L'émergence de l'architecture RISC-V

La Revue 3EI
Culture Sciences de l'Ingénieur

Édité le 24/07/2025

école normale supérieure paris—saclay

Théo BALLET[1] – Anthony JUTON[2]

[1] *Doctorant au Centre de Nanosciences et NanoTechnologies (C2N), moniteur à l'ENS Paris-Saclay - DER Sciences de l'Ingénierie Électrique et Numérique*

[2] *Professeur agrégé à l'ENS Paris-Saclay – DER Sciences de l'Ingénierie Électrique et Numérique*

Cette ressource fait partie du N°116 de La Revue 3EI du 3ième trimestre 2025.

Cette ressource présente l'architecture RISC-V, ses applications industrielles et l'intérêt qu'elle présente pour l'enseignement de l'informatique et l'introduction à la conception de circuits intégrés informatiques (co-design). Elle est suivie d'une seconde ressource présentant des exemples pratiques d'implémentation d'un cœur RISC-V sur un FPGA : « Exemples d'implémentation d'un processeur RISC-V sur un FPGA » [1].

1 - Qu'est-ce que RISC-V

Pour qu'un programme puisse fonctionner, sur un certain processeur, il utilise le vocabulaire de l'interface programme/machine de ce processeur. On parle alors de jeux d'instructions (ISA pour Instruction Set Architecture dans la langue de Shakespeare).

Dans les années 70-80 les processeurs étaient souvent programmés directement avec le langage de leur ISA, aussi appelé assembleur (asm). Aussi, pour avoir plus de performance, par exemple pour calculer des fonctions trigonométriques, on ajoutait un élément dans le processeur qui faisait ces calculs et on ajoutait des instructions pour réaliser ces opérations.

Cette philosophie a conduit à une accumulation de composants dans les cœurs des processeurs et à une accumulation d'instructions dans les ISA. On parle donc de processeurs CISC (Complex Instruction Set Computer) parce que leur vocabulaire de base est complexe.

Cette approche avait beaucoup d'avantages pour l'industrie des années 70 qui était une industrie de niche, avec peu de modèles de processeurs (principalement le Zilog z80 et le intel 8008 ainsi que quelques Motorola et Microchip) et peu de programmes à faire tourner. Cependant, dans les années 80, l'arrivée de l'ordinateur personnel en masse va bouleverser ce paradigme.

Il devient alors clair que l'industrie va être amenée à produire un nombre important de processeurs, de même le nombre de programmes grand public va connaître une forte augmentation, l'utilisation de l'assembleur implique qu'il faudra alors reprogrammer chacun de ces programmes pour chaque processeur sorti.

Pire encore, le besoin de performance a amené à l'augmentation du nombre d'instructions que chaque processeur supporte. Ainsi il devient de plus en plus dur pour un programmeur de connaître toutes les instructions d'un processeur.

La solution choisie pour faire face à ces problèmes fut le développement des compilateurs qui permettent de programmer un processeur avec un langage de haut niveau et de traduire ce langage vers l'ISA du processeur. Il n'y a plus besoin de coder chaque programme pour chaque nouveau modèle, et plus besoin de connaître par cœur un ISA.

Seulement les compilateurs originels ne sont pas très efficaces pour gérer les instructions CISC, en effet ces instructions prennent souvent plusieurs cycles pour s'effectuer (on parle de CPI, Clock Per Instruction, en général autour de 10 et plus pour une instruction CISC des années 80), il est donc très compliqué d'optimiser des programmes et de gérer les dépendances entre les instructions.

Des chercheurs de Berkley et Stanford (David Patterson et John Hennessey) ont alors poussé pour le développement de processeurs avec des jeux d'instructions plus petits, on parle de Reduced Instruction Set Computer (RISC). Ces processeurs sont plus performants parce que leurs instructions sont plus efficacement exploitées par les compilateurs. De plus ils sont plus petits et prennent moins de temps à designer, ce qui en pleine époque de la loi de Moore est important.

Au cours des années, l'écart de performance entre les processeurs CISC et RISC s'accroît si bien que les fabricants ont le choix entre se convertir au RISC ou disparaître, AMD et Intel trouveront une solution, traduire leurs instructions CISC en micro-instructions RISC pour rester compatible avec leurs anciens programmes, DEC n'y arrivera que trop tard et sera absorbé par Compaq en 1998 lui-même absorbé en 2002 par HP.

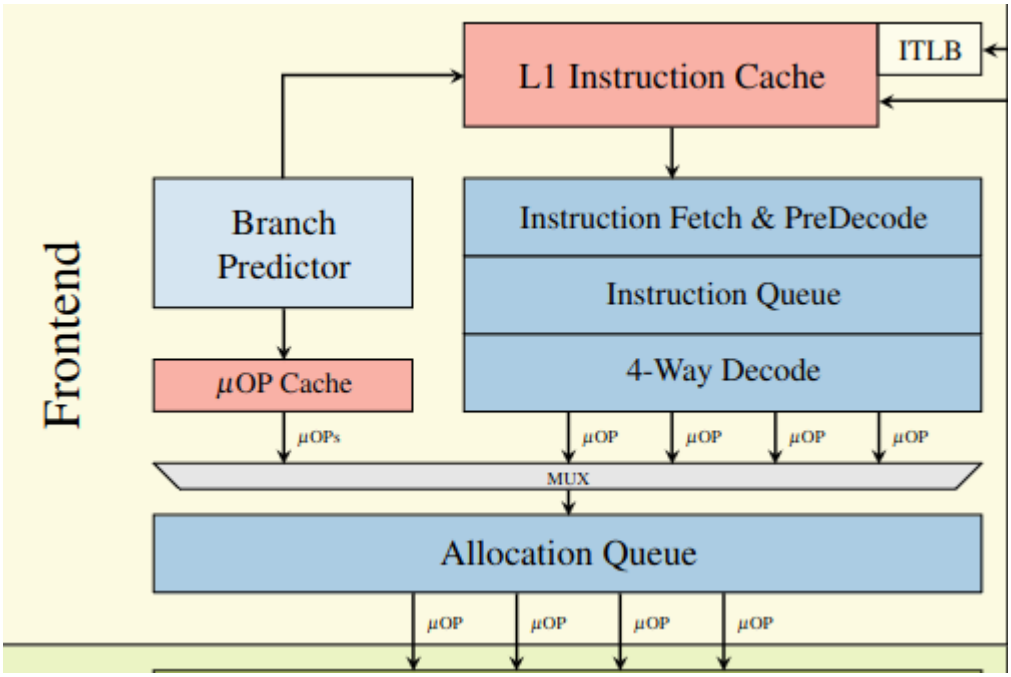


Figure 1 : Schéma bloc de la lecture et de la conversion d'instructions sur processeur x86 Skylake

On voit ici une représentation du premier étage d'un processeur Intel Skylake (2015), les instructions CISC sont lues dans un cache dédié et stockées dans une file d'attente (queue), elles sont ensuite décodées en sous-instructions RISC qui sont envoyées pour exécution aux étages de calcul plus bas.

Puis les années 2000 voient l'apparition d'un marché pour l'architecture 100% RISC : l'embarqué. En effet il y a peu de logiciels conçus pour l'embarqué, les fabricants vont pouvoir choisir leurs ISA sans avoir à se soucier de problèmes de compatibilité avec de vieux logiciels x86. Ils vont alors se tourner vers une entreprise qui vend des licences pour des design de processeurs RISC et leur compilateur : ARM. Cette structure de vente permet de séparer le design de la fabrication de processeurs, aussi elle permet à un fabricant d'obtenir des designs et d'ajouter juste ce dont il a besoin. En somme ARM permet une grande liberté de design et couvre une grande variété de besoins

qui vont de simples contrôleurs d'écran LCD, jusqu'aux cœurs A17 du dernier IPhone codesigné par Apple et ARM.

Pendant ce temps l'idée d'un ISA RISC open-source fait son chemin, et plusieurs projets sont lancés : MIPS, SPARC, SOAR, mais ils ne réussissent pas à accrocher l'industrie ni les chercheurs.

Et puis en 2010 David Patterson lance à Berkeley le projet RISC-V. Face à ARM le projet se veut simple : un ISA gratuit, simple à utiliser et à modifier pour que les chercheurs puissent tester de nouveaux designs. Mais au fur et à mesure que le projet avance, des industriels et des chercheurs de différents milieux se lancent dans des designs RISC-V. On pourra citer parmi les industriels Google puis IBM. Du côté de la recherche Lucas Benini, professeur à ETH Zurich, lance le groupe PULP, qui produira jusqu'à aujourd'hui beaucoup de design open-source allant de simples processeurs comme Snitch, jusqu'à Occamy, un processeur 432 cœurs pour l'inférence IA.

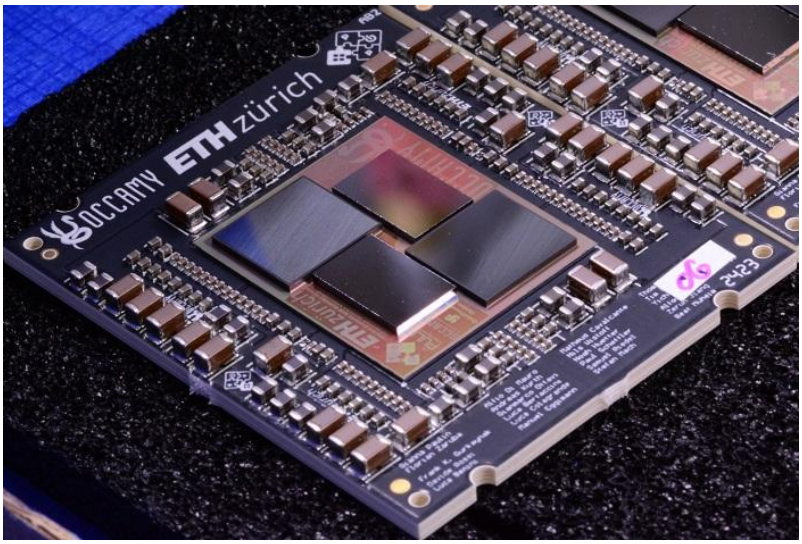


Figure 2 : Le processeur Occamy, un système sur puce 432+2 cœurs RISC-V, avec une mémoire sur puce HBM2e, une véritable prouesse technologique du groupe PULP, designée en moins d'un an par une dizaine de personnes

En 2017 David Patterson et John Hennessy reçoivent le prix Turing pour leur contribution au domaine du design de processeurs, aussi bien autour des projets RISC, et en particulier RISC-V, mais aussi pour l'influence de la publication des tomes de *Computer Architecture : A Quantitative Approach*, un livre très recommandable qui constitue la base de l'architecture des processeurs modernes.

Aujourd'hui (2025) l'architecture RISC domine des pans entiers de l'industrie.

Le succès d'ARM dans l'embarqué est tel que des tentatives apparaissent dans le domaine des serveurs et des PC, ces puces sont souvent des Systèmes Sur Puce (SOC) qui intègrent plusieurs composants, dont des accélérateurs cryptographiques, IA, GPU. et qui reposent sur une spécialisation que les processeurs x86 d'Intel ou AMD peuvent plus difficilement atteindre. Un bon exemple de ce type de percée est la série de processeur M produite par Apple.

De son côté L'architecture RISC-V a réussi à accrocher aussi bien le milieu académique que les industriels, et on voit émerger des puces dans tous les domaines, l'embarqué, le PC, le serveur. Aussi les processeurs RISC-V se sont très bien imposés dans le domaine des sous-composants, ainsi tous les GPU Nvidia modernes ont des contrôleurs RISC-V.

Ce succès est aussi fortement porté par la Chine. En effet celle-ci se trouve sujette à un embargo américain qui limite fortement son accès aux technologies de pointe, que ce soit pour la production

(ASML, TSMC) ou le design (ARM, Cadence). Les technologies OpenSource comme RISC-V sont donc une aubaine pour la Chine dans son effort dans le domaine des semi-conducteurs.

En plus de NVIDIA précédemment cité, on peut noter que le fabricant américain Microchip vend d'ores et déjà des processeurs 64 bits à cœurs RISC-V, l'anglais Raspberry Pi, un microcontrôleur PI Pico 2 avec un cœur ARM et un cœur RISC-V 32 bits, les chinois ESWIN, Kendryte et Starfive, des SoC à cœur RISC-V 64 bits et les chinois Buffalo Lab et WCH, des microcontrôleurs faible coût sur cœur RISC-V 32 bits. Ubuntu et Debian proposent des distributions Linux pour RISC-V.

Concernant le futur, il y a eu beaucoup d'annonces d'industriels sur des investissements vers RISC-V, par exemple avec l'annonce d'une future compatibilité Android sur RISC-V. Le futur nous dira si ces tentatives seront des réussites, mais il est sûr qu'à moyen terme RISC-V aura beaucoup d'opportunités de se développer.

2 - La philosophie RISC-V

Il y a donc eu plusieurs grandes enseignes de design RISC, certains privées comme ARM, d'autres publiques comme MIPS ou SPARC. RISC-V n'est donc pas le premier essai du genre, mais pourtant, on observe un engouement des industriels et du monde académique pour cette architecture. L'objectif de cette partie sera donc d'expliquer les raisons de cet intérêt.

Et à écouter David Patterson il y a plusieurs points importants qui ont rendu RISC-V aussi attirant et facile a utiliser :

2.1 - L'OpenSource

RISC-V est un projet à la base académique, il a donc été conçu pour être partagé. Le standard est accessible à tous, toutes les instructions sont publiques, les outils de compilation/simulation le sont aussi, ce qui n'est pas le cas du x86 et des ISA ARM. N'importe qui avec un ordinateur peut concevoir un design RISC-V et/ou simuler l'exécution d'un programme avec des outils gratuits, ouverts, vérifiés et fiables.

De même beaucoup de designs RISC-V sont libres de droits, vous pouvez très bien récupérer un processeur comme Ibex (produit par PULP), en faire un ASIC et le vendre dans le commerce.

2.2 - La simplicité

La base la plus petite pour un processeur RISC-V ne comporte que 40 instructions, la norme supporte des adresses sur 32 bits, elle requiert au minimum 16 registres généraux de 32 bits chacun.

La simplicité est aussi un avantage pour les industriels, parce qu'elle leur permet de changer d'ISA rapidement.

Mais ce point est aussi un avantage considérable pour l'enseignement, ainsi il est tout à fait envisageable de commencer une formation au design digital par un processeur RISC-V très simple comme Snitch. Comparé à un cours sur un processeur MIPS, le cours utilisant le RISC-V pourra se reposer sur des outils de compilation, de simulation et sur une documentation bien plus fiable et utilisée.

2.3 - La modularité

Les instructions supportées par un processeur RISC-V sont regroupées en "extensions". La grande majorité des processeurs RISC-V disponibles en ligne et utilisés dans l'enseignement sont dits RV32IMC. Ce code se décompose en trois parties :

- La **partie "RV"** qui traduit le fait que notre processeur répond à la norme RISC-V.
- Le **"32"** désigne la taille des registres et de l'espace d'adressage en bits. La norme définit aussi le RV64 et le RV128 pour un espace d'adressage 64 et 128 bits. (Il est à noter que ce sujet est assez contraint pour des raisons historiques et a fait l'objet de beaucoup de débats.)
- La **partie "IMC"** décrit les ensembles d'instructions supportées par le processeur, le "I" est la base de notre jeu d'instructions, les "M"' et "C" sont des extensions qui se rajoutent à cette base des instructions.
    - **"I"** : c'est la base de notre processeur, on y trouve la base de ce qui fait d'un processeur RISC-V une machine dite *turing complète*, c'est-à-dire une machine capable de réaliser n'importe quelle fonction calculable par une machine de Turing. Il existe une base E' qui est une version réduite de la base 'I'.
    - **"M"** : l'extension de multiplication et division des entiers, cette extension rajoute les instructions de multiplication et de division des entiers.
    - **"C"** : l'extension des instructions compressées. Là où un processeur RV32I lit des instructions sur 32 bits, un processeur RV32IC sera capable de lire des instructions de 16 bits. En effet il existe des instructions qui n'ont pas besoin de 32 bits (par exemple celles avec moins d'opérandes). L'avantage est donc que sur un programme simple, on peut quasiment diviser par deux la taille de la mémoire prise par nos instructions.

Cette extension est surtout utile dans un contexte FPGA ou embarqué où la mémoire disponible peut être faible. Si vous donnez un cours et que la mémoire des FPGA pédagogique pose problème, l'utilisation de cette extension peut vous être utile.

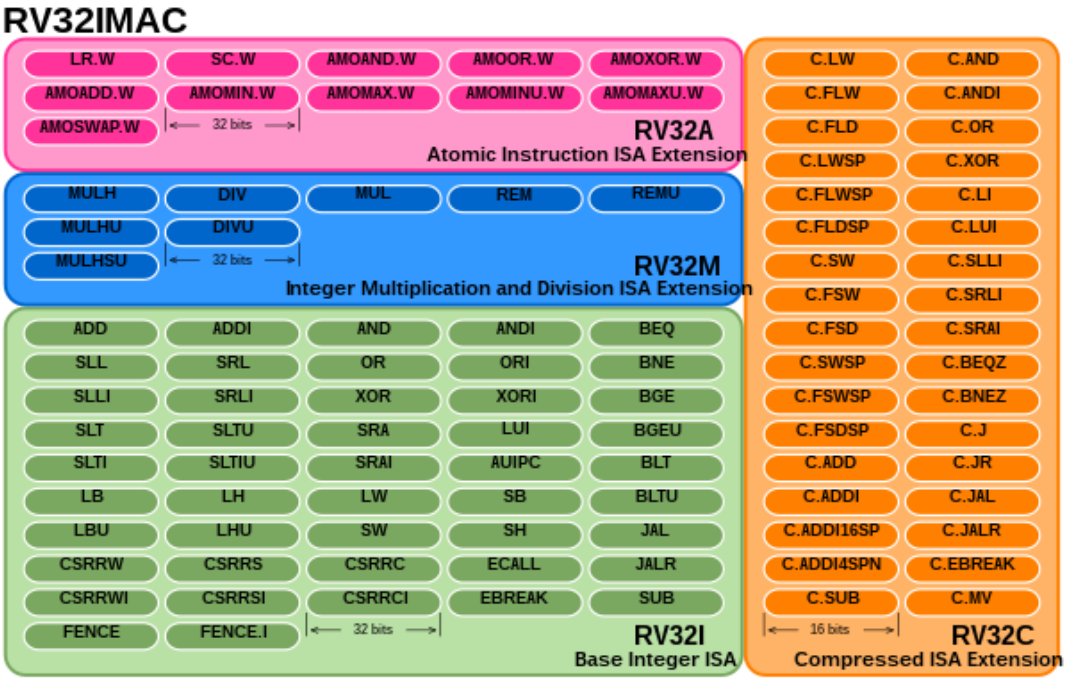


Figure 3 : Instructions supportées par un processeur RV32IMC

Il en existe aussi d'autres : F pour les flottant 32 bits, D pour les flottants 64 bits, Zicsr pour les "control and status registers", X pour des instructions non spécifiées, etc...

Pédagogiquement cette modularité peut permettre de réaliser un projet de design tout au cours d'un semestre en faisant évoluer un design en y ajoutant des modules et les extensions qui correspondent tout en continuant à bénéficier du support de la suite d'outils RISC-V.

2.4 - La stabilité

Les extensions RISC-V sont dites "gelées", en effet ces descriptions ne changent pas dans le temps. Ainsi un programmeur peut aujourd'hui compiler un projet pour un processeur RV32IMC et ce programme fonctionnera toujours sur des processeurs ayant ces extensions des années plus tard.

De même, un design RISC-V de 2014 est normalement encore fonctionnel aujourd'hui.

2.5 - La facilité à spécialiser

La norme RISC-V permet assez facilement d'ajouter des instructions spéciales aux outils de compilation et de simulation, ce qui permet de gagner de la performance très spécifiquement sur les instructions qui nous intéressent sans avoir à payer le coût d'instructions autres qui ne nous intéressent pas. On peut donc adopter une démarche CISC pour certaines applications tout en bénéficiant des avantages du RISC.

Il existe une norme pour ces instructions non spécifiées : l'extension "X". Elle est très bien documentée, et permet d'ajouter assez simplement vos nouvelles instructions au compilateur.

Un exemple de projet pédagogique connu est l'ajout d'une instruction racine carrée à un processeur RISC-V et l'ajout de cette instruction au compilateur.

2.6 - La Communauté

L'ISA RISC-V appartient aujourd'hui à la société à but non-lucratif appelée RISC-V. C'est un consortium d'académiques et d'entreprises qui prennent les décisions sur l'évolution de la norme.

Cette institution, et les engagements de ses membres sont une garantie que le projet RISC-V sera pérenne, et continuera d'être maintenu au fil des années.

2.7 - La sécurité

Au sein de ce consortium il y a plusieurs groupes de travail, le plus actif est celui de la sécurité et de la sûreté. En effet l'OpenSource a longtemps été vu comme un risque de sécurité, le secret des designs et spécifications des puces (ou autres pièces d'ingénierie) était vu comme une protection. Encore aujourd'hui des entreprises comme Nvidia, Intel, AMD et ARM utilisent le secret de leurs designs comme une protection.

Or il faut bien comprendre que cette protection n'a pas vocation à durer dans le temps. À partir du moment où un agent comprend le fonctionnement du système, il peut trouver des failles de sécurité.

Une illustration de ce genre de failles dans le monde des ISA est la découverte en 2020 d'instructions Intel secrètes qui permettent à un agent disposant d'un accès physique à la machine de modifier les programmes exécutés sur cette machine.

Cette faille n'est pas bien grave pour des serveurs, mais dans le monde de l'embarqué elle aurait été sévère.

Le fait que la norme soit OpenSource amène à ce que personne ne soit tenté d'utiliser le secret d'une instruction comme une sécurité, il a donc fallu que le consortium RISC-V fasse un travail de preuves formelles pour démontrer la sécurité de la norme. Celle-ci contient plusieurs recommandations aux designers pour assurer la sécurité de leurs designs, de même la suite d'outils RISC-V contient des outils de vérification de la sûreté et de la sécurité de designs RISC-V.

2.8 - Les défauts de l'ISA RISC-V

Les problèmes de RISC-V existent, bien que les sections précédentes puissent faire penser le contraire, par exemple il n'est pas optimal en termes de densité d'instructions.

Défaut plus important encore : la norme de calcul vectoriel RISC-V tarde à être ratifiée. Les instructions vectorielles ont été figées à date de septembre 2021, là où l'arrivée du calcul SIMD sur x86 date de 1999 ! Il faut bien reconnaître que la fondation RISC-V peut difficilement publier des normes avec la même vitesse que ses concurrents.

Le calcul SIMD réfère à des instructions qui s'appliquent sur plusieurs données en même temps, c'est une conception très efficace dans des applications très intensives en calcul parallélisable. Le calcul SIMD est devenu une nécessité pour tout un tas d'application dont l'IA, et sa réalisation la plus efficace et performante est le calcul vectoriel.

L'attente de cette norme fut donc très longue, et ceux qui voulaient travailler sur l'IA furent obligés d'utiliser des moyens détournés et moins efficaces que le calcul vectoriel : en faisant des clusters de CPU, ou en ajoutant des coprocesseurs SIMD, qui utilisent des ISA dédiées. Ces solutions sont soit moins performantes, soit doivent être programmées en assembleur. Et programmer en assembleur des fonctions pour faire de l'IA est une tâche peu enviable.

3 - Exploitation pédagogique de l'architecture RISC-V

L'ouverture de RISC-V en fait une architecture intéressante pour l'enseignement de l'architecture des processeurs et pour initier les étudiants à la conception de circuits intégrés informatiques.

MounRiver Studio (https://mounriver.com/) est un environnement de développement dédié à la programmation et au débogage des petits microcontrôleurs WCH à cœur RISC-V 32 bits. La revue Elektor a fait un essai dans son numéro de mai 2025.

Hervé Discours propose sur sa chaîne YouTube un exemple d'exploitation, basée sur son expérience en BUT GEII, de la carte Maixduino utilisant le puissant SoC Kendryte K210 à double-cœur RISC-V 64 bits : https://www.youtube.com/watch?v=C-kiOvwal0s

Pour entrer plus en profondeur dans le cœur RISC-V et lui ajouter des périphériques, il est intéressant de travailler sur son implémentation dans un FPGA, à partir d'une IP VHDL ou Verilog.

Il est possible d'utiliser la simulation. ModelSim/QuestaSim est très connu, mais il existe aussi des alternatives open source gratuites, comme Verilator (pour le Verilog et le SystemVerilog), avec des exemples d'implémentation de cœur RISC-V.

La deuxième méthode, objet de la ressource « Exemples d'implémentation d'un processeur RISC-V sur un FPGA », est l'implémentation sur FPGA, les cartes pédagogiques (Altera DE10, Digilent Basys3

par exemple) étant suffisantes pour faire tourner un cœur RISC-V. Le dépôt Git de Jacques-Olivier Klein est un autre exemple d'exploitation de l'implémentation d'un cœur RISC-V sur FPGA pour l'enseignement en BUT GEII : https://github.com/JOKleinGe1/Module_Initiation_Riscv.

La troisième méthode est de loin la plus complexe et la plus chère : la fabrication d'une puce sur silicium (ASIC). Cette méthode nécessite des suites d'outils de design très onéreux même pour le milieu éducatif (Cadence, Synopsys). Il est important de noter que là aussi des outils Opensource émergent, comme OpenLane : une suite d'outils développés par Google et basés sur des spécifications de fabrication Opensource venant de l'entreprise Skywater. Aujourd'hui ce design kit 130nm est complètement fini et utilisé dans l'industrie.

Cette ressource donne un premier aperçu de l'architecture RISC-V, ouverte et très bien documentée. Les lecteurs intéressés pourront approfondir avec les nombreux exemples d'implémentations sur FPGA dont un fourni dans ce numéro 116 de la revue 3EI [1]. Un retour sur l'usage d'un microcontrôleur basé sur RISC-V pour l'enseignement serait le bienvenu pour un prochain numéro de la revue.

Référence

[1] : Exemples d'implémentation d'un processeur RISC-V sur un FPGA, T. Ballet, juillet 2025, https://sti.eduscol.education.fr/si-ens-paris-saclay/ressources_pedagogiques/exemples-dimplementation-dun-processeur-riscv-sur-un-fpga

Exemples d'implémentation d'un processeur RISC-V sur un FPGA

La Revue 3EI
Culture Sciences de l'Ingénieur

Édité le 25/07/2025

école normale supérieure paris—saclay

Théo BALLET[1]

[1] *Doctorant au Centre de Nanosciences et NanoTechnologies (C2N), moniteur à l'ENS Paris-Saclay - DER Sciences de l'Ingénierie Électrique et Numérique*

Cette ressource fait partie du N°116 de La Revue 3EI du 3ième trimestre 2025.

Cette ressource, complément de la ressource L'émergence de l'architecture RISC-V [1] présente deux exemples d'implémentation d'un processeur RISC-V sur FPGA, en VHDL et Verilog, mettant en évidence la diversité des méthodes et langages permettant de le faire avec simplicité. Cette application pédagogique peut être menée avec des étudiants ayant déjà de bonnes connaissances en programmation des FPGA et en architecture des microprocesseurs. L'implémentation du processeur NEORV32 a été expérimentée par des étudiants de M2 (ENS Paris Saclay et M2 SETI de l'université Paris Saclay). Jacques-Olivier Klein propose une expérience similaire avec des étudiants de BUT 3 : https://github.com/JOKleinGe1/Module_Initiation_Riscv

1 - Présentation des cœurs RISC-V simples

Dans cette ressource, nous utiliserons des FPGA Xilinx (zybo z7) et Altera (DE1, DE1-SOC), cependant ces cartes sont onéreuses, elles pourront être remplacées par des cartes ICEbreaker (70 euros en Novembre 2023), ces cartes sont plus petites et reposent sur l'outil de compilation opensource Yosys. Un essai concluant d'implantation du cœur NEORV32 a aussi été réalisé sur les populaires Altera DE10 (15 euros dans le cadre des programmes éducation d'Altera) et sur une carte Nexys A7 basée sur un FPGA AMD-Xilinx Artix 7.

Les commandes d'installation et compilation présentées dans cette ressource ont été vérifiées sur une machine équipée de Linux Ubuntu 22.04 LTS.

Pour montrer la diversité et la facilité de RISC-V nous proposons ici deux processeurs venant de deux institutions différentes, avec des buts différents, et codés avec des langages différents. Cependant ces processeurs ont quelques points communs, ils sont RV32IMC ou capable de l'être avec certaines options. Ils sont relativement petits, et sont relativement anciens, ce sont donc des designs fiables, capables de rentrer sur une grande partie des FPGA pédagogiques que nous avons listés plus haut.

Le premier sera NEORV32, un processeur VHDL très simple à but pédagogique. Le deuxième sera IBEX, un processeur SystemVerilog, de qualité quasi industrielle, qui permettra aux élèves qui veulent poursuivre dans le domaine de se spécialiser.

À la fin de chaque sous-partie nous proposons des exemples de projets pédagogiques réalisables en RISC-V pour permettre aux étudiants de prendre en main ces systèmes.

Nom	CoreMark	CoreMark normalisé (1/MHz)	Taille (LUT)
Neorv small (rv32i)	33.89	0,34	1200
Neorv32 med (rv32imc)	62.50	0,63	2300
Neorv32 perf (rv32imcbu + cache)	95.23	0,95	4500
Picorv small (rv32i)	32.705	0,25	2900
Picorv32 perf (rv32im)	81.774	0,66	3100
Ibex micro (rv32ec)	113.	0,9	2195
Ibex small (rv32imc)	308.75	2,47	3200
Ibex max perf (rv32imc)	391	3,13	5300

Table 1 : Comparatif de performance sur coremark et de la taille des processeurs en Look Up Tables (LUT) : un benchmark de plusieurs instructions et tâches communes, de NEORV, PicoRV (un processeur de taille comparable codé en verilog), et Ibex, les performances en fréquence sont calculées pour un FPGA Zybo Z7-10.

On notera dans le tableau ci-dessus que malgré des tailles similaires, il y a une grande différence de performance entre des designs semi industriels comme Ibex, et des designs pédagogiques comme NEORV32.

2 - NEORV32 : VHDL

Nous connaissons l'attachement du système éducatif français pour le VHDL, nous allons donc en premier parler de NEORV32. NEORV32 n'est pas qu'un simple cœur CPU, c'est un système complet, avec des périphériques, des bloc mémoires et compatible avec plusieurs bus.

Coté bus nous avons le choix entre : l'AXI lite, wishbone, SLINK, SPI, UART, etc.

Il existe également beaucoup de périphériques optionnels : PWM, GPIO, XIP, DMA, caches.

Nous avons aussi accès à des options pour configurer le cœur RISC-V : I (jeu d'instructions minimum), M (multiplication et division d'entiers) et C (instructions de taille réduite sur 16 bits), mais aussi A pour la sécurité des accès mémoire, B pour les manipulations sur des bits précis, E pour n'utiliser que 16 registres généraux au lieu de 32, des options pour le calcul flottant, et bien d'autres options.

Tous les fichiers et documentations sont trouvables sur ce dépôt : https://github.com/ourspalois/neorv32 . Ce n'est pas le dépôt original, en effet pour simplifier ce tutorial nous avons modifié certains fichiers et ajouté d'autres qui seront grandement utiles.

L'objectif de cette partie sera d'expliquer comment arriver à un programme fonctionnel sur NEORV sur un FPGA. Pour ce faire nous allons utiliser l'approche moderne qui consiste à expliquer les choix de design hardware par des contraintes qui viennent du software, c'est-à-dire des programmes que l'on souhaite faire tourner.

2.1 - Compiler un code pour NEORV32

Nous avons un programme C/C++ que nous souhaitons faire tourner sur notre processeur. (Vous trouverez des exemples de programmes dans le dossier neorv32/sw/example du dépôt git cité).

La première étape est de convertir notre code C en assembleur RISC-V, ce travail est celui du compilateur. Votre ordinateur a déjà un compilateur (probablement x86), mais ici nous ne voulons pas compiler le code pour qu'il tourne sur le processeur de votre machine, mais sur un processeur RISC-V. Il va donc falloir télécharger un autre compilateur.

Il en existe plusieurs, les deux principaux sont clang et gcc, on choisit ici gcc par simplicité. Mais attention il faut choisir la bonne version de ce compilateur.

Pour ce faire il faut faire attention aux extensions supportées par notre processeur : en effet si on utilise un compilateur RV32IMC pour un processeur RV32I, le compilateur générera des instructions "M" et "C" que le processeur ne supportera pas. Notez que par contre il est possible d'utiliser un compilateur RV32I pour un processeur RV32IMC.

Malheureusement la seule approche fonctionnelle à ce jour reste de construire vous-même votre compilateur RISC-V, cette opération peut prendre 4h sur des machines peu puissantes.

Pour ce faire il faut installer git, et make, avec les commandes suivantes :

```
#ubuntu only $ sudo apt install gcc g++ flex bison gawk libz-dev
$ git clone https://github.com/riscv/riscv-gnu-toolchain
$ cd riscv-gnu-toolchain
$ ./configure --prefix=/opt/riscv --with-arch=rv32i --with-abi=ilp32
$ sudo make
```

Une fois le compilateur installé il faut enregistrer dans le terminal où il est situé dans votre système. Pour cela utilisez la commande : `$export PATH="/opt/riscv/bin:$PATH"`. Cette commande ne fonctionne que localement dans le terminal où vous l'avez utilisé, ce qui sera suffisant pour suivre notre tutorial. Si vous voulez continuer à utiliser le compilateur nous vous recommandons d'ajouter cette ligne à la fin du fichier ~/.bashrc .

Ici nous créons un compilateur RV32I, mais vous pouvez très bien changer pour un compilateur IMC en remplaçant l'option --with-arch=rv32imc

Pour vérifier que votre compilateur fonctionne vous pouvez utiliser la commande :

```
elf-gc$ riscv32-unknown-elf-gcc --version
riscv32-unknown c (g2ee5e430018) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

La dernière chose à télécharger sera nerov32 :

```
$ git clone https://github.com/ourspalois/neorv32
```

Par défaut NEORV contient plusieurs programmes de demo, ici nous choisissons le plus simple : faire clignoter une LED. Mais vous pouvez tout à fait en prendre d'autres. La compilation est assurée par un Makefile localisé à coté de notre code C/C++.

```
$ pushd sw/example/demo_blink_led
$ make check
$ make exe
$ popd
```

Nous avons désormais un exécutable pour programmer notre processeur, il faut maintenant choisir la façon d'envoyer ces instructions au processeur :

2.2 - Choisir le mode de programmation de NEORV32

Lorsque vous programmerez NEORV sur un FPGA (nous reviendrons sur cette étape plus tard), le processeur lira ses instructions dans une mémoire que nous définissons dans notre code RTL, il faut donc arriver à programmer cette mémoire qui sera définie sur le FPGA pour que le processeur exécute les instructions que l'on souhaite.

Vous avez alors plusieurs options :

- **Le "direct boot"** : Cette méthode demande de programmer directement l'ensemble du FPGA avec un fichier mémoire qui contient les octets qui composent les instructions de notre programme. Cette approche est simple mais elle a un grand désavantage : si vous voulez changer votre programme vous allez devoir relancer une synthèse/implémentation/programmation de votre FPGA, une opération qui peut facilement prendre 10 minutes. Dans un contexte académique cette approche peut donc facilement ralentir et frustrer des élèves.
- **Le bootloader** : Mettre sur notre FPGA un programme qui via la liaison série de la carte va fournir une interface simple pour récupérer des exécutables et les stocker sur une mémoire puis les lancer. Ce programme, comparable à un OS miniature, est appelé bootloader. Ici pour changer l'exécutable qui tourne sur notre processeur, il suffira de changer notre code, le compiler et envoyer le fichier compilé par liaison série, sans avoir à changer la configuration du FPGA.
- **Le débogueur** : Le débogueur est un outil qui permet entre autres de programmer les mémoires de notre cœur FPGA puis de lancer ou arrêter l'exécution de programmes. C'est un bloc complémentaire qui doit être synthétisé à côté du processeur.

Le choix de la méthode va influencer la description logique de notre processeur et de ces périphériques, nous devons donc choisir une configuration précise.

2.3 - Customiser notre Instance NEORV32

En effet NEORV n'est pas qu'une simple CPU, c'est un système complet, avec des mémoires et des périphériques. Et le choix des périphériques que l'on peut/doit utiliser dépend du mode de programmation choisi. Vous trouverez des configurations simples de chacun de ces modes de programmation dans le dossier **rtl/test_setups** du dépôt git.

On peut aller regarder de plus près ces fichiers VHDL.

Ici le fichier le plus simple : neorv32_test_setup_approm.vhd

```
neorv32_top_inst: neorv32_top
generic map (
   -- General --
   CLOCK_FREQUENCY            => CLOCK_FREQUENCY,
   INT_BOOTLOADER_EN          => false,
   -- RISC-V CPU Extensions --
   CPU_EXTENSION_RISCV_C      => true,
   CPU_EXTENSION_RISCV_M      => true,
   CPU_EXTENSION_RISCV_Zicntr => true,
   -- Internal Instruction memory --
   MEM_INT_IMEM_EN            => true,
   MEM_INT_IMEM_SIZE          => MEM_INT_IMEM_SIZE,
   -- Internal Data memory --
   MEM_INT_DMEM_EN            => true,
   MEM_INT_DMEM_SIZE          => MEM_INT_DMEM_SIZE,
   -- Processor peripherals --
   IO_GPIO_NUM                => 8,
   IO_MTIME_EN                => true
 )
```

On y définit les modules de NEORV que l'on veut installer, vous pouvez voir la liste des paramètres de NEORV dans le fichier **rtl/core/top.vhd**. Ainsi, dans ce fichier, on appelle un processeur RV32IMC, sans débogueur, et avec 8 paires de ports GPIO mais il existe aussi des fichiers plus lourds utilisant des fonctionnalités plus complexes de NEORV.

2.4 - Programmer notre FPGA

Nous montrons ici les étapes pour faire tourner NEORV32 sur un FPGA Xilinx zybo z7 en utilisant un bootloader (on utilise donc le fichier **rtl/test_setups/neorv32_test_setup_bootloader.vhd**). Ce processeur est compatible avec tous les modèles de FPGA, cependant face à la dégradation récente des services FPGA de Intel nous vous recommandons d'utiliser des modèles Xilinx.

La première étape est d'installer Vivado ou Quartus, attention sur les cartes digilent il est nécessaire d'importer les fichiers de description de votre carte FPGA dans Vivado et sur linux il vous faudra aussi suivre une procédure pour installer des drivers USB. Vous trouverez ces deux procédures sur ce tutoriel : https://digilent.com/reference/programmable-logic/guides/installing-vivado-and-vitis

Les tests de cette ressource ont été faits sur Vivado 2024.1, nous vous incitons à utiliser une version récente de Vivado pour maximiser les similarités avec nos écrits.

La première étape est de créer un projet Vivado, lui choisir un nom, un emplacement et renseigner le type de projet, ici "RTL project", vous pouvez cocher la case disant qu'on ne souhaite pas importer

maintenant de fichiers RTL. Il faut aussi spécifier le FPGA sur lequel vous voulez faire fonctionner votre design, onglet "boards", sélectionnez uniquement les cartes digilent, vous devriez voir votre FPGA, dans notre cas une carte DIGILENT ZYBO Z7-10. Si vous n'avez pas de carte digilent dans la liste déroulante, il faut refaire l'étape d'installation des board files deux paragraphes plus haut.

Une fois le projet créé, nous importons nos fichiers RTL : utilisez l'option "add sources" dans l'onglet **File** de Vivado. Il faudra alors importer un **dossier** et non pas des fichiers seuls, vous devrez alors renseigner le fichier **neorv32/rtl/core** et indiquer une librairie "neorv32". Il est très important de cocher l'utilisation de source de sous dossiers. Votre écran devra être similaire à la vue ci-dessous.

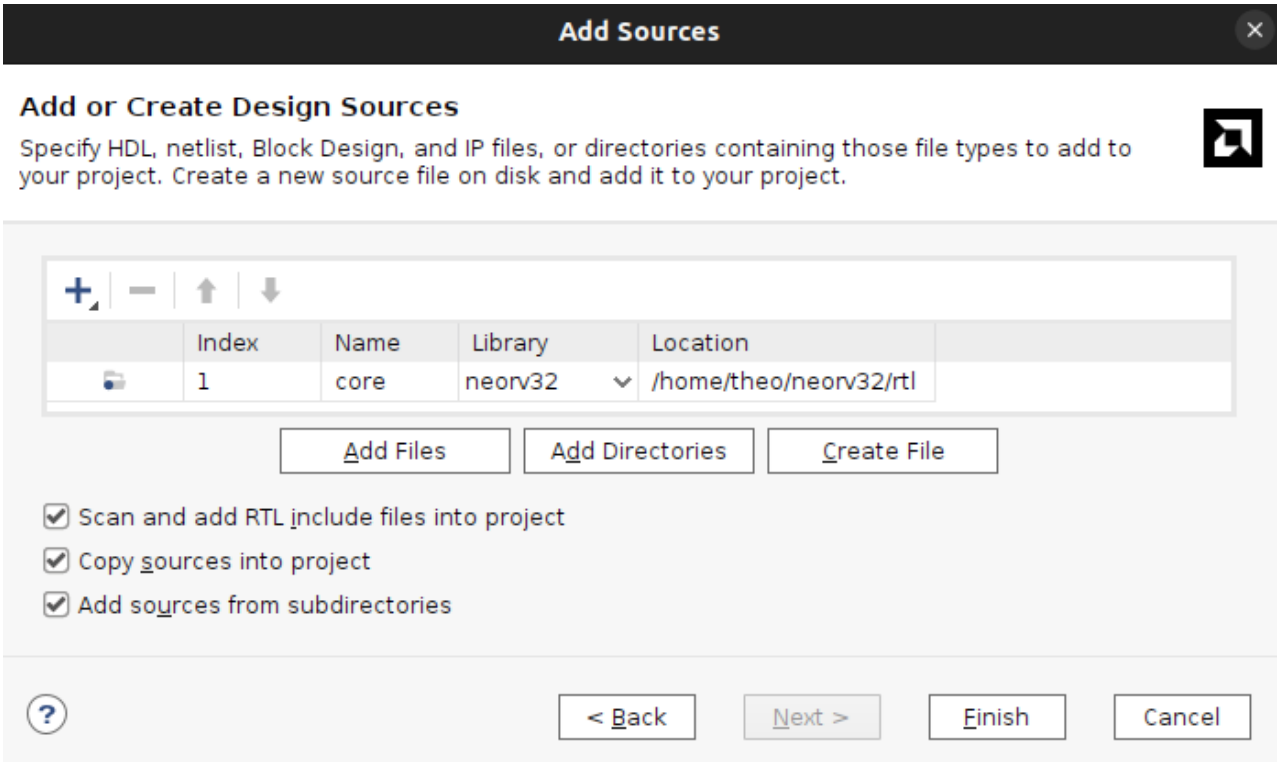
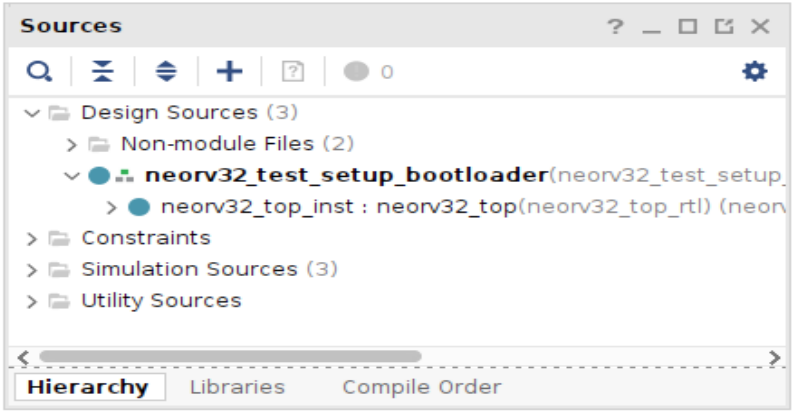


Figure 4 : Capture d'écran de l'importation des sources sur Vivado

Il faut aussi importer le fichier **rtl/test_setups/neorv32_test_setup_bootloader.vhd**. Cette fois sans plus de complexité, il faut laisser la librairie par default.

Une fois cette importation faites il reste à dire à Vivado quel est le plus haut module de notre design, le module le plus haut est appelé "top module". Théoriquement le logiciel peut le sélectionner seul mais il est toujours mieux de le faire à la main pour être sûr. Pour ce faire, utilisez un clic droit sur le fichier du module en question puis "Set as top".

Vous aurez alors la liste de sources suivante :



Dernière étape, ajuster les paramètres de notre système à notre carte : nous voyons que le haut du fichier **neorv32_test_setup_bootloader.vhd** définit des paramètres :

```
generic (
   -- adapt these for your setup --
   CLOCK_FREQUENCY   : natural := 125000000;
   MEM_INT_IMEM_SIZE : natural := 16*1024;
   MEM_INT_DMEM_SIZE : natural := 8*1024
 );
```

Le premier doit être remplacé par la fréquence de l'horloge d'entrée de notre FPGA (ou d'une PLL si on choisit de changer la fréquence de notre système). Ici notre zybo z7-10 a une fréquence d'entrée de 125MHz.

De même il faut vérifier que nos deux mémoires ne dépassent pas la capacité SRAM de notre FPGA, ici nous réservons 24KB, zybo z7-10 en a 270KB.

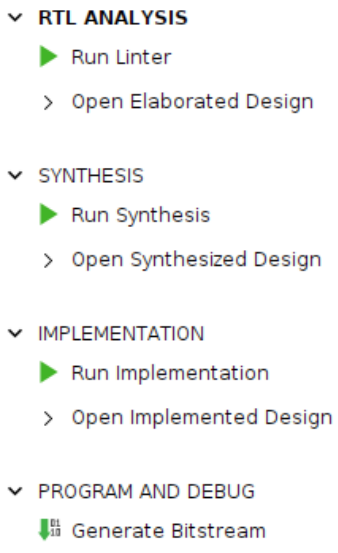
Ces paramètres permettent à notre code RTL de fonctionner correctement, par exemple la liaison série doit savoir quelle est la période de l'horloge pour mesurer la durée des signaux qu'elle reçoit. Maintenant il nous faut indiquer à Vivado la fréquence de l'horloge de notre design, aussi dans le même fichier nous précisons quel port de notre code est connecté à quelle broche du FPGA.

Pour ce faire nous importons un fichier déjà fait, via **add sources** puis "add or create constraints" nous importons ensuite le fichier **neorv32/constraints/zybo_z7_10.xdc**. Comme son nom l'indique ce fichier définit des contraintes uniquement valables pour les cartes ZYBO Z7 10. Pour le porter sur un autre FPGA il faut modifier la fréquence de l'horloge indiquée ainsi que changer les ports auxquels sont attribués des signaux.

Nous avons donc :

- Importé tous nos fichiers de description de NEORV.
- Dit à quels led/interupteurs devaient être connectés chacun des signaux.
- Défini la fréquence d'horloge de notre design.

Les deux étapes suivantes sont la compilation Vivado, et la programmation du FPGA. Sans rentrer dans le détail, la compilation Vivado (et quartus) se fait en 4 étapes visibles dans l'interfaces Vivado :



1. **RTL ANALYSIS** : vérifie qu'il n'y a pas d'erreur de syntaxe ou de logique dans notre code RTL

2. **SYNTHESIS** : compile notre code RTL en une suite de différents éléments logiques présents sur le FPGA, on parle de Look Up Table (LUT), cellules mémoires (SRAM), etc.

Une fois cette étape finie vous pourrez voir dans l'onglet "reports" un fichier "utilization" qui contient le nombre d'éléments logiques de votre FPGA utilisés par votre design.

3. **IMPLEMENTATION** : l'étape précédente nous fournit un graphe des éléments logiques utilisés par notre design et des liens logiques entre eux, seulement sur le FPGA ces composants sont fixes, le placement routage va prendre ces éléments et leur associer une position sur le FPGA puis tracer les connections entre eux.

Une fois cette étape finie vous pourrez voir la cartographie du design sur votre FPGA, comme présenté sur la figure 5 ci-dessous.

4. **PROGRAM** : le bitstream est un fichier (en général très lourd, plusieurs MB) qui décrit comment chaque élément du FPGA est configuré, son format est secret et est une grande raison du quasimonopole de Xilinx et Altera sur le marché.

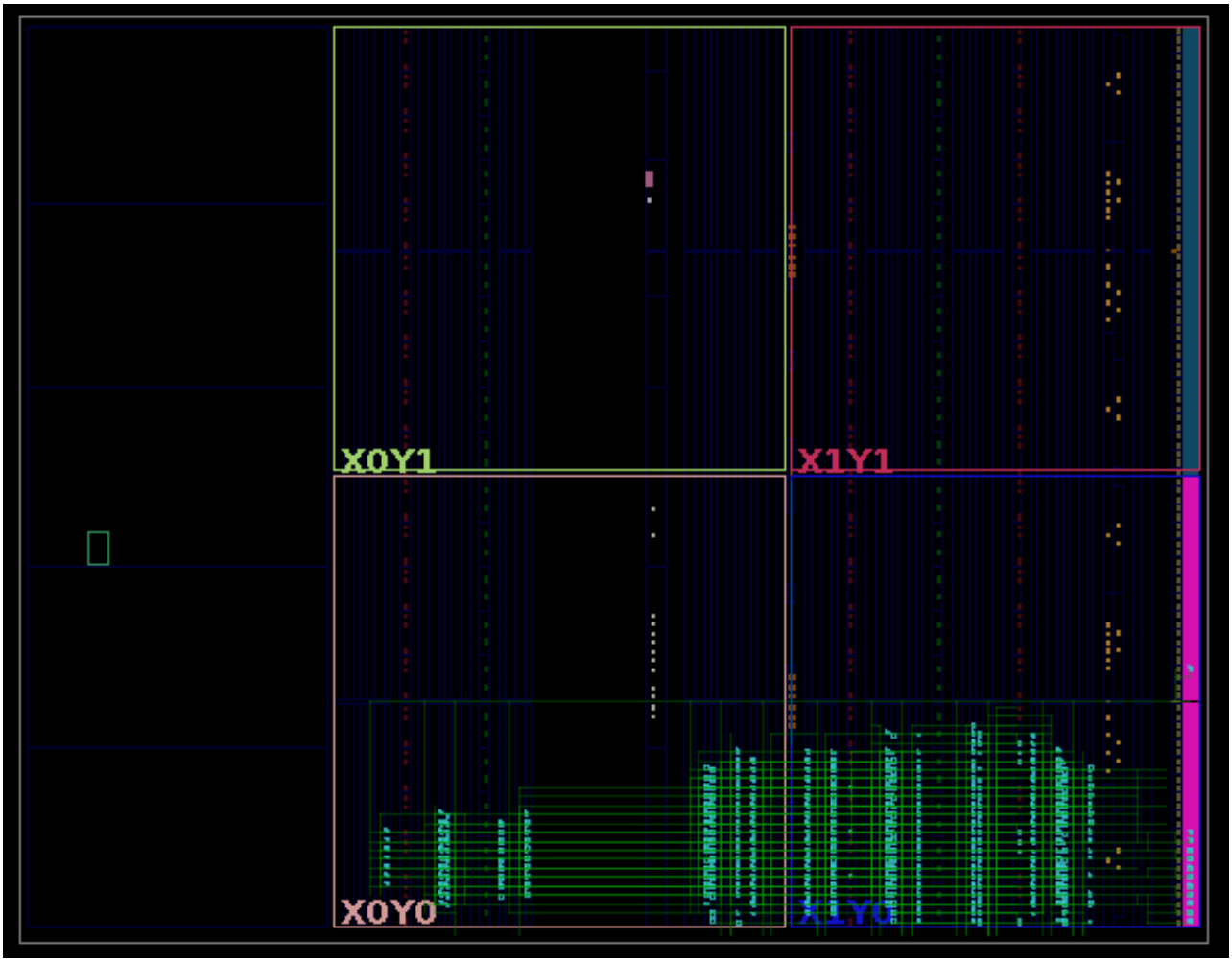


Figure 5 : Utilisation de ressources de neorv32 sur un Zybo z7-10. Les grands carrés que vous voyez sont ce qu'on appelle de domaine d'horloge, en effet les FPGA ont tendance à être des puces assez grandes si bien que le temps de propagation de l'horloge de part et d'autre de la puce devient si important qu'il faut la scinder.

2.5 - Lancer l'exécution

Cette étape dépend du type de mode de programmation choisi. On présente ici la démarche avec un bootloader. Pour un direct boot, le code ayant été chargé dans la mémoire lors de la

programmation du FPGA, il suffirait de redémarrer le processeur avec le signal **rstn_i** que nous avons connecté à un bouton.

Avec le bootloader, méthode choisie pour cet exemple, il faut vous connecter en liaison série avec un terminal adapté, nous vous conseillons teraterm sur Windows, cutecom sur linux.

Il vous faut donc un moyen de connecter le port USB de votre ordinateur aux deux ports UART de votre FPGA, vous pouvez trouver des adaptateurs USB-série sur le net sans difficulté.

Vous pouvez modifier les paramètres UART mais par défaut ils devraient être : baud_rate = 115200, data = 8bits, stop = 1bit, return =\\ r\\ n.

Une fois votre connexion établie, appuyez sur le bouton que nous avions désigné comme reset, celui-ci va relancer le processeur, vous devrez alors voir sur votre terminal le texte suivant :

```
<< NEORV32 Bootloader >>
 BLDV: Mar  7 2023
 HWV:  0x01080107
 CID:  0x00000000
 CLK:  0x05f5e100
 MISA: 0x40901106
 XISA: 0xc0000fab
 SOC:  0xffff402f
 IMEM: 0x00008000
 DMEM: 0x00002000

 Autoboot in 10s. Press any key to abort.
```

Appuyez sur une touche pour interrompre le programme, il vous fournira alors un menu :

```
Available CMDs:
  h: Help
  r: Restart
  u: Upload
  s: Store to flash
  l: Load from flash
  x: Boot from flash (XIP)
  e: Execute
 CMD:>
```

Pour uploader un programme, utilisez l'option "u", puis envoyez le binaire de votre programme via l'interface de votre terminal série. Le binaire est un fichier qui contient bit à bit le contenu de la mémoire programme, lors de l'envoi de ce fichier il faut faire attention à n'envoyer que le contenu du fichier et pas d'autres symboles, c'est une option activable sur cutecom et teraterm. Une fois le fichier envoyé le bootloader va le stocker en mémoire.

Vous pouvez ensuite lancer l'exécution avec l'option "e".

Félicitations vous avez un code qui tourne sur votre processeur embarqué sur FPGA ! ^^

2.6 - Projets pédagogiques utilisant NEORV32

Voici quelques idées de projets pour pousser plus loin et découvrir le monde de l'intégration système.

Idées de projets :

Difficulté faible : (pas besoin de coder son propre RTL)

- Utiliser le débogueur de NEORV32
- Mesurer l'impact des instructions M en termes de performance
- Mesurer l'impact des instructions C

Difficulté moyenne : (pas besoin de concevoir une architecture, conception très locale)

- Coder un timer programmable en VHDL et le connecter au processeur via le bus I2C, WHISHBONE, etc.
- Coder un accélérateur pour le calcul de certaines fonctions complexes (racine carrée, hash de crypto) et l'installer sur le bus externe de NEORV

Difficulté importante :

- Implémenter un accélérateur de calcul de certaines fonctions via l'interface X du cœur NEORV32.
- Utiliser le bus wishbone/AxiLite pour connecter deux instances de NEORV et faire un multicœur.

Exploratoire :

- Utiliser Litex pour générer une configuration multicœur de NEORV capable de faire tourner Linux.
- Concevoir un design ASIC de NEORV32.

3 - IBEX : SystemVerilog

Ibex est un processeur moderne développé par PULP sous la direction de Lucas Benini à Zurich.

Ibex est plus complexe que notre précédent processeur, cette complexité va lui permettre des performances très supérieures comme l'indique la Table 1.

Ibex dispose d'un pipeline 2 ou 3 étages nettement plus optimisé que celui de NEORV, deux bus données et instructions séparés, des options pour de vrais caches instructions et données, des modules de protection mémoire.

Il est aussi intéressant de noter que Ibex est à la base du projet OpenTitan, un projet de fabrication d'un processeur ayant une sûreté maximale, à ce titre il a fait l'objet d'une vérification formelle et fonctionnelle très poussée.

La particularité d'Ibex est aussi la modernité de ses outils de développements, Ibex utilise des gestionnaires de base de données de code pour se construire et s'interfacer avec des outils intégrés

EDA (Electronics Design Automation), de même Ibex est codé en System-Verilog 1800, une norme de 2017 qui est particulièrement nouvelle. Cette norme n'est pas supportée par Quartus (à l'exception de la version Pro, très onéreuse et peu fonctionnelle).

3.1 - Ibex-demo-system

Nous allons ici implémenter un Ibex-demo-system, c'est un microcontrôleur avec ibex en son centre, et des périphériques pour le rendre utilisable.

Vous trouverez les fichiers sur ce dépôt https://github.com/ourspalois/ibex-demo-system. Ce n'est pas le dépôt github officiel, mais un fork où nous avons corrigé un bug lié au compilateur et à ses options, ainsi que des problèmes liés à l'interface JTAG pour limiter le nombre de câbles que nous utilisons.

La particularité de Ibex est d'être développé avec des standards de programmation beaucoup plus modernes que NEORV, ainsi Ibex est composé de centaines de fichiers RTL structurés dans une arborescence complexe, là où dans NEORV tous les fichiers RTL étaient contenus dans le même dossier. Pour s'y retrouver, on utilise donc fusesoc, un outil opensource qui va gérer ces fichiers, et nous permettre de lancer des projets avec Vivado, Quartus et autres EDA.

3.2 - Utilisation de Fusesoc

Tout d'abord nous téléchargeons les fichiers de description d'Ibex-demo-system :

```
$ git clone git@github.com:lowRISC/ibex-demo-system.git
$ cd ibex-demo-system
```

Fusesoc est une librairie python, idéalement vous devriez utiliser un environnement virtuel pour l'installer :

```
$ python3 -m venv .venv
$ source .venv/bin/activate
$ pip3 install -r python-requirements.txt
```

Ici nous utilisons le module venv de python, mais vous pouvez aussi utiliser conda.

3.3 - Compiler nos programmes

Attention pour Ibex il vous faut recompiler et installer votre compilateur RISC-V vers les extensions RV32i_zicsr, une extension avec des registres de contrôle en plus. Pour ce faire il faut sur Ubuntu installer les paquets libmpc-dev, libexpat1-dev.

```
$ mkdir sw/c/build
$ pushd sw/c/build
$ cmake ..
$ make
$ popd
```

On crée un dossier pour contenir les sorties de notre compilateur, puis on lance la compilation avec l'outil Cmake, il faut aussi installer srecord (un paquet apt sur linux, qui permet de modifier des fichiers mémoire EEPROM).

Vous noterez que pour aller plus vite, sur une machine avec N cœurs cpu vous pouvez utiliser make -j N pour lancer la compilation en parallélisant sur tout le matériel disponible.

3.4 - Lancer une simulation

Pour simuler le fonctionnement de notre processeur on utilisera Verilator, un simulateur OpenSource, vous pouvez trouver les instructions pour l'installer sur ce ce tutoriel https://verilator.org/guide/latest/install.html (attention, l'installation par paquet risque de fournir une version trop vieille, il faudra alors installer et compiler le logiciel vous-même.)

Verilator est un trans-compilateur, il va compiler notre code RTL en un exécutable C/C++ qui pourra ensuite être exécuté sur notre machine.

On a donc deux étapes :

- La compilation
- L'exécution

Nous utilisons fusesoc pour contrôler Verilator, avec ces instructions :

```
$ fusesoc --cores-root=. run --target=sim --setup --build lowrisc:ibex:demo_system
```

À ce point il est possible que vous ayez des erreurs pour des libraires manquantes, la solution simple est de trouver les paquets qui leur correspondent sur votre distribution et de les installer via votre installateur de paquets. Sur Ubuntu il faudra installer le paquet libelf-dev.

Cette commande est importante à comprendre, et contient deux passages importants :

- **lowrisc:ibex:demo_system** désigne l'entité fusesoc que l'on souhaite compiler.

  Pour faire court, vous pouvez avoir différents fichiers **.core** qui décrivent des entités/composants RTL différents (un pour un module UART, un autre pour la ram, etc.), et l'entité fusesoc correspond à leurs noms. Quand nous appelons l'entité **lowrisc:ibex:demo_system** on demande en fait à fosesoc de réunir tous les fichiers de cette entité et ceux dont elle dépend.

- **target=sim** indique à fusesoc quelle est la cible de la compilation, ici la cible nommée ‘sim’.

  Cette cible est définie dans le fichier **ibex_demo_system.core** :

```
sim:
  <<: *default_target
  default_tool: verilator
  filesets_append:
    - files_verilator
  toplevel: top_verilator
  tools:
    verilator:
      mode: cc
      verilator_options:
        # Disabling tracing reduces compile times but doesn't have a
        # huge influence on runtime performance.
        - '--trace'
```

```
        - '--trace-fst' # this requires -DVM_TRACE_FMT_FST in CFLAGS below!
        - '--trace-structs'
        - '--trace-params'
        - '--trace-max-array 1024'
        - '-CFLAGS "-std=c++14 -Wall -DVM_TRACE_FMT_FST -DTOPLEVEL_NAME=top_verilator"'
        - '-LDFLAGS "-pthread -lutil -lelf"'
        - "-Wall"
        - "-Wwarn-IMPERFECTSCH"
        # RAM primitives wider than 64bit (required for ECC) fail to build in
        # Verilator without increasing the unroll count (see Verilator#1266)
        - "--unroll-count 72"
    parameters:
- PRIM_DEFAULT_IMPL=prim_pkg::ImplGeneric
```

On y définit l'outil à utiliser, ici Verilator, et ses options (taille des traces, type d'erreurs qui seront remontées, etc.)

On a donc demandé à fusesoc de réunir nos fichiers RTL, puis de les interfacer avec verilator, et de lancer la compilation de nos fichier RTL en un exécutable qui réalise une simulation de notre processeur. Il ne reste donc qu'à lancer la simulation avec son exécutable :

```
$./build/lowrisc_ibex_demo_system_0/sim-verilator/Vtop_verilator --meminit=ram,./sw/c/build/demo/hello_world/demo
```

C'est en fait un appel à notre exécutable, qui avec une option donne des valeurs pour initialiser le contenu de la mémoire de notre système.

Cette simulation va créer un port série sur votre ordinateur, vous pourrez alors interagir avec en vous y connectant via votre terminal préféré. (Vous verrez le nom du port dans la sortie du simulateur).

3.5 - Lancer une compilation vers un FPGA

Maintenant pour compiler votre programme et le lancer sur FPGA l'idée est la même que pour la simulation, à la différence que la cible de fusesoc va changer de Verilator vers Vivado.

```
$ fusesoc --cores-root=. run --target=synth_zybo_z7_10 --setup --build lowrisc:ibex:demo_system
```

Il faudra utiliser cette commande, qui comme pour Verilator construira un projet, puis l'interfacera avec Vivado. Celle-ci appelle une autre cible fusesoc qui définit les options d'appels de Vivado, notamment le numéro de série du FPGA cible, ici un Arty-7.

```
synth_zybo_z7_10:
    <<: *default_target
    default_tool: vivado
    filesets_append:
      - files_xilinx
      - files_constraints_z7_10
    toplevel: top_zyboz7_10
```

```
tools:
  vivado:
    part: "xc7z010clg400-1"  # zyboZ7-10 part number
parameters:
  - SRAMInitFile
  - PRIM_DEFAULT_IMPL=prim_pkg::ImplXilinx
flags:
  use_bscane_tap: true
```

Ce code va lancer une compilation puis un placement routage sur le FPGA, et générer un fichier projet en **.xpr** . Vous pouvez l'ouvrir avec Vivado pour voir vos rapports de compilation et observer l'état de votre projet.

À ce point fusesoc aura lancé la synthèse et le placement routage de Vivado, il reste juste à programmer votre FPGA, pour ce faire vous pouvez soit utiliser une autre commande, mais dans un contexte éducatif nous vous conseillons d'utiliser l'interface graphique de Vivado. Pour ce faire vous pouvez utiliser la commande de compilation fusesoc, puis utiliser vivado pour ouvrir le fichier **.xpr** qui vous montrera le projet comme n'importe quel autre, vous verrez que le bitstream est déjà généré, il restera à sélectionner votre FPGA et à le programmer.

Le FPGA est maintenant programmé ! Cependant nous n'avons pas encore chargé de programmes en mémoire, pour ce faire nous allons utiliser l'interface de débogage de notre processeur.

Nous utiliserons un debugger, ici OpenOCD car opensource et simple d'utilisation. Le principe du débogueur est qu'il nous permet de lire et d'écrire dans les mémoires présentes sur notre bus mémoire, dans les registres de notre processeur, on pourra aussi arrêter l'exécution de programmes. Ce port de debug fonctionne via la liaison série qu'on utilise pour programmer le FPGA, en fait on utilise une interface Xilinx qui va intégrer le cœur RISCV dans une chaîne JTAG avec notre FPGA. C'est un tour de magie puisque aucun port JTAG n'apparaît dans notre top module, mais c'est un tour qui fonctionne très bien.

```
$ ./util/load_demo_system.sh run ./sw/c/build/demo/blink_led/blink_demo
```

Vous verrez alors les leds de votre processeur s'allumer, c'est que le processeur peut faire tourner le programme, vous pourrez alors en changeant d'exécutable lancer d'autres programmes de demo : un hello world en série par exemple : **./sw/c/build/demo/hello_world/demo**.

3.6 - Projets pédagogiques avec Ibex

Ibex est massivement plus complexe que NEORV32, nous ne recommandons pas de l'aborder avant le master ou la troisième année de BUT/DUT.

Mais cette complexité vient avec des avantages, notamment en termes de performance hardware, mais aussi d'efficacité à développer. Aussi pour des élèves particulièrement intéressés par le développement hardware, il sera important de maîtriser ses modes de fonctionnement et outils car ce sont ceux de l'industrie.

Idées de projets :

Difficulté faible : (pas besoin de coder son propre RTL)

- Utiliser le port SPI pour contrôler un écran LCD.
- Mesurer l'impact des instructions M, C, et E en termes de performance.

Difficulté moyenne : (pas besoin de concevoir une architecture, conception très locale)

- Coder un timer programmable en SystemVerilog et le connecter au processeur via le bus SPI.
- Coder un accélérateur pour le calcul de certaines fonctions complexes (racine carrée, hash de crypto) et l'installer sur le bus SPI.

Difficulté importante :

- Implémenter un accélérateur de calcul de certaines fonctions dans le pipeline de Ibex, ou via son interface X.
- Changer Ibex pour un cœur RISC-V plus grand comme le CVA5 de Openhwgroup.

Exploratoire :

- Chercher à implémenter un accélérateur vectoriel sur Ibex.
- Concevoir un design ASIC de Ibex, avec des outils OpenSource comme OpenLane.

Référence :

[1] : L'émergence de l'architecture RISC-V, T. Ballet, A. Juton, Juillet 2025, https://sti.eduscol.education.fr/si-ens-paris-saclay/ressources_pedagogiques/lemergence-de-larchitecture-risc-v

Ressource publiée sur Culture Sciences de l'Ingénieur : https://sti.eduscol.education.fr/si-ens-paris-saclay