



HEIRV32 (HEIRV32)

Cours Architecture de l'ordinateur (CAr)



Orientation: Informatique et systèmes de communication (ISC)
Spécialisation: Data Engineering (DE)
Cours: Architecture de l'ordinateur (CAr)
Auteurs: Silvan Zahno, Axel Amand
Date: 28.10.2025
Version: v4.0



Contenu

1	Introduction	3
1.1	Microprocesseur HEIRV32 multi-cycle	4
1.1.1	Reference Documents	4
2	Spécifications	5
2.1	Fonctions	5
2.2	Fonctions supplémentaires	6
2.2.1	Facile	6
2.2.2	Moyen	6
2.2.3	Difficile	6
2.3	Projet HDL-Designer	7
2.4	Blocs fournis	8
2.5	Indications sur la carte fille	9
2.6	Simulateur	9
3	Composants	10
3.1	Carte Field Programmable Gate Array (FPGA) EBS	10
3.2	Boutons et Light Emitting Diodes (LEDs)	10
3.3	LEDs supplémentaires	11
3.4	Boards optionnelles	11
4	Evaluation	12
5	Guide	13
5.1	Architecture générale	13
5.1.1	Signal en	13
5.1.2	Instruction lw	13
5.1.3	Instruction sw	16
5.1.4	Instruction Type R - I	16
5.1.5	Instruction beq	16
5.1.6	Instruction jal	16
5.1.7	Instruction jalr	17
5.2	Control Unit	18
5.2.1	Main FSM	19
5.2.2	ALU Decoder	19
5.2.3	Instr. Decoder	20
6	Tests	21
6.1	Simulation	21
6.1.1	Compréhension	21
6.1.2	Automatisation des tests	21
6.2	Code	22
6.2.1	Contrôle par équivalence	22
6.2.2	Code personnalisé	23
6.3	Tips	24
	Glossaire	25

1 | Introduction

L'objectif du projet est d'appliquer directement les connaissances acquises à la fin du semestre à l'aide d'un exemple pratique. Il s'agit de créer un processeur **5-stages Reduced Instruction Set Computer architecture, open-sourced (RISCV)** réduit pour exécuter un petit programme assembleur. Le processeur est d'abord simulé puis déployé sur une **FPGA**. La connexion avec le monde extérieur peut se faire à l'aide de boutons et de leds. Ce système de processeur est représenté dans la [Figure 1](#).

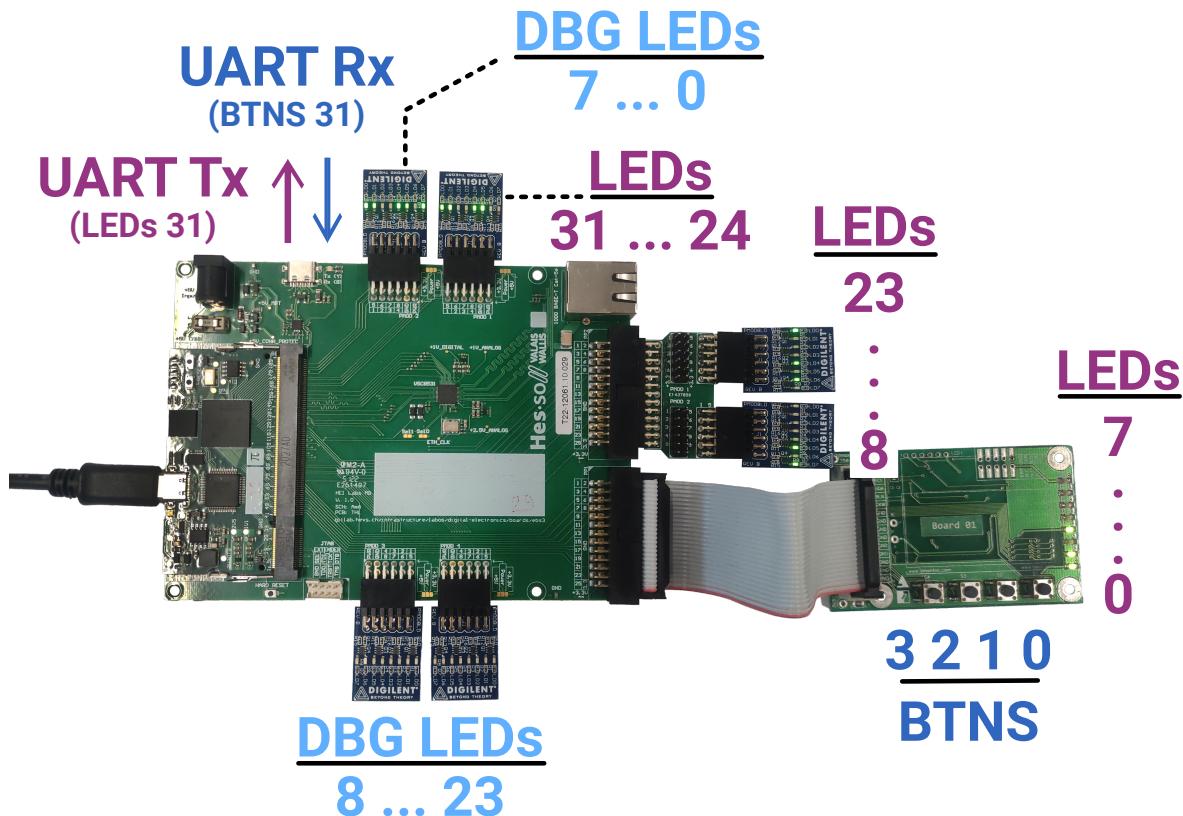


Figure 1 - Équipement du système (EBS3)

Le but est de réaliser les Specification minimales définies au [Chapitre 2](#).



1.1 Microprocesseur HEIRV32 multi-cycle

L'architecture single-cycle, complétée lors d'un laboratoire précédent, permet une approche simple de l'architecture **RISC-V**. Elle comporte toutefois plusieurs problèmes :

- la mémoire du programme et des données sont séparées
 - nécessite d'implémenter deux puces pour un seul programme
- les mémoires sont accédées de façon asynchrone
 - les vitesses maximales sont plus limitées que pour des mémoires synchrones en cas d'accès continu
 - le contrôle est plus complexe, nécessitant des timings précis, et les puces sont sujettes à des situations de concurrence
- la vitesse du processeur est limitée par l'instruction la plus longue
 - les seules améliorations possibles n'étant plus que l'évolution des technologies de transistor ainsi que la réduction des longueurs de rootage

Fondamentalement, les instructions se décomposent en 5 étapes :

- *Fetch* : l'instruction est récupérée de la mémoire
- *Decode* : l'instruction est décodée : **funct3**, **funct7**, réglages de l'ALU et des sources ...
- *Execute* : l'opération spécifiée est exécutée
- *Memory* : accède à des données spécifiques de la mémoire (facultatif)
- *Writeback* : enregistre des données dans la mémoire (facultatif)

Toutes les instructions ne nécessitant pas l'exécution de chaque étape, ces dernières peuvent être dissociées, séparées par un coup de clock \Rightarrow multi-cycle. Ainsi, la fréquence d'horloge peut être augmentée, la vitesse étant désormais limitée par l'étape la plus lente.

Les instructions les plus courtes ne nécessitant que 3 à 4 étapes bénéficieront d'un traitement plus rapide.

Les mémoires utilisées sont maintenant synchrones.



Cette architecture ouvre aussi la voie à l'implémentation d'un **système de pipeline** (c.-à-d. charger une instruction à chaque coup de clock), de **prédition de branchements**, et toute autre technique permettant d'accélérer le fonctionnement général du processeur.

Seul le multi-cycle est abordé ici.

1.1.1 Reference Documents

[1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21],
[22], [23], [24], [25], [26], [27], [28], [29]



2 | Spécifications

2.1 Fonctions

Les fonctions de base sont définies comme suit :

- La **FPGA** implémente un microprocesseur **RISC-V** 32-bits, multi-cycle, dont le fonctionnement est confirmé par simulation puis par déploiement sur une puce physique
- Le microprocesseur implémenté doit supporter les instructions suivantes :
 - Les instructions de type R : **add, sub, and, or, slt, xor, sll, srl**
 - Les instructions de type I : **addi, andi, ori, slti, xori, slli, srli**
 - Les instructions Mémoire : **lw, sw**
 - Les instructions de Saut: **beq, jal, jalr**
- En utilisant les commandes supportées, un code assembleur est écrit et testé, capable de:
 - détecter l'appui des boutons de la carte électronique boutons-LEDs [Figure 5](#)
 - contrôler les leds de la carte électronique boutons-LEDs [Figure 5](#)
 - combiner les deux fonctionnalités précédentes pour démontrer un comportement interactif
 - remplir les fonctionnalités d'un code simple (voir [Chapitre 2.2.1](#))

Pour contrôler les **LED**, il suffit d'écrire le registre **x30**.

Pour lire les boutons, il suffit de lire le registre **x31**. Une écriture sur ce registre ne modifie pas sa valeur.



Figure 2 - **RISC-V** Circuit matériel

Une architecture fonctionnelle, un code « facile » (voir [Chapitre 2.2.1](#)) et un rapport parfait valent une note maximale de



5

Un ou plusieurs codes plus évolués sont nécessaires pour obtenir une note supérieure.

2.2 Fonctions supplémentaires

Les algorithmes rapportent des points supplémentaires en fonction de leur difficulté.



Faites attention à n'utiliser que les instructions implémentées.

Utiliser un compilateur RISC-V ne vous fournira PAS de code compatible (instructions non supportées, directives de sections ...).

2.2.1 Facile

- **Gestion des boutons:** afficher des patterns différents sur les 32 LED selon l'appui des boutons. Le dernier bouton appuyé est sauvegardé pour laisser le pattern affiché.
- **Gestion des timings:** faire clignoter les LED à une fréquence donnée. L'appui des boutons permet d'augmenter/diminuer la fréquence.

2.2.2 Moyen

- **Chenillard sur les LED :** les LED s'allument une après l'autre à une fréquence modérée; un appui sur un bouton arrête l'avancée du chenillard, tandis qu'un autre le fait redémarrer de 0. Les boutons doivent être filtrés contre les rebonds de manière logicielle et un appui ne doit être pris en compte qu'une fois tant qu'il n'est pas relâché puis réappuyé.
- **LED à intensité variable:** les LED s'allument avec une Pulse Width Modulation (PWM) réglée à 50%. Un appui sur un bouton augmente l'intensité de 10% à chaque pression, tandis qu'un autre la diminue de 10%. Les boutons doivent être filtrés contre les rebonds de manière logicielle. Les boutons doivent être filtrés contre les rebonds de manière logicielle et un appui ne doit être pris en compte qu'une fois tant qu'il n'est pas relâché puis réappuyé.

2.2.3 Difficile

- **Universal Asynchronous Receiver Transmitter (UART):** lors d'un appui sur un bouton, une transmission de texte par UART est effectuée. Le texte est affiché sur un PC.
- **LED RGB sérielles:** gestion de LED s RGB sérielles type WS2812/WS2813. Un appui sur un bouton allume/éteint les LED ; deux autres permettent de passer à la couleur suivante/précédente.
- **LED respirantes:** les LED s s'allument puis s'éteignent progressivement, donnant un effet de « respiration ». Un bouton permet d'augmenter la vitesse de respiration, un autre de la diminuer. Deux autres modifient l'intensité des LED s.
- **Capteur de distance à ultrasons:** gestion d'un capteur à ultrasons PMOD Maxsonar - [Table 3](#) - afin d'afficher un niveau de distance sur les LED s.

Il est bien sûr possible de proposer vos propres idées et d'utiliser du matériel annexe (voir Chapitre 3).



Il est hautement conseillé de développer et tester votre algorithme sur les outils vus durant les laboratoires ISA ([RISC-V online interpreter](#) et [Ripes](#))

Il est possible de manuellement modifier les registres sur Ripes pour simuler les boutons.

2.3 Projet HDL-Designer

Un projet HDL-Designer prédéfini peut être téléchargé sur [Cyberlearn](#) ou cloné par [Git](#). La structure de fichier du projet se présente comme suit:

```
car_heirv
+--Board/           # Project and files for programming the fpga
|   +-concat/       # Complete VHDL file including PIN-UCF file
|   +-hds/          # Board-related VHDL files
|   +-ise/          # Xilinx ISE project
|   +-diamond/      # Lattice Diamond project
+--HEIRV32/         # Library for the components of the student solution
+--HEIRV32\_test/   # Library for the simulation testbenches
+--Libs/            # External libraries which can be used e.g. gates, io,
sequential
+--Prefs/           # HDL-Designer settings
+--Scripts/         # HDL-Designer scripts
+--Simulation/     # Modelsim simulation files
+--doc/             # Folder with additional documents relevant to the project
|   +-Board/        # All schematics of the hardware boards
|   +-Components/   # All data sheets of hardware components
|   +-HEIRV32\_MC\-\-x # This doc
+--heirv32\asm/     # Dedicated assembler for HEIRV32 (doc and execs)
+--img/             # Pictures
```



Le chemin d'accès au dossier du projet ne doit pas contenir d'espaces.



Le dossier **doc/** contient de nombreuses informations importantes: fiches techniques, évaluation de projet et documents d'aide pour HDL-Designer, pour n'en citer que quelques-uns.

Les signaux du top-level se présentent comme suit :

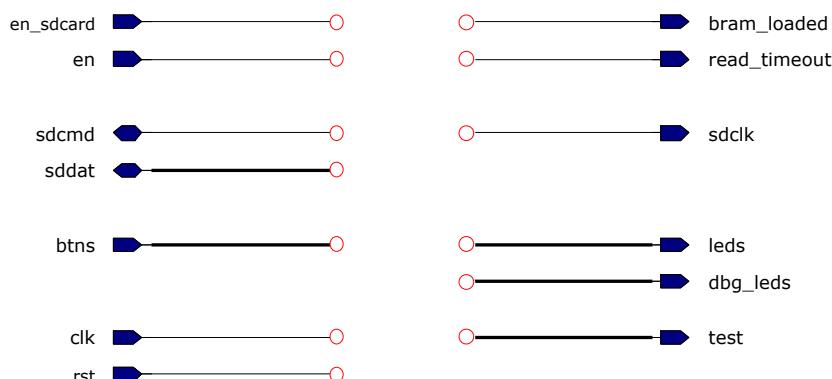


Figure 3 - Signaux Toplevel vide



Les signaux disponibles travaillent selon les groupes suivants :

- Système
 - **clk** : horloge du système, cadencée à **25MHz**
 - **rst** : reset asynchrone du système
 - **test[47:0]** : vecteur groupant plusieurs signaux du système pour automatiser les tests en simulation
 - **en_sdcard** : signal activant la lecture du code depuis la carte SD - *inutilisé par défaut dans la solution étudiante*
 - **en** : signal indépendant permettant d'activer/désactiver le processeur - *inutilisé par défaut dans la solution étudiante*
- Processeur
 - **bram_loaded** : indication que le programme a correctement été chargé depuis la carte SD, reflété sur la **LED verte**
 - **read_timeout** : indication que le programme n'a pas réussi à être trouvé/chargé depuis la carte SD
- Carte SD
 - **sdclk**, **sdcmd**, **sddat[3:0]** : horloge, commande et données de la carte SD
- Entrées/Sorties
 - **btms[3:0]** : boutons S4 à S1 de la carte électronique boutons-LEDs [Figure 4](#), directement reliés au registre **x31**
 - **leds[31:0]** : **LED** ou autres sorties activées selon le registre **x30**
 - **dbg_leds[31:0]** : signaux de debug, permettent d'activer des **LED** selon des signaux internes. *Ne sont pas liés au registre x30.*
- Port série (UART)
 - Le port USB-C présent sur la carte mère peut être utilisé comme port série UART pour communiquer avec un PC comme proposé pour un code avancé [Chapitre 2.2.3](#).
 - **uart_tx** : transmission UART depuis le processeur RISC-V, relié au registre **leds[31]**
 - **uart_rx** : réception UART vers le processeur RISC-V, relié au registre **btms[31]**

2.4 Blocs fournis

Tous les blocs nécessaires à la conception du microprocesseur sont fournis dans le bloc **heirv32_mc** et proviennent des bibliothèques suivantes:

- HEIRV32_MC
 - **controlUnit** : bloc pour le décodage des instructions
 - **heirv32_mc** : top-level
 - **instructionDataManagerSDCard** : mémoire du programme groupant instructions et data, capable de lecture et écriture, lisant le contenu de la carte SD
- HEIRV32
 - **ALU** : une version de l'ALU capable d'addition, soustraction, AND, OR, et SLT
 - **buffer*(Enable)** : buffer clockés (bascules) avec ou sans entrée enable
 - **extend** : bloc d'extension de l'instruction pour les tests, supportant les instructions I, S, B et J
 - **mux3To1ULogVec** : mux 3 vers 1 de **std_ulogic_vector**
 - **registerFile** : bloc de gestion des 32 registres, remplaçant **x31** par le vecteur **btms** - *registre de lecture des boutons* - et **x30** par le vecteur **leds** - *registre d'écriture des leds* -

La librairie **Board** contient la logique de mise en forme des signaux, prévue pour le déploiement du circuit sur [FPGA](#) .



Ne modifiez pas les noms des signaux déjà présents dans le top-level. Le vecteur **test[47:0]** est basé sur ces noms.

2.5 Indications sur la carte fille

Les **LED** d'indication sur la carte fille sont utilisées pour indiquer l'état du système, dans l'ordre:

- **LED bleue** : heartbeat du système à 2 [Hz] (**version solution**) / 8 [Hz] (**version étudiant**), montrant que le design **FPGA** est actif.

Si la carte SD n'est pas insérée, la LED ne clignote pas, le système étant tenu en reset.

- **LED verte** : indique que le **programme** a correctement été **chargé** depuis la carte SD et le processeur est activé pour l'exécution du code.

- **LED rouge** : indique que le **processeur** est **activé**.

Le bouton S4 (dés)active le processeur. Le bouton S3 active le processeur pendant un seul coup de clock lorsqu'il est désactivé. Ces deux boutons n'ont pas d'effet dans la version étudiante (processeur toujours activé).

2.6 Simulateur

La librairie **HEIRV32_test** contient le testeur **heirv32_mc_tb** afin de simuler l'exécution du processeur.

Le processeur charge en mémoire le fichier **Simulation/code_sim_bram.txt** qui est le code assembleur **Simulation/code_sim.s** compilé par **heirv32-asm/HEIRV32-ASM_xxx**.

3 | Composants

Le système se compose de 3 plaques matérielles différentes, visibles dans la figure [Figure 1](#).

- Une carte de développement [FPGA](#), voir figure [Figure 4](#).
- Une carte de contrôle à 4 boutons et 8 [LEDs](#), voir figure [Figure 5](#).
- Deux cartes [LEDs Peripheral Module \(PMod\)](#), voir figure [Figure 6](#).

3.1 Carte [FPGA](#) EBS

La carte principale est la carte de développement de laboratoire [FPGA](#) EBS 3 de l'école. Elle héberge une puce [Lattice LFE5U-25F FPGA](#) et dispose de nombreuses interfaces différentes ([UART](#), [PMod](#), PPT, Ethernet). L'oscillateur utilisé produit un signal d'horloge ([clock](#)) avec une fréquence de $f_{clk} = 100\text{MHz}$, réduit en interne par PLL à $f_{clk} = 25\text{MHz}$.

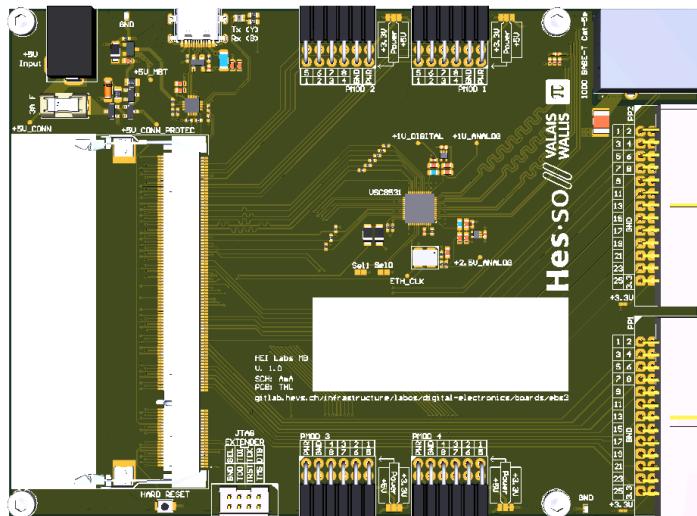


Figure 4 - Carte électronique [FPGA](#)

3.2 Boutons et [LEDs](#)

La plaque avec boutons et les [LEDs](#) [8] est connectée à la motherboard. Elle possède 4 boutons et 8 [LEDs](#) qui peuvent être utilisés dans le design. Si on le souhaite, cette plaque peut être équipée d'un affichage [Liquid Crystal Display \(LCD\)](#) [9], [30].



Figure 5 - Carte électronique boutons [LED](#) [8]

3.3 LEDs supplémentaires

Il est possible d'étendre le nombre de LED grâce à des boards d'extension dédiées.



Figure 6 - LED PMod

3.4 Boards optionnelles

Pour ajouter de la fonctionnalité à votre circuit, il est possible d'utiliser diverses cartes d'extension. Leurs documentations sont données sous le dossier [doc/ext_boards](#).

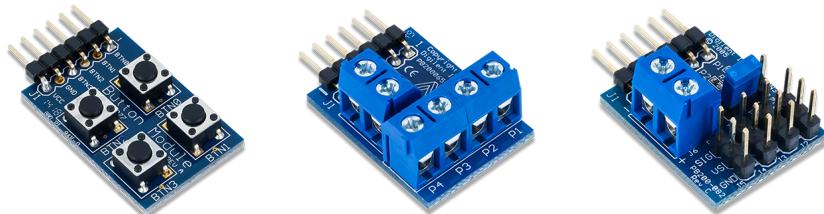


Table 1 - Inputs: PMod BTN [19], [20], PMod CON1 [21], [22], PMod CON3 [23], [24]

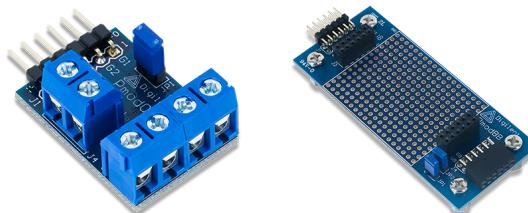


Table 2 - Outputs: PMod OD1 [26], [27], PMod BB [17], [18]



Table 3 - I/Os: PMod MAXSONAR [25], [29]

4 | Evaluation

Dans le dossier **doc/**, le fichier **evaluation-bewertung-riscv.pdf** montre le schéma d'évaluation détaillé, tableau [Table 4](#).

La note finale contient le rapport, le code ainsi qu'une présentation de votre système.

Aspects évalués	Points
Rapport	45
Introduction	3
Spécification	5
Projet	15
Vérification et validation	5
Intégration	9
Conclusion	3
Aspects formels du rapport	5
Fonctionnalité du circuit	60
Functions minimale	30
Développement supplémentaire 1	15
Développement supplémentaire 2	15
Qualité de la solution	15
Présentation	30
Total	150

Table 4 - Grille d'évaluation



La grille d'évaluation donne des indications sur la structure du rapport. Pour un bon rapport, consultez le document « Comment rédiger un rapport de projet » [\[13\]](#).

5 | Guide

Pour commencer le projet, procédez de la manière suivante :

- Lisez attentivement les spécifications et les informations présentées.
- Examinez le matériel grâce au programme préinstallé ainsi que le projet HDL-Designer.
- Parcourez les documents dans le dossier **doc/** de votre projet.
- Analysez en détail les blocs qui existent déjà.
- Développez un schéma fonctionnel détaillé. Vous devez pouvoir expliquer les signaux et leurs fonctions.
- Implémentez et simulez les différents blocs.
- Testez la solution sur la **FPGA** et trouvez les éventuelles erreurs
- Ecrivez et déployez votre propre code.

5.1 Architecture générale

Proposez d'abord une architecture sans implémenter de bloc s'axant autour du principe multi-cycle:



Figure 7 - Pipeline RISC-V

5.1.1 Signal en

L'entrée **en** permet de couper le fonctionnement du processeur.

Elle est tenue à `< 1 >` par défaut afin de pouvoir utiliser les 4 boutons sans que le système ne s'arrête.

Veillez à garder ce signal tel que déjà précâblé dans les blocs donnés !

5.1.2 Instruction lw

La création du design s'axe autour de la mémoire de programme en réfléchissant à l'instruction **lw**, instruction nécessitant les 5 étapes du pipeline.

L'instruction **lw rd, imm(rs1)** se résume ainsi:

- *Fetch*: l'instruction est récupérée de la mémoire
- *Decode*: le registre **rs1** est extrait de l'instruction, ainsi que la valeur **imm** qui est étendue sur 32 bits
- *Execute*: l'adresse mémoire est calculée en additionnant **rs1** et **imm**
- *Mem. Access*: la mémoire est pointée à l'adresse calculée précédemment, donnant la valeur à charger dans le registre **rd**
- *Write Back*: la valeur est enregistrée dans le registre **rd**

En terme de circuit, l'instruction **lw rd, imm(rs1)** donne:

- **Fetch:** le program counter **PC** sélectionne de l'information en mémoire, générant un bus **data (RD)** et **instruction (instr.)**. Les bus sont séquentiels: **Data** est lu de la mémoire à chaque coup de clock, tandis que **Instr.** n'est mis à jour que si l'entrée **IRWrite** est à '1'.

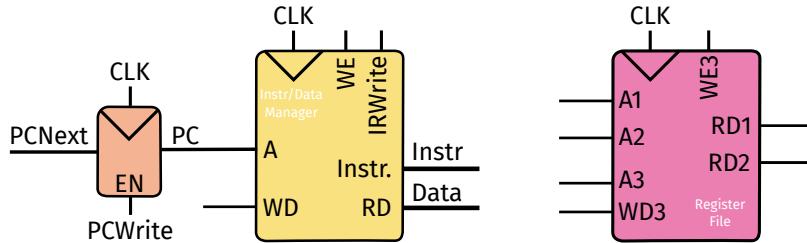


Figure 8 - Fetch

• Decode:

- l'adresse de base est contenue dans le bus **Instr.**, bits 19 downto 15. Ils sont utilisés comme adresse pour sélectionner le registre **RD1**.
- la valeur immédiate est donnée dans les bits 31 downto 20, dont le signe doit être étendu. Pour ça, le bloc **extend** permet, grâce à deux bits de contrôle, de définir si la valeur est codée sur 12, 13 ou 21 bits et donc d'étendre le signe de la valeur.
- les bascules séparant cette étape sont cachées à l'intérieur du bloc **register file**

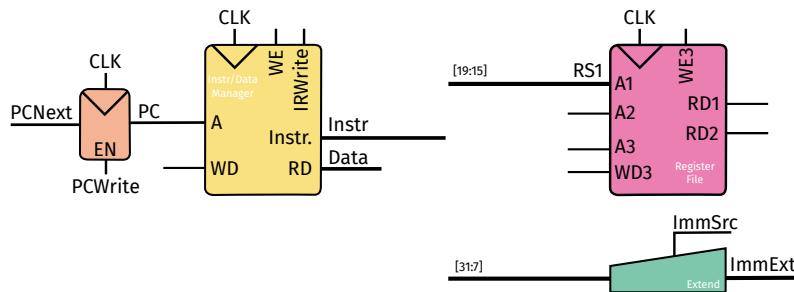


Figure 9 - Decode

- **Execute:** la valeur immédiate est ajoutée au registre lu au travers de l'ALU pour déterminer l'adresse à accéder en mémoire. La bascule de sortie permet une nouvelle fois de séparer cette étape du reste du pipeline.

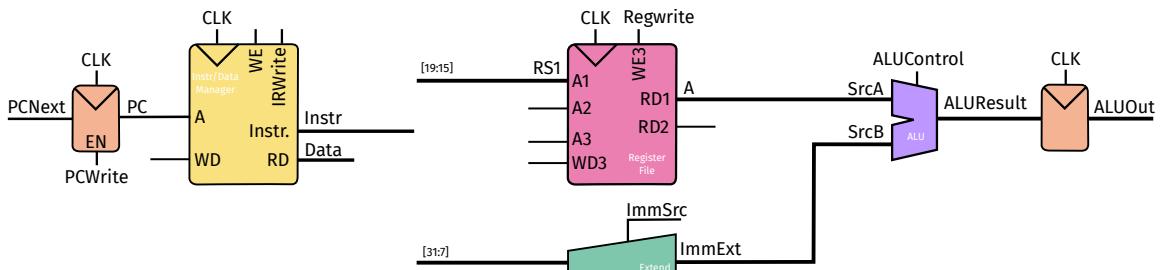


Figure 10 - Execute

- **Mem. Access:** la valeur calculée est utilisée pour relire la mémoire. A cet effet, un multiplexeur est ajouté afin de sélectionner entre le PC ou la valeur de l'ALU, contrôlé par le signal **AdrSrc**.

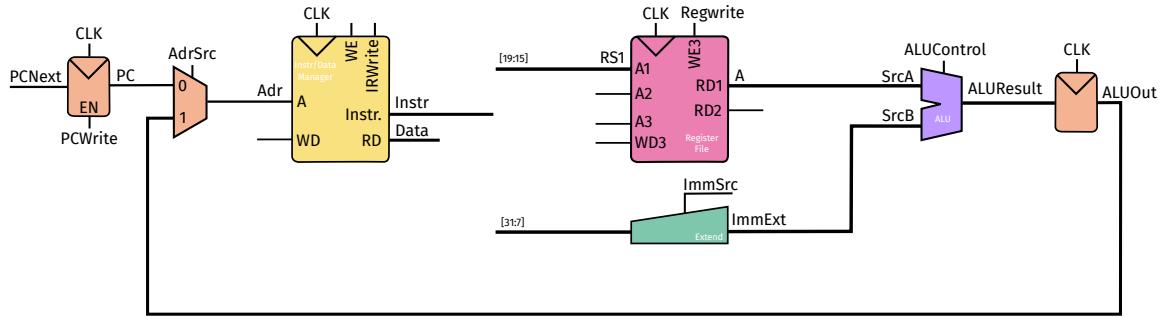


Figure 11 - Memory Access

- Write Back:** la dernière étape est d'écrire le registre de destination spécifié par les bits 11 downto 7 du bus **Instr**. Le signal **RegWrite** charge le registre pointé par A3 au prochain coup de clock. Cette valeur peut provenir du résultat de l'ALU comme ici ou de la donnée elle-même. Un multiplexeur, contrôlé par le signal **ResultSrc**, est ajouté:

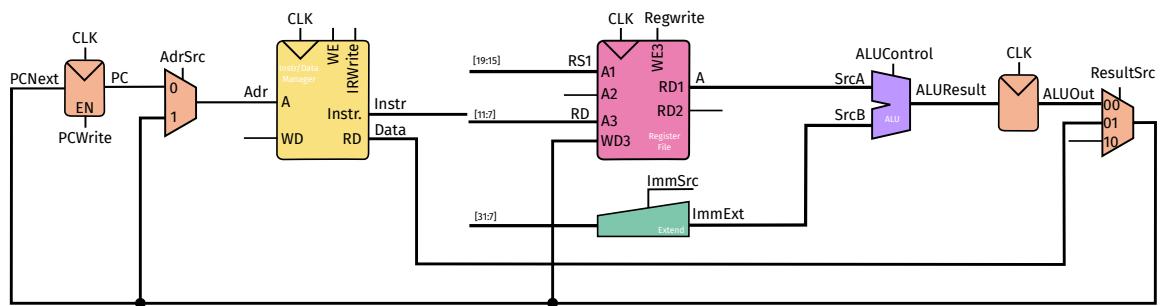


Figure 12 - Write Back

PC + 4

En parallèle de tout ça, le compteur de programme doit être incrémenté. Cela était fait avec un additionneur séparé dans l'architecture single-cycle. Hors, ici, il est possible d'utiliser l'ALU pendant l'étape de fetch car aucun calcul n'est effectué. Deux multiplexeurs sont ajoutés pour contrôler les sources de l'ALU, contrôlés par les signaux **AdrSrcA** et **AdrSrcB**. Ici, le **PC** est chargé sur A pendant que B charge la valeur 4, l'ALU étant réglé sur addition. Le calcul, devant être utilisé immédiatement (**PC** doit être enregistré pour l'utiliser plus tard), la bascule du résultat de l'ALU est bypassée et le signal ajouté au multiplexeur de sortie. L'entrée enable contrôlée par **PCWrite** est à '1', permettant de sauvegarder **PCNext = PC + 4** sur **PC**:

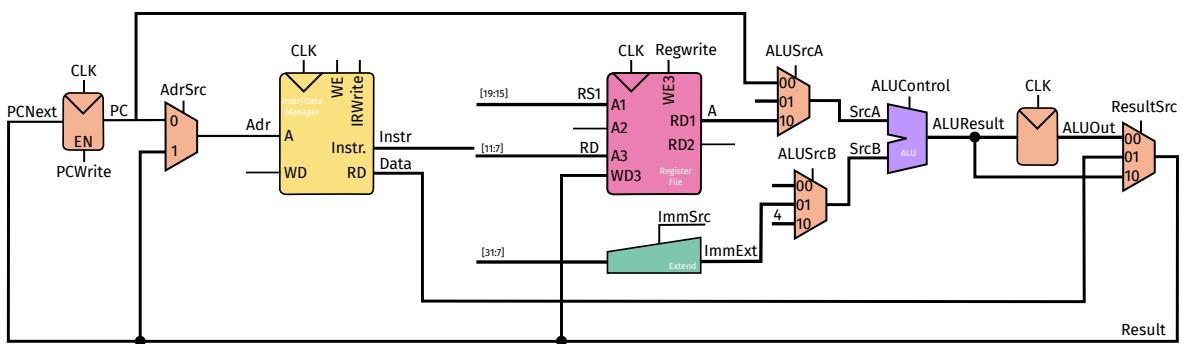


Figure 13 - PC + 4

Les signaux de contrôle sont générés par le bloc **controlUnit** abordé plus bas.



5.1.3 Instruction sw

Dans le même raisonnement, étendre le système pour supporter **sw**:

- *Fetch*: lit l'instruction depuis la mémoire pointée par **PC**.
- *Decode*: charge le registre spécifiant l'adresse de base sur **RD1**. Aussi, charge le registre contenant l'information à sauvegarder sur **RD2**.
- *Execute*: ajoute la valeur immédiate à l'adresse de base à travers l'ALU, afin de pointer vers l'adresse mémoire.
- *Write Back*: enregistre la valeur en mémoire grâce au signal **MemWrite**.

5.1.4 Instruction Type R - I

Réfléchir ensuite aux chemins nécessaires pour les instructions de type R et I:

- *Fetch*: lit l'instruction depuis la mémoire pointée par **PC**.
- *Decode*: charge les registres sources nécessaires.
- *Execute*: effectue l'opération demandée par l'instruction au travers de l'ALU.
- *Write Back*: enregistre le résultat dans le registre de destination si nécessaire.

5.1.5 Instruction beq

Ajouter l'instruction **beq** qui compare deux registres et modifie le **PC** si ces derniers sont égaux:

- *Fetch*: lit l'instruction depuis la mémoire pointée par **PC**.
- *Decode*: charge les deux registres à comparer. Comme l'ALU n'est pas utilisé, l'adresse en cas de saut peut être calculée. Toutefois, le **PC** s'est déjà incrémenté. Il est donc nécessaire d'enregistrer une ancienne valeur de **PC** (**oldPC**) à l'étape précédente afin de la charger sur la source A de l'ALU. La source B est la valeur immédiate.
- *Execute*: soustrait les deux registres et met le signal **zero** à '**1**' si le résultat est 0. Dans ce cas, le bloc de contrôle met le signal **PCWrite** à '**1**', signal permettant de charger le bus **Result** sur **PC**. Ainsi, la valeur de saut (sortie de l'ALU de l'étape précédente) est chargée: $a == b, PC = PC + \text{imm..}$ Sinon, **PCWrite** reste à '**0**' et le **PC** n'est pas modifié: $a \neq b, PC = PC + 4$.

5.1.6 Instruction jal

Ajouter l'instruction **jal** qui saute après avoir enregistré l'adresse de retour :

- *Fetch*: lit l'instruction depuis la mémoire pointée par **PC**.
- *Decode*: calcule l'adresse de saut **oldPC + imm**.
- *Execute*: enregistre l'adresse de saut comme nouveau **PC** tout en calculant l'adresse de retour **oldPC + 4** à enregistrer dans le registre de destination.
- *Write Back*: enregistre l'adresse de retour dans le registre de destination.



5.1.7 Instruction jalr

Réfléchir aux chemins nécessaires pour l'instruction **jalr**. Elle fonctionne de manière similaire à **jal**, mais au lieu de sauter à **oldPC + imm**, elle saute à l'adresse **rs1 + imm**.

- *Fetch*: lit l'instruction depuis la mémoire pointée par **PC**.
- *Decode*: laisse le temps à **rs1** d'être lu depuis le registre.
- *Execute1*: calcule l'adresse de saut **rs1 + imm**.
- *Execute2*: enregistre l'adresse de saut comme nouveau **PC** tout en calculant l'adresse de retour **oldPC + 4** à enregistrer dans le registre de destination.
- *Write Back*: enregistre l'adresse de retour dans le registre de destination.



*A noter ici que l'étape d'exécution est divisée en deux sous-étapes. Dans le cas de **JAL**, il était possible de calculer l'adresse de saut pendant l'étape de décodage car **oldPC** et **imm** sont déjà disponibles au niveau de l'ALU (le bloc d'extension de la valeur immédiate ne requiert pas de coup de clock).*

*Dans le cas de **JALR**, l'étape de décodage doit être terminée avant de pouvoir utiliser la valeur du registre **rs1**.*

Plusieurs stratégies sont envisageables, dont des modifications de l'architecture globale pour des processeurs plus complets. Dans notre cas, l'ajout d'une seconde étape d'exécution est la plus simple pour palier à ce problème.

5.2 Control Unit

Il manque encore un bloc de contrôle permettant de suivre l'étape actuelle selon le pipeline précité et générant les divers signaux de contrôle. Pour simplifier le travail, séparez la génération de signaux en trois blocs distincts:

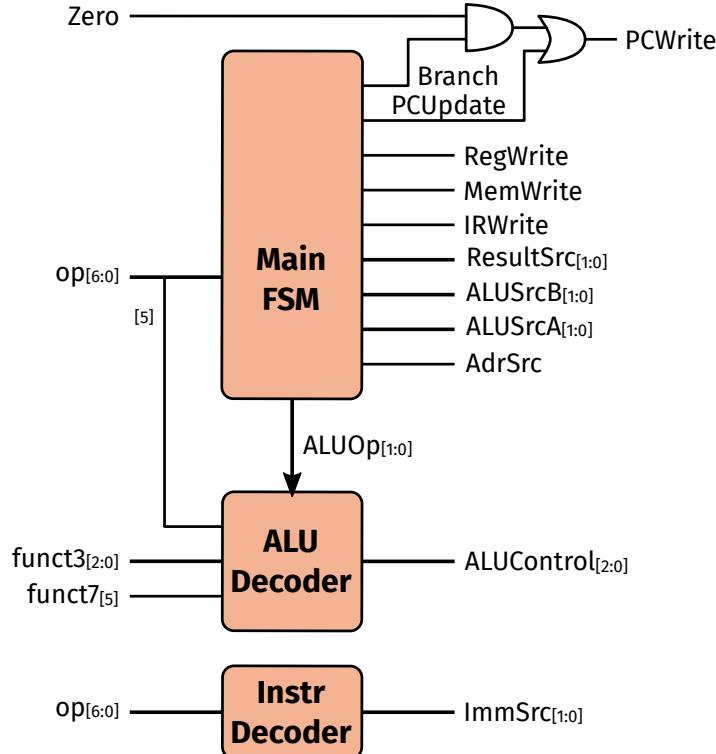


Figure 14 - Control unit



Les instructions du set RV32I sont séparées en 6 types : R, I, S, B, U et J. Il est logique que deux instructions du même type suivent le même traitement. Si ce n'était pas le cas, chaque instruction devrait être traitée différemment par le bloc controlUnit. On en dénombre 40 rien que pour le set de base, set ne permettant même pas de faire tourner un OS !



5.2.1 Main FSM

La machine d'état gère les signaux de contrôle du processeur. Elle est responsable de suivre les différentes étapes du pipeline afin de configurer chaque bloc de manière adéquate. Certaines étapes du pipeline sont communes à plusieurs instructions, d'autres spécifiques à un certain type. *Le signal **ALUOp** est expliqué sous le Chapitre 5.2.2.1.*

5.2.2 ALU Decoder

L'ALU effectue des fonctions arithmétiques et logiques selon le signal **ALUControl** telles que définies par la table suivante:

ALU Control	Operation
000	<i>add</i>
001	<i>sub</i>
010	<i>AND</i>
011	<i>OR</i>
100	<i>XOR</i>
101	<i>Set Lesser Than</i>
110	<i>Shift Left Logical</i>
111	<i>Shift Right Logical</i>
Others	-

Table 5 - Table de commande du bloc ALU

En se basant sur divers signaux, le bloc **ALU Decoder** doit être capable de contrôler l'opération mathématique/logique en cours.

5.2.2.1 Signal **ALUOp**

Il y'a des étapes du pipeline pour lesquelles l'ALU est utilisé par la FSM au lieu de l'instruction. Par exemple, pendant l'étape de **Fetch**, l'ALU est utilisé pour calculer l'adresse suivante et doit donc être configuré sur addition.

La **Main FSM** génère donc le signal **ALUOp** qui est un code de 2 bits permettant de forcer le bloc **ALU Decoder** à effectuer une opération spécifique:

operation	ALUOp_[2:0]
add	00
sub	01
instruction based	10
X	11

Table 6 - Table de commande du bloc ALUDeocder

5.2.2.2 Décodage

Certaines instructions sont aussi liées par leur fonctionnement, même si elles sont d'un type différent. Par exemple, **addi x2, x3, 30** et **add x2, x3, x4** effectuent tous deux une opération d'addition. Les instructions I et R sont donc groupées sous le même code.

La table suivante présente cette idée d'unification et liste les cas spéciaux pour le décodage:

ALUOp	funct3	Op ₅ · funct7 ₅	instr	ALUControl _[2:0]
00	---	--	<i>lw, sw</i>	000 (add)
01	---	--	<i>beq</i>	001 (sub)
10	000	0·0, 0·1, 1·0	<i>add / addi</i>	000 (add)
10	000	1·1	<i>sub</i>	001 (sub)
10	001	--	<i>sll / slli</i>	110 (sll)
10	010	--	<i>slt / slti</i>	101 (slt)
10	100	--	<i>xor / xori</i>	100 (xor)
10	101	..0	<i>srl / srli</i>	111 (srl)
10	110	--	<i>or / ori</i>	011 (or)
10	111	--	<i>and / andi</i>	010 (and)

Table 7 - Table de commande du bloc ALUDeocder

5.2.3 Instr. Decoder

Le bloc extend se base sur le type d'instruction donné par **immSrc** pour extraire et étendre le signe de la valeur immédiate:

immSrc	Type
00	<i>I</i>
01	<i>S</i>
10	<i>B</i>
11	<i>J</i>
--	<i>Others</i>

Table 8 - Table de commande du bloc extend

Le bloc **Instr. Decoder** doit donc, en se basant sur l'opérande **op[6:0]**, identifier l'instruction et en déduire la configuration correcte pour **ImmSrc**.



Complétez les blocs **Main FSM**, **ALU Decoder** et **Instr. Decoder** pour supporter les instructions listées sous le [Chapitre 2](#).



Pensez à vérifier de quel type chaque instruction fait partie (R, I, S, B, U, J). Le nom même de l'instruction peut être trompeur !

6 | Tests

6.1 Simulation

Pour simuler le circuit complet, un banc de test est disponible sous **HEIRV32_test/heirv32_mc_tb**. Il exécute le code donnée sous **Simulation/code_sim.s**.

6.1.1 Compréhension

Ouvrez et analysez le code donné.



- Quelles sont les instructions exécutées ?
- Est-il un bon candidat pour confirmer le fonctionnement de votre processeur ?

6.1.2 Automatisation des tests

Le testeur **HEIRV32_test/heirv32_mc_tb** affiche un signal **testInfo** sur la simulation permettant de savoir théoriquement quelle instruction devrait être en train de s'exécuter. En cas de non-fonctionnement du processeur, cette information ne vaut rien.

Les tests sont automatisés grâce à la procédure :

```

1  procedure checkProc(
2    msg :          string;
3    AdrArg :       unsigned(31 downto 0);
4    ALUControlArg : std_ulogic_vector(2 downto 0);
5    ALUSrcAArg :   std_ulogic_vector(1 downto 0);
6    ALUSrcBArg :   std_ulogic_vector(1 downto 0);
7    IRWriteArg :   std_ulogic;
8    PCWriteArg :   std_ulogic;
9    adrSrcArg :   std_ulogic;
10   immSrcArg :  std_ulogic_vector(1 downto 0);
11   memWriteArg : std_ulogic;
12   regwriteArg : std_ulogic;
13   resultSrcArg : std_ulogic_vector(1 downto 0)) is
14 begin
15   ...
16 end procedure checkProc;
```

Elle est utilisée dans la boucle du testeur telle que

```
checkProc("Addi, addr. 0x00 - decode", x"00000004", "000", "01", "01", '0', '0', '0',
"00", '0', '0', "00");
```

Elle permet simplement de contrôler l'état actuel du processeur par rapport à l'état attendu. En cas d'erreur, le test s'arrête.

Il se peut que votre implémentation diffère légèrement de la solution donnée suivant vos choix d'implémentation. A vous de manuellement juger et corriger au besoin le code de test.



Confirmez le fonctionnement de votre circuit.

6.2 Code

Une fois l'architecture validée par simulation, il est possible de flasher la [FPGA](#).

Pour ça, la librairie **Board** contient le top-level. Le code flashé est celui sous **Simulation/code.bin**.



Le code **Simulation/code.s** donné est vide, à l'inverse de **Simulation/code.bin**. A vous d'écrire un nouveau code une fois le fonctionnement du circuit validé.

6.2.1 Contrôle par équivalence

Testez dans un premier temps votre système au travers de la solution donnée:

- Débranchez la carte de développement [FPGA](#).
- Chargez le code sur la carte micro SD en copiant le fichier **Simulation/code.bin** en gardant le même nom. Une copie du code est aussi disponible sur la carte SD - *ne pas la supprimer de la carte*.
- Rebranchez la carte.
- Branchez l'alimentation par câble USB-C.
- Vérifiez le chargement du code, fonctionnement des boutons, allumage des [LED](#), comportement général du système.

Flashez ensuite votre processeur et réitérez les essais. Assurez-vous que tout fonctionne correctement.

Référez-vous au document [doc/Board_LFE5U-25F.pdf](#) pour le flash de la [FPGA](#).



Flashez dans un premier temps la FPGA en JTAG (de façon temporaire). Cela vous permet de revenir sur la version solution en débranchant et rebranchant l'alimentation.

Une fois le tout validé, programmez la configuration en flash afin de garder votre processeur de façon permanente.



Confirmez le fonctionnement de votre processeur sur la FPGA.



6.2.2 Code personnalisé

Lorsque le circuit est fonctionnel, écrivez votre propre code dans un fichier **.s**, capable de réagir aux appuis sur les deux boutons ainsi que d'allumer des **LED** :

- Ecrire une valeur sur le registre **x30** permet d'allumer les **LED** . La **LED 0** correspond au bit 0, la **LED 1** au bit 1 ...
- Lire le registre **x30** donne l'état actuel des **LED** .
- Lire les boutons se fait en lisant le registre **x31**. Le bit 0 correspond au bouton **S0**, le bit 1 au bouton **S1**.
- Ecrire le registre **x31** ne le modifie pas.

Testez votre code à l'aide du [RISC-V online interpreter](#) et [Ripes](#) avant de le flasher.



Ecrivez et testez votre code sur simulateur.

Une fois le code validé, compilez-le à l'aide du logiciel **heirv32-asm/HEIRV32-ASM_xxx** afin de générer un nouveau fichier **code.bin**.

Pour charger le code, copiez le fichier **xxxx.bin** généré sur la carte micro SD en le renommant **code.bin**.

Débranchez la carte de développement **FPGA** , insérez la carte micro SD, puis rebranchez la carte.



Confirmez le fonctionnement de votre code.



6.3 Tips

Ci-joint quelques conseils supplémentaires pour éviter les problèmes et les pertes de temps:

- Divisez le problème en différents blocs : utilisez pour cela le document Toplevel vide. Il est recommandé d'avoir un mélange équilibré entre le nombre de composants et la taille/complexité de ces derniers.
- Analysez les différents signaux d'entrée et de sortie, leurs types, leurs tailles ... Il est conseillé d'utiliser en partie les fiches techniques.
- Respectez le chapitre DiD « Méthodologie de conception de circuits numériques (MET) » lors de la création du système. [11].
- Respectez la marche à suivre incrémentale proposée. Abusez des tests dès que possible.
- Sauvegardez et documentez vos étapes intermédiaires. Les architectures n'ayant pas fonctionnées, les codes basiques pour tester l'architecture ... sont autant de matière à ajouter au rapport.



N'oubliez pas de vous amuser.





Glossaire

FPGA – Field Programmable Gate Array [2](#), [3](#), [5](#), [8](#), [10](#), [13](#), [22](#), [23](#)

LCD – Liquid Crystal Display [10](#)

LED – Light Emitting Diode [2](#), [5](#), [6](#), [8](#), [9](#), [10](#), [11](#), [22](#), [23](#)

PMod – Peripheral Module [10](#), [11](#)

PWM – Pulse Width Modulation [6](#)

RISCV – 5-stages Reduced Instruction Set Computer architecture, open-sourced [3](#), [4](#), [5](#), [13](#)

UART – Universal Asynchronous Receiver Transmitter [6](#), [10](#)



Bibliographie

- [1] A. Waterman, K. Asanovic, et F. Embeddev, « RISC-V Instruction Set Manual, Volume I: RISC-V User-Level ISA ». Consulté le: 4 juin 2022. [En ligne]. Disponible sur: <https://www.five-embeddev.com/riscv-isas-manual/latest/riscv-spec.html>
- [2] F. Embeddev, « RISC-V Quick Reference ». Consulté le: 4 juin 2022. [En ligne]. Disponible sur: <https://www.five-embeddev.com/quickref/tools.html>
- [3] S. L. Harris et D. M. Harris, « Digital Design and Computer Architecture RISC-V Edition », *Digital Design and Computer Architecture*. Elsevier, p. IBC1-IBC2, 2022. doi: [10.1016/B978-0-12-820064-3.00025-8](https://doi.org/10.1016/B978-0-12-820064-3.00025-8).
- [4] D. A. Patterson et J. L. Hennessy, *Computer Organization and Design - RISC-V Edition*, Second Edition. Elsevier, 2021.
- [5] Xilinx, « Spartan-3 FPGA Family ». Consulté le: 20 novembre 2021. [En ligne]. Disponible sur: <https://www.xilinx.com/products/silicon-devices/fpga/spartan-3.html>
- [6] Xilinx, « Datasheet Spartan-3E FPGA Family ». 2008.
- [7] Silvan Zahno, « Schematic: FPGA-EBS v2.2 ». 2014.
- [8] Silvan Zahno, « Schematic: Parallelport HEB LCD V2 ». 2014.
- [9] Electronic Assembly, « Datasheet: DOGM Graphics Series 132x32 Dots ». 2005.
- [10] François Corthay, Silvan Zahno, et Christophe Bianchi, « Methodologie Für Die Entwicklung von Digitalen Schaltungen ». 2021.
- [11] François Corthay, Silvan Zahno, et Christophe Bianchi, « Méthodologie de Conception de Circuits Numériques ». 2021.
- [12] Christophe Bianchi, François Corthay, et Silvan Zahno, « Wie Verfasst Man Einen Projektbericht? ». 2021.
- [13] Christophe Bianchi, François Corthay, et Silvan Zahno, « Comment Rédiger Un Rapport de Projet? ». 2021.
- [14] S. Zahno, « CAr RISC-V Summary (RISC-V) », 2022.
- [15] D. Inc, « Pmod 8LD Reference Manual ». 2015.
- [16] D. Inc, « Pmod 8LD Schematics ». 2008.
- [17] D. Inc, « Pmod BB Reference Manual ». 2016.
- [18] D. Inc, « Pmod BB Schematics ». 2007.
- [19] D. Inc, « Pmod BTN Reference Manual ». 2016.
- [20] D. Inc, « Pmod BTN Schematics ». 2005.
- [21] D. Inc, « Pmod CON1 Reference Manual ». 2015.
- [22] D. Inc, « Pmod CON1 Schematics ». 2015.
- [23] D. Inc, « Pmod CON3 Reference Manual ». 2016.



- [24] D. Inc, « Pmod CON3 Schematics ». 2005.
- [25] D. Inc, « Pmod MAXSONAR Reference Manual ». 2015.
- [26] D. Inc, « Pmod OD1 Reference Manual ». 2016.
- [27] D. Inc, « Pmod OD1 Schematics ». 2006.
- [28] Laurent Gauch, « Schematic: HEB Dot Matrix v1.0 ». 2003.
- [29] Maxbotix, « LV-MAXSONAR-EZ Datasheet ». 2021.
- [30] Sitronix, « Datasheet Sitronix ST7565R 65x1232 Dot Matrix LCD Controller/Driver ». 2006.