

# Architecture des ordinateurs

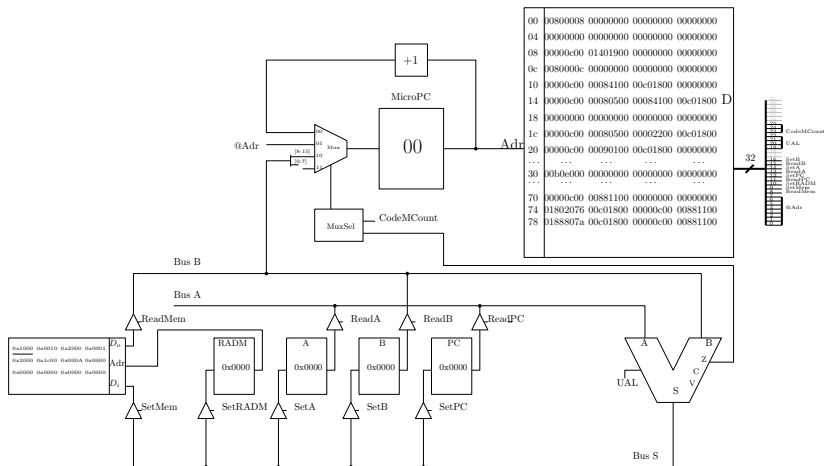
Jérémy Fix

CentraleSupélec

*[jeremy.fix@centralesupelec.fr](mailto:jeremy.fix@centralesupelec.fr)*

2017-2018

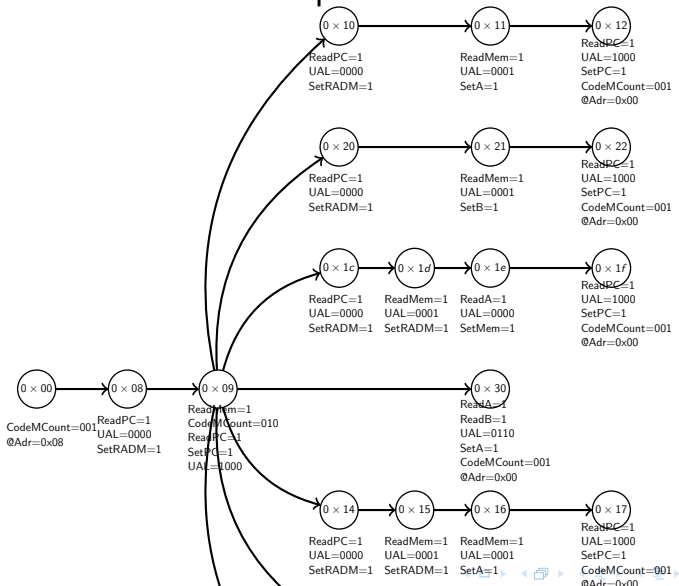
# Petit retour sur l'architecture v0



# Architecture à Jeu d'instructions (ISA)

Code instruction	Nom	Mots	Description
0x0c00	END	1	Fin du programme
0x1000	LDAi	2	Charge la valeur de l'opérande dans le registre A. [A:=opérande]
0x1400	LDAd	2	Charge la valeur dans la RAM pointée par l'opérande dans le registre A. [A:=Mem
0x1c00	STA	2	Sauvegarde en mémoire la valeur du registre A à l'adresse donnée par l'opérande.
0x2000	LDBi	2	Charge la valeur de l'opérande dans le registre B. [B:=opérande]
0x2400	LDBd	2	Charge la valeur dans la RAM pointée par l'opérande dans le registre B. [B:=Mem
0x2c00	STB	2	Sauvegarde en mémoire la valeur du registre B à l'adresse donnée par l'opérande.
0x3000	ADDA	1	Ajoute le contenu des registres A et B et mémorise le résultat dans le registre A. [
0x3400	ADDB	1	Ajoute le contenu des registres A et B et mémorise le résultat dans le registre B. [
0x3800	SUBA	1	Soutstrait le contenu des registres A et B et mémorise le résultat dans le registre A
0x3c00	SUBB	1	Soutstrait le contenu des registres A et B et mémorise le résultat dans le registre B
0x4000	MULA	1	Multiplie le contenu des registres A et B et mémorise le résultat dans le registre A
0x4400	MULB	1	Multiplie le contenu des registres A et B et mémorise le résultat dans le registre B
0x4800	DIVA	1	Divise le contenu du registre A par deux et mémorise le résultat dans A. [A:=A/2]
0x5000	ANDA	1	Calcule un ET logique entre le contenu des registres A et B et mémorise le résulta
0x5400	ANDB	1	Calcule un ET logique entre le contenu des registres A et B et mémorise le résulta
0x5800	ORA	1	Calcule un OU logique entre le contenu des registres A et B et mémorise le résulta
0x5c00	ORB	1	Calcule un OU logique entre le contenu des registres A et B et mémorise le résulta
0x6000	NOTA	1	Mémorise dans A la négation de A. [A:=!A]
0x6400	NOTB	1	Mémorise dans B la négation de B. [B:=!B]
0x7000	JMP	2	Saute inconditionnellement à l'adresse donnée par l'opérande. [PC:=opérande]
0x7400	JZA	2	Saute à l'adresse donnée par l'opérande si le contenu du registre A est nul. [PC:=
0x7800	JZB	2	Saute à l'adresse donnée par l'opérande si le contenu du registre B est nul. [PC :=

# Automate à états finis du séquenceur



# Séquenceur et programme

## Séquenceur

- Sémantique des instructions
- Générique
- Automate à états finis : état dans MicroPC, signaux de contrôle ROM[MicroPC]

## Programme

- Qu'est ce que je veux calculer ?
- Spécifique
- Séquence de codes d'instructions et de données en RAM

# Aperçu de quelques architectures

## La notre

- 2 registres banalisés : A et B
- 2 registres internes : RADM et PC
- adresses sur 16 bits, données sur 16 bits
- 1 mot de 16 bit pour l'instruction, 1 mot pour l'opérande éventuelle
- jeu d'instructions : LDAi, LDBi, ADDA, ANDB, JMP, JZA
- horloge maximale sous logisim : 4 kHz

Beaucoup d'architectures :

https:

[//en.wikipedia.org/wiki/List\\_of\\_instruction\\_sets](https://en.wikipedia.org/wiki/List_of_instruction_sets)

# Aperçu de quelques architectures

## Intel x86 IA32-64

Du Intel 8086(1978) jusqu'au Intel Core i7 (2015)

- 8 registres banalisés 32 bits EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP
- 6 registres de segments 16 bits : CS, DS, SS, ES, FS, GS
- 1 registre de status 32 bits : EFLAGS
- x registres internes : EIP (32 bits)

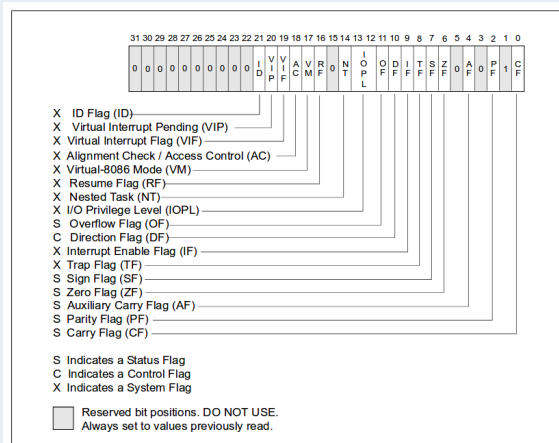
Intel 64 and IA-32 Architectures Software Developer Manuals

<https://software.intel.com/en-us/articles/intel-sdm>

# Aperçu de quelques architectures

## Intel x86 IA32-64

### Le registre de statut EFLAGS





# Aperçu de quelques architectures

## Intel x86 IA32-64: Jeu d'instructions

Beaucoup d'instructions : 1.300 mnémoniques, plusieurs modes d'adressages, ..

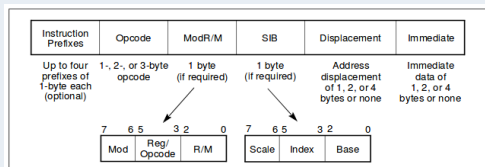


Figure 2-1. IA-32 Instruction Format

Opérations arithmétiques : ADD(0x01, 0x02, 0x03), AND(0x21, 0x22, ..)..

Branchements (in)conditionnels: JMP (0xFF), JZ, JNZ, JG, conditions à partir des bits de statut


Lecture/Ecriture : MOV(0x8B, 0xC7, ...)

# Aperçu de quelques architectures

## ARM (téléphones, tablettes, consoles)

Acorn RISC Machine architecture (1980) jusqu'au ARMv8.3-A (2016)

- 16 registres  $R_i$  32 bits dont :
  - $R0 - R3$  : arguments/résultats pour une routine
  - $R4 - R8$  : variables temporaires
  - $R9$  : Platform register
  - $R10$  : Stack limit pointer
  - $R11$  : Frame pointer
  - $R12$  : registre temporaire
  - $R13$  : Stack Pointer
  - $R14$  : Link register (adresse de retour)
  - $R15$  : Programm Counter
- 1 registre de statut CPSR 32 bits

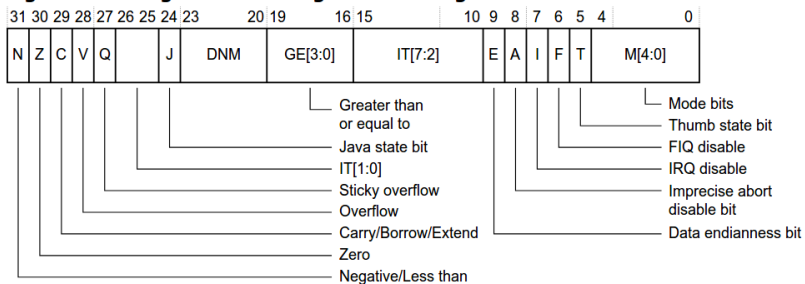
<http://infocenter.arm.com/help/index.jsp> 

# Aperçu de quelques architectures

## ARM (téléphones, tablettes, consoles)

### Le registre CPSR

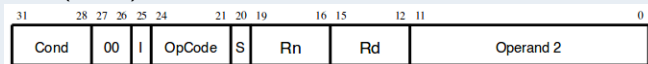
**Figure 3.4. Program status register bit assignments**



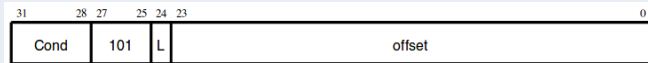
# Aperçu de quelques architectures

## ARM - Jeu d'instructions

Data Processing avec opcode: AND(0000), ADD(0100), SUB(0010)



Branchements : BX, BL, ..



et bien d'autres...

<http://infocenter.arm.com/help/index.jsp>

# La v0 est conçue, programmons la !

## Conception

- Chemin de données
- Jeu d'instructions
- Séquenceur



## Programmation

- Code machine (instructions/données)
- Calcul des adresses "à la main"
  - adresses des branchements?
  - adresses des données ?



# C'est dur et long pour le moment de programmer

## Calculer la suite de Syracuse

$$\forall n \in \mathbb{N}^*, u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

$u_0 = 127; u_n = ?$

⇔

## Code machine

```
1000 007f 1c00 0024 1c00 1000 1400 0024 2000 0001 5000 7400
001b 1400 0024 2000 0003 4000 2000 0001 3000 1c00 0024 1c00
1000 7000 0006 1400 0024 4800 1c00 0024 1c00 1000 7000 0006
```

si si, je vous assure. Donc, c'est dur et long.

# C'est dur et long pour le moment de programmer

## Calculer la suite de Syracuse

$$\forall n \in \mathbb{N}^*, u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_{n+1} + 1 & \text{sinon} \end{cases}$$

$u_0 = 127; u_n = ?$



## Code machine

```
[LDAi 007f STA 0024 STA 1000] [LDA d 0024 LDBi 0001 ANDA  
JZA 001b] [LDA d 0024 LDBi 0003 MULA LDBi 0001 ADDA STA  
0024 STA 1000 JMP 0006] [LDA d 0024 DIVA STA 0024 STA  
1000 JMP 0006]
```

# Comment faire pour simplifier la programmation ?

## Ne plus écrire en code machine<sup>1</sup>

Code machine (arg..)		Assemblage (cool!)		Python (super cool!)
1000 0001		LDAi 1		a=1
2000 0002	← <u>???</u>	LDBi 2	← <u>???</u>	b=2
3000		ADDA		a=a+b

- Langage d'assemblage ? LDAi, STA, ADDA, ..
- Langages de haut-niveau (architecture indépendant) : C++, Python, .. ?

et bien sûr comment faire la conversion vers le langage machine

---

<sup>1</sup>parfois utile tout de même



# Comment faire pour simplifier la programmation ?

## Les procédures (ou fonctions)

Définition : succession d'opérations à exécuter pour accomplir une tâche déterminée [Larousse]

Exemple de Syracuse en Python:  
Sans procédures

```
un = 127
if(un % 2 == 0):
    un = un/2
else:
    un = 3 * un + 1
if(un % 2 == 0):
    un = un/2
else:
    un = 3 * un + 1
if(un % 2 == 0):
    un = un/2
```

Avec procédures

```
def f(u):
    if(u % 2 == 0):
        return u/2
    else:
        return 3 * u + 1

u = 127
u = f(u)
u = f(u)
```

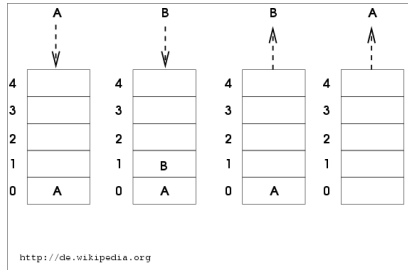


Nous avons donc trois problèmes à résoudre :

- comment partir exécuter la routine ? JMP
- comment passer les arguments à la routine ?
  - Registres dédiés : combien ?? (e.g.  $R0 - R3$  pour ARM)
  - autre chose ?
- comment revenir au programme appelant ?
  - sauvegarder l'adresse de retour dans un registre dédié : *link register* à sauvegarder lors d'appels cascades (A appelle B qui appelle C qui appelle ...), e.g. ARM
  - autre chose ?
- comment récupérer le résultat ?
  - un registre dédié ? (e.g.  $R0$  pour ARM)
  - autre chose ?

Dans notre architecture, on va répondre à ces 3 questions en utilisant une structure particulière : **la pile**

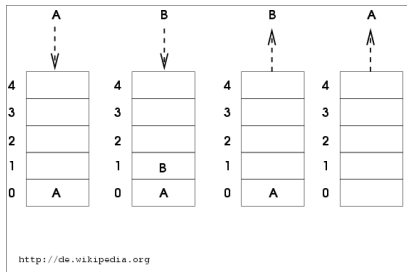
## Procédures, pile et pointeur de pile



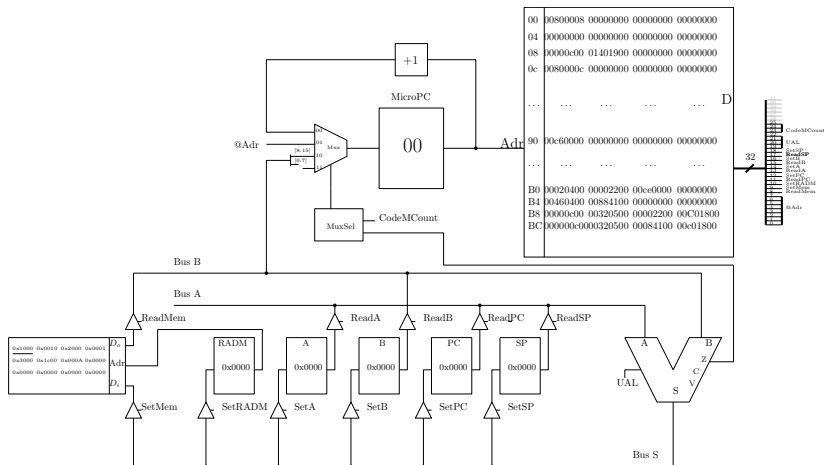
# Spécifications d'une pile

## Une pile en mémoire

- structure de données en mémoire principale (RAM)
- empiler, dépiler une valeur : sommet de pile
- ou est le sommet de pile : registre *Stack Pointer*



# Ajout de SP dans le chemin de données



# Ajout d'instructions de manipulation de la pile

## Spécifications

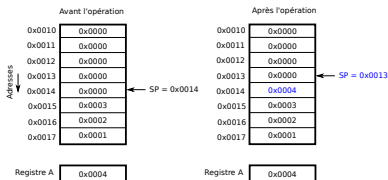
- ou placer la pile en mémoire ?
- quelle est l'adresse du sommet de la pile en mémoire ?
- comment empiler/dépiler, écrire/lire des éléments de la pile ?

## Instructions particulières

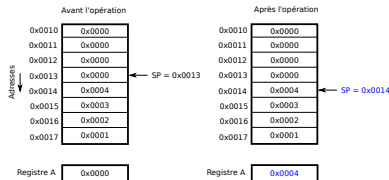
- pour manipuler le registre SP : LDSPi, STSP, INCSP, DECSP
- pour empiler/dépiler : PUSH{A,B}, POP{A, B}
- pour lire/écrire relativement à SP : POKE{A,B}, PEEK{A,B}

# PUSH, POP, POKE, PEEK, Hum?!

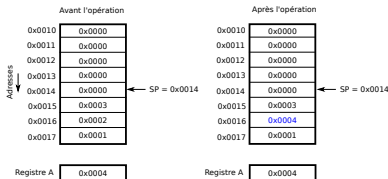
## Exemple de PUSHA



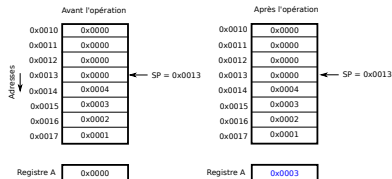
## Exemple de POPA



## Exemple de POKEA 0x0002



## Exemple de PEEKA 0x0002



→ Machines à états (b0, b4, b8, bc)?



# Utilisons la pile pour passer des arguments

```
1: function SOMME( $N$ )  
2:   Soient  $i$ ,  $res$  deux variables locales  
3:    $i \leftarrow N - 1$   
4:    $res \leftarrow 0$   
5:   while  $i \neq 0$  do  
6:      $res \leftarrow res + i$   
7:      $i \leftarrow i - 1$   
8:   return  $res$   
9: function MAIN  
10:  somme(3)
```

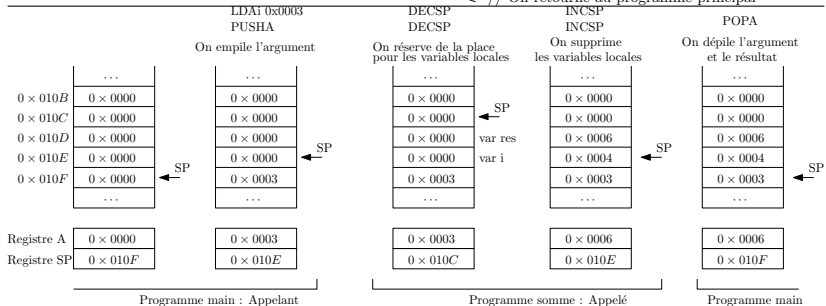
# Somme: Une première tentative

```
main : // On initialise le registre SP
LDSPi 0x010F
// On empile l'argument
LDAi 0x0003
PUSHA
Appel de somme
// On dépile l'argument
POPA
```

JMP

???

```
somme : // On réserve de la place pour
// les variables locales i et res
DECSP
DECSP
// On effectue les calculs de la routine
// en utilisant PEEKA 0x0001 et PEEKA 0x0002
// pour accéder aux variables locales
// en utilisant PEEKA 0x0003 pour l'argument
// On dépile les variables locales
INCSP
INCSP
// On retourne au programme principal
```



# Appel et retour de routines

## Spécifications

- Quand on part exécuter le code d'une procédure, il faut sauvegarder là où retourner (marque page)
- Quand on termine une procédure, il faut poursuivre le programme appelant

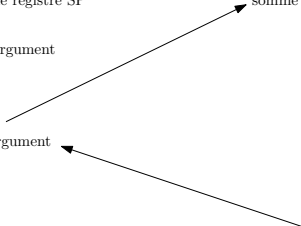
## Instructions particulières

- CALL (0xA000): "CALL op" Empile l'adresse de la **prochaine** instruction et branche
- RET (0xA800): "RET" Dépille dans PC l'adresse de retour

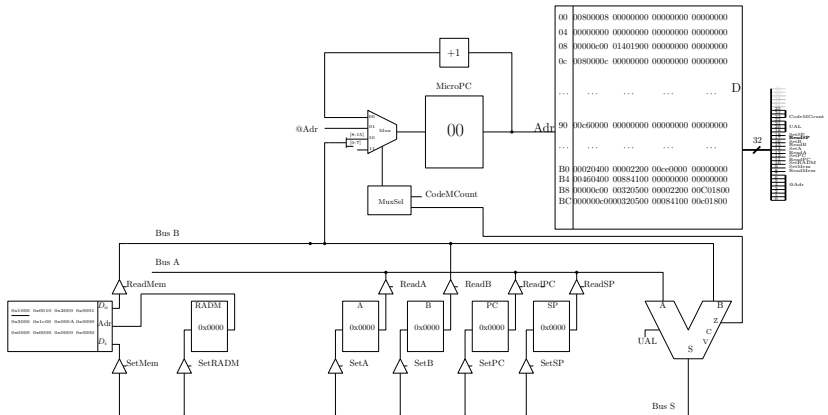
# Réalisation de ces instructions

```
main : // On initialise le registre SP
      LDSPI 0x010F
      // On empile l'argument
      LDAi 0x0003
      PUSHA
      CALL somme
      // On dépile l'argument
      POPA

      somme : // On réserve de la place pour
             // les variables locales i et res
             DECSP
             DECSP
             // On effectue les calculs de la routine
             // en utilisant PEEKA 0x0001 et PEEKA 0x0002
             // pour accder aux variables locales
             // en utilisant PEEKA 0x0003 pour l'argument
             // On dépile les variables locales
             INCSP
             INCSP
             // On retourne au programme principal
             RET
```



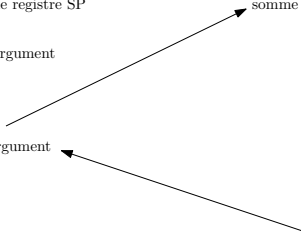
# Réalisation de ces instructions



# Réalisation de ces instructions

```
main : // On initialise le registre SP
      LDSPI 0x010F
      // On empile l'argument
      LDAi 0x0003
      PUSHA
      CALL somme
      // On dépile l'argument
      POPA

      somme : // On réserve de la place pour
             // les variables locales i et res
             DECSP
             DECSP
             // On effectue les calculs de la routine
             // en utilisant PEEKA 0x0001 et PEEKA 0x0002
             // pour accéder aux variables locales
             // en utilisant PEEKA 0x0003 pour l'argument
             // On dépile les variables locales
             INCSP
             INCSP
             // On retourne au programme principal
             RET
```



et le résultat au fait? La pile !

# Somme : une deuxième tentative

```
main : // On initialise le registre SP
```

```
LDSPi 0x0020
```

```
// On réserve de la place pour le résultat
```

```
DECSP
```

```
// On empile l'argument
```

```
LDAi 0x0003
```

```
PUSHA
```

```
CALL somme
```

```
// On dépile l'argument
```

```
POPA
```

```
// On récupère le résultat
```

```
POPA
```

```
somme : // On réserve de la place pour  
// les variables locales i et res
```

```
DECSP
```

```
DECSP
```

```
// On effectue les calculs de la routine
```

```
// en utilisant PEEKA 0x0001 et PEEKA 0x0002
```

```
// pour accéder aux variables locales
```

```
// en utilisant PEEKA 0x0003 pour l'argument
```

```
// On dépile les variables locales
```

```
INCSP
```

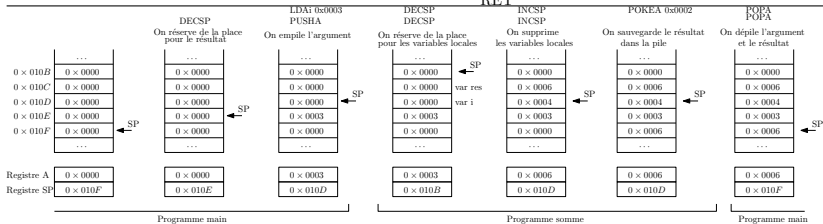
```
INCSP
```

```
// On sauvegarde le résultat dans la pile
```

```
POKEA 0x0002
```

```
// On retourne au programme principal
```

```
RET
```



# Déroulement d'un appel de routine

## Le programme appelant

- réserve de la place pour le résultat (DECSP)
- empile les arguments (PUSH)
- sauvegarde la valeur PC après lecture de l'adresse de la routine et branche sur la routine (CALL)

## Programme appelé

- lit les arguments dans la pile (PEEK),
- calcule son résultat éventuel et le sauvegarde dans la pile (POKE)
- retourne au programme appelant (RET) : le registre SP doit être restauré!

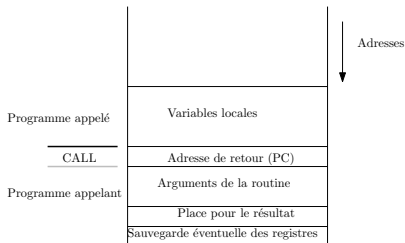
## Programme appelant

- dépile les arguments (INCSP ou POP)
- dépile le résultat (POP) et se poursuit

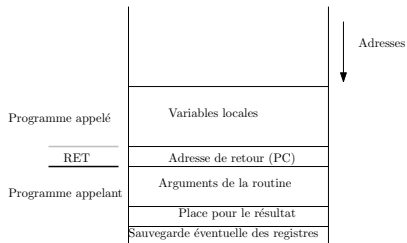


# Qui met/enlève quoi dans la pile ?

Qui met quoi dans la pile ?



Qui enlève quoi dans la pile ?



## Attention!

Dans cette version d'architecture, les accès PEEK, POKE sont relatifs au sommet de pile !

Autre possibilité : registre Base Pointer (BP) / Frame Pointer (FP)  
 Pour restaurer le registre SP, on pourrait aussi utiliser BP/FP

C'est parti pour le code machine de :

```
1: function SOMME( $N$ )  
2:   Soient  $i$ ,  $res$  deux variables locales  
3:    $i \leftarrow N - 1$   
4:    $res \leftarrow 0$   
5:   while  $i \neq 0$  do  
6:      $res \leftarrow res + i$   
7:      $i \leftarrow i - 1$   
8:   return  $res$   
9: function MAIN  
10:  somme(3)
```

## Procédures

Les procédures permettent :

- de factoriser le code
- d'éviter des bugs puisqu'on ne réécrit pas plusieurs fois le "même" code
- de rendre le code plus compact

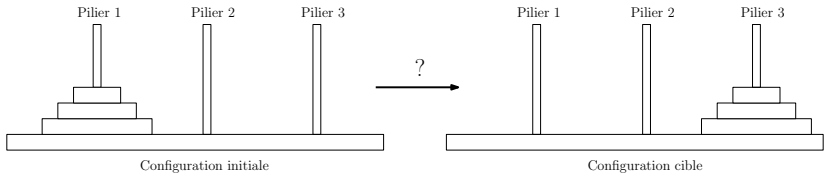
mais avec un petit surcoût à l'exécution (parfois inévitable)

## Pile

La pile permet :

- de passer des arguments à une procédure
- de récupérer le résultat d'une procédure
- de sauvegarder l'adresse de retour d'une procédure
- de sauvegarder des variables locales à la procédure

# Des fonctions récursives : Hanoï



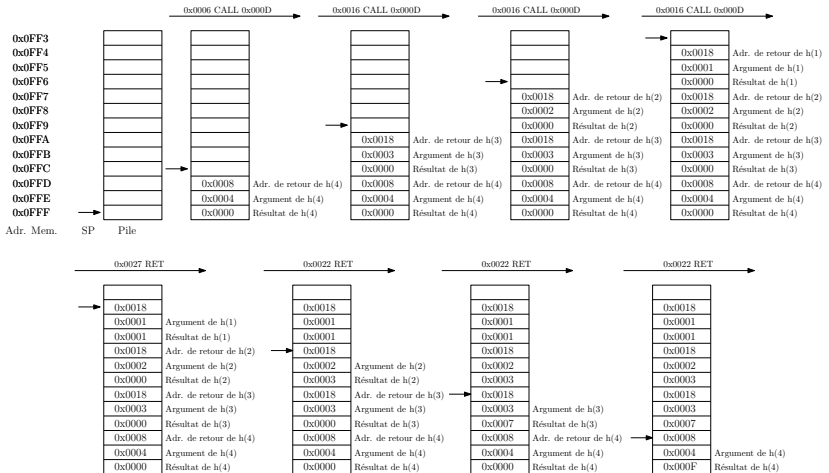
## Problème

Calculer le nombre de déplacement minimum nécessaires :

$$h(n) = \begin{cases} 1 & \text{si } n = 1 \\ 2h(n-1) + 1 & \text{sinon} \end{cases}$$

$$h(4) = ?$$

# Des fonctions récursives : Hanoi



Une réalisation impérative (for, while,..) serait plus efficace

# Au fait ..

Comment passer de la formalisation du problème à un algorithme efficace permettant de le résoudre ??

# Au fait ..

Comment passer de la formalisation du problème à un algorithme efficace permettant de le résoudre ??

Cours **FISDA**: Fondement de l'Informatique, Structures de Données et Algorithmie

Cours **Génie logiciel**: Etude des méthodes et bonnes pratiques pour le développement logiciel

# Simplifions la programmation de la machine

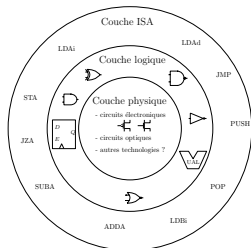
## Souhaits

- un programme moins long, moins répétitif, moins sujet aux bugs : procédures et pile
- mais on programme toujours en code machine ?!?!

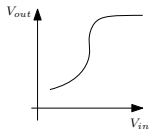
8000 0FFF	LDSPi 0x0FFF
1000 0001	LDAi 0x0001
2000 0002	LDBi 0x0002
⋮	⋮



## La couche d'assemblage



Couche physique



Couche logique

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

Couche ISA

Registres : A, B, PC

```
LDA : 0x1000
STA : 0x1c00
JMP : 0x7000
```

Couche d'assemblage

```
main: LDAi 0x0003
      STA 0x1000
      JMP 0x0020
```

Langage de haut niveau

Python

```
x = 3
print(x)
for i in range(3):
    x = x - 1
```

C

```
int x = 3;
printf("%i", x);
while(x != 0) {
    x = x - 1;
}
```

# Langage d'assemblage et assembleur

Adresse mémoire	Programme assembleur	Programme machine
0x0000	LDSPi 0x0030	8000 30
0x0002	DECSP	9400
0x0003	LDAi 0x0007	1000 7
0x0005	PUSHA	b000
0x0006	LDAi 0x0008	1000 8
0x0008	PUSHA	b000
0x0009	CALL sum	a000 20
.....	.....	.....
0x0020	sum: PEEKA 0x0003	bc00 3
0x0021	PEEKB 0x0002	cc00 2
0x0022	ADDA	3000
0x0023	POKEA 0x0004	b800 4
0x0024	RET	a800
0x0025		
0x0026		
.....		

Assembleur : programme traduisant langage d'assemblage → code machine.

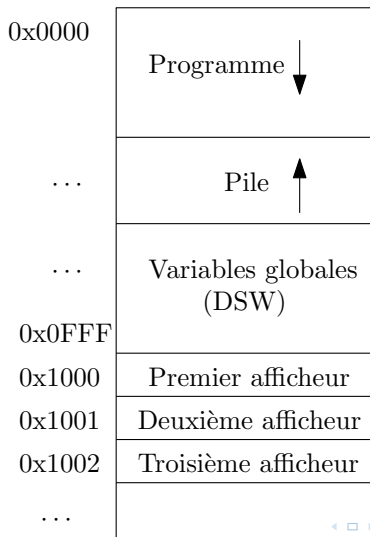
Abus : programme assembleur = programme en langage d'assemblage

## Quelques éléments de syntaxe de notre assembleur

- On utilise les mnémoniques LDAi, LDBi, STA, CALL, ...
- les valeurs en hexadécimal sans préfixe 0x
- “;” commentaire
- étiquettes : pour les branchements, pour les variables; spécial @stack@
- pseudo-instructions : e.g.
  - DSW : allocation de variables globales; par convention en début de programme!

Les variables et les étiquettes ne doivent pas être interprétables en hexadécimal

# Disposition du programme et des données en mémoire (chez nous)



# Traduction de quelques structures de contrôle

```
1: while  $i \neq 0$  do  
2:    $res \leftarrow res + i$   
3:    $i \leftarrow i - 1$ 
```

```
loop: LDAd i  
      JZA  end  
      LDBd res  
      ADDB  
      STB  res  
      LDBi 1  
      SUBA  
      STA  i  
      JMP  loop  
end:  ...
```

# Traduction de quelques structures de contrôle

## for $\Leftrightarrow$ while

```
1: for  $i = 0; i < N; i = i + 1$  do  
2:    $res \leftarrow res + i$ 
```

$\Leftrightarrow$

```
1:  $i \leftarrow 0$   
2: while  $i < N$  do  
3:    $res \leftarrow res + i$   
4:    $i \leftarrow i + 1$ 
```

## for $\Leftrightarrow$ while

```
1: for (init; condition ; incrément) do  
2:   action  
  
1: init  
2: while condition do  
3:   action  
4:   incrément
```

# Traduction de quelques structures de contrôle

1: **if**  $x \neq 0$  **then**

2:      $x \leftarrow 1$

3:  $x \leftarrow x + 1$

LDAd x

JZA end

LDAi 1

STA x

end: LDBi 1

ADDA

STA x

# Traduction de quelques structures de contrôle

1: **if**  $x == 10$  **then**

2:      $x \leftarrow 0$

3: **else**

4:      $x \leftarrow x + 1$

LDAd x

LDBi A

SUBB

JZB if

else: LDBi 1

ADDA

STA x

JMP end

if: LDAi 0

STA x

end: ....



# L'assembleur : traduction en code machine

## Exercice : Assembler le programme

```
1:  $u \leftarrow 127$ 
2: while True do
3:    $u \leftarrow next(u)$ 
4:   print( $u$ )
5: function next( $u$ )
6:   if  $u$  pair then return  $u/2$ 
7:   elsereturn  $3u + 1$ 
```

Compteur d'emplacement, table des symboles, variables globales; cf  
syr.asm

Démo : python assemble.py

**Langage de haut niveau (Python, C, C++, Scala, ..)**

C++



# Mais pourquoi ?

## Assembleur

- spécifique à une architecture
- encore dur à programmer (LDAd b; LDBd c; ADDA; STA a)
- peu de vérification syntaxique: on peut ajouter des choux et des carottes

## Langage de haut niveau

- indépendant de l'architecture
- langage plus intuitif, e.g.  $a = b + c$ , structures de contrôle, définition de fonctions
- vérification syntaxique

# Langage : Interprété ou compilé

## Langage compilé

Un programme (compilateur) convertit le code source en code machine qui peut ensuite être exécuté

Ex : C++/g++

g++ -S main.cc

g++ -o main main.cc ; hexdump -C main

./main

## Langage interprété

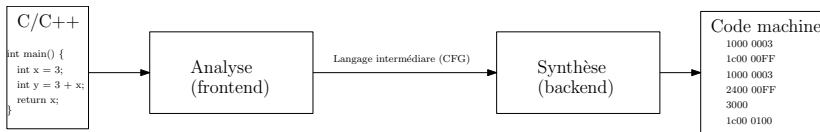
Un programme (interpréteur) interprète “à la volée” le code source

Ex : Python/ python

python main.py

⇒ Machine virtuelle python

# Brève Anatomie d'un compilateur

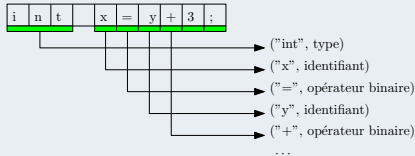


Référence : "The dragon book" Aho, Lam, Sethi, Ullman

# La phase d'analyse (frontend)

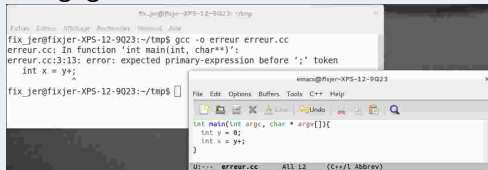
## Analyse lexicale

### Ségmentation et identification des lexèmes



## Analyse syntaxique

Construction d'un arbre syntaxique à partir des lexèmes et d'une grammaire du langage.



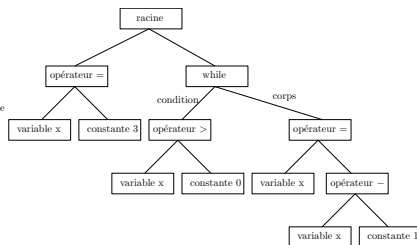
# La phase d'analyse (frontend)

```
x = 3;  
while(x > 0) {  
  x = x - 1;  
}
```

Analyse lexicale

("x", identifiant)  
("3", constante)  
("while", mot clef)  
("(", parenthese)  
("x", identifiant)  
...

Analyse syntaxique



# Génération et optimisation d'une représentation intermédiaire

## Représentation intermédiaire

- indépendante du langage source (C, C++, ..) et de l'architecture (x86, ARM, CentraleSupélec)
- facile à produire, facile à convertir en code machine
- optimisable

Ex : register transfer language, gimple, generic, three adress code, single static assignment, control flow graph, ...

Référence : “The dragon book” Aho, Lam, Sethi, Ullman



# Exemple de représentation intermédiaire : Three Address code

## Éléments de syntaxe

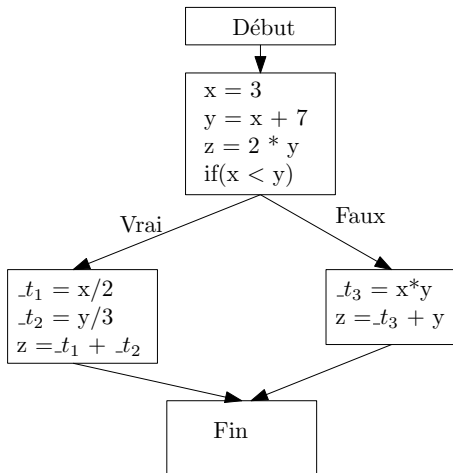
- opérations binaires : `x := y op z`
- opérations unaires : `x := op y`
- copies : `x := y`
- sauts (in)conditionnels : `goto L; If x relop y goto L`
- procédures : `param x1,.. call p, return y`
- ...

# Exemple de représentation intermédiaire : Three Adress code

```
int x = 3;
int y = 2 + 7 + x;
int z = 2*y;
if (x < y) {
    z = x/2 + y/3;
}
else {
    z = x * y + y;
}
```

```
x = 3;
_t1 = 2 + 7;
y = _t1 + x;
z = 2 * y;
_t2 = x < y;
IfZ _t2 Goto _L0;
_t3 = x / 2;
_t4 = y / 3;
z = _t3 + _t4
Goto _L1
_L0: _t5 = x * y;
     z = _t5 + z;
_L1:
```

# Exemple de représentation intermédiaire : Control Flow Graph (CFG)



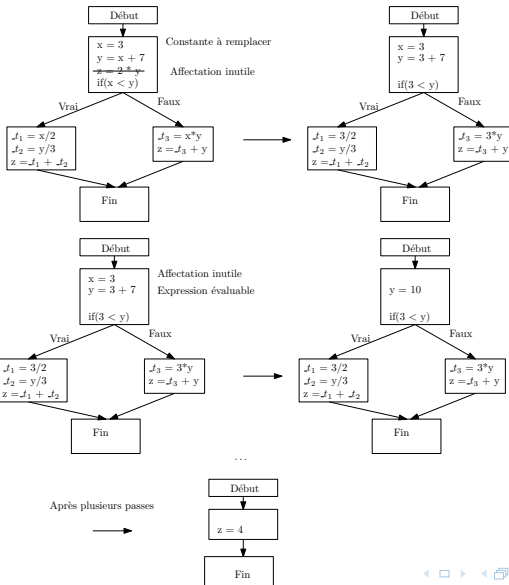
# Optimisation d'un CFG

Application répétée de quelques règles de simplification

- supprimer des affectations inutiles
- remplacer des constantes :  $\text{int } x = 3; \text{ int } y = x + 2 \Rightarrow \text{int } x = 3; \text{ int } y = 3 + 2;$
- calculer des expressions constantes :  $\text{int } y = 3 + 2; \Rightarrow \text{int } y = 5$

jusqu'à ce que plus aucune des règles ne soit applicable

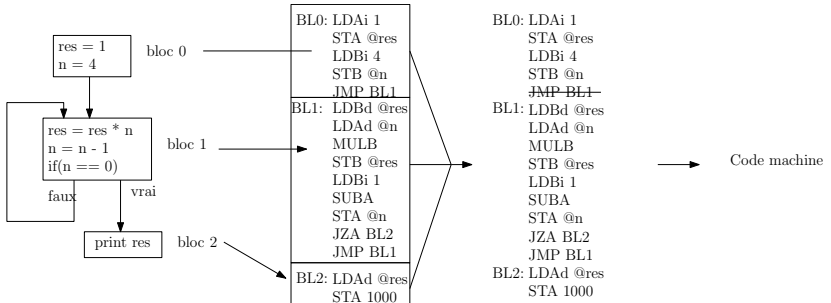
# Optimisation d'un CFG



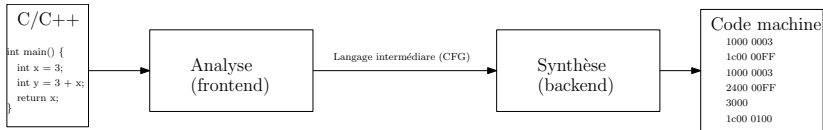
# La phase de synthèse (backend)

## Représentation intermédiaire $\Rightarrow$ Code machine

- génération du code machine de chacun des blocs
- disposition en mémoire des blocs
- optimisations éventuelles



# Et voila



Référence : "The dragon book" Aho, Lam, Sethi, Ullman