Rapport de projet

# Encodage binaire

Notre processeur utilise des instructions de 32 bits découpées en plusieurs partie. Pour chaque partie, les bits de poids fort sont à gauche.

## Découpage des 32 bits d’instruction

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |
| CONSTANTE | | | | | | | | | | | | | | | | - | Arg2 | | | Arg1 | | | DR/SR | | |  | opcode | | | TYPE | |

correspond à Immédiat ; ‘ - ’ est un bit ignoré.

* TYPE

Indique sur 2 bits quel type d’opération est effectuée. 3 types différents sont codés :

* 00 UAL
* 10 MEM
* 11 CTRL

Les opération UAL sont des calculs effectués par l’unité arithmétique et logique du processeur, le résultat est stocké dans un registre.

Les opérations MEM sont des lectures ou écritures dans la RAM.

Les opérations CTRL sont des opérations impliquant un changement de la valeur du registre PC (Program Counter) telles que des sauts conditionnels et des appels de fonctions.

* opcode

Contient sur 3 bits l’opération à effectuer. La liste des opcode est disponible à la page suivante.

C’est un bit qui indique comment doit être interprétée la constante. Si , la constante est une valeur numérique, si alors la constante est une adresse mémoire ou n’est pas utilisée.

À noter que est ignoré si l’opération codée n’utilise pas la partie constante.

* DR/SR

Une adresse sur 3 bits d’un des 8 registres. Il est utilisé comme destination lors d’un calcul UAL ou d’une lecture RAM, et comme source lors d’une écriture RAM[[1]](#footnote-1).

* Arg1 et Arg2

Les adresses sur 3 bits des registres contenant les valeurs des deux arguments.

À noter que toutes les opérations n’utilisent pas obligatoirement les arguments. Dans ce cas, Les adresses des registres arguments non utilisés sont ignorées.

* CONSTANTE

Un nombre sur 16 bits dont l’interprétation dépend de l’opération codée. Si l’opération codée n’utilise pas la valeur constante, elle est ignorée.

*Liste des opcodes*

UAL

000 ADD

001 SUB

010 AND

011 OR

100 XOR

101 SR

110 SL

111 MUL

MEM

000 LD

001 LDai

010 LDvi

011

100 ST

101 STai

110

111

CTRL

000 JEQU

001 JNEQ

010 JPET

011 JGRA

100 JMP

101

110 CALL

111 RET

## Utilisation des opcodes

* UAL

Les opération UAL utilisent le bit . Si , alors Arg1 et Arg2 sont utilisés comme paramètres (la constante est ignorée) et le résultat est stocké dans DR. Si , Arg2 est ignoré et CONSTANTE est utilisé à sa place.

* MEM

Pour toutes les instructions MEM, Arg2 est ignoré.

* + LD

Charge la valeur à l’adresse RAM contenue dans le registre Arg1 dans le registre DR. doit être égal à 0. La constante est ignorée.

* + LDai

Charge la valeur à l’adresse RAM donnée par la constante dans le registre DR. doit être égal à 0. Arg1 est ignoré.

* + LDvi

Charge la valeur donnée par la constante dans le registre DR. doit être égal à 1. Arg1 est ignoré.

* + ST

Stocke la valeur du registre SR à l’adresse RAM contenue dans le registre Arg1. doit être égal à 0. La constante est ignorée.

* + STai

Stocke la valeur du registre SR à l’adresse RAM donnée par la constante. doit être égal à 0. Arg1 est ignoré.

* CTRL

Pour toutes les opérations CTRL, doit être égal à 0, la constante est l’adresse du saut à effectuer et DR/SR est ignoré.

* + JEQU

Effectue le saut si Arg1 Arg2.

* + JNEQ

Effectue le saut si Arg1 Arg2.

* + JPET

Effectue le saut si Arg1 < Arg2.

* + JGRA

Effectue le saut si Arg1 > Arg2.

* + JMP

Saut à l’adresse donnée par la constante. Arg1 et Arg2 sont ignorés.

* + CALL

Exécute la fonction à l’adresse donnée par la constante. Arg1 et Arg2 sont ignorés.

* + RET

Retourne d’une fonction. Arg1, Arg2 et la constante sont ignorés.

# Utilisation de notre processeur

Programmer le processeur en écrivant directement le code binaire est long, fastidieux et une erreur est facilement faisable alors que trouver une erreur est compliqué. Nous avons donc décidé d’écrire un programme d’assemblage en Java qui transforme le code assembleur en code binaire directement importable dans Logisim.

## Programme d’assemblage Java

Pour utiliser notre programme Java, il faut taper la commande suivante :

java -jar Assembleur.jar fichierIN fichierOUT

fichierIN est le chemin vers le fichier contenant le code source assembleur.

fichierOUT est le chemin vers le fichier qui contiendra le code binaire. S’il n’existe pas, il sera créé. S’il existe déjà, son contenu sera remplacé.

Il affiche dans la console la liste des labels avec l’adresse correspondante ainsi que le code binaire du programme. Ces informations sont inutiles pour l’exécution mais utiles pour débugger le code binaire.

Pour utiliser le programme dans Logisim, il suffit de faire ‘‘clique droit ’’ sur la RAM, puis cliquer sur ‘‘Load Image...’’, puis sélectionner ‘‘fichierOUT’’.

## Syntaxe assembleur

Afin d’utiliser notre programme Java, le code assembleur doit avoir une syntaxe particulière.

Pour accéder aux 8 registres, on utilise R0, R1, …, R7.

* Les opérations UAL

Les opérations UAL doivent avoir la syntaxe suivante :

OP Ra Rb Rc

Où le registre Ra reçoit le résultat de l’opération OP avec Rb = Arg1 et Rc = Arg2.

Les opérations UAL en mode immédiat :

OPi Ra Rb xxxx

Le registre Ra reçoit le résultat de l’opération OP en mode immédiat avec Rb = Arg1 et xxxx est la constante.

Les 8 opérations UAL sont utilisables en mode immédiat.

* Les opérations MEM

LD Ra Rb

Stocke la valeur à l'adresse [Rb] de la RAM dans le registre Ra

LDai Ra xxxx

Stocke la valeur à l'adresse xxxx de la RAM dans le registre Ra

LDvi Ra xxxx

Stocke xxxx dans le registre Ra

ST Ra Rb

Stocke la valeur du registre Ra à l'adresse [Rb] de la RAM

STai Ra xxxx

Stocke la valeur du registre Ra à l'adresse xxxx de la RAM

* Les opérations CTRL

Pour toutes les opérations CTRL, la constante xxxx qui est une adresse peut être remplacée par un label.

JEQU Ra Rb xxxx

Va à l'adresse xxxx si Ra Rb.

JNEQ Ra Rb xxxx

Va à l'adresse xxxx si Ra Rb.

JPET Ra Rb xxxx

Va à l'adresse xxxx si Ra Rb.

JGRA Ra Rb xxxx

Va à l'adresse xxxx si Ra Rb.

JMP xxxx

Saute à l’adresse xxxx.

CALL xxxx

Appelle la fonction à l’adresse xxxx.

RET

Retourne d’une fonction.

Enfin, on peut utiliser une instruction qui n’est pas un opcode : l’instruction STOP. Lors de la traduction en binaire, si l’instruction STOP se situe à l’adresse A, alors elle est remplacée par JMP A.

Pour finir, il y a quelques restrictions :

* Un label ne peut pas être seul sur une ligne,
* Un label ne peut pas avoir le même nom qu’une instruction ou qu’un registre,
* Il ne doit pas avoir d’espace entre les labels et les ‘‘ : ’’,
* Les constantes doivent être écrites en hexadécimal,
* L’assembleur Java prends en compte la casse autant pour les labels que pour les instructions (i.e. ‘‘add R1 R2’’ ou ‘‘SUBi r1 1’’ provoqueront des erreurs d’assemblage).

1. Voir la section syntaxe assembleur. [↑](#footnote-ref-1)