

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

Architecture des ordinateurs

Wilfried Segretier
wsegreti@univ-ag.fr

Université des Antilles

17 novembre 2016



Organisation du cours

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

40h

- ▶ 20h cours
- ▶ 18h TD
- ▶ 12h TP
- ▶ 1 note CC (1h)
- ▶ 1 note CT
- ▶ 1 note TP

Sommaire

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

1 Introduction

2 Histoire de l'ordinateur et principes généraux

3 Représentation des informations

4 Circuits logiques

5 Micro-architecture

Introduction

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

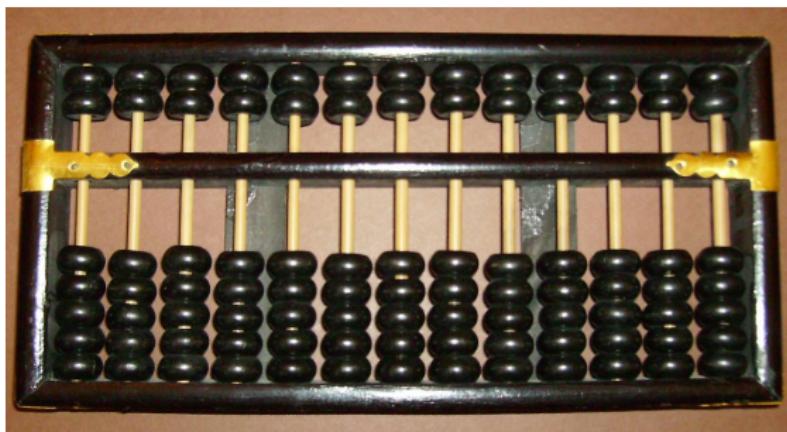
Micro-
architecture

- ▶ **Informatique** : *Science du traitement de l'**information** de manière **automatique***
- ▶ **Architecture des ordinateurs** : *Etude du fonctionnement des ordinateurs tant au niveau matériel que logiciel*
- ▶ Monde en perpetuelle mutation, évolution rapide des technologies mais pas des concepts fondamentaux !

Automatisation du calcul

- ▶ Avant XVIIe siècle : doigts, cailloux, bouliers, abaque
- ▶ XVIIe - XIXe : Machines à calculer mécaniques (certaines programmables)
- ▶ 1936-1956 : 1ere génération - tubes à vide (2nde Guerre Mondiale)
- ▶ 1956-1963 : 2eme génération - transistors
- ▶ 1963-1971 : 3ème génération - circuits intégrés
- ▶ 1971-présent : 4ème génération - microprocesseur
- ▶ Futur : ordinateurs optiques, quantiques ?

Instruments/outils rudimentaires
→ automatiser le calcul



Pascaline, Blaise Pascal, 1642

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreri@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

Machine mécanique

Additionne et soustrait des nombres à 6 chiffres
(devises : livres, deniers)
rouages internes inspirés d'horloges



Machine à différence, Babbage, 1822

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

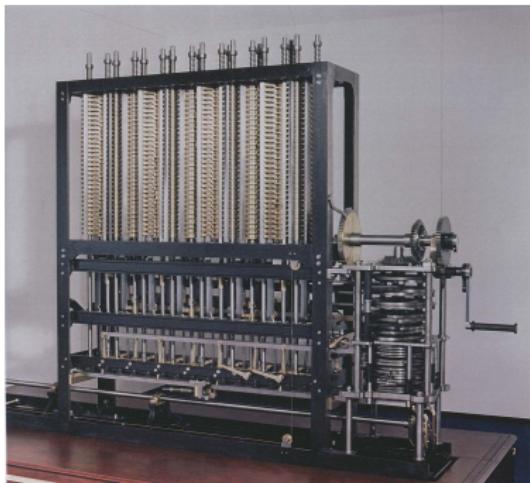
Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

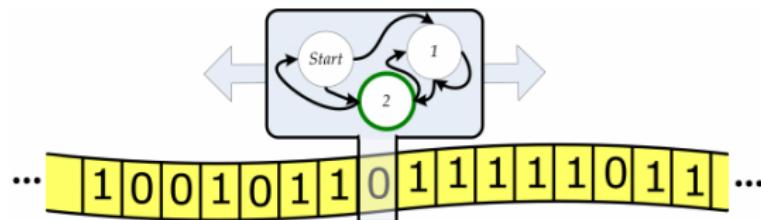


- ▶ Machine mécanique
- ▶ calcul de tables de fonctions polynomiales
- ▶ Approximation trigonométrie, logarithmique, ...
- ▶ Utilisée par les navigateurs, scientifiques, etc

Machine de Turing, 1936

Modèle théorique d'ordinateur

Thèse Church-Turing : Tout problème calculable (algorithme) correspond à une machine de Turing



- ▶ ruban infini (données)
- ▶ Tête de lecture/écriture
- ▶ Table de transition : symbole lu, état courant → symbole, direction, état

ABC (Atanasoff-Berry-Computer), 1937-1942

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

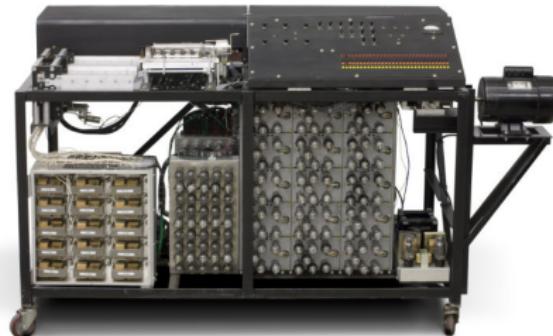
Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

John Vincent Atanasoff, Clifford Berry



Premier ordinateur numérique électronique :
Non programmable
Mise en oeuvre du binaire pour la premiere fois
Electronique a la place de la mecanique
Séparation entre mémoire et unité de calcul

Harvard Mark I, 1944

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

Howard Aiken, machine électromécanique :
Multiplication de nombres de 23 chiffres en 6 secondes
Addition en 3 dixièmes de seconde
Pas programmable !



Colossus Mark II, 1943-1945

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

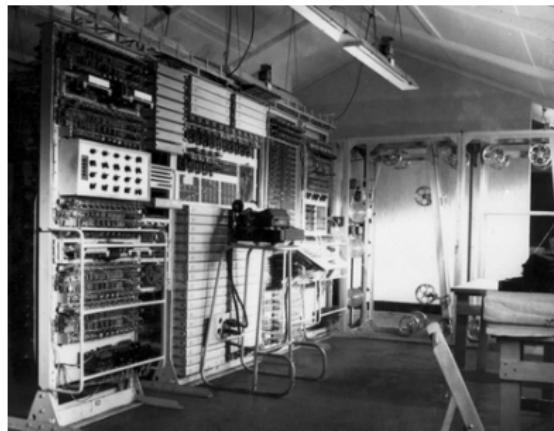
Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

- ▶ 2nde guerre mondiale : cryptanalyse
- ▶ 2 400 tubes à vide
- ▶ 5 000 opérations par seconde



ENIAC, 1945

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

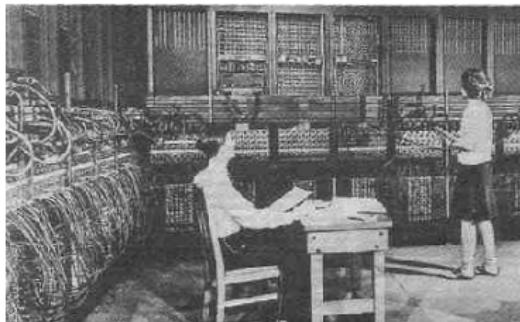
Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

Premier ordinateur électronique programmable

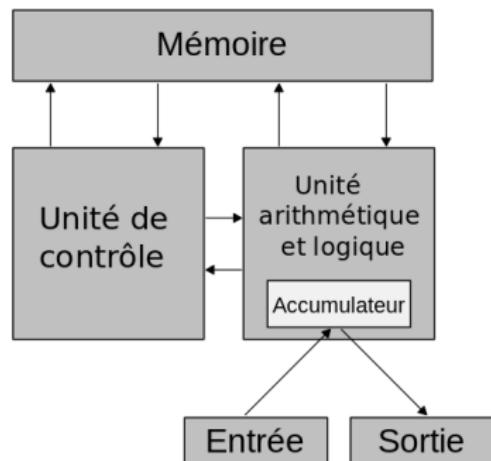


- ▶ Electronic Numerical Integrator And Calculator
- ▶ John Eckert, John Mauchly
- ▶ 18 000 tubes, 30 tonnes
- ▶ Multiplication de nombres de 10 chiffres en 3ms

Architecture de Von Neumann

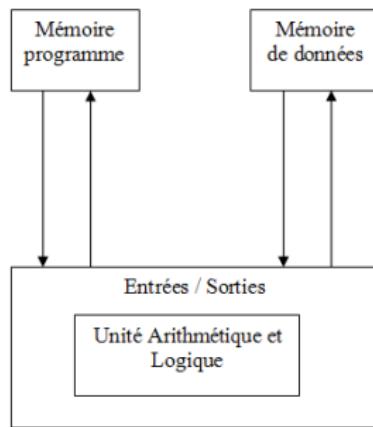
Modèle d'ordinateur Indépendant de la technologie

- ▶ Mémoire : programme + données
- ▶ Unité de contrôle : ordonnancement des opérations
- ▶ Unité Arithmétique et logique : calculs
- ▶ Dispositif d'entrée/sortie : intégrer avec le monde



Architecture de Harvard

- ▶ Programme et données séparés
- ▶ 2 bus distincts
- ▶ CPU : Accès simultané aux données et aux instructions
- ▶ Peut être plus performant que Von Neumann mais plus complexe
- ▶ microprocesseurs et microcontrôleurs spécialisés



2nde Génération 1956-1963

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

Découverte du transistor en 1948



- ▶ Premiers langages (Cobol, Fortran, ...)
- ▶ Systèmes d'exploitations
- ▶ Mémoire non volatile
- ▶ Bande magnétique

3eme Génération 1963-1971

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

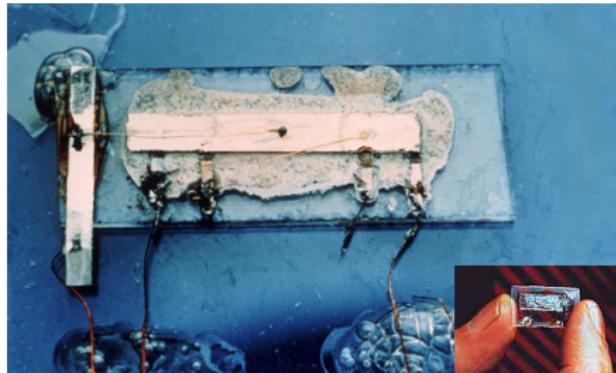
Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

Découverte du circuit intégré en 1958
Jack Kilby, Texas Instrument



- ▶ réduction consommation
- ▶ réduction encombrement
- ▶ amélioration fiabilité, performances

4eme Génération 1971-Présent

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

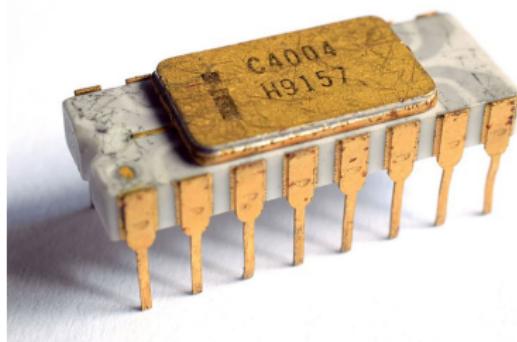
Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

Premier microprocesseur : INTEL 4004 : 1971



- ▶ Miniaturisation
- ▶ Micro-Informatique
- ▶ PC (personal computer)

Lois de Moore

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

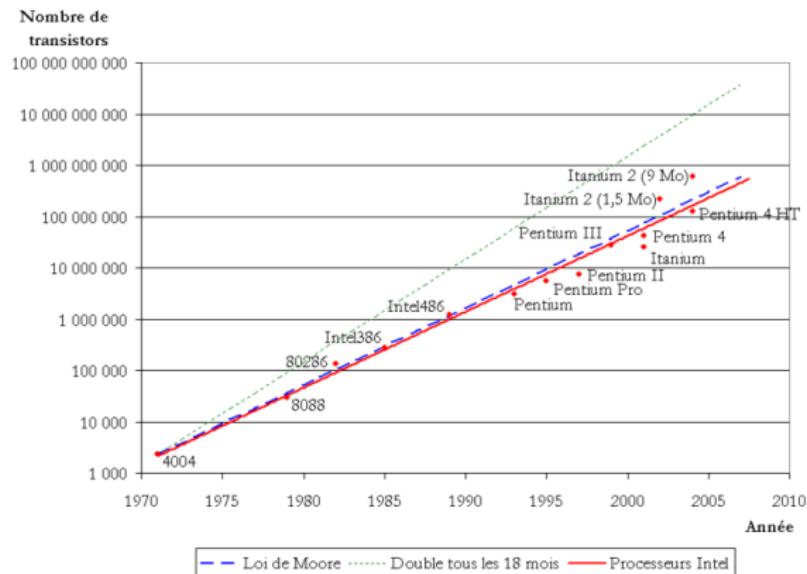
Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture



- ▶ Le nombre de transistors par circuit de même taille double, à prix constants, tous les 18 mois
- ▶ Croissance exponentielle de la puissance des ordinateurs
- ▶ Loi empirique - Limites physiques : taille des atomes

Représentation des informations

Informations sous différentes formes

- ▶ Nombre, textes, images, sons, ...
- ▶ Représentation binaire
- ▶ Ensembles de bits (binary digit {1,0})
- ▶ Ex : $1010000 \rightarrow 80 \rightarrow "P"$
- ▶ Avantage : simplicité, réalisation technique
- ▶ 1 : courant, 0 : absence de courant
- ▶ Bases 8 ($[0, 8]$) et 16 ($[0, F]$) (Octal, hexadecimal) également très utilisées (3, 4 bits)



Entiers positifs ou nuls

Formule générale : base b

$$(a_n a_{n-1} \dots a_1 a_0)_b = \sum_{i=0}^n a_i b^i$$

Binaire :

$$\begin{aligned}(110101)_2 &= 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ (110101)_2 &= 32 + 16 + 4 + 1 = 53\end{aligned}$$

N bits : de 0 à $2^N - 1$

8 bits : 0 à 255

Octal : groupes de 3 bits → $(101001)_2 \rightarrow (51)_8$

Hexadecimal : groupes de 4 bits → $(10100101)_2 \rightarrow (A5)_{16}$

Entiers relatifs : signe et valeur absolue

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

- ▶ 5 : 0|0101
- ▶ -5 : 1|0101
- ▶ Problèmes :
 - 0 représenté 2 fois
 - calculs incohérents

$$\begin{array}{r} \blacktriangleright 0 : 0|0000 \\ \blacktriangleright -0 : 1|0000 \\ \blacktriangleright 5 : 0|0101 \\ + \\ -2 : 1|0010 \\ = \\ -7 : 1|0111 \\ \blacktriangleright \text{devrait etre égal à } 3 : \\ 0|0011! \end{array}$$

Entiers relatifs : Complément à 2

- ▶ inversion bits val abs
- ▶ ajout de 1
- ▶ 5 : 0|0101
- ▶ -5 : 1|1011
- ▶ 1 fois 0, calculs cohérents
- ▶ bit de signe : N-1 bits

- ▶ Exemple : -45
- ▶ 45 : 0101101
- ▶ 1010010
- +
- 0000001
- =
- 1010011
- ▶ 50 : 0110010
- +
- 45 : 1010011
- =
- 5 : 0000101

Nombres fractionnaires : Virgule fixe

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

- ▶ Virgule gérée par le programmeur
- ▶ $a_n a_{n-1} \dots a_1 a_0, a_{-1} a_{-2} \dots a_{-p}$
- ▶ $\sum_{i=0}^n a_i b^i + \sum_{i=1}^p a_{-i} b^{-i}$
- ▶ $1101, 01101 = 13,40625$
- ▶ $8+4+1, 0.25+0.125+0.03125$

Nombres fractionnaires : Virgule flottante



- ▶ $N = M \times B^E$
- ▶ bit de signe s
- ▶ $(-1)^s \times 2^{bias-E} \times 1.M$
- ▶ IEEE 754 32 bits (simple précision) : s (1bit), E (8bits), M (23 bits)
- ▶ $(-1)^s \times 2^{E-127} \times 1.M$
- ▶ $0\ 10000000\ 1000000000000000000000000 = 3$
- ▶ $0\ 10000110\ 0010111101000000000000000 = 151,625$
- ▶ double precision : 64 bits : s (1bit), E (11bits), M (52 bits)
- ▶ double précision étendue : s (1bit), E (15bits), M (64 bits)

Nombres fractionnaires : Virgule flottante

Valeur de s, e, f	Valeur numérique
$0 < e < 255$	$(-1)^s \times 2^{e-127} \times 1.f$ (normal Nb)
$e = 0 ; f \neq 0$ (au moins un bit de f non nul)	$(-1)^s \times 2^{-126} \times 0.f$ (subnormal Nb)
$e = 0 ; f = 0$ (tous les bits de f sont à 0)	$(-1)^s \times 0.0$ (0 signé)
$s = 0; e = 255; f = 0$	$+\infty$
$s = 1; e = 255; f = 0$	$-\infty$
$s = u; e = 255; f \neq 0$	NaN (Not-a-Number)

Nombres fractionnaires : Virgule flottante

Représentation 32 bits	Représent. Hexa	Valeur
0 10000000 10000000000000000000000000000000	40400000	$(1,1 \times 2^{128-127})_{(2)} = (1,1 \times 2^1)_{(2)} = 11_{(2)}$ $3_{(10)}$
0 01111101 01000000000000000000000000000000	3EA00000	$(1,01 \times 2^{125-127})_{(2)} = (1,01 \times 2^{-2})_{(2)}$ $0,0101_{(2)} = 0,3125_{(10)}$
0 10000110 001011110100000000000000	4317A000	$= (1,0010111101 \times 2^{134-127})_{(2)}$ $= (1,0010111101 \times 2^7)_{(2)}$ $= 10010111,101_{(2)}$ $= 151,625_{(10)}$
1 01111110 00000000000000000000000000000000	BF000000	$-(1.0 \times 2^{126-127})_{(2)} = -(1.0 \times 2^{-1})_{(2)}$ $-(0.1)_{(2)} = -0.5_{(10)}$

Code de Gray (binaire réfléchi)

Frank Gray, 1953, (Code Baudot, 1874 : caractères)

- ▶ Deux digits $\{0,1\} \neq$ binaire classique
- ▶ Incrémentation : inversion d'un bit
- ▶ Pas d'état transitoire (circuit logique)
- ▶ Utilisation : capteur positionnels, tables de Karnaugh

Binaire classique

0 : 000
1 : 001
2 : 010
3 : 011
4 : 100
5 : 101

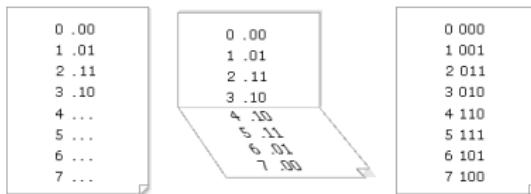
Code de Gray

0 : 000
1 : 001
2 : 011
3 : 010
4 : 110
5 : 111

Code de Gray : Construction

Trois méthodes de construction :

- 1 inversion du bit le plus à droite et qui crée un nouveau nombre
- 2 binaire réflechi : lorsque nouveau bit est nécessaire : duplication en miroir des nombres présents puis ajout de 0 pour les anciens et 1 pour les nouveaux



- 3 XOR entre binaire initial et son image décalée d'un bit vers la droite

Tables de correspondance

- ▶ *BCD Binary Coded Decimal*
- ▶ *ASCII American Standard Code for Information Interchange*
- ▶ *EBCDIC Extended Binary Coded Decimal Internal Code*
- ▶ *UNICODE - ISO/IEC 10646*

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1	!	33	21	41	!	65	41	101	A	97	61	141	a
2	2	2	"	34	22	42	"	66	42	102	B	98	62	142	b
3	3	3	#	35	23	43	#	67	43	103	C	99	63	143	c
4	4	4	\$	36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5	%	37	25	45	%	69	45	105	E	101	65	145	e
6	6	6	&	38	26	46	&	70	46	106	F	102	66	146	f
7	7	7	'	39	27	47	'	71	47	107	G	103	67	147	g
8	8	10	(40	28	50	(72	48	110	H	104	68	150	h
9	9	11)	41	29	51)	73	49	111	I	105	69	151	i
10	A	12	*	42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13	+	43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14	,	44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15	-	45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16	:	46	2E	56	:	78	4E	116	N	110	6E	156	n
15	F	17	/	47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20	0	48	30	60	0	80	50	120	P	112	70	160	p
17	11	21	1	49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22	2	50	32	62	2	82	52	122	R	114	72	162	r
19	13	23	3	51	33	63	3	83	53	123	S	115	73	163	s
20	14	24	4	52	34	64	4	84	54	124	T	116	74	164	t
21	15	25	5	53	35	65	5	85	55	125	U	117	75	165	u
22	16	26	6	54	36	66	6	86	56	126	V	118	76	166	v
23	17	27	7	55	37	67	7	87	57	127	W	119	77	167	w
24	18	30	8	56	38	70	8	88	58	130	X	120	78	170	x
25	19	31	9	57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32	:	58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33	;	59	3B	73	;	91	5B	133	{	123	7B	173	{
28	1C	34	<	60	3C	74	<	92	5C	134	\	124	7C	174	\
29	1D	35	=	61	3D	75	=	93	5D	135	^	125	7D	175	=
30	1E	36	>	62	3E	76	>	94	5E	136	_	126	7E	176	_
31	1F	37	?	63	3F	77	?	95	5F	137	-	127	7F	177	-

Exercices d'application

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

Donner sur 8 bits, les représentations signe|valeur absolue et complément à deux des entiers relatifs suivants :
-47, -114, -32, -128

Exprimer en binaire, en octal et en hexadécimal les nombres décimaux suivants (virgule fixe) :
27.625, 112.75, 96.296875

Convertir en décimal les nombres suivants :
DADA.C (hexadécimal), 270.14 (octal), 11011.01111 (binaire)

Donner la représentation en virgule flottante au format IEEE 754 des nombres décimaux suivants :
1 10000010 01000000000000000000000000000000
0 01111110 00000000000000000000000000000000

Circuits Logiques

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

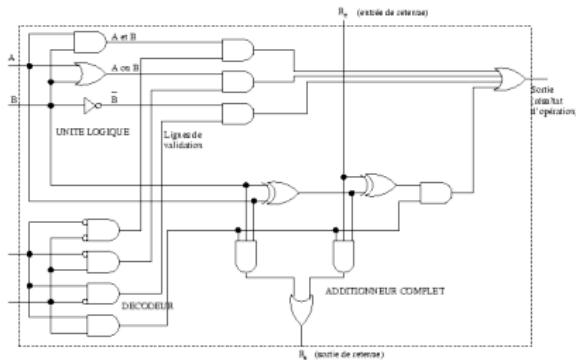
Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture



- ▶ Circuit électrique pour lequel 2 valeurs (signaux) sont possibles 1 (tension élevée) et 0 (tension faible)
- ▶ Portes logiques : permettent de combiner les signaux
- ▶ Entrée : n signaux (2), Sortie 1 signal
- ▶ Fonction logiques classiques (NON, ET, OU, XOR,...)
- ▶ Toutes fonctions de n variables possibles

Tables de vérité

Architecture des ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

Fonction logique : fonction d'une ou plusieurs variables **booléennes** (qui peut prendre la valeur 0 ou 1) dans $\{0, 1\}$.

Exemples :

$$\textcircled{1} \quad f_0(a) = \begin{cases} 0 & \text{si } a = 1 \\ 1 & \text{si } a = 0 \end{cases}$$

$$\textcircled{2} \quad f_1(a, b) = a$$

$$\textcircled{3} \quad f_2(a, b, c) = \begin{cases} c & \text{si } a = 0 \\ 1 - c & \text{si } a = 1 \end{cases}$$

Tables de vérité

fonction f_0

a	S
0	1
1	0

fonction f_2

a	b	c	S
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

ET, OU

Architecture des ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

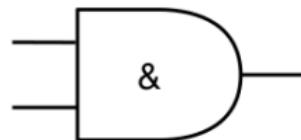
Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

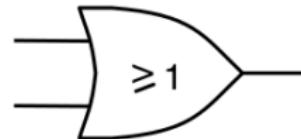
porte ET



a	b	S
0	0	0
0	1	0
1	0	0
1	1	1

$$S = f(a, b) = a \cdot b$$

porte OU

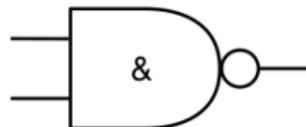


a	b	S
0	0	0
0	1	1
1	0	1
1	1	1

$$f(a, b) = a + b$$

NON-ET, NON-OU

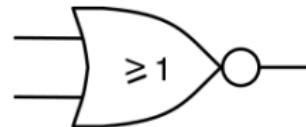
porte NON-ET



a	b	S
0	0	1
0	1	1
1	0	1
1	1	0

$$f(a, b) = \overline{a \cdot b}$$

porte NON-OU



a	b	S
0	0	1
0	1	0
1	0	0
1	1	0

$$f(a, b) = \overline{a + b}$$

Fonctions booléennes de 2 variables

2 variables : 16 fonctions

00	01	10	11
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1

$$f(a, b)$$

$$0$$

$$ab$$

$$a\bar{b}$$

$$a$$

$$\bar{a}b$$

$$b$$

$$a \oplus b$$

$$a + b$$

00	01	10	11
1	0	0	0
1	0	0	1
1	0	1	0
1	0	1	1
1	1	0	0
1	1	0	1
1	1	1	0
1	1	1	1

$$f(a, b)$$

$$a + b$$

$$a \oplus b$$

$$\bar{b}$$

$$a + \bar{b}$$

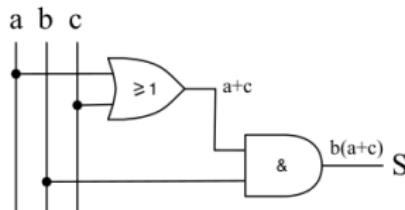
$$\bar{a}$$

$$\bar{a} + b$$

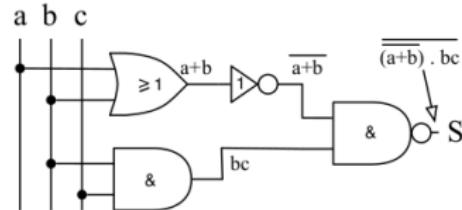
$$\bar{a}b$$

$$1$$

Circuit → Table de vérité



a	b	c	$a + c$	$b(a + c)$
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1



a	b	c	$a + b$	$\overline{a + b}$	bc	S
0	0	0	0	1	0	1
0	0	1	0	1	0	1
0	1	0	1	0	0	1
0	1	1	1	0	1	1
1	0	0	1	0	0	1
1	0	1	1	0	0	1
1	1	0	1	0	0	1
1	1	1	1	0	1	1

Règles

Constantes	$a + 0 = a$	$a \cdot 0 = 0$
	$a + 1 = 1$	$a \cdot 1 = a$
Idempotence	$a + a = a$	$a \cdot a = a$
Complémentation	$a + \bar{a} = 1$	$a \cdot \bar{a} = 0$
Commutativité	$a + b = b + a$	$a \cdot b = b \cdot a$
Distributivité	$a + (bc) = (a + b)(a + c)$ $a(b + c) = (ab) + (ac)$	
Associativité	$a + (b + c) = (a + b) + c = a + b + c$ $a(bc) = (ab)c = abc$	
Lois de De Morgan	$\overline{ab} = \bar{a} + \bar{b}$	$\overline{a + b} = \bar{a} \bar{b}$
Autres relations	$\overline{\bar{a}} = a$	$(a + b)(a + \bar{b}) = a$

Complétude de la porte NON-ET

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

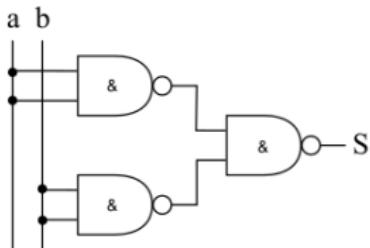
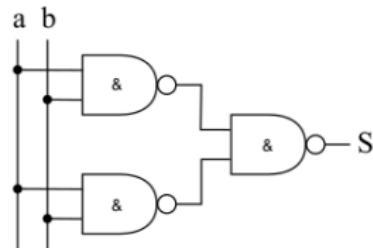
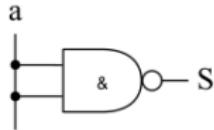
Circuits
logiques

Micro-
architecture

$$\bar{a} = \overline{a \cdot a}$$

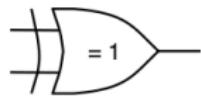
$$\begin{aligned} a \cdot b &= \overline{\overline{a} \cdot \overline{b}} \\ &= \overline{\overline{a}b \cdot a\overline{b}} \end{aligned}$$

$$\begin{aligned} a + b &= \overline{\overline{a} + \overline{b}} \\ &= \overline{\overline{a} \cdot \overline{b}} \\ &= \overline{\overline{a}a \cdot \overline{b}b} \end{aligned}$$



Porte XOR

porte XOR



a	b	S
0	0	0
0	1	1
1	0	1
1	1	0

$$\begin{aligned}
 f(a, b) &= a \oplus b \\
 &= (a + b)(\bar{a}b) \\
 &= (a + b)(\bar{a} + \bar{b}) \\
 &= a\bar{a} + a\bar{b} + b\bar{a} + b\bar{b} \\
 &= a\bar{b} + b\bar{a} \\
 &= \overline{\overline{a}\bar{b}} + \overline{b\bar{a}} = \overline{\overline{a}\cdot\overline{b\bar{a}}}
 \end{aligned}$$

Réalisation à partir des portes précédentes.

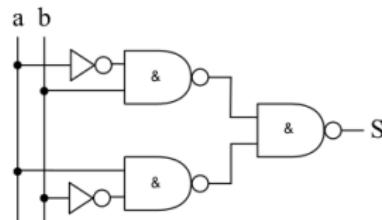
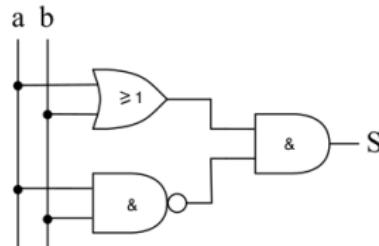
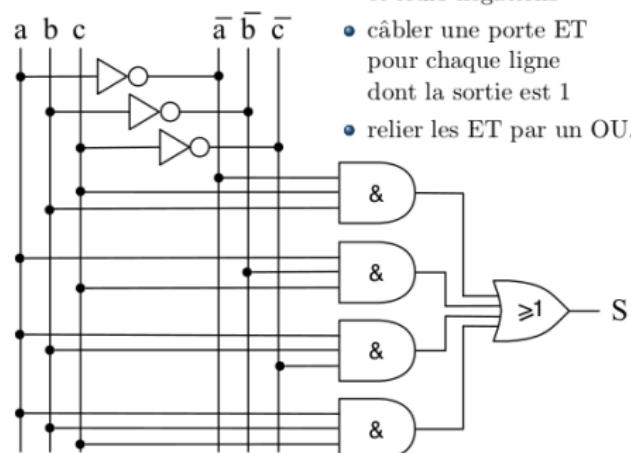


Table de vérité → Circuit

La fonction majoritaire : la valeur de sortie est celle qui apparaît majoritairement en entrée.

a	b	c	S
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



$$f(a, b, c) = \bar{a}bc + a\bar{b}c + ab\bar{c} + abc$$

Simplification de fonctions logiques

$$\begin{aligned}
 f(a, b, c) &= \overline{\overline{a + \bar{b} \cdot bc}} \\
 &= \overline{\overline{a + \bar{b}}} + \overline{bc} \\
 &= a + \cancel{b} + \bar{b} + \bar{c} \\
 &= a + 1 + \bar{c} = 1
 \end{aligned}$$

fonction majoritaire :

$$\begin{aligned}
 f(a, b, c) &= \bar{a}bc + a\bar{b}c + ab\bar{c} + abc &&= (\bar{a}b + a\bar{b})c + ab(\bar{c} + c) \\
 &= \bar{a}bc + a\bar{b}c + ab\bar{c} + abc + \cancel{abc} + \cancel{abc} &&= (a + b)(\bar{a} + \bar{b})c + ab \\
 &= (a + \bar{a})bc + (b + \bar{b})ac + (c + \bar{c})bc &&= (ac + bc)\overline{ab} + ab \\
 &= ab + ac + bc &&= (ab + ac + bc)(\overline{ab} + ab)
 \end{aligned}$$

Simplification : Méthode de Karnaugh

- ▶ Maurice Karnaugh, 1953
- ▶ Méthode graphique
- ▶ Utile pour les fonctions complexes
- ▶ Table de vérité classique → forme compacte, binaire réfléchi
- ▶ Encadrer les 1 dans des rectangles :
 - de taille 2^n
 - avec n le plus grand possible
 - prendre en compte les bords
- ▶ Formule :
 - Rectangles : équivalent à un ET entre les variables qui ont toujours 1 pour valeur dans le rectangle (négation de celles qui valent toujours 0)
 - OU entre les formules de chaque rectangle

Simplification : Méthode de Karnaugh

La fonction majoritaire

binaire

classique

a	b	c	S
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

binaire

réfléchi

a	b	c	S
0	0	0	0
0	0	1	0
0	1	1	1
0	1	0	0
1	1	0	1
1	1	1	1
1	0	1	1
1	0	0	0

$$f(a, b, c) = ab + ac + bc$$

bc	00	01	11	10
a	0	0	1	0
0	0	0	1	0
1	0	1	1	1

bc	00	01	11	10
a	0	0	1	0
0	0	0	1	0
1	0	1	1	1

Simplification : Méthode de Karnaugh

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

$\backslash cd$	00	01	11	10
$\backslash ab$	00	1	1	1
00	1	1	1	1
01	0	1	1	0
11	0	0	1	0
10	1	0	1	1

$\backslash cd$	00	01	11	10
$\backslash ab$	00	1	1	1
00	1	1	1	1
01	0	1	1	0
11	0	0	1	0
10	1	0	1	1

$$f(a, b, c, d) = \bar{a}\bar{b} + cd + a\bar{b}\bar{d} + \bar{a}b\bar{c}d = cd + \bar{a}d + \bar{b}\bar{d}$$

Circuits combinatoire et arithmétiques

- ▶ Multiplexeur : 2^n signaux en entrée (données), n lignes de sélection, 1 sortie
- ▶ Demultiplexeur : 1 signal d'entrée, n lignes de sélection, 2^n lignes de sortie
- ▶ Décodeur : Sélectionner une ligne de sortie
- ▶ Codeur : Faire correspondre une entrée parmi à n sorties
- ▶ Décaleur : décaler des bits d'entrée a gauche ou a droite
- ▶ Additionneur : additionner des bits en tenant compte des retenues

Multiplexeur

Multiplexeur $2^n \times n$

- ▶ Rediriger une donnée parmi plusieurs vers une sortie unique,
- ▶ Données : 2^n signaux en entrée,
- ▶ n lignes de sélection,
- ▶ 1 sortie

Exemple :
Multiplexeur 8×3

a	b	c	S
0	0	0	D_0
0	0	1	D_1
0	1	0	D_2
0	1	1	D_3
1	0	0	D_4
1	0	1	D_5
1	1	0	D_6
1	1	1	D_7

Multiplexeur 8 × 3

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

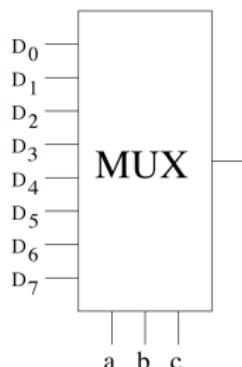
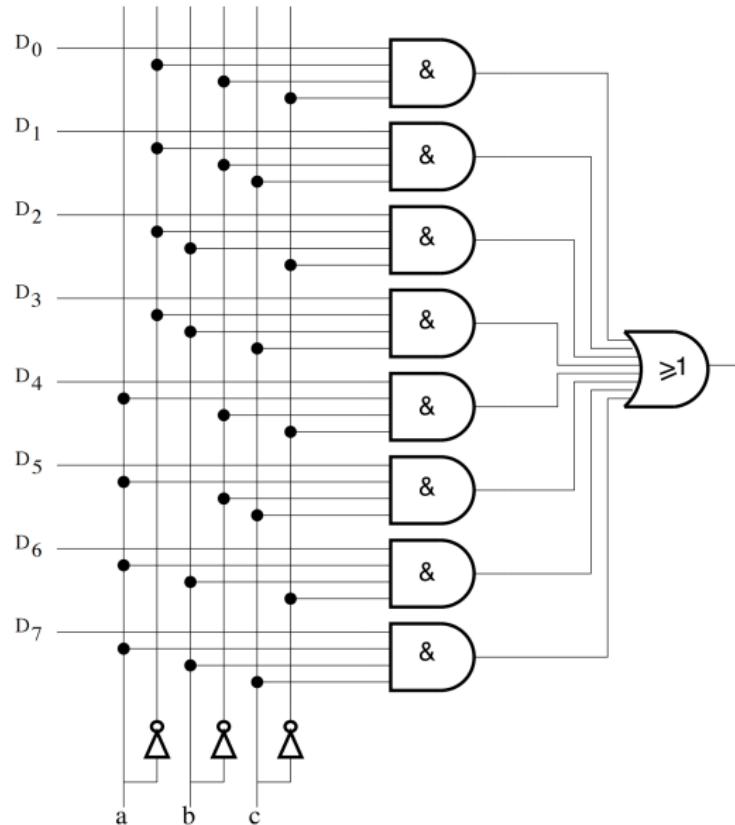
Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture



Multiplexeur Exemple : Fonction majoritaire

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

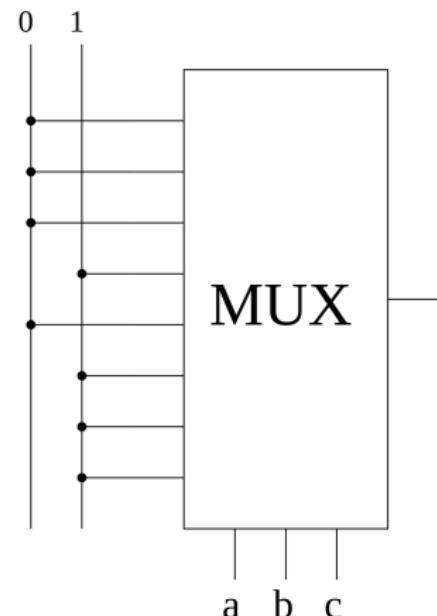
Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

a	b	c	S
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Demultiplexeur

Demultiplexeur $2^n \times n$

- ▶ Diriger une donnée d'entrée vers une sortie parmi 2^n ,
- ▶ n lignes de sélection,

a	b	c	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
0	0	0	E	0	0	0	0	0	0	0
0	0	1	0	E	0	0	0	0	0	0
0	1	0	0	0	E	0	0	0	0	0
0	1	1	0	0	0	E	0	0	0	0
1	0	0	0	0	0	0	E	0	0	0
1	0	1	0	0	0	0	0	E	0	0
1	1	0	0	0	0	0	0	0	E	0
1	1	1	0	0	0	0	0	0	0	E

Demultiplexeur 8 × 3

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

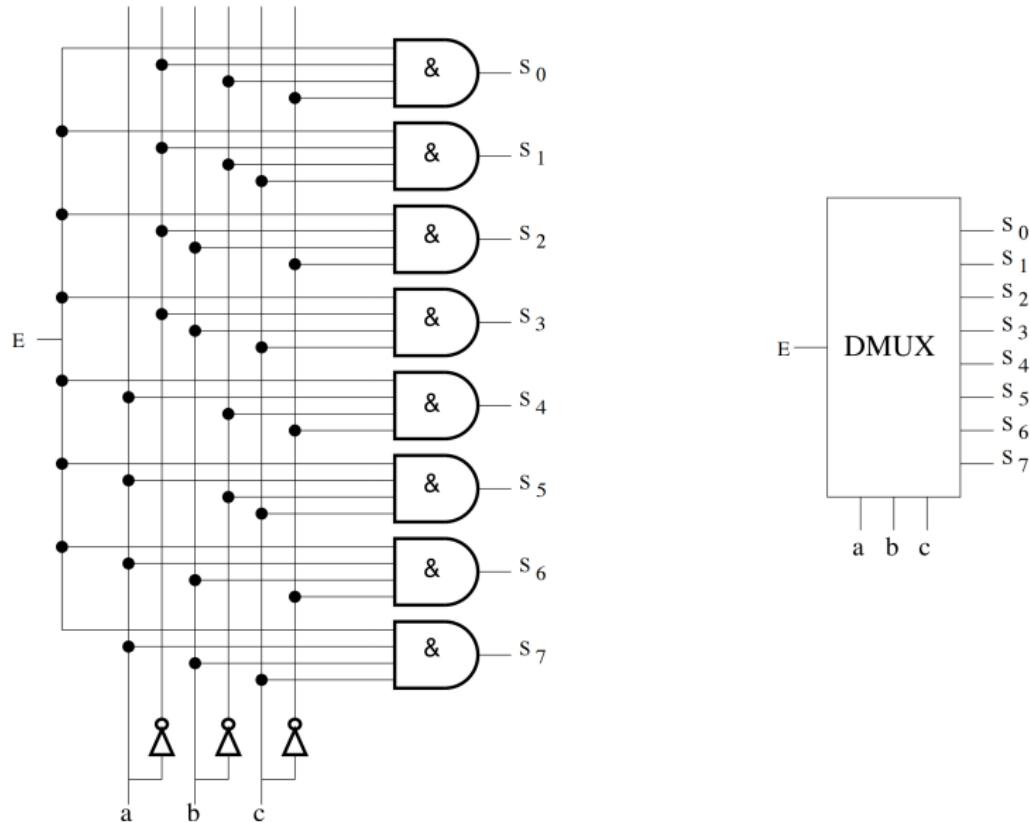
Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

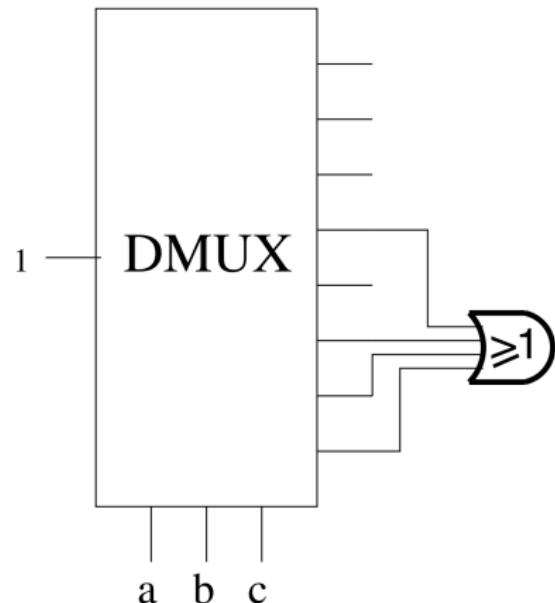
Circuits
logiques

Micro-
architecture



Démultiplexeur Exemple : Fonction majoritaire

a	b	c	S
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



Décodeur

Décodeur $2^n \times n$

- ▶ Multiplexeur dont l'entrée est toujours 1 :
- ▶ Sélectionner une parmi 2^n sorties possibles,
- ▶ n lignes de sélection,

a	b	c	S_0	S_1	S_2	S_3	S_4	S_5	S_6	S_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

Décodeur 8×3

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

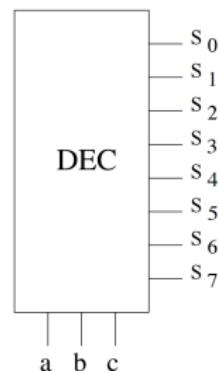
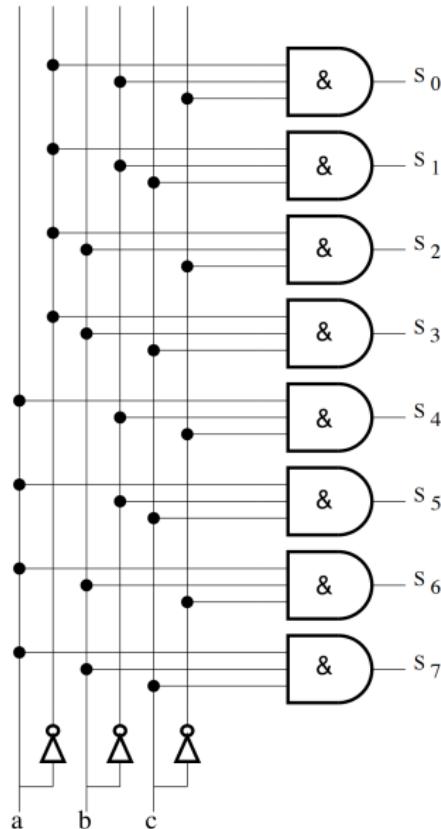
Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

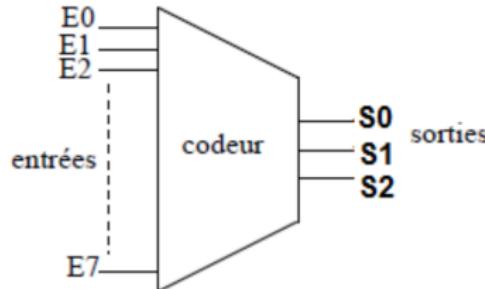


Démultiplexeur avec E=1.

Codeur

Codeur $2^n \times n$

- ▶ Inverse Décodeur
- ▶ Activation d'une parmi 2^n entrées
- ▶ n sorties



Entrée activée (= 1)	Sorties		
	S2	S1	S0
E0	0	0	0
E1	0	0	1
E2	0	1	0
E3	0	1	1
E4	1	0	0
E5	1	0	1
E6	1	1	0
E7	1	1	1

Codeur 8 × 3

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

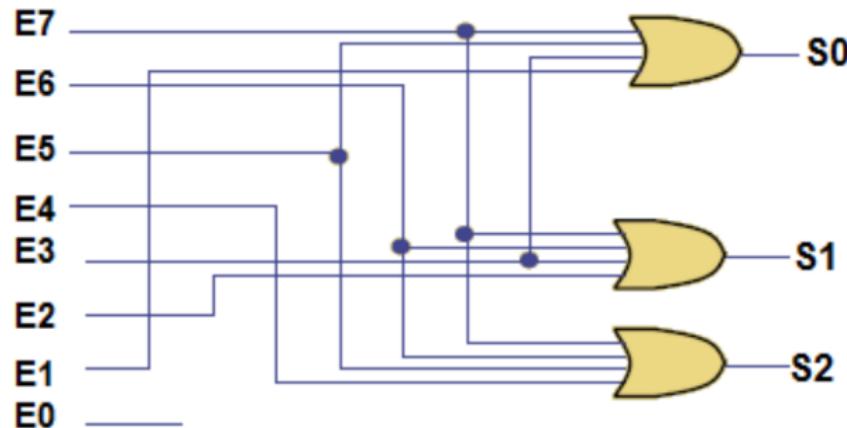
Circuits
logiques

Micro-
architecture

$$S_0 = E_1 + E_3 + E_5 + E_7$$

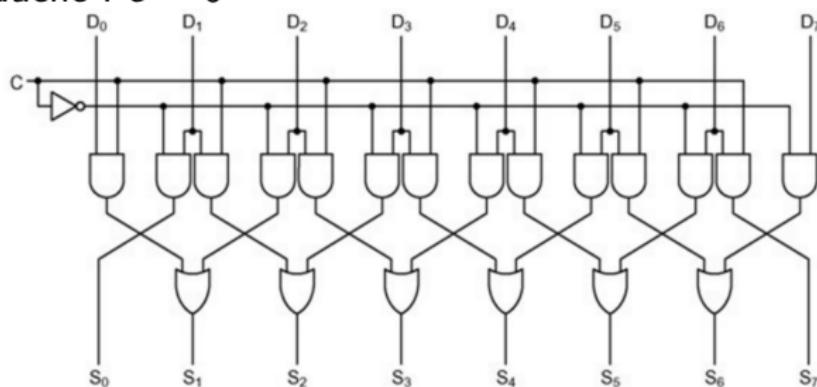
$$S_1 = E_2 + E_3 + E_6 + E_7$$

$$S_2 = E_4 + E_5 + E_6 + E_7$$



Décaleur

- ▶ Décaler n bits
- ▶ Droite : $c = 1$
- ▶ Gauche : $c = 0$



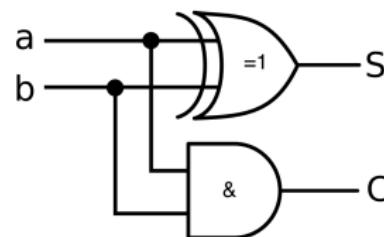
Division euclidienne/multiplication par 2

1/2 Additionneur

Additionner 2 bits a et b en gardant la retenue

- ▶ 2 entrées : bits a et b
- ▶ 2 sorties : somme ($a \oplus b$) et retenue

a	b	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

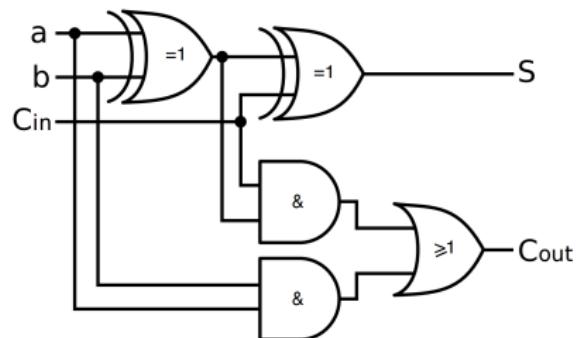


Additionneur complet

Deux 1/2 Additionneurs

- ▶ 3 entrées : bits a et b et retenue d'entrée
 - ▶ 2 sorties : somme ($a + b + \text{retenue d'entrée}$) et retenue de sortie
 - ▶ n étages pour additionner deux nombres de n bits

a	b	C_{in}	S	C_{out}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



C_{out} : 1 s'il y a une retenue.

Exercices

- 1 Donner les circuits logiques réalisant un multiplexeur et un démultiplexeur 4×2 . Montrer comment utiliser le démultiplexeur pour concevoir un décodeur 2 bits.
- 2 Concevoir un circuit permettant de détecter la parité d'un mot de 4 bits codé sur les entrées A, B, C et D. La sortie vaudra 0 si le nombre de 1 en entrée est pair (ex : 0011) et 1 sinon (ex :1000).

 - 1 Ecrire la table de vérité correspondante.
 - 2 Utiliser un multiplexeur 16×4 pour réaliser cette fonction.
 - 3 Utiliser un démultiplexeur 4×16 pour réaliser cette fonction.
- 3 Concevoir un additionneur 4 bits avec retenues d'entrée et de sortie.

Micro-architecture processeur

Architecture des ordinateurs

Wilfried Segretier
wsegreti@univ-ag.fr

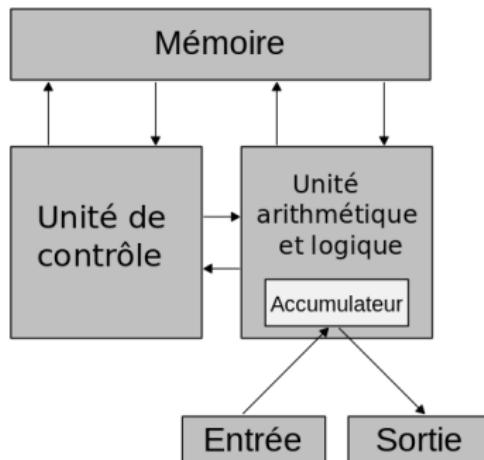
Introduction

Histoire de l'ordinateur et principes généraux

Représentation des informations

Circuits logiques

Micro-architecture

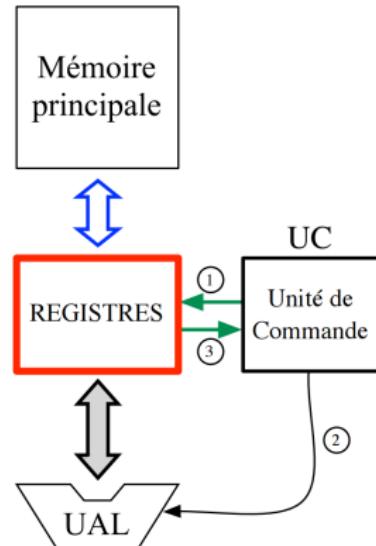


- ▶ Un processeur est composé de circuits logiques organisés
- ▶ UC (Unité de contrôle)
- ▶ UAL (Unité arithmétique et logique)
- ▶ Mécanismes d'interaction mémoire
- ▶ Registres
- ▶ Micro-instructions
- ▶ Langage machine

Processeur : fonctionnement

- 1 Activation de registres par l'UC
lecture/écriture mémoire
transfert données
vers/depuis UAL
- 2 L'UC commande l'action de l'UAL
- 3 L'état des registres permet de choisir la prochaine commande

Intéractions
mémoire/registres/UAL/UC



Registres

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

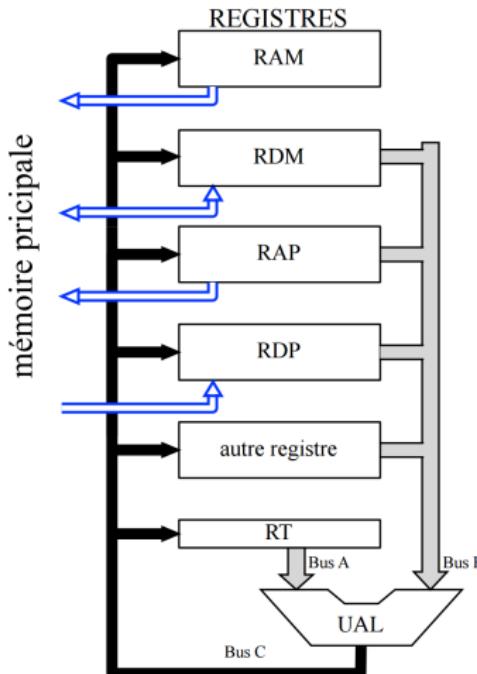
Introduction

Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

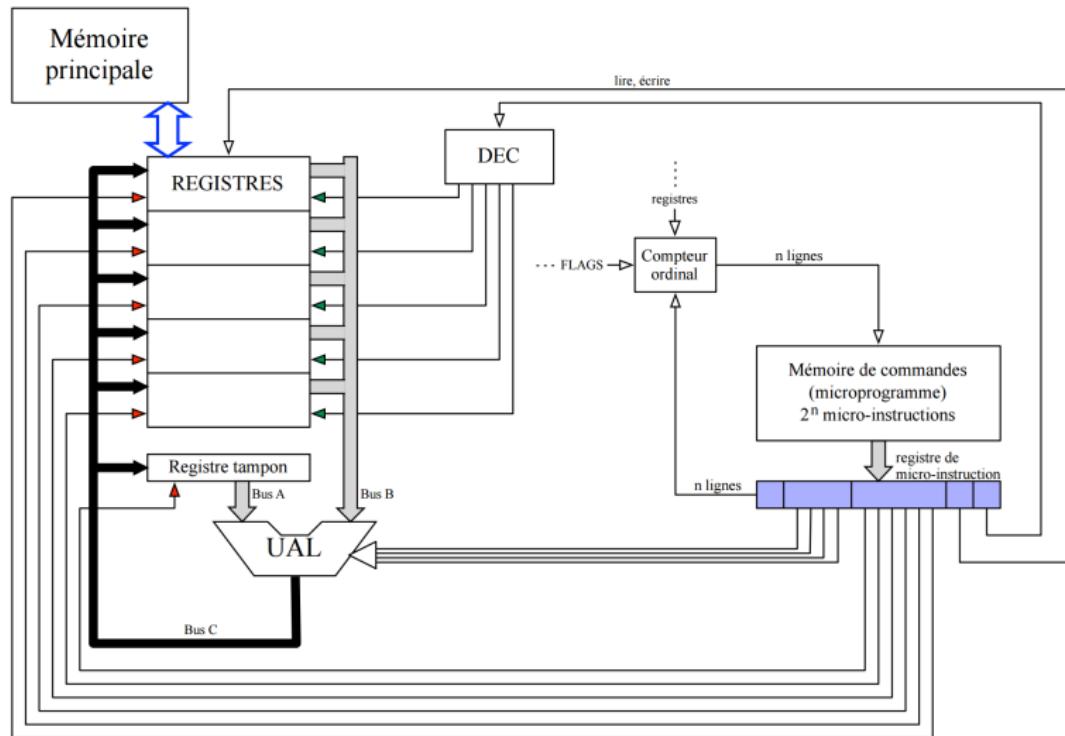
Circuits
logiques

Micro-
architecture



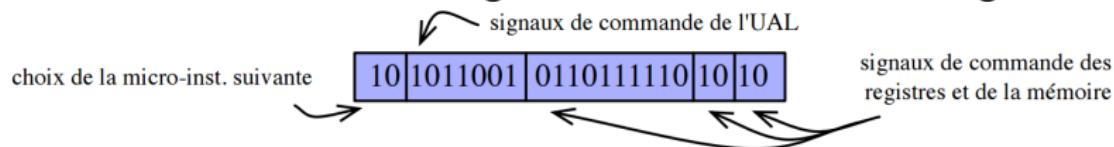
- ▶ Emplacement mémoire situés sur le processeur
- ▶ Accès le plus rapide
- ▶ Registres spécialisés
- ▶ Registre d'Adresses Mémoire (RAM)
- ▶ Registre de Données Mémoire (RDM)
- ▶ Registre d'Adresses de Programme (RAP)
- ▶ Registre de Données de Programme (RDP)
- ▶ Registre Tampon (RT)
- ▶ Registres gestion pile : SP, BP
- ▶ Registres de calcul

Exemple Micro Architecture



Micro-instructions

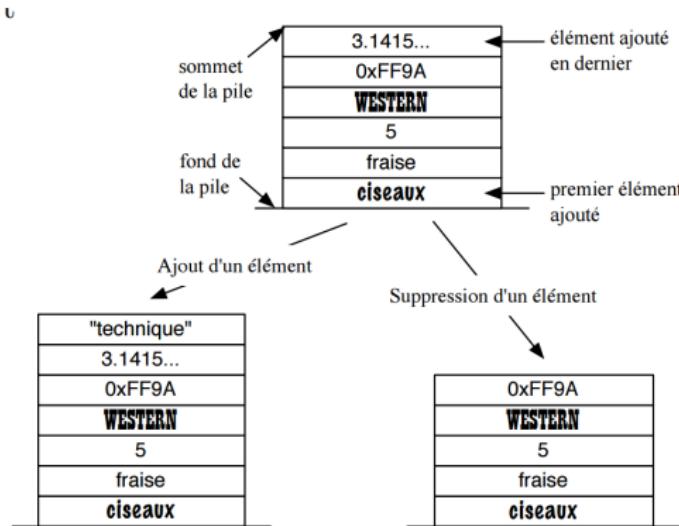
Mots binaire contenant des signaux de commande UAL, Registres



- ▶ Micro-code : Ensemble de micro-instructions en mémoire ROM (Read Only Memory)
- ▶ Instruction : bloc de micro-instructions effectuant une opération
- ▶ ADD, SUB, AND, OR, NOT
- ▶ MOV, PUSH, POP, JUMP, LOOP, CALL

Pile mémoire

- ▶ Espace de stockage en mémoire principale
- ▶ Structure LIFO (push, pop)
- ▶ taille variable, pointeurs sommet, fond



Pile : Exemple simple

Architecture
des
ordinateurs

Wilfried
Segretier
wsegreti@univ-
ag.fr

Introduction

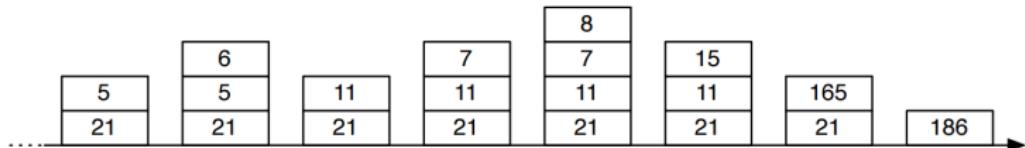
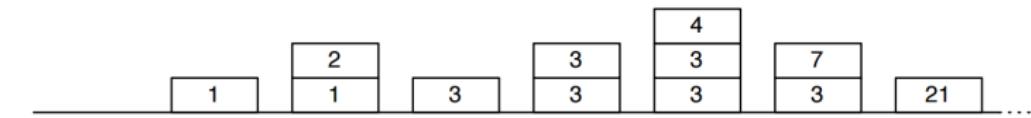
Histoire de
l'ordinateur
et principes
généraux

Représentation
des
informations

Circuits
logiques

Micro-
architecture

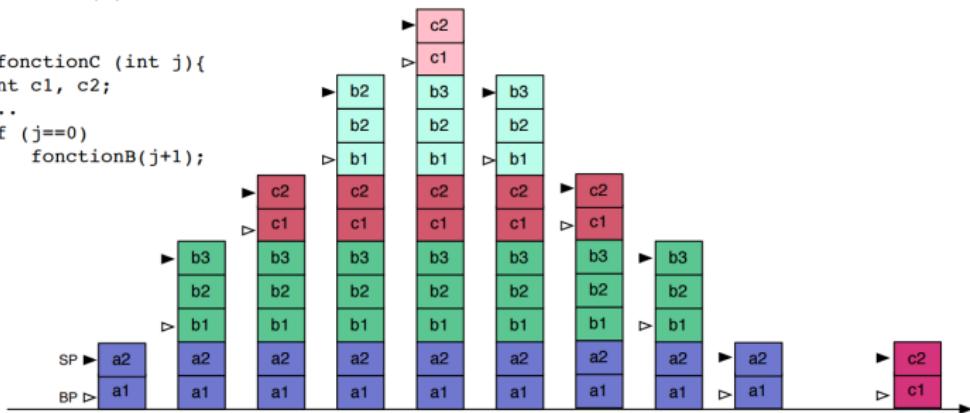
Calcul de $((1 + 2) \times (3 + 4)) + ((5 + 6) \times (7 + 8))$



Pile : Exemple variable locale C

```
void fonctionA (void){  
    int a1, a2;  
    ...  
    fonctionB(0);  
}  
  
void fonctionB (int i){  
    int b1, b2, b3;  
    ...  
    fonctionC(i);  
}  
  
void fonctionC (int j){  
    int c1, c2;  
    ...  
    if (j==0)  
        fonctionB(j+1);  
}
```

```
int main(int argc, char *argv[]){
    fonctionA();
    fonctionC(1);
    return 0;
}
```



Instruction arithmetique : ADD

- ▶ Principe : Additionner les deux premiers éléments se trouvant sur la pile puis placer le résultat au sommet de la pile
- ▶ Détails :
- ▶ RAM : Récupération du sommet de pile SP
- ▶ RDM : Récupération mot pointé par SP et mise dans RT (1er mot)
- ▶ Sélection de l'adresse en dessous de SP
- ▶ mise dans RAM et dans SP (nouveau)
- ▶ RDM : Récupération mot pointé par SP (2eme mot)
- ▶ Addition de RT et RDM : résultat dans RDM
- ▶ Ecriture du résultat en mémoire (grace a RAM)

opération sur la pile : PUSH

- ▶ Principe : mettre la variable locale V (se trouvant dans RDP) au sommet de la pile
- ▶ Détails :
- ▶ RAM : Récupération de l'adresse de la variable locale
- ▶ RDM : Récupération de la valeur de la variable locale
- ▶ RAM et SP : adresse du nouveau sommet
- ▶ Ecriture en mémoire du contenu de RDM à l'adresse pointée par RAM

Appels : CALL, RET

Architecture des ordinateurs

Wilfried Segretier
wsegepri@univ-ag.fr

Introduction

Histoire de l'ordinateur et principes généraux

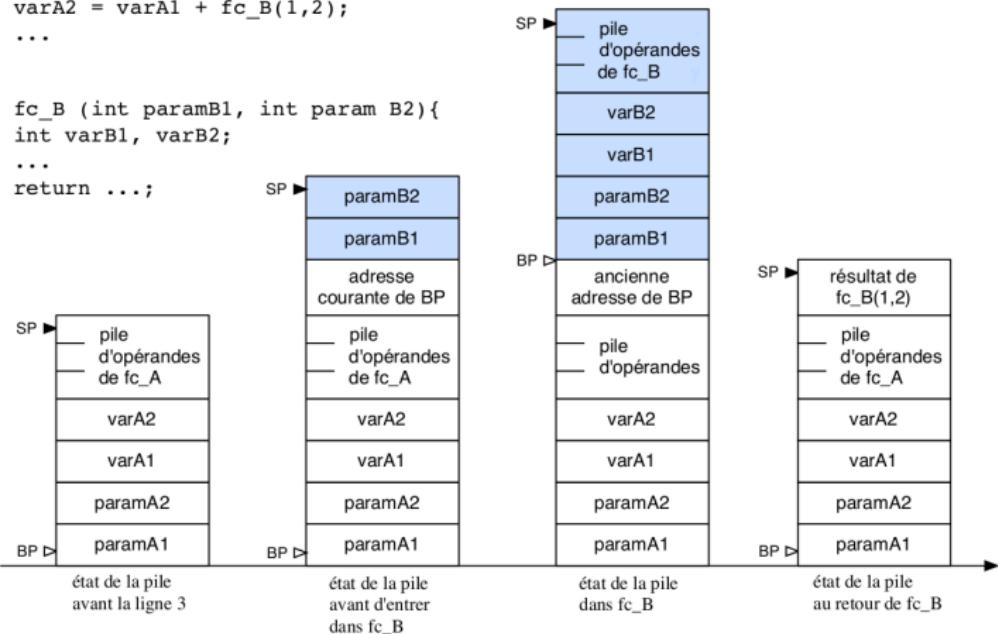
Représentation des informations

Circuits logiques

Micro-architecture

```

1  fc_A (int paramA1, int param A2){
2      int varA1, varA2;
3      ...
4      varA2 = varA1 + fc_B(1,2);
5      ...
6      return ...;
    }
```

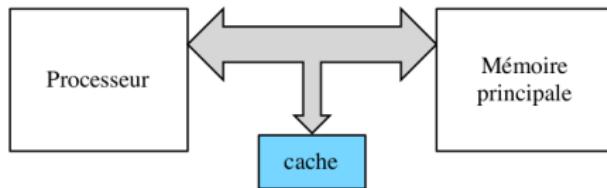


Mémoire principale, mémoire cache

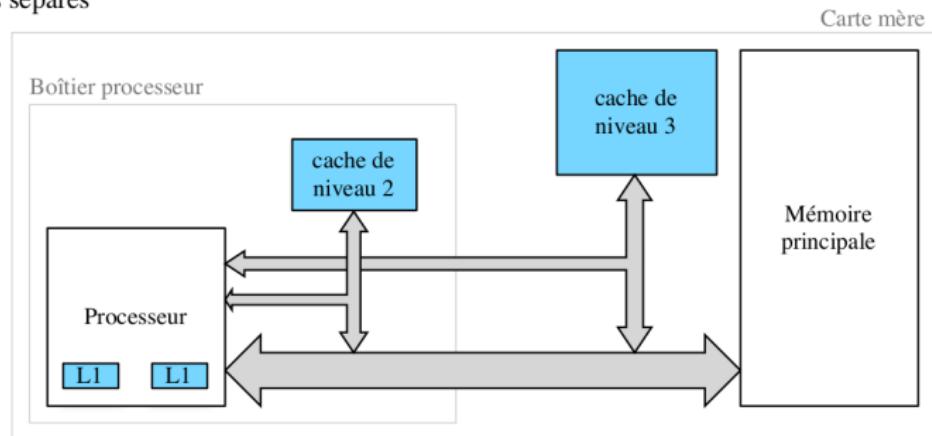
- ▶ Mémoire principale :
- ▶ Volume important, plusieurs gigaoctets, cout faible
- ▶ Accès lent, plusieurs cycles d'horloge nécessaires
- ▶ non optimal
- ▶ Solution : mémoire cache (niveau 1, niveau 2, etc ...)
- ▶ mémoire intermédiaire
- ▶ localisation plus proche (processeur)
- ▶ accès beaucoup plus rapide (en fonction du niveau)
- ▶ volume relativement faibles (quelques mégaoctets)
- ▶ cout élevé

Mémoire principale, mémoire cache

Cache unique



Caches séparés



Questions d'application

- 1 A quoi fait référence une adresse mémoire ? 1 bit ? 1 octet ? 1 mot ?
- 2 Si l'adresse d'un mot est *adr*, quelle est l'adresse du mot suivant ?
- 3 Où se situe la pile ?
- 4 Comment savoir quelle est l'adresse de la base de la pile ? du sommet de la pile ?
- 5 Comment savoir quel mot se trouve au sommet de la pile ?
- 6 Nous avons vu le fonctionnement de ADD et PUSH. Proposez un ensemble d'étapes pour SUB et POP.
- 7 Réalisez un schéma montrant l'évolution de la pile lors des calculs suivants
$$(5 \times 3 + 6 \times 2 + 1)/2$$
$$((1 + 1) \times (2 + 1)) + (3 \times 4) + (4 \times 2)$$
$$(((8 + 3 + 2) \times 2) \times ((8/2 \times 5) + 6))$$

Architecture en couches

5. Langages haut niveau

Compilation

4. Langage d'assemblage

Assembleur

3. Système d'exploitation

Appels système

2. Jeu d'instructions propre à chaque machine (ISA)

Microprogrammes : micro-instructions binaires

1. Micro-architecture (UAL, opérations, registres, ...)

Assemblage physique des portes logiques

0. Circuits logiques

2 grandes catégories de processeurs, qui se distinguent par la conception de leurs jeux d'instructions :

- **CISC (Complex Instruction Set Computer)**

- jeu étendu d'instructions complexes
- 1 instruction peut effectuer plusieurs opérations élémentaires (ex : charger une valeur en mémoire, faire une opération arithmétique et ranger le résultat en mémoire)
- instructions proches des constructions typiques des langages de haut niveau
- Exemples : Motorola 68000, x86 Intel, AMD...

- **RISC (Reduced Instruction Set Computer)**

- jeu d'instructions réduit
- 1 instruction effectue une seule opération élémentaire (micro-instruction)
- plus uniforme (même taille, s'exécute en un cycle d'horloge)
- Exemples : Motorola 6800, PowerPC, UltraSPARC (Sun), ...

Intel Architecture 32 bits : architecture des Pentium.

Aussi appelée **x86** (architecture de l'ensemble des processeurs Intel à partir du 8086).

Assembleur = programme convertissant les instructions du langage d'assemblage en micro-instructions.

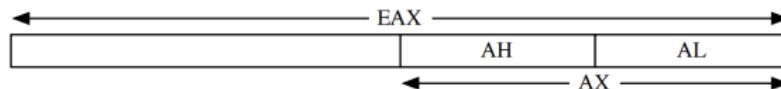
Remarque : compilateur = programme similaire pour les langages de haut niveau (C, Java, ...).

Chaque type de processeur a son propre langage machine ⇒ il a également son propre assembleur.

En TP : NASM (Netwide Assembler)

Registres généraux

- EAX : registre accumulateur (*accumulator reg.*) pour les opérations arithmétiques et le stockage de la valeur de retour des appels systèmes.
- ECX : registre compteur (*counter reg.*)
- EBX : registre de base (*base reg.*)
- EDX : registre de données (*data reg.*) pour les opérations arithmétiques et les opérations d'E/S.
- AX : 16 bits de poids faible de EAX (idem BX, CX, DX)
- AL : octet de poids faible de AX (idem BL, CL, DL)
- AH : octet de poids fort de AX (idem BH, CH ,DH)



Registres spécialisés

▷ Registres d'adresses

- ESI : pointeur source (*Extended Source Index*)
- EDI : pointeur destination (*Extended Destination Index*)
- EBP : pointeur de base (*Extended Base Pointer*)
- ESP : pointeur de pile (*Extended Stack Pointeur*)

▷ Autres registres

- EIP : pointeur d'instruction
- EFLAGS : registre d'états (drapeaux)
- CS, SS, DS, ES, FS, GS : registres de segment (16 bits) : adresses et données de programme

- Zero Flag (ZF)
1 si les deux opérandes utilisés sont égaux, 0 sinon.
- Overflow Flag (OF)
1 si le dernier résultat a provoqué un overflow, 0 sinon.
- Carry Flag (CF)
1 si la dernière opération a générée une retenue (mode positif), 0 sinon.
- Sign Flag (SF)
1 si la dernière opération a générée un résultat négatif, 0 s'il est positif ou nul.
- Parity Flag (PF)
1 si la dernière opération a générée un résultat impair, 0 s'il est pair (nombre de bits à 1).
- Interrupt Flag (IF)
1 si les interruptions sont autorisées, à 0 sinon.

Instructions

INSTRUCTION = OPÉRATION suivi d'OPÉRANDES (de 0 à 3)

▷ une opérande est :

- soit une **donnée brute** :
 - *adressage immédiat* : valeur binaire, décimale ou hexadécimale

Exemples : `mov eax, 16` (décimal)
`mov eax, 0b11` (binaire)
`mov eax, 0xff` (hexadécimal)
 - *adressage implicite* : pas spécifié explicitement, par exemple l'incrémentation (le 1 est implicite)
- soit une **adresse** : avec différents *modes d'adressage*.

! les types d'opérandes autorisés dépendent de l'opération effectuée.

Notation : A : adresse A \neq [A] : donnée à l'adresse A

Adressage

Les opérandes peuvent avoir les types suivants :

- *adressage direct* : l'opérande est une **adresse** (32 bits) en mémoire. désigne toujours le même emplacement, mais la valeur correspondante peut changer (ex : variable globale)
Exemple : `mov eax, [0x0000f13a]`
- *adressage par registre* : l'opérande est un **registre**. mode le plus courant (+ efficace)
Exemple : `mov eax, ebx`
- *adressage indirect par registre* : l'opérande une **adresse** mémoire contenue **dans un registre** (qui sert de pointeur)
Exemples : `mov eax, [esp]` (eax \leftarrow sommet de la pile)
!! `mov eax, esp` (eax \leftarrow **adresse** du sommet)
- *adressage indexé* : l'opérande une **adresse** mémoire contenue **dans un registre** associée à un **décalage**
Exemple : `mov eax, [esp+4]`

Catégories d'instructions

Grandes catégories d'opérations :

- Opérations de **transfert** :
entre la mémoire et les registres ; opérations sur la pile.
- Opérations **arithmétiques**
- Opérations **logiques**
- Opérations de **décalage** et **rotation**
multiplications et divisions rapides
- Opérations de **branchement**
sauts, boucles, **appels de fonctions**
- Opérations sur les chaînes de caractères

Transfert

- ▷ Copie de données entre mémoire et registres : **mov**
Le 1er argument est toujours la *destination* et le 2nd la *source*
Restriction sur le type d'opérandes :

```
mov registre, mémoire
mov mémoire, registre
mov registre, registre (registres de même taille!)
mov type mémoire, immédiate (type=byte, word, dword)
mov registre, immédiate
mov mémoire, mémoire impossible!!
```

- ▷ Instruction spéciale pour échanger les contenus de 2 registres ou d'un registre et d'une case mémoire : **xchg**
- Exemples : **xchg eax, ebx**
- ```
xchg eax, [0xbffffeedc]
```

# Transfert

▷ Opérations de pile : **push** et **pop**  
adressage immédiat ou par registre

|                                                 |                                          |
|-------------------------------------------------|------------------------------------------|
| <b>Exemples :</b> push      eax<br>pop      ebx | push      word 42<br>pop      word [adr] |
|-------------------------------------------------|------------------------------------------|

!! la pile est à l'envers :  
 si **esp** est l'adresse du sommet de la pile,  
 alors l'élément *en dessous* est **[esp+4]**

## Résumé

| Instruction |            |            | Description                                               |
|-------------|------------|------------|-----------------------------------------------------------|
| <b>mov</b>  | <i>dst</i> | <i>src</i> | déplace <i>src</i> dans <i>dst</i>                        |
| <b>xchg</b> | <i>ds1</i> | <i>ds2</i> | échange <i>ds1</i> et <i>ds2</i>                          |
| <b>push</b> | <i>src</i> |            | place <i>src</i> au sommet de la pile                     |
| <b>pop</b>  | <i>dst</i> |            | supprime le sommet de la pile et le place dans <i>dst</i> |

# Instructions arithmétiques

▷ Addition entière (en cpt à 2) : **add**

2 opérandes : destination et source : valeurs, registres ou mémoire (au moins 1 reg.)

positionne les **FLAGS** : Carry (CF) et Overflow (OF)

Exemples : **add ah, bl**

**add ax, bl**    opérandes imcompatibles !

▷ Addition avec retenue : **adc**

additionne les 2 opérandes et la retenue positionnée dans **CF**

Exemple : **adc eax, ebx**    ( $\text{eax} \leftarrow \text{eax} + \text{ebx} + \text{CF}$ )

▷ Multiplication entière positive : **mul**

1 seule opérande : multiplication par **eax**

le résultat est stocké dans deux registres : **edx** pour les bits de poids fort et **eax** pour les bits de poids faible

Exemple : **mul ebx**    ( $\text{edx} | \text{eax} \leftarrow \text{eax} \cdot \text{ebx}$ )

# Instructions arithmétiques

▷ Multiplication entière en cpt à 2 : **imul**

mêmes caractéristiques que **mul** avec des entiers relatifs

## Résumé

|             |            |            |                                                                                 |
|-------------|------------|------------|---------------------------------------------------------------------------------|
| <b>add</b>  | <i>dst</i> | <i>src</i> | ajoute <i>src</i> à <i>dst</i>                                                  |
| <b>adc</b>  | <i>dst</i> | <i>src</i> | ajoute <i>src</i> à <i>dst</i> avec retenue                                     |
| <b>sub</b>  | <i>dst</i> | <i>src</i> | soustrait <i>src</i> à <i>dst</i>                                               |
| <b>sbb</b>  | <i>dst</i> | <i>src</i> | soustrait <i>src</i> à <i>dst</i> avec retenue                                  |
| <b>mul</b>  | <i>src</i> |            | multiplie <b>eax</b> par <i>src</i> (résultat dans <b>edx eax</b> )             |
| <b>imul</b> | <i>src</i> |            | multiplie <b>eax</b> par <i>src</i> (cpt à 2)                                   |
| <b>div</b>  | <i>src</i> |            | divise <b>edx eax</b> par <i>src</i> ( <b>eax</b> =quotient, <b>edx</b> =reste) |
| <b>idiv</b> | <i>src</i> |            | divise <b>edx eax</b> par <i>src</i> (cpt à 2)                                  |
| <b>inc</b>  | <i>dst</i> |            | $1 + dst$                                                                       |
| <b>dec</b>  | <i>dst</i> |            | $dst - 1$                                                                       |
| <b>neg</b>  | <i>dst</i> |            | $-dst$                                                                          |

# Instructions logiques

Les opérations logiques sont des opérations bit à bit.

▷ Et logique : **and**

2 opérandes : destination et source

**Exemple :** Utilisation d'un masque pour l'extraction des 4 bits de poids faible de ax : and ax, 0b00001111

## Résumé

|     |            |            |                                                     |
|-----|------------|------------|-----------------------------------------------------|
| not | <i>dst</i> |            | place (not <i>dst</i> ) dans <i>dst</i>             |
| and | <i>dst</i> | <i>src</i> | place ( <i>src</i> AND <i>dst</i> ) dans <i>dst</i> |
| or  | <i>dst</i> | <i>src</i> | place ( <i>src</i> OR <i>dst</i> ) dans <i>dst</i>  |
| xor | <i>dst</i> | <i>src</i> | place ( <i>src</i> XOR <i>dst</i> ) dans <i>dst</i> |

# décalages/rotations

2 opérandes : un registre suivi d'un nombre de décalages  $nb$ .

▷ Décalage logique à gauche : **shl**

Insertion de  $nb$  0 au niveau du bit de poids faible.

Permet d'effectuer efficacement la **multiplication par  $2^{nb}$** .

Exemple : **shl al, 4** (ex : 01100111 → 0111**0000**)

▷ Décalage arithmétique à droite : **sar**

Insertion de  $nb$  copies du bit de poids fort à gauche.

Permet la **division rapide d'un entier relatif par  $2^{nb}$** .

Exemple : **sar al, 4** (ex : 10011110 → **11111001**)

▷ Rotation à gauche : **rol**

rotation de  $nb$  bits vers la gauche : les bits sortants à gauche sont immédiatement réinjectés à droite.

Exemple : **rol al, 3** (ex : **100**11111 → 11111**100**)

# décalages/rotations

▷ Rotation à droite avec retenue : **rcr**

Rotation de *nb* bits vers la droite en passant par la retenue : lors d'un décalage, le bit sortant à droite est mémorisé dans la retenue qui est elle-même réinjectée à gauche.

**Exemple :** rcr al, 3 (ex : 11111101, c = 0 → 01011111, c = 1)

## Résumé

|            |            |           |                                                                |
|------------|------------|-----------|----------------------------------------------------------------|
| <b>sal</b> | <i>dst</i> | <i>nb</i> | décalage arithmétique à gauche de <i>nb</i> bits de <i>dst</i> |
| <b>sar</b> | <i>dst</i> | <i>nb</i> | décalage arithmétique à droite de <i>nb</i> bits de <i>dst</i> |
| <b>shl</b> | <i>dst</i> | <i>nb</i> | décalage logique à gauche de <i>nb</i> bits de <i>dst</i>      |
| <b>shr</b> | <i>dst</i> | <i>nb</i> | décalage logique à droite de <i>nb</i> bits de <i>dst</i>      |
| <b>rol</b> | <i>dst</i> | <i>nb</i> | rotation à gauche de <i>nb</i> bits de <i>dst</i>              |
| <b>ror</b> | <i>dst</i> | <i>nb</i> | rotation à droite de <i>nb</i> bits de <i>dst</i>              |
| <b>rcl</b> | <i>dst</i> | <i>nb</i> | rotation à gauche de <i>nb</i> bits de <i>dst</i> avec retenue |
| <b>rcr</b> | <i>dst</i> | <i>nb</i> | rotation à droite de <i>nb</i> bits de <i>dst</i> avec retenue |

# Branchement

▷ Instruction de comparaison : **cmp**

Effectue une soustraction (comme **sub**), mais ne stocke pas le résultat : **seuls les drapeaux sont modifiés.**

**Exemple :** **cmp eax, ebx** (si eax=ebx, alors ZF=1)

▷ Saut conditionnel vers l'étiquette spécifiée : **jxx**

- **je, jne** : jump if (resp. if not) equal  
saute si le drapeau d'égalité (positionné par **cmp**) est à 1 (resp. à 0).
- **jge, jnge** : jump if (resp. if not) greater or equal  
saute si le résultat de **cmp** est (resp. n'est pas) *plus grand ou égal à*.
- **jl, jnl** : jump if (resp. if not) less than  
saute si le résultat de **cmp** est (resp. n'est pas) *stt plus petit que*.
- **jo, jno** : jump if (resp. if not) overflow
- **jc, jnc** : jump if (resp. if not) carry
- **jp, jnp** : jump if (resp. if not) parity
- **jcxz, jecxz** jump if cx (resp. ecx) is null  
sautent quand le registre cx (resp. ecx) est nul

# Branchement

▷ Boucle fixe : **loop**

$\text{ecx} \leftarrow \text{ecx}-1$  et saute vers l'étiquette si  $\text{ecx} \neq 0$ .

▷ Boucle conditionnelle : **loop<sub>e</sub>**

$\text{ecx} \leftarrow \text{ecx}-1$  et saute vers l'étiquette si  $\text{ZF} = 1$  et  $\text{ecx} \neq 0$ .

▷ Boucle conditionnelle : **loop<sub>ne</sub>**

$\text{ecx} \leftarrow \text{ecx}-1$  et saute vers l'étiquette si  $\text{ZF} = 0$  et  $\text{ecx} \neq 0$ .

## Résumé

|                         |            |            |                                                                         |
|-------------------------|------------|------------|-------------------------------------------------------------------------|
| <b>cmp</b>              | <i>sr1</i> | <i>sr2</i> | compare <i>sr1</i> et <i>sr2</i>                                        |
| <b>jmp</b>              | <i>adr</i> |            | saut vers l'adresse <i>adr</i>                                          |
| <b>j<sub>xx</sub></b>   | <i>adr</i> |            | saut conditionné par <i>xx</i> vers l'adresse <i>adr</i>                |
| <b>loop</b>             | <i>adr</i> |            | répétition de la boucle <i>nb</i> de fois ( <i>nb</i> dans <i>ecx</i> ) |
| <b>loop<sub>x</sub></b> | <i>adr</i> |            | répétition de la boucle conditionnée par <i>x</i>                       |

# Exemples

## Architecture des ordinateurs

Wilfried Segretier  
wsegreti@univ-ag.fr

## Introduction

## Histoire de l'ordinateur et principes généraux

## Représentation des informations

## Circuits logiques

## Micro-architecture

|                                                                                                                                                  |                                                                                                                                                                   |
|--------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <pre>         mov    ecx, [esp]         mov    eax, 0 next: add    eax, ecx       dec    ecx       cmp    ecx, 0       jne    next </pre>        | <pre>         mov    ecx, [esp]         mov    eax, 0 next: add    eax, ecx       loop   next </pre>                                                              |
| <hr/> <pre>         mov    ecx, [esp]         mov    eax, 0 test: jecxz end       add    eax, ecx       dec    ecx       jump   test end: </pre> | <pre>         mov    ebx, [esp]         mov    eax, 0         mov    ecx, 100 next: add    eax, ecx       dec    ebx       cmp    ebx, 0       loopne next </pre> |

# Structure d'un programme assembleur

Architecture  
des  
ordinateurs

Wilfried  
Segretier  
wsegreti@univ-  
ag.fr

Introduction

Histoire de  
l'ordinateur  
et principes  
généraux

Représentation  
des  
informations

Circuits  
logiques

Micro-  
architecture

- programme en assembleur = fichier texte (extension .asm)
- organisé en plusieurs **SECTIONS** (= segments)
- sections **differentes** pour les **données** et le **code**
- *directives* pour NASM  $\neq$  instructions pour le processeur
- une seule instruction par ligne, séparateur = chgt de ligne
- 1 ligne de code = 4 champs (certains **optionnels**) :

**étiquette:** instruction opérandes ; commentaire

# Données

## ▷ Données initialisées : SECTION .data

- déclarer des *données initialisées* avec la directive : **dx**
- $X = b$  (1 octet),  $w$  (2 octets) ou  $d$  (4 octets = 1 mot).
- exemples :

```

11: db 0x55 ; l'octet 0x55
 db 0x55,0x66,0x77 ; 3 octets successifs
 dw 0x1234 ; 0x34 0x12 (little endian)
 dw 'a' ; 0x61 0x00
12: dw 'ab' ; 0x61 0x62 (caractères)
13: dw 'abc' ; 0x61 0x62 0x63 0x00 (string)

```

- définir des *constantes non modifiables* avec la directive : **equ**
- exemple : nb\_lettres: equ 26

# Données

- répéter une déclaration avec la directive : **times**

- exemples :

```
p0: times 100 db 0 ; 100 fois l'octet 0x00
p1: times 28 dd 0xffffffff ; 28 fois le même mot
```

## ▷ Données *non initialisées* : SECTION .bss

- déclarer des données *non initialisées* avec la directive : **resX**
- $X = b$  (1 octet),  $w$  (2 octets) ou  $d$  (4 octets = 1 mot).
- exemples (**étiquettes obligatoires**) :

```
input1: resb 100 ; réserve 100 octets
input2: resw 1 ; réserve 2 octets
input3: resd 1 ; réserve 1 mot (4 octets)
```

!!! Étiquette = adresse de la donnée.

▷ Corps du programme : **SECTION .text**

- commencer par déclarer **global** l'étiquette de début de programme (**main**) pour qu'elle soit *visible* :

```
SECTION .text
 global main
```

```
main:
```

```
 ...
```

- fin de fichier :

```
 mov ebx, 0 ; code de sortie, 0 = normal
 mov eax, 1 ; numéro de la commande exit
 int 0x80 ; interruption 80 hex, appel au noyau
```

# Fichier squelette

Architecture  
des  
ordinateurs

Wilfried  
Segretier  
wsegreti@univ-  
ag.fr

Introduction

Histoire de  
l'ordinateur  
et principes  
généraux

Représentation  
des  
informations

Circuits  
logiques

Micro-  
architecture

```
%include "asm_io.inc"
SECTION .data
 ; données initialisées
 ;

SECTION .bss
 ; données non initialisées
 ;

SECTION .text
 global main ; rend l'étiquette visible de l'extérieur

main:
 ; programme
 ;
 mov ebx,0 ; code de sortie, 0 = normal
 mov eax,1 ; numéro de la commande exit
 int 0x80 ; interruption 80 hex, appel au noyau
```

# Ligne de commande

- *Assemblage* : créer un **fichier objet** (transformer le programme écrit en langage d'assemblage en instructions machine)

```
nasasm -g -f <format> <fichier> [-o <sortie>]
```

- Exemples :

```
nasasm -g -f coff toto.asm
nasasm -g -f elf toto.asm -o toto.o
```

- Produire un **listing** des instructions machine :

```
nasasm -g -f elf toto.asm -l toto.lst
```

- **Édition de lien** :

```
ld -e main toto.o -o toto
```

- En utilisant des bibliothèques C ou écrites en C (par exemple, **asm\_io** de P. Carter, pour les E/S) :

```
nasasm -g -f elf toto.asm
gcc toto.o asm_io.o -o toto
```

# Interruptions

- Le **flot** ordinaire d'un programme doit pouvoir être **interrompu** pour traiter des **événements** nécessitant une réponse rapide.
- Mécanisme d'**interruptions** (ex : lorsque la souris est déplacée, le programme en cours est interrompu pour gérer ce déplacement).
- Passage du contrôle à un **gestionnaire d'interruptions**.
- Certaine interruptions sont **externes** (ex : la souris).
- D'autres sont **soulevées par le processeur**, à cause d'une erreur (*traps*) ou d'une instruction spécifique (*interruption logicielle*).
- En général, le gestionnaire d'interruptions **redonne le contrôle au programme** interrompu, une fois l'interruption traitée.
- Il **restaure tous les registres** (sauf **eax**).
- Le programme interrompu s'exécute comme si rien n'était arrivé.
- Les **traps** **arrêtent** généralement le programme.

# Entrées/Sorties

**Entrées-sorties (I/O)** : échanges d'informations entre le processeur et les périphériques.

- *Entrées* : données envoyées par un périphérique (disque, réseau, clavier...) à destination de l'unité centrale.
- *Sorties* : données émises par l'unité centrale à destination d'un périphérique (disque, réseau, écran...).

Gestion par interruptions :

- permet de réagir rapidement à un changement en entrée.
- le périphérique prévient le processeur par une interruption,
- le processeur interrompt la tâche en cours, effectue l'action prévue pour cette interruption et reprend l'exécution du programme principal là où il l'avait laissée.

Gestion "haut-niveau" :

- Bibliothèques standards en C pour les E/S (pas en assembleur).
- MAIS, les conventions d'appels utilisées par C sont complexes...

# Affichage : Interruption 0x80

Architecture  
des  
ordinateurs

Wilfried  
Segretier  
wsegreti@univ-  
ag.fr

Introduction

Histoire de  
l'ordinateur  
et principes  
généraux

Représentation  
des  
informations

Circuits  
logiques

Micro-  
architecture

```
SECTION .data
msg1:db "message 1",10
lg1: equ $-msg1

SECTION .text
global main

main:
 ...
 mov edx, lg1
 mov ecx, msg1
 mov ebx, 1 ; stdout
 mov eax, 4 ; write
 int 0x80
 ...
```

# Affichage : printf

Architecture  
des  
ordinateurs

Wilfried  
Segretier  
wsegreti@univ-  
ag.fr

Introduction

Histoire de  
l'ordinateur  
et principes  
généraux

Représentation  
des  
informations

Circuits  
logiques

Micro-  
architecture

```
extern printf

SECTION .data
msg2: db "msg 2", 10, 0

SECTION .text
global main

main:
...
push msg2
call printf
...
```

# Affichage : asm io.inc

Architecture  
des  
ordinateurs

Wilfried  
Segretier  
wsegreti@univ-  
ag.fr

Introduction

Histoire de  
l'ordinateur  
et principes  
généraux

Représentation  
des  
informations

Circuits  
logiques

Micro-  
architecture

```
%include "asm_io.inc"

SECTION .data
msg3: db "msg 3", 10, 0

SECTION .text
global main

main:
 ...
 mov eax, msg3
 call print_string
 ...
```

|                           |                                                                                                                                                               |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>print_int</code>    | affiche à l'écran la valeur de l'entier stocké dans <code>eax</code>                                                                                          |
| <code>print_char</code>   | affiche à l'écran le caractère dont le code ASCII est stocké dans <code>al</code>                                                                             |
| <code>print_string</code> | affiche à l'écran le contenu de la chaîne de caractères à l'adresse stockée dans <code>eax</code> . La chaîne doit être une chaîne de type C (terminée par 0) |
| <code>print_nl</code>     | affiche à l'écran un caractère de nouvelle ligne                                                                                                              |
| <code>read_int</code>     | lit un entier au clavier et le stocke dans le registre <code>eax</code>                                                                                       |
| <code>read_char</code>    | lit un caractère au clavier et stocke son code ASCII dans le registre <code>eax</code>                                                                        |

- `%include "asm_io.inc"`
- Pour chaque fonction d'affichage,
  - il faut **charger `eax`** avec la valeur correcte,
  - utiliser une instruction **`call`** pour l'invoquer.
- ces fonctions préservent les valeurs de tous les registres,  
sauf les deux **`read`** qui modifient `eax`.

# Sous programmes

- Les sous-programmes servent à *mutualiser* du code (éviter les copier-coller)
- Exemple : les **fonctions** des langages haut niveau.
- Le **code appelant** le sous-programme et le **sous-programme** lui-même doivent se mettre d'accord sur **la façon de se passer les données** (*conventions d'appel*).
- Un **saut** peut être utilisé pour **appeler le sous-programme**, mais le **retour** pose problème.
- Le sous-programme peut être utilisé par différentes parties du programme ➤ il doit **revenir au point où il a été appelé!**
- Donc, le retour du sous-programme ne peut pas être codé “en dur” par un saut vers une étiquette.
- L'étiquette de retour doit être un **paramètre** du sous-programme.

# Exemple sans sous programme

```
%include "asm_io.inc"

SECTION .data
msg1: db "entier <10 ?",10,0
msg2: db "bravo",10,0
msg3: db "perdu",10,0

SECTION .text
 global main

main: mov eax, msg1
 call print_string
 call read_int
 cmp eax, 10
 jl ok
 mov eax, msg3
 call print_string
 jmp fin
ok: mov eax, msg2
 call print_string
fin: mov ebx, 0
 mov eax, 1
 int 0x80
```

# Sous programme simple

```
%include "asm_io.inc"

SECTION .data
msg1: db "entier <10 ?",10,0
msg2: db "bravo",10,0
msg3: db "perdu",10,0

SECTION .text
 global main
main: mov eax, msg1
 call print_string
 call read_int
 mov ebx, eax
 cmp ebx, 10
 jl ok
 jmp rate
ok: mov eax, msg2
 jmp ecx
rate: mov eax, msg3
 jmp ecx
```

# Sous programmes : Call et Ret

## Problèmes :

- Un peu compliqué
- Besoin d'autant d'étiquettes que d'appels du sous-programme.

## Solution : call et ret

- L'instruction **call** effectue un saut inconditionnel vers un sous-programme **après** avoir empilé l'adresse de l'instruction suivante
 

|     |             |          |             |          |
|-----|-------------|----------|-------------|----------|
| 11: | <b>call</b> | fonction | <b>push</b> | 12       |
| 12: | ...         |          | <b>jmp</b>  | fonction |
- L'instruction **ret** dépile une adresse et saute à cette adresse :
 

|            |            |             |
|------------|------------|-------------|
| <b>ret</b> | <b>pop</b> | <i>reg.</i> |
|            | <b>jmp</b> | <i>reg.</i> |
- Lors de l'utilisation de ces instructions, il est très important de gérer la pile correctement (**dépiler tout ce qu'on a empilé**) afin que l'adresse dépiler par l'instruction **ret** soit correcte.
- Permet d'imbriquer des appels de sous-programmes facilement.

# Exemple

```
%include "asm_io.inc"

SECTION .data
msg1: db "entier <10 ?",10,0
msg2: db "bravo",10,0
msg3: db "perdu",10,0

SECTION .text
 global main

main: mov eax, msg1
 call print_string
 call read_int
 call sb
 call print_string
 fin: mov ebx, 0
 mov eax, 1
 int 0x80
 sb: cmp eax, 10
 jl ok
 rate: mov eax, msg3
 ret
 ok: mov eax, msg2
 ret
```

# Passage des paramètres

!! Il est très important de **dépiler toute donnée qui a été empilée** dans le corps du sous-programme.

Exemple :

```
plus2: add eax, 2
 push eax
 ret ; dépile la valeur de eax !!
```

Ce code ne reviendra pas correctement !

---

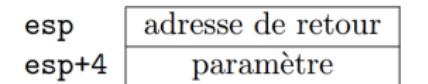
Autre problème : **passage des paramètres par registres**

- limite le nombre de paramètres ;
- mobilise les registres pour les conserver.

Solution : passer les paramètres par la pile (convention de C).

## Passage des paramètres

- Les paramètres passés par la pile sont empilés **avant** le **call**.
  - Si le paramètre doit être **modifié** par le sous-programme, c'est son **adresse** qui doit être passée.
  - Les paramètres sur la pile ne sont pas dépilerés par le sous-programme mais accédés depuis la pile elle-même :
    - Sinon, comme ils sont empilés avant le **call**, l'adresse de retour devrait être dépilerée avant tout (puis ré-empilée ensuite).
    - Si les paramètres sont utilisés à plusieurs endroits :
      - ▷ ça évite de les conserver dans un registre ;
      - ▷ les laisser sur la pile permet de conserver une copie de la donnée en mémoire accessible à n'importe quel moment.
  - Lors d'un appel de sous-programme : la pile ressemble à



- Accès au paramètre par adressage indirect : [esp+4]

# Passage des paramètres

- Si la pile est également utilisée dans le sous-programme pour stocker des données, le nombre à ajouter à `esp` change.
- Par exemple, après un `push` la pile ressemble à :

|                    |                          |
|--------------------|--------------------------|
| <code>esp</code>   | donnée du sous-programme |
| <code>esp+4</code> | adresse de retour        |
| <code>esp+8</code> | paramètre                |

- Maintenant, le paramètre est en `esp+8`, et non plus en `esp+4`.
- `esp` pour faire référence aux paramètres ⇒ erreurs possibles.
- Pour résoudre ce problème, utiliser `ebp` (base de la pile).
- La convention de C est qu'un sous-programme commence par empiler la valeur de `ebp` puis affecte à `ebp` la valeur de `esp`.
- Permet à `esp` de changer sans modifier `ebp`.
- A la fin du programme, la valeur originale de `ebp` est restaurée.

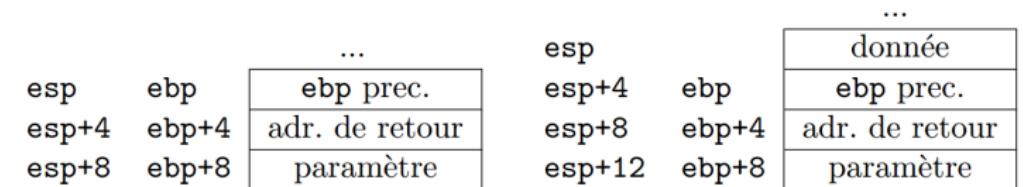
# Passage des paramètres

- Forme générale d'un sous-programme qui suit ces conventions :

`etiquette_sousprogramme:`

```
push ebp ; empile la valeur originale de ebp
 mov ebp, esp ; ebp = esp
; code du sous-programme
 pop ebp ; restaure l'ancienne valeur de ebp
 ret
```

- La pile au début du code du sous-programme ressemble à



- Le paramètre est accessible avec [ebp+8] depuis n'importe quel endroit du sous-programme.

# Passage des paramètres

- Une fois le sous-programme terminé, les **paramètres** qui ont été **empilés doivent être retirés**.
- La convention d'appel C spécifie que c'est au **code appelant** de le faire (convention différente en Pascal, par exemple).
- Un sous-programme utilisant cette convention peut être appelé comme suit :

```
push param ; passe un paramètre
call fnc
add esp, 4 ; retire le paramètre de la pile
```

- La 3ème ligne retire le paramètre de la pile en manipulant directement l'adresse du sommet de la pile.
- Une instruction pop pourrait également être utilisée, mais le paramètre n'a pas besoin d'être stocké dans un registre.

# Exemple

```
%include "asm_io.inc"

SECTION .data
msg1: db "entier <10 ?",10,0
msg2: db "bravo",10,0
msg3: db "perdu",10,0

SECTION .text
 global main
main: mov eax,msg1
 call print_string
 call read_int
 push eax
 call sb
 add esp, 4

 call print_string

fin: mov ebx,0
 mov eax,1
 int 0x80

sb: push ebp
 mov ebp, esp
 cmp dword [ebp+8], 10
 jl ok
rate: mov eax, msg3
 pop ebp
 ret

ok: mov eax, msg2
 pop ebp
 ret
```

# Template

## Architecture des ordinateurs

Wilfried  
Segretier  
wsegreti@univ-  
ag.fr

### Introduction

### Histoire de l'ordinateur et principes généraux

### Représentation des informations

### Circuits logiques

### Micro- architecture