Cours Assembleur AVR

Halim Djerroud
hdd@ai.univ-paris8.fr

7 février 2017

Table des matières

1 Microcontrôleur AVR

1.1 Architecture microcontrôleur AVR

L'architecture des microcontrôleur de AVR, est basé sur sur conception Harvard. Ce type d'architecture sépare physiquement la mémoire programme de la mémoire de données. L'accès à chacune des deux mémoires s'effectue via deux bus différents. L'avantage de ce type d'architecture contrairement a l'architecture Von Neumann, elle permet de transférer les données et les instructions à exécuter simultanément. Ainsi l'unité centrale de traitement (CPU) permet d'avoir accès simultanément a l'instruction en cours d'exécution et les données associées.

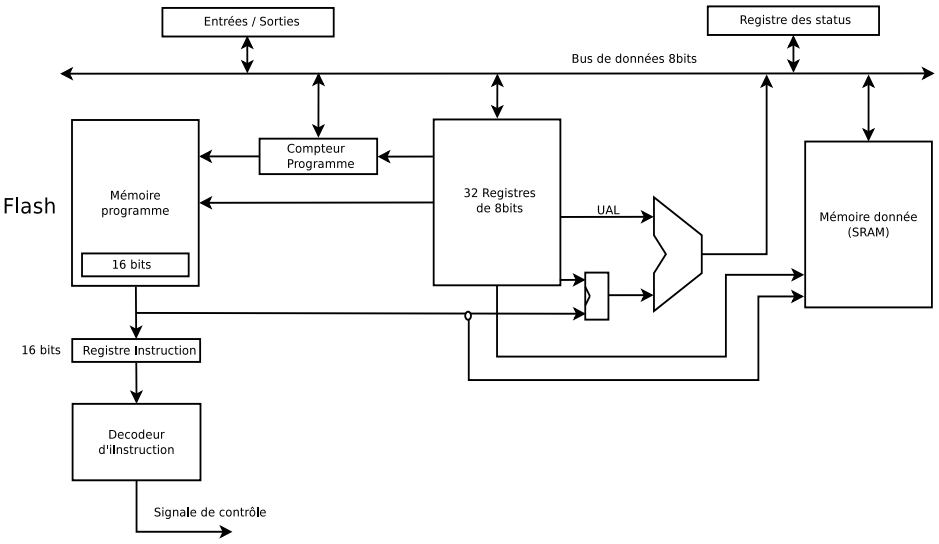
Dans l'architecture que nous avons choisi pour ce cours (ATmega 328) La mémoire programme est relié sur un bus 16 bits, par contre la mémoire de données qui ne compte qu'un bus de 8 bits.

Les microcontrôleur AVR sont conçu au tour de la technologie RISC (en anglais Reduced Instruction Set Computer), cette technologie consiste à déplacer les complexités majeures du hardware vers le software, alors que la technologie CISC (en anglais Complex Instruction Set Computer) fait exactement le contraire. La conception de la technologie RISC fait en sort de minimiser le nombre d'instructions disponibles ce qui a pour avantage de rendre la majeure partie des instructions exécutables en un seul cycle d'horloge, de plus un nombre d'instruction réduit permet de simplifier la conception des fonctions d'optimisation des compilateurs. Alors que l'architecture RISC offre une large gamme d'instructions ce qui permet effectivement de réduction du nombre d'instructions nécessaires pour exécuter le programme, mais augmente la complexité du interne microcontrôleur ce qui a pour effet d'augmenter moyennement le nombre de cycles d'horloge nécessaires pour exécuter une instruction. Dans ce cas, la fréquence de travail du système est réduite car il faut introduire une phase d'interprétation du code machine à travers des microcodes.

1.2 Diagramme microcontrôleur AVR

Dans l'architecture AVR le bus de donnée est de 8 bits. Comme nous avons déjà indiqué précédemment, les microcontrôleurs AVR respectent la conception Harvard, c'est-à-dire la mémoire programme est séparée de la mémoire données. Cette mémoire programme est de type flash, elle permet d'enregistrer des mots de 16 bits, un mot représente l'unité adressable. La mémoire programme est reliée directement au bus de données. Elle est reliée aussi à un registre qui stock l'instruction en cours d'exécution ce dernier est évidemment de taille de 16bits, ces instructions proviennent directement de la mémoire programme. Le numéro d'instruction en cours d'exécution qui doit être enregistré dans le registre d'instruction est indiqué dans un registre très important qui s'appelle Compteur

de Programme, ce dernier contient l'adresse de la prochaine instruction qui va être stocker dans le registre d'instruction.



L'une des tâches les plus importantes du microcontrôleur est de décoder l'instruction dans le registre d'instructions afin de contrôler les différents éléments du microcontrôleur, dans le schéma précédent nous avons représenté cette unité de contrôle dans un bloc que nous avons appelé Décodeur d'Instruction.

Afin de stocker les données temporaires ou résultats intermédiaires le microcontrôleur AVR prévoit un ensemble de 32 registres de 8 bits chacun. Ce bloc de registres est bien évidemment relié au bus de données. Il est aussi relié à la mémoire programme, ce qui est utile pour utiliser une donnée comme une adresse programme afin d'exécuter une instruction. Un autre lien existe avec compteur programme.

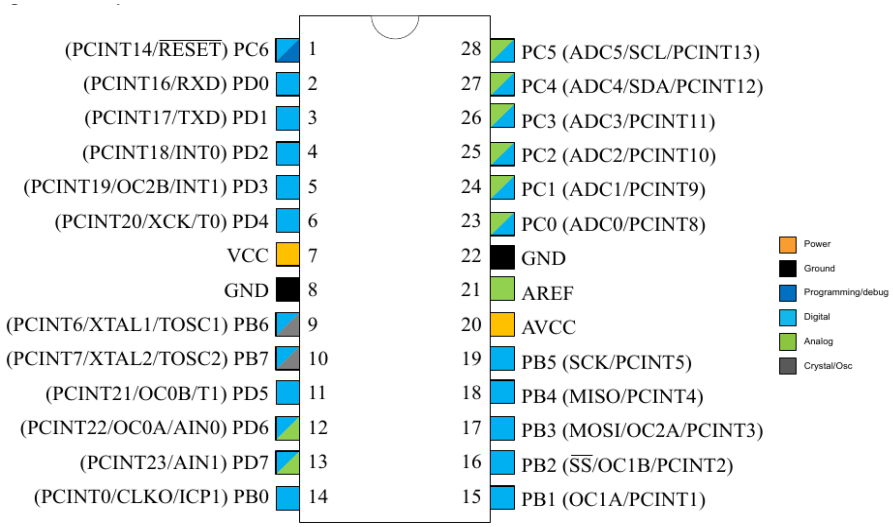
L'unité arithmétique et logique appelé communément UAL (en anglais ALU : arithmetic and logic unit) permet de réaliser les différents opérations comme son nom l'indique arithmétique tel que l'addition, la soustraction etc, et les opérations logiques tel que ET, OU, comparaison etc. Cette dernière possède deux entrées et une seul sortie qui est connectée directement au bus de données. La première entrée de l'UAL provient directement des registres de données et la second est multiplexé entre les registre de données et le registre d'instruction.

La mémoire de données est de type SRAM (en anglais : Static Random Access Memory) peux être adressée de deux façons différente, soir directement par une adresse qui provient du registre d'instruction, on appel se mode d'adressage

l'adressage direct, ou-bien via une adresse qui provient du bloc de registre, on appel se mode d'adressage l'adressage indirect.

Un autre registre aussi important appelé Registre d'état (en anglais Status Register)il est sur 8bits, ce registre permet de récolter certains informations sur l'état du microcontrôleur par exemple il permet de savoir si une division par zéro s'est produite ou par exemple une addition qui se termine avec un débordement etc.

1.3 La description des broches



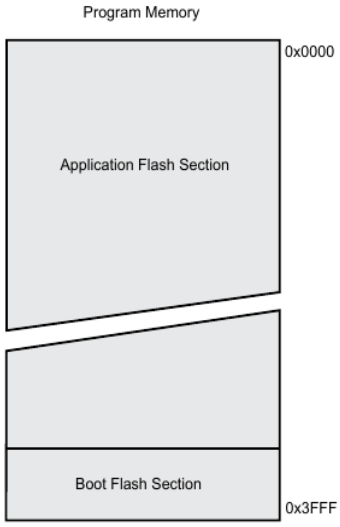
1.4 Caractéristiques

Le processeur AVR a une architecture Harvard c'est-à-dire que le programme est séparé des données. La mémoire des données comprend :

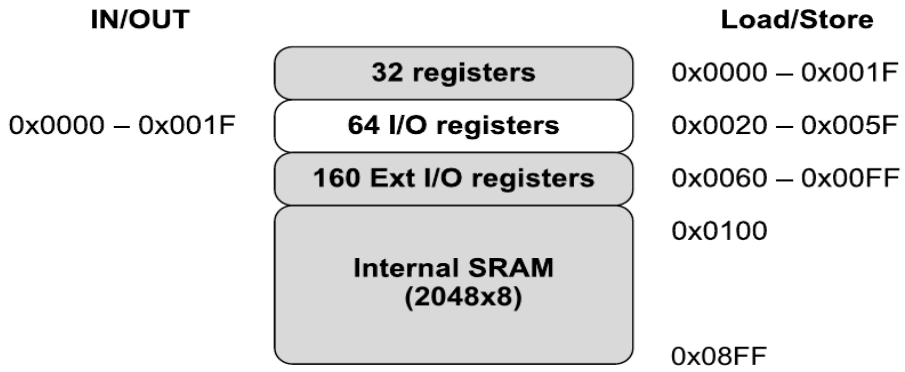
- 32 registres de l'adresse 0 à l'adresse 31. Ces registres sont des cases mémoires qui sont directement accessibles par le processeur.
- des registres E/S de l'adresse 32 à 95 Celui qu'il faut retenir est celui à l'adresse 95 qui est le registre d'état nommé SREG (Status REGister)
- un pointeur de pile (Stack Pointer) aux adresses 93 et 94 : quand on fait un appel à un sous-programme la première chose que fait cette instruction est d'empiler l'adresse de retour, c'est-à-dire écrire l'adresse de retour dans la mémoire de manière à récupérer à la fin d'exécution du sous-programme. La pile est descendante avec post-décrémentation : le push se fait par post-décrémentation et pop, par pré-incrémentation.

1.5 Le plan mémoire

Mémoire programme



Mémoire données



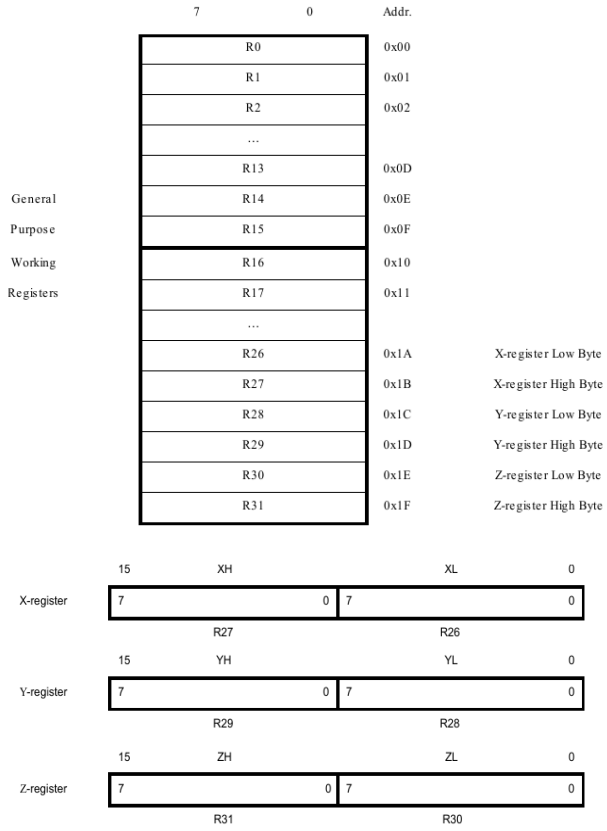
2 Cycle d'exécution d'une instruction AVR

2.1 Recherche de l'instruction

(en anglais : Instruction Fetch)

2.2 Fromat de l'instruction

3 Registres



3.1 Convention d'utilisation des registres Rx avec gcc

La rédaction des routines de langage d'assemblage à mélanger avec le code C nécessite la connaissance de la façon dont le compilateur utilise les registres.

R0 registre improvisé (scratch register) qui ne nécessite pas d'être restauré. Par contre, si on l'utilise dans une routine d'interruption, il doit être restauré.

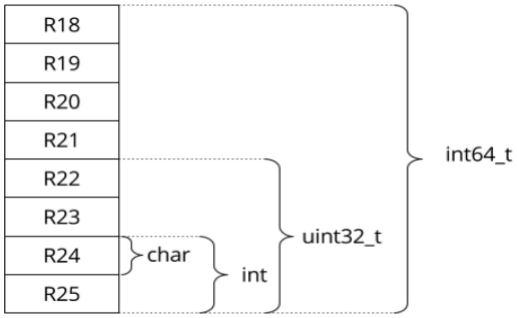
R1 toujours égal à 0. Si on le modifie, il faudra le restaurer à 0. Par exemple, l'instruction `MUL` écrit le résultat du produit dans le couple R0,R1. R1 sera donc modifié, il faudra alors le remettre à 0 à la fin du calcul.

R2 à R7 registres qui sont supposés ne pas changer de valeur. Si le programme en C appelle une routine et qu'il a modifié R2 = 5 par exemple, il suppose qu'après le retour de cette routine R2 sera encore à 5.

R8 à R25 registres utilisés pour :
— le passage des paramètres
— la valeur de retour

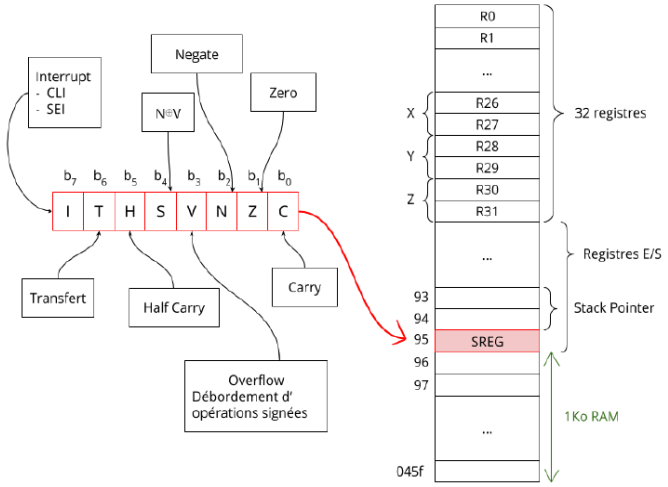
Règles à respecter :

1. les paramètres sont alignés sur des numéros pairs Par exemple, si on a la fonction suivante : `uint8_t fonc(char a, char b)`, si le premier paramètre est `R24 (R24:R25)`, le deuxième paramètre sera `R22 (R22:R23)` Si on a `fonc(uint32_t a, uint32_t b)`, `a` sera dans `R22 R23 R24 R25` et `b` dans `R18 R19 R20 R21`
2. le premier paramètre est le numéro plus élevé
3. on utilise la convention little endian



4 Registre d'état : Status Register (SREG)

Le registre SREG contient des indicateurs liés aux opérations et le bit d'autorisation générale des interruptions. Les bits de ce registre sont : Z (Zero), C (Carry), S (Sign) ...



Bit	7	6	5	4	3	2	1	0
	I	T	H	S	V	N	Z	C
Access	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Reset	0	0	0	0	0	0	0	0

Bit 7 – I – Global Interrupt Enable Ce bit est mis à la valeur logique haute (c'est-à-dire 1) pour activer l'utilisation des interruptions. Ce bit est mis à la valeur logique basse (c'est-à-dire 0) après une demande d'interruption. Il est remis à 1 par l'instruction RETI au retour d'une routine d'interruption.

Bit 6 – T – Bit Copy Storage Les instructions de copie des bits (BLD, bit lu et BST, bit emmagasiné) utilisent le bit T comme source et destination dans les opérations effectuées sur chaque bit d'un registre. Un bit d'un registre peut être copié dans le bit T par l'instruction BST alors que le bit T peut être copié dans un autre registre à travers l'instruction BLD.

Bit 5 – H – Half Carry Flag Ce bit indique qu'une opération arithmétique a généré une retenue ou un dépassement.

Bit 4 – S – Sign Bit Le bit S est donné par un OR exclusif entre le flag négatif N et celui complémenté à 2 du flag V. Il indique le signe de la donnée après avoir exécuté une opération arithmétique.

Bit 3 – V – Overflow flag Ce bit contient le résultat d'overflow et est en complément à deux. Traduit littéralement, overflow veut dire débordement. C'est une condition dans laquelle une opération arithmétique fournit un résultat de grandeur supérieure au maximum qu'un registre ou une position de mémoire peut contenir.

Bit 2 – N – Negative flag

Bit 1 – Z – Zero flag

Bit 0 – C – Carry flag Ces trois bits indiquent, respectivement, au terme d'une opération mathématique ou logique, si le résultat est négatif, si le résultat est égal à 0, si l'opération a donné lieu, en plus du résultat, à une retenue. Le registre d'état (Status Register) n'est pas automatiquement sauvegardé lorsque l'on appelle une routine d'interruption.

```
; Initialisation du registre d'état

init:
   ldi r16,0 ;
```

5 La pile

La pile est principalement utilisée pour stocker des données temporaires, pour stocker des variables locales et pour stocker des adresses de retour après des interruptions et des appels de sous-routine. La pile est mise en œuvre à mesure que les emplacements de mémoire de plus en plus élevés se situent à des niveaux inférieurs. Le Registre pointeur de pile (Stack Pointer) pointe toujours vers le haut de la pile.

AVR possède un pointeur de pile 16 bits. Le pointeur de pile pointe vers la zone de la pile SRAM de données où se trouvent les sous-programmes et les piles d'interruption. Une commande Empiler PUSH diminuera le pointeur de pile. La pile dans la SRAM de données doit être définie par le programme avant que tous les appels de sous-routine soient exécutés ou que les interruptions soient activées. Initial Stack La valeur du pointeur est égale à la dernière adresse de la SRAM interne et le Stack Pointer doit être placé au point au-dessus du début de la SRAM. Voir le tableau pour les détails du pointeur de pile.

```
; Initialisation du pointeur de pile
.equ RAMEND, 0x8ff
.equ SPL, 0x3d
.equ SPH, 0x3e

init:
   ldi r16,lo8(RAMEND)
   out SPL,r16
   ldi r16,hi8(RAMEND)
   out SPH,r16
```

6 Entrées Sorties

Port A (PA7...PA0) : C'est un port de I/O bidirectionnel. Toutes les pattes du port ont des résistances internes de pull-up. Le buffer de sortie est en mesure de fournir jusqu'à 20 mA de courant, suffisant pour piloter un afficheur à Led. Les pattes sont en haute impédances quand une condition de reset devient active, ou bien lorsque l'horloge n'est pas active. Ce port est utilisé comme multiplexer d'entrée/ sortie pour les données et les adresses quand une SRAM externe est reliée (broches 32 à 39).

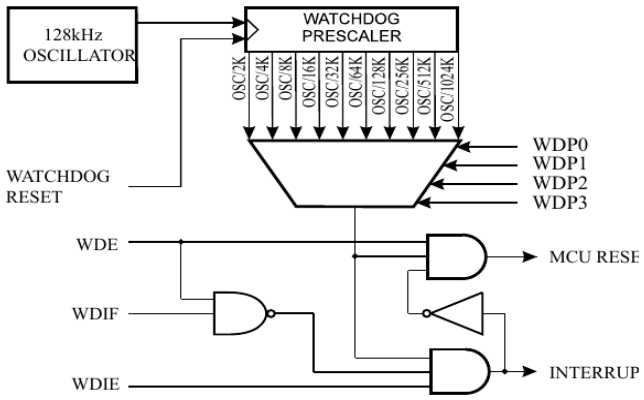
Port B (PB7...PB0) : C'est un port de I/O bidirectionnel. Toutes les pattes du port ont des résistances internes de pull-up. Le buffer de sortie est en mesure de fournir jusqu'à 20 mA de courant. Les pattes du port sont en haute impédance quand une condition de RESET devient active, ou bien quand l'horloge n'est pas active (broches 1 à 8).

Port C (PC7...PC0) : C'est un port de I/O bidirectionnel. Toutes les pattes du port ont des résistances internes de pull-up. Le buffer de sortie est en mesure de fournir jusqu'à 20 mA de courant. Les broches sont en haute impédance quand une condition de RESET devient active, ou bien quand l'horloge n'est pas active. Lorsqu'on branche de la SRAM externe, ce port est utilisé comme bus adresses en sortie vers la SRAM (broches 21 à 28).

Port D (PD7...PD0) : C'est un port de I/O bidirectionnel. Toutes les pattes du port ont des résistances internes de pull-up. Le buffer de sortie est en mesure de fournir jusqu'à 20 mA de courant. Les broches sont en haute impédance quand une condition de RESET devient active, ou bien quand l'horloge n'est pas active (broches 10 à 17).

7 watchdog

Le watchdog est un temporisateur particulier qui est utilisé dans les systèmes à microprocesseur comme sécurité pour éviter que le programme n'aille dans une impasse et donc que le système ne se bloque dans une situation non prévue par le programmeur. En pratique, le watchdog intervient et effectue le reset du microcontrôleur si celui-ci n'est pas effectué par l'instruction "WDR" (WatchDog Reset) dans le temps établi par les broches 0, 1 et 2 du registre "WDTCR". Le watchdog des microcontrôleurs AVR est temporisé par une horloge interne à 1 MHz, ce qui nous permet de comprendre qu'il peut fonctionner également en l'absence de l'horloge du système car il est indépendant de celui-ci. Le dispositif est programmé à travers le registre "WDTCR" grâce à l'utilisation des cinq premiers bits. Essayons maintenant d'en comprendre le fonctionnement en les analysant de façon détaillée.



Les bits 0, 1, et 2, comme nous l'avons déjà remarqué, servent à établir le temps qui doit s'écouler avant que le watchdog effectue le reset du micro. Ce temps dépend également de l'alimentation du micro et peut varier d'environ 15 ms (WDP0 = 0, WDP1 = 0, WDP2 = 0 et alimentation Vcc = 5 V) jusqu'à environ 6 s (WDP0 = 1, WDP1 = 1, WDP2 = 1 et alimentation Vcc = 3 V). Le bit 4 (WDTOE = Watch Dog Turn Off Enable) et le bit 5 (WDE) servent à désactiver la fonction de watchdog. Etant donné qu'il s'agit d'un système de sécurité, il serait trop risqué d'utiliser un seul bit de validation/inhibition vu que l'on ne peut pas savoir comment se comporte un programme en cas de dysfonctionnement. Donc, pour éviter des inhibitions involontaires, il est nécessaire de suivre une séquence précise de désactivation du watchdog. Il faut d'abord mettre au 1 logique aussi bien "WDTOE" que "WDE" et, ensuite, pour les quatre cycles d'horloge suivants, effectuer le reset de "WDE". De cette façon le watchdog est désactivé. Le schéma du Watchdog (figure 10) met en évidence l'oscillateur indépendant de 1 MHz, un "prescaler" et un multiplexeur. Les trois bits de contrôle vont justement agir sur le multiplexeur pour sélectionner les temporisations pour le reset.

8 L'assembleur AVR

8.1 L'assembleur

Les fichiers assembleur portent l'extension ".S", ces fichiers sont de type texte ils contiendrons le code source. Votre code assembleur peut être écrit sur un ou plusieurs fichiers. Vous pouvez utiliser un simple éditeur de texte pour écrire votre code par exemple emacs, gedit, kate etc, ou-bien utiliser un IDE (en anglais : Integrated development environment), ce dernier vous propose des fonctions avancées par exemple que assembler l'ensemble des fichier dans un projet, une coloration syntaxique, compilation, téléversement etc.

L'étape suivante, une fois le code écrit, est de rassembler ces fichiers et les traduire dans un langage compréhensible par le microcontrôleur. Cette étape se fait avec l'outil **assembleur**, qu'on appel communément le compilateur. Le compilateur d'une manière générale il vérifie les fichiers en entré, si tout est correct, alors il génère un fichier exécutable. Cependant si une erreur est détectée lors du processus de compilation alors le processus s'arrête et une ou plusieurs erreurs sont affichées avec les détails tel que le fichier dans lequel l'erreur s'est produite, le numéro de ligne ainsi que les causes possibles ;

Le fichier exécutable généré à l'étape précédente ne peut pas être chargé directement sur le microcontrôleur, pour cela nous rajoutons une étape supplémentaire qui permet de traduire cet exécutable au format binaire chargeable sur le microcontrôleur.

8.2 Structure d'un programme assembleur

Un fichier assembleur comporte deux sections, la première section appelé section de données elle commence par la ligne suivante **".section .data"**. A partir de cette ligne (jusqu'à la section suivante) l'assembler considère tout ce qui suit comme données. La seconde partie des fichiers assembleur est la partie texte elle commence par **".section .text"**, cette partie contient le code. Les fichiers se termine par la ligne **".end"**. Les trois lignes que nous venons de décrire s'appellent **des directives**, elles sont destinées au compilateur.

```
.section .data    ;; début section données
 ...

.section .text    ;; début section texte
 ...

.end              ;; fin
```

8.2.1 Section données

Cette section est utilisée pour définir les structure de données qui seront manipulés dans la partie code, ces données sont sauvegardées en mémoire.

```
mon_texte: .asciz  "Mon textes"
```

Cette ligne contient trois éléments, le premier élément **mon_texte :** est appelé **Label**, c'est un identifiant qui permet de référencer l'adresse en mémoire où la donnée "Mon texte" est stockée en mémoire . Le second élément est une directive, elle est destinée au compilateur afin de lui expliquer que ce qui suit est une suite de caractères ASCII suivit d'un '0x00', c'est-à-dire un String, autrement la deuxième partie permet de définir le type de la donnée.

8.2.2 Section texte

Cette section commence généralement par un Label qui permet d'indiquer le début du programme, par convention le label est noté main, cette dernière n'est qu'une convention de notation, afin d'indiquer au compilateur que ce Label est accessible de l'extérieur, on utilise la directive **.global :** cette directive doit précéder la définition du label en question.

```
...
.section .text
  .global main
  main :
        push 24
        push 27
          ...
          ...
        pop 27
        pop 24
        RET
.end
```

Généralement on commande par des instructions **push**, elles permettent de sauvegarder certains registres dans la pile, car la convention nous oblige de sauvegarder certains registre avant de les utiliser et les restaurer avec l'instruction **pop** avant la fin de la fonction, nous allons détaillier le fonctionnement de pile dans un chapitre dédié.

9 Chaine de compilation

9.1 Compilation

Un programme assembleur porte l'extension **".s"** Pour assembler un programme on utilise la commande **avr-as** l'option **-g**, que nous avons ajouté, permet d'insérer dans le code généré les symboles de debug, cette option est seulement nécessaire dans le cas où vous allez utiliser le dans la suite le débogueur **GDB (avr-gdb)**. L'option **-mmcu** permet de choisir le type du microcontrôleur pour lequel nous allons générer le code objet.

```
avr-as -g -mmcu=atmega328p -o hello.o hello.s
```

Après l'exécution de cette commande un fichier objet est généré dans le nom est spécifié par l'option **-o** . Il contient le code binaire mais il n'indique aucun emplacement mémoire spécifique. Cela se fait à l'étape suivante par l'éditeur de liens qui lie tous les objets sources et localise le code à une adresse physique spécifique.

```
avr-ld -o hello.elf hello.o
```

La commande **avr-ld** est appelé l'éditeur de lien, elle permet d'assembler un ou plusieurs fichier objet dans un seul fichier binaire. Le résultat de cette commande est un fichier **elf** (Executable and Linkable Format, format exécutable et liable), ce format de binaire n'est généralement pas supporté par les microcontrôleurs. Avant de pouvoir le téléverser sur le microcontrôleur, il doit être converti dans le format Intel HEX (extention des fichies .hex). Il s'agit d'un format de fichier ASCII pour les données binaires. Il est généralement largement utilisé pour la programmation de microcontrôleurs. La conversion de format se fait avec l'outil **objcopy** :

```
avr-objcopy -O ihex -R .eeprom hello.elf hello.hex
```

l'option de **-O** permet de choisir le type du fichier en sortie .

9.2 Téléversement du programme

Maintenant que nous avons préparé notre binaire nous allons nous servir de l'outil **avrdude** qui va nous permet de téléverser le programme généré sur le microcontrôleur.

```
avrdude -C /etc/avrdude.conf -p atmega328p -c arduino -P \
/dev/ttyACM0 -b 115200 -D -U flash:w:hello.hex:i
```

Option **-C** permet de spécifier le fichier de configuration. Un fichier de configuration standard est normalement déjà livré avec le paquet avrdude. Option **-p** permet de spécifier le type du microcontrôleur, l'option **-P** permet de sélectionner le nœud du périphérique USB, l'option **-b** permet de fixer débit en bauds de la transmission série, l'option **-D** permet désactive l'effacement complet de la puce et l'option **-U** spécifie le fichier à téléverser.

9.3 Makefile

Afin de vous facilité la génération de votre binaire et le réléversement vous pouvez vous servir du makefile suivant :

```
TARGET = $(notdir $(CURDIR))
SOURCES = $(wildcard *.S)
OBJECTS = $(SOURCES:.S=.o)
MCU = atmega328p
#MCU = atmega16
## settings for Uno
USBDEV = /dev/ttyACM0
BAUD = 115200
# settings for Duemilanove
#USBDEV = /dev/ttyUSB0
#BAUD = 57600
AS = avr-as
CC = avr-gcc
LD = avr-ld
OBJDUMP = avr-objdump
CPP = avr-cpp
CFLAGS = -g -mmcu=$(MCU)
ASFLAGS = -g -mmcu=$(MCU)
LDFLAGS = -Tdata=0x800100 -nostdlib
#LDFLAGS = -Tbss=0x800100 -Tdata=0x800300
AVRDUDE = avrdude
AVRDUDE_CONF = /etc/avrdude.conf

all: $(TARGET).hex

$(TARGET).elf: $(OBJECTS)
  $(CC) $(LDFLAGS) $(OBJECTS) -o $@

$(TARGET).hex: $(TARGET).elf
  avr-objcopy -O ihex -R .eeprom $(TARGET).elf $(TARGET).hex

upload: $(TARGET).hex
$(AVRDUDE) -C $(AVRDUDE_CONF) -p $(MCU) -c arduino -P $(USBDEV) -b $(BAUD) \
       -D -U flash:w:$(TARGET).hex:i

dump: $(TARGET).elf
  $(OBJDUMP) -d $(TARGET).elf

clean:
  rm -f $(TARGET).elf $(TARGET).hex $(OBJECTS)

.PHONY: clean upload dump
```

9.4 Utilisatation des C et Assembleur

L'utilisation de l'assembleur permet d'optimiser vos codes et vous produira un programme rapide, ou lorsque vous voulez utiliser des instructions spécifiques à l'architecture et non descriptibles par une syntaxe de haut niveau par exemple en C. Ou tout simplement effectuer des calcules plus efficacement par exemple additionner deux nombres de 128 bits, dans ce cas la tâche s'avère difficile en C mais très facile à coder en assembleur.

L'écriture d'un programme complet en assembleur est une tâche fastidieuse, difficile et potentiellement source de bugs, pour cela nous allons écrire des programmes C auxquels nous allons ajouter certaines fonctions en assembleur

9.5 Organisation du projet

Il n'y a pas d'exigences particulières pour l'organisation du projet. Les fichiers identifiés comme des fichiers d'en-tête, que ce soit le code C ou l'assembleur, seront placés dans le dossier **Headers** du projet. Les fichiers identifiés comme fichiers sources seront placés dans le dossier **Sources** du projet, il est tout a fait possible que certains IDE mélange les deux types de fichiers dans un même dossier, cela dépend de vote IDE.

Bien que cela ne soit pas strictement requis (vous pouvez utiliser un éditeur de texte et un compilateur en ligne de commande), tous les fichiers sources assembleur doivent avoir une extension ".S". Cela permettra à l'interface du compilateur C d'appeler automatiquement l'assembleur et l'éditeur de liens au besoin. En outre, le préprocesseur C sera automatiquement invoqué en permettant à l'utilisation de constantes symboliques.

Un des fichiers source devra produire un fichier objet "principal" pour l'éditeur de liens afin que l'éditeur de liens connaisse le point d'entrée du programme. Le plus commun est un fichier C avec une fonction appelée **"main()"**. Un fichier Assembleur avec un sous-programme nommé "main" et déclaré global (en utilisant la directive ".global") produira également un module nommé "main". La combinaison de C et d'assemblage dans un seul projet soulève plusieurs questions en fonction des besoins du programme qu'on souhaite écrire :

— Comment une routine d'assemblage peut-elle être rendue visible au programme C de telle sorte qu'une fonction C peut appeler la routine d'assemblage ?

— Comment une fonction C peut-elle être rendue visible à une routine assembleur de sorte que la routine assembleur puisse appeler la fonction C ?

— Comment les variables sont-elles passées au code assembleur ?

— Comment les variables passent-elles à la fonction C ?

— Le code assembleur et C peuvent-ils utiliser les mêmes variables globales ?

Dans la suite de cette section nous allons essayer de répondre a ces questions et donner des exemples pratique pour chaque cas.

9.5.1 Visibilité des fonctions

Une fonction de langage C doit être déclarée comme externe dans le code assembleur afin d'être **vue** par l'assembleur :

```
/* Code C */
.extern ma_function_C
```

Une routine du langage assembleur doit être déclarée comme globale dans le code assembleur afin qu'elle soit visible par le programme C. Cela se fait en utilisant la directive ".global" :

```
;; Code assembleur
.global ma_function_assembleur
```

Un fichier C qui a l'intention d'appeler la routine de langage assembleur devra disposer d'un prototype de fonction déclarant la routine de langage assembleur externe :

```
/* Code C */
extern unsigned char ma_function_assembleur (unsigned char, unsigned int);
```

9.5.2 Les variables

Le code C et le code assembleur peuvent accéder aux variables indépendamment. En pratique, il est conseillé de laisser le code C gérer les variables et passer des paramètres au code assembleur soit par valeur, soit par référence. La section 3 décrit comment les ensembles de registres sont utilisés par le compilateur C et comment les paramètres sont transmis.

10 Exemple code en C

http ://www.atmel.com/webdoc/avrassembler/avrassembler.wb$_{d}irectives.html$

Programme C

```
 #include <avr/io.h>
#include <util/delay.h>

int main (void)
{
  /* set all pins of PORTB for output*/
  DDRB = 0xFF;

  while (1) {
    /* set all pins of PORTB high */
    PORTB = 0xFF;
    _delay_ms(500);

    /* set all pins of PORTB low */
    PORTB = 0x00;
    _delay_ms(500);
  }
  return 0;
}
```

Compilation

```
avr-gcc -Os -DF_CPU=16000000UL -mmcu=atmega328p -c -o exemple_1.o exemple_1.c
```

Edition des liens

```
avr-ld exemple_1.o -o exemple_1.elf
```

Fichier Hexadécimal

```
avr-objcopy -O ihex -R .eeprom exemple_1.elf exemple_1.hex
```

Téléversement

```
avrdude -C /etc/avrdude.conf -p atmega328p -c arduino -P /dev/ttyACM0   \
    -b 115200 -D -U flash:w:exemple_1.hex:i
```