GRENOBLE INP UGA Esisar VALENCE

Architecture (SN330-SN331)

Cahier de TP, 2024-25

Contents

Lab 1
Programming in RISCV assembly 1/2

Objective

Credits: startup lab for compilation courses in Lyon and Valence, in collaboration with Matthieu Moy.

- Be familiar with the RISCV instruction set.
- Understand how it executes on the RISCV processor with the help of a simulator.
- Write simple programs, assemble, execute.
- Pair working only. (not more) **Chamilo deposit exercise number XXX : see Chamilo for instructions.**

1.1 Startup: assembling, desassembling, simulating

EXERCISE #1 ▶ **Lab preparation**
Boot on Linux !. Get the archive on chamilo, then extract:

```
tar xvzf archivename.tgz
```

On the Esisar (linux) machines, everything is already installed. If you want to install on your machines, follow the instructions on this webpage (but not during a lab session!)

`https://forge.univ-lyon1.fr/matthieu.moy/mif08-2021/-/blob/main/INSTALL.md`

EXERCISE #2 ▶ RISCV **C-compiler and simulator, first test**
In the directory `TP01/startup/`:

- Compile the provided file `ex1.c` with:
  `riscv64-unknown-elf-gcc ex1.c -o ex1.riscv`
  It produces a RISCV binary named `ex1.riscv`.
- Execute the binary with the RISCV simulator:
  `spike pk ex1.riscv`
  This should print:
  ```
  bbl loader
  42
  ```
  If you get a runtime exception, try running `spike -m100 pk ex1.riscv` instead: this limits the RAM usage of spike to 100 MB (the default is 2 GB).
- The corresponding RISCV code can be obtained in a more readable format by:
  `riscv64-unknown-elf-gcc ex1.c -S -o ex1.s -fverbose-asm`
  (have a look at the generated .s file!)

EXERCISE #3 ▶ **Documents**
Some documentation can be found in the RISCV ISA on Chamilo.

1.1.1 Assembling, disassembling

EXERCISE #4 ▶ **Hand assembling, simulation of the hex code**
Assemble by hand (on paper) the instructions:

```
1        .globl main
2 main:
3         addi a0, a0, 1
4         bne a0, a0, main
5 end:
6         ret
```

You will need the set of instructions of the RISCV machine and their associated opcode. All the info is in the (mini) ISA documentation.

To check your solution (**after** you did the job manually), you can redo the assembly using the toolchain:

```
riscv64-unknown-elf-as -march=rv64g asshand.s -o asshand.o
```

`asshand.o` is an ELF file which contains both the compiled code and some metadata (you can try `hexdump asshand.o` to view its content, but it's rather large and unreadable). The tool `objdump` allows extracting the code section from the executable, and show the binary code next to its disassembled version:

```
riscv64-unknown-elf-objdump -d asshand.o
```

Check that the output is consistent with what you found manually.

From now on, we are going to write programs using an easier approach. We are going to write instructions using the RISCV assembly.

1.1.2 RISCV **Simulator**

Code source is again in directory `TP01/startup`

EXERCISE #5 ► **Execution and debugging**

See `https://www.lowrisc.org/docs/tagged-memory-v0.1/spike/` for details on the Spike simulator.

`test_print.s` is a small but complete example using Risc-V assembly. It uses the `println_string`, `print_int`, `print_char` and `newline` functions provided to you in `libprint.s`. Each function can be called with `call print_...` and prints the content of register `a0` (`call newline` takes no input and prints a newline character).

1. First test assembling and simulation on the file `test_print.s`:
   `riscv64-unknown-elf-as -march=rv64g test_print.s -o test_print.o`
2. The `libprint.s` library must be assembled too:
   `riscv64-unknown-elf-as -march=rv64g libprint.s -o libprint.o`
3. We now link these files together to get an executable [1]:
   `riscv64-unknown-elf-gcc test_print.o libprint.o -o test_print`
   The generated `test_print` file should be executable, but since it uses the Risc-V ISA, we can't execute it natively (try `./test_print`, you'll get an error like `Exec format error`).
4. Run the simulator:
   `spike pk ./test_print`
   The output should look like:
   ```
   bbl loader
   HI CE313
   42
   a
   ```
   The first line comes from the simulator itself, the next two come from the `println_string`, `print_int` and `print_char` calls in the assembly code.
5. We can also view the instructions while they are executed:
   `spike -l pk ./test_print`
   Unfortunately, this shows all the instructions in `pk` (Proxy Kernel, a kind of mini operating system), and is mostly unusable. Alternatively, we can run a step-by-step simulation starting from a given symbol. To run the instructions in `main`, we first get the address of `main` in the executable:
   ```
   $ riscv64-unknown-elf-nm test_print | grep main
   000000000001014c T main
   ```
   This means: `main` is a symbol defined in the `.text` section (T in the middle column), it is global (capital T), and its address is 1014c (you may not have the same address, so **write somewhere yours**). Now, run spike in debug mode (`-d`)and execute code up to this address (`until pc 0 1014c`, i.e. "Until the program counter of core 0 reaches 1014c"). Press **Return** to move to the next instruction and `q` to quit:

[1] you can use any name, and/or add an extension such that .exe or .riscv for your binaries, we do not mind

```
$ spike -d pk ./test_print
: until pc 0 1014c
bbl loader
:
core   0: 0x000000000001014c (0xff010113) addi    sp, sp, -16
:
core   0: 0x0000000000010150 (0x00113423) sd      ra, 8(sp)
:
core   0: 0x0000000000010154 (0x0000e517) auipc   a0, 0xe
:
core   0: 0x0000000000010158 (0x41450513) addi    a0, a0, 1044
: q
$
```

Remark: You may want to assemble and link with a single command (which can also do the compilation if you provide `.c` files on the command-line):

```
riscv64-unknown-elf-gcc -march=rv64g libprint.s test_print.s -o main
```

In real-life, people run compilation+assembly and link as two different commands, but use a build system like a `Makefile` to re-run only the right commands.

1.2 Let us program!

Source code is now in `TP01/riscv`.

1.2.1 Simple programs to complete

EXERCISE #6 ▶ **MinMax**
During the course, we wrote a program `minmax` that computes the maximum of two 64 bits integers in memory.

- Verify that it actually works (perhaps add some `print_int` calls).
- Extend this program so that it also computes the minimum. Test.

EXERCISE #7 ▶ **Array sum**
In the exercice session we wrote a routine to increment each element of an array by a constant 1. A solution is given in `arrayinc.s`.

- Read, understand, test this program.
- Copy the source into a new file:
  `cp arrayinc.s arraysum.s`
- Edit the new file and modify so that it computes the sum of all elements of the array.

EXERCISE #8 ▶ **Palindromes**
Inspired by `http://home.wlu.edu/~lambertk/classes/210/exercises/hw7.htm`.

A given string is a palindrome if it is equal to its mirror string. For instance *kayak* and *noon* are palindroms.
In pseudo-C, the following pseudo-code actually computes whether a given word is palindromic:

```
bool isPalindrome(char *string){
    int left,right ;
    left = 0 ;
    right = len(string) - 1 ;
    while (left < right){
        if (string[left] != string[right])
            return(false);
        left += 1;
        right -= 1;
    }
    return(true)
}
```

The objective is to write the equivalent in RISCV assembly. We give you a squeleton of code in `TP01/riscv/ispal.s`. A length routine is given.

- Test the `length` routine (call from the main).
- Complete the `ispal` routine, and test.
- Explain why storing `ra` on the stack is mandatory, for which routine ?

1.2.2 Programs "from scratch"

EXERCISE #9 ▶ **Integer sum**
n being stored in memory, write a program that computes the sum $1+2+\ldots+n$, and prints the result.

EXERCISE #10 ▶ **Caesar code**

Create a file named `codecesar.s`, starting with:

```
1  # Code de César en RISCV
2  # CE313, binôme : NOM1, NOM2
3 .section .text
4 .globl main
5 main:
6         addi    sp,sp,-16
7         sd      ra,8(sp)
8
9         ...
```

A chain s being stored in memory, as well as a dec number, compute the Caesar code of the chain: shift every letter value by dec.

Your code should print the input string and the encoded one (thanks to a call to `print_string`. For instance, with the Hello World chain and a dec equal to 4:

```
Hello world!
Lipps${svph%
```

Please use a routine as shown in the palindrome exercise.

Lab 2
Programmation assembleur 2/2

Objectifs
- Écrire des programmes simples, assembler, debugger
- Écrire et débugger des programmes avec fonctions.

Préambule Vous trouverez les sources nécessaires sur Chamilo. On utilisera les moyens de compilation et d'exécution vus au premier TP ("Lab1"). Pour la section 3 on utilisera le simulateur venus: `cfhttps://venus.cs61c.org/` ce simulateur permet des appels systèmes selon la documentation qui se trouve ici: `https://github.com/kvakil/venus/wiki/Environmental-Calls` Il est également possible d'utiliser Venus pour les sections 1 et 2.

2.1 Un exercice d'algorithmique classique : méthode de Horner

D'après un examen de CE313.

On désire dans cet exercice résoudre le problème d'entrée d'un entier au clavier, sachant que l'utilisateur·rice n'entre qu'une série de caractères au clavier. Ainsi, la suite de caractères '1', '0', '2, '5' doit être interprétée en base 10 par le nombre 1025.

Pour simplifier, on suppose qu'une procédure externe a déjà stocké la chaîne de caractère en mémoire, à une adresse fixée. Il ne reste plus qu'à calculer le nombre associé, en lisant la chaîne "de gauche à droite":

- On transforme chaque caractère chiffre en le chiffre correspondant.
- On applique la méthode de Horner pour calculer le nombre désiré.

Voici le code principal de notre programme (sur 2 colonnes).

```
1  section .text
2  .globl main
3  main:
4          addi    sp,sp,-16
5          sd      ra,8(sp)
6          la a0, .LC1
7          call    println_string
8          # appel à la routine convert
9          la a0 , .LC1
10         call convert
11         # affichage du résultat
12         la a0, .res
13         ld a0, 0(a0)
14         call println_int
15         ld      ra,8(sp)
16         addi    sp,sp,16
17         jr      ra
18         ret
19 ## Partie à compléter
20 # Section données.
21         .section        .data
22         .align  3
23 .LC1:
24         .string "1025"
25
26 .res :
27         .dword 0
28 ### fin du listing
```

La méthode de Horner consiste à utiliser la décomposition en base 10:

$$1025 = ((1 \times 10 + 0) \times 10 + 2) \times 10 + 5$$

La première étape consiste donc à implémenter une routine `mul10`. On procède sans écrire de boucle, en remarquant que $10n = 2 \times n + 2 \times 2 \times 2 \times n$.

EXERCISE #1 ▶ **Multiplication sans multiplication**
Écrire la routine `mul10`, qui réalise l'opération $a_0 \leftarrow 10a_0$, en n'utilisant qu'un registre temporaire t_1, et la multiplication par 2 (à l'aide du shift: instruction `slli` "shift left logical immédiate"). Tester en réalisant 1 ou 2 appels à partir du *main*.

EXERCISE #2 ▶ **Implémentation de convert**

On fournit le pseudo-code de la routine `convert`.

```
routine convert(paramètre entree: a0){
  s1 = a0;  // copie de l'adresse de la chaîne
  t2 = 0;   // t2 calcule l'entier résultat
  s2 = nouveaucaractère() // à la bonne adresse
  tant que (s2 != '\0'){
       s2 = s2 - '0'; // <-- * à expliquer
       t2 = t2*10;
       t2 = t2+s2;
       s2 = nouveaucaractère()
  }
  memory[res] = t2 ; // stockage du résultat
}
```

1. Dérouler (sur papier) les étapes de cette routine sur la chaîne "1025". Expliquer en particulier à quoi sert la ligne étoilée.
2. *En suivant les conventions du pseudo code pour le nom des registres,* écrivez le code RISCV de la routine `convert`. Attention à l'appel à `mul10`. (On rappelle que le passage des paramètres et du résultat s'effectue via le registre `a0` en RISCV.) Tester.

2.2 Mini-exos

EXERCISE #3 ▶ **Multiplication par 6 modulo 16**

L'objet de cet exercice est de multiplier l'ensemble des données d'un tableau d'entiers 64 bits par 6 modulo 16. On pourra considérer des entiers stockés entre deux adresses *debut* et *fin*.

1. Comment réaliser la multiplication par 6 sans utiliser mul ?
2. Comment réaliser l'opération "modulo 16" à l'aide d'un calcul booléen ?

Implémenter une routine, et tester.

2.3 Appel de fonction

Dans cet exercice on souhaite réécrire ce programe en assembleur RISC-V.

```
int source[] = {3, 1, 4, 1, 5, 9, 0};
int dest[10];

int fun(int x) {
    return -x * (x + 1);
}

int main() {
    int k;
    int sum = 0;
    for (k = 0; source[k] != 0; k++) {
        dest[k] = fun(source[k]);
        sum += dest[k];
```

```
    }
    printf("sum: %d\n", sum);
}
```

On fournit le code d'initialisation des tableaux sources et destination :

```
.data
source:
    .word   3
    .word   1
    .word   4
    .word   1
    .word   5
    .word   9
    .word   0
dest:
    .word   0
    .word   0
    .word   0
    .word   0
    .word   0
    .word   0
    .word   0
    .word   0
    .word   0
    .word   0

```

EXERCISE #4 ▶ **Écriture de la fonction fun**

Réaliser la routine implémentant la fonction :

```
int fun(int x) {
   return -x * (x + 1);
}
```

EXERCISE #5 ▶ **Écriture du main**

Écrire la routine principale: tout d'abord sans considérer les conventions d'appel; les ajouter par la suite.

EXERCISE #6 ▶ **Vérification de la convention d'appel**

En utilisant le simulateur Venus (cf. `https://venus.cs61c.org/`, et la doc `https://inst.eecs.berkeley.edu/~cs61c/sp21/resources/venus-reference#tools`, vérifiez que votre code respecte la convention d'appel RISC-V en utilisant l'option "calling convention checker" du simulateur activable dans l'onglet "setting". Attention: les appels de fonction sur Venus doivent se faire avec l'instruction `jal` pour pouvoir utiliser la fonction convention checker.

TP 3
Mémoires Caches

Credits : Berkeley University

Objectifs

Les objectifs de ce TP sont :
- de comprendre le fonctionnement des mémoires caches.
- d'analyser comment l'ordonnancement des accès mémoires détermine le taux de succès en cache.
- de déterminer le meilleur ordonnancement des accès mémoire pour optimiser le taux de succès en cache.

Afin de visualiser le fonctionnement des mémoires caches, nous allons utiliser l'outil de visualisation de cache fourni par le simulateur Venus (`https://venus.cs61c.org/`, utilisé lors du TP précédent.

Pour illustrer le comportement de la mémoire cache, nous utilisons le programme du fichier `cache.s` Ce fichier est disponible sur Chamilo, et reproduit à la fin de l'énoncé.

Etude préliminaire Analysez le programme décrit dans le fichier `cache.s` pour vous faire une idée approximative de ce que fait le programme. Assurez-vous de bien comprendre ce que fait le pseudocode et ce que contiennent les registres d'arguments avant de procéder à l'analyse des différentes configurations de cache données dans la suite de ce TP.

EXERCICE #1 ▶ **Analyse du programme**

- Que fait ce programme ? Donner un pseudo-code.
- Quels sont le rôle des registres a1, a2 et a3 ?

Utilisation de Vénus Pour chacune des configurations de cache données ci-après, vous devrez répéter ces étapes :
- Charger `cache.s` dans Venus.
- Dans le code de `cache.s`, définir les paramètres de programme appropriés comme indiqué au début de chaque configuration
- Lorsque l'on exécute du code dans Venus, tout accès à la mémoire de donnée (chargement ou stockage) s'affiche (les extractions d'instructions ne sont pas affichées car les instructions sont chargées dans un cache d'instructions séparé qui n'est pas affiché dans Venus).
- Le simulateur de cache indique l'état du cache de données. Réinitialiser le code signifie donc également réinitialiser le cache (et donc remet le calcul des taux de réussite et d'échec à 0).

IMPORTANT : Il faudra placer des points d'arrêt judicieusement dans le code pour pouvoir faire des exécutions partielles, afin de comprendre l'évolution des succès et erreurs de cache.

Méthodologie Pour chaque scénario des exercices suivants, il est demandé d'observer les taux de réussite des accès au cache. Vous essayerez de **calculer le taux de réussite AVANT d'exécuter le code. Vous justifierez (par l'étude théorique) de la structure du cache les résultats obtenus par simulation.**

Pour guider votre démarche, vous pourrez vous poser (entre autres) les questions suivantes :
- Quelle est la taille de votre bloc de cache ?
- Combien d'accès consécutifs (en tenant compte de la taille du pas) peuvent être effectués dans un seul bloc ?
- Quelle quantité de données peut être stockée dans l'ensemble de la mémoire cache ?
- Quelle est la distance en mémoire entre les blocs qui correspondent au même ensemble (et qui pourraient créer des conflits) ?
- Quelle est l'associativité de votre cache ?

- À quel endroit de la mémoire cache un bloc particulier correspond-il?

EXERCICE #2 ► **Etude de la configuration 1**

Pour la première étude, on considère les configurations de la mémoire cache et du programme des figures 3.1 et 3.2. *Dans Venus, la case "ENABLE" du cache doit être activée (verte).*

Niveau de cache	1
Taille des blocs	8 octets
Nombre de blocs	4
Politique de placement	correspondance direct (direct mapped)
Degré d'associativité	1 (**pourquoi?**)
Politique de remplacement	LRU (**est-ce utile ici?**)

TABLE 3.1 : Cache L1 pour la configuration 1

Array Size (a0)	128 octets
Step Size (a1)	8
Rep Count (a2)	4
Option (a3)	0

TABLE 3.2 : Paramètres du programme pour la configuration 1 - à mettre dans le code.

Pour analyser vos résultats, il est demandé de décomposer l'adresse pour identifier les 3 champs (tag, index, offset) et ainsi comprendre comment les accès mémoires sont traités par le contrôleur de cache.

- Quelle combinaison de paramètres produit le taux de réussite que vous observez? (Indice : votre réponse devrait être de la forme suivante : "Le [paramètre A] en octets est exactement égal à [paramètre B] en octets, donc ...").
- Quel est le taux de réussite si nous augmentons arbitrairement le paramètre *repcount* d'essais (boucle externe)? Pourquoi?
- Comment pourrions-nous modifier un paramètre du programme pour augmenter notre taux de réussite?

EXERCICE #3 ► **Etude de la configuration 2**
On utilise maintenant la configuration des figures 3.3 et 3.4.

Niveau de cache	1
Taille des blocs	16 octets
Nombre de blocs	16
Politique de placement	(associatif par ensemble de N) N-Way Set Associative
Degré d'associativité	4
Politique de remplacement	LRU

TABLE 3.3 : Cache L1 pour la configuration 2

Array Size (a0)	256 octets
Step Size (a1)	2
Rep Count (a2)	1
Option (a3)	1

TABLE 3.4 : Paramètres du programme pour la configuration 2

- Combien d'accès à la mémoire y a-t-il par itération de la boucle interne? (pas celle qui implique le repcount).
- Quel est le schéma répétitif "hit/miss"? Pourquoi?

- Expliquez le taux de réussite en termes de modèle de réussite/échec.
- En gardant tout le reste inchangé, comment évolue le taux de réussite lorsque RepCount augmente ? Pourquoi ?

Supposons que nous ayons un programme qui itère à travers un très grand tableau (bien plus grand que la taille du cache) plusieurs fois. Pendant chaque Rep, nous effectuons un traitement différent aux éléments du tableau (par exemple, si Rep Count = 1024, nous effectuons 1024 traitement différents, un par Rep).

Compte tenu du programme, comment pouvons-nous restructurer ses accès aux tableaux afin d'obtenir un taux de réussite tel que celui obtenu dans ce scénario ? Nous supposons que chaque élément du tableau est modifié indépendamment des autres.

EXERCICE #4 ▶ Etude de la configuration 3

Dans cette dernière configuration, nous allons utiliser deux niveaux de cache, il faut donc spécifier "2" pour le paramètre niveaux de cache, et les paramètres des caches et du programme sont décrits aux figures 3.5, 3.6 et 3.7.

Taille des blocs	8 octets
Nombre de blocs	8
Politique de placement	correspondance directe
Degré d'associativité	1
Politique de remplacement	LRU

TABLE 3.5 : Cache L1 pour la configuration 3

Taille des blocs	8 octets
Nombre de blocs	16
Politique de placement	correspondance directe
Degré d'associativité	1
Politique de remplacement	LRU

TABLE 3.6 : Cache L2

Array Size (a0)	128 octets
Step Size (a1)	1
Rep Count (a2)	1
Option (a3)	0

TABLE 3.7 : Paramètres du programme pour la configuration 3

Grâce aux simulations vous expliquerez comment fonctionnent ces deux niveaux de cache.

- Quel est le taux de succès du cache L1 ? Celui du cache L2 ?
- Combien d'accès avons-nous dans le cache L1 ? Combien sont des échecs ?
- Combien d'accès avons-nous dans le cache L2 ? Combien sont des échecs ? Quel est le lien avec le cache L1 ?
- Quel paramètre du programme permettrait d'augmenter le taux de succès du L2 en conservant celui du L1 ?
- Quel est l'effet sur les taux de succès en L1 et L2 si on augmente le nombre de blocs en L1 ? et si l'on augmente la taille des blocs en L1 ?

```
1
2  .data
3  array:  .word   2048          # max array size specified in BYTES (DO NOT CHANGE)
4
5  .text
6  ############################################################
7  # You MAY change the code below this section
8  main:  li      a0, 256       # array size in BYTES (power of 2 < array size)
9      li      a1, 2         # step size  (power of 2 > 0)
10     li      a2, 1         # rep count  (int > 0)
11     li      a3, 1         # 0 - option 0, 1 - option 1
12 # You MAY change the code above this section
13 ############################################################
14
15     jal     accessWords    # lw/sw
16     #jal    accessBytes    # lb/sb
17
18     li      a0,10         # exit
19     ecall
20
21 # SUMMARY OF REGISTER USE:
22 #  a0 = array size in bytes
23 #  a1 = step size
24 #  a2 = number of times to repeat
25 #  a3 = 0 (W) / 1 (RW)
26 #  s0 = moving array ptr
27 #  s1 = array limit (ptr)
28
29 accessWords:
30     la      s0, array           # ptr to array
31     add     s1, s0, a0          # hardcode array limit (ptr)
32     slli    t1, a1, 2           # multiply stepsize by 4 because WORDS
33 wordLoop:
34     beq     a3, zero,  wordZero
35
36     lw      t0, 0(s0)           # array[index/4]++
37     addi    t0, t0, 1
38     sw      t0, 0(s0)
39     j       wordCheck
40
41 wordZero:
42     sw      zero,  0(s0)        # array[index/4] = 0
43
44 wordCheck:
45     add     s0, s0, t1          # increment ptr
46     blt     s0, s1, wordLoop    # inner loop done?
47
48     addi    a2, a2, -1
49     bgtz    a2, accessWords # is outer loop done?
50     jr      ra
51
52
53 accessBytes:
54     la      s0, array           # ptr to array
55     add     s1, s0, a0          # hardcode array limit (ptr)
56 byteLoop:
57     beq     a3, zero,  byteZero
58
59     lbu     t0, 0(s0)           # array[index]++
60     addi    t0, t0, 1
```

```
61      sb     t0, 0(s0)
62      j      byteCheck
63
64  byteZero:
65      sb     zero,  0(s0)         # array[index] = 0
66
67  byteCheck:
68      add    s0, s0, a1           # increment ptr
69      blt    s0, s1, byteLoop     # inner loop done?
70
71      addi   a2, a2, -1
72      bgtz   a2, accessBytes # outer loop done?
73      jr     ra
```