RFNoC Frequently Asked Questions

Contents

- 1 Configuring the Stream Endpoint Buffer Size in RFNoC
  - 1.1 What is the SEP buffer size?
  - 1.2 How do I set the SEP buffer size?
  - 1.3 To what value should I set the SEP buffer size?
- 2 USRP DRAM
  - 2.1 How much and what speed DRAM is available on each USRP?
  - 2.2 What DRAM data rates can I expect on each USRP?
  - 2.3 What can the DRAM be used for?
  - 2.4 How do I add the Replay/DMA FIFO block to my FPGA image?
    - 2.4.1 Replay Example
    - 2.4.2 DMA FIFO Example
- 3 RFNoC Clocks
  - 3.1 What clocks are available for me to use?
  - 3.2 What are the clock frequencies?
    - 3.2.1 E31x
    - 3.2.2 E320
    - 3.2.3 N300/N310
    - 3.2.4 N32x
    - 3.2.5 X3xx
    - 3.2.6 X410
    - 3.2.7 X440
  - 3.3 How do I add a clock with a different frequency?
- 4 Xilinx Vivado
  - 4.1 Do I need a Vivado license to build custom RFNoC FPGA images?
  - 4.2 Which version and edition of Vivado do I need?
  - 4.3 Can I use a different Vivado version from the one required by my UHD version?
  - 4.4 Do I need to install all components of Vivado?
- 5 Building FPGA Images
  - 5.1 Why did my FPGA build fail to meet timing constraints?
  - 5.2 My design doesn't fit in the FPGA. What can I do to reduce the size?
  - 5.3 How do I create a Vivado project for my FPGA build?
  - 5.4 My FPGA takes a long time to build. What can I do to make builds faster?
    - 5.4.1 IP Generation
    - 5.4.2 FPGA Build
  - 5.5 My FPGA build failed with a cryptic message or no message at all. How do I debug this?
  - 5.6 I get a warning saying that an IP is locked, which results in errors later in the IP generation process. How do I resolve this?
  - 5.7 I see a "CRITICAL WARNING" in the build log. Is this expected?

Each stream endpoint (SEP) has an ingress buffer to store data received from others stream endpoints. This size of this buffer affects the data transfer rate that can be achieved when streaming to that endpoint. A larger ingress buffer in the stream endpoint means that there is more space to put data, minimizing idle time on the network. Additionally, streamers can queue up data before it is needed, reducing the chance of a buffer underflow.

The stream endpoint buffer size is set by adding a parameter under the endpoint you want to configure in the RFNoC image core YAML file. There are two parameters you can use to set the stream endpoint ingress buffer size in your RFNoC image core YAML file.

- `buff_size`: Buffer size in CHDR words. The size in bytes depends on the CHDR width. For example, if the `chdr_width` parameter for the device is 64, then each CHDR word is 8 bytes. So a buff size of 32768 would be 262,144 bytes or 256 KiB. See here for an example.
- `buff_size_bytes`: Buffer size in bytes. See here for an example.

The buffer size should be a power of two in size to make optimal use of FPGA RAM resources. The default FPGA bitstreams typically set them to the largest size the FPGA can fit in order to maximize performance. Here are some general recommendations:

- Set to `0` if you don't need to send data to that SEP.
- Set to `8192` bytes (8 KiB = 1 MTU) minimum in order to stream data packets.
- Set to `32768` bytes (32 KiB = 4 MTU) in order to stream at maximum rates between SEPs on the same FPGA.
- Set to `262144` bytes (256 KiB = 32 MTU) or lager for high performance streaming between a host computer and the FPGA.

Note that the requirements are application-dependent, so optimal sizes for your application may be different. MTU refers to the maximum transmission unit, which is the largest CHDR packet supported by the FPGA.

If you need to free up FPGA resources (particularly block RAM) for your application, you can reduce the SEP buffer sizes. Just keep in mind that the maximum streaming rate may be affected.

The table below summarizes the DRAM that is connected to the USRP for use by RFNoC.

USRP DRAM Summary

USRP Model	DRAM Size	Default DRAM Speed	Default User Interface
E31x	512 MiB	16-bit @ 800 MT/s (1.6 GB/s)	2 ch x 64-bit @ 100 MHz
E320	2 GiB	32-bit @ 1333 MT/s (5.33 GB/s)	4 ch x 64-bit @ 300 MHz
N3xx	2 GiB	32-bit @ 1300 MT/s (5.2 GB/s)	4 ch x 64-bit @ 303.819 MHz
X3xx	1 GiB	32-bit @ 1200 MT/s (4.8 GB/s)	2 ch x 64-bit @ 300 MHz
X410 (100 and 200 MHz BW)	4 GiB	64-bit @ 2.0 GT/s (16.0 GB/s)	4 x 64-bit @ 250 MHz
X410 (400 MHz BW)	4 GiB per bank (8 GiB total)	64-bit @ 2.0 GT/s (16.0 GB/s) per bank (32.0 GB/s total)	4 x 128-bit @ 250 MHz (using 2 banks)
X440 (400 MHz BW)	4 GiB per bank (8 GiB total)	64-bit @ 2.4 GT/s (19.2 GB/s) per bank (38.4 GB/s total)	8 x 128-bit @ 300 MHz (using 2 banks)

X440 (1600 MHz BW)	4 GiB per bank (8 GiB total)	64-bit @ 2.4 GT/s (19.2 GB/s) per bank (38.4 GB/s total)	2 x 512-bit @ 300 MHz (using 2 banks)

DRAM performance is highly application-specific. For example, reading vs. reading and writing simultaneously, one data stream vs. multiple data streams, random access vs. sequential access, etc., can give dramatically different performance. Below are some measurements taken on different USRPs where a Null-Source-Sink RFNoC block is directly connected to a DMA FIFO block to test maximum streaming rates through the DRAM. The DRAM is shared between channels, so throughput goes down as the number of channels going through the DRAM is increased.

Example DRAM Throughput (Per Channel)

USRP Model	BIST (MB/s)	1 Ch (MS/s)	2 Ch (MS/s)	3 Ch (MS/s)	4 Ch (MS/s)
E31x	666	166	91	N/A	N/A
E320	1361	340	299	191	148
N3xx	1368	341	295	191	144
X3xx	1347	336	274	N/A	N/A
X410 (100 and 200 MHz BW)	1288	321	316	314	303
X410 (400 MHz BW)	2801	697	672	672	672
X440 (400 MHz BW)	3360	798	784	616	461
X440 (1600 MHz BW)	8118	2007	2007	N/A	N/A

Notes:

1. E31x, N3xx, and X410 were tested using UHD 4.2. E320 and X3xx were tested using UHD 4.3.
2. BIST refers to the built-in self test, which gives a measure of raw data throughput for a single channel.
3. For MS/s, we assume 4 bytes per sample (sc16).
4. X410 with 400 MHz bandwidth uses two independent memory banks, with channels 0-1 on Bank 0, and channels 2-3 on Bank 1 by default. The traffic flows on Bank 0 and Bank 1 are independent and do not affect each other. Therefore, a 4-channel configuration has the same performance as a 2-channel configuration.
5. X440 uses two independent memory banks. For 400 MHz, channels 0-3 are on Bank 0 and channels 4-7 are on Bank 1 by default. For 1600 MHz, channel 0 is on Bank 0 and channel 1 is on bank 1 by default. The traffic flows on Bank 0 and Bank 1 are independent and do not affect each other. Therefore, a 2-channel configuration has the same performance as a 1-channel configuration.

- **DMA FIFO Block:** The DMA FIFO block is used in situations where you need a large buffer to store samples.
- **Replay Block:** The Replay block is used to record and play back RF data. For example, you can record data from a host computer, then play it back over the radio. Or, record data from the radio, then play it back later to the host for analysis, or play it back to a radio at a specific timestamp. See Using the RFNoC Replay Block in UHD 4 for additional information. The Replay block also has a FIFO capability for situations in which the DMA FIFO block is not available in your FPGA image.
- **Custom Blocks:** You can also create your own RFNoC block that uses DRAM. Refer to the DMA FIFO and/or Replay blocks as examples.

If the block you want is not included by default in the FPGA image you are using, you can add it to the RFNoC image core YAML file and rebuild the FPGA image using Vivado. See Getting Started with RFNoC in UHD 4.0 for additional information on customizing an RFNoC image.

Note: DRAM is not enabled by default on E31x FPGA builds because the FPGA is not large enough to fit the default image with DRAM. You will need to remove components from your RFNoC image's YAML file to make room, then build the E31x image with the variable DRAM=1 set, or modify the E31x Makefile to enable DRAM by default.

Note: The default DRAM configuration used for X410 and X440 changes depending on the configured bandwidth. The default parameters to use for each image type is shown in the table below.

When adding the blocks to your RFNoC image core YAML file, the parameters must be set correctly for the type of USRP you intend to use. The memory data width (`MEM_DATA_W`) and address width (`MEM_ADDR_W`) must match exactly. The number of ports (`NUM_PORTS`) must not exceed the maximum number available. You can use fewer ports to save resources if you don't need all the DRAM ports.

RFNoC Block Memory Parameters

USRP Model	MEM_DATA_W	MEM_ADDR_W	NUM_PORTS (Max)
E31x	64	29	2
E320	64	31	4
N3xx	64	31	4
X3xx	64	30	2
X410 (100 and 200 MHz BW)	64	32	4
X410 (400 MHz BW)	128	32	4
X440 (400 MHz BW)	128	32	8
X440 (1600 MHz BW)	512	32	2

The DMA FIFO has a few additional parameters that should be provided. The clock rate (`MEM_CLK_RATE`) must match the value below for the built-in self test (BIST) to work correctly. The base address (`FIFO_ADDR_BASE`) and address mask (`FIFO_ADDR_MASK`) are written as Verilog constants and can be changed depending on your application. The `FIFO_ADDR_BASE` parameter contains the byte address for the first byte of the memory region to use for each port. The `FIFO_ADDR_MASK` parameter contains the address mask for each port, which tells the FIFO how much memory to use for each port. For example, an address mask of `30'h1FFFFFFF` means that 0x1FFFFFFF+1 bytes (i.e., 0x20000000 bytes or 512 MiB) will be used by the corresponding port. The address mask must be 1 less than a power of 2.

The example values in the table below use the entire memory and divide it evenly between all available ports.

DMA FIFO Parameters

USRP Model	MEM_CLK_RATE	FIFO_ADDR_BASE	FIFO_ADDR_MASK
E31x	"200e6"	"{29'h10000000, 29'h00000000}"	"{29'h0FFFFFFF, 29'h0FFFFFFF}"
E320	"300e6"	"{31'h60000000, 31'h40000000, 31'h20000000, 31'h00000000}"	"{31'h1FFFFFFF, 31'h1FFFFFFF, 31'h1FFFFFFF, 31'h1FFFFFFF}"

N3xx	"303819444"	"{31'h60000000, 31'h40000000, 31'h20000000, 31'h00000000}"	"{31'h1FFFFFFF, 31'h1FFFFFFF, 31'h1FFFFFFF, 31'h1FFFFFFF}"
X3xx	"300e6"	"{30'h20000000, 30'h00000000}"	"{30'h1FFFFFFF, 30'h1FFFFFFF}"
X410 (100 and 200 MHz BW)	"250e6"	"{32'hC0000000, 32'h80000000, 32'h40000000, 32'h00000000}"	"{32'h3FFFFFFF, 32'h3FFFFFFF, 32'h3FFFFFFF, 32'h3FFFFFFF}"
X410 (400 MHz BW)	"250e6"	"{32'h80000000, 32'h00000000, 32'h80000000, 32'h00000000}"	"{32'h7FFFFFFF, 32'h7FFFFFFF, 32'h7FFFFFFF, 32'h7FFFFFFF}"
X440 (400 MHz BW)	"300e6"	"{32'hC0000000, 32'h80000000, 32'h40000000, 32'h00000000, 32'hC0000000, 32'h80000000, 32'h40000000, 32'h00000000}"	"{32'h3FFFFFFF, 32'h3FFFFFFF, 32'h3FFFFFFF, 32'h3FFFFFFF, 32'h3FFFFFFF, 32'h3FFFFFFF, 32'h3FFFFFFF, 32'h3FFFFFFF}"
X440 (1600 MHz BW)	"300e6"	"{32'h00000000, 32'h00000000}"	"{32'hFFFFFFFF, 32'hFFFFFFFF}"

See x310_rfnoc_image_core.yml for an example of how to instantiate the Replay block in the RFNoC image core YAML description. The following is a generic example that can be used for any USRP:

```
noc_blocks:
  # Instantiate the replay block
  replay0:
    block_desc: 'replay.yml'
    parameters:
      NUM_PORTS: <see table>
      MEM_DATA_W: <see table>
      MEM_ADDR_W: <see table>

connections:
  # Connect each port of the replay block to a stream endpoint
  - { srcblk: <epN>,   srcport: out0,  dstblk: replay0, dstport: in_0 }
  - { srcblk: replay0, srcport: out_0, dstblk: <epN>,   dstport: in0  }
  - { srcblk: <epN+1>, srcport: out0,  dstblk: replay0, dstport: in_1 }
  - { srcblk: replay0, srcport: out_1, dstblk: <epN+1>, dstport: in0  }
  ... repeat for each remaining Replay port
  # Connect the replay block memory interface to the USRP DRAM
  - { srcblk: replay0, srcport: axi_ram, dstblk: _device_, dstport: dram }

Connect the DRAM clock to the block:
clk_domains:
  # Connect the DRAM clock to the replay block
  - { srcblk: _device_, srcport: dram, dstblk: replay0, dstport: mem }
```

See e320_rfnoc_image_core.yml for an example of how to instantiate the DMA FIFO block in the RFNoC image core YAML description. The following is a generic example that can be used for any USRP:

```
noc_blocks:
  # Instantiate the DMA FIFO block
  fifo0:
    block_desc: 'axi_ram_fifo.yml'
    parameters:
      NUM_PORTS: <see table>
      MEM_DATA_W: <see table>
      MEM_ADDR_W: <see table>
      FIFO_ADDR_BASE: <see table>
      FIFO_ADDR_MASK: <see table>
      MEM_CLK_RATE: <see table>

connections:
  # Connect each port of the DMA FIFO block to a stream endpoint, or insert it
  # into the data path where desired. This examples uses stream endpoints.
  - { srcblk: <epN>,   srcport: out0,  dstblk: fifo0,   dstport: in_0 }
  - { srcblk: replay0, srcport: out_0, dstblk: <epN>,   dstport: in0  }
  - { srcblk: <epN+1>, srcport: out0,  dstblk: fifo0,   dstport: in_1 }
  - { srcblk: fifo0,   srcport: out_1, dstblk: <epN+1>, dstport: in0  }
  ... repeat for each remaining FIFO port
  # Connect the DMA FIFO block memory interface to the USRP DRAM
  - { srcblk: fifo0, srcport: axi_ram, dstblk: _device_, dstport: dram }

clk_domains:
  # Connect the DRAM clock to the replay block
  - { srcblk: _device_, srcport: dram, dstblk: fifo0,  dstport: mem }
```

Each device has different clocks available. See below for a list of clocks exposed to RFNoC. Although they have intended purposes, you can use any of these clocks for any purpose. The `rfnoc_chdr_clock` is a good default choice. This clock is always available in your block, even if it is not explicitly connected in the RFNoC image YAML description.

See the table below for the clock rates. The radio clock rate depends on the master clock rate.

Clock Name	Description	Frequency
`rfnoc_chdr`	RFNoC CHDR clock	100 MHz
`rfnoc_ctrl`	RFNoC Control clock	40 MHz
`dram`	DRAM interface clock	100 MHz
`radio`	Radio interface clock	Same as master clock rate

Clock Name	Description	Frequency
`rfnoc_chdr`	RFNoC CHDR clock	200 MHz
`rfnoc_ctrl`	RFNoC Control clock	40 MHz
`dram`	DRAM interface clock	166.667 MHz

`radio`	Radio interface clock	Same as master clock rate (200 kHz to 61.44 MHz)

Clock Name	Description	Frequency
`rfnoc_chdr`	RFNoC CHDR clock	200 MHz
`rfnoc_ctrl`	RFNoC Control clock	40 MHz
`dram`	DRAM interface clock	303.819 MHz
`radio`	Radio interface clock	Same as master clock rate (122.88 MHz, 125.0 MHz, or 153.6 MHz)

Clock Name	Description	Frequency
`rfnoc_chdr`	RFNoC CHDR clock	200 MHz
`rfnoc_ctrl`	RFNoC Control clock	40 MHz
`dram`	DRAM interface clock	303.819 MHz
`radio`	Radio interface clock	Same as master clock rate (200 MHz, 245.76 MHz, or 250 MHz)

Clock Name	Description	Frequency
`rfnoc_chdr`	RFNoC CHDR clock	187.5 MHz
`rfnoc_ctrl`	RFNoC Control clock	93.75 MHz
`ce`	Compute Engine clock	214.286 MHz
`dram`	DRAM interface clock	300 MHz
`radio`	Radio interface clock	Same as master clock rate (184.32 MHz or 200 MHz)

Clock Name	Description	Frequency
`rfnoc_chdr`	RFNoC CHDR clock	200 MHz
`rfnoc_ctrl`	RFNoC Control clock	40 MHz
`dram`	DRAM interface clock	250 MHz
`radio`	Radio interface clock	122.88 MHz when master clock rate is 122.88, 245.76, or 491.52 MHz 125 MHz when master clock rate is 125, 250, or 500 MHz
`radio_2x`	Radio interface clock 2x	Twice the frequency of `radio`

Clock Name	Description	Frequency
`rfnoc_chdr`	RFNoC CHDR clock	200 MHz
`rfnoc_ctrl`	RFNoC Control clock	40 MHz
`dram`	DRAM interface clock	300 MHz
`radio0`	Radio interface clock for daughterboard 0	Daughterboard 0 master clock rate divided by 8 (e.g., 62.5 MHz if master clock rate is 500 MHz)
`radio1`	Radio interface clock for daughterboard 1	Daughterboard 1 master clock rate divided by 8
`radio0_2x`	Radio interface clock 2x for daughterboard 0	Twice the frequency of `radio0`
`radio1_2x`	Radio interface clock 2x for daughterboard 1	Twice the frequency of `radio1`

Adding custom clocks is not directly supported yet. Describing them in the YAML file will not cause them to be generated for you. If you can't use any of the available clocks, you can modify the HDL code to generate a clock.

If you only need the clock within your own RFNoC block, you can modify the HDL for your block to generate the clock that you need from one of the available clocks. To do this, add a new clock to your block's YAML description, connect the available clock to your block in the YAML description of your RFNoC image, then add a Xilinx MMCM IP instance to your block's HDL and connect the available clock to its input.

If the clock is needed by multiple RFNoC blocks, or if you want to change an existing clock, you can modify the HDL for the USRP you are using to add or change a clock. If you add a new clock to the RFNoC image core, you must also update the BSP YAML file (located in <repo>/host/include/uhd/rfnoc/core) so that the `rfnoc_image_builder` knows that the clock exists. How and where the clocks are generated varies between USRPs. Please refer to the source code for that USRP (<repo>/fpga/usrp3/top).

All RFNoC-capable USRPs use Xilinx FPGAs that require a license to use Vivado, except for E31x USRPs, which can use the free Vivado HL WebPACK Edition. Vivado is required to build FPGAs for RFNoC.

See the UHD User Manual for the latest Vivado version requirements. UHD versions 4.0 through 4.2 require Vivado 2019.1.

For E31x devices, you can use the free Vivado HL Webpack. For all other USRPs, you can use Design Edition or System Edition. We recommend Design Edition, unless you plan to use System Generator for DSP. System Generator is not required by RFNoC.

This is technically possible, but it can be a lot of work to convert and adapt all of the IP to a new Vivado version, and your custom combination of UHD and Vivado versions will not have been tested or validated by Ettus Research. Therefore, this is not recommended or supported.

No. You only need to install device support for the FPGA you intend to build. Other devices can be unchecked to save disk space. The following FPGA types are used by USRPs:

- **SoCs > Zynq-7000:** E31x, E320, N3xx
- **SOCs > Zynq UltraScale+ RFSoC:** X410
- **7 Series > Kintex-7**: X3xx

The Software Development Kit (SDK) is typically not required, but can be installed if desired.

The Cable Drivers are needed if you plan to do JTAG download or debug. Note that on Linux, the cable drivers are copied to the install folder, but are not installed onto your system automatically. See Xilinx UG973 for instructions on installing the cable drivers on Linux.

FPGAs have clocks that trigger the transfer of data between internal registers. The Vivado tool does a timing check near the end of the build to ensure that the paths from each driving register or port to each receiving register or port are not too long for the specified clock period or delay constraints. When it says "The design did not satisfy timing constraints" it means that Vivado couldn't arrange the logic on the chip in a way that meets all requirements. There are several reasons this might happen:

- You added new logic to the design with too much logic between registers. In this case, you should modify your design to make meeting timing easier.
- You added new logic, but made a mistake in which you're trying to use the wrong clock or reset, which makes it difficult to meet timing. In this case you need to correct the mistake in your design.
- The design has become too crowded, making it difficult for the tools to meet the timing requirements. In this case you need to remove something to make more room.
- Bad luck. The tools use pseudorandom algorithms to find solutions to really hard problems, and sometimes it doesn't find a good solution even when one is possible. In this case you can make a minor change to the design and build again to see if it does better the second time. If you don't change anything, Vivado will normally give you identical results for each build. In UHD 4.4 and later you can add the `BUILD_SEED=1` option to the `make` arguments to change a build seed that will affect the build results. Using a different seed number for each build will ensure that you get a unique build result each time. 0 is the default seed if not specified. Random build failures occur occasionally for some FPGA targets, in which case you should retry the build with a different seed.

The FPGA tools produce a timing report that says exactly which path failed to meet timing. Sometimes that can point you in the right direction. But sometimes the path indicated only failed because of another path that's even more difficult. Open `post_route_timing_summary.rpt` in the build output folder and search for "(VIOLATED)" to find the path(s) that failed.

Read the `post_synth_util.rpt` to determine what resource(s) you are running out of in order to know what kinds of changes are needed. Below are several easy ways to reduce the resource utilization of the FPGA.

- If you are not using all RF channels of your device, modify the FPGA YAML file to remove the DDC, DUC, and Radio blocks for the unused channels, then regenerate the FPGA code using `rfnoc_image_builder`. Note that you may need at least one Radio block for RFNoC to work properly. You may also remove the DDC and/or the DUC if your application uses full bandwidth for one or more channels and therefore doesn't require up or down conversion.
- If you are not using DRAM, remove the Replay or DMA FIFO blocks. Also, on X4xx, change the `DRAM_CH` variable to 0 in the Makefile for the FPGA target you are building.
- If you do not need all SFP ports, use a build target that matches your needs. For example, on X4xx, the "X1" option (one 10 Gbps lane) uses the least resources whereas "X4" (four 10 Gbps lanes) uses a lot more, and the "CG" option (four 25 Gbps lanes) uses the most.
- If you do not need the full bandwidth of the device, use a smaller bandwidth option. For example, on X410, the "_100" option (100 MHz bandwidth) uses less resources than the "_200" option (200 MHz bandwidth).
- Add the `crossbar_routes` definition to the FPGA YAML file to include only the crossbar paths required for your application. This is an advanced feature in UHD 4.5 and later. This must be done carefully to avoid removing essential paths. See the X440 YAML files for examples.

Other reductions are possible but require advanced knowledge of UHD and/or RFNoC to avoid breaking key functionality of the device.

Vivado supports two modes of operation known as "project mode" and "non-project mode". Project mode is more user-friendly because it creates a project file that is managed by Vivado and works natively in the Vivado GUI. Non-project mode is generally used by more advanced users who want full control over the Vivado build process and is typically used in fully scripted or automated build flows. The USRP build flow in UHD uses non-project mode. As a result, there is no Vivado project file by default.

It is possible to create a project file from the USRP build flow with the following steps:

1. Start the USRP FPGA build in the GUI by adding `GUI=1` to the `make` arguments. Example:
   `make X410_X4_200 GUI=1`
2. After the build completes, run the following command in the TCL Console of Vivado to create the project file and switch to project mode:
   `save_project_as project_name project_dir`
   In this example, "project_name" is the name you want to give the project file and "project_dir" is the directory in which you want to put the project.
3. Set the compile order to automatic:
   `set_property source_mgmt_mode All [current_project]`

This project file can now be used independently of the normal FPGA build flow in UHD. It is up to the user to update this project file as the design changes since it will not be managed by the normal build flow in UHD.

High-performance computers are recommended for FPGA builds since an FPGA build can take several hours.

The build process is divided into two steps, IP generation and the FPGA build.

This process can take several hours by default and is run automatically, if needed, when you build an FPGA target. Fortunately, this only needs to be done once for each USRP type and won't run again unless IP is changed.

You can speed up the IP generation by running this step with multiple jobs. For example:

```
$ make -j 4 X410_IP
```

This example will build four IP cores at a time. Note that this generally requires 4 times as much memory and needs at least 4 CPU cores. You can adjust the number of parallel jobs based on the amount of system memory and/or CPU cores you have available.

Unfortunately, increasing the number of jobs does not speed up FPGA performance because there is only one Vivado instance for the FPGA build. Vivado, by default, will use multiple CPU cores, where possible, but this does not significantly improve build performance since many parts of the build are not easily parallelizable.

One way to shorten the build time is to reduce the size of the design. See above on how to reduce the size of your design.

In the case where you need to build multiple FPGA types, you can use the jobs option with `make` to build multiple FPGAs simultaneously, which can dramatically reduce the time required per build. Note that this requires a significant amount of memory and CPU cores and therefore is only

recommended for systems that can handle such loads. An example is shown below for building two FPGA images in parallel:

```
$ make -j 2 X410_X4_200 X410_CG_400
```

It is also possible to open separate terminal instances and run one build in each instance to get the same effect. Do not build the same FPGA target in multiple instances, since multiple builds for the same target would conflict as they try to access and update the same files.

When you build an FPGA target, a build directory is created in the FPGA's top directory that contains all the build outputs. Here you'll find the `build.log` file as well as report files and checkpoints. Not all log information is printed to the console during build, so make sure you check the `build.log` file for details. It may contain a useful error message that was not printed to the console.

Builds often fail when Vivado encounters an internal error or runs out of memory. For internal errors, the error message is typically not very helpful and is often due to a bug in Vivado. When Vivado runs out of memory, it may immediately terminate without giving any error message at all. Consider monitoring the memory usage during the FPGA build to see if you are approaching your system's limit.

If you have made changes to the design, try building an unmodified FPGA image from scratch to ensure the build process is working properly on your system. If this works, try adding your changes incrementally until the section of code causing the problem is identified.

Note that such errors are often beyond the control of Ettus Research and reaching out to Xilinx support is a better option if it is truly a Vivado issue.

Vivado "locks" IP, for example, when it needs to be updated for the running version of Vivado or FPGA device type. This is intended to force the user to fix the issue and to avoid building incompatible IP. Build failures related to IP being locked should never occur during a normal build. The IP version in the UHD repo always matches the Vivado version required for that release of UHD.

This can happen if you have used the wrong version of Vivado or do not have the correct Vivado patches installed. Refer to the `Generation 3 USRP Build Documentation` section of the [[UHD and USRP User Manual|UHD Manual] for the required version and patches. When you run the `source setenv.sh` step to setup your environment, the script will check to make sure you are using the correct version.

In some cases, reinstalling Vivado might be required.

Once the correct Vivado version and patches are installed, you will need to remove all build products (to remove any locked IP that was generated) and retry the build. For example:

```
$ source setupenv.sh     # Setup environment and check the Vivado version
$ make cleanall          # Remove any bad IP that was generated
$ make X410_X4_200       # Start the build process again
```

There are many critical warnings that appear during the build process that can be safely ignored. For example, you may see the following:

```
CRITICAL WARNING: [Vivado 12-1790] Evaluation License Warning: This design contains one or more IP cores that use separately licensed feat
```

The FPGA builds include IP for which the licenses are included with Vivado, but Vivado prints the warnings anyway. As long as you have a Vivado license and a bitstream was successfully generated, the IP should work as expected.