Student: Todosi george Vasile

GRUPA: 3131B

PROFESOR ÎNDRUMĂTOR: ZAGAN IONEL

ASAMBLOR MIPS32

Cuprins

[Introducere 2](#_Toc502260996)

[Setul de instrucțiuni 3](#_Toc502260997)

[Valorile hexazecimale ale instrucțiunilor 14](#_Toc502260998)

[Regiștrii procesorului 16](#_Toc502260999)

[Tehnologiile utilizate și resurse necesare 17](#_Toc502261000)

[Prezentarea interfeței și instrucțiuni de utilizare 18](#_Toc502261001)

[Bibliografie 21](#_Toc502261002)

# Introducere

MIPS (Microprocessor without Interlocked Pipeline Stages) este un ISA (instruction set architecture) pentru procesoare de tip RISC (reduced instruction set computer) dezvoltate de MIPS Technologies.

Primele arhitecturi MIPS au fost pe 32 de biți mai târziu adăugându-se și versiuni pe 64 de biți. Există mai multe versiuni de MIPS incluzând MIPS I, II, III, IV și V și de asemenea cinci versiuni de MIPS32/64 (pentru implementări pe 32 și respectiv pe 64 de biți).

Arhitectura MIPS a influențat mult arhitecturile RISC ce au urmat după ea, precum Alpha.

Procesoarele MIPS sunt utilizate în sisteme embedded precum routere și gateway-uri. Inițial, MIPS a fost creat pentru uz general în anii 1980 și 1990 ele fiind regăsite în calculatoare de uz personal, stații de lucru și servere utilizate de diverse firme. Mult console de jocuri din trecut precum Nintendo 64, Sony PlayStation, Sony Playstation 2 și Sony PlayStation Portable folosesc procesoare MIPS.

# Setul de instrucțiuni

Procesorul MIPS32 are cinci tipuri de instrucțiuni: R, I, J, FR, FI. În continuare vor fi prezentate doar instrucțiunile de tip R, I, J.

1. Instrucțiunile de tip R:

Aceste instrucțiuni au formatul:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| opcode | rs | rt | rd | shamt | funct |

ADD:

|  |  |
| --- | --- |
| Description: | Adds two registers and stores the result in a register |
| Operation: | $d = $s + $t; advance\_pc (4); |
| Syntax: | add $d, $s, $t |

ADDU:

|  |  |
| --- | --- |
| Description: | Adds two registers and stores the result in a register |
| Operation: | $d = $s + $t; advance\_pc (4); |
| Syntax: | addu $d, $s, $t |

SUB:

|  |  |
| --- | --- |
| Description: | Subtracts two registers and stores the result in a register |
| Operation: | $d = $s - $t; advance\_pc (4); |
| Syntax: | sub $d, $s, $t |

SUBU:

|  |  |
| --- | --- |
| Description: | Subtracts two registers and stores the result in a register |
| Operation: | $d = $s - $t; advance\_pc (4); |
| Syntax: | sub $d, $s, $t |

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 |

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 |

NOR:

XOR:

|  |  |
| --- | --- |
| Description: | Exclusive ors two registers and stores the result in a register |
| Operation: | $d = $s ^ $t; advance\_pc (4); |
| Syntax: | xor $d, $s, $t |

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 |

SHIFT RIGHT LOGICAL:

|  |  |
| --- | --- |
| 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 |

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 |

SHIFT LEFT LOGICAL VAR.:

|  |  |
| --- | --- |
| 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 |

SHIFT RIGHT LOGICAL VAR.:

|  |  |
| --- | --- |
| 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 |

SHIFT RIGHT ARITHMETIC VAR.:

SET LESS THAN:

|  |  |
| --- | --- |
| 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 |

SET 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 |

JUMP REGISTER:

|  |  |
| --- | --- |
| Description: | Jump to the address contained in register $s |
| Operation: | PC = nPC; nPC = $s; |
| Syntax: | jr $s |

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 |

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 |

DIVIDE UNSINGED:

|  |  |
| --- | --- |
| 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 |

MULTIPLY:

|  |  |
| --- | --- |
| Description: | Multiplies $s by $t and stores the result in $LO. |
| Operation: | $LO = $s \* $t; advance\_pc (4); |
| Syntax: | mult $s, $t |

MULTIPLY UNSINGED:

|  |  |
| --- | --- |
| Description: | Multiplies $s by $t and stores the result in $LO. |
| Operation: | $LO = $s \* $t; advance\_pc (4); |
| Syntax: | multu $s, $t |

MOVE FROM HI:

|  |  |
| --- | --- |
| Description: | The contents of register HI are moved to the specified register. |
| Operation: | $d = $HI; advance\_pc (4); |
| Syntax: | mfhi $d |

MOVE FROM LO:

|  |  |
| --- | --- |
| Description: | The contents of register LO are moved to the specified register. |
| Operation: | $d = $LO; advance\_pc (4); |
| Syntax: | mflo $d |

MOVE FROM CONTROL:

1. Instrucțiuni de tip I:

Aceste instrucțiuni au formatul:

|  |  |  |  |
| --- | --- | --- | --- |
| opcode | rs | rt | immediate |

ADDI:

|  |  |
| --- | --- |
| Description: | Adds a register and a sign-extended immediate value and stores the result in a register |
| Operation: | $t = $s + imm; advance\_pc (4); |
| Syntax: | addi $t, $s, imm |

ADDIU:

|  |  |
| --- | --- |
| Description: | Adds a register and a sign-extended immediate value and stores the result in a register |
| Operation: | $t = $s + imm; advance\_pc (4); |
| Syntax: | addiu $t, $s, imm |

ANDI:

|  |  |
| --- | --- |
| 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 |

ORI:

|  |  |
| --- | --- |
| 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 |

XORI:

|  |  |
| --- | --- |
| 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 |

SET LESS THAN IMMEDIATE:

|  |  |
| --- | --- |
| 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 |

SET 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 |

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 |

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 |

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) |

LOAD BYTE UNSIGNED:

LOAD HALFWORD:

LOAD HALFWORD UNSINGED:

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 |

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) |

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) |

STORE HALFWORD:

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) |

LOAD FP SINGLE:

LOAD FP DOULBE:

STORE FP SINGLE:

STORE FP DOUBLE:

1. Instrucțiuni de tip J

Aceste instrucțiuni au formatul:

|  |  |
| --- | --- |
| opcode | immediate |

JUMP:

|  |  |
| --- | --- |
| Description: | Jumps to the calculated address |
| Operation: | PC = nPC; nPC = (PC & 0xf0000000) | (target << 2); |
| Syntax: | j target |

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 |

# Valorile hexazecimale ale instrucțiunilor

Următorul table prezintă valorile hexazecimale are instrucțiunilor cu care lucrează de fapt și de drept procesorul.

| **Mnemonic** | **Meaning** | **Type** | **Opcode** | **Funct** |
| --- | --- | --- | --- | --- |
| **add** | Add | R | 0x00 | 0x20 |
| **addi** | Add Immediate | I | 0x08 | NA |
| **addiu** | Add Unsigned Immediate | I | 0x09 | NA |
| **addu** | Add Unsigned | R | 0x00 | 0x21 |
| **and** | Bitwise AND | R | 0x00 | 0x24 |
| **andi** | Bitwise AND Immediate | I | 0x0C | NA |
| **beq** | Branch if Equal | I | 0x04 | NA |
| **bne** | Branch if Not Equal | I | 0x05 | NA |
| **div** | Divide | R | 0x00 | 0x1A |
| **divu** | Unsigned Divide | R | 0x00 | 0x1B |
| **j** | Jump to Address | J | 0x02 | NA |
| **jal** | Jump and Link | J | 0x03 | NA |
| **jr** | Jump to Address in Register | R | 0x00 | 0x08 |
| **lbu** | Load Byte Unsigned | I | 0x24 | NA |
| **lhu** | Load Halfword Unsigned | I | 0x25 | NA |
| **lui** | Load Upper Immediate | I | 0x0F | NA |
| **lw** | Load Word | I | 0x23 | NA |
| **mfc0** | Move from Coprocessor 0 | R | 0x10 | NA |
| **mfhi** | Move from HI Register | R | 0x00 | 0x10 |
| **mflo** | Move from LO Register | R | 0x00 | 0x12 |
| **mult** | Multiply | R | 0x00 | 0x18 |
| **multu** | Unsigned Multiply | R | 0x00 | 0x19 |
| **nor** | Bitwise NOR (NOT-OR) | R | 0x00 | 0x27 |
| **or** | Bitwise OR | R | 0x00 | 0x25 |
| **ori** | Bitwise OR Immediate | I | 0x0D | NA |
| **sb** | Store Byte | I | 0x28 | NA |
| **sh** | Store Halfword | I | 0x29 | NA |
| **sll** | Logical Shift Left | R | 0x00 | 0x00 |
| **slt** | Set to 1 if Less Than | R | 0x00 | 0x2A |
| **slti** | Set to 1 if Less Than Immediate | I | 0x0A | NA |
| **sltiu** | Set to 1 if Less Than Unsigned Immediate | I | 0x0B | NA |
| **sltu** | Set to 1 if Less Than Unsigned | R | 0x00 | 0x2B |
| **sra** | Arithmetic Shift Right (sign-extended) | R | 0x00 | 0x03 |
| **srl** | Logical Shift Right (0-extended) | R | 0x00 | 0x02 |
| **sub** | Subtract | R | 0x00 | 0x22 |
| **subu** | Unsigned Subtract | R | 0x00 | 0x23 |
| **sw** | Store Word | I | 0x2B | NA |
| **xor** | Bitwise XOR (Exclusive-OR) | R | 0x00 | 0x26 |

# Regiștrii procesorului

Următorul table prezintă regiștrii disponibili procesorului MIPS32.

| **Register Number** | **Conventional Name** | **Usage** |
| --- | --- | --- |
| $0 | $zero | Hard-wired to 0 |
| $1 | $at | Reserved for pseudo-instructions |
| $2 - $3 | $v0, $v1 | Return values from functions |
| $4 - $7 | $a0 - $a3 | Arguments to functions - not preserved by subprograms |
| $8 - $15 | $t0 - $t7 | Temporary data, not preserved by subprograms |
| $16 - $23 | $s0 - $s7 | Saved registers, preserved by subprograms |
| $24 - $25 | $t8 - $t9 | More temporary registers, not preserved by subprograms |
| $26 - $27 | $k0 - $k1 | Reserved for kernel. Do not use. |
| $28 | $gp | Global Area Pointer (base of global data segment) |
| $29 | $sp | Stack Pointer |
| $30 | $fp | Frame Pointer |
| $31 | $ra | Return Address |
| $f0 - $f3 | - | Floating point return values |
| $f4 - $f10 | - | Temporary registers, not preserved by subprograms |
| $f12 - $f14 | - | First two arguments to subprograms, not preserved by subprograms |
| $f16 - $f18 | - | More temporary registers, not preserved by subprograms |
| $f20 - $f31 | - | Saved registers, preserved by subprograms |

# Tehnologiile utilizate și resurse necesare

1. Tehnologii utilizate:

* Limbaj de programare: C#.
* Mediu de dezvoltare: Visual Studio Enterprise 2016
* Controale suplimentare: RichTextBoxEx by Dino Chiesa

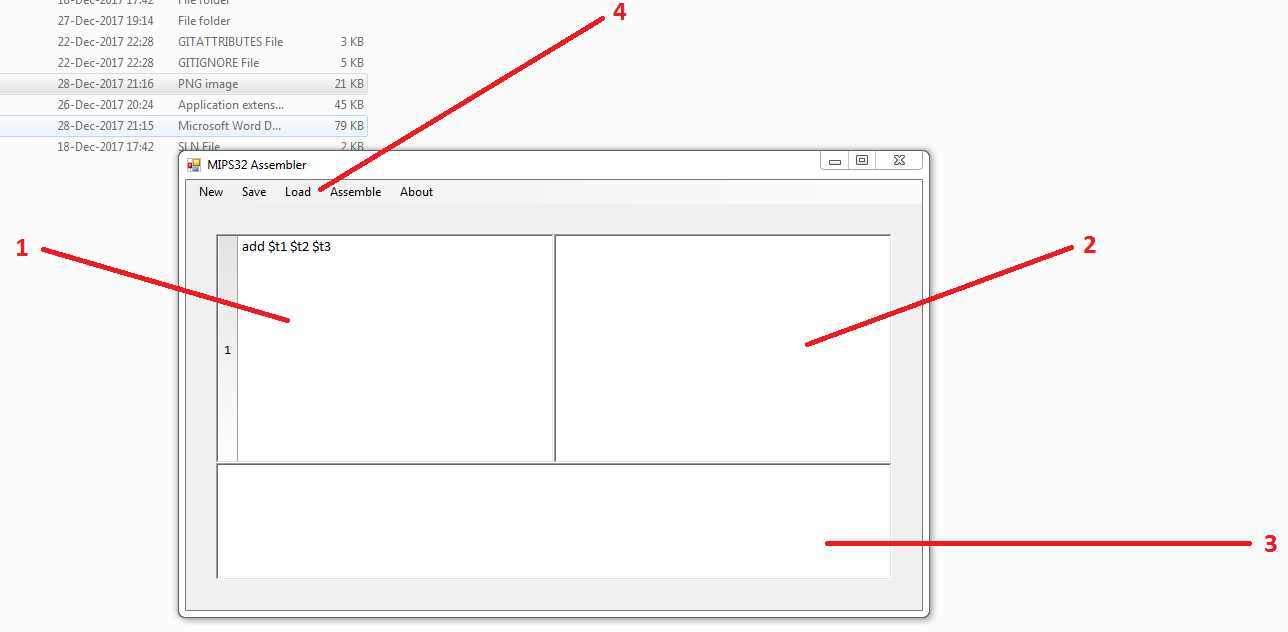
1. Resurse necesare rulării programului:

* .NET Framework 4.7.2
* CustomControls.dll distribuit cu programul
* Windows 7/8/8.1/10

# Prezentarea interfeței și instrucțiuni de utilizare

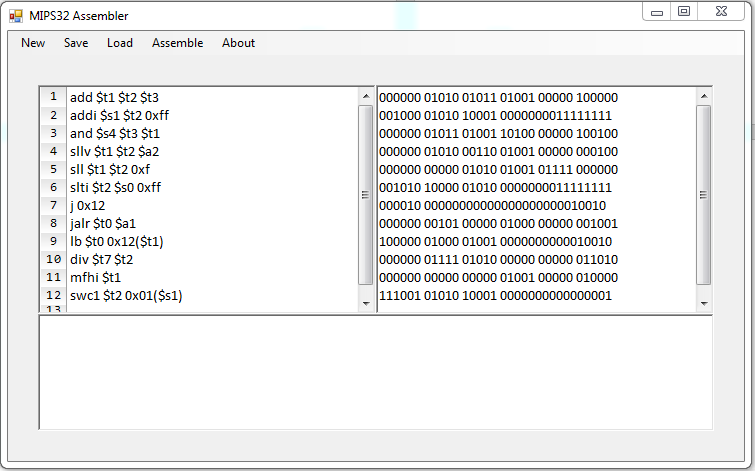
1. Interfața

Interfața programului a fost realizată într-un mod standardizat cu controale intuitive pentru o ușoară operare a programului.



1. Casetă pentru introducerea instrucțiunilor de către utilizator
2. Casetă de afișare a codului asamblat în limbaj mașină
3. Casetă de erori
4. Meniu principal:
   1. New – buton pentru crearea unui nou proiect
   2. Save – buton pentru salvarea în fișier a codului utilizator sau mașină
   3. Load – buton pentru încărcarea din fișier a codului utilizator sau a exemplelor ce vin o dată cu programul în sine
   4. Assemble – buton pentru asamblarea codului utilizator în cod mașină
   5. About – buton pentru afișarea informațiilor despre autor
5. Exemplu de utilizare

Se scriu în caseta utilizator instrucțiunile pentru procesor iar apoi se apasă butonul de asamblare. În caseta de cod mașină vor apărea instrucțiunile traduse în limbaj mașină.

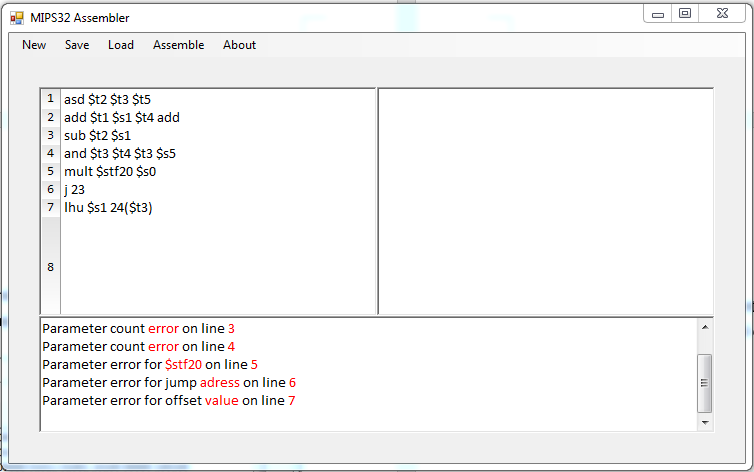


Detalii despre sintaxă:

* Numele instrucțiunilor trebuie scrise cu litere mici
* Numele regiștrilor trebuie prefixat cu „$”
* Adresele de salt si offset trebuie introduse in valori hexazecimale prefixate de „0x”
* Se permit spații multiple între parametrii instrucțiunii dar nu și între caracterele care alcătuiesc parametrul în sine

1. Exemplu de erori

În cazul în care în codul introdus există erori de sintaxă programul va avertiza utilizatorul cu privire la tipul erorii și linia la care s-a produs.



În imaginea de mai sus se pot observa câteva erori tipice:

* Nume greșit de instrucțiune pe linia 1
* Număr greșit de parametri pe linia 3
* Nume greșit de registru pe linia 5
* Valoare greșită pentru adresa de salt pe linia 6 și de offset pe linia 7

# Bibliografie

1. <http://eed.usv.ro/~zagan/teaching.php>
2. MIPS Architecture for Programmers Volume I-A, March 21 2011
3. <https://en.wikipedia.org/wiki/MIPS32>