
Jeux d'instructions

Daniel Etiemble
de@lri.fr

Les jeux d'instructions

- Ensemble des instructions d'un processeur
- Format d'instructions
 - Lié au modèle (n,m)
 - Longueur fixe ou longueur variable
 - Accès aux données
- Un jeu d'instructions RISC simple
 - NIOS II (processeur « logiciel » d'Altera)
- Les grandes caractéristiques des jeux d'instructions
 - Branchements
 - Modes d'adressage
 - Appel de fonctions

Jeux d'instructions

- Des objectifs différents selon les classes d'applications
 - Vitesse maximale (PC, serveurs)
 - Taille de code minimale (embarqué)
 - Consommation
 - essentiel pour embarqué
 - important pour tous
- Taille des instructions
 - Fixe
 - Variable
- Modèles d'exécution

Les objectifs

- Performance
 - Pipeline efficace
 - Instructions de longueur fixe
 - Décodage simple
 - Modes d'adressage simples
 - Utiliser le parallélisme d'instructions
- Taille du code
 - Minimiser la taille des instructions
 - Instructions de longueur variable (ou fixe)
 - Accès aux données efficace
 - Modes d'adressage complexes et efficaces pour applications visées
- Compatibilité binaire avec les générations précédentes
 - Fondamental pour processeurs « généralistes » : Exemple IA-32 (x86)
 - Moins important pour processeurs plus spécialisés (embarqué, traitement du signal)

Modèles d'exécution

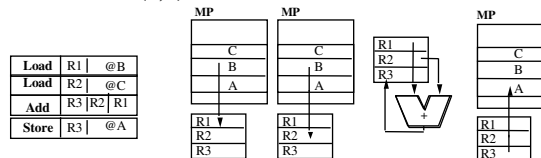
- Modèles d'exécution (n,m)
 - n : nombre d'opérandes par instruction
 - m : nombre d'opérandes mémoire par instruction
- Les modes principaux
 - RISC : (3,0)
 - Instructions de longueur fixe
 - Load et Store : seules instructions mémoire
 - IA-32 : (2,1)
 - Instructions de longueur variable

Modèle d'exécution RISC

(n,m)

n : nombre d'opérandes par instruction
 m : nombre d'opérandes mémoire par instruction
 Ex : A := B + C

LOAD-STORE (3,0)



Instructions de longueur fixe
 Seules les instructions Load et Store accèdent à la mémoire

Registres: organisation RISC

- 32 registres généraux (entiers) R0 à R31
- 32 registres flottants
- Instructions UAL et mémoire
 - Registre – registre
 - $R_d \leftarrow R_{s1} \text{ op } R_{s2}$
 - Registre – immédiat
 - $R_d \leftarrow R_{s1} \text{ op } \text{immédiat}$
 - $R_d \leftrightarrow \text{Mémoire } (R_{s1} + \text{dépl.})$

R0	alwayszero	local var A	R16
R1		local var B	R17
R2		local var C	R18
R3		local var D	R19
R4		local var E	R20
R5			R21
R6			R22
R7			R23
R8	compiler temp 1		R24
R9	compiler temp 2		R25
R10	compiler temp 3		R26
R11	compiler temp 4		R27
R12			R28
R13		stack pointer	R29
R14			R30
R15			R31

Typical RISC Processor

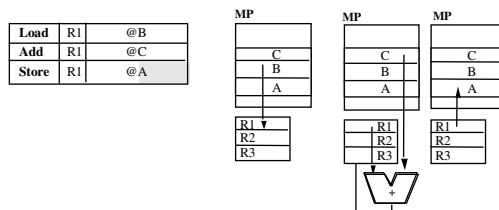
Modèle (2,1)

(n,m)

n : nombre d'opérandes par instruction
m : nombre d'opérandes mémoire par instruction

Ex : $A := B + C$

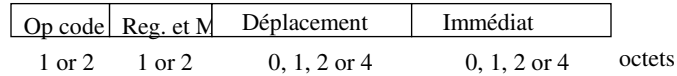
REGISTRE-MEMOIRE (2,1)



CISC compatible avec la technologie MOS des années 75-80

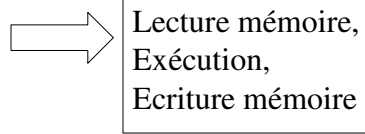
Caractéristiques IA-32

- Instructions de longueur variable



- Inst dest, source

REG REG
REG MEM
REG IMM
MEM REG
MEM IMM

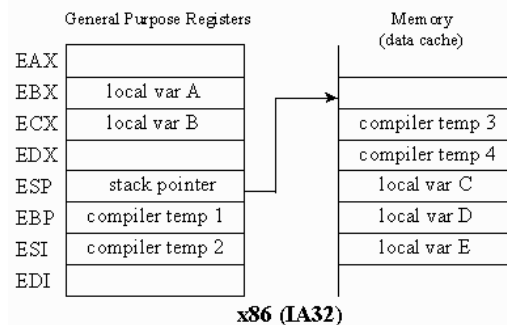


- Instructions complexes
 - Rep
- Modes d'adressage complexes

$$\text{Adresse mémoire} = \text{Rb} + \text{RI} \times \text{f} + \text{déplacement}$$

Registres : organisation IA-32

- Organisation non homogène
 - 8 registres «généraux» avec rôle spécifique
 - Registres flottants fonctionnant en pile (x87)
 - Registres «SIMD» (MMX, SSE-SSE2-SSE3)

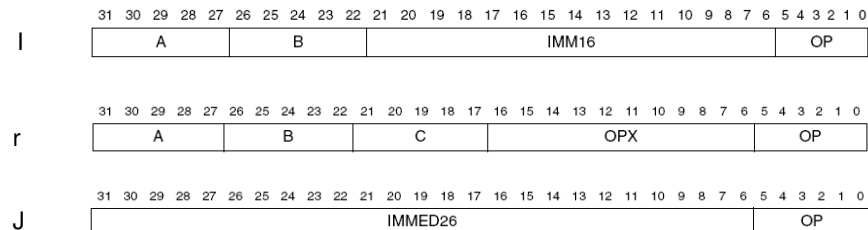


Le débat RISC-CISC pour les PC

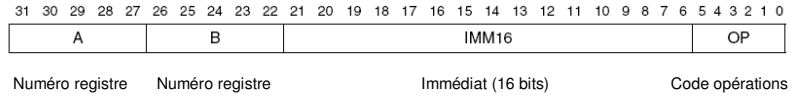
- Définition
 - RISC : modèle (3,0)
 - CISC : tous les autres
- RISC et pipeline
 - Les jeux d'instructions RISC facilitent la réalisation de pipelines performants
- « Solution » Intel et AMD pour IA-32
 - Convertir les instructions CISC en instructions RISC lors du décodage des instructions (conversion dynamique)
 - On conserve la compatibilité binaire
 - On a l'efficacité des pipelines « RISC »

Jeu d'instructions NIOS II

- Jeu d'instructions RISC simple
 - Instructions de longueur fixe : 32 bits
 - 32 registres r_0 à r_{31} avec $r_0 \equiv 0$
- Format d'instructions

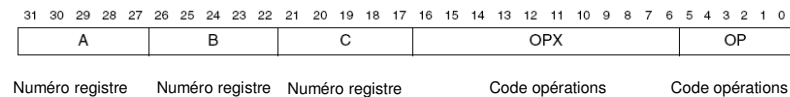


Les instructions de format I



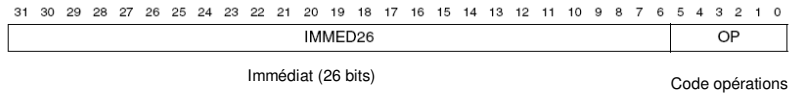
- **Instructions mémoire**
 - Adresse mémoire = [rA] + immédiat
- **Instructions arithmétiques/logiques/comparaison avec immédiat**
 - [rB] ← [rA] opération (imm16 étendu sur 32 bits)
 - Extension signée ou extension avec zéros
- **Instructions de branchement conditionnel**
 - Adresse de branchement = [PC] +4+ imm16 étendu (signé) sur 32 bits

Les instructions de format R



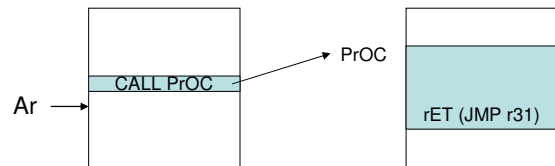
- **Instructions arithmétiques/logiques/comparaison**
 - [rC] ← [rA] opération [rB]
- **Instructions de décalage**
 - [rC] ← décalage [rA]
 - Nombre de décalages :
 - 5 bits poids faible de [rB]
 - 5 bits contenus dans OPX

Les instructions de type J



- **Instruction Call**

- range l'adresse de retour dans r31
- Saute à l'adresse = 4 poids fort de l'adresse CALLimm26|00



Les instructions mémoire

- **Chargement (load)**
 - $[rB] \leftarrow \text{Mémoire}([rA]+\text{simm16})$
 - Chargement mot de 32 bits
 - Chargement mot de 16 bits avec extension de signe ou extension zéros
 - Chargement octet avec extension de signe ou extension zéros
- **rangement (store)**
 - $\text{Mémoire}([rA]+\text{simm16}) \leftarrow [rB]$
 - rangement mot de 32 bits
 - rangement mot de 16 bits
 - rangement octet

ldw / ldwio	ldw rB, imm16(rA)	rB ← Mem32(rA+sim16)
ldb / ldbio	ldb rB, imm16(rA)	rB ← ES Mem8(rA+sim16)
ldbu / ldbuio	ldbu rB, imm16(rA)	rB ← Zero Mem8(rA+sim16)
ldh / ldhio	ldh rB, imm16(rA)	rB ← ES Mem16(rA+sim16)
ldbu / ldbuio	ldhu rB, imm16(rA)	rB ← Zero Mem16(rA+sim16)
stw / stwio	stw rB, imm16(rA)	Mem32(rA+sim16) ← rB
stb / stbio	stb rB, imm16(rA)	Mem8(rA+sim16) ← rB
sth / sthio	sth rB, imm16(rA)	Mem16(rA+sim16) ← rB

Les instructions arithmétiques et transfert

add	add rC,rA,rB	$rC \leftarrow rA + rB$
addi	addi rB,rA,imm16	$rB \leftarrow rA + \text{imm16}$
sub	sub rC,rA,rB	$rC \leftarrow rA - rB$
subi	subi rB,rA,imm16	$rB \leftarrow rA - \text{imm16}$ (pseudo : addi rB, rA, -imm16)
mul	mul rC,rA,rB	$rC \leftarrow rA * rB$ (32 bits poids faible du résultat)
muli	muli rB,rA,imm16	$rB \leftarrow rA * \text{imm16}$ (32 bits poids faible du résultat)
mulxss	mulxss rC,rA,rB	$rC \leftarrow rA * rB$ (32 bits poids fort) rA et rB signés
mulxuu	mulxuu rC,rA,rB	$rC \leftarrow rA * rB$ (32 bits poids fort) rA et rB non signés
div	div rC,rA,rB	$rC \leftarrow rA / rB$ (nombres signés)
divu	divu rC,rA,rB	$rC \leftarrow rA / rB$ (nombres non signés)

Pseudo-instructions

mov	mov rC,rA	$rC \leftarrow rA$ (add rC,rA,r0)
movi	movi rB,imm16	$rB \leftarrow \text{imm16}$ (addi rB, r0, imm16)
movui	movui rB,imm16	$rB \leftarrow \text{zimm16}$ (ori rB, r0, imm16)

Les instructions logiques

and	and rC,rA,rB	$rC \leftarrow rA \text{ et } rB$ (ET logique bit à bit)
andi	andi rB,rA,imm16	$rB \leftarrow rA \text{ et } \text{zimm16}$
or	or rC,rA,rB	$rC \leftarrow rA \text{ ou } rB$ (OU logique bit à bit)
ori	ori rB,rA,imm16	$rB \leftarrow rA \text{ ou } \text{zimm16}$
xor	xor rC,rA,rB	$rC \leftarrow rA \text{ xor } rB$ (OU exclusif bit à bit)
xori	xori rB,rA,imm16	$rB \leftarrow rA \text{ xor } \text{zimm16}$
nor	nor rC,rA,rB	$rC \leftarrow rA \text{ nor } rB$ (NOOr bit à bit)
norl	norl rB,rA,imm16	$rB \leftarrow rA \text{ nor } \text{zimm16}$
andhi	andhi rB,rA,imm16	$rB \leftarrow rA \text{ et } \text{imm16}(0000000000000000)$
orhi	orhi rB,rA,imm16	$rB \leftarrow rA \text{ et } \text{imm16}(0000000000000000)$
xorhi	xorhi rB,rA,imm16	$rB \leftarrow rA \text{ et } \text{imm16}(0000000000000000)$

Chargement d'une adresse dans un registre

movia rB, ETIQ

orhi rB, %hi ETIQ // 16 bits poids fort

ori rB, %lo ETIQ // 16 bits poids faible

Les instructions de comparaison

- **cmp*cond* rC,rA,rB**
 - $rC \leftarrow 1$ si (rA *cond* rB) **vrai** et $\leftarrow 0$ si (rA *cond* rB) **faux**
- Les autres conditions sont utilisables par pseudo-instructions

cmplt	cmplt rC,rA, rB	$rC \leftarrow 1$ si rA < rB (signés) et $\leftarrow 0$ sinon
cmpltu	cmpltu rC,rA, rB	$rC \leftarrow 1$ si rA < rB (non signés) et $\leftarrow 0$ sinon
cmpeq	cmpeq rC,rA, rB	$rC \leftarrow 1$ si rA == rB (signés) et $\leftarrow 0$ sinon
cmpne	cmpne rC,rA, rB	$rC \leftarrow 1$ si rA != rB (signés) et $\leftarrow 0$ sinon
cmpge	cmpge rC,rA, rB	$rC \leftarrow 1$ si rA >= rB (signés) et $\leftarrow 0$ sinon
cmpgeu	cmpgeu rC,rA, rB	$rC \leftarrow 1$ si rA >= rB (non signés) et $\leftarrow 0$ sinon
cmplti	cmplti rA,rA,IMM16	$rC \leftarrow 1$ si rA < simm16 (signés) et $\leftarrow 0$ sinon
cmpltui	cmpltui rA,rA,IMM16	$rC \leftarrow 1$ si rA < zimm16 (non signés) et $\leftarrow 0$ sinon
cmpeqi	cmpeqi rA,rA,IMM16	$rC \leftarrow 1$ si rA == simm16 (signés) et $\leftarrow 0$ sinon
cmpnei	cmpnei rA,rA,IMM16	$rC \leftarrow 1$ si rA != simm16 (signés) et $\leftarrow 0$ sinon
cmpgei	cmpgei rA,rA,IMM16	$rC \leftarrow 1$ si rA >= simm16 (signés) et $\leftarrow 0$ sinon
cmpgeui	cmpgeui rA,rA,IMM16	$rC \leftarrow 1$ si rA >= zimm16 (non signés) et $\leftarrow 0$ sinon

Les instructions de branchement conditionnel

- **B*cond* rA,rB,ETIQ**
 - Si (rA *cond* rB) vrai, branchement à l'adresse PC+4+SIMM16
 - Si (rA *cond* rB) faux, séquence : adresse PC+4
- Les conditions
 - eq (=), ne (≠)
 - lt, ltu (< signé et non signé)
 - ge, geu (≥ signé et non signé)
 - gt, gtu (> signé et non signé) : pseudo-instruction
 - le, leu (≤ signé et non signé) : pseudo-instruction

Instructions de décalage et rotation

- Décalages logiques à droite et à gauche
 - insertion de zéros
- Décalage arithmétique à droite
 - insertion du bit de signe
- Rotations
 - Les bits sortants (à droite/ à gauche) sont réintroduits (à gauche/ à droite)

srl	srl rC,rA, rB	$rC \leftarrow rA \gg (\text{nb spécifié dans } rB) - \text{Logique}$
srl	srl rC,rA, imm5	$rC \leftarrow rA \gg (\text{imm5}) - \text{Logique}$
sra	sra rC,rA, rB	$rC \leftarrow rA \gg (\text{nb spécifié dans } rB) - \text{Arithmétique}$
sra	sra rC,rA, imm5	$rC \leftarrow rA \gg (\text{imm5}) - \text{Arithmétique}$
sll	sll rC,rA, rB	$rC \leftarrow rA \ll (\text{nb spécifié dans } rB)$
sll	sll rC,rA, imm5	$rC \leftarrow rA \ll (\text{imm5}) - \text{Logique}$

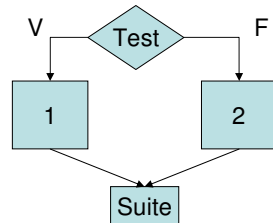
ror	ror rC,rA, rB	$rC \leftarrow \text{rotation droite de } rA (\text{nb spécifié dans } rB)$
rol	rol rC,rA, rB	$rC \leftarrow \text{rotation gauche de } rA (\text{nb spécifié dans } rB)$
rol	rol rC,rA, imm5	$rC \leftarrow \text{rotation gauche de } rA (\text{imm5})$

Utilisation et nom des registres

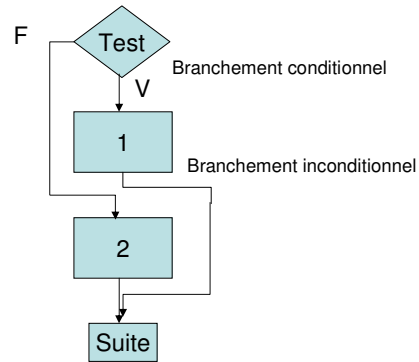
registre	Nom	Fonction
r0	zero	0x00000000
r1	at	Temporaire pour l'assembleur
r2		
.		
r23		
r24	et	Temporaire pour le traitement des exceptions
r25	bt	Temporaire pour les points d'arrêt
r26	gp	Pointeur global
r27	sp	Pointeur de pile (stack pointer)
r28	fp	Pointeur de trame (Frame pointer)
r29	ea	Adresse de retour des exceptions
r30	ba	Adresse de retour des points d'arrêt
r31	ra	Adresse de retour

Jeux d'instructions : branchements conditionnels

Si condition alors « suite d'instructions 1 » sinon « suite d'instructions2 »



Instructions de test
Instructions de branchement conditionnel



Branchements conditionnels

- Test et branchement en 1 seule instruction
 - Bcond rA,rB,ETIQ
 - Toutes les conditions (NIO S II) et EQ, NE pour MIPS
- Résultat du test dans un registre et Branchement conditionnel selon condition dans un registre
 - NIO S II
 - COMPcond (1 ou 0 dans registre) et Bcond rA,r0, ETIQ
 - MIPS
 - SLT (1 ou 0 dans un registre) et Bcond Rs1, ETIQ
 - cond : toutes les conditions sauf EQ et NE. Comparaison Rs1 et 0
- Résultat du test dans un registre code condition et branchement conditionnel selon code condition
 - ARM
 - CMP : résultats de la comparaison dans le registre code condition
 - Bcond ETIQ : branchement selon le registre code condition

Exemple : calcul de la valeur absolue

Mettre dans R1 la valeur absolue du contenu de R2

NIOS II
 MOV R1,R2
 BGT R2,R0, Suite
 SUB R1,R0,R2
 Suite :

ARM
 MOV R1,R2
 CMP R2,#0
 BGT Suite
 RSB R1,R2,#0 // r1 ← -R1
 Suite :

MIPS
 ADD R1,R2,R0
 SLT R3,R0,R2
 BGT R3,Suite
 SUB R1,R0,R2
 Suite :

ARM (*instructions conditionnelles*)
 MOV R1,R2
 CMP R2,0
 RSBLT R1,R2,#0

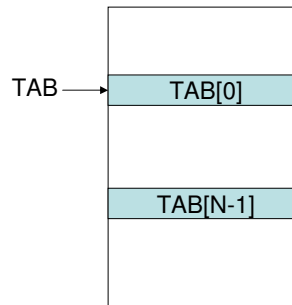
Accès à un tableau

- Exemple : somme des éléments d'un tableau de N entiers à l'adresse TAB

```
S=0
For (i=0;i<N;i++)
  S+=TAB[i];
```

Version NIOS II

```
MOVIA r2, TAB // adresse du tableau dans r2
MOV r1, 0 // r1=0
MOV r3, N // N dans r3
SRLI r3,r3,2 // 4*N dans r3
ADD r3,r3,r2 //Adresse du mot après TAB[N-1]
Deb : LDW r4,0(r2) // chargement de Tab[i]
ADD r1,r1,r4 // accumulation
ADDI r2,r2,4 // adresse de l'élément suivant
BLT r2,r3,Deb // boucle tant que r2<r3
// Résultat dans r1
```



2 instructions pour accéder à chaque élément du tableau

Modes d'adressage plus complexes

- Exemple ARM

Instructions mémoire

Instruction	Signification	Action
LDR	Chargement mot	Rd ← Mem ₃₂ (AE)
LDRB	Chargement octet	Rd ← Mem ₈ (AE)
STR	Rangement mot	Mem ₃₂ (AE) ← Rd
STRB	Rangement octet	Mem ₈ (AE) ← Rd

Modes d'adressage

Mode	Assembleur	Action
Déplacement 12 bits, Pré-indexé	[Rn, #déplacement]	Adresse = Rn + déplacement
Déplacement 12 bits, Pré-indexé avec mise à jour	[Rn, #déplacement] !	Adresse = Rn + déplacement Rn ← Adresse
Déplacement 12 bits, Post-indexé	[Rn], #déplacement	Adresse = Rn Rn ← Rn + déplacement
Déplacement dans Rm Préindexé	[Rn, ± Rm, décalage]	Adresse = Rn ± [Rm] décalé
Déplacement dans Rm Préindexé avec mise à jour	[Rn, ± Rm, décalage] !	Adresse = Rn ± [Rm] décalé Rn ← Adresse
Déplacement dans Rm Postindexé	[Rn], ± Rm, décalage	Adresse = Rn Rn ← Rn ± [Rm] décalé

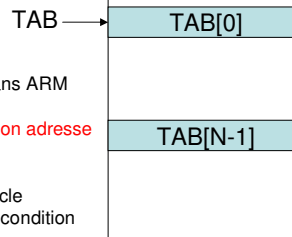
Accès à un tableau avec adressage post-indexé

- Exemple : somme des éléments d'un tableau de N entiers à l'adresse TAB

```
S=0
For (i=0;i<N;i++)
  S+=TAB[i];
```

Version ARM

```
ADR r2, TAB // adresse du tableau dans r2
MOV r0, 0 // r0=0. NB : r0 est différent de 0 dans ARM
MOV r1, N // N dans r1
Deb : LDR r3,[r2],#4 // chargement de Tab[i] et préparation adresse
// suivante
ADD r0,r0,r3 // accumulation
SUBS r1,r1,#1 // décrémentation compteur de boucle
// et positionnement du registre code condition
BGT Deb // boucle tant que r1>0
// Résultat dans r0
```



1 seule instruction pour accéder à chaque élément du tableau

La pile mémoire

- Mécanisme d'accès mémoire sans adressage explicite

- L'adresse mémoire est contenu dans un registre « pointeur de pile » ou « stack pointer » (SP)

- Instructions spécifiques d'accès

- Empilement (Push)
- Dépilement (Pop)

Push Ri

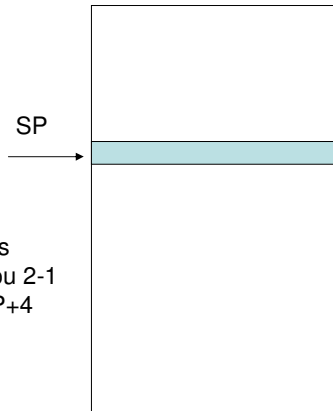
- 1) $SP \leftarrow SP - 4$
- 2) $MEM(SP) \leftarrow Ri$

POP Ri

- $Ri \leftarrow MEM(SP)$
 $SP \leftarrow SP + 4$

4 possibilités

- Ordre 1-2 ou 2-1
- SP-4 ou SP+4



Les instructions « pile »

- Pointeur de pile

- RISC : SP est un registre général
 - r27 dans NIOS II, r29 dans MIPS, r13 dans ARM
- IA-32 : SP est un registre spécifique

- Instructions Push et Pop

- RISC : implantées avec instructions ADD, SUB, Load et Store
- IA-32 : Push et Pop

- Empilement et dépilement de plusieurs registres

- ARM : les instructions STM(store multiple) et LDM (load multiple) avec le registre r13 permettent en une instruction un empilement et un dépilement de plusieurs registres
- IA-32 : PUSHA et POPA permettent d'empiler et dépiler tous les registres généraux

Les appels/retours de fonctions/procédures

- Appel de fonctions
 - Passage des paramètres
 - Par registre : nombre limité (ex : 4 pour MIPS)
 - Par pile : paramètres supplémentaires
 - Sauvegarde de l'adresse de retour
 - Dans un registre général
 - R31 (NIOS II, MIPS), R14 (ARM)
 - Dans un registre spécifique
 - LR (PowerPC)
 - Dans la pile
 - IA-32
- Retour
 - Récupération du résultat
 - Par registres
 - Par pile
 - Instruction chargeant l'adresse de retour dans PC
 - RET (≡ JMP r31 dans NIOS II), JMP R31 (MIPS)
 - RET (≡ POP PC dans IA-32)

gcc et NIOS II : appel fonction terminale

<pre> int A, B, C; int main(void){ B=10; C=15; A= max (B,C); return 0;} int max(int x, int y) { if (x>y) return x; else return y; } </pre>	<pre> 00000258 2005883a 0000025c 29000116 00000260 2805883a 00000264 f800283a 00000268 00c003c4 0000026c 00800284 00000270 deffff04 00000274 1009883a 00000278 180b883a 0000027c dfc00015 00000280 d0e00715 00000284 d0a00615 00000288 00002580 0000028c d0a00515 00000290 0005883a 00000294 dfc00017 00000298 dec00104 0000029c f800283a </pre>	<pre> max: add r2, r4, zero blt r5, r4, +0x4 (00000264) else return y: add r2, r5, zero } ret main: addi r3, zero, 0xf addi r2, zero, 0xa addi sp, sp, -0x4 add r4, r2, zero add r5, r3, zero stw ra, 0(sp) stw r3, -32740(gp) stw r2, -32744(gp) call 0x00000096 (00000258, max) stw r2, -32748(gp) add r2, zero, zero ldw ra, 0(sp) addi sp, sp, 0x4 ret </pre>
--	--	--

gcc et NIOS II : utilisation de fonctions imbriquées

```

main:
00000320 deffff04 addi sp, sp, -0x4
00000324 01800034 orhi r6, zero, 0x0
00000328 3181c904 addi r6, r6, 0x720
0000032c dfc00015 stw ra, 0(sp)
00000330 01c00284 addi r7, zero, 0xa
00000334 03801904 addi r14, zero, 0x64
00000338 03401404 addi r13, zero, 0x50
0000033c 03001184 addi r12, zero, 0x46
00000340 02c00f04 addi r11, zero, 0x3c
00000344 02800dc4 addi r10, zero, 0x37
00000348 02400a04 addi r9, zero, 0x28
0000034c 020008c4 addi r8, zero, 0x23
00000350 00c00784 addi r3, zero, 0x1e
00000354 00800644 addi r2, zero, 0x19
00000358 3009883a add r4, r6, zero
0000035c 380b883a add r5, r7, zero
00000360 33800015 stw r14, 0(r6)
00000364 33400115 stw r13, 4(r6)
00000368 33000215 stw r12, 8(r6)
0000036c 32c00315 stw r11, 12(r6)
00000370 32800415 stw r10, 16(r6)
00000374 32400515 stw r9, 20(r6)
00000378 32000615 stw r8, 24(r6)
0000037c 30c00715 stw r3, 28(r6)
00000380 30800815 stw r2, 32(r6)
00000384 31c00915 stw r7, 36(r6)
00000388 000027c0 call 0x0000009e (0000027c, tri)
0000038c 0005883a add r2, zero, zero
00000390 dfc00017 ldw ra, 0(sp)
00000394 dec00104 addi sp, sp, 0x4
00000398 f800283a ret

```

int v[10];

main()
{v[0]=100;v[1]=80;v[2]=70;v[3]=60;v[4]=55;
v[5]=40;v[6]=35;v[7]=30;v[8]=25;v[9]=10;
tri (v,10);}

change (int v[], int k, int m)
{int temp;
temp=v[k];
v[k]=v[m];
v[m]=temp;}

tri (int v[], int n)
{int i, j;
for (i = n-1; i>0; i--){
for (j=i-1; j>=0 ; j--){
if (v[j] >v [i]) change (v,j, i);}}

L3 Informatique
2007-2008

L313-Architecture des ordinateurs
D. Etiemble

33

gcc et NIOS II : fonctions imbriquées

```

tri:
0000027c defff904 addi sp, sp, -0x1c
00000280 dc800315 stw r18, 12(sp)
00000284 2cbfffc4 addi r18, r5, -0x1
00000288 dc000215 stw r19, 8(sp)
0000028c dfc00615 stw ra, 24(sp)
00000290 dc000515 stw r16, 20(sp)
00000294 dc400415 stw r17, 16(sp)
00000298 dd000115 stw r20, 4(sp)
0000029c dd400015 stw r21, 0(sp)
000002a0 2027883a add r19, r4, zero
000002a4 0480120e bge zero, r18, +0x48 (000002f0)
000002a8 957fffc4 addi r21, r18, -0x1
000002ac a823883a add r17, r21, zero
000002b0 a8000416 blt r21, zero, +0x34 (000002e8)
000002b4 900490ba slli r2, r18, 0x2
000002b8 a80690ba slli r3, r21, 0x2
000002bc 14e9883a add r20, r2, r19
000002c0 1ce1883a add r16, r3, r19
000002c4 81c00017 ldw r7, 0(r16)
000002c8 a2000017 ldw r8, 0(r20)
000002cc 880b883a add r5, r17, zero
000002d0 843fffc4 addi r16, r16, -0x4
000002d4 8c7fffc4 addi r17, r17, -0x1
000002d8 9809883a add r4, r19, zero
000002dc 900d883a add r6, r18, zero
000002e0 41c00c16 blt r8, r7, +0x30 (00000314)
000002e4 883ff70e bge r17, zero, -0x24 (000002c4)
000002e8 a825883a add r18, r21, zero
000002ec 057fe1e6 blt zero, r21, -0x48 (000002a8)

```

000002e4 883ff70e bge r17, zero, -0x24 (000002c4)

000002e8 a825883a add r18, r21, zero

000002ec 057fe1e6 blt zero, r21, -0x48 (000002a8)

000002f0 dc000617 ldw ra, 24(sp)

000002f4 dc000517 ldw r16, 20(sp)

000002f8 dc400417 ldw r17, 16(sp)

000002fc dc800317 ldw r18, 12(sp)

00000300 dcc00217 ldw r19, 8(sp)

00000304 dd000117 ldw r20, 4(sp)

00000308 dd400017 ldw r21, 0(sp)

0000030c dec00704 addi sp, sp, 0x1c

00000310 f800283a ret

00000314 00002580 call 0x00000096 (00000258, change)

00000318 883ff70e bge r17, zero, -0x58 (000002c4)

0000031c 003ff206 br -0x38 (000002e8)

tri (int v [], int n)
{int i, j;
for (i=n-1; i>0; i--){
for (j=i-1; j>=0 ; j--){
if (v[j] >v [i]) change (v,j, i); }}

change:
00000258 300e90ba slli r7, r6, 0x2

0000025c 280490ba slli r2, r5, 0x2

00000260 390d883a add r6, r7, r4

00000264 110b883a add r5, r2, r4

00000268 30c00017 ldw r3, 0(r6)

0000026c 29000017 ldw r4, 0(r5)

00000270 28c00015 stw r3, 0(r5)

00000274 31000015 stw r4, 0(r6)

00000278 f800283a ret

L3 Informatique
2007-2008

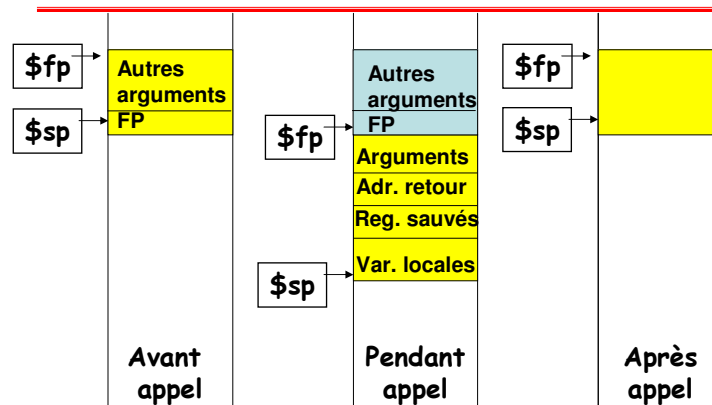
L313-Architecture des ordinateurs
D. Etiemble

34

Fonctions : la pile d'appel

- Pile d'appel = contexte d'exécution de la fonction
 - Arguments de la fonction appelée
 - Sauvegarde des registres
 - Adresse de retour (indispensable pour fonctions imbriquées)
 - Registres utilisés par la fonction (pour pouvoir les restaurer pour la fonction appelante)
 - Variables locales à la fonction appelée

Pile d'appel



FP (*Frame pointer* ou Pointeur de trame) pointe sur le premier mot d'une fonction
 Adresse stable durant l'exécution de la fonction
 SP (*Stack pointer* ou Pointeur de pile)
 SP évolue au cours de l'exécution de la fonction

FP est initialisé par SP et SP est restauré à partir de FP