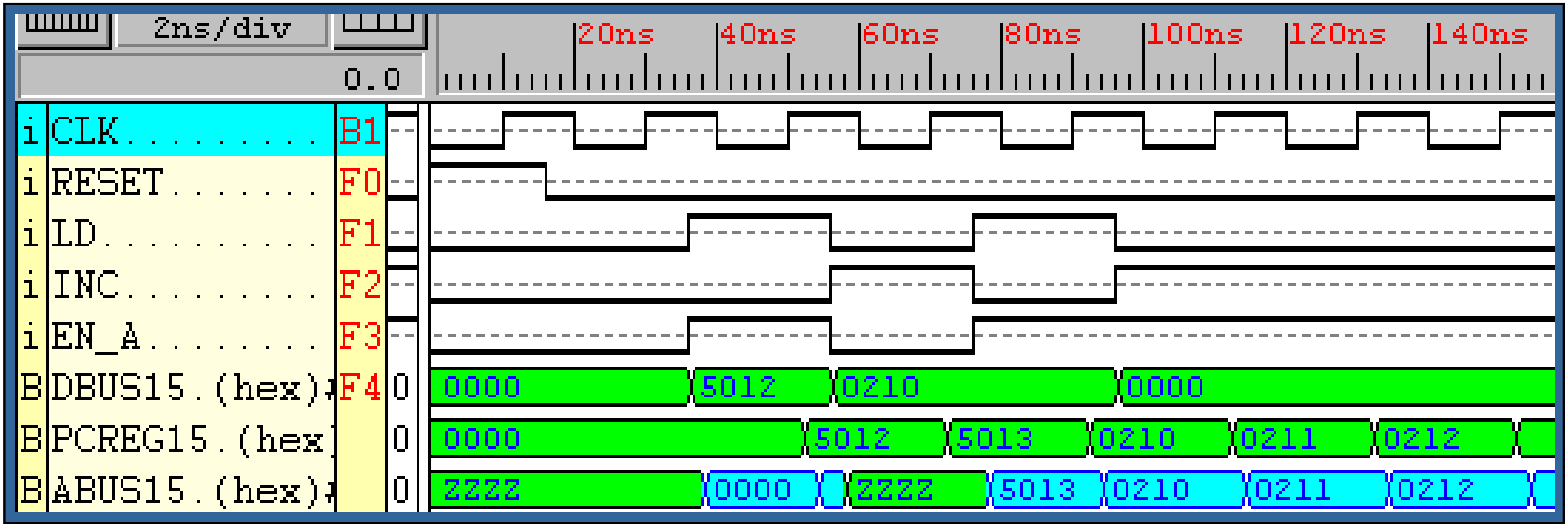


- Determine if *ACC*=0.
- If so, transfer low 12 bits of instruction word to *PC*.

Program Counter

```
entity program_counter is
    port (
        clk, en_A, ld, inc, reset: in STD_LOGIC;
        aBus: out STD_LOGIC_VECTOR(15 downto 0);
        dBus: in STD_LOGIC_VECTOR(15 downto 0)
    );
end program_counter;
architecture pcArch of program_counter is
signal pcReg: STD_LOGIC_VECTOR(15 downto 0);
begin
  process(clk) begin
       if clk'event and clk = '1' then
             if reset = '1' then
                    pcReg <= x"0000";
             elsif ld = '1' then
                    pcReg <= dBus;
             elsif inc = '1' then
                    pcReg <= pcReg + x"0001";
             end if;
       end if;
  end process;
  aBus <= pcReg when en_A = '1' else "ZZZZZZZZZZZZZZZZ";
end pcArch;
```

PC Simulation



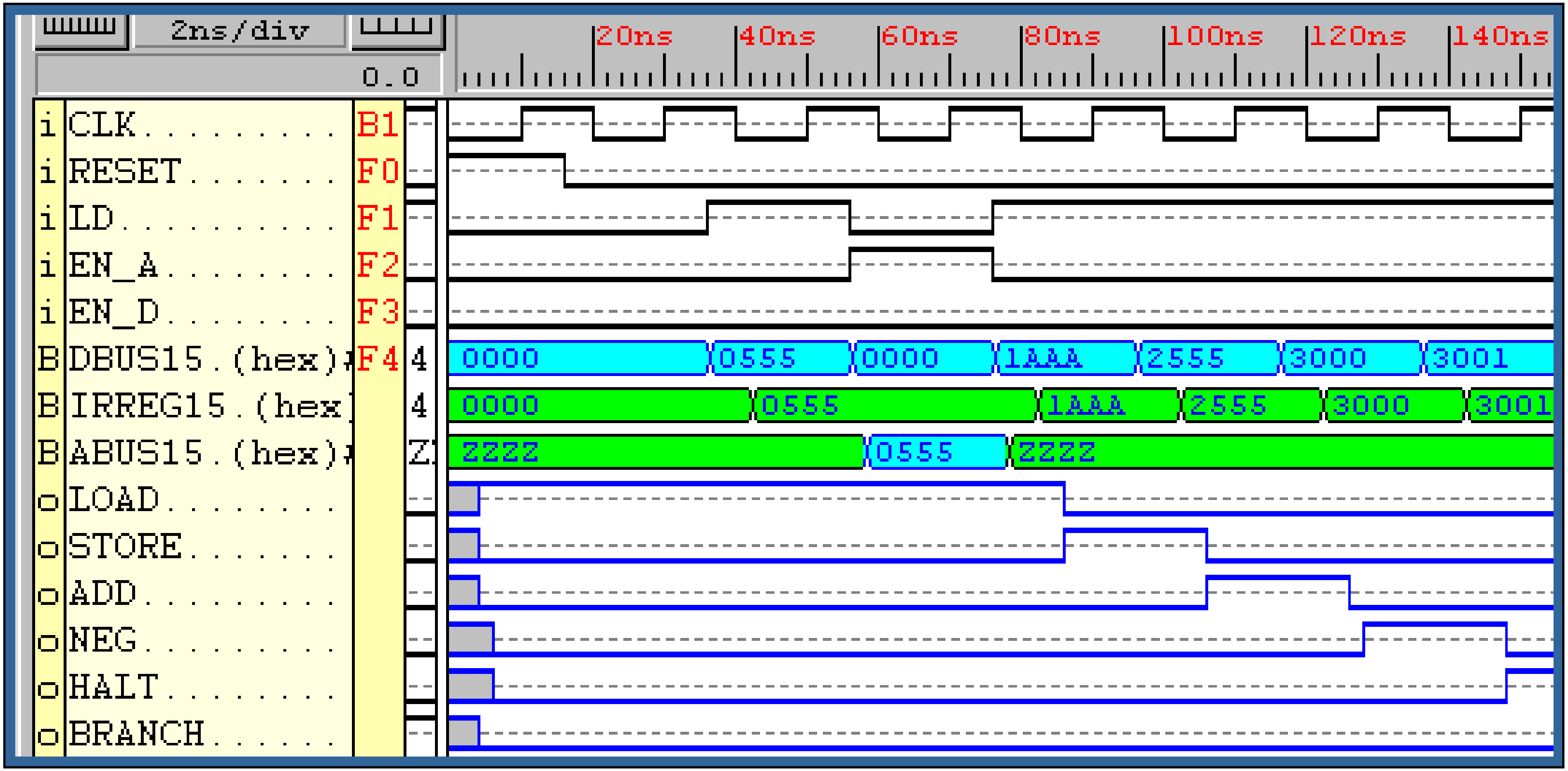
Instruction Register

```
entity instruction_register is
    port (
        clk, en_A, en_D, ld, reset: in STD_LOGIC;
        aBus: out STD_LOGIC_VECTOR(15 downto 0);
        dBus: inout STD_LOGIC_VECTOR(15 downto 0);
        load, store, add, neg, halt, branch: out STD_LOGIC;
        cbranch, iload, istore, mload, madd: out STD_LOGIC
    );
end instruction_register;
architecture irArch of instruction_register is
signal irReg: STD_LOGIC_VECTOR(15 downto 0);
begin
  process(clk) begin
       if clk'event and clk = '0' then
              if reset = '1' then
                     irReg <= x"0000";
              elsif ld = '1' then
                     irReg <= dBus;
              end if;
       end if;
  end process;
```

Instruction Register

```
  aBus <= "0000" & irReg(11 downto 0) when en_A = '1' else
        "ZZZZZZZZZZZZZZZZ";
  dBus <= "0000" & irReg(11 downto 0) when en_D = '1' else
        "ZZZZZZZZZZZZZZZZ";
  load  <= '1' when irReg(15 downto 12) = x"0"  else '0';
  store <= '1' when irReg(15 downto 12) = x"1"  else '0';
  add   <= '1' when irReg(15 downto 12) = x"2"  else '0';
  neg   <= '1' when irReg = x"3" & x"000"       else '0';
  halt  <= '1' when irReg = x"3" & x"001"       else '0';
  branch<= '1' when irReg(15 downto 12) = x"4"  else '0';
  cbranch<= '1' when irReg(15 downto 12) = x"5" else '0';
  iload <= '1' when irReg(15 downto 12) = x"6"  else '0';
  istore<= '1' when irReg(15 downto 12) = x"7"  else '0';
  mload <= '1' when irReg(15 downto 12) = x"8"  else '0';
  madd  <= '1' when irReg(15 downto 12) = x"9"  else '0';
end irArch;
```

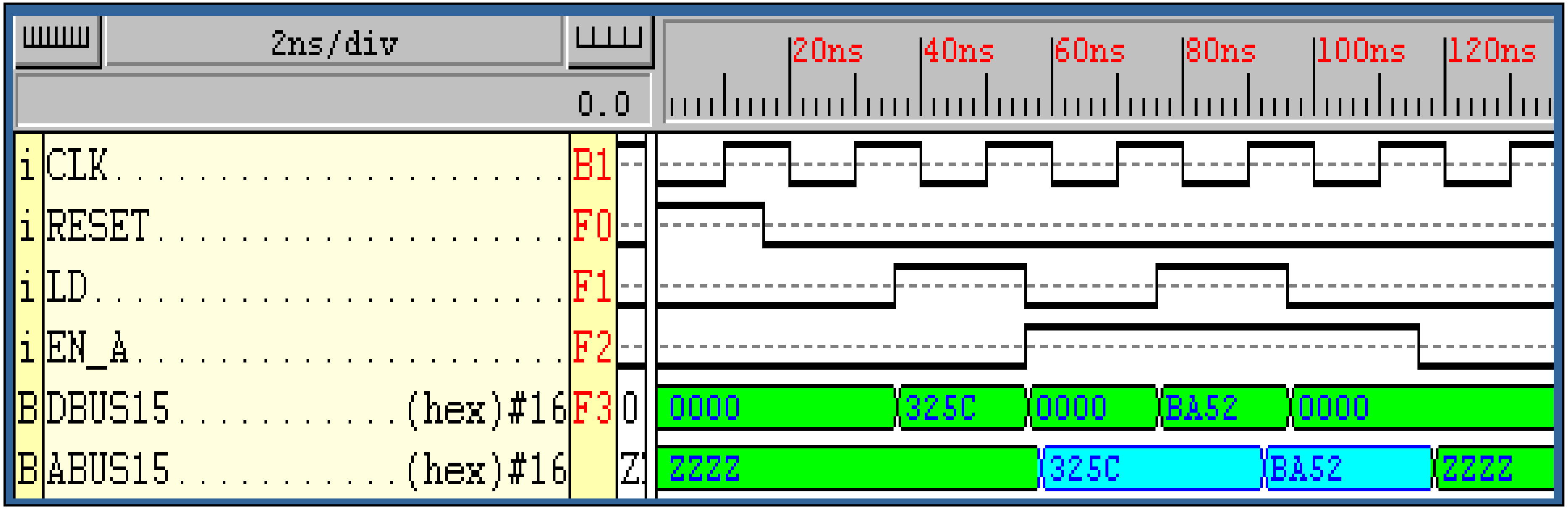
IR Simulation



Indirect Address Register

```
entity indirect_addr_register is
    port (
        clk, en_A, ld, reset: in STD_LOGIC;
        aBus: out STD_LOGIC_VECTOR(15 downto 0);
        dBus:  in STD_LOGIC_VECTOR(15 downto 0)
    );
end indirect_addr_register;
architecture iarArch of indirect_addr_register is
signal iarReg: STD_LOGIC_VECTOR(15 downto 0);
begin
  process(clk) begin
       if clk'event and clk = '1' then
             if reset = '1' then
                    iarReg <= x"0000";
             elsif ld = '1' then
                    iarReg <= dBus;
             end if;
       end if;
  end process;
  aBus <= iarReg when en_A = '1' else
         "ZZZZZZZZZZZZZZZZ";
end iarArch;
```

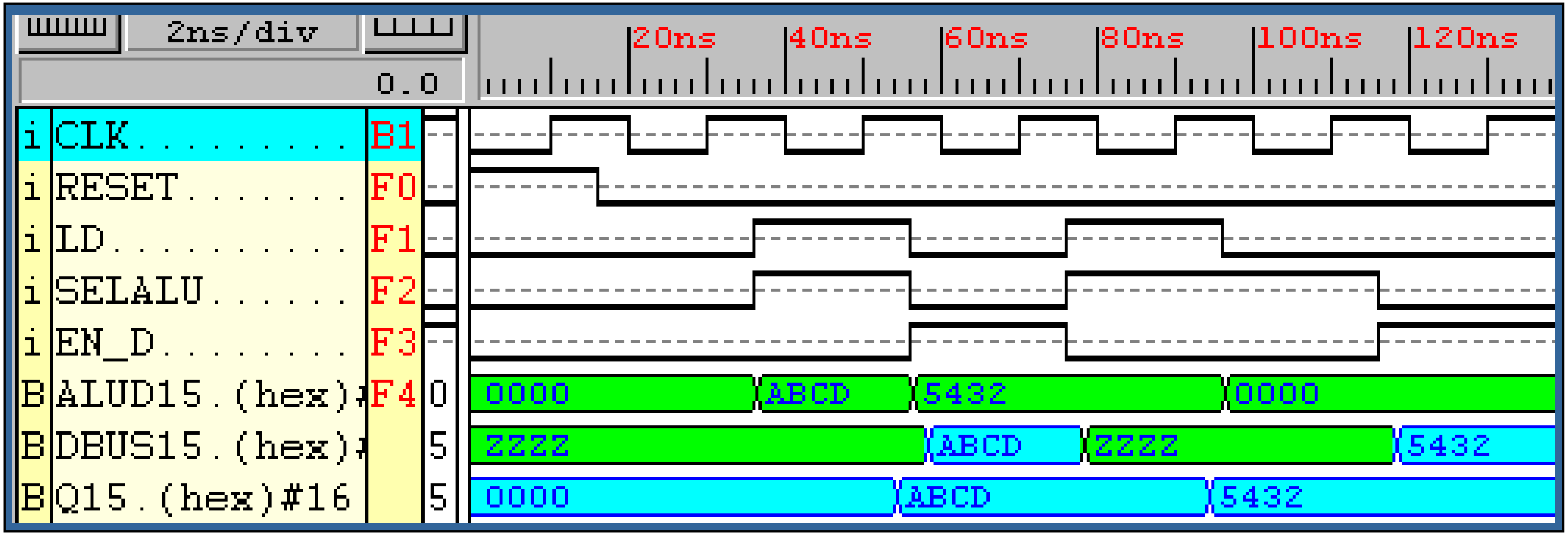
IAR Simulation



Accumulator

```
entity accumulator is
    port (
        clk, en_D, ld, selAlu, reset: in STD_LOGIC;
        aluD: in STD_LOGIC_VECTOR(15 downto 0);
        dBus, q: inout STD_LOGIC_VECTOR(15 downto 0)
    );
end accumulator;
architecture accArch of accumulator is
signal accReg: STD_LOGIC_VECTOR(15 downto 0);
begin
  process(clk) begin
       if clk'event and clk = '1' then
             if reset = '1' then
                   accReg <= x"0000";
             elsif ld = '1' and selAlu = '1' then
                   accReg <= aluD;
             elsif ld = '1' and selAlu = '0' then
                   accReg <= dBus;
             end if;
       end if;
  end process;
  dBus <= accReg when en_D = '1' else "ZZZZZZZZZZZZZZZZ";
  q <= accReg;
end accArch;
```

ACC Simulation

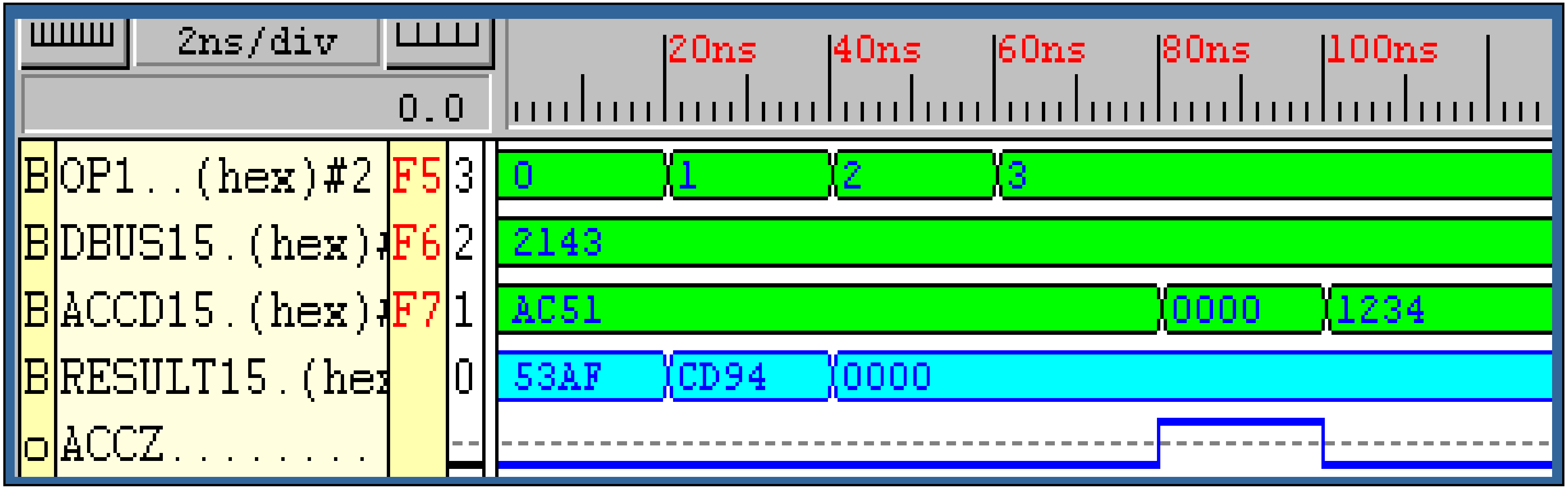


ALU

```
entity alu is
    port (
        op: in STD_LOGIC_VECTOR(1 downto 0);
        accD: in STD_LOGIC_VECTOR(15 downto 0);
        dBus: in STD_LOGIC_VECTOR(15 downto 0);
        result: out STD_LOGIC_VECTOR(15 downto 0);
        accZ: out STD_LOGIC
    );
end alu;

architecture aluArch of alu is
begin
  result <= (not accD) + x"0001" when op ="00" else
           accD + dBus when op ="01" else
           x"0000";
  accZ <= '1' when accD = x"0000" else '0';
end aluArch;
```

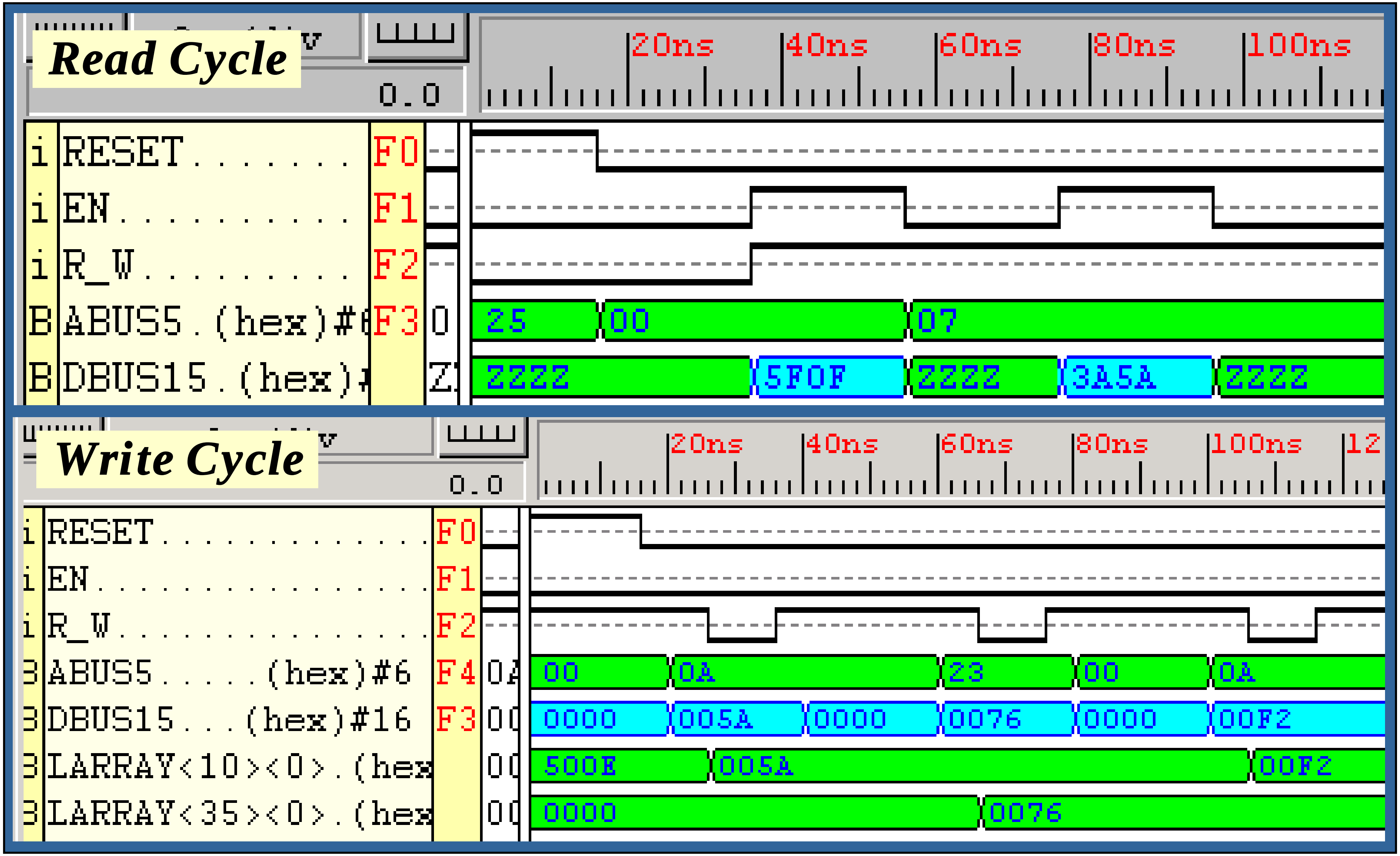
ALU Simulation



Memory

```
entity ram is
    port (
        r_w, en, reset: in STD_LOGIC;
        aBus: in STD_LOGIC_VECTOR(15 downto 0);
        dBus: inout STD_LOGIC_VECTOR(15 downto 0)
    );
end ram;
architecture ramArch of ram is
type ram_typ is array(0 to 63) of STD_LOGIC_VECTOR(15 downto 0);
signal ram: ram_typ;
begin
  process(reset, r_w) begin
       if reset = '1' then
               ram(0)  <= x"4012"; . . . ram(32) <= x"3001";
               for i in 33 to 63 loop
                      ram(i) <= x"0000";
               end loop;
       elsif r_w'event and r_w = '0' then
               ram(conv_integer(unsigned(aBus))) <= dBus;
       end if;
  end process;
  dBus <= ram(conv_integer(unsigned(aBus)))when reset = '0‘
       and en = '1' and r_w = '1' else "ZZZZZZZZZZZZZZZZ";
end ramArch;
```

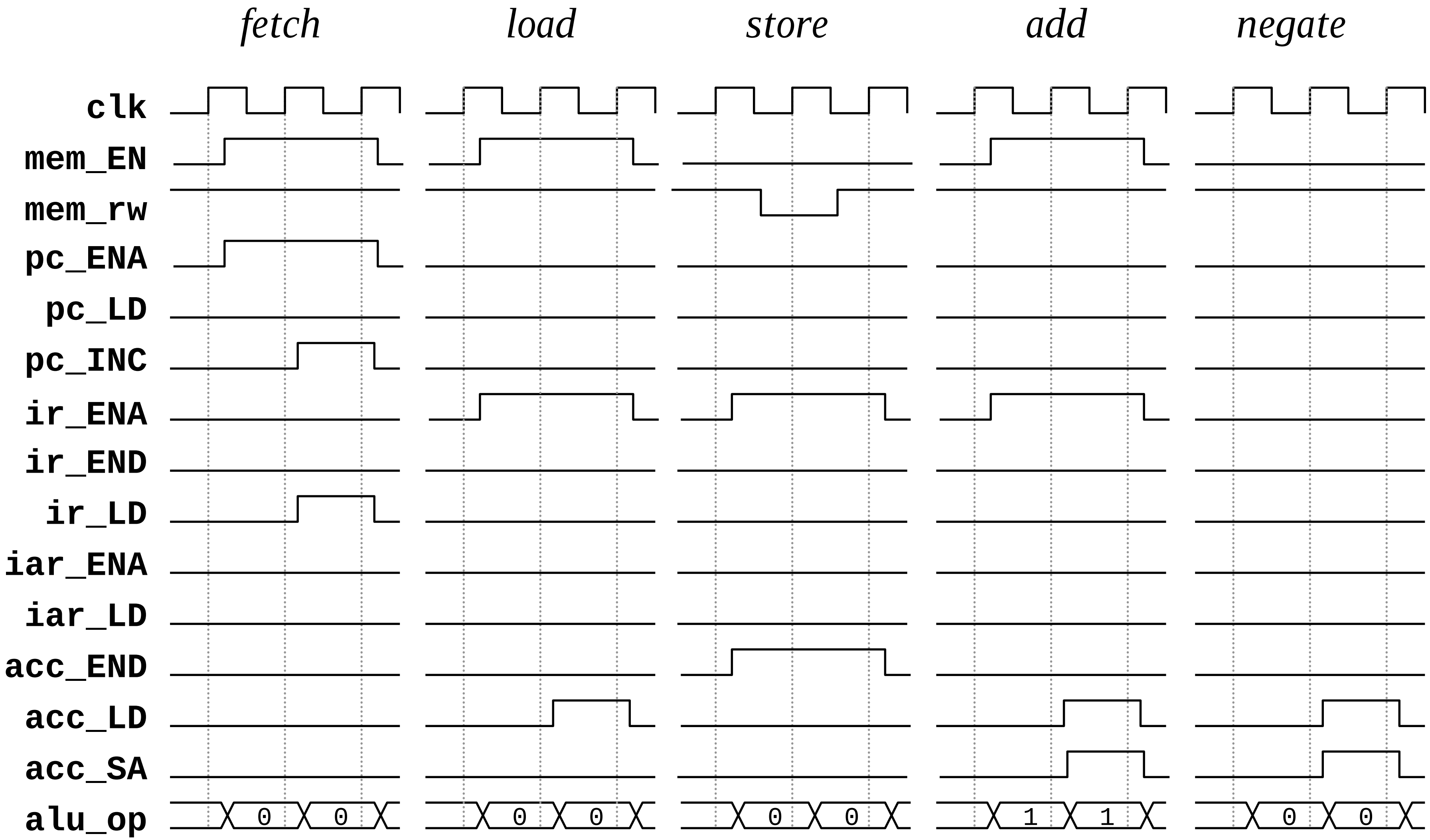
Memory Simulation



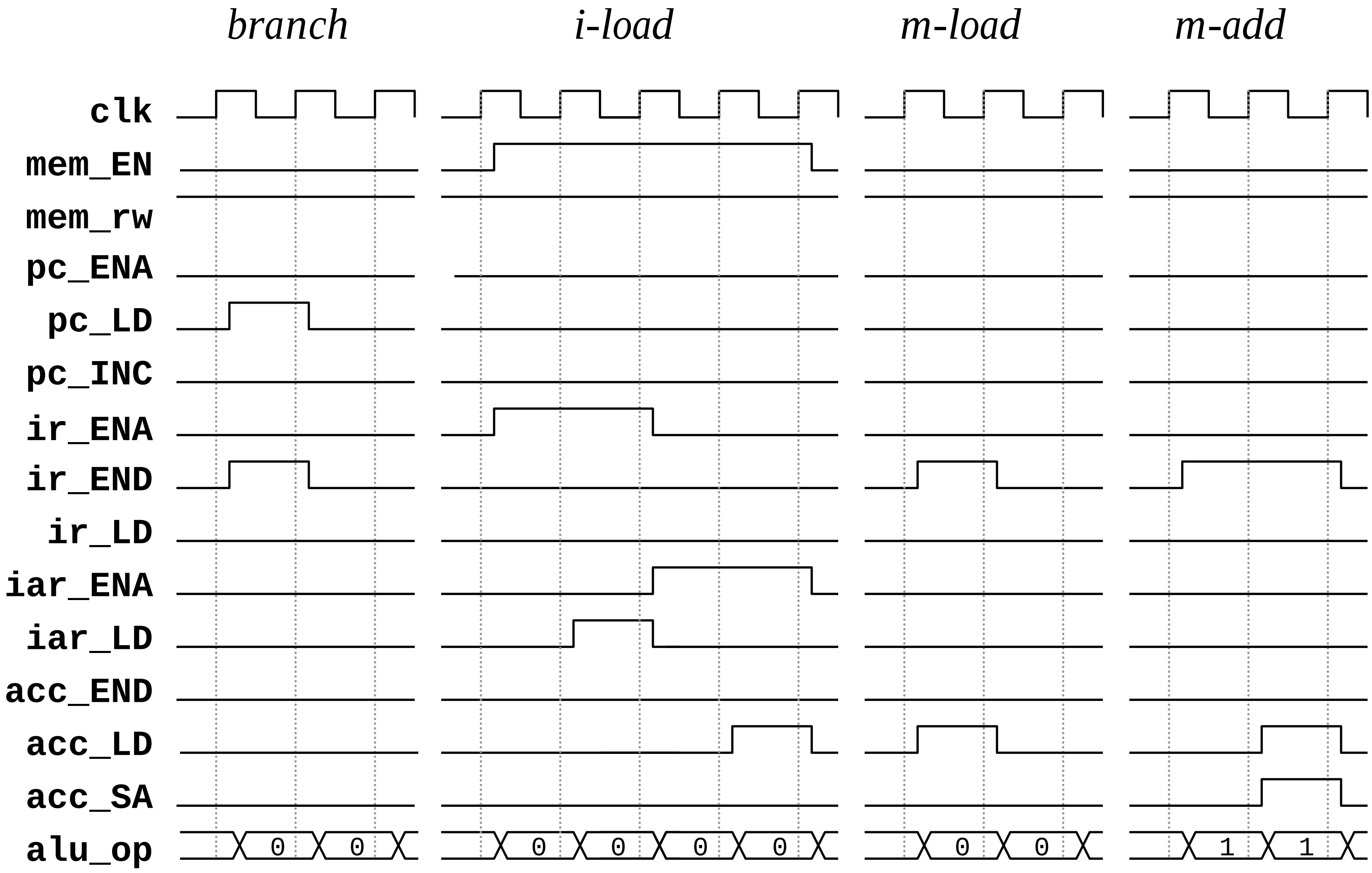
Read Cycle

Write Cycle

Signal Timing for Controller



Signal Timing for Controller



Controller

```
entity controller is
    port (
       clk, reset:                      in  STD_LOGIC;
       mem_en, mem_rw:                  out STD_LOGIC;
       pc_enA, pc_ld, pc_inc:           out STD_LOGIC;

       ir_enA, ir_enD, ir_ld:           out STD_LOGIC;
       ir_load, ir_store, ir_add:       in  STD_LOGIC;
       ir_neg, ir_halt, ir_branch:      in  STD_LOGIC;
       ir_cbranch, ir_iload:            in  STD_LOGIC;
       ir_istore, ir_dload, ir_dadd:    in  STD_LOGIC;

       iar_enA, iar_ld:                 out STD_LOGIC;
       acc_enD, acc_ld, acc_selAlu:     out STD_LOGIC;
       alu_accZ:                        in  STD_LOGIC;
       alu_op:                   out STD_LOGIC_VECTOR(1 downto 0)
    );
end controller;
```

Controller

```
architecture controllerArch of controller is
type state_type is (reset_state,
             fetch0, fetch1,
             load0, load1,
             store0, store1,
             add0, add1,
             negate0, negate1,
             halt,
             branch0, branch1,
             cbranch0, cbranch1,
             iload0, iload1, iload2, iload3,
             istore0, istore1, istore2, istore3,
             mload0, mload1,
             madd0, madd1
             );
signal state: state_type;
```

Controller

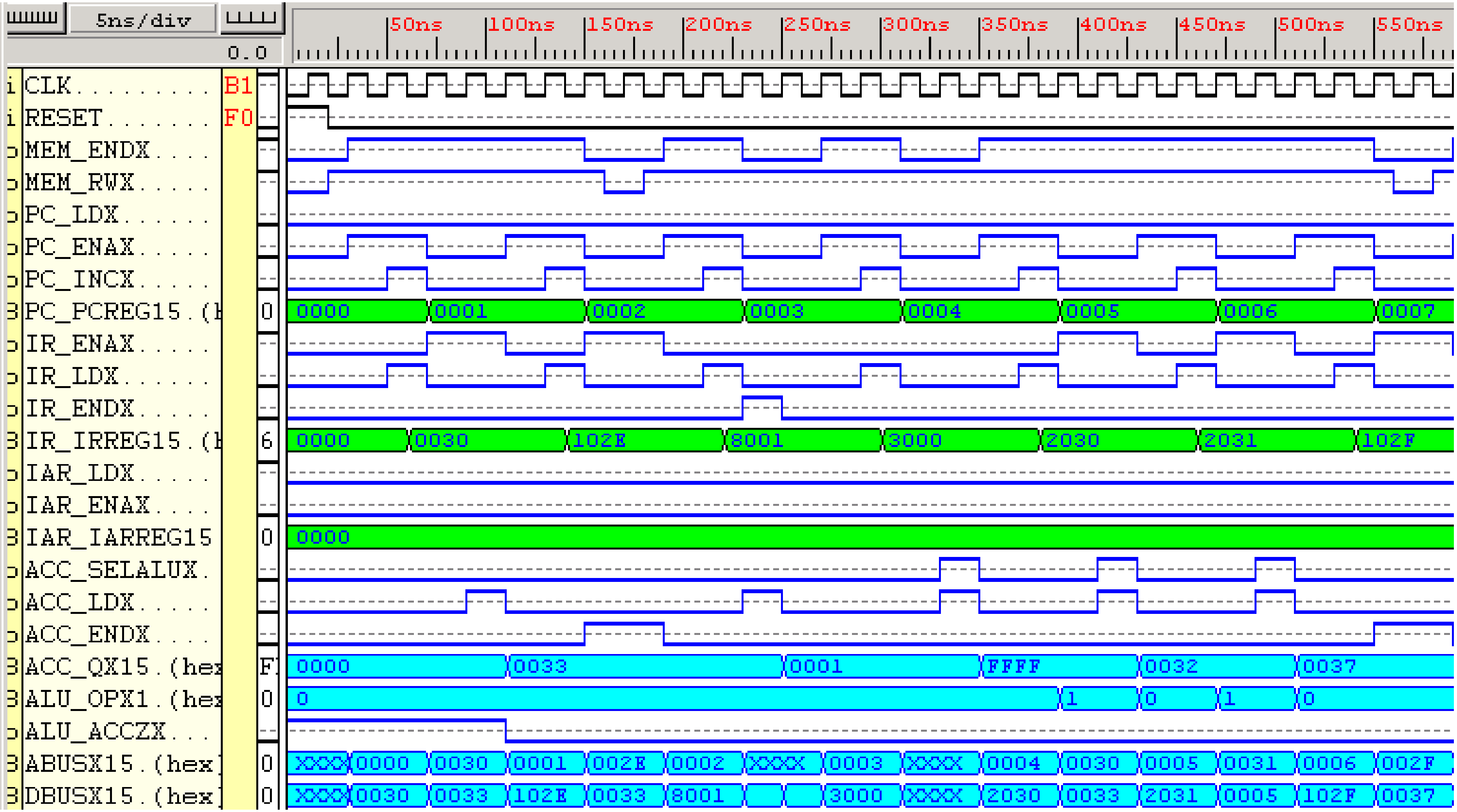
```
begin
  process(clk) begin
  if clk'event and clk = '1' then
         if reset = '1' then state <= reset_state;
           else
                 case state is
                 when reset_state => state <= fetch0;
                 when fetch0 => state <= fetch1;
                 when fetch1 =>
                          if ir_load = '1' then state <= load0;
                          elsif ir_store   = '1' then state <= store0;
                          elsif ir_add     = '1' then state <= add0;
                          elsif ir_neg     = '1' then state <= negate0;
                          elsif ir_halt    = '1' then state <= halt;
                          elsif ir_branch  = '1' then state <= branch0;
                          elsif ir_cbranch = '1' then state <= cbranch0;
                          elsif ir_iload   = '1' then state <= iload0;
                          elsif ir_istore  = '1' then state <= istore0;
                          elsif ir_mload   = '1' then state <= mload0;
                          elsif ir_madd    = '1' then state <= madd0;
                          end if;
                 when load0 =>    state <= load1;
                 when load1 =>    state <= fetch0;
                  -- yada yada yada
         end if;
  end if;
end process;
```

Controller

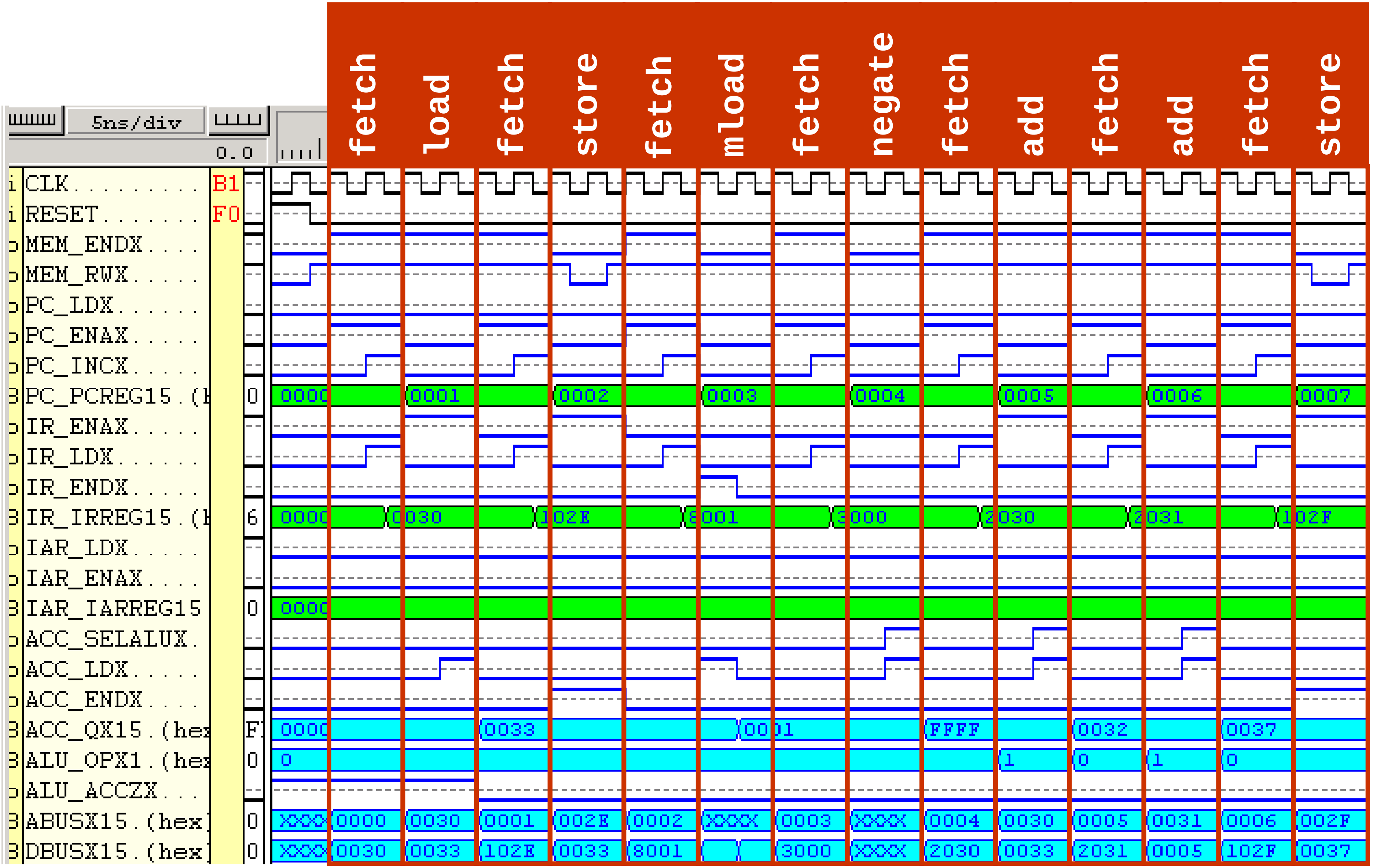
```
  process begin -- special process for memory write timing
         wait until clk = '0';
         if state = store0 or state = istore2 then
                 mem_rw <= '0';
         else
                 mem_rw <= '1';
         end if;
  end process;

  mem_enD <= '1' when
                 state =  fetch0 or state =  fetch1 or
                 state =   load0 or state =   load1 or
                 state =    add0 or state =    add1 or
                 state =  iload0 or state =  iload1 or
                 state =  iload2 or state =  iload3 or
                 state = istore0 or state = istore1
            else '0';
  pc_enA <= '1' when state =  fetch0 or state = fetch1
                else '0';
  pc_ld <= '1'  when state = branch0 or (state = cbranch0 and alu_accZ = '1')
                else '0';
  pc_inc <= '1' when state = fetch1
                else '0';
  -- yada yada yada
end controllerArch;
```

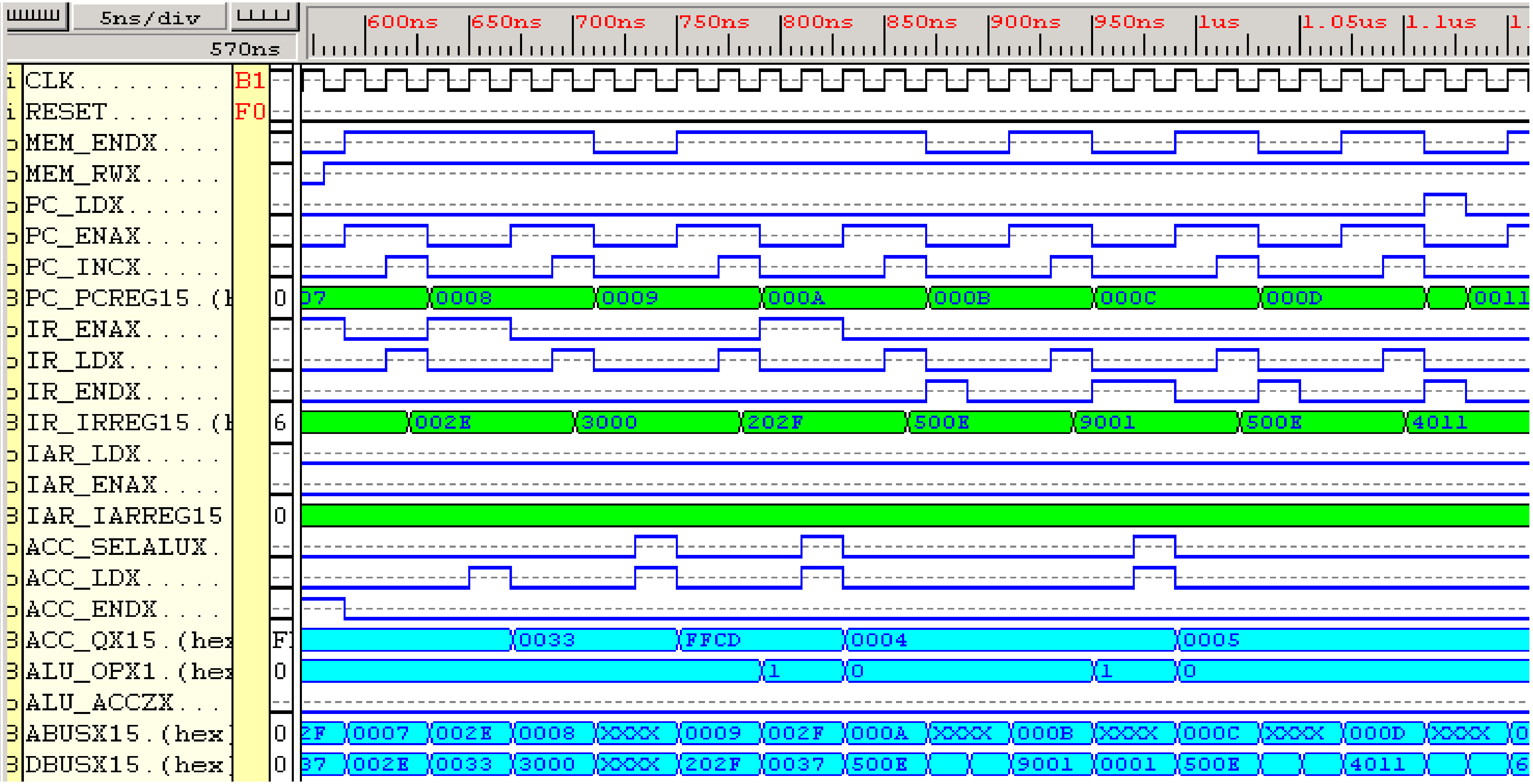
Processor Simulation



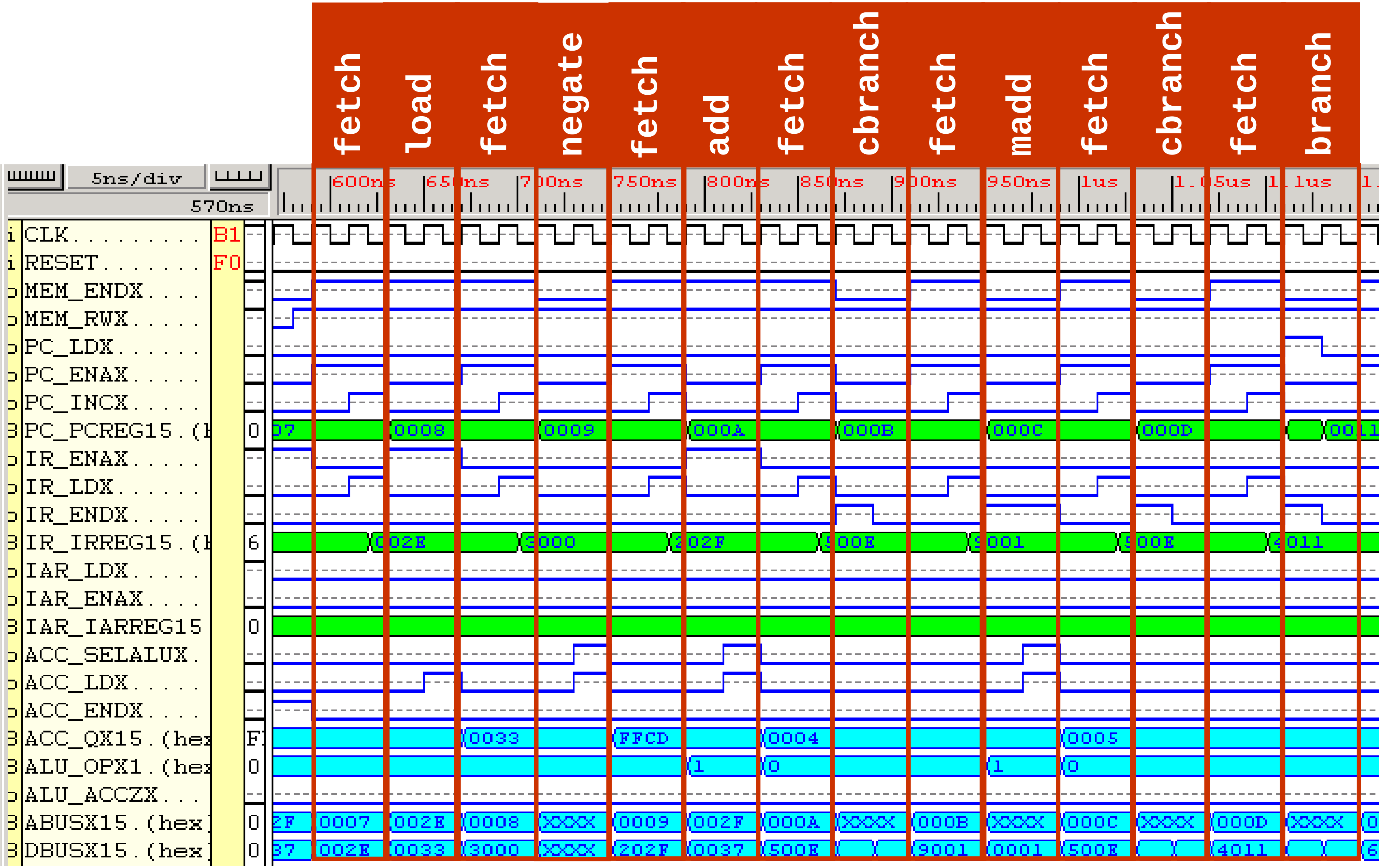
Processor Simulation



Processor Simulation



Processor Simulation



fetch load fetch negate fetch add fetch cbranch fetch madd fetch cbranch fetch branch