Laborator 04

Procesorul MIPS – versiune pe 16 biți, cu un ciclu de ceas pe instrucțiune

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

1. Obiective

Studiul, proiectarea, implementarea și testarea:

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

Familiarizarea studenților cu

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

2. Descrierea procesorului MIPS, simplificat pe 16 biți

(!) Citiți cursurile 3 (obligatoriu) și 4 (după predare) pentru a înțelege conținutul acestui laborator.

În acest laborator veți face proiectarea (în mod simplificat, prin desenarea căilor de date – proiectarea completă se face în cursul 4) și veți începe implementarea versiunii proprii 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 regiștrii de uz general, respectiv dimensiune mai mică a memoriei. În rest principiile 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 datele vor fi pe 16 biți. Formatul celor 3 tipuri de instrucțiuni este prezentat mai jos. Comparați acest format cu formatul din curs pe 32 de biți. Observați modificările / limitările.

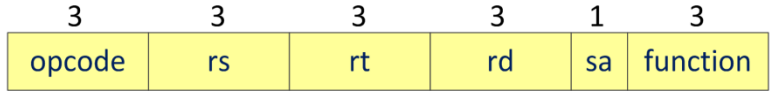

Figura 1: Instrucțiune de tip R

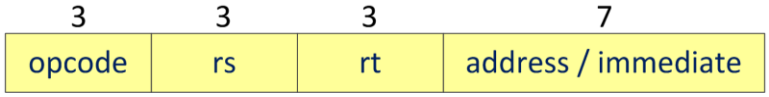

Figura 2: Instrucțiune de tip I

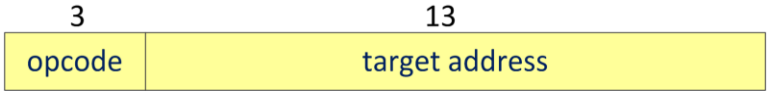

Figura 3: Instrucțiune de tip J

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
	….	…
	….	…
Instrucțiuni de tip I	Add Immediate	addi
	Load Word	lw
	Store Word	sw
	Branch on Equal	beq
	….	…
	….	…

Instrucțiuni de tip J	Jump	j

Tabel 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.

Registrului PC, contorul de program:

- Registru pe 16 biți, pe front crescător (bistabil D)

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
- Combinaţional, fără semnale de control

Blocul de regiștrii RF

- 2 adrese de citire (Read register 1, Read register 1) și o adresă de scriere (Write register)
- 8 regiștrii de câte 16-biți (rs, rt, 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 combinațional.
- 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
- Un semnal de control: MemWrite

Unitatea de extindere:

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

Unitatea aritmetico-logică ALU

- ALU efectuează operații aritmetico-logice
- (!) Identificați toate operațiile pe care ALU trebuie să le efectueze, după definirea instrucțiunilor din tabelul 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).

3. Activități practice

Citiți fiecare activitate în întregime, înainte să o începeți!

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 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, sll, and, lw, beq, j. Pentru restul instrucțiunilor faceți diagrama de procesare acasă ca temă.

Pe lângă materialul de curs, folosiți Anexa 6 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 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, O pentru hexa sau octal):

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

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 3.1 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. Practic programul vostru trebuie să fie vizibil în paralel cu codul mașină.

Opțional pentru acasă: dacă doriți extinderea programului spre ceva mai complex, sunteți liberi să o faceți!

3.3. Căile de date pentru MIPS 16 – temă, după cursul 4

Desenați căile 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 4 pentru exemple).

4. Referințe

[1] Computer Architecture Lectures 3 & 4 slides.
[2] MIPS® Architecture For Programmers, Volume I-A: Introduction to the MIPS32® Architecture, Document Number: MD00082, Revision 5.01, December 15, 2012
[3] MIPS® Architecture For Programmers Volume II-A: The MIPS32® Instruction Set Manual, Revision 6.02
[4] 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.

Anexa 6 – Instrucțiuni uzuale pentru MIPS 32

Observație: Toate valorile imediate trebuie extinse cu semn.
Excepție: pentru operații logice, imediatul se extinde cu zero.
După extindere, valorile sunt tratate ca valori pe 32 de biți, cu sau fără semn.

Pentru instrucțiunile care nu conțin imediat, singura diferență între varianta cu semn și cea fără semn (ex. ADD vs ADDU) este ca versiunea cu semn poate genera excepție în caz de depășire.

Mai jos sunt prezentate pentru fiecare instrucțiune inclusiv formatul pe biți. Detalii în plus găsiți aici: „MIPS Single Cycle Processor", John Alexander, Barret Schloerke, Daniel Sedam, Iowa State University.

ADD – Add

Description:	Adds two registers and stores the result in a register
Operation:	$d ← $s + $t; advance_pc (4);
Syntax:	add $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0000

ADDI – Add immediate

Description:	Adds a register and a signed immediate value and stores the result in a register
Operation:	$t ← $s + imm; advance_pc (4);
Syntax:	addi $t, $s, imm
Encoding:	0010 00ss ssst tttt iiii iiii iiii iiii

ADDIU – Add immediate unsigned

Description:	Adds a register and an unsigned immediate value and stores the result in a register
Operation:	$t ← $s + imm; advance_pc (4);
Syntax:	addiu $t, $s, imm
Encoding:	0010 01ss ssst tttt iiii iiii iiii iiii

ADDU – Add unsigned

Description:	Adds two registers and stores the result in a register
Operation:	$d ← $s + $t; advance_pc (4);
Syntax:	addu $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0001

AND – Bitwise and

Description:	Bitwise ands two registers and stores the result in a register
Operation:	$d ← $s & $t; advance_pc (4);
Syntax:	and $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0100

ANDI – Bitwise and immediate

Description:	Bitwise ands a register and an immediate value and stores the result in a register
Operation:	$t ← $s & imm; advance_pc (4);
Syntax:	andi $t, $s, imm
Encoding:	0011 00ss ssst tttt iiii iiii iiii iiii

BEQ – Branch on equal

Description:	Branches if the two registers are equal
Operation:	if $s == $t advance_pc (offset << 2); else advance_pc (4);
Syntax:	beq $s, $t, offset
Encoding:	0001 00ss ssst tttt iiii iiii iiii iiii

BGEZ – Branch on greater than or equal to zero

Description:	Branches if the register is greater than or equal to zero
Operation:	if $s >= 0 advance_pc (offset << 2); else advance_pc (4);
Syntax:	bgez $s, offset
Encoding:	0000 01ss sss0 0001 iiii iiii iiii iiii

BGEZAL – Branch on greater than or equal to zero and link

Description:	Branches if the register is greater than or equal to zero and saves the return address in $31
Operation:	if $s >= 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2); else advance_pc (4);
Syntax:	bgezal $s, offset
Encoding:	0000 01ss sss1 0001 iiii iiii iiii iiii

BGTZ – Branch on greater than zero

Description:	Branches if the register is greater than zero
Operation:	if $s > 0 advance_pc (offset << 2); else advance_pc (4);
Syntax:	bgtz $s, offset
Encoding:	0001 11ss sss0 0000 iiii iiii iiii iiii

BLEZ – Branch on less than or equal to zero

Description:	Branches if the register is less than or equal to zero
Operation:	if $s <= 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	blez $s, offset
Encoding:	0001 10ss sss0 0000 iiii iiii iiii iiii

BLTZ – Branch on less than zero

Description:	Branches if the register is less than zero
Operation:	if $s < 0 advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bltz $s, offset
Encoding:	0000 01ss sss0 0000 iiii iiii iiii iiii

BLTZAL – Branch on less than zero and link

Description:	Branches if the register is less than zero and saves the return address in $31
Operation:	if $s < 0 $31 = PC + 8 (or nPC + 4); advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bltzal $s, offset
Encoding:	0000 01ss sss1 0000 iiii iiii iiii iiii

BNE – Branch on not equal

Description:	Branches if the two registers are not equal
Operation:	if $s != $t advance_pc (offset << 2)); else advance_pc (4);
Syntax:	bne $s, $t, offset
Encoding:	0001 01ss ssst tttt iiii iiii iiii iiii

DIV – Divide

Description:	Divides $s by $t and stores the quotient in $LO and the remainder in $HI
Operation:	$LO ← $s / $t; $HI ← $s % $t; advance_pc (4);
Syntax:	div $s, $t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1010

DIVU – Divide unsigned

Description:	Divides $s by $t and stores the quotient in $LO and the remainder in $HI
Operation:	$LO ← $s / $t; $HI ← $s % $t; advance_pc (4);
Syntax:	divu $s, $t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1011

J – Jump

Description:	Jumps to the calculated address
Operation:	PC ← nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax:	j target
Encoding:	0000 10ii iiii iiii iiii iiii iiii iiii

JAL – Jump and link

Description:	Jumps to the calculated address and stores the return address in $31
Operation:	$31 ← PC + 8 (or nPC + 4); PC = nPC; nPC = (PC & 0xf0000000) | (target << 2);
Syntax:	jal target
Encoding:	0000 11ii iiii iiii iiii iiii iiii iiii

JR – Jump register

Description:	Jump to the address contained in register $s
Operation:	PC ← nPC; nPC = $s;
Syntax:	jr $s
Encoding:	0000 00ss sss0 0000 0000 0000 0000 1000

LB – Load byte

Description:	A byte is loaded into a register from the specified address.
Operation:	$t ← MEM[$s + offset]; advance_pc (4);
Syntax:	lb $t, offset($s)
Encoding:	1000 00ss ssst tttt iiii iiii iiii iiii

LUI – Load upper immediate

Description:	The immediate value is shifted left 16 bits and stored in the register. The lower 16 bits are zeroes.
Operation:	$t ← (imm << 16); advance_pc (4);
Syntax:	lui $t, imm
Encoding:	0011 11-- ---t tttt iiii iiii iiii iiii

LW – Load word

Description:	A word is loaded into a register from the specified address.
Operation:	$t ← MEM[$s + offset]; advance_pc (4);
Syntax:	lw $t, offset($s)
Encoding:	1000 11ss ssst tttt iiii iiii iiii iiii

MFHI – Move from HI

Description:	The contents of register HI are moved to the specified register.
Operation:	$d ← $HI; advance_pc (4);
Syntax:	mfhi $d
Encoding:	0000 0000 0000 0000 dddd d000 0001 0000

MFLO – Move from LO

Description:	The contents of register LO are moved to the specified register.
Operation:	$d ← $LO; advance_pc (4);
Syntax:	mflo $d
Encoding:	0000 0000 0000 0000 dddd d000 0001 0010

MULT – Multiply

Description:	Multiplies $s by $t and stores the result in $LO.
Operation:	$HI, $LO ← $s * $t; advance_pc (4);
Syntax:	mult $s, $t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1000

MULTU – Multiply unsigned

Description:	Multiplies $s by $t and stores the result in $LO.
Operation:	$HI, $LO ← $s * $t; advance_pc (4);
Syntax:	multu $s, $t
Encoding:	0000 00ss ssst tttt 0000 0000 0001 1001

NOOP – no operation

Description:	Performs no operation.
Operation:	advance_pc (4);
Syntax:	noop
Encoding:	0000 0000 0000 0000 0000 0000 0000 0000

Note: The encoding for a NOOP represents the instruction SLL $0, $0, 0 which has no side effects. In fact, nearly every instruction that has $0 as its destination register will have no side effect and can thus be considered a NOOP instruction.

OR – Bitwise or

Description:	Bitwise logical ors two registers and stores the result in a register
Operation:	$d ← $s | $t; advance_pc (4);
Syntax:	or $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0101

ORI – Bitwise or immediate

Description:	Bitwise ors a register and an immediate value and stores the result in a register
Operation:	$t ← $s | imm; advance_pc (4);
Syntax:	ori $t, $s, imm
Encoding:	0011 01ss ssst tttt iiii iiii iiii iiii

SB – Store byte

Description:	The least significant byte of $t is stored at the specified address.
Operation:	MEM[$s + offset] ← (0xff & $t); advance_pc (4);
Syntax:	sb $t, offset($s)
Encoding:	1010 00ss ssst tttt iiii iiii iiii iiii

SLL – Shift left logical

Description:	Shifts a register value left by the shift amount listed in the instruction and places the result in a third register. Zeroes are shifted in.
Operation:	$d ← $t << h; advance_pc (4);
Syntax:	sll $d, $t, h
Encoding:	0000 00ss ssst tttt dddd dhhh hh00 0000

SLLV – Shift left logical variable

Description:	Shifts a register value left by the value in a second register and places the result in a third register. Zeroes are shifted in.
Operation:	$d ← $t << $s; advance_pc (4);
Syntax:	sllv $d, $t, $s
Encoding:	0000 00ss ssst tttt dddd d--- --00 0100

SLT – Set on less than (signed)

Description:	If $s is less than $t, $d is set to one. It gets zero otherwise.
Operation:	if $s < $t $d ← 1; advance_pc (4); else $d ← 0; advance_pc (4);
Syntax:	slt $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 1010

SLTI – Set on less than immediate (signed)

Description:	If $s is less than immediate, $t is set to one. It gets zero otherwise.
Operation:	if $s < imm $t ← 1; advance_pc (4); else $t ← 0; advance_pc (4);
Syntax:	slti $t, $s, imm
Encoding:	0010 10ss ssst tttt iiii iiii iiii iiii

SLTIU – Set on less than immediate unsigned

Description:	If $s is less than the unsigned immediate, $t is set to one. It gets zero otherwise.
Operation:	if $s < imm $t ← 1; advance_pc (4); else $t ← 0; advance_pc (4);
Syntax:	sltiu $t, $s, imm
Encoding:	0010 11ss ssst tttt iiii iiii iiii iiii

SLTU – Set on less than unsigned

Description:	If $s is less than $t, $d is set to one. It gets zero otherwise.
Operation:	if $s < $t $d ← 1; advance_pc (4); else $d ← 0; advance_pc (4);
Syntax:	sltu $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 1011

SRA – Shift right arithmetic

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. The sign bit is shifted in.
Operation:	$d ← $t >> h; advance_pc (4);
Syntax:	sra $d, $t, h
Encoding:	0000 00-- ---t tttt dddd dhhh hh00 0011

SRL – Shift right logical

Description:	Shifts a register value right by the shift amount (shamt) and places the value in the destination register. Zeroes are shifted in.
Operation:	$d ← $t >> h; advance_pc (4);
Syntax:	srl $d, $t, h
Encoding:	0000 00-- ---t tttt dddd dhhh hh00 0010

SRLV – Shift right logical variable

Description:	Shifts a register value right by the amount specified in $s and places the value in the destination register. Zeroes are shifted in.
Operation:	$d ← $t >> $s; advance_pc (4);
Syntax:	srlv $d, $t, $s
Encoding:	0000 00ss ssst tttt dddd d000 0000 0110

SUB – Subtract

Description:	Subtracts two registers and stores the result in a register
Operation:	$d ← $s - $t; advance_pc (4);
Syntax:	sub $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0010

SUBU – Subtract unsigned

Description:	Subtracts two registers and stores the result in a register
Operation:	$d ← $s - $t; advance_pc (4);
Syntax:	subu $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d000 0010 0011

SW – Store word

Description:	The contents of $t is stored at the specified address.
Operation:	MEM[$s + offset] ← $t; advance_pc (4);
Syntax:	sw $t, offset($s)
Encoding:	1010 11ss ssst tttt iiii iiii iiii iiii

SYSCALL – System call

Description:	Generates a software interrupt.
Operation:	advance_pc (4);
Syntax:	syscall
Encoding:	0000 00-- ---- ---- ---- ---- --00 1100

XOR – Bitwise exclusive or

Description:	Exclusive ors two registers and stores the result in a register
Operation:	$d ← $s ^ $t; advance_pc (4);
Syntax:	xor $d, $s, $t
Encoding:	0000 00ss ssst tttt dddd d--- --10 0110

XORI – Bitwise exclusive or immediate

Description:	Bitwise exclusive ors a register and an immediate value and stores the result in a register
Operation:	$t ← $s ^ imm; advance_pc (4);
Syntax:	xori $t, $s, imm
Encoding:	0011 10ss ssst tttt iiii iiii iiii iiii