DIU – Enseigner l'informatique au lycée — Université de Nantes
Architecture des ordinateurs — 2019/2020

Travaux pratiques

Ces travaux pratiques portent sur la simulation d'un ordinateur RISC à l'aide du logiciel développé par Peter Higginson. Le TP consiste à prendre en main le simulateur puis à comprendre le lien entre instruction en assembleur et code machine. Un exercice permet aussi d'aborder la problèmatique de la traduction d'un algorithme en un programme en assembleur. Pour approfondir ces notions, on peut continuer en étudiant comment les notions telles qu'une instruction conditionnelle ou répétitive, une variable locale, un tableau, un appel de fonction, un appel système, etc se traduisent en assembleur.

Durée : 1 heure 30

Peter Higginson a créé plusieurs simulateurs d'ordinateur écrit en Javascript et utilisable avec un navigateur : une première version appelée LMC (Little Man Computer) et qui reposait sur un modèle proposé en 1965 (c'est-à-dire avec peu de registres), un second simulateur pour un modèle RISC (Reduced Instruction Set Computing) semblable aux processeurs ARM que l'on trouve dans les tablettes et smartphones et une version plus "simple" du simulateur RISC qui peut être utilisée pour les programmes des examens d'informatique de l'AQA pour le *A level*. Dans ce TP, nous utiliserons la version RISC que l'on trouve à l'adresse suivante :

`http://www.peterhigginson.co.uk/RISC/`

Il existe des versions partiellement traduites en français comme :

`http://www.irisa.fr/alf/downloads/puaut/simu_risc/RISC%20simulator%20by%20Peter%20Higginson.html`

Sur certains ordinateurs et certains navigateurs, l'apparence du simulateur n'est pas toujours satisfaisante à cause des tailles des polices de caractères utilisées. Edge, Chrome (ou Chromium) fonctionnent bien. Mozilla Firefox pose parfois problème. Pour ce dernier, on peut diminuer la taille de la police en cochant *Affichage→Zoom→Zoom texte seulement* et en sélectionnant *Affichage→Zoom arrière*.

1. Lancer le simulateur et charger le programme "add" en le sélectionnant dans la liste déroulante "SELECT" puis l'exécuter en cliquant sur le bouton "RUN". Lors de cette exécution, le simulateur demande 2 entiers qu'il faut fournir l'un après l'autre dans la case "Input" (penser à appuyer sur la touche *Entrée* pour valider le nombre). La somme des 2 entiers doit s'afficher dans la case "Output".
2. La simulation peut être accélérée ou ralentie dans la liste déroulante "OPTION" ou bien en utilisant les boutons "«" et "»" lorsque le programme tourne.
3. Repérer les différentes parties de l'architecture de von Neumann (unité de contrôle, unité arithmétique et logique, mémoire et entrées/sorties). Attention, comme dans tous les processeurs, la mémoire est ici divisée en 2 parties : la mémoire principale externe au processeur et les registres du processeur faisant partie soit de l'unité de contrôle (PC, IR, FLAGS) soit de l'unité arithmétique et logique.

4. Repérer le registre PC (Program Counter) servant à désigner la prochaine instruction qui va être exécutée et le registre IR (Instruction Register) contenant le code de l'instruction en cours d'exécution.
5. Trouver dans la documentation du simulateur le jeu d'instructions du processeur (bouton HELP).
6. Tester les autres programmes proposés et essayer de comprendre ce qu'ils font.

Exercice 1 (De l'assembleur au code machine)

L'architecture de von Neumann place les programmes et les données dans la mémoire principale (contrairement à l'architecture de type Harvard où le programme et les données sont dans des espaces mémoires séparés). Cela permet au programme de se modifier lui-même (méthode fortement déconseillée) mais aussi au système d'exploitation ou aux librairies de manipuler les programmes (charger un programme, charger ou décharger une librairie, etc). La représentation des données élémentaires en mémoire est assez standard sur la plupart des ordinateurs (voir le cours sur la représentation de l'information). Par contre, chaque processeur possède son propre **jeu d'instructions** avec sa représentation mnémonique en langage d'assemblage et sa représentation numérique en code machine. Par exemple, pour le programme d'addition du simulateur RISC, on a la correspondance suivante :

Adresse	Assembleur	Binaire	Hexadécimal	Décimal
0	`INP R0,2`	0111000100000010_2	7102_{16}	28930
1	`INP R1,2`	0111000100010010_2	7112_{16}	28946
2	`ADD R2,R1,R0`	0110000010001000_2	6088_{16}	24712
3	`OUT R2,4`	0111000110100100_2	$71a4_{16}$	29092
4	`HLT`	0000000000000000_2	0000_{16}	0

Dans cette table, de gauche à droite, on a l'adresse en mémoire de l'instruction, l'instruction en langage assembleur, le code machine en binaire (16 bits), en héxadécimal (4 chiffres) et en décimal.

Pour passer du langage assembleur au code machine, on utilise un **assembleur**. Ces programmes traduisent les instructions mais aussi enregistrent les étiquettes désignant des parties du code (pour faire des sauts ou des appels à des procédures) ou des données (variables globales ou tableaux stockés en mémoire principale). Les programmes traduits en code machine pourront ensuite être chargés en mémoire principale par le système d'exploitation puis être exécutés en positionnant le registre PC (*Program Counter*) sur la première instruction du programme.

> Trouver dans la documentation du simulateur RISC (dans la page sur le jeu d'instruction) comment on peut coder en code machine (d'abord en binaire puis en hexadécimal) le programme assembleur suivant :
>
> ```
> INP R1,2
> INP R2,2
> MUL R2,#100
> ADD R4,R2,R1
> OUT R4,5
> HLT
> ```

Le simulateur RISC comporte un assembleur. Aussi, on peut trouver la solution de cet exercice en tapant le programme en assembleur dans la partie gauche du simulateur et en le validant.

S'il n'y a pas d'erreur, le code machine se trouve alors dans les premières cases de la mémoire principale.

Exercice 2 (Calcul d'une moyenne)

Le but de l'exercice consiste à écrire un programme qui calcule la moyenne d'une suite de nombres positifs donnés par l'utilisateur. La lecture des nombres se termine lorsque l'utilisateur entre un nombre négatif. L'algorithme est le suivant :

s ← 0
n ← 0
Lire x
Tant que x ≥ 0 faire
    s ← s + x
    n ← n + 1
    Lire x
Fin tant que
m ← s/n
Afficher m

Le simulateur RISC ne permet pas d'utiliser des nombres avec virgule flottante. Il ne travaille que sur des entiers sur 16 bits (nombres signés entre -32768 et 32767 ou non-signés entre 0 et 65535). Aussi, pour simplifier, le calcul de la moyenne utilisera une division entière.

Traduire cet algorithme en un programme assembleur pour le simulateur RISC. Il faudra choisir un registre pour chacune des variables. La boucle *Tant que* sera traduite en un test sur la valeur de x suivi d'un saut conditionnel après la fin de la boucle et un retour au début du tant que avec un saut inconditionnel comme ci-dessous (la variable x est dans R4) :

```
                  // Début du tant que
Debut  CMP  R4,#0 // On compare R4 et la valeur 0
       BLT  Fin   // Si R4 < 0, sortir du tant que
       ...        // Corps du tant que
       BRA  Debut // Retour au début du tant que
Fin    ...        // Suite du programme après le tant que
```

Exercice 3 (Décodage du code machine)

Les programmes exécutés sont codés dans le code machine spécifique au processeur. Pour le simulateur RISC, chaque instruction est codé en un mot de 16 bits (contenu dans une case de la mémoire principale). Le processeur exécute les instructions du programme une par une en utilisant le registre **PC** (*Program Counter*). La première étape de l'unité de contrôle du processeur consiste à récupérer le code de l'instruction à exécuter en le stockant dans le registre IR (*Instruction Register*). L'unité de contrôle doit ensuite décoder le code pour comprendre ce que doit faire le processeur.

Cet exercice consiste à retrouver les instructions en assembleur correspondant au code machine suivant en jouant le rôle de l'unité de contrôle (dans la table, on a la représentation des codes en décimal non-signé, en décimal signé, en héxadécimal

sur 4 chiffres hexadécimaux et binaire sur 16 bits) :

Adresse	Non-signé	Signé	Hexadécimal	Binaire
0	29042	29042	7172_{16}	0111000101110010_2
1	20234	20234	$4f0a_{16}$	0100111100001010_2
2	61228	-4308	$ef2c_{16}$	1110111100101100_2
3	0	0	0000_{16}	0000000000000000_2

Travaux pratiques

— CORRECTION —

Corrigé exercice 1 (De l'assembleur au code machine)

Adresse	Instruction	Binaire	Hexadécimal	Décimal
0	`INP R1,2`	0111000100010010_2	7112_{16}	28946
0	`INP R2,2`	0111000100100010_2	7122_{16}	28962
1	`MUL R2,#100`	0101001001100100_2	5264_{16}	21092
2	`ADD R4,R2,R1`	0110000100010001_2	6111_{16}	24849
2	`OUT R2,5`	0111000111000101_2	$71C5_{16}$	29125
4	`HLT`	0000000000000000_2	0_{16}	0

Corrigé exercice 2 (Calcul d'une moyenne)

C'est déjà un programme complexe. La variable s est dans le registre R2, n dans R3, x dans R4 et m dans R5 :

```
        MOV  R2,#0     // s = 0
        MOV  R3,#0     // n = 0
        INP  R4,2      // lire x
                       // début du tant que
Debut   CMP  R4,#0     // on compare x et 0
        BLT  Fin       // si x < 0 continuer après le tant que
        ADD  R2,R2,R4  // s = s+x
        ADD  R3,#1     // n = n+1
        INP  R4,2      // lire x
        BRA  Debut     // retour au début du tant que
                       // suite du programme après le tant que
Fin     MOV  R5,R2     // m = s
        DIV  R5,R3     // m = m/n
        OUT  R5,4      // Afficher m
        HLT
```

Corrigé exercice 3 (Décodage du code machine)

Le décodage du code machine est souvent fastidieux. Dans le cas du simulateur RISC, il vaut mieux partir de la représentation binaire du code machine. On commence par regarder la partie gauche du code pour trouver le type d'instruction. Ensuite, on interprète les bits restant pour les arguments de l'instruction.

— Pour la première instruction, 0111000101110010_2, les 4 premiers bits 0111_2 indiquent une “instruction 7”. Il faut voir le sous-code de l’instruction qui est ici 011100010_2. Cela correspond à l’instruction `INP Rd,address`. Le numéro d’un registre général est codé sur 3 bits ce qui donne ici 111_2 soit le registre R7. Il reste 4 bits 0010_2 qui correspondent à l’adresse 2 d’entrée/sortie. L’instruction est donc **INP R7,2**.
— Pour 0100111100001010_2, la partie instruction est 01001_2 qui correspond à `UDV Rd,#immediate`. Cette instruction divise le contenu d’un registre par une constante (on suppose que les nombres ne sont pas signés). Les 3 bits 111_2 suivants indiquent le registre R7. La constante est décrite sur 8 bits ($00001010_2 = 10$). L’instruction **UDV R7,#10** permet de diviser R7 par 10.
— Pour 1110111100101100_2, les 4 premiers bits 1110_2 correspondent à l’instruction *STR Rs,direct* qui stocke le contenu d’un registre dans une case mémoire dont l’adresse est donnée par le champ *direct*. Le registre est donné par les trois bits 111_2 suivant le code de l’instruction qui correspondent à R7. L’adresse de la case mémoire est donnée par les derniers bits à droite $100101100_2 = 300$. L’instruction est donc **STR R7,300**. Elle place la valeur du registre R7 dans la case mémoire 300.
— 0000000000000000_2 est l’instruction **HLT**. Dans ce cas, seuls les 5 premiers bits 00000_2 sont utiles car cette instruction n’a pas de paramètre.