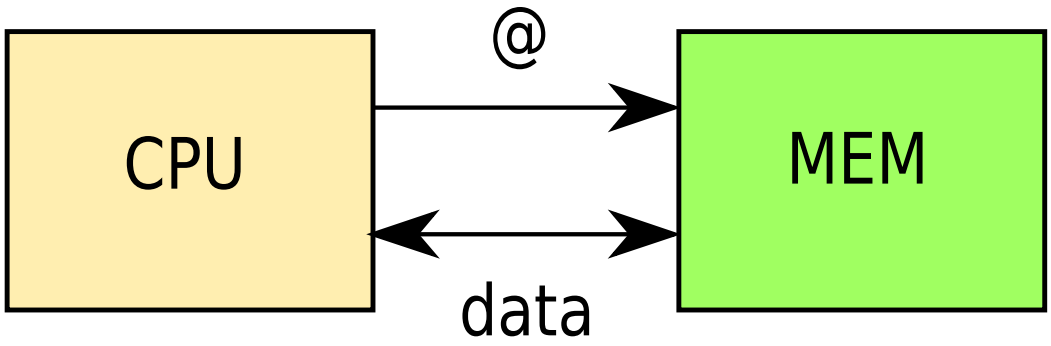
ENSTA Bretagne

Architecture des systèmes

Jean-Christophe Le Lann, L103

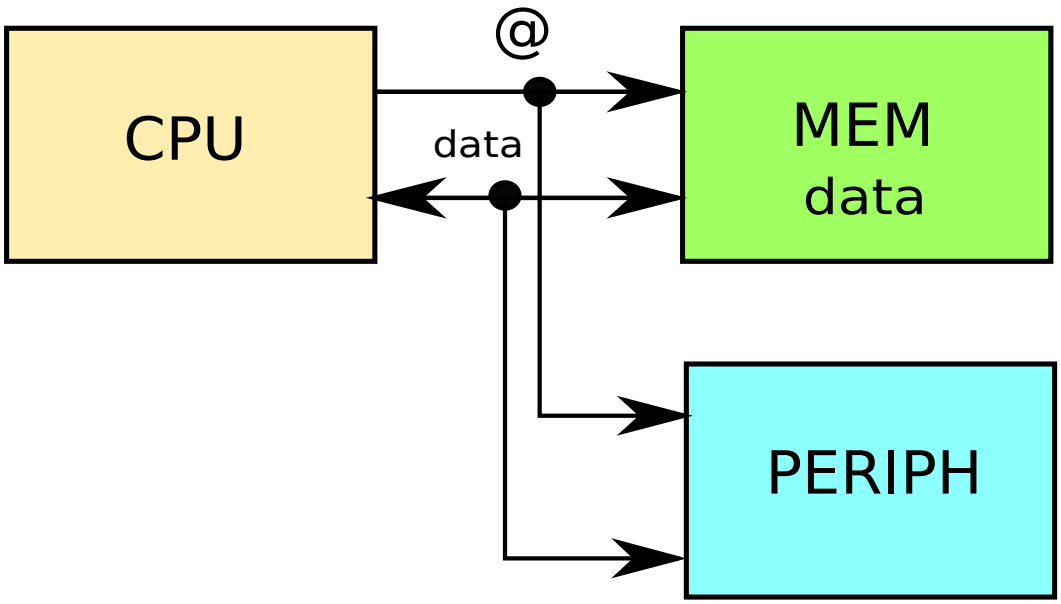
CPU et ISA

CPU : version simple



Les données et le programme résident ici dans la même mémoire : architecture de Von Neumann

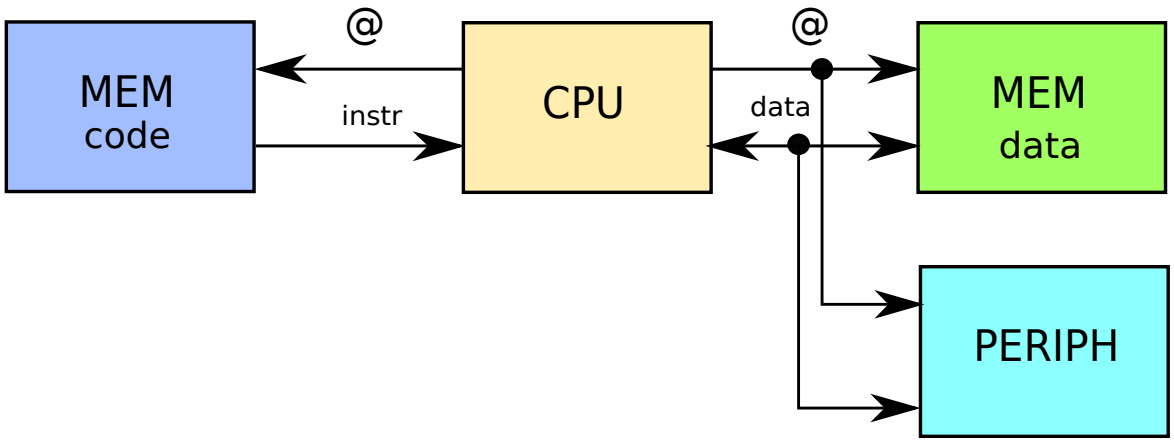
CPU : adjonction de périphériques



Périphériques « mappés » en mémoire : partagent le même espace d'adressage que les données

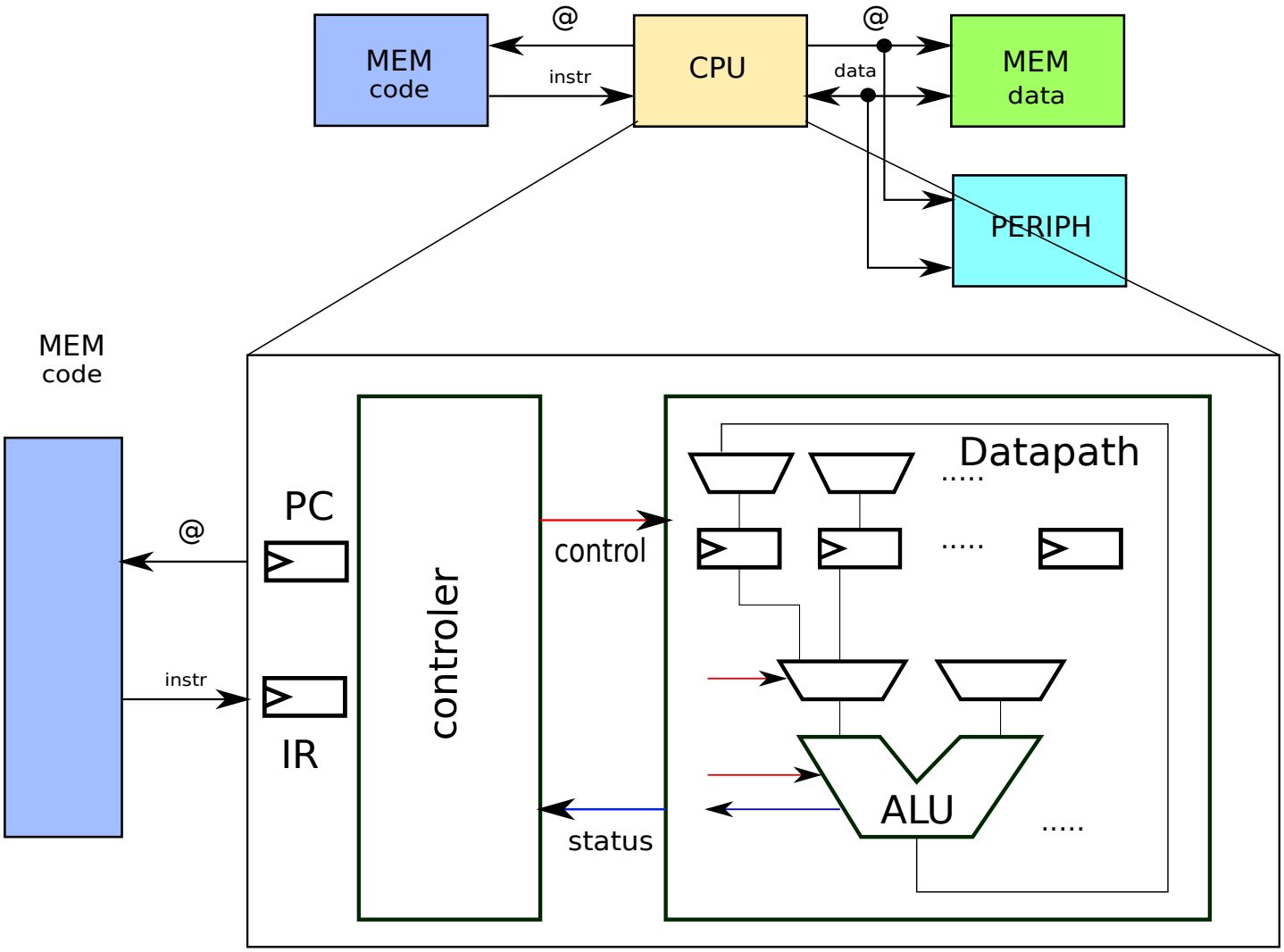
CPU : modèle Harvard

Séparation données/instructions



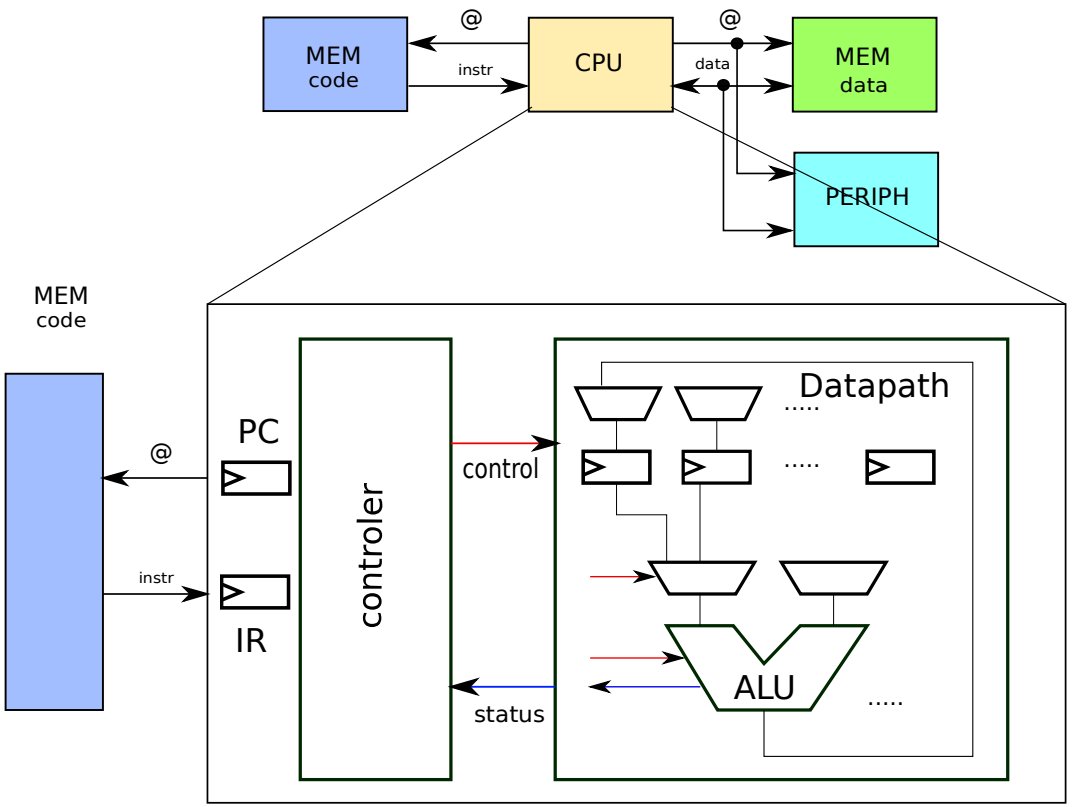
Le CPU peut accéder simultanément aux instructions et aux données

CPU : constitution interne



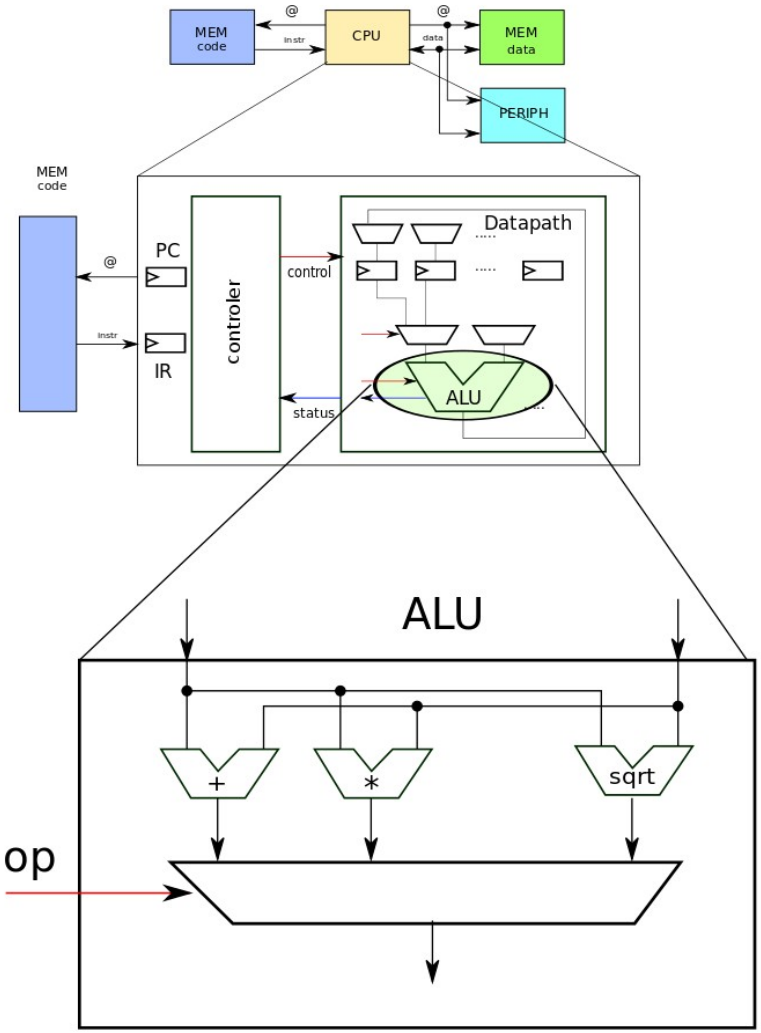
Si possible n'utiliser que les registres
= performance

Architecture VLIW



Variantes possibles
Le datapath peut posséder plusieurs ALUs/op qui calculent en parallèle : VLIW (very-long instruction word)

ALU



Une ALU se présente comme un ensemble d'opérateurs (plus ou moins partagés), multiplexés

ALU en VHDL

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.all;

entity ALU_VHDL is
  port(
    operand_1, operand_2 : in  std_logic_vector(3 downto 0);
    Operation            : in  std_logic_vector(2 downto 0);
    Carry_Out            : out std_logic;
    Flag                 : out std_logic;
    Result               : out std_logic_vector(3 downto 0));
end entity ALU_VHDL;

architecture Behavioral of ALU_VHDL is
  signal Temp : std_logic_vector(4 downto 0);
begin

  process(operand_1, operand_2, Operation, temp) is
  begin
    Flag <= '0';
    case Operation is
      when "000" =>   -- res = nib1 + nib2, flag = carry = overflow
        Temp      <= std_logic_vector((unsigned("0" & operand_1) + unsigned(operand_2)));
        Result    <= temp(3 downto 0);
        Carry_Out <= temp(4);
      when "001" =>   -- res = |nib1 - nib2|, flag = 1 iff nib2 > nib1
        if (operand_1 >= operand_2) then
          Result <= std_logic_vector(unsigned(operand_1) - unsigned(operand_2));
          Flag   <= '0';
        else
          Result <= std_logic_vector(unsigned(operand_2) - unsigned(operand_1));
          Flag   <= '1';
        end if;
      when "010" =>
        Result <= operand_1 and operand_2;
      when "011" =>
        Result <= operand_1 or operand_2;
      when "100" =>
        Result <= operand_1 xor operand_2;
      when "101" =>
        Result <= not operand_1;
      when "110" =>
        Result <= not operand_2;
      when others =>                    -- res = nib1 + nib2 + 1, flag = 0
        Temp   <= std_logic_vector((unsigned("0" & operand_1) + unsigned(not operand_2)) + 1);
        Result <= temp(3 downto 0);
        Flag   <= temp(4);
    end case;
  end process;

end architecture Behavioral;
```

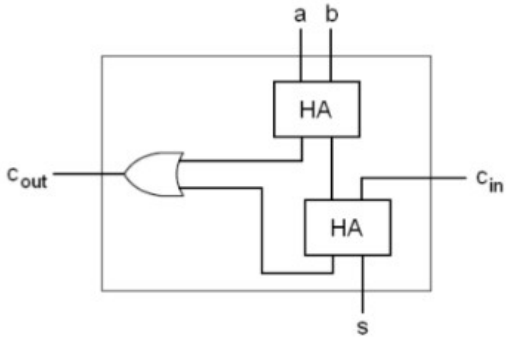
Exemple de partage de logique : + et -

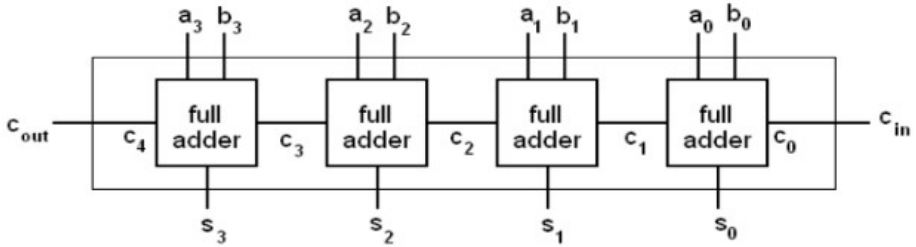
On établit la table de vérité.

a	b	f	cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Rappels

a	b	cin	S	cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



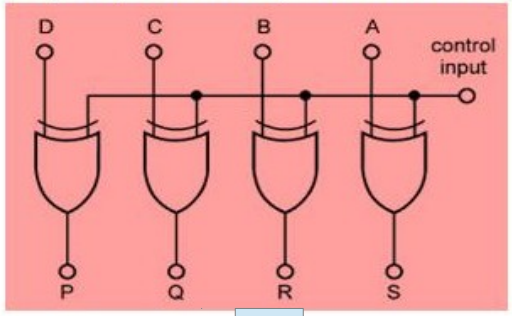


Exemple de partage de logique : + et -

When DCBA = 1011 and C.I. = 0, then, PQRS = 1011

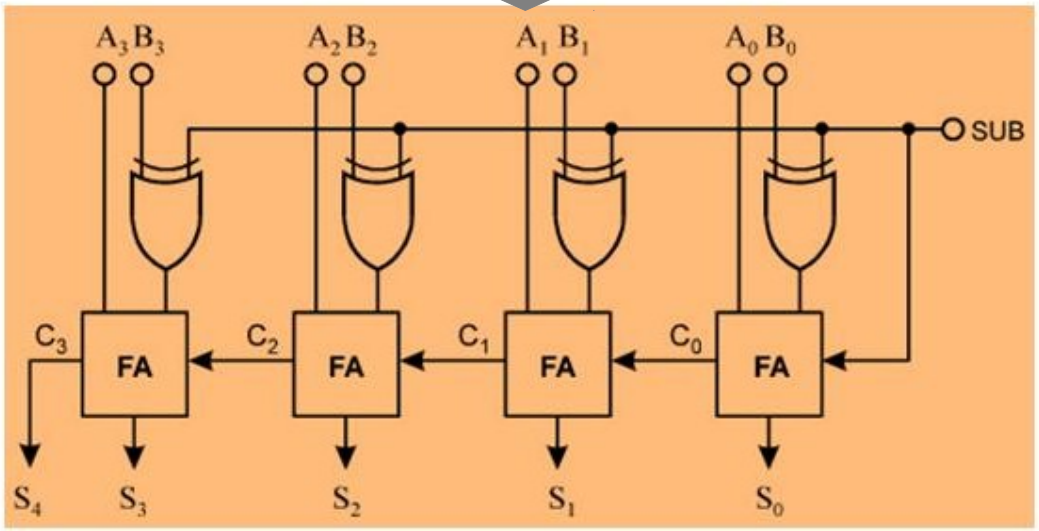
When DCBA = 1011 and C.I. = 1, then, PQRS = 0100, which is the 1's complement of 1011.

1



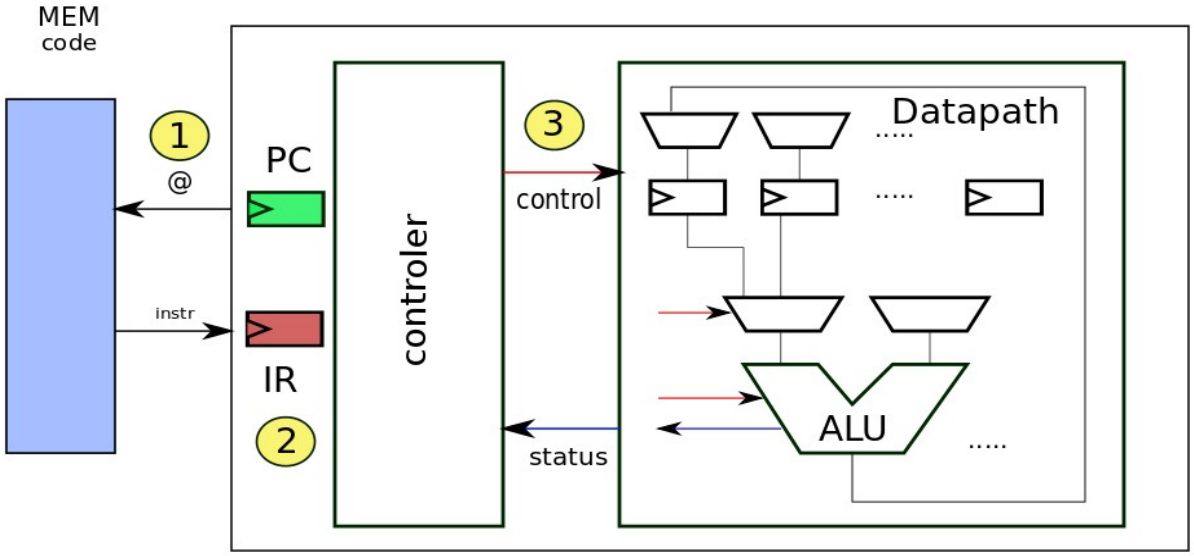
Signal de contrôle permettant la configuration du composant en additionneur Ou en additionneur

2



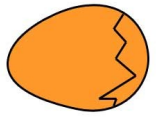
Cycle fetch-décode-exécute

Le processeur répète inlassablement ce cycle

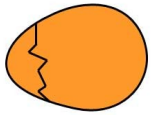


Deux remarques sur le fetch

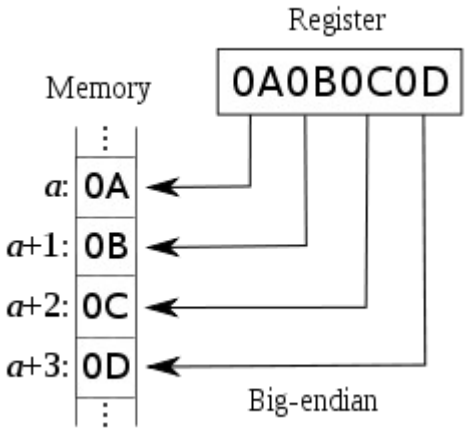
- Généralement ce sont les octets de la mémoire qui sont adressables : pour 32 bits, 4 octets. Entre 2 adresses successives : +4 (et non +1)
- Notion d'Endianness :

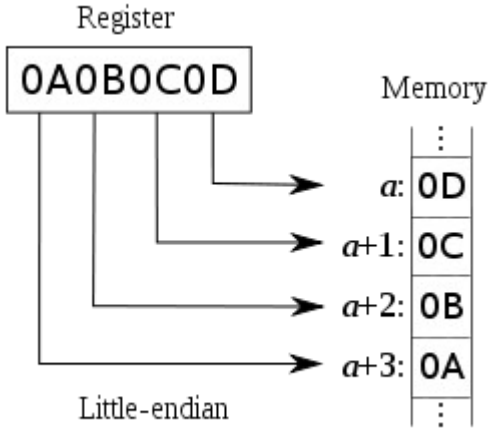


BIG ENDIAN - The way people always broke their eggs in the Lilliput land

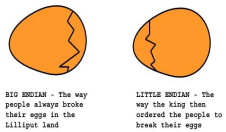


LITTLE ENDIAN - The way the king then ordered the people to break their eggs





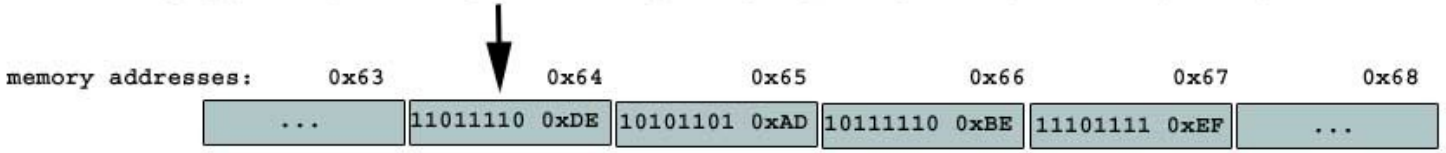
Endianness



base 10: 3735928559 = base 16: 0xDEADBEEF

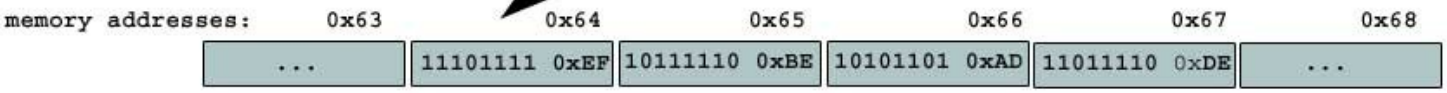
In Big Endian storage schemes, the most significant byte is stored "first" - i.e. "big end" goes to the lower address. Memory is addressed/accessed from low to high addresses

base 2: 11011110 10101101 10111110 11101111

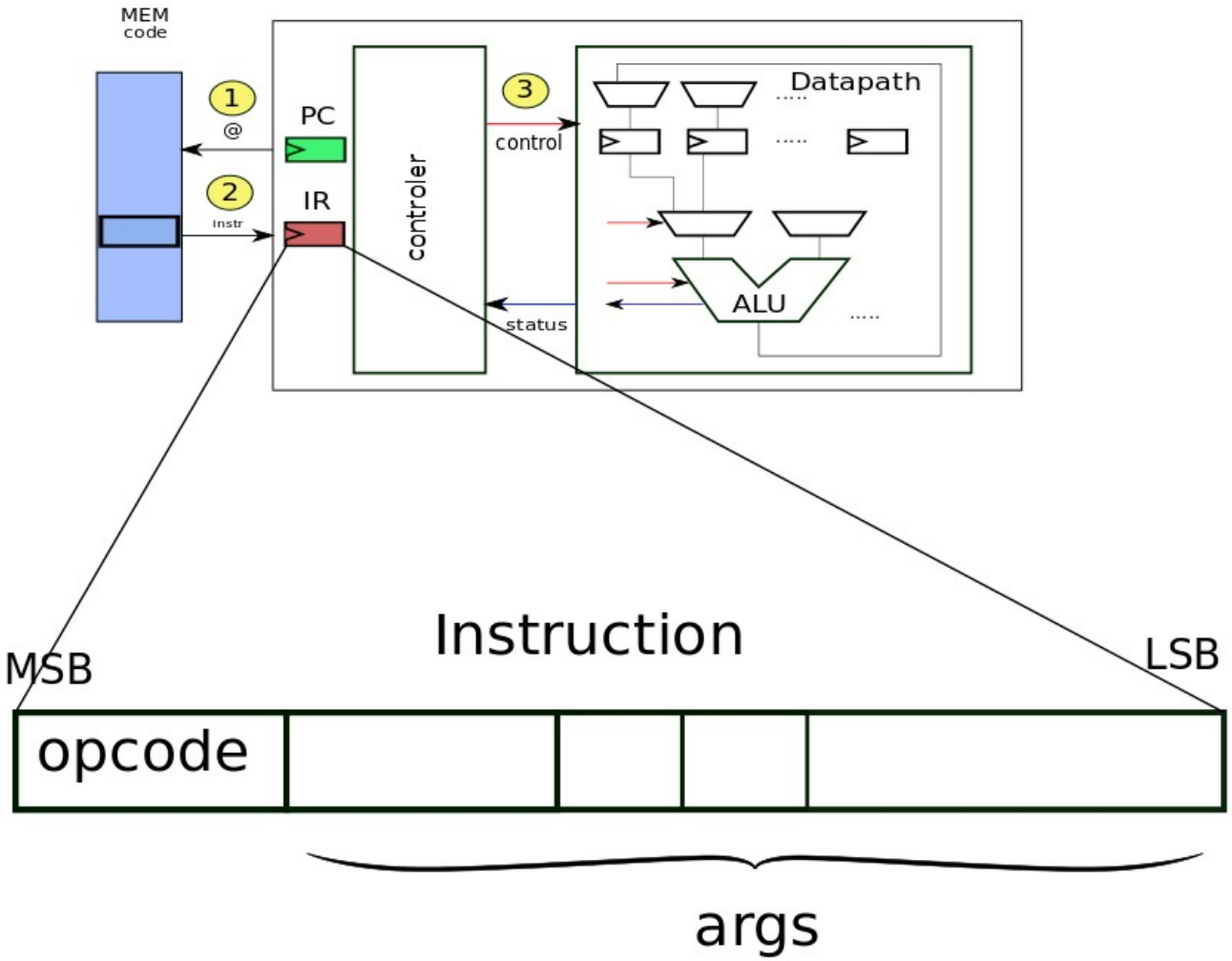


In Little Endian storage schemes, the least significant byte is stored "first" - i.e. "little end" goes to the lower address. Memory is addressed/accessed from low to high.

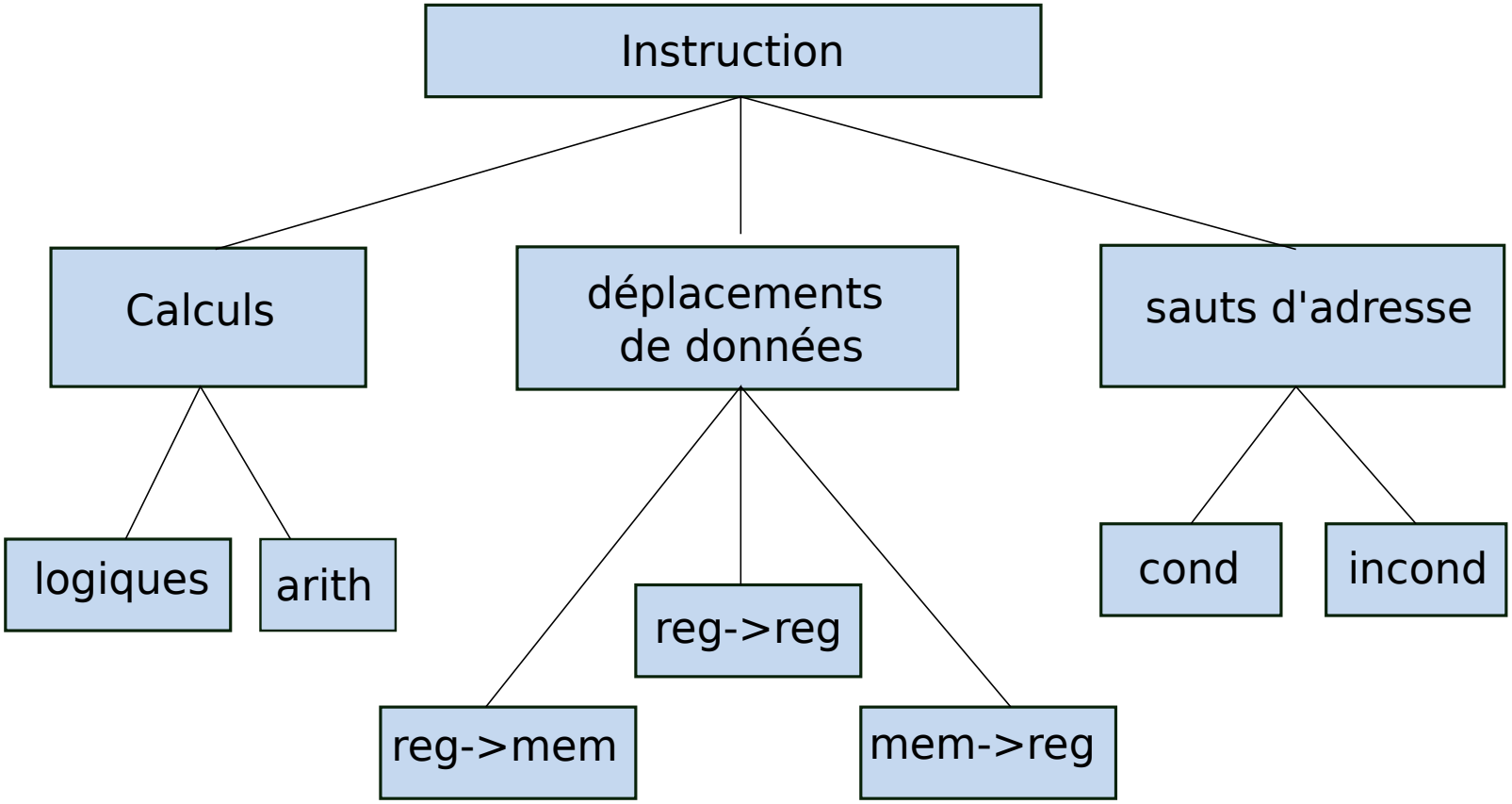
base 2: 11011110 10101101 10111110 11101111



Instructions



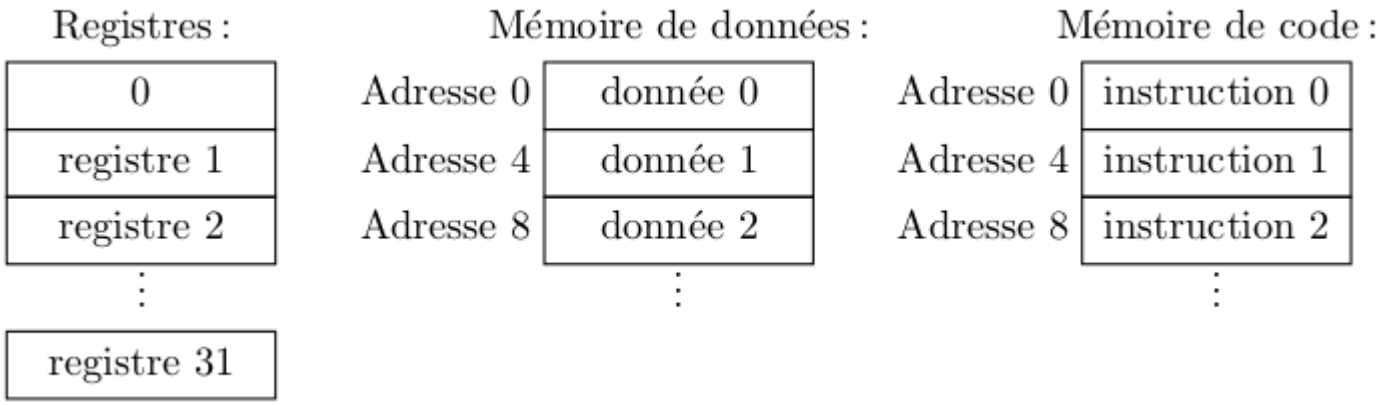
Grands types d'instructions



"store" "move" "load" "braz" "jmp"

Exemple de MIPS-X

Modèle de programmation :



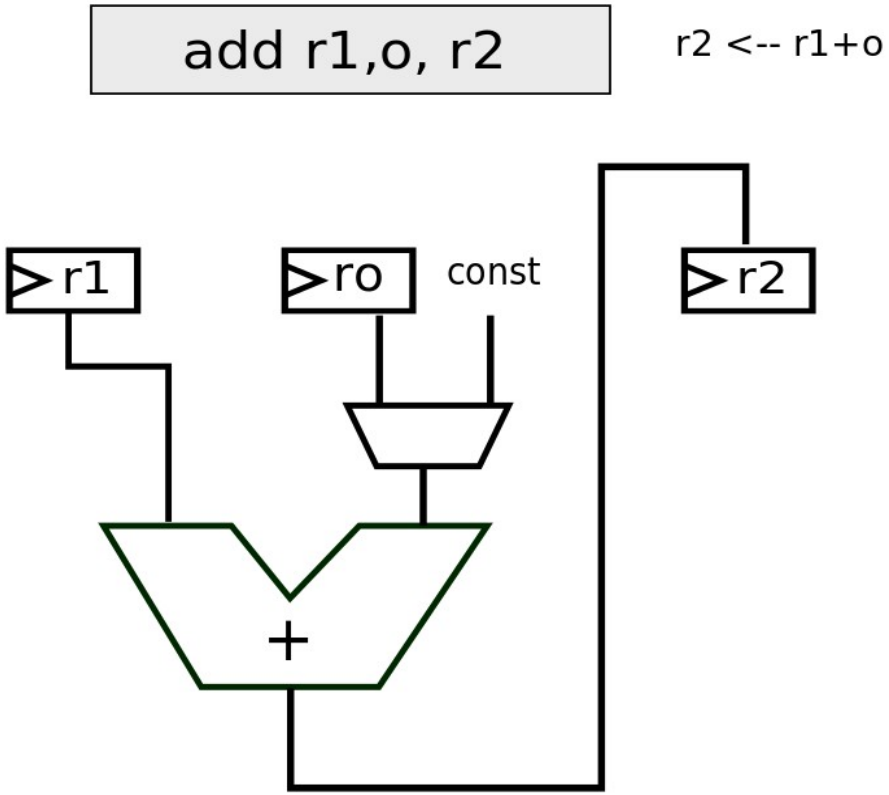
Jeu d'instruction du MIPS-X

Notations : r nom de registre (`r 0`, `r 1`, ..., `r 31`)
o nom de registre ou constante entière (`12`, `-34`, ...)
a constante entière

Syntaxe	Instruction	Effet
`add`(r_1, o, r_2)	Addition entière	r_2 reçoit $r_1 + o$
`sub`(r_1, o, r_2)	Soustraction entière	r_2 reçoit $r_1 - o$
`mult`(r_1, o, r_2)	Multiplication entière	r_2 reçoit $r_1 * o$
`div`(r_1, o, r_2)	Quotient entier	r_2 reçoit r_1/o
`and`(r_1, o, r_2)	« Et » bit à bit	r_2 reçoit r_1 « et » o
`or`(r_1, o, r_2)	« Ou » bit à bit	r_2 reçoit r_1 « ou » o
`xor`(r_1, o, r_2)	« Ou exclusif » bit à bit	r_2 reçoit r_1 « ou exclusif » o
`shl`(r_1, o, r_2)	Décalage arithmétique logique à gauche	r_2 reçoit r_1 décalé à gauche de o bits
`shr`(r_1, o, r_2)	Décalage arithmétique logique à droite	r_2 reçoit r_1 décalé à droite de o bits
`slt`(r_1, o, r_2)	Test « inférieur »	r_2 reçoit 1 si $r_1 < o$, 0 sinon
`sle`(r_1, o, r_2)	Test « inférieur ou égal »	r_2 reçoit 1 si $r_1 \leq o$, 0 sinon
`seq`(r_1, o, r_2)	Test « égal »	r_2 reçoit 1 si $r_1 = o$, 0 sinon
`load`(r_1, o, r_2)	Lecture mémoire	r_2 reçoit le contenu de l'adresse $r_1 + o$
`store`(r_1, o, r_2)	Écriture mémoire	le contenu de r_2 est écrit à l'adresse $r_1 + o$
`jmp`(o, r)	Branchement	saute à l'adresse o et stocke l'adresse de l'instruction suivant le `jmp` dans r
`braz`(r, a)	Branchement si zéro	saute à l'adresse a si $r = 0$
`branz`(r, a)	Branchement si pas zéro	saute à l'adresse a si $r \neq 0$
`scall`(n)	Appel système	n est le numéro de l'appel
`stop`	Arrêt de la machine	fin du programme

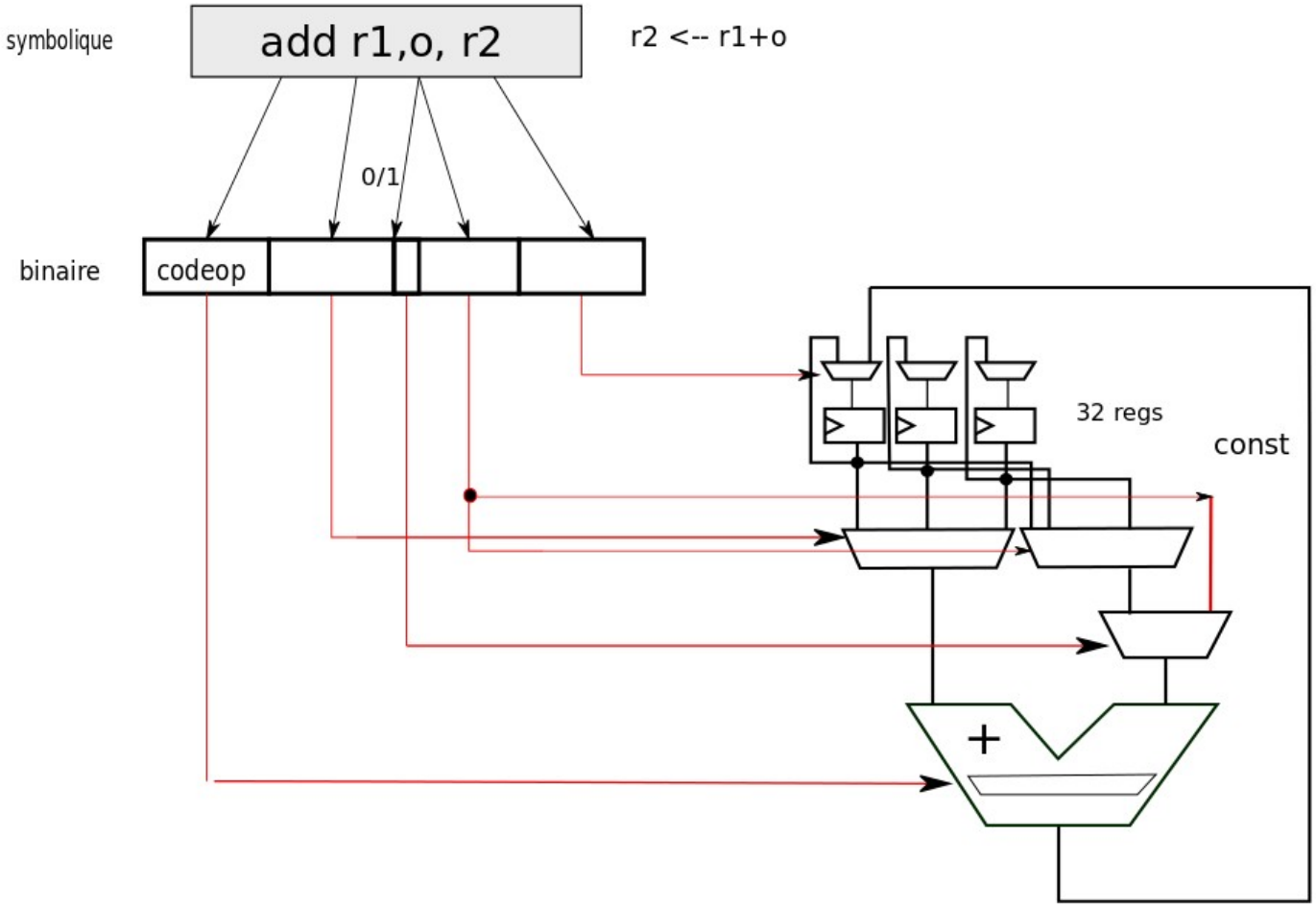
Instruction et datapath

En première instance :



Instruction et datapath

Plus près de la réalité...



Idiomes RISC

- RISC : reduced instruction set architecture
- Datapath simplifié = Performance en fréquence
- Comment contourner la pauvreté de l'ISA ?
  - ➔ Idiomes de programmation
    - Zéro comme argument
    - Registre zéro comme résultat
    - Négation booléenne et inversion de tests

Idiomes

Zéro comme argument Beaucoup d'opérations utiles s'obtiennent en fixant à zéro un des deux arguments d'une instruction, en prenant soit la constante 0, soit le registre `r 0` comme argument. Voici quelques exemples :

`add` $r_1, 0, r_2$	Copie r_1 dans r_2 (instruction `move`)
`add r 0,`n, r_2	Met la constante n dans r_2 (instruction `move`)
`sub r 0,`r_1, r_2	Met l'opposé de r_1 dans r_2 (instruction `neg`)
`braz r 0,`a	Saute à l'adresse a
`load` $r_1, 0, r_2$	Lit le mot à l'adresse (calculée) r_1
`load r 0,`a, r_2	Lit le mot à l'adresse (constante) a

Le registre zéro comme résultat Parfois, le résultat d'une opération est inutile. La manière standard de s'en débarrasser sans modifier aucun registre est de mettre `r 0` comme registre de destination. Par exemple, `jmp` a, `r 0` se branche à l'adresse a, sans mettre l'adresse de retour dans aucun registre.

Négation booléenne et inversion de tests En supposant les valeurs de vérité représentées par 0 pour « faux » et autre chose que 0 pour « vrai », l'instruction `seq` r_1, `r 0`, r_2 calcule la négation d'une valeur de vérité : si r_1 est « faux » (nul), r_2 est mis à « vrai » ; si r_1 est « vrai » (non nul), r_2 est mis à « faux ». Exemple d'application : le test « strictement plus grand » entre un registre r_1 et un registre ou une constante o, avec résultat dans r_2, se calcule par les deux instructions

```
sle   r1, o, r2
seq   r2, r 0, r2
```

La première instruction calcule la négation du résultat désiré (la négation de $r_1 > o$ est $r_1 \leq o$) ; la deuxième calcule la négation de cette négation, obtenant le résultat désiré.

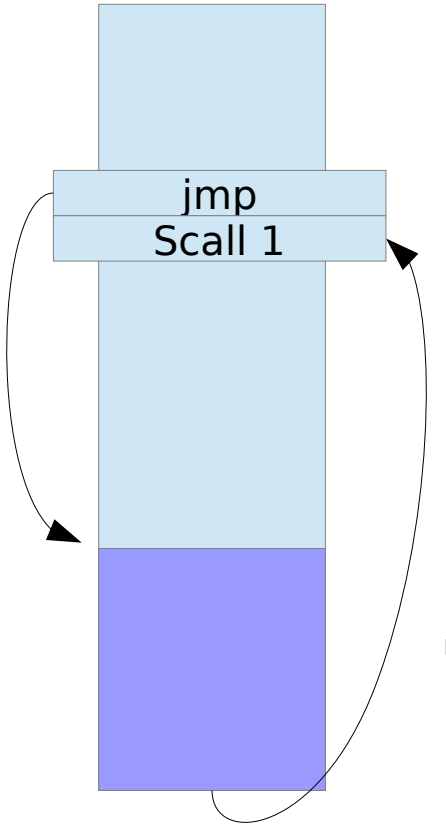
Appels systèmes

Dans MIPS-X, les appels systèmes sont simplifiés et numérotés :

0 : lecture clavier. Résultat dans R1

1 : affichage à l'écran de R1

Appel de fonction..naïf




```
Instruction  0    scall 0 (lecture d'un nombre au clavier)
             4    add r 1, 0, r 2
             8    scall 0 (lecture d'un nombre au clavier)
             12   jmp 100, r 31
             16   scall 1 (écriture d'un nombre à l'écran)
             20   stop
```

r31=16

```
Instruction  100  add r 1, r 2, r 1
             104  div r 1, 2, r 1
             108  jmp r 31, r 0
```

Appel de fonction(s) robuste

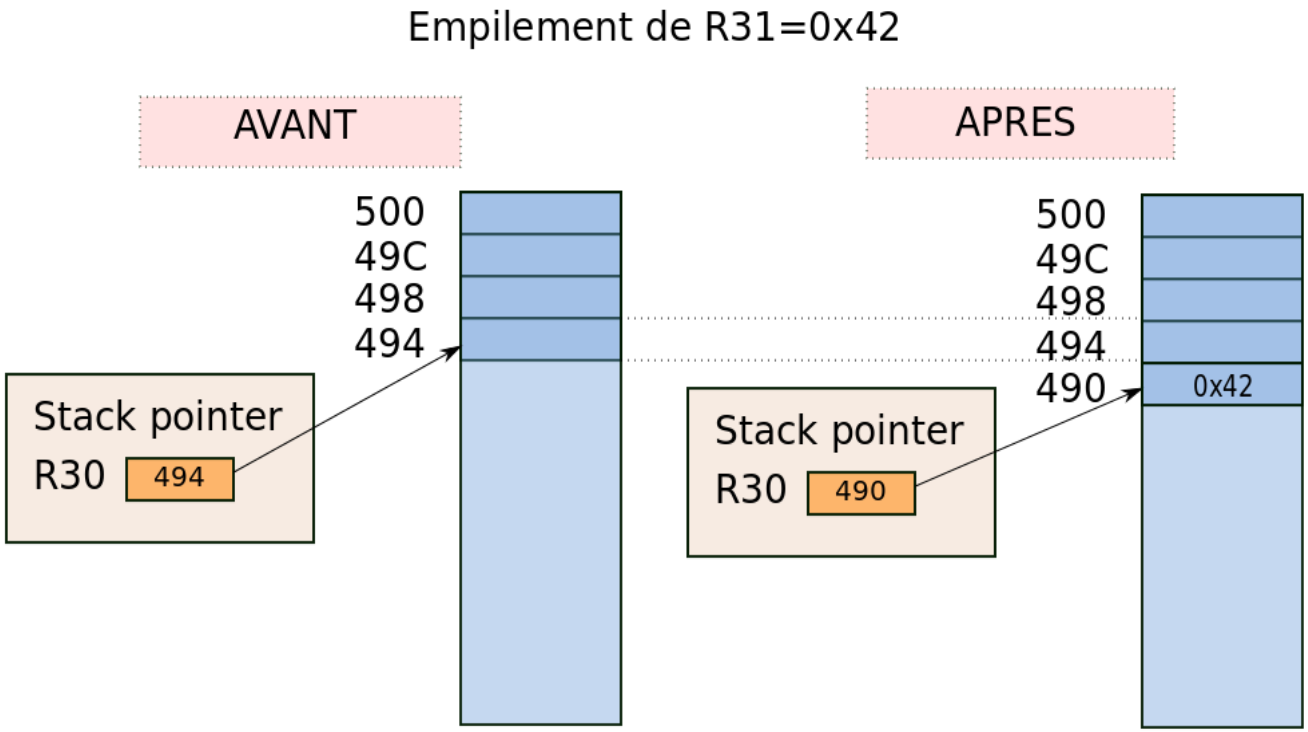
Problème : La convention précédente ne marche pas dans le cas d'appels imbriqués (ex : récursifs).

- ➔ R31 ne peut contenir qu'une seule addresse !

Solution : gérer une pile des adresses de retour

Notion de pile

Généralement la pile commence en haut de la mémoire et croît vers le bas



Appel de fonctions : la pile

Problème : pas de push, ni pop …

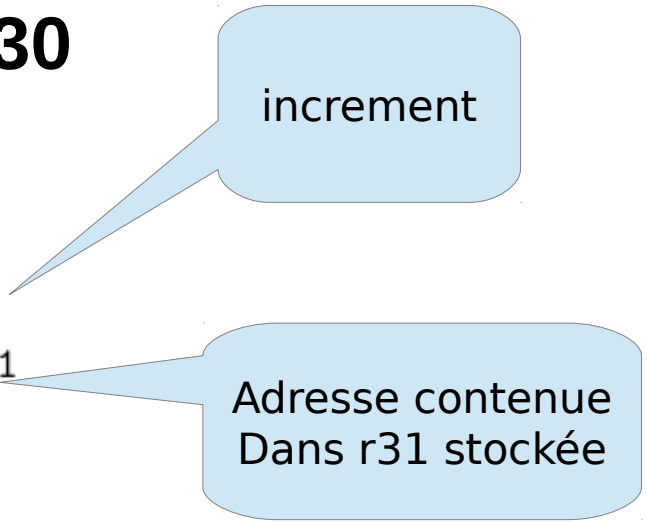
Solution : pointeur de pile dans **r30**

push

```
sub r 30, 4, r 30
store r 30, 0, r 31
```

pop

```
load r 30, 0, r 31
add r 30, 4, r 30
```



Exemple de programme

Calcul de factorielle (récursif)

```
# programme principal
        read
        jmp fact,ra
        write
        stop
# fonction fact(N)
fact:   braz r1,fact0
        sub sp,8,sp
        store sp,0,ra
        store sp,4,r1
        sub r1,1,r1
        jmp fact,ra
        load sp,4,r2
        mult r1,r2,r1
        load sp,0,ra
        add sp,8,sp
        jmp ra,r0
fact0:  add r0,1,r1
        jmp ra,r0
```

```
0  : scall 0
4  : jmp imm 16,31
8  : scall 1
12 : stop

16 : braz 1,60
20 : sub 30,imm 8, 30
24 : store 30,imm 0, 31
28 : store 30,imm 4,1
32 : sub 1,imm 1,1
36 : jmp imm 16,31
40 : load 30,imm 1,2
44 : mult 1,2,1
48 : load 30,imm 0,31
52 : add 30,imm 8,30
56 : jmp 31,0
60 : add 0, imm 1,1
64 : jmp 31,0
```

Un aperçu de l'exécution

```
0  : scall 0
4  : jmp imm 16,31
8  : scall 1
12 : stop

16 : braz 1,60
20 : sub 30,imm 8, 30
24 : store 30,imm 0, 31
28 : store 30,imm 4,1
32 : sub 1,imm 1,1
36 : jmp imm 16,31
40 : load 30,imm 1,2
44 : mult 1,2,1
48 : load 30,imm 0,31
52 : add 30,imm 8,30
56 : jmp 31,0
60 : add 0, imm 1,1
64 : jmp 31,0
```

```
Donnons une valeur au pointeur de pile :
SP=500

0 => R1= 3 (entrée au clavier)
4 => saut à 16, RA(alias R31)=8

16 => R1=3 !=0
20 => sp=492
24 => @(492+0) <= 8
28 => @(492+4) <= R1=3
32 => R1=2
36 => jmp @ 16 , RA=40

    16 => NON
    20 => SP=484
    24 => @(484+0) <= 40
    28 => @(484+4) <= R1=2
    32 => R1=1
    36 => jmp @ 16 , RA=40

        16 => NON
        20 => SP=476
        24 => @(476+0) <= 40
        28 => @(476+4) <= R1=1
        32 => R1=0
        36 => jmp @ 16, RA=40

            16 => OUI! jmp @ 60

On constate que jusqu'à présent
les arguments des fonctions ont été
empilés, ainsi que les adresses de
retour.
```

Pile (stack)

@	data
500	
496	3
492	8
488	2
484	40
480	1
476	40
472	

```
60 => R1=1
64 => jmp @ 40

40 => @(476+4) => R2=1
44 => 1*1=> R1=1
48 => @(476+0) => RA=40
52 => SP=484
56 => jmp @ 40

   40 => @(484+4) => R2=2
   44 => 1*2=> R1=2
   48 => @(484+0) => RA=40
   52 => SP=492
   56 => jmp @ 40

      40 => @(492+4) => R2=3
      44 => 2*3=> R1=6
      48 => @(492+0) => RA=8
      52 => SP=500
      56 => jmp @ 8

8  => write R1 (affiche 6)
12 => stop la machine
```

Programme d'assemblage

partir d'un code assembleur « symbolique »
et générer un code binaire :

- binaire pur[illisible] (gcc,...)
- ascii [lisible] (adapté à un testbench)
- Tableau VHDL (adapté à un testbench)

Programme d'assemblage simple

```
OPCODES={
  :nop   => 0b00000,
  :add   => 0b00001,
  :sub   => 0b00010,
  :mul   => 0b00011,
  :mult  => 0b00011, #syntax facility
  :div   => 0b00100,
  :and   => 0b00101,
  :or    => 0b00110,
  :xor   => 0b00111,
  :shl   => 0b01000,
  :shr   => 0b01001,
  :slt   => 0b01010,
  :sle   => 0b01011,
  :seq   => 0b01100,
  :load  => 0b01101,
  :store => 0b01110,
  :jmp   => 0b01111,
  :braz  => 0b10000,
  :branz => 0b10001,
  :scall => 0b10010,
  :stop  => 0b10011,
}
```

utilise

```
require_relative 'assembler'

prog={
  0 => [:add,:r0, 0,:r2],
  1 => [:add,:r2, 5,:r3],
  2 => [:add,:r2,-5,:r4],
  3 => [:sub,:r4,15,:r3],
}

Assembler.assemble prog, size=256
```

génère

```
-- JC LE LANN
-- ENSTA Bretagne
-- generated : 2014-03-07 10:08:20 +0100
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

use work.type_package.all;

package prog_test is

  constant PROGRAM_TEST : memory_type := (
    0 => x"080000c1",
    1 => x"78000c80",
    2 => x"90400000",
    3 => x"98000000",
    4 => x"00000000",
    5 => x"00000000",
    6 => x"00000000",
    7 => x"00000000",
    8 => x"00000000",
    9 => x"00000000",
    10 => x"00000000",
```

VHDL

(ici les label ne sont pas gérés)

Notion d'interruptions

Def : (« it ») événements asynchrones, inoppinés, qui invitent le processeur à se détourner de son flot de contrôle principal, afin de gérer un traitement auxiliaire, puis à reprendre ce flot là où il en était.

Pas gérée dans notre MIPS-0.

Indispensable en réalité !

Alternative au « polling » :

> "Polling is like picking up your phone every few seconds to see if you have a call. Interrupts are like waiting for the phone to ring."

Notion d'interruptions

- Le processeur reçoit un signal, lui indiquant qu'une it est survenue.

- Le processeur peut interroger le contrôleur d'interruptions afin de connaître la source de cette interruption. Connaissant cette source (son numéro), il sait alors vers quelle adresse sauter (« vecteur » d'interruptions).

- Avant d'y aller effectivement, le processeur doit sauvegarder le contexte d'exécution : ensemble des valeurs des registres de travail, au moment de l'interruption.

- Après traitement de l'interruption, le processeur restore ce contexte et continue...comme si rien ne s'était passé.