Simple Instruction Architecture

The simple instruction architecture (SIA) is designed to be an architecture that is easy to assemble, easy to create a virtual machine for and easy for beginners to computer architecture to understand. While a typical RISC (reduced instruction set computer/chip) has dozens of instructions. SIA has 16 instructions.

**Registers**

* SIA has 16 registers, numbered from 0-15 (4 bit selector).
* Unlike many RISC chips, register 0 is not a constant 0; it is a general-purpose register.
* All registers are 32 bits wide.
* All registers are interpreted as signed integers for the purpose of mathematical operations.

**Stacks**

This implementation of SIA has a stack:

* Hardware support for a stack data structure that can hold function parameters, local variables, return addresses, and return values.
* R15 is the stack register – its current address is the bottom most byte of the stack.

**Instruction Usage**

To keep the instruction count low, a number of instructions that are traditionally available on most systems are omitted in SIA. Most of these are easily replaced with one or more SIA instructions.

For example:

* There is no “clear” instruction to set a register to 0. This is easily fixed by using “sub”:

**sub r1 r1 r1** will subtract r1 from itself, giving 0.

* There is no load immediate to set a register to a particular value. This can be replaced with 2 instructions:  
   **sub r1 r1 r1 //set r1 to 0**

**addimmediate r1 25 //adds 25 to r1, making r1 = 25**

* NOP (no operation) is not available in SIA, but can be emulated with addimmediate:  
   **addimmediate r0 0 //adds 0 to R0, doing nothing**
* Branch if less than and branch if equal are provided.
* Branch if greater than can be performed by swapping the operands of branch if less.
* Branch if not equal can be performed with branch if less than and branch if greater than pointing to the same address.
* Likewise, branch if less than or equal to can be performed with a branch if less than and a branch if equal pointing to the same address.

***IMPORTANT NOTE:***

***Since all instructions are on a 16-bit boundary, the lowest bit of addresses is not stored, since it will always be 0.***

## Instructions

Note: A **program counter (PC)** is a register in a computer processor that contains the address (location) of the instruction being executed at the current time. As each instruction gets fetched, the **program counter** increases its stored value by 1.

### add (opcode 1)

Adds the values of 2 registers and places the answer in a third register.

*Instruction format: 3R*

**add r1 r2 r3** //r3 🡸 r1 + r2

### addimmediate (opcode 9)

Adds a **signed** 8 bit value to a register

*Instruction format: ai*

**addimmediate r1 -127** //adds -127 to r1

### and (opcode 2)

Preforms a bitwise ‘and’ on 2 registers and stores the result in a third register

*Instruction format: 3R*

**and r1 r2 r3** //r3 🡸 r1 & r2

### branchifequal (opcode 10)

Compares two registers – if they hold the same value, jump to an offset from the current program counter (PC).

The offset can be between -524,286 and 524,286.

The offset is the number of **words (2 bytes)** forward or back.

PC <= PC + (2 \* offset)

*Instruction format: br*

**branchifequal r1 r2 1000** //jumps to current PC + (2 \* 1000)

### branchIfless (opcode 11)

Compares two registers – if the first register is less than the second, jump to an offset from the current program counter (PC).

The offset can be between -524,286 and 524,286.

The offset is the number of **words (2 bytes)** forward or back.

PC <= PC + (2 \* offset).

*Instruction format: br*

**branchifless r1 r2 1000** //jumps to current PC + (2 \* 1000)

### call (opcode 13)

Calls a “function” – pushes the PC (program counter) of the next instruction onto the stack (R15), then jumps to the address specified by this instruction times 2 (0 – 1,073,741,824).

*Instruction format: jmp*

**call 444** //R15 = PC of next instruction and PC = 444 \* 2

### divide (opcode 3)

Divides the value of the first register by the second and places the answer in a third register.

This is integer math with the fractional portion discarded.

*Instruction format: 3R*

**divide r1 r2 r3** // r3 🡸 r1 / r2

### halt (opcode 0)

Stops the CPU.

*Instruction format: 3R* (the register values don’t matter)

**halt**

### interrupt (opcode 8)

Interrupts the CPU using a particular interrupt number.

This could be used to jump between kernel mode and user mode or to support devices. For the virtual machine, two interrupts are supported: 0 (print registers) and 1 (print out memory)

*Instruction format: int*

i**nterrupt 1** //print all registers

### jump (opcode 12)

Jumps to the location specified in the instruction times 2 (0 – 1,073,741,824)

*Instruction format: jmp*

**jump 1000** //PC = 1000 \* 2

### load (opcode 14)

Loads a register from the memory pointed to by another register plus 2 times the offset (0 to 30).

Note that both the address in the register and the offset are in words (memory locations).

*Instruction format: ls*

**load r1 r2 10** //loads r1 with the value pointed to by r2 plus 20 bytes

### multiply (opcode 4)

Multiplies the value of the first register times the second and places the answer in a third register.

*Instruction format: 3R*

**multiply r1 r2 r3** //r3 🡸 r1 \* r2

### or (opcode 6)

Performs a bitwise OR on 2 registers and stores the result in a third register

*Instruction format: 3R*

**or r1 r2 r3** //r3 🡸 r1 | r2

### pop (opcode 7)

Copies data from stack pointer through stack pointer + 3 to specified register.

Remember R15 = bottom most byte

Adds four to the stack pointer.

*Instruction format: stack*

**pop R1** //value at R15 is copied into R1, R15 = R15 plus 4

### push (opcode 7)

Subtracts four from the stack pointer. Remember R15 = bottom most byte address

Takes the value in the specified register and stores it in the memory address indicated by the stack pointer.

*Instruction format: stack*

**push R1** //R15 = R15 minus 4, R1 value is stored in R15

### return (opcode 7)

Pops the top value from the stack and jumps to that address

*Instruction format: stack*

**return**

### store (opcode 15)

Stores a register’s value into memory memory pointed to by another register plus 2 times the offset (0 to 30).

Note that both the address in the register and the offset are in words (memory locations).

*Instruction format: ls*

**store r1 r2 10** //stores r1’s value into the memory pointed to by r2 plus 20 bytes

### subtract (opcode 5)

Subtracts the value of the second register from the first and places the answer in a third register.

*Instruction format: 3R*

**subtract r1 r2 r3** //r3 🡸 r1 - r2

## Instruction Formats

### 3R

|  |  |  |  |
| --- | --- | --- | --- |
| 4 bits | 4 bits | 4 bits | 4 bits |
| OPCODE | register 1 | register 2 | destination (register 3) |

### ai

|  |  |  |
| --- | --- | --- |
| 4 bits | 4 bits | 8 bits |
| OPCODE | register 1 | immediate value (signed) |

### br

|  |  |  |  |
| --- | --- | --- | --- |
| 4 bits | 4 bits | 4 bits | 4 bits |
| OPCODE | register 1 | register 2 | top 4 bits of address offset |

|  |
| --- |
| 16 bits |
| 16 bits of address offset |

### int

|  |  |
| --- | --- |
| 4 bits | 12 bits |
| OPCODE | interrupt |

### jmp

|  |  |
| --- | --- |
| 4 bits | 12 bits |
| OPCODE | top 12 bits of jump address |

|  |
| --- |
| 16 bits |
| lower 16 bits of jump address |

### ls

|  |  |  |  |
| --- | --- | --- | --- |
| 4 bits | 4 bits | 4 bits | 4 bits |
| OPCODE | register to load/store | address register | address offset |

### stack

|  |  |  |  |
| --- | --- | --- | --- |
| 4 bits | 4 bits | 2 bits | 6 bits |
| OPCODE | register | 00 = return, 01 = push, 10 = pop | unused |