Processor Documentation

Ruben Saunders

August 2024

1 Principals

- This processor will operate a RISC instruction set.
- This processor has a word size of 64 bits, and supports both floats (4 bytes) and doubles (8 bytes).
- The instruction set will provide methods to load values into and out of registers. Then, most operations will be on registers.
- Load/store instructions operate on 32-bit immediates.
- Arithmetic and logic instructions operate on full registers, so 64-bit.

2 Registers

See below for a list of registers. All registers are 64-bit. Register names are preceded by a dollar '\$' sign.

Symbol	Name	Bit	Description		
	Special Registers				
\$ip	Instruction Pointer		Point to next address to execute as an instruction.		
\$sp	Stack Pointer		Top address of the stack.		
\$fp	Frame Pointer		Point to the next byte beyond the last stack frame.		
\$flag	Flag Register	8-64			
		5-7	Error flag. • 000: no error. • 001: invalid opcode, opcode in \$ret. • 010: segfault, address in \$ret. • 011: register segfault, register offset in \$ret. • 100: invalid syscall, opcode in \$ret. Execution status: 1=executing, 0=halted.		
		4	Can be used to halt the processor.		
		3	Zero flag. Indicates if register is zero. Updated on most instructions' dest register.		
		0-2	Comparison bits. • 000: not equal. • 001: equal. • 010: less than. • 011: less than or equal to. • 110: greater than. • 111: greater than or equal to.		
\$ret	Return Value Register		Contains value returned from function, syscall, etc. Contains process exit code on halt.		
\$zero	Zero		Hardwired to zero.		

General Purpose Registers				
\$r1 - \$r16	\$r1 - \$r16 GPRs Register for general use.			
\$s1 - \$s8	Preserved GPRs	Register for general use. Values are preserved in stack frame.		

3 Addressing Modes

An argument may be one of the following specifiers:

Argument	Size	Comment	Example
<reg></reg>	8	Register offset.	\$r1
<value></value>	2 + 32	Any listed addressing mode. 2 indicator bits, 32 for data.	0xdead
<addr></addr>	1 + 32	Any listed memory addressing mode. 1 indicator bit, 32 for data.	(0x8000)

The following table specifies possible addressing modes.

Indicator	Name	Syntax	Operation	Size
00	Immediate	imm	imm	32
01	Register	\$reg	Reg[\$reg]	8
10	Memory	(mem)	Mem[mem]	32
11	Register Indirect	n(\$reg)	Mem[Reg[\$reg] + n]	\$reg=8, \$n=24

4 Instruction Set

Notes:

- Instructions accept a conditional test suffix, unless indicated via a \square symbol.
- Mnemonics support overloading. That is, the same mnemonic can have many argument signatures. Optional arguments are listed using square brackets [optional] versus mandatory arguments <mandatory>.
- For all arithmetic and logical instructions with signatures <reg> <reg> <value>, the first register is optional. If omitted, the supplied register is duplicated. I.e., \$r, \$v becomes \$r, \$r, \$v.
- All arithmetic operations and the compare operation take a datatype.

Instruction	Syntax	Operation/Comments			
	Data Transfer				
		Load a word into a register.			
Load	load (row) (walno)	Reg[\$reg] = \$value			
Load	load <reg> <value></value></reg>	Note that any immediate is only 32-bit;			
		Use loadw for loading a 64-bit immediate.			
Load Upper	loody (mag) (volue)	Load a half-word (32-bit) into the upper half of a register.			
Load Upper	loadu <reg> <value></value></reg>	Reg[\$reg][32:] = \$value			
		Pseudo-instruction.			
		Pseudo-instruction. Loads a word into a register. load \$reg \$value[:32]			
Load Word	loadw <reg> <value></value></reg>	<pre>load \$reg \$value[:32]</pre>			
		loadu \$reg \$value[32:]			
		Note accepts a 64-bit immediate.			
		Pseudo-instruction.			
Zero	zero <reg></reg>	Zeroes/clears a register.			
		xor \$reg, \$reg			
Store	atoro (roa) (addr)	Copy from register to memory.			
Store	store <reg> <addr></addr></reg>	Mem[\$addr] = Reg[\$reg]			

Add value to a register. Subtract Subtract Subtract Subtract Subtract Subtract Subtract Subtract Subtract value Subtract value Subtract value Subtract value from a register. Reg[\$reg1] = Reg[\$reg2] = \$value Multiply Multiply register by a value. Reg[\$reg1] = Reg[\$reg2] = \$value Multiply register by a value, store as double. Reg[\$reg1] = Reg[\$reg2] : \$value Calculate the remainder when dividing a register by a value. The register is treated as a signed word, Reg[\$reg1] = Reg[\$reg2] in well \$value Calculate the remainder when dividing a register by a value. The register is treated as a signed word, Reg[\$reg1] = Reg[\$reg2] in well \$value Branching Compare \$\text{cmp} < \text{value} \text{Value} \text{Value} \text{Suffine} \text{Value} \text{Reg} \text{Sireg1} = \text{Reg} \text{Reg} \text{Sireg2} \text{Value} \text{Reg} \text{Sireg1} = \text{Reg} \text{Reg} \text{Sireg2} \text{Value} \text{Reg} \text{Sireg1} = \text{Reg} \text{Reg} \text{Sireg1} \text{Value} \text{Reg} \text{Sireg2} \text{Value} \text{Reg} \text{Sireg1} = \text{Reg} \text{Reg} \text{Sireg2} \text{Value} \text{Reg} \text{Sireg1} = \text{Reg} \text{Reg} \text{Value} \text{Reg} \text{Reg} \text{Sireg1} = \text{Reg} \text{Reg} \text{Value}	Arithmetic				
Subtract Multiply mul <reg> <reg> <value> Reg[\$reg1] = Reg[\$reg2] - \$value Multiply register by a value. Reg[\$reg1] = Reg[\$reg2] × \$value Division div <reg> <reg> <value> Stack Stack Stack Push Multiply mul <reg> <reg> <value> Stack Subtract Multiply register by a value. Reg[\$reg1] = Reg[\$reg2] - \$value Divide a register by a value, store as double. Reg[\$reg1] = Reg[\$reg2] - \$value Stack Stack Multiply register by a value. Multiply register by a value. Stack Stack Push Word The register streated as a signed word, the value as a signed word, the value as a signed half-word. The register is treated as a signed word, the value as a signed half-word. The register is treated as a signed word, the value as a signed half-word. The register is treated as a signed word, the value as a signed half-word. The register is treated as a signed word, the value as a signed word, the value as a signed half-word. The register is treated as a signed word, the value as a signed word, the value as a signed half-word. The register is treated as a signed word, the value as a signed word</value></reg></reg></value></reg></reg></value></reg></reg>		All arithmmetic operations, bar mod, expect a datatype.			
Subtract Subtract Sub \reg \reg \reg \reg \reg \reg \reg \reg	Add	add <reg> <reg> <value></value></reg></reg>			
Subtract Sub Fregs Values Reg[Sreg2] = Reg[Sreg2] - \$value Multiply register by a value Multiply register by a value Divide a register by a value Divide a register by a value Divide a register by a value, store as double. Reg[Sreg1] = Reg[Sreg2] + \$value Divide a register by a value, store as double. Reg[Sreg1] = Reg[Sreg2] Sequence Value The register is tracted as a signed word, the value as a signed word, the value as a signed half-word. Reg[Sreg1] = Reg[Sreg2] mod \$value Reg[Sreg1] = Reg[Sreg2]	Huu	add (log) (log) (value)			
Multiply mul <pre></pre>	Subtract	sub (rem) (rem) (walue)			
Beg(\$reg1) = Reg(\$reg2) \times value	Subtract	Sub (leg) (leg) (value)			
Division div <reg> <reg> <value <reg="" div="" division="" =""> <reg> <value \$value="" [\$reg1]="Reg" [\$reg2]="mod" a="" as="" by="" dividing="" division="" half-word.="" is="" reg="" register="" register<="" signed="" stage="" td="" the="" treated="" value="" value.="" word,="" =""><td>Multiply</td><td>mul (rog) (rog) (walue)</td><td></td></value></reg></value></reg></value></reg></value></reg></value></reg></value></reg></reg>	Multiply	mul (rog) (rog) (walue)			
Beg[\$reg1] = Reg[\$reg2] ÷ \$value	Withitply	mui (leg) (leg) (value)			
Modulo mod <reg> <reg> <red> <red *="" feeglareg2 ="" syalue="" td="" ="" <=""><td>Division</td><td>div <reg> <reg> <value></value></reg></reg></td><td></td></red></red></reg></reg>	Division	div <reg> <reg> <value></value></reg></reg>			
The register is treated as a signed word, the value as a signed half-word. Reg[\$reg1] = Reg[\$reg2] mod \$value Compare Compare Compare \$1 with \$2, setting comparison bits in flag register. E.g., set 1t if \$1 < \$2.	Division	uiv (log) (log) (value)			
## the value as a signed half-word. Reg[\$reg1] = Reg[\$reg2] mod \$value ## Branching Compare					
The value as a signed half-word. Reg[\$reg1] = Reg[\$reg2] mod \$value	Modulo	mod <reg> <reg> <ualue></ualue></reg></reg>			
Compare Compare Compare Compare Compare Compare Stack	Wiodulo	mod (log) (log) (value)			
Compare cmp <reg> <value> Compare \$1 with \$2, setting comparison bits in flag register. E.g., set 1t iff \$1 < \$2. Note Z flag is set depending on value, not register. Pseudo-instruction Branch to the given address if comparison matches conditional. load Jump</value></reg>			Reg[\$reg1] = Reg[\$reg2] mod \$value		
Compare cmp <reg> <value> E.g., set 1t iff \$1 < \$2. Note Z flag is set depending on value, not register. Pseudo-instruction Branch b < cnd ></value></reg>					
Note Z flag is set depending on value, not register. Pseudo-instruction Branch to the given address if comparison matches conditional. load <cnd> \$ip, \$value Pseudo-instruction. load \$ip \$value </cnd>			Compare \$1 with \$2, setting comparison bits in flag register.		
Branch b < cnd >	Compare	cmp <reg> <value></value></reg>	E.g., set lt iff \$1 < \$2.		
Branch b < cnd> <value> Branch to the given address if comparison matches conditional. load < cnd > \$ip, \$value Pseudo-instruction. load \$ip, \$value Pseudo-instruction. load \$ip, \$value Pseudo-instruction. load \$ip, \$value Logical Not not <reg> < reg> And and <reg> < reg> < value> Bitwise NOT a register. Reg[\$reg1] = ~ Reg[\$reg2] Bitwise AND between register and value. Reg[\$reg1] = Reg[\$reg2] & \$value Bitwise OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Bitwise exclusive OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Bitwise exclusive OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Bitwise exclusive or Reg[\$reg1] = Reg[\$reg2] \$value Logically shift the register right an amount. Reg[\$reg1] = Reg[\$reg2] \$value Logically shift the register left an amount. Reg[\$reg1] = Reg[\$reg2] \$value Stack Pseudo-instruction Push a 32-bit value onto the stack. sub \$sp, 8 loadu \$r1, <value> store \$r1, (\$sp) add \$sp, 4 Note for efficiency, this is implemented as an instruction. Push a 64-bit word onto the stack. sub \$sp, 8 loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value></value></reg></reg></value>			Note Z flag is set depending on value, not register.		
load <cnd> \$ip, \$value Jump</cnd>			Pseudo-instruction		
Jump	Branch	b <cnd> <value></value></cnd>	Branch to the given address if comparison matches conditional.		
Sump Jimp \ Value			load <cnd> \$ip, \$value</cnd>		
Not not <reg> <reg> Stack </reg></reg>	Iump 🗆	imp (malue)	Pseudo-instruction.		
Not not <reg> <reg> Bitwise NOT a register. Reg[\$reg1] = ~ Reg[\$reg2] </reg></reg>	Jump ⊔	Jmp <value></value>	load \$ip \$value		
Not not <reg> <reg> Bitwise NOT a register. Reg[\$reg1] = ~ Reg[\$reg2] </reg></reg>			Logical		
And and <reg> <reg> <value> Bitwise AND between register and value. Reg[\$reg1] = Reg[\$reg2] & \$value Reg[\$reg1] = Reg[\$reg2] & \$value Bitwise OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Exclusive Or</value></reg></reg>	NT /				
And and <reg> <reg> <value> Bitwise AND between register and value. Reg[\$reg1] = Reg[\$reg2] & \$value Bitwise OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Bitwise OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Bitwise exclusive-OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Bitwise exclusive-OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Legislary shift the register right an amount. Reg[\$reg1] = Reg[\$reg2] \$value Left Shift sh1 < reg> < reg> < value> Stack Pseudo-instruction Push a 32-bit value onto the stack. sub \$sp, 8 loadu \$r1, < value> store \$r1, (\$sp) add \$sp, 4 Note for efficiency, this is implemented as an instruction. Pseudo-instruction Push a 64-bit word onto the stack. sub \$sp, 8 loadw \$r1, < value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value></reg></reg>	Not	not <reg> <reg></reg></reg>	$Reg[\$reg1] = \sim Reg[\$reg2]$		
And and cres cres calles Or or cres cres calles Bitwise OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Bitwise OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Bitwise exclusive-OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Bitwise exclusive-OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Reg[\$reg1] = Reg[\$reg2] \$value Logically shift the register right an amount. Reg[\$reg1] = Reg[\$reg2] \$value Logically shift the register left an amount. Reg[\$reg1] = Reg[\$reg2] \$value Stack Pseudo-instruction Push a 32-bit value onto the stack. sub \$sp, 8 loadu \$r1, <value> store \$r1, (\$sp) add \$sp, 4 Note for efficiency, this is implemented as an instruction. Pseudo-instruction Push a 64-bit word onto the stack. sub \$sp, 8 loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value></value>	A 1				
Or or <reg> <value> Bitwise OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Bitwise exclusive-OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Bitwise exclusive-OR between register and value. Reg[\$reg1] = Reg[\$reg2] \$value Reg[\$reg1] = Reg[\$reg2] \$value Logically shift the register right an amount. Reg[\$reg1] = Reg[\$reg2] \$value Logically shift the register left an amount. Reg[\$reg1] = Reg[\$reg2] \$value Stack Pseudo-instruction Push a 32-bit value onto the stack. sub \$sp, 8 loadu \$r1, <value> store \$r1, (\$sp) add \$sp, 4 Note for efficiency, this is implemented as an instruction. Pseudo-instruction Push a 64-bit word onto the stack. sub \$sp, 8 loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value></value></value></reg>	And	and <reg> <reg> <value></value></reg></reg>	Reg[\$reg1] = Reg[\$reg2] & \$value		
Exclusive Or xor <reg> <reg> <value> Bitwise exclusive-OR between register and value. Reg[\$reg1] = Reg[\$reg2] \(\pm\$ \$\text{value}\$ \$\text{Reg}[\$\text{Feg2}] = \text{Reg}[\$\text{value}\$ \$\text{Reg}[\$\text{Shift}\$ \$\text{shr} < \text{reg} < \text{value}\$ \$\text{Logically shift the register left an amount.} \$\text{Reg}[\$\text{\$reg1}] = \text{Reg}[\$\text{\$reg2}] \(\pm\$ \$\text{\$value}\$ \$\text{Reg}[\$\text{\$reg2}] \(\pm\$ \$\text{\$value}\$ \$\text{\$reg\$}\$ \$\text{\$value}\$ \$\text{\$Reg}[\$\text{\$reg1}] = \text{Reg}[\$\text{\$reg2}] \(\pm\$ \$\text{\$value}\$ \$\text{\$reg\$}\$ \$\text{\$value}\$ \$\text{\$Reg}[\$\text{\$reg2}] \(\pm\$ \$\text{\$value}\$ \$\text{\$reg\$}\$ \$\text{\$value}\$ \$\text{\$reg\$}\$ \$\text{\$value}\$ \$\text{\$reg\$}\$ \$\text{\$value}\$ \$\text{\$reg\$}\$ \$\text{\$value}\$ \$\text{\$value}\$ \$\text{\$reg\$}\$ \$\text{\$value}\$ \$\$val</value></reg></reg>	0	(
Reg[\$reg1] = Reg[\$reg2] ⊕ \$value Right Shift	Or	or <reg> <reg> <value></value></reg></reg>	Reg[\$reg1] = Reg[\$reg2] \$value		
Right Shift shr <reg> <reg> <value> Logically shift the register right an amount. Reg[\$reg1] = Reg[\$reg2] >> \$value Left Shift shl <reg> <reg> <value> Logically shift the register left an amount. Reg[\$reg1] = Reg[\$reg2] >> \$value Logically shift the register left an amount. Reg[\$reg1] = Reg[\$reg2] <</value></reg></reg></value></reg></reg>	E1		Bitwise exclusive-OR between register and value.		
Right Shift	Exclusive Or	xor <reg> <reg> <value></value></reg></reg>	Reg[\$reg1] = Reg[\$reg2] \oplus \$value		
Left Shift shl <reg> <reg> <value> Exact Contain the register left an amount amount. </value></reg></reg>	Dimbt Chift	aha (man) (man) (malua)	Logically shift the register right an amount.		
Reg[\$reg1] = Reg[\$reg2] & \$value	Right Shift	snr (reg) (reg) (value)	Reg[\$reg1] = Reg[\$reg2] \gg \$value		
Stack Pseudo-instruction	I of Chif	ahl (man) (man) (malua)	Logically shift the register left an amount.		
Push a 32-bit value onto the stack. Push a 32-bit value onto the stack. Sub \$sp, 8 Loadu \$r1, <value> Store \$r1, (\$sp) add \$sp, 4 Note for efficiency, this is implemented as an instruction. Pseudo-instruction Push a 64-bit word onto the stack. Sub \$sp, 8 Loadw \$r1, <value> Store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value></value>	Leit Siiit	sni (reg) (reg) (value)	Reg[\$reg1] = Reg[\$reg2] « \$value		
Push a 32-bit value onto the stack. sub \$sp, 8 loadu \$r1, <value> store \$r1, (\$sp) add \$sp, 4 Note for efficiency, this is implemented as an instruction. Pseudo-instruction Push a 64-bit word onto the stack. sub \$sp, 8 loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value></value>	Stack				
Push push <value> Sub \$sp, 8 loadu \$r1, <value> store \$r1, (\$sp) add \$sp, 4 Note for efficiency, this is implemented as an instruction. Pseudo-instruction Push Word Push Word Push word onto the stack. sub \$sp, 8 loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value></value></value>			Pseudo-instruction		
Push push <value> Sub \$sp, 8 loadu \$r1, <value> store \$r1, (\$sp) add \$sp, 4 Note for efficiency, this is implemented as an instruction. Pseudo-instruction Push Word Push Word Push word onto the stack. sub \$sp, 8 loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value></value></value>			Push a 32-bit value onto the stack.		
Push push <value> store \$r1, (\$sp) add \$sp, 4 Note for efficiency, this is implemented as an instruction. Pseudo-instruction Push a 64-bit word onto the stack. sub \$sp, 8 loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value></value>					
store \$r1, (\$sp) add \$sp, 4 Note for efficiency, this is implemented as an instruction. Pseudo-instruction Push a 64-bit word onto the stack. sub \$sp, 8 loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value>	Push	push <value></value>	<u> </u>		
add \$sp, 4 Note for efficiency, this is implemented as an instruction. Pseudo-instruction Push a 64-bit word onto the stack. sub \$sp, 8 loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value>		•			
Push Word Push Word Push Word Push Word Push word pushw <value> Push & 64-bit word onto the stack. sub \$sp, 8 loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value></value>			_		
Push Word Push Word Push w value> pushw <value> pushw <value> pushw store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value></value>					
Push Word pushw <value> sub \$sp, 8 loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value></value>					
loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value>			Push a 64-bit word onto the stack.		
loadw \$r1, <value> store \$r1, (\$sp) The pop operation is not implemented due to its simplistic nature.</value>	Push Word	pushw <value></value>	sub \$sp, 8		
The pop operation is not implemented due to its simplistic nature.					
			store \$r1, (\$sp)		
T - 4 f +1 + 1		The pop operation is not imp	plemented due to its simplistic nature.		
		I.e., to pop a word from the	stack:		
Pop sub \$sp, 8	Pop	_			
And to store it in a register:		_			
load \$r1, (\$sp)		load \$r1, (\$sp)			

	Functions			
Function Call	call <addr></addr>	Call procedure at location addr.		
Tunction Can	Call (addi)	More complex than load ip, \$addr as pushes stack frame.		
Return	ret	Return from function call.		
netum	let	Restores key registers (undoes call).		
System Call	avagell (velve)	Invoke the system call mapped to the given value.		
System Call	syscall <value></value>	See the respective section for mappings.		
		Miscellaneous		
		Useless operation; do nothing.		
No-Operation \square	nop	Equivalent to or r1, 0.		
		Note For efficiency, implemented as instruction.		
		Pseudo-instruction		
Exit	exit [value]	Exit the program, optionally with an exit code in \$ret.		
	exit [Agine]	If code provided: load \$ret, <value></value>		
		syscall <opcode: exit=""></opcode:>		

4.1 Pseudo-Instructions

These are instructions which are not necessary for full functionality, but are provided for usefulness. They may be implemented using other instructions. It is up to the implementer whether to implement these as actual instructions or expand them to their equivalent form.

4.2 Instruction Layout

All instructions are encoded in a single 64-bit word. The layouts of various types is listed below. The size field stated the size in bits of this field. From top-to-bottom, the table starts at the least-significant bit.

Note, the opcode of each instruction is not decided upon; it may be any value as long as the instruction set is implemented. The only exception is nop, which maps to a fully-zeroed word.

Generic Layout This outlines the generic structure of an instruction. The first section of the table refers to the 'header'.

Bit	Purpose	Comments	
0-5	Opcode		
6-9	Conditional test	These bits are tested against \$flag to determine if instruction is executed or skipped. 1111: skip test. 1001: test if zero flag is set. 1000: test if zero flag is unset. Otherwise: match lower 3 bits to \$flag.	
10-64	Instruction dependant.		

Conditional Test Most instructions expect a conditional test field. Below shows the mapping between suffix and bit field.

Suffix	Bits	Operator	Comments
N/A	1111	N/A	Skip test.
ne / neq	0000	<i>≠</i>	Test if not equal.
eq	0001	=	Test if equal.
lt	0010	<	Test if less than.
le / lte	0011	\leq	Test if less than or equal to.
gt	0110	>	Test if greater than.
ge / gte	0111	\geq	Test if greater than or equal to.
Z	1001	= 0	Test if zero flag is set.
nz	1000	$\neq 0$	Test if zero flag is clear.

Data-Type Indicator Some instructions have a field to specify the data-type of the data being operated on. These bits are after the ordinary header, and are as follows:

Bit 0	Bit 1	Bit 0	Suffix	Comments
Decimal?	Signed?	Full or half word?	Sullix	Comments
0	0	0	hu	32-bit unsigned integer.
0	0	1	[u]	64-bit unsigned integer.
0	1	0	hi	32-bit signed integer.
0	1	1	i	64-bit signed integer.
1	0	0	f	32-bit float.
1	0	1	d	64-bit double.

Datatypes may be interpreted slightly differently, depending on the instruction.

• Arithmetic operations: the datatype refers to the type of the first data to be operated on. The last argument is always considered a 32-bit signed integer or float. That is, in add.u \$r1, -75, \$r1 is assumed to hold an unsigned 64-bit integer, but -75 is a 32-bit signed integer, while the result also be an unsigned 64-bit integer.

5 Calling Convention

Despite being a RISC processor, this processor will support explicit call and ret functions which will aid in pushing and popping a stack frame. For ease of programming, multiple actions are taken in each to maintain structure, so they are not pseudo-instructions.

5.1 Function Invocation

To call a function [at] func with n arguments:

```
\begin{array}{ll} \text{push <arg1>}\\ \dots\\ \text{push <argn>}\\ \text{push } n\times 4\\ \text{call <func>} \end{array}
```

Note when zero arguments are needed, still push 0 to indicate this.

Stack				
Before	After			
	preserved GP registers	← \$sp		
	old ip			
	old fp	$\leftarrow \$ \mathrm{fp}$		
	n bytes			
	args			
$xxx \leftarrow \$sp$	XXX			

5.2 Function Returning

To return from the function invoked in the previous sub-section, we need only a call to ret. This will restore and pop the stack frame, as well as handle any arguments the user pushed. The following operations take place:

```
Reg[$ip] = old ip
Reg[$fp] = old fp
Reg[$sp] = loc(xxx)
```

5.3 Argument Retrieval

The frame points to the top of the previous frame. Using the diagram above, it is possible to retrieve an argument from the stack. It is important to note that the size of the additional information pushed via the processor may theoretically vary, and so referencing and relying on knowledge of this size is unadvised.

$$i$$
: argument index, 0-indexed; n : number of arguments. Arg i = Reg[\$fp] - 4 * (1 + n - i)

6 System Call

System calls are core functionality abstracted inside the processor. Actions are assigned operation codes and invoked via syscall <opcode>. Optionally, each read arguments from general-purpose registers r1 onward.

Service	Opcode	Arguments	Operation	Result		
	Output					
print_int	1	r1 = integer	Print 64-bit integer.	None		
print_float	2	r1 = float	Print 32-bit float.	None		
print_double	3	r1 = double	Print 64-bit double.	None		
print_char	4	r1 = byte	Print byte as ASCII character.	None		
print_string	5	r1 = string address	Print null-terminated string at the address.	None		
	·		Input			
read_int	6	None	Read a signed 64-bit integer.	\$ret = integer		
read_float	7	None	Read a 32-bit float.	$\mathtt{$ret} = \mathtt{float}$		
read_double	8	None	Read a 64-bit double.	$\mathtt{$ret} = double$		
read_char	9	None	Read an ASCII character.	\$ret = character		
read_string	10	r1 = string address	Read a null-terminated string into given address.	None		
read_string	10	$r2 = \max length$	String is truncated to maximum length.			
Program Flow						
exit	11	None	Exit program.	None		
exit	11	Ivone	Note process exit code is located in \$ret.	None		
Debug						
print_regs	100	None	Print hexadecimal value of each register.	None		
print_mem	101	r1 = start address	Drint have desired but as of many as	None		
		Finit nexadecimal bytes of memory segment.	INOTIE			
print_stack	102	None	Print bytes of the stack.	None		