Digital Systems Design

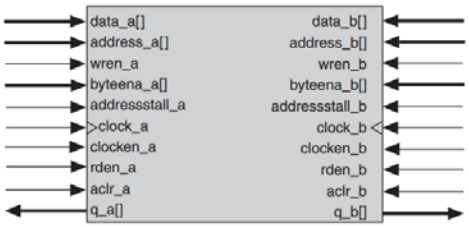
Memory Implementation on Altera CYCLONE IV Devices

Embedded Memory

- The CYCLONE IV embedded memory consists of columns of M9K memory blocks
- Each M9K block can implement various types of memory with or without parity, including true dual-port, simple dual-port, and single-port RAM, ROM, and FIFO buffers
- The M9K memory blocks include
  - input registers that synchronize writes and
  - output registers to pipeline designs and improve system performance
- M9K blocks offer a true dual-port mode to support any combination of two-port operations:
  - two reads, two writes, or one read and one write at two different clock frequencies
- When configured as RAM or ROM, an initialization file can be used to specify memory contents

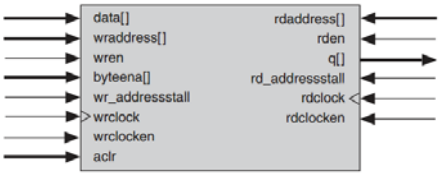
Dual-Port Memory Configuration

- True dual port operation
  - Two data input busses
  - Two data output busses
  - Two address busses
  - Two independent clocks
  - The memory blocks also enable mixed-width data ports for reading and writing to the RAM ports in dual-port RAM configuration
  - For example, the memory block can be written in ×1 mode at port A and read out in ×16 mode from port B

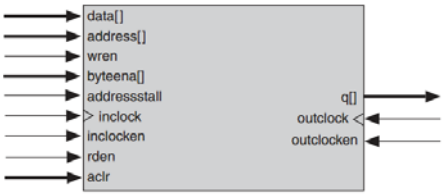


Simple Dual-Port & Single-Port Memory Configurations

- Simple Dual-Port Memory
  - One read port
  - One write port
  - Independent clocks



- Single-Port Memory
  - One address bus
  - Separate data input and output busses
  - One clock source
  - Two single-port memory blocks can be implemented in a single M9K block as long as each of the two independent block sizes is equal to or less than half of the M9K block size



Synchronous Memory

- The CYCLONE IV memory architecture can implement fully synchronous RAM by registering both the input and output signals to the M9K RAM block
- All M9K memory block inputs are registered, providing synchronous write cycles
- In synchronous operation, the memory block generates its own self-timed strobe write enable (wren) signal derived from a global clock

Asynchronous and Pseudo-Asynchronous Memory

- A circuit using asynchronous RAM must generate the RAM `wren` signal while ensuring its data and address signals meet setup and hold time specifications relative to the `wren` signal
- The output registers can be bypassed
  - Pseudo-asynchronous reading is possible in the simple dual-port mode of M9K blocks by
    - clocking the read enable and read address registers on the negative clock edge and
    - bypassing the output registers

Implementing Larger and/or Wider Memory

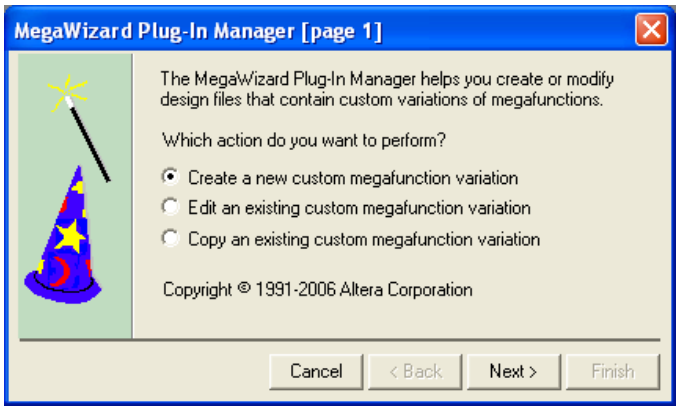
- The Quartus II software automatically implements larger memory by combining multiple M9K memory blocks
  - For example, two 256×16-bit RAM blocks can be combined to form a 256×32-bit RAM block
- M9K block usage is generally transparent to the designers VHDL code
  - M9K blocks are used as required by the design specifications (i.e. the memory length and width specified in the VHDL source)

Memory Implementation

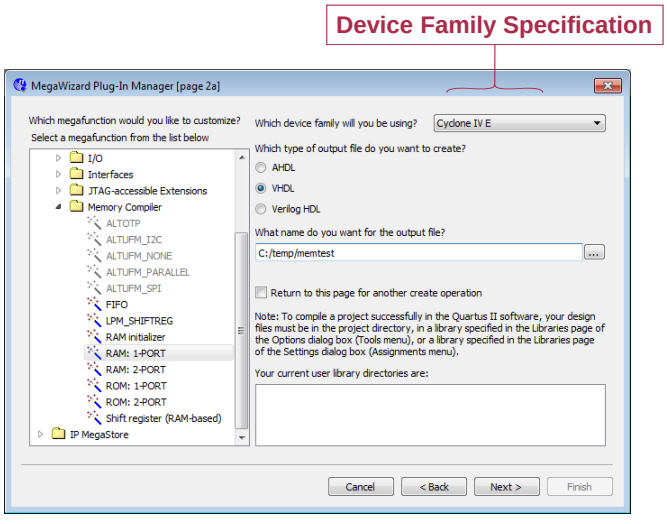
- Memory can be implemented by
  - Instantiation
    - Creating VHDL that creates an instance of a particular (predefined) memory component
    - Altera's `altsyncram` megafuncation will be most commonly used
    - Can write structural VHDL or use Quartus Megawizard Plug-in Manager to generate structural VHDL
  - Inference
    - Write behavioral VHDL that will model the memory
    - VHDL compiler can infer the memory
    - Will need to carefully construct VHDL code for the compiler to do this

MegaWizard Plug-In Manager

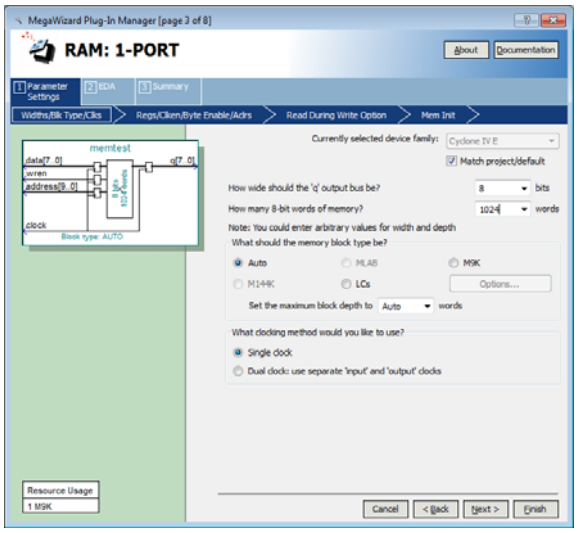
- Tools->Megawizard Plug-in Manager



MegaWizard



One Port RAM Specification

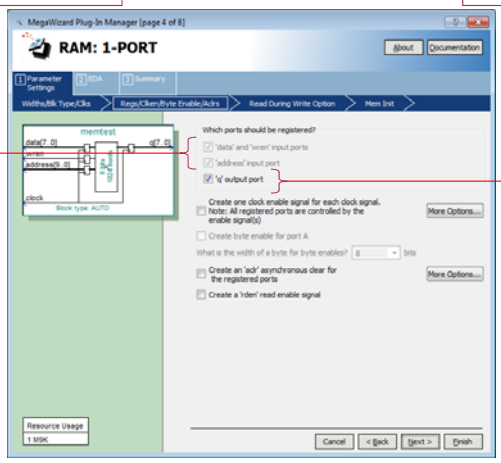


Data and address port sizes

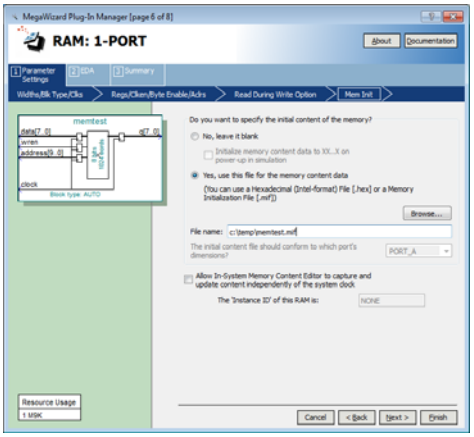
Port Registering

Input data and address will be registered

Output data will be registered



Memory Initialization



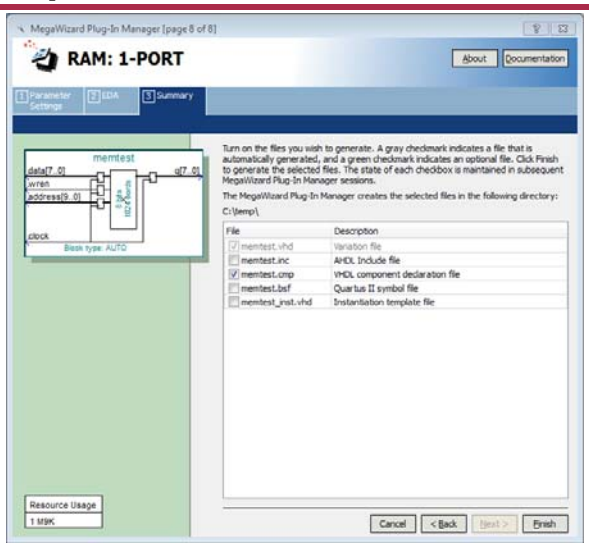

```
DEPTH = 1024;           % Memory Depth %
WIDTH = 8;              % Memory Width %

ADDRESS_RADIX = HEX;
% Address and value radixes are optional%
DATA_RADIX = HEX;

% Enter BIN, DEC, HEX, or OCT; unless          %
% otherwise specified, radixes = HEX           %

-- Specify values for addresses, which can be
-- single address or range
CONTENT
BEGIN
    0               : 01 ;
    1               : 02 ;
    2               : 03 ;
    [3..3FF]        : FF ;
% Addresses 0x003-0x3FF contain FF %
END ;
```


MegaWizard Plug-In Manager Completion



Memory VHDL Code

```
LIBRARY ieee;
USE     ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE     altera_mf.all;

ENTITY memtest IS
   PORT
   (
      address     : IN  STD_LOGIC_VECTOR (9 DOWNTO 0);
      clock       : IN  STD_LOGIC;
      data        : IN  STD_LOGIC_VECTOR (7 DOWNTO 0);
      wren        : IN  STD_LOGIC;
      q           : OUT STD_LOGIC_VECTOR (7 DOWNTO 0)
   );
END memtest;
```


Memory VHDL Code

```
ARCHITECTURE SYN OF memtest IS

   SIGNAL sub_wire0  : STD_LOGIC_VECTOR (7 DOWNTO 0);

   COMPONENT altsyncram
   GENERIC (
      clock_enable_input_a       : STRING;
      clock_enable_output_a      : STRING;
      init_file                  : STRING;
       .
       .
       .
      ram_block_type             : STRING;
      widthad_a                  : NATURAL;
      width_a                    : NATURAL;
      width_byteena_a            : NATURAL
   );
```

Memory VHDL Code

```
PORT (
        wren_a      : IN STD_LOGIC ;
        clock0      : IN STD_LOGIC ;
        address_a   : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
        q_a         : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
        data_a      : IN STD_LOGIC_VECTOR (7 DOWNTO 0)
  );
  END COMPONENT;

BEGIN
   q    <= sub_wire0(7 DOWNTO 0);
```


Memory VHDL Code

```
altsyncram_component : altsyncram
  GENERIC MAP (
     clock_enable_input_a => "BYPASS",
     clock_enable_output_a => "BYPASS",
     init_file => "c:/temp/memtest.mif",
     intended_device_family => "Cyclone IV E",
     lpm_hint => "ENABLE_RUNTIME_MOD=NO",
     lpm_type => "altsyncram",
     numwords_a => 1024,
     operation_mode => "SINGLE_PORT",
     outdata_aclr_a => "NONE",
     outdata_reg_a => "CLOCK0",
     power_up_uninitialized => "FALSE",
     widthad_a => 10,
     width_a => 8,
     width_byteena_a => 1
  )
```

Memory VHDL Code

```
 PORT MAP (
      wren_a    => wren,
      clock0    => clock,
      address_a => address,
      data_a    => data,
      q_a       => sub_wire0
   );

END SYN;
```


Viewing/Changing/Saving Memory

- Memory contents may be viewed
  - Prior to simulation
    - To verify initial contents
  - After simulation
    - To verify simulation results
- Memory contents may be saved to a new *.mif file
  - Import into other VHDL designs
  - Used by another program to verify simulation results
    - Example: MIF contents analyzed by a C or MATLAB program for verification

Memory Editor

- The Memory Editor allows you to enter, edit, and view the memory contents for a memory block implemented in an Altera device in a
  - Memory Initialization File (.mif)
  - Hexadecimal (Intel-Format) File (.hex)
- You can also use the Memory Editor to view and edit memory cells and their values during simulation.
  - You can create a new MIF or HEX File, and then specify the memory contents for a memory block in the design.
  - You can edit and adjust the memory cells and their values as needed before saving the MIF or HEX File.
- During simulation, you can open embedded memory at a breakpoint.
  - At each breakpoint, you can update current memory with simulation data, and then, if you wish, you can edit memory contents and update the Simulator with current memory contents.
- After simulation, you can view memory contents in the Simulation Report window in the Logical Memories section of the Simulation Report.

Viewing Memory

memtest.mif

Addr	+0	+1	+2	+3	+4	+5	+6	+7
000	01	02	03	FF	FF	FF	FF	FF
008	FF	FF	FF	FF	FF	FF	FF	FF
010	FF	FF	FF	FF	FF	FF	FF	FF
018	FF	FF	FF	FF	FF	FF	FF	FF
020	FF	FF	FF	FF	FF	FF	FF	FF
028	FF	FF	FF	FF	FF	FF	FF	FF
030	FF	FF	FF	FF	FF	FF	FF	FF
038	FF	FF	FF	FF	FF	FF	FF	FF
040	FF	FF	FF	FF	FF	FF	FF	FF
048	FF	FF	FF	FF	FF	FF	FF	FF
050	FF	FF	FF	FF	FF	FF	FF	FF
058	FF	FF	FF	FF	FF	FF	FF	FF
060	FF	FF	FF	FF	FF	FF	FF	FF
068	FF	FF	FF	FF	FF	FF	FF	FF
070	FF	FF	FF	FF	FF	FF	FF	FF
078	FF	FF	FF	FF	FF	FF	FF	FF
080	FF	FF	FF	FF	FF	FF	FF	FF
088	FF	FF	FF	FF	FF	FF	FF	FF
090	FF	FF	FF	FF	FF	FF	FF	FF
098	FF	FF	FF	FF	FF	FF	FF	FF
0a0	FF	FF	FF	FF	FF	FF	FF	FF
0a8	FF	FF	FF	FF	FF	FF	FF	FF
0b0	FF	FF	FF	FF	FF	FF	FF	FF
0b8	FF	FF	FF	FF	FF	FF	FF	FF
0c0	FF	FF	FF	FF	FF	FF	FF	FF
0c8	FF	FF	FF	FF	FF	FF	FF	FF
0d0	FF	FF	FF	FF	FF	FF	FF	FF
0d8	FF	FF	FF	FF	FF	FF	FF	FF
0e0	FF	FF	FF	FF	FF	FF	FF	FF

Inferred Memory

```
LIBRARY ieee;
USE     ieee.std_logic_1164.ALL;
USE     ieee.numeric_std.ALL;

ENTITY ram IS
GENERIC(
  ADDRESS_WIDTH   : integer := 4;
  DATA_WIDTH      : integer := 8
);

PORT(
  clock           : IN  std_logic;
  data            : IN  std_logic_vector(DATA_WIDTH - 1 DOWNTO 0);
  write_address   : IN  std_logic_vector(ADDRESS_WIDTH - 1 DOWNTO 0);
  read_address    : IN  std_logic_vector(ADDRESS_WIDTH - 1 DOWNTO 0);
  we              : IN  std_logic;
  q               : OUT std_logic_vector(DATA_WIDTH - 1 DOWNTO 0)
);
END ram;
```


Inferred Memory

```
ARCHITECTURE rtl OF ram IS
 TYPE RAM IS ARRAY(0 TO 2 ** ADDRESS_WIDTH - 1) OF
       std_logic_vector(DATA_WIDTH - 1 DOWNTO 0);
 SIGNAL ram_block : RAM;
BEGIN
PROCESS (clock)
BEGIN
  IF (clock'event AND clock = '1') THEN
    IF (we = '1') THEN
      ram_block(to_integer(unsigned(write_address))) <= data;
    END IF;
    q <= ram_block(to_integer(unsigned(read_address)));
  END IF;
END PROCESS;
END rtl;
```