

Architectures des processeurs langages d'assemblage et assembleur en ligne

Benoît Morgan

Intel corp

30 septembre 2024

Introduction — *plan du cours*

- ▶ Volume : 2 cours magistraux
- ▶ Architectures matérielles des processeurs
- ▶ Rappels compilation et langage C
- ▶ Outils binaires
- ▶ Assembleur
- ▶ Assembleur en ligne

Introduction — *motivation*

Prérequis des cours

- ▶ Rétro-ingénierie
- ▶ Sécurité logicielle et sécurité logicielle avancée
- ▶ Sécurité des noyaux et des hyperviseurs

Introduction — *langage d'assemblage*

- ▶ Langage de programmation
 - ▶ Dont l'objectif est la génération d'un programme
- ▶ Très proche du jeu d'instructions et de l'architecture d'un processeur
 - ▶ Intel x86, AMD x86, ARMv8, famille PowerPC, MI
- ▶ Apporte un niveau d'abstraction du jeu d'instruction des processeurs
- ▶ Généralement : une famille de processeurs = un langage d'assemblage

Plan du cours

Architectures matérielles des processeurs

Intel x86

Assembleur GNU

... en ligne

Plan du cours

Architectures matérielles des processeurs

Intel x86

Assembleur GNU

... en ligne

Processeur / Microprocesseur *quésaco ?*¹

Processeur

- ▶ Composant matériel d'une machine
- ▶ Exécute les instructions machine d'un programme
- ▶ Capacités généralistes
- ▶ "Intelligence" d'une machine

Microprocesseur = processeur + mémoire + entrées / sorties + etc.

1. Qu'est-ce que c'est ? patois de mon village

Instructions machine ?

Ou instructions

- ▶ Opération élémentaire exécutée par un processeur
- ▶ Types principaux d'instructions :
 - ▶ UAL : arithmétique ; logique ; *bit-wise*
 - ▶ Accès mémoire : stockage de l'information à moyen terme
 - ▶ Entrées / sorties : lecture ou écriture
 - ▶ Gestion du flot d'exécution : sauts ; fonctions ; interruptions
 - ▶ Système : configuration ; noyau ; machines virtuelles ; énergie

UAL : Instructions arithmétiques et logiques

Opérateur binaires

Exemple : $A + B$

- ▶ Logique : $\wedge$; $\vee$; $\oplus$; etc.

Exemple : $\top \wedge \perp = \perp$

- ▶ Arithmétique : $+$; $-$; $\times$; $\div$

- ▶ Binaire : *and*; *or*; *xor*

Exemple : $1001 \text{ or } 0010 = 1011$

Opérateurs unaire

Exemple : $\neg B$

- ▶ Logique : $\neg$, etc.

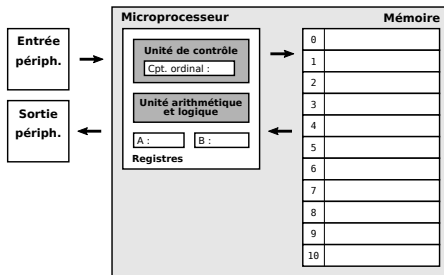
Exemple : $\neg \top = \perp$

- ▶ Arithmétique : $-$, etc.

- ▶ Binaire : $\sim$ ou not

Exemple : $\sim 1001 = 0110$

Architecture générique typique et simplifiée



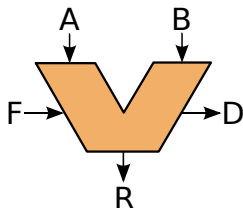
Ensemble des instructions supportées par ce processeur :
jeu d'instruction.

- ▶ Lire une entrée utilisateur : $x = \text{entrée} \mid x \in \{A, B\}$
- ▶ Écrire une entrée utilisateur : $\text{sortie} = x \mid x \in \{A, B\}$
- ▶ Addition : $x = y + z \mid x, y, z \in \{A, B\}$
- ▶ Soustraction : $x = y - z \mid x, y, z \in \{A, B\}$
- ▶ Lecture mémoire : $x = \text{MEM}[y] \mid x \in \{A, B\} \wedge y \in [0, 10]$
- ▶ Écriture mémoire : $\text{MEM}[x] = y \mid y \in [0, 10] \wedge x \in \{A, B\}$

UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



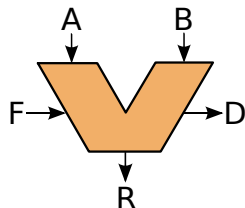
Décomposition en suite d'instruction arithmétiques :

$\text{sortie}((\text{entrée}() + \text{entrée}()) \times (\text{entrée}() + \text{entrée}()))$

UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



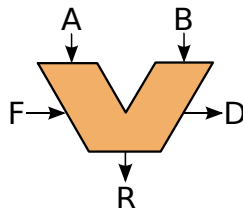
Décomposition en suite d'instruction arithmétiques :

sortie((entrée() + entrée()) × (entrée() + entrée()))

UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



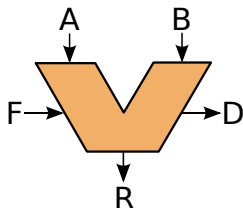
Décomposition en suite d'instruction arithmétiques :

sortie((entrée() + entrée()) × (entrée() + entrée()))

UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



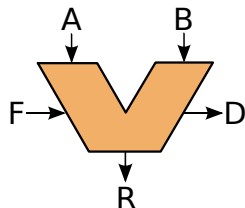
Décomposition en suite d'instruction arithmétiques :

sortie((entrée() + entrée())×(entrée() + entrée()))

UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



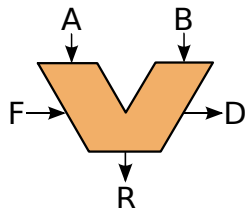
Décomposition en suite d'instruction arithmétiques :

sortie((entrée() + entrée()) × (entrée() + entrée()))

UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



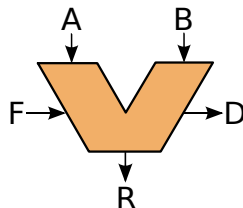
Décomposition en suite d'instruction arithmétiques :

$$\text{sortie}((\text{entrée()} + \text{entrée()})) \times (\text{entrée()} + \text{entrée()}))$$

UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



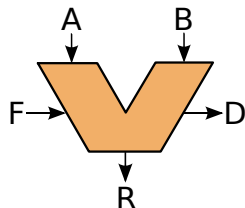
Décomposition en suite d'instruction arithmétiques :

$$\text{sortie}(\text{entrée}() + \text{entrée}()) \times (\text{entrée}() + \text{entrée}())$$

UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



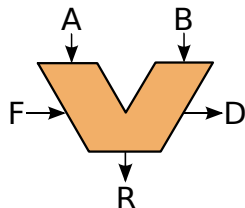
Décomposition en suite d'instruction arithmétiques :

sortie((entrée() + entrée()) × (entrée() + entrée()))

UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



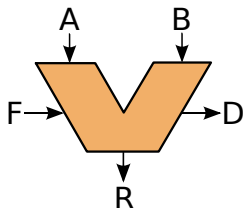
Décomposition en suite d'instruction arithmétiques :

$$\text{sortie}(\underbrace{(\underbrace{\text{entrée}() + \text{entrée}())}_{\text{}}}_{\text{}} \times \underbrace{(\underbrace{\text{entrée}() + \text{entrée}())}_{\text{}}}_{\text{}})$$

UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



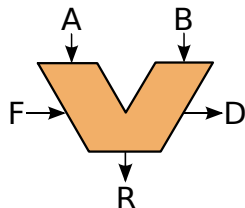
Décomposition en suite d'instruction arithmétiques :

$$\text{sortie}(\underbrace{(\underbrace{\text{entrée}() + \text{entrée}())}_{\text{A}}}_{\text{1}} \times \underbrace{(\underbrace{\text{entrée}() + \text{entrée}())}_{\text{A}}}_{\text{1}})$$

UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



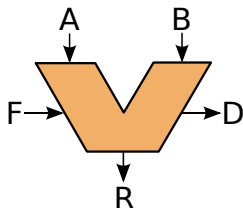
Décomposition en suite d'instruction arithmétiques :

$$\text{sortie}(\underbrace{(\underbrace{\text{entrée}() + \text{entrée}())}_{\text{B } 2}} \times \underbrace{(\underbrace{\text{entrée}() + \text{entrée}())}_{\text{A } 1})}$$

UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



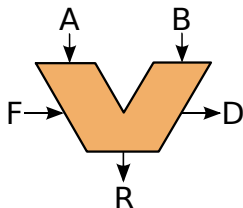
Décomposition en suite d'instruction arithmétiques :

$$\text{sortie}(\underbrace{(\underbrace{\text{entrée}() + \text{entrée}())}_{\text{B } 2}} \times \underbrace{(\underbrace{\text{entrée}() + \text{entrée}())}_{\text{A } 3, \text{ A } 1}})$$

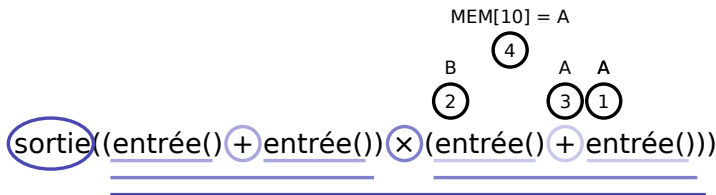
UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



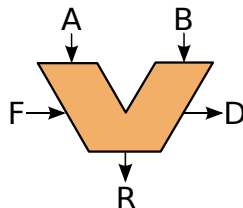
Décomposition en suite d'instruction arithmétiques :



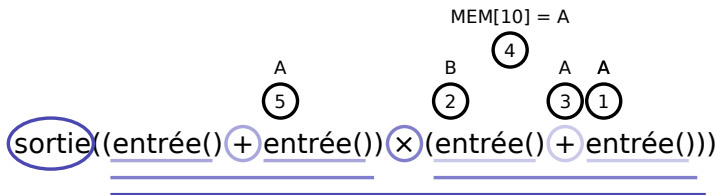
UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



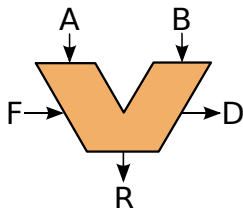
Décomposition en suite d'instruction arithmétiques :



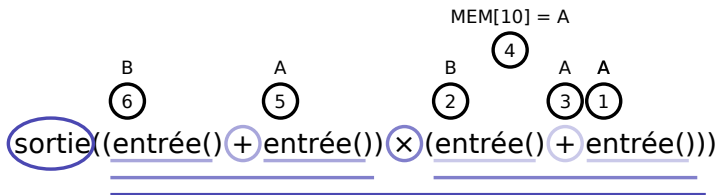
UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



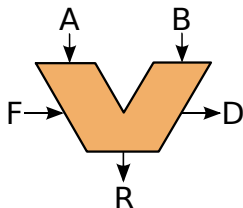
Décomposition en suite d'instruction arithmétiques :



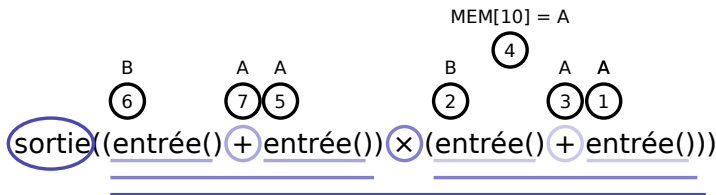
UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



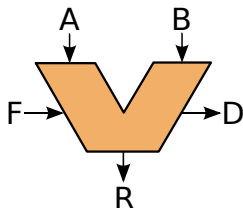
Décomposition en suite d'instruction arithmétiques :



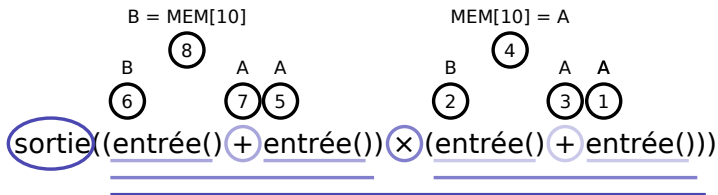
UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit} \Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



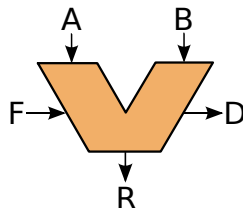
Décomposition en suite d'instruction arithmétiques :



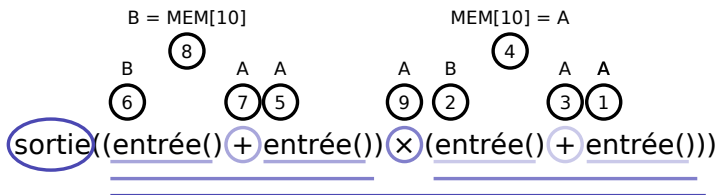
UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit} \Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



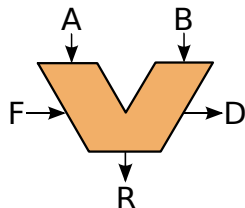
Décomposition en suite d'instruction arithmétiques :



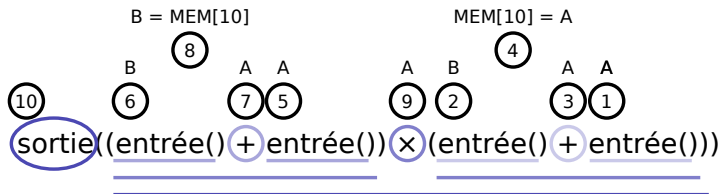
UAL : calcul arithmétique avec une UAL

Interface matérielle d'une UAL

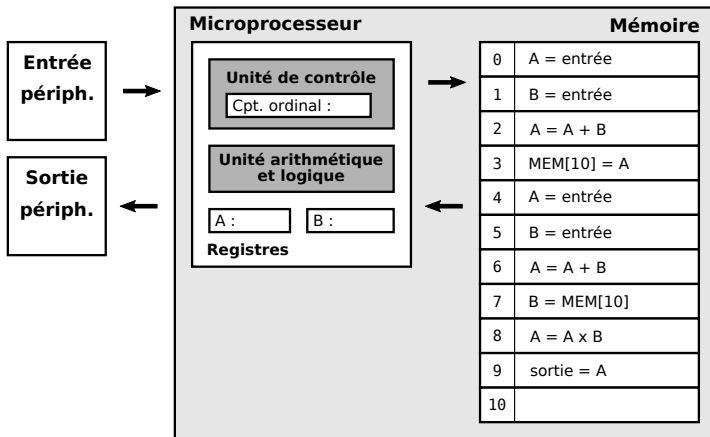
- ▶ Généralement 2 entrées dans un processeur
(Pas une carte vidéo)
- ▶ Généralement 1 sortie
- ▶ Processeur $x\text{-bit}$ $\Leftrightarrow$ UAL $x\text{-bit}$
Et registres généraux $x\text{-bit}$



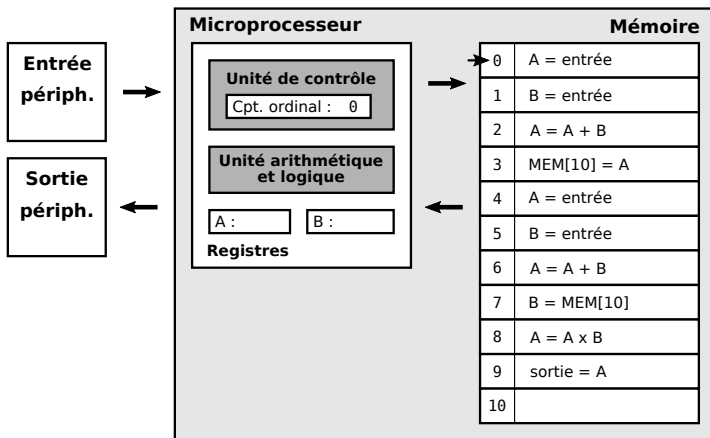
Décomposition en suite d'instruction arithmétiques :



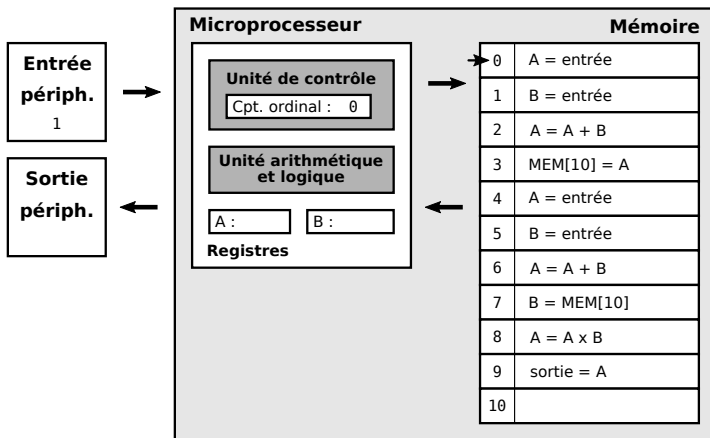
Exécution du programme



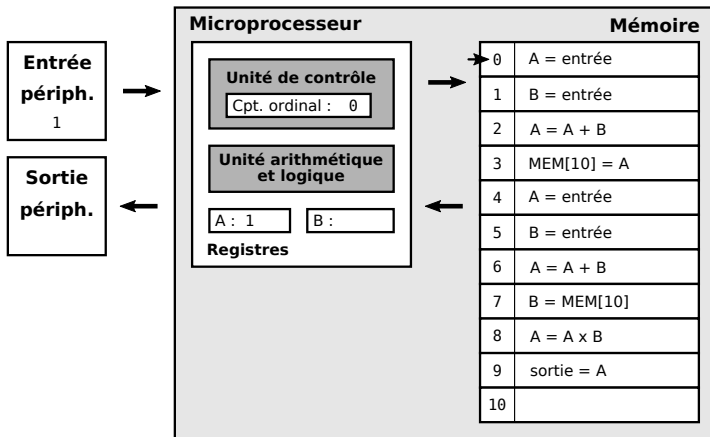
Exécution du programme



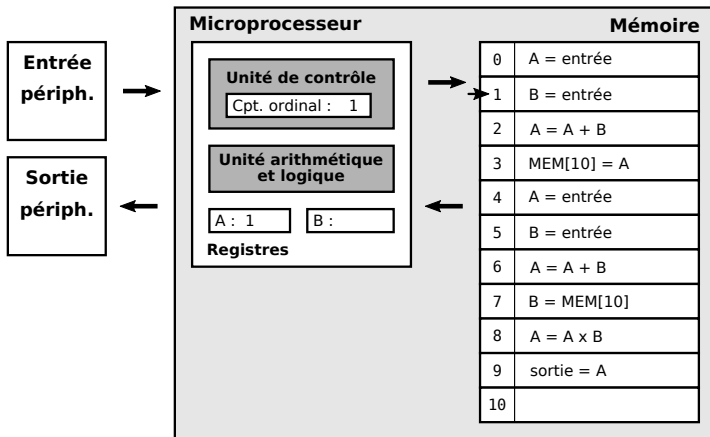
Exécution du programme



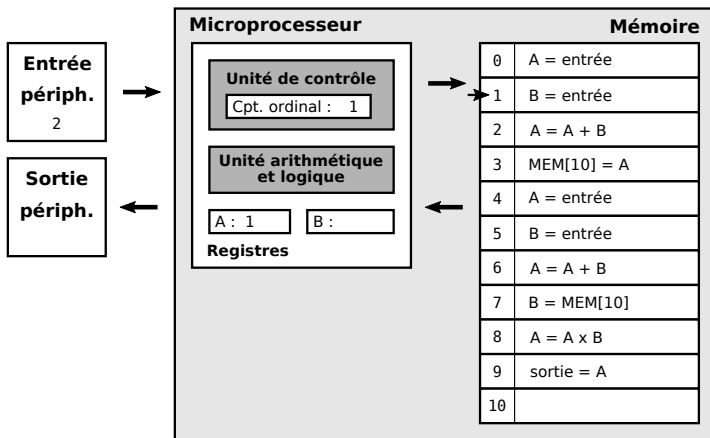
Exécution du programme



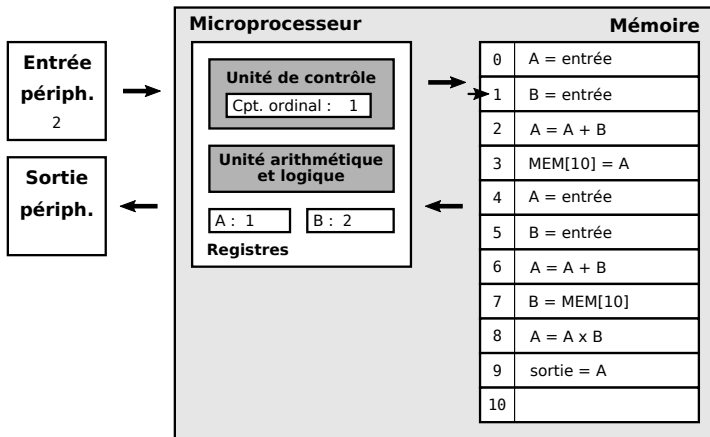
Exécution du programme



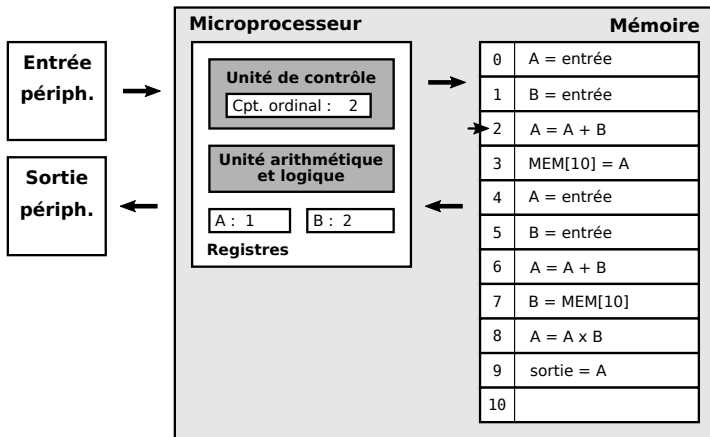
Exécution du programme



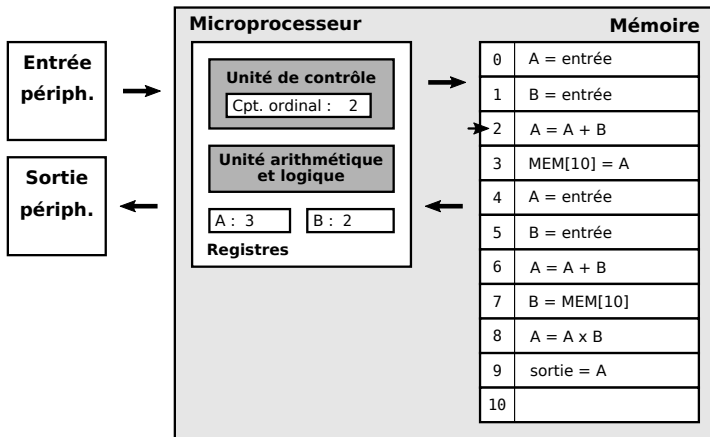
Exécution du programme



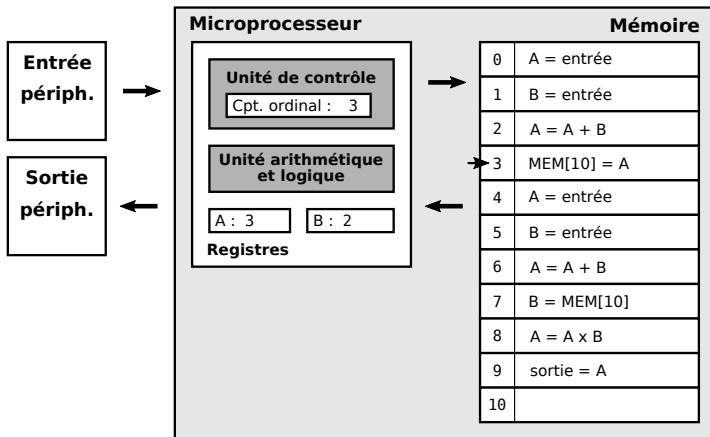
Exécution du programme



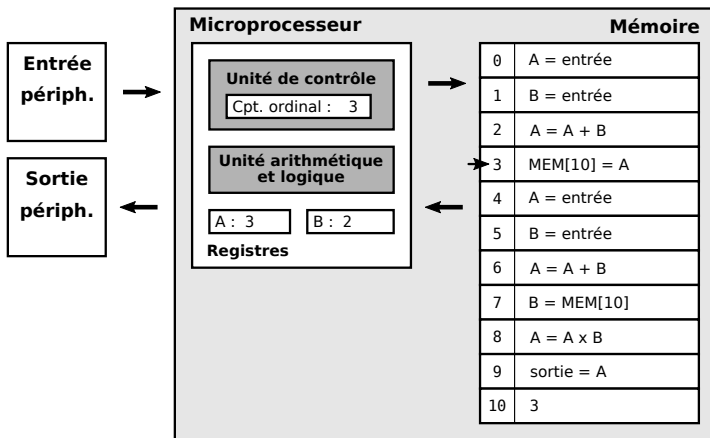
Exécution du programme



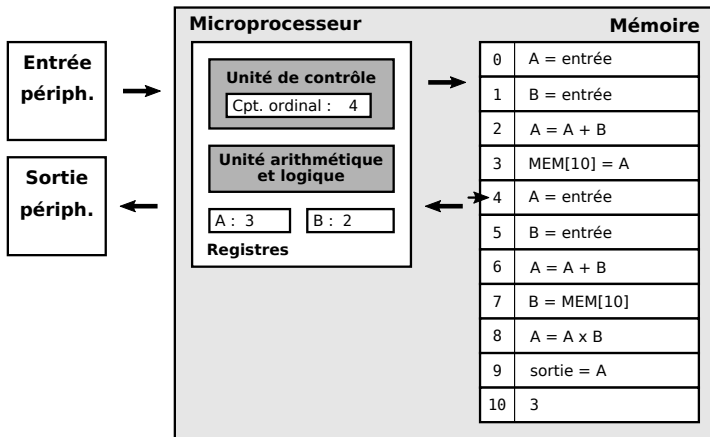
Exécution du programme



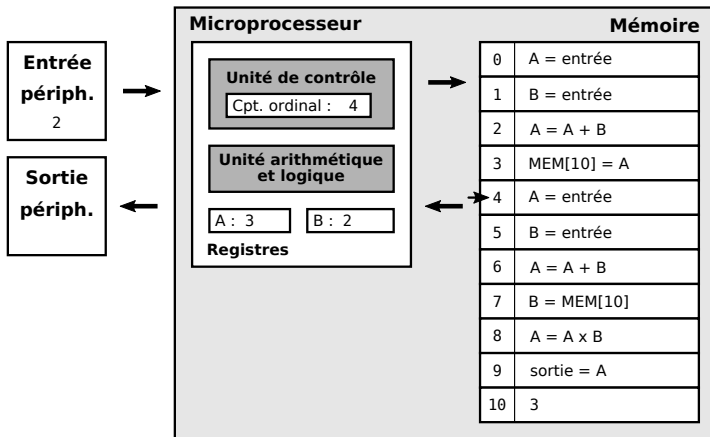
Exécution du programme



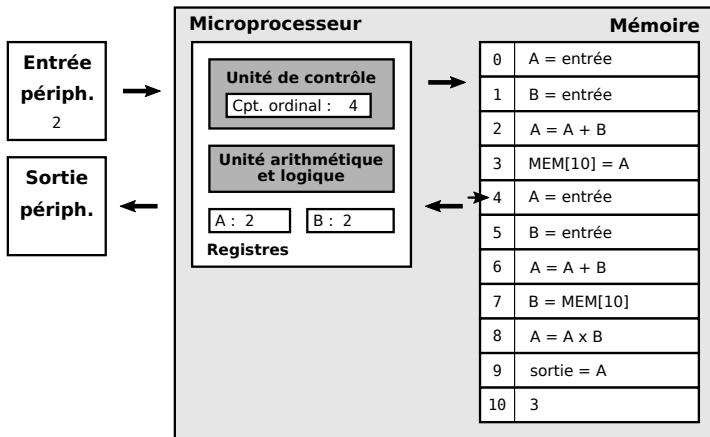
Exécution du programme



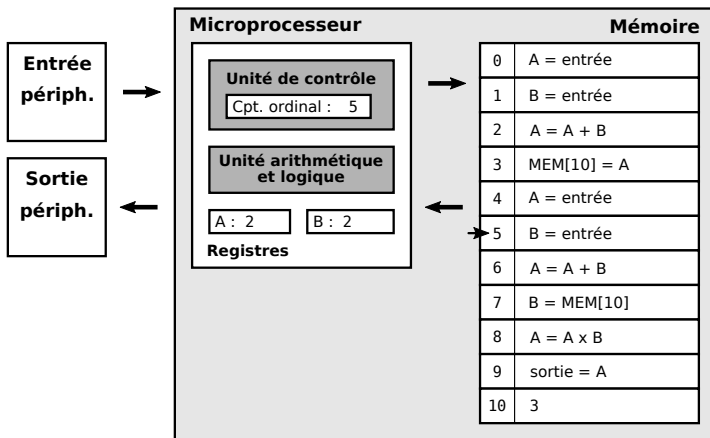
Exécution du programme



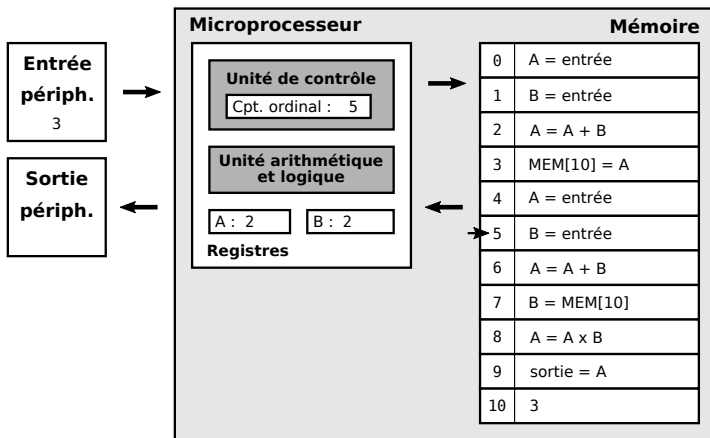
Exécution du programme



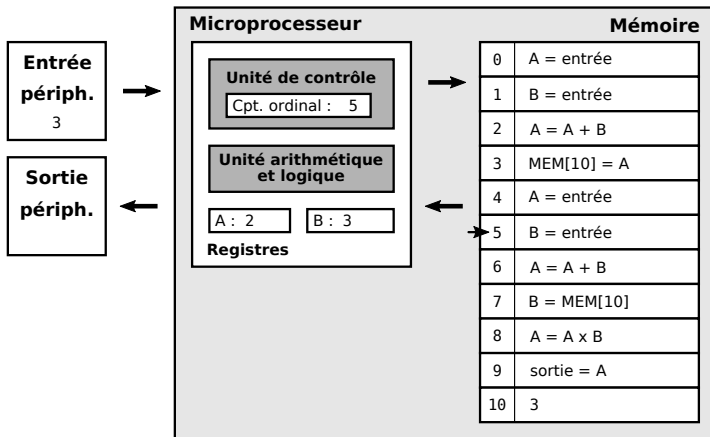
Exécution du programme



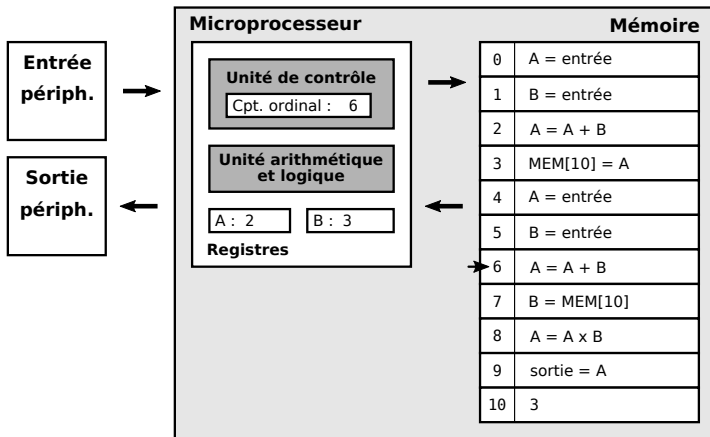
Exécution du programme



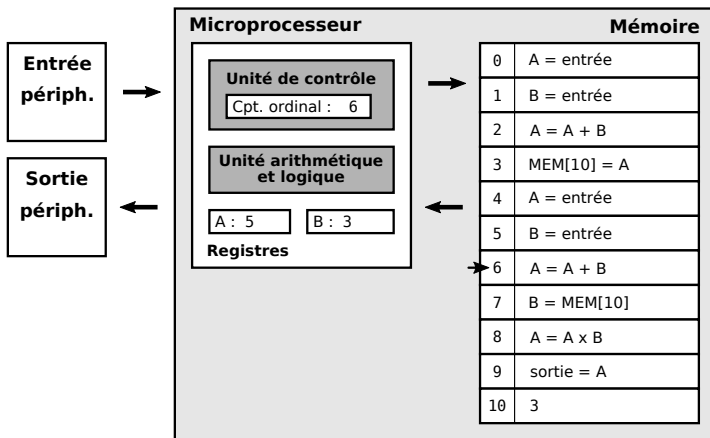
Exécution du programme



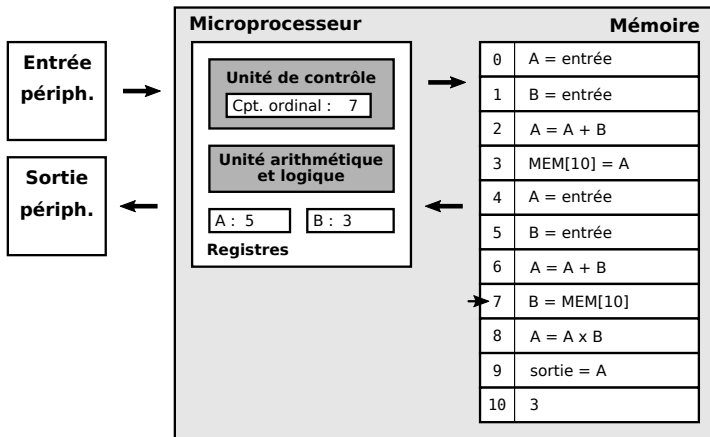
Exécution du programme



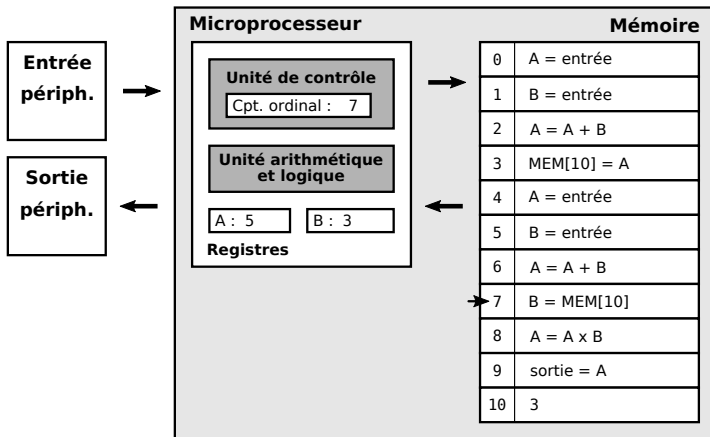
Exécution du programme



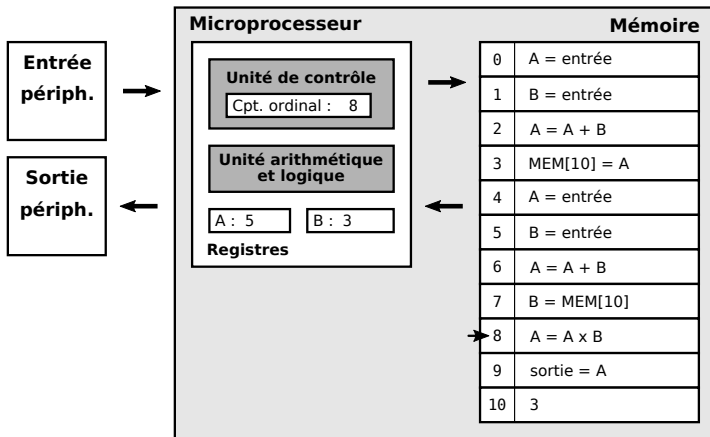
Exécution du programme



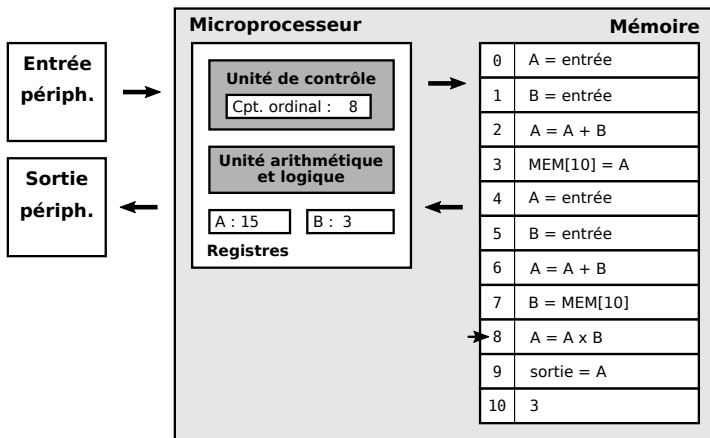
Exécution du programme



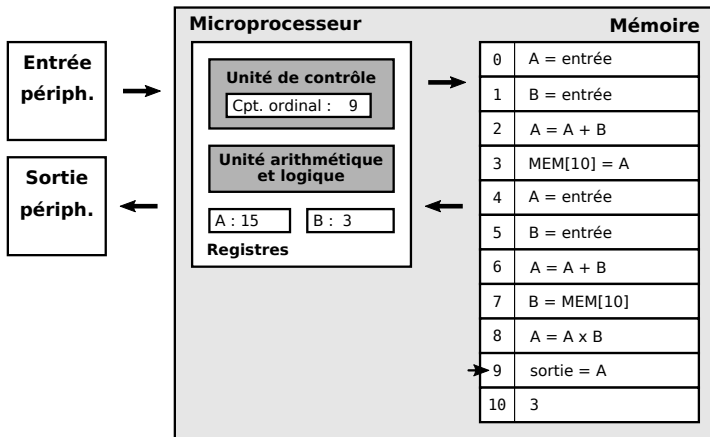
Exécution du programme



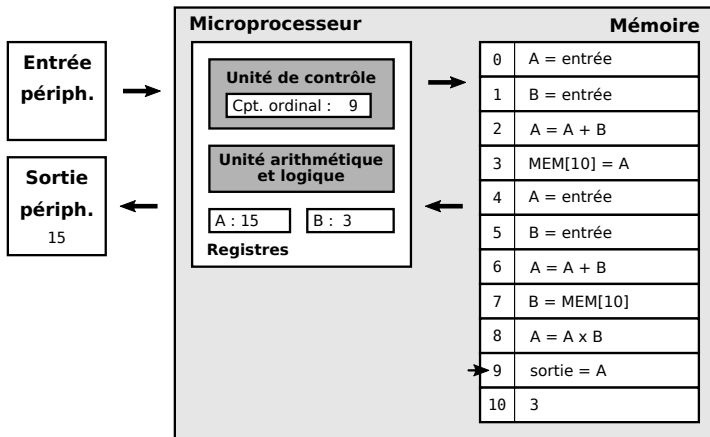
Exécution du programme



Exécution du programme



Exécution du programme



UAL : drapeaux et comparaisons

Généralement un registre spécial du processeur, mis à jour par l'UAL lors de la dernière opération

Drapeaux arithmétiques

- ▶ Z : résultat nul
- ▶ C (carry) : retenue
- ▶ O (overflow) : l'espace des nombre signés à été dépassé
- ▶ P : parité
- ▶ S : signe

Comparaison

Supportée par certain processeurs : $A \text{ CMP } B$

Mise à jour des drapeaux de l'UAL en exécutant une soustraction dont le résultat ne sera pas stocké.

→ Utile pour les structures de contrôle d'un programme.

Instructions de contrôle du flot d'exécution

Permet la mise en œuvre de tests et de boucles quand utilisées en complément avec l'UAL.

```
if (expr.) {statements} else {statements}
```

```
while (expr.) {statements}
```

```
for (statements; expr; statements) {statements}
```

Branchements

- Branchement inconditionnel : Compteur ordinal = @
Conséquence : la prochaine instruction est exécutée à @

- Branchement conditionnel :

Compteur ordinal = @ $\Leftrightarrow$ condition |
condition $\in \{Z = 1, Z = 0, C = 0, \text{etc.}\}$

Conséquence : la prochaine instruction est exécutée à @ si la condition est vérifiée, sinon on incrémente le compteur ordinal

Exemple de test (*if*)

Soit le programme C suivant

```
if (6 == entrée()) {  
    sortie(0);  
} else {  
    sortie(1);  
}
```

Peut être traduit en suite d'instructions génériques suivantes :

#	Programme	Commentaires
0	A = 6	if (6 == entrée())
1	B = entrée	
2	A = A - B	ou A CMP B
3	cpt. ordinal = 6 ssi Z = 0	
4	sortie = 0	sortie(0)
5	cpt. ordinal = 7	else
6	sortie = 1	sortie(1)
7	[..]	

Exemple de boucle *for*

Soit le programme C suivant

```
for (int i = 0; i < 10; i++) {  
    sortie(i);  
}
```

Peut être traduit en suite d'instructions génériques suivantes :

#	Programme	Commentaires
0	A = 0	int i = 0
1	B = 10	
2	A CMP B	i < 10 ?
3	cpt. ordinal = 7 ssi Z = 1	{
4	sortie = A	sortie(i)
5	A = A + 1	i++
6	cpt. ordinal = 2	}
7	[..]	

Support des procédures 1/5

Question : pourquoi est-ce qu'on ne peut pas utiliser deux simple sauts ?

#	Programme	Commentaires
0	A = 0	main() {}
1	cpt. ordinal = 10	f()
2	cpt. ordinal = 20	f2()
3	[..]	
10	A = A + 1	f() {}
11	cpt. ordinal = 2	
12	[..]	
20	A = A + 2	f2() {}
21	cpt.ordinal = 3	

Support des procédures 2/5

Question : pourquoi est-ce qu'on ne peut pas utiliser deux simple sauts ?

Réponse : plusieurs appels de la même procédure

#	Programme	Commentaires
0	A = 0	main() {}
1	cpt. ordinal = 10	f()
2	cpt. ordinal = 10	f()
3	[..]	
10	A = A + 1	f() {}
11	cpt. ordinal = 2	
12	[..]	
20	A = A + 2	f2() {}
21	cpt.ordinal = 3	

Support des procédures 2/5

Question : pourquoi est-ce qu'on ne peut pas utiliser deux simple sauts ?

Réponse : plusieurs appels de la même procédure

#	Programme	Commentaires
0	A = 0	main() {}
1	cpt. ordinal = 10	f()
2	cpt. ordinal = 10	f()
3	[..]	
10	A = A + 1	f() {}
11	cpt. ordinal = 2	
12	[..]	
20	A = A + 2	f2() {}
21	cpt.ordinal = 3	

Solution possible : nouveau registre C ; instruction d'appel de procédure : appel $x \mid x \in \mathcal{N} \Leftrightarrow \text{cpt.ordinal} = x \wedge C = \text{cpt. ordinal} + 1$; saut indirect (cf. opérandes et mode d'adressage).

Support des procédures 3/5

Avec utilisation du registre de contexte C, de l'instruction appel et du saut indirect.

#	Programme	Commentaires
0	A = 0	main() {}
1	appel 10	f() et C = 2
2	appel 10	f() et C = 3
3	[..]	
10	A = A + 1	f() {}
11	cpt. ordinal = C	cpt. ordinal = @retour
12	[..]	
20	A = A + 2	f2() {}
21	cpt. ordinal = C	cpt. ordinal = @retour

Support des procédures 4/5

Question : appel d'une procédure dans une procédure ?

#	Programme	Commentaires
0	A = 0	main() {}
1	appel 10	f() et C = 2
2	appel 10	f() et C = 3
3	[..]	
10	A = A + 1	f() {}
11	appel 20	f2() et C = 12
12	cpt. ordinal = C	cpt. ordinal = @retour
13	[..]	
20	A = A + 2	f2() {}
21	cpt. ordinal = C	cpt. ordinal = @retour

Support des procédures 4/5

Question : appel d'une procédure dans une procédure ?

Réponse : seulement un niveau de profondeur d'appel (C est écrasé)

#	Programme	Commentaires
0	A = 0	main() {}
1	appel 10	f() et C = 2
2	appel 10	f() et C = 3
3	[..]	
10	A = A + 1	f() {}
11	appel 20	f2() et C = 12
12	cpt. ordinal = C	cpt. ordinal = @retour
13	[..]	
20	A = A + 2	f2() {}
21	cpt. ordinal = C	cpt. ordinal = @retour

Support des procédures 4/5

Question : appel d'une procédure dans une procédure ?

Réponse : seulement un niveau de profondeur d'appel (C est écrasé)

#	Programme	Commentaires
0	A = 0	main() {}
1	appel 10	f() et C = 2
2	appel 10	f() et C = 3
3	[..]	
10	A = A + 1	f() {}
11	appel 20	f2() et C = 12
12	cpt. ordinal = C	cpt. ordinal = @retour
13	[..]	
20	A = A + 2	f2() {}
21	cpt. ordinal = C	cpt. ordinal = @retour

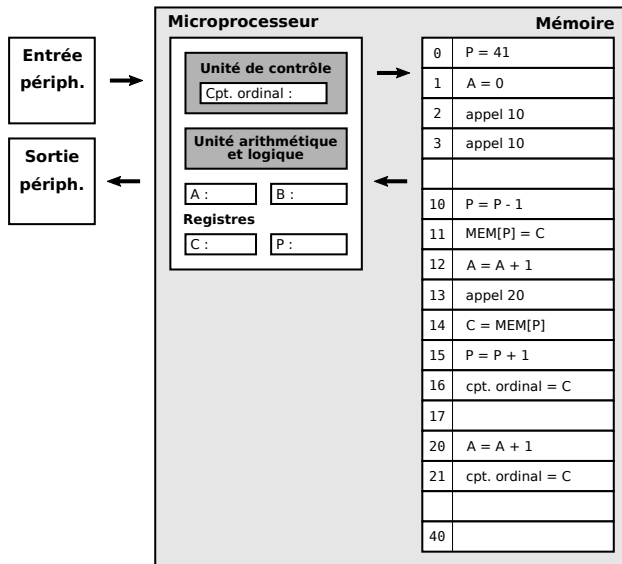
Solution possible : sauvegarder le registre C en mémoire dans une pile ; lecture mémoire indirecte ; registre de pile

Support des procédures 5/5

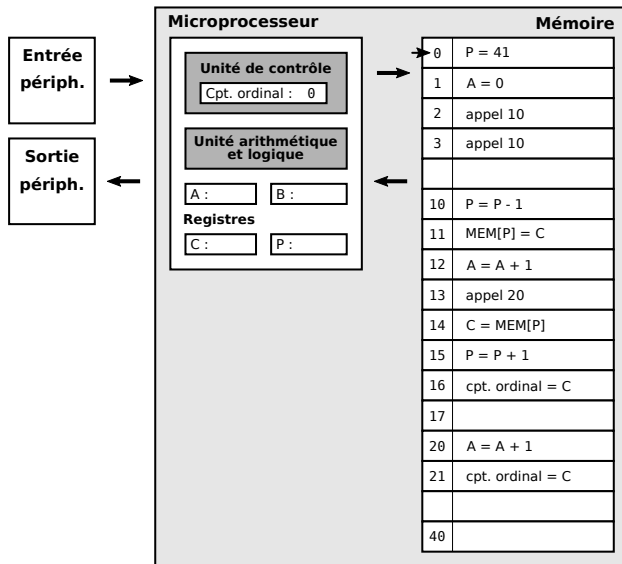
Solution : appel récursif avec pile et accès mémoire indirect

#	Programme	Commentaires
0	P = 41	Initialisation pile
1	A = 0	main() {}
2	appel 10	f() et C = 2
3	appel 10	f() et C = 3
4	[..]	
10	P = P - 1	f() {} sauvegarde du contexte
11	MEM[P] = C	
12	A = A + 1	
13	appel 20	f2() et C = 14
14	C = MEM[P]	restauration du contexte
15	P = P + 1	
16	cpt. ordinal = C	cpt. ordinal = @retour
17	[..]	
20	A = A + 2	f2() {}
21	cpt. ordinal = C	cpt. ordinal = @retour

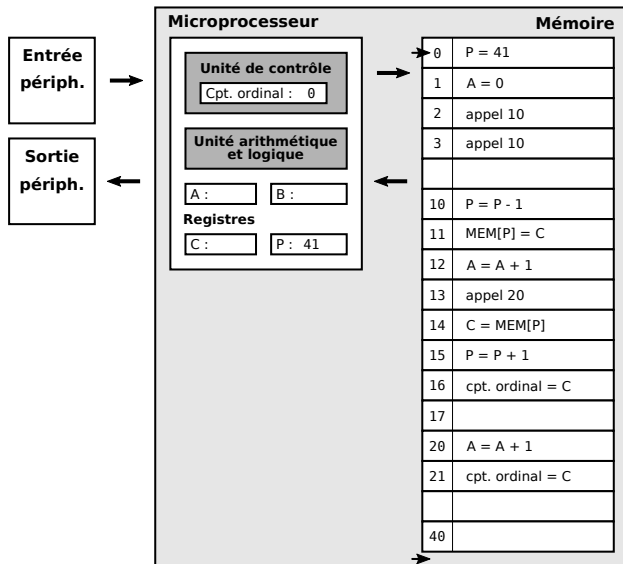
Support des procédures : exécution



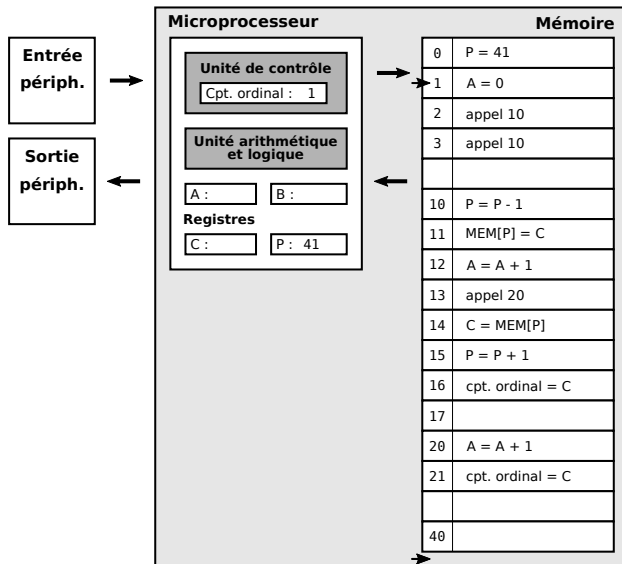
Support des procédures : exécution



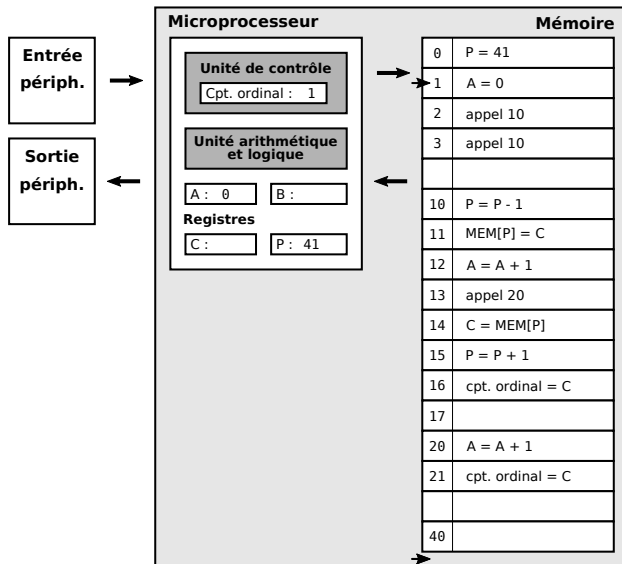
Support des procédures : exécution



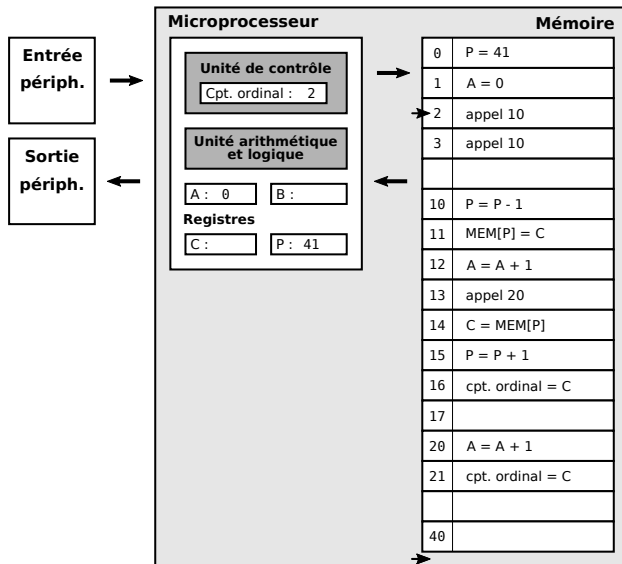
Support des procédures : exécution



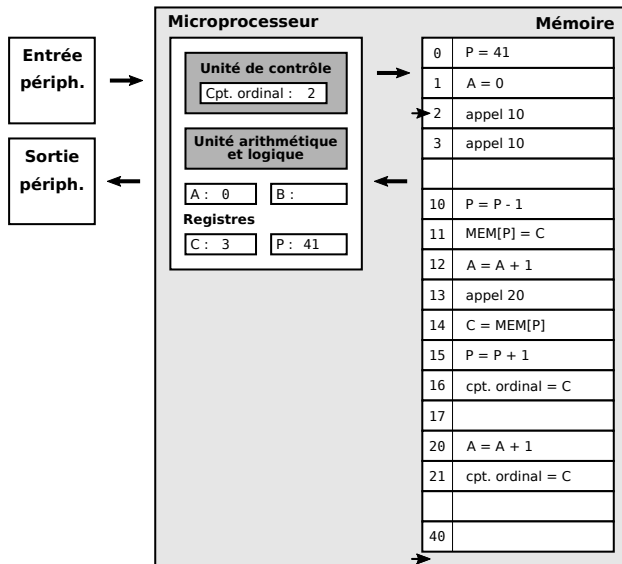
Support des procédures : exécution



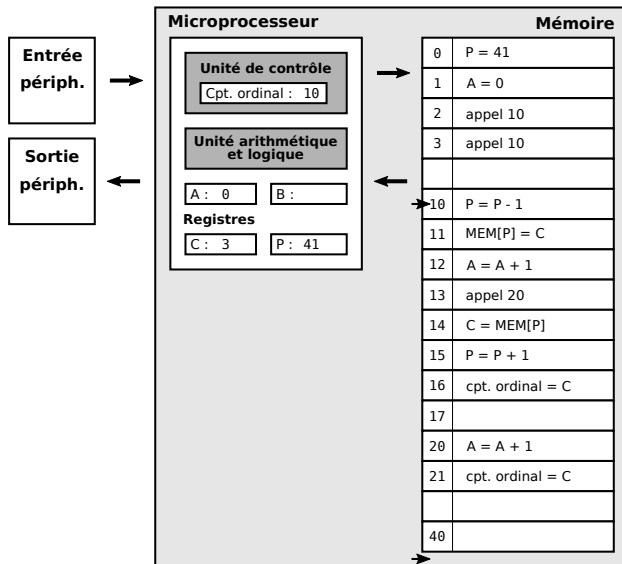
Support des procédures : exécution



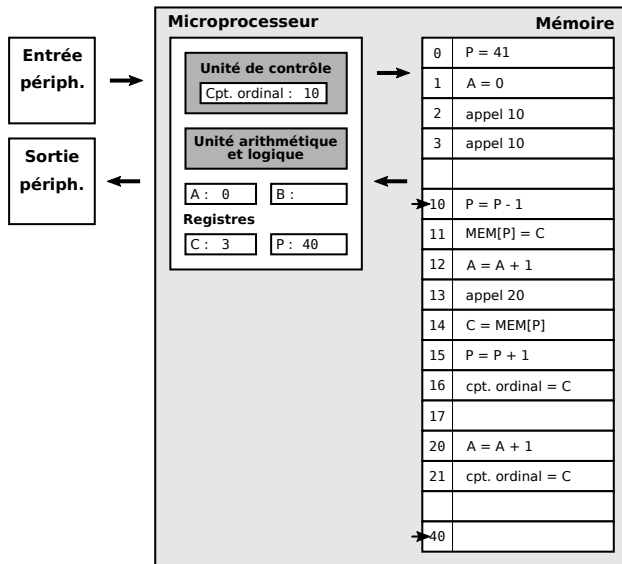
Support des procédures : exécution



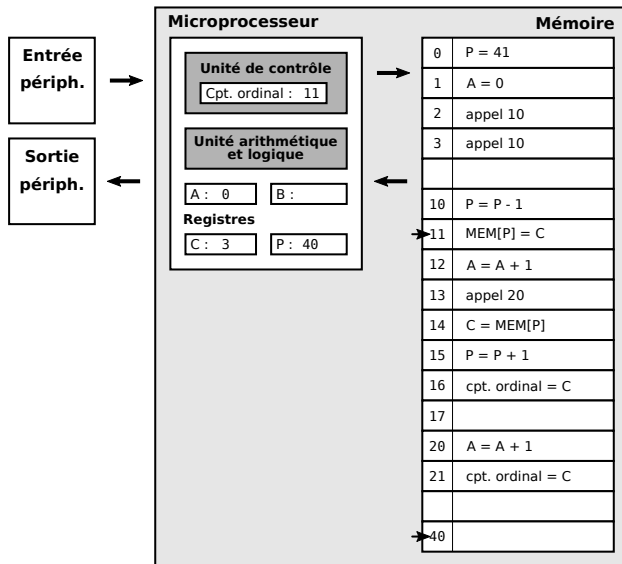
Support des procédures : exécution



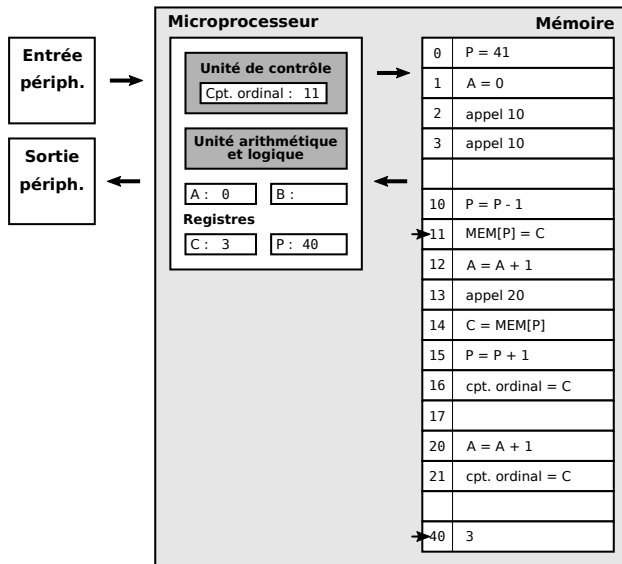
Support des procédures : exécution



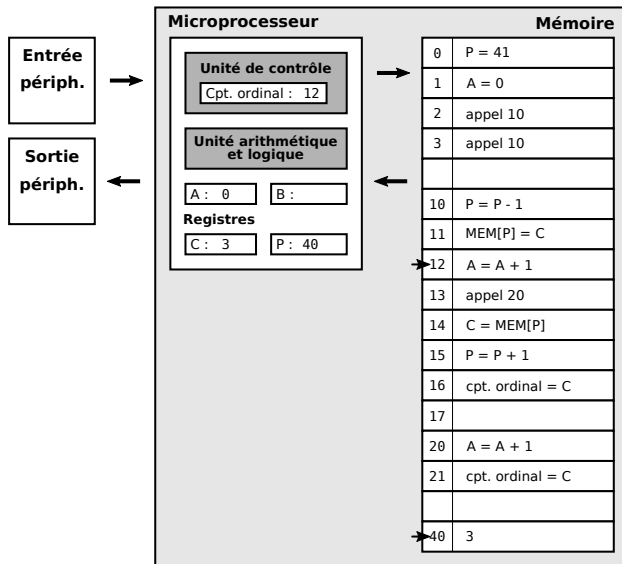
Support des procédures : exécution



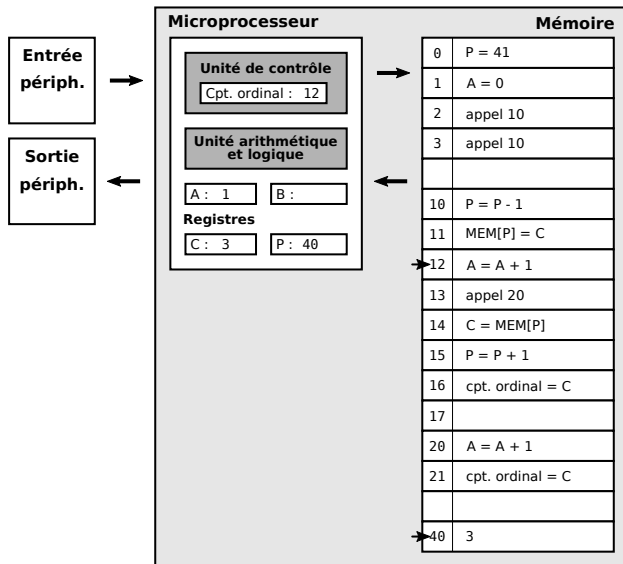
Support des procédures : exécution



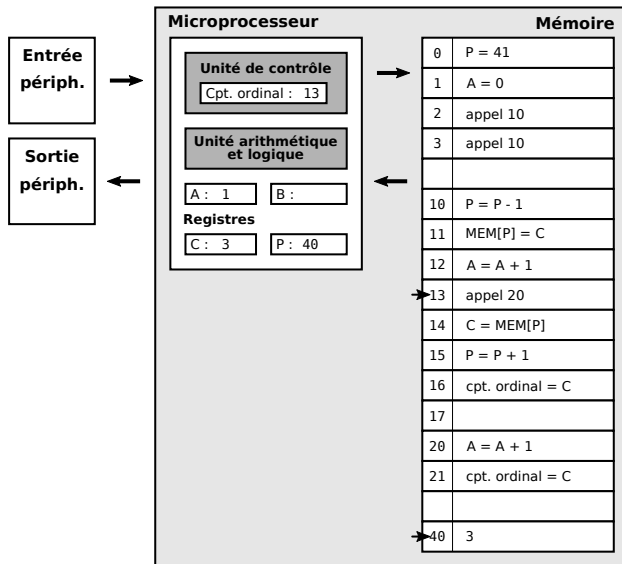
Support des procédures : exécution



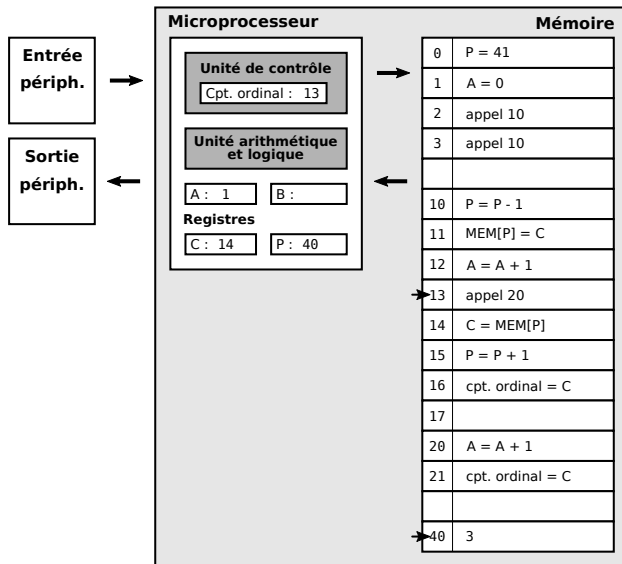
Support des procédures : exécution



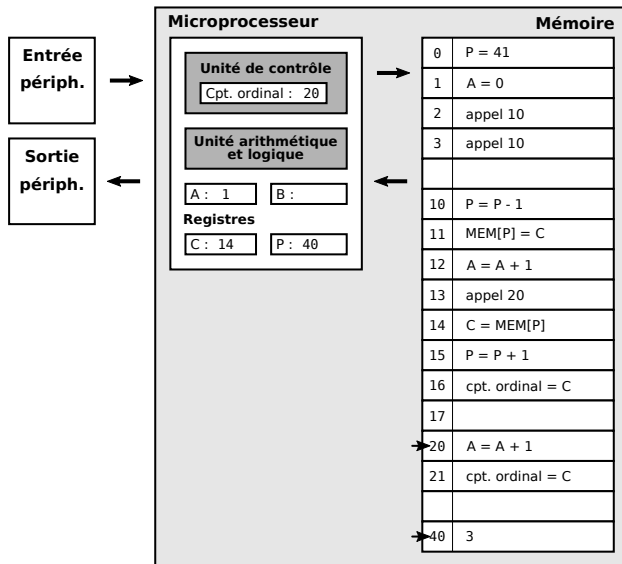
Support des procédures : exécution



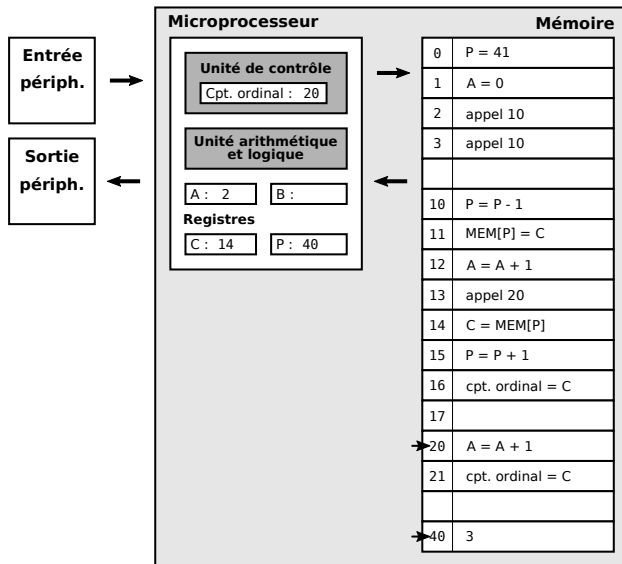
Support des procédures : exécution



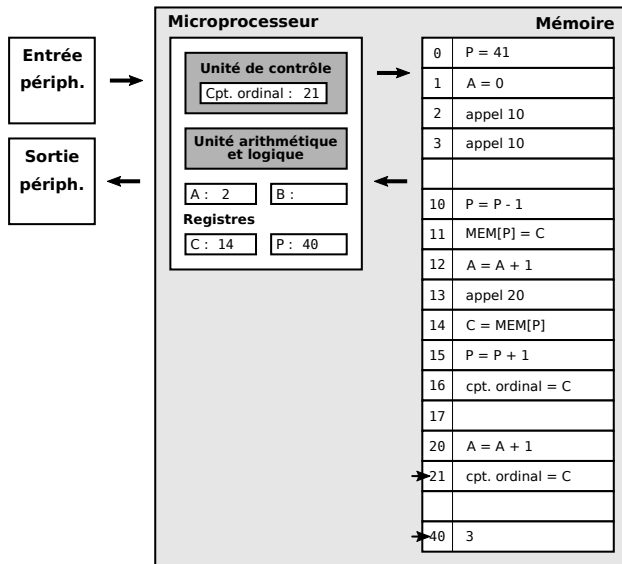
Support des procédures : exécution



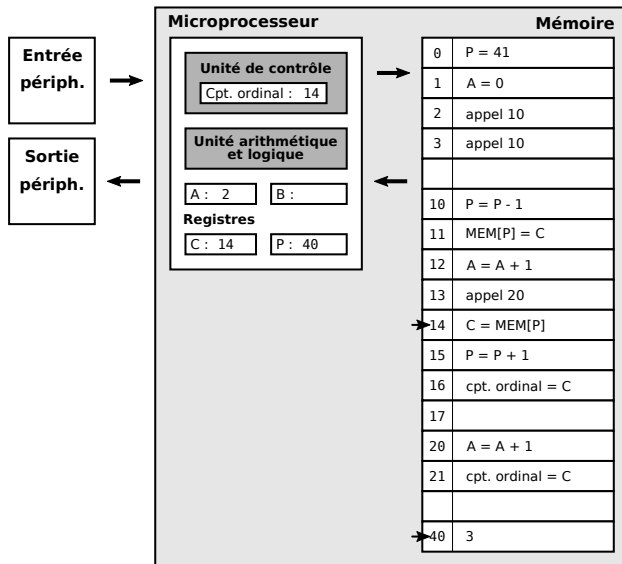
Support des procédures : exécution



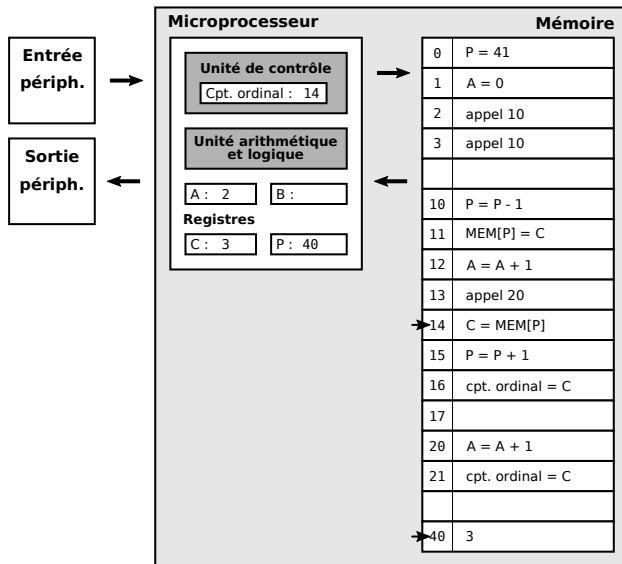
Support des procédures : exécution



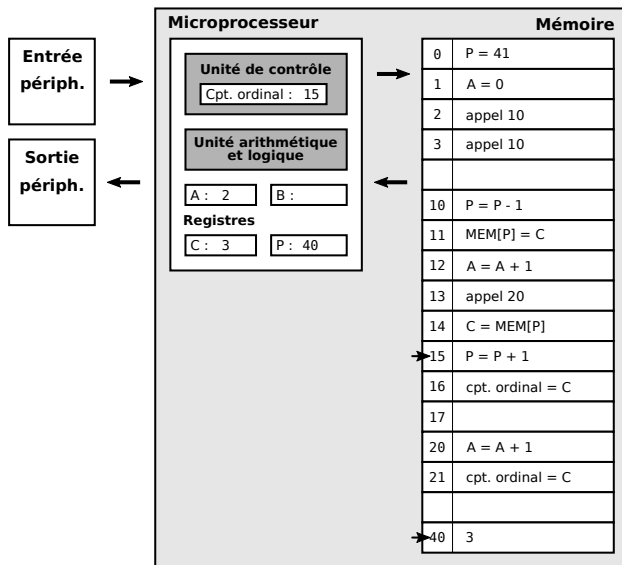
Support des procédures : exécution



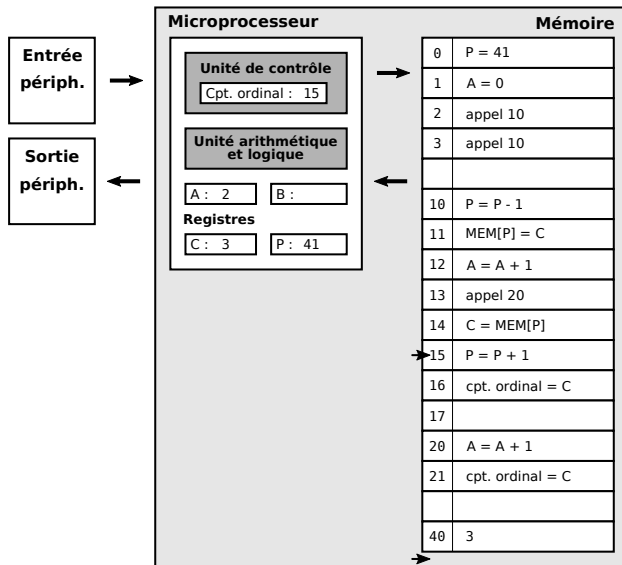
Support des procédures : exécution



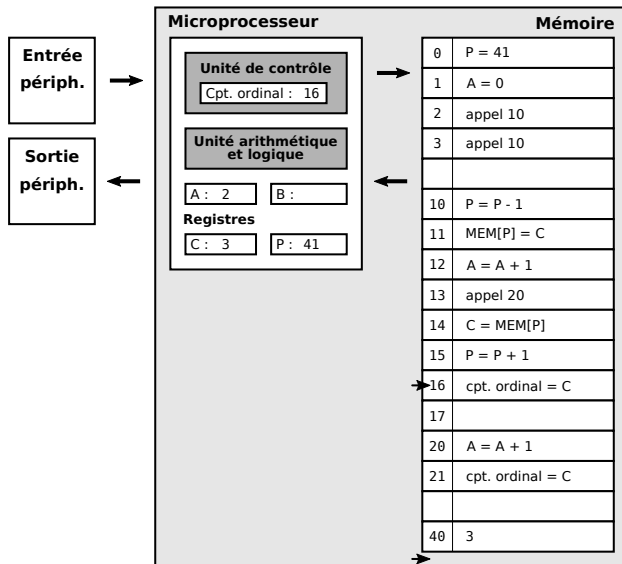
Support des procédures : exécution



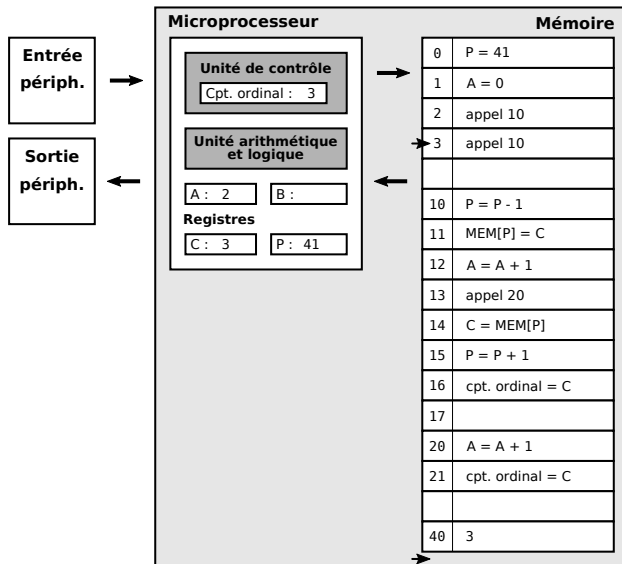
Support des procédures : exécution



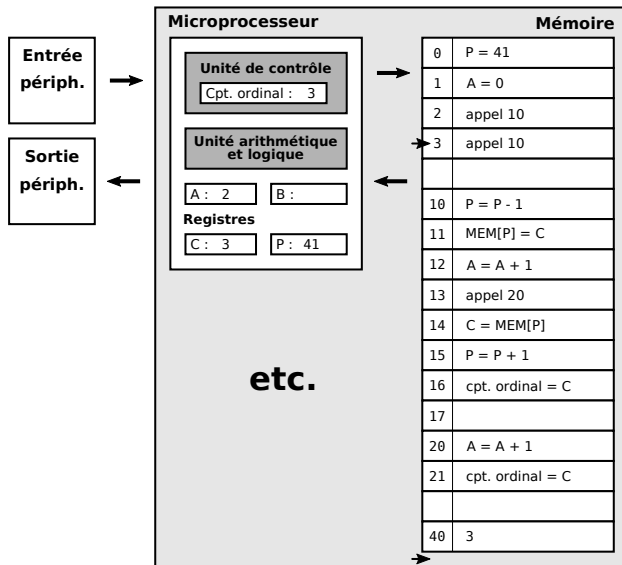
Support des procédures : exécution



Support des procédures : exécution



Support des procédures : exécution



Structure d'une instruction 1/2

Les programmes sont stockés en mémoire. Quelle est la structure d'une instruction ?

Champs usuels

Code op.	Operandes
----------	-----------

- ▶ Un code d'opération $\simeq$ l'opération à exécuter
Peut spécifier la taille des opérandes
Exemple : addition, appel, cpt. ord =, etc.
- ▶ Des opérandes $\simeq$ les paramètres de l'instruction Exemple :
adresse absolue, registre, etc.

Taille d'une instruction

- ▶ Jeu d'instruction à taille fixe
Exemple : ARM, MIPS
- ▶ Jeu d'instruction à taille variable
Exemple : Intel x86

Structure d'une instruction 2/2

Sémantique et structure d'une opérande

- ▶ Généralement interprétés en fonction du code opération
- ▶ Identifiant de registre
Exemple : 0 = A, 1 = B, etc.
- ▶ Valeur immédiate
Exemple : A = 4
- ▶ Adresse
Exemple : appel 20

Classification des processeurs

- ▶ *Reduced Instruction Set Computer* (RISC) : ARM Thumb, MIPS
Peu d'instructions, faible empreinte silicium.
- ▶ *Complex Instruction Set Computer* (CISC) : Intel X86, ARMv8
Instructions complexes simplifiantes. Énorme empreinte silicium.

Jeu d'instruction d'un processeur généraliste 1/2

Dumb16 : processeur 16-bit généraliste MIPS, jeu d'instruction de taille fixe : 32-bit [REF]. UAL sans drapeaux.

Op. Code	Mnémonique	Description
0x1	add	$r1 = r2 + r3$
0x2	sub	$r1 = r2 - r3$
0x3	shl	$r1 = r2 \ll r3$
0x4	shr	$r1 = r2 \gg r3$
0x5	or	$r1 = r2 \text{ or } r3$
0x6	and	$r1 = r2 \text{ and } r3$
0x7	equ	$r1 = r2 == r3$
0x8	lte	$r1 = r2 \leq r3$
0x9	gte	$r1 = r2 \geq r3$
0xd0	afc	$r1 = \text{valeur immédiate}$
0xd1	cop	$r1 = r2$
0xe1	lod	$r1 = \text{MEM}[\text{adresse}]$
0xe2	str	$\text{MEM}[\text{adresse}] = r1$
0xe3	lop	$r1 = \text{MEM}[r2]$
0xe4	stp	$\text{MEM}[r1] = r2$
0xf1	jmp	cpt. ordinal = adresse
0xf2	jmz	cpt. ordinal = adresse ssi $r0 == 0$
0xf1	jmp	cpt. ordinal = $\text{MEM}[\text{adresse}]$

Jeu d'instruction d'un processeur généraliste 2/2

Structure des instructions :

0	7	8	15	16	23	24	31
Ignoré				Opé. 1		Code op.	
Ignoré		Opé. 2		Opé. 1		Code op.	
Opé. 3		Opé. 2		Opé. 1		Code op.	
Valeur immédiate				Opé. 1		Code op.	
adresse				Opé. 1		Code op.	

Exemple : instruction `add` :

`r1 = r2 + r3`

Jeu d'instruction d'un processeur généraliste 2/2

Structure des instructions :

0	7	8	15	16	23	24	31
Ignoré				Opé. 1		Code op.	
Ignoré		Opé. 2		Opé. 1		Code op.	
Opé. 3		Opé. 2		Opé. 1		Code op.	
Valeur immédiate				Opé. 1		Code op.	
adresse				Opé. 1		Code op.	

Exemple : instruction add :

$r1 = r2 + r3$

0	7	8	15	16	23	24	31
ope. 2		ope. 1		ope. 0		code op.	
0x3		0x02		0x01		0x01	
r3		r2		r1		add	

Jeux d'instruction d'un processeur spécialisé 1/3

FAC : processeur 64-bit spécialisé dans le *pattern matching*, jeu d'instruction de taille variable [REF]. UAL sans drapeaux.

Op. Code	Mnémonique	Description
0x1	mask	Jump if mask not verified
0x2	equ	Jump if not equal
0x3	inf	Jump if not lower
0x4	add	Add two registers
0x5	xor	Xor two registers
0x6	hamm	Hamming distance
0xc	int	Interruption
0xd	mload	Memory load
0xe	load	Immediate load
0xf	jmp	Jump

Instruction mask

mask	r_val	r_m0	r_m1	r_adr
------	-------	------	------	-------

Saute à l'adresse stockée dans le registre **r_adr** ssi les masques des registres **r_m0** et **r_m1** ne s'appliquent pas à **r_val**

Jeux d'instruction d'un processeur spécialisé 2/3

Format des instructions

0			7	8		15	16		23	24		31	32		39
Code Op.															
Taille oper.	Sous op.	Op.	Opér. 1												
Taille oper.	Sous op.	Op.	Opér. 1		Opér. 2										
Taille oper.	Sous op.	Op.	Opér. 1		Opér. 2		Opér. 3								
Taille oper.	Sous op.	Op.	Opér. 1		Opér. 2		Opér. 3		Opér. 4						
0x0	Sous op.	Op.	Valeur												
0x1	Sous op.	Op.	Valeur												
0x2	Sous op.	Op.	Valeur												
0x3	Sous op.	Op.	Valeur												

Format des opérandes d'une instruction

0			7
Opérande			
0	2	3	7
Sélecteur	Registre		

Jeux d'instruction d'un processeur spécialisé 3/3

Exemple : instruction mask avec des opérandes de 16-bit :
 sauter à `r4[15:0]` si `r1[47:32]` (0 autorisés) et `r2[63:48]` (1 autorisés) ne s'appliquent pas sur la valeur `r0[31:16]`.

0	1	2	3	4	7	8	10	11	15	16	18	19	23	24	26	27	31	32	34	35	39								
code opération										ope. 0					ope. 1					ope. 2					ope. 3				

Jeux d'instruction d'un processeur spécialisé 3/3

Exemple : instruction mask avec des opérandes de 16-bit :
 sauter à `r4[15:0]` si `r1[47:32]` (0 autorisés) et `r2[63:48]` (1 autorisés) ne s'appliquent pas sur la valeur `r0[31:16]`.

0	1	2	3	4	7	8	10	11	15	16	18	19	23	24	26	27	31	32	34	35	39
code opération					ope. 0		ope. 1		ope. 2		ope. 3										
taille ope.		sous op.		op.	sel.	reg.	sel.	reg.	sel.	reg.	sel.	reg.									

Jeux d'instruction d'un processeur spécialisé 3/3

Exemple : instruction mask avec des opérandes de 16-bit :
 sauter à `r4[15:0]` si `r1[47:32]` (0 autorisés) et `r2[63:48]` (1 autorisés) ne s'appliquent pas sur la valeur `r0[31:16]`.

0	1	2	3	4	7	8	10	11	15	16	18	19	23	24	26	27	31	32	34	35	39
code opération						ope. 0		ope. 1		ope. 2		ope. 3									
taille ope.		sous op.		op.		sel.	reg.	sel.	reg.	sel.	reg.	sel.	reg.								
0x11						0x01		0x0a		0x13		0x20									

Jeux d'instruction d'un processeur spécialisé 3/3

Exemple : instruction mask avec des opérandes de 16-bit :
 sauter à `r4[15:0]` si `r1[47:32]` (0 autorisés) et `r2[63:48]` (1 autorisés) ne s'appliquent pas sur la valeur `r0[31:16]`.

0	1	2	3	4	7	8	10	11	15	16	18	19	23	24	26	27	31	32	34	35	39	
code opération						ope. 0		ope. 1		ope. 2		ope. 3										
taille ope.		sous op.		op.		sel.	reg.	sel.	reg.	sel.	reg.	sel.	reg.									
0x11						0x01		0x0a		0x13		0x20										
1 (16-bit)		0		1 (mask)																		

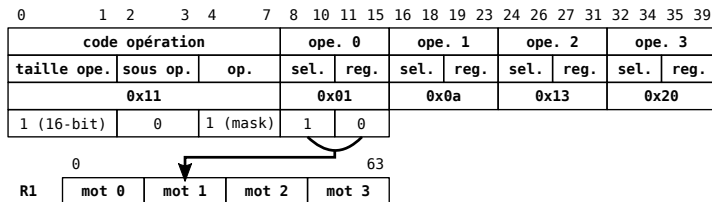
Jeux d'instruction d'un processeur spécialisé 3/3

Exemple : instruction mask avec des opérandes de 16-bit :
sauter à `r4[15:0]` si `r1[47:32]` (0 autorisés) et `r2[63:48]` (1 autorisés) ne s'appliquent pas sur la valeur `r0[31:16]`.

0	1	2	3	4	7	8	10	11	15	16	18	19	23	24	26	27	31	32	34	35	39
code opération					ope. 0		ope. 1		ope. 2		ope. 3										
taille ope.	sous op.		op.		sel.	reg.	sel.	reg.	sel.	reg.	sel.	reg.									
0x11					0x01		0x0a		0x13		0x20										
1 (16-bit)		0		1 (mask)		1	0														

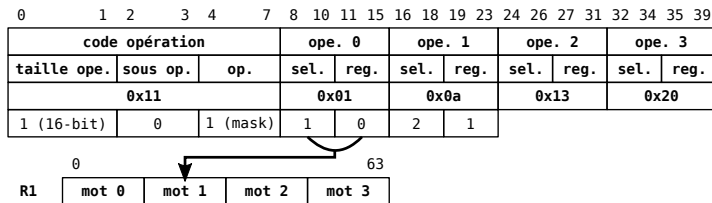
Jeux d'instruction d'un processeur spécialisé 3/3

Exemple : instruction mask avec des opérandes de 16-bit :
sauter à r4[15:0] si r1[47:32] (0 autorisés) et r2[63:48] (1 autorisés) ne s'appliquent pas sur la valeur r0[31:16].



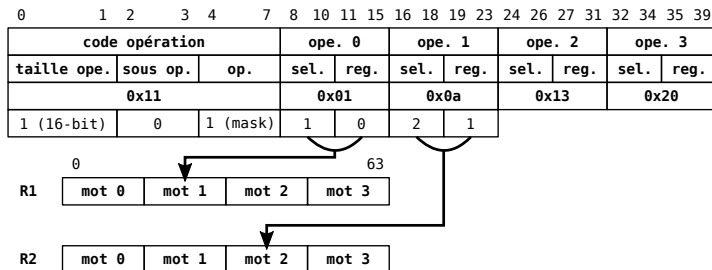
Jeux d'instruction d'un processeur spécialisé 3/3

Exemple : instruction mask avec des opérandes de 16-bit :
sauter à r4[15:0] si r1[47:32] (0 autorisés) et r2[63:48] (1 autorisés) ne s'appliquent pas sur la valeur r0[31:16].



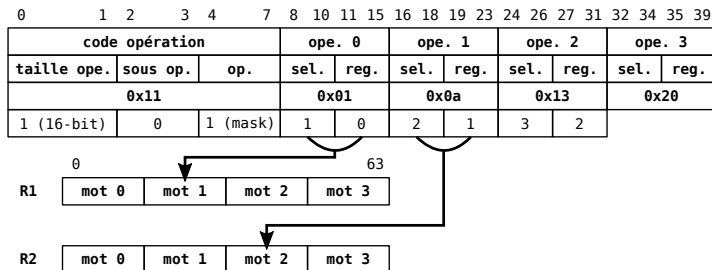
Jeux d'instruction d'un processeur spécialisé 3/3

Exemple : instruction mask avec des opérandes de 16-bit :
sauter à `r4[15:0]` si `r1[47:32]` (0 autorisés) et `r2[63:48]` (1 autorisés) ne s'appliquent pas sur la valeur `r0[31:16]`.



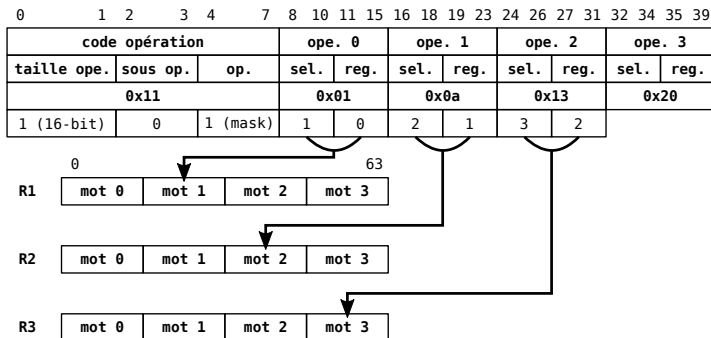
Jeux d'instruction d'un processeur spécialisé 3/3

Exemple : instruction mask avec des opérandes de 16-bit :
sauter à r4[15:0] si r1[47:32] (0 autorisés) et r2[63:48] (1 autorisés) ne s'appliquent pas sur la valeur r0[31:16].



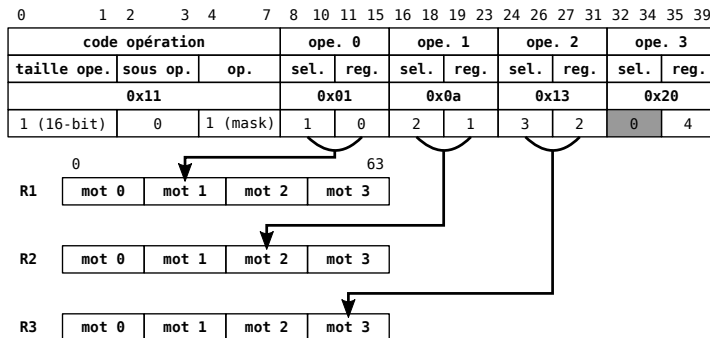
Jeux d'instruction d'un processeur spécialisé 3/3

Exemple : instruction mask avec des opérandes de 16-bit :
sauter à r4[15:0] si r1[47:32] (0 autorisés) et r2[63:48] (1 autorisés) ne s'appliquent pas sur la valeur r0[31:16].



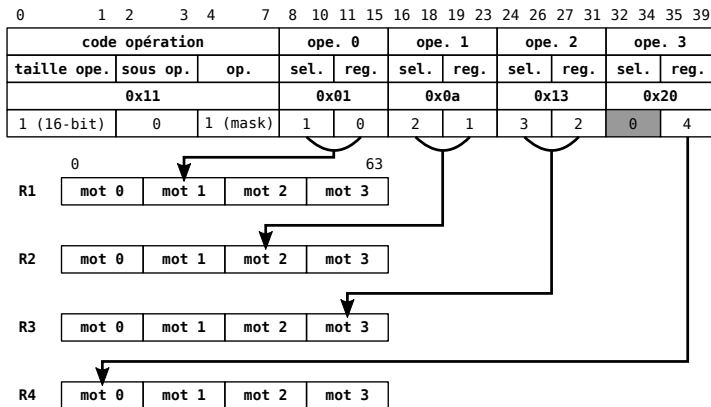
Jeux d'instruction d'un processeur spécialisé 3/3

Exemple : instruction mask avec des opérandes de 16-bit :
sauter à r4[15:0] si r1[47:32] (0 autorisés) et r2[63:48] (1 autorisés) ne s'appliquent pas sur la valeur r0[31:16].



Jeux d'instruction d'un processeur spécialisé 3/3

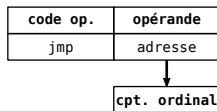
Exemple : instruction mask avec des opérandes de 16-bit :
sauter à r4[15:0] si r1[47:32] (0 autorisés) et r2[63:48] (1 autorisés) ne s'appliquent pas sur la valeur r0[31:16].



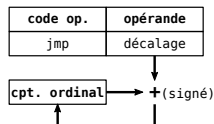
Modes d'adressage des branchements

Branchement $\Leftrightarrow$ saut $\Leftrightarrow$ mise à jour du compteur ordinal avec une **adresse** contrôlée par le programme.

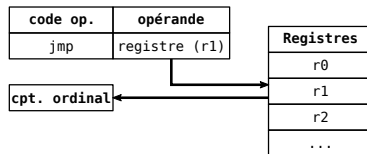
Modes d'adressage



Adressage absolu



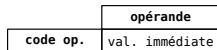
Adressage relatif



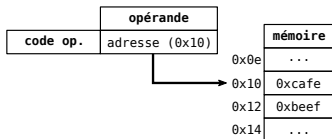
Adressage indirect

Modes d'adressage des données

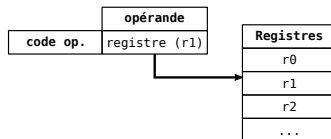
Comment est-ce que les données sont adressées par le processeur ?
Quelques modes typiques :



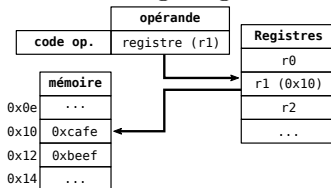
Adressage immédiat



Adressage absolu

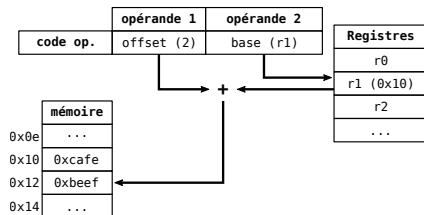


Adressage registre

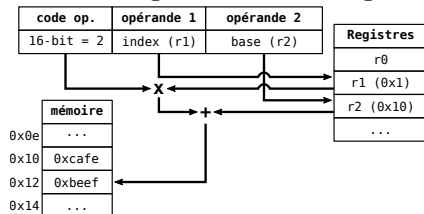


Adressage indirect registre

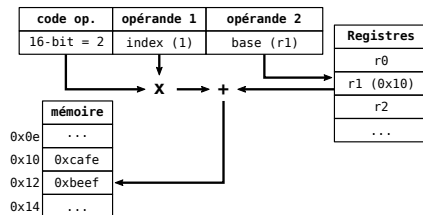
Modes d'adressage des données



Adressage base + décalage



Adressage base + index



Adressage base + index absolu

Plan du cours

Architectures matérielles des processeurs

Intel x86

Assembleur GNU

... en ligne

Généralités et culture générale

Architecture du processeur

- ▶ Processeur à jeu d'instructions étendu (CISC)
- ▶ Dépend du mode d'exécution
- ▶ Microcodé : correction de bugs (et vulnérabilités)
- ▶ 3 niveau de caches (dernier niveau partagé)
- ▶ Interconnexion PCI Express

Stratégie de rétrocompatibilité

- ▶ Support de modes historiques (8086)
 - ▶ Démarrage en mode historique
 - ▶ Configuration pas à pas des modes modernes
 - ▶ *Real mode* (16-bit) ; *protected mode* (32-bit) ; *long mode* (64-bit)
- ⇒ processeur très complexe, difficile d'obtenir des propriétés de sécurité

Architecture

Registres

- ▶ Pointeur d'instruction : `ip`
- ▶ Généraux : `a`, `b`, `c`, `d`, `r[8-15]` (IA32e, 64-bit)
+ `r[16-31]` (APX)
- ▶ Tête de pile : `sp`; base de pile : `bp`
- ▶ Index, copie mémoire : `si`, `di`
- ▶ Drapeaux d'état `[er]flags` : `c`, `z`, `o`, ...

Architecture

- ▶ Unité arithmétique et logique (ALU) de 64-bit (8 à 64)
- ▶ Unités mémoire
 - ▶ octet (`b`) : 8-bit, `[abcd][lh]` (l poid faible et h poid fort)
 - ▶ mot (`w`) : 16-bit, `[abcd]x`, `sp`, `bp`, `di`, ...
 - ▶ double mot (`l`) : 32-bit, `e[abcd]x`, `esp`, `ebp`, `edi`, ...
 - ▶ quadruple mot (`q`) : 64-bit, `r[abcd]x`, `rsp`, `rbp`, `rdi`, ...

Format générique d'une instruction x86

Instruction x86

1 byte each					
n prefixes	opcode	ModR/M	SIB	displacement	immediate

Taille maximale : 15 octets

ModR/M byte

7	6	5	3	2	0
Mod		Reg		R/M	

SIB byte

7	6	5	3	2	0
Scale		Index		Base	

- ▶ Mod : R/M is register or memory
- ▶ Reg : register operand
- ▶ R/M : register or memory operand
- ▶ SIB :

address = scale × index + base + displacement
 0x42 = 2 × rax = 0x10 + rbx = 0x20 + 0x2

Jeu d'instructions 1/5

Intel® 64 and IA-32 Architectures Software Developer's Manual

- ▶ La première opérande est la destination d'une instruction
- ▶ Une des deux opérandes peut en général être de la mémoire (ModR/M byte)
- ▶ Cette section suit la logique de la documentation du programmeur Intel x86 : **la première opérande est celle la plus à gauche**
- ▶ **/!\ Ce n'est plus vrai pour l'assembleur GNU /!**

Jeu d'instructions 2/5

Mémoire / registres

- ▶ `mov <src>, <dst>` : accès direct à la mémoire, affectations
- ▶ `push <src>, pop <dst>` : gestion de la pile
- ▶ `movs` : copie directe mémoire ($\approx$ `memcpy`, `@si++` vers `@di++`)
`rep movsb`, exécute `c` fois `movsb`

Arithmétique (mise à jour des drapeaux d'état)

- ▶ `and, or, xor` : opérateurs bit à bit
- ▶ `test <a>, <b>` : `<a> and <b>` bit à bit
- ▶ `cmp <a>, <b>` : `<b> - <a>` : compare `b` avec `a`
Utilisé en conjonction avec `jcc`.
Ex : `cmp a b; jg <etiquette>` $\leftrightarrow$ branchement ssi `a > b`
- ▶ `add, mul, sub, div` : arithmétique, `<dst> = <src> op <dst>`
- ▶ `shl, shr` : décalage à gauche / droite, `<dst> = <dst> op <dec>`

Jeu d'instructions 3/5

Adressage des accès mémoire

Adressage implicite

Exemple : push, pop, movs

- ▶ push <src> : $\text{mem}[-\text{sp}] = \text{<src>}$
- ▶ pop <dst> : $\text{<dst>} = \text{mem}[\text{sp}++]$

Adressage explicites

- ▶ adressage absolu
- ▶ adressage relatif
- ▶ adressage base + décalage
- ▶ PAS d'adressage base + index absolu
- ▶ adressage base + index

Jeu d'instructions 4/5

Instruction principale

- ▶ Sauts inconditionnels : `jmp`
- ▶ Saut conditionnels `jcc` :
 - `je adresse si [er]flags.z = 1`
 - `jne adresse si [er]flags.z = 0 ;`

Adressage mémoire simple des branchements

- ▶ Relatif
- ▶ Branchement indirect
- ▶ Absolu (**l**`jmp`)

Branchements mémoire complexes (CISC)

`ip` = accès mémoire

- ▶ Adressage absolu
- ▶ Adressage indirect registre

Jeu d'instructions 5/5

Fonctions

Appel de fonction

- Instruction : `call <adresse>`
- 1. Empile adresse de retour = `ip + taille instruction`
- 2. Saut inconditionnel à l'adresse de la fonction :
 $ip = \text{adresse fonction}$

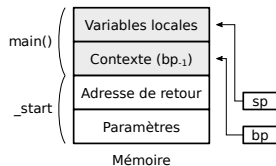
Retour de fonction

- Instruction : `ret`
- 1. Saut inconditionnel à l'adresse de retour : $ip = \text{mem}[sp]$
- 2. Depile l'adresse de retour : $sp = sp + \text{taille adresse}$

Support des contextes de fonction 1/2

Stack frame

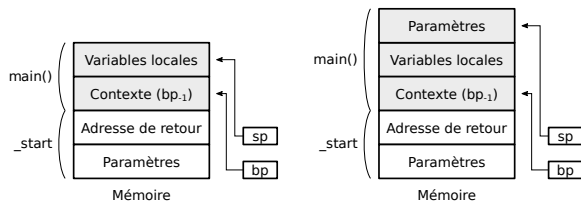
- ▶ Délimitation de l'espace mémoire d'une fonction
 - ▶ De la base de pile : **bp**, à la tête de pile : **sp**
- ⇒ Adressage simplifié des variables locales : **bp** constant



Support des contextes de fonction 1/2

Stack frame

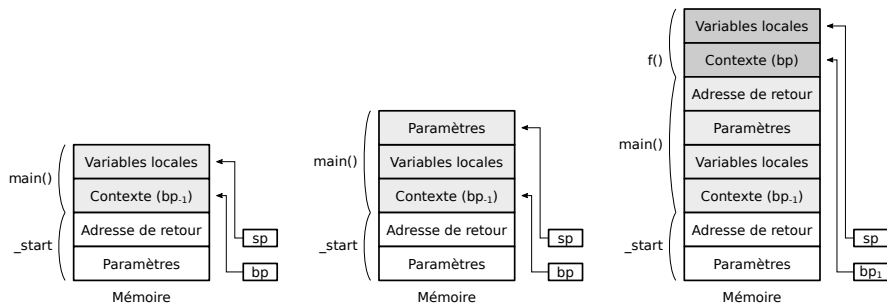
- ▶ Délimitation de l'espace mémoire d'une fonction
 - ▶ De la base de pile : **bp**, à la tête de pile : **sp**
- ⇒ Adressage simplifié des variables locales : **bp** constant



Support des contextes de fonction 1/2

Stack frame

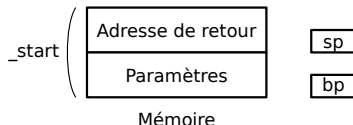
- Délimitation de l'espace mémoire d'une fonction
- De la base de pile : **bp**, à la tête de pile : **sp**
- ⇒ Adressage simplifié des variables locales : **bp** constant



Support des contextes de fonction 2/2

Gestion du contexte par une fonction

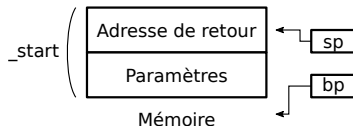
- ▶ En début de fonction
 1. Sauvegarde de la base de pile
`push bp`
 2. Base de pile = tête de pile
`bp = sp (mov)`
 3. Allocation des variable locales
`sp = sp - <taille> (sub)`
- ▶ En fin de fonction
 1. Libération des variable locales
`sp = sp + <taille> (add)`
 2. Restauration du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

Gestion du contexte par une fonction

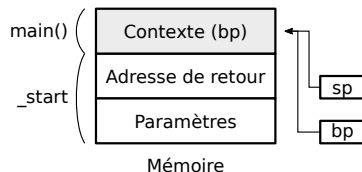
- ▶ En début de fonction
 1. Sauvegarde de la base de pile
`push bp`
 2. Base de pile = tête de pile
`bp = sp (mov)`
 3. Allocation des variable locales
`sp = sp - <taille> (sub)`
- ▶ En fin de fonction
 1. Libération des variable locales
`sp = sp + <taille> (add)`
 2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

Gestion du contexte par une fonction

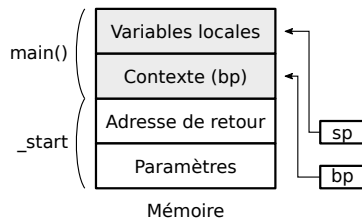
- ▶ En début de fonction
 1. Sauvegarde de la base de pile
`push bp`
 2. Base de pile = tête de pile
`bp = sp (mov)`
 3. Allocation des variable locales
`sp = sp - <taille> (sub)`
- ▶ En fin de fonction
 1. Libération des variable locales
`sp = sp + <taille> (add)`
 2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

Gestion du contexte par une fonction

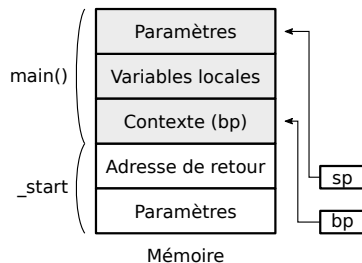
- ▶ En début de fonction
 1. Sauvegarde de la base de pile
`push bp`
 2. Base de pile = tête de pile
`bp = sp (mov)`
 3. Allocation des variable locales
`sp = sp - <taille> (sub)`
- ▶ En fin de fonction
 1. Libération des variable locales
`sp = sp + <taille> (add)`
 2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

Gestion du contexte par une fonction

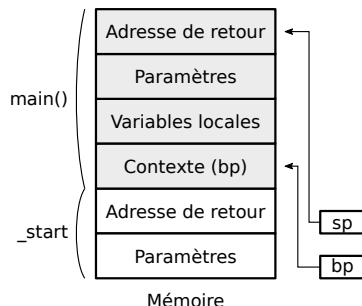
- En début de fonction
 1. Sauvegarde de la base de pile
`push bp`
 2. Base de pile = tête de pile
`bp = sp (mov)`
 3. Allocation des variable locales
`sp = sp - <taille> (sub)`
- En fin de fonction
 1. Libération des variable locales
`sp = sp + <taille> (add)`
 2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

Gestion du contexte par une fonction

- En début de fonction
 1. Sauvegarde de la base de pile
`push bp`
 2. Base de pile = tête de pile
`bp = sp (mov)`
 3. Allocation des variable locales
`sp = sp - <taille> (sub)`
- En fin de fonction
 1. Libération des variable locales
`sp = sp + <taille> (add)`
 2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

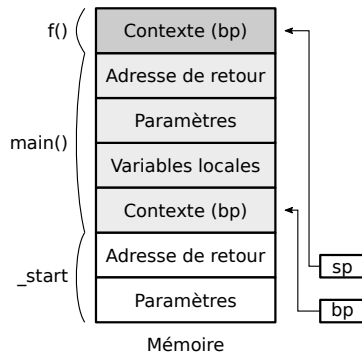
Gestion du contexte par une fonction

► En début de fonction

1. Sauvegarde de la base de pile
`push bp`
2. Base de pile = tête de pile
`bp = sp (mov)`
3. Allocation des variable locales
`sp = sp - <taille> (sub)`

► En fin de fonction

1. Libération des variable locales
`sp = sp + <taille> (add)`
2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

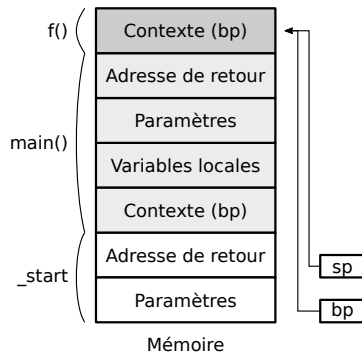
Gestion du contexte par une fonction

► En début de fonction

1. Sauvegarde de la base de pile
`push bp`
2. Base de pile = tête de pile
`bp = sp (mov)`
3. Allocation des variable locales
`sp = sp - <taille> (sub)`

► En fin de fonction

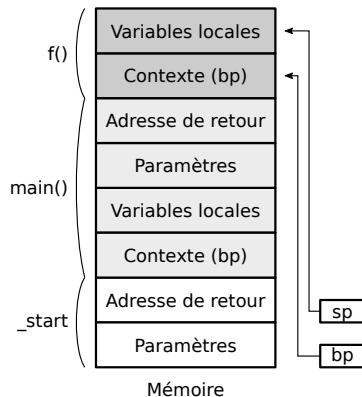
1. Libération des variable locales
`sp = sp + <taille> (add)`
2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

Gestion du contexte par une fonction

- En début de fonction
 1. Sauvegarde de la base de pile
`push bp`
 2. Base de pile = tête de pile
`bp = sp (mov)`
 3. Allocation des variable locales
`sp = sp - <taille> (sub)`
- En fin de fonction
 1. Libération des variable locales
`sp = sp + <taille> (add)`
 2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

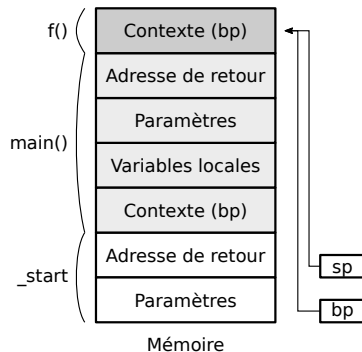
Gestion du contexte par une fonction

► En début de fonction

1. Sauvegarde de la base de pile
`push bp`
2. Base de pile = tête de pile
`bp = sp (mov)`
3. Allocation des variable locales
`sp = sp - <taille> (sub)`

► En fin de fonction

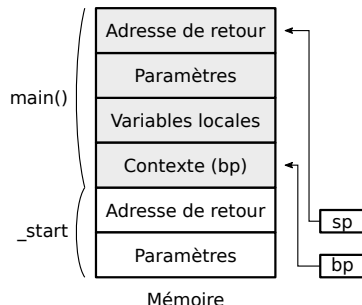
1. Libération des variable locales
`sp = sp + <taille> (add)`
2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

Gestion du contexte par une fonction

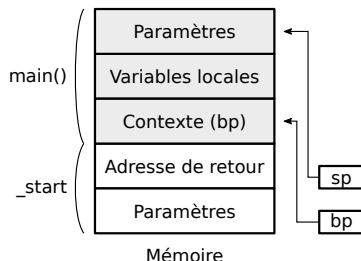
- En début de fonction
 1. Sauvegarde de la base de pile
`push bp`
 2. Base de pile = tête de pile
`bp = sp (mov)`
 3. Allocation des variable locales
`sp = sp - <taille> (sub)`
- En fin de fonction
 1. Libération des variable locales
`sp = sp + <taille> (add)`
 2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

Gestion du contexte par une fonction

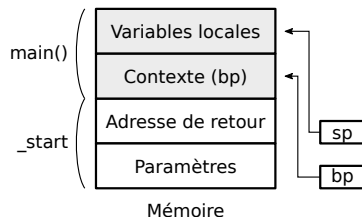
- En début de fonction
 1. Sauvegarde de la base de pile
`push bp`
 2. Base de pile = tête de pile
`bp = sp (mov)`
 3. Allocation des variable locales
`sp = sp - <taille> (sub)`
- En fin de fonction
 1. Libération des variable locales
`sp = sp + <taille> (add)`
 2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

Gestion du contexte par une fonction

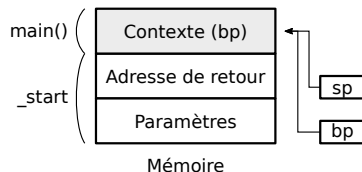
- ▶ En début de fonction
 1. Sauvegarde de la base de pile
`push bp`
 2. Base de pile = tête de pile
`bp = sp (mov)`
 3. Allocation des variable locales
`sp = sp - <taille> (sub)`
- ▶ En fin de fonction
 1. Libération des variable locales
`sp = sp + <taille> (add)`
 2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

Gestion du contexte par une fonction

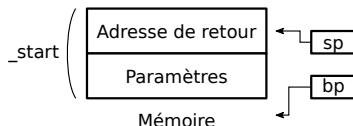
- ▶ En début de fonction
 1. Sauvegarde de la base de pile
`push bp`
 2. Base de pile = tête de pile
`bp = sp (mov)`
 3. Allocation des variable locales
`sp = sp - <taille> (sub)`
- ▶ En fin de fonction
 1. Libération des variable locales
`sp = sp + <taille> (add)`
 2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Support des contextes de fonction 2/2

Gestion du contexte par une fonction

- ▶ En début de fonction
 1. Sauvegarde de la base de pile
`push bp`
 2. Base de pile = tête de pile
`bp = sp (mov)`
 3. Allocation des variable locales
`sp = sp - <taille> (sub)`
- ▶ En fin de fonction
 1. Libération des variable locales
`sp = sp + <taille> (add)`
 2. Resturation du contexte :
`sp = bp et pop bp`
Instruction `leave`



Plan du cours

Architectures matérielles des processeurs

Intel x86

Assembleur GNU

... en ligne

Assembleur GNU

- ▶ Assembleur du projet GCC (as)
- ▶ Supporte une multitude d'architectures Intel x86, ARMv8, RISC-V, etc.
- ▶ Logiciel libre avec grande communauté
- ▶ Utilisé dans le code source de Linux et de GNU



```
.global _start                # Export du symbole
.text                         # Section de code de l'exécutable

_start:                       # Définition du point d'entrée
    # write(1, message, 13)
    mov     $1, %rax          # Write est l'appel 1
    mov     $1, %rdi          # STDOUT_FILENO est 1
    mov     $message, %rsi    # On affecte l'adresse de la chaîne
    mov     $13, %rdx         # Le nombre de caractères à écrire
    syscall                  # Appel système !

    jmp exit                  # Fin du programme

message:
    .ascii  "Hello, world\n"
```

Syntaxe générale

Étiquette : symbole associé à une adresse

`<symbole>:`

Exemple :

`_start:`

Directive assembleur : interaction avec l'assembleur

`.<directive> <param1> <param2> ...`

Exemple :

`.global _start`

Instruction

L'operande de destination est à droite

`<instruction> <opn>, ... (<op1>)`

Exemple :

`add %rax, %rbx # rbx = rbx - rax`

Syntaxe des opérandes

Registres

Préfixés avec un %

%<reg>

Exemple :

```
add %al, %ah # ah = al + ah
```

Valeur immédiate

Préfixée avec un dollar

\$<nombre>

\$<etiquette>

\$'<char>'

Exemple :

```
movw $0xcafe, %ax # ax = 0xcafe
```


Taille des opérandes

Nom de registre

Le nom du registre détermine la taille sous Intel x86

`%al`, `%ah` → 8-bit

`%ax` → 16-bit

`%eax` → 32-bit

`%rax` → 64-bit

Suffixe d'instruction

Permet de traiter des cas où l'on ne peut pas déterminer la taille

`movb` → 8-bit

`movw` → 16-bit

`movd` → 32-bit

`movq` → 64-bit

Exemple :

`pushd $0x0`

Adressage des branchements 1/2

Adressage relatif

```
jmp _start      # ip = ip + décalage  
décalage = ip - _start, calculé à la compilation  
jmp 0x1234      # ip = ip + 0x1234
```

Branchement indirect

```
jmp %rax        # ip = %rax
```

Branchement avec accès mémoire absolu

```
jmp *0x1234      # ip = MEM[0x1234]  
jmp *pointeur    # ip = MEM[pointeur]
```

Branchement avec accès mémoire indirect

```
jmp *(%rax)      # ip = MEM[rax]
```

Adressage des branchements 2/2

DEMO TIME : `1st/hello-world`

Cheat sheet GDB

<https://benoit.m0rgan.net/assets/supports/gdb-cheat-sheet.pdf>

Adressage des données 1/2

Adressage absolu

```
mov pointeur, %rax          # rax = MEM[pointeur]
```

Adressage indirect registre

```
mov (%rax), %rax           # rax = MEM[rax]
```

Adressage base + décalage

```
movw 1(%rax), %ax          # ax = MEM[rax]
```

Adressage base + index

```
mov $2, %rbx  
mov (%rax, %rbx, 2), %ax    # ax = MEM[rax + rbx × 2]
```

Adressage base + index + décalage

```
mov $2, %rbx  
mov 2(%rax, %rbx, 2), %ax   # ax = MEM[rax + 2 + rbx × 2]
```

Adressage des données 2/2

DEMO TIME : `1st/hello-world`

Cheat sheet GDB

<https://benoit.m0rgan.net/assets/supports/gdb-cheat-sheet.pdf>

Interface de programmation ou API

Application Programming Interface

- ▶ Concept générique d'interface
- ▶ Niveau sémantiques $\neq$
- ▶ *Restful API HTTP*
- ▶ Prototypes C, classes C++, etc.

```
int execve(const char *pathname, char *const argv[],  
           char *const envp[]);
```

Type d'une fonction

Visible et manipulée par le développeur

VS Interface de programmation binaire ou ABI

Application Binary Interface

- ▶ Mise en œuvre concrète d'interfaces, proche de la machine
- ▶ Allocation des arguments et des valeurs de retour
- ▶ 32-bit : GCC `cdecl`
Arguments adressés de droite à gauche dans la pile
Valeur de retour dans la pile
Pile nettoyée par l'appelant
- ▶ 64-bit : System V ABI ; Microsoft x64 calling convention
Utilisation de registres dans un ordre donné
puis de la pile en mode `cdecl` quand plus de place

Mise en œuvre par le compilateur

Exemple : System V ABI, x86 64-bit

Contrat d'interface appelant / appelé

- ▶ Les fonction sont appelées par `call` et retournent à l'aide de `ret`
- ▶ La pile est alignée sur 16 octets avant d'appeler une fonction
- ▶ Les registres `rbx`, `rsp`, `rbp`, `r12`, `r13`, `r14`, et `r15` doivent être conservés par la fonction
- ▶ La pile descend. Les paramètres placés sur la pile peuvent être modifiés par la fonction appelée

Arg #	Emplacement
Retour	<code>rax</code> si 128-bit [127:64] dans <code>rdx</code>
1	<code>rdi</code>
2	<code>rsi</code>
3	<code>rdx</code>
4	<code>rcx</code>
5	<code>r8</code>
6	<code>r9</code>
> 6	Placés dans la pile en sens inverse

Exemple : System V ABI, x86 64-bit

Fonctionnement de l'ABI exemple.

```
int test (      | int main() {
    int a1,      |     test(
    int a2,      |         1,
    int a3,      |         2,
    int a4,      |         3,
    int a5,      |         4,
    int a6,      |         5,
    int a7,      |         6,
    int a8,      |         7,
    int a9,      |         8,
    int a10,     |         9,
    int a11,     |        10,
    int a12,     |        11,
    int a13,     |        12,
    int a14) {   |        13,
return 0;        |        14);
}                |     return 0;
                  | }

```

```
$ gcc -O0 -o test test.c -g
```

Exemple : System V ABI, x86 64-bit

```

0x55555555119 <test>      push    %rbp
0x5555555511a <test+1>    mov     %rsp,%rbp
0x5555555511d <test+4>    mov     %edi,-0x4(%rbp)
0x55555555120 <test+7>    mov     %esi,-0x8(%rbp)
0x55555555123 <test+10>   mov     %edx,-0xc(%rbp)
0x55555555126 <test+13>   mov     %ecx,-0x10(%rbp)
0x55555555129 <test+16>   mov     %r8d,-0x14(%rbp)
0x5555555512d <test+20>   mov     %r9d,-0x18(%rbp)
0x55555555131 <test+24>   mov     $0x0,%eax
0x55555555136 <test+29>   pop     %rbp
0x55555555137 <test+30>   retq

0x55555555138 <main>      push    %rbp
0x55555555139 <main+1>    mov     %rsp,%rbp
0x5555555513c <main+4>    pushq   $0xe
0x5555555513e <main+6>    pushq   $0xd
0x55555555140 <main+8>    pushq   $0xc
0x55555555142 <main+10>   pushq   $0xb
0x55555555144 <main+12>   pushq   $0xa
0x55555555146 <main+14>   pushq   $0x9
0x55555555148 <main+16>   pushq   $0x8
0x5555555514a <main+18>   pushq   $0x7
0x5555555514c <main+20>   mov     $0x6,%r9d
0x55555555152 <main+26>   mov     $0x5,%r8d
0x55555555158 <main+32>   mov     $0x4,%ecx
0x5555555515d <main+37>   mov     $0x3,%edx
0x55555555162 <main+42>   mov     $0x2,%esi
0x55555555167 <main+47>   mov     $0x1,%edi
0x5555555516c <main+52>   callq  0x55555555119 <test>
0x55555555171 <main+57>   add     $0x40,%rsp
0x55555555175 <main+61>   mov     $0x0,%eax
0x5555555517a <main+66>   leaveq
0x5555555517b <main+67>   ret

```

Exemple : appel de fonction C, ABI cdecl 1/2

Exemple de programme C (x86 32-bit) (Éric Alata)

```
void f(int a, char* str) {  
    char ch[8];  
    int var;  
}  
void main(int argc, char** argv) {  
    f(4, argv[1]);  
}
```

Exemple : appel de fonction C, ABI cdecl 2/2

Décompilation du binaire : `objdump -d program` 2

```
f :      08048cb0 push  %ebp
        08048cb1 mov   %esp,%ebp
        08048cb3 sub   $0x10,%esp
        08048cb6 leave
        08048cb7 ret

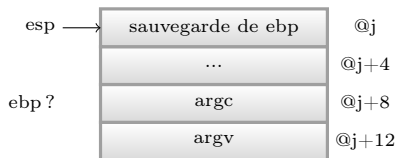
main : → 08048cb8 push  %ebp
        08048cb9 mov   %esp,%ebp
        08048cbb sub   $0x8,%esp
        08048cbe mov   0xc(%ebp),%eax
        08048cc1 add   $0x4,%eax
        08048cc4 mov   (%eax),%eax
        08048cc6 mov   %eax,0x4(%esp)
        08048cca movl  $0x4,(%esp)
        08048cd1 call  8048cb0
        08048cd6 leave
        08048cd7 ret
```



Exemple : appel de fonction C, ABI cdecl 2/2

Décompilation du binaire : `objdump -d program` 2

```
f :      08048cb0 push  %ebp
        08048cb1 mov  %esp,%ebp
        08048cb3 sub   $0x10,%esp
        08048cb6 leave
        08048cb7 ret
main :  08048cb8 push  %ebp
        → 08048cb9 mov  %esp,%ebp
        08048cbb sub   $0x8,%esp
        08048cbe mov   0xc(%ebp),%eax
        08048cc1 add   $0x4,%eax
        08048cc4 mov   (%eax),%eax
        08048cc6 mov   %eax,0x4(%esp)
        08048cca movl  $0x4,(%esp)
        08048cd1 call  8048cb0
        08048cd6 leave
        08048cd7 ret
```



Exemple : appel de fonction C, ABI cdecl 2/2

Décompilation du binaire : `objdump -d program` 2

```
f :      08048cb0  push  %ebp
        08048cb1  mov   %esp,%ebp
        08048cb3  sub   $0x10,%esp
        08048cb6  leave
        08048cb7  ret

main :  08048cb8  push  %ebp
        08048cb9  mov   %esp,%ebp
        → 08048cbb  sub   $0x8,%esp
        08048cbe  mov   0xc(%ebp),%eax
        08048cc1  add   $0x4,%eax
        08048cc4  mov   (%eax),%eax
        08048cc6  mov   %eax,0x4(%esp)
        08048cca  movl  $0x4,(%esp)
        08048cd1  call  8048cb0
        08048cd6  leave
        08048cd7  ret
```

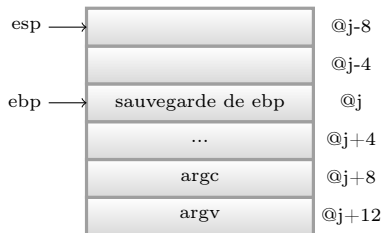


Exemple : appel de fonction C, ABI cdecl 2/2

Décompilation du binaire : `objdump -d program` 2

```
f :      08048cb0  push  %ebp
        08048cb1  mov   %esp,%ebp
        08048cb3  sub   $0x10,%esp
        08048cb6  leave
        08048cb7  ret

main :  08048cb8  push  %ebp
        08048cb9  mov   %esp,%ebp
        08048cbb  sub   $0x8,%esp
→ 08048cbe  mov   0xc(%ebp),%eax
→ 08048cc1  add   $0x4,%eax
→ 08048cc4  mov   (%eax),%eax
→ 08048cc6  mov   %eax,0x4(%esp)
        08048cca  movl  $0x4,(%esp)
        08048cd1  call  8048cb0
        08048cd6  leave
        08048cd7  ret
```

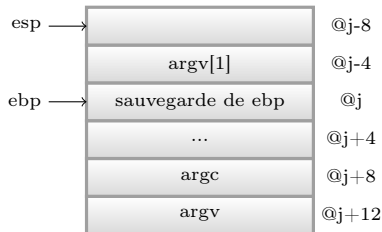


Exemple : appel de fonction C, ABI cdecl 2/2

Décompilation du binaire : `objdump -d program` 2

```
f :      08048cb0  push  %ebp
        08048cb1  mov   %esp,%ebp
        08048cb3  sub   $0x10,%esp
        08048cb6  leave
        08048cb7  ret

main :  08048cb8  push  %ebp
        08048cb9  mov   %esp,%ebp
        08048cbb  sub   $0x8,%esp
        08048cbe  mov   0xc(%ebp),%eax
        08048cc1  add   $0x4,%eax
        08048cc4  mov   (%eax),%eax
        08048cc6  mov   %eax,0x4(%esp)
        → 08048cca  movl  $0x4,(%esp)
        08048cd1  call  8048cb0
        08048cd6  leave
        08048cd7  ret
```



Exemple : appel de fonction C, ABI cdecl 2/2

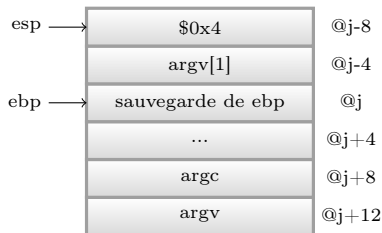
Décompilation du binaire : `objdump -d program` 2

```

f :      08048cb0  push  %ebp
        08048cb1  mov   %esp,%ebp
        08048cb3  sub   $0x10,%esp
        08048cb6  leave
        08048cb7  ret

main :   08048cb8  push  %ebp
        08048cb9  mov   %esp,%ebp
        08048cbb  sub   $0x8,%esp
        08048cbe  mov   0xc(%ebp),%eax
        08048cc1  add   $0x4,%eax
        08048cc4  mov   (%eax),%eax
        08048cc6  mov   %eax,0x4(%esp)
        08048cca  movl  $0x4,(%esp)
→ 08048cd1  call  8048cb0
        08048cd6  leave
        08048cd7  ret

```

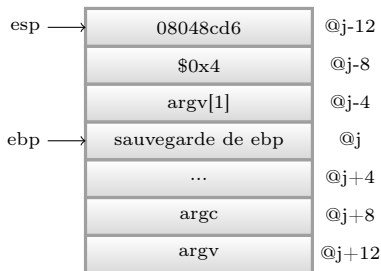


Exemple : appel de fonction C, ABI cdecl 2/2

Décompilation du binaire : `objdump -d program` 2

```
f :      → 08048cb0 push  %ebp
          08048cb1 mov   %esp,%ebp
          08048cb3 sub   $0x10,%esp
          08048cb6 leave
          08048cb7 ret

main :   08048cb8 push  %ebp
          08048cb9 mov   %esp,%ebp
          08048cbb sub   $0x8,%esp
          08048cbe mov   0xc(%ebp),%eax
          08048cc1 add   $0x4,%eax
          08048cc4 mov   (%eax),%eax
          08048cc6 mov   %eax,0x4(%esp)
          08048cca movl  $0x4,(%esp)
          08048cd1 call  8048cb0
          08048cd6 leave
          08048cd7 ret
```



Exemple : appel de fonction C, ABI cdecl 2/2

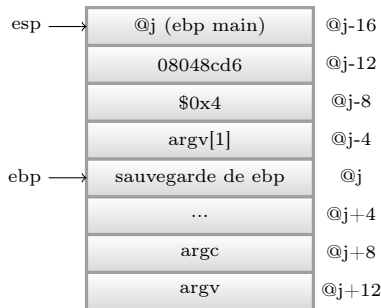
Décompilation du binaire : `objdump -d program` 2

```

f :      08048cb0  push  %ebp
        → 08048cb1  mov   %esp,%ebp
        08048cb3  sub   $0x10,%esp
        08048cb6  leave
        08048cb7  ret

main :   08048cb8  push  %ebp
        08048cb9  mov   %esp,%ebp
        08048cbb  sub   $0x8,%esp
        08048cbe  mov   0xc(%ebp),%eax
        08048cc1  add   $0x4,%eax
        08048cc4  mov   (%eax),%eax
        08048cc6  mov   %eax,0x4(%esp)
        08048cca  movl  $0x4,(%esp)
        08048cd1  call  8048cb0
        08048cd6  leave
        08048cd7  ret

```



Exemple : appel de fonction C, ABI cdecl 2/2

Décompilation du binaire : `objdump -d program` 2

```
f :      08048cb0 push  %ebp
        08048cb1 mov   %esp,%ebp
        → 08048cb3 sub   $0x10,%esp
        08048cb6 leave
        08048cb7 ret

main :  08048cb8 push  %ebp
        08048cb9 mov   %esp,%ebp
        08048cbb sub   $0x8,%esp
        08048cbe mov   0xc(%ebp),%eax
        08048cc1 add   $0x4,%eax
        08048cc4 mov   (%eax),%eax
        08048cc6 mov   %eax,0x4(%esp)
        08048cca movl  $0x4,(%esp)
        08048cd1 call  8048cb0
        08048cd6 leave
        08048cd7 ret
```

esp,ebp →

@j (ebp main)	@j-16
08048cd6	@j-12
\$0x4	@j-8
argv[1]	@j-4
sauvegarde de ebp	@j
...	@j+4
argc	@j+8
argv	@j+12

Exemple : appel de fonction C, ABI cdecl 2/2

Décompilation du binaire : `objdump -d program` 2

f :	08048cb0	push	%ebp	esp →		@j-32
	08048cb1	mov	%esp,%ebp			@j-28
	08048cb3	sub	\$0x10,%esp			@j-24
→	08048cb6	leave				@j-20
	08048cb7	ret				
main :	08048cb8	push	%ebp	ebp →	@j (ebp main)	@j-16
	08048cb9	mov	%esp,%ebp		08048cd6	@j-12
	08048cbb	sub	\$0x8,%esp		\$0x4	@j-8
	08048cbe	mov	Oxc(%ebp),%eax		argv[1]	@j-4
	08048cc1	add	\$0x4,%eax		sauvegarde de ebp	@j
	08048cc4	mov	(%eax),%eax		...	@j+4
	08048cc6	mov	%eax,Ox4(%esp)		argc	@j+8
	08048cca	movl	\$0x4,(%esp)		argv	@j+12
	08048cd1	call	8048cb0			
	08048cd6	leave				
	08048cd7	ret				

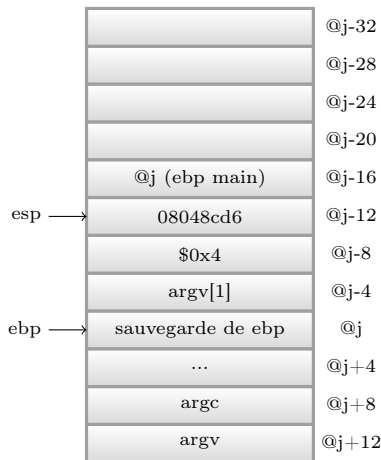
Exemple : appel de fonction C, ABI cdecl 2/2

Décompilation du binaire : `objdump -d program` 2

```

f :      08048cb0  push  %ebp
        08048cb1  mov   %esp,%ebp
        08048cb3  sub   $0x10,%esp
        08048cb6  leave
        → 08048cb7  ret
main :   08048cb8  push  %ebp
        08048cb9  mov   %esp,%ebp
        08048cbb  sub   $0x8,%esp
        08048cbe  mov   0xc(%ebp),%eax
        08048cc1  add   $0x4,%eax
        08048cc4  mov   (%eax),%eax
        08048cc6  mov   %eax,0x4(%esp)
        08048cca  movl  $0x4,(%esp)
        08048cd1  call  8048cb0
        08048cd6  leave
        08048cd7  ret

```



Exemple : appel de fonction C, ABI cdecl 2/2

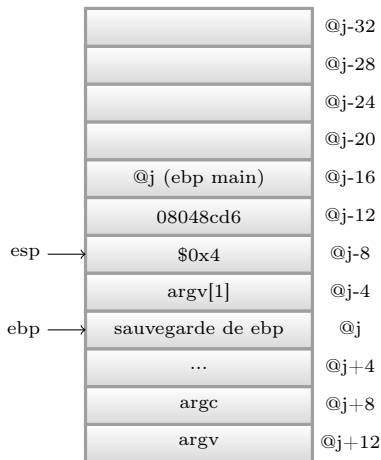
Décompilation du binaire : `objdump -d program` 2

```

f :      08048cb0  push  %ebp
        08048cb1  mov   %esp,%ebp
        08048cb3  sub   $0x10,%esp
        08048cb6  leave
        08048cb7  ret

main :   08048cb8  push  %ebp
        08048cb9  mov   %esp,%ebp
        08048cbb  sub   $0x8,%esp
        08048cbe  mov   0xc(%ebp),%eax
        08048cc1  add   $0x4,%eax
        08048cc4  mov   (%eax),%eax
        08048cc6  mov   %eax,0x4(%esp)
        08048cca  movl  $0x4,(%esp)
        08048cd1  call  8048cb0
→ 08048cd6  leave
    08048cd7  ret

```



Plan du cours

Architectures matérielles des processeurs

Intel x86

Assembleur GNU

... en ligne

Assembleur en ligne GNU

GNU inline Assembly (GAS)

```
char src[10] = "coucou !!", dst[10];  
asm volatile ("rep movsb" : :  
    "D"(&dst[0]), "S"(&src[0]), "c"(sizeof(dst)));  
puts("%s\n", dst);
```

Présentations

- ▶ Permet l'intégration de code assembleur dans du code C
- ▶ Primitive GNU asm. `__asm__` si compilateur non GNU.
- ▶ Deux modes spécifiques : *Basic Asm*; *Extended Asm*

Documentation officielle en ligne du projet GNU

<https://gcc.gnu.org/onlinedocs/gcc/Using-Assembly-Language-with-C.html>

Basic Asm

Syntaxe (grammaire)

```
asm <qualifiers> ( <instructions> )
```

instructions

Instructions assembleur GNU classiques séparées par des sauts à la ligne ou par des `' ; '`.

qualifiers

- ▶ `volatile` : implicite, évite l'optimisation
- ▶ `inline` : indique de considérer une taille minimum d'instructions générées

Problématiques d'adressage pour GCC : quelle est l'adresse de l'instruction suivant un bloc `asm` en sachant qu'il n'est pas encore traduit en assembleur par GCC...

Solution : on prévoit la taille maximale : $\text{ins.} \times \text{max. taille} \times \text{nb.}$

Basic Asm : exemple

Omagah what does that do ?

```
asm volatile (  
    "mov $60, %rax \n\t"  
    "mov $0, %rdi \n\t"  
    "syscall \n\t"  
)
```

Basic Asm : exemple

Omagah what does that do ?

```
asm volatile (  
    "mov $60, %rax \n\t"  
    "mov $0, %rdi \n\t"  
    "syscall \n\t"  
)
```

Un appel système, lequel ??

Code source linux

arch/x86/entry/entry_64.S

arch/x86/entry/syscalls/syscall_64.tbl

Code source GNU libc pour linux

/usr/include/asm/unistd_64.h

/usr/include/asm/unistd_32.h

Basic Asm : exemple

Omagah what does that do ?

```
asm volatile (
    "mov $60, %rax \n\t"
    "mov $0, %rdi \n\t"
    "syscall \n\t"
)
```

Un appel système, lequel ??

Code source linux

```
arch/x86/entry/entry_64.S
```

```
arch/x86/entry/syscalls/syscall_64.tbl
```

Code source GNU libc pour linux

```
/usr/include/asm/unistd_64.h
```

```
/usr/include/asm/unistd_32.h
```

Solution : `exit(0);`

Assembleur en ligne et chaînes constantes en C

Pourquoi met-on des retours à la ligne ?

```
asm volatile (  
    "mov $60, %rax \n\t"  
    "mov $0, %rdi \n\t"  
    "syscall \n\t"  
)
```

Assembleur en ligne et chaînes constantes en C

Pourquoi met-on des retours à la ligne ?

```
asm volatile (  
    "mov $60, %rax \n\t"  
    "mov $0, %rdi \n\t"  
    "syscall \n\t"  
)
```

En c "a" "b" "c" $\rightarrow$ "abc"
Et notre programme ?

Assembleur en ligne et chaînes constantes en C

Pourquoi met-on des retours à la ligne ?

```
asm volatile (
    "mov $60, %rax \n\t"
    "mov $0, %rdi \n\t"
    "syscall \n\t"
)
```

En c "a" "b" "c" → "abc"
Et notre programme ?

Solution : "mov \$60, %rax \r\nmov \$0, %rdi \r\nsyscall \r\n"
Sans les *new line* et les retours charriot as → une seule ligne
d'assembleur → erreur de syntaxe

"utiliser mes variables C dans mon code assembleur ? ? ?"

```
int ma-super-variable-d-interface = 0xcafecafe;  
asm volatile ( "pushd <ma-super-variable-d-interface>" ); ??
```

"utiliser mes variables C dans mon code assembleur ???"

```
int ma-super-variable-d-interface = 0xcafecafe;  
asm volatile ( "pushd <ma-super-variable-d-interface>" ); ??
```

→ Extended Asm!

Extended Asm

Sémantique

- ▶ Lecture / écriture de variables du programme C
- ▶ Sauts à des labels C
- ▶ Met en œuvre des templates de code assembleur
→ transformation de l'assembleur écrit pendant la compilation

Ce cours n'aborde pas la syntaxe `goto`

Syntaxe (grammaire)

```
asm <qualifieurs> ( <template>  
    : <sorties>  
    [ : <entrees>  
    [ : <clobbers> ] ] )
```

qualifieurs

- ▶ `volatile` : désactiver les optimisations

template

Syntaxe générale (grammaire)

Chaîne de caractères constante, template assembleur à modifier :

- ▶ Du texte fixe : % doit être échappé %% (registres)
- ▶ Des opérandes remplacées par le moteur. Commence par % : %0

Sémantique des templates

Interprétée au niveau du compilateur, assemblée par GNU `as`

1. Substitution des symboles listés dans la template par l'équivalent assembleur des expressions C d'entrée et de sortie. %0 réfère à la première des entrées ou des sorties, %1 la deuxième, etc.
2. La chaîne de caractères résultante est ensuite compilée par `as` dans un second temps

Attention : GCC ne compile pas la tamplate générée. Il ne connaît pas sa sémantique

entrees

Syntaxe générale

Zéro ou plusieurs éléments indiquant des variables C qui seront lues

`entrees := <entree>, <entrees> | <entree>`

`entree := [ [<nom>] ] <contrainte> (<expression>)`

symbole

Symbole utilisé dans la template comme nom pour une entrée

contrainte

Chaîne de caractère constante indiquant comment traduire la variable C en assembleur. Indique d'utiliser un registre ("**r**"), indique de passer la référence mémoire ("**m**"), utiliser le registre **a** ("**a**"), etc. Une liste est possible ("**mr**"), la meilleure est choisie par GCC.

expression

Une variable C, son adresse, etc.

entrees

Exemple : utilisation (inutile) des entrées

```
int main() {  
    int toto = 4;  
    int titi = 5;  
    asm volatile (  
        "mov %[toto], %%r8 \n\t"  
        "mov %1, %%r9 \n\t"  
        :: [toto] "a" (toto), "m" (titi)  
    );  
}
```

```
$ gcc -S lst/entrees.c -o /dev/stdout | grep "#APP" -A 9 -B5
```

clobbers

Sémantique

Liste de registres, non déclarés en entrée ou en sortie, modifiés par la template, ici "r8", "r9"

Exemple :

```
int main() {
    int toto = 4;
    int titi = 5;
    asm volatile (
        "mov %[toto], %%r8 \n\t"
        "mov %1, %%r9 \n\t"
        ":: [toto] \"a\" (toto), \"m\" (titi) : \"r8\", \"r9\"
    );
}
```

sorties

Syntaxe générale (grammaire)

Identique aux entrées : `sorties := entrees`

Syntaxe de `symbole` et `expression` identiques.

contrainte

Identique aux entrées. En outre le sens peut être spécifié avant les contraintes : `"<caractères-de-contrainte>"` $\leftrightarrow$ entrée / sortie ;
`"=<caractères-de-contrainte>"` $\leftrightarrow$ sortie.

Exemple : `"+rm"` ou `"=b"`

sorties

Exemple : utilisation (utile) des sorties

```
#include <stdio.h>
```

```
int main() {  
    long int timestamp;  
    asm volatile (  
        "rdtsc \n\t"  
        "mov %%edx, %0 \n\t"  
        "shll $0x20, %0 \n\t"  
        "mov %%eax, %0 \n\t"  
        : "=m"(timestamp) : : "eax", "edx");  
    printf("Timestamp(0x%016lx)\n", timestamp);  
}
```

```
$ gcc -S lst/sorties.c -o /dev/stdout | less
```

Caractères de contraintes d'entrées / sorties

Générales

- ▶ "r" : tout registre général
- ▶ "i" : entier immédiat
- ▶ "g" : "la fête", registres généraux, réf. mémoire, entier immédiat

<https://gcc.gnu.org/onlinedocs/gcc/Simple-Constraints.html#Simple-Constraints>

Intel x86

- ▶ "a"; "b"; "c"; "d" : rax; rbx; rcx; rdx
- ▶ "S"; "D" : rsi; rdi

<https://gcc.gnu.org/onlinedocs/gcc/Machine-Constraints.html#Machine-Constraints>

Modificateurs d'opérandes x86

Syntaxe

À placer entre le % et le numéro ou nom d'opérande **Exemple :**
`asm volatile ("mov %q0, 0" : : "r" (0)) ← q`

Quelques modificateurs

- ▶ "q" : écrit le registre sous forme 64-bit (rax)
- ▶ "h" : registre des huit bits de poids fort d'un mot (ah)

<https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html#x86operandsmodifiers>

Scripts d'édition des liens

https://ftp.gnu.org/old-gnu/Manuals/ld-2.9.1/html_chapter/ld_3.html

2 exemples ?

Hyperviseur mode BIOS / UEFI. *Firmware* Milkimyst