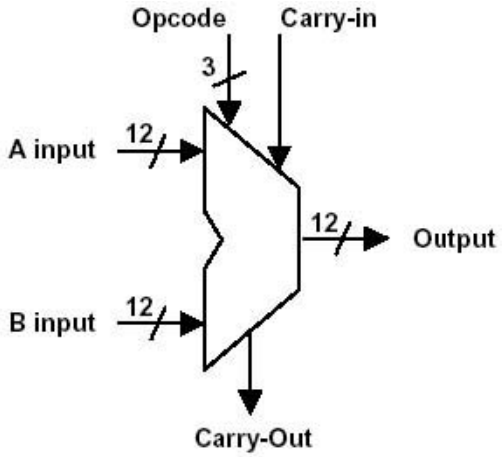
Laboratorul 8 - Unitatea aritmetică logică

1. Scopul laboratorului

În acest laborator vom proiecta o unitate fundamentală a oricarui procesor: Unitatea Aritmetică Logică (UAL, în literatura de specialitate Arithmetic Logic Unit sau ALU).



Unitatea aritmetică logică se ocupă de aproape toate calculele numerice de care are nevoie un procesor. În ciclul de prelucrare a instrucţiunilor, în diverse etape procesorul are nevoie de rezultatele unor calcule, fie că sunt solicitate de o intrucţiune explicită cum este *add* din limbajul de asamblare, fie dintr-un motiv intern (de exemplu adresa absolută a unei date se calculează în funcţie de *segment* şi *offset*, ceea ce implică nevoia unor calcule).

2. Cu ce se ocupă o UAL?

O UAL poate fi proiectată să execute, în principiu, orice operație. Totuși, cu cât operațiile devin mai complexe UAL devine mai scumpă, ocupă mai mult loc și disipă mai multă căldură. Operațiile care sunt, în general, suportate de toate UAL sunt:

- Operații logice: AND, OR, NOT, XOR, NOR, NAND, etc.
- Operații de shift: shift stânga, shift dreapta, shift circular, etc.
- Operații aritmetice: adunare, scădere, înmulțire (nu toate), împărțire (nu toate).

Există întotdeauna un compromis pe care inginerii trebuie să îl facă la proiectarea unei UAL. Ideea este că o instrucţiune poate:

- să fie executată *într-un singur ciclu de ceas*, ceea ce ar fi *rapid*, dar foarte *costisitor* din punctul de vedere al circuitului fizic (complex), cât şi din alte considerente (de exemplu, disiparea de căldură),
- să fie executată *în mai mulţi paşi*, ceea ce e evident mai *lent*, dar *uşor de proiectat şi de adăugat funcţionalităţi unui acelaşi circuit*,
- să nu fie executată ca o operaţie de sine stătătoare în UAL, ci implementată prin software pe baza operaţiilor deja suportate de UAL, ceea ce ar fi *foarte lent*, dar dă *o oarecare flexibilitate* celor care scriu programe la nivel mai înalt.

Procesoarele moderne folosesc întotdeauna prima variantă pentru instrucţiuni simple şi diverse implementări ale variantei a doua pentru operaţii de complexitate medie şi ridicată.

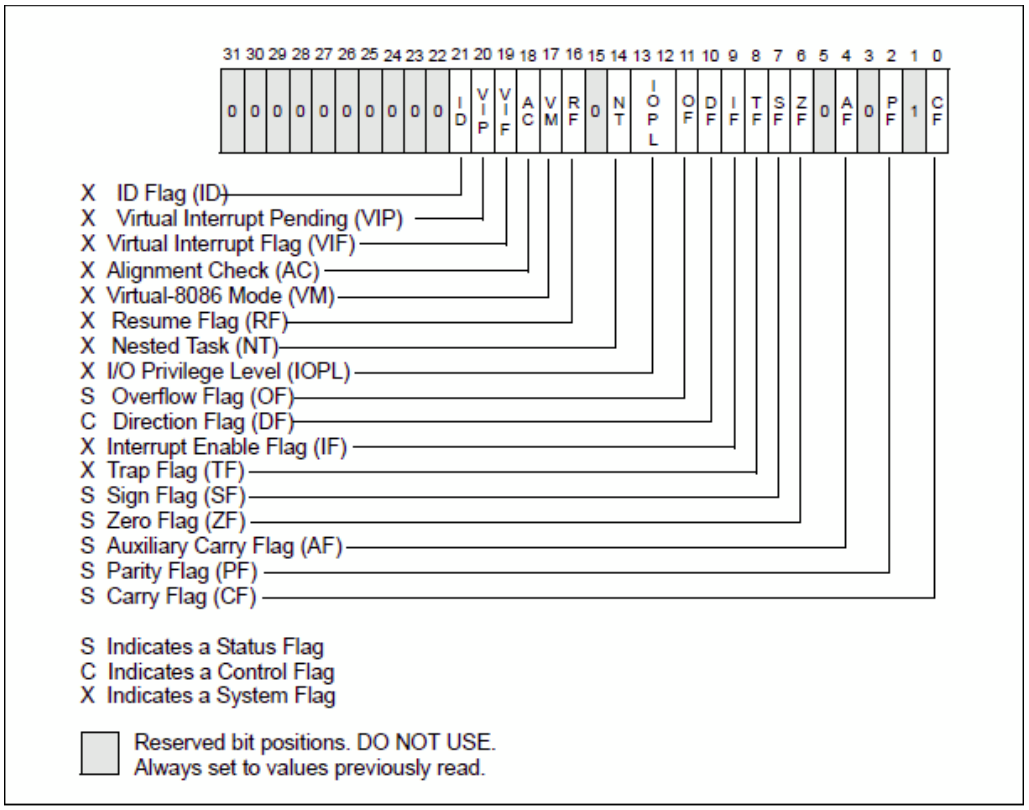
De asemenea, o UAL trebuie să fie în stare la orice moment să ruleze operaţia corespunzătoare la fiecare comandă din partea procesorului. Pentru aceasta fiecare operație are un *cod al operației* implementat în hardware, astfel încât la o instrucţiune *add* să intre în funcţiune modulul pentru adunare, la o instrucţiune *and* să intre în funcţiune poarta AND, etc.

3. Status Register

În urma calculelor făcute de UAL pot apărea anumite situații, precum overflow la adunare/scădere, care este util să fie semnalate în cadrul sistemului. În acest scop procesoarele au un **registru de status** în care prin setarea unui bit anume este semnalat un anumit eveniment. Cel mai adesea, informațiile din registrul de status (sau **Status Register**) sunt folosite pentru a modifica fluxul de execuție al programului pe baza instrucțiunilor condiționale.

Spre exemplu, procesorul calculează A + B și dorește să sară la o anumită etichetă dacă rezultatul este 0. UAL va calcula rezultatul lui A + B și va seta bitul **Z (Zero) din Status Register** dacă rezultatul adunării este 0. Instrucțiunea de salt **JZ (jump zero)** verifică bitul Z din Status Register și sare la eticheta data dacă bitul este 1, altfel execută instrucțiunea imediat următoare.

În cadrul **x86** Status Register se numește **FLAGS** și are 16 biți. El are o extensie de 32 de biți numită **EFLAGS**. Biții din cele două registre



EFLAGS Register

au următoarea semnificație:

Un exemplu mai simplu este registrul de status pentru **Atmega324** (AVR în general), care se numește **SREG** și are 8 biți:

SREG – Status Register

The AVR Status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

În ambele registre de status prezentate mai sus se regăsesc o serie de **biți de interes pentru UAL**:

- **CF sau C (carry flag)** este setat pe 1 când operația executată pe UAL are carry.
- **ZF sau Z (zero flag)** este setat pe 1 când rezultatul operației de pe UAL este 0.
- **N (negative)** este setat pe 1 când rezultatul operației de pe UAL este un număr negativ (bitul de semn al rezultatului este 1).
- **OF sau V (overflow flag)** este setat pe 1 când operația executată pe UAL produce overflow, mai exact există 2 cazuri:
  - dacă adunăm 2 numere pozitive și bitul de semn al rezultatului este 1 (rezultatul este număr negativ)
  - dacă adunăm 2 numere negative și bitul de semn al rezultatului este 0 (rezultatul este număr pozitiv)

4. Algoritmul lui Booth

Înmulțirea a două numere cu semn pe UAL este o operație complexă și costisitoare. Ținând cont că operațiile de shift sunt mai rapide ca cele de adunare, **Andrew Booth** a venit cu o propunere de algoritm pentru înmulțirea numerelor cu semn care acum îi poartă numele.

4.1. Prezentarea algoritmului

Dorim să calculăm produsul **P = M x R** unde

- **M, R** sunt numere cu semn reprezentate pe **n biți**
- **P** este un număr cu semn reprezentat pe **2 * n** biți

1. Formăm P

- P = biții cei mai nesemnificativi (din dreapta) ai lui P vor lua valoarea lui R, restul vor fi 0.
- Adăugăm un bit de '0' în dreapta lui P (LSB). Numim acest bit Z.

P are acum 2 * n + 1 biți. El va fi folosit astfel în cadrul algoritmului, iar bitul extra va fi eliminat la sfârșit pentru a obține rezultatul final.

2. Determinăm două valori auxiliare A și S

Valorile auxiliare A și S vor fi adunate lui P în cadrul algoritmului. Ele vor avea tot 2 * n + 1 biți.

- A = biții cei mai semnificativi (din stânga) vor lua valoarea lui M, restul vor fi 0.
- S = biții cei mai semnificativi (din stânga) vor lua valoarea lui -M, restul vor fi 0.

3. Verificăm primii 2 biți ai lui P

Verificăm primii 2 biți ai lui P (cei mai nesemnificativi biți):

1. dacă sunt 01 ⇒ P = P + A (ignorăm overflow)
2. dacă sunt 10 ⇒ P = P + S (ignorăm overflow)
3. dacă sunt 00 ⇒ nu facem nimic
4. dacă sunt 11 ⇒ nu facem nimic

4. Shiftam aritmetic la dreaptă pe P cu o poziţie

Atenție la Arithemtic Shift [https://en.wikipedia.org/wiki/Arithmetic_shift] vs Logical Shift [https://en.wikipedia.org/wiki/Logical_shift].

5. Repetăm pașii 3 și 4 de n ori

6. Eliminăm bitul Z

Eliminăm bitul Z, adică cel mai nesemnificativ bit al lui P. Valoarea din P este rezultatul înmulțirii.

4.2. Exemplu

Fie M = 0011 (3) (-M = 1101 (-3)) și R = 1100 (-4).

Pasul 1

Îl construim pe P prin concatenarea valorilor R și Z. Restul biților vor fi padding cu 0. Practic adunăm valoarea 2R la P.

```
P = {(0000), (R), (Z)} = 0000 1100 0
```

Pasul 2

Formăm A și S.

```
A = {(M), (0000), (Z)} = 0011 0000 0
S = {(-M), (0000), (Z)} = 1101 0000 0
```

Pașii 3 și 4

Pașii 3 și 4 vor fi efectuați de **n = 4** ori:

1. P = 0000 110***0 0***
   - Ultimii 2 biți sunt 00 (ne aflăm într-o secvența continuă de biți de 0) ⇒ Nu modificăm P
   - Îl shiftam aritmetic la dreapta pe P ⇒ P = 0000 0110 0
2. P = 0000 011***0 0***
   - Ultimii 2 biți sunt 00 ⇒ Nu modificăm P
   - Îl shiftam aritmetic la dreapta pe P ⇒ P = 0000 0011 0
3. P = 0000 001***1 0***
   - Ultimii 2 biți sunt 10 (am identificat începutul unei secvențe continue de biți de 1) ⇒ P = P + S = 1101 0011 0
   - Îl shiftam aritmetic la dreapta pe P ⇒ P = 1110 1001 1
4. P = 1110 100***1 1***
   - Ultimii 2 biți sunt 11 (suntem în interiorul unei secvențe continue de biți de 1) ⇒ Nu modificăm P
   - Îl shiftam aritmetic la dreapta pe P ⇒ P = 1111 0100 1

Pasul 5

Eliminăm bitul Z, iar P = 1111 0100 (-12) este rezultatul.

4.3. De ce merge algoritmul / Ce face?

Ideea de la care pleacă algoritmul este că orice secvența continuă de 1 dintr-un număr binar poate fi rescrisă ca diferența a două numere binare:

$$(\ldots 0\overbrace{1\ldots 1}^{n}0\ldots)_2 \equiv (\ldots 1\overbrace{0\ldots 0}^{n}0\ldots)_2 - (\ldots 0\overbrace{0\ldots 1}^{n}0\ldots)_2.$$

În cazul înmulțirii a două numere, următoarele expresii sunt echivalente:

```
P = M x 10011 =  M x (2^4 + 2^1 + 2^0) =  M x (2^5-2^4 + 2^2-2^0)
```

Practic, algoritmul lui Booth se reduce la a identifica secvențele continue de biți de 1 din termenul R pentru P = M x R și a le înlocui cu o diferență. Astfel pentru o operație de înmulțire, numărul adunărilor este redus, realizându-se în schimb mai multe shiftari.

Grafic, algoritmul face următoarea transformare:

Example 2 x 6 = 0010 x 0110:

```
    0010
  x 0110
   + 0000 shift (0 in multiplier)
  + 0010  add   (1 in multiplier)
 + 0010   add   (1 in multiplier)
+ 0000    shift (0 in multiplier)
 00001100
```

=>

```
    0010
  x 0110
   + 0000 shift (0 in multiplier)
  - 0010  sub   (first 1 in multiplier)
 + 0000   shift (middle of string of 1s)
+ 0010    add   (prior step had last 1)
 0001100
```

Tabelul de mai jos surprinde motivul pentru care algoritmul verifică biții LSB ai lui P:

- Începutul unei secvențe înseamnă că trebuie să scădem -M * 2^i, unde i e indexul bitului de început secvenței de 1
- Sfârșitul unei secvențe înseamnă că trebuie să adunăm M * 2^j, unde j e indexul bitului de sfârșit secvenței de 1

Bitul curent	Bitul din dreapta	Explicație	Exemplu
1	0	Începutul unei secvențe continue de 1	000111**10**00
1	1	Interiorul unei secvențe continue de 1	00011**11**000
0	1	Sfârșitul unei secvențe continue de 1	00**01**111000
0	0	Interiorul unei secvențe continue de 0	**00**01111000

5. TL;DR

- **Unitatea Aritmetică Logică** se ocupă de aproape toate calculele cerute de procesor, solicitate de o instrucțiune explicită sau necesară intern.
- Instrucțiuni uzuale:
  - **Operații Logice**: `AND`, `OR`, `NOT`, `XOR`, `NOR`, `NAND`, etc.
  - **Operații de shift**: `shift stânga`, `shift dreapta`, `shift circular`, etc.
  - **Operații aritmetice**: `ADD`, `SUB`, `MUL` (nu toate), `DIV` (nu toate).
- Există un compromis între *viteză* și *cost (complexitate hardware)*
- **Status Register**: Registru de flag-uri: `'ZF/Z` - Zero, `CF/C` - Carry, `N` - Negative, `OF/V` - Overflow
  - Setate de UAL în momentul producerii unor evenimente.
- **Algoritmul lui Booth** este o metodă eficientă de a înmulți două numere cu semn, care folosește operații de `shift`

6. Exerciții

Va trebui sa faceti atat **implementarea** cat si **simularea** cu seturi de date relevante.

1. **(10p)** Implementați un modul UAL cu intrări pe 4 biți. Modulul va primi la intrare 2 numere (A și B) și indicatorul unei operații ce se va efectua asupra numerelor. Ieșirea modulului va fi un număr pe 8 biți ce reprezintă rezultatul aplicării operației asupra numerelor A și B.

- **(1p)** `NAND`
- **(1p)** `XOR`
- **(2p)** `ADD` (carry look-ahead)
- **(2p)** `SUB` (folosind operatia ADD)
- **(4p)** `MUL` folosind algoritmul lui Booth

În Verilog folosim operatorii **<<<** și **>>>** pentru shiftări aritmetice. Valoarea bitului de fill se determină pe baza contextului (signed sau unsigned). Din acest motiv, pentru a shifta aritmetic la dreapta un număr negativ va trebui să specificăm contextul `signed` al valorii shiftate.

```
assign magic_shift = 4'b1010;
assign logic_right_shift = (magic_shift >> 1); // 4'0101
assign logic_left_shift = (magic_shift << 1); // 4'b0100
assign arithmetic_right_shift = (magic_shift >>> 1); // 4'b0101 <- greșit pentru numere cu semn
assign arithmetic_left_shift = (magic_shift <<< 1); // 4'0100
assign signed_arithmetic_right_shift = ($signed(magic_shift) >>> 1); // 4'b1101 <- corect, semnul se păstrează
assign signed_arithmetic_left_shift = ($signed(magic_shift) <<< 1); // 4'b0100 <- nu e relevant aici
```

În implementarea algoritmului lui Booth aveți nevoie de:

```
assign signed_arithmetic_right_shift = ($signed(magic_shift) >>> 1); // 4'b1101 <- corect, semnul se păstrează
```

În acest laborator aveți un checker util pentru ambele exerciții. Acesta are hardcodate operanzi și rezultatele unor operații, iar rolul lui este de a vă ajuta pentru testare. Pentru a-l folosi, tratați modulul test_ual.v drept unul de simulare. Rezultatele vor fi afișate in consolă, sub zona de semnale.

2. **(2p)** Implementați un mini status register pe 2 biți pentru operațiile NAND, XOR, ADD, SUB.

- Veți avea 2 flag-uri: OF (Overflow flag), Zero Flag (ZF).
- Aveți definite constante in fișierul defines.vh, pe care le puteți folosi pentru a nu hardcoda valori.

7. Resurse

* Scheletul de laborator

8. Linkuri utile

- Booth algorithm Wiki [https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm]
- Booth algorithm [http://www.vlsiip.com/download/booth.pdf]
- Exemplu Booth [http://www.massey.ac.nz/~mjjohnso/notes/59304/l5.html]
- Logical Shift [https://en.wikipedia.org/wiki/Logical_shift]
- Arithmetic Shift [https://en.wikipedia.org/wiki/Arithmetic_shift]

cn1/laboratoare/08.txt · Last modified: 2021/05/10 20:09 by adina.smeu