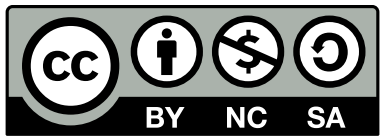
0010 - Arithmétique

Nombres, bits, additions, débordements

Jean Privat

Université du Québec à Montréal

INF2171 - Organisation des ordinateurs et assembleur
v241



Rappel

Unité centrale de traitement (UCT, *CPU*)

- Ont des **registres** qui contiennent de l'information
- Exécutent des **instructions** machines qui font des **calculs** avec ces registres

Codage de l'information

Code

- Règle de transcription de l'**information** d'un système à un autre
- En informatique: théorie des codes
  (branche de la théorie de l'information)

En INF2171

- **représentation** d'une information en un **paquet de bits**

Codage et décodage

- Transcription d'une information d'un système à l'autre

En INF2171

- Déterminer le paquet de bits qui représente une information
- Déterminer l'information représentée par un paquet de bits

Important : un paquet de bits n'a pas de signification intrinsèque

Plan

Système de numération

Système de numération

Qu'est-ce qu'un nombre ?

- Une quantité
- La mesure de quelque chose

Sa représentation ?

- $1101_{(2)}$
- `0b1101`
- $15_{(8)}$
- $\mathrm{D}_{(16)}$
- `0x0D`
- $13_{(10)}$
- XIII
- « treize »

Représentation des nombres

Les chiffres

- Les briques de bases pour représenter un nombre

Système décimal

- Il n'est pas plus « vrai » que les autres systèmes
- Mais c'est celui dont on a l'habitude

Notation positionnelle

Les puissance de la base du système

- Base 10 : 1 (un), 10 (dix), 100 (cent), 1000 (mille), etc.
- Base 2 : 1 (un), 2 (deux), 4 (quatre), 8 (huit), etc.

Exemple

$$1\,1\,0_{(10)} = 1 \times 10^2 + 1 \times 10^1 + 0 \times 10^0$$

$$1\,1\,0_{(2)} = 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6_{(10)}$$

Questions

- $110_{(8)}$? $110_{(13)}$? $110_{(64)}$?

Poids faible et fort

Les **positions** des chiffres d'un nombre ont un **impact** différent sur la valeur du nombre

- Poids **faible**: la plus petite puissance de la base
  À droite
  (bleu dans l'exemple)
- Poids **fort**: la plus grande puissance de la base
  À gauche
  (vert dans l'exemple)

Système binaire (base 2)

2 chiffres

- 0 et 1
- « bit » = *binary digit* (chiffre binaire)
- Notation: $1101_{(2)}$ ou `0b1101` (**b**inaire)

Exercices: estimez puis calculer

- $1101_{(2)} = ?$
- $10000100110100_{(2)} = ?$

Décimal $\rightarrow$ binaire

Principe

- Suite de divisions entières par 2
- On conserve les restes (en ordre inverse)

Exemple

$19/2 = 9$ reste 1

$9/2 = 4$ reste 1

$4/2 = 2$ reste 0

$2/2 = 1$ reste 0

$1/2 = 0$ reste 1

0

donc $19_{(10)} = 1\ 0\ 0\ 1\ 1_{(2)}$

Décimal $\rightarrow$ binaire (alternative)

Principe alternatif

- Suite de soustractions des puissances de 2
- On soustrait les puissances fortes en premier

Exemple

$19_{10} \geq 16$? oui on note 1, il reste 3

$3_{10} \geq 8$? non on note 0, il reste 3

$3_{10} \geq 4$? non on note 0, il reste 3

$3_{10} \geq 2$? oui on note 1, il reste 1

$1_{10} \geq 1$? oui on note 1, il reste 0

donc $19_{(10)} = 1\,0\,0\,1\,1_{(2)}$

Système hexadécimal

Le binaire (base 2), c'est bien

- 2 chiffres : 0 et 1
- Représentation électronique de la machine
- Mais trop de chiffres dans les nombres
  2147 $\rightarrow$ `0b100001100011`

L'hexadécimal (base 16), c'est mieux

- 16 chiffres : 0 à 9 et A, B, C, D, E et F (ou minuscules)
- Notations: $3\mathrm{E}8_{(16)}$ ou `0x3E8` (he**x**adécimal), voire `3E8h`
- Moins de chiffres dans les nombres
  2247 $\rightarrow$ `0x8C7`
- Passages faciles binaire $\rightarrow$ hexadécimal
  et hexadécimal $\rightarrow$ binaire

Hexadécimal $\leftrightarrow$ binaire

Principe

- On part de la droite
- 4 bits $\leftrightarrow$ 1 chiffre hexadécimal
- On utilise une table pour trouver l'équivalent

Exemple

- $100011_{(2)} \leftrightarrow$ «0010 0011» $\leftrightarrow$ «2 3» $\leftrightarrow 23_{(16)}$

Exercices

- `0x20D` ?
- `0xFFF` ?

Octal

- Base 8: chiffres de 0 à 7
- Concurrent de l'hexadécimal
  - Pas de lettres dans les nombres
  - 3 bits pour un chiffre octal
  - Mais 3 chiffres octaux $\rightarrow$ 9 bits (plus qu'un octet)

De moins en moins utilisé

Des artefacts historiques restent

- Droits des fichiers Unix (`chmod`)
- Littéraux octaux (C, C++, Java, etc.) commencent par `0`
  - `0123` $\rightarrow$ 83
  - Préférez le préfixe `0o` (si permis): `0o123` $\rightarrow$ 83
- Séquence d'échappement dans les chaines `'\123'` $\rightarrow$ `'S'`
- Outil POSIX `od` (*octal dump*)

Énigme

- Pourquoi les informaticiens confondent Halloween et Noël?

Préfixes d'unités

Normes ISO et CEI

kilo (k)	10^3	kibi (Ki)	$2^{10} = 1\,024$
méga (M)	10^6	mébi (Mi)	$2^{20} = 1\,048\,576$
giga (G)	10^9	gibi (Gi)	$2^{30} = 1\,073\,741\,824$
téra (T)	10^{12}	tébi (Ti)	$2^{40} = 1\,099\,511\,627\,776$
péta (P)	10^{15}	pébi (Pi)	$2^{50} = 1\,125\,899\,906\,842\,624$
exa (E)	10^{18}	exbi (Ei)	$2^{60} = 1\,152\,921\,504\,606\,846\,976$
mili (m)	10^{-3}		
micro (μ)	10^{-6}		
nano (n)	10^{-9}		
pico (p)	10^{-12}		

Bits, octets et mots

Représentation de l'information

Les ordinateurs actuels sont binaires

- Le bit est la plus petite **unité d'information**
- Facilement représentable électroniquement, magnétiquement, optiquement, mécaniquement, etc.
- Les bits sont partout: dans les registre de l'UCT, en mémoire (RAM, ROM), sur disque, dans les signaux réseau, etc.

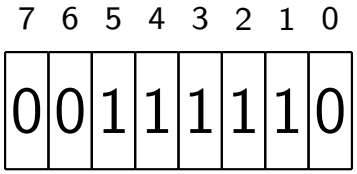
Quantité d'information

informations $\approx$ **possibilités** $\approx$ **choix**

- 1 bit $\rightarrow$ 2 informations
- 2 bits $\rightarrow$ 4 informations
- 3 bits $\rightarrow$ 8 informations
- n bits $\rightarrow$ 2^n informations

Octet

- Groupe de 8 bits (*byte*) numérotés de 0 à 7
- La plus petite unité d'information **adressable** en mémoire
- $2^8 = 256$ combinaisons de 8 bits différentes
- On dessine les bits de poids forts à gauche



Boîtes et numérotation optionnelles

- Écrire juste les huit bits, du fort au faible
- Même ceux à 0: un octet c'est exactement 8 bits
- Exemple « `00111110` »

Écriture hexadécimale

- Écrire exactement 2 chiffres hexadécimaux
- Exemple « `3E` »

L'hexadécimal s'utilise pour représenter octets et paquets d'octets

Mot

Pour une **famille** d'architectures donnée:

- la taille historique des registres
- et/ou la largeur historique des bus de données

Historique: terminologie maintenue même si l'architecture change

Exemples

- En RISC-V (et ARM), un mot = 4 octets (32 bits)
- En x86 (et Pep/8), un mot = 2 octets (16 bits)

Autres longueurs fixes

Chez RISC-V

- Octet (byte, b): 8 bits
- Demi-mot (*halfword*, h): 16 bits, 2 octets
- Mot (*word*, w), 32 bits, 4 octets
- Double-mot (*doubleword*, d) 64 bits, 8 octets
  Disponible en RV64I
- *Quadword* (q) 128 bits, 16 octets
  Disponible en RV128I

Attention

- Ne pas confondre *double-mot* et *flottant double précision* (type `double` en Java par exemple)
- On verra les flottants plus tard

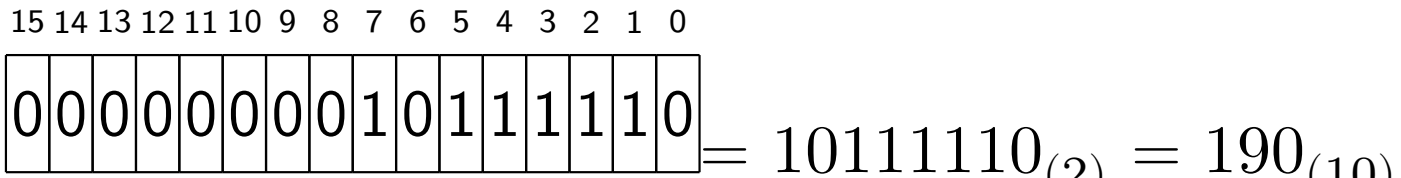
Codage des entiers

Codage des nombres entiers positifs

- Entiers **positifs** $=$ entiers **non signés** (*unsigned*)
- On a n bits (numérotés de 0 à $n-1$)
  - Assez de bits pour représenter 2^n éléments distincts
  - Donc, les entiers positifs de 0 à $2^n - 1$

Entiers non signés sur 16 bits

- de 0 à $2^{16} - 1$
- soit de 0 à $65\,535$

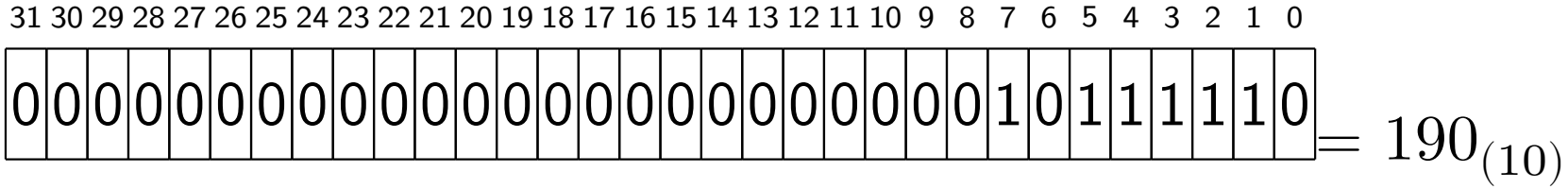


$= 10111110_{(2)} = 190_{(10)}$

Codage des nombres entiers non signés

Entiers non signés sur 32 bits

- de 0 à $2^{32} - 1$
- soit de 0 à $4\,294\,967\,295$



Question: Entiers non signés sur 64 bits

- Entier minimum ?
- Entier maximum ?
- Codage de 190 ?

Codage des entiers signés (*signed*)

- Nombres entiers **relatifs** $=$ nombres entiers **signés**
- On a n bits (numérotés de 0 à $n-1$)
- Assez de bits pour représenter 2^n entiers
- Mais on veut des entiers positifs et des négatifs
- On a besoin d'un codage spécifique:
- Codage en **complément à deux**

Complément à deux

Analogie: décrémenter un compteur kilométrique

- Odomètre avec 4 rouleaux (numérotés 0, 1, 2 et 3)
- Chaque rouleau va de 0 à 9

3 2 1 0

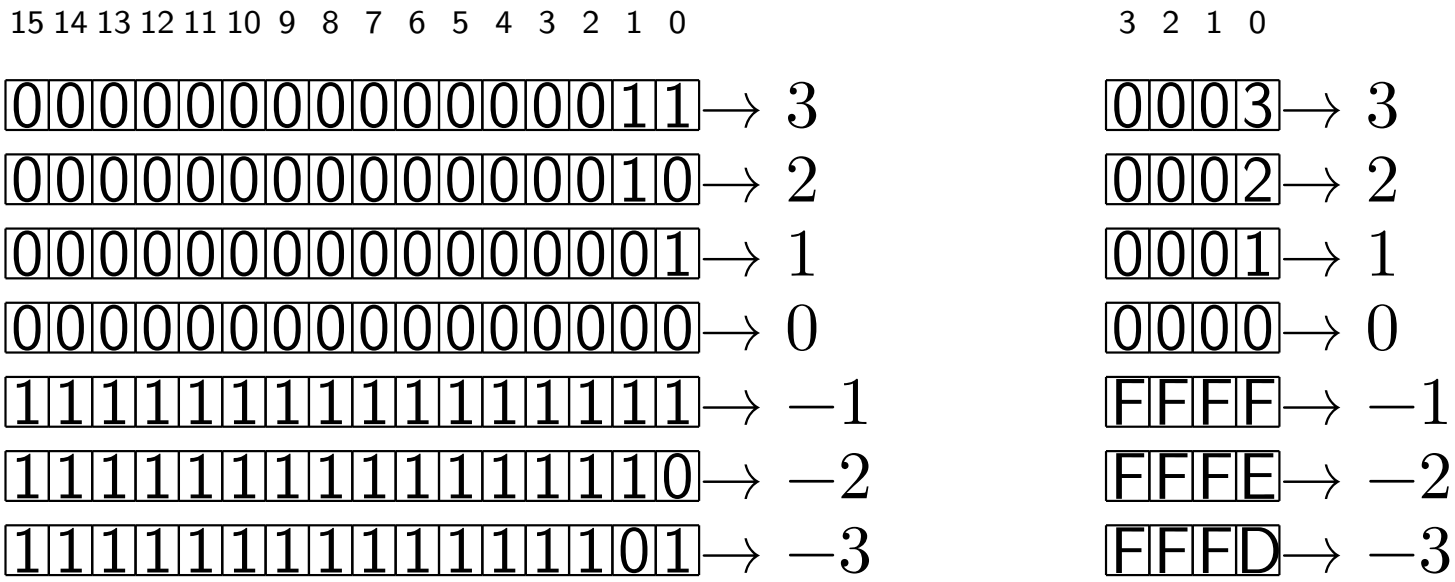
$$\boxed{0}\boxed{0}\boxed{0}\boxed{3} \rightarrow 3$$
$$\boxed{0}\boxed{0}\boxed{0}\boxed{2} \rightarrow 2$$
$$\boxed{0}\boxed{0}\boxed{0}\boxed{1} \rightarrow 1$$
$$\boxed{0}\boxed{0}\boxed{0}\boxed{0} \rightarrow 0$$
$$\boxed{9}\boxed{9}\boxed{9}\boxed{9} \rightarrow -1$$
$$\boxed{9}\boxed{9}\boxed{9}\boxed{8} \rightarrow -2$$
$$\boxed{9}\boxed{9}\boxed{9}\boxed{7} \rightarrow -3$$

Le complément d'un nombre

- Le nombre de l'autre coté du 0

Exemple: binaire signé 16bits

- 16 « rouleaux » de 0 à 1
- 4 « rouleaux » de 0 à F



- Question: comment coder -5?

Découpage de l'espace

- Rappel: n bits codent au maximum 2^n informations
- On découpe l'espace en deux (moitié-moitié)
  - Bit de poids fort $= 0 \rightarrow$ positif (ou nul)
  - Bit de poids fort $= 1 \rightarrow$ négatif
  - Soit, de -2^{n-1} à $2^{n-1} - 1$
  - Asymétrique: un seul zéro, un négatif de plus que de positifs

Nombres entiers 8 bits signés

- 00 à 7F : nombres positifs de 0 à $2^7 - 1$ (127)
- FF à 80 : nombres négatifs de -1 à -2^7 (-128)

Nombres entiers 16 bits signés

- 0000 à 7FFF : nombres positifs de 0 à $2^{15} - 1$ (32767)
- FFFF à 8000 : nombres négatifs de -1 à -2^{15} (-32768)

Questions

- 32 bits? 64 bits? 128 bits?

Différents codages des entiers

bits	octets	nom RISC-V	signé	min	max
8	1	byte	non	0	255
			oui	-128	127
16	2	halfword	non	0	$65\,535$
			oui	$-32\,768$	$32\,767$
32	4	word	non	0	$\approx 4.29 \times 10^9$
			oui	$\approx -2.15 \times 10^9$	$\approx 2.15 \times 10^9$
64	8	doubleword	non	0	$\approx 1.84 \times 10^{19}$
			oui	$\approx -9.22 \times 10^{18}$	$\approx 9.22 \times 10^{18}$
128	16	quadword	non	0	$\approx 3.40 \times 10^{38}$
			oui	$\approx -1.70 \times 10^{38}$	$\approx 1.70 \times 10^{38}$

Détermination du signe

Un nombre est-il positif, négatif ou nul?

- Bit de poids fort $= 1$? $\rightarrow$ si oui, négatif
- Tous les bits à 0 ? $\rightarrow$ si oui, nul
- Autrement, positif

En hexadécimal

- Chiffre de poids fort entre 8 et F ? $\rightarrow$ si oui, négatif
- Tous les chiffres à 0 ? $\rightarrow$ si oui, nul
- Autrement, positif

Calcul du complément

Méthode 1

- $\text{neg}(i) = 2^n - i$ où n est le nombre de bits
- Exercice: neg(00101010) = ? (8 bits signé)

Méthode 2

- Inverser les 0 et les 1
- Ajouter 1 au résultat
- C'est ce que font les micro-architectures

En hexadécimal

- Soustraire chaque chiffre de F, soit $15_{(10)}$
- Ajouter 1 au résultat
- Exercice: neg(2A) = ?

Avantages et inconvénients du complément à deux

- 👍 Les nombres de 0 à $2^{n-1}-1$ se codent pareil
- 👍 Un bit détermine le signe
- 👍 Pour les micro-architectures
  - Calcul du complément relativement facile
  - Autres calculs arithmétiques relativement faciles
  - Pas besoin d'un circuit de soustraction
  - Donc: UCT moins chères
- 👎 Pour les humains
  - Lecture des négatifs non intuitive
  - Calcul du complément difficile de tête
  - `0A` $\rightarrow 10$, `F6` $\rightarrow -10$

Codages = attention aux ambiguïtés

Rappel

- Un paquet de bits n'a pas de signification intrinsèque

On doit **préciser** le codage

- Quel est le nombre de bits (n) ? (ou nom de la taille)
  - octet (8), demi mot (16), mot (32), double mot (64), etc.
- Quel est le codage utilisé?
  - non signé, *unsigned* (u), de 0 à $2^n - 1$
  - Signé, *signed* (s), de -2^{n-1} à $2^{n-1} - 1$

Opérations arithmétiques

Arithmétique

Définition

- Science des nombres (αριθμός = nombre)

Opérations traditionnelles

- Addition et soustraction
- Multiplication et division

Exercices

- $1264 + 756 = ?$
- 2567 - 666 $= ?$
- 345 - 1264 $= ?$

Arithmétique des ordinateurs

Comme à l'école primaire

- Mais on travaille en binaire (et en hexadécimal)

Exercices

- $1001110011_{(2)} + 1010_{(2)} = ?$
- $95C_{(16)} + E1_{(16)} = ?$

Addition machine

- Micro-architecture sait faire les additions
- Comme à l'école (ou quasiment)
- Les détails plus tard, et en INF4170

Gestion des signes

Complément à deux: Tout fonctionne tout seul

- Propriété mathématique intéressante du complément à deux
- On peut **réutiliser** tel quel l'additionneur des entiers non signés
  - Pas de microarchitecture dédiée aux additions signées
  - UCT plus petites et moins chère
- On **ignore** les retenues qui **tombent** à gauche

Soustraction

- Pas de soustraction dédiée : on additionne le complément
- UCT plus petites et moins chère

Exemple en 8 bits signé

- $20 - 10 =$

Exemple en 8 bits signé

- $20 - 10 = 20 + -10 =$

Exemple en 8 bits signé

- $20 - 10 = 20 + -10 =$

$$\begin{array}{rl} & \boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0} \rightarrow 20 \\ + & \boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0} \rightarrow \text{-}10 \\ 1 & \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{0} \rightarrow 10 \end{array}$$

Exercices

- $-6 + -7 = ?$
- $10 - 20 = ?$

Autres représentations signées

Complément à 1

- On inverse tous les bits pour les négatifs
- Exemples (4 bits)
  0000 $\rightarrow$ 0, 0011 $\rightarrow$ 3, 1100 $\rightarrow$ -3, 1111 $\rightarrow$ 0
- 👎 Mauvaises propriétés mathématiques
- 👎 Deux zéros

Signe et magnitude

- Un bit pour le signe
- Les autres bits pour la valeur absolue du nombre
- Exemples (4 bits)
  0000 $\rightarrow$ 0, 0011 $\rightarrow$ 3, 1011 $\rightarrow$ -3, 1111 $\rightarrow$ -7
- 👎 Deux zéros
- 👎 Algos spécifiques nécessaires
- On verra ça avec la représentation des flottants

Autre représentation signée

Binaire décalé (*offset binary*)

- On détermine une valeur arbitraire k (le pôle)
  par exemple $k = 2^n - 1$
- Le nombre i se représente par la représentation entière de $i + k$
- Exemples (4 bits) avec k=7
  0111 $\rightarrow$ 0, 1100 $\rightarrow$ 3, 0100 $\rightarrow$ -3, 1111 $\rightarrow$ 8, 0000 $\rightarrow$ -7
- 👎 Opérations plus compliquées
- On verra ça aussi avec la représentation des flottants

Instructions arithmétique RISC-V

Rappel des registres RV64I

- RV64I?
  - **R**ISC-**V**
  - **64** bits
  - Architecture de base: juste les entiers (**I**nteger)
- 32 registres
  - de `x0` à `x31`
  - 64 bits chacun
- `x0` est le registre `zero`
  - Il contient toujours 0
  - Ranger une valeur dedans équivaut à la perdre (`/dev/null`)
- Les registres ont des **noms d'ABI** mais on verra ça plus tard

Instructions RISC-V

- **Mnémonique**: nom très court représentant l'instruction
- **Opérandes** (registres ou valeurs) séparées par des virgules
- Les détails de la syntaxe une autre fois

Types d'instructions RISC-V

- Type R: registre à registre
  - un registre résultat (destination)
  - deux registres d'opérandes (sources)
- Type I: immédiat
  - un registre résultat (destination)
  - un registre d'opérande (source)
  - une valeur numérique immédiate
- Les autres types une autre fois

Somme et différence

`add rd, rs1, rs2` (*add*, type R)

- Range dans `rd` la somme des registres `rs1` et `rs2`

`addi rd, rs1, imm` (*add immediate*, type I)

- Range dans `rd` la somme des registres `rs1` et de la valeur `imm`
- Attention: `imm` sur 12 bits (signé)
  -2^{11} à $2^{11} - 1$, soit -2048 à 2047

`sub rd, rs1, rs2` (*substract*, type R)

- Range dans `rd` la différence des registres `rs1` et `rs2`
- Note: pas de `subi`, on fait `addi` avec un négatif

Pseudoinstruction de chargement immédiat

Pseudoinstruction

- Instruction assembleur
- **Forme particulière** d'instructions machines existantes
- Permet de réduire le nombre d'instructions machines
- Donc le cout des UTC

`li rd, imm` (*load immediate*, pseudo) première forme

- Range dans `rd` la valeur immédiate `imm`
- Pseudoinstruction de `addi rd, x0, imm` (`rs` $= 0 +$ `imm`)
- Attention: `li` prend des formes différentes si `imm` n'est pas codable en 12 bits signé

Exercices

- Calculer $x1 = 837 + 1632 - 105 + 2010 + 626$

Exercices

- Calculer $x1 = 837 + 1632 - 105 + 2010 + 626$

```
li   x1, 837
addi x1, x1, 1632
addi x1, x1, -105
addi x1, x1, 2010
addi x1, x1, 626
```

Note: Le registre `x1` **accumule** les résultats intermédiaires

Exercices

- Calculer $x1 = 837 + 1632 - 105 + 2010 + 626$

```
li   x1, 837
addi x1, x1, 1632
addi x1, x1, -105
addi x1, x1, 2010
addi x1, x1, 626
```

Note: Le registre `x1` **accumule** les résultats intermédiaires

- Calculer $x2 = 500 - x1$
- Calculer $x3 = x2 + 100 - (150 - x1)$

Autres pseudoinstructions arithmétiques

`neg rd, rs1` (*negate*, pseudo)

- Range dans rd le complément de rs1
- Pseudoinstruction de `sub rd, x0, rs1` (`rd` = 0 - `rs1`)

`mv rd, rs1` (*move*, pseudo)

- Range dans rd la valeur de rs1
- Pseudoinstruction de `add rd, x0, rs1` (`rd` = 0 + `rs1`)

`nop` (*no operation*, pseudo)

- Ne fait rien
- Pseudoinstruction de `add x0, x0, x0`

Load upper immediate

- `li` permet d'initialiser 12 bits des registres
- Mais comment initialiser les bits restants?

lui rd, imm (*load upper immediate*, type U)

- Range dans `rd` la valeur immédiate `imm` décalé de 12 bits
- Range dans `rd` la valeur `imm` $\times 2^{12}$ ($=$ `imm` $\times 4096$)
- `rd`[63:12] $=$ `imm` et `rd`[11:0] $= 0$
- Attention, `imm` sur 20 bits (signé)

Exemple

```
lui x1, 1 # charge 4096 dans `x1`
lui x2, 2 # charge 8192 dans `x2`
```

Inconvénient

- Pas très ergonomique :(

li: seconde forme

- `li rd, imm`: Si `imm` pas codable sur 12 bits, alors
- l'assembleur **combine** `lui` et `addi`
  (et d'autres instructions si besoin)

Exemples

```
li x1, 4100 # devient
 lui  x1,  1          # x1 = 0x1000 = 4096
 addi x1, x1, 4       # x1 = x1+4 = 4096+4 = 4100

li x2, 3000 # devient
 lui  x2,  1          # x2 = 0x1000 = 4096
 addi x2, x2, -1096 # x2 = x2-1096 = 4096-1096 = 3000
```

Note: l'assembleur a beaucoup de liberté pour `li`

Exercices

```
li x3, 4096 # ?
li x4, 2048 # ?
```

Accès au compteur ordinal

Compteur ordinal en RISC-V

- Registre spécial `pc` (*program counter*)
- Correspond à **adresse de l'instruction en mémoire**
- N'est pas un registre général (`x0` à `x31`)
- N'est pas utilisable comme **opérande**
- Donc besoin d'instructions **dédiées**

`auipc rd, offset` (type U)

- *add upper immediate to program counter*
- Range dans `rd` la somme de `offset` $\times 2^{12}$ et de `pc`
- Attention: `offset` sur 20 bits signés
- Note: Utiliser `imm`=0 pour avoir la valeur de `pc`

Débordement

Débordement

Définition

- Le résultat d'une opération qui est **hors domaine**
- Rappel: le domaine est **fini**: n bits codent 2^n nombres
- Synonymes: dépassement de capacité, *overflow*

Exemples 8 bits non signé (0 à 255)

- 200 + 100 = 300, mais 300 n'est pas codable
- 10 - 20 = -10, mais -10 n'est pas codable

Exemples 8 bits signé (-128 à 127)

- 127 + 2 = 129 mais 129 n'est pas codable
- 0 - -128 = 128 mais 128 n'est pas codable

Effet d'un débordement

- On a **mécaniquement** un résultat
- Mais qui n'est pas le **bon** résultat

Exemple 8 bit non signé

$$\begin{array}{rr} & \boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0} \rightarrow 200 \\ + & \boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0} \rightarrow 100 \\ 1 & \boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0} \rightarrow 44 \end{array}$$

Exercices

- Comment déborde 127 + 2 (8 bits signé) ?
- Peut-on faire déborder `neg` ?

Gestion des débordements

car les débordements font parti de la vie...

Statiquement

- Déterminer d'avance les valeurs extrêmes
- Et utiliser un codage assez grand

Dynamiquement

- Détecter les débordements à l'exécution
- Programmativement: avec des calculs supplémentaires
- Matériellement: si l'architecture (UCT) peut le faire pour nous

Ignorer le problème

- *What could go wrong?*

Vol 501 d'Ariane V

- 4 juin 1996, vol inaugural d'Ariane V
- 36,7 secondes après le décollage, la fusée s'autodétruit
- 370 millions de dollars en fumée

Source photo: Agence spatiale européenne

Explication en très bref

- Systèmes de guidage inertiel récupéré d'Ariane IV, entier **16 bit**
- Ariane V différent, accélérations **5 fois** plus fortes
- 16 bits insuffisants, **débordement** du système de guidage
- Système de guidage **plante**, information de débogage **émises** (mode débug actif en prod!)
- Ordinateur de bord **interprète** les information de débug comme des **données de guidage** (un paquet de bits ça s'interprète!)
- L'ordinateur de bord ordonne une **correction** de trajectoire (or la trajectoire est bonne, il n'y a rien à corriger)
- La fusée s'écarte **brutalement** du plan de vol
- Trop brutalement, un des deux boosters se **détache**
- Situation **critique**, déclenchement de l'**autodestruction**

Source: Agence spatiale européenne

Déterminer un débordement programmativement

En non signé

- `add rd, rs1, rs2` a débordé si et seulement si `rd` $<$ `rs1`

En signé

- `add rd, rs1, rs2` a débordé si et seulement si
  - `rs1` positif, `rs2` positif et rd négatif
  - ou `rs1` négatif, `rs2` négatif et rd positif

Note: positif + négatif ne déborde jamais

Déterminer un débordement par l'architecture

RISC-V

- Pas de gestion dédiée pour les débordements d'entiers
- Raisons économiques et de performance

x86, ARM

Un registre spécial contient 4 bits nommées N (ou S), Z, V et C

- N = 1 si dernier résultat strictement **n**égatif (ou **s**igne), 0 sinon
- Z = 1 si dernier résultat nul (**z**éro), 0 sinon
- V = 1 si dernier résultat a débordé (o**v**erflow), 0 sinon
- C = capture la retenue (**c**arry) (1 ou 0)

Débordement et tailles d'architectures

RV32 vs. RV64

- La taille des registres dépend de l'architecture
- `add x1, x2, x3`
  - peut déborder en RV32I
  - mais ne pas déborder en RV64I
- La même instruction machine donne alors des résultats différents
- Source de bogues subtils si on est pas rigoureux
- Surtout si le débordement est silencieux (n'est pas vérifié)

Conclusion

Résumé

Arithmétique

- Binaire, hexadécimal, entier non signé, entier signé en complément à deux, débordement
- Octet, demi-mot, mot, double mot
- Nos premières instructions: `add`, `addi`, `sub`, `lui`, `auipc`
- Et pseudoinstructions: `li`, `mv`, `neg`, `nop`

Résumé

Tout n'est que bits

- L'information est représentable (codable) en un paquet de bit
- L'interprétation (décodage) d'un paquet de bit dépend du lecteur

Décoder 2 octets `CAFE`

- Un nombre entier 16 bits non signé: 51966
- Un nombre entier 16 bits signé: -13570
- Deux nombres entiers 8 bits non signés: 202 et 254
- Deux nombres entiers 8 bits signés: -54 et -2

Autres opérations arithmétiques

- Opérations logique bit à bit (not, and, or, xor, etc.)
- Décalages (sll, srl, sra, etc.)
- Multiplication et divisions (mul, div, rem, etc.)
- Comparaisons (beq, blt, slt, etc.)

Plus tard...

Autres codages

Il n'y a pas que les nombres entiers dans la vie

Instructions machines

- Rappel: les instruction machines sont en mémoire
- Donc codées sous forme de bits
- RISC-V
  - Chaque instruction est sur 32 bits (4 octets) par défaut
  - Extension "C" (*compressed instructions*) pour du 16 bits
  - On y reviendra...

Nombres à virgule flottante

- Approximation des nombres réels
- Exemple: types float et double en Java, C, C++, etc.
- Plus tard...

Autres codages

Données arbitraires

- Représentation programmative libre
- On **programme** nos propres représentations
- Tableaux, structures, objets, etc.
- Plus tard aussi...

Décoder 4 octets 00310023

- Nombre entier 32 bits ? 3 211 299
- Chaîne caractères UTF-16BE ? `"0#"` (plus tard)
- Instruction RISC-V ? `sb x3, 0(x2)` (plus tard)
- Etc.

La prochaine fois

Éléments d'assembleur

- Instructions et directives
- Entrées-sorties
- Branchements
- Appels de routines et bibliothèques