Virtual Core - 17/12/21

CLASSE CYCLE D'INGENIEUR ING3 CYBERSECURITE

A L'INTENTION DE :

contact@bouffard.info et thomas@trouchkine.com

---

réalisé par :
Lilian Naretto, Antoine Delay

Table des matières

1 Le core

Comment l'utiliser : ./core <Binaire> <States> -v (ou sans -v)
Pour plus d'informations, utilisez ./core -h.
Un makefile est mis à disposition, avec la simple commande "make", une génération du code est enclenchée.
Le core est constituer de plusieurs éléments :

1. Enums : BCC et OP.
2. Fonctions principales : fetch, decode, execute, launch.
3. Fonctions secondaires : getprogramsize, readinistate, concat, bitsextractor, endian_swap, in_array.
4. Constantes : MAX64BIT, R_COUNT, CMP_COUNT, DECODE_COUNT.

Les enums permettent de définir les valeurs pour les opérations ainsi que les branches.
MAX64BIT correspond à la taille maximale d'un registre 64 bits.
R_COUNT,CMP_COUNT,DECODE_COUNT définissent la taille des tableaux pour les registres, les instructions et les comparaisons.

La première fonction appellée dans ce core est launch(binaire,states,verbose), qui permet d'initialiser les variables nécessaires, comme le tableau des registres, le PC, etc...
Après initialisation des variables, cette fonction lit le fichier binaire, elle en ressort sa taille (grâce a la fonction getprogramsize(binaire)) et son contenu, dans un tableau de taille définit grâce la fonction précédente, ce tableau correspond aux instructions. Par suite, il y a une lecture du fichier STATES, pour définir les valeurs des registres, qui sont tous initialisés de base à 0.
Le booléen verbose, permet de définir l'affichage du texte.
Après, initialisation, la fonction appelle les 3 autres fonctions principales, dans l'odre : fetch, decode, execute.

1.1 Fetch

La fonction fetch est une fonction qui renvois une nouvelle valeur du PC.
Tout d'abord, elle prend en argument : la valeur du PC actuelle, l'instruction, le tableau de comparaison et enfin la verbose.
En début de fonction, il nous faut lire l'instruction (grâce a la fonnction bitsextrator) , afin de déterminer si il y a BCC, ou non. Si il y a BCC, nous switchons sur la valeur du BCC, puis nous regardons le tableau de comparaison, si les conditions sont remplies, alors la valeur du PC change en fonction du type de branch. Une nouvelle valeur de PC est alors renvoyée.

1.2 Decode

Cette fonction permet de décoder l'instruction prise en argument, elle renvois un tableau regroupant les différents élements nécessaires à l'éxecution.
On utilise la fonction bitsextractor pour ressortir les informations de l'instruction, qui sont ensuite renvoyées sous forme de tableau.

1.3 Execute

Cette fonction de type void prend en argument les différentes informations de l'instruction en cours, ainsi que le tableau des registres, et met ainsi à jour ce tableau avec les nouvelles valeurs des registres.
Nous commençons par set le tableau des comparaisons à 0, par la suite, nous enrigistrons dans uen variable "value" la valeur qui sera utilisé dans l'opération à venir, si il y a un flag, alors on prend l'immediate value, sinon, la deuxieme operande.
Nous switchons alors l'OP pour distinguer l'opération, et ainsi mettre à jour le registre de destination avec sa nouvelle valeur.
Dans le cas d'une OP de type CMP, nous mettons à jour le tableau des comparaisons, qui sera pas la suite lit dpoar le fetch dans la prochaine boucle du PC.

1.4 Fonctions secondaires

1. endian_swap : permet de switch entre du little et du big endian.
2. bitsextractor : permet d'extraire les bits d'une instruction.
3. getprogramsize : ressort la taille du tableau contenant les instructions.
4. concat : concatène 2 ints.
5. readinistate : lit le fichier des états des registres, met à jour le tableau des registres.
6. in_array : check si un char est dans un tableau de char.

2 Le compilateur

Le compilateur est écrit en langage python, il est composé des 3 fichiers suivant :

1. compiler.py
2. interpreter.py
3. m_parser.py

Utiliser le compilateur :
le compilateur peut être utilisé avec la commande suivante :
python3 compiler.py <nom_asm_file>

2.1 Le parseur

fichier : m_parser.py

Le parseur permet de découper les différentes instructions du programmme, il reçoit le fichier programme en entrée et renvoie un tableau de liste, chaque liste contenant une instruction découpé des façon suivante en fonction de l'opération :
Si l'operation fait partie de la liste suivante :
Operation destination, operand_1, operand_2

S'i l'operation est une comparaison (CMP) :
Operation operande_1, operande_2

S'il s'agit d'une affectation (MOV) :
Operation destination, operande_2

S'il s'agit d'un Branchement :
Operation operande_1

Le parseur joue aussi le rôle de nettoyeur puisqu'il va enlever les commentaires, les espaces et les tabulations qui pourrait être mis par le programmeur.
Message d'erreur du parseur :
"not known operation, syntax error" si l'operation n'est pas connu.

2.2 L'interpreteur

fichier : interpreter.py
L'interpreteur va traduire les instructions ainsi decoupé par le parser, en trame de 32 bits

Message d'erreur de l'interpreteur :
"invalid destination register or syntax" si les registres employés ne sont pas bon.

L'interpreteur prend en charge l'écriture hexadecimal des nombres.

3 Les programmes

Programme 1 : Très simple, il nous a suffit de déplacer les valeurs demandées dans les registres différents, au goutte à goutte.

Programme 2 : Il nous a suffit de faire des additions distinctes, pour diviser le resultat en 2 registres, ainsi, à r1 s'ajoute r3, et à r0 s'ajoute r2.

Programme 3 : decaler vers la gauche de 12 bits une valeur dans un registre de 64 bits :
la dificulté du programme consiste à utiliser un deuxième registre pour contenir la partie qui va "sortir" du registre lors du shift.

Le programme 3 peut être fait de plusieurs méthodes différentes, on a choisi d'utiliser une technique similaire au masque de réseaux.
1er étape : dans un registre on écrit un nombre constituer de n, 1 et de 64-n 0, avec n étant le nombre de bits shifté vers la gauche le programme. (dans l'exemple n = 12) Pour cela on utilise les operations ADD, LSH et une boucle.

2eme etape : on applique le masque ainsi créé à notre registre à shifter avec l'operation AND dans un registre différent.

3eme étape : on shift de n bits notre registre initial. on a ainsi dans 2 registres la valeur total.

Programme 4,5,6 : A partir du programmme 4, on n'a pas réussi à obtenir les résultats attendu
le programme 4,5 car sont très similaire et demande de connaître quand est-ce qu'une operation sur un registre va dépasser les 64 bits, c'etait la principale difficulté que nous n'avons pas réussi à surmonter.
Le programme 6 nous a surtout possé problème dans sa comprehension.

4 Les questions

Which parts of a 64 bits processor are 64 bits wide ?
Il s'agit ici des registres, qui sont en 64 bits.

Which instructions can potentially create a carry ?
Les instruction ADD et SUB peuvent créer un carry, dans notre core, ADC et SBC le peuvent également (pour regarder si il y a retenue après ajout de retenue).

What is the purpose of the add carry (ADC) instruction ?
Son but est d'ajouter une retenue précédente dans une addition précédente.

What are the check to realize during a branch instruction ?
Il faut regarder dans le tableau des comparaisons (dans notre core : cmpflags), si la branche de comparaison est valide, si oui, alors on update le PC.

Is it possible to pipeline the virtual core ?
On utilise un coeur entier pour nos opérations, donc non ce n'est pas possible dans notre cas, nous n'avons pas pris en compte le multitâche.