White Paper

# Using APEX 20KE CAM with the Quartus Software Design Tool

## Introduction

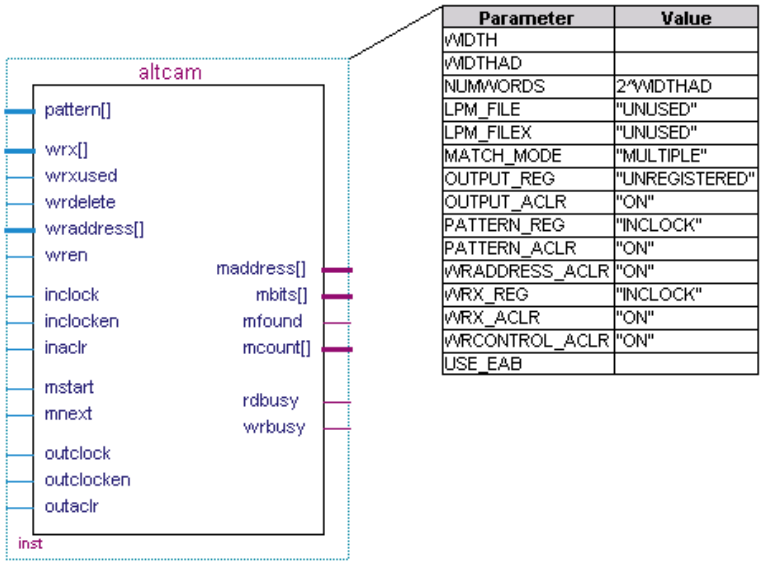
Content-addressable memory (CAM) is a form of memory that can be used to accelerate search applications. If the input pattern that is given to CAM matches one of the stored patterns, the address of the matching stored pattern is presented at its outputs. Altera® APEX™ 20KE family of devices supports ternary CAM, meaning that the stored patterns can also contain "don't care" bits along with binary 1 and 0 bits. CAM is implemented in the Quartus™ software (version 1999.10 and above) through the `altcam` megafunction.

This white paper describes how to use the Quartus software tool to implement CAM in your design. For information on how CAM can be used to optimize your design, see *AN 119 (Implementing High-Speed Search Applications with APEX CAM)*.

## The `altcam` Megafunction

Figure 1 shows the symbol for the `altcam` Megafunction. The set of parameters used to configure this megafunction is listed in Table 3.

*Figure 1. The `altcam` Megafunction*




| Parameter | Value |
|---|---|
| WIDTH | |
| WIDTHAD | |
| NUMWORDS | 2^WIDTHAD |
| LPM_FILE | "UNUSED" |
| LPM_FILEX | "UNUSED" |
| MATCH_MODE | "MULTIPLE" |
| OUTPUT_REG | "UNREGISTERED" |
| OUTPUT_ACLR | "ON" |
| PATTERN_REG | "INCLOCK" |
| PATTERN_ACLR | "ON" |
| WRADDRESS_ACLR | "ON" |
| WRX_REG | "INCLOCK" |
| WRX_ACLR | "ON" |
| WRCONTROL_ACLR | "ON" |
| USE_EAB | |

Table 1 describes the input pins of the altcam megafunction.

*Table 1. Input Ports of the altcam Megafunction*

| Port Name | Required | Description | Notes |
|---|---|---|---|
| pattern[] | Yes | Input data pattern for searching or writing | Input port WIDTH wide. |
| wrx[] | No | Pattern "don't care" bits (indicated with 1s), for writing only | Input port WIDTH wide. |
| wrxused | No | Indicates whether wrx[] should be used. | If false, writing takes two clock cycles to complete; if true, writing takes three clock cycles. If asserted during a write cycle, the value of the wrx[] port is used. Otherwise, the value of the wrx[] port has no effect. |
| wrdelete | No | Indicates that the pattern at wraddress[] should be deleted. | Deleting a pattern takes two clock cycles. pattern[], wrx[], and wrxused are ignored during delete cycles. |
| wraddress[] | No | Address for writing | Input port WIDTHAD wide. |
| wren | No | Write enable | Assert wren to start a write or delete operation. De-assert wren for a read (match) operation. |
| inclock | Yes | Clock for most inputs | |
| inclocken | No | Clock enable for inclock | |
| inaclr | No | Asynchronous clear for registers that use inclock | |
| mstart | No | Multi-match mode only: indicates that a new CAM read is starting and forces maddress[] to first match | This port is not available for single-match mode but required for multiple-match modes. In fast multiple-match mode, this port is required if the mnext port is used. |
| mnext | No | Multi-match only: advances maddress[] to next match | This port is not available for single-match mode. |
| outclock | No | Clock for mstart, mnext, and outputs | Used only if OUTPUT_REG="OUTCLOCK". If OUTPUT_REG="UNREGISTERED" or "INCLOCK" this port must remain unconnected. |
| outclocken | No | Clock enable for outclock | Used only if OUTPUT_REG="OUTCLOCK". If OUTPUT_REG ="UNREGISTERED" or "INCLOCK" this port must remain unconnected. |
| outaclr | No | Asynchronous clear for registers that use outclock | |

Table 2 describes the output pins of the altcam megafunction.

*Table 2. Output Ports of the altcam Megafunction*

| Port | Required | Description | Comments |
|---|---|---|---|
| maddress[] | No | Encoded address of current match. | Output port WIDTHAD wide. One of the output ports must be used. Altera recommends using either a combination of the maddress[] and mfound output ports, or the mbits[] output port. |
| mbits[] | No | Address of the found match. | Output port with width [NUMWORDS-1..0]. One of the output ports must be present. Altera recommends using either a combination of the maddress[] and mfound output ports, or the mbits[] output port. |
| mfound | No | Indicates at least one match. | Used with the maddress[] port. One of the output ports must be present. Altera recommends using either a combination of the maddress[] and mfound output ports, or the mbits[] output port. |
| mcount[] | No | Total number of matches. | Output port WIDTHAD wide. One of the output ports must be present. Altera recommends using either a combination of the maddress[] and mfound output ports, or the mbits[] output port. |
| rdbusy | No | Indicates that read input ports must hold their current value. | One of the output ports must be present. |
| wrbusy | No | Indicates that write input ports must hold their current value. | One of the output ports must be present. |

Table 3 list the parameters that are used to configure the altcam megafunction.

*Table 3. The altcam Megafunction Parameters*

| Parameter | Type | Required | Description |
|---|---|---|---|
| WIDTH | Integer | Yes | Width of the input pattern and stored patterns. |
| WIDTHAD | Integer | Yes | Width of wraddress[] port. WIDTHAD should be equal to CEIL [LOG2(NUMWORDS)]. |
| NUMWORDS | Integer | Yes | Number of words stored in memory. It indicates the width of the mbits[] port. In general, mbits[] value should be 2^ (WIDTHAD-1) < NUMWORDS <= 2^ WIDTHAD. |
| LPM_FILE | String | No | Name of the Memory Initialization File (**.mif**) or Hexadecimal (Intel-format) File (**.hex**) containing RAM initialization data (<*file name*>), or UNUSED. If omitted, contents default to "never match". |
| LPM_FILEX | String | No | Name of the second HEX File containing RAM initialization data (<*filename_xu.hex*>). If omitted the default is UNUSED. Bits that are 1 in this file change the meaning of the bits in the first HEX File such that the 0 bits in the first file become “don't care” bits, and the 1 bits become "never match" bits in CAM patterns. The 0 bits in this file preserve the normal meaning of the bits in the first HEX File. |
| MATCH_MODE | String | Yes | Selects between single-match mode and one of two multiple-match modes. The values are SINGLE, MULTIPLE, and FAST_MULTIPLE. If omitted, the default is MULTIPLE. |
| OUTPUT_REG | String | No | Indicates whether the outputs should be registered. Values are UNREGISTERED, INCLOCK, and OUTCLOCK. If omitted, the default is UNREGISTERED. |
| OUTPUT_ACLR | String | No | Indicates whether the outaclr port should affect the output registers. Values are ON and OFF. If omitted, the default is ON. |
| PATTERN_REG | String | No | Indicates whether pattern[] should be registered. Values are UNREGISTERED and INCLOCK. If omitted, the default is INCLOCK. |
| PATTERN_ACLR | String | No | Indicates whether the inaclr port should affect the pattern[] registers. Values are ON and OFF. If omitted, the default is ON. |
| WRADDRESS_ACLR | String | No | Indicates whether the inaclr port should affect the wraddress[] registers. Values are ON and OFF. If omitted, the default is ON. |
| WRX_REG | String | No | Indicates whether the wrx[] and wrxused ports should be registered. Values are UNREGISTERED, and INCLOCK. If omitted, the default is INCLOCK. |
| WRX_ACLR | String | No | Indicates whether the inaclr port affects the wrx[] and wrxused registers. Values are ON and OFF. If omitted, the default is ON. |
| WRCONTROL_ACLR | String | No | Indicates whether the inaclr port affects the wren register. Values are ON and OFF. If omitted, the default is ON. |

## Writing Patterns into CAM

Writing a new pattern in altcam or replacing its stored patterns with new patterns involves the use of the pattern[], wrx[], wrxused, wrdelete, wren, and wraddress[] ports. Patterns without “don’t care” bits can be written in two clock cycles, and those with “don’t care” bits require three clock cycles. During all write cycles, wren must be asserted and wraddress[] and pattern[] must remain unchanged.

If the pattern does not contain “don’t care” bits, then asserting pattern[], wren, and wraddress[] for two clock cycles is sufficient. “Don’t care” bits can be added by using the wrx[] port. Bits with 0 in wrx[] mark valid pattern bits, and bits with 1 in wrx[] mark “don’t care” pattern bits. When the wrx[] port is used, the wrx[], wrxused, pattern[], wren, and wraddress[] must be asserted for three clock cycles.

CAM entries can also be deleted by asserting wrdelete and wren for two clock cycles, during which wraddress[] should indicate the address containing the data that is to be deleted. The pattern[], wrx[], and wrxused inputs are ignored during delete cycles.

CAM can be initialized using MIF or Intel HEX format files during device configuration. The MIF format supports “don’t care” and “never match” bits. These extra bits are also supported in the Intel HEX format by using a second

HEX file. One file is used to initialize the data (0 and 1), and a second file is used to set the "don't care" and "never match" bits. If the optional second initialization file is used, it must be named <*file_xu*>.hex if the first initialization file is named <*file*>.hex. The bits that are to be matched exactly are defined by the values 0 or 1 in <*file*>.hex and 0 in <*file_xu*>.hex. All "don't care" bits that are matched in the CAM must have a value of 0 in <*file*>.hex and a value of 1 in <*file_xu*>.hex. If a word in <*file*>.hex contains a 1 that has a corresponding bit in the <*file_xu*>.hex that is also set to 1, that word will never be matched. This is shown in Table 4.

*Table 4. Format of HEX and MIF Initialization Files*

| **<*file*>.hex** | **<*file_xu*>.hex** | **<*file*>.mif Equivalent** |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |
| 0 | 1 | X |
| 1 | 1 | U |

## Reading from CAM

To read patterns/addresses from `altcam`, three different modes can be used:

- Single-match mode
- Multiple-match mode
- Fast Multiple-match mode

In multiple-match and fast multiple-match mode, an external priority encoder generates the encoded match address output `maddress[]`. As a result, when reading patterns in either of the multiple-match modes, the encoding logic will generally result in higher logic utilization than with single-match mode.

In all three modes, both encoded (`maddress[]`) and unencoded (`mbits[]`) outputs are available. External logic generates the `mfound` and `mcount[]` signals, which give the total number of matches.

## Single-Match Mode

In single-match mode (`MATCH_MODE = "SINGLE"`), only one inclock clock cycle is needed to read stored data from `altcam`.

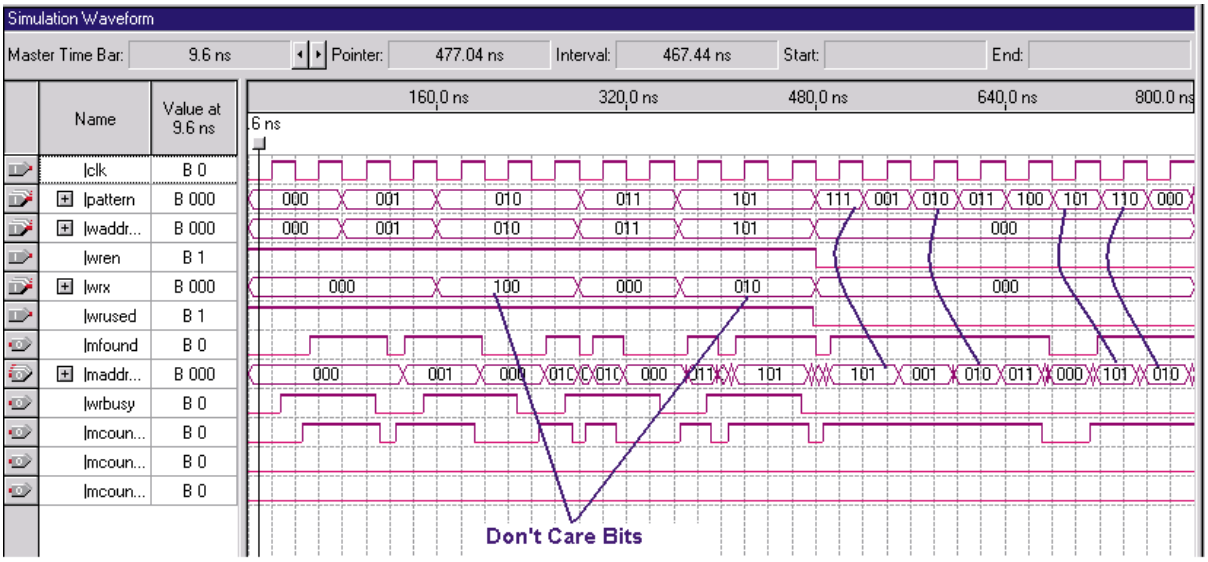
When an input pattern matches one of the stored patterns in `altcam`, match flag `mfound` will be asserted, and the address of the match will be presented on `maddress[]`. Output port `mbits[]` gives the unencoded version of the match. The output that indicates the number of matches (`mcount[]`) is always either the value 0 or 1 in this mode.

It is very important to note that in the single-match mode, `altcam` will not operate properly if there are multiple patterns stored that match the same input pattern. If this situation occurs, the Quartus software will give a warning during simulation indicating that CAM contains multiple matches.

In single-match mode, the `altcam` megafunction will support CAMs deeper than 32 words by using multiple embedded system blocks (ESBs). For input patterns with widths greater than 32 bits, `altcam` will automatically switch to fast multiple- match mode.

In order to write "don't care" bits into `altcam`, `wrused` should be asserted high, and `waddress[]`, `pattern[]` and `wren[]` should be valid for three clock cycles. The bits in `wrx[]` with `1` indicate "don't care" bits. For example, in Figure 2, at `waddress[] 010` the `wrx[]` is `100`, which means that the third bit is a "don't care" bit. As a result, reading `110` or `010` flag a match at `maddress[] 010`.

*Figure 2. Waveform for Single-Match Mode with "Don't Care'" Bits*




## Multiple-Match Mode

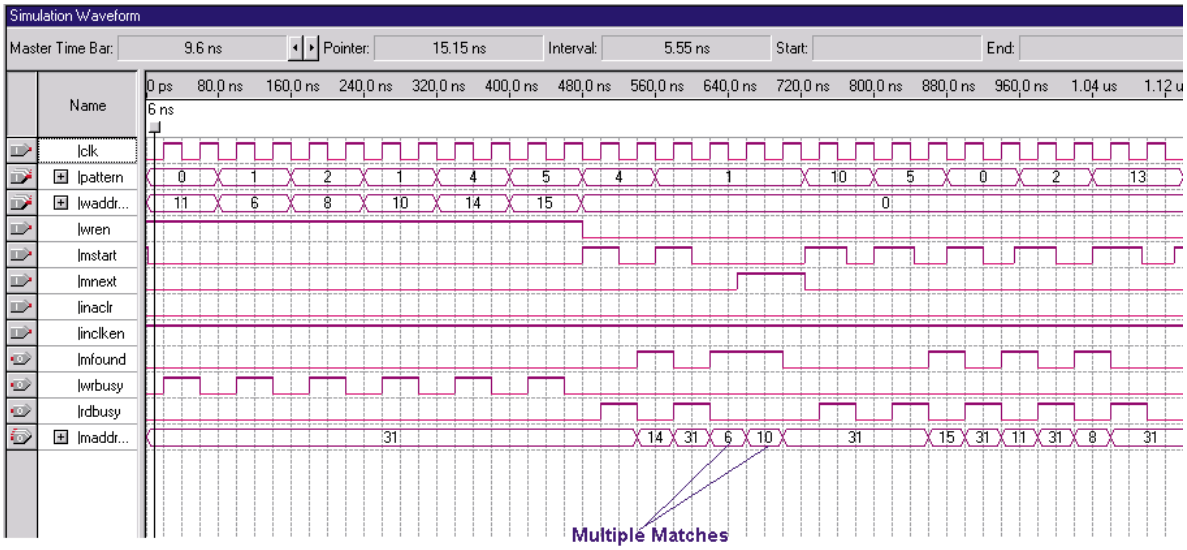
In multiple-match mode (`MATCH_MODE = "MULTIPLE"`), the megafunction takes two `inclock` clock cycles to read from `altcam` and generate valid outputs. This happens because the ESB generates 16 outputs at each clock cycle. As a result, two cycles are required to generate all 32 outputs from an ESB.

To search `altcam` for a new pattern, the pattern data should be applied to the `pattern[]` port, and the `mstart` input should be asserted high for the first clock cycle during the read cycle. If the input pattern matches any of the stored patterns, `mfound` asserts high, and `maddress[]` gives the address of the first match (after a two-cycle delay). Other match addresses can be generated on subsequent clock cycles by asserting `mnext` and holding it high for no more than two clock cycles after `mstart`. Output port `mbits[]` gives the unencoded version of the matches. Output port `mcount[]` counts the total number of matches.

In this mode, each ESB supports 31 bits of data because the most significant bit (MSB) is used to select between the even or odd outputs of ESB at each clock cycle. But multiple-match mode supports both deeper and wider CAMs by cascading 32-word × 31-bit ESBs.

Figure 3 shows the functional simulation waveform for multiple-match mode. In multiple-match mode, the `mstart` provides the lowest match address location, and `mnext` provides the consequent match locations on `maddress[]`. Signal `mnext` should be asserted not more than two clock cycles after `mstart` is asserted. In this example, data `1` has been written in two locations: `6` and `10`. Asserting `mstart` provides address location `6` on the `maddress[]` port, and asserting `mnext` enables CAM to read out the consequent location, `10`.

*Figure 3. Waveform for Multiple-Match Mode*



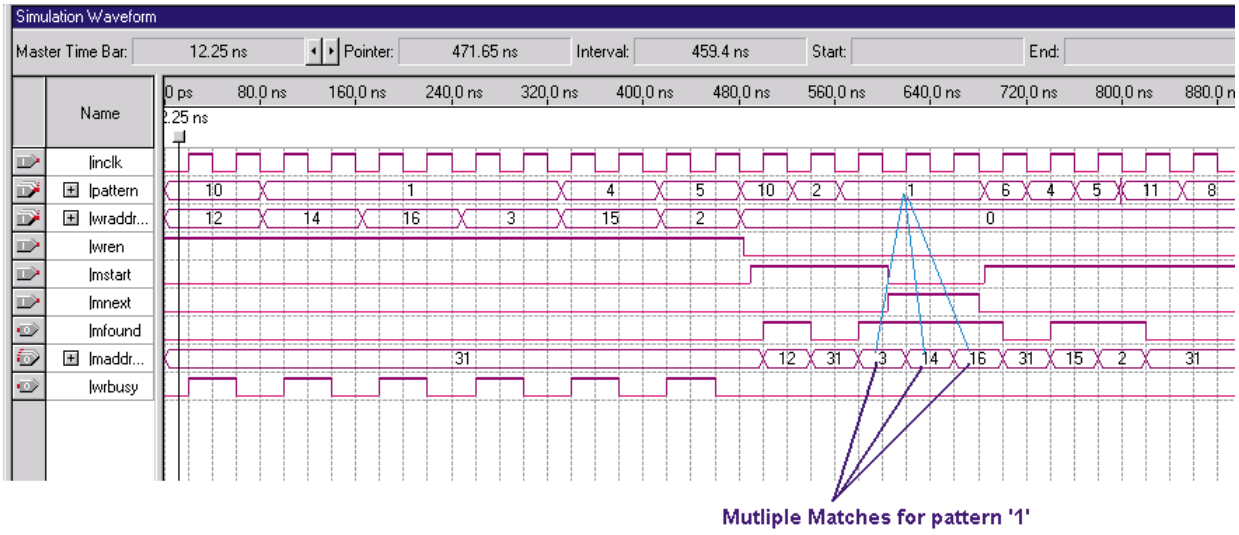

## Fast Multiple-Match Mode

Fast multiple-match mode (`MATCH_MODE = "FAST_MULTIPLE"`) is identical to multiple-match mode except that fast multiple-match mode only takes one `inclock` clock cycle to read from `altcam` and generate valid outputs. However, this quick generation uses only half of the memory available in each ESB. As a result, ESB utilization is higher, but data can be read out of `altcam` in one cycle.

Most of the input and output ports used in fast multiple-match mode are identical to multiple-match mode with a few exceptions. Ports such as `maddress[]`, `mcount[]`, `mfound`, `pattern[]`, `wrx[]`, and `wren` function the same as in multiple-match mode. The `rbusy` port is not used in fast multiple-match mode because the read does not exceed one clock cycle. Ports such as `mstart` and `mnext` are not required for this mode if the location of the matched address is not required (if `maddress[]` is not used), and only the `mbits[]` output gives the unencoded version of a matching address. If the `maddress[]` output port is used, `mstart` and `mnext` must be used to give the first and next matching addresses.

In this mode, the `altcam` megafunction supports CAMs deeper and wider than 32 words and bits by cascading the ESBs. Figure 4 shows the functional simulation result of a fast multiple-match mode.

*Figure 4. Waveform for Fast Multiple-Match Mode*




## CAM Mode Comparison

In order to compare the performance and utilization of the different CAM modes, a 32-word × 32-bit CAM was compiled for an EP20K200E-1 device. Table 5 shows the results of this comparison.

*Table 5. CAM Mode Comparison (32x32)*

| Feature | Single-Match Mode | Multiple-Match Mode | Fast Multiple-Match Mode |
|---|---|---|---|
| ESBs used | 1 | 1 | 2 |
| Logic Element used | 35.0 | 98 | 79 |
| $f_{MAX}$ | 198.89 (MHz) | 94.45 (MHz) | 190.91 (MHz) |

## Resource Usage

One ESB can implement a 32-word × 32-bit CAM. Table 6 shows the resource usage for the `altcam` megafunction.

*Table 6. Resource Usage for the `altcam` Megafunction (32x32)*

| Match Mode | CAM Patterns per ESB | CAM Pattern Bits per ESB |
|---|---|---|
| Single-match mode | 32 | 32 |
| Multiple-match mode | 32 | 31 |
| Fast multiple-match mode | 16 | 32 |

## MegaWizard Interface

The MegaWizard™ allows users to specify options for the custom megafunction variations. The MegaWizard asks questions about the preferred values for parameters or optional ports.

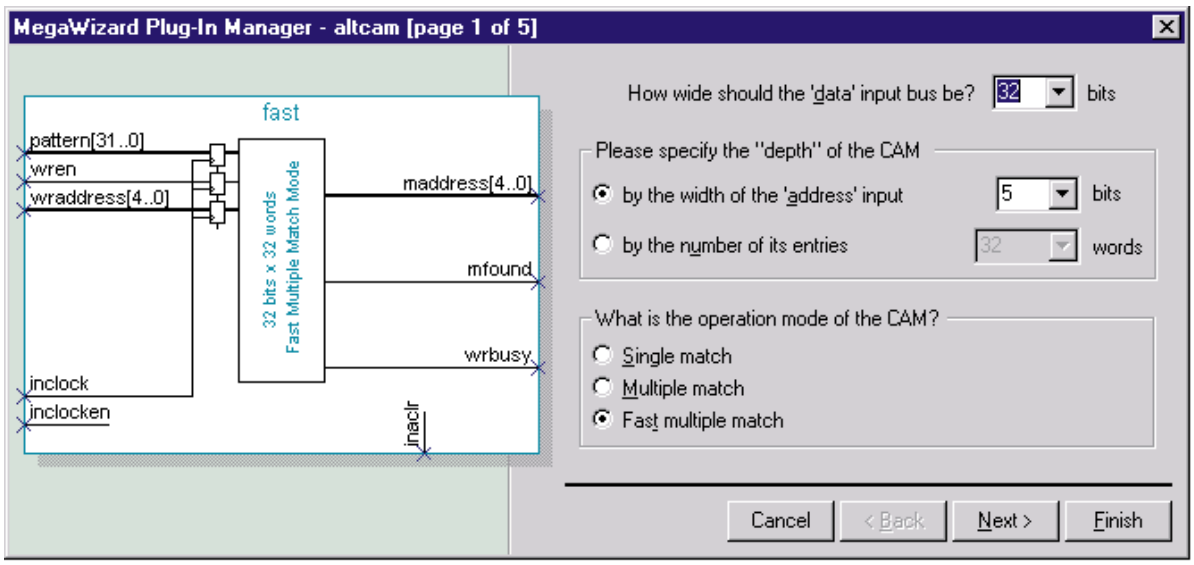
The MegaWizard Plug-In Manager automatically generates a Component Declaration file (**.cmp**) that can be used in VHDL Design Files (**.vhd**) and an Include File (**.inc**) that can be used in Text Design Files (**.tdf**) and Verilog Design Files (**.v**).

Users can start the MegaWizard Plug-In Manager in one of the following ways:

- Choose the **MegaWizard Plug-In Manager** command (`Tools` menu).
- When working in the Block Editor, click **MegaWizard Plug-In Manager** in the symbol dialog box (Insert menu).

Figure 5 shows page 1 of the `altcam` MegaWizard Plug-In Manager.

*Figure 5. Page 1 of `altcam` MegaWizard Plug-In Manager*




## Data Input Bus Width

This option allows users to select the width of the input `pattern[]` of the designed `altcam`. The pull-down menu shows the value, which goes up to 256 bits. For widths higher than 256 bits, the value must be typed in.

## Depth of CAM

This option specifies the number of word lines to the `altcam` megafunction. Two options are available: entering the number by its entries (word lines) or by the width of `address` bits.

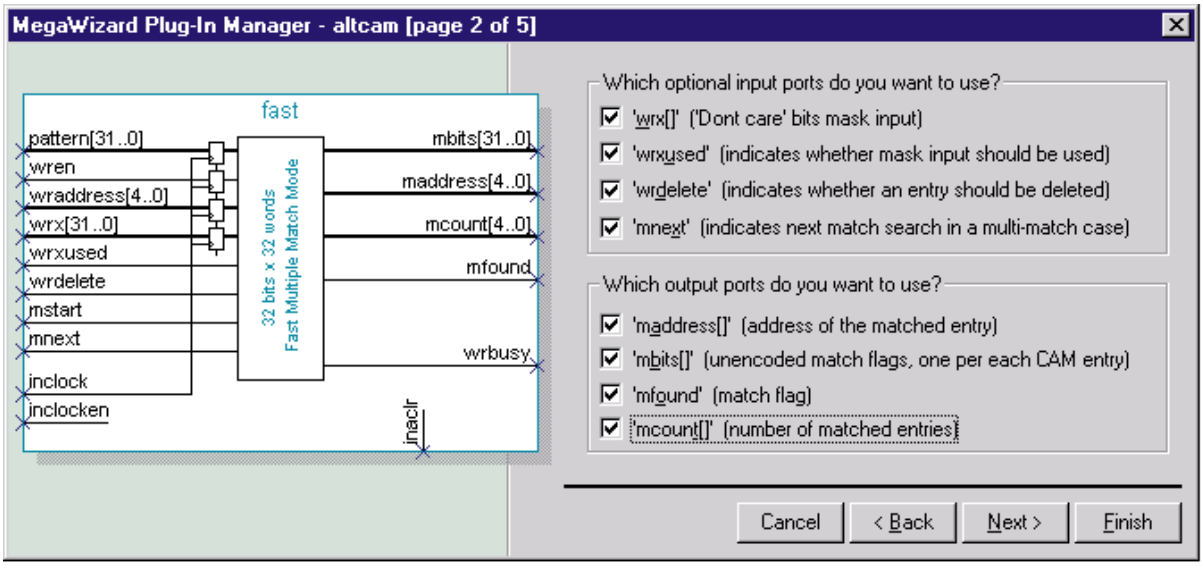
## Operation Mode of CAM

There are three options for `altcam` operation mode.

- Single match mode: Read occurs in one clock cycle, but it does not support multiple match
- Multiple match mode: Read occurs in two-clock cycles and supports multiple matches
- Fast multiple match mode: Read occurs in one clock cycle and supports multiple matches, but it only uses half of an ESB

Figure 6 shows the different options that can be selected on page 2 of the MegaWizard Plug-in Manager.

*Figure 6. Page 2 of the altcam MegaWizard*




## Optional Input Ports

`Wrxused` and `wrx[]` are the inputs that are used to write "don't care" bits into the `altcam` megafunction. When `wrxuxed` is asserted, "don't care" bits will be written into `altcam`.

`wrdelete` is the input that used to delete patterns from `altcam`.

`mnext` and `mstart` are an input pair that indicates the location of the first and subsequent matches on the `maddress[]` outputs. In fast multiple-match mode, the `mstart` and `mnext` pair are optional, but selecting `mnext` on page 2 of the MegaWizard will add the `mstart` input into the wizard. `mstart` is automatically included in the wizard upon selection of multiple-match mode.

## Output Ports

`maddress[]` gives the address of the match entry.

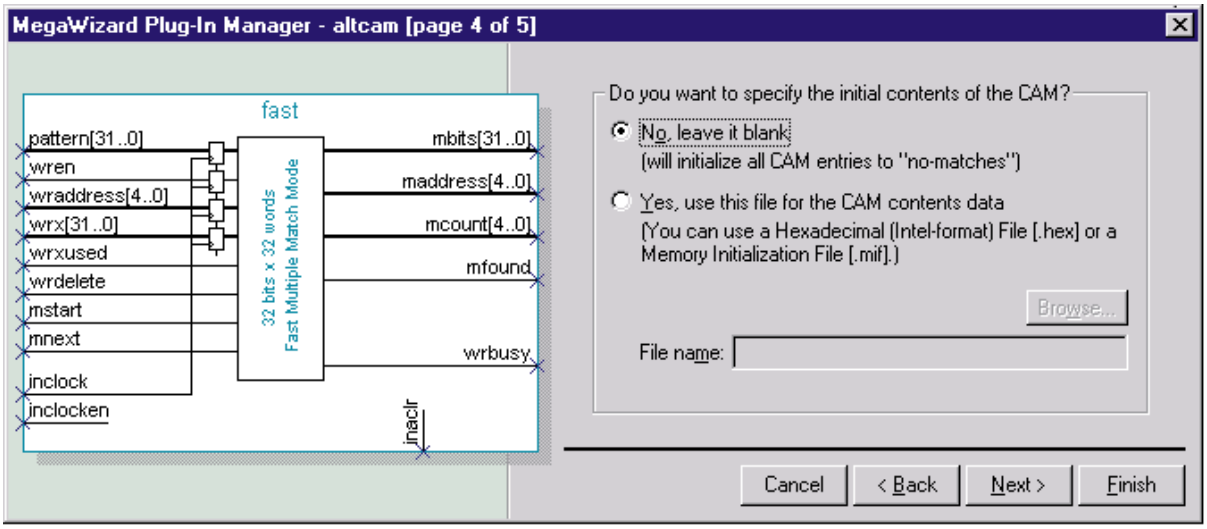
`mbits[]` gives the unencoded version of the match location. In multiple-match mode or in fast multiple-match mode, only selecting `mbits[]` (instead of `maddress[]`) will reduce the logic cell utilization because the external logic used to encode the unencoded outputs will not be implemented.

When in multiple-match mode or fast multiple-match mode, `mcount[]` gives the total number of matches found in `altcam`. In single-match mode, `mcount[]` only has a value of 0 or 1 because multiple match is not supported in this mode.

`mfound` is the output that indicates whether any match was found.

Fig 7 shows page 7 of the `altcam` MegaWizard, in which you can specify the memory initialization file.

*Figure 7. Page 4 of the altcam MegaWizard*




## Specifying the Initial Contents of altcam

If you select to specify the initial contents of altcam, then the initial memory file should be generated. If you select not to specify the initial contents of altcam, then data should be written to altcam after configuring the device.

Two types of memory files exist: Memory Initialization Files **(.mif)** and Hexadecimal Files (**.hex)**.

- MIFs specify the pre-loaded pattern in the altcam. Only one file is needed to load patterns incorporating 1, 0, X ("don't care" bits) and U ("never match" bits). This is an Altera file format and can only be used for CAM functions that are implemented in the Quartus software. MIFs are not compatible with external simulators.
- HEX files require two files to be created:
  <*file_name*>.hex for 1 and 0 patterns
  <*file_name*>_ux.hex for U and X

HEX files allow users to use the CAM function in third-party behavioral simulation.

## Examples

Figure 8 shows an altcam MegaWizard instantiation in single-match mode. In this example, "don't care" bits will be written into the altcam through "don't care" ports. Also, the outputs have been registered.

*Figure 8. Single-Match Mode `altcam` With "Don't Care" Ports*

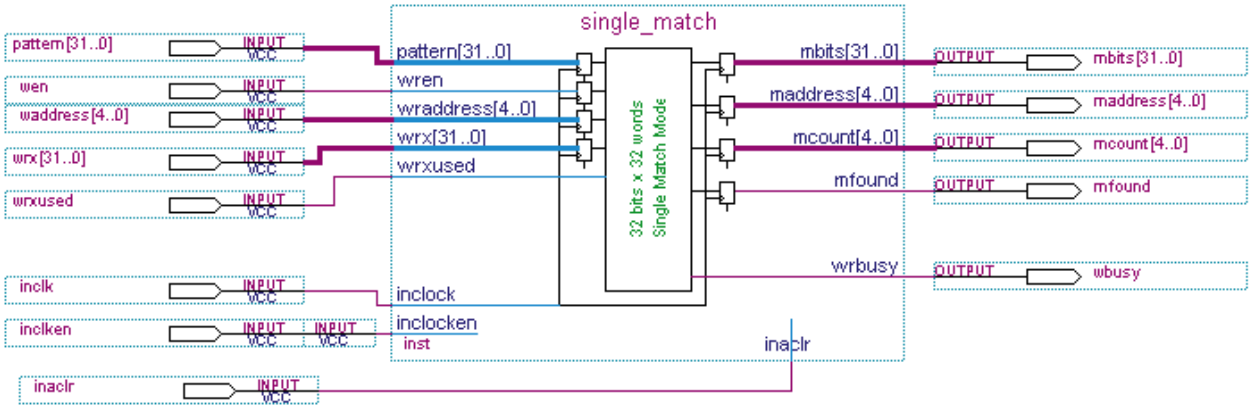
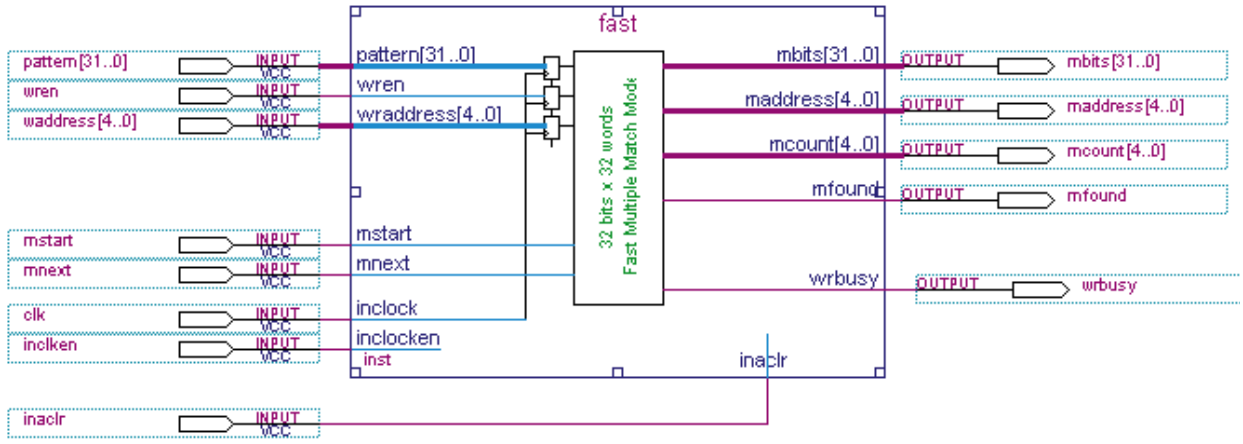


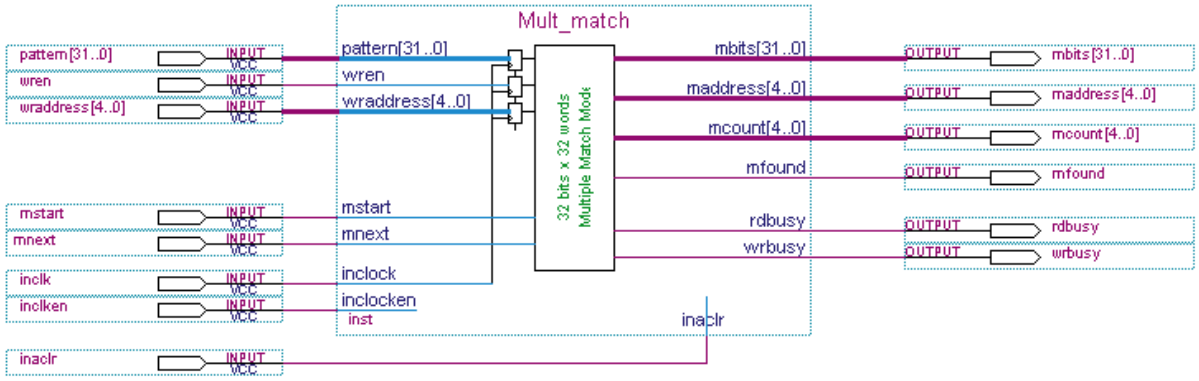

Figure 9 shows the instantiation of fast multiple-match mode. In this example, only 1 and 0 will be written into `altcam`.

*Figure 9. Fast Multiple-Match Mode `altcam`.*




When selecting multiple-match mode in the `altcam` MegaWizard (Figure 10), the `mstart` input will be automatically selected as one of input ports. The user has the option of selecting `mnext` as an input in the MegaWizard if detecting the address of all matches is required.

*Figure 10. Multiple-Match Mode `altcam` MegaWizard*




## VHDL Instantiation

The following example code instantiates the `altcam.vhd` behavioral model for the `altcam` megafunction. This particular example instantiates the function, passes parameters and connects the ports of the `altcam` to input and output pins to demonstrate the functionality of the CAM. The `altcam` function can be directly instantiated as shown here along with other RTL code for simulation. This example uses files called `cam.hex` and `cam_xu.hex` to initialize the contents. Both files are shown at the end of this document.

Cam32 × 8 instantiates the `altcam` behavioral function. This can contain code other than just the `altcam` function call.

### Cam32×8.vhd

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

entity cam32x8 is
        port
            ( pattern:    in std_logic_vector(7 downto 0);
              wrx:     in std_logic_vector(7 downto 0);
      wrxused: in std_logic;
              wrdelete:   in std_logic;
              wraddress:  in std_logic_vector(4 downto 0);
              wren:       in std_logic;
              inclock:    in std_logic;
              mstart:     in std_logic;
              mnext:      in std_logic;
              maddress:   out std_logic_vector(4 downto 0);
              mbits:      out std_logic_vector(31 downto 0);
              matchfound: out std_logic;
              mcount:     out std_logic_vector(4 downto 0);
              rdbusy:     out std_logic;
              wrbusy:     out std_logic );
end cam32x8;

architecture apex of cam32x8 is
```

```
component altcam
      generic
            ( width:            positive;
              widthad:          positive;
              numwords:         natural := 0;
              lpm_file:         string := "UNUSED";
              lpm_filex:        string := "UNUSED";
              match_mode:       string := "SINGLE";
              output_reg:       string := "UNREGISTERED";
              output_aclr:      string := "OFF";
              pattern_reg:      string := "INCLOCK";
              pattern_aclr:     string := "Off";
              wraddress_aclr:   string := "Off";
              wrx_reg:       string := "INCLOCK";
              wrx_aclr:      string := "off";
              wrcontrol_aclr:   string := "OFF" );

        port
            ( pattern:    in std_logic_vector(width -1 downto 0);
              wrx:        in std_logic_vector(width -1 downto 0);
              wrxused:    in std_logic;
              wrdelete:   in std_logic;
              wraddress:  in std_logic_vector(widthad-1 downto 0);
              wren:       in std_logic;
              inclock:    in std_logic;
              inclocken:  in std_logic := '1';
              inaclr:     in std_logic := '0';
              mstart:     in std_logic;
              mnext:      in std_logic;
              outclock:   in std_logic := '0';
              outclocken: in std_logic := '1';
              outaclr:    in std_logic := '0';
              maddress:   out std_logic_vector(widthad-1 downto 0);
              mbits:      out std_logic_vector(numwords-1 downto 0);
              mfound:     out std_logic;
              mcount:     out std_logic_vector(widthad-1 downto 0);
              rdbusy:     out std_logic;
              wrbusy:     out std_logic );

end component;
begin
U0: altcam
   generic map (width => 8, widthad => 5, lpm_file => "cam.hex",
lpm_filex => "cam_xu.hex", numwords => 32, match_mode => "MULTIPLE",
output_reg => "UNREGISTERED")
port map (pattern => pattern, wrx => wrx, wrxused => wrxused,
wrdelete => wrdelete, wraddress => wraddress, wren => wren, inclock => inclock,
mstart => mstart, mnext => mnext,
maddress => maddress, mbits => mbits, mfound => matchfound,  mcount => mcount,
rdbusy => rdbusy, wrbusy => wrbusy );
end apex;
```

## Testbench for 32 X 8 CAM in VHDL

Cam_testbench.vhd is an example testbench that demonstrates the functionality of the altcam in multiple-match mode with "don't care" bits.

### cam_testbench.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.std_logic_arith.ALL;
USE std.textio.ALL;

ENTITY CAM_testbench IS
END CAM_testbench;

ARCHITECTURE testbench OF CAM_testbench IS

      SIGNAL countclock:std_logic := '0';

COMPONENT cam32x8
PORT(

              pattern:    in std_logic_vector(7 downto 0);
              wrx:     in std_logic_vector(7 downto 0);
              wrxused: in std_logic;
              wrdelete:   in std_logic;
              wraddress:  in std_logic_vector(4 downto 0);
              wren:       in std_logic;
              inclock:    in std_logic;
              mstart:     in std_logic;
              mnext:      in std_logic;
              maddress:   out std_logic_vector(4 downto 0);
              mbits:      out std_logic_vector(31 downto 0);
              mfound: out std_logic;
              mcount:     out std_logic_vector(4 downto 0);
              rdbusy:     out std_logic;
              wrbusy:     out std_logic );
END COMPONENT;

        SIGNAL pattern : std_logic_vector(7 downto 0) := "00000000";
        SIGNAL wrx: std_logic_vector(7 downto 0) := "00000000";
        SIGNAL wrxused: std_logic := '0';
        SIGNAL wrdelete: std_logic := '0';
        SIGNAL wraddress: std_logic_vector(4 downto 0) := "00000";
        SIGNAL wren: std_logic := '0';
        SIGNAL inclock: std_logic := '1';
        SIGNAL mstart: std_logic := '0';
        SIGNAL mnext: std_logic := '0';
        SIGNAL maddress: std_logic_vector(4 downto 0);
        SIGNAL mbits: std_logic_vector(31 downto 0);
        SIGNAL mfound: std_logic;
        SIGNAL mcount: std_logic_vector(4 downto 0);
        SIGNAL rdbusy: std_logic;
```

```
        SIGNAL wrbusy: std_logic;

BEGIN

# Create a 10MHz-clock signal
        Clockin: PROCESS
        BEGIN
                  inclock <= NOT(inclock);
                  WAIT FOR 50 ns;
        END PROCESS;

# Create a 5MHz clock signal for the counter
        clockcount: PROCESS
        BEGIN
                  countclock <= NOT(countclock);
                  WAIT FOR 250 ns;
        END PROCESS;

# Generate the pattern inputs using an 8-bit counter
        PROCESS
                  variable cnt : integer range 0 to 256 := 88;
        BEGIN
                  IF cnt = 256 THEN
                  cnt := 0;
                  END IF;
                  cnt := cnt + 1;
                  wait until ((countclock'event) and (countclock = '0'));
                  pattern <= conv_std_logic_vector(cnt,8);
        END PROCESS;

        PROCESS
        BEGIN
                  WAIT FOR 300 ns;
                  mstart <= '1';
                  WAIT FOR 100 ns;
                  mstart <= '0';
                  WAIT FOR 200 ns;
                  mnext <= '1';
                  WAIT FOR 100 ns;
                  mnext <= '0';
        END PROCESS;

u1: cam32x8
PORT MAP(
        pattern => pattern, wrx => wrx, wrxused => wrxused,
        wrdelete => wrdelete, wraddress => wraddress, wren => wren,
        inclock => inclock, mstart => mstart, mnext => mnext,
        maddress => maddress, mfound => mfound, mcount => mcount,
        rdbusy => rdbusy, wrbusy => wrbusy, mbits => mbits);

END testbench;
```

## Verilog Instantiation

This example instantiates the altcam.v behavioral model for the `altcam` megafunction. This particular example instantiates the function, passes parameters and connects the ports of the `altcam` to input and output pins to demonstrate the functionality of the CAM. The `altcam` function can be directly instantiated as shown here along with other RTL code for simulation. This example uses files called cam.hex and cam_xu.hex to initialize the contents. Both files are shown at the end of this document. Cam32×8.v instantiates the `altcam` behavioral function. This can contain code other than just the `altcam` function call.

## Cam32x8.v

```
module cam32x8 (pattern, wraddress, wren, mstart, wrx, wrxused, wrdelete,
mnext, inclock, inclocken, maddress, mbits, mfound, mcount, rdbusy, wrbusy);
        input[7:0]  pattern;
        input[4:0]  wraddress;
        inputwren;
        inputmstart;
        input[7:0]  wrx;
        inputwrxused;
        inputwrdelete;
        inputmnext;
        inputinclock;
        inputinclocken;
        //inputinaclr;
        output[4:0]  maddress;
        output[31:0]  mbits;
        outputmfound;
        output[4:0]  mcount;
        outputrdbusy;
        outputwrbusy;

altcamU0 (.wrxused  (wrxused), .inclocken (inclocken), .wren (wren), .inclock
(inclock), .mstart (mstart), .wrx (wrx),
.pattern (pattern), .mnext (mnext), .wraddress (wraddress), .wrdelete
(wrdelete), .mcount (mcount),
.wrbusy (wrbusy), .maddress (maddress), .mfound (mfound), .rdbusy (rdbusy),
.mbits (mbits));
defparam
               U0.width =8,
               U0.widthad = 5,
               U0.numwords = 32,
               U0.match_mode = "MULTIPLE",
               U0.pattern_reg = "INCLOCK",
               U0.wrx_reg = "INCLOCK",
               U0.pattern_aclr = "Off",
               U0.wrx_aclr = "Off",
               U0.wrcontrol_aclr = "Off",
               U0.wraddress_aclr = "Off",
               U0.output_aclr = "Off",
               U0.lpm_file = "cam.hex",
               U0.lpm_filex = "cam_xu.hex";

endmodule
```

## Verilog TestBench

The Verilog testbench demonstrates the functionality of the altcam Megafunction. If the Modelsim simulator is used with the Verilog model, and hexadecimal initialization files are used, an extra step is required to simulate the models. A conversion must be done in order for the simulator to correctly convert the hexadecimal initialization files to a usable format when simulating Verilog code. This is done through the convert_hex2ver utility. These files need to be compiled before the simulation can be run. The following steps describe how this conversion is done.

- Obtain the convert_hex2ver.c and convert_hex2ver_lib.c files from the Altera web site at **http://www.altera.com**.
- Compile and link the source code into a library.

The following example code shows how to compile the source code within Microsoft Visual C/C++ (version 4.1 and above) on the Windows NT/98/95 operating systems.

```
cl -c -I<modelsim_dir>\include convert_hex2ver.c convert_hex2ver_lib.c
link -dll -export:init_usertfs convert_hex2ver.obj convert_hex2ver_lib.obj
<modelsim_dir>\win32\mtipli.lib
```

The following example code shows how to compile the source code with Sun C compiler on Solaris.

```
gcc -c -I<modelsim_dir>/include convert_hex2ver.c convert_hex2ver_lib.c
ld -G -B symbolic -o convert_hex2ver.so convert_hex2ver.o
convert_hex2ver_lib.o
```

- Modify modelsim.ini under the [vsim] section.

For Windows NT/98/95 operating systems, add the following line of code:

```
Veriuser = <DLL_dir>\convert_hex2ver.dll
```

For the Sun C compiler on Solaris, add the following line of code:

```
Veriuser = <SO_dir>/convert_hex2ver.so
```

The following cam_testbench.v code shows an example testbench that demonstrates the functionality of the altcam megafunction in multiple-match mode with “don’t care” bits.

```
`timescale 1 ps / 1 ps

module cam_testbench();

wire [7:0] pattern;
reg [7:0] wrx;
reg wrxused;
reg wrdelete;
reg [4:0] wraddress;
reg wren;
reg inclock;
reg inclocken;
reg mstart;
```

```
reg mnext;
wire [4:0] maddress;
wire [31:0] mbits;
wire mfound;
wire [4:0] mcount;
wire rdbusy;
wire wrbusy;

reg [6:0] cnt;
reg clock_count;

cam32x8 L0(.pattern(pattern), .wrx(wrx), .wrxused(wrxused),
.wrdelete(wrdelete), .wraddress(wraddress), .wren(wren), .inclock(inclock),
.mstart(mstart), .mnext(mnext), .maddress(maddress),
.mbits(mbits), .mfound(mfound), .mcount(mcount), .rdbusy(rdbusy),
.wrbusy(wrbusy), .inclocken(inclocken));

initial
begin

      assign inclocken = 1'b1;
      wrmask = 8'b00000000;
      wrmaskused = 1'b0;
      wrdelete = 1'b0;
      mstart = 1'b0;
      mnext = 1'b0;
end

initial cnt = 88;
assign pattern = cnt;

initial
begin
      inclock = 1'b0;
      forever #50000 inclock = ~inclock;
end

initial
begin
      clock_count = 1'b0;
      forever #250000 clock_count = ~clock_count;
end

always@(posedge clock_count)
begin
      if (cnt == 256)
            cnt = 0;
      cnt = cnt + 1;
end

initial
begin
      mstart = 1'b0;
```

```
      mnext = 1'b0;
      #300000 mstart = 1'b1;
      #100000 mstart = 1'b0;
      #200000 mnext = 1'b1;
      #100000 mnext = 1'b0;
end

endmodule
```

### Initialization file (cam.hex) for 32 × 8 bits CAM

Cam.hex contains the data to initialize the `altcam`. Note that address 0003 and 0007 contain the same data 59 (highlighted in blue). This indicates that multiple match exists in this example.

The initialization file (hex) for 32 × 8 bits CAM is as follows:

```
:0100000009f6
:0100010022dc
:01000200A25b
:0100030059a3
:0100040001fa
:01000500B04a
:0100060003f6
:01000700599f
:0100080035c2
:0100090009ed
:01000a0020d5
:01000b001Ada
:01000c0003f0
:01000d0028ca
:01000e0045ac
:01000f0024cc
:0100100009e6
:010011000De1
:0100120018d5
:010013007874
:0100140003e8
:0100150025c5
:010016001Bce
:0100170023c5
:01001800796e
:0100190033b3
:01001a001Fc6
:01001b0003e1
:01001c0011d2
:01001d0009d9
:01001e0006db
:01001f000Fd1
:00000001ff
```

## cam_ xu.hex

Cam_xu.hex contains “don’t care” information to initialize the `altcam`. Whenever the data in the file_ xu.hex file contains 0, it indicates that “don’t care” does not exist in that specific location. In this example, addresses 0003 and 0007 contain non-zero data, 02, indicating “don’t care” condition (highlighted in blue). These addresses will be matched when either data h59 or h5b is presented on the `pattern[]` input. This is due to the “don’t care” condition on the first bit (2) of addresses 0003 and 0007.

```
:0100000000ff
:0100010000fe
:0100020000fd
:0100030002fa
:0100040000fb
:0100050000fa
:0100060000f9
:0100070002f6
:0100080000f7
:0100090000f6
:01000a0000f5
:01000b0000f4
:01000c0000f3
:01000d0000f2
:01000e0000f1
:01000f0000f0
:0100100000ef
:0100110000ee
:0100120000ed
:0100130001eb
:0100140000eb
:0100150000ea
:0100160000e9
:0100170000e8
:0100180001e6
:0100190000e6
:01001a0000e5
:01001b0000e4
:01001c0000e3
:01001d0000e2
:01001e0000e1
:01001f0000e0
:00000001ff
END
```

## Conclusion

The CAM feature on APEX 20KE devices provides a powerful tool for accelerating search applications in communications designs. By using the techniques described in this white paper, you can easily integrate high-speed CAM into your design.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com