Laboratorul 03
Memorii

1. Obiective

Descrierea, implementarea și testarea:

- **Bloc de regiștri - Register File**
- **Memorii ROM - Read only Memories**
- **Memorii cu acces aleatoriu RAM - Random Access Memories**

Aprofundarea cunoștințelor legate de:

- Vivado Webpack
- Xilinx Vivado Design Suite User Guide
- Digilent Development Boards **(DDB)**
  - **Digilent Basys Board – Reference Manual**
- Artix 7 FPGA

2. Fundamente teoretice

2.1. Blocul de regiștri / Register File

Blocul de regiștri reprezintă spațiul central de stocare dintr-un procesor.

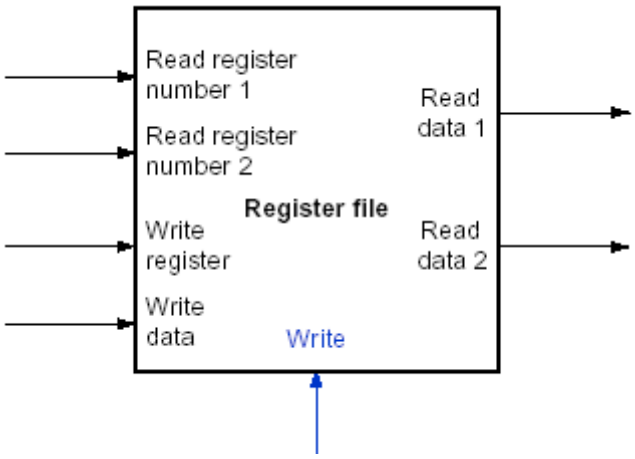

Figura 1: Un bloc de regiștri cu două porturi de citire și un port de scriere

Majoritatea operațiilor CPU implică folosirea sau modificarea datelor stocate în blocul de regiștrii. Deoarece blocul de regiștri funcționează la frecvența de ceas a procesorului, el este limitat ca mărime și trebuie să fie foarte rapid. În aplicațiile reale, un bloc de regiștri este implementat ca o memorie rapidă de tip SRAM, care permite acces multiplu simultan.

Blocul de regiștri specific procesorului MIPS (vezi figura anterioară) are două adrese de citire (*Read register number 1* și *2*) și o adresă de scriere (*Write register*). Regiștrii care corespund locațiilor indicate de cele două adrese de citire sunt livrați pe cele două porturi de ieșire *Read data 1* și *2*. Datele furnizate pe portul de scriere *Write data* sunt scrise în regiștrul indicat de adresa de scriere, dacă semnalul de control *Write* este activat. Operațiile de citire sunt asincrone (combinațional) iar operația de scriere este sincronă (front crescător). În contextul procesorului MIPS, blocul de regiștri suportă două citiri și o scriere în fiecare ciclu de ceas.

În anexa 5 este prezentată o posibila descriere în VHDL pentru un bloc de regiștri. Se va evita CTRL+C, CTRL+V…

2.2. Memorii ROM și RAM

Memoriile de tip ROM sunt o variantă particulară de stocare folosită în calculatoare, ele permițând doar operații de citire în regimul uzual de utilizare. Memoriile cu acces aleatoriu RAM reprezintă o altă variantă de stocare, prezentă sub formă de circuite integrate care permit atât citirea cât și scrierea în locațiile de memorie, cu aproximativ aceeași întârziere indiferent de ordinea de accesare a locațiilor. Aceste două tipuri de memorii sunt esențiale pentru orice procesor.

Un dispozitiv FPGA vine de obicei echipat cu un anumit volum de memorie BRAM (Block RAM). Un BRAM poate fi configurat fie ca un ROM, fie ca un RAM. În funcție de cum se scrie descrierea cu cod VHDL, unealta Xilinx XST poate infera circuitul RAM descris ca o memorie distribuită sau îl poate mapa direct pe un bloc BRAM. Memoriile distribuite sunt construite cu regiștrii, iar memoriile BRAM sunt mapate pe blocurile BRAM disponibile. Memoriile RAM distribuite consumă direct din porțile FPGA și scad frecvența de ceas, în timp ce în cazul memoriilor BRAM rămâne mai mult spațiu pe FPGA pentru logică auxiliară.

Tipul de memorie RAM inferată depinde de descrierea VHDL:

- Descrierea RAM cu citire asincronă va genera o memorie RAM distribuită.
- Descrierea RAM cu citire sincronă va genera o memorie BRAM.

XST acoperă următoarele caracteristici pentru RAM:

- Scriere sincronă
- Validarea scrierii
- Activarea RAM
- Citire sincronă sau asincronă
- Resetare pentru latch-urile de ieșire
- Resetarea datelor de ieșire
- Citire unică, dual sau multi-port
- Scriere unică sau dual
- Biți de paritate
- Etc.

Există trei moduri posibile pentru implementarea unei memorii RAM sincrone: write-first, read-first și no change. Aceste moduri sunt legate de felul cum este stabilită prioritatea pentru operațiile de citire, respectiv de scriere. O posibilă descriere VHDL pentru o memorie RAM cu modul “no change” este prezentată în Anexa 5.

To Do:

1. Accesați Language templates - VHDL → Synthesis Constructs → Coding Examples → RAM și vedeți diferențele de descriere pentru RAM distribuit și Bloc RAM.
2. Accesați Language templates - VHDL → Synthesis Constructs → Coding Examples → RAM → Block RAM → Single port pentru a vedea comparativ RAM cu read-first, respectiv cu write-first.

   (!) Concentrați-vă pe declarare și pe procesul care descrie comportamentul blocului RAM și ignorați restul codului din templates.

2.3. Declararea unui șir în VHDL

Mai jos se prezintă un exemplu de declarație și inițializare pentru un șir, care se poate folosi pentru memorii ROM, RAM, respectiv pentru un bloc de regiștri.

Mai întâi se declară un tip de șir care are N locații de câte M biți fiecare:

type <arr_type> is array (0 to N-1) of std_logic_vector(M-1 downto 0);

În continuare se declară un semnal de tipul declarat anterior:

signal r_name: <arr_type>;

Când se implementează un ROM este obligatoriu ca semnalul respectiv să fie inițializat. Opțional, se pot inițializa și memoriile RAM, respectiv blocurile de regiștri.

```
signal r_name: <arr_type> := (
“00…0”, -- M biți, folosiți reprezentarea hexazecimala când e posibil
“00…1”, --
others => “00…0” -- de dimensiunea unei locații
);
```

3. Activități practice

Este obligatoriu ca la începutul acestui laborator sa aveți un proiect funcțional care sa coincidă cu descrierea din laboratorul 2, secțiunea 3 (să conțină MPG și SSD, instanțiate în entitate top level, numită opțional *test_env*, unde se va scrie codul din acest laborator).

Folosiți **RTL Schematic** după finalizarea fiecărei activități, pentru o primă verificare înainte de încărcarea pe placă.

3.1. Implementarea memoriei ROM

Includeți o memorie ROM de 256x16 biți în proiectul test_env, in entitatea top level (nu declarați o nouă entitate!). Inițializați ROM-ul cu câteva valori arbitrare (vezi 2.3). Folosiți un numărător pe 8-biți pentru a genera adresa pentru ROM. Acest numărător va fi controlat prin intermediul MPG. Conținutul memoriei ROM de la adresa selectată se va afișa pe SSD. Comportamentul ROM-ului este asincron (se descrie într-o linie de cod). Vedeți figura de mai jos.

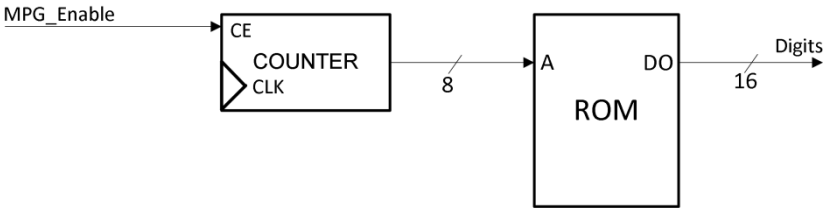

Figura 2: Schema simplă cu o memorie ROM

3.2. Implementarea Blocului de Regiștri

(!) Comentați, NU ștergeți codul de la activitatea anterioară!

Proiectați și implementați un bloc de regiștri RF (vezi 2.1) pentru placa Basys. Descrieți o nouă componenta (entitate) pentru blocul de regiștrii, în proiectul test_env. Schema în care să fie inclus blocul de regiștri RF este prezentată în figura următoare.

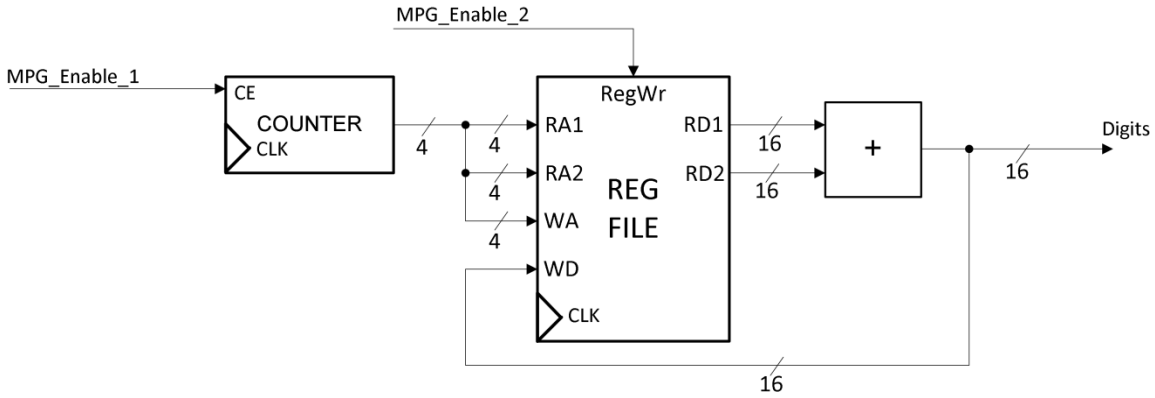

Figura 3: O schemă simplă pentru utilizarea blocului de regiștri

Folosiți un numărător pentru a genera adresele de citire și de scriere ale RF. Numărătorul este controlat de o componentă MPG. Ieșirile RF sunt adunate, iar suma este (1) afișată pe afișorul SSD și (2) este scrisă înapoi în RF. Este necesară utilizarea unei noi instanțe a MPG (port map, cu un alt buton) pentru a activa semnalul de scriere *RegWr* al RF. Circuitul obținut practic seamănă cu un circuit de înmulțire cu 2 (a + a = 2a).

Adăugați pentru numărătorul care generează adresele un mecanism de resetare asincronă pe unul din butoanele nefolosite. Astfel, după parcurgerea primelor adrese din RF, puteți reveni la adresa 0 și verifica dacă în urma primei parcurgeri s-a scris valoarea dublata în RF.

Testați pe placă!

Adăugați anumite elemente auxiliare la acest circuit pentru a utiliza un singur buton (atât pentru MPG, cât și pentru RegWr). Circuitul extins ar trebui să funcționeze la fel ca înainte din punct de vedere al dublării locațiilor din RF, și al iterării prin RF.

3.3. Memorie RAM

Înlocuiți (comentați) blocul de regiștri de la punctul anterior cu o memorie RAM. Folosiți un circuit de deplasare la stânga cu 2 poziții în loc de adunare (figura 3, deplasarea se face cu concatenare). Folosiți un singur port de adresă pentru memoria RAM, implementarea fiind cu modul write-first.

4. Referințe

[1] Manual de referință pentru placa Basys 3 (Artix 7), disponibil pe site la Xilinx
[2] Xilinx Vivado WebPACK – aici.
[3] Help online pentru VHDL, http://vhdl.renerta.com/
[4] Vivado Design Suite User Guide, Appendix C: HDL Coding Techniques
[5] XAPP463 (v2.0) March 1, 2005 Using Block RAM in Spartan-3 Generation FPGAs

Anexa 4 – Implementarea Blocului de Regiștri (specific MIPS)

```
entity reg_file is
	port (
		clk	: in std_logic;
		ra1	: in std_logic_vector (2 downto 0);
		ra2	: in std_logic_vector (2 downto 0);
		wa	: in std_logic_vector (2 downto 0);
		wd	: in std_logic_vector (7 downto 0);
		wen	: in std_logic;
		rd1	: out std_logic_vector (7 downto 0);
		rd2	: out std_logic_vector (7 downto 0)
	);
end reg_file;

architecture Behavioral of reg_file is

	type reg_array is array (0 to 7) of std_logic_vector(7 downto 0);
	signal reg_file : reg_array;

begin
	process(clk)
	begin
		if rising_edge(clk) then
			if wen = '1' then
				reg_file(conv_integer(wa)) <= wd;
			end if;
		end if;
	end process;

	rd1 <= reg_file(conv_integer(ra1));
	rd2 <= reg_file(conv_integer(ra2));

end Behavioral;
```

Anexa 4 – Implementare RAM – no change

Memorie RAM cu modul “no change”: în timpul scrierii, ieșirea din blocul de RAM nu se schimbă (rămâne ce era pe tactul anterior).

```
entity rams_no_change is
	port (	clk	: in std_logic;
		we	: in std_logic;
		en	: in std_logic;
		addr	: in std_logic_vector(5 downto 0);
		di	: in std_logic_vector(15 downto 0);
		do	: out std_logic_vector(15 downto 0));
end rams_no_change;

architecture syn of rams_no_change is

	type ram_type is array (0 to 63) of std_logic_vector (15 downto 0);
	signal RAM: ram_type;
begin

	process (clk)
	begin
		if clk'event and clk = '1' then
			if en = '1' then
				if we = '1' then
					RAM(conv_integer(addr)) <= di;
				else
					do <= RAM( conv_integer(addr));
				end if;
			end if;
		end if;
	end process;
end syn;
```