

Architecture des Processeurs

1

Les systèmes à base de processeurs

Personal
Mobile
Devices

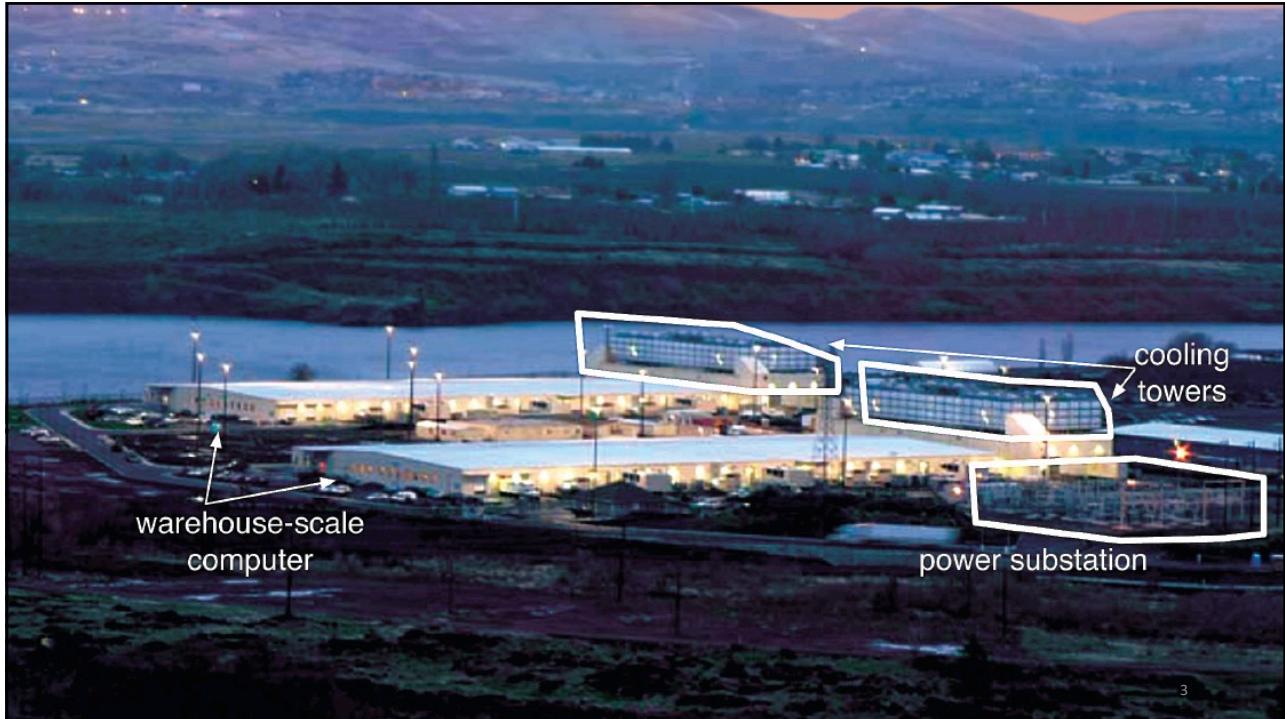


*Network
Edge
Devices*

Credits UC Berkley's CS61C.

2

1

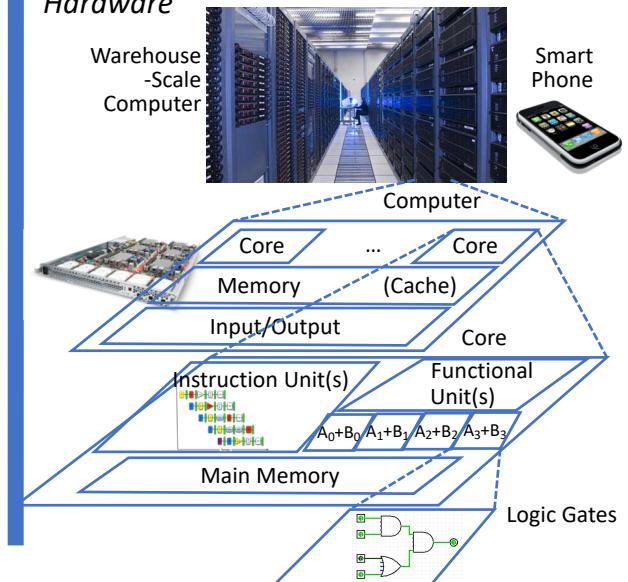


Hiérarchie des Systèmes

Software

- Des requêtes en parallèle
Alloués à des machines
- Tâches en parallèle
Alloués à un cœur
- Instructions en parallèle
>1 instruction @ one time
e.g., architecture pipeline
- Données en parallèle
>1 data item @ one time
- Des portes logiques
Toutes les portes logiques fonctionnent en parallèle simultanément

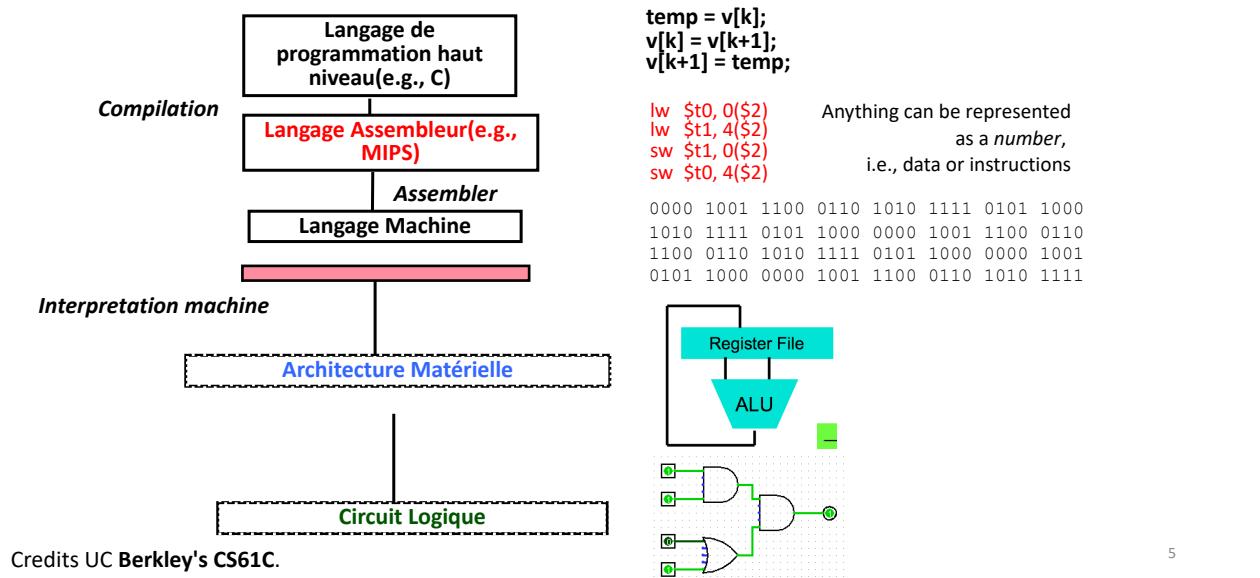
Hardware



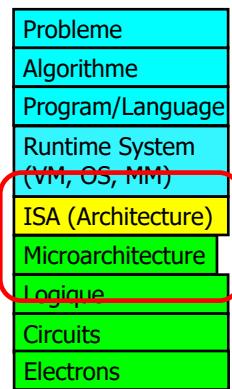
Credits UC Berkley's CS61C.

4

Différents niveaux d'abstraction/représentation



Objectifs du cours

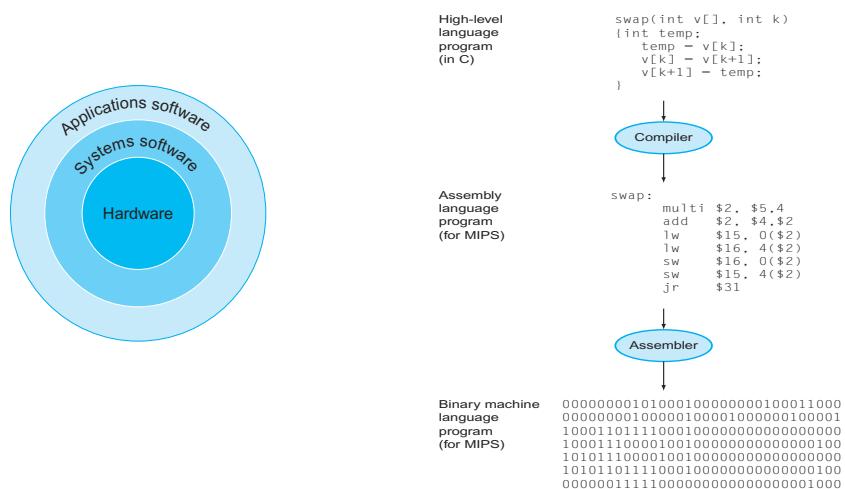


Comprendre comment un processeur fonctionne sous la couche logicielle et comment les décisions prises au niveau matériel affectent le logiciel/ programmeur

La maîtrise du matériel et du logiciel (et leur interface) permet:

- De développer de meilleurs logiciels
- De concevoir une meilleure cible matérielle
- De concevoir un meilleur système si on maîtrise les deux.

De la programmation à l'exécution



7

Questions

- Comment les programmes écrits avec des langages de haut niveau sont traduits vers un langage interprétable par le matériel?
 - Comment le matériel exécute le programme?
 - Quelle est l'interface entre le matériel et le software?
 - Comment le logiciel « pilote » le matériel pour exécuter les fonctions attendues?
 - Qu'est ce qui détermine les performances d'un système ou d'un programme?
 - Quelles techniques peuvent être mises en œuvre pour augmenter les performances du système?

8

Plan du cours

- Organisation des processeurs
 - Cycle d'instruction
 - Les composants du processeur
 - Quelques architectures
- Instructions: le langage du processeur
 - Opérations
 - Modes d'adressage
- Assembleur
- Gestion des interruptions
- Hiérarchie mémoire
 - Mémoire Cache
 - Mémoire virtuelle

9

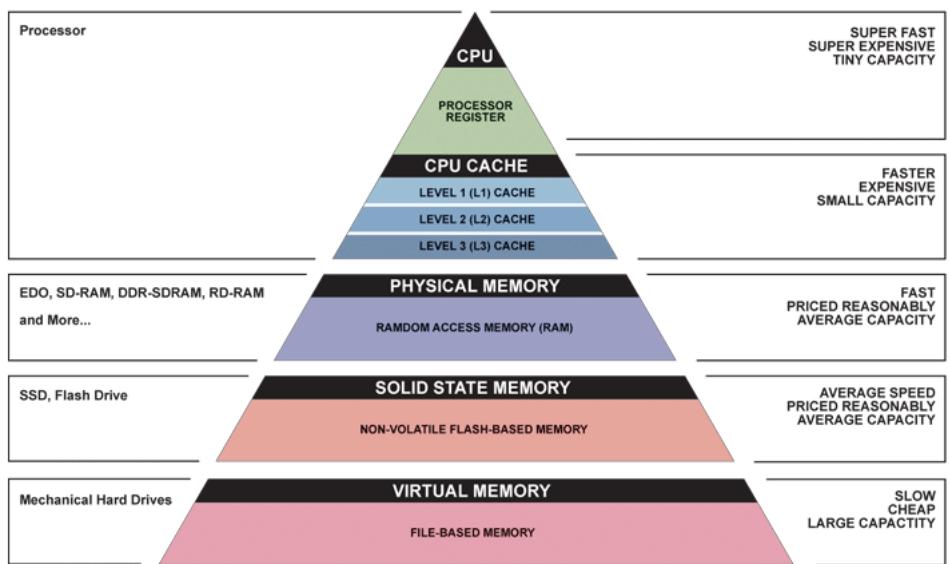
Organisation

- 10 cours magistraux
- 5 séances de travaux dirigés
- 3 séances de TP:

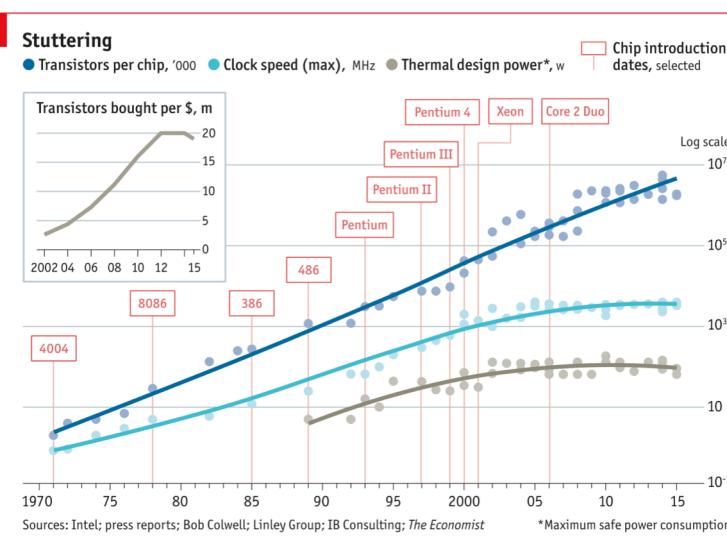
=> Au 2nd semestre pour les EIS 6 séances de programmation embarquée sur cible STM32

10

Hiérarchie Mémoire



Evolution des architectures



Les données numériques

- Les processeurs représentent les données comme des valeurs binaires
- Elément de base: bit
 - 2 valeurs possibles 0 ou 1
 - Facilement stockée communiquée ou manipulée par l'élément matériel
- On utilise plusieurs bits pour représenter des informations plus complexes
 - Byte/octet: 8 bits (256 valeurs différentes)
 - Word/mot 4 butes 2^{32} valeurs différentes
 - Text files, bases de données... (bcp de bytes)
 - Programme à exécuter

13

Conversion des nombres binaires

binaire → décimal

$$1001010_2 = ?_{10}$$

binaire	Valeur décimale
0	$0 \times 2^0 = 0$
1	$1 \times 2^1 = 2$
0	$0 \times 2^2 = 0$
1	$0 \times 2^3 = 8$
0	$0 \times 2^4 = 0$
0	$0 \times 2^5 = 0$
1	$1 \times 2^6 = 64$
	$\Sigma = 74_{10}$

14

Héxadécimal

- Problème: beaucoup de digits
 - e.g. $7643_{10} = 1110111011011_2$
- Solutions:
 - Regroupement: 1 1101 1101 1011₂
 - Héxadecimal: 1DDB₁₆
 - Octal: 1 110 111 011 011₂
16733₈

Binaire	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

15

Les grands nombres

- Décimal
- Binaire (IEC)

Symbol	nom	facteur	Valeur
K	kilo	10^3	1000
M	Mega	10^6	1000,000
G	Giga	10^9	1000,000,000
T	Tera	10^{12}	1000,000,000,000

Symbol	nom	facteur	Value
Ki	Kibi	2^{10}	1024
Mi	Mébi	2^{20}	1048,576
Gi	Gibi	2^{30}	1073,741,824
Ti	Tébi	2^{40}	1099,511,627,776

<https://en.wikipedia.org/wiki/Byte>

16

Architecture des Processeurs

Principes de Fonctionnement

17

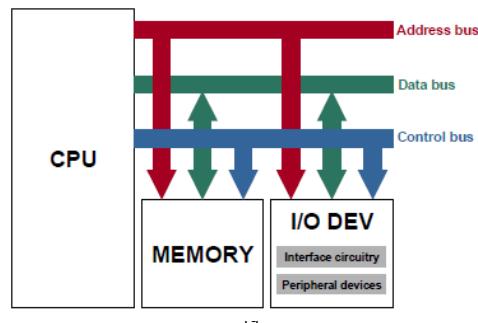
Le modèle de Von Neuman

- L'instruction est le plus petit élément spécifié dans un programme
- Le processeur est constitué de 5 éléments:
 - Processing unit (CPU)
 - Mémoire
 - Entrée
 - Sortie
 - Unité de contrôle
- Le programme est en mémoire avec les données

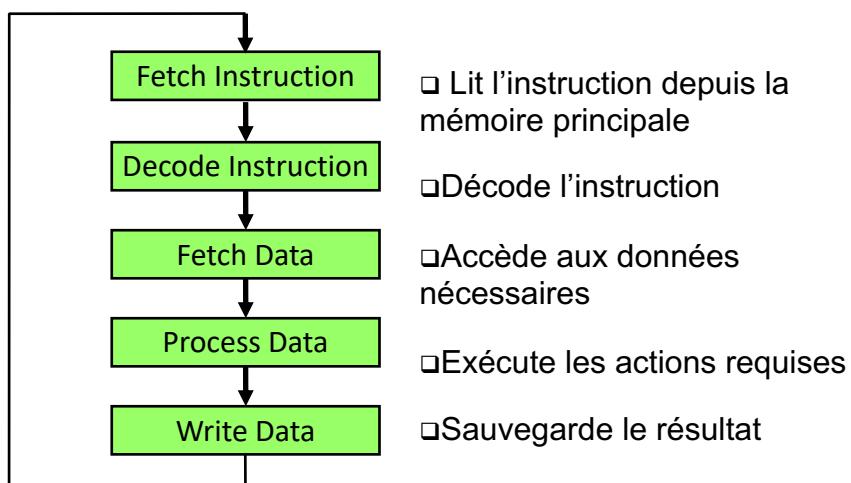
18

Organisation des processeurs

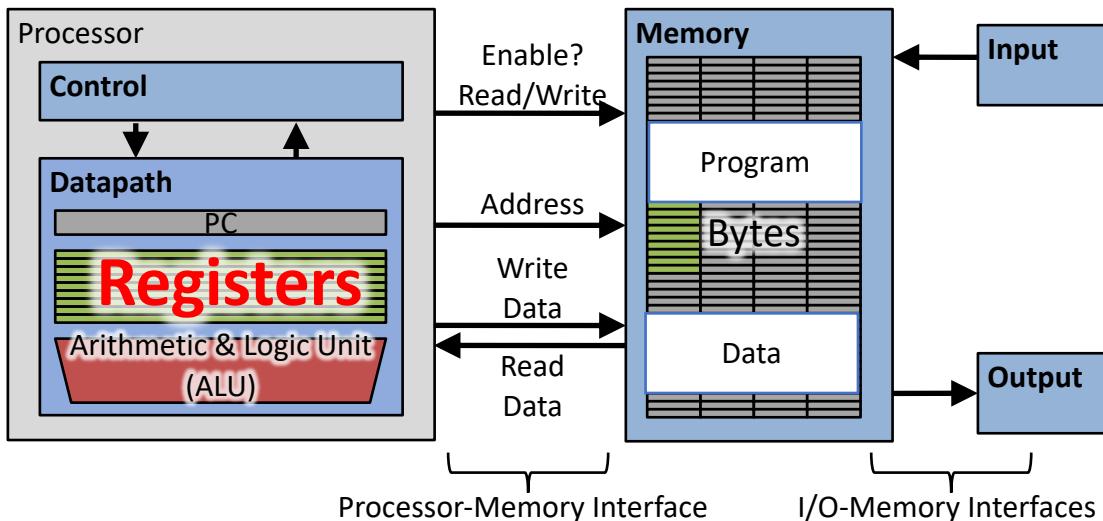
- Mémoire:
 - Contient les données et les instructions nécessaires à l'exécution du programme
- Processeur:
 - Interprète et exécute les instructions du programme
- I/O :
 - Assurent la communication entre le processeur et les éléments externes
- Bus:
 - Assure l'accès à la circuiterie du CPU



Cycle Instruction simplifié



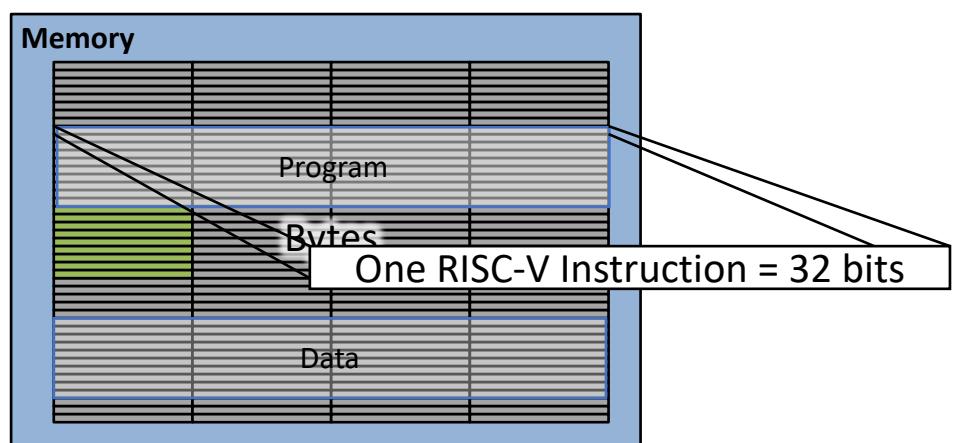
Architecture classique



Credits UC Berkeley's CS61C.

21

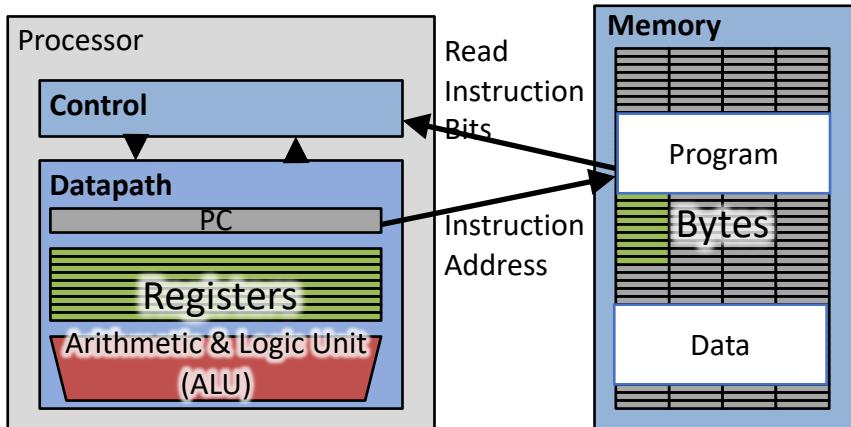
Le programme en mémoire



9/23/20

22

Exécution d'un programme



- Le PC (program counter) est un registre interne du processeur qui contient l'adresse mémoire de l'instruction à exécuter.
- L'instruction est chargée depuis la mémoire dans le processeur elle est ensuite décodée puis executée. Le PC est ensuite incrémenté (+4) pour aller à l'instruction suivante en mémoire.

23

Jeu d'instruction, Instruction Set Architecture (ISA)

- Rôle du CPU (*Central Processing Unit, aka Core*): exécuter des *instructions*
- Instructions: opérations basiques du CPU
 - Les opérations (opcode) effectuent des traitements sur des opérandes pour former des séquences
 - Des instructions supplémentaires permettent de modifier les séquences
- Les CPU appartiennent à des “familles”, chaque famille ayant son propre jeu d’instructions)
- Le jeu d’instruction particulier à un CPU est défini par l’ISA: *Instruction Set Architecture (ISA)*
 - Exemples: ARM, Intel x86, MIPS, RISC-V, IBM/Motorola PowerPC (ancien Mac), Intel IA64, ...

24

Instruction Set Architectures

- A l'origine: De nombreuses instructions dédiées à des opérations complexes
 - Les architecture VAX proposaient une instruction pour la multiplication polynomiale!
- Approche RISC (Cocke IBM, Patterson UCB, Hennessy Stanford, 1980s) – *Reduced Instruction Set Computing (Jeu d'instruction réduit)*
 - Garder le jeu d'instruction simple et petit afin d'avoir des implémentations matérielles rapide.
 - Laisser le “logiciel” réaliser les opérations complexes en composant avec des opérations simples.

25

Un cas d'étude: l'architecture RISC-V



- Initiation à l'assembleur
 - Les registres
 - Les instructions RISC-V
 - Accès à la mémoire
 - Opérations logiques
 - Séquencement

26

RISC-V Green Card (sur chamilo)

27

Qu'est ce que RISC-V?

- 5^{eme} génération de core RISC développé par UC Berkely
 - Une spécification d'ISA libre
 - De nombreuses implémentations disponibles (libres et propriétaires)
 - Un écosystème dynamique
 - Adapté à différentes types d'application (du microcontrôleur au super calculateur)
 - Des variantes 32-bit, 64-bit, et 128-bit
 - Le standard est maintenu par une fondation à but non lucratif: RISC-V Foundation

23/09/2020

28

Foundation Members (60+)

Platinum:

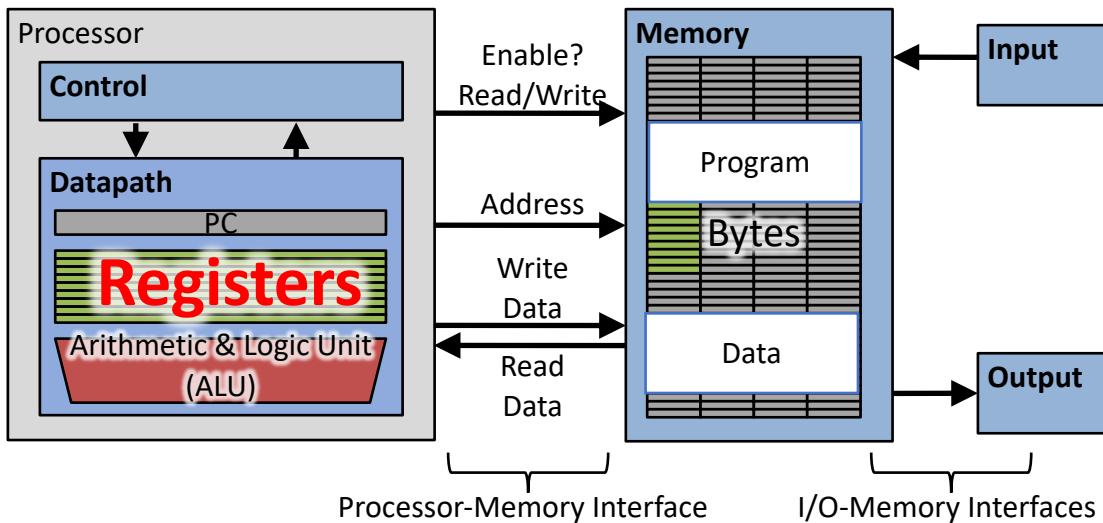
Gold, Silver, Auditors:

29

Les variables en assembleur: Les registres

- A l'inverse des langages de haut niveau comme le C ou le Java, le langage assembleur ne manipule pas des variables comme on en a l'habitude
 - Langage plus primitif et plus proche du matériel
- Les opérandes en assembleur sont placés dans des registres
 - Un nombre limité de registres (implémentés dans le matériel) pour stocker des données à manipuler
- Bénéfice: Puisque les registres sont internes au CPU, l'accès aux données est très rapide (< 1 ns)

Les registres dans le processeur



Credits UC Berkeley's CS61C.

31

Les registres du processeur RISC-V

- Il y a un nombre limité de registres
 - Solution: le code RISC-V doit être écrit avec précaution afin d'optimiser l'utilisation des registres.
- 32 registres dans l'architecture RISC-V, référencé par un index **x0 – x31**
 - Les registres ont également des noms génériques (détailés plus tard)
 - Pourquoi 32?
 - Dans la version RV32, chaque registre contient 32 bits(word) (architecture 32 bits RISC-V, il existe des implémentations 64 bits et 128 bits)
 - **x0** est spécial, il contient toujours la valeur 0
 - Dans les faits seuls 31 registres peuvent stocker des valeurs arbitraires

32

Les variables C, Java vs. les registres

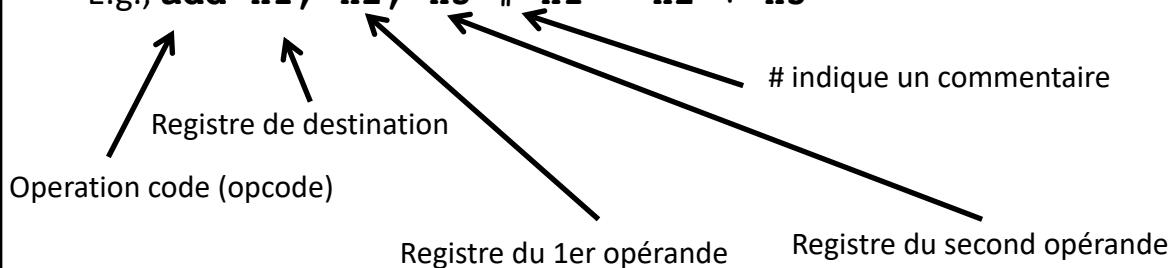
- En C (et autres langages de programmation):
 - On déclare les variables et on leur donne un type
 - Exemple: `int fahr, celsius;`
`char a, b, c, d, e;`
 - Chaque variable ne peut représenter une valeur du type issu de la déclaration (e.g., on ne traite pas un int de la même façon qu'un char)
- En assembleur:
 - Les registres n'ont pas de type;
 - L' **Opération (opcode)** détermine comment le contenu des registres doit être interprété.

33

Syntax assembleur des instructions RISC-V

- Les instructions sont formées d'un opcode et d'opérandes

• E.g., `add x1, x2, x3 # x1 = x2 + x3`



34

Addition et Soustraction d'entier

- Addition en assembleur
 - Exemple: `add x1, x2, x3` (RISC-V)
 - Equivalent à: $a = b + c$ (C)
variables C \Leftrightarrow registres RISC-V :
 $a \Leftrightarrow x1, b \Leftrightarrow x2, c \Leftrightarrow x3$
- soustraction en assembleur
 - Exemple: `sub x3, x4, x5` (RISC-V)
 - Equivalent à : $d = e - f$ (C)
variables C \Leftrightarrow registres RISC-V :
 $d \Leftrightarrow x3, e \Leftrightarrow x4, f \Leftrightarrow x5$

35

Addition et Soustraction d'entier

- Comment exécuter la déclaration C suivante?
 $a = b + c + d - e;$
- On divise en plusieurs opérations
`add x10, x1, x2 # a_temp = b + c`
`add x10, x10, x3 # a_temp = a_temp + d`
`sub x10, x10, x4 # a = a_temp - e`
- Une simple ligne de C est souvent traduite par plusieurs instructions RISC-V

36

Les valeurs immédiates

- Les valeurs immédiates sont des constantes numériques
- On les retrouve souvent dans des codes, il y a donc des instructions dédiées.
- Add Immédiat:

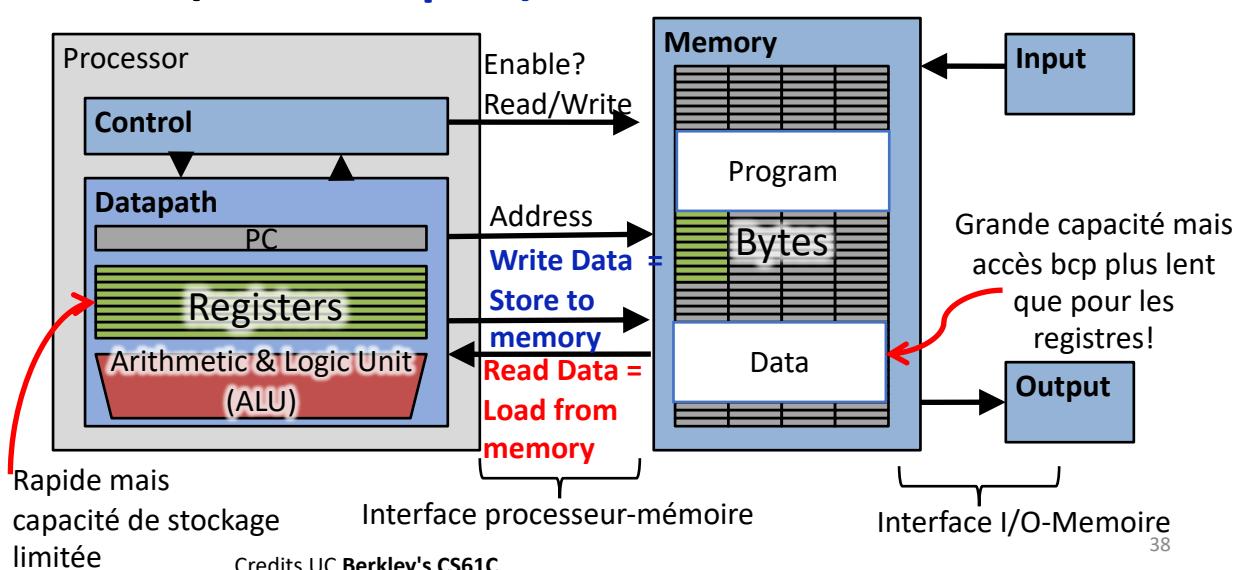

```
addi x3,x4,-10 (RISC-V)
f = g - 10           (C)
```

Les registres RISC-V `x3, x4` sont associés aux variables C: **f et g**
- La syntaxe est similaire à l'instruction addition sauf que le dernier opérande est une valeur au lieu d'un registre.

```
add x3,x4,x0 (RISC-V)
f = g           (C)
```

37

Transfert de données: Load /Store depuis/vers la mémoire



Adressage Mémoire

- La granularité d'adressage est l'octet:
 - Chaque octet est adressé
- Chaque mot (word) est donc séparé de 4 adresses
 - L'adresse d'un mot est identique à celle de l'octet de poids faible qui compose le mot (i.e. convention Little-endian)

Octet de poids faible dans un mot

...
15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

31 24 23 16 15 8 7 0

39

Transfert depuis la Mémoire vers un registre

- Code C

```
int A[100];
g = h + A[3];
```

- Load Word (**lw**) RISC-V:

```
lw x10,12(x13) # Reg x10 prends A[3]
add x11,x12,x10 # g = h + A[3]
```

Note: x13 – base register (pointeur vers A[0])
 12 – offset (en octets)

40

Transfert depuis un registre vers la mémoire

- Code C

```
int A[100];
A[10] = h + A[3];
```

- Utilisation de store word (**sw**) RISC-V:

```
lw x10,12(x13) # Temp reg x10 prends A[3]
add x10,x12,x10 # Temp reg x10 prends h + A[3]
sw x10,40(x13) # A[10] = h + A[3]
```

Note: **x13** – register de base (pointeur)
12, 40 – offsets (bytes)

x13+12 et x13+40 doivent être des multiples de 4

41

Charger et Stocker des Octets

- En plus des transferts de mot (**lw, sw**), RISC-V a également des instructions pour le transfert d'octets :
 - load byte: **lb**
 - store byte: **sb**
- Même format que **lw, sw**
- E.g., **lb x10,3(x11)**

42

Quizz

```
addi x11,x0,0x3f5
sw x11,0(x5)
lb x12,1(x5)
```

Quelle est la valeur contenue dans x12?

43

RISC-V Opérations logiques

- Utiles pour manipuler des ensembles (bits) d'un mot
 - e.g. un char dans un mot (8 bits)
- Opérations pour assembler des bits dans un mot
- ce sont les opérations logiques

Logical operations	C operators	Java operators	RISC-V instructions
Bit-by-bit AND	&	&	and
Bit-by-bit OR			or
Bit-by-bit XOR	^	^	xor
Shift left logical	<<	<<	sll
Shift right logical	>>	>>	srl

9/23/20

44

Décalage logique

- Décalage logique à gauche: **slli** `x11,x12,2`
`#x11=x12<<2`
 - On stocke dans `x11` la valeur de `x12` décalée de 2 bits vers la gauche, **insertion de 0** à droite
 - Avant: `0000 0002`_{hex}
`0000 0000 0000 0000 0000 0000 0010`_{two}
 Après: `0000 0008`_{hex}
`0000 0000 0000 0000 0000 0000 1000`_{two}

Quelle est l'opération arithmétique correspondante?

- Décalage logique à droite : **srlt** opération opposée; `>>`
 - Insertion de 0 à gauche, décalage des bits vers la droite

45

Instructions conditionnelles

- L'opération effectuée dépend du calcul en cours
- Similaire au « if ... else... »
- RISC-V: l'instruction *if* est
`beq register1, register2, L1`
 signifie: va à l'instruction indiquée par le label L1
 si (value in register1) == (value in register2)
sinon, va à la prochaine instruction
- **beq** signifie *branch if equal*
- Instructions du même type: **bne** pour *branch if not equal*

9/23/20

46

Les branchements

- **Branchement** – modifie le flot de contrôle du programme
- **Branchement conditionnel** – modifie le flot de contrôle du programme en fonction d'un résultat de comparaison
 - branch if equal (**b_{eq}**) or branch if not equal (**b_{ne}**)
 - branch if less than (**b_{lt}**) and branch if greater than or equal (**b_{ge}**)
- **Branchement inconditionnel** –
 - RISC-V instruction: *jump* (**j**)

47

Outline

- Assembly Language
- RISC-V Architecture
- Registers vs. Variables
- RISC-V Instructions
- C-to-RISC-V Patterns
- And in Conclusion ...

9/23/20

48

Exemple *if*

- A la compilation les registres suivant contiennent les variables du programme

```
f → x10      g → x11      h → x12
i → x13      j → x14
```

```
if (i == j)          bne x13,x14,Exit
    f = g + h;      add x10,x11,x12
                      Exit:
```

49

Exemple *if*

- A la compilation les registres suivant contiennent les variables du programme

```
f → x10      g → x11      h → x12
i → x13      j → x14
```

```
if (i == j)          bne x13,x14,Else
    f = g + h;      add x10,x11,x12
else                  j Exit
    f = g - h;  Else:   sub x10,x11,x12
                      Exit:
```

9/23/20

50

Magnitude Compares in RISC-V

- Until now, we've only tested equalities (`==` and `!=` in C);
General programs need to test `<` and `>` as well.
- RISC-V magnitude-compare branches:
“Branch on Less Than”

Syntax: `blt reg1,reg2, label`
 Meaning: `if (reg1 < reg2) // treat registers as signed integers
 goto label;`
- “Branch on Less Than Unsigned”

Syntax: `bltu reg1,reg2, label`
 Meaning: `if (reg1 < reg2) // treat registers as unsigned integers
 goto label;`

9/23/20

51

Boucle C en Assembleur RISC-V

```

int A[20];
int sum = 0;
for (int i=0; i<20; i++)
    sum += A[i];
                                add x9, x8, x0 # x9=&A[0]
                                add x10, x0, x0 # sum=0
                                add x11, x0, x0 # i=0
Loop:
                                lw x12, 0(x9) # x12=A[i]
                                add x10,x10,x12 # sum+=
                                addi x9,x9,4 # &A[i++]
                                addi x11,x11,1 # i++
                                addi x13,x0,20 # x13=20
                                blt x11,x13,Loop

```

52

Modèle d'exécution

Modèle d'exécution

- Modèles d'exécution (n,m)
- n : nombre d'opérandes par instruction
- m : nombre d'opérandes mémoire par instruction
- Les différents modes
 - RISC : (3,0)
 - Instructions de longueur fixe
 - Load et Store : seules instructions mémoire
 - IA-32 : (2,1)
 - Pile (0,0)
 - Tous les opérandes sont accédés via la pile

Modèles d'exécution (3,0)

- Registre Registre
 - Aussi appelé “Load/store” architecture
 - 3 opérandes possibles, seulement depuis les registres
 - Opération spécifique “load & store” depuis la mémoire.
- Exemple: $C = A + B$
 - Load R1,A / Load R2,B / add R3, R1, R2 / store C,R3

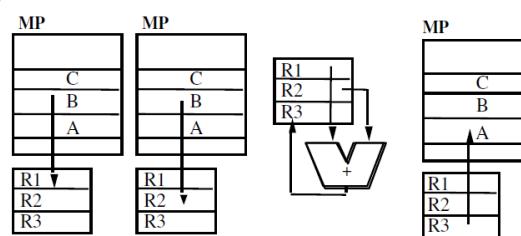
55

Modèles d'exécution (3,0)

- Instruction de longueur fixe
- Seules les instructions load, store accèdent à la mémoire

LOAD-STORE (3,0)

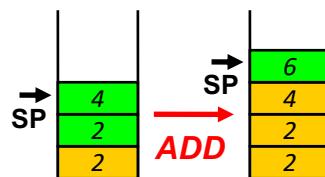
Load	R1	$@B$
Load	R2	$@C$
Add	R3	$R2 + R1$
Store	R3	$@A$



56

Modèles d'exécution (0,0)

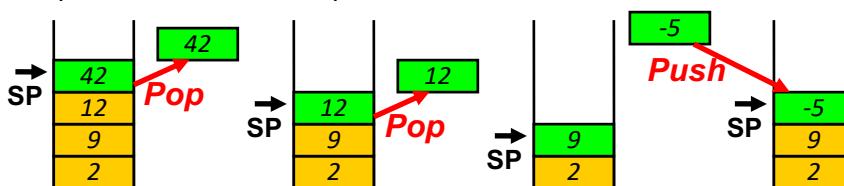
- Pile
 - Toutes les opérandes sont en pile
 - Résultat stocké en pile
 - Problème: la mémoire est lente...
 - Exemple: JVM



57

Passage par pile

- Données échangées par la pile:
 - Deux opérations: Push et Pop



- Présent dans toutes les architectures
 - Utilisés pour les adresses et les données
 - Pointeur de pile (registre spécifique)
 - Instructions dédiées

58

Modèle d'exécution (3,3)

- Mémoire Mémoire
 - Toutes les opérandes en mémoire
 - 3 opérandes par instruction
- Exemple: C=A+B;
 - Add C,A,B
- Avantages/ inconvénients
 - Le plus compact, pas de registres inutilisés
 - Instructions irrégulières format/exécution, mémoire slow
- Exemple: VAX

59

Modèle d'exécution (2,1)

- Mémoire Registre
 - Un opérande en mémoire, l'autre en registre
- Exemple: C=A+B
 - Load R1,A; Add R1,B; store C,R1;
- Avantages/inconvénients
 - Bonne densité
 - Un opérande perdu
- Intel IA-32, Motorola 68k

60

Modèle d'exécution (2,1)

REGISTRE-MÉMOIRE (2,1)

