RISC-Z Instruction Set Architecture

1 Design Requirements

RISC-Z is a Reduced Instruction Set Computing based 16-bit architecture with the following specifications:

- 16 bit words with no more than 16 unique instructions (opcodes) including a halt instruction. However, there may be multiple formats for a given type of instruction, if necessary.
- 1K 16-bit words of memory
- All operands are to be 16-bit signed integers
- Supports various C programming language constructs

This report will detail the mechanics of the RISC-Z Instruction Set Architecture including the registers and instructions. Various different required constructs of the C programming language will also be compiled down to the assembly and machine code.

2 Register Structure

register	qualifier	usage
x0	`x0`	Constant Pull Down
x1	`ra`	Return Address
x2	`fp`	Frame Pointer
x3	`sp`	Stack Pointer
x4	`pc`	Program Counter
x5-x15	`x5-x15`	General Purpose

Table 1: Registers and their general usage.

The RISC-Z architecture provides 16 registers, which are named `x0-x15`. Each register is 16 bits wide, with 5 of the 16 registers having a designated purpose. The usage of the general purpose registers are user dependent.

2.1 Constant Pull Down

The register `x0` is constantly pulled to ground. Reading from `x0` always produces `0x0000`. This is useful for quick operations.

2.2 Return Address

The register `ra` contains the return address needed when returning from subroutines to the main program.

2.3 Frame and Stack Pointer

The `fp` register points to the base of the stack. The `sp` register points to the top of the stack.

2.4 Program Counter

The program counter contains the address of the instruction being executed at the present time. It is incremented with each clock cycle.

3 Instruction Formats

The RISC-Z architecture supports seven different instruction formats: Register, Immediate, Jump, Branch, Load/Store, Shift, and Halt (**R, I, J, B, LS, S, H**). All formats have a total instruction width of 16 bits, fitting one instruction in each word of memory. Instructions consist of an `opcode`, `registers`, and/or an `immediate`.

- The opcode indicates the type and format of the instructions.
- Registers are addressed in instructions using a logical 4 bit field containing the value of a source or destination register. Registers are represented by `RS1 (source 1), RS2 (source 2), or RD (destination)`.
- Immediate are data that is to be loaded into or used to modify a register in some way, which will be encoded using two's complement.

The formats for each instruction are elaborated in the sections below.

3.1 Register Mode

Register mode is for instructions that utilize three registers, which consist of two source registers and a destination register. These instructions are used for arithmetic and other register exclusive operations.

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	
RS1	RS2	RD	OPCODE	
RS1	RS2	RD	0001	`addr`
RS1	RS2	RD	0010	`andr`
RS1	RS2	RD	0011	`nandr`
RS1	RS2	RD	0100	`orr`
RS1	RS2	RD	0101	`subr`

3.2 Immediate Mode

Immediate mode is used for instructions that use a single register and an 8 bit immediate operand. These instructions are used to easily move and increment/decrement constants in registers. Immediate values are encoded using **two's complement**.

15 14 13 12 11 10 9 8	7 6 5 4	3 2 1 0	
Immediate[7:0]	RD	OPCODE	
Immediate[7:0]	RD	0110	`addi`
Immediate[7:0]	RD	0111	`movu`
Immediate[7:0]	RD	1000	`movl`

3.3 Jump Mode

Jump mode is for calling and returning from subroutines. The `R` field indicates a return from a subroutine. Immediate values are encoded using **two's complement**.

15 14 13 12 11 10 9 8 7 6	5	4	3 2 1 0	
Immediate[9:0]	-	R	OPCODE	
Immediate[9:0]	-	0	1001	`jmp`
Immediate[9:0]	-	1	1001	`ret`

3.4 Branch Mode

Branch mode is for evaluating a conditional statement and executing an instruction based on the result. Immediate values are encoded using **two's complement**.

15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0	
RS1	RS2	Imm[3:0]	OPCODE	
RS1	RS2	Imm[3:0]	1010	`beq`
RS1	RS2	Imm[3:0]	1011	`bne`
RS1	RS2	Imm[3:0]	1100	`bgt`
RS1	RS2	Imm[3:0]	1101	`blt`

3.5 Load/Store Mode

Load/Store mode is used to load or store values from memory. The memory addresses are stored in a register. Since load and store share the same opcode, they are differentiated using the `LS` field.

15 14 13 12	11 10	9 8	7 6 5 4	3 2 1 0	
RS1	- -	LS	RD	OPCODE	
RS1	- -	10	RD	1110	`ld`
RS1	- -	01	RD	1110	`str`

3.6 Shift Mode

Shift mode is used to shift bits in a register. The `D` field indicates the shift direction, and the `A` field indicates an arithmetic shift. Immediate values should be **positive numbers** only, which will be encoded using **two's complement**.

15 14 13 12	11 10 9 8 7 6	5 4	3 2 1 0	
RS1	Immediate[5:0]	AD	OPCODE	
RS1	Immediate[5:0]	00	1111	`shl`
RS1	Immediate[5:0]	10	1111	`sal`
RS1	Immediate[5:0]	01	1111	`shr`
RS1	Immediate[5:0]	11	1111	`sar`

3.7 Halt Mode

Halt mode is used to stop execution of the program.

15 14 13 12 11 10 9 8 7 6 5 4	3 2 1 0	
- - - - - - - - - - - -	OPCODE	
- - - - - - - - - - - -	0000	`halt`

4 Instruction Usage

In the following sections, each instruction is described along with a justification for the instruction and any registers used.
Note: Only 16 opcodes are used, but there are more than 16 instructions since some share the same opcodes.

4.1 The Add Register Instruction: `addr`

`addr` adds the contents of the RS1 and RS2 registers and stores the result in the RD register.

`addr` is included in order to perform 16 bit signed addition.

Format: `addr RD, RS1, RS2`

4.2 The And Register Instruction: `andr`

`andr` performs a logical AND on the RS1 register using the mask specified by the RS2 register, and stores the result in RD.

`andr` is included in order to perform logical AND operations.

Format: `andr RD, RS1, RS2`

4.3 The Nand Register Instruction: `andr`

`nandr` performs a logical NAND on the RS1 register using the mask specified by the RS2 register, and stores the result in RD.

`nandr` is included in order to perform logical NAND operations.

Format: `nandr RD, RS1, RS2`

4.4 The Nand Register Instruction: `orr`

`orr` performs a logical OR on the RS1 register using the mask specified by the RS2 register, and stores the result in RD.

`orr` is included in order to perform logical OR operations.

Format: `orr RD, RS1, RS2`

4.5 The Subtract Register Instruction: `subr`

`subr` subtracts the contents of RS2 from the RS1 register and stores the result in the RD register.

`subr` is included in order to perform 16 bit signed subtraction.

Format: `subr RD, RS1, RS2`

4.6 The Add Immediate Instruction: `addi`

`addi` adds an 8 bit signed immediate to the lower half of RD, which is the destination register.

`addi` is included in order to quickly add a small number to a register, for things like the index of loops.

Format: `addi RD, Immediate`

4.7 The Move Upper Instruction: `movu`

`movu` places the 8 bit immediate into the upper 8 bits of the destination register.

`movu` is included in order to easily move data into a register.

Format: `movu RD, Immediate`

4.8 The Move Lower Instruction: `movl`

`movl` places the 8 bit immediate into the lower 8 bits of the destination register.

`movl` is included in order to easily move data into a register.

Format: `movl RD, Immediate`

4.9 The Jump Instruction: `jmp`

`jmp` stores the address of the next instruction in the `x1` register and sets the next instruction to be the `PC` + Label. The `jmp` instruction can jump $\pm 2^{10}$ instructions from the `PC`.

`jmp` is included in order to call subroutines.

Format: `jmp Label`

4.10 The Return Instruction: `ret`

`ret` sets the address of the next instruction to the value stored in in the x1 register in order to return to the main program.

`ret` is included in order return from subroutines.

Format: `ret`

4.11 The Branch Equal Instruction: `beq`

`beq` compares the RS1 and RS2 registers. If they are equal, then the next instruction becomes `PC` + Label. The `beq` instruction can only branch $\pm$ 16 instructions from the `PC`.

`beq` is included in order to execute instruction(s) only if an equals condition is met.

Format: `beq RS1, RS2, Label`

4.12 The Branch Not Equal Instruction: `bne`

`bne` compares the RS1 and RS2 registers. If they are not equal, then the next instruction becomes `PC` + Label. The `bne` instruction can only branch $\pm$ 16 instructions from the

PC.

bne is included in order to execute instruction(s) only if a not equal condition is met.

Format: `bne RS1, RS2, Label`

4.13 The Branch Greater Than Instruction: bgt

bgt compares the RS1 and RS2 registers. If RS1 is greater than RS2, then the next instruction becomes PC + Label. The bgt instruction can only branch ± 16 instructions from the PC.

bgt is included in order to execute instruction(s) only if a greater than condition is met.

Format: `bgt RS1, RS2, Label`

4.14 The Branch Less Than Instruction: blt

blt compares the RS1 and RS2 registers. If RS1 is less than RS2, then the next instruction becomes PC + Label. The blt instruction can only branch ± 16 instructions from the PC.

blt is included in order to execute instruction(s) only if a less than condition is met.

Format: `blt RS1, RS2, Label`

4.15 The Load Register Instruction: ld

ld stores the 16 bit word at the address specified by RS1 in RD.

ld is included in order to transfer words from memory to registers.

Format: `ld RD, RS1`

4.16 The Store Register Instruction: str

str stores the contents of RS1 at the memory address specified by RD.

str is included in order to transfer words from registers to memory.

Format: `str RD, RS1`

4.17 The Shift Logical Left Instruction: `shl`

`shl` is used to perform a logical left shift for the amount of bits specified by the immediate.

`shl` is included in order to logically shift values to the left (multiply by powers of 2).

Format: `shl RS1, Immediate`

4.18 The Shift Arithmetic Left Instruction: `sal`

`sal` is used to perform an arithmetic left shift for the amount of bits specified by the immediate.

`sal` is included in order to preserve the sign for left shifts.

Format: `sal RS1, Immediate`

4.19 The Shift Logical Right Instruction: `shr`

`shr` is used to perform a logical right shift for the amount of bits specified by the immediate.

`shr` is included in order to logically shift values to the right (divide by powers of 2).

Format: `shr RS1, Immediate`

4.20 The Shift Arithmetic Left Instruction: `sar`

`sar` is used to perform an arithmetic right shift for the amount of bits specified by the immediate.

`sar` is included in order to preserve the sign for right shifts.

Format: `sar RS1, Immediate`

4.21 The Halt Instruction: `halt`

`halt` is used to stop execution of the program.

`halt` is included in order to stop execution.

Format: `halt`

5 Pseudoinstructions

5.1 Push

The pseudoinstruction `push` may be used to place the contents of a register onto the stack.

Format: `push RS1`

It is equivalent to the following instructions:

```
addi sp, -1
str sp, RS1
```

5.2 Pop

The pseudoinstruction `pop` may be used to pop the stack into a register.

Format: `pop RS1`

It is equivalent to the following instructions:

```
ld RS1, sp
addi sp, 1
```

5.3 Move

The pseudoinstruction `mov` may be used to transfer the contents of registers. The destination register is listed first, followed by the source register.

Format: `mov RD, RS1`

It is equivalent to the following instruction:

```
addr RD, RS1, x0
```

6 Assembly Examples

The sections below are examples of how C language constructs would be translated into RISC-Z assembly and machine code.

6.1 Assignment

C Language: `x = y;`

Registers: `x = x5`, `y = x6`

Assembly: `addr, x5, x6, x0`

Machine Code: `0110 0000 0101 0001`

6.2 Addition and Subtraction

C Language: `z = x + y;`

Registers: `x = x5`, `y = x6`, `z = x7`

Assembly: `addr, x7, x5, x6`

Machine Code: `0101 0110 0111 0001`

C Language: `z = x - y;`

Registers: `x = x5`, `y = x6`, `z = x7`

Assembly: `subr, x7, x5, x6`

Machine Code: `0101 0110 0111 0101`

6.3 Logical Operations

C Language: `x &= y;`

Registers: `x = x5`, `y = x6`

Assembly: `andr x5, x5, x6`

Machine Code: `0101 0110 0101 0010`

C Language: `x |= y;`

Registers: `x = x5`, `y = x6`

Assembly: `orr x5, x5, x6`

Machine Code: `0101 0110 0101 0100`

6.4 Loops and Control Flow

C Language:
```
if (z == 0) {
   x = y;
}
else {
   z = x;
}
```

Registers: `x = x5`, `y = x6`, `z = x7`

Assembly:
```
         beq x7, x0, equal
         addr x7, x5, x0
equal:   addr x5, x6, x0
```

Machine Code:
```
0111 0000 0010 1010
0101 0000 0111 0001
0110 0000 0101 0001
```

C Language:
```
while (x > 0) {
   x--;
}
```

Registers: `x = x5`

Assembly:
```
Loop:  addi x5, -1
       bgt x5, x0, Loop
```

Machine Code:
```
1111 1111 0101 0110
0101 0000 1111 1100
```

C Language:
```
for (i = 10; i >= 0; i--) {
   x = x;
}
```

Registers: `i = x5`, `x = x6`

Assembly:
```
        addr x5, x0, x0
        addi x5, 10
Loop:   addr x6, x6, x0
        addi x5, -1
        beq x5, x0, Loop
        bgt x5, x0, Loop
```

Machine Code:
```
0000 0000 0101 0001
0000 1010 0101 0110
0110 0000 0110 0001
1111 1111 0101 0110
0101 0000 1110 1010
0101 0000 1101 1100
```

6.5 Function Calls

C Language:
```
int funct(int x) {
   int y;
   y = x + 1;
   return(y);
}
```

Registers: `x = x5, y = x6`

Assembly:
```
jmp funct
halt
funct:  ld x5, sp
        addi sp, 1
        addr x6, x5, x0
        addi x6, 1
        addi sp, -1
        str sp, x6
        ret
```

Machine Code:
```
0000 0000 1000 1001
0011 0010 0101 1110
0000 0001 0011 0110
0101 0000 0110 0001
0000 0001 0110 0110
1111 1111 0011 0110
0101 0001 0011 1110
0000 0000 0001 1001
0000 0000 0000 0000
```