connect.ed-diamond.com

La liberté jusqu’au cœur du processeur avec RISC-V | Connect - Edition Diamond

Auteurs Marteau Fabien

39-50 minutes

Quand on pense « architecture de processeurs », l'architecture x86 de Intel nous vient naturellement en tête, comme nous avons pu le voir dans l'article de GNU/Linux Magazine en septembre 2018 **[1]**. Il est naturel pour un informaticien de penser d’abord à cette architecture, puisque c'est celle massivement utilisée sur les ordinateurs personnels. Pourtant, il existe de nombreuses autres architectures de microprocesseurs, allant du tout petit microprocesseur 4 bits enfoui au microprocesseur 64 bits à instructions vectorisées, pour les supercalculateurs massivement parallèles.

Il existe également toute une série de processeurs open source souvent écrite en VHDL/Verilog, à destination des FPGA. Ces processeurs sont plus souvent des microcontrôleurs et généralement, des dérivés de «*vieux*» microprocesseurs.

RISC-V n'est pas un énième processeur libre, qui s'ajoute à la pile de processeurs «*softcore*» à utiliser sur un FPGA, comme l'on peut en trouver sur *opencore* et autre *librecore*.

RISC-V (se prononce «*risque faïlleve*») est une standardisation libre d'un jeu d'instructions, le langage d'un processeur, avec une série d’extensions permettant de couvrir l'ensemble des domaines d'application des architectures 32/64 bits (voir même 128 bits, même si la définition n'est pas encore stabilisée, le temps d'en trouver une utilité). Pas de micro en 4, 8 ou 16 bits donc. Mais, même pour la très basse consommation, le 32 bits est désormais très utilisé, comme on peut le voir avec les STM32. De plus, il existe une extension permettant d’utiliser des instructions compressées sur 16 bits, ainsi qu’une base avec moitié moins de registres (RV32E).

1. Le RISC, une architecture load/store

La formalisation de l’architecture RISC est née à Berkeley au début des années 80, à une époque où les principaux fondeurs de microprocesseurs menaient une course à qui aura le plus d'instructions, les plus grosses, les plus complexes... À cette époque, les constructeurs de processeurs ajoutaient des registres et des instructions de tailles et de formats différents, pour augmenter la puissance de calcul de leurs machines.

Cet ajout de multiples registres et instructions avait pour conséquence une augmentation du nombre de transistors, ainsi qu'une augmentation de la consommation. Cette course à l'échalote fonctionnait plutôt bien, en tenant compte de la loi de Moore. Et l'architecture x86 d'Intel est la preuve que cela fonctionne encore, puisque le format x86 domine toujours le marché des ordinateurs personnels et des serveurs.

Cependant, à Berkeley, les chercheurs tentèrent une échappée et proposèrent un nouveau modèle de microprocesseur, basé sur la simplicité. L'idée était qu'on a tout à gagner avec un jeu d'instructions plus simple, plus petit. En faisant des statistiques sur les instructions généralement utilisées par les compilateurs, ils s'étaient aperçus que la plupart de ces grosses instructions ne sont utilisées qu'à la marge, et que seule une portion minimale était massivement utilisée dans les programmes «*génériques*». Que de silicium gaspillé pour implémenter des instructions quasiment

jamais utilisées, que d'énergie gaspillée pour cadencer le décodage des instructions devenues très complexes, que de cycles d'horloges gaspillés pour cadencer le cycle d'exécution de chaque instruction...

Dans leur publication initiale, les chercheurs ont donc défini le concept de **RISC** pour *Reduced Instruction Set Computer*, en opposition à **CISC** pour *Complex Instruction Set Computer*. À Stanford, une autre équipe a elle aussi surfé sur cette vague, en lançant un jeu d'instructions qui a son petit succès encore aujourd'hui : le **MIPS**. Jeu d'instructions qui vient d'ailleurs d'être libéré, en réaction à la montée de RISC-V...

On associe très souvent le «Reduced» de RISC à un nombre réduit d'instructions. Même si les ISA de base ont souvent moins d'instructions que leurs «*équivalents*» CISC, ça n'est pas tout à fait vrai. En effet, les ISA RISC aujourd'hui ont largement plus d'instructions que les CISC du début des années 80, par exemple (le 8086 avait 80 instructions). Non, la signification du «Reduced» est surtout vis-à-vis des fonctionnalités, une instruction RISC ne doit faire qu'une seule chose, rapidement. Et notamment, la lecture et l'écriture dans la mémoire sont exécutées par des instructions séparées (*load/store*), les instructions arithmétiques faisant leurs calculs sur les registres internes. C'est pourquoi on parle également d'architecture *load/store*, plutôt que de RISC.

On pourrait donc résumer la définition de l'architecture RISC comme ceci :

- Des instructions simples qui n'exécutent qu'une seule tâche rapidement.
- Un format d'instructions de largeur fixe (32 bits dans la majorité des cas).
- Des registres internes «*génériques*».

Hormis l'architecture de type x86, la plupart des nouveaux processeurs essaient de se rapprocher de la philosophie RISC, aujourd'hui.

2. Qu'est-ce qu'un processeur RISC-V ?

Un processeur RISC-V est une implémentation du jeu d'instructions (*ISA*). En effet, le standard RISC-V ne définit que le «*langage*» du processeur. L'implémentation est à la charge du concepteur de processeur. C'est à lui de décider en combien de cycles d'horloge s'exécutera une instruction, la taille du pipeline qu'il veut implémenter, comment gérer la prédiction de branches ou le format du bus de donnée (AXI, Wishbone…).

Le nom RISC-V fait référence à la cinquième version de l'ISA RISC développée par Berkeley. Les quatre précédentes se prénomment RISC-I, RISC-II, SOAR et SPUR.

Mais le V en chiffre romain fait également référence à «*Vector*», car le but de cette ISA est de pouvoir faire du calcul parallèle au niveau données (vectorielles). C'est l'objet des extensions V et P du standard RISC-V.

Ainsi, un même jeu d'instructions pourra être implémenté sur un petit processeur avec seulement deux étages de pipeline, ou sur un très gros processeur avec réordonnancement des instructions «*à la volée*» (*Out of Order pipeline*). Le binaire sera compatible avec les deux processeurs. Seules les performances changeront.

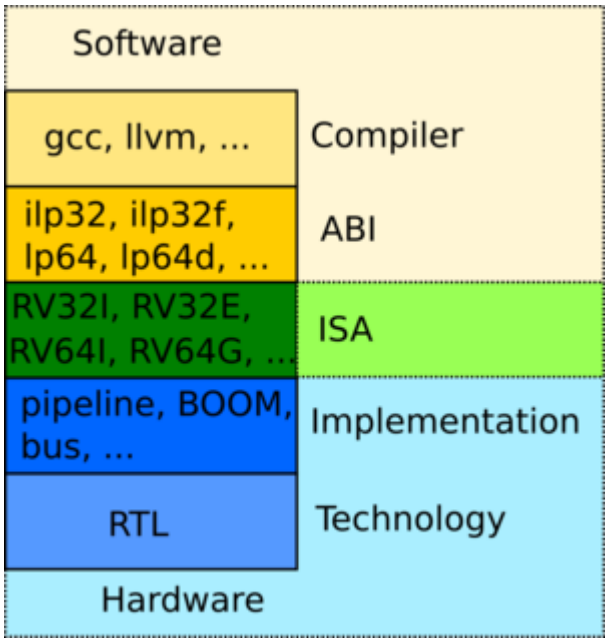


Fig. 1 : Le set d'instructions (ISA) est l'interface entre le logiciel et le matériel.

Côté logiciel, le set d'instructions est le langage dans lequel le compilateur devra convertir un programme pour qu'il fonctionne avec le microprocesseur. La définition de ce jeu d'instructions aura un impact direct sur les optimisations à effectuer pour accélérer l'exécution du code, ainsi que sur la complexité du compilateur.

Les jeux d'instructions de type **RISC** étant souvent très génériques, surtout concernant le banc de registres (*registers file*). Il est nécessaire de spécialiser certains registres pour pouvoir créer notamment des bibliothèques de fonctions qui pourront être liées au programme principal (registre d'adresses de retour, registre de pile…).

Le set d'instructions étant également très réduit, pour simplifier le codage, on définit également d'autres instructions que l'on appelle «*pseudo-instructions*» et qui seront converties à la génération du binaire.

Toutes ces définitions sont standardisées dans ce que l'on appelle une **ABI**, pour «*Application Binary Interface*». C'est l'interface entre le logiciel et le matériel, comme présenté en figure 1.

Pseudo Instructions

Les pseudo-instructions sont surtout là pour simplifier la vie du développeur logiciel et du compilateur. Elles consistent en des instructions qui seront converties ensuite en une suite d'instructions de l'ISA.

L'exemple typique est le chargement d'une constante de 32 bits dans un registre. Comme toutes les instructions sont codées sur 32 bits, il n'est pas possible de coder une valeur de 32 bits dans l'instruction. Il faut pour cela utiliser deux instructions :

- **lui rd, immediate[31:12]** : *Load unsigned immediate*, qui chargera les 20 bits supérieurs du mot.
- **addi r, x0, immediate[11:0]** : qui chargera les 12 bits inférieurs.

Ce chargement de constante est relativement fastidieux et suivant la constante à charger, il peut arriver qu'une seule instruction suffise.

Pour éviter au programmeur/compilateur de se casser la tête à chaque chargement, on définit donc la pseudo-instruction : **li rd, immediate**.

Ces pseudo-instructions font partie de l'ABI (*Application Binary Interface*).

3. Les bases et les extensions

Pour éviter à l'avenir d'avoir à gérer une compatibilité avec leurs anciennes ISA, les instructions RISC-V sont considérées comme immuables quand elles atteignent la version 2.0 de leurs définitions. À l'heure actuelle, les deux bases considérées comme stables sont **RV32I** et **RV64I**, avec des tailles de registres respectivement de 32 bits et 64 bits. L'encodage des instructions reste lui sur 32 bits.

Le jeu d'instructions est assez similaire entre le RV32I et le RV64I, nous allons parler ici de la version 32 bits.

Base	Version	Description
RV32I	2.0	Banc de registres de 32 bits, instructions arithmétiques entières simples.
RV32E	1.9	15 registres de 32 bits au lieu des 31 «normaux».
RV64I	2.0	Banc de registres de 64 bits, instructions arithmétiques entières simples.
RV128I	1.7	Banc de registres de 128 bits, instructions arithmétiques entières simples.

Les bases définissent le jeu minimum d'instructions permettant d'exécuter un programme. Mais ces instructions sont assez limitées, les fonctions arithmétiques ne comprennent pas la multiplication ou la division, par exemple. Pour éviter l'inflation du jeu *de base*, toutes les autres instructions sont et seront définies par des extensions. Pour la nomenclature, cela se traduit par une lettre majuscule accolée au nom.

Les différentes extensions sont données dans le tableau suivant. Tout comme les bases, certaines extensions sont déjà figées à leur version 2.0 et d'autres ne sont encore qu'à l'état de projet. L'objectif (ambitieux) des extensions est de pouvoir couvrir l'ensemble des besoins en matière de microprocesseurs que l'on parle de microcontrôleur très basse consommation ou de microprocesseur de calcul vectoriel massivement parallèle.

Extensions	Version	Description
M	2.0	Multiplication et division entière.
A	2.0	Instruction atomique.
F	2.0	Virgule flottante simple précision. Ajout de 32 registres 32 bits flottants (**f0-f31**) et d'un registre de statut **fcsr**.
D	2.0	Virgule flottante double précision. Agrandissement des registres flottants à 64 bits.
Q	2.0	Virgule flottante quadruple précision. Agrandissement des registres flottants à 128 bits.
L	0.1	Virgule flottante décimale. Reste à définir.

C	2.0	Instructions compressées. Cette extension ajoute des instructions codées sur 16 bits au jeu de base. Les instructions compressées fonctionnent avec deux arguments au lieu des 3 «*classiques*». Cela permet de réduire jusqu'à 60 % la taille du programme à charger.
B	0.0	Manipulation de bit. Reste à définir.
J	0.0	Instruction pour les langages compilés «*à la volée*». Reste à définir.
T	0.0	Mémoire transactionnelle, à définir.
P	0.1	Instruction SIMD (le MMX du RISC-V), à définir.
V	0.2	Instruction vectorielle. Reste à définir, mais le brouillon est déjà un peu avancé.
N	1.1	Gestion des exceptions/interruptions au niveau utilisateur.

Les extensions sont à accoler à la base choisie. Pour simplifier la nomenclature, les extensions usuelles que sont **IMAFD** sont compressées en « *G* ». Ainsi, **RV32G** et **RV64G** désignent respectivement **RV32IMAFD** et **RV64IMAFD**.

4. Format des instructions RV32I

Toutes les instructions RISC-V sont définies par des mots de 32 bits, soit 4 octets. Les instructions à exécuter par le processeur sont donc stockées dans une mémoire d'instructions et exécutées les unes après les autres, en incrémentant l'adresse mémoire de 4 à chaque cycle. Les accès à ces instructions se font via un bus d'instructions et la lecture (*fetch*) des instructions se fait par des accès 32 bits. Les accès étant cadrés, l'adresse de l'instruction suivante est donc +4 et est pointée par le registre PC.

L'intérêt d'un processeur est de pouvoir communiquer avec l'extérieur. Cette communication se fait via le bus mémoire avec les instructions *load* et *store*. Une des caractéristiques du jeu d'instructions RISC-V est que seules les instructions *load* et *store* permettent l'accès à la mémoire externe, le reste des instructions travaillera ensuite sur les registres internes du processeur.

La table des registres et de leurs fonctions dans l'ABI standard est donnée dans la table suivante :

Registre	**ABI**	**Appelant/Appelé**	**Description**
x0	Zero		Toujours à 0.
x1	ra	Appelant	Adresse de retour.
x2	sp	Appelé	Pointeur de pile.
x3	gp		Pointeur global.
x4	tp		Pointeur de thread.
x5-7	t0-2	Appelant	Variables temporaires.

x8	s0/fp	Appelé	Sauvegarde de registres ou pointeur de frame.
x9	s1	Appelé	Sauvegarde de registres.
x10-11	a0-1	Appelant	Arguments de fonction / valeurs de retour.
x12-17	a2-7	Appelant	Arguments de fonction.
x18-27	s2-11	Appelé	Sauvegarde de registres.
x28-31	t3-6	Appelant	Variables temporaires.

Un processeur RISC-V possède une base de 31 registres généraux de 32 bits (rv32I), numérotés de x1 à x31. Le registre x0 est câblé *en dur* à 0. Cela nous donne donc 32 registres de 32 bits. Ces 32 registres sont souvent appelés le banc de registres (*register file*) et sont accessibles indifféremment par toutes les instructions de l'ISA. Contrairement à d'autres architectures, les registres n'ont pas de fonctions spécifiques, et peuvent donc servir d'argument pour les fonctions arithmétiques, autant que de pointeur de pile ou de sauvegarde d'adresse pour l'appel de fonctions.

Seul le registre pointeur d'instructions PC est câblé indépendamment des registres généraux, pour simplifier le câblage et la gestion des prédictions de branches.

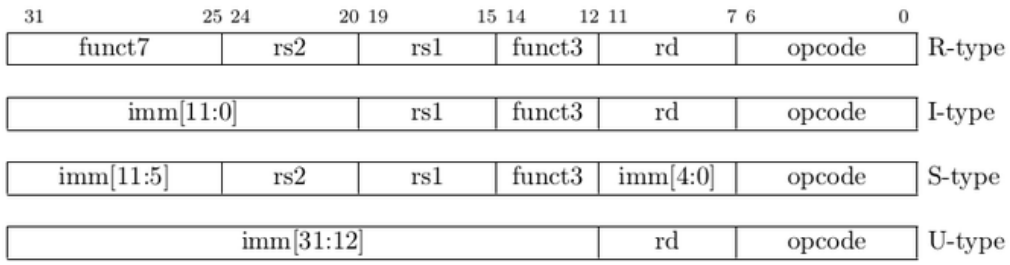


Fig. 2 : Format des instructions de la base RV32I.

Les formats des instructions RV32I sont donnés en figure 2. Dans un souci de simplicité du décodage d'instructions, les arguments (numéro de registres) **rs1** et **rs2** sont toujours placés au même endroit dans le mot de 32 bits. Même chose pour le registre de retour **rd**. Ainsi, le décodage des arguments peut s'effectuer *en dur*, sans avoir à câbler de condition spécifique.

Le code de l'instruction est donné par l'**opcode** avec parfois un complément **funct3** et **funct7**, suivant les besoins.

Comme nous pouvons le voir, les valeurs immédiates ne peuvent jamais être de 32 bits. Cependant, il est possible de charger la partie haute d'un mot de 32 bits, avec une instruction de type **U-type** (les 20 bits de poids fort), comme l'instruction **lui** et la partie basse avec une instruction de type **I-type**, comme l'instruction **addi**.

Nous n'allons pas décrire dans le détail chaque instruction RV32I, nous nous contenterons d'une vue générale des instructions pour nous rendre compte qu'elles ne sont pas si nombreuses.

Nous mettrons ensuite en application quelques instructions pour réaliser l'exemple de la LED qui clignote. Un exemple concret est toujours utile, pour bien se rendre compte du fonctionnement.

4.1 Load / Store

Nous avons vu en introduction qu'une architecture RISC peut être également décrite comme une architecture *load/store*. Ces instructions sont les seules permettant d'accéder à la mémoire de

données. Commençons donc par décrire les instructions de lecture et d'écriture dans la mémoire. La présentation des instructions est inspirée du livre **[2]**. Le nom de la fonction en assembleur correspond aux lettres en gras.

Les instructions de lecture et d'écriture dans la mémoire sont données en figure 3.

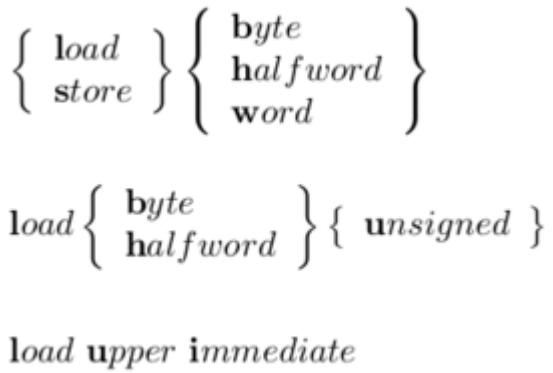


Fig. 3 : Instructions de lecture et d'écriture dans la mémoire.

Le RV32I permet de charger des mots signés de 8 (*bytes*), 16 (*halfword*) ou 32 bits (*word*). Les instructions de lecture étendent automatiquement le bit de signe pour les versions 8 et 16 bits. Pour éviter l'extension du bit de signe, il existe une version non signée (*unsigned*) pour les lectures 8 et 16 bits. Les instructions prennent en argument un registre (**rs1**) comme adresse de référence et une valeur immédiate signée d'offset, qui s'ajoute à cette adresse. La valeur chargée est stockée dans le registre donné en paramètre **rd**. Par exemple, l'instruction de chargement d'un demi-mot (*halfword*) de 16 bits est de la forme :

lh rd, offset(rs1)

L'instruction **lui** n'est pas réellement une instruction de chargement depuis la mémoire, car elle charge une constante sur 20 bits codée dans l'instruction elle-même. La possibilité de charger des constantes est importante en programmation assembleur, mais elle pose problème en architecture RISC, si l'on veut garder la règle d'une instruction sur 32 bits. Comme les registres sont eux aussi sur 32 bits, il est impossible d'encoder la totalité de la valeur d'un registre dans une instruction. La solution trouvée est de le faire en deux instructions, les 20 bits de poids fort avec l'instruction **lui**, et les 12 bits de poids faible avec la valeur «*immédiate*» d'une instruction arithmétique (souvent **addi**).

4.2 Instructions arithmétiques

Les instructions arithmétiques sont données en figure 4.

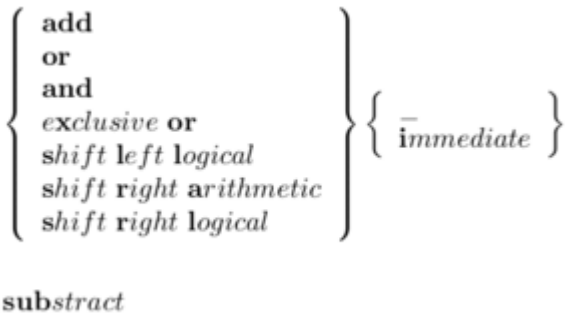


Fig. 4 : Instructions arithmétiques.

Les instructions arithmétiques prennent deux registres (**rs1** et **rs2**) en paramètres et inscrivent le résultat dans un troisième (**rd**). Il est possible de remplacer le second paramètre par une valeur immédiate sur 12 bits, le bit de poids fort sera alors étendu aux 20 bits de poids fort pour respecter le signe (*signe extended*).

En jouant avec le registre **x0** et l'extension du signe, il est par exemple possible d'utiliser l'instruction **xor** pour faire un **not** sur un registre :

xori rd, x0, -1

Contrairement à beaucoup d'autres jeux d'instructions, le RISC-V ne possède pas de registres de statut activant des drapeaux, en fonction du résultat du calcul. Les instructions de branchement incluent donc les fonctions de comparaison, décidant si l'on doit «*sauter*» (*branch*) à un autre endroit du programme.

4.3 Instructions de branchement

Les instructions de branchement sont données en figure 5.

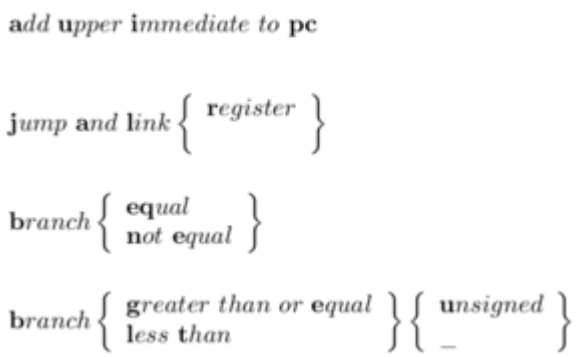


Fig. 5 : Instructions de branchement.

Les instructions de branchement sont des instructions de comparaison qui modifient le pointeur de programme (*PC* pour *program counter*). Les concepteurs du jeu d'instructions ont choisi de mettre la fonction de comparaison dans la fonction de branchement. Cela permet de simplifier la programmation, car il n'y a pas à faire attention à l'ordre d'exécution des instructions, comme on pourrait l'avoir sur ARM ou x86.

L'offset de branchement est une valeur signée qui est ajoutée au **PC**, s'il y a branchement. Cette valeur immédiate étant codée sur 12 bits, il n'est possible de brancher que sur une zone mémoire de ±2 Ko. Pour sauter plus loin, on utilisera la fonction **auipc** qui permet d'ajouter une valeur immédiate de 20 bits sur le poids fort du PC et donc, d'accéder à tout l'espace mémoire sur 32 bits.

L'instruction de saut **jal** est utilisée pour l'appel de fonction. Cette fonction va sauver l'adresse courante du PC (+4) dans le registre donné en argument (*rd*), puis «*sauter*» à l'offset donné en argument (valeur immédiate ou registre). Le pointeur initial étant sauvegardé dans un registre, il sera aisé de revenir au flux d'instructions initial.

5. Comparaison

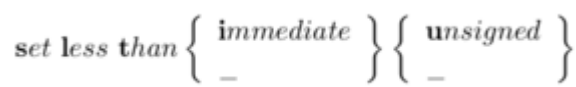
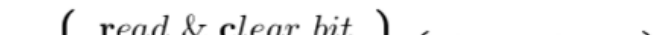


Fig. 6 : Instructions de branchement.

Une instruction de comparaison présentée figure 6 a été ajoutée, en plus des comparaisons se trouvant dans les instructions de branchement. Elle permet de mettre à 1 le registre de résultat (*rd*) si le premier argument est inférieur au second. Cette instruction se décline en valeur immédiate, signée ou non.

5.1 Registres d'états et de contrôle



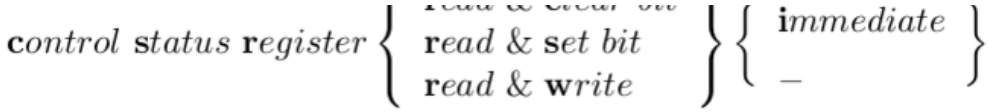


Fig. 7 : Instructions de branchement.

Le standard RISC-V possède une série de registres appelés *CSR* (4096 maximum) de contrôle et de statut, permettant de gérer l'état du processeur. Ces registres sont décrits dans le volume II de la spécification **[3]** (spécifications qui ne sont pas encore totalement définitives et évoluent, cf. **[12]**). Ils contiennent notamment des compteurs permettant de faire des mesures de performance. C'est aussi dans ces registres que l'on trouvera les informations sur la version du jeu d'instructions utilisé (avec les extensions).

Ils ne font pas partie de l'espace mémoire de données «*normal*» et peuvent être lus/écrits/modifiés de manière atomique. D'où la nécessité d'utiliser des instructions spécifiques pour les manipuler. Ces instructions sont présentées dans la figure 7.

Si l'on prend l'exemple de l'instruction *control and status register read and clear* :

csrrc rd, csr, rs1

Cette instruction va lire le registre donné en argument (*csr*) et l'écrire dans le registre *rd*. Elle va également mettre à 0 tous les bits du registre *csr* qui sont à 1 dans *rs1*.

5.2 Synchronisation de mémoire

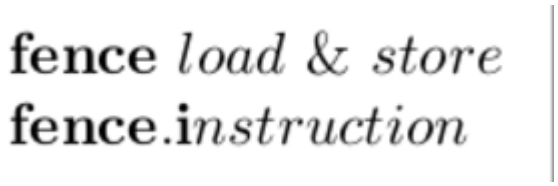


Fig. 8 : Instructions de branchement.

Le jeu d'instructions RISC-V peut être utilisé sur des systèmes à plusieurs cœurs, ainsi qu'avec de la mémoire cache. Il est donc nécessaire de fournir un mécanisme de synchronisation de la mémoire d'instructions et de données. C'est le rôle des instructions **fence.i** et **fence** que l'on peut voir figure 8. Ces deux instructions vont s'assurer que la mémoire est disponible, avant de continuer le flot d'exécution.

6. Une LED qui clignote sur RV32I (HiFive1)

Maintenant que nous avons vu l'ensemble des instructions disponibles sur une base RISC-V, nous pouvons attaquer un petit exemple de la classique LED qui clignote. La LED qui clignote est le *«Hello World»* de l'électronicien ainsi que du développeur embarqué, toute sa carrière est menée par des LED qui clignotent.

Le programme fait clignoter une LED avec une cadence de type cœur qui bat (*heartbeat*). On allume brièvement la LED, on l'éteint brièvement, puis on l'allume à nouveau brièvement et enfin, on l'éteint plus longtemps.

Bien sûr, de nos jours, le développement sur microcontrôleur se fait avec des langages de programmation plus évolués. Plus personne ne code en assembleur, à part pour des parties très spécifiques d'un programme. Mais le but ici est de se mettre au plus près du jeu d'instructions, pour bien comprendre l'esprit du RISC-V. Pour développer une «*vraie*» application, on pourra utiliser un langage évolué de son choix, vu que GCC intègre RISC-V et que LLVM est en bonne voie également. Tous les langages de programmation comme le C, C++, Rust, Ada,… peuvent être utilisés sur RISC-V sans trop de problèmes, aujourd'hui.

Pour être vraiment concrets dans notre exemple, nous allons utiliser le kit de développement

HiFive1, à base de E310 produit par SiFive, qui peut être commandé pour une cinquantaine de dollars, en passant par le site officiel. Le kit se présente sous la forme d'une carte électronique *«compatible arduino»*, comme le montre la photo de la figure 9. Une seconde version du kit avec un module Wi-Fi a été lancée, début 2019.

Fig. 9 : La carte de développement HiFive1, munie d'un processeur E310 de SiFive.

SiFive fournit le SDK et OpenOCD, ainsi que la chaîne de développement (*toolchain*) pour Linux. L'installation sous Linux à base d'Ubuntu/Debian est donc triviale. Toutes les informations pour installer les outils de développement sont disponibles dans le guide de démarrage **[4]**, nous les résumons juste ici.

Les plus motivés iront chercher les sources de la chaîne de développement sur GitHub, avec la commande suivante :

$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain

Puis, compilerons le tout avec **make**.

Les plus pressés iront eux chercher la chaîne et l'outil **openocd** précompilés pour leur système d'exploitation (les binaires Ubuntu fonctionnent très bien sur Debian) sur le site de Sifive. Puis, les décompresseront dans le répertoire de leur choix.

$ wget https://static.dev.sifive.com/dev-tools/riscv64-unknown-elf-gcc-8.2.0-2019.02.0-x86_64-linux-ubuntu14.tar.gz

$ tar -zxvf riscv64-unknown-elf-gcc-8.2.0-2019.02.0-x86_64-linux-ubuntu14.tar.gz

$ wget https://static.dev.sifive.com/dev-tools/riscv-openocd-0.10.0-2019.02.0-x86_64-linux-ubuntu14.tar.gz

$ tar -zxvf riscv-openocd-0.10.0-2019.02.0-x86_64-linux-ubuntu14.tar.gz

Pour enfin configurer leur **PATH** RISC-V dans leur fichier de configuration préféré (**.bashrc**, concernant l'auteur).

export RISCV_OPENOCD_PATH=/hifive1/riscv-openocd-0.10.0-2019.02.0-x86_64-linux-ubuntu14/

export RISCV_PATH=~/hifive1/riscv64-unknown-elf-gcc-8.2.0-2019.02.0-x86_64-linux-ubuntu14/

Pour OpenOCD, il est nécessaire de configurer le convertisseur USB-UART, en ajoutant les règles **udev** comme expliqué dans le guide de démarrage. Puis, d'ajouter le groupe *plugdev* à son utilisateur. OpenOCD va nous servir à télécharger le binaire dans le microcontrôleur.

La chaîne de développement spécifique au HiFive1 (SDK) est disponible sur un GitHub, qu'il suffit de cloner :

$ git clone --recursive https://github.com/sifive/freedom-e-sdk.git

Ce SDK inclut toutes les plateformes de développement proposées par la société SiFive, nous nous intéresserons ici à la plateforme nommée **hifive1**.

Des projets d'exemple sont donnés dans le répertoire « *software* ». Pour les compiler, il faut se placer dans le répertoire racine du SDK et faire un **make** avec la plateforme de destination (ici le HiFive1), ainsi que le nom du projet d'exemple :

$ cd freedom-e-sdk/

$ make software PROGRAM=hello BOARD=sifive-hifive1

Le téléchargement du binaire compilé se fait également avec une commande **make** :

$ make upload PROGRAM=hello BOARD=sifive-hifive1

Le programme «*hello*» ne fait qu'afficher «*hello world*» sur la seconde interface série (**ttyUSB1**), présente sur le kit. Pour simplifier le développement de notre programme assembleur, nous allons utiliser cet exemple, en remplaçant le source **hello.c** par **hello.S** :

$ cd software/hello

$ mv hello.c hello.c.save

$ vim/nano/emacs/edit hello.S

À l'origine, les **Makefile** du SDK sont prévus pour compiler du C, mais il suffit de déclarer un label **main** comme **global** dans notre code assembleur pour pouvoir «*compiler*» le programme assembleur au format S.

Le programme assembleur complet est à retrouver parmi les sources associées au magazine, nous allons le détailler ici.

Dans l'en-tête du programme, nous indiquons que nous entrons dans la section texte :

```
.section .text
```

Puis, l'on donne le cadrage des données :

```
.balign 4
```

Comme nous l'avons vu précédemment, toutes les instructions sont codées sur 32 bits. Il est donc nécessaire de cadrer sur 4 octets. Le mot clef **.balign** donne la valeur du cadrage en octets, on peut également utiliser le mot clef **.align**, qui prend en argument une puissance de 2 :

```
.align 2
```

Pour terminer l'en-tête, nous allons déclarer le label **main** comme global, pour qu'il soit reconnu par GCC comme le point d'entrée :

```
.globl main
```

Pour se simplifier un peu la vie, nous allons déclarer quelques constantes, au moyen du mot clef **.equ** :

```
.equ GPIO_BASE, 0x10012000

.equ GPIO_OUTPUT_VALUE, 0x00C

.equ RED, 1<<22

.equ GREEN, 1<<19
```

```
.equ BLUE, 1<<21

.equ LEDCONFIG, (RED | GREEN | BLUE)

.equ LEDCOLOR, ~GREEN

.equ LONGWAIT, 5000000

.equ SHORTWAIT, 1000000
```

La LED RGB (rouge, vert, bleu) est vue par le microcontrôleur comme trois LED connectées sur des GPIO. Le port de GPIO du E310 peut être vu comme un registre de 32 bits, avec une GPIO pour chaque bit. Sur le HiFive1, nous avons la LED verte sur le bit 19, rouge sur le bit 22 et bleue sur le bit 21.

Le point d'entrée du programme est donc indiqué par le label **main**.

```
main :
```

Nous allons commencer par écrire quelques constantes globales dans les registres **s1-3**, prévus à cet effet dans l'ABI, qui nous seront utiles par la suite.

```
li s1, GPIO_OUTPUT + GPIO_BASE

li s2, GPIO_OUTPUT_VALUE + GPIO_BASE

li s3, LEDCONFIG
```

Puis, nous appelons la «*fonction*» permettant de configurer les trois GPIO des LED en sortie :

```
jal ra, ledconfig
```

Cette instruction prend le registre d'adresses de retour en argument, ainsi que l'offset vers lequel il faut transférer le flux d'exécution. Nous utilisons ici le nom **ra** correspondant à **x1** dans l'ABI standard.

Dans la fonction **ledconfig**, nous allons commencer par réserver 16 octets dans la pile :

```
addi sp, sp, -16
```

pour sauvegarder l'adresse de retour :

```
sw ra, 12(sp)
```

Nous sauvegardons l'adresse de retour à l'offset 12, de manière à garder de la place pour d'autres arguments ou valeurs, si besoin.

L'utilisation de la pile ici n'est que peu utile, puisque nous n'appelons pas de fonctions dans cette fonction. Nous pourrions tout-à-fait nous en passer dans ce cas précis, mais y penser est une bonne habitude, au cas où nous aurions besoin d'appeler une sous-fonction.

Seuls 3 bits du registre de sortie des GPIO doivent être modifiés, nous devons donc commencer par lire sa valeur :

```
lw t0, 0(s1)
```

Pour ensuite mettre à 1 les trois bits concernés, au moyen d'un **or** et réécrire le registre :

```
or t0, s3, t0

sw t0, 0(s1)
```

Puis, on dépile l'adresse de retour, avant de retourner au flux d'exécution principal :

```
lw ra, 12(sp)

addi sp, sp, 16
```

ret

La pseudo-instruction **ret** permet de faire un retour au flux d'exécution principal. Cette pseudo-instruction est un raccourci pour :

jalr x0, 0(ra)

La suite du programme est une boucle infinie, appelant alternativement les fonctions permettant d'allumer et d'éteindre la LED et la fonction d'attente. Ces fonctions prennent un argument au moyen du registre **a0**, prévu pour cela dans l'ABI.

La fonction d'attente initialise le registre temporaire **t0** à 1 :

addi t0, x0, 1

Puis, l'incrémente en boucle, tant que **t0** est inférieur à la valeur donnée en argument **a0** :

waitloop:

addi t0, t0, 1

bltu t0, a0, waitloop

Ce programme de clignotement de LED permet de montrer concrètement ce qu'est un programme écrit en assembleur RISC-V. Il peut être largement amélioré. On laissera en exercice au lecteur le soin de l'optimiser, par exemple en fusionnant les deux fonctions **setColor** et **offled**. Ou en réduisant, voire en supprimant l'utilisation de la pile.

L'idée était surtout d'avoir un exemple concret mettant en œuvre un vrai processeur physique, pour sortir de la théorie et des simulations.

7. Émulation/FPGA

Il existe de nombreux simulateurs permettant de se faire la main avec RISC-V. Ces simulateurs émulent généralement le fonctionnement du jeu d'instructions seul. Cela permet de s'affranchir de l'implémentation et d'accélérer l'émulation.

Le simulateur développé initialement par Berkeley pour développer l'ISA est **Spike [5]**. Il permet d'exécuter pas-à-pas les instructions sur un processeur virtuel, muni de plusieurs cœurs en parallèle. On peut y brancher son outil de déverminage GDB préféré.

L'émulateur **Qemu** est déjà bien connu dans le monde du Libre et au-delà. Une architecture à base de cœurs rv64 et rv32 de chez SiFive a été intégrée au logiciel **[6]**. Cette architecture a permis de faire le portage de RISC-V pour Linux 5.0. Plusieurs distributions Linux ont également pu commencer l'intégration de l'architecture, grâce à cet émulateur.

Chose étonnante, sous Linux, il est également possible de faire tourner son binaire RISC-V sur un pc x86, grâce à l'interface du noyau nommée **binfmt_misc** et le moteur **rv8 [7]**.

Tous ces émulateurs fonctionnent au niveau du jeu d'instructions. Mais il est également possible de simuler le fonctionnement du processeur complet, à partir du moment où l'on a accès au code source matériel (*Hardware Description Language*). C'est notamment le cas des nombreux «*softcore*» disponibles pour FPGA.

La société Microsemi (appartenant à Microchip) propose d'ailleurs tout un environnement de développement nommé **Mi-V**, pour utiliser des processeurs RISC-V dans ses FPGA (pour PolarFire, RTG4 et Igloo2).

Les cœurs développés par la société SiFive sont synthétisables pour la plupart sur FPGA, histoire de pouvoir tester leurs processeurs en avance de phase. Ils utilisent d'ailleurs un nouveau langage de description open source nommé **Chisel** pour décrire leurs processeurs. Ce langage basé sur **Scala** permet une meilleure réutilisation du code, grâce à un langage-objet et fonctionnel. Le source reste synthétisable, car le format de sortie est du *Verilog*.

Le Picorv32 est intéressant, car optimisé pour avoir une taille minimale, ce processeur initialement destiné aux FPGA a été utilisé par Tim Edwards, pour démontrer qu'il est possible de réaliser un composant en silicium au moyen d'outils open source. Il a pour cela produit un processeur nommé **RAVEN**. Une présérie du Raven, cadencé à 150 MHz et gravé en 180 nm, a déjà été produite en 2018 et une série est en préparation chez **eFabless**.

Enfin, pour finir ce rapide tour des processeurs «*softcore*» sur FPGA, n'oublions pas de citer le **VexRiscv [8]**, développé par Charles Papon et qui a gagné le concours organisé par la fondation RISC-V. L'objectif du concours était de développer un processeur RV32I à destination d'un FPGA, en utilisant le moins de ressources possibles, mais qui soit tout de même rapide. L'originalité de ce processeur est d'être développé avec un nouveau langage de description matériel, développé par l'auteur : **SpinalHDL [9]** Ce langage est également basé sur le *Scala*.

8. Les siliciums

L'université de Berkeley a déjà produit plus d'une dizaine de variantes de processeurs RISC-V. Mais ces siliciums sont principalement des prototypes de recherche. De même, beaucoup de laboratoires de recherche, comme le **SHAKTI** de l'Institut de Technologie de Madras en Inde **[10]** ou le projet **PULP** de l'université de Zurich **[11]**, se basent désormais sur cette ISA pour leurs processeurs.

En matière de production de composants réels, les choses ont commencé à bouger en 2017, avec la sortie du premier processeur RISC-V vraiment accessible au «*grand-public*» : le E310, produit par la société SiFive. SiFive est une société sans fabrication (*fabless company*), dont les fondateurs sont issus de l'université de Berkeley.

SiFive ne pouvait rester crédible, sans sortir un premier processeur physique sur ses propres fonds. Ce qu'elle a fait avec le E310, d'autres sociétés ont ensuite emboîté le pas, avec le **Kendryte** E210 d'une société chinoise et le RV32M1, de **NXP**.

Les siliciums qui nous intéressent ici sont ceux que nous pouvons nous procurer, même si le prix n'est pas toujours tout à fait abordable pour bricoler dans son garage.

Voici donc une petite liste de processeurs/microcontrôleurs RISC-V en silicium, déjà accessibles aujourd'hui.

8.1 E310 : le processeur « Compatible arduino »

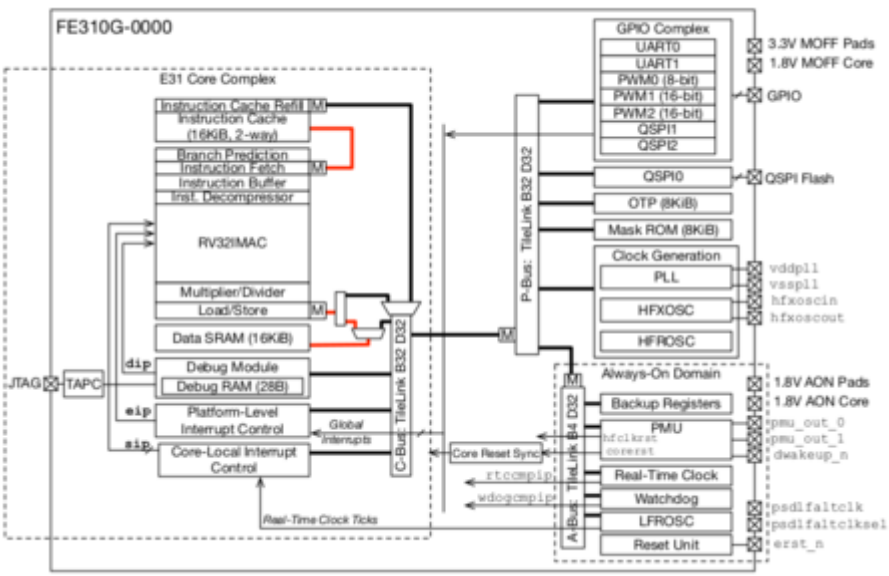


Fig. 10 : Schéma-bloc de fonctionnement du E310, constituant la carte HiFive1.

Le E310 est un microcontrôleur à cœur RV32IMAC, gravé en 180 nm et pouvant être cadencé à

320 MHz maximum. Le composant, présenté en figure 10, possède une seconde horloge permettant de cadencer un mode basse consommation, pour faire fonctionner les périphériques se trouvant dans le domaine AOD (*always on domain*) pour permettre le fonctionnement de l'horloge, du watchdog et la sauvegarde de certains registres.

SiFive propose un kit de développement basé sur le standard Arduino pour s'y initier, comme nous avons pu le tester dans le programme d'exemple.

La carte s'intègre très bien à l'IDE Arduino à partir de la version 1.8.5, il suffit d'ajouter l'URL : http://static.dev.sifive.com/bsp/arduino/package_sifive_index.json dans les préférences de l'interface : « *file* », « *preferences* », « *Additional Boards Managers URLs* ».

Puis, de se rendre dans le *board manager* : « *Tools* », « *Board* : », « *Boards Manager* », pour y installer la carte HiFive1. Il faut ensuite sélectionner «*SiFive open ocd*» comme programmeur dans le menu « *Tools* », « *Programmer* : ».

L'utilisation de l'IDE d'Arduino est parfaite pour pouvoir jouer avec son cadeau, dès réception du colis. Les puristes préféreront passer assez vite à un langage plus «*pro*».

Il est également possible de se procurer le processeur nu (25 $ les 5), pour se faire sa propre carte électronique.

Une nouvelle version, le HiFive1 Rev. B, est sortie en 2019, avec l'ajout d'un module Wi-Fi/Bluetooth à la carte de développement.

8.2 U540 : le premier processeur pour Linux

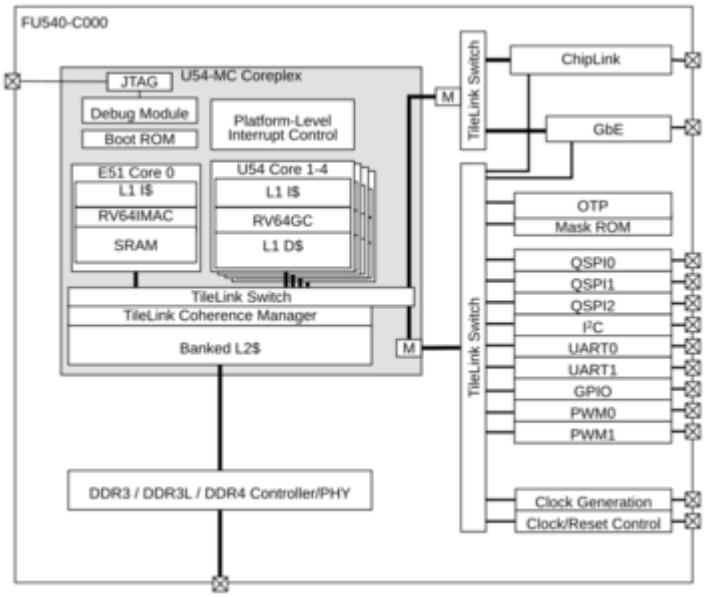


Fig. 11 : Schéma-bloc du processeur U540, constituant de la carte HiFive Unleashed.

Le U540 peut être considéré comme le premier microprocesseur physique permettant de faire fonctionner Linux. C'est un quad-cœur RV64GC, muni d'un «compagnon» RV64IMAC (nommé E51). Le schéma-bloc du processeur est donné figure 11. Gravés en 28 nm, les cœurs peuvent être cadencés jusqu'à 1.5 GHz.

Le kit de développement est proposé sous la forme d'une carte nommée HiFive Unleashed. Le tarif de 1000 $ en fait un kit plus destiné aux bureaux d'études qu'aux particuliers, mais il évite l'utilisation de FPGA et permet donc une utilisation en pleine puissance.

8.3 Kendryte K210 : l'intelligence artificielle embarquée

Le K210 est produit par une société chinoise. Gravé en 28 nm, il est composé d'un double cœur

RV64GC, cadencé à 400 MHz.

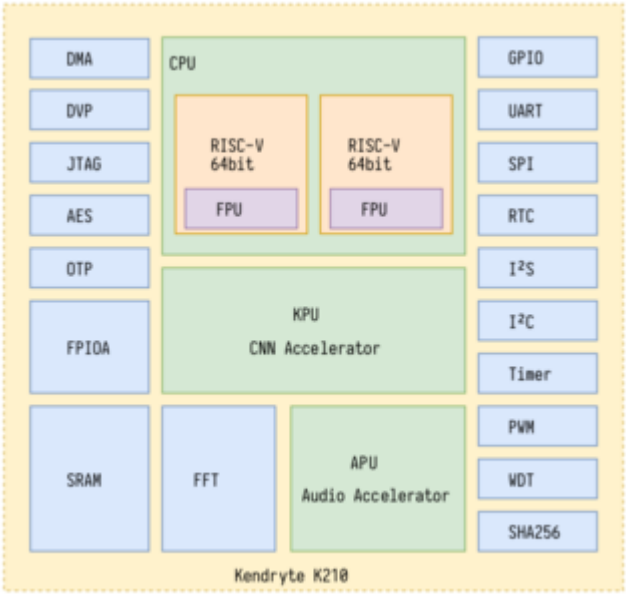


Fig. 12 : Schéma-bloc du K210 produit par Kendryte.

Le composant (figure 12) possède tout ce qu'il faut pour faire du traitement de signal audio et vidéo, ainsi qu'un coprocesseur de réseau de neurones.

Plusieurs kits intégrant le processeur sont disponibles à la vente. La plupart de ces kits utilisent un module intégré nommé SiPeed M1W permettant d'ajouter le Wi-Fi, comme nous pouvons le voir sur la photo de la figure 13.

Fig. 13 : Carte de développement SiPeed MaixGo, disponible pour environ 50 € avec caméra et écran LCD.

8.4 RV32M1 : le Wi-Fi Bluetooth avec multicœurs hétérogènes

Le RV32M1 a été lancé par NXP, pour équiper la carte VegaBoard. Cette carte est disponible à la vente pour une soixantaine de dollars.

Ce microcontrôleur est un ovni dans le monde de l'embarqué. Il possède 4 cœurs de processeurs hétérogènes, puisqu'il mixe des cœurs ARM (un cortex-M0 et un cortex M4) et deux RISC-V (RISCY et RI5CY).

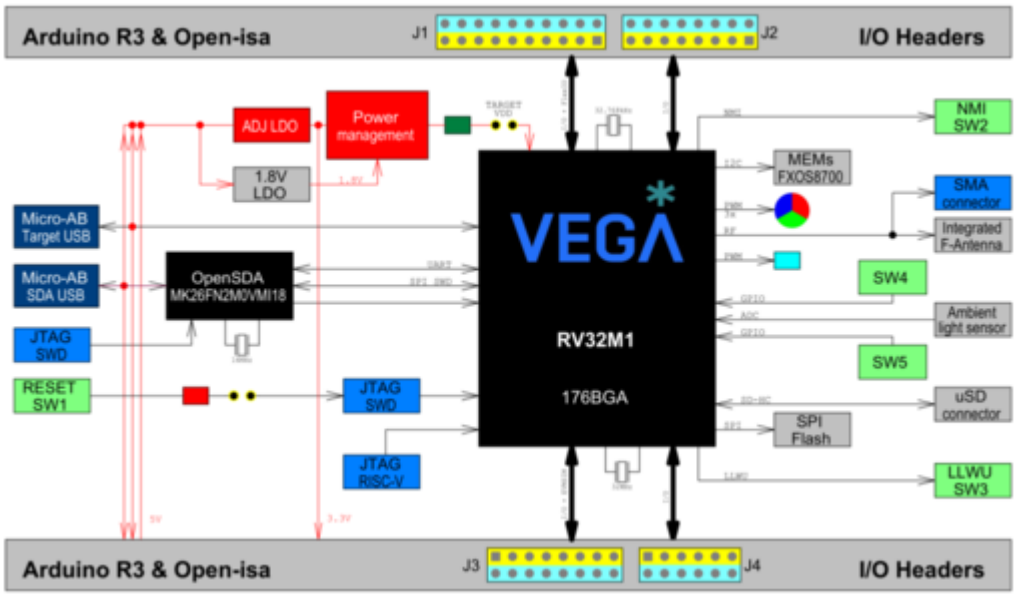


Fig. 14: La VegaBoard permettant de faire du multicœur hétérogène, avec tout l'étage radio 2.4 GHz.

Le développement de cette carte est la pierre angulaire d'un projet nommé OpenISA.

8.5 GAP8 : le parallélisme embarqué

Produit par GreenWaves Technologies, le GAP8 est un microcontrôleur massivement parallèle, muni de 8 cœurs RI5CY (RV32IMFCXpulp). Le GAP8 est une mise en application de l'architecture PULP, de l'université de Zurich. Cette architecture vise la très basse consommation dans les microcontrôleurs embarqués.

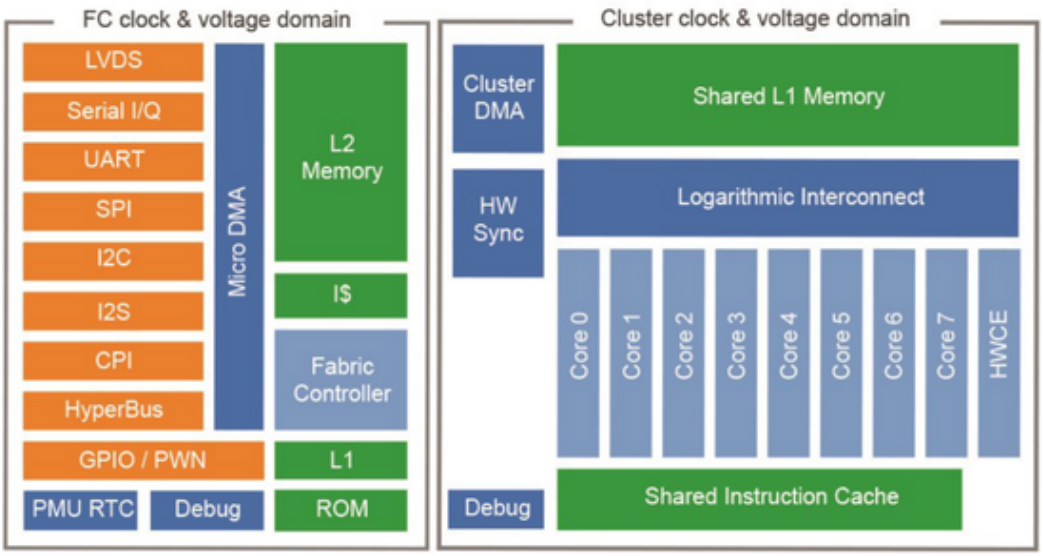


Fig. 15: Schéma-bloc donnant l'architecture du GAP8.

Il est possible de se procurer un kit de développement compatible Arduino (GAPUINO) pour une

centaine d'euros sur le site de GreenWaves Technologies (https://greenwaves-technologies.com/).

8.6 RAVEN : open source jusqu'au bout du silicium

Le RAVEN est un peu spécial, car il n'existe pas encore de kit de développement ni de possibilité de se procurer le composant, à l'heure où ces lignes sont écrites.

Cependant, ce microcontrôleur est remarquable dans la mesure où il a été conçu intégralement avec des logiciels au code ouvert. Il est muni d'un cœur RV32I nommé **Picorv32**, cadencé à 100 MHz avec une promesse de monter à 150 MHz. Il possède un convertisseur analogique numérique, ainsi que numérique analogique et est gravé en 180 nm. La version «*virtuelle*» est disponible, sans coût de licence, sur la plateforme **eFabless**.

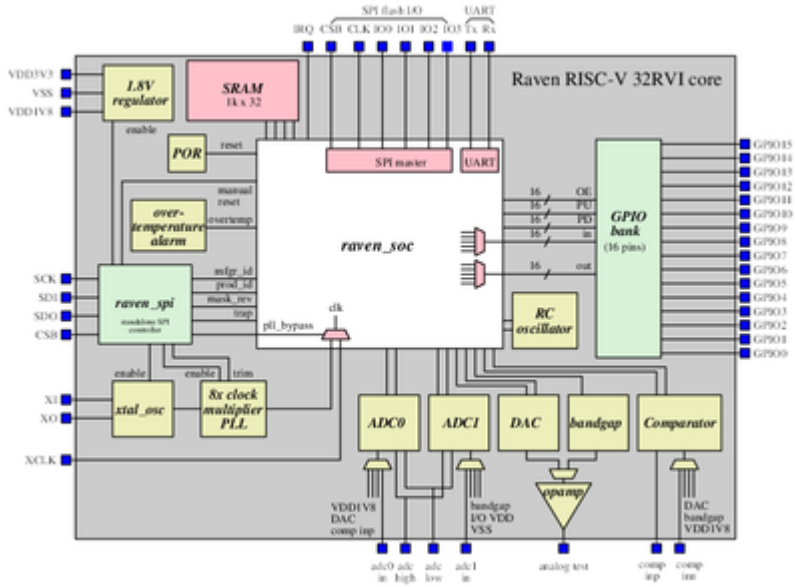


Fig. 16: Schéma-bloc du RAVEN, un microcontrôleur conçu exclusivement avec des logiciels libres.

8.7 GD32VF103 : le microcontrôleur généraliste à bas coût

Produit par la société chinoise GigaDevice, le GD32VF est une déclinaison RV32IMA de leur microcontrôleur GD32F, basée initialement sur un cœur ARM Cortex-M3.

Cadencé à 108 MHz, ce microcontrôleur embarque jusqu'à 128 Ko de flash et 32 Ko de mémoire vive.

Suivant le modèle utilisé, le GD32VF propose les périphériques suivants : SPI, I²C, USART, deux ADC 12-Bit à 1Msps, deux DAC 12-Bit, ainsi que deux bus CAN et un port USB.

Plusieurs kits de développement sont disponibles pour tester le composant, mais le Longan Nano proposé par la société Sipeed marque les esprits, avec son prix de 5 $ pour un kit incluant un écran LCD.

Fig 17 : Le kit Longan Nano, proposé à 4,9 $ sur le site de vente en ligne SeeedStudio.

9. Ceci n'est pas qu'une évolution technologique, mais une révolution industrielle

Le RISC-V est un nouveau jeu d'instructions, ce qui n'était pas arrivé depuis un certain temps. Les jeux d'instructions actuels sont surtout des évolutions de jeux d'instructions des années 70-80. Cette nouveauté a permis aux concepteurs d'apprendre des erreurs des autres, ainsi que de récupérer les bonnes idées.

RISC-V n'est pas révolutionnaire par les concepts appliqués, c'est surtout une évolution. Beaucoup de concepts (le registre câblé à 0, par exemple) rappellent ceux du MIPS. Le standard RISC-V permet d'ajouter ses propres instructions personnalisées, en plus du standard (ce que fait le RI5CY, par exemple).

Cependant, RISC-V est une révolution industrielle, car c'est un jeu d'instructions libre. N'importe quel industriel peut s'en servir pour faire son microprocesseur. Le nombre de grands noms de l'industrie électronique et informatique qui font partie de la fondation RISC-V peut nous mettre en confiance sur sa pérennité et sur son avenir.

La licence du jeu d'instructions est de type BSD (quelle surprise pour une ISA venant de Berkeley !), ce qui signifie qu'elle n'est pas «*contaminante*». Il est possible de reprendre le jeu d'instructions tel quel et de garder son architecture de processeur jalousement fermée, il est également possible de l'étendre sans rien publier. Cette «*non contamination*» rassure beaucoup les industriels et assure un certain essor pour cette ISA.

La fondation RISC-V a tout de même mis en place un système de numérotation avec l'identifiant des constructeurs (dans les registres CSR), comme on peut le trouver sur l'USB ou sur l'Ethernet avec les adresses MAC. Si l'on veut référencer son processeur et avoir un identifiant constructeur, il faut donc payer sa dîme à la fondation.

Le monde du microcontrôleur commence à être bousculé par l'arrivée de cette ISA. La société ARM, qui écrasait le marché du microprocesseur embarqué, prend ce petit nouveau très au sérieux. MIPS a également bougé, avec son annonce fracassante de la libération de son ISA sur le modèle RISC-V.

Seul Intel n'a pas encore vraiment bougé. Cela est principalement dû au fait qu'il n'y ait pas encore de processeur RISC-V rivalisant avec les performances des x86. Les extensions du jeu d'instructions permettant de rivaliser avec ceux-ci ne sont pas encore stabilisées et la finesse de gravure de ce genre de processeur nécessitant des investissements très lourds, personne ne s'est

encore positionné dessus.

Un standard libre et stable comme RISC-V permet également une simplification des logiciels de compilation, puis des systèmes d'exploitation. Linux supporte déjà l'ISA RISC-V dans son dernier kernel 5.0.

Tous les logiciels de compilation, qu'ils soient libres ou propriétaires, se sont lancés dans le portage de leurs outils pour RISC-V. GCC est déjà mature, LLVM ne va pas tarder. Et beaucoup d'OS temps-réel comme Zephyr, FreeRTOS… sont déjà bien avancés sur le sujet.

RISC-V est une révolution^{tm}, le marché des processeurs du futur devra composer avec cette ISA.

Références

[1] PRADOS Philippe, *« Au Cœur des microprocesseurs »*, *GNU/Linux Magazine n°218*, septembre 2018, pp. 66 à 81.

[2] PATTERSON David et WATERMAN Andrew, *« The RISC-V Reader, An Open Architecture Atlas »*, Strawberry Canyon LLC.

[3] Andrew Waterman, Krste Asanović, *«The RISC-V Instruction Set Manual - Volume II: Privileged Architecture»*, SiFive Inc.

[4] SiFive HiFive1 Getting Started Guide *«SiFive HiFive1 Getting Started Guide»*, https://www.sifive.com/documentation

[5] Spike RISC-V ISA Simulator, https://github.com/riscv/riscv-isa-sim

[6] RISC-V with QEMU, https://wiki.qemu.org/Documentation/Platforms/RISCV

[7] Michael Clark, Bruce Hoult, *«rv8 : a high performance RISC-V to x86 binary translator»*,CARRV 2017, octobre 2017, Boston, MA, USA.

[8] Charles Papon,*«VexRiscv»*, https://github.com/SpinalHDL/VexRiscv

[9] Charles Papon, *«SpinalHDL»*, https://github.com/SpinalHDL/SpinalHDL

[10] SHAKTI, IIT de Madras, http://shakti.org.in

[11] Parallel Ultra Low Power, http://iis-projects.ee.ethz.ch/index.php/PULP

[12] Spécification RISC-V : https://riscv.org/specifications/