GreenArrays™

Application Note AN003 *Revised 8/10/11*

SRAM Control Cluster

Mark 1

This is an Application Module for the GA144. Four nodes and 40 pins are programmed to serve as a controller for an external SRAM chip. This module is used by GreenArrays applications and tools such as eForth and polyFORTH® (version 01k and later).

Your application may share this module with GreenArrays software, or you may customize it for dedicated use by applications requiring larger storage than is available within the chip, including your own Virtual Machines. The cluster may be customized for applications varying from a single master to three or more masters, with optional notification message passing between them.

In this paper we describe the operation of the module and its interfaces, and suggest possible customizations. The text assumes you have familiarized yourself with our hardware and software technology by reading our other documents on those topics. The current editions of all GreenArrays documents, including this one, may be found on our website at http://www.greenarraychips.com . It is always advisable to ensure that you are using the latest documents before starting work.

Contents

1. Overview 2

1.1 Targeted SRAM Chips 2

2. Hardware Configuration 2

3. Interface and Usage 3

4. Software Implementation 5

4.1 Node 107 - Interface 5
4.2 Node 007 - Data Bus and Control 6
4.3 Node 008 - Control Lines 7
4.4 Node 009 - Address Bus 8

5. Timing Diagrams 9

6. Customization 9

6.1 Additional Masters 9
6.2 Different SRAM chips 9
6.3 Simplifying the Interface 9

1. Overview

The SRAM Control Cluster is an example of *software-defined I/O* in GreenArrays chips. Nodes 7, 8 and 9 have enough pins between them to manage the address, data and handshake lines for an external SRAM chip; software converts four nodes of the GA144 into a versatile controller for such a chip using minimal external components, as shown in Figure 1.

This is also an excellent example of modularity. The source code is open and may be customized as you wish. If the standard minimal functions and interface are maintained, it may be used by our supported Virtual Machines and may be shared by your applications. A similar Cluster of five nodes has been implemented for SDRAM chips.

At the time of this writing, the timing of this cluster is serviceable but has not been fully optimized.

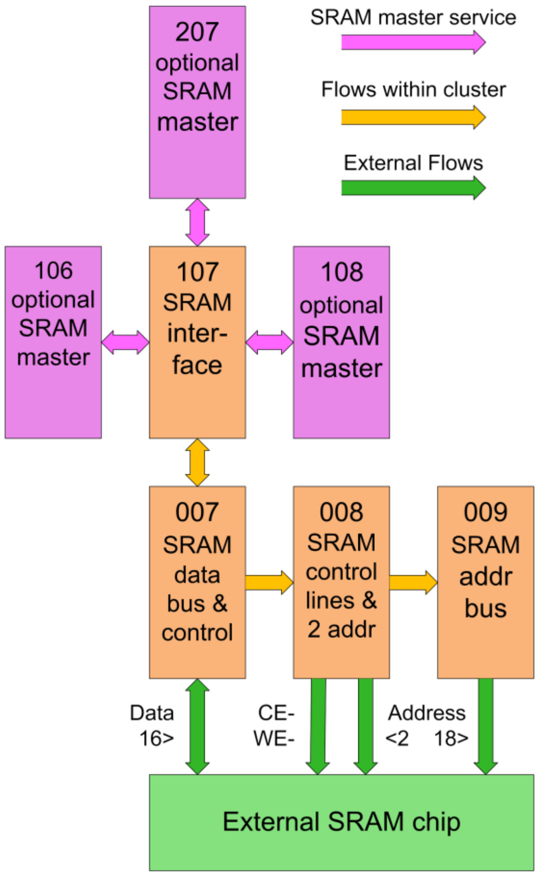

Figure 1 Cluster Topology

1.1 Targeted SRAM Chips

In this exercise, we have chosen the Cypress CY621 family of 16-bit, 1.8v SRAM chips. The part numbers prototyped are:

CY62137FV18LL-55BXVI (128k x 16)
CY62157EV18LL-55BVXI (512k x 16)
CY62167EV18LL-55BVXI (1M x 16)

The Host chip on the EVB001 Evaluation Board is configured with the latter, giving that chip one megaword (2 megabytes) of external SRAM.

2. Hardware Configuration

The sixteen low-order data lines d01..d15 from node 007 connect directly to data lines I/O00..I/O15 on the SRAM. To facilitate circuit layout, any one-to-one mapping of these pins is fine. Pins d16 and d17 of node 007 should be pulled down with resistors so that they will always read as zeroes.

The seventeen low-order address lines a01..a16 from node 009 connect directly to address lines A00..A16 on the SRAM. Like the low order data lines above, any one-to-one mapping of these pins is acceptable. However, 009.a01 *must* connect to A17 on the SRAM if you plan to use the 128 kword chip.

Pins 008.5 and 008.17 *must* connect directly to A18 and A19, respectively, on the SRAM. This requirement, like the rule for A17, allows memories smaller than 1 Mw to be addressed contiguously starting at zero. If you never plan to support a memory smaller than 1 Mw, it is then permissible to map all 20 of these address lines in any order.

Pin 008.3 connects directly with WE-, and 008.1 connects directly with CE-. Because these pins reset to weak pull-down, which would assert them and start an SRAM write operation, they should both be pulled up using a moderate resistor value such as 4.7K or 5.1KΩ.

Remaining SRAM pin CE2 should be tied to VDD and the OE-, BHE- and BLE- pins should all be grounded. For schematic and layout examples, see the Data Book on the EVB001 Evaluation Board (DB003).

3. Interface and Usage

Node 107 provides an internal interface between the SRAM cluster and up to three nodes (106, 207, and 108) that may act as independent memory masters. It provides three *atomic* 16-bit operations on memory: Read, Write, and Compare-and-exchange. The code below is available for use in the memory master nodes and illustrates the mechanism for passing requests to node 107 when **B** has been set to the appropriate port address.

```
example code for memory master nodes.
memory-access words assume that addresses and
data are 16-bit parameters with the upper two
bits zero and pages are 4-bits with the upper
14 bits zero. p.a is thus a 20-bit address.

ex@ a p - w fetch w from p.a
ex! w a p store w at p.a
mk! w f -0 set masks from w per f.
cx? w a p n - f comp-and-exch

cx? compares value at p.a to n. if same, writes
w to p.a and returns true. otherwise, only r
eturns false. x@ and x! are 16-bit versions to
access the lowest 64k of available memory.

mk! sets mask from w when f is 0;
posts stimuli when f is 1.
```

```
250 list
- user node 106, 108, or 207. 39 org
x! wa 039 dup dup or
ex! wap mk! mfp' 03A - !b - !b !b ;
x@ a-w 03C dup dup or
ex@ ap-w 03D !b !b @b ;
cx? wapn-f 03E - !b !b !b !b @b ; 040
```

A memory master node makes a request by writing a sequence of two, three or four words to the port it shares with node 107. Results, where applicable, are returned through the same port. The four primitive operations are as follow:

`ex! (w a p)` Writes a 16-bit value `w` into memory at the 20-bit address `p:a` where `p` is 4 bits long and `a` is 16 bits. The protocol is to write 3 words through the port `[-p -a w ]`. The first two words are inverted (-) to identify the operation desired. This operation proceeds in parallel with master code execution since the protocol is completed at the beginning of a relatively long memory cycle.

`ex@ (a p - w)` Reads a 16-bit value `w` from memory at the 20-bit address `p:a` as above. The protocol is to write 2 words through the port `[+p +a]` and then to read the result `[w]` .

`cx? (w a p n - f)` Examines the word of memory at `p:a` to see if it matches the value `n` . If the values match, writes `w` into that word of memory and returns true (xFFFF). If they do not match, returns false (0) without altering memory. The protocol is to write 4 words through the port `[-n +p a w]` and then to read the result `[f]` .

`mk! (w f -0)` Sets masks. Bits x8A00 in `w` identify masters by their port write signals. When `f` is zero, enables or disables masters according to `w` . When `f` is one, posts stimuli additively for masters according to `w` . The protocol is to write 3 words through the port `[+x -f w]`. A maximum of one pending stimulus is saved for each master until it is accepted by reading its comm port with the interface node, or until it is cleared by disabling that master.

Throughout this paper, the notation `-x` means the 18-bit inverse of a 16-bit value `x` .

The memory master node is responsible for guaranteeing valid arguments. This means that `p` must be in [0..15], and both `a` and `w` must be in [0..65535]. Likewise the numbers may only be inverted as shown above, in their 18-bit form, to identify operations. No execution-time error checking is done, as this would cost time, space, and energy. The results of using invalid arguments are unspecified.

The `cx?` primitive is useful in coordination of parallel processors using flags and data structures such as queues in shared memory. *Note: We chose to implement this 80486-style instruction, rather than the classical test-and-set, because compare-and-exchange has proven to be more powerful and useful.*

`mk!` empowers multiple masters to cooperate efficiently while sharing memory. Firstly, any of the masters may specify the set of masters from which the interface node will subsequently recognize requests. The permissible values for the argument `w` when `f=0` are as follow:

Hex value	Commands subsequently accepted from
0000	No commands accepted until chip is reset.
0200	Node 207 (up port) only
0800	Node 108 (left port)
0A00	Nodes 108 and 207
8000	Node 106 (right port) only
8200	Nodes 106 and 207
8800	Nodes 106 and 108
8A00	Nodes 106, 207 and 108

An initial value for this mask is selected when booting and starting the node.

Secondly, when a master discovers that it has nothing to do unless and until another master stimulates it, the interface node supports a method whereby the idle master may suspend itself and another may stimulate it. To accomplish this, the idle master simply reads the port connecting it with the interface node, and the stimulating node sets mask bits in the interface node when it wishes to stimulate another. Once set, a given stimulus bit remains set until the stimulus is passed to the corresponding node by writing a word of data to it. A node's stimulus bit is also cleared when commands from that node are disabled. The permissible values for the argument `w` when `f=1` are as follow:

Hex value	Stimuli are posted for node(s)
0000	No stimuli posted.
0200	Node 207
0800	Node 108
0A00	Nodes 108 and 207
8000	Node 106
8200	Nodes 106 and 207
8800	Nodes 106 and 108
8A00	Nodes 106, 207 and 108

When used properly, this function allows any master to suspend execution when appropriate without worries about race conditions. Proper usage follows from two simple, basic rules:

1. When a master has examined all shared memory structures and sees nothing to do until some other master changes those structures, it may then read its port to the interface node and discard whatever it receives. Execution of that master will be suspended until another master posts a stimulus to it. Upon resuming execution, the suspended master must examine all memory structures again and find nothing to do before suspending again. To avoid races, after a master has found something to do it must start over and re-examine all shared memory structures, finding nothing to do in any of them, before suspending.

2. When a master alters any memory structure in such a way as to give another master something to do, the first master must post a stimulus to the second master after altering that structure.

If a second master updates a memory structure after a first has examined it and found nothing to do, the first may still pause because the second will either have already posted a stimulus or will soon do so. There is no race under these conditions because the stimulus is stored in the interface node until the recipient node is ready to receive it.

4. Software Implementation

Please refer to Figure 1 for a visualization of the roles of the four nodes as they are discussed.

4.1 Node 107 - Interface

In order to service multiple masters with arbitrary timing in the G144A12, node 107 must continuously poll its **IO** register. The code shown here is the full version with all capabilities including 3 masters and stimulus passing. See section 6 below for ways to save time and energy when you don't need all of these capabilities. **B** has a fixed assignment of down pointing to node 007. **A** is used to access **IO** and the ports connecting to masters. The stack maintains a mask of relevant port read/write signals in **IO** underneath the constant x15555. Masters send commands to node 107 using the following protocols through a port:

Words Received				Reply Sent	Function Performed
1st	2nd	3rd	4th		
+p4	+a16	---	---	w16	e@ Read a word from SRAM at p:a
+x	-0	m16	---	---	mk! Set master enable mask
+x	-1	m16	---	---	mk! Post stimuli for master(s)
-p4	-a16	w16	---	---	e! Write a word into SRAM at p:a
-n16	+p4	a16	w16	f16	Compare-and-exchange. Write w to SRAM iff current value = n, return x0FFFF if stored or 0 if not.

```
node 107 full capability version.
polls for master requests and delivers stimuli
all requests are atomic. passes ex@ and ex!
requests on to node 007, performs cx? locally
using those primitives. the command and
stimulus mask m is maintained on the stack.

requests are variable length messages decoded
as shown below where - means 18-bit inverse of
16 bit argument.

ex@ +p +a fetch
cx? -w1 +p a w2 comp-and-exch
ex! -p -a w store
mk! +0 -f m f-0 enables each master whose port
write bit is set in m. kills pending stimulus
for any disabled master. abandons old mask.
f-1 adds a stimulus for each master whose port
write bit is set in m. caller should not post
a stimulus for any disabled master.
```

```
248 list
- interface 107 node 0 org
cx wp- 000 over push @ dup
a !b over p !b @b w pop - w1 or if
ne @ w2 dup or ff ! ;
eq then drop a !b - -p !b @ w2 !b FFFF ! ;

cmd 00A @ -if @ ' cx -until .e! - !b !b @ !b ;
then 00E @ -if mixpa
.mk! 00F - push drop drop pop if mia
..stim 011 drop drop @ 2* over - ahead swap
..mask 014 then drop and @ over over 2*
...both 016 then and or
re 017 m 15555 dup ahead swap
then .e@ 019 a !b p !b @b w ! ;
cmds mixa 01B a! cmd poll mix then io a! 01E
begin drop over over @ or and until
021 over over and if mixt 023 and and mt
. dup 10000 and if right ahead swap then
. 028 drop 1000 over and if left ahead swap
. 02D then drop 400 dup up then then
.. 02F mtba a! and or dup ! m ' re end
031 then drop 2* 2* -if right cmds ;
035 then 2* 2* 2* 2* -if left cmds ;
039 then up cmds ; 03B
```

poll is the idle loop of this node, polling the **IO** register for activity to which it must respond. On entry the stack contains a mask underneath the value x15555 with a throw-away value x on top. In the polling loop, **IO** is read, exclusive ORed with x15555 to convert its port read status bits to positive true, and tested for nonzero under the mask. The mask has a 1 for the port write bit(s) (Rw, Lw, and Uw) corresponding to those nodes from which commands are being accepted. It also has a 1 for the port read bits (Rr-, Lr- and Ur-) corresponding to each node for which a stimulus is pending. The polling loop ends when at least one node is offering a command or is trying to read a pending stimulus. On finding a master ready to be stimulated, we write a word through its comm port and clear its port read mask bit. Otherwise one command source is identified and control passes to cmd after setting **A** to point to the node offering the command.

cmd reads the incoming command message one word at a time, checking the signs of the first two as an economical way of decoding which of the four primitive functions is being requested. Memory read and write commands are passed on to node 007 for execution, with `a` always positive and the write function indicated by the inversion of the `p` argument. Compare-and-exchange is performed locally, generating one or two memory operations depending on its outcome; the operation is atomic because the memory masters are ignored throughout its execution.

The command to set the master enable mask replaces all three port write bits with the value given. In addition, for each zero port write bit the corresponding read bit is cleared; this resets any pending stimulus for a master that is being disabled.

The command to post stimuli logically ORs the given mask, shifted, with the current mask; this has the effect of recording a pending stimulus for each master whose port write bit is set in `m`. *It is your duty to set stimuli only for nodes that you have enabled.*

4.2 Node 007 - Data Bus and Control

Node 007 processes memory read and write commands at the request of node 107. This node handles the transfer of data to and from the SRAM chip on 16 of its data bus pins, and coordinates the activities of nodes 008 and 009 in driving address and control signals to the SRAM. **B** has a fixed assignment of `left` pointing to node 008. **A** is used to access `down` for communication with node 107, **IO** to control data bus direction, and `data` to read and write the data bus. The protocols for requesting operations of node 007 are as follow:

Words Received			Reply Sent	Function Performed
1st	2nd	3rd		
a16	+p	---	w16	e@ Read a word from SRAM
a16	-p	w16	---	e! Write a word into SRAM

```
node7 suspends waiting for a16, passes it and
page/r/w to nodes 8 and 9, finally controlling
the data transfer and timing until sending the
stop command.

the literals needed for writing are loaded
onto the stack and used circularly to save
time. /read's drops are free./

---- .lit. pin17 pin05 pin03 pin01
stop 3557F a19-1 a18-1 /we-1 /ce-1
```

```
246 list
- data-bus 7 node host
in 14555 lit ; out 15555 lit ;
stop 3557F lit ; target

0AA 20 org
start 020 left b! out io data stop
out io data stop in io a! in !
down a! stop !b
cmd /soid/ 031 @ a16 !b @ +p/-p -if

w16 /soid/p- 033 +p/-p !b
/- setup + 45ns @ w a push push data a!
pop ! io a! out ! 40 13 for unext stop !b
-/ in ! pop a! cmd ;

r16 /soid/p- 03C then +p/-p !b
/- setup + 55ns a push data a!
io drop out drop 50 40 for unext stop !b -/
@ w pop a! ! cmd ; 043
```

in and **out** are values to write into **IO** to set the data bus direction from the point of view of node 007.

stop is the value transmitted to node 008 for completing a read or write operation. Upon receiving it, node 008 stores that value into its IO and this drives all four of its pins high, deasserting both WE- and CS-.

start initializes **B**, sets the data bus to input mode, sets **A** to its default value pointing to node 107, and transmits a `stop` signal to node 008 before falling through into `cmd`. Upon exit, the stack has been initialized in a way that illustrates a useful technique in programming the F18: It holds a sequence of values that would otherwise have to be literals, and because the stack is circular these values will be available in sequence perpetually. The values on the stack, with **S** and **T** on the right, are as follow: `[io data stop out io data stop out | io data]`. So long as you never push more than two words that don't continue this pattern onto the stack, you will be able to pop them and the ten word pattern (with its eight word repeating part) remains intact. Advancement along the pattern is done by simply executing `drop`.

`cmd` waits for a command from the interface node and executes it. The first word received is the low order 16 bits of the address; these are passed directly to node 008 for delivery to node 009. The second word, the page address with the high order four address lines, is negative for `ex!` and positive for `ex@`.

`w16` completes `cmd` for storing into SRAM. The negative page address is passed to node 008 which starts the critical path toward completing the store operation. Register **A** is saved on the return stack, and the value to store, `w`, is pushed on top of it. We then use a four word pattern from the stack: **A** is set to `data` and the value `w` is written to that register where it will be used to drive the data bus; **A** is set to **IO** and the value `out` is written there, enabling the value `w` to be driven onto the data bus; we delay for a little less than 40 ns, finally sending the `stop` signal to node 008, setting the data bus back to input mode, and restoring the saved value of **A**.

`r16` completes `cmd` for reading from SRAM. The positive page address is passed to node 008 to start the timing sequence. Register **A** is saved on the return stack and **A** is set to `data` from the pattern. We skip the next two pattern words because the data bus is already in input mode, and then delay for a little more than 100 ns before signaling node 008 to `stop`, restoring **A**, reading the data bus and finally sending its value to node 107.

4.3 Node 008 - Control Lines

As we proceed down this pipeline, the job of each node becomes simpler. Node 008 handles requests from node 007 using two protocols:

Words Received			Function Performed
1st	2nd	3rd	
a16	+p	stop	e@ Read a word from SRAM
a16	-p	stop	e! Write a word into SRAM

```
node8 is fed a stop command during start-up,
then suspends while waiting for a16. after
starting the read or write, it again suspends
while waiting for the stop command.

bits 4..2 of the /possibly inverted/ page
value are used 'as-is' to index into the start
table, setting two address bits, write enable,
and chip enable. ** note that reads and writes
are swapped if the page 'overflows' into bit4,
with disastrous results **

cmd index .lit. pin17 pin05 pin03 pin01
w00 .0111 2556A a19-0 a18-0 /we-0 /ce-0
r00 .0000 2556E a19-0 a18-0 /we-1 /ce-0
w01 .0110 2557A a19-0 a18-1 /we-0 /ce-0
r01 .0001 2557E a19-0 a18-1 /we-1 /ce-0
w10 .0101 3556A a19-1 a18-0 /we-0 /ce-0
r10 .0010 3556E a19-1 a18-0 /we-1 /ce-0
w11 .0100 3557A a19-1 a18-1 /we-0 /ce-0
r11 .0011 3557E a19-1 a18-1 /we-1 /ce-0
```

```
244 list
- control-pins 8 node host
'r-l- 1F5 lit ; target 0 org

'start' pin control table 0-7
000 2556E r00 , 2557E r01 ,
002 3556E r10 , 3557E r11 ,
004 3557A w11 , 3556A w10 ,
006 2557A w01 , 2556A w00 ,
008 20 org

start 020 'r-l- b! io a!
cmd 024 @b stop ! a push 7 mask ..
@b a16 !b @b +p/-p dup !b
2/ 2/ and i3 a! .. @ ctrl pop a!
start ! cmd ; 02C
```

cmd waits for a `stop` signal from node 007 to end the preceding operation by de-asserting control lines; this is also done as part of initialization. Register assignments at the start of this loop are **IO** in **A** and a dual port address (that of `right` and `left`) in **B** . This illustrates another useful technique in programming the F18; because the protocols make ambiguity impossible, node 008 may communicate with nodes 007 and 009 at appropriate points in the protocols using a single value in the address register!

At the start of a new command, we receive the low order 16 bits of the memory address from node 007 and immediately pass that value to node 009 using the dual port address in **B** . We then read that same port address again, obtaining the high order four bits of the address, inverted for write. This is also passed immediately to node 009 for its use in driving the low order 18 address lines.

Finally, `cmd` isolates a three bit field of `p` consisting of `[write a19 a18]` and uses that value to index an 8 word table starting at 0 in node 008 RAM. The value from this table is written into **IO** thus appropriately driving pins 008.17 (A19), 008.5(A18), 008.3(WE-), and 008.1(CE-, asserted in all cases) to initiate an SRAM read or write operation. Finally we loop back to `cmd` where we wait for the stop signal from node 007, which is responsible for overall cycle timing.

4.4 Node 009 - Address Bus

The last node in this pipeline is responsible for assembling the low order 18 bits of the SRAM address and driving them onto the address lines. The two word commands it receives from node 008 are as follow:

Words Rcvd		Function Performed
1st	2nd	
a16	+p	Drive address bus with combination of a16 and least significant 3 bits of |p|.
a16	-p	

```
node 9 suspends while waiting for a16. it uses
the two lower page bits to output an
18-bit address.

a16 xx.aaaa.aaaa.aaaa.aaaa
p04 00.0000.0000.0000.pppp
a18 aa.aaaa.aaaa.aaaa.aapp

the code is written to minimize/equalize the
time to output the address, which must be
stable when node8 stores the 'start' command.
```

```
242 list
sram.16 address-bus 9 node
0AA 20 org

start 020 right b! .. data a! .. 3 mask
cmd m 026 @b a16 2* 2* over @b -if
028 - p04 and or a18 ! cmd ;
02A then p04 and or .. a18 ! cmd ; 02C
```

cmd waits for two words from node 008 to control the address bus. Register assignments throughout this loop are `data` in **A** and the port address of node 008 in **B** . The data bus is presumed to be in its reset state (output). When the two words arrive, the page value is complemented if necessary and its low order two bits are then combined with the 16-bit low order address value to drive the 18-bit address bus.

The coding in this version of the cluster Is appropriate if the SRAM is wired to the GA144 as described in section 2 above; placing memory masters' A16 and A17 bits on 009.A00 and A01 respectively is simply the fastest way. Other mappings may be done in this node at the expense of time and energy.

5. Timing Diagrams

These will be provided after the code has been fully optimized. The software shown in this revision is provides its masters with practical memory read cycles on the order of 250 ns using the full featured version of the interface node software, or 200 ns using the degenerate version discussed below.

6. Customization

This module was created to meet the needs for random access to external memory by a variety of memory masters that may be sharing that resource. Examples are one (or more) virtual machines with one or more memory mastering I/O modules as may occur in general purpose applications. Your application may not have the same requirements; for example it may be a continuous process needing rapid sequential or other patterns of access. You are encouraged to tailor or rewrite this code to achieve optimal performance in your particular application, both in time and in energy consumption. If random access characterizes your application there are still ways in which you might benefit by customizing this code; here are a few examples.

6.1 Additional Masters

Any of the master nodes may multiplex requests for others using tree-like flow structures.

6.2 Different SRAM chips

You may wish to support devices having different timing, which can be easily accomplished by adjusting the delays in node 007. Wider addresses can be achieved by arranging internal communications to use additional GPIO pins, or with external latches, as is most appropriate for the application. Additional GA144(s) can serve well as I/O expanders if necessary.

6.3 Simplifying the Interface

Code may be removed from node 107, reducing time and energy consumption, if its full capabilities are not needed. Here is an example of the degenerate case in which only one master is supported:

```
node 107 minimal capability version.
single master, no polling, no stimuli.
maximum speed, minimum power.

all requests are atomic. passes ex@ and ex!
requests on to node 007, performs cx? locally
using those primitives.

requests are variable length messages decoded
as shown below where - means 18-bit inverse of
16 bit argument.

ex@ +p +a fetch
cx? -w1 +p a w2 comp-and-exch
ex! -p -a w store
```

```
252 list
- degenerate sram 107 node 0 org

cx wp- 000 over push @ dup
a !b over p !b @b w pop - w1 or if
ne @ w2 dup or ff ! ;
eq then drop a !b - -p !b @ w2 !b FFFF ! ;

cmd 00A @ -if @ ' cx -until .e! - !b !b @ !b ;
then 00E @ .e@ 00F a !b p !b @b w ! ;

011 17 org
start 017 down b! right a!
run 01B cmd run ; 01D
```

GreenArrays™ Application Note AN003 *Revised 8/10/11*

Mailing Address: GreenArrays, Inc., 774 Mays Blvd #10 PMB 320, Incline Village, Nevada 89451
Printed in the United States of America
Phone (775) 298-4748 fax (775) 548-8547 email Sales@GreenArrayChips.com
Copyright © 2010-2011, GreenArrays, Incorporated