



Université d'Abomey-Calavi
The Abdus Salam International Center for Theoretical Physics
INSTITUT DE MATHEMATIQUES ET DE SCIENCES PHYSIQUES



Projet tutoré

Pour l'obtention du Diplôme de Licence spéciale des classes préparatoires
Option : **Informatique**

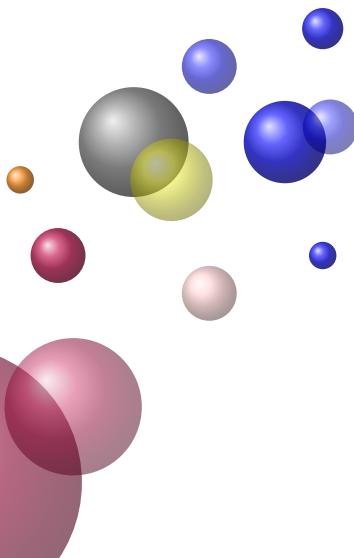
Conception d'un système d'exploitation pour raspberry

Etudiants :

David DOSSEH
Israël Jésucllo GODONOU

Encadreur :

Dr Hénoc SOUDE



Année Universitaire : 2020-2021

TABLE DES MATIÈRES

DEDICACES	iii
REMERCIEMENTS	iv
Acronyme	v
Résumé	viii
Abstract	ix
Introduction	x
1 Le Raspberry Pi	1
1.1 Présentation	1
1.1.1 Le micro ordinateur Pi	1
1.1.2 Spécifications matérielles et architectures	2
1.1.3 Les périphériques du Pi configurés	6
2 L'ARM V8	14
2.1 Présentation	14
2.2 Développement de l'ARM au fil des années	14
2.3 Propriétés du processeur ARMv8-A	15
2.4 Le processeur Cortex-A53	15
2.5 Jeux d'instructions ARMv8	16
2.5.1 Description	16
2.5.2 Les registres utilisés dans l'ISA 64 pour l'arm v8	17
2.5.3 Traitement des données	20
2.6 Privilèges et niveaux d'exception	21
2.7 Les types de privilège	21
2.8 Le privilège dans le système de mémoire	21
2.9 L'accès aux registres	22
2.10 Gestion des exceptions dans l'aarch64 armv8-A	22
2.10.1 Vecteur d'exception	23

2.10.2	Table vectorielle	24
2.11	Activation des exceptions asynchrones	25
2.12	Routage des exceptions asynchrones	25
2.13	Temporisateur générique	25
2.13.1	Les minuteries du processeur	26
2.14	Les registres du temporisateur	26
2.14.1	Configuration des temporiseurs	27
2.15	Le Compilateur croisé	28
2.15.1	La compilation croisée	28
2.15.2	La chaîne de compilation aarch64-gnu-linux	28
2.15.3	Les étapes de la compilation	28
3	Le Bootstrapping sur le Pi	31
3.1	Introduction	31
3.2	La carte SD	31
3.2.1	Paramètres du Pi : config.txt	31
3.2.2	Paramètres de l'OS : cmdline.txt	32
3.2.3	Le fichier start.elf	32
3.2.4	Le fichier fixup.dat	32
3.2.5	Le fichier bootcode.bin	33
3.3	Le bootstrapping sur le Pi	33
4	Architecture et implémentation de l'OS	34
4.1	Modélisation	34
4.2	Hardware	34
4.2.1	Le contrôleur d'interruption	34
4.2.2	Le Scheduler	39
4.3	Les composants	39
4.3.1	GPIO	39
4.3.2	Mini UART	41
4.3.3	Timer	42
5	Conclusion	45
6	Bibliographie	46
7	Webographie	48

DEDICACES

Je dédie ce travail au Saint Esprit pour sa personnalité et son oeuvre de chaque jour en moi. Je le dédie également à maman et papa pour leur travail et implication dans ma vie.

REMERCIEMENTS

Je tiens à exprimer mes remerciements à Dieu, mon Père pour qui il est et ce qu'il ne cesse d'accomplir en moi pour que je sois une bénédiction pour tout le monde entier.

J'exprime également ma reconnaissance et mes remerciements à mes chers parents, papa, maman mes frères et mes soeurs, ainsi que mes maîtres à quelque niveau que ce soit pour leur investissement dans le champ que je suis.

J'adresse aussi mess sincères remerciements à mon cher camarade David DOSSEH pour tous ses efforts et ses peines dans la réussite de ce projet, sachant qu'il n'est pas le fruit d'une seule personne, mais d'un travail à deux.

Je voudrais également remercier le Dr Hénoc SOUDE pour son suivi et son accompagnement pour l'aboutissement de ce travail. Sans lui ce travail n'aurait pas un sort.

Nous ne saurions terminer cette liste sans remercier tous ceux qui de près ou de loin ont permis la réalisation de ce travail.

ACRONYME

- **OS** : Operating System
- **Pi** : Raspberry Pi
- **ISA** : Instruction Set Architecture (Architecture de jeu d'instruction)
- **GIC** : Generic Interrupt controller

TABLE DES FIGURES

1.1	Pi modèle A	2
1.2	Pi modèle B Revision 2	3
1.3	Pi modèle 2 B	4
1.4	Pi modèle 3 B	5
1.5	Les pins du GPIO	6
1.6	Les pins du GPIO avec descriptions	7
1.7	Les registres du GPIO et leur adresse	7
1.8	Les registres du GPIO et leur adresse	8
1.9	Le registre GPFSEL1	9
1.10	Les registres GPSET	9
1.11	Les registres GPCLR	10
1.12	Les registres GPCLR	10
1.13	Les registres GPLEV	10
1.14	Les fonctions alternatives	11
1.15	Le registre principale du mini UART	13
2.1	Vue d'ensemble des versions de l'ARM	14
2.2	Caractéristiques des processeurs A53 et A57	15
2.3	Processeur A53.	16
2.4	Registres à usage général.	17
2.5	Exemple de code C et assembleur armv8.	20
2.6	Niveau d'exception	21
2.7	Etats d'exécution et niveau d'exécution	22
2.8	Structure de la table vectorielle pour EL1 en mémoire	23
2.9	Structure de la table vectorielle	24
2.10	Routage des exceptions dans EL3	25
2.11	Timer générique de l'ARM	26
2.12	Minuteurs par niveau	26
2.13	Régistres de minuterie	27
2.14	Temporisateur pour chaque niveau d'exception	27
4.1	Schéma global de l'architecture de l'os implémenté	34

4.2 Table vectorielle	35
---------------------------------	----

RÉSUMÉ

Nous avons mis en place un système d'exploitation rudimentaire pour Raspberry Pi. Pour ce faire, nous avons écrit et configuré les pilotes essentiels, ainsi qu'implémenter les concepts de base d'un OS. Plus besoin donc d'avoir une image Raspbian pour interagir avec le Pi. N'importe qui peut écrire son code ou module et l'intégrer au code existant pour construire une application selon ses besoins.

ABSTRACT

We have implemented a rudimentary operating system for raspberry. To do this, we wrote and configured the essential drivers, as well as implemented the basic concepts of an OS. So you no longer need to have a Raspbian image to interact with the Pi. Anyone can write their code or module and integrate it into existing code to build their application according to their needs.

INTRODUCTION

Avec l'avènement de l'internet des objets, l'Institut de Mathématiques et de Sciences Physique a décidé de rendre accessible ces nouvelles technologies aux jeunes étudiants. Dans ce cadre, il nous a été demandé d'implémenter un système d'exploitation permettant non seulement de connaître les différents composants d'un système d'exploitation mais aussi de découvrir les fonctionnalités de la carte Raspberry. En effet, la carte Raspberry est beaucoup utilisé dans le domaine de l'internet des objets.

Cahier des charges : Le travail consiste dans un premier temps à fournir une documentation complète présentant les périphériques du Raspberry et leurs fonctionnalités. Ensuite nous devons proposer un OS modulaire offrant les modules de bases permettant aux étudiants de construire des application selon leur besoin. Notre système doit être mono-tâche et avoir un ordonnanceur permettant de planifier les tâches selon leur ordre d'arrivé.

Réalisations : Nous avons pu offrir dans notre document une vue générale de l'architecture du processeur(ARMv8) et des explications sur les périphériques du matériel tels que : les broches **GPIO** pour la communication avec le monde extérieur(les leds par exemple) ; le **mini UART**, pour la lecture et écriture sur les ports séries ; le **timer générique de l'ARM** qui offre une utilisation du temps ; et les **interruptions**.

CHAPITRE 1

LE RASPBERRY PI

1.1 Présentation

1.1.1 Le micro ordinateur Pi

Le Raspberry Pi est un petit ordinateur, à processeur ARM, de la taille d'une carte de crédit, sorti en avril 2012. Il est conçu par les professeurs du département informatique de l'université de Cambridge dans le cadre de la **fondation Raspberry Pi**, une organisation caritative dont le but est de promouvoir les bases de la programmation informatique dans les écoles. Il fut créé afin de démocratiser l'accès aux ordinateurs et au digital making (terme anglophone désignant à la fois la capacité de résolution de problèmes et les compétences techniques et informatiques).

Cette démocratisation est possible en raison du coût réduit du Raspberry Pi, mais aussi grâce aux logiciels libres. Le Pi permet l'exécution de plusieurs variantes du système d'exploitation libre GNU/Linux, notamment Debian, et des logiciels compatibles. Il fonctionne également avec le système d'exploitation Microsoft Windows : Windows 10 IoT Core, Windows 10 on ARM (pour l'instant relativement instable), celui de Google Android Pi et même une version de l'OS/MVT d'IBM accompagnée du système APL 360.

Il est fourni nu, c'est-à-dire la carte mère seule, sans boîtier, câble d'alimentation, clavier, souris ni écran, dans l'objectif de diminuer les coûts et de permettre l'utilisation d'anciens matériels. Néanmoins des « kits » regroupant le « tout en un » sont disponibles sur le web à partir de quelques dizaines d'euros seulement.

Le Raspberry Pi a la capacité d'accueillir une carte électronique qui se fixe sur ses ports GPIO. Ces cartes peuvent ajouter des fonctionnalités, mais ont besoin des connaissances de l'utilisateur dans le domaine du codage. Elles sont surnommées « HAT », acronyme de **Hardware Attached on Top** (« matériel informatique attaché au-dessus »). le Pi est utilisé par des créateurs et bricoleurs du monde entier car son prix et sa taille le rendent très attrayant. Il existe plusieurs projets, tous plus fous les uns que les autres, qui se base sur le Pi. Nous pouvons citer : Une machine à café contrôlée par la voix, un téléphone VOIP, un bateau autonome, un serveur de backup, une station météo, un ordinateur de bord

pour ballon stratosphérique, un drone, un serveur média, des applications en domotique... Pour l'expérimenter, nous nous avons opté pour l'écriture d'un OS assez rudimentaire.

1.1.2 Spécifications matérielles et architectures

Le Pi possède un processeur ARM. Il inclut 1, 2 ou 4 ports USB, un port RJ45 et 256 Mo de mémoire vive pour le modèle d'origine (jusqu'à 8 Go sur les dernières versions). Son circuit graphique BMC VideoCore permet de décoder des flux Blu-Ray full HD (1080p 30 images par seconde), d'émuler d'anciennes consoles et d'exécuter des jeux vidéo relativement récents.

Modèles A

- Modèle 1 A

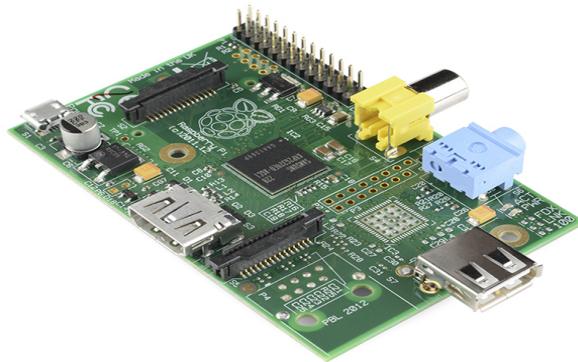


FIGURE 1.1 – Pi modèle A

Processeur : ARM1176JZF-S (ARMv6) 700 MHz Broadcom 2835 (dispose d'un décodeur Broadcom VideoCore IV, permettant le décodage H.264 FullHD 1080P et d'un VFPv2 pour le calcul des opérations à virgule) ;

RAM : 256 Mo ;

2 Sorties vidéo : Composite et HDMI ;

1 Sortie audio : stéréo Jack 3,5 mm (sortie son 5.1 sur la prise HDMI) ;

Unité de lecture-écriture de carte mémoire : SDHC / MMC / SDIO ;

1 Port USB 2.0 ;

Prise pour alimentation Micro-USB (consommation : 400 mA + périphériques) ;

Des entrées/sorties supplémentaires sont accessibles directement sur la carte mère via des pins 3v3 : GPIO, S2C, I2C, SPI ;

API logicielle vidéo : OpenGL : version embarquée OpenGL ES 2.0 ;

Décodage vidéo : 1080p30 H.264 high-profile.

— **Modèle 1 A+**

Différences avec le modèle A : Plus petit : 65 mm de long (contre 86 mm) ;
Lecteur de carte microSD en lieu et place du lecteur SD ;

GPIO : 40 broches ;

Nouveau chipset audio ;

Consommation électrique moindre ;

Prix réduit à 20\$.

Modèles B

Il existe plusieurs révisions du modèle B.

— **Modèle 1 B**

Spécificités du modèle B rev1 :

2 ports USB 2.0 au lieu de l'unique port du modèle A, mais sur un seul bus, via le composant SMSC LAN9512 ;

1 port réseau Fast Ethernet (10/100 Mbit/s) via le même composant SMSC. Le circuit LAN9512 qui gère les deux ports USB et le port réseau, est connecté au CPU via un unique port USB ; la bande passante est donc partagée entre ces trois ports.

Spécificités du modèle Rev1 + ECN0003 :

Suppression des fusibles protégeant les sorties USB ;

Suppression de la diode D14, qui pouvait provoquer des interférences avec des périphériques possédant une broche CEC, lorsque le Raspberry restait connecté sans être alimenté.

Spécificités du Raspberry Pi B Rev2 :



FIGURE 1.2 – Pi modèle B Revision 2

Implantation du reset (en reliant les broches 1 et 2 de P6) ;
Support JTAG (deux broches GPIO interchangées) ;

Support I2C (canaux primaire et secondaire inversés) ;
 Suppression de quatre signaux GPIO utilisés pour l'identification de version, et réaffectation à d'autres rôles ;
 SMSC +1V8 ;
 Deux trous de fixation ;
 Correction du marquage des LED sur la platine.

Spécificités du Raspberry Pi B 512 Mo :

Prise pour alimentation micro-USB (consommation : 700 mA) ;
 La RAM passe à 512Mo (au lieu de 256 Mo sur les modèles précédents) ;
 Le modèle est estampillé avec la référence 4G en lieu et place de l'ancienne référence 2G.

— Modèle 1 B+

Différences par rapport au modèle initial :

GPIO : 40 broches ;
4 ports USB 2.0 et meilleur comportement en cas de surcharge ;
micro SD ;
 réduction de consommation de 3,5W à 3W ;
 suppression de la prise RCA au profit d'une prise mini-jack 4 points, comprenant une sortie sonore et vidéo ;

— Modèle 2 B (Raspberry Pi 2)

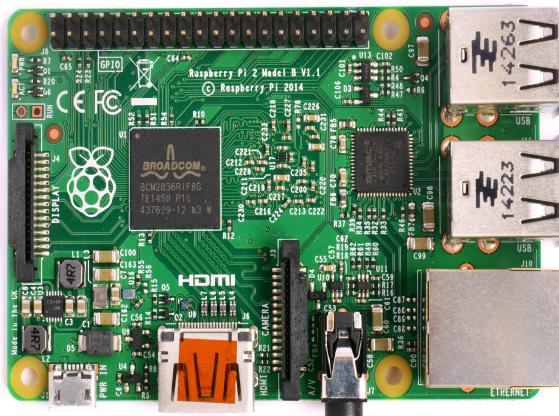


FIGURE 1.3 – Pi modèle 2 B

Le 2 février 2015, la fondation Raspberry Pi annonce la sortie du Raspberry Pi 2, plus puissant, il est équipé d'un processeur Broadcom BCM2836, quatre coeurs Cortex-A7 (ARMv7) à 900 MHz, accompagné de 1Go de RAM. Il possède les mêmes dimensions et la même connectique que le modèle B+.

— Modèle 3 B (Raspberry Pi 3)



FIGURE 1.4 – Pi modèle 3 B

Le 29 février 2016, pour le quatrième anniversaire de la commercialisation du premier modèle, la fondation Raspberry Pi annonce la sortie du Raspberry Pi 3. Comparé au Pi 2, il dispose d'un processeur Broadcom BCM2837 64 bit à quatre cœurs ARM Cortex-A53 à 1,2 GHz, d'une puce Wifi 802.11n et Bluetooth 4.1 intégrée. Il possède les mêmes dimensions et connectiques que les modèles 2 et B+. La vitesse d'horloge est 33% plus rapide que le Pi 2, ce qui permet d'avoir un gain d'environ 50-60% de performance en mode 32 bits. Il est recommandé d'utiliser un adaptateur de 2,5 A. Tous les travaux et tutoriels du Pi 2 sont parfaitement compatibles avec le Pi 3.

— Modèle 3 B+ (Raspberry Pi 3+)

Le 14 mars 2018, la fondation Raspberry Pi annonce la mise à jour du Raspberry Pi 3 vers le modèle B+. On y trouve une mise à jour du processeur Broadcom BCM2837B0 64 bit à quatre cœurs ARM Cortex-A53 cadencé à 1,4 GHz au lieu du 1,2 GHz. la puce Cypress CYW43438 est remplacée par une nouvelle puce CYW43455 supportant le WiFi Dual-band 802.11ac et la version 4.2 du Bluetooth. Et d'une prise en charge du Power over Ethernet grâce à un élément supplémentaire.

— Modèle 4 B (Raspberry Pi 4)

Le 24 juin 2019, la fondation Raspberry Pi annonce la sortie du Raspberry Pi 4. Le 10 juillet 2019, la fondation annonce des problèmes de conception, notamment l'absence d'une résistance, qui ne permettent pas à certains chargeurs USB-C d'alimenter le Raspberry Pi. La fondation annonce que le problème sera corrigé dans une révision future du produit. Dans les faits, seuls les câbles « e-marked » posent problème parce qu'ils détectent le Raspberry Pi comme étant un équipement audio. Le 28 mai 2020 est annoncée une version à 8Go de RAM Bien que Raspbian ne soit que 32 bits, les 8 Go peuvent être exploités via des processus indépendants lancés sous ce système. Une version 64 bits est annoncée, elle aussi branche de Debian, qui ne se nommera plus Raspbian, mais Raspberry Pi OS pour éviter les confusions, et identique en interface.

En novembre 2020, elle lance une version du Raspberry Pi 4 (**Raspberry Pi 400**)

permettant de monter facilement un ordinateur de bureau en intégrant dans un clavier l'ensemble des composantes d'un ordinateur personnel et auquel des périphériques peuvent être branchés (écran, souris, etc.).

Les autres versions du Pi ainsi que leurs caractéristiques sont disponibles sur le web. Pour notre expérience, nous avons eu à travailler sur le Pi 3 B+.

1.1.3 Les périphériques du Pi configurés

Malgré sa petite taille, le Raspberry Pi dispose de tous les périphériques que tout nano-ordinateur digne de ce nom doit avoir. Par ailleurs, l'avantage du Raspberry Pi est que de par sa structure monocarte (une seule carte mère), toutes les connectiques soient fixées sur un même support. Ce qui rend l'appareil tout aussi compact que pratique. Dans cette partie nous allons présenter les périphériques que nous avons utilisé dans notre OS ainsi que leur plan d'adressage.

GPIO

— Description

Une caractéristique puissante du Raspberry Pi est la rangée de broches **GPIO**, pour General Purpose Input Output (entrée / sortie à usage général), le long du bord supérieur de la carte. On dispose de 54 pins GPIO divisé en 2 banques possédant au moins deux fonctions alternatives. Mais sur le Pi, nous comptons sur l'en-tête 40 pins et ceci se trouve sur toutes les cartes Raspberry Pi actuelles (non peuplées sur Pi Zero et Pi Zero W). Avant le Pi 1 Model B+ (2014), les cartes comprenaient un en-tête plus court à 26 broches. Deux broches 5V et deux broches 3V3 sont présentes sur la carte, ainsi qu'un certain nombre de broches de masse (0V), qui ne sont pas configurables. Les broches restantes sont toutes des broches 3V3 à usage général, ce qui signifie que les sorties sont réglées sur 3V3 et que les entrées sont tolérantes à 3V3. Un pin peut être programmé à être comme entrée ou une sortie et certains d'entre eux sont par défaut désignés pour interagir avec d'autres périphériques du Pi. Il faudra donc pour cela activer sur ces pins des fonctions alternatives.

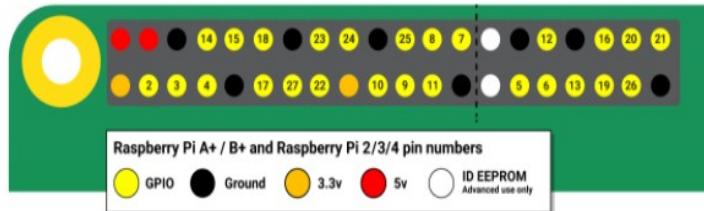


FIGURE 1.5 – Les pins du GPIO

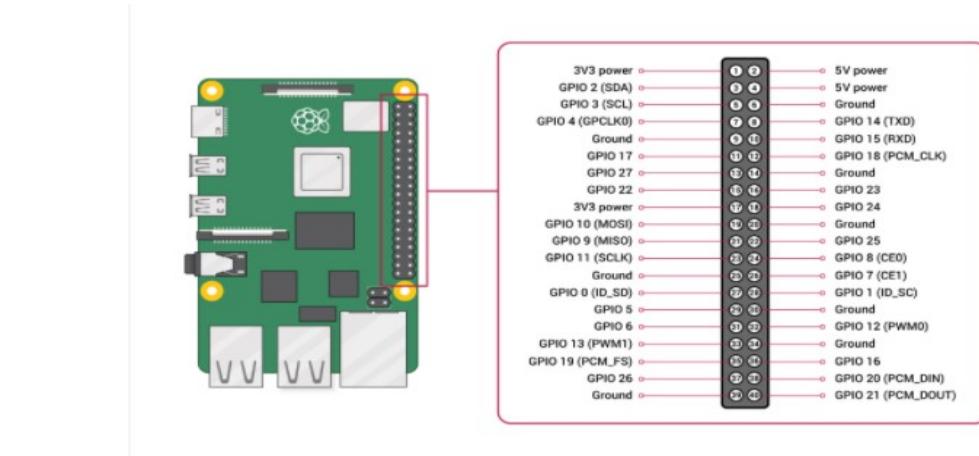


FIGURE 1.6 – Les pins du GPIO avec descriptions

— Les registres du GPIO

GPIO fournit un tas de registres. Chaque bit d'un tel registre correspond à une broche sur la carte Pi. En écrivant 1 ou 0 pour enregistrer les bits, le logiciel peut contrôler la tension de sortie sur les broches, par exemple pour allumer/éteindre les LED connectées à ces broches.

Le GPIO dispose 41 registres dont les accès sont supposés être de 32 bits. La figure suivante montre tous les registres du Pi ainsi que leur adresse.

Address	Field Name	Description	Size	Read/Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W
0x 7E20 0008	GPFSEL2	GPIO Function Select 2	32	R/W
0x 7E20 000C	GPFSEL3	GPIO Function Select 3	32	R/W
0x 7E20 0010	GPFSEL4	GPIO Function Select 4	32	R/W
0x 7E20 0014	GPFSEL5	GPIO Function Select 5	32	R/W
0x 7E20 0018	-	Reserved	-	-
0x 7E20 001C	GPSET0	GPIO Pin Output Set 0	32	W
0x 7E20 0020	GPSET1	GPIO Pin Output Set 1	32	W
0x 7E20 0024	-	Reserved	-	-
0x 7E20 0028	GPCLR0	GPIO Pin Output Clear 0	32	W
0x 7E20 002C	GPCLR1	GPIO Pin Output Clear 1	32	W
0x 7E20 0030	-	Reserved	-	-
0x 7E20 0034	GPLEV0	GPIO Pin Level 0	32	R
0x 7E20 0038	GPLEV1	GPIO Pin Level 1	32	R
0x 7E20 003C	-	Reserved	-	-
0x 7E20 0040	GPEDS0	GPIO Pin Event Detect Status 0	32	R/W
0x 7E20 0044	GPEDS1	GPIO Pin Event Detect Status 1	32	R/W
0x 7E20 0048	-	Reserved	-	-
0x 7E20 004C	GPREN0	GPIO Pin Rising Edge Detect Enable 0	32	R/W
0x 7E20 0050	GPREN1	GPIO Pin Rising Edge Detect Enable 1	32	R/W
0x 7E20 0054	-	Reserved	-	-
0x 7E20 0058	GPFEN0	GPIO Pin Falling Edge Detect Enable 0	32	R/W
0x 7E20 005C	GPFEN1	GPIO Pin Falling Edge Detect Enable 1	32	R/W

FIGURE 1.7 – Les registres du GPIO et leur adresse

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0060	-	Reserved	-	-
0x 7E20 0064	GPHEN0	GPIO Pin High Detect Enable 0	32	R/W
0x 7E20 0068	GPHEN1	GPIO Pin High Detect Enable 1	32	R/W
0x 7E20 006C	-	Reserved	-	-
0x 7E20 0070	GPLENO	GPIO Pin Low Detect Enable 0	32	R/W
0x 7E20 0074	GPLEN1	GPIO Pin Low Detect Enable 1	32	R/W
0x 7E20 0078	-	Reserved	-	-
0x 7E20 007C	GPAREN0	GPIO Pin Async. Rising Edge Detect 0	32	R/W
0x 7E20 0080	GPAREN1	GPIO Pin Async. Rising Edge Detect 1	32	R/W
0x 7E20 0084	-	Reserved	-	-
0x 7E20 0088	GPAFEN0	GPIO Pin Async. Falling Edge Detect 0	32	R/W
0x 7E20 008C	GPAFEN1	GPIO Pin Async. Falling Edge Detect 1	32	R/W
0x 7E20 0090	-	Reserved	-	-
0x 7E20 0094	GPPUD	GPIO Pin Pull-up/down Enable	32	R/W
0x 7E20 0098	GPPUDCLK0	GPIO Pin Pull-up/down Enable Clock 0	32	R/W
0x 7E20 009C	GPPUDCLK1	GPIO Pin Pull-up/down Enable Clock 1	32	R/W
0x 7E20 00A0	-	Reserved	-	-
0x 7E20 00B0	-	Test	4	R/W

FIGURE 1.8 – Les registres du GPIO et leur adresse

La configuration d'un pin nécessite l'écriture dans ces registres. Nous parlerons uniquement de certains d'entre eux. La description complète de tous les registres est disponible dans le manuel du processeur bcm2837.

Les registres GPFSELn

les registres **GPFSEL**, pour GPIO Fonction Select, sont utilisés pour définir des opérations sur les pins GPIO. Ce sont ces registres qui permettent de configurer un pin comme entrée ou sortie ou d'attribuer une fonction alternative à un pin. On distingue 6 registres GPFSEL0-5 dont chacun des 5 premiers s'occupe de 10 pins à partir du pin 0 dans l'ordre croissant et le dernier registre des 4 pins restants. Les pins sont numérotés de 0 à 53 et leur numéro ne correspond pas forcément à leur position sur le matériel. Pour configurer un pin n, il faut choisir le registre GPFSEL qui s'en occupe, accéder au n-ième champ de ce registre et écrire un nombre entre 0 et 7 en binaire sur 3 bit, selon l'objectif à atteindre. La figure suivante nous montre le registre GPFSEL1 responsable des pins 10 à 19).

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
29-27	FSEL19	FSEL19 - Function Select 19 000 - GPIO Pin 19 is an input 001 - GPIO Pin 19 is an output 100 - GPIO Pin 19 takes alternate function 0 101 - GPIO Pin 19 takes alternate function 1 110 - GPIO Pin 19 takes alternate function 2 111 - GPIO Pin 19 takes alternate function 3 011 - GPIO Pin 19 takes alternate function 4 010 - GPIO Pin 19 takes alternate function 5	R/W	0
26-24	FSEL18	FSEL18 - Function Select 18	R/W	0
23-21	FSEL17	FSEL17 - Function Select 17	R/W	0
20-18	FSEL16	FSEL16 - Function Select 16	R/W	0
17-15	FSEL15	FSEL15 - Function Select 15	R/W	0
14-12	FSEL14	FSEL14 - Function Select 14	R/W	0
11-9	FSEL13	FSEL13 - Function Select 13	R/W	0
8-6	FSEL12	FSEL12 - Function Select 12	R/W	0
5-3	FSEL11	FSEL11 - Function Select 11	R/W	0
2-0	FSEL10	FSEL10 - Function Select 10	R/W	0

FIGURE 1.9 – Le registre GPFSEL1

Les registres GPSET0 et GPSET1

Les registres **GPSET0** et **GPSET1** s'occupent respectivement des 0 - 31 ième premiers pins et des 32 - 53 ième derniers pins du GPIO. **GPSET**, pour GPIO Output Set register, est celui qui permet configurer un pin défini comme sortie. L'écriture d'un 0 dans un champ n'a aucun effet cependant l'écriture de 1 dans le champ 1 du registre, produit un effet si le pin n était déjà au préalable défini comme une sortie. La figure suivante nous montre les champs du registre.

Bit(s)	Field Name	Description	Type	Reset
31-0	SETn (n=0..31)	0 = No effect 1 = Set GPIO pin n	R/W	0

Table 6-8 – GPIO Output Set Register 0

Bit(s)	Field Name	Description	Type	Reset
31-22	-	Reserved	R	0
21-0	SETn (n=32..53)	0 = No effect 1 = Set GPIO pin n	R/W	0

Table 6-9 – GPIO Output Set Register 1

FIGURE 1.10 – Les registres GPSET

Les registres GPCLR0 et GPCLR1

GPCLR, pour GPIO Output Clear, sont des registres qui permettent de désactiver un pin défini comme sortie. Écrire 0 dans un champ n'a aucun effet, mais écrire 1 dans un champ désactive le pin correspondant s'il était configuré comme une sortie. Sinon cela n'aura aucun effet.

Bit(s)	Field Name	Description	Type	Reset
31-0	CLRn ($n=0..31$)	0 = No effect 1 = Clear GPIO pin n	R/W	0

Table 6-10 – GPIO Output Clear Register 0

Bit(s)	Field Name	Description	Type	Reset
31-22	-	Reserved	R	0

FIGURE 1.11 – Les registres GPCLR

21-0	CLRn ($n=32..53$)	0 = No effect 1 = Clear GPIO pin n	R/W	0
------	------------------------	---	-----	---

Table 6-11 – GPIO Output Clear Register 1

FIGURE 1.12 – Les registres GPCLR

Les registres GPLEV0 et GPLEV1

GPLEV, pour GPIO Level, sont des registres utilisés dans le but de connaître l'actuelle valeur d'un pin donné.

Bit(s)	Field Name	Description	Type	Reset
31-0	LEVn ($n=0..31$)	0 = GPIO pin n is low 1 = GPIO pin n is high	R/W	0

Table 6-12 – GPIO Level Register 0

Bit(s)	Field Name	Description	Type	Reset
31-22	-	Reserved	R	0
21-0	LEVn ($n=32..53$)	0 = GPIO pin n is low 1 = GPIO pin n is high	R/W	0

Table 6-13 – GPIO Level Register 1

FIGURE 1.13 – Les registres GPLEV

Les autres registres et leurs description se trouvent dans le manuel **bcm2837** du processeur.

— Les fonctions alternatives

En plus des périphériques d'entrée et de sortie simples, les broches GPIO peuvent être utilisées avec une variété de fonctions alternatives.

La plupart des broches peuvent être utilisées avec différents périphériques IO. Donc, avant d'utiliser une broche particulière, nous devons sélectionner la fonction alternative de la broche, un nombre de 0 à 5 qui peut être défini pour chaque broche et configurer quel périphérique IO est virtuellement "connecté" à la broche.

Voir la liste de toutes les fonctions alternatives GPIO disponibles dans l'image ci-dessous :

	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5
GPIO0	High	SDA0	SA5	<reserved>			
GPIO1	High	SCL0	SA4	<reserved>			
GPIO2	High	SDA1	SA3	<reserved>			
GPIO3	High	SCL1	SA2	<reserved>			
GPIO4	High	GPCLK0	SA1	<reserved>			ARM_TDI
GPIO5	High	GPCLK1	SA0	<reserved>			ARM_TDO
GPIO6	High	GPCLK2	SOE_N / SE	<reserved>			ARM_RTCK
GPIO7	High	SPI0_CE1_N	SWE_N / SBW_N	<reserved>			
GPIO8	High	SPI0_CE0_N	SD0	<reserved>			
GPIO9	Low	SPI0_MISO	SD1	<reserved>			
GPIO10	Low	SPI0_MOSI	SD2	<reserved>			
GPIO11	Low	SPI0_SCLK	SD3	<reserved>			
GPIO12	Low	PWM0	SD4	<reserved>			ARM_TMS
GPIO13	Low	PWM1	SD5	<reserved>			ARM_TCK
GPIO14	Low	TXD0	SD6	<reserved>			TXD1
GPIO15	Low	RXD0	SD7	<reserved>			RXD1
GPIO16	Low	<reserved>	SD8	<reserved>	CTS0	SPI1_CE2_N	CTS1
GPIO17	Low	<reserved>	SD9	<reserved>	RTS0	SPI1_CE1_N	RTS1
GPIO18	Low	PCM_CLK	SD10	<reserved>	BSCSL SDA / MISO	SPI1_CE0_N	PWM0
GPIO19	Low	PCM_FS	SD11	<reserved>	BSCSL SCL / MISO	SPI1_MISO	PWM1
GPIO20	Low	PCM_DIN	SD12	<reserved>	BSCSL MOSI / MISO	SPI1_MOSI	GPCLK0
GPIO21	Low	PCM_DOUT	SD13	<reserved>	BSCSL CE_N / CE_N	SPI1_SCLK	GPCLK1
GPIO22	Low	<reserved>	SD14	<reserved>	SD1_CLK	ARM_TRST	
GPIO23	Low	<reserved>	SD15	<reserved>	SD1_CMD	ARM_RTCK	
GPIO24	Low	<reserved>	SD16	<reserved>	SD1_DAT0	ARM_TDO	
GPIO25	Low	<reserved>	SD17	<reserved>	SD1_DAT1	ARM_TCK	
GPIO26	Low	<reserved>	<reserved>	<reserved>	SD1_DAT2	ARM_TDI	
GPIO27	Low	<reserved>	<reserved>	<reserved>	SD1_DAT3	ARM_TMS	
GPIO28	-	SDA0	SA5	PCM_CLK	<reserved>		
GPIO29	-	SCL0	SA4	PCM_FS	<reserved>		
GPIO30	Low	<reserved>	SA3	PCM_DIN	CTS0		CTS1
GPIO31	Low	<reserved>	SA2	PCM_DOUT	RTS0		RTS1
GPIO32	Low	GPCLK0	SA1	<reserved>	TXD0		TXD1
GPIO33	Low	<reserved>	SA0	<reserved>	RXD0		RXD1
GPIO34	High	GPCLK0	SOE_N / SE	<reserved>	<reserved>		
GPIO35	High	SPI0_CE1_N	SWE_N / SBW_N		<reserved>		
GPIO36	High	SPI0_CE0_N	SD0	TXD0	<reserved>		
GPIO37	Low	SPI0_MISO	SD1	RXD0	<reserved>		
GPIO38	Low	SPI0_MOSI	SD2	RTS0	<reserved>		
GPIO39	Low	SPI0_SCLK	SD3	CTS0	<reserved>		
GPIO40	Low	PWM0	SD4		<reserved>	SPI2_MISO	TXD1
	Pull	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5

FIGURE 1.14 – Les fonctions alternatives

UART

UART, pour Universal Asynchronous Receiver Transmitter, est un émetteur-récepteur asynchrone universel. En langage courant, c'est le composant utilisé pour faire la liaison entre l'ordinateur et le port série. l'UART est un dispositif de caractères simple permettant au logiciel d'envoyer des caractères de texte à une autre machine. Il nécessite un code logiciel très minimum.

Dans la forme la plus simple, le logiciel écrit des valeurs ASCII dans les registres UART. Le dispositif UART convertit les valeurs écrites en une séquence de hautes et basses tensions sur le fil. Cette séquence est transmise via le TTL-to-serial cable et est interprétée par un émulateur de terminal (par exemple PuTTY sous Windows, minicom sous Linux).

Les Pi sont composé de deux UART : le **Mini UART**, pour la transmission et le réception série et l'**UART0**, qui gère le périphérique Bluetooth.

UART primaire

Sur le Raspberry Pi, un UART est sélectionné pour être présent sur GPIO 14 (transmission) et 15 (réception) - c'est l'UART primaire. Par défaut, ce sera également l'UART sur lequel une console peut être présente. Notez que GPIO 14 est la broche 8 sur l'en-tête GPIO, tandis que GPIO 15 est la broche 10. C'est le Mini UART qui joue ce rôle pour le Pi 3 B+.

UART secondaire

L'UART secondaire n'est normalement pas présent sur le connecteur GPIO. Par défaut, l'UART secondaire est connecté au côté Bluetooth du contrôleur LAN/Bluetooth sans fil combiné, sur les modèles qui contiennent ce contrôleur. Il s'agit ici de l'UART0.

Qu'il soit primaire ou secondaire, le Mini UART est désactivé par défaut. Pour l'utiliser il faut configurer le Pi pour utiliser une fréquence d'horloge de base VPU fixe. C'est parce que l'horloge mini UART est liée à l'horloge principale VPU, de sorte que lorsque la fréquence d'horloge principale change, le débit en bauds UART change également. Les paramètres et peuvent être ajoutés pour modifier le comportement du mini UART.

Le mini UART a les caractéristiques suivantes :

- Fonctionnement sur 7 ou 8 bits.
- 1 bit de départ et 1 bit d'arrêt.
- Pas de parité.
- Génération de pause.
- FIFO de 8 symboles pour la réception et la transmission.
- Réception contrôlée par le logiciel, Transmission lisible par le logiciel.

- Contrôle automatique de flux avec niveau FIFO programmable.
- Débit en bauds dérivé de l'horloge système.

Aperçu des registres du Mini UART

Le registre **UX_MU_IO_REG**, situé à l'adresse 0x7E215040, est le registre principale pour accéder à la FIFO de l'UART.

Bit(s)	Field Name	Description	Type	Reset
31:8		Reserved, write zero, read as don't care		
7:0	LS 8 bits Baudrate read/write, DLAB=1	Access to the LS 8 bits of the 16-bit baudrate register. (Only If bit 7 of the line control register (DLAB bit) is set)	R/W	0
7:0	Transmit data write, DLAB=0	Data written is put in the transmit FIFO (Provided it is not full) (Only If bit 7 of the line control register (DLAB bit) is clear)	W	0
7:0	Receive data read, DLAB=0	Data read is taken from the receive FIFO (Provided it is not empty) (Only If bit 7 of the line control register (DLAB bit) is clear)	R	0

FIGURE 1.15 – Le registre principale du mini UART

Les autres registres du mini UART et leur descriptions sont décrits dans le manuel bcm2837 du processeur.

Le GPIO peut être utilisé pour configurer le comportement de différentes broches GPIO. Par exemple, pour pouvoir utiliser le Mini UART, nous devons activer les broches 14 et 15 et les configurer. Il nous faut alors avant tout attribuer la fonction alternative 5 sur chacune de ces broches à travers le registre GPSEL1, responsable des broches 14 et 15.

CHAPITRE 2

L'ARM V8

2.1 Présentation

L'architecture ARMv8-A est la dernière génération d'architecture ARM destinée au profil des applications. Le nom ARMv8 est utilisé pour décrire l'architecture globale, qui maintenant inclut à la fois l'exécution 32 bits et l'exécution 64 bits. Il introduit la capacité d'exécution avec des registres de 64 bits, tout en préservant la rétrocompatibilité avec les logiciel ARMv7 utilisant les registres 32 bits.

2.2 Développement de l'ARM au fil des années

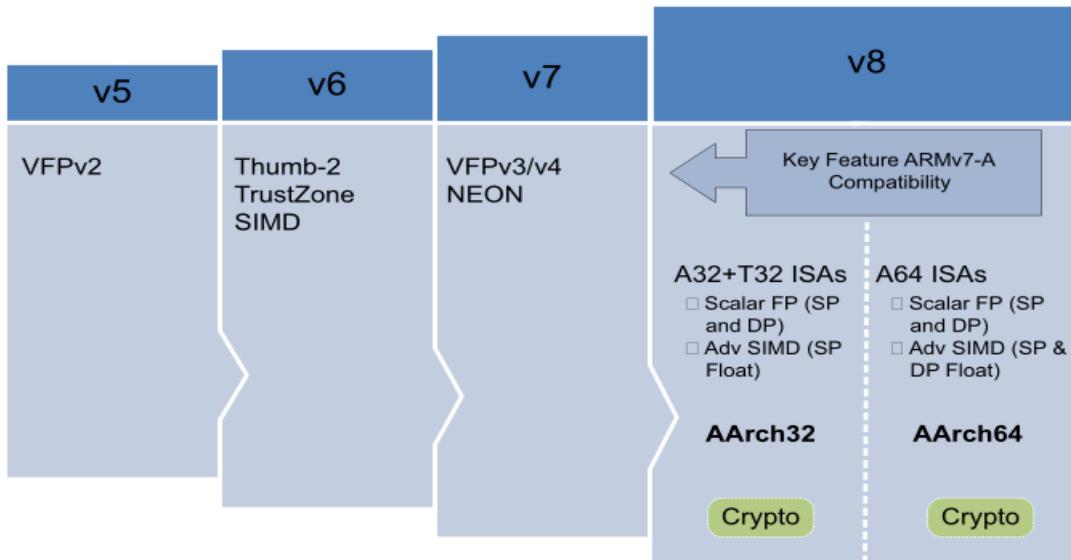


FIGURE 2.1 – Vue d'ensemble des versions de l'ARM

2.3 Propriétés du processeur ARMv8-A

La figure suivante compare les propriétés des implémentations de processeur d'ARM qui prennent en charge l'architecture ARMv8-A.

	Cortex-A53	Cortex-A57
Release date	July 2014	January 2015
Typical clock speed	2GHz on 28nm	1.5 to 2.5 GHz on 20nm
Execution order	In-order	Out of order, speculative issue, superscalar
Cores	1 to 4	1 to 4
Integer Peak throughput	2.3MIPS/MHz	4.1 to 4.76MIPS/MHz ^a
Floating-point Unit	Yes	Yes
Half-precision	Yes	Yes
Hardware Divide	Yes	Yes
Fused Multiply Accumulate	Yes	Yes
Pipeline stages	8	15+
Return stack entries	4	8
Generic Interrupt Controller	External	External
AMBA interface	64-bit I/F AMBA 4 (Supports AMBA 4 and AMBA 5)	128-bit I/F AMBA 4 (Supports AMBA 4 and AMBA 5)
L1 Cache size (Instruction)	8KB to 64 KB	48KB
L1 Cache structure (Instruction)	2-way set associative	3-way set associative
L1 Cache size (Data)	8KB to 64KB	32KB
L1 Cache structure (Data)	4-way set associative	2-way set associative
L2 Cache	Optional	Integrated
L2 Cache size	128KB to 2MB	512KB to 2MB
L2 Cache structure	16-way set associative	16-way set associative
Main TLB entries	512	1024
uTLB entries	10	48 I-side 32 D-side

FIGURE 2.2 – Caractéristiques des processeurs A53 et A57

2.4 Le processeur Cortex-A53

Le processeur Cortex-A53 est un processeur de milieu de gamme à faible consommation avec entre un et quatre coeurs dans un seul cluster, chacun avec un sous-système de cache L1, un GICv3/4 intégré en option interface et un contrôleur de cache L2 en option. Le processeur Cortex-A53 est un processeur extrêmement économique en énergie capable de prendre en charge du code 32 et 64 bits. Il offre des performances nettement plus élevées que le très réussi Processeur Cortex-A7. Il est capable de se déployer en tant que processeur d'applications autonome, ou associé au processeur Cortex-A57 dans une configuration

big.LITTLE pour des performances optimales, évolutivité et efficacité énergétique.

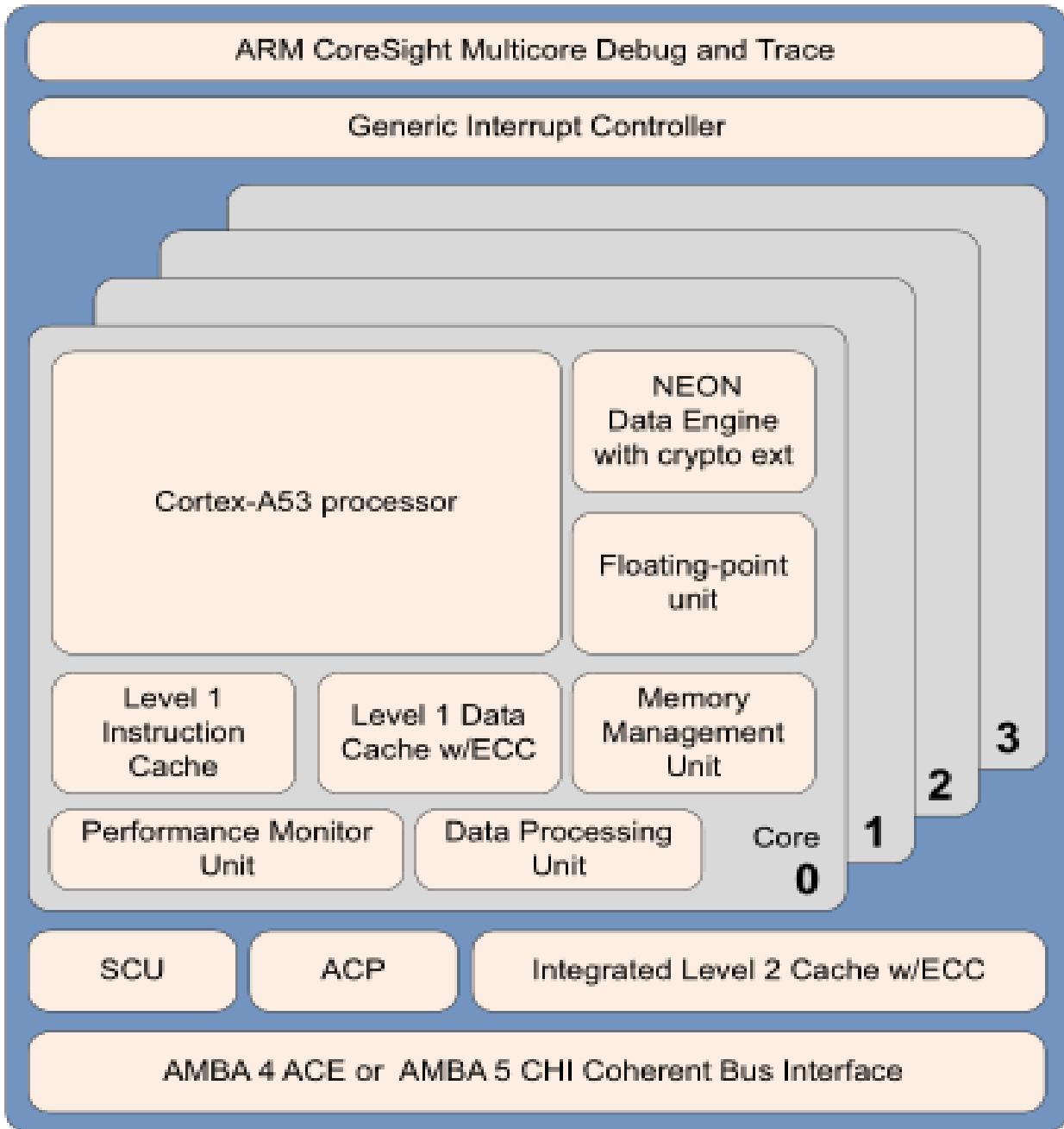


FIGURE 2.3 – Processeur A53.

2.5 Jeux d'instructions ARMv8

2.5.1 Description

Une architecture de jeu d'instructions (ISA) fait partie du modèle abstrait d'un ordinateur. Il définit comment le logiciel contrôle le processeur. L'un des changements les

plus importants introduits dans l'architecture ARMv8 est l'ajout d'un jeu d'instructions 64 bits. Ce jeu complète l'architecture existante du jeu d'instructions 32 bits. Cet ajout donne accès à des registres d'entiers et des opérations de données de 64 bits, et la possibilité d'utiliser des pointeurs de taille 64 bits vers la mémoire. Les nouvelles instructions sont appelées A64 et s'exécutent dans l'état d'exécution AArch64.

ARMv8 comprend également le jeu d'instructions ARM d'origine, désormais appelé A32 et le jeu d'instructions Thumb (T32). A32 et T32 s'exécutent tous les deux dans l'état AArch32, et fournissent une compatibilité avec ARMv7.

Bien que ARMv8-A offre une compatibilité avec les architectures ARM 32 bits, le jeu d'instructions A64 est séparé et distinct de l'ancien ISA et est codé différemment. A64 ajoute des fonctionnalités supplémentaires tout en supprimant d'autres fonctionnalités qui pourraient potentiellement limiter la vitesse ou l'efficacité énergétique des implementations hautes performances.

L'architecture ARMv8 inclut également quelques améliorations aux jeux d'instructions 32 bits (A32 et T32). Pourtant, le code qui utilise de telles fonctionnalités n'est pas compatible avec les anciennes implémentations ARMv7.

2.5.2 Les registres utilisés dans l'ISA 64 pour l'arm v8

Les registres à usage général

La plupart des instructions A64 fonctionnent sur des registres. L'architecture fournit 31 registres à usage général. Chaque registre peut être utilisé comme un registre de 64 bits (X0..X30), ou comme un registre de 32 bits (W0..W30). Ce sont deux manières distinctes de considérer le même registre. Par exemple, ce diagramme de registre montre que W0 correspond aux 32 bits inférieurs de X0 et W1 correspond aux 32 bits inférieurs de X1 :



FIGURE 2.4 – Registres à usage général.

Pour les instructions de traitement de données, le choix de X ou W détermine la taille de l'opération. L'utilisation des registres X entraînera des calculs 64 bits et l'utilisation des registres W entraînera des calculs 32 bits. **ADD W0, W1, W0**
Cet exemple effectue une addition d'entier sur 32 bits.

ADD X0, X1, X0

Cet exemple effectue une addition d'entier sur 64 bits.

Lorsqu'un registre W est écrit, comme dans l'exemple ci-dessus, les 32 premiers bits du registre X 64 bits correspondant sont mis à zéro.

Les registres systèmes

En plus des registres à usage général, l'architecture définit les registres du système. Ces registres sont utilisés pour configurer le processeur et pour contrôler les systèmes tels que le MMU et la gestion des exceptions. Les registres système ne peuvent pas être utilisés directement pour le traitement des données ou les instructions de load/store. Le contenu d'un registre système doit être lu dans un registre X, exploité, puis réécrit dans le registre système. Il existe deux instructions spécialisées pour accéder aux registres systèmes :

MSR Xd, <system register> : Lis le registre système dans Xd.

MRS <system register>, Xn : Ecris Xn dans le registre système.

Les registres systèmes sont spécifiés par leur nom.

Exemple : SCTLR_EL1

MSR X0, SCTLR_EL1 : Lis SCTLR_EL1 dans X0.

Classement des registres

Dans cette section, nous classerons les registres par groupes fonctionnels.

Groupe id

Nom	Description
MPIDR_EL1	Dans un système multiprocesseur, fournit un mécanisme d'identification PE supplémentaire à des fins de planification

Groupe pstate

Nom	Description
SCTLR_EL1	Fournit un contrôle de haut niveau du système, y compris son système de mémoire à EL1 et EL0
SCTLR_EL2	Fournit un contrôle de haut niveau du système, y compris son système de mémoire à EL2
SCTLR_EL3	Fournit un contrôle de haut niveau du système, y compris son système de mémoire à EL3
CurrentEl	Contient le niveau d'exception actuel
SPSel	Permet au pointeur de pile d'être sélectionné entre SP_EL0 et SP_ELx
DAIF	Permet l'accès aux bits du masque d'interruption

Groupe timer

Nom	Description
CNTFRQ_EL0	Rapporte la fréquence du comptage du système.
CNTPCT_EL0	Rapporte la valeur de comptage système actuelle.
CNTP_CTL_EL0	Registre de contrôle du timer physique EL1
CNTP_CVAL_EL0	Contient la valeur de comparaison de 64 bits du timer physique EL1
CNTP_TVAL_EL0	Contient la valeur du timer physique EL1
CNTCV	Indique la valeur de comptage actuelle

Groupe Exception

Nom	Description
SCR_EL3	Définit les configurations de l'état de sécurité actuelle
ESR_EL1	Contient les informations de syndrome pour une exception prise à EL1
ESR_EL2	Contient les informations de syndrome pour une exception prise à EL2
ESR_EL3	Contient les informations de syndrome pour une exception prise à EL3
ISR_EL1	Affiche le statut en attente des interruptions IRQ, FIQ ou SError
VBAR_EL1	Contient l'adresse de base vectorielle pour toute exception prise à EL1
VBAR_EL2	Contient l'adresse de base vectorielle pour toute exception prise à EL2
VBAR_EL3	Contient l'adresse de base vectorielle pour toute exception prise à EL3

Groupe Spécial

Nom	Description
ELR_EL1	Contient l'adresse vers laquelle retourner quand une exception est prise à EL1
ELR_EL2	Contient l'adresse vers laquelle retourner quand une exception est prise à EL2
ELR_EL3	Contient l'adresse vers laquelle retourner quand une exception est prise à EL3
SPSR_EL1	Maintenir l'état du processus enregistré lorsqu'une exception se passe à EL1
SPSR_EL2	Maintenir l'état du processus enregistré lorsqu'une exception se passe à EL2
SPSR_EL3	Maintenir l'état du processus enregistré lorsqu'une exception se passe à EL3
SPSR_abt	Maintenir l'état du processus enregistré lorsqu'une exception se passe en mode abandon
SPSR_fiq	Maintenir l'état du processus enregistré lorsqu'une exception se passe en mode FIQ
SPSR_irq	Maintenir l'état du processus enregistré lorsqu'une exception se passe en mode IRQ
SPSR_und	Maintenir l'état du processus enregistré lorsqu'une exception se passe en mode non défini
SP_EL0	Contient le pointeur de pile associé à EL0
SP_EL1	Contient le pointeur de pile associé à EL1
SP_EL2	Contient le pointeur de pile associé à EL2
SP_EL3	Contient le pointeur de pile associé à EL3

2.5.3 Traitement des données

Le format de base des instructions arithmétiques logiques et entières est :

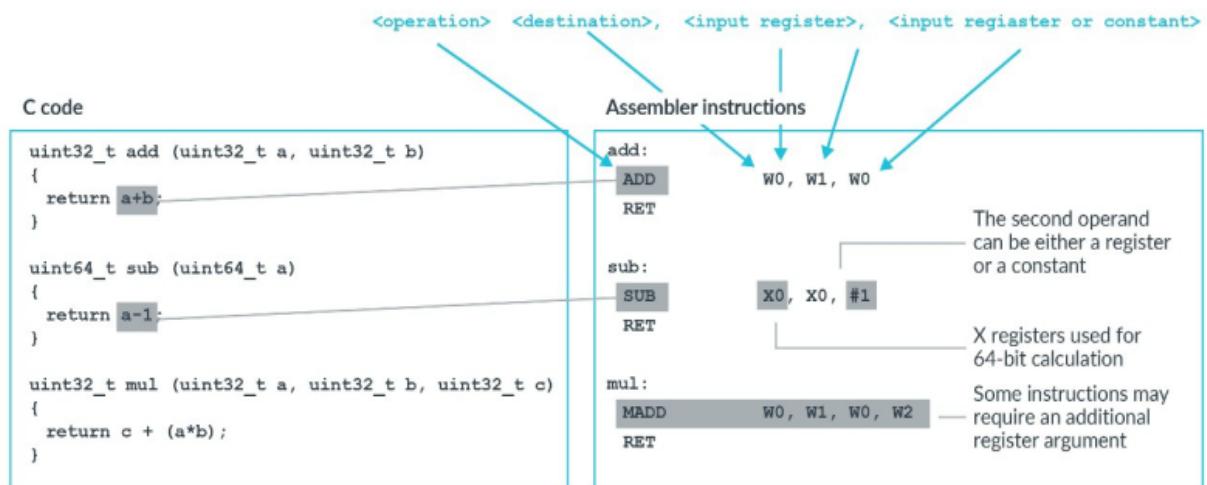


FIGURE 2.5 – Exemple de code C et assembleur armv8.

2.6 Privilèges et niveaux d'exception

Avant d'expliquer les détails du modèle d'exception Armv8-A, commençons par introduire le concept de privilège. Le logiciel moderne s'attend à être divisé en différents modules, chacun avec un niveau d'accès différent aux ressources système et processeur. Un exemple de ceci est la séparation entre le noyau du système d'exploitation, qui a un niveau élevé d'accès aux ressources systèmes, et les applications utilisateurs, qui ont une capacité plus limitée de configurer le système.

Le niveau de privilège le plus bas est appelé EL0. Un modèle d'utilisation courant a un code d'application exécuté à EL0, avec un système d'exploitation exécuté à EL1. EL2 est utilisé par un hyperviseur, avec EL3 étant réservé par un micrologiciel de bas niveau et un code de sécurité.

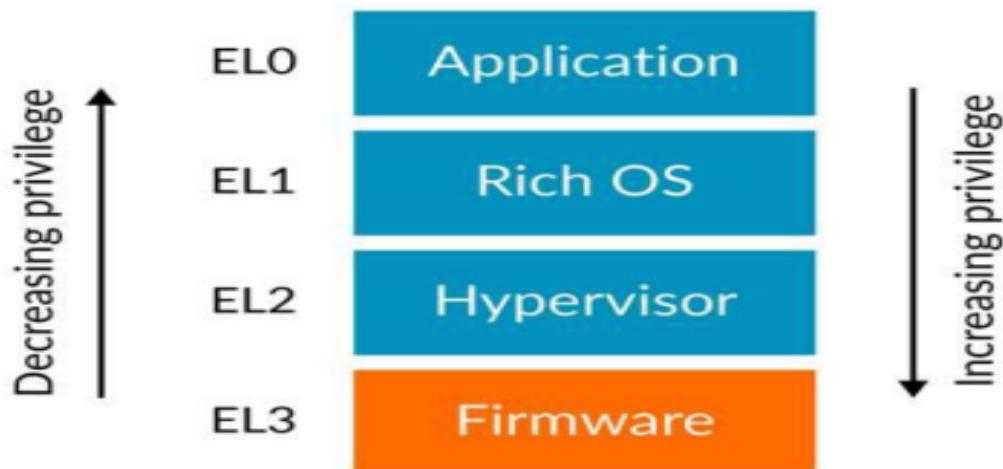


FIGURE 2.6 – Niveau d'exception

2.7 Les types de privilège

Il existe deux types de privilèges pertinents à ce sujet. Le premier est le privilège dans le système de mémoire, et le second est le privilège du point de vue de l'accès aux ressources du processeur. Les deux sont affectés par le niveau d'exception actuel.

2.8 Le privilège dans le système de mémoire

Armv8-A implémente un système de mémoire virtuelle, dans lequel une unité de gestion de mémoire (MMU) permet au logiciel d'attribuer des attributs à régions de la mémoire. Ces attributs incluent des autorisations de lecture/écriture, qui peuvent être configurées avec deux degrés de liberté. Cette configuration permet des autorisations d'accès distincts pour les accès privilégiés et non privilégiés. L'accès à la mémoire initié lorsque le processeur

s'exécute dans EL0 sera vérifié par rapport aux autorisations d'accès non privilégié. Les accès mémoires depuis EL1, EL2 et EL3 seront vérifiés par rapport aux autorisations d'accès privilégiées.

2.9 L'accès aux registres

Les paramètres de configuration des processeurs Armv8-A sont conservés dans une série de registres appelés registres système. La combinaison de paramètres dans les registres système définissent le contexte du processeur courant. L'accès aux registres du système est contrôlé par le niveau d'exception actuel.

Le nom du registre système indique le niveau d'exception le plus bas à partir duquel ce registre est accessible. Par exemple, TTBR0_EL1 est le registre qui contient l'adresse de base de la table de traduction utilisée par EL0 et EL1. Ce registre n'est pas accessible à partir de EL0, et toute tentative de le faire entraînera la génération d'une exception.

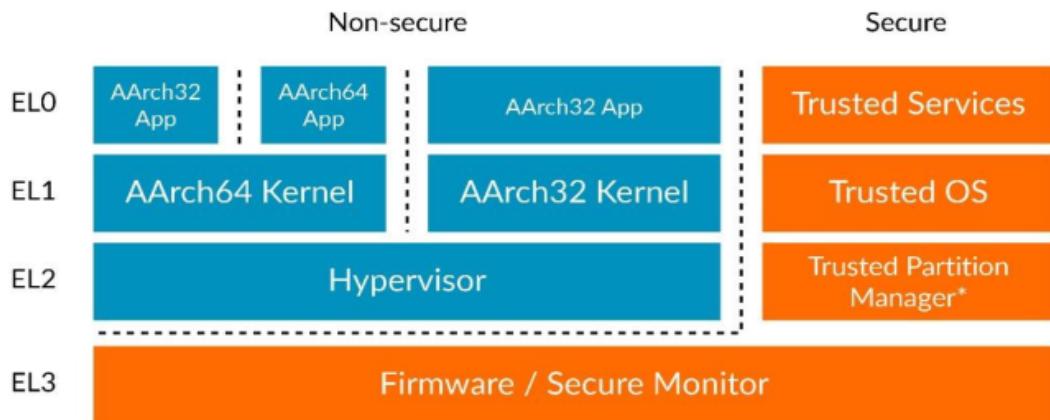


FIGURE 2.7 – Etats d'exécution et niveau d'exécution

2.10 Gestion des exceptions dans l'aarch64 armv8-A

Dans le jargon ARM64, l'exception est définie au sens large ; les interruptions sont un type particulier d'exceptions. L'initialisation des exceptions requiert la mise en place de la table vectorielle, la configuration des masques et le routage des exceptions asynchrones.

2.10.1 Vecteur d'exception

vbar_el1		Size (bytes)	Exception types	Exception taken from CPU states ...
128 (0x80)	Synchronous			
128 (0x80)	IRQ/vIRQ			EL1t (Current EL with SP0)
128 (0x80)	FIQ/vFIQ			
128 (0x80)	SError/vSError			
128 (0x80)	Synchronous			EL1h (Current EL with SPx)
128 (0x80)	IRQ/vIRQ			
128 (0x80)	FIQ/vFIQ			
128 (0x80)	SError/vSError			
128 (0x80)	Synchronous			
128 (0x80)	IRQ/vIRQ			EL0_64 (Lower EL using AArch64)
128 (0x80)	FIQ/vFIQ			
128 (0x80)	SError/vSError			
128 (0x80)	Synchronous			
128 (0x80)	IRQ/vIRQ			EL0_32 (Lower EL using AArch32)
128 (0x80)	FIQ/vFIQ			
128 (0x80)	SError/vSError			

FIGURE 2.8 – Structure de la table vectorielle pour EL1 en mémoire

Un vecteur d'exception (ou gestionnaire) est un morceau de code que le processeur exécutera lorsqu'une exception spécifique se produit. Il s'agirait normalement d'instructions de branchement qui dirigent le noyau vers le gestionnaire d'exceptions complet.

Une table de vecteurs ou table vectorielle est un tableau de vecteurs d'exception. Chaque niveau d'exception (EL) a sa propre table vectorielle. Le noyau définit 16 gestionnaires d'exceptions : 4 types (SError, fiq, irq, sync) avec chacun 4 états d'exécution (EL1t, EL1h, EL0_64, EL0_32) :

Les types d'exceptions :

Exception synchrone : Les exceptions de ce type sont toujours causées par l'instruction en cours d'exécution. Par exemple, vous pouvez utiliser une instruction **str** pour stocker certaines données dans un emplacement mémoire inexistant. Dans ce cas, une exception synchrone est générée. Les exceptions synchrones peuvent également être utilisées pour générer une "interruption logicielle". L'interruption logicielle est une exception synchrone qui est générée volontairement par une instruction. Cette technique est employée pour implémenter les appels systèmes.

Exceptions asynchrones ou (IRQ) : Ce sont des interruptions normales. Elles sont toujours asynchrones, ce qui signifie qu'ils n'ont rien à voir avec l'instruction en cours d'exécution. Contrairement aux exceptions synchrones, elles ne sont toujours pas

générées par le processeur lui-même, mais par du matériel externe.

FIQ (Fast Interrupt Request) : Ce type d'exception est appelé "interruption rapide" et existe uniquement dans le but de hiérarchiser les exceptions. Il est possible de configurer certaines interruptions comme "normales" et d'autres comme "rapides". Les interruptions rapides seront signalées en premier et seront gérées par un gestionnaire d'exceptions distinct. Linux n'utilise pas d'interruptions rapides, nous ne les avons pas aussi implémentés.

SError (System Error) : Comme IRQ et FIQ, les exceptions SError sont asynchrones et sont générées par du matériel externe. Contrairement à IRQ et FIQ, SError indique toujours une condition d'erreur.

2.10.2 Table vectorielle

Il y a des tables vectorielles dédiées pour chaque niveau d'exception.

- VBAR_EL3
- VBAR_EL2
- VBAR_EL1

La table vectorielle de AArch64 est différente de celle de AArch32. La table vectorielle dans le mode AArch64 contient 16 entrées. Chaque entrée a une taille de 128 bytes et contient au plus 32 instructions. Les tables vectorielles doivent être placées à une adresse alignée sur 2 Ko. Les adresses sont précisées en initialisant les registres VBAR_ELn.

La figure suivante montre comment la table vectorielle est structurée.

0x780	SError/vSError	Lower EL using AArch32
0x700	FIQ/vFIQ	
0x680	IRQ/vIRQ	
0x600	Synchronous	
0x580	SError/vSError	Lower EL using AArch64
0x500	FIQ/vFIQ	
0x480	IRQ/vIRQ	
0x400	Synchronous	
0x380	SError/vSError	Current EL with SPx
0x300	FIQ/vFIQ	
0x280	IRQ/vIRQ	
0x200	Synchronous	
0x180	SError/vSError	Current EL with SP0
0x100	FIQ/vFIQ	
0x080	IRQ/vIRQ	
VBAR_ELn + 0x000	Synchronous	

FIGURE 2.9 – Structure de la table vectorielle

2.11 Activation des exceptions asynchrones

Les exceptions asynchrones sont SError, IRQ et FIQ. Ils sont masqués par défaut après réinitialisation. Par conséquent, si SError, IRQ et FIQ doivent être pris, les règles de routage doivent être définies et le masque doit être nettoyé. Pour activer les interruptions, nous devons également initialiser l'interruption globale pour délivrer l'interruption au processeur.

2.12 Routage des exceptions asynchrones

Le routage d'exception asynchrone détermine quel niveau d'exception est utilisé pour gérer une exception asynchrone.

Pour router une exception asynchrone vers EL3, nous devons définir SCR_EL3.EA,IRQ,FIQ. La figure ci-dessous montre comment acheminer SError, IRQ et FIQ vers EL3.

```

MRS    X0, SCR_EL3
ORR    X0, X0, #(1<<3)      // The EA bit.
ORR    X0, X0, #(1<<1)      // The IRQ bit.
ORR    X0, X0, #(1<<2)      // The FIQ bit.
MSR    SCR_EL3, X0

```

FIGURE 2.10 – Routage des exceptions dans EL3

Si une interruption n'est pas acheminée vers EL3 ou EL2, elle est acheminée vers EL1 par défaut.

2.13 Temporisateur générique

Le temporisateur générique (Generic Timer) fournit un cadre de temporisateur standardisé pour les coeurs d'armement. La minuterie générique comprend un compteur système et un ensemble de temporiseurs par cœur, comme illustré sur la figure suivante :

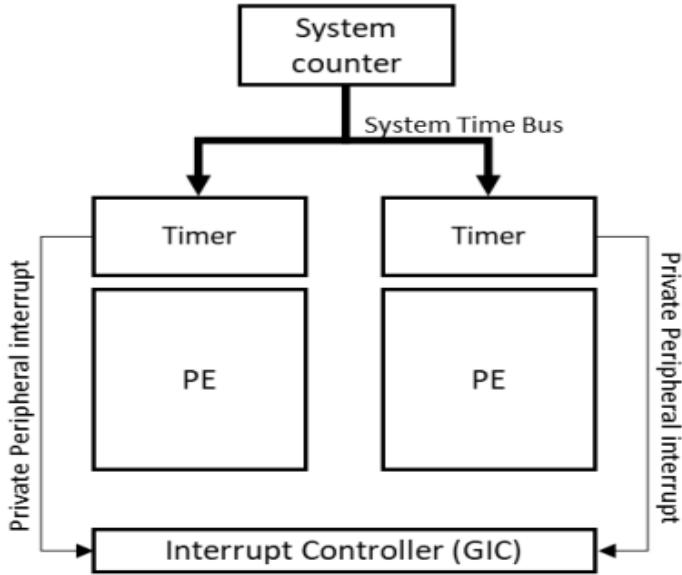


FIGURE 2.11 – Timer générique de l'ARM

Chaque cœur a un ensemble de minuteries. Ces minuteries sont des comparateurs, qui se comparent au compteur générique qui est fourni par le compteur système. L'OS peut configurer des temporiseurs pour générer des interruptions ou des événements dans les points de consigne à l'avenir. L'OS peut utiliser également le compteur générique pour ajouter des horodatages, car il donne un point de référence commun pour tous les coeurs.

2.13.1 Les minuteries du processeur

Le nombre de minuteurs fournis par un cœur dépend des extensions implémentées, comme indiqué dans le tableau suivant :

Nom du Timer	Quand la minuterie est-elle présente ?
Temporisateur physique EL1	Toujours
Temporisateur virtuel EL1	Toujours
Temporisateur physique non-sécurisé EL2	Met en oeuvre EL2
Temporisateur virtuel non-sécurisé EL2	Met en oeuvre ARMv8.1-VHE
Temporisateur physique EL3	Met en oeuvre EL3
Temporisateur physique sécurisé EL2	Met en oeuvre ARMv8.4-SecEL2
Temporisateur virtuel sécurisé EL2	Met en oeuvre ARMv8.4-SecEL2

FIGURE 2.12 – Minuteurs par niveau

2.14 Les registres du temporisateur

Chaque minuterie possède les trois registres système suivants :

Registre	Fonction
<timer>_CTL_EL<x>	Réglage de contrôle
<timer>_CVAL_EL<x>	Valeur du comparateur
<timer>_TVAL_EL<x>	Valeur de la minuterie

FIGURE 2.13 – Régistres de minuterie

Dans le nom du registre, <timer> identifie le timer auquel on accède. Le tableau suivant montre les valeurs possibles :

Timer	Préfixe du registre	EL<x>
Temporisateur physique EL1	CNTP	EL0
Temporisateur virtuel EL1	CNTV	EL0
Temporisateur physique EL2 non-sécurisé	CNTHP	EL2
Temporisateur virtuel EL2 non-sécurisé	CNTHV	EL2
Temporisateur physique EL3	CNTPS	EL1
Temporisateur physique EL2 sécurisé	CNTHPS	EL2
Temporisateur virtuel EL2 sécurisé	CNTHVS	EL2

FIGURE 2.14 – Temporisateur pour chaque niveau d'exception

CNTP_CVAL_EL0 est le registre comparateur du temporisateur physique EL1.

2.14.1 Configuration des temporisateurs

Il existe deux manières de configurer un temporisateur, soit en utilisant le registre du comparateur (CVAL), soit en utilisant le registre du temporisateur (TVAL). Le registre comparateur, CVAL, est un registre de 64 bits. L'OS écrit une valeur dans ce registre et le temporisateur se déclenche lorsque le compteur atteint ou dépasse cette valeur, comme vous pouvez le voir ici :

Condition de minuterie satisfaite : CVAL \leq nombre de ticks systèmes

Le registre temporisateur, TVAL, est un registre de 32 bits. Lorsque l'OS écrit TVAL, le processeur lit le nombre actuel du système en interne, ajoute la valeur écrite, puis remplit CVAL :

$$\text{CVAL} = \text{TVAL} + \text{Compteur système actuel}$$

Condition de minuterie satisfaite : CVAL \leq nombre de ticks systèmes

TVAL et CVAL donnent à l'OS deux modèles différents d'utilisation de la minuterie. Si l'OS a besoin d'un événement de minuterie dans X ticks de l'horloge, le logiciel peut écrire X sur TVAL.

Alternativement, si l'OS veut un événement lorsque le nombre de ticks système atteint Y, l'OS peut écrire Y dans CVAL.

TVAL et CVAL sont des manières différentes de programmer la même minuterie. Ce ne sont pas deux minuteries différentes.

2.15 Le Compilateur croisé

2.15.1 La compilation croisée

La compilation croisée ou «cross-compilation» en anglais permet de créer des exécutables depuis une certaine architecture pour une autre. En règle générale, un compilateur croisé est un compilateur qui s'exécute sur la plateforme A (l'hôte), mais génère des exécutables pour la plateforme B (la cible). Ces deux plates-formes peuvent (mais n'ont pas besoin de le faire) différer en ce qui concerne le processeur, le système d'exploitation et/ou le format exécutable. Dans notre cas, la plate-forme hôte est notre système d'exploitation actuel (Ubuntu/Linux x86_64) et la plate-forme cible est le système d'exploitation que nous sommes sur le point de créer (sur une Architecture ARM 64 bit(Aarch64)). Il est important de réaliser que ces deux plates-formes ne sont pas les mêmes ; le système d'exploitation que nous développons sera toujours différent du système d'exploitation que nous utilisons actuellement. C'est pourquoi nous devons d'abord avoir un compilateur croisé, nous rencontrerons très certainement des problèmes dans le cas contraire.

Nous devons utiliser un compilateur croisé, sauf si nous développons sur notre propre système d'exploitation. Ce qui n'est pas encore possible pour l'instant. Le compilateur doit connaître la plate-forme cible correcte (CPU, système d'exploitation), sinon nous rencontrerons des problèmes. Si nous utilisons le compilateur fourni avec notre système (le GCC natif), le compilateur ne saura pas qu'il compile entièrement autre chose. Certains didacticiels suggèrent d'utiliser notre compilateur système et de passer de nombreuses options problématiques au compilateur. Cela nous donnera certainement beaucoup de problèmes à l'avenir et la solution est d'utiliser carrément un compilateur croisé. Nous avons opté pour la chaîne **aarch64-gnu-linux**.

2.15.2 La chaîne de compilation aarch64-gnu-linux

Pour la génération de l'image de notre OS, nous utilisons la chaîne de compilation aarch64-gnu-linux. Dans cette chaîne nous avons le compilateur : **aarch64-gnu-linux-gcc**, permettant la compilation des fichiers sources en C et des fichiers sources Assembleur. Nous avons aussi le compilateur **aarch64-gnu-linux-as** pour les sources en assembleur mais nous ne l'utilisons pas. Ce compilateur génère des fichiers objets que nous passons à l'outils d'édition de lien **aarch64-gnu-linux-ld** pour l'édition de lien. Nous utilisions enfin l'outils **aarch64-gnu-linux-objcopy** pour la génération de l'image de notre OS.

2.15.3 Les étapes de la compilation

La génération de l'image de notre OS passe bel et bien par des étapes. Qui sont :

Compilation des sources

L'outils que nous utilisons dans cette premier étape est le **aarch64-gnu-linux-gcc**. Ce compilateur transforme les codes sources en C ou en Assembleur en code objet prêt à utiliser dans l'édition de lien.

Les options de compilations :

COPS = -Wall -nostdlib -nostartfiles -ffreestanding -Iinclude -mgeneral-

regs-only

ASMOPS = -Iinclude -g

COPS et **ASMOPS** sont des variables contenant les options que nous passons au compilateur lors de la compilation des sources C et Assembleur respectivement.

-Wall : Afficher tous les avertissements. Une bonne pratique.

-nostdlib : Ne pas utiliser la bibliothèque standard C. La plupart des appels dans la bibliothèque standard C finissent par interagir avec le système d'exploitation. Nous écrivons un programme bare-metal, et nous n'avons pas de système d'exploitation sous-jacent, donc la bibliothèque standard C ne va pas fonctionner pour nous de toute façon.

-nostartfiles : Ne pas utiliser de fichiers de démarrage standard. Les fichiers de démarrage sont chargés de définir un pointeur de pile initial, d'insérer les données statiques et de passer au point d'entrée principal. Nous allons faire tout cela par nous-mêmes.

-ffreestanding : l'environnement freestanding est un environnement dans lequel la bibliothèque standard n'existe peut-être pas et le démarrage du programme n'est pas nécessairement principal. L'option indique au compilateur de ne pas supposer que les fonctions standard ont leur définition habituelle

-Iinclude : Recherchez les fichiers d'en-tête dans le dossier **include**.

-mgeneral-regs-only : Utiliser uniquement des registres à usage général. Les processeurs ARM ont également des **registres NEON**. Nous ne voulons pas que le compilateur les utilise car ils ajoutent une complexité supplémentaire (puisque, par exemple, nous devrons stocker les registres pendant un changement de contexte).

-g : Inclure les informations de débogage dans le binaire ELF résultant.

-O0 : Désactivez toute optimisation du compilateur. Pour faciliter le débogage.

L'édition de lien et génération du binaire ELF et du fichier image

Après compilation des sources, nous devons passer à l'édition de lien avec l'outil **aarch64-gnu-linux-ld** et un script de linkage (**linker.ld**).

Un script de l'éditeur de liens décrit comment les sections des fichiers objet d'entrée (.c et .S) doivent être mappées dans le fichier de sortie (**kernel8.elf**) et contrôle également les adresses de tous les symboles du programme (par exemple, les fonctions et les variables).

Le fichier **kernel8.elf**, le résultat de notre construction, contient tout le code, les données et les informations de débogage. Souvent, pour exécuter un programme ELF dans l'espace utilisateur, il devrait y avoir un chargeur pour l'analyser, charger le code et

les données aux emplacements de mémoire désignés, etc. Pour notre expérience de noyau, nous n'avons pas un tel chargeur pour le noyau lui-même. Il contrôle également les adresses de tous les symboles du programme (par exemple, les fonctions et les variables).

La prochaine étape consiste à faire appel à l'outil **aarch64-gnu-linux-objcopy** et générer le **kernel8.img**, l'image du noyau à partir du binaire **kernel8.elf**. Cette image contient les données et les instructions brutes extraites du binaire **kernel8.elf**. C'est cette image qui est chargée en mémoire.

aarch64-gnu-linux-objcopy peut être utilisé pour générer un fichier image en utilisant une cible binaire (par exemple, avec l'option -O binary). Lors de la génération de notre fichier image, il produira essentiellement un vidding de mémoire du contenu du fichier binaire d'entrée.

CHAPITRE 3

LE BOOTSTRAPING SUR LE PI

3.1 Introduction

Dans ce premier chapitre, nous allons vous présenter comment le Pi charge finalement le noyau de l'OS.

3.2 La carte SD

Tirant ses origines de l'informatique embarquée, la puce BCM2835, qui est au coeur de Raspberry Pi ne possède pas, comme sur les PCs un menu du BIOS (Basic Input/Output System) où l'on peut configurer différents paramètres système de bas niveau. Au lieu de cela, il existe des fichiers textes contenant des chaînes de configurations qui sont chargés par la puce au démarrage du Pi. Au nombre de ces fichiers essentiels, nous avons : config.txt, cmdline.txt, start.elf, fixup.dat, bootcode.bin .

3.2.1 Paramètres du Pi : config.txt

Le matériel du Pi est contrôlé par les paramètres contenus dans un fichier nommé config.txt, qui se trouve dans le répertoire /boot. Ce fichier indique au Pi comment configurer ses différentes entrées et sorties, et à quelle vitesse la puce BCM283x et son module de mémoire doivent tourner.

Le fichier config.txt peut contrôler presque tous les aspects matériels du Pi. Le fichier est lu uniquement au démarrage du système. Toutes les modifications apportées pendant que le Pi est en cours d'exécution n'entreront en vigueur qu'une fois que le système aura redémarré.

Voici quelques paramètres que l'on peut configurer dans ce fichier :

- overscan_left (right, top, bottom) déplace l'image selon la direction, d'un certain nombre de pixel.
- framebuffer_width(height, depth, ignore_alpha) est une valeur exprimée en pixels, qui peut soit changer la largeur de la console, soit affecter la taille de la console,

soit contrôler le nombre de couleurs de la console en bits par pixel.

- `hdmi_mode(drive, force_hotplug, group)` définit les divers paramètres liés à la sortie HDMI.
- `init_uart_baud` : vitesse de la console série en bits par seconde. La valeur par défaut est 115200, mais des valeurs plus faibles peuvent améliorer la connexion si le Pi est utilisé avec un terminal série assez ancien.
- `enable_uart` : par défaut le Raspberry Pi 3 n'a pas de console série active. Pour l'activer, placez un 1 après cette option ; votre Pi 3 bénéficiera ainsi d'une console série, comme les précédents modèles de Pi.

3.2.2 Paramètres de l'OS : cmdline.txt

Ce fichier `cmdline.txt` contient ce qu'on appelle la ligne de commande du noyau, qui représente les options passées au noyau de l'OS au démarrage du Pi. Dans un ordinateur classique tournant sur Linux, ces options sont généralement transmises au noyau par le biais d'un outil appelé bootloader, qui possède son propre fichier de configuration. Sur le Pi, les options sont simplement saisies directement dans le fichier `cmdline.txt` pour être lues par le Pi au démarrage. Presque n'importe quelle option du noyau prise en charge par Linux peut être entrée dans le fichier `cmdline.txt` pour modifier des choses qui vont de l'apparence de la console jusqu'au chargement du système de fichiers racine. À titre d'exemple, voici le fichier `cmdline.txt` de la distribution Raspbian, qui doit être écrit sous la forme d'une seule ligne continue :

```
dwc_otg.lpm_enable=0 console=serial0,115200 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 elevator=deadline fsck.repair=yes rootwait
```

L'option `root` indique au noyau Linux où il peut trouver le système de fichiers racine, qui contient tous les fichiers et les répertoires nécessaires pour le fonctionnement du système. Dans le cas de la distribution Raspbian par défaut, lors de l'installation à partir de l'image officielle, ce système de fichiers racine se trouve sur la deuxième partition de la carte SD, dans le répertoire `mmcblk0p2`.

Le paramètre `rootwait` indique au noyau qu'il ne devrait pas essayer de démarrer le système tant que le dispositif contenant le système de fichiers racine n'est pas disponible. Sans cette option, le Pi peut se bloquer alors qu'il commence à démarrer si une carte SD relativement lente n'est pas encore prête.

3.2.3 Le fichier start.elf

C'est le firmware Pi.

3.2.4 Le fichier fixup.dat

Ce fichier de données contient des informations importantes relatives au matériel.

3.2.5 Le fichier bootcode.bin

C'est lui qui est chargé en premier, exécuté sur le GPU(pas nécessairement sur pi 4 car ce modèle a bootcode.bin dans une ROM).

3.3 Le bootstrapping sur le Pi

Lorsque le Pi s'allume, le processeur ARM est arrêté et le processeur graphique fonctionne.

Seul le GPU est activé, l'unité de traitement étant à l'arrêt. La SDRAM est elle aussi désactivée. Le socle contient une ROM dans laquelle le fabricant a enregistré un programme faisant partie de la chaîne de boot.

Ce programme est le premier de la chaîne de boot Il est inaccessible par l'utilisateur et n'est donc pas modifiable. GPU exécute ce premier programme dont le seul rôle est d'accéder à la partition FAT de la Card SD pour charger le fichier bootcode.bin.

Le fichier bootcode.bin est chargé en mémoire cache L2. La SDRAM n'est toujours pas activée. C'est la mémoire cache L2 du GPU qui est utilisé pour le chargement.

Le GPU exécute le fichier bootcode.bin. Le but est de récupérer le programme start.elf, qui se situe sur la carte SD. Le GPU, sous les ordres de bootcode.bin, active la SDRAM et transfère une copie de start.elf. Une fois start.elf chargé en mémoire, bootcode.bin lui passe le relais.

Le GPU exécute maintenant son firmware. "start.elf" répartit la mémoire entre le GPU et le CPU ARM, en fonction des paramètres de configuration dans le fichier config.txt.

Après mise en place des zones mémoires, le programme start.elf transfère l'image du noyau (kernel.img) dans la partie basse de la mémoire, zone réservée au CPU ARM. Il lit ensuite le fichier cmdline.txt qui contient les arguments passés au noyau lors de son exécution.

Le programme start.elf va encore réveiller le CPU ARM et laissé la main à ce dernier.

Le processeur ARM démarre et exécute l'image du noyau à une adresse bien spécifique.

CHAPITRE 4

ARCHITECTURE ET IMPLÉMENTATION DE L'OS

4.1 Modélisation

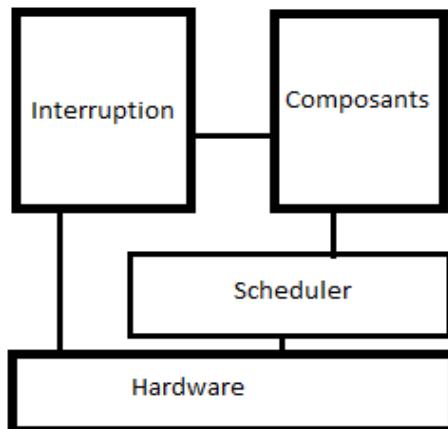


FIGURE 4.1 – Schéma global de l'architecture de l'os implémenté

4.2 Hardware

Dans cette partie de l'OS, nous implémentons les fonctionnalités essentielles d'un OS, à savoir :

- le contrôleur d'interruption : nous configurons les périphériques concerné afin de router les interruptions au niveau d'exception appropriés.
- l'ordonnanceur : il s'agit de la partie du noyau qui planifie les tâches

4.2.1 Le contrôleur d'interruption

Vecteurs d'exception, tables, etc. (entry.S)

La figure ci-dessous montre comment la table vectorielle est définie.

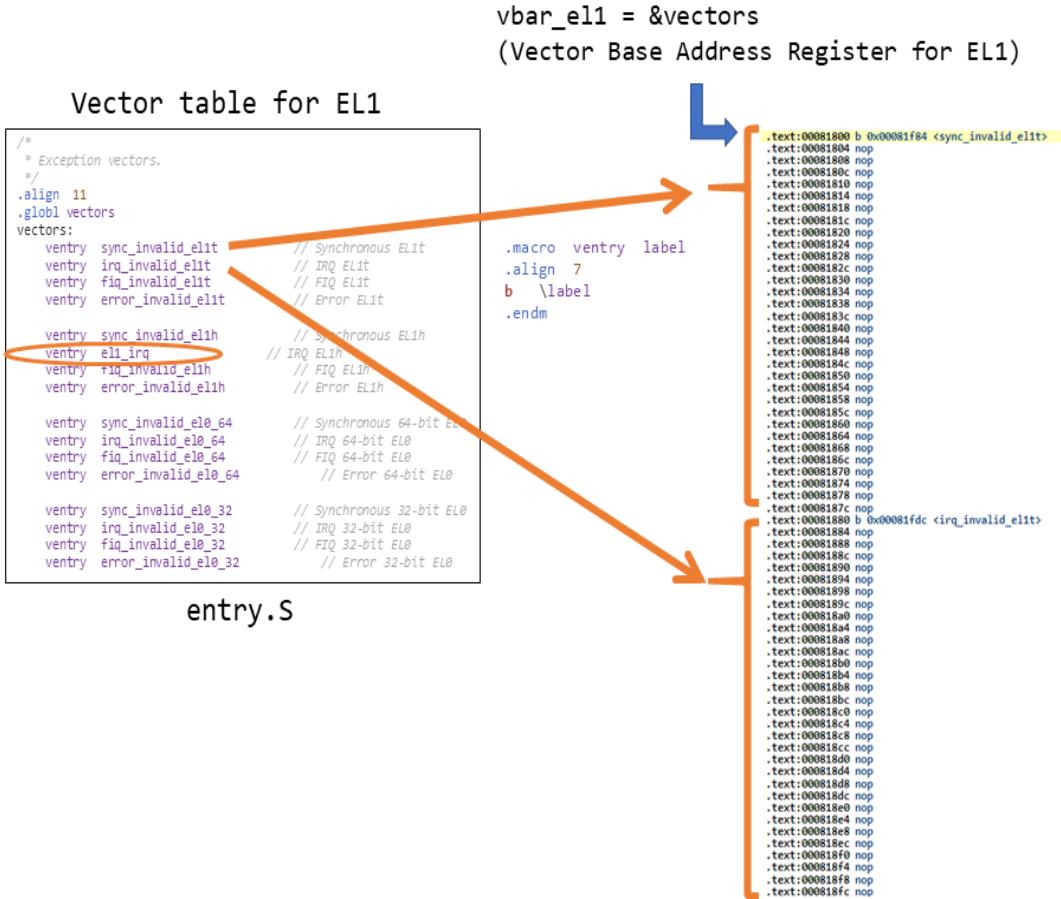


FIGURE 4.2 – Table vectorielle

La table vectorielle se compose de 16 définitions de ventry :

```

1   .align 11
2   .globl vectors
3   vectors:
4       ventry sync_invalid_el1t      // Synchronous EL1t
5       ventry irq_invalid_el1t       // IRQ EL1t
6       ventry fiq_invalid_el1t       // FIQ EL1t
7       ventry error_invalid_el1t     // Error EL1t

```

La macro ventry est utilisée pour créer des entrées dans la table vectorielle.

```

1   .macro ventry label
2   .align 7
3   b \label
4   .endm

```

Nous ne gérons les exceptions directement à l'intérieur du vecteur d'exception. Nous faisons de chaque vecteur une instruction de branchemen (b qui saute à une étiquette fournie pour la macro en tant qu'argument label.

Nous en avons besoin .align 7 car tous les vecteurs d'exception doivent être espacés d' 0x80 octets (2×7) les uns par rapport aux autres.

Faire prendre conscience au CPU de la table vectorielle (irq.S)

Nous avons préparé la table vectorielle, mais le processeur ne sait pas où elle se trouve et ne peut donc pas l'utiliser. Pour que la gestion des exceptions fonctionne, nous devons définir vbar_el1(Vector Base Address Register) sur l'adresse de la table vectorielle.

```

1      .globl irq_vector_init
2      irq_vector_init:
3          adr    x0, vectors
4          msr    vbar_el1, x0
5          ret

```

Un gestionnaire simple pour les exceptions inattendues

Dans cette expérience, nous nous intéressons uniquement à la gestion des IRQ partir de EL1h. Or notre noyau définit les 16 gestionnaires pour EL1. C'est pour faciliter la compréhension : nous voulons imprimer un message significatif au cas où notre noyau déclencherait d'autres exceptions en raison de nos erreurs de programmation.

Nous nommons tous les gestionnaires qui ne sont censés être déclenchés avec un suffixe invalid. Nous implémentons ces gestionnaires à l'aide d'une macro handle_invalid_entry :

```

1      .macro handle_invalid_entry type
2      kernel_entry
3      mov    x0, #\type
4      mrs    x1, esr_el1
5      mrs    x2, elr_el1
6      b1    show_invalid_entry_message
7      b     err_hang
8      .endm

```

La première ligne invoque une macro kernel_entry qui correspond aux premières instructions que le noyau doit exécuter pour gérer une exception/interruption.

Ensuite, nous appelons show_invalid_entry_message() et préparons 3 arguments pour cela. Les arguments sont passés dans 3 registres : x0, x1 et x2.

x0 : le type d'exception. La valeur provient de l'argument de la macro. Il peut prendre l'une de ces valeurs définies par notre code noyau. Il nous indique exactement quel gestionnaire d'exceptions a été exécuté.

x1 : informations sur les causes de l'exception. La valeur vient du registre esr_el1. Re-

marque : dans cette expérience, notre noyau s'exécute à EL1 et lorsqu'une interruption se produit, elle est gérée à EL1.

x2 : l'adresse de l'instruction en cours d'exécution lorsque l'exception se produit. La valeur provient de elr_el1. Pour les exceptions synchrones, c'est l'instruction qui provoque l'exception ; pour irqs (asynchrone), il s'agit de l'instruction terminée juste avant que irq ne se produise. Ce registre détient l'adresse à laquelle retourner.

Le code appelle ensuite la show_invalid_entry_message fonction, qui imprime des informations textuelles à UART. En revenant de cette fonction, le code s'exécute dans une boucle infinie car nous n'avons rien d'autre à faire.

gestion des exceptions et sortie

Pour gérer les exceptions valides (interruptions du timer dans notre cas), le noyau doit sauvegarder et restaurer le contexte de l'exécution "normale", c'est-à-dire passer de l'exécution normale au gestionnaire d'exceptions, l'exécuter et reprendre l'exécution interrompue. En d'autres termes, après le gestionnaire d'exceptions, nous voulons que tous les registres à usage général aient les mêmes valeurs qu'avant la génération de l'exception.

```

1      el1_irq:
2          kernel_entry
3          bl  handle_irq
4          kernel_exit

```

Retour à kernel_entry. C'est la première chose à faire dans la gestion d'une exception : sauvegarder l'état du processeur, notamment les registres x0 - x30, dans la pile. Pour ce faire, il soustrait d'abord à sp la taille du total des registres stockés (#S_FRAME_SIZE), puis remplit l'espace de la pile. Ensuite nous appelons la fonction handle_irq.

kernel_exit doit être appelé comme la dernière chose dans un gestionnaire d'exceptions. kernel_exit restaure l'état du CPU en copiant les valeurs de x0 - x30. L'ordre reflète exactement celui de kernel_entry sinon nous verrons des valeurs de registre erronées. kernel_exit Exécute enfin eret, qui revient à l'exécution normale.

```

1      .macro  kernel_entry
2      sub  sp , sp , #S_FRAME_SIZE
3      stp  x0 , x1 , [sp , #16 * 0]
4      stp  x2 , x3 , [sp , #16 * 1]
5      stp  x4 , x5 , [sp , #16 * 2]
6      stp  x6 , x7 , [sp , #16 * 3]
7      stp  x8 , x9 , [sp , #16 * 4]
8      stp  x10 , x11 , [sp , #16 * 5]
9      stp  x12 , x13 , [sp , #16 * 6]
10     stp  x14 , x15 , [sp , #16 * 7]
11     stp  x16 , x17 , [sp , #16 * 8]
12     stp  x18 , x19 , [sp , #16 * 9]
13     stp  x20 , x21 , [sp , #16 * 10]
14     stp  x22 , x23 , [sp , #16 * 11]
15     stp  x24 , x25 , [sp , #16 * 12]
16     stp  x26 , x27 , [sp , #16 * 13]
17     stp  x28 , x29 , [sp , #16 * 14]

```

```

18     str x30, [sp, #16 * 15]
19     .endm
20
21     .macro kernel_exit
22     ldp x0, x1, [sp, #16 * 0]
23     ldp x2, x3, [sp, #16 * 1]
24     ldp x4, x5, [sp, #16 * 2]
25     ldp x6, x7, [sp, #16 * 3]
26     ldp x8, x9, [sp, #16 * 4]
27     ldp x10, x11, [sp, #16 * 5]
28     ldp x12, x13, [sp, #16 * 6]
29     ldp x14, x15, [sp, #16 * 7]
30     ldp x16, x17, [sp, #16 * 8]
31     ldp x18, x19, [sp, #16 * 9]
32     ldp x20, x21, [sp, #16 * 10]
33     ldp x22, x23, [sp, #16 * 11]
34     ldp x24, x25, [sp, #16 * 12]
35     ldp x26, x27, [sp, #16 * 13]
36     ldp x28, x29, [sp, #16 * 14]
37     ldr x30, [sp, #16 * 15]
38     add sp, sp, #S_FRAME_SIZE
39     eret
40     .endm

```

Configuration du contrôleur d'interruption

Nous ne sommes intéressés que par les interruptions de minuterie. Le manuel du SoC, page 113 indique que les irq #1 et #3 proviennent de la minuterie système. Ces sources irq appartiennent au groupe irq 1, qui peut être activé à l'aide de ENABLE_IRQS_1 . enable_interrupt_controller() Active donc l' IRQ de la minuterie système au numero 1 :

```

1 void enable_interrupt_controller()
2 {
3     put32(ENABLE_IRQS_1, SYSTEM_TIMER_IRQ_1);
4 }

```

Masquage/démasquage des interruptions

De temps en temps, le noyau doit masquer/démasquer TOUTES les interruptions, afin que certaines régions de code critiques ne soient jamais interrompues. Par exemple, que se passe-t-il si une interruption se produit en plein milieu de la kernel_entry macro ? L'état du processeur serait corrompu.

Les deux fonctions suivantes (irq.S) masquent et démasquent les interruptions.

```

1 .globl enable_irq
2 enable_irq:
3     msr daifclr, #2
4     ret
5
6 .globl disable_irq
7 disable_irq:
8     msr daifset, #2
9     ret

```

Le gestionnaire d'IRQ

Nous avons un seul gestionnaire d'exceptions commun pour gérer tous les IRQs. Ce gestionnaire est défini ici :

```

1 void handle_irq(void)
2 {
3     unsigned int irq = get32(IRQ_PENDING_1);
4     switch (irq) {
5         case (SYSTEM_TIMER_IRQ_1):
6             handle_timer_irq();
7             break;
8         default:
9             printf("Unknown pending irq: %x\r\n", irq);
10        }
11 }
```

4.2.2 Le Scheduler

Ici, nous implémentons un ordonnancement où les tâches se succèdent selon l'ordre d'arrivé. Notre OS est mono-tâche, et nous avons opté pour l'ordonnancement avec l'algorithme FIFO.

4.3 Les composants

Nous avons opté pour un développement à base de composants. Les composant ou modules sont les périphériques configurés et disponibles pour construire une application.

- gpio : pour interagir avec les 40 broches et ainsi leur attribuer des fonctions.
- mini_uart : pour interagir avec la console série à travers les broches GPIOs.
- led : ce composant nous sert à allumer des leds.
- timer : il s'agit d'un composant qui permet de définir de manipuler le temps et est utilisé par d'autres composants(led)

4.3.1 GPIO

Le mappage d'adresse

Pour configurer ce module, Dans le fichier d'entête du périphérique GPIO, nous avons commencer à créer des macros au nom des registres du GPIO et nous les avons mappés avec leur adresse telle fournie dans le manuel du processeur après l'ajout de l'adresse de base du périphérique. Nous en avons fait de même pour les pins.

```

1 //GPIO address maping
2
3 #define GPIO_BASE (PERIPHERAL_BASE + 0x200000)
4
5 #define GPFSEL0 (PERIPHERAL_ABSE + 0x200000)
6 #define GPFSEL1 (PERIPHERAL_BASE + 0x200004)
7 #define GPFSEL2 (PERIPHERAL_BASE + 0x200008)
8 #define GPFSEL3 (PERIPHERAL_BASE + 0x20000C)
9 #define GPFSEL4 (PERIPHERAL_BASE + 0x200010)
10 #define GPFSEL5 (PERIPHERAL_BASE + 0x200014)
11
12 #define GPSET0 (PERIPHERAL_BASE + 0x20001C)
13 #define GPSET1 (PERIPHERAL_BASE + 0x200020)
14
15 ...
16
17 //GPIO pin mapping
18 #define GPIO0 0
19 #define GPIO1 1
20 #define GPIO2 2
21 #define GPIO3 3
22 #define GPIO4 4
23 #define GPIO5 5
24 #define GPIO6 6
25 #define GPIO7 7
26 #define GPIO8 8
27 #define GPIO9 9

```

Notons que la configuration de tous nos pilote commencera toujours par cette étape.

```

1
2 unsigned int gpio_call(unsigned int pin, unsigned int value, unsigned int
base, unsigned int field_size, unsigned int field_max)
3 {
4     unsigned int field_mask = (1 << field_size) - 1;
5
6     if (pin > field_max)
7         return 0;
8
9     if (value > field_mask)
10        return 0;
11
12     unsigned int num_fields = 32 / field_size;
13     unsigned int reg = base + ((pin / num_fields) * 4);
14     unsigned int shift = (pin % num_fields) * field_size;
15
16     unsigned int curval = get32(reg);
17     curval &= ~(field_mask << shift);
18     curval |= value << shift;
19     put32(reg, curval);
20
21     return 1;
22 }

```

Cette fonction prend en argument le pin à configurer, la valeur pour la configuration, l'adresse de base pour la configuration, le nombre de bit nécessaire pour la configuration permettant de définir le mask, le nombre de pin maximal.

Après vérification de la cohérence des valeurs entrées, on accède au registre dans lequel il nous faut écrire et on écrit à l'intérieur la valeur de configuration après avoir atteint les champs du registre dans lesquels il faut écrire.

reg contient l'adresse du registre dans lequel il faut écrire. Il est obtenu à partir de l'adresse de base du type de registre auquel il faut accéder plus un décalage approprié.
shift contient le nombre de décalage qu'il faut pour atteindre le champ du registre dans lequel il faut écrire en fonction du pin.

Nous avons définie ensuite d'autres fonctions qui s'appuie sur celle précédente

Les fonctions de configuration

La fonction de base, sur laquelle repose toute les opérations de configuration du GPIO est la fonction **gpio_call**. Nous avons les fonctions suivantes :

- La fonction **gpio_configure** permet d'attribuer une fonction à un pin ou définie un pin comme une broche d'entrée ou de sortie.
- Les fonctions **gpio_set** et **gpio_clear** permet d'activer ou désactiver un pin configuré comme sortie.

Les autres fonctions sont présentes dans le fichier source du module GPIO.

4.3.2 Mini UART

Le mini UART est un périphérique auxiliaire du Pi. Pour l'utiliser il faut d'abord l'activer et régler la fréquence de l'horloge du processeur. Après le mappage des registres, nous avons fournit une fonction qui initialise l'UART et d'autres fonctions qui permettent de transmettre ou de recevoir un caractère depuis les ports séries.

```

1 void uart_init ( void )
2 {
3     //configure alternative function 5 on GPIO14 and GPIO15 pins.
4     gpio_clear(14, 1);
5     gpio_pin_configure(14, 2);
6
7     gpio_clear(15, 1);
8     gpio_pin_configure(15, 2);
9
10    //configure pull none on GPIO14 and 15 pins
11    put32(GPPUD,0);
12    delay(150);
13    put32(GPPUDCLK0,(1<<14)|(1<<15));
14    delay(150);
15    put32(GPPUDCLK0,0);

```

```

16 // set up UART registers
17 put32(AUX_ENABLES,1); //Enable mini uart (this also
18 enables access to its registers)
19 put32(AUX_MU_CNTL_REG,0); //Disable auto flow control and
20 disable receiver and transmitter (for now)
21 put32(AUX_MU_IER_REG,0); //Disable receive and transmit
22 interrupts
23 put32(AUX_MU_LCR_REG,3); //Enable 8 bit mode
24 put32(AUX_MU_MCR_REG,0); //Set RTS line to be always
25 high
26 put32(AUX_MU_BAUD_REG,270); //Set baud rate to 115200
27 put32(AUX_MU_CNTL_REG,3); //Finally, enable transmitter
28 and receiver
29 }
```

Les fonctions suivantes permettent respectivement la transmission d'un caractère, la réception d'un caractère et la transmission d'une chaîne de caractères.

```

1 void uart_send ( char c )
2 {
3     while(1) {
4         if( get32(AUX_MU_LSR_REG)&0x20)
5             break;
6     }
7     put32(AUX_MU_IO_REG, c );
8 }
9
10 char uart_recv ( void )
11 {
12     while(1) {
13         if( get32(AUX_MU_LSR_REG)&0x01)
14             break;
15     }
16     return( get32(AUX_MU_IO_REG)&0xFF );
17 }
18
19 void uart_send_string(char* str)
20 {
21     for (int i = 0; str[i] != '\0'; i++) {
22         uart_send((char)str[i]);
23     }
24 }
```

Les fonctions **put32** et **get32** permettent respectivement d'écrire une valeur dans un registre de 32 bits et de lire une valeur dans un registre de 32 bits.

```

1 .globl put32
2 put32:
3     str w1,[x0]
4     ret
5
6 .globl get32
```

```

7      get32 :
8          ldr w0, [x0]
9          ret

```

4.3.3 Timer

Nous utilisons le timer générique Arm, qui fait partie de la conception du noyau Arm64. C'est bien, car les timers génériques existent pour tous les processeurs Armv8. Ce composant nous permet de générer à une interruption qui après un temps t donné. L'idée est par exemple de l'utiliser pour allumer des leds à des intervalles de temps donnés. Ce périphérique offre deux registres : **TVAL** et **CVAL** qui permettent de le configurer. Le matériel contient un compteur général appelé **System Counter**, qui au démarrage du matériel commence à s'incrémenter. Notre timer se base sur ce System Counter pour fonctionner.

Nous commençons à initialiser le timer lui permettons de générer des interruptions.

```

1      gen_timer_init:
2          mov x0, #1
3          msr CNTP_CTL_EL0, x0
4          ret

```

Ceci écrit 1 dans le registre de contrôle (CNTP_CTL_EL0) du temporisateur physique EL1.

Activer l'interruption de la minuterie au cœur du processeur

Les IRQ du temporisateur générique Arm sont câblées à un contrôleur/registre d'interruption par cœur. Pour le noyau 0, il s'agit de TIMER_INT_CTRL_0 situé à l'adresse 0x40000040 ;

```

1      void enable_interrupt_controller()
2  {
3      // Enables Core 0 Timers interrupt control for the generic timer
4      put32(TIMER_INT_CTRL_0, TIMER_INT_CTRL_0_VALUE);
5  }

```

Traitements des interruptions de la minuterie

Le noyau obtient un irq. Le noyau vérifie si cela vient du timer ; si c'est le cas, le noyau définit le temporisateur pour déclencher la prochaine interruption.

```

1      void handle_irq(void) {
2          // Each Core has its own pending local interrupts register
3          unsigned int irq = get32(INT_SOURCE_0);
4          switch (irq) {
5              case (GENERIC_TIMER_INTERRUPT):

```

```

6           handle_generic_timer_irq() ;
7           break ;
8           ...

```

Le gestionnaire d'exceptions EL1h appelle la fonction ci-dessus. La fonction lit INT_SOURCE_0, où le bit 1 est pour notre minuteur CNTP.

Réinitialiser la minuterie (timer.S)

Le noyau écrit une valeur delta ($1 \ll 24$) dans TVAL, demandant une interruption pour se déclencher après $1 \ll 24$ ticks.

```

1      gen_timer_reset :
2          mov x0, #1
3          ls1 x0, x0, #24
4          msr CNTP_TVAL_EL0, x0
5          ret

```

CHAPITRE 5

CONCLUSION

Au cours de notre travail, nous avons eu à découvrir certains périphériques du Raspberry ainsi que l'architecture du processeur avec certaines de ses fonctionnalités afin d'offrir une documentation les expliquant ainsi qu'une configuration de ces derniers. Nous avons eu à travailler sur les périphériques GPIO, le mini UART, le timer générique de l'ARM, les interruptions. Cette expérience nous a offert une expérience qui nous initie dans le monde des logiciels pour embarqués et celui de la programmation des processeur ARM.

CHAPITRE 6

BIBLIOGRAPHIE

- Zhao Jiong : A Heavily Commented Linux Kernel Source Code Kernel Version 0.12
- ARM Cortex-A Series Version : 1.0 Programmer’s Guide for ARMv8-A
- Christophe Blaess : Programmation système en C sous Linux
- ARM :Armv8-A Instruction Set Architecture
- ARM :Exception model
- Application Note Bare-metal Boot Code for ARMv8-A Processors Version 1.0
- QA7 ARM Quad A7 core Gert van Loo, 18 August 2014
- BCM2837 ARM Peripherals
- Construction d’un système d’exploitation pour le Raspberry Pi_BarePar Jake Sandler
- Arm Cortex-A53 MPCore Processor Revision : r0p4 Technical Reference Manual
- Raspberry Pi platform without an operating system Petr Vanc
- ARM :AArch64 Programmer’s Guides Generic Timer
- Operating System Concepts,8th Edition[A4]
- TinyOS Programming Philip Levis October 27, 2006
- Andrew S. Tanenbaum, Albert S. Woodhull - Operating systems design and implementation- Prentice Hall (1997)

- operating-systems-design-implementation-3rd-edition-1-
- Raspberry Pi 3 - Guide de l'utilisateur - Dunod

CHAPITRE 7

WEBOGRAPHIE

- https://wiki.osdev.org/Raspberry_Pi_Bare_Bones
- <https://jsandler18.github.io/>
- <https://www.valvers.com/open-software/raspberry-pi/bare-metal-programming-in-c-part-1>
- <https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/>
- <https://isometimes.github.io/rpi4-osdev/>
- <https://fxlin.github.io/p1-kernel/>
- <https://gcc.gnu.org/onlinedocs/gcc/ARM-Function-Attributes.html#ARM-Function-Attributes>
- <https://github.com/bztsrc/raspi3-tutorial>
- <https://alexandre-laurent.developpez.com/articles/hardware/raspberry-pi/installation-système/#LIII-A-1>
- <https://www.linuxfromscratch.org/lfs/>
- <https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm/downloads>
- <https://github.com/tywkeene/doom-kernel/tree/main/kernel>
- <https://github.com/metebalci/baremetal-rpi>
- <https://github.com/mstachowsky/tinykernel>

- <https://fr.wikipedia.org/wiki/TinyOS>
- <https://github.com/tinyos/tinyos-main>