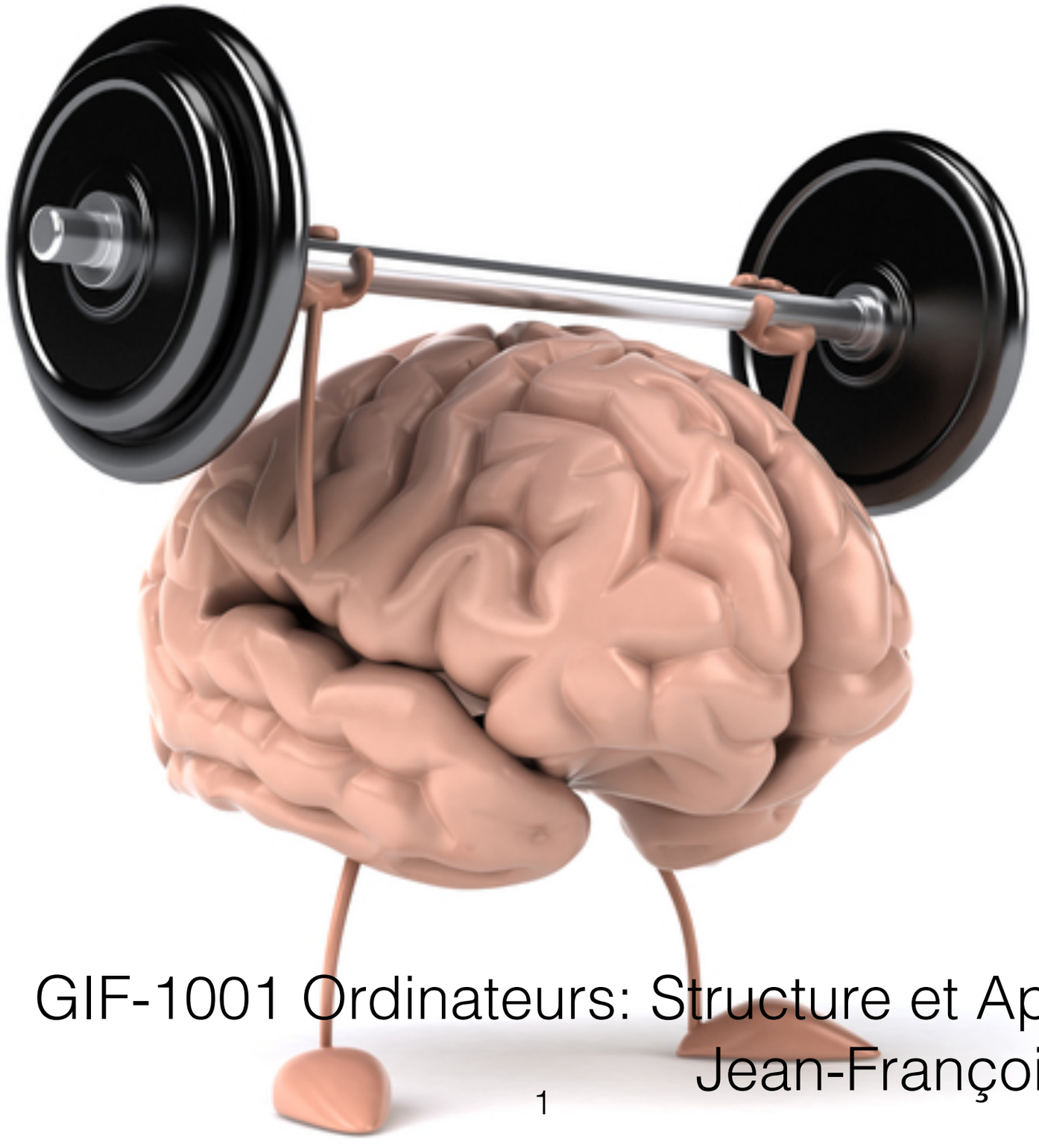
Révision mi-session



GIF-1001 Ordinateurs: Structure et Applications

Jean-François Lalonde

Stratégies *avant* l'examen

- Dans les jours précédant l'examen:
  - DORMIR
  - Étude:
    - devoirs, exercices faits en classe
    - examens des années précédentes—**ATTENTION**!
    - lecture: notes de cours, livre
    - objectifs du cours (sur le site web)
- Dans les heures précédant l'examen:
  - Alimentation adéquate (attention au «sugar crash»)
  - Économisez votre énergie intellectuelle
  - Détente

Stratégies *durant* l'examen

- *Avant* de commencer:
  - Compter les pages
  - Arracher la feuille des annexes pour un accès facile
  - Survoler l'examen et classer les questions selon leur difficulté/temps
- *Après* avoir commencé:
  - Commencer par les questions les plus faciles!
  - Bloqué? Sautez la question et prenez note d'y revenir
  - Ne pas oublier de respirer

Après l'examen…

Format des nombres

- 4 Points Hyper Importants à Retenir™

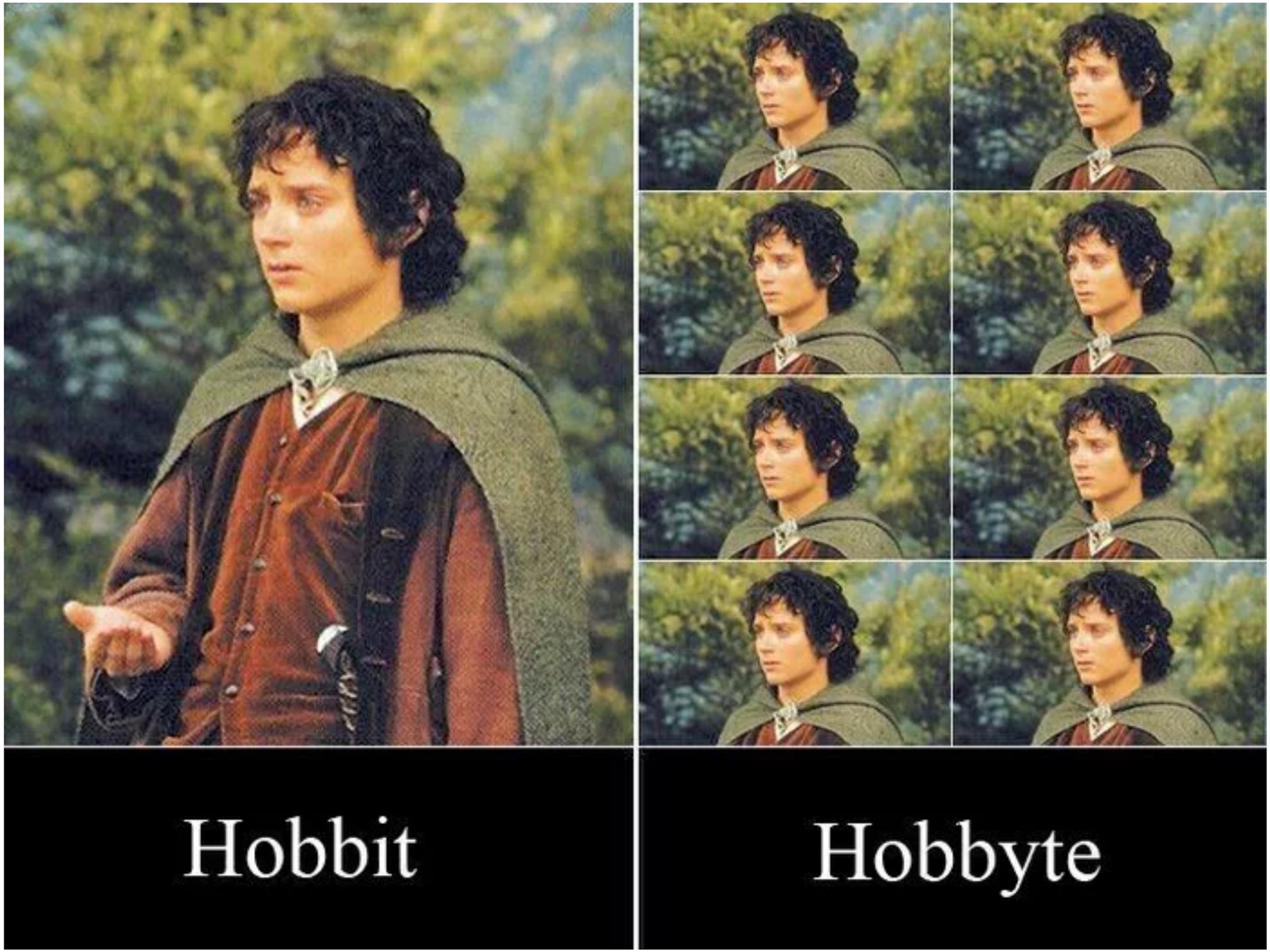
Raccourci	Explication
tout en binaire	Dans un ordinateur, tout, absolument tout, est stocké en format binaire.
# de bits prédéterminé	On utilise un nombre fini et pré-déterminé de bits pour représenter de l'information.
besoin d'une recette	À priori, nous ne pouvons savoir ce qu'une chaîne binaire signifie, il nous faut une «recette».
hexadécimal = binaire	L'hexadécimal est une façon plus compacte de représenter du binaire.

- Exemples de « recettes »:
  - complément-2 (nombres entiers signés)
  - IEEE 754 (nombres rationnels)
  - ASCII (chaîne de caractères)

Questions

- En complément-2 sur 4 bits:
  - quelles sont les valeurs minimales et maximales pouvant être représentées?
    - -8 à 7
  - 6+3=?
    - 0110 + 0011 = 1001, donc -7. Les bits de signe sont différents, donc il y a débordement.
- Exprimez la chaîne de bits suivante en hexadécimal: 01101001
  - 0x69
- Exprimez la chaîne de bits suivante en binaire: 0xA3
  - 10100011

Bits vs octets (bytes)



Mémoire

- On décrit une mémoire grâce à deux informations (indépendantes):
  - le **nombre d'adresses** possibles
    - ici: $2^8 = 256$ adresses
  - la **taille des mots** de la mémoire
    - ici: 8 bits (1 octet)

Adresses	Données							
	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0x00	0	1	0	1	1	0	1	1
0x01	1	1	1	0	1	0	0	1
0x02	0	1	0	0	1	0	0	1
0x03	0	1	0	0	1	1	0	1
0x04	0	1	1	1	1	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
0xFF	1	1	0	1	0	0	0	1

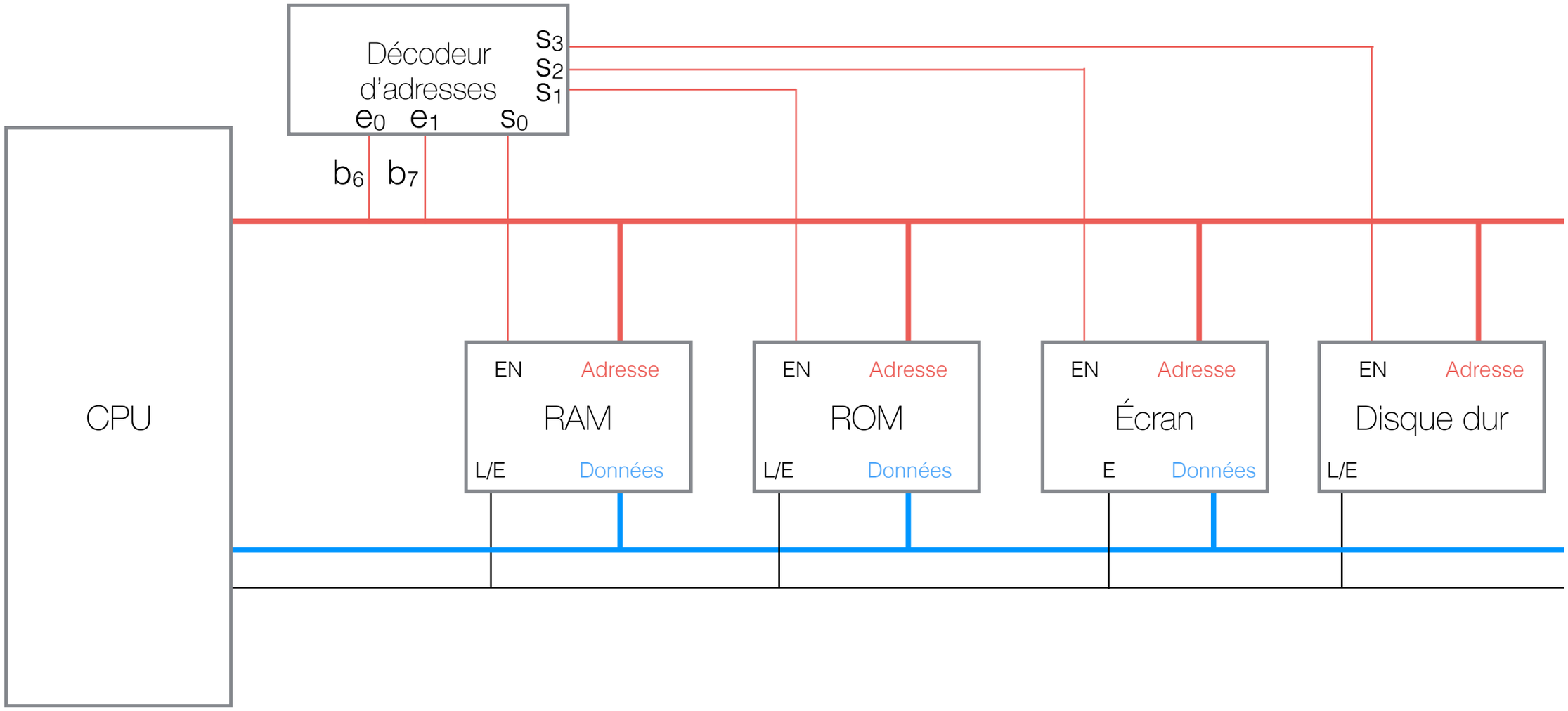
- La taille totale de la mémoire représente la quantité totale de bits pouvant être stockée dans la mémoire.
  - Elle est calculée de cette façon:

$$\text{taille mémoire} = \text{nombre d'adresses} \times \text{taille d'un mot}$$
$$= 2^8 \times 1 \text{ octet} = 256 \text{ octets}$$

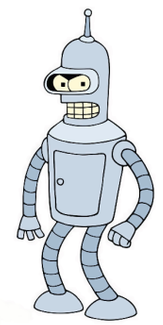
Structure interne

- Composantes principales: ALU, CCU, mémoires
- Cycle d’instructions?
  - fetch, decode, execute!
- Bus
  - Quels sont les 3 différents types?
  - À quoi sert un décodeur d’adresses?

Bus, décodeur



Adressage

- En « memory-mapped », on adresse tous les périphériques de la même façon
  - on utilise une **adresse** (ex: TP1, ARM)
- En « port-mapped », on adresse les périphériques différemment de la mémoire
  - on utilise une **instruction** spécialisée (ex: )

Questions

- Combien de mots en mémoire puis-je adresser si le bus de données et d'adresse ont tous deux 32 bits?
  - 2^{32}
- Un système « memory-mapped » possède un bus d'adresse de 16 bits. Les 4 bits les plus significatifs (MSB) sont utilisés pour le décodeur d'adresse. Combien de périphériques différents puis-je adresser?
  - $2^4 = 16$
- Ce même système possède une mémoire RAM. Si cette mémoire stocke des mots de 8 bits, quelle est la taille maximale de cette mémoire, en kilo-octets?
  - 16-4 = 12 bits d'adresses parviennent à la mémoire
  - 2^{12} x 8 bits = 2^{12} x 1 octet = 2^{12} octets = 2^2 x 2^{10} octets = 4 Ko

Intro à l’assembleur

- MOV vs LDR/STR
  - MOV = registres
  - LDR/STR = mémoire
- Instruction = binaire
  - Une autre recette!
  - Opcode, arguments

Questions

- Quel est l'équivalent binaire de ce programme? Écrivez votre réponse en hexadécimal.

Adresse	Instruction
0x0	MOV R0, #0x71
0x1	LDR R1, [R0]
0x2	JZE R1, #0x5
0x3	SUB R1, #0x1
0x4	MOV PC, #0x2
0x5	STR R1, [R0]

Jeu d'instructions, PC = 0xF

Mnémonique	**Opcode**	**Description**
MOV Rd, Rs	0000	Écriture de la valeur du registre Rs dans le registre Rd
MOV Rd, Const	0100	Écriture d'une constante dans le registre Rd
ADD Rd, Rs	0001	Addition des valeurs des registres Rd et Rs et insertion du résultat dans le registre Rd
ADD Rd, Const	0101	Addition de la valeur du registre Rd avec une constante et insertion du résultat dans Rd
SUB Rd, Rs	0010	Soustraction de la valeur Rs à l'intérieur de registre Rd.
SUB Rd, Const	0110	Soustraction d'une constante à l'intérieur du registre Rd
LDR Rd, [Rs]	1000	Chargement d'une valeur se trouvant à l'adresse Rs de l'ordinateur dans un registre.
STR Rd, [Rs]	1001	Écriture de la valeur d'un registre à l'adresse Rs de l'ordinateur.
JZE Rc, Const	1111	Saut à l'instruction située à l'adresse identifiée par la constante, mais seulement si Rc = 0 (sinon, cette instruction n'a aucun effet).
JZE Rc, Rs	1011	Saut à l'instruction située à l'adresse Rs seulement si Rc = 0 (sinon, cette instruction n'a aucun effet).

Format des instructions

Opcode	Argument 1	Argument 2
4 bits	4 bits	8 bits

```
0x4071
0x8100
0xF105
0x6101
0x4F02
0x9100
```

Questions

- Qu'est-ce que ce programme fait?

Adresse	Instruction
0x0	MOV R0, #0x71
0x1	LDR R1, [R0]
0x2	JZE R1, #0x5
0x3	SUB R1, #0x1
0x4	MOV PC, #0x2
0x5	STR R1, [R0]

Jeu d'instructions

Mnémonique	**Opcode**	**Description**
MOV Rd, Rs	0000	Écriture de la valeur du registre Rs dans le registre Rd
MOV Rd, Const	0100	Écriture d'une constante dans le registre Rd
ADD Rd, Rs	0001	Addition des valeurs des registres Rd et Rs et insertion du résultat dans le registre Rd
ADD Rd, Const	0101	Addition de la valeur du registre Rd avec une constante et insertion du résultat dans Rd
SUB Rd, Rs	0010	Soustraction de la valeur Rs à l'intérieur de registre Rd.
SUB Rd, Const	0110	Soustraction d'une constante à l'intérieur du registre Rd
LDR Rd, [Rs]	1000	Chargement d'une valeur se trouvant à l'adresse Rs de l'ordinateur dans un registre.
STR Rd, [Rs]	1001	Écriture de la valeur d'un registre à l'adresse Rs de l'ordinateur.
JZE Rc, Const	1111	Saut à l'instruction située à l'adresse identifiée par la constante, mais seulement si Rc = 0 (sinon, cette instruction n'a aucun effet).
JZE Rc, Rs	1011	Saut à l'instruction située à l'adresse Rs seulement si Rc = 0 (sinon, cette instruction n'a aucun effet).

- Ce programme:
  - lit la valeur à l'adresse 0x71
  - il la décrémente jusqu'à temps qu'elle atteigne la valeur de 0
  - il stocke le résultat (0) à la même adresse

ARM

- Connaître les éléments de base de l'architecture ARM
  - Taille des instructions?
  - Nombre de registres?
  - RISC ou CISC?
  - Structure de la mémoire et impact sur PC
    - à chaque instruction, PC = PC + 4
    - PC pointe à l'adresse de l'instruction exécutée **+ 8** (donc 2 instructions plus loin)

Assembleur ARM

- Être capable de **comprendre** des programmes simples
  - Comprendre le fonctionnement des instructions et des codes de condition
- Être capable **d'écrire** des programmes simples
  - E.g. valeur absolue, "if/ else", appel de fonction simple

Instruction	Description
`ADD Rd, Rs, Op1`	`Rd` ← `Rs + Op1`
`AND Rd, Rs, Op1`	`Rd` ← `Rs AND Op1`
`ASR Rd, Rs, #cte`	`Rd` ← `Rs` / $2^{\texttt{cte}}$
`B etiquette`	`PC` ← `adresse(etiquette)`
`BL etiquette`	`LR` ← `PC - 4`, `PC` ← `adresse(etiquette)`
`BX Rs`	`PC` ← `Rs`
`CMP Rs, Op1`	Change les drapeaux comme `Rs - Op1`
`LDR Rd, etiquette`	`Rd` ← `valeur(etiquette)`
`LDR Rd, =etiquette`	`Rd` ← `adresse(etiquette)`
`LDR Rd, [Rb, Op1]`	`Rd` ← `Mem[Rb + Op1]`
`LDR Rd, [Rb], Op1`	`Rd` ← `Mem[Rb]`, `Rb` ← `Rb + Op1`
`LDR Rd, [Rb, Op1]!`	`Rb` ← `Rb + Op1`, `Rd` ← `Mem[Rb]`
`LSL Rd, Rb, #cte`	`Rd` ← `Rb` × $2^{\texttt{cte}}$
`MUL Rd, Rn, Rs`	`Rd` ← `Rn` × `Rs`
`MVN Rd, Op1`	`Rd` ← `!Op1` (inverse les bits)
`POP {Liste Reg}`	Charge les registres en ordre croissant à partir de la pile, `SP` ← `SP - 4` × `(nombre de registres)`
`PUSH {Liste Reg}`	`SP` ← `SP + 4` × `(nombre de registres)`, Met la liste de registres sur la pile dans l'ordre décroissant
`STR Rs, etiquette`	`valeur(etiquette)` ← `Rd`
`STR Rs, [Rb, Op1]`	`Mem[Rb + Op1]` ← `Rs`
`STR Rs, [Rb], Op1`	`Mem[Rb]` ← `Rs`, `Rb` ← `Rb + Op1`
`STR Rs, [Rb, Op1]!`	`Rb` ← `Rb + Op1`, `Mem[Rb]` ← `Rs`
`SUB Rd, Rs, Op1`	`Rd` ← `Rs - Op1`

Code	Condition	Code	Condition
`CS`	Retenue (carry)	`CC`	Pas de retenue
`EQ`	Égalité	`NE`	Inégalité
`VS`	Débordement	`VC`	Pas de débordement
`GT`	Plus grand	`LT`	Plus petit
`GE`	Plus grand ou égal	`LE`	Plus petit ou égal
`PL`	Positif	`MI`	Négatif

Assembleur ARM

Écrivez un programme qui compare les deux éléments du tableau `monTableau`, et qui écrit la valeur maximale dans `monResultat`.

```
SECTION INTVEC

B main

SECTION CODE

main

; écrivez votre code ici

B main

SECTION DATA

monTableau ASSIGN32 0x123, 0x456
monResultat ALLOC32 1
```

Assembleur ARM — solution

```
SECTION INTVEC

B main

SECTION CODE

main

LDR R0, =monTableau
LDR R5, =monResultat

; chargeons les éléments du tableau dans R1 et R2
LDR R1, [R0], #4
LDR R2, [R0]

; comparons et stockons la valeur maximale
CMP R1, R2
STRGT R1, [R5] ; si R1 > R2
STRLE R2, [R5] ; sinon (R2 >= R1)

B main

SECTION DATA

monTableau ASSIGN32 0x123, 0x456
monResultat ALLOC32 1
```

Assembleur ARM

Décrivez, *en une seule phrase*, ce que fait la fonction `fonctionMystere`.

```
fonction
; R0 contient le premier paramètre
; R1 contient le deuxième paramètre
PUSH {R1,R2}

CMP R1, #0
BEQ un

MOV R2, R0

boucle

SUBS R1, R1, #1
BEQ fin

MUL R0, R0, R2
B boucle

un
MOV R0, #1

fin
POP {R1,R2}

; le résultat est stocké dans R0
BX LR
```

Assembleur ARM

Décrivez, *en une seule phrase*, ce que fait la fonction `fonctionMystere`.

```
fonctionMystere
; R0 contient le premier paramètre
; R1 contient le deuxième paramètre
PUSH {R1,R2}

CMP R1, #0
BEQ un

MOV R2, R0

boucle

SUBS R1, R1, #1
BEQ fin

MUL R0, R0, R2
B boucle

un
MOV R0, #1

fin
POP {R1,R2}

; le résultat est stocké dans R0
BX LR
```

Réponse: il calcule R0^R1 (exponentielle)