ARC TD7

QTRVSIM : un simulateur de l'architecture RISC-V
(2h séance sur machine), 9 avril 2024

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 (instuctions 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 GUI soit dans un naviguateur (`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`.
   En haut de l'interface vous voyez l'état des 32 registres avec leurs deux noms. Sur la gauche vous voyez les instructions assembleurs avec leur addresse en mémoire (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 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.
2. 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 executée. Repérez sur la schématique du processeur, la valeur du *program counter* PC. Repérez, toujours sur la schématique, la valeur en héxadécimal du registre d'instruction. Repérez la nouvelle valeur du registre `x10`, reportez vous à la liste des instructions en fin de TD pour comprendre l'instruction `lui`. Quelle est la valeur du stack pointer SP ?
3. 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 droit, 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. 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 a 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’addresse `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 les sections de code (`.text`) et data (`.data`), les directives `.org` indiquent à quelles addresses vont être chargées ces sections en mémoire. À quelle adresse commence le code ? À quelle addresse sont rangée les données ? Notamment la chaîne de caractère `"Hello world"`. Allez explorer la mémoire pour retrouver ces données.
3. Lisez le programme et comprenez les étapes :
   (a) chargement de l’adresse du port série dans `a0`.
   (b) chargement de l’addresse 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 reception, 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.
4. Suivez l’exécution pas à pas du programme à nouveau, et verifiez l’affichage du caractère à chaque étape (g) ci-dessus.

4 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 2. Ce code a été généré par le compilateur pour RISC-V avec l’options -O0, voir l’encadré ci-dessous.

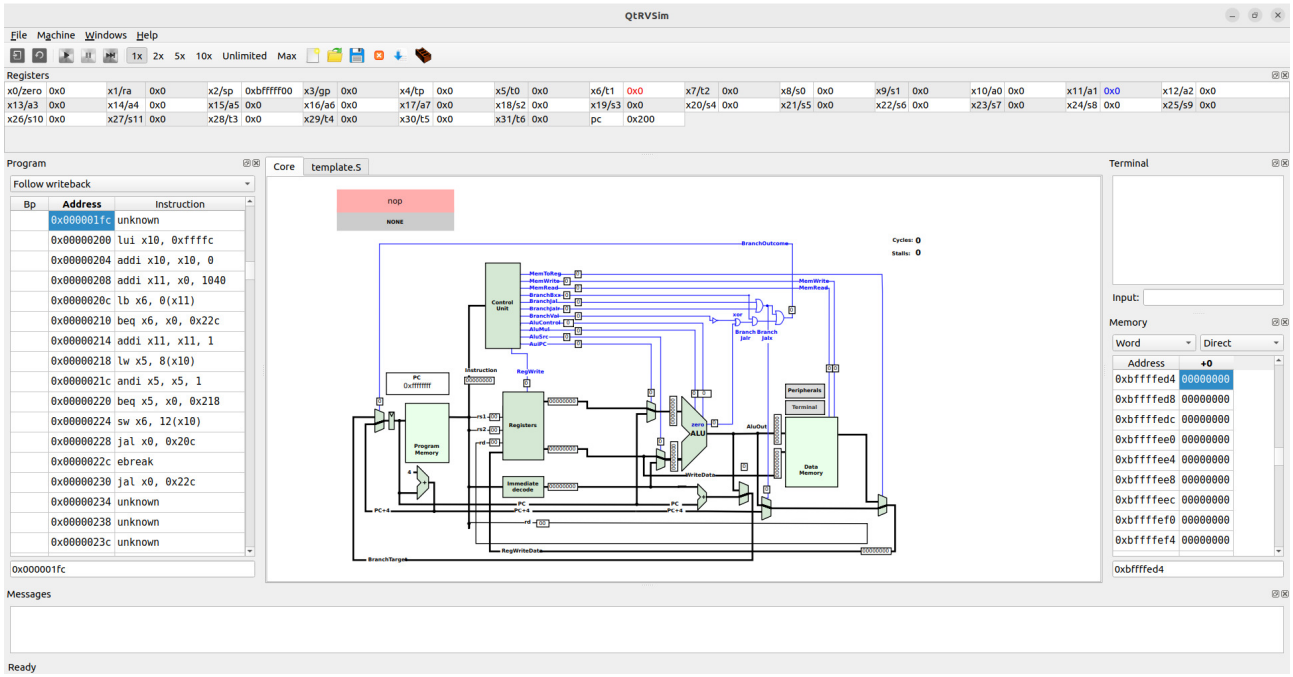

FIGURE 1 – Capture d'écran de l'interface de QtRVSim au démarrage

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-compilateur*, 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.
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 défault) 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 2

1. Chargez le programme `whileLoopQtRVSim.S` dans le simulateur, pour cela :
   (a) Fermez l'onglet `template.S`
   (b) Chargez le programme `whileLoopQtRVSim.S` en cliquant sur le bouton `Open Source` :

```
#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 2 – Simple `while` loop Risc-V assembly code generated from C-code (options -O0)

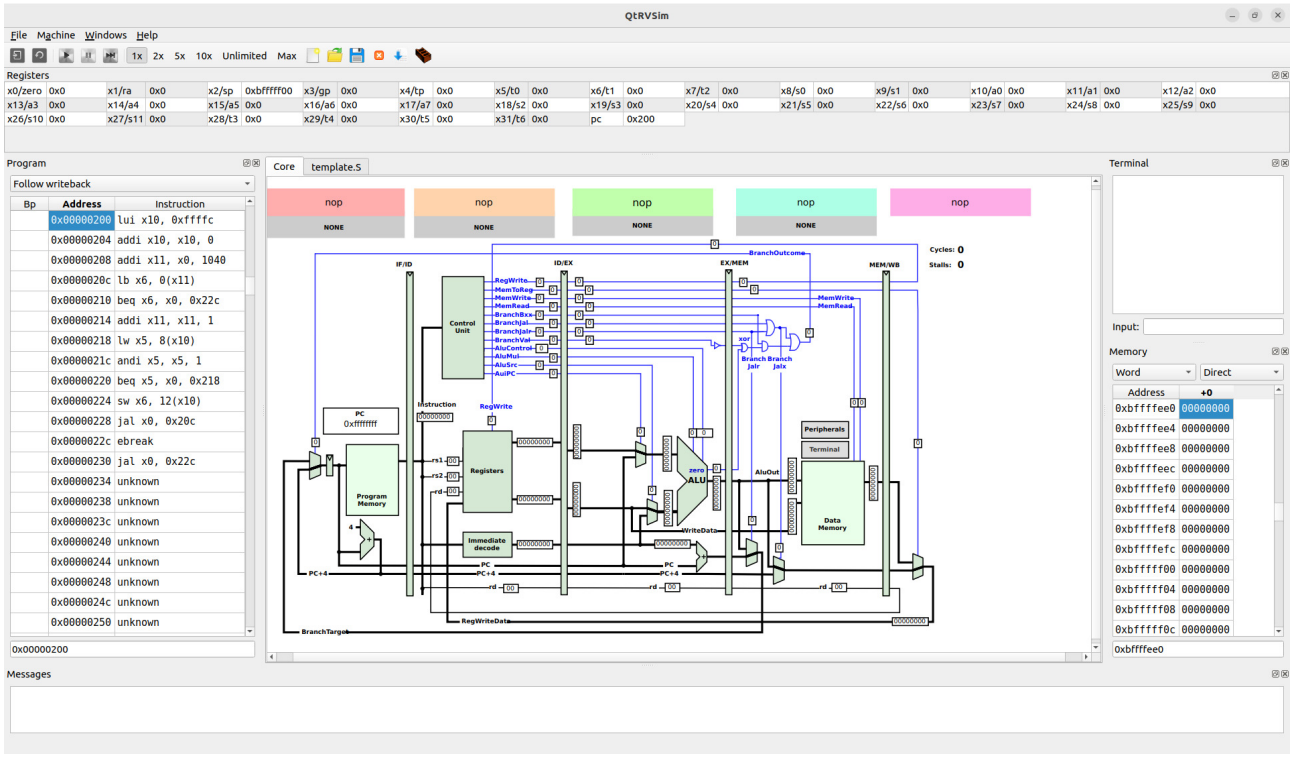

FIGURE 3 – Capture d'écran de l'interface de QtRVSim avec pipeline des instructions

(c) Compiler ce programme avec le bouton `compile source and update memory` : . le programme assembleur doit apparaître dans un nouvel onglet au centre.

(d) Lancer l'exécution pas à pas en appuyant sur `step`

2. Que fait l'instruction `addi sp,sp,-32` ?
3. Pourquoi stocke-t'on `s0` dans la pile, à quoi sert `s0` dans le programme ?
4. Allez vérifier que la valeur de la variable `x` du programme C est bien stocké dans la pile (i.e. explorez la mémoire en bas à droite) et qu'elle est mise à jour au cours de l'exécution du programme.
5. Comprenez vous maintenant le programme de la figure 2 complètement ?

5 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 selectionner le champs "*Pipeline withut hazard and without cache*" et de cliquer sur "*start empty*".
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 selectionnés 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. Est ce que le nombre de cycle nécessaire pour l'exécution du programme diminue dans la version pipelinées ?
3. En supposant que l'on puisse avoir une horloge 4 fois plus rapide pour la version pipelinée

6 Fibonacci : utilisation de la pile pour un appel récursif

1. Viusualisez 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[])
{
  printf("le resultat est %d",fib(2));
}
```

2. Charger la version assembleur dans QtRVSim, c'est le fichier `fib-risc-v-commented-O0.S` (l'appel a `printf` a été commenté car il ne marche pas dans QtRVSim).
3. Chargez le programme dans QtRVSim et suivez l'exécution pas à pas en dessinant l'état de la pile, à partir du label `fib`, en supposant que le register `$a0` contienne la valeur 2, argument transmit à `fib`

QUESTION 1 ► (question substdiaire) comment peut-on voir l'assembleur x86 pour le programme fib.

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

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

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]`
`srli`	Shift Right Logical Imm	I	`srli rd, rs1, imm`	`rd = rs1 >> imm[0:4]`
`srai`	Shift Right Arith Imm	I	`srai 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 $\geq$	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 $\geq$ (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	`auioc 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`	*Myriad sequences*	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 $\neq$ 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`	`fsgnjn.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 $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ 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 $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, 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

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