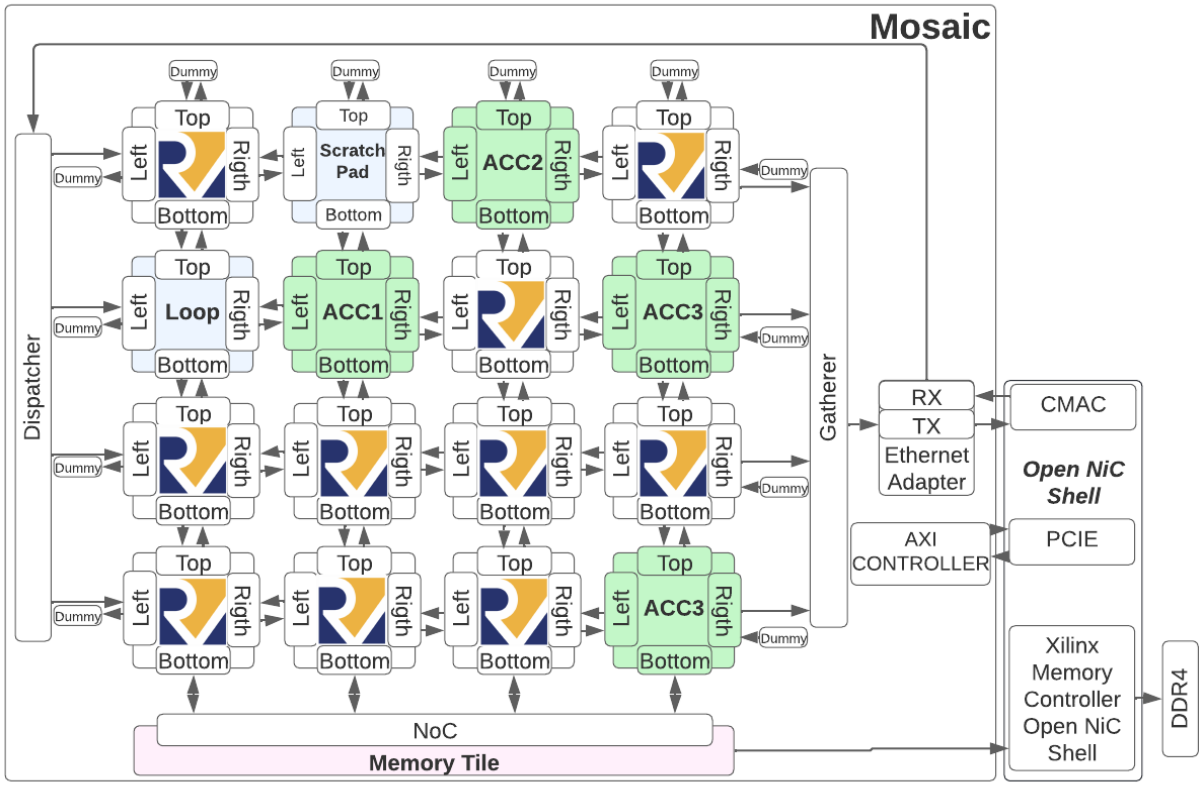
Lawrence Berkeley National Laboratory

Cyclotron Road
Berkeley, CA

Mosaic tutorial

Jun 22, 2023

1. Overview



Example of MoSAIC

MoSAIC is a multi-tiled architecture for accurate and fast message-driven computing exploration.

- MoSAIC is written in SystemVerilog.

- A tile can house a lightweight RISCV processor such as the PICORV32 or a generic accelerator.
- The tiles are connected through a lightweight network on chip (NoC) following an AXIStream protocol.
- Each tile has a hardware message queue for inter-tile communication used for message-driven computation.
- A RISCV ISA extension enables straight access to the physical message queue or remote memory through C/C++ primitives (qPut, qGet, qWait, qPoll, mPut, mGet).
- Partitioned global address scheme (PGAS).

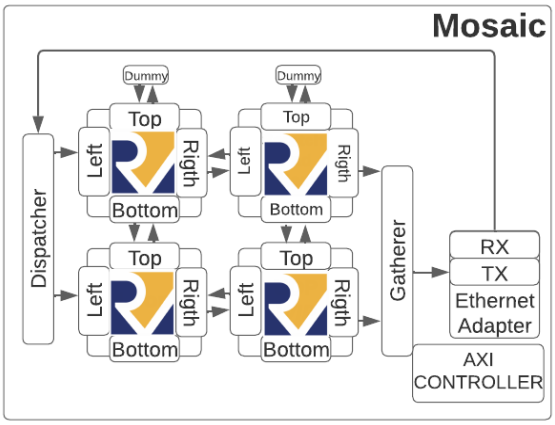
2. Dependencies

- RISCV tools set for PICORV 32. Built it following this link: picorv32. QUEMU is broken but the tools will still work.
- ICARUS Verilog: version 11 or higher
- GTKWAVE
- VIVADO

3. Getting started with MoSAIC

Let's first generate a small system with four tiles. All the tiles are equal and have a RISCV processor with an ISA extension for message queues (qISA). To generate this system we will

1. Create a testcase
2. Generate the configuration files
3. Run a simulation using i-verilog or Vivado
4. Check the simulation results.
5. When ready, create an IP to be instantiated in open-nic-shell.



3.1. Create a testcase

Create the testcase. Go to doc/tutorial_files/testcases/ and copy the mosaic_tutorial_0.pl over to tools/generate. All testcases have three sections: ***header, user,*** and ***footer***. The header and footer shouldn't need modifications for basic functionality. In the user section, we set all the configurations for MoSAIC. The minimum configuration for a functional mosaic is set by the following parameters:

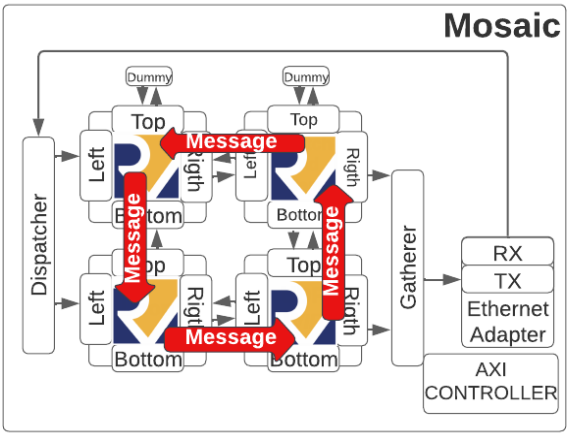
Unset

```
#- Set the number of rows (r) and columns (c) in MOSAIC array.
$param{'r'} = 2;
$param{'c'} = 2;

#- Set the type for each tile
@tile_array = (['pico', 'pico'],
               ['pico', 'pico']);

#- Set the firmware to load in each scratchpad
@pico_program  = ('test_tile_nop.hex', 'test_tile_nop.hex',
                  'test_tile_nop.hex', 'test_tile_nop.hex');
```

There are parameters to control simulation features. In this case we can set 'run_sim' to launch a simulation after creating all the configuration files.

```
$param{'run_sim'} = 1;
```

3.2. Create stimulus



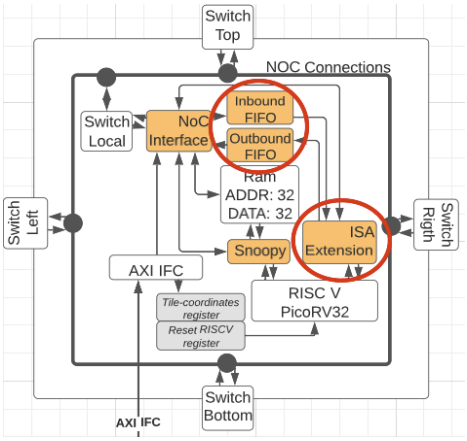
Let's create a program that passes a message counter-clockwise and each tile increments the data by 1. Go to doc/tutorial_files/c_step1 and open the file *send_msg.c*.

Each tile has an identification number or coordinate (tile_id) in (y,x) format, with column (y) and row (x). To represent x/y positions in the array we use 3 bits for a total of 6 bits for the coordinates. For our example, the corresponding tiles ids are:

	Column 0	Column 1
Row 0	000-000 (6'h0)	0001-000 (6'h8)
Row 1	000-001 (6'h1)	001-001 (6'h9)

Include the "mq.h" header in the c code. This header enables the access to the qISAextension through the functions *qPut(), qGet(), qPoll(), qWait(), mPut(), mGet()*. We will use these functions during the tutorial and explain what these are for in context.

A little detour. Let's take a look at the Tile_picorv tile. In this tile, we have an ISA extension that connects a hardware message queue with the RISCV such that we can use the software primitives mentioned above.



For the C code, we assume all the RISCV processors are running the same program and do different things, or use different data depending on the tile_id. Thus, the first task is to ask who am I? What is my tile_id? As shown below. For this, we also include the stdlib library which includes among others the atoi function.

```C/C++
#include <stdlib.h>
#include "mq.h"
#include "wrapper_mq.h"

void main (int argc, char *argv[]){
    int tile_id = atoi(argv[1]);
```

If the tile_id is (0,0) we start the game by sending the first message using the qPut(destination_tile, data) function. This will generate a packet originating from tile_00 which destination is a physical hardware queue placed at tile (0,1).

```C/C++
   if (tile_id == 0) {
     qPut(1, 1);  // qPut(dest, data)
   }
```

We will make the processors wait until they have a message in the queue using the qWait(unused, data) function. The data variable contains the first message in the queue, in this case we place it in the variable temp.

```C/C++
   qWait(0,temp);
```

Once there is something in the queue, the processor continues. We use w_qGet(unused, data) to pop the data belonging to the first message out of the queue. This will be stored in the variable ball.

```C/C++
 w_qGet(0,&ball);
```

Once it get the ball, I will send a new message to the neighboring tile using qPut(destination_tile, data). In this case I increment ball by one before sending it. A last detail, the destination for the message depends on the local tile_id.

```C/C++
   /* Set the neighboor */
   if      (tile_id == 0) {dest_tile = 1;}
   else if (tile_id == 1) {dest_tile = 9;}
   else if (tile_id == 9) {dest_tile = 8;}
   else if (tile_id == 8) {dest_tile = 0;}

   /* Send it to the neighbor */
   qPut(dest_tile,ball+1);
```

The complete code looks like this

```C/C++
#include <stdlib.h>
#include "mq.h"
#include "wrapper_mq.h"
void main (int argc, char *argv[]){
   int tile_id = atoi(argv[1]);
   int dest_tile;
   uint32_t ball;

   if (tile_id == 0) {
     qPut(1, 1);       // qISA
   }
   qWait(0,temp);      //qISA
   w_qGet(0,&ball);    //qISA
   if      (tile_id == 0) {dest_tile = 1;}
```

```
    else if (tile_id == 1) {dest_tile = 9;}
    else if (tile_id == 9) {dest_tile = 8;}
    else if (tile_id == 8) {dest_tile = 0;}
    qPut(dest_tile,ball+1); //qISA
}
```

Use the send_msg.pl script to generate the hex files for all the RISCVs. In the directory there will be send_msg32_<tile_id>.hex files, one per tile in the array.

3.3. Modify the testcase

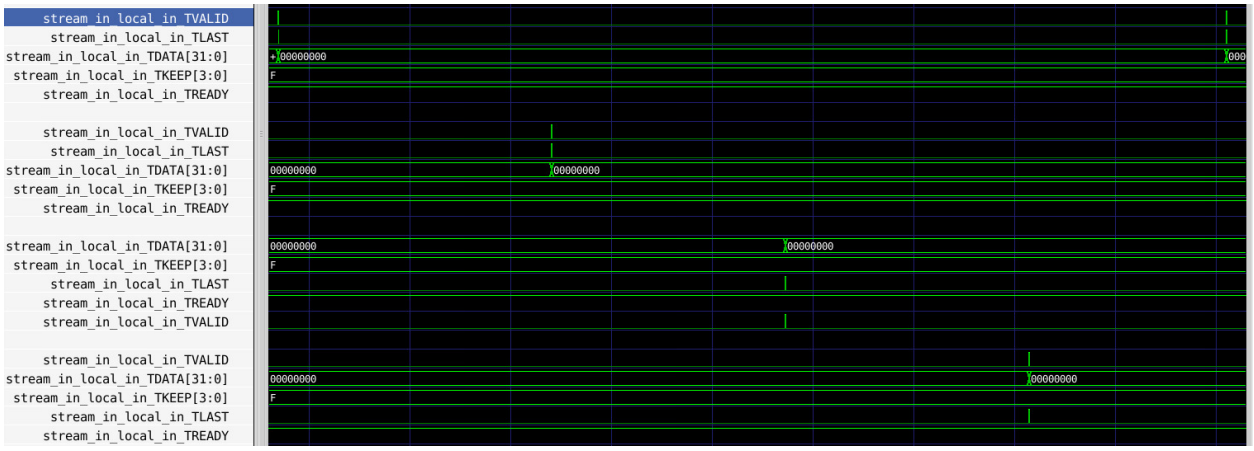
First, add the path where the hex files are, or copy the hex files to the default location. Then, add the new hex files using the @pico_program array. Finally increase the simulation time and re-launch the testcase.

```C/C++
 $param{'firmware_path'} = <set_me>;
 @pico_program  = ('send_msg32_0.hex', 'send_msg32_8.hex',
                   'send_msg32_1.hex', 'send_msg32_9.hex');
 #- Simulation Time
 $param{'sim_loop'} = 140;
```

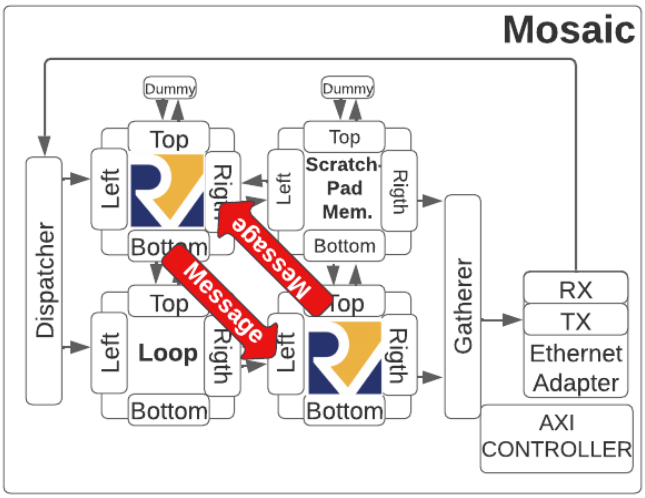
3.4. For the firmware path use an absolute path.

3.5. Check the simulation results

Go to each tile and add the stream_in_local_in_T<> signals which are the connection between the tile logic and the NoC. the diagram below shows the packets coming out of each tile as they pass the ball. If you zoom in the two packets generated at tile_00, the data for the first one is 1 (which initiates the transactions) and for the last one, the data is 5 which is the accumulated data as the message passes through the tiles.



4. Replace RISCV for accelerators



As part of the github repository there are two other type of tiles. The Tile_scratchpad contains a SRAM based scratchpad that can be read/write from/to any other tile through the NoC. The Tile_loop is a dummy tile and sends the message back as it is to its source. All Tiles are expected to have the same interface to be integrated in MoSAIC.

To replace some of the RISCV processors with these tiles we need to modify the testcase. Use @tile_array to set the type of tile that you want at each place, and @pico_program to load data in the scratchpad and leave it empty for the loop tile, since this one does not have nay memory.

```
C/C++
 @tile_array = (['pico', 'spad'],
                ['loop', 'pico']);
 @pico_program  = ('send_msg32_0.hex', 'test_tile_nop.hex',
                                   '', 'send_msg32_9.hex');
```

Modify the C code to continue sending packets between the RISCVs, generate the hex files and execute the testcase and check gtkwave to see the spad and the loop tiles as part of the array.

5. Add new accelerators

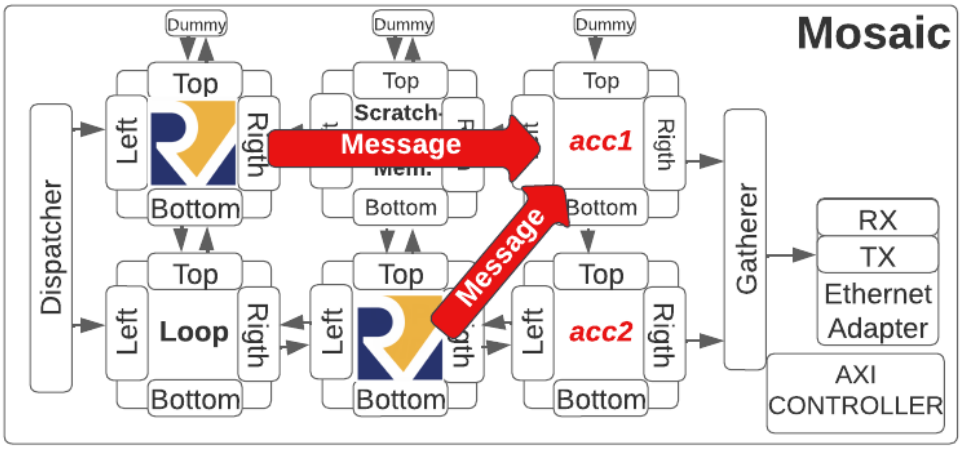
In the tutorial document there are two dummy accelerators (acc1, acc2) located in doc/tutorial_files/acc1(2)_tile.

5.1. Add the accelerators to the generation tool

Copy the doc/tutorial_files/mosaic_tutorial_3.pl to /tools/generate. This testcase adds a new section to add the accelerators to be instantiatied as part of MoSAIC. We need to add the verilog module for the Tile and assign it an alias. For example the Tile_picorv alias is pico, the Tile_scratchpad alias is spad and the Tile_loop is loop.

```
C/C++
#- Create structure to hold new types of tile (Needed)
%new_tile;

#- Pair the verilog module with an alias for each new tile (User)
$new_tile{'acc1'} = 'Tile_acc1';
$new_tile{'acc2'} = 'Tile_acc2';

#- Add it to the parameters (Needed)
$param{'new_tile'} = \%new_tile;
```

5.2. Add the accelerators to the array



Similarly to the previous step, use @tile_array to set the type of tile, and @pico_program to load data in the tiles. Set an array with 2 rows and three columns and add acc1 and acc2 in the new column. The testcase looks like this:

```
C/C++
$param{'r'} = 2;
$param{'c'} = 3;

@tile_array = (['pico', 'spad', 'acc1'],
               ['loop', 'pico', 'acc2']);

@pico_program  = ('send_msg32_0.hex', 'test_tile_nop.hex', '',
                                      '', 'send_msg32_9.hex',  '');
```

5.3. Add the source files

For icarus open the icarus/file_list.txt and add the verilog source files

```
C/C++
 ../doc/tutorial_files/acc1_tile/Tile_acc1.sv
 ../doc/tutorial_files/acc1_tile/acc_acc1.sv

 ../doc/tutorial_files/acc2_tile/Tile_acc2.sv
 ../doc/tutorial_files/acc2_tile/acc_acc2.sv
```

5.4. Create the stimulus

The tile_ids for these example are:

	Column 0	Column 1	Column 2
Row 0	000-000 (6'h0)	001-000 (6'h8)	010-000 (6'h10)(6'd16)
Row 1	000-001 (6'h1)	001-001 (6'h9)	010-001 (6'h11) (6'd17)

There are several ways to send/receive packets from the RISCV to the different tiles. We can use the qPut function as explained before. Also, some accelerators benefit from using long packets.

5.4.1. Long packets using qPut

The code below is an example of how to create a packet with a 64 bits header (two 32bits words) and 8 words (256 bits) in the payload. Use qPutH(destination_tile, packet_size_code) to create the packet header. Use qPutD(data1, data2) to add data to the payload 64 bits at a time (2 words of 32 bits).

```
C/C++
qPutH(dest_tile,3); //qPutH(dest,pkt_size), 2^3=8 words in payload
for (int i=0; i<4; i=i+1){
  qPutD(data1, data2);
  data1 = data1 + 1;
  data2 = data2 + 1;
}
```

5.4.2. Non-blocking memory Put (mPut)

Set the dest_tile and the offset within the tile to write to a memory mapped location. For example, in the scratchpad tile this writes to address 1024 in the sram.

```
C/C++
/* w1_mPut(data, dest_tile_id, offset) */
w1_mPut(data1, dest_tile, 1024);
```

Let the compiler decide where the data is. Define the variables and assign them to the accelerator address space.

```
C/C++
uint32_t data2_acc1 __attribute__ ((section(".myacc1")));
...
w2_mPut(data1,&data2_acc1);
```

5.4.3. Store (blocking)

Just assign a value to a variable that was defined in the accelerator address space.

C/C++

```
...
uint32_t data1_acc1 __attribute__ ((section(".myacc1")));
...
data1_acc1 = 0xbeefbeef
```

6. Vivado

1. Add vivado related settings in the testcase. See an example in doc/tutorial_files/testcases/mosaic_tutorial_4.pl. The parameter `vivado` indicates the generation tool that we will be using vivado. `vivado_project` tells the generation tools to create a vivado project add all the directories, files and execute the scripts required for MoSACI. `board` tells which FPGA board to use. Set it to u250 if working on Wanda or BXE, and to u280 if working on Peter.

C/C++

```
$param{'vivado'}         = 1;
$param{'vivado_project'} = 1;
$param{'board'}          = 'u250';  #- u250 vs u280
```

2. For Iverilog we got away with adding the files manually in `file_list.txt`. For Vivado, we assume all the files are in the src and build directories. Thus, copy the new accelerators files to /src/Tile.HDL.

Unset

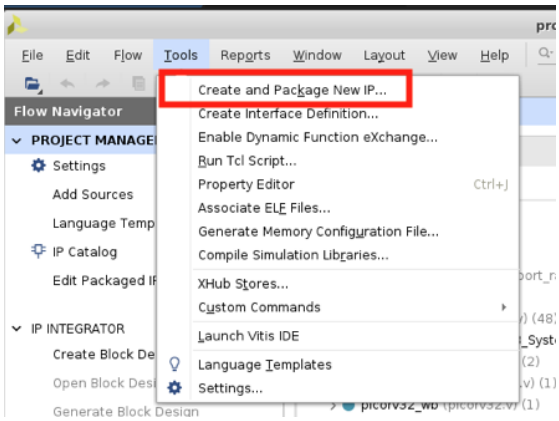
```
cp -r doc/tutorial_files/acc1_tile src/Tile.HDL/
cp -r doc/tutorial_files/acc2_tile src/Tile.HDL/
```

3. As before, execute the testcase.
4. After Vivado has finished the simulation, open a Vivado gui and open the freshly created project “mosaic” located in the vivado directory.
5. Set the testbench as top module.
6. Run a behavioral simulation

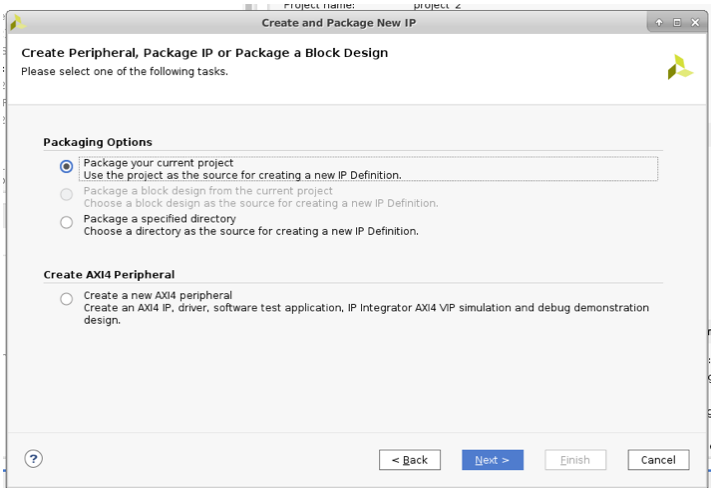
7. Add waves to the waveform viewer
8. Run a simulation and see what happens

6.1. Package MoSAIC

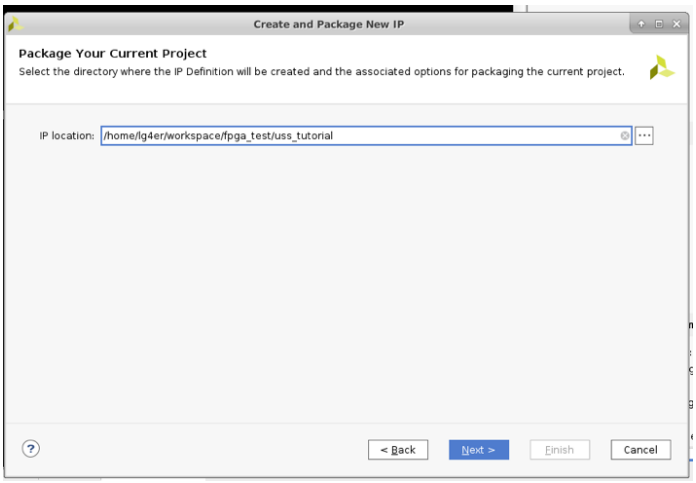
1. Set mosaic (instead of tb_mosaic) as the top module.
2. Go to Tools-> Create and package new IP



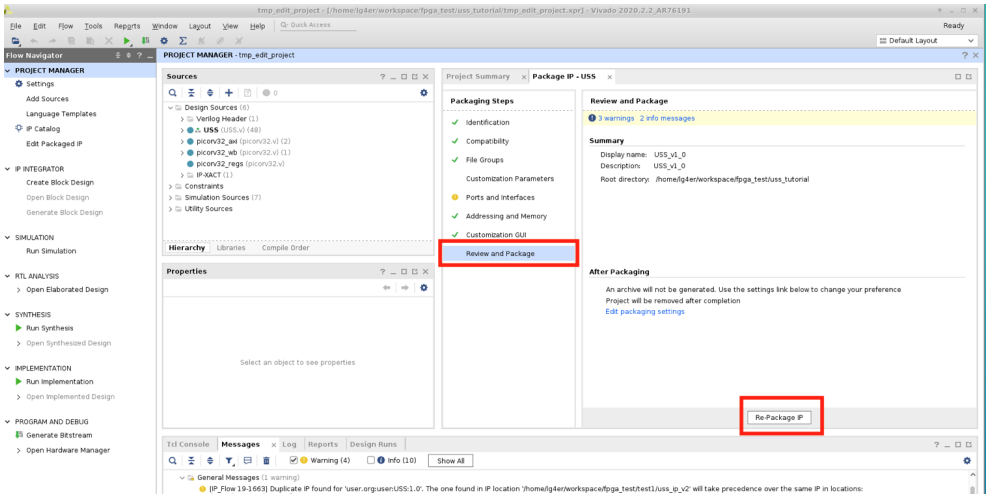
Follow the wizard until this window and choose package your current project.



Click next and give the package a name. mosaic_tutorail_ip



Click Ok on the next pop-up window and then click finish to continue. A new temporal vivado project will open. In the new Vivado window click in Review and Package, and then in package (re-package) IP.



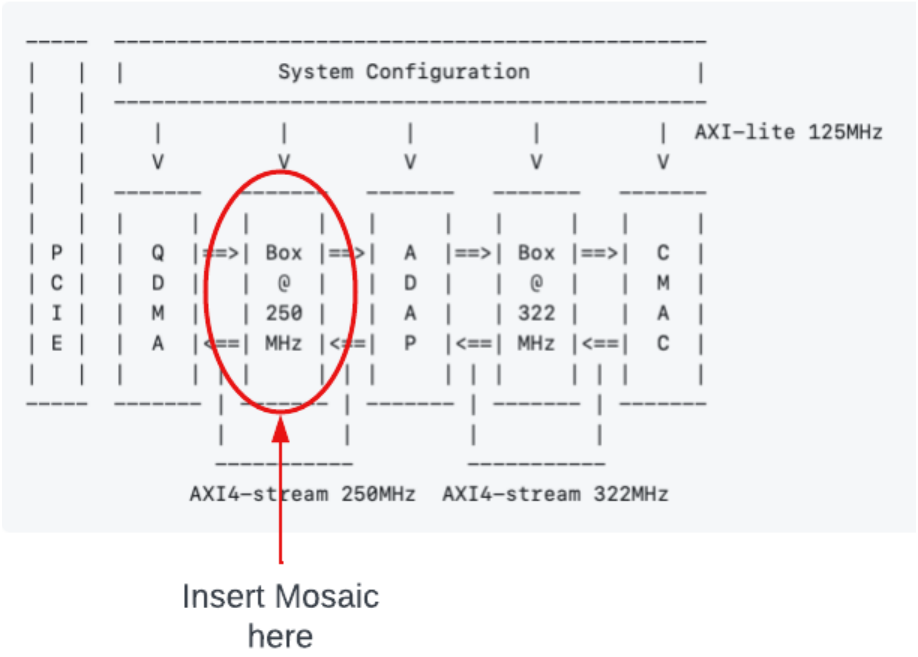
A pop-up window will show up and ask you to close the project. Say yes and that's it!

7. Open-NIC-Shell

The description below from their repository. In red highlighted what we have experimented as part of MoSAIC

OpenNIC shell delivers an FPGA-based NIC shell with 100Gbps Ethernet ports. The latest version is built with Vivado ***2020.x, 2021.x or 2022.1****. Currently, the supported boards include:*

- Xilinx Alveo U50, and
- Xilinx Alveo U55N, and
- Xilinx Alveo U55C, and
- Xilinx Alveo U200, and
- **Xilinx Alveo U250, and**
- **Xilinx Alveo U280, and**
- Xilinx Alveo U45N



7.1. Getting started

1. Clone open-nic-shell from their repository:

Unset

```
https://github.com/Xilinx/open-nic-shell
```

2. Source the configuration script to setup Vivado. In peter and wanda use

```
Unset

“source /opt/source-vitis-2022.2.sh”
```

3. Build open-nic-shell following their github instructions. For this tutorial, use the default features.

```
Unset
#Wanda or BXE
vivado -mode batch -source build.tcl -tclargs -board au250
#Peter
vivado -mode batch -source build.tcl -tclargs -board au280
```

Comming soon:

Building open-nic-shell takes some time. Leave it there and come back when it is done…

1. Apply MoSAIC patch
2. Add the packaged version of MoSAIC
3. Run synthesis and generate bitstream.
4. Load bitstream in FPGA
5. Load hex files
6. Reset the RISCVs and accelerators in the tiles
7. Get the results