N7_SN_1A

Architecture des ordinateurs - Semestre 6

TD1 - Construction d'un mini microprocesseur : mini-craps

1- Introduction

L'objectif de ce TD, et des 2 TPs qui lui sont attachés, est de réaliser, en langage shdl, un tout petit microprocesseur que l'on nommera « mini-craps ». Pour simplifier au maximum cette réalisation, on limitera les capacités de ce processeur : instructions simples et en nombre réduit. Mini-craps est une version simplifié du processeur « craps » étudié en cours.

Le rôle principal d'un microprocesseur est d'exécuter des instructions : addition, soustraction, multiplication, lecture et écriture mémoire, etc. On va illustrer cela à travers un exemple très simple écrit en langage ADA : Resultat := Variable1 + Variable2 ;

Cette instruction est traitée par le compilateur et traduite dans un premier temps en langage assembleur :

- Les variables Resultat, Variable 1 et Variable 2, de type entier, sont réservées en mémoire à des adresses que l'on va désigner par Resultat, Variable 1 et Variable2 (pour l'instant, on n'a pas besoin de connaître les valeurs exactes de ces adresses). Contrairement aux langages évolués, les noms des variables (étiquettes, labels) que nous utilisons désignent leur adresse ; et les valeurs de ces variables sont désignées par [variable]. Les […] indiquent un accès mémoire.
- Pour les raisons de simplicité évoquées dessus, mini-craps ne peut faire l'addition qu'entre registres (add rsrc1, rsrc2, rdest // rdest <- rsrc1+ rsrc2), et on doit donc lire les valeurs de Variable 1 et Variable 2 en mémoire et les copier dans des registres avant d'effectuer l'addition :
    - On a donc besoin d'une instruction de lecture mémoire qui prendra la forme : load [adresse], registre // les […] indiquent un accès mémoire à l'adresse indiquée
    - Et on impose à cette adresse d'être mise dans un registre : on a donc besoin d'une instruction qui initialise un registre avec une adresse ou une valeur numérique, qui prendra la forme : set adresse, registre ou set valeur, registre
- Le résultat de notre addition se trouvant dans un registre, on doit disposer d'une instruction d'écriture mémoire qui prendra la forme : store registre, [adresse]

Au final notre instruction initiale se traduit par la séquence suivante en langage assembleur mini-craps (on suppose disposer de registres nommés %r2, %r3, …, %r11. % précède le nom du registre pour éviter les confusions avec les noms des constantes ou des variables) :

```
set     Variable1, %r2      // adresse de Variable1
set     Variable2, %r3      // adresse de Variable2
set     Resultat, %r4       // adresse de Resultat
load    [%r2], %r5          // valeur de Variable1
load    [%r3], %r6          // valeur de Variable2
add     %r5, %r6, %r7       // %r7 <- %r5 + %r6
store   %r7, [%r4]
```

Mais ces instructions ne peuvent pas être comprises par le processeur, et elles ont besoin d'être traduites en langage machine (binaire). On fera le choix de la simplicité en codant toutes les instructions sur un mot mémoire de 32 bits (nous verrons plus tard comment cela sera fait). On traduit donc ces instructions en langage machine et on les installe en mémoire, par exemple partir de l'adresse 0 *:* ***Adresse*** ***code binaire*** dont on étudiera la structure plus loin

	0x00000000 :	0xc2000011	code binaire de la première instruction set
	0x00000001 :	0xc3000012	code binaire de la deuxième instruction set
	…		
	0x00000005 :	0x07560000	code binaire de l'instruction add
	0x00000006 :	0x97400000	code binaire de l'instruction store
		valeur	
Resultat	0x00000010 :	……	
Variable1	0x00000011 :	0x00000123	
Variable2	0x00000012 :	0x00000432	

2- Architecture générale de mini-craps (voir en parallèle mini_craps_fig.pdf)

A travers l'exemple dessus, nous voyons que notre processeur aura besoin des éléments suivants :

- Des registres qui sont nécessaires pour contenir les opérandes des instructions et certaines informations spécifiques que l'on verra plus loin.
- Une unité de calcul, ou unité arithmétique et logique (UAL) qui permettra d'exécuter les différentes opérations arithmétiques et logiques : addition, soustraction, etc.
- Une mémoire RAM qui va contenir les instructions (programme) à exécuter (code binaire), et les données du programme. Pour simplifier, on utilisera des instructions codées sur une taille fixe de 32 bits, et des données de taille fixe de 32 bits. On utilisera le module « ram » déjà utilisé au semestre 5.
- Des canaux que l'on appellera « bus » permettant d'acheminer les informations entre les modules précédents :
  - « abus » et « bbus » acheminerons les opérandes sources des instructions arithmétiques et logiques vers l'ual
  - un bus principal bi-directionnel, nommé « dbus », permettra d'acheminer les données entre les modules qui ne disposent pas de lien direct. Plusieurs entrées peuvent arriver sur ce bus, et pour éviter les courts circuits, ces entrées doivent être exclusives (une seule entrée ouverte à un instant donnée). D'où l'utilisation des 2 signaux de contrôle « dbusIn » qui permettent de sélectionner une seule entrée parmi 4 possibles :
    - pour enregistrer le résultat de l'ual dans un registre, on injecte la sortie de l'ual dans dbus (dbusIn=01) pour la présenter à l'entrée du module registre
    - pour enregistrer la donnée lue en mémoire, on injecte la sortie de la RAM dans dbus (dbusIn=10) pour la présenter à l'entrée du module registre
    - pour écrire le contenu d'un registre en mémoire, on le sort sur bbus, et on l'injecte dans dbus (dbusIn=11) pour le présenter à l'entrée de la mémoire
- D'un ensemble d'indicateurs (flags) qui indiquent l'état du résultat calculé dans l'UAL : N (négatif), Z (Zéro), V (débordement sur le bit de signe), C (retenue ou emprunt final).
- D'un séquenceur qui va gérer les différentes étapes nécessaires pour exécuter les instructions, et qui sera étudié plus loin.

3- Le module registre

Comme vu plus haut, les registres sont utilisés pour contenir des opérandes ou certaines informations spécifiques. Mini-craps sera doté de 16 registres (module reg32 réalisé au semestre 5) :

- Les registres r0 et r1 sont constants (non modifiables), et contiennent respectivement les constantes 0 et 1 (constantes utiles pour différentes opérations)
- Les registres r2, r3, …, jusqu'à r11 sont utilisés pour contenir les opérandes des instructions
- Les registres r12 et r13 sont réservés au séquenceur (non utilisables par le programmeur)
- Le registre r14 sera utilisé comme compteur ordinal (PC : Program Counter) : il contiendra l'adresse de l'instruction courante (à exécuter ou en cours d'exécution), et dont on verra l'utilité plus loin
- Le registre r15 sera utilisé comme registre instruction (IR : Instruction Register) : il contiendra le code de l'instruction qui est en cours d'exécution, et dont on verra l'utilité plus loin

Le module registres aura l'interface suivante :
registres (rst, clk, areg[3..0], breg[3..0], dreg[3..0], dataIn[31..0] : a[31..0], b[31..0], ir[31..0])

- L'entrée « areg » indique le numéro du registre que l'on souhaite lire sur la sortie « a »
- L'entrée « breg » indique le numéro du registre que l'on souhaite lire sur la sortie « b »
- L'entrée « dreg » indique le numéro du registre dans lequel on souhaite écrire l'entrée dataIn
- La sortie « IR » (r15) servira pour accéder directement au code de l'instruction courante sans passer par les sorties « a » ou « b »

En utilisant deux instances du module reg32(rst, clk, en, e[31..0] : reg[31..0]), fait au semestre 5, donner le code shdl qui permet d'instancier les registres r0, r1, r2 et r3 ; et les diriger vers les sorties « a » et « b ». Le module registres sera complété et testé en TP.

4- Le module ual

Le module ual aura l'interface suivante : ual (a[31..0], b[31..0], cmd[3..0] : s[31..0], N, Z, V, C)
On y implantera les opérations suivantes :

1. Addition et soustraction en utilisant le module adssub32 fait au semestre 5
    1. L'addition aura pour code 0000
    2. La soustraction aura le code 0001
2. Sig_extend24 qui permet d'étendre un nombre signé représenté sur les 24 bits de l'entrée « a » (a[23..0]) vers une représentation sur 32 bits sigext24[31..0]. Le code de cette opération sera 1100

La sortie N sera à 1 lorsque le résultat s[31.0] est négatif, et la sortie Z sera à 1 si le résultat est nul.
Les sorties V et C sont celles de **addsub32**.

Ecrire l'équation de sigext24[31..0] en fonction de a[23..0] en vous basant sur la formule de représentation binaire suivante :	$s = -2^{n-1}b_{n-1} + \sum_{i=0}^{n-2} 2^i b_i$

Le module ual sera complété et testé en TP.

4- Les instructions et le séquenceur

Toutes les instructions de mini-craps seront codées sur 32 bits. On utilisera les 4 bits de fort poids de ce code pour identifier les instructions, et les bits restants pour identifier leurs opérandes. Nous implanterons les instructions suivantes :

- Des instructions arithmétiques et logiques : op rs1, rs2, rdest

0 cop (3) | rdest (4) | rs1 (4) | rs2 (4) | . . . . . 16 bits libres . . . . .

  - cop (code opération) = 000 pour add (addition)
  - cop = 001 pour sub (soustraction)
  - 6 autres instructions arithmétiques et logiques pourront être ajoutées plus tard
- L'instruction set valeur24, rdest

1 1 0 0 | rdest (4) | . . . . . valeur24 . . . . .

- L'instruction load [rad1+rad2], rdest // l'adresse est une somme de 2 composantes

1 0 0 0 | rdest (4) | rad1 (4) | rad2 (4)| . . . . . 16 bits libres . . . . .

- L'instruction store rsrc, [rad1+rad2]

1 0 0 1 | rsrc (4) | rad1 (4) | rad2 (4)|. . . . . 16 bits libres . . . . .

- Pour implanter des structures de contrôle de type Si alors Sinon, Pour, …, on aura besoin d'instructions de branchement qui permettent d'effectuer un branchement à une adresse donnée si une condition donnée est vérifiée. Les instructions de branchement (b_cond adresse) auront le format binaire suivant :

1 1 1 0 | cond (4) | . . . . . déplacement24 . . . . .

  Avec déplacement24 = adresse de branchement – adresse courante

<u>Schéma d'exécution d'une instruction</u>

Lors de l'exécution d'un programme, on aura constamment besoin de connaître l'adresse de l'instruction courante pour pouvoir lire son code depuis la RAM, et on aura besoin de copier ce code dans un registre dédié pour pouvoir l'analyser et en extraire les informations nécessaires à l'exécution de l'instruction : opérandes, condition de branchement, etc.

- Le registre PC (Program Counter = r14) servira à contenir l'adresse de l'instruction courante
- Le registre IR (Instruction Register = r15) servira à contenir le code de l'instruction courante
- L'algorithme d'exécution d'un programme se présente sous la forme suivante :

  PC <- adresse de la première instruction
  Répéter
  lire le code de l'instruction courante // IR <- [PC]
  exécuter cette instruction
  passer à l'instruction suivante // PC <- PC + 1 ou PC <- adresse de branchement
  Jusqu'à Fin du programme

L'opération « exécuter cette instruction » peut nécessiter une ou plusieurs étapes en fonction de la complexité de l'instruction.

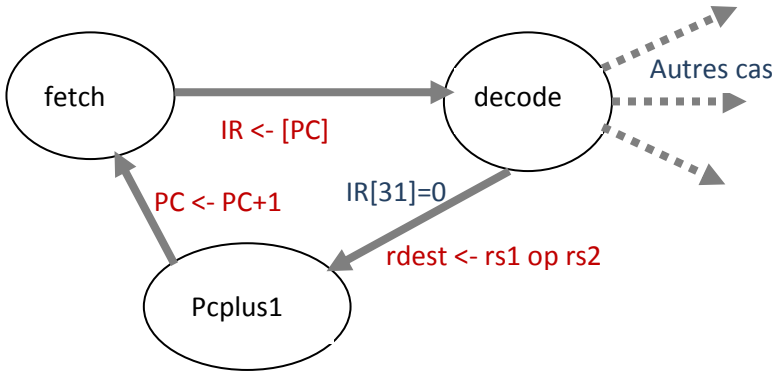
Cet algorithme s'implante sous forme d'un circuit séquentiel, que l'on peut représenter par un graphe d'états :

- L'état de départ sera appelé « fetch »
- On passe de l'état « fetch » à l'état appelé « decode » en réalisant l'opération IR <- [PC] que l'on commande de la façon suivante :
  - areg <- 1110 : le contenu de r14 (PC) mis sur abus (adresse de l'instruction courante)
  - dbusIn <- 10 : la sortie de la mémoire est injectée dans dbus
  - dreg <- 1111 : la sortie de la mémoire, passant par dbus, est enregistrée dans r15 (IR)
- A partir de l'état « decode », plusieurs possibilités se présentent :

  1- ***Instructions arithmétiques et logiques*** (add, sub, …) repérées par IR[31] = 0 : les opérandes sont présents dans les registres rs1 (IR[23..20]) et rs2 (IR[19..16].

0 cop (3) | rdest (4) | rs1 (4) | rs2 (4) | . . . . . 16 bits libres . . . . .

  - l'opération peut s'exécuter immédiatement dans l'ual : le numéro du registre rs1 sera pris dans IR[23..20] et mis sur areg pour sortir l'opérande 1 sur abus, le numéro du registre rs2 sera pris dans IR[19..16] et mis sur breg pour sortir l'opérande 2 sur bbus, et le code de l'opération IR[31..28] sera mis sur l'entrée cmd de l'ual.
  - Pour enregistrer le résultat dans rdest (IR[27..24]), on injecte la sortie de l'ual dans dbus (dbusIn = 01), et on met le numéro de registre destination (IR[27..24]) sur dreg
  - Puis, on passe à un état appelé « pcplus1 » qui doit préparer le passage à l'instruction suivante (PC <- PC + 1)



Pour simplifier l'implantation, on utilisera une bascule D par état, ce qui donne :
fetch := pcplu1 on clk set when rst
decode := fetch on clk reset when rst
pcplus1 := decode2pcplus1 on clk reset when rst
decode2pcplus1 = decode*/ir[31]

Pour commander les opérations qui vont être exécutées lors du passage d'un état à l'autre, on agira sur les 6 microcommandes « areg », « breg », « dreg », « ualCmd », « dbusIn », et « write » (pour l'écriture mémoire). On aura ainsi dans l'ordre des transitions ci-dessus :

```
areg[3..0] = fetch*''1110''      // adresse de l'instruction courante (PC=14) à lire dans la ram
           + …                   // un terme pour chaque transition
breg[3..0] = fetch*''0000''            // sortie b non utilisée lors de la lecture mémoire
           + …                   // un terme pour chaque transition
```

```
dreg[3..0] = fetch*''1111''      // code lu en mémoire enregistré dans ir (r15)
           + …                   // un terme pour chaque transition
ualCmd[3..0]  = fetch*''0000''   // ual non utilisée lors de la lecture mémoire
              + …                // un terme pour chaque transition
dbusIn[1..0]  = fetch*''10''     // sortie ram injectée dans dbus
              + …                // un terme pour chaque transition
write  = fetch*0                 // pas d'écriture en ram
       + …                       // un terme pour chaque transition
```

Les termes égaux à 0 sont neutres, mais ont été mis exprès pour vérifier que toutes les transitions ont été prises en compte.

Compléter les équations des 6 microcommandes dessus.

2- ***Instruction set*** : set valeur24, rdest

1 1 0 0 | rdest (4) | . . . . . valeur24 . . . . .

- ***Compléter le graphe pour y intégrer l'exécution de l'instruction set***
- ***Compléler le code shdl ci-dessus pour y intégrer l'exécution de cette instruction***

3- ***Instructions d'accès mémoire***

load [rad1+rad2], rdest

1 0 0 0 | rdest (4) | rad1 (4) | rad2 (4)| . . . . . 16 bits libres . . . . .

store rsrc, [rad1+rad2]

1 0 0 1 | rsrc (4) | rad1 (4) | rad2 (4)|. . . . . 16 bits libres . . . . .

Ces deux instructions nécessitent une opération commune : le calcul de l'adresse comme somme de rad1 et de rad2.

- ***Compléter le graphe pour y intégrer l'exécution des instructions load et store***
- ***Compléler le code shdl ci-dessus pour y intégrer l'ajout de nouveaux états et de nouvelles transitions***

4- ***Instruction de branchement*** : b_cond adresse

1 1 1 0 | cond (4) | . . . . . déplacement24 . . . . .

Avec déplacement24 = adresse de branchement – adresse courante (PC)

Cette instruction fonctionne selon l'algorithme suivant :

Si cond est vrai alors PC <- PC + déplacement24

Sinon PC <- PC + 1

La condition de branchement est évaluée en utilisant les indicateurs N, Z, V, et C.

On suppose disposer d'un module branch (cond[3..0], N, Z, V, C : branch_ok)

- ***Compléter le graphe pour y intégrer l'exécution de cette instruction***
- ***Compléler le code shdl ci-dessus***