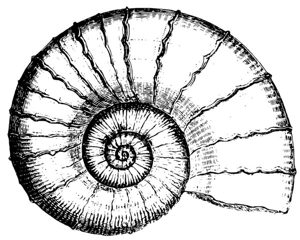
The Shell

Instruction Set Architecture



Jonathan Rice Shelley II

September 16, 2020

Version 1.0

1 Contents

2 Shell ISA Overview

The Shell ISA follows RISC design principles and is intended for a new 16-bit microprocessor. The primary goals are low cost implementation and minimal clock cycles per instruction. The Shell ISA contains 8 unique instructions encoded by a 3-bit op code. Multiple encodings of instructions results in 23 assembly instructions made available to the programmer. 16 bit data words (2's complement) and instructions are used exclusively in the microprocessor design. The programmer has access to 7 programmable registers in addition to an 8th register that is always zero. All programmer accessible registers are one word in length or 16 bits. Linear addressing of 1K 16-bit word-addressable only memory is supported.

3 Register mapping and suggested usage conventions

The table below list the user addressable registers and their intended use. In addition to the 8 registers provided, the programmer can also interact with the stack pointer register via the lsp (load stack pointer) instruction. The programmer is only required to initialize the stack pointer updating the SP's value on push and pull is handled in hardware.

GPR address	Register	Description
000	r0	always zero
001	r1	general purpose
010	r2	general purpose
011	r3	general purpose
100	r4	general purpose
101	r5	size of stack frame
110	r6	subroutine return register
111	r7	subroutine return address

4 Unique instructions

The 8 unique instructions supported by the ISA are listed below.

Instruction opcode	**Description**
000	System commands (halt)
001	Branching instructions
010	Jump and link register
011	Load upper immediate
100	Store word
101	Load word
110	ALU instructions
111	Stack instructions

5 Instruction definitions

All hardware instructions are listed below by their instruction type. x's in bit fields denote a value assigned by the programmer.

5.1 RRR-Type instructions

The RRR-type instruction group is responsible for encoding instructions that require 3 registers. The instruction type has a 4 bit function field to support additional formats. The majority of RRR-type instructions are ALU related.

	16 bit instruction encoding				
Mnemonic & operands	**OP code**	**Func 4**	**Dest register**	**Register operand 1**	**Register operand 2**
add R_1, R_2, R_3	110	0000	xxx	xxx	xxx
sub R_1, R_2, R_3	110	0001	xxx	xxx	xxx
and R_1, R_2, R_3	110	0010	xxx	xxx	xxx
or R_1, R_2, R_3	110	0011	xxx	xxx	xxx
xor R_1, R_2, R_3	110	0100	xxx	xxx	xxx
nand R_1, R_2, R_3	110	0101	xxx	xxx	xxx

5.2 RR-Type instructions

The RR-type instruction group encodes two registers. The instruction type has a 4 bit function field to support additional instruction encodings.

	16 bit instruction encoding				
Mnemonic & operands	**OP code**	**Function 4**	**Destination register**	**Source register**	**Zero padding**
lw R_1, R_2	101	0000	xxx	xxx	000
sw R_1, R_2	100	0000	xxx	xxx	000
asr R_1, R_2	110	0110	xxx	xxx	000
asl R_1, R_2	110	0111	xxx	xxx	000
cmp R_1, R_2	110	1010	xxx	xxx	000
jalr R_1, R_2	010	0000	xxx	xxx	000

5.3 R-Type instructions

The R-type instruction group encodes one register. The instruction group has a 4 bit function field to support additional instruction encodings.

	16 bit instruction encoding			
Mnemonic & operands	**OP code**	**Function 4**	**Operand register**	**Zero padding**
push R_1	111	0000	xxx	000000
pop R_1	111	0001	xxx	000000
lsp R_1	111	0010	xxx	000000

5.4 RI-Type instructions

The RI-type instruction group supports register immediate operations. The immediate operand must be less than 6 bits in length. The instruction group has a 4 bit function field to support additional instruction encodings.

	16 bit instruction encoding			
Mnemonic & operands	**OP code**	**Function 4**	**Operand register**	**Immediate operand**
addi R_1, imm	110	1000	xxx	xxxxxx
subi R_1, imm	110	1001	xxx	xxxxxx
lui R_1, imm	011	0000	xxx	xxxxxx

5.5 B-Type instructions

The B-type instruction group contains branching instructions. The group encodes an 11 bit signed immediate used to specify branch distance relative to the PC. The instruction group has a 2 bit function field to support additional instruction encodings.

	16 bit instruction encoding		
Mnemonic & operands	**OP code**	**Function 2**	**Immediate address operand**
beq *imm*	001	00	xxxxxxxxxxx
bne *imm*	001	01	xxxxxxxxxxx
bgt *imm*	001	10	xxxxxxxxxxx
blt *imm*	001	11	xxxxxxxxxxx

5.6 S-Type instructions

The S-type instruction group contains system commands. The group consist only of an opcode field followed by a 13 bit function field.

	16 bit instruction encoding	
Mnemonic & operands	**OP code**	**Function 13**
hlt	000	0000000000000

6 Programmers assembly reference

The table below list all available assembly instructions. This table serves as a quick reference for system programmers. Instruction definitions section should be referenced for the assembly instructions binary encoding.

No.	**Mnemonic**	**Description**
1	hlt	Halts program execution
2	add R_1, R_2, R_3	adds contents of registers R_2 and R_3 result stored in R_1
3	sub R_1, R_2, R_3	subtracts contents of registers R_2 and R_3 result stored in R_1
4	and R_1, R_2, R_3	logical and the contents of registers R_2 and R_3 result stored in R_1
5	or R_1, R_2, R_3	logical or the contents of registers R_2 and R_3 result stored in R_1
6	xor R_1, R_2, R_3	logical xor the contents of registers R_2 and R_3 result stored in R_1
7	nand R_1, R_2, R_3	logical nand the contents of registers R_2 and R_3 result stored in R_1
8	lw R_1, R_2	16-bit word in memory at address R_2 loaded in register R_1
9	sw R_1, R_2	contents of register R_1 stored at memory address R_2
10	asr R_1, R_2	arithmetic shift right R_2 store result in R_1
11	asl R_1, R_2	arithmetic shift left R_2 store result in R_1
12	cmp R_1, R_2	compare register R_1 with register R_2 results stored in status register
13	jalr R_1, R_2	jump to address in register R_2 store current address + 1 in register R_1
14	push R_1	push contents of register R_1 onto stack
15	pop R_1	pop 16-bit word off stack into register R_1
16	lsp R_1	(load stack pointer) contents of register R_1 transferred into stack pointer
17	addi R_1, imm	add immediate to R_1 store result in R_1
18	subi R_1, imm	subtract immediate from R_1 store result in R_1
19	lui R_1, imm	load upper immediate 10 bits then zero bottom 6 bits of register R_1.
20	beq imm	branch if last comparison instruction (No. 12) indicated $R_1 == R_2$
21	bne imm	branch if last comparison instruction (No. 12) indicated $R_1 \mathrel{!=} R_2$
22	bgt imm	branch if last comparison instruction (No. 12) indicated $R_1 > R_2$
23	blt imm	branch if last comparison instruction (No. 12) indicated $R_1 < R_2$

7 Common C language constructs compiled in assembly

Listed below are common C language constructs and how they would be implemented in the Shell ISA.
Note: Information inside asterisks is a comment.

7.1 Assignment

x = 5;	xor r3, r3, r3 addi r3, 5

7.2 Addition

x = x + y;	add r3, r3, r4
x = y + y;	add r3, r4, r4
y = x + x;	add r4, r3, r3

7.3 Subtraction

x = x - y;	sub r3, r3, r4
x = y - y;	sub r3, r4, r4
y = x - x;	sub r4, r3, r3

7.4 Logical operations

x = x & y;	and r3, r3, r4
x = x | y;	or r3, r3, r4
x = y ˆ y;	xor r3, r3, r4
y = ˜ (y & x);	nand r3, r3, r4

7.5 For loop control structure

int i;
int x = 0;
for (i = 0; i < 3; i++) {
x = x + 1;
}
	xor r1, r1, r1
	xor r2, r2, r2
	xor r3, r3, r3
	addi r3, 3
	addi r2, 1
	addi r1, 1
	cmp r1, r3
	blt -3

7.6 while loop control structure

int x = 0;
while(1) {
x = x + 1;
}
	xor r1, r1, r1
	xor r2, r2, r2
	addi r2, 3
	addi r1, 1
	jalr r7, r2

7.7 if-else control structure

if (x == 0) {	cmp r1, r0
do this	bnq 5
} else {	*do this*
do this	xor r3, r3, r3
}	addi r3, 8
	jalr r7, r3
	do this

7.8 Relational operator ”==”

if (x == y) {	cmp r1, r2
}	beq *PC relative immediate address of a label*

7.9 Relational operator ”!=”

if (x != y) {	cmp r1, r2
}	bne *PC relative immediate address of a label*

7.10 Relational operator less than

if (x < y) {	cmp r1, r2
}	blt *PC relative immediate address of a label*

7.11 Relational operator greater than

if (x > y) {	cmp r1, r2
}	bgt *PC relative immediate address of a label*

7.12 Relational operator greater than or equal to

if (x >= y) {	cmp r1, r2
}	bgt *PC relative immediate address of a label*
	bge *PC relative immediate address of a label*

7.13 Relational operator less than or equal to

if (x <= y) {	cmp r1, r2
}	blt *PC relative immediate address of a label*
	bge *PC relative immediate address of a label*

7.14 Functions (call and return) pass by value

int foo(int a, int b) {	xor r1, r1, r1
return a + b;	xor r2, r2, r2
}	xor r3, r3, r3
int main() {	addi r1, 3
foo(3, 3);	addi r2, 3
}	addi r3, 50
	lsp r3
	xor r3, r3, r3
	addi r3, *calculated address of foo label*
	push r1
	push r2
	jalr r7, r3
	pop r1
	xor r3, r3, r3
	addi r3, *calculated address of exit label*
	jalr r7, r3
	foo:
	pop r1
	pop r2
	add r1, r1, r2
	push r1
	jalr r7, r7
	exit:

7.15 Functions (call and return) pass by reference

void foo(int* a, int* b) {	xor r1, r1, r1
*a = *a + *b;	xor r2, r2, r2
}	addi r1, 3
	addi r2, 3
int main() {	xor r3, r3, r3
int a = 3;	addi r3, 50
int b = 3;	lsp r3
foo(&a, &b);	xor r3, r3, r3
}	sw r1, r3
	xor r4, r4, r4
	addi r4, 1
	sw r2, r4
	xor r2, r2, r2
	addi r2, *calculated address of foo label*
	push r3
	push r4
	jalr r7, r2
	xor r3, r3, r3
	addi r3, *calculated address of exit label*
	jalr r7, r3
	foo:
	pop r1
	pop r2
	lw r3, r1
	lw r4, r2
	add r3, r3, r4
	sw r3, r1
	jalr r7, r7
	exit:

8 Instruction usage and justification.

No.	Mnemonic	Usage and justification
1	hlt	Halts program execution
2	add R_1, R_2, R_3	add two 16 bit words
3	sub R_1, R_2, R_3	subtract two 16 bit words
4	and R_1, R_2, R_3	perform logical and between registers
5	or R_1, R_2, R_3	perform logical or between registers
6	xor R_1, R_2, R_3	perform logical xor between registers
7	nand R_1, R_2, R_3	perform logical nand between registers
8	lw R_1, R_2	load 16-bit word form memory
9	sw R_1, R_2	store 16-bit word in memory
10	asr R_1, R_2	perform an arithmetic shift right on register (reg / 2)
11	asl R_1, R_2	perform an arithmetic shift left on register (reg * 2)
12	cmp R_1, R_2	compare two 16-bit words
13	jalr R_1, R_2	unconditional jump to a subroutine store return address in register
14	push R_1	push 16-bit word onto stack
15	pop R_1	pop 16-bit word off stack
16	lsp R_1	initialize the stack pointer
17	addi R_1, imm	add an immediate to a register
18	subi R_1, imm	subtract an immediate from a register
19	lui R_1, imm	load upper 10 bits of a register and zero bottom six bits
20	beq imm	PC relative branch if two compared registers were equal
21	bne imm	PC relative branch if two compared registers were not equal
22	bgt imm	PC relative branch if a register is greater than another register
23	blt imm	PC relative branch if a register is less than another register