The Processor

Ruben Saunders

September 2024

Contents

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 Memory Layout

The emulator is simple, able to run only one program.

The memory space has three regions: reserved, RAM, and stack.

- The reserved region contains two words.
  - Program entry point (i.e., initial $ip).
  - Address of interrupt handler.

  Note, these addresses refer to offsets in RAM.
- RAM is where user code is located.
- The stack grows downwards from the top of memory, with its base indicates via the $sp register.

3 Registers

See below for a list of registers. There are a total of 32 registers, and are all 64 bits wide. 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.
$rip	Return Address		Contains the sub-routiune return address. Must be pushed onto the stack as it is not preserved.
$sp	Stack Pointer		Top address of the stack.
$fp	Frame Pointer		Point to the next byte beyond the last stack frame.
$flag	Flag Register	9–64	
		8	Interrupt status: 1=in interrupt, 0=normal. Can be used to disable all interrupts.
		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. • 101: invalid datatype, bit field 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.
$isr	Interrupt Service Register		Used to indicate active interrupts. 64-bits, so 64 available distinguishable interrupts. By setting any bit, the processor enters an interrupt state.
$imr	Interrupt Mask Register		Used to mask $isr. That is, interrupt $isr[*i*] only triggers if $imr[*i*] is set. **Default**: all bits set.
$iip	Interrupt IP		Stores $ip in occurence of an interrupt.
$ret	Return Value Register		Contains value returned from function, syscall, etc. Contains process exit code on halt.
General Purpose Registers			
$k1, $k2	Internal Registers		Used by pseudo-instructions.
$r1 – $r21	General		Register for general use.

4 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

5 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 (except `mod`) and the compare operation take a datatype.

5.1 Data Transfer

5.1.1 Load

```
load <reg> <value>
```

Loads `<value>` as a word into register `<reg>`. Note, `<value>` may only specify a 32-bit immediate, so the upper 32 bits will always be zeroed in this case. To load a full 64-bit immediate, use `loadi`.

```
Reg[reg] <- value
```

5.1.2 Load Upper

```
loadu <reg> <value>
```

Loads `<value>` as a half-word into the upper half of register `<reg>`.

```
Reg[reg][32:] <- value
```

5.1.3 Load Immediate

Pseudo-instruction

```
loadi <reg> <imm>
```

Expands to

```
load <reg> <imm>[:32]
loadu <reg> <imm>[32:]
```

Loads `<imm>` as a word into register `<reg>`.

5.1.4 Zero

Pseudo-instruction

```
zero <reg>
```

Expands to

```
xor <reg> <reg>
```

Clears (zeroes) register `<reg>`.

5.1.5 Store

```
store <reg> <addr>
```

Stores the contents of register `<reg>` at the given address `<addr>`.

```
Mem[addr] <- Reg[reg]
```

5.1.6 Convert

```
cvt<d1>2<d2> <reg> <reg>
```

Converts the second register from data-type d_1 to d_2 and store in the first register.

```
Reg[reg1] <- cvt(Reg[reg2], d1, d2)
```

5.2 Arithmetic

Note that all mnemonics, except `mod`, expect a datatype flag.

5.2.1 Addition

```
add <reg> <reg> <value>
```

Add the value in the second register to `<value>` and store in the first register.

```
Reg[reg1] <- Reg[reg2] + value
```

5.2.2 Subtraction

```
sub <reg> <reg> <value>
```

Subtract `<value>` from the value in the second register and store in the first register.

```
Reg[reg1] <- Reg[reg2] - value
```

5.2.3 Multiplication

```
mul <reg> <reg> <value>
```

Multiply the value in the second register by `<value>` and store in the first register.

```
Reg[reg1] <- Reg[reg2] × value
```

5.2.4 Division

```
div <reg> <reg> <value>
```

Divide the value in the second register by `<value>` and store in the first register as a *double*.

```
Reg[reg1] <- Reg[reg2] ÷ value
```

5.2.5 Modulo

```
mod <reg> <reg> <value>
```

Calculate the remainder when dividing the second register by the value. The register is treated as a signed word; the value as a signed half-word.

```
Reg[reg1] <- Reg[reg2]  mod  value
```

5.3 Branching

5.3.1 Compare

```
cmp <reg> <value>
```

Compare the value in the register by `<value>`; set the comparison bits in the `$flag` register appropriately. E.g., set `lt` iff `$reg < $value`. **Note** the Z flag is set depending on value, not register.

5.3.2 Branch

Pseudo-instruction

```
b<cnd> <value>
```

Expands to

```
load<cnd> $ip <value>
```

Branch to the given value if the flag's comparison bits match the conditional guard.

5.3.3 Jump

Pseudo-instruction

```
jmp <value>
```

Expands to

```
load $ip <value>
```

An unconditional branch.

5.3.4 Jump and Link

```
jal [reg] <addr>
```

Performs a function call: stores the current `$ip` in `$rip` and jumps to `<addr>`. By default, this register is `$rip`; the optional register argument changes this.

```
Reg[reg1] <- Reg[$ip]
Reg[$ip] <- addr
```

5.4 Logical

5.4.1 And

```
and <reg> <reg> <value>
```

Computer the bitwise AND of the value in the second register and `<value>` and stores the result in the second register.

```
Reg[reg1] <- Reg[reg2] & value
```

5.4.2 Not

```
not <reg> <reg>
```

Stores the inverse (bitwise NOT) of the value in the second register in the first register.

```
Reg[reg1] <- ~ Reg[reg2]
```

5.4.3 Or

```
and <reg> <reg> <value>
```

Computer the bitwise OR of the value in the second register and `<value>` and stores the result in the first register.

```
Reg[reg1] <- Reg[reg2] | value
```

5.4.4 Exclusive-Or

```
xor <reg> <reg> <value>
```

Computer the bitwise exclusive OR of the value in the second register and `<value>` and stores the result in the first register.

```
Reg[reg1] <- Reg[reg2] ^ value
```

5.4.5 Left Shift

```
shl <reg> <reg> <value>
```

Logically shift the value in the second register left by `<value>` and store the result in the first register.

```
Reg[reg1] <- Reg[reg2] ≪ value
```

5.4.6 Right Shift

```
shl <reg> <reg> <value>
```

Logically shift the value in the second register right by `<value>` and store the result in the first register.

```
Reg[reg1] <- Reg[reg2] ≫ value
```

5.5 Interrupts

5.5.1 Trigger Interrupt

Pseudo-instruction

```
int <value>
```

Expands to

```
or $isr <value>
```

Mask the `$isr` register; trigger the given interrupt given the bit mask.

5.5.2 Return from Interrupt

Pseudo-instruction

```
rti
```

Expands to

```
load $iip $ip
and $flag ~FLAG_INTERRUPT_BIT
```

Restores instruction pointer to pre-interrupt state, and unlocks future interrupts.

5.6 Miscellaneous

5.6.1 No-Operation

```
nop
```

Do nothing; consume an instruction cycle.

5.6.2 Exit

Pseudo-instruction

```
    exit [value]
```

Expands to

```
    load $ret <value>
    syscall <exit>
```

Exit or halt the given process, optionally with a provided exit code.

5.6.3 System Call

```
    syscall <value>
```

Invoke the system call mapped to the given value. See the system call section for these mappings.

```
    pop stack frame
    Reg[$ip] <- old $ip
```

5.7 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.

5.8 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.

6 Interrupts

Interrupts are events which, when triggered, alert the processor immediately. Interrupts are triggered via the $isr register and may be used to distinguish between different sources. The $isr is used to mask, or ignore, some interrupts. Note that the interrupt bit must be cleared manually. Also note that while in an interrupt, no other interrupt can be handled.

Below is listed C pseudocode for the fetch-execute cycle to understand interrupt behaviour:

```c
void fetch_execute_cycle(void) {
    if (($isr & $imr) && !($flag & FLAG_IN_INTERRUPT)) {
        handle_interrupt();
    }

    word instruction = fetch();
    execute(instruction);

    $ip += sizeof(word);
}

void handle_interrupt(void) {
    $iip = $ip;
    $flag |= FLAG_IN_INTERRUPT;
    $ip = HANDLER_OFFSET;
}

void return_from_interrupt(void) {
    $ip = $iip;
    $flag &= ~FLAG_IN_INTERRUPT;
}
```

Note the handler offset is at the fixed memory location `0x400`.

7 Subroutines

As a RISC processor, little support is provided by the processor for calling subroutines; instead, it is up to compilers or other softwares to decide upon and implement such a convention.

The process provides one instruction – `jal` – to handle calling a subroutine. Given the location in memory of the subroutine, it first caches the old instruction pointer, then loads in the subroutine's address. Note that this return address is not preserved, and hence must be cached by the programmer if multiple nested calls are required, lest the previous return address be overwritten.

To return from a subroutine is simple: load the contents of this cache into the instruction pointer register.

8 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_hex	0	`$r1` = integer	Print register as hexadecimal.	*None*
print_int	1	`$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*

9 Application Overview

The application, named `processor`, is a simple program which implements the processor detailed therein. It is called as follows:

```
$ ./processor <input_file> [flags]
```

Where the program has the following flags:

- `-o <output_file>` - defaults to `stdout`, output is written here (not including debug messages).
- `-i <input_file>` - defaults to `stdin`, input is read from here.

9.1 Debugging

There are no flags to control debugging. Rather, the debugging level is controlled by the `DEBUG` macro in `src/debug.h`, forming a bitmask of possible debug fields. This file may be modified and the application re-compiled for debugging changes to take effect.