Le processeur NIOS II

Alain MÉRIGOT

Université Paris Saclay

Le processeur NIOS II

Processeur 32 bits

Présent dans les FPGA Altera/Intel

Générable sous différentes microarchitectures

Jeu d'instruction RISC simple (très proche de MIPS)

Les accès mémoire données et instructions sont strictement alignés.

L'arrangement des données est *petit boutiste* (*little endian*).

Le processeur NIOS II (cont.)

Le NIOS comprend 32 registres de 32 bits nommés `r0` à `r31`

Certains registres ont un rôle spécifique imposé par le matériel.

- `r0` est toujours à 0 et toute écriture vers `r0` n'a aucune action (`nop`).
- `r31` contiendra l'adresse de retour des appels de procédure

D'autres registres ont un rôle imposé par des conventions liées à la chaine logicielle (assembleur, compilateur, éditeur de lien, chargeur, système d'exploitation) et il faut respecter ces conventions pour assurer un bon fonctionnement des programmes.

Jeu d'instructions

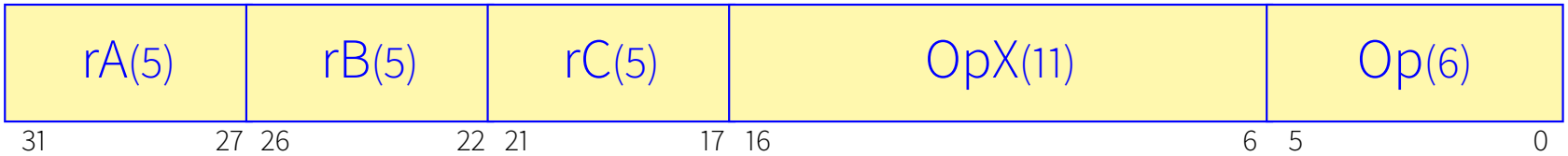
Jeux d'instruction *chargement-rangement*

- **Instructions unité arithmétique et logique** réalisent un calcul entre registres et/ou opérandes immédiats
- **Instructions d'accès mémoire** transfert entre registres et mémoire avec adressage basé
- **Instruction de contrôle de flot** branchement conditionnels, sauts, appels de procédure

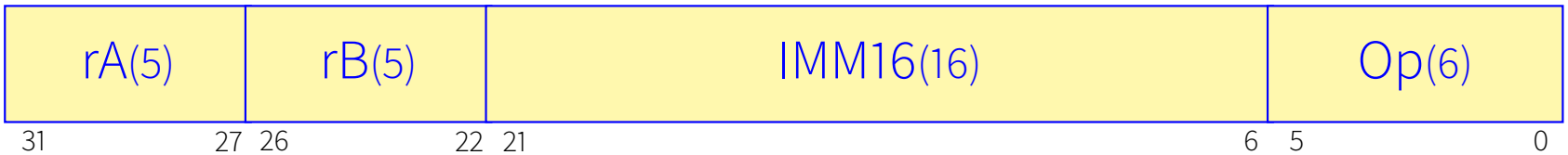
Formats des instructions

Toutes les instructions sont codées sur 32 bits et sont alignées.
3 formats :

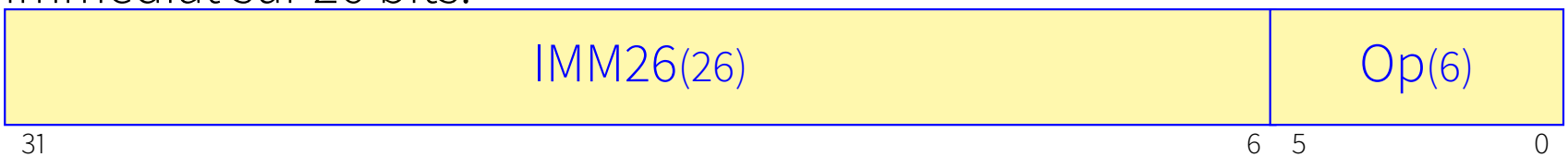
Type R (**r**egistre): décrit des instructions n'ayant que des registres comme opérandes



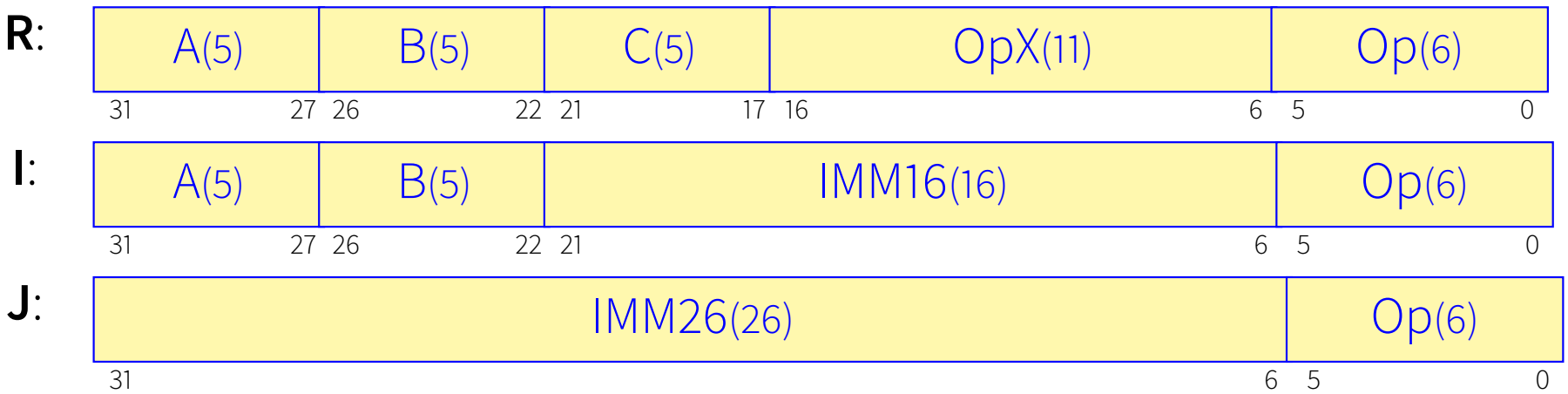
Type I (**i**mmédiat): instructions de calcul ou d'accès mémoire ayant un immédiat sur 16 bits comme opérande



Type J (***j**ump*): utilisé uniquement pour les sauts/appels avec opérande immédiat sur 26 bits.



Formats des instructions (cont.)



Les 6 bits de poids faible (champs `Op`) représentent le *code opération* qui indique le type d'instruction.

Le code `Op=0x3a` indique un format `R`. Le type d'instruction (`add`, `and`, etc) est alors codé par les 11 bits du champ `OpX`.

Les champs `A`, `B` et `C` donnent un numéro de registre sur 5 bits.

Notations :

Imm_n	immédiat sur n bits
$(\mathtt{X})_{n..m}$	bits $n..m$ de X
`rA, rB, rC`	registres 32 bits
`X:Y`	concaténation de X et Y
$\sigma(\mathtt{X})$	extension signée de `X`
`%lo(X)`	16 bits de poids faibles de `X`
`%hi(X)`	16 bits de poids forts de `X`
$\mathrm{Mem}_n[\mathtt{X}]$	n bits à l'adresse mémoire `X`
(signed)`rX`	interprétation signée de `rX`
(unsigned)`rX`	interprétation non signée de `rX`
`pc`	*program counter* ou compteur de programme

Instructions arithmétiques et logiques

Instructions arithméthiques :

add	`add`	`rC,rA,rB`	`rC`$\leftarrow$ `rA`+ `rB`	[R]
addi	`addi`	`rB,rA,Imm16`	`rB`$\leftarrow$ `rA`+ σ(`Imm`$_{16}$)	[I]
sub	`sub`	`rC,rA,rB`	`rC`$\leftarrow$ `rA`$-$ `rB`	[R]
subi	`subi`	`rB,rA,Imm16`	`rB`$\leftarrow$ `rA`$-$ σ(`Imm`$_{16}$)	[I]

Copies de données :

mov	`mov`	`rB,rA`	`rB`$\leftarrow$ `rA`	[M]
			(== `add rB,rA,r0`)	
movi	`movi`	`rB,imm16`	`rB`$\leftarrow$ 0x0000:`Imm`$_{16}$	[M]
			(== `addi rB,r0,Imm16`)	
movhi	`movhi`	`rB,imm16`	`rB`$\leftarrow$ `Imm`$_{16}$:0x0000	[M]
			(== `orhi rB,r0,Imm16`)	
movia	`movia`	`rB,label`	`rB`$\leftarrow$ adr(label)	[M]
			(== `orhi rB,r0, %hi(label)`)	
			(+ `addi rB,r0, %lo(label)`)	

Instructions arithmétiques et logiques (cont.)

Opérations logiques

nor	nor	rC,rA,rB	rC←~(rA|rB)	[R]
			nor rB,rA,r0≡rB←~rA	
and	and	rC,rA,rB	rC←rA&rB	[R]
andi	andi	rB,rA,imm16	rB←rA&(0x0000:Imm_{16})	[I]
andhi	andhi	rB,rA,imm16	rB←rA&(Imm_{16}:0x0000)	[I]
or	or	rC,rA,rB	rC←rA|rB	[R]
ori	ori	rB,rA,imm16	rB←rA|(0x0000:Imm_{16})	[I]
orhi	orhi	rB,rA,imm16	rB←rA|(Imm_{16}:0x0000)	[I]
xor	xor	rC,rA,rB	rC←rA^rB	[R]
xori	xori	rB,rA,imm16	rB←rA^(0x0000:Imm_{16})	[I]
xorhi	xorhi	rB,rA,imm16	rB←rA^(Imm_{16}:0x0000)	[I]

Instructions arithmétiques et logiques (cont.)

Comparaisons

cmpeq	cmpeq rC,rA,rB	rC←(rA== rB? 1 : 0)	[R]
cmpeqi	cmpeqi rB,rA,imm16	rB←(rA== σ(Imm$_{16}$) ? 1 : 0)	[I]
cmpne	cmpne rC,rA,rB	rC←(rA!= rB? 1 : 0)	[R]
cmpnei	cmpnei rB,rA,imm16	rB←(rA!= σ(Imm$_{16}$) ? 1 : 0)	[I]
cmpge	cmpge rC,rA,rB	rC←((signed)rA>= (signed)rB? 1 : 0)	[R]
cmpgei	cmpgei rB,rA,imm16	rB←((signed)rA>= σ(Imm$_{16}$) ? 1 : 0)	[I]
cmpgeu	cmpgeu rC,rA,rB	rC←((unsigned)rA>= (unsigned)rB? 1 : 0)	[R]
cmpgeui	cmpgeui rB,rA,imm16	rB←((unsigned)rA>= (0x0000:Imm$_{16}$) ? 1 : 0)	[I]

cmpgt (et **cmpgti**, **cmpgtu**, **cmpgtui**) pour l'opérateur $>$ [M]

cmple (et **cmplei**, **cmpleu**, **cmpleui**) pour l'opérateur $<=$ [M]

cmplt (et **cmplti**, **cmpltu**, **cmpltui**) pour l'opérateur $<$ [R]

Instructions arithmétiques et logiques (cont.)

Multiplication/division

mul	mul rC,rA,rB	$\texttt{rC} \leftarrow (\texttt{rA} \times \texttt{rB})_{31..0}$	[R]
muli	muli rB,rA,imm32	$\texttt{rB} \leftarrow (\texttt{rA} \times \sigma(\texttt{Imm}_{16}))_{31..0}$	[I]
mulxss	mulxss rC,rA,rB	$\texttt{rC} \leftarrow ((\text{signed})\texttt{rA} \times (\text{signed})\texttt{rB})_{63..32}$	[R]
mulxsu	mulxsu rC,rA,rB	$\texttt{rC} \leftarrow ((\text{signed})\texttt{rA} \times (\text{unsigned})\texttt{rB})_{63..32}$	[R]
mulxuu	mulxuu rC,rA,rB	$\texttt{rC} \leftarrow ((\text{unsigned})\texttt{rA} \times (\text{unsigned})\texttt{rB})_{63..32}$	[R]
div	div rC,rA,rB	$\texttt{rC} \leftarrow ((\text{signed})\texttt{rA} \div (\text{signed})\texttt{rB})_{31..0}$	[R]
divu	divu rC,rA,rB	$\texttt{rC} \leftarrow ((\text{unsigned})\texttt{rA} \div (\text{unsigned})\texttt{rB})_{31..0}$	[R]

Le reste de la division doit être calculé par $R = Dd - Q \times Dv$

```
div rC, rA, rB
mul rD, rC, rB
sub rD, rA, rD ; rD == rA % rB
```

Instructions arithmétiques et logiques (cont.)

Rotations/décalages

rol	rol	rC,rA,rB	$rC \leftarrow (rA_{(31-rB_{4..0})..0} : rA_{31..(32-rB_{4..0})})$ rotation gauche de $(rB_{4..0})$ bits	[R]
roli	roli	rC,rA,imm5	$rC \leftarrow (rA_{(31-Imm_5)..0} : rA_{31..(32-Imm_5)})$	[I]
ror	ror	rC,rA,rB	$rC \leftarrow (rA_{(rB_{4..0}-1..0)} : rA_{31..(rB_{4..0})})$ rotation droite de $(rB_{4..0})$ bits	[R]
sll	sll	rC,rA,rB	$rC \leftarrow (rA << (rB_{4..0}))$	[R]
slli	slli	rC,rA,imm5	$rC \leftarrow (rA << (Imm_5))$	[I]
srl	srl	rC,rA,rB	$rC \leftarrow ((\text{unsigned})rA >> (rB_{4..0}))$	[R]
srli	srli	rC,rA,imm5	$rC \leftarrow ((\text{unsigned})rA >> (Imm_5))$	[I]
sra	sra	rC,rA,rB	$rC \leftarrow ((\text{signed})rA >> (rB_{4..0}))$	[R]
srai	srai	rC,rA,imm5	$rC \leftarrow ((\text{signed})rA >> (Imm_5))$	[I]

`srl` fait un décalage *logique* avec des 0 en poids fort,
`sra` fait un décalage *arithmétique* avec extension de signe.

Accès mémoire

Chargements (`ld`) et rangements (`st`)
sur des octets (***b****yte*), des demi-mots (***h****alf*), ou des mots (***w****ord*).

ldb	`ldb rB,imm16(rA)`	$rB \leftarrow \sigma(Mem_8[rA + \sigma(Imm_{16})])$	[I]
ldbu	`ldbu rB,imm16(rA)`	$rB \leftarrow 0x000000:Mem_8[rA + \sigma(Imm_{16})]$	[I]
ldh	`ldh rB,imm16(rA)`	$rB \leftarrow \sigma(Mem_{16}[rA + \sigma(Imm_{16})])$	[I]
ldhu	`ldhu rB,imm16(rA)`	$rB \leftarrow 0x0000:Mem_{16}[rA + \sigma(Imm_{16})]$	[I]
ldw	`ldw rB,imm16(rA)`	$rB \leftarrow Mem_{32}[rA + \sigma(Imm_{16})]$	[I]
stb	`stb rB,imm16(rA)`	$Mem_8[rA + \sigma(Imm_{16})] \leftarrow rB_{7..0}$	[I]
sth	`sth rB,imm16(rA)`	$Mem_{16}[rA + \sigma(Imm_{16})] \leftarrow rB_{15..0}$	[I]
stw	`stw rB,imm16(rA)`	$Mem_{32}[rA + \sigma(Imm_{16})] \leftarrow rB$	[I]

Toutes les instructions `ld` et `st` existent en version entrée-sortie (`io`), qui évitent la mémoire cache (`ldbio, ldbuio`, etc).
Ceci permet d'accéder à des périphériques.

Instructions de contrôle de flot

Branchements

br	br	imm16	pc← pc+4+σ(Imm$_{16}$)	[I]
beq	beq	rA,rB,imm16	si (rA== rB)	[I]
			alors pc← pc+4+σ(Imm$_{16}$)	
			sinon pc← pc+4	
bne	bne	rA,rB,imm16	idem si (rA!= rB)	[I]
bge	bge	rA,rB,imm16	idem si ((signed)rA>= (signed)rB)	[I]
bgeu	bgeu	rA,rB,imm16	idem si ((unsig.)rA>= (unsig.)rB)	[I]
bgt	bgt	rA,rB,imm16	idem si ((signed)rA> (signed)rB)	[M]
bgtu	bgtu	rA,rB,imm16	idem si ((unsig.)rA> (unsig.)rB)	[M]
ble	ble	rA,rB,imm16	idem si ((signed)rA<= (signed)rB)	[M]
bleu	bleu	rA,rB,imm16	idem si ((unsig.)rA<= (unsig.)rB)	[M]
blt	blt	rA,rB,imm16	idem si ((signed)rA< (signed)rB)	[I]
bltu	bltu	rA,rB,imm16	idem si ((unsig.)rA< (unsig.)rB)	[I]

Instructions de contrôle de flot (cont.)

Sauts/appels de procédure

jmpi	`jmpi imm26`	$pc \leftarrow (pc_{31..28} : Imm_{26} \times 4)$	[J]
jmp	`jmp rA`	$pc \leftarrow rA$	[R]
call	`call imm26`	$ra \leftarrow pc + 4$ //r31 $pc \leftarrow (pc_{31..28} : Imm_{26} \times 4)$	[J]
callr	`callr rA`	$ra \leftarrow pc + 4$ //r31 $pc \leftarrow rA$	[R]
ret	`ret`	$pc \leftarrow ra$ //r31	[R]

Opérations flottantes

On peut ajouter des extensions pour le calcul flottant, uniquement en simple précision.
Les nombres flottants sont stockés dans les mêmes registres que les entiers[1].
Les instructions sont de type R.
Le support flottant comprend :

- des instructions arithmétiques et logiques : `fsubs fadds fmuls fdivs fsqrts`
- des conversions entier-flottant : `floatis fixsi round`
- des opérations simples : `fmins fmaxs fnegs fabss`
- des comparaisons : `fcmplts fcmples fcmpgts fcmpges fcmpeqs`[2] `fcmpnes`[2]

[1] contrairement à la plupart des processeurs
[2] Attention aux erreurs d'arrondi dans les comparaisons flottantes d'égalité/inégalité

Les registres Nios II

`r0`	`zero`	toujours à 0
`r1`	`at`	*assembler temporary* (utilisé par certaines macros)
`r2-r3`		valeur de retour des procédures
`r4-r7`		arguments des procédures
`r8-r15`		registres volatiles (non préservés)
`r16-r23`		registres non volatiles (à sauvegarder par la procédure appelée avant usage)
`r24`	`et`	*exception temporary*
`r25`	`bt`	*breakpoint temporary*
`r26`	`gp`	*global pointer*
`r27`	`sp`	*stack pointer*
`r28`	`fp`	*frame pointer*
`r29`	`ea`	*exception return address*
`r30`	`sstatus`	*status register/breakpoint return address*
`r31`	`ra`	*return address*

Ecriture d'un programme assembleur

Un programme assembleur comprend des **instructions** et des **directives** permettant notamment de définir des données.
Principales directives :

```
.data              # début d'un segment de données
x: .word 10        ; définit une variable de 4 octets x
                   ; et lui affecte la valeur 10 (ou 0xa)
                   ; idem pour .byte(1o), .half(2o), .dword(8o)
                   ; les données sont alignées
y: .space 16       # 16 octets non initialisés à l'adresse y
z: .ascii "hello, world!"  # Une chaine de caractères à l'adresse z
                   # Pour une chaîne C standard (terminée par un '\0'),
                   # utiliser .asciz
.equ N,12          # N sera remplacé par 12
.include "file.asm"  # inclut un fichier
.text              # début des instructions
...
.end               # fin du programme. Ignore le reste du fichier
```

Exemples de programmes assembleur

Produit scalaire de deux vecteurs

```
int v1[2]={10, -5};
int v2[2]={-3, 8};
int ps;
....
ps =
     v1[0]*v2[0]
      +v1[1]*v2[1];
...
```

```
.data
v1: .word 10,-5
v2: .word -3,8
ps: .space 4
.text
movia r8,v1     #r8=&v1[0]
ldw r9,0(r8)    #r9=v1[0]
ldw r10,4(r8)   #r10=v1[1]
movia r8,v2     #r8=&v2[0]
ldw r11,0(r8)   #r11=v2[0]
ldw r12,4(r8)   #r12=v2[1]
mul r9, r9,r11 #v1[0]*v2[0]
mul r10,r10,r12#v1[1]*v2[1]
add r9, r9,r10 #r9=ps
movia r8,ps     #r8=&ps
stw r9,0(r8)    #write ps
```

Exemples de programmes assembleur (cont.)

Mise en oeuvre de tests: maximum de deux nombres

```
      ....
int a,b, max;

if (a>b)
  max = a;
else
  max = b;
 ...
```

```
  # on suppose a->r8, b->r9,
  #     max->r10
  ble r8, r9, else#a<=b?->else
  mov r10, r8 # max <= x
  br fin
else:
  mov r10, r9 # max <= y
fin: ...
```

ou (en plus optimisé)

```
  mov r10, r8
  bgt r8,  r9, fin
  mov r10, r9
fin: ...
```

Exemples de programmes assembleur (cont.)

Mise en oeuvre de boucles.

```
for(initialisation; condition; increment) {
  corps_de_boucle;
}
```

Mise en oeuvre directe

```
        initialisation
bcle:   test condition
        si faux -> fin
        corps_de_boucle
        increment
        -> bcle (incond.)
fin:    ...
```

Version optimisée (1 seul brcht/boucle) forme « *do .. while()* »

```
        initialisation
        -> cond (incond.)
debut:  corps_de_boucle
        increment
cond:   test condition
        si vrai -> debut
fin:    ...
```

si il y au moins une itération, le premier saut peut être supprimé

Exemples de programmes assembleur (cont.)

Boucle : mise à zéro d'un tableau

```
int N;
short t[N];
for(int i=0; i<N; i++)
  t[i]=0;
```

```
// équivalent à :
int N;
short t[N], *pt;
for(int i=0, pt=&t[0];
          i<N;
          i++,pt++)
  *pt=0;
```

```
  # on suppose @t->r8, N->r9
### initialisation boucle
  mov  r10,r8 #r10=@t=&t[0]
  mov  r11,r0   #i=0
boucle:
### corps de la boucle
  sth  r0,0(r10) #*t=0
### incrément de la boucle
  addi r11,r11,1 #i++
  addi r10,r10,2 #t++(@t+=2)
  # 2==sizeof(short)
### test condition
  blt  r11,r9,boucle
       #si i<N -> boucle
fin: ...
```

Exemples de programmes assembleur (cont.)

Boucle : ondelette de Haar

```
int N;
int t[N], v[N];

for(int i=0;i<N;i++){
  if (i%1) // i impair
    t[i]=v[i-1]-v[i];
  else     // i pair
    t[i]=v[i]+v[i+1];
}
```

```
# @t->r13 @v->r14 N->r15
    mov  r16,r13  #r16=@t=&t[0]
    mov  r17,r14  #r17=@v=&v[0]
    mov  r8,r0    #i=0
bcl:ldw  r9,0(r17)#r9=v[i]
    andi r11,r8,0x01
    beq  r11,r0, pair
    ldw  r10,-4(r17)#r10=v[i-1]
    sub  r9,r10,r9#v[i-1]-v[i]
    br   cnt
pair:ldw  r10,4(r17)#r10=v[i+1]
    add  r9,r9,r10#v[i]+v[i+1]
cnt:stw  r9,0(r16)#range t[i]
    addi r8,r8,1  #i++
    addi r16,r16,4#t++ (@t+=4)
    addi r17,r17,4#v++ (@v+=4)
    blt  r8,r15,bcl # i<N?
fin: ...
```

Exemples de programmes assembleur (cont.)

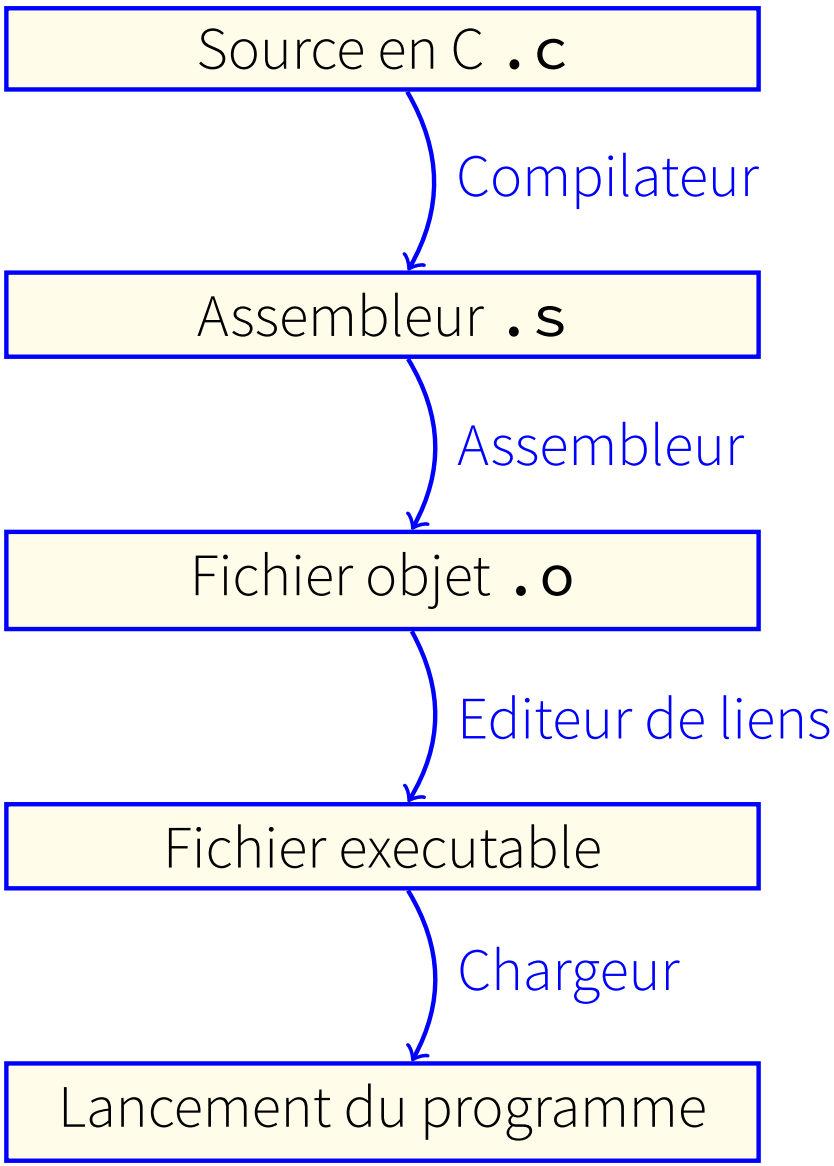
Boucle : ondelette de Haar (version optimisée)

```
int N;
int t[N], v[N];

for(int i=0;i<N;i+=2){
  t[i]=v[i]+v[i+1];
  t[i+1]=v[i]-v[i+1];
}
```

```
# @t->r13 @v->r14 N->r15
  mov  r16,r13    #r16=@t=&t[0]
  mov  r17,r14    #r17=@v=&v[0]
  mov  r8,r0      #i=0
boucle:
  ldw  r9,0(r17) #v[i]
  ldw  r10,4(r17)#v[i+1]
  add  r11,r9,r10#v[i]+v[i+1]
  stw  r11,0(r16)#t[i]
  sub  r11,r9,r10#v[i]-v[i+1]
  stw  r11,4(r16)#t[i+1]
  addi r8,r8,2    #i+=2
  addi r16,r16,8 #t+=2 (@t+=8)
  addi r17,r17,8 #v+=2 (@v+=8)
  blt  r8,r15,boucle#i<N?
fin: ...
```

Processus de développement logiciel



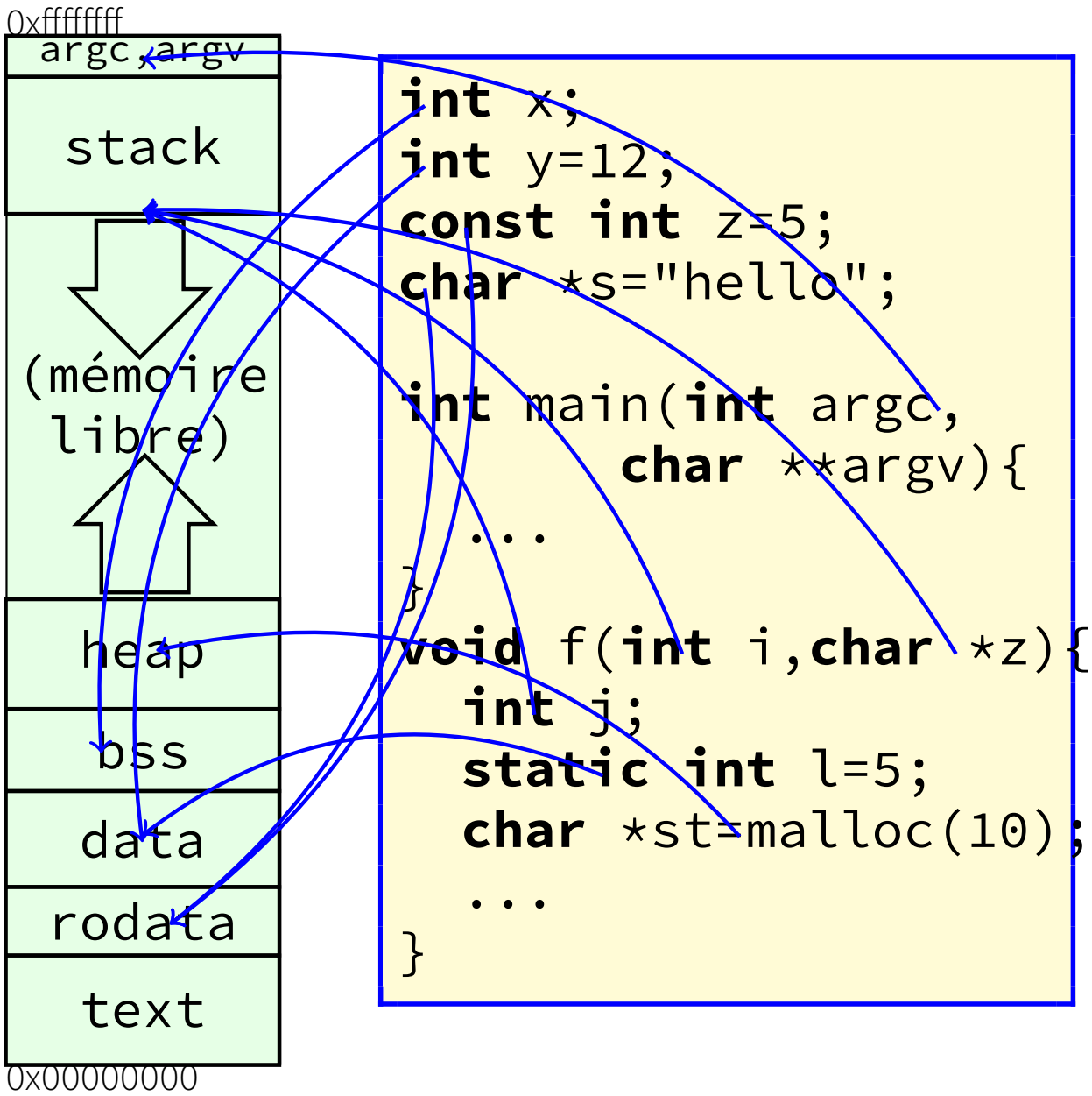
Un **compilateur** va traduire en assembleur un fichier C

L'**assembleur** convertit le programme en fichier objet. Il contient le code binaire des instructions et des symboles donnant les noms des variables, fonctions, etc

L'**éditeur de lien** (ou *linker*) rassemble les fichiers objets et les bibliothèques. Il positionne les variables globales et statiques en mémoire, ainsi que les fonctions, et modifie le code pour générer les adresses correspondantes.

Le **chargeur** (ou *loader*) demande des ressources au système d'exploitation. Il positionne divers registres et tables et lance l'exécution du programme.

Segments mémoire d'un exécutable



Segments mémoire (créés par le *loader*)

pile (`stack`) gestion des appels de fonctions

tas (`heap`) allocations dynamiques (`malloc()`)

bss données non initialisées (mises à 0 par le *runtime* avant le `main()`)

data données initialisées (dans l'exécutable et copiées par le *loader*)

rodata données constantes

text (`code`) instructions du programme

Appels de procédure

Lors d'un appel de procédure, l'adresse de retour est sauvegardée dans `ra` (`r31`).
Mais il est nécessaire de gérer une *pile* pour :

- sauvegarder les adresses de retours lors d'appels imbriqués de fonctions
- accéder aux paramètres des fonctions
- mettre les variables locales qui ne tiennent pas dans des registres
- sauvegarder des registres non volatiles avant de les modifier

La pile est gérée au moyen de deux pointeurs :

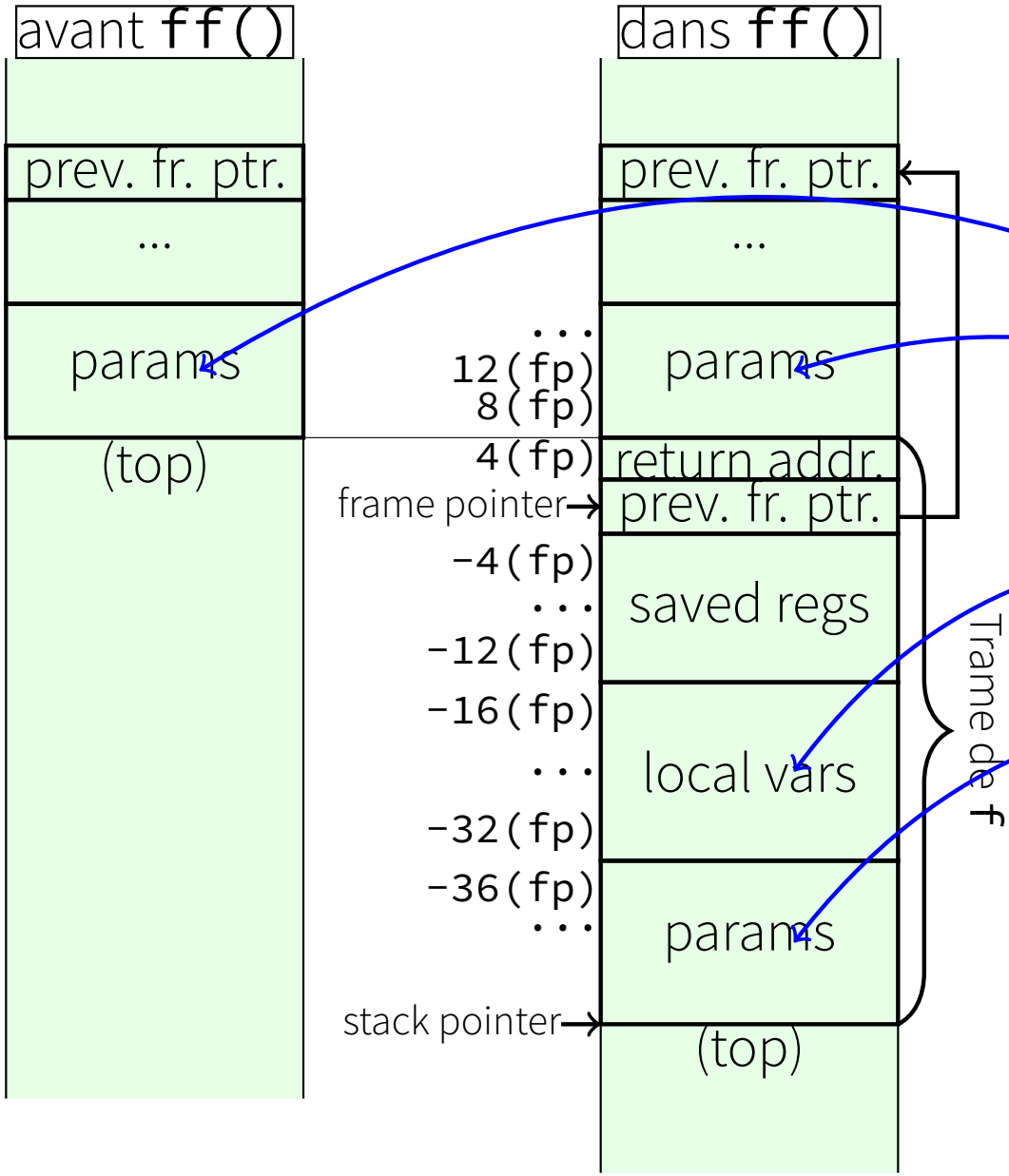
- le **pointeur de pile** (*stack pointer*) (`sp` ou `r27`) qui pointe sur la valeur courante du haut de pile
- le **pointeur de trame** (*frame pointer*) (`fp` ou `r28`) qui permet de revenir à la trame précédente (utile pour le *debug*)

Appels de procédure (cont.)

Pour assurer la compatibilité des différentes fonctions, spécification de **conventions d'appels** (*calling conventions*).

- En début de fonction, on réserve un espace suffisant dans la pile pour la trame de la fonction et les arguments des fonctions appelées. Cet espace sera libéré en fin de fonction.
- les 4 premiers arguments d'une fonction sont passés dans les registres `r4–r7`. Les arguments suivants sont dans la pile.
- la valeur de retour d'une fonction est dans le registre `r2` (et `r3` si 64 bits sont nécessaires).
- les registres `r8–r15` peuvent être utilisé sans contrainte et doivent être sauvegardés par la fonction appelante si nécessaire (registres *volatiles*)
- les registres `r16–r23` ne peuvent être utilisés par une fonction que s'ils sont préalablement sauvegardés (registres *non volatiles*).

Appels de procédure (cont.)




```
int g(int a, int b
  int x, int y, int z);
int h() {
  ...
  ff(1,2);
  ...
}
int ff(int u,int v){
  int e,f;
  e=2*u;
  f=2*v;
  f=g(f,u,e,f,5);
  return f;
}
```

Accès aux paramètres et variables locales par `fp` (ou `sp`) + un décalage.

Appels de procédure (cont.)

En début de fonction:

- réserver un espace suffisant dans la pile pour la trame de la fonction
- empiler `ra` si la fonction appelle une autre fonction (fonction *non terminale*)
- empiler `fp`
- `fp` = adr(`fp`)
- si nécessaire, sauvegarder les registres non volatiles (`r16-r23`) que l'on va utiliser

```
int ff(int u,int v){
  int e,f;
  x=2*u;
  y=2*v;
  f=g(5,u,v,x,y);
  return f;
}
```

```
ff:# début fonction ff()
   addi sp,sp,-24# alloc. trame
   stw  ra,20(sp)#sauv. adr. ret
   stw  fp,16(sp)#sv. fp. prec.
   addi fp,sp,16 # nouveau fp
   stw  r16,12(sp) # sauv. r16
     # ou    stw  r16,-4(fp)
   stw  r17,8(sp)  # sauv. r17
     # ou stw  r17,-8(fp)
   ...
```

Appels de procédure (cont.)

corps de la fonction

- `u` est dans `r4` et `v` dans `r5`[1]
- on met `x` dans `r16` et `y` dans `r17`
- calcul de `x` et `y`
- (si nécessaire, on sauvegarde les registres volatiles `r8`–`r15`)[1]
- on copie les paramètres[1] :
  - 5→`r4`
  - u→`r5`
  - v→`r6`
  - x→`r7`
  - y→pile
- appel `g()`

[2] conformément aux conventions d'appel

```
int ff(int u,int v){
  int e,f;
  x=2*u;
  y=2*v;
  f=g(5,u,v,x,y);
  return f;
}
```

```
ff:# suite fonction ff()
   slli r16,r4,1 #2*u
   slli r17,r5,1 #2*v
   mov  r6,r5
   mov  r5,r4
   addi r4,r0,5
   mov  r7,r16
   stw  r17,4(sp)
   call g
   ...
```

Appels de procédure (cont.)

fin de la fonction `ff()`

- la valeur de retour de `g()` est dans `r2`[1] et on la copie dans `f` (`r17`)
- on copie `f` de `r17` vers le registre `r2` (valeur de retour de la fonction[1]) (largement optimisable!)
- on restaure les registres `ra` et `fp`[1]
- on libère la trame
- retour de la fonction

NB : l'utilisation d'un *frame pointer* est optionnelle, mais `fp` permet d'utiliser un *debugger*.

```
int ff(int u,int v){
  int x,y;
  x=2*u;
  y=2*v;
  f=g(5,u,v,x,y);
  return f;
}
```

```
ff:# fin fonction ff()
   mov  r17,r2
   mov  r2, r17
   ldw  ra, 20(sp)
   ldw  fp, 16(sp)
   addi sp,sp,24
   ret
```

[2] conformément aux conventions d'appel

Appels de procédure (cont.)

Noter que si :

- la fonction est terminale
- on n'utilise pas de *frame pointer*
- on n'utilise que des registres volatiles

il est inutile de gérer une pile.
On peut toujours utiliser `sp` pour récupérer des arguments de la fonction, si nécessaire.

```
int g(int a, int b,
      int x, int y, int z)
{
  return a+b+x+y+z;
}
```

```
g:
    add   r2,r4,r5 # r2=a+b
    add   r2,r2,r6 # r2+=x
    add   r2,r2,r7 # r2+=y
    # z est dans la pile
    # (5e argument de g)
    # Noter que sp n'a pas bougé
    # et z est toujours en sp+4
    ldw   r8, 4(sp)# r8=z
    add   r2,r2,r8 # r2+=z
    ret
```