Stupid-RISC Instruction Set

Version: 0.0.5

Table of Contents

General

INSTRUCTION ENCODING

Any instruction might have up to two arguments. The instruction code itself is 1 byte long. Each one of the arguments also has a 1 byte size. Though some instructions like jmp or halt might only require 0 or one argument/s, each of those is still 3 bytes long.

CALLING RULES

For system instructions, both lowercase & uppercase can be used. For self-created functions, the capitalisation matters.

REGISTERS

Any number in the assembly code can't be outside the range of 0…2^7-1. The encoding is as following:

Register:

Register?	0 bit	0 bit	0 bit	Reg select	Reg select	Reg select	Reg select
1	0	0	0	0	0	1	0

This selects register 2. A register in assembly is represented with R_n where n is between 0…11. Special registers are called by there names.

UInt7 Selection

Register?	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	0	0	0	0	0	1	0

This outputs the number 2. To select negative numbers or other numbers outside this range, it'll need to be loaded from the data section or be created with arithmetic. A number in assembly is represented with N_n. Where n is within 0…127 in decimal & N is the number system.

Number Systems

Overview

Number systems are marked with a starting character like x or no starting character if decimal. If uppercase versions of the number-system-selecting-character exist, those can be uppercase and lowercase.

All numbers will be converted to decimal during the assembly process.

Limits

As the most-significant-bit is used for register defintions, the limit upper limit is 2^7-1 which is 127 (decimal) for numbers in the text section. Negative numbers can't be represented in the text section either, since only the least significant 7 bits can be set.

Define numbers out of bounds in the data section.

Decimal

Decimal requires no prefix. In dissassembly, „N“ or „n“ might be used to represent any number.

Registers

Registers have a minimum value of 128, and are usually called with the r/R prefix, which will add 128 automatically. The value after the r is decimal.

ASCII Characters

ASCII characters are supported with the prefix ’ followed by the character and the same suffix. Only the first character in this sequence will have an effect. Whitespaces are currently not supported in the encoding process, so x20 or ‘_‘ might be used instead.

Other Number Systems

System Prefix	System
x	Hexadecimal (Base 16)
o	Octal (base 8)
b	Binary (base 2)

Functions and constants

Constant defintions always start with a ., which is not a part of the constant name. The dot is followed by the name of the constant. There can't be any space between the dot and the name. The name is followed by the value seperated by an in line whitespace character.

Here’s an example constant definition:

.some_constant x0f

Later this might be used like:

add R0 some_constant

Functions are good for jumping somewhere in the code. They are defined by the functions name and a : afterwards. The function name will then point to the start of the codeblock afterwards. Functions usually have an indentation for the function blocks.

Example:

```
loop:
        jmp loop
```

This loops forever.

Comments

Comments escape everything afterwards within the same line. They are defined with a hashtag (#) like in python.

Examples:

```
# This is a very interesting comment

inc R6 # increments i
```

Start function

The start function is at the very top of the code and will usually be executed first.

ALU

The ALU is devided into three parts:

- Integer ALU (addi, etc) (having the ALU code 00)
- General Logic ALU (not, etc) (having the ALU code 01)
- Floating Point & Double ALU (addf, etc) (having the ALU code 10)

On the CPU level, arguments have 32 because they've already passed throw the register selectors. In assembly, this are 8 bits per register (1 means register, selector follows afterwards).

An example might be: *add R1, R2* which would store R1 + R2 in R1. Read the registers documentation for more information on this topic.

INTEGER ALU PART INSTRUCTION SELECTORS:

Operation	ALU Part instr. select. bit code	Description	Impl. Yet	Sim Impl. y.
add	01000000	Adds two numbers and stores them in the selected output register.	0	1
mul	01000010	Multiplies two numbers and stores them in the selected output register.	0	1
sub	01000001	Subtractions the second number from the first one and stores the ooutput in the selected output register.	0	1
div	01000011	Same as above with division.	0	1
mod	01000100	Same as above with modulo.	0	1

The integer ALU part instruction selector is very simple, the other ones are more complex though.

GENERAL LOGIC ALU PART INSTRUCTION SELECTORS:

Operation	ALU Part instr. select. Bit code	Description	Impl. Yet	Sim Impl. y.
or	Not Defined	Performs a logical or.	0	0
and	Not Defined	Performs a logical and.	0	0
xor	Not Defined	Performs a logical exclusive or.	0	0

FLOATING POINT AND DOUBLE ALU PART INSTRUCTION SELECTORS:

This entire part is subject to change and will be reformed soon.

Operation	ALU Part instr. select. Bit code	Description	Impl. Yet
Tfl	Not Defined	To float (converts an integer into a 64 bit floating point number). Subject to change.	0
Tint	Not Defined	To int (converts a float into an integer). Subject to change.	0
Tintp	Not Defined	To int with base (will put the 2^0 bit at bit 32 in the integer with an offset of the given base, which is required to add, etc, as the result can be converted back later). Subject to change.	0
Exp	Not Defined	Gets the exponent of the float/double. Subject to change.	0
Sep	Not Defined	Set exponent (modifies the exponent). Subject to change.	0
num	Not Defined	Gets the value excluding the exponent. Subject to change.	0
snum	Not Defined	Sets the value excluding the exponent. Subject to change.	0
intfltfl	Not Defined	Int float to float converts an integer float from above back to a normal float where argument 0 is the int, argument 1 is the original offset and argument 2 is the destination. Subject to change.	0

Math operations can't be done in the floating point / double state. The numbers would need to be converted for this. Example code adding two floats which are in R0 and R1 & storing the result in R0:

exp R0 R2;; Store R0's exponent in R2

tintp R0 R2 R0;; Store the value of R0 in R0 but as an integer as shown above.

tintp R1 R2 R1;; Store the value of R1 in R1 using an offset making it directly addable as shown above.

addi R0 R1 R0;; Perform the addition and store in R0

intfltfl R0 R2 R0;; Convert back to float using the original offset and store in R0;; done.

Assembly Sections

Introduction

There are two sections: The data and the text (code) section. The code section contains all the commands that should be executed, whilst the data section can define things like arrays or larger and/or mutable vaues.

Code Section

The code / text section, which is defined with <text> holds all commands and some constants. Things like functions can be defined and called here.

Data Section

The data section stores long strings of data directly after the code. The data sections data is moved to the bottom of the binary automatically. A defintion is started with the name, followed by the datatype (8b, 16b, 32b), and finished of by the data (which might be multiple). Right now, only 8b is available.

For example:

```
text 8b “Hello World“
arr 8b 15 20 24
```

The name then refers to the position in memory.

Memory operations

There are heap and stack operations. The heap grows upwards and the stack downwards. In any Stupid RISC system, there must be at least 1kB of RAM.

Units

Shortend Version	Full Name	Amount of Bits	Impl. In vm	Impl. In CPU
b	byte	8	Y	N
w	word	16	N	N
d	Double word	32	N	N
q	QWORD	64	N	N

Heap Operations

There are load („ld<unit>“) and store („st<unit>“) instructions. Load loads a <unit> at the address of the second input parameter into the register given by the first one while store byte does the opposite (storing the <unit> determined by the first parameter at the value of the second parameter).

Stack Operations

Stack operations are push („push<unit>“) and pop („pop<unit>). Push will add the argument to the stack and increment the stack pointer, whilst pop will remove the last element and store it in the first argument (register).

Assembler Calculations

The assembler is able to calculate some values in the assembly process. This might be adding two constants (or numbers) or other simple math operations.

Notation

The numbers/constants to perform the operation on must be in rectangular parantacies and split by spaces and the operator:

loadb R0 [text + offset]

Standard output

The standard output is a type of display that can only present ASCII characters and is directly in contact with the CPU.

Write

You can write to the standard output (one byte at a time) using the sow instruction. The first argument will be outputed in ASCII.

Clear

The standard output can be cleared using the soc instruction.

Jump instruction

There are conditional and unconditional jump instructions. Conditional jump instructions may require a parameter to be a certain value while an unconditional jump instruction always jumps to the desired location.

Unconditional Jump Instruction

The „jmp“ always goes to the position specified in the first argument.

Jump Zero Instruction

The „jmpz“ instruction only jumps to the location specified by the second argument, if the first argument is 0.