

Conception de circuits numériques et architecture des ordinateurs

Équipe pédagogique :

Marie Badaroux (TD), Julie Dumas (TD+TP), Florence Maraninchi (TD+TP),
Olivier Muller (TD+TP), **Frédéric Pétrot** (CM+TD), Pierre Ravenel (TD+TP)
Eduardo Tomasi Ribeiro (TP), Lionel Rieg (TD+TP),
Sébastien Viardot (TD+TP Apprentis)



Année universitaire 2024-2025

- C1 Codage des nombres en base 2, logique booléenne, portes logiques, circuits combinatoires
- C2 Circuits séquentiels
- C3 Construction de circuits complexes
- C4 Micro-architecture et fonctionnement des mémoires
- C5 Machines à état
- C6 Synthèse de circuits PC/PO
- C7 Optimisation de circuits PC/PO
- C8 **Interprétation d'instructions - 1**
- C9 Interprétation d'instructions - 2
- C10 Interprétation d'instructions - 3
- C11 Introduction aux caches

Plan détaillé du cours d'aujourd'hui

1 Contexte et préambule

- Contexte
- Préambule

2 Processeur et jeu d'instruction

- Introduction
- Jeu d'instructions
- Modes d'adressage

Plan

1 Contexte et préambule

- Contexte
- Préambule

2 Processeur et jeu d'instruction

- Introduction
- Jeu d'instructions
- Modes d'adressage

Comment une suite d'octets peut-elle contrôler le monde ?

Les processeurs sont partout : ordinateurs, mais aussi télécommunications, voitures, avions, centrales nucléaires, drones, missiles, satellites, t-shirts, écouteurs, lave linges, lampes, routes, serrures, ...

Mise en contexte

Langages de programmation

Assurent une certaine portabilité du code

Reposent sur des bibliothèques de fonctions conscientes du matériel

Génération des binaires

Compilateur : génère du code textuel optimisé adapté à la machine

Assembleur : transforme le code textuel en code binaire

Éditeur de lien : fusionne les fichiers binaires en un exécutable

Système d'exploitation

Donne accès aux services du matériel

Procède au chargement des fichiers exécutables en mémoire

Processeur

Lit l'instruction à l'adresse du reset dès la mise sous tension

Interprète les instructions jusqu'à extinction de l'alimentation

Préambule

Retour sur PC/PO

Algorithme effectuant l'addition ou la soustraction de données en fonction d'une entrée *pm*

```

1 while true do
2   switch pm do
3     case 0 do  $S := I_0 + I_1$ ;
4     case 1 do  $S := I_0 - I_1$ ;
5   end switch
6 end while

```

- I_0 et I_1 , entrées de données du circuit
- *pm*, entrée qui contrôle si l'on fait plus ou moins

Comment indiquer la séquence d'entrée ?

En la stockant dans un tableau de bascules externes

Préambule

Nouvelles variables

- $t[n]$, tableau de n bits représentant les opérations successives à effectuer, externe à la PO
- C , compteur initialement à 0, permettant d'indiquer le bit à lire dans le tableau, interne à la PO

```

1 while true do
2   |  $op := t[C]$ ;
3   |  $C := C + 1$ ;
4   | switch  $op$  do
5   |   | case 0 do  $S := I_0 + I_1$ ;
6   |   | case 1 do  $S := I_0 - I_1$ ;
7   | end switch
8 end while
```

- I_0 et I_1 , entrées de données du circuit
- op , opération contenant le bit d'indice C lu à partir du tableau externe

Préambule

Modifications

On désire à présent ajouter multiplication et division, et faire ces calculs à l'aide de 8 registres internes de 16 bits

- $op \in \{00 \equiv +, 01 \equiv -, 10 \equiv \times, 11 \equiv \div\}$
- $r[8]$, tableau de 8 registres de 16 bits, interne
- $r[i] := r[j] \text{ op } r[k]$

Comment spécifier ces registres ?

Ajouter leurs indices à chaque opération, pour indiquer registres à utiliser en entrée (source) et en sortie (destination)

Ainsi :

- $t[i]$ contient $2 + 3 \times \log_2 8 = 11$ bits
- | | | | | | | | | | | |
|----|---|-----|---|---|------------------|---|---|------------------|---|---|
| 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| op | | dst | | | src ₁ | | | src ₀ | | |
- $t[i]_{10..9} = op, t[i]_{8..6} = dst, t[i]_{5..3} = src_1, t[i]_{2..0} = src_0$

Préambule

```

1  while true do
2       $i := t[c]$  ;
3       $C := C + 1$  ;
4      switch  $i_{10..9}$  do
5          case 00 do  $r[i_{8..6}] := r[i_{5..3}] + r[i_{2..0}]$  ;
6          case 01 do  $r[i_{8..6}] := r[i_{5..3}] - r[i_{2..0}]$  ;
7          case 10 do  $r[i_{8..6}] := r[i_{5..3}] \times r[i_{2..0}]$  ;
8          case 11 do  $r[i_{8..6}] := r[i_{5..3}] \div r[i_{2..0}]$  ;
9      end switch
10 end while

```

■ i contient l'ensemble des informations permettant :

- de connaître l'opération à exécuter
- d'identifier les opérandes sources et destination de l'opération

Séquence d'opérations

Que l'on peut définir de l'extérieur en chargeant t

t contient un **programme** composé d'**instructions successives**

Plan

1 Contexte et préambule

- Contexte
- Préambule

2 Processeur et jeu d'instruction

- Introduction
- Jeu d'instructions
- Modes d'adressage

Introduction

Problématique :

Qu'est-ce qu'un programme vu du matériel ?

- suite de bits de format *ad-hoc* spécifiant un comportement *le jeu d'instructions*
- « traduction » de ce comportement en commandes matérielles
- but : modification de l'état de la machine

Problématique :

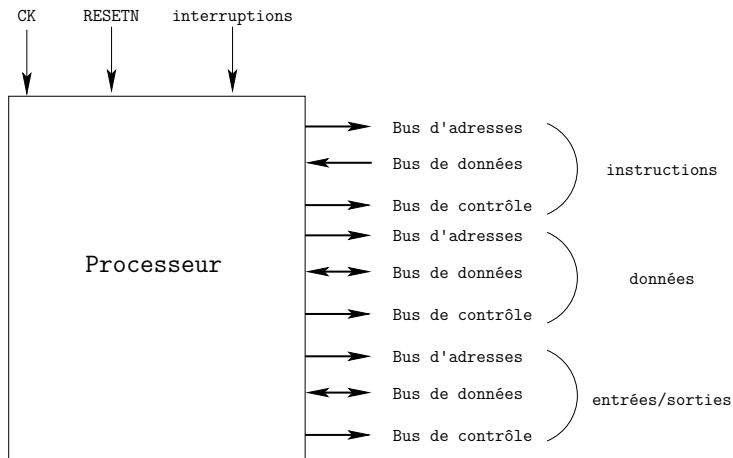
Comment exécuter des programmes ?

Solution :

- décoder les instructions pour connaître leur sémantique
- réaliser les changements induits par l'instruction

Implantation sous forme PC/PO

Moyens de communications



Classification

- Mémoire d'instructions et de données partagées :
architecture de Von Neumann
John Von Neumann, génie Hongrois naturalisé Américain, inventeur de cette structuration, 1903-1957.
- Mémoire d'instructions et de données séparées :
architecture Harvard
Du nom de l'université dans laquelle elle a été définie en 1944.

Contenu de la mémoire

Tout est affaire d'interprétation !

cf cours numéro 1

- données de tous types
- y compris les **instructions**

Instructions

Suite de n octets (plutôt que bits)

- interprétés par le matériel
- précisant :
 - les opérations à effectuer et leurs opérandes
opération `op_dest`, `op_src1`, `op_src2`, ...
 - le séquençement entre instructions
par défaut l'instruction suivante !

Ressources

Opérateurs

Unités fonctionnelles usuelles : ALU ($+$, $-$, \vee , \wedge , \oplus , ...)

Unités plus complexes : \times , \div , entières ou flottantes

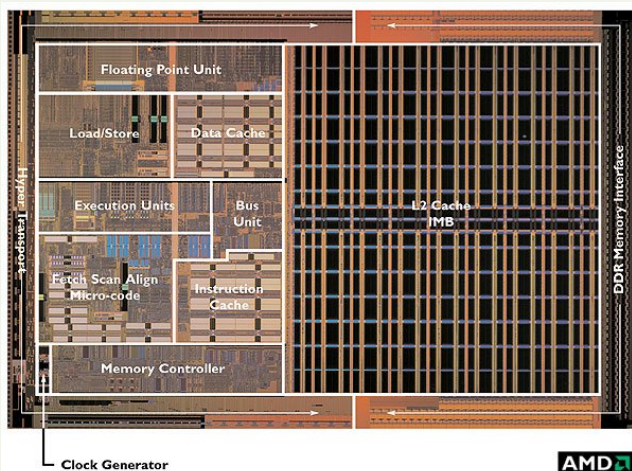
Unités flottantes : \equiv , $\sqrt{}$, \exp , \log , ...

Opérandes

- registres
- cases mémoires

Ressources

Répartition spatiale



Registres

Toujours

- pointeur de programme (*program counter* ou PC)
- instruction
- registre d'état : *flags*, interruptions, exceptions, etc.

Spécifiques, suivant les processeurs

- accumulateur : registre destination des opérations de l'ALU
- pointeur de pile : indique base de la pile
Utile pour les appels de fonctions

Généralistes, dépendant des processeurs

- registres à usage général : organisés en « banc de registres »
- peuvent être utilisés comme accumulateurs et pointeur de pile

ISA : Instruction Set Architecture

ISA : Ensemble exhaustif des instructions supportées

- sémantique : ce que fait l'instruction
- syntaxe : comment spécifier l'instruction et ses opérandes

Mnémonique

- représentation textuelle des instructions
- permet de faire de la programmation dite *assembleur*
- doit être traduit en *langage machine*

Critères de choix dans la définition d'un ISA :

- doit fournir un support efficace aux langages de programmation
- doit pouvoir être implanté efficacement
- régulier et complet

ISA

Différents types d'instructions

Opérations arithmétiques et logiques

`add, sub, sll, sra, inc, mov, ...`

Transfert de données entre registre et mémoire

`ldw, ldb, stw, stb, ...`

Déroutement du programme

Sauts inconditionnels : `jmp, ba`

Sauts conditionnels : `jcc, bcc` (cc : code conditions)

Appel et retour de fonction :

`call, jal`

`return, jr`

ISA : Différents types

Nombre d'opérandes explicites : 0, 1, 2, 3

Pour l'opération op soit \perp

- à pile : op $push(pop \perp pop)$
- accumulateur : $op \text{ addr}$ $acc := acc \perp mem[addr]$
- « 2 adresses » : $op \text{ \%ri, \%rj}$ $\%ri := \%ri \perp \%rj$
- « 3 adresses » : $op \text{ \%ri, \%rj, \%rk}$ $\%ri := \%rj \perp \%rk$

Langage machine

Format binaire codant l'instruction et ses opérandes

Pas de portabilité du code binaire

Chaque processeur (ou famille) possède son ISA propre

- un code pour un type de processeur *ne tourne pas* sur un autre type de processeur

Modes d'adressage

Définit **où** l'on va chercher une instruction

Absolu

`jmp address : $pc := address$`

Relatif à PC

`jmp offset : $pc := pc + offset$`

Registre

`jmp %ri : $pc := \%ri$`

Indirect registre

`jmp (%ri) : $pc := mem[\%ri]$`

Modes d'adressage

Ou un opérande

Registre

```
mov %ri, %rj : %rj := %ri
```

Immédiat

```
mov $12, %rj : %rj := 12
```

Absolu ou direct

```
mov %ri, address : mem[address] := %ri  
mov address, %ri : %ri := mem[address]
```

Indirect

```
mov %ri, (address) : mem[mem[address]] := %ri
```

Modes d'adressage

Indirect registre + déplacement

```
mov offset(%ri), %rj : %rj := mem[%ri + offset]
```

Indirect registre + auto-incrément / auto-décrément

```
mov (%ri)+, %rj : %rj := mem[%ri], %ri := %ri + 1  
mov (%ri)-, %rj : %rj := mem[%ri], %ri := %ri - 1
```

Indirect registre + index + déplacement

```
mov offset(%ri,%rk), %rj : %rj := mem[%ri + %rk + offset]
```

Et encore d'autres !

95% des besoins couverts en 1 instruction par :
registre, immédiat et indirect registre + déplacement

Résumé

Processeur

Charge des données en mémoire considérées comme des instructions

Passe implicitement de l'instruction courante à l'instruction suivante

Stratégie

Interprète des *instructions* décrites sous forme binaire

Agissant sur des opérandes implicites ou explicites

Opérandes en registre et/ou mémoire dont l'adresse peut être calculée de manière complexe