

Architecture des ordinateurs 2

Jean-Luc Scharbarg - ENSEEIHT - Dpt. SN

Janvier 2024

Organisation de l'enseignement

- Volume horaire

- ▶ Cours : 3×1 heure 45
- ▶ TD : 3×1 heure 45
- ▶ TP : 7×1 heure 45
- ▶ Contrôle : 1×1 heure

- Objectif

- ▶ Introduction à l'exécution d'un programme écrit dans un langage de haut niveau sur une architecture matérielle
- ▶ Introduction à l'échange d'informations entre un processeur et son environnement

D'un algorithme à son exécution

- Algorithme à exécuter

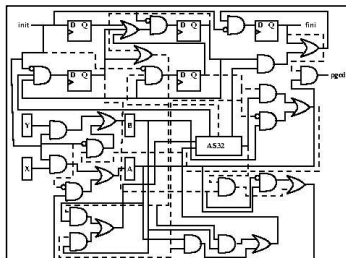
Calcul du PGCD de x et y
<pre> tq x != y faire si x > y alors x ← x-y; sinon y ← y-x; fsi; ftq; pgcd ← x;</pre>

x	y
35	21
14	
	7
7	

- solutions possibles
 - ▶ Un circuit complètement dédié
 - ▶ Un circuit configurable
 - ▶ Un circuit programmable (processeur)

De l'algorithme au circuit complètement dédié

```
1 module pgcd(clk,rst,init,x[31..0],y[31..0]:fini,pgcd[31..0])
2   debut := init on clk reset when rst
3   cmpab := /init*(debut+ab+ba) on clk reset when rst
4   ab := cmpab*asupb on clk reset when rst
5   ba := cmpab*bsupa on clk reset when rst
6   trouve := /init*(cmpab*aeqb+trouve) on clk reset when rst
7   a[31..0] := init*x[31..0]/+init*(ab*s[31..0]/+ab*a[31..0]) on clk reset when rst
8   b[31..0] := init*y[31..0]/+init*(ba*s[31..0]/+ba*b[31..0]) on clk reset when rst
9   e1[31..0] = a[31..0]*(cmpab+ab)+b[31..0]*ba
10  e2[31..0] = a[31..0]*ba+b[31..0]*(cmpab+ab)
11  AS32(e1[31..0],e2[31..0],1,s[31..0],asupb,bsupa,aeqb)
12  fini = trouve
13  pgcd[31..0] = trouve*a[31..0]
14 end module
```



Que comprend ce circuit ?

- Un ensemble de bascules pour mémoriser l'état courant

$\text{debut} := \text{init}$ on clk reset when rst

$\text{cmpab} := \text{/init} * (\text{debut} + \text{ab} + \text{ba})$ on clk reset when rst

$\text{ab} := \text{cmpab} * \text{asupb}$ on clk reset when rst

$\text{ba} := \text{cmpab} * \text{bsupa}$ on clk reset when rst

$\text{trouve} := \text{/init} * (\text{cmpab} * \text{aeqb} + \text{trouve})$ on clk reset when rst

- Des registres pour mémoriser les valeurs courantes des variables de calcul

$\text{a}[31..0] := \text{init} * \text{x}[31..0] + \text{/init} * (\text{ab} * \text{s}[31..0] + \text{/ab} * \text{a}[31..0])$ on clk ...

$\text{b}[31..0] := \text{init} * \text{y}[31..0] + \text{/init} * (\text{ba} * \text{s}[31..0] + \text{/ba} * \text{b}[31..0])$ on clk ...

- De la logique combinatoire pour effectuer les calculs

$\text{e1}[31..0] = \text{a}[31..0] * (\text{cmpab} + \text{ab}) + \text{b}[31..0] * \text{ba}$

$\text{e2}[31..0] = \text{a}[31..0] * \text{ba} + \text{b}[31..0] * (\text{cmpab} + \text{ab})$

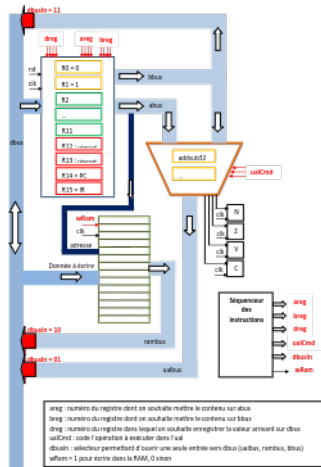
$\text{AS32}(\text{e1}[31..0], \text{e2}[31..0], 1, \text{s}[31..0], \text{asupb}, \text{bsupa}, \text{aeqb})$

De l'algorithme au processeur

```

1 /* TD1 calcul du PGCD */
2
3     set     78, %r1 // valeur de x
4     set     143, %r2 // valeur de y
5
6 pgcd:   cmp     %r2, %r1 // tant que x <= y
7         bne     skip // x=y ?
8         // x = y : return x
9         mov     %r1, %r3 // val <- x
10        ba      stop
11 skip:   bneg    sup // x>y ?
12         // x < y
13         subcc   %r2, %r1, %r2 // y <- y - x
14         ba      pgcd
15 sup:    // x > y
16         subcc   %r1, %r2, %r1 // x <- x - y
17         ba      pgcd
18 stop:   ba      stop // arrêt

```



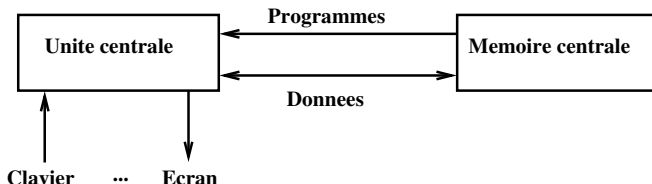
Que comprend un processeur ?

- Un ensemble de bascules pour mémoriser l'état courant du processeur
 - ▶ Où en est-il dans l'exécution de l'instruction courante ?
 - ▶ ...
- Un ensemble de registres pour mémoriser
 - ▶ les valeurs des variables du programme
 - ▶ la position dans l'exécution du programme
 - ▶ ...
- Un ensemble de circuits de calcul
 - ▶ Un additionneur/soustracteur
 - ▶ ...

Un ordinateur, c'est quoi ?

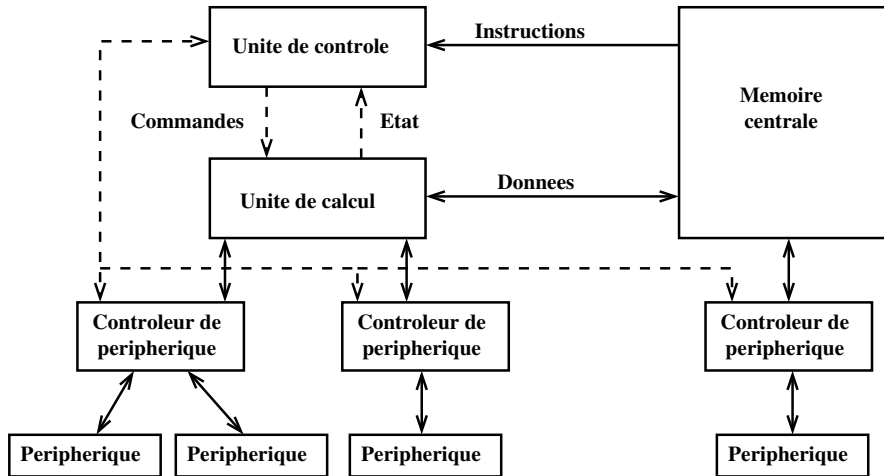
- Un ou plusieurs processeurs qui exécutent des programmes,
- Des moyens pour envoyer des ordres (clavier, souris, ...),
- Des moyens pour récupérer des résultats (écran, ...),
- Des moyens pour stocker de l'information (mémoire, disques, ...)
- Des moyens pour dialoguer avec d'autres dispositifs (interface réseau, ports, ...)

Organisation générale d'un ordinateur



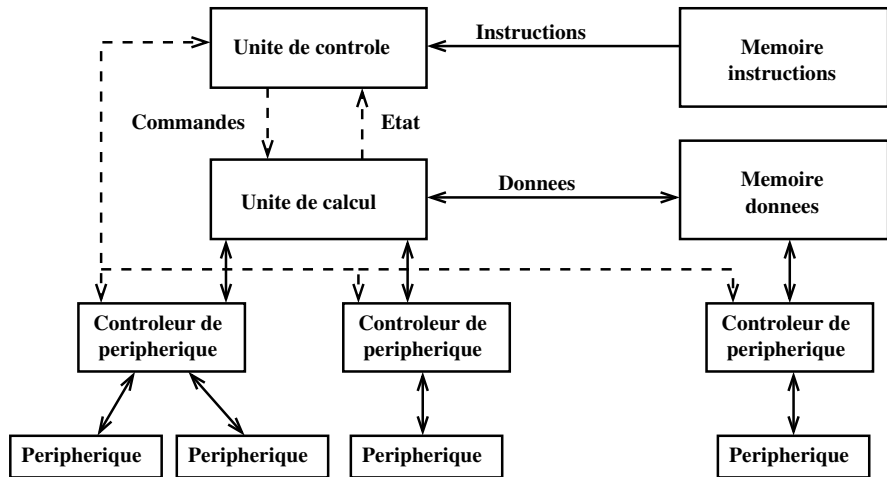
- Exécution de programmes par l'unité centrale
 - ▶ Lecture et écriture de données en mémoire centrale
 - ▶ Interactions avec l'extérieur
- Programmes et données stockés et véhiculés logiquement sous la forme de chiffres binaires ou bits (binary digits), physiquement sous la forme de signaux électroniques
- Développement d'un programme
 - ▶ Ecriture du programme dans un langage de "haut niveau" (e.g. Ada, C, ...)
 - ▶ Traduction du programme en langage machine (instructions plus rudimentaires)
 - ▶ Exécution du programme en langage machine

Modèle de Von Neuman



- Une mémoire commune aux données et aux programmes

Modèle de Harvard



- Deux mémoires séparées pour les données et les programmes

Principe général de fonctionnement d'un processeur

- Unité centrale du processeur

- ▶ Comprend l'unité de contrôle et l'unité de calcul
- ▶ S'occupe de l'interprétation et de l'exécution des programmes
 - ★ Unité de calcul = Unité Arithmétique et Logique + zone de stockage de données temporaires (registres)
 - ★ Unité de contrôle = envoi des ordres à l'unité de calcul pour l'exécution des instructions du programme \Rightarrow interprétation de chaque instruction

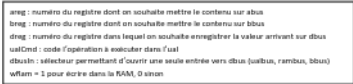
- Mémoire centrale

- ▶ Modèle de Von Neuman : les instructions et les données sont dans la même mémoire \Rightarrow architecture plus simple, pas d'accès en parallèle
- ▶ Modèle de Harvard : les instructions et les données sont dans deux mémoires distinctes (intérêt : accès simultané aux deux mémoires)

- Accès aux périphériques via des contrôleurs de périphériques

- ▶ Connexion physique du périphérique
- ▶ Synchronisation du périphérique avec l'unité centrale par un protocole assurant que toute donnée échangée est reçue une et une seule fois

L'unité centrale



Principe général de l'exécution d'une instruction

- Adresse mémoire de l'instruction courante dans le Compteur Ordinal (registre PC : Program Counter)
- Code de l'instruction courante dans le registre instructions (registre IR : Instruction Register)
- Etapes pour l'exécution d'une instruction
 - ▶ Chargement du code de l'instruction courante dans le registre IR

$$IR \leftarrow Memoire[PC]$$

- ▶ Décodage de l'instruction
- ▶ Lecture éventuelle des opérandes
- ▶ Exécution de l'opération
- ▶ Sauvegarde éventuelle du résultat

Petit exemple introductif

- Mise en œuvre d'un processeur simpliste

- ▶ Une mémoire de 16 mots de 4 bits
- ▶ Un registre de données *ACC* sur 4 bits
- ▶ Un jeu d'instructions très réduit

clr		Mise à 0 du registre <i>ACC</i>
ld	#v	Chargement de la valeur <i>v</i> dans <i>ACC</i>
st	a	Copie du contenu de <i>ACC</i> en mémoire à l'adresse <i>a</i>
add	a	$ACC \leftarrow ACC + \text{contenu du mot mémoire d'adresse } a$
jmp	a	<i>a</i> est l'adresse mémoire de l'instruction suivante

- Exemple de programme

	ld	#3	$ACC \leftarrow 3$
	st	8	$MEM[8] \leftarrow ACC$
	add	8	$ACC \leftarrow ACC + MEM[8]$
etiq:	jmp	etiq	Boucle infinie

- Codage des instructions

clr		0000						add	a	0101 $a_3 a_2 a_1 a_0$
ld	#v	0010 $v_3 v_2 v_1 v_0$						jmp	a	0100 $a_3 a_2 a_1 a_0$
st	a	0011 $a_3 a_2 a_1 a_0$								

Exercice 1

- Donner le codage du programme suivant, en supposant que la première instruction se trouve à l'adresse 0 de la mémoire.

ld	#3		$ACC \leftarrow 3$
st	8		$MEM[8] \leftarrow ACC$
add	8		$ACC \leftarrow ACC + MEM[8]$
etiq:	jmp	etiq	Boucle infinie

Petit exemple introductif : interprétation des instructions

PC \leftarrow 0; /* Le programme démarre à l'adresse 0 */

TantQue vrai **Faire**

Selon MEM[PC]

 0000₂ : /* Instruction *clr* */

 ACC \leftarrow 0;

 PC \leftarrow PC + 1;

 0010₂ : /* Instruction *ld* */

 ACC \leftarrow MEM[PC+1];

 PC \leftarrow PC + 2;

 0011₂ : /* Instruction *st* */

 MEM[MEM[PC+1]] \leftarrow ACC;

 PC \leftarrow PC + 2;

 0101₂ : /* Instruction *add* */

 ACC \leftarrow ACC + MEM[PC+1];

 PC \leftarrow PC + 2;

 0100₂ : /* Instruction *jmp* */

 PC \leftarrow MEM[PC+1];

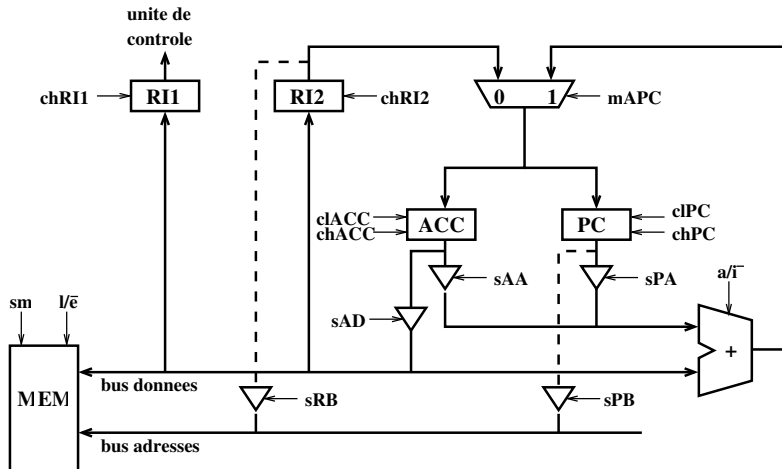
FinSelon;

FinTantQue;

Petit exemple introductif : unité de calcul

- Principe général de fonctionnement
 - ▶ Adresse de l'instruction courante dans un registre *PC*
 - ▶ Code de l'instruction courante (4 ou 8 bits) dans un ou deux registres *R/1*, *R/2*
 - ▶ L'unité de calcul charge le code de l'instruction courante, puis exécute l'opération correspondante
- Ressources de l'unité de calcul
 - ▶ Un registre *PC* sur 4 bits
 - ▶ Deux registres *R/1* et *R/2* chacun sur 4 bits
 - ▶ Un registre *ACC* sur 4 bits
 - ▶ Une unité arithmétique capable de faire des additions et des incrémentations
- Mécanisme pour l'échange d'informations avec la mémoire
 - ▶ Choix de l'opération à effectuer \Rightarrow deux signaux *sm* et I/\bar{e}
 - ★ *sm* = 0 : pas d'opération
 - ★ *sm* = 1 et I/\bar{e} = 0 : écriture en mémoire
 - ★ *sm* = 1 et I/\bar{e} = 1 : lecture de la mémoire
 - ▶ Donnée lue ou écrite \Rightarrow un bus bidirectionnel
 - ▶ Adresse de lecture ou d'écriture \Rightarrow un bus monodirectionnel

Petit exemple introductif : unité de calcul



Petit exemple introductif : traitement des instructions

● Instruction *clr*

- ▶ Chargement du code de l'instruction dans le registre d'instruction
RI1 \leftarrow **MEM[PC]** : $sPB = 1$, $l/\bar{e} = 1$, $sm = 1$, $chRI1 = 1$
- ▶ Passage au mot suivant du programme
PC \leftarrow **PC** + 1 : $sPA = 1$, $mAPC = 1$, $chPC = 1$
- ▶ Mise à 0 du registre *ACC*
ACC \leftarrow 0 : $clACC = 1$

● Instruction *ld v*

- ▶ Chargement du code de l'instruction dans le registre d'instruction
RI1 \leftarrow **MEM[PC]** : $sPB = 1$, $l/\bar{e} = 1$, $sm = 1$, $chRI1 = 1$
- ▶ Passage au mot suivant du programme
PC \leftarrow **PC** + 1 : $sPA = 1$, $mAPC = 1$, $chPC = 1$
- ▶ Chargement de *v* dans le registre d'instruction
RI2 \leftarrow **MEM[PC]** : $sPB = 1$, $l/\bar{e} = 1$, $sm = 1$, $chRI2 = 1$
- ▶ Passage au mot suivant du programme
PC \leftarrow **PC** + 1 : $sPA = 1$, $mAPC = 1$, $chPC = 1$
- ▶ Copie de *v* dans le registre *ACC*
ACC \leftarrow **RI2** : $chACC = 1$

Petit exemple introductif : traitement des instructions

● Instruction *st a*

- ▶ Chargement du code de l'instruction dans le registre d'instruction
 $RI1 \leftarrow MEM[PC] : sPB = 1, I/\bar{e} = 1, sm = 1, chRI1 = 1$
- ▶ Passage au mot suivant du programme
 $PC \leftarrow PC + 1 : sPA = 1, mAPC = 1, chPC = 1$
- ▶ Chargement de *v* dans le registre d'instruction
 $RI2 \leftarrow MEM[PC] : sPB = 1, I/\bar{e} = 1, sm = 1, chRI2 = 1$
- ▶ Passage au mot suivant du programme
 $PC \leftarrow PC + 1 : sPA = 1, mAPC = 1, chPC = 1$
- ▶ Copie du contenu de *ACC* en mémoire à l'adresse *a*
 $MEM[RI2] \leftarrow ACC : sRB = 1, sAD = 1, sm = 1$

● Instruction *add a* (début)

- ▶ Chargement du code de l'instruction dans le registre d'instruction
 $RI1 \leftarrow MEM[PC] : sPB = 1, I/\bar{e} = 1, sm = 1, chRI1 = 1$
- ▶ Passage au mot suivant du programme
 $PC \leftarrow PC + 1 : sPA = 1, mAPC = 1, chPC = 1$
- ▶ Chargement de *v* dans le registre d'instruction
 $RI2 \leftarrow MEM[PC] : sPB = 1, I/\bar{e} = 1, sm = 1, chRI2 = 1$

Petit exemple introductif : traitement des instructions

- Instruction *add a* (fin)

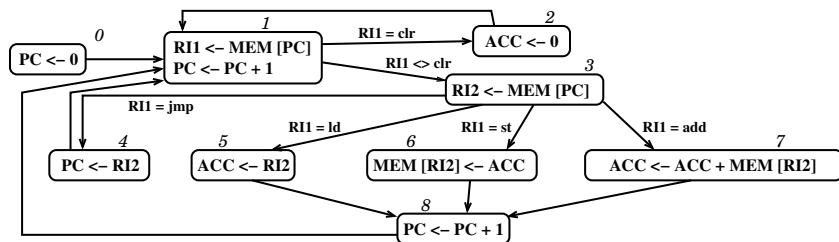
- ▶ Passage au mot suivant du programme
 $PC \leftarrow PC + 1 : sPA = 1, mAPC = 1, chPC = 1$
- ▶ Ajout de $MEM[a]$ au contenu de ACC
 $ACC \leftarrow ACC + MEM[RI2] : sAA = 1, sRB = 1, l/\bar{e} = 1, sm = 1, a/\bar{i} = 1, mAPC = 1, chACC = 1$

- Instruction *jmp a*

- ▶ Chargement du code de l'instruction dans le registre d'instruction
 $RI1 \leftarrow MEM[PC] : sPB = 1, l/\bar{e} = 1, sm = 1, chRI1 = 1$
- ▶ Passage au mot suivant du programme
 $PC \leftarrow PC + 1 : sPA = 1, mAPC = 1, chPC = 1$
- ▶ Chargement de v dans le registre d'instruction
 $RI2 \leftarrow MEM[PC] : sPB = 1, l/\bar{e} = 1, sm = 1, chRI2 = 1$
- ▶ Copie de a dans le registre PC
 $PC \leftarrow RI2 : chPC = 1$

Petit exemple introductif : unité de commande

- Initialisation de PC à 0, puis répétition à l'infini du traitement d'une instruction



- Construction du circuit correspondant en utilisant les mêmes principes que pour le circuit de calcul du PGCD de deux entiers
- Un circuit possible
 - Une bascule D pour chaque état de l'unité de commande
 - Génération des commandes en fonction de l'état actif
 - ★ Exemple : $chACC = 1$ pour les états 5 et 7

Exercice 2

- Dérouler l'exécution du programme suivant (Etats du graphe, contenu des registres)

ld	#3		$ACC \leftarrow 3$
st	8		$MEM[8] \leftarrow ACC$
add	8		$ACC \leftarrow ACC + MEM[8]$
etiq:	jmp	etiq	Boucle infinie

Caractéristiques principales du CRAPS

- Version simplifiée du SPARC version 8
- Processeur 32 bits
- Mémoire de 2^{32} octets
- Architecture de type *big-endian*
- Chaque instruction est codée sur un mot de 32 bits
- Processeur RISC (Reduced Instruction Set Computer) : jeu d'instructions réduit
- Machine de type *load/store* : un petit nombre d'instructions spécialisées pour le transfert des données entre la mémoire centrale et les registres
- 16 registres généraux, un compteur ordinal, un registre instruction, un registre d'états, ...
- **On va considérer un mini-CRAPS, version simplifiée du CRAPS**

Langages de programmation des processeurs

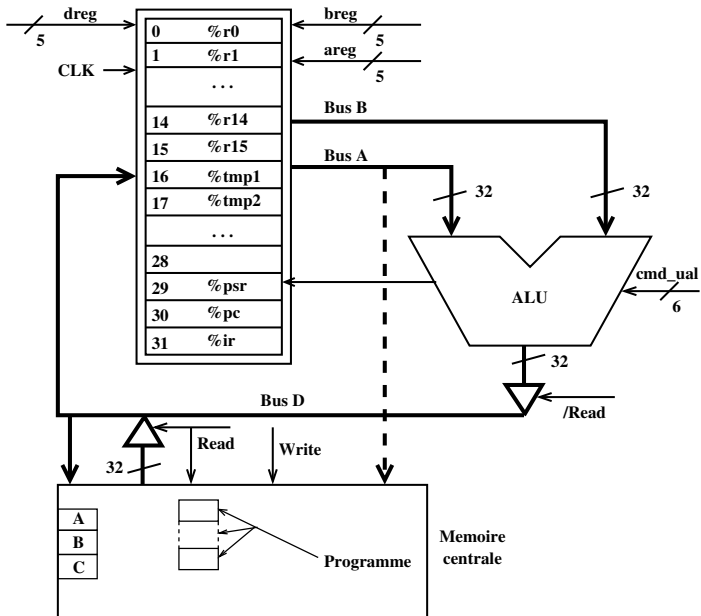
- Le processeur ne comprend que le binaire \Rightarrow toute instruction s'exprime sous la forme d'une séquence de 0 et de 1
- L'écriture d'un programme en binaire est difficile (peu lisible \Rightarrow risque élevé d'erreur)
- Association d'un nom à chaque instruction pour faciliter le travail
- Une instruction du processeur est une opération élémentaire de l'unité de calcul \Rightarrow un programme nécessite de très nombreuses instructions et manque souvent de structure
- Utilisation de langages structurés de plus haut niveau (une instruction du langage correspond à une séquence d'instructions du processeur) : Java, Pascal, C, ...
- Compilateurs pour traduire les programmes en langage de haut niveau en programmes en langage du processeur
- Un compilateur différent pour chaque processeur

Exemple de programme

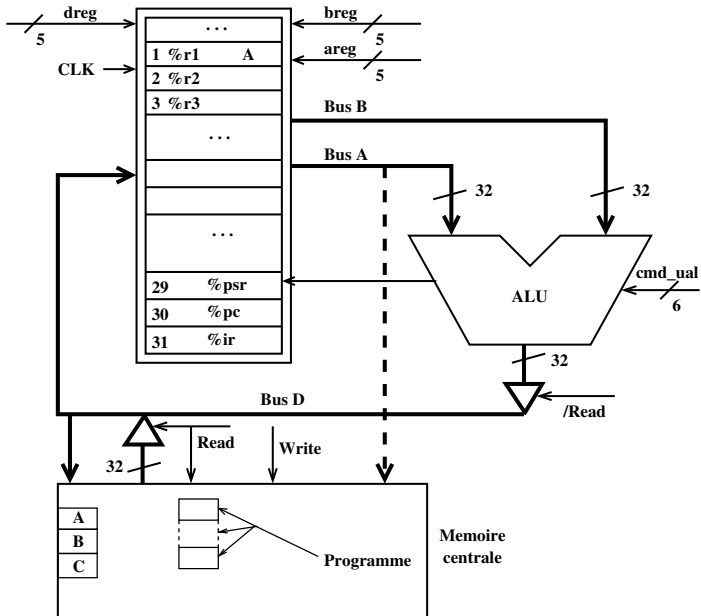
```
if (A <= B)
    C = A + B;
else
    C = A - B;
```

- Addition/soustraction de deux variables avec résultat dans une troisième variable
- Comparaison avec saut éventuel
- Saut inconditionnel

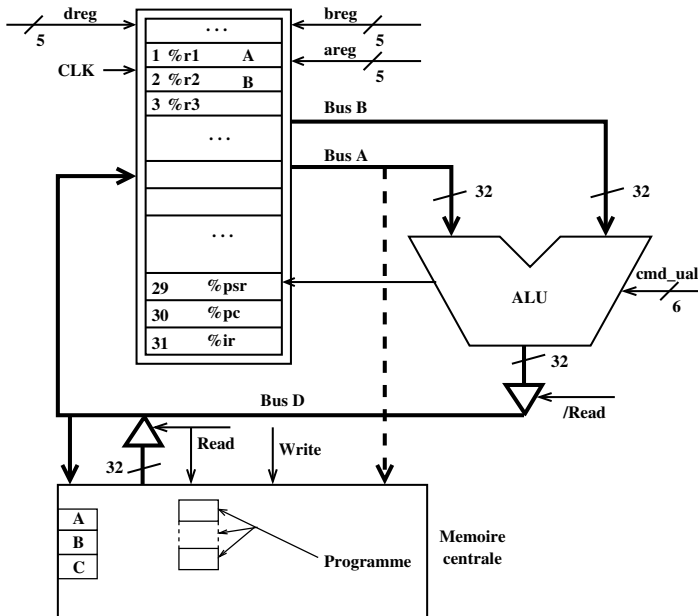
Point de départ : A et B en mémoire



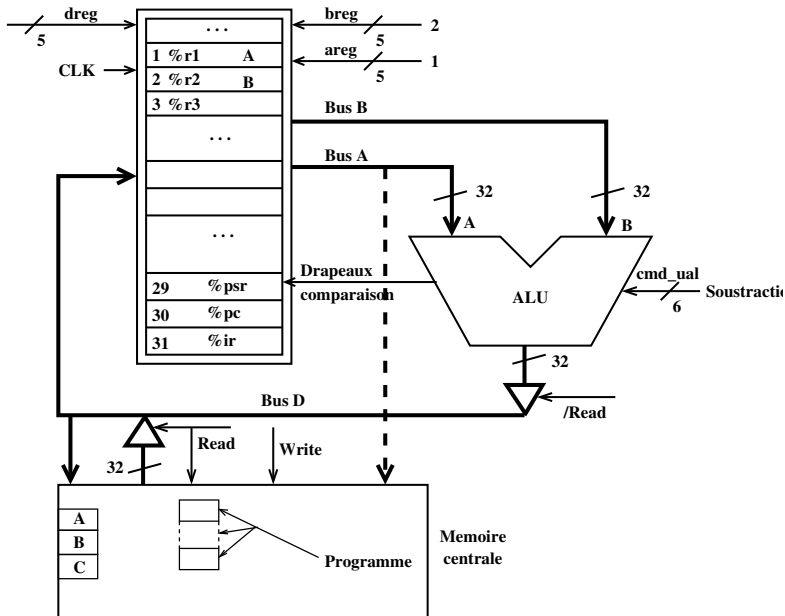
Chargement de A dans un registre



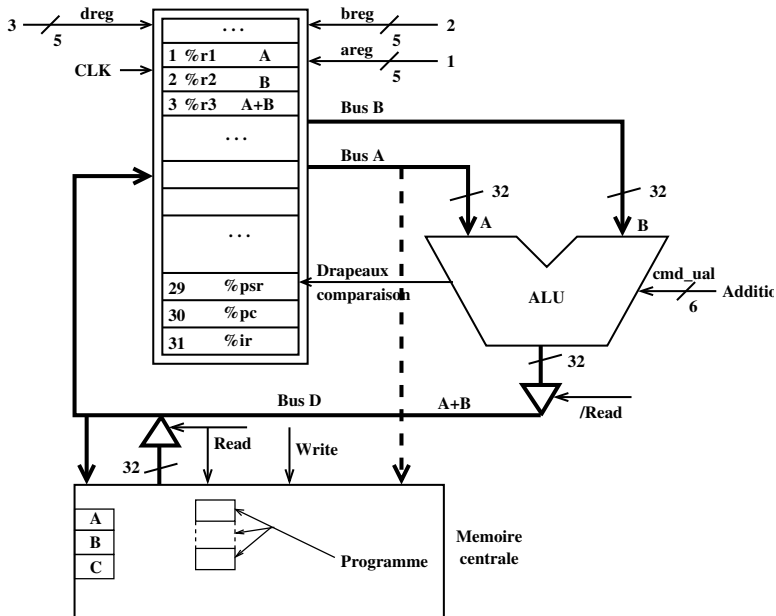
Chargement de B dans un registre



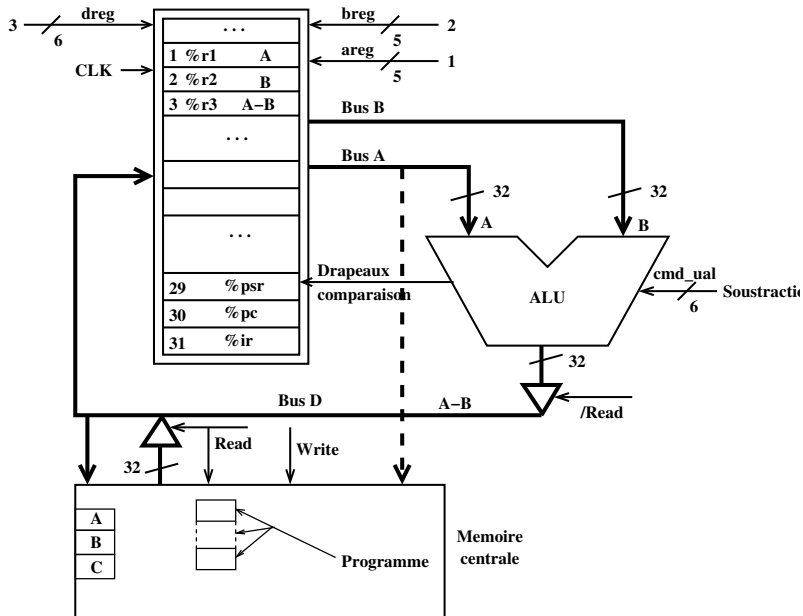
Comparaison de A et de B



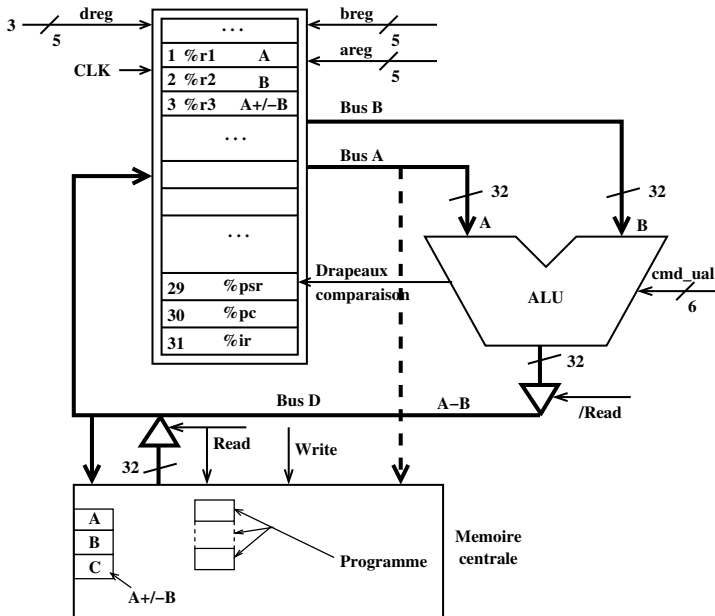
Calcul de $A + B$



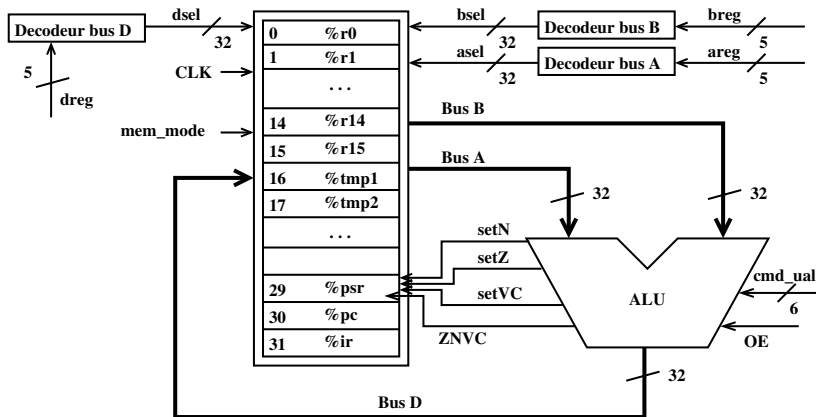
Calcul de $A - B$



Sauvegarde du résultat en mémoire

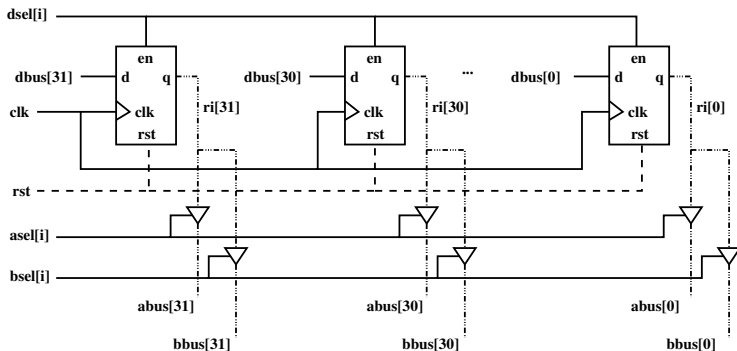


Architecture pour l'addition



- 16 registres utilisateurs (%r0 à %r15), 1 registre d'état (%psr)
- 3 bus pour les échanges de données entre l'UAL et les registres

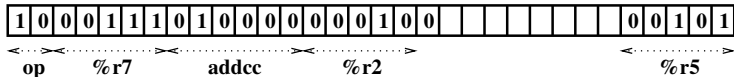
Le composant de mémorisation : le registre %ri



- Le registre contient la valeur $\%ri[31] \dots \%ri[0]$
- A chaque front montant de l'horloge clk , la valeur sur le bus D est chargée dans le registre ssi $dse[i] = 1$
- rst permet la remise à 0 du registre, indépendamment de l'horloge
- La valeur contenue dans le registre est placée sur le bus A et/ou le bus B en fonction des valeurs de $asel[i]$ et $bsel[i]$

Déroulement de l'instruction *addcc %r2 %r5 %r7*

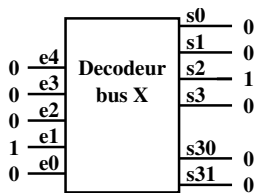
- Code de l'instruction



- 1 Mise en place des entrées pour l'opération d'addition
 - ▶ Le contenu du registre *%r2* est placé sur le bus *A* (**asel[2] = 1**)
 - ▶ Le contenu du registre *%r5* est placé sur le bus *B* (**bssel[5] = 1**)
 - ▶ La commande d'addition est envoyée à l'unité arithmétique et logique (**cmd_alu = 010000**)
- 2 Opération d'addition
 - ▶ L'unité arithmétique et logique effectue l'addition des deux valeurs et place le résultat sur le bus *D*
 - ▶ L'unité arithmétique et logique génère les indicateurs ZNVC
- 3 Sauvegarde du résultat
 - ▶ Le résultat de l'addition est mémorisé dans *%r7* (**dsel[7] = 1**)
 - ▶ Les valeurs de ZNVC sont mémorisées dans le registre *%psr* (**setN = 1, setZ = 1, setVC = 1**)

Principe des décodeurs associés aux bus

- Activation de la ligne de sortie s_i avec $i = e_5 e_4 e_3 e_2 e_1 e_0$



$$s_0 = \overline{e_4} \cdot \overline{e_3} \cdot \overline{e_2} \cdot \overline{e_1} \cdot \overline{e_0}$$

$$s_1 = \overline{e_4} \cdot \overline{e_3} \cdot \overline{e_2} \cdot \overline{e_1} \cdot e_0$$

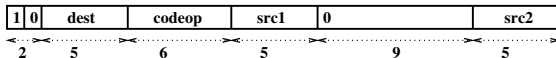
$$s_2 = \overline{e_4} \cdot \overline{e_3} \cdot \overline{e_2} \cdot e_1 \cdot \overline{e_0}$$

...

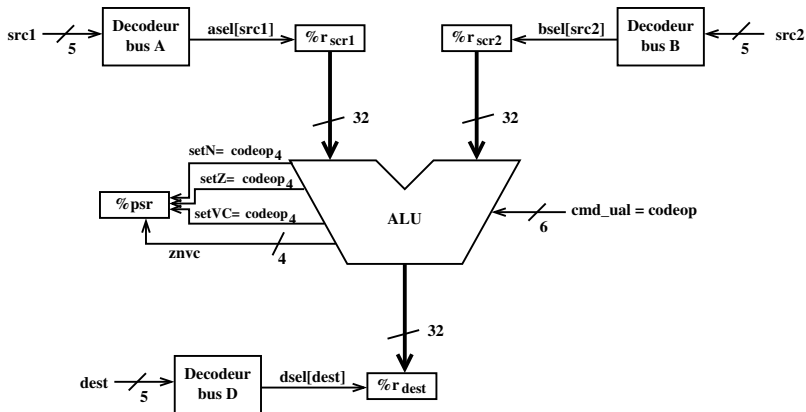
$$s_{31} = e_4 \cdot e_3 \cdot e_2 \cdot e_1 \cdot e_0$$

Synthèse d'une instruction d'addition ou de soustraction

- Code de l'instruction



- Vue d'ensemble du déroulement de l'instruction



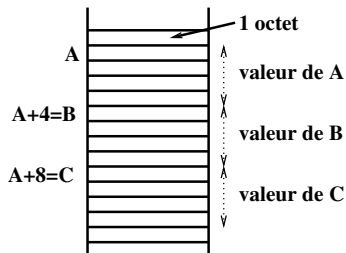
Exercice 3

- Donner les valeurs des champs de l'instruction

orcc %r2,%r4,%r6

Opération d'addition avec opérandes et résultat en mémoire

- Opération à exécuter : $C \leftarrow A + B$



Une adresse memoire par octet

Valeurs rangees dans l'ordre big endian
(poids forts d'abord)

Memoire centrale

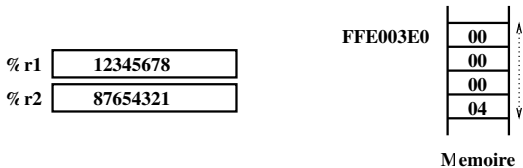
- Pas d'instructions arithmétiques ou logiques avec des opérandes situés en mémoire \Rightarrow il faut passer par des registres
- Trois étapes pour l'opération d'addition
 - Chargement des opérandes A et B dans des registres
 - Addition du contenu des deux registres avec résultat dans un registre
 - Stockage du résultat en mémoire

Copie d'une valeur de la mémoire vers un registre

- Informations nécessaires pour l'exécution de l'opération de copie
 - ▶ Code stipulant qu'il s'agit d'une opération de copie de la mémoire vers un registre
 - ★ Le code opération d'une instruction CRAPS est sur 6 bits
 - ▶ Adresse mémoire concernée
 - ★ Espace d'adressage du CRAPS : 2^{32} octets \Rightarrow une adresse mémoire est sur 32 bits
 - ▶ Numéro du registre concerné
 - ★ 32 registres \Rightarrow numéro de registre sur 5 bits
- 43 bits pour coder la copie d'une valeur de la mémoire vers un registre
- Processeur RISC \Rightarrow toute instruction est codée sur 32 bits
- Il n'est pas possible de coder en une instruction la copie d'une valeur de la mémoire vers un registre
- Solution adoptée par le CRAPS : 3 instructions
 - ▶ Chargement, en deux étapes, de l'adresse mémoire concernée dans un registre,
 - ▶ Utilisation d'un adressage indirect : chargement de la donnée dont l'adresse mémoire se trouve dans le registre

Illustration de la séquence d'instructions mise en œuvre

- Exemple considéré : copie de la valeur située à l'adresse mémoire $FFE003E0_{16}$ dans le registre $\%r2$
- Point de départ



- Copie des poids forts de l'adresse dans $\%r1$

```
sethi 0b111111111111000000000000,%r1
```

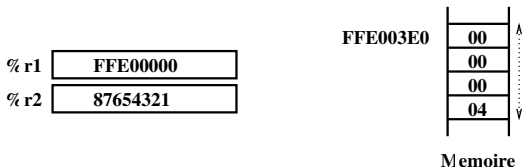
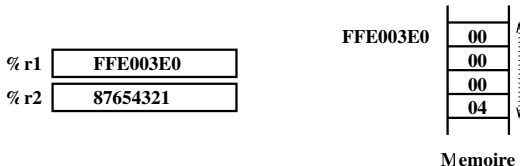


Illustration de la séquence d'instructions mise en œuvre

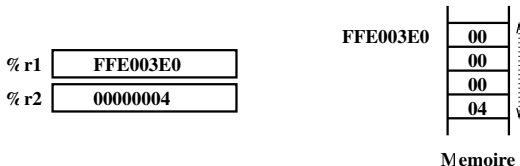
- Copie des poids faibles de l'adresse dans %r1

```
orcc %r1, 0b1111100000,%r1
```



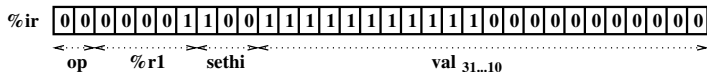
- Chargement de la donnée dont l'adresse mémoire est dans %r1 dans le registre %r2

```
ld [%r1],%r2
```



Exécution de l'instruction *sethi val_{31...10}, %r1*

- Codage de l'instruction



- Etapes de l'exécution de l'instruction

- ▶ Décalage à gauche de 2 positions de *%ir*, résultat dans *%r1*

- ★ *%ir* sur le bus A (**asel[31] = 1**)
- ★ *%r21* (constante 2) sur le bus B (**bsel[21] = 1**)
- ★ Décalage à gauche (**cmd_alu 110101**)
- ★ Résultat dans *%r1* (**dsel[1] = 1**)

%r1

0	0	0	0	0	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- ▶ Décalage à gauche de 8 positions de *%r1*, résultat dans *%r1*

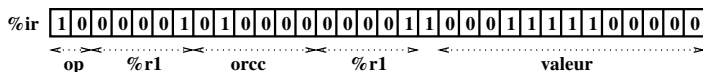
- ★ *%r1* sur le bus A (**asel[1] = 1**)
- ★ *%r23* (constante 8) sur le bus B (**bsel[23] = 1**)
- ★ Décalage à gauche (**cmd_alu 110101**)
- ★ Résultat dans *%r1* (**dsel[1] = 1**)

%r1

1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Exécution de l'instruction *orcc %r1, val9...0, %r1*

- Codage de l'instruction



- Etapes de l'exécution de l'instruction

- ▶ Extension des 13 bits de poids faibles de *%ir* sur 32 bits, résultat dans *%tmp1*

- ★ *%ir* sur le bus A (**asel[31] = 1**)
- ★ Extension de 13 bits à 32 bits (**signext13, cmd_alu 100000**)
- ★ Résultat dans *%tmp1* (**dsel[16] = 1**)

%tmp1

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- ▶ Addition entre *%r1* et *%tmp1*

- ★ *%r1* sur le bus A (**asel[1] = 1**)
- ★ *%tmp1* sur le bus B (**bsel[16] = 1**)
- ★ Addition (**cmd_alu 010000**)
- ★ Résultat dans *%r1* (**dsel[1] = 1**)

%r1

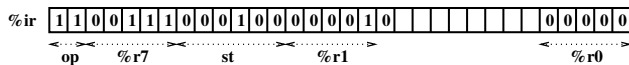
1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Copie d'une valeur d'un registre vers la mémoire

- Séquence d'instructions mise en œuvre

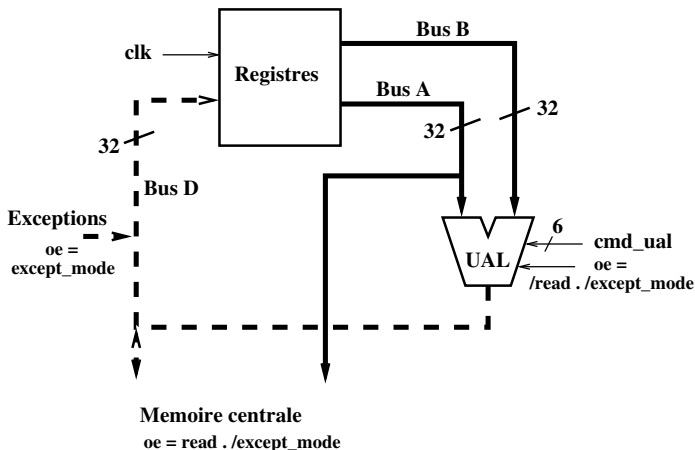
```
sethi    val_31...10, %r1
orcc     %r1, val_9...0, %r1
st       %r7, [%r1]
```

- Codage de l'instruction `st %r7, [%r1]`



- Etapes de l'exécution de l'instruction
 - ▶ Calcul de l'adresse mémoire $\%r1 + \%r0$ dans *%tmp1*
 - ★ Identique au calcul mis en œuvre pour l'instruction *ld*
 - ▶ Ecriture en mémoire à l'adresse contenue dans *%tmp1*, la donnée à écrire est dans *%r7*
 - ★ *%tmp1* sur le bus *A* (**asel[16] = 1**)
 - ★ *%r7* sur le bus *B* (**bsel[7] = 1**)
 - ★ Copie du bus *B* sur le bus *D* (**cmd_alu 101000**)
 - ★ Ordre d'écriture envoyé à la mémoire, La valeur présente sur le bus *D* est écrite en mémoire à l'adresse présente sur le bus *A*

Architecture d'ensemble intégrant la mémoire



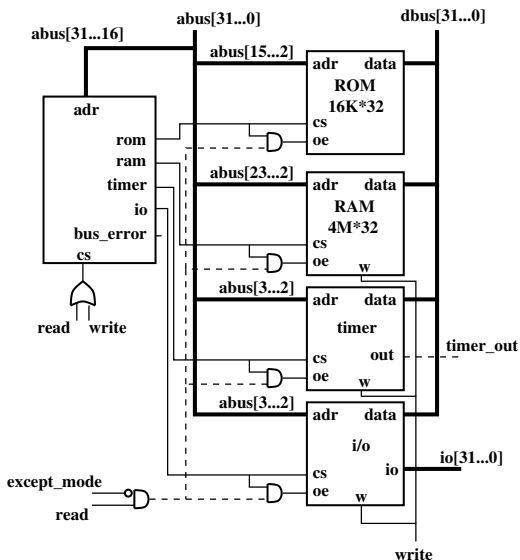
- Ecriture sur le bus D via des portes 3-états
 - ▶ UAL : $OE = \overline{read} \cdot \overline{except_mode}$
 - ▶ Mémoire : $OE = read \cdot \overline{except_mode}$
 - ▶ Exception : $OE = except_mode$

Cartographie de la mémoire

0000 0000 ... 0000 FFFF	ROM (64K)
0040 0000 ... 0040 000B	Timer
0060 0000 ... 0060 000F	Entrees/sorties
0100 0000 ... 01FF FFFF	RAM (16 MO)

- ROM : Read Only Memory, mémoire en lecture seule
- RAM : Random Access Memory, mémoire en lecture/écriture
- Timer : pour la configuration, la mise en route et l'arrêt du timer
- Les entrées/sorties sont vues comme des adresses mémoire
- Accès à des adresses mémoire en dehors des plages définies ⇒ exception *bus_error*

Structure du sous-système mémoire



Instructions de branchement pour les ruptures de séquence

- Structures de contrôle nécessitant des ruptures de séquence

- ▶ Instructions conditionnelles

Si condition alors traitement1		Si condition allera sinon traitement1 allera fin
Sinon traitement2	sinon :	traitement2
Finsi	finsi :	

- ▶ Itérations

Tantque Condition faire traitement	tq:	Si non condition allera fintq traitement allera tq
Fintantque	fintq :	

- Deux types de ruptures de séquences

- ▶ Les branchements inconditionnels : **allera** *eti*q
- ▶ Les branchements conditionnels : **Si** *condition* **allera** *eti*q

Exemple de programme

```

si A > B alors
  C ← A + B;
sinon
  C ← A - B;
finssi;
  
```



```

si A <= B allera sinon
  C ← A + B;
allera finssi
sinon: C ← A - B;
finssi:
  
```

Chargement
adresses
memoire
A et B

Memoire		@
...		
sethi A 31..10 , %r1		x
orcc %r1, A 9..0 , %r1		x+4
ld [%r1], %r2		x+8
sethi B 31..10 , %r3		x+12
orcc %r3, B 9..0 , %r3		x+16
ld [%r3], %r4		x+20
		x+24
		x+28
add %r2, %r4, %r5		x+32
sethi C 31..10 , %r6		x+36
orcc %r6, C 9..0 , %r6		x+40
st %r5, [%r6]		x+44
		x+48
sub %r2, %r4, %r5		x+52=sinon
sethi C 31..10 , %r6		x+56
orcc %r6, C 9..0 , %r6		x+60
st %r5, [%r6]		x+64
		x+68=finssi
...		

Principes des instructions de branchement

- L'instruction de branchement inconditionnel (*ba etiq*) force la mise à jour du registre *%pc*
 - ▶ remplace l'incrémentation de *%pc* par l'affectation de *%pc* avec l'adresse de l'étiquette
- L'effet de l'instruction de branchement conditionnel (*ble etiq*) dépend de la valeur de la condition
 - ▶ Condition vraie \Rightarrow affectation de *%pc* avec l'adresse de l'étiquette
 - ▶ Condition fausse \Rightarrow incrémentation de *%pc* (pas de rupture de séquence)
- La condition associée au branchement conditionnel porte sur les indicateurs ZNVC
- Mise en œuvre de l'opération : **si** $A \leq B$ **allera sinon**
 - ▶ Calcul de $A - B$ pour obtenir les indicateurs ZNVC
 - ★ *subcc %r2, %r4, %r0*
 - ▶ Branchement à sinon si résultat négatif ou nul

$Z \text{ or } (N \text{ xor } V)$

★ *ble sinon*

Codage de l'adresse de l'étiquette associée au branchement

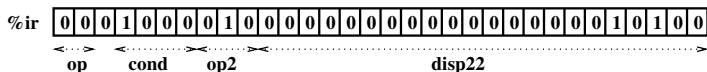
Memoire	@
...	
ba finsi	x+48
sub %r2, %r4, %r5	x+52=sinon
sethi C 31..10 , %r6	x+56
orcc %r6, C 9..0 , %r6	x+60
st %r5, [%r6]	x+64
	x+68=finisi
...	

adresse cible : $x + 68$
deplacement : $+ 20$

- Branchement absolu : c'est l'adresse absolue de l'instruction cible du branchement qui est codée dans l'instruction (exemple : $x + 68$)
 - ▶ Adresse de l'étiquette sur 32 bits
 - ▶ Ne peut pas être codée dans une instruction CRAPS
- Branchement relatif : c'est le déplacement jusqu'à l'instruction cible du branchement qui est codé dans l'instruction (exemple : $+20$)
 - ▶ Déplacement limité par le nombre de bits utilisé pour le codage
 - ▶ C'est la solution utilisée par le CRAPS

Exécution de l'instruction *ba fin*

- Codage de l'instruction



- Etapes de l'exécution de l'instruction

- ▶ Extension des 22 bits de *%ir* sur 32 bits, résultat dans *%tmp1*
 - ★ *%ir* sur le bus A (**asel[31] = 1**)
 - ★ Extension de 22 bits à 32 bits (signext22, **cmd_alu = 100001**)
 - ★ Résultat dans *%tmp1* (**dsel[16] = 1**)

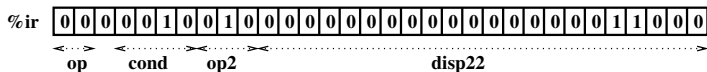
%tmp1

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- ▶ Addition entre *%tmp1* et *%pc*, résultat dans *%pc*
 - ★ *%tmp1* sur le bus A (**asel[16] = 1**)
 - ★ *%pc* sur le bus B (**bse[30] = 1**)
 - ★ Addition (**cmd_al = 000000**)
 - ★ Résultat dans *%pc* (**dse[30] = 1**)

Exécution de l'instruction *ble sinon*

- Codage de l'instruction



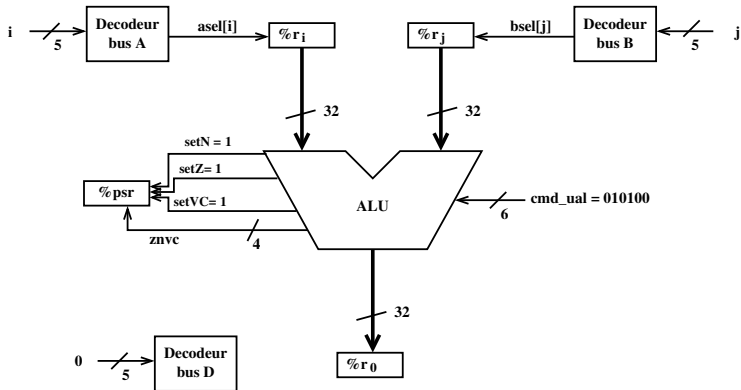
- Instruction sans effet lorsque la condition est fausse
- Etapes de l'exécution de l'instruction lorsque la condition est vraie

- Extension des 22 bits de *%ir* sur 32 bits, résultat dans *%tmp1*

- ★ *%ir* sur le bus A (**asel[31] = 1**)
- ★ Extension de 22 bits à 32 bits (**signext22, cmd_alu = 100001**)
- ★ Résultat dans *%tmp1* (**dsel[16] = 1**)

Exécution d'une opération de comparaison

- Comparer les valeurs entières contenues dans deux registres $\%ri$ et $\%rj$
 - ▶ Calcul de $\%ri - \%rj$ avec mise à jour des drapeaux
 - ▶ On ne garde pas le résultat \Rightarrow on l'envoie dans $\%r0$
- Instruction à exécuter : `subcc %ri, %rj, %r0`
- Le registre $\%r0$ contient toujours 0 \Rightarrow écriture sans effet



Le programme complet

```

si A > B alors
  C ← A + B;
sinon
  C ← A - B;
finsi;
    
```



```

si A <= B allera sinon
  C ← A + B;
allera finsi
sinon: C ← A - B;
finsi:
    
```

Chargement
adresses
memoire
A et B

Memoire		@
...		
sethi A 31..10 , %r1		x
orcc %r1, A 9..0 , %r1		x+4
ld [%r1], %r2		x+8
sethi B 31..10 , %r3		x+12
orcc %r3, B 9..0 , %r3		x+16
ld [%r3], %r4		x+20
subcc %r2, %r4, %r0		x+24
ble sinon		x+28
add %r2, %r4, %r5		x+32
sethi C 31..10 , %r6		x+36
orcc %r6, C 9..0 , %r6		x+40
st %r5, [%r6]		x+44
ba finsi		x+48
sub %r2, %r4, %r5		x+52=sinon
sethi C 31..10 , %r6		x+56
orcc %r6, C 9..0 , %r6		x+60
st %r5, [%r6]		x+64
		x+68=finsi
...		

Instructions du processeur CRAPS

- Instructions arithmétiques et logiques

Instruction		Effet	
add	%ri, reg/cst, %rj	addition	$\%rj \leftarrow \%ri + \text{reg/cst}$
addcc	%ri, reg/cst, %rj	addition	$\%rj \leftarrow \%ri + \text{reg/cst}$
sub	%ri, reg/cst, %rj	soustraction	$\%rj \leftarrow \%ri - \text{reg/cst}$
subcc	%ri, reg/cst, %rj	soustraction	$\%rj \leftarrow \%ri - \text{reg/cst}$
umulcc	%ri, reg/cst, %rj	multiplication	$\%rj \leftarrow \%ri \times \text{reg/cst}$
udivcc	%ri, reg/cst, %rj	division	$\%rj \leftarrow \frac{\%ri}{\text{reg/cst}}$
andcc	%ri, reg/cst, %rj	et logique	$\%rj \leftarrow \%ri \text{ and } \text{reg/cst}$
orcc	%ri, reg/cst, %rj	ou logique	$\%rj \leftarrow \%ri \text{ ou } \text{reg/cst}$
xorcc	%ri, reg/cst, %rj	ou exclusif	$\%rj \leftarrow \%ri \text{ xor } \text{reg/cst}$
xnorcc	%ri, reg/cst, %rj	<u>ou exclusif</u>	$\%rj \leftarrow \%ri \text{ xnor } \text{reg/cst}$
sll	%ri, reg/cst, %rj	décalage gauche	$\%rj \leftarrow \%ri \ll \text{reg/cst}$
srl	%ri, reg/cst, %rj	décalage droite	$\%rj \leftarrow \%ri \ll \text{reg/cst}$
sethi	val22, %ri	affectation	$\%ri \leftarrow \text{val22}$

reg/cst : registre ou constante signée sur 13 bits

val22 : constante non signée sur 22 bits

Instructions du processeur CRAPS

- Instructions de chargement/stockage

Instruction		Effet
ld	$[\%ri+reg/cst], \%rj$	lecture mémoire 32 bits
ldub	$[\%ri+reg/cst], \%rj$	lecture mémoire 8 bits
st	$\%ri, [\%rj+reg/cst]$	écriture mémoire 32 bits
stb	$\%ri, [\%rj+reg/cst]$	écriture mémoire 8 bits

reg/cst : registre ou constante signée sur 13 bits

- Instructions de contrôle

Instruction		Effet
call	adr	appel de sous-programme
jmp	adr, $\%ri$	saut avec lien de retour
ba	adr	branchement inconditionnel
bcond	adr	branchement si cond vrai

Détail des instructions de branchement

- Branchements conditionnels associés à un seul drapeau

Instruction	Signification	Test
ba	Branch Always	1
be	Branch on Equal	Z
bne	Branch on Not Equal	not Z
bneg	Branch on NEGative	N
bpos	Branch on POSitive	not N
bcs	Branch on Carry Set	C
bcc	Branch on Carry Clear	not C
bvs	Branch on oVerflow Set	V
bvc	Branch on oVerflow Clear	not V

Détail des instructions de branchement

- Branchements conditionnels associés à une arithmétique signée

Instruction	Signification	Test
bg	Branch on Greater	not (Z or (N xor V))
bge	Branch on Greater or Equal	not (N xor V)
bl	Branch on Less	N xor V
ble	Branch on Less or Equal	Z or (N xor V)

- Branchements conditionnels associés à une arithmétique non signée

Instruction	Signification	Test
bgu	Branch on Greater, Unsigned	not (Z or C)
bcc	Branch on greater or equal, unsigned	not C
bcs	Branch on less than, unsigned	C
bleu	Branch on Less or Equal, Unsigned	Z or C

Instructions synthétiques

Instruction	Effet	mise en œuvre
clr %ri	met à 0 %ri	orcc %r0,%r0,%ri
mov %ri,%rj	copie %ri dans %rj	orcc %ri,%r0,%rj
incc %ri	incrémente %ri	addcc %ri,1,%ri
decc %ri	décrompte %ri	subcc %ri,1,%ri
notcc %ri,%rj	%rj ← complément de %ri	xnorcc %ri,%ri,%rj
set val _{31...0} ,%ri	copie val dans %ri	sethi val _{31...10} ,%ri orcc %ri,val _{9...0} ,%ri
setq val _{12...0} ,%ri	copie val dans %ri	orcc %r0,val _{12...0} ,%ri
cmp %ri, %rj	compare %ri et %rj	subcc %ri,%rj,%r0
tst %ri	test de nullité et signe de %ri	orcc %ri,%r0,%r0
negcc %ri	inverse %ri	subcc %r0,%ri,%ri
nop	pas d'opération	sethi 0,%r0
jmp %ri	branchement à l'adresse absolue %ri	jmpl %ri,%r0
push %ri	empile %ri	sub %r30,4,%r30 st %ri,[%r30]
pop %ri	dépille %ri	ld [%r30],%ri add %r30,4,%r30

Exemple de programme

- Multiplication par deux des éléments d'un tableau T de N entiers

Pour i **de** 0 **à** $N-1$ **faire**

$T[i] \leftarrow T[i] \times 2;$

Fpour;

- Programme CRAPS correspondant

```

                                set    T, %r1
                                set    N, %r2
                                ld     [%r2], %r2
                                clr    %r3
pour:                          cmp    %r2, %r3
                                be     fpour
                                ld     [%r1], %r4
                                sll    %r4, 1, %r4
                                st     %r4, [%r1]
                                add    %r1, 4, %r1
                                inccc  %r3
                                ba     pour
fpour:                          ...
```

Exercice 4

- Que calcule le programme suivant ?

```
set    X, %r1
ld     [%r1], %r1
set    Y, %r2
ld     [%r2], %r2
set    Z, %r3
ld     [%r3], %r3
set    M, %r4
subcc  %r1, %r2, %r0
bge    S1
subcc  %r2, %r3, %r0
bge    S2
st     %r3, [%r4]
ba     fin
S2:    st     %r2, [%r4]
ba     fin
S1:    subcc  %r1, %r3, %r0
bge    S3
st     %r3, [%r4]
ba     fin
S3:    st     %r1, [%r4]
fin:   ba     fin
X:     .word  50
Y:     .word  30
Z:     .word  40
M:     .word  0
```

Classes d'instructions d'un processeur

- Instructions arithmétiques et logiques
 - ▶ Addition, soustraction, incrémentation, décrémentation
 - ▶ Et logique, ou logique, non logique, ...
 - ▶ Décalages logiques et arithmétiques, rotations
 - ▶ Multiplication, division
- Instructions de mouvement
 - ▶ De la mémoire vers un registre
 - ▶ D'un registre vers la mémoire
 - ▶ D'un registre vers un registre
- Instructions de branchement
 - ▶ Branchements inconditionnels ou conditionnels
 - ▶ Branchements absolus ou relatifs
- Instructions système

Décalages et rotations

- Décalage à gauche de x positions \Rightarrow Multiplication par 2^x

Avant

v_7	v_6	v_5	v_4	v_3	v_2	v_1	v_0
v_5	v_4	v_3	v_2	v_1	v_0	0	0

Après ($x = 2$)

- Décalage logique à droite de x positions

Avant

v_7	v_6	v_5	v_4	v_3	v_2	v_1	v_0
0	0	v_7	v_6	v_5	v_4	v_3	v_2

Après ($x = 2$)

- Décalage arithmétique à droite de x positions \Rightarrow Division par 2^x

Avant

v_7	v_6	v_5	v_4	v_3	v_2	v_1	v_0
v_7	v_7	v_7	v_6	v_5	v_4	v_3	v_2

Après ($x = 2$)

- Rotation à gauche de x positions

Avant

v_7	v_6	v_5	v_4	v_3	v_2	v_1	v_0
v_5	v_4	v_3	v_2	v_1	v_0	v_7	v_6

Après ($x = 2$)

- Rotation à droite de x positions

Avant

v_7	v_6	v_5	v_4	v_3	v_2	v_1	v_0
v_1	v_0	v_7	v_6	v_5	v_4	v_3	v_2

Après ($x = 2$)

Différents modes d'adressage

- L'adressage immédiat

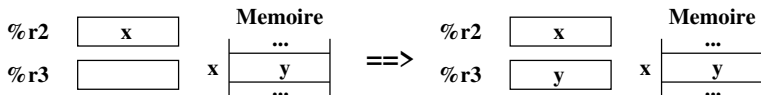
- ▶ La valeur de l'opérande est directement dans l'instruction
- ▶ Processeur RISC \Rightarrow l'intervalle des valeurs possibles est limité
- ▶ Exemple : `add %r2, 5, %r3` ($\%r3 \leftarrow \%r2 + 5$)
La valeur 5 est codée dans l'instruction

- L'adressage direct registre

- ▶ La valeur de l'opérande se trouve dans un registre
- ▶ Exemple : `add %r2, 5, %r3` ($\%r3 \leftarrow \%r2 + 5$)
Le numéro du registre où se trouve la valeur est codé dans l'instruction

- L'adressage indirect registre

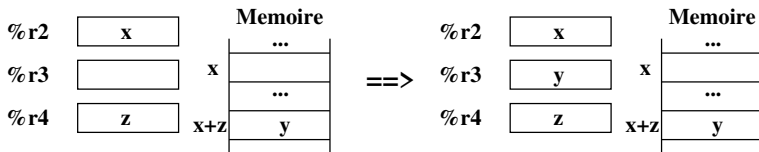
- ▶ L'adresse de l'opérande se trouve dans un registre
- ▶ Exemple `ld [%r2], %r3`



Différents modes d'adressage

- L'adressage indexé

- ▶ L'adresse de l'opérande est obtenue après ajout d'un index
- ▶ Exemple : `ld [%r2+%r4],%r3`



- L'adressage direct mémoire

- ▶ L'adresse de l'opérande se trouve dans l'instruction
- ▶ Non présent sur le CRAPS

- L'adressage indirect mémoire

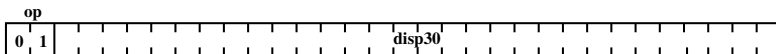
- ▶ L'adresse de l'adresse de l'opérande se trouve dans l'instruction
- ▶ Peut permettre d'accélérer la manipulation des pointeurs
- ▶ Non présent sur le CRAPS

Codage des instructions

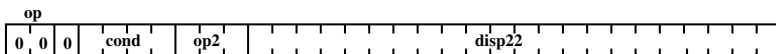
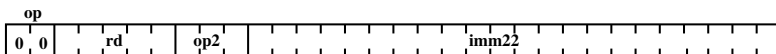
- Doit permettre de déterminer
 - ▶ L'instruction à exécuter
 - ▶ Les opérandes de cette instruction
- Le codage doit faciliter le décodage par le processeur
 - ▶ Les codes opérations sont structurés
 - ▶ Les opérandes sont toujours codés au même endroit
 - ▶ Pour les instructions arithmétiques ou logiques, la commande à envoyer à l'UAL est "en clair"

Structure binaire des groupes d'instructions du CRAPS

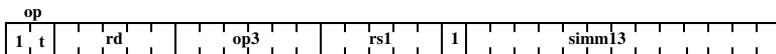
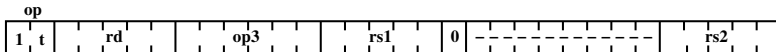
- Format 1 : *call*



- Format 2 : *sethi* et branchements



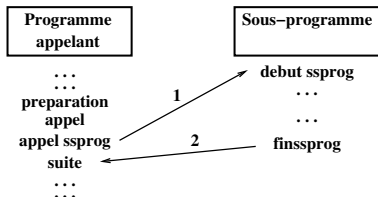
- Format 3 : accès mémoire, instructions arithmétiques



t = 1 : acces memoire, t = 0 : instruction arithmetique

La mise en œuvre de sous-programmes

- Principe général de l'appel d'un sous-programme
 - ▶ Préparation des données nécessaires à l'exécution du sous-programme
⇒ passage des paramètres
 - ▶ Branchement à la première instruction du sous-programme ⇒ rupture de séquence
 - ▶ Exécution de la séquence d'instructions correspondant au sous-programme
 - ▶ A la fin de cette séquence, retour au programme appelant (à l'instruction qui suit l'appel au sous-programme)

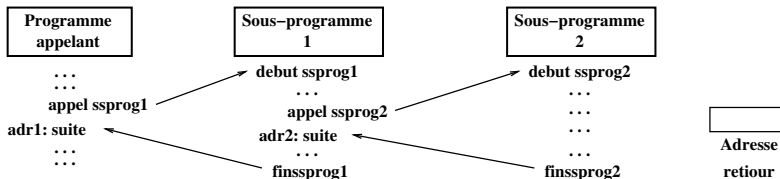


La nécessité d'utiliser une pile

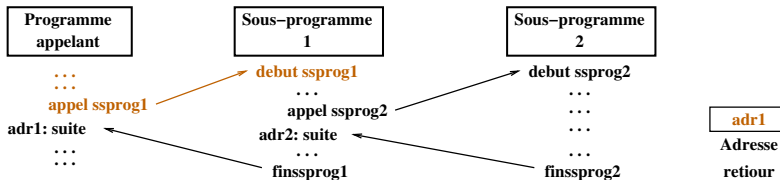
- L'utilisation de registres ne permet pas d'imbriquer des appels de sous-programmes
 - ▶ Un seul registre pour l'adresse de retour \Rightarrow tout nouvel appel écrase l'adresse de retour précédente
 - ▶ Les paramètres d'un sous-programme donné sont toujours dans les mêmes registres \Rightarrow Problématique pour mettre en œuvre une fonction récursive
- Un sous-programme utilise (modifie) des registres \Rightarrow le programme appelant devrait connaître ces registres
- La solution : l'utilisation d'une pile
 - ▶ Structure permettant de sauvegarder un nombre de données déterminé par la taille de la pile
 - ▶ Deux opérations possibles
 - ★ **empiler** : sauvegarde d'une donnée en sommet de pile
 - ★ **dépiler** : récupération de la donnée en sommet de pile
- La pile est le plus souvent située en mémoire centrale

Appels imbriqués sans pile

• Etape 1

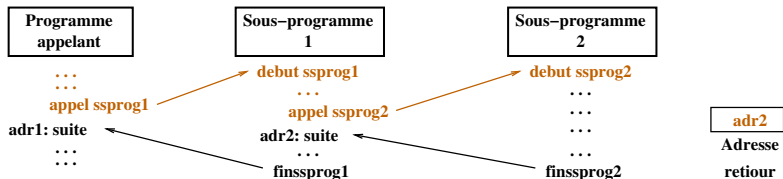


• Etape 2

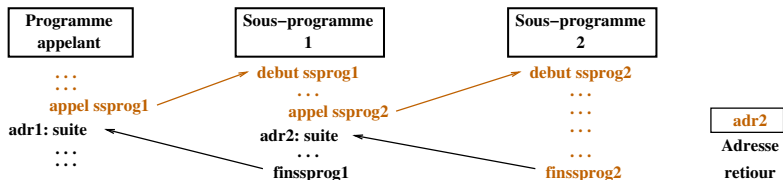


Appels imbriqués sans pile

• Etape 3

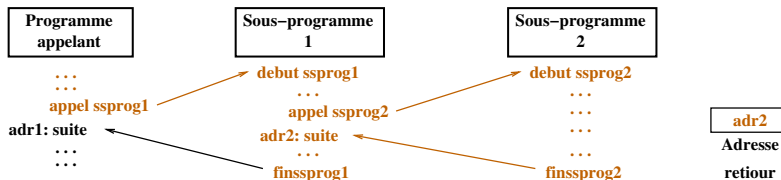


• Etape 4



Appels imbriqués sans pile

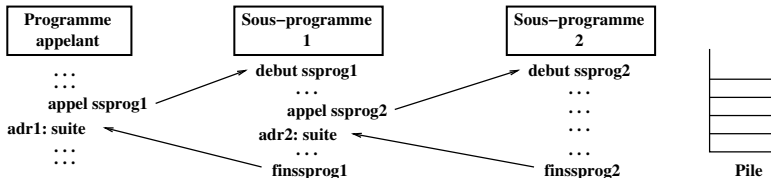
- Etape 5



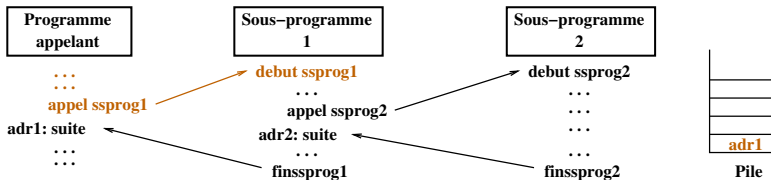
- \Rightarrow On a perdu le première adresse de retour

Appels imbriqués avec une pile

• Etape 1

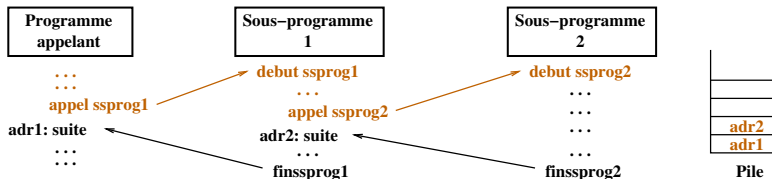


• Etape 2

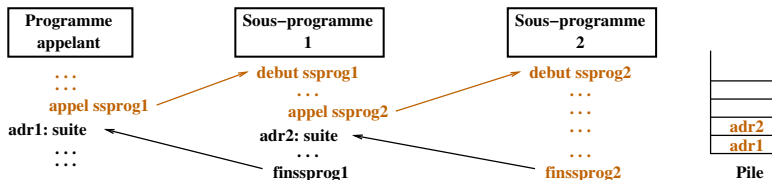


Appels imbriqués avec une pile

• Etape 3

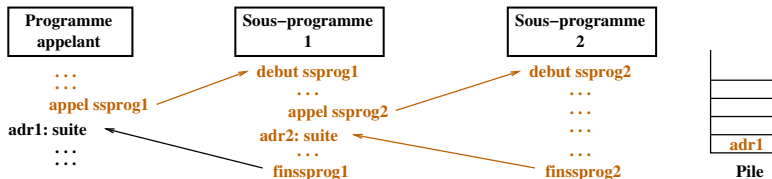


• Etape 4

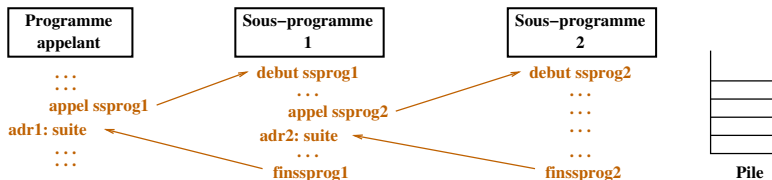


Appels imbriqués avec une pile

• Etape 5



• Etape 6



La mise en œuvre d'une pile

- Choix de l'ordre dans lequel on empile
 - ▶ Pile montante : par adresses décroissantes
 - ▶ Pile descendante : par adresses croissantes
- Pointeur de sommet de pile : permet de savoir où on se situe dans la pile. Deux solutions
 - ▶ Pointeur sur la dernière donnée empilée
 - ▶ Pointeur sur le premier emplacement libre
- La pile du CRAPS
 - ▶ Pile montante
 - ▶ Pointeur de sommet de pile : le registre %r30
 - ▶ %r30 pointe sur la dernière donnée empilée
 - ▶ Opération empiler : *push %ri*
sub %r30, 4, %r30
st %ri, [%r30]
 - ▶ Opération dépiler : *pop %ri*
ld [%r30], %ri
add %r30, 4, %r30

Exercice 5

- On considère la pile du CRAPS avec une adresse de base à 2000 (décimal)
- Au départ, la pile est vide
- On effectue la séquence d'opérations suivante

empiler (14)

empiler (7)

empiler (3)

$x \leftarrow \text{dépiler} ()$

$y \leftarrow \text{dépiler} ()$

empiler ($x + y$)

empiler (18)

$x \leftarrow \text{dépiler} ()$

$y \leftarrow \text{dépiler} ()$

empiler ($y - x$)

- Donner la valeur du pointeur de sommet de pile et le contenu de la pile à l'issue de cette séquence

Les entrées/sorties

- Objectif : permettre au processeur d'échanger des données avec l'extérieur
- Une entrée : une donnée transférée d'un périphérique (clavier, disque, ...) vers le processeur
- Une sortie : une donnée transférée du processeur vers un périphérique
- Problèmes à résoudre
 - ▶ Définir les adresses des entrées et des sorties
 - ▶ Réaliser la connexion physique entre le périphérique et le processeur
 - ▶ Mettre en œuvre un protocole qui garantisse qu'il n'y a, ni perte de données, ni duplication
- Modèle fonctionnel



Protocole de synchronisation processeur-périphérique

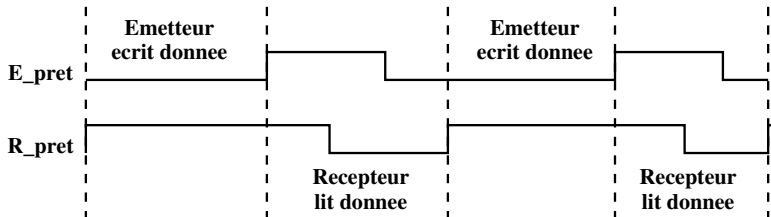
- Pas d'hypothèse sur le rythme des lectures et des écritures du processeur et des périphériques
- Pas d'hypothèse sur les vitesses relatives du processeur et des périphériques
- On suppose qu'il n'y a pas d'erreurs de transmission
 - ▶ Hypothèse réaliste dans le cadre d'un ensemble processeur-périphériques
 - ▶ Simplifie considérablement le protocole de synchronisation
- Un exemple de protocole de synchronisation : la poignée de mains
 - ▶ Protocole mis en œuvre entre un émetteur et un récepteur
 - ▶ Le récepteur ne lit la donnée que lorsque l'émetteur lui signale qu'il y a une nouvelle occurrence de la donnée
 - ▶ L'émetteur n'écrit une nouvelle donnée que lorsque le récepteur lui signale qu'il a lu l'occurrence précédente

Le protocole de poignée de mains

- Protocole fondé sur deux signaux binaires de synchronisation
 - ▶ $E_prêt$: émetteur prêt, écrit par l'émetteur et lu par le récepteur
 - ▶ $R_prêt$: récepteur prêt, écrit par le récepteur et lu par l'émetteur
- Déroulement des opérations
 - ▶ Du côté de l'émetteur
 - 1 $E_prêt$ est initialisé à faux
 - 2 L'émetteur produit la prochaine occurrence de la donnée
 - 3 L'émetteur attend que $R_prêt$ soit à vrai
 - 4 Dès que c'est le cas, l'émetteur envoie la donnée au récepteur
 - 5 L'émetteur positionne $E_prêt$ à vrai
 - 6 L'émetteur attend que $R_prêt$ passe à faux
 - 7 L'émetteur positionne alors $E_prêt$ à faux
 - ▶ Du côté du récepteur
 - 1 Le récepteur positionne $R_prêt$ à vrai
 - 2 Le récepteur attend que $E_prêt$ soit à vrai
 - 3 Dès que c'est le cas, le récepteur lit une occurrence de la donnée
 - 4 Le récepteur positionne $R_prêt$ à faux
 - 5 Le récepteur attend que $E_prêt$ passe à faux

Le protocole de poignée de mains

- Stricte alternance entre l'écriture d'une nouvelle donnée par l'émetteur et sa lecture par le récepteur

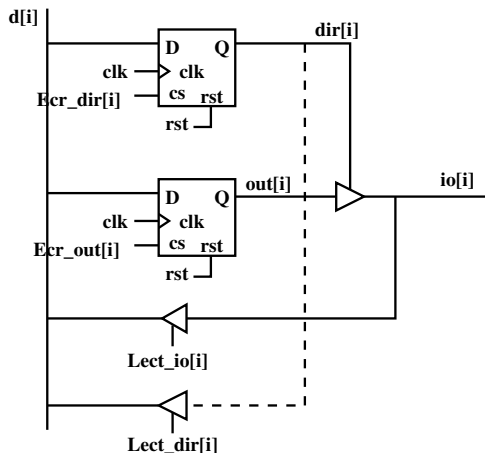


Définition des adresses des entrées et des sorties

- Sur certains processeurs, instructions d'entrées-sorties spécifiques, avec des numéros de ports
- Exemple de la famille x86
 - ▶ *IN AL,PORT* : lecture d'un octet de *PORT* vers *AL*
 - ▶ *OUT PORT,AL* : écriture d'un octet de *AL* vers *PORT*
- Sur d'autres processeurs, entrées-sorties projetées en mémoire \Rightarrow accès aux entrées et aux sorties via des instructions de lecture et d'écriture en mémoire
- Exemple du CRAPS
 - ▶ Possibilité d'allumer ou d'éteindre des leds
 - ★ Ecriture à l'adresse mémoire 0xB0000000
 - ▶ Possibilité de lire l'état de switches
 - ★ Lecture à l'adresse 0x90000000

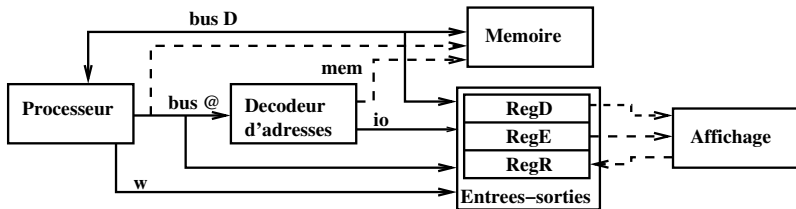
Connexion physique entre un processeur et un périphérique

- Pouvoir configurer une ligne en entrée ou en sortie
- Pouvoir stocker les valeurs courantes des sorties
- Pouvoir lire les valeurs des entrées
- Un exemple de mise en œuvre



Programmation de l'affichage d'un texte

- Mise en œuvre d'un programme réalisant des échanges de données entre le processeur et un ou plusieurs périphériques
- Un petit exemple illustratif : un processeur envoie une suite de caractères à un dispositif d'affichage
- Organisation matérielle du système



- Les ports d'entrées-sorties ne sont pas configurables
- *RegD* contient le caractère en cours de transmission
- *RegE* permet à l'émetteur (le processeur) d'envoyer des informations de contrôle au récepteur (le dispositif d'affichage)
- *RegR* permet au récepteur d'envoyer des informations de contrôle à l'émetteur

Programmation de l'affichage d'un texte

- Séquence de caractères stockée dans le tableau *Texte* qui contient *N* caractères
- Synchronisation des échanges entre le processeur et le dispositif d'affichage par un protocole de poignée de mains
 - ▶ Signal *E_Prêt* : le bit de poids faible de *RegE*
 - ▶ Signal *R_Prêt* : le bit de poids faible de *RegR*
- Algorithme d'envoi du texte au dispositif d'affichage

$\text{RegE} \leftarrow 0;$

pour *i* **depuis** 0 **jqa** *N*-1 **faire**

 /* boucle d'attente active */

répéter $\text{ep} \leftarrow \text{RegR}$ et 1; **jqa** $\text{ep} = 1;$

$\text{RegD} \leftarrow \text{Texte}[i];$

$\text{RegE} \leftarrow 1;$

 /* boucle d'attente active */

répéter $\text{ep} \leftarrow \text{RegR}$ et 1; **jqa** $\text{ep} = 0;$

$\text{RegE} \leftarrow 0;$

finpour

Le mécanisme d'interruptions

- Motivation

- ▶ Synchronisation des échanges de données \Rightarrow attente de la valeur d'un signal pour pouvoir effectuer une opération
- ▶ Solution adoptée jusqu'à présent : lecture en permanence du signal, jusqu'à ce qu'il ait la valeur voulue (attente active) \Rightarrow le processeur ne fait rien d'autre
- ▶ On peut espacer la lecture du signal \Rightarrow augmentation du délai moyen entre l'apparition de la valeur souhaitée et l'opération correspondante

- Une solution

- ▶ Mise en place d'un mécanisme qui prévient automatiquement le processeur lorsque le signal a la valeur voulue \Rightarrow un coup de sonnette
- ▶ Définir le comportement du processeur à la réception de cette information

Le mécanisme d'interruptions

- Définition

- ▶ Une interruption est une rupture dans le flot d'exécution des instructions, provoquée par un événement exceptionnel (\Rightarrow exception) et conduisant à l'exécution d'une séquence de traitement de cet événement et, si possible, à la reprise du flot interrompu (\Rightarrow s'apparente un peu à un appel de sous-programme, mais à un instant non connu à l'avance)

- Deux types d'exceptions

- ▶ Exceptions internes (traps) : provoquées par l'exécution d'une instruction par le processeur \Rightarrow synchrones au déroulement du programme en cours
 - ★ Instruction illégale
 - ★ Bus error
 - ★ Division par 0
 - ★ Erreur d'alignement
 - ★ ...
- ▶ Exceptions externes (interruptions) : généralement événement d'entrées-sorties \Rightarrow asynchrones au déroulement du programme en cours

Principe de la prise en compte d'une interruption

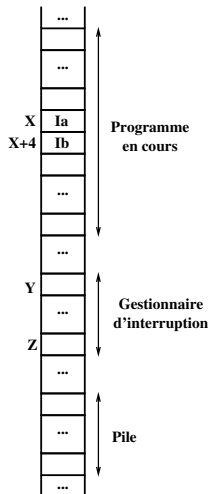
- Architecture matérielle



- A l'activation du signal **irq**, si la prise en compte des interruptions est autorisée
 - ▶ Fin de l'exécution de l'instruction en cours
 - ▶ Sauvegarde dans la pile des informations nécessaires à la reprise du programme en cours
 - ▶ Calcul de l'adresse de la procédure de traitement de l'interruption (gestionnaire d'interruption, routine d'interruption)
 - ▶ Branchement à cette adresse ⇒ exécution du gestionnaire d'interruption
 - ▶ Fin de l'exécution du gestionnaire d'interruption ⇒ restitution des informations nécessaires à la reprise du programme interrompu
 - ▶ Exécution de la suite du programme interrompu

Principe de la prise en compte d'une interruption

● Exemple



Au moment de l'interruption

PC vaut **X**, exécution de **la**

Fin de l'exécution de **la**

PC (**X+4**) et autres infos dans la pile

PC \leftarrow **Y**

Exécution du gestionnaire d'interruptions

Fin du gestionnaire (**PC** vaut **Z**)

Récupère **PC** et autres infos de la pile

Exécution de **lb** et de la suite du programme
interrompu

Principe de la prise en compte d'une interruption

- Les points à traiter

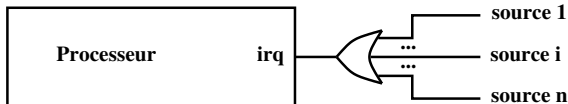
- ▶ Mise en œuvre de l'autorisation ou de l'interdiction des interruptions
- ▶ Prise en compte de différents types d'interruptions
 - ★ Niveaux de priorités
 - ★ Autorisation/interdiction sélective
- ▶ Informations à sauvegarder pour la reprise du programme interrompu
- ▶ Calcul de l'adresse du gestionnaire d'interruption

Autorisation/interdiction des interruptions

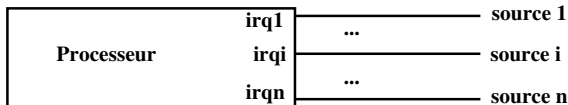
- Information binaire \Rightarrow un bit qui dit si les interruptions sont autorisées (bit à 0) ou interdites (bit à 1)
- Bit placé dans le registre d'état (PSR)
- Quelques règles classiques d'autorisation/interdiction des interruptions
 - ▶ Interruptions interdites au démarrage du processeur
 - ▶ Interruptions autorisées lorsqu'on souhaite communiquer avec un périphérique d'entrées/sorties
 - ▶ Interruptions interdites lorsqu'on exécute le gestionnaire d'interruptions (le gestionnaire d'interruptions ne s'interrompt pas lui-même)
- Exemple de fonctionnement
 - < Initialisation du processeur >
 - (Interruptions interdites, bit de PSR à 1)
 - < On autorise les interruptions (bit de PSR à 0) >
 - < Une interruption arrive >
 - < On interdit les interruptions (bit de PSR à 1) >
 - < On exécute le gestionnaire d'interruptions >
 - < On autorise les interruptions (bit de PSR à 0) >

Différentes sources d'interruptions

- Différentes possibilités pour le gérer
 - ▶ Une seule entrée au niveau du processeur



- ★ Source d'interruption déterminée en consultant les sources dans un ordre le plus souvent prédéfini \Rightarrow ordre de priorité entre les sources
- ▶ Plusieurs entrées d'interruptions sur le processeur \Rightarrow une priorité allouée à chaque entrée



- ★ Traitement de l'interruption en attente la plus prioritaire, dès que le niveau d'interruption correspondant est autorisé
- ★ Niveau de priorité des interruptions autorisés précisé par des bits du registre d'état

Préparer la reprise du programme interrompu

- Adresse de l'instruction courante du programme interrompu
 - ▶ PC sauvegardé dans la pile
- Niveau de priorité des interruptions autorisés et indicateurs ZNVC
 - ▶ PSR sauvegardé dans la pile
- Contenu de l'ensemble des registres utilisateur : plusieurs solutions
 - ▶ Sauvegarde systématique de tous les registres dans la pile par le processeur \Rightarrow long
 - ▶ Sauvegarde des seuls registres modifiés par le gestionnaire dans la pile, par le programmeur

Calcul de l'adresse du gestionnaire d'interruptions

- Solution avec une seule entrée d'interruptions
 - ▶ Un seul gestionnaire d'interruptions placé à une adresse fixe (ex : 0x38 pour le Z80)
 - ▶ Le gestionnaire d'interruptions ne doit pas déborder de l'espace qui lui est alloué en mémoire \Rightarrow effectuer un branchement ailleurs en mémoire
 - ▶ Variante de l'adresse fixe : adresse du gestionnaire dans un registre, initialisé au démarrage du processeur
- Solution de base avec plusieurs entrées d'interruptions
 - ▶ Un gestionnaire par ligne d'interruptions
 - ▶ Chaque gestionnaire à une adresse prédéfinie \Rightarrow table des adresses des gestionnaires d'interruptions (vecteurs d'interruptions) \Rightarrow accès à la case correspondant à l'interruption à traiter
 - ▶ Remplacement de la table par un calcul si
 - ★ Tous les gestionnaires ont la même taille
 - ★ Ils sont les uns à la suite des autres en mémoire
- Solution si plus de gestionnaires que de lignes
 - ▶ Interruption \Rightarrow la source fournit le numéro du gestionnaire d'interruptions à exécuter
 - ▶ Une entrée supplémentaire pour la lecture du numéro de gestionnaire