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	6-64	
		5	Error status: 1=error, 0=ok. Indicates if exited gracefully.
		4	Execution status: 1=executing, 0=halted.
		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.
$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 generic value has the following format.

Indicator	Name	Syntax	Operation	Size
00	Immediate	`imm`	`imm`	
01	Memory	`(mem)`	`Mem[mem]`	
10	Register	`$reg`	`Reg[$reg]`	8
11	Register Offset	`n($reg)`	`Reg[$reg] + n`	`$reg`=8, `n` (signed)

Note: immediates and memory addresses are 32-bit, registers are 8-bit.

4 Instruction Set

Notes

- All instructions, unless indicated otherwise by a square □, contain three bits which exceed the opcode: a `test` bit to indicate if a conditional check is considered, followed by two bits equivalent to the first two bits in the flag register for comparison. If the `test` bit is 0, the check is skipped. This is indicated in the mnemonic by appending the mnemonic with a comparison code. This is indicated using a comparison postfix after a question mark: z, nz, eq, ne, lt, le, gt, ge. E.g., `add?eq`.

Instruction	Syntax	Operation/Comments
Data Transfer		
Load	`load <reg> <value>`	Load data into a register. `Reg[$reg] = $value`
Load Upper	`loadu <reg> <value>`	Load data into the upper half of a register. `Reg[$reg][32:] = $value`

Load Long	`loadl <reg> <value>`	*Pseudo-instruction.*
		Loads a 64-bit value into a register.
		`load $reg $value[:32]`
		`loadu $reg $value[32:]`

Zero	`zero <reg>`	*Pseudo-instruction.* Zeroes/clears a register. `xor $reg, $reg`
Store	`store <reg> <addr>`	Copy from register to memory. `Mem[$addr] = Reg[$reg]`
Arithmetic **Note**, the second `<reg>` is optional; if not present, registers are the same.		
Add	`add <reg> <reg> <value>`	Add value to a register. `Reg[$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. `Reg[$reg1] = Reg[$reg2]` ÷ `$value`
Integer division	`idiv <reg> <reg> <value>`	Divide a register by a value, cast result to integer. `Reg[$reg1] =` ⌊ `Reg[$reg2]` ÷ `$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.
Jump □	`jmp <addr>`	*Pseudo-instruction.* `load ip <addr>`

Branch □	`b<cnd> <addr>`	Branch to the given address if comparison matches conditional. See `$flag` register.
Logical		
Not	`not <reg> <reg>`	Bitwise NOT a register. `Reg[$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]` $\oplus$ `$value`
Logical Right Shift	`shr <reg> <reg> <value>`	Logically shift the register right an amount. `Reg[$reg1] = Reg[$reg2]` $\gg$ `$value`
Logical Left Shift	`shl <reg> <reg> <value>`	Logically shift the register left an amount. `Reg[$reg1] = Reg[$reg2]` $\ll$ `$value`
Stack		

Push	`push <value>`	Push a value onto the stack.
		`Mem[Reg[sp]] = $value`
		`Reg[sp] += 4`

Push Long	`pushl <value>`	*Pseudo-instruction*
		Push a 64-bit value onto the stack.
		`push $value[:32]`
		`push $value[32:]`

Pop	`pop [reg]`	*Pseudo-instruction* Pop value from the stack, load into register if provided. `sub sp, 4` If register: `load $reg, (sp)`
Pop Long	`popl [reg]`	*Pseudo-instruction* Pop a 64-bit value from the stack, load into register if provided. `sub sp, 8` If register: `loadl $reg, (sp)`
Functions		
Function Call	`call <value>`	Call procedure at location `value`. More complex than `load ip, $value` as pushes stack frame.
Store Arguments	`stargs <value> <value> ...`	*Pseudo-instruction* Push all argument values onto the stack. Useful shorthand for function call. `push $value1` ... `push $value`n `push` n **Note** assembler caches this n.
No Arguments	`noargs`	*Pseudo-instruction* Tells assembler that the next function call expects no arguments. `push 0` **Note** caches $n = 0$.
Load Argument	`ldarg <reg>` i	*Pseudo-instruction* Load the ith argument (assuming all 32-bit) into the register. `load $reg,` off`(fp)` **Note** see "retrieving arguments" for off calculation. **Note** number of arguments, n, is cached by the assembler.
Return	`ret`	Return from function 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`. Implemented as actual operation for efficiency.
Exit	`exit`	*Pseudo-instruction* Exit/halt the program. `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.

Argument Size The "addressing mode" section covers argument formatting, while here clarifies the size in bits of each argument type.

Argument	Size	Comment
`<reg>`	8	Register offset.
`<value>`	2 + 32	Any listed addressing mode. 2 indicator bits, 32 for data.
`<addr>`	2 + 32	Any listed addressing mode **except** immediate. 2 indicator bits, 30 for data.

Generic Layout A generic instruction layout, including the optional conditional testing. Most instructions take this format.

Bit	Purpose	Comments
0-5	Opcode	
6	Conditional test	If `test=1`: The next three bits are compared to respective bits in $flag. The instruction is only executed if they match. Otherwise, the test is skipped.
7-9	Test bits	Same as bits 0-2 in $flag.
10-64	Instruction dependant.	

Data-Type Inidicator 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	hs	32-bit signed integer.
0	1	1	s	64-bit signed integer.
1	0	0	f	32-bit float.
1	0	1	d	64-bit double.

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 `$r2` = max length	Read a null-terminated string into given address. String is truncated to maximum length.	*None*
Program Flow				
exit	11	*None*	Exit program.	*None*
exit2	12	`$r1` = exit code	Exit program with the given code.	*None*