**Documentație procesor MIPS32**

cu ciclu unic

**Arhitectura Calculatoarelor**

Student: Codorean Luca-Andrei, CTI-ro, 30223

Cadru didactic îndrumător: Nandra Constantin

*Aprilie, 2024*

Această documentație cuprinde elementele folosite în realizarea procesorului MIPS32 realizat în cadrul laboratorului de Arhitectura Calculatoarelor. Va fi descrisă fiecare pas de implementare, astfel încât înțelegerea acestuia să fie mult mai simplă. În implementarea procesorului este folosit un limbajul de descriere hardware, VHDL. Suportul hardware propriu-zis este o placă de dezvoltare Nexys A7, produsă de DIGILENT.

Cuprins

[Instrucțiunile procesorului 3](#_Toc164375050)

[Descrierea semnalelor de control pentru fiecare instrucțiune 4](#_Toc164375051)

[Codul conceput pentru rularea pe procesor 5](#_Toc164375052)

[Trasarea execuției programului de test pentru MIPS32 6](#_Toc164375053)

# Instrucțiunile procesorului

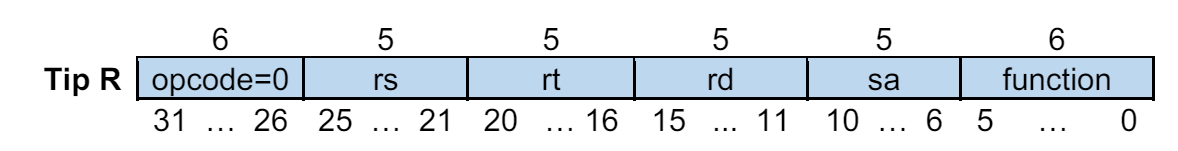
Procesorul implementat este capabil să realizeze trei tipuri de operații de bază, care permit conceperea de programe in limbaj de asamblare. Aceste instrucțiuni folosesc 32 de biți pentru implementare.

Un aspect care definitoriu pentru aceste instrucțiuni este opcode-ul, anume ultimii 6 biți, care definesc operația propriu-zisă. În cazul primei categorii de instrucțiuni descrise, anume R-Type, instrucțiunile au același opcode, însă ceea ce realizează diferența dintre ele sunt primii 6 biți, anume *function.*

Categoriile de instrucțiuni sunt:

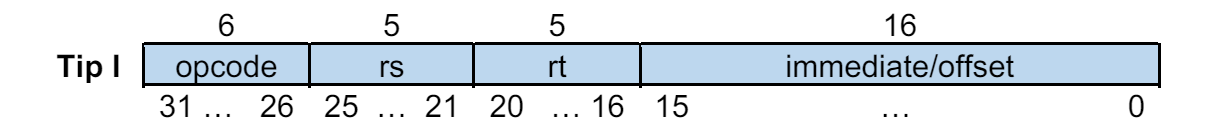
1. R-Type – din această categorie fac parte instrucțiunile care se realizează între regiștrii procesorului. Din această categorie fac parte instrucțiuni aritmetice de bază. Pe lângă acestea, pot să fie incluse și operații logice. În prezent, procesorul are implementate:
   * 1. Operații aritmetice: ADD, SUB, MOD; respectiv
     2. Operații logice: AND, OR, XOR.

Implementarea altor operații precum SLL sau SRL este o operațiune relativ simplă. Ele însă nu au fost implementate deoarece programul suport gândit să ruleze pe acest procesor nu le folosește. Structura unei asemenea instrucțiuni este descrisă de imaginea atașată.



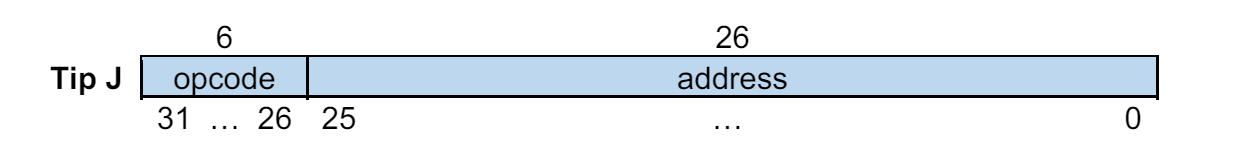
Figură 1. Formatul unei instrucțiuni de tip R.

1. Immediate – din această categorie fac parte instrucțiuni care realizează operații între regiștrii și o valoare imediată, care poate să fie interpretată de pildă ca un offset sau ca o valoare care urmează să fie încărcată într-un registru. Din această categorie fac parte:
   * 1. Operații aritmetice: ADDI, MODI;
     2. Instrucțiuni de salt: BEQ, BNEQ;
     3. Instrucțiuni de operare a memoriei: LW, SW.



Figură 2. Formatul unei instrucțiuni de tip I.

1. J – instrucțiuni care realizează salt la o anumită adresă. Din această categorie, în configurația actuală, se regăsește o singură instrucțiune, jump, care are formatul:



Figură 3. Formatul unei instrucțiuni de tip J.

O sumarizare a acestor instrucțiuni se poate observa în tabelul atașat.

Tabel 1. Distribuirea instrucțiunilor pe categorii.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Tip R** | |  | | --- | | Addition | | Subtraction | | Modulo | | Logical AND | | Logical OR | | Logical XOR |   Shift Logical Left  Shift Logical Right | |  | | --- | | ADD $DEST, $SRC, $TGT | | SUB $DEST, $SRC, $TGT | | MOD $DEST, $SRC, $TGT | | AND $DEST, $SRC, $TGT | | OR $DEST, $SRC, $TGT | | XOR $DEST, $SRC, $TGT  SLL $DEST, $TGT, HHHHH  SLR $DEST, $TGT, HHHHH | |
| Tip I | |  | | --- | | Load Word | | Store Word | | Branch on equal | | Branch not on equal | | Addition with immediate | | Modulo with immediate | | |  | | --- | | LW $TGT, OFFSET($SRC) | | SW $TGT, OFFSET($SRC) | | BEQ $SRC, $TGT, OFFSET | | BNEQ $SRC, $TGT, OFFSET | | ADDI $SRC, $TGT, IMM | | MODI $SRC, $TGT, IMM | |
| Tip J | Jump | J ADDR |

## Descrierea semnalelor de control pentru fiecare instrucțiune

Mai mult decât atât, fiecărei instrucțiunile implementate îi sunt atașate o colecție de flag-uri de control, care definesc comportamentul execuției pe componentele procesorului. Flag-urile sunt descrise în tabelul atașat ulterior.

Tabel 2. Descrierea semnalelor de control pentru fiecare instrucțiune

O imagine care conține text, număr, Paralel, joc cuvinte încrucișate

Descriere generată automat

Sub lista cu instrucțiunile și semnalele de control se poate observa un rând „FLAGS INDEX” care descrie poziția fiecărui semnal în vectorul de semnale.

# Codul conceput pentru rularea pe procesor

Codul gândit pentru rularea pe procesor își dorește să rezolve o problemă relativ simplă. Se dă un șir de 15 numere întregi care se pot scrie pe 32 de biți. Să se efectueze operația logică OR între suma totală a numerelor, respectiv suma numerelor care sunt multiplu de 3.

|  |
| --- |
| #include <stdio.h>  int main(void) {      int sumMul3 = 0, product = 0;  int array[15] = {73, 26, 8, 52, 96, 31, 38, 31, 25, 5, 43, 32, 13, 9, 22};      for (int i = 0; i < 15; i++) {          if (array[i] % 3 == 0) sumMul3 += array[i];          product += array[i];    }    printf("%d", product | sumMul3);    return 0;  } |

Figură 4. Rezolvarea problemei propuse în C

Figura de mai sus descrie implementarea problemei într-un limbaj de programare bine cunoscut, precum C. În continuare, vom scrie același cod, însă în limbaj de asamblare. În dreptul fiecărei linii, se va explica fiecare pas realizat și motivul adăugării instrucțiunii.

Tabel 3. Traducerea programului în limbaj de asamblare.

|  |  |  |
| --- | --- | --- |
| **Index** | **Instrucțiune** | **Descriere** |
| 0 | XOR $5, $5, $5 | *Se inițializează valoarea neutră pentru suma parțială.* |
| 1 | ADDI $6, $0, 0 | *Se inițializează valoarea neutră pentru suma totală.* |
| 2 | ADD $1, $0, $0 | *Se inițializează contorul pentru bucla.* |
| 3 | ADDI $4, $0, 15 | *Se încarcă limita pentru șirul de numere.* |
| 4 | XOR $2, $2, $2 | *Se inițializează indexul locației de memorie pentru elementul curent din șir.* |
| 5 | XOR $7, $7, $7 | *Se inițializează un registru auxiliar, folosit pentru a stoca valori temporare.* |
| 6 | BEQ $4, $1, 8 | Se verifică dacă indexul curent a ajuns la limita șirului, și dacă da, ieși din buclă. |
| 7 | LW $3, 0($2) | *Se încarcă în $3 elementul de la indexul din $2.* |
| 8 | MODI $3, $7, 3 | *Se încarcă în registrul auxiliar valoarea șir[$2] % 3* |
| 9 | BNEQ $7, $0, 1 | *Dacă în registrul $7 nu se află valoarea 0, atunci nu adăuga la suma parțială* |
| 10 | ADD $5, $3, $5 | Se adăugă la suma parțială valoarea din $3. |
| 11 | ADD $6, $3, $6 | Se adăugă la suma totală valoarea din $3. |
| 12 | ADDI $2, $2, 4 | Se incrementează indexului pentru șir. |
| 13 | ADDI $1, $1, 1 | Se incrementează indexului care ține contorul curent. |
| 14 | J 6 | Se face salt la începutul buclei. |
| 15 | XOR $7, $7, $7 | Se resetează registrul auxiliar, pentru a putea fi folosit într-o nouă operație. |
| 16 | OR $7, $6, $5 | Se realizează operația propriu-zisă de OR, și se salvează rezultatul în registrul $7. |
| 17 | SW $7, 60($0) | Valoarea din registrul $7 se scrie în memorie. |

În continuare, vom traduce fiecare instrucțiune din limbajul de asamblare, în cod mașină, astfel încât procesorul să poată să realizeze manipularea acestora.

Tabel 4. Trecerea de la limbaj de asamblare la cod mașină.

|  |  |  |  |
| --- | --- | --- | --- |
| **Index** | **Instrucțiune în asamblare** | **Instrucțiune în cod mașină** | **Valoarea hexa-zecimală a instrucțiunii** |
| 0 | XOR $5, $5, $5 | 000000\_00101\_00101\_00101\_00000\_000001 | 0x00A52801 |
| 1 | ADDI $6, $0, 0 | 000001\_00110\_00000\_\_\_0000000000000000 | 0x04C00000 |
| 2 | ADD $1, $0, $0 | 000000\_00001\_00001\_00001\_00000\_000001 | 0x00210801 |
| 3 | ADDI $4, $0, 15 | 000001\_00100\_00000\_\_\_0000000000001111 | 0x0480000F |
| 4 | XOR $2, $2, $2 | 000000\_00010\_00010\_00010\_00000\_000001 | 0x00421001 |
| 5 | XOR $7, $7, $7 | 000000\_00111\_00111\_00111\_00000\_000001 | 0x00E73801 |
| 6 | BEQ $4, $1, 8 | 001000\_00001\_00100\_\_\_0000000000001000 | 0x20240008 |
| 7 | LW $3, 0($2) | 100000\_00010\_00011\_\_\_0000000000000000 | 0x80430028 |
| 8 | MODI $3, $7, 3 | 000010\_00111\_00011\_\_\_0000000000000011 | 0x08E30003 |
| 9 | BNEQ $7, $0, 1 | 001100\_00000\_00111\_\_\_0000000000000001 | 0x30070001 |
| 10 | ADD $5, $3, $5 | 000000\_00101\_00011\_00101\_00000\_100000 | 0x00A32820 |
| 11 | ADD $6, $3, $6 | 000000\_00110\_00011\_00110\_00000\_100000 | 0x00C33020 |
| 12 | ADDI $2, $2, 4 | 000001\_00010\_00010\_\_\_0000000000000100 | 0x04420004 |
| 13 | ADDI $1, $1, 1 | 000001\_00001\_00001\_\_\_0000000000000001 | 0x04210001 |
| 14 | J 6 | 111111\_\_\_\_\_00000000000000000000000110 | 0xFC000006 |
| 15 | XOR $7, $7, $7 | 000000\_00111\_00111\_00111\_00000\_000001 | 0x00E73801 |
| 16 | OR $7, $6, $5 | 000000\_00101\_00110\_00111\_00000\_111100 | 0x00A6383C |
| 17 | SW $7, 60($0) | 110000\_00111\_00000\_\_\_0000000001010110 | 0xC0E0003C |

### Trasarea execuției programului de test pentru MIPS32

Pentru a observa flow-ul datelor în instrucțiunile programului, a fost conceput tabelul de trasare a execuției. Acest tabel prezintă modul în care datele sunt preluate de instrucțiuni, și ulterior manipulate. De asemenea, pentru a observa comportamentul procesorului în întregime, tabelul descrie ieșirile din fiecare componentă principală a acestuia.

Pe tabel există câteva celule care sunt colorate, care scot în evidență anumite valori „cheie” din rulare. Se poate observa că bucla se rulează o singură dată, astfel că, în instrucțiunile de după jump-ul din linia 16, valoarea cu care se lucrează este valoarea de la prima adresă din șirul de numere. În mod normal, după o rulare completă, cele două valori ar *putea* să fie diferite.

Pe exemplul dat, instrucțiunea din lina 12 nu este rulată, întrucât codul nu ajunge în acel punct din cauza rezultatului operației de *Branch not on equal* din linia 11.

Tabel 5. Trasarea execuției programului de test pentru MIPS32

