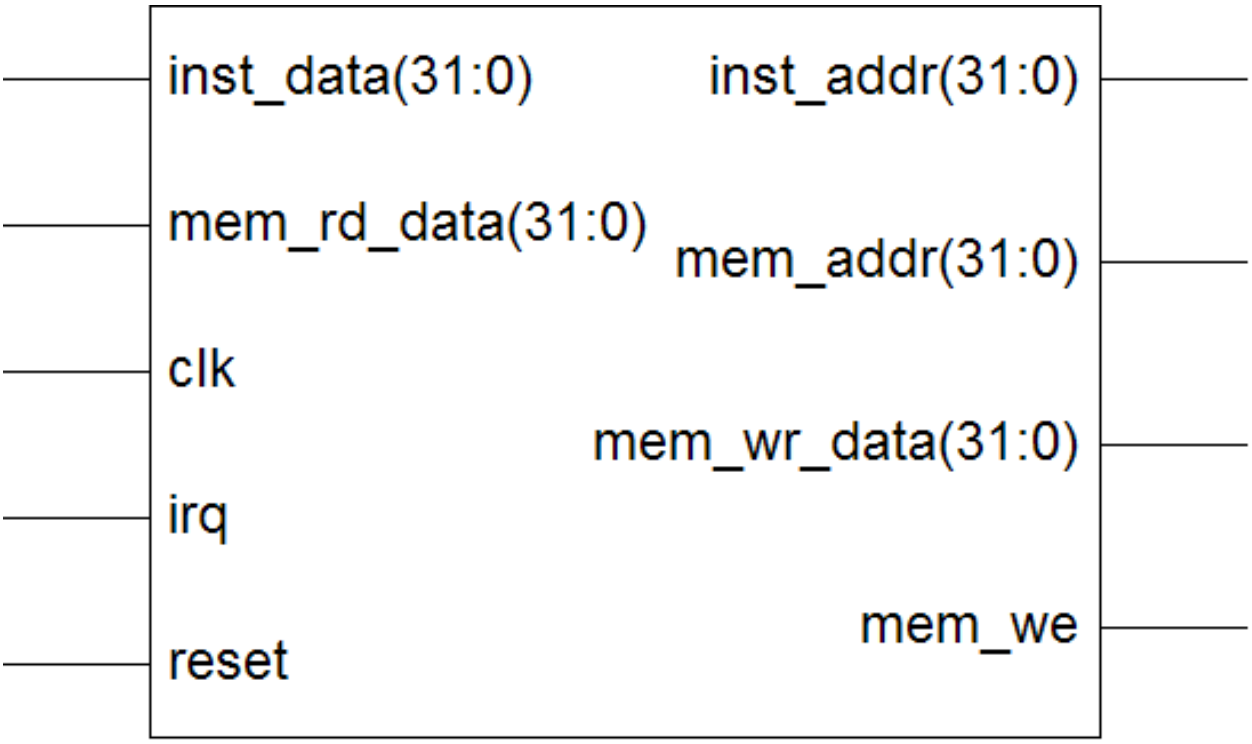
Laboratorul 5 – Implementarea unui System-on-Chip

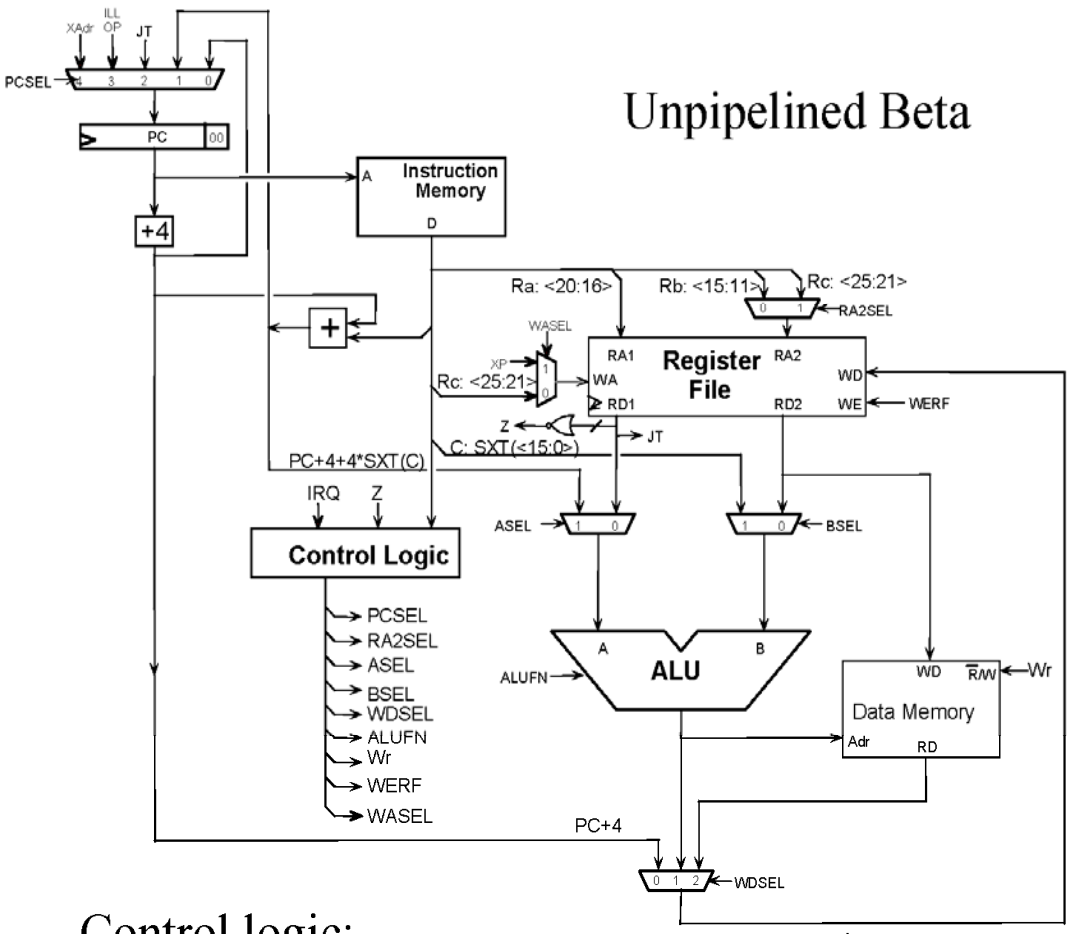
Obiective

Acest laborator isi propune sa prezinte modul de realizare a unui System-on-Chip (SoC). In cadrul laboratorului vor fi prezentate metodele de proiectare a unui sistem folosind elemente constructive deja descrise ce vor fi reutilizate. Scopul acestuia este de a va familiariza cu reutilizarea codului disponibil si cu necesitatile unui sistem de calcul construit cu ajutorul unor componente proiectate independent.
Pentru a obtine punctajul acordat acestui laborator, trebuie sa prezentati laborantului functionarea corecta in hardware a proiectului dumneavoastra.

Proiectarea

Procesorul utilizat se numeste Beta si este un procesor pe 32 de biti cu arhitectura RISC dezvoltat la MIT. Pentru mai multe detalii privind implementarea puteti consulta cursul aferent de la MIT (6.004). Prezentam in continuare interfata acestui procesor si schema bloc:





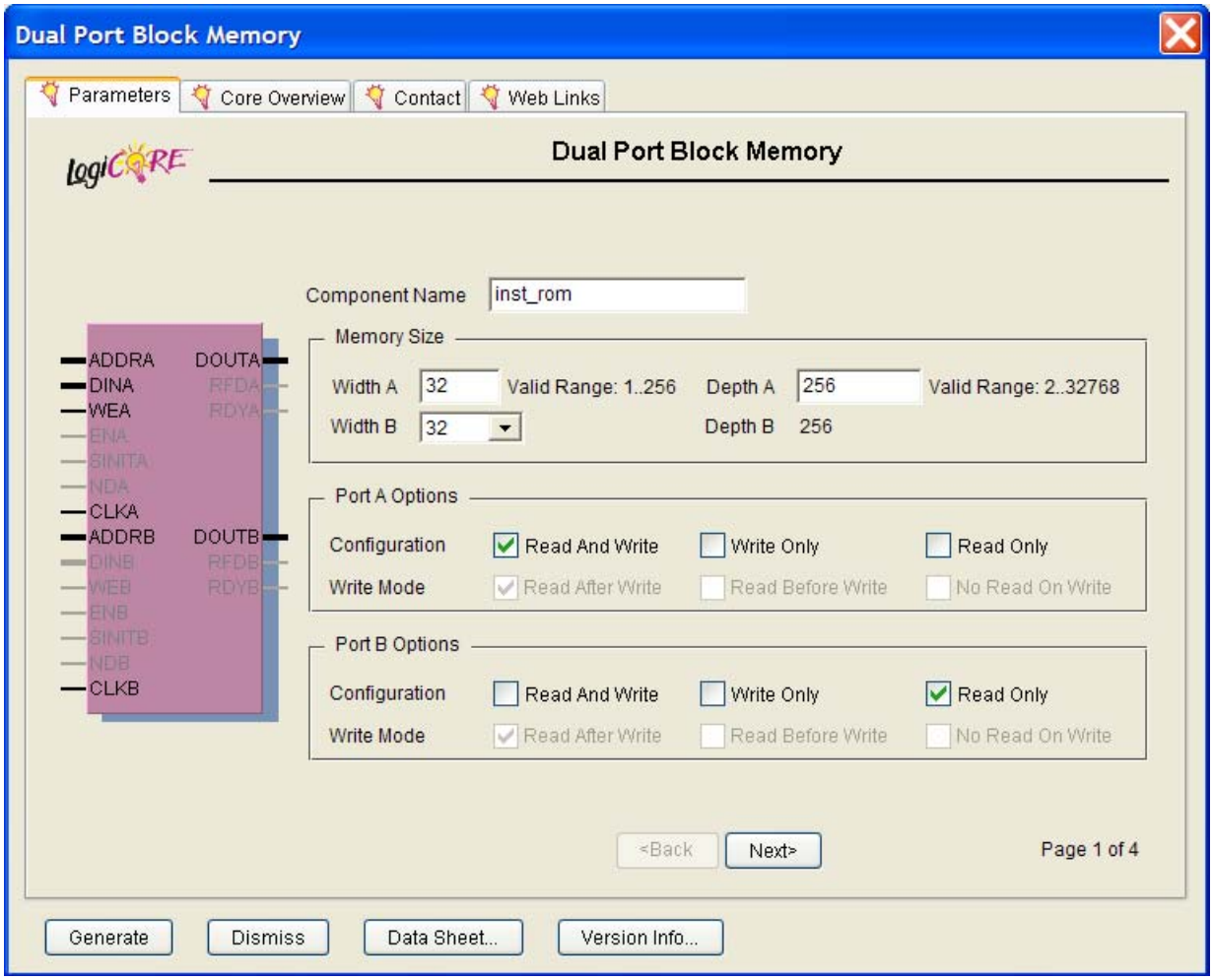
Control logic:

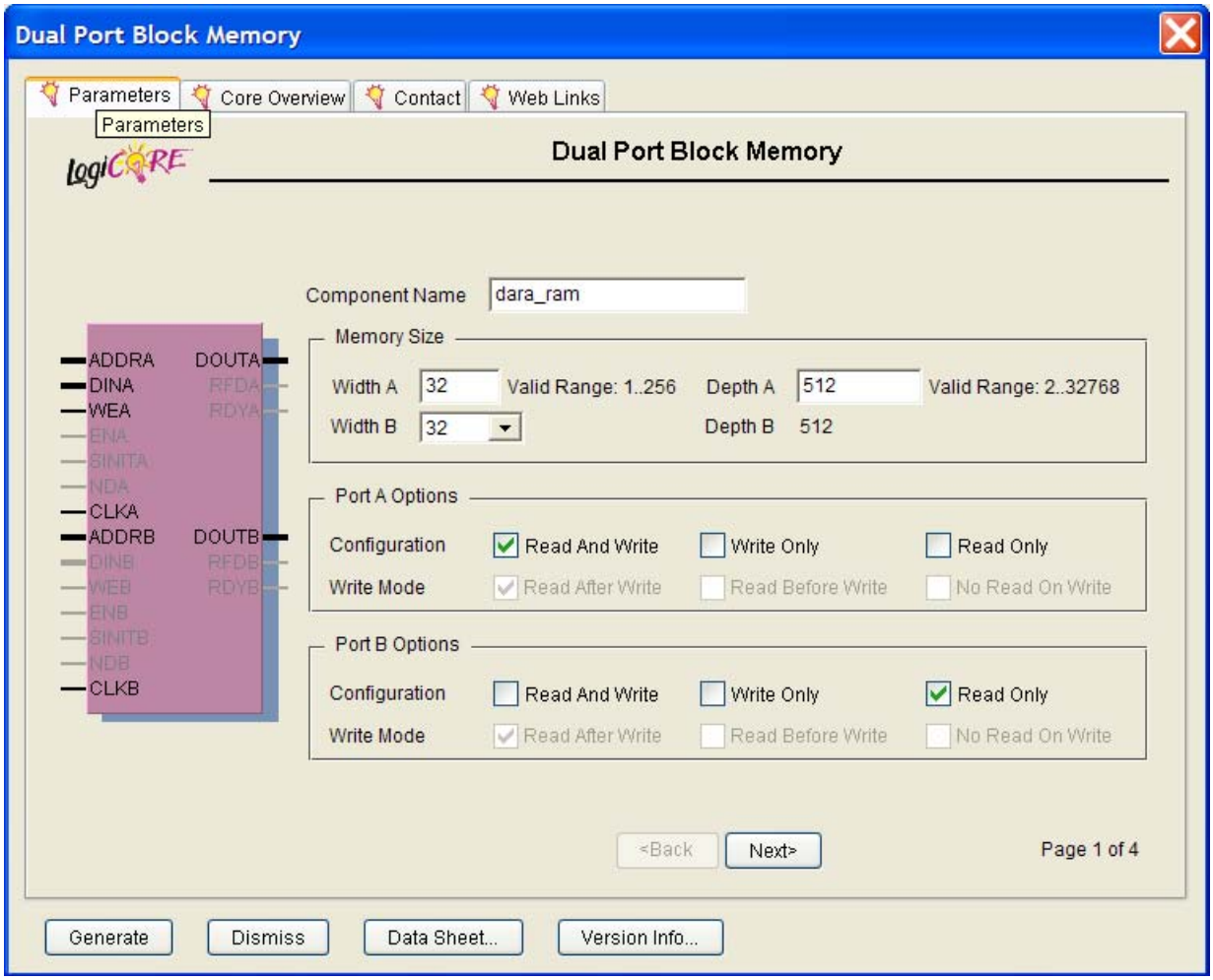
	OP	OPC	LD	ST	JMP	BEQ	BNE	LDR	Illop	IRQ
ALUFN	F(op)	F(op)	"+"	"+"	–	–	–	"A"	–	–
WERF	1	1	1	0	1	1	1	1	1	1
BSEL	0	1	1	1	–	–	–	–	–	–
WDSEL	1	1	2	–	0	0	0	2	0	0
WR	0	0	0	1	0	0	0	0	0	0
RA2SEL	0	–	–	1	–	–	–	–	–	–
PCSEL	0	0	0	0	2	Z ? 1 : 0	Z ? 0 : 1	0	3	4
ASEL	0	0	0	0	–	–	–	1	–	–
WASEL	0	0	0	–	0	0	0	0	1	1

Deoarece ne propunem sa implementam un sistem simplu, nu vom utiliza toate facilitatile oferite de procesorul Beta (nu vom folosi intreruperile si de aceea pe portul irq se va conecta la masa). Pentru a putea analiza efectele executiei instructiunilor de catre procesor, acesta trebuie sa functioneze sincron cu un ceas cu o perioada protrivita. Vom diviza ceasul disponibil cu frecventa de 50MHz (utilizat pentru afisarea VGA) utilizand un contor pe 26 de biti ce se incrementeaza permanent. Bitul cel mai semnificativ al acestui contor va constitui semnalul de ceas pentru procesor. Astfel ceasul este divizat. La procesor se pot conecta doua memorii, una de date si una de instructiuni. Amandoua au latimea cuvantului de 32 de biti. Teoretic, adancimea celor 2 memorii poate fi calculata astfel incat adresa sa fie pe 32 de biti. Deoarece dorim sa afisam in permanenta continutul celor 2 memorii vom face un compromis alegand adancimile de 512 (memoria de date) si 256 (memoria de instructiuni). Aceasta alegere este justificata in paragraful referitor la afisarea memoriilor pe monitor. Din punct de vedere al procesorului memoria de instructiuni este ROM, iar memoria de date este RAM.

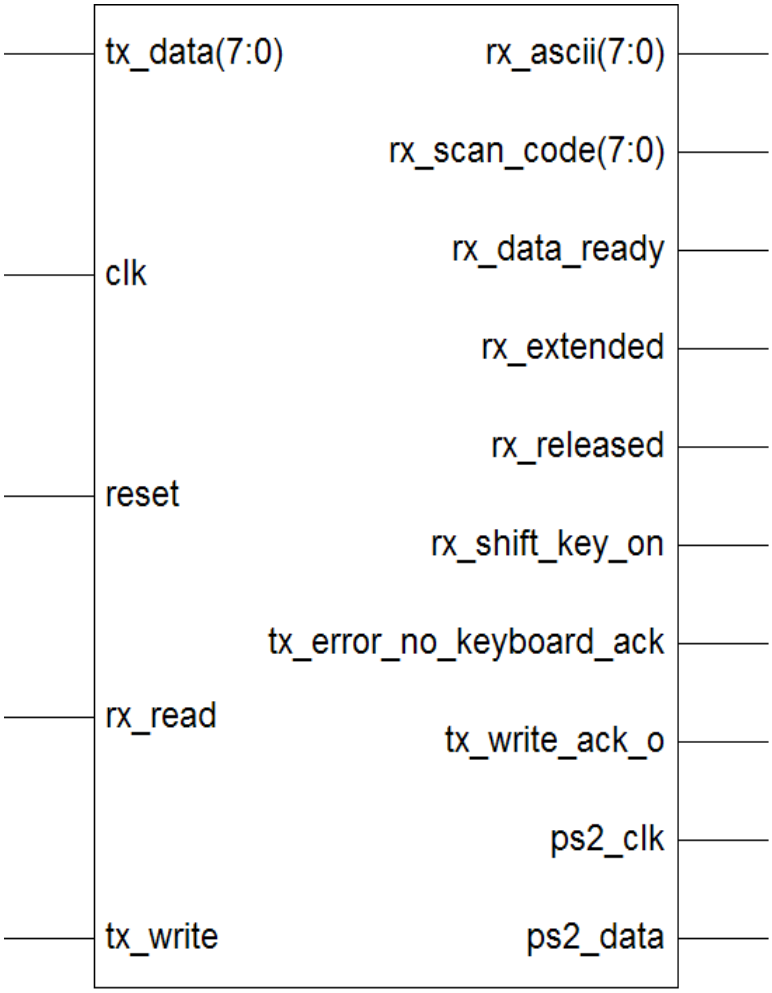
Pentru cele doua memorii vom utiliza IP-uri facilitand astfel utilizarea resurselor specifice disponibile in cadrul FPGA-ului. Deoarece la memorii trebuie sa aiba acces pe de o parte procesorul si pe de alta parte modulul de afisare VGA, vom folosi memorii biport. Desi memoria de instructiuni este din punct de vedere al procesorului o memorie ROM, ea trebuie sa poata fi scrisa inainte de a putea executa programul. Pentru a scrie aceasta memorie vom folosi un mecanism specializat pentru a citi de la tastatura (PS/2) adresa instructiunii si instructiunea ce doreste a fi introdusa in memorie. In concluzie, ambele memorii vor avea portul A de tip RW (in memoria de instructiuni scrie tastatura si citeste procesorul, iar in memoria de date scrie si citeste procesorul), iar portul B va fi de tip RO deoarece modulul de afisare VGA doar citeste continutul memoriilor pentru al afisa.

Atata timp cat se introduc de la tastatura instructiuni in memorie, procesorul nu trebuie sa functioneze si deci magistrala de arese a memoriei de instructiuni (desi partajata) va fi atribuita alternativ in functie de valoarea unui switch extern ce stabileste modul de functionare. Acelasi switch determina si pornirea ceasului procesorului.


Dual Port Block Memory
Parameters
Core Overview
Contact
Web Links
Dual Port Block Memory
Component Name
inst_rom
ADDRA
DINA
WEA
ENA
SINITA
NDA
CLKA
ADDRB
DINB
WEB
ENB
SINITB
NDB
CLKB
DOUTA
RFDA
RDYA
DOUTB
RFDB
RDYB
Memory Size
Width A
32
Valid Range: 1..256
Depth A
256
Valid Range: 2..32768
Width B
32
Depth B
256
Port A Options
Configuration
Read And Write
Write Only
Read Only
Write Mode
Read After Write
Read Before Write
No Read On Write
Port B Options
Configuration
Read And Write
Write Only
Read Only
Write Mode
Read After Write
Read Before Write
No Read On Write
<Back
Next>
Page 1 of 4
Generate
Dismiss
Data Sheet...
Version Info...

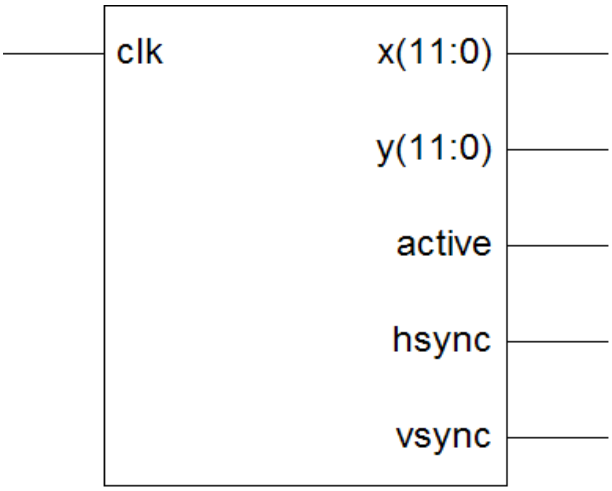

Dual Port Block Memory
Parameters
Core Overview
Contact
Web Links
Parameters
Dual Port Block Memory
Component Name
dara_ram
ADDRA
DINA
WEA
ENA
SINITA
NDA
CLKA
ADDRB
DINB
WEB
ENB
SINITB
NDB
CLKB
DOUTA
RFDA
RDYA
DOUTB
RFDB
RDYB
Memory Size
Width A
32
Valid Range: 1..256
Depth A
512
Valid Range: 2..32768
Width B
32
Depth B
512
Port A Options
Configuration
Read And Write
Write Only
Read Only
Write Mode
Read After Write
Read Before Write
No Read On Write
Port B Options
Configuration
Read And Write
Write Only
Read Only
Write Mode
Read After Write
Read Before Write
No Read On Write
<Back
Next>
Page 1 of 4
Generate
Dismiss
Data Sheet...
Version Info...

Introducerea de la tastatura a instructiunilor in memoria de instructiuni se face prin intermediul unui registru temporar de 40 de biti (primii 8 constituie adresa iar ceilalti 32 instructiunea ce se doreste a fi introdusa). Cand in acest registru se gasesc datele ce se doresc a fi introduse in memorie, cu ajutorul unui buton se transmite comanda de a scrie in memorie (pe busul de adrese se pune adresa, pe busul de intrare de date se pun datele si se activeaza semnalul WE). Astfel la urmatorul front de ceas, datele vor fi scrise in memoria de instructiuni.

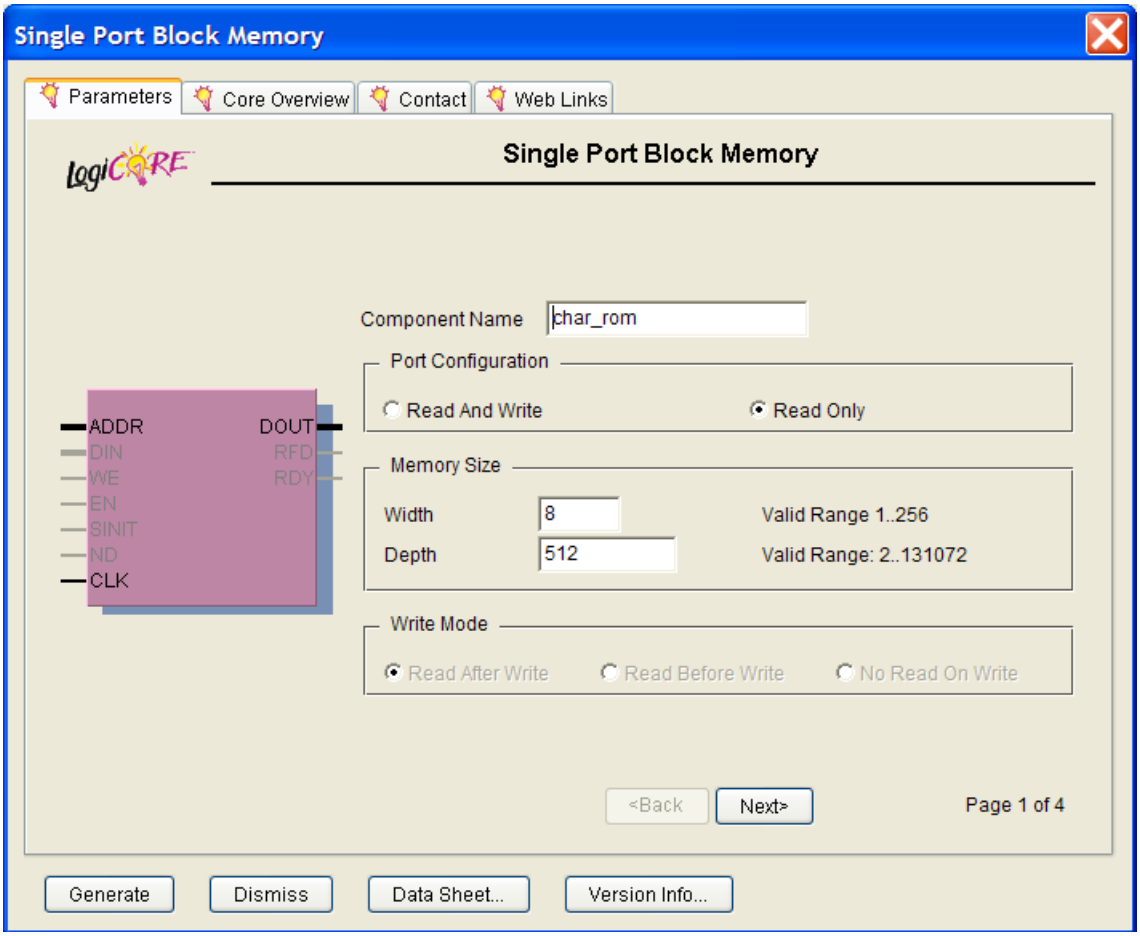


Pentru a citi de la tastatura in registrul temporar se foloseste un controler de tastatura (PS/2) ce poate citi orice tasta. Cand se detecteaza ca s-a citit o tasta aceasta trebuie validata si codul trebuie transformat astfel incat se se accepte numai urmatoarele valori (reprezentabile pe 4 biti): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E si F. Aceste operatii se fac cu ajutorul uinui LookUpTable. Pe masura ce se introduce un caracter, in cadrul registrului temporar se deplaseaza la stanga cu patru pozitii valoarea anterioara si se introduce pe pozitiile cel mai putin semnificative noul cod al caracterului.

Afisarea continutului memoriilor cat si a registrului temporar in care se introduce de la tastatura pe monitor se realizeaza cu ajutorul unui controler VGA care genereaza semanalele de sincronizare necesare si intoarce coordonatele pixelului curent:



Pentru a afisa caractere, avem nevoie de o memorie ROM care sa contina harta caracterelor:



Caracterele sunt reprezentate in matrici de 8x8 pixeli, iar fiecare linie constituie un cuvant in memoria de caractere. Pentru a extrage valoarea pixelului curent, pe baza pozitiei date de X si Y stabilim ce caracter este si ce pixel din matricea caracterului ne intereseaza. Cei mai putin semnificatifi 3 biti din X respectiv Y ne vor spune care pixel din caracterul dorit trebuie afisat. Astfel putem stabili culoarea pixelului curent.

Pentru a putea afisa continutul celor doua memorii am facut compromisul de a alege dimensiunile lor de 256 si respectiv 512 cuvinte. Un cuvant are 32 de biti ce se reprezinta cu ajutorul a 8 caractere (fiecare caracter se poate reprezentat pe 4 biti). In concluzie, un cuvant ocupa 64 de pixeli pe orizontala si 8 pixeli pe verticala. Care caracter din cadrul cuvantului este desemnat de valoarea X[5:3]. Organizam afisarea dupa cum urmeaza: 4 coloane de 64 de biti pe orizontala si cate 64 de cuvinte de 8 biti pe verticala (memoria de instructiuni 4x64=256 cuvinte) si 8 coloane de 64 de biti pe orizontala di cate 64 de cuvinte a 8 biti pe verticala (memoria de date 8x64=512 cuvinte). Reprezentarea memoriilor ocupa 768 de pixeli pe orizontala si 512 pixeli pe verticala. Deoarece rezolutia de afisare corespunzatoare unui ceas de 50MHz este de 800x600 pixeli, ne ramane o zona in partea de jos a ecranului pe care o putem folosi pentru afisarea registrului temporar.

Coloana in care ne aflam se calculeaza pe baza bitilor superiori din X (X[8:6]), iar linia curenta se calculeaza pe baza bitilor superiori din Y (Y[8:3]). Daca afisam din memoria de instructiuni sau din memoria de date se decide pe baza conditiei (X<256). Daca Y[9] este 1 atunci inseamna ca am trecut de limita pe verticala de 512 pixeli si nu mai trebuie sa afisam din continutul memoriei.

Adresele pentru memorii se calculeaza astfel: pentru memoria de instructiuni {X[7:6],Y[8:3]}, iar pentru memoria de date {tmp,X[7:6],Y[8:3]} unde tmp este 1 daca X[9:8]este 2 si 0 altfel.

Utilizarea sistemului

Pentru a utiliza sistemul astfel creat, se scriu programe in cod masina pentru procesorul Beta utilizand documentatia aferenta cursului de la MIT. Programele se introduc in memoria de instructiuni incepand cu adresa 0 la adrese succesive si se executa dupa pornirea procesorului (si trecere din modul de introducere de date in modul executie).

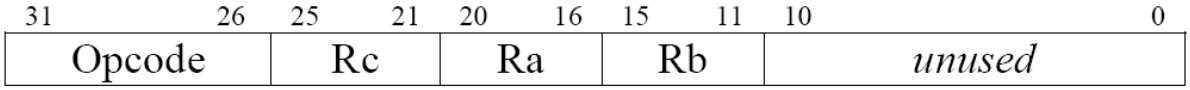
Codificarea instructiunilor

Fiecare instructiune este reprezentata pe 32 de biti. Memoria de date este accesata prin instructiuni de load si store. Instructiunile de ramificare sunt separate de intructiunile de

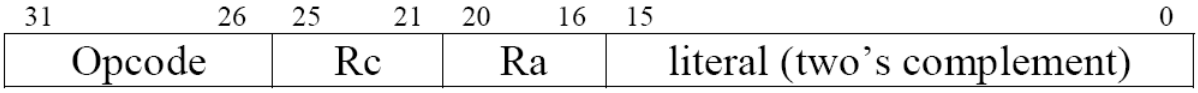
conditionare. Instructiunile de ramificare testeaza valoarea unui registru care poate fi rezultatul unei comparatii anterioare.

Exista doar doua tipuri de instructiuni: Cu Literal si Fara Literal. Instructiunile fara literal sunt instructiunile aritmetice si logice ce au operanzii situati in doua registre si rezultatul este depus in al treilea registru. Instructiunile cu literal sunt toate celelalte instructiuni. Ca si orice alta valoare cu semn in cadrul procesorului Beta, literalul este reprezentat in complement fata de 2.

Fara Literal



Cu Literal



In continuare se prezinta tabelul instructiunilor si codurile operatiilor aferente:

Mnemonic	*Opcode*
ADD	0x20
ADDC	0x30
AND	0x28
ANDC	0x38
BEQ	0x1D
BNE	0x1E
CMPEQ	0x24
CMPEQC	0x34

Mnemonic	*Opcode*
CMPLE	0x26
CMPLEC	0x36
CMPLT	0x25
CMPLTC	0x35
DIV	0x23
DIVC	0x33
JMP	0x1B
LD	0x18

Mnemonic	*Opcode*
LDR	0x1F
MUL	0x22
MULC	0x32
OR	0x29
ORC	0x39
SHL	0x2C
SHLC	0x3C
SHR	0x2D

Mnemonic	*Opcode*
SHRC	0x3D
SRA	0x2E
SRAC	0x3E
SUB	0x21
SUBC	0x31
ST	0x19
XOR	0x2A
XORC	0x3A

Operatiile exacutate sunt prezentate in continuare pentru fiecare instructiune in parte (in ordine alfabetica are mnemonicii):

Usage: ADD(Ra,Rb,Rc)

Operation: PC ← PC + 4

Reg[Rc] ← Reg[Ra] + Reg[Rb]

Usage: ADDC(Ra,literal,Rc)

Opcode:	110000	Rc	Ra	literal

Operation:
PC ← PC + 4
Reg[Rc] ← Reg[Ra] + SEXT(literal)

Usage: AND(Ra,Rb,Rc)

Opcode:	101000	Rc	Ra	Rb	*unused*

Operation:
PC ← PC + 4
Reg[Rc] ← Reg[Ra] & Reg[Rb]

Usage: ANDC(Ra,*literal*,Rc)

Opcode:	111000	Rc	Ra	literal

Operation:
PC ← PC + 4
Reg[Rc] ← Reg[Ra] & SEXT(literal)

Usage: BEQ(Ra,label,Rc)
BF(Ra,label,Rc)

Opcode:	011101	Rc	Ra	literal

Operation:
literal = ((OFFSET(label) – OFFSET(current instruction)) / 4)–1
PC ← PC + 4
EA ← PC + 4*SEXT(literal)
TEMP ← Reg[Ra]
Reg[Rc] ← PC
if TEMP = 0 then PC ← EA

Usage: BNE(Ra,label,Rc)
BT(Ra,label,Rc)

Opcode:	011110	Rc	Ra	literal

Operation:
literal = ((OFFSET(label) – OFFSET(current instruction)) ÷ 4)–1
PC ← PC + 4
EA ← PC + 4*SEXT(literal)
TEMP ← Reg[Ra]
Reg[Rc] ← PC
if TEMP ≠ 0 then PC ← EA

Usage: CMPEQ(Ra,Rb,Rc)

Opcode:	100100	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
if Reg[Ra] = Reg[Rb] then Reg[Rc] ← 1 else Reg[Rc] ← 0

Usage: CMPEQC(Ra,literal,Rc)

Opcode:	110100	Rc	Ra	literal

Operation: PC ← PC + 4
if Reg[Ra] = SEXT(literal) then Reg[Rc] ← 1 else Reg[Rc] ← 0

Usage: CMPLE(Ra,Rb,Rc)

Opcode:	100110	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
if Reg[Ra] ≤ Reg[Rb] then Reg[Rc] ← 1 else Reg[Rc] ← 0

Usage: CMPLEC(Ra,literal,Rc)

Opcode:	110110	Rc	Ra	literal

Operation: PC ← PC + 4
if Reg[Ra] ≤ SEXT(literal) then Reg[Rc] ← 1 else Reg[Rc] ← 0

Usage: CMPLT(Ra,Rb,Rc)

Opcode:	100101	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
if Reg[Ra] < Reg[Rb] then Reg[Rc] ← 1 else Reg[Rc] ← 0

Usage: CMPLTC(Ra,literal,Rc)

Opcode:	110101	Rc	Ra	literal

Operation: PC ← PC + 4
if Reg[Ra] < SEXT(literal) then Reg[Rc] ← 1 else Reg[Rc] ← 0

Usage: DIV(Ra,Rb,Rc)

Opcode:	100011	Rc	Ra	Rb	*unused*

Operation: PC ← PC + 4
Reg[Rc] ← Reg[Ra] / Reg[Rb]

Usage: DIVC(Ra,literal,Rc)

Opcode:

110011	Rc	Ra	literal

Operation:

PC ← PC + 4

Reg[Rc] ← Reg[Ra] / SEXT(literal)

Usage: JMP(Ra,Rc)

Opcode:

011011	Rc	Ra	0000000000000000

Operation:

PC ← PC+4

EA ← Reg[Ra] & 0xFFFFFFFC

Reg[Rc] ← PC

PC ← EA

Usage: LD(Ra,literal,Rc)

Opcode:

011000	Rc	Ra	literal

Operation:

PC ← PC+4

EA ← Reg[Ra] + SEXT(literal)

Reg[Rc] ← Mem[EA]

Usage: LDR(label,Rc)

Opcode:

011111	Rc	11111	literal

Operation:

literal = ((OFFSET(label) – OFFSET(current instruction)) / 4)–1

PC ← PC + 4

EA ← PC + 4*SEXT(literal)

Reg[Rc] ← Mem[EA]

Usage: MUL(Ra,Rb,Rc)

Opcode:

100010	Rc	Ra	Rb	*unused*

Operation:

PC ← PC + 4

Reg[Rc] ← Reg[Ra] * Reg[Rb]

Usage: MULC(Ra,literal,Rc)

Opcode:

110010	Rc	Ra	literal

Operation:

PC ← PC + 4

Reg[Rc] ← Reg[Ra] * SEXT(literal)

Usage: OR(Ra,Rb,Rc)

101001	Rc	Ra	Rb	*unused*

Operation:
PC ← PC + 4
Reg[Rc] ← Reg[Ra] | Reg[Rb]

Usage: ORC(Ra,*literal*,Rc)

111001	Rc	Ra	literal

Operation:
PC ← PC + 4
Reg[Rc] ← Reg[Ra] | SEXT(literal)

Usage: SHL(Ra,Rb,Rc)

101100	Rc	Ra	Rb	*unused*

Operation:
PC ← PC + 4
Reg[Rc] ← Reg[Ra] << $\text{Reg[Rb]}_{4:0}$

Usage: SHLC(Ra,*literal*,Rc)

111100	Rc	Ra	literal

Operation:
PC ← PC + 4
Reg[Rc] ← Reg[Ra] << $\text{literal}_{4:0}$

Usage: SHR(Ra,Rb,Rc)

101101	Rc	Ra	Rb	*unused*

Operation:
PC ← PC + 4
Reg[Rc] ← Reg[Ra] >> $\text{Reg[Rb]}_{4:0}$

Usage: SHRC(Ra,*literal*,Rc)

111101	Rc	Ra	literal

Operation:
PC ← PC + 4
Reg[Rc] ← Reg[Ra] >> $\text{literal}_{4:0}$

Usage: SRA(Ra,Rb,Rc)

101110	Rc	Ra	Rb	*unused*

Operation:
PC ← PC + 4
Reg[Rc] ← Reg[Ra] >> $\text{Reg[Rb]}_{4:0}$

Usage: SRAC(Ra,*literal*,Rc)

Opcode:

111110	Rc	Ra	literal

Operation:

PC ← PC + 4

Reg[Rc] ← Reg[Ra] >> $literal_{4:0}$

Usage: ST(Rc,literal,Ra)

Opcode:

011001	Rc	Ra	literal

Operation:

PC ← PC+4

EA ← Reg[Ra] + SEXT(literal)

Mem[EA] ← Reg[Rc]

Usage: SUB(Ra,Rb,Rc)

Opcode:

100001	Rc	Ra	Rb	*unused*

Operation:

PC ← PC + 4

Reg[Rc] ← Reg[Ra] – Reg[Rb]

Usage: SUBC(Ra,literal,Rc)

Opcode:

110001	Rc	Ra	literal

Operation:

PC ← PC + 4

Reg[Rc] ← Reg[Ra] – SEXT(literal)

Usage: XOR(Ra,Rb,Rc)

Opcode:

101010	Rc	Ra	Rb	*unused*

Operation:

PC ← PC + 4

Reg[Rc] ← Reg[Ra] ^ Reg[Rb]

Usage: XORC(Ra,*literal*,Rc)

Opcode:

111010	Rc	Ra	literal

Operation:

PC ← PC + 4

Reg[Rc] ← Reg[Ra] ^ SEXT(literal)

Un exemplu de implementare este prezentat in continuare:

```
module SoC_top(clk, clr, store, sw, ps_data, ps_clk, hsync, vsync, r2, r1, r0, g2, g1, g0, b2, b1, b0, led, cled);
    input clk;
    input clr;
    input store;
    input sw;
    inout ps_data;
    inout ps_clk;
    output hsync;
    output vsync;
    output r2, r1, r0, g2, g1, g0, b2, b1, b0;
    output [7:0] led;
    output cled;

    reg r2, r1, r0, g2, g1, g0, b2, b1, b0;
    wire active;
    wire [11:0] x;
    wire [11:0] y;
            wire [7:0] led;
            wire cled;
            reg [25:0] cnt;

            wire [0:31] display_word;
            wire [31:0] douti;
            wire [7:0]  addri;
            wire [0:3]          display_byte,display;
            wire [2:0]  dtx,dty;
            wire [2:0]          tmp;
```

```
reg bclk,reset,irq=1'b0;

wire [31:0] inst_addr;
wire [31:0] inst_data;
wire [31:0] mem_addr;
wire [31:0] mem_wr_data;
wire [31:0] mem_rd_data;
wire mem_we;
wire [8:0]  video_rd_addr;
wire [31:0] video_rd_data;

wire [8:0]  addrc;
wire [0:7] doutc;
wire bord,mem_display,ps_display,current_instr,current_data,ind;
wire new_scancode,new_data;
wire [4:0] dsb0,dsb1,dsb2,dsb3;

wire [7:0] scancode;
wire [3:0] data_from_ps2;

reg [3:0] ps_instr [9:0];
reg [3:0] ps_pointer;
reg [3:0] char_pointer;
reg saved;
reg wea,instr_clk;
reg [7:0] addra;
reg [31:0] dina;
```

```
        reg rx_read;

        wire rx_extended,rx_released,rx_shift_key_on,tx_write_ack_o,tx_error_no_keyboard_ack;
        wire [7:0] rx_ascii;

       VGATimer #(800,64,120,56, 600,23,6,37, 1) timer (clk, hsync, vsync, active, x, y);

        char_rom char(addrc,~clk,doutc);

        scancod2cod code(scancode,rx_released,data_from_ps2,new_data);
        ps2_keyboard_interface
ps2(clk,~clr,ps_clk,ps_data,rx_extended,rx_released,rx_shift_key_on,scancode,rx_ascii,new_scancode,rx_read,8'd0,
1'b0,tx_write_ack_o,tx_error_no_keyboard_ack);

        inst_rom inst(addra,addri,instr_clk,~clk,dina,inst_data,douti,wea);
        dara_ram
data(mem_addr[10:2],video_rd_addr,bclk,~clk,mem_wr_data,mem_rd_data,video_rd_data,mem_we);

        beta beta(bclk,reset,irq,inst_addr,inst_data,mem_addr,mem_rd_data,mem_we,mem_wr_data);

        always @(posedge clk)
          rx_read <= (new_data)? saved: ~saved;

        always @(negedge clr or posedge bclk)
                if (~clr) reset <= 1;
                else reset <= 0;

        always @(negedge clr or posedge clk)
                if (~clr) cnt <= 0;
```

```
            else cnt <= cnt+1;


        always @(negedge clr or posedge clk)
            if (~clr) bclk <= 0;
            else if (sw) bclk <= cnt[25];
            else bclk <= 0;


        always @(posedge clk)
          if (new_data) begin
             if (~saved) begin
              saved <= 1;
              ps_instr[ps_pointer] <= data_from_ps2;
              if (ps_pointer<9) ps_pointer <= ps_pointer+1;
              else ps_pointer <= 0;
             end
            end else saved <= 0;


        always @(posedge clk)
            if (store&~sw) begin
              dina                                                                                                    <=
{ps_instr[2],ps_instr[3],ps_instr[4],ps_instr[5],ps_instr[6],ps_instr[7],ps_instr[8],ps_instr[9]};
              addra <= {ps_instr[0],ps_instr[1]};
              wea <= 1;
              instr_clk <= cnt[25];
            end else begin
              addra <= inst_addr[9:2];
              wea <= 0;
              instr_clk <= cnt[25];
```

```
            end

assign addrc = {2'b11,display,dty};

assign addri={x[7:6],y[8:3]};
assign tmp=(x[9:8]==2'b10) ? 1'b1 : 1'b0;
assign video_rd_addr={tmp,x[7:6],y[8:3]};
assign ind = (x<256);
assign display_word=(ind)? douti:video_rd_data;

assign dsb0={x[5:3],2'b00};
assign dsb1={x[5:3],2'b01};
assign dsb2={x[5:3],2'b10};
assign dsb3={x[5:3],2'b11};

assign display_byte={display_word[dsb0],display_word[dsb1],display_word[dsb2],display_word[dsb3]};
assign display= (y[9])? ps_instr[char_pointer]:display_byte;
assign dtx=x[2:0];
assign dty=y[2:0];
assign bord=((x==0)|(x==799)|(x[5:0]==6'b000000)&(y<512)|(y==0)|(y==599)|(y==511));
assign mem_display=(y<=511)&(x<767);
assign ps_display=(y[9])&(ind)&(y[5:3]==3'b101)&((x[8:3]>10)&(x[8:3]<13)|(x[8:3]>15)&(x[8:3]<24));
assign current_instr=(addri==inst_addr[9:2])&(ind);
assign current_data=(video_rd_addr==mem_addr[10:2])&(~ind);

assign cled=cnt[25];//new_data;

always @(posedge clk)
```

```
begin
 case (x[8:3])
     6'd11 : char_pointer <= 4'd0;
     6'd12 : char_pointer <= 4'd1;
     6'd16 : char_pointer <= 4'd2;
     6'd17 : char_pointer <= 4'd3;
     6'd18 : char_pointer <= 4'd4;
     6'd19 : char_pointer <= 4'd5;
     6'd20 : char_pointer <= 4'd6;
     6'd21 : char_pointer <= 4'd7;
     6'd22 : char_pointer <= 4'd8;
     6'd23 : char_pointer <= 4'd9;
     default : char_pointer <= 4'd0;
   endcase
 r2 <= active&&((ps_display&&doutc[dtx]&&(ps_pointer>char_pointer))||(mem_display&&doutc[dtx]&&~current_data))
;
   r1 <= active&&((ps_display&&doutc[dtx])||(mem_display&&doutc[dtx]&&ind&&~current_data));
 r0 <= active&&((ps_display&&doutc[dtx])||(mem_display&&doutc[dtx]&&(x[6]^^ind)&&~current_data));
 g2 <= active&&((ps_display&&doutc[dtx]&&(ps_pointer>char_pointer))||(mem_display&&doutc[dtx]&&~current_instr))
;
   g1 <= active&&((ps_display&&doutc[dtx])||(mem_display&&doutc[dtx]&&ind&&~current_instr));
   g0 <= active&&((ps_display&&doutc[dtx])||(mem_display&&doutc[dtx]&&(x[6]^^ind)&&~current_instr));
 b2 <= active&&((ps_display&&doutc[dtx]&&(ps_pointer>char_pointer))||(mem_display&&doutc[dtx]&&~current_instr
&&~current_data)||bord);
```

```
                b1                                                                                                  <=
active&&((ps_display&&doutc[dtx])||(mem_display&&doutc[dtx]&&ind&&~current_instr&&~current_data)||bord
);

            b0                                                                                                      <=
active&&((ps_display&&doutc[dtx])||(mem_display&&doutc[dtx]&&(x[6]^^ind)&&~current_instr&&~current_dat
a)||bord);

              end

endmodule
```