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.
		4	Execution status: 1=executing, 0=halted. 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	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>	8	Register offset.	$r1
<value>	2 + 32	Any listed addressing mode. 2 indicator bits, 32 for data.	0xdead
<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 ☐ 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	load <reg> <value>	Load a word into a register. Reg[$reg] = $value **Note** that any immediate is only 32-bit; Use loadw for loading a 64-bit immediate.
Load Upper	loadu <reg> <value>	Load a half-word (32-bit) into the upper half of a register. Reg[$reg][32:] = $value
Load Word	loadw <reg> <value>	*Pseudo-instruction.* Loads a word into a register. load $reg $value[:32] loadu $reg $value[32:] **Note** accepts a 64-bit immediate.
Zero	zero <reg>	*Pseudo-instruction.* Zeroes/clears a register. xor $reg, $reg
Store	store <reg> <addr>	Copy from register to memory. Mem[$addr] = Reg[$reg]

Arithmetic		
All arithmmetic operations, bar `mod`, expect a datatype.		
Add	`add <reg> <reg> <value>`	Add value to a register. `Reg[$reg1] = Reg[$reg2] + $value`
Subtract	`sub <reg> <reg> <value>`	Subtract value from a register. `Reg[$reg1] = Reg[$reg2] - $value`
Multiply	`mul <reg> <reg> <value>`	Multiply register by a value. `Reg[$reg1] = Reg[$reg2] × $value`
Division	`div <reg> <reg> <value>`	Divide a register by a value, store as double. `Reg[$reg1] = Reg[$reg2] ÷ $value`
Modulo	`mod <reg> <reg> <value>`	Calculate the remainder when dividing a register by a value. The register is treated as a signed word, the value as a signed half-word. `Reg[$reg1] = Reg[$reg2]` mod `$value`

Branching		
Compare	`cmp <reg> <value>`	Compare `$1` with `$2`, setting comparison bits in flag register. E.g., set `lt` iff `$1 < $2`. **Note** Z flag is set depending on value, not register.
Branch	`b<cnd> <value>`	*Pseudo-instruction* Branch to the given address if comparison matches conditional. `load<cnd> $ip, $value`
Jump ☐	`jmp <value>`	*Pseudo-instruction.* `load $ip $value`

Logical		
Not	`not <reg> <reg>`	Bitwise NOT a register. `Reg[$reg1] = ∼ Reg[$reg2]`
And	`and <reg> <reg> <value>`	Bitwise AND between register and value. `Reg[$reg1] = Reg[$reg2] & $value`
Or	`or <reg> <reg> <value>`	Bitwise OR between register and value. `Reg[$reg1] = Reg[$reg2] | $value`
Exclusive Or	`xor <reg> <reg> <value>`	Bitwise exclusive-OR between register and value. `Reg[$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`

Stack		

Push	`push <value>`	*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 Word	`pushw <value>`	*Pseudo-instruction*
		Push a 64-bit word onto the stack.
		`sub $sp, 8`
		`loadw $r1, <value>`
		`store $r1, ($sp)`

Pop	The pop operation is not implemented due to its simplistic nature.
	I.e., to pop a word from the stack:
	`sub $sp, 8`
	And to store it in a register:
	`load $r1, ($sp)`	

Functions		
Function Call	`call <value>`	Call procedure at location `value`. More complex than `load ip, $value` as pushes stack frame.
Return	`ret`	Return from function call. Restores key registers (undoes `call`).
System Call	`syscall <value>`	Invoke the system call mapped to the given value. See the respective section for mappings.

Miscellaneous		
No-Operation ☐	`nop`	Useless operation; do nothing. Equivalent to `or r1, 0`. **Note** For efficiency, implemented as instruction.
Exit	`exit [value]`	*Pseudo-instruction* Exit the program, optionally with an exit code in $ret. If code provided: `load $ret, <value>` `syscall <opcode: exit>`

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`	$\neq$	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 Decimal?	**Bit 1** Signed?	**Bit 0** Full or half word?	**Suffix**	**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:

```
push <arg1>
...
push <argn>
push n
call <func>
```

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

Stack			
Before		**After**	
		preserved GP registers	← $sp
		old ip	
		old fp	← $fp
		n	
		args	
`xxx`	← $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 pointer 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.	`$ret` = float
read_double	8	*None*	Read a 64-bit double.	`$ret` = double
read_char	9	*None*	Read an ASCII character.	`$ret` = character
read_string	10	`$r1` = string address `$r2` = max length	Read a null-terminated string into given address. String is truncated to maximum length.	*None*
Program Flow				
exit	11	*None*	Exit program. **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 `$r2` = segment length	Print hexadecimal bytes of memory segment.	*None*
print_stack	102	*None*	Print bytes of the stack.	*None*