Fusion SRAM/FIFO Blocks

Introduction

As design complexity grows, greater demands are placed upon an FPGA's embedded memory. Actel Fusion devices provide the flexibility of true dual-port as well as two-port SRAM blocks. The embedded memory, along with built-in, dedicated FIFO control logic, can be used to create cascading RAM blocks and FIFOs without using additional logic gates.

Architecture

The Fusion devices feature up to 504 kbits of RAM in 4,608-bit blocks (Figure 1). The total embedded SRAM memory for each device can be found in the *Fusion Family of Mixed-Signal FPGAs* datasheet. These memory blocks are arranged along the top and bottom of the device to allow better access from the core and I/O (in some device it is only available at the bottom of the device). Every RAM block has a flexible, hardwired embedded FIFO controller, enabling the user to implement efficient FIFOs without sacrificing user gates.

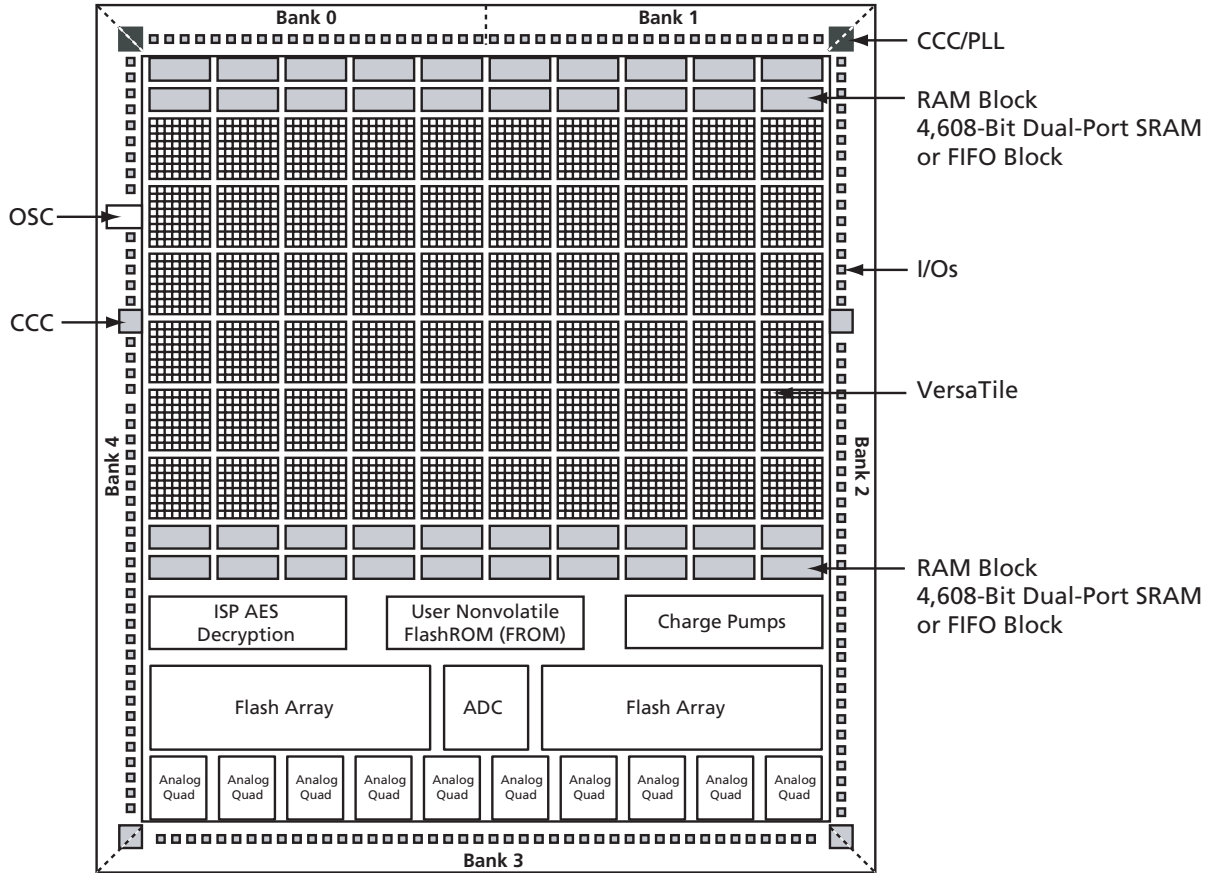


Figure 1 • **Fusion Device Architecture Overview (AFS600)**

Memory blocks can be configured with many different aspect ratios, but are generically supported in the macro libraries as one of two memory elements: RAM4K9 or RAM512X18. The RAM4K9 is configured as a true dual-port memory block, and the RAM512X18 is configured as a two-port memory block. Dual-port memory allows the RAM to both read from and write to either port independently. Two-port memory allows the RAM to read from one port and write to the other using a common clock or independent read and write clocks. If needed, the RAM4K9 blocks can be configured as two-port memory blocks. The memory block can be configured as FIFO by combining the basic memory block with dedicated FIFO controller logic. The FIFO macro is named FIFO4KX18.

Clocks for the RAM blocks can be driven by the VersaNet (global resources) or by regular nets. When using local clock segments, the clock segment region that encompasses the RAM blocks can drive the RAMs. In the dual-port configuration (RAM4K9), each memory block port can be driven by either rising-edge or falling-edge clocks. Each port can be driven by clocks with different edges. While only a rising-edge clock can drive the physical block itself, the Actel Designer software will automatically bubble push the inversion to properly implement the falling edge trigger for the RAM block.

Memory Configuration

Variable aspect ratio and cascading allow users to configure the memory in the width and depth required. Fusion RAM can be configured as 1, 2, 4, 9, or 18 bits wide. By cascading the memory blocks, any multiple of those widths can be created. The RAM memory blocks can be from 256 to 4,096 bits deep, depending on the aspect ratio, and the blocks can also be cascaded to create deeper areas. Refer to the aspect ratios available for each macro cell in the "SRAM Features" section on page 4. The largest continuous configurable memory area is equal to half the total memory available on the device because the RAM is separated into two groups, one on each side of the device.

The Actel SmartGen core generator will automatically configure and cascade both RAM and FIFO blocks. Cascading is accomplished using dedicated memory logic and does not consume user gates for depths up to 4,096 bits deep and widths up to 18, depending on the configuration. Deeper memory will utilize some user gates to multiplex the outputs.

Generated RAM and FIFO macros can be created as either structural VHDL or Verilog for easy instantiation into the design. Users of Actel Libero® Integrated Design Environment (IDE) can create a symbol for the macro and incorporate it into a design schematic.

Table 2 on page 3 shows the number of memory blocks required for each of the supported depth and width memory configurations, and for each depth and width combination. For example, a 256-bit deep by 32-bit wide two-port RAM would consist of two 256x18 RAM blocks. The first 18 bits would be stored in the first RAM block, and the remaining 14 bits would be implemented in the other 256x18 RAM block. This second RAM block would have 4 bits of unused storage. Similarly, a dual port memory block that is 8,192 bits deep and 8 bits wide would be implemented using 16 memory blocks. The dual-port memory would be configured in a 4,096x1 aspect ratio. These blocks would then be cascaded 2 deep to achieve 8,192 bits of depth and 8 wide to achieve the 8 bits of width.

Table 1 shows the maximum potential width and depth configuration for each Fusion device.

Table 1 • **Memory Availability per Device**

Device	RAM Blocks	Maximum Potential Width[1]		Maximum Potential Depth[2]	
		Depth	Width	Depth	Width
AFS090	6	256	108 (6x18)	24,576 (4,094x6)	1
AFS250	8	256	144 (8x18)	32,768 (4,094x8)	1
AFS600	24	256	432 (24x18)	98,304 (4,096x24)	1
AFS1500	60	256	1,080 (60x18)	245,760 (4,096x60)	1

Notes:

1. *Maximum potential width uses the two-port configuration.*
2. *Maximum potential depth uses the dual-port configuration.*

Table 2 • **RAM and FIFO Memory Block Consumption**

		Depth									
		256		512	1,024	2,048	4,096	8,192	16,384	32,768	65,536
		Two-Port	Dual-Port	Dual-Port	Dual-Port	Dual-Port	Dual-Port	Dual-Port	Dual-Port	Dual-Port	Dual-Port
Width	1	Number Block	1	1	1	1	1	1	2	4	8	16 x 1
		Configuration	Any	Any	Any	1,024 x 4	2,048 x 2	4,096 x 1	2 x (4,096 x 1) Cascade Deep	4 x (4,096 x 1) Cascade Deep	8 x (4,096 x 1) Cascade Deep	16 x (4,096 x 1) Cascade Deep
	2	Number Block	1	1	1	1	1	2	4	8	16	32
		Configuration	Any	Any	Any	1,024x4	2,048 x 2	2 x (4,096 x 1) Cascaded Wide	4 x (4,096 x 1) Cascaded 2 Deep and 2 Wide	8 x (4,096 x 1) Cascaded 4 Deep and 2 Wide	16 x (4,096 x 1) Cascaded 8 Deep and 2 Wide	32 x (4,096 x 1) Cascaded 16 Deep and 2 Wide
	4	Number Block	1	1	1	1	2	4	8	16	32	64
		Configuration	Any	Any	Any	1,024 x 4	2 x (2,048 x 2) Cascaded Wide	4 x (4,096 x 1) Cascaded Wide	4 x (4,096 x 1) Cascaded 2 Deep and 4 Wide	16 x (4,096 x 1) Cascaded 4 Deep and 4 Wide	32 x (4,096 x 1) Cascaded 8 Deep and 4 Wide	64 x (4,096 x 1) Cascaded 16 Deep and 4 Wide
	8	Number Block	1	1	1	2	4	8	16	32	64	
		Configuration	Any	Any	Any	2 x (1,024 x 4) Cascaded Wide	4 x (2,048 x 2) Cascaded Wide	8 x (4,096 x 1) Cascaded Wide	16 x (4,096 x 1) Cascaded 2 Deep and 8 Wide	32 x (4,096 x 1) Cascaded 4 Deep and 8 Wide	64 x (4,096 x 1) Cascaded 8 Deep and 8 Wide	
	9	Number Block	1	1	1	2	4	8	16	32		
		Configuration	Any	Any	Any	2 x (512 x 9) Cascaded Deep	4 x (512 x 9) Cascaded Deep	8 x (512 x 9) Cascaded Deep	16 x (512 x 9) Cascaded Deep	32 x (512 x 9) Cascaded Deep		
	16	Number Block	1	1	1	4	8	16	32	64		
		Configuration	25 6x 18	256 x 18	256 x 18	4 x (1,024 x 4) Cascaded Wide	8 x (2,048 x 2) Cascaded Wide	16 x (4,096 x 1) Cascaded Wide	32 x (4,096 x 1) Cascaded 2 Deep and 16 Wide	32 x (4,096 x 1) Cascaded 4 Deep and 16 Wide		
	18	Number Block	1	2	2	4	8	18	32			
		Configuration	256 x 8	2 x (512 x 9) Cascaded Wide	2 x (512 x 9) Cascaded Wide	4 x (512 x 9) Cascaded 2 Deep and 2 Wide	8 x (512 x 9) Cascaded 4 Deep and 2 Wide	16 x (512 x 9) Cascaded 8 Deep and 2 Wide	16 x (512 x 9) Cascaded 16 Deep and 2 Wide			
	32	Number Block	2	4	4	8	16	32	64			
		Configuration	2 x (256 x 18) Cascaded Wide	4 x (512 x 9) Cascaded Wide	4 x (512 x 9) Cascaded Wide	8 x (1,024 x 4) Cascaded Wide	16 x (2,048 x 2) Cascaded Wide	32 x (4,096 x 1) Cascaded Wide	64 x (4,096 x 1) Cascaded 2 Deep and 32 Wide			
	36	Number Block	2	4	4	8	16	32				
		Configuration	2 x (256 x 18) Cascaded Wide	4 x (512 x 9) Cascaded Wide	4 x (512 x 9) Cascaded Wide	4 x (512 x 9) Cascaded 2 Deep and 4 Wide	16 x (512 x 9) Cascaded 4 Deep and 4 Wide	16 x (512 x 9) Cascaded 8 Deep and 4 Wide				
	64	Number Block	4	8	8	16	32	64				
		Configuration	4 x (256 x 18) Cascaded Wide	8 x (512 x 9) Cascaded Wide	8 x (512 x 9) Cascaded Wide	16 x (1,024 x 4) Cascaded Wide	32 x (2,048 x 2) Cascaded Wide	64 x (4,096 x 1) Cascaded Wide				
	72	Number Block	4	8	8	16	32					
		Configuration	4 x (256 x 18) Cascaded Wide	8 x (512 x 9) Cascaded Wide	8 x (512 x 9) Cascaded Wide	16 x (512 x 9) Cascaded Wide	16 x (512 x 9) Cascaded 4 Deep and 8 Wide					

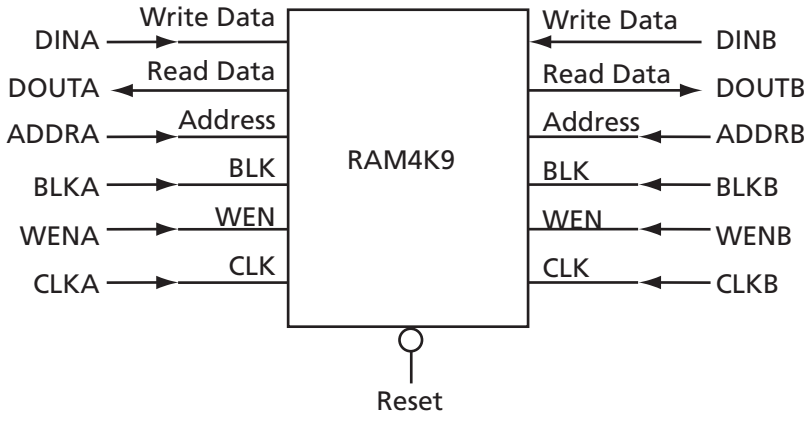
Note: *Memory configurations represented by grayed cells are not supported.*

SRAM Features

RAM4K9 Macro

The RAM4K9 is the dual-port configuration of the RAM block (Figure 2). The RAM4K9 nomenclature refers to both the deepest possible configuration and the widest possible configuration that the dual-port RAM block can assume, and does not denote a possible memory aspect ratio. The RAM block can be configured to the following aspect ratios: 4,096x1, 2,048x2, 1,024x4, and 512x9. The RAM4K9 is fully synchronous and has the following features:

- Two ports that allow fully independent reads and writes at different frequencies
- Selectable pipelined or nonpipelined read
- Active low block enables for each port
- Toggle control between read and write mode for each port
- Active low asynchronous reset
- Pass-through write data or hold existing data on output. In pass-though mode, the data written to the write port will immediately appear on the read port.
- Designer software will automatically facilitate falling edge clocks by bubble pushing the inversion to previous stages.



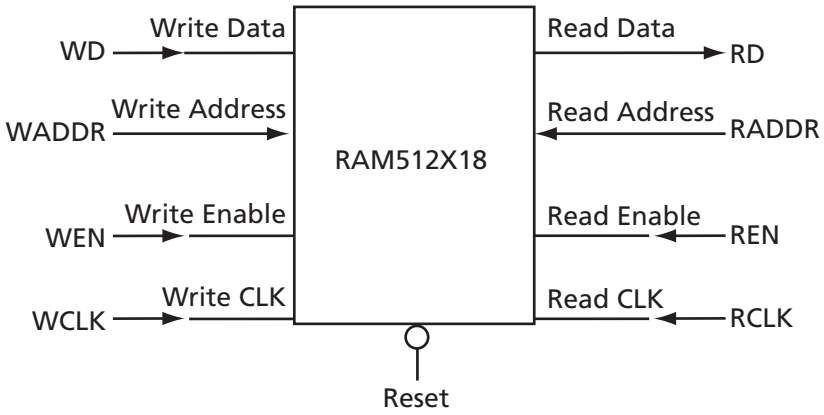
Note: *For timing diagrams of the RAM signals, refer to the* Fusion Family of Mixed-Signals FPGAs *datasheet.*

Figure 2 • **RAM4K9 Configuration**

RAM512X18 Macro

The RAM512X18 is the two-port configuration of the same RAM block (Figure 3 on page 5). Like the RAM4K9 nomenclature, the RAM512X18 nomenclature refers to both the deepest possible configuration and the widest possible configuration that the two-port RAM block can assume. In two-port mode, the RAM block can be configured to either the 512x9 aspect ratio or the 256x18 aspect ratio. The RAM512x18 is also fully synchronous and has the following features:

- Dedicated read and write ports
- Active low read and write enables
- Selectable pipelined or nonpipelined read
- Active low asynchronous reset
- Designer software will automatically facilitate falling-edge clocks by bubble pushing the inversion to previous stages.



Note: *For timing diagrams of the RAM signals, refer to the* Fusion Family of Mixed-Signals FPGAs *datasheet.*

Figure 3 • Fusion **Two-Port RAM Block Diagram**

FIFO Features

The FIFO4KX18 macro is created by merging the RAM block with dedicated FIFO logic (Figure 4). Since the FIFO logic can only be used in conjunction with the memory block, there is no separate FIFO controller macro. As with the RAM blocks, the FIFO4KX18 nomenclature does not refer to a possible aspect ratio, but rather to the deepest possible data depth and the widest possible data width. The FIFO4KX18 can be configured into the following aspect ratios: 4,096x1, 2,048x2, 1,024x4, 512x9, and 256x18. In addition to being fully synchronous, the FIFO4KX18 also has the following features:

- Four FIFO flags: Empty, Full, Almost-Empty, and Almost-Full
- EMPTY flag is synchronized to the read clock
- FULL flag is synchronized to the write clock
- Both Almost-Empty and Almost-Full flags have programmable thresholds
- Active low asynchronous reset
- Active low block enable
- Active low write enable
- Active high read enable
- Ability to configure the FIFO to either stop counting after the empty or full states are reached or to allow the FIFO counters to continue
- Designer software will automatically facilitate falling-edge clocks by bubble pushing the inversion to previous stages.

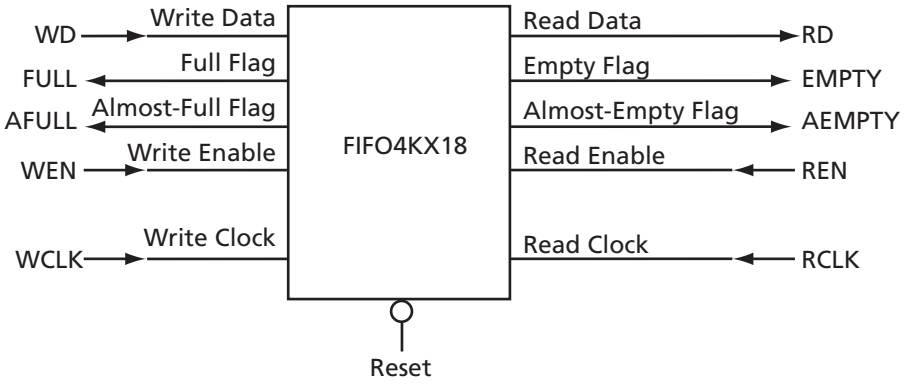


Figure 4 • Fusion **FIFO Block Diagram**

The FIFOs maintain a separate read and write address. Whenever the difference between the write address and the read address is greater than or equal to the almost-full value (AFVAL), the Almost-Full flag is asserted. Similarly, the Almost-Empty flag is asserted whenever the difference between the write address and read address is less than or equal to the almost-empty value (AEVAL).

Due to synchronization between the read and write clocks, the Empty flag will de-assert after the second read clock edge from the point that the write enable asserts. However, since the Empty flag is synchronized to the read clock, it will assert after the read clock reads the last data in the FIFO. Also, since the Full flag is dependent on the actual hardware configuration, it will assert when the actual physical implementation of the FIFO is full.

For example, when a user configures a 128x18 FIFO, the actual physical implementation will be a 256x18 FIFO element. Since the actual implementation is 256x18, the Full flag will not trigger until the 256x18 FIFO is full, even though a 128x18 FIFO was requested. For this example, the Almost-Full flag can be used instead of the Full flag to signal when the 128th data word is reached.

In order to accommodate different aspect ratios, the almost-full and almost-empty values are expressed in terms of data bits instead of data words. SmartGen translates the user's input, expressed in data words, into data bits internally. SmartGen will allow the user to select the thresholds for the Almost-Empty and Almost-Full flags, in terms of either the read data words or the write data words, and make the appropriate conversions for each flag.

After the empty or full states are reached, the FIFO can be configured so the FIFO counters either stop or continue counting. For timing diagrams of the FIFO signals, refer to the *Fusion Family of Mixed-Signal FPGAs* datasheet.

Initializing the Fusion RAM/FIFO

The SRAM blocks can be initialized with data to use it as a lookup table (LUT). Data initialization can be accomplished either by loading the data through the design logic or through the UJTAG interface. The UJTAG macro is used to allow access from the JTAG port to the internal logic in the device. By sending the appropriate initialization string to the JTAG Test Access Port (TAP) Controller, the designer can put the JTAG circuitry into a mode which allows the user to shift data into the array logic through the JTAG port using the UJTAG macro.

A user interface is required to receive the user command, initialization data, and clock from the UJTAG macro. The interface must synchronize and load the data into the correct RAM block of the design. The main outputs of the user interface block are the following:

- Memory block chip select: Selects a memory block for initialization. The chip selects signals for each memory block that can be generated from different user-defined pockets or simple logic such as a ring counter (see below).
- Memory block write address: Identifies the address of the memory cell that needs to be initialized.
- Memory block write data: The interface block receives the data serially from the UTDI port of the UJTAG macro and loads it in parallel into the write data ports of the memory blocks.
- Memory block write clock: Drives the WCLK of the memory block and synchronizes the write data, write address, and chip-select signals.

Figure 5 shows the user interface between UJTAG and the memory blocks.

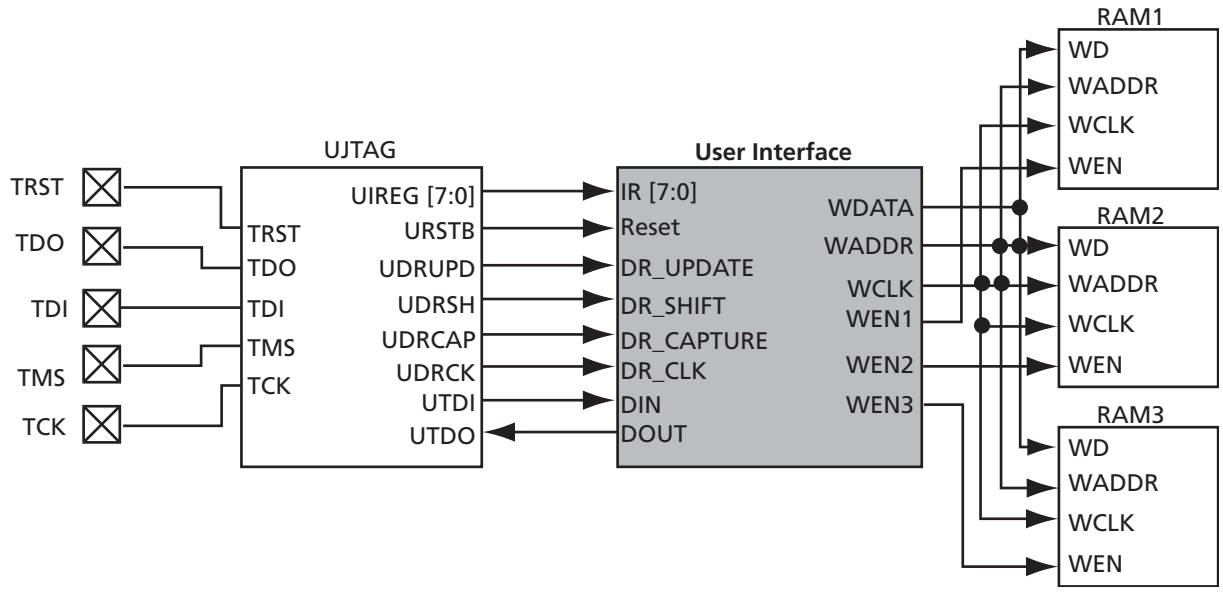


Figure 5 • **Interfacing TAP Ports and SRAM Blocks**

An important component of the interface between the UJTAG macro and the RAM blocks is a serial-in/parallel-out shift register. The width of the shift register should equal the data width of the RAM blocks. The RAM data arrives serially from the UTDI output of the UJTAG macro. The data must be shifted into a shift register clocked by the JTAG clock (provided at the UDRCK output of the UJTAG macro).

Then, after the shift register is fully loaded, the data must be transferred to the write data port of the RAM block. To synchronize the loading of the write data with the write address and write clock, the output of the shift register can be pipelined before driving the RAM block.

The write address can be generated in different ways. It can be imported through the TAP using a different instruction opcode and another shift register, or generated internally using a simple counter. Using a counter to generate the address bits and sweep through the address range of the RAM blocks is recommended, since it reduces the complexity of the user interface block and the board-level JTAG driver.

Moreover, using an internal counter for address generation speeds up the initialization procedure, since the user only needs to import the data through the JTAG port.

The designer may use different methods to select among the multiple RAM blocks. Using counters along with demultiplexers is one approach to set the write enable signals. Basically, the number of RAM blocks needing initialization determines the most efficient approach. For example, if all the blocks are initialized with the same data, one enable signal is enough to activate the write procedure for all of them at the same time. Another alternative is to use different opcodes to initialize each memory block. For a small number of RAM blocks, using counters is an optimal choice. For example, a ring counter can be used to select among multiple RAM blocks. The clock driver of this counter needs to be controlled by the address generation process.

Once the addressing of one block is finished, a clock pulse is sent to the (ring) counter to select the next memory block.

Figure 6 illustrates a simple block diagram of an interface block between UJTAG and RAM blocks.

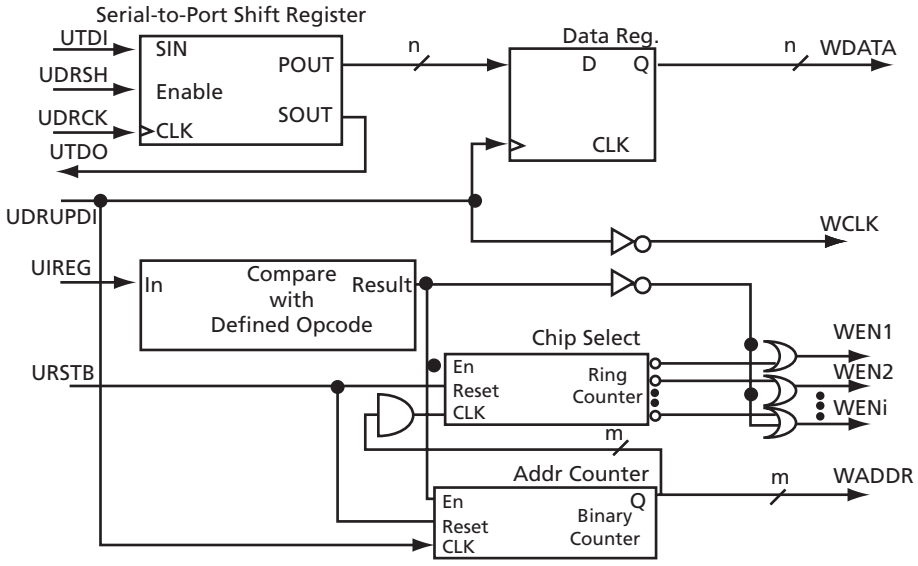


Figure 6 • **Block Diagram of a Sample User Interface**

In the circuit shown in Figure 6, the shift register is enabled by the UDRSH output of the UJTAG macro. The counters and chip-select outputs are controlled by the value of the TAP Instruction Register. The comparison block compares the UIREG value with the "start initialization" opcode value (defined by user). If the result is true, the counters start to generate addresses and activate the WEN inputs of appropriate RAM blocks.

The UDRUPD output of the UJTAG macro, also shown in Figure 6, is used for generating the write clock (WCLK) and synchronizing the data register and address counter with WCLK. UDRUPD is high when the TAP Controller is in the Data Register Update state, which is an indication of completing the loading of one data word. Once the TAP Controller goes into the Data Register Update state, the UDRUPD output of the UJTAG macro goes high. Therefore, the pipeline register and the address counter place the proper data and address on the outputs of the interface block. Meanwhile, WCLK is defined as the inverted UDRUPD. This will provide enough time (equal to the UDRUP high time) for the data and address to be placed at the proper ports of the RAM block before the rising edge of WCLK. The inverter is not required if the RAM blocks are clocked at the falling edge of the write clock. An example of this is described in the "Example of RAM Initialization" section.

Example of RAM Initialization

This section of the document presents a sample design in which a 4x4 RAM block is being initialized through the JTAG port. A test feature has been implemented in the design to read back the contents of the RAM after initialization to verify the procedure.

The interface block of this example performs two major functions: initialization of the RAM block and running a test procedure to read back the contents. The clock output of the interface is either the write clock (for initialization) or the read clock (for reading back the contents). The Verilog code for the interface block is included in the Appendix on page 14.

For simulation purposes, users can declare the input ports of the UJTAG macro for easier assignment in the testbench. However, the UJTAG input ports should not be declared on the top level during synthesis. If the input ports of the UJTAG are declared during synthesis, the synthesis tool will instantiate input buffers on these ports. The input buffers on the ports will cause Compile to fail in Designer.

Figure 7 on page 9 shows the simulation results for the initialization step of the example design.

The CLK_OUT signal, which is the clock output of the interface block, is the inverted DR_UPDATE output of the UJTAG macro. It is clear that it gives sufficient time (while the TAP Controller is in the Data Register Update state) for the write address and data to become stable before loading them into the RAM block.

Figure 8 presents the test procedure of the example. The data read back from the memory block matches the written data, thus verifying the design functionality.

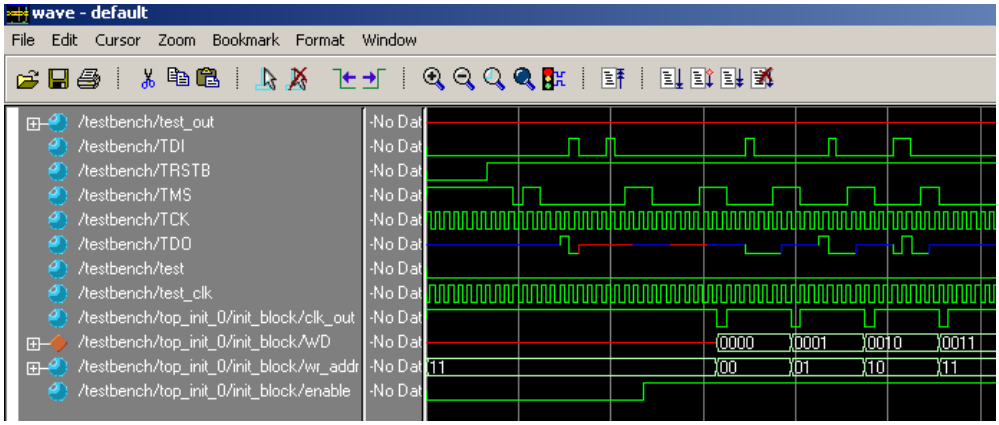


Figure 7 • **Simulation of Initialization Step**

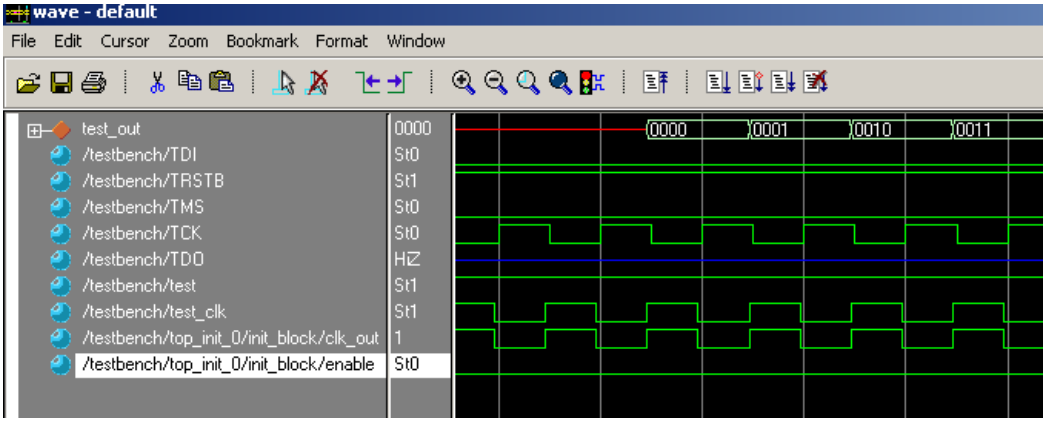


Figure 8 • **Simulation of the Test Procedure of the Example**

The ROM emulation application is based on RAM block initialization. If the user's main design has access only to the read ports of the RAM block (RADDR, RD, RCLK, and REN), and the contents of the RAM are already initialized through the TAP, then the memory blocks will emulate ROM functionality for the core design. In this case, the write ports of the RAM blocks are accessed only by the user interface block, and the interface is activated only by the TAP Instruction Register contents.

Users should note that the contents of the RAM blocks are lost in the absence of applied power. However, the 1 kbit of Flash memory, FROM, in Fusion can be used to retain data after power is removed from the device. Refer to the *Fusion FlashROM (FROM)* application note for more information.

Software Support

The SmartGen core generator is the easiest way to select and configure the memory blocks (Figure 9). SmartGen automatically selects the proper memory block type and aspect ratio, and cascades the memory blocks based on the user's selection. SmartGen also configures any additional signals that may require tie-off.

SmartGen will attempt to use the minimum number of blocks required to implement the desired memory. When cascading, SmartGen will configure the memory for width before configuring for depth. For example, if the user requests a 256x8 FIFO, SmartGen will use a 512x9 FIFO configuration, not 256x18.

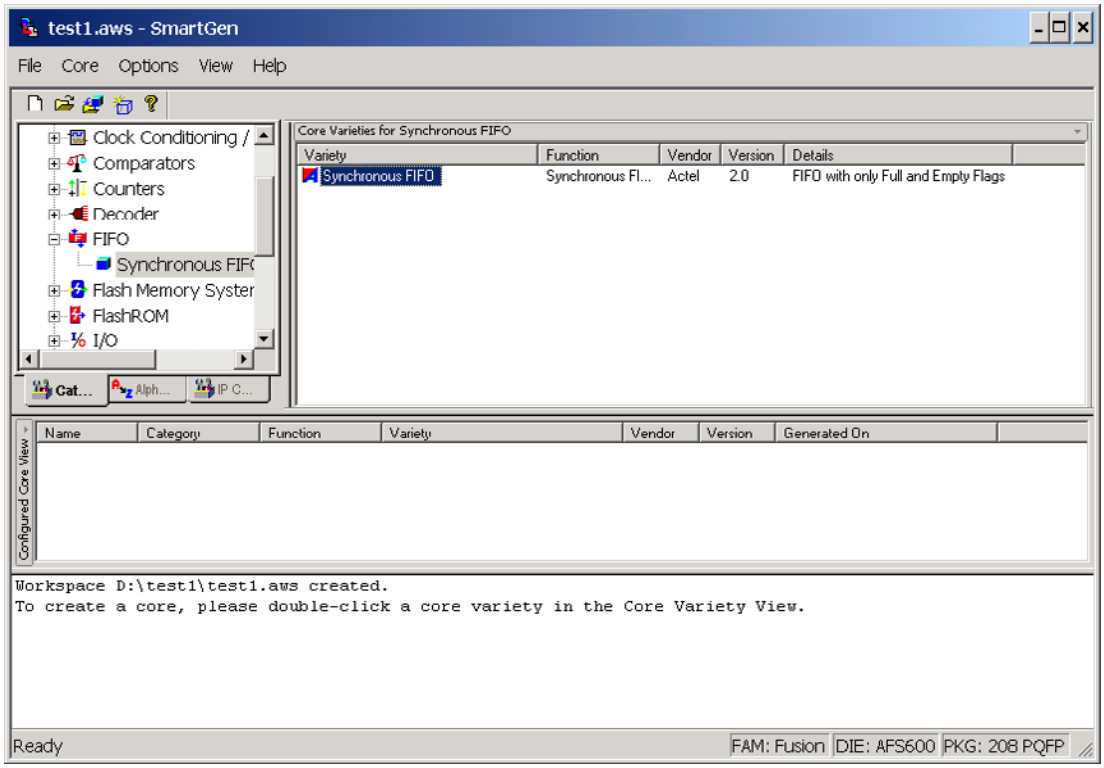


Figure 9 • **SmartGen Core Generator Interface**

SmartGen enables the user to configure the desired RAM element to use either a single clock for read and write, or two independent clocks for read and write. The user can select the type of RAM as well as the width/depth and several other parameters (Figure 10).

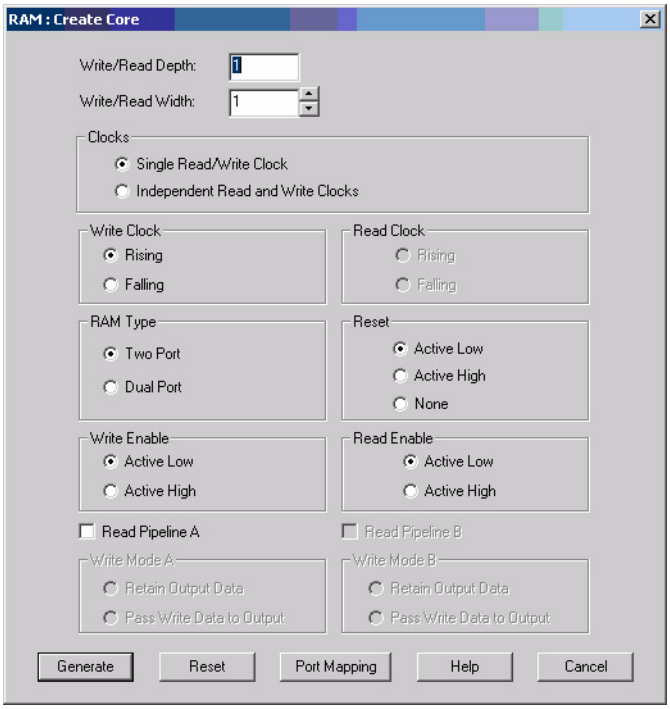


Figure 10 • **SmartGen Memory Configuration Interface**

SmartGen also has a Port Mapping option that allows the user to specify the names of the ports generated in the memory block (Figure 11).

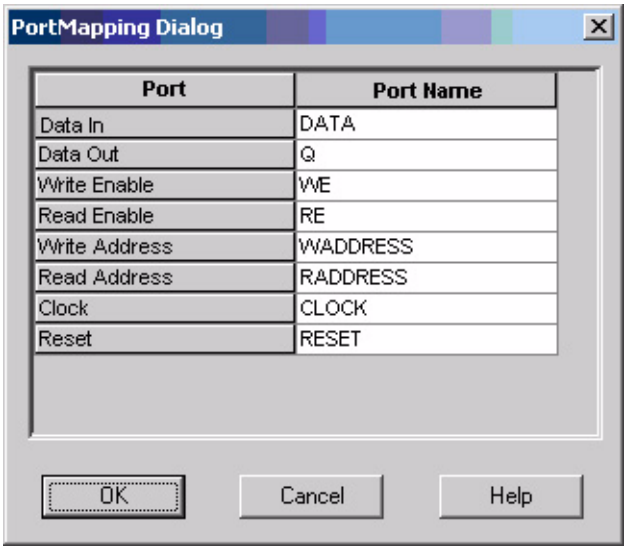


Figure 11 • **Port Mapping Interface for SmartGen Generated Memory**

SmartGen also configures the FIFO according to user specifications. Users can select no flags, static flags, or dynamic flags. Static flag settings are configured using configuration flash and cannot be altered without reprogramming the device. Dynamic flag settings are determined by register values and can be altered without reprogramming the device by reloading the register values either from the design or through the UJTAG interface described in the "Initializing the Fusion RAM/FIFO" section on page 6.

SmartGen can also configure the FIFO to continue counting after the FIFO is full. In this configuration, the FIFO write counter will wrap after the counter is full and continue to write data. With the FIFO configured to continue to read after the FIFO is empty, the read counter will also wrap and re-read data which was previously read. This mode can be used to continually read back repeating data patterns stored in the FIFO (Figure 12).

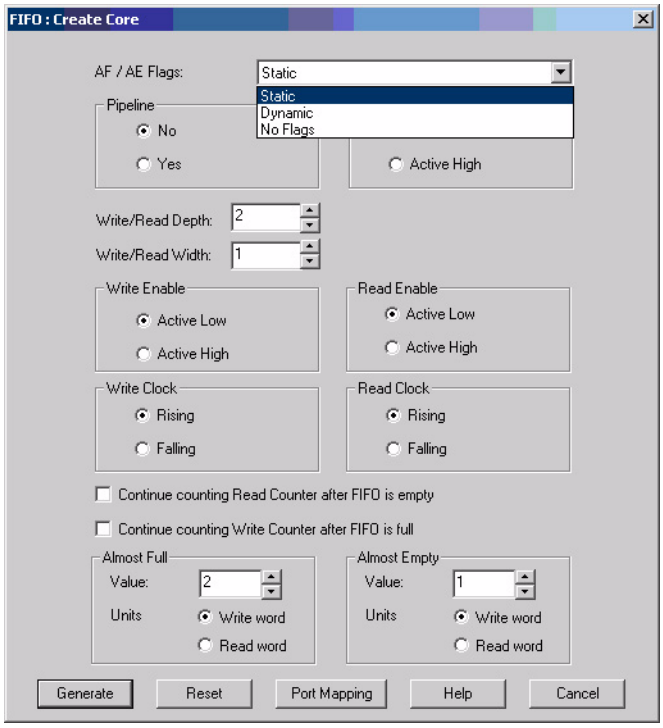


Figure 12 • **SmartGen FIFO Configuration Interface**

FIFOs configured using SmartGen can also make use of the port mapping feature to configure the names of the ports.

Limitations

Users should be aware of the following limitations when configuring SRAM blocks for Fusion:

- SmartGen does not track the target device in a family, so it cannot determine if a configured memory block will fit in the target device.
- Dual-port RAMs with different read and write aspect ratios are not supported.
- Cascaded memory blocks can only use a maximum of 64 blocks of RAM.
- The Full flag of the FIFO is sensitive to the maximum depth of the actual physical FIFO block, not the depth requested in the SmartGen interface.

Conclusion

Fusion devices provide users with extremely flexible SRAM blocks for most design needs, with the ability to choose between an easy-to-use dual-port memory or a wide-word two-port memory. Used with the built-in FIFO controllers, these memory blocks also serve as highly-efficient FIFOs, which do not consume user gates when implemented. The Actel SmartGen core generator provides a fast and easy way to configure these memory elements for use in designs.

Related Documents

Application Notes

Fusion FlashROM (FROM)

http://www.actel.com/documents/Fusion_FROM_AN.pdf

Datasheets

Fusion Family of Mixed-Signal FPGAs

http://www.actel.com/documents/Fusion_DS.pdf

Appendix

Interface Block

```
`define Initialize_start 8'h22 //INITIALIZATION START COMMAND VALUE
`define Initialize_stop 8'h23 //INITIALIZATION START COMMAND VALUE
module interface(IR, rst_n, data_shift, clk_in, data_update, din_ser, dout_ser, test,
test_out,test_clk,clk_out,wr_en,rd_en,write_word,read_word,rd_addr, wr_addr);
input [7:0] IR;
input [3:0] read_word; //RAM DATA READ BACK
input rst_n, data_shift, clk_in, data_update, din_ser; //INITIALIZATION SIGNALS
input test, test_clk; //TEST PROCEDURE CLOCK AND COMMAND INPUT
output [3:0] test_out; //READ DATA
output [3:0] write_word; //WRITE DATA
output [1:0] rd_addr; //READ ADDRESS
output [1:0] wr_addr; //WRITE ADDRESS
output dout_ser; //TDO DRIVER
output clk_out, wr_en, rd_en;
wire [3:0] write_word;
wire [1:0] rd_addr;
wire [1:0] wr_addr;
wire [3:0] Q_out;
wire enable, test_active;
reg clk_out;
//SELECT CLOCK FOR INITIALIZATION OR READBACK TEST
always @(enable or test_clk or data_update)
begin
case ({test_active})
1 : clk_out = test_clk ;
0 : clk_out = !data_update;
default : clk_out = 1'b1;
endcase
end
assign test_active = test && (IR == 8'h23);
assign enable = (IR == 8'h22);
assign wr_en = !enable;
assign rd_en = !test_active;
assign test_out = read_word;
assign dout_ser = Q_out[3];

//4-bit SIN/POUT SHIFT REGISTER
```

```
shift_reg data_shift_reg (.Shiften(data_shift), .Shiftin(din_ser), .Clock(clk_in),
.Q(Q_out));


//4-bit PIPELINE REGISTER


D_pipeline pipeline_reg (.Data(Q_out), .Clock(data_update), .Q(write_word));


//


addr_counter counter_1 (.Clock(data_update), .Q(wr_addr), .Aset(rst_n),
.Enable(enable));
addr_counter counter_2 (.Clock(test_clk), .Q(rd_addr), .Aset(rst_n), .Enable(
test_active));
endmodule
```

Interface Block/UJTAG Wrapper

This example is a sample wrapper, which connects the interface block to the UJTAG and the memory blocks.

```
// WRAPPER
module top_init (TDI, TRSTB, TMS, TCK, TDO, test, test_clk, test_ out);
input TDI, TRSTB, TMS, TCK;
output TDO;
input test, test_clk;
output [3:0] test_out;
wire [7:0] IR;
wire reset, DR_shift, DR_cap, init_clk, DR_update, data_in, data_out;
wire clk_out, wen, ren;
wire [3:0] word_in, word_out;
wire [1:0] write_addr, read_addr;
UJTAG UJTAG_U1
(.UIREG0(IR[0]),.UIREG1(IR[1]),.UIREG2(IR[2]),.UIREG3(IR[3]),.UIREG4(IR[4]),
.UIREG5(IR[5]),.UIREG6(IR[6]),.UIREG7(IR[7]),.URSTB(reset),.UDRSH(DR_shift),.UDRCAP(
DR_cap),.UDRCK(init_clk),
.UDRUPD(DR_update),.UT-DI(data_in),.TDI(TDI),.TMS(TMS),.TCK(TCK),
.TRSTB(TRSTB),.TDO(TDO),.UT-DO(data_out));
mem_block RAM_block (.DO(word_out), .RCLOCK(clk_out), .WCLOCK(clk_out), .DI(word_in),
.WRB(wen),
.RDB(ren), .WAD-DR(write_addr), .RADDR(read_addr));
interface init_block (.IR(IR), .rst_n(reset), .data_shift(DR_shift),
.clk_in(init_clk),
.data_update(DR_update), .din_ser(data_in), .dout_ser(data_out), .test(test),
.test_out(test_out), .test_clk(test_clk), .clk_out(clk_out), .wr_en(wen),
```

```
.rd_en(ren), .write_word(word_in), .read_word(word_out), .rd_addr(read_addr),
.wr_addr(write_addr));
endmodule
```

Address Counter

```
module addr_counter (Clock, Q, Aset, Enable);
input Clock;
output [1:0] Q;
input Aset;
input Enable;
reg [1:0] Qaux;
always @(posedge Clock or negedge Aset)
begin
if (!Aset)
Qaux <= 2'b11;
else if (Enable)
Qaux <= Qaux + 1;
end
assign Q = Qaux;
endmodule
Pipeline Register:
module D_pipeline (Data, Clock, Q);
input [3:0] Data;
input Clock;
output [3:0] Q;
reg [3:0] Q;
always @ (posedge Clock)
Q <= Data;
endmodule
```

4x4 RAM Block (Created by SmartGen Core Generator)

```
module mem_block(DI,DO,WADDR,RADDR,WRB,RDB,WCLOCK,RCLOCK);
input [3:0] DI;
output [3:0] DO;
input [1:0] WADDR, RADDR;
input WRB, RDB, WCLOCK, RCLOCK;
wire WEBP, WEAP, VCC, GND;
VCC VCC_1_net(.Y(VCC));
GND GND_1_net(.Y(GND));
INV WEBUBBLEB(.A(WRB), .Y(WEBP));
```

```
RAM4K9 RAMBLOCK0(.ADDRA11(GND), .ADDRA10(GND), .ADDRA9(GND),
.ADDRA8(GND), .ADDRA7(GND), .ADDRA6(GND), .ADDRA5(GND),
.ADDRA4(GND), .ADDRA3(GND), .ADDRA2(GND), .ADDRA1(
RADDR[1]), .ADDRA0(RADDR[0]), .ADDRB11(GND), .ADDRB10(GND)
, .ADDRB9(GND), .ADDRB8(GND), .ADDRB7(GND), .ADDRB6(GND),
.ADDRB5(GND), .ADDRB4(GND), .ADDRB3(GND), .ADDRB2(GND),
.ADDRB1(WADDR[1]), .ADDRB0(WADDR[0]), .DINA8(GND), .DINA7(
GND), .DINA6(GND), .DINA5(GND), .DINA4(GND), .DINA3(GND),
.DINA2(GND), .DINA1(GND), .DINA0(GND), .DINB8(GND),
.DINB7(GND), .DINB6(GND), .DINB5(GND), .DINB4(GND),
.DINB3(DI[3]), .DINB2(DI[2]), .DINB1(DI[1]), .DINB0(DI[0])
, .WIDTHA0(GND), .WIDTHA1(VCC), .WIDTHB0(GND), .WIDTHB1(
VCC), .PIPEA(GND), .PIPEB(GND), .WMODEA(GND), .WMODEB(GND)
, .BLKA(WEAP), .BLKB(WEBP), .WENA(VCC), .WENB(GND), .CLKA(
RCLOCK), .CLKB(WCLOCK), .RESET(VCC), .DOUTA8(), .DOUTA7(),
.DOUTA6(), .DOUTA5(), .DOUTA4(), .DOUTA3(DO[3]), .DOUTA2(
DO[2]), .DOUTA1(DO[1]), .DOUTA0(DO[0]), .DOUTB8(),
.DOUTB7(), .DOUTB6(), .DOUTB5(), .DOUTB4(), .DOUTB3(),
.DOUTB2(), .DOUTB1(), .DOUTB0());
INV WEBUBBLEA(.A(RDB), .Y(WEAP));
endmodule
```

www.actel.com

Actel Corporation

2061 Stierlin Court
Mountain View, CA
94043-4655 USA
Phone 650.318.4200
Fax 650.318.4600

Actel Europe Ltd.

Dunlop House, Riverside Way
Camberley, Surrey GU15 3YL
United Kingdom
Phone +44 (0) 1276 401 450
Fax +44 (0) 1276 401 490

Actel Japan
www.jp.actel.com

EXOS Ebisu Bldg. 4F
1-24-14 Ebisu Shibuya-ku
Tokyo 150 Japan
Phone +81.03.3445.7671
Fax +81.03.3445.7668

Actel Hong Kong
www.actel.com.cn

Suite 2114, Two Pacific Place
88 Queensway, Admiralty
Hong Kong
Phone +852 2185 6460
Fax +852 2185 6488

51900114-0/11.05