

# Incrément 1

17 septembre 2018

- Objectif de l'UE

# Introduction

## Assembleur MIPS

- Supports de travail
  - Le sujet du projet
  - Le site web internet
    - Des supports de cours
    - Le sujet du projet
    - Les objectifs de chaque incrément
    - Des informations sur les outils utiles
    - Les fiches pédagogiques
    - Un code de base pour démarrer (step 0)
  - Documentation MIPS
  - Alternance BE 2h // TP 4h // travail personnel // rendu pour chaque incrément

# Introduction

- Notation
  - Pour chaque incrément (4)
    - Code + Makefile + Readme
    - Rapport
  - Examen final (1h sur machine)
  - Une note d'observation par le tuteur

# Projet

## MIPS

Qu'est ce que c'est ?

# Projet

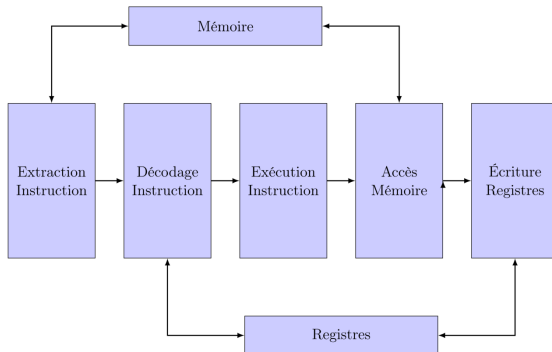
## Réaliser un assembleur pour le microprocesseur MIPS

- Langage de programmation
- Processus de transformation en binaire

# MIPS

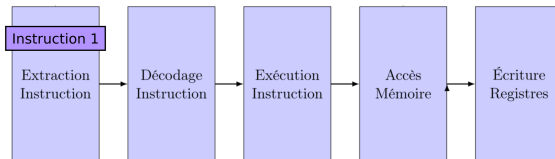
## Le pipeline

- Le pipeline
  - Pipeline
  - Instructions à taille constante de 32 bits (4 octets)
  - un cycle d'horloge pour chaque instruction



# MIPS

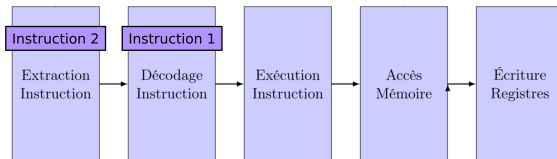
## Fonctionnement du pipeline





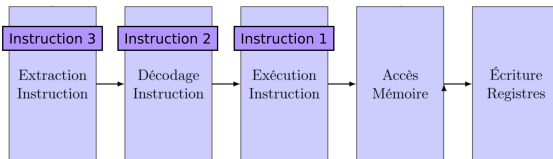
# MIPS

## Fonctionnement du pipeline



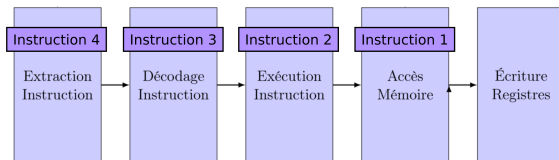
# MIPS

## Fonctionnement du pipeline



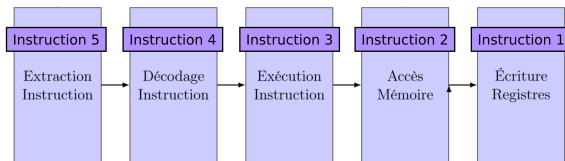
# MIPS

## Fonctionnement du pipeline



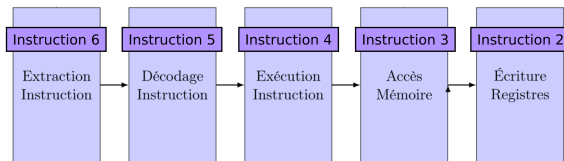
# MIPS

## Fonctionnement du pipeline



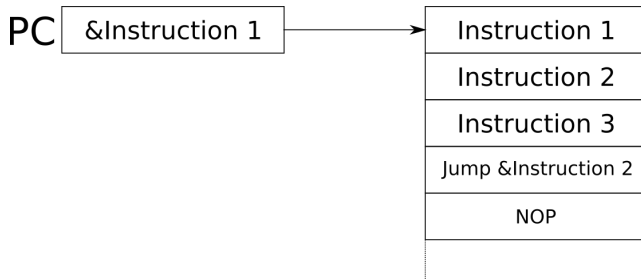
# MIPS

## Fonctionnement du pipeline



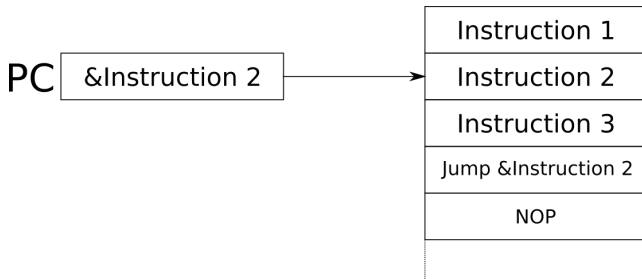
# MIPS

## Compteur programme



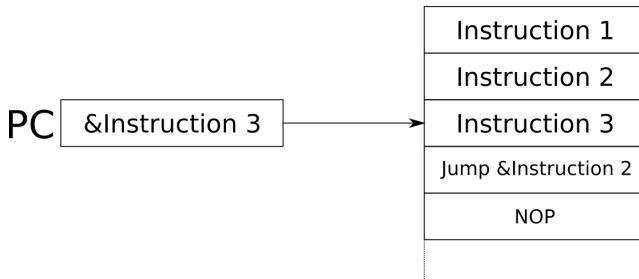
# MIPS

## Compteur programme



# MIPS

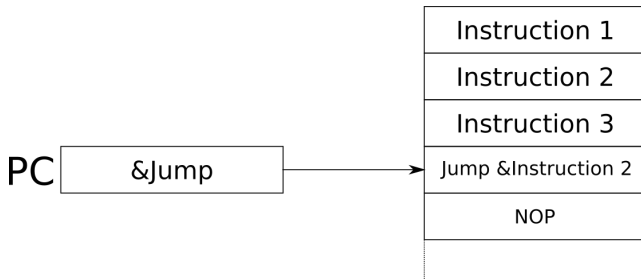
## Compteur programme





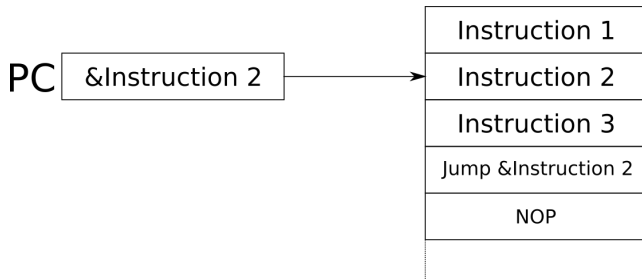
# MIPS

## Compteur programme



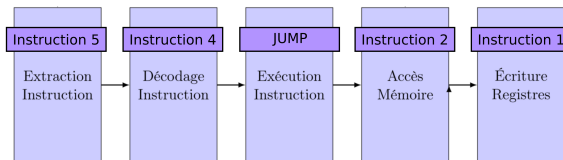
# MIPS

## Compteur programme



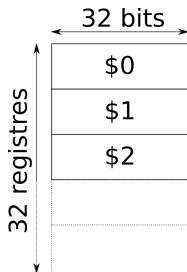
# MIPS

## Compteur programme



# MIPS

## Registres



Mnémonique	Registre	Usage
\$zero	\$0	Registre toujours nul, même après une écriture
\$at	\$1	<i>Assembler temporary</i> : registre réservé à l'assembleur
\$v0, \$v1	\$2, \$3	Valeurs retournées par une sous-routine
\$a0-\$a3	\$4-\$7	Arguments d'une sous-routine
\$t0-\$t7	\$8-\$15	Registres temporaires
\$s0-\$s7	\$16-\$23	Registres temporaires, préservés par les sous-routines
\$t8, \$t9	\$24, \$25	Deux temporaires de plus
\$k0, \$k1	\$26, \$27	kernel (réservés!)
\$gp	\$28	Global pointer (on évite d'y toucher!)
\$sp	\$29	<i>Stack pointer</i> : pointeur de pile
\$fp	\$30	Frame pointer (on évite d'y toucher!)
\$ra	\$31	<i>Return address</i> : utilisé par certains instructions (JAL) pour sauver l'adresse de retour d'un saut

# Instructions

## Représentation hexadécimale

L'hexadécimale, à quoi ça sert ?? Comment on l'écrit ??

# MIPS

## Mémoire

- 4Go de mémoire, adressable par octet
- une adresse est codée sur un entier non signé de 4 octets, 32 bits
- encodage des valeurs sur plusieurs octets (e.g. : un entier signé 32 bit) en Big Endian
- la mémoire d'un processus est segmenté en sections :
  - section .text
  - section .data
  - section .bss
  - (pile .stack)

# MIPS

## Exemple de binaire section .text

- Supposons que l'instruction encodée en hexa 0x00641020 se présente sur le micro-processeur
- En binaire : 0000 0000 0110 0100 0001 0000 0010 0000
- Que fait le processeur ? Cf sujet page 7

# MIPS

## Exemple de binaire section .text

- Supposons que l'instruction encodée en hexa 0x00641020 se présente sur le micro-processeur
- En binaire : 0000 0000 0110 0100 0001 0000 0010 0000
- Que fait le processeur ? Cf sujet page 7
- ADD \$2, \$3, \$4
- donc le processeur va additionner les valeurs contenues dans les registres 3 et 4, et placer le résultat dans le registre 2.



# Langage assembleur : exemple

```
# allons au ru
.set noreorder
.text
    Lw $t0 , lunchtime
    LW $6, -200($7)
    ADDI $t1,$zero,8
boucle:
    BEQ $t0 , $t1 , byebye
    NOP
    addi $t1 , $t1 , 1
    J boucle
    NOP
byebye:
    JAL viteviteauru

.data
lunchtime:
    .word 12
    .word menu
    .asciiz "ils_disent:_\"au_ru!\""
.bss
menu:
    .space 24
```

# Langage assembleur

## Directives

Plusieurs catégories :

- Sectionnement de programme
- Définition de données

Directive	Description
<code>.text</code>	Ce qui suit doit aller dans le segment TEXT
<code>.data</code>	Ce qui suit doit aller dans le segment DATA
<code>.bss</code>	Ce qui suit doit aller dans le segment BSS
<code>.set option</code>	Instruction à l'assembleur pour inhiber ou non certaine options. Dans notre cas seule l'option <i>noreorder</i> est considérée
<code>.word w1, ..., wn</code>	Met les <i>n</i> valeurs sur 32 bits dans des mots successifs (ils doivent être alignés!)
<code>.byte b1, ..., bn</code>	Met les <i>n</i> valeurs sur 8 bits dans des octets successifs
<code>.asciiz s1, ..., sn</code>	Met les <i>n</i> chaînes de caractères à la suite en mémoire. Chaque chaîne est terminée par <code>\0</code> .
<code>.space n</code>	Réserve <i>n</i> octets en mémoire. Les octets sont initialisés à zéro.

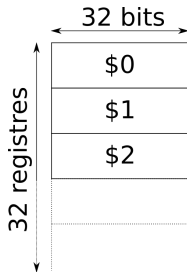
# Langage assembleur

## Valeurs

- entiers signés ou non signés
  - codés en décimal : 5 -6 987
  - ou en hexa : 0xFFFF 0xA1
  - signés ou non
- les chaînes de caractères, entre quote : "je suis une chaine"

# Langage assembleur

## Registres



Mnémonique	Registre	Usage
\$zero	\$0	Registre toujours nul, même après une écriture
\$at	\$1	<i>Assembler temporary</i> : registre réservé à l'assembleur
\$v0, \$v1	\$2, \$3	Valeurs retournées par une sous-routine
\$a0-\$a3	\$4-\$7	Arguments d'une sous-routine
\$t0-\$t7	\$8-\$15	Registres temporaires
\$s0-\$s7	\$16-\$23	Registres temporaires, préservés par les sous-routines
\$t8, \$t9	\$24, \$25	Deux temporaires de plus
\$k0, \$k1	\$26, \$27	kernel (réservés!)
\$gp	\$28	Global pointer (on évite d'y toucher!)
\$sp	\$29	<i>Stack pointer</i> : pointeur de pile
\$fp	\$30	Frame pointer (on évite d'y toucher!)
\$ra	\$31	<i>Return address</i> : utilisé par certains instructions (JAL) pour sauver l'adresse de retour d'un saut

# Langage assembleur

## Étiquette

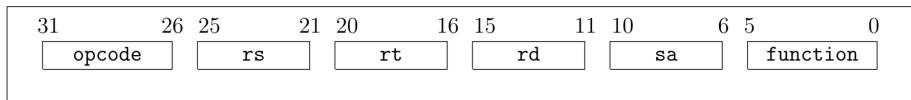
- Désignation symbolique d'une adresse en mémoire
- Valeur de l'étiquette : cette adresse ("depuis le début de la section où elle est déclarée").
- Déclaration : Nom, suivi de ' : ' . Exemple : etiquette :
- N'est déclarée qu'une seule fois dans le programme
- Nom := caractères alphanumériques et underscore \_ : a b 8 \_ etc.
- Un nom d'étiquette commence exclusivement par une lettre ou par un underscore \_
- Valeur d'une étiquette ? Exemple :
  - lunchtime vaut 0x0 (depuis debut section data)
  - boucle vaut 0x0C ... Oups ?
  - viteviteauru : valeur inconnue .... Oups ?

# Langage assembleur

## Instructions

3 types d'instructions :

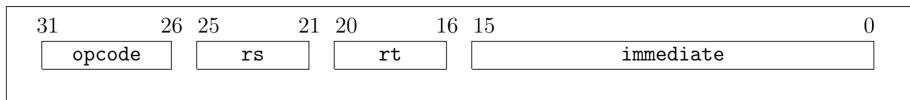
- Type R
- Exemple : ADD rd, rs, rt



## Instructions

3 types d'instructions :

- Type I
- Exemple : `ADDI rt,rs, immediate`

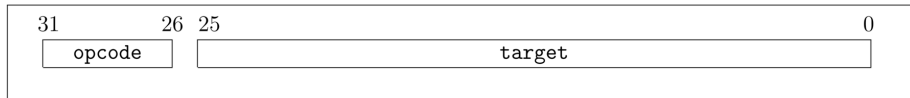


# Langage assembleur

## Instructions

3 types d'instructions :

- Type J
- Exemple : J target





# Instructions

## Exemple d'assemblage d'une instruction

- Soit dans le code assembleur la ligne :
- `ADD $2, $3, $4`
- Après assemblage, que devient cette instruction dans le code binaire généré par votre programme ?
- Cf sujet p7 ou p59

# Instructions

## Exemple d'assemblage d'une instruction

- Soit dans le code assembleur la ligne :
- `ADD $2, $3, $4`
- Après assemblage, que devient cette instruction dans le code binaire généré ?
- Cf sujet p7 ou p59
- `0000 0000 0110 0100 0001 0000 0010 0000`
- ou en hexa : `0x00641020`

# Instructions


## Exemple d'assemblage d'une instruction

Lw \$6, - 200(\$7)

# Instructions

## Exemple d'assemblage d'une instruction

**Lw** \$6, -200(\$7)  
1000 11

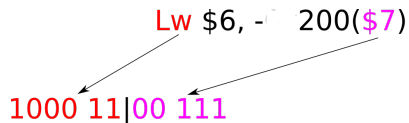


# Instructions

## Exemple d'assemblage d'une instruction

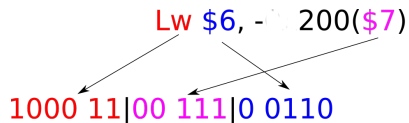
Lw \$6, -200(\$7)

1000 11|00 111



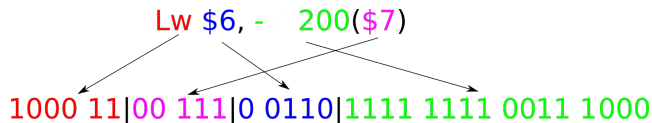
# Instructions

## Exemple d'assemblage d'une instruction



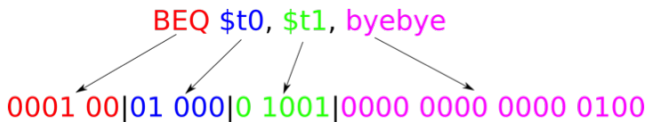
# Instructions

## Exemple d'assemblage d'une instruction



# Instructions

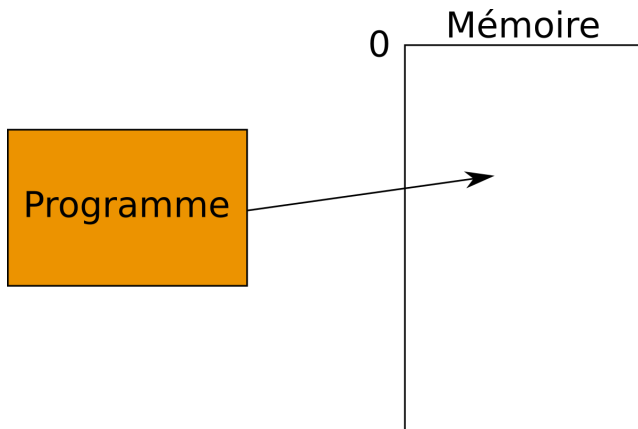
## Exemple d'assemblage d'une instruction



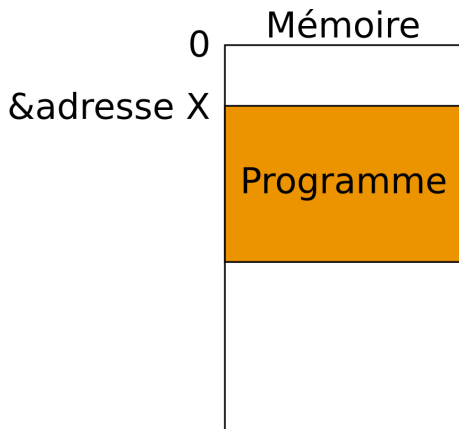
Oups ? Pourquoi 0100 (soit 4 en décimal) ?



# Chargement dynamique en mémoire

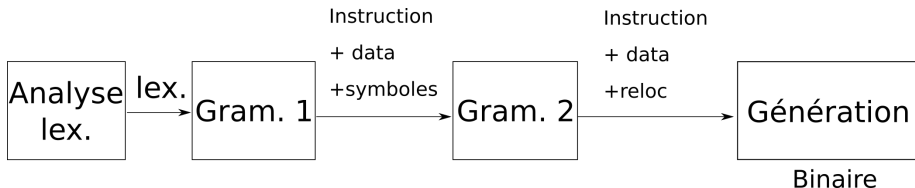


# Chargement dynamique en mémoire

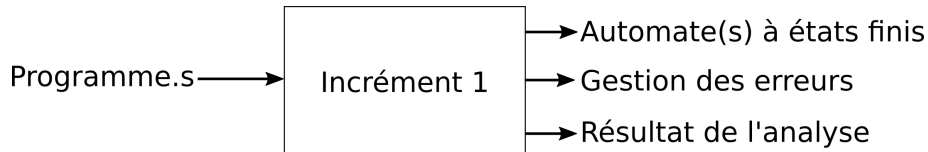


# Le projet

Division du projet en 4 étapes :



# Incrément 1 : analyse lexicale



# Incrément 1 : analyse lexicale

- Analyse lexicales detection de groupe de caractères formant un lexème
- Analyse des lexèmes, détection d 'erreurs
- ENTREE : le code assembleur
- SORTIE : série des lexèmes (typés), dans le Terminal

```
# Un commentaire...
eti:  .byte - 4
      ADD $2,$3,$4
      J  0xABCD
```

[COMMENT	]	Un commentaire...
[NL	]	\n
[SYMBOLE	]	eti
[DEUX_PTS	]	:
[DIRECTIVE	]	.byte
[VAL_DECIMAL	]	-4
[NL	]	\n
[SYMBOLE	]	ADD
[REGISTRE	]	\$2
[VIRGULE	]	,
[REGISTRE	]	\$3
[VIRGULE	]	,
[REGISTRE	]	\$4
[NL	]	\n
[SYMBOLE	]	J
[VAL_HEX	]	0xABCD

# Incrément 1 : analyse lexicale

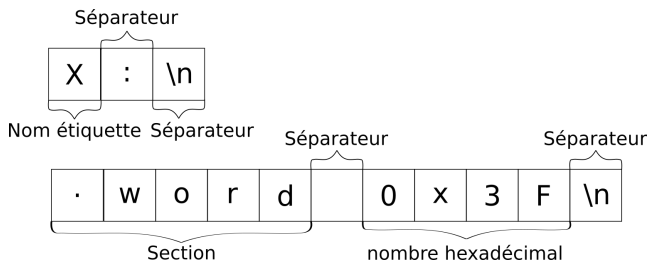
## Exemple

X	:	\n
---	---	----

.	w	o	r	d		0	x	3	F	\n
---	---	---	---	---	--	---	---	---	---	----

# Incrément 1 : analyse lexicale

## Exemple



# Incrément 1 : analyse lexicale

Quelle structure de données ?



# Incrément 1 : analyse lexicale

Quelle structure de données ?

Quelque chose comme....

```
1
2 // STRUCTURE DE DONNEE POUR LEXEME
3
4 struct lex {
5     char * lex_value ;
6     int type ;      // ou type énuméré ?
7     int ligne ; // pour debug //// ou unsigned ?
8     int valeur ; ou autre ??? ou pas ???
9 } ;
10 typedef struct lex lex_t ;
11
12 // => un module lex.h lex.c ?
13
14 // PUIS :
15 // Liste de lexème
16 // OU MIEUX :
17 // File (FIFO, queue) de lexèmes
18
19 // => un module file_lexeme.h file_lexeme.c ?
20
```

==> allocation dynamique des chaînes. Usage de strdup() ?

# Incrément 1 : analyse lexicale

Quelles peuvent être les catégories (types) de lexèmes dans le cadre de l'assembleur MIPS ? Comment les représenter dans le code ?

# Analyse lexicale

Comment faire pour coder l'analyseur lexical (et donc construire la liste ou file de lex\_t en mémoire) ?

```
# allons au ru
.set noreorder
.text
    Lw $t0 , lunchtime
    LW $6, -200($7)
    ADDI $t1,$zero,8
boucle:
    BEQ $t0 , $t1 , byebye
    NOP
    addi $t1 , $t1 , 1
    J boucle
    NOP
byebye:
    JAL viteviteauru

.data
lunchtime:
    .word 12
    .word menu
    .asciiz "ils_disent:_\"au_ru!\""
.bss
menu:
    .space 24
```

# Automate à Etats finis

# Automate à Etats finis

Comment l'implanter ?

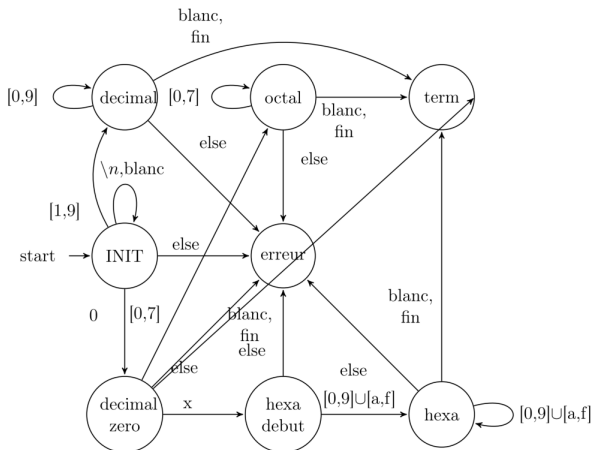
# Automate à Etats finis

## Comment l'implanter ?

- Analyse caractère par caractère
- Une variable pour encoder l'état courant de l'automate ("on est en train de lire ca")
- Analyse du caractère courant en fonction de l'état courant
  - Dont usage des fonctions prédicat isalpha(), isdigit() et consoeurs.
  - Voir : man isalpha, etc.
- Transition vers le nouvel etat *ou erreur*

# Automate à Etats finis

## Exemple



Cf exemple de codage p34

# Automate à Etats finis

Oups !

- `200($at)` vs `200 ( $at )`
- `0xcafe,0xdada` vs `0xcafe , 0xdada`
- `etiquette :.word 0xFFFFFFFF#com`  
vs `etiquette : .word 0xFFFFFFFF #com`
- Délimiteurs (notamment pour les nombres) :  
espaces (space, tabulation, retour chariot) ainsi que `,` `( )` `.` `-` `+` `#` :



# Automate à Etats finis

Oups! les chaines

- "ceci n'est pas un commentaire #coucou"
- "il dit \"vite au ru\" \t! miam! \n"
- "un backslash dans une chaine? \\"
- char spéciaux : \" \\  
• mais encore... \n \t \r \f \b \a \' \?

# Automate à Etats finis

## Gestion des erreurs ?

- des retours à l'utilisateur SIGNIFIANTS, écrits dans stderr
- puis arrêt du programme. Après une ou plusieurs erreurs ?
- centralisation des messages d'erreurs ?

```
1
2 typedef enum {
3     LEX_ERR_HEX,
4     LEX_ERR_ETIQ, ...
5 } lex_error_t;
6
7 char * let_error_str [] = {
8     "Format hexadecimal erroné",
9     "étiquette invalide"
10    ...
11 } ;
12
```

# Développement piloté par les tests

- Commencer par ECRIRE d'abord DES JEUX DE TEST (programmes .s)
- Ainsi que la façon dont doit réagir votre programme à chaque test
- Dont des tests sensés mettre le programme EN ERREUR pour vérifier que ces erreurs sont bien gérées
- Dont des tests qui poussent votre programme dans ses limites (très longues chaines, très longs nom d'étiquette... que sais-je!)
- J'évaluerai la qualité et la complétude de vos jeux de test

# Le code fourni

- Voir le site du module
- Vous devez modifier le prototype de la fonction `getNextToken()`