CS 152B Final Report: FPGA-Accelerated `simdcore`

Raayan Dhar **Christopher Milan** **Gabriel Castro**

Abstract

We discuss the design and implementation of a Single Instruction, Multiple Data (`SIMD`) processor on the Basys3 FPGA. We motivate and present a complete hand-designed Instruction Set Architecture (ISA) for our processor, specialized `SIMD` instructions, practical implementation on the Basys3 FPGA, challenges faced, and applications of our work. Through this project, we demonstrate the programmability and power of FPGAs. Source code for all our software is open-source and can be found on Github.

1 Introduction

With the ever-growing demand for computational performance in areas such as digital signal processing, image processing, artificial intelligence, scientific computing, data encryption, error-correction, and others, the importance of efficient, parallelizable hardware designs has significantly increased. *Single Instruction, Multiple Data* (`SIMD`) architectures provide an efficient solution by executing the same instruction on multiple data points simultaneously, significantly improving computational throughput and energy efficiency.

Field Programmable Gate Arrays (FPGAs) have emerged as versatile platforms for implementing custom hardware solutions, combining programmability with parallelism and specialized hardware acceleration capabilities. Leveraging the FPGA's architecture, designers can create optimized computatiuonal module tailored specifically for performance-critical tasks.

In this work, we present the design, implementation, and practical details of a `SIMD` processor on the Basys3 FPGA development board. We introduce a carefully hand-designed Instruction Set Architecture (ISA) optimized for `SIMD` operations, inspired by the MIPS ISA, with a particular focus on efficient and practical *vector* instructions. We describe our approach to instruction selection, encoding, hardware implementation, and provide insight into the key design considerations and challenges encountered. Through this practical realization, we not only validate the effectiveness of our custom ISA but also highlight the versatility and power of FPGA-based processor design. Finally, we provide working examples of `SIMD`-accelerated programs which can run on our synthesized FPGA processor and discuss other potential applications scenarios and advantages of our approach.

2 Related Work

`SIMD` architectures have long been integral to enhancing computational efficiency in data-parallel tasks by enabling the simultaneous execution of a single instruction across multiple data points. In this section, we discuss notable implementations and designs of `SIMD` processors and principles.

A pioneering implementation of a `SIMD` instruction set is the Intel Streaming `SIMD` Extensions (SSE) (Diefendorff, 1999), introduced in 1999. SSE operates on single precision floating-point data using 128-bit vector registers labeled `XMM0` through `XMM7`.

Another implementation is the MIPS `SIMD` architecture (MSA) (Technologies, 2013), which extends the traditional MIPS instruction set to support 128-bit wide vector registers. MSA facilitates operations on various data types, including 8-, 16-, 32-, and 64-bit integers, as well as fixed-point and floating-point numbers.

In the x86 architecture domain, Intel's Advanced Vector Extensions (AVX) (Corporation, 2008) are a more modern implementation, introduced in 2008 and still used in recent microprocessor generations, such as AMD's Zen 5 (amd, 2024) (2024) and Intel's Sapphire Rapids (int, 2023) (2023). AVX introduced 256-bit `YMM` registers, allowing for parallel processing of multiple floating-point operations. Subsequent iterations, such as AVX2 and AVX-512 expanded these capabilities to include integer operations and wider 512-bit `ZMM` registers, respectively.

We implement our `SIMD` architecture to work with integers only. One important application of vectorized integer operations are parallelizable *encryption* algorithms. A notable example of explicit support for this application are the Intel Advanced Encryption Standard Instructions (AES-NI) (Gueron, 2010). A wider version of AES-NI, AVX-512 Vector AES instructions (VAES) is found in AVX-512, and includes instructions like `GF2P8AFFINEINVQB` (Galois Field Affine Transformation Inverse) (Intel Corporation, 2025a) and `VPCLMULQDQ` (Carry-Less Multiplication Quadword) (Intel Corporation, 2025b), both used in encryption algorithms.

3 Design, Methodology, & Implementation

3.1 Architecture Design

We implement a von-Neumann style computer architecture, with scalar registers, memory, an ALU, support for jump instructions, I/O, and the addition of *vector* registers. Our architecture utilizes 256 scalar registers which are 16 bits wide and 256 vector registers which are 128 bits wide, meaning we have 8 lanes. All arithmetic operations operate on the `int16` datatype.

3.2 Instruction Set Architecture

We hand-designed an ISA based on the MIPS architecture (Hennessy and Patterson, 2007) reduced instruction set computer (RISC) ISA. Unlike the x86 ISA, we use 32-bit fixed-width instruction lengths [1] for simplicity, and all instructions use the same bit-fields for efficient and easy decoding. We include 8 bits for the `opcode`, `arg1` specifies the destination register and `arg2` and `arg3` specify the source registers. We include all the instructions in [2], and highlight a select few here.

Table 1: Bit-field Breakdown of our 32-bit instructions

Bits	Field
0–7	`opcode`
8-15	`arg1`
16-24	`arg2`
25-32	`arg3`

3.3 `SIMD` Vector Instructions

We implement and include all basic arithmetic operations on vector registers, as well as a set of bitwise operations (`and, or, xor, not`). We also implement `veq` and `vgt`, which checks equality or greater than, respectively, and sets to 1 for each lane for which the condition was true. We also include two scalar-to-vector broadcast instructions. A detailed table showing instruction formats can be found in [2].

3.3.1 `scatter` & `gather` Instructions

Here we highlight our implemented **scatter** and **gather** instructions. These are "lane permutation" primitives, and allow an arbitrary re-ordering of the eight 16-bit lanes in the vector. We use a special "vperm" register. If we execute `mov vperm, vA` and execute `scatter` $l_0, l_1, l_2, l_4, l_5, l_6, l_7$, it has the effect of "scattering" the contents of the old `vperm` across its eight lanes in the order we specify.

$$\text{(new)}\texttt{vperm}[i] = \text{(old)}\texttt{vperm}[l_i], \quad i = 0, \ldots 7$$

If we instead execute `gather` $g_0, g_1, g_2, g_3, g_4, g_5, g_6, g_7$, we do the inverse of "scatter":

$$\text{(new)}\texttt{vperm}[g_i] = \text{(old)}\texttt{vperm}[i], \quad i = 0, \ldots, 7.$$

and we would execute `mov vB, vperm` to write the gathered result to `vB`. `scatter` and `gather` instructions commonly appear in `SIMD` instructions extensions since they tackle irregular or non-contiguous data layouts, and enable easy permutation of data across vector lanes.

3.3.2 I/O Instructions

To facilitate user interaction with our FPGA-based processor, we also introduce basic I/O instructions, which allow for communication with the processor over UART. This facility is made available through the `inl, inh, outl, outh` instructions, which can send or receive data over the serial interface.

3.4 Programming

The quintessential program for such a processor is, of course, a system monitor. Inspired by the Apple I's Woz Monitor (`WozMon`) (SB-Projects, n.d.), we implemented such a monitor. The key features of this program are facilitating examination of memory, directly editing memory, and running other programs. More precisely, the core features are:

- **Memory Examination:** Using the format `xxxx[.yyyy]`, where `xxxx` and `yyyy` are hexadecimal 16-bit numerals, the monitor will dump the contents of memory located between address `xxxx` and `yyyy`, inclusive, in a format reminiscent of the Unix `hexdump` utility. If `yyyy` is not specified, then just the 16-bit word located at `xxxx` will be shown.

- **Editing Memory:** Using the format `xxxx:A . . . Z`, where `xxxx` is as described above and `A . . . Z` is some sequence of 16-bit words encoded into 4-digit hexadecimal values each, this sequence will be written to memory, starting at the address specified in `xxxx`.

- **Running Programs:** Using the format `xxxxR`, where `xxxx` is as above, the program counter will "jump" to the address specified by `xxxx`, that is, start executing the program located in memory at the address specified by `xxxx`.

A listing of the assembly source for this hex monitor is available in the appendix A.3. This style of hex monitor exercises almost all the features of the processor, excluding, of course, the SIMD capability. Programs which exercise the SIMD hardware can be found in the later sections. It also serves as an excellent Basic Input Output System (eg. BIOS), which can be loaded into the initial memory of the processor as soon as it boots. In fact, this is exactly how our pre-programmed hardware, works: upon initial power-on or reset the of the device, this hex monitor is loaded, serving as a "bootloader" of sorts. Other programs can be injected into the initial RAM during pre-programming, or can be programmed into the device's memory at runtime using the memory editing facility, and then subsequently run through the execution facility.

3.5 Software Tooling

In order to facilitate writing programs, we found it useful to construct a few softwares that run on a supporting development device. While it would be possible to manually construct programs in hexadecimal, and load them onto the device through the hex monitor, these softwares allowed us to simplify the development process.

3.5.1 Assembler

It is well understood that a using an assembler has immense productivity implications: rather than memorizing opcodes and instruction formats, an assembler allows programmers to write programs just using the instruction mnemonics (ie. program in English, not binary). As such, we developed an assembler which translates from the comparatively easy to remember mnemonics to machine code. The Haskell Language source is available here. Other quality-of-life features present in industry-standard assemblers have also been implemented, such as comments, labels, and two assembler directives:

- `.db`, which allows the assembly source to encode aribtrary binary data (ie. in order to encode ASCII strings), and

- `.org`, which allows for setting the memory offset of the entire program.

The `.org` directive is especially important when linking multiple assembler sources together.

3.5.2 Linker

It is desirable to split source code into separate files, both to facilitate iterative compilation, but also to support namespacing for labels: by separating assembly source into different files, label names can be reused without fear of collisions. However, this causes an obvious conflict, in the sense that we must load exactly one binary into memory. As such, we also developed a linker tool, which "links" binary machine code files together. Rather than implement a custom object format, binaries in our system are simply flat binaries. As a result, the process for linking is dead-simple: simply bitwise OR the files together. The C Language source for this tool is here. A notable downside of flat binaries is the lack of symbol resolution: ie. if you wish to reference a label located in a separate source file, you must first calculate the offset, and then manually encode this offset into your source, a time consuming and brittle process. In the future, the assembler could be updated to output files in a format more amenable to symbol resolution, such as the Executable and Linkable Format (ELF) or the Common Object File Format (COFF), which could allow for a more versatile linker.

3.5.3 Simulator

In order to facilitate rapid prototyping, we also developed a simulator for our hardware, allowing us to compare the functionality of real hardware and simulation, and also develop programs on supplemental x86 and ARM machines. The initial simulator was written in the Python Language, source for which is available . While the Python simulator was useful at first, we soon found that in order to accurately simulate the behavior of the I/O instructions, it was necessary to write a C Language simulator. The source for the C Language simulator is available here, and can be compiled with a C99-compliant compiler on a POSIX-1.2008 compliant system.

3.5.4 Higher-Level Languages

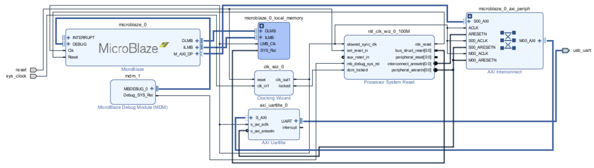
While writing in assembly language is much easier than hand-encoding machine instructions, writing programs in higher-level languages is still much prefered to assembly programming for most tasks. As such, a compiler for such a language is in development (with source available here). The design of this language is heavily inspired by B, a language developed by Ken Thompson and Dennis Ritchie at Bell Labs, prior to their work on C. In short, B is C without types (Ritchie, 1996). Since our hardware only supports two register sizes, and one of them simply being a vector form of the other register type, B is a natural choice to base our language semantics off.

3.6 Implementation

The main complexity of our project comes from our ISA design and surrounding details of making the processor work, rather than working with peripherals. Before going straight to hardware, we discussed and agreed on an ISA design as we already detailed in 3.2. However, before implementing on hardware, we verified our ISA design by writing a simulator in Python and an assembler in Haskell, so we could both run and examine (`objdump` capabilities) programs written in our ISA. We also implement a testing framework in Python to verify programs at scale.

3.6.1 Hardware Implementation on Basys 3

We originally implemented most of our implementation using Verilog (register files, control, vector and scalar ALUs, etc). However, difficulties and challenges with combining modules together, the overwhelming numbers of tests to run, and the upcoming deadlines pushed us to use the MicroBlaze softcore (block diagram below) instead, and writing our implementation in `C`.



Thanks to the careful ISA design, implementing our core in `C` was more straightforward and we were able to successfully implement all our instructions. We include example testing programs in the Appendix under A.1.

4 Results, Examples, & Applications

Our implementation successfully demonstrates the viability and efficiency of FPGA-bsaed `SIMD` processors. Our hand-designed ISA can effectively accelerate computation for applications that can exploit a data-parallel model.

4.1 Examples & Applications

We showcase practical applications of our `SIMD` processor by including several representative programs. We focus on numerical applications, as the `SIMD` programming model is highly applicable in these areas. A demonstrative program is the `uppercase.s` [A.1.9] program, which cleanly converts strings to uppercase over vector registers, using instructions like `vgt` for efficient comparison. Another demonstrative program is `reverse.s` [A.1.6], which uses the `scatter` instruction, capable of permuting the vector lanes, in order to reverse the elements in a vector register. We also include some other interesting programs, common in scientific and machine learning applications:

- `inner_product.s` [A.1.2], which makes use of vector registers and the `gather` instruction to calculate an inner product between two vector registers
- `matmul.s` [A.1.3], which operates over a matrix in memory and efficiently calculates elements of the output matrix using vector instructions
- `mse.s` [A.1.4], which calculate the mean squared error between two vectors.
- `relu.s` [A.1.5], which calculates the $\mathrm{ReLU}(v)$ activation function over a vector efficiently using the `vgt` instruction.
- And others [A.1.7, A.1.8], which can be found in our appendix.

4.2 Results

The result of our project is a complete, end-to-end SIMD processor stack, encompassing a hand-designed ISA, software tooling (compiler, assembler, simulator, and testing framework), and a hardware implementation on the Basys3 FPGA. Our Python-based simulator and Haskell assembler enabled us to verify and iterate on the ISA before committing to hardware. Once correctness was established, we synthesized our design onto a MicroBlaze softcore running on the Basys3 FPGA board and wrote programs to demonstrate the applicability and expressiveness of our ISA. Our onboard UART interface (Wozmon-style monitor) allows easy verification of program I/O and memory inspection and writing.

In summary, we demonstrate that a simple, fixed-width ISA with eight-lane `int16` vecotrs can be fully realized on a low-end FPGA, proving that our ISA is flexible and sustainable across board complexity and resources.

5 Conclusion

Our project demonstrates the applicability and potential of FPGA-accelerated SIMD architectures to improve computational performance. We introduce a hand-designed ISA, inspired by MIPS, to show that a simple set of fixed-width instructions are sufficient for a wide range of integer-based SIMD operations, such as encryption or other parallel arithmetic tasks. Our implementation on hardware (Basys3) highlights the FPGA's flexibility and efficiency.

Future work could involve extending our ISA to support floating-point operations and implement additional vector instructions optimizations to support more numerical computing applications, such as those in physics, machine learning, signal processing, etc. Additionally, it could involve creating a separate coprocessor to perform the vector operations, extracting more performance from the FPGA. However, we leave these additions to a future work.

References

2023. Sapphire rapids. https://en.wikipedia.org/wiki/Sapphire_Rapids. Accessed 2025-06-02.

2024. Zen 5. https://en.wikipedia.org/wiki/Zen_5. Accessed 2025-06-02.

Intel Corporation. 2008. *Intel Advanced Vector Extensions Programming Reference*. Document No. 319433-002 (March 2008).

Keith Diefendorff. 1999. Pentiumiii = pentiumii + sse: Internet sse architecture boosts multimedia performance. *Microprocessor Report*.

Shay Gueron. 2010. Intel advanced encryption standard (aes) instruction set white paper. https://www.intel.com/content/dam/doc/white-paper/advanced-encryption-standard-new-instructions-set-paper.pdf. Intel White Paper.

John L. Hennessy and David A. Patterson. 2007. *Computer Organization and Design: The Hardware/Software Interface*, third edition. Morgan Kaufmann.

Intel Corporation. 2025a. Gf2p8affineinvqb—galois field affine transformation inverse. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A–L (GF2P8AFFINEINVQB entry). Accessed 2025-06-06.

Intel Corporation. 2025b. Vpclmulqdq—carry-less multiplication quadword. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2C: Instruction Set Reference, V (VPCLMULQDQ entry). Accessed 2025-06-06.

Dennis M. Ritchie. 1996. The development of the c language. *History of programming languages-II*.

SB-Projects. n.d. Apple 1 woz monitor. https://www.sbprojects.net/projects/apple1/wozmon.php. Accessed 2025-06-04.

MIPS Technologies. 2013. Mips simd architecture, revision 1.03. Technical report, MIPS Technologies. White Paper.

A Appendix

A.1 Example Programs

A.1.1 Hadamard Product

Element-wise multiplication of two vectors.

```
mov s0, 0x8000;
mov v0, [s0+0];
mov v1, [s0+0x20];
mul v2, v0, v1;
mov s0, 0x8040;
mov [s0+0], v2;
halt;
```

A.1.2 Inner Product

Calculates the inner product between two vectors.

```
mov s0, 0x8000;
mov v0, [s0+0];
mov v1, [s0+0x20];
mul v2, v0, v1;
mov vperm, v2;
gather 0, 1, 2, 3, 4, 5, 6, 7;
mov v3, vperm;
mov vperm, v2;
gather 4, 5, 6, 7, 0, 1, 2, 3;
mov v4, vperm;
add v2, v3, v4;
mov vperm, v2;
gather 0, 1, 2, 3, 0, 1, 2, 3;
mov v3, vperm;
mov vperm, v2;
gather 2, 3, 0, 1, 2, 3, 0, 1;
mov v4, vperm;
add v2, v3, v4;
mov vperm, v2;
gather 0, 1, 0, 1, 0, 1, 0, 1;
mov v3, vperm;
mov vperm, v2;
gather 1, 0, 1, 0, 1, 0, 1, 0;
mov v4, vperm;
add v2, v3, v4;
mov s1, 0x8040;
mov [s1+0], v2;
halt;
```

A.1.3 Matrix Multiplication

Performs an 8×8 matrix multiplication using vector registers.

```
mov s0, 0; ; i=0
mov s1, 8; ; m=8
mov s31, 8; ; n=8
mov s2, 1; ; increment
OUTER_LOOP:
cmp s0, s1;
jge END ;
; LOAD a row of A
mov s4, 0x8000;
mov s5, 16;
mul s6, s5, s0; ; i16 (assuming 8 elements)
add s25, s4, s6; row = base_addr + i16
mov v0, [s25+0];
mov s3, 0; ; j=0
INNER_LOOP:
cmp s3, s31;
jge NEXT ;
; LOAD a column of B
mov s7, 0x8200; ; base
mov s8, 2;
mul s9, s3, s8; ; j * 2
add s7, s7, s9; ; base_addr + j2 (get column)
add s10, s7, s9; ; (base_addr + j2) + j*2 (get elements in column)
mov s11, [s7+0]; ;
mov s12, [s7+16];
mov s13, [s7+32];
mov s14, [s7+48];
mov s15, [s7+64];
mov s16, [s7+80];
mov s17, [s7+96];
mov s18, [s7+112];
mov v1, s11, 0x01;
mov v1, s12, 0x02;
mov v1, s13, 0x04;
mov v1, s14, 0x08;
mov v1, s15, 0x10;
mov v1, s16, 0x20;
mov v1, s17, 0x40;
mov v1, s18, 0x80;
; Compute inner product
mul v2, v0, v1;
mov vperm, v2;
gather 4, 5, 6, 7, 0, 1, 2, 3;
mov v3, vperm;
add v2, v2, v3;
mov vperm, v2;
gather 2, 3, 0, 1, 2, 3, 0, 1;
mov v3, vperm;
add v2, v2, v3;
mov vperm, v2;
gather 1, 0, 1, 0, 1, 0, 1, 0;
mov v3, vperm;
add v2, v2, v3;
; scratchpad to extract single element from vector
mov s16, 0x8800;
mov [s16+0], v2;
mov s17, [s16+0];
; s17 contains C[i][j]
mov s16, 0x8400; ; base address for C matrix
mov s19, 16; ; row size (8 elements * 2 bytes each)
mul s18, s0, s19; ; i * 16 (no adjustment needed now)
mov s20, 2; ; element size
mul s21, s3, s20; ; j * 2
add s18, s18, s21; ; i16 + j2
add s22, s16, s18; ; base_addr + i16 + j2
mov [s22+0], s17;
add s3, s3, s2; ; j++
j INNER_LOOP ;
NEXT:
add s0, s0, s2; ; i++
j OUTER_LOOP ;
END:
halt;
```

A.1.4 Mean Squared Error

Calculates the mean squared error, common in machine learning, operating on vector registers.

```
mov s0, 0x8000;
mov v0, [s0+0];
mov v1, [s0+0x20];
neg v5, v1;
add v2, v0, v5;
mul v2, v2, v2;
mov vperm, v2;
gather 4, 5, 6, 7, 0, 1, 2, 3;
mov v4, vperm;
add v2, v2, v4;
mov vperm, v2;
gather 2, 3, 0, 1, 2, 3, 0, 1;
mov v4, vperm;
add v2, v2, v4;
mov vperm, v2;
gather 1, 0, 1, 0, 1, 0, 1, 0;
mov v4, vperm;
add v2, v2, v4;
mov s5, 8;
mov v10, s5;
div v9, v2, v10;
mov s5, 0x8040;
mov [s5+0], v9;
halt;
```

A.1.5 ReLU Activation

Implements the ReLU activation function commonly used in neural networks, operating over our vector registers.

```
mov s0, 0x8000;
mov v0, [s0+0];
mov s0, 0;
mov v1, s0, 0xFFFF;
vgt v2, v0, v1;
and v3, v0, v2;
mov s1, 0x8020;
mov [s1+0], v3;
halt;
```

A.1.6 Reverse

Permutes the values in a vector register in reverse order.

```
mov s0, 0x8000;
mov v0, [s0+0];
mov vperm, v0;
scatter 7, 6, 5, 4, 3, 2, 1, 0;
mov v2, vperm;
mov s0, 0x8020;
mov [s0+0], v2;
halt;
```

A.1.7 Transpose

Permutes values at specified indices within a vector register.

```
mov s0, 0x8000;
mov v0, [s0+0];
mov vperm, v0;
gather 0, 4, 1, 5, 2, 6, 3, 7;
mov v2, vperm;
mov s0, 0x8040;
mov [s0+0], v2;
halt;
```

A.1.8 Vector Addition

Adds two vectors together.

```
mov s0, 0x8000;
mov v0, [s0+0];
mov v1, [s0+0x20];
add v2, v0, v1;
mov s0, 0x8040;
mov [s0+0], v2;
halt;
```

A.1.9 Uppercase

Convert lowercase characters to uppercase in a vectorized fashion.

```
;; uppercase.s - converts strings to uppercase (SIMD demo)
.org 0x1000
START:
  mov s30, 0x3e20 ; "> "
  outh s30 ;
  outl s30 ;
  mov s0, 0x8000 ; buffer address
  mov s1, 0 ; input byte
READC:
  inl s1 ;
  jeq READC ; wait for input
  mov [s0 + 0], s1 ;
  mov s30, 2 ;
  add s0, s0, s30 ;
  cmp s1, 0xa ; newline
  jeq GO ;
  j READC ;
GO:
  mov s1, 0x8000 ; current address
LOOP:
  cmp s0, s1 ; are we done?
  jeq START ;
  mov v0, [s1 + 0] ; read 8 bytes
  mov s2, 0x60 ; 'a'-1
  mov v1, s2 ; broadcast
  mov s2, 0x7b ; 'z'+1
  mov v2, s2 ;
  vgt v1, v0, v1 ; v1 = v0 >= 'a'
  vgt v2, v2, v0 ; v2 = v0 <= 'z'
  and v1, v1, v2 ; v1 in [a-z]
  mov s2, 0xffe0 ; -0x20
  mov v2, s2 ;
  and v1, v1, v2 ;
  add v0, v0, v1 ;
  mov [s1 + 0], v0 ;
  mov s3, 0 ; byte counter
OUT:
  mov s2, [s1 + 0] ;
  outl s2 ;
  mov s2, 1 ;
  add s1, s1, s2 ;
  add s3, s3, s2 ;
  cmp s0, s1 ;
  jeq START ;
  cmp s3, 8 ;
  jne OUT ;
  j LOOP ;
```

A.2 ISA Specifications

Table 2: Instruction encodings

op (5)	ctrl (3)	arg1	arg2	arg3	Operation
00000	000	s$1	s$2	—	mov s$1, s$2
00000	001	s$1	s$2	imm8	mov s$1, [s$2 + imm8]
00000	010	s$1	imm8	s$2	mov [s$1 + imm8], s$2
00000	011	s$1	hi(imm16)	lo(imm16)	mov s$1, imm16
00000	100	v$1	v$2	—	mov v$1, v$2
00000	101	v$1	s$2	imm8	mov v$1, [s$2 + imm8]
00000	110	s$1	imm8	v$2	mov [s$1 + imm8], v$2
00000	111	v$1	s$2	imm8	mov v$1, s$2, imm8
00001	000	s$1	s$2	s$3	add s$1, s$2, s$3
00001	001	s$1	s$2	s$3	mul s$1, s$2, s$3
00001	010	s$1	s$2	—	neg s$1, s$2
00001	011	s$1	s$2	s$3	div s$1, s$2, s$3
00001	100	s$1	s$2	s$3	and s$1, s$2, s$3
00001	101	s$1	s$2	s$3	or s$1, s$2, s$3
00001	110	s$1	s$2	s$3	xor s$1, s$2, s$3
00001	111	s$1	s$2	—	not s$1, s$2
00010	000	v$1	v$2	v$3	add v$1, v$2, v$3
00010	001	v$1	v$2	v$3	mul v$1, v$2, v$3
00010	010	v$1	v$2	—	neg v$1, v$2
00010	011	v$1	v$2	v$3	div v$1, v$2, v$3
00010	100	v$1	v$2	v$3	and v$1, v$2, v$3
00010	101	v$1	v$2	v$3	or v$1, v$2, v$3
00010	110	v$1	v$2	v$3	xor v$1, v$2, v$3
00010	111	v$1	v$2	—	not v$1, v$2
00011	000	v$1	v$2	v$3	veq v$1, v$2, v$3
00011	001	v$1	v$2	v$3	vgt v$1, v$2, v$3
00011	010	lane-bits	lane-bits	lane-bits	scatter <8 lanes>
00011	011	lane-bits	lane-bits	lane-bits	gather <8 lanes>
00011	100	v$	—	—	mov vperm, v$
00011	101	v$	—	—	mov v$, vperm
00100	000	hi(imm16)	lo(imm16)	—	j imm16
00100	001	hi(imm16)	lo(imm16)	—	jeq imm16
00100	010	hi(imm16)	lo(imm16)	—	jne imm16
00100	011	hi(imm16)	lo(imm16)	—	jge imm16
00100	100	hi(imm16)	lo(imm16)	—	jle imm16
00100	101	hi(imm16)	lo(imm16)	—	jgt imm16
00100	110	hi(imm16)	lo(imm16)	—	jlt imm16
00100	111	s$	—	—	jr s$
00101	000	s$1	s$2	—	cmp s$1, s$2
00101	001	s$1	hi(imm16)	lo(imm16)	cmp s$1, imm16
00101	010	hi(imm16)	lo(imm16)	s$	cmp imm16, s$
00101	011	imm8	—	—	set imm8
00110	000	s$	—	—	inl s$
00110	001	s$	—	—	inh s$
00110	010	s$	—	—	outl s$
00110	011	s$	—	—	outh s$
00111	000	—	—	—	halt

A.3 Hex Monitor Assembler Source

```
;; wozmon.s - a "bootloader" and memory editor
;; heavily inspired by the original wozmon

  mov s10, 0xff00 ; offset of first address
  mov s30, 0x3e20 ; "> "
  outh s30 ;
  outl s30 ;
INPUT:
;; s0 - offset into buffer
;; s1 - input byte
;; s2 - nibble selector
;; s3 - in progress word
  mov s0, 0 ; counter
  mov s30, 0 ;
  mov [s10 + 0], s30 ; reset
  mov [s10 + 2], s30 ;
INPUT_CONT:
  mov s1, 0 ; input
  mov s2, 0 ; which nibble (0, 1, 2, 3)
  mov s3, 0 ; in progress byte value
INPUT_LOOP:
  inl s1 ;
  jeq INPUT_LOOP ;
  cmp s1, 0xa ; newline
  jeq GO ; process command
  cmp s1, 0x3a ; ':'
  jeq EDIT ;
  cmp s1, 0x52 ; 'R'
  jne DONT_R ;
  mov s30, [s10 + 0] ;
  jr s30 ;
DONT_R:
  cmp s1, 0x2e ; '.'
  jne OK ;
DOT:
  inl s1 ;
  jeq DOT ;
OK:
  cmp s1, 0x39 ; '9'
  jgt HEX ;
  mov s30, 0xffd0 ; -'0'
  add s1, s1, s30 ;
  j WRITE ;
HEX:
  mov s30, 0xffc9 ; -'A'
  add s1, s1, s30 ;
WRITE:
  cmp s2, 0 ;
  jeq WRITE_0 ;
  cmp s2, 2 ;
  jlt WRITE_1 ;
  jeq WRITE_2 ;
  add s3, s3, s1 ;
  add s30, s10, s0 ; compute offset
  mov [s30 + 0], s3 ;
  mov s30, 2 ;
  add s0, s0, s30 ;
  j INPUT_CONT ;
WRITE_0:
  mov s30, 0x1000 ; 2 ** 12
  mul s3, s1, s30 ; << 12
  mov s2, 1 ;
  j INPUT_LOOP ;
WRITE_1:
  mov s30, 0x100 ; 2 ** 8
  mul s31, s1, s30 ; << 8
  add s3, s3, s31 ;
  mov s2, 2 ;
  j INPUT_LOOP ;
WRITE_2:
  mov s30, 0x10 ; 2 ** 4
  mul s31, s1, s30 ; << 4
  add s3, s3, s31 ;
  mov s2, 3 ;
  j INPUT_LOOP ;

GO:
;; s0 - data to print
;; s1 - mode (0 is print address, 1 is dereferencing)
;; s2 - offset to inspect
  mov s0, [s10 + 0] ; first we print address
  mov s1, 0 ;
PRINT:
  mov s20, 0 ; nibble index
GO_LOOP:
  cmp s20, 0 ;
  jeq O0 ;
  cmp s20, 2 ;
  jlt O1 ;
  jeq O2 ;
  cmp s20, 4 ;
  jeq PRINT_DONE ;
  mov s31, s0 ;
  j O ;
O0:
  mov s30, 0x1000 ; 2 ** 12
  div s31, s0, s30 ; first nibble
  j O ;
O1:
  mov s30, 0x100 ; 2 ** 8
  div s31, s0, s30 ; second nibble
  j O ;
O2:
  mov s30, 0x10 ; 2 ** 4
  div s31, s0, s30 ; third nibble
  j O ;
O:
  mov s30, 0xF ;
  and s31, s31, s30 ;
  cmp s31, 0x9 ;
  jgt WALPH ;
  mov s30, 0x30 ; '0'
  j W ;
WALPH:
  mov s30, 0x37 ; 'A' - 0xa
W:
  add s31, s31, s30 ;
  outl s31 ;
  mov s30, 1 ;
  add s20, s20, s30 ;
  j GO_LOOP ;
PRINT_DONE:
  cmp s1, 0 ;
  jne DEREF ;
  mov s30, 0x3a20 ; ": "
  outh s30 ;
  outl s30 ;
  mov s1, 1 ;
  mov s2, s0 ;
  mov s0, [s2 + 0] ;
  j PRINT ;
DEREF:
  mov s30, [s10 + 2] ;
  cmp s30, 0 ; if dst is null, we don't print a range
  jeq END ;
  cmp s2, s30 ;
  jge END ;
  mov s31, 2 ;
  add s2, s2, s31 ;
  mov s0, [s2 + 0] ;
  mov s30, 0x20 ;
  outl s30 ;
  j PRINT ;
END:
  mov s30, 0x0a ; "\n"
  outl s30 ;
  j 0 ;

  mov s30, 0x45 ;

EDIT:
;; s0 - current address to write to
;; s1 - nibble idx
;; s2 - nibble
;; s3 - word to write
  mov s0, [s10 + 0] ;
EDIT_LOOP:
  mov s1, 0 ;
  mov s2, 0 ;
  mov s3, 0 ;
ECONT:
  inl s2 ;
  jeq ECONT ;
  cmp s2, 0xa ; newline
  jeq 0 ;
  cmp s2, 0x39 ; '9'
  jgt EH ;
  mov s31, 0xffd0 ; -'0'
  j EW ;
EH:
  mov s31, 0xffc9 ; 0xA-'A'
```

```
EW:
  add s2, s2, s31 ;
  cmp s1, 0 ;
  jeq EW0 ;
  cmp s1, 2 ;
  jlt EW1 ;
  jeq EW2 ;
  add s3, s3, s2 ;
  mov [s0 + 0], s3 ;
  mov s31, 2 ;
  add s0, s0, s31 ;
  j EDIT_LOOP ;
EW0:
  mov s30, 0x1000 ;
  mul s3, s2, s30 ;
  mov s1, 1 ;
  j ECONT ;
EW1:
  mov s30, 0x100 ;
  mul s31, s2, s30 ;
  add s3, s3, s31 ;
  mov s1, 2 ;
  j ECONT ;
EW2:
  mov s30, 0x10 ;
  mul s31, s2, s30 ;
  add s3, s3, s31 ;
  mov s1, 3 ;
  j ECONT ;
DUMB:
  outl s30 ;
  j DUMB ;
```