S16X4 Data Sheet

Samuel A. Falvo II

2012-Aug-20

- 64KiB Addressing Space
- SS0 Load/Store Stack Architecture
- 8-Bit and 16-Bit Memory Accessors
- 16-Bit Internal Architecture
- All Instructions Execute in 1 Clock Cycle
- Wishbone Bus Compatible
- Easy to Program

1 Introduction

The Steamer16 architecture, first defined by Myron Plichota in December 1999 (see Appendix A), specifies an exceptionally small, relatively high performance microprocessor with good qualities for deep-embedded applications and easy synthesis on small programmable logic devices. Originally consisting of 9 instructions and four opcodes per instruction word, the design was later revised to include only 8 instructions, allowing for five opcodes per 16-bit word. The S16X4 microprocessor is a specific variant of the original Steamer16 concept. The processor design caters to ease of programming and easy integration with other components in an FPGA development platform.

1.1 Addressing

The S16X4, without the use of external hardware, supports a flat, unified address space. Unlike most processors that provide at least two addressing modes, the S16X4 only offers one addressing mode: stack indirect. All memory accessors take their effective address from the top of the parameter stack.

1.2 Data Types

The S16X4 supports two data types: bytes and words. The most basic addressing unit on the S16X4 is an octet (8-bit) byte. All addresses consist of 16 bits, thus limiting the S16X4 to 65536 bytes of directly addressible space.

Words consist of two bytes, back to back, placed on an even address boundary. The least-significant byte appears at the lower address; thus making the S16X4 a little-endian machine. When addressing 16-bit words, the S16X4 currently ignores bit 0 of an address. Hence, the two pointers $AAAA and $AAAB both refer to the same word of memory, even though they refer to two different bytes within the same word. For compatibility with later generations of processors, all pointers to 16-bit entities should have bit 0 clear.

2 Internal Architecture

Figure 1 illustrates the block diagram of the S16X4 processor.

2.1 Parameter Stack

The parameter stack provides the working storage and the means for expression evaluation. The S16X4 can hold up to three words of data at any given time, arranged as a stack. The three words are labelled X, Y, and Z, with Z representing the top of the stack. Presently, only one instruction, LIT, can push data onto the stack. All other instructions either replace Z directly or consume data from the stack on an as-needed basis. When popping values off the stack, the processor retains the current value for X. In the example below, values filled in from popping the stack appear in *italics*, while new or computed values appear in **bold**.

Instruction	X	Y	Z
LIT $1111	-	-	**$1111**
LIT $2222	-	$1111	**$2222**
LIT $5555	$1111	$2222	**$5555**
ADD	$1111	*$1111*	**$7777**
SWM	$1111	*$1111*	*$1111*

While having only three stack elements might seem constraining to someone familiar with either Forth or Java, it turns out you can do anything that (at least) an accumulator- or register/memory-architecture CPU can do. For example, suppose you're writing a program that needs to store a value at a location Y bytes beyond the start of a buffer whose pointer can be found in an array of pointers starting at address $\$44 + S$. This is a genuinely complex addressing mode by anyone's measure. Here's an example S16X4 program that computes this complex effective address:[1]

[1]The corresponding W65C816 microprocessor instruction for this operation is STA ($44,S),Y.

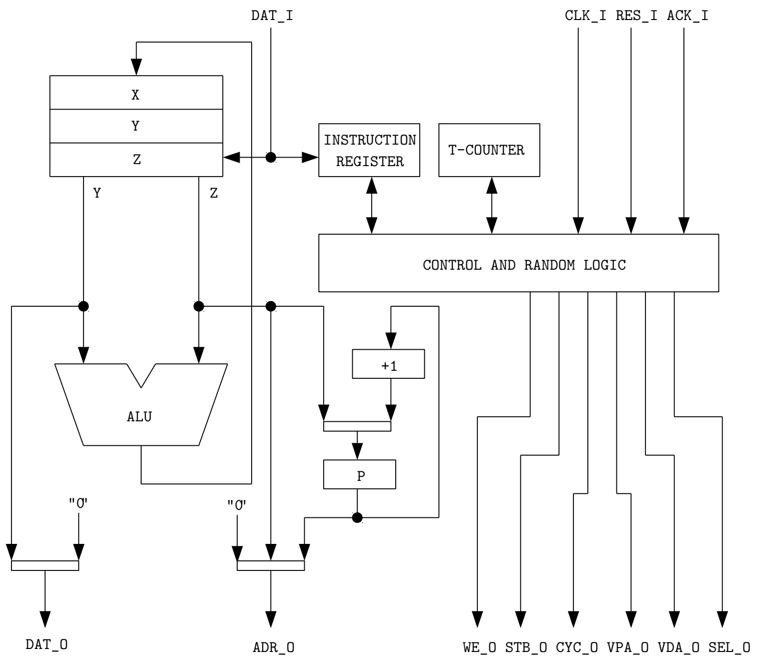


Figure 1: Block Diagram of the S16X4 Processor.

```
LIT a         ; Fetch the datum to store.
FWM
LIT s         ; Reference our array of pointers.
FWM
LIT $44       ; We want element 34 in the array.
ADD
FWM
LIT y         ; Calculate final storage address.
FWM
ADD
SWM           ; Store the datum.
```

Observe that no more than two stack cells (Y and Z, specifically) are involved in computing the effective address for the subsequent store instruction, allowing X to hold onto the value we wish to store during effective address calculation.

2.2 ALU

The arithmetic/logic unit, or ALU, computes identities, sums, bitwise ANDs, and bitwise exclusive-ORs of Y and Z. The NOP, ADD, AND, and XOR instructions respectively selects which of these results will be chosen as the result.

2.3 Instruction Register

The instruction register holds the most recently fetched package of instructions. Since S16X4 opcodes consume only 4 bits in memory, the S16X4 packs four instructions per instruction word, as illustrated below.

Slot 1	Slot 2	Slot 3	Slot 4
15..12	11..8	7..4	3..0

The processor will execute instructions starting first with slot 1, and continuing up to and including slot 4 before fetching another instruction.

The S16X4 fetches a new instruction word if it detects that all remaining opcodes are NOP instructions. This includes the case where all four slots contain NOPs.

2.4 Program Counter Register

The P register holds the location of the next program literal or instruction word. It contains only 15 bits, hard-wiring bit 0 to zero to enforce even addressing while fetching instructions.

2.5 T-Counter

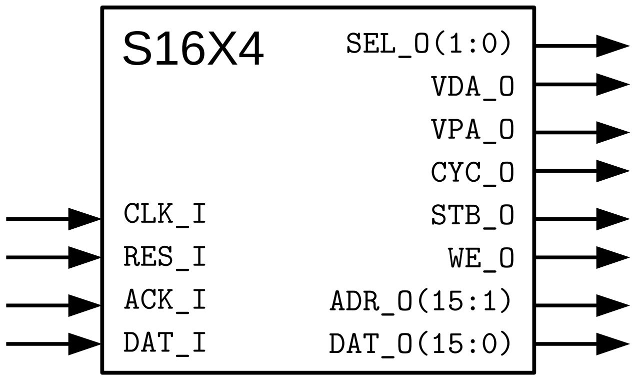
This internal register controls when the processor should fetch an instruction word, when it is safe to perform non-program memory references, et. al. In essence, it determines which slot in the instruction register is currently being executed, and defines extra "slots" for handling reset logic, instruction fetching, etc.

2.6 Control and Random Logic

The random logic portion of the processor interprets the currently executing instruction, if any, the T-counter, as well as the current state of any bus transaction in progress to decide how best to proceed. It can either fetch another instruction word, advance to the next, delay the current bus transaction until some external device says it's OK to proceed, etc.

3 Hardware Interface

3.1 Logic Symbol



3.2 Signal Descriptions

The S16X4 complies with Wishbone B3 bus master standards with a 16-bit port size and 8-bit granularity.

3.2.1 SYSCON Signals

CLK_I. Provides the standard time-base for the processor. All processor state transitions occur on the rising edge of this signal.

RES_I. When asserted during the rising edge of CLK_I, the processor resets to its power-on default state, immediately commencing a new instruction packet fetch. See section 4 for more details.

3.2.2 MASTER Signals

ACK_I. If negated upon the rising edge of CLK_I during a read or write bus transaction, the processor will insert a wait-state, holding the entire processor state as-is. If asserted during a CLK_I rising edge, the bus transaction completes, thus allowing the processor to make progress. However, if the processor does not require the use of the bus, namely when either CYC_O or STB_O are negated, the state of ACK_I will be ignored, allowing the processor to make progress at the CLK_I frequency regardless of external bus activity.

ADR_O(15:1). This 15-bit bus provides the word address for an external memory or peripheral to decode. This bus is valid only while STB_O remains asserted; if STB_O is negated, the value on this bus is undefined.

CYC_O. This signal serves as a flag to external bus arbitration logic that the S16X4 wishes to use the bus, asserting it when it has data to transfer, and negating it otherwise. As such, CYC_O qualifies *all* other bus signals except for RES_I and CLK_I.

DAT_I(15:0). When reading from external memory or peripherals, the addressed memory or peripheral puts its data on this bus. This bus must be valid at least by the time ACK_I is asserted. The processor senses the state of this bus only when it asserts STB_O and negates WE_O; it ignores it otherwise.

DAT_O(15:0). When writing to external memory or peripherals, the addressed memory or peripheral must accept data from this bus. This bus will remain valid at least until ACK_I is asserted and the following CLK_I rising edge. The processor will place valid data on this bus only while asserting STB_O. Peripherals must never trust the value on this bus when the processor negates STB_O.

SEL_O(1:0). These signals indicates which byte-lanes are valid. SEL_O(1) indicates that the processor expects valid data on DAT_I(15:8) or that the it drives valid data on DAT_O(15:8), while SEL_O(0) does the same respectively for DAT_I(7:0) and DAT_O(7:0). STB_O qualifies these signals. Assertion of both signals implies a full 16-bit transfer.

STB_O. This signal qualifies a single bus transfer. As of this writing, the S16X4 supports neither read-modify-write bus transactions nor burst transactions; thus, the S16X4 maps individual transfers to their own transactions by driving STB_O and CYC_O with the same state for any given clock cycle. It's important to remember, however, that CYC_O requests the use of the bus, while STB_O qualifies a single bus transfer. Future generations of this processor may exploit this semantic of the Wishbone bus without notice. Thus, device-enable and acknowledgement signals should be derived from STB_O $\wedge$ CYC_O, not just CYC_O, and certainly never from STB_O alone.

WE_O. If asserted, the current bus transaction is a write cycle (data on DAT_O(15:0)). If negated, the processor will expect an external peripheral or memory to drive data on its DAT_I(15:0) bus. STB_O qualifies this signal.

VPA_O. Also known as TGA_O(1). When asserted, the address on the bus refers to a valid program address. This signal will be asserted while the processor is fetching either an instruction packet or instruction operands. STB_O qualifies this signal.

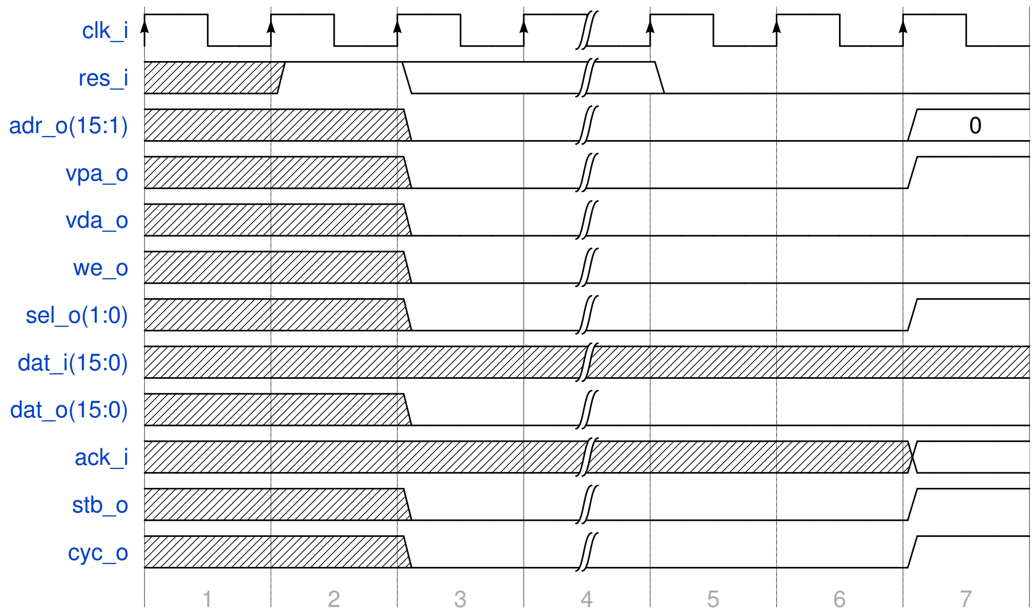
VDA_O. Also known as TGA_O(0). When asserted, the address on the bus refers to some kind of data. If VPA_O is negated, the address corresponds to data memory or peripherals; otherwise, the fetch is for an instruction operand. If negated, the fetch must be for an instruction packet. STB_O qualifies this signal. To summarize:

VPA_O	VDA_O	Type of Memory Access
0	**0**	**Not possible except when (CYC_O$\wedge$STB_O) = 0 as well.**
0	**1**	**Data memory access (fetch or store).**
1	**0**	**Instruction packet fetch.**
1	**1**	**Instruction operand fetch.**

3.3 Timing Diagrams

3.3.1 Reset Timing

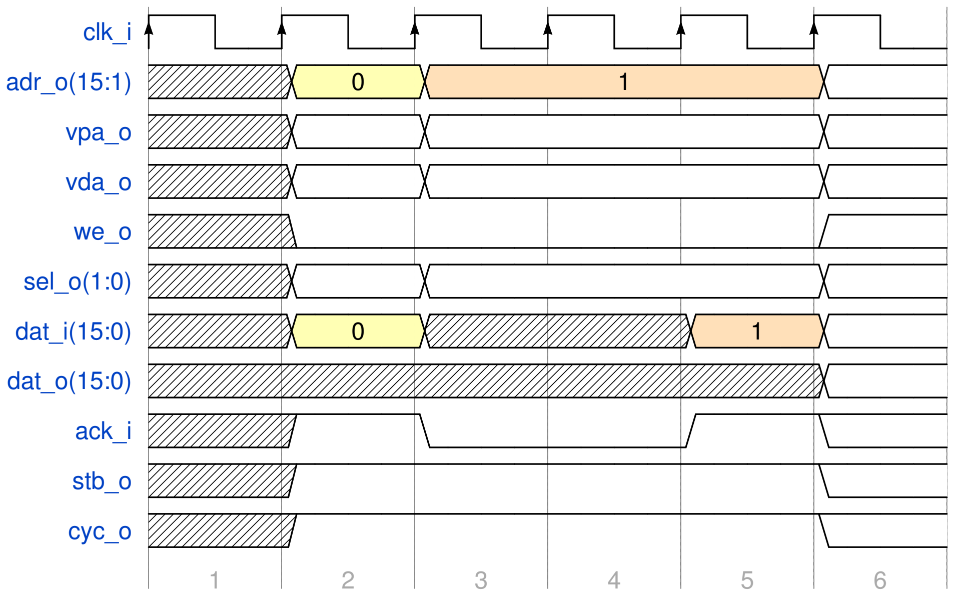
Reset timing follows standard Wishbone B3 recommendations. For any rising clock edge, the S16X4 will reset immediately if RES_I becomes asserted, and will continue to reset for as long as RES_I remains asserted, plus one cycle thereafter.



3.3.2 Memory Read

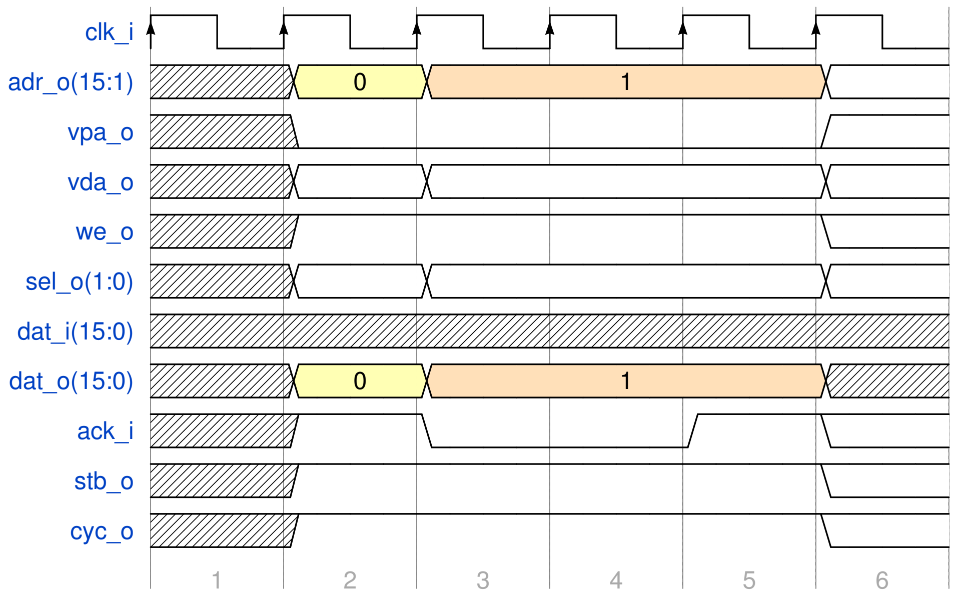
Read timing follows standard Wishbone B3 recommendations. For logic that responds fast enough, single-cycle transactions (as seen in clock cycle 2) are supported; just make sure the data is valid on the DAT_I inputs no later than the minimum set-up time to the next rising clock edge.[2] For slower devices, or for those devices which require additional clocking requirements (e.g., synchronous memories), the ACK_I signal may be negated as long as necessary to insert wait states, as per cycles 3, 4, and 5.

[2]Consult the timing reports generated by your Verilog synthesis for these data. Since this document describes the behavior of a Verilog model, no concrete timing information can be provided.



3.3.3 Memory Write

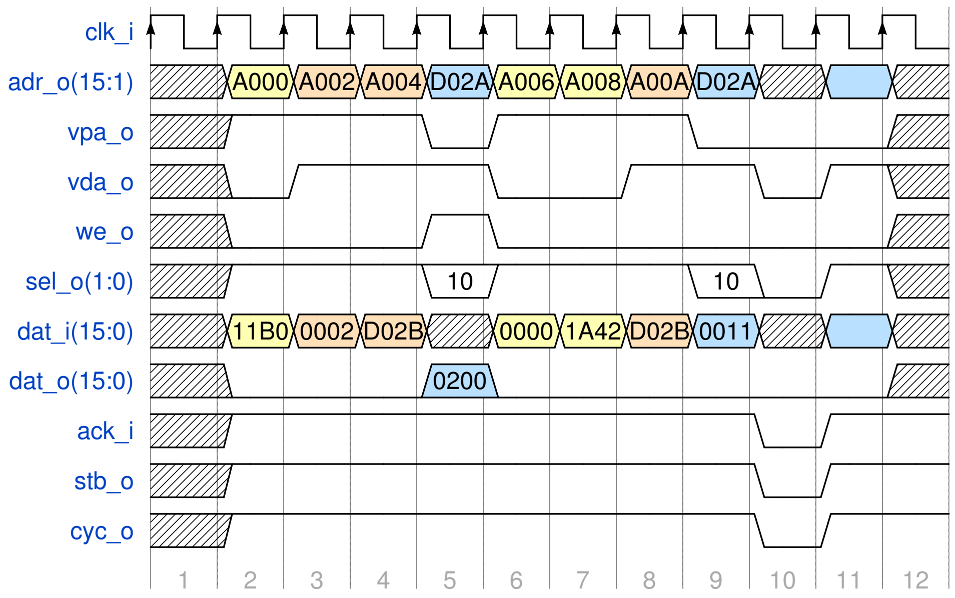
Write timing follows standard Wishbone B3 recommendations. For logic that responds fast enough, single-cycle transactions (as seen in clock cycle 2) are supported. For slower devices, or for those devices which require additional clocking requirements (e.g., synchronous memories), the ACK_I signal may be negated as long as necessary to insert wait states, as per cycles 3, 4, and 5. Observe that VPA_O will never assert during a write cycle.



3.3.4 Instruction Fetch Timing

Instruction fetches occur by issuing memory read cycles to program space (VPA=1, VDA=0; see cycles 2, 6, and 7). Depending on the instructions fetched, between zero and four execution cycles occur, whereby the state of the processor bus indicates program operand fetches (VPA=1, VDA=1; see cycles 3, 4, and 8), data memory reads (see cycles 9 and 11) and writes (see cycle 5). The following timing diagram shows a sample trace with no wait states, involving three instruction word fetches and their subsequent instruction execution cycles and effects. Observe how instructions which operate exclusively on the parameter stack don't require external memory access, and so release the bus completely (see cycle 10). Also observe how an instruction word consisting entirely of NOP instructions results in an immediate fetch of the subsequent instruction word (see cycles 6 and 7).

Calculating the number of cycles an instruction word takes follows a fairly simple formula. If we let n equal the number of instructions in an instruction word, then that word will take $n+1$ cycles to execute (remembering to include the fetch cycle as well). Note that *trailing* NOP instructions do not count as instructions, since the S16X4 will fetch the next instruction word when it sees all subsequent instructions do nothing productive. Leading NOPs, however, *do* count.



4 Initialization

After reset, the S16X4 fetches its first instruction at address $0000. The state of the parameter stack remains undefined until explicitly initialized in software.

5 Instruction Set

Unless stated otherwise, the S16X4 will always consume an appropriate number of elements from the parameter stack prior to generating its results.

5.1 NOP (0)

Function

```
X := X
Y := Y
Z := Z
P := P
```

Description

Takes no action for one execution cycle.

5.2 LIT (1)

Function

```
X := Y
Y := Z
Z := WordAt(P)
P := P+2
```

Description

Fetches the word pointed at by the program counter, placing it onto the parameter stack. The program counter automatically increments to the next word.

5.3 FWM (2)

Function

```
X := X
Y := Y
Z := WordAt(Z∧$FFFE)
P := P
```

Description

Fetches a word from memory. Z must hold the address to fetch from; note that bit 0 of the pointer is ignored. After completing the bus transaction, Z will contain the word stored at that location.

5.4 SWM (3)

Function

```
WordAt(Z∧$FFFE) := Y
X := X
Y := X
Z := X
P := P
```

Description

Stores the contents of Y into the word addressed by Z. Note that bit 0 of Z is ignored.

5.5 ADD (4)

Function

```
X := X
```

```
Y := X
Z := Z+Y
P := P
```

Description

Adds two words on the parameter stack, yielding a single result.

5.6 AND (5)

Function

```
X := X
Y := X
Z := Z∧Y
P := P
```

Description

Computes the bit-wise AND of Y and Z, yielding a single result.

5.7 XOR (6)

Function

```
X := X
Y := X
Z := Z⊕Y
P := P
```

Description

Computes the bit-wise exclusive-OR of Y and Z, yielding a single result.

5.8 ZGO (7)

Function

```
X := X
Y := X
Z := X
P := (Y≠0)→P ; Z
```

Description

If Y holds a non-zero value, continue processing with the next instruction in the current instruction word. Otherwise, jump to the address contained in Z.

5.9 FBM (A)

Function

```
X := X
Y := Y
Z := ZeroExtend(ByteAt(Z))
P := P
```

Description

Fetches the byte contained at the address in Z. Note all 16-bits of Z comprise the byte address.

5.10 SBM (B)

Function

```
ByteAt(Z) := Y∧$00FF
X := X
Y := X
Z := X
P := P
```

Description

Stores the byte held in Y(7:0) at the memory location referenced by Z. Note all 16-bits of Z comprise the byte address.

5.11 GO (E)

Function

```
X := X
Y := X
Z := Y
P := Z
```

Description

Unconditionally jumps to the address in Z. The processor discards the current instruction register contents, causing an immediate fetch for a new instruction word.

5.12 NZGO (F)

Function

```
X := X
```

```
Y := X
Z := X
P := (Y≠0)→Z ; P
```

Description

If Y holds a non-zero value, branch to the address in Z. Otherwise, continue with the next instruction in the currently fetched instruction word, if any.

6 Roadmap

6.1 Support for Interrupts

When working with multitasking or otherwise with event-driven external stimuli, support for interrupts proves valuable. The S16X4 presently lacks interrupts; however, future revisions to the processor will add support for them.

Preliminarily speaking, I anticipate using address $0004 as the interrupt handler entry point. For this reason, good S16X4 coding practices suggests using a LIT/GO combination at location $0000, like so:

```
ORG $0000

XREF cold_boot_entry_point
XREF interrupt_handler_entry_point

; $0000 - reset entry point
LIT cold_boot_entry_point
GO

; $0004 - interrupt entry point
LIT interrupt_handler_entry_point
GO

; ...
```

Saving application context to memory, such as will be necessary to support context-switching in any multitasking environment, will require at least two additional parameter stack registers. For this reason, any S16X4 model that supports interrupts will likely support *three* extra stack registers, U, V, and W, behind the current X, Y, and Z. This should give sufficient room to either ignore the contents of X, Y, and Z completely[3], or enough room to save and restore application X, Y, and Z values to/from memory using any effective address computation you choose.

[3]For this approach to work, the interrupt handler will need to be proven to never use more than three stack elements under any circumstances.

6.2 32- or 64-Bit Successors

Contemporary applications work extensively with graphics data, like it or not. The freedom and flexibility of a fully bitmapped display, however, is paid for in memory consumed by a frame buffer. A simple 640×480, monochrome bitmapped display consumes 38KiB of memory; more than half that addressible by the S16X4. For a true-color display at the same resolution, a framebuffer will consume a minimum of 921 600 bytes of storage. Additionally, the instruction set for the S16X4 lacks many useful primitives, including but not limited to multiplication, bit-shifts, etc. To overcome these limitations in a performant manner often involves the use of a look-up table of some kind.

To that end, working with a larger address space motivates the consideration for wider data paths inside the processor core itself. A hypothetical 32-bit variant of the S16X4 would have several useful characteristics:

- It can pack 8 4-bit instructions per instruction word fetched. This matches nicely the average basic block length of 8 instructions.[4] This translates to a small yet measurable performance increase, particularly useful inside inner-loops, for the microprocessor will need to issue fewer instruction fetches for the same amount of work done. It also reduces the size of a program by roughly halving the number of instruction words fetched for a given number of operands fetched. It decreases the best-case instruction delays from 1.250 cycles to 1.125 cycles.

- If we widen the instructions to 5 bits, we can add support for many missing features, such as an internal return stack for fast subroutine support, bit-shift operators, multiply-step instructions, etc. We pay for this by packing fewer instructions per instruction word, however; with 5-bit opcodes, we can reliably pack only 6 5-bit opcodes. Good code density can still be expected, but not as good as having 8.

- Support for a flat 4GiB address space, providing more than enough storage space for large bitmapped displays or extensive look-up tables.

A 64-bit variant of the architecture may not offer any useful advantage, again excepting for the case of larger address space. This poses an interesting design challenge: does one simply widen everything to 64-bits including the instruction word, or can one be made which retains an essentially 32-bit instruction word (for the purposes of best code density) while retaining an essentially 64-bit data word width? If the latter approaches are taken, how do we handle 64-bit literals in an otherwise 32-bit instruction stream? A 64-bit Steamer adaptation will require further research before any predictions on its characteristics can be made.

[4]http://piotrbania.com/all/articles/bb_instr_stats.html , accessed 2012-Aug-23.

6.3 Macro-instruction Execution

Many instruction patterns emerge as you become more familiar with assembly-level programming of the S16X4 processor. For example, pushing a literal address then executing a GO instruction, takes two cycles to execute, not including instruction word fetches. It's conceivable, then, that the processor can recognize this sequence of instructions (opcodes 1, E, in that order) and implement an inline jump behavior in a single cycle. Conditional branches most likely would benefit the most from this optimization, as they're most likely to be found inside loop bodies.

Fetching a word from memory and using it to operate on the parameter stack happens quite frequently. For example, to add two variables in memory together, and store it in a third, you might use this code:

```
LIT var1
FWM
LIT var2
FWM
ADD
LIT var3
SWM
```

The first and perhaps most obvious optimization involves the processor recognizing a fetch/operate pattern (in this case, opcodes 2, 4 in that order). The processor, in this case, would have a modified data path inside the ALU circuitry, allowing Z to appear on the address bus, while the DAT_I signals route directly to the ALU Z input. In this way, fetching a value and adding, ANDing, or XORing can be accomplished in a single cycle instead of two.

You might think it'd be worth optimizing LIT/FWM combinations since they happen frequently as well. However, doing so continues to require two bus cycles (one to fetch the address, and one to fetch what's at that address), so the net result is a wash.

Since it'd generally prove difficult to recognize instruction sequences across instruction word boundaries, this optimization would make more sense on wider Steamer architectures (32-bit or larger), where greater opportunities for larger patterns exist to be recognized.

7 Original Steamer16 Announcement

The following e-mail message has been edited only so as to fit the typesetting limitations used in this data-sheet.

```
To: <MISC>
Subject: 16-bit stack machine implemented on a Cypress CY37128 CPLD
From: "Myron Plichota" <myron.plichota@xxxxxxxxxxxx>
Date: Fri, 31 Dec 1999 11:01:53 -0500
```

I have developed a 16-bit zero-operand stack machine that I call Steamer16. It fits on the Cypress CY37128 CPLD in an 84-pin PLCC package. Using the 125 MHz speed grade, wirewrapped operation at 20 MHz is predicted by the simulator.

Unfortunately, a dual-stack Forth architecture doesn't fit in the 128 macrocells available. Consequently the design isn't a true Forth chip, but it is a zero-operand stack machine nonetheless. In the future I would like to fit a true Forth architecture to one of the CPLD or FPGA architectures that include on-chip RAM blocks for the stacks.

Being fearfull of actually fitting the design to the target device, the instruction set and architecture was minimized to a ridiculous extent, and it indeed just barely fits. In the future, more elaborate implementations may be implemented on larger devices not suitable for hobby projects due to exotic packaging. For this reason, the documentation contains nerdy phrases typical of growth-path specifications, but don't let that distract you from understanding the Steamer16 initial implementation that exists today.

I plan to design a companion chip, also using the CY37128 to provide a timer, parallel I/O, a funnel shifter, memory decoder/wait state logic, and glue logic for a 16-bit 3-port multiplier/accumulator.

I think it might be bad netiqette to attach the 40Kbyte zip file I have available because of the load on the MISC server. It contains the assembler, JEDEC file, and side documentation. Interested parties should e-mail me for a copy. Please withold any technical questions until having read the documentation package.

BTW, I am well aware of the shortcomings of the Steamer16 implementation, so please don't take me to task over it. My defense is: 1) it fits on a low-cost CPLD in a package hobbyists can deal with 2) companion chips can alleviate some of the shortcomings 3) at 20 MHz, it can clunk through inelegant code sequences quickly (sic)

Following is an excerpt from the assembler documentation (STASM.TXT), part of the zipped package.

Happy New Millenium, MISCers! Myron Plichota

****

Programming Model:

The Steamer architecture consists of a program counter (P) and a 3-deep RPN evaluation stack (TOP, 2ND, 3RD). P is cleared on reset. The stack registers are undefined until loaded under program control. There is no program status word or carry flag. P addresses instruction groups, not necessarily individual instructions. Steamer architecture mandates operations on natural size words without forbidding other data types.

Steamer16 implements the Steamer architecture in 16 bits, with no enhancements.

Stack diagrams:

Stack diagrams are used to describe instruction behavior by showing both the inputs on the stack and the results in a concise notation. The input list is on the left-hand side of the "--" before/after separator, the results are on the right-hand side. eg. (3RD 2ND TOP -- 3RD 2ND TOP)

The input list shows the proper order of input entry in left-to-right order. The input list shows only the requisite stack entries.

The output list shows all three entries. The symbols x, y, and z, are used to denote the original values of any surviving independent stack entries.

Instruction Descriptions: opcodes are in hexadecimal order

```
NOP, {0} ( -- x y z)           no operation
lit, {8} ( -- y z data)        P++ read memory at P, increment P
@,   {9} ( addr -- x y data)   read memory at addr
!,   {A} ( data addr -- x x x) write data to memory at addr
+,   {B} ( n1 n2 -- x x n1+n2) add 2ND to TOP
AND, {C} ( n1 n2 -- x x n1&n2) and 2ND to TOP
OR,  {D} ( n1 n2 -- x x n1|n2) or 2ND to TOP
XOR, {E} ( n1 n2 -- x x n1^n2) exclusive-or 2ND to TOP
zgo, {F} ( flg addr -- x x x)  if flg equals 0 then jump to addr
                               else continue
```

Notes:
  1) 3RD is sticky. When the stack shrinks it holds its value.
  2) lit, is the only instruction that grows the stack, destroying 3RD.
  3) The Steamer16 instruction set contains no additions to the Steamer required instruction set.

4) Opcodes {1..7} are implemented as no operation and are not part of the Steamer required instruction set.

Instruction Timing:

Steamer16 executes all instructions in 1 clock cycle. A quartet fetch cycle is required when the current quartet has finished executing or a jump is taken. For sequential execution, quartets are fetched and executed in 5 clocks.

Software delays are deterministic and may be counted from the fetch of any quartet.

The adder for the +, instruction is implemented as a cascade of 8 2-bit ripple-carry adder cells. Running on a 125 MHz part, the maximum clock frequency is 20 MHz for unambiguous results.

Instruction timing is not mandated in the Steamer architecture.