16. Code machine et langage assembleur

Principes de fonctionnement des ordinateurs

Jonas Lätt

Centre Universitaire d'Informatique

Contenu du cours

Partie I: Introduction

Partie II: Codage de l'information

Partie III: Circuits logiques

Partie IV: Architecture des ordinateurs

1. Introduction
2. Histoire de l'informatique
3. Information digitale et codage de l'information
4. Codage des nombres entiers naturels
5. Codage des nombres entiers relatifs
6. Codage des nombres réels
7. Codage de contenu média
8. Portes logiques
9. Circuits logiques combinatoires et algèbre de Boole
10. Réalisation d'un circuit combinatoire
11. Circuits combinatoires importants
12. Principes de logique séquentielle
13. Réalisation de la bascule DFF
14. Architecture de von Neumann
15. Réalisation des composants
16. **Code machine et langage assembleur**
17. Architecture d'un processeur
18. Performance et micro-architecture
19. Du processeur au système

Langage assembleur de l'architecture RISC-V

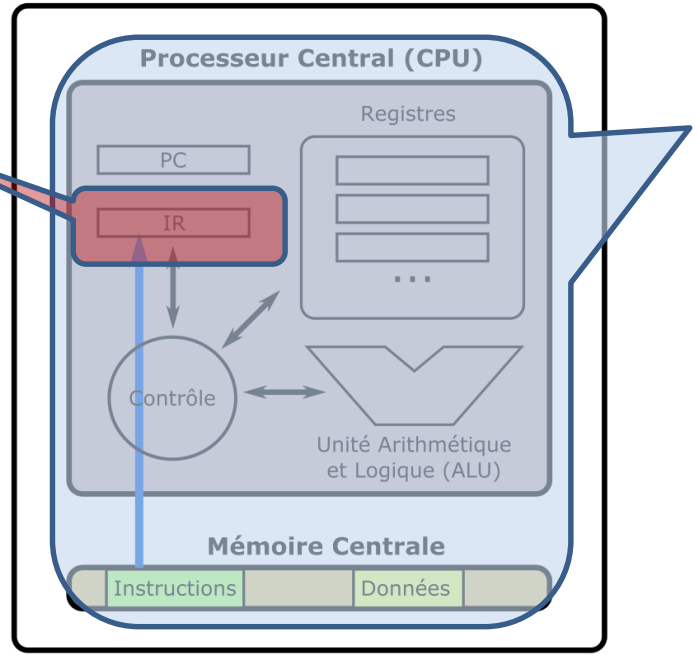
L'architecture RISC-V

- Une architecture libre et ouverte
- Développée par un consortium, origine à l'université de Berkeley
- Malgré son origine académique, elle est destinée à être utilisée dans des ordinateurs réels.
- Une architecture RISC (Reduced Instruction Set Computer): Jeu d'instructions limité (et donc, architecture relativement simple).
- Il existe des versions 32-bit, 64-bit, 128-bit de l'architecture. Nous étudierons la version 32-bit (taille des registres).

Les pièces manquantes

Jeu d'instructions:

- Format des instructions?
- Types d'instructions à disposition?



Réalisation de l'architecture

- Comment connecter les différentes pièces?
- Quelles opérations le circuit peut-il exécuter?

Architecture d'un ordinateur

Vision abstraite

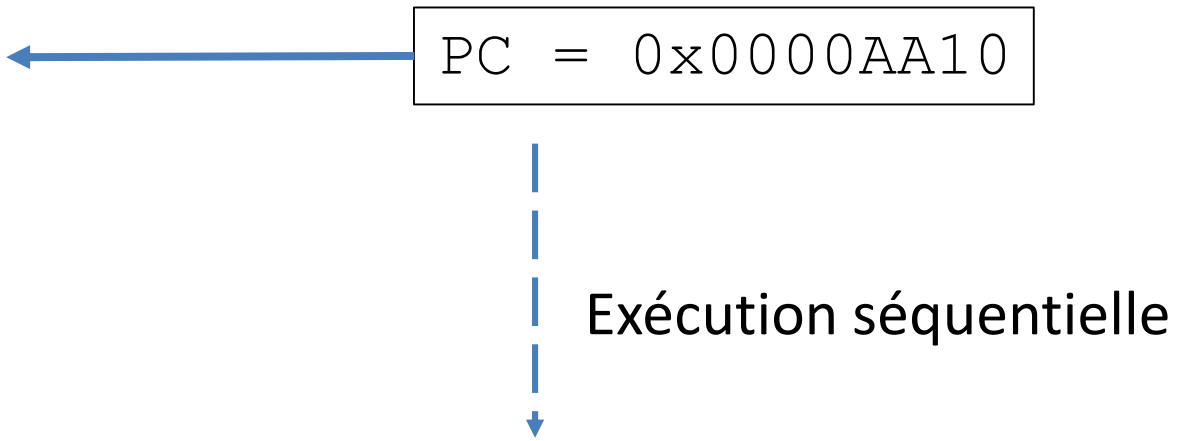
Vision concrète

Le langage machine

Un programme dans la mémoire centrale:

(Exemple pour l'architecture RISC-V)

Adresse	Instruction
⋮	⋮
0x0000AA10	0x00C580B3
0x0000AA14	0x00E68133
0x0000AA18	0x402080B3
⋮	⋮

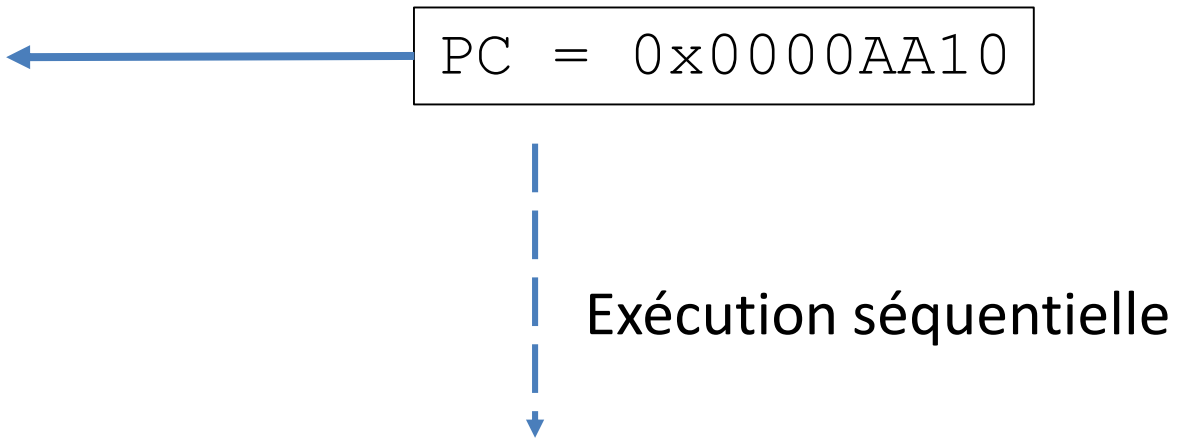


Le langage machine

Un programme dans la mémoire centrale:

(Exemple pour l'architecture RISC-V)

Adresse	Instruction
⋮	⋮
0x0000AA10	0x00C580B3
0x0000AA14	0x00E68133
0x0000AA18	0x402080B3
⋮	⋮

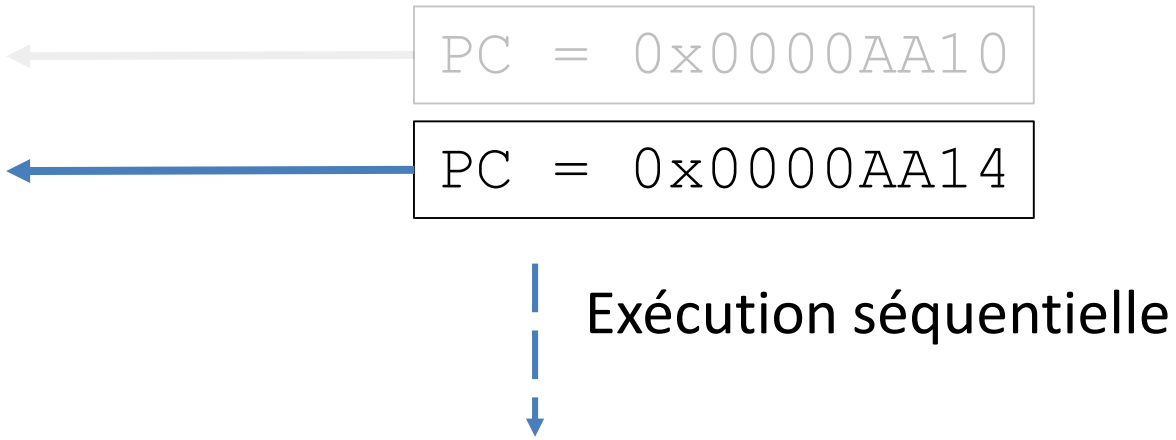


Le langage machine

Un programme dans la mémoire centrale:

(Exemple pour l'architecture RISC-V)

Adresse	Instruction
⋮	⋮
0x0000AA10	0x00C580B3
0x0000AA14	0x00E68133
0x0000AA18	0x402080B3
⋮	⋮



Types d’instruction

- Opérations arithmétiques et logiques
- Transferts de données
- Branchements
- Autres (accès aux périphériques de l’ordinateur, …)

Code machine et mnémoniques

Adresse	Instruction
⋮	⋮
0x0000AA10	0x00C580B3
0x0000AA14	0x00E68133
0x0000AA18	0x402080B3
⋮	⋮

00000000011000101100000000010110011

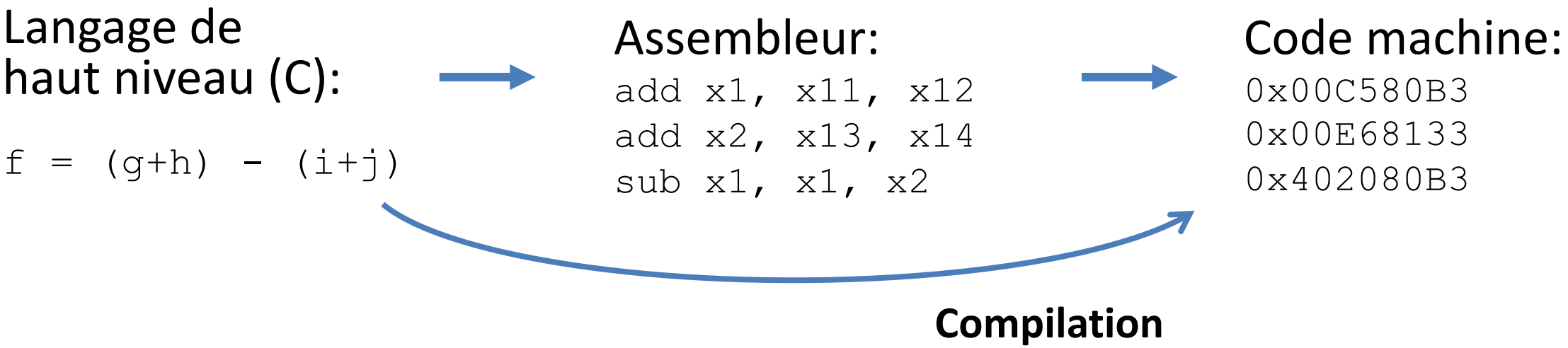
Choix de l'opération

0000000	01100	01011	000	00001	0110011
	12	11		1	
	x12	**x11**	**add**	**x1**	

(Mnémoniques)

Langage assembleur: **`add x1, x11, x12`**

Langages de haut niveau et compilateurs



Variables et langage machine

- Langage de haut niveau: la variable est une quantité abstraite pour laquelle «de la mémoire» est réservée.
- Langage machine: dans la mémoire centrale, dans un registre, ou les deux. Utilisation des registres pour de meilleures performances.

Assembleur RISC-V - Registres

L'architecture RISC-V met a disposition 32 registres:

- **x0**: Un registre constant qui vaut toujours zéro.
- **x1-x31**: Les autres registres, accessibles en lecture/écriture.

Aperçu des instructions assembleur

Instructions Arithmétiques

add	(Add) Addition de deux entiers provenant de registres.
addi	(Add Immediate) Addition d'un entier d'un registre et d'une constante.
sub	(Subtract) Soustraction de deux entiers provenant de registres.

Instructions Logiques

and	(AND) ET logique entre deux entiers provenant de registres.
andi	(AND Immediate) ET entre un entier d'un registre et une constante.
or	(OR) OU logique entre deux entiers provenant de registres.
ori	(OR Immediate) OU entre un entier d'un registre et une constante.
srli	(Shift Right Logical) Décalage vers la droite des bits d'un registre.
slli	(Shift Left Logical) Décalage vers la gauche des bits d'un registre.

Aperçu des instructions assembleur

Transfert de données

lw	(Load word) Copie de 4 octets de la mémoire vers un registre.
sw	(Store word) Copie de 4 octets d'un registre vers la mémoire.

Branchements

j	(Jump) Branchement inconditionnel vers une adresse constante.
beq	(Branch On Equal) Branchement conditionnel en cas d'égalité.
bne	(Branch On Not Equal) Branchement conditionnel en cas d'inégalité.
blt	(Branch On Less-Than) Branchement conditionnel (plus-petit-que pour entiers relatifs).
bge	(Branch on Greater-Equal) Branchement conditionnel (plus-grand-ou-égal pour entiers relatifs).

Remarque: cet aperçu d'instructions RISC-V est incomplet. D'autres instructions sont listées dans une table du polycopié.

Instructions arithmétiques

Format Standard:	**`Opération`** `reg-dest, reg-src1, reg-src2`
Format Immédiat:	**`Opération`** `reg-dest, reg-src1, constante`

Exemple:

Code C correspondant:

```
f = (g+h) - (i+7);
```

Valeurs en entrée:

```
x1 - g, x2 - h,
x3 - i
```

Valeur en sortie:

```
x4 - f
```

Code assembleur:

```
# x5 = g+h
# x6 = i+7
# x4 = x5-x6
```

Transfert de donneés

- **lw** («load word»): transfert de données de la mémoire centrale vers un registre.
- **sw** («store word»): transfert de données d'un registre vers la mémoire centrale.

Syntaxe:

- Toute adresse en mémoire se décompose en une **adresse de base** et un **offset**.
- Idée: On considère toutes les données en mémoire comme des **tableaux de valeurs** («**Array**»)

Qu’est-ce qu’un tableau de valeurs «Array»

- Tableau de valeurs: des valeurs disposées de manière consécutive en mémoire.
- Il suffit de mémoriser l’adresse de base, et on peut accéder à n’importe quel élément.

	Adresse	Données
	⋮	⋮
Adresse de base A →	0x00FABCD0	0x00001111
Offset de 4 →	0x00FABCD4	0x00006666
Offset de 8 →	0x00FABCD8	0x0000FFFF
	0x00FABCDC	0x00000000
	⋮	⋮

Exemple en langage C:

```
int A[] = {0x1111, 0x6666, 0xFFFF, 0x0000};
```

Transfert de données

Format:	**Opération** reg, offset(regaddr)

Exemple:

Code C correspondant:

```
A[0] = h + A[5];
```

Valeurs en entrée:

```
x1 – Adresse de base du tableau
de valeurs A
x2 - h
```

Code assembleur:

```
# x4 = A[5] (A[5] se trouve à
# un offset de 5*4=20 octets de
# l'adresse de base A).
# x4 = x4 + h
# A[0] = x2
```

Branchements dans le flux d'instrutions

- A chaque fois que les instructions ne s'exécutent pas de manière séquentielle, on parle d'un branchement dans le flux d'instructions.
- Dans les langages de haut niveau, ces branchements s'expriment à l'aide de **structures de contrôle**: **if**, **for**, **while**, …
- En langage machine, elle s'expriment à l'aide de «sauts» explicites: les branchements.

Contrôle sélectif du flux d'instruction («if-else»)

- Dans le langage C, on peut exprimer un if-else, en n'utilisant rien d'autre que des instructions «goto».
- C'est cette philosophie «goto» qui sera utilisée pour exprimer un contrôle sélectif en assembleur.

Code C, version if-else:

```
if (i==j) {
    f = f-i;
} else {
    f = g+h;
}
```

Code C, version "goto":

```
if (i==j) goto L1;
f = g+h;
goto L2;
L1: f = f-i;
L2:
```

Branchements dans le flux d'instruction

Saut inconditionnel:	`j adresse`
Saut conditionnel:	`Opération reg1, reg2, adresse`

Emplacement en mémoire que pourrait par exemple occuper l'instruction.

Exemple:

Code C correspondant:

```
if (a==b) goto L1
 ...
L1: ...
```

Valeurs en entrée:

```
x3 – a
x4 - b
```

Code assembleur:

```
(0x0000BB10) beq x3, x4, 4
(0x0000BB14) ...
(0x0000BB18) ... destination
(0x0000BB1C) ...

# Une autre possibilité est:
(0x0000BB10) beq $s3, $s4, L1
        ...
(0x0000BB18) L1:...
```

Les sauts s'expriment par une adresse relative (positive/négative) qui est un multiple de deux (car les instructions peuvent prendre 2 ou 4 octets en mémoire).

Par souci de simplicité, on peut représenter l'adresse en mémoire d'une instruction par un **label**, dont le nom est librement choisi.

Contrôle sélectif du flux d'instruction

Code C, version if-else:

```
if (i==j) {
    f = f-i;
} else {
    f = g+h;
}
```

Code C, version "goto":

```
if (i==j) goto L1;
f = g+h;
goto L2;
L1: f = f-i;
L2:
```

Valeurs en entrée:

```
x1 – f, x2 – g, x3 - h
x4 – i, x5 - j
```

Code assembleur:

```
beq x4, x5, L1          # if i==j goto L1
add x1, x2, x3          # f = g+h
j L2                    # goto L2
L1: sub x1, x1, x4      # L1: f = f-i
L2:
```

Contrôle itératif du flux d'instruction («for/while»)

De même, des boucles *for* ou *while* peuvent être exprimées en formalisme «goto»:

Code C, version "for":

```
for ( int i=i1; i<=i2; i+=increment)
{
     g = g + data[i];
}
```

Code C, version "goto":

```
int i = i1;
Loop: g = g + data[i];
      i = i + increment;
      if (i <= i2) goto Loop;
```

Contrôle itératif du flux d'instruction

Code C, version "goto":

```
int i = i1;
Loop: g = g + data[i];
      i = i + increment;
      if (i <= i2) goto Loop;
```

Valeurs en entrée:

```
x1 – g, x2 – i2, x3 – i1
x4 – increment, x5 – Base du tableau "data"
```

Code assembleur:

```
Loop: slli x6, x3, 2     # x6=4*i
      add x6, x6, x5     # x6=data+4*i
                         # (x6 contient l'adresse
                         #  de data[i])
      lw x7, 0(x6)       # x7 = data[i]
                         #(obtenu à l'adresse x6)
      add x1, x1, x7     # g=g+data[i]
      add x3, x3, x4     # i=i+increment
      ble x3, x2, Loop   # if (i<=i2) goto Loop
```

Contrôle itératif du flux d'instruction

Le registre s3 adoptera, au fur et à mesure, les valeurs i1, i1+1, i1+2, … i2

En langage assembleur RISC-V, il faut manuellement calculer les adresses en mémoire: adresse du i-ème élément = adresse de base + 4*i

Les divisions ou multiplications par une constante qui est une puissance de 2 s'effectuent rapidement par des bit-shift: un shift à gauche de 2 positions correspond à une multiplication par 4.

Rappel: pour cette comparaison, x2 et x3 sont considérés comme des entiers relatifs.

Code assembleur:

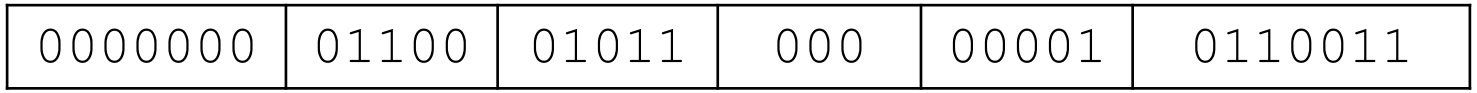
```
Loop: slli x6, x3, 2      # x6=4*i
      add x6, x6, x5      # x6=data+4*i
                          # (x6 contient l'adresse
                          #  de data[i])
      lw x7, 0(x6)        # x7 = data[i]
                          #(obtenu à l'adresse x6)
      add x1, x1, x7      # g=g+data[i]
      add x3, x3, x4      # i=i+increment
      ble x3, x2, Loop    # if (i<=i2) goto Loop
```

Format du code machine RISC-V

Dans le langage assembleur RISC-V, nous trouvons trois types d'instructions:

- *Format R*: Instructions arithmétiques / logiques
- *Format I*: Instructions arithmétiques / logiques «immédiates» (avec constante); instructions d'accès à la mémoire
- *Format SB*: Branchements conditionnels
- *… autres formats disponibles*

Chacun de ces trois types suit une convention de codage. Exemple pour le format R:



`add x1, x11, x12`

Format R («Registre»)

Exemples:

```
add x1, x11, x12
xor x1, x11, x12
```

	7 bits	*5 bits*	*5 bits*	*3 bits*	*5 bits*	*7 bits*
R	funct7	rs2	rs1	funct3	rd	Opcode

Format R («Registre»)

`add x1, x11, x12`

	7 bits	*5 bits*	*5 bits*	*3 bits*	*5 bits*	*7 bits*
R	funct7	rs2	rs1	funct3	rd	Opcode
	0000000	01100	01011	000	00001	0110011
	0	*12*	*11*	*0*	*1*	*51*
	neg-b = 0	**`x12`**	**`x11`**	**`add`**	**`x1`**	arithm./logique

000	add / sub
001	sll
110	or
111	and

Format I («Immédiat»)

Exemples:

```
addi x1, x11, 42
lw   x1, 8(x2)
beq  x1, x2, L1
```

	12 bits	*5 bits*	*3 bits*	*5 bits*	*7 bits*
I	imm[11:0]	rs1	funct3	rd	Opcode

Format I («Immédiat»)

Exemples:

`addi x15, x1, -50`

	12 bits	*5 bits*	*3 bits*	*5 bits*	*7 bits*
I	imm[11:0]	rs1	funct3	rd	Opcode
	111111001110	00001	000	01111	0010011
	-50	1	0	15	19
	constante	x1	addi	x15	arithm./logique immédiat

Format SB («Store / Branch»)

Exemple: **`beq x10, x19, Loop`**

	7 bits	*5 bits*	*5 bits*	*3 bits*	*5 bits*	*7 bits*
SB	imm7	rs2	rs1	funct3	imm5	Opcode

Comment assigner une constante à un registre?

- Il n'y a pas de commande dédiée, pour respecter le principe d'un jeu d'instructions minimal.
- Solution: utiliser le registre x0 qui équivaut à la constante nulle.

```
addi x5, x0, 3    Assigner la valeur 3 à x5
```