

ARC TD6 et TD7

QTRVSIM : un simulateur de l'architecture RISC-V
(2x2h séance sur machine), 12 juin 2024

Objectifs pédagogiques de ce TD/TP :

1. comprendre la pile, les registre, l'appel de fonction, l'assembleur en vrai quoi

Attention : version corrigée

1 Mise en place

Nous allons utiliser le simulateur QtRVSim (<https://github.com/cvut/qtrvsim>) développé par l'Université de Prague à des fins pédagogiques pour comprendre le principe de fonctionnement des machines RISC ainsi que le pipeline des instructions dans un processeur RISC.

QtRVSim est un simulateur du jeu d'instruction du RISC-V (*instruction set simulator* : ISS). Nous nous contenterons de la version la plus simple du jeu d'instruction : RV32I (instructions riscv 32 bit en entier, c'est à dire sans multiplieur cablé). La description du jeu d'instruction du RISC-V est disponible en fin de ce sujet de TD ou, de manière complète, sur la page de l'organisation en charge du RISC-V (<https://riscv.org/>).

QtRVSim est installé sur les machines du département (commande `qtrvsim_gui`), vous pouvez l'installer sur votre machine (instructions sur le README du github : <https://github.com/cvut/qtrvsim>), ou vous pouvez simplement l'utiliser dans un navigateur avec la version compilée pour WebAssembly :

<https://comparch.edu.cvut.cz/qtrvsim/app/>.

Une publication décrivant brièvement le simulateur est disponible ici : <https://comparch.edu.cvut.cz/publications/ewC2022-Dupak-Pisa-Stepanovsky-QtRvSim.pdf>

2 Lancement de QtRVSim

Télécharger les exemples assembleur disponibles dans l'archive `qtrvsim-files.tar` sur Moodle. Nous allons commencer avec l'exemple intégré à QtRVSim qui modélise l'écriture sur le port série de la chaîne "Hello world".

1. Lancer QtRVSim, soit en utilisant la commande `qtrvsim_gui` dans un shell, soit dans un navigateur web (<https://comparch.edu.cvut.cz/qtrvsim/app/>), laissez cochée la case "No pipeline no cache" et cliquer sur Example.

vous devez voir quelque chose qui ressemble à ce qui montré en Figure 1, nous simulerons un RISC-V sans pipeline des instructions (i.e. 1 cycle par instruction) sur le programme donné dans l'onglet `template.S`. Si vous ne voyez pas l'architecture du processeur comme sur la figure 1, il est dans l'onglet "Core".

En haut de l'interface vous voyez l'état des 32 registres avec leurs deux noms (les 32 registres du RISC-V sont rappelés en fin de sujet de TP, page 12, notez qu'en activant l'options `Machine` → `Mnemonics registers`, vous pouvez voir les noms de registre de l'ABI et plutôt que les noms `x1`, `x2`, ...).

Sur la gauche vous voyez les instructions assembleur avec leur adresse en mémoire . Sur la droite en bas vous voyez l'état de la mémoire que vous pouvez explorer en entrant une adresse mémoire dans la case tout en bas à droite.

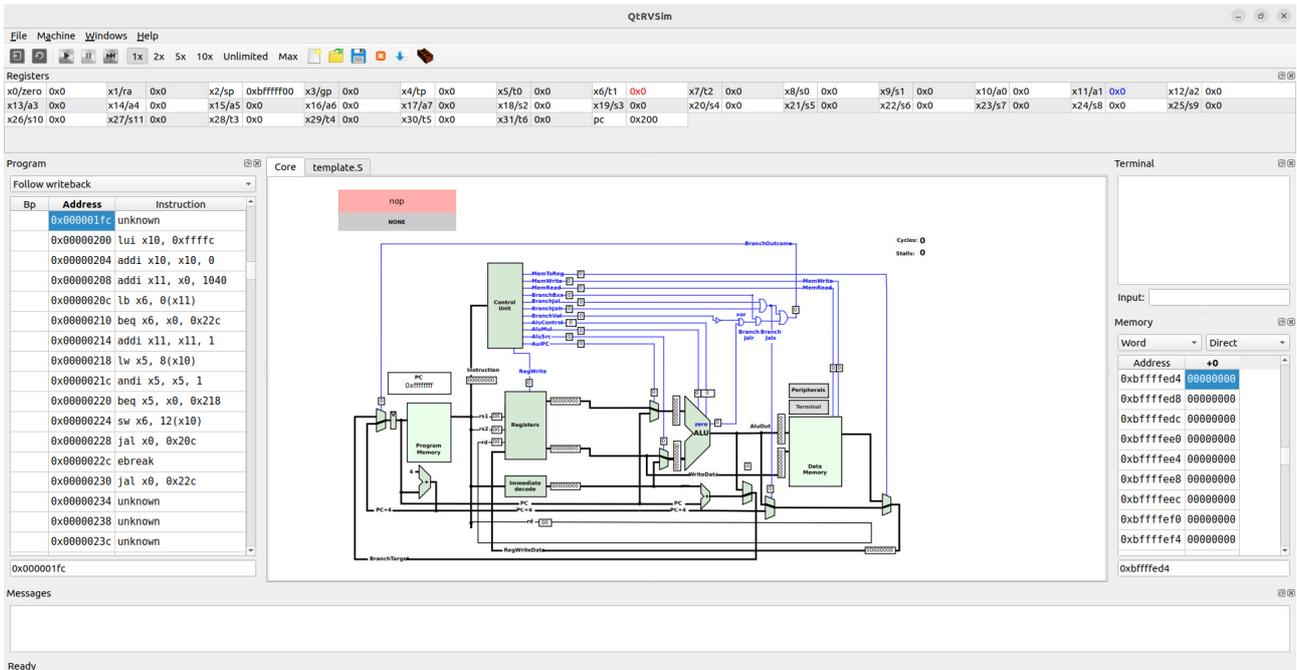


FIGURE 1 – Capture d'écran de l'interface de QtRVSim au démarrage

Bien vérifier qu'il on bien cliqué sur Exemple pour que tout le monde ai la même interface.

2. La première instruction à exécuter est : `lui x10, 0xffffc`, que fait-elle ? quelle est l'instruction correspondante dans le programme `template.S` de la Fig. 2 (l'instruction `lui` est expliquée en annexe page 13 du sujet de TP).

la valeur de `x10` est `0xffffc000` l'instruction `lui Imm (load upper immediate)` charge la valeur immédiate décalée de 12 bit : $rd = Imm \ll 12$. Dans notre cas, `0xffffc` décalé de 12 bit donne bien le résultat : `0xffffc000`, ce qui est bien la valeur de la macro `SERIAL_PORT_BASE`. C'est donc bien la traduction de l'instruction `li a0, SERIAL_PORT_BASE` du programme `template.S`

3. Cliquez une fois sur le bouton "step" : L'instruction `lui x10 0xffffc` est maintenant grisée sur la gauche, c'est qu'elle vient d'être exécutée. Repérez sur la schématique du processeur (i.e. onglet "core"), la valeur du *program counter* PC. Repérez, toujours sur la schématique, la valeur en hexadécimal du registre d'instruction. Repérez la nouvelle valeur du registre `x10`. Quelle est la valeur du stack pointer SP?

- la valeur de PC est `0x200` (on le voit au dessus de "program memory" sur la schématique). Cette valeur a été mise dans l'assembleur par la ligne `.org 0x00000200 (Start address after reset)`.
- la valeur du registre d'instruction est `0xffffc537`, on le voit en petit sous le commentaire "instruction", c'est le codage de l'instruction `lui x10 0xffffc`.
- la valeur de `x10` est `0xffffc000`, l'instruction `lui Imm (load upper immediate)` charge la valeur immédiate décalée de 12 bit : $rd = Imm \ll 12$. Dans notre cas, `0xffffc` décalé de 12 bit donne bien le résultat : `0xffffc000`
- la valeur du stack pointer est `0xbffff0`, il n'y a rien dans la pile.

4. Cliquez répétitivement sur le "step" : Vous voyez les instructions successives s'exécuter dans l'architecture RISC-V et vous voyez

dans la fenêtre "Terminal" en haut à droite, s'afficher petit à petit "Hello world". Nous allons comprendre ce programme dans la question suivante.

3 Compréhension du programme `template.S`

1. Cliquer sur `File` → `reload simulation` et cliquer sur l'onglet `template.S` dans la fenêtre centrale. Le programme `template.S` commenté est montré en Fig. 2. Ce programme émule l'écriture d'une chaîne de caractères, caractère par caractère, sur un port série (qui est modélisé par le terminal en haut à droite).

Il y a d'abord la définition d'un certain nombre de macros. par exemple celle-ci :

```
.equ SERIAL_PORT_BASE, 0xffffc000
```

qui définit l'adresse du port série comme étant `0xffffc000`. Cela signifie que si l'on écrit un caractère à l'adresse `0xffffc000`, ce caractère sera envoyé dans le terminal (il ne s'agit pas d'une caractéristique du processeur RISC-V mais plutôt de la carte mère sur laquelle va être soudée la puce du processeur RISC-V).

2. Repérez, sur le programme `template.S` dans `QtRVSim`, les sections de code (`.text`) et de données (`.data`), les directives `.org` indiquent à quelles adresses vont être chargées ces sections en mémoire. À quelle adresse commence le code? À quelle adresse sont rangées les données? Notamment la chaîne de caractère "Hello world". Allez explorer la mémoire pour retrouver ces données.

on trouve `.org 0x0000200` qui indique que le code est à l'adresse `0x200` et effectivement la première instruction est à l'adresse `0x200`. et ensuite `.org 0x400` qui indique que les données sont à l'adresse `0x400`.

Effectivement si on tape `0x400` en bas à droite, on voit dans la mémoire les quatre entier 1, 2, 3, 4 puis la chaîne de caractère en ASCII à l'adresse `0x410` (qui est 1040 en décimal)

3. Sachant que le code ASCII de 'l' est `0x6C`, celui de 'e' est `0x65` et celui de 'H' est `0x48`, comprenez-vous le contenu de la case `0x410`?

Le contenu de la case `0x410` est `0x6C6C6548`, il s'agit bien du début de la chaîne de caractère : `Hello` mais arrangées à l'envers `lleH` car le `riscV` est *little endian* : en *little endian*, les octets de poids faible sont mis à gauche du mot, contrairement à la manière dont on écrit un entier.

4. Lisez le programme de `template.S`, comparez avec les instructions dans la mémoire pour voir la correspondance et comprenez les étapes :
 - (a) chargement de l'adresse du port série dans `a0` (on a vu que cette instruction est transformée en `lui`).
 - (b) chargement de l'adresse de la chaîne de caractère "Hello world" dans `a1` (sachant que `1040 = 0x410`).
 - (c) chargement du caractère pointé par `a1` dans `t1` (rappelez vous que `lb` ne charge qu'un octet).
 - (d) Si le caractère chargé dans `t1` est 0 on sort du programme par le label `end_char`.
 - (e) Sinon on incrémente `a1` de 1 (`a1` pointe alors sur le prochain caractère).
 - (f) les trois instructions suivant le label `tx_busy` sont là pour vérifier que le port série est disponible en réception, vous n'êtes pas censé les comprendre.
 - (g) on écrit le caractère sur le port série (instruction `sw t1, SERP_TX_DATA_REG_o(a0)`)
 - (h) on reboucle en branchant à l'étape (c) ci-dessus.
5. Suivez l'exécution pas à pas du programme à nouveau, et vérifiez l'affichage du caractère à chaque étape (g) ci-dessus.

```

_start:

loop:
    li    a0, SERIAL_PORT_BASE           // load base address of serial port
    addi  a1, zero, text_1               // load address of text

next_char:
    lb    t1, 0(a1)                      // load one byte after another
    beq   t1, zero, end_char             // is this the terminal zero byte
    addi  a1, a1, 1                      // move pointer to next text byte

tx_busy:
    lw    t0, SERP_TX_ST_REG_o(a0)       // read status of transmitter
    andi  t0, t0, SERP_TX_ST_REG_READY_m // mask ready bit
    beq   t0, zero, tx_busy              // if not ready wait for ready condition
    sw    t1, SERP_TX_DATA_REG_o(a0)     // write byte to Tx data register
    jal   zero, next_char                // unconditional branch to process next byte

end_char:
    ebreak // stop continuous execution, request developer interaction
    jal   zero, end_char

.org 0x400
.data

data_1: .word 1, 2, 3, 4 // example how to fill data words

text_1: .asciz "Hello world.\n" // store zero terminated ASCII text

```

FIGURE 2 – Exemple template .S de QtRVSim commenté. Les macros en majuscule sont définies au début du fichier (voir dans QtRVSim). Les registres du RISC-V sont détaillés en page 12

4 version pipelinée du RISC-V

QtRVSim propose une simulation de la version pipelinée du RISC-V que nous avons vu en cours. pour cela il suffit de faire une nouvelle simulation en cliquant sur *file -> new simulation*, de sélectionner le champs *“Pipeline with hazard unit and cache”* et de cliquer sur *“Example”*.

La nouvelle version de l’architecture pipelinée, ressemble à celle présentée en figure 3. Les grandes barres verticales sont les registres séparant les étapes du cycle de Von Neuman, on voit clairement les 5 étapes de pipeline. Lors de la simulation, à chaque step, vous verrez les chemins sélectionnés s’activer en gras dans l’architecture pour suivre le *datapath*.

1. Cliquer successivement pour "Step" vous voyez physiquement quelle instructions sont dans les zone *Instruction Fetch (IF)*, *Instruction Decode (ID)*, *Execute (Ex)*, *Memory (Mem)*, *Write back (WB)*.
2. Observez ce qu’il se passe lorsqu’on arrive à l’instruction *ebreak*, pouvez vous comprendre ce qu’il se passe?

Lorsque l’on arrive à *ebreak*, l’instruction suivante est d’abord chargé comme si de rien était, mais comme le *jal* (exécuté avant *ebreak*) a été décodé et exécuté, le prochain cycle remplace les deux instructions suivant *ebreak* par des *nop*, il s’agit de deux “bulles” qui vont permettre de “corriger” l’instruction suivante et de charger le *lb* au prochain cycle (cycle ou d’ailleurs l’instruction *break* n’est pas exécuté, elle est à son tour remplacée par un *nop*). On a donc 3 bulles insérées.

Ici il faut qu’ils comprennent la notion de bulle dont j’ai parlé en cours. Accessoirement ça explique pourquoi on n’avait pas de bonne performance en pipeliné parce qu’il y a une mauvaise prédiction de l’instruction suivante à chaque tour de boucle donc on perd 3 cycles à chaque tour de boucles, pour une boucle de 8 cycles, c’est trop.

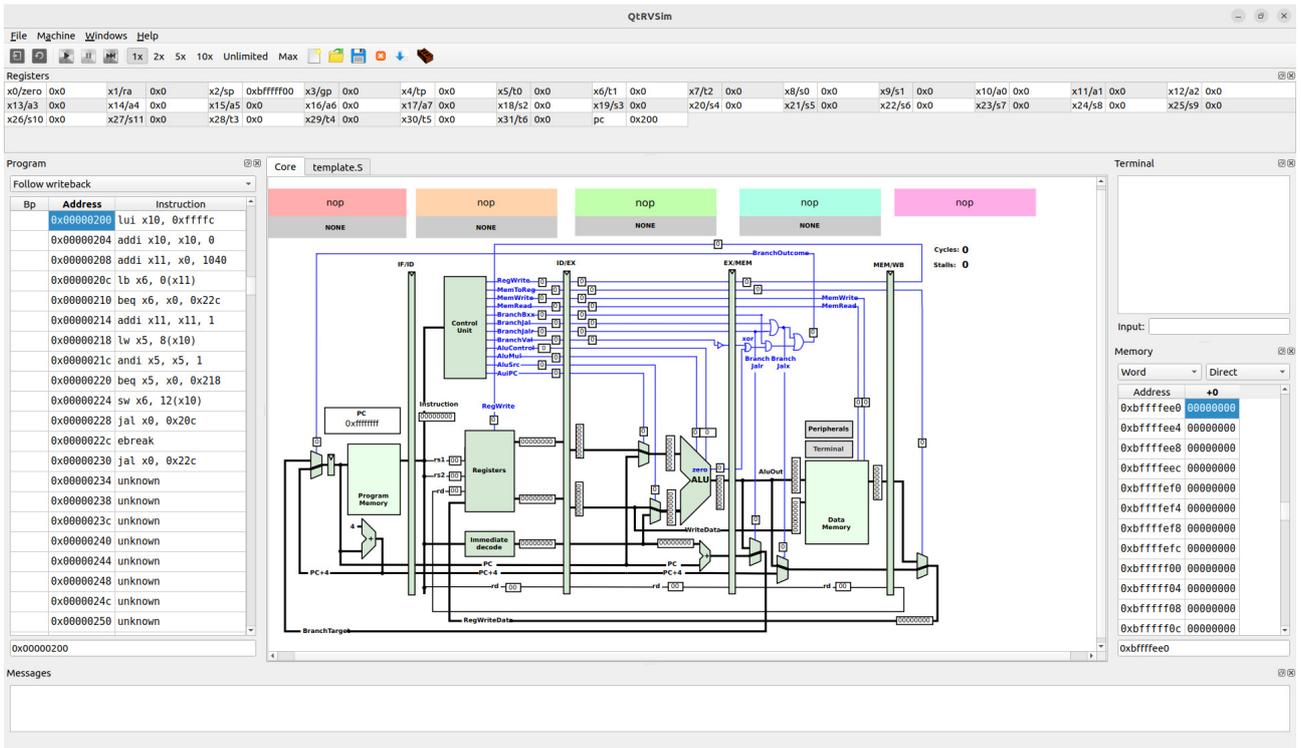


FIGURE 3 – Capture d'écran de l'interface de QtRVSim avec pipeline des instructions

5 Utilisation de la pile : une simple boucle while

Cette section étudie l'exécution de ce programme C simple :

```
int main()
{
    int x = 10;
    while (x != 0)
        x = x-1;
    return x;
}
```

Le code assembleur pour ce programme est affiché en Figure 4. Ce code a été généré par le compilateur pour RISC-V avec l'option -O0, voir l'encadré ci-dessous.

```

#pragma qtrvsim show terminal
#pragma qtrvsim show registers
#pragma qtrvsim show memory

.globl _start
.globl __start

.org 0x00000200

.text

__start:
_start:
    addi    sp,sp,-32    # reserve 32 bytes in stack
    sd     s0,24(sp)    # Store s0 in stack (s0 used by function)
    addi    s0,sp,32    # s0 <- fp (frame pointer)
    li     a5,10        # a5 <- 10
    sw     a5,-20(s0)   # Store a5 in stack
    j      L2

L3:
    lw     a5,-20(s0)   # get a5 from stack
    addiw  a5,a5,-1     # a5 <- a5 - 1
    sw     a5,-20(s0)   # store a5 in stack

L2:
    lw     a5,-20(s0)   # get a5 from stack
    sext.w a5,a5        # sign extension (32 -> 64 bits)
    bne    a5,zero,L3   # branch L3 is a5 != 0
    lw     a5,-20(s0)   # get a5 from stack again (here a5 = 0)
    mv     a0,a5        # a0 <- a5 (result of main)
    ld     s0,24(sp)    # restore s0
    addi    sp,sp,32    # restore sp
    jr     ra           # return from main

#pragma qtrvsim tab core

```

FIGURE 4 – Simple while loop Risc-V assembly code generated from C-code (options -O0)

Générer du code pour QtRVSim

Générer du code assembleur avec le compilateur `gcc` se fait simplement avec l'option `-S`. Mais pour générer du code pour RISC-V, il faut un *cross-compileur*, c'est à dire un compilateur qui s'exécute sur une machine Intel mais qui génère du code RISC-V. Cette chaîne de compilation est disponible pour linux, par exemple ici <https://github.com/riscv-collab/riscv-gnu-toolchain> ou plus simplement avec le paquet `gcc-riscv64-linux-gnu` sur Ubuntu (`gcc-riscv64-linux-gnu` est installé sur les machines du département).

Une fois le compilateur installé, il suffit de compiler le programme avec la commande suivante :

```
riscv64-linux-gnu-gcc -S whileLoop.c -o whileLoop.S
```

Si l'on fait ça, le code généré pour la fonction `main` est le suivant :

```
main:
    li a0,0
    ret
```

En effet le compilateur `riscv64-linux-gnu-gcc` a optimisé le code (optimisation `-O2` par défaut) et il a été capable de se rendre compte qu'à la fin de la boucle, `x` vaudrait 0, donc il a remplacé tout le programme par un simple `return 0`. C'est une bonne illustration de la puissance des optimisations des compilateurs aujourd'hui.

Pour éviter cette optimisation, on peut par exemple forcer à ne pas faire d'optimisation avec l'option `-O0` :

```
riscv64-linux-gnu-gcc -S whileLoop.c -o whileLoop.S -O0
```

Le code généré n'est toujours pas directement exécutable par QtRVSim parce qu'un certain nombre de directives mises en place par `gcc` ne sont pas comprises par QtRVSim, mais il est facile de les remplacer à la main et de garder le code assembleur, c'est ce qui est fait dans la figure 4

1. Repassez en mode *no pipeline, no cache* (*File* → *new simulation* → *start empty*)
2. Chargez le programme `whileLoopQtRVSim.S` dans le simulateur, pour cela :
 - (a) Fermez l'onglet `template.S` si il est encore ouvert (*File* → *Close source*)
 - (b) Chargez le programme `whileLoopQtRVSim.S` en cliquant sur le bouton *Open Source* : . le programme assembleur doit apparaître dans un nouvel onglet au centre.
 - (c) **Compiler** ce programme avec le bouton *compile source and update memory* :  (n'oubliez pas ! sinon il n'est pas pris en compte pas `qtrvsim`).
 - (d) Lancer l'exécution pas à pas en appuyant sur *step*
3. Que fait l'instruction `addi sp, sp, -32` ?

Elle réserve 32 octets dans la pile pour l'exécution de la fonction `main`. le programme n'utilise pas toute la pile, juste deux emplacements mais ça doit être une conséquence l'option `-O0`.

4. A quoi servent les manipulation sur `s0` dans les deux première instruction ? (rappeler vous le rôle de `s0`, cf page 12)

`s0` est le *frame pointeur*, c'est par `s0` qu'on accède aux différent champ de l'enregistrement d'activation (frame) de la fonction `main`. Notamment `a5` sera stocké à `s0+20`. L'ancienne version de `s0/fp`, i.e. le *frame pointer* de la fonction appelant `main`, est stockée dans la pile, à `sp+20`. Et la nouvelle valeur de, `s0/fp` est mis à la valeur du sommet de pile avant modification du sommet de pile (i.e. `sp+32`). Un petit schéma au tableau pour leur expliquer

- Allez vérifier que la valeur de la variable `x` du programme C est bien stockée dans la pile (i.e. explorez la mémoire en bas à droite) et qu'elle est décrémentée au cours de l'exécution du programme.

On entre l'adresse de la pile `0xbffffee0` dans la case en bas à droite, et on tape retour chariot. on voit ensuite 4 cases plus bas, la valeur `0x000000a` (à l'adresse `0xbffffedc`).

- Appuyer sur `step` jusqu'à ce que la valeur de `x` en mémoire atteigne 0, comprenez vous les 5 instructions restantes?

```
lw    a5,-20(s0)    # get a5 from stack again (here a5 = 0)
mv    a0,a5        # a0 <- a5 (result of main: 0)
lw    s0,24(sp)    # restore frame pointeur of fcnt calling main
addi  sp,sp,32     # restore sp
jr    ra          # return from main
```

- Comprenez vous maintenant le programme de la figure 4 complètement?

6 Fibonacci : utilisation de la pile pour un appel récursif

- Visualisez le programme `fib.c` ci dessous, comprenez son exécution. Si vous voulez le compiler et l'exécuter, utilisez la commande suivante :

```
gcc fib.c -o fib
```

Puis exécutez le programme en tapant `./fib`

```
#include <stdio.h>

int fib (int i)
{
    if (i<=1) return(1);
    else return(fib(i-1)+fib(i-2));
}

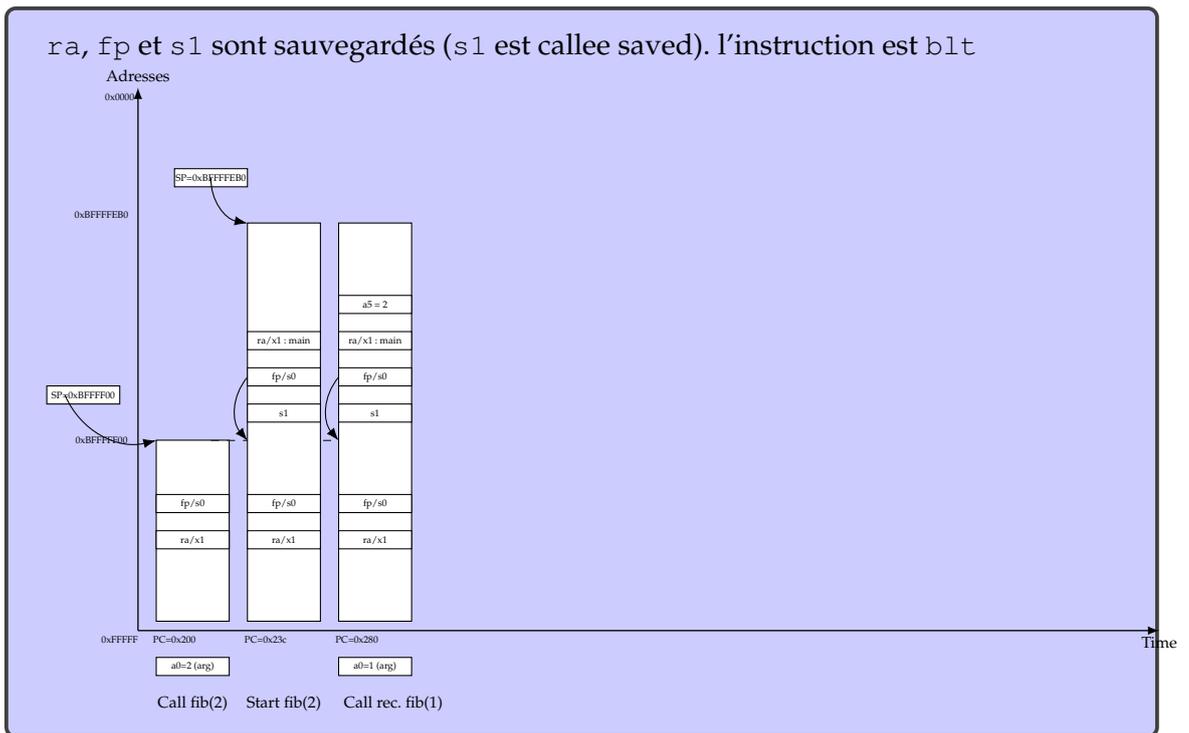
int main (int argc, char *argv[])
{
    fib(2);
}
```

- Charger la version assembleur dans QtRVSim, c'est le fichier `fib-00-QTRVSim.S` (l'appel a `printf` a été commenté car il ne marche pas dans QtRVSim).
- Comprenez le programme assembleur `fib-00-QTRVSim.S`, il a été commenté en Fig. 5
- Chargez et compilez le programme dans QtRVSim. Suivez l'exécution pas à pas en dessinant l'état de la pile, à partir du label `fib`, en supposant que le registre `$a0` contienne la valeur 2, argument transmit à `fib`, suivez les étapes suivantes :
 - Appuyez sur `step` jusqu'à l'appel de `fib` (instruction `jalr ra, 20(t1)` à l'adresse `0x228`. dessinez la pile (à l'envers, i.e. grandes adresses en bas) à ce moment du programme. A quoi correspond ce `jalr`? que contient le registre `a0`?

Ce `jalr` correspond au `call fib`, c'est donc l'appel de la fonction `fib`, l'argument est dans le registre `a0 : 2`



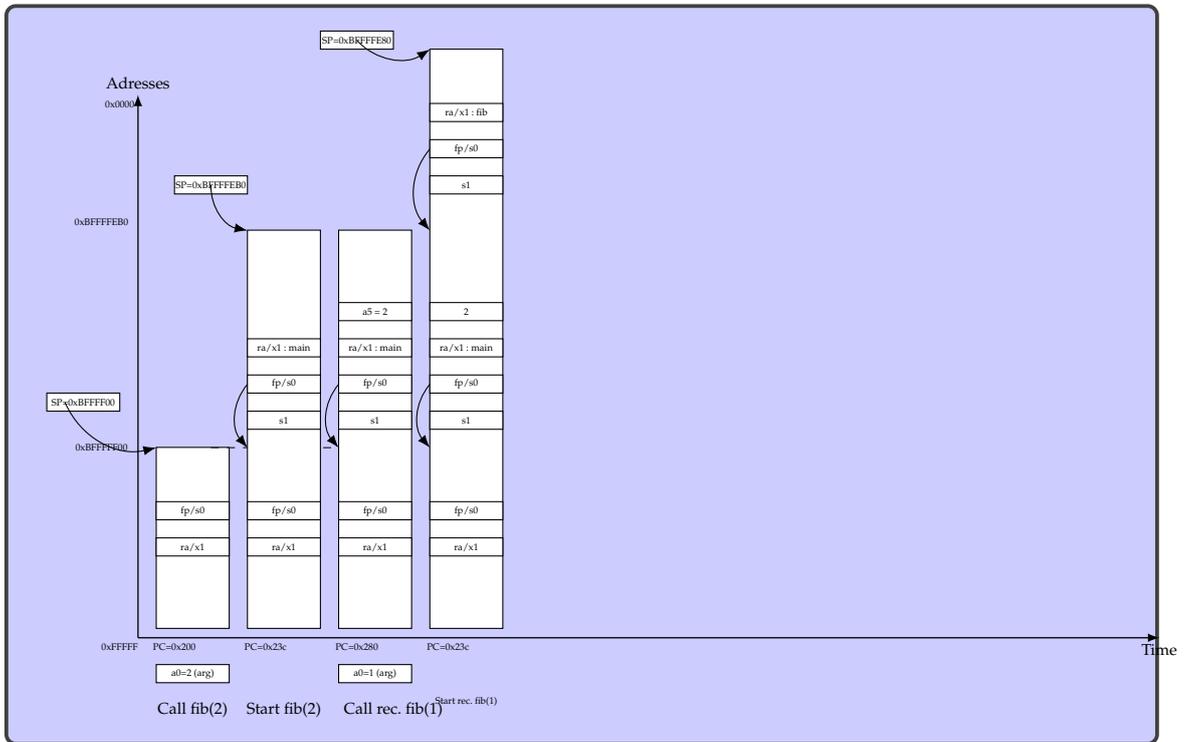
- Appuyer sur step pour visualiser les registres sauvegardés dans la pile, quels sont-ils et pourquoi sont-ils sauvegardés? ou est l'instruction de test $i \leq 1$? allez jusqu'à cette instruction et dessinez la pile dans cet état.



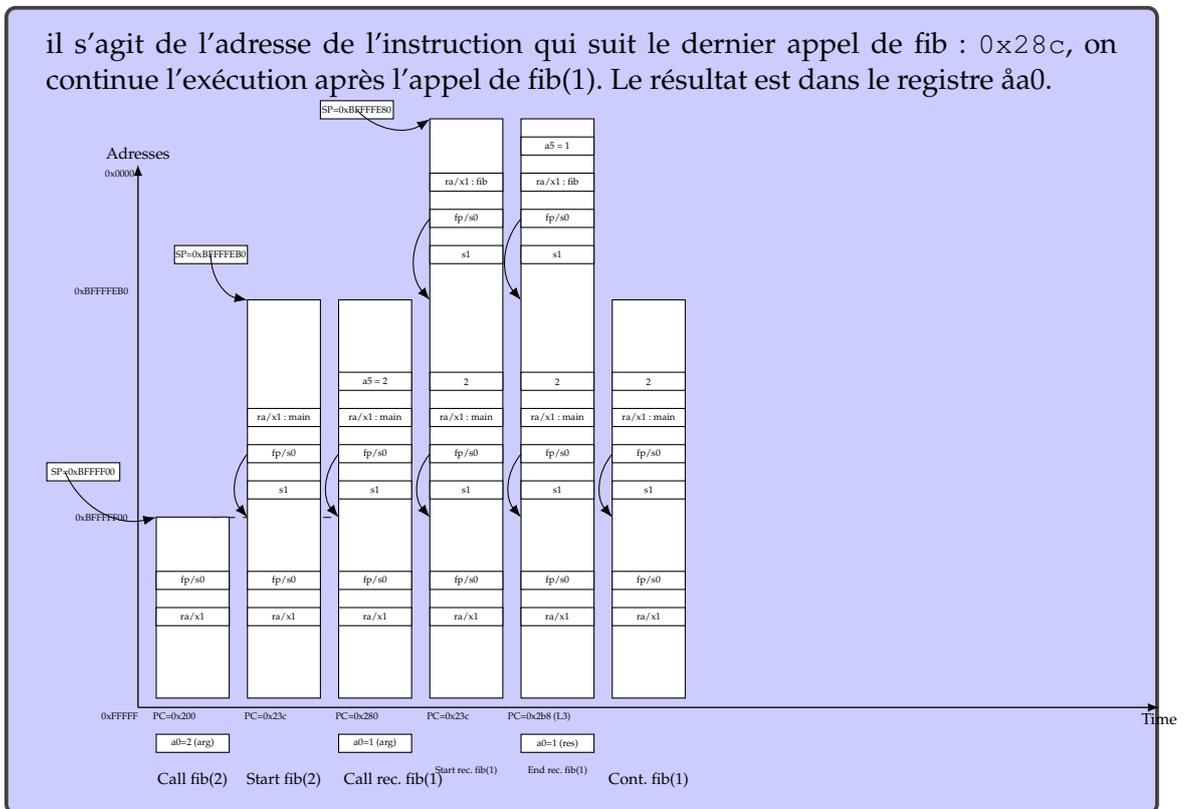
- Appuyer sur step jusqu'à l'instruction d'appel récursif à fib (adresse 0x288), quelle est la valeur de a0 à ce moment là, et que représente cette valeur?

a0 contient la valeur 1 qui est i-1 ou i est l'argument de fib, on va donc appeler fib(i-1)

- Appuyer 4 fois sur step et dessiner le nouvel état de la pile.



- Appuyez sur step jusqu'à l'instruction de comparaison à 1 (`blt`), cette fois ci la fonction prend l'autre branche (puisque $i \leq 1$) et va simplement retourner 1. Visualisez pas à pas la restauration des registres `ra`, `s0` et `s1` quelle va être l'adresse de retour indiquée dans `ra`? Dessinez le nouvel état de la pile après le retour de l'appel `fib(1)`. ou est le résultat de l'appel `fib(1)` ?



- Continuer à dérouler le programme pour le deuxième appel récursif (`fib(0)`), quel est le résultat de l'appel à `fib(2)` et dans quel registre est-il mis. Dessinez la pile après le retour à la fonction `main`

Toujours dans `a0`, le résultat est 2 : 1+1. La pile est dessinée dans la Fig. 6

```

fib:
    addi    sp,sp,-48    #set stack for fib (48 Bytes = 12 words)
    sw     ra,40(sp)    #store return adress in stack
    sw     s0,32(sp)    #store s0 in stack (callee saved)
    sw     s1,24(sp)    #store s1 in stack (callee saved)
    addi    s0,sp,48    #set frame pointer to old SP
    mv     a5,a0        #get function argument
    sw     a5,-36(s0)   #store argument (i) in stack
    lw     a5,-36(s0)   #? it seems that prev. instr. set a5 to 0
    sext.w a4,a5        #a4 <- i
    li     a5,1         #a5 <- 1
    bgt    a4,a5,L2     # branch to L1 if i > 1
    li     a5,1         # (else branch) set R5 to 1 (result)
    j      L3           # branch L3

L2:
    lw     a5,-36(s0)   #get i (argument of fib)
    addiw  a5,a5,-1     #compute i-1
    sext.w a5,a5        #useless ?
    mv     a0,a5        #set i-1 in argument register (s0)
    call   fib          #recursive call to fib(i-1)
    mv     a5,a0        #get result from recursive call fib(i-1)
    mv     s1,a5        #put result in s1
    lw     a5,-36(s0)   #get i (argument of fib)
    addiw  a5,a5,-2     #compute i-2
    sext.w a5,a5        #useless ?
    mv     a0,a5        #set i-2 in argument register (s0)
    call   fib          #recursive call to fib(i-2)
    mv     a5,a0        #get result from recursive call fib(i-2)
    addw   a5,s1,a5     #a5 <- fib(i-1) + fib(i-2)
    sext.w a5,a5        #useless?

L3:
    mv     a0,a5        #put result (1) in a0
    lw     ra,40(sp)    #restore return adresse
    lw     s0,32(sp)    #restore frame pointer
    lw     s1,24(sp)    #restore s1
    addi    sp,sp,48    #remove fib stack
    jr     ra           #return to caller code

```

FIGURE 5 – Code assembleur RiscV commenté correspondant au programme `fib.c` (generated from C-code options `-O0`)

5. Vérifier sur la figure 6 que vous comprenez bien l'évolution de la pile avec des appel de fonctions.
6. Comment peut-on voir l'assembleur x86 pour le programme `fib`.

```
gcc fib.c -S -o fibx86.s
```

L'idée c'est de leur laisser chercher sur internet (ou avec man!) l'option de `gcc`. par contre une fois trouvé, on n'est pas tenu de leur expliquer, perso je ne suis pas fluent en assembleur x86.

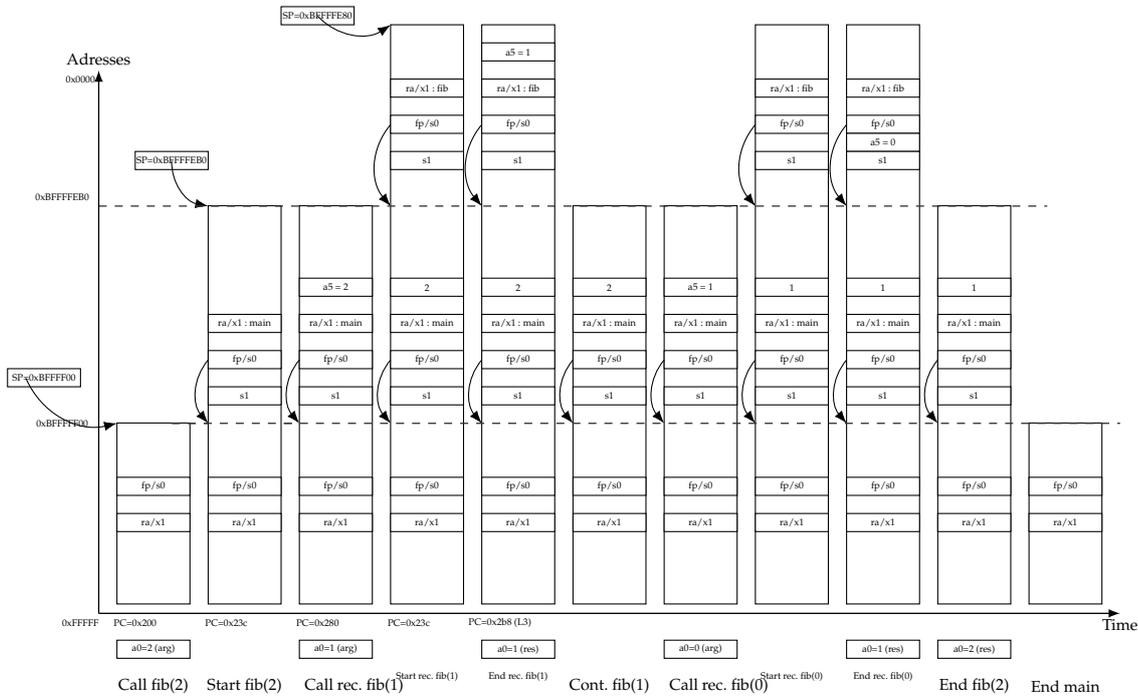


FIGURE 6 – Différents état de la pile lors de l'exécution de l'appel fib(2)

A Appendix 1 : quick reference de l'ISA RISC-V

RISC-V card obtained from James Zhu from Berkeley University.

Registers

| Register | ABI Name | Description | Saver |
|----------|----------|-----------------------|--------|
| x0 | zero | Zero constant | — |
| x1 | ra | Return address | Caller |
| x2 | sp | Stack pointer | Callee |
| x3 | gp | Global pointer | — |
| x4 | tp | Thread pointer | — |
| x5-x7 | t0-t2 | Temporaries | Caller |
| x8 | s0 / fp | Saved / frame pointer | Callee |
| x9 | s1 | Saved register | Callee |
| x10-x11 | a0-a1 | Fn args/return values | Caller |
| x12-x17 | a2-a7 | Fn args | Caller |
| x18-x27 | s2-s11 | Saved registers | Callee |
| x28-x31 | t3-t6 | Temporaries | Caller |
| f0-7 | ft0-7 | FP temporaries | Caller |
| f8-9 | fs0-1 | FP saved registers | Callee |
| f10-11 | fa0-1 | FP args/return values | Caller |
| f12-17 | fa2-7 | FP args | Caller |
| f18-27 | fs2-11 | FP saved registers | Callee |
| f28-31 | ft8-11 | FP temporaries | Caller |

RISC-V Instruction Set

Core Instruction Formats

| | | | | | | | | | | | | | | |
|-----------------------|----|----|-----|----|-----|----|--------|----|-------------|----|--------|---|---|--------|
| 31 | 27 | 26 | 25 | 24 | 20 | 19 | 15 | 14 | 12 | 11 | 7 | 6 | 0 | |
| funct7 | | | rs2 | | rs1 | | funct3 | | rd | | opcode | | | R-type |
| imm[11:0] | | | | | rs1 | | funct3 | | rd | | opcode | | | I-type |
| imm[11:5] | | | rs2 | | rs1 | | funct3 | | imm[4:0] | | opcode | | | S-type |
| imm[12 10:5] | | | rs2 | | rs1 | | funct3 | | imm[4:1 11] | | opcode | | | B-type |
| imm[31:12] | | | | | | | | | rd | | opcode | | | U-type |
| imm[20 10:1 11 19:12] | | | | | | | | | rd | | opcode | | | J-type |

RV32I Base Integer Instructions

| Inst | Name | FMT | Usage | Description (C) |
|--------|-------------------------|-----|--------------------|------------------------------|
| add | ADD | R | add rd, rs1, rs2 | rd = rs1 + rs2 |
| sub | SUB | R | sub rd, rs1, rs2 | rd = rs1 - rs2 |
| xor | XOR | R | xor rd, rs1, rs2 | rd = rs1 ^ rs2 |
| or | OR | R | or rd, rs1, rs2 | rd = rs1 rs2 |
| and | AND | R | and rd, rs1, rs2 | rd = rs1 & rs2 |
| sll | Shift Left Logical | R | sll rd, rs1, rs2 | rd = rs1 << rs2 |
| srl | Shift Right Logical | R | srl rd, rs1, rs2 | rd = rs1 >> rs2 |
| sra | Shift Right Arith* | R | sra rd, rs1, rs2 | rd = rs1 >> rs2 |
| slt | Set Less Than | R | slt rd, rs1, rs2 | rd = (rs1 < rs2)?1:0 |
| sltu | Set Less Than (U) | R | sltu rd, rs1, rs2 | rd = (rs1 < rs2)?1:0 |
| addi | ADD Immediate | I | addi rd, rs1, imm | rd = rs1 + imm |
| xori | XOR Immediate | I | xorii rd, rs1, imm | rd = rs1 ^ imm |
| ori | OR Immediate | I | orii rd, rs1, imm | rd = rs1 imm |
| andi | AND Immediate | I | andi rd, rs1, imm | rd = rs1 & imm |
| slli | Shift Left Logical Imm | I | slli rd, rs1, imm | rd = rs1 << imm[0:4] |
| srlr | Shift Right Logical Imm | I | srlr rd, rs1, imm | rd = rs1 >> imm[0:4] |
| srair | Shift Right Arith Imm | I | srair rd, rs1, imm | rd = rs1 >> imm[0:4] |
| slti | Set Less Than Imm | I | slti rd, rs1, imm | rd = (rs1 < imm)?1:0 |
| sltiu | Set Less Than Imm (U) | I | sltiu rd, rs1, imm | rd = (rs1 < imm)?1:0 |
| lb | Load Byte | I | lb rd, imm(rs1) | rd = M[rs1+imm][0:7] |
| lh | Load Half | I | lh rd, imm(rs1) | rd = M[rs1+imm][0:15] |
| lw | Load Word | I | lw rd, imm(rs1) | rd = M[rs1+imm][0:31] |
| lbu | Load Byte (U) | I | lbu rd, imm(rs1) | rd = M[rs1+imm][0:7] |
| lhu | Load Half (U) | I | lhu rd, imm(rs1) | rd = M[rs1+imm][0:15] |
| sb | Store Byte | S | sb rd, imm(rs1) | M[rs1+imm][0:7] = rs2[0:7] |
| sh | Store Half | S | sh rd, imm(rs1) | M[rs1+imm][0:15] = rs2[0:15] |
| sw | Store Word | S | sw rd, imm(rs1) | M[rs1+imm][0:31] = rs2[0:31] |
| beq | Branch == | B | beq rs1, rs2, imm | if(rs1 == rs2) PC += imm |
| bne | Branch != | B | bne rs1, rs2, imm | if(rs1 != rs2) PC += imm |
| blt | Branch < | B | blt rs1, rs2, imm | if(rs1 < rs2) PC += imm |
| bge | Branch ≥ | B | bge rs1, rs2, imm | if(rs1 ≥ rs2) PC += imm |
| bltu | Branch < (U) | B | bltu rs1, rs2, imm | if(rs1 < rs2) PC += imm |
| bgeu | Branch ≥ (U) | B | bgeu rs1, rs2, imm | if(rs1 ≥ rs2) PC += imm |
| jal | Jump And Link | J | jal rd, imm | rd = PC+4; PC += imm |
| jalr | Jump And Link Reg | I | jalr rd, rs1, imm | rd = PC+4; PC = rs1 + imm |
| lui | Load Upper Imm | U | lui rd, imm | rd = imm << 12 |
| auipc | Add Upper Imm to PC | U | auipc rd, imm | rd = PC + (imm << 12) |
| ecall | Environment Call | I | ecall | Transfer control to OS |
| ebreak | Environment Break | I | ebreak | Transfer control to debugger |

Pseudo Instructions

| Pseudoinstruction | Base Instruction(s) | Meaning |
|---------------------------|---|---------------------------------|
| la rd, symbol | auipc rd, symbol[31:12] addi rd, rd, symbol[11:0] | Load address |
| l{b h w d} rd, symbol | auipc rd, symbol[31:12] l{b h w d} rd, symbol[11:0] (rd) | Load global |
| s{b h w d} rd, symbol, rt | auipc rt, symbol[31:12] s{b h w d} rd, symbol[11:0] (rt) | Store global |
| fl{w d} rd, symbol, rt | auipc rt, symbol[31:12] fl{w d} rd, symbol[11:0] (rt) | Floating-point load global |
| fs{w d} rd, symbol, rt | auipc rt, symbol[31:12] fs{w d} rd, symbol[11:0] (rt) | Floating-point store global |
| nop | addi x0, x0, 0 | No operation |
| li rd, immediate | <i>Myriad sequences</i> | Load immediate |
| mv rd, rs | addi rd, rs, 0 | Copy register |
| not rd, rs | xori rd, rs, -1 | One's complement |
| neg rd, rs | sub rd, x0, rs | Two's complement |
| negw rd, rs | subw rd, x0, rs | Two's complement word |
| sext.w rd, rs | addiw rd, rs, 0 | Sign extend word |
| seqz rd, rs | sltiu rd, rs, 1 | Set if = zero |
| snez rd, rs | sltu rd, x0, rs | Set if ≠ zero |
| sltz rd, rs | slt rd, rs, x0 | Set if < zero |
| sgtz rd, rs | slt rd, x0, rs | Set if > zero |
| fmv.s rd, rs | fsgnj.s rd, rs, rs | Copy single-precision register |
| fabs.s rd, rs | fsgnjx.s rd, rs, rs | Single-precision absolute value |
| fneg.s rd, rs | fsgnjn.s rd, rs, rs | Single-precision negate |
| fmv.d rd, rs | fsgnj.d rd, rs, rs | Copy double-precision register |
| fabs.d rd, rs | fsgnjx.d rd, rs, rs | Double-precision absolute value |
| fneg.d rd, rs | fsgjnd.d rd, rs, rs | Double-precision negate |
| beqz rs, offset | beq rs, x0, offset | Branch if = zero |
| bnez rs, offset | bne rs, x0, offset | Branch if ≠ zero |
| blez rs, offset | bge x0, rs, offset | Branch if ≤ zero |
| bgez rs, offset | bge rs, x0, offset | Branch if ≥ zero |
| bltz rs, offset | blt rs, x0, offset | Branch if < zero |
| bgtz rs, offset | blt x0, rs, offset | Branch if > zero |
| bgt rs, rt, offset | blt rt, rs, offset | Branch if > |
| ble rs, rt, offset | bge rt, rs, offset | Branch if ≤ |
| bgtu rs, rt, offset | bltu rt, rs, offset | Branch if >, unsigned |
| bleu rs, rt, offset | bgeu rt, rs, offset | Branch if ≤, unsigned |
| j offset | jal x0, offset | Jump |
| jal offset | jal x1, offset | Jump and link |
| jr rs | jalr x0, rs, 0 | Jump register |
| jalr rs | jalr x1, rs, 0 | Jump and link register |
| ret | jalr x0, x1, 0 | Return from subroutine |
| call offset | auipc x1, offset[31:12] jalr x1, x1, offset[11:0] | Call far-away subroutine |
| tail offset | auipc x6, offset[31:12] jalr x0, x6, offset[11:0] | Tail call far-away subroutine |
| fence | fence iorw, iorw | Fence on all memory and I/O |

B Uniquement pour les enseignant : Compiler soit même un programme pour QtRVSim

Le compilateur `riscv64-linux-gnu-gcc` génère un code qu'il est nécessaire retoucher un peu à la main si on veut le simuler dans QtRVSim.

A titre d'exemple, prenons le code `CNP.c` (Fig. B), on notera qu'il n'y a aucune appel à `printf` bien sûr.

Le programme `CNP.c` que l'on va compiler pour QtRVSim

On rappelle que la commande pour compiler ce programme en un programme assembleur RISC-V est :

```
riscv64-linux-gnu-gcc -S CNP.S -o CNP.S
```

Le programme généré est montré en partie en Fig. B.

A noter que la figure B est obtenue en demandant à Chatgpt : "peux tu aussi m'expliquer ce programme riscV là et mettre tes explications en commentaire? :". Ca n'explique pas le fonctionnement du programme mais quand on ne connaît pas un assembleur, ça paraît utile. cependant je n'ai pas vérifié que c'était juste, on peut se poser la question quand on voit que Chatgpt a inversé les position des labels `.L2` et `.L3` (mais le code correspondant à chaque label est le bon).

```
.file      "CNP.c"
          .option pic
          .text
          .align      1
          .globl      cnp
          .type       cnp, @function
cnp:
    addi    sp,sp,-48
    sd     ra,40(sp)
    sd     s0,32(sp)
    addi    s0,sp,48
    mv     a5,a0
    mv     a4,a1
    sw     a5,-36(s0)
    [.....]
    lw     a5,-40(s0)
    sext.w a5,a5
    bne    a5,zero,.L3
.L2:
    li     a5,1
    sw     a5,-28(s0)
    j     .L4
.L3:
    lw     a5,-36(s0)
    addiw  a5,a5,-1
    [.....]
main:
    addi    sp,sp,-32
    sd     ra,24(sp)
    sd     s0,16(sp)
    addi    s0,sp,32
    [...]
```

Le programme `CNP.S` compilé avec `riscv64-linux-gnu-gcc`

```

.file "CNP.c"           // Directive pour indiquer le nom du fichier source

.option pic             // Option pour générer du code position-indépendant
.text                  // Début du segment de code

.align 1               // Alignement sur une adresse de puissance de 2 égale à 2^1 = 2 octets

.globl cnp              // Déclare la fonction cnp comme étant globalement accessible
.type cnp, @function   // Indique que cnp est une fonction

cnp:                   // Début de la fonction cnp
    addi sp,sp,-48     // Alloue 48 octets (12 registres) dans la pile pour les variables locales
    sd ra,40(sp)       // Sauvegarde la valeur de ra (registre de retour) dans la pile à l'offset 40
    sd s0,32(sp)       // Sauvegarde la valeur de s0 (registre de base de la pile) dans la pile à l'offset 32
    addi s0,sp,48      // Initialise s0 à la position actuelle de la pile + 48

    mv a5,a0           // Copie la valeur de a0 dans a5
    mv a4,a1           // Copie la valeur de a1 dans a4
    sw a5,-36(s0)      // Stocke la valeur de a5 à l'offset -36 de la
    [...]
    lw a5,-40(s0)      // Charge la valeur à l'offset -40 de la pile dans a5
    mv a4,a5           // Copie la valeur de a5 dans a4
    lw a5,-36(s0)      // Charge la valeur à l'offset -36 de la pile dans a5
    sext.w a4,a4       // Étend la valeur de a4 à 32 bits en signant
    sext.w a5,a5       // Étend la valeur de a5 à 32 bits en signant
    beq a4,a5,.L2      // Saut à .L2 si a4 est égal à a5

.L3:                   // Étiquette .L3
    lw a5,-40(s0)      // Charge la valeur à l'offset -40 de la pile dans a5
    sext.w a5,a5       // Étend la valeur de a5 à 32 bits en signant
    bne a5,zero,.L3    // Saut à .L3 si a5 n'est pas égal à zéro

.L2:                   // Étiquette .L2
    li a5,1            // Charge immédiatement la valeur 1 dans a5
    sw a5,-28(s0)      // Stocke la valeur de a5 à l'offset -28 de la position de la pile s0
    j .L4              // Saut à .L4

[...]
main:                  // Début de la fonction main
    addi sp,sp,-32     // Alloue 32 octets (8 registres) dans la pile pour les variables locales
    sd ra,24(sp)       // Sauvegarde la valeur de ra (registre de retour) dans la pile à l'offset 24
    sd s0,16(sp)       // Sauvegarde la valeur de s0 (registre de base de la pile) dans la pile à l'offset 16
    addi s0,sp,32      // Initialise s0 à la position actuelle de la pile + 32

```

Le programme CNP.S compilé avec riscv64-linux-gnu-gcc

Les changements à effectuer sur le fichier de la figure B sont les suivants :

- Enlever les directives du compilateur gcc et introduire les pragmas nécessaires à QtRVSim
- Renommer le label main en _start et la mettre au début du programme.
- Renommer les labels .Lx en Lx (QtRVSim n'aime pas les labels commençant par un point).

Enfin rechargez le fichier CNP_QTRVSim.S dans QtRVSim et n'oubliez pas de vérifier qu'il compile en exécutant "compile source and update memory" sans erreurs! (vérifiez dans la console, si il y a des erreurs le binaire n'est pas chargé en mémoire.

```

#pragma qtrvsim show terminal
#pragma qtrvsim show registers
#pragma qtrvsim show memory

.globl _start
.globl __start

.org 0x00000200

.text

_start:
    addi sp,sp,-32
    sd ra,24(sp)
    sd s0,16(sp)
    addi s0,sp,32
    mv a5,a0
    [...]
cnp:
    addi sp,sp,-48
    sd ra,40(sp)
    sd s0,32(sp)
    addi s0,sp,48
    [...]
    ld ra,40(sp)
    ld s0,32(sp)
    addi sp,sp,48
    jr ra

#pragma qtrvsim tab core

```

Le programme CNP.S modifié à la main pour QtRVSim