Application Report
SPRA568A - February 2002

TMS320C6000 EMIF to External Flash Memory

Kyle Castille Digital Signal Processing Solutions

ABSTRACT

Interfacing external flash memory to the Texas Instruments TMS320C6000™ digital signal processor (DSP) is simple compared to previous generations of TI DSPs. The TMS320C6000 advanced external memory interface (EMIF) provides a glueless interface to a variety of external memory devices. The sample code described in this application report can be downloaded from http://www.ti.com/lit/zip/SPRA568.

This document describes the following:

- EMIF control registers and asynchronous interface signals
- Flash functionality and performance considerations
- Full example using AMD's AM29LV800
- Full example using AMD's AM29LV040

Contents

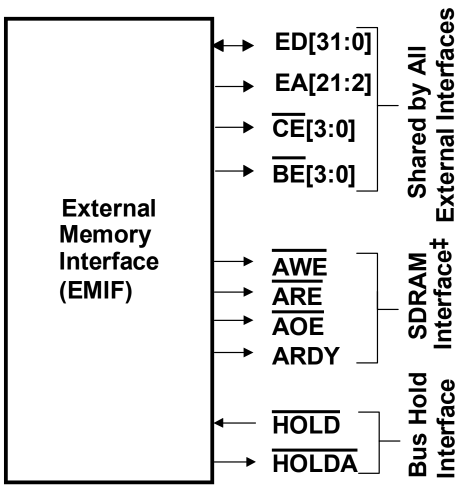
List of Figures

List of Tables

1 Overview of EMIF

1.1 EMIF Signal Descriptions

Figure 1 shows a basic block diagram of the EMIF asynchronous interface. The EMIF is the interface between external memory and the other internal units of the C6000™. The signals described in Table 1 focus on the asynchronous interface and the shared interface signals.



† The C64x has two EMIFs (EMIFA and EMIFB). A prefix "A" can be placed in front of a signal name indicating it is an EMIFA signal whereas a prefix "B" indicates an EMIFB signal. In generic EMIF areas of discussion throughout this document, the prefix "A" and "B" may be omitted from the signal name.
‡ MUXed with SDRAM and SBSRAM pins on C621x/C671x/C64x

Figure 1. Basic Block Diagram of C6000 EMIF†

C6000 is a trademark of Texas Instruments.

Table 1. EMIF Signal Descriptions: Shared Signals and Asynchronous Signals†

C620x/ C670x Interface	C61x/ C671x Interface	C64x EMIFA Interface	C64x EMIFB Interface	(I/O/Z)	Description
CLKOUT1	ECLKOUT	AECLKOUT1	BECLKOUT2	O	Clock. Used as asynchronous interface timing reference.
ED(31:0)	ED(31:0)	AED(63:0)	BED(15:0)	I/O/Z	Data I/O. Data input/output from external memories and peripherals.
EA(21:2)	EA(21:2)	AEA(22:3)	BEA(20:1)	O/Z	External address output. Drives the specified bits of the byte address.
$\overline{\text{CE0}}$	$\overline{\text{CE0}}$	$\overline{\text{ACE0}}$	$\overline{\text{BCE0}}$	O/Z	External $\overline{\text{CE0}}$ chip-select. Active-low chip-select for CE space 0.
$\overline{\text{CE1}}$	$\overline{\text{CE1}}$	$\overline{\text{ACE1}}$	$\overline{\text{BCE1}}$	O/Z	External $\overline{\text{CE1}}$ chip-select. Active-low chip-select for CE space 1.
$\overline{\text{CE2}}$	$\overline{\text{CE2}}$	$\overline{\text{ACE2}}$	$\overline{\text{BCE2}}$	O/Z	External $\overline{\text{CE2}}$ chip-select. Active-low chip-select for CE space 2.
$\overline{\text{CE3}}$	$\overline{\text{CE3}}$	$\overline{\text{ACE3}}$	$\overline{\text{BCE3}}$	O/Z	External $\overline{\text{CE3}}$ chip-select. Active-low chip-select for CE space 3.
$\overline{\text{BE}}$(3:0)	$\overline{\text{BE}}$(3:0)	$\overline{\text{ABE}}$(7:0)	$\overline{\text{BBE}}$(1:0)	O/Z	Byte enables. Active-low byte strobes. Individual bytes and halfwords can be selected for both read and write cycles. Decoded from 2 LSBs (least significant bits) of the byte address.
ARDY	ARDY	AARDY	BARDY	I	Ready. Asynchronous ready input used to insert wait states for slow memories and peripherals, such as flash memory.
$\overline{\text{AOE}}$	$\overline{\text{AOE/}}$ SDRAS/ SSOE	$\overline{\text{AAOE/}}$$\overline{\text{ASDR}}$AS/ ASOE	$\overline{\text{BAOE/}}$ BSDRAS/ BSOE	O/Z	Output enable. Active-low output enable for asynchronous memory interface.
$\overline{\text{AWE}}$	$\overline{\text{AWE/}}$ SDWE/ SSWE	$\overline{\text{AAWE/}}$ ASDWE/ ASWE	$\overline{\text{BAWE/}}$ BSDWE/ BSWE	O/Z	Write strobe. Active-low write strobe for asynchronous memory interface.
$\overline{\text{ARE}}$	$\overline{\text{ARE/}}$ SDCAS/ SSADS	$\overline{\text{AARE/}}$ ASDCAS/ ASADS/ ASRE	$\overline{\text{BARE/}}$ BSDCAS/ BSADS/ BSRE	O/Z	Read strobe. Active-low read strobe for asynchronous memory interface.
$\overline{\text{HOLD}}$	$\overline{\text{HOLD}}$	$\overline{\text{AHOLD}}$	$\overline{\text{BHOLD}}$	I	Active-low external bus-hold (3-state) request
$\overline{\text{HOLDA}}$	$\overline{\text{HOLDA}}$	$\overline{\text{AHOLDA}}$	$\overline{\text{BHOLDA}}$	O	Active-low external bus-hold acknowledge

† The C64x has two EMIFs (EMIFA and EMIFB). The prefix “A” in front of a signal name indicates it is an EMIFA signal whereas a prefix “B” indicates an EMIFB signal. In generic EMIF areas of discussion throughout this document, the prefix “A” and “B” may be omitted from the signal name.

1.2 EMIF Registers

Control of the EMIF and the memory interfaces it supports is maintained through a set of memory-mapped registers within the EMIF. EMIF registers should not be modified while the register is in use. The memory-mapped registers are shown in .

Table 2. EMIF Memory Mapped Registers

Byte Address			
EMIF and EMIFA	**EMIFB †**	**Abbreviation**	**Description**
0x01800000	0x01A80000	GBLCTL	EMIFx Global Control
0x01800004	0x01A80004	CE1CTL	EMIFx CE1 Space Control
0x01800008	0x01A80008	CE0CTL	EMIFx CE0 Space Control
0x0180000C	0x01A8000C		Reserved
0x01800010	0x01A80010	CE2CTL	EMIFx CE2 Space Control
0x01800014	0x01A80014	CE3CTL	EMIFx CE3 Space Control

† EMIFA and EMIFB are available on C64x only.

1.2.1 CE Space Control Registers

The four CE space control registers correspond to the four CE spaces supported by the EMIF (see Figure 2 and Figure 3). The MTYPE field identifies the memory type for the corresponding CE space. All fields of the CE space control register are used for the flash interface. When programming an external flash for a TMSC620x/C670x device, MType is set to a standard 32-bit asynchronous interface, but this is not necessary for the TMS320C621x/C671x/C64x™. When reading from flash, the MType is set according to the width of the external interface. The remaining fields control the strobe timing according to the specific flash selected.

31–28	27–22	21–20	19–16
WRITE SETUP	WRITE STROBE	WRITE HOLD	READ SETUP
R/W, +1111	R/W, +111111	R/W,+11	R/W, +1111

15–14	13–8	7	6–4	3–2	1–0
Reserved	READ STROBE	Rsv	MTYPE	Reserved	READ HOLD
R,+11	R/W,+111111	R,+0	R/W,+010	R,+0	R/W,+11

Figure 2. C6201/C6202/C6701 EMIF CE Space Control Register Diagram

31–28	27–22	21–20	19–16
WRITE SETUP	WRITE STROBE	WRITE HOLD	READ SETUP
R/W,+1111	R/W,+111111	R/W,+11	R/W,+1111

15–14	13–8	7–4	3	2–0
TA	READ STROBE	MTYPE	Write Hold MSB	READ HOLD
R/W,+11	R/W,+111111	R/W,+010	R/W,+0	R/W,+011

Figure 3. C6211/C6711/C64x EMIF CE Space Control Register Diagram

C64x is a trademark of Texas Instruments.

Table 3. EMIF CE Space Control Registers Bitfield Description

Field	Description
READ SETUP WRITE SETUP	Setup width. Number of clock† cycles of setup for address (EA) and byte enables ($\overline{BE}$(0–3)) before read strobe (ARE) or write strobe (AWE) falling. On the first access to a CE space, this is also the setup after CE falling.
READ STROBE WRITE STROBE	Strobe width. The width of read strobe ($\overline{ARE}$) and write strobe ($\overline{AWE}$) in clock† cycles.
READ HOLD WRITE HOLD	Hold width. Number of clock† cycles that address (EA) and byte strobes ($\overline{BE}$(0–3)) are held after read strobe (ARE) or write strobe (AWE) rising. These fields are extended by one bit on the C6211/C6711/C64x.

MTYPE	Memory type
	C6201/C6202/C6701 only:
	MTYPE = 000b: 8-bit-wide ROM (CE1 only)
	MTYPE = 001b: 16-bit-wide ROM (CE1 only)
	MTYPE = 010b: 32-bit-wide asynchronousinterface
	C6211/C6711/C64x only:
	MTYPE = 0000b: 8-bit-wide asynchronous interface
	MTYPE = 0001b: 16-bit-wide asynchronous interface
	MTYPE = 0010b: 32-bit-wide asynchronous interface
	MTYPE = 1100b: 64-bit-wide asynchronous interface§

TA‡	Turnaround time. Controls the number of ECLKOUT cycles between a read and a write or between two reads.

† Clock = CLKOUT1 for C620x/C670x. Clock = ECLKOUT for C6211/C6711. Clock = ECLKOUT1 for C64x
‡ Applies to C6211/C6711/C64x only
§ Applies to C64x only

1.3 C620x/C670x EMIF ROM Modes

The C620x/C670x EMIF supports 8-, 16-, and 32-bit-wide ROM access modes, as selected by the MTYPE field in the EMIF CE space control register. In reading data from these narrow-width memory spaces, the EMIF packs multiple reads into one 32-bit value. This mode is primarily intended for word access to 8-bit and 16-bit ROM devices, and operates as follows:

- Read operations always read 32 bits, regardless of the access size or the memory width. For example, a byte read from an 8-bit ROM mode reads 4 bytes, and extracts the correct byte needed by the DMA (direct memory access) or CPU.
- The address is shifted up appropriately to provide the correct address to the narrow memory. The shift amount is 1 bit for a 16-bit ROM and 2 bits for an 8-bit ROM. Thus, the high address bits are shifted out and accesses wrap around, if that CE space uses the entire EA bus.
- The EMIF always reads the lower addresses first and packs these in the LSBytes, and packs subsequent accesses into the higher-order bytes. Thus, the expected packing format in ROM is always little-endian, regardless of the value of the LENDIAN bit.
- Write operations to a ROM space are the same as write accesses to a 32-bit asynchronous CE space. No address shifting or byte packing is done.

1.4 C6211/C6711/C64x EMIF x8/x16/x32/x64 Asynchronous Modes

The C6211/C6711 EMIF supports 8- and 16-bit bus widths and the C64x supports 8-, 16-, 32-, and 64-bit bus widths for all memory types, including asynchronous MTypes. When reading data from these narrow-width memory spaces, the EMIF packs multiple reads into one 32-bit value, or 64-bit value for C64x EMIFA. This mode is used for both reading from, and writing to, external memories.

- Read operations occur according to the size of the data requested. For example, a byte read from an 8-bit ROM mode reads only 1 byte. A word read from an 8-bit ROM reads 4 bytes.
  - The address is shifted up appropriately to provide the correct address to the narrow memory. The shift amount is 1 bit for a 16-bit ROM and 2 bits for an 8 bit ROM. Thus, the high address bits are shifted out and accesses wrap around, if that CE space uses the entire EA bus.
  - The EMIF always reads the low addresses first and packs these according to the endianness of the system. The packing format can be little-endian or big-endian.
- Write operations occur according to the size of the data being written. For example, a byte write to an 8-bit ROM writes only one 8-bit value to the correct external address. A word write to an 8-bit asynchronous device writes four successive bytes.
  - The address is shifted up appropriately to provide the correct address to the narrow memory. For example, on a C6211 device with 32-bit bus widththe shift amount is 1 bit for a 16-bit asynchronous memory and 2 bits for an 8-bit asynchronous memory. Thus, the high address bits are shifted out and accesses wrap around, if that CE space uses the entire EA bus.
  - The EMIF always writes the low addresses first and packs these according to the endianness of the system. The packing format can be little-endian or big-endian.

1.4.1 Byte Lane Alignment on the C6211/C6711 EMIF

The C6211/C6711 EMIF offers the capability to interface to 32-, 16-, and 8-bit-wide memories. Depending on the endianness of the system, a different byte lane is used for all memory interfaces. The alignment required is shown in Figure 4.

Note that BE3 always corresponds to ED[31:24], BE2 always corresponds to ED[23:16], BE1 always corresponds to ED[15:8], and BE0 always corresponds to ED[7:0], regardless of endianness.

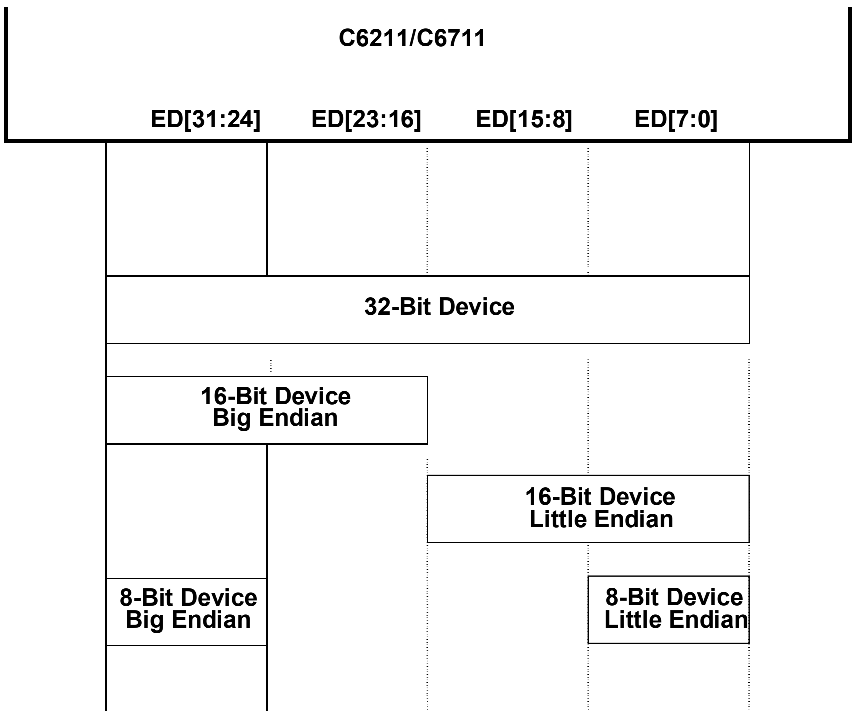


Figure 4. Byte Lane Alignment vs. Endianness on the C6211/C6711

1.4.2 C64x Byte-Lane Alignment

The C64x EMIFA offers the capability to interface to 64-, 32-, 16-, and 8-bit-wide memories. EMFIB supports interfaces to 16-bit and 8-bit memories. Figure 5 and Figure 6 show the byte lanes used on the C64x for EMIFA and EMIFB.

Unlike the previous C6000 devices, the external memory on the C64x is always right-aligned to the ED[7:0] side of the bus. The endianness mode determines whether byte lane 0 (ED[7:0]) is accessed as byte address 0 (little endian) or as byte address N (big endian), where 2^N is the memory width in bytes.

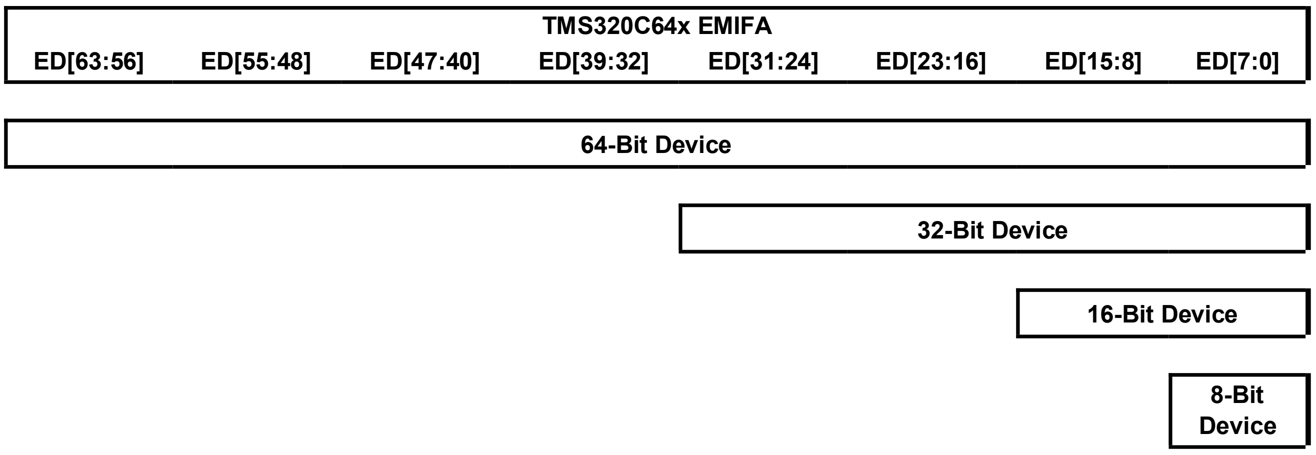


Figure 5. EMIFA (64-Bit Bus) Byte Alignment by Endianness

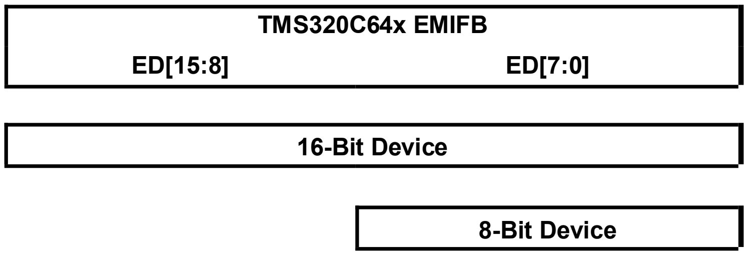


Figure 6. EMIFB (16-Bit Bus) Byte Alignment by Endianness

2 Flash Memory Interface

The asynchronous interface offers users configurable memory cycle types used to interface to a variety of memory and peripheral types, including SRAM, EPROM, and flash, as well as FPGA and ASIC designs. However, this section focuses on the interface between the EMIF and flash memory, which is very similar to a ROM or EEPROM interface.

Table 3 shows that 8-, 16-, 32-, and 64-bit-wide (C64x EMIFA only) configurations are supported by the EMIF via its asynchronous interface. Flash memory is commonly available on the market in either 8-bit-wide devices or configurable 8-/16-bit-wide devices. The configurable devices generally have 16 data I/O lines, but a mode select pin tells the device whether to operate in 8- or 16-bit mode.

If more depth is required than can be provided via an 8- or 16-bit-wide flash memory device, then these devices can be used in parallel to create a x32/x64 bit interface. Due to the asynchronous interface, some logic will be needed between the RY/$\overline{\text{BY}}$ and the EMIF $\overline{\text{ARDY}}$ signal to verify that all the flash devices are ready. If glue is not used in the interface, then all the D[x] bits from the Flash will need to be polled to verify that all the 8-or 16-bit ROMs are ready.

Table 4 lists the EMIF asynchronous interface pins and their mapping to pins on common flash memory. Figure 7 shows an interface to 8-/16-bit wide standard flash utilizing the 16-bit mode. In 8-bit mode, D[15] operates as the least significant bit of the address, thus giving a 2^{n+2}-byte address space. Figure 8 shows an interface using an 8-bit-wide device.

Notice that the diagram shows no common clock interface between the C6000 and flash, as indicated by the term *asynchronous*. The EMIF still uses the internal clock to coordinate the timing of its signals, however, the flash responds to the signals at its inputs regardless of any clock.

Table 4. EMIF Asynchronous Interface Pins

EMIF Signal	Flash Signal	Function
$\overline{\text{AOE}}$	$\overline{\text{OE}}$	Output enable. Active-low during the entire period of a read access
$\overline{\text{AWE}}$	$\overline{\text{WE}}$	Write enabl. Active-low during a write transfer strobe period
$\overline{\text{ARE}}$	N/A	Read enable. Active-low during a read transfer strobe period. Although not connected to flash memory, it is still used logically to determine when the data is read by the EMIF.
ARDY	RY/$\overline{\text{BY}}$	Ready input used to insert wait states into the memory cycle. Hardware method determines if flash memory is currently in program cycle or erase cycle. RY/BY high indicates device is ready for next operation. Low indicates that device is busy in either program or erase cycle (not on all devices).
N/A	$\overline{\text{BYTE}}$	For 8-/16-bit devices, determines if the device will be used in byte mode or in double-byte mode. BYTE-low selects byte mode. BYTE-high selects double-byte mode (not on all devices).

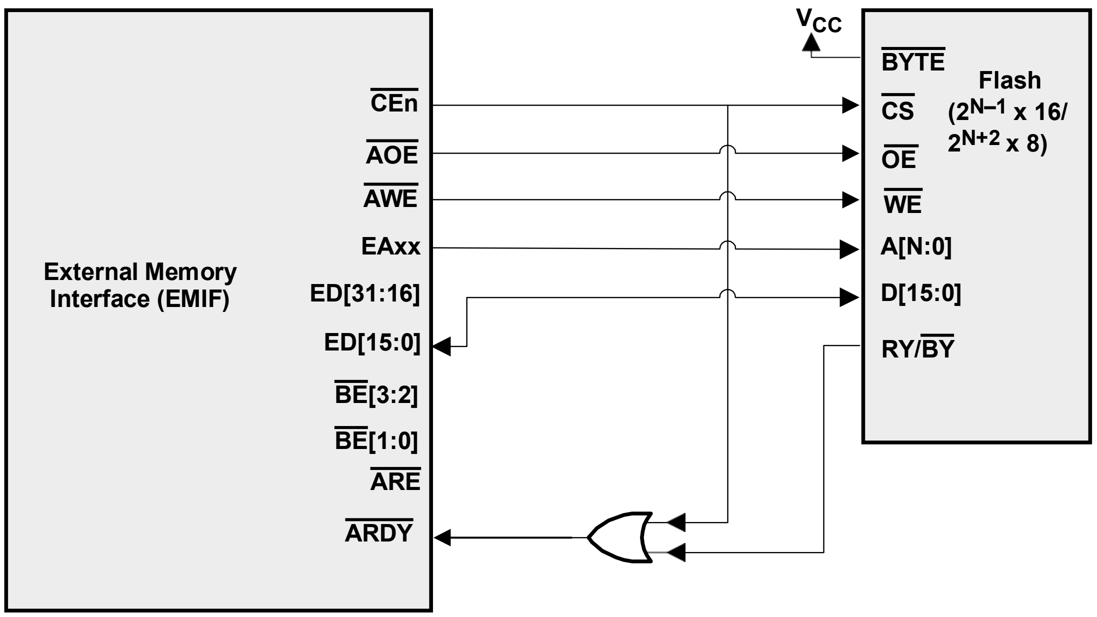


Figure 7. EMIF 8-/16-Bit Flash Interface With ARDY Interface (16-Bit Mode)

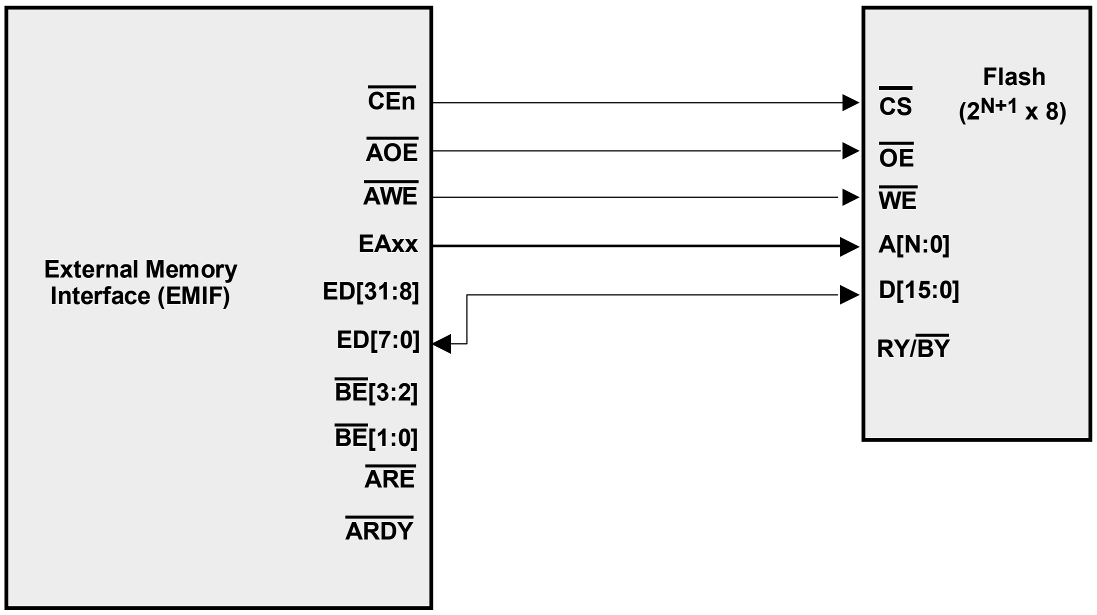


Figure 8. EMIF 8-Bit Flash Interface Without ARDY Interface

2.1 Flash Functionality and Common Commands

Flash is a read-only memory device that has the capability of being reprogrammed in a target system, providing the user with a cost-efficient means of maintaining a system that may require code or data changes in the future. Typically, flash read cycles are compatible with the asynchronous timing provided by the C6000. Writing data to flash is more complex, requiring a sequence of writes to command registers in order to execute a command.

Table 5 lists a few of the common JEDEC compatible commands for x8/x16 devices. Devices that are x8 only have a slightly different programming scheme, which is shown in Table 6. The user should verify the programming codes for a particular device in the appropriate data sheet.

Table 5. Flash Commands for AMD's AM29LV800B Devices†

		First		Second		Third		Fourth		Fifth		Sixth	
Command	Cycles	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data	Addr	Data
Read	1	RA	RD										
Reset	1	X	F0h										
Chip Erase (8-bit)	6	AAA	AA	555	55	AAA	80	AAA	AA	555	55	AAA	10
Chip Erase (16-bit)	6	555	AA	2AA	55	555	80	555	AA	2AA	55	555	10

† Although this programming format is common to many x8/x16 devices, the programming codes for a particular device should be verified before using.

NOTES: 1. RA = read address

- RD = read data
- PA = program address
- PD = program data

Table 5. Flash Commands for AMD's AM29LV800B Devices† (Continued)

		First		Second		Third		Fourth		Fifth		Sixth	
Command	**Cycles**	**Addr**	**Data**	**Addr**	**Data**	**Addr**	**Data**	**Addr**	**Data**	**Addr**	**Data**	**Addr**	**Data**
Program (8-bit)	4	AAA	AA	555	55	AAA	A0	PA	PD				
Program (16-bit)	4	555	AA	2AA	55	555	A0	PA	PD				

† Although this programming format is common to many x8/x16 devices, the programming codes for a particular device should be verified before using.

NOTES: 1. RA = read address
2. RD = read data
3. PA = program address
4. PD = program data

Table 6. Flash Commands for AMD's AM29LV040B Devices†

		First		Second		Third		Fourth		Fifth		Sixth	
Command	**Cycles**	**Addr**	**Data**	**Addr**	**Data**	**Addr**	**Data**	**Addr**	**Data**	**Addr**	**Data**	**Addr**	**Data**
Read	1	RA	RD										
Reset	1	X	F0h										
Chip erase (8 bit)	6	5555	AA	2AAA	55	5555	80	5555	AA	2AAA	55	5555	10
Program (8 bit)	4	5555	AA	2AAA	55	5555	A0	PA	PD				

† Although this programming format is common to many x8/x16 devices, the programming codes for a particular device should be verified before using.

NOTES: 1. RA = read address
2. RD = read data
3. PA = program address
4. PD = program data

2.1.1 Read/Reset Command

Issuing the reset command to the flash memory activates the read mode. The device remains in this mode until another valid command sequence is input in the command register. The device enters this mode by default on hardware reset of if an invalid command sequence is entered. In addition, when any chip or erase operation finishes, the device returns to read mode.

2.1.2 Chip Erase Command

Chip erase is a six-bus-cycle command sequence that must be executed if reprogramming a device. This command physically erases the entire address space. Before data can be programmed to the device, the addresses to be programmed must be erased, either by erasing the whole chip or the sector in which the addresses reside. After the last rising edge of $\overline{\text{WE}}$, the chip erase operation begins, and any additional commands written to the device are ignored until the erase operation is complete.

2.1.3 Program Command

The program command writes new data to the device. For each byte, double byte, or word to be programmed, first a three-write-cycle sequence is issued, followed by the address and data to actually be written to the device. The rising edge of $\overline{\text{WE}}$ starts the program operation. Additional commands written to the device during the program operation are ignored until the program operation is complete.

2.1.4 Other Commands

Although the commands listed above are common to most devices, some devices may have their own version of these commands, which may or may not be JEDEC-compatible. Other common commands include a *sector erase command* that allows the user to erase only a single sector of the flash memory. This command would take less time than erasing the whole device, and is useful if only a small amount of data or code is going to be changed. Another common feature is known as *unlock bypass*, which allows 2-cycle write commands rather than the standard 4-cycle writes.

Other less common features include *erase suspend/erase resume*, which allows the erase command to be paused in a given sector so that reads or writes to another sector can be performed. With this command, the sector being erased cannot be read or written to until the erase operation is complete.

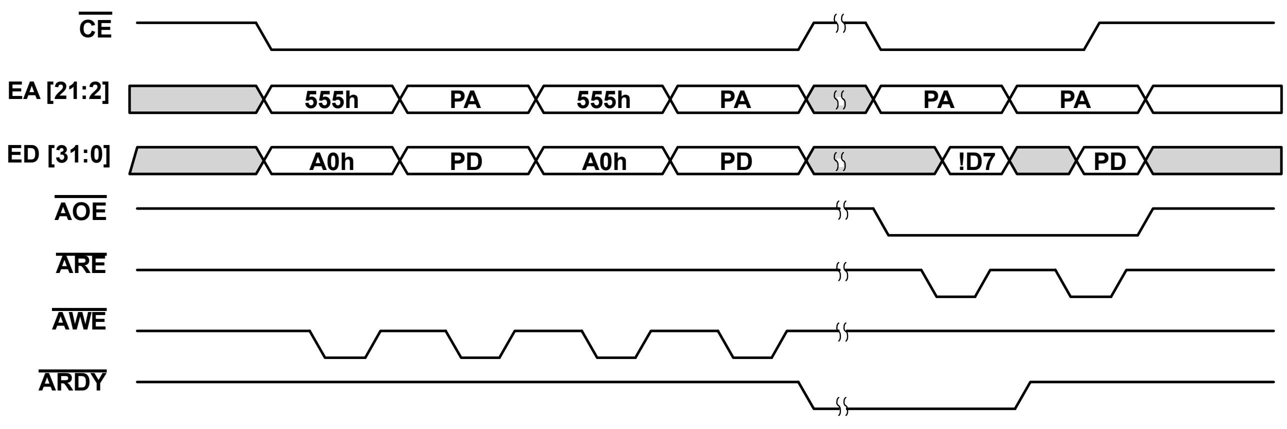
2.2 Device Status

The status of the device during a program operation or chip erase operation can be determined either by software or by both hardware and software, depending on the functionality of the specific flash memory.

Software monitoring can be done by reading the state of the D[7] pin, which is called the *data polling* pin. The status of this bit during a program operation is the complement of the bit written to this pin during the program command, until the program operation completes. When the program operation completes, a data read returns the correct data on all pins, including D[7]. During an erase operation, the data read on this pin is a zero. When the erase operation is complete, a read returns a one on D[7]. Depending on the functionality of the flash, additional pins are used in combination to define the function currently in operation.

Figure 9 shows a waveform illustrating a program command sequence, followed by data polling to determine when the program operation completes. This diagram assumes that either the RY/$\overline{\text{BY}}$ signal is NOT connected to the ARDY pin of the EMIF, or the flash device does NOT support RY/$\overline{\text{BY}}$ functionality. The RY/$\overline{\text{BY}}$ signal is shown to illustrate the beginning and end of the program operation. As seen, before the program operation is completed, the RY/$\overline{\text{BY}}$ signal is low, and a read from the address just programmed returns the complement of the programmed data at bit 7. When the program operation completes, a read returns valid data, indicating that the device is ready for the next operation.

If the flash cannot finish a program or erase operation for some reason, the D[5] pin can be used to detect a time-out condition. A value of 1 on D[5], while D[7] is still not equal to data, indicates a time out. To execute any more commands, a reset command must be issued to the flash.



Program Command Using Data Polling

Figure 9. Program Command Without ARDY Interface – Software Monitoring

Most flash memory devices have this data polling functionality, whether or not they include the RY/$\overline{BY}$ pin. However, data polling is not possible when the RY/$\overline{BY}$ pin is connected to the ARDY pin of the C6000 because, if the flash device is busy (indicated by a low level on the RY/$\overline{BY}$ pin and the ARDY pin of the C6000), the C6000 cannot complete a read to the flash address space until the flash memory completes its current operation and RY/$\overline{BY}$ is sent high.

If the device being used has the RY/$\overline{BY}$ function, no software intervention is necessary. A low level on this pin informs the EMIF to extend any future read or write cycles as long as necessary until the state of the pin changes, informing the EMIF that the flash is ready.

Figure 10 shows a waveform illustrating a program command sequence, followed by a read to the same address location. This figure assumes that the RY/$\overline{BY}$ signal is connected to the ARDY input of the C6000. The read cycle is extended as long as there is a low level on the ARDY pin. Although this example uses a read cycle to illustrate the ARDY operation, if the next cycle after the program command were a write, the same cycle extension would occur. Therefore, the C6000 can begin writing the next program sequence at any time without any software control because the EMIF prevents any additional reads or writes until the ARDY signal goes high.

Although the RY/$\overline{BY}$ signal provides a simple interface to the EMIF, there is a limitation if the operation is unable to complete. If software polling were used without the ARDY interface, this condition could be detected by monitoring the D[5] pin. This condition cannot be detected with the ARDY interface because, if the operation cannot complete, the ARDY input will be low and a read cannot complete to the flash. This can be worked around by forcing a hardware reset, if the ARDY signal is low for a specified number of cycles.

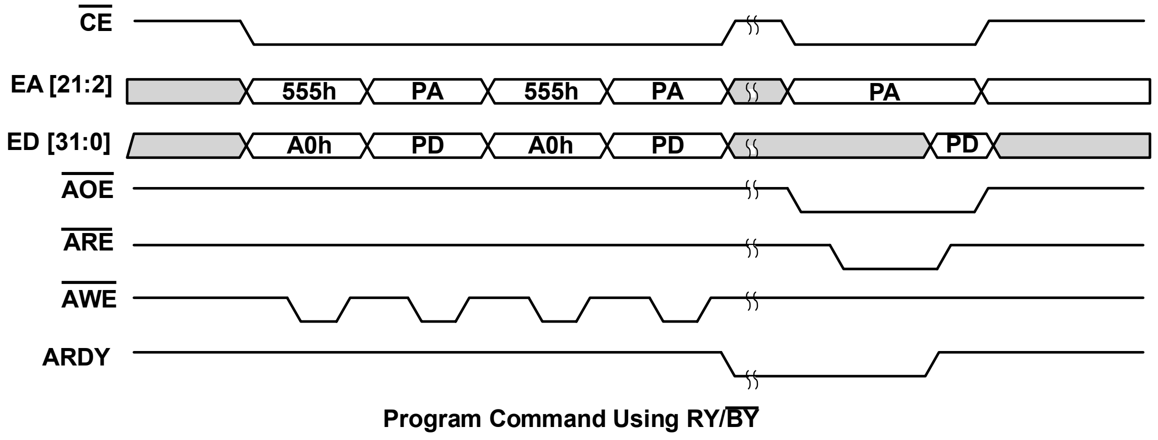


Figure 10. Program Command With ARDY Interface – Hardware Monitoring

2.3 Byte Addressing and Shifting on the EA Bus

When the CPU or DMA accesses the EMIF bus, the external address is shifted according to the bus width of the external device and the size of the access. For most memory interfaces this is transparent to the user, since the EMIF automatically handles alignment and packing issues. The same is true for reads from a Flash memory. During writes to a Flash memory, however, the address presented to the external Flash must be carefully controlled, since the address bus is used to control the type of command issued to the flash.

During writes to a flash memory device, the EMIF must make sure that the address bits sent across the bus to the flash device will be correctly received on the correct pins. With the C6x0x devices, only 32-bit writes are supported so the address shifting is always consistent. The C6x1x and C64x devices allow 8-bit, 16-bit, 32-bit, and 64-bit-wide write accesses, depending on the device.

For the C621x/C671x and C64x devices, no problems occur for an 8-bit interface, since the desired opcode can be used directly. In this mode, the logical address is sent out directly on the external address bus, EA[LSB+20:LSB] ,as can be seen in Table 7. A0 is always driven out on the LSB of the external address. For all the other bus widths, the logical address must be shifted appropriately to force the opcode to be presented correctly on the external address pins. A 16-bit interface requires the opcode to be left shifted one position so that OPCODE[0] = logical address A1 = EA[LSB]. Table 7 shows the shifting that is required by the logical byte addresses for the various widths available on the C6000 devices.

For example, if a user wished to write the value of 0x555 onto the flash, the width of the interface must always be considered. The value will always be stored into a byte address, so the number of byte-enable bits will increase along with the width of the interface. For an 8-bit interface, the value's least significant bit, A0, will match with the first address bit on the EA line. If the interface is changed to 16-bits, then from Table 7 it can be seen that A1 is the first logical bit to be aligned with the EA line, because A0 is viewed as the byte enable bit. Since the entire 0x555 value is required, one left shift must be manually added in order for the logical byte address to include the least significant bit of the value. The same procedure is required for a 32- and 64-bit interface, with a left shift of 2 and 3 required, respectively.

This example is modeled below with a few lines of simple code. The code represents the value 0x555 transferred from the EMIF to a register on the flash memory. As stated above, the value must be shifted N times depending on the width of the interface.

```
MVKL  0x555,  A5
MVKH  0X555,  A5
SHL A5, N, A5
STW val_reg, *A5
```

Table 7. Byte Address to EA Mapping for Asynchronous Memory Widths

	EA Line																			
C620x/C621x/ C670x/C671x	EA 21	EA 20	EA 19	EA 18	EA 17	EA 16	EA 15	EA 14	EA 13	EA 12	EA 11	EA 10	EA 9	EA 8	EA 7	EA 6	EA 5	EA 4	EA 3	EA 2
C64x EMIFA	EA 22	EA 21	EA 20	EA 19	EA 18	EA 17	EA 16	EA 15	EA 14	EA 13	EA 12	EA 11	EA 10	EA 9	EA 8	EA 7	EA 6	EA 5	EA 4	EA 3
C64x EMIFB	EA 20	EA 19	EA 18	EA 17	EA 16	EA 15	EA 14	EA 13	EA 12	EA 11	EA 10	EA 9	EA 8	EA 7	EA 6	EA 5	EA 4	EA 3	EA 2	EA 1
Mtype Width	**Logical Byte Address**																			
x64	A22	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3
x32	A21	A20	A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2
x16	A20	A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1
x8	A19	A18	A17	A16	A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0

Table 8 shows the values of the opcode, and then shows its logical byte address. For all the interfaces other than the 8-bit-wide interface, a shift is required, as can be seen in the code segment above. This shift is required for the value of 0x555 to be correctly represented on the external address pins.

Table 8. Example of Write to Flash Memory and Address Shifting

	Interface Width	Desired Opcode	N Value	Logical Address†	External Address	Assembly Command‡
C620x/C670x	x32	0x555	2	0x1554	EA[21:2] = 0x555	STW
C621x/C671x	x8	0x555	0	0x555	EA[21:2] = 0x555	STB
	x16	0x555	1	0xAAA	EA[21:2] = 0x555	STH
	X32	0x555	2	0x1554	EA[21:2] = 0x555	STB
C64x EMIFA	x8	0x555	0	A0x555	EA[22:3] = 0x555	STB
	x16	0x555	1	0xAAA	EA[22:3] = 0x555	STH
	X32	0x555	2	0x1554	EA[22:3] = 0x555	STB
	x64	0x555	3	0x2AA8	EA[22:3] = 0x555	STDW

† Logical Address = Opcode left shifted by N

‡ This command will replace the "STW" command in the code for the example.

Table 8. Example of Write to Flash Memory and Address Shifting (Continued)

	Interface Width	Desired Opcode	N Value	Logical Address†	External Address	Assembly Command‡
C64x EMIFB	x8	0x555	0	0x555	EA[20:1] = 0x555	STB
	x16	0x555	1	0xAAA	EA[20:1] = 0x555	STH

† Logical Address = Opcode left shifted by N
‡ This command will replace the "STW" command in the code for the example.

2.4 Programmable ASRAM Parameters

The EMIF allows a high degree of programmability for shaping asynchronous accesses. The programmable parameters that allow this are:

- Read setup/write setup: The time between the beginning of a memory cycle ($\overline{\text{CE}}$ low, address valid) and the activation of the read or write strobe
- Read strobe/write strobe: The time between the activation and deactivation of the read ($\overline{\text{ARE}}$) or write strobe ($\overline{\text{AWE}}$)
- Read hold/write hold: The time between the deactivation of the read or write strobe and the end of the cycle (which may be either an address change or the deactivation of the $\overline{\text{AOE}}$ signal)
- Turnaround (C621x/C671x/C64x only): The time between the end of a read access and the beginning of a write access

These parameters are programmable in terms of clock cycles via fields in the EMIF CE space control registers. For the C620x/C670x, these fields are programmable in terms of CLKOUT1 cycles. For the C621x/C671x/C64x, these fields are programmable in terms of ECLKOUT (or ECLKOUT1) cycles. Separate setup, strobe, and hold parameters are available for read and write accesses. The SETUP, HOLD, and STROBE fields represent actual cycle counts, in contrast to SDRAM parameters, which are the cycle counts – 1.

2.5 Margin Considerations

The flash interface is typically a low-performance interface compared to synchronous memory interfaces, high-speed asynchronous memory interfaces, and high-speed FIFO interfaces. For this reason, this application report pays little attention to minimizing the amount of margin required when programming the asynchronous timing parameters. The approach used requires approximately 10 ns of margin on all parameters, which is not significant for a 100-ns read or write cycle. See Table 9 through Table 13. For additional details on minimizing the amount of margin, see the *TMS320C6000 EMIF to External Asynchronous SRAM Interface* (SPRA542).

Table 9. Recommended Timing Margin

Timing Parameter	Recommended Margin
Output setup	~10 ns
Output hold	~10 ns
Input setup	~10 ns
Input hold	~10 ns

Table 10. EMIF – Input Timing Requirements (Input Data)

Timing Parameter	Definition
t_{isu}	Data setup time, read D before CLKOUT1 high
t_h	Data hold time, read D after CLKOUT1 high

Table 11. EMIF – Output Timing Characteristics (Data, Address, Control)

Timing Parameter	Definition
t_d	Output delay time, CLKOUT1 high to output signal valid

Table 12. ASRAM – Input Timing Requirement

Timing Parameter	Definition
$t_{xw(m)}$	Time from control/data signals active to $\overline{\text{AWE}}$ inactive
$t_{wp(m)}$	Write pulse width
$t_{ih(m)}$, $t_{wr(m)}$	Maximum of either write recovery time or data hold time
$t_{rc(m)}$	Length of the read cycle
$t_{wc(m)}$	Length of the write cycle

Table 13. ASRAM – Output Timing Characteristics

Timing Parameter	Definition
$t_{acc(m)}$	Access time, from EA, $\overline{\text{BE}}$, $\overline{\text{AOE}}$, $\overline{\text{CE}}$ active to ED valid
$t_{oh(m)}$	Output hold time

2.6 Asynchronous Reads

Figure 11 illustrates an asynchronous read cycle with a setup/strobe/hold timing of 1/2/1. An asynchronous read proceeds as follows:

- At the beginning of the setup period
  - $\overline{\text{CE}}$ becomes active low.
  - $\overline{\text{AOE}}$ becomes active low.
  - $\overline{\text{BE}}$[3:0] becomes valid.
  - EA becomes valid.
  - C620x/C670x: For the first access, setup has a minimum value of 2; after the first access, setup has a minimum value of 1 (see Figure 11).
  - C6211/C6711/C64x: Setup is always a minimum of 1 (see Figure 12).

- At the beginning of a strobe period
  - $\overline{\text{ARE}}$ becomes active low.
- At the beginning of a hold period
  - $\overline{\text{ARE}}$ becomes inactive high.
  - Data is sampled on the clock rising edge concurrent with the beginning of the hold period (end of the strobe period) just prior to the $\overline{\text{ARE}}$ low-to-high transition.
- At the end of the hold period
  - $\overline{\text{AOE}}$ becomes inactive as long as another read access to the same $\overline{\text{CE}}$ space is not scheduled for the next cycle.
  - C620x/C670x: After the last access (burst transfer or single access), CE stays active for seven minus the value of read-hold cycles. For example, if READ HOLD = 1, CE stays active for six more cycles. This does not affect performance, but merely reflects the EMIF overhead (see Figure 11).
  - C6211/C6711/C64x: CE goes inactive at the end of the hold period (see Figure 12).

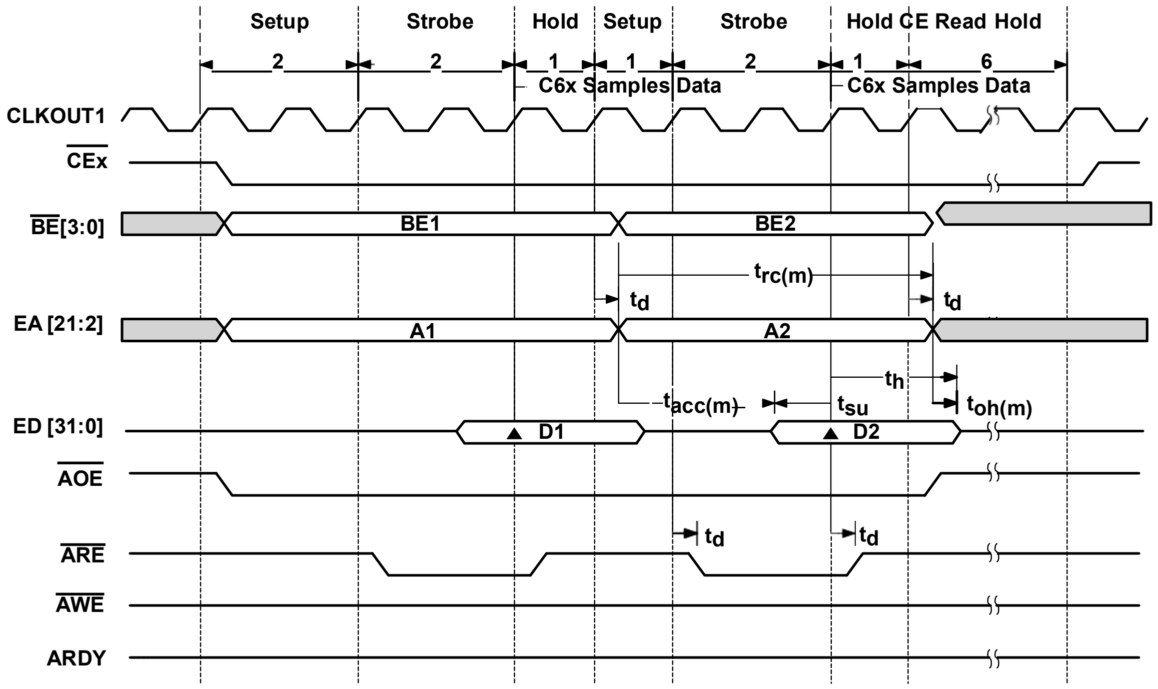


Figure 11. C620x/C670x Asynchronous Read Timing Example (1/2/1)

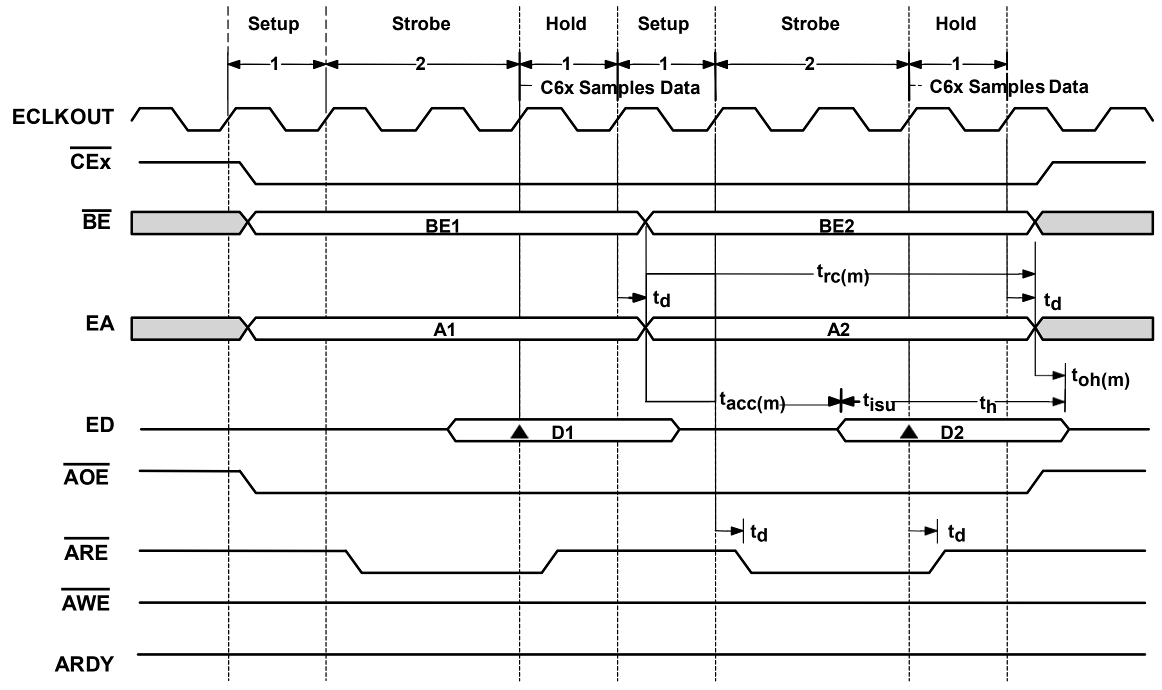


Figure 12. C6211/C6711/C64x Asynchronous Read Timing Example (1/2/1)

2.6.1 Setting Read Parameters for a Specific Flash Memory

Notice in Figure 11 and Figure 12 that the actual timing used by the C6000 to determine when read data is valid is based on the $\overline{\text{ARE}}$ signal. Data is actually read on the rising clock edge corresponding to the cycle prior to which $\overline{\text{ARE}}$ goes high, which is the end of the STROBE period. However, shows that $\overline{\text{ARE}}$ is not connected to asynchronous SRAM. This is pointed out to stress the significance of the SETUP, STROBE, and HOLD times for the C6000 and, to compare them to the significant timing parameters of actual ASRAM.

Flash is not synchronized to any clock; however, it does have a maximum access time (t_{acc}) that relates when the output data is valid after receiving the required inputs. Thus, the data should be sampled at a time t_{acc} plus t_{isu} after the inputs are valid, which, as mentioned, should correspond to the end of the strobe period.

Therefore, when defining the parameters for the C6000 for SETUP, STROBE, and HOLD, the following constraints apply:

- SETUP + STROBE $\geq (t_{acc(m)} + t_{su} + t_{dmax})/t_{cyc}$
- SETUP + STROBE + HOLD $\geq (t_{rc(m)})/t_{cyc}$
- HOLD $\geq (t_h - t_{dmin} - t_{oh(m)})/t_{cyc}$

Normally, SETUP can be set to 1 cycle, then STROBE can be solved for using constraint (1). HOLD can then be solved for using constraint (2). Of course, the smallest value possible should be used for all three parameters to satisfy the constraints while giving the necessary timing margin.

2.7 Asynchronous Writes

Figure 13 illustrates back-to-back asynchronous write with a setup/strobe/hold of 1/1/1. An asynchronous write proceeds as follows.

- At the beginning of the setup period
  - $\overline{\text{CE}}$ becomes active-low.
  - $\overline{\text{BE}}$[3:0] becomes valid.
  - EA becomes valid.
  - ED becomes valid.
  - C620x/C670x: For the first access, setup has a minimum value of 2; after the first access setup has a minimum value of 1 (see Figure 13).
  - C6211/C6711/C64x: Setup is always a minimum of 1 (see Figure 14).
- At the beginning of a strobe period
  - $\overline{\text{AWE}}$ becomes active-low.
- At the beginning of a hold period
  - $\overline{\text{AWE}}$ becomes inactive-high.
- At the end of the hold period
  - ED goes into the high-impedance state only if another write to the same $\overline{\text{CE}}$ space is not scheduled for the next cycle.
  - C620x/C670x: If no write accesses are scheduled for the next cycle and write HOLD is set to 1 or greater, CE will stay active for three cycles after the programmed HOLD period. If write HOLD is set to 0, $\overline{\text{CE}}$ will stay active for four more cycles. This does not affect performance, but merely reflects the EMIF overhead (see Figure 13).
  - C6211/C6711/C64x: CE goes inactive at the end of the hold period (see Figure 14).

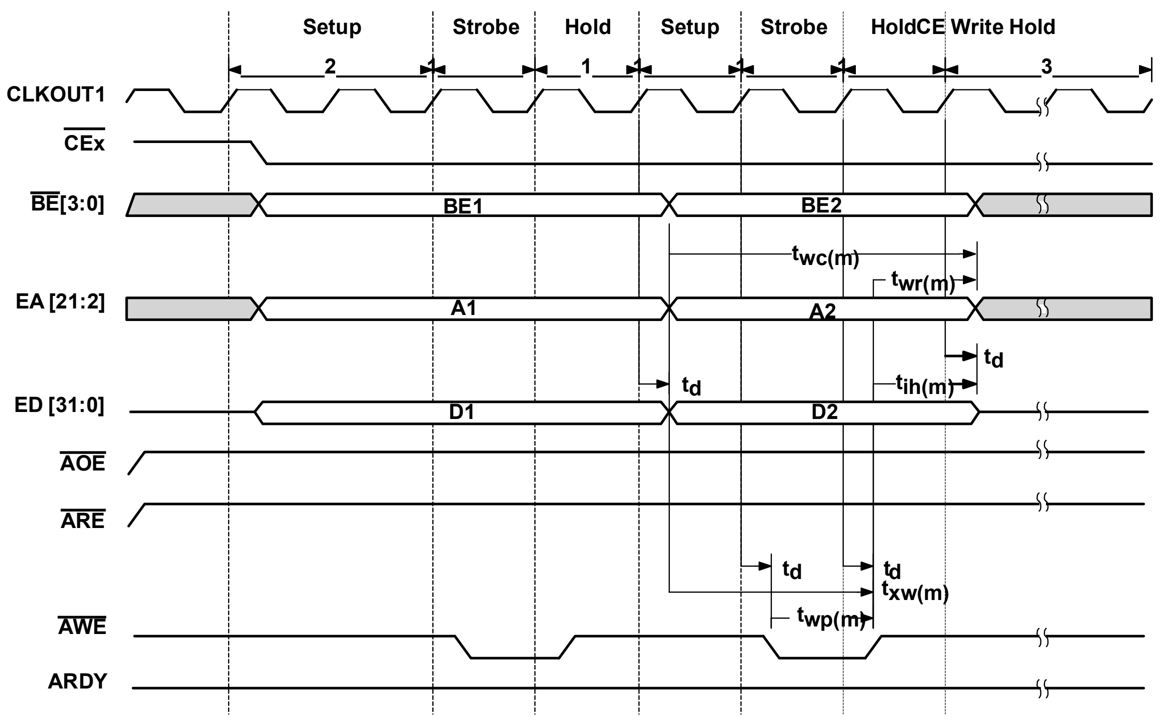


Figure 13. C620x/C670x Asynchronous Write Timing Example (1/1/1)

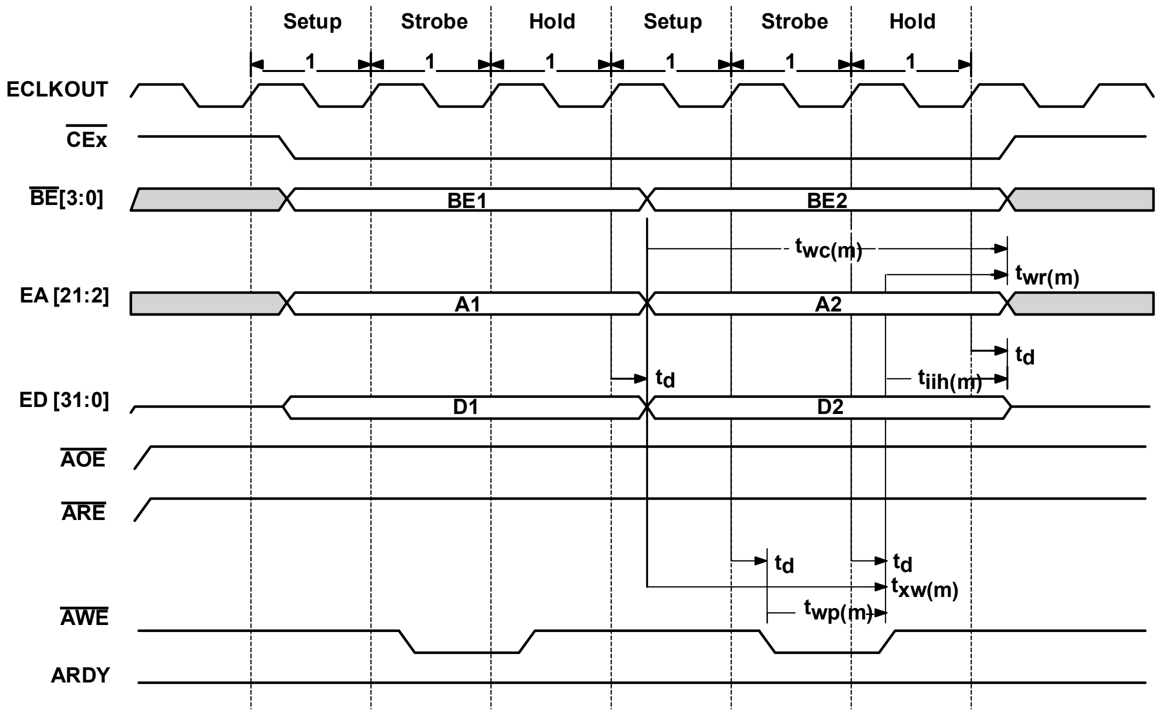


Figure 14. C6211/C6711/C64x Asynchronous Write Timing Example (1/1/1)

2.7.1 Setting Write Parameters for a Specific Asynchronous SRAM

For an ASRAM write, the SETUP, STROBE, and HOLD parameters should be set according to the following constraints:

- STROBE $\geq (t_{wp(m)})/t_{cyc}$.
- SETUP + STROBE $\geq (t_{xw(m)}) /t_{cyc}$
- HOLD $\geq (\max(t_{ih(m)}, t_{wr(m)}))/t_{cyc}$
- SETUP + STROBE + HOLD $\geq (t_{wc(m)})/t_{cyc}$

2.8 Read-to-Write Timing for C6211/C6711/C64x

The C6211/C6711/C64x EMIF offers an additional parameter, TA, that defines the turnaround time between read and write cycles (see Figure 15). This parameter protects against the situation in which the output turn-off time of the memory is longer than the time it takes to start the next write cycle. If this were the case, the C6211/C6711/C64x could drive data at the same time as the memory, causing contention on the bus.

The fact that the C620x/C670x asynchronous interface does not have this feature does not cause problems because the read cycle of these devices append a CE hold period that protects against bus contention.

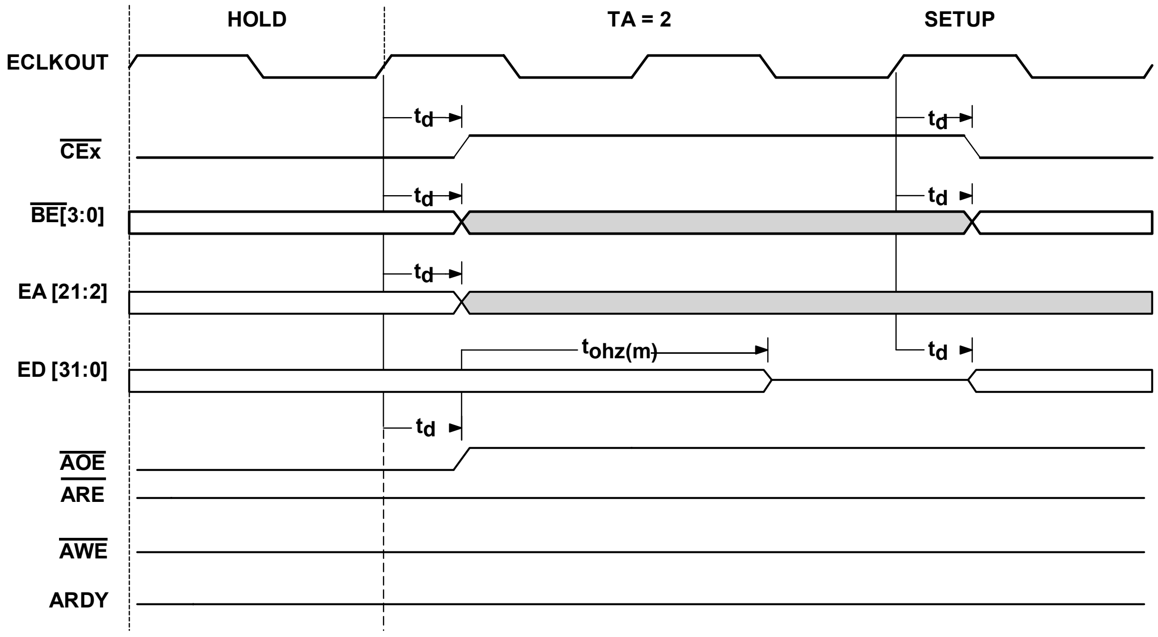


Figure 15. Turnaround Time on C6211/C6711/C64x

2.8.1 Setting TA Parameters for a Specific Asynchronous SRAM

The turnaround time on the C6211/C6711/C64x should be set as follows:

TA >= ($t_{ohz(m)}$)/t_{cyc}

2.8.2 MTYPE Setting for the C620x/C670x

The MTYPE setting can specify different interfaces to asynchronous memory: 8-bit ROM, 16-bit ROM, or 32-bit asynchronous. How the flash is used dictates which of these modes should be selected.

During normal read operation, the MTYPE field should be set to either 8-bit ROM or 16-bit ROM, depending on the specific flash used. When either ROM mode is selected, the EMIF automatically shifts the address appropriately so that the correct data is accessed.

During flash programming, it is simpler to use the 32-bit interface and treat the FLASH as if it were 32 bits wide.

2.8.3 MTYPE Setting for the C6211/C6711/C64x

The MType setting on the C6211/C6711/C64x can select between the 8-, 16-, 32-, or 64-bit (C64x only) asynchronous interface, which is used for both read and write access. For read access, the appropriate bus width should be chosen in the MType field to match the width of the external bus. In this way, the user can do 8-, 16-, and 32-bit accesses and the EMIF takes care of the necessary byte packing.

If an 8- or 16-bit interface is used, 32-bit or 64-bit accesses should not be done because a specific sequence should be used to program the flash and the external address sequence will change, depending on the byte re-ordering necessary. If a 32-bit or 64-bit interface is selected with the MType field, the address can be shifted internally to ensure that the proper sequence is used.

During flash programming, the Mtype field should be programmed according to the width of the memory for reads and writes.

3 Full Example for Programming AMD's AM29LV800-90 (ARDY Interface)

This section walks through the configuration steps required to implement AMD's AM29LV800-90, which is an 8M-bit part organized as 1M x 8 bit or 512K x 16 bit, depending on the state of the mode-select pin.

For this implementation, the following assumptions are made:

- Flash is used in address space CE1, configured as a 16-bit-wide ROM.
- Clock speed is 200 MHz; therefore t_{period} is 5 ns.
- The interface utilizes the RY/$\overline{\text{BY}}$ function of the flash memory; therefore, no software polling is needed.

3.1 Hardware Interface

The hardware interface with the AM29LV800 flash memory is identical to that shown in with the CE1 signal used. As shown in , the CE1 output is logically ORed with the RY/$\overline{\text{BY}}$ signal from the flash memory before being tied to the ARDY input of the EMIF. This is done to prevent this signal from interfering with access to any other asynchronous memory in other CE spaces.

For example, if a program sequence is written to the flash, the only way the EMIF can recognize that the RY/$\overline{\text{BY}}$ signal is low is with an access to CE1. This allows a low signal to pass on to ARDY and extends the cycle until the flash is finished with the program operation. If after the program sequence CE2 is accessed instead, the input to ARDY will be high and not inhibit access to CE2 because CE1 is still high.

The reason CE1 is selected for the memory space in this example is because of the boot processes supported by the C6000. The C6000 can be set up to transfer data from CE1 to address 0 with the DMA immediately after reset, which is a very good use for flash memory. If flash is used at CE1, semi-permanent boot code can be stored there. The state of the Bootmode pins at reset lets the processor know which type of memory is located at CE1.

3.2 Register Configuration

Table 14 through Table 17 summarize the timing characteristics of TI's C6201B and AMD's AM29LV800-90, which are used to calculate the values for the CE0 space configuration register. This data was taken from *TMS320C6201, TMS320C6201B FDigital Signal Processors* (SPRS051) and the AM29LV800 data sheet.

Table 14. C6201B EMIF – Input Requirements

Timing Parameter	Definition	Min	Max	Unit
t_{su}	Setup time, read ED before CLKOUT1 high	4		ns
t_h	Data hold time, read D after CLKOUT1 high	0.8		ns
t_{su}	Setup time, ARDY valid before CLKOUT1 high	3		ns
t_h	Data hold time, ARDY valid after CLKOUT1 high	1.8		ns

Table 15. C6201B EMIF – Output Timing Characteristics

Timing Parameter	Definition	Min	Max	Unit
t_d	Output delay time, CLKOUT1 high to output signal valid	–0.2	4	ns

Table 16. ASRAM Input Requirements From EMIF for AM29LV800-90

Timing Parameter	Definition	Min	Max	Unit
$t_{xw(m)}$	Time from control/data signals active to AWE inactive	45		ns
$t_{wp(m)}$	Write pulse width	35		ns
$t_{ih(m)}$, $t_{wr(m)}$	Maximum of either write recovery time or data hold time	10		ns
$t_{rc(m)}$	Length of the read cycle	90		ns
$t_{wc(m)}$	Length of the write cycle	90		ns

Table 17. ASRAM – Output Timing Characteristics for AM29LV800-90

Timing Parameter	Definition	Min	Max	Unit
$t_{acc(m)}$	Access time from EA, $\overline{\text{BE}}$, $\overline{\text{AOE}}$, CE active to ED valid		90	ns
$t_{oh(m)}$	Output hold time	0		

3.2.1 *Read Calculations*

- SETUP = 1, based on the suggestion stated in section 2.6.1, *Setting Read Parameters for a Specific Flash Memory*.
- SETUP + STROBE $\geq (t_{acc(m)} + t_{su} + t_{dmax})t_{cyc}$

  Therefore,

  $$\begin{aligned}\text{STROBE} &\geq (t_{acc(m)} + t_{su} + t_{dmax})t_{cyc} - \text{SETUP}\\ &\geq (90\text{ ns} + 4\text{ ns} + 4\text{ ns})\ 5\text{ ns} - 1\\ &\geq 19.6\text{ cycles} - 1\text{ cycle} = 18.6\text{ cycles}\end{aligned}$$

  STROBE = 21 cycles; t_{margin} = 12 ns
- SETUP + STROBE + HOLD $\geq (t_{rc(m)})t_{cyc}$

  Therefore,

  $$\begin{aligned}\text{HOLD} &\geq (t_{rc(m)})t_{cyc} - \text{SETUP} - \text{STROBE}\\ &\geq (90\text{ ns})5\text{ ns} - 1 - 21 = -4\text{ cycles}\end{aligned}$$

  HOLD = 0 cycles because it cannot be negative; t_{margin} = 20 ns
- HOLD $\geq (t_h - t_{dmin} - t_{oh(m)})t_{cyc}$

  Therefore,

  $$\begin{aligned}\text{HOLD} &\geq (t_h - t_{dmin} - t_{oh(m)})t_{cyc}\\ &\geq (0.8\text{ ns} - (-0.2\text{ ns}) - 0\text{ ns})5\text{ ns} = 0.2\text{ cycles}\end{aligned}$$

  HOLD = 3 cycles; t_{margin} = 14 ns

The margin recommended is met with the settings specified in bold.

3.2.2 Write Calculations

- STROBE $\geq (t_{wp(m)})/t_{cyc}$
  $\geq (35\text{ ns})/5\text{ ns} = 7$ cycles

  STROBE = 9 cycles; $t_{margin} = 10$ ns
- SETUP + STROBE $\geq (t_{xw(m)})/t_{cyc}$

  Therefore,

  SETUP $\geq (t_{xw(m)})/t_{cyc}$ – STROBE
  $= (45\text{ ns})/5\text{ ns} - 9$ cycles
  $= 0.0$ cycles

  SETUP = 2 cycle; $t_{margin} = 10$ ns
- HOLD $\geq (\max(t_{ih(m)}, t_{wr(m)}))/t_{cyc}$
  $\geq (10\text{ ns})/5\text{ ns} = 2$ cycles

  HOLD = 3 cycles; $t_{margin} = 5$ ns
- SETUP + HOLD + STROBE $\geq t_{wc}$

  SETUP + STROBE + HOLD = 15 cycles = 75 ns. This requirement not is satisfied.

  If **STROBE = 14 cycles**, the sum of the three parameters is 20 cycles or 100 ns, which is greater than 90 ns; $t_{margin} = 10$ ns.

3.2.2.1 MType Setting

Because Figure 16 illustrates flash programming with the C6201B, the MTYPE field is set for the 32-bit asynchronous interface. This allows us to treat the flash as a 32-bit-wide device and to increment the program destination pointer by 4 bytes so that no address shifting is necessary to program the proper address. The control addresses still must be shifted so that the proper address shows up on the address lines. This is illustrated in the sample code in Appendix A.

Using the above calculations, the CE space control register can now be properly configured. Figure 16 shows the CE1 space control register with the properly assigned values for each field. The value to be used is 0x82811220.

31 28	27 22	21 20	19 16
WRITE SETUP	WRITE STROBE	WRITE HOLD	READ SETUP
0010	001110	11	0001

15 14	13 8	7	6 4	3 2	1 0
rsv	READ STROBE	Rsv	MTYPE	Reserved	READ HOLD
11	010101	0	010	00	11

Figure 16. EMIF CE1 Space Control Register Diagram for AM29LV800-90

3.3 Software Control

Software polling is not necessary because this interface utilizes the RY/$\overline{BY}$ signal as an input to the EMIF via the ARDY input, and any read or write to the flash is automatically extended until the flash is ready to respond. Because software polling is not necessary, the software algorithm is simple.

3.3.1 Read Operation

For this flash device, the device is in read mode automatically on hardware reset; thus, no special steps must be taken if the device is to be used only for reading code or data. At the first convenient time, the EMIF registers should be set as described in section 3.2.

3.3.2 Write Operation

The write operation is slightly more involved but is still simple because software polling is not required. Figure 17 contains flowcharts of the two operations required to write new data to the flash memory. This flowchart and code example assumes that the source data has been written to CE2 via the HPI or by other means not discussed in this document. This example illustrates erasing, then programming the flash memory. For the source code used, refer to Appendix A.

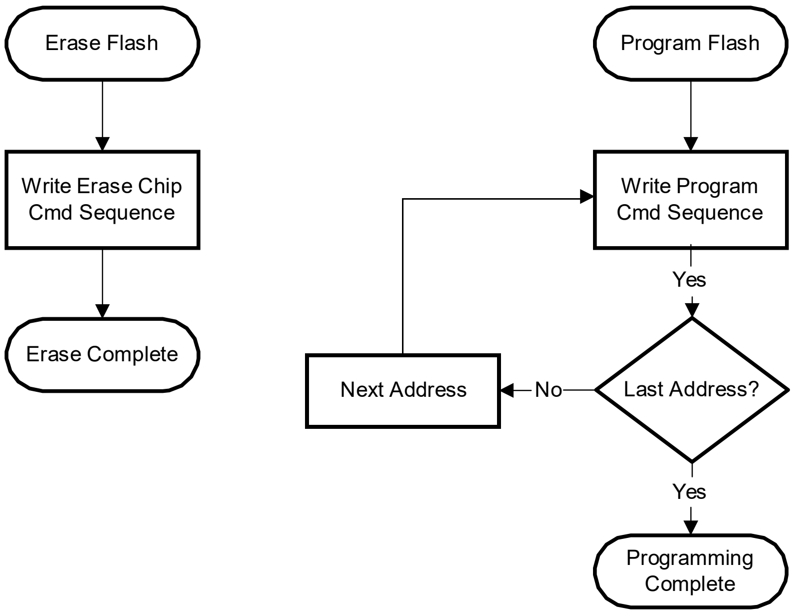


Figure 17. Erase Chip and Program Command Flow Charts

4 Full Example for Programming AMD's AM29LV040-70

This section walks through the configuration steps required to implement TI's AM29LV040-70, which is a 4M-bit part organized as 512K x 8 bit, with the C6201B. See Table 18 through Table 21.

This implementation makes the following assumptions:

- Flash is used in address space CE1, configured as a 16-bit-wide ROM.
- Clock speed is 200 MHz; therefore, t_{period} is 5 ns.
- The interface does **not** utilize the RY/$\overline{\text{BY}}$ function of the flash memory; therefore, software polling is required.

4.1 Hardware Interface

The hardware interface with the AM29LV040 flash memory is identical to that shown in Figure 8, with the CE1 signal used.

The reason CE1 is selected for the memory space in this example is the boot processes supported by the C6201B. The C6201B can be set up to transfer data from CE1 to address 0 with the DMA immediately after reset, which is a very good use for flash memory. If flash is used at CE1, semi-permanent boot code can be stored there. The state of the Bootmode pins at reset lets the processor know which type of memory is located at CE1.

Table 18. C6201B EMIF – Input Requirements

Timing Parameter	Definition	Min	Max	Unit
t_{su}	Setup time, read ED before CLKOUT1 high	4		ns
t_h	Data hold time, read D after CLKOUT1 high	0.8		ns
t_{su}	Setup time, ARDY valid before CLKOUT1 high	3		ns
t_h	Data hold time, ARDY valid after CLKOUT1 high	1.8		ns

Table 19. C6201B EMIF – Output Timing Characteristics

Timing Parameter	Definition	Min	Max	Unit
t_d	Output delay time, CLKOUT1 high to output signal valid	–0.2	4	ns

Table 20. Input Requirements for AM29LV040-70

Timing Parameter	Definition	Min	Max	Unit
$t_{xw(m)}$	Time from control/data signals active to AWE inactive	45		ns
$t_{wp(m)}$	Write pulse width	35		ns
$t_{ih(m)}$, $t_{wr(m)}$	Maximum of either write recovery time or data hold time	10		ns
$t_{rc(m)}$	Length of the read cycle	70		ns
$t_{wc(m)}$	Length of the write cycle	70		ns

Table 21. Output Timing Characteristics for AM29LV040-70

Timing Parameter	Definition	Min	Max	Unit
$t_{acc(m)}$	Access time from EA, $\overline{\text{BE}}$, $\overline{\text{AOE}}$, CE active to ED valid		70	ns
$t_{oh(m)}$	Output hold time	0		

4.1.1 *Read Calculations*

- SETUP = 1, based on the suggestion stated in the section 2.6.1, *Setting Read Parameters for a Specific Flash Memory.*
- SETUP + STROBE $\geq (t_{acc(m)} + t_{su} + t_{dmax})/t_{cyc}$

  Therefore,

  STROBE $\geq (t_{acc(m)} + t_{su} + t_{dmax})/t_{cyc}$ – SETUP
  $\geq$ (70 ns + 4 ns + 4 ns)/ 5 ns –1
  $\geq$ 15.6 cycles – 1 cycle = 14.6 cycles

  STROBE = 17 cycles; t_{margin} = 12 ns
- SETUP + STROBE + HOLD $\geq (t_{rc(m)})/t_{cyc}$

  Therefore,

  HOLD $\geq (t_{rc(m)})/t_{cyc}$ – SETUP – STROBE
  $\geq$ (70 ns)/5 ns – 1 – 17 = –4 cycles

  HOLD = 0 cycles because it cannot be negative; t_{margin} = 20 ns
- HOLD $\geq (t_h - t_{dmin} - t_{oh(m)})/t_{cyc}$

  Therefore,

  HOLD $\geq (t_h - t_{dmin} - t_{oh(m)})/t_{cyc}$
  $\geq$ (0.8 ns – (–0.2 ns) – 0 ns)/ 5 ns = 0.2 cycles

  HOLD = 3 cycles; t_{margin} = 14 ns

With the settings specified in bold, the margin recommended is met.

4.1.2 Write Calculations

- STROBE $\geq (t_{wp(m)})/t_{cyc}$
  $\geq (35\text{ ns})/\ 5\text{ ns} = 7\text{ cycles}$

  STROBE = 9 cycles; $t_{margin} = 10\text{ ns}$
- SETUP + STROBE $\geq (t_{xw(m)})/t_{cyc}$

  Therefore,

  $$\begin{aligned}\text{SETUP} &\geq (t_{xw(m)})/t_{cyc} - \text{STROBE}\\ &= (45\text{ ns})/5\text{ ns} - 9\text{ cycles}\\ &= 0.0\text{ cycles}\end{aligned}$$

  SETUP = 2 cycle; $t_{margin} = 10\text{ ns}$
- HOLD $\geq (\max(t_{ih(m)}, t_{wr(m)}))/t_{cyc}$
  $\geq (10\text{ ns})/5\text{ ns} = 2\text{ cycles}$

  HOLD = 3 cycles; $t_{margin} = 5\text{ ns}$
- SETUP + HOLD + STROBE $\geq t_{wc}$

  SETUP + STROBE + HOLD = 14 cycles = 70 ns. This requirement does not provide any margin.

  If **STROBE = 11 cycles**, the sum of the three parameters is 16 cycles or 80 ns, which is greater than 70 ns. $t_{margin} = 10\text{ ns}$.

4.1.2.1 MType Setting

Because Figure 18 illustrates flash programming, the MTYPE field is set for a 32-bit asynchronous interface. This allows us to treat the flash as a 32-bit-wide device and to increment the program destination pointer by 4 bytes so that no address shifting must be done to program the proper address. The control addresses must still be shifted so that the proper address shows up on the address lines. This is illustrated in the sample code in Appendix B.

Using the above calculations, the CE space control register can now be properly configured. See Figure 18.

31 – 28	27 – 22	21 – 20	19 – 16
WRITE SETUP	WRITE STROBE	WRITE HOLD	READ SETUP
0010	001011	11	0001

15 – 14	13 – 8	7	6 – 4	3 – 2	1 – 0
rsv	READ STROBE	Rsv	MTYPE	Reserved	READ HOLD
11	010001	0	010	00	11

Figure 18. EMIF CE1 Space Control Register Diagram for AM29LV040-70

4.2 Software Control

Software polling is necessary because this interface does not utilize the RY/$\overline{\text{BY}}$ signal as an input to the EMIF via the ARDY input, and the EMIF does not automatically wait until the embedded algorithm is complete.

4.2.1 Read Operation

On hardware reset, this flash device is automatically in read mode; thus, no special steps must be taken if the device is used only to read code or data. At the first convenient time, the EMIF registers should be set as described in section 3.2.

4.2.2 Write Operation

The write operation is slightly more involved. Software polling must be used to detect completion of the embedded program or erase algorithm. Figure 19 contains flow charts of the two operations required to write new data to the flash memory – erase and program. In each of these algorithms, device polling must be done using software. The polling algorithm is shown in Figure 20. This flow chart and code example assumes that the source data is written to CE2 via the HPI or other means not discussed in this document. This example illustrates erasing, then programming the flash memory. For source code, see Appendix B.

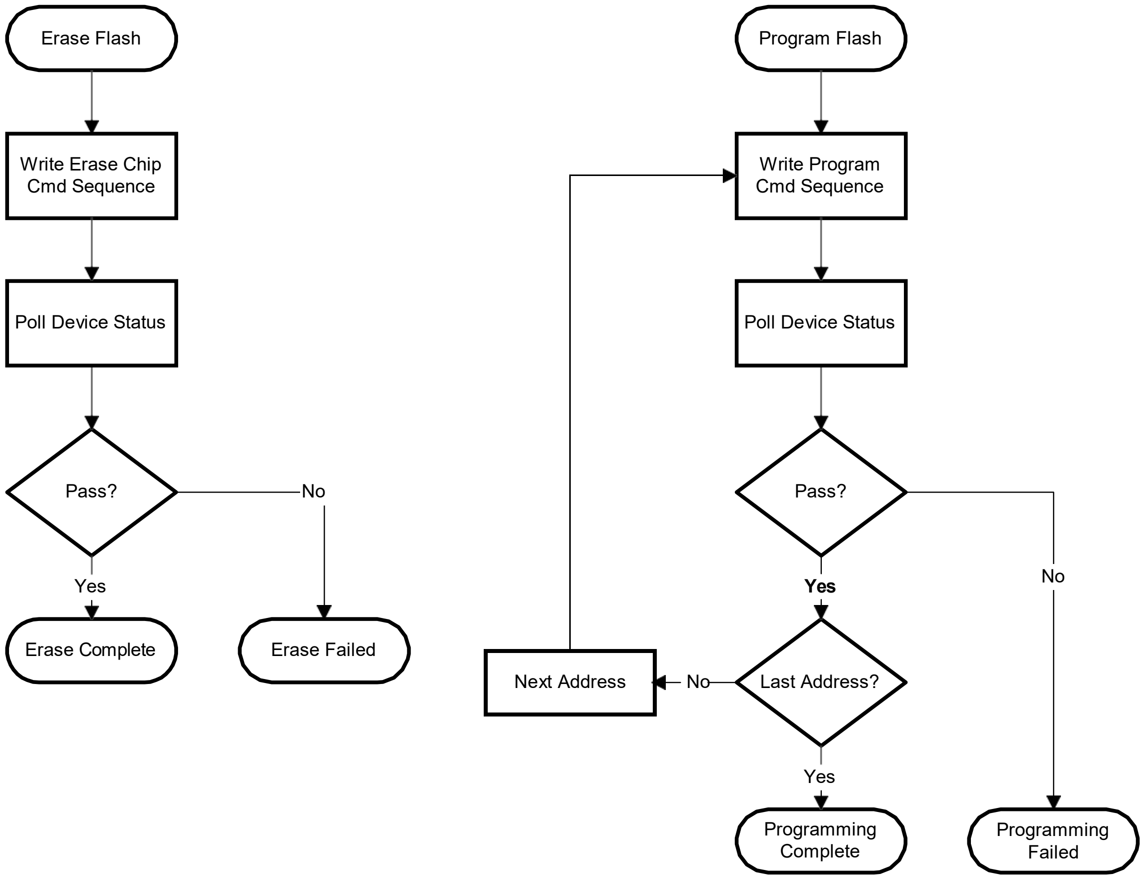


Figure 19. Erase Chip and Program Command Flow Charts

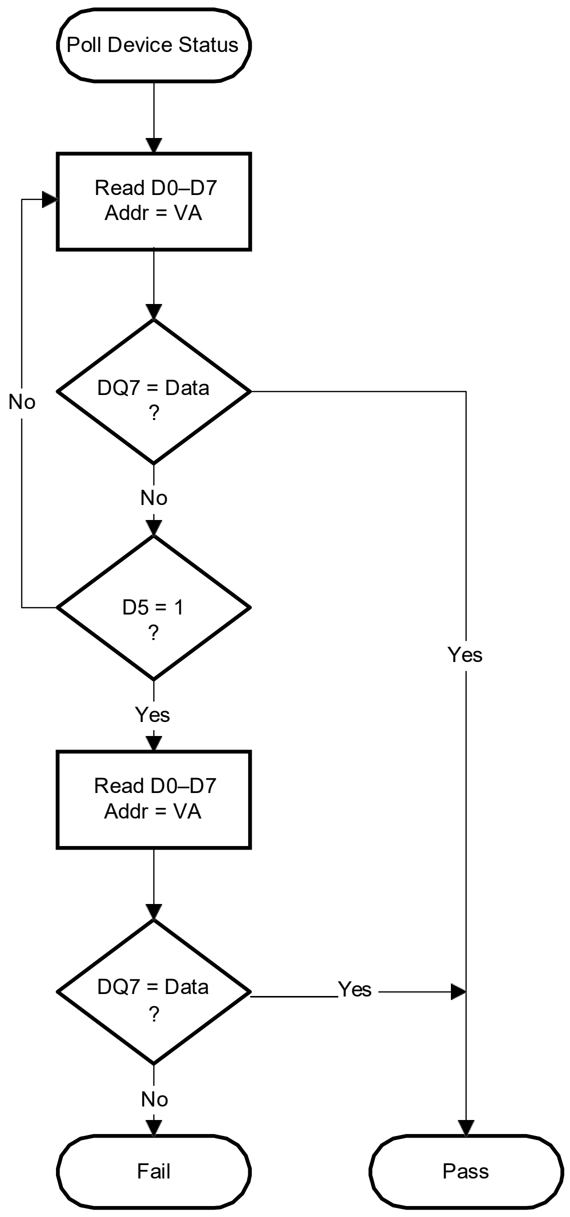


Figure 20. Data Poll Flow Chart

5 References

1. *TMS320C6000 EMIF to External Asynchronous SRAM Interface* (SPRA542).
2. *TMS320C6201Digital Signal Processor* (SPRS051).
3. *TMS320C6202 Fixed-Point Digital Signal Processor* (SPRS072).
4. *TMS320C6211, TMS320C6211B Fixed-Point DSPs* (SPRS073).
5. *TMS320C6701 Floating-Point DSP* (SPRS067).
6. *TMS320C6711, TMS320C6711B Floating-Point DSPs* (SPRS088).
7. *TMS320C6000 Peripherals Reference Guide* (SPRU190).
8. *TMS320C6000 Peripheral Support Library Programmers Reference* (SPRU273).
9. *AM29LV800 Data Sheet*, Advanced Micro Devices.
10. *AM29LV040 Data Sheet*, Advanced Micro Devices.

Appendix A Sample Code for Programming AM29LV800B (ARDY Interface)

```
/****************************************************************************/
/* Hardware.c
/* Written by: Kyle Castille
/* Updated by: Michael Haag (6/27/01)
/* This program will create a dummy data buffer with an incrementing count
/* in internal memory, and then program this data to the AM29LV800-90 Flash
/* which is a 1M x 8/512k x 16, 90 ns Flash memory.
/*
/* This program assumes that the ARDY interface is used, so no software
/* checking is done to detect end of operation for erase or program
/*
/****************************************************************************/
#define  CE1_ADDRS   0x01400000
#define    INT_MEM   0x80000000
#define   CE1_CNTRL  0x01800004
#define  FLASH_ADDRS  CE1_ADDRS
#define  SRC_ADDRS   INT_MEM
#define  LENGTH      0x400
#define  TRUE        1
#define  FALSE       0

#include <csl.h>
#include <csl_emif.h>

void load_source (short * source, int num_words);
void erase_flash(int * flash_addrs);
void program_flash(short * source, int * flash_addrs, int num_words);
void emif_config();
void
main(){
 int * flash_ptr = (int *)FLASH_ADDRS;
 short * src_ptr = (short *)SRC_ADDRS;

 /* initialize the CSL library */
 CSL_init();
```

```
   emif_config();
   load_source(src_ptr, LENGTH);
   erase_flash(flash_ptr);
   program_flash(src_ptr, flash_ptr, LENGTH);
   printf(”Successful erase and program!!!”);
}

/******************************************************************************/
/* emif_config :Routine to configure the Emif for operation with              */
/*    AM29LV800-90 at CE1. This routine sets the CE1 control register         */
/*    for a 32 bit asynchronous memory interface with the following           */
/*    parameters:                                                              */
/*       Mtype = 010                                                           */
/*       Read Setup/Strobe/Hold = 1/21/3                                       */
/*       Write Setup/Strobe/Hold = 2/13/3                                      */
/*                                                                              */
/******************************************************************************/

void emif_config()
{
   /* Create Global Control Register field */
   Uint32 global_ctl = EMIF_GBLCTL_RMK(
         EMIF_GBLCTL_NOHOLD_0,
         EMIF_GBLCTL_SDCEN_DISABLE,
         EMIF_GBLCTL_SSCEN_DISABLE,
         EMIF_GBLCTL_CLK1EN_ENABLE,
         EMIF_GBLCTL_CLK2EN_DISABLE,
         EMIF_GBLCTL_SSCRT_CPUOVR2,
         EMIF_GBLCTL_RBTR8_HPRI);

   /* Create CE1 Control Register field */
   Uint32 ce1_control = EMIF_CECTL_RMK(
         EMIF_CECTL_WRSETUP_OF(2),
         EMIF_CECTL_WRSTRB_OF(14),
         EMIF_CECTL_WRHLD_OF(3),
         EMIF_CECTL_RDSETUP_OF(1),
         EMIF_CECTL_RDSTRB_OF(21),
         EMIF_CECTL_MTYPE_ASYNC32,
         EMIF_CECTL_RDHLD_OF(3) );
   EMIF_configArgs(
      EMIF_GBLCTL_OF(global_ctl),    /* global control     */
      EMIF_CECTL_OF(0x00000018),     /* CE0 control        */
      EMIF_CECTL_OF(ce1_control),    /* 32-bit async mem   */
      EMIF_CECTL_OF(0x00000018),     /* CE2 control        */
      EMIF_CECTL_OF(0x00000018),     /* CE3 control        */
      EMIF_SDCTL_OF(0x0388F000),     /* SDRAM control      */
      EMIF_SDTIM_OF(0x00800040)      /* SDRAM timing       */
            );
}

/*********************************************************************************/
/* load_source :Routine to load the source memory with data. This routine     */
/*    loads an incrementing count into the source memory for                    */
/*    demonstration purposes.                                                    */
/* Inputs:                                                                       */
/*  source_ptr :  Address to be used as the source buffer                        */
/*  length     :  Length to be programmed                                        */
/*                                                                                /
/*********************************************************************************/

void load_source(short * source_ptr, int length)
{
 int i;
 for (i = 0; i < length; i ++){
  * source_ptr++ = i;
 }
}

/*********************************************************************************/
/* erase_flash : Routine to erase entire FLASH memory AM29LV800 (1M x 8bit/      */
/*    512k x 16bit)                                                              */
/* Inputs:                                                                       */
/*    flash_ptr: Address of the FLASH PEROM                                      */
/*                                                                               */
/*********************************************************************************/

void erase_flash(int * flash_ptr)
{
      /* Control addresses are left shifted so that   */
      /* they appear correctly on the EMIF's EA[19:2] */
      /* Byte address << 2 == Word Address      */
   int * ctrl_addr1 = (int *) ((int)flash_ptr + (0x555 << 2));
   int * ctrl_addr2 = (int *) ((int)flash_ptr + (0x2aa << 2));;
   * ctrl_addr1 = 0x00aa;  /* Erase sequence writes to addr1 and addr2  */
   * ctrl_addr2 = 0x0055;  /* with this data     */
   * ctrl_addr1 = 0x0080;
   * ctrl_addr1 = 0x00aa;
   * ctrl_addr2 = 0x0055;
   * ctrl_addr1 = 0x0010;
}
/********************************************************************************/
/* program_flash: Routine to program FLASH AM29LV800                            */
/* Inputs:                                                                      */
/*    flash_ptr   : Address of the FLASH                                       */
/*    source_ptr  : Address of the array containing the code to program        */
/*    length            : Length to be programmed                              */
/*                                                                              */
/********************************************************************************/

void program_flash(short * source_ptr, int * flash_ptr, int length)
{
   int i;
         /* Control addresses are left shifted so that   */
         /* they appear correctly on the EMIF’s EA[19:2] */
         /* Byte address << 2 == Word Address   */
   int * ctrl_addr1 = (int *) ((int)flash_ptr + (0x555 << 2));
   int * ctrl_addr2 = (int *) ((int)flash_ptr + (0x2aa << 2));;

   for (i = 0; i < length; i++){
    * ctrl_addr1 = 0x00aa;
    * ctrl_addr2 = 0x0055;
    * ctrl_addr1 = 0x00a0;
    * flash_ptr++ = * source_ptr++;
 }
}
```

Appendix B Sample Code for Programming AM29LV040

```
/*****************************************************************************/
/* Software.c                                                                */
/* Written by: Kyle Castille                                                 */
/* Updated by: Michael Haag (6/27/01)                                        */
/* This program will create a dummy data buffer with an incrementing count   */
/* in internal memory, and then program this data to the AM29LF040-70 Flash  */
/* which is a 512k x 8, 70 ns Flash memory.                                  */
/* This program assumes that the ARDY interface is NOT used, therefore       */
/* monitoring is done to detect end of operation                             */
/*****************************************************************************/
#define  CE1_ADDRS   0x01400000
#define    INT_MEM   0x80000000
#define   CE1_CNTRL  0x01800004
#define FLASH_ADDRS CE1_ADDRS
#define  SRC_ADDRS   INT_MEM
#define  LENGTH      0x400
#define  TRUE        1
#define  FALSE       0

#include <csl.h>
#include <csl_emif.h>

void load_source (unsigned char * source, int num_words);
int erase_flash(int* flash_addrs);
int program_flash(unsigned char * source, int * flash_addrs, int num_words);
int poll_data(int *,unsigned char );
void emif_config();

void
main(){
 int pass = TRUE;
 int * flash_ptr = (int *)FLASH_ADDRS;
 unsigned char * src_ptr = (unsigned char *)SRC_ADDRS;

 /* initialize the CSL library */
 CSL_init();

 emif_config();
 load_source(src_ptr, LENGTH);
 pass = erase_flash(flash_ptr);
 if (pass){
  pass = program_flash(src_ptr, flash_ptr, LENGTH);
  if (!pass)
   printf(”Failed in program operation”);
  else
   printf(”Successful erase and program!!!”);
  }
  else
   printf(”Failed in erase operation”);
}

/*********************************************************************************/
/* emif_config :Routine to configure the Emif for operation with                 */
/*    AM29LV040-70 at CE1. This routine sets the CE1 control register            */
/*    for a 32 bit asynchronous memory interface with the following              */
/*    parameters:                                                                */
/*       Mtype = 010 (32-bit async)                                             */
/*       Read Setup/Strobe/Hold = 1/17/3                                         */
/*       Write Setup/Strobe/Hold = 2/11/3                                        */
/*                                                                               */
/*********************************************************************************/
void emif_config()
{
 /* Create Global Control Register field */
 Uint32 global_ctl = EMIF_GBLCTL_RMK(
      EMIF_GBLCTL_NOHOLD_0,
      EMIF_GBLCTL_SDCEN_DISABLE,
      EMIF_GBLCTL_SSCEN_DISABLE,
      EMIF_GBLCTL_CLK1EN_ENABLE,
      EMIF_GBLCTL_CLK2EN_DISABLE,
      EMIF_GBLCTL_SSCRT_CPUOVR2,
      EMIF_GBLCTL_RBTR8_HPRI);
 /* Create CE1 Control Register field */
 Uint32 ce1_control = EMIF_CECTL_RMK(
      EMIF_CECTL_WRSETUP_OF(2),
      EMIF_CECTL_WRSTRB_OF(11),
      EMIF_CECTL_WRHLD_OF(3),
      EMIF_CECTL_RDSETUP_OF(1),
      EMIF_CECTL_RDSTRB_OF(17),
      EMIF_CECTL_MTYPE_ASYNC32,
      EMIF_CECTL_RDHLD_OF(3) );

 EMIF_configArgs(
    EMIF_GBLCTL_OF(global_ctl),     /* global control             */
    EMIF_CECTL_OF(0x00000018),      /* CE0 control                */
    EMIF_CECTL_OF(ce1_control),     /* 32-bit async mem           */
    EMIF_CECTL_OF(0x00000018),      /* CE2 control                */
    EMIF_CECTL_OF(0x00000018),      /* CE3 control                */
    EMIF_SDCTL_OF(0x0388F000),      /* SDRAM control              */
    EMIF_SDTIM_OF(0x00800040)       /* SDRAM timing               */
     );
}

/*******************************************************************************/
/* load_source :Routine to load the source memory with data. This routine        */
/*    loads an incrementing count into the source memory for                     */
/*    demonstration purposes.                                                    */
/* Inputs:                                                                       */
/*    source_ptr :   Address to be used as the source buffer                    */
/*    code_ptr   :   Length to be programmed                                    */
/*                                                                               */
/*******************************************************************************/

void load_source(unsigned char * source_ptr, int length)
{
 int i;
 for (i = 0; i < length; i ++){
  * source_ptr++ = i;
 }
}
```

```
/******************************************************************************/
/* erase_flash : Routine to erase entire FLASH memory AM29LV040 (512Kx8bit)      */
/* Inputs:                                                                        */
/* flash_ptr: Address of the FLASH                                                */
/* Return value:                                                                  */
/* Returns TRUE if passed, or FALSE if failed. Pass or failure is                 */
/* determined during the poll_data routine.                                       */
/*                                                                                */
/******************************************************************************/

int erase_flash(int * flash_ptr)
{
      /* Control addresses are left shifted so that   */
      /* they appear correctly on the EMIF's EA[19:2] */
      /* unsigned char << 2 == Word       */
 int * ctrl_addr1 = (int *) ((int)flash_ptr + (0x555 << 2));
 int * ctrl_addr2 = (int *) ((int)flash_ptr + (0x2aa << 2));

 int pass = TRUE;

 * ctrl_addr1 = 0xaa;   /* Erase sequence writes to addr1 and addr2 */
 * ctrl_addr2 = 0x55;   /* with this data       */
 * ctrl_addr1 = 0x80;
 * ctrl_addr1 = 0xaa;
 * ctrl_addr2 = 0x55;
 * ctrl_addr1 = 0x10;

 pass = poll_data(flash_ptr, (unsigned char) 0xff);
 if (!pass)
  printf("failed erase\n\n");
 return pass;
}
```

```
/*****************************************************************************/
/* program_flash: Routine to program FLASH AM29LV040(512K x 8bit)            */
/* Inputs: q                                                                 */
/* flash_ptr: Address of the FLASH PEROM                                     */
/* code_ptr : Address of the array containing the code to program            */
/* Return value:                                                             */
/* Returns TRUE if passed, or FALSE if failed. Pass or failure is            */
/* determined during the poll_data routine.                                  */
/*                                                                           */
/*****************************************************************************/

int program_flash(unsigned char * source_ptr, int * flash_ptr, int length)
{
 int i;
 unsigned char data;
 int pass;
      /* Control addresses are left shifted so that */
      /* they appear correctly on the EMIF's EA[19:2] */
      /* Short << 1 == Word     */
 int * ctrl_addr1 = (int *) ((int)flash_ptr + (0x555 << 2));
 int * ctrl_addr2 = (int *) ((int)flash_ptr + (0x2aa << 2));;

 for (i = 0; i < length; i++){

  * ctrl_addr1 = 0x00aa;
  * ctrl_addr2 = 0x0055;
  * ctrl_addr1 = 0x00a0;

  * flash_ptr++ = data = * source_ptr++;
      pass = poll_data(flash_ptr-1, data);
 }
 if (!pass)
  printf("Failed at address %x \n\n", (int) flash_ptr);
 return pass;
}
```

```
/********************************************************************************/
/* poll_data: Routine to determine if Flash has successfully completed the       */
/* program or erase algorithm. This routine will loop until                      */
/* either the embedded algorithm has successfully completed or                   */
/* until it has failed.                                                          */
/*                                                                               */
/* Inputs:                                                                       */
/* prog_ptr : Address just programmed                                            */
/* prog_data: Data just programmed to flash                                      */
/* Return value:                                                                 */
/* Returns TRUE if passed, or FALSE if failed.                                   */
/*                                                                               */
/********************************************************************************/

int poll_data(int * prog_ptr, unsigned char prog_data)
{
 unsigned char data;
 int fail = FALSE;

 do {
  data = (unsigned char) * prog_ptr;
  if (data != prog_data)            /* is D7 != Data? */
  {
   if ((data & 0x20) == 0x20)       /*is D5 = 1 ?  */
   {
    data = (unsigned char) * prog_ptr;
    if (data != prog_data)          /* is D7 = Data?  */
      fail = TRUE;
    else
      return TRUE;      /* PASS  */
     }
    }
      else
   return TRUE;      /* PASS  */
 } while (!fail);
 return FALSE;       /* FAIL  */
}
```