

Introduction à la programmation assembleur pour gameboy

Une chose à comprendre:

Programmer en assembleur, c'est comme écrire à la main le fichier binaire que lit la gameboy.

Il faut indiquer où l'on écrit dans la mémoire.

Chaque ligne de code correspond à une case dans la mémoire.

Ecrire à certains emplacements spécifiques de la mémoire permet d'interagir avec le hardware.

Une chose à comprendre:

Programmer en assembleur, c'est comme écrire à la main le fichier binaire que lit la gameboy.

Il faut indiquer où l'on écrit dans la mémoire.

Chaque ligne de code correspond à une case dans la mémoire.

Ecrire à certains emplacements spécifiques de la mémoire permet d'interagir avec le hardware.

Il n'y a pas de contexte (toutes les "variables" sont globales).

Il n'y a pas de type (tout est nombre).

Une chose a comprendre:

Programmer en assembleur, c'est comme écrire à la main le fichier binaire que lit la gameboy.

Il faut indiquer où l'on écrit dans la mémoire.

Chaque ligne de code correspond à une case dans la mémoire.

Ecrire à certains emplacements spécifiques de la mémoire permet d'interagir avec le hardware.

Il n'y a pas de contexte (toutes les "variables" sont globales).

Il n'y a pas de type (tout est nombre).

Il n'y a pas de boucles.

Il n'y a pas de if.

Il y a des jumps conditionnels.

Début	Fin	Description
\$0000	\$7FFF	Premiers 32ko du ROM de la cartouche
\$8000	\$9FFF	8ko de VRAM
\$A000	\$BFFF	8ko de RAM interne à la cartouche (pour les sauvegardes, si il y en a)
\$C000	\$DFFF	8ko de RAM de travail
\$E000	\$FDFF	Echo des 8ko de RAM précédents (miroir)
\$FE00	\$FE9F	OAM (sorte de VRAM supplémentaire)
\$FEA0	\$FEFF	Rien :)
\$FF00	\$FF7F	Registres de configuration pour le GPU et l'APU
\$FF80	\$FFFE	Petit morceau de RAM supplémentaire (HRAM)
\$FFFF		Registre spécial concernant les interruptions

Les variables:

Premier type de variable, les registres.

Registres de travail: A, B, C, D, E, H, L

Registres spéciaux: F, PC, SP

Les variables:

Premier type de variable, les registres.

Registres de travail: A, B, C, D, E, H, L

Registres spéciaux: F, PC, SP

Chaque registre de travail peut contenir UN octet (8 bit).

En cas de besoin, on peut combiner deux registres pour avoir un registre de 16 bits. On peut utiliser les paires AF, BC, DE, et HL.

Note: Attention avec AF, si l'on utilise plus de 12 bits dessus certaines fonctions risquent de se comporter bizarrement, encore plus si on utilise les 16 bits. De manière générale il faut éviter le plus possible de manipuler des registres de 16 bits.

Les variables:

Les registres particuliers:

Les variables:

Les registres particuliers:

-A: Accumulateur

Les résultats des calculs sont stockés dans A.

Les variables:

Les registres particuliers:

-A: Accumulateur

Les résultats des calculs sont stockés dans A.

-F: Flags

Contient des indications sur le résultat du dernier calcul, réutilisé par certaines instructions.

Les flags sont des bits du registre F. Ils ont des petits noms.

Z: vaut 1 si le résultat du dernier calcul est 0.

N: vaut 1 si le résultat du dernier calcul est négatif.

C: vaut 1 si le résultat du dernier calcul a overflow (Carry).

Les variables:

Les registres particuliers:

-A: Accumulateur

Les résultats des calculs sont stockés dans A.

-F: Flags

Contient des indications sur le résultat du dernier calcul, réutilisé par certaines instructions.

Les flags sont des bits du registre F. Ils ont des petits noms.

Z: vaut 1 si le résultat du dernier calcul est 0.

N: vaut 1 si le résultat du dernier calcul est négatif.

C: vaut 1 si le résultat du dernier calcul a overflow (Carry).

-HL: On l'utilise pour stocker les adresses, mais c'est plus une convention.

En revanche certaines instructions ne fonctionnent qu'avec ce registre.

Les variables:

Les registres particuliers:

-A: Accumulateur

Les résultats des calculs sont stockés dans A.

-F: Flags

Contient des indications sur le résultat du dernier calcul, réutilisé par certaines instructions.

Les flags sont des bits du registre F. Ils ont des petits noms.

Z: vaut 1 si le résultat du dernier calcul est 0.

N: vaut 1 si le résultat du dernier calcul est négatif.

C: vaut 1 si le résultat du dernier calcul a overflow (Carry).

-HL: On l'utilise pour stocker les adresses, mais c'est plus une convention.

En revanche certaines instructions ne fonctionnent qu'avec ce registre.

-SP: Stack Pointer

16 bits, adresse de la prochaine entrée de la stack (on y reviendra)

Les variables:

Les registres particuliers:

-A: Accumulateur

Les résultats des calculs sont stockés dans A.

-F: Flags

Contient des indications sur le résultat du dernier calcul, réutilisé par certaines instructions.

Les flags sont des bits du registre F. Ils ont des petits noms.

Z: vaut 1 si le résultat du dernier calcul est 0.

N: vaut 1 si le résultat du dernier calcul est négatif.

C: vaut 1 si le résultat du dernier calcul a overflow (Carry).

-HL: On l'utilise pour stocker les adresses, mais c'est plus une convention.

En revanche certaines instructions ne fonctionnent qu'avec ce registre.

-SP: Stack Pointer

16 bits, adresse de la prochaine entrée de la stack (on y reviendra)

-PC: Program Counter

16 bits, adresse de la prochaine instruction à exécuter (on y reviendra aussi, c'est important)

Les variables:

Les registres particuliers:

-A: Accumulateur

Les résultats des calculs sont stockés dans A.

-F: Flags

Contient des indications sur le résultat du dernier calcul, réutilisé par certaines instructions.

Les flags sont des bits du registre F. Ils ont des petits noms.

Z: vaut 1 si le résultat du dernier calcul est 0.

N: vaut 1 si le résultat du dernier calcul est négatif.

C: vaut 1 si le résultat du dernier calcul a overflow (Carry).

-HL: On l'utilise pour stocker les adresses, mais c'est plus une convention.

En revanche certaines instructions ne fonctionnent qu'avec ce registre.

-SP: Stack Pointer

16 bits, adresse de la prochaine entrée de la stack (on y reviendra)

-PC: Program Counter

16 bits, adresse de la prochaine instruction à exécuter (on y reviendra aussi, c'est important)

Les variables:

Les pointeurs et le Heap.

Avec 7 registres on ne va pas aller loin.

Le heap est une zone de la mémoire ou l'on peut écrire et lire.

On peut donc y stocker des valeurs. Mais pour ca il faut connaitre leur adresse.

L'assembleur permet d'utiliser des constantes pour nomer des adresse et eviter de donner les adresse en hexa (BallX c'est quand meme mieux que \$C001).

On accède à la valeur d'une variable a partir de son adresse en la déréférençant.

Syntaxe: `(\$C001)` ou `(BallX)`.

Début	Fin	Description
\$0000	\$7FFF	Premiers 32ko du ROM de la cartouche
\$8000	\$9FFF	8ko de VRAM
\$A000	\$BFFF	8ko de RAM interne à la cartouche (pour les sauvegardes, si il y en a)
\$C000	\$DFFF	8ko de RAM de travail
\$E000	\$FDFF	Echo des 8ko de RAM précédents (miroir)
\$FE00	\$FE9F	OAM (sorte de VRAM supplémentaire)
\$FEA0	\$FEFF	Rien :)
\$FF00	\$FF7F	Registres de configuration pour le GPU et l'APU
\$FF80	\$FFFE	Petit morceau de RAM supplémentaire (HRAM)
\$FFFF		Registre spécial concernant les interruptions

Les variables:

ATTENTION:

Il n'y a aucun système de contexte. Les variables, que ce soit les registres ou le heap, sont partagées!
Il est très facile de corrompre les variables d'un autre bout du programme en les écrasant.

```
DEF MA_FONCTION_1:
```

```
  A ← B
```

```
DEF MA_FONCTION_2:
```

```
  A ← B
```

```
  B ← 2
```

```
  MA_FONCTION_1
```

```
  PRINT A
```

```
B ← 3
```

```
MA_FONCTION_2
```

Ce pseudo code donne 2, et pas 3!

Les instructions:

Les instructions agissent presque toutes sur l'accumulateur (registre A).

Les instructions (presque tout le temps) des opérandes (~ arguments).

La nature des opérandes à de l'importance. Une instruction ne réagit pas de la même façon avec des registre ou des adresse déréférencées...

Les instructions:

- ld: assigne une valeur à un registre ou a un emplacement en mémoire.

``ld a,b` \Leftarrow a \leftarrow b`

``ld a,(ADDR)` \Leftarrow a \leftarrow *ADDR`

``ld (ADDR),a` \Leftarrow *ADDR \leftarrow a`

``ld (ADDR1),(ADDR2)` NE MARCHE PAS !!!`

Les instructions:

- ld: assigne une valeur à un registre ou a un emplacement en mémoire.

`ld a,b`	<=>	$a \leftarrow b$
`ld a,(ADDR)`	<=>	$a \leftarrow *ADDR$
`ld (ADDR),a`	<=>	$*ADDR \leftarrow a$
`ld (ADDR1),(ADDR2)` NE MARCHE PAS !!!		

- add: additionne une valeur à l'accumulateur.

`add 8`	<=>	$a += 8$
`add \$2F`	<=>	$a += 0x2F$
`add b`	<=>	$a += b$

Les instructions:

- ld: assigne une valeur à un registre ou a un emplacement en mémoire.

`ld a,b`	<=>	$a \leftarrow b$
`ld a,(ADDR)`	<=>	$a \leftarrow *ADDR$
`ld (ADDR),a`	<=>	$*ADDR \leftarrow a$
`ld (ADDR1),(ADDR2)` NE MARCHE PAS !!!		

- add: additionne une valeur à l'accumulateur.

`add 8`	<=>	$a += 8$
`add \$2F`	<=>	$a += 0x2F$
`add b`	<=>	$a += b$

- xor, and, sub, or, ect... comme add (Pas de multiplication)

Les instructions:

- ld: assigne une valeur à un registre ou a un emplacement en mémoire.

`ld a,b`	<=>	$a \leftarrow b$
`ld a,(ADDR)`	<=>	$a \leftarrow *ADDR$
`ld (ADDR),a`	<=>	$*ADDR \leftarrow a$
`ld (ADDR1),(ADDR2)` NE MARCHE PAS !!!		

- add: additionne une valeur à l'accumulateur.

`add 8`	<=>	$a += 8$
`add \$2F`	<=>	$a += 0x2F$
`add b`	<=>	$a += b$

- xor, and, sub, or, ect... comme add (Pas de multiplication)

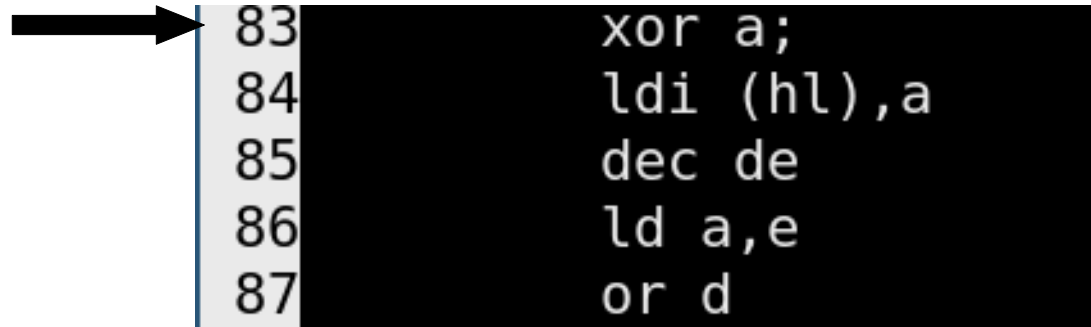
- inc, dec: incremente/decremente un registre (Attention, ne marche pas sur les registre 16 bits).

inc a	<=>	$a ++$
dec a	<=>	$a --$

Les structures de controle:

Retour sur le Program Counter

Le program counter est l'adresse de la prochaine instruction a executer.

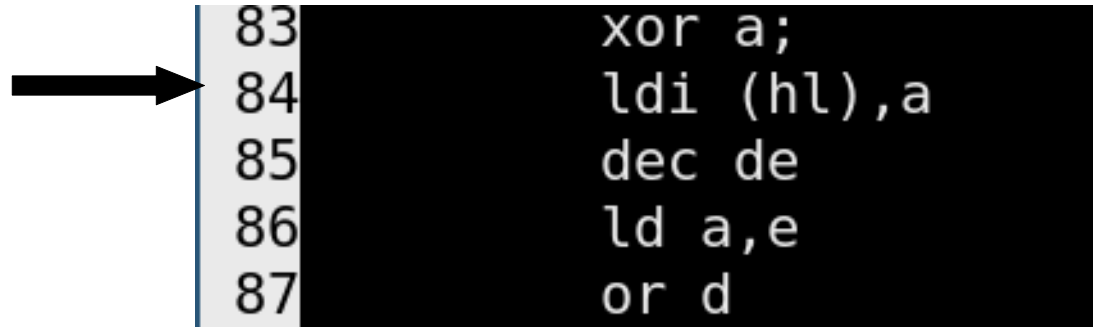


```
83  xor a;  
84  ldi (hl),a  
85  dec de  
86  ld a,e  
87  or d
```

Les structures de controle:

Retour sur le Program Counter

Le program counter est l'adresse de la prochaine instruction a executer.

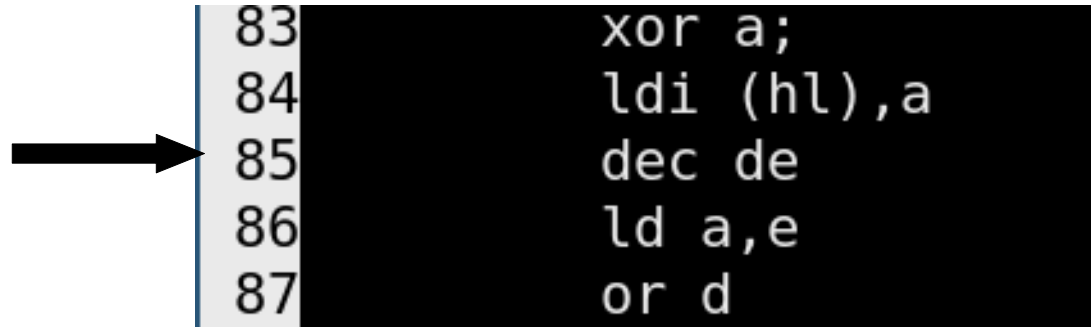


83	xor a;
84	ldi (hl),a
85	dec de
86	ld a,e
87	or d

Les structures de controle:

Retour sur le Program Counter

Le program counter est l'adresse de la prochaine instruction a executer.

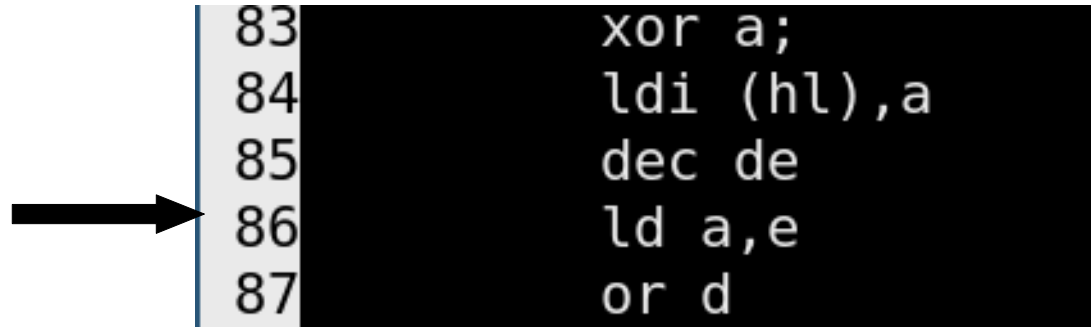


83	xor a;
84	ldi (hl),a
85	dec de
86	ld a,e
87	or d

Les structures de controle:

Retour sur le Program Counter

Le program counter est l'adresse de la prochaine instruction a executer.

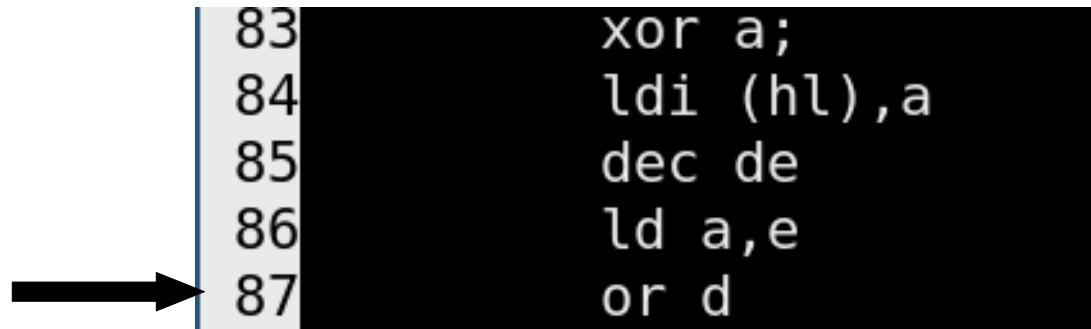


83	xor a;
84	ldi (hl),a
85	dec de
86	ld a,e
87	or d

Les structures de controle:

Retour sur le Program Counter

Le program counter est l'adresse de la prochaine instruction a executer.



A diagram showing a list of assembly instructions with their corresponding line numbers. A black arrow points to the instruction at line 87, indicating it is the next instruction to be executed. The instructions are: 83 xor a;, 84 ldi (hl),a, 85 dec de, 86 ld a,e, and 87 or d.

83	xor a;
84	ldi (hl),a
85	dec de
86	ld a,e
87	or d

Les structures de controle:

Si l'on veut modifier le déroulement du programme, on peut utiliser l'instruction `jp` pour modifier la valeur du PC.

On peut utiliser des labels pour indiquer un emplacement dans le programme au lieu de donner l'address hardcodé (plus explicite, pas besoin de la modifier a chaque fois qu'on modifier le programme, ect...)



```
40      nop; adviced from nintendo. nop j
41      jp start
```

////////////////////////////////////

```
52  start:
53      di
54      ld sp,$FFF4
55      xor a
56      ldh ($26),a
```

Les structures de controle:

Si l'on veut modifier le déroulement du programme, on peut utiliser l'instruction `jp` pour modifier la valeur du PC.

On peut utiliser des labels pour indiquer un emplacement dans le programme au lieu de donner l'address hardcodé (plus explicite, pas besoin de la modifier a chaque fois qu'on modifier le programme, ect...)



```
40      nop; adviced from nintendo. nop j
41      jp start
```

////////////////////////////////////

```
52  start:
53      di
54      ld sp,$FFF4
55      xor a
56      ldh ($26),a
```

Les structures de controle:

Si l'on veut modifier le déroulement du programme, on peut utiliser l'instruction `jp` pour modifier la valeur du PC.

On peut utiliser des labels pour indiquer un emplacement dans le programme au lieu de donner l'address hardcodé (plus explicite, pas besoin de la modifier a chaque fois qu'on modifier le programme, ect...)

```
40      nop; adviced from nintendo. nop j
41      jp start
```

////////////////////////////////////



```
52  start:
53      di
54      ld sp,$FFF4
55      xor a
56      ldh ($26),a
```

Les structures de controle:

Si l'on veut modifier le déroulement du programme, on peut utiliser l'instruction `jp` pour modifier la valeur du PC.

On peut utiliser des labels pour indiquer un emplacement dans le programme au lieu de donner l'address hardcodé (plus explicite, pas besoin de la modifier a chaque fois qu'on modifier le programme, ect...)

```
40      nop; adviced from nintendo. nop j
41      jp start
```

////////////////////////////////////



```
52  start:
53      di
54      ld sp,$FFF4
55      xor a
56      ldh ($26),a
```

Les structures de controle:

On peut rajouter des conditions sur le saut.

Le plus courant est de faire un saut si le résultat du dernier calcul est non nul (Z=0).

C'est fait avec l'instruction `jr nz,ADDR`

```
111      ld b,4
112      xor a
113  loop:
114      ld (hl),a
115      add 8
116      inc l
117      dec b
118      jr nz,loop
```


Les structures de controle:

On peut rajouter des conditions sur le saut.

Le plus courant est de faire un saut si le résultat du dernier calcul est non nul (Z=0).

C'est fait avec l'instruction `jp nz,ADDR`

```
111      ld b,4
112      xor a
113  loop:
114      ld (hl),a
115      add 8
116      inc l
117      dec b
118      jr nz,loop
```

$B \leftarrow 4$

$A \leftarrow 0$ // $A = A \text{ xor } A$

Loop:

$*hl \leftarrow A$

$A += 8$

$l++$ // $hl++$

$b--$

si $b \neq 0$, goto Loop

Les structures de controle:

On peut rajouter des conditions sur le saut.

Le plus courant est de faire un saut si le résultat du dernier calcul est non nul (Z=0).

C'est fait avec l'instruction `jp nz,ADDR`

```
111      ld b,4
112      xor a
113  loop:
114      ld (hl),a
115      add 8
116      inc l
117      dec b
118      jr nz,loop
```

$B \leftarrow 4$

$A \leftarrow 0$

do{

$*hl \leftarrow A$

$A += 8$

$l++$

// hl ++

$b --$

} while($b \neq 0$)

Les structures de controle:

On peut rajouter des conditions sur le saut.

Le plus courant est de faire un saut si le résultat du dernier calcul est non nul (Z=0).

C'est fait avec l'instruction `jp nz,ADDR`

```
111      ld b,4
112      xor a
113 loop:
114      ld (hl),a
115      add 8
116      inc l
117      dec b
118      jr nz,loop
```

```
A ← 0
For (b=4; b != 0; b--){
    *hl ← A
    A += 8
    l++           // hl ++
}
```

Les structures de controle:

NOTE COORECTIVE: `jr` fonctionne comme `jp`, mais pour les sauts de moins de 7 bit de longueur.
En contrepartie `jr` est plus rapide.

```
111      ld b,4
112      xor a
113  loop:
114      ld (hl),a
115      add 8
116      inc l
117      dec b
118      jr nz,loop
```

Les structures de controle:

Instruction `cp`.

`cp` revient a faire `sub`, mais sans modifier l'accumulateur. En revanche, les FLAGS sont modifiés.

```
368 cp 42
369 jp nz, neq
370 ; do stuff
371 neq:
```

```
If (A == 42){
  Do stuff
}
```

```
368 cp 42
369 jp z, eq
370 ; do stuff
371 eq:
372
```

```
If (A != 42){
  Do stuff
}
```

Les structures de controle:

Instruction ``cp``.

``cp`` revient a faire ``sub``, mais sans modifier l'accumulateur. En revanche, les FLAGS sont modifiés.

```
368 cp 42
369 jp c, inf
370 ; do stuff
371 inf:
372
```

```
If (A >= 42){
  Do stuff
}
```

```
368 cp 42
369 jp nc, sup
370 ; do stuff
371 sup:
372
```

```
If (A <= 42){
  Do stuff
}
```

PAUSE CODE

- Dans le fichier canvas.s, la gameboy est initialisé et les sprites sont chargés.
- Vous pouvez commencer a coder dans l'emplacement réservé.
- Ce code est exécuter entre chaque refresh de l'écran. La balle est affichée en (BallX, BallY) sur l'écran.
- Pouvez décommenter une la section plus bas pour exécuter du code quand la touche UP ou la touche DOWN est pressée.
- Vous pouvez faire du son avec les instructions
`call lowbeep`
et
`call hibeep`
- compilez avec `./make-gb.sh canvas.s`, et lancer avec l'émulateur le fichier `canvas.s.gb` pour tester.

PAUSE CODE

Petite instruction pratique que j'ai oublié: si vous voulez vérifier la valeur d'un bit en particulier:

``bit $3,b`` test le 3ième bit du registre b. Ensuite on peut utiliser ``jp z,addr`` par exemple.

Les fonctions

- C'est quoi une fonction?

Les fonctions

- C'est quoi une fonction?

C'est un bout de code qui peut être appelé "à peu près" n'importe où, n'importe quand.

Les fonctions

- C'est quoi une fonction?

C'est un bout de code qui peut être appelé "à peu près" n'importe où, n'importe quand.

Problèmes:

- Comment passer des arguments?

- Comment faire pour avoir un code qui fonctionne dans n'importe quel contexte sans casser ce qu'il se passe autour?

- Accessoirement, comment retourner une valeur?

Les fonctions

Passer des arguments, retourner une valeur:

Les variables sont globale. Techniquement on passe TOUTES les variables en argument...

En pratique on ne connaît pas le contexte dans lequel la fonction est appelé, il faut donc utiliser des convention.

Par exemple, on peut dire que le registre B est l'argument de la fonction. Cela veut dire qu'avant d'appeler la fonction, il faut mettre son argument dans le registre B.

Les registres dans lesquels on place les arguments sont choisis par convention. C'est mieux d'avoir la même convention pour TOUT le code.

Comme A est l'accumulateur, et que la plupart des opérations agissent sur A, on l'utilise pour retourner une valeur.

Les fonctions

Les fonctions, comment ça marche en fait?

Les fonctions

Les fonctions, comment ça marche en fait?

- C'est juste un jump, sauffff que.....

Les fonctions

Les fonctions, comment ça marche en fait?

- C'est juste un jump, sauffff que.....

Comment on fait pour revenir a notre point de départ a la fin de la fonction?

Les fonctions

Les fonctions, comment ça marche en fait?

- C'est juste un jump, sauffff que.....

Comment on fait pour revenir a notre point de départ a la fin de la fonction?

Vous vous souvenez du PC? Il suffit de le stocker quelque part pour se souvenir d'où on est partie, et donc le récupérer pour retourner au point de départ à la fin de la fonction.

saufffff que.....

Les fonctions

Les fonctions, comment ça marche en fait?

- C'est juste un jump, sauffff que.....

Comment on fait pour revenir a notre point de départ a la fin de la fonction?

Vous vous souvenez du PC? Il suffit de le stocker quelque part pour se souvenir d'où on est partie, et donc le récupérer pour retourner au point de départ à la fin de la fonction.
saufffff que.....

Comment faire quand on utilise plusieurs fonction? Avec et si on fait une fonction récursive? On va écraser le PC sauvegardé et on ne pourra plus sortir de la première fonction!

Les fonctions

La stack (la pile).

C'est une zone mémoire, sur le principe du First In First Out. Le sommet de la stack est désigné par l'adresse dans le registre SP (Stack Pointer).

Quand on met une valeur dans la stack, on décrépente SP, quand on retire une valeur, on incrémente SP.
(La stack a la tête en bas)

Ces opérations sont faites avec `push` et `pop`:

`push af`	<=>	place AF dans la stack et met a jour SP
`push hl`	<=>	place HL dans la stack et met a jour SP
`pop hl`	<=>	place le sommet de la pile dans HL et met a jour SP
`pop af`	<=>	place le sommet de la pile dans AF et met a jour SP

Les fonctions

Appeler une fonction:

`call function`	\Leftrightarrow	`push pc` `jump function`
`ret`	\Leftrightarrow	`pop pc`

Les fonctions

Début	Fin	Description
\$0000	\$7FFF	Premiers 32ko du ROM de la cartouche
\$8000	\$9FFF	8ko de VRAM
\$A000	\$BFFF	8ko de RAM interne à la cartouche (pour les sauvegardes, si il y en a)
\$C000	\$DFFF	8ko de RAM de travail
\$E000	\$FDFF	Echo des 8ko de RAM précédents (miroir)
\$FE00	\$FE9F	OAM (sorte de VRAM supplémentaire)
\$FEA0	\$FEFF	Rien :)
\$FF00	\$FF7F	Registres de configuration pour le GPU et l'APU
\$FF80	\$FFFE	Petit morceau de RAM supplémentaire (HRAM) ← Stack: 128 octets :'(
\$FFFF		Registre spécial concernant les interruptions

Les fonctions

Pour la culture:

En C, le compilateur utilise la stack pour passer des arguments a une fonction.
La fonction les récupères en mémoire à l'emplacement $SP+1+n^{\circ}\text{argument}$.

Ici on a pas la place pour ca.

On a pas la place pour faire des fonctions récursives non plus.

Les fonctions

Un dernier problème: comment on fait pour programmer une fonction sans tout casser le reste du code?

Les fonctions

Un dernier problème: comment on fait pour programmer une fonction sans tout casser le reste du code?

Pour programmer il faut des registres.

Les registres sont globaux.

Les fonctions

Un dernier problème: comment on fait pour programmer une fonction sans tout casser le reste du code?

Pour programmer il faut des registres.

Les registres sont globaux.

Comment je fait pour pas ecraser les valeurs supers importantes utiliser par le code qui m'appel et don je ne sais rien?

Les fonctions

Encore des conventions: registres caller saved et callee saved.

Les fonctions

Encore des conventions: registres caller saved et callee saved.

Pour pouvoir utiliser un registre sans risque, il faut sauvegarder sa valeur (traditionnellement dans la stack) AVANT d'appeler la fonction et les recharger APRES le retour de la fonction, ou JUSTE APRES l'appel de la fonction et JUSTE AVANT le retour (DANS la fonction).

```
287  
288     push b  
289     call function  
290     pop b  
291
```

B est caller saved

```
293 function:  
294     push c  
295     ...  
296     pop c  
297     ret
```

C est callee saved

Les fonctions

Encore des conventions: registres caller saved et callee saved.

Un registre caller saved est toujours utilisable dans une fonction.

Un registre callee saved n'est jamais modifié par l'appel d'une fonction.

```
287  
288      push b  
289      call function  
290      pop b  
291
```

B est caller saved

```
293 function:  
294      push c  
295      ...  
296      pop c  
297      ret
```

C est callee saved

Les fonctions

Les conventions d'appel:

- Comment on passe les arguments?
- Quels registres sont callee saved, quels registres sont caller saved?

~~Les fonctions~~ Les interruptions

Les interruptions sont comme des fonctions, SAUF QUE

- Elles sont appelées par le processeur, pas par le programme.
- Elles peuvent être appelées “n’importe quand” (on peut les désactiver au besoin, mais le programme ne sais pas si il vient d’exécuter une interruption)
- L’instruction de retour est ``reti`` (ret interruption) et pas ``ret``

~~Les fonctions~~ Les interruptions

Les interruptions sont comme des fonctions, SAUF QUE

- Elles sont appelées par le processeur, pas par le programme.
- Elles peuvent être appelées “n’importe quand” (on peut les désactiver au besoin, mais le programme ne sais pas si il vient d’exécuter une interruption)
- L’instruction de retour est `reti` (ret interruption) et pas `ret`

DONC:

TOUT LES REGISTRES sont callee saved. (Le reste du programme ne peut pas sauvegarder les registres parce qu’il ne sais pas qu’il doit le faire)

TOUT LES REGISTRES MODIFIEE DOIVENT ETRE SAUVEGARDE.