Wizard Pseudo Machine

Volume One, Programmer's Guide, revision 0.0.10

Tuomo Petteri Venäläinen

March 28, 2013

Contents

I Preface 5

1 Notes 7

2 Changes 9

II Pseudo Machine 11

3 Architecture 13
- 3.1 Memory Map . . . 14

4 Instruction Set 15
- 4.1 Instruction Reference . . . 15
- 4.2 Machine Operations . . . 16
- 4.3 Reference . . . 17
  - 4.3.1 Opcode Format . . . 17
  - 4.3.2 Instruction Set . . . 17

5 Assembly 21
- 5.1 Syntax . . . 21
- 5.2 Assembler Directives . . . 22
  - 5.2.1 Input Directives . . . 22
    - 5.2.1.1 .include . . . 22
    - 5.2.1.2 .import . . . 23
  - 5.2.2 Link Directives . . . 23
    - 5.2.2.1 .org . . . 23
    - 5.2.2.2 .space . . . 23
    - 5.2.2.3 .align . . . 23
    - 5.2.2.4 .globl . . . 23
  - 5.2.3 Data Directives . . . 23
    - 5.2.3.1 .long . . . 23
    - 5.2.3.2 .byte . . . 23
    - 5.2.3.3 .short . . . 24
    - 5.2.3.4 .asciz . . . 24
  - 5.2.4 Preprocessor Directives . . . 24
    - 5.2.4.1 .define . . . 24
- 5.3 Input and Output . . . 25
  - 5.3.1 Simple Program . . . 25

5.4 Threads 25
5.4.1 Example Program 26
5.5 Hooks 26
5.5.1 Pre-Defined Hooks 26
5.5.2 Hook Interface 26
5.5.3 Example Program 26
5.6 Interrupts 27
5.6.1 Break Points 27
5.6.2 Interrupt Interface 28
5.6.3 Keyboard Input 28
5.6.3.1 Keyboard Interrupt Handler 28
5.6.3.2 Keyboard Support Code 28

Part I

Preface

Chapter 1

Notes

Brief

Wizard Pseudo Machine, WPM, is a software-based virtual machine. It can be programmed in its own instruction set / assembly dialect, using AT&T syntax assembly. The incorporated assembler (**zas**) does not yet support pretranslated objects, but translates assembly code into bytecode on-the-fly. This bytecode is then handled to the pseudo machine (**wpm**) for execution.

The pseudo machine instruction set is designed to be a close match to basic operations of the C programming language.

Background and Future

Wizard Pseudo Machine project started as an attempt to create a tool for educational purposes. The machine is a virtual processor with custom instruction set that is close to existing ones; goals include making this instruction set simple, relatively complete, and useful for C language operations in case someone wants to create a C compiler or code generator for an existing one.

At the time of this writing, the virtual assembler is in good shape. The assembler provides not only .include to place other files into the stream verbatim, but also .import to 'link' with other assembly files with access to their global symbols.

The project is by no means complete yet. I wish for useful libraries for audio and graphics, a useful stock 'standard' library, and lots of other things to happen in the near future. :)

Chapter 2

Changes

0.0.10

- added section **Instruction Reference** along with CPU-flag management by some instructions
- added PDF-index and hyperlink-TOC

0.0.9

- grammatical and layout corrections

0.0.8

- added .asciz
- added new instruction **hook** to invoke high level services; this lets us run them with native code in the virtual machine

0.0.7

- adding more content
- cleaning up

0.0.6

- still fixing typoes
- added a new **Architecture** Chapter

0.0.5

- Added text on threading as well as a couple of assembly examples

0.0.4

- changed the title, made a couple of mistakes there; the book is now correctly called **Wizard Pseudo Machine**

0.0.3

- added new subsection Opcode Format

0.0.2

- reorganised some assembly sections; added .space, .long, .short, and .byte

0.0.1

- changed the assembler to use .include and .import instead of #include and #import
- changed the term 'argument' to 'operand' in many places in this booklet

Part II

Pseudo Machine

Chapter 3

Architecture

Wizard Pseudo Machine is an architecture with 32-bit machine words; however, room has been left in the implementation for 64-bit suppport.

The pseudo machine supports flat 4-gigabyte address space, of which some is mapped for interrupt vector, interrupt handlers, thread and interrupt stacks, graphics, and other purposes.

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

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

We follow the **von Neumann architecture**, so we basically have 3 abstractions; **CPU**, **memory**, and **input/output**.

The machine is a purposefully **RISC**-like **load-store** approach, meaning there is only a single load-store instruction (**mov**) that deals with memory-addressed operands.

3.1 Memory Map

Notes

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 memory size is currently specified as **MEMSIZE**
- thread stacks live at **MEMSIZE - thrid * THRSTKSIZE**, i.e. at top of [virtual] address space

Chapter 4

Instruction Set

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

4.1 Instruction Reference

Mnemonic	Operand #1	Operand #2	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
shra	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;	s, z
cmp	r, i	r	compare two values and set flags	s, z
mul	r, i	r	op2 = op2 * op1;	o, s, z
div	r, i	r	op2 = op2 / op1;	s, 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

TODO: finish the table above

4.2 Machine Operations

The following is a C-code snippet listing machine instructions and their IDs in opcodes.

```
#define OPNOT      0x01 // 2's complement
#define OPAND      0x02 // logical AND
#define OPOR       0x03 // logical OR
#define OPXOR      0x04 // logical exclusive OR
#define OPSHL      0x05 // shift left (fill with zero)
#define OPSHR      0x06 // arithmetic shift right (fill with sign)
#define OPSHRL     0x07 // logical shift right (fill with zero)
#define OPROR      0x08 // rotate right
#define OPROL      0x09 // rotate left
#define OPINC      0x0a // increment by one
```

```
#define OPDEC      0x0b // decrement by one
#define OPADD      0x0c // addition
#define OPSUB      0x0d // subtraction
#define OPCMP      0x0e // compare
#define OPMUL      0x0f // multiplication
#define OPDIV      0x10 // division
#define OPMOD      0x11 // modulus
#define OPBZ       0x12 // branch if zero
#define OPBNZ      0x13 // branch if not zero
#define OPBLT      0x14 // branch if less than
#define OPBLE      0x15 // branch if less than or equal to
#define OPBGT      0x16 // branch if greater than
#define OPBGE      0x17 // branch if greater than or equal to
#define OPBO       0x18 // branch if overflow
#define OPBNO      0x19 // branch if no overflow
#define OPBC       0x1a // branch if carry
#define OPBNC      0x1b // branch if no carry
#define OPPOP      0x1c // pop from stack
#define OPPUSH     0x1d // push to stack
#define OPMOV      0x1e // load/store 32-bit longword
#define OPMOVB     0x1f // load/store 8-bit byte
#define OPMOVW     0x20 // load/store 16-bit word
#define OPJMP      0x21 // jump to given address
#define OPCALL     0x22 // call subroutine
#define OPENTER    0x23 // subroutine prologue
#define OPLEAVE    0x24 // subroutine epilogue
#define OPRET      0x25 // return from subroutine
#define OPLMSW     0x26 // load machine status word
#define OPSMSW     0x27 // store machine status word
#define OPRESET    0x28 // reset into well-known state
#define OPNOP      0x29 // dummy operation
#define OPHLT      0x2a // halt execution
#define OPBRK      0x2b // breakpoint
#define OPTRAP     0x2c // trigger a trap (software interrupt)
#define OPIRET     0x2d // return from interrupt handler
#define OPTHR      0x2e // start new thread at given address
#define OPCMPSWAP  0x2f // atomic compare and swap
#define OPINB      0x30 // read 8-bit byte from port
#define OPOUTB     0x31 // write 8-bit byte to port
#define OPINW      0x32 // read 16-bit word
#define OPOUTW     0x33 // write 16-bit word
#define OPINL      0x34 // read 32-bit long
#define OPOUTL     0x35 // write 32-bit long
```

4.3 Reference

4.3.1 Opcode Format

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

Opcode Structure

```
struct wpmopcode {
    unsigned inst     : 8; // instruction ID
    unsigned unit     : 2; // unit ID
    unsigned arg1t    : 3; // argument #1 type
    unsigned arg2t    : 3;      // argument #2 type
    unsigned reg1     : 6; // register #1 ID + addressing flags
    unsigned reg2     : 6; // register #2 ID + addressing flags
    unsigned size     : 2;      // 1..3, shift count
    unsigned res      : 2;      // reserved
    int32_t  args[2];
} __attribute__ ((__packed__));
```

Notes

- **inst** is the instruction ID; 0 is invalid
- **unit** is a future unit ID; ALU, FPU, VPU (SIMD), GPU?
- **reg1 and reg2** are source and destination register IDs
- **operation size** can be calculated as **op->size « 2**
- **res**-bits are reserved for future extensions
- **args** contains 0, 1, or 2 32-byte addresses or values

4.3.2 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**>

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
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)
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
N/A	reset	none	reset machine
N/A	nop	none	no operation
N/A	hlt	none	halt machine
N/A	brk	none	breakpoint
N/A	trap	r/i src	trigger software interrupt
N/A	cli	none	disable interrupts
N/A	sti	none	enable interrupts
N/A	iret	none	return from interrupt handler
N/A	thr	r/i dest	start thread at dest
N/A	cmpswap	r/i src, m dest	atomic compare and swap
inb()	inb	r/i src	read byte (8-bit)
outb()	outb	r/i src, r/i dest	write byte (8-bit)
inw()	inw	r/i src	read word (16-bit)
outw()	outw	r/i src, r/i dest	write word (16-bit)
inl()	inl	r/i src	read longword (32-bit)
outl()	outl	r/i src, r/i dest	write longword (32-bit)

Chapter 5

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 t́extbf$'. 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

5.2.1 Input Directives

5.2.1.1 .include

The **.include** directive takes the syntax

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

to insert file.asm into the translation stream verbatim.

5.2.1.2 .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.

5.2.2 Link Directives

5.2.2.1 .org

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

5.2.2.2 .space

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

5.2.2.3 .align

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

5.2.2.4 .globl

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

5.2.3 Data Directives

5.2.3.1 .long

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

5.2.3.2 .byte

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

5.2.3.3 .short

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

5.2.3.4 .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.

5.2.4 Preprocessor Directives

5.2.4.1 .define

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

<hook.def>

```
.define PZERO  0
.define PALLOC 1
.define PFREE  2
```

you can then use the symbolic names like

```
.include <hook.def>
.import <bzero.asm>

memalloc:
        mov     $16384, %r0
        hook    $PALLOC
        mov     %r0, ptr
        ret

memzero:
        mov     ptr, %r0
        mov     $4096, %r1
        hook    $PZERO
        ret

memfree:
        mov     ptr, %r0
        hook    $PFREE
        ret

_start:
        call    memalloc
        call    memzero
        call    memfree
        hlt
```

```
ptr:     .long   0x00000000
```

5.3 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

5.3.1 Simple Program

The following code snippet prints the string ḧello+̈ 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:
        mov         $msg, %r0
        movb        *%r0, %r1
        mov         $0x01, %r2
        cmp         $0x00, %r1
        bz          done
loop:
        inc         %r0
        outb        %r1, %r2
        movb        *%r0, %r1
        cmp         $0x00, %r1
        bnz         loop
done:
        hlt
```

5.4 Threads

Wizard 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.

5.4.1 Example Program

The following piece of code shows simple utilisation of threads.

```
.import <bzero.asm>

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

_start:
        thr         $memzero
        hlt
```

5.5 Hooks

Hooks exist to provide system services. Hooks invoke native code in the virtual machine to do things such as manage memory and I/O.

5.5.1 Pre-Defined Hooks

Number	Name	Purpose
0x00	PZERO	zero given number of pages at given address
0x01	PALLOC	allocate given number of bytes from dynamic segment
0x01	PFREE	free region at given address

5.5.2 Hook Interface

Hook **parameters** are passed **in registers**. Hook **return value** is stored **in r0**. Here is the current interface definition.

- PZERO takes two arguments; destination address in r0, and region size (in bytes) in r1. PZERO returns nothing.
- PALLOC takes one argument; allocation size in r0. PALLOC returns allocated address or zero on failure.
- PFREE takes one argument; allocation address in r0. PFREE returns nothing.

5.5.3 Example Program

The following programs uses hooks to accomplish 3 tasks: allocate 16384 bytes of memory, zero it, and finally free it. In reality this alone is useless, but it serves as an example.

```
.import <bzero.asm>

alloc:
mov     $16384, %r0
hook    $1
mov     %r0, ptr
ret

zero:
mov     ptr, %r0
mov     $16384, %r1
hook    $0
ret

free:
mov ptr, %r0
hook $2
ret

_start:
call    alloc
call    zero
call    free
hlt

ptr:    .long    0x00000000

_foo:   .space   4096, 0xff
```

5.6 Interrupts

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

5.6.1 Break Points

The **brk** instruction triggers a breakpoint interrupt. The default action is to print a stack trace and continue execution.

The **use** of brk is simple; just use the zero-operand instruction in your assembly file:

```
brk ; trigger breakpoint
```

5.6.2 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.

5.6.3 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.

5.6.3.1 Keyboard Interrupt Handler

TODO: example interrupt handler

5.6.3.2 Keyboard Support Code

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