Manuel de référence de l'environnement Capua

V 1.0 – Janvier 2016

Historique du document

Date	Remarque
Janvier 2016	Création du document
Février 2016	Mise à jour du graphique de l'architecture et ajout de documentation concernant l'horloge et le dispositif STMP.

Table of Contents

Introduction

L'environnement Capua est noyau virtuel conçu pour aider les novices en matière de programmation en langage d'assemblage, et de la programmation de systèmes en général, à découvrir les fondements liés auxdits concepts. Capua a été mis au point après avoir constaté une baisse des capacités des étudiants nouvellement diplômés à évoluer dans un environnement de programmation de bas niveau. Par conséquent, les fonctionnalités de Capua sont réduites au strict minimum de manière à ce que les utilisateurs puissent tirer profit de la simplicité de sa conception dans un contexte d'apprentissage.

Capua par rapport à Spartacus

Spartacus est la machine virtuelle (MV) complète, alors que Capua est le noyau à partir duquel la machine virtuelle est conçue. Spartacus offre tout ce qui est nécessaire pour l'exécution d'un code d'usager. Il offre à Capua (le noyau) un contrôleur de mémoire mappé en mémoire, une matrice mémoire et un contrôleur d'entrée-sortie. L'architecture complète est basée sur un dispositif de mémoire mappée. Au moment de la rédaction du présent document, la MV Spartacus ne prenait en charge aucune forme d'interruption. Cette fonctionnalité sera mise au point ultérieurement.

Capua

Capua est une architecture de type *load/store* (charger/stocker). Ainsi, seules les instructions de chargement (**`MEMR`**) et de stockage (**`MEMW`**) peuvent accéder à la mémoire. Veuillez noter que les instructions relatives aux piles ont également accès à la mémoire, mais de façon beaucoup plus restreinte. Capua est un noyau 32 bits composé de quatre registres généraux, d'un registre d'adresse d'instruction et d'un registre à drapeaux. Un des quatre registres généraux peut être utilisé comme registre de pointeur de pile. Dans ce cas, on ne peut plus utiliser ce registre comme registre général à moins que sa valeur soit sauvegardée dans une adresse mémoire connue au préalable.

Registres

Les registres généraux sont tous de 32 bits. Ces registres sont nommés **`A`**, **`B`**, **`C`** et **`S`**. Le registre **`S`** (pour **S**tack) est le registre de pointeur de pile désigné. Les autres registres généraux n'ont pas d'utilisation précise, sauf dans le cadre du recours aux instructions **`MUL`** et **`DIV`**. Des renseignements supplémentaires à ce sujet sont présentés dans la section pertinente. Il est important de prendre note du fait que les registres généraux manipulent des entiers représentés sur 32 bits signés.

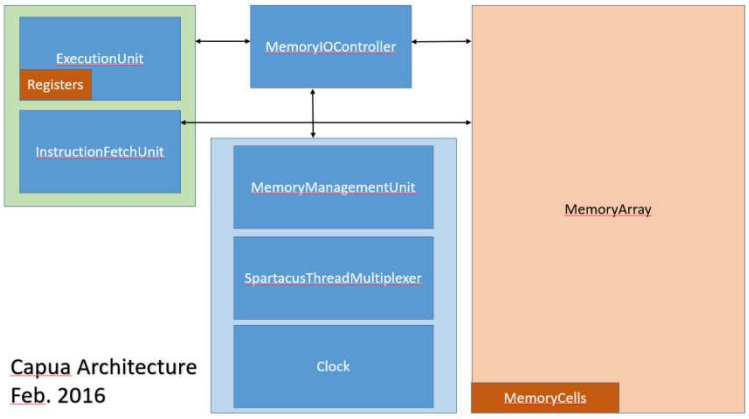
Le registre d'adresse d'instruction est nommé **`I`** et n'est pas accessible aux utilisateurs. Le registre à drapeaux est nommé **`FLAGS`** et n'est pas non plus accessible aux utilisateurs. Ces deux registres sont modifiés par un effet secondaire des instructions. Veuillez noter qu'à l'heure actuelle, il n'existe aucun drapeau de débordement. Par conséquent, si vous avez besoin de cette fonctionnalité, vous devez l'intégrer vous-même dans votre logiciel.

Mémoire

À l'heure actuelle, aucune unité de gestion de mémoire n'est offerte sur la MV. On accède donc directement à la mémoire. L'adresse de début de la mémoire est définie dans **`Configuration.Configuration.MEMORY_START_AT`**. La valeur recommandée par défaut pour cette adresse est **`0x40000000`**. La fin de la mémoire est définie comme **`Configuration.Configuration.MEMORY_END_AT`**. La MV possède 1 Mo de mémoire par défaut. On ne recommande pas de modifier l'adresse de début de la mémoire, car cela pourrait entraîner des problèmes avec le dispositif de mémoire mappée. On peut accéder au dispositif de mémoire mappée par l'intermédiaire des adresses répertoriées sous l'adresse de début de la mémoire.

Architecture générale

Le graphique ci-dessous représente l'organisation générale de la conception de la MV.



Comme vous pouvez le constater, l'architecture est très simple, mais elle ressemble à celle d'un vrai ordinateur.

Section 1 – Jeu d'instructions

Format général

Le jeu d'instructions de Capua compte actuellement 21 instructions différentes. La plupart d'entre elles peuvent faire appel à des registres ou à des valeurs immédiates (ou à un mélange des deux) aux fins d'exécution. La longueur des instructions varie; par conséquent, un décalage de saut non valide entraînera un comportement indéfini en fonction de ce qui se trouve dans la mémoire à l'endroit de destination de l'instruction du saut. Les quatre premiers bits définissent le format de chaque instruction exécutée. Les quatre bits suivants définissent une instruction au sein d'un ensemble de formats. La section suivante donne des précisions quant aux divers formats d'instruction disponibles dans l'architecture de Capua en décrivant leurs propriétés binaires.

Format 1 – Mnémonique (`0b1111`)

Le format 1 décrit des instructions qui ne font appel à aucun registre ni à aucune valeur immédiate. Les instructions **`NOP`** et **`RET`** sont des exemples de ce format. Étant donné que les instructions du format 1 ont uniquement besoin d'un format et d'un

identificateur d'instruction, elles sont longues d'un octet.

0000	0000
Format	Instruction

Format 2 – Mnémonique registre (`0b0111`)

Ce format décrit des instructions qui font appel à un seul registre. Les instructions **`NOT`** et **`PUSH`** (lorsqu'elles sont utilisées avec un registre) constituent des exemples de ce format. Comme pour tous les formats d'instruction, le premier octet décrit le format et les codes d'instruction. Dans le cas de l'instruction au format 2, le second octet sert à préciser le registre utilisé pendant l'opération.

0000	0000	0000 0000
Format	Instruction	Registre

Format 3 – Mnémonique immédiate (`0b1000`)

Ce format est similaire aux instructions du format 2. Cependant, il fait appel à une valeur immédiate plutôt qu'à un registre. L'instruction **`PUSH`** est disponible dans les formats 2 et 3. Les instructions du format 3 sont longues de cinq octets.

0000	0000	0000 0000 0000 0000 0000 0000 0000 0000
Format	Instruction	Immédiate (4 octets)

Format 4 – Mnémonique immédiate registre (`0b0110`)

Ce format regroupe des instructions telles **`MOV`**, **`SHR`** et **`XOR`**. La valeur immédiate de ce format agit à titre de source, alors que le registre constitue la destination. L'opération est appliquée à la valeur du registre de destination au moyen de la valeur immédiate fournie dans l'instruction. Les instructions de ce format sont longues de six octets.

0000	0000	0000 0000 0000 0000 0000 0000 0000 0000	0000 0000
Format	Instruction	Immédiate (4 octets)	Registre de destination

Format 5 – Mnémonique taille immédiate registre (`0b0000`)

Les instructions de ce format sont utilisées pour accéder à la mémoire. En fonction de l'instruction utilisée, la valeur immédiate peut être une adresse mémoire (en cas de lecture à partir de la mémoire) ou une valeur (en cas d'écriture dans la mémoire). Le registre doit également contenir des données appropriées en fonction de l'utilisation. Le champ de la taille permet de préciser la longueur des données que l'on souhaite lire ou écrire depuis ou sur la mémoire. Les instructions de ce format sont particulières en ce que, à des fins d'optimisation du codage des instructions, le registre de destination et

la valeur immédiate sont inversés dans l'instruction codée. Cela permet à ces instructions d'atteindre une longueur de six octets.

0000	0000	0000	0000	0000 0000 0000 0000 0000 0000 0000 0000
Format	Instruction	Taille	Registre de destination	Immédiate (4 octets)

Format 6 – Mnémonique taille registre registre (`0b0001`)

Comme pour le format 5, les instructions de ce format sont utilisées pour accéder à la mémoire. La seule différence est que la valeur immédiate est remplacée par un registre supplémentaire. Ce format ne requiert pas non plus d'inversion des champs. Ces instructions sont longues de deux octets.

0000	0000	0000	00	00
Format	Instruction	Taille	Registre source	Registre de destination

Format 7 – Mnémonique taille registre immédiate (`0b0010`)

L'utilisation de ce format est similaire à celle des formats 5 et 6. Le codage de cette instruction est

identique au codage du format 5. Toutefois, les champs sont dans le bon ordre.

0000	0000	0000	0000	0000 0000 0000 0000 0000 0000 0000 0000
Format	Instruction	Taille	Registre source	Immédiate (4 octets)

Format 8 – Mnémonique taille immédiate immédiate (`0b0011`)

Ce format est également utilisé comme les formats précédents (5, 6, 7), mais il utilise des valeurs immédiates tant pour la source que la destination. Ces instructions sont longues de 10 octets. Il s'agit du type d'instruction le plus long possible dans cette architecture.

0000	0000	0000	0000 0000 0000 0000 0000 0000	0000 0000 0000 0000 0000 0000
Format	Instruction	Taille	Immédiate source (4 octets)	Immédiate destination (4 octets)

Format 9 – Mnémonique drapeau immédiate (`0b0100`)

On utilise ce format pour des instructions qui font appel à un modificateur d'instruction (drapeau). L'instruction **`JMP`** est un exemple de ce format. Les instructions de ce format sont longues de six octets.

0000	0000	0000 0000	0000 0000 0000 0000 0000 0000 0000 0000
Format	Instruction	Drapeau	Immédiate

Format 10 – Mnémonique drapeau registre (`0b0101`)

Ce format est similaire au format 9. La différence est qu'il utilise un registre plutôt qu'une valeur immédiate. L'instruction **`JMP`** est également disponible dans ce format. Les instructions de ce format sont longues de deux octets.

0000	0000	0000	0000
Form	Instruction	Flag	Immediate

Format 11 – Mnémonique registre registre (`0b1001`)

Il s'agit du dernier format d'instruction disponible sous cette architecture. L'instruction MOV est disponible dans le format 11. Les instructions de ce format sont longues de deux octets.

0000	0000	0000	0000
Format	Instruction	Registre source	Registre de destination

Instructions description

La syntaxe du langage d'assemblage de Capua se fonde sur la notation de destination ou de source.

Lorsqu'une instruction contient deux éléments, qu'il s'agisse de valeurs immédiates ou de registres, le premier correspond toujours à la source et le second, à la destination. La liste complète des instructions est présentée ci-dessous. Après chaque description d'instruction se trouve un tableau présentant les divers formats d'instruction disponibles.

ADD

Cette instruction permet l'addition de nombres entiers. Le tableau ci-dessous présente les diverses variantes de l'instruction d'addition à l'aide d'exemples et d'indication des formats connexes.

Format	Exemple
4	`ADD #0xFF $A`
11	`ADD $B $A`

AND

L'instruction **`AND`** est utilisée pour les opérations binaires. Il s'agit d'une comparaison binaire qui suit la table de vérité **`AND`** normale :

`1 and 1 = 1`

`1 and 0 = 0`

`0 and 1 = 0`

`0 and 0 = 0`

Il est clair que cette vérification est étendue à toute la longueur des éléments comparés (32 bits).

Format	Exemple
4	`AND #0xFF $A`
11	`AND $B $A`

CALL

Il est indispensable de respecter les conditions préalables relatives aux instructions pour utiliser l'instruction **`CALL`**. Dans le cas contraire, des problèmes surviendront (une erreur de mémoire, en général). Avant d'utiliser cette instruction, vous devez vous assurer que le pointeur de pile S enregistre les points dans une zone de la mémoire valide et disponible. L'instruction **`CALL`** entraînera le déplacement de l'adresse qui la suit en haut de la pile de manière à ce que cette adresse soit utilisée comme adresse de retour.

Format	Exemple
3	`CALL #0x40000010`
2	`CALL $A`

CMP

L'instruction de comparaison est la seule qui permet la modification du registre **`FLAGS`**. À l'heure actuelle, le registre **`FLAGS`** n'est large que de trois bits. Le bit supérieur (**`0b100`**) sera configuré lorsque l'élément

source de la comparaison correspondra à l'élément de destination. Le bit central (**0b010**) est configuré lorsque la source est inférieure à la destination et le bit le plus à droite (**0b001**) est configuré lorsque la source est supérieure à la destination.

Format	Exemple
4	`CMP #0xFF $A`
11	`CMP $A $B`

DIV

L'instruction div permet d'effectuer une division. L'élément source sera divisé par l'élément de destination. Le résultat de la division est placé dans le registre **A** et le reste de la division est placé dans le registre **B**.

Format	Exemple
11	`DIV $C $B`

JMP

L'instruction de saut permet de sauter, de manière conditionnelle ou non, des parties du code en consultant la valeur du registre **FLAGS**. Cette instruction est particulière, car elle utilise un drapeau indicateur pour permettre à l'utilisateur de choisir les conditions dans lesquelles le saut devrait avoir lieu. Les conditions possibles sont répertoriées ci-dessous.

<> Le saut est toujours effectué.

<E> ou **<Z>** Le saut est effectué si 0b100 est défini (Drapeau E).

<L> Le saut est effectué si 0b010 est défini (Drapeau L).

<H> Le saut est effectué si 0b001 est défini (Drapeau H).

Il est possible de regrouper ces drapeaux indicateurs pour établir des conditions complexes. Par exemple, l'indicateur **<LE>** pourrait permettre au saut d'avoir lieu si le drapeau E ou le drapeau L est défini. Veuillez noter que la valeur immédiate ou le registre utilisés par cette instruction doit posséder une adresse mémoire valide permettant de récupérer un code. Des problèmes surviennent dans le cas contraire.

Format	Exemple
9	`JMP <E> #0x40000010`
10	`JMP <LH> $B`

JMPR

L'instruction **JMPR** est identique à l'instruction **JMP**, avec une différence majeure, cependant. La valeur immédiate ou le registre utilisé(e) par l'instruction **JMPR** doit posséder un décalage relatif par rapport au registre d'adresse d'instruction I où le saut doit

mener. Par exemple, si on utilise la valeur immédiate **#0xFF**, l'unité d'exécution commencera à lire les instructions situées **0xFF** octets après l'instruction **JMPR**.

Format	Exemple
9	`JMPR <E> #0x10`
10	`JMPR <LH> $B`

MEMR

L'instruction **MEMR** permet l'exécution d'une instruction de lecture de la mémoire. L'instruction **MEMR** utilise un indicateur de taille qui permet à l'utilisateur de lire d'un à quatre octets de la mémoire à un registre. La source peut être un registre ou une valeur immédiate, mais il doit s'agir d'une adresse mémoire valide. On définit l'indicateur de taille à l'aide des caractères **[** et **]**.

Format	Exemple
5	`MEMR [4] #0x40000000 $B`
6	`MEMR [4] $A $B`

MEMW

L'instruction **MEMW** permet l'exécution d'une instruction d'écriture dans la mémoire. À l'instar de l'instruction **MEMR**, elle utilise un indicateur de taille. En raison de sa nature, l'instruction **MEMW** est disponible en un nombre de formats supérieur à

celui de l'instruction **MEMR**. La destination peut être un registre ou une valeur immédiate, mais il doit s'agir d'une adresse mémoire valide.

Format	Exemple
8	`MEMW [4] #0xFF #0x40000000`
5	`MEMW [4] #0xFF $A`
7	`MEMW [4] $B #0x40000000`
6	`MEMW [4] $B $A`

MOV

L'instruction **MOV** permet de déplacer des données entre registres ou de charger une valeur immédiate dans un registre. Cette instruction pourrait être utilisée, par exemple, pour configurer le pointeur de pile **S** dans une zone de la mémoire valide afin de pouvoir utiliser la pile.

Format	Exemple
4	`MOV #0x40000200 $S`
11	`MOV $A $B`

MUL

L'instruction **MUL** permet la multiplication d'entiers entre deux registres. Il n'est pas possible d'utiliser l'instruction **MUL** avec une valeur immédiate. Puisque la multiplication peut résulter en nombres supérieurs à 32 bits, le registre **B** contient les 32 bits supérieurs et le registre **A** contient les 32 bits

inférieurs du nombre obtenu. Le nombre devrait se lire sous la forme **B:A** après la multiplication.

Format	Exemple
11	`MUL $A $B`

NOP

L'instruction **NOP** est l'instruction de non-opération. Elle ne fait rien. Comme Capua n'a pas besoin d'un alignement sur quatre octets, l'instruction **NOP** est habituellement inutile. Toutefois, elle peut être utilisée pour remplir la mémoire au moment du démarrage afin de faciliter le développement.

Format	Exemple
1	`NOP`

NOT

L'instruction **NOT** inversera les bits d'un registre. Par exemple, si le registre **A** est égal à **0x01** avant l'instruction **NOT**, il sera égal à **0xFFFFFFFE** après l'instruction **NOT**. Évidemment, l'instruction NOT ne peut qu'être utilisée avec un registre.

Format	Exemple
2	`NOT $A`

OR

L'instruction **OR** est une opération OU au niveau du bit. Elle travaille suivant la logique ou les règles booléennes. Voici la table de vérité

1 or 1 = 1

0 or 1 = 1

1 or 0 = 1

0 or 0 = 0

Elle aura une incidence sur tous les bits d'un registre.

Form	Example
4	`OR #0xFF $A`
11	`OR $A $B`

POP

L'instruction **POP** enlève 32 bits de la pile et réduit le pointeur de pile de quatre octets vers l'arrière. Les données qui étaient en haut de la pile seront disponibles dans le registre précisé par l'instruction **POP**. Pour que l'instruction **POP** soit utilisée de manière sécuritaire, le pointeur de pile **S** doit être réglé à une adresse mémoire valide.

Form	Example
2	`POP $A`

PUSH

L'instruction **PUSH** ajoutera une valeur de 32 bits sur le dessus de la pile et augmentera le pointeur de pile **S** de quatre octets vers l'avant. Pour que l'instruction **PUSH** soit utilisée de manière sécuritaire, le pointeur de pile **S** doit être établi à une adresse mémoire valide avant d'utiliser l'instruction **PUSH**.

Format	Exemple
3	`PUSH #0xFF`
2	`PUSH $A`

RET

L'instruction de retour est habituellement utilisée pour revenir d'un appel. Elle prendra l'élément en haut de la pile et lui attribuera la valeur du pointeur d'instruction I. Pour utiliser l'instruction **RET**, l'appelant doit s'assurer que le haut de la valeur de la pile est un pointeur vers une région de la mémoire où le code peut être récupéré par l'unité d'exécution. Sinon, il peut y avoir des conséquences fâcheuses.

Format	Exemple
1	`RET`

SHL

La mnémonique d'instruction **SHL** représente un déplacement vers la gauche. Il permet à son utilisateur de déplacer la valeur d'un registre de **X** bits

vers la gauche, où **X** est un chiffre de 1 à 8. Les valeurs excédentaires sont simplement perdues. Par exemple, considérons la valeur **0x80000001**. Le déplacement vers la gauche d'un bit de cette valeur fera en sorte que celle-ci deviendra **0x00000002**

Format	Exemple
4	`SHL #0x01 $A`
11	`SHL $B $A`

SHR

L'instruction **SHR** est la même que l'instruction **SHL**, sauf que le déplacement se produit vers la droite. Ici, le déplacement vers la droite d'un bit de la valeur **0x80000001** produira la valeur **0x40000000**.

Form	Example
4	`SHR #0x01 $A`
11	`SHR $B $A`

SUB

L'instruction **SUB** permet de soustraire des valeurs. Il est important de comprendre que l'opération s'effectue comme suit :

```
destination = destination - source
```

Elle ne s'effectue pas dans le sens contraire. Attention. Il est également important de savoir que l'instruction **SUB** ne prend en compte que les

nombres entiers sous forme de complément à deux, 32 bits, signés.

Format	Exemple
4	`SUB #0x01 $A`
11	`SUB $B $A`

XOR

L'instruction **`XOR`** est une opération au niveau du bit, comme l'instruction **`AND`** et l'instruction **`OR`**. Elle suit la logique normale **`XOR`** (ou exclusif) selon la table de vérité suivante :

`1 xor 1 = 0`

`1 xor 0 = 1`

`0 xor 1 = 1`

`0 xor 0 =0`

Elle peut être utilisée pour régler la valeur d'un registre à 0.

Format	Exemple
4	`XOR #0x01 $A`
11	`XOR $B $A`

Section 2 - Chaîne d'outils

Comme Capua est une architecture, elle a besoin de son propre assembleur. Pour ce projet, on a élaboré trois outils visant à favoriser la mise en œuvre du logiciel et l'essai de la plateforme.

— Assembleur

— Éditeur de liens

— Débogueur

Les sections qui suivent expliquent chaque outil. Veuillez garder à l'esprit que chacun de ces outils a été élaboré, en premier lieu, pour permettre à Capua de faire l'objet d'un essai. Le deuxième objectif de ces outils était de permettre le développement du logiciel. Par conséquent, ces outils comportent des bogues et, en tant que développeur, vous devez les utiliser avec prudence, exactement comme ce texte l'explique. Sachez que ces bogues sont en voie d'être corrigés, mais le fait est que ce projet est d'une telle ampleur et le temps tellement restreint qu'il est impossible de régler tous les problèmes rapidement. Au moment de rédiger ce document, certaines caractéristiques n'ont pas été testées. Le présent manuel a également pour but d'élaborer des directives de développement.

Aperçu du processus de développement

La chaîne d'outils fournie avec Capua permet d'effectuer un processus complet de développement, de l'assemblage du code à son exécution, à l'aide d'un débogueur. Aucun compilateur n'est fourni. L'éditeur de liens permet de relier plusieurs fichiers. Il est important de comprendre que le but de cette chaîne d'outils est de fournir des fichiers binaires plats qui peuvent s'exécuter de manière autonome dans l'environnement Capua. Veuillez noter qu'il est possible d'utiliser la chaîne d'outils pour élaborer des fichiers binaires plus complexes et non autonomes. Cependant, leur élaboration exige un environnement hôte et le fichier binaire généré produira quand même un fichier binaire entièrement relié de façon statique. Le développeur devra donc avoir recours à son imagination pour que le tout fonctionne... C'est toutefois possible.

Aucune expérience en matière d'applications exécutées seules

Les programmeurs qui abordent le développement dans Capua sans expérience des systèmes ou des applications exécutées seules doivent comprendre que le développement dans un environnement simplifié n'est pas exempt d'embûches. Rien n'est prévu pour vous aider. Vous êtes responsable de la gestion de votre propre mémoire et de la gestion

binaire de la mémoire. Capua ne fournit pas de « segments » de quelque forme que ce soit. Si vous faites des erreurs dans l'allocation et la gestion de votre mémoire, vous allez détruire votre propre code en l'exécutant. Par contre, vous pouvez faire preuve d'audace en utilisant, par exemple, plusieurs piles en même temps et en passant des unes aux autres, au besoin. Mais ce n'est pas conseillé! ;)

Assembleur

Aperçu

Capua possède également son propre assembleur. Il est facile à utiliser et le code d'implémentation est plutôt simple. Avant de commencer à utiliser l'assembleur, vous devez être au courant de certains éléments. Tout d'abord, rappelez-vous toujours que la version actuelle de l'assembleur a été écrite comme un outil d'essai pour Capua. L'assembleur est assez perfectionné pour écrire le code et pour l'assembler, mais veillez soigneusement à respecter la syntaxe décrite ici, car les messages d'erreur ne sont pas toujours faciles à comprendre. De plus, dans son état actuel, l'assembleur est très sensible à la typographie. Toutes les parties d'une instruction doivent être séparées par un espace blanc. Cela comprend le commentaire de fin de ligne :

```
MOV $A $B;ce n'est pas assez bien
```

```
MOV $A $B ; c'est bien
```

De toute évidence, un grand nombre de ces problèmes sont en réalité faciles à régler. Sentez-vous libre de participer et de contribuer à l'élaboration du code. Il s'agit, après tout, d'un projet de code source ouvert. Pour l'instant, en raison de la limite de temps disponible, il a été convenu de garder ces petits bogues (petits mais ennuyeux), puisque les directives suivantes permettront au programmeur de contourner les problèmes. En cas de pépin, la meilleure façon de résoudre votre problème est d'examiner le code de près. Le code n'est pas compliqué du tout. La base du code de la chaîne d'outils est, cependant, beaucoup plus problématique que le code VM. Ces outils ont été développés avec l'intention de les remplacer le plus rapidement possible par des outils plus stables. Lors de l'exécution de l'assembleur, le fichier d'entrée est transformé en un fichier *.o*. Veuillez noter que l'extension a été choisie en raison de la signification historique d'un fichier *.o*. Le format de fichier utilisé par la chaîne d'outils Capua a été réalisé (conçu serait exagéré ici) avec l'intention de pouvoir relier plusieurs fichiers pour qu'ils forment un fichier binaire plat. Le format actuel est également très simple à comprendre et constitue un mélange de données XML et binaires.

Format de fichier .o de Capua

Le format de fichier de Capua est très simple. Il existe, toutefois, un bogue associé à ce format de fichier. Le bogue est évident si vous essayez de le chercher. Il aurait pu facilement être évité au moment du développement, mais comme la création de la chaîne d'outils a dû se faire rapidement, ce bogue a été introduit en toute connaissance de cause dans cette version de l'assembleur. Le format de fichier sera modifié plus tard pour éviter ce problème. Le bogue peut être évité par un programmeur prudent. Mais, comme je trouve cela comique, je ne vous dirai pas ce que c'est et vous laisse vous débrouiller à ce sujet;)

Le format lui-même est très simple. Voici le format général :

```
<AssemblySize></AssemblySize>
<ExternalSymbols>
    <refName></refName>
    <refAdd></refAdd>
</ExternalSymbols>
<InternalSymbols>
    <refName>END</refName>
    <refAdd></refAdd>
    ...
</InternalSymbols>
<Text>...</Text>
```

Le tableau suivant explique toutes les parties du format de fichier en détail.

Balise	Explication
AssemblySize	Cette balise contient la taille prévue de la version finale reliée de ce fichier. La taille est encodée en format de stockage binaire gros boutiste à l'intérieur de la balise. Le calcul de taille a été intégré dans l'assembleur puisqu'il était plus facile et plus rapide de le placer là. De plus, l'information est tout de suite accessible à l'éditeur de liens lorsque celui-ci doit calculer l'adresse à l'échelle du fichier parmi plusieurs fichiers qui doivent être reliés.
ExternalSymbols	Cette balise contient une liste de balises (couples refName/refAdd). Les symboles présents dans la liste ExternalSymbols sont considérés comme « globaux » et, par conséquent, sont à la disposition de l'éditeur de liens lorsque celui-ci relie plusieurs objets.
InternalSymbols	Cette balise est la même que ExternalSymbols sauf que les symboles énumérés dans cette balise ne sont pas accessibles à l'éditeur de lien au moment de relier plusieurs fichiers. À l'origine, cela avait pour but de prévenir les problèmes de collision de noms à l'échelle globale (externe) au moment de la liaison. Le processus de liaison a plus tard ajouté le nom de fichier source aux symboles, de sorte que tous les symboles sont disponibles (internes et externes) lorsque l'on utilise le débogueur, tout en évitant la collision.
refName	Une balise refName doit se trouver à l'intérieur d'une balise ExternalSymbols ou d'une balise InternalSymbols. Elle doit également être suivie d'une balise refAdd. La balise refName contient simplement la version texte du nom de symbole. Le nom de symbole est déterminé par la référence/appellation de la mémoire dans le code assembleur écrit par le programmeur.
refAdd	Cela suit une balise refName et indique simplement le décalage où le symbole peut se trouver à partir de l'adresse 0 (par rapport au début du fichier). Notez que le décalage est relatif à un fichier entièrement relié, pas à un fichier objet. Cette adresse remplace éventuellement le nom de symbole lorsque le fichier est relié.

Text	La balise de texte contient le fichier binaire assemblé du fichier objet. Un examen attentif révélera la présence de noms de symboles à l'intérieur de la balise de texte. Ceux-ci sont remplacés au moment de la liaison. Les symboles sont présents dans la section de texte, comme : « :SymbolName: ». Il ne s'agit PAS d'une solution parfaite (ni même d'une bonne solution), mais elle est rapide. (Gardez toujours à l'esprit qu'à l'origine, ces outils ont été écrits pour tester l'unité d'exécution, pas pour écrire un code utilisable).

Et voilà! Le format de fichier est aussi simple que cela.

Écriture du code assembleur

Toutes les instructions ont déjà été traitées dans la section précédente. Par conséquent, cette partie ne portera pas sur le code, mais sur les directives de l'assembleur. Une chose importante à noter et qui n'a pas été explicitement mentionnée jusqu'à maintenant, est que, contrairement à d'autres architectures, le langage assembleur et l'assembleur n'ont pas la capacité d'accéder indirectement à la mémoire. Vous pouvez cependant obtenir un comportement similaire en faisant des calculs de décalage, manuellement, dans le code (amusez-vous bien, oui, c'est pénible). Le tableau suivant répertorie les directives disponibles et montre comment les utiliser.

Directive	Description
NomDeSymbole:	Un nom arbitraire, seul, sur une ligne suivi d'un caractère « : » indique un symbole. L'assembleur l'ajoutera à la liste des symboles et le code sera en mesure de faire référence à NomDeSymbole partout où une valeur immédiate pourrait être utilisée. Notez que pour utiliser le nom de symbole dans le code, vous devez faire précéder le nom par un « : ». Aucun « : » ne serait nécessaire après le nom de symbole lors de l'utilisation du nom dans le code. Par exemple : **`MOV :stackAddress $A`** L'adresse liée de :stackAddress serait mise comme valeur immédiate dans l'instruction mov affichée. On pourrait aussi l'utiliser dans les boucles : **`JMP <> :loopStart`**
.global NomDeSymbole:	Permet à l'assembleur d'ajouter un symbole à la liste des symboles externes. Notez que même si le nom de symbole est suivi du caractère « : », vous devez quand même « déclarer » ce symbole sur sa propre ligne.
;	Le caractère « ; », comme dans beaucoup d'autres langages assembleur, indique un commentaire. Ceux-ci peuvent être soit sur leur propre ligne, soit à la fin d'une ligne, à la suite d'une instruction. Veuillez noter que, dans le cas où un commentaire est mis après une instruction, un espace doit séparer la fin de l'instruction du début du commentaire (le caractère « ; »).
.dataAlpha	On peut l'utiliser n'importe où tant qu'il se trouve sur sa propre ligne. Cette directive est suivie par un espace blanc et un texte au format libre. Le texte n'a pas besoin d'être mis entre guillemets. En fait, il NE DOIT PAS être mis entre guillemets. La chaîne se termine à la fin de la ligne. L'assembleur ajoute un caractère de fin (0x00) à la fin de la chaîne au moment de l'assemblage. Exemple d'utilisation : **`testString:`**

	.dataAlpha Ceci est une chaîne de test Veuillez noter qu'aucun commentaire ne peut suivre cette ligne.
.dataNumeric	C'est la même chose que .dataAlpha sauf que cela permet au programmeur d'utiliser des valeurs numériques de 32 bits. Exemple d'utilisation : **testInt:** **.dataNumeric 0xFFFFFFFF**
$	Est le préfixe de registre. Chaque registre, lorsqu'il est utilisé dans une instruction assembleur, doit être précédé par le caractère « $ ».
#	Est le préfixe de valeur immédiate. Chaque valeur immédiate (sauf lors de l'utilisation d'un symbole) doit être précédée par le caractère « # ». De multiples variantes sont possibles pour les valeurs immédiates : **#0xFF** **#255** **#-1** **#0b11111111**

Exemple de programme court :

Le tableau suivant montre simplement un programme qui calcule la longueur d'une chaîne.

```
; Ceci calcule la longueur de la chaîne de test
.global start:

start:
JMP <> :stackSetup ;Sauter au-dessus de la chaîne

testString:
.dataAlpha Ceci est une chaîne de test

stackSetup:
    MOV :stack $S ;La pile est maintenant utilisable

codeStart:

    PUSH :testString
```

```
    CALL :strlen
    SUB #0x4 $S ;stack reset

end:

;Ci-après vient le calcul de la longueur

;strlen(stringPointer)
strlen:
    MOV $S $A
    SUB #0x4 $A; Calculer le décalage de paramètre
    MEMR [4] $A $A ;Obtenir le paramètre dans le registre
A
    MOV $A $C ;Garder le pointeur au début de la chaîne
lenover:
    MEMR [1] $A $B
    CMP #0x00 $B ;Sommes-nous à la fin de la chaîne?
    JMP <E> :gotlen
    ADD #0x1 $A
    JMP <> : lenover ;Non, pas à la fin, revenir en arrière
gotlen:
    SUB $C $A ;A a la longueur de la chaîne à ce stade.
    RET ;Valeur de retour dans le registre A

; C'est à la fin du programme.
; Aucun risque d'écraser le programme
stack:
```

Aide

Vous pouvez obtenir de l'aide sur l'utilisation de l'assembleur en exécutant l'assembleur avec l'option « -h ».

Éditeur de liens

Aperçu

Capua possède également son propre éditeur de liens. Cet éditeur de liens est destiné à être utilisé pour aider à la création de fichiers binaires plats.

L'éditeur de liens peut relier plusieurs fichiers[1]. Étant donné que les fichiers binaires produits par l'éditeur de liens sont destinés à être utilisés « lorsque l'application est exécutée seule », aucun établissement de liens dynamique n'est disponible. Toutes les adresses, après l'établissement de liens, sont codées en dur dans le fichier binaire résultant. Cela signifie que l'éditeur de liens a besoin de connaître, au moment de l'établissement de liens, l'adresse mémoire où le fichier binaire doit être chargé. Si l'adresse de chargement n'est pas fournie, le fichier binaire lié est chargé à l'adresse MEMORY_START_AT (la valeur par défaut est 0x40000000). Cela va bien pour effectuer des test puisque l'unité d'exécution commence à chercher les instructions à cette adresse.

Attention

Lors de la liaison de plusieurs fichiers, l'ordre dans lequel vous dites à l'éditeur de liens d'insérer les fichiers revêt une IMPORTANCE MAJEURE. Les fichiers sont placés dans le fichier binaire final dans leur ordre d'insertion dans l'éditeur de liens.

Remarque sur les symboles

[1] Au moment de l'écriture, cette fonctionnalité est dans le code, mais doit encore être testée

L'éditeur de liens génère un fichier *« .sym »* dans le même dossier que le fichier binaire final. Ce fichier est simplement une liste de symboles et d'adresses. Ce fichier, le cas échéant, est chargé lorsque vous exécutez le fichier binaire dans le débogueur et vous permet d'utiliser des noms de symbole à la place de l'adresse mémoire lors de l'exécution du fichier binaire dans le contexte du débogueur. Notez que tous les symboles présents dans ce fichier ont été modifiés avec le nom de leur fichier d'origine comme préfixe. Les noms de symbole eux-mêmes sont également mis en majuscules.

Renseignements sur l'utilisation

L'éditeur de liens est facile à utiliser. Pour obtenir des renseignements sur l'utilisation, veuillez exécuter l'éditeur de liens avec l'option « -h ».

Débogueur

Le débogueur est vraiment simple. Il a presque toutes les fonctionnalités de base offertes par la plupart des débogueurs. À l'heure actuelle, il existe trois grandes limitations. La première est que vous ne pouvez pas modifier une valeur de registre ou une valeur de mémoire avec le débogueur. Vous ne pouvez pas non plus « sauter au-dessus » d'un appel de fonction facilement. Pour cela, vous devriez placer un point d'arrêt au retour de l'appel de

fonction. La dernière limitation est que vous ne pouvez pas simplement « recharger » le programme sans relancer le débogueur. Toutes ces caractéristiques sont notées et seront ajoutées à un moment donné à l'avenir. L'utilisation du débogueur est simple et facile. Il suffit d'exécuter le débogueur avec l'option « -h » pour apprendre à le lancer. Une fois lancé, le débogueur s'arrête dès le début de votre fichier binaire. Vous pouvez accéder au menu d'aide du débogueur en tapant « h » ou « help » à l'invite du débogueur.

Section 3 – Matériel mappé en mémoire

La machine virtuelle prend en charge le matériel mappé en mémoire. Le matériel mappé en mémoire est accessible à des adresses mémoire spécifiques. Des adresses différentes ont des significations différentes. Chaque dispositif a un comportement spécifique. Au moment de la rédaction de ce document, seule l'unité de gestion de la mémoire (MMU) était accessible dans la mémoire.

Memory Management Unit

L'unité de gestion de la mémoire n'est pas aussi complexe que ce qu'elle est en général dans de vrais ordinateurs. La MMU de Capua a un seul but : gérer les autorisations d'accès à la mémoire. Ces autorisations peuvent être définies par octet. Aucun segment n'est disponible dans cette architecture.

Utilisation de la MMU

La MMU est mappée à l'adresse **`0x20000000`**. Si des octets sont modifiés à cette adresse, la MMU agit en fonction du contenu des octets modifiés. Le tableau suivant explique la disposition de la mémoire pour la MMU.

Adresse	Longueur (octet)	Objet
`0x20000000`	4	Contient l'adresse mémoire où une modification de l'autorisation aura lieu. Si elle est configurée à la valeur `0x40000000`, cette adresse est là où la MMU commencera à modifier les autorisations.
`0x20000004`	1	Longueur de la modification. Si elle est configurée, par exemple, sur `0xF`, la MMU, lors de la modification de l'autorisation modifiera l'autorisation pour les `0xF` octets suivant l'adresse qui a été écrite à `0x20000000`.
`0x20000005`	5	Le bit le plus significatif de l'octet situé à cette adresse, s'il est défini, provoque, lorsqu'une écriture est effectuée dans la MMU, le déclenchement de l'action de modification de l'autorisation. Les trois bits suivants sont utilisés pour définir les valeurs d'autorisation (lecture, écriture ou exécution). Voici la définition exacte du format de cet octet : `0b1000 0000` – Déclencher l'action de la MMU `0b0100 0000` – Si défini, ajouter l'autorisation de lecture `0b0010 0000` – Si défini, ajouter l'autorisation d'écriture `0b0001 0000` – Si défini, ajouter l'autorisation d'exécution

Veuillez noter que la MMU ne réinitialise pas sa propre mémoire, après le déclenchement d'une action. Le programmeur est responsable de suivre l'état de la MMU.

L'horloge

Capua dispose également d'une horloge. L'horloge peut être utilisée, par exemple, comme source d'entropie. L'horloge est mappée à l'adresse **`0x20000100`**. La lecture de 4 octets à cette adresse permet à l'utilisateur d'obtenir le temps, en secondes, depuis l'époque. L'écriture à des adresses appartenant à l'horloge est interdite. Cela provoque une erreur de mémoire.

Spartacus Thread Multiplexer

En mode de jeu, un dispositif supplémentaire est disponible à l'adresse **`0x20000200`**. Ce dispositif est le Spartacus Thread Multiplexer. Ce dispositif permet à un joueur (disponible uniquement en mode de jeu) de démarrer un nouveau thread, même s'il n'y a pas de système d'exploitation en place pour prendre en charge cette action. Le STM est un dispositif spécial qui a un accès privilégié à l'environnement de jeu (résidant à l'extérieur de la machine virtuelle). Son fonctionnement est très simple. Un joueur qui veut lancer un nouveau thread écrit simplement l'adresse où le thread doit commencer à récupérer le code à l'adresse du STM (**`0x20000200`**). Juste après une écriture à cette adresse, l'environnement de jeu génère un nouveau contexte vide (**`A=0, B=0, C=0, S=0, I=x, FLAGS=0`**. où x est égal à l'adresse écrite à l'adresse du STM). Ce contexte sera alors planifié par l'environnement de jeu (en entrelaçant les threads de

joueurs spécifiques les uns avec les autres). Un utilisateur peut avoir un maximum de 3 threads en cours d'exécution sur le système en même temps. Aucune instruction n'est disponible pour supprimer un thread. On peut supprimer un thread en forçant une exception système. Écrire à l'adresse du STM lorsque l'on n'est pas en mode de jeu ne crée pas une exception. L'écriture sera acceptée dans la mémoire tampon du STM mais aucune action ne sera effectuée. Après la génération d'un nouveau thread, les données écrites à l'adresse du STM y restent jusqu'à la réalisation d'une nouvelle écriture à l'adresse du STM. Le contenu de l'adresse du STM peut également être lu.

Section 4 – Exemple étape par étape

Cette section présente simplement des instructions étape par étape qui utilisent la chaîne d'outils pour générer un fichier binaire exécutable. Le fichier sera ensuite exécuté dans le débogueur.

Étape 1 – Écriture du programme

Pas d'aide ici. Écrivez un programme. Si vous voulez suivre l'exemple, vous pouvez utiliser le programme qui est fourni dans le tableau suivant. Veuillez noter que cet exemple n'utilise pas le registre **S** comme pointeur de pile, mais plutôt comme registre général.

```
; Ceci calcule le nombre de Fibonacci
.global start:

codeStart:
  MOV #0 $A
  MOV #1 $B

fiboStart:
  MOV $A $C
  ADD $B $C
  MOV $B $A
  MOV $C $B
  MOV $A $S ;Le nombre de Fibonacci sera dans $S après
cette opération
  JMP <> :fiboStart
  MOV #1 $A
  MOV #2 $B
end:
```

Enregistrez ce programme dans un fichier.

Étape 2 – Assemblage du fichier

Ici, vous utilisez simplement le script **`Assembler.py`** disponible à la racine de Spartacus afin de générer un fichier **`.o`**. Le tableau suivant montre comment utiliser le script **`Assembler.py`**.

```
C:\Spartacus>python Assembler.py -i fibonaccie.txt
Assembler about to begin, following options will be used
  input file:             fibonaccie.txt
  output file:            fibonaccie.o
Assembler done, output file has been written to fibonaccie.o

C:\Spartacus>
```

Étape 3 – Établissement des liens du fichier

Tout comme pour l'étape d'assemblage, vous appelez ici le script **`Linker.py`** afin de générer le fichier binaire final. Veuillez noter que l'éditeur de liens génère également un fichier **`.sym`**. Vous pouvez vous débarrasser de ce fichier si vous ne voulez pas l'utiliser. Le tableau suivant montre l'établissement des liens du fichier **`.o`** généré précédemment.

```
C:\Spartacus>python Linker.py -i fibonaccie.o -o fibonaccie.bin
Linker about to begin, following options will be used
  input file:             ['fibonaccie.o']
  symbols file:           fibonaccie.sym
  output file:            fibonaccie.bin
Linker done, output file has been written to fibonaccie.bin

C:\Spartacus>
```

Étape 4 – Exécution du fichier binaire dans le débogueur

Vous êtes maintenant prêt à exécuter le fichier binaire en utilisant le débogueur. Cela est facile; il suffit d'utiliser le script **`Debugger.py`**, également disponible à la racine du projet Spartacus. Le tableau suivant montre le fichier **`fibonaccie.bin`** chargé dans le débogueur, prêt à être exécuté.

```
C:\Spartacus>python Debugger.py -i fibonaccie.bin
Debug session about to begin, following options will be used
  input file:            fibonaccie.bin
  symbols file:          fibonaccie.sym
Building Capua execution environment
Loading ('fibonaccie.bin',) in memory
Done loading ('fibonaccie.bin',) into memory
Loading symbols from file fibonaccie.sym
Done loading symbols
Debugging session is ready to be used. Have fun!

Next instruction to be executed:
0x40000000 : MOV #0x0 $A
('fibonaccie.bin',):
```

Étape 5 – Amusez-vous bien!

Et voilà! Pour plus d'aide, veuillez communiquer avec nous ou examiner le code!