Memory Initialization on the EBSA-285

Application Note

October 1998

Order Number: 278152-001

Application Note

Contents

Tables

1.0 Introduction

This document provides a brief overview of the techniques used to initialize the memory subsystem on the EBSA-285 evaluation board.

The EBSA-285 evaluation board uses 168-pin Dual In-line Memory Modules (DIMMs) for its volatile memory. These DIMMs come in different configurations and sizes, and the 21285 PCI support device can control a range of DIMMs.

This application note describes the differences between DIMMs and how to uniquely distinguish them in software.

Use this note in conjunction with the *21285 Core Logic for SA-110 Microprocessor Data Sheet* (order number 278115) and the *EBSA-285 Evaluation Board Reference Manual* (order number 278136).

1.1 Memory on EBSA-285

A DIMM (Dual In-line Memory Module) is a small circuit board that contains a number of memory devices with the data signals routed to a standard connector.

The socket on the EBSA-285 accepts only 3.3 V SDRAM DIMMs. The EBSA-285 uses a combination of chip-select and byte-select (dqm) lines to provide a single 32-bit data path to the 21285 and the processor.

Each DIMM consists of one or more arrays; an array contains one or more banks of memory. An array is a block of memory with shared addressing, unique control, and its own data path. Where an array contains multiple devices sharing the data path, each independently controlled block of memory is called a bank. Each array may be a different size, but usually all arrays on a single DIMM are the same size. The primary variation between DIMM types is the multiplexing of address lines to provide row, column, and bank address to the DIMM.

2.0 EBSA-285 Memory System

On the EBSA-285, the SA-110 memory system is managed entirely by the 21285. The memory controller on the 21285 can control from one to four arrays of SDRAMs. The EBSA-285 uses 168-pin DIMMs, which provide a 64-bit path to memory, and handles the interface to enable 32-bit access for the StrongARM** and PCI.

2.1 Supported Memory Types

The 21285 groups DIMMs into five different address multiplexing modes (see Table 1).

Table 1. Array Sizes

SDRAM Type			Address Bits			SDRAMs in Array	Array Size	Row/Column Multiplexer Mode
Banks	Depth	Width	Bank	Row	Col.			
8Mb Parts								
2	128K	32	1	9	8	1	1MB	000
16Mb Parts								
2	256K	32	1	10	8	1	2MB	000
2	512K	16	1	11	8	2	4MB	001
2	1M	8	1	11	9	4	8MB	001
2	2M	4	1	11	10	8	16MB	001
64Mb Parts								
2	1M	32	1	12	8	1	8MB	010
4	512K	32	2	11	8	1	8MB	011
2	2M	16	1	13	8	2	16MB	010
4	1M	16	2	12	8	2	16MB	100
2	4M	8	1	13	9	4	32MB	010
4	2M	8	2	12	9	4	32MB	100
2	8M	4	1	13	10	8	64MB	010
4	4M	4	2	12	10	8	64MB	100

2.2 DIMM Address Mapping

The DIMM types supported by the 21285 have been selected to give the greatest breadth of coverage against the amount of address line switching. Note that for all DIMM types, the pinouts for the row address are invariant and contiguous address lines are used.

The SDRAM addresses are driven on the multiplexed address bits ma[12:0] and bank address bits ba[1:0]. Address line a[19] is always a bank address, a[17:9] are always row addresses, and a[8:2] are column addresses. Address lines a[25:20] and a[18] are bank, row, or column address lines, as specified in Table 2. The usage is configured by programming the address multiplex bits in the SDRAM Address and Size Register for the specific array.

Note: The dashes (-) in Table 2 indicate that the address is not used by the SDRAM in this configuration. The pin is driven by the 21285 and its value can be either 0 or 1.

"ap" in Table 2 indicates the autoprecharge bit that is used by the SDRAM during column address time. A low (deasserted) indicates no autoprecharge, which occurs during read and write commands when there is another burst pending to the same page. A high (asserted) indicates autoprecharge, which occurs during read and write commands of the last burst.

Table 2. SDRAM Addresses

Mode		BA		SDRAM Address ma[12:0]												
		1	0	12	11	10	9	8	7	6	5	4	3	2	1	0
000*	Row	-	19	-	-	-	-	17	16	15	14	13	12	11	10	9
	Col.	-	-	-	-	-	-	ap	18	8	7	6	5	4	3	2
000*	Row	-	19	-	-	-	18	17	16	15	14	13	12	11	10	9
	Col.	-	-	-	-	-	ap	-	20	8	7	6	5	4	3	2
001	Row	-	19	-	-	21	18	17	16	15	14	13	12	11	10	9
	Col.	-	-	-	-	ap	23	22	20	8	7	6	5	4	3	2
010	Row	-	19	23	22	21	18	17	16	15	14	13	12	11	10	9
	Col.	-	-	-	-	ap	25	24	20	8	7	6	5	4	3	2
011	Row	20	19	-	-	21	18	17	16	15	14	13	12	11	10	9
	Col.	-	-	-	-	ap	-	-	22	8	7	6	5	4	3	2
100	Row	20	19	-	22	21	18	17	16	15	14	13	12	11	10	9
	Col.	-	-	-	-	ap	25	24	23	8	7	6	5	4	3	2

* Bit 0 of the Array size, in the SDRAM Address and Size Register, determines the address line routing for the different Mode 0 type DIMMs.

2.3 21285 Memory Controller Registers

The memory controller on the 21285 is accessed via the SDRAM Timing Register, which controls:

- row precharge timing
- RAS-to-CAS delay
- CAS latency
- Row Cycle time
- Refresh interval
- Parity enable

There is one register, the SDRAM Address and Size Register, for each of the four arrays, 0 through 3. These four registers define the start address, size, and address multiplexing for each of the four SDRAM arrays. Software must ensure that the arrays of SDRAM are mapped so that there is no overlap of addresses.

- The arrays do not need to be the same size. However, the start address of each array must be naturally aligned to the size of the array. For example, an 8 MB array must start on an address that is divisible by 8 MB.
- The arrays do not need to form a contiguous address space. However, to form a contiguous address space with different size arrays, place the largest array at the lowest address, next largest array above, and so on.

3.0 Initialization on Power On

When the EBSA-285 is first switched on, the 21285 holds all byte enable lines (dqm) high (deasserted) for all arrays to keep the DIMMs in reset. The first thing that must be done is to read and write to the 21285 DRAM mode registers. The read enables the refresh mechanism so that the memory contents are maintained even when not being accessed. All of the DRAM mode registers must be written, even if some of the arrays are not being used, as it is this unique operation that causes the 21285 to remove the hold on the dqm lines, allowing them to be used to access memory.

Next, the SDRAM Timing Register is written, followed by the SDRAM Address and Size registers. Finally, after waiting for at least 8 refresh cycles, the memory can be accessed.

3.1 Implementation

All of this initialization is done in an ARM** Assembly Language macro - obviously, there is no memory available yet, so a high-level language such as C cannot be used. These macros allow a range of registers to be passed as variables, and the macros can use these variable names to allow greater understanding of the function of the code.

This macro is called INIT_MEM and is defined in the file target.s, which can be found in the Angel* and uHAL sources included in the EBSA-285 Evaluation Kit. The registers used to pass variables are fixed (with the ASSERT pseudo instruction) because nearly all of the other registers are used within the macro. Three other macros (SETUP_RAM, SIZE_RAM, and RAM_REG) are called from within INIT_RAM to clarify the instruction flow.

Example 1. Macro Definition

This macro would be called as follows:

```
INIT_RAM          r0, r1, r5
```

and is defined as:

```
    ; ---------------------------------------------------------------------

    ; INIT_RAM

    ; --------

    ; ANGEL and uHAL macro to initialize memory on start-up. Compatibility

    ; with Angel specifies a1, a2 and v2 as calling registers and that a1

    ; must not be used.

    ; WARNING: This macro uses lots of registers!

    ; $w1 (a1), $w2 (a2), $w3 (v2), v3, v4, v5, v6, a3 and a4

MACRO

INIT_RAM       $w1, $w2, $w3

    ; With such pressure on registers, have to make sure the calling

    ; registers are just so!

    ASSERT ($w1 = r0)

    ASSERT ($w2 = r1)

    ASSERT ($w3 = r5)
```

If the memory has already been initialized, the refresh must be disabled before writing to the array mode registers (as described in the *21285 Core Logic for SA-110 Microprocessor Data Sheet*). The full definitions of the constants and offsets can be found in the EBSA-285 file platform.s:

Example 2. Constant Definitions

```
; /* DC21285 Addresses */
DC21285_DRAM_A0MR         EQU        0x40000000
DC21285_DRAM_A1MR         EQU        0x40004000
DC21285_DRAM_A2MR         EQU        0x40008000
DC21285_DRAM_A3MR         EQU        0x4000C000
CSR_BASE                  EQU        0x42000000
; /* Offsets from CSR_BASE */
DRAM_TIMING               EQU        0x10C
DRAM_ADDR_SIZE_0          EQU        0x110
DRAM_ADDR_SIZE_1          EQU        0x114
DRAM_ADDR_SIZE_2          EQU        0x118
DRAM_ADDR_SIZE_3          EQU        0x11C
MAX_SDRAM                 EQU        4
; /* SD_OFFSET is the difference between SDRAM Array Register banks */
SD_OFFSET                 EQU        (DC21285_DRAM_A1MR - DC21285_DRAM_A0MR)
; /* Row/Col Mux modes shifted into correct position (bits 6, 5 and 4) */
MUX_MODE0                 EQU        0x00
MUX_MODE1                 EQU        0x10
MUX_MODE2                 EQU        0x20
MUX_MODE3                 EQU        0x30
MUX_MODE4                 EQU        0x40
MUX_MASK                  EQU        0x78
; /* Bit patterns for SDRAM memory sizes */
; [Other definitions removed for clarity]
SDSIZE_64M                EQU        0x07
; /* handy sizes - don’t have to remember bit patterns */
SZ_64M                    EQU        0x04000000
; /* See 21285 data sheet: */
; [Other definitions removed for clarity]
DC21285_Trp2              EQU        0x1
DC21285_Tdal3             EQU        (0x1 << 2)
DC21285_Trcd2             EQU        (0x2 << 4)
DC21285_Tcas2             EQU        (0x2 << 6)
```

```
DC21285_Trc4              EQU        (0x1 << 8)
DC21285_Tcd0              EQU        (0x0 << 11)
DC21285_Parity0           EQU        (0x0 << 12)
; Tref, No Parity, CDT=0, Trc=4, Tcas=2, Trcd=2, Tdal(Trp+Trdl)=3, Trp=2
;       0          0      010      10      01               01
saTrp                     EQU        DC21285_Trp2
saTdal                    EQU        DC21285_Tdal3
saTrcd                    EQU        DC21285_Trcd2
saTcas                    EQU        DC21285_Tcas2
saTrc                     EQU        DC21285_Trc4
saTcd                     EQU        DC21285_Tcd0
saParity                  EQU        DC21285_Parity0
SDRAM_MODEBITS            EQU        (saParity + saTcd + saTrc + saTcas + saTrcd +
                                      saTdal + saTrp)
; /* Tref=1 */
INITIAL_TREF              EQU        (0x01 << 16)  ; Top half of word
INITIAL_TIMING            EQU        (INITIAL_TREF + SDRAM_MODEBITS)
; /* Tref=1a (64 ms x 4096 rows) = 15.xx us */
SETUP_TREF                EQU        (0x1a << 16)  ; Top half of word
SETUP_TIMING              EQU        (SETUP_TREF + SDRAM_MODEBITS)
REFRESH_COUNT             EQU        32            ; No. of cycles scaled per wait
REFRESH_WAIT              EQU        8             ; Max. wait per SDRAM
CAS_OFFSET                EQU        (saTcas + saTdal)
```

Example 3. Setup Code

```
; If memory is already active, need to disable refresh before
        ; re-initializing.
        LDR         $w3, =CSR_BASE
        MOV         $w2, #0
        STR         $w2, [$w3, #DRAM_TIMING]
        ; If refresh is just happening, wait 50 clock cycles until it expires
        MOV         $w1, #50
11
        SUBS        $w1, $w1, #2
        BGT         %B11                        ; Until done
        ; Read from each Mode Register array to enable precharge
        MOV         $w1, #MAX_SDRAM
        MOV         $w2, #DC21285_DRAM_A0MR
1
        LDR         $w3, [$w2]                  ; Just read, don't care about data
        ADD         $w2, $w2, #SD_OFFSET        ; Next
        SUBS        $w1, $w1, #1
        BGT         %B01                        ; Until done
        ; Now write to the Mode registers to set the operational mode of the
        ; SDRAM. MUST be done to all banks, even if not fitted, to stop the
        ; 21285 forcing DQM High, which it does after reset.
        MOV         $w1, #MAX_SDRAM
        MOV         $w2, #DC21285_DRAM_A0MR
2
        STR         $w2, [$w2, #CAS_OFFSET]     ; It's the ADDRESS which is important
        ADD         $w2, $w2, #SD_OFFSET        ; Next
        SUBS        $w1, $w1, #1
        BGT         %B02                        ; Until done

        ; Setup Dram timing & DISABLE all arrays by default.
        ;
        ; Setting the refresh interval to be minimum & enabling refresh.
        ; After (up to) 8 refresh cycles the memory will be ready to use, each
```

```
; refresh cycle is 32 instruction cycles.
        LDR         $w3, =CSR_BASE
        LDR         $w2, =INITIAL_TIMING
        MOV         $w1, #DRAM_TIMING
        STR         $w2, [$w3, $w1]
        MOV         $w2, #MAX_SDRAM
        MOV         $w2, $w2, LSL #2
        ADD         $w1, $w1, $w2             ; Registers are after timing register
        MOV         $w2, #0                   ; Disable each array & move to address 0
3
        STR         $w2, [$w3, $w1]
        SUBS        $w1, $w1, #4
        BGT         %B03                      ; Until done
        ; Wait around until memory is active
        MOV         $w1, #REFRESH_WAIT        ; Number of refresh cycles to wait
4
        MOV         $w2, #REFRESH_COUNT       ; Length of each cycle
5
        SUBS        $w2, $w2, #2              ; 2 instructions in this loop
        BGT         %B05                      ; Until cycle complete
        SUBS        $w1, $w1, #1              ; Another refresh cycle gone..
        BGT         %B04                      ; Until done
```

The final step in a system with a fixed memory subsystem would be to write to the 21285 DRAM Address and Size registers for each array. These registers hold the start address, size, and enable/disable status for each array (see Section 3.1.6 and Section 3.1.7).

3.1.1 How to Differentiate DIMM Types

In systems that have a fixed memory subsystem, the previous procedure is sufficient. However, in a dynamic system such as the EBSA-285, where additional DIMMs may be added, some mechanism must be implemented to determine the current configuration.

DIMM mode types may be determined using simple memory aliasing tests. The processor address lines used for row, column, and bank addressing to the DIMM are dependent upon the DIMM type, as outlined in Table 2. If a DIMM is configured incorrectly, at least one of the high-order address lines (a[25:18] excluding a[19]) will be mapped incorrectly with an address line of a higher order than the memory size. The "missing" address bit will then address the same location irrespective of its value. This is known as memory aliasing. For example, if the 2nd entry in Table 2 is configured when the first entry is fitted, a[20] will supply the most significant bit of the column address instead of a[18]. Addresses 0x0 and 0x80000 will access the same location because in both cases a[20] equals zero.

By using the table of SDRAM addresses (Table 2), the relationship between invalid modes and aliased locations can be determined.

There are two keys to ensuring proper identification of all supported DIMMs:

- There is only one Mode type 3 DIMM, and its array size is always 8 MB.
- The two kinds of Mode type 0 DIMMs can be distinguished by selecting the larger type and waiting for the memory test to determine the size.

Mode type 0 DIMM arrays hold only small amounts of memory (< 4 MB), and some mode type 1 arrays may be only 4 MB.

3.1.2 Empty Memory Arrays

Initially, all arrays are assumed to be the maximum 64 MB in size and set to Mode type 2.

Example 4. Setting Up the SDRAM Registers

```
LDR             v5, =(SDSIZE_64M + MUX_MODE2)

; v3 holds 4 bytes each containing the bottom 8 bits of the

; DRAM_ADDR_SIZE registers: all set to biggest SDRAM part.

ADD             v3, v5, v5, LSL #8            ; 0x00bb <- 0x00b0 + 0x000b

ADD             v3, v3, v3, LSL #16           ; 0xbbbb <- 0xbb00 + 0x00bb

; 00 -> 08 -> 16 -> 24, shifts are done in this order

LDR             v4, =0x18100800

; Fix the default SDRAM registers

SETUP_RAM       $w1, $w2, $w3, v3, v4, v5, v6, a3
```

A simple write/read to a low memory location is used to determine if memory is fitted in this array.

Example 5. Testing for Empty Arrays

```
; First test each array to see if any memory present. If no memory is found,
; invalidate the appropriate byte of v3 by blanking it out. When memory is
; found, try to determine Mux mode.
    MOV         v3, #0                      ; Blank result
    MOV         a3, #MAX_SDRAM
    ADD         v5, $w1, #0xf000            ; Pointer to max_ram + 1
    LDR         $w2, =0x12345678            ; Footprint
6
    SUB         v5, v5, #SZ_64M             ; Base of next ram array
    STR         $w2, [v5]
    MVN         $w2, $w2                    ; Invert
; This inversion guarantees that the data bus isn't left floating
; with our test footprint when there is no memory present.
    STR         $w2, [v5, #4]
    MVN         $w2, $w2                    ; Back to original
    LDR         $w3, [v5]
    SUBS        $w3, $w3, $w2               ; Did it write?
; No, clear mask, but leave as 64M to make other memory tests easier.
    LDRNE       $w3, =SDSIZE_64M
    BNE         %F07
```

3.1.3 Simple Tests for Most DIMM Types

Most of the mode types can be determined by using simple aliasing tests (as described in Section 3.1.1). If a particular type of DIMM is set up incorrectly, one or more address lines will not be routed correctly. Any write to the initial location + the given address line may be aliased at the initial location. Some tests look for this aliasing effect; others specifically look for no aliasing.

Mode type 0 DIMMs do not use address line A21. Therefore, if the initial location is aliased at location + A21, the array is type 0.

Note: Because bit 0 of the size is used to route the address lines differently, this bit must be set to 0 initially to enable the larger array size to be found. This code checks for Mode 0 type DIMMs with a 64 MB array size (in Example 11). This combination can only occur during initialization because the maximum array size is 2 MB.

Mode type 2 DIMMs use address line A23 on ma[12]. Therefore, if the initial location is not aliased, the array is type 2.

Mode type 4 DIMMs use address line A22 on ma[11] (as do type 2s, but these have already been determined).

Address line A24 can find most type 1 arrays, because it drives A22 on type 1s and so maps to a valid location. This location will be mapped correctly on type 3 arrays.

Example 6. Finding the DIMM Type

```
; Yes some found, now test for different memory layouts:
; Having initially set all arrays to Mode 2, the address lines are
; driven as if for that array. So test bits correspond to that mode.
; Address line A21 distinguishes MUX_MODE0. Write inverse footprint
; at v5 + A21 - if it is written at v5, array is MUX_MODE0.
    MVN         $w2, $w2                ; Now look for holes with invert
    ADD         $w3, v5, #0x00200000    ; Add Bit21
    STR         $w2, [$w3]
    MVN         $w2, $w2                ; Back to original
    LDR         $w3, [v5]
    SUBS        $w3, $w3, $w2           ; Is original footprint still there?
; No, Mode 0 array
    LDRNE       $w3, =(SDSIZE_64M + MUX_MODE0)
    BNE         %F07
; Address line A23 distinguishes MUX_MODE2. Write inverse footprint
; at v5 + A23 - if it is written at v5, array is _not_ MUX_MODE2.
    STR         $w2, [v5]
    MVN         $w2, $w2                ; Now look for holes with invert
```

```
    ADD         $w3, v5, #0x00800000          ; Add Bit23
    STR         $w2, [$w3]
    MVN         $w2, $w2                      ; Back to original
    LDR         $w3, [v5]
    SUBS        $w3, $w3, $w2                 ; Is original footprint still there?
; Yes, Mode 2
    LDREQ       $w3, =(SDSIZE_64M + MUX_MODE2)
    BEQ         %F07
; Address line A22 distinguishes MUX_MODE4. Write inverse footprint
; at v5 + A22 - if it is written at v5, array is _not_ MUX_MODE4.
    STR         $w2, [v5]
    MVN         $w2, $w2                      ; Now look for holes with invert
    ADD         $w3, v5, #0x00400000          ; Add Bit22
    STR         $w2, [$w3]
    MVN         $w2, $w2                      ; Back to original
    LDR         $w3, [v5]
    SUBS        $w3, $w3, $w2                 ; Is original footprint still there?
; Yes, Mode 4
    LDREQ       $w3, =(SDSIZE_64M + MUX_MODE4)
    BEQ         %F07
; Mode 1 or Mode 3?
; Address line A24 can find most MUX_MODE1 Dimms, since it drives A22
; on Mode 1 Dimms, but smaller size arrays will still mirror at
; v5 + A22 (4MB). Best hope is to sort most of the Mode 1s and then
; catch 'odd' sized Mode 3s.
    STR         $w2, [v5]
    MVN         $w2, $w2                      ; Now look for holes with invert
    ADD         $w3, v5, #0x01000000          ; Add Bit24
    STR         $w2, [$w3]
    MVN         $w2, $w2                      ; Back to original
    LDR         $w3, [v5]
    SUBS        $w3, $w3, $w2                 ; Is original footprint still there?
; Yes, Mode 1 array
    LDREQ       $w3, =(SDSIZE_64M + MUX_MODE1)
```

```
; No, Mode 3 array (probably)
    LDRNE       $w3, =(SDSIZE_64M + MUX_MODE3)
7
; Okay, add this bank to the mask.
    ADD         v3, $w3, v3, LSL #8
    SUBS        a3, a3, #1
    BGT         %B06                        ; Repeat until done
```

3.1.4 Sizing the Arrays

Having determined the mode types for each array, the Address and Size registers must be set to the new values before the arrays can be sized correctly. A simple loop is used to write to locations through the maximum 64 MB of the array. This loop writes from the top down, thus avoiding any aliasing. Another loop then reads from the bottom up, checking the written data. If a location does not contain the expected data, it signals the top of the array has been passed. If an array is type 3 and the size found is not 8 MB, the array is reset as type 1 and the sizing test is repeated for this array. This procedure is used for all four arrays supported by the 21285.

3.1.5 Final Determination

Smaller size type 1 arrays will still alias at the initial location + A22, because this is greater than the size of the smallest 4 MB DIMMs. The solution is to assume that these arrays are type 3, and then check that the size really is 8 MB - if not, the array is type 1. The SDRAM Address and Size registers are written with the calculated array modes.

Example 7. Sample 7 Scanning Memory Size

```
    MOV        a4, v3                       ; Copy array masks
    ; 00 -> 08 -> 16 -> 24, shifts are done in this order
    LDR        v4, =0x18100800
    ; Fix the default SDRAM registers
    SETUP_RAM  $w1, $w2, $w3, v3, v4, v5, v6, a3
    ; Top bit means the test for offset == 0 is a valid completion test.
    LDR        a3, =0x80081018
    LDR        v6, =0                       ; Store for array of sizes found
8
    LDR        v3, =SZ_1M                   ; Test memory in 1MB steps
9
    MOV        v4, #64                      ; i.e. 64 steps per array
    SIZE_RAM   $w1, $w2, $w3, v3, v4, v5
    ; Memory sized for this bank
    AND        v5, a3, #0x3f                ; Read next array offset (no end flag)
    MOV        $w2, a4, LSR v5              ; Current DRAM_ADDR_SIZE mask
    AND        $w2, $w2, #MUX_MASK
    CMP        $w2, #MUX_MODE3              ; Check for Mode 3
    BNE        %F10
    CMP        v4, #8                       ; Can only be 8MB!
    BEQ        %F10
    ; Anything which gets here was a badly configured Mode 1 array. Clear
```

```
    ; the bad mode byte & set array to mode 1. Then do size test again.
    LDR         v4, =0xFF                   ; Mask out the Mode 3 byte
    MOV         v4, v4, LSL v5
    BIC         a4, a4, v4
    LDR         v4, =(SDSIZE_64M + MUX_MODE1)
    MOV         v4, v4, LSL v5
    ADD         a4, a4, v4                  ; Put new Mode 1 byte in its place
    MOV         v4, v4, LSR v5              ; Back to Mode 1, unshifted
    ; Reset the mode for this bank.
    LDR         v3, =(CSR_BASE + DRAM_ADDR_SIZE_0)
    MOV         v5, v5, LSR #1              ; Offset is twice register offset
    LDR         $w2, [v3, v5]               ; Read SIZE_ADDR
    BIC         $w2, $w2, #0xFF             ; Only interested in base offset
    ADD         $w2, $w2, v4                ; new mode info
    STR         $w2, [v3, v5]               ; Store back in SIZE_ADDR
    MOV         v5, v5, LSL #1              ; Restore offset
    ADD         $w1, $w1, #SZ_64M           ; Move back to top of this array
    B           %B08                        ; Do this array again as Mode 1
10
    ; Size determined, add to array.
    MOV         v4, v4, LSL v5              ; Shuffle up to correct byte
    ADD         v6, v6, v4                  ; Add the size of this array
    MOVS        a3, a3, LSR #8              ; Down to next byte (8 bits) & array
    BNE         %B09                        ; Until done
```

3.1.6 Making Memory Contiguous

Because the arrays are spaced to allow the maximum possible memory sizes, the memory map at this point is not contiguous. Also, the size of each array is determined independently and each may be a different size (in practice, only arrays on different DIMMs will differ in size). Another point to consider is that the array registers physically map to array enables on the DIMM. One DIMM may have only one array, creating a gap where the other array would be. The 21285 can map any array to any address, provided that the start address of each array is naturally aligned to its size.

In this example, due to the pressure on register usage, the bytes that hold the memory size are converted into the bit mask used in the 21285 Address and Size registers.

Example 8. Sample 8 Memory Sizes to Bit Masks

```
; v6 now contains the size of each memory array in multiples of 1 MB. This is
; very similar to the DRAM_ADDR_SIZE register array, so convert to that format
; by determining least significant bit for each array.
; Since each array must start naturally aligned with the size of the array,
; the only way to have contiguous memory is to have the arrays sorted into
; size order - largest first.
    LDR         a3, =0x18100800             ; Normal array order.
    AND         v5, v6, #0xff
    RAM_REG     v5, $w1                     ; Get DRAM_ADDR_SIZE mask for array 1
    MOV         v6, v6, LSR #8
    ADD         v6, v6, $w1, LSL #24        ; Replace size with mask
    AND         v5, v6, #0xff
    RAM_REG     v5, $w2                     ; Get mask for array 2
    MOV         v6, v6, LSR #8
    ADD         v6, v6, $w2, LSL #24        ; Replace size with mask
    AND         v5, v6, #0xff
    RAM_REG     v5, $w3                     ; Get mask for array 3
    MOV         v6, v6, LSR #8
    ADD         v6, v6, $w3, LSL #24        ; Replace size with mask
    AND         v5, v6, #0xff
    RAM_REG     v5, v4                      ; Get mask for array 4
    MOV         v6, v6, LSR #8
    ADD         v6, v6, v4, LSL #24         ; Replace size with mask
```

Because there are a maximum of four arrays, a simple bubble sort will arrange the arrays starting with the largest first. As well as sorting the sizes, the physical array numbers to which they apply are sorted too.

Example 9. Sample 9 Simple Bubble Sort

```
    ; Simple bubble sort with additional byte swap in array-order register
    CMP         $w1, $w2                    ; array 1 < array 2?
    BGE         %F21
    MOV         a4, $w1                     ; Yes, swap sizes 1 & 2
    MOV         $w1, $w2
    MOV         $w2, a4
    AND         v3, a3, #0xff               ; Now swap offsets 1 & 2
    AND         v5, a3, #0xff00
    MOV         a3, a3, LSR #16             ; clear offsets 1 & 2
    MOV         a3, a3, LSL #16
    ADD         a3, a3, v3, LSL #8          ; offset 1 -> offset 2
    ADD         a3, a3, v5, LSR #8          ; offset 2 -> offset 1
21
    CMP         $w1, $w3                    ; array 1 < array 3?
    BGE         %F22
    MOV         a4, $w1                     ; Yes, swap sizes 1 & 3
    MOV         $w1, $w3
    MOV         $w3, a4
    AND         v3, a3, #0xff               ; Now swap offsets 1 & 3
    AND         v5, a3, #0xff0000
    BIC         a3, a3, #0xff0000           ; clear offsets 1 & 3
    BIC         a3, a3, #0xff
    ADD         a3, a3, v5, LSR #16         ; offset 3 -> offset 1
    ADD         a3, a3, v3, LSL #16         ; offset 1 -> offset 3
22
    CMP         $w1, v4                     ; array 1 < array 4?
    BGE         %F23
    ; Note: just put size 1 in size 4, since the 1 value isn't used.
    MOV         v4, $w1
    AND         v3, a3, #0xff               ; Now swap offsets 1 & 4
    AND         v5, a3, #0xff000000
    BIC         a3, a3, #0xff000000         ; clear offsets 1 & 4
    BIC         a3, a3, #0xff
    ADD         a3, a3, v5, LSR #24         ; offset 4 -> offset 1
```

```
    ADD         a3, a3, v3, LSL #24          ; offset 1 -> offset 4
23
    CMP         $w2, $w3                     ; array 2 < array 3?
    BGE         %F24
    MOV         a4, $w2                      ; Yes, swap sizes 2 & 3
    MOV         $w2, $w3
    MOV         $w3, a4
    AND         v3, a3, #0xff00              ; Now swap offsets 2 & 3
    AND         v5, a3, #0xff0000
    BIC         a3, a3, #0xff0000            ; clear offsets 2 & 3
    BIC         a3, a3, #0xff00
    ADD         a3, a3, v5, LSR #8           ; offset 3 -> offset 2
    ADD         a3, a3, v3, LSL #8           ; offset 2 -> offset 3
24
    CMP         $w2, v4                      ; array 2 < array 4?
    BGE         %F25
    ; Note: just put size 2 in size 4, since the 2 value isn't used.
    MOV         v4, $w2
    AND         v3, a3, #0xff00              ; Now swap offsets 2 & 4
    AND         v5, a3, #0xff000000
    BIC         a3, a3, #0xff000000          ; clear offsets 2 & 4
    BIC         a3, a3, #0xff00
    ADD         a3, a3, v5, LSR #16          ; offset 4 -> offset 2
    ADD         a3, a3, v3, LSL #16          ; offset 2 -> offset 4
25
    CMP         $w3, v4                      ; array 3 < array 4?
    BGE         %F26
    ; Note: don't need to swap sizes 3 & 4, since the values aren't used.
    AND         v3, a3, #0xff0000            ; Now swap offsets 3 & 4
    AND         v5, a3, #0xff000000
    BIC         a3, a3, #0xff000000          ; clear offsets 3 & 4
    BIC         a3, a3, #0xff0000
    ADD         a3, a3, v3, LSL #8           ; offset 3 -> offset 4
    ADD         a3, a3, v5, LSR #8           ; offset 4 -> offset 3
26
```

3.1.7 Writing the Final Configuration

Now that the arrays are sorted by size order, the multiplexer modes are read back from the 21285 and added to the array size bit masks. These values can then be written back to the Address and Size registers to complete the memory initialization. The total memory size found is returned in r5.

Example 10. Sample 10 Completing Initialization

```
; Read previously set MUX_MODE & BANKS from 21285

LDR         $w3, =(CSR_BASE + DRAM_ADDR_SIZE_3); -> last ADDR_SIZE re.

LDR         v3, [$w3], #-4                ; Current array 4 ADDR_SIZE

AND         v3, v3, #MUX_MASK             ; Strip out good Mux mode/bank value

LDR         v5, [$w3], #-4                ; Current array 3 ADDR_SIZE

AND         v5, v5, #MUX_MASK

ADD         v3, v5, v3, LSL #8            ; v3 = v5 + (v3 << 8);

LDR         v5, [$w3], #-4                ; Current array 2 ADDR_SIZE

AND         v5, v5, #MUX_MASK

ADD         v3, v5, v3, LSL #8            ; v3 = v5 + (v3 << 8);

LDR         v5, [$w3]                     ; Current array 1 ADDR_SIZE

AND         v5, v5, #MUX_MASK

ADD         v3, v5, v3, LSL #8            ; v3 = v5 + (v3 << 8);

ADD         v3, v3, v6                    ; Add size masks to modes

MOV         v4, a3                        ; Byte order needs to be in v4

; Finally, fix the 21285 SDRAM registers for calculated sizes

SETUP_RAM  $w1, $w2, $w3, v3, v4, v5, v6, a3

; set up the return arguments.

MOV         $w3, $w1

MEND
```

3.1.8 Supplementary Macros

To reduce the complexity of the INIT_RAM macro, some common functionality has been broken out into supplementary macros.

SETUP_RAM sets the memory timing register and the array size registers on the 21285. Because there are a maximum of four arrays, the Address Size masks and offsets between arrays (in megabytes) are kept as bytes in registers $w4 and $w5. This dramatically simplifies the final setup, where each array could have different mode types and memory sizes.

Note: This code checks for Mode 0 type DIMMs with a 64 MB array size. This combination can only occur during initialization (maximum array size is 2 MB). Because bit 0 of the size is used to route the address lines differently, this bit must be set to 0 to enable the larger array size to be found.

Example 11. Sample 11 Memory Timing and Array Size Registers

```
    ; ------------------------------------------------------------------
    ; SETUP_RAM
    ; ---------
    ; ANGEL and uHAL macro to setup memory timing/size registers in 21285
    ; on start-up. 21285 only has 1 timing register, so DIMMs must work with
    ; same timing. Each array is then configured for no. of banks & size.
    ;          $w1 -> returns the size of memory allocated
    ;          $w4 -> byte array of ADDR_SIZE masks (33221100)
    ;          $w5 -> byte array of offsets between arrays (max 64 * 1MB)
    ;          $s2, $s3, $s6 - $s8 -> scratch registers.
    MACRO
    SETUP_RAM  $w1, $s2, $s3, $w4, $w5, $s6, $s7, $s8
    MOV        $w1, #DRAM_BASE              ; Start of RAM
    LDR        $s3, =CSR_BASE
    ; This enables the test for offset == 0 to be a valid completion test.
    ADD        $w5, $w5, #0x80000000
    LDR        $s2, =SETUP_TIMING
    STR        $s2, [$s3, #DRAM_TIMING]     ; Default timing
    ADD        $s3, $s3, #DRAM_ADDR_SIZE_0  ; -> 1st ADDR_SIZE reg.
1
    AND        $s6, $w5, #0x3f              ; Read next array offset
    MOV        $s2, $w4, LSR $s6            ; Shuffle register value down
    AND        $s2, $s2, #0xff              ; Register value (- base addr)
    ANDS       $s8, $s2, #0x7               ; Mask array size & test for empty
```

```
    ; When mask is empty, array is disabled
    BEQ         %F02
    ; Else, convert bit pattern to array size = (1 << ($s8 - 1))
    ; Only increment memory if the mask is not empty
    SUB         $s8, $s8, #1
    MOV         $s7, #1
    MOV         $s7, $s7, LSL $s8
    MOV         $s7, $s7, LSL #20             ; array size * 1M
    ADD         $s2, $s2, $w1                 ; add base of this array
    ADD         $w1, $w1, $s7                 ; Offset to next array
    ; If Array size is 64MB and mode is 0, must use 32MB to
    ; differentiate correct size (using bit18 on row ma[9])
    CMP         $s2, #0x7                     ; NOTE: No bits set for Mode 0
    SUBEQ       $s2, $s2, #1
2
    ; Writing zero into the ADDR_SIZE register disables this array
    MOV         $s6, $s6, LSR #1              ; Offset is twice register offset
    STR         $s2, [$s3, $s6]               ; Set up 21285 register
    MOVS        $w5, $w5, LSR #8              ; Get next ADDR_SIZE byte offset
    BNE         %B01                          ; Until done
    MEND
```

The SIZE_RAM macro is a totally flexible macro that allows the start address, step size, and number of steps in the memory test to be specified.

Example 12. Sample 12 Sampling Memory for Array Size

```
    ; ------------------------------------------------------------------------
    ; SIZE_RAM
    ; ---------
    ; ANGEL and uHAL macro to scan memory to evaluate size. Assumes that
    ; there are gaps between arrays so that memory will return invalid
    ; data (either from a gap or mirrored memory).
    ;          $w4 -> step size for memory test
    ;          $w5 -> number of steps (max array size / step size)
    ;          -> returns memory size found as multiple of no. of steps
    ;          $w1 -> address of top of current array (max possible)
    ;          -> returns address of bottom of current array
    ;          $w2, $w3, $w6 -> scratch registers.
    MACRO
    SIZE_RAM    $w1, $w2, $w3, $w4, $w5, $w6
    MOV         $w6, $w5
    ADD         $w1, $w1, #0x3c00
1
    SUB         $w1, $w1, $w4
    SUBS        $w5, $w5, #1                ; Countdown 64 -> 0 write
    STR         $w5, [$w1]                  ; footprint in memory
    BGT         %B1
    MOV         $w3, $w1                    ; Pointer to base_ram
    STR         $w5, [$w3, #4]              ; Shouldn't need to worry about
2                                           ; floating data bus, but..
    LDR         $w2, [$w3]                  ; Now check looking for unexpected
    SUBS        $w2, $w2, $w5               ; memory contents
    BNE         %F3                         ; End of array found
    ADD         $w3, $w3, $w4
    ADD         $w5, $w5, #1                ; Count up 0 -> 64 read
    CMP         $w5, $w6
    BLT         %B2
3
    MEND
```

RAM_REG is a simple macro to convert an array size into the appropriate mask value used by the DRAM_ADDR_SIZE registers on the 21285. The primary benefit of having this as a macro is to avoid typing errors in multiple uses.

Example 13. Sample 13 Array Size to Mask Value

```
    ; ---------------------------------------------------------------------
    ; RAM_REG
    ; -------
    ; ANGEL and uHAL macro to convert the memory size into a size mask for
    ; the DRAM_ADDR_SIZE registers. Looks for the LSB (since each array is
    ; a power of 2 in size) and returns the number of zero’s bits which is
    ; the mask..
    ;          $w1 -> size of array (multiple of 1MB)
    ;          $w2 -> returns mask value
    MACRO
    RAM_REG    $w1, $w2                    ; Convert to DRAM_ADDR_SIZE mask
    MOV        $w2, #0
    CMP        $w1, $w2
    BEQ        %F02                        ; Empty - nothing to do
1
    ADD        $w2, $w2, #1
    TST        $w1, #1                     ; LSB set?
    MOV        $w1, $w1, LSR #1
    BEQ        %B01                        ; No, try next shuffled down bit
2
    MEND
```

4.0 Summary

In a fixed memory subsystem, SDRAM can be initialized very quickly and simply. Where memory can be added or taken away, initialization is more complex due to the vagaries of DIMM addressing and the stipulation that each address array must start on a naturally aligned boundary.

For a list of DIMMs known to work with the EBSA-285, see the EBSA-285 Evaluation Board Reference Manual.

Because this code is freely reusable in StrongARM** systems and distributed with evaluation kits, it is expected that new designs should be able to implement and verify SDRAM memory subsystems very rapidly.

Support, Products, and Documentation

If you need technical support, a *Product Catalog*, or help deciding which documentation best meets your needs, visit the Intel World Wide Web Internet site:

http://www.intel.com

Copies of documents that have an ordering number and are referenced in this document, or other Intel literature may be obtained by calling **1-800-332-2717** or by visiting Intel's website for developers at:

http://developer.intel.com

You can also contact the Intel Massachusetts Information Line or the Intel Massachusetts Customer Technology Center. Please use the following information lines for support:

For documentation and general information:	
Intel Massachusetts Information Line	
United States:	1–800–332–2717
Outside United States:	1–303-675-2148
Electronic mail address:	techdoc@intel.com

For technical support:	
Intel Massachusetts Customer Technology Center	
Phone (U.S. and international):	1–978–568–7474
Fax:	1–978–568–6698
Electronic mail address:	techsup@intel.com