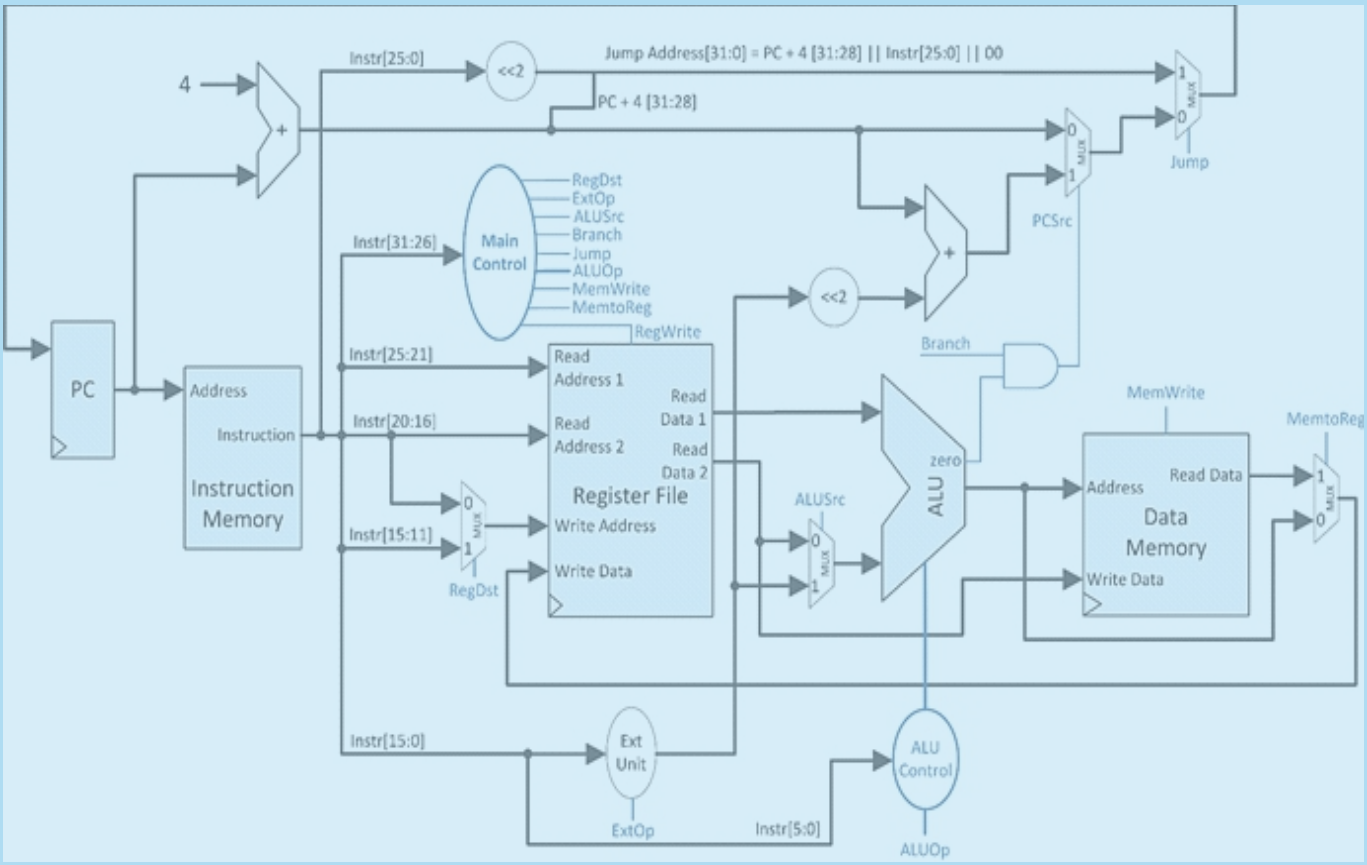
Florin ONIGA **Mihai NEGRU**

ARHITECTURA CALCULATOARELOR

Îndrumător de laborator



UTPRESS
Cluj-Napoca, 2019
ISBN 978-606-737-350-9

Florin ONIGA **Mihai NEGRU**

ARHITECTURA CALCULATOARELOR

Îndrumător de laborator

Editura UTPRESS
Cluj-Napoca, 2019
ISBN 978-606-737-350-9

Editura U.T.PRESS
Str.Observatorului nr. 34
400775 Cluj-Napoca
Tel.:0264-401.999 / Fax: 0264 - 430.408
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Director: Ing. Călin D. Câmpean

Recenzia: Prof.dr.ing. Radu DĂNESCU
Conf.dr.ing. Tiberiu MARIŢA

Copyright © 2019 Editura U.T.PRESS
Reproducerea integrală sau parţială a textului sau ilustraţiilor din această carte este posibilă numai cu acordul prealabil scris al editurii U.T.PRESS.

ISBN 978-606-737-350-9
Bun de tipar: 18.02.2019

Prefață

Acest îndrumător de laborator se adresează în principal studenților din anul 2, ciclul de licență, domeniul Calculatoare și Tehnologia Informației, din cadrul Facultății de Automatică și Calculatoare. Îndrumătorul poate fi util oricui dorește să înțeleagă noțiuni fundamentale despre Arhitectura Calculatoarelor prin construirea și testarea unui procesor de tip MIPS pe 16 biți, folosind limbajul de descriere hardware VHDL, în variantele ciclu unic și pipeline.

Îndrumătorul conține 12 lucrări de laborator și mai multe anexe. Primele 3 laboratoare sunt introductive, construindu-se bazele necesare pentru dezvoltarea procesorului. Procesorul MIPS, în varianta ciclu unic, este dezvoltat gradual în laboratoarele 4-8. În laboratoarele 9-10 se dezvoltă versiunea pipeline a procesorului MIPS, pornind de la versiunea ciclu unic construită anterior. În ultimele două lucrări de laborator se implementează protocolul UART folosind automate cu stări finite și se integrează acest protocol de comunicație cu procesorul MIPS.

Aceasta este prima versiune publicată în limba română a îndrumătorului (versiunea anterioară, în engleză, a fost publicată în 2015). Autorii îi mulțumesc în special domnului profesor Gheorghe Farkas, pe care l-au avut ca mentor în domeniul Arhitecturii Calculatoarelor și care a avut o contribuție semnificativă în predarea acestui subiect pentru o perioadă de 15+ ani. De asemenea, autorii le mulțumesc colegilor, foști sau actuali, pentru observațiile constructive care au dus la îmbunătățirea conținutului acestui îndrumător.

Autorii vă doresc o lectură plăcută și activă!

Cuprins

Obiectivele generale ale laboratorului de AC

Obiectivul principal al acestui laborator este dezvoltarea și testarea de procesoare didactice de tip MIPS folosind limbajul de descriere hardware VHDL, mediul de dezvoltare Xilinx Vivado Webpack și plăci de dezvoltare Digilent (Digilent Development Boards – DDB). Parcurgerea completă și corectă a activităților din fiecare laborator și a temelor suplimentare este obligatorie pentru asimilarea și înțelegerea conceptelor prezentate.

Principalele teme abordate sunt:

- Proiectare cu uneltele Xilinx Vivado Webpack şi – DDB
- Proiectarea de componente hardware VHDL sintetizabile, implementate şi testate pe DDB
- Proiectarea, descrierea în VHDL și testarea pe DDB a uneia sau a mai multor variante de procesor MIPS (ciclu unic și pipeline)
- Înțelegerea arhitecturilor de tip ciclu unic și pipeline
- Implementarea protocolului de comunicație serială și integrarea acestuia cu procesoarele construite.

Laboratorul 1

1. Introducere în mediul software/hardware de dezvoltare VHDL

1.1. Obiective

Familiarizarea studenților cu:

- **Xilinx Vivado WebPack** – ***Vivado Design Suite User Guide***
- **Xilinx® Synthesis Technology (XST)**
- **Xilinx Artix7 FPGA**
- Plăci de dezvoltare Digilent - Digilent Development Boards **(DDB)**
  - Digilent Basys 3 Board – Reference Manual.

1.2. Resurse necesare (uneltele sunt instalate deja pe stațiile de lucru)

Placa de dezvoltare Digilent:

- Basys 3 – Artix 7, Manualul de referință [1].

Mediul de dezvoltare:

- Xilinx VIVADO WebPACK e parte a Vivado Design Suite – HLx Editions.

Pentru VHDL - experiența anterioară de la disciplinele de specialitate sau http://vhdl.renerta.com/.

1.3. Reguli de scriere în VHDL

Pentru a crește eficiența de a scrie cod și, mai ales, pentru a reduce semnificativ riscul de a greși, se vor respecta următoarele reguli:

- Nu veți crea o nouă entitate pentru orice circuit. Nu creați entități separate pentru circuite simple (de bază): porți logice, bistabili, multiplexoare, numărătoare, decodificatoare etc.
- (similar) Nu abuzați de descrierea structurală până la nivel de granularitate de porți logice!
- Bazați-vă în principal pe descrierea comportamentului, ținând cont permanent de ce tip de circuit se va sintetiza din descrierea făcută
- Veți crea entități noi doar pentru părțile semnificative ale proiectului (acest lucru se menționează explicit în lucrările de laborator)
- Nu descrieți niciodată un proces atât de complicat încât să nu știți sigur ce tip de circuit se va sintetiza. Mențineți descrierea simplă, spre circuitele uzuale.

Semnalele se vor declara, între *architecture* și *begin*, folosind strict tipul *std_logic*, respectiv *std_logic_vector*:

- Declararea unui semnal de 1 bit

```
signal sig_name : std_logic := '0';
```

- Declararea unui semnal de N biți

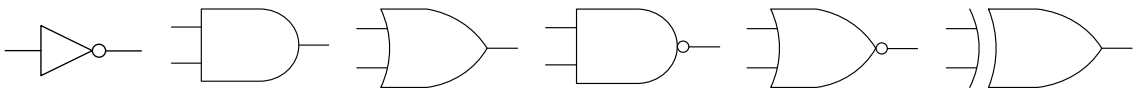
```
signal sig_name : std_logic_vector(N-1 downto 0) := "00….0";
```

- Inițializare, exemple
  - semnal16 biți, binar "0000000000000000";
  - semnal 32 biți, baza 16 x"00000000";
  - semnal N biți, binar (others => '0');

1.4. Circuite de bază

1.4.1. Porțile logice

Acestea sunt cele mai simple circuite combinaţionale. Fiecare tip de poartă efectuează operația logică asociată, conform regulilor algebrei booleene.



NOT / NEGARE | AND / ȘI | OR / SAU | NAND / ȘI NEGAT | NOR / SAU NEGAT | XOR / SAU EXCLUSIV

Figura 1-1: Diagramele porților logice

A	NOT
0	1
1	0

A	B	AND	OR	NAND	NOR	XOR
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0

Tabel 1.1: Tabele de adevăr, A și B sunt operanzi de intrare

Descrierea în VHDL a unei porți logice se face prin atribuire concurentă, în afara proceselor, fără entități suplimentare. Exemple:

```
O <= not A;
O <= A and B;
O <= A or B;
O <= A nand B;
O <= A nor B;
O <= A xor B;
```

1.4.2. Multiplexorul

Multiplexorul (mux) este un circuit care permite selecția unuia dintre mai multe semnale de intrare, care este transmis mai departe pe ieșirea multiplexorului. Un mux cu 2^n intrări are n biți de selecție care vor alege care intrare va fi transmisă mai departe spre ieșire.

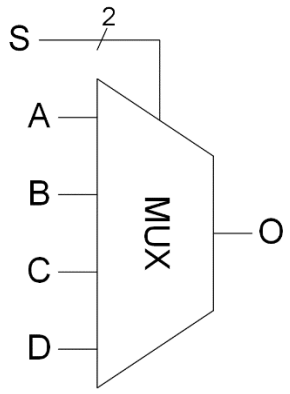


Figura 1-2: Mux 4-la-1

Pini intrare/ieșire	Descriere
A, B, C, D	Intrările de date
S	Selecția
O	Ieșirea de date

Tabel 1.2: Mux 4-la-1, descrierea pinilor

XST permite mai multe stiluri pentru a descrie un multiplexor: în *process* cu if/then/else sau cu case, sau concurențial cu when/else. Exemple:

- **Mux 2:1**

```
O <= A when S = '0' else B;
```

Sau

```
process(S, A, B)
begin
   if(S = ‘0’) then
      O <= A;
   else
      O <= B;
   end if;
end process;
```

- **Mux 4:1**

```
process(S, A, B, C, D)
begin
   case S is
      when "00"    => O <= A;
      when "01"    => O <= B;
      when "10"    => O <= C;
      when others => O <= D;
   end case;
end process;
```

1.4.3. Decodificatorul

Ca definiție generală, decodificatorul este un circuit combinaţional care are N intrări și M ieșiri de un bit, valorile de pe cei N biți de intrare fiind transformate în valori corespondente pe cei M biți de ieșire. Varianta uzuală este cea de decodificator n-la-2^n. Pentru varianta n-la-2^n, cei n biți de intrare au rol de selecție și vor activa doar una dintre ieșiri (cea cu indexul egal cu valoarea de pe intrare), restul rămânând inactive.

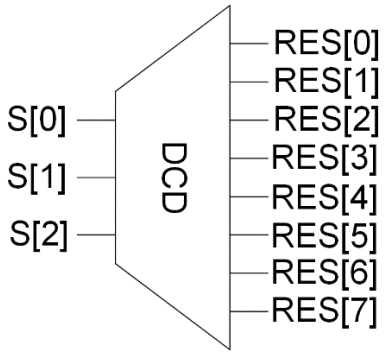


Figura 1-3: Decodificator 3-la-8

Pini intrare/ieșire	Descriere
S	Selector
RES	Ieșire de date

Tabel 1.3: Decodificator 3-la-8, descrierea pinilor

Un mod uzual de a descrie un decodificator este cu case (în *process*):

```
process(S)
begin
   case S is
      when "000"    => RES <= “00000001”;
      when “001"    => RES <= “00000010”;
      when “010"    => RES <= “00000100”;
      when “011"    => RES <= “00001000”;
      when “100"    => RES <= “00010000”;
      when “101"    => RES <= “00100000”;
      when “110"    => RES <= “01000000”;
      when others   => RES <= “10000000”;
   end case;
end process;
```

1.4.4. Bistabilul D (D Flip-Flop)

Bistabilul D este un circuit electronic care se poate afla doar într-una dintre cele două stări stabile, iar tranziția între cele două stări se poate face numai pe frontul semnalului de ceas. Bistabilul D poate memora un bit de date. XST recunoaște din descrierea VHDL bistabili D cu următoarele posibilități de control: Set/Reset asincron, Set/Reset sincron sau activare a ceasului. În acest îndrumător se folosesc implicit bistabili D cu scriere pe frontul crescător/pozitiv al ceasului. Circuitul de tip registru funcționează la fel ca bistabilul D, dar este pe mai mulți biți.

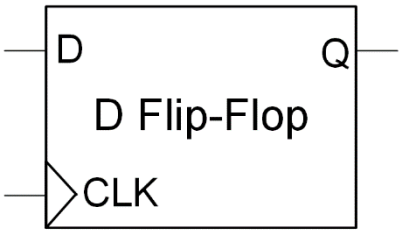


Figura 1-4: Diagrama pentru bistabilul D

Pini intrare/ieșire	Descriere
D	Semnalul de intrare (date)
CLK	Semnalul de ceas
Q	Semnalul de ieșire (date)

Tabel 1.4: Descrierea pinilor pentru bistabilul D

Fiind circuit secvențial, cu stare, descrierea unui bistabil D se face într-un *process*. Pentru a identifica frontul crescător al semnalului de ceas, se pot folosi variantele de mai jos:

```
if (CLK'event and CLK='1') then…
        sau
if rising_edge(CLK) then…
```

Exemplu de **bistabil D sincron**:

```
process(clk)
begin
   if rising_edge(clk) then
      Q <= D;
   end if;
end process;
```

1.4.5. Numărătorul

Numărătorul este un circuit care numără de câte ori apare un eveniment particular. Evenimentul este de obicei definit în strânsă legătură cu semnalul de ceas, un exemplu tipic de eveniment fiind frontul crescător de ceas. XST recunoaște numărătoare cu semnale de control pentru: set/reset asincron, set/reset sincron, încărcare (Load) asincronă/sincronă, activare a numărării și direcția de numărare (în sus, în jos, selectiv sus/jos).

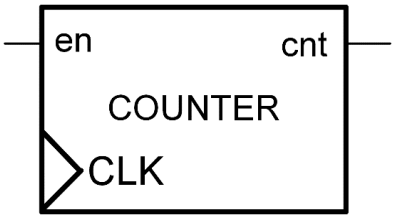


Figura 1-5: Numărător (generic ca număr de biți), crescător, cu activarea numărării

Pini intrare/ieșire	Descriere
CLK	Semnalul de ceas
en	Semnal de activare/validare a numărării
cnt	Ieșirea de date

Tabel 1.5: Numărător, crescător, cu activarea numărării, descrierea pinilor

Exemplu de descriere în VHDL:

```
process(clk, en)
begin
   if rising_edge(clk) then
      if en = '1' then
         cnt <= cnt + 1;
      end if;
   end if;
end process;
```

1.5. Activități practice

Notă: dacă este necesar, consultați help-ul online pentru VHDL indicat la **Resurse**. Nu începeți activitățile până nu ați studiat cu atenție paginile anterioare.

1.5.1. Implementați un proiect VHDL simplu în Xilinx Vivado prin parcurgerea atentă și completă a tutorialului descris în Anexa 1.

1.5.2. Adăugați un numărător binar pe 16 biți, direcțional, (binary up/down simple 16-bit counter) în proiectul test_env, prin descrierea comportamentului numărătorului în arhitectura entității test_env. Încercăm controlarea de la un buton a numărătorului.

Mai întâi, declarați un semnal de 16 biți (tip STD_LOGIC_VECTOR) în arhitectură, înainte de begin. Dacă este necesar (până când vă redobândiți capacitatea de descriere în VHDL), folosiți Language Templates (Anexa 1) pentru a extrage descrierea comportamentală a numărătorului.

Folosiți unul dintre butoanele din porturile entității pentru a controla ceasul numărătorului, ca un semnal de activare a ceasului (se va folosi un if suplimentar inclus în corpul if-ului care testează frontul crescător de ceas, pentru a testa condiția ca semnalul butonului să fie 1).

Folosiți unul dintre switch-uri pentru a controla direcția de numărare (if suplimentar).

Transmiteți, cu atribuire concurentă, cei 16 biți ai numărătorului pe LED-uri.

Dacă descrierea este corectă, se poate vizualiza schema circuitului rezultat.

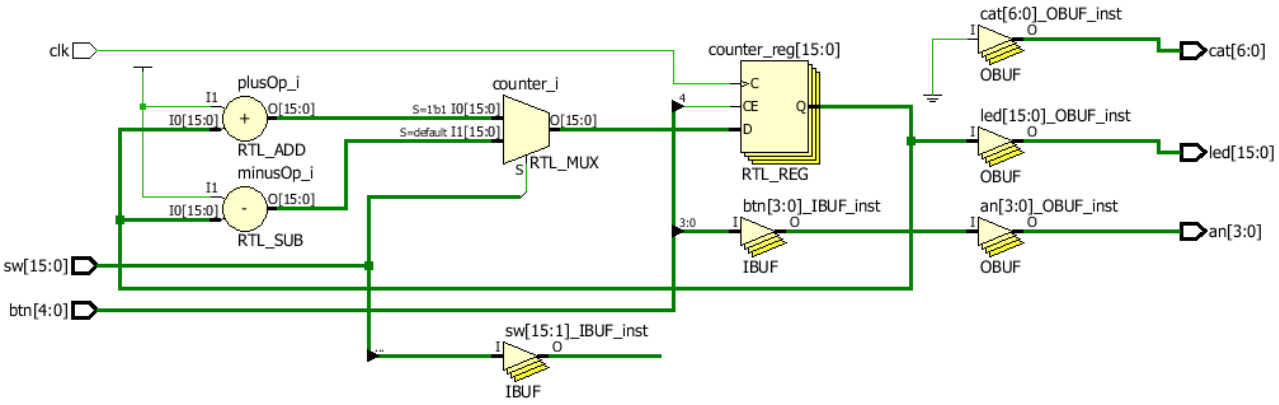


Figura 1-6: Schema circuitului realizat până acum, se poate vizualiza cu click pe **RTL Analysis – Elaborated Design → Schematic în panoul Flow Navigator**

Obțineți noul fișier *.bit (**Generate Bitstream**). Încărcați proiectul pe placa Basys. Controlați numărătorul de la buton și switch.

…Ce probleme apar?

1.5.3. Generator de Monoimpuls Sincron (durată o perioadă de ceas) – MonoPulse Generator - MPG

La acest punct lucrați în proiectul de la punctul 3.2.

În proiectele viitoare veți avea nevoie să controlați pas cu pas circuite secvențiale, cu scopul de a trasa și testa fluxul de date și de control din circuitele implementate. Va fi necesar un semnal, ENABLE, de activare/validare a frontului crescător de ceas.

Circuitul necesar, care activează un semnal de ENABLE o singură dată la o apăsare a butonului, este prezentat în figura următoare.

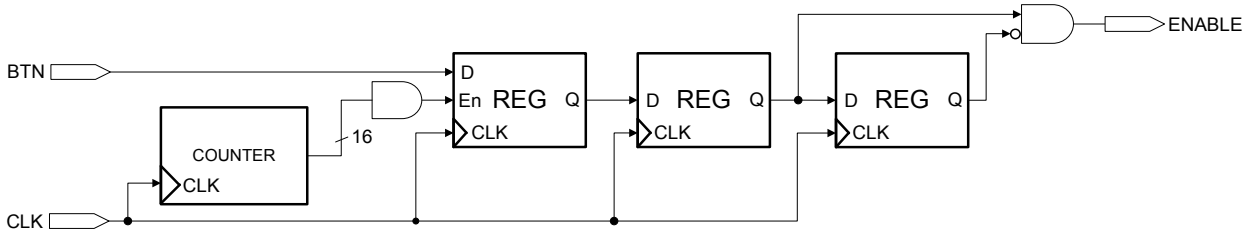


Figura 1-7: Generator de monoimpuls sincron

Rolul primului registru, împreună cu numărătorul, este de a asigura robustețe la utilizarea butoanelor uzate fizic, când pot să apară activări multiple ale semnalului ENABLE la o apăsare de buton. În funcție de uzură, este posibil să fie nevoie de mai mulți biți ai numărătorului (17-20+) pe care să se aplice un ŞI logic, astfel încât să se mărească intervalul de eșantionare al butonului.

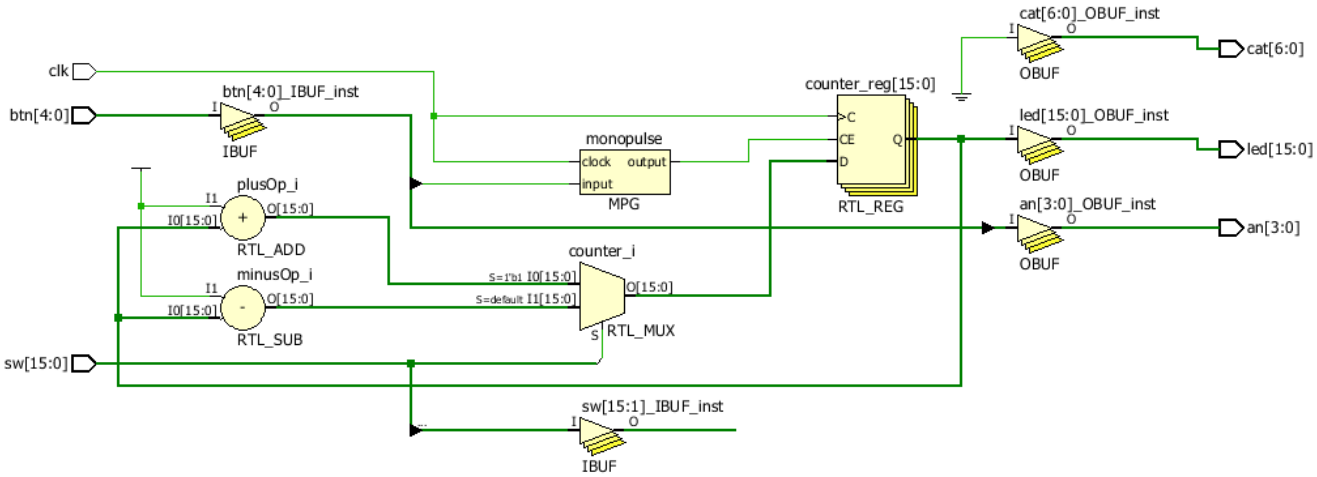
MPG-ul va fi implementat într-o entitate / fișier sursă nou (meniu **File\Add Source**) și va fi folosit în entitatea *test_env* prin declarerea cu *component* în secțiunea de declarare a semnalelor, respectiv instanțierea cu *port map* după begin în arhitectură. Nu uitați să adăugați bibliotecile necesare (vezi Anexa 1).

Intern, componentele din diagrama MPG, registre/bistabili, numărător, porțile AND, se vor descrie comportamental prin declararea semnalelor necesare, respectiv a proceselor și a atribuirilor concurente în arhitectura MPG. Vezi cursul 1 pentru indicii legate de descrierea VHDL.

Pași de urmat:

a. Desenați diagrama de timp a circuitului MPG (hârtie sau la tablă)
b. Scrieți și verificați codul VHDL pentru acest circuit
c. Includeți MPG în entitatea *test_env*. Vezi indicațiile de mai sus
d. Folosiți ieșirea ENABLE ca semnal de activare a ceasului numărătorului de 8 biți adăugat la pasul 3.2 în *test_env* (se adaugă condiția ca ENABLE să fie 1, în locul butonului, acolo unde se testează apariția frontului crescător de ceas în numărător).

Nu uitați de **RTL Analysis – Elaborated Design → Schematic …**



Încărcați pe placa Basys!

1.5.4. Creați un nou proiect, de exemplu test_new, folosind aceleași porturi ca pentru primul proiect. Practic, trebuie să reparcurgeți tutorialul din Anexa

1, fără a adăuga nimic în arhitectura test_new. Acum, implementați circuitul de mai jos în arhitectura test_new.

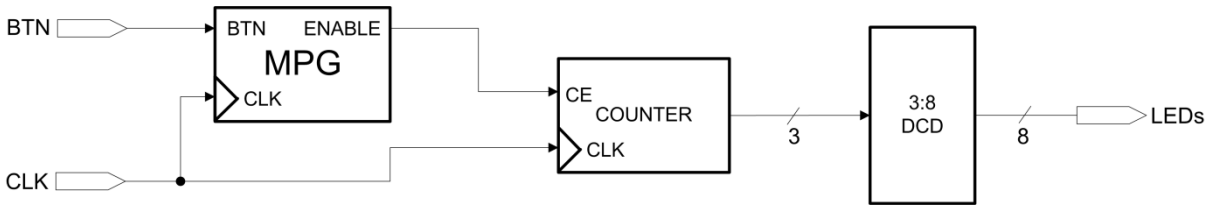


Figura 1-8: Schema circuitului pentru activitatea curentă

Trebuie să adăugați fișierul sursă MPG în noul proiect (**File\Add Sources** sau în panoul **Flow Navigator** la **Project Manager / Add Sources**). MPG se importă cu *component* / *port map* în entitatea *test_new*, iar restul componentelor se descriu în arhitectura *test_new* fără (!) entități adiționale. Adăugați un numărător pe 3 biți, și un decodificator 3:8 biți, folosind doar semnale declarate în arhitectura *test_new*, respectiv procese/atribuiri concurente.

Nu uitați de **RTL Analysis – Elaborated Design → Schematic** …

Încărcați pe placa Basys!

Temă

1. Finalizați activitățile neterminate la laborator.
2. (pentru studenți) Recitiți cu atenție **Regulamentul Laboratorului de AC** și reparcurgeți tutorialul din Anexa 1 de creare a unui proiect nou – din laboratorul 2 aceste noțiuni se consideră învățate! Atenție la aspectele din tutorial care nu au fost relevante pentru acest prim proiect: ele vor fi necesare în viitor.
3. (*pentru studenți, acest punct se consideră implicit pentru următoarele laboratoare)* Citiți materialul pentru laboratorul următor!

1.6. Referințe

[1] Manual de referinţă 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/.

Laboratorul 2

2. Extinderea proiectului curent: afișorul pe 7 segmente

2.1. Obiective

Descrierea, implementarea și testarea pentru:

- **Afișorul pe 7 segmente**
- **Unitatea Aritmetică-Logică simplă (Arithmetic Logic Unit - ALU).**

Aprofundarea cunoștințelor legate de:

- Vivado Webpack
- Xilinx Vivado Design Suite User Guide
- Digilent Development Boards **(DDB)**
  - Digilent Basys 3 Board – Reference Manual
- Artix 7 FPGA.

2.2. Afișorul pe 7 segmente cu 4 cifre

Placa Basys 3 vine echipată cu un afișor pe 7 segmente cu 4 cifre (Seven Segment Display - SSD). Pe scurt, această interfață folosește șapte leduri pentru fiecare cifră; fiecare cifră este activată de un semnal de anod. Toate semnalele interfeței SSD (7 semnale comune de catod și 4 semnale distincte de anod) sunt active pe 0. Semnalele de catod controlează ledurile care se aprind de pe acele cifre care au semnalul de anod activ (de exemplu dacă se activează toate 4 anodurile, atunci se va afișa aceeași cifră pe cele 4 poziții).

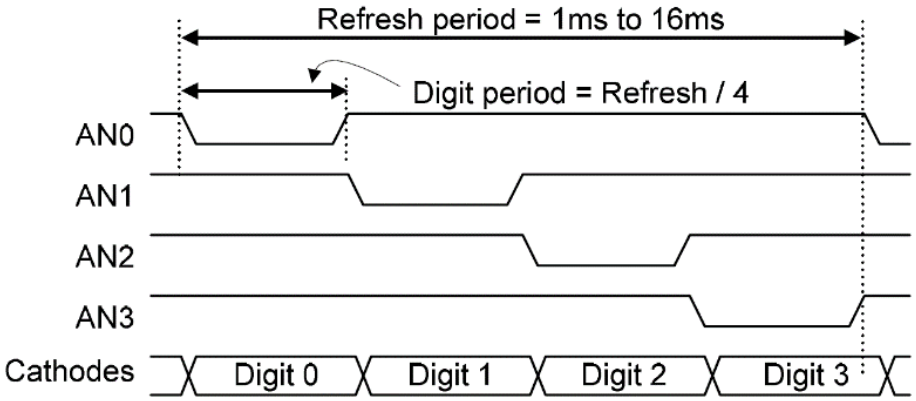


Figura 2-1: Diagrama de timp pentru SSD [1]

Pentru a afișa 4 cifre diferite pe SSD, este necesară implementarea unui circuit care trimite cifrele pe semnalele de catod ale SSD în concordanță cu diagrama de timp a SSD. Perioada maximă de reîmprospătare (refresh) este astfel calculată încât ochiul uman să nu perceapă aprinderea și stingerea succesivă a

fiecărei cifre de pe SSD (16 ms <=> 60 Hz). Se realizează astfel o afișare ciclică a cifrelor (la un moment dat doar o cifră este afișată, dar ochiul nu percepe acest aspect). Deschideți manualul de referință Basys și citiți secțiunea legată de afișorul pe 7 segmente.

În figura de mai jos se prezintă o posibilă implementare a circuitului de afișare pe SSD. Intrările sunt 4 semnale de câte 4 biți (cifrele de afișat) și semnalul de ceas al plăcii; ieșirile sunt reprezentate de semnalele de anod (an) și semnalele de catod (cat), toate active pe zero.

În acest circuit există un automat cu stări finite, cu o implementare particulară. Care componentă îl reprezintă, și care sunt semnalele lui de control (ieșirile) ?

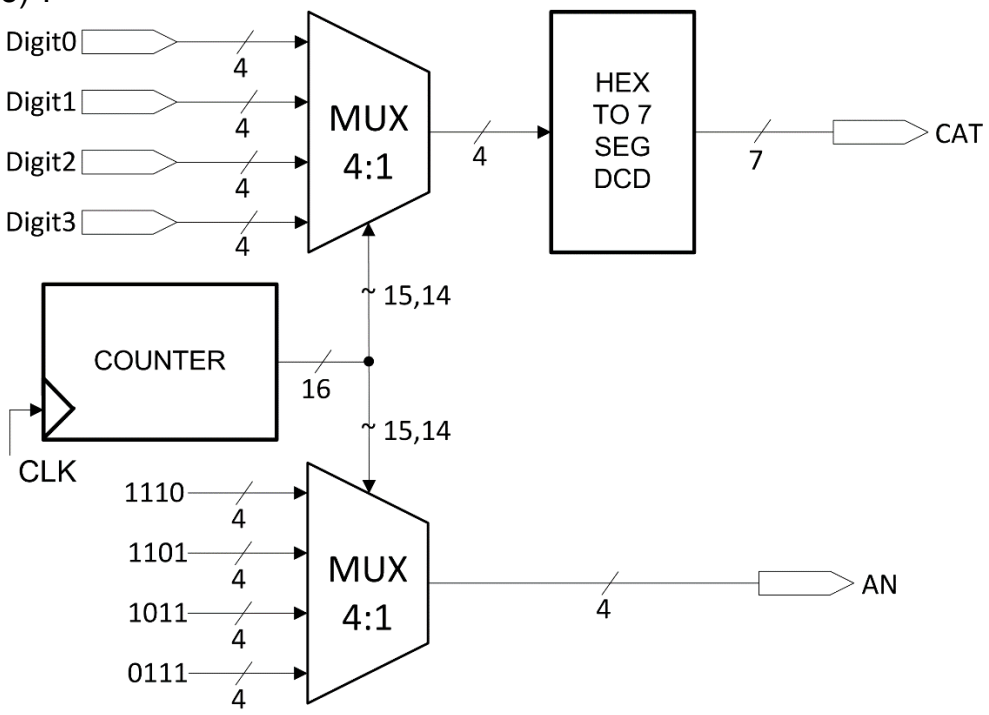


Figura 2-2: Schema circuitului de afișare SSD

2.3. Structura generală a proiectelor pentru placa Basys în acest îndrumător de laborator

După ce veți termina activitatea 2.5.1 din acest laborator, toate proiectele viitoare vor avea o structură generală ca în figura următoare. Această structură oferă interfața necesară pentru lucrul cu placa Basys. Descrierea circuitelor specifice fiecărui laborator (comportamental și / sau prin instanțierea unor componente) se va face în „nor”.

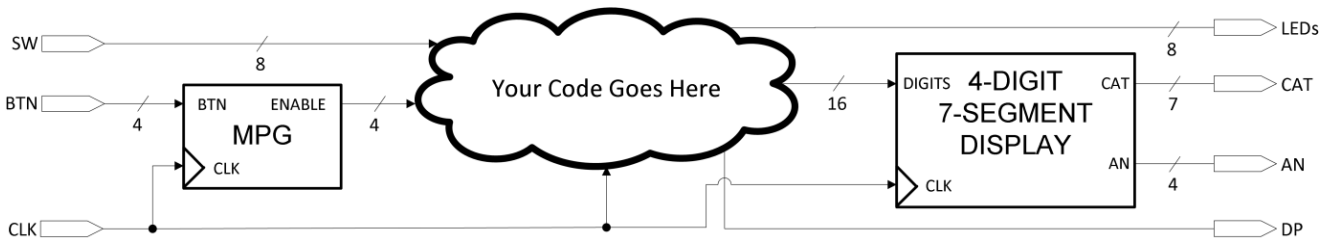


Figura 2-3: Schema generală a viitoarelor proiecte

2.4. Operații tipice pentru ALU

Fiecare operație (cu circuitul asociat) care urmează este descrisă pe scurt din punct de vedere teoretic, după care urmează indicii privind implementarea în VHDL (mult simplificată față de descrierea teoretică, ajungând o linie de cod VHDL pentru majoritatea operațiilor).

2.4.1. Sumatoare (Adders)

Un sumator este un circuit digital care efectuează adunarea între numere. În calculatoarele moderne, sumatoarele sunt plasate în unitatea aritmetică-logică (ALU), împreună cu alte operații. Sumatoarele pe mai mulți biți (un exemplu pe 8 biți este prezentat în Figura 2-4) sunt realizate prin înlănțuirea mai multor sumatoare simple, pe 1 bit. Ecuațiile booleene pentru un sumator complet pe 1 bit sunt:

$$Sum = A\ xor\ B\ xor\ Cin$$
$$Cout = (A\ and\ B)\ or\ (A\ and\ Cin)\ or\ (B\ and\ Cin)$$

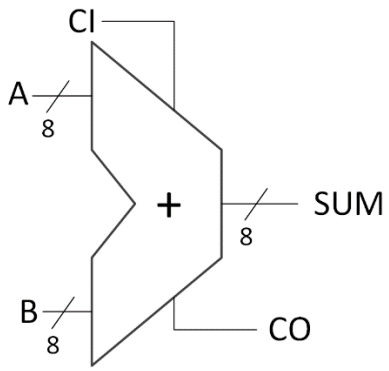


Figura 2-4: Sumator complet pe 8 biți (cu Carry in – transport de intrare și Carry out – transport de ieșire)

VHDL: …suma <= termen1 + termen2

- în acest laborator ignorăm problema transportului!

2.4.2. Scăzătoare (Subtractors)

Un scăzător este un circuit digital care efectuează operația de scădere. În hardware, pentru numere reprezentate în complement față de 2, scăderea se realizează prin adunarea cu negativul scăzătorului.

$$A - B = A + \overline{B} + 1$$

VHDL: …diferența <= termen1 - termen2

2.4.3. Circuite de deplasare (Shifters)

Un circuit de deplasare este un circuit digital care translatează un cuvânt, conținând un sir de biți, cu un număr specificat de poziții. Există două tipuri de circuite de deplasare:

- Deplasare logică – pe pozițiile rămase goale se pune 0
- Deplasare aritmetică – la deplasare spre dreapta se extinde bitul de semn.

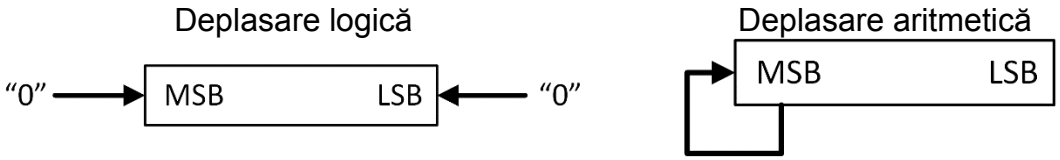


Figura 2-5: Tipuri de deplasare, pozițiile extreme de biți sunt MSB (Most Significant Bit) și LSB (Least Significant Bit)

În general, pentru ca Xilinx să recunoască explicit un circuit ca fiind de deplasare, circuitul trebuie să fie combinaţional, cu două intrări și o ieșire, și să se folosească doar operațiile predefinite de deplasare (sll, srl etc.) sau operatorul de concatenare &.

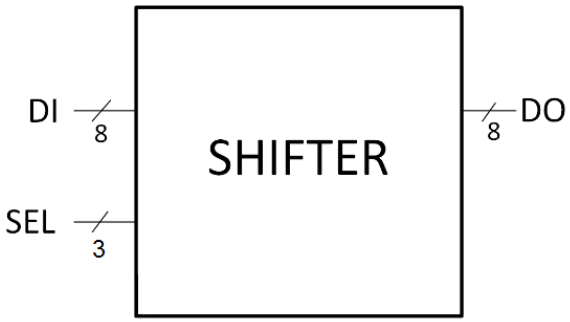


Figura 2-6: Circuit de deplasare pe 8 biți, DI intrare, DO ieșire, SEL cantitatea de deplasare

VHDL: …exemplu pe 32 de biți, deplasare stânga cu 5 poziții

```
iesire(31 downto 0) <= intrare(26 downto 0) & ”00000”
```

Ca lectură opțională, în anexa 2 din acest document se prezintă un circuit de deplasare combinaţional realizat prin cascadarea mai multor niveluri de multiplexoare, care poate efectua deplasare variabilă (selecția S, de la 0 până la maxim 7 poziții).

2.4.4. Detector de zero (Zero Detector)

Detectorul de zero pe n biți este implementat de obicei printr-o poartă NOR, cu intrarea pe n biți, ieșirea de 1 bit. Acest circuit este folosit în special în unitățile ALU pentru a evalua condiția de salt (de exemplu pentru instrucțiunea *Branch on Equal* la procesorul MIPS).

VHDL: …atribuire concurentă cu *when/else*

```
Zero <= '1' when DI=0 else '0';
```

2.4.5. Extindere cu semn/zero (Sign/Zero Extender)

Circuitul de extindere cu semn / zero din procesorul MIPS este folosit în operații aritmetice-logice pentru care unul dintre operanzi este un imediat pe 16 biți. Extensia este necesară deoarece ALU lucrează cu operanzi pe 32 de biți.

VHDL: …în funcție de valoarea unui bit de selecție (cu semn / zero) folosiți operatorul de concatenare **&** pentru a construi semnalul extins de ieșire (pentru extensie cu semn, se testează bitul de semn pentru a determina dacă se concatenează cu biți de 0 sau de 1).

2.4.6. Comparatoare

Un comparator este un circuit digital care compară două numere în formă binară și generează un semnal de 1 bit, cu valoarea 1 sau 0, care arată dacă numerele comparate sunt egale sau nu. În ALU, comparația a două numere se face efectuând operația de scădere (fără scrierea rezultatului într-o destinație) după care se verifică dacă rezultatul este zero (vezi detectorul de zero).

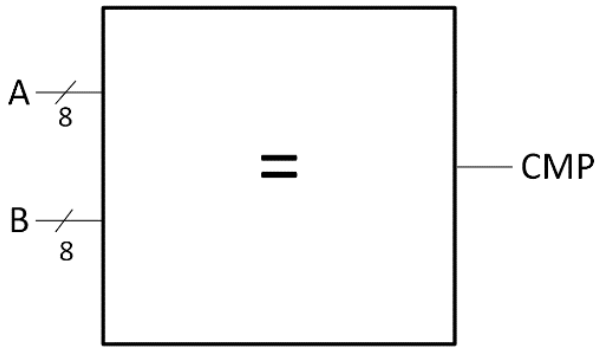


Figura 2-7: Comparator de numere pe 8-biți, A și B numerele de comparat, CMP semnalul de stare care indică dacă sunt sau nu egale

VHDL: …scădere în ALU cu testarea rezultatului, sau atribuire concurentă cu *when/else*

```
CMP <= '1' when A = B else '0';
```

2.5. Activități practice

2.5.1. Implementarea circuitului de afișare pentru SSD, 4 cifre

Se continuă lucrul în proiectul din laboratorul anterior (*test_env*). Descrieți în VHDL o componentă (entitate separată, SSD, adăugați o sursă nouă în proiect) care să implementeze circuitul de afișare pe 4 cifre, conform schemei din Figura 2-2. **În noua entitate creată, folosiți doar semnale declarate în arhitectură și procese pentru a descrie numărătorul și cele două multiplexoare (cu case),**

și folosiți Language Templates pentru a implementa "HEX TO 7 SEG DCD" (componenta din afișor care transformă cifra pe 4 biți în combinația de 7 leduri aprinse/stinse).

Declarați noua componentă în arhitectura entității *test_env* și instanțiați-o. Conectați un numărător pe 16 biți la cele 4 intrări ale afișorului SSD. Numărătorul se va incrementa doar la apăsarea unui buton (folosiți componenta MPG pentru a valida incrementarea numărătorului). Acum, structura proiectului trebuie să semene cu structura descrisă în Figura 2-3.

2.5.2. O unitate aritmetică-logică simplă - ALU

Folosind proiectul anterior (care conține MPG și SSD), implementați o ALU simplă cu următoarele funcții: ADD, SUB, SHIFT LEFT 2, SHIFT RIGHT 2, conform diagramei din figura următoare.

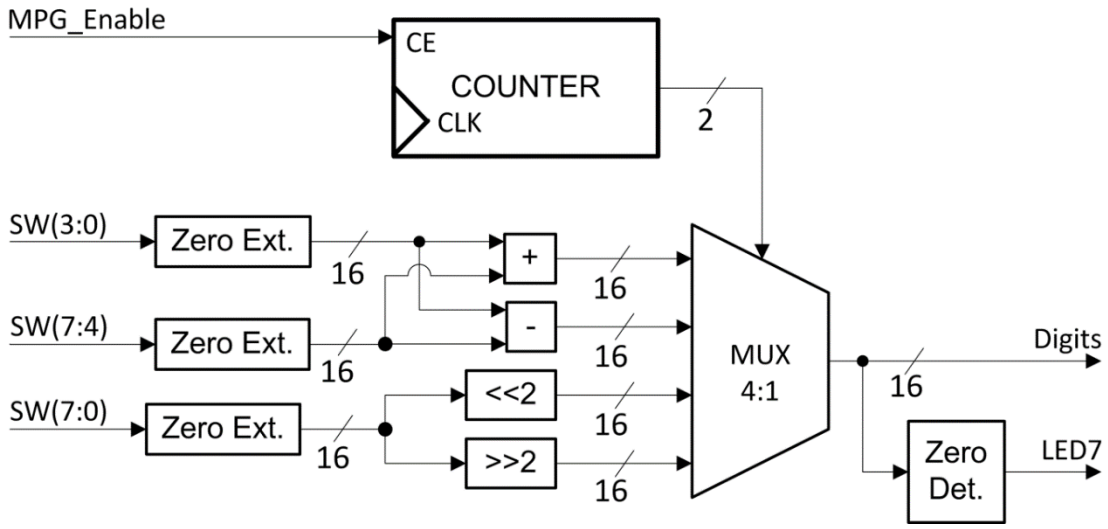


Figura 2-8: Schema unei ALU simple

Rezultatul pe 16 biți al operațiilor din ALU trebuie trimis (afișat) pe interfața SSD (Digit3...Digit0). Folosiți un numărător pe 2 biți (controlat prin buton, via MPG) pentru a selecta operația ALU dorită.

Operanzi de intrare ai ALU sunt comutatoarele (switches) plăcii Basys. Descrierea VHDL a acestei ALU se va face strict (fără entitate nouă) în arhitectura *test_env*, folosind doar semnale interne, procese și atribuiri concurente.

2.6. Referințe

[1] Manual de referinţă 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.

Laboratorul 3

3. Memorii

3.1. Obiective

Descrierea, implementarea și testarea pentru:

- **Bloc de registre (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 3 Board – Reference Manual
- Artix 7 FPGA.

3.2. Fundamente teoretice

3.2.1. Blocul de registre / Register File

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

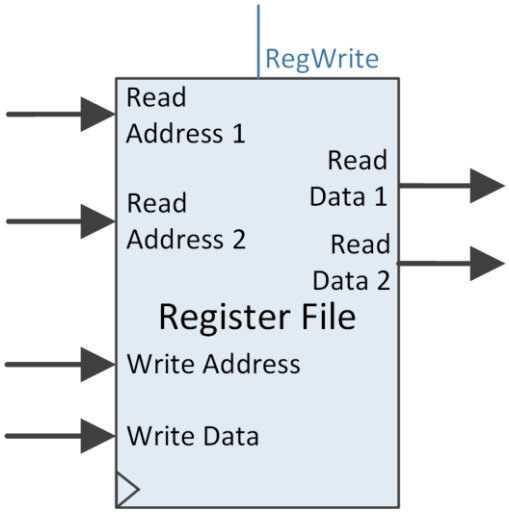


Figura 3-1: Un bloc de registre cu două porturi de citire și un port de scriere, specific procesorului MIPS

Majoritatea operațiilor dintr-un procesor implică folosirea sau modificarea datelor stocate în blocul de registre. Deoarece blocul de registre funcționează la frecvența de ceas a procesorului, el este limitat ca dimensiune și trebuie să fie foarte rapid. În aplicațiile reale pe plăci FPGA, un bloc de registre este implementat

folosind bistabili și tabelele de asociere (LUT) din circuitul FPGA, generând astfel o memorie rapidă, care permite acces multiplu simultan. Pe procesoare se folosește de obicei tehnologie SRAM (sau similară) pentru implementarea blocului de registre.

Blocul de registre specific procesorului MIPS (vezi figura anterioară) are două adrese de citire (Read Address 1 și 2) și o adresă de scriere (Write Address). Conținutul registrelor care corespund locațiilor indicate de cele două adrese de citire sunt livrate pe cele două porturi de ieșire Read Data 1 și 2. Datele furnizate pe portul de scriere Write Data sunt scrise în registrul indicat de adresa de scriere, dacă semnalul de control RegWrite este activat. Operațiile de citire sunt asincrone (combinaţionale), iar operația de scriere este sincronă (pe front crescător). În contextul procesorului MIPS, blocul de registre suportă două citiri și o scriere în fiecare ciclu de ceas.

În anexa 3 este prezentată o posibilă descriere în VHDL pentru un bloc de registre cu 8 registre de câte 8 biți fiecare. Se va evita CTRL+C, CTRL+V…

3.2.2. 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 obișnuit 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 același timp de acces 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, [1]). Un BRAM poate fi configurat fie ca un ROM, fie ca un RAM. În funcție de cum se descrie în 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 registre, iar memoriile BRAM sunt mapate direct pe blocurile BRAM disponibile. Memoriile RAM distribuite consumă direct din porțile FPGA și pot să afecteze 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 [4]: *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 (256 de locații de câte 16 biți fiecare) cu modul "*no change*" este prezentată în Anexa 4.

De analizat:

- Accesați Language templates - VHDL → Synthesis Constructs → Coding Examples → RAM și vedeți diferențele de descriere pentru RAM distribuit și Bloc RAM
- 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 *Language templates*.

3.2.3. Declararea unui șir în VHDL

În continuare, se prezintă un exemplu de declarare și inițializare pentru un șir, care se poate folosi pentru memorii ROM, RAM, respectiv pentru un bloc de registre. 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);
```

Se declară un semnal de tipul declarat anterior:

```
signal r_name: <arr_type>;
```

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

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

3.3. Activități practice

Este obligatoriu ca la începutul acestui laborator să aveți un proiect funcțional care să coincidă cu descrierea din laboratorul 2, secțiunea 2.3 (să conțină MPG și SSD, instanțiate în entitate top level, denumită 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.

3.3.1. Implementarea memoriei ROM

Includeți o memorie ROM de 256x16 biți în proiectul *test_env*, în entitatea top level (nu declarați o nouă entitate!). Inițializați ROM-ul cu câteva valori arbitrare (vezi). 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 (via Digits). Comportamentul ROM-ului este asincron (se descrie într-o linie de cod). Vedeți figura următoare pentru schema circuitului.

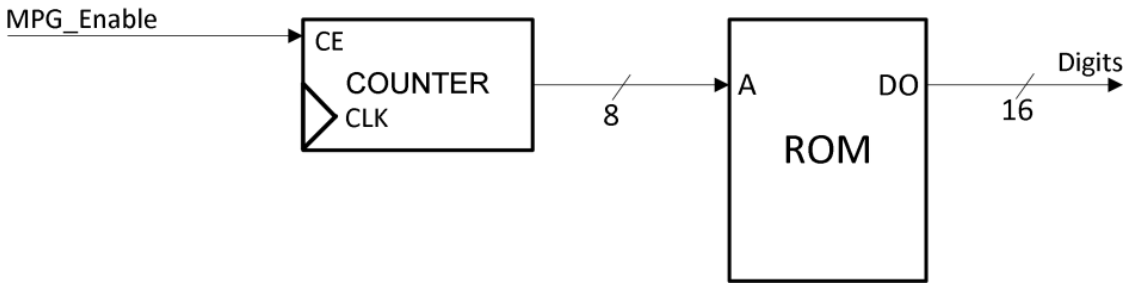


Figura 3-2: Schemă simplă cu o memorie ROM

Testați pe placa de dezvoltare!

3.3.2. Implementarea Blocului de Registre

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

Proiectați și implementați un bloc de registre RF pentru placa Basys. Descrieți o nouă componentă (entitate) pentru blocul de registre, în proiectul *test_env*. Schema în care să fie inclus blocul de registre RF este prezentată în Figura 3-3.

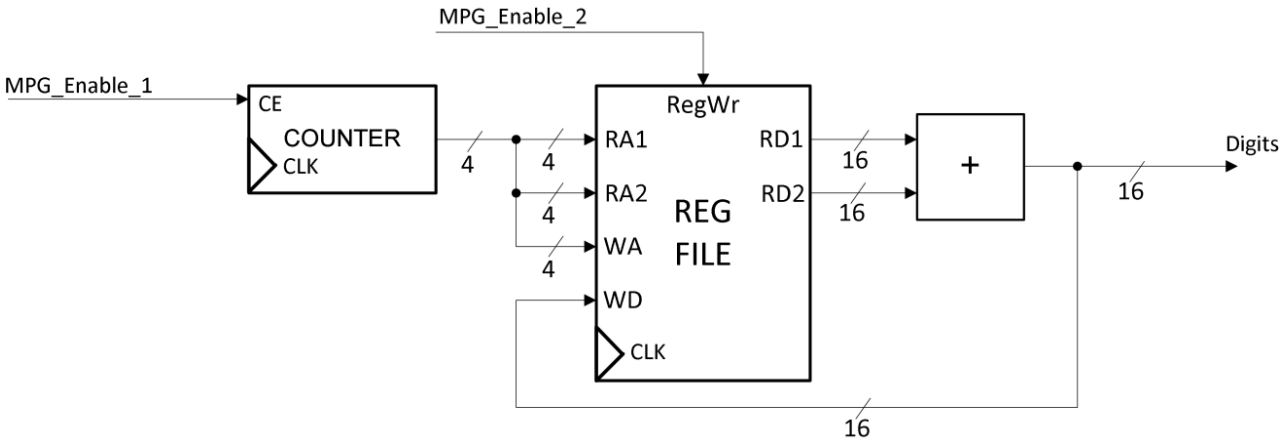


Figura 3-3: O schemă simplă pentru utilizarea blocului de registre

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 pregătită pentru scriere 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 dublată în RF.

Testați pe placă!

3.3.3. Memoria RAM

Înlocuiți (comentați) blocul de registre de la punctul anterior cu o memorie RAM. Folosiți un circuit de deplasare la stânga cu 2 poziții în loc de adunare (diferența față de Figura 3-3 este că sumatorul este înlocuit cu un circuit de deplasare, care se descrie cu concatenare pe biți). Folosiți un singur port de adresă pentru memoria RAM, implementarea fiind cu modul *write-first*.

3.4. 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.

Laboratorul 4

4. Procesorul MIPS, ciclu unic – versiune pe 16 biți (1)

Definirea instrucțiunilor / scrierea programului de test (asamblare / cod mașină)

4.1. Obiective

Studiul, proiectarea, implementarea și testarea:

- **Procesorului MIPS, pe 16 biți, un ciclu de ceas / instrucțiune (ciclu unic / single-cycle).**

Familiarizarea studenților cu:

- Proiectarea procesorului: Definirea instrucțiunilor / scrierea programului de test (asamblare / cod mașină)
- Vivado Webpack
- Xilinx Vivado Design Suite User Guide.

4.2. Descrierea procesorului MIPS, simplificat pe 16 biți

(!) Pentru studenți: citiți cursurile 3 (obligatoriu) și 4 (după predare) pentru a înțelege conținutul acestui laborator. Este necesară familiarizarea în prealabil cu procesorul MIPS pe 32 de biți, ciclu unic, descris în [2] de către Patterson și Hennessy.

În acest laborator veți face proiectarea (în mod mult simplificat, prin desenarea căii de date – proiectarea completă se face în cursul 4) și veți începe implementarea propriei versiuni a procesorului MIPS pe 16 biți, referit în continuare ca MIPS 16.

Acest microprocesor va fi o versiune simplificată a procesorului MIPS 32 descris la curs. Ce înseamnă simplificat? Setul de instrucțiuni va fi mai mic, dimensiunea instrucțiunilor/a cuvântului va fi pe 16 biți, și, implicit, vom avea un număr redus de registre de uz general, respectiv o dimensiune mai mică pentru memorie. În rest, principiile de proiectare din curs rămân valabile (calea de date, control).

Principalul motiv pentru simplificarea pe 16 biți este dat de modalitățile restrânse de afișare de pe placa de dezvoltare (leduri, afișorul SSD). Astfel, se evită mecanisme suplimentare de multiplexare la afișare (pentru numere de 32 biți), și se ușurează procesul de trasare / testare a programului exemplu pe procesorul implementat.

Dimensiunea/lățimea instrucțiunilor și a datelor va fi pe 16 biți. Formatul celor 3 tipuri de instrucțiuni este prezentat în figurile următoare. Comparați acest format cu formatul din curs pe 32 de biți. Observați modificările / limitările.

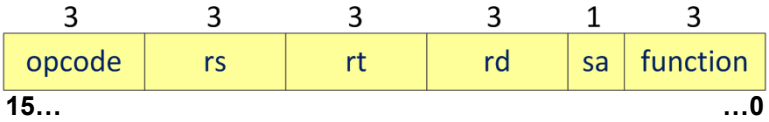


Figura 4-1: Formatul pentru instrucțiune de tip R

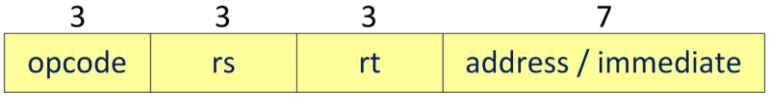


Figura 4-2: Formatul pentru instrucțiune de tip I

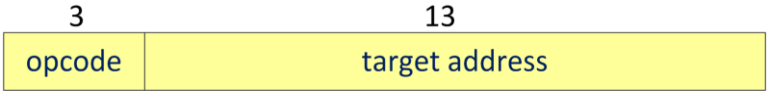


Figura 4-3: Formatul pentru instrucțiune de tip I

Aceste formate de instrucțiuni respectă formatele extinse din MIPS 32 ISA, exceptând numărul de biți alocați pentru fiecare câmp.

Câmpul **opcode** este pe 3 biți. Pentru instrucțiunile de tip I și J, **opcode** codifică într-un mod unic instrucțiunea care se va executa. În cazul instrucțiunilor de tip R, în conformitate cu standardul MIPS, câmpul **opcode** este 0 iar funcția / operația pentru ALU este codificată în câmpul **function**, pe 3 biți. Rezultă implicit că procesorul MIPS 16 va putea implementa maxim 15 instrucțiuni:

- 7 instrucțiuni de tip I, respectiv J
- 8 instrucțiuni de tip R.

Mai jos se prezintă setul minimal de instrucțiuni, de fiecare tip, care se vor implementa pe procesorul MIPS 16. Pe pozițiile rămase libere respectați indicațiile care vor urma sau, opțional, puteți defini alte instrucțiuni, dacă aveți nevoie de ele pentru programul în asamblare pe care îl veți scrie (cu justificare).

Instrucțiuni de tip R	Addition	add
	Subtraction	sub
	Shift Left Logical (with shift amount – sa)	sll
	Shift Right Logical (with shift amount – sa)	srl
	Logical AND	and
	Logical OR	or
	…. de definit!	…
	…. de definit!	…
Instrucțiuni de tip I	Add Immediate	addi
	Load Word	lw
	Store Word	sw
	Branch on Equal	beq
	…. de definit!	…
	…. de definit!	…
Instrucțiuni de tip J	Jump	j

Tabel 4.1: Instrucțiuni pentru MIPS 16

Urmează descrierea caracteristicilor pentru elementele principale ale procesorului MIPS 16, (!) valabile atât pentru laboratorul curent, cât și (=mai ales)

pentru laboratoarele viitoare. Aceste elemente sunt similare cu cele de la MIPS 32, dar reduse la 16 biți.

Registrului PC, contorul de program:

- Registru pe 16 biți, pe front crescător.

Memoria de instrucțiuni ROM:

- Un port de intrare: adresa instrucțiunii
- Un port de ieșire: conținutul instrucțiunii (16 biți)
- Cuvântul de memorie este de 16 biți, selectat de adresa instrucțiunii
- Citire combinațională, fără semnale de control.

Blocul de registre RF:

- 2 adrese de citire (Read Address 1, Read Address 2) și o adresă de scriere (Write Address)
- 8 registre de câte 16 biți (rs, rt și rd codificați pe 3 biți!)
- 2 ieșiri de 16 biți: Read Data 1 și Read Data 2
- O intrare pe 16 biți: Write Data
- Permite acces multiplu: 2 citiri asincrone și o scriere sincronă (front crescător de ceas). Pe parcursul operației de citire, RF se comportă ca un circuit combinaţional
- Un semnal de control RegWrite. Când acesta este activat, datele prezente pe Write Data sunt scrise sincron în registrul indicat de adresa de scriere.

Memoria de date RAM:

- O intrare de adresă pe 16 biți: Address
- O intrare de date pe 16 biți: Write Data
- O ieșire de date pe 16 biți: Read Data
- Citire combinațională
- Un semnal de control: MemWrite, pentru validarea scrierii sincrone.

Unitatea de extindere:

- Un semnal de control ExtOp
- ExtOp = 1 → Extindere cu semn
- ExtOp = 0 → Extindere cu zero.

Unitatea aritmetică-logică ALU:

- ALU efectuează operații aritmetice-logice
- (!) Identificați toate operațiile pe care ALU trebuie să le efectueze, după definirea instrucțiunilor din Tabel 4.1. Este recomandat să alegeți încă 2 instrucțiuni de tip R și 2 de tip I pe care să le definiți
- Identificați câți biți de control sunt necesari pentru a codifica operațiile ALU (semnalul ALUOp).

4.3. Activități practice

Citiți fiecare activitate în întregime, înainte să o începeți!

4.3.1. Definirea instrucțiunilor pentru MIPS 16 – activitate hârtie / instrument de scris

Adăugați la alegere încă 2 instrucțiuni de tip R și 2 de tip I, pentru a avea complet setul de instrucțiuni suportate de procesorul MIPS 16.

Pentru cele 15 instrucțiuni (Tabel 4.1 plus cele 4 alese), urmăriți pașii din cursul 3 de definire a instrucțiunilor (format pe biți, stabiliți fiecare individual codificarea **opcode / function**, descriere, RTL abstract, diagrama de procesare). Pe durata laboratorului, definiți toate instrucțiunile, dar faceți diagrama de procesare doar pentru add, lw, beq, j. Pentru restul instrucțiunilor, faceți diagrama de procesare acasă ca temă.

Pe lângă materialul de curs, folosiți Anexa 5 ca referință pentru instrucțiunile MIPS 32. Pentru implementarea de la laborator a procesorului MIPS 16, se va ignora partea de excepții în caz de depășire (ex. pentru add).

Dați un exemplu de codificare pe biți (codul mașină) pentru fiecare instrucțiune (inclusiv pentru operanzii instrucțiunii). Ex. *add $2, $4, $3* => ”…*cei 16 biți*…”.

Atenție: pentru a crește lizibilitatea codificării pe biți a instrucțiunii folosiți simbolul ”_” între câmpurile instrucțiunii (opcode, rs etc.), atât pe hârtie, cât și în VHDL (este suportat de limbaj, nu are nici un efect în șirul de biți). Pentru VHDL este obligatorie specificarea de binar în fața ”B” șirului de biți (sau X pentru hexa):

```
B"001_010_011_100_1_111" este echivalent cu "0010100111001111"
```

4.3.2. Programul de testare pentru MIPS 16

Scrieți un program cu instrucțiunile implementate (hârtie / pix). Descrieți programul în asamblare, apoi fiecare instrucțiune în cod mașină (codificarea pe 16 biți, binar, cu separatorul “_” între câmpuri).

Din motive pe care le veți înțelege doar când veți face testarea programului pe procesorul implementat pe placă (peste câteva laboratoare), scrieți programul în așa fel încât să existe cel puțin:

1. O instrucțiune de scriere într-un registru, urmată de instrucțiuni care folosesc registrul respectiv ca registru sursă
2. O instrucțiune de scriere într-o locație de memorie, urmată de instrucțiuni care vor citi acea locație de memorie și vor folosi valoarea în calcule.

Folosiți rezultatul activității 3.3.1 din laboratorul 3 (memoria ROM legată la un numărător care generează adresele). Introduceți programul scris în cod mașină în memoria ROM și verificați pe placa de dezvoltare. **Atenție**: la inițializarea memoriei scrieți cu comentariu în dreptul fiecărei intrări descrierea în asamblare a instrucțiunii respective / respectiv codul mașină în baza 16. Practic, programul vostru trebuie să fie vizibil în paralel cu codul mașină pentru a ușura procesul de testare / depanare pe placă.

Temă pentru acasă: extinderea programului spre ceva mai complex, până în laboratorul 5!

4.3.3. Calea de date pentru MIPS 16

Această activitate se va face acasă, imediat după participarea la cursul în care se proiectează procesorul.

Desenați calea de date pentru procesorul MIPS 16 pe care îl implementați. Asigurați-vă că includeți componentele necesare astfel încât cele 15 instrucțiuni să se execute corect.

Pornind de la descrierea RTL abstract, identificați / stabiliți valorile pentru semnalele de control necesare fiecărei instrucțiuni. Completați un tabel cu semnalele de control și valorile lor (consultați cursul pentru exemple).

4.4. Referințe

[1] Arhitectura Calculatoarelor, note de curs (online) 3 & 4.
[2] D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
[3] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012.
[4] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02.
[5] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
- Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.

Laboratorul 5

5. Procesorul MIPS, ciclu unic – versiune pe 16 biți (2)

Unitatea pentru extragerea instrucțiunilor / Instruction Fetch - IF

5.1. Obiective

Studiul, proiectarea, implementarea și testarea:

- **Unității de IF pentru procesorul MIPS, pe 16 biți, un ciclu de ceas / instrucțiune (ciclu unic).**

5.2. Descrierea procesorului MIPS, simplificat pe 16 biți - continuare

(!) Este obligatorie studierea cursurilor 3 și 4 (și participarea la aceste cursuri) pentru a înțelege activitățile din acest laborator.

Ciclul de execuție a unei instrucțiuni MIPS are următoarele etape / faze (curs 4):

- IF – Extragerea Instrucțiunii / Instruction Fetch
- ID/OF – Decodificarea Instrucțiunii / Extragerea Operanzilor
  Instruction Decode / Operand Fetch
- EX – Execuție / Execute
- MEM – Memorie / Memory
- WB – Scriere Rezultat / Write Back.

Implementarea proprie a procesorului MIPS-16, pe care o începeți în acest laborator (și o veți termina în următoarele) va fi partiționată în 5 componente (entități noi). Aceste componente se vor declara și instanția în proiectul *test_env*, în entitatea principală (test_env, probabil…).

Partiționarea procesorului în entitățile asociate cu etapele de execuție nu prezintă beneficii explicite pentru procesorul MIPS 16 cu ciclu unic. Utilitatea acestei partiționări o veți înțelege în laboratoarele viitoare, când se va implementa versiunea pipeline a procesorului MIPS 16!

În acest laborator veți proiecta, descrie în VHDL și implementa / testa unitatea de extragere a instrucțiunii Instruction Fetch – IF, pentru versiunea proprie a procesorului MIPS 16 cu ciclu unic de ceas pe instrucțiune.

Calea de date a procesorului MIPS (versiunea 32 de biți [2]) este prezentată în figura următoare, împreună cu unitatea de control (și semnalele aferente). Pentru a evita aglomerarea schemei, semnalele de control nu au mai fost legate explicit la destinații, dar se pot identifica ușor după nume.

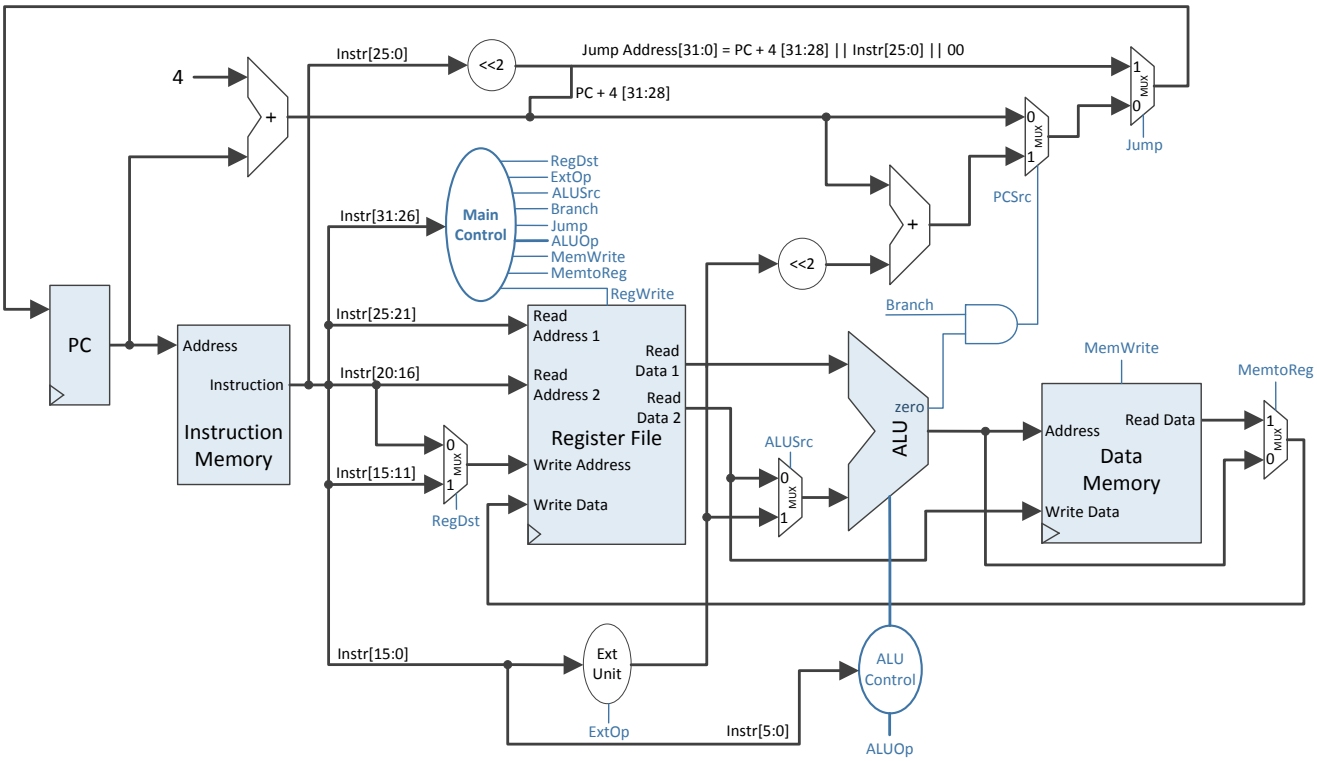


Figura 5-1: MIPS 32 cu ciclu unic de ceas, calea de date + control

Mai jos sunt revizitate cele 3 formate de instrucțiuni pentru MIPS 16, descrise în laboratorul anterior:

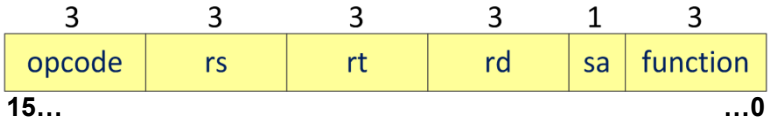


Figura 5-2: Instrucțiune de tip R

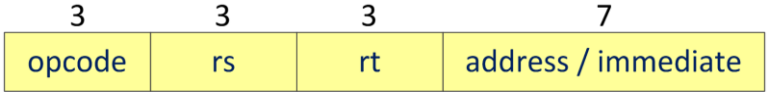


Figura 5-3: Instrucțiune de tip I

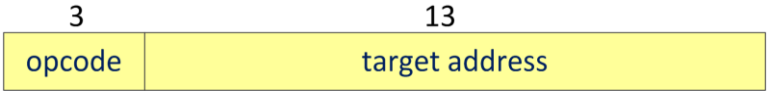


Figura 5-4: Instrucțiune de tip J

Unitatea de extragere a instrucțiunilor IF conține următoarele elemente principale (nu se vor descrie suplimentar entități!):

- Program Counter
- Instruction Memory (ROM)
- Adder.

În plus, mai există două MUX-uri pentru stabilirea adresei următoare. Consultați laboratorul anterior pentru caracteristicile acestor componente pentru MIPS 16. Calea de date pentru unitatea IF, MIPS 32, este prezentată în Figura 5-5.

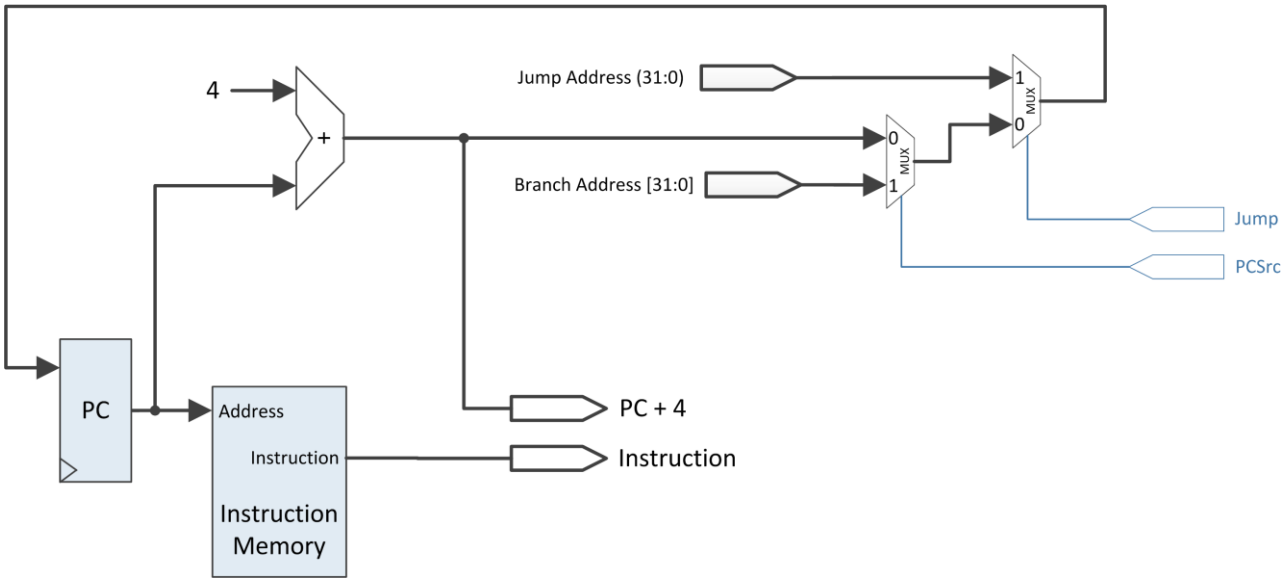


Figura 5-5: Calea de date pentru IF din procesorul MIPS 32

Ca ieșiri, unitatea IF furnizează instrucțiunea curentă și adresa următoarei instrucțiuni, pentru execuție secvențială. În cazul instrucțiunilor jump (j) și branch (beq), unitatea IF primește ca intrări adresa de salt pentru jump, respectiv adresa țintă pentru branch, împreună cu semnalele de control care vor selecta adresa următoarei instrucțiuni care se va executa (noua valoare a PC).

Intrările unității IF sunt:

- Semnalul de ceas (pentru …. PC)
- Adresa de branch
- Adresa de jump
- Semnalul de control Jump
- Semnalul de control PCSrc (pentru branch).

Ieșirile unității IF sunt:

- Instrucțiunea de executat în ciclul de ceas curent, în procesorul MIPS
- Adresa următoarei instrucțiuni de executat, mod secvențial (PC + 4).

Semnificația semnalelor de control:

- Jump = 1 → PC ← Jump Address
- Jump = 0 → PC ← (PC + 4 dacă PCSrc = 0 sau adresa de branch dacă PCSrc = 1).

5.3. Activități practice

Citiți fiecare activitate în întregime, înainte să o începeți practic!

Resurse necesare (de avut la începerea laboratorului!):

- Rezultatele obținute pentru toate activitățile din laboratoarele anterioare
- 15 instrucțiuni definite pentru propria voastră implementare a procesorului MIPS 16 (Laboratorul 4, activitatea 4.3.1)
- RTL abstract / formatul celor 15 instrucțiuni, scrise pe hârtie
- Calea de date pentru MIPS 16 hârtie / instrument de scris (Laboratorul 4, activitatea 4.3.3, folosiți Figura 5-1 din acest laborator ca referință, plus cursul 4 pentru detalii)
- Proiect Xilinx cu test_env, care să includă cel puțin memoria ROM (va juca rolul memoriei de instrucțiuni), inițializată la declararea semnalului cu programul vostru personalizat, scris în cod mașină (Laboratorul 4, activitatea 4.3.2).

5.3.1. Proiectarea / implementarea unității IF

Ținând cont de descrierea unității IF din Figura 5-5, descrieți o nouă componentă (entitate) pentru IF în proiectul *test_env*, ca primă unitate implementată a procesorului MIPS 16. Toate câmpurile de date sunt pe 16 biți!

Entitatea IF va conține elementele descrise în Figura 5-5, care nu se vor descrie cu entități suplimentare!

Memoria de instrucțiuni va fi memoria ROM din laboratorul anterior, cu programul scris în cod mașină. (!) Nu măriți dimensiunea memoriei ROM (la 2^{16} locații), ci folosiți mai puțini biți din cei 16 ai PC pentru adresarea ROM (cei mai puțin semnificativi, 4, 5, 6 după caz).

Sumatorul se va descrie cu +1 în VHDL (nu +2 pentru MIPS 16, respectiv +4 pentru MIPS 32), deoarece în VHDL memoria ROM ați declarat-o având cuvântul pe 16 biți (în loc de 8). De aici încolo vom referi ieșirea PC + 4 ca PC + 1 pentru varianta pe 16 biți.

Contorul de program, registrul PC, va fi un registru (bistabil D, 16 biți) cu scriere pe front crescător.

Atenție! Pentru a avea un bun control al circuitului la testarea pe placă, scrierea pe front de ceas a lui PC cu noua valoare se va face doar la apăsarea unui buton de pe placa Basys. Folosiți un semnal de enable de la MPG, ca intrare în entitatea IF, pentru a valida scrierea în PC. Această idee se va repeta în viitor pentru fiecare element unde se fac scrieri sincrone (blocul de registre și memoria de date). În plus, folosiți un al doilea buton (altă instanță a MPG, a doua intrare de enable în entitatea IF) pentru a reseta registrul PC la valoarea zero (se va reveni ușor la prima instrucțiune din ROM în timpul testării).

5.3.2. Testarea unității IF

În entitatea test_env, declarați și instanțiați unitatea IF. Conectați unitatea IF împreună cu MPG (2 instanțe / enable) și cu afișorul SSD disponibile deja în proiectul *test_env*.

Pentru conectarea cu SSD, se vor afișa ambele ieșiri din IF (instrucțiunea și PC + 1) folosind un mecanism de multiplexare. Folosiți switch-ul sw(7) pentru selecția multiplexorului:

- sw(7) = 0 → se afișează instrucțiunea pe SSD
- sw(7) = 1 → se afișează următoarea valoare secvențială a PC, și anume PC + 1, pe SSD.

Folosiți cele două butoane care trec prin MPG pentru a reseta, respectiv a controla scrierea în registrul PC. Veți simula astfel fluxul normal, secvențial, de execuție a instrucțiunilor.

Pentru testarea salturilor se vor simula astfel de salturi, prin legarea a două switch-uri pe semnalele de control care intră în IF:

- Folosiți sw(0) pentru semnalul de control Jump.
- Folosiți sw(1) pentru semnalul de control PCSrc.

De asemenea, deoarece elementele care calculează adresele de salt se vor descrie doar în laboratoarele viitoare, folosiți valori constante (“hard-coded”) în descrierea VHDL, pe care le veți mapa la intrările Jump Address și Branch Address ale componentei IF:

- De exemplu: puteți folosi x”0000” pentru Jump Address ca un mecanism alternativ de resetare a PC (salt la prima instrucțiune)
- Folosiți o valoare intermediară pentru adresa de branch (această valoare trebuie să fie adresa unei instrucțiuni din interiorul programului existent în memoria ROM de instrucțiuni, codificat binar).

5.4. Referințe

[1] Arhitectura Calculatoarelor, note de curs (online) 3 & 4.
[2] D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
[3] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012.
[4] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02.
[5] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
- Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.

Laboratorul 6

6. Procesorul MIPS, ciclu unic – versiune pe 16 biți (3)

Unitatea de decodificare a instrucțiunilor ID (Instruction Decode)
Unitatea de Control UC

6.1. Obiective

Studiul, proiectarea, implementarea și testarea:

- **Unității de decodificare a instrucțiunii, ID, pentru procesorul MIPS, pe 16 biți, un ciclu de ceas / instrucțiune (ciclu unic)**
- **Unității principale de control, UC, pentru procesorul MIPS, pe 16 biți, un ciclu de ceas / instrucțiune (ciclu unic).**

6.2. Descrierea procesorului MIPS, simplificat pe 16 biți - continuare

(!) Este obligatorie studierea cursurilor 3 și 4 pentru a înțelege activitățile din acest laborator, precum și cunoașterea în detaliu a noțiunilor din laboratoarele 4 și 5.

Ciclul de execuție a unei instrucțiuni MIPS are următoarele etape / faze (curs 4):

- IF – Extragerea Instrucțiunii / Instruction Fetch
- ID/OF – Decodificarea Instrucțiunii / Extragerea Operanzilor
  Instruction Decode / Operand Fetch
- EX – Execuție / Execute
- MEM – Memorie / Memory
- WB – Scriere Rezultat / Write Back.

Implementarea proprie a procesorului MIPS-16, pe care o continuați în acest laborator (și o veți termina în următoarele) este partiționată în 5 componente (entități noi). Aceste componente se vor declara și instanția în proiectul *test_env*, în entitatea principală (test_env, probabil…). Utilitatea acestei partiționări o veți înțelege în laboratoarele viitoare, când se va implementa versiunea pipeline a procesorului MIPS 16!

În acest laborator veți proiecta, descrie în VHDL și implementa / testa unitatea de decodificare a instrucțiunii Instruction Decode – ID, împreună cu unitatea principală de control UC, pentru versiunea proprie a procesorului MIPS 16 cu ciclu unic de ceas pe instrucțiune.

Calea de date a procesorului MIPS (versiunea 32 de biți [2]) este prezentată în figura următoare, împreună cu unitatea de control (și semnalele aferente). Pentru a evita aglomerarea schemei, semnalele de control nu au mai fost legate explicit la destinații, dar se pot identifica ușor după nume.

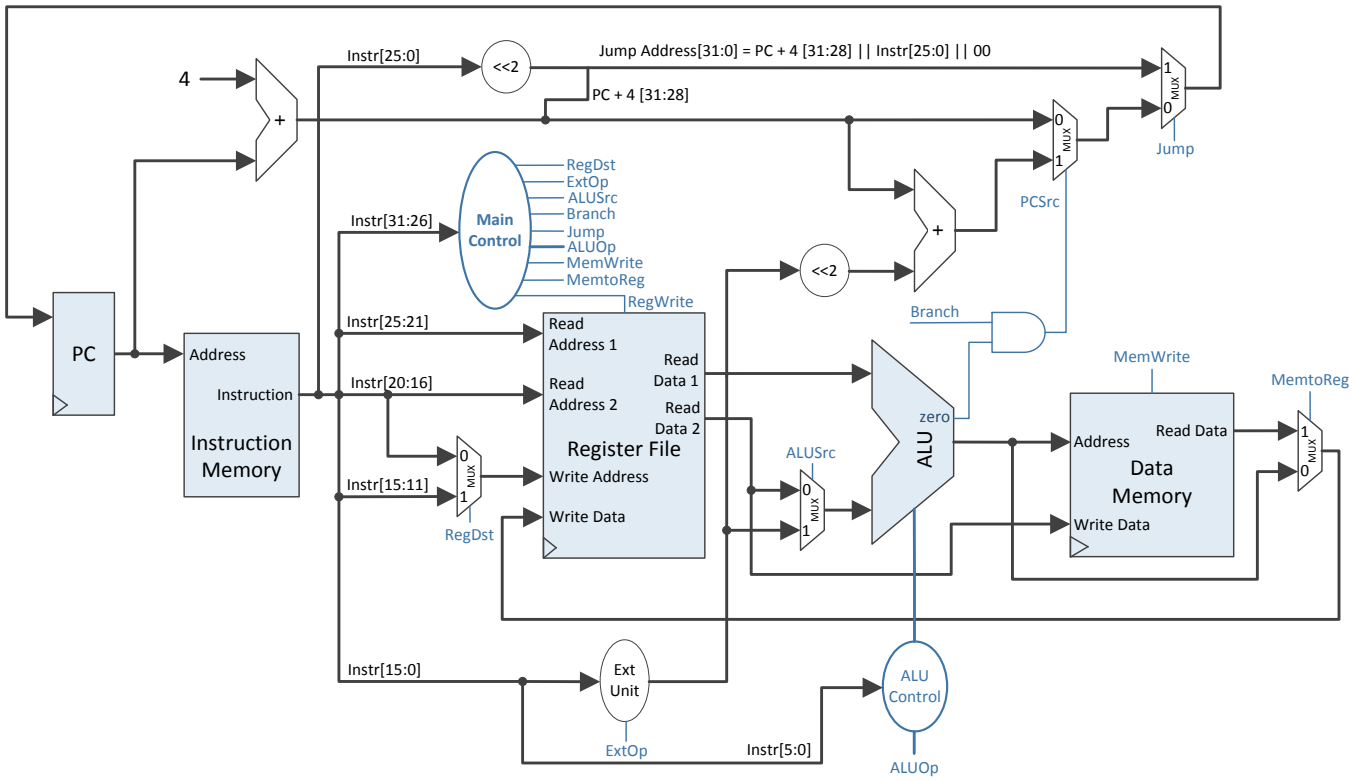


Figura 6-1: MIPS 32 cu ciclu unic de ceas, calea de date + control

Mai jos sunt revizitate cele 3 formate de instrucțiuni pentru MIPS 16, descrise în laboratoarele anterioare:

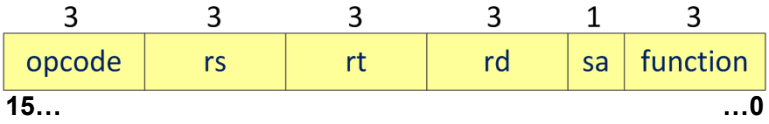


Figura 6-2: Instrucțiune de tip R

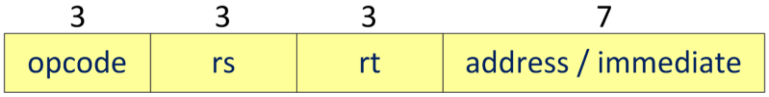


Figura 6-3: Instrucțiune de tip I

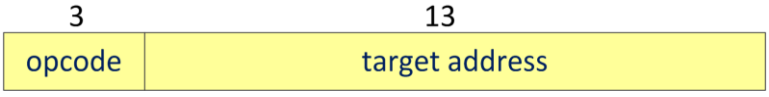


Figura 6-4: Instrucțiune de tip J

Unitatea de decodificare a instrucțiunilor ID (include extragerea operanzilor) conține următoarele elemente principale:

- Bloc de registre / Register File – RF
- Multiplexor
- Unitate de extensie semn / zero.

Consultați laboratorul 4 pentru caracteristicile acestor elemente din procesorul MIPS 16. Amintiți-vă, din laboratorul 3, despre blocul de registre RF,

unde citirile sunt asincrone, și doar scrierea este sincronă, pe front crescător de ceas.

Calea de date MIPS 32 pentru unitatea ID este prezentată în Figura 6-5.

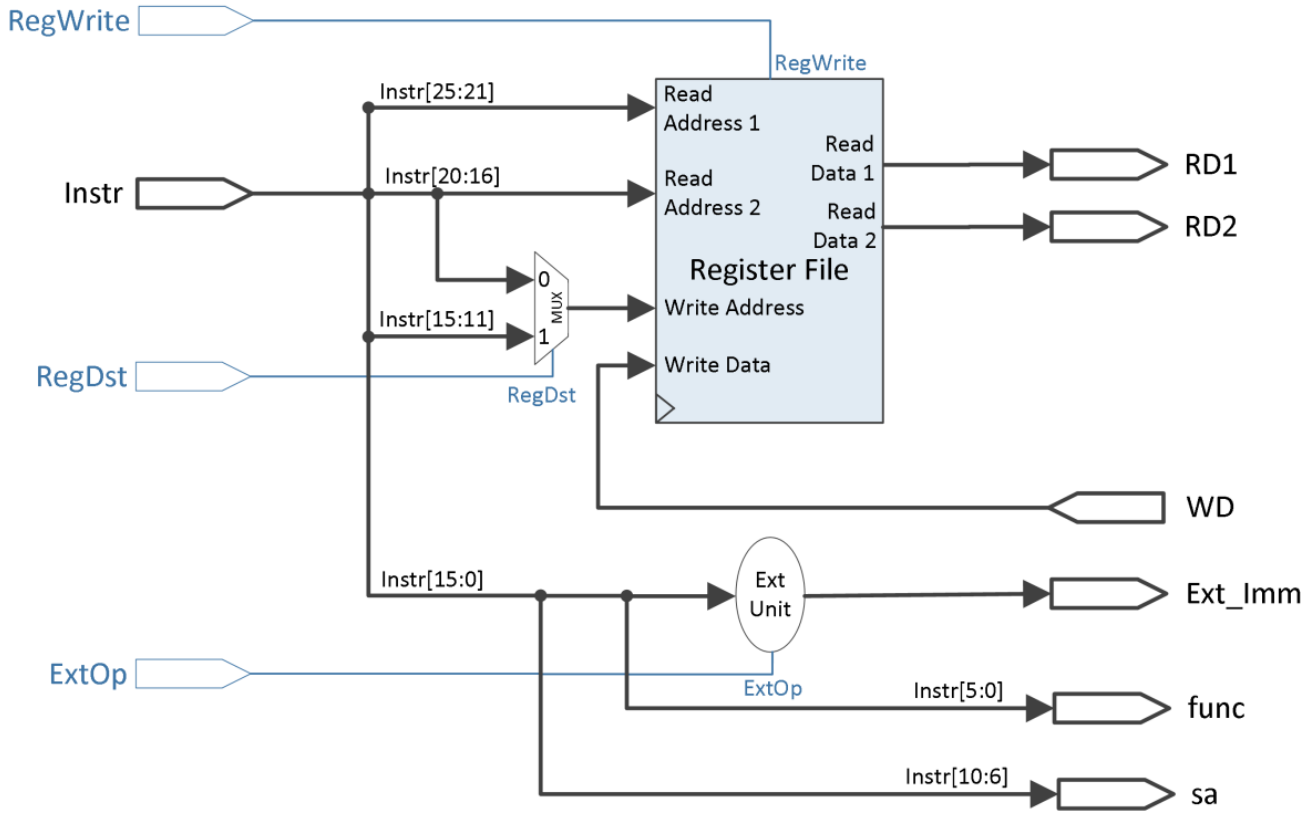


Figura 6-5: Unitatea ID, pentru MIPS 32, calea de date

Unitatea ID are ca ieșiri câmpurile de date (Read Data 1, Read Data 2 și Extended Immediate) folosite de procesor în fazele următoare de execuție ale instrucțiunii. În plus, câmpul function este furnizat mai departe către unitatea locală de control ALU.

Intrările unității ID (MIPS 32) sunt:

- Semnalul de ceas folosit pentru scriere în RF
- Instrucțiunea Instr pe 32 de biți
- Datele care se scriu în RF, 32 de biți, pe WD
- Semnale de control:
  - RegWrite – validarea scrierii în RF
  - RegDst – selectează adresa de scriere în RF
  - ExtOp – selectează tipul de extensie pentru câmpul immediate: cu zero sau cu semn.

Ieșirile unității ID sunt:

- Valoarea registrului de la adresa rs, 32-biți, RD1
- Valoarea registrului de la adresa rt, 32-biți, RD2
- Imediatul extins la 32-biți, Ext_Imm
- Câmpul func, pe 6 biți
- Câmpul sa, pe 5 biți.

Semnificația semnalelor de control este:

- RegDst = 1 → pe Write Address din RF ajunge câmpul rd al instrucțiunii (Instr[15:11])
- RegDst = 0 → pe Write Address din RF ajunge câmpul rt al instrucțiunii (Instr[20:16])

- RegWrite = 1 → se validează, pe front crescător de ceas, scrierea valorii pe 32 de biți de pe Write Data în registrul dat de Write Address, în RF

- ExtOp = 0 → imediatul pe 16 biți este extins cu zero
- ExtOp = 1 → imediatul pe 16 biți este extins cu semn.

Unitatea de control principală UC, cu semnalele de control generate, este prezentată în Figura 6-6. Consultați cursul 4 ([1]) sau [2] pentru descrierea completă a acestor semnale pentru procesorul MIPS.

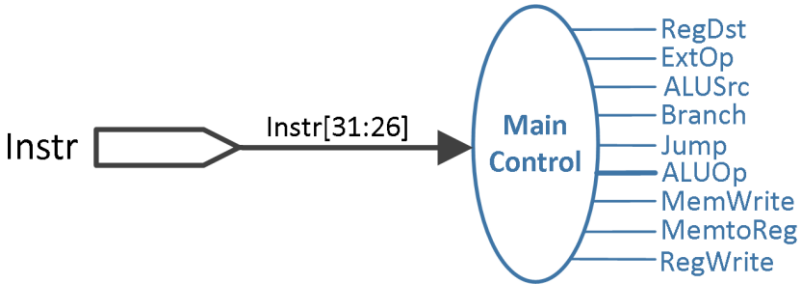


Figura 6-6: Unitatea principală de control UC pentru MIPS 32, cu ciclu unic

Intrarea în UC constă din cei 6 biți ai codului de operație (câmpul opcode), iar ieșirile sunt reprezentate de semnalele de control care merg în calea de date (mai puțin semnalul ALUCtrl, care va merge la unitatea secundară de control ALU). Există 8 semnale de 1 bit, iar semnalul ALUOp este de 2 sau mai mulți biți, în funcție de necesar (pentru cele 15 instrucțiuni pe care le-ați ales). Din acest motiv, în figura de mai sus, ALUOp este desenat cu linie îngroșată. În funcție de instrucțiunile alese pe lângă cele impuse, e posibil să aveți în versiunea proprie a MIPS 16 și alte semnale de control.

6.3. Activități practice

Citiți fiecare activitate în întregime, înainte să o începeți!

Resurse necesare (de avut la începerea laboratorului!):

- Rezultatele obținute pentru activitățile din laboratoarele 3 și 4
- Proiectul *test_env*, cu unitatea IF implementată, fără erori (Laboratorul 5, activitatea 5.3.2).

6.3.1. Proiectarea / implementarea unității ID

Ținând cont de descrierea unității ID din Figura 6-5, descrieți o nouă componentă (entitate) pentru ID în proiectul *test_env*, ca parte a procesorului MIPS 16. Toate câmpurile de date sunt pe 16 biți!

Entitatea ID va conține elementele prezentate în Figura 6-5, care nu se vor descrie cu entități suplimentare, cu excepția RF (vezi laboratorul 3)!

Atenție! Pentru scrierea în blocul de registre RF, este necesar ca scrierea să fie controlată și de la unul dintre butoane, la fel cum este controlată scrierea în registrul PC. Practic, în RF trebuie folosită ieșirea enable a aceluiași MPG folosit pentru activarea scrierii în PC (din unitatea IF), ca o condiție suplimentară (se adaugă un *if*) pe lângă testarea frontului de ceas și a lui RegWrite.

Folosiți o abordare simplă pentru descrierea unității de extensie, ca în laboratorul 2 (when / else, case, sau if…)

Atenție! Aveți grijă la transformarea câmpurilor de date din Figura 6-5 (MIPS 32) în versiunea proprie de implementare a MIPS pe 16 biți. Citiți în avans paragraful cu **Atenție**! din secțiunea de testare 6.3.3.

6.3.2. Proiectarea / implementarea unității UC

Primul pas este identificarea valorilor pentru fiecare semnal de control, în funcție de instrucțiune. Vezi tabelul completat pentru activitatea 4.3.3 din laboratorul 4! În cazul în care nu aveți la dispoziție acest tabel, care trebuia făcut ca tema pentru laboratorul anterior, consultați cursul 4 și stabiliți valorile pentru semnalele de control ale celor 15 instrucțiuni (pentru a reuși testarea pe placă, în timp util, scrieți aceste semnale pentru 4-6 instrucțiuni, și terminați acasă).

Unitatea UC se poate implementa fie ca o nouă entitate în proiectul *test_env*, fie ca un simplu proces de decodificare în arhitectura test_env. Ca o sugestie suplimentară, semnalele de control se recomandă a fi declarate individual, pentru o bună lizibilitate a descrierii VHDL. Pentru descriere se recomandă folosirea unui proces, cu o structură case (după *opcode*). Pentru a evita ca pe fiecare ramură „when” a *case*-ului să se enumere și atribuie toate semnalele de control, acestea se vor inițializa cu valoarea 0 în același proces dar înainte de *case*, urmând ca pe fiecare ramură să fie puse pe 1 doar semnalele necesare (restul rămân implicit pe 0 dacă nu se specifică altfel).

6.3.3. Testarea unităților ID și UC

Lucrați în proiectul *test_env*, finalizat din laboratorul anterior! În entitatea test_env declarați și instanțiați unitatea ID, respectiv unitatea UC (dacă ați declarat-o ca entitate).

Conectați unitatea ID împreună cu unitatea existentă IF. Ieșirea unității IF, cei 16 biți ai instrucțiunii, reprezintă o intrare în ID.

Conectați semnalele necesare generate de UC la unitățile IF și ID.

Următoarele elemente din fluxul de execuție se vor implementa abia în laboratoarele care urmează. Din acest motiv, pentru a verifica în acest laborator (!) scrierea în RF folosiți sumatorul din laboratorul 3 pentru a aduna ieșirile RD1 și RD2 ale ID, iar ieșirea din sumator se leagă pe intrarea WD a ID. **Observație**: acum RegWrite este controlat de către unitatea de control, deci scrierea sumei în

RF se va face doar pentru acele instrucțiuni din programul vostru unde se face scriere în blocul de registre RF.

Pentru a conecta cu SSD semnalele prezente în calea de date, de la IF și ID, trebuie să extindeți multiplexorul cu două intrări din laboratorul anterior, folosind ca selecție 3 switch-uri:

- sw(7:5) = 000 → se afișează instrucțiunea pe SSD
- sw(7:5) = 001 → se afișează următoarea valoare secvențială a PC, și anume PC + 1, pe SSD
- sw(7:5) = 010 → se afișează RD1 pe SSD
- sw(7:5) = 011 → se afișează RD2 pe SSD
- sw(7:5) = 100 → se afișează WD pe SSD
- …restul semnalelor relevante pentru testare…

În timpul testării este foarte important să puteți vizualiza valoarea curentă a semnalelor de control. Există (cel puțin) 8 semnale de 1 bit, plus ALUOp. Se vor folosi ledurile plăcii Basys 3 pentru a afișa valoarea semnalelor de control.

La fel ca în laboratorul anterior, folosiți cele două butoane care trec prin MPG pentru a reseta PC, respectiv a controla scrierea în registrul PC și în RF. Veți simula astfel fluxul normal, secvențial, de execuție a instrucțiunilor.

Deocamdată, la fel ca în laboratorul anterior, folosiți valori constante (“hard-coded”) în descrierea VHDL, mapate la intrările Jump Address și Branch Address ale componentei IF.

6.4. Referințe

[1] Arhitectura Calculatoarelor, note de curs (online) 3 & 4.
[2] D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
[3] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012.
[4] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02.
[5] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
- Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.

Laboratorul 7

7. Procesorul MIPS, ciclu unic – versiune pe 16 biți (4)

Unitatea de execuție a instrucțiunilor (Instruction Execute) EX
Unitatea de Memorie MEM
Unitatea de scriere a rezultatelor (Write-Back) WB

7.1. Obiective

Studiul, proiectarea, implementarea și testarea, pentru procesorul MIPS, pe 16 biți, un ciclu de ceas / instrucțiune (ciclu unic) a:

- **Unității de execuție a instrucțiunii, EX**
- **Unității de memorie, MEM**
- **Unității de scriere a rezultatului, WB**
- **Restului conexiunilor necesare pentru terminarea implementării MIPS 16.**

7.2. Descrierea procesorului MIPS, simplificat pe 16 biți - continuare

(!) Este obligatorie studierea cursurilor 3 și 4 pentru a înțelege activitățile din acest laborator, precum și cunoașterea în detaliu a noțiunilor din laboratoarele 4, 5 și 6!

Ciclul de execuție a unei instrucțiuni MIPS are următoarele etape / faze (curs 4):

- IF – Extragerea Instrucțiunii / Instruction Fetch
- ID/OF – Decodificarea Instrucțiunii / Extragerea Operanzilor
  Instruction Decode / Operand Fetch
- EX – Execuție / Execute
- MEM – Memorie / Memory
- WB – Scriere Rezultat / Write Back.

Implementarea proprie a procesorului MIPS-16, pe care o finalizați în acest laborator este partiționată în 5 componente (entități noi). Aceste componente au fost sau se vor declara și instanția în proiectul *test_env*, în entitatea principală (test_env, probabil…).

În acest laborator veți proiecta, descrie în VHDL, și implementa / testa unitatea de execuție a instrucțiunii Instruction Execute – EX, unitatea de memorie – MEM, unitatea de scriere a rezultatului Write Back – WB, precum și restul conexiunilor necesare pentru versiunea proprie a procesorului MIPS 16 cu ciclu unic de ceas pe instrucțiune.

Calea de date a procesorului MIPS (versiunea 32 de biți [2]) este prezentată în figura următoare, împreună cu unitatea de control (și semnalele aferente).

Pentru a evita aglomerarea schemei, semnalele de control nu au mai fost legate explicit la destinații, dar se pot identifica ușor după nume.

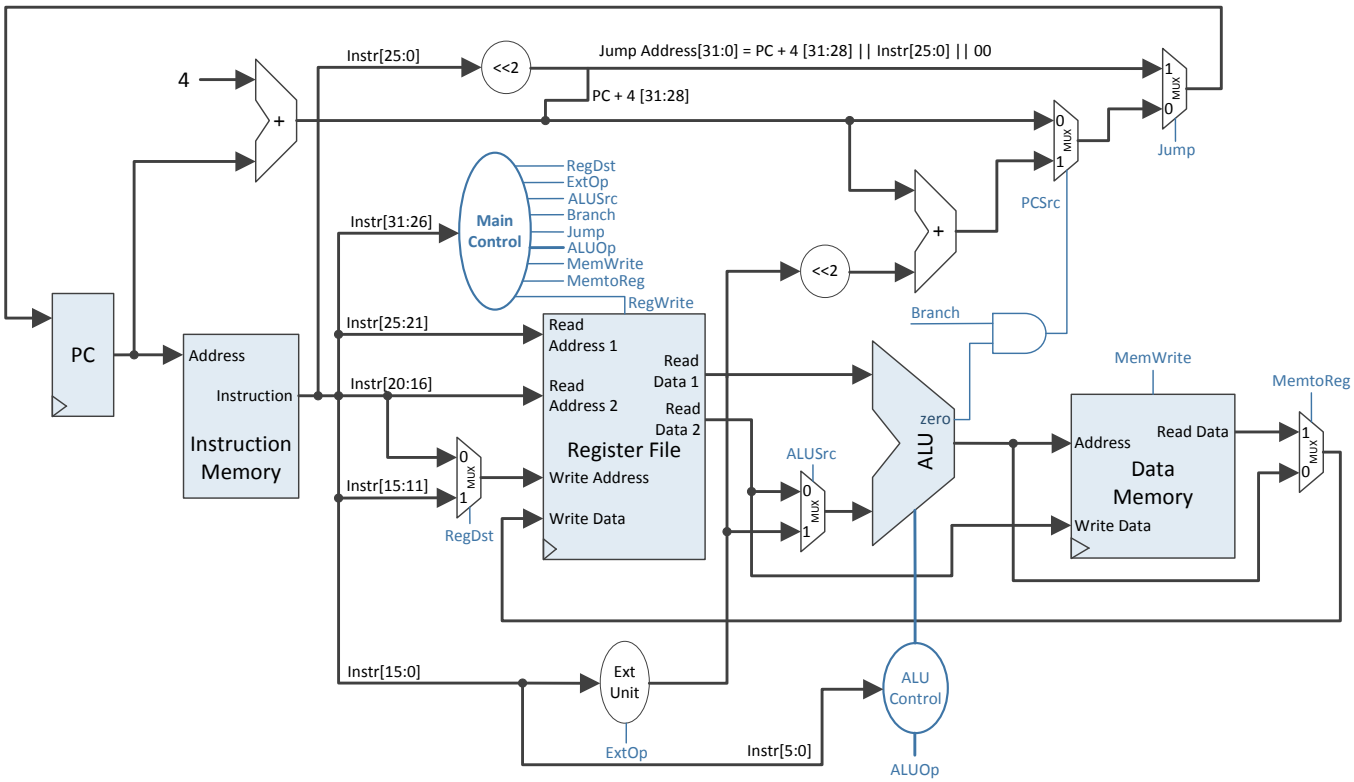


Figura 7-1: MIPS 32 cu ciclu unic de ceas, calea de date + control

Mai jos sunt revizitate cele 3 formate de instrucțiuni pentru MIPS 16, descrise în laboratoarele anterioare:

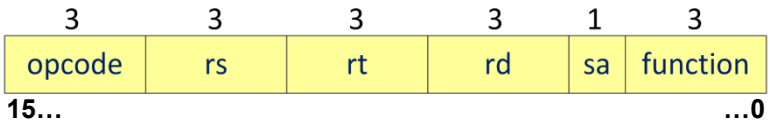


Figura 7-2: Instrucțiune de tip R

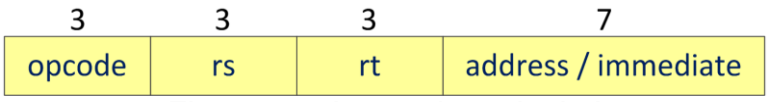


Figura 7-3: Instrucțiune de tip I

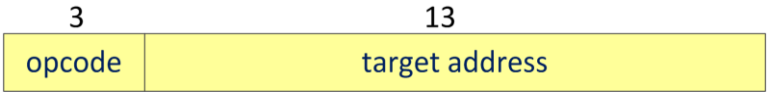


Figura 7-4: Instrucțiune de tip J

Unitatea de execuție EX conține următoarele elemente principale:

- Unitatea aritmetică-logică (ALU)
- Unitatea locală de control ALU
- Multiplexor
- Deplasare la stânga cu 2 și sumatorul pentru calculul adresei de salt (branch).

Consultați laboratoarele 2 și 4 pentru caracteristicile acestor elemente pentru procesorul MIPS 16.

Calea de date MIPS 32 pentru unitatea EX este prezentată în Figura 7-5.

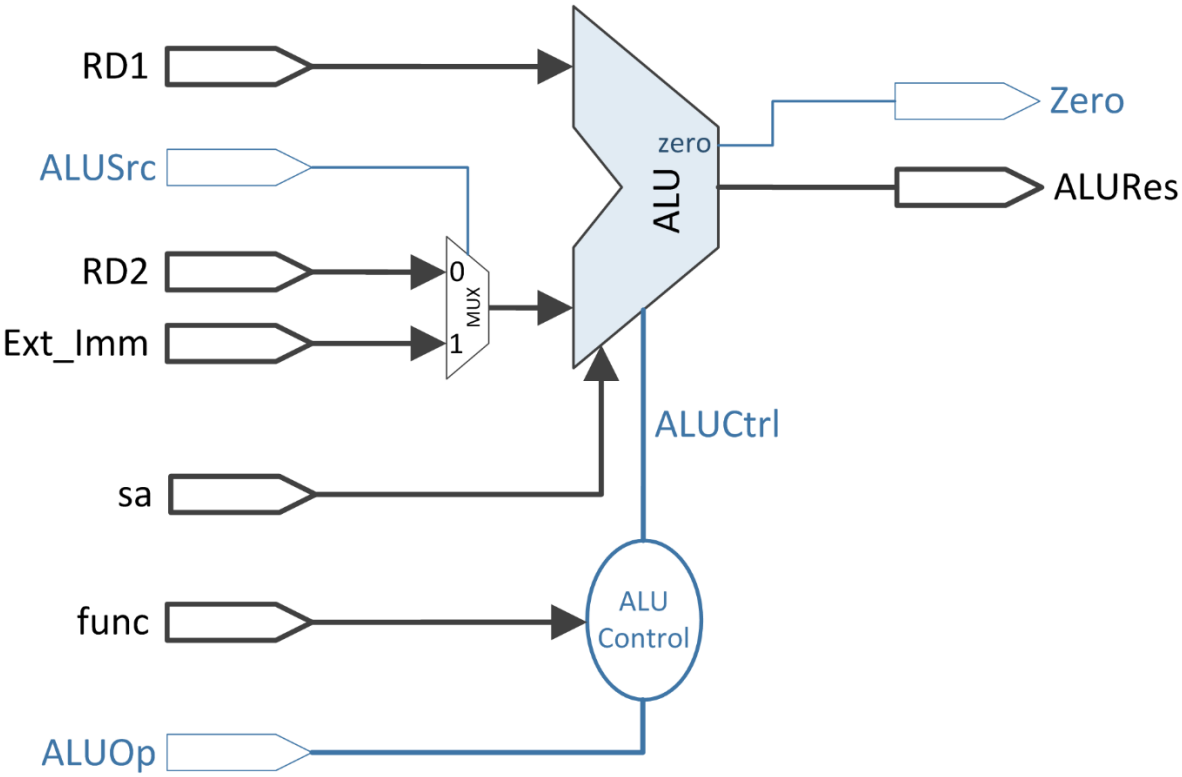


Figura 7-5: Unitatea EX, pentru MIPS 32, calea de date

Unitatea EX furnizează ca ieșire rezultatul ALU care va fi folosit fie ca rezultat al instrucțiunilor artimetice-logice și scris în blocul de registre RF, fie ca adresă pentru Memoria de Date în cazul instrucțiunilor lw și sw. În plus, ALU furnizează pentru ieșire semnalul de stare Zero, care arată dacă rezultatul operației curente din ALU este egal cu 0 (Zero = 1, dacă rezultatul este 0, sau 0, altfel). Pentru a ușura în viitor conversia procesorului MIPS 16 în implementarea pipeline, unitatea EX conține, de asemenea, calculul adresei de salt condiționat (branch).

Intrările unității EX (MIPS 32) sunt:

- Adresa următoarei instrucțiuni de executat, mod secvențial (PC + 4)
- Valoarea registrului de la adresa rs, 32-biți, RD1
- Valoarea registrului de la adresa rt, 32-biți, RD2
- Imediatul extins la 32-biți, Ext_Imm
- Câmpul func, pe 6 biți
- Câmpul sa, pe 5 biți
- Semnale de control:
  - ALUSrc – selecție între Read Data 2 și Ext_Imm pentru intrarea a doua din ALU
  - ALUOp – codul pentru operația ALU furnizat de către unitatea principală de control UC.

Ieșirile unității EX (MIPS 32) sunt:

- Adresa de salt Branch Address, 32-biți
- Rezultatul ALURes, 32-biți
- Semnalul Zero de 1 bit.

Semnificația semnalelor de control este:

- ALUSrc = 0 → valoarea Read Data 2 merge pe intrarea a doua ALU
- ALUSrc = 1 → valoarea Ext_Imm merge pe intrarea a doua ALU

- ALUOp → este definit în unitatea principală de control UC în conformitate cu operațiile care trebuie executate în ALU.

Adresa de salt condiționat este calculată cu următoarea formulă:

Branch Address ← PC + 4 + S_Ext(Imm) << 2;

Semnalul Zero împreună cu semnalul de control Branch sunt folosite în exteriorul lui EX pentru calcula semnalul PCSrc, folosit în IF pentru a selecta valoarea care va merge spre contorul de program (PC+4 sau Branch Address).

Unitatea locală de control ALU stabilește operațiile de executat de către ALU prin codificarea lor în semnalul ALUCtrl. Pentru instrucțiuni de tip I, codificarea lui ALUCtrl depinde doar de valoarea semnalului ALUOp (setat din unitatea de control UC). Pentru instrucțiuni de tip R, ALUOp are aceeași valoare, iar valoarea lui ALUCtrl este definită pe baza valorii din intrarea func (câmpul function al instrucțiunii).

Unitatea de memorie MEM este simplă, conținând un singur element:

- Memoria de date (Data Memory).

Calea de date MIPS 32 pentru unitatea MEM este prezentată în figura următoare.

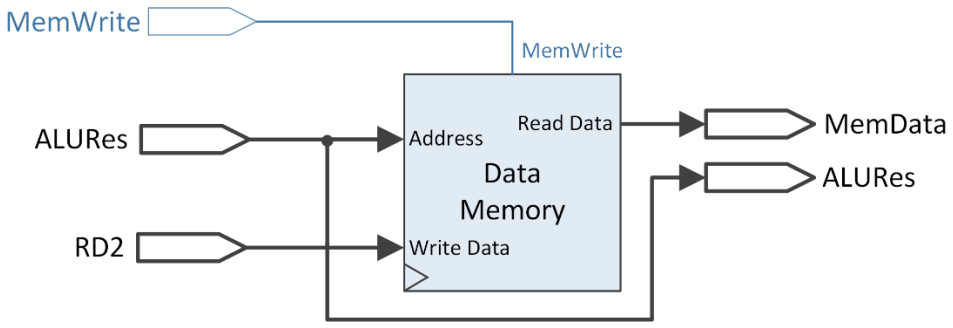


Figura 7-6: Unitatea MEM, pentru MIPS 32, calea de date

Memoria de date este o memorie RAM cu citire asincronă (similar cu RF) și scriere sincronă pe front crescător de ceas, cu validarea scrierii prin semnalul de control MemWrite. O descriere similară pentru RAM (! dar cu citire sincronă) a fost făcută în laboratorul 3.

Intrările unității MEM (MIPS 32) sunt:

- Semnalul de ceas (pentru scriere sincronă în memorie)

- Semnalul ALURes, 32 de biți – furnizează adresa pentru memorie (are efect doar pentru load sau store word)
- Semnalul RD2, 32 de biți – a doua ieșire din blocul de registre RF (util doar pentru instrucțiunea store word) care merge pe câmpul Write Data (portul de scriere) al memoriei
- Semnalul de control MemWrite.

Ieșirile unității MEM (MIPS 32) sunt:

- MemData, 32 de biți, reprezintă datele de pe portul de citire Read Data al memoriei de date (doar pentru instrucțiunea load word)
- ALURes, 32 de biți, acest semnal reprezintă de asemenea rezultatul operațiilor aritmetice-logice, care trebuie stocat în blocul de registre, deci trebuie furnizat la ieșirea unității MEM ca intrare pentru unitatea următoare, WB.

Singurul semnal de control prezent în această unitate este MemWrite:

- MemWrite = 0 → Nu se scrie nimic în memoria de date
- MemWrite = 1 → Valoarea din semnalul RD2 este scrisă sincron la adresa de memorie indicată de valoarea semnalului ALURes.

Unitatea Write Back – WB este simplă, constând din ultimul multiplexor (dreapta) din Figura 7-1, iar restul componentelor neincluse până acum în unitățile descrise sunt: poarta ȘI pentru generarea semnalului PCSrc, și logica de calcul a adresei de salt Jump Address.

Semnalul de control pentru multiplexorul din WB este MemtoReg, el selectând ce valoare se va scrie înapoi în blocul de registre RF din unitatea ID:

- MemtoReg = 0 → valoarea semnalului ALURes va ajunge pe intrarea Write Data a RF,
- MemtoReg = 1 → valoarea semnalului MemData va ajunge pe intrarea Write Data a RF.

Semnalul de control PCSrc se generează cu o poartă ȘI:

- PCSrc ← Branch and Zero.

7.3. Activități practice

Citiți fiecare activitate în întregime, înainte să o începeți!

Resurse necesare (de avut la începerea laboratorului!):

- Rezultatele obținute pentru activitățile din laboratoarele 2, 3, 4, 5 și 6!
- Proiectul *test_env*, cu unitățile IF, ID și UC implementate, legate, fără erori.

7.3.1. Proiectarea / implementarea unității EX

Ținând cont de schema unității EX din Figura 7-5, descrieți o nouă componentă (entitate) pentru EX în proiectul *test_env*, ca parte a procesorului MIPS 16. Toate câmpurile de date sunt pe 16 biți!

Entitatea EX va conține elementele prezentate în Figura 7-5, care nu se vor descrie cu entități suplimentare!

Folosiți o linie de cod VHDL (deplasarea cu 2 poziții nu mai e necesară… de ce?) pentru calculul adresei de salt Branch Address.

Folosiți un proces (cu case…) pentru descrierea ALU, asemănător cu descrierea făcută în laboratorul 2. Deplasamentul pe 1 bit, câmpul sa, este folosit doar pentru operații de deplasare logică sau aritmetică.

Folosiți un proces (tot cu case…) pentru implementarea unității locale de control ALU. Codificarea semnalului ALUCtrl depinde de cele 15 instrucțiuni suportate de varianta voastră a procesorului MIPS 16 (lucru indicat prin ALUOp și/sau function), și definește operațiile aritmetice-logice care trebuie suportate / cunoscute de către ALU. Indiciu: Pentru instrucțiunile de tip R, se poate face pe ramura ”when” corespunzătoare încă un case, după câmpul function...

Atenție! Aveți grijă la transformarea câmpurilor de date din Figura 7-5 (MIPS 32) în versiunea proprie de implementare a MIPS pe 16 biți.

7.3.2. Proiectarea / implementarea unității MEM

Descrieți o nouă componentă MEM (entitate) pentru unitate de memorie, ca în Figura 7-6. Folosiți implementarea RAM prezentată în laboratorul 3, și schimbați operația de citire astfel încât să devină asincronă (se scoate de sub testul de front crescător). Descrieți totul doar comportamental în arhitectura entității MEM.

Atenție! Din cauza citirii asincrone, va rezulta o memorie RAM distribuită (vezi laboratorul 3), care nu se poate mapa pe modulele de block RAM ale plăcii. Pentru a evita un necesar de resurse mai mare decât suportă placa de dezvoltare, asigurați-vă că memoria RAM are o dimensiune redusă (ex. 32, 64, 128 de elemente, în loc de 2^{16}) iar la adresă folosiți biții cei mai puțin semnificativi din ALURes (câți sunt necesari).

Atenție! Pentru memoria RAM, este necesar ca scrierea să fie controlată de la unul dintre butoane, la fel cum este controlată scrierea în registrul PC, respectiv în RF. Practic, în memoria RAM trebuie folosită ieșirea enable a aceluiași MPG folosit pentru activarea scrierii în PC (din unitatea IF) și în RF (din unitatea ID), ca o condiție suplimentară (if) pe lângă testarea frontului de ceas și a lui MemWrite.

7.3.3. Descrierea restului logicii necesare pentru procesor

În entitatea test_env declarați și instanțiați unitățile EX și ID. Conectați toate semnalele acestor unități în calea de date existentă.

Adăugați multiplexorul pentru unitatea WB direct în arhitectura test_env.

Finalizați implementarea procesorului cu descrierea, în arhitectura test_env, a logicii de calcul a adresei de salt Jump Address (fără circuitul de deplasare cu 2 poziții… de ce?), și o poartă ȘI pentru evaluarea semnalului PCSrc. Completați restul conexiunilor necesare în calea de date / control, ca în Figura 7-1 (Jump Address, Branch target address, write back la RF etc.).

7.3.4. Aproape de destinație: testarea propriului procesor MIPS 16

În acest moment trebuie să aveți procesorul MIPS 16 implementat complet în proiectul *test_env*. În funcție timpul disponibil (alocat laboratorului curent), se

poate testa procesorul. Testarea este subiectul principal pentru următorul laborator.

Pentru afișarea pe SSD a semnalelor relevante din calea de date, de la toate unitățile, trebuie să extindeți multiplexorul din laboratorul anterior:

- sw(7:5) = 000 → se afișează instrucțiunea pe SSD
- sw(7:5) = 001 → se afișează următoare valoare secvențială a PC (PC + 1), pe SSD
- sw(7:5) = 010 → se afișează RD1 pe SSD
- sw(7:5) = 011 → se afișează RD2 pe SSD
- sw(7:5) = 100 → se afișează Ext_Imm pe SSD
- sw(7:5) = 101 → se afișează ALURes pe SSD
- sw(7:5) = 110 → se afișează MemData pe SSD
- sw(7:5) = 111 → se afișează WD pe SSD.

În timpul testării este foarte important să puteți vizualiza valoarea curentă a semnalelor de control. Există (cel puțin) 8 semnale de 1 bit, plus ALUOp. Se vor folosi ledurile plăcii Basys 3 pentru a afișa valoarea semnalelor de control.

Dacă este necesar, pentru depanarea procesorului MIPS (= trasarea atentă pas cu pas a programului scris în ROM) pe placa de dezvoltare, se pot afișa și alte semnale pe SSD / LED-uri: Branch Address, Jump Address, ALUCtrl etc.

Acum, încărcați procesorul pe placa de dezvoltare, și executați pas cu pas (de la buton prin MPG) programul din memoria ROM. Verificați cu atenție, la fiecare instrucțiune, valorile prezente pe semnalele relevante din calea de date, afișate pe SSD / LED-uri, pentru a stabili dacă instrucțiunea se execută corect. Verificați scrierea în memorie / RF urmărind valoarea locațiilor destinație din instrucțiunea curentă în instrucțiunile următoare, unde ele apar ca operanzi sursă (le vedeți pe SSD valorile din calea de date). **ETC…**

Temă: (ușurează mult testarea pe placă, instrument de scris / hârtie) - trasarea execuției programului pe hârtie, dacă există un ciclu repetitiv – cel puțin o iterație. Pentru fiecare instrucțiune trasată se notează pe hârtie operanzii sursă, rezultatul, valorile semnalelor interne relevante (RD1, RD2, ALURes, Branch Address etc.) care se pot afișa pe SSD.

7.4. Referințe

[1] Arhitectura Calculatoarelor, note de curs (online) 3 & 4.
[2] D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
[3] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012.
[4] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02.
[5] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
- Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.

Laboratorul 8

8. Procesorul MIPS, ciclu unic – versiune pe 16 biți (5)

Evaluarea / testarea / corectarea procesorului

8.1. Obiective

În acest laborator se va testa funcționarea procesorului pe placa de dezvoltare. Procesul de testare se bazează mai ales pe urmărirea execuției programului încărcat anterior (laboratorul 4) în memoria ROM a procesorului.

Dacă e cazul, se vor identifica și corecta eventuale erori rezultate din proiectare sau din descrierea VHDL.

8.2. Testarea propriului procesor MIPS 16

În acest moment trebuie să aveți procesorul MIPS 16 implementat complet și corect, conform cerințelor din laboratoarele anterioare, în proiectul *test_env*.

Pentru afișarea pe SSD a semnalelor relevante din calea de date, de la toate unitățile, aveți la dispoziție din laboratorul anterior:

- sw(7:5) = 000 → se afișează instrucțiunea pe SSD
- sw(7:5) = 001 → se afișează următoare valoare secvențială a PC (PC + 1), pe SSD
- sw(7:5) = 010 → se afișează RD1 pe SSD
- sw(7:5) = 011 → se afișează RD2 pe SSD
- sw(7:5) = 100 → se afișează Ext_Imm pe SSD
- sw(7:5) = 101 → se afișează ALURes pe SSD
- sw(7:5) = 110 → se afișează MemData pe SSD
- sw(7:5) = 111 → se afișează WD pe SSD
- dacă este necesar pentru depanarea procesorului MIPS (= trasarea atentă pas cu pas a programului scris în ROM) pe placa de dezvoltare, se pot afișa și alte semnale pe SSD / LED-uri: Branch Address, Jump Address, ALUCtrl etc. De exemplu, dacă nu funcționează BEQ afișati suplimentar imediatul extins cu semn și Branch Address pentru a depista mai ușor unde este eroarea.

În timpul testării este foarte important să puteți vizualiza valoarea curentă a semnalelor de control. Există (cel puțin) 8 semnale de 1 bit, plus ALUOp. Se vor folosi ledurile plăcii Basys 3 pentru a afișa valoarea semnalelor de control.

Acum, încărcați procesorul pe placa de dezvoltare, și executați pas cu pas (de la buton prin MPG) programul din memoria ROM. Verificați cu atenție, la fiecare instrucțiune, valorile prezente în semnalele relevante din calea de date, afișate pe SSD / LED-uri, pentru a stabili dacă instrucțiunea se execută corect. Folosiți

trasarea programului făcută pe hârtie, la temă, pentru a valida valorile obținute în timpul rulării pe placă.

În funcție de corectitudinea implementării, există două posibile abordări.

Varianta 1 (the easy way out): executați până la sfârșit programul și, la ultimele instrucțiuni, vedeți rezultatul corect (ex. suma șirului, sau calculul unei formule etc.) caz în care procesorul funcționează corect.

Varianta 2 (no easy way out): rezultatul final nu e cel așteptat, deci se verifică pas cu pas fiecare instrucțiune, inclusiv semnalele de control. Vezi în continuare indicii!

Se verifică succesiunea corectă a instrucțiunilor, inclusiv funcționarea instrucțiunilor de salt: în principal se urmărește cum se modifică valoarea lui PC.

La instrucțiuni de tip R sau I se verifică propagarea corectă a datelor plecând de la codul mașină al instrucțiunii, ieșirile din blocul de registre, imediatul extins, ieșirea ALURes, WD etc., prin vizualizarea acestor câmpuri pe afișorul de 7 segmente. Aici este foarte utilă trasarea în detaliu a programului, făcută anterior pe hârtie.

Pentru lw/sw cu memoria, se procedează la fel.

Pentru a testa că rezultatele instrucțiunilor se scriu înapoi în registrul destinație sau în locația de memorie, se folosește următoarea strategie: se urmărește valoarea locațiilor destinație din instrucțiunea curentă în instrucțiunile următoare în care ele apar ca operanzi sursă (le vedeți pe SSD valorile din calea de date). De exemplu:

addi $2, $0, 5 -- în $2 ar trebui să se încarce valoarea 5
….
Alte instrucțiuni (care nu modifică $2!)
….
add $3, $2, $4 -- în acest moment, pe câmpul RD1 la ieșirea RF ar trebui să fie valoarea 5 (vizualizare pe SSD)

În cazul în care ceva nu coincide cu ce ați trasat pe foaie, localizați eroarea (unde e în procesor) și reveniți la codul VHDL pentru a încerca să o corectați.

8.3. Referințe

[1] Arhitectura Calculatoarelor, note de curs (online) 3 & 4.
[2] D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
[3] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012.
[4] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02.
[5] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
- Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.

Laboratorul 9

9. Procesorul MIPS, pipeline – versiune pe 16 biți (1)

Transformarea versiunii MIPS 16 ciclu unic în versiunea pipeline

9.1. Obiective

Studiul, proiectarea și implementarea:

- **Procesorului MIPS 16, versiunea pipeline.**

9.2. Transformarea procesorului MIPS 16, versiunea ciclu unic, într-un procesor de tip pipeline

Calea de date a procesorului MIPS (versiunea 32 de biți [2]) este prezentată în figura următoare, împreună cu unitatea de control (și semnalele aferente). Pentru a evita aglomerarea schemei, semnalele de control nu au mai fost legate explicit la destinații, dar se pot identifica ușor după nume.

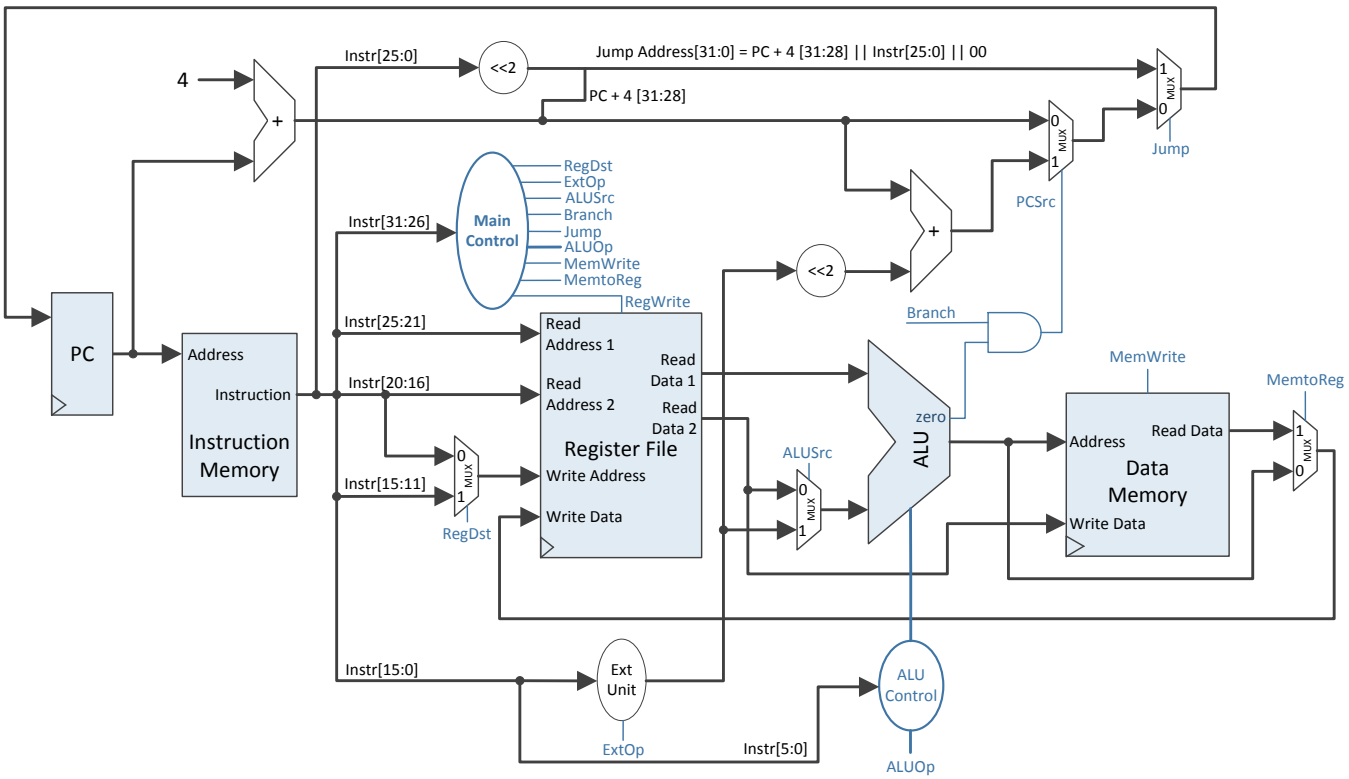


Figura 9-1: MIPS 32 cu ciclu unic de ceas, calea de date + control

Ciclul de execuție a unei instrucțiuni MIPS are următoarele etape / faze (laboratoarele anterioare și curs 4):

- IF – Instruction Fetch
- ID/OF – Instruction Decode / Operand Fetch
- EX – Execute
- MEM – Memory
- WB – Write Back.

Mai jos sunt revizitate cele 3 formate de instrucțiuni pentru MIPS 16, descrise în laboratorul anterior:

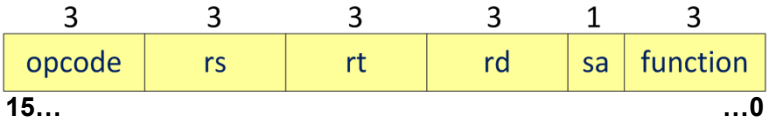


Figura 9-2: Instrucțiune de tip R

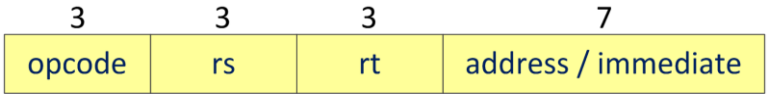


Figura 9-3: Instrucțiune de tip I

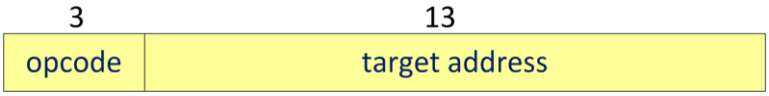


Figura 9-4: Instrucțiune de tip J

Principala problemă a procesorului MIPS cu ciclu unic este lungimea căii critice, cea mai lungă fiind pentru instrucțiunea load word (vezi curs 4, respectiv [2]). Timpul necesar pentru transmiterea (combinațională) și stabilizarea nivelurilor logice de-a lungul căii critice trebuie să fie acoperit de durata ciclului de ceas. Astfel, rezultă un ciclu lung de ceas, care este potrivit doar pentru lw, pentru restul instrucțiunilor fiind prea mare (se irosește timp de execuție).

Pentru a reduce durata ciclului de ceas, soluția este secționarea căii critice cu elemente de stocare sincrone (de tip registru). Aceste elemente se interpun practic între unitățile procesorului MIPS 32, care coincid și cu fazele de execuție ale instrucțiunii: IF, ID, EX, MEM și WB. Astfel, se pot executa până la 5 instrucțiuni consecutive din program, fiecare aflându-se într-una dintre cele 5 faze de execuție, în funcție de ordinea de succesiune. Unitățile din varianta pipeline vor fi referite în continuare ca **etaje.**

Calea de date și partea de control pentru procesorul MIPS 32, în varianta pipeline, sunt prezentate în Figura 9-5.

Fiecare registru intermediar va fi referit în funcție de poziția lui între etaje. Registrul dintre etajul IF și etajul ID este IF/ID, dintre ID și EX este ID/EX etc.

Rolul registrelor dintre etaje este de a păstra rezultatele intermediare ale instrucțiunilor, pentru a furniza aceste rezultate intermediare etajului următor, în următorul ciclu de ceas.

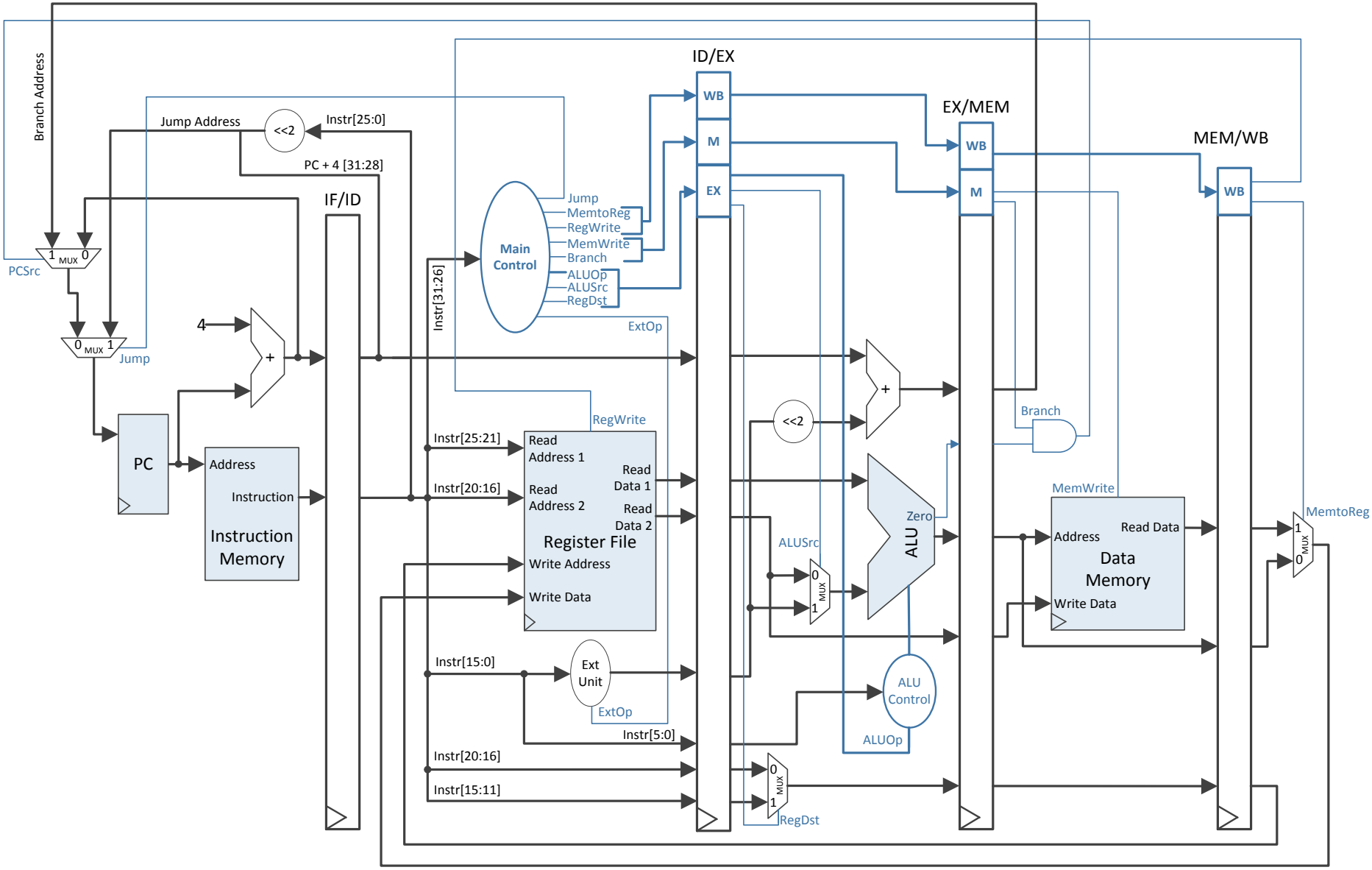


Figura 9-5: Procesorul MIPS 32 pipeline [2], obținut prin secționarea căii de date MIPS 32 cu ciclu unic, din Figura 9-1

În plus, execuția în calea de date depinde de valorile semnalelor de control, valori care sunt specifice fiecărei instrucțiuni. Astfel, prin intermediul registrelor intermediare (începând cu ID/EX) trebuie transmise și valorile semnalelor de control, pentru fiecare etaj următor. Pe schema de pe pagina anterioară, semnalele de control sunt grupate simbolic după numele etajului unde trebuie să ajungă.

Practic, semnalele de control sunt transmise în tandem cu rezultatele intermediare ale instrucțiunii, pentru fiecare etaj care urmează.

La cursul dedicat variantei pipeline a procesorului MIPS se discută pe larg acest subiect, detaliile descrise până acum fiind doar minimul necesar pentru a începe transformarea procesorului din ciclu unic în pipeline.

O diferență notabilă pe schemă, față de gruparea în unități pe care ați făcut-o deja pentru MIPS 16 cu ciclu unic, este că multiplexorul care selectează adresa registrului destinație este plasat în etajul EX, nu în ID, cum îl aveți în varianta proprie. Această diferență este utilă doar dacă în versiunea pipeline s-ar implementa tehnici mai avansate pentru evitarea hazardurilor (laboratorul următor), ca înaintarea datelor (forwarding). Există 2 variante posibile pentru a transpune această modificare în implementarea VHDL:

1. Îl lăsați în ID (așa cum e acum în MIPS 16), în acest caz semnalul de control RegDst nu se mai transmite prin ID/EX ca pe schema pipeline, ci se ia direct de la UC. UC și ID se află în același etaj de pipeline.
2. Îl mutați în EX, modificând porturile de intrare / ieșire ale unităților ID și EX, și transmiteți RegDst conform schemei.

În rest, semnalul de control MemRead se va ignora (nu apare pe schema pipeline din laborator), la fel ca la MIPS 16 cu ciclu unic, pentru a evita modificări suplimentare în calea de date.

Un alt semnal care nu apare explicit pe schema, dar a fost folosit și la MIPS 16 cu ciclu unic, este semnalul *sa*, de 1 bit. Acest semnal trebuie transmis prin registrul intermediar ID/EX către unitatea EX (unde e legat deja la ALU), în caz contrar instrucțiunile de deplasare care îl folosesc nu vor funcționa corect.

9.3. Activități practice

Citiți fiecare activitate în întregime, înainte să o începeți!

Resurse necesare (de avut la începerea laboratorului!):

- Proiect Xilinx cu *test_env*, care să conțină implementarea proprie a procesorului MIPS 16 cu ciclu unic, corectă și completă.

9.3.1. Verificarea procesorului MIPS 16 cu ciclu unic

În cazul în care în laboratorul 8 nu ați reușit testarea completă a procesorului, se recomandă acest lucru înainte de a începe următoarele activități.

Pentru ca versiunea pipeline pe care o veți obține să fie corectă, este obligatoriu ca procesorul (ciclu unic) de la care porniți să fie funcțional.

9.3.2. Proiectarea registrelor intermediare (hârtie / instrument de scris)

Pentru fiecare registru intermediar, enumerați ce câmpuri trebuie să stocheze, în funcție de particularitățile propriei versiuni a procesorului MIPS 16.

Folosiți schema din Figura 9-5, dar (!) țineți cont de particularitățile procesorului vostru: e pe 16, NU pe 32 de biți, și, în funcție de instrucțiunile implementate, calea de date s-ar putea să conțină elemente adiționale.

De exemplu, pentru primul registru intermediar, IF/ID, trebuie memorate două câmpuri (le denumim generic cu numele etajului din care provin):

- IF.PC+1 pe 16 biți
- IF.Instruction pe 16 biți.

Rezultă un necesar de 32 de biți pentru IF/ID. Descrieți în mod similar registrele ID/EX, EX/MEM, MEM/WB.

Pentru fiecare registru intermediar, când faceți descrierea pe biți a câmpurilor de intrare, vizualizați unitățile asociate (cea de intrare, respectiv destinația, ex. pentru IF/ID unitățile IF, respectiv ID) în laboratoarele 5, 6, respectiv 7. Identificați aceste câmpuri în lista de porturi de intrare/ieșire pentru fiecare unitate implicată. Acest pas va fi foarte util pentru activitatea următoare, când registrele trebuie descrise în VHDL.

9.3.3. Descrierea registrelor intermediare în VHDL

Pentru această activitate se va lucra în entitatea principală test_env (unde sunt instanțiate și legate unitățile principale).

Atenție: pe măsură ce avansați cu procesul de secționare, vor apărea mici modificări necesare pentru unitățile existente. Veți remarca aceste modificări dacă studiați cu atenție, în paralel, schema secționată din Figura 9-5, și schemele particulare ale unităților descrise în laboratoarele 5, 6 și 7. De exemplu, unitatea ID necesită adăugarea unui port de intrare pentru adresa de scriere în blocul de registre, adresă care se va întoarce (via port map) din ultimul registru intermediar MEM/WB.

Nu se vor declara entități noi pentru registrele intermediare. Pentru fiecare registru se va declara un semnal de dimensiune potrivită, conform rezultatelor primei activități, iar comportamentul se va descrie printr-un proces, unde are loc un transfer sincron pe front crescător de ceas. Pentru a face posibilă testarea pas cu pas pe placă, fiecare registru va fi controlat și de un semnal de la buton (! același monopuls folosit pentru validarea scrierii în PC, RF și memoria de date).

Realizați secționarea propriu-zisă prin folosirea câmpurilor din registrele intermediare ca intrări în etajul (unitatea) care urmează, sau transmiterea lor mai departe spre registrul intermediar următor. Unitatea care „urmează” nu e neapărat cea situată la dreapta registrului, ci următoarea în fluxul de date/semnale de control. De exemplu, valoarea semnalului de control RegWrite transmisă până în registrul intermediar MEM/WB trebuie legată pe intrarea aferentă din unitatea ID.

Exemplu (se poate folosi și concatenare dacă se dorește): pentru IF/ID se declară semnalul RegIF_ID, de 32 de biți, iar comportamentul se descrie într-un proces unde:

```
pe front crescător de ceas
        RegIF_ID(31..16) <= PC+1
        RegIF_ID(15..0)  <= Instruction
```

Unde PC+1 și Instruction sunt porturile de ieșire corespunzătoare din unitatea IF.

Pozițiile din RegIF_ID corespunzătoare celor două câmpuri se transmit mai departe:

- Poziția corespunzătoare PC+1 merge ca intrare în registrul intermediar următor ID_EX
- Poziția corespunzătoare Instruction este transmisă spre unitatea ID (modificați port map).

Ca alternativă, se pot declara semnale noi (incluzând în numele semnalelor etajele adiacente registrului) pentru fiecare câmp individual dintr-un registru intermediar:

```
pe front crescător de ceas
        PC_IF_ID <= PC + 1;
        Instruction_IF_ID <= Instruction;
```

9.3.4. Temă (hârtie / instrument de scris)

Desenați schema propriei versiuni a procesorului pipeline MIPS 16, reprezentată de calea de date completă și unitatea de control cu semnalele asociate.

9.4. Referințe

[1] Arhitectura Calculatoarelor, note de curs (online) 3 & 4, curs pipeline.
[2] D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
[3] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012.
[4] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02.
[5] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
- Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.

Laboratorul 10

10. Procesorul MIPS, pipeline – versiune pe 16 biți (2)

Modificarea programului pentru evitarea hazardurilor și testarea pe placă

10.1. Obiective

Principalele obiective ale acestui laborator:

- **Înțelegerea principalelor tipuri de hazarduri și a tehnicilor de tratare**
- **Rescrierea programului din memoria ROM pentru a evita hazardurile**
- **Testarea procesorului MIPS 16 pipeline, cu programul modificat prin inserarea de NoOp, fără hazarduri.**

10.2. Transformarea procesorului MIPS 16 cu ciclu unic într-un procesor de tip pipeline – recapitulare din laboratorul 9

Ciclul de execuție a unei instrucțiuni MIPS are următoarele etape / faze, care se regăsesc în versiunea pipeline (Figura 10-1) ca etaje:

- IF – Instruction Fetch
- ID/OF – Instruction Decode / Operand Fetch
- EX – Execute
- MEM – Memory
- WB – Write Back.

Pentru a reduce durata ciclului de ceas, soluția aleasă a fost secționarea căii critice cu elemente de stocare sincrone (de tip registru). În versiunea pipeline se pot executa până la 5 instrucțiuni consecutive din program, fiecare aflându-se într-una din cele 5 faze de execuție, în funcție de ordinea lor în program.

Față de versiunea ciclu unic, execuția unei instrucțiuni în pipeline poate dura până la 5 perioade de ceas. Acest aspect duce la mai multe situații când execuția programului, așa cum a fost scris pentru versiunea ciclu unic, nu ar fi corectă.

Exceptând instrucțiunile de salt, fiecare instrucțiune intră în pipeline și, după 5 perioade de ceas, își termină execuția. Dacă instrucțiunile imediat următoare unei instrucțiuni I care actualizează un registru ar încerca să folosească rezultatul produs de I, execuția nu ar fi corectă. Acesta este doar un tip de hazard din trei posibile, subiect care se va discuta mai concret în următoarea parte a laboratorului.

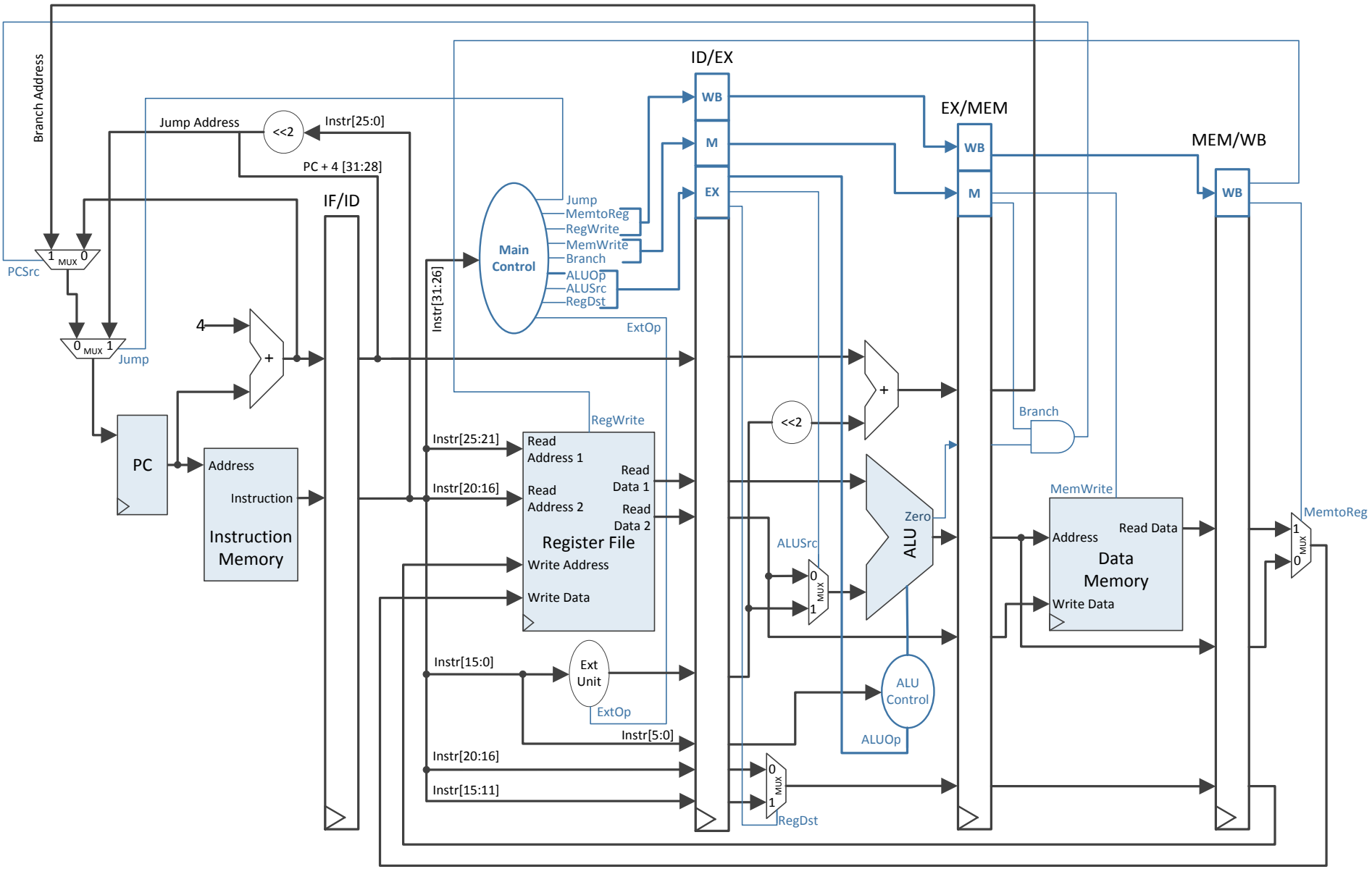


Figura 10-1: Procesorul MIPS 32 pipeline [2], obținut prin secționarea căii de date MIPS 32 cu ciclu unic

10.3. Hazarduri în procesorul MIPS pipeline

Hazardurile sunt situații în care o instrucțiune următoare nu poate fi executată corect în următoarea perioadă de ceas. Există trei tipuri de hazard, în funcție de cauza lor:

1. **Hazard structural (dependență de resurse)**
   - Două instrucțiuni încearcă să folosească simultan o resursă, în scopuri diferite → constrângeri de resurse.
2. **Hazard de date (dependență de date)**
   - Încercarea de a folosi valori înainte să fie actualizate de instrucțiunile anterioare din program
   - Pentru o instrucțiune care a ajuns în etajul ID, operanzii sursă sunt în curs de calculare în etajele următoare de pipeline (de către instrucțiuni anterioare).
3. **Hazard de control/comandă (dependență de condiții, control)**
   - Decizia pentru un salt condiționat nu se știe înainte de evaluarea condiției saltului și calcularea adresei de ramificare pentru PC
   - Apare la trecerea prin pipeline a instrucțiunilor care influențează PC (ramificări, salturi).

Aceste hazarduri au fost prezentate pe larg în cursul de pipeline, sau în [2], a cărui studiere este obligatorie în prealabil.

În continuare, se discută cele 3 tipuri de hazarduri pe exemple concrete de cod. Soluțiile optime, hardware, care au fost discutate la curs, se bazează pe tehnici de înaintare sau așteptare/anulare. Pentru varianta MIPS 16 pipeline, din motive de timp, în acest laborator se va implementa doar soluția software, modificând programul astfel încât să nu mai existe hazarduri.

Modificarea programului se va face inserând instrucțiuni de tip NoOp (NoOperation) între instrucțiunile unde apare hazard! Practic, e varianta software a tehnicilor de așteptare.

Instrucțiunea NoOp este una dintre instrucțiunile pe care le aveți deja, care, în urma execuției, nu trebuie să schimbe nimic în elementele de memorare din procesor – în blocul de registre și memoria de date, iar PC se incrementează normal (ex. de NoOp pentru MIPS: sll $0, $0, 0, sau add $0,$0, $0 etc.).

10.3.1. Hazard structural

Hazardul structural apare atunci când, pe același ciclu de ceas, instrucțiuni aflate etaje diferite din pipeline încearcă să folosească aceeași componentă din procesor.

De interes pentru implementarea de la laborator este hazardul structural care apare la utilizarea blocului de registre, pentru instrucțiuni situate la distanță de 3. În exemplul următor, presupunem că instr1 și instr2 nu au hazard cu restul instrucțiunilor.

Hazard **structural la RF**
add **$1**, $1, $2
instr1
instr2
add $3, **$1**, $4

Diagrama pipeline (pe fiecare pe ciclu de ceas indicăm în ce etaj se află fiecare instrucțiune):

Instr\Clk	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	…
add **$1**, $1, $2	IF	ID	EX	MEM	**WB**				
instr1		IF	ID	EX	MEM	WB			
instr2			IF	ID	EX	MEM	WB		
add $3, **$1**, $4				IF	**ID**	EX	MEM	WB	

Pe durata ciclului de ceas CC5, valoarea nouă a lui $1, generată de prima instrucțiune, se află în etajul WB, nefiind scrisă încă în RF. Astfel, în etajul ID, a patra instrucțiune va citi valoarea veche a lui $1 din blocul de registre, iar la tranziția CC5->CC6 această valoare veche (incorectă din punctul de vedere al programului care se execută) se va propaga în etajul EX.

Există două soluții posibile:

1. **Recomandat**: Se modifică blocul de registre RF astfel încât scrierea să se facă în mijlocul perioadei de ceas (se testează frontul descrescător clk=0 & clk'event sau falling_edge(clk)). În acest caz, citirea din RF fiind asincronă, în a doua jumătate a lui CC5 apare în ID noua valoare (corectă) a lui $1, valoare care este propagată mai departe în etajul EX la tranziția CC5->CC6.
2. (*the lazy approach*) Se introduce o instrucțiune NoOp:

Fără hazard
add **$1**, $1, $2
instr1
instr2
NoOp
add $3, **$1**, $4

Atenție!: În explicarea hazardurilor următoare se consideră că ați ales soluția 1. Dacă ați mers pe varianta 2, țineți cont de diferențe și, unde este necesar, introduceți câte un NoOp suplimentar, în plus față de cele inserate.

10.3.2. Hazard de date

Hazardul de date (tip *Read After Write*, sau *Load data hazard*) apare la acele instrucțiuni care folosesc ca operanzi sursă valori care sunt în curs de calculare de către instrucțiuni anterioare, nefiind actualizate încă.

(!)Pentru a lămuri exact unde apar aceste hazarduri, este utilă diagrama pipeline, precum și o bună stăpânire teoretică a execuției instrucțiunilor în pipeline (cum trec prin fiecare etaj, când se citesc operanzii, când se scrie destinația, vezi curs pentru detalii suplimentare).

Exemplul următor conține majoritatea hazardurilor de date pe care ar trebui să le întâlniți în programul propriu, din procesorul vostru MIPS 16 pipeline. Atenție, spre deosebire de programul vostru, exemplul de mai jos nu are un scop anume!

Instr. Nr.	Program
1	add $1, $2, $3
2	add $3, $1, $2
3	add $4, $1, $2
4	add $5, $3, $2
5	lw $3, 5($5)
6	add $4, $5, $3
7	sw $3, 6($5)
8	beq $3, $4, -6

Procesul de identificare a hazardurilor începe de la primele instrucțiuni, se rezolvă cu inserare de NoOp-uri și se continuă până la ultima instrucțiune. Dacă există îndoieli, diagrama de pipeline (pe foaie se pot folosi săgeți între etaje) poate ajuta la identificarea hazardurilor. Exemplu:

Instr\Clk	CC1	CC2	CC3	CC4	CC5	CC6	CC7	CC8	CC9
add $1, $2, $3	IF	ID	EX	MEM	**WB($1)**				
add $3, $1, $2		IF	**ID($1)**	EX	MEM	**WB($3)**			
add $4, $1, $2			IF	**ID($1)**	EX	MEM	WB		
add $5, $3, $2				IF	**ID($3)**	EX	MEM	**WB($5)**	
lw $3, 5($5)					IF	**ID($5)**	EX	MEM	**WB($3)**

Instr\Clk	CC4	CC5	CC6	CC7	CC8	CC9	CC10	CC11	…
add $5, $3, $2	IF	**ID($3)**	EX	MEM	**WB($5)**				
lw $3, 5($5)		IF	**ID($5)**	EX	MEM	**WB($3)**			
add $4, $5, $3			IF	**ID($3, $5)**	EX	MEM	**WB($4)**		
sw $3, 6($5)				IF	**ID($3)**	EX	MEM	WB	
beq $3, $4, -6					IF	**ID($4)**	EX	MEM	WB

Hazardurile se rezolvă pe rând, de la început, cu primul întâlnit. Rezolvarea unui hazard dintre 2 instrucțiuni consecutive rezolvă implicit eventuale hazarduri între prima instrucțiune și cea de la distanță +2. Exemplu: între instrucțiunea 1 și 2, și 1 și 3 există hazard de tip RAW (după **$1**). Se rezolvă mai întâi hazardul dintre 1 și 2. Conform diagramei, instrucțiunea 2 ar trebui decalată (întârziată) cu 2 cicluri (să aibă faza ID pe ciclul CC5) => 2 NoOp. În acest moment, toate instrucțiunile următoare se vor decala tot cu 2 cicluri, deci între 1 și 3 hazardul dispare de la sine.

Între instrucțiunea 2 și 4 există hazard RAW după **$3**, care se poate rezolva, conform diagramei pipeline, prin decalarea cu 1 ciclu. Rezultă un NoOp inserabil fie după instrucțiunea 2, fie înainte de 4.

Pe aceeași logică se continuă cu restul hazardurilor de date…

Rezultă următoarea rescriere a programului, fără hazarduri:

Instr. Nr.	Program
1	add $1, $2, $3
2	NoOp
3	NoOp
4	add $3, $1, $2
5	NoOp
6	add $4, $1, $2
7	add $5, $3, $2
8	NoOp
9	NoOp
10	lw $3, 5($5)
11	NoOp
12	NoOp
13	add $4, $5, $3
14	NoOp
15	sw $3, 6($5)
16	beq $3, $4, -6

10.3.3. Hazard de control

Hazardurile de control apar la instrucțiuni de salt, unde instrucțiunile care urmează după instrucțiunea de salt intră implicit în execuție.

În cazul instrucțiunilor de salt condiționat (beq, bne etc), în forma de bază a procesorului MIPS pipeline, vor intra implicit în execuție următoarele 3 instrucțiuni. Modalitatea simplă, dar nu cea mai eficientă (vezi curs pipeline), este să se insereze 3 NoOp în program, după instrucțiunea de salt condiționat.

În cazul instrucțiunilor de salt necondiționat j, conform descrierii pipeline din Figura 10-1, aceste instrucțiuni se finalizează (execută saltul) când sunt în etajul ID. Rezultă că intră în execuție doar instrucțiunea imediat următoare. Soluția simplă este să se insereze un NoOp după instrucțiunea j. O soluție mai eficientă (!) este să se mute (în loc de NoOp) instrucțiunea care era înainte de j, care practic oricum trebuia să se execute. Mutarea se poate face cu condiția ca această instrucțiune să nu fie tot de salt (j, beq etc.) și să nu apară hazard de date cu instrucțiunile aflate la adresa unde se sare.

10.4. Activități practice

Citiți fiecare activitate în întregime, înainte să o începeți!

Resurse necesare (de avut la începerea laboratorului!):

- Proiect Xilinx cu *test_env*, care să conțină implementarea proprie a procesorului MIPS 16 pipeline, completă și corectă.

10.4.1. Analiza programului și eliminarea hazardurilor (hârtie / instrument de scris)

Pe baza exemplului prezentat anterior, în 10.3, identificați hazardurile din programul vostru. Inserați instrucțiunile NoOp acolo unde este necesar. Desenați diagrama pipeline pentru cel puțin 5 instrucțiuni succesive din programul vostru (sau tot, dacă nu identificați ușor hazardurile).

Atenție, în urma introducerii de NoOp-uri va trebui să ajustați adresele din instrucțiunile de salt din program.

Actualizați programul corectat (cod mașină) în memoria ROM de instrucțiuni.

10.4.2. Testarea și evaluarea procesorului MIPS 16 pipeline

Testați procesorul MIPS pipeline pe placa de dezvoltare.

Există două variante, ca nivel de detaliu:

1. Dacă transformarea în pipeline a fost corectă, fără greșeli la maparea porturilor etc., atunci este suficient să urmăriți rezultatele finale ale programului vostru (trebuie să fie identice cu rezultatele din varianta cu ciclu unic).
2. Dacă rezultatele nu coincid, atunci este cazul să faceți o trasare pas cu pas a programului.

Folosiți același mecanism de afișare a câmpurilor intermediare din procesor ca la varianta cu ciclu unic (mux-ul pe switch-uri pentru afișare pe SSD). Țineți cont că, spre deosebire de MIPS 16 cu ciclu unic, acum veți avea 5 instrucțiuni consecutive în execuție (unele NoOp), deci câmpurile vizualizate pe SSD, respectiv semnalele de control de pe leduri, nu vor fi toate de la aceeași instrucțiune. Dacă este nevoie afișați alte câmpuri pe SSD, pentru depanare.

În funcție de câmpurile din procesor (semnale de control, Jump Address, Branch Address, Read Data 1,2, ALUResult etc) care nu au valorile corecte la instrucțiunile urmărite, reveniți la descrierea VHDL pentru identifica și corecta erorile.

10.4.3. Optimizarea procesorului MIPS 16 pipeline (opțional)

Dacă, în urma testării, s-a confirmat corectitudinea versiunii pipeline a procesorului, acesta poate fi extins cu:

a. Unitatea de înaintare (forwarding)
b. Unitatea de detecție a hazardurilor
   - Mutarea evaluării condiției de la beq în etajul ID
   - Tehnici de așteptare (stall) pentru a rezolva hazardul de date de tip RAW, care apare la lw, și hazardul de control de la instrucțiunile beq și j.

Detalii despre aceste posibilități de extindere se găsesc în cursul de pipeline sau în literatura de specialitate [2]. După integrarea acestor optimizări, procesorul

pipeline MIPS 16 va fi complet și va rezolva hazardurile în hardware, fără a mai fi necesară inserarea de NoOp în programe.

10.5. Referințe

[1] Arhitectura Calculatoarelor, note de curs (online) 3 & 4, curs pipeline.
[2] D. A. Patterson, J. L. Hennessy, “Computer Organization and Design: The Hardware/Software Interface”, 5th edition, ed. Morgan–Kaufmann, 2013.
[3] MIPS® Architecture for Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012.
[4] MIPS® Architecture for Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02.
[5] MIPS32® Architecture for Programmers Volume IV-a: The MIPS16e™ Application-Specific Extension to the MIPS32™ Architecture, Revision 2.62.
- Chapter 3: The MIPS16e™ Application-Specific Extension to the MIPS32® Architecture.

Laboratorul 11

11. Automate cu stări finite / Comunicație serială (1)

Transmisia serială

11.1. Obiective

Studiul, proiectarea, implementarea și testarea pentru:

- **Automate (mașini) cu stări finite**
- **Comunicația serială - Transmisia serială.**

11.2. Fundamente teoretice

11.2.1. Automate cu stări finite

Automatele sau mașinile cu stări finite (Finite State Machine – FSM) se pot folosi pentru a descrie o unitate de control. Un FSM este un model abstract pentru descrierea comportamentului. Un FSM constă dintr-un set finit de stări (la un moment dat se poate afla strict într-una dintre aceste stări), tranzițiile între stări și acțiunile asociate cu fiecare stare.

În mod implicit, Xilinx încearcă să recunoască FSM-urile scrise în codul VHDL. În acest scop, în anexa 6 (adaptat după [1]) se prezintă cele 3 tipuri de descriere pentru un FSM, care sunt recunoscute automat de Xilinx, la sintetizare: cu 1, 2 sau 3 procese.

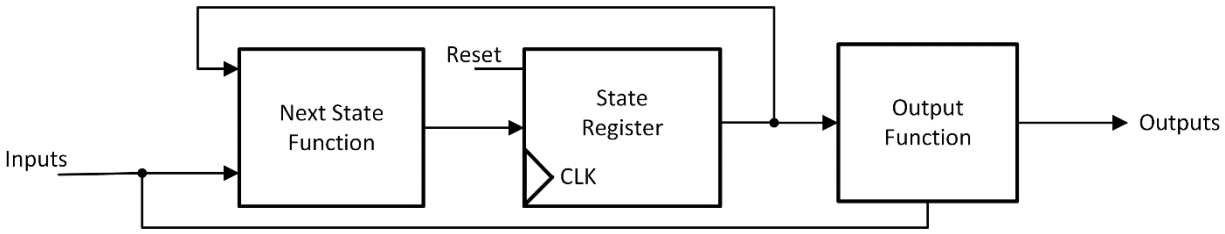


Figura 11-1: Reprezentarea FSM (include Mealy și Moore) conform XST User Guide [1]

Practic, stările FSM-ului sunt declarate în mod generic, cu numele lor sub formă de enumerare (exemplu cu 4 stări):

```
…
type state_type is (nume_stare1,nume_stare2,nume_stare3,nume_stare4);
signal state : state_type ;
…
```

Numele stării va fi folosit pentru a referi starea în descrierea VHDL, în loc de o anumită codificare numerică. Astfel, descrierea VHDL a FSM-ului este la un nivel mai înalt, simbolic, ușor de urmărit. În mod automat, Xilinx Vivado este

capabil să aplice diferite tehnici de codificare a stărilor, utilizatorul putând alege tehnica din **Synthesize Settings – fsm_extraction**: Auto (implicit), One-hot, Gray, Johnson, Sequential. O descriere detaliată (pro/contra) a acestor moduri se găsește în XST User Guide [1].

11.2.2. Comunicație serială - UART

Comunicația serială are ca principiu de bază transmisia sau recepția datelor bit cu bit, un singur bit fiind transmis/recepționat la un moment dat. Deoarece în sistemele de calcul datele sunt reprezentate pe octet (sau multiplu), se folosește un port serial cu rolul de a converti fiecare octet într-un șir de biți (0 sau 1) și viceversa. Portul serial conține un circuit electronic numit Universal Asynchronous Receiver/Transmitter (UART) care face conversia efectivă.

La transmisia unui octet, UART transmite mai întâi bitul de START, urmat de biți de date (în mod uzual 8 biți, dar e posibil și cu 5, 6 sau 7 biți), urmați de bitul de STOP. Protocolul e repetat pentru fiecare octet din secvența care trebuie trimisă.

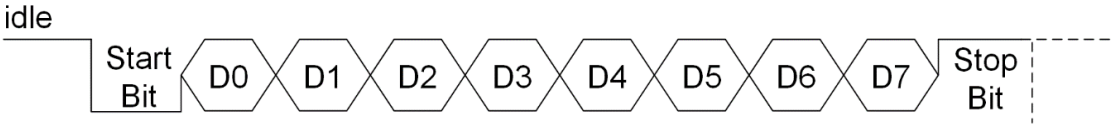


Figura 11-2: Diagrama de timp pentru transmisia serială, exemplu pe 8 biți

Transmisia serială nu implică existența unui semnal de ceas comun la transmițător și receptor. În schimb, se folosește o rată de eșantionare, generată independent la sursă și destinație, numită *baud rate* (**număr de biți transmiși pe secundă**). Valorile uzuale pentru baud rate sunt 2400, 4800, 9600 și 19200. Practic, un bit este valid pe linia de transmisie pentru un interval dat de timp, egal cu inversul baud rate.

Câteva aspecte importante:

- Dacă nu este începută transmisia, linia serială este în starea IDLE, pe 1
- Bitul START este întotdeauna 0, iar biții de date sunt transmiși în ordinea inversă a semnificației, primul este LSB iar ultimul este MSB
- Bitul STOP este întotdeauna 1
- Durata bitului STOP poate avea mai multe valori: 1, 1.5 sau 2 perioade de bit (1/baud rate)
- Pe lângă biți de START și STOP se poate folosi un bit adițional de paritate, împreună cu datele, pentru a detecta eventuale erori de transmisie.

Datele transmise prin comunicație serială sunt codificate folosind coduri ASCII (vezi anexa 7). De exemplu, dacă se dorește trimiterea caracterului ‘C’, se va transmite codul ASCII pe 8 biți al caracterului, în acest caz 01000011 (43h). Dacă se folosesc 8 biți de date, 1 bit de STOP, fără bit de paritate, atunci șirul de biți care trebuie transmis pentru caracterul ‘C’ este de forma (START) (DATA) (STOP):

LSB **(0 1 1 0 0 0 0 1 0 1)** MSB.

Pe acest exemplu, pentru un caracter reprezentat pe 8 biți, trebuie transmiși în total 10 biți. Astfel, se poate calcula numărul de caractere care se pot trimite pe secundă. La un baud rate de 9600, rezultă 9600/10 = 960 caractere pe secundă.

În figura următoare se prezintă încă un exemplu, și anume procesul de transmisie pentru litera „A”.

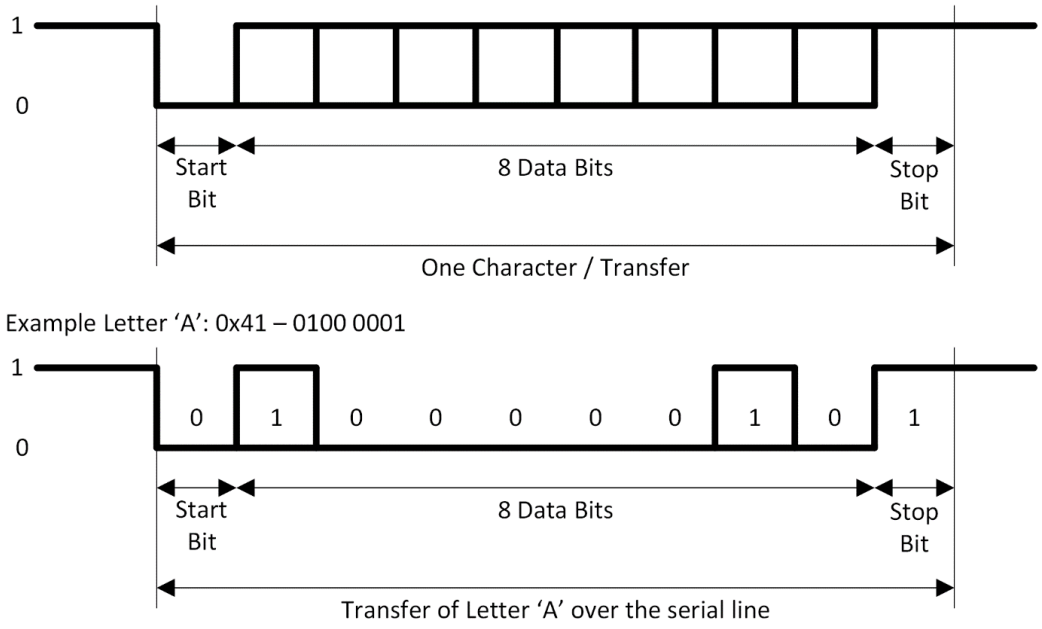


Figura 11-3: Exemplu de transmisie serială (litera A, 8 biți de date, fără paritate)

În cazul recepției, trebuie citit (eșantionat) semnalul de pe linia serială la o rată mai mare decât rata de transmisie. Dacă s-ar citi la rata de transmisie, din cauza nesincronizării perfecte (baud rate este generat independent la sursa, respectiv destinație), există riscul unor decalaje care pot cauza citirea dublă a aceluiași bit, sărirea peste un bit, ratarea bitului de start etc. Astfel, este importantă detecția mijlocului bitului de start, biții de date fiind apoi citiți aproximativ în jurul mijlocului intervalului de bit. Se elimină practic riscul de decalaj, deoarece la fiecare bit de start se reîncepe eșantionarea din mijlocul intervalului de bit. Cea mai uzuală schemă de supraeșantionare este folosirea unei rate de 16 ori mai mare decât baud rate. Concret, fiecare bit care vine pe linia serială este eșantionat (citit) de 16 ori, însă doar una dintre citiri este salvată (cea din mijloc). Problema recepției se va relua în laboratorul viitor.

Orice circuit UART conține un registru de deplasare (shift register), acesta fiind folosit în mod clasic pentru conversia datelor din forma paralelă în forma serială, și invers. În acest laborator (și în următorul), comportamentul acestui registru se va descrie mai simplu (nu independent), ca parte a automatului cu stări finite.

11.3. Activități practice

11.3.1. USB-UART

Plăcile de dezvoltare Basys 3 (Artix 7) sunt dotate cu un circuit USB UART (marca FTDI) care comunică prin același cablu cu care se face programarea plăcii. Consultați manualul de referință pentru Basys 3 (secțiunea cu USB UART).

Descărcați și deschideți aplicația HTERM care vă oferă un terminal pentru comunicație serială între calculatorul pe care lucrați și placa de dezvoltare.

Trebuie să definiți două porturi noi (std_logic) în entitatea test_env pentru comunicație serială (liniile de transmisie și recepție): RX (in) și TX (out).

Atribuiți pini pentru aceste două porturi în fișierul XDC (vezi primul laborator, la metodologia de atribuire a unui pin nou). Folosiți manualul de referință (primul laborator) al plăcii de dezvoltare pentru a determina cei doi pini ai UART FTDI de pe placă.

Atenție! Portul TX din test_env (placa de dezvoltare) trebuie să corespundă fizic cu RX de pe modulul FTDI, iar portul RX din test_env trebuie să corespundă cu pinul TX al modulului FTDI.

11.3.2. FSM pentru transmisie serială

Mai întâi, trebuie să descrieți în entitatea test_env un **generator pentru semnalul de baud rate**, pentru o valoare de 9600 biți pe secundă. Folosiți un numărător, la frecvența plăcii, pentru a genera semnalul BAUD_ENable (’0’ valoare normală, este pus pe ‘1’ la intervalul de bit, interval care se măsoară în cicluri de ceas, după care se reîncepe numărarea). Atenție, durata cât stă BAUD_ENable ‘1’ pe este egală cu perioada de ceas.

Pentru generarea baud rate (Basys 3 are 100 Mhz):

- La ceas de 25 MHz, se generează ‘1’ la fiecare 25Mhz/9600=2604 tacți
- La ceas de 50 MHz, se generează ‘1’, la fiecare 5208 tacți
- La ceas de 100 MHz, se generează ‘1’, la fiecare 10416 tacți.

Definiți o nouă entitate pentru FSM-ul de transmisie, cu porturile conform figurii următoare.

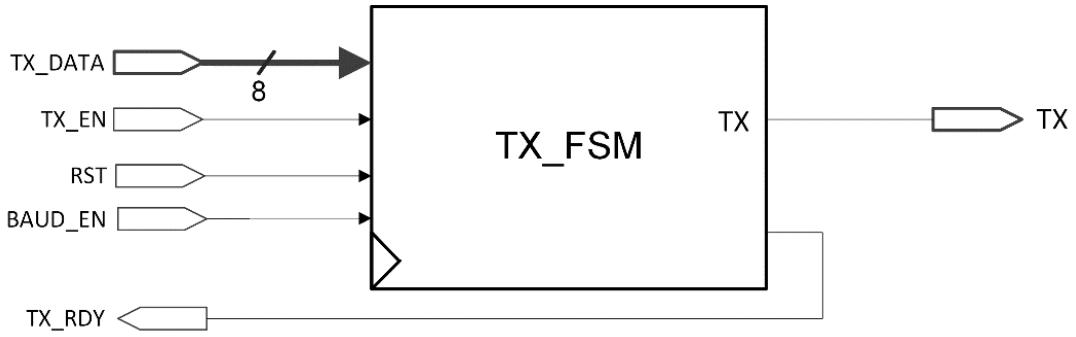


Figura 11-4: Entitatea cu FSM-ul de transmisie TX_FSM

Diagrama în detaliu a TX_FSM este prezentată în Figura 11-5, implementând protocolul de transmisie serială pentru caractere de 8 biți. O tranziție între stări poate avea loc pe frontul crescător de ceas doar dacă BAUD_ENable este '1'. Astfel, se asigură că un bit rămâne pe linia de transmisie pentru intervalul dat de baud rate. Semnalul BIT_CNT are o funcționalitate de numărător în interiorul FSM-ului, el reprezentând poziția bitului curent de transmis din TX_DATA. El trebuie incrementat în starea *bit* și trebuie resetat după fiecare caracter transmis (se poate reseta în starea *idle* sau în toate stările exceptând *bit*).

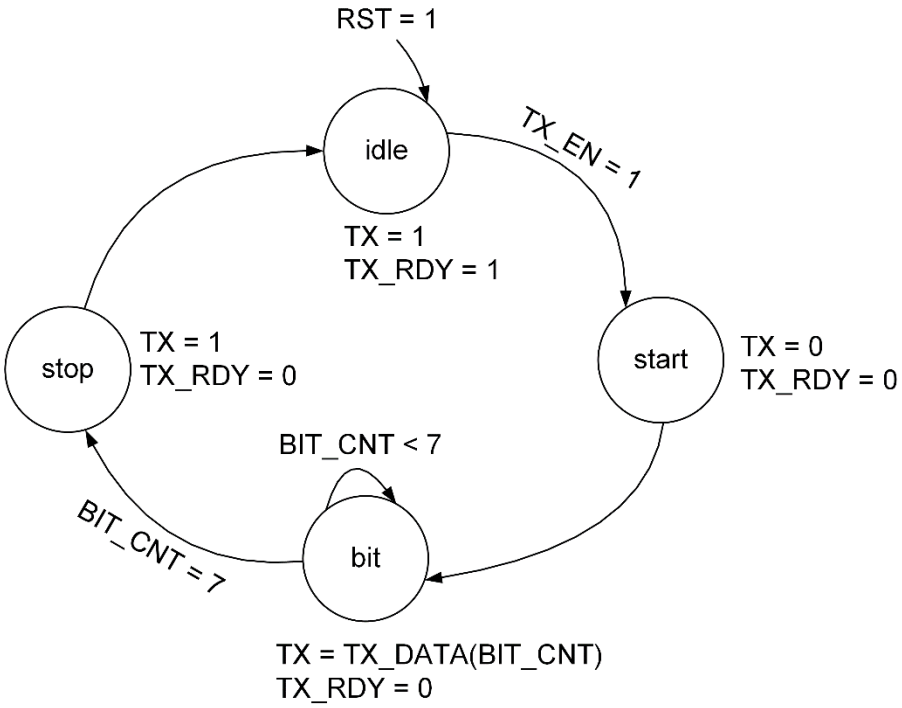


Figura 11-5: Diagrama TX_FSM

Descrieți în VHDL comportamentul entității TX_FSM, din test_env. Folosiți o descriere FSM cu 2 sau cu 3 procese (vezi anexa 6)! Descrierea cu un proces necesită atenție sporită la valorile semnalelor de ieșire, care vor apărea întârziate cu o perioadă de ceas dacă nu sunt generate cu o stare mai devreme.

Pentru a testa transmisia serială între placa de dezvoltare și calculator, conectați intrarea TX_DATA a FSM-ului la cele 8 switch-uri ale plăcii, pe care veți alege diferite coduri ASCII.

Trebuie generat semnalul TX_EN care să pornească transmisia. Acesta trebuie controlat de la un MPG, dar (!) nu direct, deoarece el trebuie să rămână pe '1' până apare și BAUD_ENable='1' (altfel nu se va ieși din starea *idle* – este improbabil ca pulsul generat de MPG să fie pe același ciclu de ceas cu BAUD_ENable). Se va descrie un bistabil D, pe front crescător, care are *set* de la MPG, și *reset* de la BAUD_ENable. Ieșirea din acest bistabil este semnalul dorit TX_EN. Pe semnalul RST al FSM-ului legați '0' sau un alt MPG.

Acum se va testa comunicația între placa de dezvoltare și calculator. Parametrii pentru comunicația serială sunt: 1 bit de START, 1 bit de STOP, 8 biți de date, fără bit de paritate, baud rate de 9600. Asigurați-vă că aceste setări apar în aplicația de pe calculator HTERM / hyper terminal, și alegeți corect portul serial în aplicație (COM1, 2, 3 etc). Pentru a identifica exact care port serial este cel cu

placa, vizualizați în aplicație lista de porturi seriale înainte și după legarea modulului prin cablul USB la calculator.

11.3.3. Comunicație I/O cu procesorul MIPS

Conectați TX_FSM cu procesorul MIPS pe care l-ați implementat (puteți folosi MIPS cu ciclu unic sau versiunea pipeline, o versiune funcțională).

Trebuie să trimiteți 16 biți de date din procesorul vostru către calculatorul pe care lucrați. Acești biți ar trebui să reprezinte rezultatul final al programului vostru (sau unul dintre rezultate). În funcție de unde este stocat acest rezultat (în blocul de registre sau în memoria de date), adăugați o instrucțiune nouă la finalul programului vostru care să acceseze acea locație. Implicit, valoarea va apărea în calea de date, și o puteți transmite prin UART.

Exemplu particular

La terminarea programului, rezultatul este în $7, iar PC este 0x0020. Adăugați o nouă instrucțiune la program: addi $7, $7, 0, la adresa 0x0021 în memoria ROM de instrucțiuni. Definiți un registru de 16 biți care este scris cu valoarea din semnalul RD1 (sau ALURes!), scrierea fiind validată de valoarea lui PC (egală cu adresa instrucțiunii adăugate, atenție dacă testați PC+1…).

Definiți și descrieți în VHDL metodologia de transfer a celor 4 caractere conținute în registru (pe calculator trebuie să apară codificarea alfa numerică, la fel ca pe SSD). Folosiți un decodificator/ROM pentru a genera reprezentarea ASCII (8 biți) pentru o cifra hexa (4 biți). Trebuie să implementați un mecanism de parcurgere a celor 4 cifre din registru, una câte una, si fiecare cifră trebuie transmisă prin intermediul TX_FSM. Indiciu: amintiți-vă de mecanismul de parcurgere a cifrelor la SSD (sau implementați la registrul de 16 biți un mecanism de deplasare cu 4 poziții), și folosiți TX_RDY pentru a trece la următoarea cifră (TX_RDY arată că TX_FSM este în *idle*, deci poate începe o nouă transmisie).

11.4. Referințe

[1] XST User Guide.
[2] Digilent Basys 3 Board – Reference Manual.
[3] http://www.asciitable.com/.
[4] http://www.der-hammer.info/terminal/.

Laboratorul 12

12. Automate cu stări finite / Comunicație serială (2)

Recepția serială

12.1. Obiective

Studiul, proiectarea, implementarea și testarea pentru:

- **Automate (mașini) cu stări finite**
- **Comunicația serială - Recepția serială.**

12.2. Fundamente teoretice

12.2.1. Mecanismul de (supra)eșantionare pentru recepția serială UART

La transmiterea unui octet, UART transmite mai întâi bitul de START, urmat de biți de date (în mod uzual 8 biți, dar e posibil și cu 5, 6 sau 7 biți), urmați de bitul de STOP. Protocolul e repetat pentru fiecare octet din secvența care trebuie trimisă.

Transmisia serială nu implică existența unui semnal de ceas comun la transmițător și receptor. În schimb se folosește o rată de eșantionare, generată independent la sursă și la destinație, numită *baud rate* (**număr de biți transmiși pe secundă**). Valorile uzuale pentru baud rate sunt 2400, 4800, 9600 și 19200. Practic, un bit este valid pe linia de transmisie pentru un interval dat de timp, egal cu inversul baud rate. Mai multe detalii legate de transmisie se regăsesc în laboratorul anterior.

În cazul recepției, trebuie citit (eșantionat) semnalul de pe linia serială, și extrasă, bit cu bit, valoarea transmisă de la sursă.

La prima vedere, rata de eșantionare la recepție ar trebui să coincidă cu rata de eșantionare (baud rate) cu care s-a transmis șirul de biți. Greșit! Dacă s-ar citi la rata de transmisie, din cauza nesincronizării perfecte (baud rate este generat independent la sursă, respectiv destinație; este comunicație asincronă = fără un semnal de ceas comun), există riscul unor decalaje care pot cauza citirea dublă a aceluiași bit, sărirea peste un bit, ratarea bitului de start etc. Frecvența de apariție pentru aceste erori este proporțională cu diferența dintre ratele de eșantionare la transmițător și receptor. Chiar dacă ar fi diferențe foarte mici, la un număr suficient de eșantionări consecutive (pentru biți consecutivi) eroare va apărea. De exemplu, la o diferență de 0.1% între cele două rate, la 1000 de biți consecutivi eroarea apare o dată. În acest caz, dacă avem 10 biți / caracter, rezultă că un caracter din 100 va fi recepționat greșit.

Problema se rezolvă prin supraeșantionare, adică semnalul de pe linia serială este citit la o rată mai mare decât rata de transmisie. Acest lucru permite detecția mijlocului bitului de start, biții de date fiind apoi citiți aproximativ în jurul mijlocului intervalului de bit (Figura 12-1). Se elimină practic riscul de decalaj, deoarece la fiecare nou caracter se reia eșantionarea din mijlocul bitului de start =

resincronizare. Cea mai uzuală schema de supraeșantionare este folosirea unei rate la recepție de 16 ori mai mare decât rata folosită la transmisie. Concret, fiecare bit care vine pe linia serială este eșantionat (citit) de 16 ori, însă doar una dintre citiri este salvată (cea din mijloc). Întârzierea maximă de detecție a bitului de start este cel mult 1/16 din intervalul de bit.

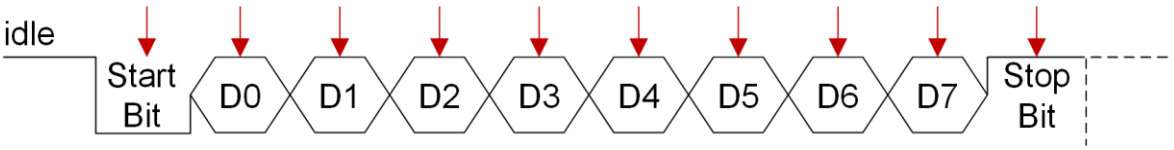


Figura 12-1: Diagrama de timp pentru transmisia serială, exemplu pe 8 biți, săgețile roșii indicând momentul ideal când linia ar trebui citită la receptor

12.3. Activități practice

12.3.1. FSM pentru recepția serială

La fel ca la transmisie, mai întâi trebuie să descrieți în test_env un generator pentru semnalul de eșantionare (baud rate), pentru o valoare de 9600 (biți pe secundă), dar ținând cont de nevoia de supraeșantionare. Diferența față de transmisie este ca acest semnal trebuie să aibă o rată de 16 ori mai mare. Folosiți un numărător pentru a genera semnalul BAUD_ENable (’0’ valoare normală, este pus pe ‘1’ la intervalul de bit, interval care se măsoară în cicluri de ceas, după care se reîncepe numărarea). Durata cât stă BAUD_ENable ‘1’ pe este egală cu perioada de ceas.

Pentru generarea baud rate (Basys 3 – 100 Mhz):

- La ceas de 25 MHz, se generează ‘1’ la fiecare 25Mhz/(9600*16)=163 tacți
- La ceas de 50 MHz, se generează ‘1’, la fiecare 326 tacți
- La ceas de 100 MHz, se generează ‘1’, la fiecare 651 tacți.

Definiți o nouă entitate pentru FSM-ul de recepție, cu porturile conform figurii următoare.

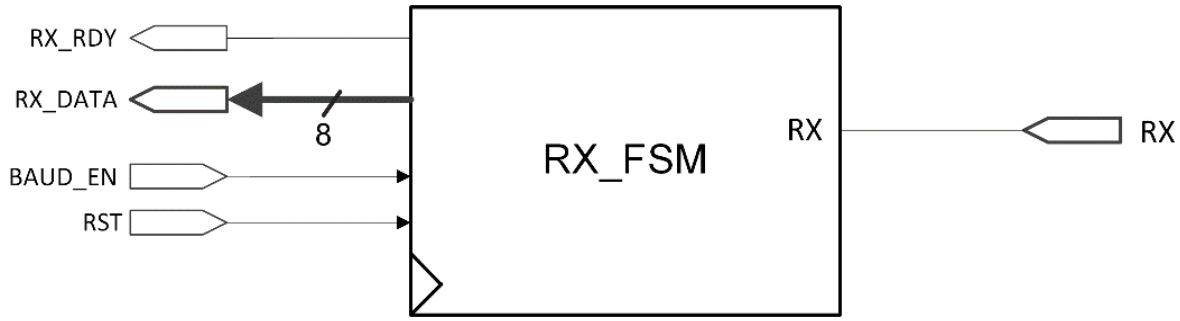


Figura 12-2: Entitatea cu FSM-ul de recepție RX_FSM

Diagrama în detaliu a RX_FSM este prezentată în Figura 12-3. O tranziție între stări poate avea loc pe frontul crescător de ceas doar dacă BAUD_ENable este ‘1’.

Astfel se asigură că eșantionarea liniei seriale (prin acțiunile de verificare a lui RX din stările *idle*, *start* și *bit*) se face la o rată de 16 ori mai mare ca rata de transmisie.

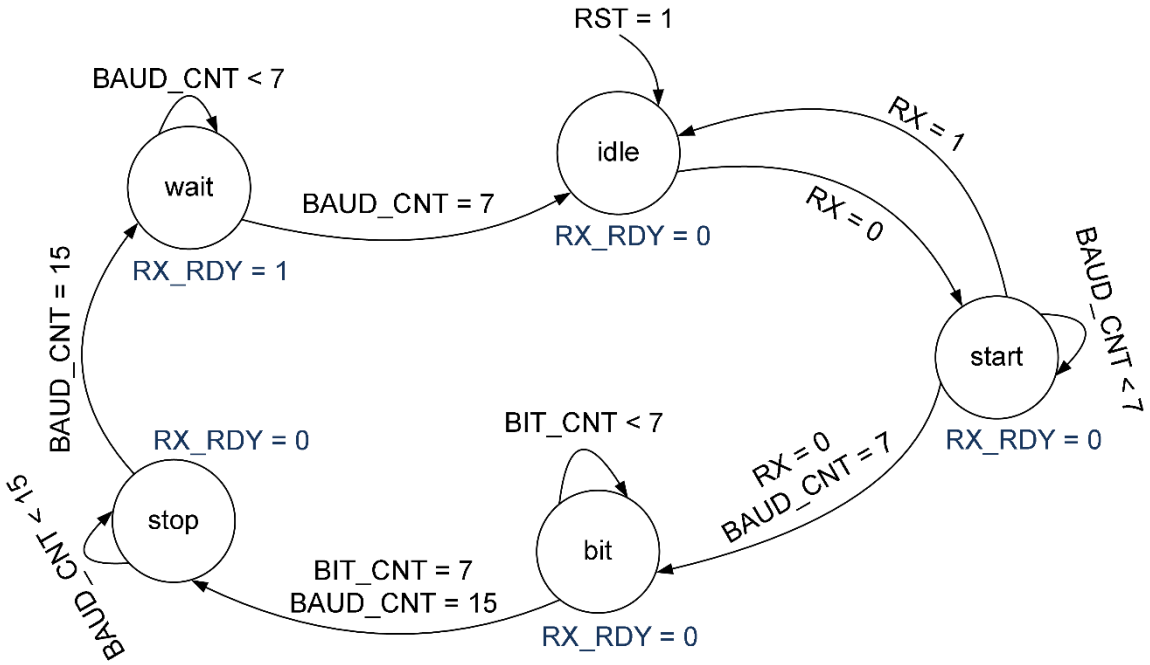


Figura 12-3: Diagrama RX_FSM

Cei 8 biți citiți se vor stoca în RX_DATA (în RX_FSM, pe starea *bit*), care este un semnal pe 8 biți. RX_RDY este setat pe 1 după ce s-a terminat recepția unui caracter (acest semnal este util doar la activitatea 12.3.2).

În RX_FSM sunt necesare două semnale auxiliare cu funcționalitate de numărător: BIT_CNT și BAUD_CNT.

Semnalul BIT_CNT are o funcționalitate la fel ca în TX_FSM, de numărător în interiorul FSM-ului, el reprezentând poziția din RX_DATA unde se stochează bitul curent. El trebuie incrementat în starea *bit*, de fiecare dată când se ajunge la mijlocul bitului curent, și trebuie resetat după fiecare caracter transmis (se poate reseta în starea *idle* sau în toate stările exceptând *bit*).

Semnalul BAUD_CNT contorizează câte activări (valori de 1) are semnalul BAUD_ENable, el fiind utilizat pentru a verifica dacă s-a ajuns în mijlocul intervalului de bit sau la sfârșit de interval.

În mare, funcționarea RX_FSM este următoarea:

1. În starea *idle*, BAUD_CNT este setat pe 0, se trece în *start* dacă pe linia serială apare 0 (RX=0).
2. În starea *start*, BAUD_CNT este incrementat și folosit pentru a eșantiona linia serială (RX). Dacă s-a ajuns la mijlocul bitului de start (BAUD_CNT=7 și RX=0) atunci BAUD_CNT este resetat și se trece la starea *bit*.
3. În starea *bit*, BAUD_CNT este implicit incrementat, până ajunge la valoarea 15 (mijlocul bitului curent). Dacă a ajuns la 15, se salvează bitul curent (RX_DATA[BIT_CNT]=RX), se resetează BAUD_CNT (se reîncepe numărarea până la 15) și se incrementează BIT_CNT. Dacă BIT_CNT=7 și BAUD_CNT=15, atunci sunt resetate și se trece în starea *stop.*

4. În *stop* se așteaptă să treacă încă un interval de bit pe linia serială, interval care acoperă practic jumătate din ultimul bit de date și jumătate din bitul de STOP (amintiți-vă că s-a început de la jumătatea bitului de START, deci în *stop* s-a sărit din *bit* la jumătatea ultimului bit de date).
5. În *wait* se semnalează că s-a recepționat un caracter (RX_RDY=1), se mai așteaptă o jumătate de interval de bit (să se termine bitul de STOP) și se trece în *idle*.

Descrieți în VHDL comportamentul entității RX_FSM, din test_env. Folosiți o descriere FSM cu 2 sau cu 3 procese (vezi anexa 6).

Conectați ieșirea RX_DATA a lui RX_FSM la afișorul SSD. Se va afișa pe SSD (2 cifre) codul ASCII al caracterelor transmise de pe calculator către placa FPGA.

Acum, se va testa comunicația între placa de dezvoltare și calculator. Parametrii pentru comunicația serială sunt: 1 bit de START, 1 bit de STOP, 8 biți de date, fără bit de paritate, baud rate de 9600. Asigurați-vă că aceste setări apar în aplicația de pe calculator HTERM / hyper terminal, și alegeți corect portul serial în aplicație (COM1, 2, 3 etc). Pentru a identifica exact care port serial este cel cu placa, vizualizați în aplicație lista de porturi seriale înainte și după legarea modulului prin cablul USB la calculator.

12.3.2. Comunicare I/O cu procesorul MIPS – opțional

Conectați RX_FSM cu procesorul MIPS pe care l-ați implementat (puteți folosi MIPS cu ciclu unic sau versiunea pipeline, o versiune funcțională).

Trebuie să trimiteți 16 biți de date de pe calculator (HTERM) către procesorul vostru. Acești biți ar trebui să reprezinte unul dintre operanzii de intrare ai programului vostru. În funcție de unde este localizat acest operand (în blocul de registre, în memoria de date), veți defini ce câmp din procesor se va scrie cu datele recepționate pe linia serială.

Țineți cont că pe linia serială se recepționează câte 8 biți, care reprezintă codul ASCII al unei cifre. Pentru a transmite 16 biți către procesor de pe calculator, trebuie trimise 4 cifre hexazecimale, fiecare fiind codificată pe linia serială prin codul ei ASCII. Folosiți un decodificator/ROM pentru a genera cifra de 4 biți asociată cu codul ASCII pe 8 biți primit pe linia serială. Concatenați cele 4 cifre primite prin 4 transferuri pentru a obține data pe 16 biți pe care o veți scrie în câmpul dorit din procesor.

12.4. Referințe

[1] XST User Guide.
[2] Digilent Basys 3 Board – Reference Manual.
[3] http://www.asciitable.com/.
[4] http://www.der-hammer.info/terminal/.

A. Anexa 1 – VIVADO Quick Start Tutorial

VIVADO Quick Start Tutorial, adaptat pentru versiunea 16.4 (în versiunile mai noi pot fi mici diferențe !)

Pornirea mediului VIVADO

Dublu click pe icoana de pe desktop sau se pornește din **Start→Programs→Xilinx Design Tools→Vivado 16.4**

Acces la Help

Pentru a accesa Help-ul:

- Din meniul principal lansați **Help/Documentation and Tutorials**. Veți accesa informații diverse despre crearea și mentenanța întregului ciclu de dezvoltare în Vivado.

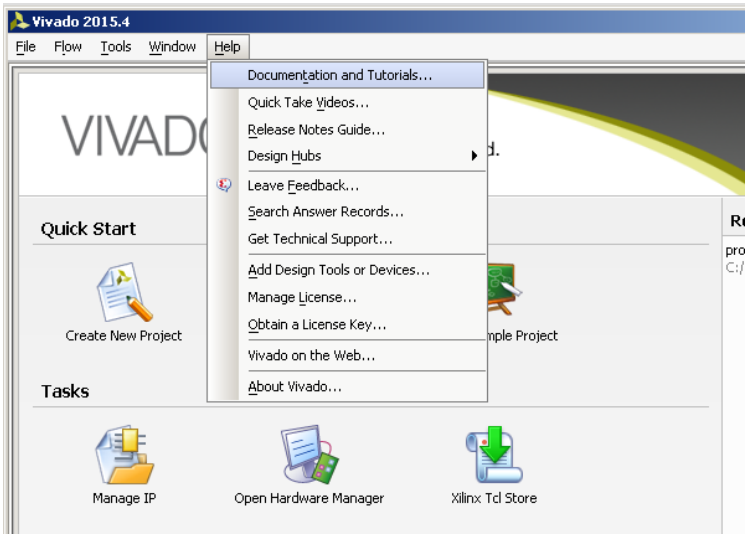


Figura A-1: VIVADO Help

Crearea unui nou proiect

Creați un nou proiect Vivado, care va fi dedicat pentru dispozitivul FPGA prezent pe placa de dezvoltare Basys 3, FPGA Artix 7.

Pentru a crea noul proiect:

1. Selectați din meniu **File → New Project... .** Se va deschide *New Project Wizard*. Click **Next**.
2. Introduceți **test_env** în câmpul *Name*.
3. Introduceți sau accesați o locație (cale de director, amintiți-vă regulile!) pentru proiectul nou. Un subdirector test_env se va crea automat (*Create project subdirectory* trebuie să fie bifat). Click **Next**.
4. Ați ajuns la alegerea tipului proiectului. Selectați *RTL Project*. Bifați *Do not specify sources…* le veți crea mai târziu. Click **Next**.

5. Se aleg proprietățile plăcii. Completați proprietățile conform listei de mai jos:
   - Product Category: **All**
   - Family: **Artix-7**
   - Package: **cpg236**
   - Speed Grade: **-1**
   - Temperature grade: **în 2016.4 nu contează**
   - În tabel alegeți varianta **xc7a35tcpg236-1**.
   - **Next**…

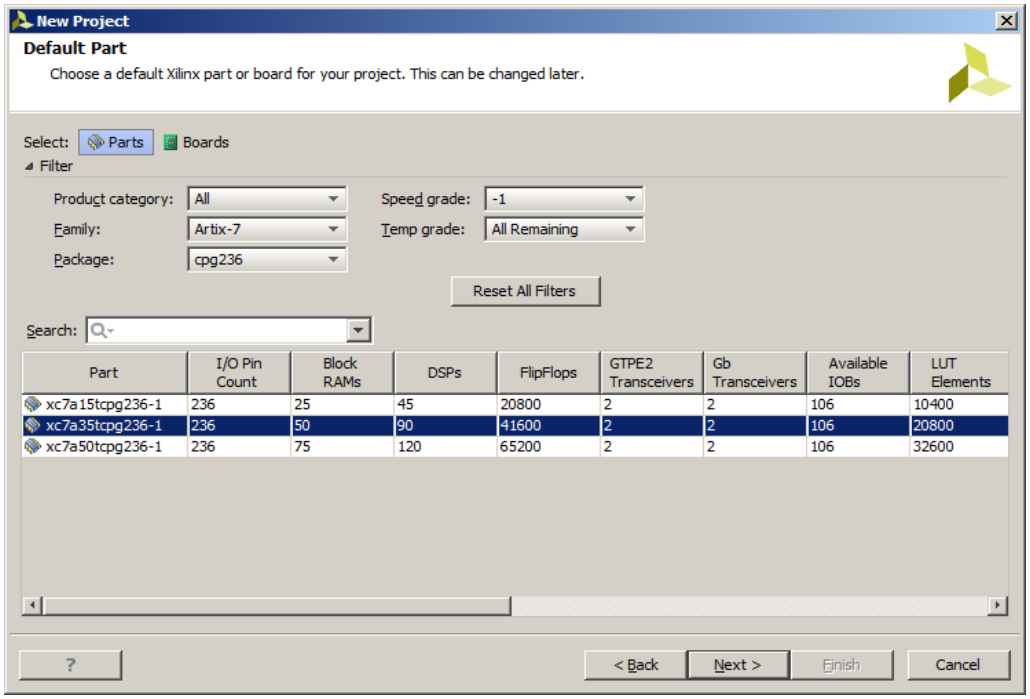


Figura A-2: Alegerea proprietăților plăcii

6. Apăsați **Finish**, se va deschide noul proiect, gol deocamdată.

Crearea unui fișier sursă VHDL

1. Click în meniu pe **File\Add Sources**. Sau: în panoul **Flow Navigator** (de obicei este situat în stânga) la **Project Manager / Add Sources**.
2. Selectați *Add or create design sources*, click **Next**.
3. Apăsați *Create file*. Pe dialogul apărut selectați la *File type*: VHDL, introduceți numele fișierului *test_env* (! Nu e obligatoriu să fie același nume cu proiectul), lăsați *File location* neschimbat. Acest nume (*test_env*) îl va avea și entitatea creată automat în acest fișier. Apăsați **OK**.

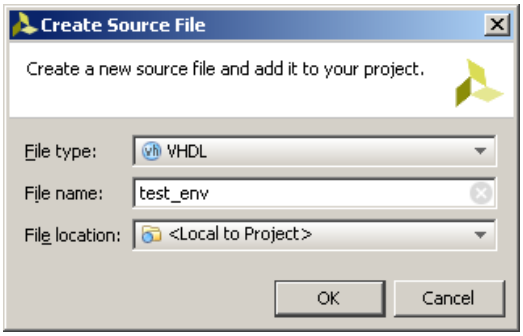


Figura A-3: Crearea unui fișier sursă VHDL

4. Apăsați **Finish**. Se va crea noul fișier / entitate și se va deschide automat dialogul de definire a porturilor **Define Module**.
5. Declarați porturile pentru entitatea principală care se va crea, completând informația pentru porturi ca în figura următoare. **Atenție: aceste porturi sunt definite în mod particular pentru placa Basys 3, fiind în principiu suficiente pentru majoritatea proiectelor dezvoltate pe parcursul acestui semestru.** Apăsați **OK**.

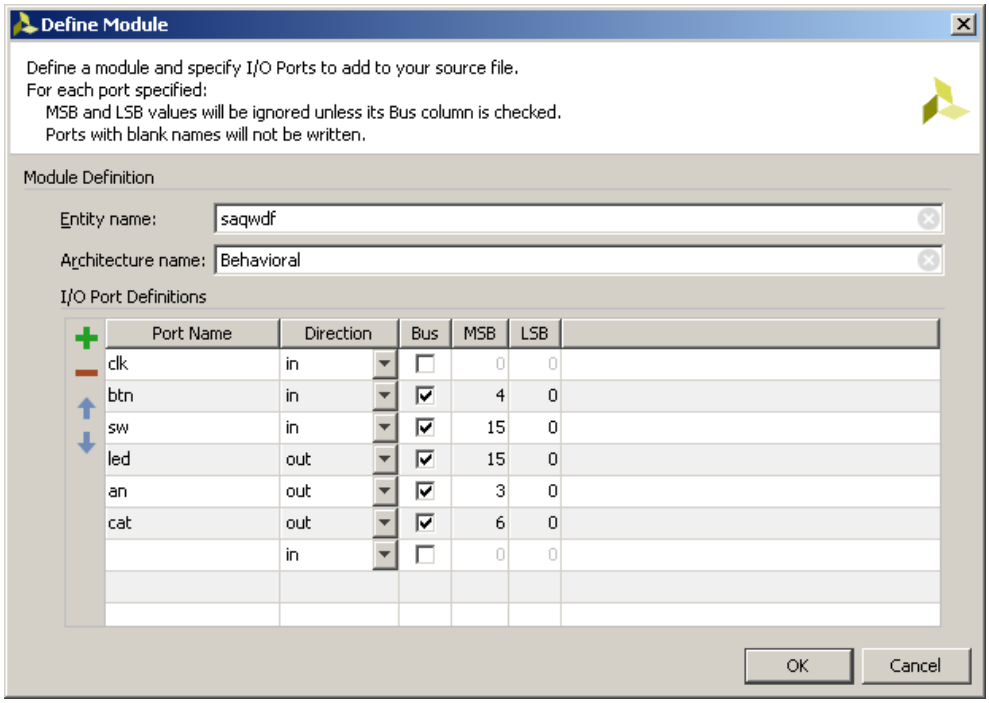


Figura A-4 : Definirea porturilor prin interfața Vivado

6. S-a finalizat crearea noului fișier sursă.

Observație: se poate sări peste pasul de definire a porturilor în interfață (pas 5), caz în care se pot declara (sau corecta/modifica) porturile în secțiunea de declarare a entității create în noul fișier.

Fișierul sursă care conține declararea entității *test_env* și arhitectura ei este afișat acum în mediul Vivado (vezi Figura A-5, pe pagina următoare), iar în zona **Hierarchy** (panoul **Sources**) apare ca modul principal (Top Module) al proiectului curent. Dacă nu vă apare fereastra de editare pentru fișierul nou creat, dați dublu click pe test_env în panoul **Sources**.

De reținut: în proiectele cu mai multe surse, dacă se schimbă în mod accidental entitatea Top Module, se poate seta alta ca Top Module prin click dreapta pe sursa dorită în **Hierarchy,** după care se apasă pe **Set as Top**.

Verificare: Accesați **Project Summary**, la secțiunile **Synthesis** și **Implementation** veți vedea la câmpul *Part* numele dispozitivului (plăcii) ales. Pentru Basys 3, trebuie să fie **xc7a35tcpg236-1.** Dacă nu coincide, înseamnă că ați sărit peste pasul 6 la **Crearea unui nou proiect.** În meniul principal, accesați

Tools/Project Settings, la secțiunea **General**, câmpul *Project Device*, apăsați "…" și reintroduceți proprietățile corecte ale plăcii.

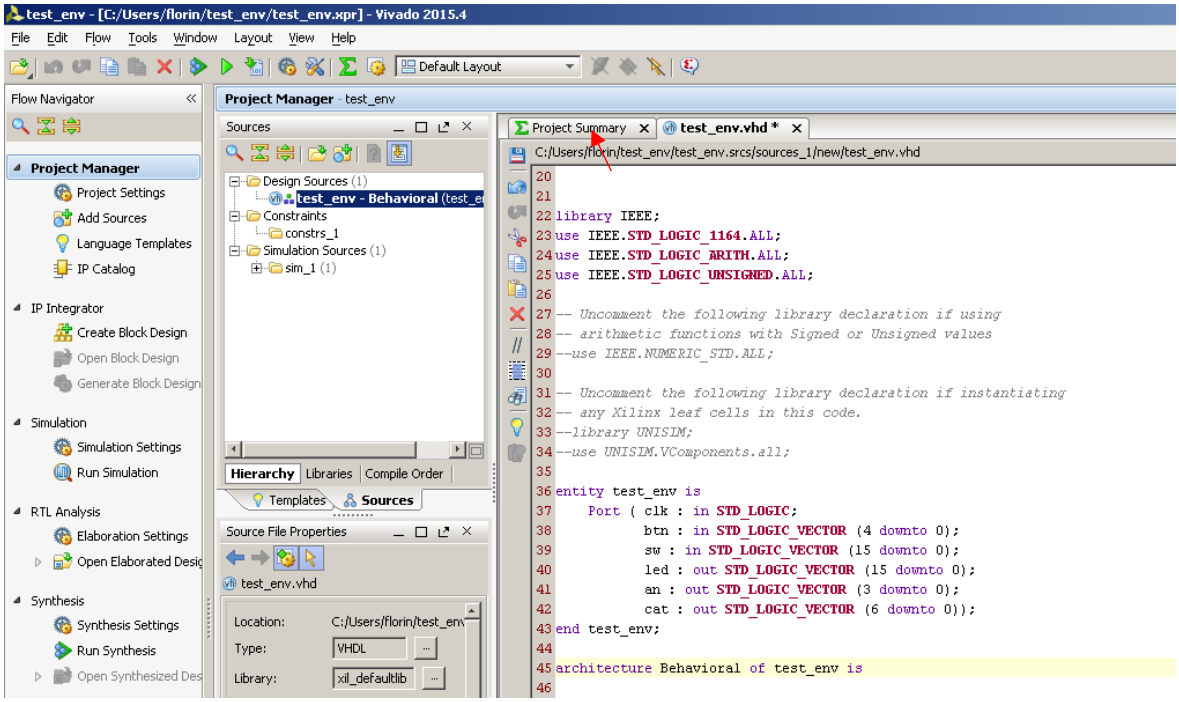


Figura A-5: Noul proiect în Vivado. Project Summary e indicat de săgeata roșie

În fereastra de editare, asigurați-vă că următoarele librării sunt incluse în zona de declarare a librăriilor în *test_env*:

use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

Dacă nu sunt incluse, adăugați-le. Valabil pentru orice proiect (sau fișier sursă) viitor care se va crea de la zero!

(*De revenit aici când se va cere…*) Folosirea Language Templates (VHDL) – opţional

Următorul pas în procesul de creare a noii surse VHDL este adăugarea descrierii comportamentale pentru entitatea Top Level a proiectului. O modalitatea facilă (pentru început, până vă reamintiți limbajul VHDL) este să folosiți exemple de cod din Vivado Language Templates pe care să le particularizați pentru entitatea voastră.

1. Plasați cursorul de editare sub begin în secțiunea architecture a entității unde doriți să adăugați codul.
2. Deschideți Language Templates prin selectarea din meniu a **Tools → Language Templates… .**
3. Navigați în ierarhie prin simbolul "+", către exemplele dorite de cod: **VHDL → Synthesis Constructs → Coding Examples → …**
4. Selectați componenta dorită în ierarhie, copiați codul și inserați-l în destinație.
5. Închideți **Language Templates**.

6. Înlocuiți denumirea implicită a semnalelor din codul inserat cu denumirea semnalelor din entitatea pe care o descrieți.

Editarea finală / sintetizarea sursei VHDL (descrierea comportamentului)

1. În general, veți adăuga componente (cu *component*) și / sau declarații de semnal în secțiunea de declarații între architecture și begin.
2. Apoi, veți adăuga restul codului (instanțierea componentelor – *port map*, descrierea comportamentală – *process* sau atribuiri concurente etc.) între begin și end.
3. **Pentru acest prim proiect** adăugați (fără *Copy/Paste*!) următoarele atribuiri concurente după begin:

```
led <= sw;                  --leagă switch-urile la leduri
an  <= btn(3 downto 0);     --anozii de la SSD la butoane
cat <= (others=>'0');       --catozii de la SSD activi
```

4. Salvați fișierul cu **File → Save File** sau **Ctrl + S**.
5. În zona *Hierachy (panoul Sources)* selectați entitatea *Top L*evel, în cazul de față *test_env*.
6. Vizualizați circuitul rezultat sub formă schematică (relevant mai ales pentru următoarele proiecte, unde vor fi circuite mai complexe): în panoul **Flow Navigator** click pe **RTL Analysis – Elaborated Design → Schematic.** Se va deschide o schemă bloc a circuitului principal, dublu click pe diverse componente pentru a vedea organizarea internă. **Ar trebui să recunoașteți cel puțin o parte a componentelor declarate!** Aceasta este o primă metodă de verificare ca ați descris și legat corect componentele dorite.
7. Corectați eventualele erori care sunt raportate în zona **Console / Messages** (partea de jos a Vivado). Începeți procesul de corectură cu prima eroare!
8. Sintetizați proiectul: în zona Flow Navigator (stânga) click pe **Run Synthesize.** Dacă sunt erori nu mergeți mai departe cu pasul următor (**Implementation**).
9. Corectați eventualele erori care sunt raportate în zona **Console / Messages** (partea de jos a Vivado). Începeți procesul de corectură cu prima eroare!

Obs. Erorile primare de sintaxă vhdl (sau de constrângeri) se văd după salvarea surselor vhdl modificate, în zona de consolă (jos) la categoria Critical Warnings, cu condiția să fiți în Project Manager (dacă sunt deschise alte sub-ferestre: Synthetized Design, Elaborated Design, .., se închid, apoi click în meniu Flow/Project Manager). Se rezolvă aceste Critical Warnings și, pe urmă, se merge mai departe.

Acum ați finalizat procesul de creare a sursei VHDL, fără erori de sintaxă!

Implementarea proiectului

1. Implementați proiectul: în meniu **Flow / Run Implementation** sau în panoul **Flow Navigator** la **Implementation / Run Implementation**.

2. Corectați eventualele erori și avertismente (atenție: o parte dintre avertismente nu necesită corectură, dacă sunt irelevante: ex. anumite semnale sunt declarate dar nu sunt legate încă în proiect).

Stabilirea constrângerilor pentru locațiile de pini (atribuirea pinilor)

Specificați care locații de pini de pe placa de dezvoltare vor fi atribuiți porturilor din proiect (entitatea Top Level *test_env*). Există mai multe metode, mai jos aveți abordarea recomandată (rapidă, fără probleme de compatibilitate):

1. În particular, pentru porturile definite în acest tutorial, găsiți constrângerile definite aici. Descărcați fișierul în directorul personal sau al proiectului. Asigurați-vă că fișierul salvat are extensia *.xdc, și nu *.xdc.txt (cum frecvent se salvează din Chrome/Firefox), altfel nu va fi interpretat corect în Vivado.
2. Click în meniu pe **File\Add Sources**. Sau: în panoul **Flow Navigator** la **Project Manager / Add Sources**.
3. Selectați *Add or create constraints*, click **Next**.
4. Apăsați *Add file*. Pe dialogul apărut selectați fișierul de constrângeri descărcat la pasul anterior. Apăsați **OK**.

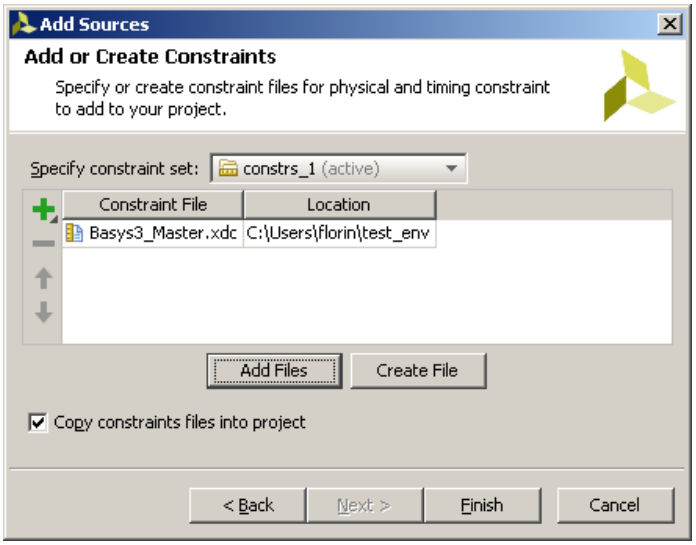


Figura A-6: Adăugarea fișierului de contrângeri la proiect

5. Bifați *Copy constraints files into project* și apăsați **Finish**. Fișierul de constrângeri este adăugat la proiect și vizibil în panoul **Sources** la **Constraints**.

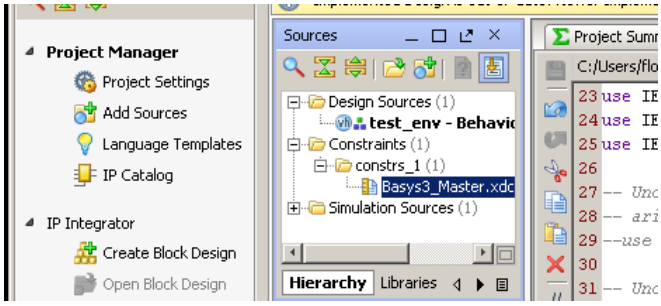


Figura A-7: Fișierul de contrângeri a fost adăugat

6. Click dreapta pe fișierul de constrângeri din zona **Sources** și apăsați *Set as Target Constraint File* pentru a-l selecta ca fiind asociat cu dispozitivul (placa) de testare.
7. Pentru a edita fișierul de constrângeri dați dublu-click pe el în zona **Sources**. O constrângere uzuală (între un port din entitatea principală și o locație de pin de pe placă) este definită prin două linii de sintaxă. Exemplu pentru butonul din centru:

```
set_property PACKAGE_PIN U18 [get_ports {btn[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {btn[0]}]
```

unde U18 reprezintă numele pinului de pe placă asociat cu butonul central, iar btn[0] este portul din entitate cu care vrem să asociem butonul central.

Pentru proiecte viitoare, când va fi necesară adăugarea de noi constrângeri la alți pini, pentru fiecare pin se vor scrie cele două linii în fișierul de constrângeri. Numele pinului (locației) de pe placă se poate găsi în manualul de referință Basys 3, sau pe placă, lângă locația dorită, în paranteze.

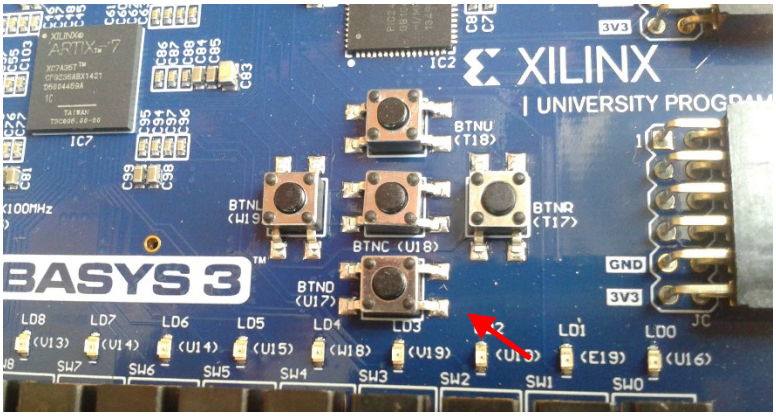


Figura A-8: Numele locațiilor de pini imprimate pe placă. Cu săgeată roșie: pentru butonul central locația de pin este U18

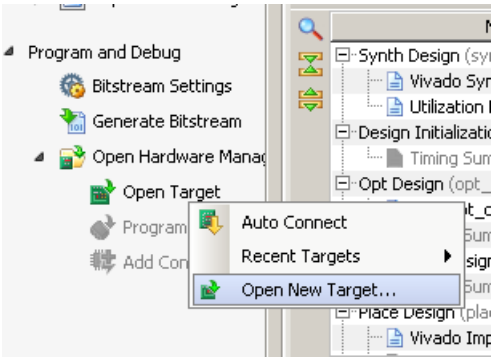
Generarea fișierului de programare (*.bit)

În panoul **Flow Navigator / Program and debug** sau din meniul **Flow** se apasă **Generate Bitstream**. Se va crea fișierul *.bit care se va încărca pe placă la pasul următor.

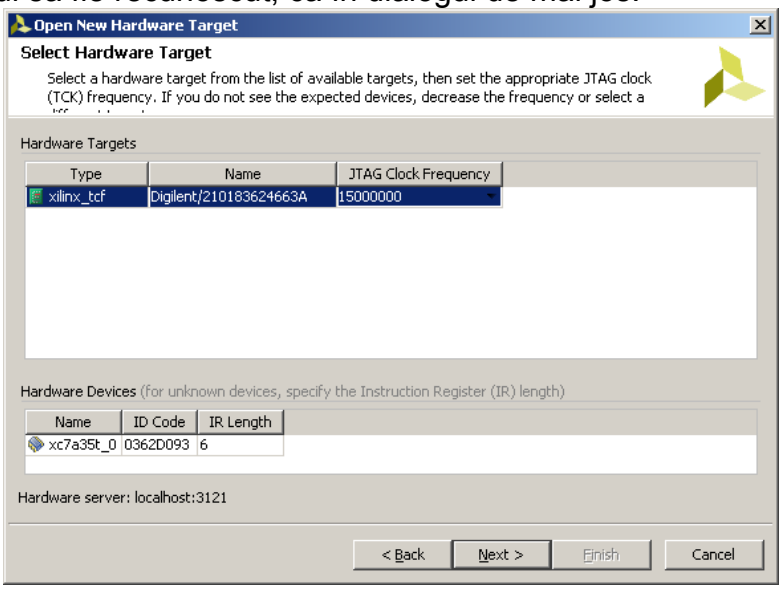
În acest moment, dacă nu există erori, fișierul test_env.bit a apărut în structura de directoare ale proiectului. Altfel, citiți erorile semnalate în zona de consolă și corectați-le. Erorile din acest pas sunt de obicei din cauză că ați sărit peste pasul 6 la **Crearea unui nou proiect.** În meniul principal, accesați **Tools/Project Settings**, la secțiunea **General**, câmpul *Project Device*, apăsați "…" și reintroduceți proprietățile corecte ale plăcii.

Încărcarea proiectului (*.bit) pe placa de dezvoltare Basys 3

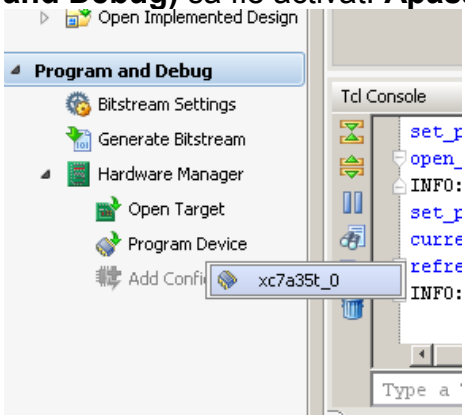
1. Conectați placa la portul USB (**nu forțați** portul de microUSB, asigurați-vă ca îl introduceți în poziția corectă!). Porniți placa de la switch-ul de lângă portul de microUSB.
2. În panoul **Flow Navigator**, secțiunea **Program and Debug/Hardware Manager**, dați click pe **Open Target**, iar în meniul deschis alegeți **Open New Target**.



3. **Next**. Lăsați *Local server* la câmpul Connect to. **Next**. Ar trebui ca dispozitivul să fie recunoscut, ca în dialogul de mai jos.



4. **Next. Finish**.
5. Dacă placa a fost recunoscută, ar trebui ca **Program Device (Flow Manager / Program and Debug)** să fie activat. **Apăsați-l**.



6. Apare un meniu pop-up cu lista de dispozitive conectate, unul singur în cazul de față (Basys 3 – **xc7a35t_0**). **Apăsați-l**.
7. În dialogul deschis trebuie să selectați calea spre fișierul *.bit cu programarea (e posibil să fie deja completată). Acest fișier se găsește de obicei în directorul proiectului, în calea *nume_proiect*.runs/impl_1/. Pentru proiectul test_env: test_env.runs/impl_1/test_env.bit. Click **Program**.
8. În acest moment proiectul este încărcat pe placă. Experimentați (switch-uri, butoane, leduri)!
9. Pentru re-programare, dacă nu e activ **Program Device**, se poate apăsa **Open Target / Auto Connect**.

Probleme legate de programare și eventuale soluții

Problemă

Placa Basys 3 nu este recunoscută.

a) Încercați alt port USB, dacă începe automat un proces de instalare a driver-ului și vă cere drepturi de administrator, cereți ajutorul profesorului.
b) Încercați cu alt cablu.
c) Încercați altă placă (raportați asta profesorului).
d) Reporniți mediul Vivado / Calculatorul.
e) Schimbați stația de lucru.

B. Anexa 2 – Circuit de deplasare combinaţional

Un circuit de deplasare se poate implementa și cu o secvență multi-nivel de multiplexoare 2:1. Ieșirile de pe fiecare nivel de multiplexoare se conectează deplasat la intrările următorului nivel, în funcție de distanța de deplasare dorită. În figura următoare se prezintă un circuit de deplasare la dreapta pe 8 biți, realizat cu 3 niveluri de multiplexoare.

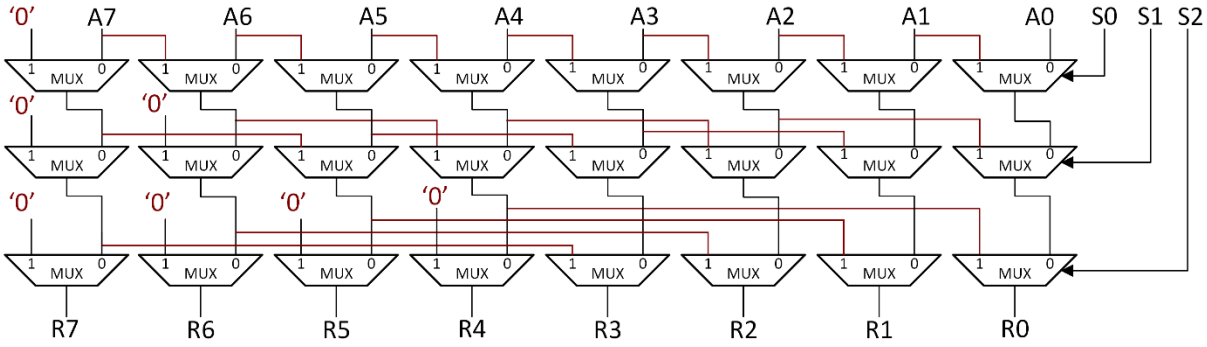


Figura B-1: Circuit de deplasare la dreapta realizat cu mux-uri

Pe pagina următoare se prezintă un exemplu de implementare în VHDL a unui circuit de deplasare pe 5 biți, cu diferite posibilități de control a operației de deplasare.

Exemplu de circuit de deplasare în VHDL, cu număr variabil de poziții:

- sw(4 downto 0) semnal pe 5 biți care poate fi deplasat la stânga sau la dreapta (aritmetic)
- sw(6:5) arată câte poziții se deplasează: 0, 1, 2 or 3
- sw(7) arată direcția de deplasare
- sw(7) = 0 – deplasare la stânga
- sw(7) = 1 – deplasare la dreapta aritmetică
- rezultatul este afișat pe ledurile plăcii de dezvoltare.

Descrierea în VHDL cu două procese:

```
process(sw)
begin
   if sw(5) = '1' then   -- shift with 1 position
      if sw(7) = '0' then
         shift1 <= sw(3 downto 0) & '0';   -- shift left
      else
         shift1 <= sw(4) & sw(4 downto 1);  -- shift right arithmetic
      end if;
   else
      shift1 <= sw(4 downto 0);
   end if;
end process;

process(sw, shift1)
begin
   if sw(6) = '1' then    -- shift with 2 position
      if sw(7) = '0' then
         shift2 <= shift1(2 downto 0) & "00";     -- shift left
      else
         shift2 <= shift1(4) & shift1(4) & shift1(4 downto 2);-- shift right arithmetic
      end if;
   else
      shift2 <= shift1;
   end if;
end process;

led <= shift2;
```

C. Anexa 3 – Implementare pentru blocul de registre (8x8 biți)

```
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;
```

D. Anexa 4 – Implementare RAM (256x16 biți)

Exemplul următor este de memorie RAM, mod “no change”:

```
entity rams_no_change is
        port (  clk     : in std_logic;
                we      : in std_logic;
                en      : in std_logic;
                addr    : in std_logic_vector(7 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 255) 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;
```

E. Anexa 5 – Instrucțiuni uzuale (selecție) pentru MIPS 32

Se prezintă, pentru fiecare instrucțiune reprezentativă, inclusiv formatul pe biți. Setul complet de instrucțiuni poate fi consultat în referințele din laboratorul 4, cu subiectul MIPS® Architecture for Programmers.

ADD – Add

Descriere	Adună două registre și memorează rezultatul în al treilea
Operație	$d ← $s + $t; PC ← PC + 4;
Sintaxă	add $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 100000

ADDI – Add immediate

Descriere	Adună un registru cu o valoare imediată și memorează rezultatul în alt registru
Operație	$t ← $s + imm; PC ← PC + 4;
Sintaxă	addi $t, $s, imm
Format	001000 sssss ttttt iiiiiiiiiiiiiiii

AND – Bitwise and

Descriere	ȘI logic între două registre și memorează rezultatul în al treilea
Operație	$d ← $s & $t; PC ← PC + 4;
Sintaxă	and $d, $s, $t
Format	000000 ssssst tttt ddddd 00000 100100

BEQ – Branch on equal

Descriere	Salt condiționat dacă este egalitate între două registre
Operație	if $s == $t then PC ← PC + 4 + (offset << 2); else PC ← PC + 4;
Sintaxă	beq $s, $t, offset
Format	000100 sssss ttttt iiiiiiiiiiiiiiii

BGEZ – Branch on greater than or equal to zero

Descriere	Salt condiționat dacă un registru este mai mare sau egal cu 0
Operație	if $s >= 0 PC ← PC + 4 + (offset << 2); else PC ← PC + 4;
Sintaxă	bgez $s, offset
Format	000001 sssss 00001 iiiiiiiiiiiiiiii

BGTZ – Branch on greater than zero

Descriere	Salt condiționat dacă un registru este mai mare ca 0
Operație	if $s > 0 PC ← PC + 4 + (offset << 2); else PC ← PC + 4;
Sintaxă	bgtz $s, offset
Format	000111 sssss 00000 iiiiiiiiiiiiiiii

BNE – Branch on not equal

Descriere	Salt condiționat dacă două registre sunt diferite
Operație	if $s != $t PC ← PC + 4 + (offset << 2); else PC ← PC + 4;
Sintaxă	bne $s, $t, offset
Format	000101 sssss ttttt iiiiiiiiiiiiiiii

J – Jump

Descriere	Salt la adresă absolută
Operație	PC ← ((PC +4) & 0xf0000000) | (target << 2);
Sintaxă	j target
Format	000010 iiiiiiiiiiiiiiiiiiiiiiiiii

JAL – Jump and link

Descriere	Salt la adresă absolută și memorarea adresei de revenire în $31
Operație	$31 ← PC + 8; PC ← ((PC +4) & 0xf0000000) | (target << 2);
Sintaxă	jal target
Format	000011 iiiiiiiiiiiiiiiiiiiiiiiiii

JR – Jump register

Descriere	Salt la adresa absolută conținută de registrul $s
Operație	PC ← $s;
Sintaxă	jr $s
Format	000000 sssss 00000 00000 00000 001000

LW – Load word

Descriere	Un cuvânt din memorie este încărcat într-un registru
Operație	$t ← MEM[$s + offset]; PC ← PC + 4;
Sintaxă	lw $t, offset($s)
Format	100011 sssss ttttt iiiiiiiiiiiiiiii

NOOP – no operation, echivalentă cu SLL $0, $0, 0 (fără efect în procesor)

Descriere	Nu se efectuează nici o operație
Operație	PC ← PC + 4;
Sintaxă	noop
Format	000000 00000 00000 00000 00000 000000

OR – Bitwise or

Descriere	SAU logic între două registre și memorează rezultatul în al treilea
Operație	$d ← $s | $t; PC ← PC + 4;
Sintaxă	or $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 100101

ORI – Bitwise or immediate

Descriere	SAU logic între un registru și o valoare imediată, memorează rezultatul în alt registru
Operație	$t ← $s | imm; PC ← PC + 4;
Sintaxă	ori $t, $s, imm
Format	001101 sssss ttttt iiiiiiiiiiiiiiii

SLL – Shift left logical

Descriere	Deplasare logică la stânga pentru un registru, rezultatul este memorat în altul, se introduc zerouri
Operație	$d ← $t << h; PC ← PC + 4;
Sintaxă	sll $d, $t, h
Format	000000 sssss ttttt ddddd hhhhh 000000

SLLV – Shift left logical variable

Descriere	Deplasare logică la stânga pentru un registru, cu un număr de poziții indicat de alt registru, iar rezultatul este memorat într-un al treilea registru
Operație	$d ← $t << $s; PC ← PC + 4;
Sintaxă	sllv $d, $t, $s
Format	000000 sssss ttttt ddddd 00000 000100

SLT – Set on less than (signed)

Descriere	Dacă $s < $t, $d este inițializat cu 1, altfel cu 0
Operație	PC ← PC + 4; if $s < $t $d ← 1; else $d ← 0;
Sintaxă	slt $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 101010

SLTI – Set on less than immediate (signed)

Descriere	Dacă $s este mai mic decât un imediat, $t este inițializat cu 1, altfel cu 0
Operație	PC ← PC + 4; if $s < imm $t ← 1; else $t ← 0;
Sintaxă	slti $t, $s, imm
Format	001010 sssss ttttt iiiiiiiiiiiiiiii

SRA – Shift right arithmetic

Descriere	Deplasare aritmetică la dreapta pentru un registru, rezultatul este memorat în altul. Se repetă valoarea bitului de semn
Operație	$d ← $t >> h; PC ← PC + 4;
Sintaxă	sra $d, $t, h
Format	000000 00000 ttttt ddddd hhhhh 000011

SRL – Shift right logical

Descriere	Deplasare logică la dreapta pentru un registru, rezultatul este memorat în altul, se introduc zerouri
Operație	$d ← $t >> h; PC ← PC + 4;
Sintaxă	srl $d, $t, h
Format	000000 00000 ttttt ddddd hhhhh 000010

SUB – Subtract

Descriere	Scade două registre și memorează rezultatul în al treilea
Operație	$d ← $s - $t; PC ← PC + 4;
Sintaxă	sub $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 100010

SW – Store word

Descriere	Valoarea unui registru este stocată în memorie la o anumită adresă
Operație	MEM[$s + offset] ← $t; PC ← PC + 4;
Sintaxă	sw $t, offset($s)
Format	101011 sssss ttttt iiiiiiiiiiiiiiii

XOR – Bitwise exclusive or

Descriere	SAU exclusiv logic între două registre, memorează rezultatul în alt registru
Operație	$d ← $s ^ $t; PC ← PC + 4;
Sintaxă	xor $d, $s, $t
Format	000000 sssss ttttt ddddd 00000 100110

F. Anexa 6 – Implementări pentru automatele cu stări finite

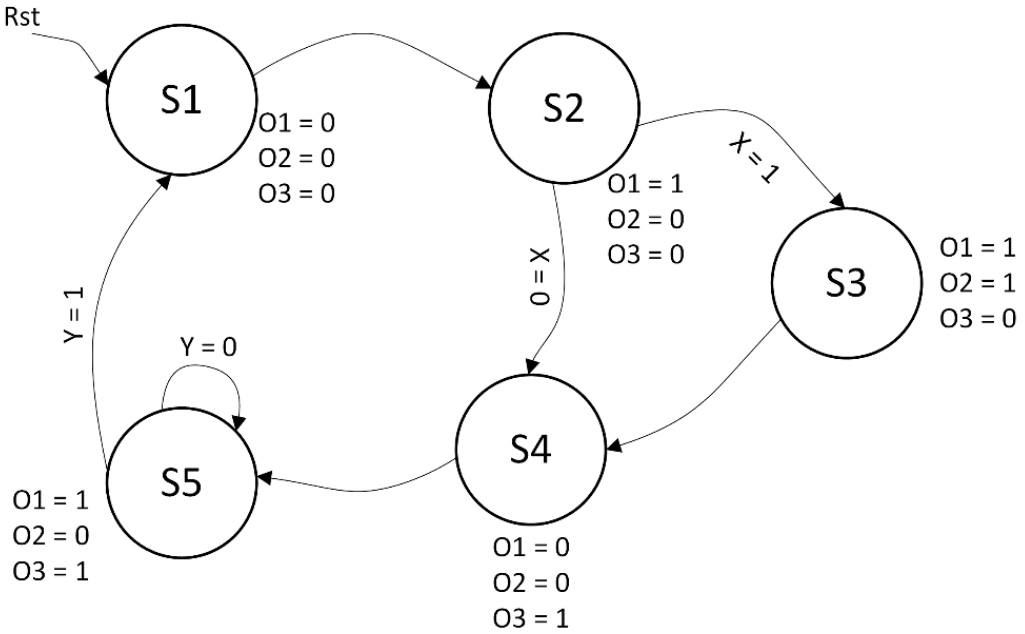


Figura F-1: Exemplu FSM (adaptat după XST User Guide)

Pini intrare/ieșire	**Descriere**
clk	Semnal de ceas (front crescător)
Rst	Reset asincron (activ pe 1)
X, Y	Intrări FSM
O1, O2, O3	Ieșiri FSM

Tabel F.1: Descrierea pinilor FSM

În mod implicit, Xilinx încearcă să recunoască FSM-urile scrise în codul VHDL. Pe paginile următoare se prezintă cele 3 tipuri de descriere pentru FSM-ul prezentat în Figura F-1, care sunt recunoscute automat de Xilinx, la sintetizare: cu 1, 2 sau 3 procese.

În implementarea cu un proces, toate atribuirile (starea următoare, decodificarea semnalelor de ieșire) se fac sincron, pe frontul crescător de ceas.

În implementarea cu 2 procese, în primul proces se calculează și atribuie sincron starea următoare. Decodificarea stării curente, pentru a stabili valoarea ieșirilor, se face combinaţional în al doilea proces.

În varianta de implementare cu 3 procese, două sunt combinaţionale: cel care decodifică starea curentă în valoarea ieșirilor și cel care calculează valoarea stării următoare. Al treilea proces este sincron, realizând tranziția între stări (atribuirea sincronă a stării următoare la starea curentă).

Exemplu de descriere VHDL: **FSM cu un proces**

```
entity fsm_1 is
port (
        clk, rst, x, y      : IN std_logic;
        o1, o2, o3          : OUT std_logic
);
end entity;

architecture beh1 of fsm_1 is
type state_type is (s1, s2, s3, s4, s5);
signal state : state_type;
begin

process (clk, rst, x, y)
begin
        if (rst ='1') then
                state <=s1;
                o1<='0'; o2<='0'; o3<='0';
        elsif (clk='1' and clk'event) then
                case state is
                        when s1 =>      state <= s2;
                                        o1<='1'; o2<='0'; o3<='0';
                        when s2 =>      if x = '1' then
                                                state <= s3;
                                                o1<='1'; o2<='1'; o3<='0';
                                                else
                                                state <= s4;
                                                o1<='0'; o2<='0'; o3<='1';
                                        end if;
                        when s3 =>      state <= s4;
                                        o1<='0'; o2<='0'; o3<='1';
                        when s4 =>      state <= s5;
                                        o1<='1'; o2<='0'; o3<='1';
                        when s5 =>      if y = ‘1’ then
                                                state <= s1;
                                                o1<='0'; o2<='0'; o3<='0';
                                        else
                                                state <= s5;
                                                o1<='1'; o2<='0'; o3<='1';
                                        end if;
                end case;
        end if;
end process;

end beh1;
```

Exemplu de descriere VHDL: **FSM cu 2 procese**

```
entity fsm_2 is
port (
        clk, rst, x, y      : IN std_logic;
        o1, o2, o3          : OUT std_logic
);
end entity;

architecture beh1 of fsm_2 is
type state_type is (s1, s2, s3, s4, s5);
signal state : state_type;
begin

process1: process (clk, rst, x, y)
begin
        if (rst ='1') then
                state <=s1;
        elsif (clk='1' and clk'event) then
                case state is
                        when s1 =>      state <= s2;
                        when s2 =>      if x = '1' then
                                                state <= s3;
                                        else
                                                state <= s4;
                                        end if;
                        when s3 =>      state <= s4;
                        when s4 =>      state <= s5;
                        when s5 =>      if y = '1' then
                                                state <= s1;
                                        else
                                                state <= s5;
                                        end if;
                end case;
        end if;
end process process1;


process2: process (state)
begin
        case state is
                when s1 => o1<='0'; o2<='0'; o3<='0';
                when s2 => o1<='1'; o2<='0'; o3<='0';
                when s3 => o1<='1'; o2<='1'; o3<='0';
                when s4 => o1<='1'; o2<='0'; o3<='0';
                when s5 => o1<='1'; o2<='0'; o3<='1';
        end case;
end process process2;
end beh1;
```

Exemplu de descriere VHDL: **FSM cu 3 procese**

```
entity fsm_3 is
port (
        clk, rst, x, y      : IN std_logic;
        o1, o2, o3          : OUT std_logic
);
end entity;

architecture beh1 of fsm_3 is
type state_type is (s1, s2, s3, s4, s5);
signal state, next_state : state_type;
begin

process1: process (clk, rst)
begin
        if (reset ='1') then
                state <=s1;
        elsif (clk='1' and clk'event) then
                state <= next_state;
        end if;
end process process1;

process2: process (state, x, y)
begin
        case state is
                when s1 =>      next_state <= s2;
                when s2 =>      if x = '1' then
                                        next_state <= s3;
                                else
                                        next_state <= s4;
                                end if;
                when s3 =>      next_state <= s4;
                when s4 =>      next_state <= s5;
                when s5 =>      if y = '1' then
                                        next_state <= s1;
                                else
                                        next_state <= s5;
                                end if;
        end case;
end process process2;

process3: process (state)
begin
        case state is
                when s1 => o1<='0'; o2<='0'; o3<='0';
                when s2 => o1<='1'; o2<='0'; o3<='0';
                when s3 => o1<='1'; o2<='1'; o3<='0';
                when s4 => o1<='1'; o2<='0'; o3<='0';
                when s5 => o1<='1'; o2<='0'; o3<='1';
        end case;
end process process3;
end beh1;
```

G. Anexa 7 – Tabel cu coduri ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

Figura G-1: ASCII Codes (http://www.asciitable.com/)