Ce dépôt inclut notre compilateur, un cross assembleur et le code VHDL de notre processeur.

Fonctionnalités du parser

- Déclaration de variables
- Affectation de variables
- Calculs (addition, soustraction, division, multiplication)
- Fonction main
- Branches (if, else)
- Conditions (!, ||, &&, ==, !=, <, <=, >, >=)
- Boucle while
- printf

Fonctionnalités du compilateur

Toutes les fonctionnalités du parser sauf les boucles while (par manque de temps).

Fonctionnalités du processeur

- Affectation, calculs (sauf division), copie, store, load
- Pipeline à 5 étage
- Gestion des aléas si écriture puis lecture dans le même registre

Installation

Voici les commandes à entrer dans un terminal pour récupérer, compiler et exécuter le projet:

```
# Clone le dépôt git avec les submodules
git clone --recurse-submodules https://git.etud.insa-toulouse.fr/ysimard/projet_systeme_info.git
cd projet_systeme_info
# Compile le compilateur et le cross assembleur
make
# Exécute le compilateur et le cross assembleur sur le fichier test.c
make test_cross_bin.txt
```

Description des dépôts

Compilateur (compilator-2000)

Structure

Ce dépôt contient tous les fichiers nécessaires au fonctionnement du parser et du compilateur, ainsi que des tests unitaires pour la table des symboles. Il contient aussi un interpréteur (dossier `interpreter`), fourni par les professeurs, pour vérifier le code assembleur généré.

En plus des fichiers `al.lex` et `as.y` classiques, ce projet contient plusieurs fichiers `C` utilitaires. `asm_instructions` se charge de modifier le tableau d'instruction assembleur et de l'écrire dans un fichier. `symbol_table` s'occupe de modifier et lire la table des symboles. `symbole_table.test.c` execute des tests unitaires sur le fichier précédent. `yacc_util` est un ensemble de fonctions utilitaires pour simplifier la manipulation de la table des symboles et des instructions assembleur, utilisé par `as.y`.

Compilation et lancement

Un `Makefile`est fourni pour compiler le projet, qui s'utilise comme suit:

- Compiler le compilateur

```
make compilateur
```

- Compiler les tests unitaires

```
make test
```

- Nettoyer le projet

```
make clean
```

Une fois compilé, le compilateur s'exécute de la manière suivante (en remplaçant `INPUT_FILE` par le fichier `C` souhaité et `OUTPUT_FILE` par le fichier destination voulu):

```
./compilateur [OUTPUT_FILE] < INPUT_FILE
```

Pour lancer les tests unitaires, exécuter la commande suivante:

```
./test
```

Pour vérifier le code assembleur à l'aide de l'interpréteur, copier le contenu du fichier de sortie du compilateur dans le fichier `interpreter/input.txt`, compiler l'interpréteur à l'aide de la commande `make`, et exécuter la commande suivante:

```
./interpreter < input.txt
```

Particularités de l'implémentation

Tables

Toutes nos tables sont implémentées sous la forme de tableaux. Nous avons préféré cette approche plutôt que des listes chaînées car nous itérons souvent dessus, ce qui est très lent quand les données sont éparpillées dans la mémoire, comme dans le cas des listes chaînées. Ceci implique cependant que nous avons un nombre d'instructions et un nombre de symboles maximum.

Tests unitaires

Nous avons écrit des tests unitaires pour la table des symboles. Dans ces tests, nous créons une table des symboles, ajoutons des éléments, et vérifions qu'ils sont bien présents après. Nous testons aussi les erreurs retournées par les fonctions.

Fonctions utilitaires

Pour éviter d'avoir un fichier yacc trop imposant, nous avons créé un fichier `yacc_util.c` qui contient toutes les fonctions qui écrivent des instructions dans la table des instructions.

Cross-assembleur (crossassembler-2000)

Le cross-assembleur a été écrit en majorité par Paul Faure. Nous l'avons simplement adapté pour notre compilateur et notre processeur. Il est optimisé car il ne génère par un LOAD et un STORE pour chaque instruction. Il se souvient de quelle donnée est présente dans quel registre.

Compilation et lancement

Compiler

```
make
```

Nettoyer le projet

```
make clean
```

Processeur (processor-2000)

Structure

Ce projet contient un fichier `.vhd` pour chaque composant du processeur, à savoir l'ALU (`ALU.vhd`), la mémoire de données (`data_memory.vhd`), la mémoire d'instructions (`instruction_memory.vhd`), et le banc de registres (`registers.vhd`). Chacun de ces composants possède un fichier de test associé (nommés `COMPOSANT_test.vhd`). Ensuite, tous ces composants sont connectés grâce à une pipeline dans le fichier `CPU.vhd`, possédant lui aussi un fichier de test.

Simulation et tests

Ouvrir le projet dans ISE et lancer la simulation. Le fichier `config_simu.wcfg` contient une configuration du simulateur qui permet de visualiser tous les signaux intéressants.

Particularités de l'implémentation

Multiplexeurs

Les multiplexeurs sont implémentés avec des `std_logic_vector`, pour lesquels chaque booléen correspond à une instruction. Cela permet d'ajouter facilement de nouvelles instructions : il suffit de rajouter à chaque vecteur un booléen.

Tests d'intégration

Nous avons testé les trois composants (compilateur, cross-assembleur et processeur) sur le fichier `test.c`. Nous avons constaté en simulation que la mémoire était bien mise à jour avec les bonnes valeurs. Nous n'avons cependant pas eu le temps de tester sur la carte physique.