

PROJET DE PROGRAMMATION

CAHIER DES BESOINS

Environnement d'exécution y86+HCL



BANDET Alexis
GAISSET Valentin
GUISSET Romain
SIMBA Florian

Professeur :
Mme. ZANON-BOITO
Francieli

Janvier 2019

Table des matières

1	Introduction	2
2	Mise en contexte	2
3	État de l’art	4
3.1	Application web [2]	4
3.2	Application x86 (standalone)	4
4	Description des besoins	5
4.1	Besoins fonctionnels	6
4.2	Besoins non fonctionnels	11
4.3	Classement des besoins par ordre de priorité	12
4.4	Changements des besoins	12
4.5	L’interface utilisateur	12
5	Planification des tâches	15
6	Annexes	18
.1	Description de l’architecture	19

1 Introduction

Le projet **Environnement d'exécution y86+HCL** a pour but d'améliorer un simulateur déjà existant de ce dit environnement.

Le **y86** est une simplification de l'architecture x86 et a été créé à des fins pédagogiques, pour permettre à des étudiants d'appréhender l'architecture des processeurs en faisant abstraction de détails complexes.

Il a été créé par R. E. Bryant et D. R. O'Hallaron, deux professeurs de la *Carnegie Mellon University*.

Les outils [3] fournis par ces deux enseignants se font vieux et peu pratiques à utiliser, surtout pour des débutants qui ne sont pas forcément à l'aise avec la compilation en C. Pour palier à ce soucis, différentes applications web ont déjà été développées, plus ou moins complètes, plus ou moins modulaires, etc. De part sa conception il contient une double dépendance lors de la modification du jeu d'instruction, ce qui oblige la modification de plusieurs fichiers lors de l'ajout ou de la modification d'une instruction du processeur. Le processus de modification de *l'instruction set* est alors en parti dénaturé et se transforme en compréhension de la compilation du programme.

Nos clients, **Aurélien ESNARD** et **François PELLEGRINI** ont donc proposé ce projet dans le but d'améliorer l'environnement de travail des futurs étudiants. Plusieurs objectifs sont de mise, à savoir :

- Une **interface intuitive** et agréable.
- Gestion du **HCL** (**H**ardware **C**ontrol **L**anguage) afin de :
 - Modifier des instructions existantes
 - Créer de nouvelles instructions
- Avoir une vue détaillée de l'**état du processeur** (stages).
- L'application doit être exécutée sur plusieurs système d'exploitation.
- L'application doit être **accessible au plus grand nombre**, notamment aux personnes souffrant d'un handicap.
- L'application doit être **modulaire** et **documentée** afin de simplifier l'ajout de fonctionnalités ultérieurement. Cela doit notamment permettre l'implémentation de différentes versions du simulateur (*sequential, pipelined*).

2 Mise en contexte

La suite du document traitera d'éléments liés au y86. Afin d'en faciliter la compréhension, nous allons expliquer ici quelques une de ces notions.

1. Fichiers **.ys** :

Fichier source contenant le code y86 à compiler.

```

1
2 .pos 0
3 Init:
4     irmovl Stack, %ebp
5     irmovl Stack, %esp
6
7 .pos 0x100
8 Stack:
9

```

FIGURE 1 – Code d'un programme y86 (.ys)

2. Fichiers .yo :

Fichier objet contenant le code compilé à partir du .ys source.

0x0000:		.pos 0
0x0000:		Init:
0x0000:	30f500010000	irmovl Stack, %ebp
0x0006:	30f400010000	irmovl Stack, %esp
0x000c:		.pos 0x100
0x0100:		Stack:

FIGURE 2 – fichier objet (.yo) généré à partir d'un code source (.ys)

De gauche à droite, pour chaque ligne :

- (a) L'adresse de l'instruction (en hexadécimal)
- (b) Le codage de l'instruction (en hexadécimal)
- (c) L'instruction même, telle qu'elle est dans le code source

3. .hcl :

Fichier décrivant le comportement du processeur face à une instruction.

```

143  ## Select input A to ALU
144  int aluA = [
145      icode in { IRRMOVL, IOPL } : valA;
146      icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
147      icode in { ICALL, IPUSHL } : -4;
148      icode in { IRET, IPOPL } : 4;
149      # Other instructions don't need ALU
150  ];

```

FIGURE 3 – Exemple de code HCL

Dans cet exemple, si l’instruction est IRRMOVL, alors valA sera mise dans aluA. Le code HCL est ensuite compilé dans le langage utilisé par le simulateur. De cette manière, la simulation pourra appeler les fonctions générées et ainsi savoir comment traiter chaque instruction.

3 État de l’art

3.1 Application web [2]

Actuellement, la modification de quelques fichiers du simulateur web est nécessaire pour ajouter ou modifier une instruction :

- **ace/mode-y86.js** qui contient les expressions régulières acceptées par le compilateur.
- **assem.js** qui contient les fonctions nécessaires pour effectuer la conversion du code assembleur en code binaire.
- **general.js** qui contient les listes d’instructions avec leurs codages (*icode* & *ifun*).
- **syntax.js** qui contient une liste de toutes les syntaxes possibles.
- **instr.js** qui contient un tableau avec les actions de chaque instruction. Ce fichier pourra nous être utile pour récupérer l’état du processeur pour chaque étage au cas par cas).

Nous sommes entrain de réfléchir comment bien prendre en compte les modifications de l’utilisateur tout en gardant le sens pédagogique de l’outil avec un code flexible et modulaire.

3.2 Application x86 (standalone)

Cette application est le programme originel permettant de simuler un processeur y86 ainsi que sa mémoire. Elle a été créée par les professeurs R. E. Bryant et D. R. O’Hallaron, de la Carnegie Mellon University. Elle propose un mode graphique (Tcl/Tk) ainsi qu’un mode textuel pour une

utilisation depuis un terminal. Le comportement du processeur face aux instructions peut être modifié en utilisant le langage HCL. Ce fichier HCL est converti en fichier source C, compilé, puis lié au simulateur durant la phase d'édition de lien. Chaque modification de ce fichier nécessite donc de recompiler le simulateur (ou à minima de ré-effectuer l'édition de liens).

Plus d'informations à propos de cette application sont disponibles sur le wiki du GitLab [\[1\]](#).

4 Description des besoins

Nous présenterons ci-dessous les différents besoins auxquels devra répondre l'application développée. Du fait que nous partirons très probablement d'une application déjà existante, certains besoins seront donc déjà implémentés.

Une des principales contraintes étant la facilité d'accès pour les étudiants et la portabilité, les besoins seront décrits dans l'optique du développement d'une application web.

4.1 Besoins fonctionnels

Fonctionnalités de l'application :

1. Saisie du code y86

(a) Éditeur de code y86

Il doit respecter les contraintes suivantes :

i. Saisir du code (y86 uniquement)

ii. Afficher les numéros de lignes

iii. Colorer syntaxiquement le texte saisi

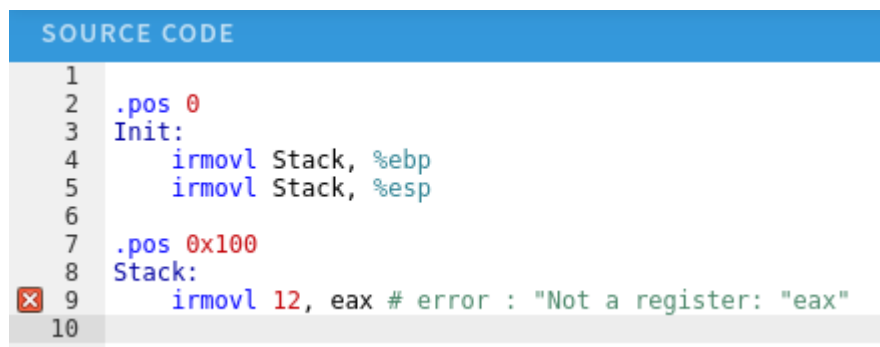
La coloration doit permettre de distinguer les différents éléments du langage, à savoir :

- Les instructions en bleu foncé
- Les constantes en rouge
- Les labels en noir
- Les registres en bleu clair
- Les commentaires en vert

iv. Analyser statiquement le code afin d'aider à son écriture

Si jamais le code saisi n'est pas valide, une croix rouge doit apparaître à gauche de l'éditeur. Si la souris est au dessus de cette croix, l'erreur doit être affichée.

L'analyse statique (ainsi que la coloration syntaxique) devront s'adapter à l'ajout / modification d'instructions. Voir 4 pour plus de détails.



The screenshot shows a code editor window titled "SOURCE CODE". The code is as follows:

```
1
2 .pos 0
3 Init:
4     irmovl Stack, %ebp
5     irmovl Stack, %esp
6
7 .pos 0x100
8 Stack:
9     irmovl 12, eax # error : "Not a register: "eax"
10
```

Line 9 has a red 'x' icon to its left, indicating an error. The error message "# error : 'Not a register: 'eax'" is displayed in green text to the right of the code line.

FIGURE 4 – Éditeur de code y86 utilisant l'analyse statique de code

(b) Charger un fichier de code y86 .ys

Bouton pour charger un fichier .ys depuis l'ordinateur de l'utilisateur. Le contenu de ce fichier est ensuite mis dans l'éditeur de code y86 décrit précédemment.

2. Compilation

L'application doit être capable de générer du code objet (**.yo**) à partir du code présent dans l'**éditeur de code y86**.

Attention cependant, le compilateur y86 devra pouvoir s'adapter à l'ajout / modification d'instruction. Se référer au besoin **Personnalisation du simulateur** pour ça (voir 4). Les contraintes suivantes doivent être respectées :

(a) Bouton de compilation

Il permet de lancer la compilation de **ys** vers **yo**. Le code source est récupéré dans le fenêtre d'édition de code.

(b) Affichage du code objet généré

Une fenêtre doit afficher le code **.yo** généré. L'affichage du code objet doit respecter le format des fichiers **.yo**. Un exemple est disponible dans la partie de mise en contexte (voir 2).

(c) Gestion des erreurs de compilation

Si le code en entrée n'est pas valide, l'erreur de compilation doit être affichée, en rouge, ainsi que la ligne incriminée.

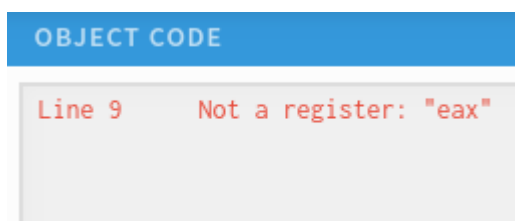


FIGURE 5 – La fenêtre du code objet affiche l'erreur en rouge

3. Exécution

Si du code objet est présent (dans la fenêtre de compilation vue précédemment), l'utilisateur doit pouvoir exécuter ce code, d'une traite ou étape par étape, et doit pouvoir voir les valeurs contenues dans les registres, dans la mémoire ainsi que les drapeaux de conditions. Cela correspond à l'état de la machine. Parallèlement à ça, l'état du processeur doit aussi être visible, c'est à dire les valeurs contenues dans les différents registres du processeur après l'exécution d'une instruction.

Voici les besoins détaillés :

(a) Affichage de l'état de la machine

i. Affichage des registres

Les registres de `%eax` à `%edi` (nom, valeur en base 8, valeur en base 10)

Exemple : `%eax 0x00000100 256`

ii. **Affichage des drapeaux de condition**

Ces drapeaux sont utilisés pour savoir si des "jumps" seront effectués. Ils sont de la forme (nom, valeur en base 10)

Exemple : **SF 0**

iii. **Le statut du processeur**

Il permet de voir le statut à proprement parlé (OK, HALT) ainsi que le compteur d'instruction, qui contient l'adresse de la prochaine instruction à exécuter.

REGISTERS			FLAGS		
%eax	0x00000000	0	SF	0	ZF 0 OF 0
%ecx	0x00000000	0			
%edx	0x00000000	0			
%ebx	0x00000000	0			
%esp	0x00000100	256			
%ebp	0x00000100	256			
%esi	0x00000000	0			
%edi	0x00000000	0			

STATUS	
STAT	HLT
ERR	
PC	0x000d

FIGURE 6 – Registres du processeur

(b) **Affichage de l'état du processeur**

Affiche les différents *stages* du processeur, à savoir **Fetch**, **Decode** / **Read**, **Execute**, **Memory** et **PC update** en séquentiel.

Chaque *stage* affiche une ou plusieurs valeur. Ces valeurs peuvent être de forme textuelle, ou bien numérique.

Cette fenêtre est mise à jour après chaque exécution d'une instruction.

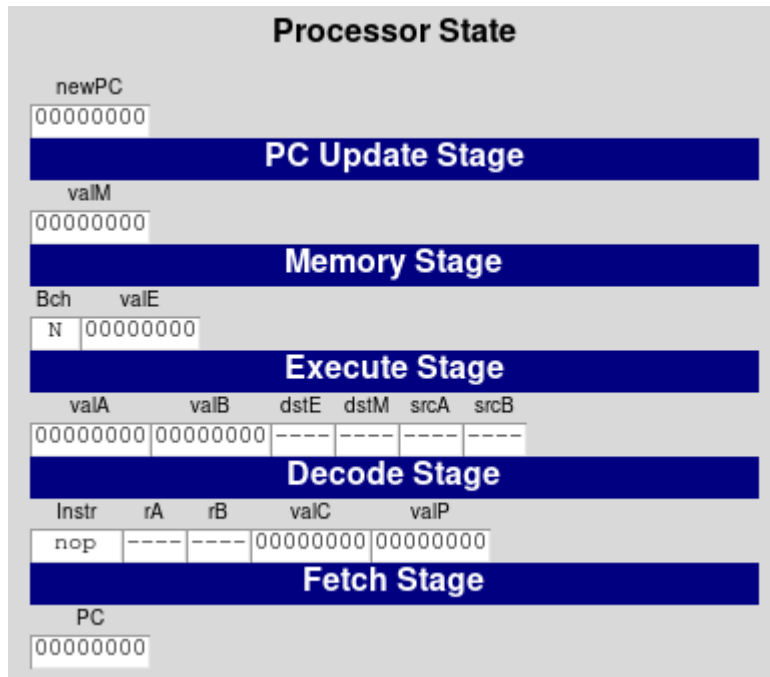


FIGURE 7 – Vue des différents *stages* du processeur

(c) **Aide à la visualisation des accès mémoire**

La mémoire utilisée par le processeur est affichée sous forme de liste. On peut y voir l'adresse ainsi que le contenu.

ADDR	VALUE
0000	30f41000 ◀ EBP
0004	0000404f
0008	10000000
000c	00000000
0010	00000000 ▶ ESP
0014	00000000
0018	00000000
001c	00000000

FIGURE 8 – Affichage de la mémoire. On peut y voir les valeurs des registres %ebp et %esp ainsi que les cases accédées (en bleu)

Lorsqu'un accès mémoire sera effectué, l'utilisateur devra en être notifié (par exemple saut sur la case concernée, pointée par une flèche ou mise en surbrillance). Cela à pour but de permettre à l'étudiant de facilement visualiser l'impact de son code sur la mémoire.Q

4. Personnalisation du simulateur

Le simulateur a pour principale contrainte d'être personnalisable. Son comportement devra pouvoir être changé de manière relativement aisée. Par exemple, on pourrait souhaiter implémenter une version *pipelined* du processeur, en plus de la version *sequential* de base.

Un autre but pédagogique consiste à permettre aux élèves d'insérer / modifier des instructions. Afin de se représenter les dépendances entre les différents modules (HCL, yas, ISA, simulateur, etc...), un diagramme disponible en annexe 12 montre une potentielle esquisse d'architecture, ainsi que les dépendances entre les différents composants.

(a) Personnalisation des instructions

i. Éditeur d'instructions

C'est depuis cet éditeur que l'on ajoutera / supprimera / modifiera une instruction. Cet éditeur a plusieurs objectifs. Premièrement, vérifier que le nombre de couple (icode, ifun) n'est jamais dupliqué et qu'il n'y a pas plus de 16 instructions. Dans un second temps, il doit faciliter la vie des étudiants en simplifiant la saisie de nouvelles instructions.

```
{"irmovl", HPACK(I_IRMOVL, 0), 6, I_ARG, 2, 4, R_ARG, 1, 0 },
{"rmmovl", HPACK(I_RMMOVL, 0), 6, R_ARG, 1, 1, M_ARG, 1, 0 },
{"mrmovl", HPACK(I_MRMOVL, 0), 6, M_ARG, 1, 0, R_ARG, 1, 1 },
{"addl",   HPACK(I_ALU, A_ADD), 2, R_ARG, 1, 1, R_ARG, 1, 0 },
{"subl",   HPACK(I_ALU, A_SUB), 2, R_ARG, 1, 1, R_ARG, 1, 0 },
{"andl",   HPACK(I_ALU, A_AND), 2, R_ARG, 1, 1, R_ARG, 1, 0 },
{"xorl",   HPACK(I_ALU, A_XOR), 2, R_ARG, 1, 1, R_ARG, 1, 0 },
```

FIGURE 9 – Codage d'une instruction dans la version x86. De gauche à droite, l'identifiant, le couple (icode, ifun), la taille en octets de l'instruction, le type du premier argument (constante / registre / adresse mémoire), la position du premier argument une fois l'instruction encodée, la taille en octets du premier argument, le type du second argument, la position du second argument une fois l'instruction encodée, la taille en octets du second argument

Une version plus visuelle est donc attendue afin de limiter les erreurs en abstrayant le codage.

ii. Utilisation du HCL

Le HCL est le code à éditer pour indiquer comment traiter une instruction. Une fois le code HCL saisi, il faudra le compiler en JavaScript afin que le simulateur puisse y accéder. Ce besoin requiert donc :

A. Éditeur de HCL

Cet éditeur permettra de saisir du code HCL. A l'image de l'éditeur de code y86, l'analyse statique de code ainsi que la coloration syntaxique propre au HCL serait souhaitable.

B. Bouton de compilation

Ce bouton convertit le code HCL présent dans l'éditeur en code JavaScript qui pourra être appelé par le simulateur.

C. Fenêtre de visualisation de résultat

Adjacente à l'éditeur de code HCL, cette fenêtre devra afficher le code JavaScript généré à partir du HCL. A l'instar de la fenêtre de compilation du y86, les erreurs de compilation devront y être affichées en rouge si il en existe.

4.2 Besoins non fonctionnels

1. Interface multilingue

Nous allons devoir utiliser des fichiers de *strings.xml* pour afficher le texte de l'interface. Cela permet, dès le début de la conception de l'application, de supporter toute les langues. Cependant nous allons uniquement implémenter l'anglais, l'objectif de ce besoin est simplement de proposer une modularité de plus au niveau de l'interface.

2. Automatisation des tâches

Une fonctionnalité souhaitée est l'automatisation de certaines tâches, notamment pour les correcteurs. L'idée serait de faire des requêtes paramétrées qui retourneraient un résultat. A ce jour, la principale utilisation serait la récupération d'un dump de l'état de la machine après exécution d'un programme, avec un HCL et un IS donné, afin de voir avec quel registre / "case mémoire" le programme interagit.

Exemple :

```
wget <hostname> ?file=test.y86&hcl=student.hcl&conf=student.json  
&action=memorydump&breakline=10
```

Ce système implique une certaine vigilance lors de la phase de conception de l'application. En effet, la requête paramétrée étant reçue par le serveur, ce sera à lui de la traiter. Si le code du simulateur est exécuté seulement du côté du client, il faut penser à un moyen de traitement autonome pour le serveur. Plusieurs pistes sont bien évidemment envisageables. Ces dites pistes seront abordées dans un autre document que celui-ci.

3. Simulateur y86

Il s'agit du module qui exécutera les instructions contenues dans le .yo en simulant le processeur. Il appellera les fonctions fournies par le script généré à partir du HCL. A priori, ce module ne sera pas modifiable par l'utilisateur, mais seulement par les développeurs.

4. Accessibilité

L'application devra être ouverte aux personnes souffrant d'un trouble de la vision. La pensée principale vient aux personnes atteintes de cécité totale pour qui les outils actuels ne sont pas adaptés. Pour cela le document de référence de l'administration française¹ peut être d'une grande aide, ainsi que le recours à un spécialiste de la question, comme M. Samuel THIBAUT.

4.3 Classement des besoins par ordre de priorité

Dans l'intérêt de mener à bien ce projet, nous classerons les besoins selon leur priorité afin de ne pas disperser nos efforts.

nous garderons le simulateur actuel mais en y ajoutant :

- État du processeur
- Personnalisation du simulateur (Ajout d'instructions)
- Automatisation des tâches
- Interface multilingue
- Aide à la visualisation des accès mémoire

4.4 Changements des besoins

1. Automatisation des tâches

Ce besoin nous a été exprimé sous la forme décrite en section 2. Cela requiert un *back-end* qui traiterait ce type de requête, or l'application que nous allons développer est uniquement constituée d'un *front-end*. Le besoin risque donc de changer.

4.5 L'interface utilisateur

L'interface utilisateur telle que prototypée propose de mettre en place un système d'onglets et de boutons afin d'interagir avec l'utilisateur. Pour cela les trois ou quatre modes seront représentés par un onglet chacun. Le mode *Instruction set*, permettant l'ajout d'instruction, pourrait être intégré au mode *HCL* par la suite si cela permet une meilleure ergonomie.

Chaque onglet posséderait des interfaces différentes :

1. Référentiel Général d'Accessibilité pour les Administrations (RGAA)

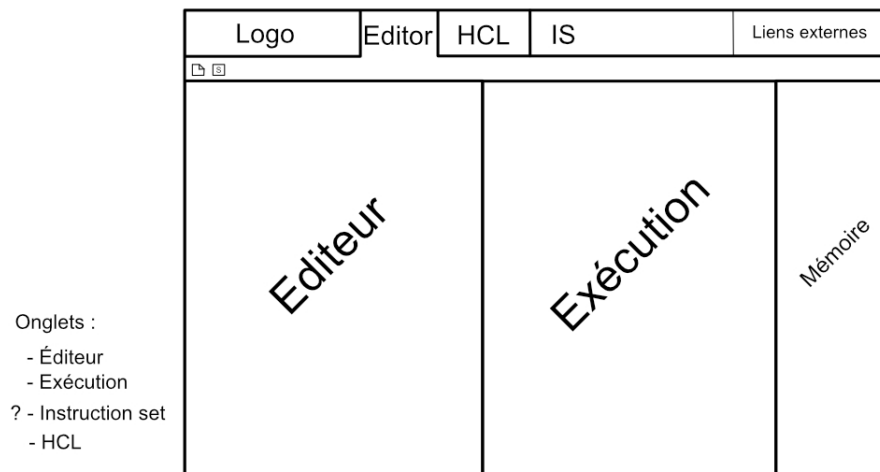
- **Éditeur** : Possède un éditeur de texte pour modifier les fichier .ys, une fenêtre d'exécution où l'utilisateur peut observer les instructions s'exécuter étape par étape (step-by-step) et un dump de la mémoire. Il peut aussi posséder les clés pour exécuter le programme tel que le boutons "Step" ou "Assemble". Ces derniers pourrons tout aussi bien être global (commun à tous les onglets). C'est une décision qui devra être prise en fonction des résultats de l'expérimentation.
- **HCL** : Le mode qui regroupe les fonctionnalités nécessaires au bon développement du HCL par l'utilisateur. Cette interface regroupe un éditeur de texte pour éditer le HCL et la vu des différents états du processeurs (Stages) à chaque instruction.
- **Mode avancé** : Cette vue proposée par le client permet de visualiser en parallèle le code qui s'exécute comme dans le mode Éditeur et le Stages du processeur comme dans le mode HCL. L'intérêt principal de ce mode est de permettre à l'utilisateur de comprendre ce qu'il se passe dans le processeur à chaque instruction.

L'ergonomie primera sur le choix finale de l'interface. Cette dernière doit cependant permettre de :

- visualiser l'état du processeur comme décrite figure 7.
- visualiser l'état de la mémoire.
- éditer le fichier .hcl et .ys
- visualiser l'état des registres et des flags.

Outre ces éléments de visualisation, l'interface doit aussi permettre de compiler et d'exécuter le programme. Soit d'un coup, soit étape par étape. Une fonction pour revenir au début de l'exécution (Reset) et une pour sauvegarder ou charger un fichier hcl ou ys extérieur sont aussi souhaitées.

En revanche, la possibilité de faire des breakpoints, bien que pratique, ne fait pas parti des besoins fonctionnels. L'accessibilité aux utilisateurs atteint de troubles visuelles ne doit pas obligatoirement se faire via cette interface.



Boutons :

- ⚡ Assemble
- ⌂ Reset
- > Set
- >> Continue
- 📄 Load
- 💾 Save
- Example
- Wiki
- Gitlab

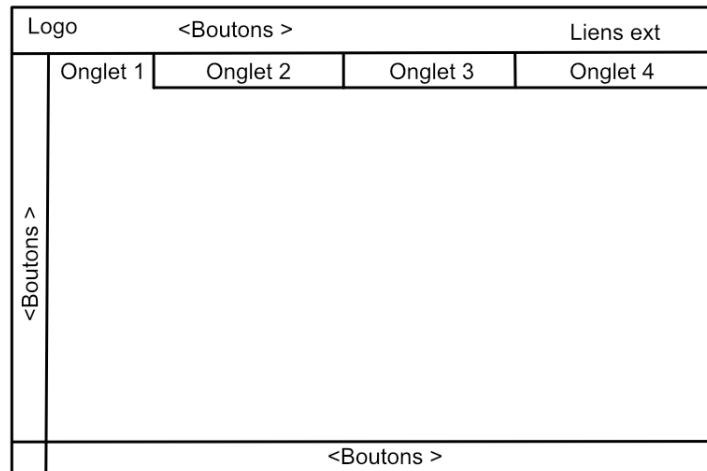


FIGURE 10 – Prototypes de l'interface utilisateur

5 Planification des tâches

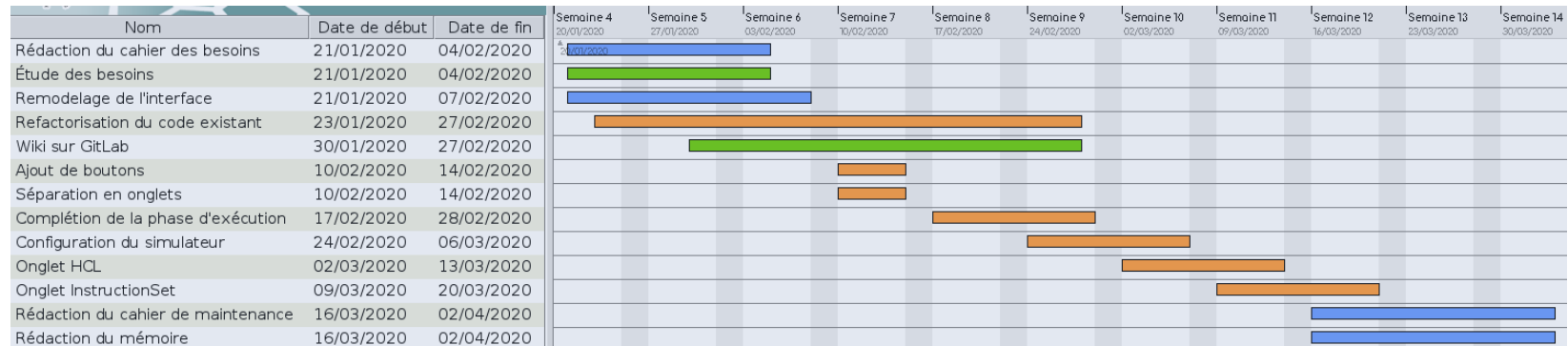


FIGURE 11 – Diagramme de Gantt

Légende :

- Rédaction de documents
- Travail de recherches
- Développement

Nous avons choisit de ne pas assigner les tâches à des membres du projet, car ce projet implique une importante phase d'étude des besoins. Suivit d'une phase de développement entièrement basée sur la précédente. Or certains besoins sont voués à changer car ils dépendent entre autre de la réalisation préalable d'autre besoins (voir la section 4.4).

Références

- [1] https://gitlab.inria.fr/computer_architecture/y86-environment/-/wikis/existant/application_x86. Description de l'application x86 utilisée pour l'UE Architecture des processeurs.
- [2] Aurélien ESNARD. <https://dept-info.labri.fr/ENSEIGNEMENT/archi/js-y86/index.html>. Simulateur y86 web utilisé à l'université de Bordeaux.
- [3] D. R. O'Hallaron R. E. BRYANT. <https://dept-info.labri.fr/ENSEIGNEMENT/archi/js-y86/index.html>. Application standalone originale pour simuler un environnement y86.

6 Annexes

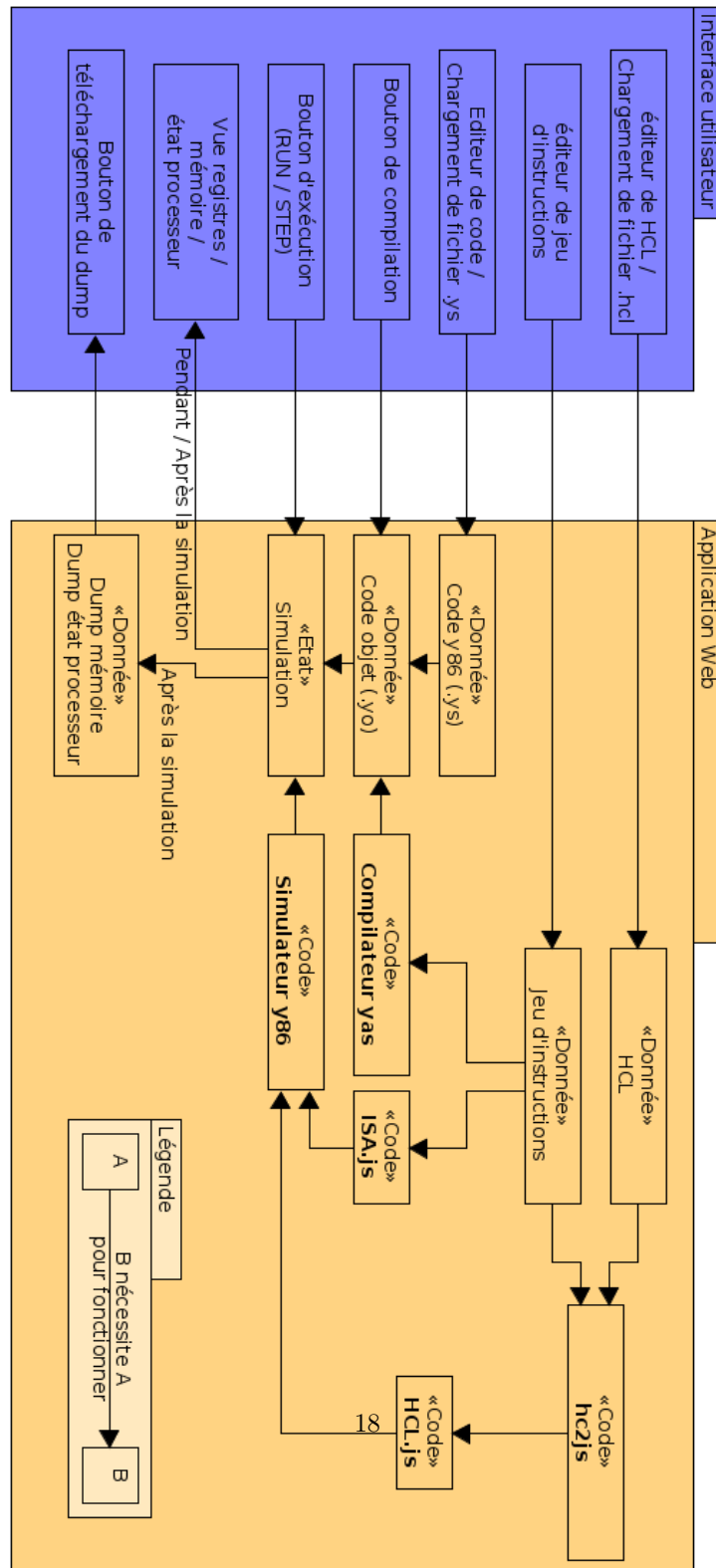


FIGURE 12 – Potentielle architecture pour une application web standalone

.1 Description de l'architecture

L'architecture affichée précédemment montre les différents composants ainsi que les inter-dépendances, symbolisées par des flèches. Ainsi, on peut clairement voir les modules "directeurs" de l'application. Par exemple, on peut remarquer que l'édition du jeu d'instruction sera essentielle pour compiler du code y86, permettre au simulateur de s'exécuter et compiler le HCL.js. Cette séparation des différents composants a pour objectif de simplifier le principe de modularité en réduisant le couplage au maximum.