Matti Schneider-Ghibaudo, Romain Giraud, Clément Léger
Mr. Lippi
Programmation Système
30/01/09

PROJET : EMULATION DU MICRO-PROCESSEUR PROCSI

Table des matières

I. Présentation du programme

Le sujet de ce projet était de créer un émulateur de PROCSI, micro-processeur étudié en cours d'architecture machine et assembleur. Il s'agit d'un composant assez basique, implémentant la plupart des fonctionnalités standard des processeurs (différents modes d'adressage, indirections…).

Notre rendu est donc constitué d'une version parfaitement fonctionnelle de cet émulateur, mais offre aussi de nombreuses autres fonctionnalités :

- chargement de programmes depuis un fichier source ou un fichier binaire assemblé
- assemblage du code à la volée (en lecture depuis un fichier source)
- désassemblage du code (en lecture depuis un fichier binaire)
- assemblage du code dans un fichier
- calcul des PC pour chacune des instructions avant exécution
- interface utilisant un code couleur facilitant l'utilisation
- gestion de breakpoints à l'exécution
- gestion d'instructions supplémentaires (SHL, SHR, AND, OR…)
- gestion de modes d'adressages supplémentaires pour plusieurs instructions
- affichage dynamique des instructions avant leur exécution ; ainsi, si le programme se modifie lui-même durant son exécution, les instructions réelles sont toujours visibles
- affichage d'informations avancées sur la machine virtuelle (sens d'empilage…)
- système de protection des erreurs fatales telles que adresses mémoires invalides ou encore boucles infinies ("SuperRecover")

Ce jeu de fonctionnalités déjà assez impressionnant en lui-même l'est d'autant plus que le code a été écrit avec en tête la plus grande adaptabilité possible. Vous pourrez ainsi facilement modifier plusieurs paramètres d'affichages (couleurs, niveau de verbosité…) via quelques instructions préprocesseur.

Cependant, cette évolutivité n'a pas empêché de gros efforts pour suivre les principes d'encapsulation. Ainsi, des headers sont disponibles pour chacun des fichiers source (à l'exception du *main*), chacun ne renfermant que les informations strictement nécessaires aux autres fonctions. On peut aussi noter l'absence de toute variable globale, et la définition de types et structures encapsulant la totalité des éléments partagés entre plusieurs composants.

Nous allons à présent voir comment utiliser cet émulateur++.

II. Compilation du programme

Nous avons utilisé l'utilitaire cmake pour générer automatiquement des makefile. Pour pouvoir compiler notre programme, vous aurez besoin de cet utilitaire vous aussi. Vous pouvez télécharger une

version correspondant à votre configuration sur le site officiel s'il n'est pas installé par défaut sur votre système.

Une fois l'utilitaire installé, placez-vous dans le dossier racine du projet, et tapez `cmake .` .

Les makefiles nécessaires ont ainsi été générés. Pour compiler le projet à proprement parler, vous devrez simplement taper :

```
make
```

III. Utilisation du programme

Au lancement

Trois utilisations différentes sont possibles :

- l'exécution d'un fichier binaire : `./procsi BINARY_FILE`

- la compilation et l'exécution d'un fichier source à la volée : `./procsi --source SOURCE_FILE`

- la compilation d'un code source en fichier binaire : `./procsi --compile BINARY_FILE SOURCE_FILE`

La compilation du programme se déroule en plusieurs étapes :

- la première est le parsing du fichier source. Le parser fait plusieurs passes sur la source. En effet, il faut pouvoir connaître les adresses exactes avant l'assemblage (lors d'un JMP par exemple).

- la seconde étape est la création du tableau de mots machine à partir des informations du preprocessing de la source

- finalement, suivant les options, soit on lance la machine virtuelle, soit on exporte le tableau dans un fichier binaire pour une utilisation future.

Description de l'exécution

En premier lieu, le debugger est initialisé. Il récupère le programme (soit d'un fichier source, soit d'un fichier binaire), le charge, puis initialise la machine virtuelle en chargeant le programme dans sa mémoire.

C'est ce debugger qui va s'occuper de la communication entre l'utilisateur et la machine virtuelle. C'est en quelque sorte un mini-shell (on peut par exemple noter que l'historique des appels est disponible à l'aide des flèches directionnelles…). L'utilisateur a à sa disposition de nombreuses fonctions permettant d'analyser le programme et son fonctionnement. Tapez "help" à l'invite de commande pour afficher la liste de commandes disponibles et une description de leur effet.

Description des commandes

Exécuter le programme en entier se fait à l'aide de la commande "run". On peut noter que les instructions désassemblées affichées à chaque pas d'exécution ne sont pas nécessairement celles correspondant à l'appel de "program". En effet, les instructions affichées sont celles réellement exécutées et non celles écrites à l'assemblage ; ainsi, lors d'une boucle, il est possible de répéter

plusieurs fois les mêmes instructions, et on verra une grande différence entre "program" et l'exécution réelle. Un bon exemple est visible à l'exécution du fichier *examples/callret.procsi*.

Quitter le débogueur s'effectue par "quit", ou par la combinaison clavier Ctrl+D.

La commande "program" affiche une version désassemblée du programme. Elle permet aussi d'obtenir l'adresse individuelle de chaque instruction.

Pour n'afficher que la prochaine instruction de la file d'exécution, utilisez la commande "instr" ; pour l'exécuter, tapez "step".

Plutôt que d'exécuter un programme instruction par instruction, il est aussi possible d'interrompre son exécution à l'aide de points d'arrêts placés par la commande "breakpoint" :

Cette commande prend deux arguments :

• "add" ou "rm" pour respectivement ajouter ou supprimer un point d'arrêt.

• le PC (ou adresse d'instruction) où le programme devra être interrompu

• sans arguments, elle se contente de lister les breakpoints déjà définis

Pour vérifier le bon fonctionnement du programme, il est possible d'afficher l'état des registres processeur ou encore de la mémoire à l'aide de la commande "display".

Cette dernière peut avoir plusieurs effets :

• l'affichage d'un registre, se faisait par "reg X" où X un numéro de registre (ou PC, SP, SR pour les registres spéciaux)

• l'affichage d'un mot mémoire, par "mem X", X étant l'adresse mémoire souhaitée

• sans argument, l'affichage de tous les registres de la machine virtuelle

Enfin, pour obtenir des informations détaillées sur la machine virtuelle, vous pouvez utiliser la commande "info".

Réinitialiser la machine virtuelle et recharger le code PROCSI s'effectue à l'aide de "reload". Les registres et la mémoire sont donc effacés et réinitialisés à 0 (cette valeur est définissable dans *sivm.h*).

Les fichiers sources

Les fichiers sources doivent être au format texte. Des commentaires peuvent être ajoutés comme dans tous les langages ; le symbole de début de commentaire est ";", comme vous pouvez le voir dans les fichiers d'exemple fournis (dans le dossier examples). Les opérandes doivent être séparées par des virgules.

Il est intéressant de noter que les labels sont autorisés, ce qui facilite l'utilisation des sauts, qu'on peut ainsi utiliser même sans connaître l'adresse exacte de l'instruction.

Voici les opérations que vous pouvez utiliser dans vos fichiers sources : LOAD, STORE, ADD, SUB, JMP, JEQ, CALL, RET, PUSH, POP, MOV, AND, OR, SHL, SHR, HALT.

IV. Fonctionnalités avancées

1. Utilisation de la commande program

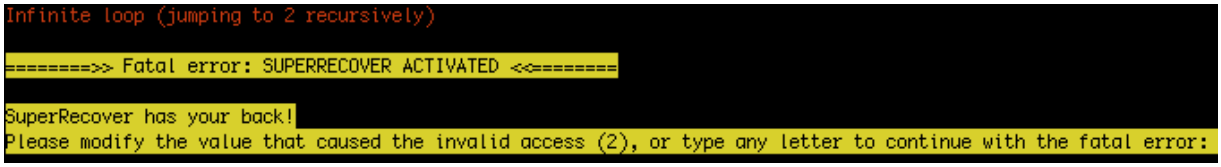
Cette commande est très importante dans la mesure où c'est elle qui affiche le pré-calcul des valeurs du PC avant exécution, ce qui permet donc de repérer les valeurs intéressantes en cas d'activation du SuperRecover.

Elle est également très intéressante de par le fait qu'elle affiche tout le programme chargé sous forme désassemblée, ce même depuis un fichier binaire.

Enfin, on peut noter que les opérandes sont colorées en fonction de leur type (registre, accès indirect…), ce qui facilite la lecture du code désassemblé.

2. SuperRecover

Si un programme arrive, à l'exécution, à certains types d'erreur fatales (accès à une zone mémoire inexistante, boucle infinie…), vous pourrez voir notre système SuperRecover en action :



Vous pouvez exécuter le fichier source *examples/superrecover.procsi* pour une démonstration.

Pensez à afficher program *avant l'exécution, afin de noter le PC de l'instruction HALT, pour pouvoir la choisir comme destination via SuperRecover.*

3. Instructions CALL et RET

Nous avons proposé des implémentations plus puissantes de CALL et RET que les standards qui se contentent d'empiler le PC courant pour y revenir via RET ; en effet, un certain nombre de registres peut être empilé par ces instructions puis dépilé et remis dans les registres correspondants automatiquement. Pour utiliser ces fonctionnalités, modifiez la valeur des constantes PARAM_REGS_START et PARAM_REGS_END. Elles sont par défaut configurées pour recouvrir toute la plage des registres, c'est-à-dire qu'aucun des registres n'est empilé ni dépilé automatiquement (fonctions standards).

Exemple de code assembleur avancé

Le fichier *examples/g33k.procsi* est un bon exemple de code utilisant des instructions complexes. Nous vous invitons à l'exécuter.

V. Limitations actuelles

Nous pourrions notamment étendre ce projet avec un jeu d'instruction plus complet. En effet, nous sommes ici limités à 16 instructions puisque le bitfield "mode" des structures d'instructions ne possède que 4 bits. Cependant, cette limitation provient de la définition de PROCSI et non de notre fait, car ces 4 bits ne peuvent représenter toutes les instructions listées dans les spécifications du micro-processeur.

Nous aurions également aimé ajouter une possibilité d'injecter du code assembleur à la volée dans un programme lors de l'exécution, et sauvegarder ensuite le résultat compilé.

La fonctionnalité SuperRecover pourrait également être généralisée, par exemple dans le cas d'utilisation de registres au-delà des limites. Il serait également intéressant d'autoriser l'utilisation de commandes telles que "info" ou "program" en mode SuperRecover, mais l'implémentation actuelle du système rend ce genre de choses impossible, car la fonction SuperRecover est totalement indépendante de tous les autres jeux d'exécution (elle a d'ailleurs été conçue dans ce but).

La détection de la fin du programme même sans instruction HALT pourrait être très pratique pour éviter les JMP hasardeux.

Enfin, nous aurions aimé pouvoir ajouter des breakpoints en donnant le numéro de ligne (voire un label) plutôt qu'une valeur du registre PC, et une grande partie du code nécessaire à cela est déjà écrite, mais le temps nous a manqué pour finaliser l'implémentation de cette fonctionnalité.

VI. Difficultés rencontrés

Nous n'avons eu qu'un seul véritable problème, qui nous a fait perdre quelques-unes des précieuses dernières heures disponibles : un dépassement de capacité que seul Valgrind a été capable de découvrir, après avoir clamé que "the 'impossible' happened"…

Une autre source d'errances fut l'inclusion récursive de différents fichiers après avoir tenté un refactor du code en extrayant plusieurs définitions de structures du fichier *sivm.h*. Nous avons dû finir par accepter que l'utilisation des typedef rendait la définition externe récursive impossible.

Enfin, nous avons dû mettre à jour plusieurs fois la définition de la structure représentant une instruction, au fur et à mesure que la précision des modes utilisables prenait de l'importance au vu des différents composants réalisés ; cela n'a cependant pas été source de problèmes à proprement parler.