Using Data2Mem in a Non-EDK design

-Deepesh Man Shakya / Stephen MacMahon

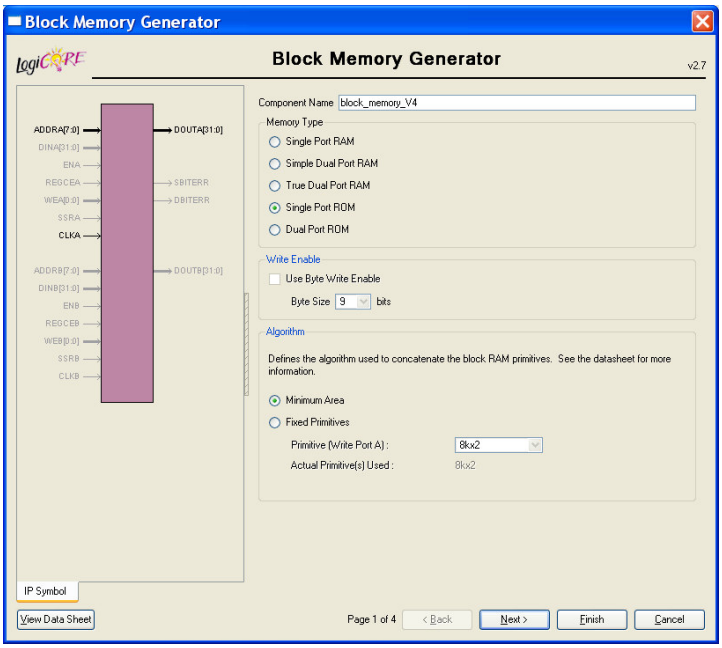
Introduction

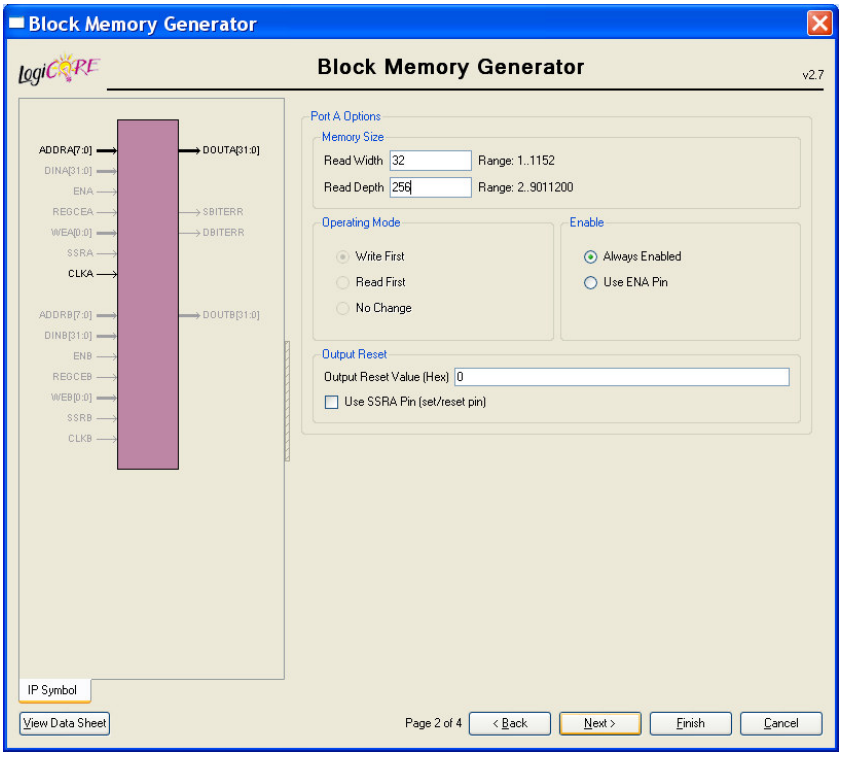
Data2mem is used to initialize block ram memory without needing to re-implement the design. In an EDK design, generation of BMM (Block Memory Map) file is handled within EDK but in a non-EDK design it is necessary to create a BMM file manually. To initialize memory in a design, data2mem is used. The input files required to generate new bit file with new bram data are: original bit file, bmm file and the mem file. Mem file contains the BRAM data.

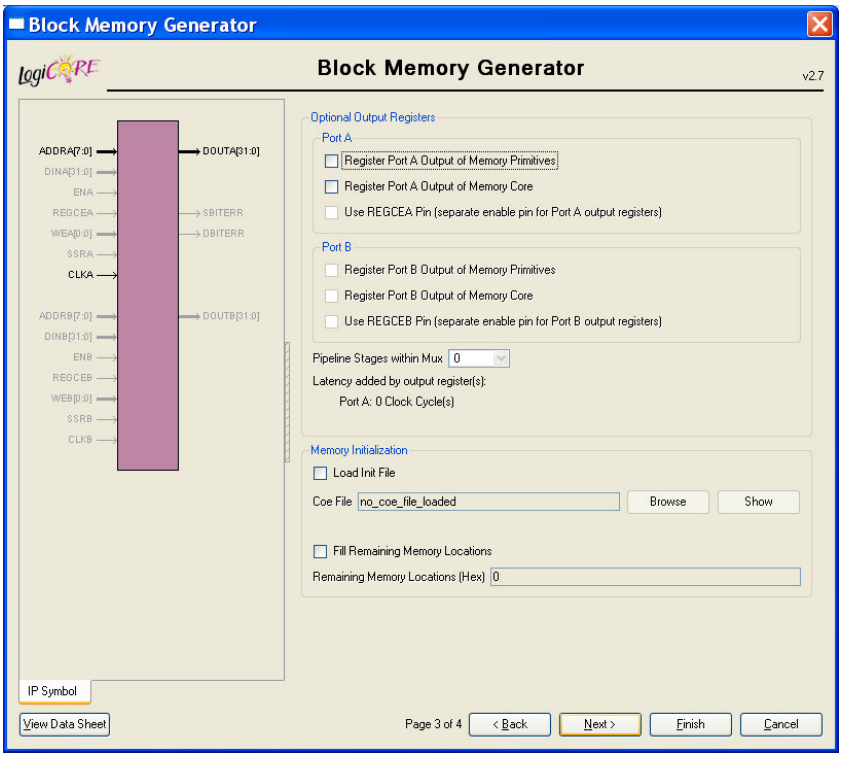
In this document, we show how data2mem can be used to initialize memory in a non-EDK design. We start with coregen to generate a Block Memory, instantiate this in a top level vhdl file in ISE, create a BMM file and invoke data2mem file to initialize the memory in the design. Later, we will dump the bit file (with bram initialized) to verify whether the content of the mem file has actually been initialized in the memory location of the bit file.

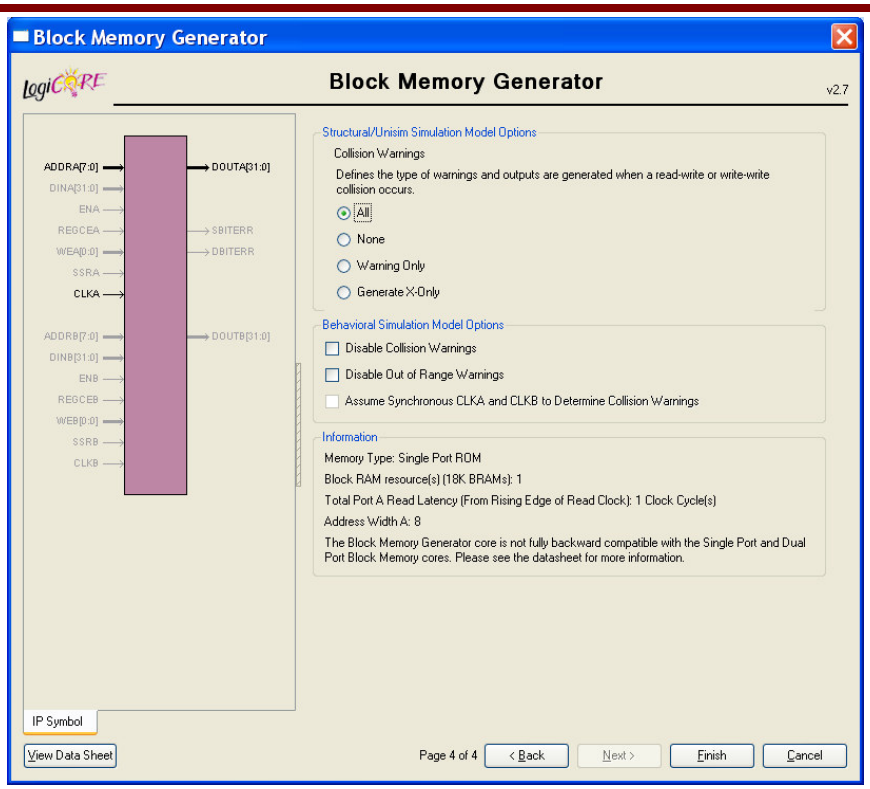
In the remainder of this document, we show the step-by-step procedure to demonstrate the flow described above.

1. Generate Block Memory of data width 32 bits as shown in the screen shots from coregen below.




Block Memory Generator
Block Memory Generator
v2.7
ADDRA[7:0]
DINA[31:0]
ENA
REGCEA
WEA[0:0]
SSRA
CLKA
ADDRB[7:0]
DINB[31:0]
ENB
REGCEB
WEB[0:0]
SSRB
CLKB
DOUTA[31:0]
SBITERR
DBITERR
DOUTB[31:0]
Port A Options
Memory Size
Read Width 32 Range: 1..1152
Read Depth 256 Range: 2..9011200
Operating Mode
Write First
Read First
No Change
Enable
Always Enabled
Use ENA Pin
Output Reset
Output Reset Value (Hex) 0
Use SSRA Pin (set/reset pin)
IP Symbol
View Data Sheet
Page 2 of 4
< Back
Next >
Finish
Cancel


Block Memory Generator
Block Memory Generator
v2.7
ADDRA[7:0]
DINA[31:0]
ENA
REGCEA
WEA[0:0]
SSRA
CLKA
ADDRB[7:0]
DINB[31:0]
ENB
REGCEB
WEB[0:0]
SSRB
CLKB
DOUTA[31:0]
SBITERR
DBITERR
DOUTB[31:0]
Optional Output Registers
Port A
Register Port A Output of Memory Primitives
Register Port A Output of Memory Core
Use REGCEA Pin (separate enable pin for Port A output registers)
Port B
Register Port B Output of Memory Primitives
Register Port B Output of Memory Core
Use REGCEB Pin (separate enable pin for Port B output registers)
Pipeline Stages within Mux 0
Latency added by output register(s):
Port A: 0 Clock Cycle(s)
Memory Initialization
Load Init File
Coe File no_coe_file_loaded
Browse
Show
Fill Remaining Memory Locations
Remaining Memory Locations (Hex) 0
IP Symbol
View Data Sheet
Page 3 of 4
< Back
Next >
Finish
Cancel



2. Instantiate the generated block memory in top level vhdl file as shown below:

```
entity top is
    Port ( clka : in  STD_LOGIC;
           addra : in  STD_LOGIC_VECTOR (7 downto 0);
           douta : out  STD_LOGIC_VECTOR (31 downto 0));
end top;

architecture Behavioral of top is

   COMPONENT block_memory_v4
   PORT(
      clka: IN std_logic;
      addra: IN std_logic_VECTOR(7 downto 0);
      douta: OUT std_logic_VECTOR(31 downto 0));
   END COMPONENT;

begin

   v4_block_memory: block_memory_v4 PORT MAP(
      clka => clka,
      addra => addra,
      douta => douta
   );

end Behavioral;
```

3. Next step is to create a bmm file. The bmm file created for the test design for this document is as shown below:

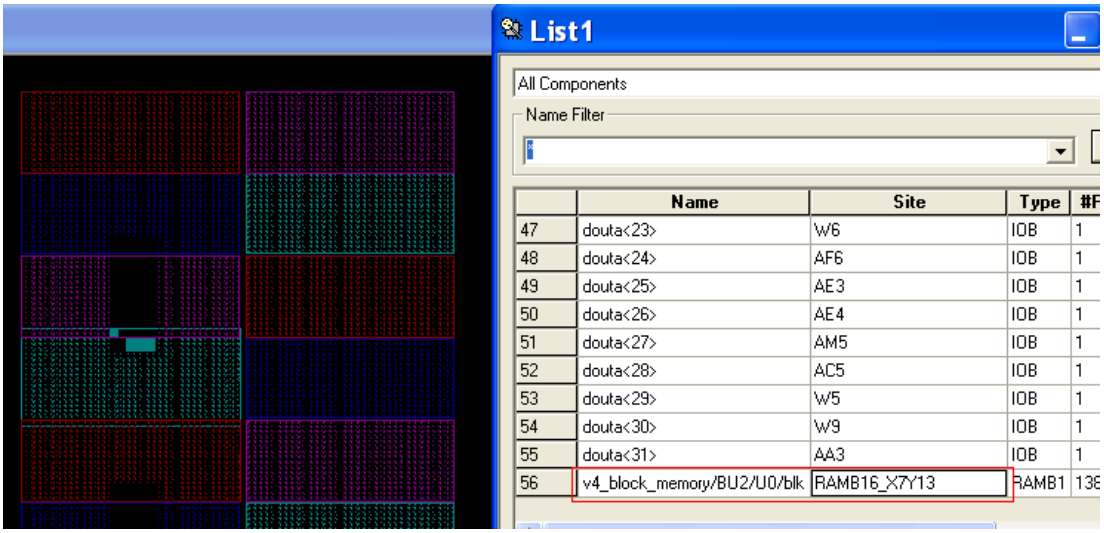
```
ADDRESS_SPACE v4_minor RAMB16 [0x00000000:0x000007FF]
            BUS_BLOCK

      v4_block_memory/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram
.r/v4_noinit.ram/SP.WIDE_PRIM.SP [31:0] PLACED = X7Y13;
            END_BUS_BLOCK;
END_ADDRESS_SPACE;
```

There are two main questions in writing bmm file besides the syntax as shown above:

a. How to find the bram location?

b. How to calculate address range?

For 'a', open FPGA editor after PAR and locate BRAM as shown below:



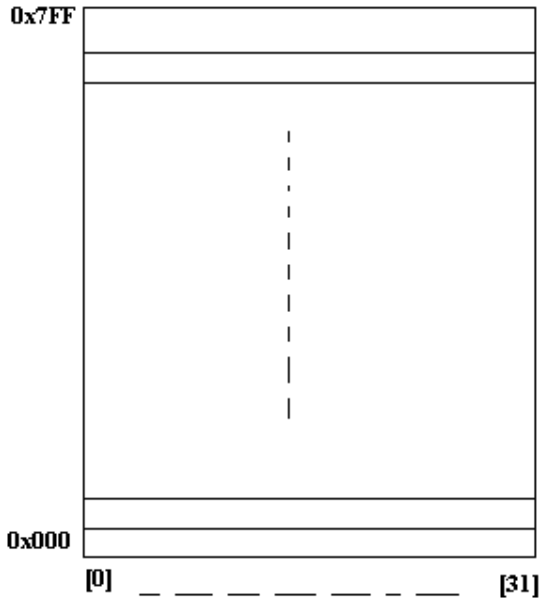
Highlight the BRAM in the red box above and copy the memory location as shown below:

```
site "RAMB16_X7Y13",  type = RAMB16  (RPM grid X253Y104)
site "RAMB16_X7Y13",  type = RAMB16  (RPM grid X253Y104)
site "RAMB16_X7Y13",  type = RAMB16  (RPM grid X253Y104)
<RAMB16>; bel <v4_block_memory/BU2/U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v4_noinit.ram/SP.WIDE_PRIM.SP>.
```

The question in ‘b’ was how to calculate the address range:

```
ADDRESS_SPACE v4_minor RAMB16 [0x00000000:0x000007FF]
```

There is one memory block used in this example design. The size of the memory block is 16Kbit. Since the data width we selected in block memory coregen GUI is 32, the memory will be arranged as shown below in the figure:



There are 512 lines in the above memory with 4 bytes (32 bits) in each line. If we calculate the total size it will be: 512 x 32 = 16384bit = 16Kbit.

4. The content of the mem file is as shown below. This initializes first 16bytes of the memory.

```
@0000
00000000
@4
DEADBEEF
@8
DEADABBA
@C
DEADFEED
```

For more information on writing mem file please visit the answer record below:
http://www.xilinx.com/support/answers/14384.htm

5. Now, we have all three required files to run data2mem. Following data2mem command line is used to initialize the block memory with the mem file described in ‘4’ above.

```
data2mem -bm top_bd.bmm -bt top.bit -bd test.mem tag v4_minor -o b
top_mem.bit
```

6. top_mem.bit file will have its bram initialized with the mem file. To verify whether this has been done or not, the bit file can be dumped using following command:

data2mem -bm top_bd.bmm -bt top_mem.bit -d > dump_mem.txt

Open dump_mem.txt and search for 'blk_mem_generator/valid.cstr/ramloop[0].ram This is a part of the memory path in step '3' above.

```
BRAM data, Column 07, Row 13. Design instance "v4_block_memory/BU2/
       U0/blk_mem_generator/valid.cstr/ramloop[0].ram.r/v4_noinit.ram/SP.WIDE_PRIM.SP".
00000000:   00 00 00 00 DE AD BE EF DE AD AB BA DE AD FE ED 00 00 00 00 00 00 00 00
00000020:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000080:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000A0:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000C0:   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Using RAMB18 in Spartan device

In a Spartan device you can instantiate a *RAMB16_S18*. The RAMB16_S18 primitive is a 16K-bit Data + 2K- bit Parity Memory, Single-Port Synchronous Block RAM with 18-bit Port.

The attraction of using this is that you can use the extra 2K as storage. However, issues arrive when Data2mem tries to map the memory. This causes Data2mem to crash due to data corruption in the tools.

The error occurs when the Data2mem tries to map the data into the final Block RAM.
This is shown in the example below:
If there are 16 blocks with 18 bits per line in the .BMM file. Each block is 1024 Bytes in size. So the range is $0x0000:1024_{10} = 0x00000000:000003FF_{16}$

If the .MEM file contains 20 bits of data per line. This Data is broken up as follows:
Bits (0 - 1) are 2 discarded bits.
Bits (2 - 3) are the 2 Parity bits.
Bits (4 - 19) are the 16 Data bits.

So if the .MEM file contained the data:
0x00000000
AAAAA
BBBBB
….
….

This is broken down in to:
0x00000000
1010 1010 1010 1010 1010
1011 1011 1011 1011 1011
…
…

Where:
1010 1010 1010 1010 1010
1011 1011 1011 1011 1011
Bits (0 - 1) are 2 discarded bits.
Bits (2 - 3) are the 2 Parity bits.

So if you want to fill 1 block of RAM you will need 1024 Lines in the .MEM file. In this case there are 16 blocks of RAM so (16 x 1024 = 16384) 16384 Lines are needed to fill the entire memory.

However, Data2mem crashes when the last block (block 16) is being filled. This is due to data corruption. The workaround and example are seen on next page.

To workaround this issue the Data2mem process will have to be broken up into two stages.

Stage 1: Fill the first 15 blocks with your .MEM file. So if you want to fill 15 blocks you will need 1024 x 15 = 15360 lines in your .mem file. The Data2mem command to use here is

data2mem -bm 16_Block.bmm -bd 15360_Lines.mem -bt Original.bit -o b Intermediate.bit

Here it can be see that the 16 blocks are used in the .BMM file but only 15 blocks are being populated as we are only using 15360 Lines in the .MEM file (15360 / 1025 = 15).
This creates an Intermediate.bit file, as this will be passed into the next command line.

Stage two: Fill the remaining block (block 16). Since we only need to fill one block, only 1024 lines are needed in the .MEM file. This is seen in the command below:

data2mem -bm 1_Block.bmm -bd 1024_Lines.mem -bt Intermediate.bit -o b Finished.bit

Here it can be see that only one block RAM was used in the .BMM to map the final 1024 lines into RAM. This created a Finished.bit file. To verify that the memory has been mapped correctly we can dump the Finished.bit file. This is shown below:

data2mem -bm 16_Block.bmm -bt Finished.bit -d > Dump.txt

This Dump.txt file can be used to verify the correct data has been placed in the correct block RAM decided by the .BMM file. To search the Dump.txt file easily, open the .BMM file and use each instantiation to search for that particular block RAM in the Dump.txt file.

Note: A Change Request has been filed against this (CR 488463). Hopefully, this will be fixed in EDK 11.1i

Note: To download the files that were used in this demonstration go to the link below:
http://xirweb/~stephenm/Projects/Data2mem_Tests.zip