

UPSimulator

Maxence KLEIN

Véronique REYNAUD

Guillaume DESJOUIS

2020

Table des matières

1	Présentation générale du projet	3
1.1	Objectifs	3
1.2	Structure	3
2	Choix techniques	4
2.1	Langage jouet	4
2.2	Parsing	5
2.3	Code Assembleur	5
2.4	Modèle de processeur	5
2.5	Interface utilisateur	5
2.6	Gestion de la documentation	5
3	Organisation	5
3.1	Planification	5
3.2	Répartition des tâches	5

1 Présentation générale du projet

1.1 Objectifs

Le projet UPSIMULATOR a pour objectif de développer un simulateur de processeur à visée pédagogique. Celui-ci doit permettre d'appréhender la chaîne conduisant d'un programme écrit dans un langage de haut niveau au détail de l'exécution à l'échelle du processeur. Pour cela, le projet doit permettre :

- la production d'un code source dans un langage jouet ;
- la compilation du code source et la production d'une version assembleur et binaire de celui-ci. Le simulateur doit permettre l'usage de différents modèles (taille des mots binaires, nombre de registre, ...) ;
- le suivi de l'exécution (registres, mémoire, pointeur, appels à l'UAL,...) ;

Les choix techniques retenus pour chaque fonctionnalités sont développés ci-après.

1.2 Structure

diag UML ou équivalent

2 Choix techniques

2.1 Langage jouet

Le langage jouet doit permettre à l'utilisateur de produire un exemple de code simple reprenant les principales structures (boucles, branchements conditionnels,...)

```

1  s = 0
2  i = 0
3  m = input()
4  while i < m:
5      if i % 2 == 0:
6          s = s + i
7          if s < 10:
8              y = s * 2
9              print(y)
10         i = i + 3
11     print(s)

```

Listing 1 – Exemple de code dans le langage jouet

Expressions admissibles

Les expression admissibles sont présentées dans la table 1 ci-dessous.

TABLE 1 – Expressions admissibles

Variable		x
Entier		n
Opérations arithmétiques	Somme	e1 + e2
	Différence	e1 - e2
	Produit	e1 * e2
	Division entière	e1 / e2
	Reste	e1
	Opposé	-e1

Opérations logiques		
Binaires	Egalité	e1 == e2
	Différence	e1 != e2
	Inégalités	e1 < e2
		e1 > e2
		e1 <= e2
		e1 >= e2
	Et	e1 and e2
		e1 & e2
	Ou	e1 or e2
		e1 e2
Unaire	inverse bit à bit	~e1
	négation logique	not e1

Liste de commandes admissibles

Affectation

```
x=e
```

avec e une expression logique ou arithmétique.

Branchement conditionnel

```

if e :
    c1
elif e2:
    c2
else:
    c3

```

avec e1 et e2 des expressions et c1, c2 et c3 des commandes.
Les branchement `else` et `elif` sont optionnels.

Boucle

```
while e :  
    c1
```

avec e une expression et c une commande.

Lecture clavier

```
input()
```

Ecriture sur la sortie courante

```
print(v)
```

avec e une expression

Indentations

Le code est indenté comme en python afin de détecter les blocs :

- L'indentation n'augmente qu'après un : lié à une structure `if` ou `while`
- L'indentation ne peut diminuer que atteindre un niveau précédemment atteint.

Commentaires

Les commentaires sont repérés par le caractère `##` .

```
aaaaaaaaa:  
    aaaaaaaaa  
    aaaaaaaaa  
        aaaaaaaaa # pas valable il manque : avant  
    aaaaaaaaa  
    aaaaaaaaa  
aaaaaaaaa  
aaaaaaaaa:  
    aaaaaaaaa  
    aaaaaaaaa # pas valable. Ce niveau a été atteint avant  
    ↪ mais pas dans le même bloc  
aaaaaaaaa
```

Listing 2 – Langage jouet - Commentaires et indentations

2.2 Parsing

Une étape d'analyse du code (parsing) est nécessaire en amont de la production du code assembleur. Cette étape a pour objet :

- d'assurer que la syntaxe du langage jouet est respectée
- de permettre la construction d'un arbre représentant les différentes structures du code source afin de pouvoir produire le code assembleur et le binaire associé

2.3 Code Assembleur

2.4 Modèle de processeur

2.5 Interface utilisateur

2.6 Gestion de la documentation

3 Organisation

3.1 Planification

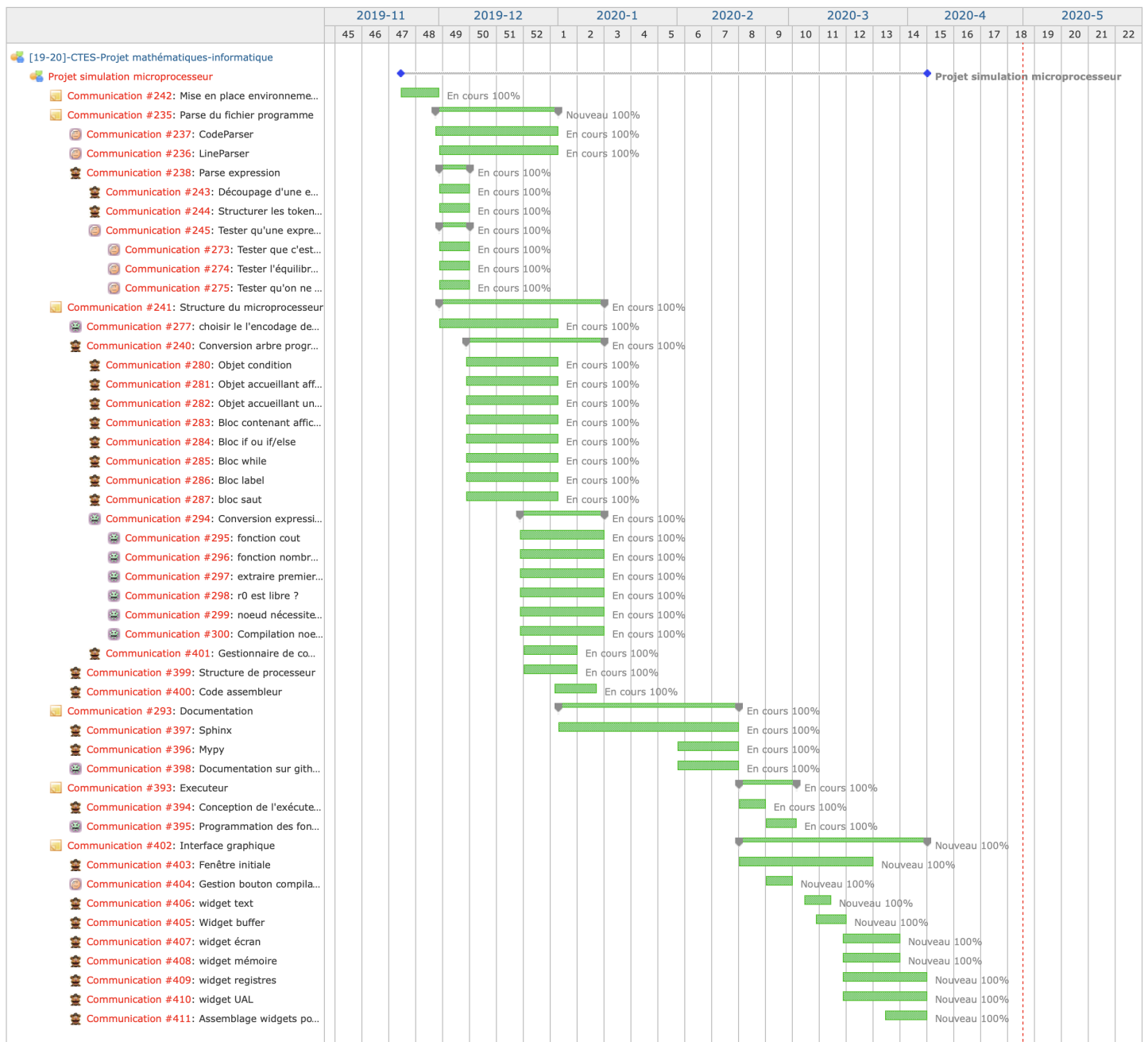


FIGURE 1 – Diagramme de Gantt du projet

3.2 Répartition des tâches

Table des figures

1	Diagramme de Gantt du projet	6
---	--	---

Liste des tableaux

1	Expressions admissibles	4
---	-----------------------------------	---