Comment programmer ULg01 ?

Sommaire

Un programme pour la machine β

100000000110010000101000000000000
011001001011000000000000000100100

- Pénible à lire ; pénible à écrire.

- Il faut autre chose :

```
ADD(r3,r4,r5)
ST(r5,0x24,r16)
```

- Pour cela, il faut un programme qui traduise une notation symbolique en une séquence de bits : un assembleur.

Un langage d'assemblage universel

- Plutôt que d'écrire un assembleur spécifique à l'architecture β, nous allons définir quelques mécanismes généraux qui permettent de traiter une grande variété de langages d'assemblage.
- Ce que produit l'assembleur est une séquence d'octets, par exemple

  `0x80   0x64   0x28   0x00`

- Le langage d'entrée de l'assembleur est une séquence d'expressions constantes dont la valeur représente chaque fois un octet, par exemple :

  `128 80+20 2*0x14 0`

  Une constante préfixée par `0x` est interprétée en hexadécimal ; préfixée par `0b` elle est interprétée en binaire ; et, sans préfixe, en décimal.

- On introduit alors quelques mécanismes qui facilitent l'écriture de cette séquence d'expressions.

Les mécanismes du langage d'assemblage

- On permet des définitions de la forme

  identificateur = *expression*

  où un *identificateur* est une chaîne de lettres et de chiffres de longueur quelconque mais commençant par une lettre. Une fois défini, un identificateur peut apparaître dans une expression. On peut alors écrire

```
X = 128
X X-28 2*0x14 X-X
```

- On utilise un *identificateur* spécial "." qui dénote la position de l'octet suivant à ajouter dans la séquence générée. Initialement, "." vaut 0 ; on peut modifier la valeur de ".", par exemple (lire par colonne)

`X = 128`	`.  = .+ 4`
`X X-28`	`2*0x14 X-X`

  laisse un espace de 4 octets a la suite des 2 premiers octets générés.

- On peut affecter une position à un identificateur en écrivant

  identificateur :

  Cela revient à écrire

  identificateur = .

  Exemple :

```
X: .  = .+4
0xA6
```

  réserve (par exemple pour une variable) un espace de 4 octets à une position représentée par l'identificateur `X`.

Un mécanisme puissant : les macros

- Les macros permettent de définir une partie de programme paramétrée qui peut être réutilisée à volonté. La définition d'une macro a la forme

  `.macro` *nom(p1,...,pn) corps*

  où *p1,...,pn* sont les paramètres (formels) de la macro. Ils peuvent être en nombre quelconque. Le *corps* de la macro est un programme (une séquence d'expressions) dans lequel les paramètres formels peuvent apparaître.

- Un appel à une macro s'écrit

  nom(v1,...,vn)

  et est remplacé par le corps de la macro dans lequel les paramètres formels ont été remplacés par les valeur des expressions *v1,...,vn*. Les paramètres sont évalués une fois, au moment où la macro est invoquée.

- Une macro générant quatre valeurs consécutives peut être définie par

```
.macro consec(n) n n+1 n+2 n+3
```

  et donc `consec(6)` est assemblé en 6 7 8 9.

Des macros définissant l'assembleur β

D'abord quelques macros générales utiles.

```
.macro WORD(x)  (x)%256   (x)/256       | little endian
.macro LONG(x)  WORD(x) WORD((x) >> 16)
```

Macros de codage des instructions

```
.macro ENC_NOLIT(OP, Ra, Rb, Rc) LONG((OP<<26)+((Rc%32)<<21)
                                      +((Ra%32)<<16)+((Rb%32)<<11))
.macro ENC_LIT(OP, Ra, Rc, Lit)  LONG((OP<<26)+((Rc%32)<<21)
                                      +((Ra%32)<<16)+(Lit % 0x10000))
.macro ENC_ADRLIT(OP, Ra, Rc, Label)  ENC_LIT(OP, Ra, Rc, (label-(.+4))>>2)
```

Définition des registres

```
r0 = 0b0
...
r31 =  0b11111
```


Instructions arithmétiques et logiques

```
.macro ADD(Ra,Rb,Rc)          ENC_NOLIT(0b100000,Ra,Rb,Rc)
.macro ADDC(Ra,Lit,Rc)        ENC_LIT(0b110000,Ra,Rc,Lit)
.macro SUB(Ra,Rb,Rc)          ENC_NOLIT(0b100001,Ra,Rb,Rc)
.macro SUBC(Ra,Lit,Rc)        ENC_LIT(0b110001,Ra,Rc,Lit)
.macro DIV(Ra,Rb,Rc)          ENC_NOLIT(0b100011,Ra,Rb,Rc)
.macro DIVC(Ra,Lit,Rc)        ENC_LIT(0b110011,Ra,Rc,Lit)
.macro MUL(Ra,Rb,Rc)          ENC_NOLIT(0b100010,Ra,Rb,Rc)
.macro MULC(Ra,Lit,Rc)        ENC_LIT(0b110010,Ra,Rc,Lit)


.macro CMPEQ(Ra,Rb,Rc)        ENC_NOLIT(0b100100,Ra,Rb,Rc)
.macro CMPEQC(Ra,Lit,Rc)      ENC_LIT(0b110100,Ra,Rc,Lit)
.macro CMPLE(Ra,Rb,Rc)        ENC_NOLIT(0b100110,Ra,Rb,Rc)
.macro CMPLEC(Ra,Lit,Rc)      ENC_LIT(0b110110,Ra,Rc,Lit)
.macro CMPLT(Ra,Rb,Rc)        ENC_NOLIT(0b100101,Ra,Rb,Rc)
.macro CMPLTC(Ra,Lit,Rc)      ENC_LIT(0b110101,Ra,Rc,Lit)
```



```
.macro AND(Ra,Rb,Rc)          ENC_NOLIT(0b101000,Ra,Rb,Rc)
.macro ANDC(Ra,Lit,Rc)        ENC_LIT(0b111000,Ra,Rc,Lit)
.macro OR(Ra,Rb,Rc)           ENC_NOLIT(0b101001,Ra,Rb,Rc)
.macro ORC(Ra,Lit,Rc)         ENC_LIT(0b111001,Ra,Rc,Lit)
.macro XOR(Ra,Rb,Rc)          ENC_NOLIT(0b101010,Ra,Rb,Rc)
.macro XORC(Ra,Lit,Rc)        ENC_LIT(0b111010,Ra,Rc,Lit)

.macro SHR(Ra,Rb,Rc)          ENC_NOLIT(0b101101,Ra,Rb,Rc)
.macro SHRC(Ra,Lit,Rc)        ENC_LIT(0b111101,Ra,Rc,Lit)
.macro SHL(Ra,Rb,Rc)          ENC_NOLIT(0b101100,Ra,Rb,Rc)
.macro SHLC(Ra,Lit,Rc)        ENC_LIT(0b111100,Ra,Rc,Lit)
.macro SRA(Ra,Rb,Rc)          ENC_NOLIT(0b101110,Ra,Rb,Rc)
.macro SRAC(Ra,Lit,Rc)        ENC_LIT(0b111110,Ra,Rc,Lit)
```


Instructions d'accès à la mémoire

```
.macro LD(Ra,Lit,Rc)          ENC_LIT(0b011000,Ra,Rc,Lit)
.macro LDR(label,Rc)          ENC_ADRLIT(0b011111,0,Rc,label)
.macro ST(Rc,Lit,Ra)          ENC_LIT(0b011001,Ra,Rc,Lit)
```

Instructions de branchement

```
.macro BNE(Ra,label,Rc)       ENC_ADRLIT(0b011110,Ra,Rc,label)
.macro BT(Ra,label,Rc)        BNE(Ra,label,Rc)
.macro BEQ(Ra,label,Rc)       ENC_ADRLIT(0b011101,Ra,Rc,label)
.macro BF(Ra,label,Rc)        BEQ(Ra,label,Rc)
.macro JMP(Ra,Rc)             ENC_LIT(0b011011,Ra,Rc,0)
```

Définir de nouvelles instructions à l'aide de macros

Exemples

```
.macro MOVE(Ra,Rc)      ADD(Ra,r31,Rc)    | R[Rc] <- R[Ra]
.macro CMOVE(C,Rc)      ADDC(r31,C,Rc)    | R[Rc] <- C

.macro NOP()            ADDC(r31,r31,r31) | ne rien faire

.macro BR(label)        BEQ(r31,label,r31)
```


Un programme en assembleur β

```
.include macros          | Chargement des macros
       LDR(input,r0)     | données dans r0
       BR(bitrev)
input: LONG(0x12345)

bitrev:
       CMOVE(32,r2)     | 32 bits à traiter
       CMOVE(0,r1)      | registre de résultat
loop:  ANDC(r0,1,r3)    | le premier bit dans r3
       SHLC(r1,1,r1)    | le transférer
       OR(r3,r1,r1)     |  dans r1
       SHRC(r0,1,r0)    | preparer bit suivant
       SUBC(r2,1,r2)    | decompter bits à traiter
       BNE(r2,loop)     | boucler si pas fini
```


La machine β et les langages de haut niveau

Nous allons voir que la machine β permet d'implémenter les concepts de programmation utilisés dans les langages de haut niveau et en particulier les appels de procédures.

Considérons un exemple.

```
int fact(int n)
{
    if (n>0)
        return n*fact(n-1);
    else
        return 1;
}

fact(4);
```


Pour gérer des procédures, il faut trouver une solution aux problèmes suivants :

1. passer des arguments à la procédure,

2. permettre à la procédure de disposer de variables locales,

3. permettre à la procédure de renvoyer une valeur,

4. permettre à la procédure d'appeler d'autres procédures, y compris elle-même (récursion).

Sauver l'adresse de retour

- Pour sauver l'adresse de retour, il est assez naturel d'utiliser un registre. Le registre r28 (Linkage Pointer, LP) sera réservé à cet effet.
- On peut donc appeler une procédure par `BR(label,LP)`.
- Et le retour d'une procedure se fait par `JMP(LP,r31)`.
- Mais comment gérer les arguments et la récursion ?

Gérer une pile dans β

- Une pile est un concept très utile pour gérer les procédures ainsi que d'autres problèmes de programmation.
- Une pile est une zone de mémoire de taille variable pour laquelle on n'a un accès direct qu'au dernier élément inséré.
- Nous implémenterons une pile sous la forme d'une séquence de mots mémoire et nous utiliserons un registre dédicacé (Stack Pointer, `SP = r29`) comme pointeur de pile.

- Ajouter ou enlever un élément à la pile se fera alors à l'aide des instructions suivantes.

```
.macro PUSH(Ra)         ADDC(SP,4,SP) ST(Ra,-4,SP)
.macro POP(Ra)          LD(SP,-4,Ra)  ADDC(SP,-4,SP)
```

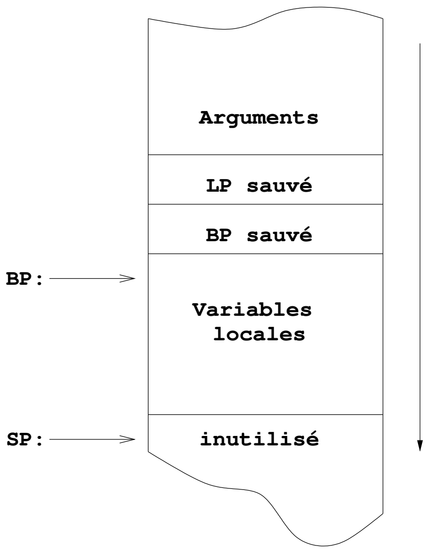
- Réserver de l'espace sur la pile peut se faire comme suit.

```
.macro ALLOCATE(k)      ADDC(SP,4*k,SP)
.macro DEALLOCATE(k)    SUBC(SP,4*k,SP)
```


Gérer les procédures avec une pile

- Les arguments sont mis sur la pile avant l'appel à une procédure.
- La procédure appelée utilise la zone de la pile au delà des arguments pour ses besoins propres.
- La zone de la pile utilisée par une procédure s'appelle son *cadre* (*frame*). Il est intéressant d'avoir un pointeur qui indique le début de ce cadre (Base of frame Pointer, `BP = r27`).
- Pour permettre la récursion, on sauve LP et BP sur la pile à chaque appel de procédure.

Organisation générale de le pile



Implémentation des appels de procédure

Dans la procédure appelante

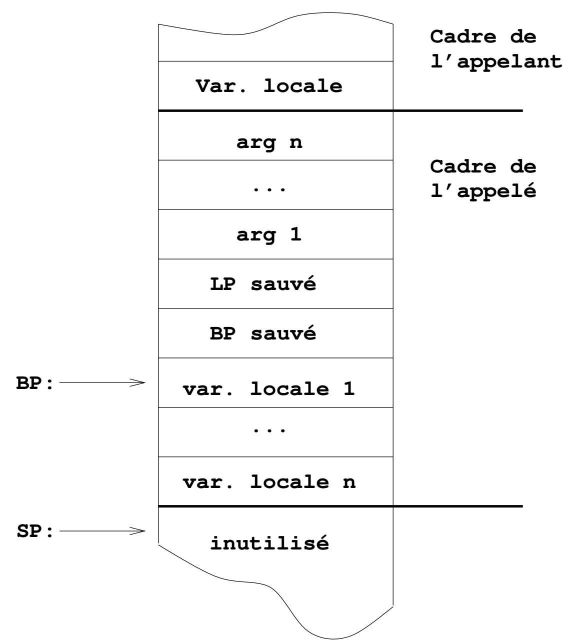
```
PUSH(argn)      | les arguments sont placés en ordre
...             | inverse
PUSH(arg1)
BR(f,LP)
DEALLOCATE(n)
```

Au début de la procédure appelée

```
f:  PUSH(LP)
    PUSH(BP)
    MOVE(SP,BP)
    ALLOCATE(espace) | allocation d'espace local
    (sauver autres registres si nécessaire)
```


A la fin de la procédure appelée

```
(restaurer les registres sauvés)
MOVE(val,r0) | une valeur de retour est placée en r0
MOVE(BP,SP)   | restaurer le SP de l'appelant
POP(BP)       | restaurer le BP de l'appelant
POP(LP)       | restaurer l'adresse de retour
JMP(LP,R31)   | retour
```

L'accès aux variables locales

Pour accéder à la i-ème variable locale, on utilise

```
LD(BP,(i-1)*4,rx)

ST(rx,(i-1)*4,BP)
```

Pour accèder au j-ème argument, on utilise

```
LD(BP,-4*(j+2),rx)

ST(rx,-4*(j+2),BP)
```


L'implémentation de la factorielle

```
fact: PUSH(LP)          | sauver l'adresse de retour
      PUSH(BP)          | sauver le cadre précédent
      MOVE(SP,BP)       | initialiser le cadre courant
      PUSH(r1)          | r1 sera utilisé, le sauver
      LD(BP,-12,r1)     | charger l'argument n en r1
      BNE(r1,big)       | comparer n à 0
      ADDC(r31,1,r0)    | n=0, renvoyer 1
      BR(rtn)           | aller à la séquence de retour

big:  SUBC(r1,1,r1)     | calculer n-1 en r1
      PUSH(r1)          | mettre l'argument sur la pile
      BR(fact,LP)       | appel récursif
      DEALLOCATE(1)     | libérer l'espace des arguments
      LD(BP,-12,r1)     | charger n en r1
      MUL(r1,r0,r0)     | n*fact(n-1) en r0
```



```
rtn:  POP(r1)           | restaurer r1
      MOVE(BP,SP)       | restaurer le SP
      POP(BP)           | restaurer le BP
      POP(LP)           | restaurer l'adresse de retour
      JMP(LP,r31)       | retrour
```


Variables non locales

- La technique vue jusqu'à présent ne permet pas de traiter les variables non locales.
- Pour ce faire, on utilise des "liens statiques".
- Un lien statique pointe du cadre de pile courant vers le cadre de pile de la procédure dans laquelle la procédure courante a été lexicalement définie.
- On accède alors à une variable non locale en remontant le nombre nécessaire de liens statiques et en accédant ensuite à une variable locale.