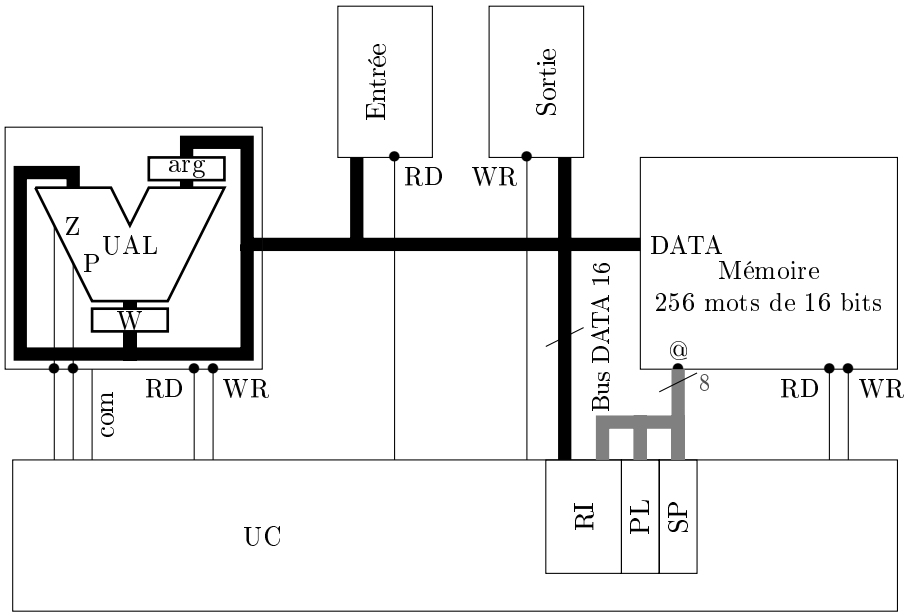
Microprocesseur et langage assembleur

Un langage assembleur est indissociable du microprocesseur pour lequel il est prévu. On travaillera sur un microprocesseur simplifié et le langage assembleur qui va avec. Les deux sont suffisamment développés et proche de la réalité pour comprendre les bases des vrais langages assembleurs et les vrais processeurs.

μP = microprocesseur

Structure d'ensemble



Unité arithméthique et logique (UAL)

L'UAL fait les calculs.

- W est le registre de travail (Work). Le résultat des calculs est toujours écrit dans W.
- Les calculs ont toujours pour opérandes *arg* d'abord, et W en second. Par exemple, une opération `ADD` fait le calcul $W = W + arg$
- Z et P informent sur le résultat d'un calcul. Z indique que le résultat est $= 0$; P indique que le résultat est ≥ 0. On les appelle bits d'état.
- L'UAL est piloté par l'UC grâce aux commandes RD, WR et com.
  - RD (Read) ordonne à l'UAL de placer le contenu de W sur le bus DATA,
  - WR (Write) ordonne à l'UAL de recopier dans *arg* le contenu du bus DATA
  - com indique à l'UAL l'opération a effectuer

La mémoire

C'est un ensemble de 256 mots de 16 bits. On indique le mot auquel on s'adresse en indiquant son **adresse**, c'est à dire un nombre sur 8 bits, porté par le bus d'adresse (que l'on note souvent @)

La mémoire est pilotée par l'UC par l'intermédiaire de RD, WR et @

- RD ordonne à la mémoire de placer le contenu du mot à l'adresse @ sur le bus DATA
- WR ordonne à la mémoire de placer le contenu du bus DATA dans le mot à l'adresse @

Il est important de noter que la mémoire contiendra à la fois le programme et les données. On parle d'une architecture **Von Neumann**.

Entrée et sortie

Un mircroprocesseur doit pouvoir communiquer avec l'extérieur et a donc des entrées et sorties. Elles seront très simplifiées ici.

- RD met le contenu en entrée sur le bus DATA
- WR met le contenu du bus DATA en sortie

Unité de Commande (UC)

C'est elle qui envoie tous les signaux permettant de mener à bien chaque instruction du programme et le déroulement du programme. Pour cela, l'UC répète toujours à l'identique l'exécution d'un **cycle instruction**.

Cycle instruction : l'UC effectue les étapes suivantes

1) Placer sur le bus @ le contenu de PL. Lire la mémoire, *PL = Pointeur de ligne, contient la ligne en cours.* L'écrire dans RI.
2) RI = Registre instruction. Contient la ligne d'instruction en cours. L'UC va analyser le mot pour savoir quoi faire.
3) Selon le contenu de RI, UC génère tous les signaux nécessaires.

   Exemple : exécution de `ADD #3`

   i. UC place `3` sur Bus DATA et commande RD à UAL de sorte que `3` arrive dans *arg*,

   ii. UC commande ADD à UAL, de sorte que l'UAL fait le calcul $W = W + arg$.
4) UC incrémente PL pour être prêt pour l'instruction suivante.

SP (Stack Pointer) est le pointeur de pile. C'est plus compliqué, pas indispensable dans un premier temps. Détails plus loin.

Important : Le bus DATA relie tous les organes. Si UC écrit `3` sur le bus DATA, tous les organes le voient. Il est donc important que l'UC puisse dire :
- à qui s'adresse ce message. C'est le rôle des commandes WR.
- que personne d'autre ne doit écrire sur le bus en même temps. Les RD indiquent aux organes quand ils doivent écrire sur le bus.

Jeu d'instructions – μP de type RISC = Reduced Instruction Set Computer

Le *mu*P possède un **jeu d'instruction**, c'est à dire un ensemble de commande qu'il est capable de comprendre et d'exécuter. Une instruction aura
- un mot clé, par exemple `ADD` pour l'addition,
- éventuellement des arguments, par exempe dans `ADD @x`, l'argument est le contenu de `@x`.
- un codage, car l'instruction devra être stockée dans la mémoire du μP. Par exemple, `ADD @x` pourra être codée (dépend de la case mémoire correspondant à `@x`) : `1000001000010001`

Le microprocesseur envisagé ici utilise des mots de 16 bits. Chaque instruction sera donc traduite par un mot de 16 bits.

Remarque : Dans le μP, c'est bien le binaire qui sera utilisé. Mais écrire des lignes comme `1000001000010001` peut être très fastidieux. On pourra préférer l'écriture hexadécimale `0x8211`. On pourra même s'abstenir du `0x` s'il n'y a pas de risque de confusion.

Les types d'arguments

- Certains instructions, comme `HALT`, n'ont pas d'argument.
- D'autres instructions comme `STR` ne peuvent opérer que sur une adresse mémoire.

  Par exemple `STR @x` stocke le contenu du registre de travail dans la mémoire `@x`.

  Autre exemple, `JMP L` est un saut. L'argument est une étiquette qui désigne un numéro de ligne du programme.
- D'autres instructions comme `ADD` peuvent recevoir divers types d'arguments.
  - `ADD @x` : on ajoute le contenu de `@x`, dans ce cas l'argument désigne une adresse mémoire,
  - `ADD #3` : on ajoute la valeur `3`, dans ce cas l'argument est une **valeur littérale**.
  - `ADD POP` : on ajoute le dessus de la pile. *Voir plus loin l'explication sur la pile.*

On choisit de coder le type d'arguments sur 2 bits :
- `00` : pas d'argument,
- `01` : argument littéral comme `#3`,
- `10` : argument adresse, comme `@x` ou pour un saut,
- `11` : argument pris sur la pile.

Codage

Codage constant : Le processeur utilise des mots de 16 bits. On choisit de plus un codage de format constant.

opcode[6]	type[2]	argument[8]

- **opcode :** c'est le code désignant l'instruction demandée. Par exemple, `100000` est l'opcode pour `ADD`. L'opcode utilisera toujours les 6 premiers bits du mot.
- **type :** c'est le type d'argument. On l'a présenté dans l'encadré précédent.
- **argument :** c'est le code pour l'argument, 8 bits.

Exemple : `1000001000010001` → | `100000` | `10` | `00010001` |

- Le premier bloc `100000` désigne l'instruction `ADD`,
- le second bloc `10` désigne un argument de type adresse,
- le troisième bloc `00010001` correspond à la valeur binaire 17.

On a donc une instruction `ADD @17`. Par exemple, cela pourrait correspondre au cas où la variable x a été stockée à la case mémoire d'adresse `17`.

Remarque : Vous notez que l'on perd des choses en passant en binaire. En effet, on perd l'information disant que l'adresse `17` correspond à x. C'est ce genre de chose qui rend la lecture du binaire très difficile.

Instructions sans argument

- **HALT** | `000000` | `00` | `00000000` | Provoque l'arrêt du programme.
- **NOP** | `000001` | `00` | `00000000` |
  Instruction qui ne fait rien. Peut-être utilisée quand il faut attendre un certain temps.
- **POP** | `101111` | `00` | `00000000` | Place le dessus de la pile dans W.
- **PUSH** | `110000` | `00` | `00000000` | Place W sur le dessus de la pile.

Instructions UAL

Elles acceptes les quatre types d'argument. Elles agissent toutes sur le contenu du registre de travail W. Les instructions ne diffèrent que par leur opcode :

- **ADD :** | `100000` | `xx` | `xxxxxxxx` | $W + arg \to W$
- **SUB :** | `100001` | `xx` | `xxxxxxxx` | $W - arg \to W$
- **MUL :** | `100010` | `xx` | `xxxxxxxx` | $W \times arg \to W$
- **DIV :** | `100011` | `xx` | `xxxxxxxx` | $W \div arg \to W$
- **MOD :** | `100100` | `xx` | `xxxxxxxx` | $W \,\%\, arg \to W$
- **OR :** | `100101` | `xx` | `xxxxxxxx` | W or $arg \to W$
- **AND :** | `100110` | `xx` | `xxxxxxxx` | W and $arg \to W$
- **XOR :** | `100111` | `xx` | `xxxxxxxx` | W xor $arg \to W$
- **CMP :** | `101000` | `xx` | `xxxxxxxx` | $W - arg$
  Exceptionnellement, `CMP` ne modifie pas le contenu de W. Seuls les bits Z et P sont modifiés. Cela sert à préparer les instructions de saut qui exploitent Z et P.
- **MOV :** | `101001` | `xx` | `xxxxxxxx` | $arg \to W$
- **INV :** | `101010` | `xx` | `xxxxxxxx` | not $arg \to W$
- **NEG :** | `101011` | `xx` | `xxxxxxxx` | $-arg \to W$

Autres instructions

- **OUT** | `101100` | `xx` | `xxxxxxxx` |
  Place l'argument en sortie. Tout type d'argument accepté.
- **INP** | `101101` | `10` | `xxxxxxxx` |
  Place l'entrée dans la case mémoire spécifiée par l'adresse en argument.
- **STR** | `101110` | `10` | `xxxxxxxx` |
  (STORE) Place W dans la case mémoire spécifiée par l'adresse en argument.

Les instructions de saut

Le programme est lu une ligne après l'autre, dans l'ordre. Un saut permet de passer à une certaine ligne au lieu de la suivante. Parfois, le saut ne s'exécute que conditionnellement.

L'argument est toujours de type adresse et correspond à l'adresse mémoire à laquelle il faut sauter (le programme étant stocké dans la mémoire, un numéro de ligne est aussi une adresse)

- **JMP ou B ou GOTO :** `010000 | 10 | xxxxxxxx` Saut (Jump) à la ligne *arg*.
  On peut dire aussi *Branchement*, d'où le B.
- **BEQ :** `010001 | 10 | xxxxxxxx`
  Saut à la ligne *arg* **si** dernier calcul égal à zéro — si `bit Z == 1`
- **BNE :** `010010 | 10 | xxxxxxxx` Saut **si** dernier calcul non égal à zéro — si `Z != 0`
- **BGT :** `010011 | 10 | xxxxxxxx` Saut **si** dernier calcul > 0 — si `Z != 0 et P == 1`
- **BGE :** `010100 | 10 | xxxxxxxx` Saut **si** dernier calcul ≥ 0 — si `P == 1`
- **BLT :** `010101 | 10 | xxxxxxxx` Saut **si** dernier calcul < 0 — si `P == 0`
- **BLE :** `010110 | 10 | xxxxxxxx` Saut **si** dernier calcul ≤ 0 — si `Z == 0 ou P == 0`

Cas particulier des grands littéraux

Considérons l'instruction suivante : `ADD #151`

- `ADD` a l'opcode `100000`,
- on reconnaît un argument de type littéral, donc de code `01`,
- en binaire 151 s'écrit `10010111`

On obtient donc le code `100000 | 01 | 10010111`

Quelle taille peuvent atteindre les littéraux ? La mémoire contient des mot de 16 bits, il serait donc naturel de pouvoir utiliser des nombres de 16 bits. Mais dans le codage que l'on vient de présenter, on ne dispose que de 8 bits pour le littéral...

On décide donc que dans les cas où le littéral est trop long, exceptionnellement, l'instruction sera écrite dans deux cases mémoires successives.

Exemple : `ADD #945`

- `ADD` a l'opcode `100000`,
- on reconnaît un argument de type littéral, donc de code `01`,
- en binaire 945 s'écrit `1110110001`

On constate que le littéral prend trop de place pour 8 bits. On décide donc d'utiliser un 2e mot pour l'instruction. L'instruction sera alors codée :

`100000 | 01 | 11111111` `0000001110110001`

Quand l'argument est `11111111`, il faut lire le littéral dans le mot de 16 bits qui suit.

Remarque : Les différents μPs, et donc leurs langages assembleur, utilisent des façons diverses de régler ce type de problème. Il n'y a pas de méthode unique, seulement des standards qui peuvent s'imposer quand une idée, ou l'entreprise à l'origine de l'idée, rencontre du succès.

Pile

La pile doit être considérée comme une zone mémoire dans laquelle on place les résultats de calculs intermédiaires. On place un résultat sur la pile avec `PUSH` et on récupère ce résultat avec `POP`.

```
MOV @y
ADD #7
PUSH
MOV @x
ADD #2
MUL POP
```

Exemple : considérons le calcul `(x + 2) * (y + 7)`. Ce calcul serait traduit : On voit que le calcul `y + 7` a éta mis de côté avec `PUSH` le temps de faire le calcul `x + 2`. À la fin, `MUL POP` multiplie W (c'est à dire `x + 2`) avec le résultat gardé précédemment.

Je ne détaille pas plus ici. La pile fait l'objet de tout un cours en Terminale.

label DATA

On peut écrire une ligne, par exemple `x DATA 15` qui permet d'initialiser dès le début de programme une variable `x` avec la valeur `15`. Si on a placé ce code en 12e ligne programme, alors `x` sera placé à la 12e case mémoire (adresse 11).