Centralna procesna enota s fiksno ožičeno nadzorno enoto (ang. hardwired CPU)

V VHDL programirajte arhitekturo strukture fiksno ožičene centralne procesne enote. Podana je entiteta izdelane strukture **cpu**. :Število delovnih registrov je podano s parametrom (**nr_regs**), širina posameznega delovnega registra je (**reg_width**), število naslovov RAM pomnilnika je (**ram_nr_addr**),število naslovov ROM pomnilnika je (**rom_nr_addr**). Podano je I/O vodilo (**IOBUS**) s pripadajočimi priključki izhoda (**IOBUS_Data_out**), vhoda (**IOBUS_Data_in**), tipa operacije (**IOBUS_WnR**). ROM pomnilnik je na fiksno ožičeno centralno procesno enoto povezan preko programskega naslova (**ProgMem_Addr**) in programskega vhoda (**IR**).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
USE work.reg_file_functions.all;
USE work.cpu_datapath_functions.all;
USE work.cpu_functions.all;
use ieee.math_real.all;

entity cpu is
      generic(    nr_regs     : natural := 8;
      reg_width   : natural := 16;
      ram_nr_addr : natural := 4;
      rom_nr_addr : natural := 4
      );
      PORT (clk,  -- clock input
      nRST  : in  std_logic;  -- reset input    (active '0')
      IOBUS_WnR   : out std_logic;  -- io bus write input  (active '1')
      IOBUS_Data_in     : in  std_logic_vector(reg_width - 1 downto 0);-- io bus data input
      IOBUS_Address     : out std_logic_vector(reg_width - 1 downto 0);-- io address bus
      IOBUS_Data_out    : out std_logic_vector(reg_width - 1 downto 0);-- io bus data output
      ProgMem_Addr      : out std_logic_vector(reg_width - 1 downto 0);-- program memory address
      IR    : in  std_logic_vector(reg_width - 1 downto 0)  -- program memory data input
      );
end cpu;
```

Povzetek nabora ukazov glede na obliko naslavljanja:

(a) Registrsko naslavljanje (primer: ADD):

R[DR]<= R[SA] + R[SB]

(b) Takojšnje (immediate) naslavljanje (primer: ADI):

R[DR]<= R[SA] + op op je 3−bitni operand konstante, ki ga od MSB do širine vodila dopolnimo z ničlami (ang. zero fill)

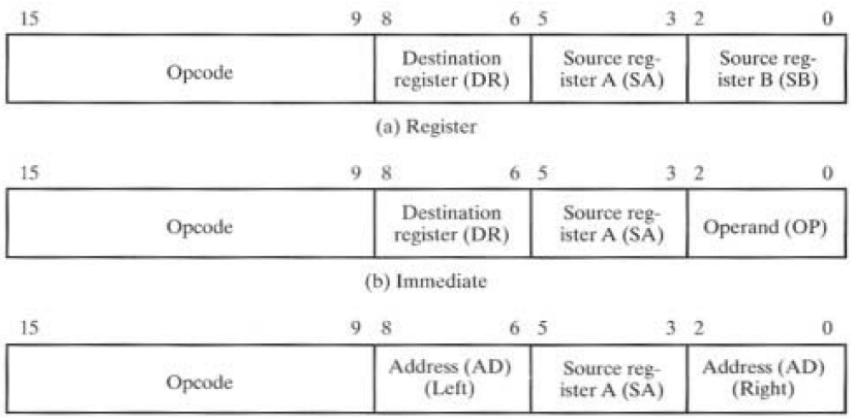
(b) Posredno (indirect) naslavljanje (primer: LOAD, STORE):

R[DR]<= M[R(SA)]

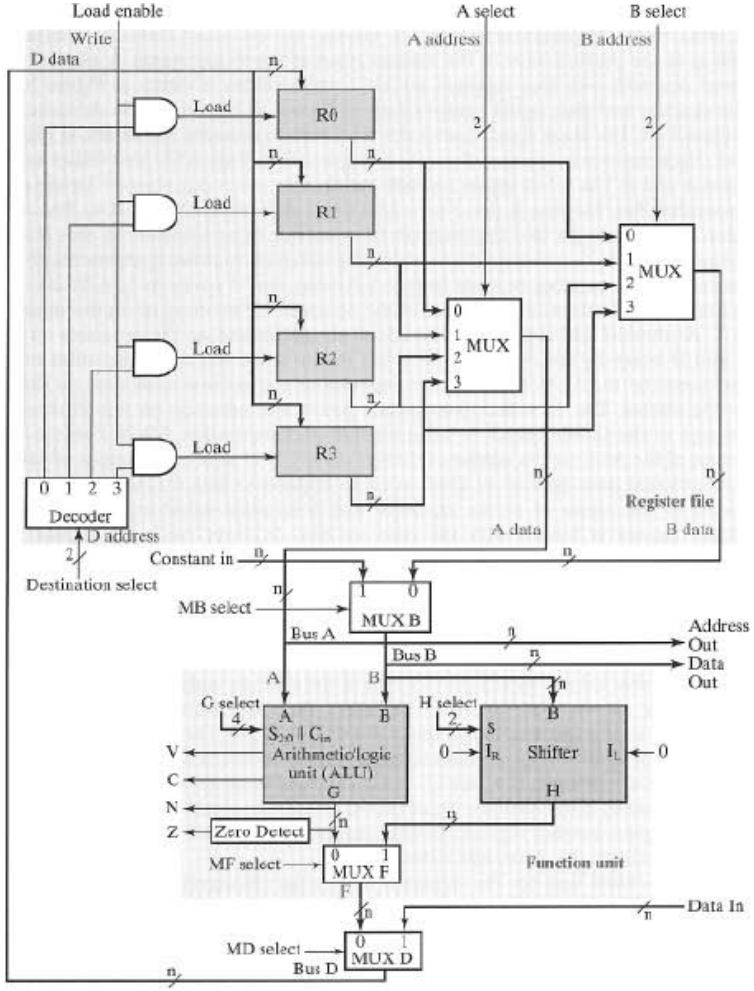
(c) Skok (primer: BRN, ob uspešni vejitvi):

PC <= PC + se(AD) AD je 6−bitni relativni skok IR[8:6]IR[2:0], ki je of MSB do širine vodila dopolnjen z MSB mestom (ang. se − sign extension), tako da predstavlja odmik v dvojiškem komplementu

Instruction	Opcode	Mnemonic	Format	Description	Status Bits
Move A	0000000	MOVA	RD,RA	R[DR] ← R[SA]	N, Z
Increment	0000001	INC	RD,RA	R[DR] ← R[SA] + 1	N, Z
Add	0000010	ADD	RD,RA,RB	R[DR] ← R[SA] + R[SB]	N, Z
Subtract	0000101	SUB	RD,RA,RB	R[DR] ← R[SA] − R[SB]	N, Z
Decrement	0000110	DEC	RD,RA	R[DR] ← R[SA] − 1	N, Z
AND	0001000	AND	RD,RA,RB	R[DR] ← R[SA] ∧ R[SB]	N, Z
OR	0001001	OR	RD,RA,RB	R[DR] ← R[SA] ∨ R[SB]	N, Z
Exclusive OR	0001010	XOR	RD,RA,RB	R[DR] ← R[SA] ⊕ R[SB]	N, Z
NOT	0001011	NOT	RD,RA	R[DR] ← $\overline{R[SA]}$	N, Z
Move B	0001100	MOVB	RD,RB	R[DR] ← R[SB]	
Shift Right	0001101	SHR	RD,RB	R[DR] ← sr R[SB]	
Shift Left	0001110	SHL	RD,RB	R[DR] ← sl R[SB]	
Load Immediate	1001100	LDI	RD, OP	R[DR] ← zf OP	
Add Immediate	1000010	ADI	RD,RA,OP	R[DR] ← R[SA] + zf OP	
Load	0010000	LD	RD,RA	R[DR] ← M[SA]	
Store	0100000	ST	RA,RB	M[SA] ← R[SB]	
Branch on Zero	1100000	BRZ	RA,AD	if (R[SA] = 0) PC ← PC + se AD	
Branch on Negative	1100001	BRN	RA,AD	if (R[SA] < 0) PC ← PC + se AD	
Jump	1110000	JMP	RA	PC ← R[SA]	



Podatkovna pot (ang. datapath)



Slika 1: Izvedba podatkovne poti fiksno ožičene CPU.

Slika 1 prikazuje podatkovno pot (ang. datapath) fiksno ožičene centralne procesne enote. Podatkovna pot je sestavljena iz polja registrov (ang. register file), dveh izbiralnikov vodil (ang. bus multiplexer) in aritmetično logične enote (ALU). Podatkovna pot na sliki 1 vsebuje tudi vzporedni pomikalnik podatkov (ang. barrel shifter), ki ga v realizaciji ne bomo implementirali, kar pomeni da ukazov pomikanja ne bo. Polje delovnih registrov vsebuje 8 registrov (R0 do R7), na sliki 1 so zaradi preglednosti narisani samo štirje. Vhod v polje delovnih registrov je podatkovno vodilo (D). Vpisovanje v delovne registre je izvedeno s signalom (LE) (ang. load enable). Vsebina se v izbrani delovni register vpiše, ko je LE='1'. Izbira registra, kamor se podatek z vodila (D) vpiše je določena s ciljnim naslovom (ang. destination select). Z izbiralnikoma vodil (ang. bus multiplexer) A in B določamo registra, ki se pojavita na izhodnih vodilih A in B skladno z izvornima naslovoma (A select, B select). Izhod vodila B je vezan na izbiralnik vodil (MUX B), ki določa, ali bo na vhodu aritmetično logične enote izbrana konstanta (`constant in`) ali vodilo B. Vodili A in B sta povezani na izhodno vhodno/izhodno (`I/O`) vodilo, tako da je vodilo A (ang. A bus) vezano na naslovni vhod I/O vodila (`address_out`), vodilo B pa na podatkovni izhod I/O vodila (`data_out`). Na izhodu aritmetično logične enote se na sliki 1 nahaja izbiralnik vodil (`MUX F`), ki določa ali je izbran izhod ALU ali izhod vzporednega pomikalnika podatkov. V našem primeru vzporednega pomikalnika ne bomo vključevali v podatkovno pot, zato izbiralnika (`MUX F`) ne potrebujemo. Na koncu podatkovne poti je izbiralnik vodil (`MUX D`), ki določa ali se bo v polje delovnih registrov vpiše rezultat operacije, ali vhod z I/O vodila (`Data_in`).

1. Ustvarite VHDL datoteko **cpu_datapath_functions.vhd** v kateri boste znotraj VHDL paketa (**PACKAGE**) programirali funkcije, potrebne za izvedbo sinteze podatkovne poti fiksno ožičene centralne procesne enote.. V datoteki uporabljamo funkcije splošnega polja registrov (**reg_file_functions**). V datoteki (**cpu_datapath_functions.vhd**) sta podani deklaraciji komponent podatkovne poti (**cpu_datapath**) in aritmetično logične enote (**ndn_alu_cla.vhd**):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.reg_file_functions.all;

PACKAGE cpu_datapath_functions IS

	component cpu_datapath is
	generic( nr_regs  : natural := 4;
	reg_width   : natural := 8);
	PORT (clk,  -- clock input
	RW    : in std_logic;   -- register write input (active '1')
	nRST  : in std_logic;   -- reset input    (active '0')
	DA,   -- destination register number select input
	AA,   -- A, B bus register number select input
	BA    : in  std_logic_vector( sizeof(nr_regs - 1) - 1 downto 0);
	MB    : in std_logic;   -- constant/operand bus B bus multiplexer control signal
	MD    : in std_logic;   -- external data/alu result multiplexer control signal     ALU_mode    : in
std_logic;  --mode of alu operation (M in upper table)
	ALU_function : in std_logic_vector(2 downto 0); -- function of ALU (F in upper table)
	ALU_N_bit   : out std_logic;  -- Negative bit of alu operation
	ALU_C_bit   : out std_logic;  -- Carry bit of alu operation
	ALU_V_bit   : out std_logic;  -- Overflow bit of alu operation
	ALU_Z_bit   : out std_logic;  -- Zero bit of alu operation
	Const_in    : in  std_logic_vector(reg_width - 1 downto 0);--constant input bus
	Data_in     : in  std_logic_vector(reg_width - 1 downto 0);--data input bus input
	Address_out : out std_logic_vector(reg_width - 1 downto 0);--address bus output
	Data_out    : out std_logic_vector(reg_width - 1 downto 0) --data bus output
	);
	end component;
```

```
component ndn_alu is
      generic( n: natural := 8 );
      port( M     :     in    std_logic;
      --način delovanja ('0' => aritmetični, '1' => logični)
      F     :     in    std_logic_vector(2 downto 0);
      -- funkcijski vhod za operacije
      X, Y  :     in    std_logic_vector(n-1 downto 0);
      S     :     out std_logic_vector(n-1 downto 0);
      Negative,
      Cout,
      Overflow,
      Zero,
      Gout,
      Pout  :     out   std_logic );
      end component;

END cpu_datapath_functions;
```

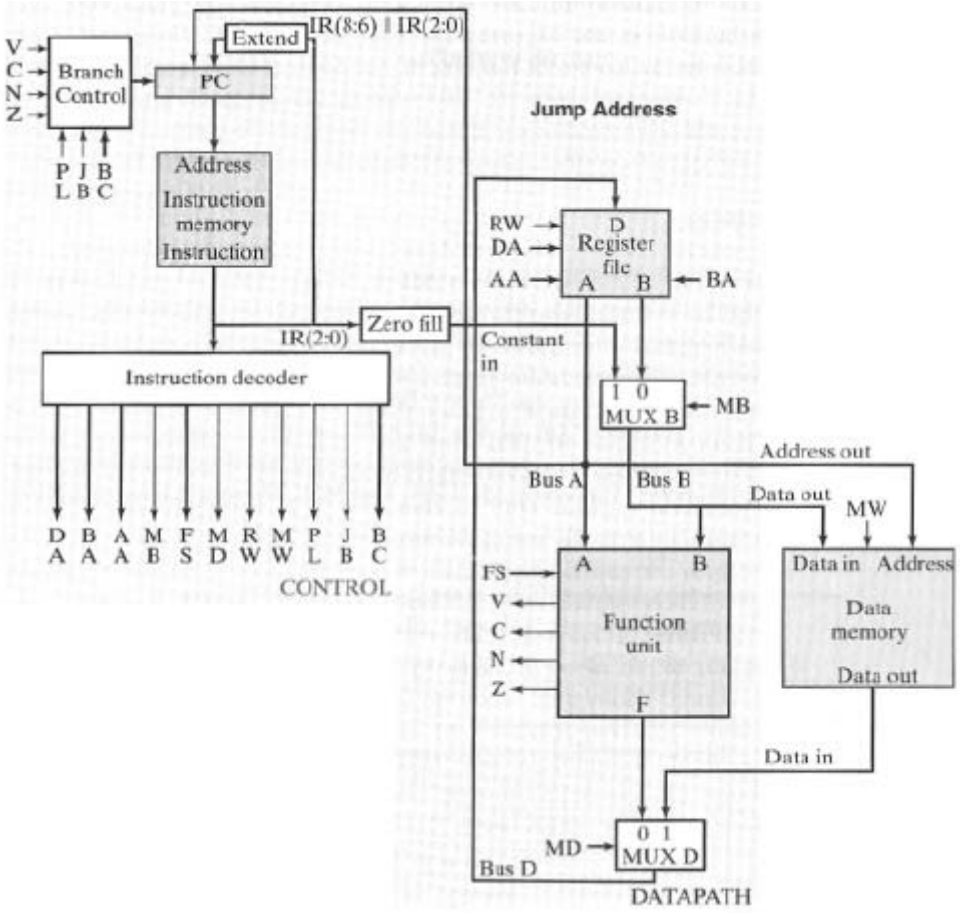
2. Ustvarite *nov projekt*, v katerega dodajte VHDL datoteko **cpu_datapath.vhd** iz predloge projekta v imeniku **cpu_datapath**. V datoteki (**cpu_datapath.vhd**) programirajte entiteto in arhitekturo strukture podatkovne poti (**cpu_datapath**) z uporabo povezovalnega stavka po sliki 1 *brez* izbiralnika vodil (MUX F) in vzporednega pomikalnika. Podana je entiteta strukture, imena in opisi signalov sovpadajo z imeni na sliki 1:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE work.reg_file_functions.all;
USE work.cpu_datapath_functions.all;
use ieee.math_real.all;

entity cpu_datapath is
      generic(    nr_regs     : natural := 4;
      reg_width   : natural := 8);
      PORT (clk,  -- clock input
      RW    : in std_logic;   -- register write input (active '1')
      nRST  : in std_logic;   -- reset input    (active '0')
      DA,         -- destination register number select input
      AA,         -- A, B bus register number select input
      BA    : in  std_logic_vector( sizeof(nr_regs - 1) - 1 downto 0);
      MB    : in std_logic;   -- constant/operand bus B bus multiplexer control signal
      MD    : in std_logic;   -- external data/alu result multiplexer control signal     ALU_mode    : in
std_logic;  --mode of alu operation (M in upper table)
      ALU_function      : in std_logic_vector(2 downto 0); -- function of ALU (F in upper table)
      ALU_N_bit   : out std_logic;  -- Negative bit of alu operation
      ALU_C_bit   : out std_logic;  -- Carry bit of alu operation
      ALU_V_bit   : out std_logic;  -- Overflow bit of alu operation
      ALU_Z_bit   : out std_logic;  -- Zero bit of alu operation
      Const_in    : in  std_logic_vector(reg_width - 1 downto 0);--constant input bus
      Data_in     : in  std_logic_vector(reg_width - 1 downto 0);--data input bus input
      Address_out : out std_logic_vector(reg_width - 1 downto 0);--address bus output
      Data_out    : out std_logic_vector(reg_width - 1 downto 0) --data bus output
      );
end cpu_datapath;
```

Pri preverjanju pravilnosti delovanja uporabite priloženo datoteko testnih vrednosti (**cpu_datapath_tb.vhd**) in s simulacijo preverite pravilnost delovanja podatkovne poti.

Nadzorno vezje za skoke in vejitve (ang. branch control)



Slika 2: Arhitektura enostavnega mikroračunalnika.

Za izvedbo skočnih ukazov moramo dodati nadzorno vezje za skok in vejitve (ang. branch control), ki je prikazano na sliki 2. Vezje je vsebuje komponento programskega števca (ang. program counter - PC), ki se ob normalnem izvajanju programa povečuje. Ko nadzorno vezje za skok in vejitve prejme ustrezen ukaz ukaznega dekoderja (ang. instruction decoder), se števec *naloži* z neko vrednostjo glede na stanje bitov ukaznega dekoderja. Naloži se seveda vedno, ko gre za skok (ang. jump) ali ko gre za *izpolnjen* pogoj vejitve (ang. branch taken). Če pogoj za vejitev ni izpolnjen, števec šteje (PC<=PC+1). Biti ukaznega dekoderja so PL (ang. PC load), JB (ang. jump/branch), BC (ang. branch control). Bit PL določa kdaj gre za operacijo nalaganja števca. Pri ukazih, ki niso skočni, je '0', sicer je '1'. Bit JB določa ali gre za brezpogojni skok (ang. jump) ali za vejitev (ang. branch). Bit BC določa za kateri tip pogojne vejitve gre: Če je BC <= '0', gre za vejitev ob (Z='1'), kar povzema ukaz (BRZ). Če je BC <= '1', gre za vejitev ob (N='1'), kar povzema ukaz (BRN). Biti N, C, V in Z so vhodi v nadzorno vezje za skok in vejitve. Delovanje nadzornega vezja za skok in vejitve povzema spodnja tabela:

PL	JB	BC	Z	N	programski števec (PC)
0	X	X	X	X	PC + 1
1	0	0	1	X	vejitev (Z='1')
1	0	0	0	X	ni vejitve (Z='0')=> PC + 1
1	0	1	X	1	vejitev (N='1')
1	0	1	X	0	ni vejitve (N='0')=> PC + 1
1	1	X	X	X	jump

Ob uspešnem skoku mora nadzorno vezje vzporedno naložiti (ang. parallel load) absolutni naslov mesta skoka. Ob uspešni vejitvi mora nadzorno vezje trenutnemu stanju programskega števca prišteti relativni odmik (ang. relative offset) in tako izračunati novo vrednost programskega števca. Relativni odmik naj bo polne širine vodila (16 bitov). Podatkovna pot nadzornega vezja za skok in vejitve (ang. branch control) je sestavljena iz seštevalnika in izbiralnika vodila. Izbiralnik vodila določa ali se bo v števec naložil absolutni ali relativni odmik. Relativni odmik se s seštevanjem trenutnega izhoda števca in vhoda za odmik (**JB_address**). Vrednost vhoda za odmik (**JB_address**) bo v nadaljevanju privzeta v dvojiškem komplementu, s čimer omogočimo vejitve nazaj na že izvedeno kodo.

3. Iz podanega opisa in vezja na sliki 2 narišite podatkovno pot nadzornega vezja za skok in vejitve in sliko shranite v imenik projekta. Ime slike naj bo (**branch_ctrl.jpg**). Na sliki označite krmilne signale, ki boste uporabljali v VHDL arhitekturi. Vrednosti krmilnih signalov za omogočanje štetja (**CE**) in vzporedno nalaganje števca (**nLOAD**) opišite z logično funkcijo.
4. Ustvarite VHDL datoteko **branch_ctrl_functions.vhd** v kateri boste znotraj VHDL paketa (**PACKAGE**) programirali funkcije, potrebne za izvedbo nadzorne enote za vejitve in skoke (ang. branch jump control unit). V datoteki uporabljamo funkcije splošnega polja registrov (**reg_file_functions**). V datoteki (**cpu_datapath_functions.vhd**) so podane deklaracije komponent CLA seštevalnika (**cla_add_n_bit**), enote tvorjenja in širjenja CLA seštevalnika (**cla_gp**) ter števca z vzporednim nalaganjem (**counter**):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.reg_file_functions.all;

PACKAGE branch_ctrl_functions IS

	COMPONENT cla_add_n_bit IS
	generic(n: natural := 8);
	PORT (	Cin   :	in    std_logic ;
	X, Y  :	in    std_logic_vector(n-1 downto 0);
	S     :	out   std_logic_vector(n-1 downto 0);
	Gout,
	Pout,
	Cout  :	out   std_logic);
	END COMPONENT;

	COMPONENT cla_gp IS
	PORT (	Cin, x, y : IN STD_LOGIC;
	S, Cout, g, p : OUT STD_LOGIC );
	END COMPONENT;
```

```
        COMPONENT counter is
        generic( ctr_size: natural := 4);
        PORT (      clk,  -- signal ure
        nCLR,       -- signal za brisanje števca (aktiven '0')
        nLOAD,      -- signal za nalaganje števca (aktiven '0')
        ENP, ENT : IN std_logic;      -- signala za omogoèanje štetja (aktiven '1')
        RCO : out std_logic;    -- izhodni prenos na naslednjo stopnjo (rco)
        x : in std_logic_vector(ctr_size - 1 downto 0);      -- vhod za vzporedno nalaganje
        Q : out std_logic_vector(ctr_size - 1 downto 0)      -- izhodno štetje
        );
        end COMPONENT;
END branch_ctrl_functions;
```

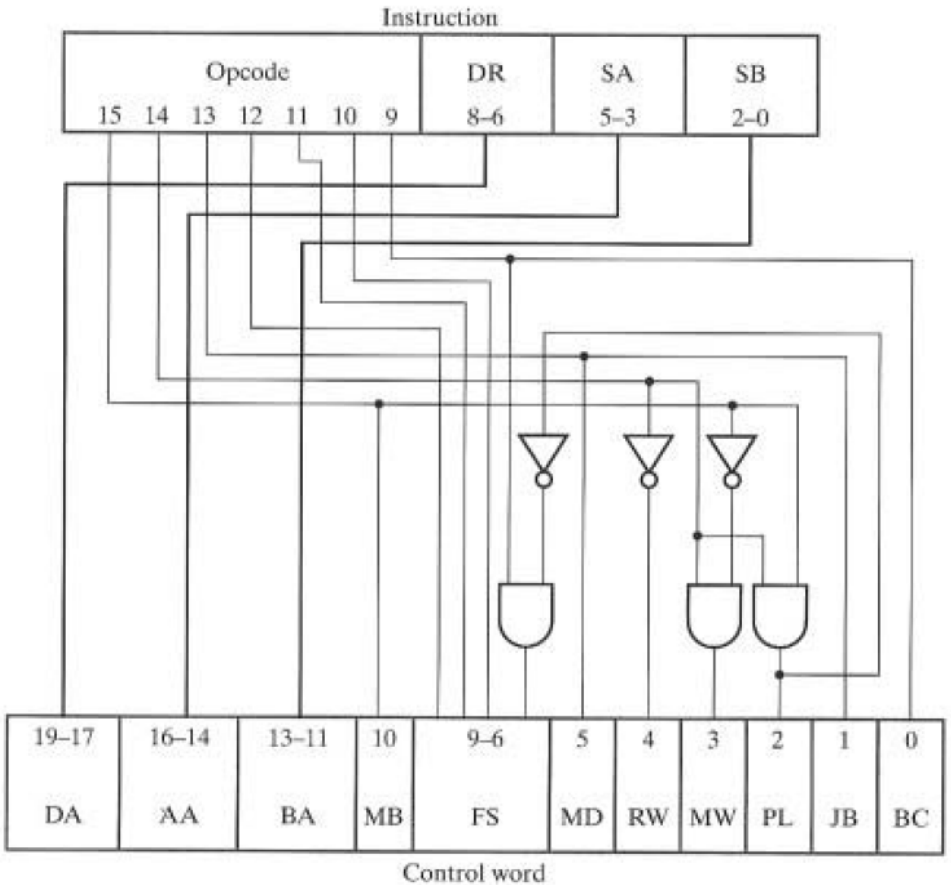
5. Ustvarite *nov projekt*, v katerega dodajte VHDL datoteki **branch_ctrl.vhd** in **branch_ctrl_functions.vhd** iz predloge projekta v imeniku **cpu_branch_ctrl**. V datoteki (**branch_ctrl.vhd**) programirajte entiteto in arhitekturo strukture nadzornega vezja za skok in vejitve (ang. branch control) (**branch_ctrl**) z uporabo povezovalnega stavka. Parameter (**ctr_width**) določa število bitov štetja programskega števca. Vhodni signal (**JB_address**) predstavlja vrednost odmika: Pri skoku (**JMP**) se ta vrednost naloži vzporedno v programski števec, pri vejitvi (**BRX**) se ta vrednost prišteje trenutni vrednosti programskega števca. Rezultat seštevanja se nato vzporedno vpiše v programski števec. Vhod (**nRST**) asinhrono postavlja vrednost programskega števca na 0. Vhodi **N** (negativno), **C** (prenos), **V** (preliv), **Z** (nič) predstavljajo pogoje za različne vejitve in izvirajo iz rezultata aritmetično logične enote. Od tipičnih vejitev sta izdelani samo (**BRZ**) in (**BRN**), zato bosta pomembna samo bita (**N**) in (**Z**). Krmilni vhodi nadzornega vezja so (**PL, JB, BC**). Izhod nadzornega vezja je programski števec (**PC**).

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
USE work.reg_file_functions.all;
USE work.branch_ctrl_functions.all;
use ieee.math_real.all;

entity branch_ctrl is
      generic( ctr_width      : natural := 16);
      PORT (clk,  -- clock input
      nRST, -- reset input    (active '0')
      N,    -- negative bit from ALU operation
      C,    -- carry bit from ALU operation
      V,    -- overflow bit from ALU operation
      Z,    -- zero bit from ALU operation
      PL,   -- Increment (PC = PC + 1) when inactive, or load when active (PL = '1')
      JB,   -- jump/branch when PL='1' (jump => PL = '1', load counter with predefined value)
      BC    : in  std_logic;  --branch control (when '0' -> check Z bit, when '1' check N bit)
      JB_address  : in  std_logic_vector(ctr_width - 1 downto 0);
      PC    : out std_logic_vector(ctr_width - 1 downto 0)
      );
end branch_ctrl;
```

Pri preverjanju pravilnosti delovanja uporabite *priloženo* datoteko testnih vrednosti (**branch_ctrl_tb.tbw**) in s simulacijo preverite pravilnost delovanja podatkovne poti.

Ukazni dekoder:



Slika 3: Ukazni dekoder.

Za izvedbo krmilne enote podatkovne poti, nadzornega vezja za skok in vejitve ter I/O vodila moramo dodati ukazni dekoder, katerega naloga je, da ukaz (ang. instruction) prekodira v niz bitov nadzorne besede (ang. control word). Nadzorna beseda vsebuje signale na sliki 1, ki so povzeti v spodnjem seznamu:

```
RW      -- vpis delovnega registra (aktiven '1')
DA      -- ciljni naslov delovnega registra (povezuje se
na D_select)
AA      -- izvorni naslov delovnega registra na vodilu A
(povezuje se na A_select)
BA      -- izvorni naslov delovnega registra na vodilu B
(povezuje se na B_select)
MB      -- krmilni signal izbiralnika na vodilu B
MD      -- krmilni signal izbiralnika na vodilu D
ALU_mode     -- način delovanja ALU
ALU_function       -- funkcija ALU
MW      -- krmilni signal za vpis RAM pomnilnika (vpis =
'1', branje = '0')
PL      -- skočni/neskočni ukaz
JB      -- vejitev/skok
BC      -- vrsta vejitve
```

ALU, ki smo jo izdelali pri predmetu NDN (**ndn_alu.vhd**), se rahlo razlikuje od ALU iz slike 2 (signal FS), zato signal FS tvorimo tako da način delovanja ALU (**ALU_mode**) postavimo na MSB mesto signala FS, funkcijo ALU (**ALU_function**) postavimo na spodnja tri mesta signala FS:

```
ALU_mode <= ir(12);
ALU_function <= ir(11 downto 10) & (ir(9) and (not
PL));
```

Za povezovanje naslova relativnega skoka moramo vrednost relativnega odmika razširiti z bitom predznaka. To z uporabo (`numeric.std.all`) knjižnice storimo z ukazom:

```
JB_address <= STD_LOGIC_VECTOR(resize(signed(ir(8 downto 6) & ir(2 downto 0)), JB_address'length));
```

Podobno storimo za prilagoditev vsebovanega operanda konstante (`Const_in`), le da v tem primeru konstanto dopolnimo z ničlami do širine vodila z ukazom:

```
Const_in <= STD_LOGIC_VECTOR(resize(unsigned(ir(2 downto 0)), Const_in'length));
```

Vse ostale signale komponent in izhodne entitete lahko povežemo neposredno, le povezovanje programskega števca (**PC**) na izhodni priključek naslova programskega spomina (**ProgMem_addr**) in povezovanje signala (**MW**) na izhodni priključek za krmiljenje načina delovanja I/O enote RAM spomina (**IOBUS_WnR**) moramo izvesti posebej.

6. Ustvarite nov projekt, v katerega dodajte vse VHDL datoteke iz projektov podatkovne poti in nadzornega vezja za skok in vejitve (ang. branch control). Iz predloge projekta v imeniku **cpu** kopirajte tudi datoteko (**cpu.vhd**). V datoteki (**cpu.vhd**) programirajte entiteto in arhitekturo strukture fiksno ožičene centralne procesne enote z uporabo povezovalnega stavka. Parameter (**nr_regs**) določa število delovnih registrov. Parameter (**reg_width**) določa širino delovnega registra in s tem določa širino vseh vodil v arhitekturi. Število naslovov RAM pomnilnika je (**ram_nr_addr**), število naslovov ROM pomnilnika je (**rom_nr_addr**). Podano je tudi I/O vodilo (**IOBUS**) s pripadajočimi priključki izhoda (**IOBUS_Data_out**), vhoda (**IOBUS_Data_in**), tipa operacije (**IOBUS_WnR**). ROM pomnilnik je na fiksno ožičeno centralno procesno enoto povezan preko programskega naslova (**ProgMem_Addr**) in podatkovnega vhoda (**IR**).

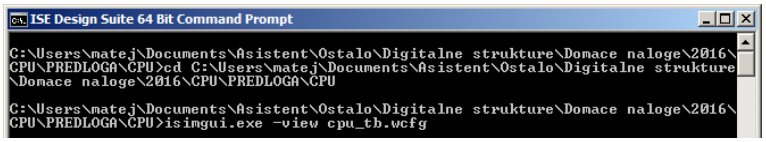
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
USE work.reg_file_functions.all;
USE work.cpu_datapath_functions.all;
USE work.cpu_functions.all;
use ieee.math_real.all;

entity cpu is
      generic(    nr_regs     : natural := 8;
      reg_width   : natural := 16;
      ram_nr_addr : natural := 4;
      rom_nr_addr : natural := 4
      );
      PORT (clk,        -- clock input
      nRST  : in  std_logic;  -- reset input    (active '0')
      IOBUS_WnR   : out std_logic;  -- io bus write input   (active '1')
      IOBUS_Data_in     : in  std_logic_vector(reg_width - 1 downto 0);-- io bus data input
      IOBUS_Address     : out std_logic_vector(reg_width - 1 downto 0);-- io address bus
      IOBUS_Data_out    : out std_logic_vector(reg_width - 1 downto 0);-- io bus data output
      ProgMem_Addr      : out std_logic_vector(reg_width - 1 downto 0);-- program memory address
      IR    : in  std_logic_vector(reg_width - 1 downto 0)  -- program memory data input
      );
end cpu;
```

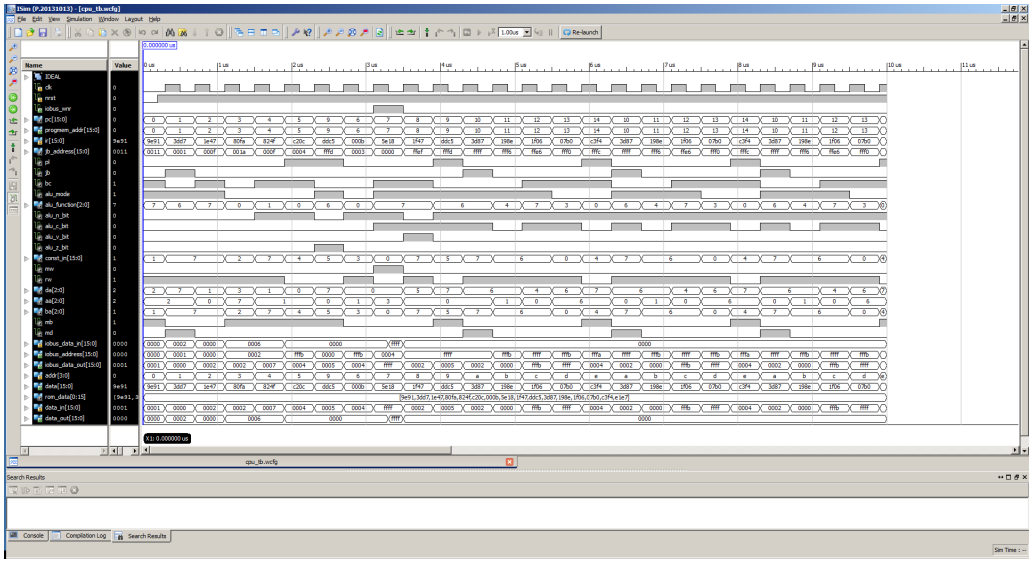
Interna signala (**MB**) in (**MD**) sta tipa (**std_logic**). Če ju želite uporabiti kot naslovni vhod 2/1 izbiralnika vodil (muxnto1_bus.vhd), ga morate pretvoriti v enobitni vektor tipa (**std_logic_vector**). V predlogi je zato definirana spremenljivka **MD_vector : std_logic_vector(0 downto 0)**, vektor z enim elementom z indeksom 0. Ta spremenljivka predstavlja vmesnik iz tipa (**std_logic**) na tip (**std_logic_vector**). Tipa (**std_logic**) namreč ne morete neposredno povezati na (**std_logic_vector**), saj bo VHDL javil napako. Signal (**MD**) vežete na edini element vektorskega signala: **MD_vector(0) <= MD;**. Podobna logika velja pri signalu MB.

Za preverjanje pravilnosti delovanja je podan enostaven program v zbirnem jeziku (**TEST_ROM.asm**) s pripadajočo strojno kodo (**TEST_ROM.hex**) in začetno vsebino RAM (**TEST_RAM.hex**). Spominski komponenti ROM in RAM sta povezani v priloženi datoteki testnih vrednosti (**cpu_tb.vhd**). Če želite, lahko lastni program z ukazi zbirnega jezika zapišite v datoteko (**rom.asm**), njegovo strojno kodo pa v (**rom.hex**) in (**ram.hex**). S simulacijo preverite pravilnost izvajanja testnega programa. Rezultat simulacije testnega programa v zbirniku je podan v datoteki (**cpu_IDEAL.xwv**), ki si jo lahko v okolju Xilinx ISE 10.1 ogledate s programom (**isimwave**) (Start→Run→Isimwave).

Za prikaz rezultatov simulacije testnega programa v zbirniku (***.wcfg**, ***.wdb**) v okolju Xilinx ISE 14.7 najprej zaženete ukazno okno z registriranimi Xilinx spremenljivkami: V operacijskem sistemu Windows izberete (Start→All programs→Xilinx Design Tools→ISE Design Suite 14.7→Accessories). Glede na vrsto vašega operacijskega sistema (32/64 bitni) izberete ukazno okno (**ISE Design Suite 32 Bit Command Prompt**) oz. (**ISE Design Suite 64 Bit Command Prompt**) in ga poženete. V ukaznem oknu z ukazom **CD** zamenjajte imenik na mesto, kjer se nahaja arhiv predloge vaje in izberite podmapo predloge CPU, v kateri se nahajata datoteki rezultatov simulacije testnega programa v zbirniku (**cpu_tb_isim_beh.wdb** in **cpu_tb.wcfg**). V ukaznem oknu vtipkajte ukaz (**isimgui.exe -view cpu_tb.wcfg**) in pojavi se okno z rezultati simulacije. Prikazano okno primerjajte s svojimi rezultati.



Slika 4: Ukazna vrstica ISE Design Suite 64 Bit Command Prompt.



Slika 5: Okno statičnega prikaza rezultatov simulacije iSim.

Tabela 1: Testni program v zbirnem in strojnem jeziku.

ROM addr.	CMD	opcode(bin)	cmd(bin)	cmd(hex)	OPERATION
	RST				
0	LDI	1001111	1001111010010001	9E91	R(2)←1
1	LOADA	0011110	0011110111010111	3DD7	R(7)← M(R2)
2	MOVEB	0001111	0001111001000111	1E47	R(1)← R(7)
3	ADI	1000000	1000000011111010	80FA	R(3)← R(7)+ 010
4	SBI	1000001	1000001001001111	824F	R(1)← R(1)- 111
5	BRN	1100001	1100001000001100	C20C	IF(R(1) < 0) PC = PC + 4
6	AplusB	0000000	0000000000001011	000B	R(0)← R1 + R3
7	STOREB	0101111	0101111000011000	5E18	M(R3)← R(0)
8	MOVEB	0001111	0001111101000111	1F47	R(5)← R(7)
9	BRZ	1101110	1101110111000101	DDC5	IF(R(0)=0) PC = PC - 3
10	LOADA	0011110	0011110110000111	3D87	R(6)← M(R0)
11	AxorB	0001100	0001100110001110	198E	R(6)← R1 XOR R6
12	MOVEB	0001111	0001111100000110	1F06	R(4)← R(6)
13	Aminus1	0000011	0000011110110000	07B0	R(6)← R(6) - 1
14	BRN	1100001	1100001111110100	C3F4	IF(R(6)<0) PC = PC - 4
15	JMP	1110000	1110000111100111	E1E7	R(4)

Tabela 2: Vsebina RAM pomnilnika iz datoteke ram.hex

Address	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Contents	0000	0002	0006	FFF2	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000	0000
	0	2	6	−14	0	0	0	0	0	0	0	0	0	0	0	0

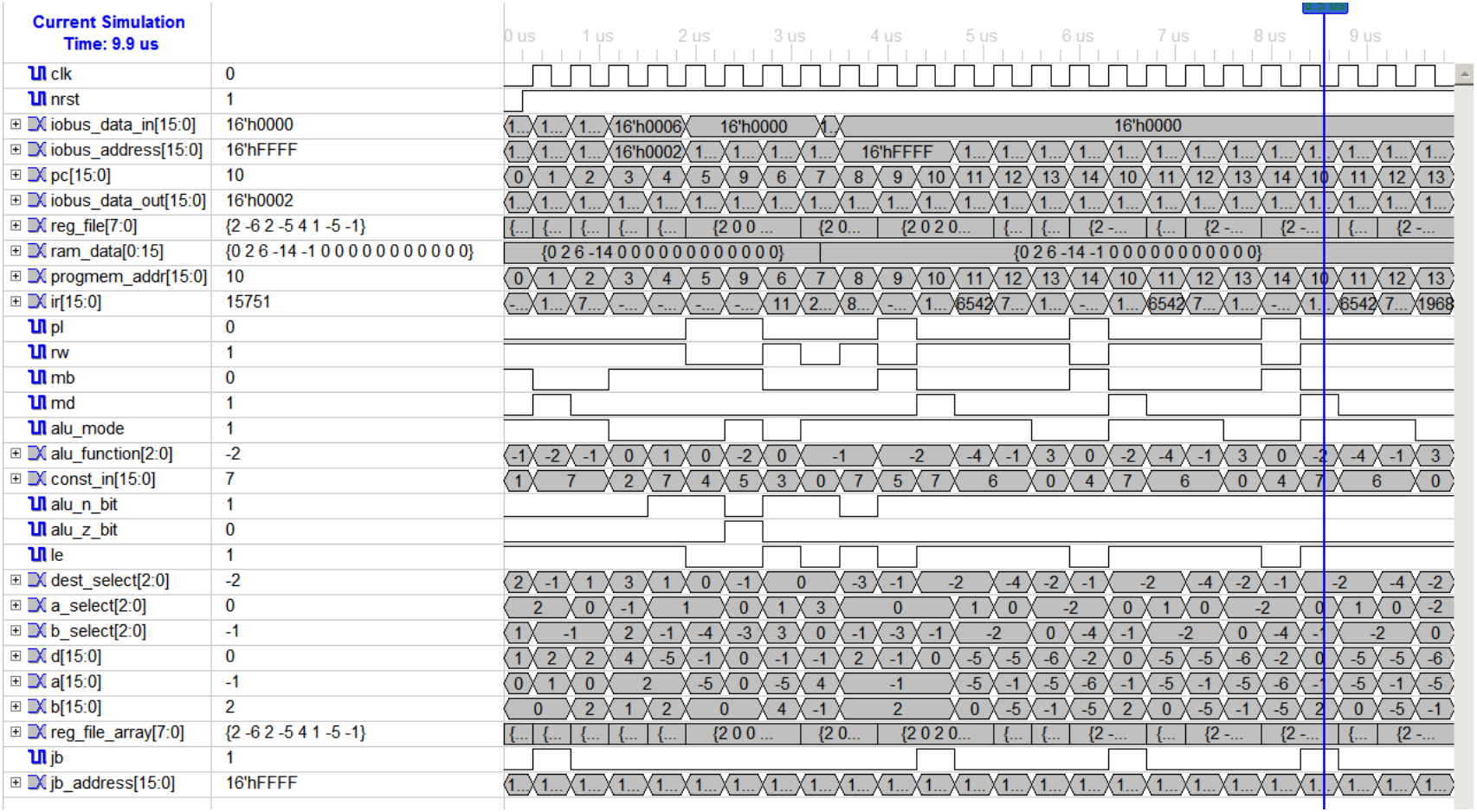
Tabela 3: Izvajanje testnega programa (ang. trace) v zbirnem jeziku z vsebino registrov R0…R7 ter prvih petih lokacij RAM pomnilnika.

ROM ADDR.	CMD	OPERATION	Next Addr (PC)	R0	R1	R2	R3	R4	R5	R6	R7	RAM0	RAM1	RAM2	RAM3	RAM4
	RST		0	0	0	0	0	0	0	0	0	0	2	6	-14	0
0	LDI	R(2)←1	1	0	0	1	0	0	0	0	0	0	2	6	-14	0
1	LOADA	R(7)← M(R(2))	2	0	0	1	0	0	0	0	2	0	2	6	-14	0
2	MOVEB	R(1)← R(7)	3	0	2	1	0	0	0	0	2	0	2	6	-14	0
3	ADI	R(3)← R(7)+ 010	4	0	2	1	4	0	0	0	2	0	2	6	-14	0
4	SBI	R(1)← R(1)- 111	5	0	-5	1	4	0	0	0	2	0	2	6	-14	0
5	BRN	IF(R(1) < 0) PC = PC + 4	9	0	-5	1	4	0	0	0	2	0	2	6	-14	0
9	BRZ	IF(R(0)=0) PC = PC - 3	6	0	-5	1	4	0	0	0	2	0	2	6	-14	0
6	AplusB	R(0)← R(1) + R(3)	7	-1	-5	1	4	0	0	0	2	0	2	6	-14	0
7	STOREB	M(R3)← R(0)	8	-1	-5	1	4	0	0	0	2	0	2	6	-14	-1
8	MOVEB	R(5)← R(7)	9	-1	-5	1	4	0	2	0	2	0	2	6	-14	-1
9	BRZ	IF(R(0)=0) PC = PC - 3	10	-1	-5	1	4	0	2	0	2	0	2	6	-14	-1
10	LOADA	R(6)← M(R(0))	11	-1	-5	1	4	0	2	0	2	0	2	6	-14	-1
11	AxorB	R(6)← R1 XOR R(6)	12	-1	-5	1	4	0	2	-5	2	0	2	6	-14	-1
12	MOVEB	R(4)← R(6)	13	-1	-5	1	4	-5	2	-5	2	0	2	6	-14	-1
13	Aminus1	R(6)← R(6) - 1	14	-1	-5	1	4	-5	2	-6	2	0	2	6	-14	-1
14	BRN	IF(R(6)<0) PC = PC - 4	10	-1	-5	1	4	-5	2	-6	2	0	2	6	-14	-1
10	LOADA	R(6)← M(R(0))	11	-1	-5	1	4	-5	2	0	2	0	2	6	-14	-1
11	AxorB	R(6)← R1 XOR R(6)	12	-1	-5	1	4	-5	2	-5	2	0	2	6	-14	-1
12	MOVEB	R(4)← R(6)	13	-1	-5	1	4	-5	2	-5	2	0	2	6	-14	-1
13	Aminus1	R(6)← R(6) - 1	14	-1	-5	1	4	-6	2	-5	2	0	2	6	-14	-1
14	BRN	IF(R(6)<0) PC = PC - 4	10	-1	-5	1	4	-6	2	-5	2	0	2	6	-14	-1

Zaporedje ukazov od PC=10 do PC=14 se ponavlja.

Spremenjena vsebina registrov ter prvih petih lokacij RAM pomnilnika je označena rdeče.

Pri ukazu **LOADA** je vrednost registra **R(0)** enaka -1_{10}. Slednje pomeni, da se v RAM naloži lokacija **M(R0)**, ki v danem primeru pomeni zadnjo lokacijo - lokacijo 15 (same enice). Program z dano vsebino RAM nikdar ne doseže vrednosti ukaza **JMP R(4)**. Če bi na lokacijo RAM15 zapisali število, ki je večje od 1, bi se program končal.



Slika 6: Potek izvajanja testnega programa v programu isimwave.

Tabela 4: Nekatere operacije z ALU in poljem registrov.

15	**14**	**13**	**12**	**11, 10, 9**		
MB	NOT(RW)	MD	ALU_mode	ALU_function	Koda operacije	Izvedba operacije
0	0	0	0	000	A plus B	R(DR)←R(SA)+ R(SB)
0	0	0	0	001	A minus B	R(DR)←R(SA)- R(SB)
0	0	0	0	010	A plus 1	R(DR)←R(SA)+1
0	0	0	0	011	A minus 1	R(DR)←R(SA)-1
0	0	0	0	100	A plus A	R(DR)←LSL(R(SA))
0	0	0	0	101	minus 1 (2'K)	R(DR)← –1
0	0	0	1	000	A and B	R(DR)←R(SA) AND R(SB)
0	0	0	1	001	A nand B	R(DR)←R(SA) NAND R(SB)
0	0	0	1	010	A or B	R(DR)←R(SA) OR R(SB)
0	0	0	1	011	A nor B	R(DR)←R(SA) NOR R(SB)
0	0	0	1	100	A xor B	R(DR)←R(SA) XOR R(SB)
0	0	0	1	101	A xnor B	R(DR)←R(SA) XNOR R(SB)
0	0	0	1	110	TEST A	N, C, V, Z ←R(SA)
1	0	0	1	111	LDI (load OP)	R(DR)←zero fill(OP)
1	0	0	0	000	ADI (add OP)	R(DR)← R(SA) + zero fill(OP)
1	0	0	0	001	SBI (subtract OP)	R(DR)← R(SA) - zero fill(OP)
0	0	1	1	110	LOAD A	R(DR)← M(R(SA))
0	0	0	1	110	MOVE A	MOVE R(DR)←R(SA)
0	0	0	1	111	MOVE B	MOVE R(DR)←R(SB)
0	1	0	1	111	STORE B	M(R(DR))← R(SB)

LSL – logični pomik levo, ki je ekvivalent nepredznačenemu množenju z 2. MSB mesto se prenese v bit prenosa (C).
zero fill – dopolnitev do MSB z ničlami – nepredznačena širitev mest.

Tabela 5: Primera vejitev in brezpogojnega skoka.

15	**14**	**13**	**12**	**11, 10**	**9**		
`PL <= ir(14) and ir(15)`		`JB`	`ALU_mode`	`ALU_function`	`BC se prekriva z ALU_function(0)`	`Koda operacije`	`Izvedba operacije`

1	1	0	1	11	0	BRZ R(SA)	IF (R(SA)=0)
							PC←PC+SXT(AD)
							ELSE
							PC←PC+1

1	1	0	0	00	1	BRN R(SA)	IF (R(SA)=0)
							PC←PC+SXT(AD)
							ELSE
							PC←PC+1

1	1	1	0	00	0	JMP	PC←R(SA)

SXT – razširitev predznaka (ang. sign extension) – predznačena širitev mest.

Naložite samo datoteke:
`cpu_datapath.vhd`,
`branch_ctrl.vhd`,
`cpu.vhd`.
ne vseh datotek. Na strežniku je namreč omejitev števila naloženih datotek kot tudi velikosti posamezne datoteke. Omejitev števila datotek je sicer precej visoka (200), a jo z CTRL+A lahko kaj hitro dosežete.