UpdateMEM User Guide

UG1580 (v2022.1) May 23, 2022

Xilinx is creating an environment where employees, customers, and partners feel welcome and included. To that end, we're removing non-inclusive language from our products and related collateral. We've launched an internal initiative to remove language that could exclude people or reinforce historical biases, including terms embedded in our software and IPs. You may still find examples of non-inclusive language in our older products as we work to make these changes and align with evolving industry standards. Follow this link for more information.

AMD
XILINX

AMD
XILINX

Table of Contents

AMD XILINX

Chapter 1

Introduction

A single device, with one or more embedded processors as well as programmable logic, needs a single boot image, which must contain the merged CPU software and FPGA bitstream images. The UpdateMEM utility (updatemem) is a data translation tool to map contiguous blocks of data across multiple block RAMs that constitute a contiguous logical address space.

With the combination of Zynq®-7000 SoC devices or MicroBlaze™ embedded processor, on the UltraScale™, UltraScale+™, or 7 series devices, UpdateMEM merges the CPU software image of an executable and linkable format (ELF) file into the FPGA bitstream created by the Vivado Design Suite and the write_bitstream command, by mapping the ELF data onto the memory map information (MMI) for the block RAMs in the design. For Versal® devices, the similar command is write_device_image instead of write_bitstream.

Navigating Content by Design Process

Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. All Versal® ACAP design process Design Hubs and the Design Flow Assistant materials can be found on the Xilinx.com website. This document covers the following design processes:

- **Embedded Software Development:** Creating the software platform from the hardware platform and developing the application code using the embedded CPU. Also covers XRT and Graph APIs.

AMD XILINX

Chapter 2

Using UpdateMEM

For embedded processor based designs, the UpdateMEM (`updatemem`) command merges CPU software images into bitstream files, to initialize the block RAM memory within the target Xilinx device. The UpdateMEM command can also take an ELF file or CPU Software Image as an input and write out MEM files for simulation purposes. The UpdateMEM command takes the following inputs:

- A bitstream (BIT) file, or RDCO file for Versal devices, which is initially generated by the Vivado® Design Suite implementation tools. You can create a bitstream or RCDO file from an implemented design using the `write_bitstream` or `write_device_image` Tcl commands respectively. These files are binary data files that contain the bit images of the design, to be downloaded to a Xilinx devices. The UpdateMEM command reads a BIT or RCDO file as an input, and writes a BIT or RCDO file as an output
- The memory-map information (MMI) file is a text file that describes how individual block RAMs on the Xilinx device are grouped together to form a contiguous address space called an address block.

  The Vivado Design Suite writes the MMI file automatically and places that file into the `<project>.runs/impl_1` folder when generating the bitstream, or you can manually write that information using the `write_mem_info` command. The UpdateMEM command uses the MMI file to identify the physical Block RAM resources that map to a specific address range. For more information on the MMI file, see Block RAM Memory Map Info File.
- The Vivado Design Suite writes the SMI file (memory-map information file for simulation) automatically and places that file into the `<project>.sim/sim_x/behav` folder when simulation is run on the design.
- An executable and linkable format (ELF) file, which is a product of the Vitis™ integrated design environment, is a binary data file that contains an executable program image ready for running on an embedded processor. The ELF file contains the data to be mapped by UpdateMEM into the address ranges of the block RAMs.
- Optionally, a memory (MEM) file is a manually created text file that describes contiguous blocks of data to initialize or populate a specified address space. The UpdateMEM command can use the MEM file in place of an ELF file. See Memory Files for more information.
- An instance ID of the embedded processor in the design, to associate the ELF or MEM file with the processor.

The UpdateMEM command populates contiguous blocks of data defined in ELF or MEM files, across multiple block RAMs of a Xilinx device mapped by the MMI file. The UpdateMEM command merges the memory information into a bitstream file for configuring and programming the target Xilinx device.

The UpdateMEM command also lets you merge multiple data files for multiple processors in designs that have more than one embedded processor. In this case, the `-data` and `-proc` options must be specified in pairs, with the first `-data` file providing the software image or memory content for the first `-proc` specified. The second `-data` applies to the second `-proc`, and so on.

This command returns the name of the bitstream file created from the inputs, or returns an error if it fails.

Examples

The following example reads the specified MEM info file, ELF file, and bitstream file, and generates the merged bitstream file:

```
updatemem -meminfo top.mmi -data hello_world.elf -bit top.bit \
-proc design_1_i/microblaze_1 -out top_meminfo.bit
```

The following example shows the use of UpdateMEM in a block design that has two embedded microblaze processors, one with an associated ELF file, and the other using a MEM file. Notice this requires two passes of the `updatemem` command, with the output bitstream of the first acting as the input bitstream of the second:

```
updatemem -bit top.bit -meminfo top.mmi -data top1.elf \
-proc system_i/microblaze_1 -out first_out.bit
updatemem -bit first_out.bit -meminfo top.mmi -data top2.mem \
-proc system_i/microblaze_2 -out top_out.bit
```

To convert an ELF file into a MEM file for simulation flows, use the following command:

```
updatemem -data top1.elf -meminfo top1.smi -proc design_1_i/microblaze_0
```

Arguments for updatemem

- `-meminfo <arg>`: (Required) Name of the memory mapping information (MMI) file for the implemented design or memory mapping information for simulation (SMI) file. This file can be generated using the `write_mem_info` Tcl command.

- `-data <arg>`: (Required) Name of the Executable and Linkable Format (ELF) file, or a MEM file to map into BRAM addresses.
- `-writememfile`: Output.`mem` file. Translates the ELF file and writes the information to the specified `.mem` file, which can be used in simulation flows. This option is applicable only to processor based designs. This argument is still supported but not recommended to be used.
- `-bit <arg>`: (Required) Name of the bit input bitstream (BIT) file. If the file extension is missing, an extension of `.bit` is assumed. For Versal® devices, the extension of .rcdo should be used instead of .bit.

  Note: The UpdateMEM command can only be used with unencrypted bitstream files.
- `-proc <arg>`: (Required) Instance path of the embedded processor.

  TIP: *You can specify multiple processors for the UpdateMEM command in cases where a design has multiple embedded processors. In this case the `-data` and `-proc` options must be specified in pairs, with the first `-data` argument applying to the first `-proc` argument. However, the UpdateMEM command can take either an ELF file or a MEM file in a single run, but cannot use both `-data` formats at the same time even when specifying multiple processors.*
- `-out <arg>`: (Required) Specify the name of output file, without suffix. The file has a suffix of `.bit` applied automatically. For Versal devices, the extension of .rcdo should be used instead of .bit.
- `-force`: (Optional) Overwrite the specified output file if it already exists.

For Versal devices, there is a three stage process for using updatemem to generate a new boot file.

1. Run updatemem on original `.rcdo` file to generate the new `.rcdo` file.

   ```
   updatemem -meminfo ./design_1_wrapper.mmi -data ./elf_file.elf -proc
   design_1_i/microblaze_0 -bit design_1_wrapper.rcdo -out
   design_1_wrapper_new.rcdo
   ```
2. Edit the `.bif` file to point to the new generated `.rcdo` file. Inside the `.bif` file there will be a field called "file" that is set to the original `.rcdo` file. Edit this field to set the newly generated `.rcdo` file.
3. Run bootgen to regenerate a new `.pdi` file to run on HW.

   ```
   bootgen -arch versal -image design_1_wrapper.bif -w -o boot.pdi
   ```

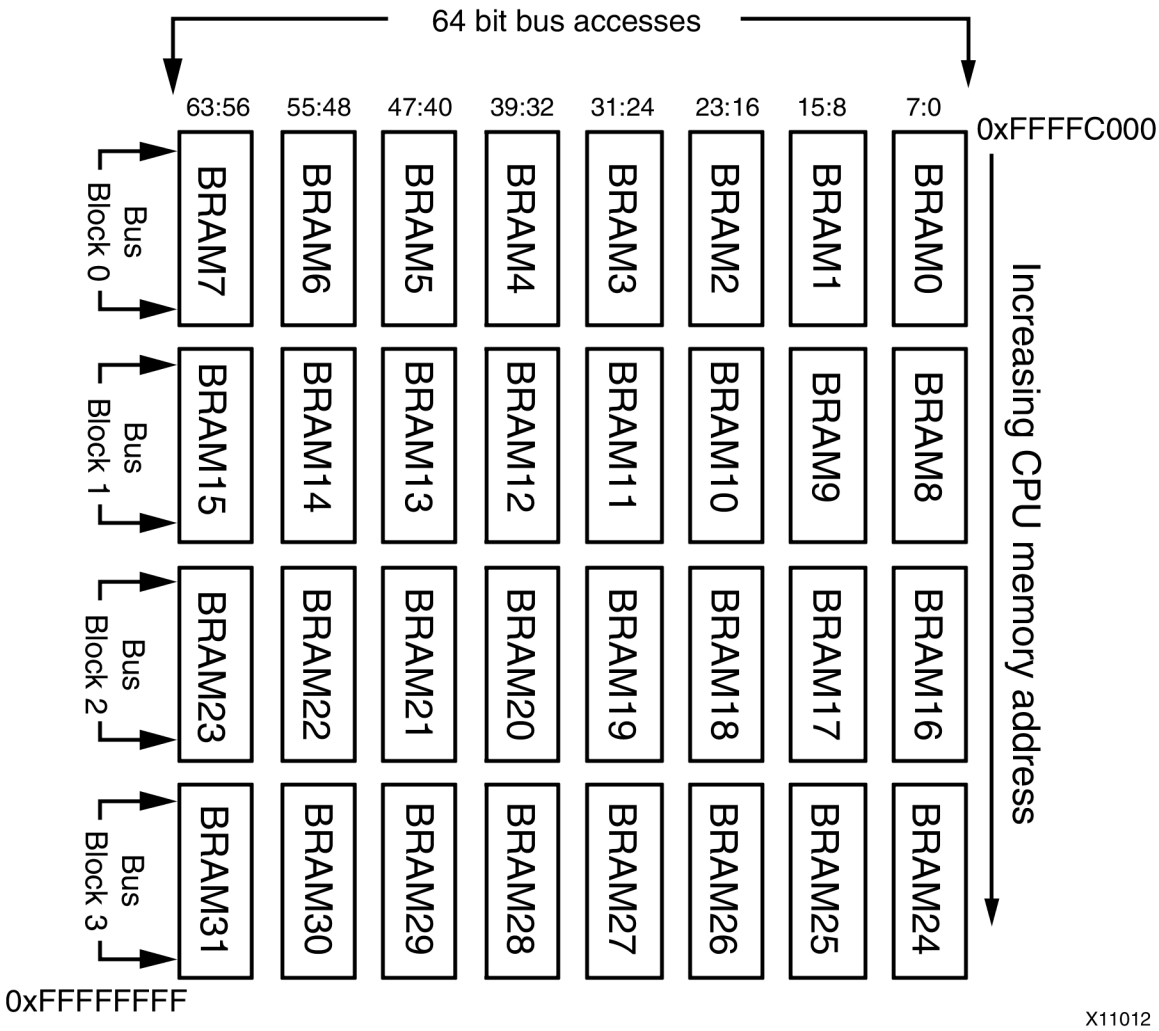
Note: A PDI file is the Versal equivalent to the `.bit` file for pre Versal devices.

Block RAM Memory Map Info File

The following are design considerations for block RAM-implemented address spaces, and the definition of memory map info (MMI) files:

- The block RAMs come in fixed-size widths and depths, where CPU address spaces might need to be much larger in width and depth than a single block RAM. Consequently, multiple block RAMs must be logically grouped together to form a single CPU address space as seen in the following figure.
- A single CPU bus access is often multiple bytes wide of data, for example, 32 or 64 bits (4 or 8 bytes) at a time.
- CPU bus accesses of multiple data bytes might also access multiple block RAMs to obtain that data. Therefore, byte-linear CPU data must be interleaved by the bit width of each block RAM and by the number of block RAMs in a single bus access. However, the relationship of CPU addresses to block RAM locations must be regular and easily calculable.
- CPU data must be located in a block RAM-constructed memory space relative to the CPU linear addressing scheme, and not to the logical grouping of multiple block RAMs.
- Address space must be contiguous, and in whole multiples of the CPU bus width. Bus bit lane interleaving is allowed only in the sizes supported by the Virtex device block RAM port sizes.
- Addressing must account for the differences in instruction and data memory space. Because instruction space is not writable, there are no address width restrictions. However, data space is writable and usually requires the ability to write individual bytes. For this reason, each bus bit lane must be addressable.
- The size of the memory map and the location of the individual block RAMs affect the access time. Evaluate the access time after implementation to verify that it meets the design specifications.

Figure 1: **Block RAM Address Space**



The address space in the figure above consists of four bus blocks: Bus Block 0 through 3.

- CPU bus accesses are eight block RAMs (64 bits) wide, with each column of block RAMs occupying an 8-bit wide slice of a CPU bus access called a Bit Lane.
- Each row of eight block RAMs in a bus access are grouped together in a Bus Block. Hence, each Bus Block is 64 bits wide and 4096 bytes in size.
- The entire collection of block RAMs is grouped together into a contiguous address space called an Address Block.

The upper right corner address is `0xFFFFC000`, and the lower left corner address is `0xFFFFFFFF`. Because a bus access obtains eight data bytes across eight block RAMs, byte-linear CPU data must be interleaved by 8 bytes in the block RAMs.

In this example using a 64-bit data word indexed by bytes from left to right as [0:7], [8:15]:

- Byte 0 goes into the first byte location of bit lane block RAM7, byte 1 goes into the first byte location of Bit Lane block RAM6; and so forth, to byte 7.
- CPU data byte 8 goes into the second byte location of Bit Lane block RAM7, byte 9 goes into the second byte location of Bit Lane block RAM6 and so forth, repeating until CPU data byte 15.
- This interleave pattern repeats until every block RAM in the first bus block is filled.
- This process repeats for each successive bus block until the entire memory space is filled, or the input data is exhausted.

As described in MMI File Syntax, the order in which bit lanes and bus blocks are defined controls the filling order. For the sake of this example, assume that bit lanes are defined from left to right, and bus blocks are defined from top to bottom.

This process is called *bit lane mapping*, because these formulas are not restricted to byte-wide data. This is similar, but not identical, to the process embedded software programmers use when programmed CPU code is placed into the banks of fixed-size EPROM devices.

The important distinctions to note between the two processes are, as follows:

- Embedded system developers generally use a custom software tool for byte-lane mapping for a fixed number and organization of byte-wide storage devices. Because the number and organization of the devices cannot change, these tools assume a specific device arrangement. Consequently, little or no configuration options are provided.

  By contrast, the number and organization of FPGA block RAMs are completely configurable (within FPGA limits). Any tool for byte-lane mapping for block RAMs must support a large set of device arrangements.
- Existing byte-lane mapping tools assume an ascending order of the physical addressing of byte-wide devices because that is how board-level hardware is built. By contrast, FPGA block RAMs have no fixed usage constraints and can be grouped together with block RAMs anywhere within the FPGA fabric. Although this example displays block RAMs in ascending order, block RAMs can be configured in any order.

Memory Map Information File (MMI) Features

A memory map information (MMI) file is an XML file designed for computer parsing. It is similar to high-level computer programming languages in using the following features:

- Block structures by XML keyword tags or directives. MMI maintains similar structures in groups or blocks of data. MMI creates blocks to delineate address space, bus access groupings, and comments.
- Symbolic name usage: MMI uses names and keywords to refer to groups or entities (improving readability), and uses names to refer to address space groupings and Block RAMs.

MMI observes the following conventions:

- Keywords are case-sensitive.
- Indenting is for clarity only.
- White space is ignored except where it delineates items or keywords.
- Line endings are ignored. You can have as many items as you want on a single line.
- Numbers can be entered as decimal or hexadecimal. Hexadecimal numbers use the `0xXXX` notation form.

> **CAUTION!** *The MMI file does not get generated if a design does not contain a processor or XPM Memories.*

MMI File Syntax

The memory map info (MMI) file is an XML file that syntactically describes how individual block RAMs make up a contiguous logical data space. You can create an MMI file from an open implemented design in the Vivado Design Suite using the `write_mem_info` Tcl command. The implemented design provides the needed placement information of the block RAM resources.

UpdateMEM uses the MMI file as input to direct the translation of data into the proper initialization form. The Example MMI file below shows the XML-based syntax used to describe the organization of block RAM usage.

```
<?xml version="1.0" encoding="UTF-8"?>
<MemInfo Version="1" Minor="0">
  <Processor Endianness="Little" InstPath="design_1_i/microblaze_0">
    <AddressSpace
Name="design_1_i_microblaze_0.design_1_i_microblaze_0_local_memory_dlmb_bram
_if_cntlr" Begin="0" End="8191">
      <BusBlock>
        <BitLane MemType="RAMB32" Placement="X2Y17">
          <DataWidth MSB="15" LSB="0"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
        <BitLane MemType="RAMB32" Placement="X3Y17">
          <DataWidth MSB="31" LSB="16"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
      </BusBlock>
    </AddressSpace>
 </Processor>
  <Processor Endianness="Little" InstPath="design_1_i/microblaze_1">
    <AddressSpace
Name="design_1_i_microblaze_1.design_1_i_microblaze_1_local_memory_dlmb_bram
_if_cntlr" Begin="0" End="8191">
      <BusBlock>
        <BitLane MemType="RAMB32" Placement="X4Y13">
          <DataWidth MSB="15" LSB="0"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
        <BitLane MemType="RAMB32" Placement="X4Y14">
          <DataWidth MSB="31" LSB="16"/>
```

```
          <AddressRange Begin=”0” End=”2047”/>
          <Parity ON=”false” NumBits=”0”/>
        </BitLane>
      </BusBlock>
    </AddressSpace>
 </Processor>
  <Processor Endianness="Little" InstPath="design_1_i/processing_system7_0">
    <AddressSpace
Name="design_1_i_processing_system7_0.design_1_i_axi_bram_ctrl_0"
Begin="1073741824" End="1073750015">
      <BusBlock>
        <BitLane MemType="RAMB32" Placement="X2Y18">
          <DataWidth MSB="15" LSB="0"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
        <BitLane MemType="RAMB32" Placement="X2Y19">
          <DataWidth MSB="31" LSB="16"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
      </BusBlock>
    </AddressSpace>
  </Processor>
  <Config>
    <Option Name="Part" Val="xc7z020clg484-1"/>
  </Config>
</MemInfo>
```

Address Map Definitions (Multiple Processor Support)

UpdateMEM supports multiple processors using the following XML tags:

```
<Processor Endianness="Little” InstPath=”design_1_i/processing_system7_0”>
</Processor>
```

IMPORTANT! *Although Processor Endianness is defined in the MMI file, it is not supported by UpdateMEM.*

Address Space Definitions

The outermost definition of an address space comprises the following components:

```
<AddressSpace
Name="design_1_i_processing_system7_0.design_1_i_axi_bram_ctrl_0”
Begin=”1073741824” End=”1073750015”>
</AddressSpace>
```

The `ADDRESS_SPACE` and `/ADDRESS_SPACE` tags define a single contiguous address space. The mandatory `Name=` following the `ADDRESS_SPACE` tag provides a symbolic name for the entire address space. Referring to the address space name is the same as referring to the entire contents of the address space.

An MMI file can contain multiple `ADDRESS_SPACE` definitions, even for the same address space, as long as each `ADDRESS_SPACE` name is unique.

Next is the beginning and ending address values that the Address Space occupies by using the `Begin=` and `End=` pair.

BusBlock Definitions (Bus Accesses)

Inside an `ADDRESS_SPACE` definition are a variable number of sub-block definitions called *BusBlocks*, as shown in the following example:

```
<BusBlock>
</BusBlock>
```

Each Bus Block contains block RAM Bit Lane definitions that are accessed by a parallel CPU bus access.

The order in which the bus blocks are specified defines which part of the address space a Bus Block occupies. The lowest addressed Bus Block is defined first, and the highest addressed Bus Block is defined last. The top-to-bottom order in which Bus Blocks are defined also controls the order in which UpdateMEM fills those Bus Blocks with data.

Bit-Lane Definitions (Memory Device Usage)

A bit-lane definition determines which bits in a CPU bus access are assigned to particular block RAMs. Each definition takes the form of `MemType` with Placement data, followed by the bit numbers and `AddressRange` the bit lane occupies. The syntax is, as follows:

```
<BitLane MemType="RAMB32" Placement="X2Y19">
          <DataWidth MSB="31" LSB="16"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
```

IMPORTANT! *Although bit-lane parity is defined in the MMI file, it is not supported by UpdateMEM.*

Typically, the bit numbers are given in the following order:

```
<DataWidth MSB=bit_num LSB=bit_num>
```

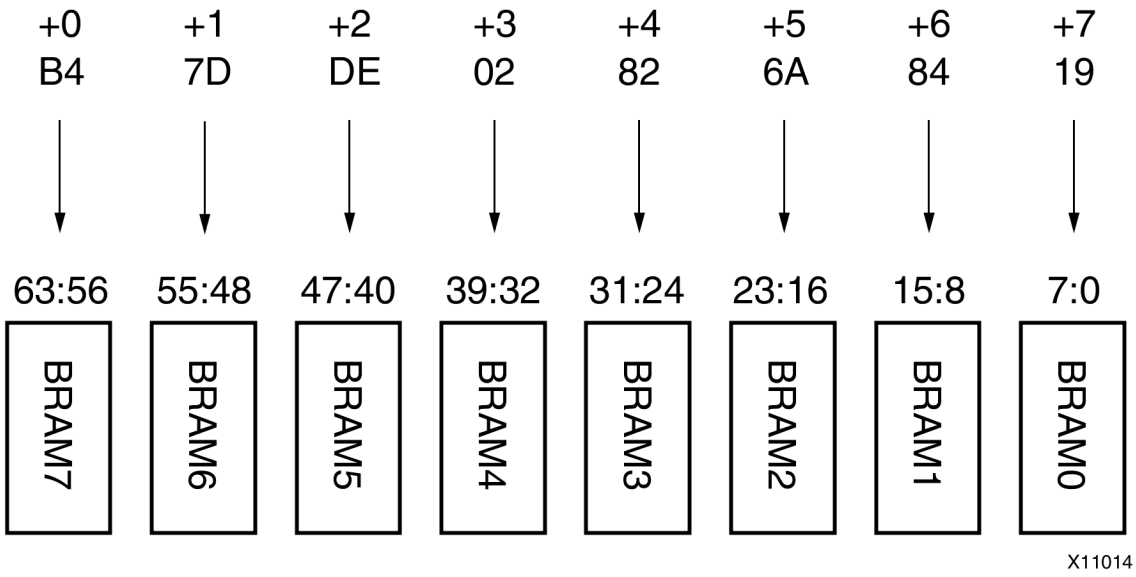
If the order is reversed to have the least significant bit (LSB) first and the most significant bit (MSB) second, UpdateMEM bit-reverses the bit-lane value before placing it into the block RAM.

As with BusBlocks, the order in which bit-lanes are defined is important. But in the case of bit-lanes, the order infers which part of BusBlock CPU access a bit-lane occupies. The first bit-lane defined is inferred to be the most significant bit-lane value, and the last defined is the least significant bit-lane value. In the following figure, the most significant bit-lane is `BRAM7`, and the least significant bit-lane is `BRAM0`. As seen in example Figure 1: Block RAM Address Space layout, this corresponds with the order in which the Bit Lanes are defined.

When UpdateMEM inputs data, it takes data from data input files in Bit Lane sized chunks, from the most right value first to the left most. For example, if the first 64 bits of input data are `0xB47DDE02826A8419` then the value 0xB4 is the first value to be set into a block RAM.

Given the Bit Lane order, BRAM7 is set to 0xB4, BRAM6 to 0x7D, and so on until BRAM0 is set to 0x19. This process repeats for each successive Bus Block access block RAM set until the memory space is filled or until the input data is exhausted. The following figure expands the first Bus Block to illustrate this process.

Figure 2: **Bit Lane Fill Order**



The Bit Lane definitions must match the hardware configuration. If the MMI is defined differently from the way the hardware actually works, the data retrieved from the memory components will be incorrect.

Bit Lane definitions also have some optional syntax, depending on what device type keyword is used in the Address Block definition.

When specifying block RAM cells, the physical row and column location within the FPGA device can be indicated. Following are examples of the physical row and column location:

```
Placement="X3Y5"
```

Use the `Placement=` keyword to assign the corresponding block RAM to a specific resource location in the FPGA device. In this case the block RAM is placed at column 3 and row 5 in the FPGA device.

In addition to using correct syntax for bit-lane and BusBlock definitions, you must take into account the following limitations:

- While the examples in this document use only byte-wide data widths for clarity, the same principles apply to any data width for which a block RAM is configured.
- There cannot be any gaps or overlaps in bit-lane numbering. All bit-lanes in an Address Block must be the same number of bits wide.
- The bit-lane widths are valid for the memory device specified by the device type keyword.
- The amount of byte storage occupied by the Bit Lane block RAMs in a BusBlock must equal the range of addresses inferred by the start and end addresses for a BusBlock.
  - All BusBlocks must be the same number of bytes in size.
  - A block RAM instance name can be specified only once.
  - A BusBlock must contain one or more valid bit-lane definitions.
  - An address Block must contain one or more valid BusBlock definitions.

  UpdateMEM checks for all these conditions and transmits an error message if it detects a violation.

Memory Files

A Memory (MEM) file is a manually edited text file that describes contiguous blocks of data. that can be used in place of the ELF file. The format of MEM files is an industry standard, consisting of two basic elements:

- Hexadecimal address specifier: An address specifier is indicated by an @ character followed by the hexadecimal address value. There are no spaces between the @ character and the first hexadecimal character.
- Hexadecimal data values: Hexadecimal data values follow the hexadecimal address value, separated by spaces, tabs, or carriage-return characters.

Because the MEM file is in hexadecimal format, each character represents four bits, or a nibble, in the memory.

Hexadecimal data values can consist of as many hexadecimal characters as desired. However, when a value has an odd number of hexadecimal characters, the first hexadecimal character is assumed to be a zero. For example, hexadecimal values A, C74, and 84F21 are interpreted as the values 0A, 0C74, and 084F21 respectively.

IMPORTANT! *The common 0x hexadecimal prefix is not allowed. Using this prefix on MEM file hexadecimal values is flagged as a syntax error.*

There must be at least one data value following an address, up to as many data values that belong to the previous address value. Following is an example of the most common MEM file format:

```
@0000 3A @0001 7B @0002 C4 @0003 56 @0004 02
@0005 6F @0006 89...
```

UpdateMEM requires a less redundant format. An address specifier is used only once at the beginning of a contiguous block of data. The previous example is rewritten as:

```
@0000 3A 7B C4 56 02 6F 89...
```

The address for each successive data value is derived according to its distance from the previous address specifier. A MEM file can have as many contiguous data blocks as required. While the gap of address ranges between data blocks can be any size, no two data blocks can overlap an address range.

TIP: *UpdateMEM allows the free-form use of both // and /*...*/ commenting styles in the MEM file.*

The Vivado Design Suite also supports a MEM File format for memory initialization as described at this link in the *Vivado Design Suite User Guide: Synthesis* (UG901). The MEM File format supported by the Vivado Design Suite is different from the file format supported by UpdateMEM.

You should define the MEM file structure for Vivado tools to match the synthesis view of the memory as an array, which adheres to the Verilog language specification. The MEM file used for UpdateMEM should include spaces to match the `<Datawidth>` tag as defined in the memory map info (MMI) file. For more information, see MMI File Syntax.

According to the Verilog language specification, the memory is treated as an array, so for Vivado synthesis the MEM file for a 64k memory (256x256 array) should look as follows:

```
@00000000
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

Note: White space and/or comments are used to separate the numbers.

For the UpdateMem command, which has a post implementation physical view of the memory, the MEM file should look as follows:

```
@00000000
aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbb
```

Note: For UpdateMEM, the spaces that separate the words are determined by the MSB and LSB attributes of the `<Datawidth>` tag defined in the MMI file.

AMD XILINX

Chapter 3

Xilinx Parameterized Macros Memories

Xilinx Parameterized Macros (XPM) is a tool for creating RAM and ROM structures according to user-specified requirements. Within the XPM code, you specify a number of generics including memory size, clocking mode, ECC mode, and so forth. These requirements are then converted by the Vivado synthesis tool into the appropriate size and style of memory array.

XPMs are simple, lightweight, in-line customizable, solutions for common HDL flow use cases. They can also be considered as simple parameterizable IP. XPMs are synthesizable SystemVerilog-based HDL delivered with the Vivado Design Suite.

For details on XPMs, see the *Vivado Design Suite User Guide: System-Level Design Entry* (UG895).

For details on the various XPMs and their parameterization options, see the *UltraScale Architecture Libraries Guide* (UG974).

***Note*:** In the 2018.1 Vivado release and later, XPMs are enabled automatically in Project Mode, and in Non-Project Mode are used automatically during synthesis/implementation.

Because XPMs are used in RTL flows (or non-processor based designs), the `UpdateMEM` command needs a MEM (`.mem`) file as an argument; it cannot take an ELF file as an argument.

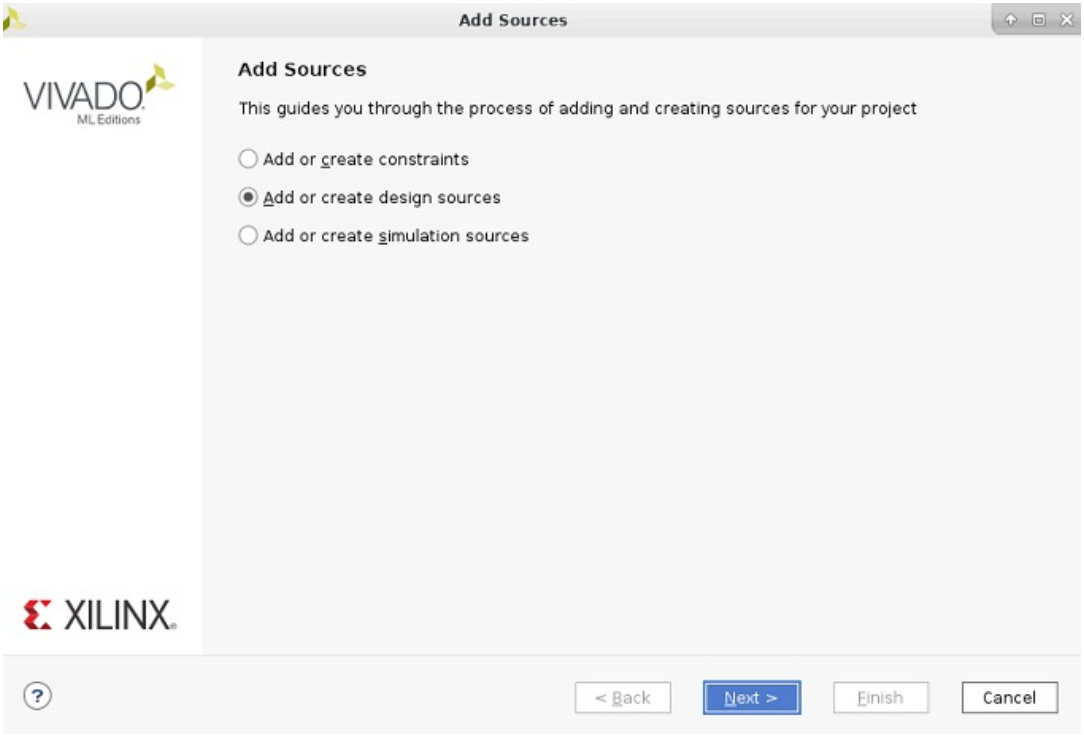
The limitations to using UpdateMEM with XPM memories are, as follows:

- ROM configurations need a MEM file prior to synthesis.
- ECC is not supported.

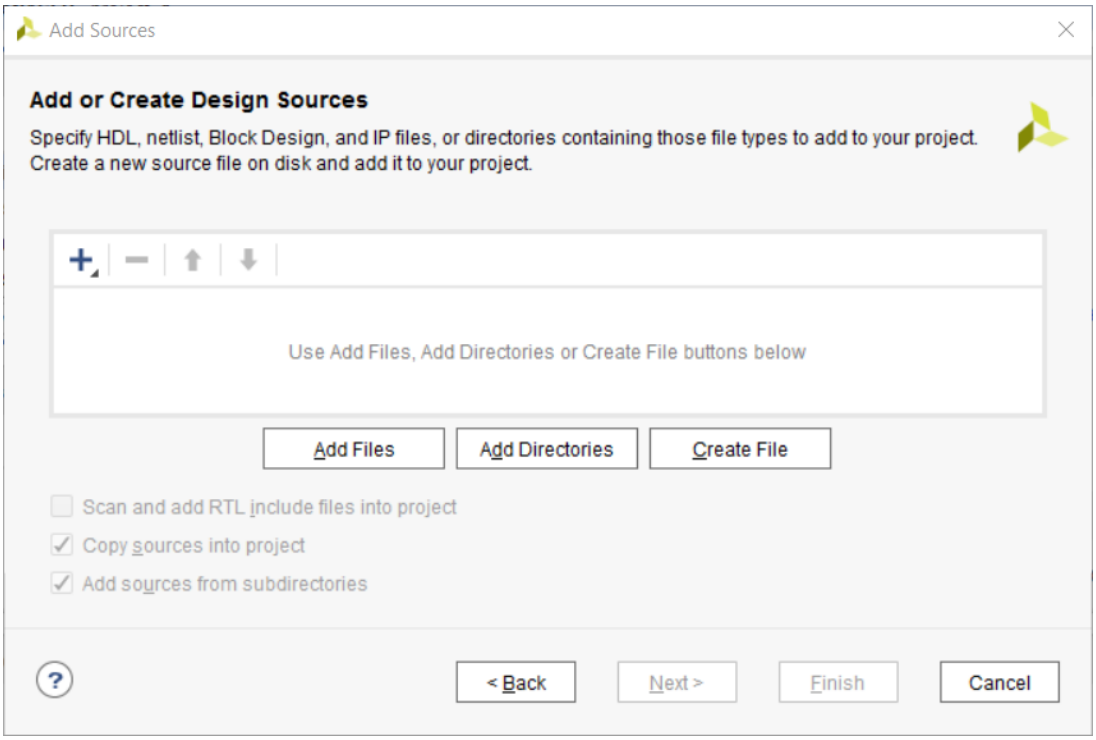
Using XPM Memory in Vivado

To use XPM Memory in Vivado you need to create design sources for the XPM memory. Follow the following steps to create XPM memory.

1. Launch Vivado and create a Project.
2. In the Sources window, right-click **Design Sources**, and select **Add Sources** from the popup menu.

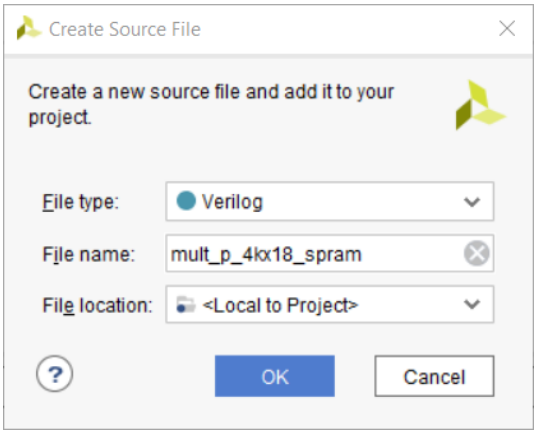


3. In the Add or Create Design Sources page, click **Create File**.

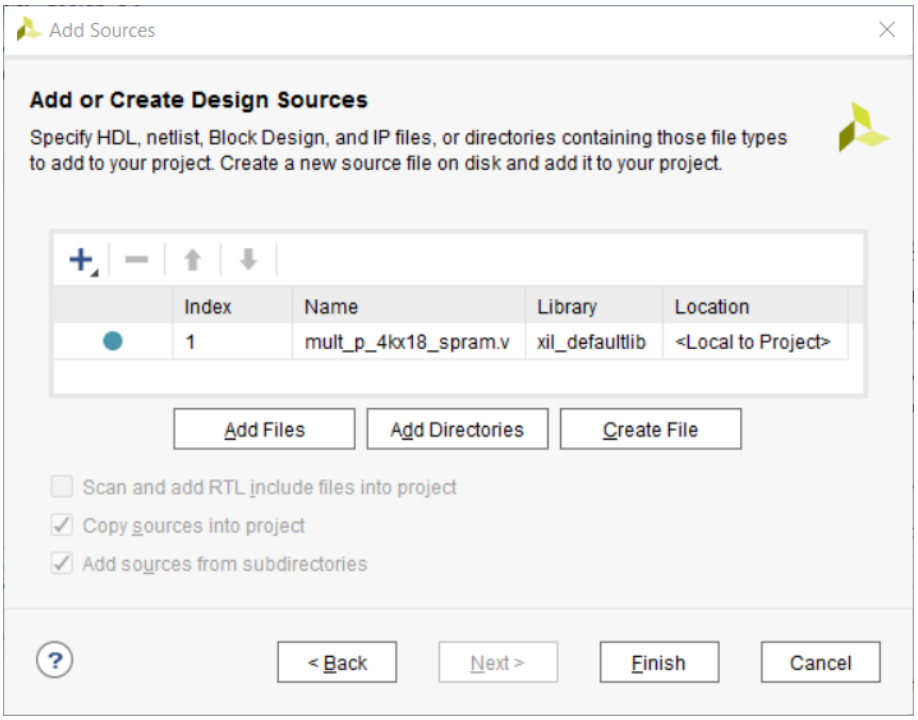


4. In the Create Source File dialog box, specify the HDL language of your choice from the File type drop-down menu, and type a name for the memory block being created in the File name field.

5. Keep the File location to its default value <Local to Project>.
6. Click **OK**, as shown in the following figure.

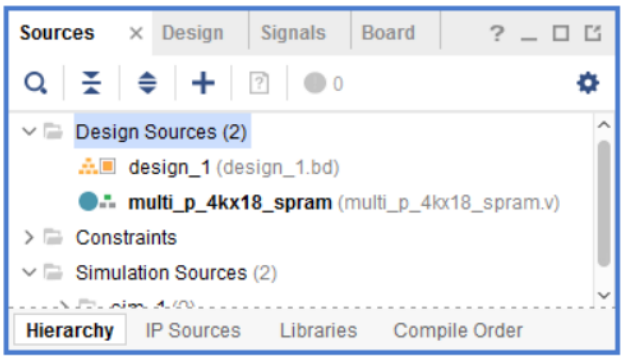


7. In the Add or Create Design Sources page, click **Finish**.

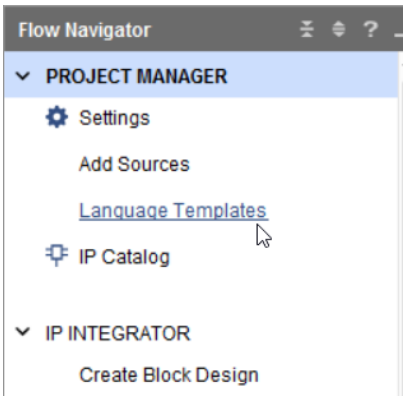


8. The Define Module dialog box opens. Click **Cancel** to dismiss the dialog box.
9. The Define Module dialog box asks to confirm that you indeed do not want to create the template for the HDL file.
10. Click **Yes**.

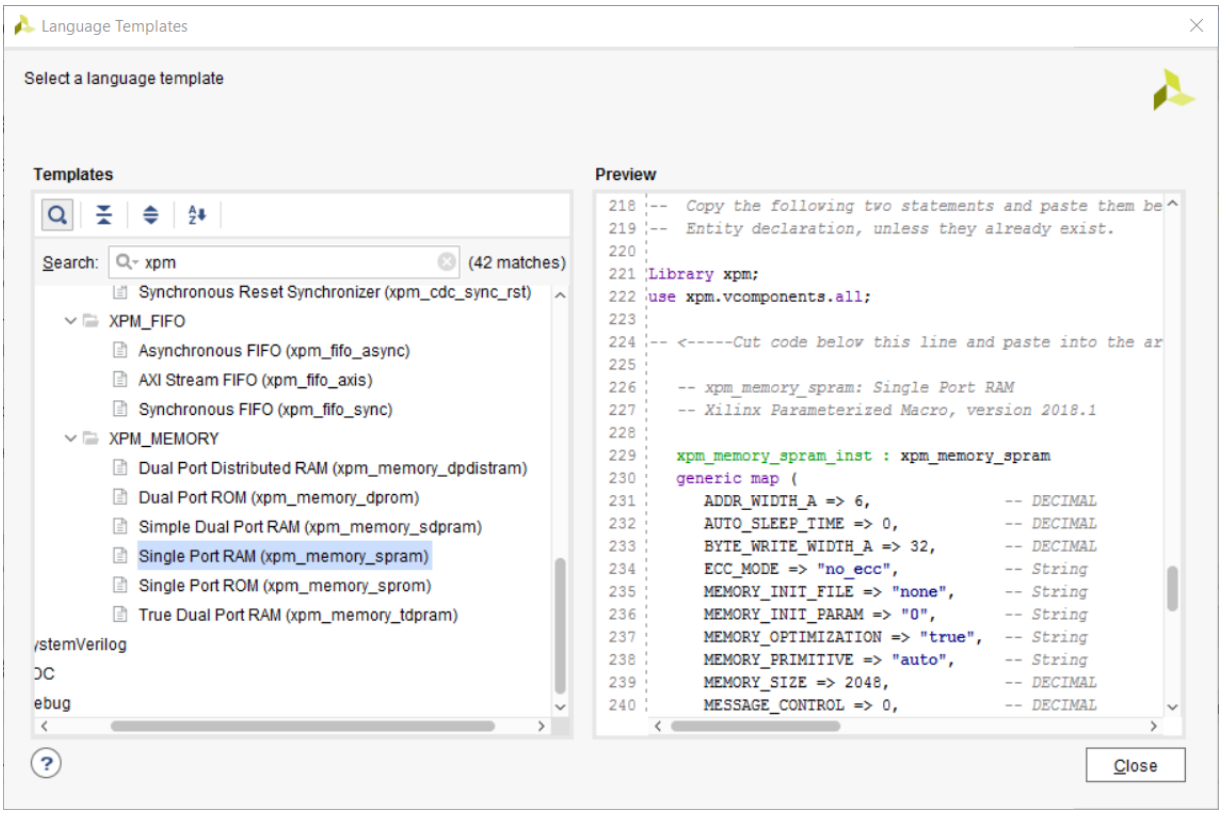
    This example copies a pre-existing XPRM template in the next steps into the HDL file.
11. Now you can see the newly created Verilog file in the Sources window.



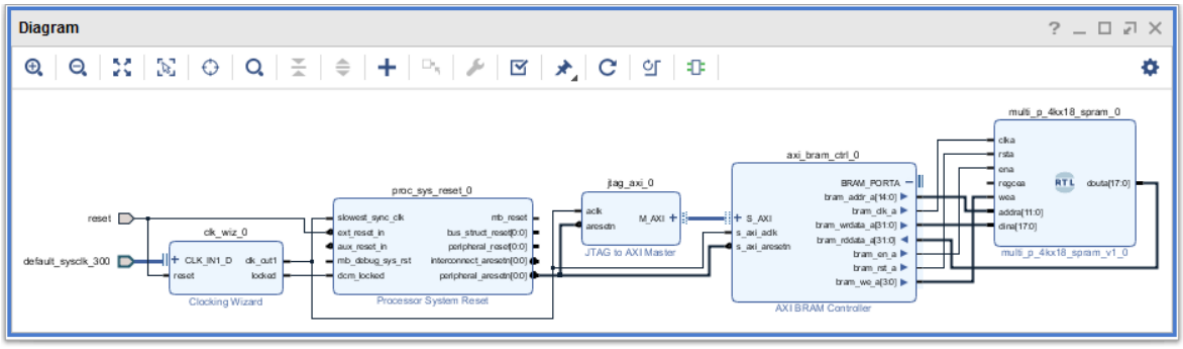
12. In the Flow Navigator, under Project Manager, click **Language Templates**.



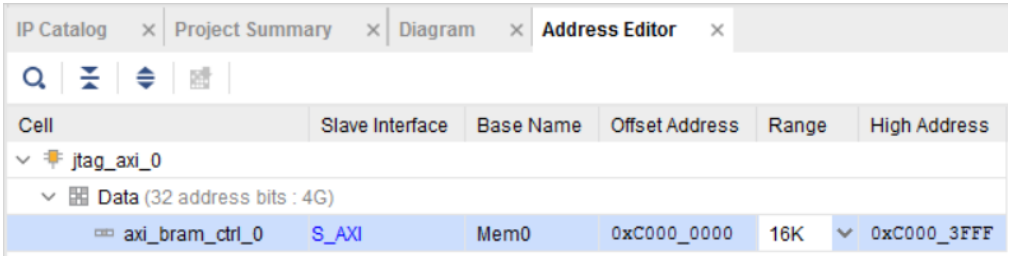
13. The Language Template dialog box opens. In the Search field type xpm and select the template for the appropriate HDL code (VHDL/Verilog), shown in the following figure.



14. Cut and paste the template for the Single Port RAM memory and add the instantiation template to the HDL file. Complete the definition of the HDL file by adding the appropriate entity and/or module definition. Set the USE_MEM_INIT_MMI parameter in the instantiation to 1.
15. Integrate your XPM memory block with the rest of the design. You can use the IP Integrator tool to integrate the XPM memory as a RTL module.



16. Set the appropriate depth of the memory instantiated in the Address Editor.



17. Generate output products, synthesize, implement, and create the bitstream for the design.
18. If you have a `mem` file, you can use that to populate the initialization strings of the XPM memory using the following `updatemem` command as an example:

```
updatemem -meminfo <mmi_file_name>.mmi -data <mem_file_name>.mem -bit
<bit file
name>.bit -proc <path to xpm memory instance> -out <output bit file
name>.bit
```

19. You can also use the `-debug` switch to see the `init_strings` of the XPM memory. Below is an example of using the `-debug` switch.

```
updatemem -debug -meminfo <mmi_file_name>.mmi -data <mem_file_name>.mem -
bit <bit
file name>.bit -proc <path to xpm memory instance> -out <output bit file
name>.bit >
dmp.txt<
```

AMD XILINX

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

***Note*:** For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

1. Zynq-7000 SoC Verification IP Data Sheet (DS940)

2. Zynq UltraScale+ MPSoC Verification IP (DS941)
3. *Zynq UltraScale+ MPSoC: Embedded Design Tutorial* (UG1209)
4. *Zynq-7000 SoC: Embedded Design Tutorial* (UG1165)
5. *Triple Modular Redundancy (TMR) LogiCORE IP Product Guide* (PG268)
6. *UltraScale Architecture-Based FPGAs Memory IP LogiCORE IP Product Guide* (PG150)
7. *Zynq-7000 SoC Technical Reference Manual* (UG585)
8. *Zynq-7000 SoC and 7 series Devices Memory Interface Solutions* (UG586)
9. *Vitis Unified Software Platform Documentation*
10. *Zynq-7000 SoC Software Developers Guide* (UG821)
11. *Vivado Design Suite Tcl Command Reference Guide* (UG835)
12. *Zynq-7000 SoC Packaging and Pinout Product Specifications* (UG865)
13. *Vivado Design Suite User Guide: Design Flows Overview* (UG892)
14. *Vivado Design Suite User Guide: Using the Vivado IDE* (UG893)
15. *Vivado Design Suite User Guide: System-Level Design Entry* (UG895)
16. *Vivado Design Suite User Guide: Synthesis* (UG901)
17. *Vivado Design Suite User Guide: Using Constraints* (UG903)
18. *Vivado Design Suite User Guide: Dynamic Function eXchange* (UG909)
19. *ISE to Vivado Design Suite Migration Guide* (UG911)
20. *Zynq-7000 SoC PCB Design Guide* (UG933)
21. *Vivado Design Suite Tutorial: Embedded Processor Hardware Design* (UG940)
22. *UltraScale Architecture Libraries Guide* (UG974)
23. *Vivado Design Suite User Guide: Designing IP Subsystems using IP Integrator* (UG994)
24. *Vivado Design Suite Tutorial: Designing IP Subsystems Using IP Integrator* (UG995)
25. *Zynq UltraScale+ Device Packaging and Pinouts Product Specification User Guide* (UG1075)
26. *Zynq UltraScale+ Device Technical Reference Manual* (UG1085)
27. *Vivado Design Suite User Guide: Creating and Packaging Custom IP* (UG1118)
28. *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
05/23/2022 Version 2022.1	
Initial release.	N/A

Please Read: Important Legal Notices

Copyright