SUNYAT

SUNYAT v.0x0 Programming Guide

revised Nov. 29, 2008

Overview

The SUNYAT v0x0 (pronounced “SUE-knee-at hex zero”) virtual machine (VM) simulates a simple computer with a character terminal for input and output. Everything about its design targets understandability rather than speed or performance. In this way it is appropriate for educational use when examining the basics of computer organization and assembly/machine language programming. The VM is also paired with a working assembler/linker application which provides an expressive programming syntax and promotes a comprehension of how the target binary is produced without requiring it. Further, the VM, assembler/linker, and example programs are distributed under the Educational Community License, Version 2.0, and may therefore be used, modified, and distributed relatively freely under its terms.

Architecture

The SUNYAT is a fully 8 bit architecture (all registers and buses are 8 bits wide.) There are 12 system registers (see Figure 1):

- Eight general purpose registers (R0—R7) are initialized to '0', '7', '2', '8', '2', '0', '0', '7' on system launch (the primary author's wedding date). Each of these registers are directly manipulable by user programs.
- The Program Counter (PC) contains the memory address of the next instruction to execute. This register can only be indirectly manipulated via the JMP, CALL, and RET instructions. PC is initially set to execute code at address 0x00.
- The high and low Instruction Registers (IRH and IRL respectively) hold the currently executing instruction. These are not accessible to user programs.
- The Stack Pointer (SP) contains the memory address of the top element in the system stack. SP initially points just beyond RAM indicating an empty stack.

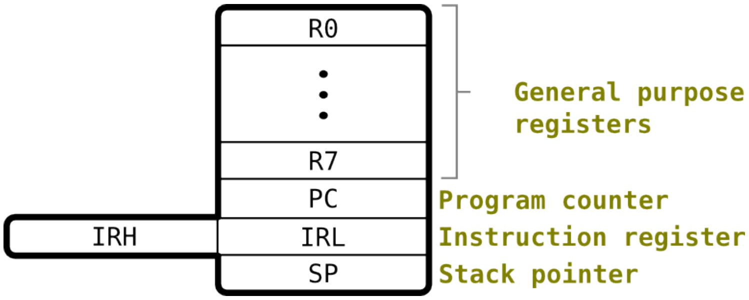


Figure 1: SUNYAT system registers

Considering the 8 bit nature of the SUNYAT, there are 2^8 (256) addressable byte-sized words of memory (see Figure 2). The first 254 (addresses 0x00—0xFD) are RAM. 0xFE and 0xFF are the terminal input and output addresses, respectively. Writing to the terminal output address using the STOR or STORP instructions will cause the terminal to display the ASCII character associated with the byte written... this includes non-printable characters such as carriage returns and line feeds. Reading from the terminal input address using the LOAD or LOADP instructions will read the next character from the host computer's standard input. Since the standard input is buffered, a read from the terminal input may cause the SUNYAT application to wait until the buffer is ready... usually when the user hits the “Enter” key. If the buffer already contains characters, the read will complete without further interaction from the user.

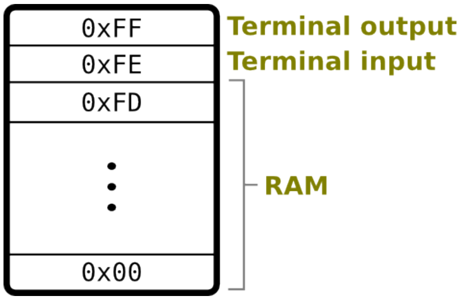


Figure 2: SUNYAT memory map

An interesting element of the SUNYAT design is its clock cycle. For the sake of simplicity (both in the implementation of the VM and user program analysis) every cycle of the VM conducts a complete fetch-decode-execute cycle, regardless of the complexity of an instruction, operands, or addressing mode. This results in a reliable one cycle time for all instructions. This includes terminal reads which might require user interaction... still only one clock cycle to the VM. This does not reflect the timing of real CPUs (particularly I/O communications) but it does provide a simple basis on which the beginning assembly programmer might predict the complete runtime of an application given known input.

Instructions

The following pages detail the SNUYAT's 32 assembly instructions and encodings. For each of these instruction definitions the following symbols are used for the sake of brevity:

Symbol	*Meaning*
opcode	The 5 bit machine code equivalent for the instruction. Each instruction description will detail this bit pattern
reg_A *or* **reg_B**	Identifies one of the eight general purpose registers (R0—R7). In the encoding, this is an unsigned 3 bit number (e.g., R6 = 110_2)
immediate	Signed 8 bit number.
address	Unsigned 8 bit memory address

MOV - register to register

General usage: `MOV reg_A reg_B`

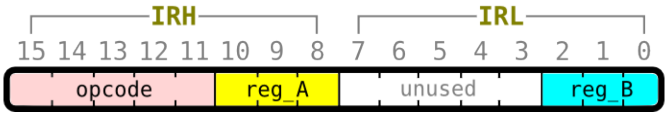
Description: Copies the value in reg_B into reg_A

Example usage: `MOV R3 R1`

Affected flags: none

Opcode: 00000_2

Encoding:



MOV – immediate to register

General usage: `MOV reg_A immediate`

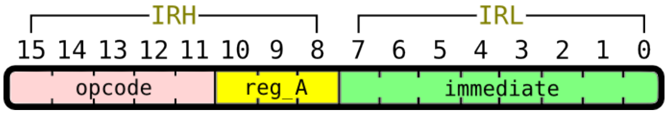
Description: Loads the immediate value into reg_A

Example usage: `MOV R4 -15`

Affected flags: none

Opcode: 00001_2

Encoding:



ADD – register to register

General usage: `MOV reg_A reg_B`

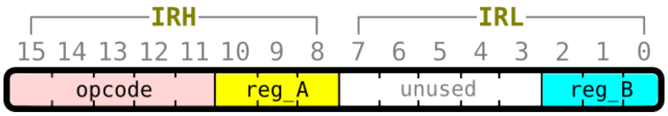
Description: Adds reg_B to reg_A, storing the result in reg_A

Example usage: **`ADD R2 R7`**

Affected flags: Zero and Sign

Opcode: 00010_2

Encoding:



ADD – immediate to register

General usage: **`ADD reg_A immediate`**

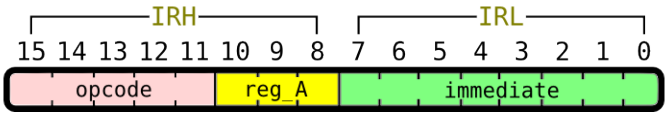
Description: Adds immediate to reg_A, storing the result in reg_A

Example usage: **`ADD R6 3`**

Affected flags: Zero and Sign

Opcode: 00011_2

Encoding:



SUB – register to register

General usage: **`SUB reg_A reg_B`**

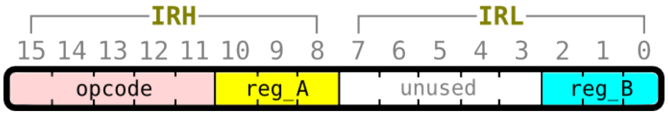
Description: Subtracts reg_B from reg_A, storing the result in reg_A

Example usage: **`SUB R1 R0`**

Affected flags: Zero and Sign

Opcode: 00100_2

Encoding:



SUB – immediate to register

General usage: **`SUB reg_A immediate`**

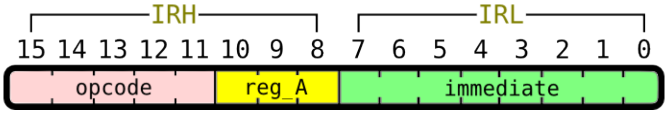
Description: Subtracts immediate from reg_A, storing the result in reg_A

Example usage: **`SUB R5 64`**

Affected flags: Zero and Sign

Opcode: 00101_2

Encoding:



MUL – register to register

General usage: **`MUL reg_A reg_B`**

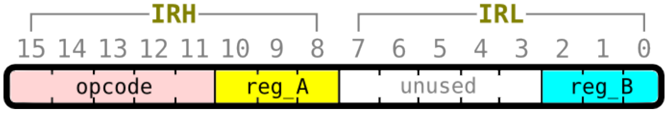
Description: Multiply reg_B and reg_A, storing the result in reg_A

Example usage: **`MUL R2 R7`**

Affected flags: Zero and Sign

Opcode: 00110_2

Encoding:



MUL – immediate to register

General usage: **`MUL reg_A immediate`**

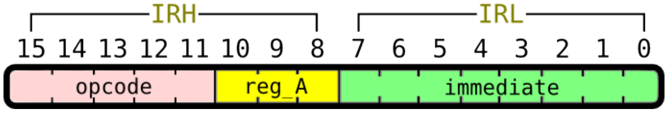
Description: Multiply immediate and reg_A, storing the result in reg_A

Example usage: **`MUL R5 -4`**

Affected flags: Zero and Sign

Opcode: 00111_2

Encoding:



DIV – register to register

General usage: **`DIV reg_A reg_B`**

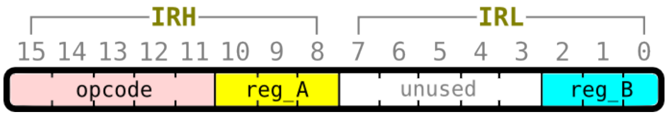
Description: Divides reg_A by reg_B, storing the result in reg_A

Example usage: **`DIV   R6   R4`**

Affected flags: Zero and Sign

Opcode: 01000_2

Encoding:



DIV – immediate to register

General usage: **`DIV   reg_A   immediate`**

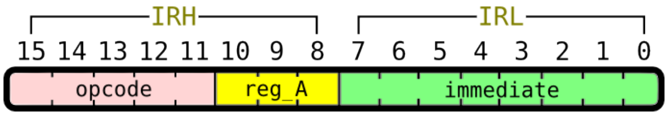
Description: Divides reg_A by immediate, storing the result in reg_A

Example usage: **`DIV   R2   5`**

Affected flags: Zero and Sign

Opcode: 01001_2

Encoding:



CMP – register to register

General usage: **`CMP   reg_A   reg_B`**

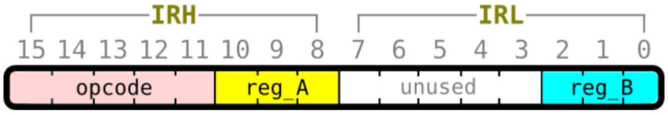
Description: Compares the two register values via subtraction but does not store the result. However, the flags are set based on the result of the subtraction.

Example usage: **`CMP   R3   R7`**

Affected flags: Zero and Sign

Opcode: 01010_2

Encoding:



CMP – immediate to register

General usage: **`CMP reg_A immediate`**

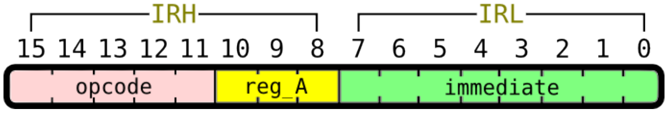
Description: Compares the register value and immediate via subtraction but does not store the result. However, the flags are set based on the result of the subtraction.

Example usage: **`CMP R0 2`**

Affected flags: Zero and Sign

Opcode: 01011_2

Encoding:



JMP

General usage: **`JMP address`**

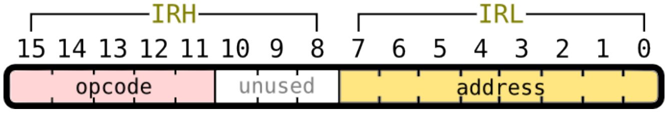
Description: Jump (branch) unconditionally to the code beginning at address. Sets the PC to address. The address will typically be provided as a label, but can be written as an immediate, as well.

Example usage: **`JMP !finished`**

Affected flags: none

Opcode: 01100_2

Encoding:



JEQ

General usage: **`JEQ address`**

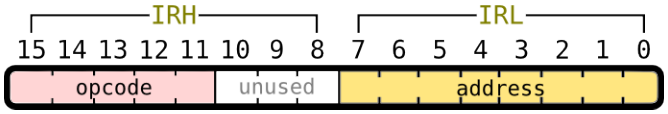
Description: Jump (branch) to the code beginning at address if the previous CMP found an equality or if an ALU instruction's result was zero... in either case the Zero flag would be high. Sets the PC to address. The address will typically be provided as a label, but can be written as an immediate, as well.

Example usage: **`JEQ !find_offset`**

Affected flags: none

Opcode: 01101_2

Encoding:



JNE

General usage: **`JNE   address`**

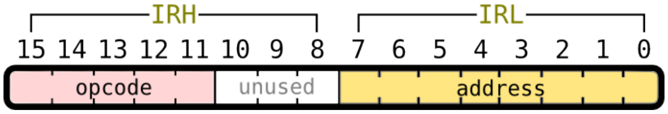
Description: Jump (branch) to the code beginning at address if the previous CMP found an inequality or if an ALU instruction's result was not zero... in either case the Zero flag would be low. Sets the PC to address. The address will typically be provided as a label, but can be written as an immediate, as well.

Example usage: **`JNE   !distribute`**

Affected flags: none

Opcode: 01110_2

Encoding:



JGR

General usage: **`JGR   address`**

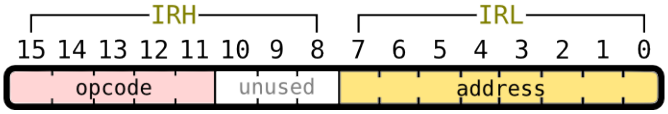
Description: Jump (branch) to the code beginning at address if the previous CMP found the left operand to be greater than the right or if an ALU instruction's result was positive but not zero... in either case the Zero flag would be low and the Sign flag low. Sets the PC to address. The address will typically be provided as a label, but can be written as an immediate, as well.

Example usage: **`JGR   !no_trigger_found`**

Affected flags: none

Opcode: 01111_2

Encoding:



JLS

General usage: **`JLS   address`**

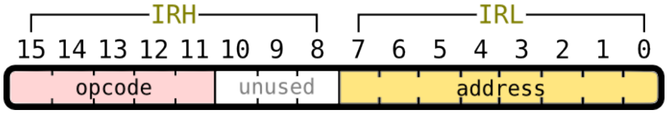
Description: Jump (branch) to the code beginning at address if the previous CMP found the left operand to be less than the right or if an ALU instruction's result was negative... in either case the Sign flag would be high. Sets the PC to address. The address will typically be provided as a label, but can be written as an immediate, as well.

Example usage: **`JLS   !negate`**

Affected flags: none

Opcode: 10000_2

Encoding:



CALL

General usage: **`CALL   address`**

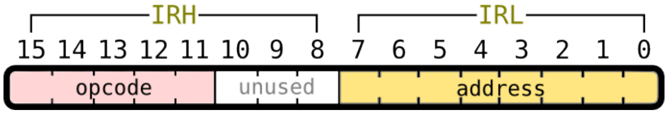
Description: Call function beginning at address. This pushes the address after the CALLing line of code to the system stack, and then sets the PC to address. The address will typically be provided as a label, but can be written as an immediate, as well.

Example usage: **`CALL   !is_even`**

Affected flags: none

Opcode: 10001_2

Encoding:



RET

General usage: **`RET`**

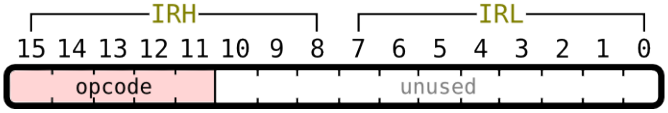
Description: Returns from a function call. This pops the top of the system stack into the PC... presuming this was the address pushed to the stack by a previous CALL. RETurning when the stack is empty is the signal to halt the VM and print the total number of clock cycles executed by the application.

Example usage: **RET**

Affected flags: none

Opcode: 10010_2

Encoding:



AND – register to register

General usage: **AND reg_A reg_B**

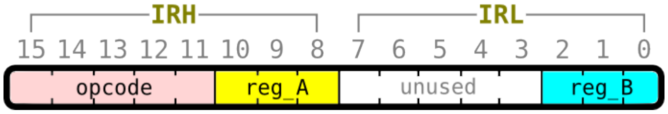
Description: Perform a bitwise AND on reg_A and reg_B, storing the result in reg_A

Example usage: **AND R0 R4**

Affected flags: Zero and Sign

Opcode: 10011_2

Encoding:



AND – immediate to register

General usage: **AND reg_A immediate**

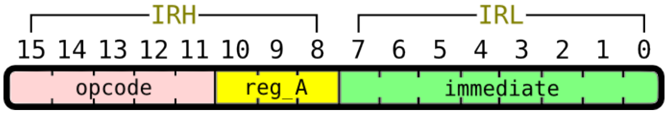
Description: Perform a bitwise AND on reg_A and immediate, storing the result in reg_A

Example usage: **AND R1 0b_0010_1001**

Affected flags: Zero and Sign

Opcode: 10100_2

Encoding:



OR – register to register

General usage: **OR reg_A reg_B**

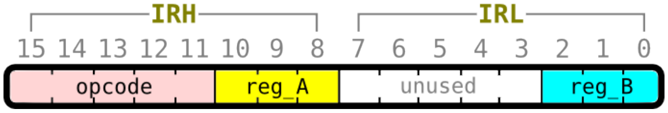
Description: Perform a bitwise OR on reg_A and reg_B, storing the result in reg_A

Example usage: **`OR R4 R5`**

Affected flags: Zero and Sign

Opcode: 10101_2

Encoding:



OR – immediate to register

General usage: **`OR reg_A immediate`**

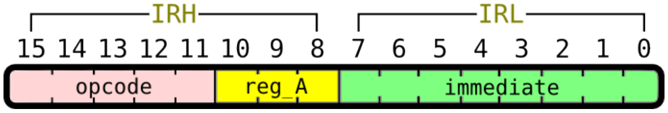
Description: Perform a bitwise OR on reg_A and immediate, storing the result in reg_A

Example usage: **`OR R6 0b_0110_0011`**

Affected flags: Zero and Sign

Opcode: 10110_2

Encoding:



XOR – register to register

General usage: **`XOR reg_A reg_B`**

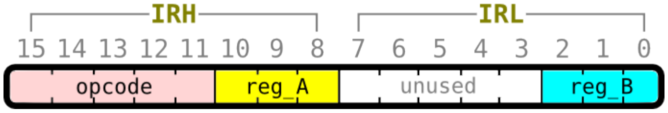
Description: Perform a bitwise EXCLUSIVE-OR on reg_A and reg_B, storing the result in reg_A

Example usage: **`XOR R2 R3`**

Affected flags: Zero and Sign

Opcode: 10111_2

Encoding:



XOR – immediate to register

General usage: **`XOR   reg_A   immediate`**

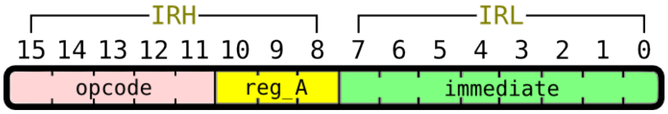
Description: Perform a bitwise EXCLUSIVE-OR on reg_A and immediate, storing the result in reg_A

Example usage: **`XOR   R0   0b_0100_0100`**

Affected flags: Zero and Sign

Opcode: 11000_2

Encoding:



NEG

General usage: **`NEG   reg_A`**

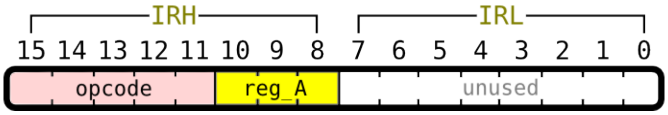
Description: Perform two's complement negation on reg_A, storing the result in reg_A

Example usage: **`NEG   R3`**

Affected flags: Zero and Sign

Opcode: 11001_2

Encoding:



LOAD

General usage: **`LOAD   reg_A   address`**

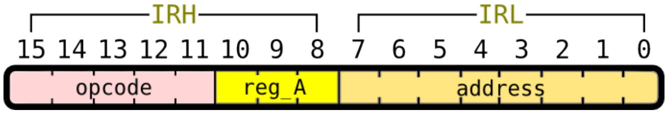
Description: Loads (copies) a value from the given memory address into reg_A.

Example usage: **`LOAD   R7   width`**

Affected flags: none

Opcode: 11010_2

Encoding:



LOADP

General usage: **LOAD reg_A reg_B**

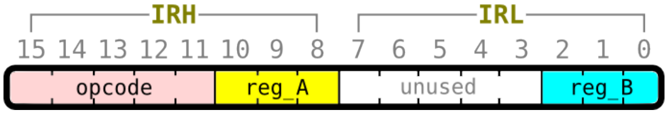
Description: Loads (copies) a value from the memory address in reg_B into a reg_A.

Example usage: **LOAD R3 R5**

Affected flags: none

Opcode: 11011_2

Encoding:



STOR

General usage: **STOR address reg_A**

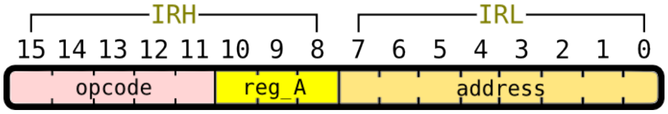
Description: Stores (copies) the value from reg_A to the given memory address.

Example usage: **LOAD count reg_A**

Affected flags: none

Opcode: 11100_2

Encoding:



STORP

General usage: **STORP reg_A reg_B**

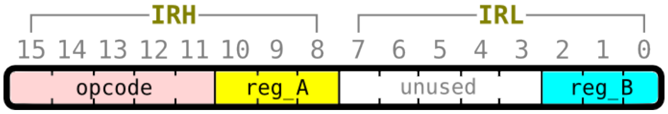
Description: Stores (copies) a value from reg_B into the memory address in reg_A.

Example usage: **STORP R1 R4**

Affected flags: none

Opcode: 11101$_2$

Encoding:



PUSH

General usage: **`PUSH reg_A`**

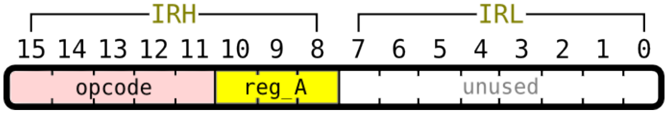
Description: Pushes (copies) the value in reg_A to the top of the system stack. This is accomplished by first decrementing SP and then storing the at the new address in SP.

Example usage: **`PUSH R3`**

Affected flags: none

Opcode: 11110$_2$

Encoding:



POP

General usage: **`POP reg_A`**

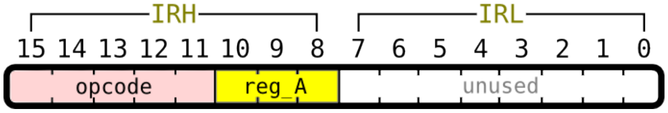
Description: Pops (copies) the value at the top of the system stack into reg_A. This is accomplished by first copying the value at the address in SP and then incrementing SP.

Example usage: **`POP R0`**

Affected flags: none

Opcode: 11111$_2$

Encoding:


IRH
IRL
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
opcode
reg_A
unused