UNIVERSITÉ DU QUÉBEC À MONTRÉAL

PROJET DE SESSION

DANS LE CRADE DU COURS
INF600E - CRÉATION DE LANGAGES INFORMATIQUES

MAI 2021

1. Présentation

Pour mon projet de session j'ai décidé de concevoir à l'aide de SableCC un assembleur pour le microprocesseur 8/16-bits de Western Design W65C816S (qui a comme ancêtre le microprocesseur 6502), notamment utilisée dans la console Super Nintendo.

Je n'ai pas réussi à tout implémenter ce que je voulais mais mon assembleur supporte un quinzaine d'instructions de 65816, les étiquettes (nommées labels dans ce document), des portées (scopes) pour les variable et un autre type pour les labels, plusieurs types d'expressions et d'instructions détaillées plus loin.

La documentation des trois sources suivantes ont été consultées pour la syntaxe des instructions et pour avoir des idées d'instructions. Cependant leur code source n'a pas été consulté.

asar: https://github.com/RPGHacker/asar
xkas-plus: https://github.com/devinacker/xkas-plus
bass: https://github.com/ARM9/bass

2. Structure des fichiers

Le projet comporte des fichiers d'exemples et des fichiers de test. On peut exécuter les test avec le fichier tests.sh à la racine du projet. Lorsque un test est exécuté, il prend le fichier de test dans le répertoire `tests`, l'exécute et compare le fichier binaire de sortie avec le fichier correspondant dans le répertoire `expected`. Ces test ont été fait à mesure que j'avançais dans le projet pour ne pas introduire de nouveaux bugs donc les premiers sont plus simples que les derniers.

Lors de l'exécution d'un ficher source ou des tests, les fichiers résultats sont placés dans le dossier `results`, dossier qui est crée à ce moment s'il n'existe pas déja. En plus de crée un fichier binaire, il y a aussi écriture à la console et dans un fichier texte de l'assemblage du code source. Pour désactiver cette option il suffit de commenter la ligne 257 `writeBinaryToText();` dans la classe `CompilationEngine.java`.

3. Analyse sémantique et interprétation

Mon assembleur a 4 phases. Pour les deux premières phase, soit l'analyse sémantique, j'ai décidé de créer une class GlobalScope, dont une instance est crée pour le main et une à chaque déclaration de macro. Cet instance de GlobalScope contient au moins un VariableScope et optionellement un ou plusieurs scope de label appelé namespace. J'ai décidé de donner leur propre type de scope aux labels et aux variables pour par exemple avoir un scope de variable (e.g. un if) dans un namespace et dans ce if il y aurait usage de label déclaré dans le namespace contenant le if. Je trouvais que cette approche rendait les composantes plus indépendants les uns des autres et cela répondait à ce que je voulais faire.

La première phase crée les scopes, ajoute les macros au MacroTable et ajoute les déclarations de labels au scope courant (ou namespace courant si on est dans ce dernier). La deuxième phase fait tout le restant des vérifications sémantiques. Je me suis inspiré du TP3 pour la vérification sémantique des élément que j'ai pris de celui-ci. Pour les nombres j'ai deux types de nombres: des entiers réguliers, positif ou négatif et ceux-cis sont utilisés dans les variables régulières et conditions des instructions if ou while. L'autre type est utilisé spécifiquement pour les arguments d'instruction (ArgNum) et doit obligatoirement être positif et ne pas dépasser `0xFFFFFF` (entier maximum sur système 24-bit).

Pour les labels, je vérifie qu'un même nom n'est pas déclaré deux fois dans le même namespace ou dans le même global scope et que toutes les utilisations de label sont faite avec un label qui existe. Cependant dans un namespace imbriqué dans un autre, on peut déclarer dans le namespace enfant un label ayant un même nom que dans le parent. Par exemple, un label `x` dans un namespace `b` qui lui est dans un namespace `a` aura comme nom `a-b-x` à l'interprétation. Pour les labels anonymes (+ + ou - -), il y a vérification qu'il n'y a pas deux déclaration ou utilisation de suite et qu'il n'y a pas de + + ou - - traînant qui ne peut être utilisé.

Il y a quelques vérifications sémantiques spécifiques à des instructions, par exemple REP et SEP qui doivent avoir un maximum de un octet d'argument (e.g. `SEP #$20` et non `SEP #$2030`) et qui ne supportent pas les labels (e.g. `SEP label_a`) pour des raisons de ne pas ajouter une phase de plus pusique ces deux instructions ont une incidence sur la longeur de d'autres instructions. Il y a aussi vérification que l'instruction `LDA #const` ou `LDX #const` (e.g. `LDA #$2030`) est sur un maximum de 2 octets. Pour les instruction `MVN` et `MVP`, il y a vérification que les deux arguments ont 1 octet et qu'il n'y a pas d'utilisation de label. Finalement pour les instruction de 2 octets d'argument maximum et qui supporte une longeur explicite (.b, .w ou .l), il y a par exemple vérification que le .l (long 24-bit) n'est pas utilisé pour une instruction de 2 octets d'argument.

La troisième phase est l'interprétation. Chaque instruction est convertie en

tableau de byte qui lui est ajouté à une liste ayant comme clé la combinaison du node de l'instruction et le macro ID courant pour faire une distinction advenant l'appel d'un même macro plusieurs fois. Ce macro ID est incrémenté à chaque fois qu'un appel de macro est fait (le main a un ID de 0). Chaque déclaration de label est ajouté au Frame courant avec le préfix de label courant, le macro ID courant et le offset du label.

La dernière phase a certain éléments d'interprétation mais seulement pour tracer le macro ID courant et le préfix de label courant. Tous les instructions utilisant un label sont alors mises-à-jour avec le offset du label correspondant ou bien dans le cas de branchements il y a calcul du saut négatif ou positif. À la fin de la visite de l'arbre il y a écriture du fichier binaire.

4. Variables

Pour les types, 3 seraient supportés soit int (8-bit à 24-bit), bool (true ou false) et string (ASCII étendu):

```
int var_a = 8 (décimal)
int var_a = $08 (hexadécimal)
int var_a = %1000 (binaire)
bool var_b = true
string var_c = "exemple"
```

5. Opérations

```
int var_a = 8 * 6 (multiplication)
int var_a = 8 / 6 (division entière)
int var_a = 8 MOD 6 (modulo)
int var_a = 8 / 6 (division entière)
int var_a = 8 + 6 (addition)
int var_a = 8 - 6 (soustraction)
int var_a = 8 << 2 (décalage gauche)
int var_a = 8 >> 2 (décalage droite)
int var_a = 8 | 2 (OU)
int var_a = 8 & 2 (ET)
int var_a = 8 ^ 2 (XOR)
int var_a = 8 < 2 (plus petit)
int var_a = 8 <= 2 (plus petit ou égal)
int var_a = 8 == 2 (égal)
int var_a = 8 != 2 (inégal)
int var_a = 8 >= 2 (plus grand ou égal)
int var_a = 8 > 2 (plus grand)
int var_a = {8 > 9} && {9 > 10} ET logique)
int var_a = {8 > 9} || {9 > 10} (OU logique)
bool var_b = !{{8 > 9} || {9 > 10}} (négation)
```

6. Instructions 65816 supportées

https://www.defence-force.org/computing/oric/coding/annexe_2/

Opcode	Hexadécimal	Longeur	Exemple	Exemple (hex)
BEQ	0xF0	2	beq `label_a`	F0 XX
BNE	0xD0	2	`bne label_a`	D0 XX
CMP	0xC5	2	`cmp $15`	C5 15
CMP	0xCD	3	`cmp $1501`	CD 01 15
CMP	0xCF	4	`cmp $150001`	CF 01 00 15
CMP	0xC9	2	`cmp #$80`	C9 80
CMP	0xC9	3	`cmp #$8000`	C9 00 80
DEC	0x3A	1	`dec a`	3A
DEC	0xC6	2	`dec $15`	C6 15
DEC	0xCE	3	`dec $1501`	CE 01 15
INC	0x1A	1	`inc a`	3A
INC	0xE6	2	`inc $15`	E6 15
INC	0xEE	3	`inc $1501`	EE 01 15
JMP	0x4C	3	`jmp $1501`	4C 01 15
JMP	0x5C	4	`jmp $150001`	5C 01 00 15
LDA	0xA5	2	`lda $15`	A5 15
LDA	0xAD	3	`lda $1501`	AD 01 15
LDA	0xAF	4	`lda $150001`	AF 01 00 15
LDA	0xA9	2	`lda #$80`	A9 80
LDA	0xA9	3	`lda #$8000`	A9 00 80
LDA	0xB5	2	`lda $15,x`	B5 15
LDA	0xBD	3	`lda $1501,x`	BD 01 15
LDA	0xBF	4	`lda $150001,x`	BF 01 00 15
LDX	0xA6	2	`ldx $15`	A6 15
LDX	0xAE	3	`ldx $1501`	AE 01 15
LDX	0xA0	2	`ldx #$80`	A0 80
LDX	0xA0	3	`ldx #$8000`	A0 00 80
MVN	0x54	3	`mvn $7E, $7F`	54 7E 7F
MVP	0x44	3	`mvp $7E, $7F`	44 7E 7F
REP	0xC2	2	`rep #$30`	C2 30
RTS	0x60	1	`rts`	60
SEP	0xE2	2	`sep #$20`	E2 20
STA	0x85	2	`sta $15`	85 15
STA	0x8D	3	`sta $1501`	8D 01 15
STA	0x8F	4	`sta $150001`	8F 01 00 15
STA	0x95	2	`sta $15,x`	95 15
STA	0x9D	3	`sta $1501,x`	9D 01 15
STA	0x9F	4	`sta $150001,x`	9F 01 00 15

7. Instructions supportées

Utilisation d'une variable int dans l'instruction:

```
int var_a = 8
lda {var_a + 2}
// lda $0A
```

Utilisation du format décimal ou binaire:

```
lda #16
// lda #$10
lda #%0100
// lda #$04
```

Spécification optionelle de la longeur de l'instruction (byte(.b), word(.w) ou long(.l)):

```
int var_a = $1020
lda.b #{var_a}
// lda #$20
lda.l {var_a}
// lda $001020
```

8. Étiquettes (labels)

Déclaration nécessitant le ":":

```
lda #$04
label_a:
beq label_b
dec
bra label_a
label_b:
rts
```

Puisqu'un label est un nom avec un index (pc), support de l'index avec opération arithmétiques + et -:

```
org($000000)
lda data+{$3000}
sta $1001
rts

org($200000)
data:

// résultat:
lda $203000
sta $1001
rts
```

Labels anonymes (+ + branchement vers le bas, - - branchement vers le haut):

```
01 -- lda $10
02 beq ++    // branchement en 04
03 lda $11
04 ++ sta $12
05 bne --    // branchement en 01
06 -- lda $13
07 lda $14
08 bne --    // branchement en 06
19 lda $15
10 beq ++    // branchement en 12
11 lda $16$
12 ++ sta $17
```

Portée de labels (namespace). Utile pour réutiliser des noms:

```
01 macro macro_a() {
02    label_b:
03    namespace a {
04       label_a:
05       lda $11
06       beq label_a   // branche en 04
07       namespace b {
08          label_a:
09          lda $12
10          beq label_a   // branche en 08
11          beq label_b   // branche en 02
12       }
13       lda $13
14       beq label_a   // branche en 04
15    }
16    beq label_a   // branche en 19
17 }
18
19 label_a:
20 lda $14
21 macro_a();
22 lda $15
```

9. Macros

Macros (mot réservé macro) avec arguments optionels:

```
int var_a = 8
int reg = $1000

macro div_8() {
asl
asl
asl
}

macro div(int num, int register) {
lda #{num}
div_8()
sta {register}
}

div(var_a, reg)
rts

// résultat:
lda #$08
asl
asl
asl
sta $1000
rts
```

10. Conditions et boucles

Condition if:

```
int var_a = 2
bool var_b = true

lda $10
if (var_a >= 2 && var_b) {
    sta $20
}
rts

// résultat:
lda $10
sta $20
rts
```

Boucle while:

```
int var_a = 0
while (var_a < 2) {
    lda {$1000 + var_a}
    sta {$2000 + var_a + 1}
    var_a = var_a + 1
}

// résultat
lda $1000
sta $2001
lda $1001
sta $2002
```

11. Autres

Écriture d'octets (db, dw, dl): Écrit un ou plusieurs octets (à partir de int, variables, expressions ou labels), utile pour ce qui n'est pas du code.

```
db $00, $01
// 00 01

int var_a = $080102
dw var_a
// 02 01

dl {$100000 + $003000}, $2000
// 00 03 10 00 20 00
```

Fonction org(): Met le program counter (pc) à une certaine addresse pour l'assemblage.

```
// les instructions seront assemblées à l'adresse 0x1000 dans le fichier binaire
org($001000)
```

Fonctions hex() et bin(): Converti un int en string pour l'affichage avec print(). Par défault, le format décimal est utilisés si aucune fonction n'est spécifiée.

```
hex(64)
// $40

bin($08)
// %00001000
```

Fonction pc(): Retourne le program counter sous forme de integer.

```
int pc = pc()
```

11. Autres (suite)

Fonctions print() et println(): Écrit à la console.

```
org($100000)
println("pc = " + hex(pc()))
// console: pc = $100000

int var_a = %1000 * %1000
println("var_a = " + var_a)
// console: var_a = 64

print("résultat = " + {2 * 8 + 6})
// résultat = 22
```

Fonction fill(pattern, int num): Écrit le pattern (int, string , variable, expression) num fois (int).

```
fill $FF, 3
// FF, FF, FF

fill {$0100 + $20}, 2
// 20 01 20 01
```