

Architecture Matérielle :
Projet — *Conception d'un mini-processeur*

Justin BABONNEAU

2 janvier 2023

Introduction

Table des matières

1	TD 5	4
1.1	Exercice 1 - Exécution du premier programme	4
1.2	Exercice 2 - Gestion complète de l'addition et de la soustraction	7
1.3	Exercice 3 - Entrées/sorties et gestion du saut <code>jeq</code>	10
1.4	Exercice 4 - Gestion de la mémoire et des autres sauts conditionnels	12
1.5	Exercice 5 - Améliorations diverses	16
2	ALU (Arithmetic Logic Unit)	18
2.1	Bits de contrôle :	18
2.1.1	Zero	18
2.1.2	Carry / Overflow	18
2.1.3	Sign	18
3	Bits de contrôle globaux	19
4	Autres bits de contrôles	20
5	Changements apportés au compilateur	21
6	Codes de tests	22

1 TD 5

1.1 Exercice 1 - Exécution du premier programme

Pour le moment, le processeur fourni ne supporte que les instructions `ldi` et `addi`. Faire appel à toute autre instruction conduit à un comportement non spécifié. Le but de cet exercice est d'ajouter le support des sauts inconditionnels, afin de pouvoir simuler l'exécution du programme suivant :

```
ldi r1, 42
addi r0, r1, 17
end: jmp end
```

1.1 Donner la valeur de chaque registre à la fin de l'exécution de ce programme.

A la fin de l'exécution de ce programme, on a :

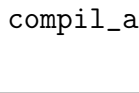
registre	<i>r0</i>	<i>r1</i>
valeur	59	42

1.2 Traduire à la main le code assembleur en langage machine, sachant que la première instruction sera placée à l'adresse 0000.

On traduit le code assembleur en langage machine (binaire) :

instruction	adresse	op	flags	rd	rs rt / imm5
<code>ldi r1, 42</code>	0000	000	01	001	00101010 (imm8)
<code>addi r0, r1, 17</code>	0001	010	00	000	001 (rs) 10001 (imm5)
<code>end: jmp end</code>	0002	111	00 (imm13(high ₁))	000 (imm13(high ₂))	00000000 (imm13(low))

1.3 Compiler le code assembleur avec `digcomp`. Comparer le contenu des fichiers ainsi produits à la réponse de la question précédente.



compil_addi.png

FIGURE 1 – Compilation du programme cité ci-dessus avec Digcomp.

Etant donné que la partie `.hi` contient les 8 premiers bits de la ligne et que la partie `.lo` contient les 8 derniers, on a (en traduisant les résultats hexadécimaux précédents en binaire) :

adresse	.hi	.lo
0000	000 01 001 (0x09)	00101010 (0x2a)
0001	010 00 000 (0x40)	001 10001 (0x31)
0002	111 00 000 (0xe0)	00000010 (0x02)

Les valeurs obtenues correspondent à celles trouvées précédemment.

FIGURE 2 – Résultat renvoyé pour le `.hi`FIGURE 3 – Résultat renvoyé pour le `.lo`

1.4 Charger les fichiers obtenus dans Diglog, et déterminer le chemin suivi par les données lors de l'exécution de l'instruction `ldi`.

Note : Pour faire une exécution pas à pas, on utilisera un générateur en guise d'horloge, sur lequel on pourra cliquer afin de passer au front (montant ou descendant) suivant.

Lors de l'instruction `ldi`, on lit sur le front montant la valeur 42 grâce aux imm (`0b00101010`) et sur le front descendant, on écrit dans le registre `r0` cette valeur.

On a maintenant cette horloge :

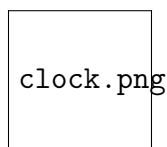


FIGURE 4 – Horloge pas à pas.

1.5 Avancer d'un cycle, et déterminer le chemin suivi par les données lors de l'exécution de l'instruction `addi`.

Lors de l'instruction `addi`, on lit sur le front montant la valeur du registre `r0` (c'est-à-dire 42 ou `0b00101010`) et sur le front descendant, on additionne (avec l'ALU en mode addition) cette valeur avec 17 (`0b10001`) et on l'écrit dans le registre `r1`.

1.6 Avancer à nouveau d'un cycle, et identifier comment récupérer l'adresse à laquelle il faudra être après le saut.

Pour récupérer l'adresse à laquelle il faudra être après le saut, il faut que l'on détermine l'écart relatif entre la position actuelle du pointeur courant (PC) et la position à laquelle on veut aller. C'est pourquoi il faut une ALU en mode soustracteur.

1.7 Déterminer le rôle des bits de contrôle *write_reg*, *arg2_imm* et *res_imm*, puis compléter le tableau suivant (mettre X si la valeur du bit de contrôle n'a pas d'influence) :

instruction	<i>do_jump_abs</i>	<i>write_reg</i>	<i>arg2_imm</i>	<i>res_imm</i>	<i>do_sub</i>
nop	0	0	X	X	0
ldi	0	1	X	1	0
addi	0	1	1	0	0
subi	0	1	1	0	1
add	0	1	0	1	0
sub	0	1	0	1	1
jmp	1	0	X	X	0

1.8 Modifier le processeur afin de gérer l'instruction de saut jmp. Utiliser pour cela le bit de contrôle *do_jump_abs* (le bit *do_jcc* servira plus tard pour les sauts conditionnels).

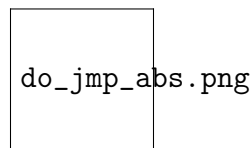


FIGURE 5 – Bit de contrôle *do_jump_abs*.

J'ai ajouté le bit de contrôle *do_jump_abs*.

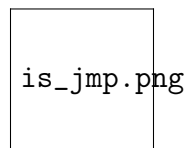


FIGURE 6 – Bit de contrôle *is_jump* (seule la partie absolue importe pour l'instant).

J'ai également ajouté le bit de contrôle *is_jump*.

J'ai aussi ajouté l'arrivée de ce bit de contrôle comme bit de contrôle du compteur de programme.

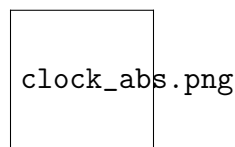


FIGURE 7 – Ajout du bit de contrôle *do_jump_abs* au compteur de programme.

1.2 Exercice 2 - Gestion complète de l'addition et de la soustraction

2.1 Donner la liste des instructions dont l'exécution nécessite d'utiliser l'ALU en tant que soustracteur.

Les instructions suivantes ont besoin de l'ALU en tant que soustracteur :

- les instructions de soustraction : `subi`, `sub`
- les instructions de lecture en RAM : `st`, `ld`;
- les sauts conditionnels : `jeq`, `jle`, `jlt`, `jne`.

2.2 Modifier le processeur afin d'obtenir la bonne valeur pour le bit de contrôle `do_sub`. Vérifier que le processeur gère désormais aussi l'instruction `subi`.

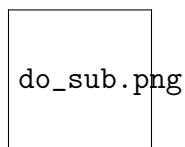


FIGURE 8 – Bit de contrôle `do_sub`

Le processeur gère bien l'instruction `subi`.

2.3 Modifier le banc de registre afin de pouvoir lire les valeurs de deux registres (pas forcément différents) à chaque cycle.

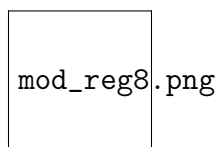


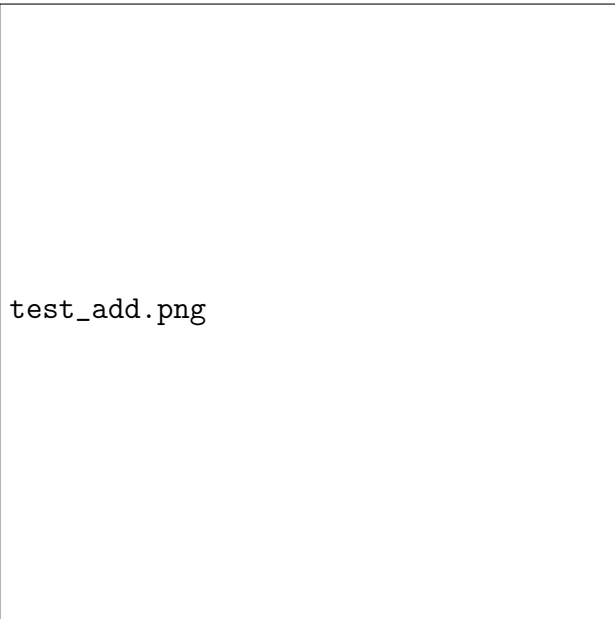
FIGURE 9 – Ajout de la seconde rangée de multiplexeurs sur `reg8` pour lire la valeur d'un deuxième registre

2.4 Pour chaque instruction, déterminer le nombre de lectures de registres à effectuer, ainsi que les bits contenant les numéros de registres à lire.

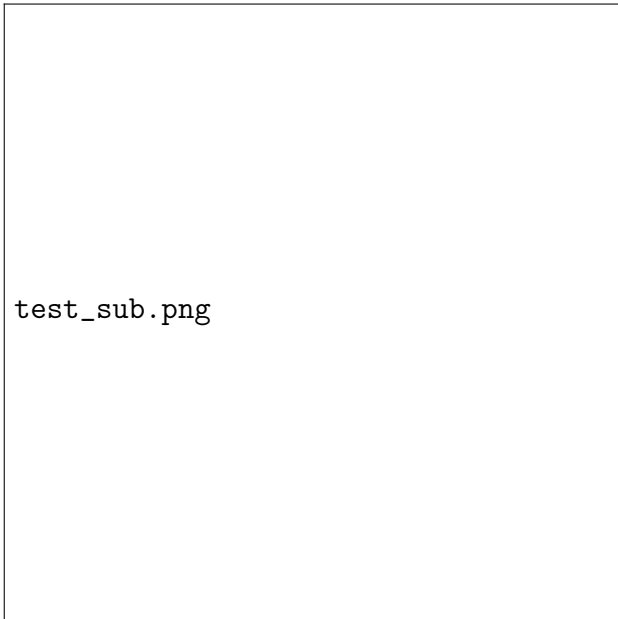
instr	lecture(s)	bits pertinents
nop	0	X
ldi	1	rd
not	1	rd
lsr	2	rd, rs
or	3	rd, rs, rt
and	3	rd, rs, rt
addi	2	rd, rs
add	3	rd, rs, rt
subi	2	rd, rs
sub	3	rd, rs, rt
muli	2	rd, rs
mul	3	rd, rs, rt
st	2	rd, rs
ld	2	rd, rs
out	1	rd
in	1	rd
jr	2	rd, rs
jeq	2	rd, rs
jle	2	rd, rs
jlt	2	rd, rs
jne	2	rd, rs
jmp	0	X

2.5 Ajouter le support des instructions `add` et `sub`, et proposer un code assembleur de test. Vous devrez sûrement modifier le calcul des bits de contrôle introduits dans l'exercice précédent.

Les instructions `add` et `sub` sont désormais supportées.



test_add.png



test_sub.png

FIGURE 10 – Test de l'instruction `add`.

FIGURE 11 – Test de l'instruction `sub`.

Les bits de contrôle n'ont pas dû être changés (car on avait déjà prévu les cas correspondant à ces deux instructions).

1.3 Exercice 3 - Entrées/sorties et gestion du saut jeq

Afin de pouvoir procéder à de vrais tests, il nous manque deux choses :

- un coté interactif pour accélérer/faciliter les tests, c'est-à-dire la possibilité de saisir des entrées au clavier et d'afficher des résultats sur un écran ;
- notre première instruction de saut conditionnel, afin d'augmenter significativement l'expressivité dans les programmes codés en assembleur.

Pour le premier point, le fichier *io.lgf* fournit déjà un clavier et un écran. Il est possible de récupérer une touche saisie au clavier (fils *kb0* à *kb7*) à condition de positionner le bit de contrôle *in* à 1. De plus, il est possible d'afficher le caractère dont le code ASCII est la valeur *RD* (fils *RD0* à *RD7*) à condition de positionner le bit de contrôle *out* à 1.

3.1 Faire en sorte que les bits de contrôle *in* et *out* reçoivent la bonne valeur.

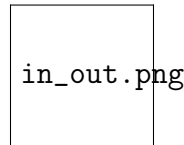


FIGURE 12 – Gestion des bits de contrôle *in* et *out*.

3.2 Tester le clavier et l'écran. Que se passe-t-il si on récupère les données du clavier alors qu'aucune touche n'a été frappée ? Et si de nombreuses touches ont été frappées depuis la dernière récupération de touche ?

Si l'on récupère les données du clavier alors qu'aucune touche n'a été frappée, on obtient 255 (0b11111111). Si de nombreuses touches ont été frappées, la plus ancienne est remplacée par la touche nouvellement pressée. Le *buffer* (tampon) du clavier fonctionne donc comme une file (First In First Out).

3.3 Rappeler comment on peut tester que deux valeurs entières sont égales. Modifier l'ALU afin d'avoir une nouvelle sortie, nécessaire à la réalisation du test.

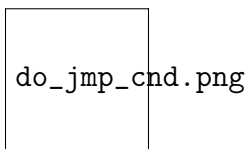
Pour tester que deux valeurs sont égales, on peut vérifier que leur différence est nulle.

On a ajouté un bit **Zero** qui permet de savoir si la valeur en sortie de l'ALU est nulle.

3.4 Afin de gérer les instructions de saut conditionnel, nous allons utiliser le bit de contrôle *do_jcc*, qui vaudra 1 si et seulement si l'instruction courante est de type saut conditionnel et qu'il convient d'effectuer le saut en question.

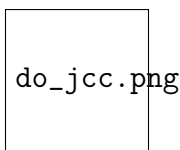
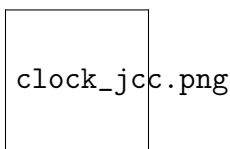
Faire en sorte que ce bit de contrôle reçoive la bonne valeur lors de l'exécution d'un *jeq*.

J'ai décidé de renommer le bit de contrôle en *do_jmp_cnd* (car j'utilise *do_jcc* quand je fais effectivement [la condition est également vérifiée] le saut conditionnel).

FIGURE 13 – Bit de contrôle *do_jump_cnd*

3.5 Modifier le processeur afin de mettre correctement à jour le pointeur d'instruction PC lors de l'exécution d'une instruction `jeq`.

En plus de l'ajout de la question précédente, on a ajouté un bit de contrôle *do_jcc*, l'arrivée d'un *do_jcc* comme bit de contrôle à l'horloge.

FIGURE 14 – Bit de contrôle *do_jcc*.FIGURE 15 – Ajout du bit de contrôle *do_jcc* au compteur de programme.

3.6 Tester le bon fonctionnement des sauts conditionnels dans le cas d'un saut :

1. en avant (vers une adresse plus grande que la valeur courante dans PC) ;
2. en arrière (vers une adresse plus petite que la valeur courante dans PC).

Les deux codes suivants fonctionnent à présent.

3.7 Écrire un code assembleur qui récupère les touches réellement saisies par l'utilisateur, les affichent, et s'arrête dès que l'utilisateur a appuyé sur la touche *Entrée* (*Cr*, de code ASCII 13).

3.8 Écrire un code assembleur qui récupère un entier $n \in [1, 9]$ et un caractère c , puis qui affiche à l'écran un carré de taille n et composé de caractères c .



FIGURE 16 – Code de test du saut dans le passé.

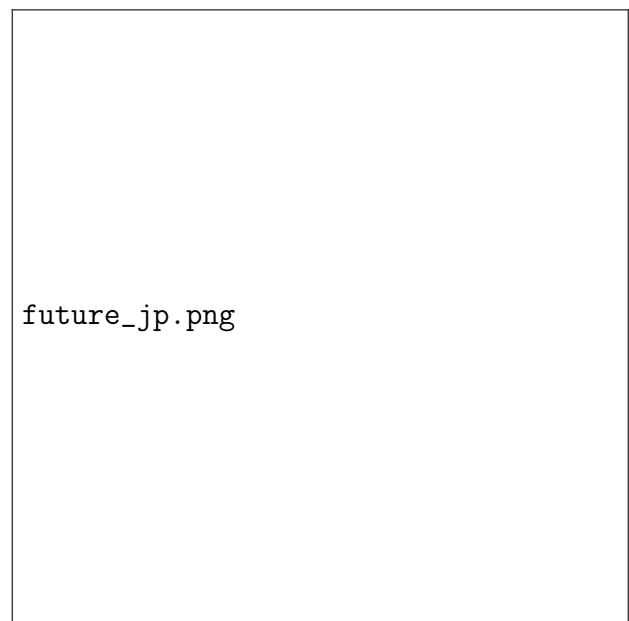


FIGURE 17 – Code de test du saut dans le futur.

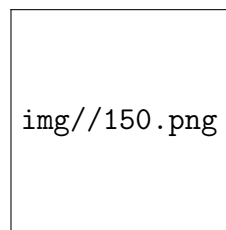


FIGURE 18 – Code assembleur.

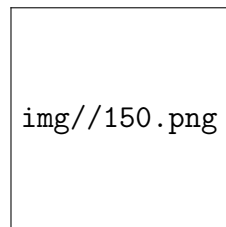


FIGURE 19 – Code assembleur.

1.4 Exercice 4 - Gestion de la mémoire et des autres sauts conditionnels

4.1 Ajouter le support des instructions `ld` et `st`. Pour cela, il faudra identifier le parcours que les données devront suivre, ajouter des multiplexeurs au besoin, et utiliser au minimum un nouveau bit de contrôle : *write_mem*.

Note : Pour l’instant, on ne traitera pas la partie *imm5* de ces instructions.

On a ajouté les bits de contrôle suivants :

- *mem_access* qui vaut 1 lors des instructions `st` et `ld` (et 0 sinon) ;
- *read_mem* qui vaut 1 lors de l’instruction `ld` (et 0 sinon) ;
- *write_mem* qui vaut 1 lors de l’instruction `st` (et 0 sinon).

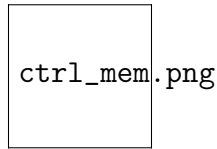


FIGURE 20 – Bits de contrôle liés à la mémoire

4.2 Écrire un code assembleur qui récupère les touches saisies par l'utilisateur et les stocke en mémoire jusqu'à la saisie de la touche *Entrée*, puis qui réécrit les données ainsi récupérées à l'écran en ordre inverse.

Exemple : La saisie de "123<Entrée>" donnera donc lieu à l'affichage de "321".

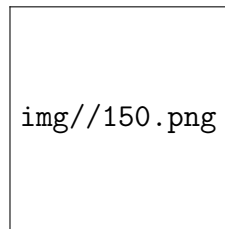


FIGURE 21 – Code assembleur.

4.3 Modifier l'ALU afin d'avoir 3 nouvelles sorties, correspondant aux drapeaux S (signe), C (carry), et O (overflow).

Remarquons d'abord que *Carry* et *Overflow* sont confondus puisqu'avoir une retenue revient à dépasser la capacité de l'ALU donc par exemple seuls *Carry* et *Sign* sont à implémenter effectivement.

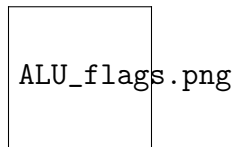


FIGURE 22 – Ajout des drapeaux à l'ALU.

4.4 Démontrer que, pour a et b deux entiers signés sur 8 bits différents, $a \leq b$ est équivalent à $S = O$, où S et O sont obtenus suite au calcul de $b - a$ par l'ALU.

Soient a et b deux entiers signés sur 8 bits. Alors :

$$\exists (a_i)_{0 \leq i \leq 7} \in \{0, 1\}^7 \mid a = \sum_{i=0}^7 2^i a_i$$
$$\exists (b_i)_{0 \leq i \leq 7} \in \{0, 1\}^7 \mid b = \sum_{i=0}^7 2^i b_i$$

Notons S et O les bits de Signe (*Sign*) et de Retenue (*Overflow*) renvoyés par l'ALU après le calcul de $b - a$.

Montrons que $a \leq b \iff S = O$

\implies Supposons que $a \leq b$.

- Comme $a \leq b$, $b - a \geq 0$ d'où $\boxed{S = 0}$.
- Comme $a \leq b$, il existe $k_0 \in \llbracket 0; 7 \rrbracket$ tel que :
 - $\forall i \in \llbracket k_0 + 1; 7 \rrbracket, a_i = b_i$ (I)
 - $a_{k_0} = 0$ (II)
 - $b_{k_0} = 1$ (III)

Par conséquent, $c := a - b = a + \overline{\text{xor}(b, 255)} + 1$ va vérifier :

- $c_{k_0} = C_{k_0-1}$ en notant C_i la retenue d'ordre $i \in \llbracket 0; 7 \rrbracket$ dans le calcul de $b - a$.
- $\forall i \in \llbracket k_0 + 1; 7 \rrbracket, c_i = 1$

En effet, on a :

$$\begin{aligned}
 c_{k_0} &= a_{k_0} - b_{k_0} + C_{k_0-1} \\
 &= a_{k_0} + \overline{b_{k_0}} + C_{k_0-1} \\
 &\stackrel{II \wedge III}{=} 0 + 0 + C_{k_0-1} \\
 c_{k_0} &= C_{k_0-1} \text{ et donc } C_{k_0} = 0
 \end{aligned}$$

$$\begin{aligned}
 \forall i \in \llbracket k_0 + 1; 7 \rrbracket, c_i &= a_i + \overline{b_i} + C_i \\
 &\stackrel{I}{=} a_i + \overline{a_i} + 0 \\
 c_i &= 1
 \end{aligned}$$

Puisque que l'on n'a pas dépassé la limite de 8 bits, on a donc $\boxed{O = 0}$.

Ainsi, on a $S = 0 = O$ et finalement $\boxed{S = O}$.

\impliedby Supposons que $S = O$.

Procédons par une disjonction de cas :

- Supposons que $S = 0$.
Par conséquent, d'après l'hypothèse, $O = 0$. Donc : $b - a \geq 0$ sans avoir dépassé c'est-à-dire que $\boxed{a \leq b}$.
- Supposons que $S = 1$. D'après l'hypothèse, on a $O = 1$ autrement dit $b - a \leq 0$ et a dépassé la capacité de l'ALU ce qui revient à dire que le signe est faussé et que $b - a \geq 0$ ou encore $\boxed{a \leq b}$.

4.5 Utiliser le résultat précédent et ce qui a déjà été à la section précédente pour ajouter le support des instructions `jlt`, `jle` et `jne`.

Le support de ces instructions a été ajouté à la question **3.4**.

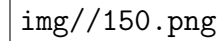
A square box with a thin black border. Inside the box, the text "img//150.png" is centered in a monospaced font.

FIGURE 23 – Code assembleur.

4.6 Utiliser le code assembleur qui effectue à l'aide d'une boucle la multiplication par 10 de la valeur stockée dans *r0*.

Comment peut-on faire ce calcul beaucoup plus efficacement ?

4.7 Écrire un code assembleur qui récupère les touches saisies par l'utilisateur et les stocke en mémoire jusqu'à la saisie de la touche *Entrée*, puis qui réécrit à l'écran les données ainsi récupérées mais en majuscules.

Exemple : La saisie de "Abc1<*Entrée*>" donnera donc lieu à l'affichage de "ABC1".

1.5 Exercice 5 - Améliorations diverses

Certaines questions de cet exercice impliquent de compléter le code source du compilateur.

5.1 Améliorer l'ALU et modifier le chemin de données de façon à supporter les instructions logiques (catégorie 001).

J'ai ajouté à l'ALU le support de `mul / muli`, des opérations logiques telles que `not`, `lsr`, `or` et `and`.

Pour cela, j'ai décidé de changer les bits de contrôle en les faisant correspondre aux bits d'opération et de drapeaux de l'instruction et par conséquent, j'ai eu besoin d'un multiplexeur dans les cas où l'opération nécessite une somme ou une addition mais n'en est pas une à proprement parler.

On a donc :

instruction	op	flags
<code>not</code>	001	00
<code>lsr</code>	001	01
<code>or</code>	001	10
<code>and</code>	001	11
<code>add / addi</code>	010	0X
<code>sub / subi</code>	010	1X
<code>mul / muli</code>	011	0X
<code>jeq</code>	110	00
<code>jle</code>	110	01
<code>jlt</code>	110	10
<code>jne</code>	110	11

5.2 Écrire un code qui lit un entier saisi au clavier.

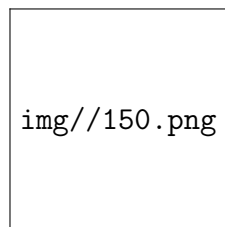


FIGURE 24 – Code assembleur.

5.3 Écrire un code qui effectue la division par 10 de la valeur stockée dans `r0`.

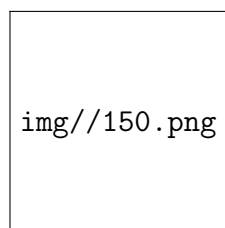


FIGURE 25 – Code assembleur.

5.4 Écrire un code qui lit deux entiers au clavier, puis affiche le résultat de leur multiplication (modulo 256) à l'écran.

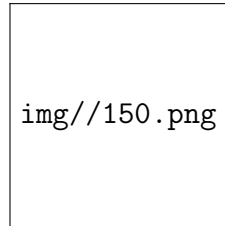


FIGURE 26 – Code assembleur.

5.5 Compléter le support des instructions `ld` et `st` afin de gérer la partie `imm5` de l'instruction.

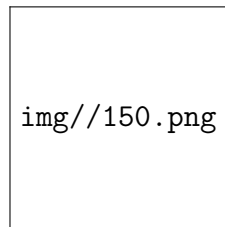


FIGURE 27 – Code assembleur.

5.6 Ajouter le support de l'instruction `jr`.

J'ai ajouté un bit de contrôle `jr`.

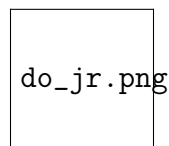


FIGURE 28 – Bit de contrôle `do_jr`

J'ai également ajouté un bit de contrôle `do_jr`, l'arrivée d'un `do_jcc` comme bit de contrôle à l'horloge.

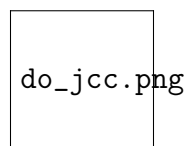


FIGURE 29 – Bit de contrôle `do_jcc`.

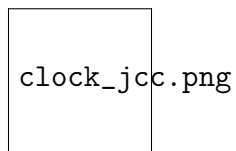


FIGURE 30 – Ajout du bit de contrôle *do_jcc* au compteur de programme.

2 ALU (Arithmetic Logic Unit)

2.1 Bits de contrôle :

2.1.1 Zero

2.1.2 Carry / Overflow

2.1.3 Sign

3 Bits de contrôle globaux

instr	op	flags	zb	ma	mb	lg	mo
nop	000	00	X	X	X	X	X
ldi	000	01	X	X	X	X	X
	000	10	X	X	X	X	X
	000	11	X	X	X	X	X
not	001	00	1	1	X	X	0
lsr	001	01	0	0	0	0	0
or	001	10	0	1	1	1	1
and	001	11	0	0	0	1	0
addi	010	00	0	0	0	0	0
add	010	01	0	0	0	0	0
subi	010	10	0	1	0	0	1
sub	010	11	0	1	0	0	1
muli	011	00	X	X	X	X	X
mul	011	01	X	X	X	X	X
compi	011	10	0	1	0	0	1
comp	011	11	0	1	0	0	1
st	100	00	0	0	0	0	0
ld	100	01	0	0	0	0	0
out	100	10	X	X	X	X	X
in	100	11	X	X	X	X	X
jr	101	00	X	X	X	X	X
	101	01	X	X	X	X	X
	101	10	X	X	X	X	X
	101	11	X	X	X	X	X
jeq	110	00	0	1	0	0	1
jle	110	01	0	1	0	0	1
jlt	110	10	0	1	0	0	1
jne	110	11	0	1	0	0	1
jmp	111	00	X	X	X	X	X
jmp	111	01	X	X	X	X	X
jmp	111	10	X	X	X	X	X
jmp	111	11	X	X	X	X	X

4 Autres bits de contrôles

5 Changements apportés au compilateur

6 Codes de tests

Conclusion