Rapport Virtual Core

SAIFI Sophiane

Implémentation

Je vais suivre l'ordre d'implémentation et donc parler du compilateur en premier, pour ensuite parler du cœur. J'ai codé le compilateur en python, donc j'ai créé des dictionnaires contenant les opcodes / branch codes et leur valeur en décimale. Les fonctions createInstruction_opcode et createInstruction_branch utilisent les bitwise operators pour encoder les valeurs sur 32 bits.

La fonction main quant à elle ouvre un fichier .s contenant des instructions en assembleur-like, les transforme en instructions de 32 bits et les écrit dans un fichier binaire. Pour compiler un fichier entrez cette commande : python3 compiler.py <file.s>

J'ai ensuite codé le cœur en C. J'avais d'abord essayé de lire les instructions 4 octets par 4 octets, mais je me suis rendu compte que cette approche ne séparait pas bien les fonctions fetch, decode et execute. J'ai donc décidé de créer deux structures, instructions_t qui a comme membre l'instruction récupérée par la fonction fetch et les différentes valeurs à decoder d'une instruction 32 bits.

```c
typedef struct instruction_t
{
    uint32_t raw_instr;
    int opcode;
    int dest;
    int op1;
    int op2;
    int iv_flag;
    int iv;
    int num_instr;
    int bcc;
    int offset;
    int flag_offset;
} instruction_t;
```

et la structure core_t qui a comme membre le pc, les valeurs de registres et le carry_flag.

```c
typedef struct core_t
{
    int pc;
    uint64_t registers[16];
    int carry_flag;
    int branch_flag[6];
} core_t;
```

Ces structures permettent donc de séparer les fonctions fetch, decode et execute. Le cœur est composé de 5 fonctions.

- La fonction little_to_big_endian est assez explicite.
- La fonction initialize_registers récupère les valeurs des registres d'un fichier state et les attribue dans le tableau registers de la structure cœur.
- La fonction fetch prend en argument un pointeur de pointeur d'instruction, qu'on peut comprendre comme un tableau dynamique d'instructions. On lit donc le fichier binaire 32 bits par 32 bits et on enregistre ces 32 bits dans le champ raw_instr de la structure instruction_t. On enregistre aussi le nombre d'instructions dans chacune des structures, pour pouvoir y accéder peu importe l'instruction sur laquelle on travaille.
- La fonction decode utilise les bitwise operators pour décoder l'instruction dans chaque champ de la structure.
- La fonction execute est la plus longue. Dans les faits, la fonction fait uniquement un switch case de tous les opcodes. La seule partie qui doit être traitée c'est le cas où l'opcode est CMP, où le cœur regarde l'instruction d'après pour modifier le PC. On utilise le flag_offset pour savoir si le jump est positif ou négatif.

Utilisation

Pour utiliser le cœur, il suffit de compiler le fichier file.s en binaire en utilisant le compilateur, puis d'exécuter le cœur compilé avec le binaire et le state en arguments. J'ai créé un Makefile permettant de lancer le code en 1 seule ligne, avec **make run**, **make run_128** et **make run_lshift**

Programmes

init_test :

Pour init_test, on initialise les registres à 0 et on doit leur attribuer des valeurs arbitraires. Pour ce faire, on fait des left shifts puis on a overlap les bits avec des XOR jusqu'à atteindre la valeur requise.

lshift64_128_test:

Pour le left shift, on fait un right shift de la valeur demandée de taille 64 - 12. 12 ici représente la valeur 0xc donnée initialement. Ensuite on fait un left shift de notre registre à left shift de taille 12. On a donc stocké la taille finale dans deux registres différents.

add128_test:

Pour cet exercice, je pense avoir compris le nécessaire pour qu'il fonctionne, mais mon cœur n'a pas l'air de me donner le bon résultat. Si j'ai bien compris, la retenue se trouve sur les 64 bits de poids faible, donc j'ai pensé à rajouter 1 à l'addition des 64 bits de poids faible. J'ai donc utilisé le carry_flag qui prends la valeur 1 si la valeur après l'addition était plus petite que l'une des opérandes. Malheureusement, ce code n'a pas l'air de fonctionner, alors que pense avoir compris ce qu'il fallait faire.

Questions

1. Which parts of a 64 bits processor are 64 bits wide ?

The parts of a 64-bits processor that are 64 bits are the address bus, the data bus and the registers.

2. Which instructions can potentially create a carry ?

The instructions ADD, SUB, ADC and SBC can create a carry.

3. What is the purpose of the add carry (ADC) instruction ?

C’est l’instruction d’addition avec retenue. Le résultat de l'opération est stocké dans un registre et la retenue est mise à jour pour être utilisée dans la prochaine opération d'addition avec retenue. Ce résultat est égal à l’addition des deux opérandes plus la valeur de la retenue.

4. What are the checks to realize during a branch instruction ?

Il faut d’abord vérifier la condition de branchement, ici le flag de branchement. Ensuite vérifier la valeur de l’offset, ainsi que son signe. Ensuite il faut modifier le PC (program counter) en fonction du signe et la valeur de l’offset.

5. Is it possible to pipeline the virtual core ?

Si j’ai bien compris, pipeline est le fait d'exécuter plusieurs instructions en même temps, et on peut exécuter différentes instructions en même temps. La première idée qui me viendrait en tête serait de faire de la programmation parallèle pour faire on pourrait faire décoder une instruction au cœur pendant qu’il en exécute une autre. Je pense donc qu’il est possible de pipeline ce cœur virtuel.