N7_SN_1A

Architecture des ordinateurs - Semestre 6

BE Dettes – 3 heures

Tous le travail réalisé doit être copié dans un fichier portant votre nom et envoyé par mail à hamrouni@n7.fr

1- Instructions et séquenceur de mini-craps 6 points

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 . . . . .
- Etc.

Schéma d'exécution d'une instruction

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

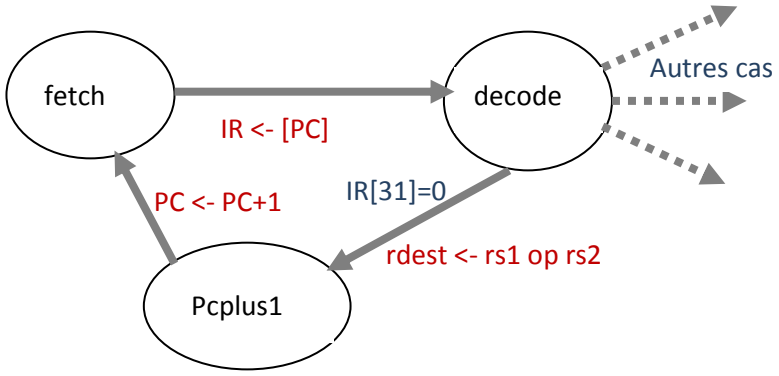
L'opération « exécuter cette instruction » peut nécessiter une ou plusieurs étapes en fonction de la complexité de l'instruction.
Cet algorithme s'implante sous forme d'un circuit séquentiel, que l'on peut représenter par un graphe d'états :

- 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 := decode2pcplus1al on clk reset when rst
decode2pcplus1al = 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
+ decode2pcplus1al*ir[23..20]
+ pcplus1 * "1110" // un terme pour chaque transition

```
breg[3..0] = fetch*''0000''                    // sortie b non utilisée lors de la lecture mémoire
           + decode2pcplus1al*ir[19..16]
           + pcplus1 * "0001"                  // r1  = 1

dreg[3..0] = fetch*''1111''          // code lu en mémoire enregistré dans ir (r15)
           + decode2pcplus1al*ir[27..24]
           + pcplus1 * "1110"

ualCmd[3..0]    = fetch*''0000''             // ual non utilisée lors de la lecture mémoire
                + decode2pcplus1al*ir[31..28]
                + pcplus1 * "0000"

dbusIn[1..0]    = fetch*''10''               // sortie ram injectée dans dbus
                + decode2pcplus1al*"01"
                + pcplus1 * "01"

write   = fetch*0          // pas d'écriture en ram sauf pour l'instruction store
        + …
```

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

- Ecrire dans « shdlsandbox » le module sequenceur(rst, clk, ir[31..16] : fetch, decode, pcplus1, areg[3..0], breg[3..0], dreg[3..0], ualcmd[3..0], dbusin[1..0], write)
- Tester ce séquenceur avec l'entrée ir = 0001 0111 0110 0101

2- ***Instruction set*** : set valeur24, rdest // rdest (32 bits) <- valeur (24 bits)

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

L'ual dispose de l'instruction sigext24 (code = 1100) : S <- A-24 étendu sur 32 bits.

- ***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***
- ***Tester le séquenceur***

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***
- ***Tester le séquenceur***

2- Programmation en assembleur 8 points

A- Dans « craps-sandbox », écrire le sous-programme « inserer » qui implante l'algorithme suivant :

```
last <- Tab[lastIndex]
index <- lastIndex
Tantque index > 0 et Tab|index - 1] > last faire
        Tab[index] <- Tab[index – 1]
        index <- index – 1
Fintantque
Tab[index] <- last
```

Les paramètres de ce sous-programme doivent être passés dans les registres r1, r2, etc.

On peut facilement voir que si les éléments d'indices 0 à lastIndex-1 sont déjà triés (ordre croissant), cet algorithme permet d'insérer l'élément d'indice lastIndex au bon endroit, et qu'il constitue la base du tri par insertion.

B- Ecrire un programme qui permet de tester ce sous-programme. Tester.

C- Ecrire le sous-programme trier_insertion, ainsi qu'un programme test, et tester.

D- Ecrire le sous-programme verifier_tableau_trie. Le tester dans le programme B.

3- Entrées/sorties & Interruptions 6 points

« Craps » et son simulateur disposent d'entrées/sorties minimalistes :

- Entrées : 16 switches (interrupteurs permettant d'entrer 0 ou 1) : La lecture des 16 switches se fait via l'adresse 0x90000000 avec l'instruction ld (load)
- Sorties : 16 leds permettant d'afficher 16 bits en parallèle : L'écriture sur les leds se fait via l'adresse 0xB0000000 avec l'instruction st (store)

« Craps » et son simulateur disposent d'une seule interruption (bouton IT), dont le traitant doit être implanté à l'adresse 1. Un programme qui utilise cette IT doit donc avoir la forme :

```
        ba prog         // 1ere instruction du programme à l'adresse 0
traitant_IT :   …..     // traitant à l'adresse 1
                reti
prog :          ….
```

A- Ecrire et tester le sous-programme afficher_leds_7_0 qui affiche une valeur (8 bits), passée en paramètre dans %r1, sur les 8 leds de faible poids et met à 0 les 8 leds de fort poids.

B- Ecrire et tester le sous-programme afficher_leds_15_8 qui affiche une valeur (8 bits), passée en paramètre dans %r1, sur les 8 leds de fort poids et met à 0 les 8 leds de faible poids.

C- Ecrire et tester un sous-programme « sleep » qui boucle durant N secondes (N passé en paramètre). Ajuster la durée de la boucle en testant sur le simulateur.

D- Ecrire et tester le programme de tirage « pile ou face ». Ce programme affiche en continue et de façon alternée un compteur1 (incrémenté toutes les secondes) sur les 8 bits de faible poids des leds, et un compteur2 (incrémenté toutes les 2 seconds) sur les 8 bits de fort poids des leds ; et gèle l'affichage durant 3 secondes lorsque l'utilisateur clique sur le bouton « IT ». ***Tester***.