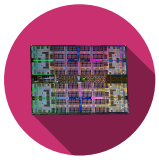


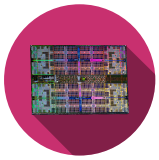
Single-Cycle RISC-V

Labor Architecture des ordinateurs

Contenu

1 Introduction

Ce laboratoire permet d'approcher l'architecture RISC-V à l'aide d'un processeur supportant un set d'instructions réduit, capable d'exécuter un petit programme assembleur en ne nécessitant qu'un seul coup de clock par instruction (single-cycle). Nommé **HEIRV32_SC**, ce dernier peut



être simulé ou déployé directement sur une puce FPGA. La connexion avec le monde extérieur peut se faire à l'aide de boutons et de leds.

2 Architecture

Hormis l'incrémentation du program counter, le reste du circuit fonctionne de manière purement combinatoire. Une instruction est donc exécutée à chaque coup de clock. Le circuit est le suivant :

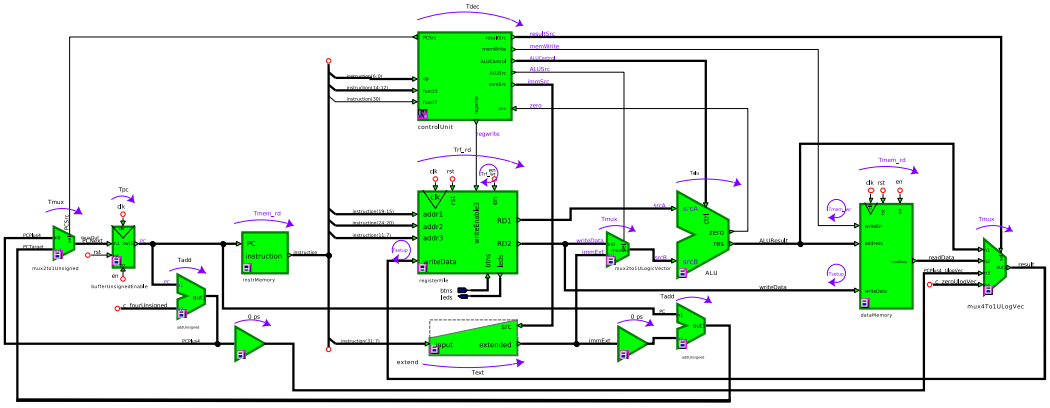
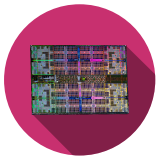


Fig. 1. – Top level

2.1 Analyse

Déterminez les diverses fonctionnalités du circuit:

1. Quel est le rôle de chaque bloc ?
2. Identifiez les 5 étapes du pipeline RISC-V et quels blocs y sont liés.
3. A quoi servent chaque signaux ? Quelle est leur taille ?



3 Control Unit

Une des principale caractéristique différençiant un type de processeur d'un autre est la façon dont sont encodées les instructions, liée à la structure interne. Par exemple, l'opération d'addition d'un nombre avec un registre existe tout autant dans l'architecture RISC-V qu'en **x86** ou **ARM**, mais l'opcode généré est tout autre.

Pour l'opération `addi x0 x0 1` en RISC-V (`add r0, r0, 1` en ARM), le résultat est :

- ARM : 0x 01 00 80 E2
- RISC-V : 0x 00 10 00 13

L'illustration Fig. 2 montre le bloc de contrôle qui reçoit tous les signaux d'entrée nécessaires pour générer les différents signaux de contrôle. Il est responsable de la configuration du circuit selon l'instruction donnée. Pour simplifier le circuit, **controlUnit** peut encore être divisé en deux sous-blocs **mainDecoder** et **ALUDecoder** :

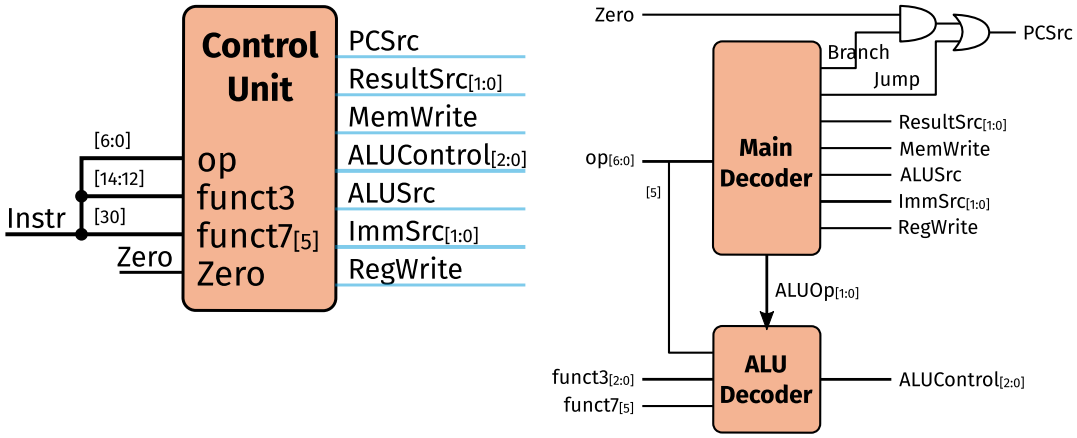


Fig. 2. – Control Unit

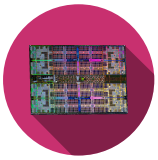
3.1 Réalisation

Complétez le bloc **controlUnit** afin de supporter les instructions suivantes :

- R-type : add, sub, slt, or, and
- I-type : addi, slti, ori, andi, lw
- S-type : sw
- B-type : beq
- J-type : jal

Pour cela, écrivez une table de vérité pour les deux sous-blocs **mainDecoder** et **ALUDecoder** en vous référant aux tables Fig. 3 et Fig. 4. La table Fig. 5 vous donne plus d'indications sur le signal **ALUOp** transitant entre les deux blocs.

Les blocs sont réalisés en VHDL directement. Des codes de référence sont disponibles dans les blocs concernés.



Pour créer de tels blocs dans HDL Designer, lorsque vous sélectionnez le type de contenu d'un bloc, choisissez **VHDL File** → **Architecture**, et contrôlez que le language soit défini sur **VHDL 2008**. Sur la page suivante, **Architecture** correspond au nom de la vue (un bloc peut avoir différents contenus) et **Entity** au nom du bloc (mainDecoder par exemple).

3.1.1 Extend

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
01	S
10	B
11	J

Fig. 3. – Table de commande du bloc extend

3.1.2 ALU

L'ALU réalise les fonctions arithmétiques et logiques suivant le signal **ALUSrc** selon la table suivante :

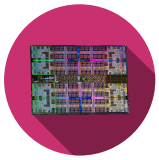
ALUControl	Operation
000	**add**
001	**substract**
010	**AND**
011	**OR**
101	**Set Lesser Than**
Others	-

Fig. 4. – Table de commande du bloc ALU

3.1.3 ALUOp

Afin d'éviter les doublons, $\mathrm{op}_{[6:0]}$ n'est pas repris dans le bloc **aluDecoder**. A la place, **mainDecoder** génère le signal **ALUOp**, plus petit, qui regroupe plusieurs types d'instructions travaillant de la même manière, selon un code arbitraire. 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 suivant présente cette idée d'unification et liste les cas spéciaux pour le décodage:



ALUOP	Funct3	Op_5 / $\mathrm{funct7}_5$	instr	$\mathrm{ALUControl}_{[2:0]}$
00	---	--	LW, SW	000 (add)
01	---	--	BEQ	001 (sub)
10	000	00, 01, 10	ADD, ADDI	000 (add)
10	000	11	SUB	001 (sub)
10	010	--	SLT, SLTI	101 (slt)
10	110	--	OR, ORI	010 (or)
10	111	--	AND, ANDI	011 (and)

Fig. 5. – Table de décodage du bloc ALUDecoder

4 Simulation

Sous le dossier **Simulation** se trouvent les codes utilisés pour simuler et déployer le système, en trois formats :

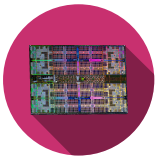
- **.s** : code assembleur, en clair
- **.bin** : code compilé; peut être analysé à l'aide d'outils comme Ghidra ou de désassembleurs en ligne
- **.txt** : version compilée compatible avec la mémoire de la FPGA pour le déploiement

Le code est compilé à l'aide du programme **HEIRV32-ASM**, disponible avec sa documentation sous le dossier du même nom.

Le testeur **heirv32_sc_tb** utilise le code **codesim.bin** lors de la simulation:

```
# Base code :
  # riscvtest.s
  # Sarah.Harris@unlv.edu
  # David_Harris@hmc.edu
  # 27 Oct 2020

main:
  addi x2, x0, 5      # x2 = 5
  addi x3, x0, 12     # x3 = 12
  addi x7, x3, -9     # x7 = 12 - 9 = 3
  or   x4, x7, x2     # x4 = (3 or 5) = 7
  and  x5, x3, x4     # x5 = (12 AND 7) = 4
  add x5, x5, x4      # x5 = 4 + 7 = 11
  beq x5, x7, end     # shouldn't be taken
  slt x4, x3, x4      # x4 = (12 < 7) = 0
  beq x4, x0, around  # should be taken
  addi x5, x0,0       # shouldn't execute
around:
  slt x4, x7, x2      # x4 = (3 < 5) = 1
  add x7, x4, x5      # x7 = (1 + 11) = 12
  sub x7, x7, x2      # x7 = (12 - 5) = 7
  sw x7, 84(x3)       # [96] = 7
  lw x2, 96(x0)       # x2 = [96] = 7
  add x9, x2, x5      # x9 = (7 + 11) = 18
```



```
  jal x3, end          # jump to end, x3 = 0x44
  addi x2, x0,1        # shouldn't execute
end:
  add x2, x2, x9       # x2 = (7 + 18) = 25
  sw x2, 0x20(x3)      # [100] = 25
done:
  beq x2, x2, main     # infinite loop
```

4.1 Analyse

Analysez dans un premier temps le code **codesim.s** afin d'en comprendre le contenu. Est-ce un bon candidat pour tester votre système ?

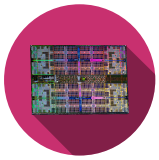
Déterminez le temps nécessaire pour effectuer la boucle du programme une fois, sachant que le système est cadencé à **50 MHz** pour les boards EBS3, et **66 MHz** pour les boards EBS2.

Le simulateur est réglé par défaut sur EBS3. Pour le modifier, ouvrez le bloc **heirv32_sc_tb** et faîtes un clic droit sur le bloc **heirv32_sc_tester** → **Change Default View** → **testEBS2**.

Lancez la simulation et confirmez le fonctionnement du système.

Répondez aux questions suivantes:
1. Quelles sont les faiblesses de cette architecture single-cycle ?
2. Déterminez l'instruction la plus longue et retracez son chemin.
3. Quelle est la vitesse de clock maximale envisageable ?

Les temps de chaque sous-parties du système sont donnés dans le top-level.



5 Déploiement (facultatif)

5.1 EBS3

Lors du déploiement sur la FPGA, le système sélectionne le code **code_sc.*** comme source.

Analysez et comprenez le code en question. Que devrait-il arriver ?

La librairie **Board** contient le top level nécessaire au déploiement. Ouvrez-le et analysez le circuit : comment fonctionnent **en_step_n** et **en_full_n** ?

Le fichier **Board/concat/car-labs.lpf** contient la définition et emplacements des pins du système. Le branchement de l'électronique se fait selon l'image suivante :

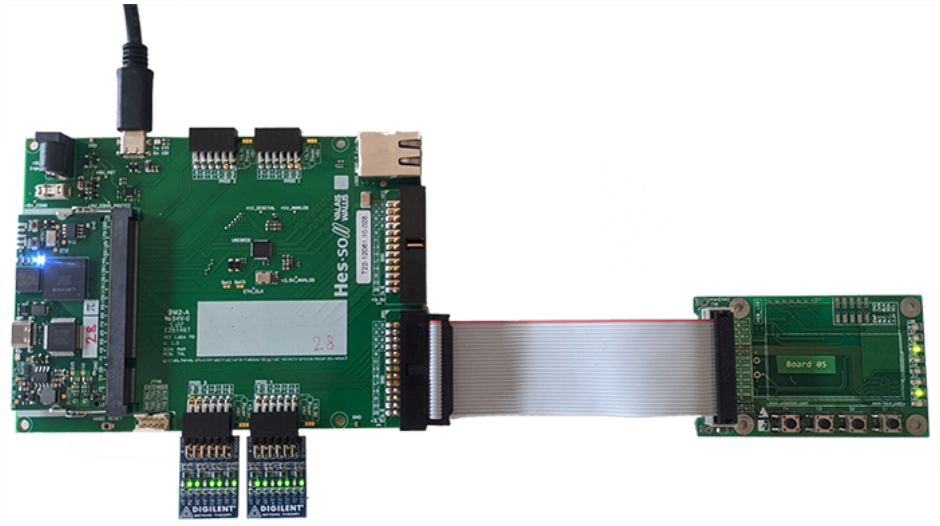
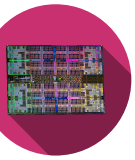


Fig. 6. – Branchement du matériel

Déployez et testez le système sur la FPGA.

Au démarrage du système, l'entrée **en** de HEIRV32SC est à 0 !

Référez-vous au document **BoardLFE5u-25F.pdf** pour la synthèse et le flash par JTAG de la board.



A Single-cycle architecture