Compte Rendu DM1

BONANCIO SKORA Lucas Eduardo

December 2024

1 Gestion de projet et tests des modules

En lieu d'utiliser le logiciel Quartus, j'ai fait mon devoir en utilisant les logiciels GHDL et GTkwave. Cela veut dire que pour compiler et simuler mon projet, il Faut utiliser ces logiciels. Le projet entier est aussi hébergé dans le dépôt git https://github.com/LucasEBSkora/projet-SoC.

En place de vérifier les testbenchs des modules individuels manuellement, j'ai créé des tests capables de vérifier automatiquement si les tests ont réussi. Le testbench de nom `nom_tb` imprime `"Testbench <nom> succesful!"` s'il a réussi et `"Testbench <nom> failed!"` s'il y a eu des problèmes, précédé par des messages qui décrivent les valeurs inattendues.

Pour être plus facile d'utiliser, j'ai fait un fichier make avec des commandes pour construire le processeur et exécuter les *testbenchs*. Pour exécuter un testbench nommé `testbench/<nom>_tb*.vhd` et voir les formes d'onde du circuit, vous pouvez faire `make TESTBENCH=nom`. Pour juste voir si le test automatique a réussi, faites les commandes "make TESTBENCH=nom test". Pour vérifier si tous les tests des modules ont réussi, vous pouvez faire exécuter le script testall.sh.

Pour voir le résultat de l'exécution d'un programme, vous pouvez changer le chemin PROGRAM_FILE dans le fichier `testbench/instruction_test_tb_.vhd` pour le fichier que vous voulez exécuter. Pour simuler le programme, utilisez la commande `make TESTBENCH=instruction_test`.

2 Exécution du programme store_01.s

Pour vérifier que mon processeur exécute bien le programme `store_01.s`, je vais simuler à la main les effets de son exécution (Table 1) et après comparer avec le graphe généré avec gtkwave (qui est gardé dans les fichiers `traces/dm1_test_sw.gtkw` et `traces/dm1_test_sw.vcd` et peut être visualisé avec la commande `gtkwave traces/dm1_test_sw.gtkw`) montré dans l'image 1.

Table 1: Simulation à la main du programme store_01.s

Adresse instruction	Commande RISC-V	Résultat
0	addi x1,x0,10	x1←10
4	addi x5,x0,0	x5←0
8	sw x1,0(x5)	*0←10
12	lw x2,0(x5)	x2←10
16	lw x3,0(x5)	x3←10
20	lw x4,0(x5)	x4←10
24	lw x6,0(x5)	x6←10
28	addi x1,x0,20	x1←20
32	addi x5,x0,1	x5←1
36	sw x1,1(x5)	*0←20
40	lw x2,1(x5)	x2←20
44	lw x3,1(x5)	x3←20
48	lw x4,1(x5)	x4←20
52	lw x6,1(x5)	x6←20
56	addi x1,x0,30	x1←30
60	addi x5,x0,2	x5←2
64	sw x1,2(x5)	*4←30
68	lw x2,2(x5)	x2←30
72	lw x3,2(x5)	x3←30
76	lw x4,2(x5)	x4←30
80	lw x6,2(x5)	x6←30
84	addi x7,x0,1	x7←1
88	lw x2,1(x7)	x2←20
92	lw x3,1(x7)	x3←20
96	lw x4,1(x7)	x4←20
100	lw x6,1(x7)	x6←20
104	lw x2,3(x7)	x2←30
108	lw x3,3(x7)	x3←30
112	lw x4,3(x7)	x4←30
116	lw x6,3(x7)	x6←30

Dans l'image 1, `addr_instr` est l'adresse de l'instruction courante, `clk` est le signal d'horloge, `we` est le *write enable* de la mémoire de données, `addr` est

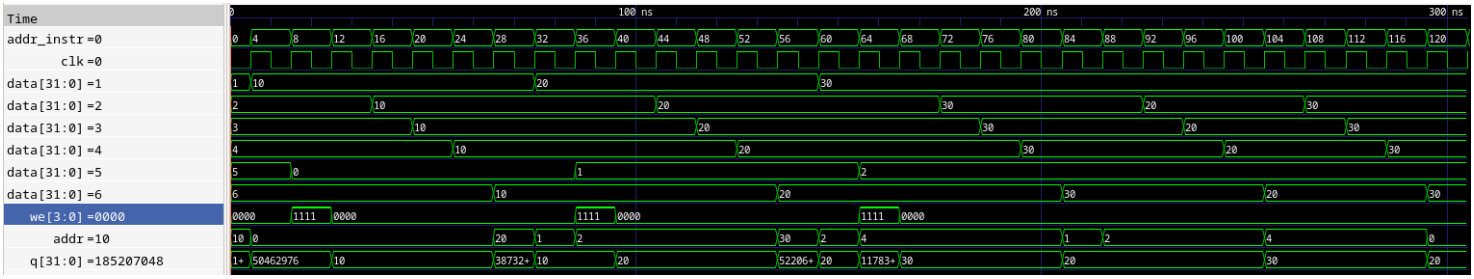

Figure 1: Simulation avec ghdl et mon processeur du programme store_01.s

l'adresse courante pour lecture/écriture dans la mémoire de données, `q` est la sortie de la mémoire et les signaux appelés `data` sont les valeurs des registres de r1 à r7, en ordre. On note que le résultat d'une instruction est visible que dans le cycle d'horloge suivante, donc le résultat de l'instruction 0 doit être vérifié dans les moments où addr_instr = 4.

Si on compare les résultats des instructions dans la table avec les valeurs des registres et la sortie de la mémoire dans la figure, on voit que les deux sont d'accord. Dans le moment où l'adresse change de 8 à 12, on voit le premier `sw`, et dans la transition de 12 à 16, le premier `lw`. On peut voir aussi que si l'adresse d'écriture/lecture n'est pas un multiple de 4, l'adresse est arrondie au multiple de 4 inférieur le plus proche. Cela est visible par exemple dans les transitions entre 36-40 ("$*2 \leftarrow 20$") et entre 40-44 ("$r2 \leftarrow *2$").

3 Exécution du programme store_02.s

La même stratégie sera utilisée pour montrer que mon processeur exécute correctement le programme `store.s`, avec la Table 2 et l'image 2 qui montre les signaux gardés dans les fichiers `traces/dm1_test_sbsh.gtkw` et `traces/dm1_test_sbsh.vcd`.

Il peut paraître que quelques instructions ont des résultats non intuitifs. Par exemple, il semblerait peut-être plus naturel que le résultat de l'instruction sb x3,1(x5) (addresse 88) avec x3=0x01020304 et *x5=0x12345604 aura le résultat *x5=0x12340304. Pourtant, l'instruction `sb` prend toujours les bits moins significatifs du registre d'origine, même si on n'écrit pas dans le bit moins significatif du mot de la RAM. En d'autres mots, si on fait sb x3,1(x5) avec n'importe quelle valeur de x5 (même s'il n'est pas multiple de 4), on va écrire l'octet moins significatif de x3 dans l'adresse (1+x5) de la RAM. La même logique s'applique à la commande `sh`.

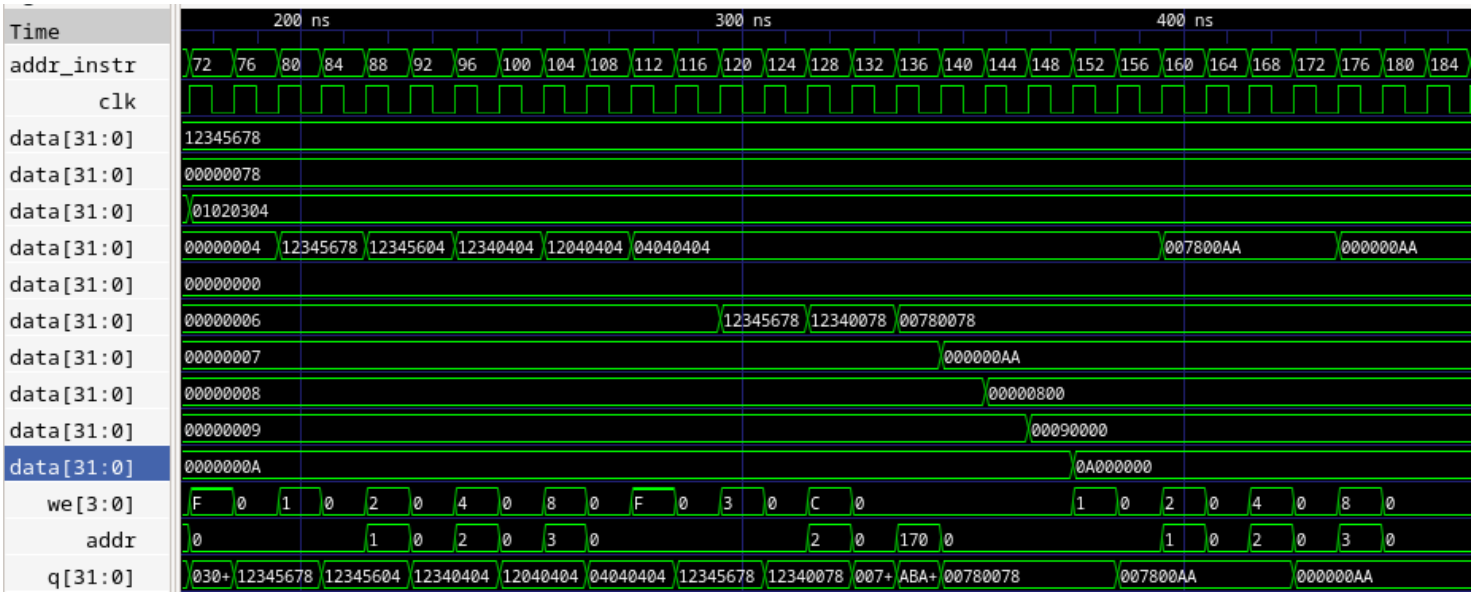

Figure 2: Simulation avec ghdl et mon processeur du programme store_02.s

Les signaux montrés dans la figure 2 sont les mêmes que ceux de la figure 1, sauf que ici les signaux data sont les valeurs des registres de x1 à x10. Comme les instructions jusqu'à 72 n'utilisent pas les instructions sw, sh et sb, je ne vais pas les montrer pour améliorer la légibilité.

Si on compare les signaux de la figure avec la table, on voit que les instructions ont eu l'effet attendu. Dans les transitions entre les instructions dans les adresses 80-84, 88-92, 96-100 et 104-108, nous pouvons voir que l'instruction sb va stocker l'octet correct (le moins significatif) dans la position correcte, et les

transitions entre les adresses 152-156 et 168-172 montrent la même chose pour l'instruction sh.

4 Conclusion

Avec les tests automatisés et la vérification de l'exécution des programmes cités, je suis convaincu de la correction de mon implantation du processeur. Ce devoir et le module, comme un tout, ont été très intéressants pour comprendre le fonctionnement des ordinateurs en général et de l'architecture RISC-V spécifiquement.

Table 2: Simulation à la main du programme store_02.s

Adresse instruction	Commande RISC-V	Résultat
0-36	plusiers	x1←0x12345678 x2←0x00000078
40	addi x5,x0,0	x5←0
44-68	plusiers	x3←0x01020304
72	sw x1,0(x5)	*0←0x12345678
76	lw x4,0(x5)	x4←0x12345678
80	sb x3,0(x5)	*0←0x12345604
84	lw x4,0(x5)	x4←0x12345604
88	sb x3,1(x5)	*0←0x12340404
92	lw x4,0(x5)	x4←0x12340404
96	sb x3,2(x5)	*0←0x12040404
100	lw x4,0(x5)	x4←0x12040404
104	sb x3,3(x5)	*0←0x04040404
108	lw x4,0(x5)	x4←0x04040404
112	sw x1,0(x5)	*0←0x12345678
116	lw x6,0(x5)	x6←0x12345678
120	sh x2,0(x5)	*0←0x12340078
124	lw x6,0(x5)	x6←0x12340078
128	sh x2,2(x5)	*0←0x00780078
132	lw x6,0(x5)	x6←0x00780078
136	addi x7,x0,0x000000AA	x7←0x000000AA
140	slli x8,x8,8	x8←0x00000800
144	slli x9,x9,16	x9←0x00090000
148	slli x10,x10,24	x10←0x0A000000
152	sb x7,0(x5)	*0←0x007800AA
156	lw x4,0(x5)	x4←0x007800AA
160	sb x8,1(x5)	*0←0x007800AA
164	lw x4,0(x5)	x4←0x007800AA
168	sb x9,2(x5)	*0←0x000000AA
172	lw x4,0(x5)	x4←0x000000AA
176	sb x10,3(x5)	*0←0x000000AA
180	lw x4,0(x5)	x4←0x000000AA