This work is licensed under a CC BY 4.0 license.©2023. Départements Informatique & SML, Esisar

1 A first program

```
.section .text
.globl main
main:
        li t1, 1                # t1 = 1
        la t2, data             # t2 = @ labeled by data
        ld t3, 0(t2)            # t3 = *data
loop:
        addi t1, t1, 2          # t1 = t1 + 2
        addi t3, t3, -1         # t3 = t3 - 1
        blt zero, t3, loop      # 0 < t3 => @ labeled by loop
        add a0, zero, t1        # return t1
        ret

.section .rodata
data:
        .dword 6                # double word = 64 bits
        .string "Hello RISCV"
```

Assembling + linking (via gcc): in a *terminal*:

```
riscv64-unknown-elf-gcc prog.s -o prog.riscv
```

Execution (simulation) If the previous command succeeds :

```
spike pk prog.riscv
```

2 Registers

32 general-purpose registers 32 or 64 bits: (x0, ...x31), but, x0 is wired to 0, and registers have symbolic names that we **should use when programming** :

Register	ABI name	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary/alternate link register
x6-7	t1-2	Temporaries
x8	s0/fp	Saved register/frame pointer
x9	s1	Saved register
x10-11	a0-1	Function args/return values
x12-17	a0-a7	Function args
x18-27	s2-s11	Saved registers
x28-31	t3-6	Temporaries

3 Instructions and pseudo instructions

For programming, we do not really care about pseudo or non pseudo instructions

Arithmetic instructions (in registers):

Instruction	Meaning
nop	No operation
add rd, rs1, rs2	Register add: rd = rs1 + rs2
sub rd, rs1, rs2	rd = rs1 - rs2
xor rd, rs1, rs2	rd = rs1 ^ rs2
or rd, rs1, rs2	rd = rs1 | rs2
and rd, rs1, rs2	rd = rs1 & rs2
sll rd, rs1, rs2	Shift Left Logical (rd = rs1 << rs2)
li rd, immediate	Load immediate
addi rd, rs1, imm	rd = rs1 + imm
mv rd, rs	Copy register
not rd, rs	One's complement
neg rd, rs	Two's complement

Load/Stores:

lb rd, rs1, imm	rd = M[rs1+imm][0:7] (Load byte)
lw rd, rs1, imm	rd = M[rs1+imm][0:31] (word)
sb	M[rs1+imm][0:7] = rs2[0:7] (Store byte)
sw	M[rs1+imm][0:31] = rs2[0:31]
la rd, symbol	Load address

Branching:

beq rs1,rs2, lbl	if(rs1 == rs2) jump to lbl
bne	Branch if not equal
blt,bge,bgt,ble...	...
beqz rs, offset	Branch if = zero
bltz,bgez,...	...
j lbl	Jump
ret	Return from subroutine
call lbl	Call subroutine

4 Stack handling in RISCV (Figure 1)

Register sp is reserved for the stack (used by functions to store local variables, works like a stack data structure). The RISCV ABI specifies that sp should store the address of the last value pushed on the stack.

Pushing a (value from a) register on the stack is thus :

```
addi sp, sp, -8        # place for a 64 bit register
sd REGISTERNAME, 0(sp)
```

and pop :

```
ld REGISTERNAME, 0(sp)
addi sp, sp, 8
```

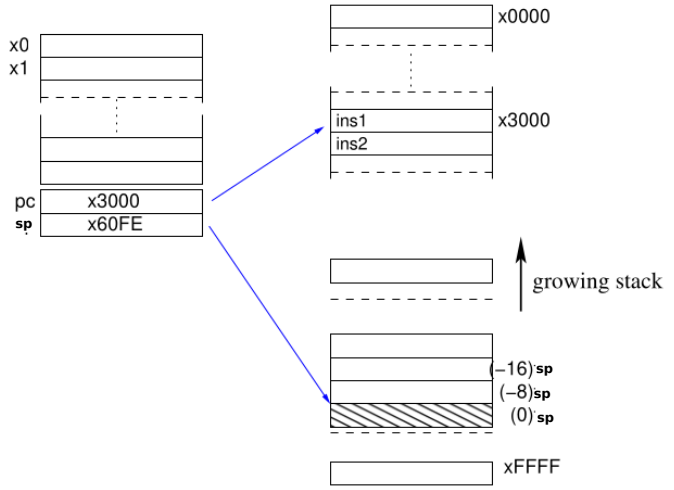

Figure 1: A modelisation of memory while executing RISCV programs. **Warning!** stack grows through decreasing addresses !

(Physical) Addresses are fake, and should not suggest a 16bit addressing mode.

5 Function calls

As a general rule, the **saved registers** `s0` to `s11` are preserved across function calls, while the others are not.

- For the caller: `si` registers will keep their values after the call; others not (you might have to save them on the stack).
- For the callee: be careful `si` registers should keep their value (or be saved and restored before `ret`).

```
prog:  # caller code
  # perhaps give a0 an input value
  call sub_prog
  # perhaps get a result in a0
  ...
  ret
```

```
sub_prog:  # code of the callee
  addi sp, sp, -8
  sd ra, 0(sp)    # return address is saved !
  # Some registers may have to be saved
  # Function's body
  ...
  # Some registers may have to be restored
  ld ra, 0(sp)    # restore return address
  addi sp, sp, 8
  ret
```