Zilog Z80

tTh

24 avril 2019

Table des matières

1 Introduction

Le `Z80` est un descendant du 8080 d'Intel, lequel a été le premier microprocesseur réellement utilisé à grande échelle. Le `Z80` a apporté de nombreuses améliorations à son ancètre : des registres et des instruction supplémentaires pour l'aspect logiciel, et bien d'autres choses coté hardware (que nous verrons peut-être plus tard).

Ce processeur a été utilisé dans de nombreux ordinateurs dans les années 80. Parmi les plus connus, nous pouvons citer le fabuleux `TRS80` [1], le `ZX80` de Sir Clive Sinclair, l'Amstrad CPC, la grande série des MSX [2], et bien d'autres, plus discrets, comme le Hector ou le Jupiter ACE. On le retrouve également dans la Gameboy de Nintendo et certaines calculettes de Texas-Instrument.

Il existe aussi de nombreux émulateurs qui reproduisent plus ou moins bien quelques un de ces anciens ordinateurs. Nous allons par la suite utiliser le très classique `z80pack` dans sa version de base, nous allons donc faire du *baremetal* comme des grands. Son utilisation est expliquée page 8.

2 Le processeur

Rentrons maintenant dans le vif du sujet, ce fameux cpu `Z80`. Il fait partie de la famille des microprocesseurs 8 bits, c'est à dire qu'il peut traiter 2^8 (soit 256) valeurs différentes en une seule opération. Il peut aussi assez facilement traiter des mots de 16 bits, soit 65536 valeurs différentes.

Ces 16 bits permettent d'adresser 64 kilo-octets de mémoire. Même pas de quoi lire une `gif89a` de chaton mignon. Mais quand même assez pour jouer au Manoir de Mortevielle.

Pour communiquer avec le monde extérieur, le `Z80` dispose d'un espace d'adresse indépendant pour les entrées/sorties, qui permet de raccorder 256 périphériques sur le bus. Ils sont gérés par des instructions spécifiques : `IN` et `OUT`.

2.1 À l'intérieur

À l'intérieur de la puce, il existe des sortes de petites cases mémoire individuelles et directement connectées à l'unité de logique et de calcul. On appelle ça des **registres**,

Le premier de ces registres est l'accumulateur, ainsi nommé en hommage à la plus ancienne des opérations arithmétiques, l'addition [3]. C'est sur lui que vont travailler la plupart des instructions du processeur. Sa taille est de 8 bits.

Le second registre est appelé `F` (abréviation de flags), ou *registre d'état* en français. Il ne contient pas vraiment un octet, mais des bits individuels qui indiquent

1. Aka : Fabulous Trash-80
2. Un ancien mais très beau *fail* de Microsoft
3. Opération très dévalorisée de nos jours par les reptiliens mondialo-capitalistes

chacun une condition différente, par exemple "*l'opération qui vient d'être faite a donné un résultat égal à zéro*".

nom	utilisation
S	signe : est-ce négatif ?
Z	est-ce zéro ?
AC	
P/O	parity, overflow
N	
C	carry, retenue

Ce registre d'état est essentiel à la gestion harmonieuse des flux d'exécution, il est mis à jour par certaines opérations (piège : mais pas toutes) sur l'accumulateur, et nous verrons cela vers la page 6.

Ensuite, nous avons trois registres d'usage plus ou moins général : `HL` plutôt spécialisé dans les adresses, puis `BC` et `DE`,. Chacun d'eux ayant une largeur de 16 bits, ils peuvent être 'coupés' en deux registres de huit bits.

Pour finir, nous pourrions aussi citer `IX`, `IY` et d'autres petites choses exotiques qui nous seront utiles dans un avenir plus ou moins lointain.

2.2 À l'extérieur

Il y a des BUS, trois ensembles de fils qui transmettent chacun un bit, unité élémentaire d'information et permettent au processeur de discuter avec le grand monde, qui pour lui est d'abord constitué de sa mémoire RAM et d'un contact avec son humain personnel.

Chacun de ces ensembles de fil a un rôle précis :

— Adresse, pour dire où est ce truc (16 fils) ;
— Donnée, pour connaitre la valeur du truc (8 fils) ;
— Contrôle, pour gérer tout le vortex (n fils).

Prenons rapidement l'exemple de la lecture par l'unité de calcul d'un octet résident dans un boitier de mémoire vive.

Cet octet a une adresse qui est exprimée sur 16 bits. Donc pour lire notre octet, ces 16 bits seront présentés sur le bus d'adressage, puis un des fils de contrôle va changer d'état pour indiquer au boitier mémoire qu'il doit fournir la donnée qu'il contient. Ce boitier mettra alors les bits de l'octet sur les 8 fils du bus de données, lequel sera alors lu par le CPU.

Voilà, vous venez de *fetcher* votre premier *opcode*, et c'est le début d'un long voyage...

3 Premier exemple

Depuis la naissance du seul véritable langage évolué, il est devenu obligatoire, comme premier exemple de programme, d'afficher une petite salutation : c'est le très classique « Hello World » que voici :

```
        ORG     100H
START:
        LD      SP,STACK        ; put the stack in a safezone
        LD      HL,HELLO        ; take location of the message
        CALL    OUTSTR          ; write it to the asr33
        HALT
; ---------------------------------
HELLO:
        DEFB    'Hello, World !',13,10,13,10,0
; ---------------------------------
OUTSTR:
        LD      A,(HL)          ; next char -> A
        OR      A               ; 0 ?
        RET     Z               ; yes, done
        OUT     (1),A           ; no, print it
        INC     HL              ; increase pointer to string
        JP      OUTSTR
; ---------------------------------
        DEFS    100H
STACK:
        END
```

Ce texte, compréhensible par un humain, est appelé le **code source**. Il devra être traduit dans un format que le processeur peut comprendre, qui sera le **binaire**. Nous verrons comment faire page 6.

Mais revenons à notre premier exemple, et décomposons-le ligne par ligne.

3.1 Boilerplate

La première ligne (`ORG 100H`) est une **directive** qui s'adresse à l'assembleur, pour lui demander de se positionner à l'adresse 256, après la première page mémoire, qui est reservé à une autre magie.

La seconde (`START:`) est un **label**, une étiquette qui prendra la valeur de l'adresse courante et permettra de s'y référer plus tard sous une forme symbolique..

La troisième (`LD SP,STACK`), est une véritable **instruction** destinée à être exécutée par le processeur pour charger une certaine valeur dans le *stack-pointer* `SP`. Elle doit être là pour une raison que nous verrons par la suite.

C'est fini pour les préparatifs, nous voici enfin au coeur du sujet, quand notre code applicatif se déclenche enfin...

3.2 Le Pointeur

Comme décrit dans notre cahier des charges, nous souhaitons afficher un message, c'est à dire une chaine de caractère. Il existe plusieurs façons de stocker ces chaines dans la mémoire d'un ordinateur. Nous allons choisir une convention d'un désuet classicisme qui a fait ses preuves depuis l'epoch.

Notre chaine est identifiée par le label `HELLO:` (que le processus d'assemblage va associer à une adresse en mémoire) et déclarée grâce à la directive `DEFB` (abréviation de *define byte(s)*) suivie du texte de notre message encadré par une paire de simples quotes. Mais ce n'est pas tout. Il y a ensuite deux codes numériques, 13 et 10, qui représentent les codes ASCII de *carriage return* et *linefeed*, séquence de passage à la ligne. Et pour finir, un **octet null marquera la fin de notre message**. Relisez bien : *octet null* et *fin de chaine*, croyez-moi, c'est important.

Il faut maintenant que le CPU connaisse l'adresse du message. Cette adresse est sur 16 bits, et l'instruction `LD HL,HELLO` va la charger dans le registre HL dont c'est justement le métier. Bravo, vous venez de découvrir le meilleur ami [4] du codeur punk : le **pointeur** ! Le concept est fondamental, mais parfois peu évident à comprendre [5].

3.3 Action !

Rappelez vous l'allumage de plaque (section 3.1) et le chargement du registre `SP`. Nous allons en découvrir la raison maintenant avec l'appel d'une fonction.

À la fin du code source, vous pouvez voir la directive `DEFS 100H` (abréviation de *define space*) qui va réserver un espace de 256 octets. Cette réservation est suivie du label `STACK:` et le registre `SP` sera chargée (voir étape précédente) avec l'adresse de la **fin** de cette zone réservée. Cette zone est appelée **pile**, ou **stack**.

Revenons à l'exécution de notre programme. Le registre `HL` vient d'être chargé avec l'adresse du premier caractère du message. Le processeur va alors exécuter l'instruction `CALL OUTSTR` qui aura deux effets complémentaires :

- L'adresse de l'instruction suivante (dans notre cas, c'est `HALT`) est empilée (sauvegardée) dans la pile. `SP` est modifié pour pointer sur une adresse libre.
- Le flot d'exécution est transféré à l'instruction dont le label est `OUTSTR` par le chargement de son adresse dans le *program counter* (`PC`).

3.4 Talking to the asr33

Nous venons donc d'entrer dans notre premier sous-programme (la fonction) dont le but est de transférer le message vers l'écran. Au moment de l'appel, l'adresse mémoire de la chaine est stockée dans le registre `HL`.

L'instruction `LD A,(HL)` va charger dans l'accumulateur l'octet pointé par `HL`, le premier char de notre chaine. Ensuite nous allons regarder si nous ne serions pas arrivés au bout de la chaine, marqué par un octet nul.

4. Fred, tu peux te calmer un peu ?
5. Quand le sage pointe la valeur, le fou regarde la lune.

Point suivant : comment détecter cet octet *null* ? Le premier octet de notre chaine vient juste d'être chargé dans l'accumulateur, il ne reste plus qu'à le tester et à agir en conséquence.

Nous allons maintenant avoir affaire à une particularité du Z80 : sa gestion des drapeaux. Lors du chargement d'une donnée l'accu, ils ne sont **pas** positionnés, il faut pour ça effectuer une opération (arithmétique ou logique) sur le contenu de **A**.

Dans notre cas, nous allons faire un ET de l'accu avec lui-même, ce qui ne change pas son contenu, mais place les drapeaux dans le bon état.

4 Le kit z80pack

On le trouve dans `http://autometer.de/unix4fun/` et dans notre contexte d'initiation, nous n'allons utiliser que deux éléments :

— l'assembleur
— le simulateur

Ce pack contient aussi des émulations assez avancées de machines et systèmes d'exploitation du siècle dernier, que nous verrons peut-être au printemps.

4.1 Assembleur

L'assembleur est le logiciel qui va transformer notre **code source** (une longue suite de mnémoniques) en données binaires que le processeur va comprendre et tenter d'exécuter.

```
z80asm -ofile -f[b|m|h] -l[file] -s[n|a] -e<num>
                                 -x -v -dsymbol ... file ...

A maximum of 512 source files is allowed. If the filename of a
source doesn't have an extension the default extension ".asm"
will be concatenated. Source file names may have a path, the
maximum length of a full qualified filename is 2048 characters.
For relative paths the extension must be used, because all
characters after a "." would be used as extension!

Option o:
To override the default name of the output file. Without
this option the name of the output file becomes the name of
the input file, but with the extension ".bin" or ".hex". The
output file may have a path, the maximum length is limited to
2048 characters.

Option f:
Format of the output file:
        -fb -> binary file
        -fm -> binary file with Mostek header
```

```
        -fh -> Intel hex

Option l:
Without this option no list file will be generated. With -l
a list file with the name of the source file but extension
".lis" will be generated. An optional file name with path
(2048 characters maximum) may be added to this option.

Option s:
This option writes the unsorted symbol table (-s), sorted by
name (-sn) or sorted by address (-sa) into the list file. This
option only works together with option -l.

Option e:
Set maximal symbol lenght to <num>, the default is 8.

Option x:
Don't output data in pass 2 into object file for DEFS. This
only works if unallocated data isn't followed by any code or
initialised data!  Useful for CP/M BIOS's, where unallocated
data doesn't need to be part of the system image, if the complete
image won't fit on the system tracks.

Option v:
Verbose operation of the assembler.

Option d:
This option predefines symbols with a value of 0.
The number of this option is not limited in the command line.

Pseudo Operations:

Definition of symbols and allocation of memory:
         ORG    <expression>    - set program address
<symbol> EQU    <expression>    - define constant symbol
<symbol> DEFL   <expression>    - define variable symbol
<symbol> DEFB   <exp,'char',..> - write bytes in memory
<symbol> DEFW   <exp,exp..>     - write words (16 bits) in memory
<symbol> DEFM   <'string'>      - write character string in memory
<symbol> DEFS   <expression>    - reserve space in memory

Conditional assembly:
IFDEF   <symbol>        - assemble if symbol defined
IFNDEF  <symbol>        - assemble if symbol not defined
IFEQ    <exp1,exp2>     - assemble if equal
IFNEQ   <exp1,exp2>     - assemble if not equal
ELSE                    - else for all conditionals
ENDIF                   - end of conditional assembly
```

```
Manipulation of list file:
PAGE    <expression>    - number of lines/page
EJECT                   - skip to new page
LIST                    - listing on
NOLIST                  - listing off
TITLE   <'string'>      - define title for page header

Others:
INCLUDE <filename>      - include another source file
PRINT   <'string'>      - print string to stdout in pass one

Operator Precedence for the node based parser from Didier Derny:
        unary + - ~
        * / %
        + -
        < >
        &
        ^
        |
        ( )
```

Certaines de ces options méritent quelques explications.

L'option `-l` demande un listing dans lequel nous pourrons lire les messages d'erreur. L'option `-fh` demande un fichier compréhensible par le simulateur.

4.2 Simulateur

Une fois que notre code est écrit, puis assemblé avec succès, il est temps de tenter de le faire fonctionner. N'ayant pas (ou pas encore ?) de véritable Z80 sous la main, nous allons utiliser un simulateur : `z80sim`.

```
usage:  z80sim -z -8 -s -l -i -u -m val -f freq -x filename
        -z = emulate Zilog Z80
        -8 = emulate Intel 8080
        -s = save core and CPU
        -l = load core and CPU
        -i = trap on I/O to unused ports
        -u = trap on undocumented instructions
        -m = init memory with val (00-FF)
        -f = CPU clock frequency freq in MHz
        -x = load and execute filename
```

Une fois l'émulateur lancé, on a à notre disposition quelques commandes permettant d'agir sur le programme qui tourne :

```
r filename[,address]      read object into memory
d [address]               dump memory
l [address]               list memory
m [address]               modify memory
f address,count,value     fill memory
v from,to,count           move memory
p address                 show/modify port
g [address]               run program
t [count]                 trace program
return                    single step program
x [register]              show/modify register
x f<flag>                 modify flag
b[no] address[,pass]      set soft breakpoint
b                         show soft breakpoints
b[no] c                   clear soft breakpoint
h [address]               show history
h c                       clear history
z start,stop              set trigger adr for t-state count
z                         show t-state count
c                         measure clock frequency
s                         show settings
! command                 execute UNIX command
q                         quit
```

5 vt102

Première étape, utiliser les séquences d'échappement de la `vt102` pour faire de la démo dans un *xterm*. Un bon TP pour la suite, à l'université de Rouleau de Printemps, déja programmée pour fin avril 2019..

```
; ----------------------------------------
;        ****** module vt100 ******

CLS:
        LD      HL,esc_cls
        CALL    OUTSTR
        RET

esc_cls:        DEFB    1BH,'[3J',1BH,'[H',1BH,'[2J',0

; ----------------------------------------
```

6 Et après ?

Et après, il reste beaucoup de choses trop cool à faire.

Pour monter d'un niveau, il faudra *patcher* un peu le simulateur pour ajouter des périphériques virtuels, mais ça n'est pas insurmontable.

— envoyer du MIDI ;
— écrire un système de fichiers ;
— gérer un affichage graphique.

7 Conclusion

Bloub...

Index