# Architecture du processeur

Pluvinage Lucas, Fehr Mathieu, Dang-nhu Hector, Voss Malachi

January 16, 2017

## 1 Organisation du projet

Le projet est organisé en plusieurs dosseiers :

- as contient un compilateur allant de la syntaxe ARM vers les instructions lues par notre assembleur
- minijazz contient une version modifiée de minijazz
- mjblocks contient les fontions minijazz représentant le processeur
- panda contient un compilateur du langage bamboo (fortement inspiré du basic) vers le langage ARM
- simulateur_netlist contient un compilateur de netlists, pour la simulation
- clock contient des scripts pour éxecuter facilement l'horloge

## 2 Modèle de processeur et architecture globale

Nous avons choisi de beaucoup nous inspirer des processeurs ARM, qui ont l'avantage d'avoir un jeu d'instruction à la fois simple et complet, tout en possédant une certaine efficacité grace aux conditionnement des instructions, permettant de réduire le nombre de jump à effectuer dans les instructions assembleurs.

### 2.1 Mémoires et registres

Le processeur que nous utilisons possède 16 registres, dont un program counter. Il possède une unique ROM addressée sur 16 bits, dont les éléments sont des entiers binaires de 32 bits. Notre processeur n'utilises pas de RAM, bien que les registres soient modélisés sous forme de RAM adressée sous 4 bits. La ROM contient le programme sous forme d'opcodes, générés par du code assembleur ARM.

Le registre rF est le program counter, et pointe sur l'instruction courante. Le registre rE est utilisé par les instructions "branch and link" qui sauvegarde la valeur de rF avant de faire un saut.

### 2.2 Opcodes

Une instruction assembleur est écrite sous 32 bits, et est structurée ainsi :

| 0-3 | 4 | 5 | 6 | 7 | 8-10 | 11 | 12-15 | 16-19 | 20-23 | 24-27 | 28-31 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| COND | 0 | 0 | I | OPCODE | | Set flag | Registre rd | Registre op1 | Opérande op2 | | |
| COND | 0 | 0 | 1 | OPCODE | | Set flag | Registre rd | Registre op1 | Shift | Constante | |
| COND | 0 | 0 | 0 | OPCODE | | Set flag | Registre rd | Registre op1 | Shift | | r2 |
| COND | 1 | 0 | 1 | linking | Offset | | | | | | |
| COND | 1 | 1 | 1 | 0 | Port | | | | Valeur | | |

#### 2.2.1 Flags

A chaque opération, si le booléen set flag est activé ou que l'opération est une opération sans registre de destination, alors les flags sont recalculés :

- N : La dernière instruction renvoie un entier négatif

- Z : La dernière instruction renvoie zéro
- C : La dernière instruction fait un dépassement dans la représentation non signée
- V : La dernière instruction fait un dépassement dans la représentation signée

#### 2.2.2 Conditionnelles

Voici les codes des conditions d'executions des opérations:

| Code | Nom | Condition sur les flags | Signification (pour CMP ou SUB) |
|---|---|---|---|
| 0000 | EQ | Z | Égalité |
| 0001 | NEQ | $\bar{Z}$ | Non égalité |
| 0010 | CS/HS | C | Carry set |
| 0011 | CC/LO | $\bar{C}$ | Carry clear |
| 0100 | MI | N | Négatif |
| 0101 | PL | $\bar{N}$ | Positif ou nul |
| 0110 | VS | V | Overflow signé |
| 0111 | VC | $\bar{V}$ | Pas d'overflow signé |
| 1000 | HI | C and $\bar{Z}$ | Strictement plus grand non signé |
| 1001 | LS | $\bar{C}$ or Z | Plus petit non signé |
| 1010 | GE | N == V | Plus grand signé |
| 1011 | LT | N != V | Strictement plus petit signé |
| 1100 | GT | $\bar{Z}$ and (N == V) | Strictement plus grand signé |
| 1101 | LE | Z or (N != V) | Plus petit signé |
| 1111 | AL | | Toujours exécuté |

# 3 Instructions assembleur

La syntaxe pour effectuer les opérations en assembleur suit la syntaxe spécifiée sur les processeurs ARM. Nous avons créer un compilateur qui permet de compiler un programme ARM vers nos instructions sur 32 bits. Le programme est sité dans le dossier am.

## 3.1 Opérations arithmétiques et booléennes

Voici les opérations assembleur arithmétiques et booléennes :

| Opcode | Nom assembleur | effet |
|---|---|---|
| 0000 | ADD | R[rd] = op1 + op2 |
| 0001 | ADC | R[rd] = op1 + op2 + c |
| 0010 | RSB | R[rd] = op2 - op1 |
| 0011 | RSC | R[rd] = op2 - op1 + c - 1 |
| 0100 | CMP | set les flags sur op1 - op2 |
| 0101 | CMN | set les flags sur op1 + op2 |
| 0110 | SUB | R[rd] = op1 - op2 |
| 0111 | SBC | R[rd] = op1 - op2 + c - 1 |
| 1000 | AND | op1 and op2 |
| 1001 | TST | set les flags sur op1 and op2 |
| 1010 | ORR | op1 or op2 |
| 1011 | BIC | op1 nand op2 |
| 1100 | NOT | not op2 |
| 1101 | MOV | op2 |
| 1110 | EOR | op1 xor op2 |
| 1111 | TEQ | set les flags sur op1 xor op2 |

Où c représente le carry

## 3.2 Instruction de sauts

Deux instructions sont disponibles: branch (B) et branch and link (BL). Branch s'occupe simplement de positionner le program counter (rF) à un autre endroit du code. Branch and link sauvegarde en plus la

valeur du program counter incrémenté (rF+1) dans rE. Cela permet de faire des retours très facilement en faisant un MOV rF, rE.

### 3.3 Instruction spécifique au processeur

L'instruction WAI a été ajoutée. L'opération prends deux paramètres, un port et une valeur sur 12 bits. Le program counter reste bloqué sur l'instruction tant que le port n'envoie pas la valeur indiquée.

Cette instruction à été ajouté pour permettre une synchronisation avec un élément externe au processeur, comme une horloge.

### 3.4 Barrel shifter

Le barrel shifter permet d'effectuer les opérations LSL, ASL, LSR, ASR, ROR et RRX sur l'opérande 2. Il permet aussi de faire un move avec un entier supérieur à 8 bits.

| Opcode | Nom assembleur | effet |
|---|---|---|
| 00 | LSL/ASL | effectue un shift gauche |
| 01 | ROR | effectue une rotation droite |
| 01 | RRX | effectue une rotation droite avec le bit de carry |
| 10 | LSR | effectue un shift droit logique |
| 11 | ASR | effectue un shift gauche arithmétique |

### 3.5 ALU

L'ALU prend en entrée deux valeurs sur 32 bits op1 et op2, le carry sortant de la dernière instruction et l'opcode de l'instruction actuelle. On a réorganisé les opcodes afin d'optimiser les calculs de l'ALU, pour n'avoir qu'un seul adder.

Ainsi les 8 premiers opcode (0XXX) correspondent à des instructions utilisant l'adder. Pour transformer l'adder en soustracteur il suffit par exemple de calculer l'opposé de op1, ce qui se fait en calcul incr(not(op1)) où incr désigne l'incrémenteur. Ainsi on arrive avec un jeu de mux à rediriger les bonnes valeurs dans l'ALU. Les flags sortant correspondent soit au flag de l'adder, soit au flag calculé en sortie des opérations logiques. Dans le cas des opérations logiques, les flags C et V sont toujours à 0, et seuls les flags Z et N sont calculés.

## 4 Utilisation du processeur pour créer une montre

Deux programmes assembleurs ont été écrits pour la montre : un programme simulant une montre en temps réel, et un autre le simulant en temps accéléré (aussi vite que le processeur le permet).

Les deux programmes ne diffèrent que de peu. La seule différence est que l'une attends un signal donné par une horloge pour changer de seconde.

### 4.1 Programme assembleur

Le programme est structuré en plusieurs blocs assembleurs, s'appellant pas système de branching. Il y a un bloc pour les unités de minutes, les dizaines de minutes, les unités d'heures... La seule exception étant pour les secondes, qui sont regroupés en un bloc. Chaque bloc s'occupe d'appeler les blocs d'unités plus faibles autant de fois qu'il le faut. De cette manière, il n'est pas nécessaire de garder en mémoire la valeur des unités.

La date (hormi l'année) et l'heure est stocké dans le registre r0. L'année est elle stockée dans le registre r1. Il faut noter qu'on ne stocke pas les valeurs en elles même, mais les valeurs des digits de chaque chiffre en décimal. Cela permet de relier directement les 2 registres à des afficheurs 7 segments, pour les afficher en temps réel au moment de leur modification.

Pour éviter d'avoir des heures impossibles affichées comme 3h60, notre représentation permet de modifier toutes les valeurs (sauf les années) en une unique instruction. Cependant, pendant quelques cycles, il sera affiché une année antérieure lors du passage à la nouvelle année.

Pour démarer à la date courante, nous créons via le script getdate.py une ram qui représente l'état de la montre à la date donnée. En spécifiant la ram lors de la simulation du procésseur, on démarre donc à la date courante.

## 4.2 Affichage en 7 segment

Les registres r0 et r1 étant relié dans le circuit à des convertisseurs 7 segments, nous avons décidé de les afficher en utilisant la SDL2, qui tourne dans un thread en parallèle au simulateur.