FUMERON Rémi
EL HASSANI AMRANI Saad
MERCIER Clément
SIMON Benjamin

3A Electronique
Filière SE
2018 - 2019

Rapport: Projet Avancé

LUAcore

Présentation du projet

Le langage LUA

LUA est un langage de programmation de haut niveau et interprété. Il est caractérisé par son allocation de mémoire ainsi que son typage dynamiques.
On retrouve LUA dans beaucoup d'applications pour l'embarqué, dans le développement réseau et des jeux vidéos.

Parmis les avantages de ce langage:

- Sa légèreté: En effet, le compilateur, l'interpréteur et les librairies standards de Lua n'occupent que peu de place dans la mémoire une fois compilés, beaucoup moins que les autres langages tel que python ou C++.
- Sa rapidité: C'est l'un des langages de programmation interprété le plus rapide.
- Sa simplicité: LUA est connu par sa syntaxe qui est très simple et facile à maîtriser.

Objectifs du projet

Un des intérêts de Lua réside dans sa facilité d'intégration dans d'autres applications. Le problème se situe dans le fait que Lua soit un langage de script interprété ce qui limite ses performances. Afin de palier à ce manque de performances, il existe notamment LuaJIT (*Lua Just-In-Time Compiler*) permettant de compiler les instructions Lua haut niveau vers le langage assembleur bas niveau. Cependant ce compilateur bien que performant n'est pas portable puisqu'il nécessite d'être réécrit pour différentes familles de processeurs.

L'idée est de ce projet est donc de pouvoir compiler du bytecode lua tout en s'affranchissant de dépendances architecturales. Pour cela, on souhaite réaliser un processeur softcore spécialisé via la synthèse de haut niveau HLS. Il doit être capable de lire un bytecode Lua et de l'exécuter.

Démarche de réalisation

Après avoir compris l'objectif principal du projet, une planification et une répartition des tâches était nécessaire entre les membres du groupe.

Les deux principales étapes par laquelle nous avons commencé étaient : l'étude du Bytecode, des instructions et de la machine virtuelle LUA.
On a aussi créé plusieurs programmes LUA afin de constituer un ensemble de référence pour faire des *tests Bench.*

Une fois tout ceci réalisé, on s'est focalisé sur le *parsing* du bytecode et la programmation de notre propre VM.
Pour faire la transition vers le matériel, il a été judicieux de penser aux fonctionnalités et caractéristiques à implémenter. On a établi une liste des instructions qui sont faciles à intégrer, celles qu'on pourrait envisager d'intégrer par la suite et aussi celles que nous jugeons difficile à intégrer.
En se focalisant sur les instructions que nous comptions implémenter, on a établit l'architecture du processeur.

Le test de l'architecture sera fait à travers une VM. Puis potentiellement en systemC pour aller à l'implémentation sur carte FPGA.

Etude du langage LUA

La première étape était donc de comprendre comment le langage LUA s'exécute sur une machine. Pour cela, une étude de la machine virtuelle LUA ainsi que des fichiers binaires utilisés par celle-ci est nécessaire.

Testbench de programmes

Afin de composer un répertoire de référence pour mieux comprendre les comportements du langage LUA, nous avons créé quelques programmes LUA qui manipulent différents types de variables et exécutent plusieurs opérations.
Ensuite à l'aide d'un Makefile adapté, on génère et répertorie les fichiers bytecodes et les fichiers *Listing (Bytecode desassemblé)* associés.
Liste de nos programmes de référence:

- **add.lua**
- **fibo.lua**
- **helloworld.lua**
- **matrix_prod.lua**
- **pi.lua**

Si on prend l'exemple de *fibo.lua*, les fichiers générés par le compilateur LUa *LUAC* de celui-ci seront: *fibo.bin* et *fibo.asm.*

C'est fichiers nous permettent de mieux comprendre le fonctionnement de la machine virtuelle, de ses instructions et nous indiquent comment gérer le Bytecode.

```
main <src/fibo.lua:0,0> (13 instructions, 52 bytes at 0x7fffde2c3870)
0+ params, 8 slots, 0 upvalues, 8 locals, 4 constants, 0 functions
    1    [1] LOADK       0 -1    ; 0
    2    [3] LOADK       1 -2    ; 0
    3    [4] LOADK       2 -2    ; 0
    4    [6] LOADK       3 -3    ; 0
    5    [7] LOADK       4 -4    ; 0
    6    [7] MOVE        5 0
    7    [7] LOADK       6 -2    ; 0
    8    [7] FORPREP     4 3 ; to 12
    9    [8] MOVE        3 2
    10   [9] ADD         2 2 1
    11   [10]    MOVE        1 3
    12   [7] FORLOOP     4 -4    ; to 9
    13   [12]    RETURN      0 1
constants (4) for 0x7fffde2c3870:
    1   0
    2   0
    3   0
    4   0
locals (8) for 0x7fffde2c3870:
    0   N   2   13
    1   fnm1    3   13
    2   fn  4   13
    3   temp    5   13
    4   (for index) 8   13
    5   (for limit) 8   13
    6   (for step)  8   13
    7   i   9   12
upvalues (0) for 0x7fffde2c3870:
```

Exemple d'un fichier asm: ***fibo.asm***

La structure du bytecode LUA

Le bytecode correspondant à un fichier source LUA généré par le compilateur contient toutes les informations nécessaires pour pouvoir produire un comportement voulu. Le bytecode sera un point de départ sur lequel notre processeur ou bien la machine virtuelle va se baser. Donc il est essentiel de bien comprendre comment il est organisé, pour pouvoir en tirer toutes les données utiles.

```
00000000  1B 4C 75 61 51 00 01 04 08 04 04 01 0B 00 00 00  .LuaQ...........
00000010  00 00 00 00 40 68 65 6C 6C 6F 2E 6C 75 61 00 00  ....@hello.lua..
00000020  00 00 00 00 00 00 00 00 00 02 02 05 00 00 00 01  ................
00000030  00 00 00 64 00 00 00 00 00 00 00 47 40 00 00 1E  ...d.......G@...
00000040  00 80 00 02 00 00 00 03 08 00 00 00 04 02 00 00  .Ç.............
00000050  00 00 00 00 62 00 01 00 00 00 00 00 00 00 00 00  .....b..........
00000060  00 00 00 05 00 00 00 05 00 00 00 01 01 00 02 04  ................
00000070  00 00 00 44 00 00 00 4C 00 80 00 47 00 00 00 1E  ...D...L.Ç.G....
00000080  00 80 00 01 00 00 00 04 02 00 00 00 00 00 00 00  .Ç.............
00000090  64 00 00 00 00 00 04 00 00 00 05 00 00 00 05 00  d...............
000000A0  00 00 05 00 00 00 05 00 00 00 01 00 00 00 02 00  ................
000000B0  00 00 00 00 00 00 63 00 00 00 00 00 03 00 00 00  ......c.........
000000C0  01 00 00 00 02 00 00 00 00 00 00 00 61 00 05 00  ............a...
000000D0  00 00 04 00 00 00 05 00 00 00 05 00 00 00 05 00  ................
000000E0  00 00 05 00 00 00 01 00 00 00 02 00 00 00 00 00  ................
000000F0  00 00 61 00 01 00 00 00 04 00 00 00 00 00 00 00  ..a.............
```

Ci-dessus est un exemple de bytecode correspondant au code source suivant :

```
--hello.lua

local a = 8
function b(c) d = a + c end
```

On peut distinguer trois parties dans un bytecode LUA.
La première partie est le « Header ». On y retrouve surtout les normes sur lesquelles le bytecode est écrit.
Le Header comporte 12 bytes qui sont répartis comme suit :

SIGNATURE	VERSION NUMBER	FORMAT VERSION	ENDIANNESS FLAG 0= big endian 1=little endian	SIZE OF INT (IN BYTES) (DEFAULT 4)	SIZE OF SIZE_T (IN BYTES) (DEFAULT 4)	SIZE OF INSTRUCTION (IN BYTES) (DEFAULT 4)	SIZE OF LUA_NUMBER (in bytes) (DEFAULT 8)	INTEGRAL FLAG 0=floating point 1=integral number type
4 bytes	**1 byte**	**1 byte**	**1 byte**	**1 byte**	**1 byte**	**1 byte**	**1 byte**	**1 byte**

Pour l'exemple précédent, on peut découper son Header de la façon suivante :

SIGNATURE	VERSION NUMBER	FORMAT VERSION	ENDIANNESS FLAG 0= big endian 1=little endian	SIZE OF INT (IN BYTES) (DEFAULT 4)	SIZE OF SIZE_T (IN BYTES) (DEFAULT 4)	SIZE OF INSTRUCTION (IN BYTES) (DEFAULT 4)	SIZE OF LUA_NUMBER (in bytes) (DEFAULT 8)	INTEGRAL FLAG 0=floating point 1=integral number type
0X1B4C7561	**0X51** (version 5.1)	**0x00** (official version)	**0x01** (Little endian)	**0x04**	**0x04**	**0x04**	**0x08**	**0x00** (Floating point)

Ensuite on retrouve le « Function Block ». C'est la partie principale d'un bytecode. On y retrouve la liste des constantes utilisées, la liste des instructions utilisées, le nombre de registres utilisés ainsi que d'autres paramètres. Le Function block dans un bytecode est décomposé comme suit :

Source name	**String**
Line defined	**Integer**
Last line defined	**Integer**
Number of upvalues	**1 byte**
Numbe of parameters	**1 byte**
Is_vararg flag	**1 byte**
Maximum stack size (number of registers	**1 byte**
List of instructions	**List**
List of constants	**List**

Finalement on retrouve la liste des « Function Prototypes ». Comme son nom l'indique, c'est une liste de prototypes de fonctions. Chaque élément est composé d'un « Function Block » et donc on retrouvera les mêmes informations que dans la partie précédente mais qui dépendront cette fois des fonctions déclarées dans le fichier source. Dans le cas où aucune fonction n'est déclarée, cette liste sera vide.
Le bytecode contient aussi d'autres éléments optionnels utiles pour le debug.
Maintenant, on va s'intéresser aux formats des listes de constantes et d'instructions :

Constant list

Size of constant list (nombre de constantes)	**Integer**
[Type of Constant	**[1 byte** 0=LUA_TNIL/1=LUA_TBOOLEAN 3=LUA_TNUMBER/4=LUA_TSTRING
Const]	**NOT EXIST/ 1 OR 0 / NUMBER / STRING]**

Le premier élément de la liste détermine le nombre de constantes que celle-ci contient. Ensuite, pour chaque constante, on retrouve 1 byte qui détermine son type puis la valeur de la constante.

Instruction List

Size of Code (nombre d'instructions)	**Integer**
Virtual machine instructions	**[Instruction]**

Comme pour la liste des constantes, le premier élément de la liste des instructions détermine le nombre d'instructions que celle-ci contient. Ensuite on retrouve les instructions les unes après les autres. Leur taille est spécifiée par le Header (par défaut 4 bytes).

La constante de type *String* est un peu particulière au niveau du bytecode.

String Structure

String Data size	**Size_t**
String Data	**Bytes**

Pour chaque chaîne de caractère, on retrouve deux éléments. La longueur de la chaîne puis des bytes représentants les caractères en ASCII.

Exploitation du bytecode

Après avoir analysé et compris la répartition du bytecode, plusieurs programmes ont été créés pour trier et organiser les informations contenues dans ce fichier binaire.

Parser

Tout d'abords, on a choisi d'établir une classe Parser contenant toute les informations disponibles dans le fichier.

```
class Parser{
  private:
    int indiceParser;
  //HEADER//
  unsigned int signature  ;
  unsigned char versionLua         ;
  unsigned char formatVersion      ;
  unsigned char endianness         ;
  unsigned char sizeInt            ;
  unsigned char sizeSize_t         ;
  unsigned char sizeInstruction    ;
  unsigned char sizeLuaNumber      ;
  bool isIntegral                  ;
  //FUNCTION BLOCK//
  FunctionBlock mainFunctionBlock;
  //FUNCTION PROTOTYPES//
  std::vector <FunctionBlock*>  listPrototypeFunction;
```

On remarque la présence d'une instanciation de la classe FunctionBlock qu'on a créée et qui contient comme attributs :

```
class FunctionBlock{

  private:

  Valua          sourceName        ;
  unsigned int   lineDefined       ;
  unsigned int   lasLineDefined    ;
  unsigned char numberOfUpvalues   ;
  unsigned char numberOfParameter ;
  unsigned char isVarargFlag       ;
  unsigned char maxStackSize       ;
  std::vector <unsigned int>  listInstruction ;
  std::vector <Valua>        listConstants    ;
```

Valua est une classe qui représente la valeur d'une variable LUA et donc prend en charge les 4 types présents dans un programme LUA.
Pour revenir au comportement du Parser, il s'agit d'un indice qui parcourt un vecteur d' `unsigned char` représentant le bytecode et charge les valeurs des attributs au fur et à mesure. Pour cela, on a créé deux méthodes de la classe Parser `getUint32()` et `getUint64()` afin de récupérer la bonne valeur de chaque élément selon les spécifications données par le Header.

```
uint32_t getUint32(std::vector<unsigned char> tabCode, int indice); //Bytecode Vector//4bytes index start
uint64_t getUint64(std::vector<unsigned char> tabCode, int indice); //Bytecode Vector//8bytes index start
```

On a créé aussi d'autres méthodes qui vont se baser sur les deux dernières afin de pouvoir parser le Bytecode correctement.

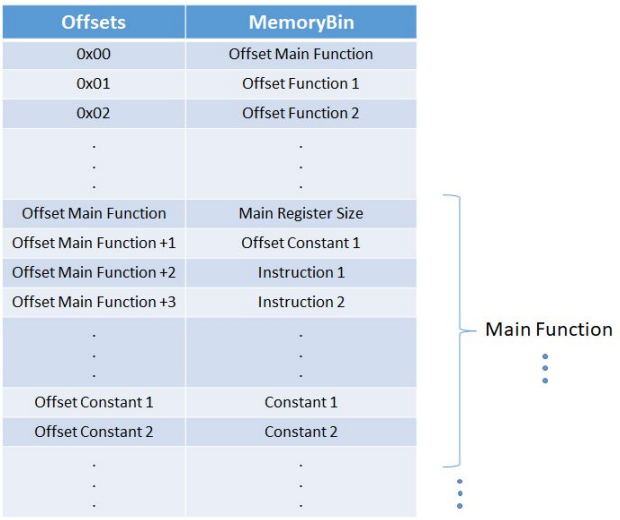
```
ceci est un bytecode lua

HEADER                          //Constant list//  :
Signature          -> 1B4C7561  Constant list size : 2
version Lua        -> 00000051  Const 0 - val = 8256
format Version     -> 00000000  Const 1 - val = b
endianness         -> 00000001
size int           -> 00000004  ---FUNCTION PROTOTYPES---
size Size_t        -> 00000008
size instruction   -> 00000004  sizeOfProtoList: 1
size Lua Number    -> 00000008
Is Integral ?      -> 00000000  Function Prototype -- 0
                                //Instruction list//  :
---MAIN FUCTION BLOCK---         instruction size: 00000004
                                 instruction numero 1 : 44000000
//Instruction list//  :          instruction numero 2 : 4c008000
 instruction size: 00000005      instruction numero 3 : 47000000
 instruction numero 1 : 01000000 instruction numero 4 : 1e008000
 instruction numero 2 : 64000000
 instruction numero 3 : 00000000 //Constant list//  :
 instruction numero 4 : 47400000 Constant list size : 1
 instruction numero 5 : 1e008000 Const 0 - val = d
```

Slicer

Trois informations contenuS dans le Bytecode nous seront nécessaires : la liste des instructions, la liste des constantes ainsi que le nombres de registres utilisés par la fonction. Or le placement dans le fichier Bytecode original de ces informations n'est pas optimisé pour être lu par notre matériel. Le choix a donc été fait de réorganiser les données importantes du

Bytecode dans un nouveau fichier binaire, qui lui sera charger dans la mémoire programme du processeur.
Celui-ci va donc contenir les instructions et les constantes utilisées, ainsi que le nombre de registres. Ces informations seront organisées d'une manière très précise dans le fichier comme le montre la figure ci-dessous.



On commence par positionner les offsets correspondants au début de chaque bloc de fonction. Le début d'un bloc d'une fonction est marqué par le nombre de registre de celle-ci. Ensuite on trouve l'offset de la première constante parmis la liste des constantes de la fonction. Puis on positionne toutes les instructions et après, toutes les constantes.

Pour des raisons de complexité, on a décidé de travailler qu'avec des entiers dans un premier temps. Pour cela, parmis les programmes LUA de tests qu'on a préalablement réalisés, on a choisi de réaliser le travail avec le fichier bytecode add.bin qui correspond au fichier source suivant:

```
--add.bin

local a=1
local b = 2
local c = 3
local d = 4
local e = 5
local f = 6
local g = 7
local h = 8
local i = 9
d = a + 9
```

On obtient un fichier *memorybin* de la forme suivante:

```
00000000  01 00 00 00 09 00 00 00 0E 00 00 00 01 00 00 00  ................
00000010  41 40 00 00 81 80 00 00 C1 C0 00 00 01 01 01 00  A@..üÇ..┴└......
00000020  41 41 01 00 81 81 01 00 C1 C1 01 00 01 02 02 00  AA..üü..┴┴......
00000030  CC 00 42 00 1E 00 80 00 01 00 00 00 02 00 00 00  ╠.B...Ç.........
00000040  03 00 00 00 04 00 00 00 05 00 00 00 06 00 00 00  ................
00000050  07 00 00 00 08 00 00 00 09 00 00 00              ............
```

qui va correspondre au schéma de la mémoire décrit à l'instant.

```
--  MEMORY CONTENT  --
Memory size is 23
 0 :        1
 1 :        9
 2 :        e
 3 :        1
 4 :     4041
 5 :     8081
 6 :     c0c1
 7 :    10101
 8 :    14141
 9 :    18181
10 :    1c1c1
11 :    20201
12 :   4200cc
13 :   80001e
14 :        1
15 :        2
16 :        3
17 :        4
18 :        5
19 :        6
20 :        7
21 :        8
22 :        9
```

On voit alors que la fonction 0 démarre à la position 1, et à la position 1 on lit que la fonction utilise 9 registres pour s'exécuter, et que les constantes commence à l'offset E (14).

La machine virtuelle LUA

Grâce au Parser créé, notre propre machine virtuelle LUA a pu être développée. Celle ci permet plusieurs choses :

- De comprendre le fonctionnement interne du LUA au niveau de ses instructions
- De disposer d'un mode pas à pas permettant de connaître l'état des registres internes de la machine à chaque instant
- D'avoir une référence de comportement pour le futur processeur

Pour la création de ce programme il a fallu coder l'exécution de chaque instructions LUA en C++, ce qui a permis de détecter les points bloquants pour l'implémentation de celle-ci en hardware. On peut citer notamment les instructions permettant de faire de la programmation orienté objet au niveau de la VM.
Pour l'exécution du programme, la VM programmée appelle la fonction du parser pour récupérer la liste des instructions et des constantes du premier *function block*. Le *program counter* s'initialise alors à 0, et l'instruction d'indice 0 s'exécute. Il suffit alors d'incrémenter le PC, ou lui additionner une valeur dans le cas d'un saut, pour connaître la prochaine instruction qui doit être exécutée.

Ayant ainsi connaissance du fonctionnement de la VM Lua, et surtout des différentes instructions qu'elle utilise, l'architecture matérielle du processeur peut être créée.

Architecture du processeur

L'étude de la machine virtuelle Lua a amené plusieurs problèmes, notamment concernant la gestion de la mémoire. Les registres utilisés peuvent contenir 4 types de données : nulle, booléen, chaîne de caractères ,nombre ou encore référence vers tableau contenant d'autres types de données. De plus, certaines instructions sont spécialisés pour gérer l'aspect orienté objet du LUA, chose qui est difficile à réaliser au niveau de la porte logique et du registre. toute ces caractéristiques du Lua amènent donc à faire des choix sur les instructions à implémenter sur le processeur.

Choix des instructions à implémenter

La première étape est donc de décider quelles instructions peuvent être implémentées en matériel. Le premier point bloquant est donc les registres de taille variable ainsi que les références vers des tableaux de données. Il a donc été décidé de ne pas implémenter les chaînes de caractères et les tableaux Lua dans le processeur.
Les variables globales utilisant les chaînes de caractères pour fonctionner, elles n'ont aussi pas été implémentées. Dans un premier temps, les appels de fonction ne sont pas ajoutés au processeur, ce qui simplifie la gestion de la file de registres en enlevant le besoin de créer une pile pour restaurer le PC et les registres de données.

Le set d'instructions choisi pour être implémenté dans la première version du processeur comprend 15 instructions sur les 37 totales de la machine virtuelle Lua complète. Celui-ci est principalement constitué des instructions de déplacement de données (Move, load, … , de calcul (Add, sub, …), de branchement (Eq, Lt, …) et de saut, permettant ainsi la gestion complète de la programmation non fonctionnelle basique.

Mais même avec ce set réduit, ces instructions restent complexes, effectuant de nombreuses opérations en une seul fois que le processeur doit lui aussi effectuer en une période d'horloge.

Instructions Lua		
Opcode	Operande	Description
0	MOVE	Copy a value between registers
1	LOADK	Load a constant into a register
2	LOADBOOL	Load a boolean into a register
12	ADD	Addition operator
13	SUB	Subtraction operator
14	MUL	Multiplication operator
18	UNM	Unary minus operator
19	NOT	Logical NOT operator
22	JMP	Unconditional jump
23	EQ	Equality test
24	LT	Less than test
25	LE	Less than or equal to test
26	TEST	Boolean test, with conditional jump
27	TESTSET	Boolean test, with conditional jump and assignment
31	FORLOOP	Iterate a numeric for loop
32	FORPREP	Initialization for a numeric for loop

Structure du processeur

Pour pallier ce problème, l'idée était de découper chaque instruction Lua en plusieurs microinstructions. Ce type de processeur est alors un CISC *Complex Instruction Set Computer*, c'est-à-dire qu'il contient un microcode décrivant comment chaque macroinstruction doit être exécutée.

Schéma global

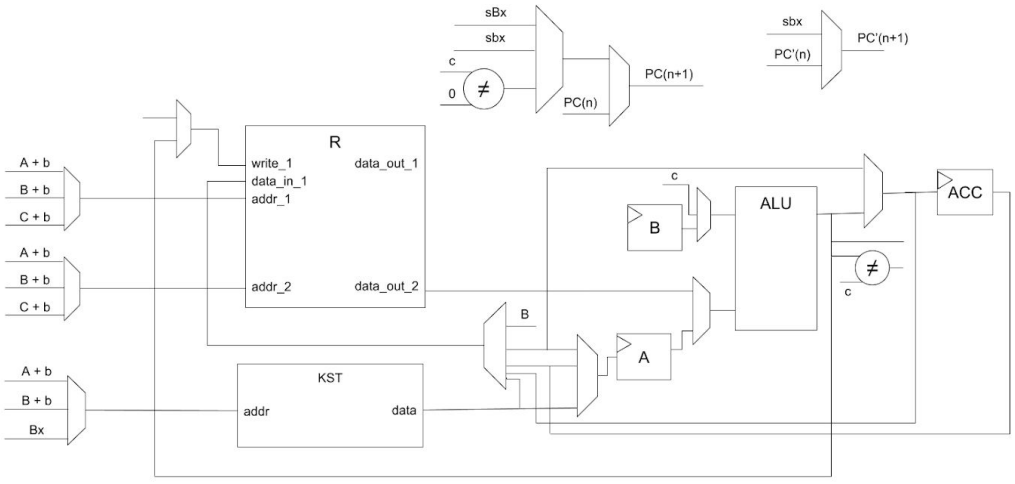
Le processeur possède donc 3 mémoires :

- La mémoire programme, contenant le fichier binaire du Slicer. Il ya donc dans cette mémoire à la fois les macroinstruction et les constantes du programme
- Le microcode du processeur
- La file de registre contenant les données

Ces mémoires sont complétées par un certain nombre de registres de contrôle, qui sont :

- Le PC et le PC', respectivement le compteur programme des macro et des microinstructions
- ALUA et ALUB les registres d'entrées de l'ALU
- ACC un registre stockant le résultat de l'ALU

Le schéma suivant présente l'organisation de ces registres autour de l'ALU



La liste complète des microinstructions utilisées, ainsi que les microcodes correspondant aux instructions Lua est disponible en annexes. L'exécution d'une macroinstruction se fait donc en plusieurs étapes, qui sont détaillées dans la partie suivante.

Fonctionnement du microcode

Comme exemple, la macroinstruction Lua ADD est utilisé. Celle-ci possède trois opérande : A, B et C. Elle récupère les constantes ou les registres d'indice A et B, les additionne pour ensuite stocker le résultat dans le registre C. Le microcode de cette instruction est le suivant :

LALUA	0	0
LALUB	1	0
ALUADD		
SAVEACC	2	0
JMPGLB	1	0

Les deux première microinstruction charge le registre ou la constante d'indice A dans le registre ALUA, et le registre ou la constante d'indice B dans ALUB. La somme est alors calculée par la microinstruction ALUADD, et le résultat est stocké dans le registre ACC. Le contenu du registre ACC est finalement copié dans le registre d'indice C par la microinstruction SAVEACC. La dernière instruction JMPGLB, indique au programme qu'il doit passer à la macroinstruction suivante, donc incrémenter le PC.

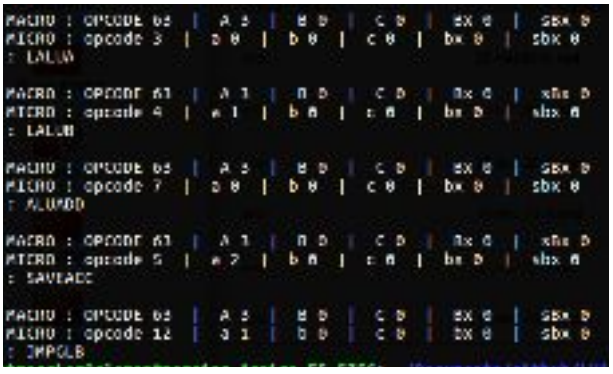
Pour vérifier le bon fonctionnement du microcode, une nouvelle machine virtuelle a été programmée. Celle-ci simule le comportement du processeur, en utilisant le programme venant du Slicer, ainsi que le microcode compilé.

Machine virtuelle microcodée

L'objectif de cette machine virtuelle est donc de simuler le comportement du processeur. Celui-ci fonctionne donc de manière similaire pour le chargement des macro et des microinstructions. Les étapes d'exécution sont les suivante :

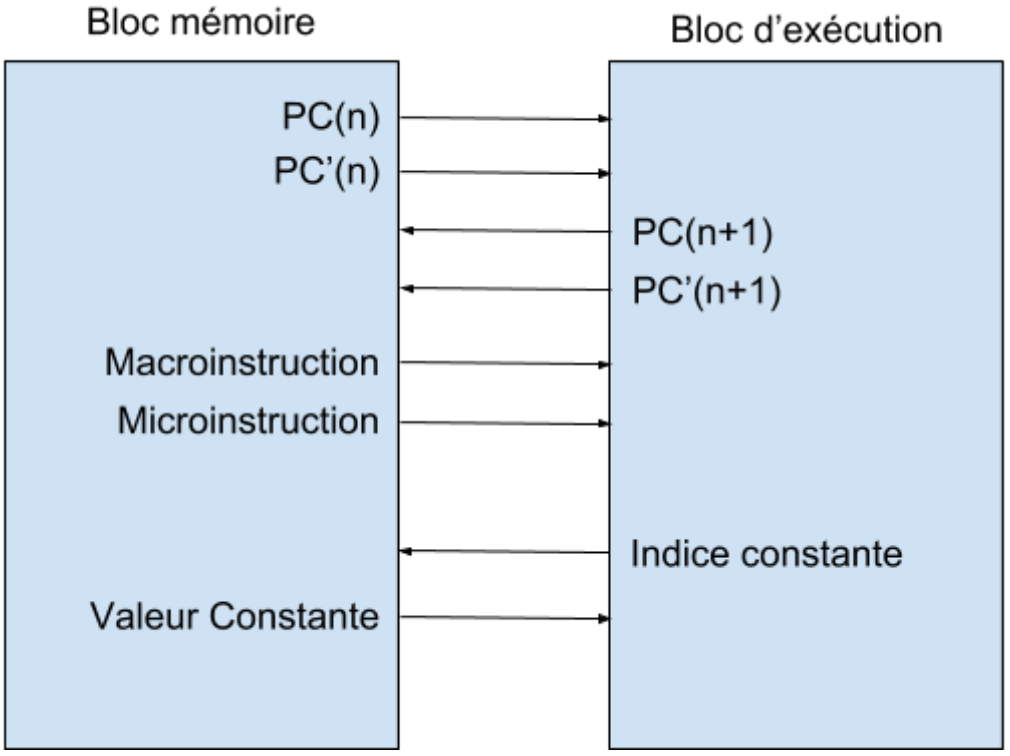
- La macroinstruction est chargée, les opérandes et l'opcode sont lus
- Utilisant l'opcode, l'offset correspondant est lu dans le code du microcode
- Le PC' prend la valeur de l'offset, et la microinstruction correspondante est chargée
- Cette microinstruction est exécutée, puis la suivante est chargée
- Lorsque la microinstruction JMPGLB est lue, le PC est mis à jour.
- La nouvelle macroinstruction est chargée, et le cycle recommence.

On peut voir ci après une capture d'écran de la sortie de la machine virtuelle, pour la macroinstruction ADD



Réalisation matériel du processeur

Comme mentionné précédemment, une synthèse haut niveau à partir d'un code SystemC a été choisi pour réaliser la description matérielle. Pour cela, le processeur est divisé en deux blocs : un blocs contenant la mémoire programme et le microcode, s'occupant de charger les fichiers binaires correspondants, ainsi que le bloc d'exécution qui récupère les valeurs des macro et microinstruction courantes, avec leurs compteurs programme.



Le bloc d'exécution, contenant la file de registre, exécute la microinstruction, en chargeant si besoin une constante depuis le bloc mémoire. Il renvoie ensuite les nouvelles valeurs des PC et PC'. Si le PC a changé, le bloc mémoire va alors actualiser le PC', et envoyer les nouvelles macro et microinstructions. Ce découpage permet ainsi de concentrer les parties de lecture de fichier binaires dans un seul module SystemC, et la communication entre les blocs mémoire et d'exécution se fait alors avec des objets FIFO, simplifiant la simulation du processeur.

Conclusion

Bien que l'idée et l'objectif de ce projet soient très intéressants, le réaliser complètement, étant donné sa complexité, le serait-il aussi d'un point vue pratique? C'est une question rhétorique à laquelle on va s'abstenir de répondre.

Néanmoins, ce fût un projet très instructif pour tous les membres du groupe.
Pour notre dernier projet académique, nous avons essayé d'être le plus autonome possible, cela nous a permis d'approfondir nos compétences dans la matière de gestion de projet. Une gestion qui était menée à bien par notre chef de projet, malgré quelques problèmes d'effectif survenus.

Malheureusement, nous avons pas pu aller jusqu'à l'implémentation de notre architecture sur carte FPGA, un objectif partiel qui était fixé au départ.
En contrepartie, notre machine virtuelle marche correctement pour l'ensemble des instructions qu'on a décidé d'intégrer, ce qui nous indique que pour le set d'instructions restreinte choisi, notre architecture est fonctionnelle.

Les étapes qui nous restaient à réaliser sont :

- Réalisation des blocs en systemC
- Synthèse HLS
- Tests sur carte FPGA
- Elargir *set* d'instructions et l'intégration des appels de fonctions

Annexe
Liste des microinstructions et microcodes

Définition des termes

A : Opérande A de la macroinstruction
B : Opérande B de la macroinstruction
C : Opérande C de la macroinstruction
sBx : Opérande sBx de la macroinstruction
a : Opérande a de la microinstruction
b : Opérande b de la microinstruction
c : Opérande c de la microinstruction
sbx : Opérande sbx de la microinstruction
ACC : Accumulateur de l'ALU
PC : Compteur Programme des macroinstructions
PC' : Compteur Programme des microinstructions

Microinstructions

LOADREG a

R(A+a) := R(B)

Stocke le registre B dans le registre A + a.

LOADCST a

R(A+a) := K(Bx)

Charge la constante d'indice Bx dans le registre A + a.

LOADOPT a

R(A+a) := B; if (C) PC'++

Charge l'operande B dans le registre A + a. Si C, on saute la prochaine microinstruction.

LALUA a b

If (a=0) ALU(A) := RK(A + b)
If (a=1) ALU(A) := RK(B + b)
If (a=2) ALU(A) := ACC

Charge le registre ou la constante B + b, A + b ou l'ACC dans le registre A de l'ALU.

LALUB a b

If (a=0) ALU(B) := RK(A + b)
If (a=1) ALU(B) := RK(B + b)
If (a=2) ALU(B) := ACC

Charge le registre ou la constante B + b, A + b ou l'ACC dans le registre A de l'ALU.

SAVEACC a b

If (a=0) R(A+b) := ACC If (a=1) R(B+b) := ACC If (a=2) R(C+b) := ACC

Charge le registre ou la constante A + b ou B + b dans le registre B de l'ALU.

LOADACC a b

If (a=0) ACC := R(A+b) If (a=1) ACC := R(B+b) If (a=2) ACC := R(C+b)

Charge le registre ou la constante A + b ou B + b dans le registre B de l'ALU.

ALUADD

ACC := ALU(A) + ALU(B)

Effectue l'addition entre ALU(A) et ALU(B) et la stocke dans l'ACC.

ALUSUB

ACC := ALU(A) - ALU(B)

Effectue la soustraction entre ALU(A) et ALU(B) et la stocke dans l'ACC.

ALUMUL

ACC := ALU(A) * ALU(B)

Effectue le produit entre ALU(A) et ALU(B) et le stocke dans l'ACC.

ALUUNM

ACC := - ALU(A)

Calcul l'opposé de ALU(A), et le stocke dans ACC

ALUNOT

ACC := not ALU(A)

Effectue un non logique sur le booléen de ALU(A), et le stocke dans ACC.

JMPGLB a sbx

If (a==1) PC := PC + sbx else PC := PC + sBx

Effectue un saut inconditionnel relatif par rapport à la position actuelle du PC. La valeur du saut est le sbx de la microinstruction s'il est différent de 0, sinon c'est le sBx de la macroinstruction.

JMPLCL a sbx

PC' := PC' + sbx

Effectue un saut inconditionnel relatif par rapport à la position actuelle du PC'.

ALUEQ a

If ((ALU(A) == ALU(B)) ~= (C|a)) PC'++

Test l'égalité entre ALU(A) et ALU(B), si le résultat est égal au booléen C|a, alors le l'instruction suivante du microcode est sautée.

ALULT a

If ((ALU(A) < ALU(B)) ~= (C|a)) PC'++

Test l'égalité entre ALU(A) et ALU(B), si le résultat est égal au booléen C|a, alors le l'instruction suivante du microcode est sautée.

ALULE a

If ((ALU(A) <= ALU(B)) ~= (C|a)) PC'++

Test l'égalité entre ALU(A) et ALU(B), si le résultat est égal au booléen C|a, alors le l'instruction suivante du microcode est sautée.

TST

If not (ALU(A) == C) then PC'++

Test si le booléen ALU(A) est différend de l'opérande booléen C. Si le test est vrai, le PC' s'incrémente.

TSTSET a

If (ALU(A) == C) then R(A + a) := ALU(A) else PC'++

Test si le booléen ALU(A) est égal à l'opérande booléen C. Si le test est vrai, le registre R(A+a) prend la valeur de ALU(A), sinon le PC' s'incrémente.

Microcode

MOVE A B

```
LOADREG   0
JMPGLB    1     0
```

LOADK A B

```
LOADCST  0
JMPGLB   1  0
```

LOADBOOL A B

```
LOADOPT  0
JMPGLB   1  0
JMPGLB   1  1
```

ADD A B C

```
LALUA0   0
LALUB1   0
ALUADD
SAVEACC  2  0
JMPGLB   1  0
```

SUB A B C

```
LALUA0   0
LALUB1   0
ALUSUB
SAVEACC  2  0
JMPGLB   1  0
```

MUL A B C

```
LALUA0   0
LALUB1   0
ALUMUL
SAVEACC  2  0
JMPGLB   1  0
```

UNM A B

```
LALUA0   0
ALUUNM
SAVEACC  0  0
JMPGLB   1  0
```

NOT A B

```
LALUA    0    0
ALUUNM
SAVEACC  0    0
JMPGLB   1    0
```

JMP sBx

```
JMPGLB   0    0
```

EQ A B C

```
LALUA    0    0
LALUB 1       0
ALULT 0
JMPGLB   1    0
JMPGLB   1    1
```

LT A B C

```
LALUA 0       0
LALUB 1       0
ALULT 0
JMPGLB   1    0
JMPGLB   1    1
```

LE A B C

```
LALUA 0       0
LALUB 1       0
ALULE 0
JMPGLB   1    0
JMPGLB   1    1
```

TEST A C

```
LALUA 0       0
TST      0
JMPGLB   1    0
JMPGLB   1    1
```

TESTSET A B C

```
LALUA 1 0
TSTSET 0
JMPGLB 1 0
JMPGLB 1 1
```

FORLOOP A sBx

```
LALUA 0 0
LALUB 0 2
ALUADD
SAVEACC 0 0
LALUA 0 2
ALUUNM
LALUB 2 0
ALULT 1
JMPLCL 0 1
JMPLCL 0 7
LALUA 0 0
LALUB 0 1
ALULE 1
JMPGLB 1 0
LOADACC 0 0
SAVEACC 0 3
JMPGLB 0 0
LALUA 0 1
LALUB 0 0
ALULT 1
JMPGLB 1 0
LOADACC 0 0
SAVEACC 0 3
JMPGLB 0 0
```

FORPREP A sBx

```
LALUA 0 0
LALUB 0 2
ALUSUB
SAVEACC 0 0
JMPGLB 0 0
```