Annexe A
Documentation

A.1 Références

Le logiciel LOGISIM que nous utilisons en TP pour les circuits est disponible sur la page :

`http://www.cburch.com/logisim/index.html`

C'est un outil à vocation pédagogique, qui permet de dessiner et de simuler des circuits logiques simples.

Le LC-3 est un processeur développé dans un but pédagogique par Yale N. Patt et J. Patel dans [*Introduction to Computing Systems : From Bits and Gates to C and Beyond*, McGraw-Hill, 2004]. Des sources et exécutables sont disponibles à l'adresse :

`http://highered.mcgraw-hill.com/sites/0072467509/`

A.2 Mémos pour la programmation du LC3

Description du LC-3

La mémoire et les registres : La mémoire du LC-3 est organisée par mots de 16 bits, avec un adressage également de 16 bits (adresses de $(0000)_H$ à $(FFFF)_H$).

Le LC-3 comporte 8 registres généraux 16 bits : R0, ..., R7. R6 est réservé pour la gestion de la pile d'exécution, et R7 pour stocker l'adresse de retour des routines. Il comporte aussi des registres spécifiques 16 bits : PC (*Program Counter*), IR (*Instruction Register*), PSR (*Program Status Register*) qui regroupe plusieurs drapeaux.

Le PSR contient trois bits N, Z, P, indiquant si la dernière valeur (regardée comme le code d'un entier naturel en complément à 2 sur 16 bits) placée dans l'un des registres, R0, ..., R7 est négative strictement pour N, nulle pour Z, ou positive strictement pour P.

Les instructions :

syntaxe	**action**	**NZP**
NOT DR,SR	DR <- not SR	*
ADD DR,SR1,SR2	DR <- SR1 + SR2	*
ADD DR,SR1,Imm5	DR <- SR1 + SEXT(Imm5)	*
AND DR,SR1,SR2	DR <- SR1 and SR2	*
AND DR,SR1,Imm5	DR <- SR1 and SEXT(Imm5)	*
LEA DR,label	DR <- PC + SEXT(PCoffset9)	*
LD DR,label	DR <- mem[PC + SEXT(PCoffset9)]	*
ST SR,label	mem[PC + SEXT(PCoffset9)] <- SR	
LDR DR,BaseR,Offset6	DR <- mem[BaseR + SEXT(Offset6)]	*
STR SR,BaseR,Offset6	mem[BaseR + SEXT(Offset6)] <- SR	
BR[n][z][p] label	Si (cond) PC <- PC + SEXT(PCoffset9)	
NOP	No Operation	
RET	PC <- R7	
JSR label	R7 <- PC ; PC <- PC + SEXT(PCoffset11)	

Directives d'assemblage :

`.ORIG adresse`	Spécifie l'adresse à laquelle doit commencer le bloc d'instructions qui suit.
`.END`	Termine un bloc d'instructions.
`.FILL valeur`	Réserve un mot de 16 bits et le remplit avec la valeur constante donnée en paramètre.
`.BLKW nombre`	Cette directive réserve le nombre de mots de 16 bits passé en paramètre.
`;`	Les commentaires commencent par un point-virgule.

Les interruptions prédéfinies : `TRAP` permet de mettre en place des *appels système*, chacun identifié par une constante sur 8 bits, gérés par le système d'exploitation du LC-3. On peut les appeler à l'aide des macros indiquées ci-dessous.

instruction	macro	description
`TRAP x00`	`HALT`	termine un programme (rend la main à l'OS)
`TRAP x20`	`GETC`	lit au clavier un caractère ASCII et le place dans R0
`TRAP x21`	`OUT`	écrit à l'écran le caractère ASCII placé dans R0
`TRAP x22`	`PUTS`	écrit à l'écran la chaîne de caractères pointée par R0
`TRAP x23`	`IN`	lit au clavier un caractère ASCII, l'écrit à l'écran, et le place dans R0

Constantes : Les constantes entières écrites en hexadécimal sont précédées d'un `x` (en décimal elles peuvent être précédées d'un # optionnel) ; elles peuvent apparaître comme paramètre : des instructions du LC3 (opérandes immédiats, attention à la taille des paramètres), des directives `.ORIG`, `.FILL` et `.BLKW`.

Codage des instructions LC3

On donne ici un tableau récapitulatif du codage des instructions LC3.

syntaxe	action	NZP	codage															
			opcode				arguments											
			F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
NOT DR,SR	DR ← not SR	*	1	0	0	1	DR			SR			1	1	1	1	1	1
ADD DR,SR1,SR2	DR ← SR1 + SR2	*	0	0	0	1	DR			SR1			0	0	0	SR2		
ADD DR,SR1,Imm5	DR ← SR1 + SEXT(Imm5)	*	0	0	0	1	DR			SR1			1	Imm5				
AND DR,SR1,SR2	DR ← SR1 and SR2	*	0	1	0	1	DR			SR1			0	0	0	SR2		
AND DR,SR1,Imm5	DR ← SR1 and SEXT(Imm5)	*	0	1	0	1	DR			SR1			1	Imm5				
LEA DR,label	DR ← PC + SEXT(PCoffset9)	*	1	1	1	0	DR			PCoffset9								
LD DR,label	DR ← mem[PC + SEXT(PCoffset9)]	*	0	0	1	0	DR			PCoffset9								
ST SR,label	mem[PC + SEXT(PCoffset9)] ← SR		0	0	1	1	SR			PCoffset9								
LDR DR,BaseR,Offset6	DR ← mem[BaseR + SEXT(Offset6)]	*	0	1	1	0	DR			BaseR			Offset6					
STR SR,BaseR,Offset6	mem[BaseR + SEXT(Offset6)] ← SR		0	1	1	1	SR			BaseR			Offset6					
BR[n][z][p] label	Si (cond) PC ← PC + SEXT(PCoffset9)		0	0	0	0	n	z	p	PCoffset9								
NOP	No Operation		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
RET	PC ← R7		1	1	0	0	0	0	0	1	1	1	0	0	0	0	0	0
JSR label	R7 ← PC; PC ← PC + SEXT(PCoffset11)		0	1	0	0	1	PCoffset11										

Traduction de programmes en langage d'assemblage

Il vous est demandé de toujours commencer par écrire un pseudo-code pour le programme ou la routine demandé, en faisant apparaître les registres que vous allez utiliser pour effectuer vos calculs, et en ajoutant tous les commentaires utiles. Vous traduirez ensuite votre pseudo-code vers le langage d'assemblage du LC3 en utilisant les règles de traduction suivantes.

Traduction d'un « bloc if » : On suppose que la condition d'entrée dans le bloc consiste simplement en la comparaison du résultat d'une expression arithmétique `e` à 0. Dans ce qui suit, `cmp` désigne une relation de comparaison : $<$, $\leq$, $=$, $\neq$, $\geq$, $>$. On note `!cmp` la relation contraire de la relation `cmp`, traduite dans la syntaxe des bits `nzp` de l'instruction `BR`. Si par exemple `cmp` est $<$, alors `BR!cmp` désigne `BRpz` (pour « positive or zero »).

```
/* En pseudo-code */
if e cmp 0 {
  corps du bloc
}
```

```
; En langage d'assemblage du LC3
    evaluation de e
    BR!cmp endif    ; branchement sur la sortie du bloc
    corps du bloc
endif:
```

Traduction d'un bloc « if-else » : On ajoute simplement un label `else`.

```
/* En peudo-code */
if e cmp 0 {
  corps du bloc 1
}
else {
  corps du bloc 2
}
```

```
; En langage d'assemblage du LC3
    evaluation de e
    BR!cmp else    ; branchement sur le bloc else
    corps du bloc 1
    BR endif       ; branchement sur la sortie du bloc
else:
    corps du bloc 2
endif:
```

Traduction d'une « boucle while » :

```
/* En peudo-code */
while e cmp 0 {
  corps de boucle
}
```

```
; En langage d'assemblage du LC3
loop:
     evaluation de e
     BR!cmp endloop  ; branchement sur la sortie de boucle
     corps de boucle
     BR loop         ; branchement inconditionnel
endloop:
```

Quelques « astuces » à connaître :

— Initialisation d'un registre à 0 : `AND Ri,Ri,#0`

— Initialisation d'un registre à une constante `n` (représentable en complément à 2 sur 5 bits) :

```
AND Ri,Ri,#0
ADD Ri,Ri,n
```

— Calcul de l'opposé d'un entier (on calcule le complément à 2 de Rj dans Ri) :

```
NOT Ri,Rj
ADD Ri,Ri,#1
```

— Multiplication par 2 de Rj, résultat dans Ri : `ADD Ri,Rj,Rj`

— Copie du contenu de Rj dans Ri : `ADD Ri,Rj,#0`