AIE Programming Guide

By

System View Inc.

Table of Contents

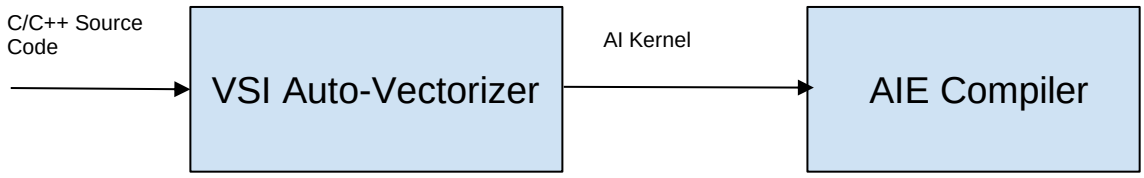
1- Auto-Vectorizer: C/C++ programming guide 4
Auto-Vectorizer: Design flow 4
Auto-Vectorizer: Required Modifications 19
1. Source code top function return type and input/output arguments 19
2. Conditional statement vectorization. 20
3. Inner loop function vectorization 21
4. sqrt 22
5. Not supported types 22
6. manual unrolling 22
7. initialization of complex type 23
8. Inlining functions in the top level function 23
Auto-Vectorizer: Features 23
1. VSI Complex Operations 24
2. Stream support 25
3. Fast Inv Sqrt 29
4. VSI Trigonometric Flags 29
5. Partial Vectorization 30
2- AIE parameter buffers: 33
VSI Approach: 34
1. Source code modifications: 34
2. GUI modifications: 34
3- RunTime Parameters in VSI 37
Supported RunTime Parameter Features in VSI 37
VSI Approach 37
4- Power Estimation 41
VSI Approach 41
5- Debugging 42
Co-simulation 42
Running AIE simulator (for AIE kernels only) 44
..... 44
Running X86 simulator (for AIE Kernels only) 44
..... 45
Running X86 simulator (for AIE + RTL) 45
6- Performance Analysis 54

Report Summary 54
Trace Report..... 55
Features of Trace Report 56

1- Auto-Vectorizer: C/C++ programming guide

Auto-Vectorizer: Design flow

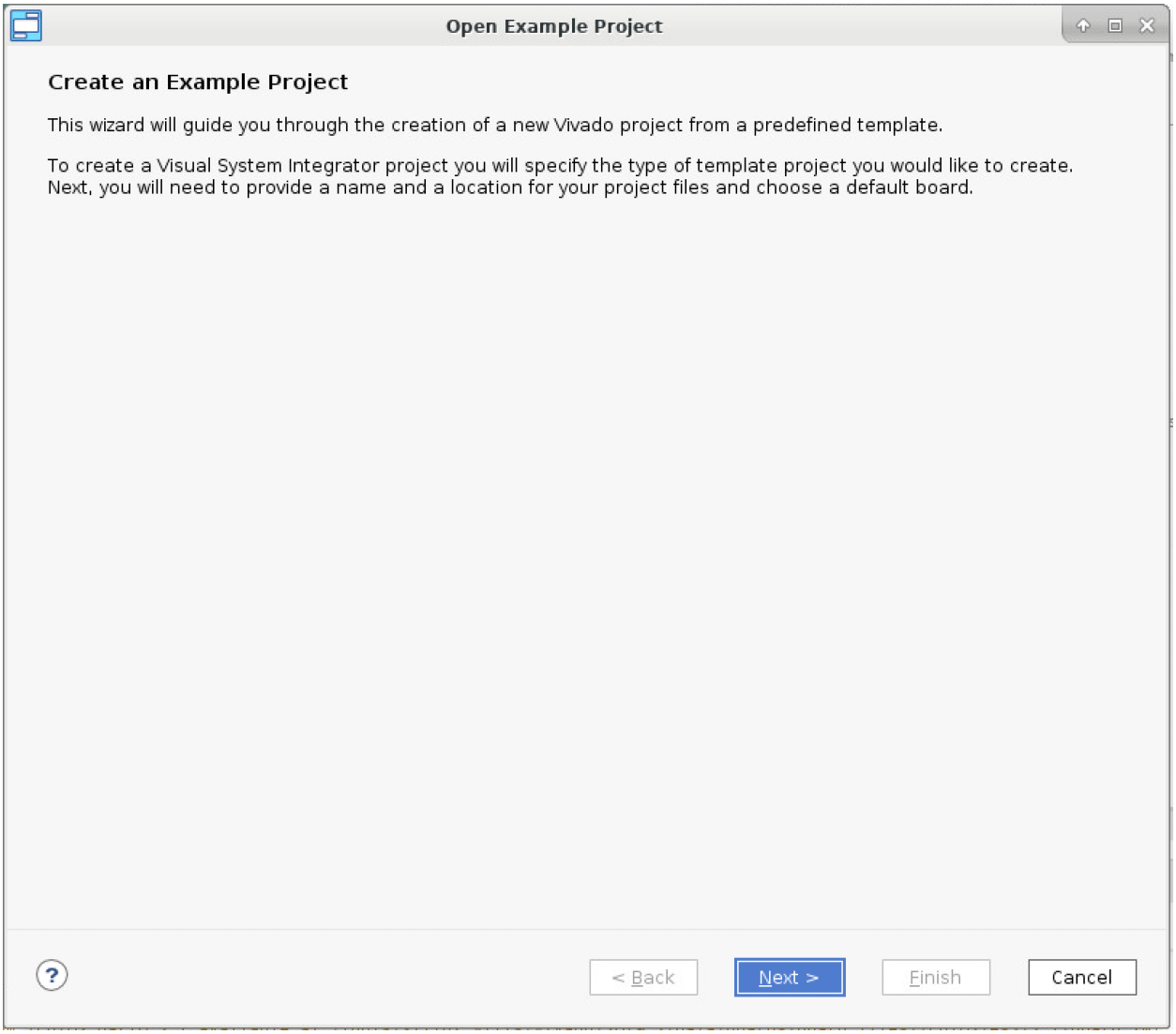
VSI Auto-Vectorizer generates AI kernel from C/C++ source code and all necessary host code. After generating AI Kernel and host code Xilinx AIE compiler compiles it and generates the executables to run in AI Engine. In the compilation flow, VSI auto-vectorizer sits in front of the AIE Compiler. Following figure shows the compilation flow:



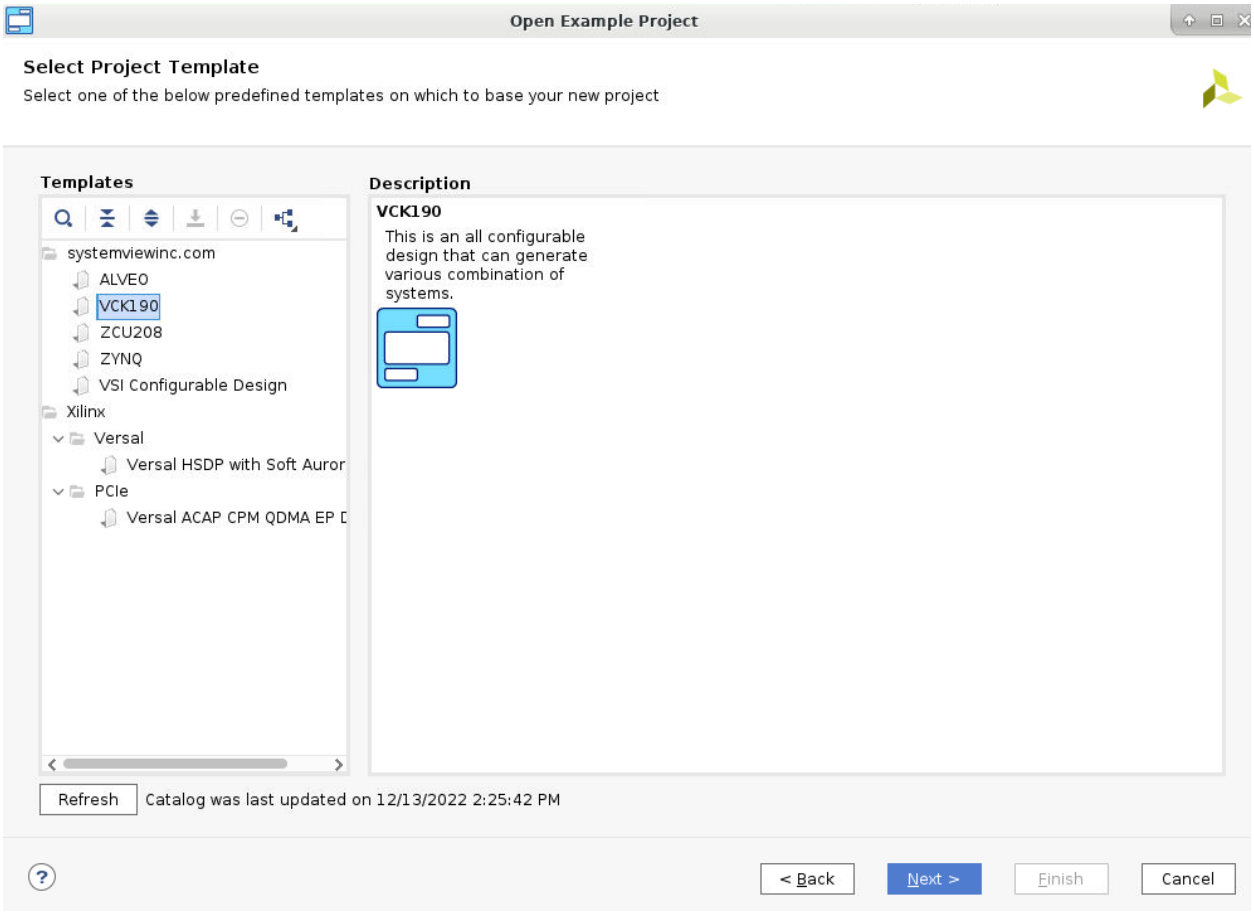
Following steps shows creating a versal platform ,versal system and AIE Vectorized code.

1. Start the VSI tool:
    a. Open a Command Terminal and type: **vsi**
2. Under “Quick Start”, select “Open Example Project”.

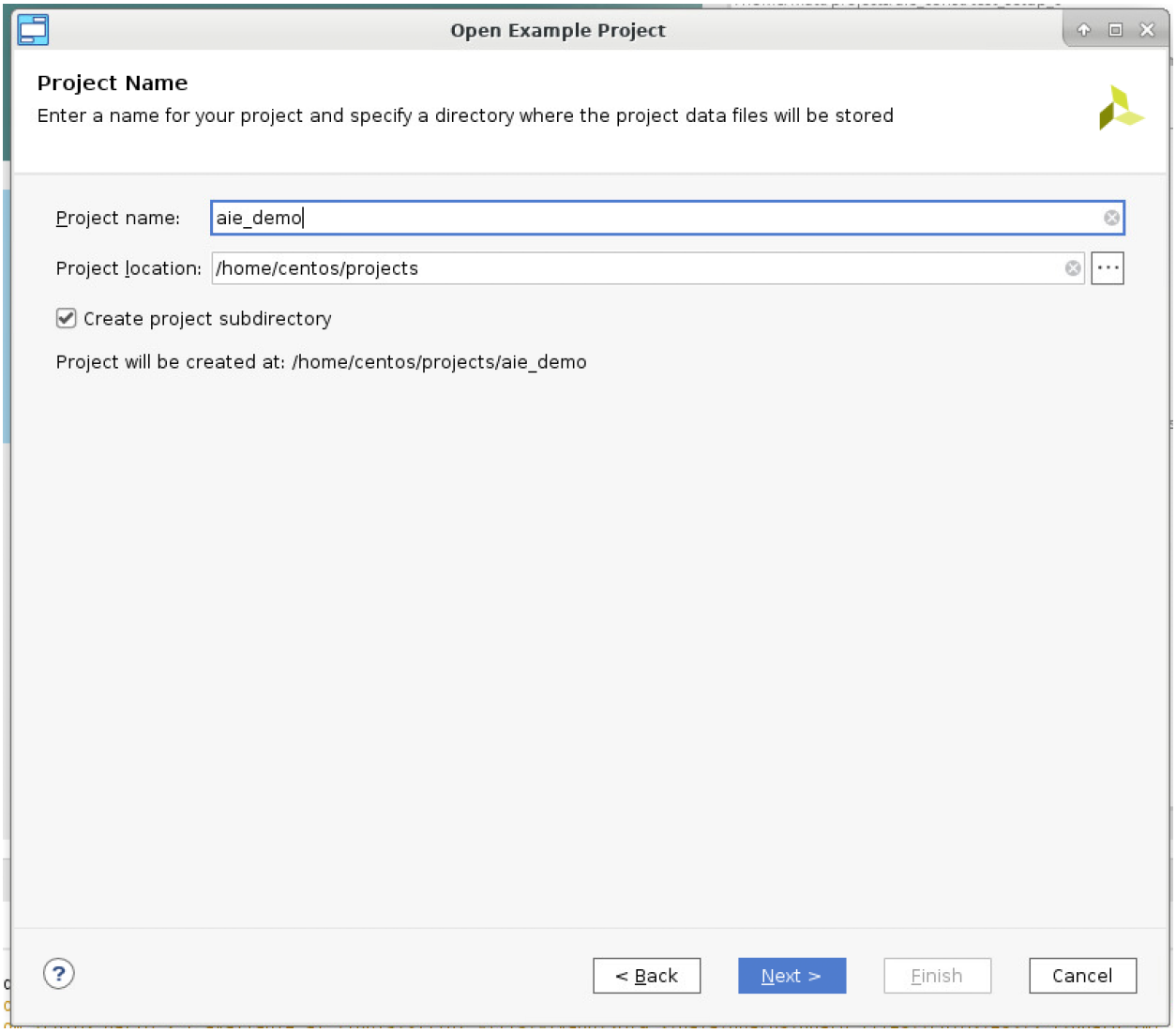
    a. Under “Create an Example Project”, select Next.



b. Under “Select Project Template”, Select VCK190 and Select Next.

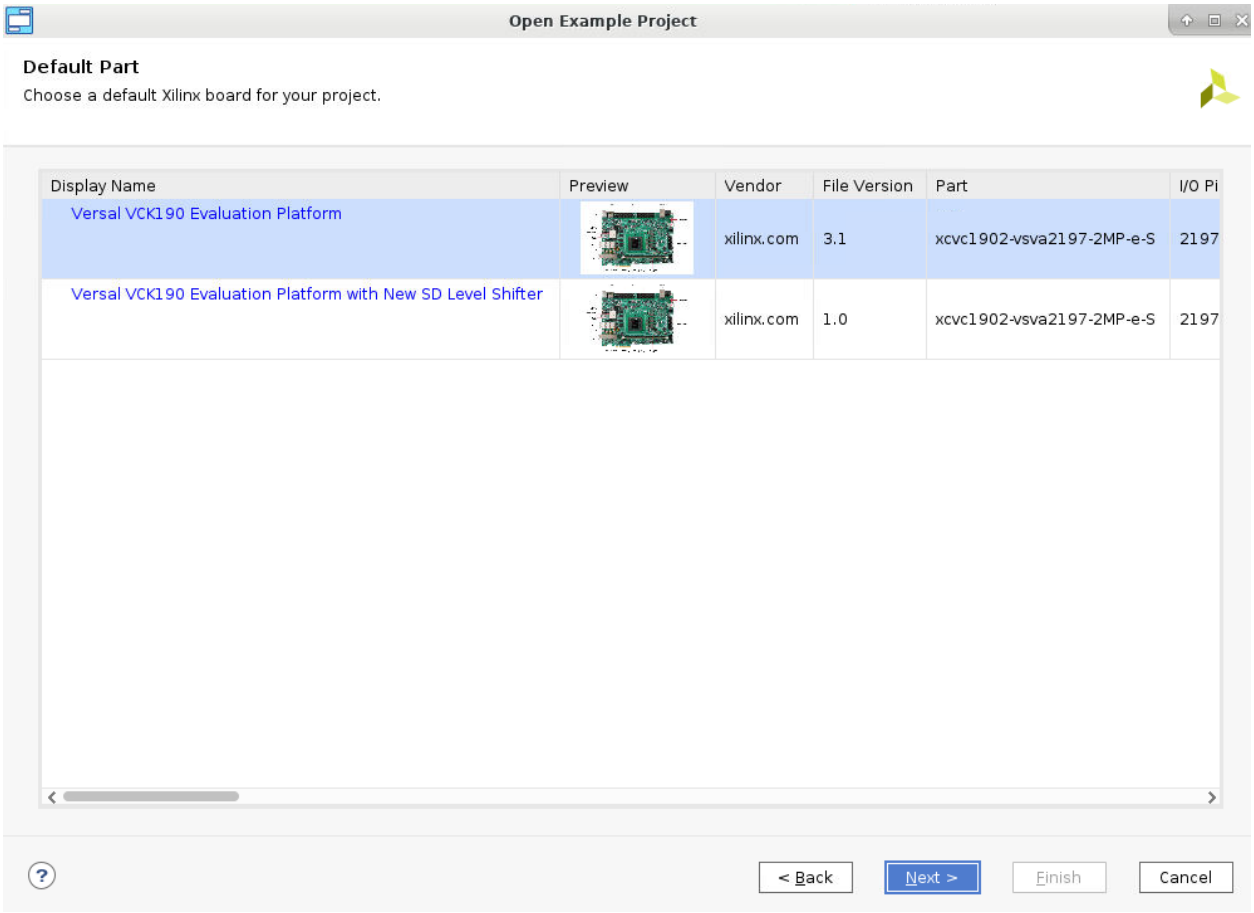


c. Under the “Project Name” set the name and location for the example project:
   i. Set the “Project name” to “aie_demo”
   ii. Set the “Project location” to “/home/centos/projects”
   iii. If not already selected, check “Create project subdirectory”

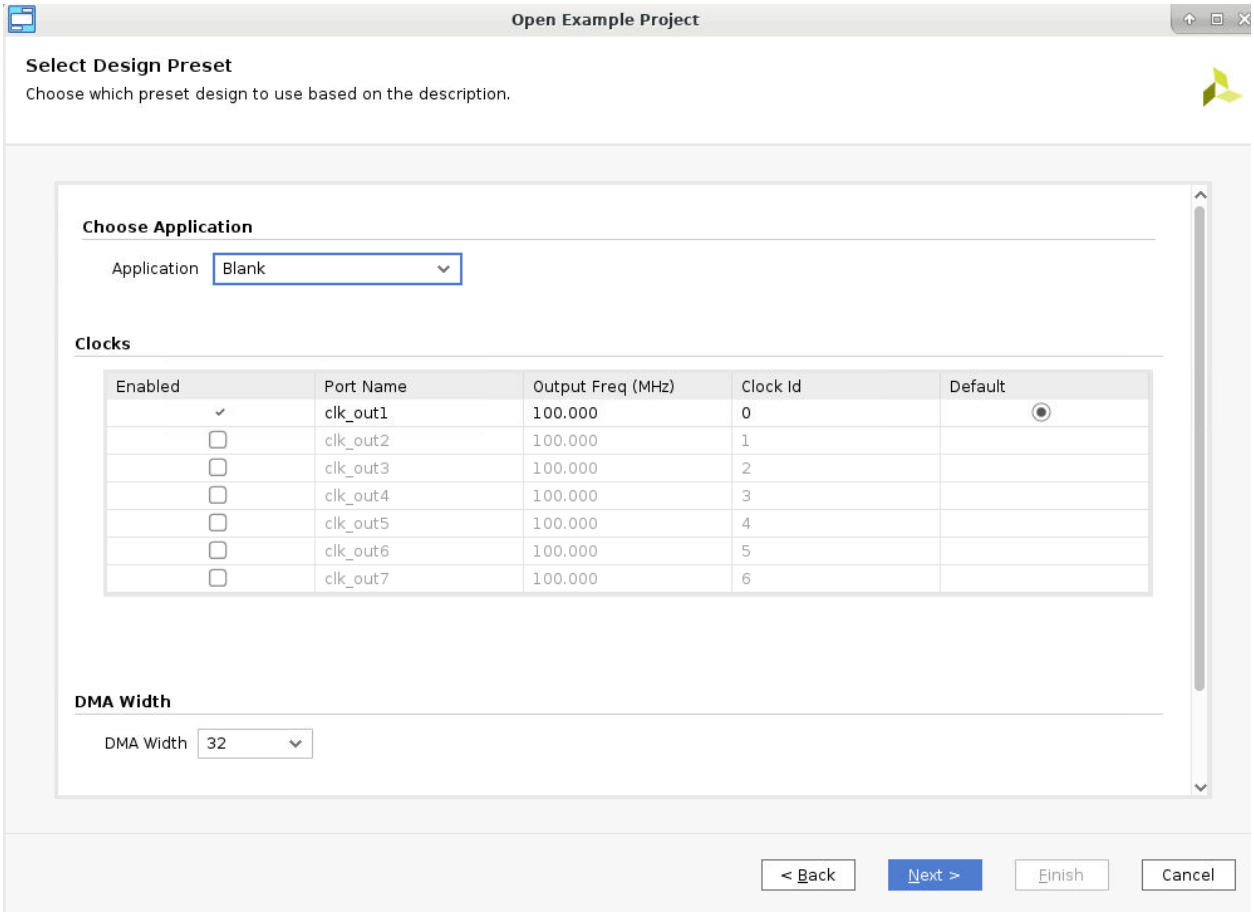


e. Under the “Default Part” window, select Next.

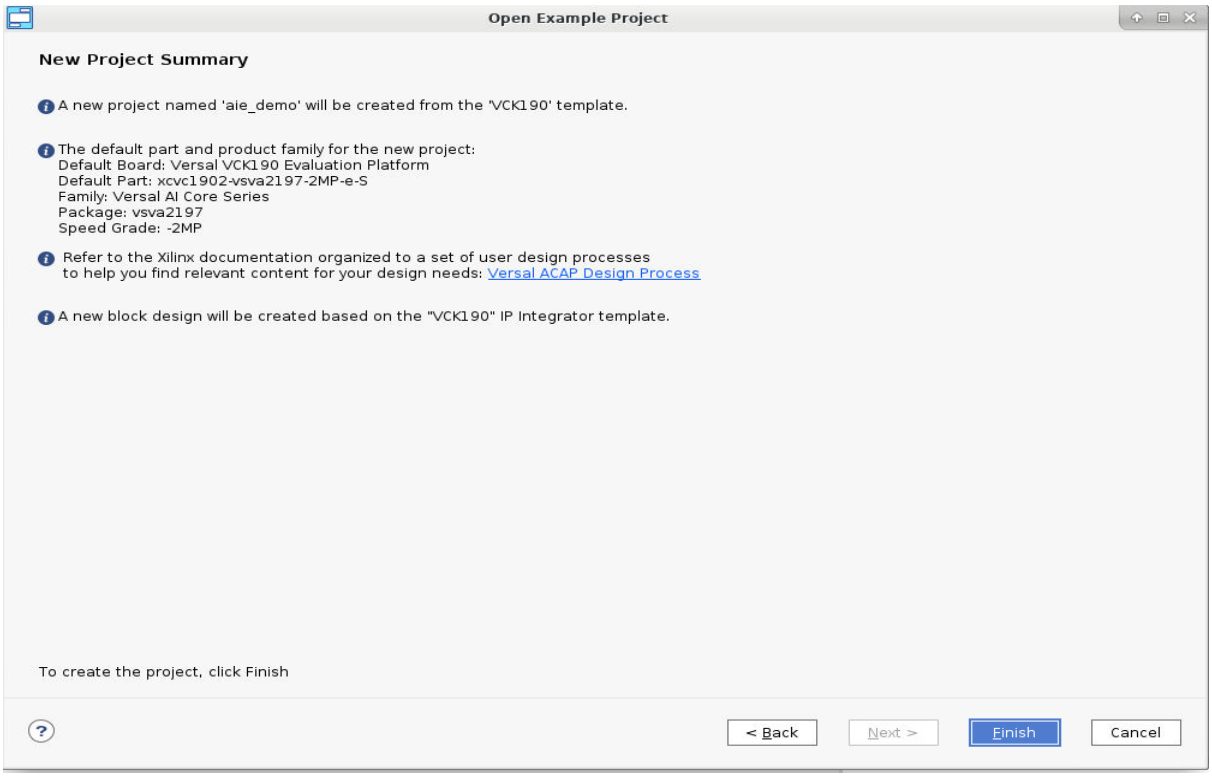
i. In this example, the part is automatically set in the platform, so selecting the part is not necessary.



f. Under “Configure Project” select a platform and application:
    i. VCK190 is the board used for this tutorial. VSI has an example platform for this board already created. In this example our entire application will be built to run within the vck190 board.
    ii. Leave the Application blank. The application will be built at a later time.
    iii. Also leave the Clock, Data Width and Number of DMAs to their default values for this example

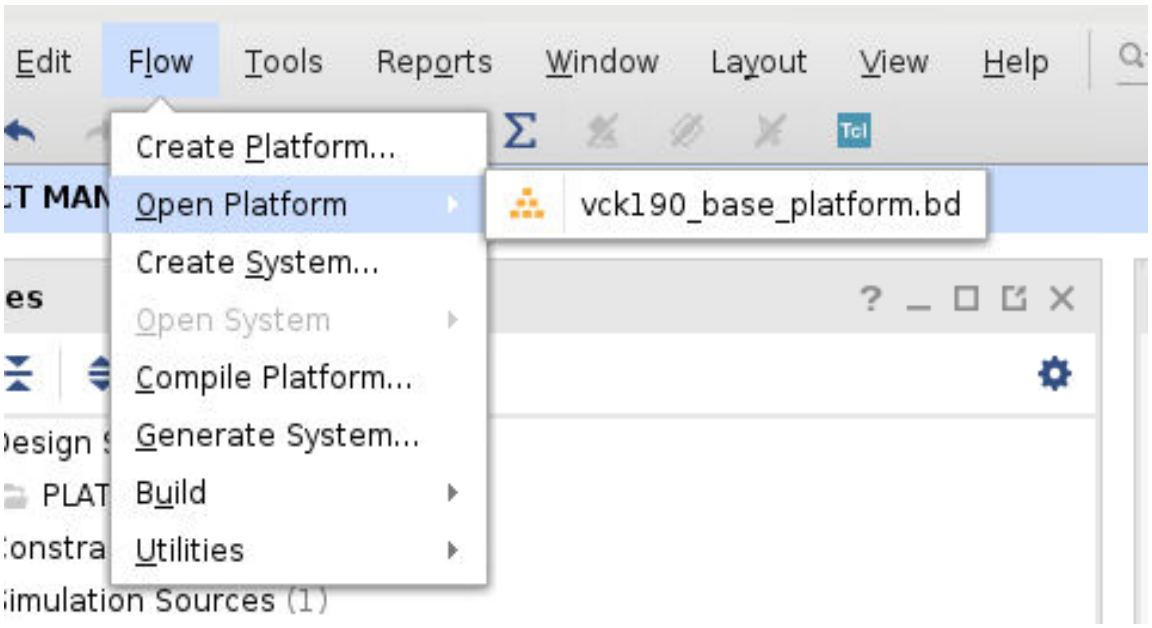


g. Under “New Project Summary”, select Finish.

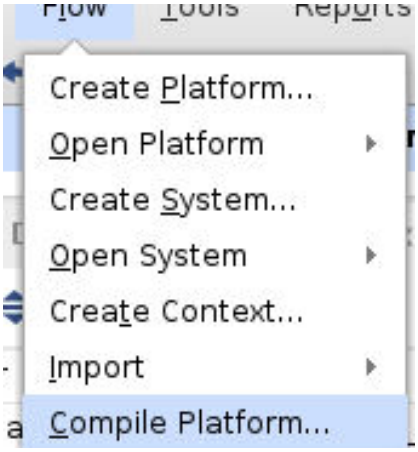


3. Open the Platform:

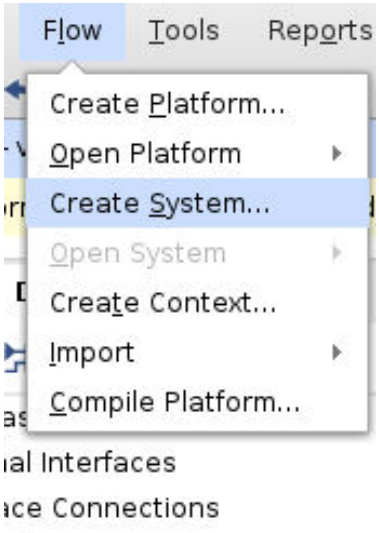
From the menu bar select: Flow -> Open Platform -> vck190_base_platform.bd



4. Compile the Platform:
    i. From the menu bar select: Flow -> Compile Platform -> Select Ok

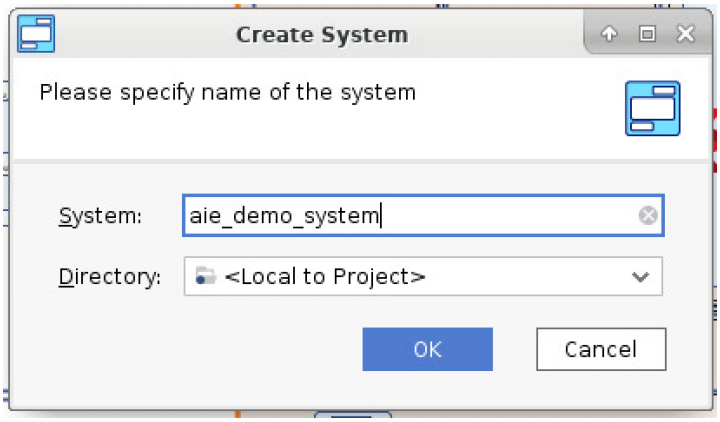


5. Create the System:
    a. From the menu bar select: Flow -> Create System



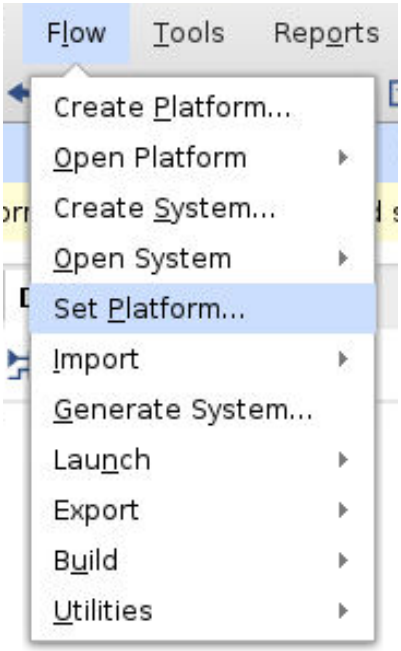
6. In the Create System window:
    b. Set the System name to “aie_demo_system”

c. Leave the directory “<Local to Project>”
d. Select “OK”

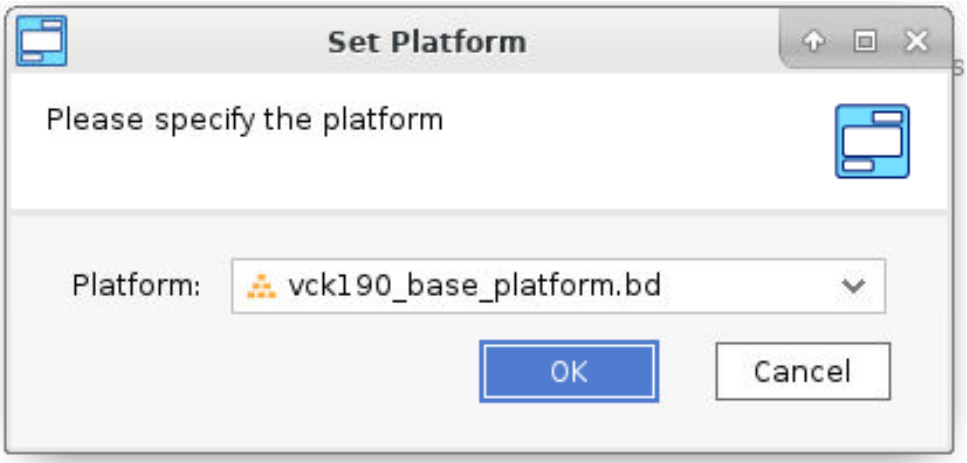


Import the platform created:

e. From the menu bar select: Flow -> Set Platform... -> Select Ok

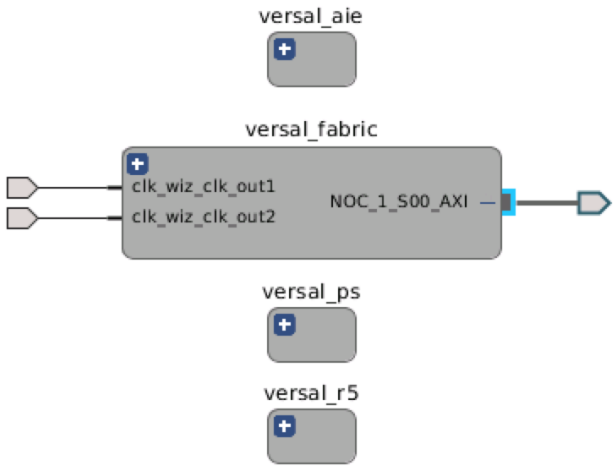


f. Set the “Platform” as “vck190_base_platform.bd”
g. Select “OK”



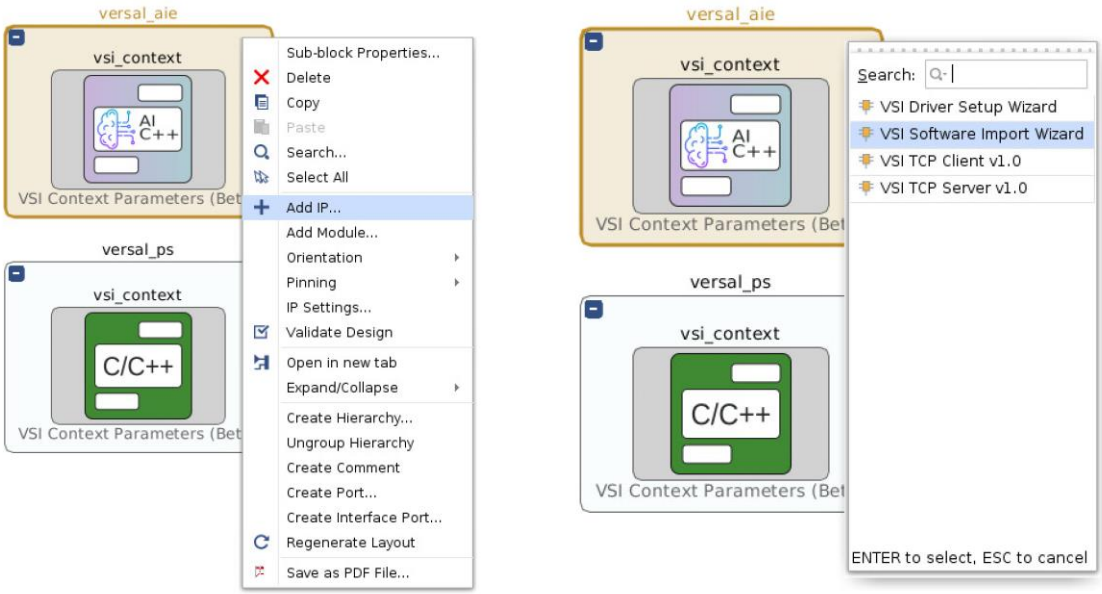
h. Importing the platform created the execution contexts in the system canvas screen, and it also tells VSI which meta data file to read from when generating the system.

After importing the platform there will be four execution contexts in the system, each one representing an execution context from the platform.

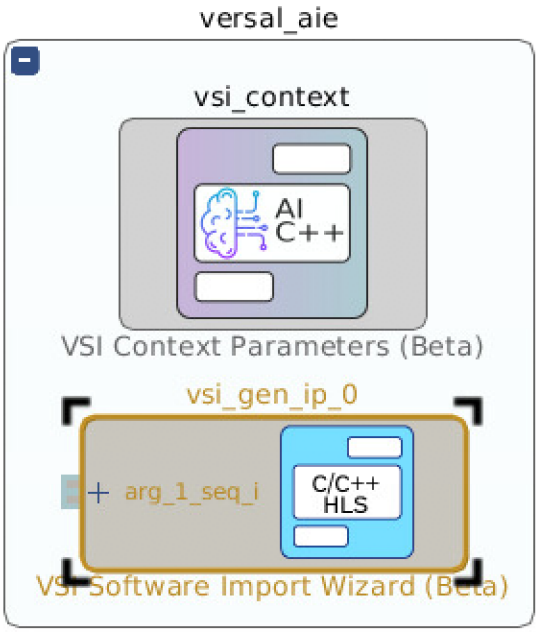


7. We will now start building our system. First we will import some of our code into the AI engines

a. Click on the “versal_aie” hierarchy context
b. Within the “versal_aie” hierarchy context right-click and select “Add IP...”
c. Select “VSI Software Import Wizard”

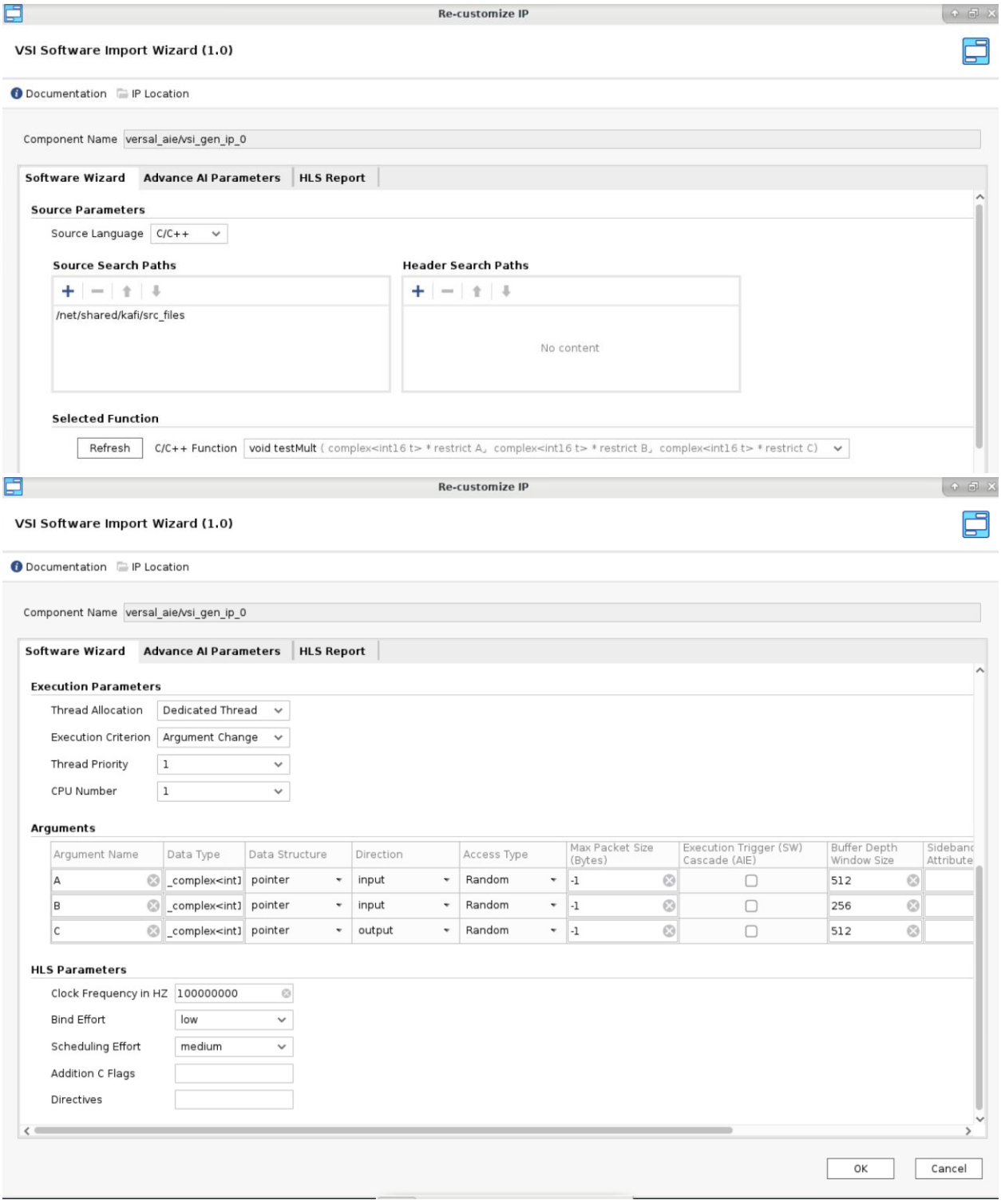


8. Double click on the created block to configure the software import wizard

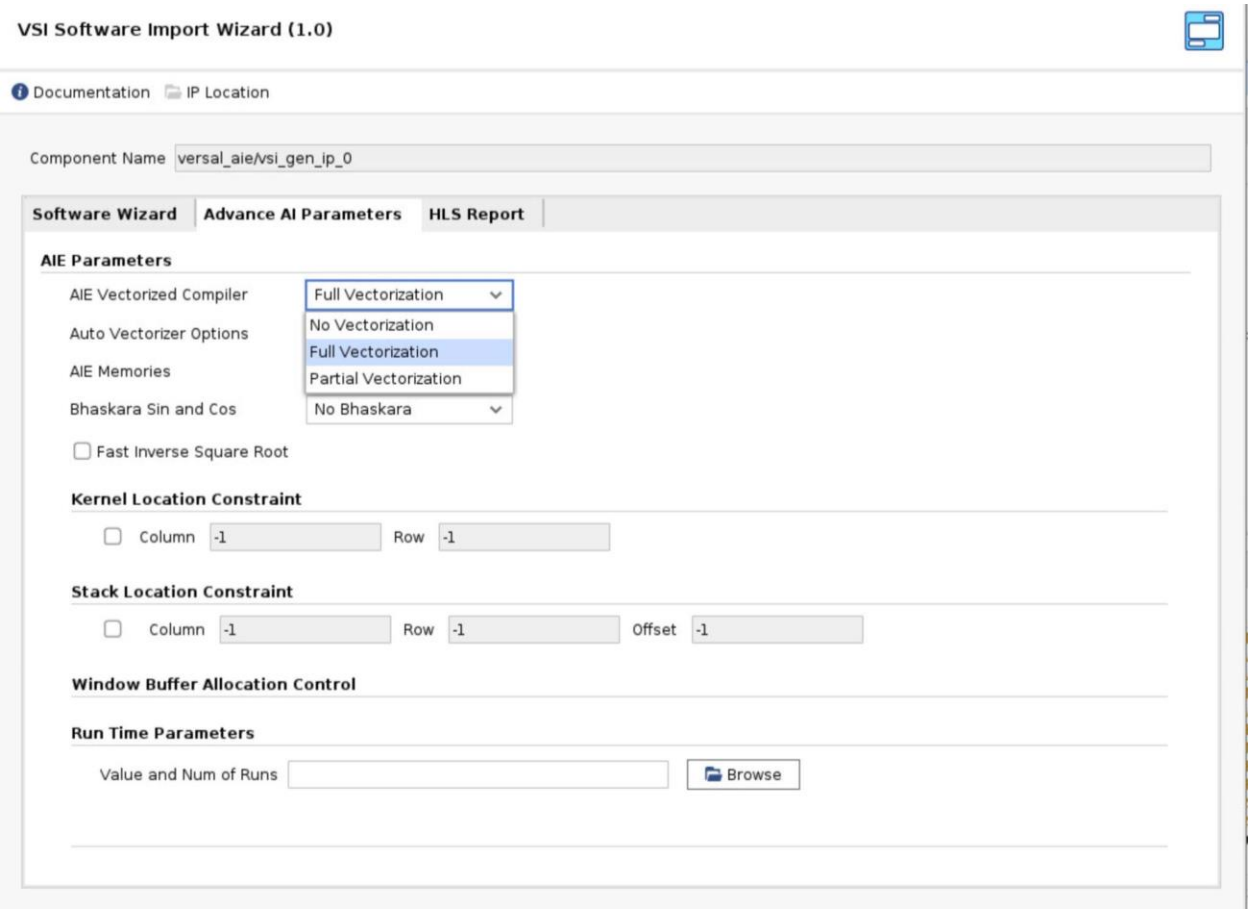


9. The import software wizard is a powerful IP that allows users to import their software from any source file into the system. Within the “Software Import Wizard”:

a. Under Source Search Paths, click "+" to set the “Source Directory”
b. Set C/C++ Function that needs to vectorize.
c. Set the direction of input/output
d. Set access type: “**random**” for “**window type**” and “**sequential**” for “**stream type**”
e. For window type set “Buffer depth window size” to the size of your window buffer in Bytes.

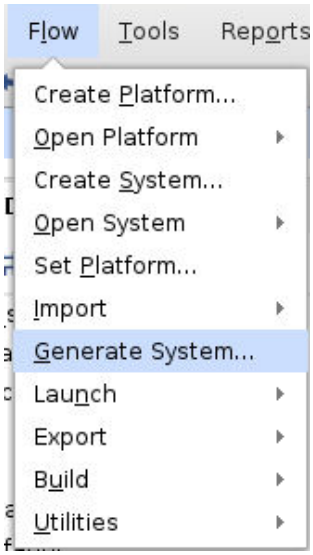

Re-customize IP
VSI Software Import Wizard (1.0)
Documentation
IP Location
Component Name
versal_aie/vsi_gen_ip_0
Software Wizard
Advance AI Parameters
HLS Report
Source Parameters
Source Language
C/C++
Source Search Paths
/net/shared/kafi/src_files
Header Search Paths
No content
Selected Function
Refresh
C/C++ Function
void testMult (complex<int16 t> * restrict A, complex<int16 t> * restrict B, complex<int16 t> * restrict C)
Re-customize IP
VSI Software Import Wizard (1.0)
Documentation
IP Location
Component Name
versal_aie/vsi_gen_ip_0
Software Wizard
Advance AI Parameters
HLS Report
Execution Parameters
Thread Allocation
Dedicated Thread
Execution Criterion
Argument Change
Thread Priority
1
CPU Number
1
Arguments
Argument Name
Data Type
Data Structure
Direction
Access Type
Max Packet Size (Bytes)
Execution Trigger (SW) Cascade (AIE)
Buffer Depth Window Size
A
_complex<int1
pointer
input
Random
-1
512
B
_complex<int1
pointer
input
Random
-1
256
C
_complex<int1
pointer
output
Random
-1
512
HLS Parameters
Clock Frequency in HZ
100000000
Bind Effort
low
Scheduling Effort
medium
Addition C Flags
Directives
OK
Cancel

f. Click the Advance AI parameter tab and select the “Full Vectorization” option as the “AIE Vectorized Compiler” (See the “Partial Vectorization” section for more information on partial vectorizing case)



g. Select “OK”

h. From the menu bar select: Flow -> Generate System -> Select OK

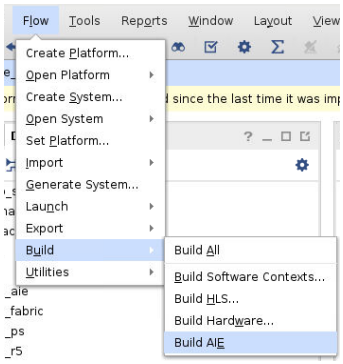


Generate System will generate the AIE vectorized code for the project .

```
vsi_gen: INFO(67) FPGA Device found block type  IOB @ location X=1 Y=0
vsi_gen: INFO(67) FPGA Device found block type  IOB @ location X=2 Y=0
vsi_gen: INFO(67) FPGA Device found block type  IOB @ location X=3 Y=0
vsi_gen: INFO(67) FPGA Device found block type  IOB @ location X=4 Y=0
vsi_gen: INFO(67) FPGA Device found block type  IOB @ location X=5 Y=0
vsi_gen: INFO(67) FPGA Device found block type  IOB @ location X=6 Y=0
vsi_gen: INFO(67) FPGA Device found block type  IOB @ location X=7 Y=0
vsi_gen: INFO(67) FPGA Device found block type  IOB @ location X=8 Y=0
vsi_gen: INFO(67) FPGA Device found block type  IOB @ location X=9 Y=0
vsi_gen: INFO(67) FPGA Device found block type  IOB @ location X=10 Y=0
vsi_gen: INFO(67) FPGA Device found block type  IOB @ location X=11 Y=0
vsi_gen: INFO(67) FPGA Device found block type  IOB @ location X=11 Y=0
vsi_gen: WARNING(51) VALIDATE: duplicate block names found:  {/vsi_platform/versal_aie/vsi_common_driver_0} vsi_common_driver_0: /vsi_platform/versal_ps
vsi_gen: WARNING(11) System has no blocks in 'Hardware' contexts 'versal_fabric'
vsi_gen: 2021-06-14 17:10:37,208 - soa.PathWalker - INFO - processing templates at path = /home/kafi/projects/vsi/staging/common/templates
vsi_gen: 2021-06-14 17:10:38,052 - soa.PathWalker - INFO - generating: /home/kafi/vsi_test/project_10/vsi_auto_gen/sw/system_1/versal_aie/src/versal_aie.cxx | Duration time: 0.157 [s]
vsi_gen: 2021-06-14 17:10:38,096 - soa.PathWalker - INFO - generating: /home/kafi/vsi_test/project_10/vsi_auto_gen/hls/system_1/Makefile | Duration time: 0.000 [s]
vsi_gen: 2021-06-14 17:10:38,097 - soa.PathWalker - INFO - processing templates at path = /home/kafi/vsi_test/project_10/vsi_auto_gen/.SystemView/soat/templates

Generating AIE vectorized code for /versal_aie/vsi_gen_ip_0 block:

vsi_vectorizing: /net/shared/kafi/src_files/vector_i16_test.cc:12:2: remark: the cost-model indicates that interleaving is not beneficial [-Rpass-analysis=loop-vectorize]
vsi_vectorizing:         for (int i = 0; i < 1024; i++) {
vsi_vectorizing:         ^
vsi_vectorizing: /net/shared/kafi/src_files/vector_i16_test.cc:12:2: remark: vectorized loop (vectorization width: 8, interleaved count: 1) [-Rpass-loop-vectorize]
vsi_AIE_backend: License(valid):
vsi_AIE_backend:    company: "System View"
vsi_AIE_backend:    expiry_date: Wed Dec 19 00:00:00 2029 PST
vsi_AIE_backend:    host_id: "7085C2DAA0E9"
vsi_AIE_backend:    name: "Sandeep"
vsi_AIE_backend: File: vsi.lic.7085C2DAA0E9

AIE vectorized code for /versal_aie/vsi_gen_ip_0 is generated in /home/kafi/vsi_test/project_10/vsi_auto_gen/aie_vectorized/vsi_gen_ip_0

1
```

If you like to compile and simulate your code, please continue the following steps:

d. From the menu bar select: Flow -> Build -> Build AIE



e. After it build successfully, see the "AIE Simulator" section for simulating your code with aiesimulator.

Auto-Vectorizer: Required Modifications

1. Source code top function return type and input/output arguments

The auto-vectorizer C/C++ source code top function should be void type and inputs/outputs are pointer type. In order to generate optimized code, users should use __**restrict**__ keyword in inputs/outputs pointer type declaration.

Original code-

```
float * vector_test(float * A, float * B) {

	float * tmp = new float[64]

	#pragma clang loop vectorize(enable)
	for (int i = 0 ; i < 64; i++) {
		tmp[i] = A[i] * B[i];
	}

	return tmp;
}
```

Modified code-

```
void vector_test(float * __restrict__ A, float * __restrict__ B, float *
__restrict__ C) {

	#pragma clang loop vectorize(enable)
	for (int i = 0 ; i < 64; i++) {
		C[i] = A[i] * B[i];
	}
}
```

2. Conditional statement vectorization.

In order to vectorize the conditional statement and help auto-vectorizer generate optimized code. Users need to do following modifications:

Code example:

Original code-

```
void vector_i16_test(int32_t * __restrict__ A, int32_t * __restrict__ B,
int32_t * __restrict__ C) {

	#pragma clang loop vectorize(enable)
	for (int i = 0 ; i < 64; i++) {
		if(A[i] > 32)
			C[i] = B[i] * 4;
		else
			C[i] = 0;
	}
}
```

Modified code-

```
void vector_i16_test(int32_t * __restrict__ A, int32_t * __restrict__ B,
int32_t * __restrict__ C) {

	#pragma clang loop vectorize(enable)
	for (int i = 0 ; i < 64; i++) {
int tmp = B[i]; // the load should be outside the if condition
int tmp_out = 0;
		if(A[i] > 32)
			tmp_out = tmp * 4;

C[i] = tmp_out;
	}
```

```
}
```

3. Inner loop function vectorization

In auto-vectorizer C/C++ source code, if the inner loop has a function call, then the function should be **static inlinable** in order to vectorize the inner loop.

Original code-

```
float sampleFunc( float input_val) {
	if(input > 16)
		return sinf(input_val)*2.0f;
	else
		return cosf(input_val)+6.0f;

}

void vector_test(float * __restrict__ A, float * __restrict__ B, float *
__restrict__ C) {

	#pragma clang loop vectorize(enable)
	for (int i = 0 ; i < 64; i++) {
		C[i] = A[i] * sampleFunc(B[i]);
	}
}
```

Modified code-

```
static inline float sampleFunc( float input_val) {
	if(input > 16)
		return sinf(input_val)*2.0f;
	else
		return cosf(input_val)+6.0f;

}

void vector_test(float * __restrict__ A, float * __restrict__ B, float *
__restrict__ C) {
```

```
        #pragma clang loop vectorize(enable)
        for (int i = 0 ; i < 64; i++) {
              C[i] = A[i] * sampleFunc(B[i]);
        }
  }
```

4. sqrt

In auto-vectorizer C/C++ source code, the **sqrtf** should be used instead of the **sqrt**.

5. Not supported types

Currently, the following types are not supported in the auto-vectorizer:

- double
- long long
- int64_t

6. manual unrolling

If the user unrolls a loop manually, he should use the interleave count 1 pragma at the top of the loop as well:

Original code-

```
      #pragma clang loop vectorize(enable)
      for (k = 0; k < TRAINING_BLOCK_SIZE; k++) {
            for (i = 0; i < N_CHAN*TDOF; ++i) {
               _complex<float> x = snapshot[k*N_CHAN*TDOF+i] *
cconj(snapshot[k*N_CHAN*TDOF+j]);
                  conv_index = i*N_CHAN*TDOF + j;
                  covariance[conv_index] += (x);

      }

   }
```

Modified code-

```
      #pragma clang loop vectorize(enable) interleave_count(1)
      for (k = 0; k < TRAINING_BLOCK_SIZE; k++) {
            …
```

```
        }

    }
```

7. initialization of complex type

To initialize a complex type, use the following method:

```
_complex<float> accum(0.0f,0.0f);
```

Initializing a complex type with separated real and imag, is **not** recommended:

```
accum.real() = 0.0f;
accum.imag() = 0.0f;
```

8. Inlining functions in the top level function

We support “stream” functionality only on the top level function. So if inside the top level function, there is a call for another function with **some of the arguments of the top function**, inlining that function allows us to keep using “streams” without further modification.

The alternative would be to copy the stream argument into a buffer and then call the inner function.

9. The C++ Standard Template Library (STL)

The STL is not supported.

Auto-Vectorizer: Features

1. VSI Complex Operations

Auto vectorizer supports complex arithmetic operations and generates vector complex intrinsic functions/API in AIE Engine kernel. To use the complex type feature, users need to include the "_complex.h" header file in the C/C++ source code.

```
#include "_complex.h"
```

Auto-Vectorizer supports following complex arithmetic operations for float, int32 and int16 types.

- Complex Addition.
- Complex Subtraction.
- Complex Multiplication.
- Complex Multiplication with int/float.
- Complex Multiply-Accumulate.
- Complex Multiply-Subtraction.
- Complex Conjugate.
- Complex Normalization.
- Complex Reduction

Code example:

```
#include "_complex.h"

void complex_float(_complex<float> * __restrict__ A,
                   _complex<float> * __restrict__ B,
                    _complex<float> * __restrict__ C) {

   _complex<float> complex_var(2.0f, 2.5f);
   float float_var = 5.5f;

   #pragma clang loop vectorize(enable)
   for (int i = 0 ; i < 64; i++)
       C[i] =  A[i] + B[i];                    // complex float addition
//     C[i] =  A[i] - B[i];                    // complex float subtraction
//     C[i] =  A[i] * B[i];                    // complex float multiplication
//     C[i] =  A[i] * float_var;               // complex float multiply with float
//     C[i] =  A[i] + B[i] * complex_var;      // complex float multiply-accumulate
//     C[i] =  A[i] - B[i] * complex_var;      // complex float multiply-subtraction
//     C[i] =  A[i] + B[i] * float_var;        // complex mult-accumulate with float
//     C[i] =  A[i] - B[i] * float_var;        // complex mult-subtraction with float
//     C[i] =  cconj(A[i]+B[i]);               // complex conjugate
//     C[i] =  cnorm(A[i]+B[i]);               // complex normalization
//     C[0] += (A[i] + B[i]);                  // complex reduction
//     C[0] -= (A[i] + B[i]);                  // complex reduction
}
void complex_int32(_complex<int32_t> * __restrict__ A, _complex<int32_t> *
__restrict__ B, _complex<int32_t> * __restrict__ C) {

   _complex<int32_t> complex_var(2, 3);
   int32_t int32_var = 5;
```

```
   #pragma clang loop vectorize(enable)
   for (int i = 0 ; i < 64; i++)
       C[i] =  A[i] + B[i];                    // complex int32_t addition
//     C[i] =  A[i] - B[i];                    // complex int32_t subtraction
//     C[i] =  A[i] * B[i];                    // complex int32_t multiplication
//     C[i] =  A[i] * int32_var;               // complex int32 multiply with int32
//     C[i] =  A[i] + B[i] * complex_var;      // complex int32_t multiply-accumulate
//     C[i] =  A[i] - B[i] * complex_var;      // complex int32_t multiply-subtraction
//     C[i] =  A[i] + B[i] * int32_var;        // complex mult-accumulate with int32
//     C[i] =  A[i] - B[i] * int32_var;        // complex mult-subtraction with int32
//     C[i] =  cconj(A[i]+B[i]);               // complex conjugate
//     C[i] =  cnorm(A[i]+B[i]);               // complex normalization
//     C[0] += (A[i] + B[i]);                  // complex reduction
//     C[0] -= (A[i] + B[i]);                  // complex reduction

}


void complex_int16(_complex<int16_t> * __restrict__ A, _complex<int16_t> *
__restrict__ B, _complex<int16_t> * __restrict__ C) {

   _complex<int16_t> complex_var(2, 3);
   int16_t int16_var = 5;

   #pragma clang loop vectorize(enable)
   for (int i = 0 ; i < 64; i++)
       C[i] =  A[i] + B[i];                    // complex int16_t addition
//     C[i] =  A[i] - B[i];                    // complex int16_t subtraction
//     C[i] =  A[i] * B[i];                    // complex int16_t multiplication
//     C[i] =  A[i] * int16_var;               // complex int16_t multiply with float
//     C[i] =  A[i] + B[i] * complex_var;      // complex int16_t multiply-accumulate
//     C[i] =  A[i] - B[i] * complex_var;      // complex int16_t multiply-subtraction
//     C[i] =  A[i] + B[i] * int16_var;        // complex mult-accumulate with int16_t
//     C[i] =  A[i] - B[i] * int16_var;        // complex mult-subtraction with int16_t
//     C[i] =  cconj(A[i]+B[i]);               // complex conjugate
//     C[i] =  cnorm(A[i]+B[i]);               // complex normalization
//     C[0] += (A[i] + B[i]);                  // complex reduction
//     C[0] -= (A[i] + B[i]);                  // complex reduction

}
```

2. Stream support

VSI Auto-vectorizer supports stream arguments for int16, int32, float, cint16, cint32 and cfloat type. In C/C++ source code, users can use top function complex arguments and using VSI auto-vectorizer, users can generate AIE kernels with input/output stream types. Following examples show the generated AIE kernel with stream arguments from C/C++ source code:

Example-1: Source Code:

```
#include "_complex.h"
void stream_test(_complex<int32_t> * __restrict__ A,
                 _complex<float> * __restrict__ B,
                 _complex<int32_t> * __restrict__ C0,
                 _complex<float> * __restrict__ C1) {

   _complex<int32_t> three(3,3);
   _complex<float> two(2.0f,2.0f);

   #pragma clang loop vectorize(enable)
   for (int i = 0 ; i < 64; i++)
       C0[i] =  A[i] + three;

   #pragma clang loop vectorize(enable)
   for (int i = 0 ; i < 64; i++)
       C1[i] =  B[i] + two;
}
```

Generated AIE Kernel Code:

```
void __stream_test(input_stream_cint32 *  restrict  A,
                   input_stream_cfloat *  restrict  B,
                   output_stream_cint32 *  restrict  C0,
                   output_stream_cfloat *  restrict  C1) {
 int64_t call_i;
 v16int32 broadcast_splat58;
 int64_t index;
 int64_t index__PHI_TEMPORARY;
 int64_t index_next;
 double call_i44;
 v16float broadcast_splat72;
 int64_t index63;
 int64_t index63__PHI_TEMPORARY;
 int64_t index_next64;

 call_i =  /*tail*/ vsi_cconstructor(3, 3);
 broadcast_splat58 =
as_v16int32(aie::broadcast<cint32,8>(aie::vector<cint32,8>(as_v8cint32(vsi_complex_upd
_elem(null_v16int32(),0u,call_i))).get(0)));
 index__PHI_TEMPORARY = 0;
for (;;)      /* Syntactic loop 'vector.body prevent chess from unrolling */
chess_unroll_loop(1) chess_loop_count(8) chess_require_pipelining(1)
{
vector_body:
 index = index__PHI_TEMPORARY;
 v16int32 wide_load =
as_v16int32(concat(readincr_v2(A),readincr_v2(A),readincr_v2(A),readincr_v2(A)));
 v16int32 tmp__1 = as_v16int32(aie::add(aie::vector<cint32,
8>(as_v8cint32(wide_load)),aie::vector<cint32, 8>(as_v8cint32(broadcast_splat58))));
 v16int32 stream_store_tmp_0 = tmp__1;
 writeincr_v2(C0, as_v2cint32(ext_v(stream_store_tmp_0, 0)));
 writeincr_v2(C0, as_v2cint32(ext_v(stream_store_tmp_0, 1)));
 writeincr_v2(C0, as_v2cint32(ext_v(stream_store_tmp_0, 2)));
```

```
 writeincr_v2(C0, as_v2cint32(ext_v(stream_store_tmp_0, 3)));

 index_next = ((int64_t)(((int64_t)index) + ((int64_t)8)));
 if (chess_copy(index_next) == 64) {
   break;
 } else {
   index__PHI_TEMPORARY = index_next;   /* for PHI node */
   continue;
 }
 } /* end of syntactic loop 'vector.body' */
vector_ph62:
 call_i44 =  /*tail*/ vsi_cfconstructor(2.000000e+00, 2.000000e+00);
 broadcast_splat72 =
as_v16float(aie::broadcast<cfloat,8>(aie::vector<cfloat,8>(as_v8cfloat(vsi_complex_upd
_elem(null_v16float(),0u,call_i44))).get(0)));
 index63__PHI_TEMPORARY = 0;   /* for PHI node */

for (;;)      /* Syntactic loop 'vector.body59 prevent chess from unrolling */
chess_unroll_loop(1) chess_loop_count(8) chess_require_pipelining(1)
{
vector_body59:
 index63 = index63__PHI_TEMPORARY;
 v16int32 wide_load70 = as_v16int32(concat(readincr_v2((input_stream_cint32
*)B),readincr_v2((input_stream_cint32 *)B),readincr_v2((input_stream_cint32
*)B),readincr_v2((input_stream_cint32 *)B)));
 v16float tmp__2 = as_v16float(aie::add(aie::vector<cfloat,
8>(as_v8cfloat(wide_load70)),aie::vector<cfloat, 8>(as_v8cfloat(broadcast_splat72))));
 v16float stream_store_tmp_1 = tmp__2;
 writeincr_v2((output_stream_cint32 *)C1, as_v2cint32(ext_v(stream_store_tmp_1, 0)));
 writeincr_v2((output_stream_cint32 *)C1, as_v2cint32(ext_v(stream_store_tmp_1, 1)));
 writeincr_v2((output_stream_cint32 *)C1, as_v2cint32(ext_v(stream_store_tmp_1, 2)));
 writeincr_v2((output_stream_cint32 *)C1, as_v2cint32(ext_v(stream_store_tmp_1, 3)));

 index_next64 = ((int64_t)(((int64_t)index63) + ((int64_t)8)));
 if (chess_copy(index_next64) == 64) {
   break;
 } else {
   index63__PHI_TEMPORARY = index_next64;   /* for PHI node */
   continue;
 }
 } /* end of syntactic loop 'vector.body59' */
for_cond_cleanup15:
 return;
}
```

Example-2: Source Code:

```
static int32_t coeff_0[8] =  {1,2,3,4,5,6,7,8};
static float coeff_1[8]   =  {2.0f,3.0f,4.0f,5.0f,6.0f,7.0f,8.0f,9.0f};

void stream_test(int32_t * __restrict__ A, float * __restrict__ B,
          int32_t * __restrict__ C0, float * __restrict__ C1) {

   while(1) {
```

```
#pragma clang loop vectorize(enable)
       for (int i=0 ; i < 8; i++) {
           *C0++ = (*A++ * coeff_0[i]);
           *C1++ = (*B++ * coeff_1[i]);
       }
   }
}
```

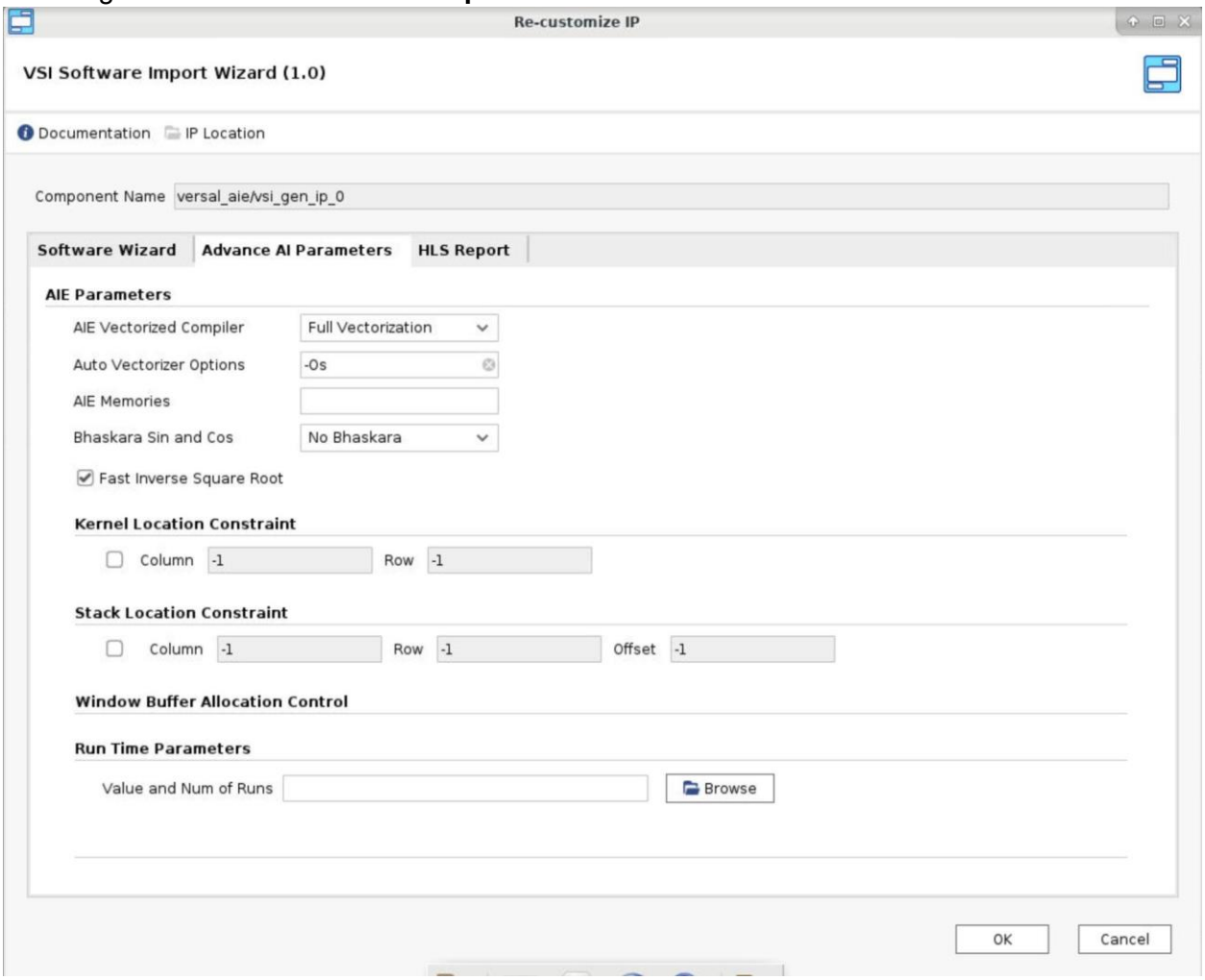
Generated AIE Kernel Code:

```
static int32_t _ZL7coeff_0[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
static float _ZL7coeff_1[8] = { 2.000000e+00, 3.000000e+00, 4.000000e+00,
5.000000e+00, 6.000000e+00, 7.000000e+00, 8.000000e+00, 9.000000e+00 };

void __stream_test(input_stream_int32 *  restrict  A,
                   input_stream_float *  restrict  B,
                   output_stream_int32 *  restrict  C0,
                   output_stream_float *  restrict  C1) {
 int64_t index;
 int64_t index__PHI_TEMPORARY;

for (;;)      /* Syntactic loop 'vector.ph prevent chess from unrolling */
chess_unroll_loop(1)
{
vector_ph:
 index__PHI_TEMPORARY = 0;   /* for PHI node */

for (;;)      /* Syntactic loop 'vector.body prevent chess from unrolling */
chess_unroll_loop(1)
{
vector_body:
 index = index__PHI_TEMPORARY;

 v8int32 wide_load = concat(readincr_v4(A),readincr_v4(A));
 v8int32 wide_load26 = *((v8int32*)(&_ZL7coeff_0[((int64_t)index)]));
 v8int32 stream_store_tmp_0 = srs(mul(wide_load26 , wide_load),0);
 writeincr_v4(C0, ext_v(stream_store_tmp_0, 0));
 writeincr_v4(C0, ext_v(stream_store_tmp_0, 1));

 v8float wide_load27 = concat(readincr_v4(B),readincr_v4(B));
 v8float wide_load28 = *((v8float*)(&_ZL7coeff_1[((int64_t)index)]));
 v8float stream_store_tmp_1 =
as_v8float(aie::mul(aie::vector<float,8>(wide_load28),aie::vector<float,8>(wide_load27
)));
 writeincr_v4(C1, ext_v(stream_store_tmp_1, 0));
 writeincr_v4(C1, ext_v(stream_store_tmp_1, 1));

 if (chess_copy(index) == 0) {
   goto vector_ph;
 } else {
   index__PHI_TEMPORARY = ((int64_t)(((int64_t)index) + ((int64_t)8)));   /* for PHI
node */
   continue;
 }
```

```
 } /* end of syntactic loop 'vector.body' */
 } /* end of syntactic loop 'vector.ph' */
}
```

3. Fast Inv Sqrt

Auto vectorizer supports the fast inv sqrt function (with lower precision) that can be used by checking the box of "**Fast Inverse Square Root**" in the GUI of each kernels as follow:



4. VSI Trigonometric Flags

VSI Auto-vectorizer generates sine and cosine function in various modes (AIE intrinsic and Baskara). User can use the sine and cosine function modes using command line arguments as follows:

```
$./clang_cmd_local.sh -Os sin_fast_test sin_fast_test --trigonometricFlag=original
```

Generated code:
```
v8float tmp__1 = llvm_sin_v8f32(wide_load);
v8float tmp__1 = llvm_cos_v8f32(wide_load);
```

Sine Range: Any input radian
CoSine Range: Any input radian

```
$./clang_cmd_local.sh -Os sin_fast_test sin_fast_test --trigonometricFlag=baskaraOriginal
```

Generated code:
```
v8float tmp__1 = sinBaskara(wide_load);
v8float tmp__1 = sinBaskara(wide_load);
```

Sine Range: [0 … PI]
CoSine Range: [-PI/2 … PI/2]

```
$./clang_cmd_local.sh -Os sin_fast_test sin_fast_test --trigonometricFlag=baskaraUndefRange
```

Generated code:
```
v8float tmp__1 = sinBaskaraUndefRange(wide_load);
v8float tmp__1 = cosBaskaraUndefRange(wide_load);
```

Sine Range: Any input radian
CoSine Range: Any input radian

5. Partial Vectorization

If the user already has a code in aie format, and there is just one or few functions that are still in c++, he can use the “partial vectorization” to only vectorize the required functions, by putting the c++ functions with the following format:

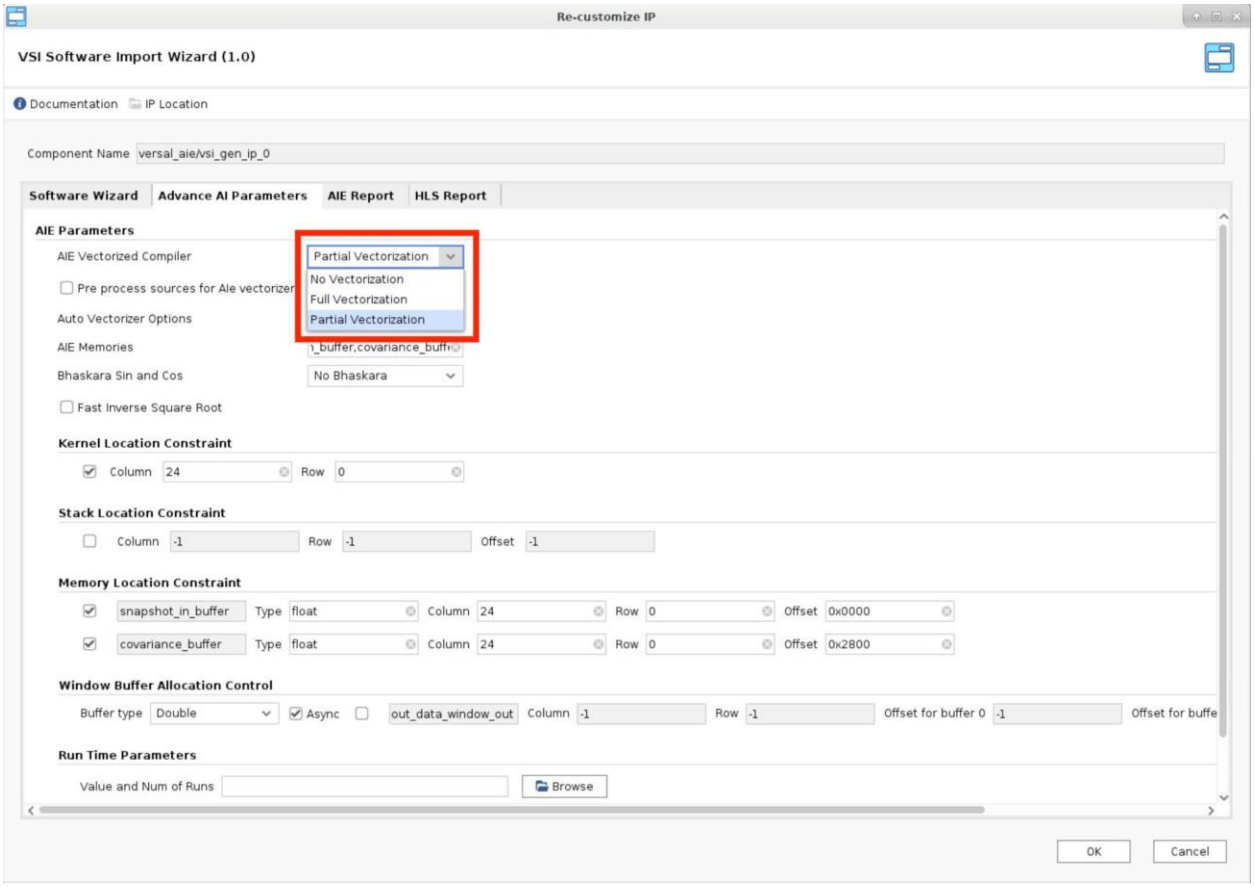
#ifdef __VSI_AUTOVEC__

    //c++ functions
    …

#else
    #include "file_name.cpp"
    //User’s normal aie codes
    …
    // **call for vectorized c++ functions**

#endif

Then after inserting this file to VSI (through vsi_gen_ip) in the “Advance AIE Parameters”, choose “Partial Vectorization” in the “AIE Vectorized Compiler”



The c++ functions in the **__VSI_AUTOVEC__** section will be vectorized and can be called in the **#else** section.

The partial vectorization can also be used when we need to control the input or output behavior, in this case we write a top function in aie syntax (with maybe window_acauire/ window release) and call our main code in it, and put the main code in **__VSI_AUTOVEC__** section to be vectorized.

Example of a file (covariance.cc) for partial vectorizing:

```
#ifdef __VSI_AUTOVEC__
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include "_complex.h"
```

```
void stap_compute_covariance_estimate(
       _complex<float> * __restrict__ covariance,
       _complex<float> * __restrict__ snapshot )
{
    _complex<float> zero(0.0f, 0.0f);
    int i, j, k, conv_index;
    for (k = 0; k < TRAINING_BLOCK_SIZE; k++) {
              for (i = 0; i < N_CHAN*TDOF; ++i)
              {
                     #pragma clang vectorize loop interleave_count(1)
                     for (j = 0; j <N_CHAN*TDOF /*j <= i*/; ++j)
                            {
                                   _complex<float> x = snapshot[k*N_CHAN*TDOF+i] *
cconj(snapshot[k*N_CHAN*TDOF+j]);
                                   conv_index = i*N_CHAN*TDOF + j;
                                   if (j>i) {
                                          x = zero;
                                   }
                                   covariance[conv_index] += (x);
                            }
              }
    }

}

#else

#include <adf.h>
#include <aie_api/aie.hpp>
#include <aie_api/aie_adf.hpp>
#include "covariance.cpp"  // this is the name of the VSI output generated vectorized
source file


#define INTERNAL_ITERATION_NUM 8


void covariance_top(
   // snapshot input from RDMA
   input_stream_float  * snapshot_stream_in,
       // outputs from this kernel
   output_window_float * final_out_data_window_out) {

       for ( int internal_iteration = 0 ; internal_iteration < INTERNAL_ITERATION_NUM;
internal_iteration++) {
              for (int i = 0; i < ( sizeof(snapshot_in_buffer)/ sizeof(float) ); i++ )
{
                     snapshot_in_buffer[i] = readincr(snapshot_stream_in);
              }
              for (int i = 0; i < ( sizeof(covariance_buffer)/ sizeof(float) ); i++ ) {
                     covariance_buffer[i] = 0.0f;
              }
```

```
            struct class__complex* covariance_complex = (struct
class__complex*)covariance_buffer;
            struct class__complex* snapshot_complex_in = (struct
class__complex*)snapshot_in_buffer;
          struct class__complex* final_data_complex_out = (struct
class__complex*)final_data_out_buffer;

            // calling the vectorized stap_compute_covariance_estimate
            stap_compute_covariance_estimate(covariance_complex,
snapshot_complex_in);

            window_acquire(final_out_data_window_out);
            // Send final data out
        for (int i = 0; i < ( sizeof(covariance_buffer)/ sizeof(float) ); i++ ) {
            window_writeincr(final_out_data_window_out, covariance_buffer[i]);
        }
        window_release(final_out_data_window_out);
      }
}

#endif
```

2- AIE parameter buffers:

Because AI Engine local memory is at a premium, it is much more efficient for the AI Engine compiler to manage the arrays explicitly for specific kernels than to leave a large amount of stack or heap space on every processor. The compiler allocates a limited amount of static heap space for such arrays [1].

VSI Approach:

To configure the AIE parameter buffers, two modifications are required:

1. Source code modifications:

The arrays that you want to define as parameter buffers should be first defined in a header file called **kernels.h**, such as:

```
#ifndef USER_PARAMETER_H
#define USER_PARAMETER_H

#ifndef __VSI_AUTOVEC__
#ifdef __X86SIM__
      thread_local
#endif
    int32 lutarray[8]  = {1, 2, 3, 4, 5, 6, 0, 0} ;
#endif
```

The header file should be included in the kernel source file and the buffers can be accessed inside the kernel function directly.

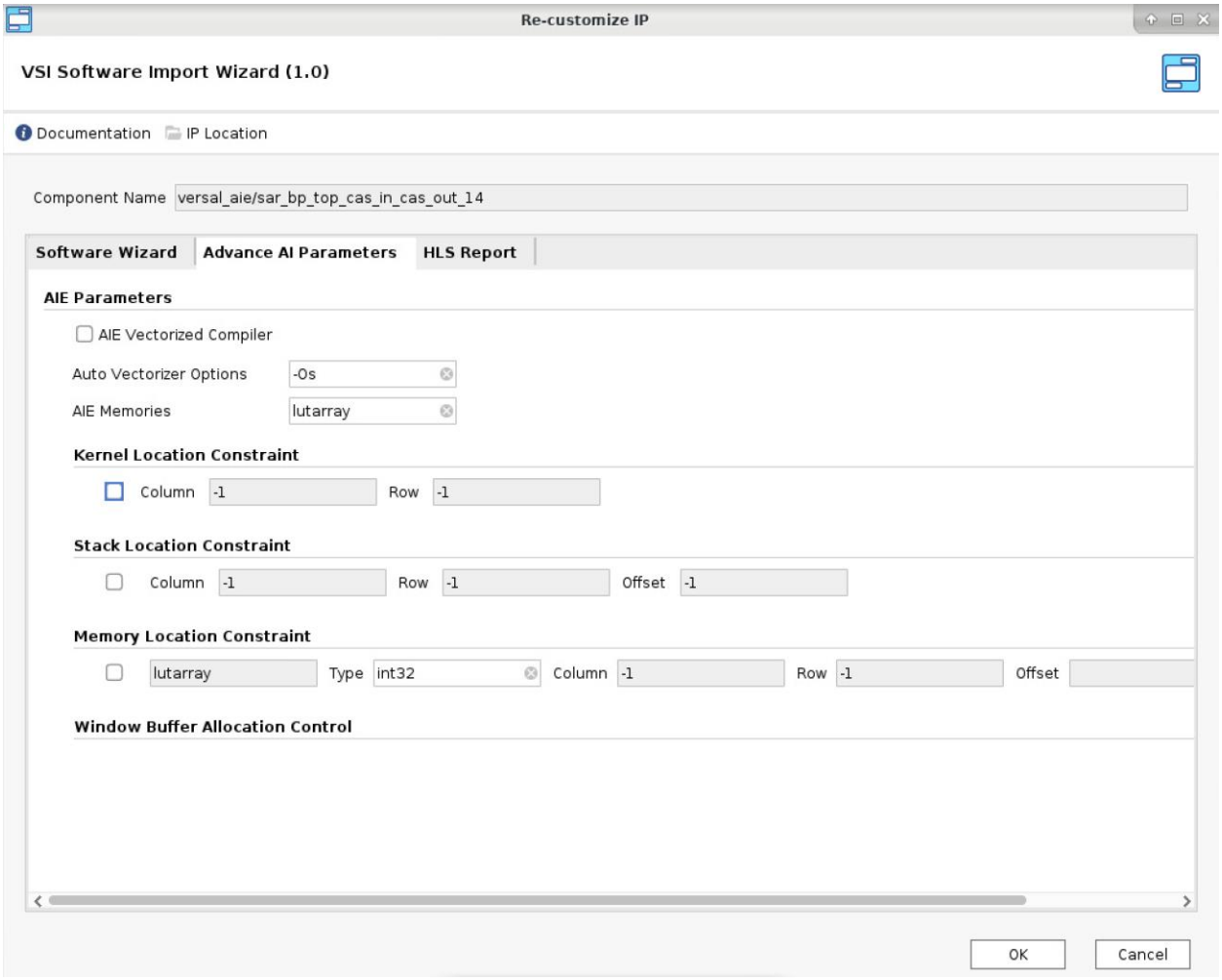
2. GUI modifications:

2-1- Double click on your kernel's block (vsi_gen_ip)

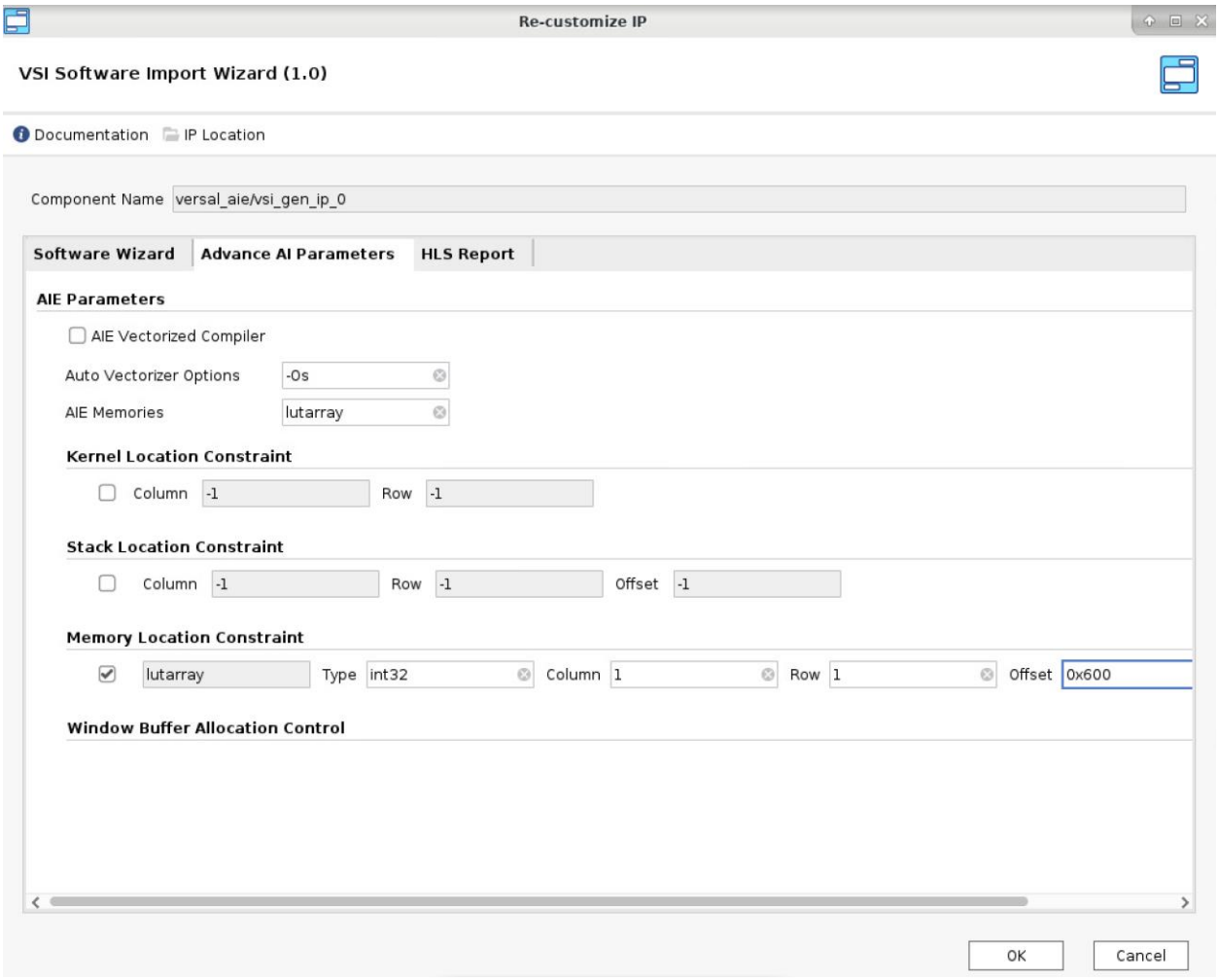
2-2- In the "Advanced AI Parameters tab", type the name of the arrays in front of the "AIE Memories" input as a comma separated list

2-3- Deactivate the "AIE Memories" box, then the "Memory Location Constraint" appears.

2-4- If you don't want to put location constraints for these buffers, just type in the type of the array in front of the "Type" box



2-5- If you need to add the location constraints for them, click the box next to each array name and add the “column” “row” and “offset” for them (offset can be in Hex or Decimal).


Re-customize IP
VSI Software Import Wizard (1.0)
Documentation
IP Location
Component Name
versal_aie/vsi_gen_ip_0
Software Wizard
Advance AI Parameters
HLS Report
AIE Parameters
AIE Vectorized Compiler
Auto Vectorizer Options
-Os
AIE Memories
lutarray
Kernel Location Constraint
Column
-1
Row
-1
Stack Location Constraint
Column
-1
Row
-1
Offset
-1
Memory Location Constraint
lutarray
Type
int32
Column
1
Row
1
Offset
0x600
Window Buffer Allocation Control
OK
Cancel

3- RunTime Parameters in VSI

This section describes the Run Time Parameters (RTP) flow in VSI.
Run time parameters are used to modify the behavior of the graph based on some dynamic condition. Chapter 5 of [1] describes the details of run time parameters.

Supported RunTime Parameter Features in VSI

- Both scalar and array parameters are supported.
- Constant (input port) and passed by reference are supported (passed by value is not supported).
- Structs and pointers cannot be passed as run-time parameters.
- Sync/Async RTPs are supported.
- The valid scalar data types supported are:

  int16, int32, cint16, cint32, float, cfloat

Formal Parameter	Port Class
T &	Inout
Const T &	Input
Const T (&)[..]	Input
T (&)[..]	Inout

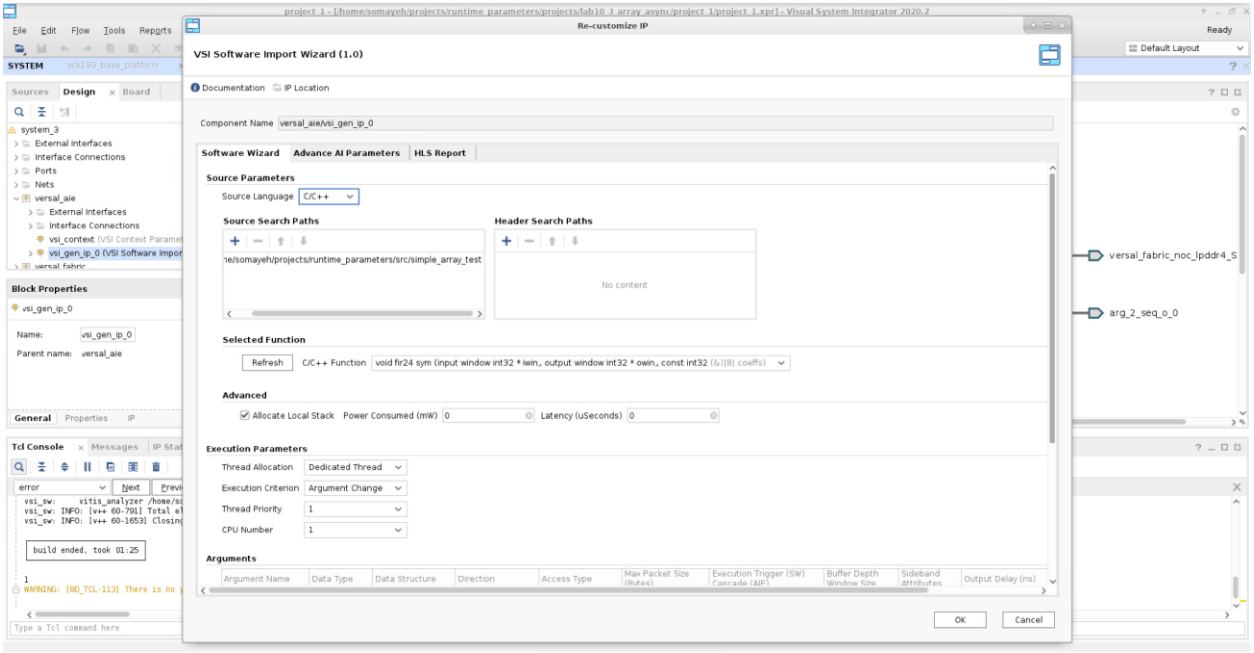
VSI Approach

1. Insert your code into the VSI software wizard.

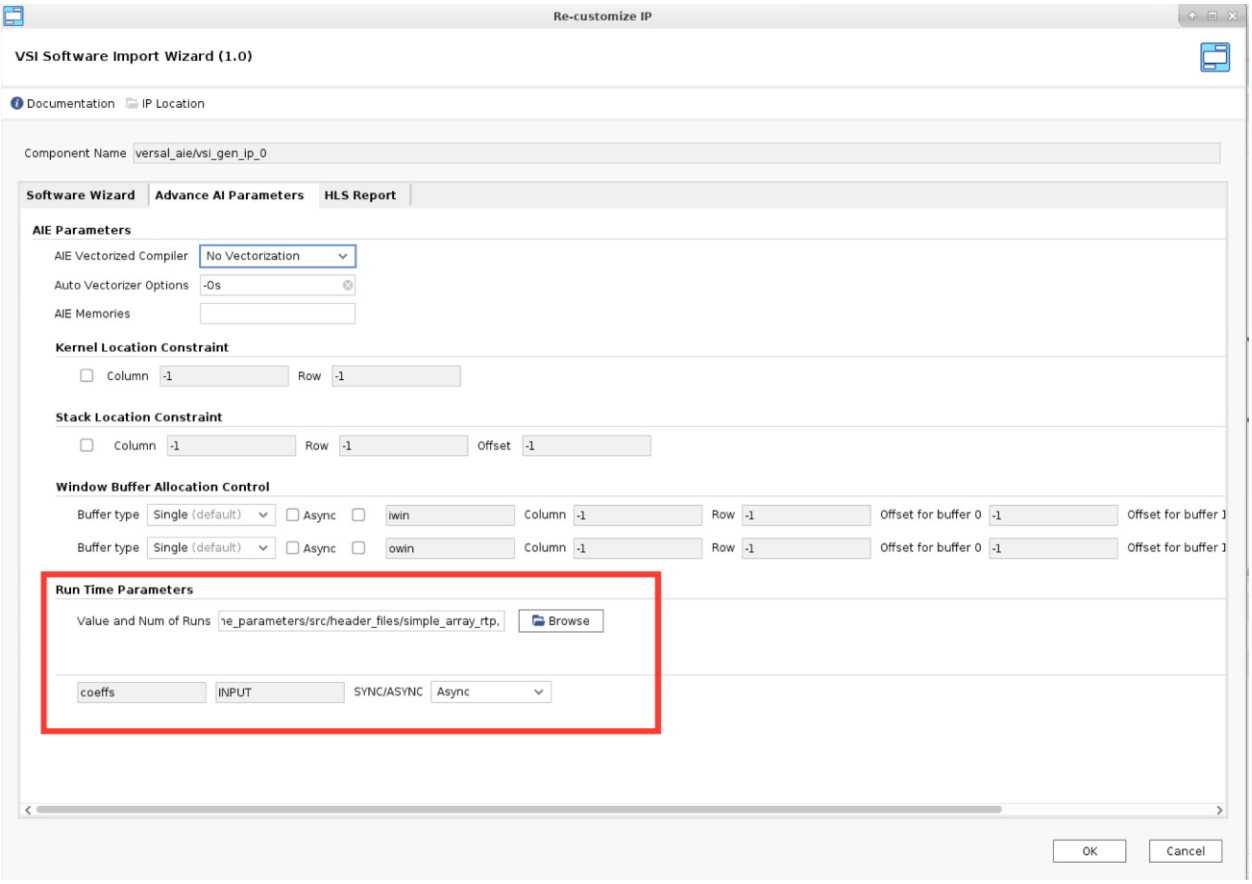
We insert the following function:

void fir24_sym(input_window_int32 *iwin, output_window_int32 *owin, const int32 (&coeffs)[8])

that has one input window (iwin), one input run time parameter (coeffs) and one output window (owin).



The VSI will parse the function and recognize the RunTime parameters (the arguments that are not window or stream types) and show them in the second tab “**Advanced AI Parameters**”, in the “**Run Time Parameter**” section:



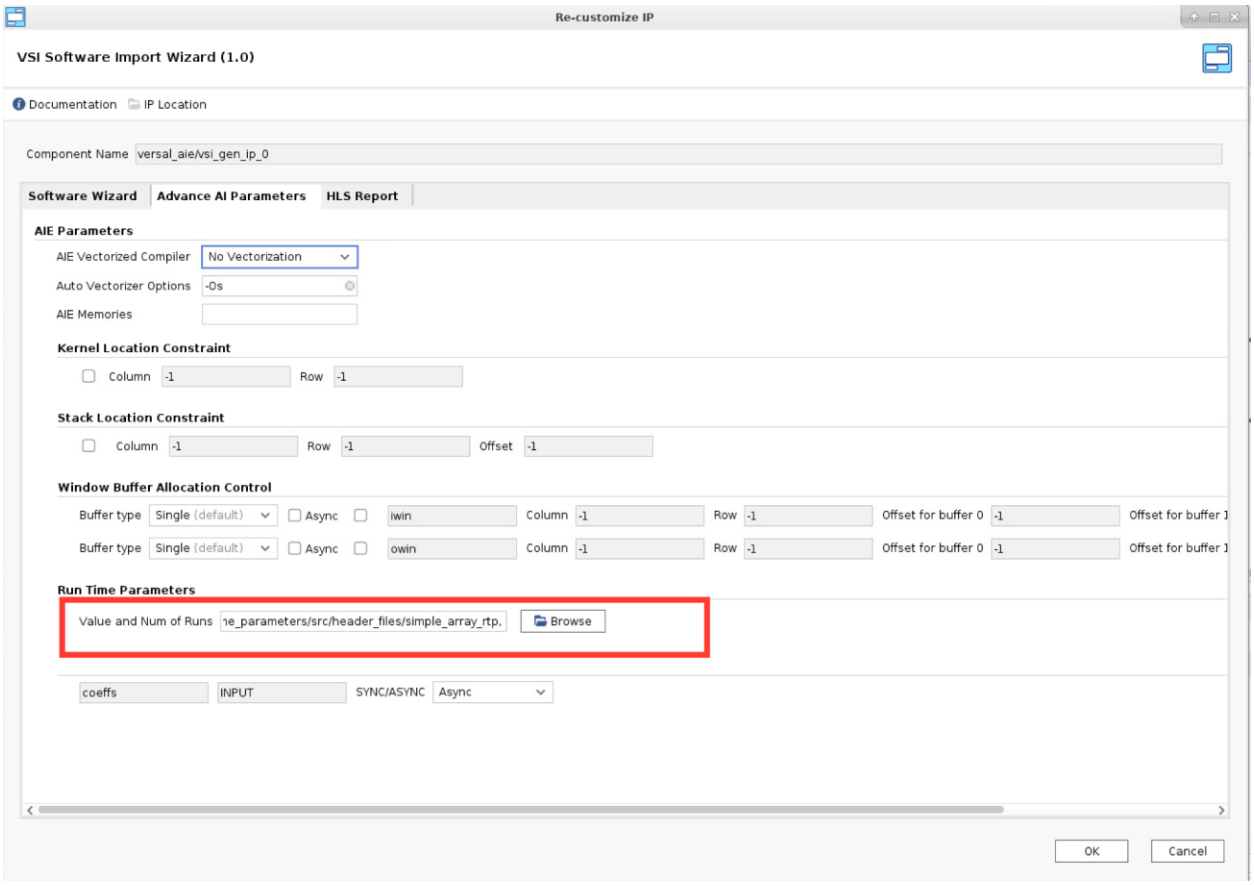
2. Determine if the RTPs are Syn/Async in the GUI, here we chose Asynchronous for coeffs parameter.
    a. An **asynchronous** parameter can be updated at any time, that value is observed on the next and any subsequent kernel run.
    b. A kernel that has a **sync** parameter does not execute until sync parameters have been written. Upon a write, the kernel executes once, reading the new updated value. After completion, the kernel is blocked from executing until the parameter is updated again.

3. For each RTP, you must determine the value/values and how many times each update should run. Create a file called **runtime_param.h** with the following format for all the RTPs:
   parameter_name = {value/values}{number of runs}

   For example, for coeffs array, we would like to update the values to {1, 2, 3, 4, 5, 6, 7, 8} and run it once then update the values to {10, 11, 12, 13, 14, 15, 16, 17} and run it 2 times so the runtime_param.h file will be as follow:

   int coeffs[8] = {1, 2, 3, 4, 5, 6, 7, 8}{1};
   int coeffs[8] = {10, 11, 12, 13, 14, 15, 16, 17}{2};

4. Add the recently generated runtime_param.h file to the VSI:
   In the second tab "**Advanced AI Parameters**", in "**Run Time Parameter**" section, browse for the runtime_param.h file in the "**Value and Num of Runs**" field as follow:



5. That is all, now you can close the Software Wizard, "**generate system**" and "**build AIE**", and simulate it to see the results.

4- Power Estimation

In order to estimate the power of an AIE kernel, the following steps are required:

1. Run vcdanalyze –vcd foo.vcd
2. This will generate a file called events.txt, in ./trdata.aiesim folder
3. This text file is what generates the waveform viewed in the GUI, and thus has name information embedded to the events
4. Finding the first and last event associated with the kernel top function name and extracting the start and end times
5. Run vcdanalyze with the –xpe flag with start and end times

VSI Approach

VSI automated the mentioned steps in a bash script called, power_estimation.sh in the following path:

<path_to_your_project>/vsi_auto_gen/sw/<system_name>/versal_aie/power_estimate.sh

After the AIE simulator has been ran, you just need to run the power_estimate.sh file:

sh ./<path_to_your_project>/vsi_auto_gen/sw/<system_name>/versal_aie/power_estimate.sh

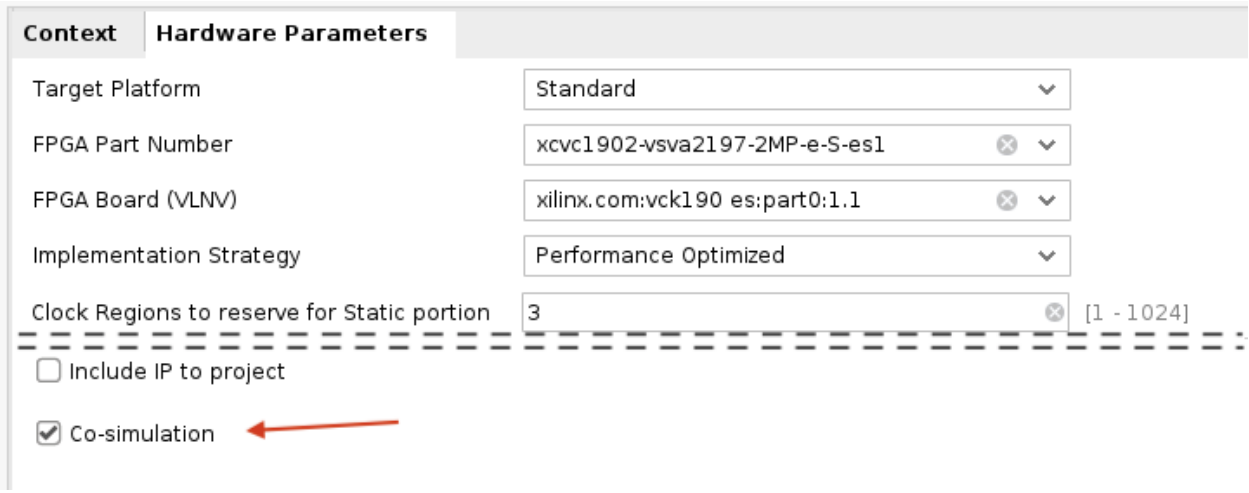
that will generate the XPE file that is located in ./aiesim_xpe/[top_level_name].xpe
The .xpe file is the input into the Xilinx XPE power Excel spreadsheet that is currently only supported by MS Windows operating system.

5- Debugging

Co-simulation

Currently, co-simulation only works with a versal base system.

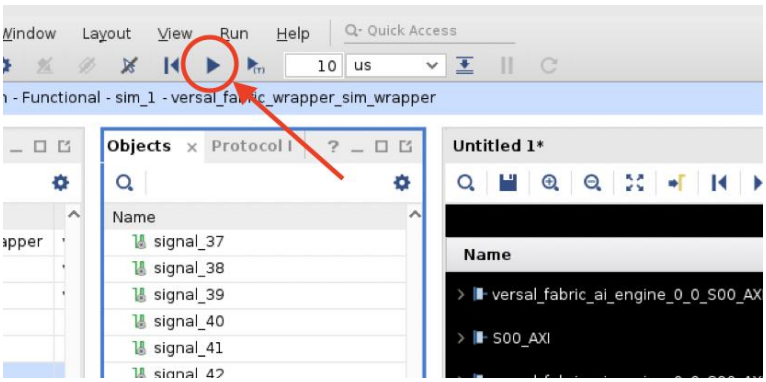
1. To enable cosimulation, open a system canvas and set the Co-simulation checkbox in the Versal Fabric context IP.



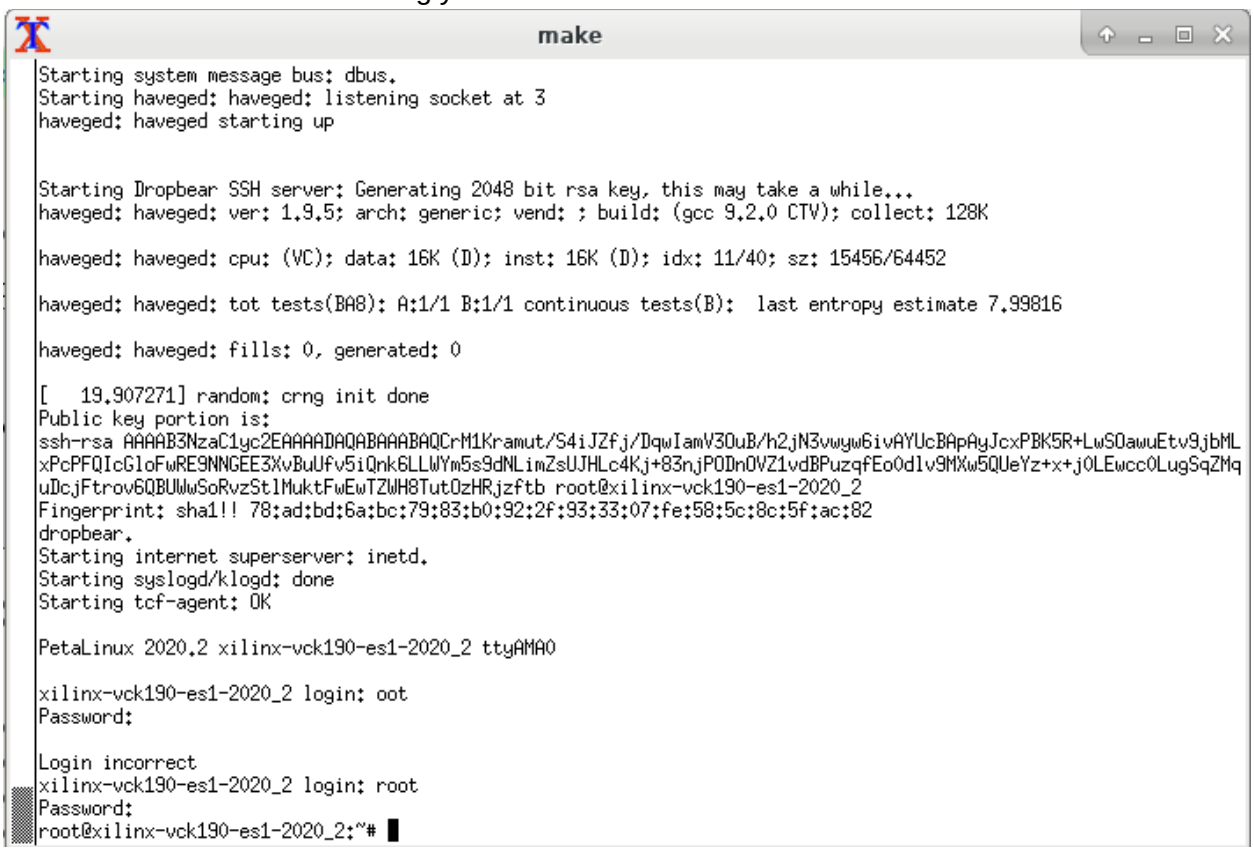
2. To generate require files:
    a. Generate System.
    b. Build Software if it wasn't built previously (Flow -> Build-> Build Software...)
    c. Build AIE (Flow -> Build-> Build AIE)
    d. Build hardware (Flow -> Build-> Build Hardware...)
    *This step will generate only co-simulation project, bitstream will not be generated.

3. Launch emulation:
    a. To launch in **non-GUI** mode run the next command in TCL terminal:
    vsi::launch_co_simulation <hw_context_name>

    b. To launch in **GUI** mode run the next command in TCL terminal:
    vsi::launch_co_simulation <hw_context_name> -mode gui
    At GUI mode you can add weirs to observe and hit “Run All”



4. You will see a terminal emulating your board:



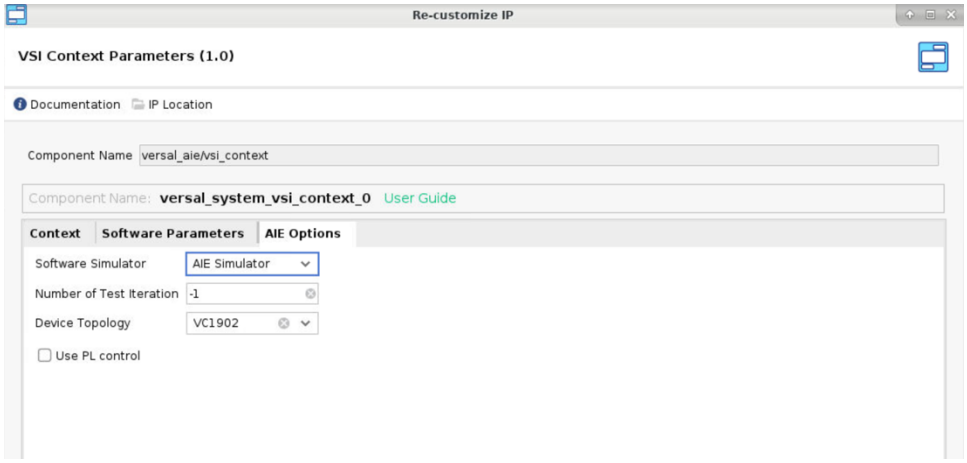
Login: root
Password: root

Go into the ***/mnt/sd-mmcblk0p1/*** directory.
In this directory, you will find a folder that has the same name as your software context.
Run a launcher from this folder:
 # software_context_name>
 # ./logrun.sh

If you have multiple software contexts, run the launcher which belongs to this context!

Running AIE simulator (for AIE kernels only)

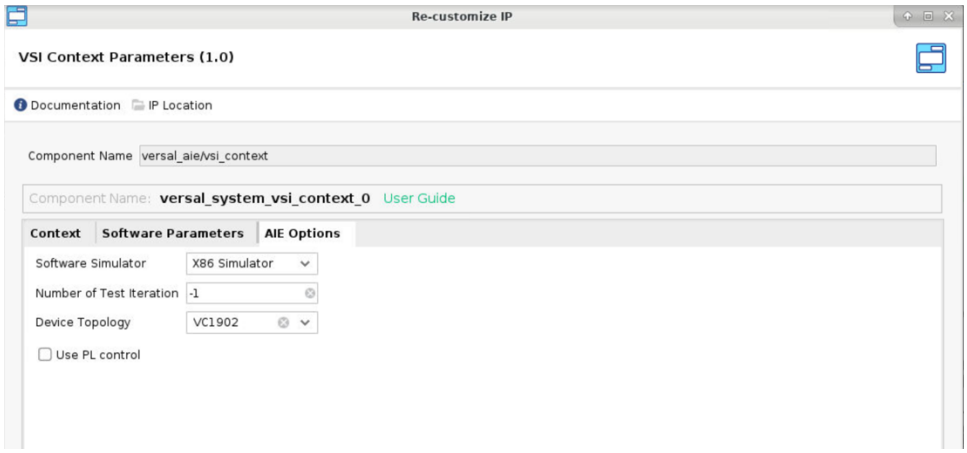
1. To enable X86simulation, open a system canvas and select the "**Software Simulator**" as "**X86 Simulator**" in the context IP.



2. Generate System.
3. Build AIE
4. and then Run the simulator as follow:
    a. cd <proj_dir>/vsi_auto_gen/sw/system_1/build/versal_aie/versal_aie
    b. sh ./../../../versal_aie/run_simulation.sh
    c. The first time you run it, you will receive the error that could not open input file, you need to provide the input data file for it:
    Error: Error: Could not open input file :
    ../vsi_system_versal_aie_vsi_gen_ip_0_A!
    d. then run it again and check the outputs in the ./x86simulator_output

Running X86 simulator (for AIE Kernels only)

5. To enable X86simulation, open a system canvas and select the "**Software Simulator**" as "**X86 Simulator**" in the context IP.



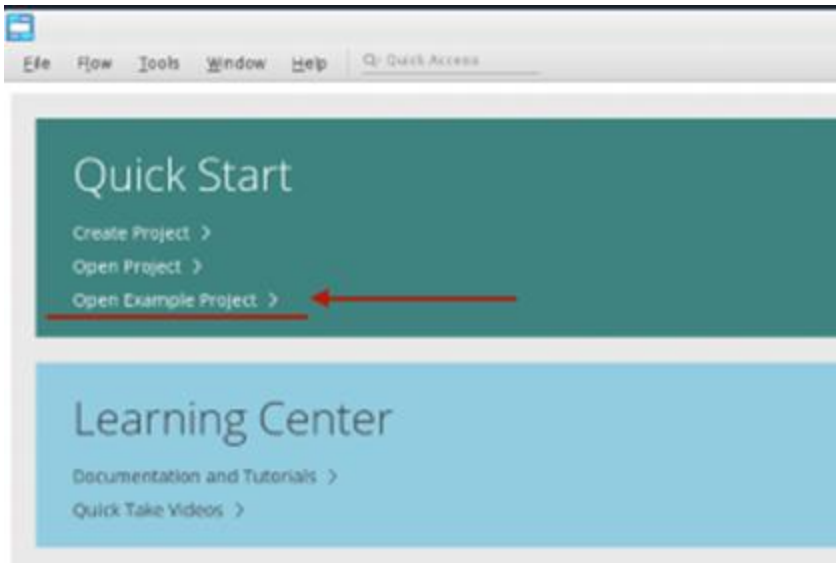
6. Generate System.
7. Build AIE
8. and then Run the simulator as follow:
    a. cd <proj_dir>/vsi_auto_gen/sw/system_1/build/versal_aie/versal_aie
    b. sh ./../../../versal_aie/run_simulation.sh
    c. The first time you run it, you will receive the error that could not open input file, you need to provide the input data file for it:
    Error: Error: Could not open input file :
    ../vsi_system_versal_aie_vsi_gen_ip_0_A!
    d. then run it again and check the outputs in the ./x86simulator_output

Running X86 simulator (for AIE + RTL)

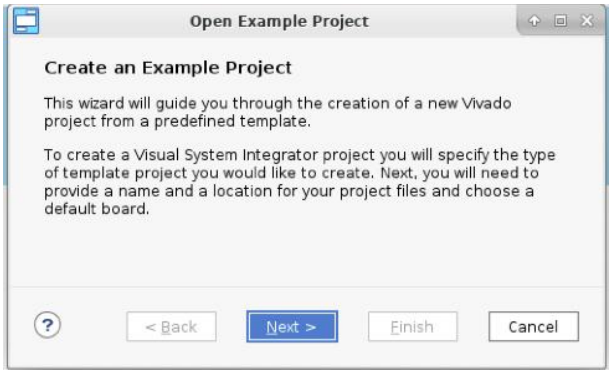
AIe Emulation Example Project

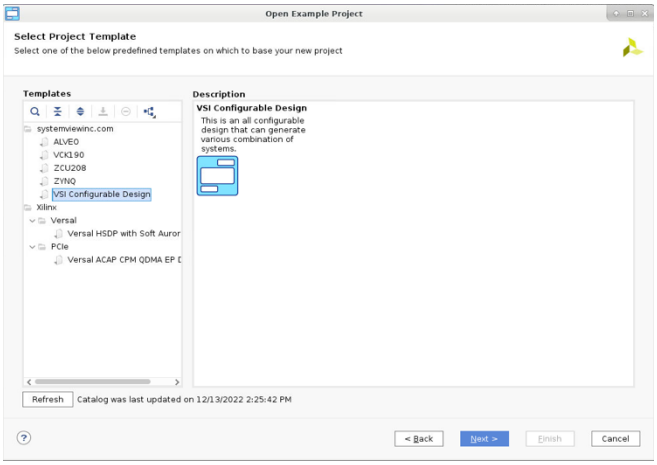
Create project

1. Click “Open Example Project”

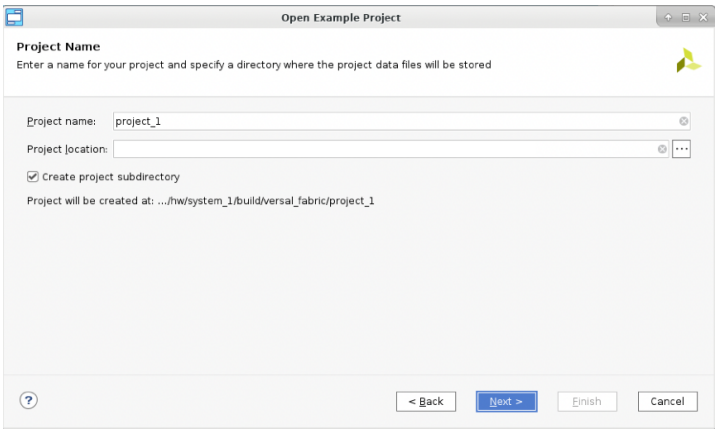


2. Click the “Next” button in the “Create an Example Project” and in the “Select Project Template” window select “VSI Configurable Design”.

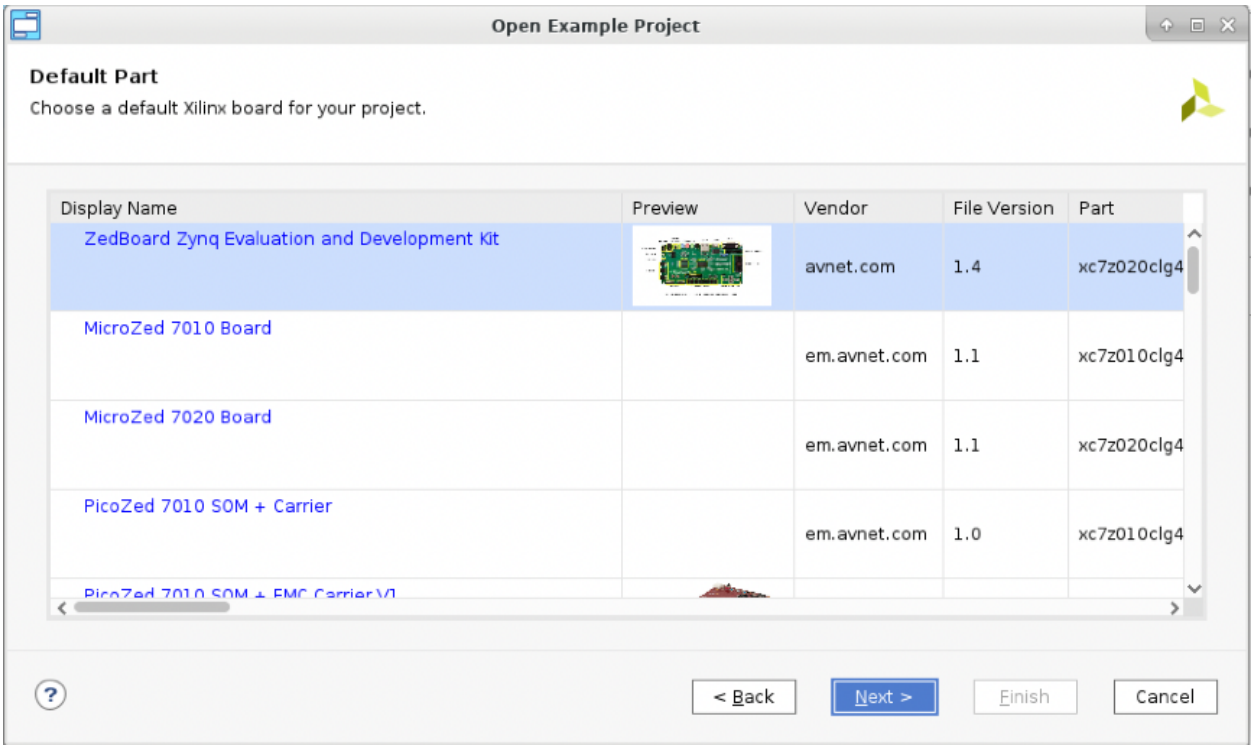




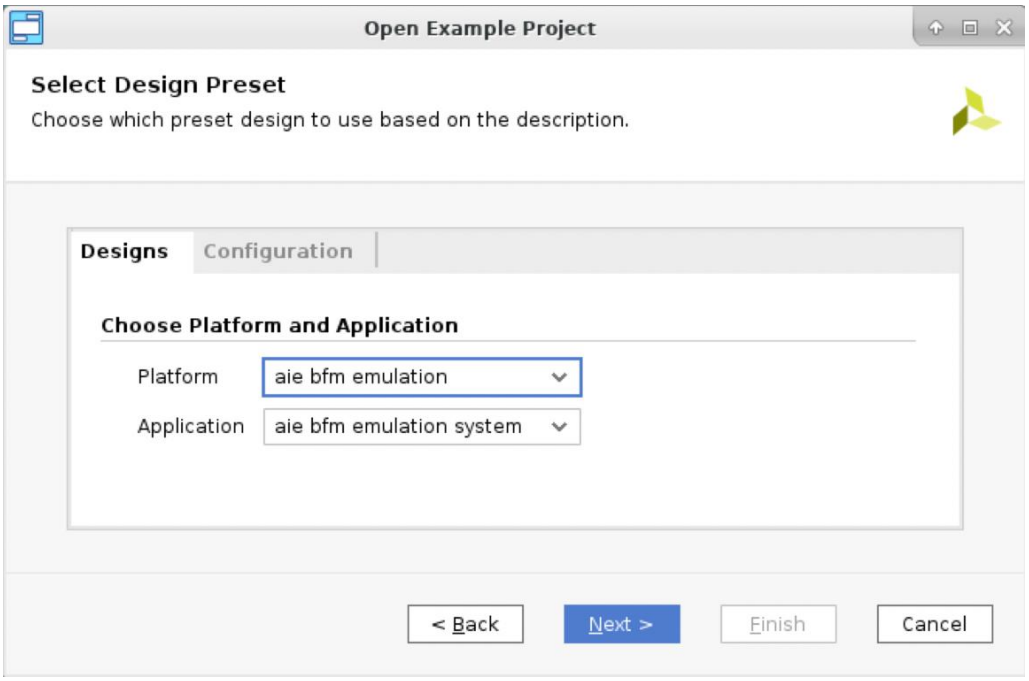
3. Set project name and location at the appear dialog window, and click “Next”:



4. At the “Default Part” dialog window select any board, and click “Next”:



5. At the “Design Preset” window select: Platform->aie_bfm_emulation, Application->aie_bfm_emulation_system, and click “Next”



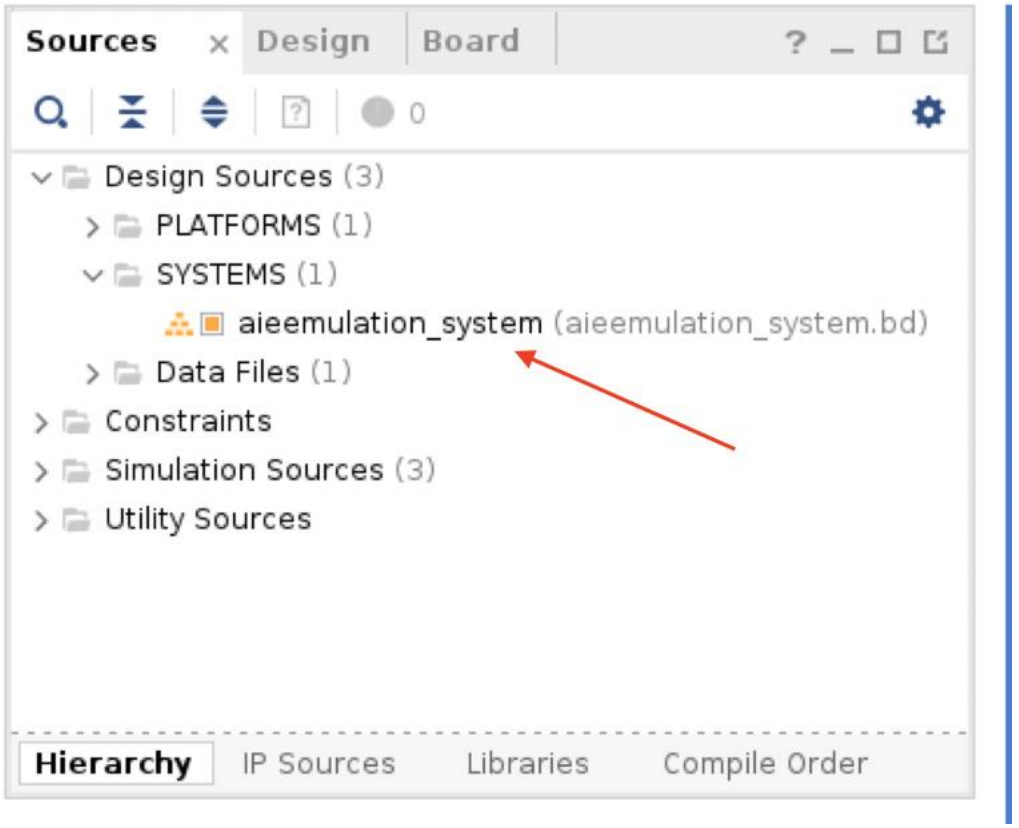
6. Click “Finish”

Now you have an example of aie emulation project.

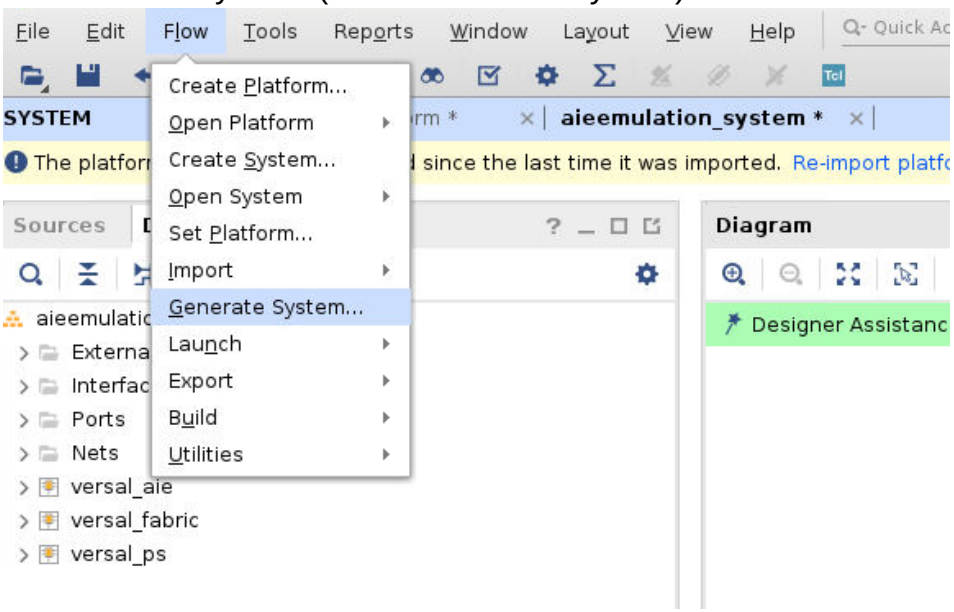
Build

To build it, you need to follow the following steps:

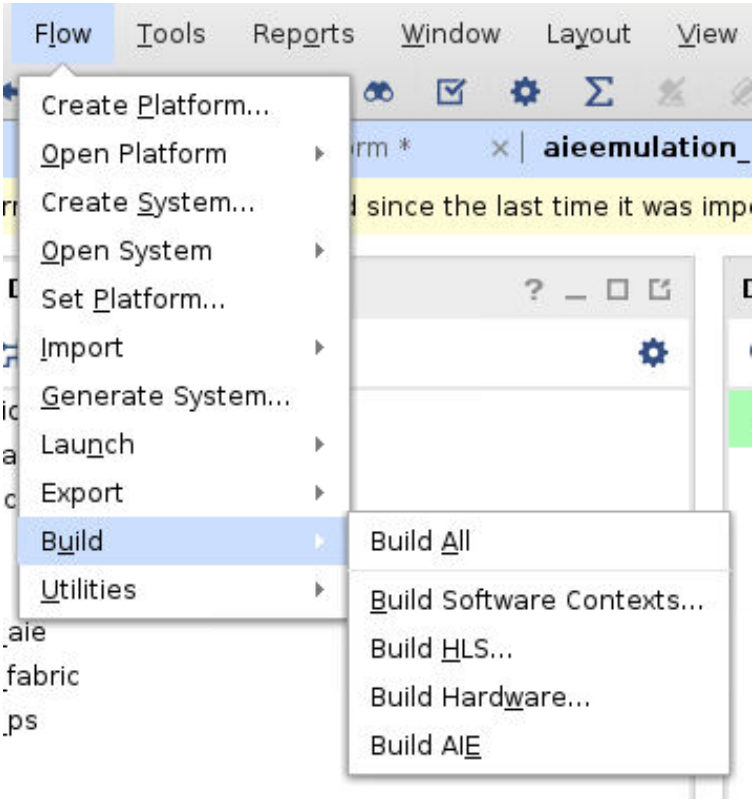
1. Open your system (Double click on system diagram).



2. Generate System: (Flow->Generate System)



3. Build targets:
   - Flow -> Build -> Build Software Context
   - Flow -> Build -> Build AIE
   - Flow -> Build -> Build HLS
   - Flow -> Build -> Build Hardware

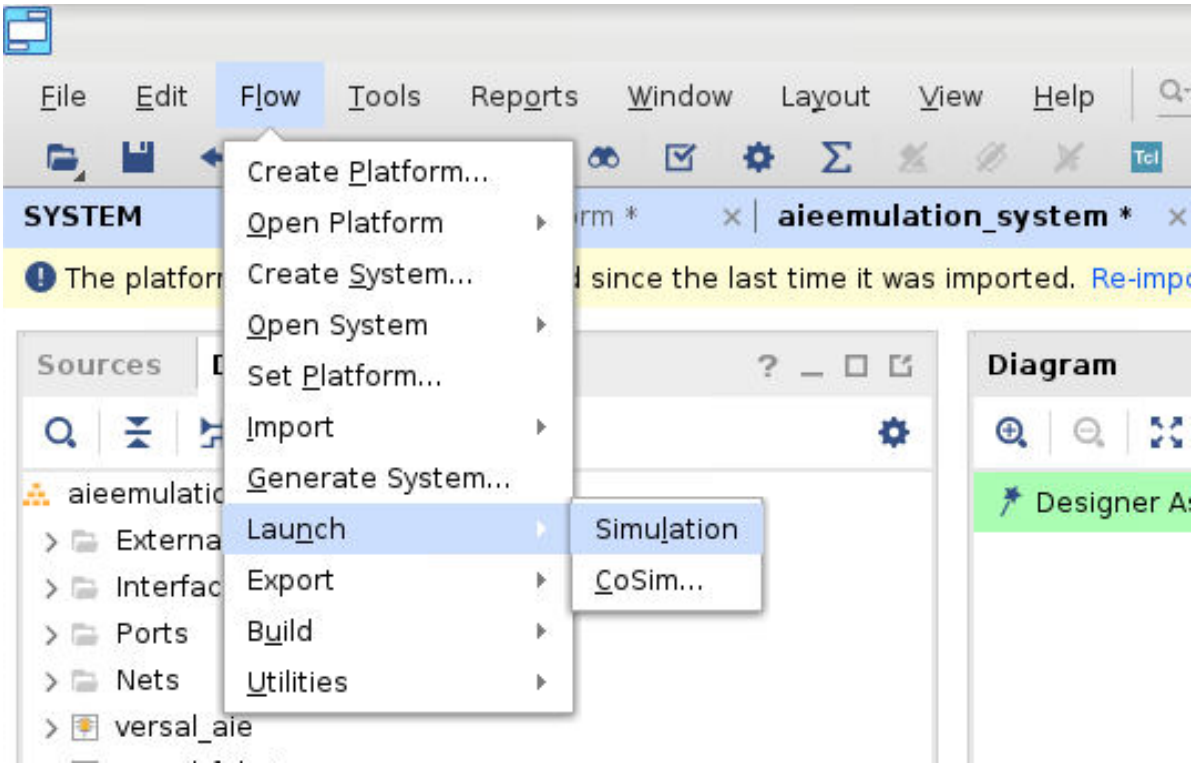


Launch

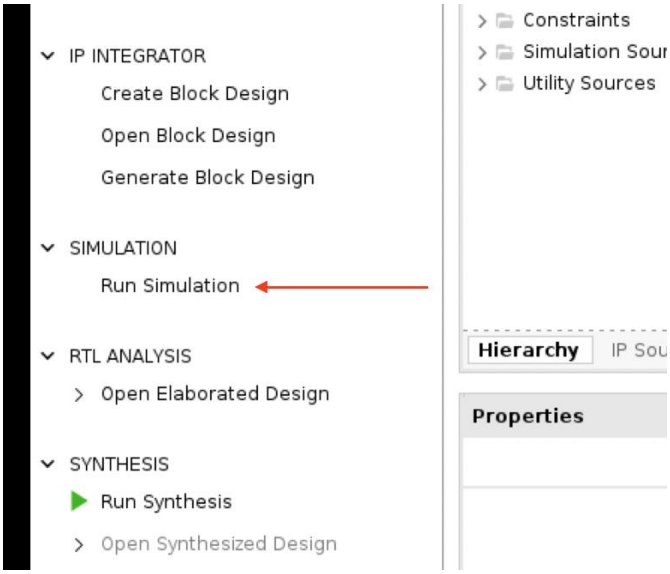
Launching an example design has two steps, running simulation and running software.

Running simulation

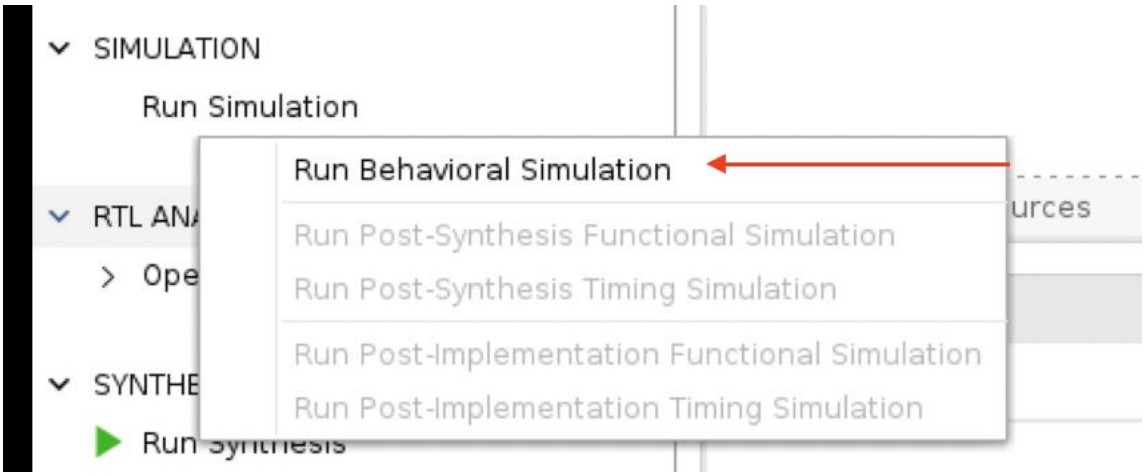
Launch Simulation (Flow->Launch->Simulation)



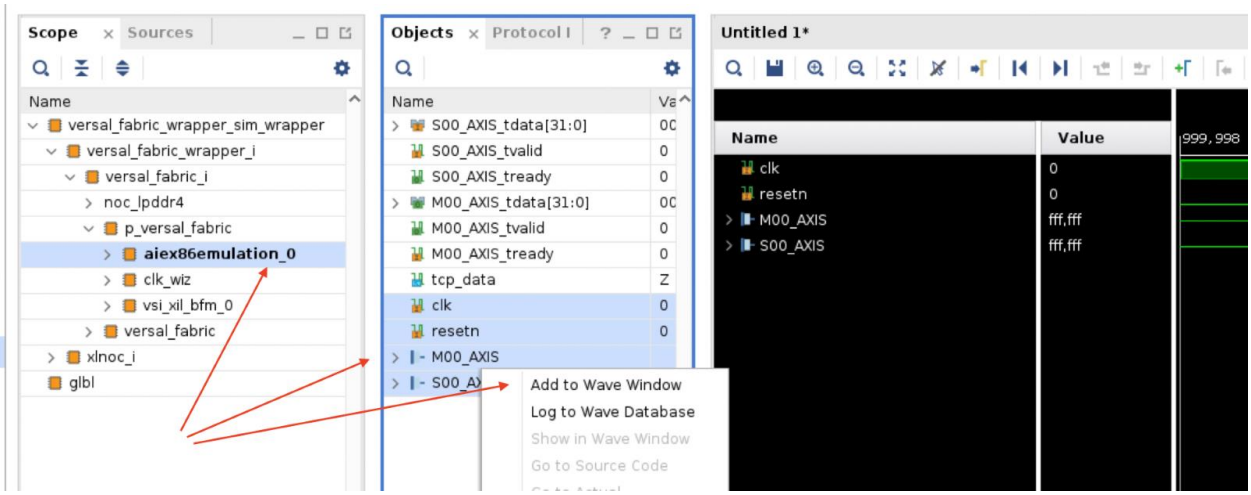
Because of using the Vivado simulation engine, Launch simulation will open a Vivado project generated at previous steps. At the opened Vivado project, click “Run Simulation”:



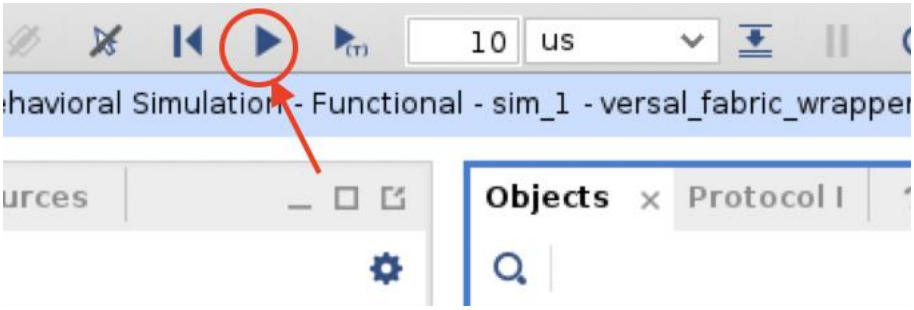
In the drop-down menu, click on “Run Behavioral Simulation.”



In the simulation window, you can add wires and interfaces that you need to analyze. Add interfaces of the AIE emulation IP to see data transactions:



Click “Run All” to button:



Running software

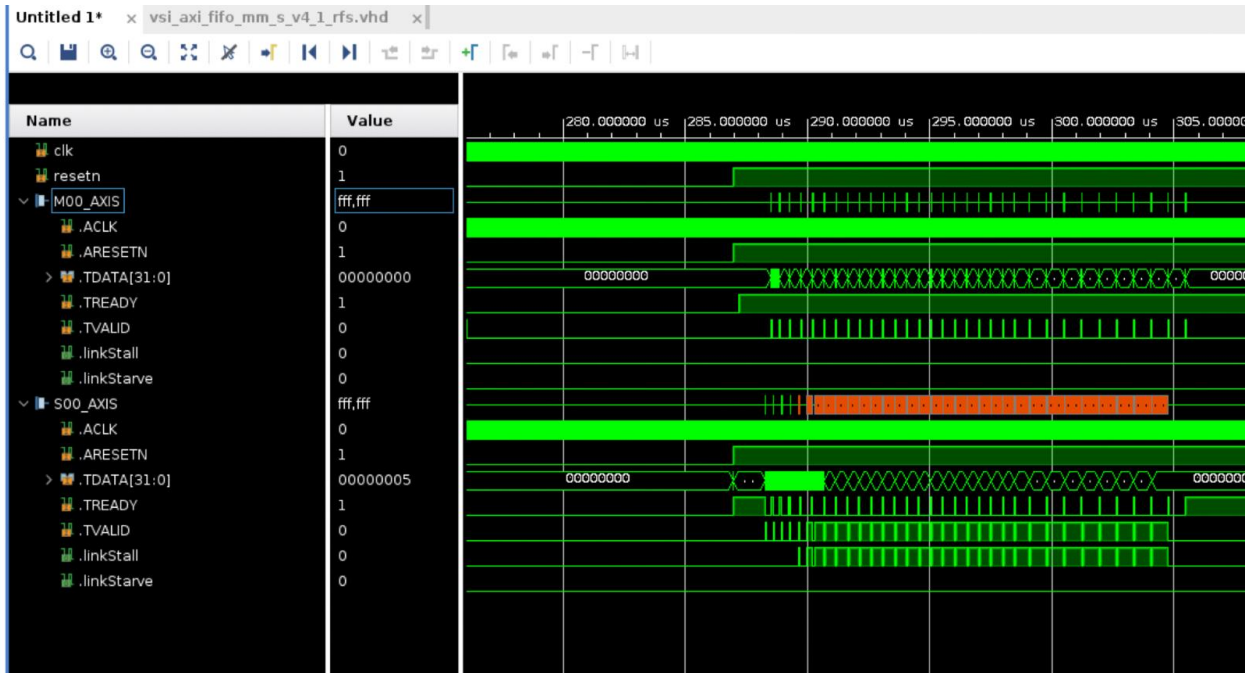
Open a new terminal, and navigate to the folder with generated software:

```
cd <path_to_your_project>/vsi_auto_gen/output_files/aieemulation_system/versal_ps
```

Run generated script:

```
./logrun.sh
```

After software execution, you will see a message “Done” in your terminal.
In your simulation window, you may observe transactions between RTL simulation and AIe emulation:



6- Performance Analysis

After running the system in simulation, a run_summary report is generated when the application has been properly configured. If you didn't configure your design, see the Design Flow to get more information about how to generate your design and vectorize it and see Running AIE Simulator.

During the simulation of the AI Engine graph, the AI Engine simulator, captures performance and activity metrics and writes the report to the following directory: <proj_dir>/vsi_auto_gen/sw/<system_name>/build/versal_aie/versal_aie /**aiesimulator_output**

The generated summary is called **default.aierun_summary**.

The run_summary can be viewed in the Vitis Analyzer. The summary contains a collection of reports, capturing the performance profile of the AI Engine application captured as it runs. To visualize the report, use the Vitis Analyzer as follow:

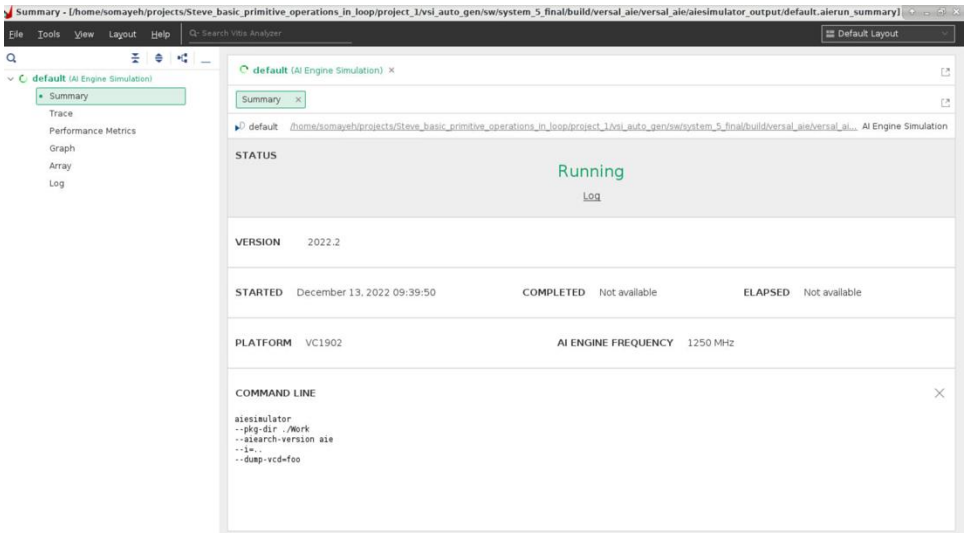
cd <proj_dir>/vsi_auto_gen/sw/<system_name>/build/versal_aie/versal_aie /aiesimulator_output

vitis_analyzer default.aierun_summary

The Vitis Analyzer opens displaying the Summary page of the report. The Report Navigator view of the tool lists the different reports that are available in the summary.

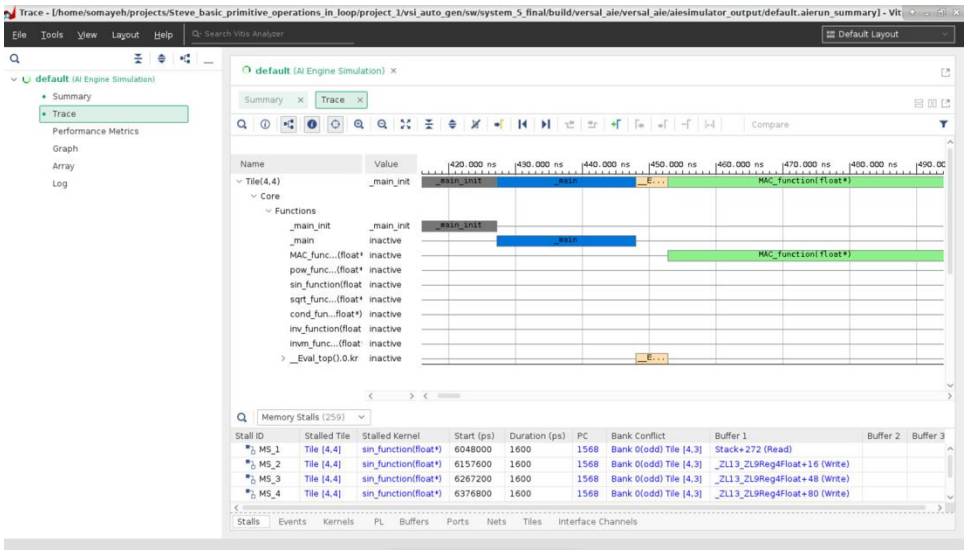
Report Summary

This is the top-level of the report, and reports the details of the run, such as date, tool version, and the command-line used to launch the simulator.



Trace Report

Event trace provides a systematic way of collecting system level traces for the program events, providing direct support for generation, collection, and streaming of hardware events as a trace. The following image shows the Trace report open in the Vitis Analyzer.



_main
Core main function. This is different from the function used in the top-level file.
_main_init
Kernel init function that runs once per graph execution.
_cxa_finalize
Calls destructors of global C++ objects.
_fini

This section holds executable instructions that terminate the process. When a program exits normally, the system runs the code in this section.

Features of Trace Report

Features of the trace report include the following.

- Each tile is reported. Within each tile the report includes core, DMA, locks, and I/O.
- There is a separate timeline for each kernel mapped to a core. It shows when the kernel is executing (blue/green) or stalled (red) due to memory conflicts or waiting for stream data.
- By using lock IDs in the core, DMA, and locks sections you can identify how cores and DMAs interact with one another by acquiring and releasing locks.
- If a lock is not released, a red bar extends through the end of simulation time.
- Clicking the left or right arrows takes you to the start and end of a state, respectively.
- By using the bar lines at the start and end of a function, you can measure the latency of that particular function

For a complete understanding of the Vitis Analyzer, see UG1393 Xilinx Document (Using the Vitis Analyzer in the Vitis Unified Software Platform Documentation: Application Acceleration Development) [2].

[1]https://docs.xilinx.com/viewer/book-attachment/q_Yc6QkQHbaC2~Qz9NTtmg/bOzzi7rGdZVaHbXp_k3WtA
[2] https://docs.xilinx.com/r/en-US/ug1393-vitis-application-acceleration/Using-the-Vitis-Analyzer