

Fonctionnement des Ordinateurs

TP2 - Développement embarqué sur plateforme MIPS

B. Quoitin
Faculté des Sciences
Université de Mons

Résumé

Les objectifs de ce TP sont le renforcement de votre compréhension du jeu d'instructions MIPS et de la programmation en langage d'assemblage, la découverte du langage C, et une introduction aux mécanismes d'entrées/sorties, timers et interruptions. La différence majeure avec le TP précédent est que les programmes qui seront développés seront exécutés sur une plateforme réelle au lieu d'un simulateur.

Table des matières

1	Découverte de la plateforme	2
1.1	Caractéristiques du processeur	2
1.2	Carte d'extension	2
1.3	Précautions d'emploi	2
1.4	Documentation	3
2	Hello World	4
2.1	Connexion processeur-LED	4
2.2	Contrôler l'état d'un port	4
2.3	Application : contrôle de la broche RG6	4
2.4	Programmer le microcontrôleur	4
2.5	Attendre un temps défini	7
2.6	Auto-évaluation	7
A	Annexes	8
A.1	Organisation interne du PIC32	8
A.2	Connexion processeur-LED	8
A.3	Structure d'une broche d'un port GPIO.	8
A.4	Code objet : ELF	9
A.5	Linker	10

Introduction

Les objectifs de ce TP sont le renforcement de votre compréhension du jeu d'instructions MIPS et de la programmation en langage d'assemblage, la découverte du langage C, et une introduction aux mécanismes d'entrées/sorties, timers et interruptions. La différence majeure avec le TP précédent est que les programmes qui seront développés seront exécutés sur une plateforme réelle au lieu d'un simulateur. Travailler avec une plateforme réelle amène des difficultés supplémentaires telles que la lecture de documentation technique et la compréhension de quelques notions d'électronique numérique. Elle a l'avantage de permettre à un programme d'interagir avec le monde réel. Afin d'atteindre ces objectifs, le TP vise à la construction d'une application complète : une calculatrice.

Le TP est découpé en étapes qui vont permettre de construire progressivement les blocs de base de l'application. Chaque étape devrait être réalisable en une séance de 2 heures de TP. La première étape vise l'interaction entre le processeur et des éléments externes tels que LEDs, boutons et interrupteurs. En particulier, le premier programme à réaliser devra faire clignoter une LED. Il s'agit de l'équivalent embarqué du programme "Hello World". Les étapes suivantes introduiront des éléments nouveaux tels que des timers et des interruptions, deux mécanismes nécessaires, notamment au clignotement, mais également à la gestion des boutons (anti-rebonds).

Le TP repose sur deux langages de programmation. Pour la première étape, la programmation en langage d'assemblage MIPS sera utilisée afin d'en renforcer votre compréhension et de faire le lien avec le TP précédent sur le simulateur SPIM. Cependant, la programmation en langage d'assemblage s'avèrera rapidement une limitation étant donnée la complexité de l'application finale visée. Le langage C sera utilisé dès la seconde étape. La syntaxe du langage C est relativement proche du langage Java. Il est conseillé de lire l'introduction au langage C fournie en annexe de cet énoncé.

Remerciements

Merci à ALAIN BUYS, DAVID HAUWEELE, JÉRÔME DAUGE et M. CHARLIER pour leurs relectures attentives et/ou leurs commentaires sur des versions précédentes de ce document.

1 Découverte de la plateforme

La plateforme matérielle utilisée dans ce TP est la carte électronique *ChipKit Uno32* conçue par la société Digilent Inc. Une photographie de cette carte est présentée à la Figure 1. La carte comporte un microcontrôleur *PIC32MX320F128H* fabriqué par la société Microchip Inc. Il s'agit du circuit intégré situé au centre de la carte. La carte comporte un port USB qui fournit l'alimentation électrique et qui permet de charger un programme dans la mémoire du microcontrôleur à partir d'un ordinateur.

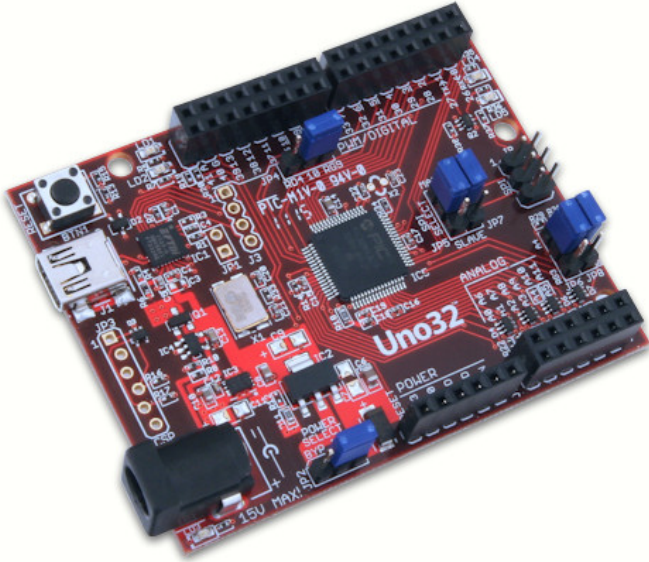


FIGURE 1 – Carte ChipKit Uno32 (source : Digilent Inc).

La carte ChipKit comporte plusieurs LEDs, chacune portant un nom de la forme *LDx*. La première, notée *LD1*, est allumée lorsque la carte est sous tension. Les LEDs *LD2* et *LD3* sont actives lorsque l'ordinateur communique avec le microcontrôleur. Les LEDs *LD4* et *LD5* sont connectées au microcontrôleur et peuvent être commandées par programme. La carte comporte un bouton nommé *RESET* qui permet de ré-initialiser le microcontrôleur (le faire redémarrer à une certaine adresse).

1.1 Caractéristiques du processeur

Le microcontrôleur *PIC32MX320F128H* contient un cœur MIPS32 (M4K) responsable de l'exécution des instructions. L'architecture de ce processeur est identique à celle vue au chapitre 4 du cours. Le microcontrôleur peut fonctionner à une cadence allant jusqu'à 80MHz. Cette cadence est contrôlée par un oscillateur à quartz. Le quartz est le petit rectangle métallique situé à côté du microcontrôleur sur la carte ChipKit sur lequel il est noté 8.000.

Le microcontrôleur est en fait un système complet dans un circuit intégré (*System on Chip*). Il comporte par exemple de la mémoire SRAM (16KB) et de la mémoire Flash (128KB). Les bus qui connectent ces mémoires au cœur MIPS sont internes au microcontrôleur. La mémoire Flash est destinée à contenir le programme à exécuter¹. Le microcontrôleur ne dispose pas de mémoire cache.

1. Contrairement à un ordinateur où les programmes sont typiquement stockés sur un disque dur ou SSD, les programmes du microcontrôleur sont stockés dans une mémoire Flash située à l'intérieur du microcontrôleur

De plus, le microcontrôleur contient un grand nombre de périphériques simples qui sont également connectés au cœur MIPS via des bus internes. Ces périphériques ont des fonctions variées telles que des ports d'entrées/sorties, des *timers* (au nombre de 5), des convertisseurs analogique/numérique (ADC), ainsi que des interfaces avec des bus de communication (UART, SPI et I²C). Les ports d'entrées/sorties permettent au processeur d'interagir avec le monde extérieur. C'est à travers ces ports qu'il sera possible de commander l'état des LEDs ou de déterminer l'état des boutons de la carte ChipKit.

1.2 Carte d'extension

La carte d'extension *Basic I/O Shield* peut être connectée à la carte ChipKit afin d'y ajouter de nouvelles entrées/sorties. La carte ChipKit Uno32 ne comporte en effet que deux LEDs contrôlables par le microcontrôleur. La Figure 2 montre une photographie de la carte Basic I/O Shield.

La carte Basic I/O Shield comporte les éléments suivants : 4 boutons poussoirs *BTN1-4*, 4 interrupteurs à glissière *SW1-4*, 8 LEDs *LD1-8*², un potentiomètre *VR1* (résistance variable), un écran graphique OLED de 128x32 pixels, un thermomètre (circuit *TMPN75A*), 4 entrées/sorties numériques et 4 sorties numériques de puissance.

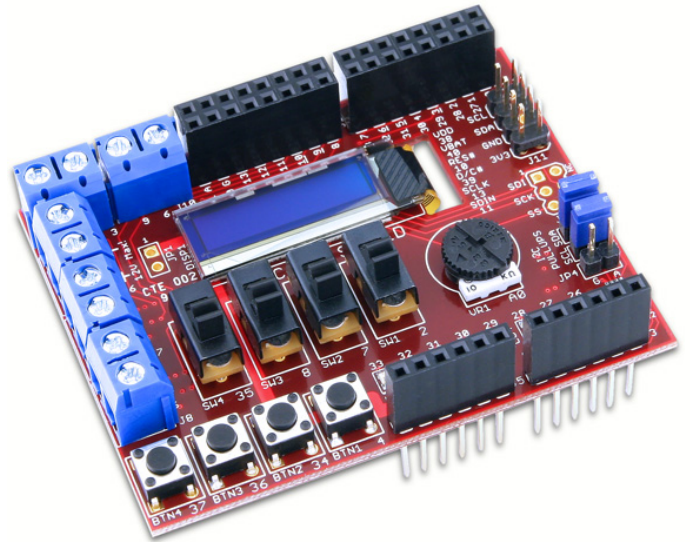


FIGURE 2 – Carte Basic I/O Shield (source : Digilent Inc).

1.3 Précautions d'emploi

ATTENTION Les cartes électroniques ChipKit Uno32 et Basic I/O Shield avec lesquelles vous allez travailler sont des composants fragiles ! Il est nécessaire de les manipuler avec précautions. Veuillez respecter scrupuleusement les recommandations suivantes.

— Sensibilité à l'électricité statique.

- Ne pas mettre ses doigts sur les composants et connecteurs.

2. Des LEDs de la carte ChipKit et de la carte d'I/O portent le même nom. En cas d'ambiguïté, la carte sur laquelle la LED se trouve sera désignée explicitement.

- Manipuler les cartes en les tenant par la tranche.
- **Sensibilité aux contraintes mécaniques.**
 - Ne pas exercer de pression (en particulier sur l'écran OLED).
 - Ne pas exercer de torsion.
- **Configuration.**
 - Ne pas changer la position des cavaliers (*jumpers*).
 - Ne pas brancher d'alimentation électrique. Seul le port USB sera utilisé pour alimenter la carte.
 - Il n'est pas interdit de connecter d'autres périphériques / composants sur la carte de développement : veuillez cependant consulter les assistants auparavant !!
 - Débrancher le port USB avant de (dé-)connecter les cartes ChipKit Uno32 et Basic I/O Shield.

1.4 Documentation

Afin d'utiliser la carte de développement et le processeur, il sera parfois nécessaire de se référer à leur documentation. Les documents importants sont les suivants

- **Datasheet du PIC32MX320F128H.** Il s'agit de la documentation technique du microcontrôleur. Celle-ci décrit tout ce qu'il est nécessaire de savoir à propos du microcontrôleur : les détails du cycle d'instruction, l'organisation de la mémoire, l'emplacement et la signification des registres spéciaux, les caractéristiques électriques, etc. Cette documentation peut être téléchargée librement à partir du site web de Microchip Inc., à l'adresse suivante et dans la section *Documentation & Software / Reference Manual*.
<http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en532434>
- **Datasheet ChipKit Uno32 et Basic I/O Shield.** Il s'agit de la documentation technique de la carte de développement. Les schémas électriques permettent notamment de déterminer quelles broches du microcontrôleur sont reliées à quel composant externe (p.ex. LED, bouton on interrupteur). La documentation de la carte ChipKit Uno32 peut être téléchargée librement sur le site de la société Digilent Inc., à l'adresse suivante :
http://www.digilentinc.com/Data/Products/CHIPKIT-UNO32/chipKIT-Uno32-RevC_rm.pdf
 Le schéma électrique de la carte ChipKit Uno32 peut être téléchargé à l'adresse suivante :
http://www.digilentinc.com/Data/Products/CHIPKIT-UNO32/ChipKIT\%20Uno32_bysa_c_sch.pdf

Cette documentation ne se lit pas de bout en bout comme un roman : elle totalise plusieurs centaines de pages et est assez indigeste. De plus, seule une petite partie de cette documentation sera utile pour les exercices proposés dans ce TP. A plusieurs reprises, il vous sera demandé de vous référer à la documentation. Cependant, vous serez guidés et les sections où les informations peuvent être trouvées seront indiquées.

2 Hello World

L'objectif de cette première étape est de comprendre comment un programme peut contrôler des éléments externes au processeur tels que des LEDs et des interrupteurs. Le premier programme à réaliser devra faire clignoter la LED LD4 située sur la carte ChipKit Uno32. Le programme sera écrit en langage d'assemblage.


Un résumé des étapes à suivre afin de réaliser ce programme est présenté ci-dessous. Le détail des étapes sera expliqué dans les sections qui suivent.

1. **Connexion de la LED LD4 au processeur (Section 2.1).** Identifier à quel port d'entrée/sortie du microcontrôleur la LED LD4 est connectée.
2. **Contrôler l'état du port (Section 2.2).** Comprendre comment il est possible par programme de contrôler l'état 0 / 1 d'un port d'entrée/sortie et par conséquent l'état allumé / éteint de la LED.
3. **Programmer le microcontrôleur (Section 2.4).** Ecrire un programme en langage d'assemblage MIPS qui contrôle la LED, le convertir en binaire et le télécharger dans la mémoire Flash du microcontrôleur au travers d'une connexion USB.

2.1 Connexion processeur-LED

Afin de déterminer à quel port d'entrée/sortie du microcontrôleur la LED LD4 est connectée, il est nécessaire de consulter le schéma électrique de la carte ChipKit Uno32. L'extrait du schéma électrique qui concerne les LED LD4 et LD5 est repris à la Figure 12 en annexe. Le schéma indique que

- LD4 est connectée à la broche RG6
- LD5 est connectée à la broche RF0

 **Note :** il n'est pas possible de changer à quel port une LED est connectée. Ce sont les ingénieurs qui ont conçu la carte ChipKit qui ont pris cette décision.

2.2 Contrôler l'état d'un port

Le microcontrôleur PIC32MX320F128H possède plusieurs ports d'entrées/sorties appelés GPIO (*General Purpose Input Output*). Ces ports peuvent servir pour effectuer des entrées/sorties numériques mais sont souvent multiplexés avec d'autres fonctions telles que des convertisseurs analogique/numérique. Les ports sont au nombre de 7 et sont nommés PORTA à PORTG. Un port peut contrôler jusqu'à 32 broches du microcontrôleur. Par exemple, la broche RG6 à laquelle la LED LD4 est connectée est contrôlée par le PORTG. Il s'agit de la broche 6 de ce port.

Les périphériques du microcontrôleur tels que les ports GPIO sont contrôlés au travers de registres appelés *Special Function Registers* (SFRs). Au contraire des registres GPRs qui font partie du cœur MIPS et dont il a été question durant le cours, les SFRs ne servent pas à stocker des résultats temporaires, mais uniquement à configurer et interagir avec les périphériques. De plus, les SFRs se situent à l'extérieur du cœur MIPS. Ils sont accédés via les bus qui séparent le cœur MIPS des périphériques (ceci peut être observé sur la Figure 13 en annexe).

Les SFRs sont accédés de la même façon que des cellules mémoires, avec des instructions *load* et *store* : ce sont des registres *memory mapped*. Une partie de l'espace d'adressage est réservé à ces registres. Dans le cas du PIC32, il s'agit de l'espace compris

entre les adresse 0xBF800000 et 0xBF8FFFFFF. Chaque registre a une taille de 32 bits et est associé à une adresse mémoire.

Chaque port est contrôlé par au moins 4 registres de 32 bits. Soit le port x , les registres associés à ce port sont les suivants :

- $TRISx$: le bit n contrôle la direction (entrée ou sortie) de la broche n du port x . La valeur 0 correspond à une sortie³ et la valeur 1 à une entrée. La valeur par défaut est 1 (entrée).
- $LATx$: le bit n correspond à la valeur à présenter en sortie sur la broche n du port x .
- $PORTx$: la lecture du bit n permet d'obtenir la valeur présente sur la broche n du port x . En écriture, le comportement est identique à $LATx$.
- $ODCx$: le bit n configure la broche n du port x en "collecteur ouvert" (*open drain*). Ceci n'a de sens que si cette broche est configurée en sortie. La valeur 0 désactive l'*open drain*. La valeur par défaut est 0 (désactivé). — Nous n'utilisons pas ce registre dans les TPs.

Exemple. Prenons l'exemple de la broche RG6 associée à la LED LD4. Cette broche fait partie du PORTG. La Figure 3 montre un extrait de la documentation du PIC32 relative aux 4 registres associés au PORTG. On constate que tous les bits des registres ne sont pas implémentés (certains sont "grisés"). Seuls les broches 2, 3, 6-9 sont disponibles sur le PIC32MX320F128H. Pour chaque registre, l'adresse mémoire est indiquée. Par exemple, $TRISG$ est accessible à l'adresse 0xBF886180.

2.3 Application : contrôle de la broche RG6

Afin d'allumer ou d'éteindre la LED, il faut effectuer les actions suivantes :

1. Configurer la broche comme une sortie en mettant le bit 6 du registre $TRISG$ à 0. Afin de mettre ce bit 6 à 0, un ET bit à bit est effectué entre la valeur de $TRISG$ et la constante 0xFFFFFBBF. **Il est important de comprendre pourquoi cette valeur est utilisée :** tous les bits valent 1 sauf le bit 6 qui vaut 0.
2. Ecrire l'état de la broche (et donc de la LED) dans le bit 6 du registre $LATG$. Un bit à 0 correspond à l'extinction de LD4 tandis qu'un bit à 1 correspond à l'allumage de LD4. Afin de mettre le bit 6 à 1, un OU bit à bit est effectué entre la valeur de $LATG$ et la constante 0x00000040 (tous les bits valent 0 sauf le bit 6 qui vaut 1).

Le bout de code en langage d'assemblage correspondant à la configuration de RG6 en sortie est montré à la Figure 4. La partie correspondant à l'allumage de LD4 est montrée à la Figure 5. Il est important de noter que dans les deux exemples, les constantes $TRISG$ et $LATG$ désignent les adresses mémoires des registres correspondants. Pour lire ou écrire la valeur de ces registres, des instructions *lw* et *sw* sont nécessaires.

2.4 Programmer le microcontrôleur

La Figure 6 illustre un programme complet écrit en langage d'assemblage MIPS et destiné à faire changer l'état de la LED LD4. Comme dans les programmes écrits pour le simulateur SPIM lors du TP précédent, le point d'entrée du programme est désigné par le

3. Le nom $TRIS$ vient de *Tri-State* qui correspond à un état "haute-impédance" dans lequel la sortie du port est déconnectée de la broche, ce qui permet de lire le port (le considérer en entrée).


```

1 TRISG=0xBF886180
2
3      li      $a2, TRISG
4      lw      $a0, 0($a2)
5      li      $a1, 0xFFFFFBBF
6      and     $a0, $a0, $a1
7      sw      $a0, 0($a2)

```

FIGURE 4 – Configuration de RG6 en sortie.

```

1 LATG=0xBF8861A0
2
3      li      $a2, LATG
4      lw      $a0, 0($a2)
5      li      $a1, 0x00000040
6      or      $a0, $a0, $a1
7      sw      $a0, 0($a2)

```

FIGURE 5 – Niveau logique haut (1) écrit dans RG6.

label `main`. Ce label doit être publié comme symbole global avec la directive `.global` ou `.globl`. Les directives `.ent` (*entry*) et `.end` marquent respectivement le début et la fin d’une fonction et sont destinées au débogage éventuel.

Assemblage. Contrairement au simulateur SPIM, il est nécessaire d’assembler le code écrit en langage d’assemblage. L’assembleur se charge de traduire le code en langage d’assemblage en une suite d’instructions MIPS. L’assembleur se charge également de positionner les instructions et données à des adresses dans des segments séparés. Le résultat est appelé code objet⁴. L’assemblage du programme `mainla.s` de la Figure 6 avec l’assembleur `xc32-as` est montré à la Figure 7. L’option `-O0` demande à l’assembleur de n’effectuer aucune optimisation. L’option `-o` spécifie le nom du fichier produit.

Le code objet résultant est placé dans un fichier appelé `mainla.o`. Ce fichier est organisé selon le format ELF. L’annexe A.4 décrit ce format et les outils utiles pour en lire le contenu.

4. Attention, “code objet” n’a rien à voir avec de la programmation orienté-objet.

```

1 TRISG=0xBF886180
2 LATG=0xBF8861A0
3
4      .text
5      .globl main
6      .ent main
7  main:
8          # configuration RG6 en sortie
9          li      $a2, TRISG
10         lw      $a0, 0($a2)
11         li      $a1, 0xFFFFFBBF
12         and     $a0, $a0, $a1
13         sw      $a0, 0($a2)
14         # chargement valeur LATG dans $a2
15         li      $a2, LATG
16         lw      $a0, 0($a2)
17  main_loop:
18         # extinction LD4 (LATG6=0)
19         li      $a1, 0xFFFFFBBF
20         and     $a0, $a0, $a1
21         sw      $a0, 0($a2)
22         # allumage LD4 (LATG6=1)
23         li      $a1, 0x00000040
24         or      $a0, $a0, $a1
25         sw      $a0, 0($a2)
26         b       main_loop
27     .end main

```

FIGURE 6 – Programme complet changeant en permanence l’état de RG6 (`mainla.s`).

```

bqu@:~$ xc32-as -O0 -o mainla.o mainla.s
bqu@:~$

```

FIGURE 7 – Assemblage du programme `mainla.s`.

Édition des liens. Le code objet généré par l’assembleur n’est pas directement utilisable pour programmer le microcontrôleur. Il est nécessaire de le ré-arranger et de le combiner avec d’autres codes objet. C’est le rôle d’un programme appelé éditeur de liens (*linker*). Les tâches remplies par le *linker* sont décrites plus en détails à l’annexe A.5.

L’utilisation du *linker* pour produire un programme complet est

Virtual Address (BF88_#)	Register Name	Bit Range	Bits																All Resets
			31/15	30/14	29/13	28/12	27/11	26/10	25/9	24/8	23/7	22/6	21/5	20/4	19/3	18/2	17/1	16/0	
6180	TRISG	31:16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
		15:0	—	—	—	—	—	—	TRISG9	TRISG8	TRISG7	TRISG6	—	—	TRISG3	TRISG2	—	—	03cc
6190	PORTG	31:16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
		15:0	—	—	—	—	—	—	RG9	RG8	RG7	RG6	—	—	RG3	RG2	—	—	xxxx
61A0	LATG	31:16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
		15:0	—	—	—	—	—	—	LATG9	LATG8	LATG7	LATG6	—	—	LATG3	LATG2	—	—	xxxx
61B0	ODCG	31:16	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
		15:0	—	—	—	—	—	—	ODCG9	ODCG8	ODCG7	ODCG6	—	—	ODCG3	ODCG2	—	—	0000

Legend: x = unknown value on Reset, — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

Note 1: All registers in this table have corresponding CLR, SET and INV registers at their virtual addresses, plus offsets of 0x4, 0x8 and 0xC, respectively. See [Section 12.1.1 “CLR, SET and INV Registers”](#) for more information.

FIGURE 3 – Documentation des registres associés au PORTG (Source : Microchip Inc).

montrée à la Figure 9. L'option `-T ldscript.ld` mentionne le *linker script* à utiliser⁵. L'option `-o` spécifie le nom du fichier binaire produit. Ce dernier est également stocké au format ELF.

```
bqu@:~$ xc32-gcc -T ldscript.ld -o
main1a.bin main1a.o
bqu@:~$
```

FIGURE 9 – Edition des liens du programme `main1a.o`.

Téléchargement. Il reste une dernière étape à réaliser afin que le programme puisse être téléchargé sur le microcontrôleur. Le format ELF contient des tas de détails qui sont inutiles au microcontrôleur et qui servent essentiellement au débogage. Avant de charger le programme dans le microcontrôleur, il faut en extraire juste les instructions et les données qui devront être copiées en mémoire Flash

5. Le *linker script* est à télécharger sur la plateforme Moodle.

et SRAM. Cette étape est réalisée avec l'outil `pic32-objcopy` et elle est illustrée à la Figure 10. Le fichier produit est stocké au format Intel Hex (option `-O ihex`) dans le fichier `main1a.hex`.

```
bqu@:~$ xc32-objcopy -O ihex main1a.bin
main1a.hex
bqu@:~$
```

FIGURE 10 – Production du fichier au format Intel HEX.

Il est maintenant possible de charger le programme dans le microcontrôleur. Pour cela, l'outil `avrdude` est utilisé tel qu'exposé à la Figure 8. L'option `-C` spécifie un autre fichier de configuration (celui contient la définition du processeur PIC32MX320F128H et est disponible sur Moodle). L'option `-p` spécifie le processeur cible. L'option `-c` spécifie le type de programmeur. L'option `-P` spécifie le port USB à utiliser. L'option `-U` spécifie les actions à réaliser avec le programmeur : écriture (w) en flash du fichier `main1a.hex`.

```
bqu@:~$ avrdude -C avrdude.conf -p pic32-360 -c stk500 -P/dev/ttyUSB0 -U
flash:w:main1a.hex

avrdude: AVR device initialized and ready to accept instructions

Reading | ##### | 100% 0.05s

avrdude: Device signature = 0x504943
avrdude: NOTE: FLASH memory has been specified, an erase cycle will
        be performed. To disable this feature, specify the -D option.
avrdude: current erase-rewrite cycle count is -1145324613 (if being tracked)
avrdude: erasing chip
avrdude: reading input file "main1a.hex"
avrdude: input file main1a.hex auto detected as Intel Hex
avrdude: writing flash (5332 bytes):

Writing | ##### | 100% 1.23s

avrdude: 5332 bytes of flash written
avrdude: verifying flash memory against main1a.hex:
avrdude: load data flash data from input file main1a.hex:
avrdude: input file main1a.hex auto detected as Intel Hex
avrdude: input file main1a.hex contains 5332 bytes
avrdude: reading on-chip flash data:

Reading | ##### | 100% 0.83s

avrdude: verifying ...
avrdude: 5332 bytes of flash verified

avrdude: safemode: Fuses OK

avrdude done. Thank you.

bqu@:~$
```

FIGURE 8 – Téléchargement du programme dans le microcontrôleur.

Les LEDs LD2 et LD3 devraient clignoter durant le téléchargement et la vérification. Une fois le téléchargement terminé, le *bootloader* devrait lancer l'exécution du programme *main1a* automatiquement.

2.5 Attendre un temps défini

Le programme créé à la section précédente (voir Figure 6) s'exécute au rythme de 80 millions d'instructions par seconde. Seules quelques instructions séparent les changements d'état de la broche RG6 du microcontrôleur. Par conséquent, la broche RG6 change d'état environ 20 millions de fois par seconde (cela génère un signal rectangulaire de fréquence égale à environ 10MHz). La LED LD4 semble allumée en permanence.

Afin de percevoir un clignotement, il est nécessaire d'espacer les changements d'état de RG6. Pour ce faire, une boucle "qui ne fait rien" durant un certain nombre d'itérations est introduite entre les changements d'état⁶.

Le code de la Figure 11 peut être inséré entre 2 changements d'état de RG6 de façon à introduire un délai. Notez que ce délai doit être introduit à 2 endroits. La fréquence obtenue avec ce délai devrait être égale à environ 1,33Hz.

```

1 | DELAY=10000000
2 |
3 |         li      $a3, DELAY
4 | loop:
5 |         addi    $a3, $a3, -1
6 |         bgtz    $a3, loop

```

FIGURE 11 – Boucle "qui ne fait rien".

2.6 Auto-évaluation

Questions simples

- Assurez-vous d'avoir bien compris le programme *main1a* et d'être capable de l'assembler, de le linker et de le télécharger dans le microcontrôleur. **Vérifiez que le résultat attendu est celui obtenu (clignotement de la LED) !!!**
- Afin de vérifier votre compréhension du programme *main1a*, changez-le de façon à faire clignoter la LED LD5 plutôt que la LED LD4. Ceci nécessite d'identifier le port contrôlant LD5 et les registres associés MAIS AUSSI de déterminer les valeurs à placer dans ces registres.
- Comment pourriez-vous mesurer manuellement et relativement précisément la fréquence de clignotement effective, i.e. celle qui est effectivement produite par l'exécution sur le microcontrôleur du programme avec délais ?

Questions supplémentaires.

- Déterminez à quelle adresse votre fonction *main* est placée en mémoire après l'exécution de l'éditeur de liens.
- Comment auriez-vous pu calculer manuellement la fréquence de changement de la broche RG6 en l'absence de délai (20 millions de changements par seconde) ? Pour rappel, le microcontrôleur exécute 80 millions d'instructions par seconde.

- Ecrivez une fonction *delay* qui prend la longueur du délai en argument (dans *a0*). Remplacez les boucles "qui ne font rien" par des appels à cette fonction. N'oubliez pas de sauvegarder le contenu du registre *ra*. Attention également aux registres que la fonction *delay* pourrait modifier (p.ex. *a0*) alors qu'ils sont peut-être utilisés par l'appelant.
- Pourriez-vous calculer précisément la valeur de la constante *DELAY* de façon à avoir une fréquence de clignotement de 1Hz exactement ? Quelle est la difficulté de ce calcul ?

Questions avancées.

- Allez lire le fichier source *crt0.S* dont le code objet est inclus automatiquement par le *linker* à votre programme. Déterminez quelles sont les opérations effectuées par le code de *crt0* et comment elles s'enchaînent.
- Faites varier l'intensité d'une LED avec la modulation en largeur d'impulsion (PWM — *Pulse Width Modulation*). L'idée de base de la modulation en largeur d'impulsion est d'allumer et éteindre régulièrement la LED, à une fréquence élevée fixe (de sorte que l'oeil ne perçoive pas le clignotement), par exemple à 1KHz. La durée d'un cycle est alors d'1ms.

Afin de faire varier l'intensité de la LED, la fraction (pourcentage) du temps durant laquelle la LED est allumée est variée. Plus cette fraction est grande (p.ex. 90%, i.e. 0,9ms), plus la LED éclaire fortement. Inversement, plus cette fraction est petite (p.ex. 10%, i.e. 0,1ms), moins la LED éclaire fortement.

6. Dans la suite du TP, nous verrons des moyens plus précis et plus efficaces d'attendre qu'un délai soit écoulé : *timers* et interruptions.

A Annexes

Cette annexe donne des détails supplémentaires sur les TP. **La lecture de cette annexe est optionnelle.**

A.1 Organisation interne du PIC32

La Figure 13 illustre le contenu du microcontrôleur PIC32. Il est possible d'y voir le coeur MIPS32 qui est relié au travers d'une *bus matrix* à la mémoire RAM et à la mémoire Flash. Le coeur MIPS est également relié aux ports d'entrées/sorties $PORTx$ (gauche du schéma) et aux autres périphériques (droite du schéma) au travers d'un *peripheral bridge*.

A.2 Connexion processeur-LED

Cette section donne des détails supplémentaires sur la façon dont le processeur et une LED sont connectés électriquement. Ces connexions sont illustrées sur l'extrait du schéma électrique de la carte ChipKit montré à la Figure 12.

Dans un schéma électrique, plusieurs symboles sont utilisés pour désigner le type des différents composants. Une LED est symbolisée par une flèche terminée d'une barre, à côté de laquelle deux petites flèches figurent l'émission de lumière. Le symbole à 3 branches, l'une munie d'une flèche désigne un transistor. Le symbole "zigzag" désigne une résistance. $VCC3V3$ et GND désignent respectivement les pôles positif et négatif de l'alimentation du circuit. $RG6$ et $RF0$ désignent des broches du microcontrôleur qui sont liées à des ports d'entrée/sortie.

Le principe du circuit est très simple. Le transistor est utilisé comme un interrupteur commandé par un courant électrique. Un courant peut circuler du collecteur à l'émetteur si un courant circule de la base à l'émetteur. Si c'est le cas, on dit que le transistor est "passant" (interrupteur fermé), sinon on dit que le transistor est "bloquant" (interrupteur ouvert).

Le transistor Q2A agit comme un interrupteur pour le circuit composé de LD4 et R35. Lorsque le transistor Q2A est "passant",

un courant peut circuler du collecteur vers l'émetteur, et donc à travers LD4. Cette situation est illustrée par la flèche rouge à la Figure 12b. La résistance R35 sert à limiter le courant qui traverse LD4. Lorsque le transistor est "bloquant" (i.e. interrupteur ouvert), aucun courant ne peut traverser LD4.

Pour que Q2A soit passant, il faut qu'un courant le traverse de la base à l'émetteur. Un tel courant est présent si la différence de potentiel entre la broche $RG6$ et GND est supérieure à environ $0.7V$ ⁷. La documentation du PIC32 mentionne que si un port d'entrée/sortie est en état logique haut (1), la broche correspondante présente une tension supérieure à $2,4V$ (V_{OH}). Si le port est en état logique bas (0), la tension est inférieure à $0,4V$ (V_{OL}).

Un circuit identique contrôle LD5 à partir de la broche $RF0$ du microcontrôleur et au travers du transistor Q2B.

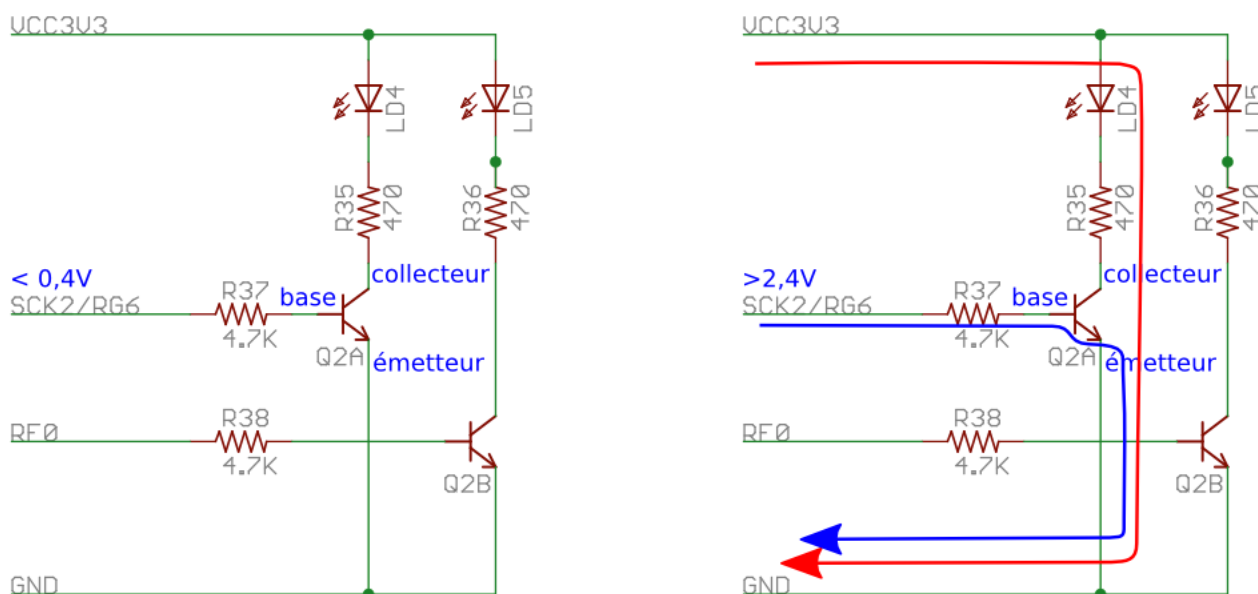
A.3 Structure d'une broche d'un port GPIO.

La Figure 14 illustre la structure générale d'une seule broche d'un port GPIO. La partie connectée à l'extérieur du microcontrôleur est désignée par *I/O pin* et se situe à la droite du schéma. Le terme *I/O Cell* désigne l'électronique qui permet de contrôler l'état de la broche (le triangle dirigé vers la droite) ou de lire l'état de la broche (triangle dirigé vers la gauche).

La partie gauche, intitulée *Dedicated Port Module*, schématise la logique de contrôle associée à cette broche. Les parties intéressantes sont les bascules bistables de type D nommée $ODCx$, $TRISx$ et $LATx$ qui se trouvent au centre du schéma. Ces bascules implémentent chacune 1 bit des registres correspondants. La sortie du port est désactivée si la bascule $TRISx$ vaut 0 et est activée sinon. La valeur écrite vers l'*I/O Cell* provient de la bascule $LATx$. La valeur lue peut être obtenue en lisant $PORTx$. Il est à noter qu'en lecture, $PORTx$ n'est pas vraiment un registre (il ne s'agit pas de la lecture d'une bascule bistable).

Les entrées D des différentes bascules associées à la broche n

7. Il s'agit d'une caractéristique propre au transistor utilisé



(a) LED LD4 éteinte.

(b) LED LD4 allumée.

FIGURE 12 – Extrait du schéma de la carte ChipKit Uno32 (source : Digilent Inc).

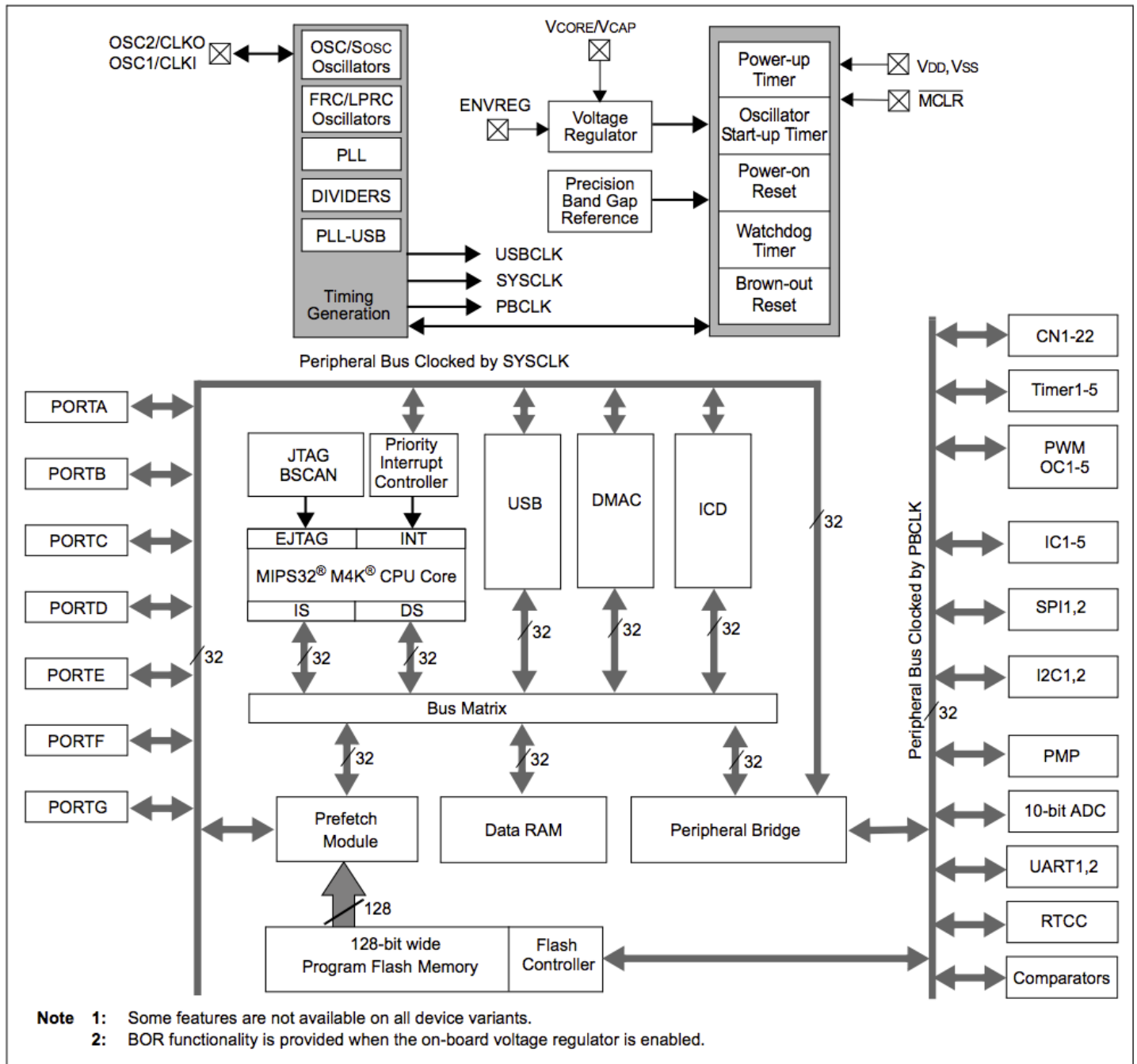


FIGURE 13 – Aperçu de l'organisation interne du microcontrôleur PIC32 (source : Microchip Inc).

du port x sont connectées au bit n du bus de données. Les entrées CK (clock) des bascules sont connectées au signal SYSCLK qui est l'horloge du bus. Les signaux de lecture ou d'écriture des bascules tels que RD TRIS x et WR TRIS x ne sont activés que lorsque l'adresse correspondante du registre TRIS x est présentée sur le bus d'adresse.

A.4 Code object : ELF

Le format des fichiers objets utilisés dans ce TP est ELF (*Executable and Linkable Format*), un format largement utilisé dans le monde UNIX⁸ pour stocker des programmes exécutables, des li-

brairies et du code objet.

Il est possible d'inspecter le contenu de ce fichier avec plusieurs commandes standard telles que nm et objdump. La Figure 15 illustre comment la commande xc32-objdump peut être utilisée avec l'option -d pour désassembler le fichier objet produit à la Section 2.4. Cette commande va notamment lire chacune des instructions en binaire et afficher sa signification en langage d'assemblage.

Dans la sortie de la commande xc32-objdump, on peut voir, à gauche, la position des instructions en mémoire, affichée en hexadécimal. Par exemple, l'instruction qui suit le label main est à l'adresse 0x0, alors que l'instruction qui suit le label main_loop est à l'adresse 0x24. Ensuite, le code d'instruction de 32 bits

8. Le format ELF est utilisé sous Linux par exemple.

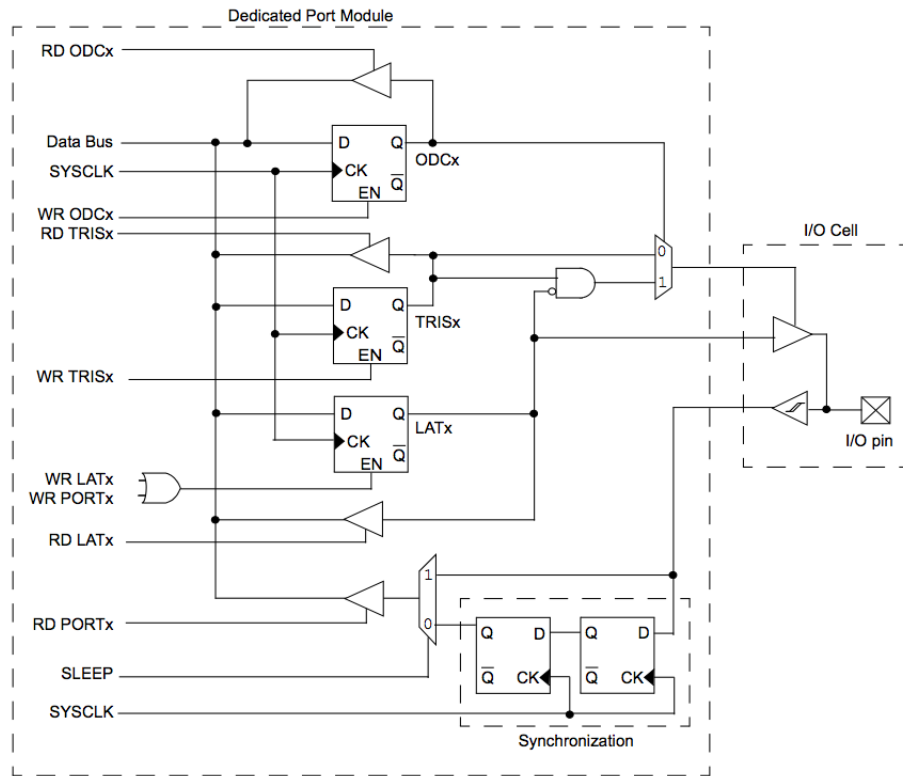


FIGURE 14 – Structure d'une broche d'un port GPIO du PIC32.

```
bqu@:~$ xc32-objdump -d main1a.o

src/main1a.o: file format elf32-tradlittlemips

Disassembly of section .text:

00000000 <main>:
  0: 3c060000    lui     a2,0x0
  4: 24c60000    addiu   a2,a2,0
  8: 8cc40000    lw      a0,0(a2)
 c: 2405ffbf    li      a1,-65
10: 00852024    and     a0,a0,a1
14: acc40000    sw      a0,0(a2)
18: 3c06bf88    lui     a2,0xbf88
1c: 34c661a0    ori     a2,a2,0x61a0
20: 8cc40000    lw      a0,0(a2)

00000024 <main_loop>:
24: 2405ffbf    li      a1,-65
28: 00852024    and     a0,a0,a1
2c: acc40000    sw      a0,0(a2)
30: 24050040    li      a1,64
34: 00852025    or      a0,a0,a1
38: acc40000    sw      a0,0(a2)
3c: 1000fff9    b       24 <main_loop>
40: 00000000    nop
bquoitin$
```

FIGURE 15 – Désassemblage du fichier objet main1a.o.

est affiché, en hexadécimal. La première instruction a le code 0x3c06bf88, ce qui correspond à l'instruction `lui`. Cette instruction et la suivante (`ori`) sont utilisées pour implémenter la pseudo-instruction `li` utilisée dans le programme en langage d'assemblage. On peut également constater que l'assembleur s'est permis d'ajouter une instruction à notre programme : un `nop` a été inséré après le `b main_loop`. Cette instruction est nécessaire à cause du *branch delay slot* introduit par le *pipeline* MIPS à la suite des instructions de branchement⁹.

A.5 Linker

Le *linker* remplit principalement les deux tâches suivantes :

1. **Adjonction de code objet supplémentaire.** Le linker adjoint au code objet `main1a.o` du code supplémentaire chargé d'initialiser certaines parties de la mémoire (notamment la pile) et certains registres. Cette initialisation est effectuée par des codes objets fournis avec les outils de compilation (notamment les fichiers `crt0.o`¹⁰ et `processor.o`¹¹).
2. **Re-positionnement du code en mémoire.** Le linker se charge de positionner les instructions et données à leurs adresses définitives. Les sections de code (p.ex. `.text`) et de données (p.ex. `.data`) doivent être positionnées à des emplacements précis. Ces emplacements sont typiquement

9. Il est possible de demander à l'assembleur de ne pas modifier le code fourni en ajoutant la directive `.set noreorder` dans le programme en langage d'assemblage. Cela permet de désactiver l'insertion d'instructions `nop`. Attention, cette directive est à utiliser avec précaution.

10. Le fichier `crt0.S` se trouve dans `pic32-libs/libpic32/startup`.

11. Le fichier `processor.o` se trouve dans `pic32-libs/proc/32MX320F128H`

documentés dans la datasheet du microcontrôleur. De plus, les 4GB d'espace adressable par le coeur MIPS ne sont pas tous implémentés dans le microcontrôleur. Certaines parties de cet espace d'adressage correspondent à de la mémoire SRAM, d'autres à de la mémoire Flash et d'autres encore à des registres SFRs (mappés en mémoire). Finalement, le microcontrôleur contient déjà un programme appelé *bootloader* qui se charge de recevoir les nouveaux programmes via la liaison USB. Ce *bootloader* occupe une partie de la mémoire flash qu'il ne faut pas écraser.

La position des différentes zones de mémoire du microcontrôleur et la correspondance avec les sections (`.text`, `.data`, etc) est décrite au linker dans un fichier appelé *linker script*. Un *linker script* spécifique doit être utilisé avec la plateforme ChipKit Uno32. Celui-ci est fourni sur la plateforme Moodle.

Le résultat produit par le *linker* est stocké au format ELF. Il est par conséquent possible de le décortiquer avec `objdump`¹². Cela permet par exemple d'observer à quelles adresses les différentes instructions et positions en mémoire ont été déplacées.

12. Cette manipulation est laissée comme exercice au lecteur