C. Mouilleron
christophe.mouilleron@ensiie.fr

Description du jeu d'instructions

Introduction

Diglog fournit un bloc SRAM de 8Ko où chaque octet est adressable. Ainsi, on peut :

— fournir une adresse sur 13 bits ($2^{13} = 8192 = 8$K) pour récupérer la valeur (1 octet = 8 bits) précédemment stockée à cette adresse,

— fournir une adresse sur 13 bits et une donnée sur 8 bits à écrire à cette adresse.

Afin de profiter pleinement du bloc SRAM, nous allons travailler avec un pointeur d'instruction sur 13 bits nommé PC dans la suite.

Pour avoir un jeu d'instructions simple, il convient d'imposer une certaine régularité dans la façon de représenter les instructions. En particulier, nous imposons ici que toutes les instructions machines soient de même longueur. Utiliser un codage sur 8 bits forcerait à faire trop de restrictions sur la nature des instructions et le nombre d'arguments autorisés. Nous opterons donc pour un codage sur 16 bits, décrit à la section 1. Le détail des instructions utilisables sera donné à la section 2.

Pendant que vous réaliserez dans Diglog le processeur gérant le jeu d'insctruction proposé, vous serez amener à faire des tests avec des vrais programmes. Il serait cependant bien trop fastidieux de générer à la main et pour chaque programme les fichiers binaires à charger dans les blocs SRAM de Diglog. Par conséquent, nous fournissons un compilateur dont l'utilisation est expliquée à la section 3. Enfin, la section 4 fournit une description sommaire de la syntaxe à utiliser pour écrire un code assembleur accepté par le compilateur.

1 Les différents types d'encodages pour les instructions

Les instructions sont codées sur 16 bits, composés des éléments suivants :

op (3 bits) catégorie de l'instruction (opération arithmétique, opération mémoire, etc.)

flags (2 bits) drapeaux pour désigner une instruction dans sa catégorie

rd/rs/rt (3 bits) numéro d'un registre

immX (X bits) constante entière sur X bits, signée ou non-signée suivant le contexte.

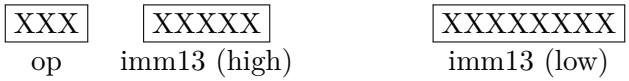
Suivant l'instruction, le nombre et la nature des arguments varient. Cela nous amène à considérer 4 types d'encodages.

Type 1a :

Type 1b :

Type 2 :

Type 3 :



Pour le type 1a, les deux derniers bits ne servent pas et sont donc mis à 0.

Pour les numéros de registres, rs et rt désignent en général des registres lus, alors que rd désigne le registre dans lequel sera écrit le résultat de l'instruction. Il y a cependant quelques exceptions.

2 Liste complète des instructions machine

La liste complète des instructions machine est donnée par le tableau suivant.

op	flags	nom	type	effet	conditions
000	00	nop	2	aucun	
	01	ldi	2	rd ← imm8	
001	00	not	1b	rd ← complément à 1 de rs	
	01	lsr	1b	rd ← rs décalé de imm5 bit à droite, rempli avec 0	$0 \leq \text{imm5} \leq 31$
	10	or	1a	rd ← OU bit à bit entre rs et rt	
	11	and	1a	rd ← ET bit à bit entre rs et rt	
010	00	addi	1b	rd ← rs + imm5	$0 \leq \text{imm5} \leq 31$
	10	subi	1b	rd ← rs − imm5	
	01	add	1a	rd ← rs + rt	
	11	sub	1a	rd ← rs − rt	
011	00	muli	1b	rd ← rs × imm5 mod 256	$0 \leq \text{imm5} \leq 31$
	01	mul	1a	rd ← rs × rt mod 256	
100	00	st	1b	MEM[rs+imm5] ← rd	$-16 \leq \text{imm5} \leq 15$
	01	ld	1b	rd ← MEM[rs+imm5]	
	10	out	2	affiche le caractère contenu dans rd à l'écran	
	11	in	2	rd ← caractère issu du clavier	
101	00	jr	1b	PC ← 256 × rd + rs	
110	00	jeq	1b	si rd = rs : PC ← PC + imm5 + 1	$-16 \leq \text{imm5} \leq 15$
	01	jle	1b	si rd ≤ rs : PC ← PC + imm5 + 1	
	10	jlt	1b	si rd < rs : PC ← PC + imm5 + 1	
	11	jne	1b	si rd ≠ rs : PC ← PC + imm5 + 1	
111	-	jmp	3	PC ← imm13	

3 Utilisation et limitations du compilateur

Si votre code source est dans le fichier `code.asm`, vous pouvez le compiler en lançant la commande :

```
$ digcomp code.asm
```

En cas de succès, le compilateur produira les fichiers `code.asm.hi` et `code.asm.lo` qu'il faudra charger dans les deux blocs SRAM (le `.hi` à gauche, et le `.lo` à droite) du processeur dans Diglog.

4 Syntaxe pour la programmation assembleur

Le code assembleur attendu par le compilateur doit être composé d'une séquence d'instructions. Chaque ligne du code doit contenir 0 ou 1 instruction, et peut commencer par un label qui permettra de désigner cette ligne dans une instruction de saut. Ainsi, une ligne du programme est valide si elle est de l'une des formes suivantes :

— vide

— uniquement composée d'un label, suivi de `:`

— uniquement composée d'une instruction assembleur

— composée d'un label, suivi de `:` puis d'une instruction assembleur.

Le symbole # et tout ce qui suit jusqu'au prochain saut de ligne est considéré comme un commentaire et est donc ignoré par le compilateur. En particulier, une ligne commençant par # est donc considérée comme vide.

Un label est un nom pour désigner une ligne. Il doit commencer par une lettre, et contenir uniquement des lettres, des chiffres ou le caractère `_`. Par ailleurs, un label ne peut pas être le nom d'une instruction ou d'un registre.

Une instruction est composée du nom de l'instruction (cf section 2), suivi d'un blanc (espace, tabulation), suivi de la liste des arguments séparés par des virgules. Le registre numéro i est désigné par `r`i. Les constantes immédiates seront données au format décimal, sauf pour les instructions de saut où on donnera le label de la ligne de destination à la place.

Limitations et pièges à éviter

1. Pour le moment, les instructions des catégories 001 (logique), 011 (ALU autre) et 101 (autre) ne sont pas gérées par le compilateur.

2. Certaines instructions n'utilisent jamais leurs derniers arguments. Dans ce cas, il convient de ne pas donner ces arguments. On écrira donc :

```
nop
in r0
```

et pas

```
nop r0, 0
in r0, 0
```

3. La partie `imm5` pour les instructions `ld` et `st` n'est pas gérée. Le compilateur mettra toujours 0 en guise de valeur immédiate, et il est impératif de ne pas fournir de valeur dans le code assembleur.

4. Attention à l'ordre des arguments pour l'instruction `st`. Le code

```
st r0, r1
```

correspond à l'opération de lecture de la valeur stockée dans `r0` afin de l'écrire en mémoire à l'adresse stockée dans `r1`.

Sucres syntaxiques

1. On peut utiliser `add` (resp. `sub`) à la place de `addi` (resp. `subi`).

2. On peut écrire `j` au lieu de `jmp` et `je` au lieu de `jeq`.

3. On peut utiliser le mot-clé `mov` pour réaliser un certain nombre d'opérations mémoire :

syntaxe	traduction
`mov rX, cste`	`ldi rX, cste`
`mov rX, rY`	`addi rX, rY, 0`
`mov rX, (rY)`	`ld rX, rY`
`mov (rX), rY`	`st rY, rX`