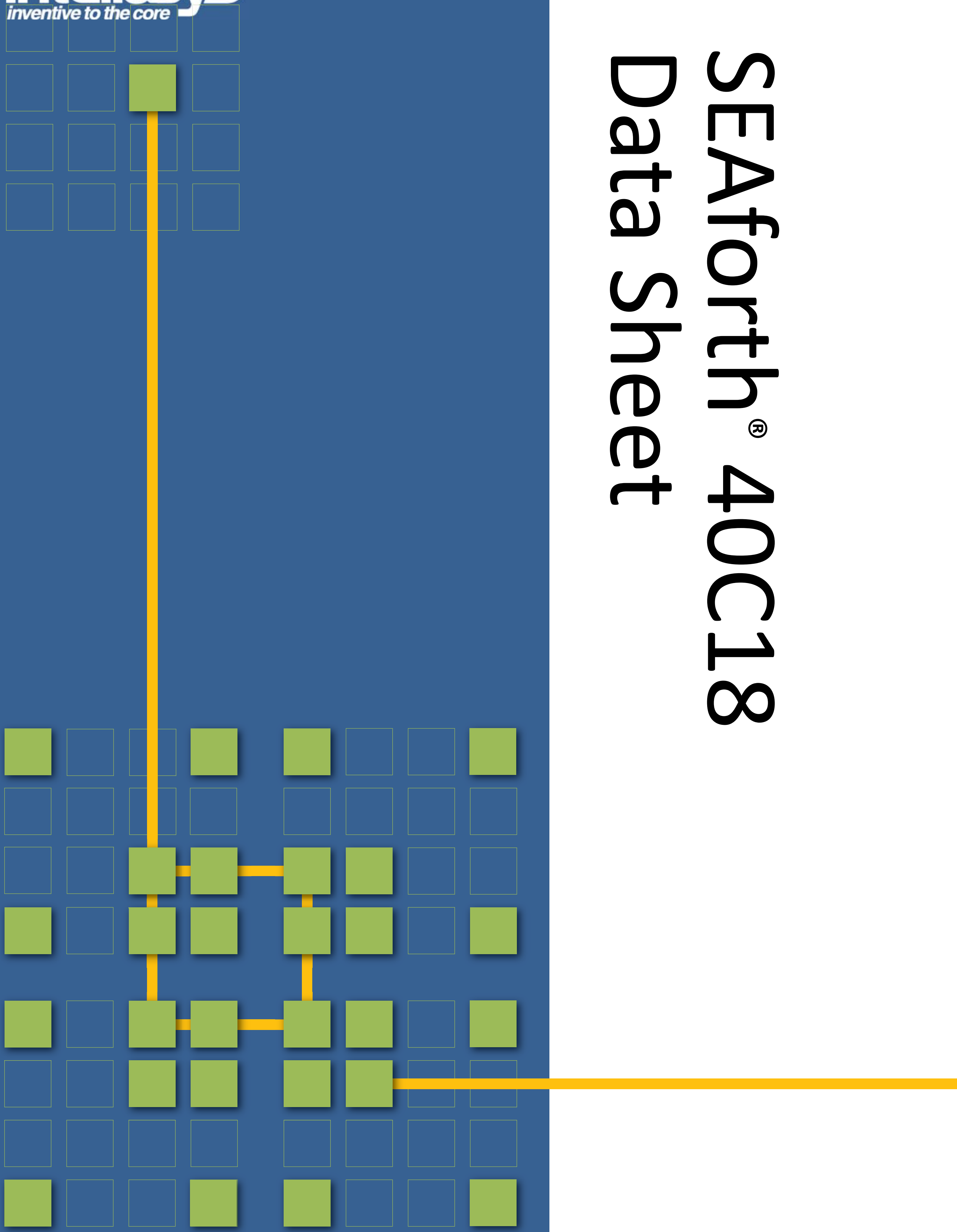
intellaSys®
inventive to the core

SEAforth® 40C18 Data Sheet



Current revision: 9/23/08

Contents

List Of Figures

List Of Tables

Chapter 1 Introduction of the SEAforth Array Processor

Send feedback

The SEAforth 40C18 is one of the IntellaSys Scalable Embedded Array multicore processors. It has an array of 40 corSEAforth 40C18 Data Sheet (Preliminary)es; each of the C18 cores is a complete computer, with its own ROM, RAM, and inter-processor communication. Together they can deliver up to 26 billion operations per second. The SEAforth 40C18 is a perfect embedded computer solution for consumer applications that demand high processing power and low power dissipation.

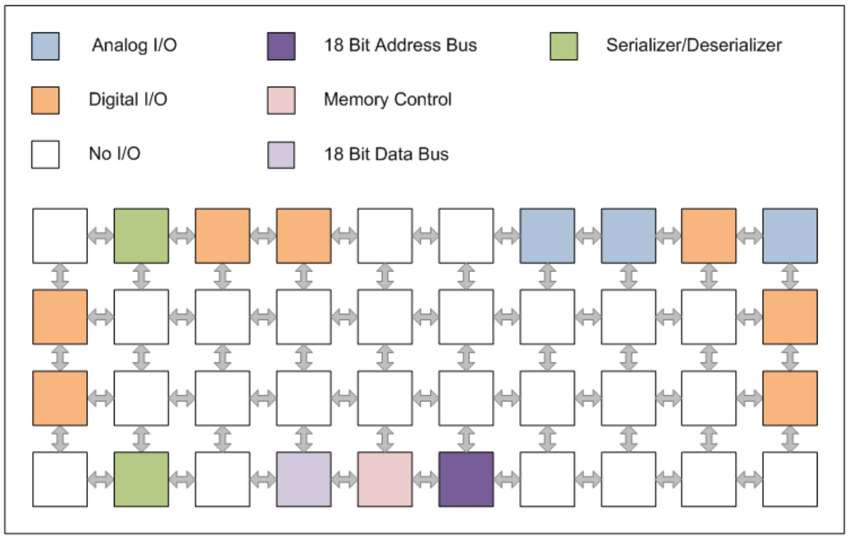


Figure 1.1 Block Diagram of SEAforth 40C18

With 40 cores to work with, designers can dedicate groups of them to specific tasks such as FFT and DFT algorithms. The result is a tightly-coupled, extremely versatile user-defined group of dedicated processors assigned to specific tasks. Some can be doing highly compute-intensive audio processing, while others handle wireless interfaces, external memory, and user interface functions. And since each core has its own ROM and RAM, there is less need to go to external memory.

Each core runs asynchronously, at the full native speed of the silicon. During interprocessor communication, synchronization happens automatically; the programmer does not have to create synchronization methods. Adjacent cores communicate through dedicated ports. A core waiting for data from a neighbor goes to sleep, dissipating less than one microwatt. Likewise, a core sending data to a neighbor not ready to receive it goes to sleep until that neighbor accepts it.

A wake up occurs almost instantly, upon the rising edge of the synchronizing signal. With the wake up logic controlling power use, there is no need for complex power control strategies. Power is conserved as a natural consequence of good program design. External I/O signals may also be used to wake up sleeping processors. The small size and low power make the SEAforth 40C18 a good value both in terms of MIPS per dollar and MIPS per milliwatt.

I/O ports on the SEAforth 40C18 are highly configurable because they are controlled by firmware. The 4-wire SPI port, the 2-wire serial ports, and the single-bit GPIO ports can be programmed to perform a large variety of functions. With the available processing power, wireless solutions become possible without the need for separate wireless chips. Ports can be programmed to support I2C, I2S, asynchronous serial, or synchronous serial ports. Serial ports can also be used to connect multiple SEAforth 40C18s.

In addition to serial I/O, two nodes have two dedicated parallel I/O ports. These can be used for parallel I/O, or when combined, can drive an external memory device.

1.1 The Core

Send feedback

Each core is an 18-bit, stack-oriented machine designed for maximum execution speed and minimum power consumption. The instruction set consists of 32 basic opcodes and each core uses a *data stack* for manipulating parameters and a *return stack* for control flow nesting. Each core has its own ROM (for the IntellaSys-supplied BIOS) and its own RAM for user code and data. To reduce bottlenecks, each core's code is stored in its own memory.

At boot time, code is loaded into the appropriate core's RAM. There are several ways this can be done, but the simplest is from FLASH memory through an SPI port. A single FLASH chip can load multiple SEAforth 40C18 chips that are interconnected.

1.2 Nodes and Cores

Send feedback

Whenever the generic architecture is referred to, it is described in terms of a *core*. *Nodes* are specific instances of the core. Nodes have position and reference numbers.

1.3 Number Notation

Send feedback

All numbers in this document that are preceded by $ are in hexadecimal representation. (The use of $, rather than 0x0, is a preferred notation in Forth.) All other numbers, unless specified otherwise, are in decimal.

- 10 equals decimal 10
- $10 equals decimal 16

Chapter 2 The C18 Core

2.1 Core Overview

Send feedback

Each C18 core in a SEAforth device is identical to the others in terms of opcodes and architecture. Individual cores have different I/O options, and the ROM-based firmware differs slightly as well. Each core is an 18-bit computer that is a Forth stack machine. Its instruction set consists of 32 basic opcodes. It uses a data stack for parameters and a return stack for control flow. A C18 core block diagram is shown in Figure 2.1.

Forth is a stack-oriented language; since many opcodes obtain their operands directly from the stacks, they are known as zero-operand opcodes. As a result, all opcodes are 5 bits in length, allowing multiple opcodes to be packed into and executed from a single 18-bit instruction word.

Table 2.1 C18 Registers

Name	Description
P	10-bit P register (9 bits used for addressing)
I	18-bit Instruction register.
T, S	Top and Second of 10 18-bit data stack registers.
R	Top location of 9 18-bit return stack registers, accessible via **`push`**/**`pop`**, **`call`**/**`return`**, ;:.
A	18-bit general purpose, addressing, and auto-increment addressing register.
B	9-bit addressing register.

The registers are summarized in Table 2.1, C18 Registers. The P register in the C18 core is 10 bits wide (its 9 low bits are used for addressing). Its output drives the address bus during sequential program access, but not during changes in sequential flow. Regardless of the address source, whenever a new instruction is latched into the instruction register, P is loaded via the incrementer. Each instruction word typically contains multiple opcodes; these are selected by the slot selector and fed to the decoder and control logic. This logic will direct the overall function of the core.

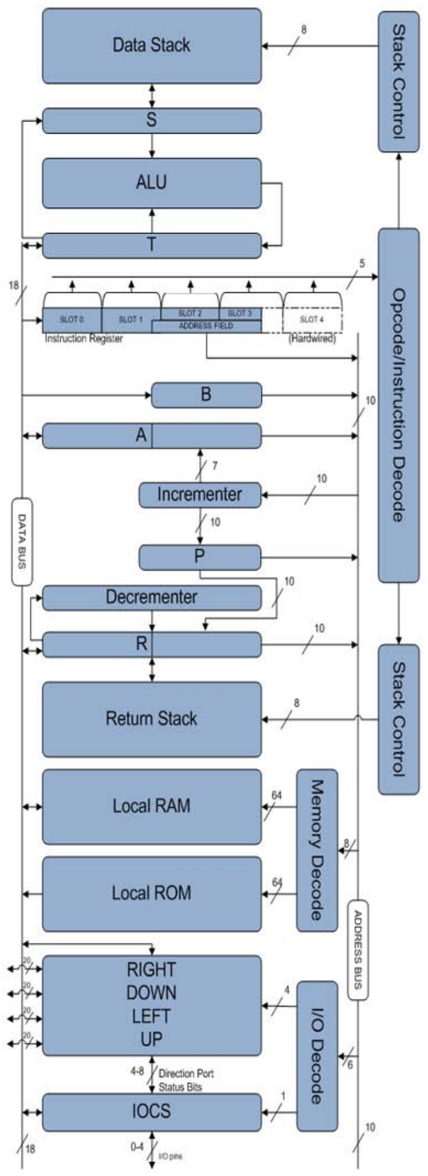


Figure 2.1 SEAforth C18

The data stack is used for all arithmetic operations. Its two uppermost locations can be referenced directly, as T (Top) and S (Second). Together, T and S feed the ALU, whose result is directed back to T.

The return stack is used for subroutines and as a loop counter and for temporary storage. Its uppermost location is referred to as R. Stack operation is covered in more detail later in this chapter.

There are two addressing registers: A and B. A is 18 bits wide. It can be read as well as written and can thus be used for addressing or temporary storage. It is also used in the multiply step. Register B is 9 bits wide. It can be written but not read. It is used by fetch and store opcodes that use B as the pointer.

Each core has 64 words of local RAM and 64 words of local ROM. Each C18 is connected to neighbors by synchronizing communication ports. Each core runs asynchronously using its own internal time-base, but are automatically synchronized to each other whenever they communicate vis these ports.

Overall execution flow is driven by the decode and control logic block. Sequential execution sequence begins with the value in the P register being driven onto the address bus.

The address bus is used in two places: it selects a memory (or in some cases I/O) location; and it drives the incrementer, which in turn presents an incremented value back to the P register.

The addressed memory location drives the data bus; the value on the data bus is latched into the Instruction register, and the value in the first slot is fed to the decode and control logic block. This logic controls what the ALU and other elements do. If the decode and control logic block sees that none of the remaining opcodes need the address bus, it allows the incremented address value from the P register onto the address bus, thus beginning prefetch of the next instruction word.

2.2 Stack Structure

Send feedback

The C18 is a dual-stack computer. It has a data stack for parameters manipulated by the ALU, and a return stack for nested return addresses used by **`call`**, **`;:`**, and **`return`** opcodes. The return stack is also used by **`push`**, **`pop`**, **`next`**, and **`unext`** opcodes.

Each stack is an array of registers. The top two positions on the data stack are named T (for Top) and S (for Second). Below these is a circular array of eight more stack registers. One of the eight registers in the circular array is selected as the register below S at any time. The ten data stack registers are 18 bits wide.

The top of the return stack is named R. Below R is a circular array of eight return stack registers. One of the eight registers in this array is selected as the register below R at any time.

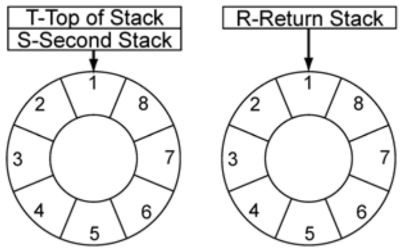


Figure 2.2 Data and Return Stacks

The **addressing section** of the P register is 10 bits wide, but the return stack is 18 bits wide. This is done to allow the return stack to be used to hold 'data' variables, as well as addresses. The **`push`** and **`pop`** opcodes transfer 18-bit data between T in the data stack and R in the return stack.

Figure 2.3 shows an example of how 10-bit addresses are pushed onto the 18-bit return stack. The opcodes **`unext`** and **`next`** use the top value of the return stack R, as an 18-bit loop counter.

In this example the top of the stack is $3FFFF, and below that $02103, as shown in (1).

An opcode (for example, a **`call`** residing in Slot 0 of memory location $030) executes. The low 10 bits of the P register (already incremented to $031) are pushed into R. The previous contents of R move down the stack, but the upper half of R retains its value as well, as shown in (2).

When a **return** executes, the $031 is popped and R contains the item one place down on the stack, namely $3FFFF, as shown in (3). In (4), a **pop** opcode pops the $3FFFF and leaves $02103 in R.

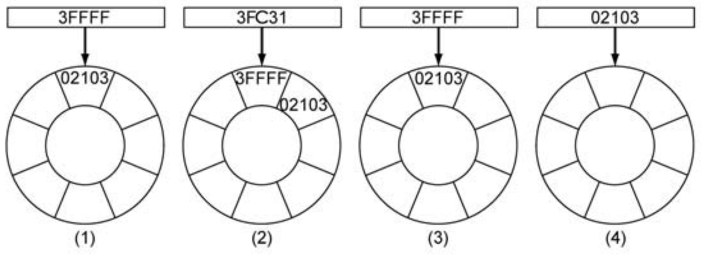


Figure 2.3 10-Bit Addresses on the 18-Bit Return Stack

2.2.1 Stack Operation

Send feedback

Because the C18 core stacks have circular lists of registers at the bottom, the stacks cannot overflow or underflow out of the stack area. Instead, they wrap around the ring of eight stack registers. Because the stacks have finite depth, pushing anything to the top of a stack means something on the bottom is overwritten, as shown in Figure 2.4. Pushing more than 10 items to the data stack, or more than 9 items to the return stack must be done with the knowledge that it results in the item at the bottom of the stack being overwritten. The concept of stacks being full or empty represents an assumption in software, as the hardware always deals with 10 items in the data stack and 9 items in the return stack.

As shown in the sequence from (1) to (12), stack entries move down through the top two locations and are written into sequential locations. ‘Old’ data in the stack is overwritten. The first overwrite of ‘new’ data occurs at (11), where the first value that was pushed onto the stack ‘a’ is overwritten. Continued writes to the stack will cause elements to be written into the ring indefinitely. There is no hardware detection of stack overflow or underflow conditions.

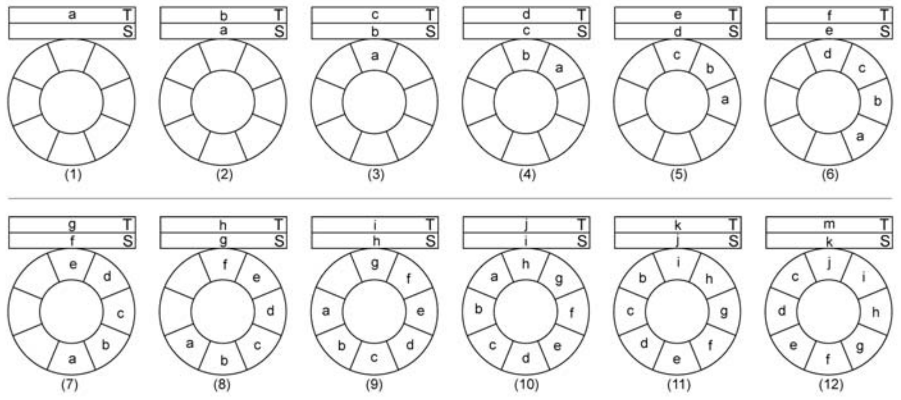


Figure 2.4 Stack Filling and Overflow

When a stack is popped, the lowest item in the bottom eight will be duplicated, as shown in Figure 2.5. A full stack is shown in (1). As the stack is repeatedly popped, items from the circular buffer are moved up. After the eighth pop, the pointer has returned to it where it was (9), and the system will pop the same values again, as shown in (10)-(12).

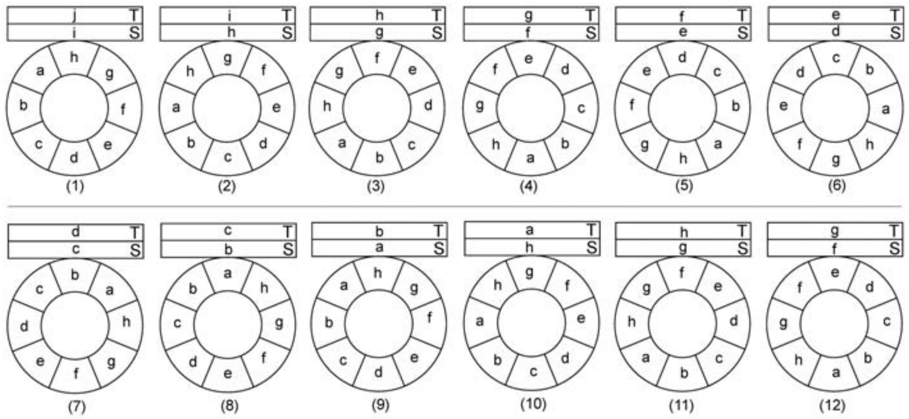


Figure 2.5 Stack Pop Operation

There is no limit to how many times the eight items can be read in sequence off of the stack. Algorithms that cycle through a set of parameters that repeat in eight, four, or two items on the data stack (or eight, four, or two items on the return stack) can repeatedly read them from the stack as the bottom registers will just wrap

2.2.2 Stack Techniques

Send feedback

Software can take advantage of the circular buffers at the bottom of stacks. Having bounded stacks allows two unconventional usages:

- circular reuse
- programming with abandon

Circular reuse is *reading* beyond the "end" of the stack. Because the stacks are implemented as arrays, reading a value (**pop**, **drop**) just moves the pointer to the previous location modulo the stack size, leaving the data in place. When read repeatedly, the pointer eventually returns to that position, where the same data will be read again. This saves reloading literals or calculated values from memory when they are used repetitively.

Programming with abandon is *writing* beyond the "end" of the stack. After writing to a location, the pointer moves to the next location modulo 8, the stack size. When a stack is written repeatedly, the effect is to overwrite previously written data which has been "abandoned." This saves crucial opcode space (and time) compared to explicitly moving the stack pointer back (**drop**, **pop**).

For example, this technique can be used in a loop that waits for a start bit. The code reads the input bit from IOCS and loops, using **-if**, until it sees the bit 'true.' Since **-if** does not remove the top item on the stack, the loop leaves a new value in T each time. After 10 loops the old values at the bottom of the stack are being overwritten; billions of values may be put on the stack in this loop. When it exits the loop, it acts as if the stack was empty. No additional code is required to reset a stack pointer to get an empty stack at the end of the loop. In both situations it's important to distinguish between intentionally and inadvertently causing the behavior: one is a feature, the other is a bug!

2.3 Program Memory

Send feedback

Each C18 core executes code only from its own local memory and ports. Although RAM is mapped from $000 through $07F and ROM from $080 through $0FF, the arrays are only 64 physical 18-bit words long and are thus each mapped twice (address bit 6 ($040) is not actually decoded).

For example, address $008 decodes to the same RAM word as $048. Additionally, the incrementer is only seven bits wide; thus the next logical ROM word after $0FF is $080, which decodes to the same memory word as $0C0.

2.4 I/O Ports and Registers

Send feedback

I/O in the SEAforth 40C18 is visible to software in one of three ways:

- Digital I/O accessible through IOCS, DATA, and UP addresses.
- Interprocessor Communication, internal I/O accessed via direction ports, as opposed to GPIO.
- Analog I/O accessible through IOCS, DATA, and UP registers.

IntellaSys reserves the term *port* for the interprocessor communication function. These locations are not data storage elements. The term 'register' is used for an address location, which can hold data for an indefinite period of time.

See Chapter 4 for more information on ports and interprocessor communication.

Chapter 3 Processor Opcode Descriptions

The C18 core has 32 opcodes, each five bits long. The 18-bit instruction word contains four opcode 'slots,' numbered Slot 0, Slot 1, Slot 2, and Slot 3. (Slot 3 is short.) The five-bit opcodes reside in slots in the instruction word, as shown at the top of Table 3.5, Address Length. By convention, an 'instruction word' is the 18-bit value held in the Instruction Register, and an 'opcode' is the five-bit value in a slot.

3.1 Instruction Set

Send feedback

There are 32 opcodes in the C18 instruction set. Opcodes that reference memory are listed in Table 3.1; ALU-oriented opcodes are listed in Table 3.2.

Table 3.1 Summary of C18 Instruction Set – Memory-Class Opcodes

Name	Pronunciation	Type	Helps +	Brief Description
`call`	`CALL`	Branch	Yes	Calls a subroutine.
`; return`	`RETURN`	Branch	Yes	Returns from a subroutine.
`jump`	`JUMP`	Branch	Yes	Transfer control to a new location. Used to construct loops.
`;:`	`COROUTINE`	Branch	Yes	Transfer control to a coroutine.
`if`	`IF`	Branch	Yes	If T is 0, a branch occurs.
`-if`	`MINUS IF`	Branch	Yes	If the MSB of T is 0, a branch occurs.
`next`	`NEXT`	Branch	Yes	Used to construct loops and other control structures.
`unext`	`MICRO NEXT`	Branch	Yes	Used to construct loops and other control structures.
`!b`	`STORE B`	Memory	No	Store top of data stack at B.
`!a`	`STORE A`	Memory	No	Store top of data stack at A.
`!p+`	`STORE P+`	Memory	No	Store top of stack at P register, increment.
`!a+`	`STORE A+`	Memory	No	Store top of stack at A, increment.
`@b`	`FETCH B`	Memory	No	Location at B is pushed to data stack.

Table 3.1 Summary of C18 Instruction Set – Memory-Class Opcodes *(continued)*

Name	Pronunciation	Type	Helps +	Brief Description
@a	**FETCH A**	Memory	No	Location at A is pushed to data stack.
@a+	**FETCH A+**	Memory	No	Location at A is pushed to data stack, A incremented.
@p+	**FETCH P+**	Memory	No	Loads a literal, Increments address in P register.

Table 3.2 Summary of C18 Instruction Set – ALU-Class Opcodes

Name	Pronunciation	Type	Helps +	Brief Description
. (nop)	**NO-OP**	Stack	Yes	Fills space and time.
push	**PUSH**	Stack	No	Pops data stack, pushes return stack.
pop	**POP**	Stack	No	Pops return stack, pushes data stack.
dup	**DUP**	Stack	No	Dups top of data stack.
drop	**DROP**	Stack	No	Drops top of data stack.
over	**OVER**	Stack	No	A copy of the second stack element is pushed onto the stack.
b!	**B STORE**	Register	No	B register is loaded from data stack.
a!	**A STORE**	Register	No	A register is loaded from data stack.
a@	**A FETCH**	Register	No	A is pushed onto data stack.
and	**AND**	Bit-wise	No	T, S bit-wise ANDed to T.
xor	**XOR**	Bit-wise	No	T, S bit-wise exclusive-ORed to T.
not	**NOT**	Bit-wise	No	T is one’s-complemented.
+	**PLUS**	Math	No	T, S added; placed in T.
+*	**PLUS STAR**	Math	No	Partial-multiply building block for T, S, and A.
2/	**TWO SLASH**	Math	No	T is right-shifted, sign bit (MSB) is preserved.
2*	**TWO STAR**	Math	No	T is left-shifted, zero as LSB.

3.2 Opcode tables

Send feedback

Each individual opcode description consists of five items: Name, Pronunciation, Slots, “Helps**+**”, and a stack comment. The *name* is how it is referred to in source code. The *pronunciation* is what we call the opcode in spoken descriptions. The *slots* column tells what instruction word slots this opcode may occupy (see Section 3.4 for a discussion of slots). “Helps **+**” tells whether or not this opcocde can immediately precede a **+** opcode (see Section 3.9.1). The stack comment describes the effect a given opcode has upon the data stack and the return stack (see Stack Notation, below).

3.2.1 Stack Notation

Send feedback

Stack comments describe the preconditions and postconditions of the data and return stack.

Data stack comments are of the following form

`D: precondition - postcondition`

Return stack comments are as follows

`R: precondition - postcondition`

For example:

```
D: x1 x2 - x3
```

x2 is the top item on the stack and x1 is the before the operation, second item on the stack.

x3 is the only item on the stack after the operation.

Each precondition and postcondition can have 0 or more arguments. Data and return stack effects are only shown when the corresponding stack is affected; however, if neither stacks is changed “D: - “ is put into the table.

3.3 Opcodes

Send feedback

This section describes the actual opcodes, organized into groups of related functions.

3.3.1 Branch Opcodes

Send feedback

CALL Opcode: `call`
Type: Branch

Name	Pronunciation	Slots	Helps +	Stack
`call`	`CALL`	`0, 1, 2`	`Yes`	`R: - a`

`call` pushes a copy of R onto the return stack and places the address in the current P register into the low 10 bits of R. The next instruction word is fetched from the address determined by the address field. The incremented address is loaded into the P register. The high 8 bits of R are unchanged.

References

Section 2.2 regarding 9-Bit Addresses on the 18-Bit return stack
Section 3.5, Address Length
Section 3.6, Address Increment Rules
Section 3.7, Branch Opcodes.

RETURN Opcode: `;`
Type: Branch

Name	Pronunciation	Slots	Helps +	Stack
`; return`	`RETURN`	`0, 1, 2, 3`	`Yes`	`R: a -`

`return` fetches the next instruction word from the address given by the low order 10 bits of R. The return stack is popped, replacing all 18 bits of R with the next value down. The incremented address is loaded into the P register.

Any unused slots in the instruction word containing the **`return`** are skipped and execution resumes from Slot 0 of the new instruction word.

References

Section 2.2, 9-Bit Addresses on the 18-Bit return stack
Section 3.6, Address Increment Rules

JUMP Opcode: `jump`
Type: Branch

Name	Pronunciation	Slots	Helps +	Stack
`jump`	`JUMP`	`0, 1, 2`	`Yes`	`D: -`

`jump` fetches the next instruction word from the address determined by the address field. The incremented address is loaded into the P register.

References

Section 3.5, Address Length
Section 3.6, Address Increment Rules
Section 3.7, Branch Opcodes

COROUTINE Opcode: `;:`
Type: Branch

Name	Pronunciation	Slots	Helps +	Stack
`:;`	`COROUTINE`	`0, 1, 2`	`Yes`	`R: x1 – x2`

`;:` fetches the next instruction word from the address given by the low order 10 bits of R. The address in the current P register is loaded into the low 10 bits of R without change to the deeper return stack. The incremented address is then loaded into the P register. Any unused slots in the instruction word containing **`;:`** are skipped and execution resumes from Slot 0 of the new instruction word.

The effect is the same as if the address from the P register were swapped with the low 10 bits of R before fetching the next instruction word and incrementing the new address in the P register. The use of this opcode can be thought of as either a calculated or vectored **`call`** or as a **`;:`**; that is, two functions that each can **`call`**/**`continue`** execution in the other at the other's last **`exit`**/**`call`** point.

This can also be thought of as a primitive task switch.

References

Section 3.6, Address Increment Rules.

IF Opcode: `if`
Type: Branch

Name	Pronunciation	Slots	Helps +	Stack
`if`	`IF`	`0, 1, 2`	`Yes`	`D: x - x`

`if` tests the T register. If T is zero, the address of the next instruction word is taken from the branch address field; otherwise, the current P register address is used. The next instruction word is fetched from the selected address. The incremented address is loaded into the P register.

References

Section 3.5, Address Length
Section 3.6, Address Increment Rules
Section 3.7, Branch Opcodes

MINUS IF Opcode: `-if`
Type: Branch

Name	Pronunciation	Slots	Helps +	Stack
`-if`	`MINUS IF`	`0, 1, 2`	`Yes`	`D: x - x`

`-if` tests the sign bit of the T register. If T is positive, the address of the next instruction word is taken from the branch address field, otherwise the current P register address is used. The next instruction word is fetched from the selected address. The incremented address is loaded into the P register.

References

Section 3.5, Address Length
Section 3.6, Address Increment Rules
Section 3.7, Branch Opcodes

NEXT Opcode: `next`
Type: Branch

Name	Pronunciation	Slots	Helps +	Stack

`next`	`NEXT`	`0, 1, 2`	`Yes`	`R: n - (n-1)`
				`R: 0 -`

`next` tests the R register. If R is non-zero, the address of the next instruction word is taken from the branch address field; otherwise, the current P register address is used. The next instruction word is fetched from the selected address. The incremented address is loaded into the P register.

In the case where R is non-zero, all 18 bits are decremented and the new value replaces R. When R is zero, the return stack is popped and R is replaced with the next item down.

The effect of **`next`** is to create a loop with R as the loop counter. The number in R represents the number of remaining times that **`next`** will branch to the top of the loop, or one less than the number of times the loop body is to be executed.

The initial value of R must be set outside the loop, before it begins. The loop count must be pushed onto the return stack explicitly but it will be removed automatically when the loop completes. Code can modify this value, for example to cause early termination of the loop. If, within the loop, the return stack is not balanced, whatever is in R will be treated by **`next`** as loop count.

References

Section 3.5, Address Length
Section 3.6, Address Increment Rules
Section 3.7, Branch Opcodes

MICRO NEXT Opcode: `unext`
Type: Branch

Name	Pronunciation	Slots	Helps +	Stack

`unext`	`MICRO NEXT`	`0, 1, 2, 3`	`Yes`	`R: n - (n-1)`
				`R: 0 -`

`unext`, pronounced “micro-next,” functions similarly to **`next`**, but does not consume an address field. **`unext`** needs no address argument because it does not fetch a new instruction word, either when looping or for the fall through case. Instead, **`unext`** changes the slot pointer to Slot 0 for looping or else falls through to the following slot. When **`unext`** is in Slot 3, responsibility for fetching the next instruction word is passed to Slot 4.

`unext` tests the R register. If R is non-zero, the slot pointer is set to Slot 0; otherwise, the current slot pointer is allowed to advance. The next opcode is decoded from the current Instruction Word content and current slot pointer.

In the case where R is non-zero, the full 18-bit value of R is decremented and the new value replaces R. When R is zero, the return stack is popped and R is replaced with the next item down.

The effect of **`unext`** is to create a short loop with R as the loop counter. The number in R represents the number of remaining times that **`unext`** will return to Slot 0, or one less than the number of times the loop body is to be executed. The initial value of R must be set outside the loop before it begins. The loop count must be pushed onto the return stack explicitly, but it will be removed automatically when the loop completes. Code can modify this value: for example, to cause an infinite loop.

The loop body may contain from zero to three opcodes, depending upon the slot number of the **`unext`**. **`unext`** may be placed in any slot from 0 to 3. The compiler will verify that the address specified by **`begin`** or **`for`** for the loop start points to the same instruction word that **`unext`** will be placed into.

3.3.2 Memory Opcodes

Send feedback

These opcodes provide access to memory locations.

FETCH A Opcode: `@a`
Type: Memory

Name	Pronunciation	Slots	Helps +	Stack
`@a`	`FETCH A`	`0, 1, 2`	`No(*)`	`D: - x`

`@a` causes the contents of the location specified by the A register to be read and pushed onto the data stack. The A register remains unchanged. Refer to Figure 2.4, Stack Operation, for details.

(*) This opcode does help Add opcodes if the **`@a`** is in Slot 2 and the Add opcode is in Slot 3. Refer to Section 3.9, Prefetch, for details.

STORE A Opcode: `!a`
Type: Memory

Name	Pronunciation	Slots	Helps +	Stack
`!a`	`STORE A`	`0, 1, 2`	`No(*)`	`D: x -`

`!a` causes an element to be popped from the data stack and written to the location specified by the A register. The A register remains unchanged. Refer to Figure 2.4, Stack Operation, for details.

(*) This opcode does help Add opcodes if the **`!a`** is in Slot 2 and the Add opcode is in Slot 3. Refer to Section 3.9, Prefetch, for details.

FETCH A+ Opcode: `@a+`
Type: Memory

Name	Pronunciation	Slots	Helps +	Stack
`@a+`	`FETCH A+`	`0, 1, 2`	`No(*)`	`D: - x`

`@a+` causes the contents of the location specified by the A register to be read and pushed onto the data stack. The A register is then incremented. Refer to Section 3.6, Address Increment Rules. Refer to Figure 2.4, Stack Operation, for details.

(*) This opcode does help Add opcodes if the **`@a+`** is in Slot 2 and the Add opcode is in Slot 3. Refer to Section 3.9, Prefetch, for details.

STORE A+ Opcode: `!a+`
Type: Memory

Name	Pronunciation	Slots	Helps +	Stack
`!a+`	`STORE A+`	`0, 1, 2`	`No(*)`	`D: x -`

`!a+` causes an element to be popped from the data stack and written to the location specified by the A register. The A register is then incremented. Refer to Section 3.6, Address Increment Rules. Refer to Figure 2.4, Stack Operation, for details.

(*) This opcode does help Add opcodes if the **`!a+`** is in Slot 2 and the Add opcode is in Slot 3. Refer to Section 3.9, Prefetch, for details.

FETCH B Opcode: `@b`
Type: Memory

Name	Pronunciation	Slots	Helps +	Stack
`@b`	`FETCH B`	`0, 1, 2`	`No(*)`	`D: - x`

`@b` causes the contents of the location specified by the B register to be read and pushed onto the data stack. The B register remains unchanged.

(*) This opcode does help Add opcodes if the **`@b`** is in Slot 2 and the Add opcode is in Slot 3. Refer to Section 3.9, Prefetch, for details.

STORE B Opcode: `!b`
Type: Memory

Name	Pronunciation	Slots	Helps +	Stack
`!b`	`STORE B`	`0, 1, 2`	`No(*)`	`D: x -`

`!b` causes an element to be popped from the data stack and written to the location specified by the B register. All other stack elements move up. Refer to Figure 2.4, Stack Operation, for details. The B register remains unchanged.

(*) This instruction does help Add instructions if the **`!b`** is in Slot 2 and the Add instruction is in Slot 3. Refer to Section 3.9, Prefetch, for details.

STORE P+ Opcode: `!p+`
Type: Memory

Name	Pronunciation	Slots	Helps +	Stack
`!p+`	`STORE P+`	`0, 1, 2, 3`	`No(*)`	`D: x -`

`!p+` causes an element to be popped from the data stack and written to the location specified by the P register. Refer to Section 3.6, Address Increment Rules. All other stack elements move up. Refer to Figure 2.4, Stack Operation, for details.

(*) This opcode does help Add opcodes if **`!p+`** is in Slot 2 and the Add opcode is in Slot 3, or **`!p+`** is in Slot 3 and the Add opcode is in the following Slot 0.

FETCH P+ Opcode: `@p+`
Type: Memory

Name	Pronunciation	Slots	Helps +	Stack
`@p+`	`FETCH P+`	`0, 1, 2, 3`	`No(*)`	`D: - n`

`@p+` is the 'load literal' instruction. It fetches a word of program memory from the address in the P register and pushes it onto the data stack. Address increment rules apply; refer to Table 3.6, Address Increment.

(*) This instruction does help Add instructions if **`@p+`** is in Slot 2 and the Add instruction is in Slot 3, or **`@p+`** is in Slot 3 and the Add instruction is in the following Slot 0.

3.3.3 Stack Opcodes

Send feedback

The opcodes described in this section manage the data and return stacks.

NOP Opcode: `.`
Type: Stack

Name	Pronunciation	Slots	Helps +	Stack
`. nop`	`NOP`	`0, 1, 2, 3`	`Yes`	`D: -`

`nop` has no stack effect. It is used to fill space and time. It allows internal signals to settle, see Section 3.9.1, Addition Opcodes.

PUSH Opcode: `push`
Type: Stack

Name	Pronunciation	Slots	Helps +	Stack

`push`	`PUSH`	`0, 1, 2`	`No`	`D: x –`
				`R: - x`

`push` causes the top item to be popped from the data stack and pushed onto the return stack. Refer to Figure 2.4, Stack Operation, for details.

POP Opcode: `pop`
Type: Stack

Name	Pronunciation	Slots	Helps +	Stack

`pop`	POP	`0, 1, 2`	No	`R: x -`
				`D: - x`

pop causes the top item to be popped from the return stack and pushed onto the data stack. Refer to Figure 2.4, Stack Operation, for details.

DUP Opcode: `dup`
Type: Stack

Name	Pronunciation	Slots	Helps +	Stack
`dup`	DUP	`0, 1, 2, 3`	No	`D: x - x x`

dup causes the element at the top of the data stack (the T register) to be replicated and pushed back onto the data stack. The S register and T register will then contain the same value. The former contents of S are pushed down the stack.

DROP Opcode: `drop`
Type: Stack

Name	Pronunciation	Slots	Helps +	Stack
`drop`	DROP	`0, 1, 2`	No	`D: x -`

drop pops the data stack. Refer to Figure 2.4, Stack Operation, for details.

OVER Opcode: `over`
Type: Stack

Name	Pronunciation	Slots	Helps +	Stack
`over`	OVER	`0, 1, 2`	No	`D: x1 x2 – x1 x2 x1`

over causes the second element in the data stack (S register) to be replicated and pushed onto the data stack. Refer to Figure 2.4, Stack Operation, for details.

3.3.4 Register Opcodes

Send feedback

The instructions described in this section provides access to processor registers.

A STORE Opcode: `a!`
Type: Register

Name	Pronunciation	Slots	Helps +	Stack
`a!`	`A STORE`	`0, 1, 2`	No	`D: x -`

`a!` causes the 18-bit A register to be loaded with the number popped from the data stack. All other stack elements move up. Refer to Figure 2.4, Stack Operation, for details.

A FETCH Opcode: `a@`
Type: Register

Name	Pronunciation	Slots	Helps +	Stack
`a@`	`A FETCH`	`0, 1, 2`	No	`D: - x`

`a@` causes the contents of the 18-bit A register to be pushed onto the data stack. The A register remains unmodified.

B STORE Opcode: `b!`
Type: Register

Name	Pronunciation	Slots	Helps +	Stack
`b!`	`B STORE`	`0, 1, 2`	No	`D: x -`

B STORE causes the 9-bit B register to be loaded with the number popped from the data stack. All other stack elements move up. Refer to Figure 2.4, Stack Operation, for details.

3.3.5 Bit-Wise Boolean Opcodes

Send feedback

NOT Opcode: `not`
Type: Bit-wise

Name	Pronunciation	Slots	Helps +	Stack
`not`	NOT	`0, 1, 2`	No	`D: x1 – x2`

`not` causes the top value of the data stack (T register) to be bit-wise inverted.

AND Opcode: `and`
Type: Bit-wise

Name	Pronunciation	Slots	Helps +	Stack
`and`	AND	`0, 1, 2`	No	`D: x1 x2 – x3`

`and` causes the top two values in the data stack (T register and S register) to be popped from the data stack, bit-wise anded and the result pushed back onto the stack. All other stack elements move up. Refer to Figure 2.4, Stack Operation, for details.

XOR Opcode: `xor`
Type: Bit-wise

Name	Pronunciation	Slots	Helps +	Stack
`xor`	XOR	`0, 1, 2`	No	`D: x1 x2 – x3`

`xor` causes the top two values in the data stack (T register and S register) to be popped from the data stack, bit-wise exclusive ORed and the result pushed back onto the stack. All other stack elements move up. Refer to Figure 2.4, Stack Operation, for details.

3.3.6 Mathematical Opcodes

Send feedback

TWO SLASH Opcode: 2/
Type: Math

Name	Pronunciation	Slots	Helps +	Stack
`2/`	`TWO SLASH`	`0, 1, 2`	No	`D: x1 – x2`

This opcode is called ‘two slash’ after the mnemonic. A slash **/** is used to imply division by 2. Two-slash causes the top value in the data stack (T register) to shifted right one bit position. The most significant bit remains unchanged; that is, the operation sign-extends.

TWO STAR Opcode: 2*
Type: Math

Name	Pronunciation	Slots	Helps +	Stack
`2*`	`TWO STAR`	`0, 1, 2`	No	`D: x1 – x2`

This opcode called 'two star' after the mnemonic. An asterisk * is used to imply multiplication by 2. Two-star causes the top value in the data stack (T register) to be shifted left one bit position. A zero is shifted into the low order bit position.

PLUS Opcode: +
Type: Math**+** in Standard mode

Name	Pronunciation	Slots	Helps +	Stack
`+`	`PLUS`	`0, 1, 2, 3`	No	`D: x1 x2 – x3`

+ pops the top two elements from the data stack, adds them, and pushes the result on to the data Stack. Refer to Appendix 2.4, Stack Operation, and Section 3.9.1, Addition Opcodes, for details on the add opcode.

In extenede mode opperation, **+** replaces T with the sum of S, T, and the latched carry bit. The carry bit out of T17 is saved in the carry latch at the end of the operation.

Note: Standard mode operations never alter nor even reference the carry bit (the carry in is always treated as 0). Additionally, the carry latch state is preserved so that Extended mode operations always see the carry as left by the previous Extended mode operation.

Addition of large numbers is done in Extended mode by first clearing the carry bit, then adding pairs of 18 bit words of the numbers, starting with the low order word, so that the carry propagates upward through the sum. Subtraction is done by adding the one's complement (**not**) of one of the numbers to the other in the same way, except that in this case the initial carry is set to 1, thus completing the two's complementation of one of the arguments.

See Section 3.10 setting and clearing carry latch

PLUS STAR Opcode: +*
Type: Math

Name	Pronunciation	Slots	Helps +	Stack
`+*`	`PLUS STAR`	`0, 1, 2, 3`	No	`D: x1 x2 A: – x1 x3`

+* is used as a building block to multiply two numbers (in S and A), and works by computing a series of partial products, which are added as they are generated. The result is 36 bits in T.A (Register T and Register A), with the LSW in A. Note: T should be cleared before the first **+*** in a set.

If bit 0 (LSB) of A is 0, then the 37-bit (sign extended) T.A register is simply shifted right one bit. However, if bit 0 of A is 1, the contents of S is first added to T before the whole 37-bit T.A is shifted. In either case, the LSB of T is shifted into the MSB of A.

The rules for ripple carry and the potential need for a **nop** or other delay which applies to the **+** opcode also applies here. Refer to Section 3.9.1, Addition Opcodes (page 40). Preceding a **+***with a **nop** will always provide sufficient time for the ALU to settle, but that section also details other approaches.

The result of executing 18 **+*** instructions will be a signed 18x18 multiply producing a 36-bit product of S and A; however, it is possible, through careful treatment of data, to perform much shorter multiplications. See below for an example.

Extended Mode Considerations

In extended mode, when P9 is set, the ALU always includes the previous carry value. This is not a desirable operation for the **+*** instruction, but due to the optimized ALU architecture (which recalculates the sum immediately when either T or S is changed) it cannot be avoided. Therefore it is necessary to first clear the carry when operating in Extended mode. Another simple technique is to define "multiply" words outside the EA address range. NOTE: **+*** never latches a new carry-in value.

8-bit by 4-bit Multiply Example

For shorter multiplications, the simple technique of pre-shifting the longer number may be used. Thus, to multiply an 8-bit number by a 4-bit number, first shift the 8-bit number left (using **2***) four places. Then, the 12-bit result will be left in T after four **+*** steps! Here is some example code:

Initial registers:

T=00 0000 0000 0000 nnnn   four bit argument in T

S=ss ssss snnn nnnn 0000   sign extended 8-bit argument in S, shifted left 4 times

Code:

```
dup a!                    \ 4-bit number to A
. +* . +* . +* . +*       \ 4 steps of safe multiply
push drop pop             \ discard S (optional)
```

Result:

T=ss ssss snnn nnnn nnnn

Note: in this example, the important bits of T are already zero in the 4-bit argument, so there is no need to explicitly clear the whole register.

3.4 Opcode Packing

Send feedback

Slot 3 consists of 3 bits. Opcodes loaded into Slot 3 are truncated; the execution unit, when fetching the three bits from Slot 3, will append 00 and interpret the opcode accordingly. Permitted Slot 3 opcodes are summarized in Table 3.3.

Table 3.3 Slot 3 Legal Opcodes

Name	Opcode		
	Hex	upper 3 bits	lower 2 bits
`return`	00	000	00
`unext`	04	001	00
`@p+`	08	010	00
`!p+`	0c	011	00
`+*`	10	100	00
`+`	14	101	00
`dup`	18	110	00
`nop`	1c	111	00

3.5 Address Length

Send feedback

The size and interpretation of the branch address field is affected by opcode packing. The field is always right justified within the same instruction word as the opcode that controls it. The slot number of the branch opcode will determine the number of high order bits available to the address field. Each of the branch cases, and its affect upon address size and effective address calculation, are illustrated in Table 3.4.

Example 1: If the opcode is in Slot 0, there is room for a full nine bits of address and a tenth bit to set extended mode.

Example 2: In Slot 1, there is room for eight address bits. Note that eight bits will reach all of RAM and ROM, but cannot reach the I/O ports. In other words, P8 is always set to 0 when branching from slot 1. P9 is not affected by Slot 2 banches.

Example 3: In Slot 2, 3 address bits are available. These three bits are combined with five bits of the P register to form an 8-bit address. P8 is set to 0 when branching from Slot 2. Note that the address is in the same 8-word page as the value of the P register when the opcode executes. P9 is not affected by Slot 2 banches.

Table 3.4 Address length

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode Slot	Slot 0					Slot 1					Slot 2					Slot 3		
Example 1	branching opcode					reserved			10-bit address									
Address									I9	I8	I7	I6	I5	I4	I3	I2	I1	I0
Example 2						branching opcode					8-bit address							
Address									p9	0	I7	I6	I5	I4	I3	I2	I1	I0
Example 3											branching opcode					3-bit address		
Address									p9	0	p7	p6	p5	p4	p3	I2	I1	I0

3.6 Address Increment Rules

Send feedback

It's important to understand how the Address Increment mechanism works, as it affects both instruction fetch and operand fetch and store. You may wish to refer to Figure 3.1 to aid in understanding the following material.

The Address Incrementer takes its input from the address bus and calculates a new address in parallel with the bus access time. The results are made available to either the P register or the A register. The use of the incremented address, if any, is determined by the current opcode.

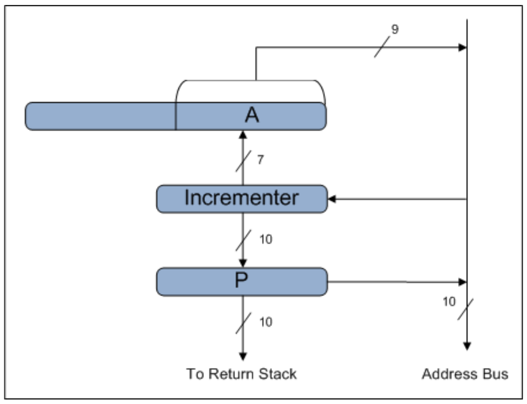


Figure 3.1 Incrementer

3.6.1 Address-Increment Opcode*s*

Send feedback

The normal process of fetching and latching each instruction word will latch the output of the Incrementer to the P register. In addition, **`@p+`** and **`!p+`** will send the result to the P register as well. **`@a+`** and **`!a+`** send the incremented address to the A register. These four opcodes are referred to as address-increment opcodes, which together with the instruction fetch process constitute all cases of use of the address incrementer result.

Table 3.5 Address-Increment Opcode Example

Address	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
N	.					.					@p+					.		
N+1	literal value																	
N+2																		

Table 3.5 shows an Address-Increment Opcode example. By the time the **`@p+`** is actually executing, the address bus value has gone back through the Incrementer and is pointing at location N+1. The control logic allows this value to be used to load the value from memory, and as that is occurring, the value N+1 is fed to the Incrementer, and the control logic causes the value N+2 to be latched into the P register.

(Note that the other slots in the instruction word can be other ALU or memory opcodes, and do not need to be **nop**s, as shown in the example.)

The incrementer increments by one. Address increment rules are covered in Section 3.6.2.

3.6.2 Address Increment Rules

Send feedback

Logic built into the address increment function affects all usage cases. The effects of this logic are summarized in Table 3.6.

Within either the ROM or RAM address spaces, the carry propagates within the low 7 bits. At the 128-word boundaries within this address space, the incremented address will wrap back to the beginning of the page. Since there are 64 words of RAM and 64 words of ROM, RAM and ROM each appear in two different ranges. RAM is addressed from 0 to 63 and from 64 to 127. Likewise, ROM is addressed from 128 to 191 and from 192 to 255.

When Bit 8 of the address is 1, the address is in I/O space, and increment is suppressed. When a register is pointing to a port it stays pointing to that port. This means that instruction fetch, literal fetch, and literal store can be used when executing from a port. This is a common and useful programming technique. A **`call`** from a port will return back to the port.

Table 3.6 Address increment

9	8	7	6	5	4	3	2	1	0	Address Space	Increment Behavior
X	1	A	A	A	A	A	A	A	A	I/O Port	Increment is suppressed
X	0	1	X	A	A	A	A	A	A	ROM	Wrap to 080 from 0FF
X	0	0	X	A	A	A	A	A	A	RAM	Wrap to 000 from 07F

3.6.3 Extended Mode

Send feedback

Address increment does not affect Extended mode. P9 Is not changed by address incriment. To set P9 to 0 or 1, See Section 3.10.

3.7 Branch Opcodes

Send feedback

Branch opcodes change the location from which the next instruction is fetched. A branch will put the target address onto the address bus. This address is used to fetch the next instruction. The address is then incremented, per incrementation rules, and latched into the P register.

When executing from RAM or ROM, the value in the P register is never the location from which the currently-executing instruction word was fetched. (When executing from a port, the P register retains the address of the port.) This affects the available branch page for Slot 2 branch opcodes.

When executing out of RAM or ROM, the Address Incrementer has increased the address by at least one; sometimes by more than one, depending on the opcode sequence. This value is the one that will be latched into the P register to start the next fetch cycle.

In some cases the P register may have been incremented more than once. This occurs whenever an opcode in a previous slot within the same instruction word has executed a P register-relative opcode, such as a literal load or store. For example, both **`@p+`** and **`!p+`** will increment the P register and therefore change the location from which the next instruction will be fetched, as described in Section 3.6, Address Increment Rules.

Three examples are shown in Table 3.7, Branch Opcode Examples. In the first example, when the **`call`** executes, the P register has the value N+1 in it, and that value will be pushed on the stack.

In the second example, Slot 1 has a literal load opcode, **`@p+`**. The P register will have moved past this and contain the value N+2, which will be pushed onto the stack when the **`call`** executes.

In the third example, the two preceding literal loads have caused the P register to increment to N+3. Address computation for Slot-2 opcodes (as described previously) limits the opcode to the page pointed to by the P register *at the time the opcode executes.*

Table 3.7 Branch Opcode Examples

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
N											`call`					`3-bit address`		
N+1																		
N						`@p+`					`call`					`3-bit address`		
N+1	`literal`																	
N+2																		
N	`@p+`					`@p+`					`call`					`3-bit address`		
N+1	`literal`																	
N+2	`literal`																	
N+3																		

3.8 Understanding Opcode Timing

Send feedback

The C18 core is a fully asynchronous design. Signals propagate through combinatorial paths and are latched or otherwise used as needed. For example, the ALU is fed by the T and S registers and computes all possible arithmetic operations in parallel, using combinatorial logic. A given opcode (for example **`xor`**) selects the corresponding output from the ALU and the core moves on to the next opcode.

In general this is both invisible and irrelevant to the programmer, but an awareness of it makes it easier to understand certain aspects of C18 core operation, including zero test, addition opcodes, and 'Slot 4.'

3.8.1 Execution Times

Send feedback

Opcodes can be divided into two distinct sets: ALU and memory. (Refer to Table 3.1 and Table 3.2, Summaries of C18 Instruction Set, for details.) The ALU class opcodes (except for **+** and **+***) are the fastest, and execute in 1 unit. Other classes have access times as shown in Table 3.8.

To convert execution time units to nanoseconds, refer to the AC timing, Section 8.3.

Table 3.8 Execution times

Opcode Class	Opcode Subclass	Execution Time
ALU	ALU except add	**1 unit**
	Add – see Section 3.9.1, Addition Opcodes	**1 or 2**
Memory	I/O register access	**2* units**
	Port access (if ready)	**3* units**
	ROM or RAM access	**3.6* units**
	03, 05 data and address bus access	**3.5* units**
	36, 37, 39 ADC read access	**3.6* units**

* Times are approximate.

3.9 Prefetch

Send feedback

To maximize throughput, the C18 CPU begins an instruction fetch as soon as possible. As noted earlier, the Instruction register has four opcode slots, numbered Slot 0 to Slot 3. A slot multiplexer feeds the opcode in each slot, in turn, to the execution logic. When one of these opcodes is a memory opcode, that opcode will use the address bus; therefore, the next instruction fetch cannot begin until after the completion of all uses of the address bus by any instruction in the Instruction register.

If the last opcode in an instruction word affects the data bus, (see Table 3.9) there is no prefetch. Otherwise, prefetch will start as soon as possible. Once the instruction fetch has started, the time allotted to the

Slot 3 opcode will be extended until the instruction fetch completes. There are four cases of instruction fetch illustrated in Figure 3.2, Prefetch. In each case, the vertical red line shows the start of the instruction fetch operation, and the horizontal blue line shows the extension of Slot 3 that results.

In Case 1, if the memory bus is available before Slot 4, then it is possible to prefetch the next instruction. However instruction fetch must not replace the contents of the Instruction register before the last opcode in the Instruction register completes execution. Consequently, even if no opcodes reference memory, the prefetch begins at the the end of Slot 1.

In Case 2, the last memory opcode is in Slot 1. The instruction fetch will start at the beginning of Slot 2 and Slot 3 will be stretched by more than one extra opcode time.

In Case 3, the last memory opcode is in Slot 2. Slot 3 will be stretched to last as long as the entire fetch cycle.

In Case 4, there is a memory opcode in Slot 3. This memory opcode must be allowed to complete before the address bus is used for the instruction fetch. To allow for this there is a special Slot 4, with a hardwired **.** (NOP) that will be executed after slot three. There is no prefetch, and slot 4 is a full memory fetch time.

Table 3.9 Opcodes that affect prefetch

unext	**!a**	**!a+**	**@a**	**@p+**
!b	**!p+**	**@b**	**@a+**	

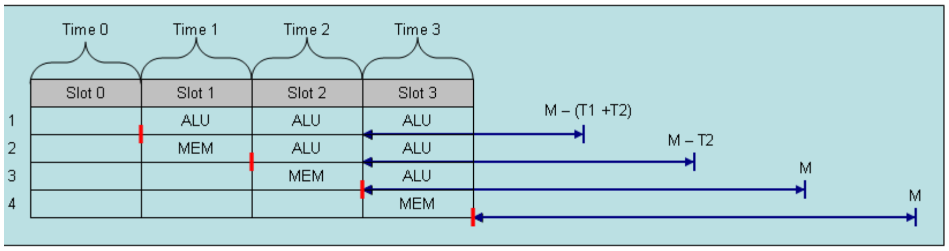


Figure 3.2 Prefetch Diagram

3.9.1 Addition Opcodes

Send feedback

+ and **+*** are the only two stack opcodes that may take more than one cycle to complete. The addition logic in the ALU, which is used by **+** and **+***, will propagate the carry across nine bits before the result is latched.

The ALU is asynchronous logic, and continuously recalculates all possible ALU results, updating every time T or S changes.

Since any change in T or S propagates carry nine bits in a single cycle, one extra cycle of settling time is required before the addition logic will be valid for an 18-bit result. In a single cycle the low-order nine bits are guaranteed. Note that addresses can be calculated in a single cycle.

+ and **+*** should not be used immediately after any opcode which affects the value of T or S. However, if there is an intervening opcode which does not affect T or S, the add operation will work correctly. These opcodes are marked in the following descriptions as 'Helps plus.'

If it is not possible to arrange the opcode sequence such that a helps-plus opcode occurs just before a **+** or **+***, a **nop** should be inserted. Whenever a **+** in Slot 3 is stretched by more than one extra opcode time (Cases 2 and 3 from Section 3.9, Prefetch) there will be enough time for the bits in the ALU to stabilize for a **+** or **+***, which is in Slot 3 to return correct results.

Whenever there is a Slot 4 nop (Case 4 from Section 3.9, Prefetch) there will be enough time for the bits in the ALU to stabilize for a **+** or **+***, which is in Slot 0 of the next instruction word to return correct results.

Exept when transitioning into Extended mode, a **+** in Slot 0 of an instruction word that is reached via any branch opcode will also have had time to settle, even in the fall-through case, because no transfers modify T.

3.10 Extended Arithmetic

Send feedback

The SEAforth 40C18 includes a feature to extend the instruction set for the purpose of making arithmetic on large numbers more efficient. Such arithmetic is required by, for example, cryptographic algorithms that involve modulo multiplication and exponentiation. Only **+** and **+*** are affected by Extended mode. See opcode description of each for details.

3.10.1 Mechanism

Send feedback

The P9 is not used for addressing memory as such; P9 is not decoded when referencing memory and or ports. However, P9 is carried in P, and is pushed to R and popped from R like any other bit of P.

When a **jump** or **call** is done with an address field smaller than 13 bits (i.e., other than Slot 0), P9 is *unchanged* by the **jump**. P9 is *always* set or cleared by a Slot 0 **call** or **jump**, or a **return** instruction, and is set to zero on reset for all normal node starting addresses.

3.10.2 Helps Plus

Send feedback

If a **jump**, **call**, **next**, **if**, **-if**, **;:**, or **return** instruction changes the state of P9 in either direction, that instruction must not be considered to assist addition in providing time for carry propagation. Instead, change in state of P9 resets all "help" because it is an event which can start a new carry propagation process in the ALU. Whether or not it does depends on the state of the ALU and on whether the carry into the ALU actually changes due to the event.

3.10.3 Carry Latch

Send feedback

There is a carry latch, which is defined as holding a true carry bit, 0 or 1. It is not initialized to any particular state on reset. The latch is only changed by **+**, and only when P9 is set, placing the core in Extended Mode.

3.10.4 Setting and Clearing Carry Latch

Send feedback

Examples of ways to obtain the current value of the carry latch:

```
0 # dup . + ( – carry bit)  \ Given 0 on top of the stack
                            \ returns carry, clears carry bit.
                            \ Returns carry bit as numeric 1 or 0.

dup dup not . + ( -n)  \ returns -1 if the carry was 0, 0 if the
                       \ carry was 1, and leaves the carry unchanged.
```

Examples of ways to clear and set the carry latch:

```
dup xor dup . + ( – carry bit)          \ Clears carry
dup xor not dup . +  ( - n)             \ Sets carry
```

`. +` Clear carry by adding any two throwaway *positive* values, or any others with sum <3FFFF.

`. +` Set carry by adding any two throwaway *negative* values, or any others whose sum is >3FFFF. These two boundary conditions are biased so that correct results do not depend on the prior state of the latched carry bit.

Chapter 4 Interprocessor Communications

4.1 C18 Ports

Send feedback

The SEAforth chip family uses a flexible mechanism to make it easy for the individual cores to communicate. Each C18 core has a communication port between it and each of its neighbors. These ports between cores are bidirectional, half duplex, asynchronous communication devices.

Ports act as a sort of mailbox between adjacent cores. Ports are explicitly addressed within the address space. Every core has two, three, or four direction ports, depending on location. Each direction port connects its core to the neighboring core. A direction port is not a storage element, but a data transfer element. When one core writes to a port, the binary values are asserted until the receiving core reads them, but they are not stored. They are merely asserted.

The purpose of a communication port is to move words of data and instruction from one core to another with the minimum possible overhead in individual transmissions. The data or instructions can be sent or recieved in successive transmissions.

4.2 Understanding Directions

Send feedback

These ports are mapped into memory space on common addresses. To understand how it works, it's helpful to first be clear on the terminology used by SEAforth to indicate direction.

By convention, the terms Right, Down, Left, and Up are used to describe directions, but adjacent cores always use the *same* direction to talk to their neighbors. For example, Node 00 and Node 10 always use their Down ports to talk to each other, regardless of the actual direction of the communication. The local directions and their port addresses are in Table 4.1, Interprocessor Ports.

Table 4.1 *Interprocessor Ports*

Label	Port (Blocking)
`Right`	`$1D5`
`Down`	`$115`
`Left`	`$175`
`Up`	`$145`

As shown in Figure 4.1 SEAforth Directional Definitions, nodes with *red* between them, connect to each other with their *right* ports. Nodes with *green* between them connect to each other with their *left* ports. Nodes with *blue* arrows between them connect to each other with their *down* ports. Nodes with *pink* between them connect to each other with their *up* ports. Edge nodes are missing one direction port, and corner nodes are missing two direction ports.

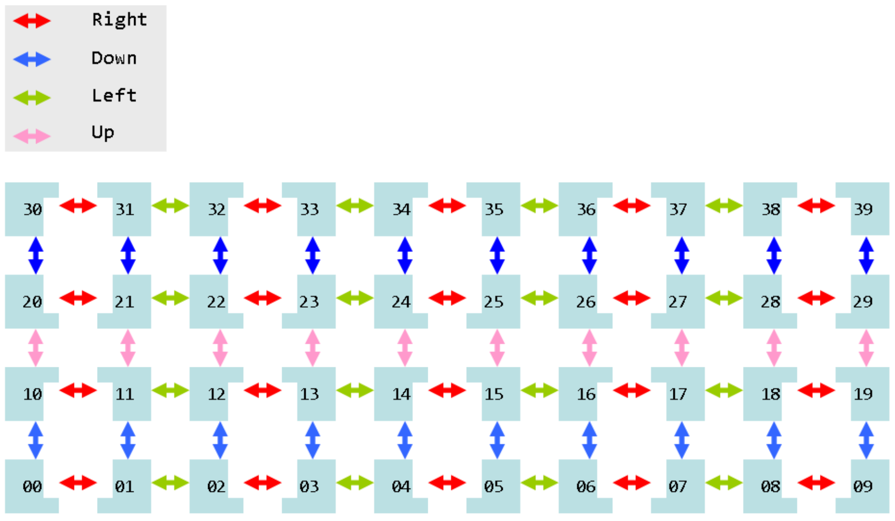


Figure 4.1 SEAforth Directional Definitions

The four direction ports are accessed by setting their corresponding bits. The four bits represents bits 7:4 of the address; bit 8 is always 1 for I/O, and the low nibble is always $5. When the bits are 0101, no port is selected. Flipping a bit selects its port. Multiple ports can be selected, as shown.

Table 4.2 Direction Ports

Name	R (bit 7)	D (bit 6)	L (bit 5)	U (bit 4)	Port	Result
`----`	0	1	0	1	$155	No port selected
`---U`	0	1	0	0	$145	Up
`--L-`	0	1	1	1	$175	Left
`--LU`	0	1	1	0	$165	Left, Up
`-D--`	0	0	0	1	$115	Down
`-D-U`	0	0	0	0	$105	Down, Up
`-DL-`	0	0	1	1	$135	Down, Left
`-DLU`	0	0	1	0	$125	Down, Left, Up
`R---`	1	1	0	1	$1D5	Right
`R--U`	1	1	0	0	$1C5	Right, Up
`R-L-`	1	1	1	1	$1F5	Right, Left
`R-LU`	1	1	1	0	$1E5	Right, Left, Up
`RD--`	1	0	0	1	$195	Right, Down
`RD-U`	1	0	0	0	$185	Right, Down, Up
`RDL-`	1	0	1	1	$1B5	Right, Down, Left
`RDLU`	1	0	1	0	$1A5	Right, Down, Left, Up

4.3 Interprocessor Read / Write

Send feedback

Each node shares up to four interprocessor communication ports with its neighbors. These ports between cores are bidirectional, half duplex, asynchronous communication devices. Neighbors share a single common port. Interprocessor communication is blocking and self-synchronizing. Each interprocessor communication port connects directly to its neighbor. There is no register or FIFO; one port's wires are connected directly to a neighbor's wires. When node A writes to node B, node A will block, or sleep, until node B performs its read. Likewise, if node B reads before node A has written, node B will block, or sleep, until node A performs its write.

In addition to providing interprocessor communication, this method synchronizes the two cores. Both nodes resume execution when the transfer is complete. Overall power consumption is reduced because a sleeping core consumes virtually zero power, and programming is simplified because the programmer doesn't have to create and execute complex semaphore mechanisms.

Blocking can be reduced by testing the status bits in the IOCS register before performing the read or write operation. You can choose to have the sender of information block and the recipient test the write-pending status bit, or you can choose to have the recipient block and the sender test the read-pending status bit.

The information passed through the ports can be either data or instructions. The core has the ability to directly execute instructions from ports by jumping to or calling a port address.

Port synchronization is achieved when, between two neighbor nodes, one is reading and the other writing. Should the neighbors both attempt to read or both write, the handshake for their common port will never complete, and both will hang until a hardware reset.

Writing to or reading from undefined addresses can have indeterminate results.

A core may read or write a single port, or it may read or write with multiple ports selected.

4.3.1 Single Port Operations

Send feedback

Software in the two communicating cores must be designed to start in the correct direction and to remain in agreement as to direction of communication thereafter.

Whichever core reads or writes a port first is suspended, until the other core writes or reads it.

There are no race conditions possible in single port operations, because of the positive interlock in which both cores must see both read and write lines high, in order to acknowledge the port operation and continue execution.

4.3.2 Multiport Reads

Send feedback

Two, three, or four selected ports may be read at the same time.

An otherwise idle core can make its services available to multiple neighbor cores, by executing a multiport read; for example to **`r-l-`**. This read request will be satisfied when the first of its neighbors to need it responds with a write request. Care need only be taken that two neighbors do not simultaneously respond, and that multiple requests are not posted before the core executes the multiport read.

This is supported and works without race problems, so long as the neighbors cooperate or coordinate such that only one of them writes to a core that is waiting (or planning to wait) on a multiport read.

4.3.3 Multiport Writes

Send feedback

Two, three, or four selected ports may be written at the same time. Like the multiport read, only intended uses are supported. The write completes when at least one of the ports is read at the other end. By design, the port handshake lines (both read and write) for all selected ports will be pulled low by each node on completion of any port read or write.

Chapter 5 External Memory and I/O

5.1 External Memory

Send feedback

Node 03, Node 04, and Node 05 may be used together to access external memory. Node 04's IOCS register controls four I/O pins as well as wakeup lines to the UP port of Node 03 (bit 9) and Node 05 (bit 7). The value put into those bits is the value 'seen' in either Node 03 or Node 05.

Node 03 and Node 05 each have an 18-bit bidirectional bus connected to their UP port, which may be accessed using either the UP address, in which case the node sleeps until it receives a wakeup handshake from Node 04, or the DATA address, which does not cause the node to sleep. By default, the node will complete its handshake (to UP) when Node 04 sets these bits high, but this is controllable (in the same way as all top and bottom edge nodes control their pin wakeup) by the value in IOCS bit 11.

IOCS bit 12 in Nodes 03 and 05 controls the direction of that node's bus pins as a group. When set high, the pad drivers for each bit are enabled and the pad is prevented from setting the data latch. When bit 12 is low, the drivers are off, and the pad latch is set from the data at the pad. Note that reading the UP or DATA port delivers data from the latch, which may differ from the data actually at the pad should the internal pad drivers be in contention with external drivers (and IOCS bit 12 is high to enable the drivers). Writes to the bus pad registers will will not change the pad, if the pad is in input mode. Therefore, the correct direction of the pad must be set before a reliable read or write may be performed.

Note that when reading from the UP address, the data on the pins when the node wakes is the value captured. When writing to the UP address, however, the value is latched in the output register when the node begins to sleep, so it is output while asleep, continuing after being awakened until either a new value is written or the output drivers are turned off.

5.2 Digital I/O

Send feedback

Cores with a single I/O pin can use that pin for any desired input or output function. For input use, the pin can be programmed as a floating input or a weakly pulled down input. For output, it can be programmed to Vdd or Vss out. Refer to the DC specifications section for details on voltage levels and currents.

5.3 Pin Wakeup

Send feedback

All cores with external pins can be awakened from read or write to the LEFT or UP port with no neighbor by an I/O assertion on the pin associated with bit 17 of IOCS of that core. The same mechanism and port addresses are used. Refer to Figure 4.1, SEAforth Directional Definitions, you will see that for cores on the edge, either the UP port, the LEFT port, or both are always the ones *not* involved in interprocessor communication with a neighbor.

The wakeup pin can be set to be active high or active low on nodes in the top or bottom row; side nodes wake up on high only. Wakeup level is controlled by bit 11 in the node's IOCS register, and defaults to high at reset. Setting bit 11 to low requests active low wakeup.

5.3.1 Digital Pin Wakeup

Send feedback

On digital pins, the ability to detect a wakeup condition depends on the port address decode changing. As a result, consecutive reads of the pin port should be done in different instruction words.

If two pin reads are done in the same instruction word without otherwise changing the address bus, and the pin state has not changed, the detection circuit will fail to see the necessary condition the second time around, and the core will wait for the pin value to change away from and then back to the wakeup state.

5.3.2 Analog I/O Node Pin Wakeup

Send feedback

On the three analog cores, the ability to detect a wakeup condition does not depend on address decode changing; therefore, it is legal to have multiple ADC register reads within the same instruction word.

5.4 Analog Cores

Send feedback

Nodes 36, 37 and 39 act as analog to digital and digital to analog conversion devices and have analog in and analog out pins.

5.4.1 Analog to Digital Conversion

Send feedback

Analog to digital conversion is done with a voltage-controlled oscillator and a counter. The VCO output is defined in units of counts per millivolt per nanosecond. For example, at 400 mV in, the counter will advance approximately 6.6 counts per nanosecond. The recommended operating range for the ADC is 400 mV to 1500 mV. A graph of the transfer function is shown in Figure 5.1.

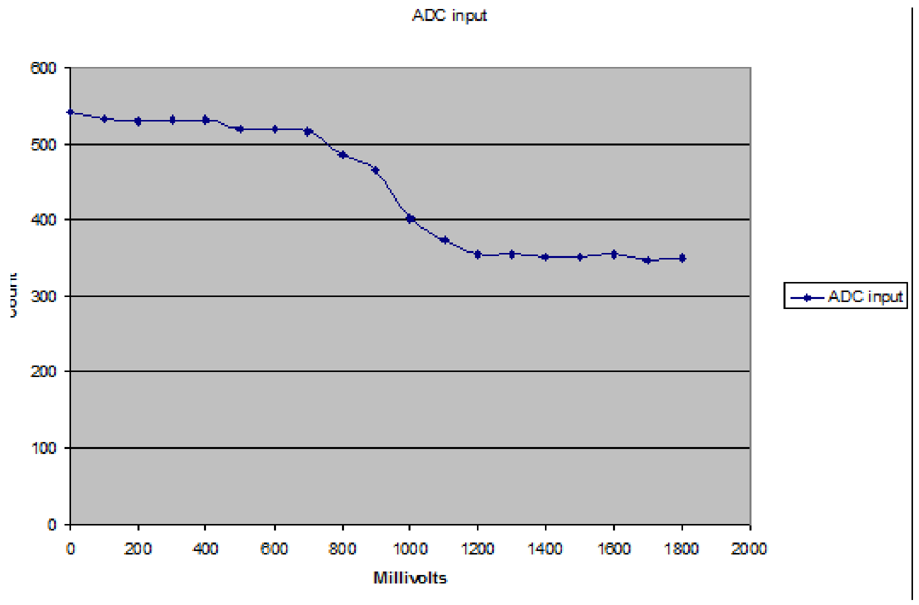


Figure 5.1 ADC Count vs. Input Voltage (Typical)

Conversion is done by reading the counter bits. To read from ADC send data to DATA ($141) then fetch from DATA. The counter counts up, and the difference between successive counter reads gives the speed of the oscillator. From the speed of the oscillator, the input voltage can be calculated. Note that the VCO runs more slowly for higher voltages.

The folowing sample code, takes two samples from the VCO and subtracts them. (The sample code assumes A has the address of the ADC register.)

```
dup !a @a not dup!a @a      \ read oscillator twice
. + not                     \ subtract from last sample
```

Note that this technique allows high precision at a slow sample rate, or a fast sample rate at reduced precision. The approximate time required to reach the full 18-bit count is shown in Table 5.1.

Table 5.1 ADC Time to Max Count

Input voltage	Time to reach full count (262,144)	units
400 mV	40	µS
1500 mV	75	µS

For maximum overall processing speed, the difference calculation and any linearization may be done by a neighbor processor, allowing the ADC processor to sample more frequently.

Writing to IOCS register bits 14 and 13 turns a Voltage Controlled Oscillator on or off as shown in Table 5.2.

Table 5.2 Control Bits for ADC Operation

OPN	Result
0 0	VCO on pin input
0 1	VCO on vdd input for calibration
1 0	VCO off
1 1	VCO on vss input for calibration

5.4.2 Digital to Analog Conversion

Send feedback

Digital to analog conversion is done via a binary-weighted array of current sources. The DAC has a current-mode output. Figure 5.2 shows curve of DAC output with varying loads.

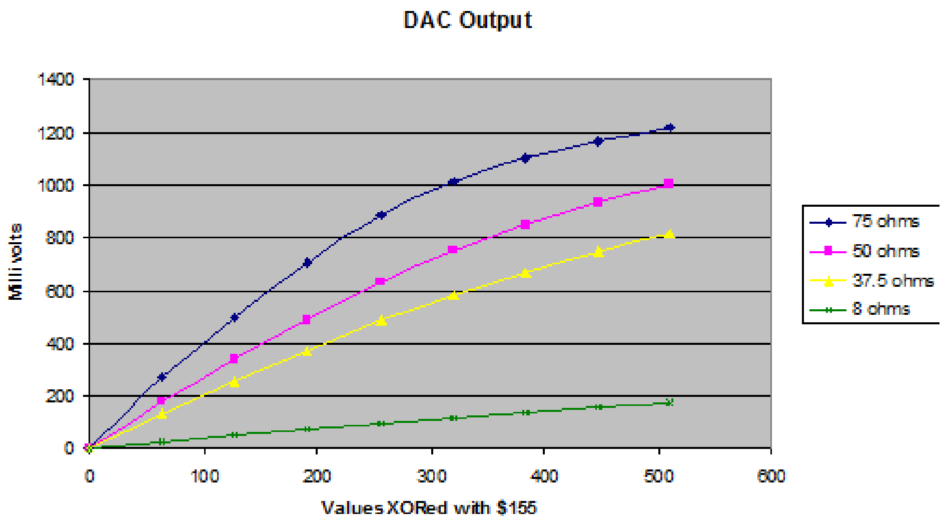


Figure 5.2 DAC Output

On Nodes 36, 37 and 39, the IOCS register contains a field for the DAC output value. The 9-bit digital value to be converted should be XORed with $155 and written to bits B8-B0. Figure 5.2 shows curve of DAC output.

5.5 Serdes

Send feedback

Serdes is short for a high-speed SERializer/DESerializer that communicates 18-bit data and/or instructions between chips. It is implemented as a register that can be stored into, fetched from and shifted left. Note that the serdes hardware provides a serial bit stream that can be used for purposes other than communication, for instance, as a function generator.

There are two pins required: clock and data. The clock pad is input-output. Output is from a ring-oscillator located at the pad, which currently runs at ~400 MHz. Input is from the clock of a neighboring chip. The data pad is also input-output. Output is from bit 17 of the serdes register. Input is to bit 0. Input vs. output of both pads is specified (as usual, high for output) by bit 17 of IOCS.

5.5.1 Operation

Send feedback

To send:

- Write the 1st word to address DATA.
- Turn on clock by writing $20000 to IOCS.
- Write remaining words to address UP, which waits until the previous word is sent.
- Turn off clock by writing 0 to IOCS after appropriate delay. Each word requires 19 clocks. Missing or extra clocks will confuse the receiver.

To receive:

- Write $3FFFE to address DATA. Leave $3FFFE on the stack.
- Read words from address UP, which waits until words are available. Store them away so that $3FFFE remains at top of stack. It counts the 18 bits shifted in.

To execute code:

- Write $3FFFE as above.
- **`jump`** to UP.
- Each instruction word must leave $3FFFE on the stack when returning to read the serdes.

Chapter 6 I/O Register Detail Descriptions

The IOCS (I/O Control and Status Register) is found at location $15D on each processor. The IOCS provides the current status of each core's shared wake/sleep communication ports. Reading from this location returns input and status information. Writing to this location controls I/O pins on nodes with pins.

6.1 Reading Port Status from IOCS

Send feedback

Reading the IOCS will return bits indicating the presence of pending reads or writes. Bits 16:9 contain four 2-bit fields in the canonical order RDLU (Right-Down-Left-Up). Each 2-bit field contains a bit indicating whether the neighbor is waiting, and a bit indicating whether the neighbor is waiting while executing a port write. Bit field definitions are shown in Table 6.1. IOCS Interprocessor Port Status Definitions.

Note that the pending read bits are active low, and the pending write bits are active high. Read requests by sleeping RDLU neighbors are read as '0' in bit 16, bit 14, bit 12, and bit 10 respectively, in the IOCS register. Write requests by sleeping neighbors are read as '1' in bit 15, bit 13, bit 11, and bit 9 respectively, in the IOCS register. The mnemonic RDLU may be used to remember which bits refer to which ports; R (Right) is 16/15; D (Down) is 14/13; L (Left) is 12/11; and U (Up) is 10/9.

Nodes at edges and corners do not have LEFT and/or UP neighbors. Reads from IOCS in these directions return the *inverse* of the data written to the register bit. This allows some nodes (ones with nothing to the LEFT) to read their wakeup state.

Table 6.1 IOCS Interprocessor Port Status Definitions

Bit Position	16	15	14	13	12	11	10	9
Direction	Right		Down		Left		Up	
Status	$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW	$\overline{\text{UR}}$	UW

6.2 Bit I/O with IOCS

Send feedback

In addition to providing the status of interprocessor communication ports, the IOCS register is used for pin I/O on most edge and corner nodes. There may be up to four I/O pins, which are readable on bits 17, 1, 3, and 5. Note that a high voltage on the pin will be read as a ‘1’.

A write operation will set the state of I/O pins, if such pins are available for the given core. Bits are configured as input or output using a pair of control bits. In all cases, the function of the bits is as shown in Table 6.2, I/O Pin Configuration Control Bits. Reading a pin that is outputting a value will return the pin value. If an external device overpowers or slows down the value set internally, then the value read may be different from the value written.

When the MSB of the control bit pair is 0, the pin drivers are disabled, except for an optional weak pull-down. When the MSB is 1, the pin drivers are enabled, and the LSB is driven onto the pin. A ‘1’ produces a logic-high voltage.

Table 6.2 I/O Pin Configuration Control Bits

Data Ctrl (C)	Data Ctrl (D)	Function
0	0	Input
0	1	Weak pull-down
1	0	Output Vss
1	1	Output Vdd

6.3 Node Tables Legend

Send feedback

Section 6.4 contains tables describing the IOCS pins for each node on the chip. Each of the 40 cores has its own table describing both internal and external I/O controlled through IOCS. The tables are descibed in terms of bits of IOCS. This section provides the keys for reading the tables.

6.3.1 Input Status (Read)

Send feedback

Input Status describes the representation of bits read from the IOCS. There are two groups of significant bits:

- **Neighbor status:** bits 9 through 16 are reserved for the status of neighbors. On edge cores, not all of these bits are defined.
- **Pin Status:** Odd pins 1 through 7 and 17 are reserved for reading from data pins DX, where X represents the pin number.

  All other input bits are reserved for future use.

6.3.2 Neighbor Node.

Send feedback

The Neighbor Node is the node to which the corresponding bits of the IOCS are attached. When you read the IOCS, it is reporting the status of its neighbor node(s).

6.3.3 Output Control

Send feedback

Output to the IOCS can control specific pins, or serve to wake up the attached neighbor node.

- **Pin Control:** When writing to IOCS, the Output Control bits control both the mode and the output of the pin. CX, (where X represents the bit in IOCS that the pin is read from) represents the mode control for the output of that pin. DX (where X represents the bit in IOCS that the pin is read from) represents the data output from that pin. See Table 6.2 for pin configuration.
- **Wakeup Level Control:** All nodes with pins in the top and bottom row have a Wake Direction (WD) in IOCS bit 11, noted in the Node Tables as WD. The WD sets the node's pin state for wakeup. When sleeping on pin 17, the node wakes up if state in bit 11 is same as input from bit 17.

6.3.4 Analog Nodes

Send feedback

Bits 0 through 8 of the IOCS are reserved for DAC output. All values sent to the DAC must be XORed with $155 before being written to the IOCS.

See Section 5.4.1 for details of ADC usage.

6.3.5 SERDES

Send feedback

Bit 17 controls SERDES on/off. A value of 1 turns it on, while 0 turns it off.

6.3.6 Memory and Address

Send feedback

- **Output Control:** **bd** is the bus direction. A value of 1 specifies output; 0 is input.
- **Data/Address Bus:** When in input mode, bits in the data register represent bits read from pins. When in output mode, bits in data register represent bits written to pins.

  Note that it is not possible to read from a data register when in output mode, and it is not possible to write to a data register when in input mode.
- **Read only pins:** Nodes 03 and 05 have read-only bits at bit 17 of IOCS. On reset, these read-only pins read back as 1. This is the same as writing 0 into bits 9 and 7 respectively of Node 04 IOCS, which is the reset state for Node 04.

  Writing 1 into Node 04 IOCS bit 9 causes bit 17 of Node 3's IOCS to read back as zero. Writing 1 into Node 04's IOCS bit 7 causes bit 17 of Node 05 IOCS to read back as zero.

On reset, the WLB wakeup polarity bit in Nodes 03 and 05 is in the same state as it would be by writing zero into WLB. In this state, Nodes 03 and 05 will wake up when their bits 17 read back as zero, which is accomplished by writing 1 into Node 04's appropriate IOCS bits. If, on the other hand, WLB is written as 1 by Nodes 03 or 05, they will wake up in the reset state of their pin 17s.

6.4 Node Tables

Send feedback

Node 00	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW													
Neighbor Node		Node 01		Node 10														
Output Control																		

Node 01	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW											
Neighbor Node		Node 00		Node 11		Node 02												
Output Control	SER DES						wlb											

For serdes usage refer to Section 5.5.

Node 02	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW											
Neighbor Node		Node 03		Node 12		Node 01												
Output Control																		

Node 03	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW											
Neighbor Node		Node 02		Node 13		Node 04												
Output Control						bd*	wlb											

*bd, bus direction. 1 is output. 0 is input.

Data Bus	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Data Register	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

For external memory usage refer to Section 5.1.

Node 04	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW		D9		D7		D5		D3		D1	
Neighbor Node		Node 05		Node 14		Node 03			To Node 03		To Node 05							
Output Control	C17	D17					wlb		$\overline{\text{D9}}$ *		$\overline{\text{D7}}$ *		C5	D5	C3	D3	C1	D1

* For usage of D9 and D7 see Section 6.3.6

Node 05	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW											
Neighbor Node		Node 04		Node 15		Node 06												
Output Control						bd*	wlb											

*bd, bus direction. 1 is output. 0 is input.

Addr Bus	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Data Register	D17	D16	D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0

For external memory usage refer to Section 5.1.

Node 06	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW											
Neighbor Node		Node 07		Node 16		Node 05												
Output Control																		

Node 07	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW											
Neighbor Node		Node 06		Node 17		Node 08												
Output Control																		

Node 08	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW											
Neighbor Node		Node 09		Node 18		Node 07												
Output Control																		

Node 09	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW													
Neighbor Node		Node 08		Node 19														
Output Control																		

Node 10	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW			$\overline{\text{UR}}$	UW								D1	
Neighbor Node		Node 11		Node 00				Node 20										
Output Control	C17	D17															C1	D1

Node 11	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW	$\overline{\text{UR}}$	UW									
Neighbor Node		Node 10		Node 01		Node 12		Node 21										
Output Control																		

Node 12	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW	$\overline{\text{UR}}$	UW									
Neighbor Node		Node 13		Node 02		Node 11		Node 22										
Output Control																		

Node 13	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW	$\overline{UR}$	UW									
Neighbor Node		Node 12		Node 03		Node 14		Node 23										
Output Control																		

Node 14	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Sta-tus (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW	$\overline{UR}$	UW									
Neighbor Node		Node 15		Node 04		Node 13		Node 24										
Output Control																		

Node 15	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW	$\overline{UR}$	UW									
Neighbor Node		Node 14		Node 05		Node 16		Node 25										
Output Control																		

Node 16	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW	$\overline{UR}$	UW									
Neighbor Node		Node 17		Node 06		Node 15		Node 26										
Output Control																		

Node 17	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW	$\overline{UR}$	UW									
Neighbor Node		Node 16		Node 07		Node 18		Node 27										
Output Control																		

Node 18	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW	$\overline{UR}$	UW									
Neighbor Node		Node 19		Node 08		Node 17		Node 28										
Output Control																		

Node 19	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{RR}$	RW	$\overline{DR}$	DW			$\overline{UR}$	UW								D1	
Neighbor Node		Node 18		Node 09				Node 29										
Output Control	C17	D17															C1	D1

Node 20	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{RR}$	RW	$\overline{DR}$	DW			$\overline{UR}$	UW								D1	
Neighbor Node		Node 21		Node 30				Node 10										
Output Control	C17	D17															C1	D1

Node 21	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW	$\overline{UR}$	UW									
Neighbor Node		Node 20		Node 31		Node 22		Node 11										
Output Control																		

Node 22	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW	$\overline{UR}$	UW									
Neighbor Node		Node 23		Node 32		Node 21		Node 12										
Output Control																		

Node 23	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW	$\overline{UR}$	UW									
Neighbor Node		Node 22		Node 33		Node 24		Node 13										
Output Control																		

Node 24	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW	$\overline{UR}$	UW									
Neighbor Node		Node 25		Node 34		Node 23		Node 14										
Output Con-trol																		

Node 25	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW	$\overline{\text{UR}}$	UW									
Neighbor Node		Node 24		Node 35		Node 26		Node 15										
Output Control																		

Node 26	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW	$\overline{\text{UR}}$	UW									
Neighbor Node		Node 27		Node 36		Node 25		Node 16										
Output Control																		

Node 27	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW	$\overline{\text{UR}}$	UW									
Neighbor Node		Node 26		Node 37		Node 28		Node 17										
Output Control																		

Node 28	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW	$\overline{\text{UR}}$	UW									
Neighbor Node		Node 29		Node 38		Node 27		Node 18										
Output Control																		

Node 29	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW			$\overline{\text{UR}}$	UW								D1	
Neighbor Node		Node 28		Node 39				Node 19										
Output Control	C17	D17															C1	D1

Node 30	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW													
Neighbor Node		Node 31		Node 20														
Output Control																		

Node 31	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW											
Neighbor Node		Node 30		Node 21		Node 32												
Output Control	SER DES						wlb											

For serdes usage refer to Section 5.5.

Node 32	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW						D5		D3		D1	
Neighbor Node		Node 33		Node 22		Node 31												
Output Control	C17	D17					wlb						C5	D5	C3	D3	C1	D1

Node 33	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW										D1	
Neighbor Node		Node 32		Node 23		Node 34												
Output Control	C17	D17					wlb										C1	D1

Node 34	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW											
Neighbor Node		Node 35		Node 24		Node 33												
Output Control																		

Node 35	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)		$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW											
Neighbor Node		Node 34		Node 25		Node 36												
Output Control																		

Node 36	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{RR}$	RW	$\overline{DR}$	DW	$\overline{LR}$	LW											
Neighbor Node		Node 37		Node 26		Node 35												
Output Control							wlb			DAC value. Value must be XORed with $155								

Data Port	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Read Action	ADC Value – current contents of VCO-driven 18-bit counter.																	

For analog node usage refer to Appendix 5.4.

Node 37	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW											
Neighbor Node		Node 36		Node 27		Node 38												
Output Control							wlb			DAC value. Value must be XORed with $155								

Data Port	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Read Action	ADC Value – current contents of VCO-driven 18-bit counter.																	

For analog node usage refer to Appendix 5.4.

Node 38	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	$\overline{\text{RR}}$	RW	$\overline{\text{DR}}$	DW	$\overline{\text{LR}}$	LW											
Neighbor Node		Node 39		Node 28		Node 37												
Output Control	C17	D17					wlb											

Node 39	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Input Status (Read)	D17	RR	RW	DR	DW													
Neighbor Node		Node 38		Node 29														
Output Control							wlb			DAC value. Value must be XORed with $155								

Data Port	**17**	**16**	**15**	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Read Action	ADC Value – current contents of VCO-driven 18-bit counter.																	

For analog node usage refer to Appendix 5.4.

Chapter 7 Pinout and Package

The SEAforth 40C18 is packaged in a 10 x 10 mm 88-pin QFN. There are 88 connections. See Appendix 7.1 for list of Powerup pins. I/O pins, their signals, and their functions are listed in Table 7.2. The pinouts are also shown in Figure 7.1.

7.1 Power-On and Resets

Send feedback

Application of power to the SEAforth 40C18 puts all digital I/O pins into a passive state, with a weak pull-down. The reset pin resets the core and its I/O pin states.

The reset line is active low. It should be low and be held low during power up. The reset line may be released to inactive (high) state after the minimum delay prescribed in Electrical Specifications. Both cores and I/O interfaces are reset by this signal.

The chip may be reset at any time by asserting the reset line low for the minimum time prescribed in Electrical Specifications.

Table 7.1 Powerup pins

Signal Name	Pins
vddc	17, 49, 62
vssc	18, 50, 61
vddi	4, 19, 28, 40, 47, 64, 82
vssi	5, 20, 29, 41, 48, 63, 83
vdda	74
vssa	75

Table 7.2 SEAforth 40C18 I/O Pin Signal List

Signal Name	Function	Pin	Pin Type	Node	Control Register	Power-up State
d00	Memory Data	1	I/O	03	up	output low
d01	Memory Data	2	I/O	03	up	output high
d02	Memory Data	3	I/O	03	up	output low
do	Digital I/O	6	I/O	20	IOCS	weak pulldown
di	Digital I/O	7	I/O	20	IOCS	weak pulldown
d03	Memory Data	8	I/O	03	up	output high
d04	Memory Data	9	I/O	03	up	output low
d05	Memory Data	10	I/O	03	up	output high
d06	Memory Data	11	I/O	03	up	output low
d07	Memory Data	12	I/O	03	up	output high
d08	Memory Data	13	I/O	03	up	output low
di	Digital I/O	14	I/O	10	IOCS	weak pulldown
do	Digital I/O	15	I/O	10	IOCS	weak pulldown
d09	Memory Data	16	I/O	03	up	output high
d10	Memory Data	21	I/O	03	up	output low
d11	Memory Data	22	I/O	03	up	output high
d12	Memory Data	23	I/O	03	up	output low
d13	Memory Data	24	I/O	03	up	output high
d14	Memory Data	25	I/O	03	up	output low
do	Digital I/O	26	I/O	01	IOCS	weak pulldown
di	Digital I/O	27	I/O	01	IOCS	weak pulldown
d15	Memory Data	30	I/O	03	up	output high
d16	Memory Data	31	I/O	03	up	output low
d17	Memory Data	32	I/O	03	up	output high
di	Digital I/O	33	I/O	04	IOCS	weak pulldown
do	Digital I/O	34	I/O	04	IOCS	weak pulldown

Table 7.2 SEAforth 40C18 I/O Pin Signal List *(continued)*

Signal Name	Function	Pin	Pin Type	Node	Control Register	Power-up State
do	Digital I/O	35	I/O	04	IOCS	weak pulldown
do	Digital I/O	36	I/O	04	IOCS	weak pulldown
a17	Memory Address	37	I/O	05	up	output high
a16	Memory Address	38	I/O	05	up	output low
a15	Memory Address	39	I/O	05	up	output high
a14	Memory Address	42	I/O	05	up	output low
a13	Memory Address	43	I/O	05	up	output high
a12	Memory Address	44	I/O	05	up	output low
a11	Memory Address	45	I/O	05	up	output high
a10	Memory Address	46	I/O	05	up	output low
di	Digital I/O	51	I/O	19	IOCS	weak pulldown
do	Digital I/O	52	I/O	19	IOCS	weak pulldown
a09	Memory Address	53	I/O	05	up	output high
a08	Memory Address	54	I/O	05	up	output low
a07	Memory Address	55	I/O	05	up	output high
a06	Memory Address	56	I/O	05	up	output low
a05	Memory Address	57	I/O	05	up	output high
a04	Memory Address	58	I/O	05	up	output low
di	Digital I/O	59	I/O	29	IOCS	weak pulldown

Table 7.2 SEAforth 40C18 I/O Pin Signal List *(continued)*

Signal Name	Function	Pin	Pin Type	Node	Control Register	Power-up State
do	Digital I/O	60	I/O	29	IOCS	weak pulldown
a03	Memory Address	65	I/O	05	up	output high
a02	Memory Address	66	I/O	05	up	output low
a01	Memory Address	67	I/O	05	up	output high
a00	Memory Address	68	I/O	05	up	output low
ai	Analog Input	69	I/O	39	up	high impediance
ao	Analog Output	70	I/O	39	IOCS	high impediance
di	Digital I/O	71	I/O	38	IOCS	weak pulldown
ao	Analog Output	72	I/O	37	IOCS	high impediance
ai	Analog Input	73	I/O	37	up	high impediance
ai	Analog Input	76	I/O	36	up	high impediance
ao	Analog Output	77	I/O	36	IOCS	high impediance
di	Digital I/O	78	I/O	33	IOCS	weak pulldown
do	Digital I/O	79	I/O	33	IOCS	weak pulldown
do	Digital I/O	80	I/O	32	IOCS	weak pulldown
do	Digital I/O	81	I/O	32	IOCS	weak pulldown
do	Digital I/O	84	I/O	32	IOCS	weak pulldown
di	Digital I/O	85	I/O	32	IOCS	weak pulldown
di	Digital I/O	86	I/O	31	IOCS	weak pulldown
do	Digital I/O	87	I/O	31	IOCS	weak pulldown
rs	reset	88	Input active low	none		

IOCS resets to 15555 (as number) which can be used to predict most of the following.

- Serdes resets to SR (IOCS.17) 0, meaning receive, and then executes-ROM at $0AA which finishes initializing the serdes to receive 18 bitwords. Both serdes pins (1 and 17, data and clock) are high impedance when receiving.

- Analog out (ao) pins: The DAC's reset to logical zero (155 in IOCS) which, in effect, means high impedance. The DAC is a current source from Vdd(a) and when logically zero no current is being sourced. It is designed to drive a resistive load and there is no internal pull down. (high impedance).
- Analog in (ai) pins: The A/D converters reset to “VCO OFF” state (10 in IOCS.14,13) to conserve power. A/D input pins are always high impedance.
- Nodes 3 and 5 buses: WD (IOCS.12) resets 1 meaning OUTPUT mode The output pins reset to 2aaaa meaning that pin a0 is Vss, a1 is Vdd, etc.

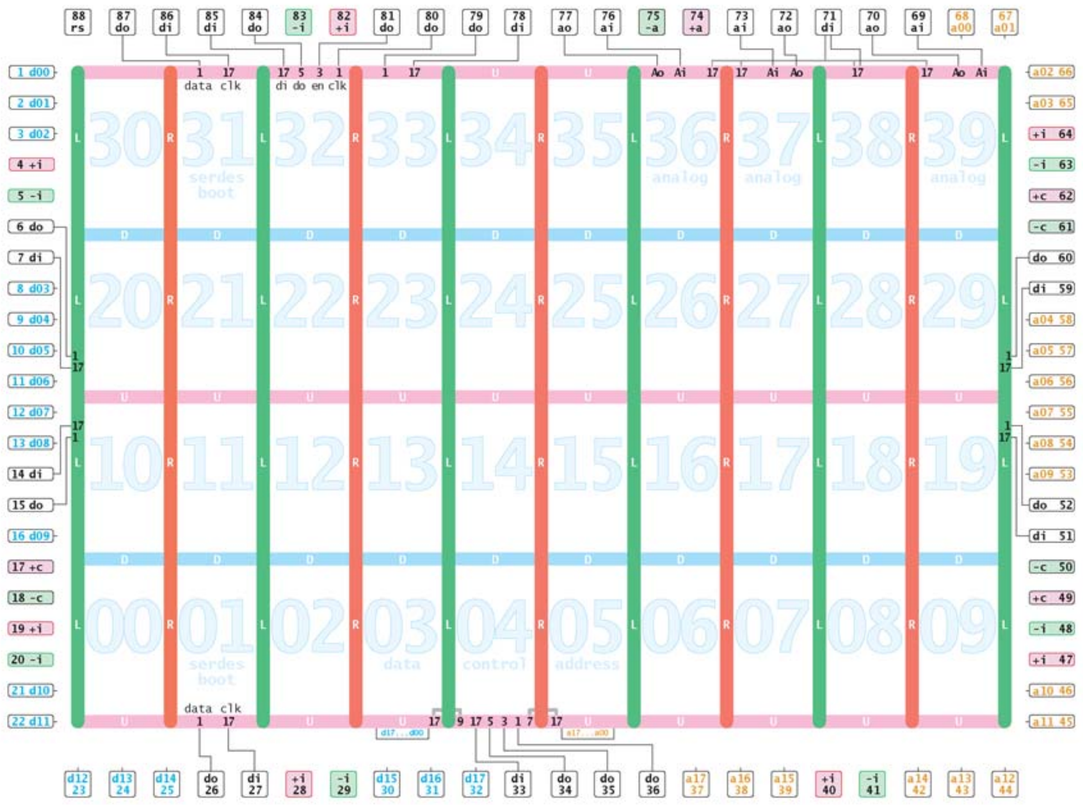
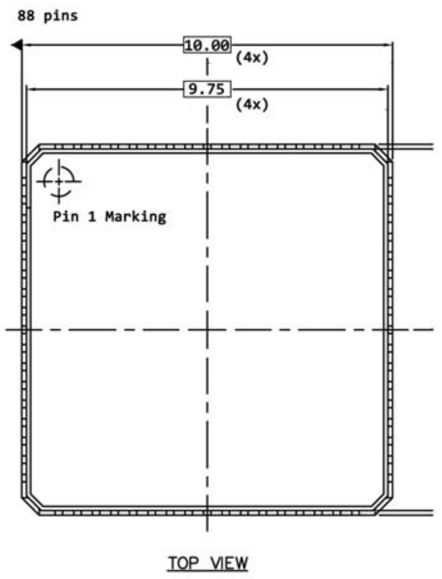


Figure 7.1 Package Outline Drawing



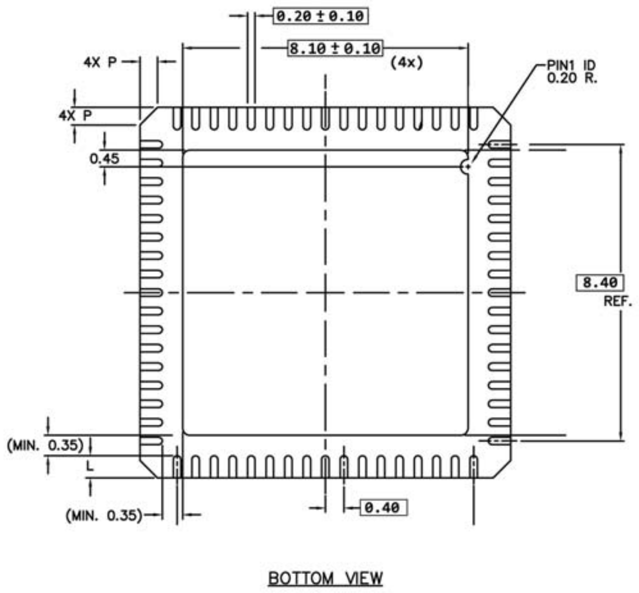


Figure 7.2 Package, front and back view

All dimensions in millimeters.

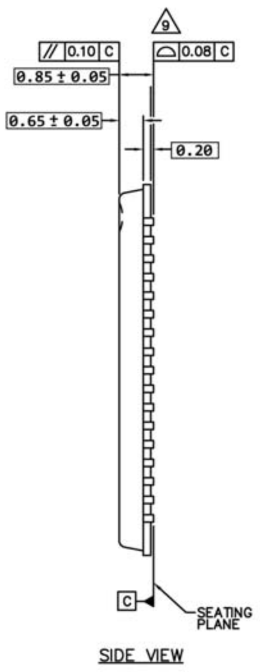


Figure 7.3 Package, side view

All dimensions in millimeters.

Chapter 8 Electrical Specifications

All numbers in Chapter 8 are preliminary and should not be used as the basis for detailed designs. As soon as The S40C18 7G electrical specifications have been measured, this data sheet will be updated.

8.1 DC Specifications

Send feedback

Recommended and maximum DC operating conditions are listed in Appendix 8.1, Appendix 8.2, and Appendix 8.3.

Table 8.1 Absolute Maximum Ratings

Symbol	Description	Min.	Max.	Unit
VDDC	Core	-0.3	2.0	V
VDDI	I/O Power	-0.3	2.0	V
Vesd	Maximum ESD Stress Voltage (3 stresses maximum)		2000	V
Ieos	Maximum DC Input Current (electrical over-stress) for any non-supply pin (Note 1)		5	mA
Tstorage	Maximum Storage Temperature	-40	125	C

Note 1: Total current must not raise VDDI above 2 V.

Table 8.2 Voltage and Temperature Operating Conditions

Symbol	Description	Min.	Nom.	Max.	Unit
VDDC	Core	1.65	1.8	1.95	V
VDDI	I/O Power	1.65	1.8	1.95	V
Tcase	Package Operating Temperature, Commercial Version	0		70	°C
Tcase	Package Operating Temperature, Extended Version	-20		+85	°C

Table 8.3 Device Characteristics

Symbol	Description	Min.	Typ.	Max.	Units
CPIN	I/O Pin Capacitance		2.5		pF
IIH	Input leakage current (Note 1)			10	µA
IOZ	Leakage current to ground, output tristateone pin with 1.8v on pin (room temp)		0.1		µA
IOPD	Max current to ground, weak pull-down one pin with 1.8v on pin (room temp)		41		µA
VIH	Input high voltage	0.8xVDDI		VDDI+0.3	V
VIL	Input low voltage	-0.3		0.2xVDDI	V
ISH	Max current sourced in saturation (room temp)		22.3		mA
ISL	Max current sunk in saturation (room temp)				

8.2 Analog Specifications

Send feedback

Analog Performance Characteristics for the ADC and DAC are listed in Appendix 8.4. Note that the ADC is a variable resolution device, with a maximum resolution of 18 bits.

Table 8.4 Analog Characteristics

Symbol	Description	Typ.	Units
VINA	ADC input voltage	0.4 -1.25	V
	1.8V in ADC running all cores asleep	2.4*	mA
	0.0V in ADC running all cores asleep	1.3*	mA
	DAC 75 ohm load max output 1.25V out	17	mA
	A/D Counter with 1400mV in	6.6	Cnts/nS
	A/D Counter with 400mV in	16.4	Cnts/nS

8.3 AC Specifications

Send feedback

The SEAforth 40C18 is an asynchronous part, and as such has no specific clock speed. The I/O pin toggle rate is the measure of device speed based on a software loop which toggles an I/O pin as rapidly as possible.

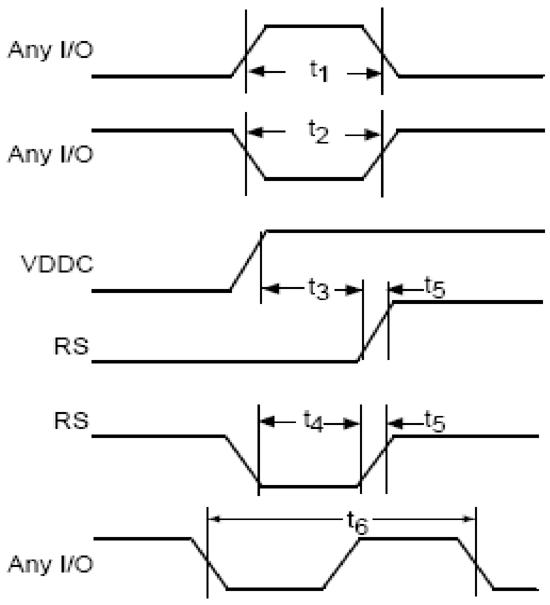


Figure 8.1 AC Timing Diagrams

Table 8.5 AC Timing Specifications

Symbol	Description	Min.	Max.	Units
t1	Minimum detectable high pulse, any I/O pin	20*	34*	nS
t1	Min. detectable high pulse using Wake up	9.8*	10.4*	nS
t2	Min. time to read low input	20*	34*	nS
t2	Min. time to read low input using wake up	9.8*	10.4*	nS
t3	Delay from VddC stable to end reset	200**	34*	nS
t4	Reset duration	200**	34*	nS
t5	Reset rise time	1	***	mS
t6	I/O pin toggle rate		90	MHz

* Includes time for code to change bit 0 pad, when input is bit 17 pad

** Internal reset only reqires less than 10 nS. However, I/O pins are capacitive loads and if any output line is high, or is being driven high, at the time of reset, it will take additional time for the weak pulldown enabled by the reset signal to bleed the charge off this capacitance. If the pin is not connected, we calculate that it will take an estimated 112 nS for one time constant to drain the pad well below threshold voltage. Temperature and process variation may affect this, hence the additional time alloted above. However, if the pin is connected, then the capacitance of the load must be taken into consideration and a longer reset may be warranted. This does not necessarily matter on all I/O pins, but certain pins such as bit 17 of serial boot nodes, and the clock lines of SERDES nodes, must be low immediately (within nS for the former and pS for the latter) or the boot nodes will begin acting on those high pins. Likewise if there are other circuits outside the SEAForth chip that must be reset, then as a system consideration it may be necessary to hold SEAForth in reset until these other circuits have safely reset.

*** Rise time should be as fast as feasible given the constraints of the board layout. The actual requirement depends on the noise level and frequency components of the noise present on the reset line. Set rise time fast enough that false threshold crossings do not last more than 100 pS.

Appendix A: Internal Data Representations and Levels

Logic levels used internally to the C18 cores for the representation of address and data have been selected by the designers to minimize the space, time and power that are used by each core. All of the various representations are internally consistent and generally handled transparently to the programmer through the operation of the compiler and through specification. Occasionally, these choices in representation are visible or may actually require some run-time adaptation.

Two cases in which code is affected are in the operation of the Core 0 external data bus, and in the representation of the VCO used for calculating analog input values.

A.1 Opcode Representations

Send feedback

The representation of opcode values varies according to slot number, and although this is handled automatically and transparently by the compiler, it is nevertheless visible in code dumps and deserves a brief explanation here.

The opcode bits within the instruction register have alternating representation when interpreted by the opcode decode logic. Because the opcode fields have odd length, the slots begin on alternating odd and even bit positions. Thus the alternating interpretation affects different bits in adjacent opcodes. When the opcodes are delivered to the instruction decode logic by the slot multiplexer, they all have the same representation, referred to as the "canonical" opcode value. This is different from what is seen in the slots of the instruction word.

Conversion between the binary value of an instruction word and this internal canonical value is done by complementing the instruction word with $15555, as shown in Table 9.1.

The hex values shown are for reference. Because opcodes are 5 bits long, the actual 18-bit word when represented in hex notation, will contain different hex digits. For example, the instructions in the sequence **and xor drop nop** have canonical values of $15, $16, $17, and $1C, but when assembled together in that order in a single instruction word will appear as $3E3EA. A second example is shown graphically in Table 9.1.

Table 9.1 Opcode Transformation Illustration

	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Opcode Slot	Slot 0					Slot 1					Slot 2					Slot 3		
4 **nop**s	1	1	1	0	0	1	1	1	0	0	1	1	1	0	0	1	1	1
xor $15555	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Result	1	0	1	1	0	0	1	0	0	1	1	0	1	1	0	0	1	0
Result (hex)	$16					$9					$16					$2		
Result (dump)	$2		$C				$9				$B				$2			

Table 9.2 Summary of SEAforth Opcodes – by Value

Name	Canonical Value (hex)	Canonical Value (binary)	Slot 0/2 Value (binary)	Slot 0/2 Value (hex)	Slot 1 Value (binary)	Slot 1 Value (hex)	Slot 3 Value (binary)
return	00	0 0000	0 1010	0A	1 0101	15	101
;:	01	0 0001	0 1011	0B	1 0100	14	not allowed
jump	02	0 0010	0 1000	08	1 0111	17	not allowed
call	03	0 0011	0 1001	09	1 0110	16	not allowed
unext	04	0 0100	0 1110	0E	1 0001	11	100
next	05	0 0101	0 1111	0F	1 0000	10	not allowed
if	06	0 0110	0 1100	0C	1 0011	13	not allowed
-if	07	0 0111	0 1101	0D	1 0010	12	not allowed
@p+	08	0 1000	0 0010	02	1 1101	1D	111
@a+	09	0 1001	0 0011	03	1 1100	1C	not allowed
@b	0A	0 1010	0 0000	00	1 1111	1F	not allowed
@a	0B	0 1011	0 0001	01	1 1110	1E	not allowed
!p+	0C	0 1100	0 0110	06	1 1001	19	110
!a+	0D	0 1101	0 0111	07	1 1000	18	not allowed
!b	0E	0 1110	0 0100	04	1 1011	1B	not allowed

Table 9.2 Summary of SEAforth Opcodes – by Value *(continued)*

Name	Canonical Value (hex)	Canonical Value (binary)	Slot 0/2 Value (binary)	Slot 0/2 Value (hex)	Slot 1 Value (binary)	Slot 1 Value (hex)	Slot 3 Value (binary)
`!a`	0F	0 1111	0 0101	05	1 1010	1A	not allowed
`+*`	10	1 0000	1 1010	1A	0 0101	05	001
`2*`	11	1 0001	1 1011	1B	0 0100	04	not allowed
`2/`	12	1 0010	1 1000	18	0 0111	07	not allowed
`not`	13	1 0011	1 1001	19	0 0110	06	not allowed
`+`	14	1 0100	1 1110	1E	0 0001	01	000
`and`	15	1 0101	1 1111	1F	0 0000	00	not allowed
`xor`	16	1 0110	1 1100	1C	0 0011	03	not allowed
`drop`	17	1 0111	1 1101	1D	0 0010	02	not allowed
`dup`	18	1 1000	1 0010	12	0 1101	0D	011
`pop`	19	1 1001	1 0011	13	0 1100	0C	not allowed
`over`	1A	1 1010	1 0000	10	0 1111	0F	not allowed
`a@`	1B	1 1011	1 0001	11	0 1110	0E	not allowed
`nop`	1C	1 1100	1 0110	16	0 1001	09	010
`push`	1D	1 1101	1 0111	17	0 1000	08	not allowed
`b!`	1E	1 1110	1 0100	14	0 1011	0B	not allowed
`a!`	1F	1 1111	1 0101	15	0 1010	0A	not allowed

Index