Inferring Microchip RTG4 RAM Blocks

Synopsys® Application Note, April 2021

Microchip RTG4 devices support the RAM1K18_RT and RAM64X18_RT RAM macros. This application note provides a general description of the Microchip RTG4 RAM block components and describes how to infer it with the Synplify® Pro synthesis tool.

See the following topics for details:

RTG4 RAM Blocks

RTG4 devices support two types of RAM macros:

- RAM1K18_RT
- RAM64X18_RT

RAM1K18_RT

RAM1K18_RT memory block has the following features:

- 18,432 memory bits with ECC and 24,576 memory bits without ECC.
- Two independent data ports, A and B.
- For dual-port mode, both ports of the memory block have word widths less than or equal to 18 bits.
- Infer single-port, simple dual-port, and true dual-port RAMs.

- For two-port mode, port A is the read-port and port B is the write-port.
- Optional pipeline register with a separate enable and synchronous reset at the read-data port.
- Synchronous read and write operations.
- Does not allow read and write operations on the same location at the same time. It has no collision prevention and detection.
- Feedthrough write mode is not supported.
- Read-enable control for dual-port and two-port modes.
- Allows read from both ports at the same location.
- Raises flags to indicate single-bit-correct and double-bit-detect when ECC is enabled.

RAM64X18_RT

RAM64X18_RT memory block has the following features:

- Two independent read-data ports A and B, and one write-data port C.
- Write operations that are always synchronous.
- For both read-data ports, the address can be set as synchronous or asynchronous.
- Two read-data ports that have output registers for pipeline mode operation.
- Allows read from both ports A and B at the same location.
- Does not allow read and write on the same location at the same time. It has no collision prevention or detection.
- Allows read from both ports at the same location.
- Raises flags to indicate single-bit-correct and double-bit-detect when ECC is enabled.

Inferring RTG4 RAM Blocks

The synthesis tool identifies the RAM structure from the RTL and implements RAM1K18_RT or RAM64X18_RT block.

The RAM block is selected for mapping using the following criteria:

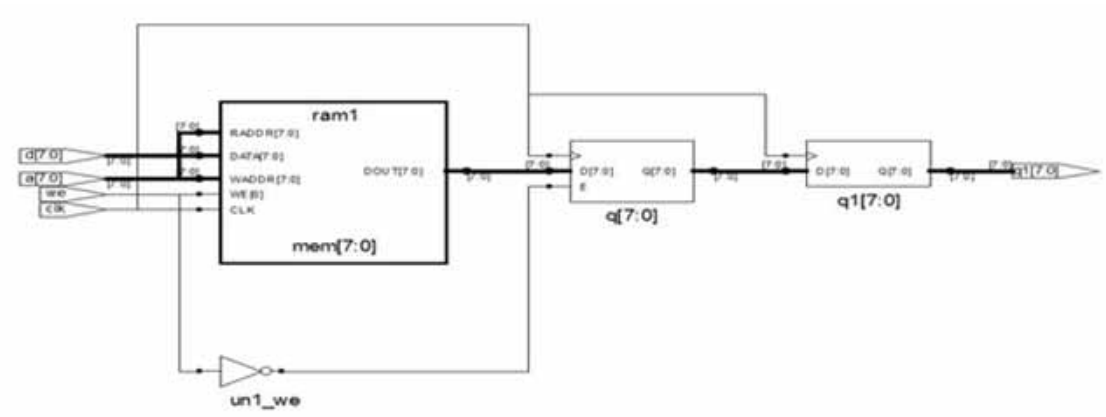
1. For true dual-port synchronous read memory, the synthesis tool maps the RAM1K18_RT block regardless of its memory size.
2. For simple dual-port, single-port, or three-port synchronous memory:
   - If the size of the memory is 4608 bits or more, the software maps to RAM1K18_RT.
   - If the size of the memory is more than 12 bits and less than 4608 bits, the software maps to RAM64X18_RT.
   - If the size of memory is less than or equal to 12 bits, the software maps to registers.
3. For simple dual-port, single-port, or three-port asynchronous memory, if the size of the memory is 12 bits or more, the software maps to RAM64X18_RT. If the size of the memory is less than 12 bits, the synthesis tool maps to registers.

Note: You can override this default behavior using the syn_ramstyle attribute. See Controlling Inference With syn_ramstyle Attribute, on page 4.

Pipeline Register Packing

The synthesis tool performs pipeline register packing as described below:

- The tool extracts a pipeline register at the output of the block RAM and packs it in the RAM1K18_RT block. The RTL view below shows the pipeline register.



- The pipeline register q1 [7:0] is not packed when the register q [7:0] has asynchronous/synchronous reset.
- The pipeline register q1 [7:0] can have asynchronous reset or synchronous reset or clock enable.

Controlling Inference With syn_ramstyle Attribute

Use the syn_ramstyle attribute to manually control how the RTG4 RAM blocks are inferred, as described below. To map to:

- RAM1K18_RT—use syn_ramstyle = "lsram"
- RAM64x18_RT—use syn_ramstyle = "uram"
- registers—use syn_ramstyle = "registers"

You can apply the attribute globally, or to a RAM instance or module.

Constraint File Syntax and Example

```
define_attribute {signalName[bitRange]} syn_ramstyle {string}
define_global_attribute syn_ramstyle {string}
```

When editing a constraint file to apply the syn_ramstyle attribute, ensure to include the range of the signal with the signal name. For example:

```
define_attribute {mem1[7:0]} syn_ramstyle {registers};
define_attribute {mem2[7:0]} syn_ramstyle {lsram};
define_attribute {mem3[7:0]} syn_ramstyle {uram};
```

Verilog Syntax and Example

```
object /* synthesis syn_ramstyle = "string" */ ;
```

where object is a register definition signal. The data type is string. Here is an example:

```
module ram4 (datain,dataout,clk); output [31:0] dataout;
input clk;
input [31:0] datain;
reg [7:0] dataout[31:0] /* synthesis syn_ramstyle="uram" */;
// Other code
```

VHDL Syntax and Example

```
attribute syn_ramstyle of object : objectType is "string" ;
```

where object is a signal that defines a RAM or a label for a component instance. Data type is string.

```
library ieee;
use ieee.std_logic_1164.all;
library synplify;

entity ram4 is
port (d : in std_logic_vector(7 downto 0);

addr : in std_logic_vector(2 downto 0);
we : in std_logic;
clk : in std_logic;
ram_out : out std_logic_vector(7 downto 0) );
end ram4;

architecture rtl of ram4 is
type mem_type is array (127 downto 0) of std_logic_vector
   (7 downto 0);
signal mem : mem_type;
-- mem is the signal that defines the RAM

attribute syn_ramstyle : string;
attribute syn_ramstyle of mem : signal is "lsram";
-- Other code
```

Note the following:

- If your RTL code includes a true dual-port synchronous read memory, then you cannot use syn_ramstyle = "uram" to infer RAM64X18_RT, since true dual-port mode is not supported. The synthesis software ignores this attribute and infers RAM1K18_RT.
- If your RTL code includes asynchronous memory, then you cannot use syn_ramstyle = "lsram" to infer RAM1K18_RT, since asynchronous memory is not supported. The synthesis software ignores this attribute and infers RAM64X18_RT.
- When you do not want to use RAM resources, apply the value of registers to the syn_ramstyle attribute on the RAM instance name or signal driven by the RAM.
- The syn_ramstyle attribute supports the values of no_rw_check and rw_check. By default, the synthesis tool does not generate glue logic for read/write address collision. Use syn_ramstyle="rw_check" to insert glue logic for read write address collision. Use syn_ramstyle="no_rw_check" to prevent glue logic insertion. See Read/Write Address Collision Check, on page 66.

Read/Write Address Collision Check

The synthesis software does not perform read/write address collision check when RAM is inferred. The tool does not insert glue logic around RAM to prevent the read/write address collision during write operation.

If read and write to the same address occurs simultaneously in your design, use syn_ramstyle="rw_check" or enable the Read Write Check on RAM option on the Implementation Options Device panel, to perform read/write address collision check. The tool then inserts glue logic around the RAM to prevent read/write address collision during the write operation, while retaining the RTL behavior.

The different modes include:

- Write-first mode—write operations precede read when a collision occurs. Data is first written into memory and then the same data is read. Since RTG4 doesn't support write-first mode for RAM, any RAM with write-first mode in RTL is mapped using no-change mode of RAM1K18_RT.
- Read-first mode—since RTG4 doesn't support read-first mode, any RAM with read-first mode is implemented using the no-change mode of RAM1K18_RT with the Read Write option set to OFF.
- No change mode—output of the RAM does not change when a collision occurs.

Note the following, if the Read Write Check on RAM option is enabled:

- Since RTG4 RAM1K18_RT architecture does not support read-before-write mode, any RAM with read-first mode is mapped to URAM or registers for Single Port RAMs.
- When a single-port RAM with write-first mode is mapped to RAM1K18_RT in no-change mode, write-through glue logic is created around the RAM.
- For no-change mode, no glue logic is created because the RAM output does not change when a collision occurs.
- For simple dual-port and true dual-port RAM in write-first mode, glue logic is created and RAM is inferred in no-change mode. Glue logic is also created for single-port RAM when it is mapped to RAM64K18_RT.
- If read/write check creates glue logic, then the pipeline register cannot be packed into the block RAM.

RAM Uses BLK Signal to Reduce Power Consumption

By default, the tool fractures wide RAMs by splitting the data width to improve timing.

From Synplify® Pro 2014.09M-SP2 onwards, the tool is enhanced to use the BLK pin of the RAM for reducing power consumption by fracturing wide RAMs on the address width.

To enable this feature, set global option low_power_ram_decomp 1 in the project file (*.prj). The tool uses this option to fracture wide RAMs on the address width to infer RAM in low power mode. The tool uses the BLK pin to select a RAM for a particular address and OR gates at the output to select the output from RAM blocks.

The control of individual RAM inference to turn on or turn off low power mode is supported through the synthesis attribute syn_ramstyle.

Add syn_ramstyle = "low_power" to turn on low power inference if the global option is set to off.
Add syn_ramstyle = "no_low_power" to turn off low power inference if the global option is on.

Write Byte-Enable Support for RAM

In case of RAM with n write enables to control writing of data into memory locations, the Synplify Pro compiler creates n sub-instances of RAMs with different write enables. The Synplify Pro mapper merges these multiple RAM blocks into single or multiple block RAMs, depending on the threshold and number of write-enables.

The write byte-enable (A_WEN/B_WEN [1:0]) pin of block RAM primitives is configured to control the write operation in block RAMs.

RAMINDEX Property Switch

To disable generation of the RAMINDEX property in RAM1K18_RT and RAM64X18_RT RAM blocks, add the `disable_ramindex` switch to the project file (prj):

```
Usage (disable RAMINDEX): set_option  -disable_ramindex 1
```

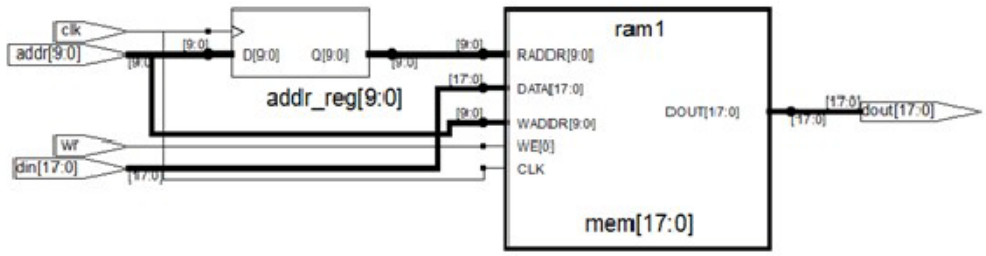
Coding Style Examples

Here are examples of inferring RAM1K18_RT and RAM64X18_RT RAM blocks.

Example 1: Single-Port RAM – RAM1K18_RT (Write-First Mode)

The following design is a single-port RAM with synchronous read and write operation. The same address is used for read and write operations in the write-first mode.

```
module ram_singleport_addreg (clk,wr,addr,din,dout);
input clk;
input [17:0] din;
input wr;
input [9:0] addr;
output [17:0] dout;

reg [9:0] addr_reg;
reg [17:0] mem [0:1023] ;
assign dout = mem[addr_reg]; always@(posedge clk)
begin
   addr_reg <= addr;
      if(wr)
         mem[addr]<= din;
end
endmodule
```



Since Libero does not support RAM in write-first mode, the synthesis tool infers RTG4 RAM1K18_RT in no-change mode. In this case, RTL is written to infer RAM in write-first mode, but the tool infers RAM in no-change mode. By default, the Automatic Read/Write Check insertion for RAM is OFF and address collision detection logic is not implemented, so there could be a simulation mismatch.

If there is a simulation mismatch, you can turn ON the Automatic Read/Write Check insertion for RAM UI option under Implementation Options or you can apply the attribute syn_ramstyle="rw_check" on the RAM instance and the tool will add address collision detection logic.

Resource Usage Report for ram_singleport_addreg

This section of the log file (.srr) shows resource usage details.

Mapping to part: rt4g150cg1657-1

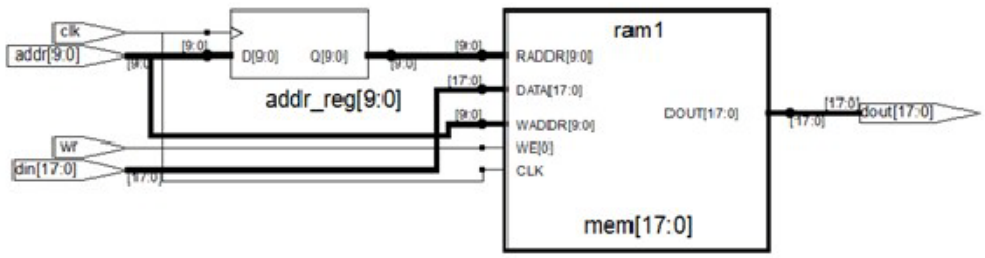
Block Rams (RAM1K18_RT): 1

Sequential Cells:
SLE 0uses

Example 2: Single-Port RAM – RAM64X18_RT (Write-First Mode)

The following design is a single-port RAM with synchronous read and write operation. The same address is used for read and write operation in the write-first mode.

```
module ram_singleport_addreg (clk,wr,addr,din,dout);

input clk;
input [17:0] din;
input wr;
input [5:0] addr;

output [17:0] dout;

reg [5:0] addr_reg;
reg [17:0] mem [0:64] ;

assign dout = mem[addr_reg];

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



The tool infers RTG4 RAM64X18_RT.

Resource Usage Report for ram_singleport_addreg

Mapping to part: rt4g150cg1657-1

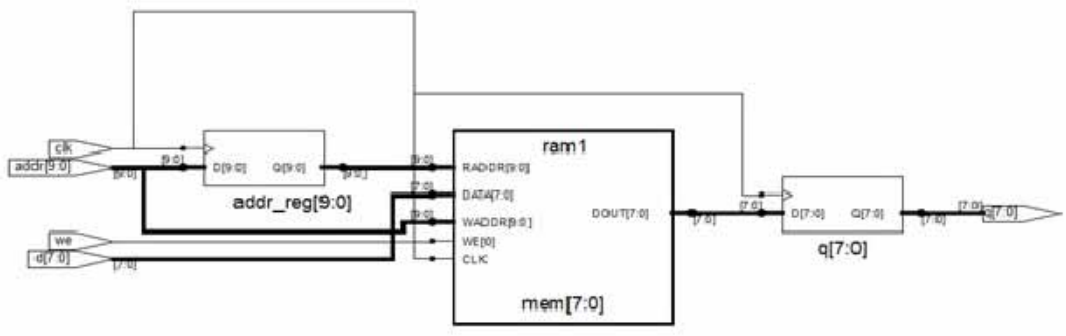
Cell usage:
CLKINT 1 use
RAM64x18_RT 1 use

Sequential Cells:
SLE 0 uses

Example 3: Single-Port RAM with Pipeline Register – RAM1K18_RT (Write-First Mode)

The following design is a single-port RAM with one pipeline register on the read port in the write-first mode.

```
module ram_singleport_pipereg (clk,we,addr,d,q);

input [7:0] d;
input [9:0] addr;
input clk, we;

reg [9:0] addr_reg;
output reg [7:0] q;

reg [7:0] mem [1023:0] ;

always @(posedge clk) begin
addr_reg <= addr;
if(we)
mem[addr] <= d;
end

always @ (posedge clk )
begin

q <= mem[addr_reg];
end
endmodule
```



Since RTG4 does not support write-first mode for RAM1K18_RT, the synthesis tool infers RAM1K18_RT in no-change mode. In this case, RTL is written to infer RAM in write-first mode, but the tool infers RAM in no-change mode. By default, the Automatic Read/Write Check insertion for RAM option is OFF so address collision detection logic is not implemented, due to which there could be a simulation mismatch.

In case of simulation mismatch, you can turn ON the Automatic Read/Write Check insertion for RAM UI option under Implementation Options or you can apply the attribute syn_ramstyle="rw_check" on the RAM instance and the tool will add address collision detection logic.

Resource Usage Report for ram_singleport_pipereg

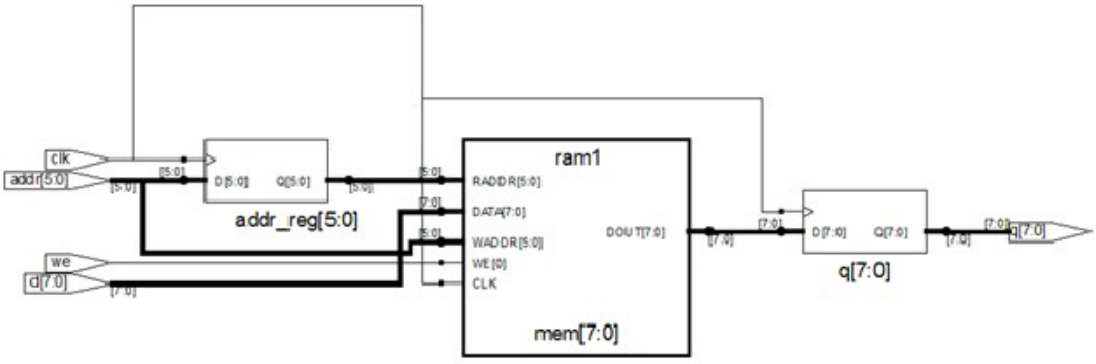
Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 1 use
RAM1K18_RT 1 use

Sequential Cells:
SLE 0 uses

Example 4: Single-Port RAM with Pipeline Register – RAM64X18_RT (Write-First Mode)

The following design is a single-port RAM with one pipeline register on the read port in the write-first mode.

```
module ram_singleport_pipereg(clk,we,addr,d, q);

input [7:0] d;
input [5:0] addr;
input clk, we;

reg [5:0] addr_reg;
output reg [7:0] q;

reg [7:0] mem [63:0] ;

always @(posedge clk) begin
addr_reg <= addr;
if(we)
mem[addr] <= d;
end

always @ (posedge clk )
begin

q <= mem[addr_reg];
end
endmodule
```



The tool infers RTG4 RAM64X18_RT.

Resource Usage Report for ram_singleport_pipereg

Mapping to part: rt4g150cg1657-1

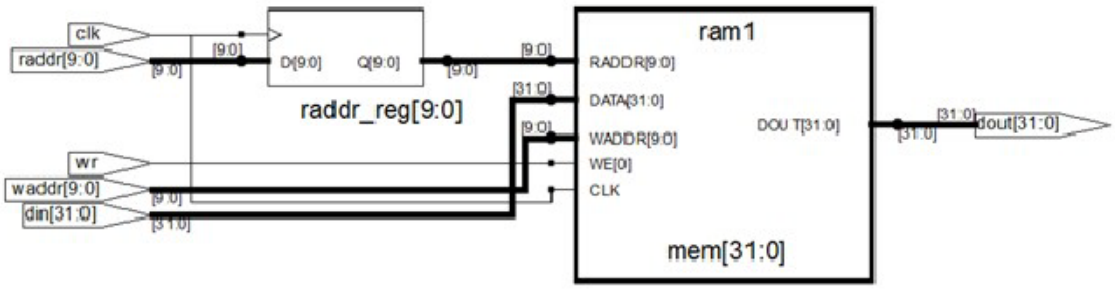
Cell usage:
CLKINT 1 use
RAM64x18_RT 1 use

Sequential Cells:
SLE 0 uses

Example 5: Simple Dual-Port RAM – RAM1K18_RT (Write-First Mode)

The following design is a simple dual-port (two port) RAM with synchronous read/write operation. Different read and write address are used in the write-first mode.

```
module ram_2port_raddreg(clk, wr, raddr, din, waddr, dout);

input clk;
input [31:0] din;
input wr;
input [9:0] waddr, raddr;

output [31:0] dout;

reg [9:0] raddr_reg;
reg [31:0] mem [0:1023];

assign dout = mem[raddr_reg];

always@ (posedge clk)
begin
   raddr_reg <= raddr;
      if (wr)
         mem[waddr] <= din;

end
endmodule
```



Since RTG4 does not support write-first mode for RAM1K18_RT, the synthesis tool infers RAM1K18_RT in no-change mode. In this case, RTL is written to infer RAM in write-first mode, but the tool infers RAM in no-change mode. By default, the Automatic Read/Write Check insertion for RAM option is OFF so address collision detection logic is not implemented, due to which there could be a simulation mismatch.

In case of simulation mismatch, you can turn ON the Automatic Read/Write Check insertion for RAM UI option under Implementation Options or you can apply the attribute syn_ramstyle="rw_check" on the RAM instance and the tool will add address collision detection logic.

Resource Usage Report for ram_2port_raddreg

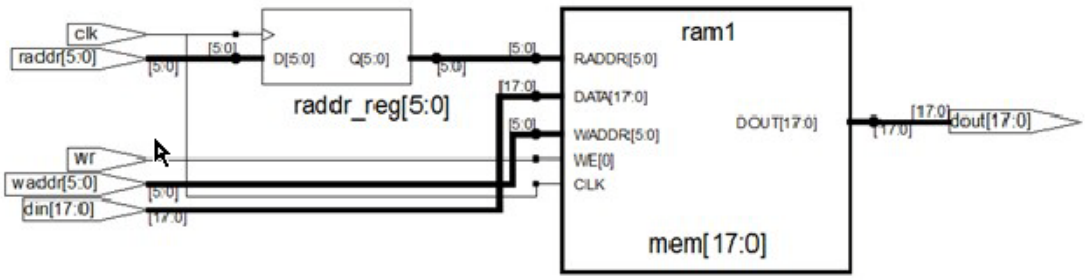
Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 1 use
RAM1K18_RT 2 uses

Sequential Cells:
SLE 0 uses

Example 6: Simple Dual-Port RAM – RAM64X18_RT (Write-First Mode)

The following design is a simple dual-port (two port) RAM with synchronous read/write operation. Different read and write addresses are used in the write-first mode.

```
module ram_2port_raddreg(clk, wr, raddr, din, waddr, dout);

input clk;
input [17:0] din;
input wr;
input [5:0] waddr, raddr;

output [17:0] dout;

reg [5:0] raddr_reg;
reg [17:0] mem [0:63];

assign dout = mem[raddr_reg];

always@(posedge clk)
begin
   raddr_reg <= raddr;
      if(wr)
         mem[waddr]<= din;

end
endmodule
```



The tool infers RTG4 RAM64X18_RT.

Resource Usage Summary for ram_2port_raddreg

Mapping to part: rt4g150cg1657-1

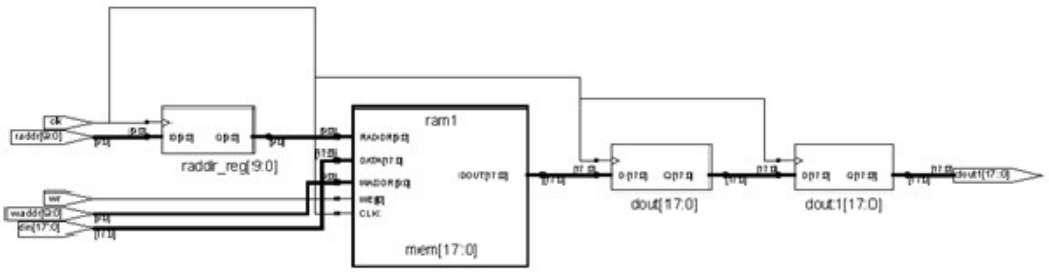
Cell usage:
CLKINT 1 use
RAM64x18_RT 1 use

Sequential Cells:
SLE 0 uses

Example 7: Simple Dual-Port RAM with Pipeline Register – RAM1K18_RT (Write-First Mode)

The following design is a simple dual-port (two port) RAM with synchronous read/write operation with pipeline register in write-first mode.

```
module ram_2port_pipe(clk,wr,raddr,din,waddr,dout1);

input clk;
input [17:0] din;
input wr;
input [9:0] waddr,raddr;

output [17:0] dout1 ;
reg [9:0] raddr_reg;
reg [17:0] mem [0:1023];
reg [17:0] dout, dout1;

always@(posedge clk)
begin
   raddr_reg <= raddr;
   dout <= mem[raddr_reg];
      if(wr)
         mem[waddr]     <= din;

end

always @(posedge clk)
begin

   dout1 <= dout;
end

endmodule
```



Since RTG4 does not support write-first mode for RAM1K18_RT, the synthesis tool infers RAM1K18_RT in no-change mode. In this case, RTL is written to infer RAM in write-first mode, but the tool infers RAM in no-change mode. By default, the Automatic Read/Write Check insertion for RAM option is OFF so address collision detection logic is not implemented, due to which there could be a simulation mismatch.

In case of simulation mismatch, you can turn ON the Automatic Read/Write Check insertion for RAM UI option under Implementation Options or you can apply the attribute syn_ramstyle="rw_check" on the RAM instance and the tool will add address collision detection logic.

Note: The output pipeline register dout1 is not packed into the RAM. Only register dout is packed in the RAM.

Resource Usage Report for ram_2port_pipe

Mapping to part: rt4g150cg1657-1

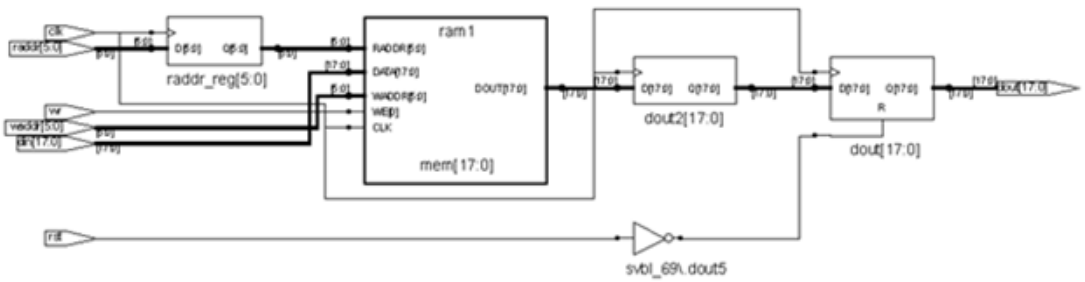
Cell usage:
CLKINT 1 use
RAM1K18_RT 1 use

Sequential Cells:
SLE 18 uses

Example 8: Simple Dual-Port RAM with Pipeline Register – RAM64X18_RT (Write-First Mode)

The following design is a simple dual-port (two port) RAM with synchronous read and write operation with pipeline register in write-first mode.

```
module ram_2port_pipe(clk,wr,raddr,din,waddr,dout, rst);

input clk;
input [17:0] din;
input wr, rst;
input [5:0] waddr,raddr;

output [17:0] dout;

reg [5:0] raddr_reg;
reg [17:0] mem [0:63];

reg [17:0] dout2, dout;
wire [17:0] dout1;

assign dout1 = mem[raddr_reg];

always@(posedge clk)
begin
   raddr_reg <= raddr;
      dout2 <= dout1;
      if(wr)
         mem[waddr] <= din;

end

always @(posedge clk or negedge rst)
begin
   if (~ rst )
      dout <= 8'b0;
   else
      dout <= dout2;
end
endmodule
```



The tool infers RTG4 RAM64X18_RT.

Note: The output pipeline register dout is not packed in the RAM.

Resource Usage Summary for ram_2port_pipe

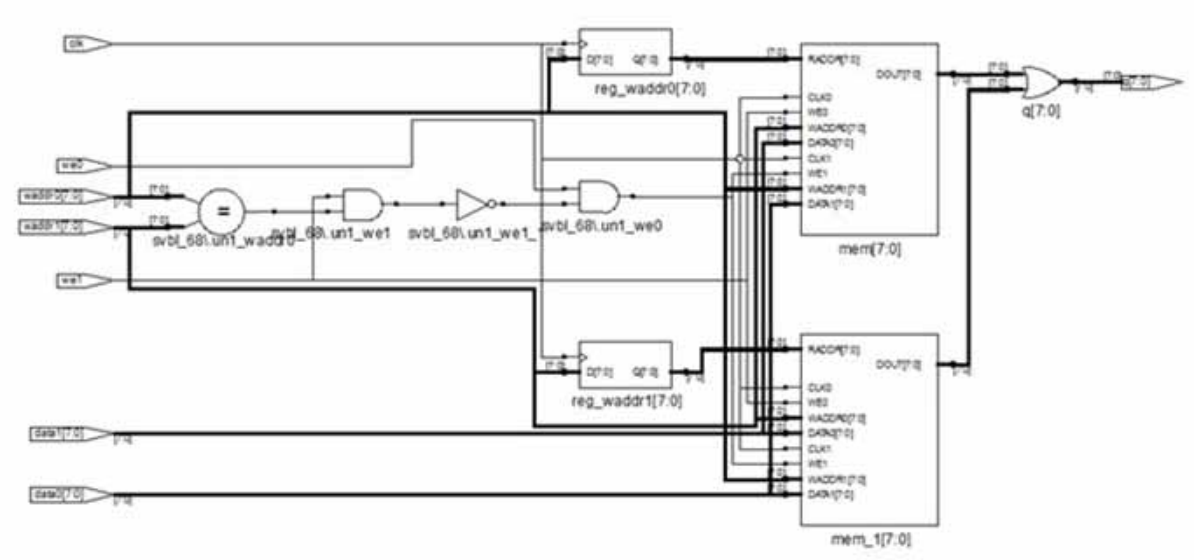
Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 2 uses
RAM64x18_RT 1 use

Sequential Cells:
SLE 18 uses

Example 9: True Dual-Port RAM (Single Clock)

The following design is a true dual-port RAM with two read and write ports and one clock.

```
module ram_dport_reg(data0,data1,waddr0, waddr1,we0,we1,clk,q);

parameter d_width = 8;
parameter addr_width = 8;
parameter mem_depth = 256;

input [d_width-1:0] data0, data1;
input [addr_width-1:0] waddr0, waddr1;
input we0, we1, clk;

output [d_width-1:0] q;

reg [d_width-1:0] mem [mem_depth-1:0];
reg [addr_width-1:0] reg_waddr0, reg_waddr1;

wire [d_width-1:0] q0, q1;

assign q = q0 | q1;

assign q0 = mem[reg_waddr0];
assign q1 = mem[reg_waddr1];

always @(posedge clk)

begin
if (we0)
mem[waddr0] <= data0;
if (we1)
mem[waddr1] <= data1;

reg_waddr0 <= waddr0;
reg_waddr1 <= waddr1;

end
endmodule
```



The tool infers RTG4 RAM1K18_RT.

Resource Usage Summary for ram_dport_reg

Mapping to part: rt4g150cg1657-1

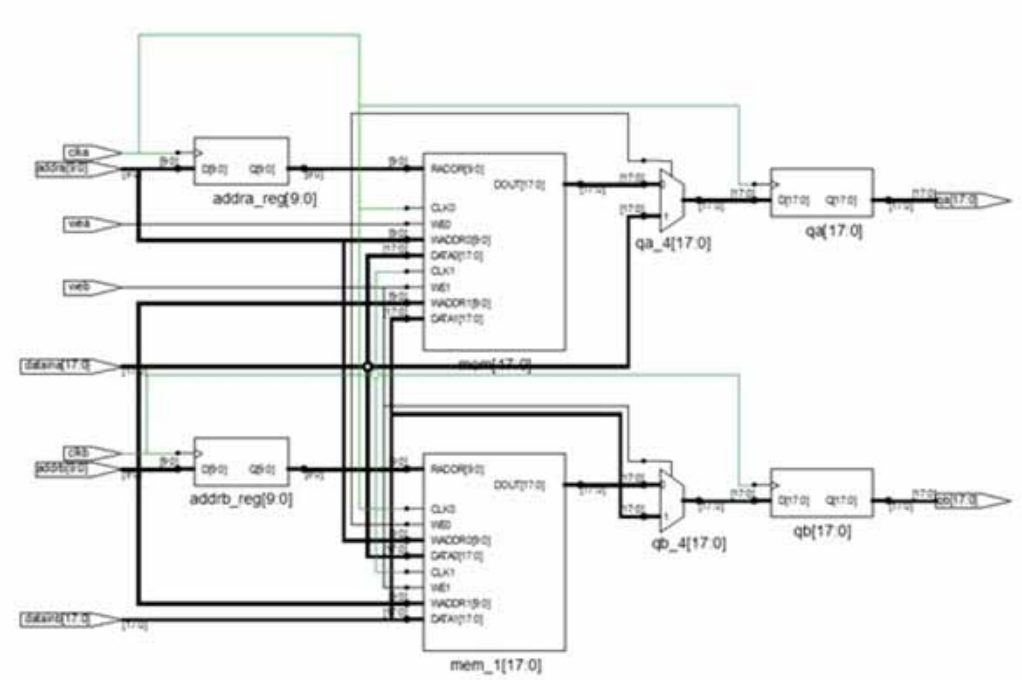
Cell usage:
CLKINT 1 use
RAM1K18_RT 1 use
CFG2 8 uses
CFG3 1 use
CFG4 5 uses

Sequential Cells:
SLE 0 uses

Example 10: True Dual-Port RAM (Multiple Clocks)

The following design is a true dual-port RAM with two read and writes ports and two clocks.

```
module test (clka,clkb,wea,addra,dataina,qa,web,addrb,datainb,qb);

parameter addr_width = 10;
parameter data_width = 18;

input clka,clkb,wea,web;

input [data_width - 1 : 0] dataina,datainb;
input [addr_width - 1 : 0] addra,addrb;

output reg [data_width - 1 : 0] qa,qb;

reg [addr_width - 1 : 0] addra_reg, addrb_reg;
reg [data_width - 1 : 0] mem [(2**addr_width) - 1 : 0] ;

always @ (posedge clka)
begin
   addra_reg <= addra;
   if(wea)
   mem[addra] <= dataina;
end

always @ (posedge clkb)
begin
   addrb_reg <= addrb;
   if(web)
   mem[addrb] <= datainb;
end

always @ (posedge clka)
begin
   if(~wea)
   qa <= mem[addra_reg];
   else qa <= dataina;
end

always @ (posedge clkb)
begin

   if(~web)
   qb <= mem[addrb_reg];
   else qb <= datainb;
end


endmodule
```



The tool infers RTG4 RAM1K18_RT with output registers qa and qb inferred outside the RAM using SLE's.

Resource Usage Summary for test

Mapping to part: rt4g150cg1657-1

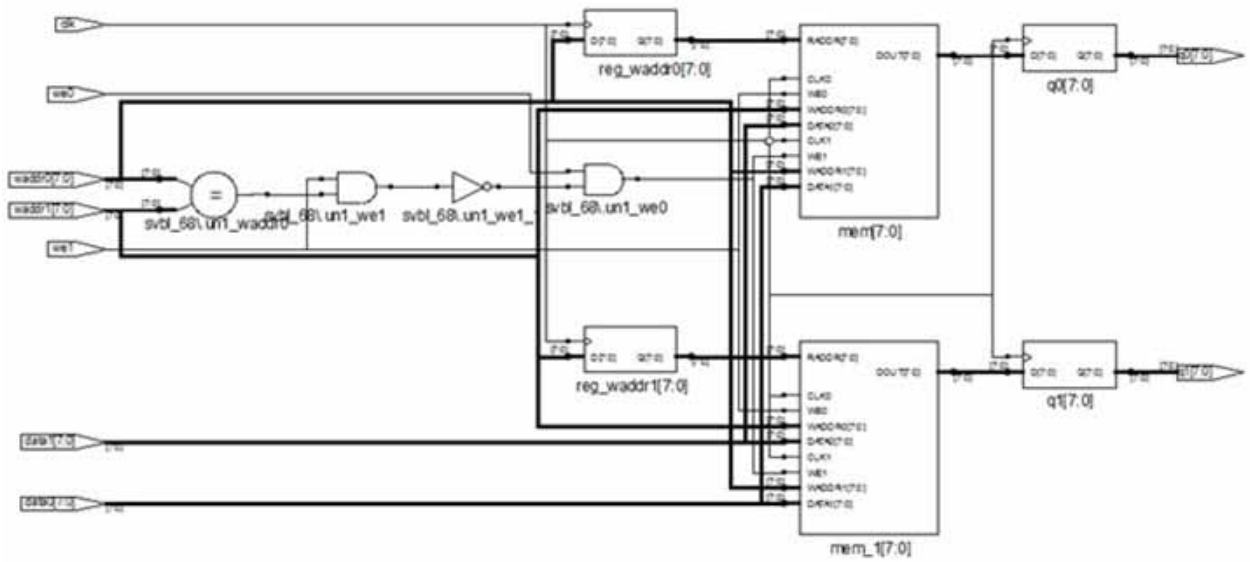
Cell usage:
CLKINT 2 use
RAM1K18_RT 1 use
CFG3 36 uses

Sequential Cells:
SLE 36 uses

Example 11: True Dual-Port RAM with Pipeline Register

The following design is a true dual-port RAM with two read and write ports and one clock with one pipeline register.

```
module ram_dport_reg(data0,data1,waddr0, waddr1,we0,we1,clk,q0, q1);

parameter d_width = 8;
parameter addr_width = 8;
parameter mem_depth = 256;

input [d_width-1:0] data0, data1;
input [addr_width-1:0] waddr0, waddr1;
input we0, we1, clk;

output [d_width-1:0] q0, q1;

reg [d_width-1:0] mem [mem_depth-1:0] ;
reg [addr_width-1:0] reg_waddr0, reg_waddr1;
reg [d_width-1:0] q;

reg [d_width-1:0] q0, q1;

always @(posedge clk)
begin
if (we0)
mem[waddr0] <= data0;
if (we1)
mem[waddr1] <= data1;

reg_waddr0 <= waddr0;
reg_waddr1 <= waddr1;

end

always @ (posedge clk)
begin
   q0 <= mem[reg_waddr0];
   q1 <= mem[reg_waddr1];
end
endmodule
```



The tool infers RTG4 RAM1K18_RT.

Resource Usage Summary for ram_dport_reg

Mapping to part: rt4g150cg1657-1

Cell usage:
CLKINT 1 use
RAM1K18_RT 1 use
CFG3 1 use
CFG4 5 uses

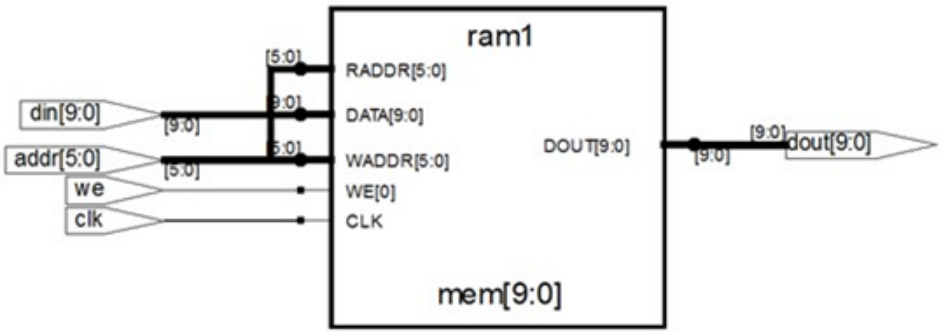
Sequential Cells:
SLE 0 uses

Example 12: Single-Port RAM (Asynchronous Read) RAM64X18_RT (Read-First Mode)

The following design is a single-port RAM with asynchronous read in read-first mode.

```
module test_RTL (dout, addr, din, we, clk);

parameter data_width = 10;
parameter address_width = 6;
parameter ram_size = 64;

output [data_width-1:0] dout;
input [data_width-1:0] din;
input [address_width-1:0] addr;
input we, clk;
```

```
reg [data_width-1:0] mem [ram_size-1:0];

always @(posedge clk) begin
   //register the address for read operation
   if(we) mem[addr] <= din;
   end

assign dout = mem[addr];

endmodule
```



The tool infers RTG4 RAM64X18_RT.

Resource Usage Summary for test_RTL

Mapping to part: rt4g150cg1657-1
Cell usage:
RAM64x18_RT 1 use

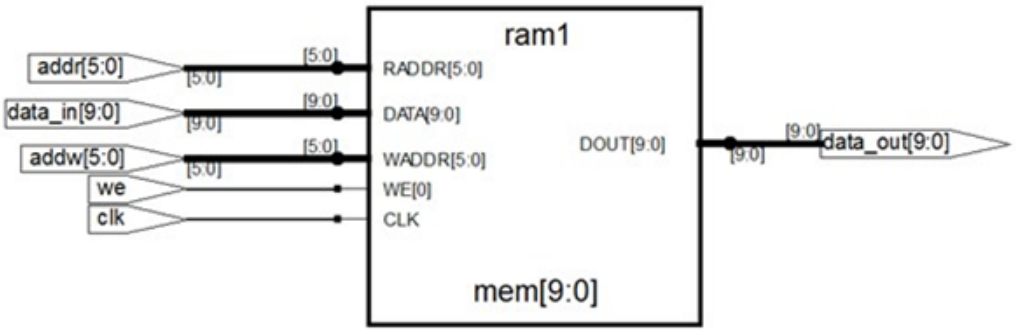
Sequential Cells:
SLE 0 uses

Example 13: Simple Dual-Port RAM (Asynchronous Read) RAM64X18_RT (Read- First Mode)

The following design is a simple dual-port RAM with asynchronous read in read-first mode.

```
module test_RTL (addr,addw, we, clk , data_out, data_in);

parameter ADDR_WIDTH = 6;
parameter ADDW_WIDTH = 6;
parameter DATA_WIDTH = 10;
parameter MEM_DEPTH = 64;

input [ADDR_WIDTH-1:0]addr;
input [ADDW_WIDTH-1:0]addw;
input clk, we;
input [DATA_WIDTH-1 : 0]data_in;
output [DATA_WIDTH-1 : 0]data_out;

reg [DATA_WIDTH-1 : 0]mem[MEM_DEPTH-1 :0] ;
```

```
assign data_out = mem[addr];
always @(posedge clk)
begin
   if (we)
   mem[addw] <= data_in;
end

endmodule
```



The tool infers RTG4 RAM64X18_RT.

Resource Usage Summary for test_RTL

Mapping to part: rt4g150cg1657-1

Cell usage:
RAM64x18_RT 1 use

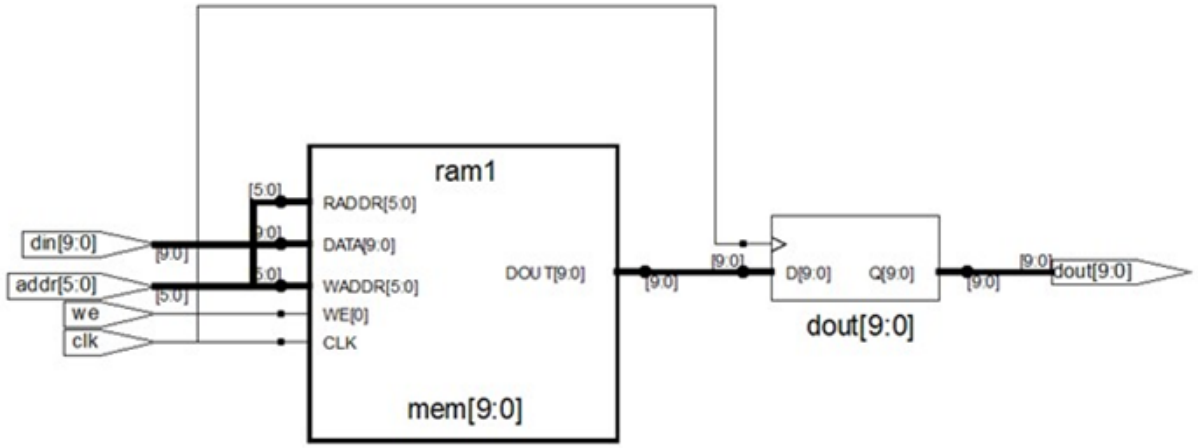
Sequential Cells:
SLE 0 uses

Example 14: Single-Port RAM (Asynchronous Read) with Pipeline Register RAM64X18_RT (Read-First mode)

The following design is a single-port RAM with asynchronous read and pipeline register at its output in read-first mode.

```
module test (dout, addr, din, we, clk);

parameter data_width = 10;
parameter address_width = 6;
parameter ram_size = 64;

output reg [data_width-1:0] dout;
input [data_width-1:0] din;
input [address_width-1:0] addr;
input clk;
input we;
wire[data_width-1:0] dout_net;
reg[data_width-1:0] mem [ram_size-1:0];
```

```
always @(posedge clk) begin
   if (we)
      mem[addr] <= din;
end
assign dout_net = mem[addr];

always @(posedge clk) begin
   dout <= dout_net;
end

endmodule
```



The tool infers RTG4 RAM64X18_RT.

Resource Usage Summary for test

Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 1 use
RAM64x18_RT 1 use

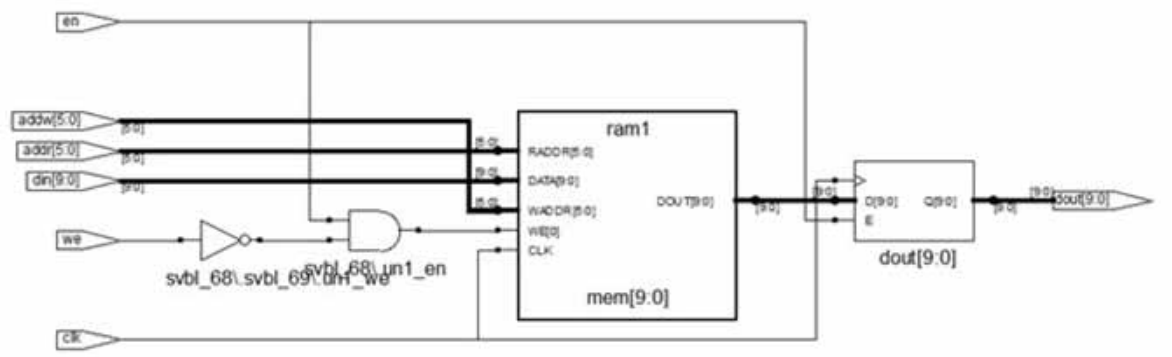
Sequential Cells:
SLE 0 uses

Example 15: Simple Dual-Port RAM (Asynchronous Read) and Pipeline Register with Clock Enable RAM64X18_RT (No Change Mode)

The following design is a simple dual-port RAM with asynchronous read and output pipeline register with clock enable.

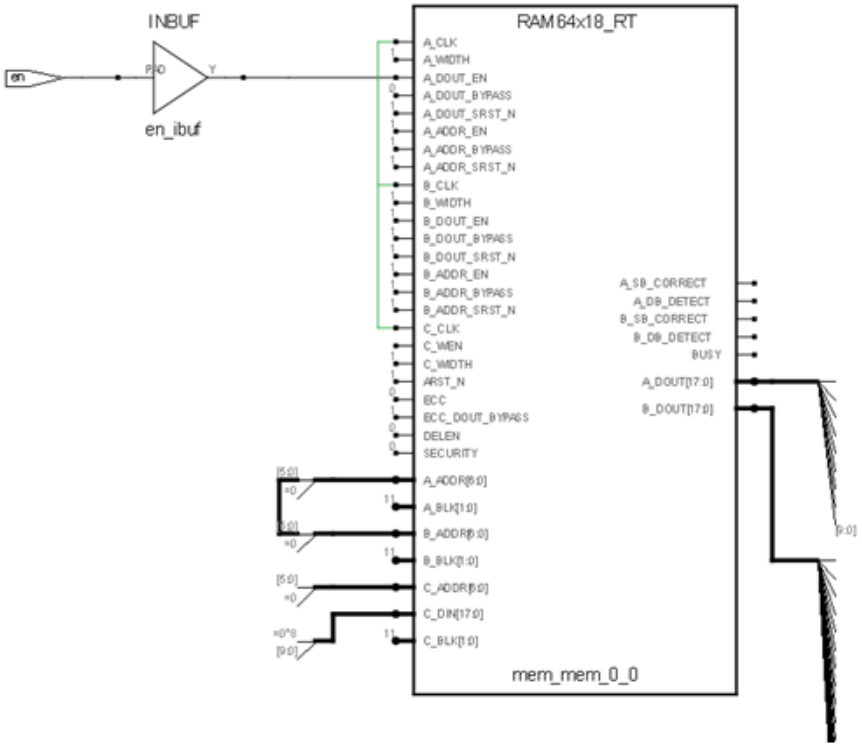
```
module test_RTL (dout, addr, din, we, clk, en, addw);

parameter data_width = 10;
parameter address_width = 6;
parameter ram_size = 64;

output reg [data_width-1:0] dout;
input [data_width-1:0] din;
input [address_width-1:0] addr, addw;
input we, clk, en;
reg [data_width-1:0] mem [ram_size-1:0];
wire [data_width-1:0] dout_reg ;

assign dout_reg = mem[addr];
always @(posedge clk) begin
   if (en) begin
      if(!we)
         mem[addw] <= din;
   end
end

always @(posedge clk) begin

   if (en) begin
      dout <= dout_reg;
   end
end
endmodule
```

SRS (RTL) View



The tool infers RTG4 RAM64X18_RT with enable en packing.

SRM (Technology) View



Resource Usage Summary for test_RTL

Mapping to part: rt4g150cg1657-1

Cell usage:
CLKINT 1 use
RAM64x18_RT 1 use
CFG2 1 use

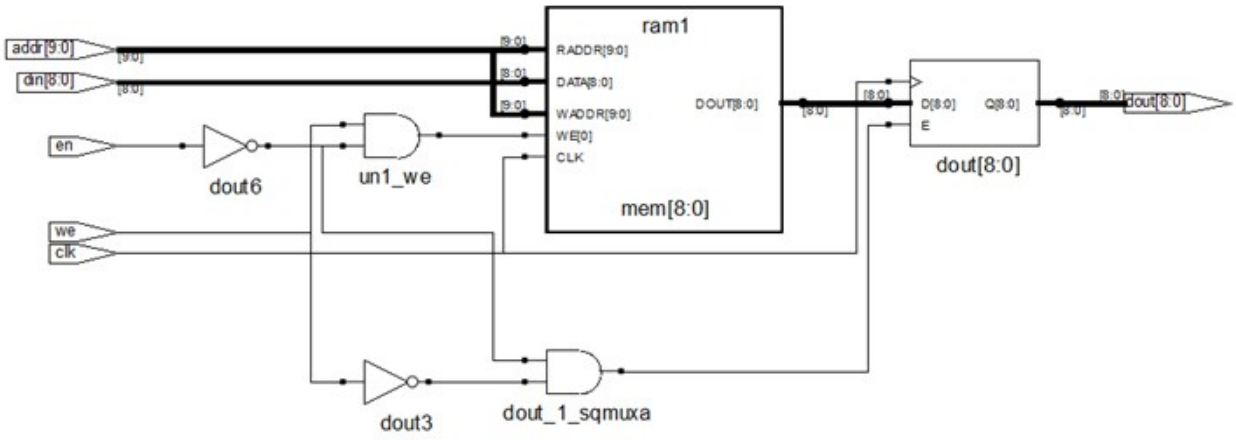
Sequential Cells:
SLE 0 uses

Example 16: Single-Port RAM RAM1K18_RT (No Change Mode)

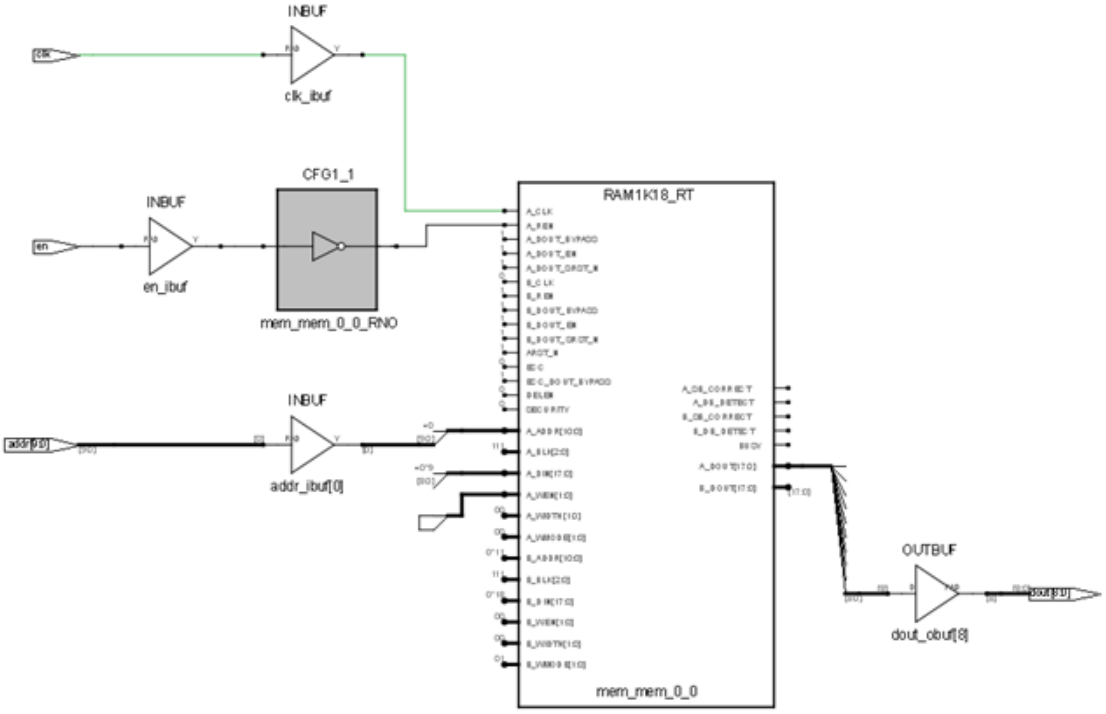
The following design is a single-port RAM with no change mode.

```
module test_RTL (dout, addr, din, we, clk, en);

parameter data_width = 9;
parameter address_width = 10;
parameter ram_size = 1024;

output reg [data_width-1:0] dout;
input [data_width-1:0] din;
input [address_width-1:0] addr;

input we, clk, en;

reg [data_width-1:0] mem [ram_size-1:0];

always @(posedge clk)

   begin
      if (!en)
      begin
         if(we)
         mem[addr] <= din;
         else
            dout <= mem[addr];
   end
end
endmodule
```

SRS (RTL) View



SRM (Technology) View



The tool infers RTG4 RAM1K18_RT without glue logic for read/write collision check with enable en packing.

Resource Usage Summary for test_RTL

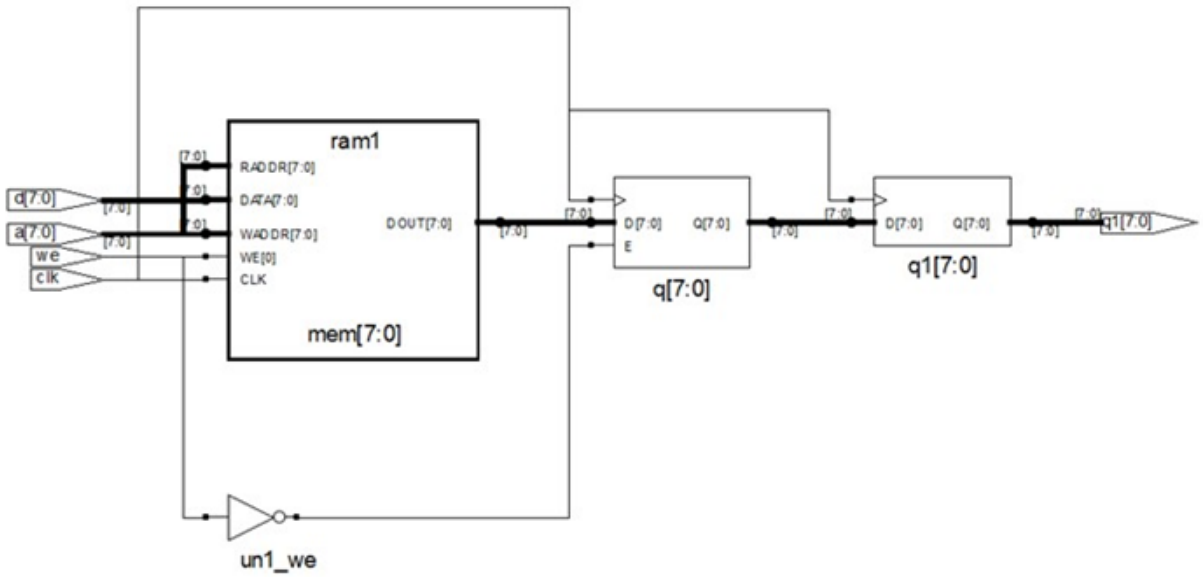
Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 1 use
RAM1K18_RT 1 use
CFG1 1 use
CFG2 1 use

Sequential Cells:
SLE 0 uses

Example 17: Single-Port RAM with One Pipelined Register on the Read Port (Sync-Sync) (No Change Mode)

The following design is a single-port RAM with one pipeline register on the read port using no change mode.

```
module ram_singleport_pipereg(clk,we,a,d,q1);

input [7:0] d;
input [7:0] a;
input clk, we;

reg [7:0] q;
output [7:0] q1;
reg [7:0] q1;
reg [7:0] mem [255:0];

always @(posedge clk)
   begin
      if(we)
         mem[a] <= d;
      else
         q <= mem[a];
   end
always @ (posedge clk)
   begin
      q1 <= q;
   end
endmodule
```



The tool infers RTG4 RAM64X18_RT. The pipeline register is mapped outside the RAM along with logic for no change mode with enable en packing.

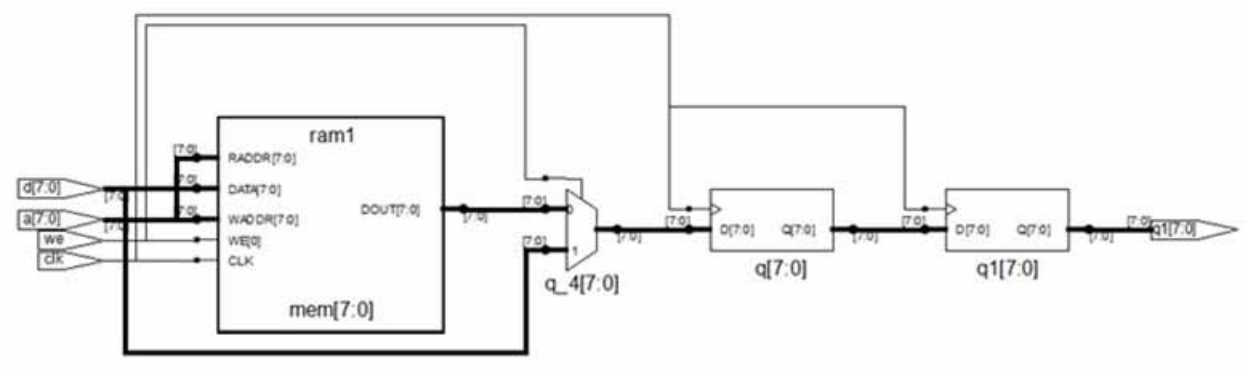
Resource Usage Summary for ram_singleport_pipereg

```
Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT      1 use
RAM64x18_RT     2 uses
CFG1     1 use
CFG2     2 use
CFG3     8 uses

Sequential Cells:
SLE9 uses
```

Example 18: Single-Port RAM with One Pipelined Register on the Read Port (Sync-Sync)

The following design is a single-port RAM with one pipeline register on the read port using write-first mode.

```
module ram_singleport_pipereg(clk,we,a,d,q1);

input [7:0] d;
input [7:0] a;
input clk, we;

reg [7:0] q;
output [7:0] q1;

reg [7:0] q1;
reg [7:0] mem [255:0];

always @(posedge clk)
   begin
      if(we)
         mem[a] <= d;
   end
always @ (posedge clk)
   begin
      if(we)
         q <= d;
      else
         q <= mem[a];
         q1 <= q;
   end
endmodule
```



The tool infers two RTG4 RAM64X18_RT.

Resource Usage Summary for ram_singleport_pipereg

Mapping to part: rt4g150cg1657-1

Cell usage:
CLKINT 1 use
CFG2 2 uses
CFG3 8 uses
RAM64x18_RT 2 uses

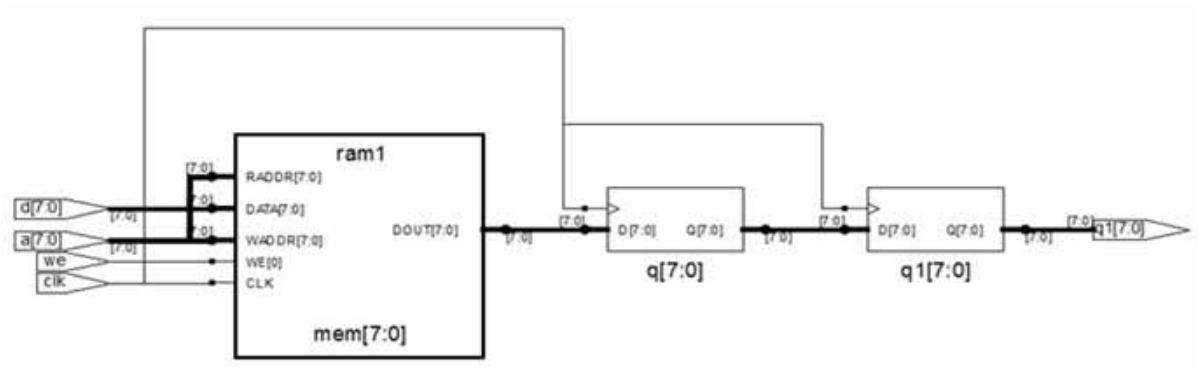
Sequential Cells:
SLE 9 uses

Example 19: Single-Port RAM with One Pipelined Register on the Read Port (sync-sync)

The following design is a single-port RAM with one pipeline register on the read port.

```
module ram_singleport_pipereg(clk,we,a,d,q1);

input [7:0] d;
input [7:0] a;
input clk, we;

reg [7:0] q;
output [7:0] q1;

reg [7:0] q1;
reg [7:0] mem [255:0];

always @(posedge clk)
```

```
begin
      if(we)
         mem[a] <= d;
   end
always @ (posedge clk)
   begin
      q <= mem[a];
      q1 <= q;
   end
endmodule
```



The tool infers two RTG4 RAM64X18_RT blocks.

Resource Usage Summary for ram_singleport_pipereg

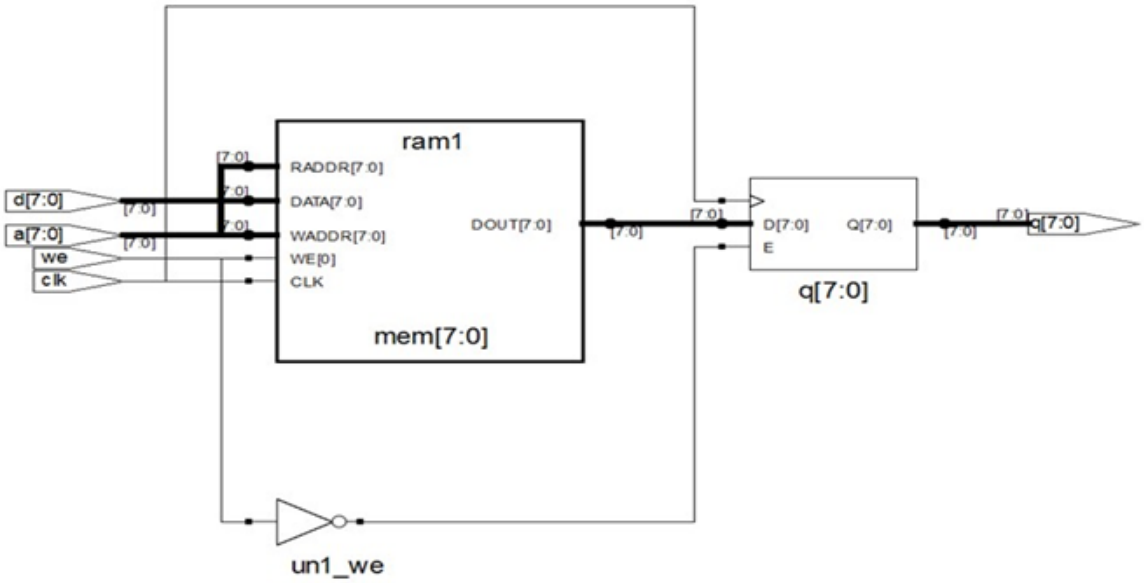
Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 1 use
RAM64x18_RT 2 uses
CFG2 2 uses
CFG3 8 uses

Sequential Cells:
SLE 9 uses

Example 20: Single-Port RAM with Synchronous Read Without Pipeline Register (Sync-Async) (No Change Mode)

The following design is a single-port RAM with synchronous read without pipeline register using no change mode.

```
module ram_singleport_pipereg(clk,we,a,d,q);
input [7:0] d;
input [7:0] a;
input clk, we;
output [7:0] q;
reg [7:0] q;
reg [7:0] mem [255:0];

always @(posedge clk)

   begin
      if(~we)
         mem[a] <= d;
      else
         q <= mem[a];
   end
endmodule
```



The tool infers two RTG4 RAM64X18_RT along with logic for no change mode with enable en packing.

Resource Usage Summary for ram_singleport_pipereg

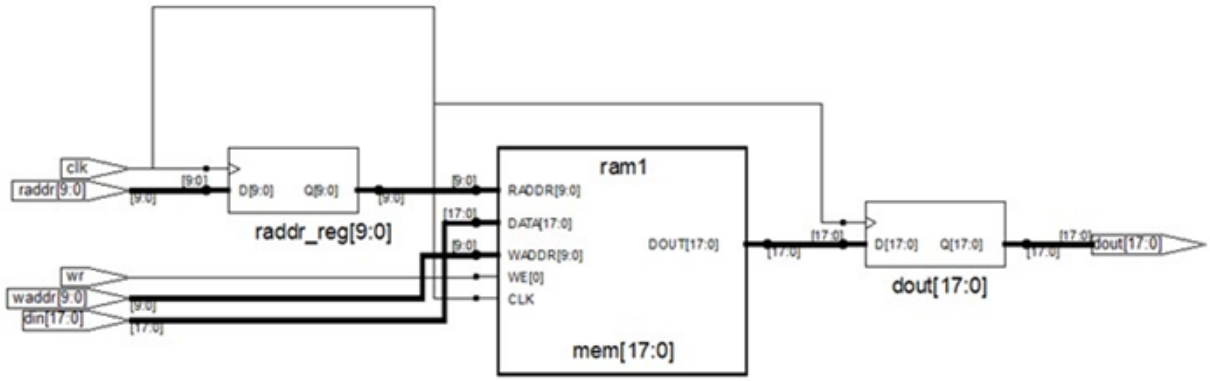
Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 1 use
RAM64x18_RT 2 uses
CFG2 2 uses
CFG3 8 uses

Sequential Cells:
SLE 1 use

Example 21: Simple Dual-Port RAM with Output Register

The following design is a single-port RAM with output register.

```
module ram_2port_pipe(clk,wr,raddr,din,waddr,dout);

input clk;
input [17:0] din;
input wr;
input [9:0] waddr,raddr;

output [17:0] dout;

reg [9:0] raddr_reg;
reg [17:0] mem [0:1023];
reg [17:0] dout;

always@(posedge clk)
   begin
      raddr_reg <= raddr;
      dout <= mem[raddr_reg];
      if(wr)
         mem[waddr]<= din;

   end
endmodule
```



The tool infers RTG4 RAM1K18_RT.

Resource Usage Summary for ram_2port_pipe

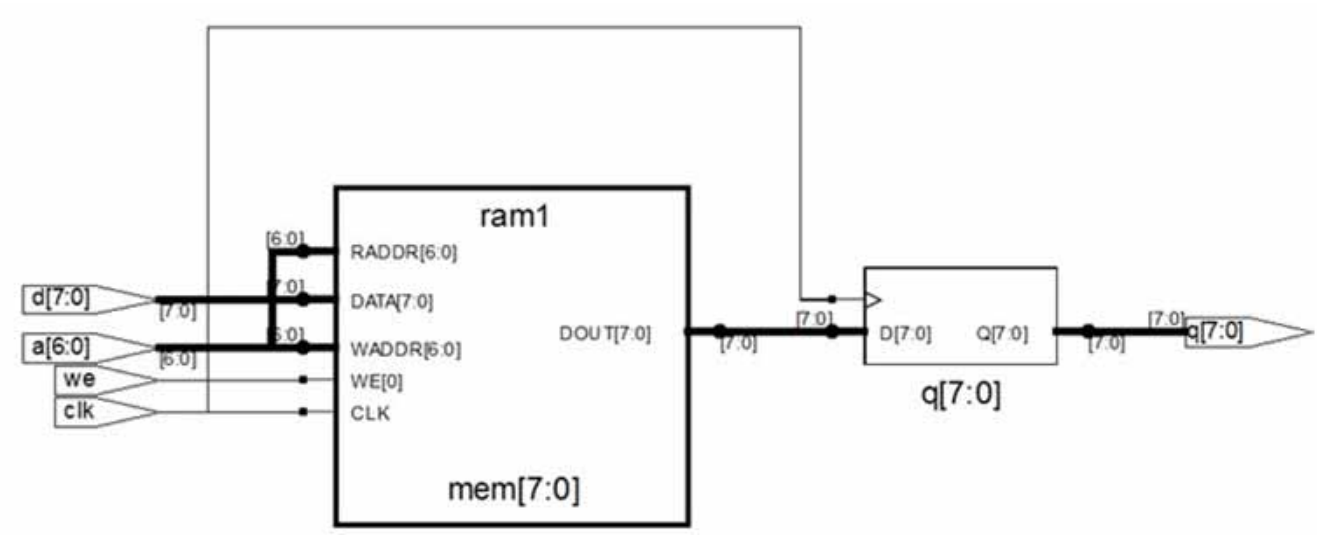
Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 1 use
RAM1K18_RT 1 use

Sequential Cells:
SLE 0 uses

Example 22: Single-Port RAM with Output Registers (VHDL)

The following design is a single-port RAM with output registers.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all; entity ram_singleport_outreg is
port (d: in std_logic_vector(7 downto 0);
   a: in integer range 127 downto 0;
   we: in std_logic;
   clk: in std_logic;
   q: out std_logic_vector(7 downto 0) );
end ram_singleport_outreg;

architecture rtl of ram_singleport_outreg is
type mem_type is array (127 downto 0) of
std_logic_vector (7 downto 0);
signal mem: mem_type;
begin
   process(clk)
   begin
      if (clk'event and clk='1') then
            q <= mem(a);
            if (we='1') then
            mem(a) <= d;
            end
         if; end if;
end process; end rtl;
```



The tool infers RTG4 RAM64X18_RT.

Resource Usage Summary for ram_singleport_outreg

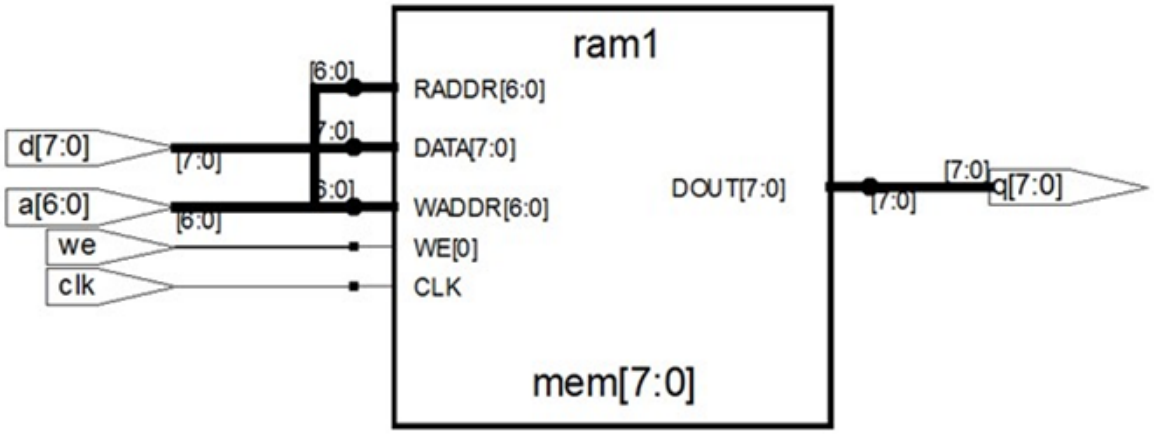
Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 1 use
RAM64x18_RT 1 use

Sequential Cells:
SLE 0 uses

Example 23: Single-Port RAM with Asynchronous Read (VHDL)

The following design is a single-port RAM with asynchronous read.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ram_singleport_noreg is
   port (d : in std_logic_vector(7 downto 0);
   a : in std_logic_vector(6 downto 0);
   we : in std_logic;
   clk : in std_logic;
   q : out std_logic_vector(7 downto 0) );
end ram_singleport_noreg;

architecture rtl of ram_singleport_noreg is
type mem_type is array (127 downto 0) of
   std_logic_vector (7 downto 0);
signal mem: mem_type;
begin process

(clk)
   begin
      if rising_edge(clk) then
         if (we = '1') then
            mem(conv_integer (a)) <= d;
      end if;
   end if;
end process;
q <= mem(conv_integer (a));
end rtl;
```



The tool infers RTG4 RAM64X18_RT.

Resource Usage Summary for ram_singleport_noreg

Mapping to part: rt4g150cg1657-1

Cell usage:
RAM64x18_RT 1 use

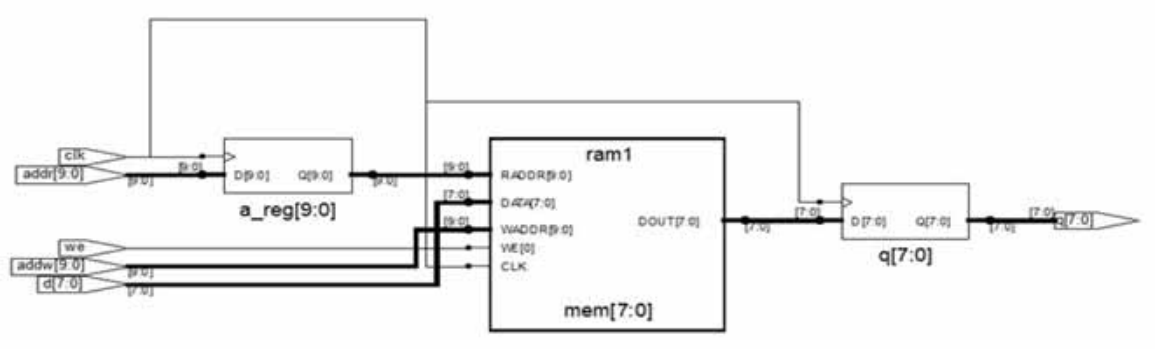
Sequential Cells:
SLE 0 uses

Example 24: Simple Dual-Port RAM with Output Register and Read Address Register (VHDL)

The following design is a simple dual-port RAM with output and read address registers.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity ram_simpledualport_outreg is
   port (d: in std_logic_vector(7 downto 0);
   addr: in integer range 1023 downto 0;
   addw: in integer range 1023 downto 0;
   we: in std_logic;
   clk: in std_logic;
   q: out std_logic_vector(7 downto 0) );
end ram_simpledualport_outreg;
```

```
architecture rtl of ram_simpledualport_outreg is
type mem_type is array (1023 downto 0) of std_logic_vector (7 downto 0);
signal mem: mem_type;
signal a_reg : integer range 1023 downto 0;
begin

   process (clk)
      begin
      if (clk'event and clk='1' ) then
         a_reg <= addr;
      end if;
   end process;

   process(clk)
      begin
         if (clk'event and clk='1') then
            q <= mem(a_reg);
               if (we='1') then
                   mem(addw) <= d;
               end if;
         end if;
   end process;
end rtl;
```



The tool infers RTG4 RAM1K18_RT.

Resource Usage Summary for ram_simpledualport_outreg

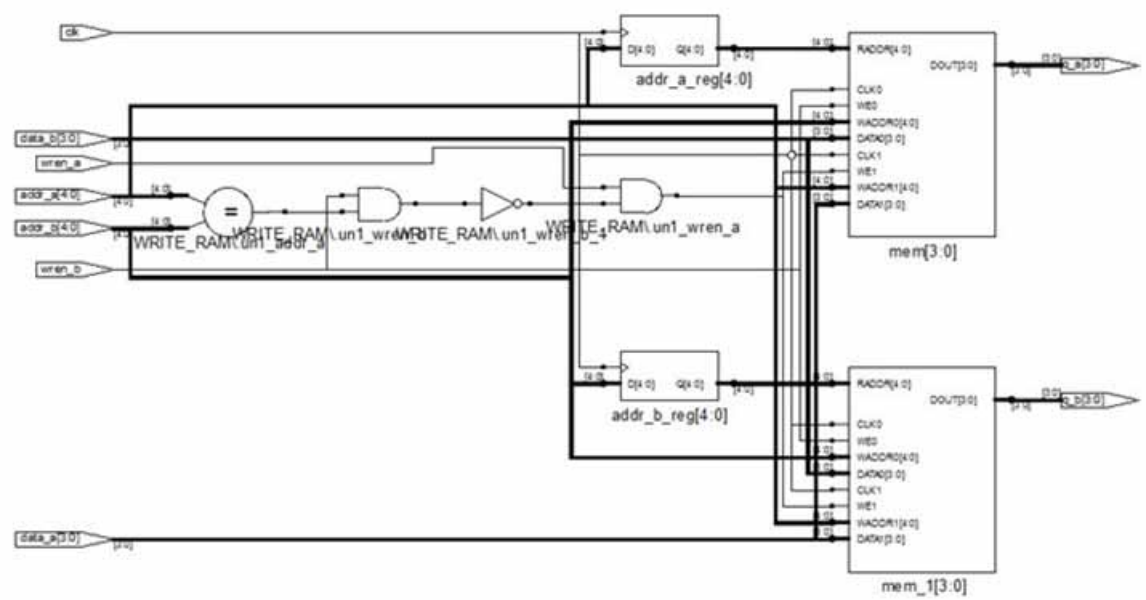
Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 1 use
RAM1K18_RT 1 use

Sequential Cells:
SLE 0 uses

Example 25: True Dual-Port RAM with Read Address Register (VHDL)

The following design is a true dual-port RAM with read address register.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity ram_dp_reg is
generic (data_width : integer := 4;
address_width :integer := 5 );
port (data_a:in std_logic_vector(data_width-1 downto 0);
data_b:in std_logic_vector(data_width-1 downto 0);
addr_a:in std_logic_vector(address_width-1 downto 0);
addr_b:in std_logic_vector(address_width-1 downto 0);
wren_a:in std_logic;
wren_b:in std_logic;
clk:in std_logic;
q_a:out std_logic_vector(data_width-1 downto 0);
q_b:out std_logic_vector(data_width-1 downto 0) );
end ram_dp_reg;

architecture rtl of ram_dp_reg is
type mem_array is array(0 to 2**(address_width) -1) of
std_logic_vector(data_width-1 downto 0);
signal mem : mem_array;

signal addr_a_reg : std_logic_vector(address_width-1 downto 0);
signal addr_b_reg : std_logic_vector(address_width-1 downto 0);
begin
   WRITE_RAM : process (clk)
      begin
         if rising_edge(clk) then
            if (wren_a = '1') then
               mem(to_integer(unsigned(addr_a))) <= data_a;
            end if;
               if (wren_b='1') then
                   mem(to_integer(unsigned(addr_b))) <= data_b;
         end if;
            addr_a_reg <= addr_a;
            addr_b_reg <= addr_b;
      end if;
end process WRITE_RAM;

q_a <= mem(to_integer(unsigned(addr_a_reg)));
q_b <= mem(to_integer(unsigned(addr_b_reg)));

end rtl;
```



The tool infers RTG4 RAM1K18_RT.

Resource Usage Summary for ram_dp_reg

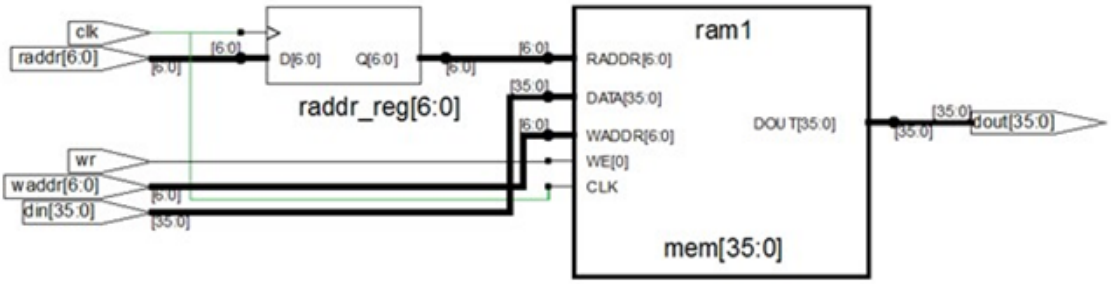
Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 1 use
RAM1K18_RT 1 use
CFG3 1 use
CFG4 3 uses

Sequential Cells:
SLE 0 uses

Example 26: Simple Dual-Port (Two-Port) RAM with Read Address Register (512 x 36 Configurations)

The following design is a simple dual-port RAM with read address register.

```
module ram_2port_addreg_512x36(clk,wr,raddr,din,waddr,dout);

input clk;
input [35:0] din;
input wr;
input [6:0] waddr,raddr;

output [35:0] dout;

reg [6:0] raddr_reg;
reg [35:0] mem [0:511];
wire [35:0] dout;

assign dout = mem[raddr_reg];

always@(posedge clk)
begin
      raddr_reg <= raddr;
      if(wr)
      mem[waddr]<= din;

end
endmodule
```



The FPGA synthesis tool infers RTG4 RAM1K18_RT.

Resource Usage Report for ram_2port_addreg_512x36

Mapping to part: rt4g150cg1657-1

Cell usage:
CLKINT 1 use
RAM1K18_RT 1 use

Sequential Cells:
SLE 0 uses

Example 27: True Dual-Port RAM with Output Registered, Pipelined, and Non-pipelined Version (VHDL)

The following design is a true dual-port RAM with output registered, pipelined, and non-pipelined version.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity RAM_inference_examples is
      generic (data_width     :integer := 32;
      addr_width     :integer := 10;
      depth       :integer := 1024;
      testcase    :integer := 1); --- change to 1,2,3 to use variations in coding
style of Dual port RAM
   port(clk        :in std_logic;
            reset_n    :in std_logic;
            re_n       :in std_logic;
            we_n       :in std_logic;
            data_in    :in std_logic_vector(data_width-1 downto 0);
            data_out   :out std_logic_vector(data_width-1 downto 0);
            addr_0     :in std_logic_vector(addr_width-1 downto 0);
            addr_1     :in std_logic_vector(addr_width-1 downto 0);
            r_wen_0    :in std_logic;
            r_wen_1    :in std_logic;

            data_in_0   :in std_logic_vector(data_width-1 downto 0);
            data_out_0  :out std_logic_vector(data_width-1 downto 0);
            data_in_1   : in std_logic_vector(data_width-1 downto 0);
            data_out_1  :out std_logic_vector(data_width-1 downto 0)
   );

end RAM_inference_examples;

architecture DEF_ARCH of RAM_inference_examples is
type mem_type is array (depth-1 downto 0) of std_logic_vector (data_width-1 downto
0);
signal BRAM_store   :mem_type;
signal int_addr_0   :integer range 0 to 4096;
signal int_addr_1   :integer range 0 to 4096;
signal rd_addr    :integer range 0 to 4096;
signal wr_addr    :integer range 0 to 4096;
signal data_out_tmp   :std_logic_vector(data_width-1 downto 0);
signal data_out_0tmp  :std_logic_vector(data_width-1 downto 0);
signal data_out_1tmp :std_logic_vector(data_width-1 downto 0);

begin
```

Case 1 - Dual-port without pipelining (registered data_out ports)

```
case_num1 : if testcase = 1 generate

int_addr_0 <= CONV_INTEGER(addr_0);
int_addr_1 <= CONV_INTEGER(addr_1);

process(clk)
begin
   if rising_edge(clk) then
   -- port 0
         if (r_wen_0 = '0') then
            BRAM_store(int_addr_0)   <= data_in_0;
else
   data_out_0      <= BRAM_store(int_addr_0);
end if;
   -- port 1
         if (r_wen_1 = '0') then
            BRAM_store(int_addr_1)   <= data_in_1;

         else
            data_out_1 <= BRAM_store(int_addr_1);
            end if;
      end if;
   end process;

end generate;
```

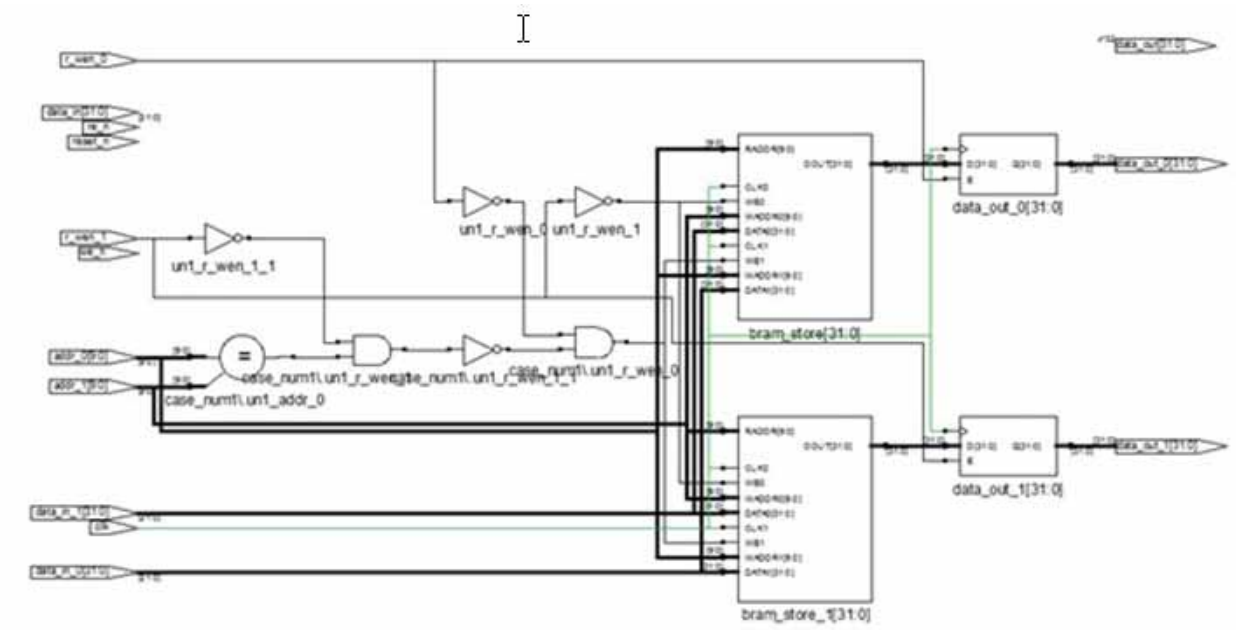
Case 2 - Dual-port with pipelining (registered data_out ports)

```
case_num2 : if testcase = 2 generate

   int_addr_0 <= CONV_INTEGER(addr_0);
   int_addr_1 <= CONV_INTEGER(addr_1);

process(clk)
begin

         if rising_edge(clk) then
         -- port 0
               if (r_wen_0 = '0') then
                     BRAM_store(int_addr_0)        <= data_in_0;
         else
            data_out_0tmp    <= BRAM_store(int_addr_0);
         end if;
         -- port 1
                  if (r_wen_1 = '0') then
                           BRAM_store(int_addr_1) <= data_in_1;
                  else
                           data_out_1tmp <= BRAM_store(int_addr_1);

                 end if;

                 data_out_0<= data_out_0tmp;
                 data_out_1<= data_out_1tmp;
      end if;
   end process;
end generate;

end def_arch;
```

Case 3 - Dual-port with pipelining (registered read address)

```
case_num3 : if testcase = 3 generate

   process(clk)
   begin
         if rising_edge(clk) then
         -- port 1
               if (r_wen_0 = '0') then
                     BRAM_store(int_addr_0) <= data_in_0;
                else
                     int_addr_0 <= CONV_INTEGER(addr_0);
                end if;
         -- port 1
                if (r_wen_1 = '0') then
                      BRAM_store(int_addr_1) <= data_in_1;
                else
                      int_addr_1 <= CONV_INTEGER(addr_1);
               end if;

               data_out_0 <= BRAM_store(int_addr_0);
               data_out_1 <= BRAM_store(int_addr_1);
      end if;
   end process;
end generate;

end def_arch;
```

Results of generic testcase set to 1



The tool infers RTG4 RAM1K18_RT.

Resource Usage Report for RAM_inference_examples

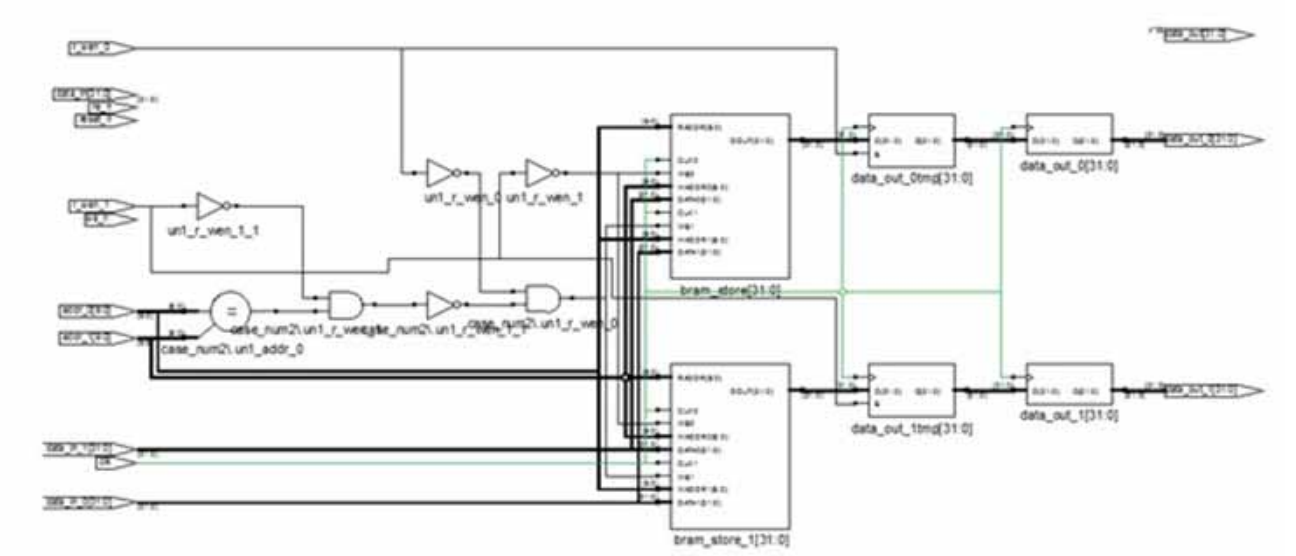
Mapping to part: rt4g150cg1657-1
Cell usage:

CLKINT	1 use
RAM1K18_RT	2 uses
CFG1	1 use
CFG4	7 uses

Sequential Cells:

SLE	0 uses

Results of generic testcase set to 2



The tool infers RTG4 RAM1K18_RT.

Resource Usage Report for RAM_inference_examples

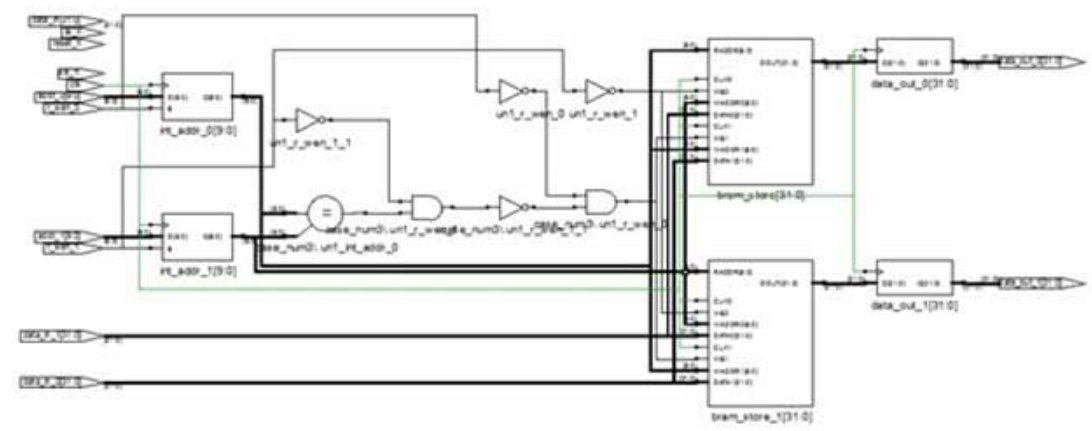
Mapping to part: rt4g150cg1657-1
Cell usage:

CLKINT	1 use
RAM1K18_RT	2 uses
CFG1	1 use
CFG4	7 uses

Sequential Cells:

SLE	0 uses

Results of generic testcase set to 3



The tool infers RTG4 RAM1K18_RT.

Resource Usage Report for RAM_inference_examples

Mapping to part: rt4g150cg1657-1

Cell usage:

CLKINT	1 use
RAM1K18_RT	2 uses
CFG1	1 use
CFG2	1 use
CFG3	1 use
CFG4	6 uses

Sequential Cells:

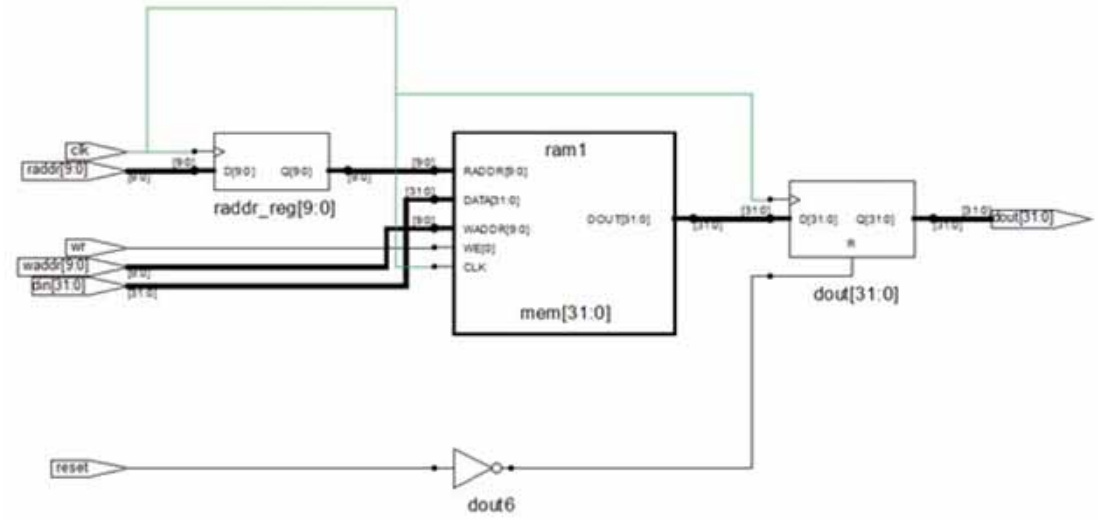
SLE 20 uses

Example 28: Simple Dual-Port (Two-Port) RAM with Asynchronous Reset for Pipeline Register

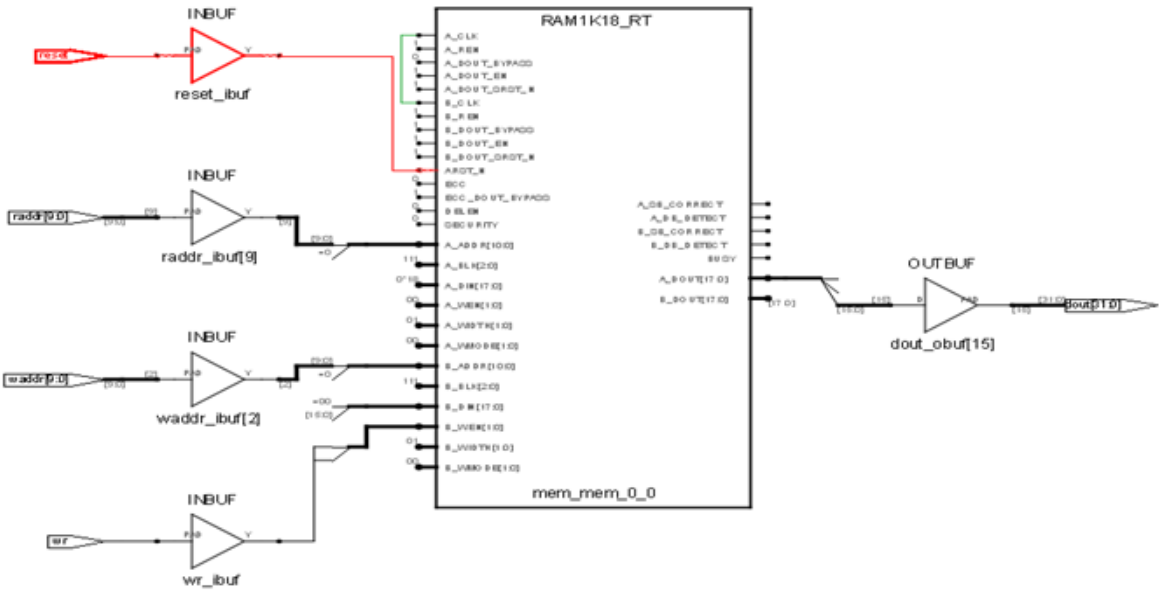
The following design is a simple dual-port RAM1K18 with asynchronous reset for pipeline register.

```
module ram_2port_addreg_areset(clk,wr,raddr,din,waddr,dout,reset);
input clk,reset;
input [31:0] din;
input wr;
input [9:0] waddr,raddr;
output [31:0] dout;
reg [31:0] dout;
reg [31:0] mem [0:1023];
reg [9:0] raddr_reg;

always@(posedge clk or negedge reset)
begin
   if (!reset)
      dout <= 0;
   else
      dout <= mem[raddr_reg];
end

always@(posedge clk )
begin
   if(wr)
      mem[waddr]       <= din;
      raddr_reg <= raddr;
end
endmodule
```

SRS (RTL) View



SRM (Technology) View



The tool infers RTG4 RAM1K18_RT with asynchronous reset packing.

Resource Usage Report for ram_2port_addreg_areset

Mapping to part: rt4g150cg1657-1
Cell usage:

CLKINT	1 use
RAM1K18_RT	2 uses
CFG2	0 uses

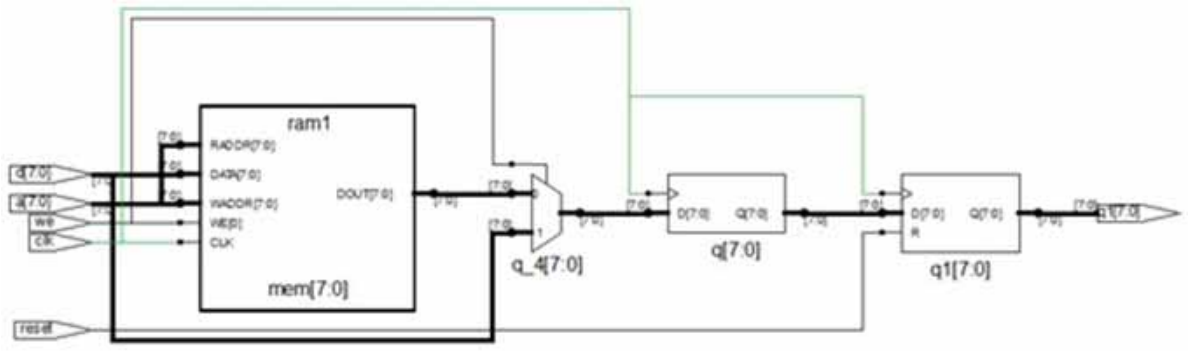
Sequential Cells: SLE 0 uses

Example 29: Single-Port RAM with Synchronous Reset for Pipeline Register (RAM64x18_RT)

The following design is a single-port RAM with synchronous reset for pipeline register.

```
module ram_singleport_writefirst_pipe_areset(clk,we,a,d,q1,reset);
input [7:0] d;
input [7:0] a;input clk, we,reset;
reg [7:0] q;
output [7:0] q1;
reg [7:0] q1;
reg [7:0] mem [255:0];

always @(posedge clk) begin
if(we)
mem[a] <= d;
end
```

```
always @ (posedge clk)
begin
if(we)
q <= d;
else
q <= mem[a];
end
always @ (posedge clk )
begin
if (reset)
   q1 <= 0;
else
   q1 <= q;
end
endmodule
```



The tool infers RTG4 RAM64x18_RT.

Resource Usage Report for ram_singleport_writefirst_pipe_areset

Mapping to part: rt4g150cg1657-1

Cell usage:

CLKINT	1 use
RAM64x18_RT	2 uses
CFG2	2 uses
CFG3	8 uses

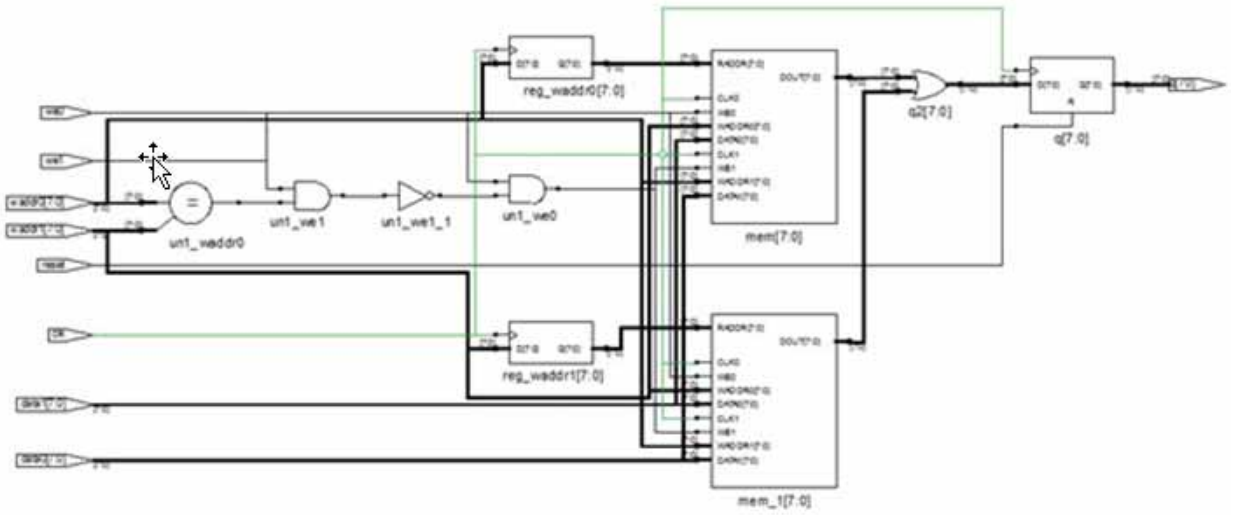
Sequential Cells:

SLE	9 uses

Example 30: True Dual-Port RAM with Asynchronous Reset for Pipeline Register (RAM1K18_RT)

The following design is a true dual-port RAM with asynchronous reset for pipeline register.

```
module ram_dport_addreg_pipe_areset(data0,data1,waddr0,
waddr1,we0,we1,clk,q,reset); parameter d_width = 8;
parameter addr_width = 8;
parameter mem_depth = 256;
input [d_width-1:0] data0, data1;
input [addr_width-1:0] waddr0, waddr1;
input we0, we1, clk,reset;
output [d_width-1:0] q;
reg [d_width-1:0] mem [mem_depth-1:0];
reg [addr_width-1:0] reg_waddr0, reg_waddr1;
reg [d_width-1:0] q;

wire [d_width-1:0] q0, q1;
wire [d_width-1:0] q2;

assign q2 = q0 | q1;
assign q0 = mem[reg_waddr0];
assign q1 = mem[reg_waddr1];

always @(posedge clk)
begin
if (we0)
mem[waddr0] <= data0;
if (we1)
mem[waddr1] <= data1;
reg_waddr0 <= waddr0;
reg_waddr1 <= waddr1;
end

always @(posedge clk or posedge reset)
begin
   if(reset)
      q <=
      0;
   else
      q <= q2;
end
endmodule
```



The tool infers RTG4 RAM1K18_RT.

Resource Usage Report for ram_dport_addreg_pipe_areset

Mapping to part: rt4g150cg1657-1

Cell usage:

CLKINT	1 use
RAM1K18_RT	1 use
CFG1	1 use
CFG2	8 uses
CFG3	1 use
CFG4	5 uses

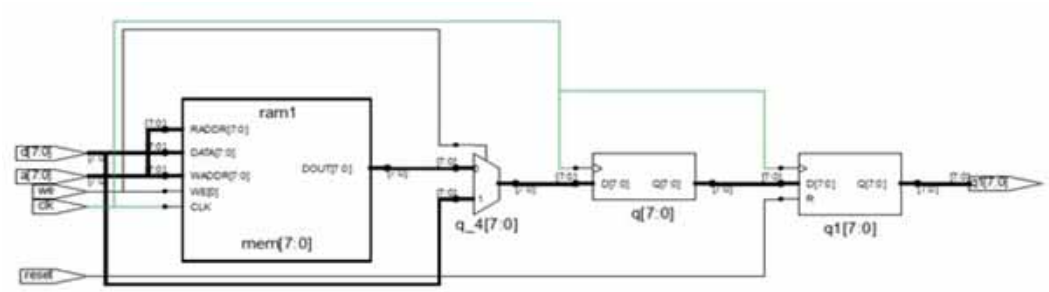
Sequential Cells:

SLE	8 uses

Example 31: Single-Port RAM with Synchronous Reset for Pipeline Register (RAM64x18_RT) (syn_ramstyle=rw_check)

The following design is a single-port RAM with synchronous reset for pipeline register.

```
module ram_singleport_writefirst_pipe_sreset(clk,we,a,d,q1,reset);
input [7:0] d;
input [7:0] a;
input clk, we,reset;
reg [7:0] q;
output [7:0] q1;
reg [7:0] q1;
reg [7:0] mem [255:0] /* synthesis syn_ramstyle="rw_check" */;
always @(posedge clk) begin
if(we)
mem[a] <= d;
end

always @ (posedge clk)

begin
if(we)
q <= d;
else
q <= mem[a];
end
always @ (posedge clk )
begin
if (reset)
q1 <= 0;
else
q1 <= q;
end
endmodule
```



The tool infers RTG4 RAM64X18_RT with glue logic.

Resource Usage Report for ram_singleport_writefirst_pipe_sreset

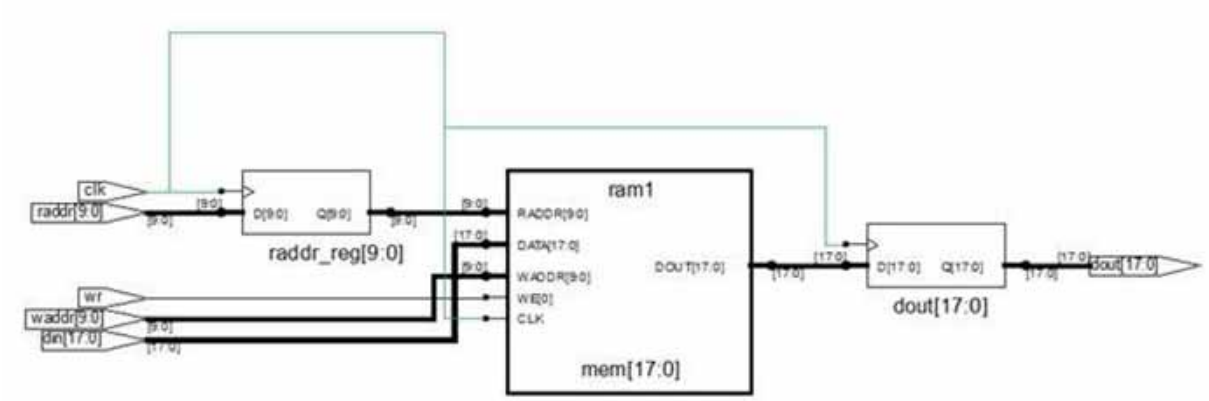
Mapping to part: rt4g150cg1657-1
Cell usage:

CLKINT	1 use
RAM64x18_RT	2 uses
CFG2	10 uses
CFG3	8 uses
CFG4	8 uses
Sequential Cells:	
SLE	18 uses

Example 32: Simple Dual-Port RAM with Output Register Using syn_ramstyle="rw_check"

The following design is a single-port RAM with output register using syn_ramstyle="rw_check".

```
module ram_2port_pipe(clk,wr,raddr,din,waddr,dout);
input clk;
input [17:0] din; input wr;
input [9:0] waddr,raddr;
output [17:0] dout;
reg [9:0] raddr_reg;
reg [17:0] mem [0:1023] /* synthesis syn_ramstyle= "rw_check" */;
reg [17:0] dout;
always@(posedge clk)
begin
raddr_reg <= raddr;
dout <= mem[raddr_reg];
if(wr)
mem[waddr] <= din;
end
endmodule
```



The FPGA synthesis tool infers RTG4 RAM1K18_RT along with glue logic for read/write address check.

Resource Usage Report for ram_2port_pipe

Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 1 use
RAM1K18_RT1 use
CFG2 1 use
CFG4 24 uses

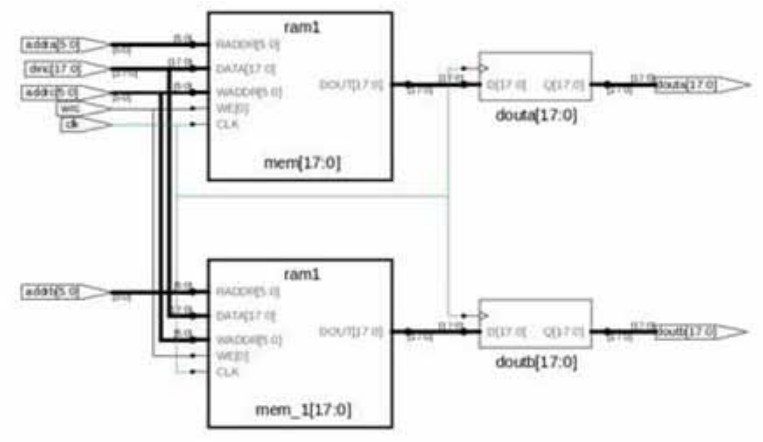
Sequential Cells: SLE 57 uses

Example 33: Three-Port RAM with Synchronous Read

The following design is a Verilog example for three-port RAM with synchronous read.

```
module ram_infer15_rtl(clk,dinc,douta,doutb,wrc,addra,addrb,addrc);
input clk;
input [17:0] dinc;
input wrc;
input [5:0] addra,addrb,addrc;
output [17:0] douta,doutb;
reg [17:0] douta,doutb;
reg [17:0] mem [0:63];
always@(posedge clk)
begin
if(wrc)
mem[addrc] <= dinc;
end

always@(posedge clk)
begin
douta <= mem[addra];
end

always@(posedge clk)
begin
doutb <= mem[addrb] ;
end
endmodule
```

RTL view



The tool infers one RAM64X18_RT.

Resource Usage Summary for ram_infer15_rtl

Mapping to part: rt4g150cg1657-1

Cell usage:
CLKINT 1 use
RAM64x18_RT 1 use

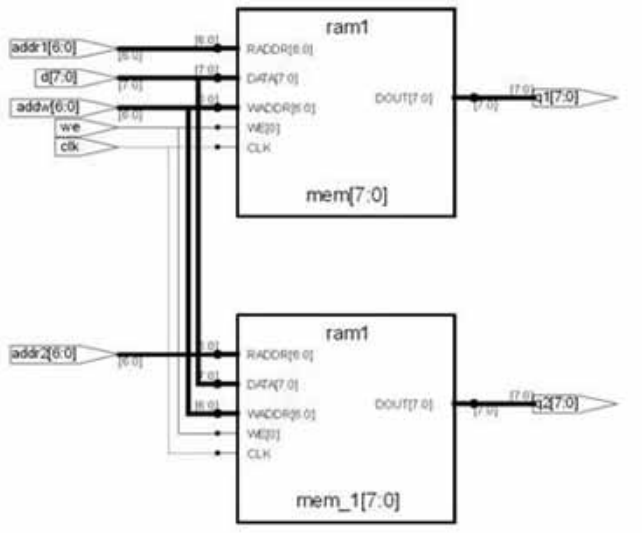
Sequential Cells:
SLE 0 uses

Example 34: Three-Port RAM with Asynchronous Read

The following design is a VHDL example for three-port RAM with asynchronous read.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ram_singleport_noreg is
port (d : in std_logic_vector(7 downto 0);
addw : in std_logic_vector(6 downto 0);
addr1 : in std_logic_vector(6 downto 0);
addr2 : in std_logic_vector(6 downto 0);
we : in std_logic;
clk : in std_logic;
q1 : out std_logic_vector(7 downto 0);
q2 : out std_logic_vector(7 downto 0) );
end ram_singleport_noreg;
architecture rtl of ram_singleport_noreg is
type mem_type is array (127 downto 0) of
std_logic_vector (7 downto 0);
signal mem: mem_type;
begin
```

```
process (clk)
begin
if rising_edge(clk) then
if (we = '1') then
mem(conv_integer (addw)) <= d;
end if;
end if;
end process;
q1<= mem(conv_integer (addr1));
q2<= mem(conv_integer (addr2));
end rtl;
```



The tool infers one RAM64X18_RT.

Resource Usage Report for ram_singleport_noreg

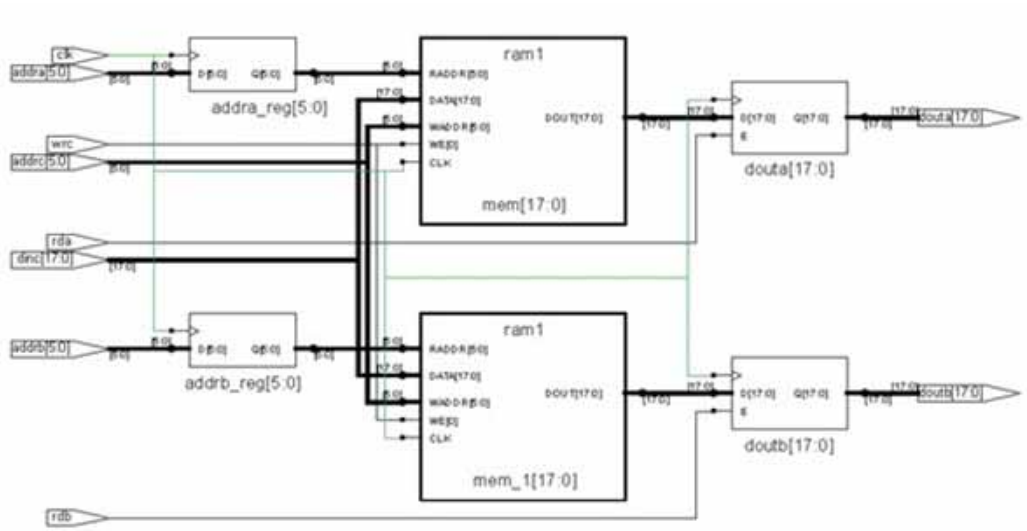
Mapping to part: rt4g150cg1657-1
Cell usage:
RAM64x18_RT 1 use

Sequential Cells:
SLE 0 uses

Example 35: Three-Port RAM with Read Address and Pipeline Register

The following design is an example for three-port RAM with read address and pipeline register.

```
module ram_infer(clk,dinc,douta,doutb,wrc,rda,rdb,addra,addrb,addrc);
input clk;
input [17:0] dinc;
input wrc,rda,rdb;
input [5:0] addra,addrb,addrc;
output [17:0] douta,doutb;
reg [17:0] douta,doutb;
reg [5:0] addra_reg, addrb_reg;

reg [17:0] mem [0:63];
always@(posedge clk)
begin
addra_reg <= addra;
addrb_reg <= addrb;

if(wrc)
mem[addrc] <= dinc;
end

always@(posedge clk)
begin
if(rda)
douta <= mem[addra_reg];
end

always@(posedge clk)
begin
if(rdb)
doutb <= mem[addrb_reg] ;
end
endmodule
```



The tool infers one RAM64X18_RT.

Resource Usage Report for ram_infer

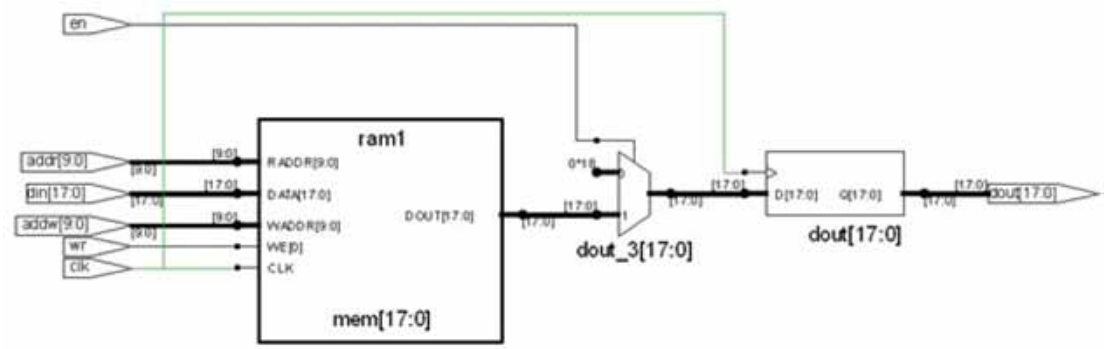
Mapping to part: rt4g150cg1657-1
Cell usage:
CLKINT 1 use
RAM64x18_RT 1 use

Sequential Cells:
SLE 0 uses

Example 36: Simple Dual-Port RAM with Enable on Output Register

The following design is an example for simple dual-port RAM with enable on output register. When enable is deasserted, the RAM output is 0.

```
module ram_singleport_outreg_areset_en_rtl(clk,wr,addr,addw, din,dout,en);

output [17:0] dout;
input [17:0] din;
input [9:0] addr, addw;
input clk, wr, en;
reg [17:0] dout;
reg [17:0]mem[1023:0] ;

always@(posedge clk)
begin
if(wr)
   mem[addw] <= din;
end

always@(posedge clk)

begin
if(en)
   dout <= mem[addr];
else
   dout <= 0;
end

endmodule
```



The tool infers one RAM1K18_RT using A_BLK pin for enable en. enable en pin is mapped using A_BLK or B_BLK pin on RAM1K18_RT only when one port of RAM1Kx18_RT is used for reading and another port for writing.

Resource Usage Report for ram_singleport_outreg_areset_en_rtl

Mapping to part: m2s050tfbga896std
Cell usage:

CLKINT	1 use
RAM1K18_RT	1 use

Sequential Cells:

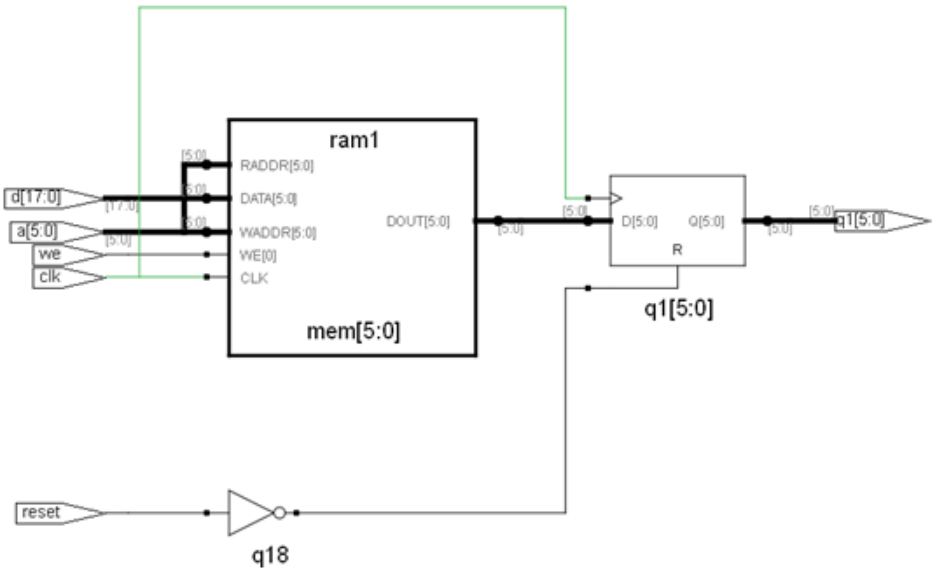
SLE	0 uses

Example 37: Single-Port RAM with Asynchronous Reset (RAM64x18_RT)

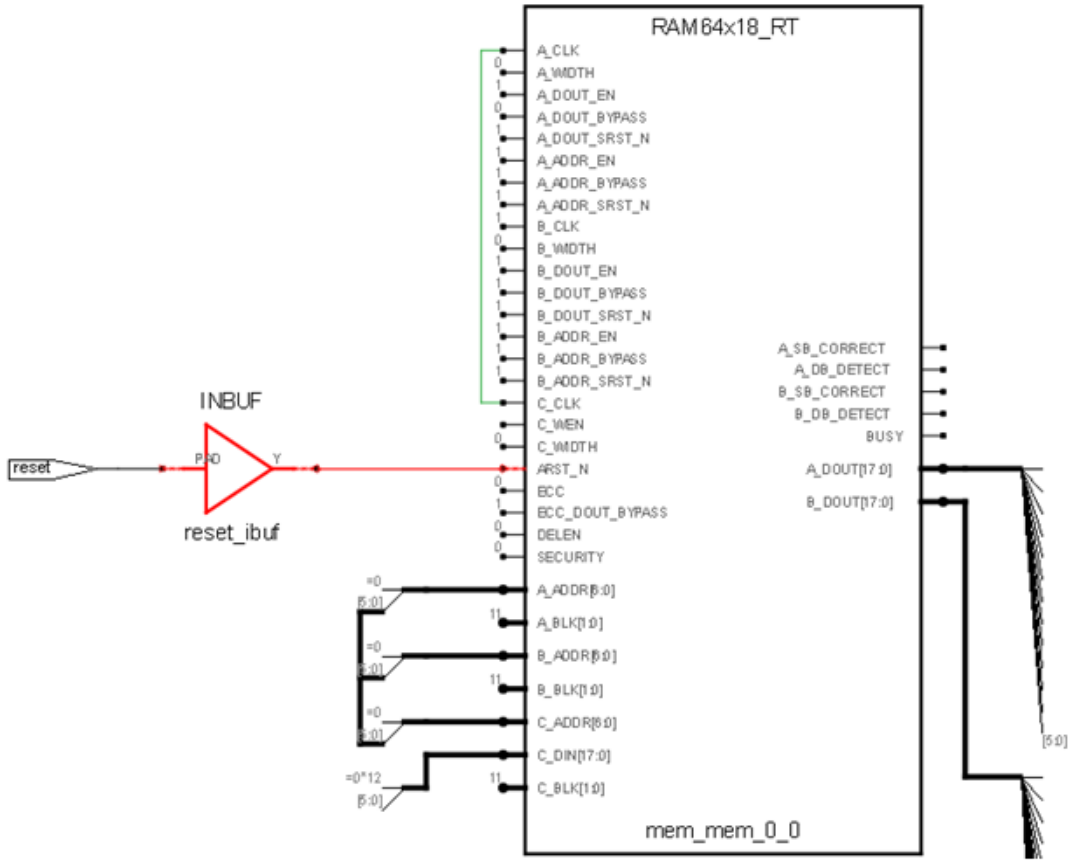
The following design is a single-port RAM with asynchronous reset.

```
module ram_singleport_areset(clk,we, a,d,q1,reset);
input [17:0] d;
input [5:0] a;
input clk, we,reset;
output reg [5:0] q1;
reg [17:0] mem [63:0];

always @(posedge clk)
begin if(we)
mem[a] <= d;
end

always @ (posedge clk or negedge reset)
begin
if (!reset)
q1 <= 0;
else
q1 <= mem[a];
end
endmodule
```

SRS (RTL) View



SRM (Technology) View



The tool infers RTG4 RAM64x18_RT with asynchronous reset packing.

Example 38: Simple Dual-Port RAM64x18_RT in Low Power Mode

For the 128x18 RAM configuration, the tool fractures the data width and infers two RAM64x18_RT RAM blocks in 128x12 mode.

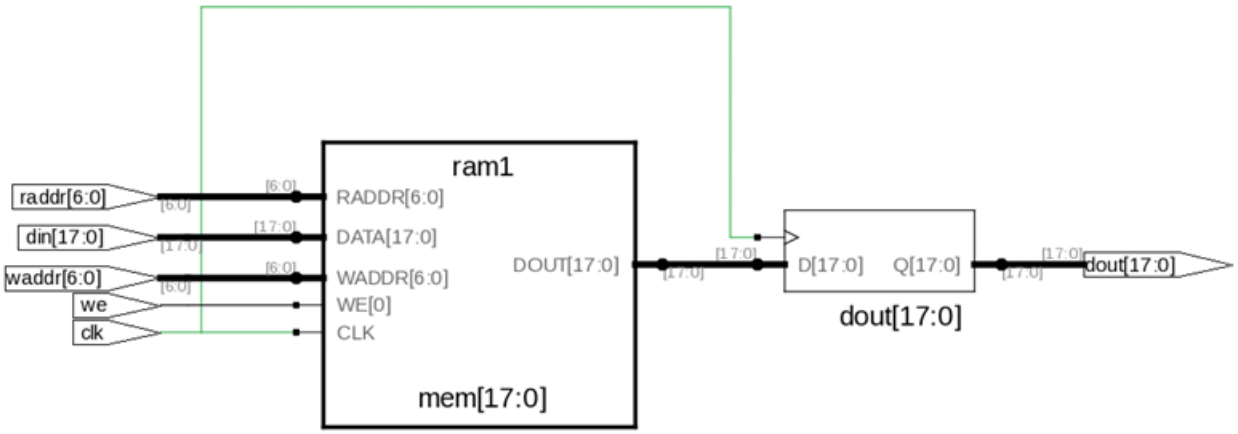
When you set the global option low_power_ram_decomp 1 in the project file (*.prj), the tool fractures the address width to infer two RAM64x18_RT blocks in 64x18 mode. The tool connects the MSB bit of address to the BLK pin and OR gates at the output to select the output from the two RAM blocks.

RTL

```
`ifdef synthesis
module test (raddr, waddr, clk, we, din, dout);
`else
module test_RTL (raddr, waddr, clk, we, din, dout);
`endif

parameter ADDR_WIDTH = 7;
parameter DATA_WIDTH = 18;
parameter MEM_DEPTH = 128;

input [ADDR_WIDTH-1:0]raddr;
input [ADDR_WIDTH-1:0]waddr;
input clk, we;
output[DATA_WIDTH-1 : 0]dout;
input [DATA_WIDTH-1 : 0]din;

reg [DATA_WIDTH-1 : 0]dout;
reg [DATA_WIDTH-1 : 0]mem[MEM_DEPTH-1 :0] ;

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

always@(posedge clk) begin
   if(we)
     mem[waddr] <= din;
end
endmodule
```

Project file option is set_option -low_power_ram_decomp 1.

SRS View (RTL)



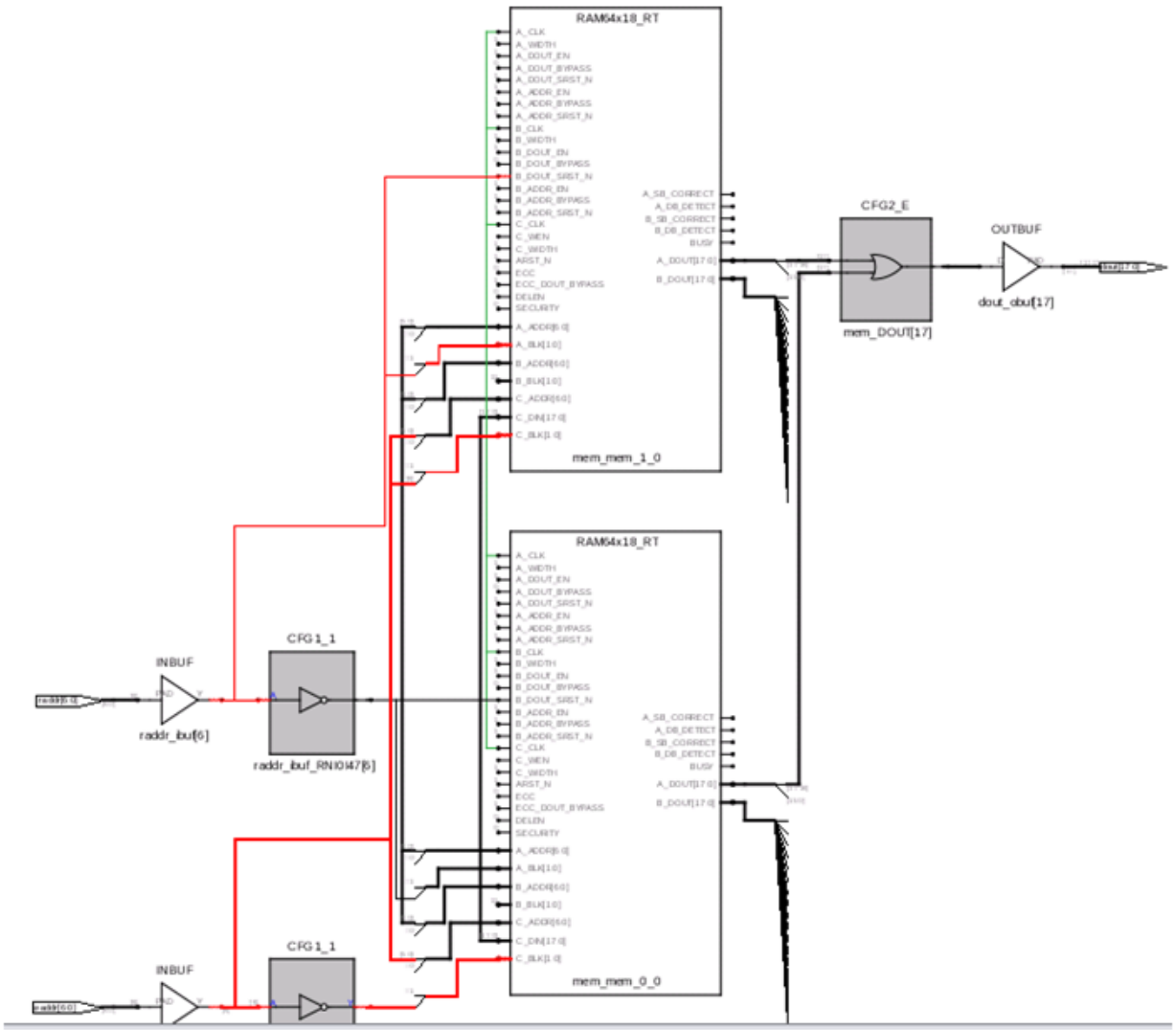
Resource Usage

Cell usage:
CLKINT 1 use
CFG1 4 uses
CFG2 20 uses

SLE 0 uses

Block Rams (RAM64x18_RT): 2

SRM (Technology) View



Example 39: Simple Dual-Port RAM1K18_RT in Low Power Mode

For 2Kx18 RAM configuration, the tool fractures the data width and infers two RAM1K18_RT RAM blocks in 2Kx9 mode.

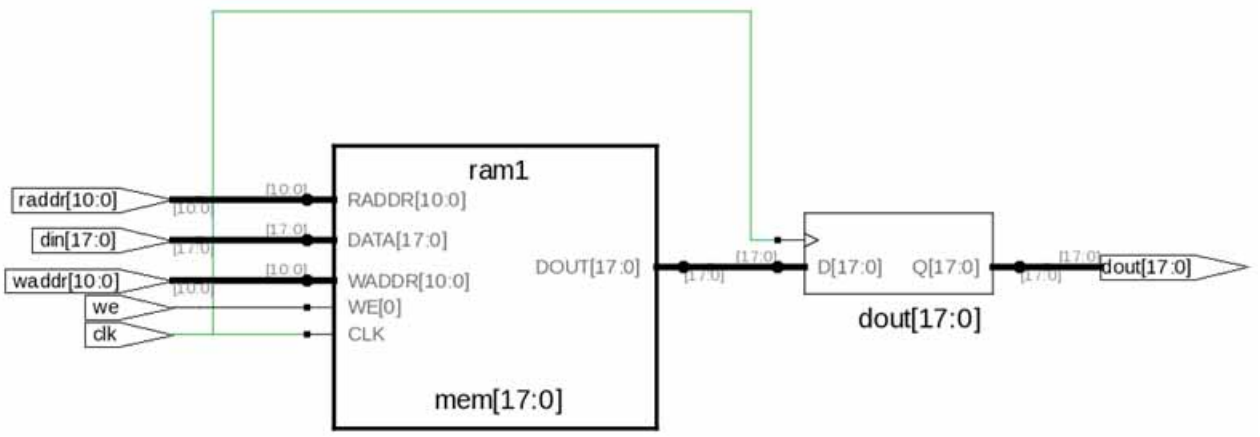
By setting the global option low_power_ram_decomp 1 in the project file (*.prj), the tool will fracture the address width to infer two RAM1K18_RT blocks in 1Kx18 mode. The tool will connect the MSB bit of address to the BLK pin and OR gates at the output, to select the output from two RAM blocks.

RTL

```
module test (raddr, waddr, clk, we, din, dout);
parameter ADDR_WIDTH = 11;
parameter DATA_WIDTH = 18;
parameter MEM_DEPTH = 2048;
input [ADDR_WIDTH-1:0] raddr;
input [ADDR_WIDTH-1:0] waddr;
input clk, we;
output[DATA_WIDTH-1 : 0]dout;

input [DATA_WIDTH-1 : 0]din;
reg [DATA_WIDTH-1 : 0]dout;
reg [DATA_WIDTH-1 : 0]mem[MEM_DEPTH-1 :0] ;
always@(posedge clk) begin
dout <= mem[raddr];
end
always@(posedge clk) begin
   if(we)
     mem[waddr] <= din;
end
endmodule
```

Project File option is set_option -low_power_ram_decomp 1.

SRS View (RTL)



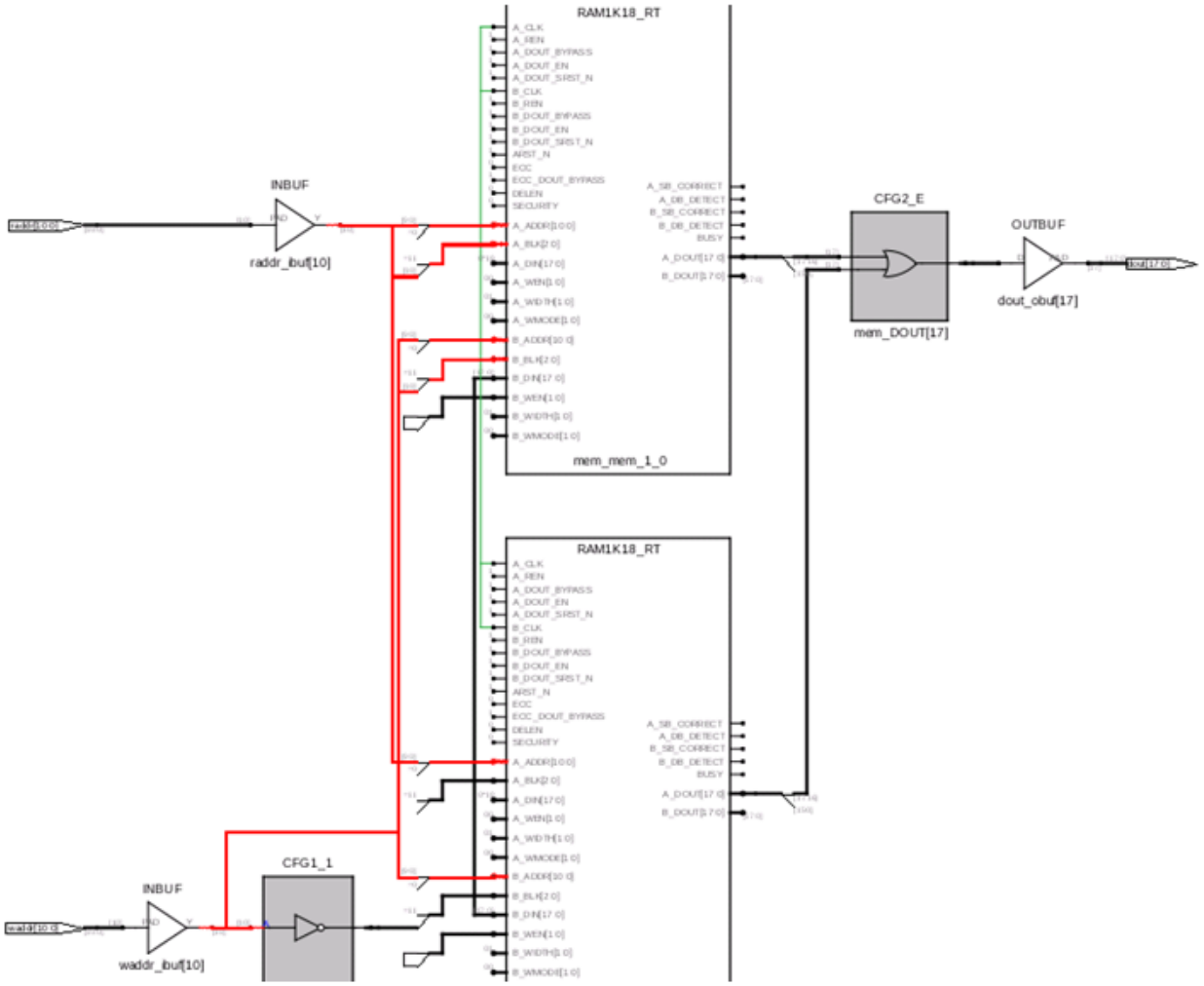
Resource Usage

Cell usage:

CLKINT	1 use
CFG1	2 uses
CFG2	20 uses
SLE	0 uses

Block Rams (RAM1K18_RT): 2

SRM (Technology) View



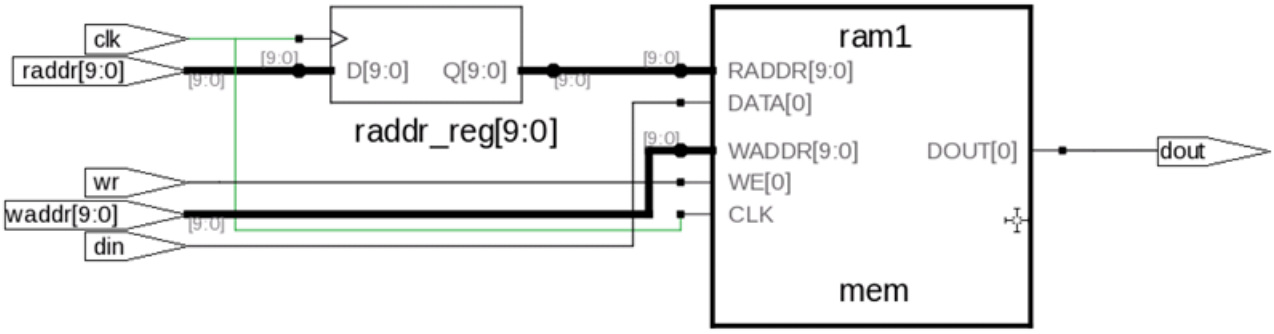
Example 40: Simple Dual-Port RTG4 RAM with x1 Configuration

The following design is an example for simple dual-port RAM with x1 data width configuration for the RTG4 device.

RTL

```
`define synthesis 1
module ram_2port_addreg_1kx1(clk,wr,raddr,din,waddr,dout);

input clk;
input  din;
input wr;
input [9:0] waddr,raddr;
output dout;
reg [9:0] raddr_reg;
reg  mem [0:1023] ;
wire  dout;

assign dout = mem[raddr_reg]  ;

always@(posedge clk)
begin
    raddr_reg <= raddr;
   if(wr)
      mem[waddr]  <= din;
end
endmodule
```

SRS View (RTL)



Resource Usage

Cell usage:
CLKINT 1 use
CFG4 8 uses

SLE 3 uses

Block Rams (RAM64x18_RT): 8

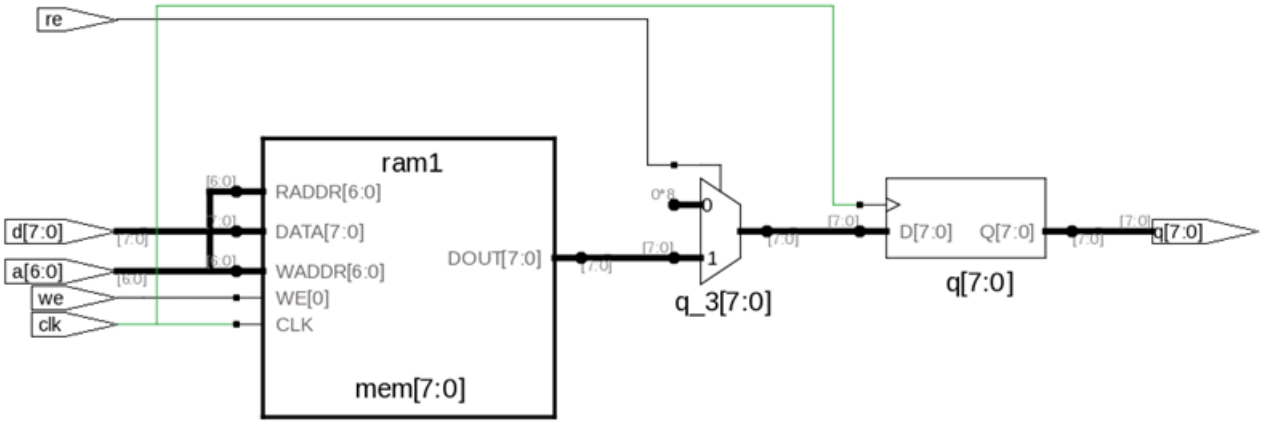
Example 41: Single-Port RTG4 RAM (VHDL)

The following design is a VHDL example for RTG4 RAM with Read Enable to read from RAM and output of RAM set to 0 when Read Enable is de-asserted.

RTL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ram_test is
   port (d: in std_logic_vector(7 downto 0);
      a: in integer range 127 downto 0;
      we: in std_logic;
      re: in std_logic;
      clk: in std_logic;
      q: out std_logic_vector(7 downto 0) );
end ram_test;

architecture rtl of ram_test is
type mem_type is array (127 downto 0) of std_logic_vector (7 downto 0);
signal mem: mem_type;

attribute syn_ramstyle : string;
attribute syn_ramstyle of mem : signal is "lsram";

begin
   process(clk)
   begin
      if (clk'event and clk='1') then
       --q <= mem(a);
         if (we='1') then
          mem(a) <= d;
         end if;

    if (re='1') then
      q <= mem(a);
    else
      q <= "00000000";
    end if;

  end if;
end process;

end rtl;
```

SRS View (RTL)



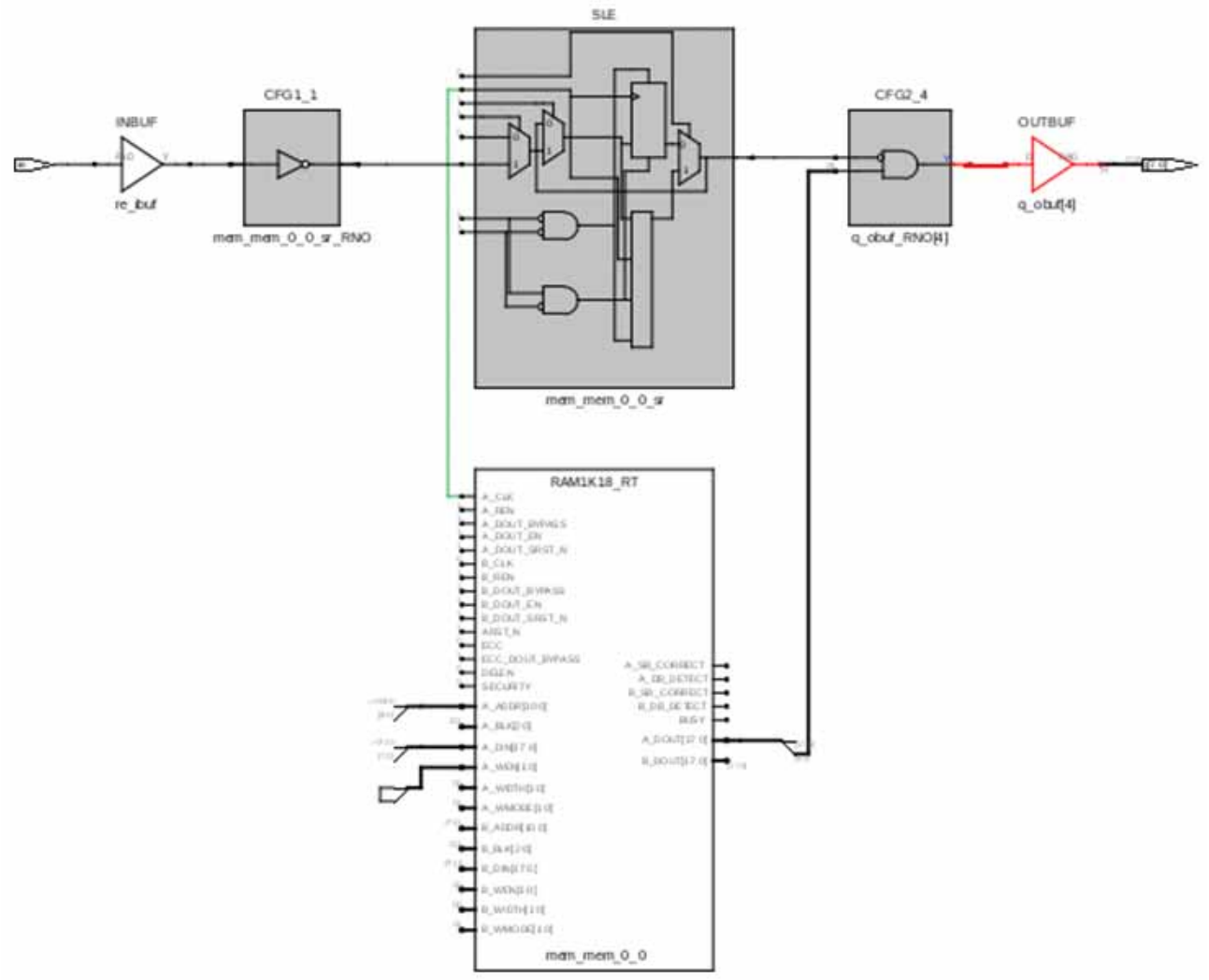
Resource Usage

Cell usage:
CLKINT 1 use
CFG1 1 uses
CFG2 8 uses

SLE 1 use

Block Rams (RAM1K18_RT): 1

SRM (Technology) View



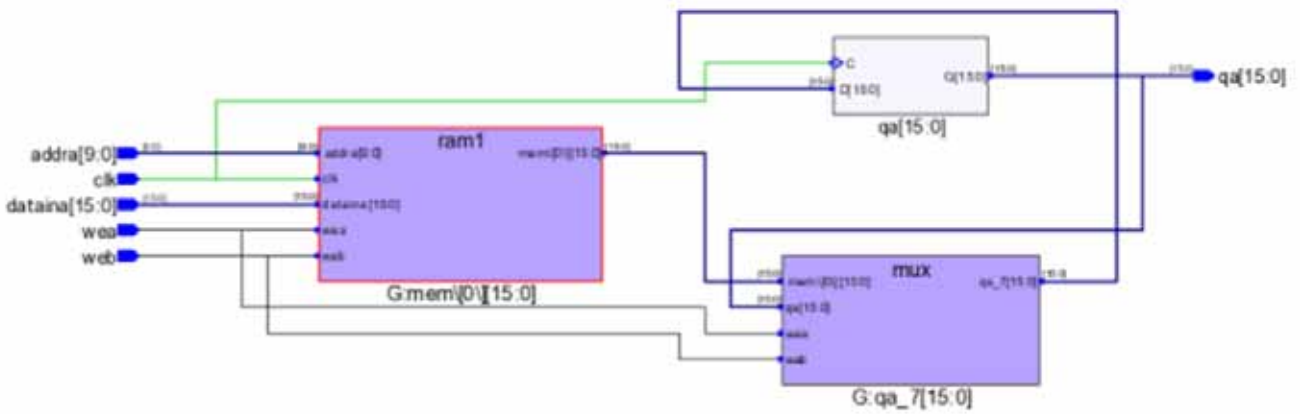
Example 42: RTL Coding Style for 1Kx16 Single-Port RAM with 2 Write Byte-Enables

In the RTL below, there are two write-enables, wea and web. These are used to control write access to the RAM.

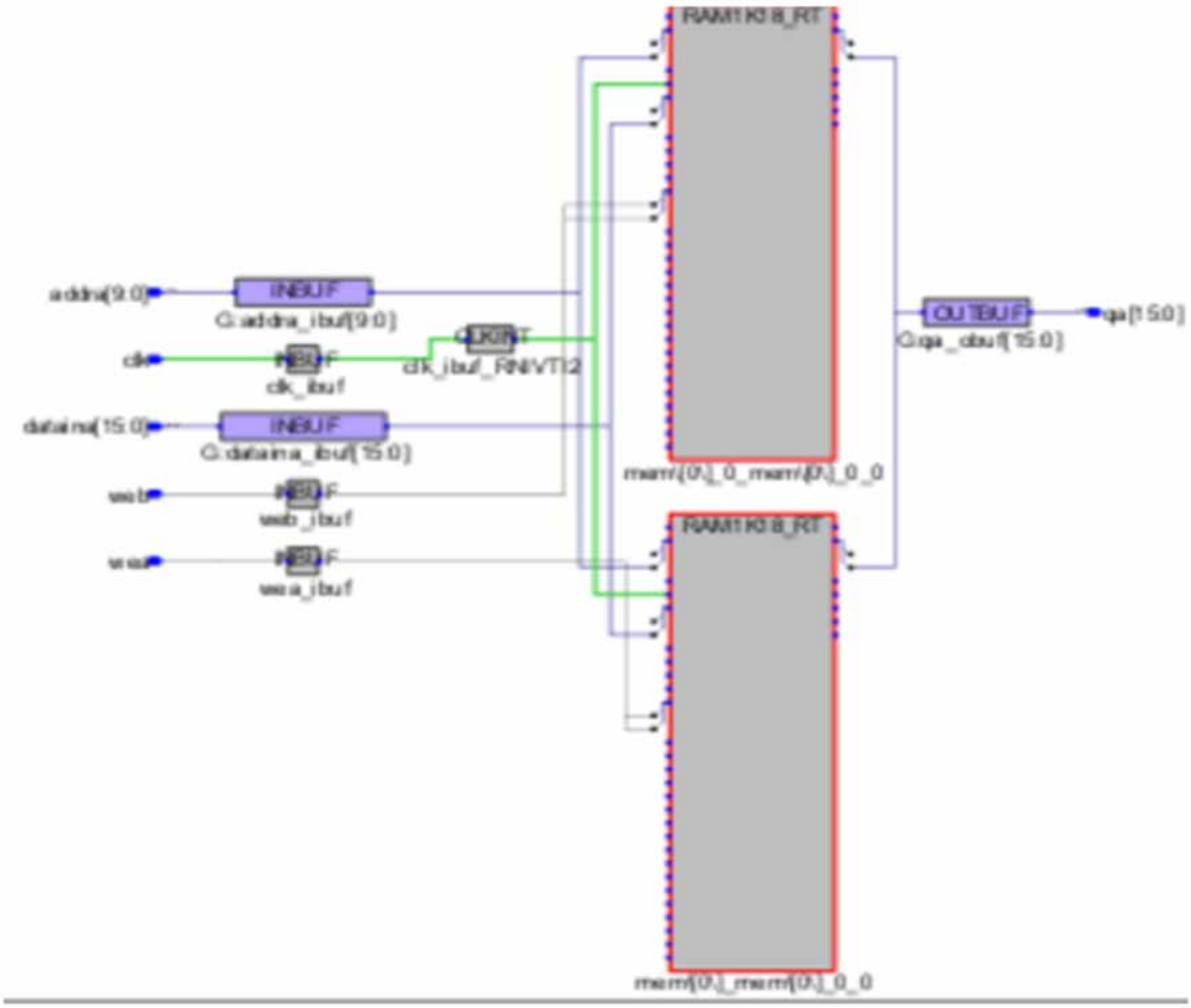
RTL

```
module ram_wb_singleport_2wen(clk,wea,addra,dataina,qa,web);

parameter addr_width =10;
parameter data_width = 16;

input clk,wea,web;
input [data_width - 1 : 0] dataina;
input [addr_width - 1 : 0] addra;
output reg [data_width - 1 : 0] qa;

reg [data_width - 1 : 0] mem [(2**addr_width) - 1 : 0];

always @ (posedge clk)
begin
      if(web) mem[addra][7:0] <= dataina[7:0];
      if(wea) mem[addra][15:8] <= dataina[15:8];
end

always @ (posedge clk)
begin

      if (~web)
            qa[7:0] <= mem[addra][7:0];
      if (~wea)
            qa[15:8] <= mem[addra][15:8];

end
endmodule
```

SRS View



SRM View



Resource Usage Report

CLKINT 1 use
CFG1 1 use
CFG2 8 uses

SLE 1 use
Total Block RAMs (RAM1K18_RT) : 2 of 209 (0%)
Total LUTs: 9

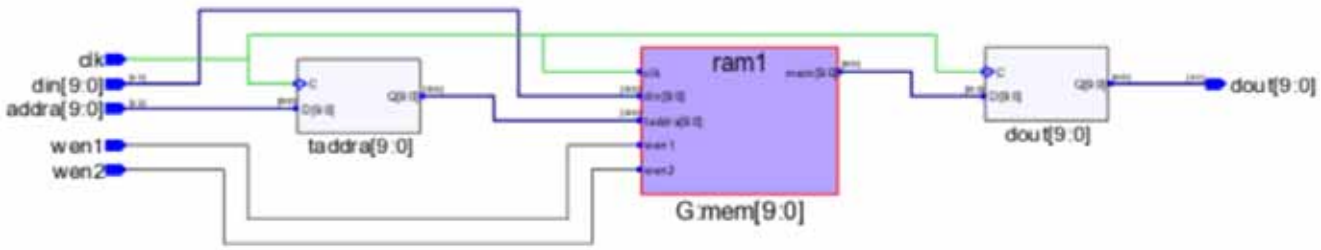
Example 43: RTL Coding Style for 1Kx10 Single-Port RAM with Write Byte-Enables

In the RTL below, there are two write-enables. These are used to control write access to the RAM.

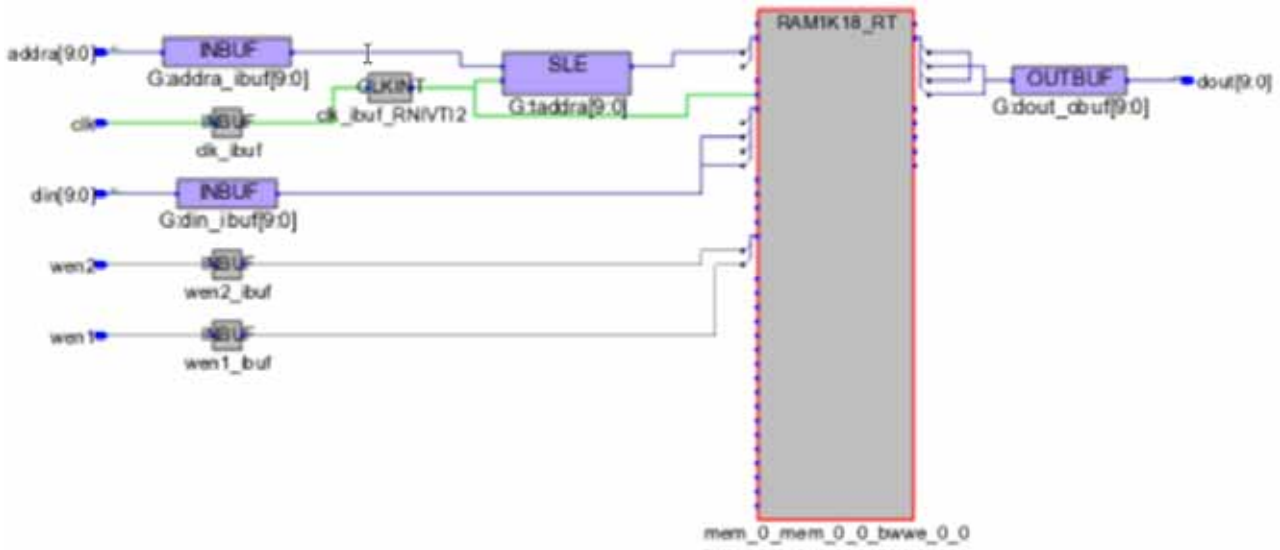
RTL

```
module ram_wb_wen_1addr(din, dout, addra, clk, wen1, wen2);
input [9:0] din;
input wen1;
input wen2;
input [9:0] addra;
input clk;
output reg [9:0] dout;

localparam max_depth=1024;
localparam min_width=10;

reg [9:0] taddra;
reg [min_width-1:0]  mem_ram[max_depth-1:0];

always @(posedge clk)
begin
      taddra<=addra;
   if(wen1)
      mem_ram[taddra][4:0]<=din[4:0];

  if(wen2)
      mem_ram[taddra][9:5]<=din[9:5];
end

always @(posedge clk)
begin
      dout <= mem_ram[taddra];
end
endmodule
```

SRS View



SRM View



Resource Usage Report

SLE 10 uses
Total Block RAMs (RAM1K18_RT) : 1 of 209 (0%)
Total LUTs: 0

Example 44: RTL Coding Style for 1Kx8 Simple Dual-Port RAM with Write Byte-Enables

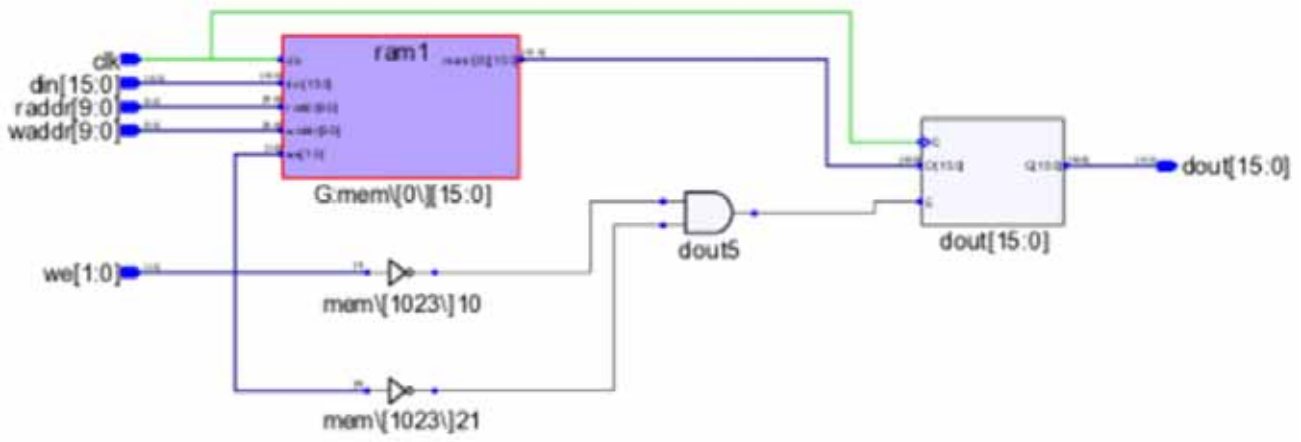
In the RTL below, there are two write-enables, we[1:0]. These are used to control write access to simple dual-port RAM.

RTL

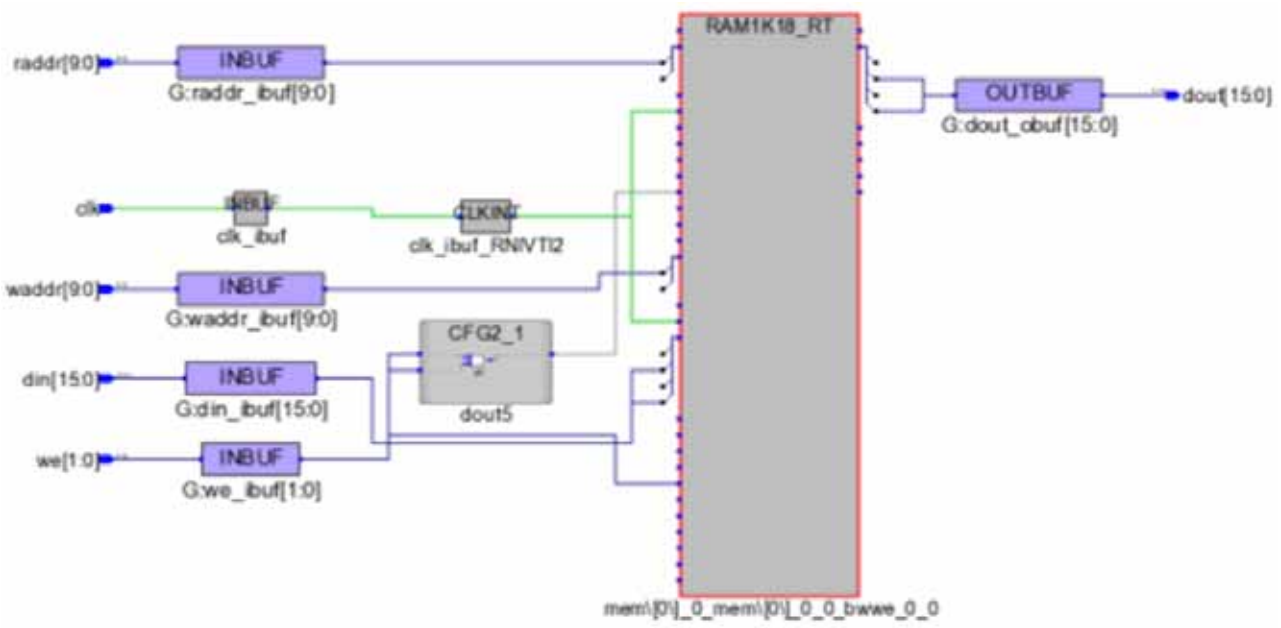
```
module test (raddr, waddr, clk, we, din, dout);
parameter ADDR_WIDTH = 10;
parameter DATA_WIDTH = 16;
parameter MEM_DEPTH = 1024;
input [ADDR_WIDTH-1:0]raddr;
input [ADDR_WIDTH-1:0]waddr;
input clk;
input [1:0] we;
output reg[DATA_WIDTH-1 : 0]dout;
input [DATA_WIDTH-1 : 0]din;

reg [ADDR_WIDTH-1:0]raddr_reg;
reg [DATA_WIDTH-1 : 0]mem[MEM_DEPTH-1 :0]/* synthesis syn_ramstyle="lsram"*/ ;
```

```
always @(posedge clk) begin
      if (we[1])
         mem[waddr][7:0] <= din[7:0];
      if (we[0])
         mem[waddr][15:8] <= din[15:8];
end

always @(posedge clk) begin
      if(~we[0] & ~we[1])
            dout <= mem[raddr];
   end
endmodule
```

SRS View



SRM View



Resource Usage Report

SLE 0 uses
Total Block RAMs (RAM1K18_RT) : 1 of 209 (0%)
Total LUTs: 1

Example 45: RTL Coding Style for 3-Port RAM with Write Byte-Enable

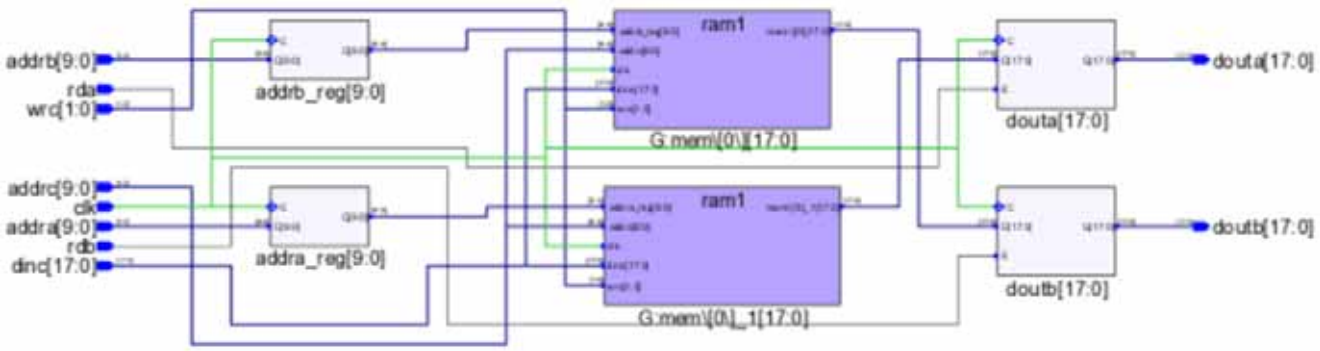
In the RTL below, there are write-enable (wrc[1:0]) pins. They are used to control write access to the RAM.

RTL

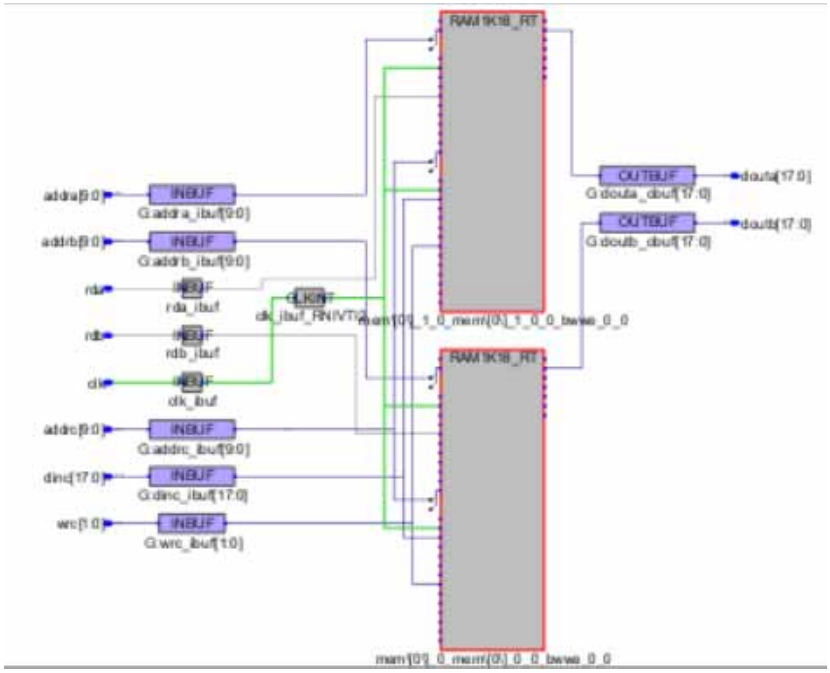
```
module ram_wb_3port_1wen(clk,dinc,douta,doutb,wrc,
                rda,rdb,addra,addrb,addrc);
input clk;
input [17:0] dinc;
input [1:0] wrc;
input rda,rdb;
input [9:0] addra,addrb,addrc;
output [17:0] douta,doutb;

reg [17:0] douta,doutb;
reg [9:0] addra_reg, addrb_reg;
reg [17:0] mem [0:1023];
```

```
always@(posedge clk)
begin

addra_reg <= addra;
addrb_reg <= addrb;

if(wrc[0])
        mem[addrc][8:0] <= dinc[8:0];
if(wrc[1])
        mem[addrc][17:9] <= dinc[17:9];
end

always@(posedge clk)
begin
if(rda)
     douta <= mem[addra_reg];
end

always@(posedge clk)
begin
if(rdb)
     doutb <= mem[addrb_reg];
end
endmodule
```

SRS View



SRM View



Resource Usage Report

SLE 0 uses
Total Block RAMs (RAM1K18_RT) : 2 of 209 (0%)
Total LUTs: 0

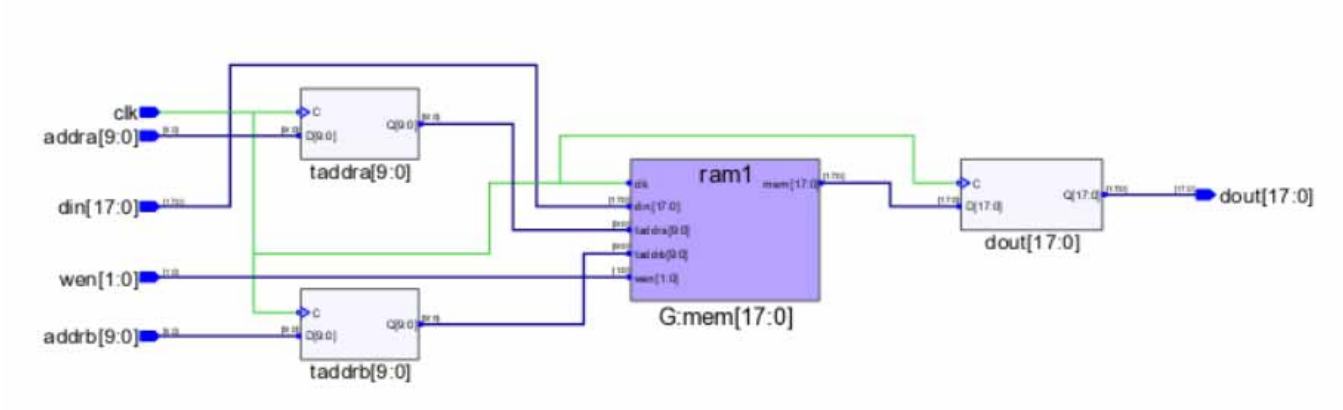
Example 46: RTL Coding Style for Two-Port RAM with Write Byte-Enable

In the RTL below, there is one write-enable (wen[1:0]) pin, which is used to control write access:

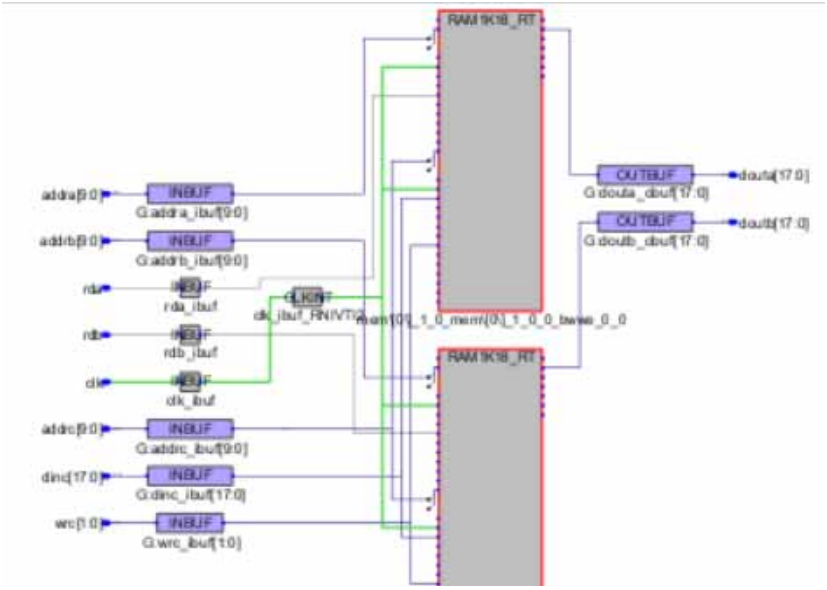
RTL

```
module ram_wb_wen_2addr(din ,dout, addra, addrb, clk, wen);

input [17:0] din;
input [1:0] wen;
input [9:0] addra;
input [9:0] addrb;
input clk;
output reg [17:0] dout;

localparam max_depth=1024;
localparam min_width=18;

reg [9:0] taddra;
reg [9:0] taddrb;
reg [min_width-1:0]  mem_ram[max_depth-1:0];

always @(posedge clk)
begin
   taddra<=addra;
   taddrb<=addrb;

   if(wen[0])
            mem_ram[taddra][8:0]<=din[8:0];

   if(wen[1])
            mem_ram[taddra][17:9]<=din[17:9];
end
always @(posedge clk)
begin
      dout <= mem_ram[taddrb];
end
endmodule
```

SRS View



SRM View



Resource Usage Report

SLE 10 uses
Total Block RAMs (RAM1K18_RT) : 1 of 209 (0%)
Total LUTs: 0

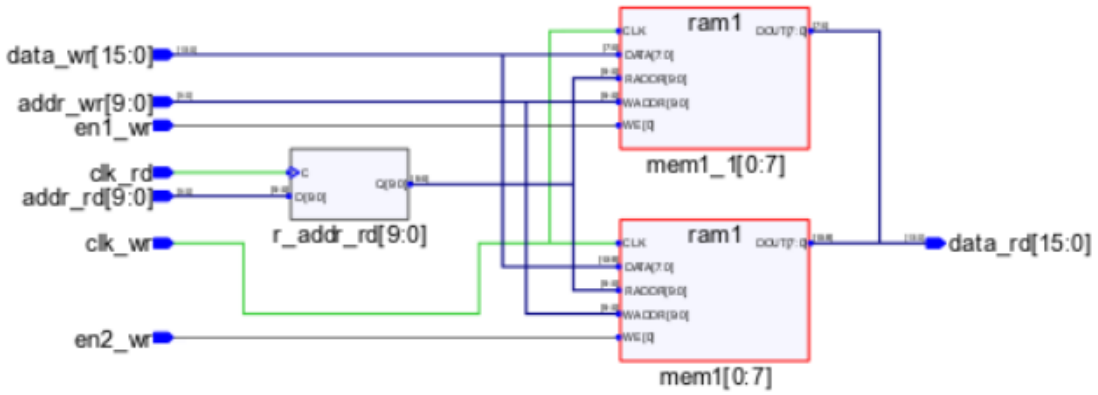
Example 47: VHDL RTL Coding Style for Two-Port RAM with Write Byte-Enables

In the RTL below, there are two write-enable (en1_wr and en2_wr) pins. They are used to control write access with different read and write clocks:

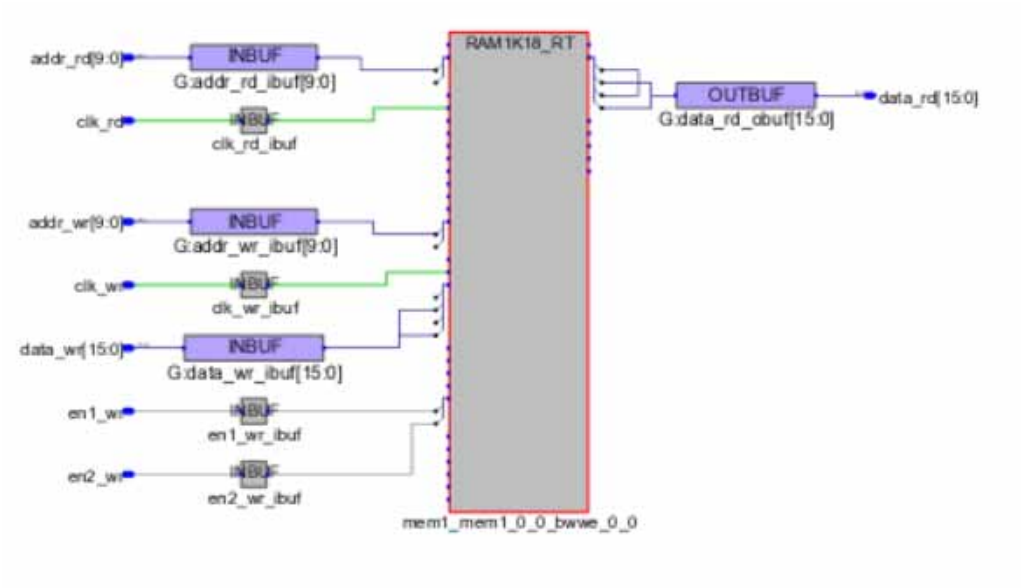
RTL

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity test_LSRAM_1kx16 is
   port (clk_wr: in std_logic;
         clk_rd: in std_logic;
         en1_wr: in std_logic;
         en2_wr: in std_logic;
         addr_wr: in std_logic_vector(9 downto 0);
         data_wr: in std_logic_vector(15 downto 0);
         addr_rd: in std_logic_vector(9 downto 0);
         data_rd: out std_logic_vector(15 downto 0)
         );
end test_LSRAM_1kx16;

architecture behave of test_LSRAM_1kx16 is

   type mem_type is array (1023 downto 0) of std_logic_vector(15 downto 0);
   signal MEM1: mem_type;
   signal r_addr_rd: std_logic_vector(9 downto 0);

begin

   process(clk_wr)
   begin

   if rising_edge(clk_wr) then

      if (en1_wr = '1') then
   MEM1(CONV_INTEGER(addr_wr(9 downto 0)))(7 downto 0)<= data_wr(7 downto 0);
end if;

if (en2_wr = '1') then
      MEM1(CONV_INTEGER(addr_wr(9 downto 0)))(15 downto 8) <= data_wr(15 downto 8);
end if;

   end if;
end process;

   data_rd <= MEM1(CONV_INTEGER(r_addr_rd));

process(clk_rd)
begin

   if rising_edge(clk_rd) then
      r_addr_rd  <= addr_rd;
      end if;

   end process;
end behave;
```

SRS View



SRM View



Resource Usage Report

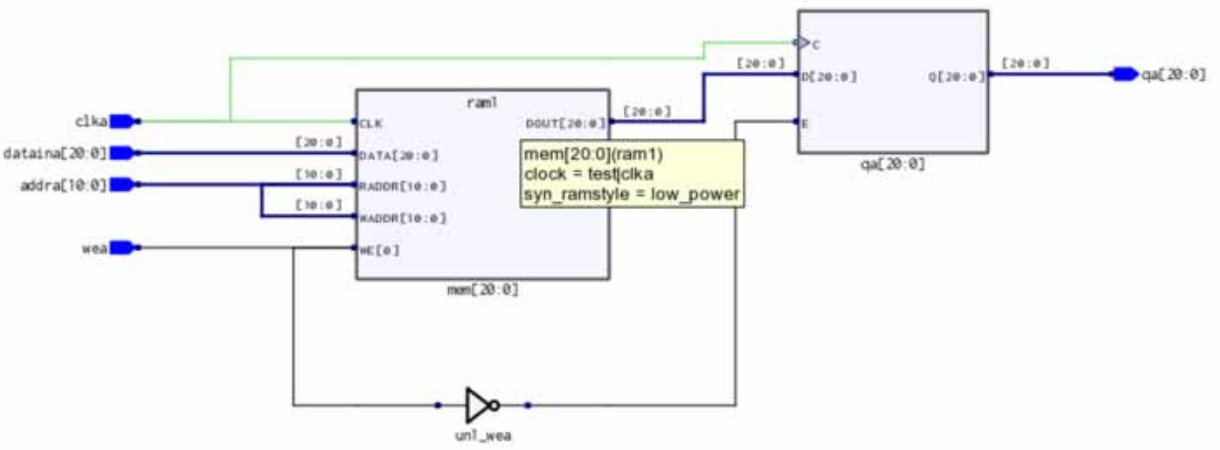
SLE 0 uses
Total Block RAMs (RAM1K18_RT) : 1 of 209 (0%)
Total LUTs: 0

Example 48: Single Port RAM with syn_ramstyle = "low_power" and Global Power 0ption is Off

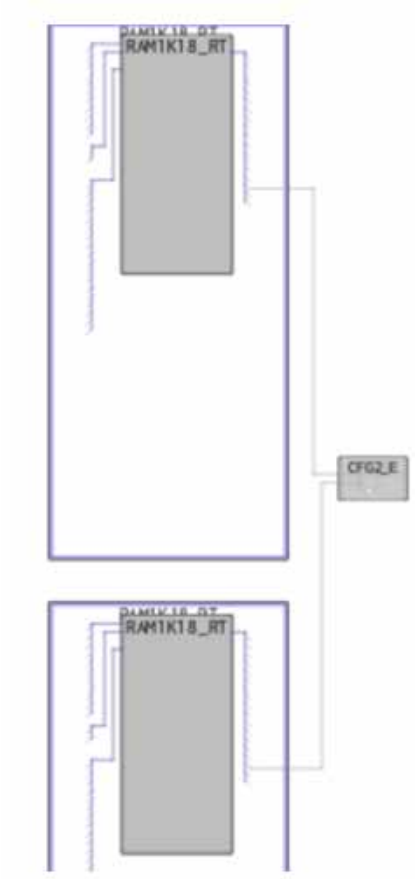
RTL

```
module test (clka,wea,addra,dataina,qa);
parameter addr_width = 11;
parameter data_width = 21;
input clka,wea;
input [data_width - 1 : 0] dataina;
input [addr_width - 1 : 0] addra;

output reg [data_width - 1 : 0] qa;

reg [data_width - 1 : 0] mem [(2**addr_width) - 1 : 0]/* synthesis syn_ramstyle =
"low_power" */;

always @ (posedge clka)
begin
   if(wea) mem[addra] <= dataina;
   else
         qa <= mem[addra];
end
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

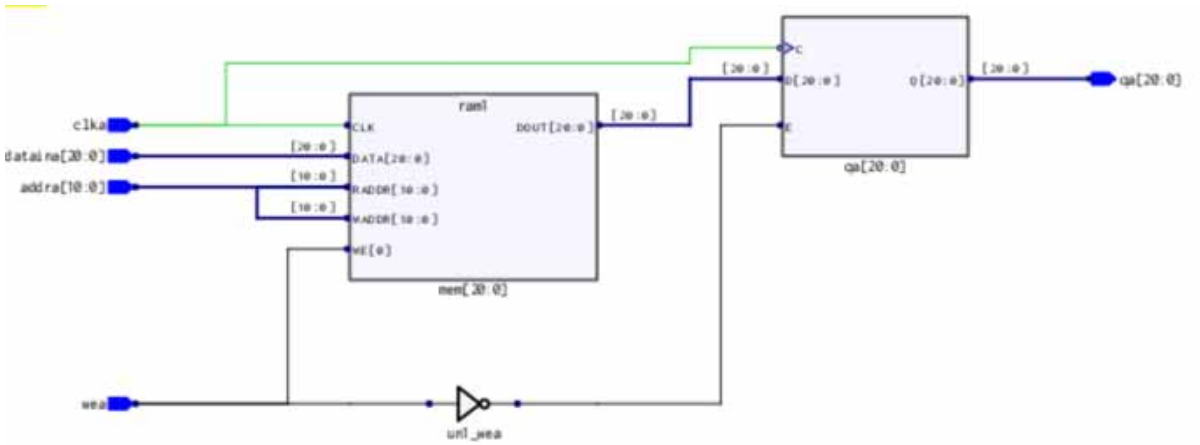
Cell usage:
CLKINT 1 use
CFG1 2 uses
CFG2 23 uses
CFG4 21 uses
SLE 23 uses
Block RAMs (RAM1K18_RT) : 4 - RAMs inferred in low-power mode
Total LUTs: 46

Example 49: Single Port RAM with syn_ramstyle = "no_low_power" and Global Power Option is On

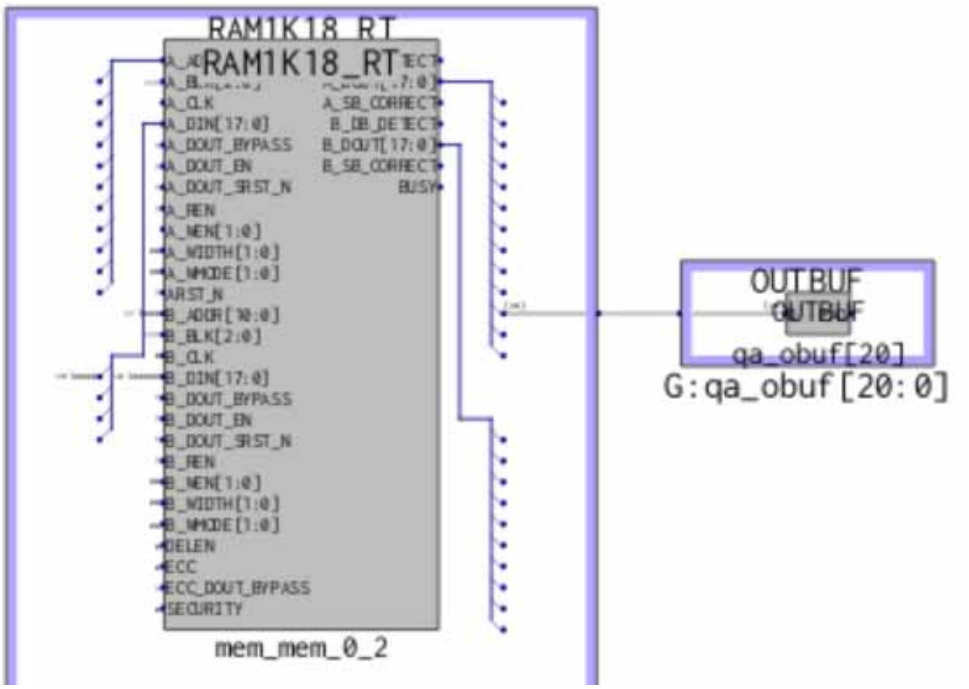
RTL

```
module test (clka,wea,addra,dataina,qa);
parameter addr_width = 11;
parameter data_width = 21;
input clka,wea;
input [data_width - 1 : 0] dataina;
input [addr_width - 1 : 0] addra;
output reg [data_width - 1 : 0] qa;

reg [data_width - 1 : 0] mem [(2**addr_width) - 1 : 0]/* synthesis syn_ramstyle =
"no_low_power" */;

always @ (posedge clka)
begin
      if(wea) mem[addra] <= dataina;
      else
            qa <= mem[addra];
end
endmodule
```

SRS (RTL) View



SRM (Technology) View



Resource Usage

Cell usage:
CLKINT 1 use
SLE 0 uses
RAM/ROM usage summary
Total Block RAMs (RAM1K18_RT) : 3 of 952 (0%)
Total LUTs: 0

Current Limitations

For successful RTG4 RAM inference with the Synplify Pro software, it is important that you use a supported coding structure, because there are limitations to what the synthesis tool infers. Currently, the tool does not support the following:

- ROM inference is not supported for RAM1K18_RT and RAM64X18_RT.
- Large RAMs are broken down into multiple RAM64X18_RT or RAM1K18_RT blocks. Only one type of RAM block can be used.
- RAMs which can be mapped into a single RAM primitive are not fractured on the address to infer multiple RAM blocks.