Valhalla Processor Unit 0 - V0

Volume One, Programmer's Guide, revision 0.0.1

Tuomo Petteri Venäläinen

April 10, 2018

Contents

Part I

Preface

Chapter 1

Notes

Brief

Valhalla Processor Unit 0, V0, 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.

I'm planning to experiment with FPGA-based processor design in the future.

Part II

Architecture

Chapter 2

Overview

Notes

V0 is an architecture with 32-bit machine words; room has been left in the implementation for future 64-bit suppport.

V0 words are little-endian (LSB) in byte-order.

2.1 Register File

Brief To keep life simple, V0 starts small. The register types in use are scalar, temporary, and floating-point ones.

2.1.1 General-Purpose Registers

Registers	Width	Purpose	Example
R0..R15	32-bit	Scalar Registers	Integer Addition
T0..T15	64-bit	Temporary Registers	Multiplication
F0..F15	64-bit	Floating-Point Registers	Floating-Point Operations

2.1.2 Special Registers

Name	Brief	Purpose	Writable
PC	Program Counter	Instruction Offset/Pointer	No
FP	Frame Pointer	Stack Frame for Functions	Yes
SP	Stack Pointer	Local [Variable] Stack	Yes
MSW	Machine Status Word	Processor Status and Features	Low 16 Bits (POP)

2.2 Memory Subsystem

Brief

As a 32-bit system, it's natural to support full 4-gigabyte 32-bit address space with some provisions; I/O ports are implemented as a separate address space, memory-mapped I/O has reserved regions, and so does the operating system space.

2.3 Input-Output

Brief Inspired by former Intel, AMD, and competitor PC-processors, we have a reasonable-size I/O-port address space of 65536 ports.

I will reserve ports for things such as keyboard input, text-mode screen output, as well as timers and other standard peripherals.

Chapter 3

Computation Units

3.1 Address Unit

Brief

Address calculations done by the processor.

3.2 Scalar Unit

Brief

It may help you to think of the scalar units as a simple calculator of sorts.

3.2.1 Scalar Adder

0x0 - carry/borrow

Mnemonic	Opcode	Brief	Notes
NOP	0x00	No Operation	Dummy or Delay Operation
INC	0x01	Increment By One	
DEC	0x02	Decrement By One	
ADD	0x04	Addition	SIGN-bit, SATU-bit
ADC	0x05	Addition with Carry	0x01 mean use carry
SUB	0x06	Subtraction	SIGN-bit, SATU-bit
SBB	0x07	Subtraction with Borrow	0x01 -> borrow

3.2.2 Scalar Shifter

0x01 - arithmetical shift

Mnemonic	Opcode	Brief
SHR	0x08	Shift Right Logical
SAR	0x09	Shift Right Arithmetical
SLL	0x0a	Shift Left Logical
SAL	0x0b	Shift Left Arithmetical

3.2.3 Scalar Logic

Mnemonic	Opcode	Brief
NOT	0x0c	Negation
AND	0x0d	Conjunction
XOR	0x0e	Exclusive Disjunction
OR	0x0f	Disjunction

3.2.4 Scalar Reciprocal

Brief

Unsigned and signed reciprocal are used for unsigned and signed division by "reverse multiplication", respectively.

0x01 - signed operation

Mnemonic	Opcode	Brief
CRU	0x10	Unsigned Reciprocal
CRS	0x11	Signed Reciprocal

3.2.5 Scalar Multiplication

Brief

Unsigned and signed multiplication or "division" by multiplying with a precalculated reciprocal value.

0x01 - signed operation

Mnemonic	Opcode	Brief
MLU	0x12	Unsigned Multiplication
MLS	0x13	Signed Multiplication

3.3 Memory Operations

3.3.1 Load-Store

Brief

LDR an STR instructions are used to load registers from and store registers into memory, respectively.

Mnemonic	Opcode	Brief
LDR	0x14	Load Register
MLS	0x15	Store Register

3.3.2 Stack

Brief

Stack operations are used to implement call conventions.

0x01 - many-bit

Mnemonic	Opcode	Brief
PSH	0x16	Push Register
PSM	0x17	Push Many Registers
POP	0x18	Pop Register
POM	0x19	Pop Many Registers

3.4 Control Flow

3.4.0.1 Flow Unit

Brief

The Flow Unit controls program control flow by the means of branches.

Mnemonic	Opcode	Brief
JMP	0x20	Branch Unconditionally; Absolute
JMP	0x21	Branch Unconditionally; Relative
BZ	0x22	Branch If Zero
BNZ	0x23	Branch if Non-Zero
BEQ	0x22	Synonymous with Branch if Zero (Equal)
BNE	0x23	Synonymous with Branch if Non-Zero (Not Equal)
BLT	0x24	Branch if Less Than
BLE	0x25	Branch if Less Than or Equal
BGT	0x26	Branch if Greater Than
BLE	0x27	Branch if Greater Than or Equal
BO	0x28	Branch if Overflow
BNO	0x29	Branch if No Overflow
BC	0x2a	Branch if Carry Set
BNC	0x2b	Branch if Carry Not Set
CSR	0x2c	Call Sub-Routine (narg)
BEG	0x2d	Begin Sub-Routine (nvar)
FIN	0x2e	Finish Sub-Routine (nvar)
RET	0x2f	Return From Sub-Routine (val)

3.5 Input-Output

Brief

Input-Output operations are used to configure and communicate with auxiliary devices.

low 3 bits - size shift count

Mnemonic	Opcode	Brief
RDB	0x30	Read 8-Bit Byte
RDH	0x31	Read 16-Bit Half-Word
RDW	0x32	Read 32-Bit Half-Word
RDL	0x33	Read 64-Bit Long-Word
WRB	0x34	Read 8-Bit Byte
WRH	0x35	Read 16-Bit Half-Word
WRW	0x36	Read 32-Bit Half-Word
WRL	0x37	Read 64-Bit Long-Word
IOR	0x39	Set or Query I/O Read Permission
IOW	0x3a	Set or Query I/O Write Permission

3.5.0.1 System Unit

3.6 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)
3G	system space	operating system
MEMSIZE to TLS	dynamic segment	free space for slab a locator
3.5G - NTHR * THRSTKSIZE	thread-local storage (TLS)	allocated on-demand
3.5 gigabytes	graphics memory mapped devices	draw buffer

Chapter 4

Instruction Set

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

4.1 Instruction Reference

4.1.1 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

4.1.2 Instruction Table

TOREDO

4.1.3 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 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

```
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.

5.2 Assembler Directives

5.3 Input Directives

5.3.1 .include

The **.include** directive takes the syntax

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

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

5.3.2 .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.

5.4 Link Directives

5.4.1 .org

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

5.4.2 .space

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

5.4.3 .align

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

5.4.4 .globl

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

5.5 Data Directives

5.5.1 .long

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

5.5.2 .byte

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

5.5.3 .short

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

5.5.4 .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.

5.5.5 Preprocessor Directives

5.6 .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
```

5.7 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

5.8 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
```

5.9 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.

5.10 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
```

5.11 Interrupts

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

5.11.1 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.

5.11.2 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.

5.11.3 Keyboard Interrupt Handler

TODO: example interrupt handler

5.11.3.1 Keyboard Support Code

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