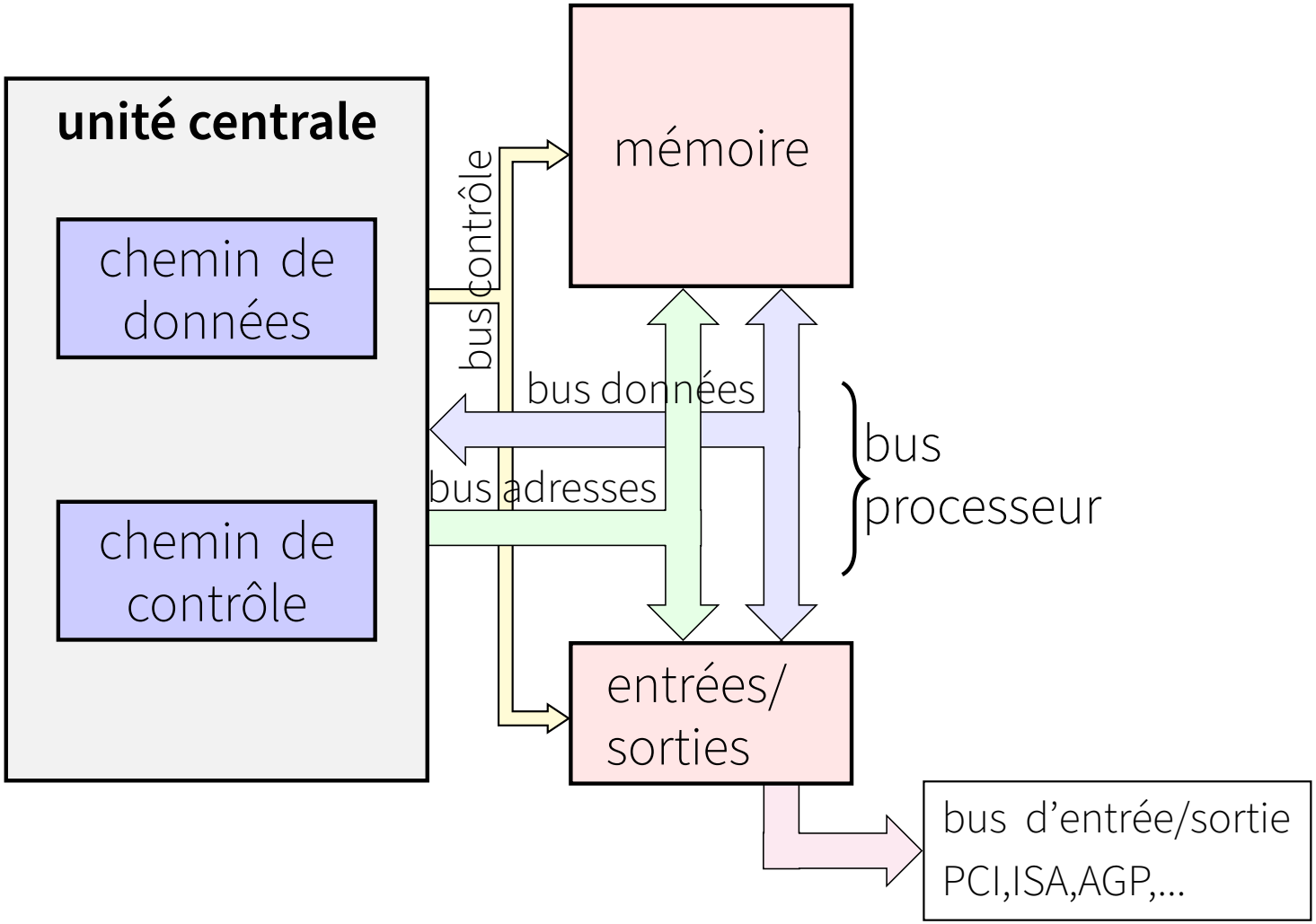
T1 Architecture des processeurs

Les jeux d'instructions

A. Mérigot

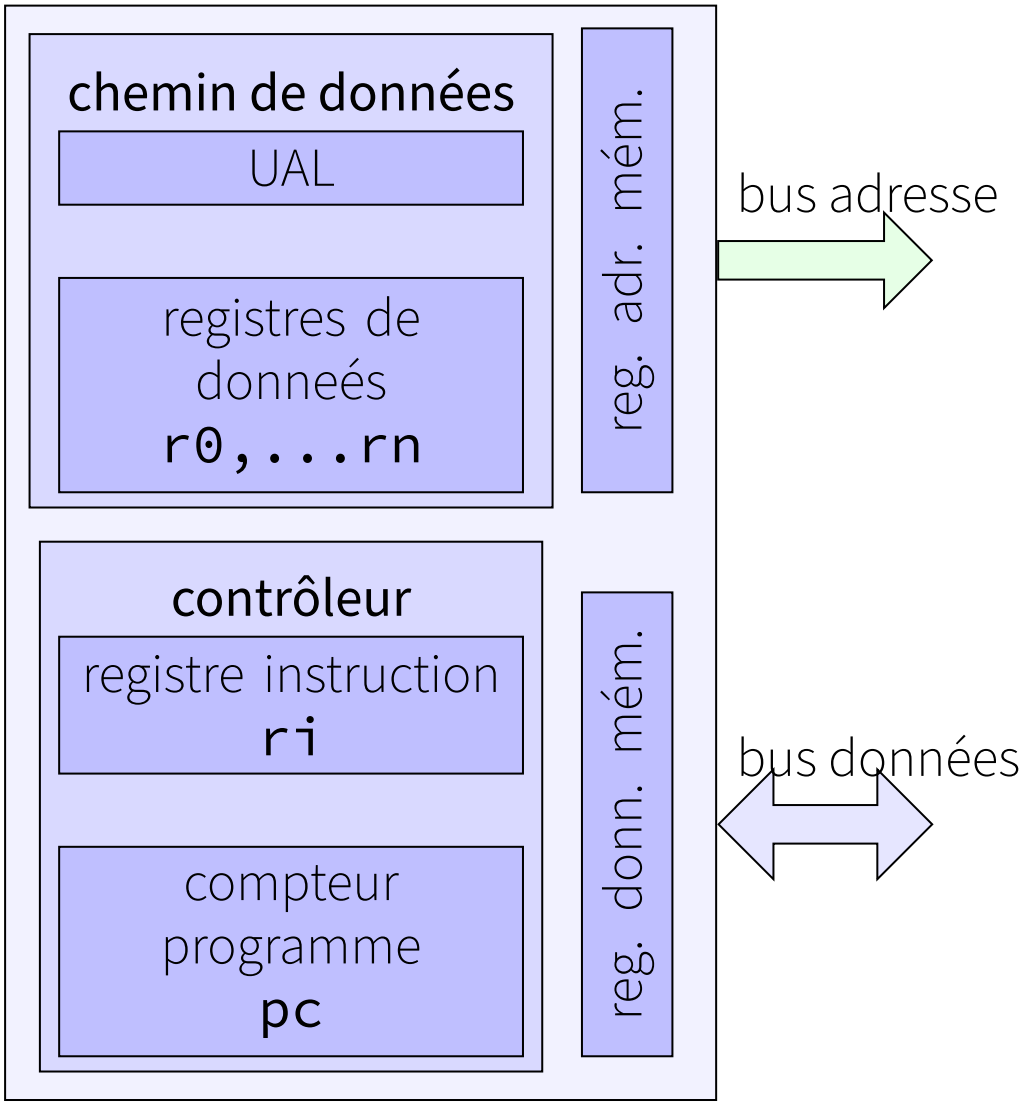
M2 SETI 2022-23

Organisation d'un ordinateur



Organisation générale d'un ordinateur :
unité centrale (UC ou CPU) + mémoire + entrées-sorties communiquant par des *bus*.

Organisation d'un ordinateur (cont.)



Principaux organes d'une unité centrale

Contrôleur : séquence l'exécution des instructions

- compteur de programme (`pc`) contient l'adresse de la prochaine instruction à exécuter
- registre d'instruction `ri`

Chemin de données : effectue les calculs.

- unité arithmétique et logique
- registres de données

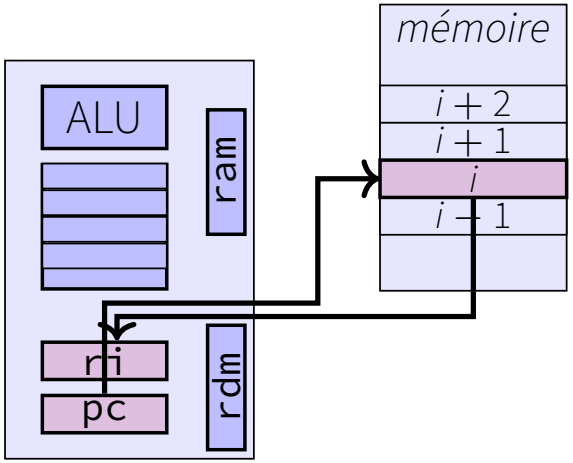
Exécution d'une instruction

L'exécution d'une instruction nécessite une suite d'actions.

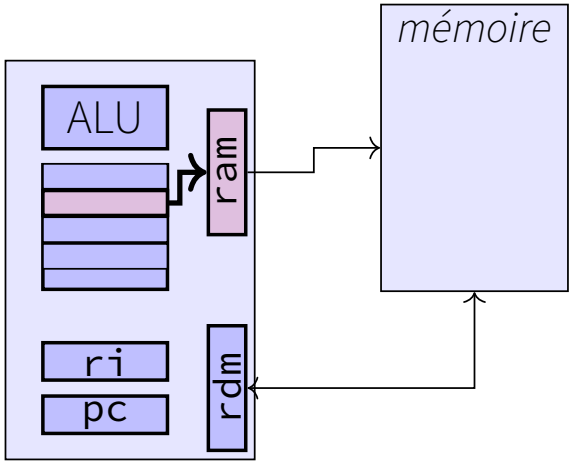
- aller chercher la prochaine instruction en mémoire (à l'adresse `pc`)
- incrémenter `pc` pour préparer l'instruction suivante
- décoder l'instruction
- aller chercher les opérandes
- effectuer le calcul
- ranger le résultat

Exécution d'une instruction (cont.)

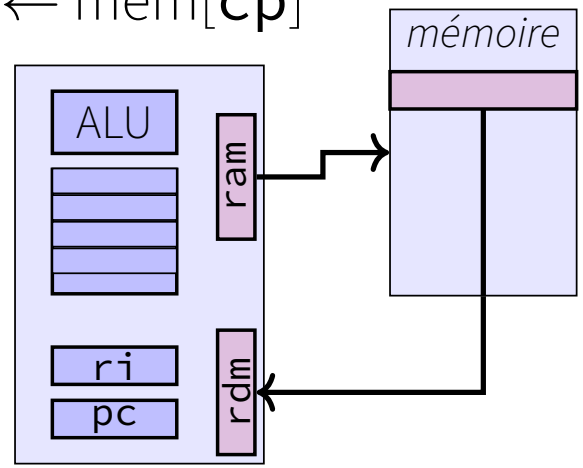
`load r4, (r2)` (réalise `r4` ← mem[`r2`])



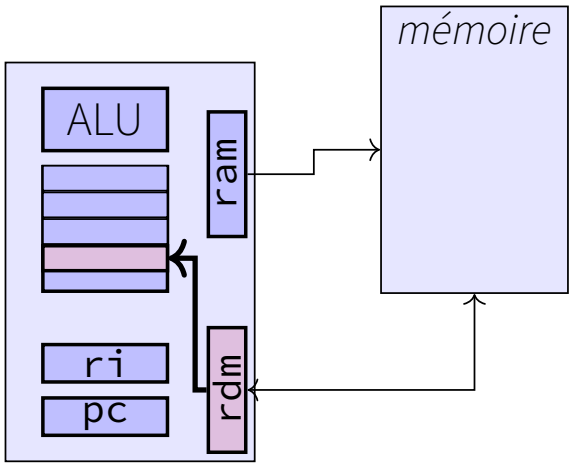
1. acquisition instruction
   `ri` ← mem[`cp`]



2. `ram` ← `r2`



3. `rdm` ← mem[`ram`]



4. `r4` ← `rdm`

Exécution d'une instruction (cont.)

pipeline

Depuis les années 80, ces différentes opérations s'effectuent en *pipeline*.

L'exécution de chaque instruction est découpée en *étages*.

Différentes instructions s'exécutent simultanément à différents étages.

On peut lancer 1 instruction par cycle.

$$t_e = n_i \times \overline{CPI} \times T_c$$

Réduit fortement le CPI des instructions (idéalement 1 cy par instruction).

Principales étapes

Instruction ALU	Instruction mémoire	Branchement
Lecture instruction & increm. **pc**	Lecture instruction & increm. **pc**	Lecture instruction & increm. **pc**
Décodage instr. & lecture opérandes	Décodage instr. & lecture opérandes	Décodage instr. & calcul adr. brcht & copie dans **pc**
Exécution	Calcul adresse	
	Accès mémoire	
Rangement résultat	Rangement résultat	

Modèles d'exécution et jeux d'instruction

Modèles d'exécution (*n*,*m*)

- *n* : nombre d'opérandes explicites (source(s) et destination) par instruction arithmétique et logique
- *m* : nombre d'opérandes mémoire par instruction arithmétique et logique

Influence la structure du jeu d'instruction (*instruction set architecture* ou ISA)

RISC : (3,0)

- les instructions arithmétiques et logiques ne concernent que les registres
- `load` et `store` : seules instructions mémoire
- Instructions de longueur fixe

IA-32 (x86) : (2,1)

- instructions mélangeant accès mémoire et calcul

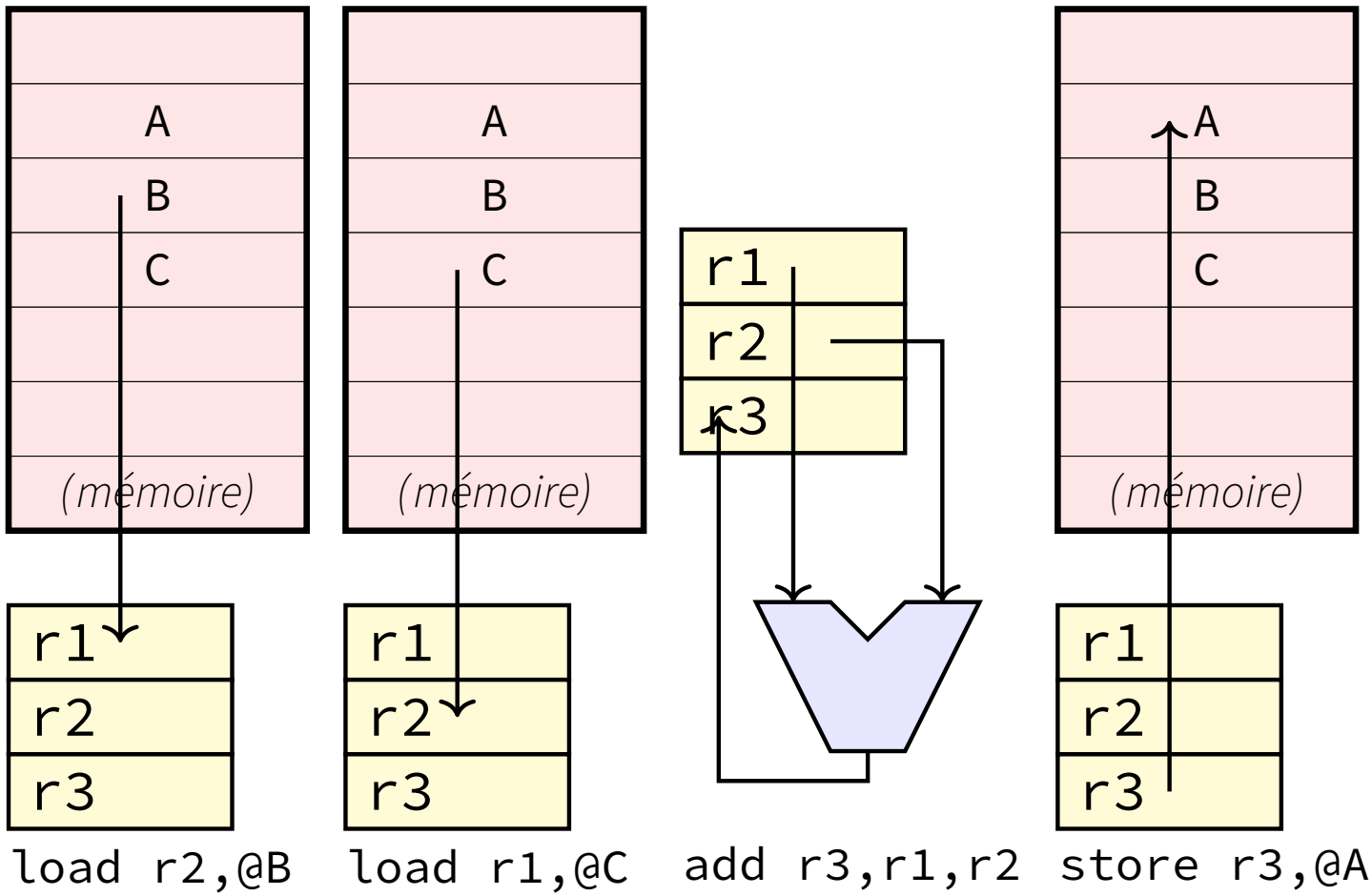
Pile (0,0)

- tous les opérandes sont accédés via la pile

Modèle d'exécution RISC (3,0)

Egalement appelé *registre–registre* ou *load-store* (mips, power-pc, arm, nios, risc-V, etc)
Il n'y a d'accès mémoire que pour les instructions `load` et `store`
Exemple : `A = B + C`

```
load  r1,@B
load  r2,@C
add   r3,r1,r2
store r3,@A
```



Modèle d'exécution RISC (3,0) (cont.)

Registres dans le modèle RISC

- 32 registres généraux (entiers) `r0` à `r31`
  - pas de rôle matériel spécifique pour la plupart des registres
  - *conventions logicielles* pour cohérence entre sous-programmes, système, etc.
- 32 registres flottants `f0` à `f31`
- instructions UAL et mémoire
  - registre – registre
    Rd ← **Rs1** op **Rs2**
  - registre – immédiat
    Rd ← **Rs1** op *immédiat*
  - accès mémoire
    Rd ↔ Mémoire (**Rs1** + dépl.)

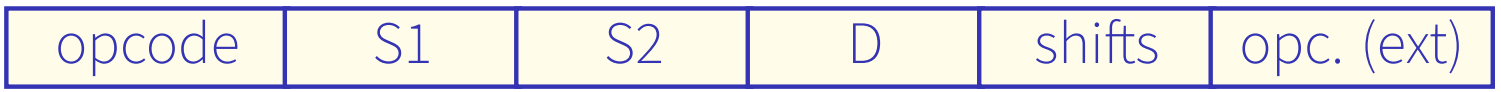
Registres MIPS

r0	$zero(=0)	r16	$s0
r1	$at	r17	$s1
r2	$v0	r18	$s2
r3	$v1	r19	$s3
r4	$a0	r20	$s4
r5	$a1	r21	$s5
r6	$a2	r22	$s6
r7	$a3	r23	$s7
r8	$t0	r24	$t8
r9	$t1	r25	$t9
r10	$t2	r26	$k0(réservé)
r11	$t3	r27	$k1(réservé)
r12	$t4	r28	$gp
r13	$t5	r29	$sp
r14	$t6	r30	$fp
r15	$t7	r31	$ra(ret. ad.)

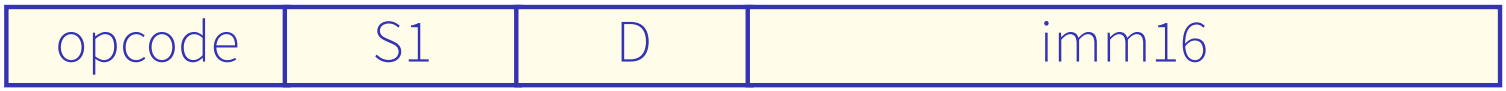
Modèle d'exécution RISC (3,0) (cont.)

Formats d'instructions RISC : UAL et mémoire

MIPS

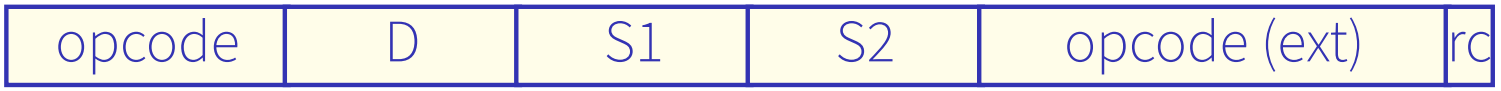


RR

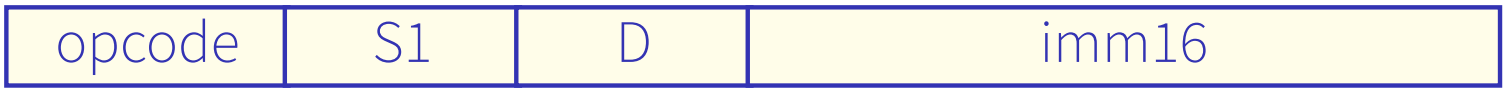


RI

Power PC



RR



RI

Modèle d'exécution RISC (3,0) (cont.)

Formats d'instructions RISC : Branchements et sauts

On distingue *branchements conditionnels* (BC) (adresse relative à **PC**)
et *sauts inconditionnels* (*Jump*)/appels de procédure (*Call*) (adresse absolue).

MIPS

BC

J,C

Power PC

BC

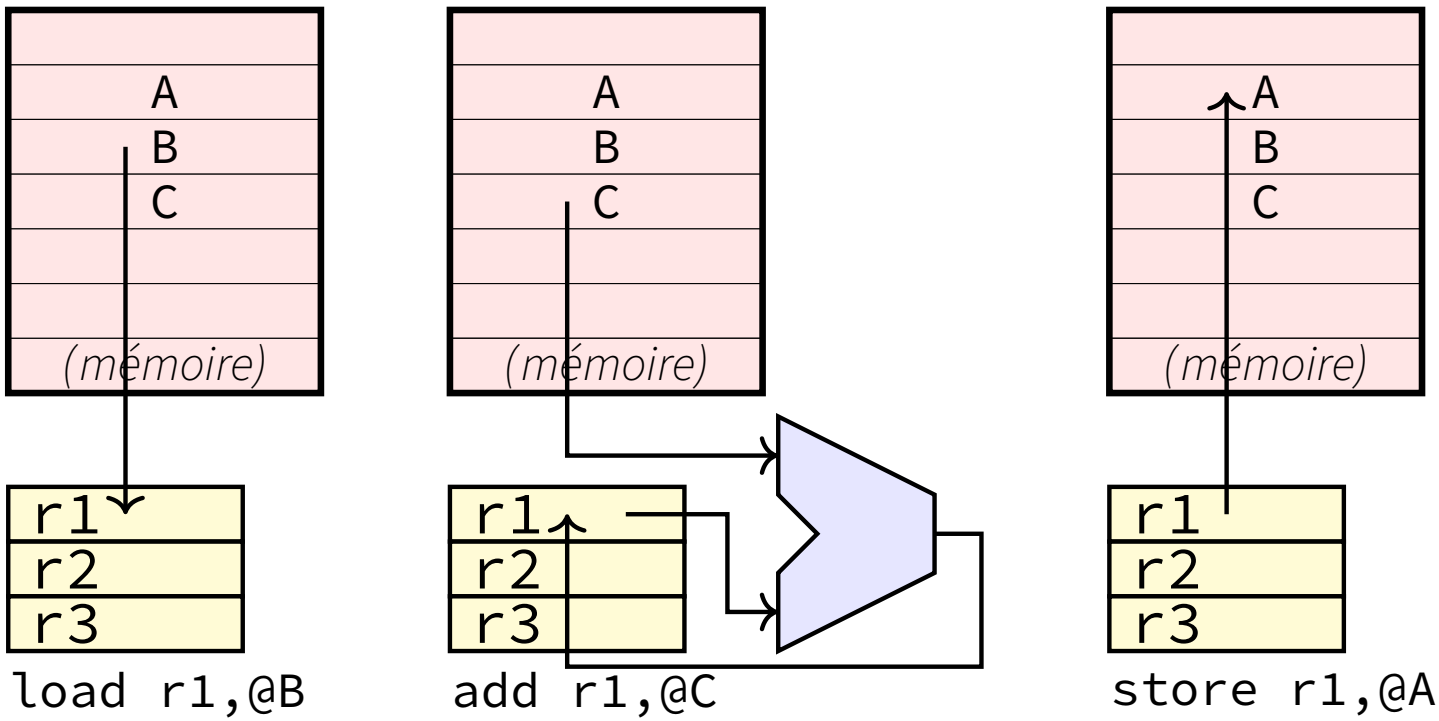
J,C

MIPS : les branchements combinent une comparaison entre deux registres
Power PC : on utilise les résultats d'une comparaison précédente stockée dans un registre condition

Modèle d'instructions (2,1) (IA-32 et x86-64)

2 opérandes explicites/instruction arithmétique et logique
le premier opérande peut servir de source et de destination
un des opérandes peut être en mémoire
Modèle également appelé *registre-mémoire*
Exemple : `A = B + C`

```
load  r1,@B
add   r1,@C
store r1,@A
```



Modèle d'instructions (2,1) (IA-32 et x86-64) (cont.)

Jeux d'instruction IA-32 (Intel Architecture 32 bits) (processeurs x86)
Extension 64 bits depuis pentium *core* x86-64
Instructions de longueur variable : de 1 à 6 octets, plus des « préfixes » (au maximum 15 octets)

OpCode	Reg. et MA	Déplacement	Immédiat
1 ou 2 o.	1 ou 2 o.	1, 2 ou 4 o.	1,2 ou 4 o.

- instr. dest source (dest ← dest op source)
  - `reg reg`
  - `reg mem`
  - `reg imm`
  - `mem reg`
  - `mem imm`
- instruction complexes (repétitions)
- mode d'adressage complexe :
  adr. mém. = `Rb` + `Ri` × scale(1,2,4 ou 8) + dépl(1,2 ou 4 o)

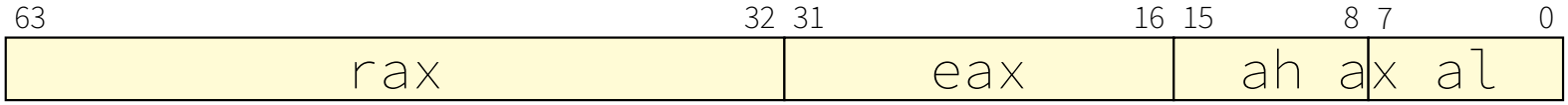
Les instructions `mem reg` ou `mem imm` nécessitent :

- lecture mémoire
- exécution
- écriture mémoire

Modèle d'instructions (2,1) (IA-32 et x86-64) (cont.)

Organisation non homogène

- 8 registres « généraux »
  - `ax`,`bx`,`cx`,`dx`,`si`,`di`, `sp`,`bp`
  - peuvent être utilisés en 8, 16, 32 ou 64 bits suivant leur nom

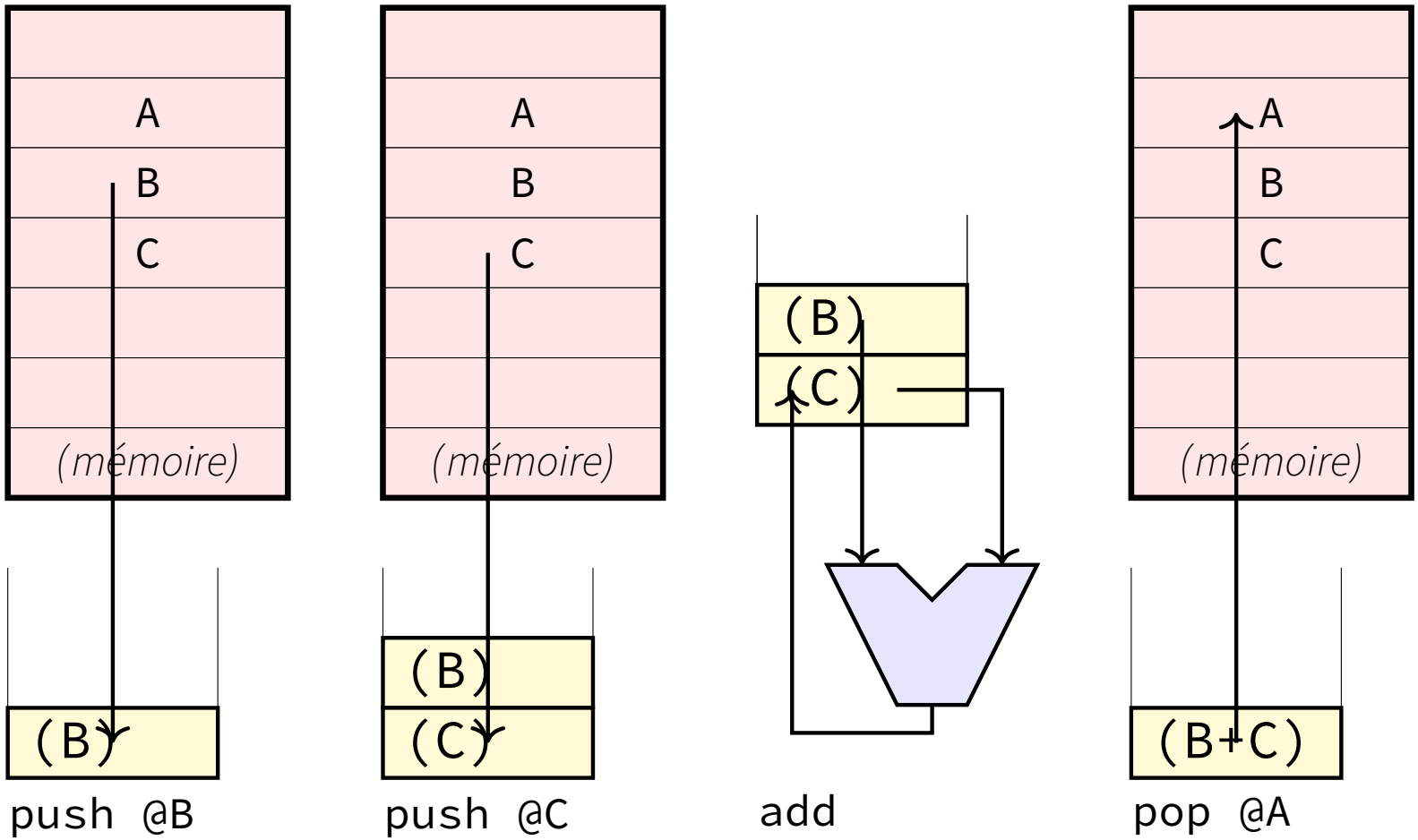


  - peuvent avoir un rôle spécifique suivant les instructions
    - `ax` *accumulateur*, destination par défaut de certaines opérations
    - `bx` *base*, décalages accès mémoire
    - `cx` *counter*, compteur pour instructions itérées
    - `dx` *data*, destination par défaut de certaines opérations
    - `di`, `si`, *index*, utilisés comme destination et source pour les instructions de manipulation de chaînes de caractères
    - `sp`, `bp` jouent le rôle de *pointeur* de pile et de trame
- registres flottants fonctionnant en pile (x87)
- registres « SIMD » (MMX, SSE-SSE2-SSE3-SSE4, AVX-AVX2-AVX512)

Modèle d'instructions en pile

Modèle « pile » (0,0) : registres organisés en pile, transferts entre mémoire et haut de la pile, calculs entre opérandes en haut de la pile.
Mise en oeuvre de `A = B + C`

```
push @B
push @C
add
pop  @A
```



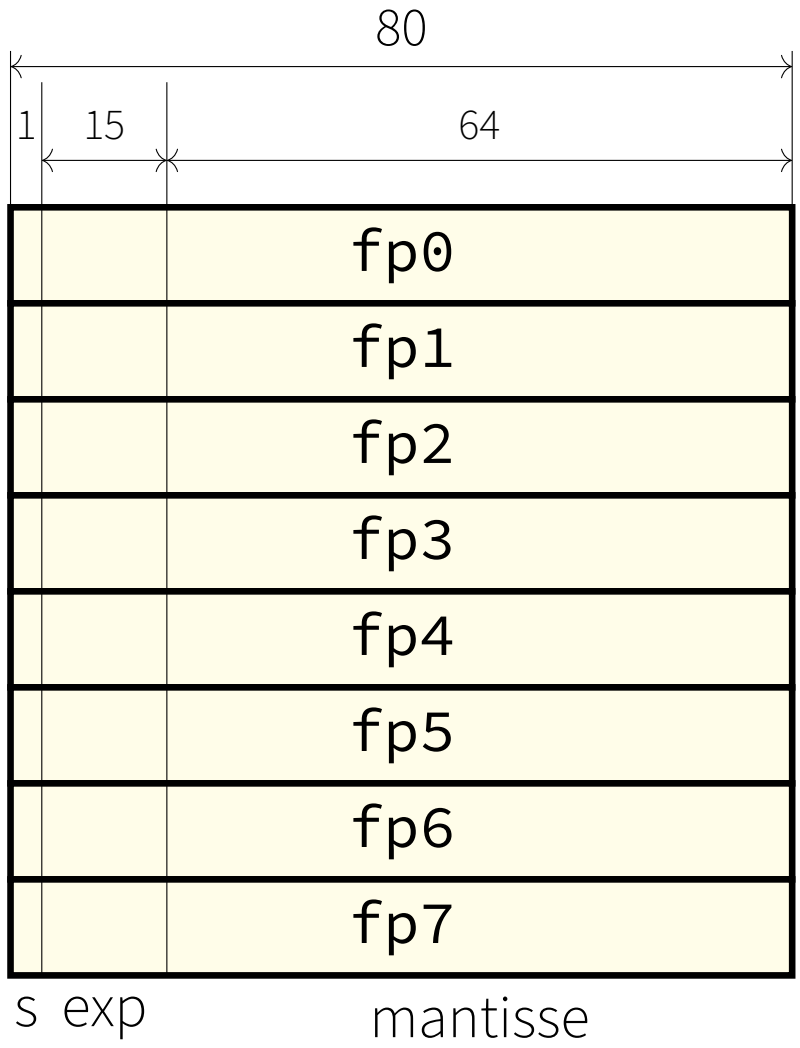
Utilisé dans quelques CPUs historiques, machine virtuelle java, calcul flottant IA-32.

Modèle d'instructions en pile (cont.)

instruction flottantes IA-32

Instructions flottantes IA-32

- matériel pour calcul rapide sur nombres flottants
  - opérations flottantes en parallèle avec les opérations entières
  - historiquement implanté comme un coprocesseur (8087-80387)
  - CPU et FPU échangent les données via la mémoire
- 8 registres spécifiques 80 bits (IEEE-754 *extended precision*), organisés en *pile*



Modèle d'instructions en pile (cont.)

instruction flottantes IA-32

Exemples de codes opérations flottants 80x87 FPU

- `fadd` : addition
- `fsub` : soustraction
- `fmul` : multiplication
- `fdiv` : division
- `fdivr` : division
- `fsin` : sinus (radians)
- `fcos` : cosinus (radians)
- `fsqrt` : racine carrée
- `fabs` : valeur absolue
- `fyl2x` : calcule Y * log2(X)
- `fyl2xp1` : calcule Y * log2(X+1)

Certaines instructions existent en version *pile* (`faddp`,`fsubp`, etc) et *registre* (`fadd`,`fsub`, etc).

Modèle d'instructions en pile (cont.)

instruction flottantes IA-32

- `st`=`st(0)`=sommet de pile
  Utilisation de `st(0)` implicite si non spécifié
- `fld` @Var:
  Charge `st(0)` depuis mém[Var].
  Empile `st(`*i*`)` dans `st(`*i*`+1)` pour *i*=0..6
- `fstp` @Var:
  Sommet de pile en mém[Var]
  Dépile `st(`*i*`)` dans `st(`*i*`-1)` pour *i*=1..7
- `fst` @Var :
  Sommet de pile en mém[Var].
  Laisse inchangé `st(0)`
- `fmulp` :
  `st(0)`=`st(0)` × `st(1)`,
  `st(`*i*`)`= `st(`*i* + 1`)` pour *i*=1..7
- `fmul st(0),st(`*i*`)`:
  `st(0)`=`st(0)` × `st(`*i*`)`

Calcul de res = X + Y × Z

Modèle pile

```
fld @X
fld @Y
fld @Z
fmulp
faddp
fstp @res
```

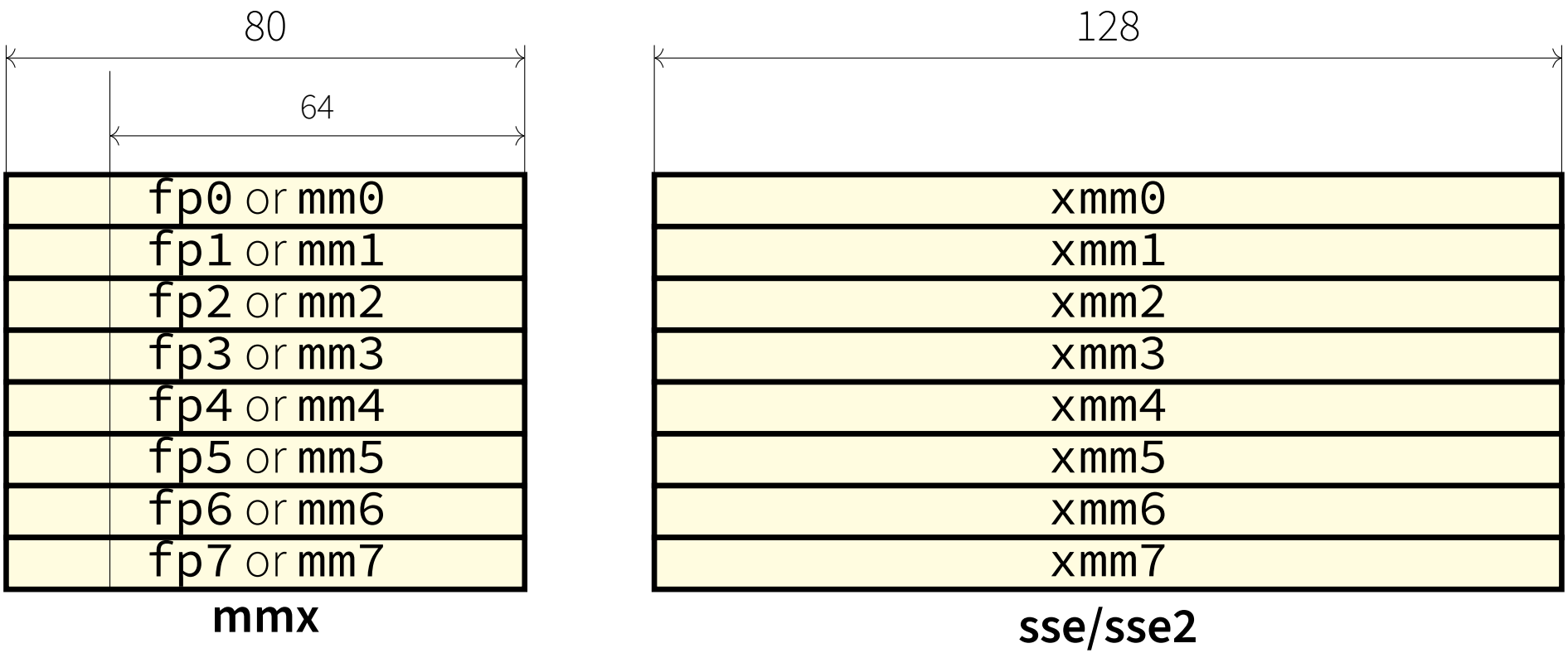
Modèle registre

```
fld @X
fld @Y
fld @Z
fmul st(0), st(1)
fadd st(0), st(2)
fstp @res
```

Instructions SIMD

registres SIMD MMX et SSE

Permettent des traitements parallèles sur plusieurs données.
Intel MMX (entier), SSE-SSE2-SSE3-SSE4, AVX-AVX2-AVX512.
Registres MMX, SSE et SSE2 du pentium



Registres `MMX` partagés avec reg. flottants.
Registres `xmm0`–`xmm7` séparés. 8 registres additionnels `xmm8`–`xmm15` en x86-64.

Instructions SIMD (cont.)

extensions AVX

SSE : opérations 128 bits
2 **doubles**, 4 **floats**

Depuis *Sandy Bridge* (2011) extensions AVX

- 16 registres 256 bits **ymm0** à **ymm15**
- formats
  - 8 **floats**
  - 4 **doubles**
  - 8 **ints**, 16 **shorts**, 32 **chars**
- instructions à 3 opérandes (à la RISC)
- instructions $d = (a \times b) + c$
  (*fused multiply-add* FMA)

512 256	255 128	127 0
zmm0	ymm0	xmm0
zmm1	ymm1	xmm1
zmm2	ymm2	xmm2
zmm3	ymm3	xmm3
zmm4	ymm4	xmm4
zmm5	ymm5	xmm5
zmm6	ymm6	xmm6
zmm7	ymm7	xmm7
zmm8	ymm8	xmm8
zmm9	ymm9	xmm9
zmm10	ymm10	xmm10
zmm11	ymm11	xmm11
zmm12	ymm12	xmm12
zmm13	ymm13	xmm13
zmm14	ymm14	xmm14
zmm15	ymm15	xmm15
zmm16	ymm16	xmm16
zmm17	ymm17	xmm17
zmm18	ymm18	xmm18
zmm19	ymm19	xmm19
zmm20	ymm20	xmm20
zmm21	ymm21	xmm21
zmm22	ymm22	xmm22
zmm23	ymm23	xmm23
zmm24	ymm24	xmm24
zmm25	ymm25	xmm25
zmm26	ymm26	xmm26
zmm27	ymm27	xmm27
zmm28	ymm28	xmm28
zmm29	ymm29	xmm29
zmm30	ymm30	xmm30
zmm31	ymm31	xmm31

Depuis *knights landing* (2016), AVX512 : 32 registres **zmm0–zmm31** sur 512 bits.
Introduit aussi de nombreuses instructions spécialisées (AI, crypto, etc)

Microopérations du pentium

- le RISC permet une grande efficacité à l'exécution, contrairement au format (2,1).
- le maintien d'un jeu d'instructions garde la compatibilité binaire.

$\Rightarrow$ solution Intel :

- conserver le jeu d'instructions IA-32
- et traduire dynamiquement les instructions IA-32 en instructions RISC (μops) [118 bits, 3 sources, 2 destinations].

Instructions x86	μops RISC
`add eax, [ebp + d8]`	`load temp, [ebp + d8]` `add eax, temp`

`add [ebp + d8], eax`	`load temp, [ebp + d8]`
	`add eax, temp`
	`store eax, [ebp+d8]`

`push ecx`	`sub esp, 4` `store [esp], ecx`

Jeu d'instructions ARM

différentes versions arm

Plusieurs versions du processeur ARM avec des jeux d'instructions partiellement différents.

v1, v2 : ARM1, ARM2, ARM3

v3 : ARM7 (game boy *advance*, PDA Palm)

v4 : strongARM

v5 : ARM XScale

v6 : ARM11 (Raspberry Pi 1)

v7 : ARM Cortex A7 (Raspberry Pi-2, tablettes, *smartphones*), ARM Cortex M3 (Arduino), ARM Cortex R4,R5 (temps réel). 'A' Application, 'M' Microcontroleur, 'R' *Real time*

v8 : 64 bits ARM Cortex A53 (Raspberry Pi-3), Cortex A72 (Raspberry Pi-4), Cortex A78 (Apple M1), Cortex M23(32 bits), Cortex R52, Cortex X1

v9 : 64 bits (2022) Cortex X2, Cortex A710, Cortex A510

Jeu d'instructions ARM (cont.)

registres

Initialement instructions sur 32 bits (AArch32)
Version 16bits THUMB (à partir de v4) (permet d'avoir un code plus compact et une fonctionnalité (presque) identique)
Version THUMB2 : permet de mélanger instructions 16bits et 32bits (à partir de v6)
jazelle DBX : support pour l'interprétation directe de *bytecode* java
Neon : extension SIMD (128 bits)

16 registres `r0` à `r15` visibles par le programmeur

- `r15` = `pc` compteur de programme,
- `r14` = `lr` Registre de lien (retour de procédure),
- `r13` = `sp` Pointeur de pile

Registre d'état `APSR` (*application program status register*)

- bits : `N` (<0), `Z` (=0), `C` (retenue), `V` (*oVerflow*)
  Permet de prendre en compte des comparaisons.

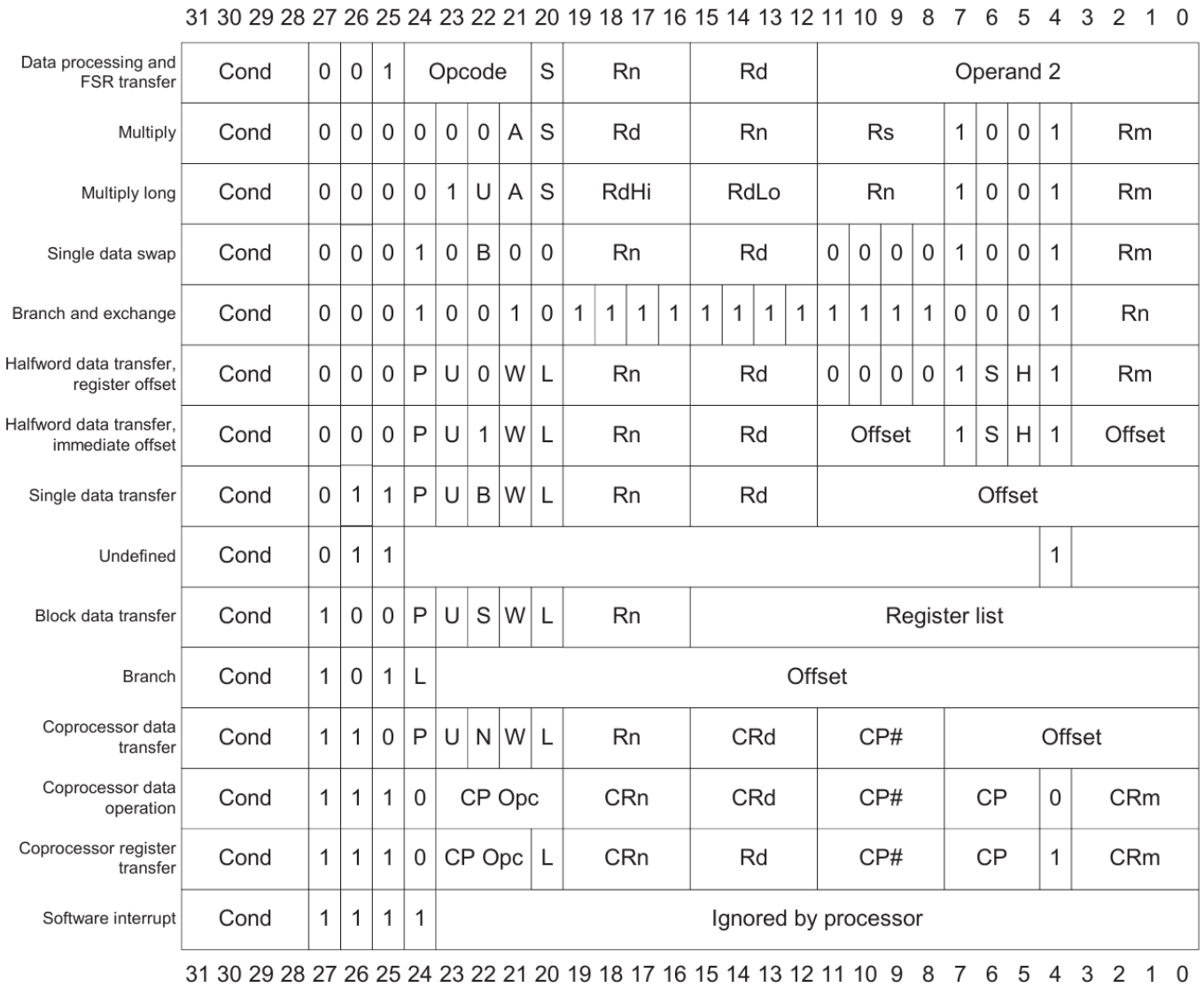
Jeu d'instructions ARM (cont.)

caractéristiques

- toutes les instructions sont conditionnelles : suppression de certains branchements
- modes d'adressage complexes
  - registre + déplacement 8 bits (avec rotation)
  - registre + registre avec décalage
  - pré ou post indexation
- transfert de blocs de registres
- combinaison d'opérations UAL avec décalage/rotation sur le deuxième opérande
- multiplication par constante
- format général des instructions

  si condition **alors** Rd ← Rn *op* shift/rotate (opérande 2)

Instructions ARM



Jeu d'instructions ARM (cont.)

exécution conditionnelle

Le champ **cond** (28–31) décrit la combinaison des bits du APSR **NZCV** qui déterminent la condition.
Le bits 20 (**S**) indique si l'instruction doit mettre à jour le registre APSR.

Exemple : instruction **mov** (copie d'un registre dans un autre)

`mov r1,r2` : copie `r2` dans `r1` (toujours)

`moveq r1,r2` : copie `r2` dans `r1` si = (*EQual*).
Correspond au bit **Z**=1 (la comparaison **cmp** effectue une soustraction)

`movshi r1,r2` : copie `r2` dans `r1` si > (*unsigned HIgher*).
Nécessite le bit **Z**=0 (résultat non nul) et le bit **C** à 1 (la comparaison génère une retenue si le 2ème opérande est >).
En plus met à jour (*Set*) les bits **ZNCV** du APSR selon la valeur de **R2**.

14 conditions existent en fonction des combinaisons de **ZNCV**.

Les conditions occupent 4 bits dans le code.
Utilisation d'une instruction *if-then* (**`it`**) en THUMB (16 bits). Permet de définir une condition et le comportement (vrai/faux) des 4 instructions suivantes.

Jeu d'instructions ARM (cont.)

registre APSR

Code	Suffix	Description	Flags
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
0010	CS / HS	Carry set / unsigned higher or same	C
0011	CC / LO	Carry clear / unsigned lower	!C
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	!N
0110	VS	Overflow	V
0111	VC	No overflow	!V
1000	HI	Unsigned higher	C and !Z
1001	LS	Unsigned lower or same	!C or Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	any
1111	NV	Never	any

Instructions THUMB

	Format	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move shifted register	01	0	0	0	Op		Offset5					Rs			Rd		
Add and subtract	02	0	0	0	1	1	1	Op	Rn/ offset3			Rs			Rd		
Move, compare, add, and subtract immediate	03	0	0	1	Op		Rd			Offset8							
ALU operation	04	0	1	0	0	0	0	Op				Rs			Rd		
High register operations and branch exchange	05	0	1	0	0	0	1	Op		H1	H2	Rs/Hs			RdHd		
PC-relative load	06	0	1	0	0	1	Rd			Word8							
Load and store with relative offset	07	0	1	0	1	L	B	0	Ro			Rb			Rd		
Load and store sign-extended byte and halfword	08	0	1	0	1	H	S	1	Ro			Rb			Rd		
Load and store with immediate offset	09	0	1	1	B	L	Offset5					Rb			Rd		
Load and store halfword	10	1	0	0	0	L	Offset5					Rb			Rd		
SP-relative load and store	11	1	0	0	1	L	Rd			Word8							
Load address	12	1	0	1	0	SP	Rd			Word8							
Add offset to stack pointer	13	1	0	1	1	0	0	0	0	S	SWord7						
Push and pop registers	14	1	0	1	1	L	1	0	R	Rlist							
Multiple load and store	15	1	1	0	0	L	Rb			Rlist							
Conditional branch	16	1	1	0	1	Cond				Softset8							
Software interrupt	17	1	1	0	1	1	1	1	1	Value8							
Unconditional branch	18	1	1	1	0	0	Offset11										
Long branch with link	19	1	1	1	1	H	Offset										

Jeu d'instructions ARM (cont.)

Les instructions conditionnelles permettent de supprimer des branchements.

- réduction du nombre d'instructions
- suppression des aléas de branchement

Exemple calcul de `r3 = max(r1,r2)`

```
# Code MIPS avec
#   brancht conditionnel
  add r3,r1,r0;r3← r1
  slt r4,r2,r1;r4←(r2<r1)
  bgtz r4, fin
    ;si r4>0 aller à fin
  add r3,r2,r0;r3← r2
fin : ...
    ;r3 == max(r1,r2)
```

```
# Code ARM sans
#   brancht conditionnel
  mov r3,r1;r3← r1
  cmp r1,r2
    ;positionne ZNCV
  movlt r3,r2
    ;r3←r2 si (N!=V)
fin : ...
```

Jeu d'instructions ARM (cont.)

AArch64

Dans la v8 du processeur Arm (64 bits), introduction d'un nouveau jeu d'instructions (AArch64) avec des changements importants.

- 32 registres 64 bits
- adresses sur 64 bits
- `pc` n'est plus un registre général
- idem pour pointeur de pile (`sp`)
- les instructions ne sont plus conditionnelles
- possibilité de calcul en flottant *half-precision*
- extensions SIMD SVE (*scalable vector extension*) (jusqu'à 2048 bits)

RISC versus CISC

Loi de Hennessy-Patterson

$$t_{prog} = n_i \times T_c \times \overline{CPI} \quad (1)$$

avec :

- nombre d'instructions exécutées n_i
- période de l'horloge du processeur T_c,
- nombre moyen de cycles nécessaires pour exécuter une instruction $\overline{CPI}$.

Si on a n_i, T et un $\overline{CPI} = 5$ pour un processeur CISC, généralement pour un processeur RISC de même technologie, on aura :

- $n_i' = 1.2 n_i$ (légère augmentation du nombre d'instructions)
- $T' = 0.7T$ (instructions plus simples $\Rightarrow$ T horloge$\searrow$)
- $\overline{CPI}' = 1.4$ (et même $\overline{CPI} < 1$ pour les processeurs modernes superscalaires) soit $\overline{CPI}' = 0.28\overline{CPI}$

$t' = \times 0.7 \times n_i \times 1.4 \times T \times 0.28 \times \overline{CPI} = \times 0.27 \times t$ soit une accélération de **3.6**