Flexible Universal CPU Simulator

Projet de TER année 2006-2007

Encadré par Frédéric Mallet

Benjamin Baudouin
Olivier Orabona
Laurent Rodriguez

Sommaire

Partie I : Présentation du projet

1. Présentation générale

Notre projet FUCS (Flexible Universal CPU Simulator) a pour but de simuler le comportement d'un processeur au niveau des registres. Nous avons pour cela établi un cahier des charges détaillé, un site Internet[1] ainsi qu'un Wiki. Simuler un processeur signifie que nous attachons une grande importance à l'exactitude du comportement. Cette exactitude passe par la modélisation d'une interface polyvalente et d'une implémentation « type » d'un processeur très répandu : l'Intel x86. Tout au long du développement nous nous sommes attachés à avoir un résultat visible pour l'utilisateur. C'est à cette fin qu'une interface graphique a été prévue.

Notre projet se présente sous la forme d'une bibliothèque d'application assurant une interface uniforme entre une interface graphique « cliente » et un sous système de simulation caché de l'interface. Le langage de programmation C++ a été choisi car il est le plus en rapport avec notre conception logicielle.

Ce projet se veut définitivement orienté sur le modèle Open Source. Nous avons donc adopté une initiative et un mode de travail qui correspondent aux projets Open Source.

2. Objectifs

Les possibilités d'utilisation d'une telle application sont nombreuses, et nous avons défini dans le cahier des charges une liste assez conséquente d'objectifs. Mais pour des contraintes de temps et de ressources humaines nous avons dû nous limiter à quelques objectifs prioritaires.

- *modéliser un processeur* : l'idée de simuler un processeur x86 avec un jeu d'instructions CISC s'est décidée par l'abondance de documentation présente sur Internet.
- *concevoir une interface graphique* : afin d'avoir un résultat visuel acceptable et pouvant servir de démonstration de notre bibliothèque de simulation, la conception d'une interface graphique s'est revélée être la plus prometteuse en terme d'ergonomie. Nous avons choisi la bibliothèque wxWidget de par sa conception orientée objet et parce qu'elle est très portable.
- *simuler un programme test simple* : pour démontrer qu'une simulation de processeur est possible, il nous fallait un programme test simple

1 http://fucs.free.fr

car les contraintes de temps nous auraient empêchés de simuler un jeu d'instruction complet CISC. Nous nous sommes donc plus focalisés sur la modélisation du fonctionnement du processeur et nous avons validé cette modélisation par la simulation d'un programme dont on pouvait prévoir l'état à n'importe quel moment.

3. Contraintes

La réalisation d'un tel projet a fait ressortir des contraintes dont nous avons du tenir compte dès le cahier des charges :

- L'architecture 32bits d'un processeur x86
- La plateforme de développement

Nous avons orienté notre développement autour d'une simulation d'un processeur type. Ceci a posé clairement les limites de notre interface car nous ne pouvons pas dire à ce jour si notre modèle de conception peut permettre de simuler n'importe quel autre type de processeur, qu'il soit à jeu d'instructions différent, à architecture différente ou qu'il fasse partie d'une catégorie de processeurs n'ayant que peu de similitudes avec un processeur « classique ».

D'autre part, la plateforme de développement a dû être choisie en fonction des possibilités matérielles de chacun. En l'occurence, nous avons opté pour le système d'exploitation Linux sur PC car il est famillier et connu de tous et propose des outils de développement gratuits.

Partie II : Gestion du projet

1. Ressources humaines

Le travail n'a pu démarrer tant que nous n'étions pas sûr de l'acceptation du projet par le comité TER. Nous n'étions que deux au début, Olivier et Laurent, puis nous avons été rejoints par Benjamin. Des problèmes d'organisation se sont présentés au fur et à mesure du déroulement du TER.

2. Gestion du temps

Dans le cahier des charges, la répartition des tâches s'est faite suivant le diagramme suivant.

	Benjamin	Olivier	Laurent
Phase de mise à niveau (15/03/07 – 30/03/07)			
C++	X		X
Assembleur	X		
UML	X		X
Algorithme de chiffrement RSA	X		X
Phase de modélisation (09/04/07 – 21/04/07)			
Programmes tests simples		X	
IHM		X	
Implémentation RSA + jeu d'instructions nécessaire			X
Modélisation du processeur	X	X	X
Modélisation du moteur de simulation	X	X	X
Phase d'implémentation (24/04/07 – 12/05/07)			
Création d'un framework de composants	X	X	X
Création d'un chargeur de programmes RAW	X	X	X
Implémentation des fonctions de base d'exécution	X	X	X
Tests d'intégration	X	X	X

A la date du 16 mai, nous pouvons considérer que la répartition s'est faite suivant ce diagramme :

	Benjamin	Olivier	Laurent
Phase de mise à niveau (15/03/07 – 15/04/07)			
C++	X		X
Assembleur	X		
UML	X		X
Phase de modélisation (09/04/07 – 24/04/07)			
Programmes tests simples		X	
Modélisation du processeur		X	
Modélisation du moteur de simulation		X	
Phase d'implémentation (9/04/07 – 16/05/07)			
IHM	X		X
Création d'un framework de composants		X	
Création d'un chargeur de programmes RAW		X	
Implémentation des fonctions de base d'exécution	X	X	X
Tests d'intégration	X	X	X

Comme nous pouvons le constater, les deux diagrammes ne correspondent pas.

- La phase de mise à niveau, en C++ et wxWidgets, a pris plus de temps que prévu. Le C++ est connu pour être un langage de haut niveau assez complexe à appréhender et la bibliothèque graphique wxWidgets a souffert d'un manque de documentation extensive. L'achat d'un livre sur cette bibliothèque s'est d'ailleurs révélé nécessaire mais a ralenti l'implémentation de l'IHM. C'est aussi pour cette raison qu'il n'y a eu quasiment aucune modélisation de l'interface utilisateur.

- La répartition des tâches s'est faite comme prévu, Benjamin et Laurent se chargeant de la partie graphique. Ayant commencé plus tôt la modélisation et l'implémentation d'un embryon de bibliothèque, j'ai eu très rapidement besoin d' « interface » utilisateur afin de valider l'intégration de l'ensemble de l'architecture logicielle de FUCS. J'ai donc dû entamer l'écriture d'une pseudo interface en mode texte, cmdFucs.

- Nous avons dû renoncer à simuler une implémentation de l'algorithme de chiffrement RSA. Cette décision fut prise suite aux recommandations faites lors de la présoutenance. Le nombre d'instructions à simuler étant trop important, nous aurions perdu trop de temps à programmer le décodeur d'instructions CISC au détriment d'autres parties du simulateur.
- La modélisation d'un processeur avec un jeu d'instructions CISC s'est révélée bien plus complexe que prévue, essentiellement dûe au décodage des instructions et à leur complexité intrinsèque.

3. Gestion des risques

Lors de l'élaboration du cahier des charges, nous avions déjà identifié des risques potentiels :

- L'approfondissement des connaissances en C++
- L'apprentissage de la bibliothèque graphique wxWidgets
- La présence d'un étudiant salarié

L'approfondissement des connaissances en C++ a duré plus longtemps que prévu pour Benjamin et Laurent car c'est un langage de programmation bien plus complexe à appréhender que le Java. Les connaissances initiales de ce langage devaient être revues et, avec les contraintes liées au planning (examens, semaine de révision, etc.), ont allongé les délais de cette partie.

L'apprentissage de la bibliothèque graphique a elle aussi été plus complexe qu'initialement prévu. L'absence d'une documentation et de tutoriaux bien expliqués sur Internet ont rendu le développement hasardeux et plus long. L'achat d'un livre sur wxWidget s'est avéré d'ailleurs indispensable mais compte tenu des délais, il n'a pu être exploité de manière optimale.

Benjamin, étudiant salarié, a bénéficié d'un planning ajusté. Ayant été affecté au développement de la partie graphique avec Laurent, ils ont dû se synchroniser. Cette synchronisation ne s'est pas faite aussi facilement. L'information circulant assez mal sur les points à faire et ceux déjà faits a laissé Benjamin en état d'attente. Il était aussi chargé de tester l'application au fur et à mesure sur sa machine, afin d'être sûr que les applications pouvaient être compilées sur des machines différentes.

En plus de ces problèmes, d'autres se sont ajoutés en cours de route :

- L'apprentissage des outils de développement GNU : notamment des outils automake et autoconf qui sont incontournables dans le développement de projets Open Source dans le monde Unix et Linux. Leur prise en main a été plus longue et a abouti à un allongement des délais

non négligeable dans la mise en place des projets.

- Le planning de chacun est aussi à l'origine d'une série de retards. En effet, pour des raisons personnelles, nous avons eu des jours où le travail n'a pu avancer. Ces retards et leurs origines sont tous décrits dans la page d'avancée personnelle de chacun.

4. Gestion des activités

La liste des activités et les pourcentages d'accomplissement sont détaillés dans le diagramme suivant :

Comme nous pouvons le constater, l'architecture manque de tests rigoureux et d'une validation permettant de justifier de sa flexibilité.

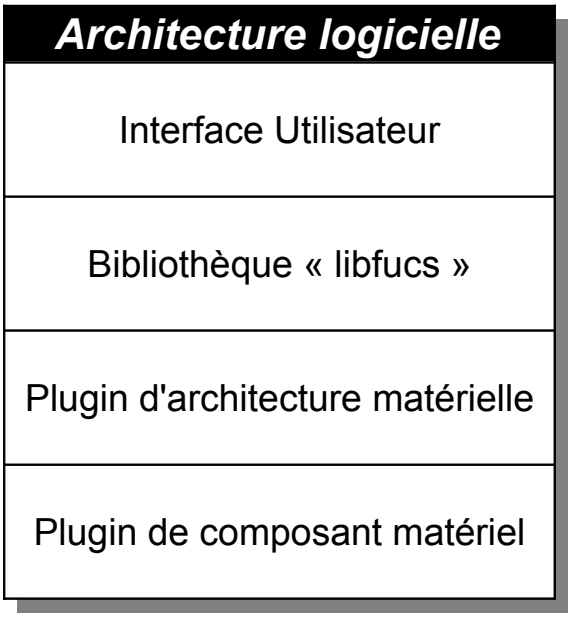
D'autre part, nous tenons à signaler que les pourcentages donnés ici sont à titre indicatif. Nous n'avons pas établi de règle fixe et nous avons toujours essayé de garder les mêmes fonctionnalités d'une version à l'autre afin d'éviter de casser la compatibilité ascendante. Elle n'est certes pas un critère car actuellement les programmes et bibliothèques sont encore dans une phase de développement intense, et leurs interfaces ne sont pas encore suffisamment finalisées pour imaginer changer de version officiellement.

Activité	Pourcentage	To Do
Site Web	90,00%	Mise en production du gestionnaire de bogues
libfucs	*80,00%*	*Détails ci dessous*
Interface « proxy »	100,00%	
Chargement d'un programme « raw »	100,00%	
Tests unitaires / intégration	50,00%	Architecture de tests unitaires automatiques
Intégration d'autoconf dans le projet	95,00%	Quelques bugs et validation
fucs_x86_generic	*60,00%*	*Détails ci dessous*
Gestion du processeur	80,00%	Tests unitaires
Gestion de la mémoire	80,00%	Tests unitaires
Gestion du pipeline	50,00%	Précision dans le décodage, vérification de la flexibilité de l'architecture, complétion de l'unité de décodage CISC
Intégration d'autoconf dans le projet	95,00%	Quelques bugs et validation
wxFucs	*50,00%*	*Détails ci dessous*
Fenêtre de choix de perspectives	100,00%	
Partie « software » de FUCS	20,00%	
Fenêtre de débogage	70,00%	
Interraction avec les registres	90,00%	
Interraction avec la memoire	95,00%	
Affichage du code	20,00%	
Objectifs optionnels	0,00%	(voir cahier des charges)
Intégration d'autoconf dans le projet	90,00%	

Partie III : Choix techniques

1. Architecture logicielle

L'architecture logicielle choisie et approuvée par notre encadrant peut être représentée par un modèle en couches. Nous allons les décrire ci-dessous, de la partie la plus visible à la partie la moins visible – du point de vue de l'utilisateur - :



Cette architecture en couches permet de sceller chaque couche de l'application. Ainsi, l'interface utilisateur n'a pas besoin de connaître l'architecture matérielle sous jacente pour afficher le résultat de la simulation. Nous avons donc une indépendance totale et nous pouvons interchanger chaque élément sous libfucs sans que cela ne crée d'incidence sur l'interface utilisateur.

Afin d'achever ce niveau d'indépendance, la bibliothèque libfucs doit répondre au patron de conception « proxy » et sert donc de point d'entrée pour l'interface utilisateur. Elle remplit le rôle de moteur de simulation en chargeant une architecture matérielle ainsi qu'en contrôlant son exécution.

Le plugin d'architecture matérielle et le plugin de composant matériel peuvent ne faire qu'un, car les composants matériels sont fortement dépendants de l'architecture matérielle qui les utilise. Cependant, afin d'offrir un maximum de flexibilité dans la conception de processeur, nous avons jugé utile de laisser au programmeur le loisir de pouvoir interchanger des composants matériels de plusieurs architectures différentes. Nous avons

donc du refléter cette contrainte dans le modèle initialement prévu à trois couches. L'architecture matérielle peut donc charger n'importe quel autre plugin de composant et l'utiliser. Un principe de chainage peut être mis en place à la manière du patron « composite », déléguant les tâches non gérées par un plugin, à son parent.

2. Bibliothèque libfucs

La bibliothèque libfucs est le moteur de simulation. Cliente de l'interface graphique, elle contrôle tout le processus sous jacent de simulation matérielle. Elle est constituée essentiellement de classes d'interfaces qui sont à implémenter par les plugins d'architecture matérielle et leurs composants.

Ces classes d'interfaces sont au nombre de quatre.

- l*a classe d'interface CPU* qui modélise les fonctionnalités basiques dûes à un processeur. Il s'agit essentiellement de la manipulation de registres, qu'ils soient internes ou externes. Ainsi, les processeurs x86 ont depuis quelques générations adopté un principe de coeur à architecture RISC tout en conservant les registres et le jeu d'instructions CISC. Les registres externes sont donc liés au jeu d'instructions CISC, alors que le CPU, une fois décodées les instructions par l'unité de décodage du pipeline, manipulera des registres internes RISC.
- *la classe d'interface Memory* qui se charge de la représentation des données en mémoire. Cette dernière a dû bénéficier de plus d'optimisations que les autres. La raison principale est qu'une modélisation d'un processeur x86 sur 32bits nécessite un bus de données mémoire de 32bits, ce qui autorise une allocation sur 4Go. Or nous ne pouvons nous permettre d'allouer ni statiquement, ni dynamiquement, une telle quantité de mémoire. Nous avons donc du prendre des mesures afin de ne stocker que les emplacements mémoire réellement utilisés, et adopter des valeurs par défaut pour les autres.
- *la classe d'interface Pipeline* qui utilise une approche plus souple afin de n'avoir que peu de contraintes sur les composants (unités logiques) formant le pipeline. En effet, si nous décidons de fournir une architecture matérielle où le pipeline n'utilise jamais la mémoire, nous devrions pouvoir concevoir un tel pipeline sans avoir une obligation d'un minimum requis. Le pipeline est donc la partie de l'architecture matérielle la plus flexible.
- *la classe d'interface Observer*. Chaque classe d'interface possède une version spécifique implémentant un patron de conception « observer ». Ce patron permet de notifier de tout événement se

déroulant un instant donné dans une classe donnée sans violer l'encapsulation de cette dernière. Ainsi il est possible au niveau de l'interface utilisateur d'enregistrer une classe « observer » spécifique au CPU, à la mémoire ou au pipeline, et tout événement se produisant dans chacun de ceux-ci sera automatiquement transmis à la classe « observer » qui pourra ainsi mettre à jour l'interface, quelle qu'elle soit. Nous avons donc une bien meilleure souplesse dans le rapport d'événements et un retour utilisateur accru.

3. IHM

L'interface graphique propose un outil de débogage permettant d'intervenir sur la valeur des différents registres, des différentes zones mémoires ainsi que sur l'exécution de l'application à déboguer.

Elle proposera à terme les différents outils graphiques optionels définis dans le cahier des charges tel que l'exécution controlée d'une application, des outils d'optimisation et de validation d'application, ainsi qu'un jeu d'outils permettant d'agir sur la bibliothèque de manière ergonomique et aisée pour la conception, la comparaison d'architectures de processeurs ainsi qu'un outil pédagogique destiné à l'apprentissage illustré du fonctionnement interne de processeurs.

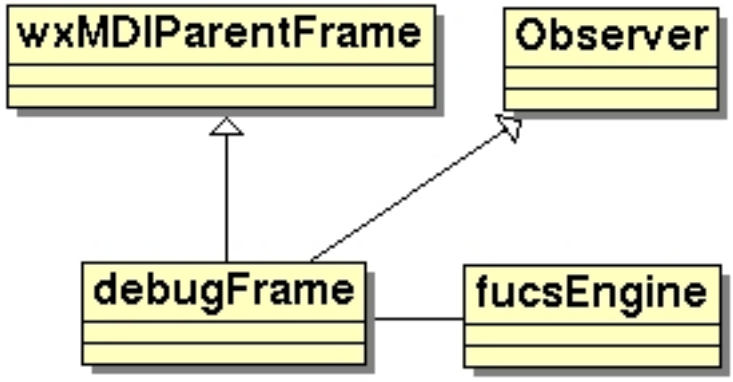
Une fenêtre proposant un choix sur les différents outils a été conçue afin de simplifier le lancement de l'application par un unique programme exécutable.

Un ensemble de symboles a été choisi permettant une mémorisation rapide, ainsi qu'un repérage facile de l'outil voulu. Une aide contextuelle est fournie afin d'identifier chaque outil par le simple survol de son symbole à la souris.

L'outil graphique de débogage d'application contient un menu permettant d'accomplir les tâches se rapportant au chargement d'architectures et d'applications, au contrôle d'exécution, et à l'aide en ligne. Une barre d'outils permet d'accéder plus rapidement aux fonctions du menu les plus utilisées. La surface de la fenêtre est divisée en trois zones redimensionnables, permettant l'affichage et la modification du code ainsi que du contenu des registres et zones mémoires.

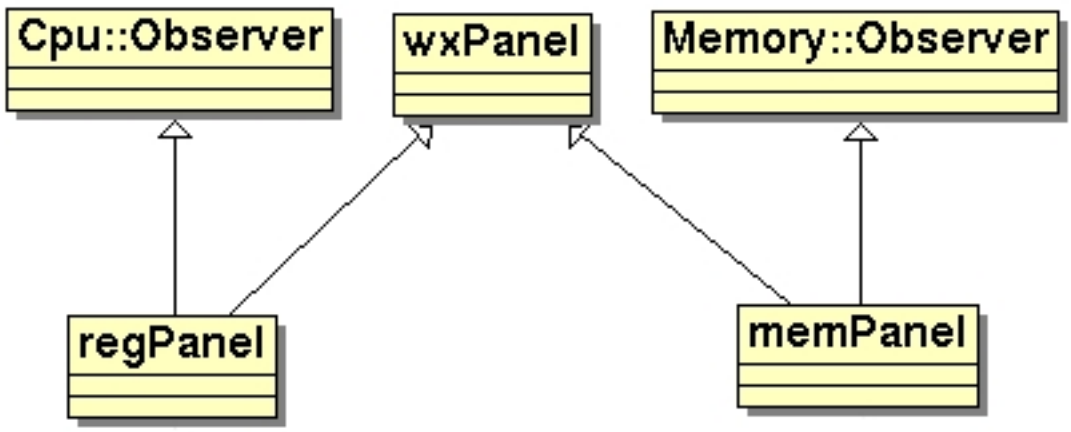
On le décompose selon la hiérarchie suivante (nous nous bornons à la plus pertinente dans l'implémentation, les objets servant à la mise en page, menu et barre d'outils ne sont que peu ou pas présentés).

La fenêtre est composée de trois « wxSashLayoutWindow » permettant la division de sa surface en trois panneaux redimensionnables à volonté par l'utilisateur.

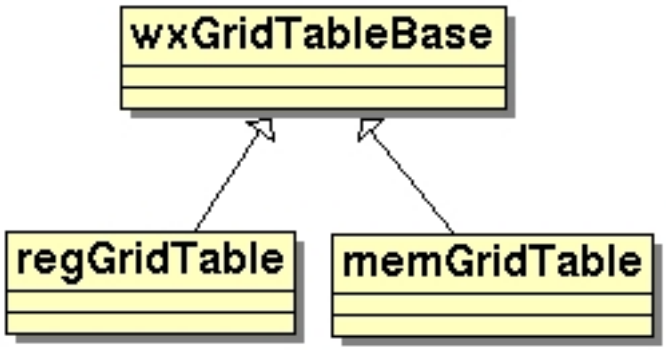


La fenêtre étend wxMDIParentFrame ce qui lui permet de contenir et de gérer les wxSashLayoutWindow qui la composent. L'interface Observer est implémentée afin de surveiller et afficher les erreurs internes au fucsEngine.

Les panneaux d'affichage des registres et des zones mémoires sont contenus dans deux des wxSashLayoutWindow de la fenêtre, et implémentent chacun un écouteur sur la partie de libFucs concernée. Cela sera détaillé plus loin.



Chaque panneau contient une instance de wxGrid remplie par des instances de regGridTable et memGridTable détaillées par la suite.



De manière plus approfondie, les parties affichage et modification de mémoires et registres passent par l'extention des classes wxGridTableBase permettant de simuler des tables virtuelles représentant le contenu de la mémoire et des registres en espace constant.

Ceci est possible car l'objet interroge la mémoire à chaque fois que nécessaire pour afficher les données requises par la zone visible de la table. Il ne stocke pas l'information et ne nécessite ainsi pas d'espace mémoire supplémentaire pour la représenter. Ces requêtes augmentent la complexité en temps de l'application mais de manière linéaire et cela reste insensible lors de l'utilisation.

Les changements d'état provoqués par l'exécution sont connus grâce à l'implémentation des interfaces Observer correspondantes (Memory::Observer pour l'affichage de la mémoire et Cpu::Observer pour celui des registres) ce qui permet la mise à jour de l'affichage en accord avec le nouveau contenu du registre ou espace mémoire modifié. Ceci assure la concordance des valeurs affichées avec celles contenues dans les différents objets représentés, même si la zone d'affichage reste la même.

Le changement manuel de valeurs est possible grâce à l'accès à l'objet fucsEngine instancié par la fenêtre principale de l'outil. L'objets wxGrid le permet aisément par simple sélection à la souris ou au clavier de la case représentant la zone mémoire ou registre à modifier, et la saisie de la nouvelle valeur à lui donner, ces valeurs sont directement mises à jour dans la mémoire via l'interface de la bibliothèque.

Partie IV : Conclusion

1. Aspects positifs et négatifs

Tout au court du développement, nous avons été confrontés à des problèmes liés à la gestion de projet. Nous avions anticipé une partie d'entre eux mais malheureusement d'autres problèmes n'avaient pas été prévus. La gestion du projet telle qu'elle aurait dû être pour tout projet « open source » a souffert de problèmes de synchronisation entre les différents membres. Les outils mis à notre disposition ne se sont pas avérés être utilisés aussi régulièrement que nécessite un tel projet avec ses contraintes temporelles. Malgré ces contretemps, nous avons réussi dans notre entreprise. Le sujet, complexe et nécessitant beaucoup de prérequis a finalement pu être complété en temps et en heure. Nous avons tous appris quelque chose dans cette expérience. Que ce soit d'un point de vue technique mais surtout dans notre façon de nous autogérer. Ce critère est d'ailleurs souvent un point important lors d'un recrutement et nous avons tous pu nous confronter à ce que ce TER représentait : un premier pas dans la vie active.

2. Résultats obtenus

Nous avions établi au départ un cahier des charges qui prévoyait l'étude de l'algorithme de chiffrement RSA. Celui ci fut supprimé pour des raisons de délais qui ne pouvaient être tenus. Néanmoins, tous nos objectifs prioritaires ont été accomplis, rendant ainsi possible une démonstration dont nous aurons la joie de vous faire part lors de notre soutenance orale – dans la mesure où cela est possible – mercredi 23 mai.

3. Futur

Comme vous avez pu le lire dans les section « nouvelles » du site Internet, nous avons d'ores et déjà entamé un processus d'enregistrement du projet sur SourceForge. Cette initiative souligne à quel point nous considérons que ce projet a de l'avenir dans le monde du logiciel libre et qu'il ne s'achèvera pas en même temps que notre année. Nous pensons continuer le développement tout au long de l'été, toujours suivant nos disponibilités, mais en ayant tous appris de nos erreurs. Nous sommes impatients à l'idée d'avoir des réactions d'utilisateurs, de nouvelles idées et du retour sur expérience utilisateur.