

2-d ISA

L'ensemble des instructions = **Instruction Set Architecture**

Un modèle de machine:

- Unités fonctionnelles
- Registres
- Mémoire

Un modèle d'exécution

- Parallélisme d'instruction
- VLIW, Superscalaire

Un modèle d'instructions

- CISC/RISC, microcode
- Modes d'adressage
- Format d'instructions

2-d ISA: modèle machine

Mémoire

- Unité de stockage pour mémoriser le programme et les données
- La mémoire est divisée en des **cellules mémoire**,
 - Chacune possède un numéro unique, son **adresse**. Adressage sur M bits = 2^M cellules. Actuellement, M=64
 - Contiennent toutes le même nombre de bits. N bits = 2^N valeurs possibles. Pas de typage (instruction, entier, flottant, ...)
- Taille mémoire: $2^M * N / 8$ octets.

2-d ISA: modèle machine

- Deux adresses consécutives sont adjacentes.
- Cellule mémoire: plus petite unité mémoire manipulable par les instructions
- Différents types de mémoire (vu de l'ISA)
 - **Random Access Memory (RAM)**: on accède à n'importe quelle adresse, dans n'importe quel ordre. Contraire d'un accès séquentiel. Perd les données quand pas alimenté
 - **Read-Only Memory (ROM)**: comme RAM, mais seulement en lecture. D'habitude pour programmes, systèmes d'exploitation...Non volatile.
 - **Registres**: sert de stockage temporaire. Accès très rapide

2-d ISA: modèle machine

- Utilisation particulière mémoire: **pile**
 - Zone mémoire accédée que par un registre contenant l'adresse du sommet de pile
 - Accès en empilant/dépilant des valeurs de la pile vers registres (push/pop instructions sur x86)
 - Utilisé pour des appels de fonction (voir cours compilation), pour stocker variables locales à une fonction.
 - Début de fonction: met sur la pile toutes les variables locales
 - Fin de fonction: on depile toutes les valeurs locales

2-d ISA: modèle machine

Unités de traitement

- **Unités de contrôle:** responsable décodage instruction à partir de la mémoire
- **Unités fonctionnelles**
 - La plus simple: unité arithmétique et logique (UAL ou ALU), fait tous les calculs arithmétiques et logiques
 - Spécialisés: unité flottante (calcul flottant), voire spécialisés pour certains types d'instructions
 - Il peut y avoir plusieurs unités d'un même type pour calcul parallèle (voir apres)

2-d ISA: modèle machine

Registres

- “Variables”, zone mémoire rapide pour stockage valeurs temporaires
- Nombre limité, taille en bits fixée. Ensemble des registres: **banc des registres**
- **Registres de calcul:** généralistes (stocke d'importe quoi), entier ou flottant, vectoriels

2-d ISA: modèle machine

Registres

- **Registres spéciaux:**
 - Compteur ordinal (program counter): stocke l'adresse de la prochaine instruction à exécuter
 - Registre d'instruction (instruction register): stocke l'instruction en cours
 - Registre de pile: indique le sommet d'une pile en mémoire pour des instructions de manipulation de pile.
 - Registres d'état: divers flags booléens pour indiquer le résultats d'opérations (overflow, resultat d'un test, ...)

2-d ISA: instructions

Instructions

RISC

reduced instruction set computer

- Jeu limité d'instructions
- Unité de controle simple
- Une instruction est traitée directement par l'unité de controle
- Programmes asm plus longs

CISC

complex instruction set computer

- Jeu d'instruction très étendu
- Unité de controle très compliquée
- Les instructions correspondent à du microcode (un petit programme)
- Programmes courts

2-d ISA: instructions

RISC

Architectures

- MIPS, Sparc, ARM, Power/PowerPC

S'appuie sur compilateur pour optimisations

Seuls les LOAD/STORE peuvent accéder à la mémoire, les autres opérandes sont des registres

Beaucoup de registres

CISC

Architectures

- X86, 68x, VAX

Maintenant, CISC microcodé comme du RISC.

Beaucoup d'instructions (difficile à toutes utiliser par le compilateur)

2-d ISA: instruction

RISC

Ma, Mb: adresses d'une valeur

r0, r1, r2, r3 registres

```
ld r1 , Ma
```

```
ld r2, Mb
```

```
ld r3, Mc
```

```
mul  r0, r1, r2
```

```
add  r0, r0, r3
```

```
st Ma, r0
```

CISC

```
ld r1, Ma
```

```
ld r2, Mb
```

```
fma  r3 = r1, r2, (Mc)
```

```
st Ma, r3
```

2-d ISA: instruction

Modes d'adressage

Comment accède-t-on à une valeur en mémoire ? Différents modes

- En mettant directement son adresse (mode direct)

```
L1: ld r1, 20000    // r1 = value at address 20000
```

```
add r0,r0,1    // r0 = r0+1
```

```
cmp r0,r2    // compare r0 and r2
```

```
jeq L1    // jump to L1 if r0==r2
```

- En utilisant un registre contenant la valeur de l'adresse (mode indirect)

```
mov r0, 20000
```

```
ld r1,(r0)
```

2-d ISA: instruction

- L'adresse est calculée à partir de la valeur dans un registre ajoutée à une valeur constante (adressage indexé)

```
mov r0, 18000      // r0 = 18000
```

```
ld r1, (r0+2000)   // r1 = value at address 20000
```

- L'adresse est calculée à partir de la valeur dans un registre ajoutée à un autre registre (adressage basé indexé)

```
mov r2, 128
```

```
cmp r1, r2
```

```
mov r1, 0
```

```
jlo L1 // r1 <= r2 ?
```

```
L1: ld r3, (r0+r1)
```

Que fait ce code ??

```
add r1, r1, 1
```

2-d ISA: instruction

- L'adresse est calculée à partir de la valeur dans un registre ajoutée à une valeur constante (adressage indexé)

```
mov r0, 18000      // r0 = 18000
```

```
ld r1, (r0+2000)   // r1 = value at address 20000
```

- L'adresse est calculée à partir de la valeur dans un registre ajoutée à un autre registre (adressage basé indexé)

```
mov r2, 128        cmp r1,r2
```

```
mov r1,0           jlo L1
```

```
L1: ld  r3, (r0+r1)  Ceci est une boucle parcourant 128
```

```
add r1,r1,1        cases mémoires consécutives à partir de r0
```

2-d ISA: instruction

Exemple Fibonacci: $A[i] = A[i-1] + A[i-2]$

```
int A[100];
```

```
int i;
```

```
A[0]=1;
```

```
A[1]=1;
```

```
for (i=2; i<100; i++)
```

```
    A[i] = A[i-1] + A[i-2]
```

2-d ISA: instruction

- Sur x86-32 et MIPS:
 - Une case mémoire = 1 octet
 - $A[i]$ prend 4 octets, les adresses de $A[i]$ et $A[i+1]$ sont séparées par 4 octets
 - Les registres utilisés sont sur 4 octets.
 - Sur x86-32, tout type d'adressage autorisé, pas plus d'une référence mémoire par instruction
 - Sur MIPS, que les load/store et pas d'adressage basé indexé

2-d ISA: instruction

X86 32 bits. ebp: contient adresse de A[2]. eax = 100. ebx = i*4

```
L:  movl %ecx,-4(%ebp,%ebx)  // ecx = value at ebp+ebx-4
    addl %ecx,-8(%ebp,ebx)   // ecx = ecx + value at ebp+ebx-8
    movl (%ebp,%ebx),%ecx    // store ecx at address ebp+ebx
    addl %ebx,4              // move to next value i
    cmpl %ebx,%eax           // is loop finished ? If no, goto L
    jle  L
```

2-d ISA: instruction

MIPS

add \$t4, \$0, \$t0 // t0 = &A[2], t4=A[2]

addi \$t4,\$t4,392 // t4 = &A[100]

L: lw \$t1, -4(\$t0) // t1 = value of A[i-1]

lw \$t2, -8(\$t0) // t2 = value of A[i-2]

add \$t3, \$t1, \$t2 // t3 = t1+t2

sw \$t3, (\$t0) // store t3 to A[i]

addi \$t0, \$t0, 4 // i=i+1

ble \$t0, \$t4, L // goto L if not finished

2-d ISA: instruction

- MIPS documentation:

google MIPS assembly code

- Intel

<http://www.intel.com/products/processor/manuals/>

2-d ISA: instructions MIPS

Petit tour des instructions MIPS

- Plusieurs catégories d'instructions
- Codage RISC



Routeur Cisco



chien Aibo



Sony PSP

2-d ISA: instructions MIPS

Instructions arithmétiques

- Les instructions sont du format

OP \$t1 , \$t2, \$t3 pour faire \$t1 = \$t2 **OP** \$t3

avec **OP** = add, sub, mul, div, and, or, xor, nor

et \$t1, \$t2, \$t3 les opérandes.

- \$t2 et \$t3 sont les registres sources
- \$t1 est le registre destination
- Il y a 32 registres de 32bits de \$0 à \$31. \$0 vaut toujours 0. Les registres \$t0..\$t9 sont d'autres noms pour des registres \$n (n>0).

2-d ISA: instructions MIPS

Codage instructions arithmétiques:

opcode	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- opcode: codage de l'opération effectuée (000000 pour add et sub, ...)
- rs, rt: numéro des registres sources
- rd: numéro du registre destination
- shamt: valeur constante pour 2 instructions de décalage
- func: codage de l'opération arithmétique effectuée (000000 pour add, 000001 pour sub, ...)

2-d ISA: instructions MIPS

Instructions immédiates

- Les instructions sont du format

OP **\$1** , **\$2**, **cst** pour faire **\$1 = \$2 OP cst**

avec **OP** = addi, subi, muli, divi, andi, ori, xori, nori

et **\$1**, **\$2**, **cst** les opérandes.

- **cst** est sur 16bits (signé, en complément à 2)

3- Schémas de prog. assembleur

If..then..else:

```
if (b1 == b2) {
    S1
} else {
    S2
}
```



```
bne $t1,$t2, Else
```

```
S1
```

```
j Endif
```

```
Else: S2
```

```
Endif:
```

2-d ISA: instructions MIPS

Instructions d'accès mémoire

- Les instructions sont du type

lw *\$t0*, *cst(\$t1)* pour placer la valeur à l'adresse *\$t1+cst* dans *\$t0*

sw *\$t0*, *cst(\$t1)* pour placer la valeur de *\$t0* à l'adresse *\$t1+cst*

- *cst* est facultatif (on peut ne pas l'indiquer)
- variantes de load: *lw*, *la* (load address), *lb* (load byte). Idem pour store.

2-d ISA: instructions MIPS

Instructions de branchement conditionnel

beq \$t0, \$t1, label pour faire: si $\$t0 == \$t1$ alors va au label

bne \$t0, \$t1, label pour faire: si $\$t0 \neq \$t1$ alors va au label

- label n'est pas une adresse mais un déplacement par rapport à l'adresse courante (sur 16 bits, signé en complément à deux).
- L'adresse du branchement est: adresse courante + label
- Quand on écrit l'assembleur, label est une étiquette. L'assembleur calcule le déplacement automatiquement

2-d ISA: instructions MIPS

Codage instructions immédiates, d'accès mémoire et de branchement

opcode	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

- opcode est un numéro correspondant au type d'instruction (lw, sw, beq, bne, addi, ...)
- rs et rt sont les deux registres opérandes de l'instruction
- immediate est la valeur immédiate (constante signée sur 16 bits). Son sens dépend de l'instruction
 - une valeur à ajouter à l'adresse pour lw et sw
 - une valeur pour le calcul arithmétique
 - le déplacement pour le branchement conditionnel

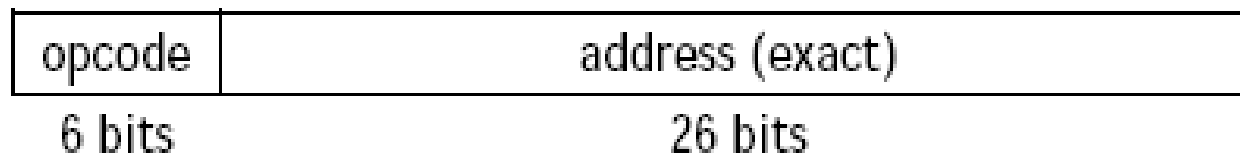
2-d ISA:instructions MIPS

Instructions de branchement

j **address** pour continuer l'exécution à l'adresse **address**

- **address** devra être une adresse multiple de 4 (toutes les instructions sont codées sur 4 octets)

Codage:



2-d ISA: instructions MIPS

Conclusion jeu d'instruction MIPS

- on n'en a vu qu'une petite partie ! Mais on a vu tous les formats
- opcode: principe général à tous les assembleurs, numéro identifiant le type d'instruction
- si on voulait plus de registres -> instructions plus longues
- on va faire un peu de programmation assembleur dans la partie suivante !

2-d ISA: instructions

Exercice de design d'un jeu d'instruction:

Proposez un codage de jeu d'instruction pour une machine qui a les instructions suivantes, prenant le moins de bits possibles.

- OP r1, r2, r3 avec OP=add,sub,mul,div,and,or,xor et qui fait $r1 = r2 \text{ OP } r3$
- OP r1, r2, cst avec OP=add,sub,mul,div,and,or,xor avec cst sur 16 bits
- load r1, (r2+cst) qui fait $r1 = \text{valeur stockée à l'adresse } r2 + \text{cst}$, avec cst sur 12 bits
- store (r2+cst),r1, avec cst sur 12 bits
- OP r1,r2,label avec OP=beq, bne, blo, bgt et label sur 12 bits
- Il y a 64 registres en tout.

2-e Implication théorique du codage

- Le codage avec des 0-1 n'est pas limitatif (Kleene)
- Les instructions qu'on a vu permettent
 - d'écrire tout programme (modèle PRAM)
 - d'écrire tout algorithme (hypothèse Church)

On va montrer pourtant qu'on ne peut pas tout programmer, ou de façon équivalente, que certains problèmes n'ont pas de solution algorithmique

2-e Implication théorique du codage

Un programme est fait d'instructions et de données

- toutes les instructions et les données sont codées de façon finies
- un programme est une liste finie d'instructions

=> **on peut représenter un programme par un nombre entier**

=> on peut énumérer les programmes par ordre croissant

- les données de départ d'un programme sont finies

=> **les données peuvent être représentées par un seul entier**

(Codage de Kleene)

2-e Implication théorique du codage

Théorème de la Halte (Turing)

Est-il possible de décider, pour tout programme et donnée en entrée, si son execution va terminer ou s'il va tourner indéfiniment ?

Ce problème n'a pas de solution algorithmique, on dit qu'il est

indécidable