N7_SN_1A

Architecture des ordinateurs - Semestre 6

TP1 & 2 - Implantation d'un mini microprocesseur : mini-craps

ATTENTION :

1- **Si vous ne disposez d'un module supposé fait au semestre 5, ou si vous n'êtes pas sûr du votre, prenez en une copie disponible sur la page moodle.**

2- **il faut respecter l'interface des modules (noms et ordre des entrées/sorties) pour pouvoir utiliser le module craps et les fichiers de test fournis.**

3- **Il est inutile de passer à l'étape suivante si l'étape courante n'est pas validée.**

Module registre

module registres (rst, clk, areg[3..0], breg[3..0], dreg[3..0], datain[31..0] : a[31..0], b[31..0], pc[31..0], ir[31..0]) // la sortie pc est ajoutée pour faciliter les tests

- 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 »

1- En utilisant deux instances du module reg32(rst, clk, en, e[31..0] : reg[31..0]) fait au semestre 5, instancier les registres r0, r1, r2 et r3; et les diriger vers les sorties « a » et « b ». On enlèvera la sortie ir[31..0] dans cette première étape. (fait en TD)
Pensez à l'intérêt d'utiliser un module décodeur fait en semestre 5.
Tester ce module en écrivant des valeurs quelconques dans r0, r1, r2 et dans r3, et en vérifiant le contenu des registres r0, r1, r2 et r3. *Temps moyen = 10 mn*

2- Compléter le module registres, et le valider en utilisant le fichier de test fourni. **Pour simplifier, on n'implantera pas les registres r8, r9, r10 et r11.** *Temps moyen = 10 mn*

Module ual

module 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 (cmd[3..0])
    2. La soustraction aura le code 0001
2. Sigextend24 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] (fait en TD). Le code de cette opération sera 1100. Il n'est pas nécessaire de faire un module pour Sigextend24.

Ecrire le module ual et le valider en utilisant le fichier de test fourni.

Temps moyen = 20 mn

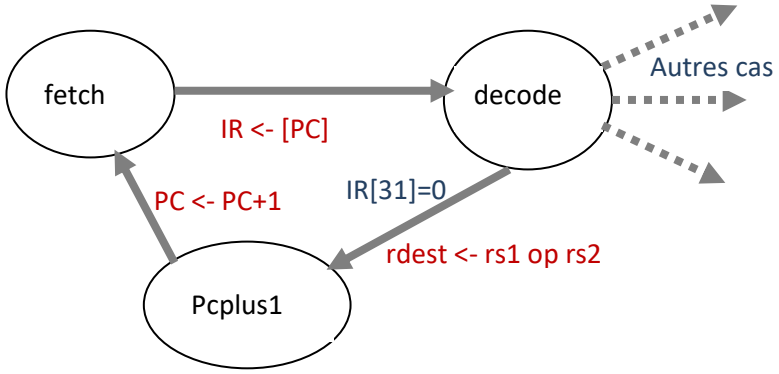
Les instructions et le séquenceur

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 suivant :



Cette implantation sera faite dans un module appelé « sequenceur » :
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, setFlags)
et sera réalisée et validée de façon incrémentale.

1- ***Instructions arithmétiques et logiques*** : op %rs1, %rs2, %rdest

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

A- Ecrire les équations des états fecth, decode, et pcplus1 ; et des microcommandes areg, breg, dreg, ualCmd, dbusIn et write. (fait en TD)

La sortie setFlags sera mise à 1 lors de l'exécution de l'opération (passage de decode vers pcplus1), afin de commander la mémorisation des indicateurs N, Z, V, et C (faite l'extérieur).

Les équations de areg, breg, dreg, ualCmd et dbusIn seront complétées progressivement lors des prochaines étapes. Pour en assurer une bonne lisibilité et en faciliter la vérification, il est fortement conseillé d'écrire un min-terme par ligne ; chaque min-terme correspondant à une transition.

Valider cette première étape en utilisant le fichier de test (sequenceur_test) fourni (et qu'il faut lire et comprendre pour pouvoir le compléter lors des étapes suivantes).

Temps moyen = 15 mn

B- Prendre une copie du module minicraps.sdhl sur moodle, le lire, le comprendre, et le compléter. C'est une traduction fidèle du schéma général de minicraps, avec deux petits ajouts qui permettent de lire le contenu d'un registre et le contenu de la ram (voir commentaires dans le module).

Valider son fonctionnement en utilisant les fichiers add_sub_ram et minicraps_add_sub_test fournis. Le chargement dans le simulateur doit se faire dans l'ordre : fichier ram, puis fichier de test. *Temps moyen = 20 mn*

Bien lire et comprendre ces deux fichiers pour pouvoir en générer lors des étapes suivantes. Le programme testé se trouve, en version binaire dans le fichier ram, et en commentaire, en tête du fichier de test.

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

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

A- Compléler le module sequenceur.shdl pour y intégrer le traitement de cette instruction. Compléter le fichier sequenceur_test et valider le module sequenceur. *Temps moyen = 15 mn*

B- Valider le fonctionnement de minicraps en utilisant les fichiers set_ram et minicraps_set_test fournis. *Temps moyen = 5 mn*

---------- Objectif qui doit être atteint à la fin de la première séance ----------

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.

A- Compléler le module sequenceur.shdl pour y intégrer le traitement de ces instructions. Compléter le fichier sequenceur_test et valider le module sequenceur. *Temps moyen = 25 mn*

B- Valider le fonctionnement de minicraps en utilisant les fichiers load_store_ram et minicraps_load_store_test fournis. *Temps moyen = 5 mn*

Bien lire et comprendre ces deux fichiers pour pouvoir en générer lors des étapes suivantes.

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

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

Les instructions de branchement permettent de se déplacer dans le code (passer d'une instruction vers une autre située en amont ou en aval, et référencée par son adresse, et fonctionnent selon l'algorithme suivant :

Si cond est vrai alors PC <- PC + déplacement24

Sinon PC <- PC + 1

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

Prendre une copie du module branch qui évalue si la condition, indiquée en entrée, est vraie en fonction des indicateurs N, Z, V et C.

La table suivante fournit la liste des instructions de branchement évaluées dans branch.shdl.

Instruction	cond (code)	Opération : branch …	Flags vérifiés
ba	1000 (8)	always	1
beq (be, bz)	0001 (1)	on equal	Z
Bne (bnz)	1001 (9)	on not equal	Not Z
ble	0010 (2)	on less or equal	(N xor V) or Z
bg (bgt)	1010 (10 / A)	on greater	Not ((N xor V) or Z)
bl	0011 (3)	on less	N xor V
bge	1011 (11 / B)	on greater or equal	Not (N xor V)
bleu	0100 (4)	on less or equal unsigned	Z or C
bgu	1100 (12 / C)	on greater unsigned	Not (Z or C)
blu (bcs)	0101 (5)	less unsigned (carry set)	C
bgeu (bcc)	1101 (13 / D)	greater or equal unsigned	Not C
bneg (bn)	0110 (6)	on negative	N
bpos (bnn)	1110 (14 / E)	on positive	Not N
bvs	0111 (7)	on oVerflow set	V
bvc	1111 (15 / F)	on oVerflow clear	Not V

Les instructions de branchement permettent de mettre en place les structures de contrôle : Si Sinon, Répéter, TantQue, etc. Par exemple, soit la boucle suivante en langage algorithmique, et une implantation possible en assembleur :

Index ← 8		set 8, %r2	// r1 = index
Répéter	Boucle :	…	// Actions
Actions		sub %r2, 1, %r2	// positionne N, Z, V, et C
Index ← Index - 1		bne Boucle	// branchement à boucle si
Jusqu'à Index=0			résultat du sub /= 0 (i.e. Z=0)

Déplacement24 = adresse de bne – adresse Boucle. Donc, si Actions consomment 3 instructions, déplacement24 sera égal à -4 = 0xfffffc, et le code de l'instruction bne sera 0xe9fffffc

A- Compléler le module sequenceur.shdl pour y intégrer le traitement de cette instruction. Compléter le fichier sequenceur_tst et valider le module sequenceur. *Temps moyen = 20 mn*

B- Enregistrer le code binaire du programme suivant dans un fichier ram (somme_tab_ram) :

```
00000000 :  set     10, %r2          // adresse du tableau
            set     0, %r4           // somme <- 0
            set     5, %r3           // index <- N-1 (N = nombre d'éléments du tableau)
boucle:     load    [%r2+%r3], %r5 // @tab[index] = r2+r3 = base + index
            add     %r5, %r4, %r4
            sub     %r3, %r1, %r3  // Index <- Inex - 1
            bgeu    boucle           // branchement à boucle si r3 >= 0 (non signé)
stop :      ba      stop             // fin du programme
00001000 :  valeur du 1er élément du tableau (tab[0])
00001001 :  valeur du 2ème  élément du tableau (tab[1])
            …
```

Tester manuellement jusqu'au 1er retour sur boucle. Quel est le nombre de cycles que consomme ce premier passage dans la boucle ? *Temps moyen = 20 mn*

Ecrire un fichier somme_tab_test pour valider l'exécution complète de ce programme.
On se limitera à tester l'état du programme à chaque passage sur la 1ère instruction de la boucle (avant exécution), et on vérifiera le résultat final. *Temps moyen = 15 mn*