Zero Pseudo Machine

Volume One, Programmer's Guide, revision 0.0.1

Tuomo Petteri Venäläinen

June 24, 2017

Contents

Part I

Preface

Chapter 1

Notes

Brief

Zero Pseudo Machine, ZPM, is a software-based virtual machine. The machine is programmed in its own assembly dialect; the instruction set is reminiscent of many current RISC-like implementations.

Part II

Pseudo Machine

Chapter 2

Architecture

Notes

ZPM is an architecture with 32-bit machine words; room has been left in the implementation for 64-bit suppport.

ZPM words are little-endian (LSB) in byte-order.

There exists an instruction, **thr**, to start executing new threads from desired code locations in memory.

2.1 Memory Organization

Notes

System page size is 4096 bytes.

Address	Purpose	Brief
0	interrupt vector	interrupt handler descriptors
4096	keyboard buffer	keyboard input queue
8192	text segment	application program code (read-execute)
8192 + TEXTSIZE	data segment	program data (read-write)
DATA + DATASIZE	BSS segment	uninitialised data (runtime-allocated and zeroed)
MEMSIZE - 3.5 G	dynamic segment	free space for slab allocator
3.5 gigabytes	graphics	32-bit ARGB-format draw buffer

- the VM's 'physical' memory size is currently specified as **MEMSIZE**; this memory is mapped 1-to-1 to virtual address space
- thread stacks live at **MEMSIZE - thrid * THRSTKSIZE**, i.e. at top of 'physical' address space

Chapter 3

Instruction Set

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

3.1 Instruction Reference

3.1.1 Instruction Set

Operands

The table below lists operand types.

- **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 register (MSW). This is documented here on per-instruction basis.

- **z** stands for zero flag (ZF)
- **c** stands for carry flag (CF)
- **o** stands for overflow flag (OF)
- **s** stands for sign flag (SF)

TODO: stack/call conventions for certain instructions such as THR

3.1.2 Instruction Table

Mnemonic	Source	Destination	Brief	Flags
not	r	N/A	src = src;	z
and	r, i	r	dest = dest & src;	z
or	r, i	r	dest = dest | src;	N/A
xor	r, i	r	dest = dest ŝrc;	z
shl	r, i	r	dest = dest « src;	o, c
shr	r, i	r	dest = dest » src; (zero-fill)	z
sar	r, i	r	dest = dest » src; (sign-fill)	z
rol	r, i	r	dest = dest ROL src;	c
ror	r, i	r	dest = dest ROR src;	c
inc	r, i	N/A	src++;	o
dec	r, i	N/A	src–;	o, z
add	r, i	r	dest = dest + src;	o, z
sub	r, i	r	dest = dest - src;	s, z
cmp	r, i	r	compare two values and set flags	s, z
mul	r, i	r	dest = dest * src;	o, s, z
div	r, i	r	dest = dest / src;	s, z
jmp	r, i	branch to src	N/A	
bz	r, i	N/A	branch to src if (CF == 0)	N/A
bnz	r, i	N/A	branch to src if (CF != 0)	N/A
blt	r, i	N/A	branch to src if (SF != 0)	N/A
ble	r, i	N/A	branch to src if (SF != 0) || (ZF == 0)	N/A
bgt	r, i	N/A	branch to src if (SF != 0) && (ZF != 0)	N/A
bge	r, i	N/A	branch to src if (SF != 0) || (ZF == 0)	N/A
bo	r, i	N/A	branch to src if (OF != 0)	N/A
bno	r, i	N/A	branch to src if (OF == 0)	N/A
bc	r, i	N/A	branch to src if (CF != 0)	N/A
bnc	r, i	N/A	branch to src if (CF == 0)	N/A
pop	r	N/A	pop top of stack	N/A
popa	N/A	N/A	pop all registers from stack	N/A
push	r, i	N/A	push value on stack	N/A
pusha	N/A	N/A	push all register on stack	N/A
lda	r, i, m	r	load 32-bit longword	N/A
sta	r	r, i, m	store 32-bit longword	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
thr	m	N/A	start new thread at address	N/A
ltb	r, i	N/A	load thread-local storage base address	N/A
ldr	r, m	r	load special register	N/A
str	r	r, m	store special register	N/A
rst	m	N/A	reset machine	N/A
hlt	m	N/A	halt machine	N/A
inb	r, i	N/A	read 8-bit byte from input port	N/A
inw	r, i	N/A	read 16-bit word from input port	N/A
inl	r, i	N/A	read 32-bit longword from input port	N/A
outb	r, i	N/A	write 8-bit byte to input port	N/A
outw	r, i	N/A	write 16-bit word to input port	N/A
outl	r, i	N/A	write 32-bit longword to input port	N/A

3.1.3 I/O Port Map

I/O Address Space

The I/O address space maps 65,536 I/O ports to unsigned 16-bit address space.

Port #	Name	Default
0x0000	STDIN	keyboard input
0x0001	STDOUT	console output
0x0002	STDERR	error output

Chapter 4

Assembly

4.1 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

```
val:        .long 0xb4b5b6b7
```

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

Instructions

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

```
lda         8(%r0), %r1
```

would load the value from address **r0** + **8** to the register **r1**.

Operands

Register operand names are prefixed with a '%. Immediate constants and direct addresses are prefixed with a ́extbf$'. 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

lda         $(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

```
lda         val, %r0
```

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

Indexed Addressing

Indexed addressing takes the syntax

```
lda         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

```
lda         *%r0, %r1
```

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

```
lda         *val, %r0
```

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

Note that the first example above was functionally equivalent with

```
lda         (%r0), %r1
```

Immediate Addressing

Immediate addressing takes the syntax

```
lda         $str, %r0
```

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

4.2 Assembler Directives

4.3 Input Directives

4.3.1 .include

The **.include** directive takes the syntax

```
.include \<stdio.inc\>
```

to insert <**stdio.inc**> into the translation stream verbatim.

4.3.2 .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.

4.4 Link Directives

4.4.1 .org

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

4.4.2 .space

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

4.4.3 .align

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

4.4.4 .globl

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

4.5 Data Directives

4.5.1 .long

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

4.5.2 .byte

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

4.5.3 .short

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

4.5.4 .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.

4.5.5 Preprocessor Directives

4.6 .define

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

<**hook.def**>

```
.define STDIN  0
.define STDOUT 1
.define STDERR 2
```

4.7 Input and Output

The pseudo machine uses some predefined ports for keyboard and console I/O. The currently predefined ports are

Port	Use	Notes
0x00	keyboard input	interrupt-driven
0x01	console output	byte stream
0x02	error output	directed to console by default

4.8 Simple Program

The following code snippet prints the string ̈hello+̈ a newline to the console. Note that the string is saved using the standard C convention of NUL-character termination.

```
msg:        .asciz      "hello\n"

.align      4

_start:
        sta         $msg, %r0
        ldb         *%r0, %r1
        ldb         $0x01, %r2
        cmp         $0x00, %r1
        bz          done
loop:
        inc         %r0
        outb        %r1, %r2
        ldb        *%r0, %r1
        cmp         $0x00, %r1
        bnz         loop
done:
        hlt
```

4.9 Threads

The pseudo machine supports hardware threads with the **thr** instruction. It takes a single argument, which specifies the new execution start address; function arguments should be passed in registers.

4.10 Example Program

The following piece of code shows simple utilisation of threads.

```
.import <bzero.asm>

memzero:
        lda         $65536, %r0         // address
        lda         $4096, %r1          // length
        call        bzero
        hlt

_start:
        thr         $memzero
        hlt
```

4.11 Interrupts

Software- and CPU-generated interrupts are often refered to as **traps**. I call those and hardware-generated **interrupt requests** interrupts, collectively.

4.11.1 Interrupt Interface

The lowest page (4096 bytes) in virtual machine address space contains the **interrupt vector**, i.e. a table of interrupt handler addresses to trigger them.

Interrupt handler invokations only push the **program counter** and **old frame pointer**, so you need to reserve the registers you use manually. This is so interrupts could be as little overhead as possible to handle.

4.11.2 Keyboard Input

In order to read keyboard input without polling, we need to hook the **interrupt 0**. This is done in two code modules; an interrup handler as well as other support code.

I will illustrate the interrupt handler first.

4.11.3 Keyboard Interrupt Handler

TODO: example interrupt handler

4.11.3.1 Keyboard Support Code

TODO: queue keypresses in 16-bit values; 32-bit if full Unicode requested.