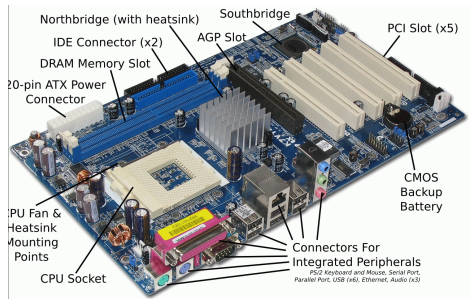
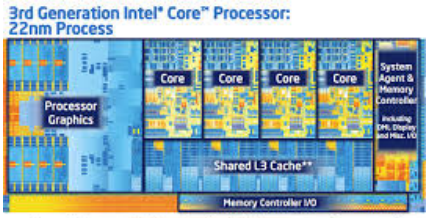
Architecture des ordinateurs

Cécile Germain
cecile.germain@lri.fr





Plan

1. Représentation de l'information
   Comment faire exploser un satellite
2. Architecture Logicielle
   Compiler les langages de haut niveau
3. Algèbre de Boole et circuits logiques
4. Micro-Architecture
   Fabriquer une UC avec des circuits très simples
5. Mémoire
   Le problème central
6. Introduction aux processeurs performants

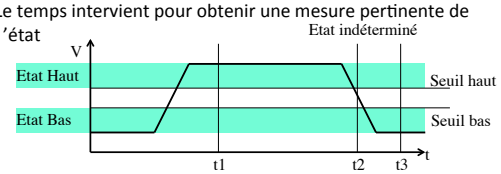
1. Représentation de l'information

Plan

- Définitions
- Représentation des entiers
- Représentation des caractères
- Représentation des réels

L ’information - Définition

- Analogique : grandeur continue
- ou Numérique (= digitale) :
  Information = connaissance d ’un état parmi un nombre fini d ’états possibles
- Support : grandeur physique continue - tension
- Le temps intervient pour obtenir une mesure pertinente de l ’état



Quantité d’information

- 1 bit = quantité d ’information liée à la connaissance d ’1 état parmi 2 -> codage par 0 et 1

- Avec 2 bits, on peut coder 4 états
  - France 00
  - GB 01
  - Italie 10
  - Espagne 11

- Avec 3 bits, on peut coder 8 états
  - France 000
  - GB 001
  - Italie 010
  - Espagne 011
  - Allemagne 100
  - Luxembourg 101
  - Belgique 110
  - Grèce 111

Quantité d'information

- 1 bit = quantité d 'information liée à la connaissance d '1 état parmi 2 -> codage par 0 et 1
- Avec n bits, on peut coder 2^n états, donc la quantité d'information contenue dans la connaissance d'1 état parmi *N* est

$$\lceil \mathrm{Log}_2(N) \rceil \text{ bits}$$

Ordres de grandeurs

Kilo	2^{10}	10^3
Mega	2^{20}	10^6
Giga	2^{30}	10^9
Tera	2^{40}	10^{12}
Peta	2^{50}	10^{15}

Codage de l'information : quelle *information* ?

- Tous les processeurs
  - Nombres
    - Entiers naturels et relatifs
    - Réels
  - Caractères
  - Instructions
- Certains processeurs représentent et traitent les vecteurs d'entiers et de réels
  - Processeurs « multimédia »
  - Console de jeux
- En général, pas les enregistrements (record), listes et autres types complexes

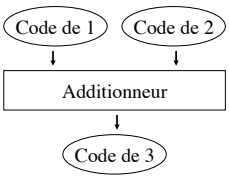
Codage de l'information : quel *codage*?

Doit permettre de réaliser des

opérateurs matériels

fonctionnels et rapides.

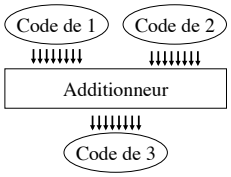
Opérateur matériel =
circuit électronique
non programmable
exemple: un additionneur



Codage et opérateurs

Sur un nombre fixe de bits

caractéristique du processeur et de la technologie des Circuits Intégrés, typiquement 32 ou 64 bits



Notations (humaines)

- Binaire : 00101100
  Peu pratique pour le traitement manuel
- Hexadécimale : 1 digit (chiffre) hexadécimal (base 16) par quartet (4 bits)

  0x2C

  La notation hexadécimale ne préjuge pas du contexte d'interprétation

  exemple : ' A ' est codé par 0x41

Plan

- Définitions
- Représentation des entiers
  - Entiers naturels
  - Entiers relatifs
- Représentation des caractères
- Représentation des réels

Codage des entiers naturels

Sur *n* bits, écriture en base 2

$$N = \sum_{i=0}^{n-1} x_i 2^i$$

Sur 8 bits : 10010001 code $1+2^3+2^7$

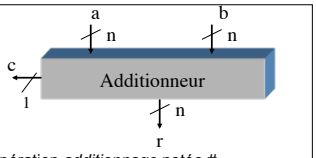
Sur *n* bits, les nombres représentables sont

$[0 \ldots 2^n-1]$

Sur 8 bits : [0, 255]

L 'additionneur

- Additionneur : circuit matériel qui effectue l 'addition naïve des entiers naturels sur *n* bits et tronque éventuellement la retenue



Opération *additionnage* notée #
a, b opérandes; r résultat ; c retenue (Carry)

a	b	r	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Additionneur 1 bit

L 'additionneur

Le résultat du calcul dans l 'additionneur peut être faux : *retenue*

Sur 3 bits

001 # 001 = 010 Correct : 1 + 1 = 2

001 # 111 = 000 Faux : 1 + 7 ≠ 0

est l'addition modulo 2^n

Addition des naturels

Le résultat du calcul dans l 'additionneur peut être faux

```
Un programme de test
entier_non_signé : val ;
tant que val <> 0 faire
  val = val + 1;
fin tant que
afficher (' Quelle erreur ! ');
```

Le résultat dépend

- du langage de programmation
- du compilateur et des options

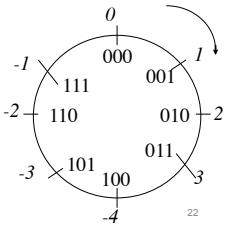
Codage des entiers relatifs en complément à 2 sur n bits

- Définition
  - Nombres représentables
    $[-2^{n-1} \dots 2^{n-1} - 1]$
  - Si N ≥ 0, N est codé comme un entier naturel
  - Si N < 0, N est codé par la représentation en naturel de
    $2^n + N$

Codage des entiers relatifs en complément à 2 sur n bits

- Définition
  - Nombres représentables
    $[-2^{n-1} \dots 2^{n-1} - 1]$
  - Si N ≥ 0, N est codé comme un entier naturel
  - Si N < 0, N est codé par la représentation en naturel de
    $2^n + N$
- Exemple 3 bits

Nombre	Code
0	000
1	001
2	010
3	011
-1	De 8 -1 = 7, soit 111
-2	De 8 - 2 = 6, soit 110
-3	De 8 – 3 = 5, soit 101
-4	De 8 – 4 = 4, soit 100



Propriétés du codage en complément à 2

1. Le bit de poids fort
   des positifs est 0
   des négatifs est 1
2. La représentation de -1 est le mot où tous les bits sont à 1 : FF (8bits), FFFF (16bits) etc.
3. L'opposé d 'un relatif s'obtient en complémentant bit à bit et en ajoutant 1
   Exemples
   2 est codé par 010 sur 3 bits. Comment coder –2 ?
   Complément bit à bit : 101
   \+ 1
   -2 est donc codé par 110
   Dans l'autre sens, 110 -> 001 -> 010

Extension de signe

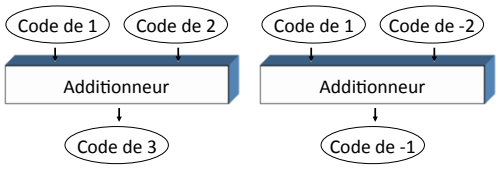
Soit un relatif représenté en complément à 2 sur n bits. Sa représentation en complément à 2 sur m>n bits s'obtient en répliquant le bit de poids fort sur les bits manquants

5 est représenté sur 8 bits par 0b00000101
5 est représenté sur 16 bits par 0b0000000000000101
-5 est représenté sur 8 bits par 0b11111011
-5 est représenté sur 16 bits par 0b1111111111111011

Notation: $ES_m(Imm_n)$

Opérations arithmétiques

Le circuit additionneur peut effectuer aussi l'addition des relatifs *s'ils sont codés en complément à 2* : 1 seul circuit



Addition des relatifs

Le résultat du calcul dans l'additionneur peut être faux :

overflow

Sur 3 bits :

001 # 001 = 010 correct : 1 + 1 -> 2
001 # 110 = 111 correct : 1 + (-2) = -1
111 # 111 = 110 correct : -1 + -1 = -2
011 # 001 = 100 faux : 3 + 1 n 'est pas codable
110 # 101 = 011 faux :-2 + (-3) n 'est pas codable

- Attention: overflow <> retenue

Addition des relatifs

On démontre :

L'additionnage des codes de deux relatifs N et M est égal à N + M si et seulement si N + M est codable

L'additionneur est aussi satisfaisant que possible

L'additionnage des codes de deux relatifs N et M est égal à N +M si et seulement si

- *N et M sont de signes contraires*
- *Ou bien N et M sont de même signe, et le bit de signe du résultat est égal à la retenue*

Facile à tester en matériel

Autres codages

- En excès : sur n bits, excès à 2^{n-1}

$$N = N' - 2^{n-1}$$

où N' est l'interprétation du codage en naturels

- Signe et valeur absolue

Conclusion

- Avec les codages choisis, l'additionneur effectue l'addition des naturels et des relatifs aussi bien que possible: tout résultat représentable est correctement calculé.
- Mais le résultat peut être faux s'il n'appartient pas à l 'ensemble d'entiers représentables
  - Analyser l'application et le programme pour garantir que le résultat reste codable
  - Si ce n'est pas possible, utiliser les ressources de l 'environnement pour détecter l'erreur. C'est toujours faisable en utilisant seulement les résultats de l'addition.

Plan

- Définitions
- Représentation des entiers
- Représentation des caractères
- Représentation des réels

Codage des caractères

- ASCII American Standard Code for Information Interchange : 7 bits + 1 bit de parité
  - Caractères alphanumériques latins non accentués + caractères spéciaux
- Latin -1 : 8 bits, standard ISO 8859-1.
  - Caractères alphanumériques latins accentués, extension de l'ASCII 7 bits.
- Unicode : 16 bits,
  - Caractères d 'alphabets variés

Codes ASCII 7 bits

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

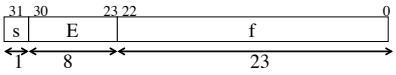
Plan

- Définitions
- Représentation des entiers
- Représentation des caractères
- Représentation des réels

Codage des réels

- Problèmes :
  - Arrondir
  - Représenter équitablement des nombres très grands et très petits
- Principe : représentation normalisée par mantisse et exposant, avec un choix judicieux des valeurs de l 'exposant

Codage IEEE 754 simple précision



E = interprétation en naturel

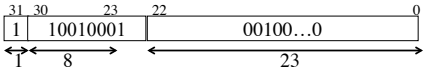
Pour E ≠ 0 et 255,

La valeur codée est : $(-1)^s . 2^{E-127} . 1,f$

où 1,f doit être interprété en base 2

Soit e = E -127. Alors $-126 \leq e \leq 127$

Codage IEEE 754 simple précision



Que représente 0xC8900000 ?

1100 1000 1001 0000 …0

s = 1 ; E = 10010001 = 145 donc e = 18

f = 001 0000 …0 donc m = $1 + 2^{-3}$

Le nombre codé est $-2^{18}(1 + 2^{-3})$

Codage IEEE 754 simple précision

- Codage de 2,5

  $2,5 = 5x2^{-1} = 101_2x2^{-1} = 1,01_2x2^1$

  E = 127+1 = 128 f = 01

  codage : 0 10000000 010…0=0x40200000
- Codage de 0,75

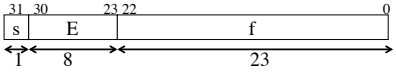
  $0,75 = 3x2^{-2} = (2+1)x2^{-2} = (1 + 2^{-1})\ 2^{-1} = 1,1_2x2^{-1}$

  E = 127-1 = 126 f = 1

  codage : 0 01111110 100…0=0x3F400000

Codage IEEE 754 simple précision

Les cas exceptionnels (voir + loin interprétation)



Nom	E	f	valeur
Dénormalisé	0	non 0	$(-1)^s.2^{-127}.0,f$
Zéro	0	0	0
Infini	255	non 0	$(-1)^s\infty$
NaN	255	0	Not a Number (NaN)

L'addition des flottants

1. Comparaison des exposants — $2,5 = 2^1x1,01_2$; $3,25 = 2^1x1,101_2$
2. Décalage à droite de la mantisse du plus petit nombre — Différence 0, Pas de décalage
3. Somme des mantisses — $1,101_2 + 1,01_2 = 10,111_2$
4. Normalisation du résultat — $10,111_2\ x2^1 = 1,0111_2\ x2^2$
5. Traitement cas exceptionnels overflow, underflow, zero — Non

Résultat = $2^2x1,0111_2 = 5,75$

L'addition des flottants

1. Comparaison des exposants — $2,5 = 2^{1}x1,01_2$; $0,75 = 2^{-1}x1,1_2$
2. Décalage à droite de la mantisse du plus petit nombre — Différence 2 ; $0,75 = 2^{1}x0,011_2$
3. Somme des mantisses — $1,01_2 + 0,011_2 = 1,101_2$
4. Normalisation du résultat — Normalisé
5. Traitement cas exceptionnels overflow, underflow, zero — Non

Résultat = $2^{1}x1,101_2 = 3,25$

L'addition des flottants

1. Comparaison des exposants — $2^{30} = 2^{30}x1,0..0_2$; $3,25 = 2^{1}x1,101_2$
2. Décalage à droite de la mantisse du plus petit nombre — Différence 29 Mantisse à 0
3. Somme des mantisses — $1,101_2 + 0,0_2 = 1,101_2$
4. Normalisation du résultat — Normalisé
5. Traitement cas exceptionnels overflow, underflow, zero — Arrondi

Résultat = 2^{30}

Erreurs d'arrondi

- Le résultat d'une addition est souvent faux
- Mais le standard impose que le résultat soit le résultat exact arrondi, avec des choix possibles
  - Au plus près / Valeur Inférieure / Valeur supérieure
  - Mais $2^{30} + 3,25$ a toujours pour résultat 2^{30}
- Réalisé avec 3 bits supplémentaires seulement

Erreurs d'arrondi

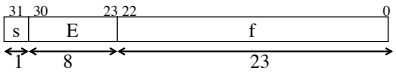
- L'addition n'est plus associative

$$(-2^{30} + 2^{30}) + 3{,}25 = 3{,}25$$

$$-2^{30} + (2^{30} + 3{,}25) = 0$$

- Le traitement des erreurs d'arrondi dans les méthodes numériques est une composante essentielle de l'informatique numérique
  - Méthodes mathématiques
  - Très bien connu pour l'algèbre linéaire et encapsulé dans des bibliothèques dont l'utilisation est impérative
  - Beaucoup moins bien en simulation *ab initio*

Cas exceptionnels : codage du résultat en IEEE 754 simple précision



Nom	E	f	valeur
Dénormalisé	0	non 0	$(-1)^s.2^{-127}.0,f$
Zéro	0	0	0
Infini	255	0	$(-1)^s\infty$
NaN	255	Non 0	Not a Number (NaN)

Arithmétique des cas exceptionnels

- Arithmétique étendue
  - Infini : trop grand
  - NaN : not a number, opération hors de l'ensemble de définition. NaN est absorbant.
  - Règles classiques sur l'infini
- Dénormalisé simple précision
  - Motivation
    - La différence de deux nombres représentables peut ne pas l'être.

    $1{,}11\ldots1x2^{-126} - 1{,}11\ldots0x2^{-126} = 0{,}0\ldots01x2^{-126} = 2^{-129}$ n'est pas représentable
    - Donc les tests x==y (comparaison des bits) et x-y = 0 ne seraient pas équivalents
  - Evidemment, le même problème existe pour la différence de deux dénormalisés

Traitement des erreurs

La norme impose l'enregistrement des opérations erronées dans des drapeaux persistants

Drapeau	Condition	Résultat
Underflow	Nb trop petit	0, xmin, nombre dénormalisé
Overflow	Nb trop grand	+-∞ ou xmax
Division par 0	Division par 0	+-∞
Invalid Operation	Résultat NaN, opérandes non NaN	NaN
Inexact	Arrondi	Résultat arrondi

Traitement des opérations erronées

- Comportements souhaitables
  - Arrêt immédiat
    - $x/(1+x^2)$ pour $x = 2^{65}$ produit x^2=infini, résultat 0 alors que le résultat est de l'ordre de 1/x, donc représentable. Pas d'intérêt à continuer
  - Aucune action – continuer dans l'arithmétique étendue
    - Arrondi presque toujours
    - Dénormalisé si pas de soustraction problématique
  - Traitement par l'application
    - Méthodes d'exploration : la valeur peut tomber en dehors de l'ensemble de définition
    - Dénormalisé si soustraction problématique
    - Etc.
- Les choix sont paramétrés par des bit du mot d'état, eux même positionnés via des options de compilation ou des appels à des fonctions de la *libm* à l'exécution

Plus sur les exceptions

- L'initialisation du mot de contrôle est effectuée dans crt0.o – avec lequel tout programme est automatiquement lié ("pré-main")
  - Machine-spécifique: le mot de contrôle n'a pas le même format
- Positionnement par fonctions de la libm
  - http://www.unix.com/man-page/All/3/fenv/
  - C99 ne supporte que
    - FE_NOMASK_ENV which represents an environment where every exception raised causes a trap to occur
    - FE_DFL_ENV This is the environment setup at program start and it is defined by ISO C to have round to nearest, all exceptions cleared and a non-stop (continue on exceptions) mode.
  - Certaines fonctions permettent un contrôle plus fin, mais ne sont pas portables
    - Feenableexcept(except)
    - Mais ne sont pas disponibles sur tous les systèmes : http://www.gnu.org/software/hello/manual/gnulib/feenableexcept.html

Un exemple

```
#ifndef _GNU_SOURCE
  # define _GNU_SOURCE
#endif
#include <stdio.h>
#include <fenv.h>
main () {
  double x, y, z;
  z= 1/(x-x);
  printf ("%E\n", z);
}
```

```
> gcc prog.c -lm
> prog
> INF
```

Pourquoi ?

Par défaut, les exceptions sont silencieuses

Un exemple

```
#ifndef _GNU_SOURCE
  # define _GNU_SOURCE
#endif
#include <stdio.h>
#include <fenv.h>
main () {
  fesetenv(FE_NOMASK_ENV);
  double x, y, z;
  z= 1/(x-x);
  printf ("%E\n", z);
}
```

```
> gcc prog.c -lm
> prog
> Floating exception
```

Pourquoi ?

On a demandé de positionner le masque des exceptions

Un exemple

```
#ifndef _GNU_SOURCE
  # define _GNU_SOURCE
#endif
#include <stdio.h>
#include <fenv.h>
main () {
  double x, y, z;
  fesetenv(FE_NOMASK_ENV);
  z= 1/(x-x);
  printf ("%E\n", z);
}
```

```
> gcc -ffast-math prog.c -lm
> prog
> INF
```

Pourquoi ?

Le programme positionne le masque, mais l'option de compilation

La norme IEEE 754

- Codage simple précision (32 bits), double précision (64 bits), simple étendue et double étendue
- Objectif : reproductibilité
  - Spécification des arrondis et des cas exceptionnels
  - Arithmétique étendue
- Interface de programmation
  - Le type des résultats en tant qu'exceptionnel est consultable par des fonctions de la *libm*
  - Options de compilation pour l'interface avec masquage ou non

2. Architecture logicielle

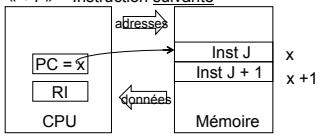
Plan

1. Le modèle de Von-Neumann
2. Format des instructions
3. Instructions arithmétiques
4. Instructions d'accès mémoire
5. Instructions de contrôle
6. Procédures

Le modèle d'ordinateur de Von Neumann

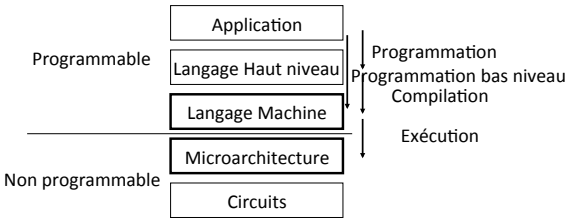
- *Mémoire = Instruction et données*
- *Processeur : Tant que <vrai> faire*
  Lire instruction : RI <- Mem[PC] ; PC <- PC + "1"
  Exécuter instruction : suite d'actions dépendant de l'instruction

RI : Registre Instructions
« +1 » = Instruction suivante



EN MATERIEL

Les niveaux d'un système informatique

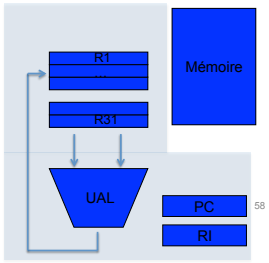


Architecture logicielle

- Le processeur tel que le voit le compilateur, ou un programmeur d'applications spécifiques bas niveau : interface de programmation
- Spécifié par un langage-machine = jeu d'instructions = "assembleur »
- Abstraction qui peut correspondre à une grande variété de processeurs matériels : x86 du 8086 (1980) au Pentium IV (2000)
- Dans toute la suite, on considère un sous-ensemble du jeu d'instruction MIPS

Le processeur : partie opérative

- Contraintes
  - en matériel
  - calcul uniquement sur les registres
- Caractéristiques RISC (Reduced Instruction Set Computer)
- Registres : organes de stockage à l'intérieur du processeur
  - Chacun contient 1 mot
- UAL : opérations additionnage, soustraction, division et multiplication par puissances de 2 etc.

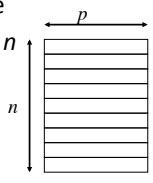


Caractéristiques élémentaires du CPU

- Largeur du chemin de données = taille de l'UAL : 32 ou 64 bits. 32 bit pour MIPS.
- R0 toujours 0 pour MIPS
- Chaque registre a la même taille que le chemin de données

Caractéristiques élémentaires mémoire

- Tableau de mots de largeur fixe *p*
  - *Mémoire organisée par octets p* = 8
  - *Mémoire organisée par mots de 16 bits p* = 16
- Accessible chacun par leur adresse
- Taille = nombre de mots mémoire *n*



Un exemple

Instruction d'affectation LHN

```
int A, B, C ;
A =  B + C;
```

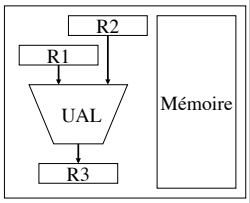
- Comment sont stockées les variables A, B et C ?
- Comment effectuer le calcul ?

Un exemple

Instruction d'affectation LHN

```
int A, B, C ;
A =  B + C;
```

Charger B depuis la mémoire vers le CPU
LOAD R1, "adresse de B"
Charger C depuis la mémoire vers le CPU
LOAD R2, "adresse de C"
Effectuer : R3 <- R1 + R2
ADD R3, R2, R1
Stocker le résultat en mémoire
STORE R3, "adresse de A"



Langage de Haut Niveau

- Structures de données : Scalaires, Tableaux, Listes, Produits,…
- Affectations : A = B +C + D
- Contrôle
  - Séquence : Instruction 1 ; Instruction2
  - Conditionnelles
  - Boucles
  - Appels de procédures et fonctions

Types de données

- Un type décrit en général : encombrement mémoire et les opérateurs admissibles
- Ici les opérateurs sont matériels, donc les types sont :
  - Entiers signés et non signés : taille du chemin de données
  - Octets : 8 bits
  - Flottants : 32 ou 64 bits

Documents

- Toutes les information nécessaires sur le MIPS sont données dans le cours et en TD
- Documentation de référence
  - www.cs.cornell.edu/courses/cs3410/2008fa/**mips**_vol1.pdf
  - www.cs.cornell.edu/courses/cs3410/2008fa/**MIPS**_Vol2.pdf

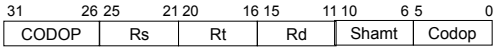
Plan

1. Le modèle de Von-Neumann
2. Format des instructions
   - format
   - codes opérations
   - opérandes
3. Instructions arithmétiques
4. Instructions d'accès mémoire
5. Instructions de contrôle
6. Procédures

Format des instructions

Qu'est-ce qu'une instruction ?

– Chaîne de bits respectant un format



000000 00001 00010 00011 00000 100000

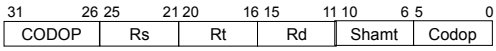
0x00221420

– Version lisible par un humain, « assembleur »

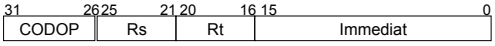
```
ADD R3,R1,R2
```


Formats MIPS

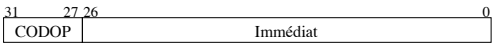
- Registre-Registre



- Registre-Immédiat



- Sauts



Opérandes

- 2 opérandes d'entrée et 1 résultat = 3 opérandes à décrire dans l'instruction
- Modes d'accès (« d'adressage ») à un opérande
  - Registre
    - l'opérande est dans un registre
    - le registre est décrit par son numéro, sur 5 bits
  - Immédiat
    - l'opérande est dans l'instruction, sur les 16 bits de poids faible
    - problématique, voir plus loin
  - Le résultat est toujours en registre !
- Format R-R : tous les opérandes en registre
- Format R-I : un des opérandes source est un immédiat

Quelques codops MIPS (1)

opcode	bits 28..26							
	0	1	2	3	4	5	6	7
bits 31..29	000	001	010	011	100	101	110	111
0 000	*SPECIAL* δ	*REGIMM* δ	*J*	JAL	BEQ	BNE	BLEZ	BGTZ
1 001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2 010	*COP0* δ	*COP1* δ	*COP2* θδ	*COP3* θδ	BEQL φ	BNEL φ	BLEZL φ	BGTZL φ
3 011	β	β	β	β	*SPECIAL2* δ	JALX ε	ε	*
4 100	LB	LH	LWL	LW	LBU	LHU	LWR	β
5 101	SB	SH	SWL	SW	β	β	SWR	CACHE
6 110	LL	LWC1	LWC2 θ	PREF	β	LDC1	LDC2 θ	β
7 111	SC	SWC1	SWC2 θ	*	β	SDC1	SDC2 θ	β

Quelques codops MIPS (2)

function	bits 2..0							
	0	1	2	3	4	5	6	7
bits 5..3	000	001	010	011	100	101	110	111
0 000	SLL	*MOVCI* δ	SRL	SRA	SLLV	*	SRLV	SRAV
1 001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2 010	MFHI	MTHI	MFLO	MTLO	β	*	β	β
3 011	MULT	MULTU	DIV	DIVU	β	β	β	β
4 100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5 101	*	*	SLT	SLTU	β	β	β	β
6 110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7 111	β	*	β	β	β	*	β	β

Qulques codops MIPS

rt	bits 18..16							
	0	1	2	3	4	5	6	7
bits 20..19	000	001	010	011	100	101	110	111
0 00	BLTZ	BGEZ	BLTZL φ	BGEZL φ	*	*	*	*
1 01	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2 10	BLTZAL	BGEZAL	BLTZALL φ	BGEZALL φ	*	*	*	*
3 11	*	*	*	*	*	*	*	*

Plan

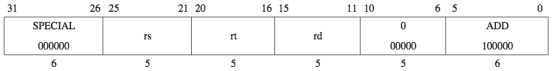
1. Le modèle de Von-Neumann
2. Format des instructions
3. Instructions arithmétiques et logiques
   - présentation
   - instructions arithmétiques
   - instructions logiques
   - décalages
4. Instructions d'accès mémoire
5. Instructions de contrôle
6. Procédures

Présentation

- Instructions arithmétiques
  - ADD, SUB et variantes : addition, soustraction arithmétiques
- Instructions logiques
  - AND, OR, XOR et variantes : opérations logiques bit à bit

Instructions d'addition (1)

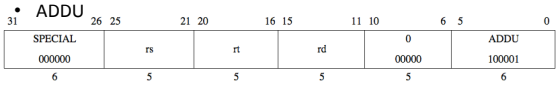
- ADD



  - Format R-R
  - Syntaxe ADD Rd, Rs, Rt
  - Effet Rd <- Rs + Rt si résultat représentable en interprétation en relatifs, trap on overflow sinon
  - Exemple ADD R1, R2, R3
    - Initialement R1= 0x00000000 et R2 =0x12345678
    - code 0b000000 00010 00011 00001 00000 100000 = 0x00430820
    - si initialement R3 = 0x00000001, effet R1 = 0X12345678
    - si initialement R3 = 0x70000000, effet R1 inchangé et exception

Instructions d'addition (2)

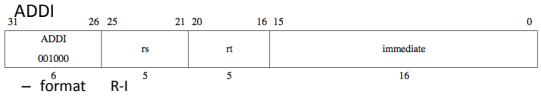
- ADDU



  - Format R-R
  - Syntaxe ADDU Rd, Rs, Rt
  - Effet Rd <- Rs + Rt au sens additionnage
  - Exemple ADD R1, R2, R3
    - Initialement R1= 0x00000000 et R2 =0x12345678
    - code 0b000000 00010 00011 00001 00000 100001 = 0x00430821
    - si initialement R3 = 0x00000001, effet R1 = 0X12345678
    - si initialement R3 = 0x70000000, effet R1 = 0x82345678

Instructions d'addition (3)

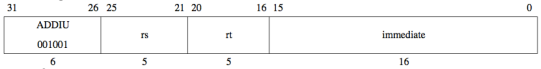
- ADDI



  - format R-I
  - Syntaxe ADDI Rt, Rs, Imm_{16}
  - Effet Rt <- Rs + $ES_{32}(Imm_{16})$ si résultat représentable en interprétation en relatifs, trap on overflow sinon
  - Exemples. Initialement R1= 0x00000000 et R2 = 0x12345678
    - ADDI R1, R2, 9
      - Code 0b001000 00010 00001 0...01001 = 0x20410009
      - Résultat R1 = 0x12345681
    - ADDI R1, R2, -1
      - Code 0b001000 00010 00001 1...1 = 0x2041FFFF
      - Résultat R1 = 0x12345677

Instructions d'addition (4)

- ADDIU



  - format R-I
  - Syntaxe ADDIU Rt, Rs, Imm_{16}
  - Effet Rt <- Rs + $ES_{32}(Imm_{16})$ au sens additionnage

Instruction addition R-I

Application : compiler l'évaluation des expressions contenant des constantes

Y = X + 12

Se compile en

< charger X dans R1>
ADDI R2, R1, 12
<ranger R2 dans Y>

Autres instructions arithmétiques

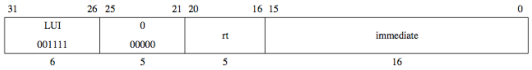
Non utilisées dans ce cours

Pseudo-instructions

Soustraction R-R

Arithmetic Operations			
ADD	RD, RS, RT	RD = RS + RT	(overflow trap)
ADDI	RD, RS, const16	RD = RS + const16*	(overflow trap)
ADDIU	RD, RS, const16	RD = RS + const16*	
ADDU	RD, RS, RT	RD = RS + RT	
CLO	RD, RS	RD = CountLeadingOnes(RS)	
CLZ	RD, RS	RD = CountLeadingZeros(RS)	
LA	RD, label	RD = Address(label)	
LI	RD, imm32	RD = imm32	
LUI	RD, const16	RD = const16 << 16	
MOVE	RD, RS	RD = RS	
NEGU	RD, RS	RD = –RS	
SEB[R2]	RD, RS	RD = RS7:0*	
SEH[R2]	RD, RS	RD = RS15:0*	
SUB	RD, RS, RT	RD = RS – RT	(overflow trap)
SUBU	RD, RS, RT	RD = RS – RT	

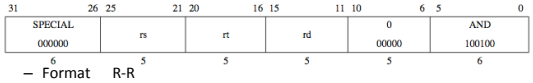
Instruction LUI



- Format R-I
- Syntaxe LUI Rt, Imm16
- Effet Rt <- Imm16||0x0000
- Exemple LUI R1, 0x1234 Effet R1 = 0x12340000
- Usage
  - Compiler les expressions faisant intervenir des constantes entière sur plus de 16 bits, par exemple Y= X + 0x12345678
  - En « assembleur », on utilise la pseudo-instruction LI, qui est implémentée à partir de LUI (voir TD). On ne peut pas avoir 32 bits dans une instruction 32 bits !

Instructions logiques (1)

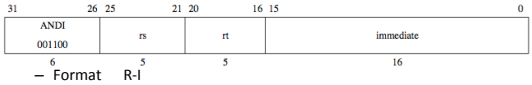
- AND



  - Format R-R
  - Syntaxe AND Rd, Rs, Rt
  - Effet Rd <- Rs AND Rt au sens bit-à-bit
  - Exemple AND R1, R2, R3
    - Initialement R1= 0x00000000 et R2 =0x12345678
    - si initialement R3 = 0x00000041, effet R1 = 00000040

Instructions logiques (2)

- ANDI



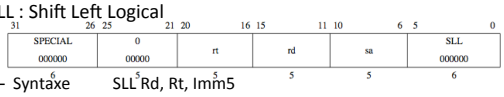
  - Format R-I
  - Syntaxe ANDI Rt, Rs, Imm16
  - Effet Rd <- Rs AND EZ(Imm16) au sens bit-à-bit
  - Exemple. Initialement R1= 0x00000000 et R2 =0x12345678
    - ANDI R1, R2, 0x0041 effet R1 = 0x00000040
    - ANDI R1, R2, 0xFF41 effet R1 = 0x00005640

Autres instructions logiques

Logical and Bit-Field Operations		
AND	R_D, R_S, R_T	$R_D = R_S \,\&\, R_T$
ANDI	R_D, R_S, const16	$R_D = R_S \,\&\, \text{const16}^{\varnothing}$
EXT[R2]	R_D, R_S, P, S	$R_S = R_{S\text{-}S\text{-}1:P}{}^{\varnothing}$
INS[R2]	R_D, R_S, P, S	$R_{D\text{-}S\text{-}1:P} = R_{S\text{-}S\text{-}1:0}$
NOP		No-op
NOR	R_D, R_S, R_T	$R_D = \sim(R_S \mid R_T)$
NOT	R_D, R_S	$R_D = \sim R_S$
OR	R_D, R_S, R_T	$R_D = R_S \mid R_T$
ORI	R_D, R_S, const16	$R_D = R_S \mid \text{const16}^{\varnothing}$
WSBH[R2]	R_D, R_S	$R_D = R_{S\text{-}23:16} :: R_{S\text{-}31:24} :: R_{S\text{-}7:0} :: R_{S\text{-}15:8}$
XOR	R_D, R_S, R_T	$R_D = R_S \oplus R_T$
XORI	R_D, R_S, const16	$R_D = R_S \oplus \text{const16}^{\varnothing}$

Instructions de décalage (1)

- Format R-I, bits 21-25 non significatifs (0)
- SLL : Shift Left Logical



  - Syntaxe SLL Rd, Rt, Imm5
  - Effet Rd <- Rt décalé à gauche de Imm5 interprété en non signé
    NB: Multiplication non signée par 2^{Imm5} si le résultat est représentable
  - Exemple
    SLL R2, R1, 8
    Initial R1 = 0x00070000
    Résultat R2 = 0x07000000

Instructions de décalage (2)

- SRL : Shift Right Logical
  - Syntaxe SRL Rd, Rt, Imm5
  - Effet Rd <- Rt décalé à droite de Imm5, avece extension à 0
    NB: Division en entiers naturels par 2^{Imm5} si représentable
- SRA : Shift Right Arithmetic
  - Syntaxe SRL Rd, Rt, Imm5
  - Effet Rd <- Rt décalé à droite de Imm5, avec extension de signe
    NB: Division en entiers relatifs par 2^{Imm5} si représentable
- Exemples
  - SRL R1, R2, 12 avec R2=0xFFFFFFFF, résultat R1 = 0x000FFFFF
  - SRA R1, R2, 12 avec R2=0xFFFFFFFF, résultat R1 = 0xFFFFFFFF
- Variantes R-R : SLLV, SRLV, SRAV

Instructions conditionnelles

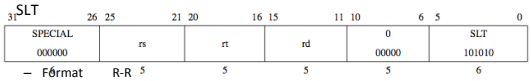
En MIPS, limitées aux MOV

- MOVN
  - Syntaxe MOVN Rd, Rs, Rt
  - Effet si Rt <>0 Rd <- Rs
- MOVZ
  - Syntaxe MOVZ Rd, Rs, Rt
  - Effet si Rt == 0 Rd <- Rs

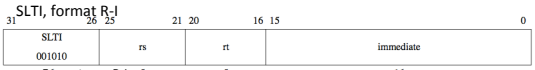
```
Min_Signe (R8, R9)

SLT     R1, R8, R9
MOVN R9, R8, R1
```

Calcul de condition

- SLT



  - Format R-R
  - Syntaxe SLT Rd, Rs, Rt
  - Effet si Rs < Rt en signé, Rd <- 1, sinon Rd <- 0. Pas de trap on overflow
- SLTI, format R-I



  - Format R-I
  - Syntaxe SLTI Rt, Rs, Imm16
  - Effet si Rs < Imm16 en signé, Rd <- 1, sinon Rd <- 0. Pas de trap on overflow
- SLTU, SLTIU : comparaison non signée

"L'assembleur"

- Représentation alphanumérique du format, pour la commodité de l'interaction homme-machine, par exemple ADD R1,R2,R3
  - Programmation directe – EN TP
  - Résultat de la compilation (sous Unix, Option -S)
- Ou programme de codage : alphanum -> binaire.
  - Unix *as*

Plan

1. Le modèle de Von-Neumann
2. Format des instructions
3. Instructions arithmétiques
4. Instructions d'accès mémoire
   - chargement
   - rangement
   - représentation des variables en mémoire
5. Instructions de contrôle
6. Procédures

Instruction d'accès mémoire

- Chargement : de la mémoire vers le processeur
- Rangement : du processeur vers la mémoire
- MIPS : uniquement format R-I
- Point délicat :
  - la mémoire est organisée par octets
  - mais le chemin de données est 32 bits

Représentation des variables en mémoire

- Dans quel ordre placer les variables multi-octets ?

int x = 0x12345678 ;

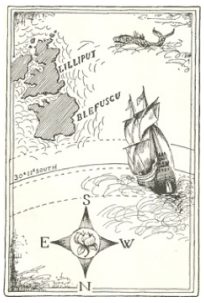
Mémoire organisée Big Endian

0x00001000	12
0x00001001	34
0x00001002	56
0x00001003	78

Mémoire organisée Little Endian

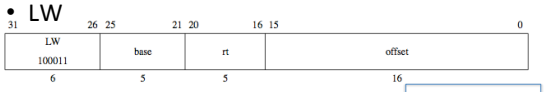
0x00001000	78
0x00001001	56
0x00001002	34
0x00001003	12

Endian, ~~Indian~~



Chargement

- LW



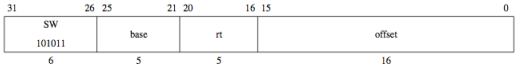
  - Syntaxe LW Rt, Imm16 (base)
  - Effet Rd <- Mem_{32} [base + ES(Imm16)]
  - NB: alignement (voir + loin)
  - Exemples
    - LW R2, 0(R1) R2 <- 0x12345678
    - LW R2, 4(R1) R2 <- 0xABCDEF09

Etat initial
R1 = 0x00001000

12	*1000*
34	*1001*
56	*1002*
78	*1003*
AB	*1004*
CD	*1005*
EF	*1006*
09	*1007*

Rangement

- SW



  - Syntaxe SW Rt, Imm16 (base)
  - Effet Mem_{32} [base + ES(Imm16)]<- Rt
  - Alignement

Instructions d'accès mémoire

Etat initial

R1 = 0x00001000
R2 = 0xEF125634

12	*1000*
34	*1001*
56	*1002*
78	*1003*
AB	*1004*
CD	*1005*
EF	*1006*
09	*1007*

Effet des instructions

SW R2, 0(R1)

EF	*1000*
12	*1001*
56	*1002*
34	*1003*
AB	*1004*
CD	*1005*
EF	*1006*
09	*1007*

Autres instructions d'accès mémoire

Load and Store Operations		
LB	Rd, off16(Rs)	Rd = mem8(Rs + off16")*
LBU	Rd, off16(Rs)	Rd = mem8(Rs + off16")@
LH	Rd, off16(Rs)	Rd = mem16(Rs + off16")*
LHU	Rd, off16(Rs)	Rd = mem16(Rs + off16")@
LW	Rd, off16(Rs)	Rd = mem32(Rs + off16")
LWL	Rd, off16(Rs)	Rd = LoadWordLeft(Rs + off16")
LWR	Rd, off16(Rs)	Rd = LoadWordRight(Rs + off16")
SB	Rs, off16(Rt)	mem8(Rt + off16") = Rs7:0
SH	Rs, off16(Rt)	mem16(Rt + off16") = Rs15:0
SW	Rs, off16(Rt)	mem32(Rt + off16") = Rs
SWL	Rs, off16(Rt)	StoreWordLeft(Rt + off16", Rs)
SWR	Rs, off16(Rt)	StoreWordRight(Rt + off16", Rs)
ULW	Rd, off16(Rs)	Rd = unaligned_mem32(Rs + off16")
USW	Rs, off16(Rt)	unaligned_mem32(Rt + off16") = Rs

Représentation des variables en mémoire

- Les types élémentaires des LHN correspondent à un nombre précis d'octets

Caractères	*char*	1
Entiers courts	*short*	2
Entiers	*int*	4
Flottants simple précision	*float*	4
Flottants double précision	*double*	8

- Alignement : à la compilation, les variables sont alignées sur leurs frontières naturelles = adresses multiples de leurs tailles

Représentation des variables en mémoire

- Tableaux : stockés séquentiellement à partir de l'adresse du premier élément

Adresse(tab[i]) = Adresse(tab[0]) +i*sizeof(élément_tableau)

```
int v [N], k, temp ;
temp = v[k] ;
v[k] = v [k+1];
v[k+1] = temp
```

Assembleur

```
<Initialement, R1 = @v[0], R2 = k>
SLL     R2, R2, 2    ; deplacement <- k*4
ADD     R2, R2, R1   ; R2 <- @ v[k]
LW      R3, 0(R2)    ;  temp = R3 <- v[k]
LW      R4, 4(R2)    ; R4 <- v[k+1]
SW      R4, 0(R2)    ; v[k] <- R4
SW      R3, 4(R2)    ; v[k+1] <- temp
```

Plan

1. Le modèle de Von-Neumann
2. Format des instructions
3. Instructions arithmétiques
4. Instructions d'accès mémoire
5. Instructions de contrôle
6. Procédures

Instructions de branchement

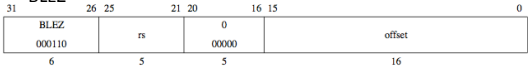
- Comment réaliser les ruptures de séquence ?
- Branchements : déplacement relatif

  PC <- PC + depct
- Sauts

  PC <- adresse
- NB :
  - PC pointe l'instruction qui suit le branchement
  - Les instructions occupent 4 octets et sont alignées
  - En MIPS vrai, l'instruction qui suit le branchement ou le saut (delay slot) est exécutée. Le cours et les TD ne sont pas conformes au MIPS sur ce point.

Instructions de branchement (1)

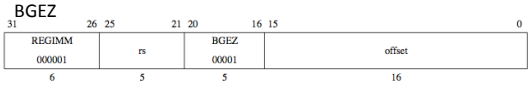
- BLEZ



  - Format R-I
  - Syntaxe BLEZ Rs, Imm16
  - Effet si Rs <= 0, PC <- PC + ES(Imm16||0b00)
  - Exemple
    - BLEZ R0, IMM16 est un branchement inconditionnel
    - SI BLEZ est à l'adresse 0x10000000, BLEZ R0, 16 branche à l'adresse 0x10000044

Instructions de branchement (2)

- BGEZ



  - Format R-I
  - Syntaxe BGEZ Rs, Imm16
  - Effet si Rs>=0, PC <- PC + ES(Imm16||0b00)
  - Exemple : placer dans R1 la valeur absolue de R2

```
        MOV R1, R2
        BGEZ R1, suite
        NEGU R1, R1
suite   …
```


Autres instructions de branchement

Jumps And Branches (Note: One Delay Slot)		
B	off18	PC += off18*
BAL	off18	RA = PC + 8, PC += off18*
BEQ	Rs, Rt, off18	if Rs = Rt, PC += off18*
BEQZ	Rs, off18	if Rs = 0, PC += off18*
BGEZ	Rs, off18	if Rs ≥ 0, PC += off18*
BGEZAL	Rs, off18	RA = PC + 8; if Rs ≥ 0, PC += off18*
BGTZ	Rs, off18	if Rs > 0, PC += off18*
BLEZ	Rs, off18	if Rs ≤ 0, PC += off18*
BLTZ	Rs, off18	if Rs < 0, PC += off18*
BLTZAL	Rs, off18	RA = PC + 8; if Rs < 0, PC += off18*
BNE	Rs, Rt, off18	if Rs ≠ Rt, PC += off18*
BNEZ	Rs, off18	if Rs ≠ 0, PC += off18*

Compilation des conditionnelles

```
Si <condition>
alors
    Bloc1
sinon
    Bloc2
```

```
        Bcond lvrai
        Instructions du bloc 2
        BA lsuite
lvrai:  Instructions Bloc1
suite:  …
```

Boucles

- Tant que : un branchement conditionnel + un branchement inconditionnel
- Répéter : un seul branchement
- Les boucles for avec des constantes sont des compilées comme des boucles répéter

```
BCL :   <bcht cond fausse> SUITE
        <corps boucle>
        <bcht incond> BCL
```

```
BCL :   <corps boucle>
        <bcht cond vraie> BCL
```


Boucles

```
int a[100], b[100];
for (i=0; i < 100; i++) a[i] = b[i] +10
```

```
      ADDI R6,R5,400 # R4 <- première adresse invalide
loop: LW R7,0(R5)    # R7 <- b[i]
      ADDI R7,R7,10  # R7 <- b[i]+10
      SW R7,0(R4)    # a[i] <- R7
      ADDI R5,R5,4   # i++ pour b
      ADDI R4,R4,4   # i++ pour a
      BNE R5,R6,loop # branche si i < n
```

R5 = @b[0]
R4 = @a[0]

Boucles

```
char a[100], b[100];
for (i=0; i < 100; i++) a[i] = b[i] +10
```

```
      ADDI R6,R5,100 # R4 <- première adresse invalide
loop: LB R7,0(R5)    # R7 <- b[i]
      ADDI R7,R7,10  # R7 <- b[i]+10
      SB R7,0(R4)    # a[i] <- R7
      ADDI R5,R5,1   # i++ pour b
      ADDI R4,R4,1   # i++ pour a
      BNE R5,R6,loop # branche si i < n
```

R5 = @b[0]
R4 = @a[0]

Boucles

```
int a[100], b[100];
for (i=0; i < 99; i++) a[i+1] = b[i] +10
```

```
      ADDI R6,R5,396 # R4 <- première adresse invalide
loop: LW R7,0(R5)    # R7 <- b[i]
      ADDI R7,R7,10  # R7 <- b[i]+10
      SW R7,4(R4)    # a[i] <- R7
      ADDI R5,R5,4   # i++ pour b
      ADDI R4,R4,4   # i++ pour a
      BNE R5,R6,loop # branche si i < n
```

R5 = @b[0]
R4 = @a[0]

Types de branchements

- MIPS : branchement sur registre
- RCC + Instructions conditionnelles : ARM

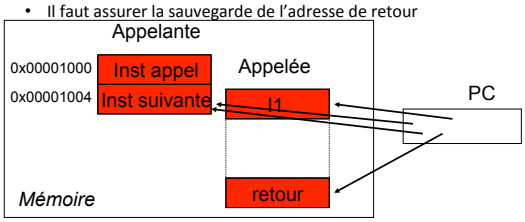
Plan

1. Le modèle de Von-Neumann
2. Format des instructions
3. Instructions d'accès mémoire
4. Instructions arithmétiques
5. Instructions de contrôle
6. Procédures

Procédures

Appel et retour de procédure

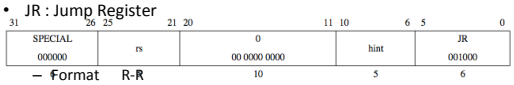
- La procédure peut être à une adresse éloignée
- L'appelante connaît l'adresse de l'appelée, mais le contraire n'est pas vrai
- Il faut assurer la sauvegarde de l'adresse de retour



Support pour l'appel et le retour de procédure

- JAL: Jump and Link

  - Format Sauts
  - Syntaxe JAL Imm26
  - Effet R31 <- PC+4 ; PC <- PC31:28||Imm26||0b00
  - Aussi JALR, format R-R
- JR : Jump Register



  - Format R-R
  - Syntaxe JR Rs
  - Effet PC<- Rs
  - Usage typique : JR R31

Pile d'exécution

- Appels de procédures imbriqués : l'adresse de retour serait perdue
- Les procédures peuvent modifier certains registres
- Conventions logicielles sur
  - Le stockage en mémoire de l'adresse de retour et des paramètres
  - Le registre pointeur de pile : R30
  - l'usage des registres :
    - R8-R15 et R25-R26 ne sont pas sauvegardés par l'appelée
    - R16-R23 doivent être sauvegardés
    - **R4-R7 passage des paramètres, R2 valeur de retour**

Procédure terminale – version simplifiée

```
int foo (int x, int y) {
return (x+y);
}

void main () {
int a, b,c;
….
c= foo (a, b);
printf (« %d », c);
}
```

```
foo:
    ADD R2, R4,R5
# a est passé par R4, b par
R5, valeur de retour dans R2
    JR R31

main:
    … # R4 <-a, R5 <-b
    JAL foo
    MOVE R1, R2
    JAL printf
```


Procédure non terminale – version simplifiée

```
int foo (int x, int y) {
    return (x+y);
}

void bar() {
int a, b,c;
    ….
    c= foo (a, b);
    printf (« %d », c);
}

void main() {
 bar();
}
```

```
foo:
    ADD R2, R4,R5
# a passé par R4, b par R5,
valeur de retour dans R2
    JR R31

bar:
    … # R4 <-a, R5 <-b
    JAL foo
    MOVE R1, R2
    JAL printf
```


Procédure non terminale – version simplifiée

```
int foo (int x, int y) {
    return (x+y);
}

void bar() {
int a, b,c;
    ….
    c= foo (a, b);
    printf (« %d », c);
}

void main() {
 bar();
}
```

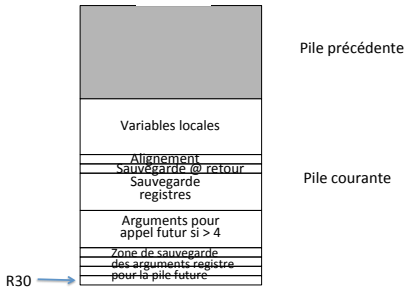
```
foo:
    ADD R2, R4,R5
# a passé par R4, b par R5,
valeur de retour dans R2
    JR R31

bar:
    … # R4 <-a, R5 <-b
    JAL foo
    MOVE R1, R2
    JAL printf
```

R31 = @ MOVE !!!

Pile d'exécution

- Appels de procédures imbriqués : l'adresse de retour serait perdue
- Les procédures peuvent modifier certains registres
- Conventions logicielles sur
  - Le stockage en mémoire de l'adresse de retour et des paramètres
  - Le registre pointeur de pile : R30
  - l'usage des registres :
    - R8-R15 et R25-R26 ne sont pas sauvegardés par l'appelée
    - R16-R23 doivent être sauvegardés
    - R4-R7 passage des paramètres, R2 valeur de retour

Pile d'exécution

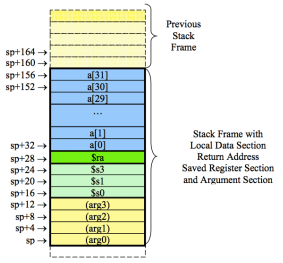


Exemple

```
int g( int x, int y ) {
    int a[32];
    ... (calculs utilisant x, y, a);
    a[1] = f(y,x,a[2]);
    a[0] = f(x,y,a[1]);
    return a[0]; }
```

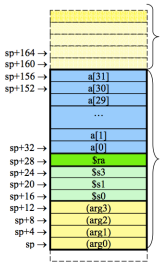
g:
# start of prologue
# push stack frame
addiu $sp,$sp,(-160)
# save registers
sw $ra,28($sp) #R31
sw $s0,16($sp) #R16
sw $s1,20($sp) #R17
sw $s3,24($sp) #R18
# end of prologue

```
int g( int x, int y ) {
    int a[32];
    ... (calculs utilisant x, y, a);
    a[1] = f(y,x,a[2]);
    a[0] = f(x,y,a[1]);
    return a[0]; }
```



g:
start of body
... (computations)
# save $a0 and $a1 in caller's stack
sw $a0,160(sp) # sv R4(var x)
sw $a1,164(sp) # sv R5(var y)
# first call to function f
lw $a0,164(sp) # arg0 is y
lw $a1,160(sp) # arg1 is x
lw $a2,40(sp) # arg2 is a[2]
jal f # call f
# store value of f into a[0]
sw $v0,36(sp) #R2

```
int g( int x, int y ) {
    int a[32];
    ... (calculs utilisant x, y, a);
    a[1] = f(y,x,a[2]);
    a[0] = f(x,y,a[1]);
    return a[0]; }
```



g:
…
# start of epilogue
# restore value of saved registers
lw $s0,16($sp)
lw $s1,20($sp)
lw $s3,24($sp)
# restore the return address
lw $ra,28($sp)
# pop stack frame
addiu $sp,$sp,160
# end of epilogue
jr $ra # return

```
int g( int x, int y ) {
    int a[32];
    ... (calculs utilisant x, y, a);
    a[1] = f(y,x,a[2]);
    a[0] = f(x,y,a[1]);
    return a[0]; }
```