Zen Processor Unit

Volume One, Programmer's Guide, revision 0.0.1

Tuomo Petteri Venäläinen

June 29, 2014

Contents

I Preface 5

1 Notes 7

II The Machine 9

2 Architecture 11

3 Instruction Set 13
3.1 Instruction Reference . . . 13
3.1.1 Instruction Set . . . 13
3.1.2 Instruction Table . . . 14
3.1.2.1 Base Instruction Set . . . 14
3.1.2.2 Rational Number Extensions . . . 15
3.1.2.3 SIMD Extensions . . . 15

4 Reference 17
4.1 Opcode Format . . . 17
4.2 Instruction Set . . . 17
4.2.1 Cheat Sheet . . . 19

III The Assembler 21

5 Zen Assembly 23
5.1 Syntax . . . 23
5.2 Assembler Directives . . . 24
5.2.1 .include . . . 24
5.2.2 .import . . . 25
5.2.3 .org . . . 25
5.2.4 .space . . . 25
5.2.5 .align . . . 25
5.2.6 .globl . . . 25
5.2.7 .long . . . 25
5.2.8 .byte . . . 25
5.2.9 .short . . . 26
5.2.10 .asciz . . . 26
5.2.11 .define . . . 26

Part I

Preface

Chapter 1

Notes

Brief

Zen Processor Unit aims to be a flexible, minimalistic, RISC- like load-store architecture. It's first going to be implemented as a virtual machine; hopefully later in hardware using FPGA. :)

Background and Future

One of the main goals of the project is to supply a fast-enough FPGA- implemented CPU for running old games using ScummVM. :)

Part II

The Machine

Chapter 2

Architecture

Zen Processor Unit is a load-store architecture, i.e. there's just one family of instructions (MOV*) to interface with memory; the rest of the instruction set operates on registers.

Native word size is **32-bit**. Words are little endian, i.e. lowest byte first.

Extended data types include rational numbers with signed 32-bit nominator and denominator as well as 32-bit and 64-bit IEEE754 floating point types (float and double, usually).

Rudimentary SIMD support lets you do certain operations on several argument pairs in parallel; e.g., you could pack 4 8-bit bytes into 64-bit register as 16-bit integers and then add them, optionally with signed or unsigned saturation.

Chapter 3

Instruction Set

The ZEN instruction set was designed to resemble the C language closely, as well as support a RISC-oriented set of typical machine operations.

3.1 Instruction Reference

3.1.1 Instruction Set

Operands

- **i** stands for immediate operand
- **r** stands for register operand
- **m** stands for memory operand

Flags

Certain instructions set bits in the machine status word (MSW). This is documented here on per-instruction basis.

- **C** stands for carry flag (CF)
- **I** stands for interrupt flag (IF)
- **Z** stands for zero flag (ZF)
- **V** stands for overflow flag (VF)

TODO: stack/call conventions for certain instructions

3.1.2 Instruction Table

3.1.2.1 Base Instruction Set

Mnemonic	Source	Destination	Brief	Flags
not	r	N/A	op1 = op1;	Z
and	r/i	r	op2 = op2 & op1;	Z
or	r/i	r	op2 = op2 | op1;	N/A
xor	r/i	r	op2 = op2 ôp1;	Z
shr	r/i	r	op2 = op2 » op1; (zero-fill)	Z
sar	r/i	r	op2 = op2 » op1; (sign-fill)	Z
shl	r/i	r	op2 = op2 « op1;	O, C
ror	r/i	r	op2 = op2 ROR op1;	C
rol	r/i	r	op2 = op2 ROL op1;	C
inc	r/i	N/A	op1++;	O
dec	r/i	N/A	op1–;	O, Z
add	r/i	r	op2 = op2 + op1;	O, Z
sub	r/i	r	op2 = op2 - op1;	Z
cmp	r/i	r	Compare two values and set flags	Z
mul	r/i	r	op2 = op2 * op1;	O, Z
div	r/i	r	op2 = op2 / op1;	Z
mod	r/i	r	op2 = op2 % op1;	N/A
bz	r/i	N/A	branch to op1 if (CF == 0)	N/A
bnz	r/i	N/A	branch to op1 if (CF != 0)	N/A
blt	r/i	N/A	branch to op1 if (SF != 0)	N/A
ble	r/i	N/A	branch to op1 if (SF != 0) || (ZF == 0)	N/A
bgt	r/i	N/A	branch to op1 if (SF != 0) && (ZF != 0)	N/A
bge	r/i	N/A	branch to op1 if (SF != 0) || (ZF == 0)	N/A
bo	r/i	N/A	branch to op1 if (OF != 0)	N/A
bno	r/i	N/A	branch to op1 if (OF == 0)	N/A
bc	r/i	N/A	branch to op1 if (CF != 0)	N/A
bnc	r/i	N/A	branch to op1 if (CF == 0)	N/A
pop	N/A	N/A	pop top of stack	N/A
push	r/i	N/A	push value on stack	N/A
pusha	r/i	N/A	push all registers on stack	N/A
mov	r/i/m	r/i/m	load or store 32-bit longword	N/A
movb	r/i/m	r/i/m	load or store 8-bit byte	N/A
movw	r/i/m	r/i/m	load or store 16-bit word	N/A
movq	r/i/m	r/i/m	load or store 64-bit word	N/A
jmp	r/i	N/A	unconditional branch	N/A
call	r/i	N/A	call subroutine; construct stack frame	N/A
enter	N/A	N/A	enter subroutine	N/A
leave	N/A	N/A	leave subroutine	N/A
ret	N/A	N/A	return from subroutine	N/A
lmsw	r/i	N/A	load machine status word	N/A
smsw	m	N/A	load machine status word	N/A

3.1.2.2 Rational Number Extensions

Mnemonic	Source	Destination	Brief	Flags
radd	r/i	r/i	add two rational numbers	N/A
rsub	r/i	r/i	subtract two rational numbers	N/A
rmul	r/i	r/i	multiply two rational numbers	N/A
rdiv	r/i	r/i	divide two rational numbers	N/A

3.1.2.3 SIMD Extensions

Mnemonic	Source	Destination	Brief	Flags
vshrw	r/i	r/i	shift 16-bit words right (zero-fill)	N/A
vsarb	r/i	r/i	shift 8-bit bytes right (sign-fill)	N/A
vsarw	r/i	r/i	shift 16-bit words right (sign-fill)	N/A
vshlb	r/i	r/i	shift 8-bit bytes left	N/A
vshlw	r/i	r/i	shift 16-bit words left	N/A
vaddb	r/i	r/i	add 8-bit bytes	N/A
vaddbs	r/i	r/i	add 8-bit bytes with signed saturation	N/A
vaddbu	r/i	r/i	add 8-bit bytes with unsigned saturation	N/A
vaddw	r/i	r/i	add 16-bit words	N/A
vsubb	r/i	r/i	subtract 8-bit bytes	N/A
vsubbs	r/i	r/i	subtract 8-bit bytes with signed saturation	N/A
vsubbu	r/i	r/i	subtract 8-bit bytes with unsigned saturation	N/A
vsubw	r/i	r/i	subtract 16-bit words	N/A
vmulb	r/i	r/i	multiply 8-bit bytes	N/A
vmulbs	r/i	r/i	multiply 8-bit bytes with signed saturation	N/A
vmulbu	r/i	r/i	multiply 8-bit bytes with unsigned saturation	N/A
vmulw	r/i	r/i	multiply 16-bit words	N/A
vdivb	r/i	r/i	divide 8-bit bytes	N/A
vdivw	r/i	r/i	divide 16-bit words	N/A
vunpkbs	r/i	r/i	unpack 4 signed 8-bit bytes into 16-bit ones in a 64-bit word	N/A
vunpkbu	r/i	r/i	unpack 4 unsigned 8-bit bytes into 16-bit ones in a 64-bit word	N/A

Chapter 4

Reference

4.1 Opcode Format

The following C structure is what the stock assembler uses for opcode output.

Opcode Structure

```
struct zpuop {
    unsigned flg     : 4;       // instruction flags */
    unsigned inst    : 7;       // numerical instruction ID
    unsigned sflg    : 5;       // INDIR, INDEX, IMMED, ADR, REG
    unsigned src     : 4;       // 4-bit source register ID
    unsigned dflg    : 5;       // INDIR, INDEX, IMMED, ADR, REG
    unsigned dest    : 4;       // 4-bit destination register
    unsigned argsz   : 3;       // operation size is 1 << (opsize + 1) bytes
    int32\_t  args[EMPTY];      // optional arguments
};
```

Notes

- **flg** is per-instruction flags
- **inst** is the instruction ID
- **sflg and dflg** are source and destination addressing bits
- **src and dest** are source and destination register IDs
- **argsz** is the size of arguments; not necessarily register size
- **args** contains 0, 1, or 2 32- or 64-byte addresses or values

4.2 Instruction Set

Operand Types

- **r** stands for register operand

- **i** stands for immediate operand value
- **a** stands for immediate direct address operand
- **p** stands for indirect address operand
- **n** stands for indexed address operand
- **m** stands for all of **a, i, and n**

Notes

- C language doesn't specify whether right shifts are arithmetic or logical
- Arithmetic right shift fills leftmost 'new' bits with the sign bit, logical shift fills with zero; left shifts are always fill rightmost bits with zero

Instructions

Below, I will list machine instructions and illustrate their relation to C.

Notes

- the **inb()** and other functions dealing with I/O are usually declared through **<sys/io.h>**

4.2.1 Cheat Sheet

C Operation	Instruction	Operands	Brief
	not	r dest	reverse all bits
&	and	r/i src, r dest	logical AND
|	or	r/i src, r dest	logical OR
ˆ	xor	r/i src, r dest	logical exclusive OR
«	shl	r/i cnt, r dest	shift left by count
»	shr	r/i cnt, r dest	arithmetic shift right
	shrl	r/i cnt, r dest	logical shift right
N/A	ror	r/i cnt, r dest	rotate right by count
N/A	rol	r/i cnt, r dest	rotate left by count
++	inc	r dest	increment by one
–	dec	r dest	decrement by one
+	add	r/i cnt, r dest	addition
-	sub	r/i cnt, r dest	subtraction
==, != etc.	cmp	r/i src, r dest	comparison; sets MSW-flags
*	mul	r/i src, r dest	multiplication
/	div	r/i src, r dest	division
%	mod	r/i src, r dest	modulus
==, !	bz	none	branch if zero
!=, (val)	bnz	none	branch if not zero
<	blt	none	branch if less than
<=	ble	none	branch if less than or equal
>	bgt	none	branch if greater than
>=	bge	none	branch if greater than or equal
N/A	bo	none	branch if overflow
N/A	bno	none	branch if no overflow
N/A	bc	none	branch if carry
N/A	bnc	none	branch if no carry
dest = *sp++	pop	r dest	pop from stack
–sp = src	push	r src	push onto stack
push(regs)	pusha	N/A	push all registers
dest = src	mov	r/i/m src, r/m dest	load/store longword (32-bit)
dest = src	movb	r/i/m src, r/m dest	load/store byte (8-bit)
dest = src	movw	r/i/m src, r/m dest	load/store word (16-bit)
dest = src	movq	r/i/m src, r/m dest	load/store word (64-bit)
N/A	jmp	r/m dest	continue execution at dest
N/A	call	a/p dest	call subroutine
N/A	enter	none	subroutine prologue
N/A	leave	none	subroutine epilogue
N/A	ret	none	return from subroutine
N/A	lmsw	r/i dest	load machine status word
N/A	smsw	r/i src	store machine status word

Part III

The Assembler

Chapter 5

Zen Assembly

5.1 Syntax

AT&T Syntax

We use so-called AT&T-syntax assembly. Perhaps the most notorious difference from Intel-syntax is the operand order; AT&T lists the source operand first, destination second, whereas Intel syntax does it vice versa.

Symbol Names

Label names must start with an underscore or a letter; after that, the name may contain underscores, letters, and digits. Label names end with a ':', so like

```
value: .long 0xb4b5b6b7
```

would declare a longword value at the address of **value**.

Instructions

The instruction operand order is source first, then destination. For example,

```
mov 8(%r0), val
```

would store the value from address **r0 + 8** to the address of the label **val**.

Operands

Register operand names are prefixed with a '**%**. Immediate constants and direct addresses are prefixed with a ́textbf$'. Label addresses are refered to as their names without prefixes.

The assembler supports simple preprocessing (of constant-value expressions), so it is possible to do things such as

```
.define FLAG1 0x01
.define FLAG2 0x02

mov $(FLAG1| FLAG2), %r1
```

Registers

Register names are prefixed with '%'; there are 16 registers r0..r15. For example,

```
add %r0, %r1
```

would add the longword in r0 to r1.

Direct Addressing

Direct addressing takes the syntax

```
mov val, %r0
```

which moves the longword at **address val** into r0.

Indexed Addressing

Indexed addressing takes the syntax

```
mov 4(%r0), %r1
```

where 4 is an integral constant offset and r0 is a register name. In short, this would store the value at the address **r0 + 4** into r1.

Indirect Addressing

Indirect addresses are indicated with a **'*'**, so

```
mov *%r0, %r1
```

would store the value from the **address in the register r0** into register r1, whereas

```
mov *val, %r0
```

would move the value **pointed to by val** into r0.

Note that the first example above was functionally equivalent with

```
mov (%r0), %r1
```

Immediate Addressing

Immediate addressing takes the syntax

```
mov $str, %r0
```

which would store the **address of str** into r0.

5.2 Assembler Directives

sectionInput Directives

5.2.1 .include

The **.include** directive takes the syntax

```
.include <file.asm>
```

to insert file.asm into the translation stream verbatim.

5.2.2 .import

The **.import** directive takes the syntax

```
.import <file.asm>
```

or

```
.import <file.obj>
```

to import foreign assembly or object files into the stream. **Note** that only symbols declared with **.globl** will be made globally visible to avoid namespace pollution.

sectionLink Directives

5.2.3 .org

The **.org** directive takes a single argument and sets the linker location address to the given value.

5.2.4 .space

The **.space** directive takes a single argument and advances the link location address by the given value.

5.2.5 .align

The **.align** directive takes a single argument and aligns the next label, data, or instruction to a boundary of the given size.

5.2.6 .globl

The **.globl** directive takes one or several symbol names arguments and declares the symbols to have global visibility (linkage).

sectionData Directives

5.2.7 .long

.long takes any number of arguments and declares in-memory 32-bit entities.

5.2.8 .byte

.byte takes any number of arguments and declares in-memory 8-bit entities.

5.2.9 .short

.short takes any number of arguments and declares in-memory 16-bit entities.

5.2.10 .asciz

.asciz takes a C-style string argument of characters enclosed within double quotes ('"'). Escape sequences '\n' (newline), '\t' (tabulator), and '\r' (carriage return) are supported.

sectionPreprocessor Directives

5.2.11 .define

.define lets one declare symbolic names for constant (numeric) values. For example, if you have