Tutoriel TI-Z80

Un tutoriel pour programmer en assembleur sur TI-Z80. Les exemples sont montrés pour la TI-83 avec Venus et la TI-83+ avec ION ou MirageOs. Ce tutoriel ne se veut pas exhaustif, et des références sont données vers d'autres ressources.

Outils requis

Il vous faut: un ordinateur sous Windows XP/Vista ou Linux ou Mac Os X. Un éditeur de texte de votre choix.

Compiler

Le compileur SPASM (lien), le signeur Wabbit (lien), ce script de compilation (lien).

Emuler

Ensuite, un émulateur: selon votre plate-forme, si vous êtes sous Windows ou Mac OS, prenez WabbitEmu (lien). Si vous êtes sous Linux, prenez TilEm (lien). Vous allez passer beaucoup plus de temps à tester vos programmes sur émulateur que sur vraie calculette, c'est nettement plus pratique.

- Récupérer ROM
- Récupérer shell (de ticalc.org)
- Envoyer un programme à l'émulateur (shell, jeu)
- Les états: revenir à l'état enregistré

Tester en vrai

Pour tester sur une véritable calculette, il vous faut un cable PC-TI ainsi qu'un logiciel de transmission de programmes via ce cable: TILP fera l'affaire. (plateformes?). Je vous laisse lire la documentation. C'est moins important de savoir utiliser TILP que d'utiliser un émulateur.

Derniers prérequis

Vous avez tous les outils ? (Sauf peut-être TILP mais c'est pas grave) Alors on peut continuer.

Je suppose que vous savez où trouver la ligne de commande dans votre système, et comment taper une commande dans un répertoire précis.

Je parlerai principalement de TI-83 et TI-83+ pour le reste du tutoriel, mais sachez que la TI-84+ exécute également les programmes TI-83+.

Précisions

Avant de commencer, il est bon d'avoir une idée de ce qu'on pourra faire. Certaines choses

seront possibles, d'autres pas. En comparaison au BASIC, la programmation en assembleur permet de complètement s'accaparer la calculette. Quand votre programme assembleur tourne, il ne se passe rien d'autre. Il n'y a pas non plus de protection: si votre programme plante, il se peut que la calculette se mette à exécuter n'importe quoi et finisse par se réinitialiser, auquel cas vous perdez toutes les données non archivées. Vous pouvez également accéder finement au matériel de la calculette, comme le port I/O, l'écran LCD et le port USB (pour les calculettes les plus récentes). Vous pouvez aussi aller changer des données dans les variables de l'OS de la calculette (à savoir les variables habituelles ou d'autres programmes).

Qu'est-ce qu'on ne peut pas faire en assembleur? Théoriquement, rien qui ne soit au-delà des capacités techniques de la machine. En pratique, la programmation assembleur est assez pénible, car contrairement à des langages plus élaborés comme le C ou le Python, il n'y a pas de compileur qui va vous dire que la logique de votre programme est cassée. Le compileur SPASM transcrira betement votre programme en code machine, ne s'arrêtant que si cette transcription est impossible.

Autre chose importante à savoir, vos programmes ne tourneront pas sur toutes les calculettes Z80 existantes sans quelques précautions. D'abord, au niveau binaire, les programmes de TI-82, TI-83 ou TI-83+ n'ont pas le même en-tête, par conséquent il est impossible d'écrire un programme qui sera accepté par ces trois modèles. D'autre part, vos programmes peuvent dépendre de bouts de codes déjà présents “en dur” dans une calculette, mais qui ne sont pas forcément présents dans une autre, ou bien sont présents mais à une autre adresse. Ces incompatibilités seront réglées par des fonctions simples du compileur SPASM, vous permettant d'avoir un code source unique pour générer des programmes sur plusieurs modèles de calculettes.

Un premier programme assembleur

Fix Me! faire programme exemple1.z80, et vion.inc . exemple1.z80 fait quelques opérations sur les registres (LD, ADD) et affichages à l'aide d'une routine intégrée dans le code (pas de rom call)µ. Vérifier les retours à la ligne pour windows...

Entrons dans le vif du sujet avec **un premier programme (sous forme de deux fichiers) que vous pouvez télécharger** sur votre ordinateur. Nous allons le compiler et l'exécuter sur émulateur.

Placez-vous dans le répertoire où se trouvenr les deux fichiers, et tapez la commande :

```
./build exemple1
```

Vous obtenez une sortie qui vous indique que tout s'est bien passé. Regardez dans le répertoire, un fichier `exemple1.83p` et `exemple1.8xp` sont apparus. Ce sont deux fichiers issus du même programme, pouvant s'exécuter respectivement sur TI-83 et TI-83+/84+.

Exécutez l'émulateur de votre choix. Vous pouvez le démarrer en mode TI-83, auquel cas vous devez envoyer le programme Venus à l'émulateur, ou TI-83+, auquel cas c'est MirageOS qu'il vous faut. Envoyez ensuite le fichier issu de la compilation de exemple1.z80 correspondant au modèle de la calculette. Sous TI-83, exécutez-le via le menu PRGM. Sous TI-83+, exécutez-le via le menu APPS, puis MirageOS.

Si tout va bien, cela veut dire que vous avez compilé votre premier programme assembleur et vous savez le faire tourner sur un émulateur. Si cela vous pose problème, essayez de trouver ce qui ne va pas avant d'avancer plus loin dans le tutoriel.

Vous ne savez pas encore ce qui se passe dans ce programme assembleur, mais nous allons y venir.

Les registres

La programmation en assembleur consiste à dire au processeur exactement ce qu'il doit faire. Un programme est constitué essentiellement d'une suite d'instructions, disponibles parmi les instructions supportées par le processeur courant. Vous pouvez voir la liste des instructions du Z80 ici (lien). Il y en a beaucoup, mais on va procéder dans l'ordre.

Les registres sont une partie essentielle du processeur Z80. Ce sont des emplacements qui peuvent contenir des valeurs, et qu'on peut manipuler pour divers calculs. Les principaux registres sont A, B, C, D, E, H et L. Ce sont chacun des registres 8 bits. (si vous ne savez pas ce que c'est, lien). Certains registres peuvent être vus comme des registres 16 bits: BC, DE et HL. En effet, certaines instructions du processeur Z80 s'effectuent directement sur ces registres 16 bits.

L'opération la plus simple est l'assignation, représentée par les instructions de type “LD” (pour “load”, charger):

```
ld a,10
call dispAln   ; afficher le registre A à l'écran
ld hl,4000
call dispHLln  ; afficher HL
ld b,40
ld a,b       ; mettre B dans A
call dispAln
ld de,666
ld h,d
ld l,e
call dispHLln  ; afficher HL
```

Vous remarquez que `ld a,b` signifie charger b dans a, et non pas l'inverse. Les lignes commençant par `call` indiquent un appel à une fonction d'affichage, nous y reviendrons plus tard.

On peut également additionner ou soustraire des registres:

```
ld a,100
ld b,20
ld c,2
add a,b     ; A reçoit A+B
add a,c     ; A reçoit A+C
add a,100   ; A reçoit A+100
call dispAln
sub 80      ; A reçoit A-80
call dispAln
```

Référez-vous au jeu d'instruction pour voir quelles sont les instructions d'addition et de soustraction disponibles. Les registres 16 bits ne sont pas délaissés:

```
ld bc,6
ld de,60
ld hl,500
add hl,de
add hl,bc
add hl,100
call dispHLln
```

Remarquez aussi qu'il n'existe pas d'instructions permettant de multiplier, de diviser ou de faire des maths avancées. Le processeur Z80 reste plutot simple, et il faut donc écrire soi-même le bout de code qui fera l'opération adéquate. Ou bien le récupérer.

La mémoire

La précédente section présentait les diverses opérations d'assignation et d'addition/soustraction sur les registres. Le processeur Z80 peut également accéder à la mémoire de la machine, et y lire ou écrire des données. Par exemple, l'hypothétique code suivant:

```
ld a,66
ld (1000),a
ld a,50
call dispAln
ld a,(1000)
call dispAln
```

Ce programme afficherait successivement 50 et 66, en utilisant l'emplacement mémoire à l'adresse 1000 comme stockage intermédiaire.

Une telle possibilité soulève une question importante quant à la programmation sur TI: peut-on stocker ses données où on veut en mémoire ? La réponse est **non**: l'OS de la calculette contient de nombreuses variables situées un peu partout dans la mémoire, et les zones libres sont relativement peu nombreuses, mais très connues. Dans les écrits en anglais on appelle cela des “saferams” (mémoires sûres). Elles sont définies dans toutes les en-têtes utilisées par les programmeurs, qui n'ont ensuite plus qu'à s'y référer. La plus grande d'entre elle est souvent appelée “apd buffer”, “apd_buf” ou “saferam1”, et fait 768 octets. Ainsi, le programme suivant s'exécutera sans emcombre sur une vraie calculette:

```
ld a,66
ld (saferam1),a  ; on stocke la valeur au début de la zone libre
ld a,50
call dispAln
ld a,(saferam1)
call dispAln
```

Si vous regardez dans vion.inc, vous verrez plusieurs lignes impliquant saferam1, dont:

```
saferam1 .equ $xxxx
```

et

```
saferam1 .equ $xxxx
```

Sans ces lignes, SPASM ne saurait pas comment interpréter le mot “saferam1” et vous produirait une erreur. “saferam1” est défini plusieurs fois dans vion.inc, une fois par modèle de calculette, car cette zone libre n'est pas à la même adresse sur toutes les calculettes.

A noter que l'adresse est spécifiée en hexadécimal, en raison du préfixe $. Pour plus de précisions à ce sujet, (lien).

Rien ne vous empêche de définir vos propres mots correspondant à des adresses en mémoire, et si vous regardez le code de quelques programmes assembleurs existants, c'est souvent ce que l'on voit au début des programmes:

```
perso_x = saferam1   ; syntaxe alternative au .equ
perso_y = perso_x + 1  ; on laisse une place de 1 octet pour perso_x
perso_ammo = perso_y = 1
perso_sante = perso_ammo + 2 ; on laisse une place de 2 octets pour perso_ammo
```

Ainsi votre code peut devenir plus explicite, et vous pouvez dorénavant utiliser les mots que vous avez défini:

```
ld a,0
ld (perso_x),a
ld (perso_y),a
ld (perso_sante),a
ld hl,0
ld (perso_ammo),hl
```

Je ne ferai pas l'inventaire des instructions existantes et inexistantes quant à l'accès mémoire. Sachez simplement qu'elles sont moins nombreuses que celles qui manipulent les registres.

Appels, sauts et boucles

Pour l'instant, vous savez faire des opérations simples avec les registres et la mémoire de la machine. Mais l'ordre d'exécution d'un programme est rarement simple, et les appels à des sous-parties du programmes sont fréquents.

Nous avons vu dans la section précédente les labels servant à indiquer des endroits précis de la mémoire. Voici les labels servant à indiquer des endroits précis de votre programme, afin que vous puissiez y sauter ou en faire appel. Le bout de code suivant:

```
debut_boucle
 ld a,111
 call dispAln
 jp debut_boucle ; saute au label debut_boucle
```

est une boucle sans fin qui affiche la valeur 111 à l'écran encore et encore. L'instruction JP sert à sauter à un label situé n'importe où dans votre programme. Vous pouvez aussi utiliser l'instruction JR (“jump relative”, saut relatif) si le saut n'est pas trop éloigné: JR a l'avantage de prendre 1 octet de moins que JP dans votre programme.

Une boucle infinie n'est pas très intéressante, alors regardons l'instruction DJNZ:

```
 ld b,54
debut_boucle
 ld a,b
 call dispAln
 djnz debut_boucle
```

DJNZ décrémente le registre B, teste s'il est alors égal à zéro. Si non, il saute au label indiqué, si oui, le programme continue de s'exécuter sans saut. Ainsi le bout de code précédent affiche toutes les valeurs entre 54 et 1.

Les appels se font avec l'instruction CALL. Les bouts de codes vus jusqu'à présent contenaient la ligne:

```
call dispAln
```

Regardons un peu ce qu'il y a au label dispAln:

```
dispAln
 [...]
 ret
```

L'instruction RET signifie “retour à l'appelant”. Ainsi, un programme contenant la ligne `call dispAln` s'exécute jusqu'à cette ligne, puis exécute tout le code contenu entre le label `dispAln` et le premier RET rencontré, et revient exécuter tout ce qui est après la ligne `call dispAln`.

L'instruction `call` sert à appeler des sous-parties de code qu'on appelle souvent “routines”. “Fonction” serait également un terme approprié. En général, il est important de voir quelles sont les parties de votre code qui se répètent, afin de les classer sous forme de routines et de les appeler depuis divers endroits du programme.

Vous remarquez que le programme `exemple1.z80` se termine lui aussi par un RET. C'est normal: ce RET sert à indiquer au processeur un retour à l'appelant, qui est dans ce cas... le shell que vous utilisez! On y reviendra plus tard.

Une dernière chose pour démystifier un peu tout cela. J'ai dit plus haut que les labels sont un peu comme les mots indiquant des adresses mémoire de la section précédente. En vérité, c'est exactement la même chose, et le compilateur ne voit aucune différence entre les deux. Pourquoi ? Sachez qu'à l'exécution, votre programme assembleur est lui aussi situé en mémoire, à un endroit qui peut être accédé si vous connaissez la bonne adresse. Et le compileur a besoin de connaitre cette adresse à laquelle est votre programme quand il est exécuté. En effet, la valeur exacte des labels est une adresse mémoire. Vous avez parfaitement le droit d'écrire:

```
call 1000
```

ou

```
jp 3333
```

dans votre programme (cela va certainement planter cependant).

Pour indiquer au compileur quelle est l'adresse de début du programme dans la mémoire, on utilise la macro `.org`. Ce n'est pas une instruction Z80, mais une indication que le compileur comprend. Regardez à la fin de vion.inc, vous trouverez:

```
.org $9327
```

pour la TI-83. Ainsi, tous les labels du progamme sont remplacés par les valeurs exactes de la mémoire calculées à partir de 1) l'adresse de début indiquée par `.org` et 2) la taille des instructions placées entre le début du programme et le label courant.

Cela nous permet de conclure en remarquant que c'est là une fonction essentielle des shells sur calculette: placer votre programme à un endroit prédéterminé de la mémoire avant son exécution. Si le programme était placé à un autre endroit, le moindre appel ou saut interne arriverait à une mauvaise adresse et c'en serait fini de l'exécution correcte de votre programme.

Appels et sauts conditionnels

Je vous ai caché l'existence d'un autre registre du processeur : F. F pour “flags” (drapeaux). Les flags sont des bits dont la valeur indique si oui ou non la précédente opération effectuée par le processeur a eu un effet précis. Ce registre F sert principalement de conteneur à ces flags. Vous ne pouvez pas utiliser F pour y charger des valeurs ou faire des calculs. L'utilité du registre F est indirecte: les instructions JP, JR, CALL et RET existent sous différentes variantes qui tiennent compte de la valeur d'un flag précis.

Les flags les plus utilisés sont Z (zéro), C (carry ou retenue) et S (signe). Le seul moyen de savoir précisément comment ces flags sont modifiés par diverses instructions est de regarder le jeu d'instructions du Z80. Cependant, on peut s'en tirer en se rappelant certaines choses. Z est allumé si l'instruction précédente a produit un résultat nul. C est allumé si l'instruction précédente a produit une retenue. Par exemple, additionner deux registre 8 bits quand le résultat fait plus de 8 bits et nécessite alors une retenue. S est allumé ou éteint selon le signe du résultat.

Par exemple:

```
ld a,(perso_x)
ld b,50
cp b    ; comme un SUB B, mais sans modifier ce qui es dans A. Modifie les flags.
jp z,perso_x_limite_50 ; si perso_x == 50, saut
```

Une autre façon de le faire (pas forcément plus efficace):

```
ld hl,perso_x
ld a,50
cp (hl)                ; compare avec ce qui est à l'adresse mémoire perso_x
jp z,perso_x_limite_50 ; si perso_x == 50, saut
```

Les rajouts aux instructions JP, JR, CALL et RET sont les suivants: Z pour zero, NZ pour non zero, C pour carry, NC pour non carry, P pour positive, M pour negative. Il y en a d'autres, comme PE pour parity even et PO pour parity odd (respectivement: pair et impair).

Si vous voulez voir une utilisation intensive des sauts et appels conditionnels, vous pourrez jetez un oeil au code de l'exemple qui viendra dans quelques sections. Les routines les plus optimisées ont tendance à exploiter un maximum de possibilités du Z80.

Mettre des données dans son programme

On sait maintenant que programmer en assembleur, c'est indiquer au compileur quelles sont les instructions à insérer dans son programme. Mais on peut aussi lui indiquer d'insérer des données, qu'elles soient numériques ou textuelles.

Les valeurs numériques s'insèrent avec .db (declare byte: déclarer octet) ou .dw (declare word, déclare mot (un mot est ici l'ensemble de deux octets, donc 16 bits)). Par exemple:

```
 ld a,(valeur1)
 call dispAln
[...]
valeur1 .db 42
```

affichera 42. De même:

```
 ld hl,(valeur2)
 call dispHLln
[...]
valeur2 .dw 1111
```

affichera 1111. Pour le texte, on utilise également .db, mais suivi d'une chaine de caractères :

```
 ld hl,chaine1
 call dispString
[...]
chaine1 .db "Bonjour !",0
```

Ici, `dispString` est une routine qui affiche une chaine à l'adresse HL finie par zéro (d'où le `, 0` après la chaine ci-dessus). À noter que tout est question d'interprétation. Le texte déclaré au label `chaine1` est représenté sous forme de valeurs ASCII (lien), et c'est la routine dispString qui interprète ces valeurs comme du texte.

En parlant d'interprétation, on peut déjà avoir un petit aperçu de comment faire du graphisme. On appelle “sprite” un élément de graphisme qu'on affiche à l'écran et qui existe sous forme de tableau indiquant la valeur de chaque pixel (comme on est en noir et blanc, chaque élément du tableau indique “allumé” ou “éteint”). Voici un sprite typique:

```
sprite1
  .db %01111110
  .db %01100110
  .db %11100111
  .db %11100111
  .db %11100111
  .db %11100111
  .db %01100110
  .db %01111110
```

Il fait 8 pixels sur 8, et est représenté sous forme de 8 octets écrits de façon binaire (d'où le % devant chaque valeur). Nous verrons plus tard comment afficher un sprite.

Au bit près

Certaines opérations calculatoires ou graphiques nécessitent de manipuler des valeurs contenues dans les registres au bit près. Je parlerai ici de trois types d'instructions: les instructions logiques, les instructions de décalage et de rotation et enfin les accès directs aux bits.

Opérations logiques

Les principales instructions logiques sont OR, AND et XOR. Ce sont trois fonctions booléennes (c'est-à-dire qui font des opérations sur des valeurs VRAI ou FAUX) définies ainsi:

```
0 OR 0 = 0
1 OR 0 = 1
0 OR 1 = 1
1 OR 1 = 1
```

```
0 AND 0 = 0
1 AND 0 = 0
0 AND 1 = 0
1 AND 1 = 1
```

```
0 XOR 0 = 0
1 XOR 0 = 1
0 XOR 1 = 1
1 XOR 1 = 0
```

Les instructions OR, AND et XOR viennent en plusieurs versions, permettant d'effectuer ces opérations bit par bit entre 2 registres, entre un registre et une valeur ou entre un registre et le contenu d'un emplacement mémoire.

Autre opération logique: l'inversion (ou NOT):

```
NOT 0 = 1
```

```
NOT 1 = 0
```

Elle est représentée uniquement par l'instruction CPL, qui inverse chaque bit du registre A.

Vous verrez plus tard que ces instructions sont très utiles pour les routines graphiques. L'affichage d'un sprite consiste à effectuer une opération OR entre lui-même et une zone mémoire dédiée à l'affichage.

Mais on peut trouver d'autres utilités à ces instructions. Par exemple, voici une version de DJNZ qui utilise le registre 16 bits BC comme compteur, en s'aidant de l'instruction OR:

```
 ld bc,30000
boucle
 ld h,b
 ld l,c
 call dispHLln
 dec bc
 ld a,b       ;
 or c         ; si BC == 0, alors A==0
 jp nz,boucle
```

De même, vous trouverez souvent XOR A à la place de LD A,0, car cette instructions éteint tous les bits du registre A en un seul octet et plus rapidement.

Il faut se rappeler que les flags sont modifiés par ces instructions logiques. On peut en tirer parti, par exemple, ainsi:

```
ld a,(valeur1)    ; charge la valeur dans A; ne modifie aucun flag
or a              ; ORe A avec lui-même (pas de changement), modifie les flags
jp z,valeur_nulle
```

Décalage et rotation

Il existe une série d'instructions permettant d'effectuer des décalages et des rotations de bits sur les registres 8 bits uniquement.

Les instructions RL et RR effectuent une rotation vers la gauche et vers la droite sur les 9 bits constitués des 8 bits du registre indiqué et du bit/flag carry placé du coté de la rotation.

RLC et RRC font de même, mais uniquement sur les 8 bits du registre indiqué (cependant le flag carry est modifié).

SLA et SRL effectuent un décalage des 8 bits du registre indiqué vers la gauche et la droite, en remplaçant la place vide par un bit 0 et en plaçant le bit sorti du registre dans le flag carry.

D'autres sont moins souvent utilisés: SRA est comme SLA mais conserve le bit 7 tel qu'il était avant décalage. Fix Me! utile pour nombres négatifs? SLL est comme SRL mais remplace le bit 0 par 1. Fix Me! utile pour quoi?

Accès direct aux bits

SET b,r et RES b,r resp. allument et éteignent le bit b (b allant de 0 à 7) du registre r.

BIT b,r copie l'inverse du bit b du registre r vers le flag Z (si le bit est nul, le flag Z devient 1, et inversement).

Par exemple, pour tester si le registre c contient un nombre négatif:

```
bit 7,c
jp nz,c_negatif
```

En effet, un nombre 8 bit supérieur à 128 (ie dont le 7e bit est allumé) peut être interprété comme un nombre négatif.

La pile

Quand on est à court de registres, et qu'on ne veut pas utiliser d'emplacements mémoire (dont les instructions sont plus lentes que celles impliquant des registres), il existe une solution alternative: la pile.

La pile peut être vue comme une pile d'assiettes, où la dernière assiette posée est la première qu'on peut reprendre. Le posage et le reprenage d'assiettes sont effectués par les instructions PUSH et POP. Ces instructions permettent de copier des registres 16 bits (pour rappel, BC, DE, HL mais aussi AF dans ce cas particulier) de et vers la pile.

Concrètement, la pile est un emplacement mémoire réservé, dont l'adresse (changeante) est stockée dans le registre SP (Stack Pointer, pointeur de la pile), que le programmeur n'est pas censé manipuler (sauf à l'occasion de quelques instructions très particulières). Lors d'un PUSH, SP est décrémenté de 2, et le contenu du registre 16 bits donné est copié à l'adresse pointée par SP. Lors d'un POP, c'est l'inverse: le contenu de l'adresse pointée par SP est copié dans le registre donné, puis SP est incrémenté de 2. Ainsi, plus on empile des données, plus SP recule. Il existe donc une valeur limite à SP, mais elle est assez élevée (Fix Me!), donc il y a rarement de quoi se faire du souci.

Cette capacité de sauvegarder et restaurer des registres 16bits est pratique pour lors d'appel à des routines pouvant détruire leur contenu. Vous verrez (et écrirez) souvent du code comme ceci:

```
push hl
push de
push bc
push af
call affiche_nain_de_jardin_3D
pop af
pop bc
pop de
pop hl
```

Tant que nous y sommes, revenons à CALL et RET. Nous avons vu plus haut que RET revenait au dernier appelant. Comment la machine sait-elle où se trouve l'appelant ? Grâce à la pile.

Pour expliquer cela, je dois évoquer un autre registre: PC (Program Counter). Encore plus obscur que SP, il semble n'être directement manipulable avec aucune instruction, et pourtant il est indispensable et sa valeur change sans cesse. Et pour cause: ce registre contient l'adresse mémoire de l'instruction actuellement exécutée.

Lorsque CALL xxxx est exécuté, il est l'équivalent d'un PUSH PC+3 , JP xxxx. PC+3 car PC pointe sur le début de l'instruction CALL xxxx qui fait 3 octets, et on veut qu'au retour de la routine, l'exécution du programme continue après cette instruction (sinon on bouclerait à l'infini).

Lorsque RET est exécuté, il est l'équivalent d'un POP temp ; JP temp avec temp qui serait un hypothétique registre temporaire. Ainsi, vous pouvez très bien placer ceci dans votre code:

```
  push destination
  ret
[...]
destination
  ; suite du programme
```

Ce qui est l'équivalent d'un `jp destination`.

Il faut bien veiller à ce que l'état de la pile soit cohérent. De nombreux bogues viennent d'un mauvais emplilement ou dépilement.

Un petit exercice: savez vous comment afficher l'adresse courant du Program Counter? Solution: Fix Me! exemple2.z80 (call affpc; affpc: pop hl; call dispHLln)

Un peu de pratique: affichage d'un sprite

Nous allons maintenant résumer tout ce que nous avons vu avec un programme graphique. Vous pouvez le récupérer ICI. Fix Me! vion.inc , exempleSprite.z80, routines.inc (pour mettre trucs trop compliqués comme dispsprite ou fastcopy)

Vous pouvez le compiler et le tester sur émulateur. Ce programme affiche un sprite rebondissant sur les bords de l'écran pendant quelques temps avant de s'arrêter. Le code du fichier routine.inc est encore trop avancé pour que nous puissions le comprendre maintenant. Je vous laisse lire le contenu de exempleSprite.z80, qui fait appel à des choses que vous avez vues dans les sections précédentes.

(gif animé)

(sprite animé en 8 étapes (temps, and %00000111 pour choix du sprite), se déplaçant àl'écran (pos_x , pos_x, vel_x,vel_y dans saferam) , rebondissant sur les bords de l'écran (tests), pendant un certain temps (djnz ou equivalent 16 bits), bords de l'ecran texture qui défile (RLC),) (pas de lecture du clavier)

Rom calls

Les calculettes TI contiennent dans leur mémoire des centaines de routines servant à l'exécution de l'OS. Ces routines peuvent également être appelées à partir de programmes assembleur: ce sont les Rom Calls.

Les divers modèles de calculettes (82, 83, 83+...) contiennent des Rom Calls pour la plupart identiques, mais situées à des adresses différentes selon le modèle. C'est pour cela que les programmeurs utilisent un fichier d'en-tête (comme vion.inc que vous connaissez un peu), définissant les adresses correspondant à chaque nom de rom call sur les divers modèles.

Il existe des listes de rom calls avec une explication de l'effet de chacune. Vous en trouverez une ici (lien wikiti).

Sur TI-83, ces Rom Calls s'appellent simplement avec `call`. Sur TI-83+, un appel à une rom call nécessite d'effectuer quelques opérations sur la mémoire; en effet, elle se présente sous forme de plusieurs banques de mémoires interchangeables. Pour éliminer cette différence dans le code source, on utilise le mot BCALL qui est remplacé par la ou les instructions adéquates par SPASM au moment de la compilation.

Vous pouvez récupérer ce programme d'exemple (interactif, cette fois) qui utilise trois Rom Calls:

- afficher du texte
- effacer l'écran
- lire le clavier (getcsc)

Ce programme attend que vous pressiez sur la touche 2nd pour quitter. Presser sur les touches Fleche Gauche ou Fleche Droite décale le texte affiché.

SPASM

SPASM est le logiciel que vous utilisez pour convertir votre code source en code machine. Il s'occupe de reconnaitre les instructions, ainsi que les données (numériques ou texte) et de les convertir en les bonnes valeurs.

Il remplace les mots par les adresses mémoire correspondantes, et remplace également les labels situés dans votre code par de véritables adresses mémoires. Pour cela, il se base sur l'adresse du début du programme en mémoire, indiquée par `.org`, et sur la taille des instructions précédant le label.

Il est également capable d'effectuer quelques opérations arithmétiques. Par exemple, le code:

```
ld a,(34+(16*3))
```

est remplacé à la compilation par `ld a,82`. Il peut également remplacer des mots définis par leur valeur. Par exemple:

```
HAUTEUR = 64
LARGEUR = 92
ld a,HAUTEUR+LARGEUR
```

C'est intéressant pour rendre son code plus lisible et éviter les “valeurs magiques” (ie les nombres apparaissant dans le code sans explication).

SPASM est capable d'inclure un fichier dans un autre avec `#INCLUDE fichier.Z80`, qui peut être placé n'importe où dans le code.

Il peut aussi inclure du code dans le code final en fonction de conditions. Par exemple, le code:

```
#DEFINE TI83
#IFDEF TI83
 ld hl,bonjour83
#ELSE
 ld hl,bonjour83plus
#ENDIF
 call _vputs
```

sera converti en:

```
 ld hl,bonjour83
 call _vputs
```

Cette fonction est essentielle pour faire facilement un programme à la fois sur différents modèles de calculettes. Le script `build`, si vous regardez son code, consiste à copier le code source donné dans un fichier temporaire contenant une fois `#DEFINE TI83` et l'autre `#DEFINE TI83P`. Dans les deux cas, les adresses des rom calls, entre autre, sont par conséquent changées.

SPASM permet également de définir des macros. Dans vion.inc vous trouverez la définition de la macro `BCALL` (qui dépend elle-même de la présence de `#DEFINE TI83` ou `#DEFINE TI83P`). Une macro peut avoir zéro, un ou plusieurs arguments.

D'autres fonctions peuvent être effectuées par SPASM, mais pour le moment vous avez vu le plus important. Une dernière chose : vous pouvez ajouter l'argument `-T` à l'appel de spasm dans le script `build`. Cela indique à SPASM de sortir le listing du programme compilé, qui est sa représentation en hexadécimal mise cote-a-cote avec le code source correspondant. Cela vous permet de vérifier que tous les remplacements on été faits correctement. Utile en cas de débogage.

Shells (Venus, ION, MirageOS ...) et conséquences

Les shells sonts présentés comme indispensables pour exécuter des programmes assembleur sur calculette. Ce n'est pas vrai pour les modèles relativement récents (à partir de TI-83), qui permettent de lancer des programmes assembleur sans rien installer. Cependant, un shell offre des fonctions quasi-essentielles pour les programmes assembleur. Hormis l'interface graphique permettant de lancer des programmes, je parlerai de deux fonctions.

La première est la copie du programme à un endroit précis de la mémoire. Je l'ai déjà évoqué, donc vous savez pourquoi c'est important.

La deuxième est la recopie du programme à sa place initiale après exécution. La motivation semble moins claire: en effet, le programme est toujours à l'endroit où il était lors de son exécution, pourquoi l'y recopier? Parce que vous pouvez avoir changé des valeurs *à l'intérieur* de votre programme et voulez les conserver. Cela vous permet par exemple de gérer les sauvegardes. Une solution alternative consiste à sauvegarder les données dans une variable de l'OS, mais cela nécessite des manipulations plus compliquées.

Regardez l'exemple suivant pour une illustration simple Fix Me!. (exemple qui affiche un compteur incrémenté)

Par curiosité, vous pouvez voir comment c'était avant, les shells, prenez par exemple (Fix Me! lien sur ticalc.org), et voyez ce qui est fait pour 1) corriger les labels lors du démarrage du programme 2) réécrire les données modifiées à la fin de l'exécution du programme. C'était du travail de Vrai Programmeur, non? :)

Lecture rapide du clavier : direct input

Nous avons vu dans la section Rom Call une rom call permettant de lire le clavier. Cela vous permet déjà de créer des programmes interactifs. Cependant, cette rom call est limitée car 1) ne permet pas de lire la pression multiple de touches et 2) lente. C'est pourquoi nous allons voir une alternative bien plus “bas niveau”: le direct input.

Nous devons faire la connaissance de nouvelles instructions: `IN` et `OUT`. Ces instructions permettent de lire et écrire des données via les ports du processeur. Selon le modèle de calculette, il y a plus ou moins de ports disponibles. Les ports les plus utilisés et communs à tous les modèles de calculettes sont ceux du clavier, du port link et (sauf pour les vieux modèles) de l'écran. Vous trouverez la liste complète des ports sur (wikiti lien).

Pour la lecture du clavier, il va falloir communiquer avec le port ??. La lecture se fait via un échange écriture/lecture sur le port: on envoie sur le port la zone du clavier qu'on veut lire, et on lit sur ce port quelles sont les touches actuellement pressées. En lisant le port, on récupère cette

donnée dans un registre dont les bits indiquent alors la pression des touches. Un exemple:

Fix Me!

Les mots indiquant les zones du clavier et les valeurs pour lire les touches sont définis dans vion.inc; ce sont les mêmes pour tous les modèles de calculettes.

Vous pouvez récupérer ici Fix Me! une nouvelle version duprogramme qui affiche un sprite. Cette fois, vous pouvez controller sa trajectoire avec les fleches, et quitter avec 2nd.

Retour sur les graphismes

Nous avons jusqu'à présent affiché du texte ou des sprites en faisant appel à des routines, que nous n'avons pas encore explorées. Nous allons maintenant voir ce qui se passe dans ces boites noires.

Tout se passe par l'intermédiaire d'une zone mémoire appelée plotsscreen ou graph buffer. Cette zone fait 768 octets, à savoir 64*12: 12 octets pour chaque ligne de l'écran (12*8=96 pixels) et 64 lignes. Cette zone de 768 octets est ensuite “copiée” vers l'écran LCD par une rom call appellée _grbufcopy, ou bien une routine bien plus rapide appelée `fastcopy`. Il faut passer par une telle routine car l'affichage sur écran LCD se fait via un port du processeur. Sur les modèles TI-85 et TI-86, la zone mémoire plotsscreen est matériellement automatiquement répercutée sur l'écran. Sur tous les autres modèles, il faut communiquer avec le pilote de l'écran. Heureusement, ce n'est pas notre problème (sauf si vous voulez vous lancer dans des optimisations bas niveau).

Pour allumer un pixel à une coordonnée (x,y) de l'écran, il faut allumer un bit précis dans la zone plotsscreen. Lequel? Le tout premier octet de la zone correspond aux 8 pixels en haut à gauche de l'écran, et le dernier aux 8 pixels en bas à droite de l'écran. Pour allumer le pixel (x,y) de l'écran, il faut déjà retrouver l'octet de plotsscreen concerné, puis retrouver le bit de l'octet concerné. Jetez un oeil à cette routine d'affichage de pixel, vous en savez maintenant assez pour savoir ce qui s'y passe (lien).

Une routine d'affichage de sprite doit reprendre la même logique pour commencer à fonctionner. La différence est qu'au lieu d'afficher juste un pixel, il faut répercuter chaque octet des données d'un sprite dans la zone plotsscreen, par une opération de OR, XOR ou AND logique. Regardez ici (lien) pour le code commenté d'une routine XOR de sprite 8*8 pixels.

Les jeux qui veulent être un peu plus travaillés dans leur aspect utilisent des routines de AND/XOR pour l'affichage de sprite. Chaque sprite est défini en 2 parties: le masque AND et le sprite XOR. Le but du masque AND est d'éteindre des pixels, afin de donner de l'opacité à un élément de graphisme autrement qu'avec du noir. Puis, la partie XOR rallume les pixels noirs de l'élément graphique. Il laisse la liberté également d'avoir des pixels inversés. En résumé:

```
AND XOR effet sur le pixel existant
0   0           éteint
0   1           allumé
1   0           aucun
1   1           inverse
```

Tilemaps

La plupart des jeux nécessite un décor de fond. Ce décor peut être une image enregistrée, mais une solution plus efficace est celle du “tilemap” (littéralement, carte de dalles). Une tilemap est un tableau dont chaque élément est une dalle à afficher. Quasiment tous les jeux en 2D sur consoles de jeux utilisent ce système. Une tilemap est constituée d'un jeu de dalles et de la tilemap

elle-même. Le jeu de dalle est un ensemble de sprites indexés, c'est-à-dire qu'à un nombre entier donné correspond un sprite. La tilemap est un simple “tableau” (c'est à dire une zone de mémoire contenant des nombres et interprétée comme tableau) contenant ces index.

Pour une routine de tilemap facile à comprendre, à défaut d'être la plus optimisée, voir ici pour un programme d'exemple. L'intérêt d'une routine de tilemap est d'être plus optimisée que des appels successifs à une routine de sprite par dalle.

Code auto-modifiant

Les OPcodes sont simplement des données en mémoire interprétées comme des instructions par le processeur si le PC pointe dessus. On peut donc les modifier en d'autres OPcodes, le processeur n'y verra que du feu.

Introduction du mot spécial $, qui signifie “l'adresse courante”.

Exemple:

```
 ld hl,666
 ld (emplacement),hl
[...]
emplacement = $ + 1
 ld hl,0
```

En effet, l'OPcode correspondant à `ld hl,xxxx` est `00100001 alalalal ahahahah` avec `alalalal` étant l'octet bas et `ahahahah` l'octet haut du nombre `xxxx`. Il suffit donc d'avoir un label qui pointe sur l'emplacement mémoire où se trouve les 2e et 3e octets de cette OPcode pour pouvoir la changer à la volée et faire charger n'importe quelle valeur dans hl.

On peut aller plus loin et modifier une OPcode en une autre. Par exemple, dans fastcopy, un nop ou un cpl peuvent faire passer la routine d'un affichage normal à un affichage inversé.

Nombres positifs et négatifs

Autre chose qui vous fera relativiser l' “intelligence” (ou plutot le manque de) du Z80, est son ignorance des nombres négatifs. En somme, c'est à vous de savoir si un registre ou une adresse mémoire est censée contenir un nombre pouvant être positif ou négatif, ou bien étant toujours positif. Sa seule contribution à la reconnaissance des nombres négatifs est le flag S (signe), et les instructions de branchement et de saut dépendant de la condition P ou M.

Grâce à l'encodage utilisé pour représenter les nombres positifs ou négatifs sous forme binaire, vous pouvez additionner un nombre positif et un nombre négatif, le résultat sera le même que la soustraction de deux nombres positifs correspondants.

Interruptions

J'écris cette partie avec un peu de réticence, car l'effort nécessaire à expliquer comment fonctionne cet aspect de la programmation assembleur est assez important vis-à-vis de la “récompense”.

Pour le Z80, une interruption est un bout de code exécuté lorsque le processeur capte un tic d'horloge, qui survient environ 140 fois par seconde. Ce bout de code s'exécute par un appel, comme un `call`, s'intercalant avec l'exécution normale du processeur. Cela permet de faire une

sorte de multitâche simple, mais périlleux, car on a voir qu'il faut être prudent.

Le processeur peut s'exécuter dans divers modes, en utilisant les instructions `IM` (interrupt mode). `IM 0` ne peut pas être utilisé, `IM 1` exécute l'interruption standard du TI-OS (celle qui fait s'éteindre la calculette au bout de quelques minutes, entre autrs), et `IM 2` vous permet d'exécuter votre interruption personnalisée. Les instructions `EI` et `DI` (enable/disable interruptions) permettent d'allumer et éteindre les interruptions. Il est prudent d'éteindre les interruptions pendant qu'on règle les paramètres de son interruption personnalisée.

En mode `IM 2`, à chaque tic de l'horloge, le processeur appelle ce qui est à l'adresse contenue à l'adresse (i*256)+X où i est le contenu du registre i (que nous n'avons encore pas vu) et X est une valeur inconnue (qui dépend de ce qui est dans le bus du processeur).

Vous pouvez définir le contenu du registre `I` avec une instruction `LD I,A`, et par conséquent il vous faut donc remplir toute la zone mémoire de I*256 à I*257 avec des données telles que quand on prend deux octets A et B à la suite, l'adresse mémoire (B*256+A) contient l'adresse de l'interruption. La solution à cette complication est de remplir cette zone avec le même octet A, et de placer l'interruption à l'adresse (A*256 + A). La valeur de A est à choisir judicieusement selon les zones de mémoire libres disponibles sur la calculette. Sur TI-83+, $9A convient, sur TI-83 $85.

L'initialisation d'une interruption se fait donc ainsi:

Fix Me! code

Il faut ensuite écrire le code de l'interruption elle-même. Comme ce code s'exécute au beau milieu du code ordinaire, il faut préserver l'état des registres. Pour cela, on utilise habituellement les shadow registers (registres de remplacement) spécialement conçus pour cette tâche. En l'espace de deux instructions, on peut échanger les registres normaux et les registres de remplacement, comme on peut le voir dans cet exemple

```
mon_interruption
 ex af,af'
 exx        ; de même pour bc,de,hl
 push hl    ; \
 inc hl     ;  corps de la routine d'interruption
 pop hl     ; /
```

```
 ld     a,$08    ;\
 out    ($03),a  ; |On resette les triggers pour que
 ld     a,$0F    ; |l'interruption redémarre la prochaine fois
 out    ($03),a  ;/  (source alexis guinamard)
```

```
 ex af,af'  ; restaure les registres
 exx        ;
 ei         ; réactive les interruptions (indispensable)
 reti       ; ret spécial pour revenir d'une interruption
```

La partie “resetter les triggers” est un mystère pour moi; le tutoriel d'Alexis Guinamard ne précise pas ce que cela signifie, et Asm in 28 Days, bien qu'expliquant l'utilité du port 3, n'explique pas pourquoi il influe sur les interruptions.

Il faut faire attention à ce qui se passe quand votre programme assembleur se termine. Si vous n'êtes pas en mode `IM 1`, le TI-OS ne fonctionnera pas comme prévu.

L'utilité? Vous pouvez maintenant compter le nombre d'images par seconde de votre programme. Ce ne sera pas très précis car la vitesse du processeur dépend de l'état des piles, mais ça vous donnera une bonne idée. Certains on utilisé les interruptions pour jouer de la

musique sur le port I/O, ou réguler le nombre d'images par seconde.

Pour un exemple concret des interruptions, regardez le code de Fix Me! ticalc.org ...

Les émulateurs ont une simulation des interruptions assez variable. VirtualTI et TilEm sont mauvais. WabbitEmu est le plus fidèle au matériel.

Vous trouverez une explication plus détaillée en anglais dans Fix Me! lien (Learn ASM in 28 days). Cependant, ce chapitre contient une erreur, car il faut bel et bien remplir toute la zone mémoire entre I*256 et I*257 (inclus).

Enfin, l'instruction `HALT` met le processeur en pause jusqu'à l'arrivée de la prochaine interruption. Si elles sont désactivées (`DI`), vous pouvez considérer votre programme comme planté. Cette instruction est utile pour économiser de l'énergie dans une boucle qui attend, par exemple, la pression d'une touche.

Emulateur et débogage

Programmer en assembleur efficacement implique écrire prudemment le code, et savoir déboguer. Pour cette deuxième partie, il existe diverses méthodes: rajouter des bouts de code qui affichent des valeurs internes du programme (registres...) pour repérer la partie qui plante, mettre des assertions, ou bien déboguer avec un émulateur.

Pour la deuxième partie, je vous renvoie au jeu de routines d'assertions (lien def) qui vous fera gagner du temps.

Nous allons voir icicomment déboguer avec un émulateur. Je vais prendre TilEm; WabbitEmu fonctionne de manière similaire.

Le débogueur d'un émulateur est un outil permettant de stopper l'émulation et d'exécuter le processeur pas à pas, en ayant une vision de l'état des registres et du contenu de la mémoire. Il est possible de placer des breakpoints, c'est-à-dire de marquer une case mémoiree pour que l'émulation s'arrête quand le processeur s'apprête à éxécuter le code qui s'y trouve. Un bon émulateur permet de retrouver aisément à quoi correspondent les données présentes dans la mémoire de la machine, en se basant sur les adresses mémoire connues (rom calls, saferams...) et sur les labels définis dans le code source.

Il faut appeler spasm avec l'option -L pour générer un fichier de définitions de labels. Dans ce fichier figurent des lignes de la forme LABEL = $1234.

Puis dans TilEm, appuyer sur F11 pour déclencher le débogueur, faire clic droit et choisir “Load symbols” et prendre le fichier généré. Ensuite, lancez votre programme, et dans le débogueur, placez-vous à l'adresse mémoire correspondant à progstart ($9327 sur TI-83, Fix Me! sur TI-83+): vous voyez le code de votre programme, avec les sauts et appels internes (JP, CALL) faisant figurer les labels de votre code source à la place de valeurs inexpliquées. À vous maintenant de jouer et de repérer les zones de votre code qui posent problème, en placant les breakpoints adéquats.

Le port link

- principe de base : envoyer des données avec IN et OUT
- routines utiles : Bell de timendus

Voilà, c'est fini!

- Vous n'avez pas tout vu, mais vous pouvez trouver ailleurs
- Remarques, erreurs, contributions, étendre la FAQ : écrivez-nous
- A l'avenir: mise a jour pour SPASM 2 et éventuellment WabbitEmu Linux

FAQ

ti/tutoriel.txt · Last modified: 2008/07/17 16:45 by gh