Stratix II Memory Block EDA Functional Description

Version 1.1
March 22, 2004

by
Altera Corporation

Author:	Altera Corporation
System:	
Subsystem:	
Keywords:	Stratix II, RAM, Memory block

Table of Contents:

1 Overview

This document is intended for EDA tool developers working with the Stratix II memory architecture. It is a companion to the "Stratix II RAM WYSIWYG Description" and should be used in conjunction with it. This document provides information about the Stratix II memory architecture to allow EDA vendors to infer these blocks optimally.

2 The Stratix II Memory Block Description

The Stratix II Memory block (also referred to as the RAM block) is a block that provides storage resources. As in the Stratix architecture, there are 3 sizes of memory blocks for various functions. The M512 RAM block is a simple dual-port RAM and can store up to 576 (64x9) bits data. The M4K RAM block is a true dual-port RAM and can store up to 4608(512x9) bits data. The M-RAM block is also a true dual-port RAM and can store up to 589824(64Kx9) bits data. The following picture shows the high-level functional block of the Stratix II RAM.

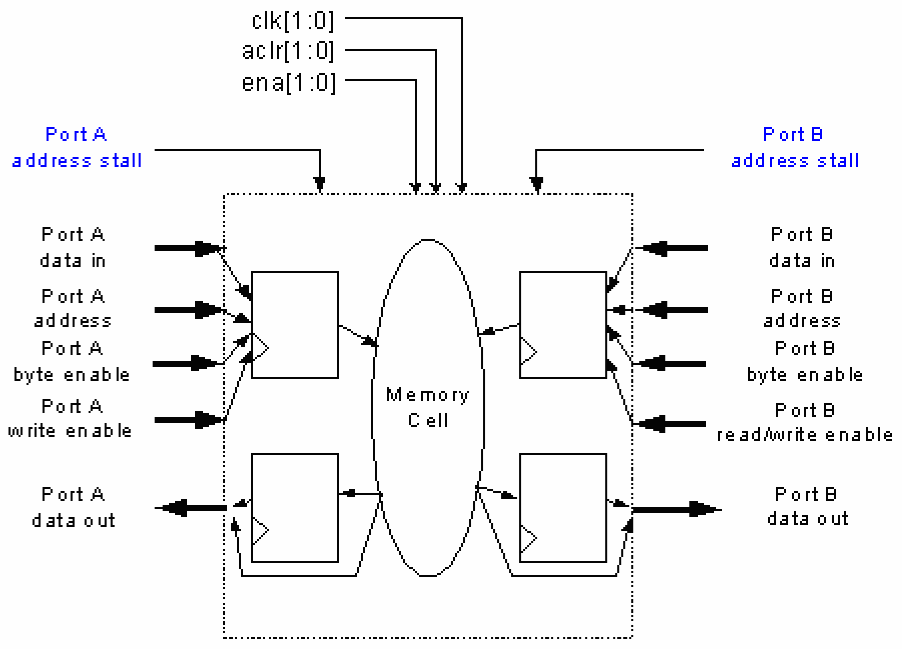
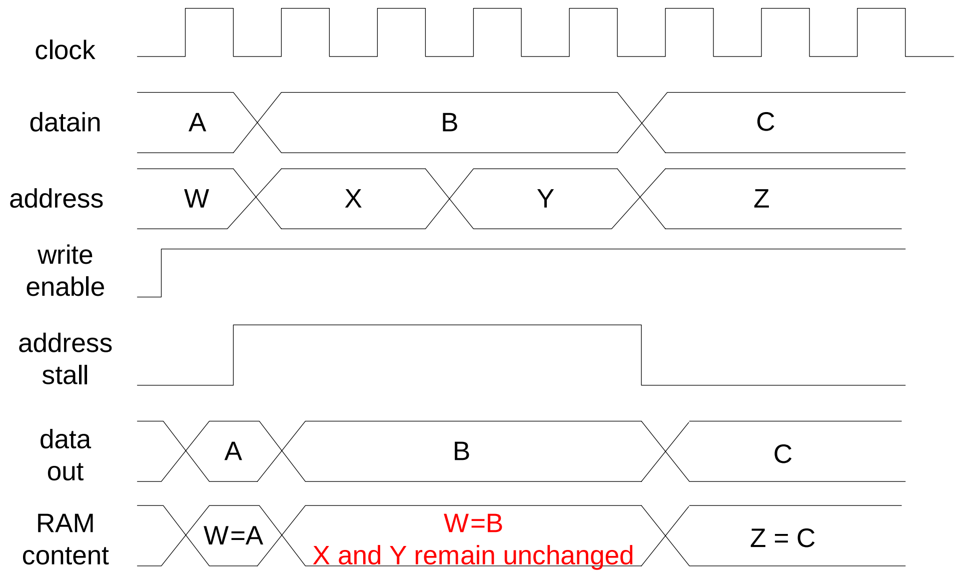


Figure 1: Stratix II RAM

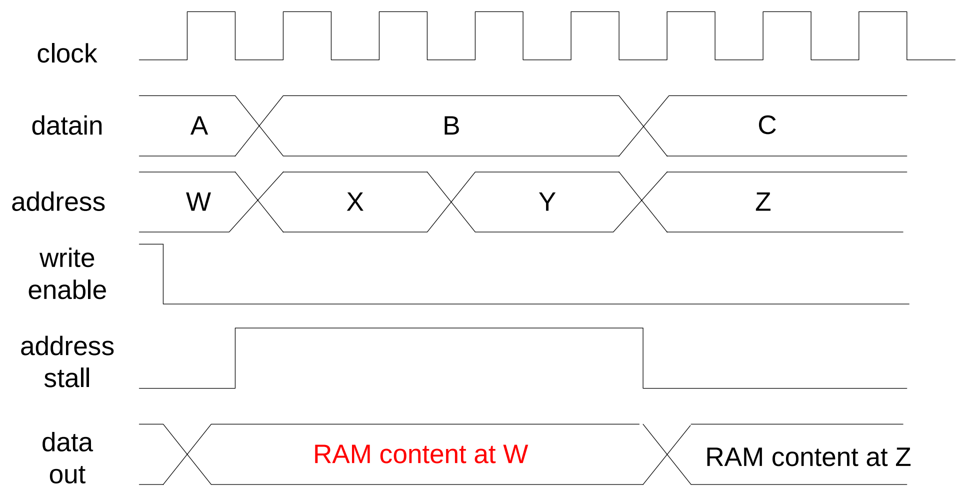
From the feature aspect, the Stratix II memory is almost a superset of the Stratix memory with the exception of asynchronous clear on the input registers, which is only available in Stratix families. In addition, the following enhancements are added in the Stratix II memory.

2.1 Address-Stall

Address-Stall feature has been added to the M4K RAM block and the M-RAM block. Address stall is a feature that can hold the previous address value for as long as the stall signal is enabled. This feature is added to improve the efficiency in cache-miss applications. It actually behaves like an additional clock-enable on the clock input of the address registers. The following waveforms show its behavior during a read/write cycle.



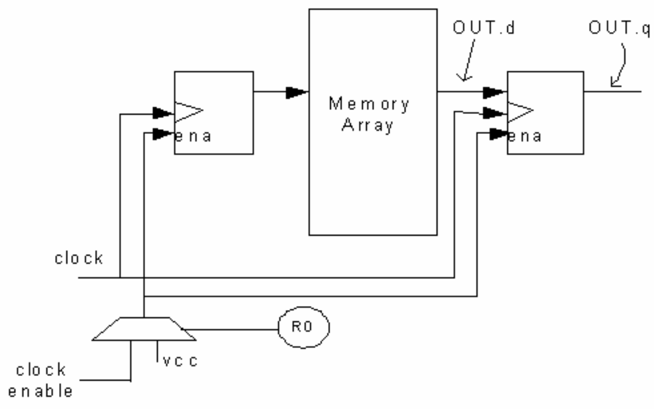
Address Stall during write cycle
assuming output is unregistered



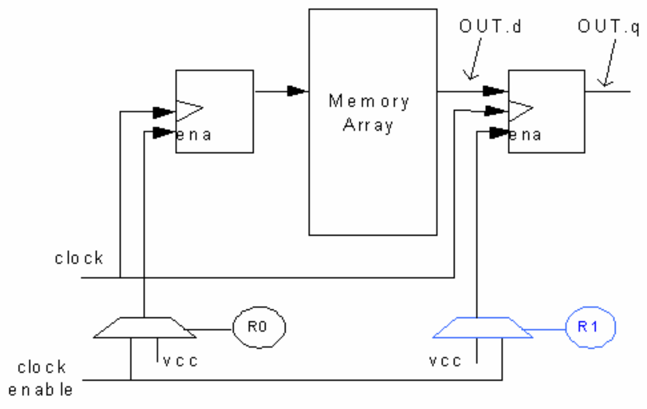
Address Stall during read cycle
assuming output is unregistered

2.2 Improved Clock-Enable Control

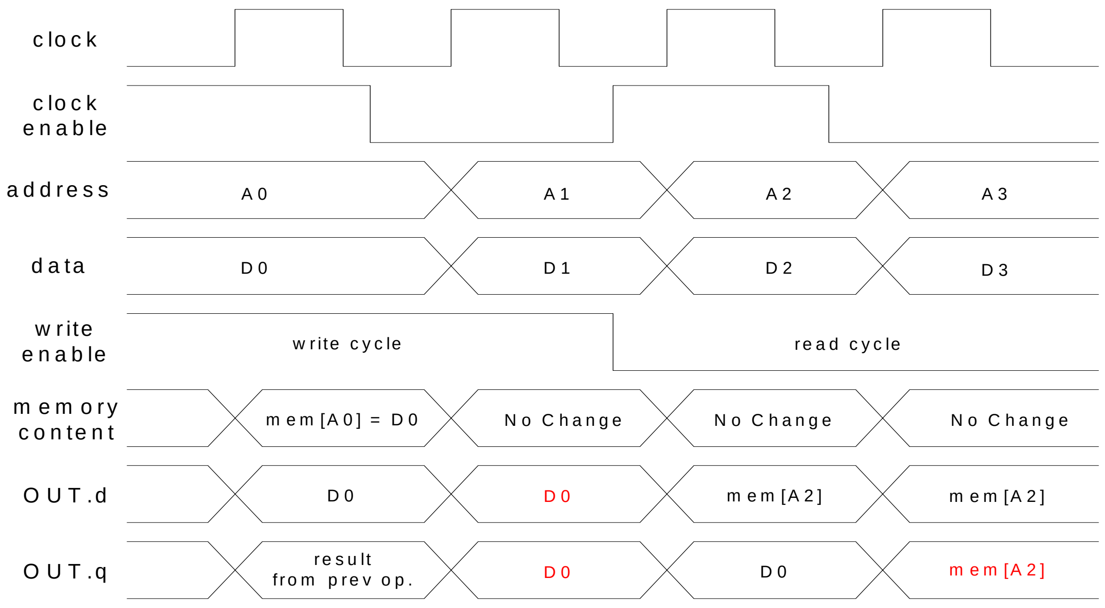
The Stratix II memory blocks have more flexible clock-enable control. In the Stratix architecture, once the clock is paired with clock enable, all registers fed by the clock will get the clock enable. This has been causing problems in our DCFIFO (dual-clock FIFO) megafunction implementation. The DCFIFO megafunction requires two clocks, one for read and one for write. However, it also requires gating the read clock on the output registers with clock enables while leaving the read clock not-gated on read input registers. This is something the Stratix hardware cannot support. And this incurs some area overhead on the DCFIFO implementation in the Stratix architecture. In the Stratix II architecture, this should no longer be a problem with the additional flexibility of independent control of the clock enable on input registers vs. output registers. The following waveforms describe the behavior in details.



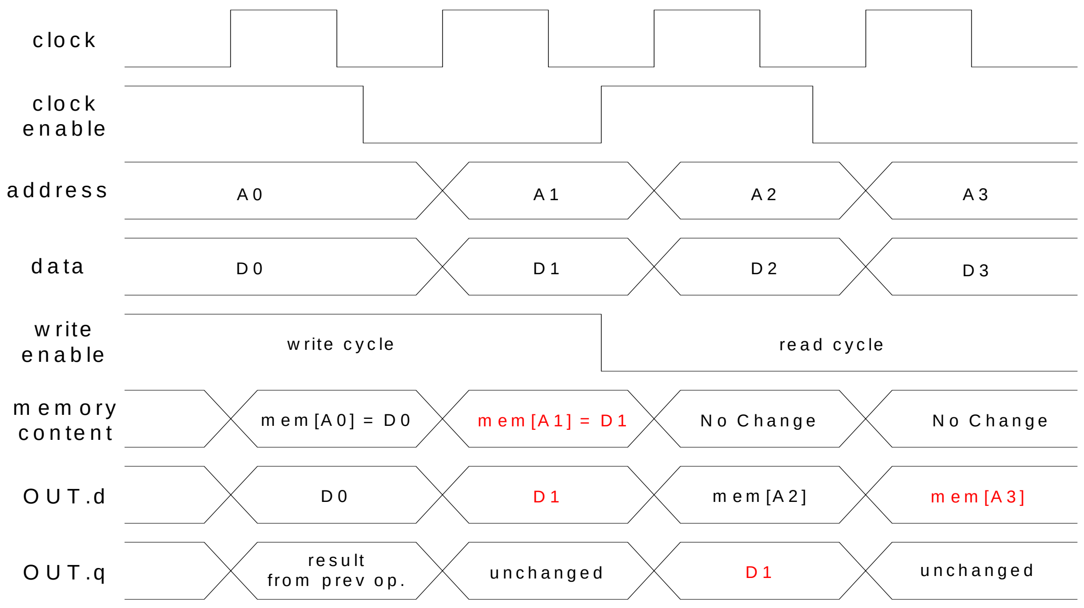
Stratix



Stratix II



Example: Single-Port RAM with registered outputs
CLOCK_ENABE_INPUT_A = NORMAL
CLOCK_ENABLE_OUTPUT_A = BYPASS



Example: Single-Port RAM with registered outputs
port_a_disable_ce_on_input_registers = on,
port_a_disable_ce_on_output_registers = off

3 Instantiating RAM Blocks

RAM blocks can be instantiated by either instantiating the RAM_BLOCK WYSIWYGs directly, or by instantiating the RAM related megafunctions. As with the Stratix family, we recommend instantiating the altsyncram megafunction instead of the WYSIWYGs directly due to the complexity of RAMs.

The same megafunctions used to support Stratix RAM blocks are updated to support Stratix II RAM blocks, and these megafunctions are backwards compatible with the Stratix megafunctions. As a result, a megafunction instantiated for Stratix devices can also be compiled for Stratix II devices.

Here are the new ports and parameters supported in the Stratix II RAM blocks.

Two new ports:

```
    addressstall_a                    : INPUT = GND;
    addressstall_b                    : INPUT = GND;
```

Four new parameters:

```
    CLOCK_ENABLE_INPUT_A = "NORMAL",  -- “NORMAL” or “BYPASS”
    CLOCK_ENABLE_OUTPUT_A = " NORMAL ", -- “NORMAL” or “BYPASS”
    CLOCK_ENABLE_INPUT_B = "NORMAL",  -- “NORMAL” or “BYPASS”
    CLOCK_ENABLE_OUTPUT_B = " NORMAL ", -- “NORMAL” or “BYPASS”
```

And as said earlier, the Stratix II RAM does not have the asynchronous clear on the input registers. So the following parameters should either not be used in the Stratix II RAM or they should be set to “NONE”

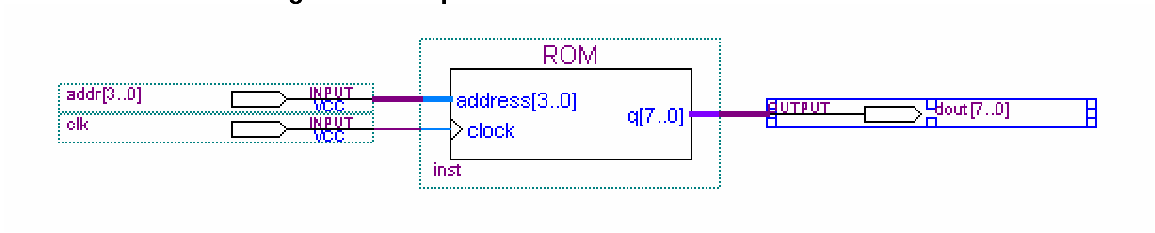
```
    ADDRESS_ACLR_A = "NONE",
    INDATA_ACLR_A = "NONE",
    WRCONTROL_ACLR_A = "NONE",
    BYTEENA_ACLR_A = “NONE”,

    RDCONTROL_ACLR_B = "NONE",
    INDATA_ACLR_B = "NONE",
    WRCONTROL_ACLR_B = "NONE",
    ADDRESS_ACLR_B = "NONE",
    BYTEENA_ACLR_B = “NONE”,
```

4 Inferring RAM Blocks

As in the Stratix family, synthesis tools can infer RAM blocks from the HDL source. The followings are the examples for each operation modes that can be inferred as RAM.

4.1 ROM

4.1.1 ROM with unregistered output



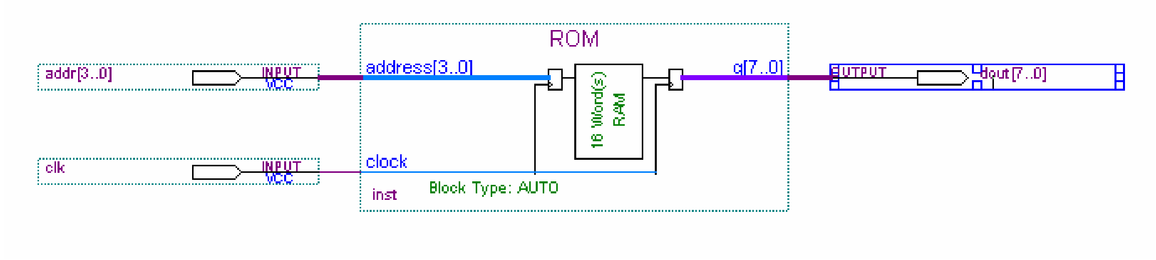
Example 1:

```
// ROM with unregistered output
module ROM(clk, addr, dout);

input clk;
input [3:0] addr;
output [7:0] dout;

reg [7:0] dout;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
        mem[0] = 8'b01100011;
        mem[1] = 8'b11111111;
        mem[2] = 8'b00001111;
        mem[3] = 8'b11110000;
        mem[4] = 8'b10000000;
        mem[5] = 8'b00000001;
        mem[6] = 8'b00001000;
        mem[7] = 8'b00110000;
        mem[8] = 8'b00110011;
        mem[9] = 8'b11001100;
        mem[10] = 8'b11010011;
        mem[11] = 8'b10010001;
        mem[12] = 8'b11000011;
        mem[13] = 8'b10100101;
        mem[14] = 8'b10101010;
        mem[15] = 8'b00000000;

        dout = mem[addr];
end

endmodule
```

Example 2:

```
// ROM with unregistered output
module ROM(clk, addr, dout);

input clk;
input [3:0] addr;
output [7:0] dout;

reg [7:0] dout;

always @ (posedge clk) begin
        case (addr)
        0: dout = 8'b01100011;
        1: dout = 8'b11111111;
        2: dout = 8'b00001111;
        3: dout = 8'b11110000;
        4: dout = 8'b10000000;
        5: dout = 8'b00000001;
        6: dout = 8'b00001000;
        7: dout = 8'b00110000;
        8: dout = 8'b00110011;
        9: dout = 8'b11001100;
        10: dout = 8'b11010011;
        11: dout = 8'b10010001;
        12: dout = 8'b11000011;
        13: dout = 8'b10100101;
        14: dout = 8'b10101010;
        15: dout = 8'b00000000;
        endcase
end

endmodule
```

4.1.2 ROM with registered output



Example 1: Power-up 0 (Chip behavior)

// ROM with registered output (Not power up reading)

```
// The first clock cycle will not read out the RAM content.
// It will output the value of buffer tmpout, which is 0.
module ROM(clk, addr, dout);

input clk;
input [3:0] addr;
output [7:0] dout;

reg [7:0] tmpout;
reg [7:0] dout;

always @ (posedge clk) begin
        case (addr)
        0: tmpout <= 8'b01100011;
        1: tmpout <= 8'b11111111;
        2: tmpout <= 8'b00001111;
        3: tmpout <= 8'b11110000;
        4: tmpout <= 8'b10000000;
        5: tmpout <= 8'b00000001;
        6: tmpout <= 8'b00001000;
        7: tmpout <= 8'b00110000;
        8: tmpout <= 8'b00110011;
        9: tmpout <= 8'b11001100;
        10: tmpout <= 8'b11010011;
        11: tmpout <= 8'b10010001;
        12: tmpout <= 8'b11000011;
        13: tmpout <= 8'b10100101;
        14: tmpout <= 8'b10101010;
        15: tmpout <= 8'b00000000;
        endcase

        dout <= tmpout;
end

endmodule
```

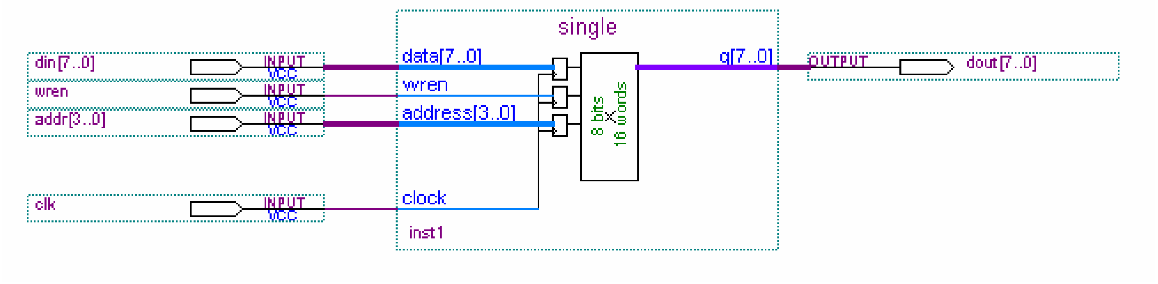
Example 2: Power up reading at address 0 (not chip behavior)
Note, the following example does not exactly match the chip behavior at the first clock cycle after chip powers up. The following code will output initialized content at address 0 at the first clock cycle, but the hardware will output 0. This is a tiny difference and should still be inferred as ROM with an information message on the start-up difference.

```
// ROM with registered output (Power up reading at address 0)
// The first clock cycle will read out content at address[0]
module ROM(clk, addr, dout);
```

```
input clk;
input [3:0] addr;
output [7:0] dout;

reg [7:0] dout;
reg [3:0] addr_reg;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
        mem[0] = 8'b01100011;
        mem[1] = 8'b11111111;
        mem[2] = 8'b00001111;
        mem[3] = 8'b11110000;
        mem[4] = 8'b10000000;
        mem[5] = 8'b00000001;
        mem[6] = 8'b00001000;
        mem[7] = 8'b00110000;
        mem[8] = 8'b00110011;
        mem[9] = 8'b11001100;
        mem[10] = 8'b11010011;
        mem[11] = 8'b10010001;
        mem[12] = 8'b11000011;
        mem[13] = 8'b10100101;
        mem[14] = 8'b10101010;
        mem[15] = 8'b00000000;

        addr_reg <= addr;
        dout <= mem[addr_reg];
end

endmodule
```

4.2 Single-Port RAM

4.2.1 Single-Port RAM with unregistered output



Example 1: Power up to 0 (Chip behavior)

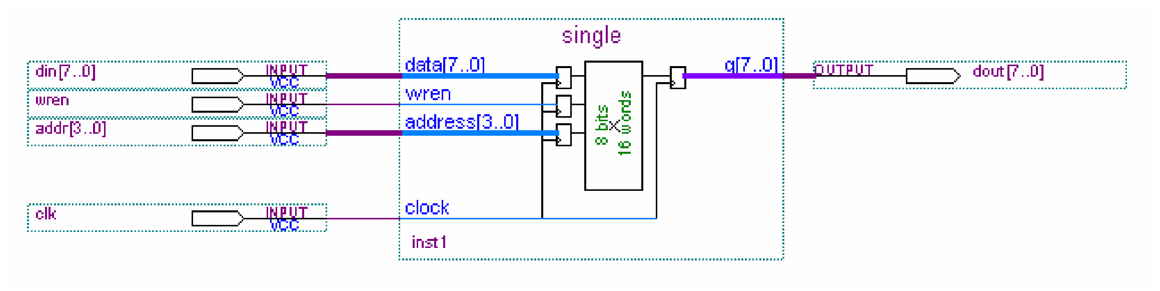
```
// Single-Port RAM with unregistered output
// Power up to 0
module single (clk, din, addr, wren, dout);

input clk, wren;
input [3:0] addr;
input [7:0] din;
output [7:0] dout;

reg [7:0] dout;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
  if (wren)
    mem[addr] = din;                // Note: This is the blocking assignment. The output will
  dout = mem[addr];                 //       get the written data in the same clock cycle.
end

endmodule
```

Example 2: Power up reading at address 0 (not chip behavior)
Note the following Verilog example does not exactly match the chip behavior before the first clock cycle after chip powers up. The following code will output initialized content at address 0 before the first clock cycle, but the hardware will output 0. This is a tiny difference and should still be inferred as single-port RAM with an information message on the start-up difference.

```
// Single-Port RAM with unregistered output
// Power up to memory content at address [0]
module single (clk, din, addr, wren, dout);

input clk, wren;
input [3:0] addr;
input [7:0] din;
output [7:0] dout;

reg [3:0] addr_reg;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
  addr_reg <= addr;
  if (wren)
    mem[addr] <= din;
```

```
end

assign dout = mem[addr_reg];

endmodule
```

4.2.2 Single-Port RAM with registered output



Example 1: Power up 0 (chip behavior)

```
// Single-Port RAM with registered output (Not power-up reading at address 0)
// The first clock cycle will output 0 instead of content at address 0.
module single (clk, din, addr, wren, dout);

input clk, wren;
input [3:0] addr;
input [7:0] din;
output [7:0] dout;

reg [7:0] dout;
reg [7:0] tmpout;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
  if (wren)
    mem[addr] = din;
  tmpout <= mem[addr];
  dout <= tmpout;
end

endmodule
```

Example 2: Power up reading address 0

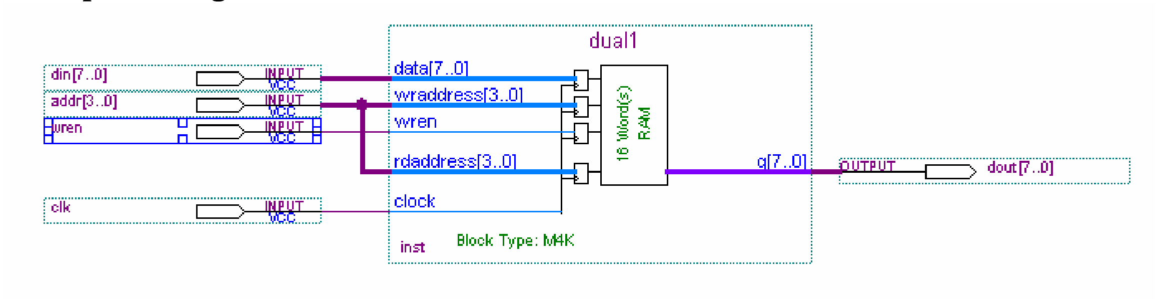
Note the following example can be different from the chip behavior at the first clock cycle when chip starts up if the single-port RAM has pre-initialized content. The chip will show 0 at the first clock cycle while the following code will show memory content at address 0 at the first clock cycle. However, if the RAM is not pre-initialized or content 0 is preset to 0, it can be mapped perfectly to single-port RAM.

```
// Single-Port RAM with registered output (Power up reading at address 0)
// The first clock cycle will read out content at address 0
module single (clk, din, addr, wren, dout);

input clk, wren;
input [3:0] addr;
input [7:0] din;
output [7:0] dout;

reg [7:0] dout;
reg [3:0] addr_reg;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
  addr_reg <= addr;                          // Note: We use non-blocking assignment here
  if (wren)                                  //          so the output will be delayed by 1 cycle
    mem[addr] = din;
  dout = mem[addr_reg];
end

endmodule
```

4.3 Simple Dual-Port

4.3.1 Simple Dual-Port RAM with unregistered output

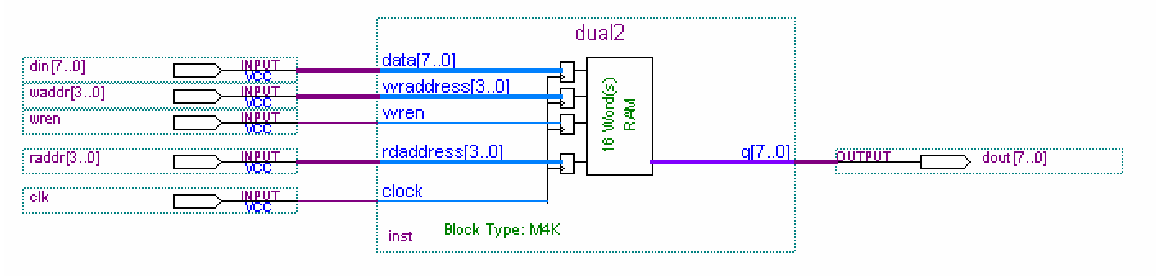
Example 1: Single clock, read/write addresses are the same



```
// Dual-port RAM with unregistered output
// single-clock with the same read and write address
module dual1 (clk, din, addr, wren, dout);

input clk, wren;
input [3:0] addr;
input [7:0] din;
output [7:0] dout;
```

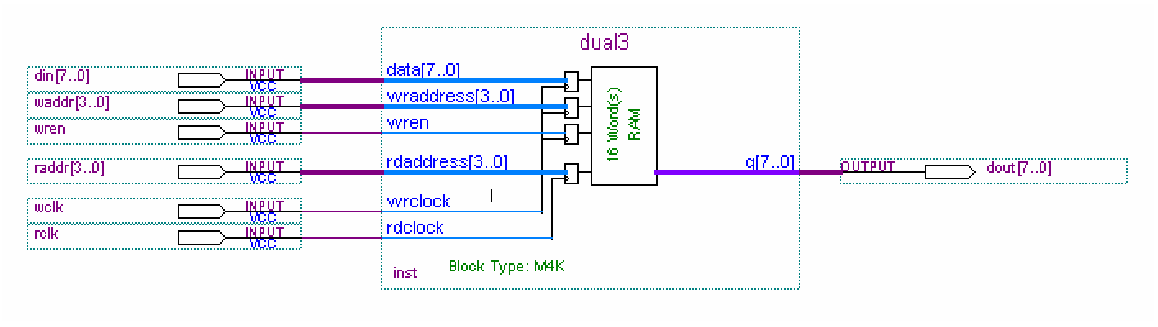
```
reg [7:0] dout;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
  if (wren)                          // Note: This is the non-blocking assignment. The output will
    mem[addr] <= din;           //        get the old data at the current write address
  dout <= mem[addr];           //        during the write cycle. => can't map to single-port
end                          //   READ_DURING_WRITE_MODE_MIXED_PORTS = "OLD_DATA"

endmodule
```

Example 2: Single clock, read/write addresses are different



```
// Dual-port RAM with unregistered output
// single-clock with separate read/write address
module dual2 (clk, din, raddr, waddr, wren, dout);

input clk, wren;
input [3:0] raddr;
input [3:0] waddr;
input [7:0] din;
output [7:0] dout;

reg [7:0] dout;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
  if (wren)
    mem[waddr] <= din;                  // Note: This is the non-blocking assignment. The output will
  dout <= mem[raddr];            //  get the old data at the current write address during the
end                              //  write cycle if read address and write address are the same.
                             //   READ_DURING_WRITE_MODE_MIXED_PORTS = "OLD_DATA"
endmodule
```

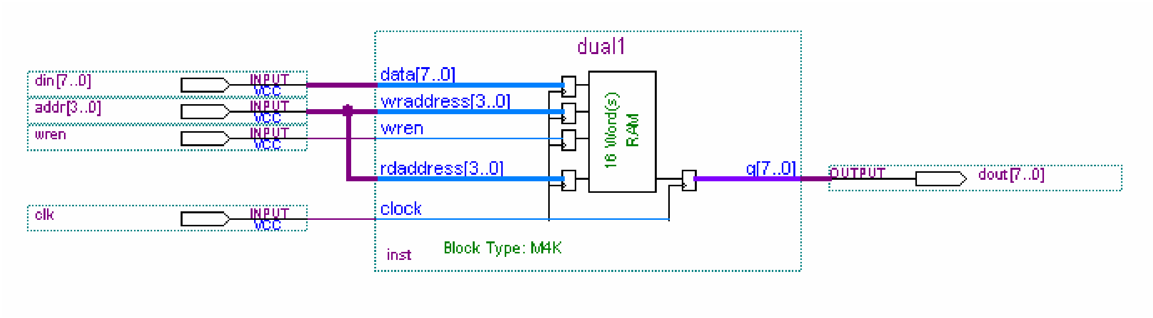
Example 3: Dual clock, read/write addresses are different



```
// Dual-port RAM with unregistered output
// Separate read/write clocks with separate read/write addresses
module dual3 (rclk, wclk, din, raddr, waddr, wren, dout);

input rclk, wclk, wren;
input [3:0] raddr;
input [3:0] waddr;
input [7:0] din;
output [7:0] dout;

reg [7:0] dout;
reg [7:0] mem[15:0];

always @ (posedge wclk) begin
  if (wren)
    mem[waddr] <= din;
end

always @ (posedge rclk) begin
  dout <= mem[raddr];
end

endmodule
```

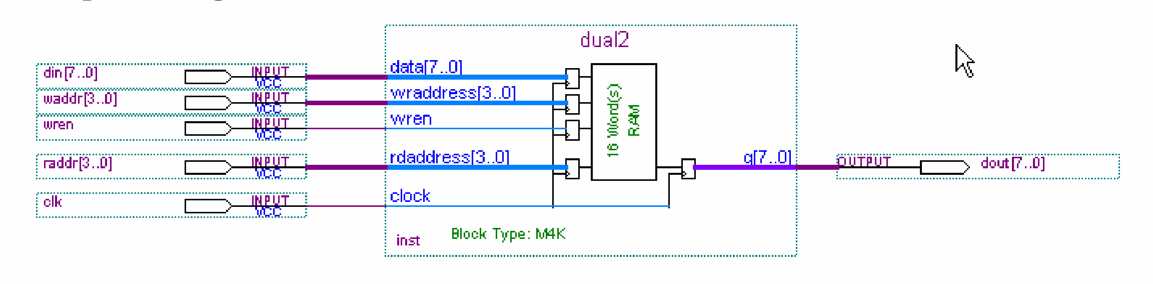
4.3.2 Simple Dual-Port RAM with registered output

Example 1: Single clock, read/write addresses are the same



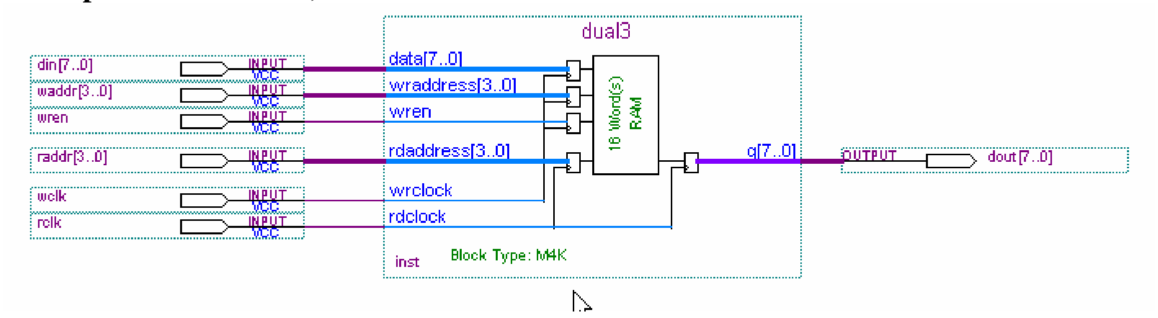
```
// Dual-port RAM with registered output
// single-clock with the same read and write address
module dual1 (clk, din, addr, wren, dout);

input clk, wren;
input [3:0] addr;
input [7:0] din;
output [7:0] dout;

reg [7:0] tmpout;
reg [7:0] dout;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
  if (wren)
    mem[addr] <= din;
  tmpout <= mem[addr];
  dout <= tmpout;
end

endmodule
```

Example 2: Single clock, read/write addresses are different



```
// Dual-port RAM with registered output
// single-clock with separate read/write address
module dual2 (clk, din, raddr, waddr, wren, dout);

input clk, wren;
input [3:0] raddr;
input [3:0] waddr;
input [7:0] din;
output [7:0] dout;

reg [7:0] tmpout;
reg [7:0] dout;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
  if (wren)
    mem[waddr] <= din;                 // Note: This is the non-blocking assignment. The output will
  tmpdout <= mem[raddr];     //   get the old data at the current write address during the
  dout <= tmpdout;                 //   write cycle if read address and write address are the same.
end                                  //   READ_DURING_WRITE_MODE_MIXED_PORTS = “OLD_DATA”

endmodule
```

Example 3: Dual clock, read/write addresses are different



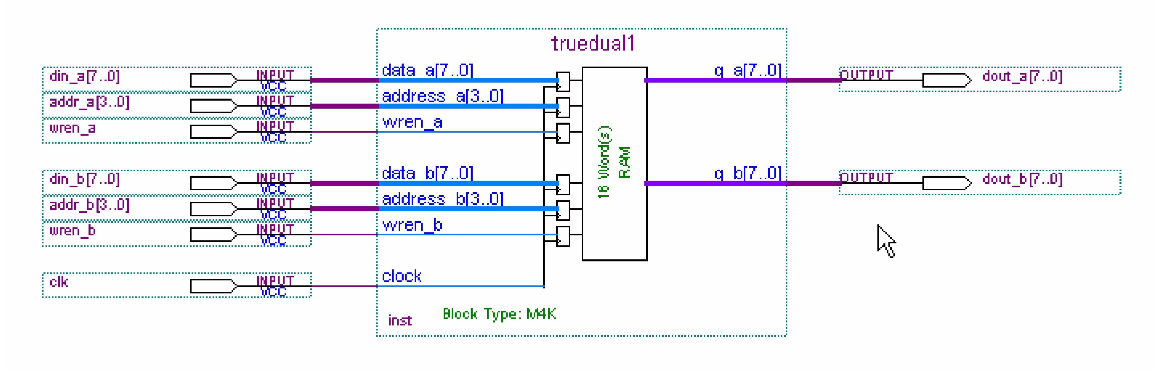
```
// Dual-port RAM with registered output
// Separate read/write clocks with separate read/write addresses
module dual3 (rclk, wclk, din, raddr, waddr, wren, dout);

input rclk, wclk, wren;
input [3:0] raddr;
input [3:0] waddr;
input [7:0] din;
output [7:0] dout;

reg [7:0] tmpdout;
reg [7:0] dout;
reg [7:0] mem[15:0];

always @ (posedge wclk) begin
```

```
  if (wren)
    mem[waddr] <= din;
end

always @ (posedge rclk) begin
  tmpdout <= mem[raddr];
  dout <= tmpdout;
end

endmodule
```

4.4 True Dual-Port

4.4.1 True Dual-Port RAM with unregistered output

Example 1: Single clock for both ports



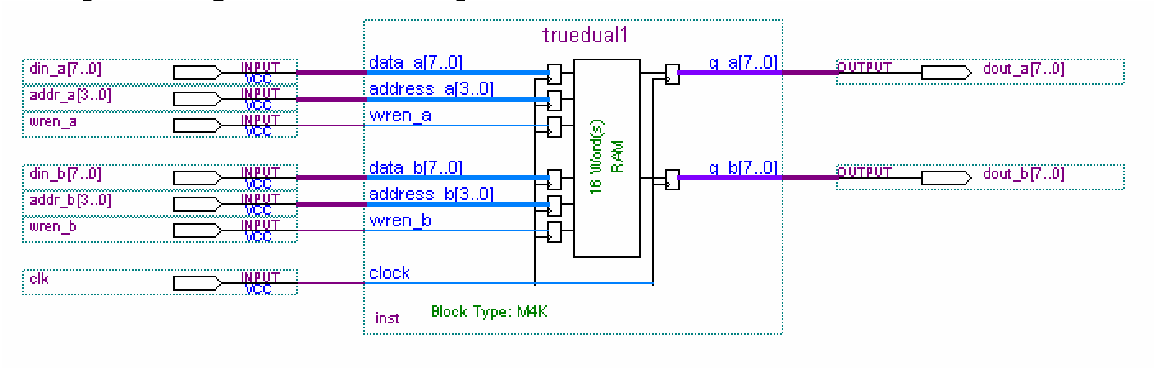
```
// True dual-port RAM with unregistered output
// single-clock for both port A and port B
module truedual1 (clk, din_a, addr_a, wren_a, din_b, addr_b, wren_b, dout_a, dout_b);

input clk;
input wren_a, wren_b;
input [3:0] addr_a, addr_b;
input [7:0] din_a, din_b;
output [7:0] dout_a, dout_b;

reg [7:0] dout_a;
reg [7:0] dout_b;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
  if (wren_a)
     begin
       mem[addr_a] <= din_a;
```

```
      dout_a <= din_a;
    end

  if (wren_b)
    begin
      mem[addr_b] <= din_b;
      dout_b <= din_b;
    end

  if (!wren_a)
    dout_a <= mem[addr_a];

  if (!wren_b)
    dout_b <= mem[addr_b];

end  // always

endmodule
```

Example 2: Dual clocks (port A and port B use separate clocks)
This is not possible in Verilog. The language does not allow a register being modified in two different always statement.

4.4.2 True dual-Port RAM with registered output

Example 1: Single clock for both ports



```
// True dual-port RAM with registered output
// single-clock for both port A and port B
module truedual1 (clk, din_a, addr_a, wren_a, din_b, addr_b, wren_b, dout_a, dout_b);

input clk;
input wren_a, wren_b;
input [3:0] addr_a, addr_b;
input [7:0] din_a, din_b;
output [7:0] dout_a, dout_b;

reg [7:0] tmpdout_a;
reg [7:0] tmpdout_b;
reg [7:0] dout_a;
reg [7:0] dout_b;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
  if (wren_a)
     begin
       mem[addr_a] <= din_a;
       tmpdout_a <= din_a;
     end

  if (wren_b)
    begin
       mem[addr_b] <= din_b;
       tmpdout_b <= din_b;
    end

  if (!wren_a)
    tmpdout_a <= mem[addr_a];

  if (!wren_b)
    tmpdout_b <= mem[addr_b];

  dout_a <= tmpdout_a;
  dout_b <= tmpdout_b;

end  // always

endmodule
```

Example 2: Dual clock (separate read/write clock)
This is not possible in Verilog. The language does not allow a register being modified in two different always statement.

4.5 Address Stall

```
// Single-Port RAM with unregistered output and address stall
// Power up to 0
module single (clk, din, addr, wren, addr_stall, dout);
```

```
input clk, wren;
Input addr_stall;
input [3:0] addr;
input [7:0] din;
output [7:0] dout;
reg [7:0] dout;
reg [7:0] mem[15:0];
reg [3:0] real_addr;

always @ (posedge clk) begin
    if (!addr_stall)
          real_addr = addr;
    if (wren)
        mem[real_addr] = din;                // Note: This is the blocking assignment. The output will
    dout = mem[real_addr];    //        get the written data in the same clock cycle.
end
endmodule
```

4.6 Independently configured clock-enable on input registers vs output registers

Example 1: Single-Port RAM with registered output. Clock of input registers is not gated by clock-enable

```
// Single-Port RAM with registered output. Output powers up to 0.
// CLOCK_ENABLE_INPUT_A = BYPASS, CLOCK_ENABLE_OUTPUT_A = NORMAL
module single (clk, ce, din, addr, wren, dout);

input clk, wren, ce;
input [3:0] addr;
input [7:0] din;
output [7:0] dout;

reg [7:0] dout;
reg [7:0] tmpout;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
    if (wren)
         mem[addr] = din;
    tmpout <= mem[addr];

    if (ce)
      dout <= tmpout;
end

endmodule
```

Example 2: Single-Port RAM with registered output. Clock of output registers is not gated by clock-enable

```
// Single-Port RAM with registered output. Output powers up to 0.
// CLOCK_ENABLE_INPUT_A = NORMAL, CLOCK_ENABLE_OUTPUT_A = BYPASS
module single (clk, ce, din, addr, wren, dout);

input clk, wren, ce;
input [3:0] addr;
input [7:0] din;
output [7:0] dout;

reg [7:0] dout;
reg [7:0] tmpout;
reg [7:0] mem[15:0];

always @ (posedge clk) begin
    if (ce) begin
            if (wren)
                mem[addr] = din;
            tmpout <= mem[addr];
    end

    dout <= tmpout;
end

endmodule
```