Computer Architecture
SimpleRISC Specification

Memory Model

The memory is byte-addressable. Multi-byte entities are stored in little endian form. The memory contains the program instructions, the static data, and the stack.

Registers

There are a total of 16 registers: `r0` to `r15`. Each register is 4 bytes wide. 16 registers require 4 bits for encoding. `r0` is encoded as `0000`, `r1` as `0001`, and so on.

Table 1: Registers in the custom ISA

Register	Purpose
`r0` to `r13`	General purpose
`r14` / `sp`	Stack Pointer
`r15` / `ra`	Return Address Register
`PC`	Program Counter

Instruction Formats

Table 2: Instruction formats in the SimpleRISC ISA

register type					
unused	`rs2`	`rs1`	`rd`	`immediate`	`opcode`
14 bits	4 bits	4 bits	4 bits	1 bit	5 bits
immediate type					
immediate		`rs1`	`rd`	`immediate`	`opcode`
18 bits		4 bits	4 bits	1 bit	5 bits
branch type					
offset					`opcode`
27 bits					5 bits

Instructions

Table 3: Instructions in the *SimpleRISC* ISA

Operation	Opcode	Format
add	00000	add rd, rs1, (rs2 / imm)
sub	00001	sub rd, rs1, (rs2 / imm)
mul	00010	mul rd, rs1, (rs2 / imm)
div	00011	div rd, rs1, (rs2 / imm)
mod	00100	mod rd, rs1, (rs2 / imm)
cmp	00101	cmp rs1, (rs2 / imm)
and	00110	and rd, rs1, (rs2 / imm)
or	00111	or rd, rs1, (rs2 / imm)
not	01000	not rd, (rs2 / imm)
mov	01001	mov rd, (rs2 / imm)
lsl	01010	lsl rd, rs1, (rs2 / imm)
lsr	01011	lsr rd, rs1, (rs2 / imm)
asr	01100	asr rd, rs1, (rs2 / imm)
nop	01101	nop
ld	01110	ld rd, imm[rs1] (rd ← [rs1+imm])
st	01111	st rd, imm[rs1] ([rs1+imm] ← rd)
beq	10000	beq offset
bgt	10001	bgt offset
b	10010	b offset
call	10011	call offset
ret	10100	ret
end	10101	end

Example Program

Factorial by Iteration

Consider the following program to compute the factorial of a number stored in memory at address `r0`. Assume that the number is greater than 2. Save the result in memory at the address `r0` + 4.

```
.main:
        ld r2, [r0]
        mov r1, 1
.loop:
        mul r1, r1, r2
        sub r2, r2, 1
        cmp r2, 1
        bgt .loop
        st r1, 4[r0]
        end
```

Function Calling Convention

Function arguments and return values can either be passed through the stack or through registers. Passing through the stack is the most powerful, and the least simple of the different alternatives. This method is discussed here.

Caller Behavior

- The caller function first pushes onto the stack all registers whose values it wishes to preserve for use *after* the function call. This includes the `ra` register.

  Pushing a value means decrementing the stack pointer by the size of the datum pushed (in bytes), and then performing a store to the address pointed to by the stack pointer. Similarly, popping a value means performing a load from the address pointed to by the stack pointer, and then incrementing the stack pointer by the appropriate number. Note that what is explained is the typical behavior – you may optimize the number of additions and subtractions.

- It then pushes all the arguments onto the stack.
- It sets the stack pointer `sp` to point to the top of the stack.
- It then invokes the `call` instruction. This saves the address of the next instruction (`PC` + 4) in the `ra` register. It then jumps to the first instruction in the function.
- Once the called function returns, it finds the return values in the addresses starting from the stack pointer `sp`.
- It then pops out all the register values it had earlier preserved, including `ra`.

Callee Behavior

- To access the arguments, it does so relatively based on the value of the stack pointer `sp`.
- As part of its work, it may perform further memory operations in the stack space, but only in addresses strictly lesser than the top of the calling function's activation block.
- Once it is done with its work, it removes all the function arguments from the stack (if it has not done so already).
- It then pushes all the values to be returned onto the stack. It does so in the positions just above the top of the calling function's activation block.
- It then jumps to the address contained in the `ra` register.

Example Programs

Factorial by Recursion: Program I

Consider the following program to compute the factorial of a number stored in `r0`. Assume that the number is greater than 2. Save the result in `r1`.

```
.factorial:
        cmp r0, 1          /* compare (1,num) */
        beq .return
        bgt .continue
        b .return
.continue:
        sub sp, sp, 8      /* create space on the stack */
        st r0, [sp]        /* push r0 on the stack */
        st ra, 4[sp]       /* push the return address register */
        sub r0, r0, 1      /* num = num - 1 */
        call .factorial    /* result will be in r1 */
        ld r0, [sp]        /* pop r0 from the stack */
        ld ra, 4[sp]       /* restore the return address */
        mul r1, r0, r1     /* factorial(n) = n * factorial(n-1) */
        add sp, sp, 8      /* delete the activation block */
        ret
.return:
        mov r1, 1
        ret
.main:
        mov r0, 10
        call .factorial /* result in r1 */
```