Using Memory in ispXPLD™ 5000MX Devices

March 2005

Technical Note TN1030

Introduction

This document describes memory usage and flow in the Lattice ispXPLD™ family of devices. A brief overview of the ispXPLD's memory resources are presented along with the parameterizable memory elements supported by Lattice's ispLEVER™ design tool.

The ispXPLD architecture is built around the Multifunction Block (MFB), which can be configured either as traditional logic or as memory. When it is configured as memory it can function as dual-port SRAM, pseudo-dual port SRAM, single-port SRAM, FIFO, or CAM memory.

Multifunction Blocks

The ispXPLD architecture allows the MFB to be configured as a variety of memory blocks as detailed in Table 1.

Table 1. MFB Memory Configurations

Memory Mode	Configurations
Dual-port	8,192 x 1 4,096 x 2 2,048 x 4 1,024 x 8 512 x 16
Single-port, Pseudo Dual Port, FIFO	16,384 x1 8,192 x 2 4,096 x 4 2048 x 8 1024 x 16 512 x 32
CAM	128 x 48

Once the MFB has been configured as memory, no other logic can be implemented in that block. The one exception is a FIFO block that requires 32 data outputs, as it will need an additional MFB for flag generation. To generate the control circuitry for the four FIFO flags, four macrocells and six inputs are required. This leaves 28 macrocells and 62 logic inputs for generic logic implementation in the additional MFB.

Initializing Memory

In each of the memory modes it is possible to specify the power-on state of each bit in the memory array. This allows the memory to be used as ROM if desired. Each bit in the memory array can have one of four values: 0, 1, X (always match) or U (never match). Note that X (always match) and U (never match) values only apply for CAM. For all other memory modes, use ones and zeroes.

Increased Depth And Width

Memory that requires a depth or width that is greater than that supported by a single MFB, can be supported by cascading multiple blocks. For dual port, single port, and pseudo-dual port memory blocks, additional width is easily provided by sharing address lines. Additional depth is supported, by multiplexing the RAM output. For FIFO and CAM modes additional width is supported through the cascading of MFBs. For FIFOs up to four MFBs can be cascaded for additional width ranging to 128 bits. The CAM can also cascade up to four MFBs to provide additional width, allowing the implementation of a CAM to a maximum of 128x192.

Lattice's ispLEVER design tool automatically combines blocks to support the memory size specified in the user's design.

Bus Size Matching

All of the memory modes apart from the CAM mode support different widths on each of the ports. The RAM bits are mapped LSB word 0 to MSB word 0, LSB word 1 to MSB word 1 and so on. Although the word size and number of words for each port varies this mapping scheme applies to each port.

With CAM memory the port size is 48 bits for a single MFB. Cascading up to three additional MFBs can expand the width of the CAM to 192 bits. Use of the mask register allows for comparisons of less than 48 bits but the port width will still be 48 bits. This is more fully explained in the CAM description later in this document.

Different Data Bus Widths on Two Ports

True Dual Port and Pseudo Dual Port modes support different data bus widths. The two ports in the memory can have different data bus widths. In True Dual Port Mode different widths for read/write port A and read/write port B is supported. In Pseudo Dual Port mode, different widths for the read and write ports can also be specified.

While the two ports are operating with different data bus widths, the addressing scheme ensures that the RAM location addressed with each address follow a certain order. Each word written on a wider side can be read as successive multiple words on a narrower port. For example if port A writes 32-bit words, and port B reads 8-bit words, one word written on port A is read as four consecutive words from port B.

Supported Memory Modes

True Dual-Port SRAM Mode

In Dual-Port SRAM mode, the Read/Write address lines share two independent read/write ports, and can access up to 8,192-bits of memory. Data widths of 1, 2, 4, 8, and 16 are supported by the MFB. Figure 1 shows the block diagram of the dual port SRAM.

Write data, address, chip select and read/write signals are always synchronous (registered). The output data signals can be synchronous or asynchronous. Resets are asynchronous. All inputs on the same port share the same clock, clock enable, and reset selections. All outputs on the same port share the same clock, clock enable, and reset selections. Port A and Port B are completely independent from a data width and from a control standpoint. There may be instances when both data ports attempt to write to the memory. This should be avoided as there is no arbitration logic to control which port controls the writing of data into the memory. Table 2 shows the possible sources for the clock, clock enable and initialization signals for the various registers.

Figure 1. Dual-Port SRAM Block Diagram

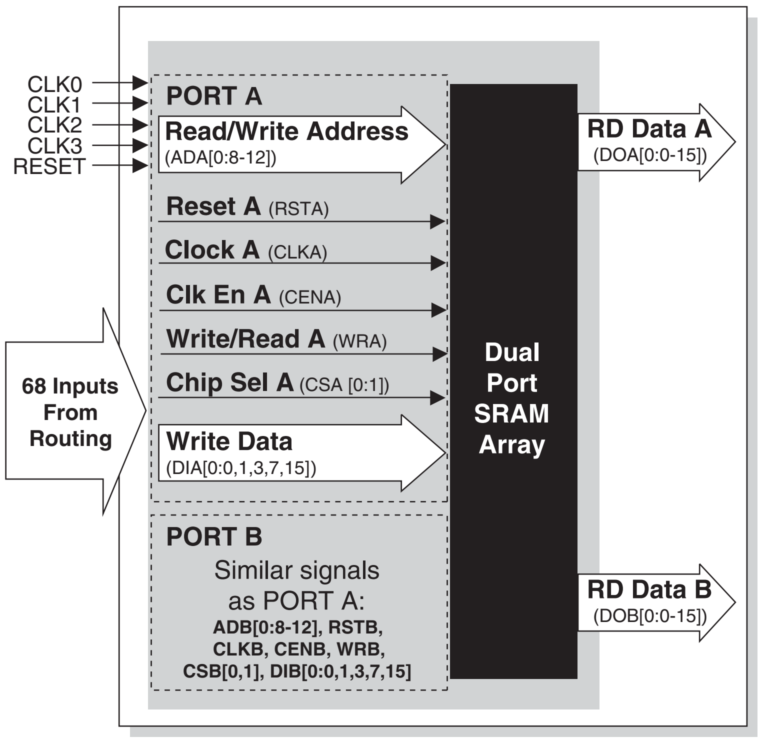


Table 2. Register Clock, Clock Enable and Reset in Dual-Port SRAM Mode

Register	Input	Source
Address, Write Data, Read Data, Read/Write, and Chip Select	Clock	Selected from CLKA (CLKB) or one of the global clocks (CLK0 - CLK3). The selected signal can be inverted if desired.
	Clock Enable	Selected from CENA (CENB) or two of the global clocks (CLK1 - CLK 2). The selected signal can be inverted if required.
	Reset	Created by the logical OR of the global reset signal and RSTA (RSTB). RSTA (RSTB) can be inverted if desired.

Pseudo Dual-Port SRAM Mode

In Pseudo Dual-Port SRAM mode the MFB is configured as an SRAM memory with independent read and write ports that access the same 16,384-bits of memory. Data widths of 1, 2, 4, 8, 16 and 32 are supported by the MFB. Figure 2 shows the block diagram of the Pseudo Dual-Port SRAM

Write data, write address, chip select and write enable signals are always synchronous (registered.) The read data and read address signals can be synchronous or asynchronous. Reset is asynchronous. All write signals share the same clock, and clock enable. All read signals share the same clock and clock enable. Reset is shared by both the read and write signals. Table 3 shows the possible sources for the clock, clock enable and initialization signals for the various registers.

Figure 2. Pseudo Dual-Port SRAM Block Diagram

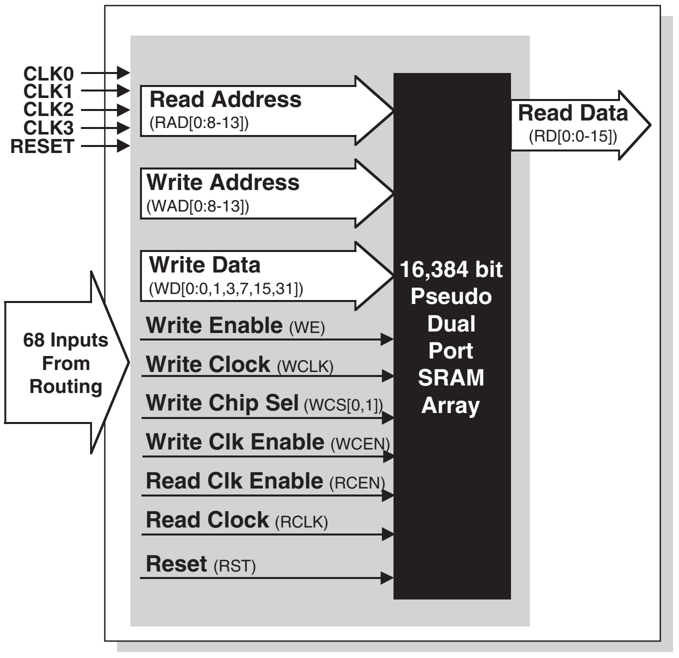


Table 3. Register Clock, Clock Enable, and Reset in Pseudo Dual-Port SRAM Mode

Register	Input	Source
Write Address, Write Data, Write Enable, and Write Chip Select	Clock	Chosen from WCLK or one of the global clocks (CLK0 -CLK3). The selected signal can be inverted if desired.
	Clock Enable	Chosen from WCEN or two of the global clocks (CLK0 - CLK3). The selected signal can be inverted if desired.
	Reset	Created by the logical OR of the global reset signal and RST. RST may have inversion if desired.
Read Data and Read Address	Clock	Chosen from RCLK or one of the global clocks (CLK0 - CLK3). The selected signal can be inverted if desired.
	Clock Enable	Chosen from RCEN or two of the global clocks (CLK1 - CLK2). The selected signal can be inverted if desired.
	Reset	Created by the logical OR of the global reset signal and RST. RST may have inversion if desired.

Single-Port SRAM Mode

In Single-Port SRAM mode, one port is shared by the Read/Write address lines, and can access up to 16,384-bits of memory. Data widths of 1, 2, 4, 8, 16 and 32 are supported by the MFB. Figure 3 shows the block diagram of the single-port SRAM.

Write data, write address, chip select and write enable signals are always synchronous (registered). The read data and read address signals can be synchronous or asynchronous. Reset is asynchronous. All signals share a common clock, clock enable, and reset. Table 4 shows the possible sources for the clock, clock enable and reset signals.

Figure 3. Single-Port SRAM Block Diagram

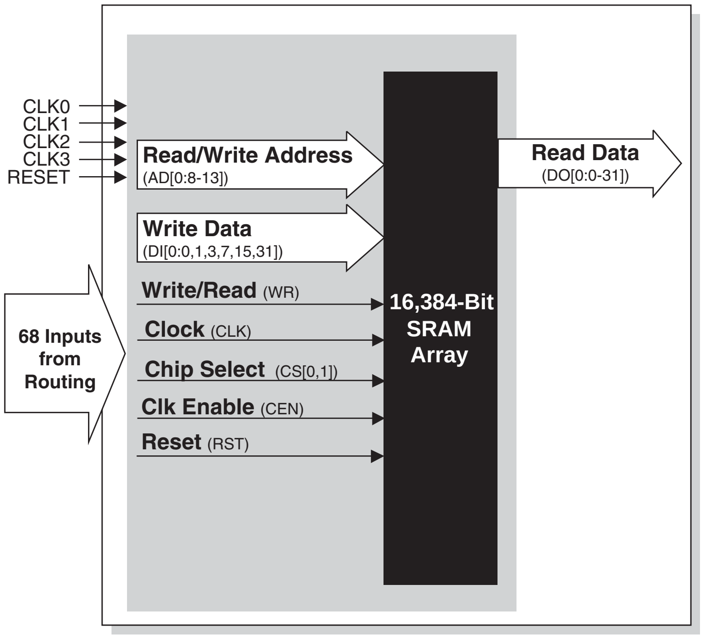


Table 4. Register Clock, Clock Enable, and Reset in Single-Port SRAM Mode

Register	Input	Source
Address, Write Data, Read Data, Read/Write, and Chip Select	Clock	CLK or one of the global clocks (CLK0 - CLK3). Each of these signals can be inverted if required.
	Clock Enable	CEN or two of the global clocks (CLK1 - CLK 2). Each of these signals can be inverted if required.
	Reset	Created by the logical OR of the global reset signal and RST. RST is routed by the multifunction array from GRP, with inversion if desired.

FIFO Mode

In FIFO mode the multifunction array is configured as a FIFO (First In First Out) buffer with built in control. The read and write clocks are independent of each other but can be tied together if the application requires it. Four flags show the status of the FIFO: Full, Almost Full, /Empty and /Almost Empty. /Empty and /Almost Empty are negative true signals. The thresholds for almost full and almost empty are programmable by the user. It is possible to reset the read pointer, allowing support of frame retransmit in communications applications.

The Almost Full and /Almost Empty flags indicate the status of the stack pointer with respect to Full and /Empty. Almost Full is an offset subtracted from the highest memory address, and /Almost Empty is an offset added to the lowest memory address. These flags are defined in the instantiation template via the lpm_amempty_flag and lpm_amfull_flag parameters. Values for both can range from 1 to the maximum number of address locations.

In this mode one port accesses 16,384-bits of memory. Data widths of 1, 2, 4, 8, 16 and 32 are supported by the MFB. Figure 4 shows the block diagram of the FIFO. These MFB blocks are cascaded to create FIFO sizes larger than 16K. Cascading sometimes requires extra logic elements like counters that occupy additional macrocells. For width cascading, no external logic is required if the depth can fit into the maximum depth of a single block. However, for depth cascading, external counters are needed. For example, in a 16K x 16 (depth x width) FIFO configuration, no extra counters are needed. This is width cascading. For 32K X 1 FIFO, counters will be needed since the depth (32K) is larger than the depth available in a single MFB (16K). When the configuration needs both width and depth cascading, the software optimizes it to the maximum depth with width cascading.

Write data, write enable, flag outputs and read enable are synchronous. The Write Data, Almost Full flag and Full flag share the same clock and clock enables. Read outputs are synchronous. The Read Data, /Empty flag and /Almost Empty flag share the same clock and clock enables. Reset is shared by all signals. Table 5 shows the possible sources for the clock, clock enable and reset signals for the various registers.

The Full and Almost Full flags are based on the write port. The Full and Almost Full flags change state only on write clock rising edges. The Empty and Almost Empty flags are based on the read port. The /Empty and /Almost Empty flags change state only on read clock rising edges. FIFO flag latency is reduced when both read/write clocks are left running. If the read/write clocks are free running, data is inserted/retrieved by synchronously controlling the Read/Write Enable strobes.

Once the Full or Empty flag goes active, the internal FIFO pointer is frozen at the last active value, and no operation is performed on the memory. In other words, Writes to the FIFO after the full flag goes active or Reads after the Empty flag goes active are ignored.

Figure 4. FIFO Block Diagram

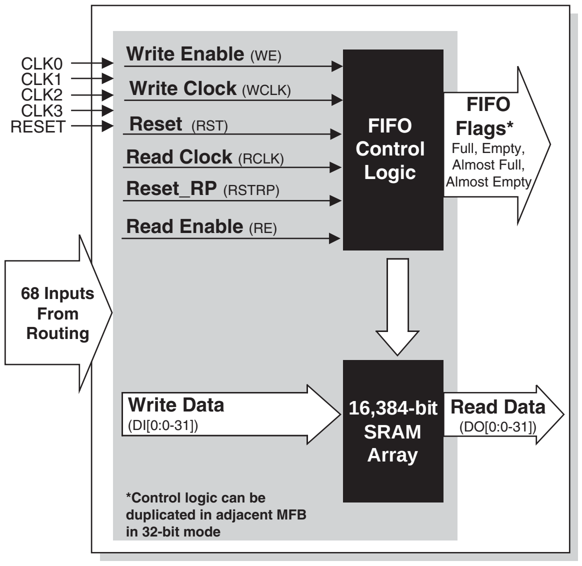


Table 5. Register Clocks, Clock Enables, and Initialization in FIFO Mode

Register	Input	Source
Write Data, Write Enable	Clock	WCLK or one of the global clocks (CLK0 - CLK3). Each of these signals can be inverted if required.
	Clock Enable	WEN or two of the global clocks (CLK1 - CLK 2). Each of these signals can be inverted if required.
	Reset	N/A
Full and Almost Full Flags	Clock	WCLK or one of the global clocks (CLK0 - CLK3). Each of these signals can be inverted if required.
	Clock Enable	WEN or two of the global clocks (CLK1 - CLK 2). Each of these signals can be inverted if required.
	Reset	Created by the logical OR of the global reset signal and RST. RST is routed by the multifunction array from GRP, with inversion if desired
Read Data, Empty and Almost Empty Flags	Clock	RCLK or one of the global clocks (CLK0 - CLK3). Each of these signals can be inverted if required.
	Clock Enable	REN or two of the global clocks (CLK1 - CLK 2). Each of these signals can be inverted if required.
	Reset	Created by the logical OR of the global reset signal and RST. RST is routed by the multifunction array from GRP, with inversion if desired

CAM Mode

In CAM mode the multifunction array is configured as a ternary Content Addressable Memory (CAM.) CAM behaves like a reverse memory where the input to the memory is data and the output is an address at which the input data is located. It can be used to perform a variety of high-performance look-up functions. As such CAM has two modes of operation. In write or update mode the CAM behaves as a RAM, and the data is written to the supplied address. When reading or comparing, data is supplied to the CAM. If that data matches any of the entries in the CAM array the Match or Multi-Match (if there is more than one match) flag is set to true, and the lowest address with matching data is the output. The MFB can be configured as a CAM that contains 128 entries of 48 bits. Figure 5 shows the block diagram of the CAM

To further enhance the flexibility of the CAM a mask register is available for both update and compare operations. If the mask register is enabled during updates, bits corresponding to those set to a '1' in the mask register are not updated. If it is enabled during compare operations, bits corresponding to those set to a '1' in the mask register are not included in the compare. A Write Don't Care signal allows don't cares to be programmed into the CAM if desired. As with other write operations, the mask register controls this.

Data is written into the Mask Register by enabling the WrMask (Write Mask) bit. When WrMask is enabled during write mode, the data written into the CAM array is also automatically stored in the Mask Register. Once the mask is configured, it can be used during Update and Compare modes by enabling the EnMask (Enable Mask) bit.

The WrDC (Write Don't Care) bit functions much like the EnMask and WrMask inputs. When in write mode and WrDC is set to a 1, the data written to the CAM array during the next write operation is stored in the Write Don't Care register.

The Write/Comp Data, Write Address, Write Enable, Write Chip Select, and Write Don't Care signals are synchronous. The CAM Output signals, Match flag, and Multi-Match flag signals can be either synchronous or asynchronous. The Enable mask register input is not latched but must meet setup and hold times relative to the Write Clock. All inputs must use the same clock and clock enable signals. All outputs must use the same clock, and clock enable signals. Reset is common for both inputs and outputs. Table 6 shows the allowable sources for clock, clock enable, and reset for the various CAM registers.

For additional information on the CAM memory in the ispXPLD family of devices please refer to application note AN8071, *Content Addressable Memory Applications for ispXPLD Devices.*

Figure 5. CAM Mode

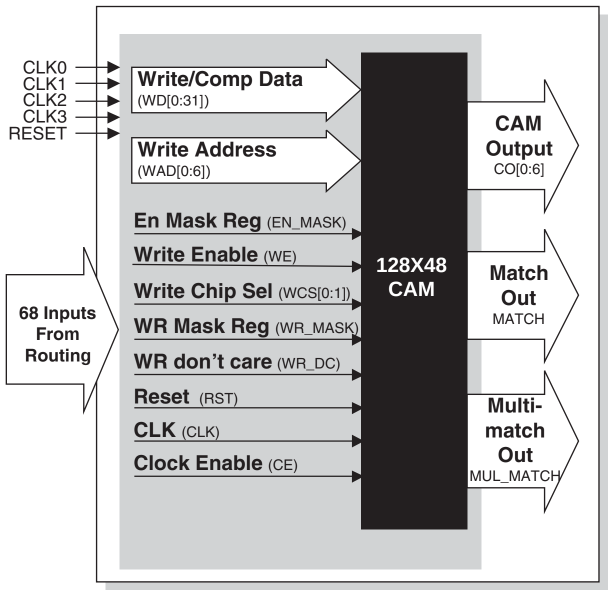


Table 6. Register Clocks, Clock Enables, and Initialization in CAM Mode

Register	Input	Source
Write data, Write address, Enable mask register, Write enable, write chip select, and write don't care, CAM Output, Match, and Multimatch	Clock	CLK or one of the global clocks (CLK0 - CLK3). Each of these signals can be inverted if required.
	Clock Enable	WE or two of the global clocks (CLK1 - CLK 2). Each of these signals can be inverted if required.
	Reset	Created by the logical OR of the global reset signal and RST. RST is routed by the multifunction array from GRP, with inversion if desired

Including Memory in ispXPLD 5000MX Designs

To use memory in ispXPLD 5000MX designs the desired memory function must be instantiated into the HDL source code describing the design. This is done in the form of LPM primitives, which are passed through the synthesis tool to Lattice backend design tools. These backend tools then work to implement the memory requested in the source code. The remainder of this document details the LPM primitives and example templates.

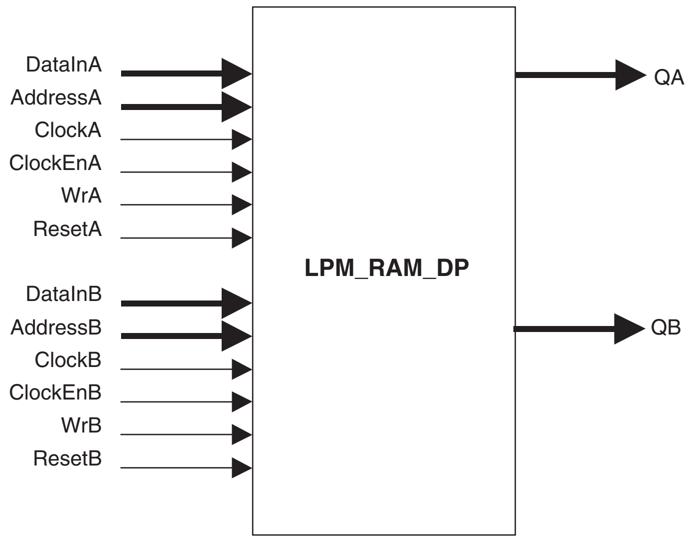
Configurable Memory Primitives

Configurable memory primitives are provided to allow easy configuration of the MFBs. These primitives are added to your design source. With the addition of parameters these memory primitives can be easily configured to match your design needs.

This section describes the six types of configurable memory primitives that are supported.

• LPM_RAM_DP	– Dual-Port RAM
• LPM_RAM_DP_PSEUDO	– Pseudo Dual Port RAM
• LPM_RAM_DQ	– Single-port RAM
• LPM_FIFO_DC	– FIFO
• LPM_CAM	– CAM
• LPM_ROM	– ROM

True Dual-Port Random Access Memory (LPM_RAM_DP)



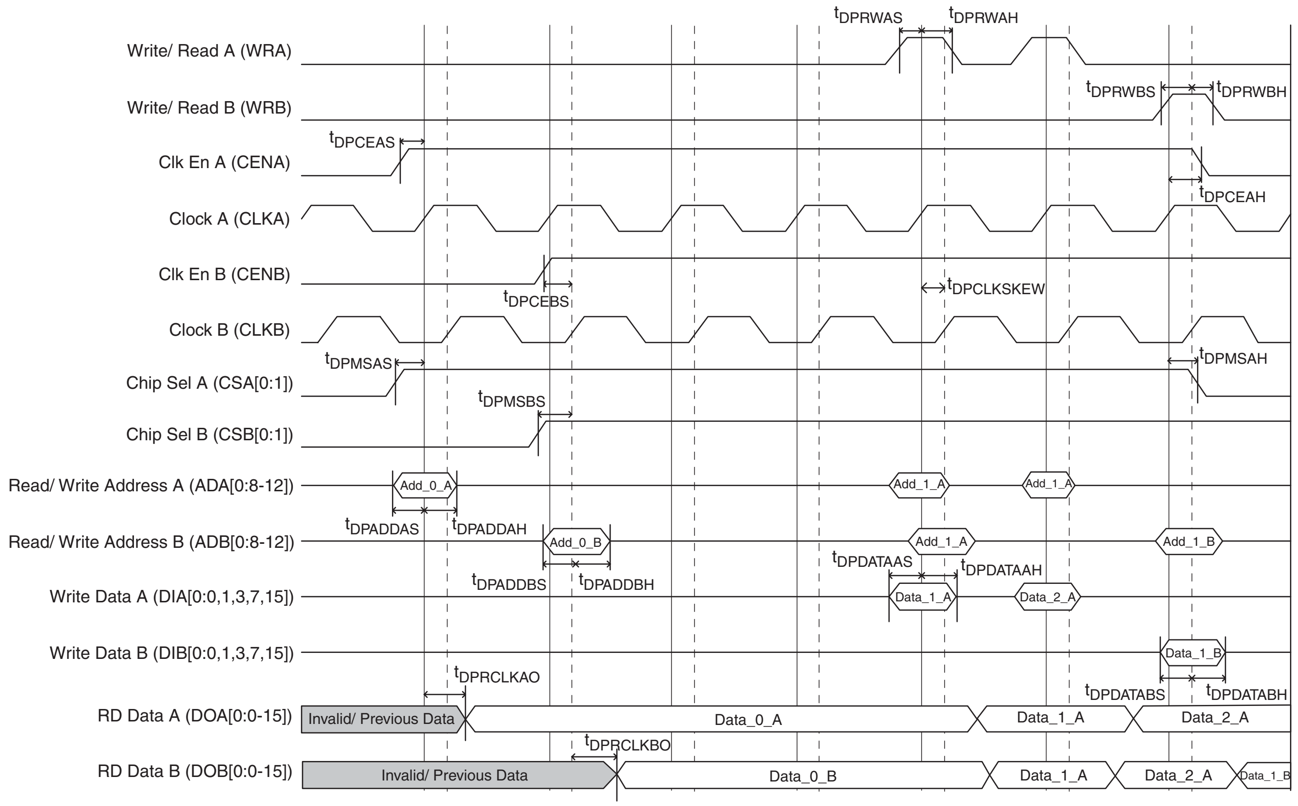
Ports

Port	Type	Description	Comments
QA	Out	Data out, port A	Port width is user defined
DataInA	In	Data in, port A	Port width is user defined
AddressA	In	Address, port A	Address depth is user defined
ClockA	In	Clock, port A	
ClockEnA	In	Clock Enable, port A	
WrA	In	Write Enable, Port A	
ResetA	In	Reset, port A	Asynchronous Reset
QB	Out	Data out, port B	Port width is user defined
DataInB	In	Data in, port B	Port width is user defined
AddressB	In	Address, port B	Address depth is user defined
ClockB	In	Clock, port B	
ClockEnB	In	Clock Enable, port B	
WrB	In	Write Enable, Port B	
ResetB	In	Reset, port B	Asynchronous Reset

Properties

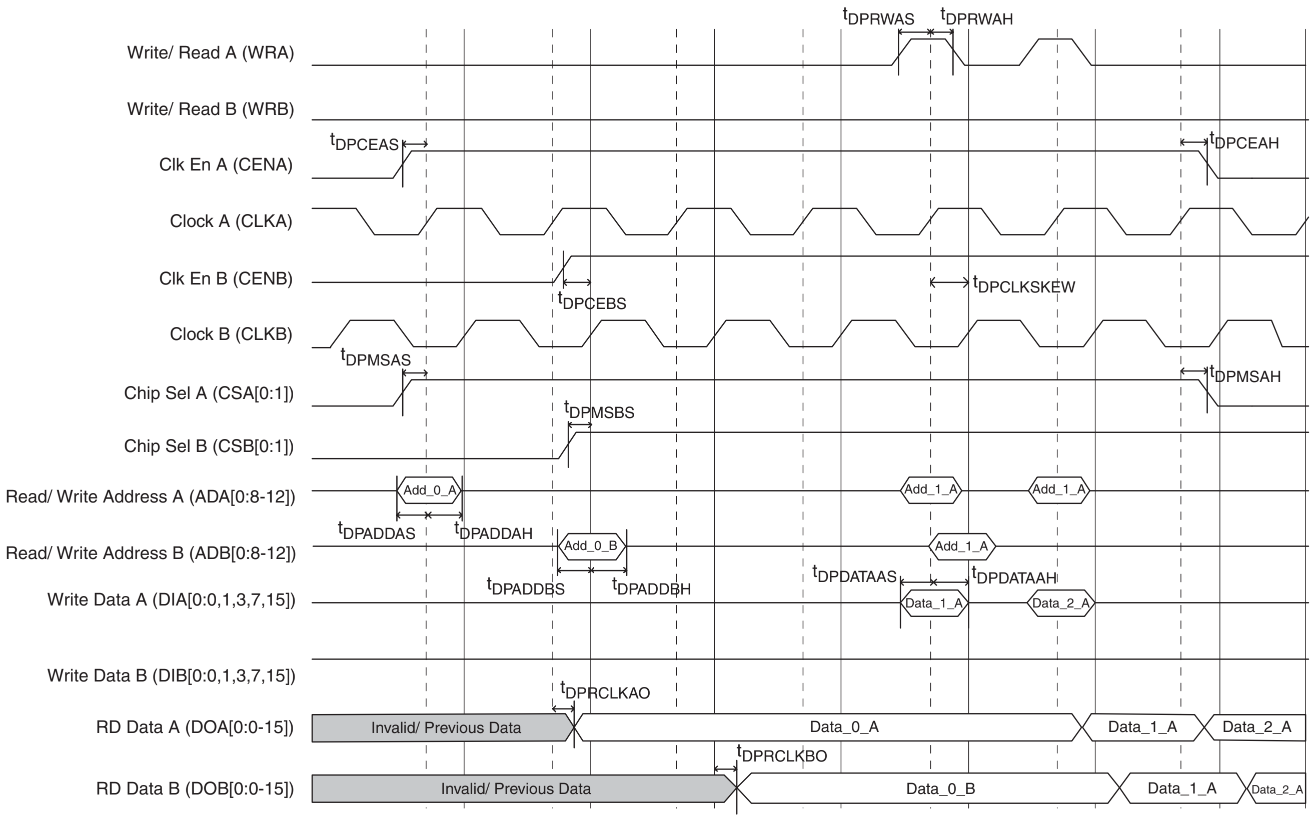
Parameter	Description	Comments	Value
lpm_widtha	Defines data width for port A	User-defined	Number of data bits
lpm_widthada	Defines address width for port A	User-defined	Number of address lines
lpm_numwordsa	Defines memory depth for port A	User-defined	Number of address locations
lpm_widthb	Defines data width for port B	User-defined	Number of data bits
lpm_widthadb	Defines address width for port B	User-defined	Number of address lines
lpm_numwordsb	Defines memory depth for port B	User-defined	Number of address locations
lpm_outdata	Defines read data to be synchronous or asynchronous	User-defined	Registered or Unregistered
lpm_indata	Defines write data to be synchronous	Synchronous	Registered
lpm_addressa_control	Defines that port A address lines will be synchronous	Synchronous	Registered
lpm_addressb_control	Defines that port B address lines will be synchronous	Synchronous	Registered
lpm_init_file	Defines initialization file	File for initializing data in the RAM	Name of the initialization file

True Dual Port RAM with Asynchronous Read



Note: While one port is writing and the other port tries to read or write at the same memory location, there must be a minimum $t_{DPCLKSKEW}$ between the two clocks.

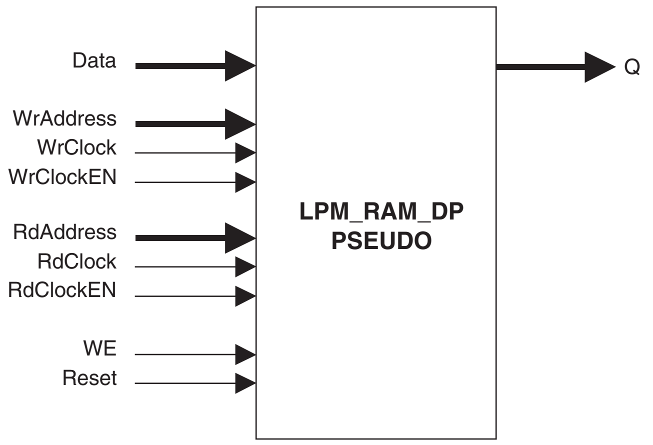
True Dual Port RAM with Synchronous Read



Note: While one port is writing and the other port tries to read or write at the same memory location, there must be a minimum $t_{DPCLKSKEW}$ between the two clocks.

For timing numbers, please refer to the ispXPLD 5000MX Data Sheet.

Pseudo Dual-Port Random Access Memory (LPM_RAM_PSEUDO)



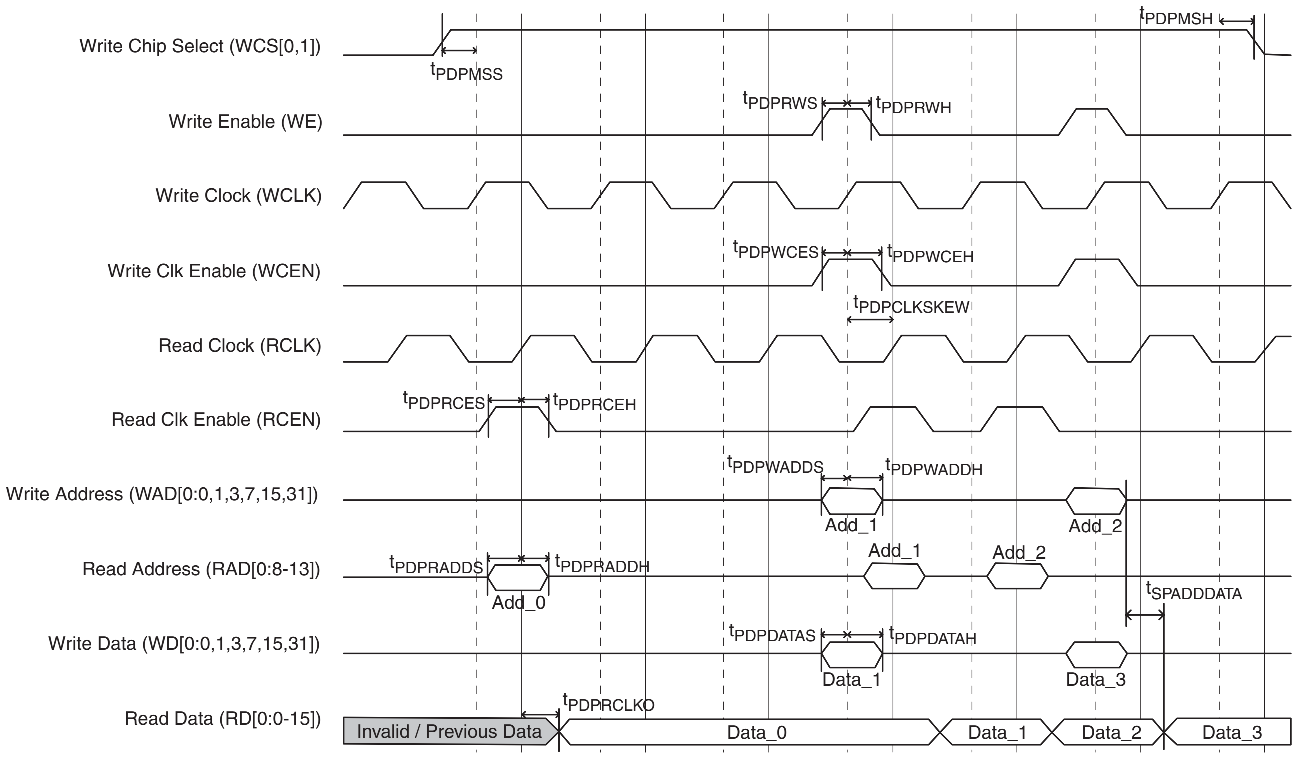
Ports

Port	Type	Description	Comments
Q	Out	Data out	Port width is user defined
Data	In	Data in	Port width is user defined
WrAddress	In	Write Address	Address Depth is user defined
WrClock	In	Write Clock	
WrClockEN	In	Write Clock Enable	
RdAddress	In	Read Address	Address Depth is user defined
RdClock	In	Read Clock	
RdClockEN	In	Read Clock Enable	
WE	In	Write Enable	
Reset	In	Reset	Asynchronous Reset

Properties

Parameter	Description	Comments	Value
lpm_widthw	Defines data width for write port	User-defined	Number of data bits to write
lpm_widthadw	Defines address width for write port	User-defined	Number of write address lines
lpm_numwordsw	Defines memory depth for write	User-defined	Number of address locations
lpm_widthr	Defines data width for read port	User-defined	Number of data bits to read
lpm_widthadr	Defines address width for read port	User-defined	Number of read address lines
lpm_numwordsr	Defines memory depth for read	User-defined	Number of read address locations
lpm_outdata	Defines read data to be synchronous or asynchronous	User-defined	Registered or unregistered
lpm_addressr_control	Defines that read address lines will be synchronous	Synchronous	Registered
lpm_addressw_control	Defines that write address lines will be synchronous	Synchronous	Registered
lpm_init_file	Defines initialization file	File for initializing data in the RAM	Name of the initialization file

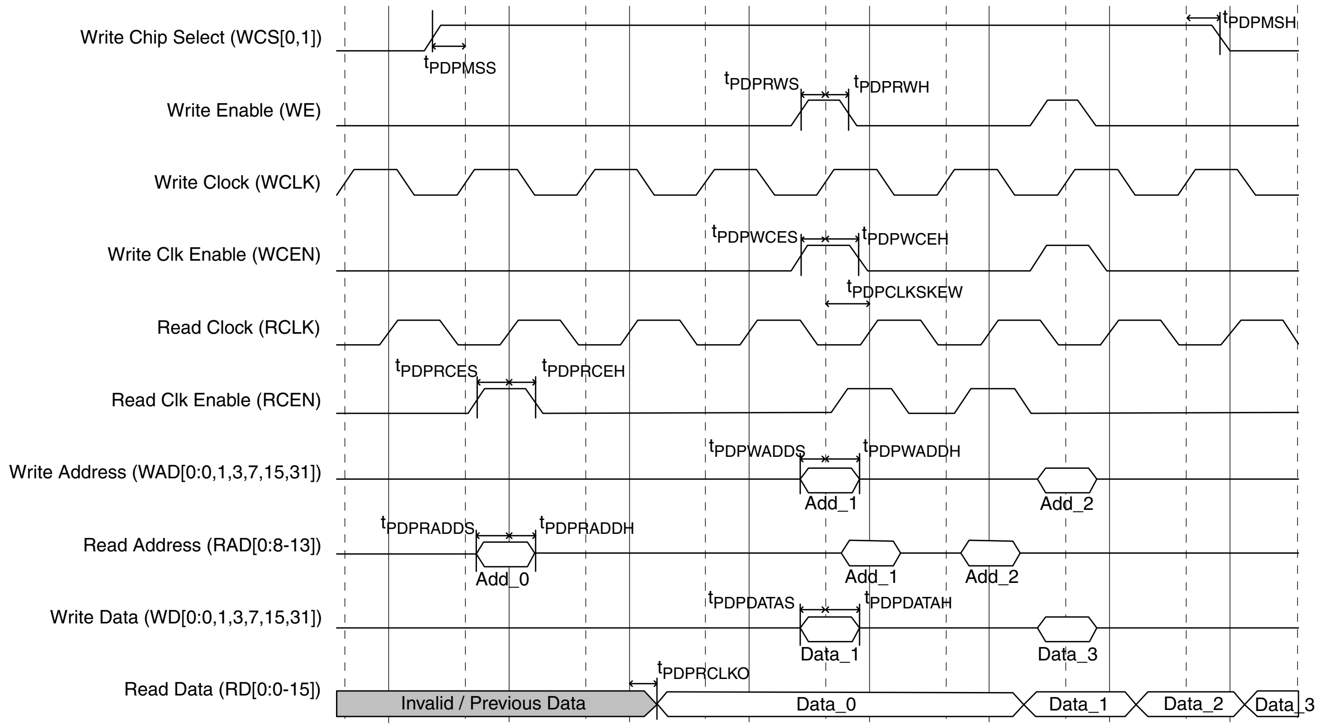
Pseudo Dual Port RAM with Asynchronous Read



Notes:
While Write port is writing and the Read port tries to read at the same memory location, there must be a minimum $t_{PDPCLKSKEW}$ between the two clocks. As shown above, if Add_1 is where the the read and write is occurring then there should be a minimum clock skew of $t_{PDPCLKSKEW}$ between the RCLK and WCLK.

Further, when we read from an address and in the next Write clock cycle, we start writing to that address, then the Read Data gets updated $t_{SPADDDATA}$ after the address is stable. This is shown, when we are reading Add_2 and the Read Data is Data_2. In the next write clock cycle, Add_2 is witten with Data_3. The Read Data gets updated $t_{SPADDDATA}$ after the Add_2 is stable. Both Data_2 and Data_3 are from the same location Add_2.

Pseudo Dual Port RAM with Synchronous Read

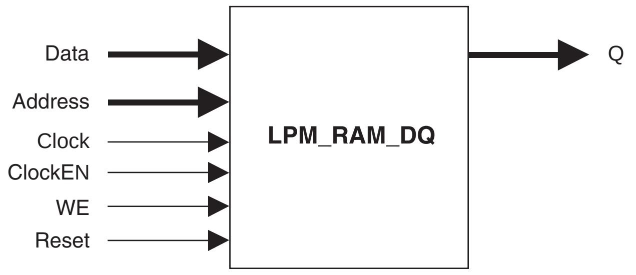


Notes:
While the Write port is writing and the Read port tries to read at the same memory location, there must be a minimum $t_{PDPCLKSKEW}$ between the two clocks. As shown above, if Add_1 is where the the read and write is occurring then there should be a minimum clock skew of $t_{PDPCLKSKEW}$ between the RCLK and WCLK.

Further, when we read from an address and in the next Write clock cycle, we start writing to that address, then the Read Data gets updated $t_{SPADDDATA}$ after the address is stable. This is shown when we are reading Add_2 and the Read Data is Data_2. In the next write clock cycle, Add_2 is witten with Data_3. The Read Data gets updated $t_{SPADDDATA}$ after the Add_2 is stable. Both Data_2 and Data_3 are from the same location, Add_2.

For timing numbers, please refer to the ispXPLD 5000MX Data Sheet.

Single-Port RAM (LPM_RAM_DQ)



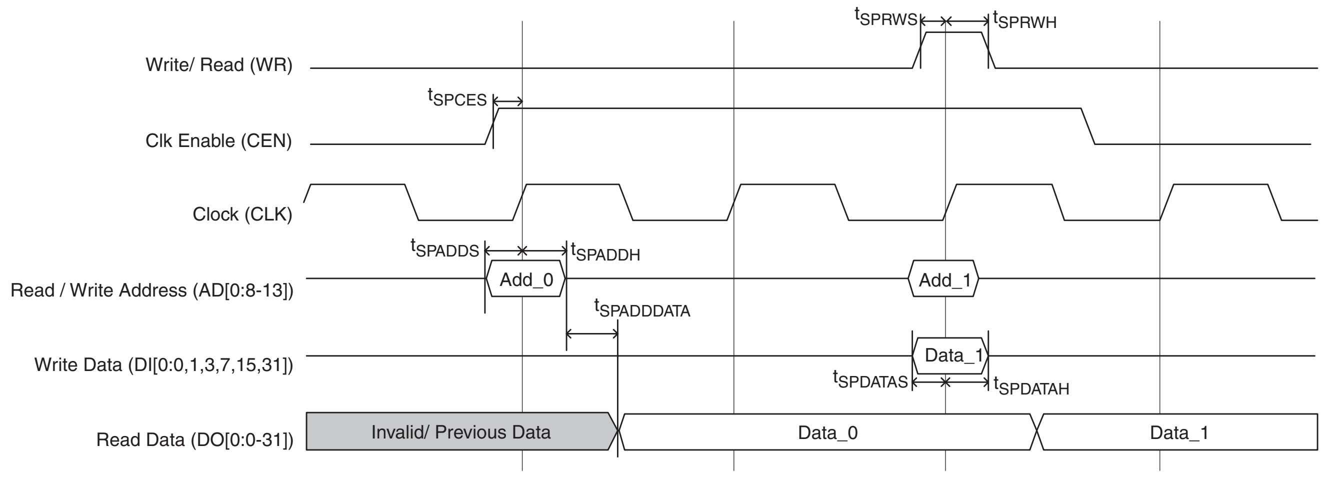
Ports

Port	Type	Description	Comments
Q	Out	Data Out	Port width is user defined
Data	In	Data In	Port width is user defined
Address	In	Read/Write Address	Port width is user defined
Clock	In	Clock	
ClockEn	In	Clock Enable	
WE	In	Write Enable	
Reset	In	Reset	Asynchronous Reset

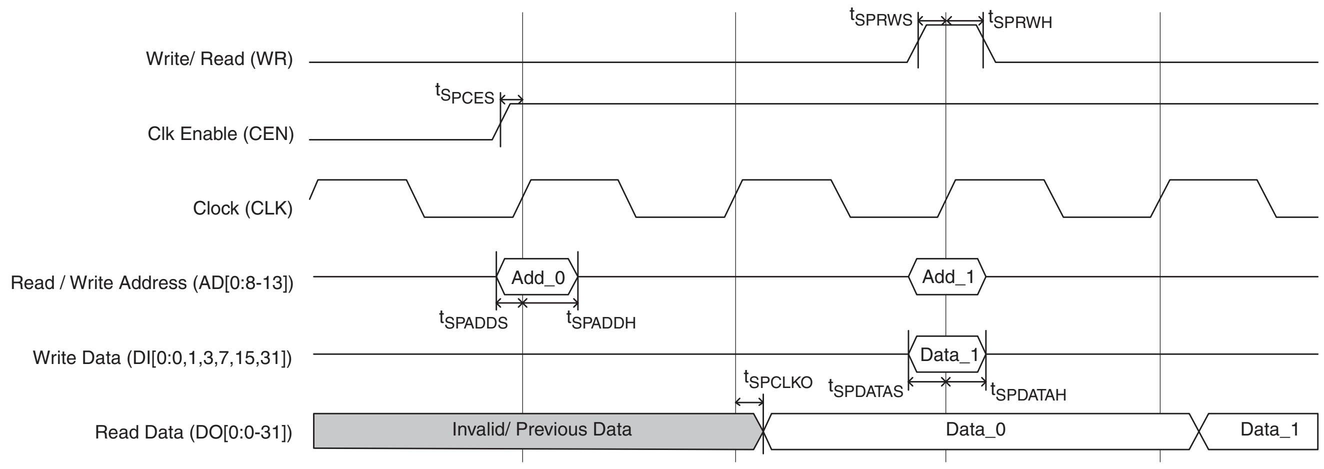
Properties

Parameter	Description	Comments	Value
lpm_width	Defines Data width	User-defined	Number of data bits
lpm_widthad	Defines address width	User-defined	Number of address lines
lpm_numwords	Defines memory depth	User-defined	Number of address locations
lpm_outdata	Defines read data to be synchronous or asynchronous	User-defined	Registered or unregistered
lpm_address_control	Defines the value of the read address lines. In unregistered mode, the output toggles at each address change. In registered mode, Q is toggled by the clock.	User-defined	Registered or unregistered
lpm_init_file	Defines initialization file	File for initializing data in the RAM	Name of the initialization file

Single Port RAM with Asynchronous Read

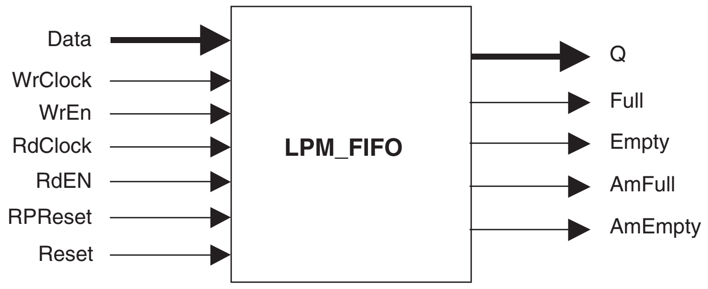


Single Port RAM with Synchronous Read



For timing numbers, please refer to the ispXPLD 5000MX Data Sheet.

First-In-First-Out Memory (LPM_FIFO)



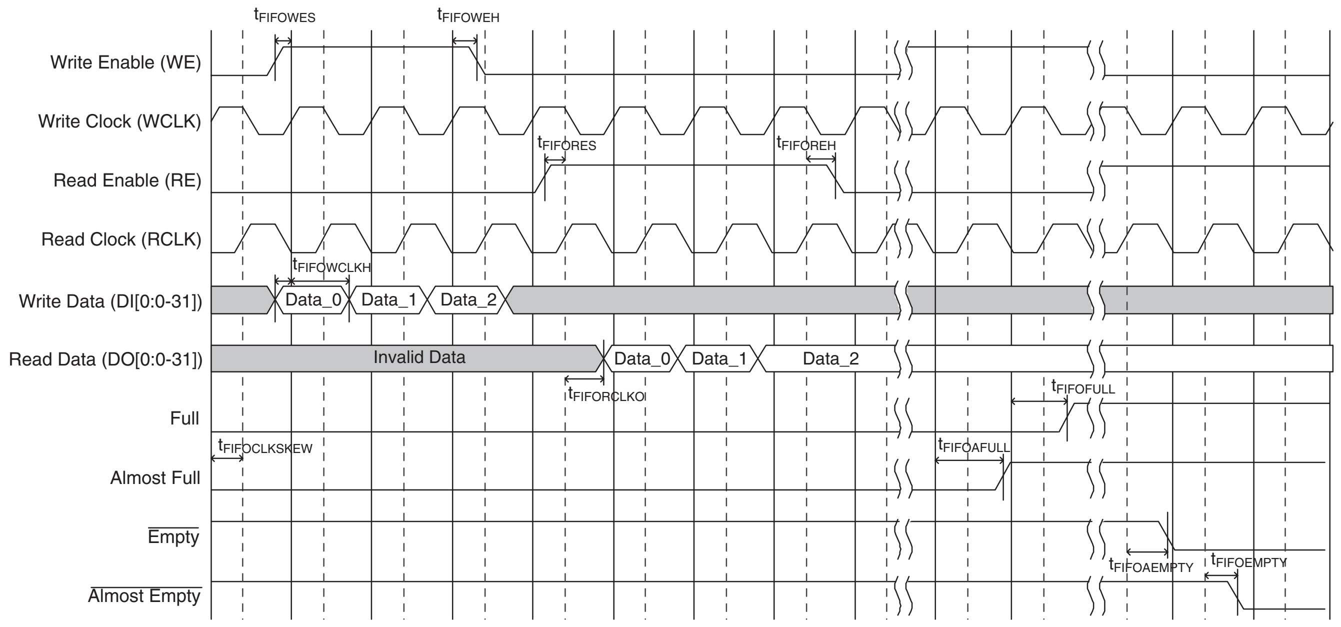
Ports

Port	Type	Description	Comments
Q	Out	Data Out	Port width is user defined
Full	Out	Flag	Set when FIFO is Full, user defined
Empty	Out	Flag	Clear (logic “0”) when FIFO is empty
AmFull	Out	Flag	Set at user defined value
AmEmpty	Out	Flag	Clear (logic “0”) at user defined value
Data	In	Data In	Port width is user defined
WrClock	In	Write Clock	
WrEn	In	Write Enable	
RdClock	In	Read Clock	
RdEn	In	Read Enable	
RPReset	In	Read Control Pointer	
Reset	In	Reset	Asynchronous Reset

Properties

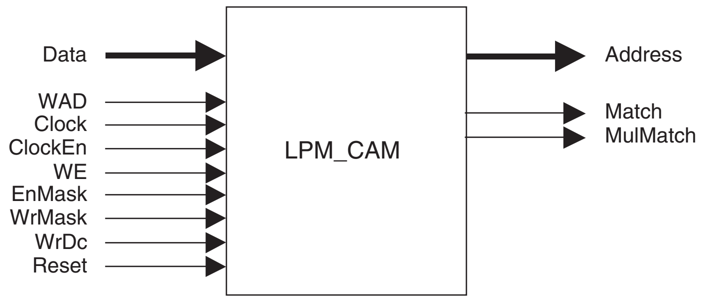
Parameter	Description	Comments	Value
lpm_width	Defines data width	User-defined	Number of data bits
lpm_widthu	Defines address width	User-defined	Number of address lines required to access lpm_numwords FIFO entries
lpm_numwords	Defines memory depth	User-defined	Number of data entries the FIFO can store
lpm_amfull_flag	Almost full flag	User-defined offset	Offset subtracted from lpm_numwords
lpm_amempty_flag	Almost empty flag	User-defined offset	Offset added to address 0

FIFO



For timing numbers, please refer to the ispXPLD 5000MX Data Sheet.

Content Addressable Memory (LPM_CAM)



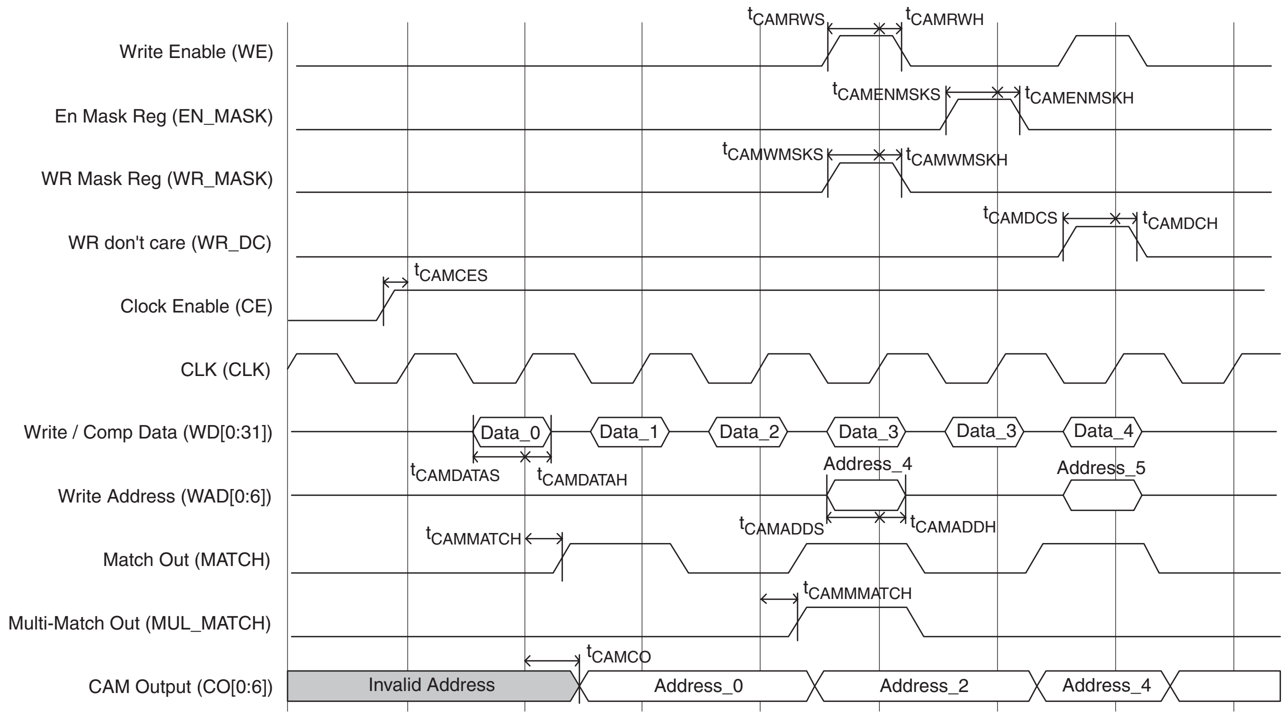
Ports

Port	Type	Description	Comments
Address	Out	Write Address	Port width is user defined
Match	Out	Flag	Set when match
MulMatch	Out	Flag	Set when Multiple matches
Data	In	Data In	Port width is user defined
Wad	In	Write Address	Port width is user defined
Clock	In	Clock	
ClockEn	In	Clock Enable	
We	In	Write Enable	
EnMask	In	Enable Mask Register	Enables use of global mask register
WrMask	In	Write Mask Register	Enables writing to the Mask Register
WrDC	In	Write Don’t Care	Don’t Cares can be written to the CAM
Reset	In	Reset	Asynchronous Reset

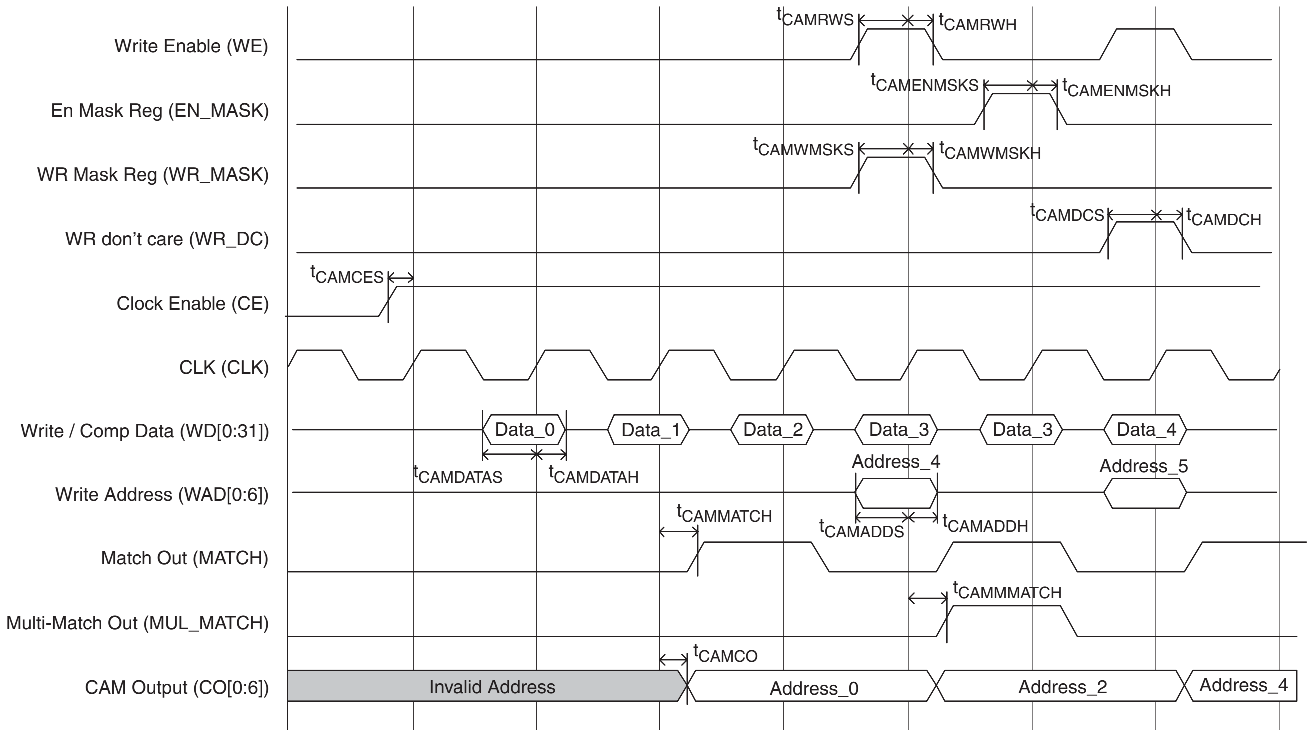
Properties

Parameter	Description	Comments	Value
lpm_width	Defines data width	User-defined	Number of data bits
lpm_widthad	Defines address width	User-defined	Number of address lines
lpm_numwords	Defines memory depth	User-defined	Number of address locations
lpm_init_file	Defines initialization file	File for initializing data in the CAM	Name of the initialization file

CAM with Asynchronous Read

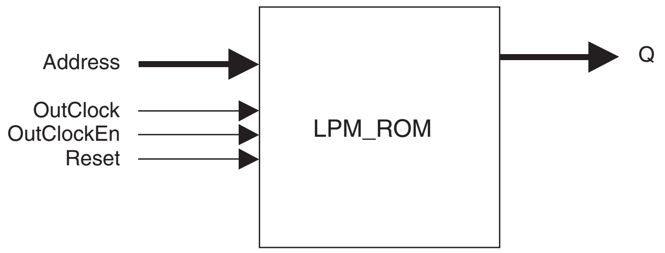


CAM with Synchronous Read



For timing numbers, please refer to the ispXPLD 5000MX Data Sheet.

Read-Only Memory (LPM_ROM)



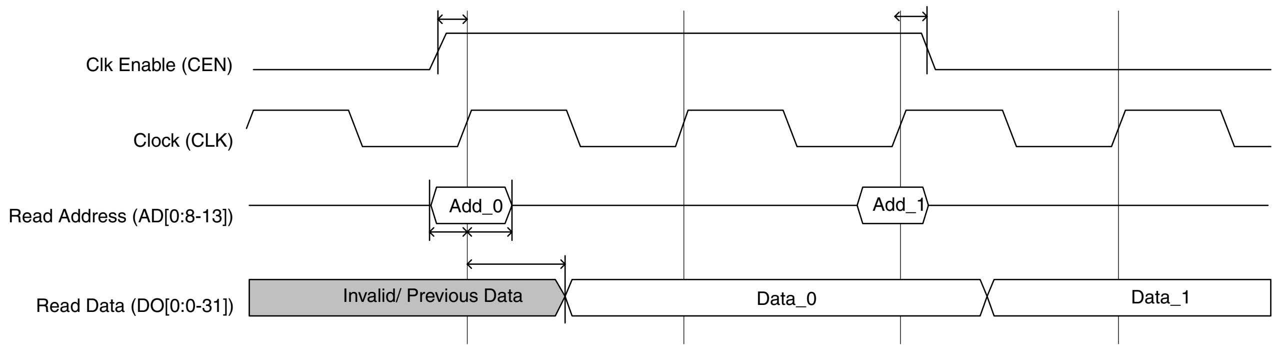
Ports

Port	Type	Description	Comments
Q	Out	Data Out	Port width is user defined
Address	In	Read Address	Port width is user defined
OutClock	In	Clock	
OutClockEn	In	Clock Enable	
Reset	In	Reset	Asynchronous Reset

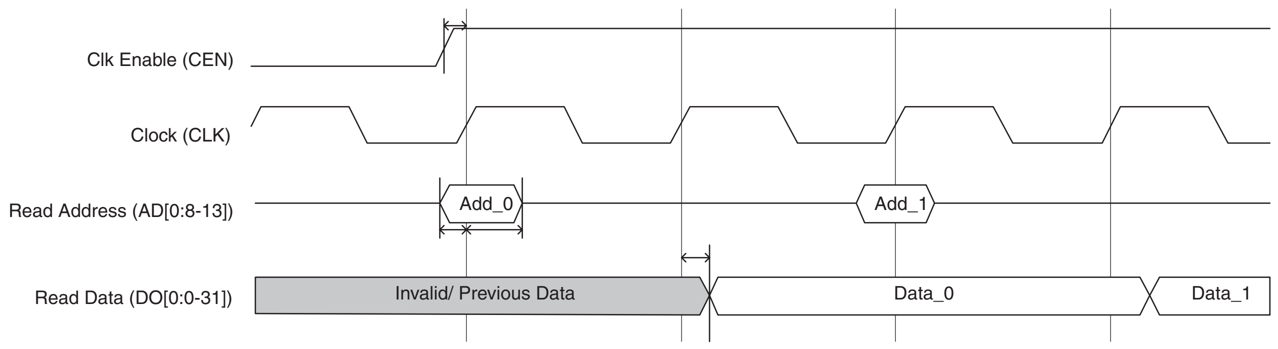
Properties

Parameter	Description	Comments	Value
lpm_width	Defines data width	User-defined	Number of data bits
lpm_widthad	Defines address width	User-defined	Number of address lines
lpm_numwords	Defines memory depth	User-defined	Number of address locations
lpm_outdata	Defines read data to be synchronous or asynchronous	User-defined	Registered or unregistered
lpm_address_control	Defines the value of the read address lines. In unregistered mode, the output toggles at each address change. In registered mode, Q is toggled by the clock.	User-defined	Registered or unregistered
lpm_init_file	Defines initialization file	File for initializing data in the ROM	Name of the initialization file

ROM with Asynchronous Read



ROM with Synchronous Read



For timing numbers, please refer to the ispXPLD 5000MX Data Sheet.

Appendix A. Memory Primitive Source Examples (Verilog)

Note: The Verilog templates shown here can also be found in the software examples directory:

\<isptools_instalation_directory>\ispcpld\examples\ispXPLD\verilog

True Dual-Port Random Access Memory (LPM_RAM_DP)

```
module tramdp8kx2x2(
           QA,
           QB,
           DataInA,
           AddressA,
           DataInB,
           AddressB,
           ClockA,
           ClockEnA,
           ClockB,
           ClockEnB,
           WrA,
           WrB,
           ResetA,
           ResetB);
output [1:0] QA;
output [1:0] QB;
input [1:0] DataInA;
input [12:0] AddressA;
input [1:0] DataInB;
input [12:0] AddressB;
input ClockA,ClockEnA,ClockB,ClockEnB,WrA,WrB,ResetA,ResetB;

L_RAMDP
U0(.QA(QA),.QB(QB),.DataInA(DataInA),.AddressA(AddressA),.DataInB(DataInB),.A
ddressB(AddressB),.ClockA(ClockA),.ClockEnA(ClockEnA),.ClockB(ClockB),.ClockE
nB(ClockEnB),.WrA(WrA),.WrB(WrB),.ResetA(ResetA),.ResetB(ResetB));

defparam U0.lpm_widtha=2;
defparam U0.lpm_widthada=13;
defparam U0.lpm_numwordsa=8192;
defparam U0.lpm_widthb=2;
defparam U0.lpm_widthadb=13;
defparam U0.lpm_numwordsb=8192;
defparam U0.lpm_outdata    = “REGISTERED”;
defparam U0.lpm_addressa_control  = “REGISTERED”;
defparam U0.lpm_addressb_control  = “REGISTERED”;
defparam U0.lpm_init_file = “RAM_init”;

endmodule


module L_RAMDP(
           QA,
           QB,
           DataInA,
           AddressA,
           DataInB,
           AddressB,
           ClockA,
           ClockEnA,
           ClockB,
           ClockEnB,
           WrA,
           WrB,
           ResetA,
           ResetB);

   parameter lpm_type = “LPM_RAM_DP”;
   parameter lpm_widtha     = 1;
   parameter lpm_widthada   = 1;
   parameter lpm_numwordsa  = 1;
   parameter lpm_widthb     = 1;
   parameter lpm_widthadb   = 1;
   parameter lpm_numwordsb  = 1;
   parameter lpm_indata     = “REGISTERED”;
   parameter lpm_outdata    = “UNREGISTERED”;
   parameter lpm_addressa_control  = “REGISTERED”;
   parameter lpm_addressb_control  = “REGISTERED”;
   parameter lpm_hint = “UNUSED”;
   parameter lpm_init_file = “dummy”;

           output [lpm_widtha-1:0] QA;
           output [lpm_widthb-1:0] QB;
           input [lpm_widtha-1:0] DataInA;
           input [lpm_widthada-1:0] AddressA;
           input [lpm_widthb-1:0] DataInB;
           input [lpm_widthadb-1:0] AddressB;
           input      ClockA,ClockEnA,ClockB,ClockEnB,WrA,WrB,ResetA,ResetB;

endmodule  //lpm_ramdp
```

Pseudo Dual-Port Random Access Memory (LPM_RAM_PSEUDO)

```
module tramdps16kwx2rx2(
           Q,
           Data,
           WrAddress,
           RdAddress,
           WrClock,
           WrClockEn,
           RdClock,
           RdClockEn,
           WE,
           Reset);
output [1:0] Q;
input [1:0] Data;
input [13:0] WrAddress;
input [13:0] RdAddress;
input WrClock,WrClockEn,RdClock,RdClockEn,WE,Reset;

L_RAMDPS
U0(.Q(Q),.Data(Data),.WrAddress(WrAddress),.RdAddress(RdAddress),.WrClock(WrC
lock),.WrClockEn(WrClockEn),.RdClock(RdClock),.RdClockEn(RdClockEn),.WE(WE),.
Reset(Reset));

defparam U0.lpm_widthw=2;
defparam U0.lpm_widthadw=14;
defparam U0.lpm_numwordsw=16384;
defparam U0.lpm_widthr=2;
defparam U0.lpm_widthadr=14;
defparam U0.lpm_numwordsr=16384;
defparam U0.lpm_outdata    = “REGISTERED”;
defparam U0.lpm_addressr_control  = “REGISTERED”;
defparam U0.lpm_init_file=”RAM_init”;


endmodule

module L_RAMDPS(
           Q,
           Data,
           WrAddress,
           RdAddress,
           WrClock,
           WrClockEn,
           RdClock,
           RdClockEn,
           WE,
           Reset);

   parameter lpm_type = “LPM_RAM_DP_PSEUDO”;
   parameter lpm_widthw     = 1;
   parameter lpm_widthr     = 1;
   parameter lpm_numwordsw  = 1;
   parameter lpm_widthadw   = 1;
```

```
        parameter lpm_widthadr   = 1;
        parameter lpm_numwordsr  = 1;
        parameter lpm_indata     = “REGISTERED”;
        parameter lpm_outdata    = “UNREGISTERED”;
        parameter lpm_addressw_control  = “REGISTERED”;
        parameter lpm_addressr_control  = “REGISTERED”;
        parameter lpm_hint = “UNUSED”;
        parameter lpm_init_file = “dummy”;


                output [lpm_widthr-1:0] Q;
                input [lpm_widthw-1:0] Data;
                input [lpm_widthadw-1:0] WrAddress;
                input [lpm_widthadr-1:0] RdAddress;
                input WrClock,WrClockEn,RdClock,RdClockEn,WE,Reset;

     endmodule // lpm_ram_dp_pseudo
```

Random Access Memory (LPM_RAM_DQ)

```
module tramdq16kx2(
           Q,
           Data,
           Address,
           Clock,
           ClockEn,
           WE,
           Reset);
output [1:0] Q;
input [1:0] Data;
input [13:0] Address;
input Clock,ClockEn,WE,Reset;

L_RAMDQ
U0(.Q(Q),.Data(Data),.Address(Address),.Clock(Clock),.ClockEn(ClockEn),.WE(WE
),.Reset(Reset));

defparam U0.lpm_width=2;
defparam U0.lpm_widthad=14;
defparam U0.lpm_numwords=16384;
defparam U0.lpm_outdata=”REGISTERED”;
defparam U0.lpm_address_control=”REGISTERED”;
defparam U0.lpm_init_file=”RAM_init”;

endmodule

module L_RAMDQ(
           Q,
           Data,
           Address,
           Clock,
           ClockEn,
           WE,
           Reset);

   parameter lpm_type = “LPM_RAM_DQ”;
   parameter lpm_width     = 1;
   parameter lpm_numwords  = 1;
   parameter lpm_widthad   = 1;
   parameter lpm_indata     = “REGISTERED”;
   parameter lpm_outdata    = “UNREGISTERED”;
   parameter lpm_address_control  = “REGISTERED”;
   parameter lpm_hint = “UNUSED”;
   parameter lpm_init_file = “dummy”;

           output [lpm_width-1:0] Q;
           input [lpm_width-1:0] Data;
           input [lpm_widthad-1:0] Address;
           input Clock,ClockEn,WE,Reset;

endmodule // lpm_ram_dq
```

First-In-First-Out Memory (LPM_FIFO_DC)

```
module test_fifo16kx2
(Q,Full,Empty,Almost-
Full,AlmostEmpty,Data,WrClock,WrEn,RdClock,RdEn,Reset,RPReset);

output [1:0] Q;
output Full,Empty,AlmostFull,AlmostEmpty;
input [1:0] Data;
input WrClock,WrEn,RdClock,RdEn,Reset,RPReset;

L_FIFO U0(.Q(Q),
          .Full(Full),
          .Empty(Empty),
          .AlmostFull(AlmostFull),
          .AlmostEmpty(AlmostEmpty),
          .Data(Data),
          .WrClock(WrClock),
          .WrEn(WrEn),
          .RdClock(RdClock),
          .RdEn(RdEn),
          .Reset(Reset),
          .RPReset(RPReset)
          );

defparam U0.lpm_width=2;
defparam U0.lpm_widthu=14;
defparam U0.lpm_numwords=16384;
defparam U0.lpm_amfull_flag=11;
defparam U0.lpm_amempty_flag=11;

endmodule

module                                              L_FIFO(Q,Full,Empty,Almost-
Full,AlmostEmpty,Data,WrClock,WrEn,RdClock,RdEn,Reset,RPReset) ;

parameter lpm_type = "LPM_FIFO_DC";
parameter lpm_width  = 1;
parameter lpm_widthu  = 1;
parameter lpm_numwords = 2;
parameter lpm_amfull_flag=1;
parameter lpm_amempty_flag=1;
parameter lpm_hint = "UNUSED";

output [lpm_width-1:0] Q;
output Full;
output Empty;
output AlmostFull;
output AlmostEmpty;
input  [lpm_width-1:0] Data;
input  WrClock;
input  WrEn;
input  RdClock;
input  RdEn;
input  Reset;
input  RPReset;

endmodule // lpm_fifo
```

Content Addressable Memory (LPM_CAM)

```
module
tcam128x48                                                  (Address,Match,Mul-
Match,Wad,Data,Clock,ClockEn,We,EnMask,WrMask,WrDc,Reset);

output [6:0] Address;
output Match,MulMatch;
input [47:0] Data;
input [6:0] Wad;
input Clock,ClockEn,We,EnMask,WrMask,WrDc,Reset;

L_CAM
U0(.Address(Address),.Match(Match),.MulMatch(MulMatch),.WrAddress(Wad),.Data(
Data),.Clock(Clock),.ClockEn(ClockEn),.WE(We),.EnMask(EnMask),.WrMask(WrMask)
,.WrDC(WrDc),.Reset(Reset));

defparam U0.lpm_width=48;
defparam U0.lpm_widthad=7;
defparam U0.lpm_numwords=128;
defparam U0.lpm_init_file= "CAM_init";

endmodule

module                                      L_CAM(Address,Match,MulMatch,WrAd-
dress,Data,Clock,ClockEn,WE,EnMask,WrMask,WrDC,Reset);

parameter lpm_type = "LPM_CAM";
parameter lpm_width  = 1;
parameter lpm_widthad  = 1;
parameter lpm_numwords = 1;
parameter lpm_hint = "UNUSED";
parameter lpm_init_file = "dummy";

output [lpm_widthad-1:0] Address;
output Match;
output MulMatch;
input [lpm_widthad-1:0] WrAddress;
input  [lpm_width-1:0] Data;
input  Clock;
input  ClockEn;
input  WE;
input  EnMask;
input  WrMask;
input  WrDC;
input  Reset;

endmodule // lpm_cam
```

Read-Only Memory (LPM_ROM)

```
module test_rom16kx2(
          Q,
          Address,
          OutClock,
          OutClockEn,
          Reset);
output [1:0] Q;
input [13:0] Address;
input OutClock,OutClockEn,Reset;

L_ROM
U0(.Q(Q),.Address(Address),.OutClock(OutClock),.OutClockEn(Out-
ClockEn),.Reset(Reset));

defparam U0.lpm_width=2;
defparam U0.lpm_widthad=14;
defparam U0.lpm_numwords=16384;
defparam U0.lpm_outdata=”REGISTERED”;
defparam U0.lpm_address_control=”UNREGISTERED”;
defparam U0.lpm_init_file =”ROM_init”;

endmodule

module L_ROM(
          Q,
          Address,
          OutClock,
          OutClockEn,
          Reset);

   parameter lpm_type = “LPM_ROM”;
   parameter lpm_width     = 1;
   parameter lpm_numwords  = 1;
   parameter lpm_widthad   = 1;
   parameter lpm_outdata    = “REGISTERED”;
   parameter lpm_address_control  = “UNREGISTERED”;
   parameter lpm_hint = “UNUSED”;
         parameter lpm_init_file = “dummy”;

         output [lpm_width-1:0] Q;
         input [lpm_widthad-1:0] Address;
         input OutClock,OutClockEn,Reset;

endmodule // lpm_rom
```

Appendix B. Memory Primitive Source Examples (VHDL)

Note: The VHDL templates shown here can also be found in the software examples directory:

\<isptools_instalation_directory>\ispcpld\examples\ispXPLD\VHDL

True Dual-Port Random Access Memory (LPM_RAM_DP)

```
library IEEE;
use IEEE.std_logic_1164.all;
LIBRARY lc5kmx;
USE lc5kmx.components.all;

entity tramdp8kx2x2 is

    port (
           DataInA          : in std_logic_vector( 1 downto 0);
           AddressA         : in std_logic_vector( 12 downto 0);
           DataInB          : in std_logic_vector( 1 downto 0);
           AddressB         : in std_logic_vector( 12 downto 0);
           ClockA           : in std_logic;
           ClockEnA         : in std_logic;
           ClockB           : in std_logic;
           ClockEnB         : in std_logic;
           WrA              : in std_logic;
           WrB              : in std_logic;
           ResetA           : in std_logic;
           ResetB           : in std_logic;
           QA               : out std_logic_vector(1 downto 0);
           QB               : out std_logic_vector(1 downto 0));
end tramdp8kx2x2 ;


architecture behave of tramdp8kx2x2 is

component L_RAMDP
     generic(
            LPM_TYPE       : string := “LPM_RAM_DP”;
            LPM_WIDTHA     : positive := 1;
            LPM_WIDTHADA   : positive := 1;
            LPM_NUMWORDSA  : positive := 2;
            LPM_WIDTHB     : positive := 1;
            LPM_WIDTHADB   : positive := 1;
            LPM_NUMWORDSB  : positive := 2;
            LPM_INDATA     : string :=”REGISTERED”;
            LPM_OUTDATA    : string :=”UNREGISTERED”;
            LPM_ADDRESSA_CONTROL    : string :=”REGISTERED”;
            LPM_ADDRESSB_CONTROL    : string :=”REGISTERED”;
            LPM_INIT_FILE  : string := “dummy”;
            LPM_HINT       : string :=”UNUSED”);

     port(
            DataInA : in std_logic_vector(LPM_WIDTHA-1 downto 0);
            AddressA:in std_logic_vector(LPM_WIDTHADA-1 downto 0);
            DataInB : in std_logic_vector(LPM_WIDTHB-1 downto 0);
            AddressB:in std_logic_vector(LPM_WIDTHADB-1 downto 0);
            ClockA        : in std_logic := ‘0’;
            ClockEnA      : in std_logic := ‘0’;
            ClockB        : in std_logic := ‘0’;
            ClockEnB      : in std_logic := ‘0’;
            WrA           : in std_logic;
            WrB           : in std_logic;
            ResetA        : in std_logic;
            ResetB        : in std_logic;
            QA     : out std_logic_vector(LPM_WIDTHA-1 downto 0);
            QB     : out std_logic_vector(LPM_WIDTHB-1 downto 0));
end component ;

begin
U0: L_RAMDP
         generic map (
                      LPM_WIDTHA    => 2,
                      LPM_WIDTHADA  => 13,
                      LPM_NUMWORDSA => 8192,
                      LPM_WIDTHB    => 2,
                      LPM_WIDTHADB  => 13,
                      LPM_NUMWORDSB => 8192,
    LPM_INDATA => “REGISTERED”,
                      LPM_OUTDATA   => “UNREGISTERED”,
                      LPM_ADDRESSA_CONTROL => “REGISTERED”,
                      LPM_ADDRESSB_CONTROL => “REGISTERED”,
                      LPM_INIT_FILE => “RAM_init”)
         port map (
                      DataInA       => DataInA,
                      AddressA      => AddressA,
                      DataInB       => DataInB,
                      AddressB      => AddressB,
                      ClockA        => ClockA,
                      ClockEnA      => ClockEnA,
                      ClockB        => ClockB,
                      ClockEnB      => ClockEnB,
                      WrA           => WrA,
                      WrB           => WrB,
                      ResetA        => ResetA,
                      ResetB        => ResetB,
                      QA            => QA,
                      QB            => QB);
end behave;
```

Pseudo Dual-Port Random Access Memory (LPM_RAM_PSEUDO)

```
library IEEE;
use IEEE.std_logic_1164.all;
LIBRARY lc5kmx;
USE lc5kmx.components.all;

entity tramdps16kwx2rx2 is

    port (
           Data         : in std_logic_vector(1 downto 0);
           WrAddress    : in std_logic_vector(13 downto 0);
           RdAddress    : in std_logic_vector(13 downto 0);
           WrClock      : in std_logic;
           WrClockEn    : in std_logic;
           RdClock      : in std_logic;
           RdClockEn    : in std_logic;
           WE           : in std_logic;
           Reset        : in std_logic;
           Q            : out std_logic_vector(1 downto 0));
end tramdps16kwx2rx2 ;


architecture struct of tramdps16kwx2rx2 is

component L_RAMDPS
  generic(
   lpm_type             : string := “LPM_RAM_DP_PSEUDO”;
   lpm_widthw           : integer := 1;
   lpm_widthr           : integer := 1;
   lpm_numwordsw        : integer := 1;
   lpm_widthadw         : integer := 1;
   lpm_widthadr         : integer := 1;
      lpm_numwordsr         : integer := 1;
     lpm_indata           : string := “REGISTERED”;
   lpm_outdata          : string := “UNREGISTERED”;
       lpm_addressw_control : string := “REGISTERED”;
   lpm_addressr_control : string := “REGISTERED”;
         lpm_init_file        : string := “dummy”;
   lpm_hint             : string := “UNUSED”);


  port(
           Data      : in std_logic_vector(lpm_widthw-1 downto 0);
           WrAddress:in std_logic_vector(lpm_widthadw-1 downto 0);
           RdAddress:in std_logic_vector(lpm_widthadr-1 downto 0);
           WrClock     : in std_logic := 0;
           WrClockEn   : in std_logic := 0;
           RdClock     : in std_logic := 0;
           RdClockEn   : in std_logic := 0;
           WE          : in std_logic;
           Reset       : in std_logic;
           Q       : out std_logic_vector(lpm_widthr-1 downto 0));
end component ;
```

```
begin
lpm_gen: L_RAMDPS
         generic map (
                      lpm_widthw    => 2,
                      lpm_widthadw  => 14,
                      lpm_numwordsw => 16384,
                      lpm_widthr    => 2,
                      lpm_widthadr  => 14,
        lpm_numwordsr => 16384,
                lpm_indata   => "REGISTERED",
                lpm_addressr_control => "REGISTERED",
                      lpm_init_file => "RAM_init",
        lpm_indata   => "UNREGISTERED",
                      )

         port map (   Data           => Data,
                      WrAddress      => WrAddress,
                      RdAddress      => RdAddress,
                      WrClock        => WrClock,
                      WrClockEn      => WrClockEn,
                      RdClock        => RdClock,
                      RdClockEn      => RdClockEn,
                      WE             => WE,
                      Reset          => Reset,
                      Q              => Q);
end struct;
```

Random Access Memory (LPM_RAM_DQ)

```
library IEEE;
use IEEE.std_logic_1164.all;
LIBRARY lc5kmx;
USE lc5kmx.components.all;

entity tramdq16kx2 is

    port (
           Data            : in std_logic_vector( 1 downto 0);
           Address         : in std_logic_vector( 13 downto 0);
           Clock           : in std_logic;
           ClockEn         : in std_logic;
           WE              : in std_logic;
           Reset           : in std_logic;
           Q               : out std_logic_vector(1 downto 0));
end tramdq16kx2 ;


architecture behave of tramdq16kx2 is

component L_RAMDQ
      generic (
              LPM_TYPE      : string := “LPM_RAM_DQ”;
              LPM_WIDTH     : positive := 1;
              LPM_WIDTHAD   : positive := 1;
              LPM_NUMWORDS  : positive := 2;
              LPM_INDATA    : string :=”REGISTERED”;
              LPM_OUTDATA   : string :=”UNREGISTERED”;
              LPM_ADDRESS_CONTROL   : string :=”REGISTERED”;
              LPM_INIT_FILE : string := “dummy”;
              LPM_HINT      : string :=”UNUSED”);
      port (
            Data    : in std_logic_vector(LPM_WIDTH-1 downto 0);
  Address : in std_logic_vector(LPM_WIDTHAD-1 downto 0);
Clock         : in std_logic := ‘0’;
            ClockEn       : in std_logic := ‘0’;
            WE            : in std_logic;
            Reset         : in std_logic;
            Q       : out std_logic_vector(LPM_WIDTH-1 downto 0));
end component ;

begin
U0: L_RAMDQ
         generic map (
                      LPM_WIDTH     => 2,
                      LPM_WIDTHAD   => 14,
                      LPM_NUMWORDS  => 16384,
                      LPM_ADDRESS_CONTROL => “UNREGISTERED”,
                      LPM_INIT_FILE => “RAM_init”,
                      LPM_OUTDATA   => “UNREGISTERED”)
         port map (
                      Data          => Data,
                      Address       => Address,
                      Clock         => Clock,
                      ClockEn       => ClockEn,
                      WE            => WE,
                      Reset         => Reset,
                      Q             => Q);
end behave;
```

First-In-First-Out Memory (LPM_FIFO_DC)

```
LIBRARY IEEE;
USE IEEE.std_logic_1164.all;
LIBRARY lc5kmx;
USE lc5kmx.components.all;

entity tfifo4kx4 is

    port (
           Data          : in std_logic_vector(3 downto 0);
           WrClock       : in std_logic;
           WrEn          : in std_logic;
           RdClock       : in std_logic;
           RdEn          : in std_logic;
           Reset         : in std_logic;
           RPReset       : in std_logic;
           Q             : out std_logic_vector(3 downto 0);
           Full          : Out std_logic;
           Empty         : Out std_logic;
           AlmostFull    : Out std_logic;
           AlmostEmpty   : Out std_logic);
end tfifo16kx1 ;


architecture struct of  tfifo4kx4  is

component L_FIFO
  generic (
 lpm_type      : string  := “LPM_FIFO_DC”;
 lpm_width     : integer := 1;
 lpm_widthu    : integer := 1;
 lpm_numwords  : integer := 2;
 lpm_amfull_flag: integer :=1;
 lpm_amempty_flag: integer :=1;
 lpm_hint      : string  := “UNUSED”);

  port (
 Data        :  in  std_logic_vector (lpm_width-1 downto 0);
 WrClock       :  in  std_logic;
 WrEn          :  in  std_logic;
 RdClock       :  in  std_logic;
 RdEn          :  in  std_logic;
 Reset         :  in  std_logic;
 RPReset       :  in  std_logic;
 Q           :  out std_logic_vector (lpm_width-1 downto 0);
 Full          :  out std_logic;
 Empty         :  out std_logic;
 AlmostFull    :  out std_logic;
 AlmostEmpty   :  out std_logic);
end component ;

begin
U0: L_FIFO
```

```
            generic map (LPM_WIDTH    => 4,
                         LPM_WIDTHU   => 12,
        LPM_AMFULL_FLAG => 1,
            LPM_AMEMPTY_FLAG => 1,
                         LPM_NUMWORDS => 4096)

            port map (   Data        => Data,
                         WrClock     => WrClock,
                         WrEn        => WrEn,
                         RdClock     => RdClock,
                         RdEn        => RdEn,
                         Reset       => Reset,
                         RPReset     => RPReset,
                         Q           => Q,
                         Full        => FULL,
                         Empty       => EMPTY,
                         AlmostFull  => AlmostFull,
                         AlmostEmpty => AlmostEmpty );
    end struct;
```

Content Addressable Memory (LPM_CAM)

```
library IEEE;
use IEEE.std_logic_1164.all;
LIBRARY lc5kmx;
USE lc5kmx.components.all;

entity tcam128x48 is

    port (
           Data         : in std_logic_vector(47 downto 0);
           WrAddress    : in std_logic_vector(6 downto 0);
           ClockEn      : in std_logic;
           Clock        : in std_logic;
           We           : in std_logic;
           EnMask       : in std_logic;
           WrMask       : in std_logic;
           WrDc         : in std_logic;
           Reset        : in std_logic;
           Address      : out std_logic_vector(6 downto 0);
           Match        : Out std_logic;
           MulMatch     : Out std_logic);
end tcam128x48 ;


architecture struct of  tcam128x48 is

component L_CAM
  generic (
    lpm_type               : string  := “LPM_CAM”;
    lpm_width              : integer := 1;
    lpm_numwords           : integer := 1;
    lpm_widthad            : integer := 1;
    lpm_init_file   : string := “dummy”;
    lpm_init_flag   : integer := 0;
    lpm_hint               : string  := “UNUSED”);
   port(
           Data      : in  std_logic_vector(lpm_width-1 downto 0);
           WrAddress     : in  std_logic_vector(6 downto 0);
           ClockEn       : in  std_logic;
           Clock         : in  std_logic;
           WE            : in  std_logic;
           EnMask        : in  std_logic;
           WrMask        : in  std_logic;
           WrDC          : in  std_logic;
           Reset         : in  std_logic;
           Address       : out std_logic_vector(6 downto 0);
           Match         : Out std_logic;
           MulMatch      : Out std_logic);
end component ;

begin
U0: L_CAM
         generic map (LPM_WIDTH    => 48,
                      LPM_WIDTHAD  => 7,
                      LPM_NUMWORDS => 128,
                     LPM_INIT_FILE => “CAM_init”,
                     LPM_INIT_FLAG => 1)

         port map (   Data        => Data,
                      WrAddress   => WrAddress,
                      ClockEn     => ClockEn,
                      Clock       => Clock,
                      WE          => We,
                      EnMask      => EnMask,
                      WrMask      => Wrmask,
                      WrDC        => WrDc,
                      Reset       => Reset,
                      Address     => Address,
                      Match       => Match,
                      MulMatch    => MulMatch);
end struct;
```

Read-Only Memory (LPM_ROM)

```
library IEEE;
use IEEE.std_logic_1164.all;
LIBRARY lc5kmx;
USE lc5kmx.components.all;

entity trom16kx2 is

    port (
           Address         : in std_logic_vector( 13 downto 0);
           OutClock        : in std_logic;
           OutClockEn      : in std_logic;
           Reset           : in std_logic;
           Q               : out std_logic_vector(1 downto 0));
end trom16kx2 ;


architecture struct of trom16kx2 is

component L_ROM
  generic (
    lpm_type                : string  := "LPM_ROM";
    lpm_width               : integer := 1;
    lpm_numwords            : integer := 2;
    lpm_widthad             : integer := 1;
    lpm_outdata             : string  := "UNREGISTERED";
    lpm_address_control     : string  := "REGISTERED";
          lpm_init_file           : string := "dummy";
    lpm_hint                : string  := "UNUSED");

  port (
          Address : in  std_logic_vector (lpm_widthad-1 downto 0);
          OutClock      : in  std_logic;
          OutClockEn    : in  std_logic;
          Reset         : in  std_logic;
          Q       : out std_logic_vector (lpm_width-1 downto 0));
end component ;

attribute syn_black_box: boolean;
attribute syn_black_box of L_ROM: component is true;

begin
U0: L_ROM
         generic map (
                      LPM_WIDTH      => 2,
                      LPM_WIDTHAD    => 14,
                      LPM_NUMWORDS   => 16384,
                      LPM_OUTDATA    => "REGISTERED",
                      LPM_ADDRESS_CONTROL => "REGISTERED",
                      LPM_INIT_FILE => "ROM_init")
         port map (
                      Address        => Address,
                      OutClock       => OutClock,
                      OutClockEn    => OutClockEn,
                      Reset         => Reset,
                      Q             => Q);
end struct;
```

Appendix C. Initialization File Usage Guide

Introduction

The initialization file is a text file primarily used for preloading user-specified data into the memory array. This file is mainly used for configuring ROM, but is optional for dual-port, pseudo dual port and single port SRAM, FIFO and CAM modes.

Figure 6 is an example of an initialization file.

Figure 6. Sample Initialization File (20x32)

```
11111111111111110000000000010001
11111111111111100000000000010000
11111111111111011111111111111111
11111111111110111111111111111111
11111111111110101111111111111111
11111111111110011111111111111111
11111111111110001111111111111111
11111111111101111111111111110001
11111111111101101111111111110001
11111111111101001111111111110001
11111111111100110000000000100100
11111111111100100000000000100100
11111111111100010000000000100100
11111111111100000000000000100100
11111111111011110000000000100100
11111111111011010000000000000110
00000000000100010000000000000110
00000000000100000000000000000110
00000000000011110000000000000110
00000000000011100000000000000110
```

Address locations are numbered sequentially from 0 to lpm_numwords -1. The first or topmost entry corresponds to the initialization data at address 0, and the last entry to address LPM_NUMWORDS-1. Bits are read right to left, starting from the LSB to MSB. In the initialization file shown above for example, the top right-most bit correlates to bit 0 of Address 0, while the bottom left-most bit correlates to bit 31 of Address 19. Initialization data can only be entered in binary format.

Data depth and width are defined by the size of the user instantiated memory. The number of rows corresponds to the number of address locations in the array (depth), and the number of columns matches the data width. Inputs are specified in binary format, and each bit can either be a 1, 0, X (don't care) or a U (undefined). Note that 'X' and 'U' inputs only apply for CAM. Excess bits and/or undefined characters in the initialization file are flagged as errors during compilation.

An initialization file can have any name, but it should match the filename specified in the HDL source file. The file cannot have a trailing three-character extension or file type. Initialization filenames with trailing extensions are not valid, and therefore flagged as errors during compilation.

To preload memory using an initialization file, simply define the 'lpm_init_file' parameter in your top-level HDL source file and specify the initialization file name. Figure 7 shows an example of a VHDL ROM module using an initialization file. In this case, the 'lpm_init_file: string: = "ROM_init";' declaration was added into the component instantiation. The same concept applies for Verilog designs. The example shown in Figure 8 has been modified to include the 'defparam U0.lpm_init_file="init1";' declaration.

Figure 7. VHDL ROM instantiation with lpm_init_file defined

```
entity trom512x121 is

    port (
           Address          : in std_logic_vector( 8 downto 0);
           OutClock         : in std_logic;
           OutClockEn       : in std_logic;
           Reset            : in std_logic;
           Q                : out std_logic_vector(120 downto 0));
end trom512x121 ;


architecture struct of trom512x121 is

component L_ROM
  generic (
    lpm_type                : string  := "LPM_ROM";
    lpm_width               : integer := 1;
    lpm_numwords            : integer := 1;
    lpm_widthad             : integer := 1;
    lpm_outdata             : string  := "UNREGISTERED";
    lpm_address_control     : string  := "REGISTERED";
    lpm_init_file           : string  := "ROM_init";
    lpm_hint                : string  := "UNUSED");
  port (
            Address        : in  std_logic_vector (lpm_widthad-1 downto 0);
            OutClock       : in  std_logic;
            OutClockEn     : in  std_logic;
            Reset          : in  std_logic;
            Q              : out std_logic_vector (lpm_width-1 downto 0));
end component ;

begin
U0: L_ROM
         generic map (
                      LPM_WIDTH      => 121,
                      LPM_WIDTHAD    => 9,
                      LPM_NUMWORDS   => 512,
                      LPM_OUTDATA    => "REGISTERED",
                      LPM_ADDRESS_CONTROL => "REGISTERED",
                      LPM_INIT_FILE => "ROM_init")
         port map (
                      Address        => Address,
                      OutClock       => OutClock,
                      OutClockEn     => OutClockEn,
                      Reset          => Reset,
                      Q              => Q);
end struct;
```

Figure 8. Verilog ROM instantiation with lpm_init_file defined

```
module test_rom512x121(
           Q,
           Address,
           OutClock,
           OutClockEn,
           Reset);
output [120:0] Q;
input [8:0] Address;
input OutClock,OutClockEn,Reset;

L_ROM          U0(.Q(Q),.Address(Address),.OutClock(OutClock),.OutClockEn(Out-
ClockEn),.Reset(Reset));

defparam U0.lpm_width=121;
defparam U0.lpm_widthad=9;
defparam U0.lpm_numwords=512;
defparam U0.lpm_outdata="UNREGISTERED";
defparam U0.lpm_init_file="init1";

endmodule

module L_ROM(
           Q,
           Address,
           OutClock,
           OutClockEn,
           Reset);

   parameter lpm_type = "LPM_ROM";
   parameter lpm_width     = 1;
   parameter lpm_numwords  = 1;
   parameter lpm_widthad   = 1;
   parameter lpm_outdata    = "UNREGISTERED";
   parameter lpm_address_control  = "REGISTERED";
   parameter lpm_hint = "UNUSED";
   parameter lpm_init_file="dummy";

           output [lpm_width-1:0] Q;
           input [lpm_widthad-1:0] Address;
           input OutClock,OutClockEn,Reset;

endmodule // lpm_rom
```

For additional examples, refer to Appendices A and B.

Common Mistakes and Error Messages

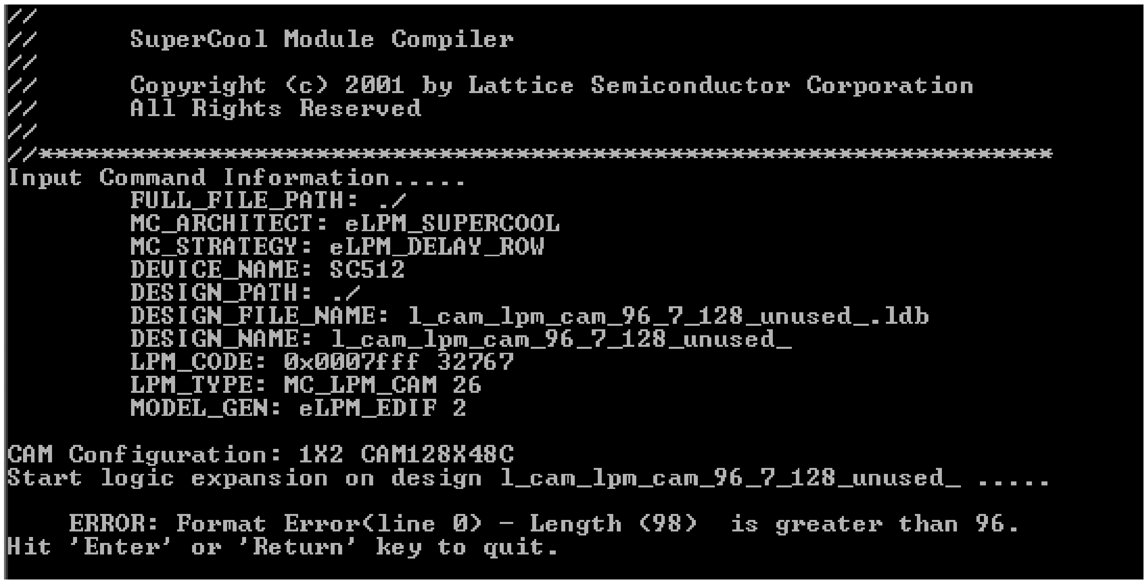
Most initialization file issues are related to the memory file size or the filename format. The most common errors in generating initialization files are:

1. Specifying an incorrect depth (number of rows) or width (number of columns)
2. Using invalid filenames (i.e. CAM_init.dat)
3. Using invalid characters (i.e. Use of any other character aside from a ‘1’, ‘0’ or ‘X’)

Below are some sample error messages that can help diagnose an initialization file problem. For reference, the following example uses a 96X128 CAM.

Figure 9 shows the error generated when the initialization file data width exceeds the predefined CAM width. Because the CAM configuration width is set at 96, and the initialization file data has 98 bits, an error is generated by the compiler.

Figure 9. Error: Data width is greater than the defined memory width



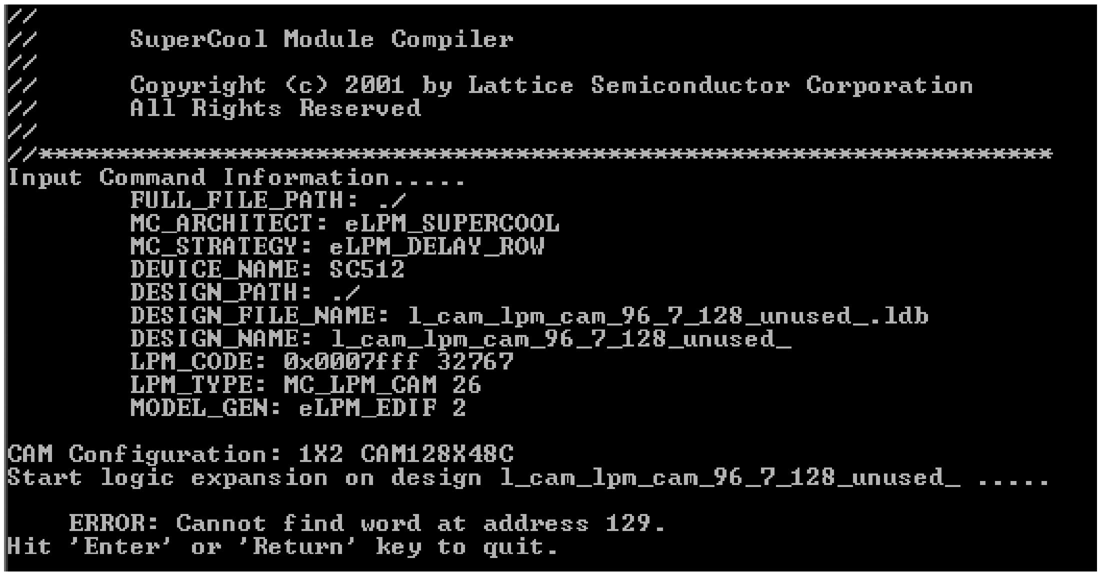
```
//
//      SuperCool Module Compiler
//
//      Copyright (c) 2001 by Lattice Semiconductor Corporation
//      All Rights Reserved
//
//*******************************************************************
Input Command Information.....
        FULL_FILE_PATH: ./
        MC_ARCHITECT: eLPM_SUPERCOOL
        MC_STRATEGY: eLPM_DELAY_ROW
        DEVICE_NAME: SC512
        DESIGN_PATH: ./
        DESIGN_FILE_NAME: l_cam_lpm_cam_96_7_128_unused_.ldb
        DESIGN_NAME: l_cam_lpm_cam_96_7_128_unused_
        LPM_CODE: 0x0007fff 32767
        LPM_TYPE: MC_LPM_CAM 26
        MODEL_GEN: eLPM_EDIF 2

CAM Configuration: 1X2 CAM128X48C
Start logic expansion on design l_cam_lpm_cam_96_7_128_unused_ .....

    ERROR: Format Error(line 0) - Length (98)  is greater than 96.
Hit 'Enter' or 'Return' key to quit.
```

Figure 10 shows an error generated by the compiler when the initialization file data exceeds the total word count in the CAM array. By definition, the CAM can only hold 128 words. Since the initialization file has 129 words (rows), the compiler automatically errors out.

Figure 10. Error: Data depth is greater than the defined number of words



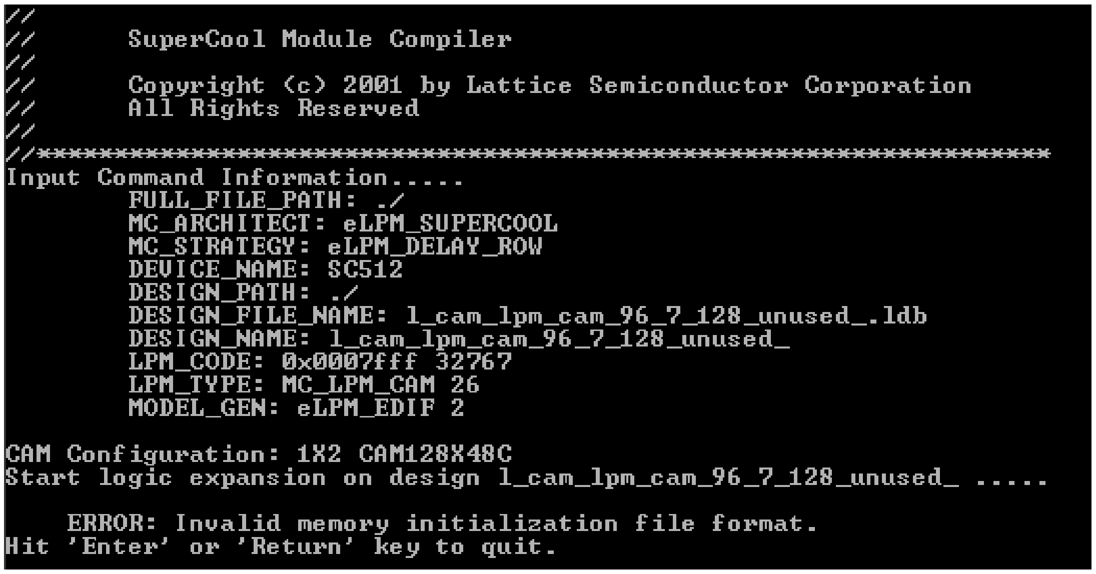
```
//
//      SuperCool Module Compiler
//
//      Copyright (c) 2001 by Lattice Semiconductor Corporation
//      All Rights Reserved
//
//*******************************************************************
Input Command Information.....
        FULL_FILE_PATH: ./
        MC_ARCHITECT: eLPM_SUPERCOOL
        MC_STRATEGY: eLPM_DELAY_ROW
        DEVICE_NAME: SC512
        DESIGN_PATH: ./
        DESIGN_FILE_NAME: l_cam_lpm_cam_96_7_128_unused_.ldb
        DESIGN_NAME: l_cam_lpm_cam_96_7_128_unused_
        LPM_CODE: 0x0007fff 32767
        LPM_TYPE: MC_LPM_CAM 26
        MODEL_GEN: eLPM_EDIF 2

CAM Configuration: 1X2 CAM128X48C
Start logic expansion on design l_cam_lpm_cam_96_7_128_unused_ .....

    ERROR: Cannot find word at address 129.
Hit 'Enter' or 'Return' key to quit.
```

Figure 11 illustrates the error generated when invalid characters are detected in the initialization file. In this case, invalid ASCII characters are inserted into the file (not shown) to show the error message.

Figure 11. Error: Invalid characters are used in the initialization file



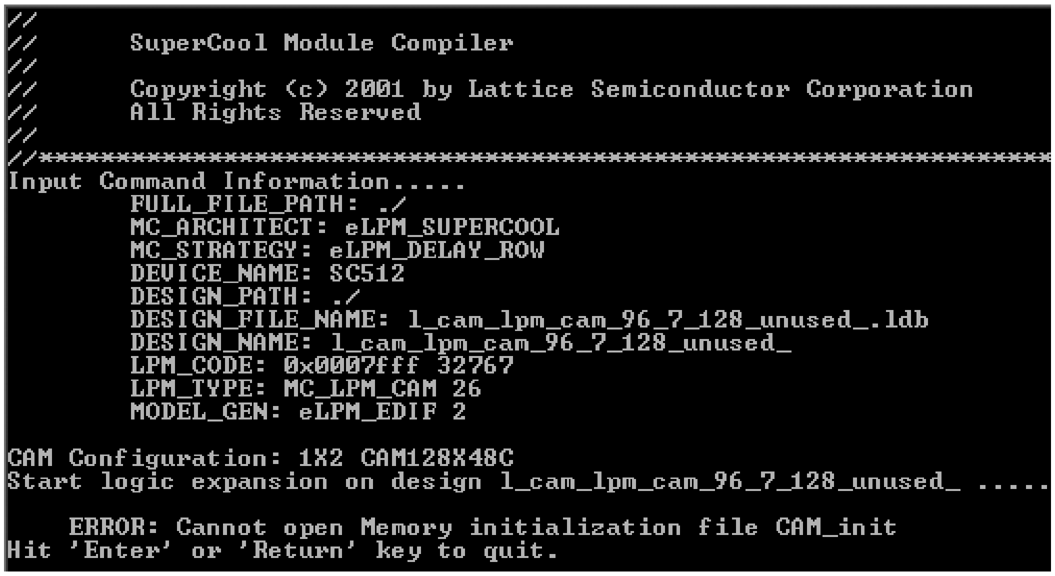
```
//
//      SuperCool Module Compiler
//
//      Copyright (c) 2001 by Lattice Semiconductor Corporation
//      All Rights Reserved
//
//*******************************************************************
Input Command Information.....
        FULL_FILE_PATH: ./
        MC_ARCHITECT: eLPM_SUPERCOOL
        MC_STRATEGY: eLPM_DELAY_ROW
        DEVICE_NAME: SC512
        DESIGN_PATH: ./
        DESIGN_FILE_NAME: l_cam_lpm_cam_96_7_128_unused_.ldb
        DESIGN_NAME: l_cam_lpm_cam_96_7_128_unused_
        LPM_CODE: 0x0007fff 32767
        LPM_TYPE: MC_LPM_CAM 26
        MODEL_GEN: eLPM_EDIF 2

CAM Configuration: 1X2 CAM128X48C
Start logic expansion on design l_cam_lpm_cam_96_7_128_unused_ .....

    ERROR: Invalid memory initialization file format.
Hit 'Enter' or 'Return' key to quit.
```

The compiler also flags incorrectly named initialization files. Figure 12 shows an example where the initialization file is associated with a three-character file extension. In this case the lpm_init_file = “CAM_init.dat” definition is included in the source file. Upon compilation, the compiler is unable to resolve the ‘.DAT’ file extension and errors out.

Figure 12. Error: Initialization file is associated with a three-character extension or file



```
//
//      SuperCool Module Compiler
//
//      Copyright (c) 2001 by Lattice Semiconductor Corporation
//      All Rights Reserved
//
//*************************************************************
Input Command Information.....
        FULL_FILE_PATH: ./
        MC_ARCHITECT: eLPM_SUPERCOOL
        MC_STRATEGY: eLPM_DELAY_ROW
        DEVICE_NAME: SC512
        DESIGN_PATH: ./
        DESIGN_FILE_NAME: l_cam_lpm_cam_96_7_128_unused_.ldb
        DESIGN_NAME: l_cam_lpm_cam_96_7_128_unused_
        LPM_CODE: 0x0007fff 32767
        LPM_TYPE: MC_LPM_CAM 26
        MODEL_GEN: eLPM_EDIF 2

CAM Configuration: 1X2 CAM128X48C
Start logic expansion on design l_cam_lpm_cam_96_7_128_unused_ .....

    ERROR: Cannot open Memory initialization file CAM_init
Hit 'Enter' or 'Return' key to quit.
```

Technical Support Assistance

Hotline:	1-800-LATTICE (North America)
	+1-408-826-6002 (Outside North America)
e-mail:	techsupport@latticesemi.com
Internet:	www.latticesemi.com