Implementace procesoru ve VHDL

INP - cvičení 2

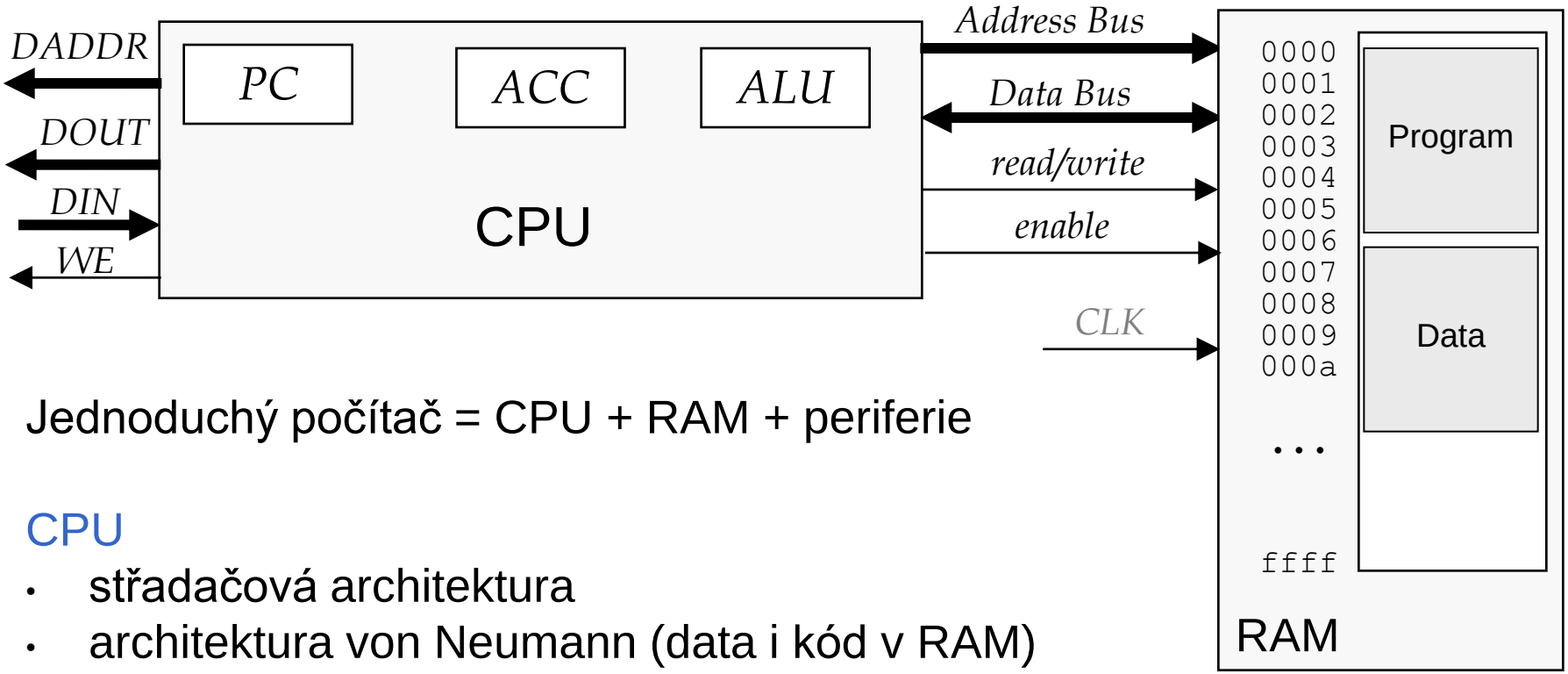
Zdeněk Vašíček, 2013
vasicek@fit.vutbr.cz

Implementace procesoru ve VHDL

1. Volba architektury
   - registrová, střadačová, zásobníková
   - Harward, Von Neumann
   - specifikace datové šířky
2. Návrh instrukční sady s ohledem na HW implementaci
3. Návrh blokového schema datové cesty
4. Přepis blokového schema do VHDL, implementace konečného automatu dle požadavků instrukční sady
   - v případě jednodušších procesorů obvykle jedna komponenta, jeden process

Realizace jednoduchého počítače

Zadání



Jednoduchý počítač = CPU + RAM + periferie

CPU

- střadačová architektura
- architektura von Neumann (data i kód v RAM)
- vstupně-výstupní 16-bitové rozhraní

RAM

- synchronní paměť s organizací N x 16 bitů
- komunikace přes sběrnici (šetření zdroji)

Sada instrukcí

Zadání

Operační znak	Instrukce	Popis
0000	halt	zastav provádění programu
0001	negate	vytvoř dvojkový doplněk z ACC
0002	accdec	zvyš hodnotu ACC o jedna
0003	accinc	sniž hodnotu ACC o jedna
000F	nop	prázdná operace
01xx	outp	zapiš hodnotu ACC na port s adresou **xx**
02xx	inp	načti do ACC hodnotu z portu s adresou **xx**
1xxx	mload	nahrej do ACC hodnotu **xxx**
2xxx	dload	nahrej do ACC hodnotu z adresy **xxx**
3xxx	iload	nahrej do ACC hodnotu, která je uložena na adrese, kterou definuje obsah buňky **xxx**
4xxx	dstore	ulož hodnotu z ACC na adresu **xxx**
5xxx	istore	ulož hodnotu z ACC na adresu, která je určena hodnotou paměťové buňky s adresou **xxx**

Sada instrukcí

Zadání

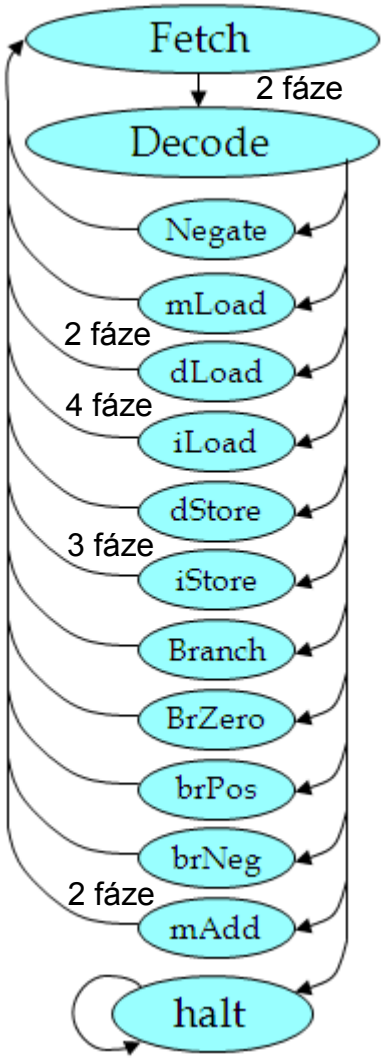
Operační znak	Instrukce	Popis
6xxx	branch	změň PC na **xxx**
7xxx	brzero	změň PC na **xxx** jestliže ACC = 0
8xxx	brpos	změň PC na **xxx** jestliže ACC > 0
9xxx	brneg	změň PC na **xxx** jestliže ACC < 0
Axxx	madd	přičti k ACC obsah paměťové buňky na adrese **xxx**
Fxxx	ijump	nepřímý skok na adresu uloženou na adrese **xxx**

Pomocí této poměrně strohé instrukční sady lze implementovat libovolný algoritmus.

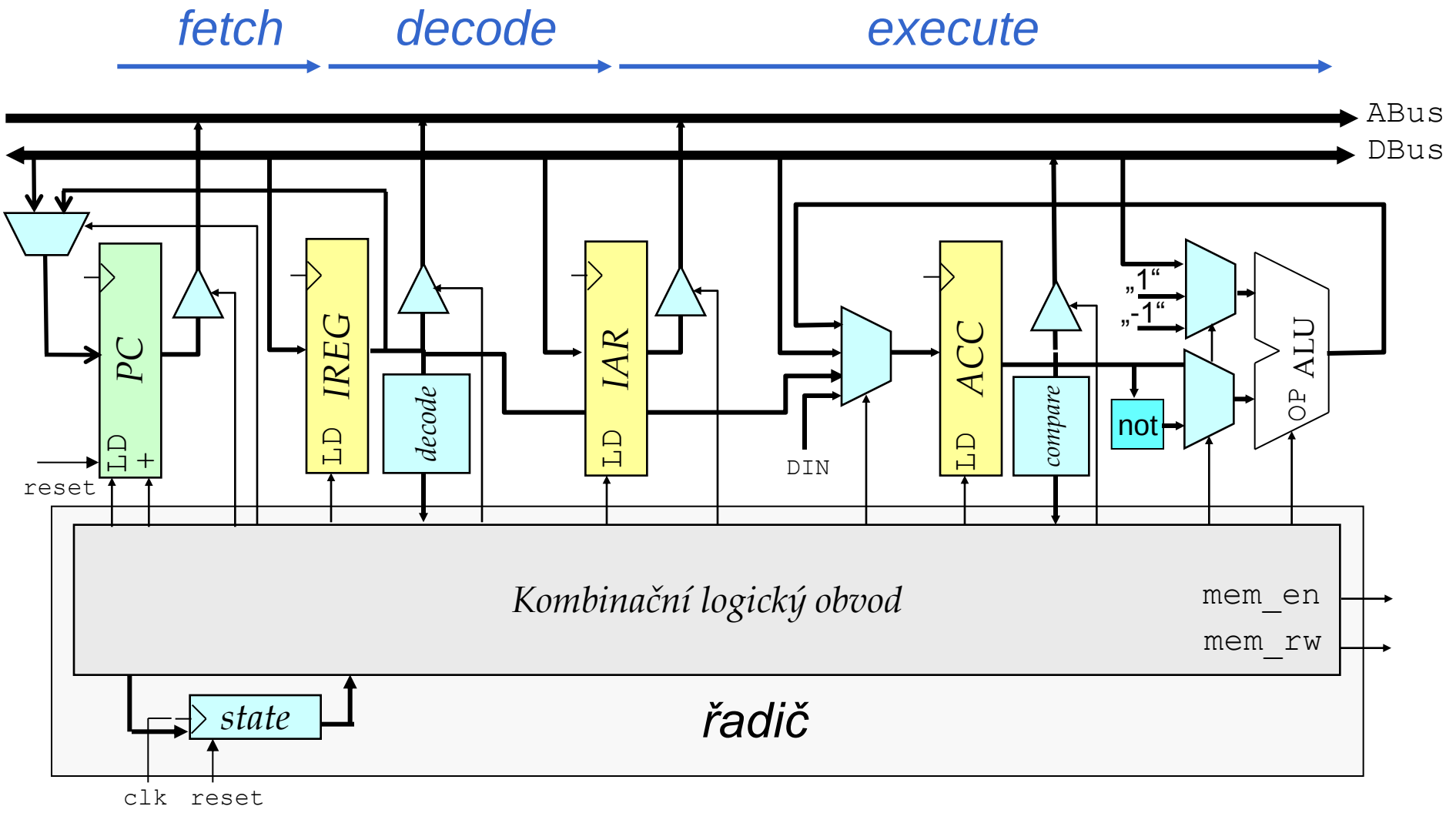
Instrukční cyklus - FSM

Procesor neustále vykonává tyto operace

- Načtení instrukce (Instruction fetch)
  - *PC* je použit pro čtení slova z paměti
  - *PC* je inkrementován, zápis slova do instrukčního registru
- Dekódování instrukce (Instruction decode)
  - podle nejvyšších 4,8,12,16 bitů se určí, co se bude dělat
  - aktivují se příslušné obvody
- Provedení instrukce (Instruction execution)
  - načtení dalších potřebných slov
  - zápis do paměti
  - modifikace *PC, ACC* apod.
  - může trvat různý počet taktů dle typu instrukce

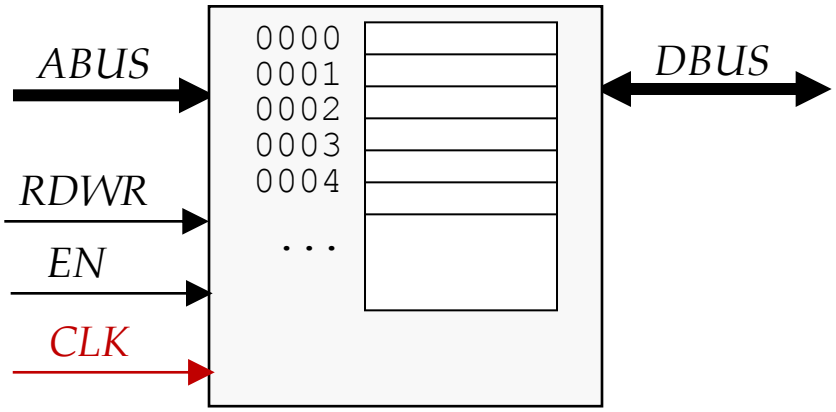


Architektura procesoru a jeho činnost



RAM

- RAM (*random access memory)*
  - když *EN =1 a RDWR =1*, *DBUS* obsahuje hodnotu uloženou na pozici, kterou definuje *ABUS*
  - když *EN =1 a RDWR = 0*, hodnota na *DBUS* přepíše data uložená na adrese vystavené na *ABUS*
  - jinak je na DBUS stav vysoké impedance



- Pro implementaci budeme uvažovat synchronní paměť RAM, protože lze na rozdíl od asynchronní varianty efektivně implementovat v FPGA

- VHDL implementace paměti RAM
  - efektivní implementace: nesmí mít RESET, musí být synchronní
  - možnosti zápisu: process (signal, shared variable) nebo strukturní popis využívající vestavěné blokové synchronní paměti BRAM

VHDL model synchronní paměti RAM

```
entity ram is
   port (
      CLK, EN, RDWR: in STD_LOGIC;
      ABUS: in STD_LOGIC_VECTOR(15 downto 0);
      DBUS: inout STD_LOGIC_VECTOR(15 downto 0)
   );
end ram;
```

```
 type t_ram is array (0 to 2**10-1) of std_logic_vector (15 downto 0);
 signal ram: t_ram := (x"0201",x"0003", x"0101", others=>x"0000");

 DBUS <= dout when EN ='1' else (others => 'Z');

 process (CLK)
 begin
    if (CLK'event) and (CLK = '1') then
       if (EN = '1') then
          if (RDWR = '0') then
             ram(conv_integer(ABUS(9 downto 0))) <= DBUS;
          end if;
          dout <= ram(conv_integer(ABUS(9 downto 0)));
       end if;
    end if;
 end process;
```

Paměť jako pole 16b slov

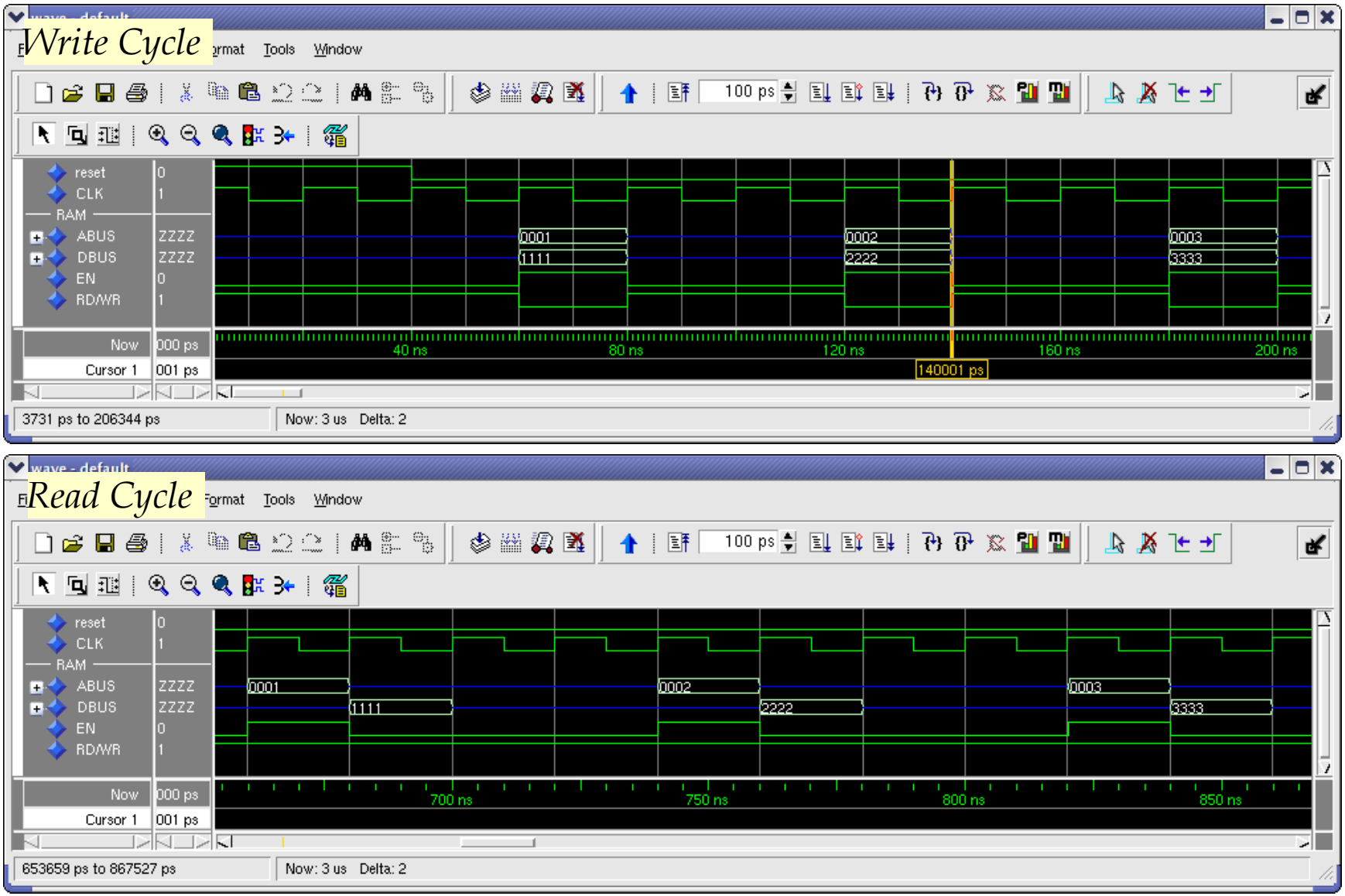
Inicializace obsahu paměti (není nutné)

Zápis na pozici určenou adresou

čtení z pozice určené adresou

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

Simulace a časování synchronní RAM



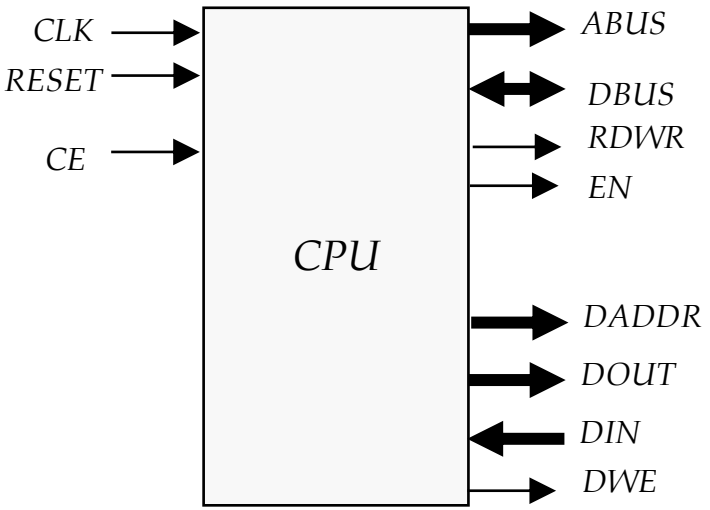
Model CPU ve VHDL

Entita

```
entity cpu is
 port (
   RESET : in std_logic;
   CLK   : in std_logic;
   CE    : in std_logic;

   -- BUS
   ABUS  : out std_logic_vector(15 downto 0);
   DBUS  : inout std_logic_vector(15 downto 0);

   EN    : out std_logic;
   RDWR  : out std_logic

   -- I/O port
   DADDR : out std_logic_vector(7 downto 0);
   DOUT  : out std_logic_vector(15 downto 0);
   DIN   : in  std_logic_vector(15 downto 0);
   DWE   : out std_logic;
 );
end cpu;
```

Asynchronní nulování

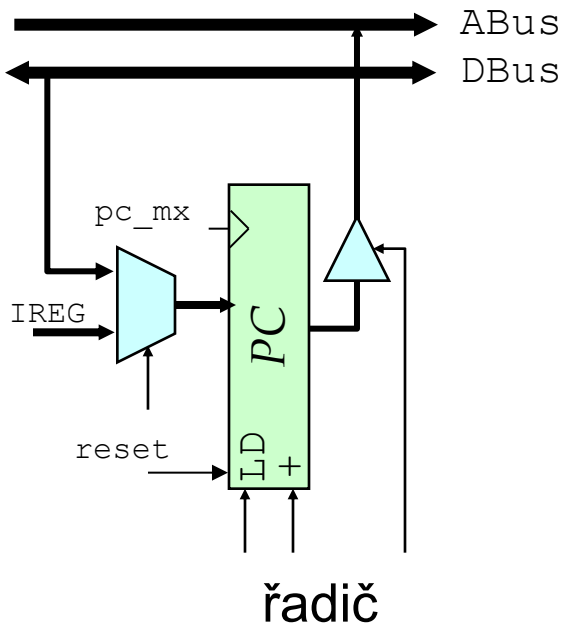
Povolení činnosti (chip enable)



Implementace datové cesty

Programový čítač (PC)

```
signal pc_reg : std_logic_vector(15 downto 0);
signal pc_ld  : std_logic;
signal pc_inc : std_logic;

-- Program counter PC
pc_cntr: process (RESET, CLK)
begin
   if (RESET='1') then
      pc_reg <= (others=>'0');
   elsif (CLK'event) and (CLK='1') then
      if (pc_ld='1') then
         pc_reg <= pc_mx;
      elsif (pc_inc='1') then
         pc_reg <= pc_reg + 1;
      end if;
   end if;
end process;

pc_mx <= "0000" & ireg_reg(11 downto 0) when
                  pc_mx_sel="00" else DBUS

-- Tristate driver
ABUS <= pc_reg when (pc_abus = '1')
               else (others => 'Z');
```

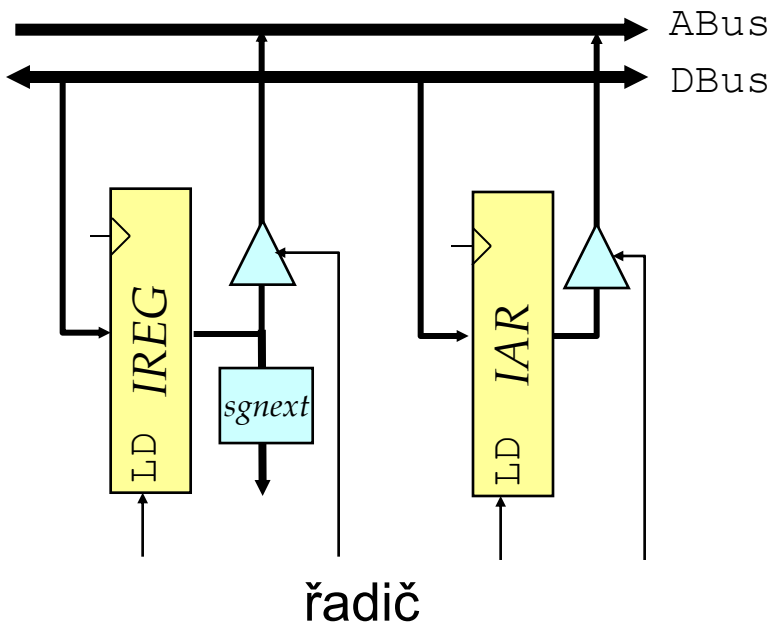


Hodnotu čítače lze

- inkrementovat
- přepsat hodnotou na DBUS (instrukce ijump)
- přepsat hodnotou v IREG (skoky ne/podmíněné)

Instrukční a adresový registr (IREG, IAR)

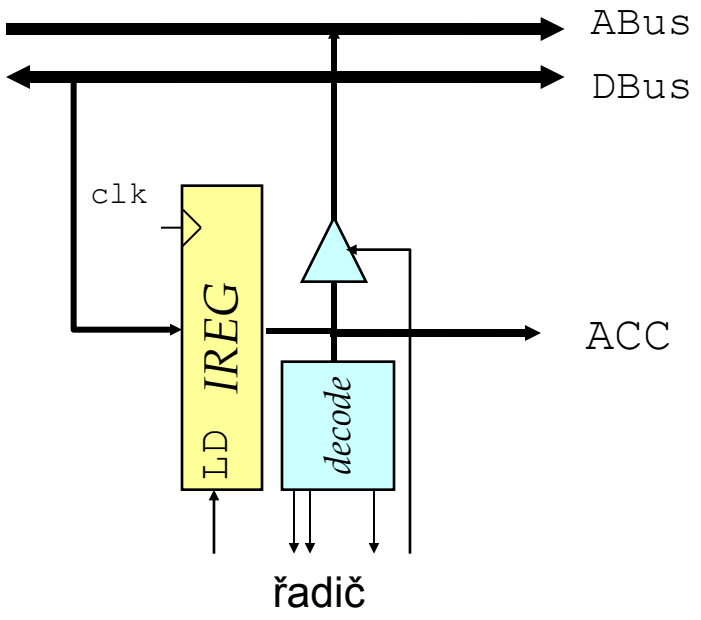
```
signal ireg_reg : std_logic_vector(15 downto 0);
signal ireg_ld  : std_logic;

-- Instruction register IREG
ireg: process (RESET, CLK)
begin
  if (RESET='1') then
     ireg_reg <= (others=>'0');
  elsif (CLK'event) and (CLK='1') then
     if (ireg_ld='1') then
        ireg_reg <= DBUS;
     end if;
  end if;
end process;

-- Indirect address register IAR
iar: process (RESET, CLK)
begin
   if (RESET='1') then
      iar_reg <= (others=>'0');
   elsif (CLK'event) and (CLK='1') then
      if (iar_ld='1') then
         iar_reg <= DBUS;
      end if;
   end if;
end process;
```




```
-- Tristate driver
ABUS <= "0000" & ireg_reg(11 downto 0)
        when (ireg_abus = '1')
        else (others=>'Z');

ABUS <= iar_reg when (iar_abus = '1')
                else (others=>'Z');

--IREG sign extension
ireg_sgnext <= "1111" when ireg_reg(11)='1'
               else "0000";
```

Instrukční dekodér

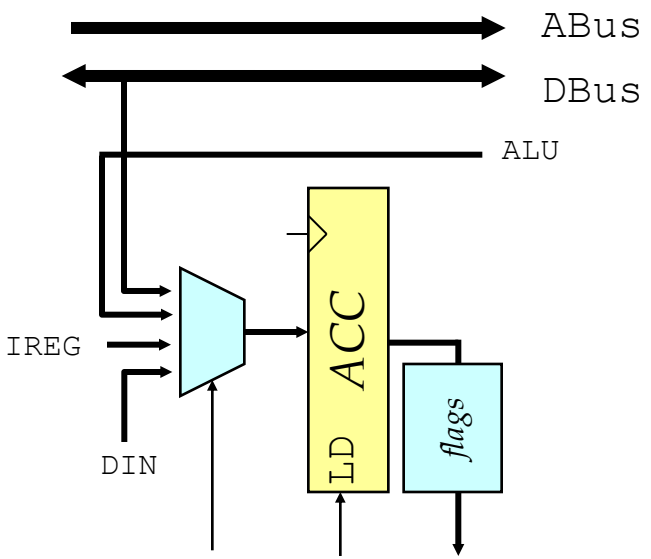
```
type inst_type is (halt, negate, mload, dload, iload,
   dstore, istore, branch, brzero, … );
signal ireg_dec  : inst_type;

--Instruction decoder
process (ireg)
begin
 case (ireg(15 downto 12)) is
   when X"0" =>
      case (ireg_reg(11 downto 8)) is
         when X"0" =>
            case (ireg_reg(3 downto 0)) is
               when X"0"   => ireg_dec <= halt;
               when X"1"   => ireg_dec <= negate;
               …
               when others => ireg_dec <= halt;
            end case;
         when X"1" => ireg_dec <= outp;
         when X"2" => ireg_dec <= inp;
         when others => ireg_dec <= halt;
      end case;
   when X"1" => ireg_dec <= mload;
   when X"2" => ireg_dec <= dload;
   …
   when others => ireg_dec <= halt;
 end case;
end process;
```



Operační znak	Instrukce
0000	halt
0001	negate
…	…
01xx	outp
02xx	inp
1xxx	mload
2xxx	dload
…	…
Fxxx	ijump

Akumulátor (ACC)

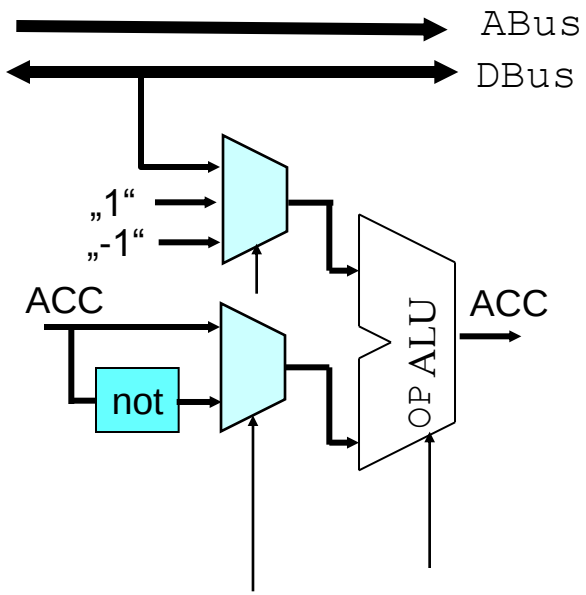
```
signal acc_reg   : std_logic_vector(15 downto 0);
signal acc_ld    : std_logic;
signal acc_mx    : std_logic_vector(15 downto 0);
signal accmx_sel : std_logic_vector(1 downto 0);
signal acc_zero  : std_logic;
signal acc_neg   : std_logic;
signal acc_pos   : std_logic;

-- ACC data multiplexor
with accmx_sel select
acc_mx <= ireg_sgnext & ireg_reg(11 downto 0)
          when "00",
          DBUS     when "01",
          DIN      when "10",
          alu_out when others;

-- Accumulator register ACC
accreg: process(RESET, CLK)
begin
   if (RESET='1') then
      acc_reg <= (others=>'0');
   elsif (CLK'event) and (CLK='1') then
      if (acc_ld='1') then
         acc_reg <= acc_mx;
      end if;
   end if;
end process;
```

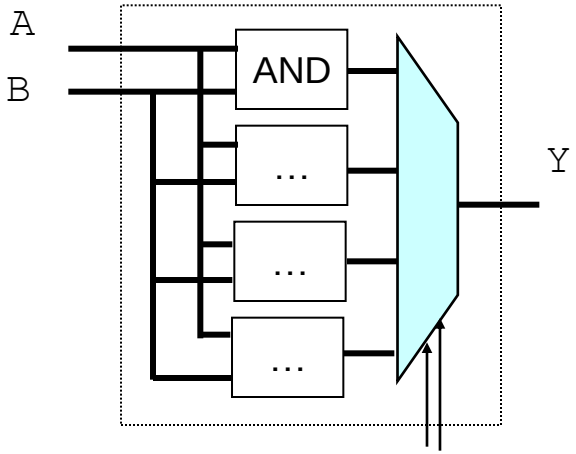



```
-- ACC flags comparator
acc_zero <= '1' when (acc_reg = X"0000")
                else '0';
acc_neg  <= '1' when (acc_reg(15) = '1')
                else '0';
acc_pos  <= '1' when (acc_reg(15) = '0')
                     and (acc_zero='0')
                else '0';
```

Aritmeticko logická jednotka (ALU)

```
type aluoper_type is (alu_add, alu_and, … );
signal alu_oper : aluoper_type;
signal alu_op0, alu_op1 : std_logic_vector(15 downto 0);
signal alu_out : std_logic_vector(15 downto 0);
signal alu_mx1_sel : std_logic_vector(1 downto 0);
signal alu_mx2_sel : std_logic;

-- Operands multiplexors
alu_op0 <= DBUS    when alu_mx1_sel="00"
           X"0001" when alu_mx1_sel="01"
                   else X"1111";

alu_op1 <= acc_reg when alu_mx2_sel='0'
                   else (not acc_reg);

-- ALU
with alu_oper select
   alu_out <= alu_op0  +  alu_op1 when alu_add,
              alu_op0 and alu_op1 when alu_and,
              …
              alu_op1 when others;
```





Navržená ALU je schopna

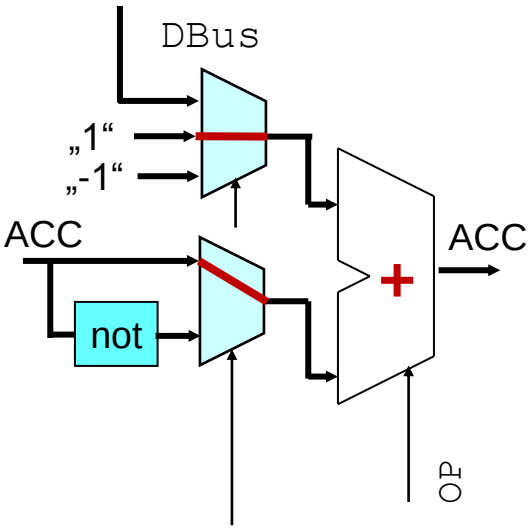
- inkrementovat a dekrementovat obsah ACC
- vytvořit jedničkový (negace) a dvojkový doplněk k ACC
- provádět operace: add, sub, and, … s ACC a hodnotou na DBUS

Přepsání ACC hodnotou na DBUS je implementováno v ACC

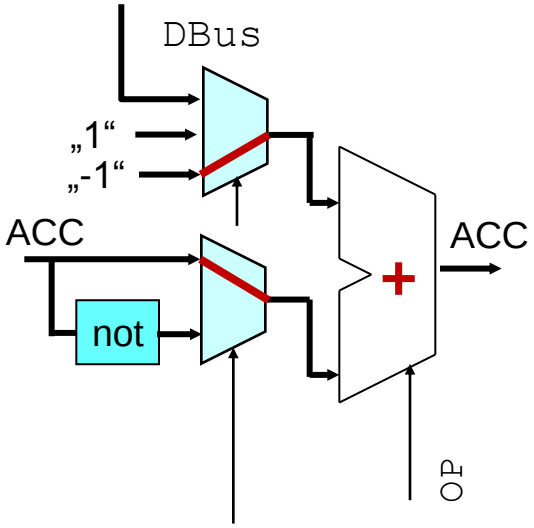
Aritmeticko logická jednotka (ALU)

Podporované operace nad ACC

- Inkrementace obsahu ACC (ACC <= ACC + 1)



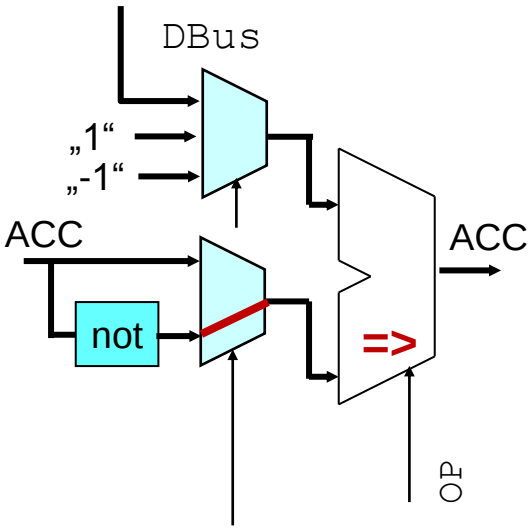
- Dekrementace obsahu ACC (ACC <= ACC - 1)



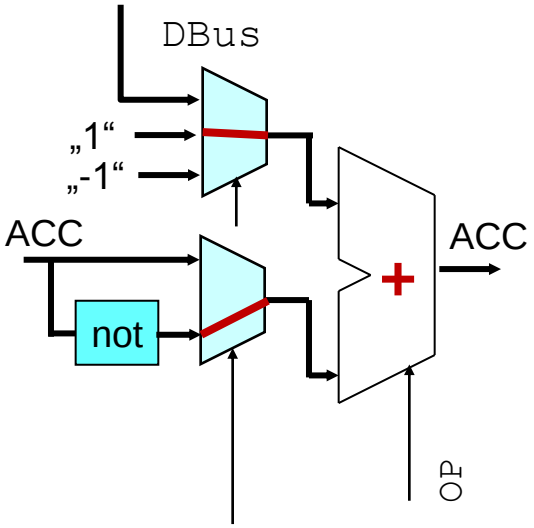
Aritmeticko logická jednotka (ALU)

Podporované operace nad ACC

- Negace ACC
  (jedničkový doplněk)
  (ACC <= not ACC)



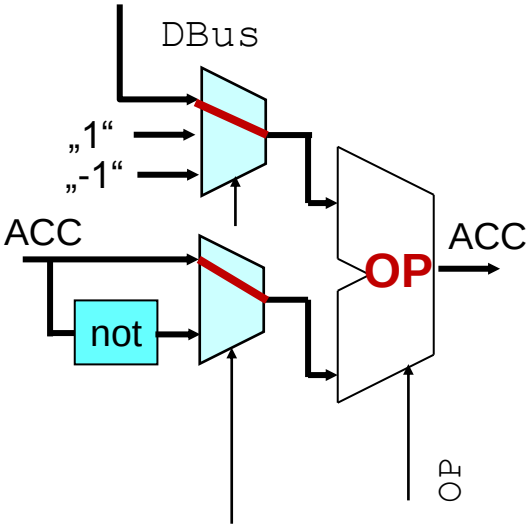
- Inverze ACC
  (dvojkový doplněk)
  (ACC <= neg ACC)



Aritmeticko logická jednotka (ALU)

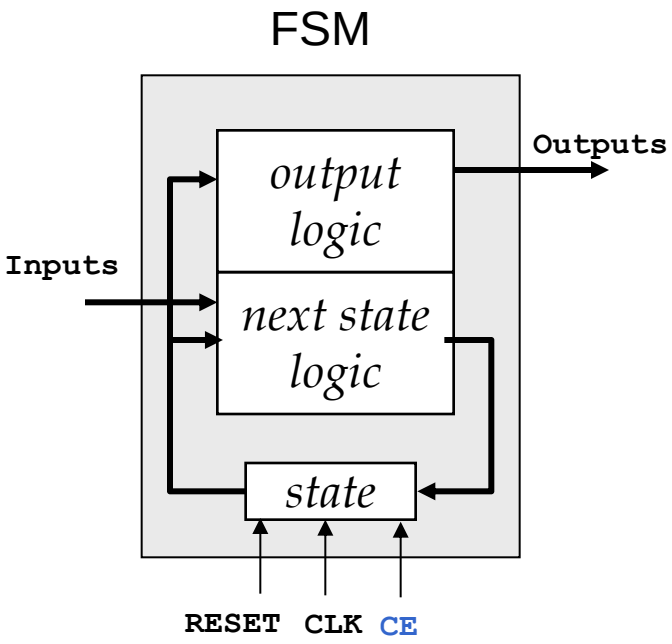
Podporované operace nad ACC

- Operace s DBUS a ACC dle možností ALU
  (ACC <= DBUS op ACC)



Implementace řídicí cesty

Řadič



deklarace signálů

```
type fsm_state is (sidle, sfetch0, sfetch1, sdecode,
   sbranch, shalt, snop, smload, snegate, smadd0,
   smadd1, sdload0, sdload1, sdstore, siload0,
   siload1, siload2, siload3, sistore0, sistore1,
   sistore2, saccdec, saccinc, sijump0, sijump1,
   soutp, sinp);

signal pstate : fsm_state;
signal nstate : fsm_state;
```

registr aktuálního stavu

```
--FSM present state
fsm_pstate: process (RESET, CLK)
   begin
      if (RESET='1') then
         pstate <= sidle;
      elsif (CLK'event) and (CLK='1') then
         if (CE = '1') then
            pstate <= nstate;
         end if;
      end if;
end process;
```

Řadič – 1/3

logika následujícího stavu a výstupní logika

```
--FSM next state logic,
--Output logic (Moore FSM)
nsl: process (pstate,
    ireg_dec, acc_zero,
    acc_pos, acc_neg)
begin
 -- INIT
 EN <= '0';
 DWE <= '0';
 RDWR <= '1';
 ireg_ld <= '0';
 ireg_abus <= '0';
 pc_inc <= '0';
 pc_ld  <= '0';
 pc_abus <= '0';
 pc_mx_sel <= "00";
 iar_ld <= '0';
 iar_abus <= '0';
 acc_mx_sel <= "00";
 acc_ld <= '0';
 alu_mx1_sel <= "01";
 alu_mx2_sel <= '0';
 alu_oper <= alu_add;
 dbus_sel <= '0';
```

```
case pstate is
-- IDLE
when sidle =>
   nstate <= sfetch0;

-- INSTRUCTION FETCH
when sfetch0 =>
   nstate <= sfetch1;
   pc_abus <= '1';
   EN <= '1';

when sfetch1 =>
   ireg_ld <= '1';
   nstate <= sdecode;

-- INSTRUCTION DECODE
when sdecode =>
  case ireg_dec is
   when halt =>
     nstate <= halt;
   when nop =>
     nstate <= snop;
   when branch =>
     nstate <= sbranch;
   when brzero =>
     if (acc_zero='1') then
         nstate <= sbranch;
     else
         nstate <= snop;
     end if;
```

```
   when brpos =>
      if (acc_pos='1') then
        nstate <= sbranch;
      else
        nstate <= snop;
      end if;
   when brneg =>
      if (acc_neg='1') then
        nstate <= sbranch;
      else
        nstate <= snop;
      end if;
   when accdec =>
      nstate <= saccdec;
   when accinc =>
      nstate <= saccinc;
   when add =>
      nstate <= sadd0;
   when dload =>
      nstate <= sdload0;
   when dstore =>
      nstate <= sdstore;
   when iload =>
      nstate <= siload0;
   when istore =>
      nstate <= sistore0;
   ...
   when others =>
      nstate <= shalt;
end case;
```

Q: proč je při vykonávání instrukce brzero, brpos, brneg přechod do stavu snop?

Řadič – 2/3

logika následujícího stavu a výstupní logika

```
-- HALT
when shalt =>
   nstate <= shalt;

-- BRANCH
when sbranch =>
   nstate <= sfetch0;
   pc_ld <= '1';

-- NOP
when snop =>
   nstate <= sfetch0;
   pc_inc <= '1';

-- LOAD IMMEDIATE
when smload =>
   nstate <= sfetch0;
   acc_mx_sel <= "00";
   acc_ld <= '1';
   pc_inc <= '1';

-- NEGATE
when snegate =>
   nstate <= sfetch0;
   acc_mx_sel <= "11";
   alu_oper <= alu_add;
   alu_mx1_sel <= "01";
   alu_mx2_sel <= '1';
   acc_ld <= '1';
   pc_inc <= '1';
```

```
-- ACC DEC
when saccdec =>
   nstate <= sfetch0;
   acc_mx_sel <= "11";
   alu_oper <= alu_add;
   alu_mx1_sel <= "11";
   alu_mx2_sel <= '0';
   acc_ld <= '1';
   pc_inc <= '1';

-- ACC INC
when saccinc =>
   nstate <= sfetch0;
   acc_mx_sel <= "11";
   alu_oper <= alu_add;
   alu_mx1_sel <= "01";
   alu_mx2_sel <= '0';
   acc_ld <= '1';
   pc_inc <= '1';

-- ADD
when smadd0 => --phase 0
   nstate <= smadd1;
   ireg_abus <= '1';
   EN <= '1';
```

```
when smadd1 => --phase 1
   nstate <= sfetch0;
   alu_oper <= alu_add;
   acc_mx_sel <= "11";
   alu_mx1_sel <= "00";
   alu_mx2_sel <= '0';
   acc_ld <= '1';
   pc_inc <= '1';

-- LOAD DIRECT
when sdload0 => --phase 0
   nstate <= sdload1;
   ireg_abus <= '1';
   EN <= '1';

when sdload1 => --phase 1
   nstate <= sfetch0;
   acc_mx_sel <= "01";
   acc_ld <= '1';
   pc_inc <= '1';

-- STORE DIRECT
when sdstore =>
   nstate <= sfetch0;
   dbus_sel <= '1';
   ireg_abus <= '1';
   EN <= '1';
   RDWR <= '0';
   pc_inc <= '1';
```

Řadič – 3/3

logika následujícího stavu a výstupní logika

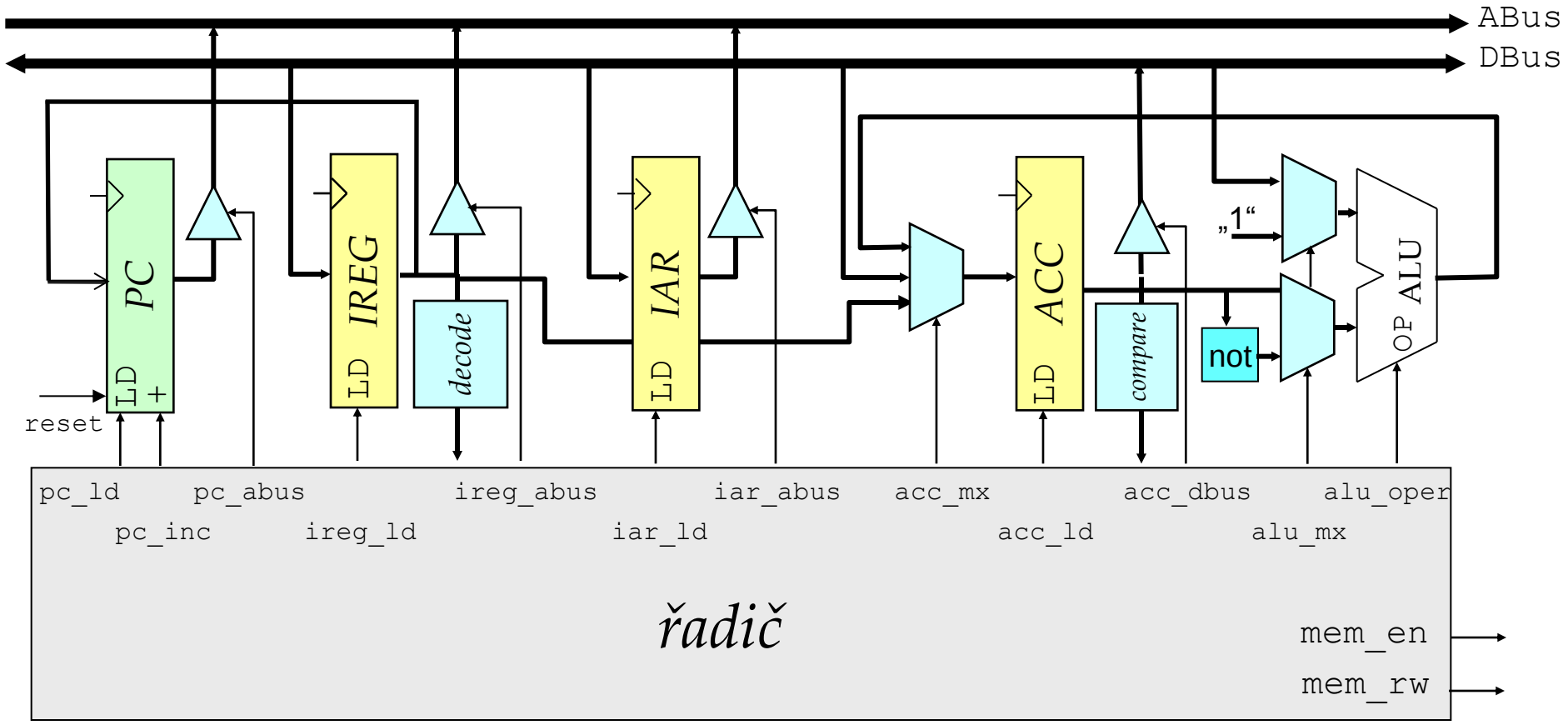
```
-- LOAD INDIRECT
when siload0 => --phase 0
   nstate <= siload1;
   ireg_abus <= '1';
   EN <= '1';

when siload1 => --phase 1
   nstate <= siload2;
   iar_ld <= '1';

when siload2 => --phase 2
   nstate <= siload3;
   iar_abus <= '1';
   EN <= '1';

when siload3 => --phase 3
   nstate <= sfetch0;
   acc_mx_sel <= "01";
   acc_ld <= '1';
   pc_inc <= '1';
```

```
-- STORE INDIRECT
when sistore0 => --phase 0
   nstate <= sistore1;
   ireg_abus <= '1';
   EN <= '1';

when sistore1 => --phase 1
   nstate <= sistore2;
   iar_ld <= '1';

when sistore2 => --phase 2
   nstate <= sfetch0;
   iar_abus <= '1';
   dbus_sel <= '1';
   EN <= '1';
   RDWR <= '0';
   pc_inc <= '1';
```

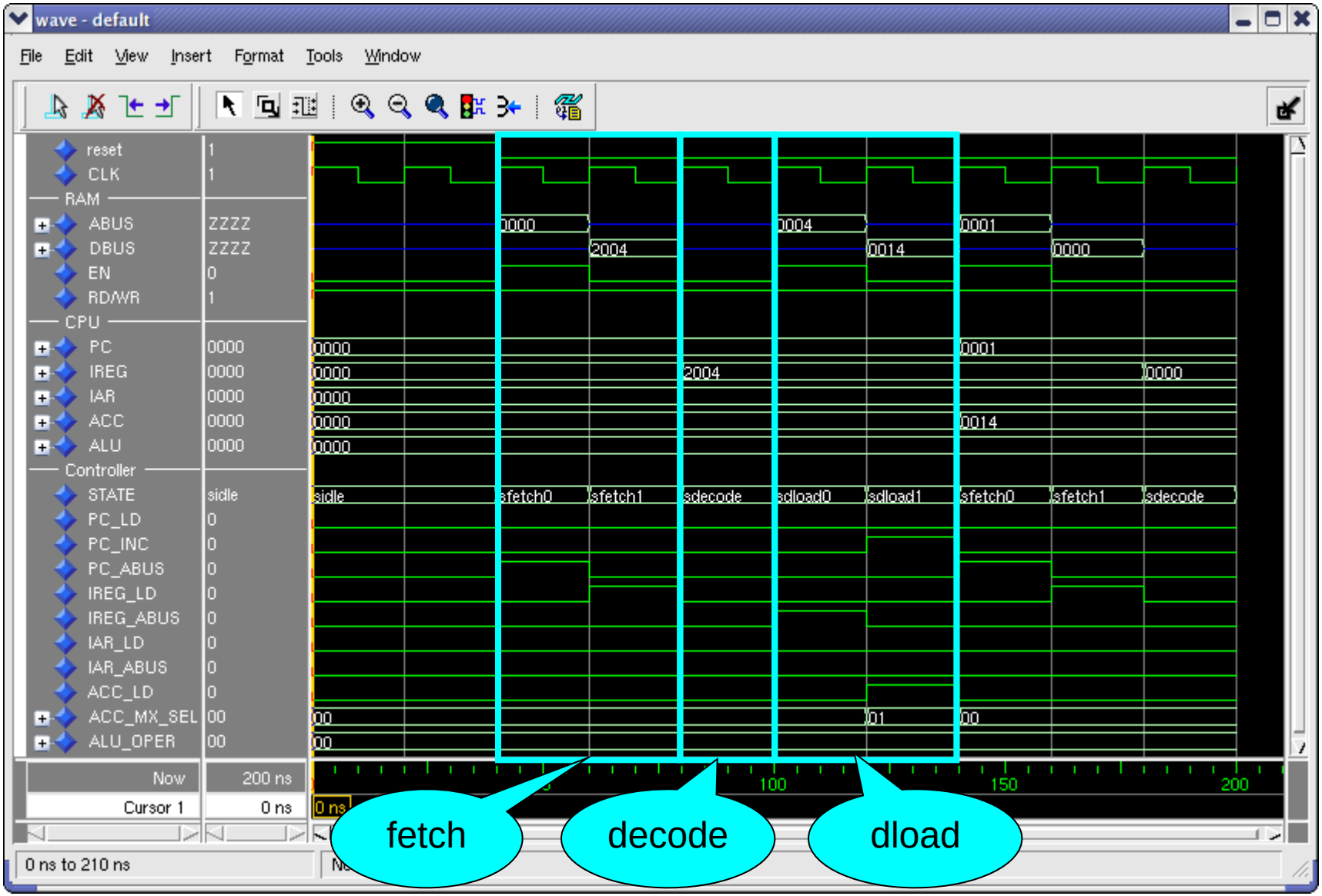
```
-- INDIRECT JUMP
when sijump0 => --phase 0
   nstate <= sijump1;
   ireg_abus <= '1';
   EN <= '1';

when sijump1 => --phase 1
   nstate <= sfetch0;
   iar_ld <= '1';
   pc_ld <= '1';
   pc_mx_sel <= "11";

-- PORT OUTPUT
when soutp =>
   nstate <= sfetch0;
   DWE <= '1';
   pc_inc <= '1';

-- PORT INPUT
when sinp =>
   nstate <= sfetch0;
   acc_mx_sel <= "10";
   acc_ld <= '1';
   pc_inc <= '1';

when others =>
   null;

end case;
end process;
```

Q: proč vykonání intrukce iload trvá 4 a istore jen 3 takty?

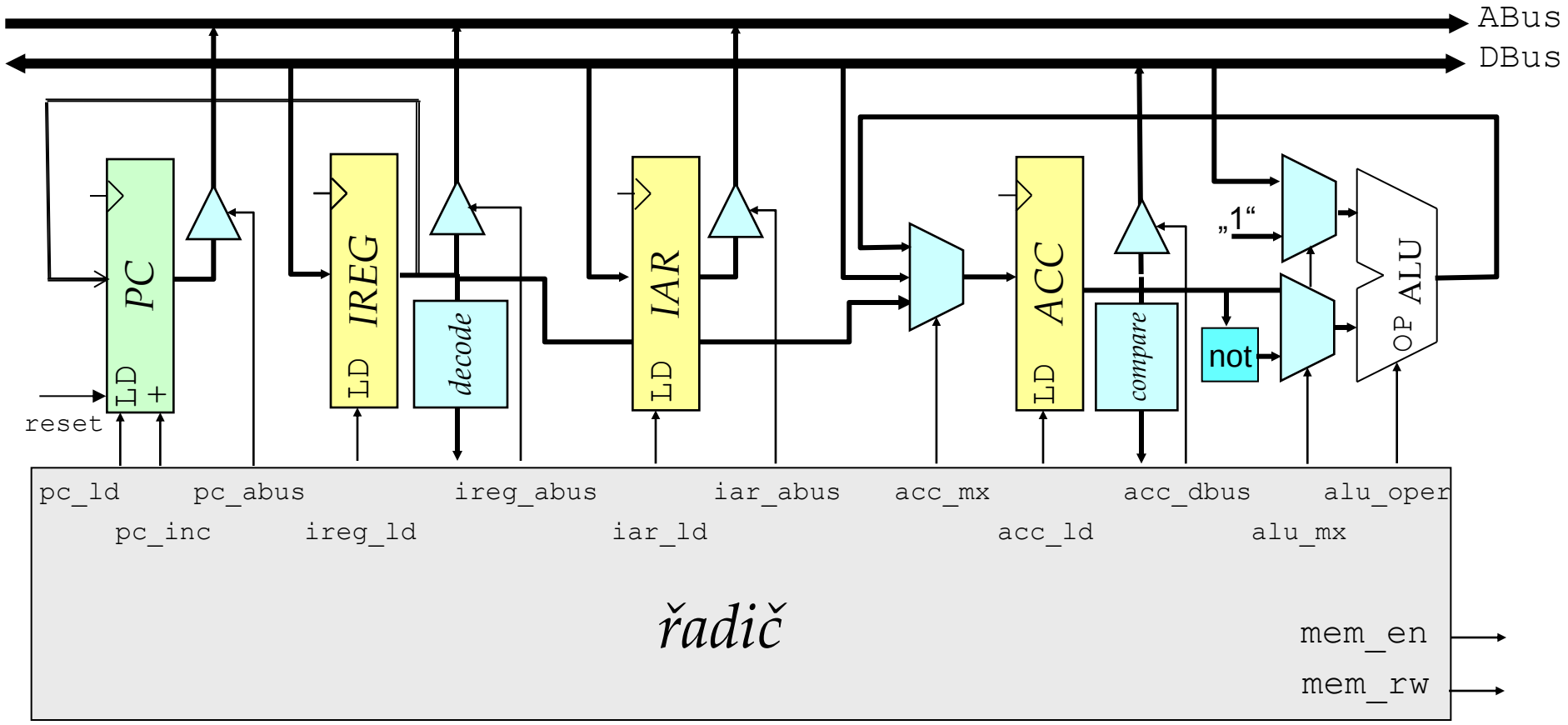
Příklad: DLOAD acc, m[04]



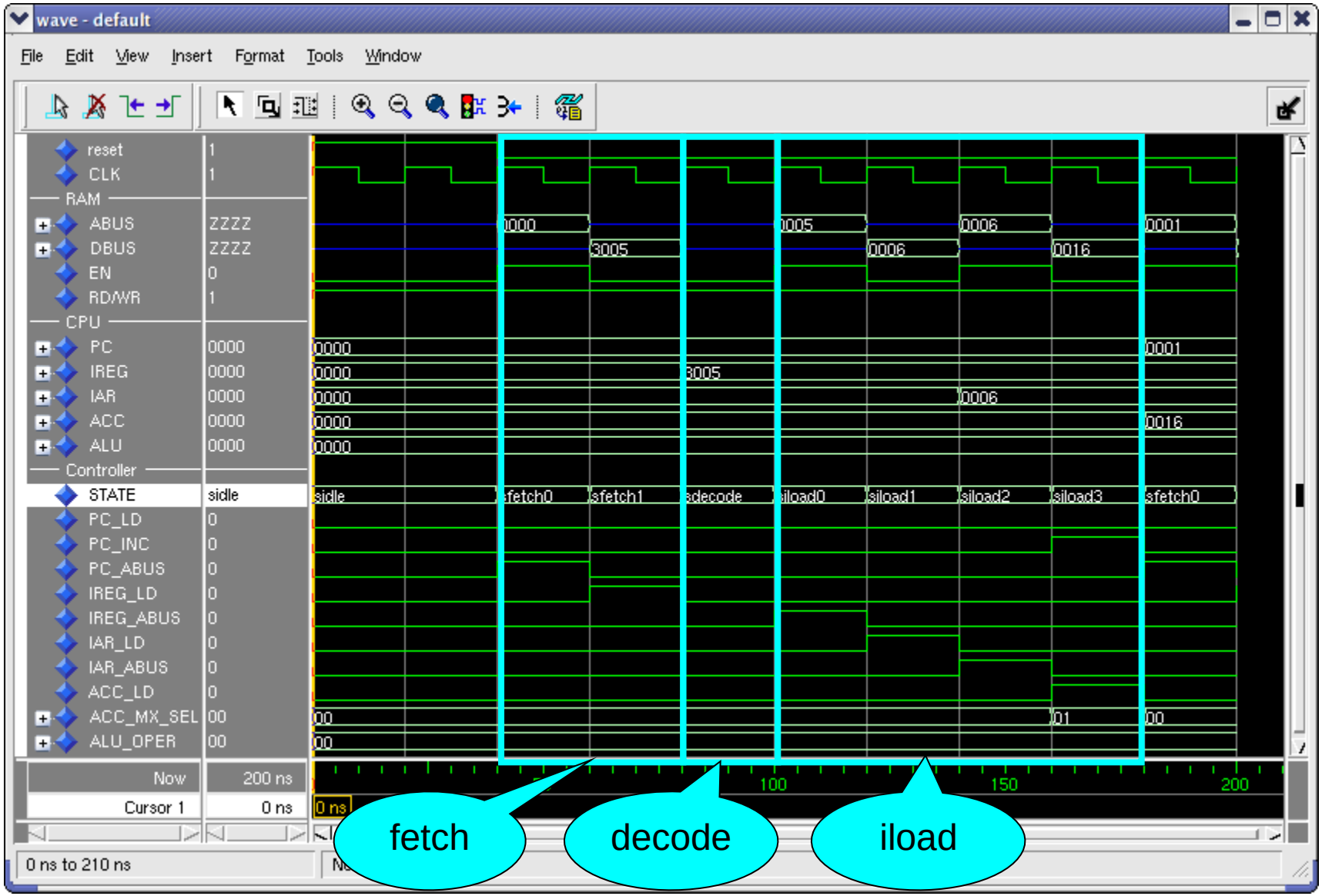
Příklad: DLOAD acc, m[04]



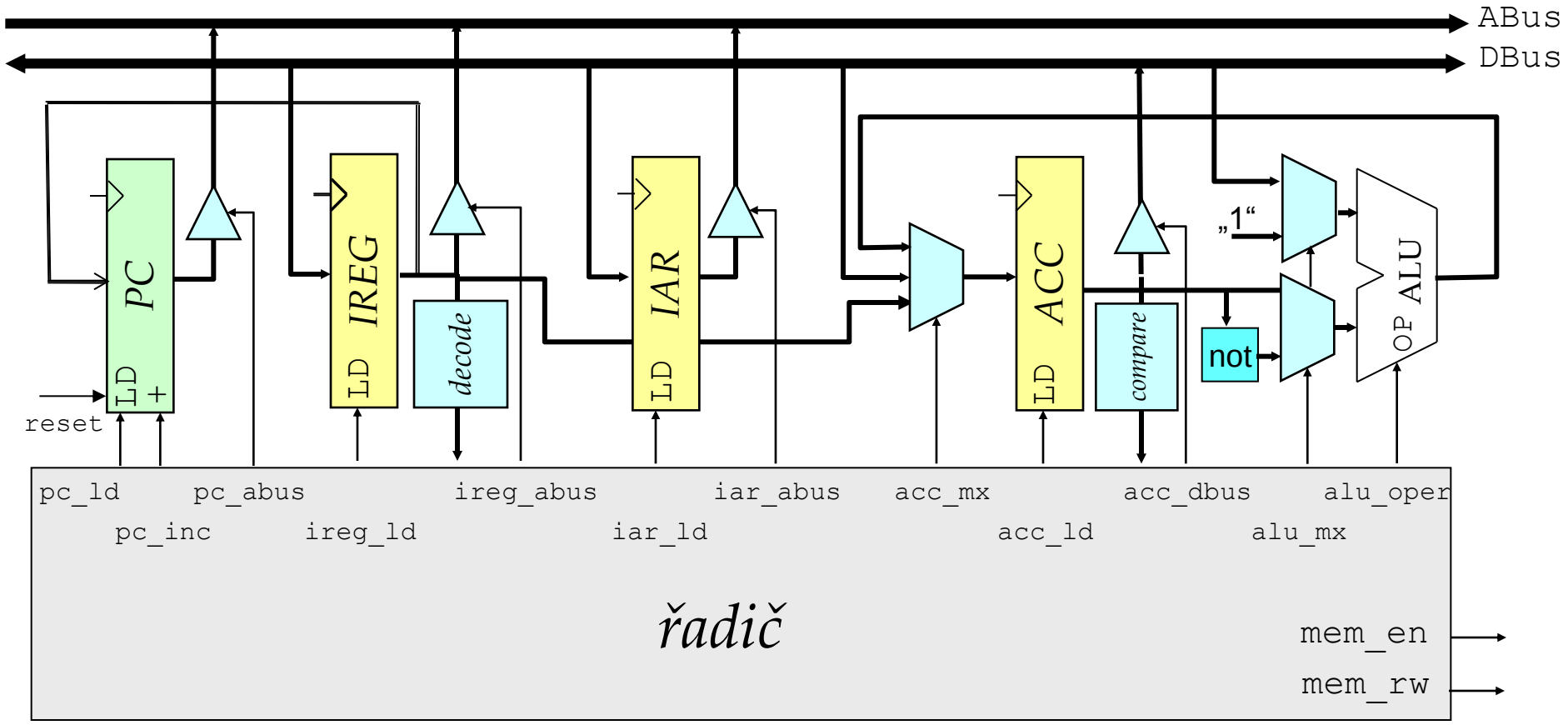
Příklad: ILOAD acc, *m[05]



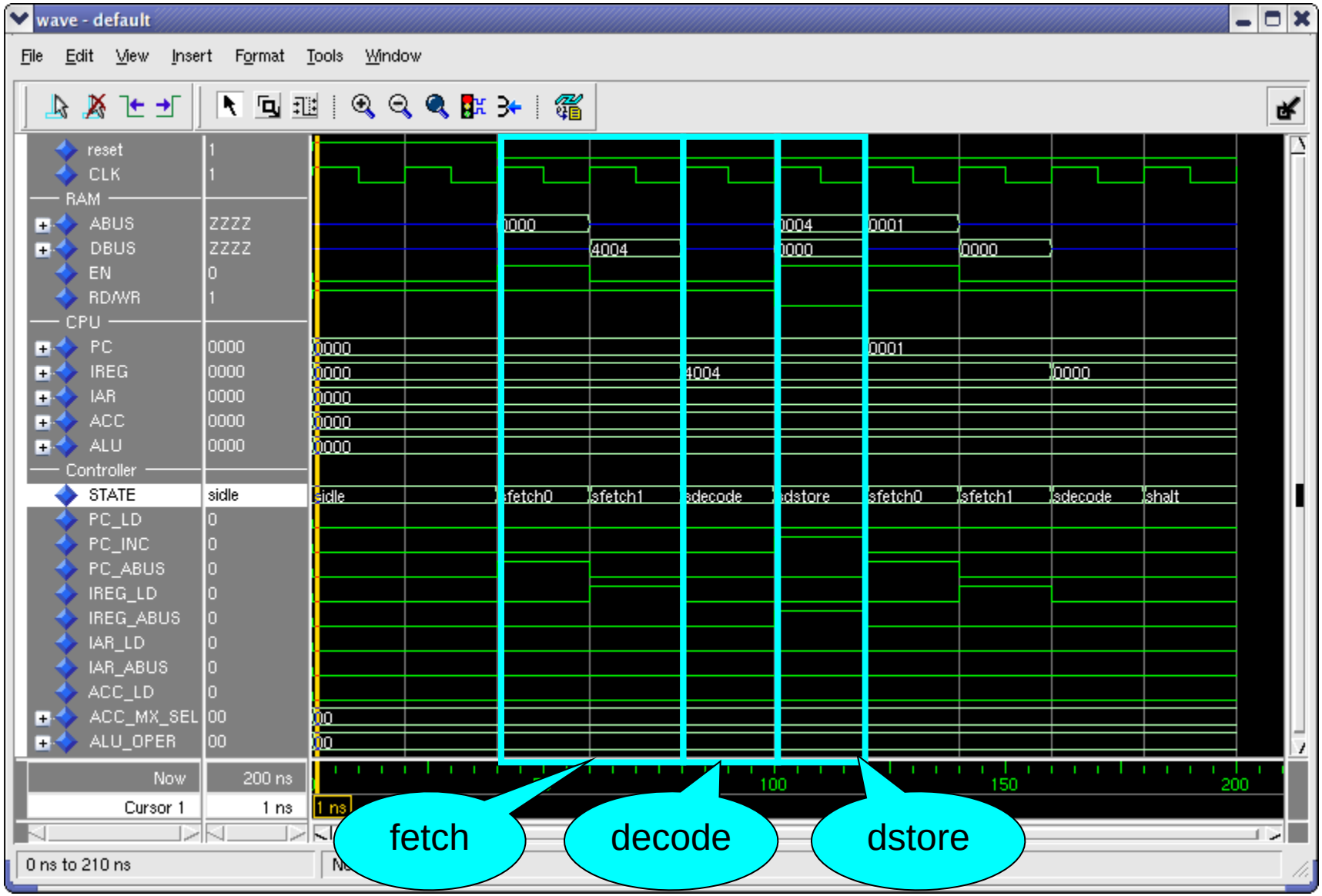
Příklad: ILOAD acc, *m[05]



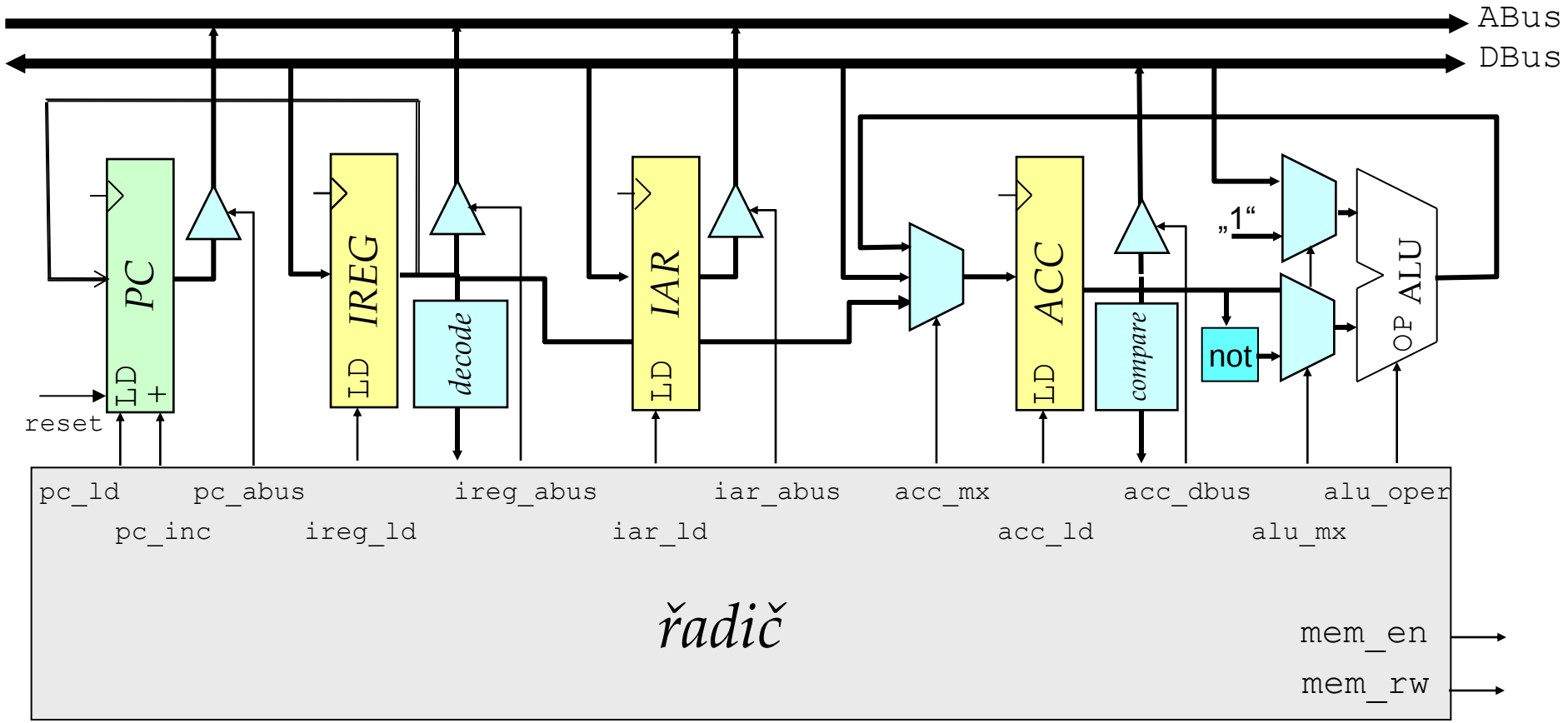
Příklad: DSTORE m[04], acc



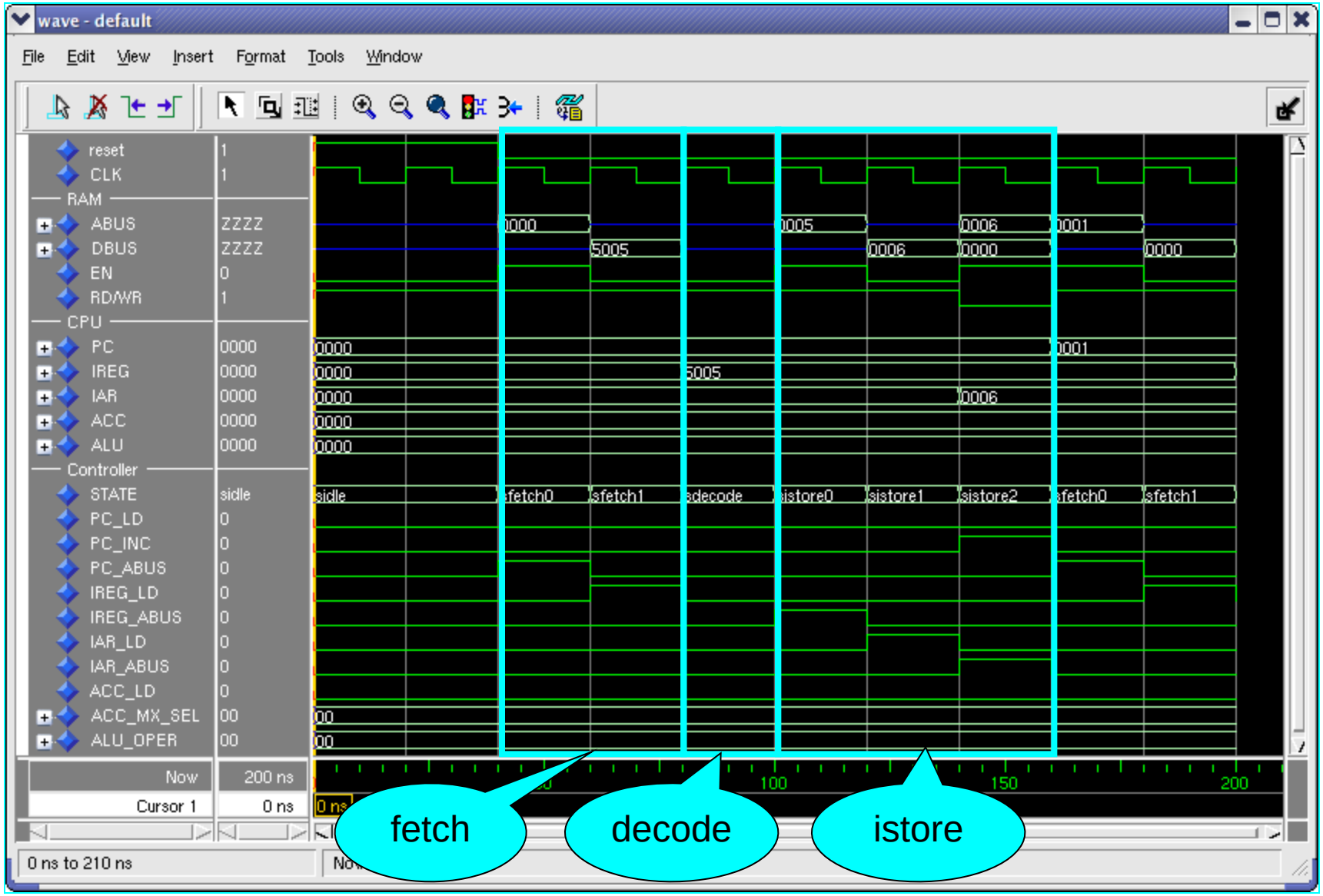
Příklad: DSTORE m[04], acc



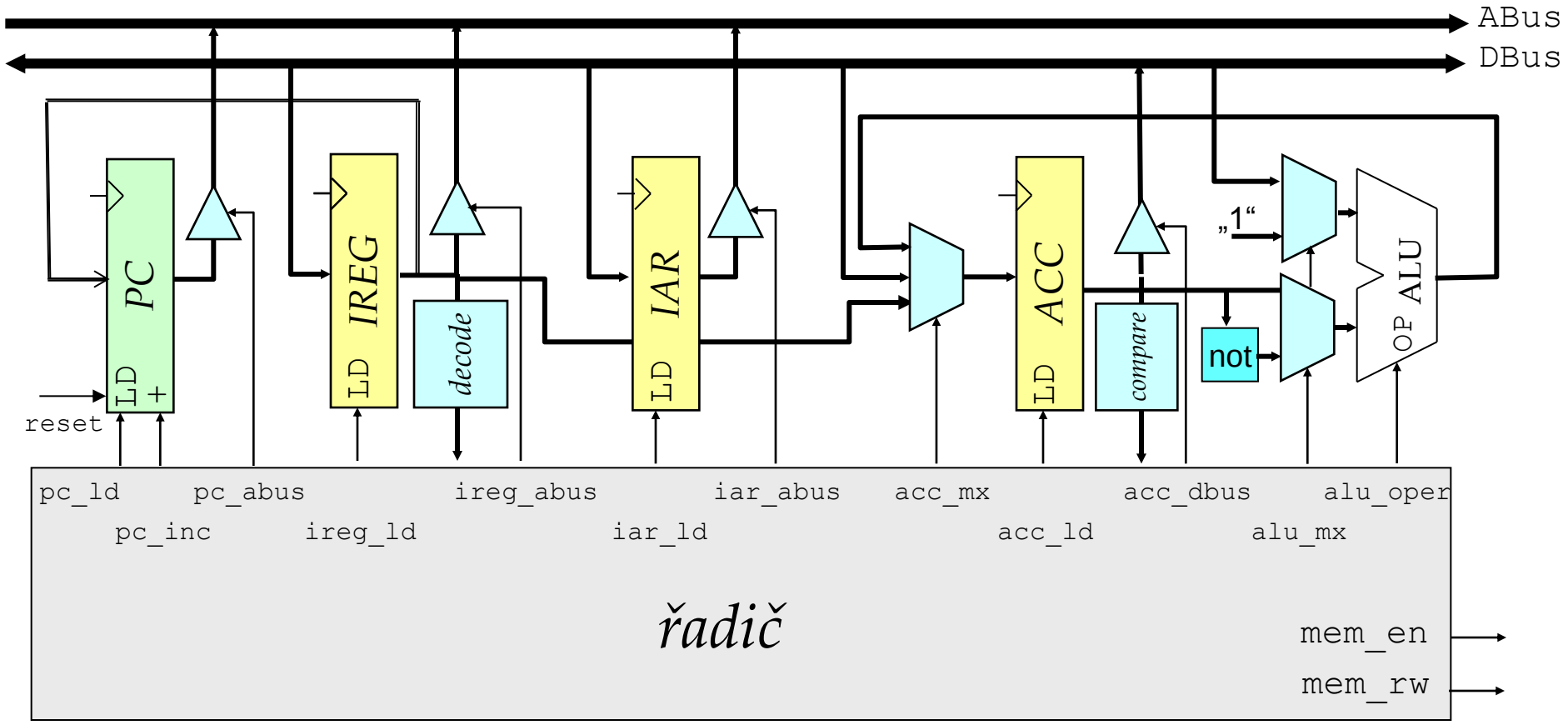
Příklad: ISTORE *m[05], acc



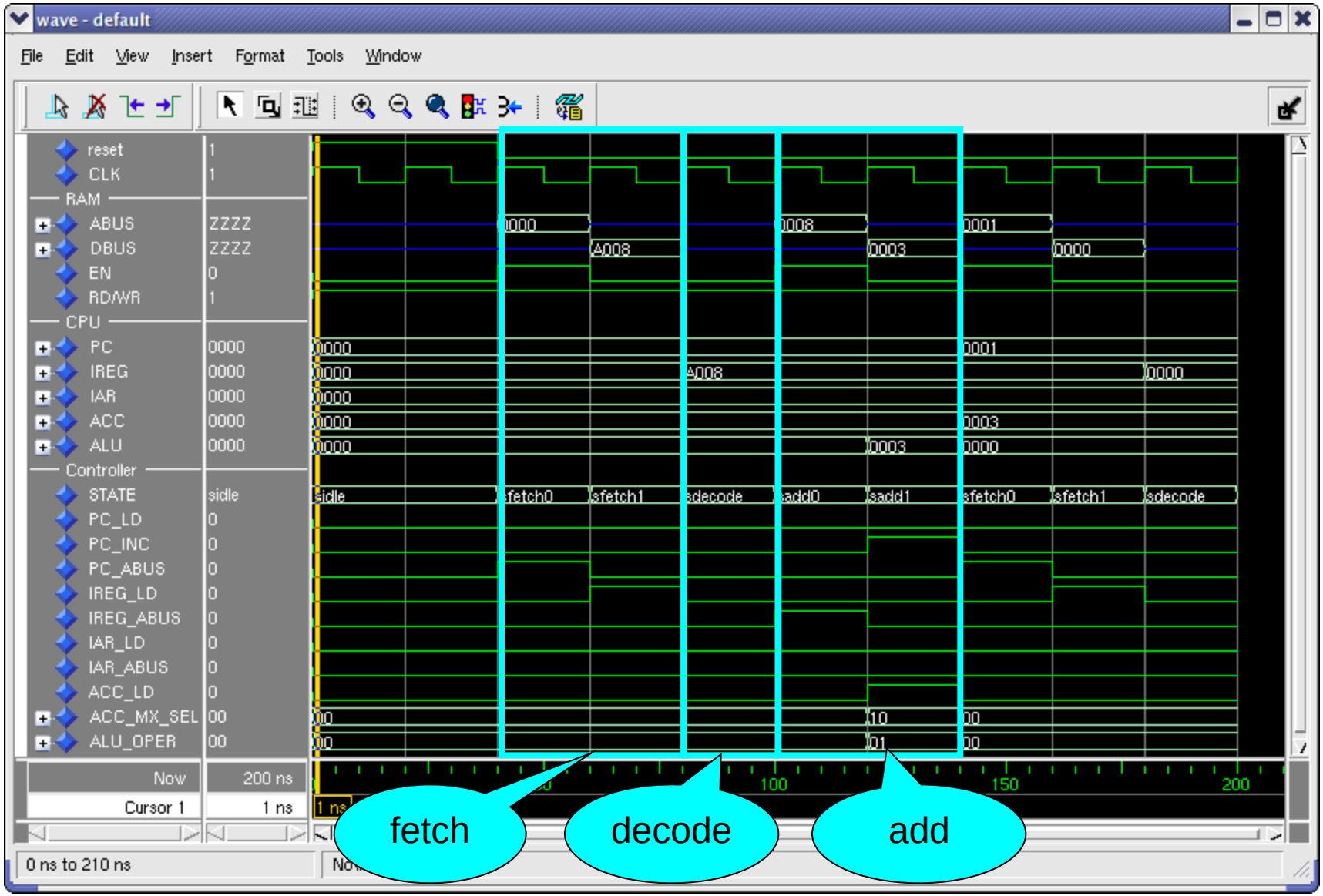
Příklad: ISTORE *m[05], acc



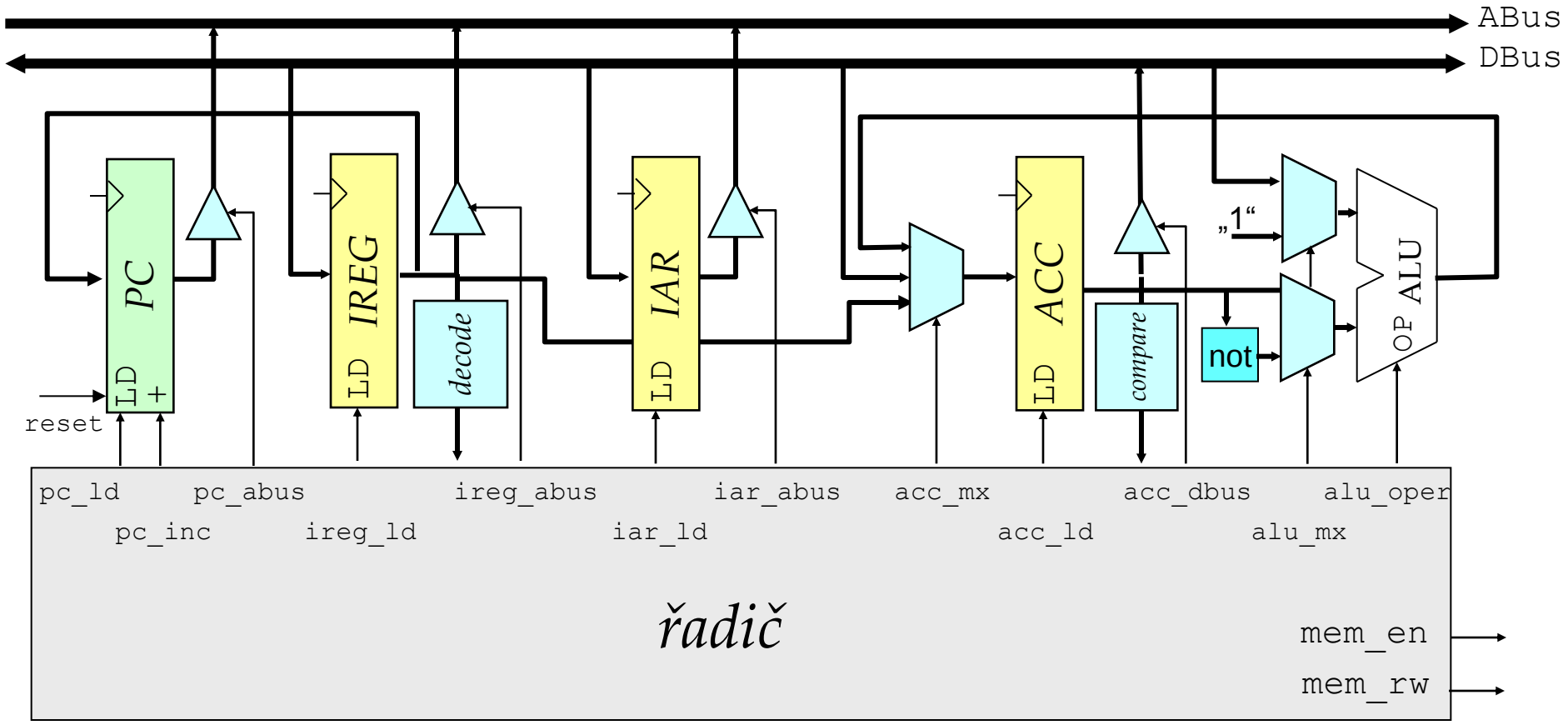
Příklad: ADD acc, m[08]



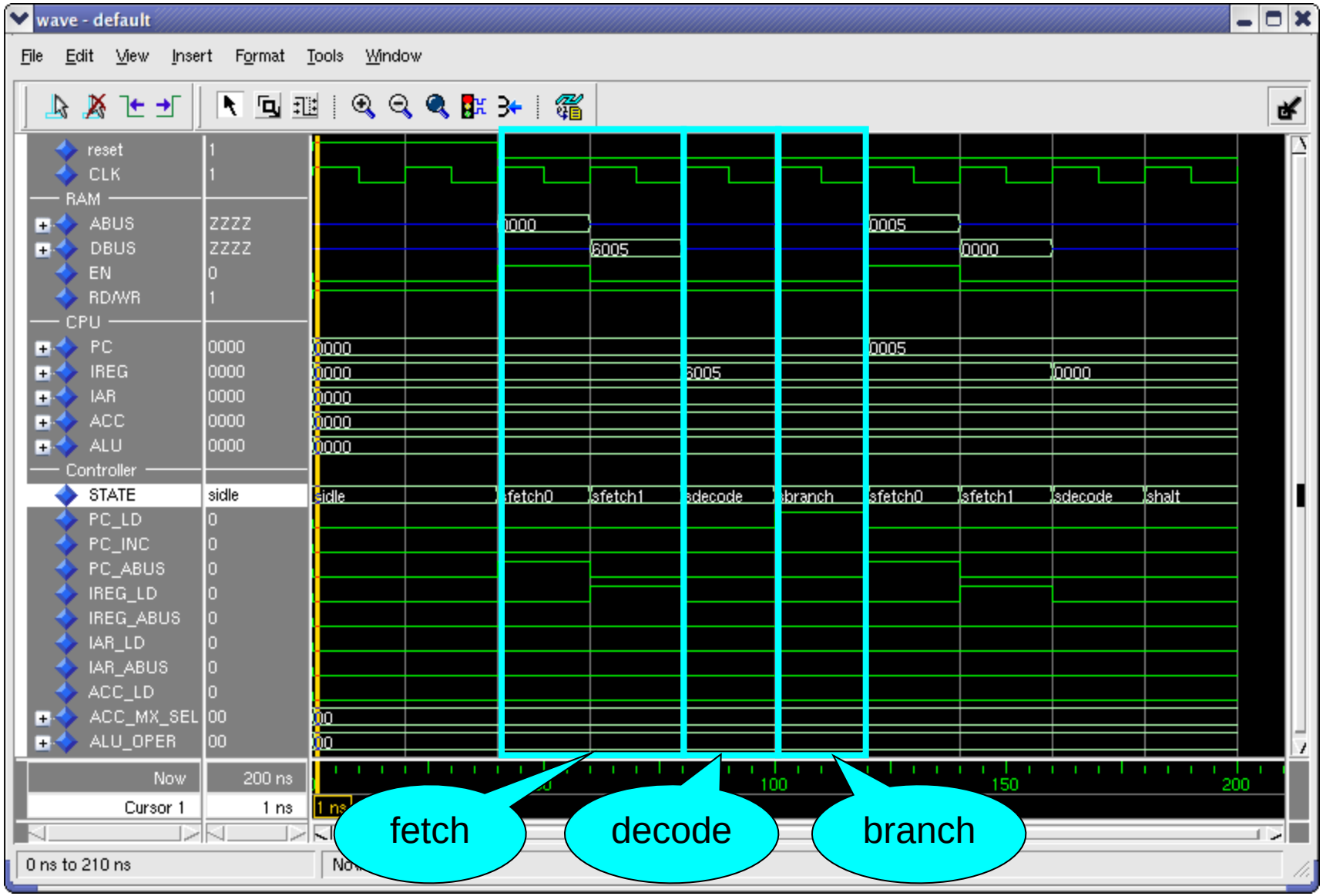
Příklad: ADD acc, m[08]



Příklad: BRANCH 05



Příklad: BRANCH 05

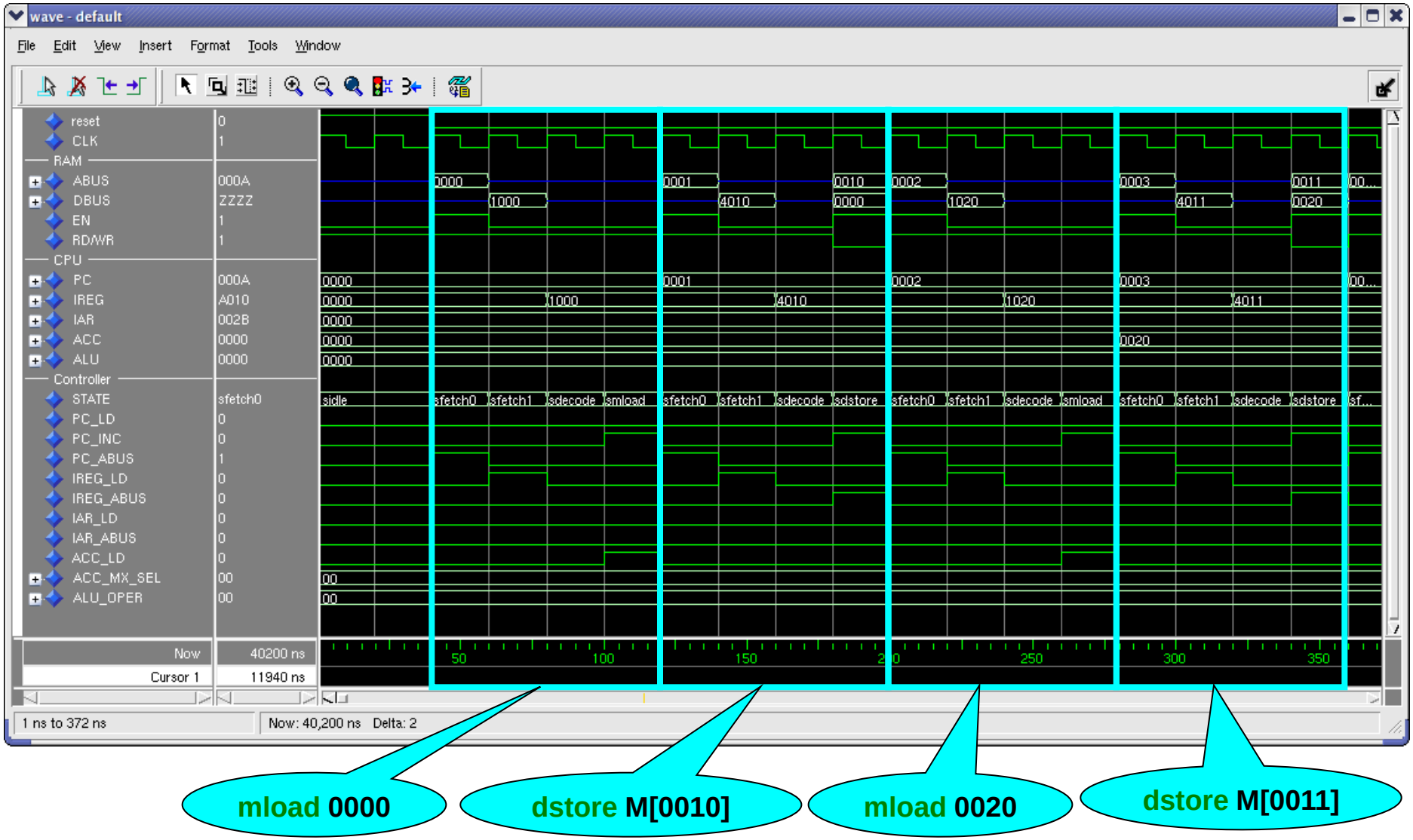


Příklad programu

Sečti hodnoty uložené na adresách 20-2f a zapiš výsledek na adresu 10.

Adresa		*Instrukce*		*Komentář*
0000	(start)	1000	mload 0000	ACC <= 0
0001		4010	dstore M[0010]	M[0010] <= ACC
0002		1020	mload 0020	ACC <= 0020
0003		4011	dstore M[0011]	M[0011] <= ACC
0004	(loop)	1030	mload 0030	ACC <= 0030
0005		0001	negate	ACC <= -ACC
0006		a011	add M[0011]	ACC <= ACC + M[0011]
0007		700f	if 0 branch 000f	goto end
0008		3011	iload *M[0011]	ACC <= *M[0011]
0009		a010	add M[0010]	ACC <= ACC + M[0010]
000a		4010	dstore M[0010]	M[0010] <= ACC
000b		1001	mload 0001	ACC <= 1
000c		a011	add M[0011]	ACC <= M[0011]+ACC
000d		4011	dstore M[0011]	M[0011] <= ACC
000e		6004	branch 0004	goto loop
000f	(end)	0000	halt	halt
0010				Store sum here
0011				Pointer to “next” value

Simulace programu



mload 0000

dstore M[0010]

mload 0020

dstore M[0011]