

1. Systèmes de numération

Système unaire

- ▶ Chaque nombre $n \in \mathbb{N}$ se représente par $\overbrace{1 \dots 1}^{n \text{ fois}}$
- ▶ L'addition correspond à la concaténation
- ▶ Pas concis

Représentation positionnelle

- ▶ Généralisation du système décimal à une base $b \in \mathbb{N}_{\geq 2}$
- ▶ *Systèmes particuliers*: binaire ($b = 2$), octal ($b = 8$), décimal ($b = 10$), hexadécimal ($b = 16$)
- ▶ *Chiffres*: éléments de $\{0, 1, \dots, b - 1\}$
- ▶ *Chiffres au-delà de 9*: A = 10, B = 11, ..., F = 15, ...
- ▶ *Valeur de x en base b* : $x_b = x_{n-1} \cdot b^{n-1} + \dots + x_1 \cdot b^1 + x_0 \cdot b^0$
- ▶ *Exemple*: $8B5_{16} = 8 \cdot 16^2 + 11 \cdot 16^1 + 5 \cdot 16^0$
- ▶ Les zéros tout à gauche ne changent rien: $(0 \dots 0x)_b = x_b$

Conversions

- ▶ b à 10: $x_0 + b \cdot (x_1 + b \cdot (x_2 + b \cdot (\dots + b \cdot x_{n-1})))$
- ▶ 10 à b : diviser à répétition par b et concaténer les restes de droite à gauche, par ex. $6_2 = 110$:
 $6 \div 2 = 3$ reste 0, $3 \div 2 = 1$ reste 1, $1 \div 2 = 0$ reste 1
- ▶ b à b^m : remplacer chaque bloc de taille m par sa valeur en base b^m , par ex. si $b^m = 2^3$: $10110 \rightarrow 26$
- ▶ b^m à b : éclater chaque symbole vers sa représentation de taille m en base b , par ex. si $b^m = 2^3$: $73 \rightarrow 111011$

Addition

- ▶ Comme en base 10: additionner chiffre à chiffre en base b et propager une retenue vers la gauche

Fractions

- ▶ *Exemple*: $(11,01)_2 = 1 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 3,25$
- ▶ *Chiffres non significatifs*: $(0 \dots 0x,y0 \dots 0)_b = (x,y)_b$

2. Programmation en langage d'assemblage: ARMv8

Registres

- ▶ *Registres*: x_0-x_{30} (64 bits) ou w_0-w_{30} (sous-registres 32 bits)
- ▶ *Usage libre*: x_0-x_7 (arguments) et $x_{19}-x_{28}$ (sauveg. par l'appelé)
- ▶ *Usage semi-libre*: x_9-x_{15} (sauvegardés par l'appelant)

Organisation du code

- ▶ *Ligne*: **étiquette**: opcode operandes // **Commentaire**
- ▶ *Étiquette*: nom symbolique d'une ligne de code
- ▶ *Exemple*: `impair:`

```
mov    x20, 3           // tmp = 3
mul    x20, x20, x19    // tmp = tmp * n
add    x19, x20, 1      // n = tmp + 1
```

Quelques instructions

<code>mov</code>	<code>xd, v</code>	$x_d \leftarrow v$	où v est regis. ou const.
<code>add</code>	<code>xd, xn, v</code>	$x_d \leftarrow x_n + v$	où v est regis. ou const.
<code>mul</code>	<code>xd, xn, xm</code>	$x_d \leftarrow x_n \cdot x_m$	
<code>udiv</code>	<code>xd, xn, xm</code>	$x_d \leftarrow x_n \div x_m$	

Données statiques

- ▶ *Adresse divisible par k* : `.align k`
- ▶ *Alloue k octets consécutifs*: `.skip k`
- ▶ *1, 2, 4, 8 octets*: `.byte v`, `.hword v`, `.word v`, `.xword v`
- ▶ *Chaîne de car.*: `.asciz s`

Segments de données

- ▶ *Instructions*: `.section ".text"`
- ▶ *Données en lecture seule*: `.section ".rodata"`
- ▶ *Données initialisées*: `.section ".data"`
- ▶ *Données non-initialisées*: `.section ".bss"`

Entrée/sortie (de haut niveau via C)

- ▶ *Affichage*: `printf(&format, val1, val2, ...)`
- ▶ *Lecture*: `scanf(&format, &var1, &var2, ...)`
- ▶ *Format nombres*: int32 (%d), uint32 (%u), uint32-hex (%X), 64 bits via préfixe l, par ex. int64 (%ld)

3. Architecture des ordinateurs

Architecture et organisation

- ▶ *Architecture*: spécification des services des composants
- ▶ *Organisation*: description physique des composants

Architecture de von Neumann

- ▶ *Mémoire principale*: stocke les programmes et leurs données
- ▶ *Processeur*: unité centrale de traitement de l'ordinateur
- ▶ *Unités d'entrée/sortie*: contrôlent les périphériques
- ▶ *Bus*: systèmes de communication entre les composants

Mémoire principale

- ▶ Suite de cellules d'octets identifiées par des *adresses* uniques
- ▶ Une adresse peut référer à: 1 octet (8 bits), 2 octets (*demi-mot*), 4 octets (*mot*), 8 octets (*double mot*)
- ▶ Quantité de mémoire utilisable limitée par taille des adresses

- ▶ *Big-endian*: `[00, 58, 40, 0F]` vaut `0058400F`
- ▶ *Little-endian*: `[00, 58, 40, 0F]` vaut `0F405800`
- ▶ *Alignement*: adresser 2^k octets à une adresse qui n'est pas un multiple de 2^k — parfois: *interdit*, souvent: *ralentit l'accès*

Processeur

- ▶ *Jeu d'instructions* élémentaires, par ex: $\overbrace{\text{add } x_{10}, x_{11}, x_{12}}^{\text{code d'opér.}} \overbrace{\quad \quad \quad}^{\text{opérandes}}$
- ▶ *Registres*: cellules de mémoire interne, très rapide d'accès
- ▶ *Code machine*: traduction des instructions en suite de bits
- ▶ *Compteur d'instruction*: pointe vers prochaine instruction
- ▶ *Unité de contrôle*: coordonne l'exécution des instructions
- ▶ *Unité arithmétique et logique*: calculs sur \mathbb{Z} et chaînes de bits
- ▶ *Pipeline*: parallélisation des étapes d'exécution
- ▶ *RISC*: instructions simples, taille fixe, mémoire-ou-autre

4. Nombres entiers

Représentation des entiers signés

► Compl. à 2: $\text{val}(x_{n-1} \dots x_1 x_0) = -x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i$

bits	000	001	010	011	100	101	110	111
valeur	0	1	2	3	-4	-3	-2	-1

- Représentables sur n bits: $[-2^{n-1}, 2^{n-1} - 1]$
- Bit de signe: négatif ssi bit de gauche = 1
- Ajout de bits: répéter bit de signe à gauche: $101 \rightarrow 1 \dots 101$
- Changement de signe: $010 \xrightarrow{\text{complément}} 101 \xrightarrow{+1} 110$

Opérations arithmétiques

- Addition: comme les entiers non signés
- Soustraction: addition/changement de signe: $a - b = a + (-b)$
- Report: lors d'une retenue sur la somme des bits de poids fort
- Débordement: lorsque le résultat ne peut pas être représenté

► Multiplication et division non signées: comme en base 10:

$$\begin{array}{r} \times \quad 101 \quad (5) \\ \quad \quad 11 \quad (3) \\ \hline \quad \quad 101 \\ \quad 101 \\ \hline + \quad 1111 \quad (15) \end{array} \qquad \begin{array}{r} 10011 \quad | \quad 11 \\ - \quad 11 \quad | \quad 00110 \\ \hline \quad \quad 111 \\ - \quad 11 \\ \hline \quad \quad \quad 1 \end{array}$$

- Mult. signée: étendre opérandes à $2n$ bits et garder $2n$ bits faibles du résultat (s'implémente sans extension explicite)
- Division signée: calculer $|a| \div |b|$ et ajuster signe

Codes de condition

- Codes: N (négatif), Z (zéro), C (report), V (débordement)
- Codes modifiés par: **cmp**, **adds**, **subs**, **negs**, **adcs**, **sbcs**
- Comparaison: codes mis à jour via soustraction bidon
- Accès aux codes: avec **b.condition** etiq
- Accès au report: «**adc** rd, rn, rm» $\equiv r_d \leftarrow r_n + r_m + C$

5. Accès aux données

Adresses

- Numérique: entier non négatif, souvent en hexadécimal
- Symbolique: chaîne représentant une adresse à déterminer

Modes d'adressage

- Mode: méthode pour récupérer la valeur d'un opérande
- Récapitulatif des modes:

Nom	Valeur récupérée	Exemple
immédiat	$i \mapsto i$	mov x0, 42
direct	$a \mapsto \text{mem}[a]$	—
par registre	$n \mapsto \text{reg}[n]$	mov x0, x1
indirect	$a \mapsto \text{mem}[\text{mem}[a]]$	—
indirect par registre	$n \mapsto \text{mem}[\text{reg}[n]]$	ldr x0, [x1]
indir. par reg. indexé	$n, i \mapsto \text{mem}[\text{reg}[n] + i]$	ldr x0, [x1, i]
indir. par reg. indexé pré-incrémenté	$\text{reg}[n] \leftarrow \text{reg}[n] + i$, suivi de $n, i \mapsto \text{mem}[\text{reg}[n]]$	ldr x0, [x1, i]!
indir. par reg. indexé post-incrémenté	$n, i \mapsto \text{mem}[\text{reg}[n]]$, suivi de $\text{reg}[n] \leftarrow \text{reg}[n] + i$	ldr x0, [x1], i
relatif	$i \mapsto \text{mem}[\text{reg}[pc] + i]$	ldr x0, var

Accès mémoire sur ARMv8

► Chargement et stockage:

# octets	chargement	stockage
1	ldrb wd, a	strb wd, a
2	ldrh wd, a	strh wd, a
4	ldr wd, a	str wd, a
8	ldr xd, a	str xd, a

► Autres instructions:

```
adr r, etiq // charge adr(etiq) dans reg. r
mov r, s // charge reg. s dans reg. r
mov r, i // charge valeur i dans reg. r
```

Assemblage

- Assembleur: instructions \rightarrow code machine; la plupart des adresses symboliques \rightarrow adresses numériques
- Éditeur de liens: fichiers objets \rightarrow fichier exécutable; recalcule certaines adresses; adresses symboliques \rightarrow numériques

6. Tableaux

Généralités

- Tableau: collection d'éléments identifiés par des indices
- Éléments: tous de même taille, contigus en mémoire
- Indice: d -uplet i où $d \geq 1$ est la dimension
- Bornes: $0 \leq i_j < n_j$ pour chaque dimension j
- Taille: $n_0 \cdot n_1 \dots n_{d-1}$ éléments
- Types: le type des éléments est implicite
- Exemples de tableau 1D et tableau 2D:

0	01010101	(0,0)	2
1	11110000	(0,1)	33
2	01101101	(1,0)	65535
3	11111111	(1,1)	73
4	11110101	(2,0)	9000
		(2,1)	255

$n_0 = 5$
5 éléments

$n_0 = 3, n_1 = 2$
6 éléments

Calcul d'adresse

- Index: adresse relative à laquelle est stocké un élément
- Calcul: si a = adresse du tableau et k = nombre d'octets d'un élément, alors l'adresse d'un élément correspond à:

$$\begin{array}{l} a + \underbrace{i \cdot k}_{\text{index élém. } i \text{ (tableau 1D)}} \\ a + \underbrace{(i \cdot n_1 + j) \cdot k}_{\text{index élém. } (i, j) \text{ (tableau 2D)}} \end{array}$$

Allocation/accès mémoire

► Tableau non initialisé:

```
.section ".bss"
.align 2
tab: .skip 3*2*2 // n0 * n1 * # octets
```

► Tableau initialisé:

```
.section ".data"
tab: .hword 2, 33, 65535, 73, 9000, 255 // six demi-mots
```

► Accès: avec **str**/**ldr** (ou variantes) + modes d'adressage

7. Programmation structurée

Séquence

- Composition séquentielle d'instructions
- Une instruction de haut niveau peut nécessiter plusieurs instructions de bas niveau; par ex. « `x19 *= 7` » devient:

```
mov    x20, 7
mul    x19, x19, x20
```

Sélection

- Exécution conditionnelle d'instructions (`if`, `switch`, ...)
- *Implémentation*: branchements avant:

```
if (cond(xd, xn)) {
    // code si
}
else {
    // code sinon
}

si:
    cmp    xd, xn
    b.-cond  sinon
    // code si
    b      fin
sinon:
    // code sinon
fin:
```

- *Conditions multiples*: obtenues avec plusieurs sélections

Itération

- Exécution répétée d'instructions (`while`, `do while`, `for`, ...)
- *Implémentation*: branchements arrière, et parfois avant:

```
while (cond(xd, xn)) {
    // code
}

boucle:
    cmp    xd, xn
    b.-cond  fin
    // code
    b      boucle
fin:
```

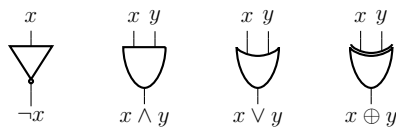
Sous-programmes

- Permettent de modulariser le code en sous-routines
- Registres partagés par programme et sous-programmes
- *Arguments*: passés par valeur ou adresse dans x_0-x_7 (en ordre)
- *Appel*: « `bl sprog` » assigne $x_{30} \leftarrow pc+4$ et branche à `sprog`:
- *Retour*: « `ret` » branche vers l'adresse de retour x_{30}
- *Sauvegarde*: l'appelé doit rétablir les registres x_{19} à x_{30}

8. Circuits logiques

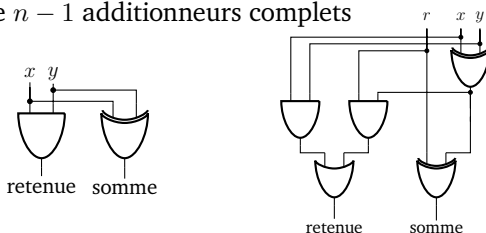
Circuits

- « Blocs » de base constitués de portes logiques qui permettent d'implémenter l'ordinateur:



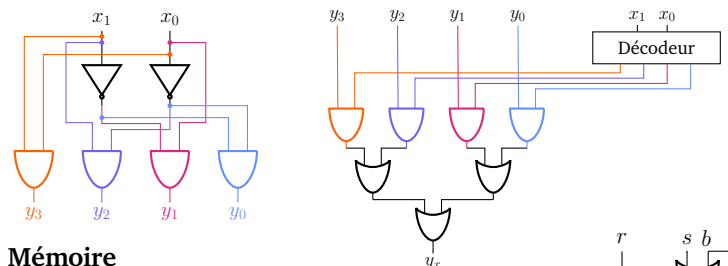
Arithmétique

- *Demi-additionneur*: somme de deux bits
- *Additionneur complet*: somme de deux bits et d'une retenue
- *Addition*: somme sur n bits avec un demi-additionneur et une cascade de $n - 1$ additionneurs complets



Décodeur

- *Décodeur*: sur entrée x , sortie: $y_x = 1$ et $y_j = 0$ pour $j \neq x$
- *Multiplieur*: sur entrée x , sélectionne le bit y_x
- *Instructions*: décodables/exécutables à l'aide de tels circuits



Mémoire

- *Circuits séquentiels*: peuvent mémoriser des bits
- *Verrou*: stocke un bit b , remise à 0 avec r , et mise à 1 avec s

