Stupid-RISC Instruction Set

Version: 0.0.2

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?	0 bit	0 bit	0 bit	Reg select	Reg select	Reg select	Reg select
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.

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

Operation	ALU Part instr. select. bit code	Description	Impl. Yet	Sim Impl. y.
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

Operation	ALU Part instr. select. Bit code	Description	Impl. Yet
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.