Traineeships in Advanced Computing for High Energy Physics (TAC-HEP)

GPU & FPGA module training: Part-2

Week-3: *Hands-on with vivado_hls, Intro to Pragmas*

Lecture-6: April 5th 2023

Varun Sharma

University of Wisconsin – Madison, USA

So Far...

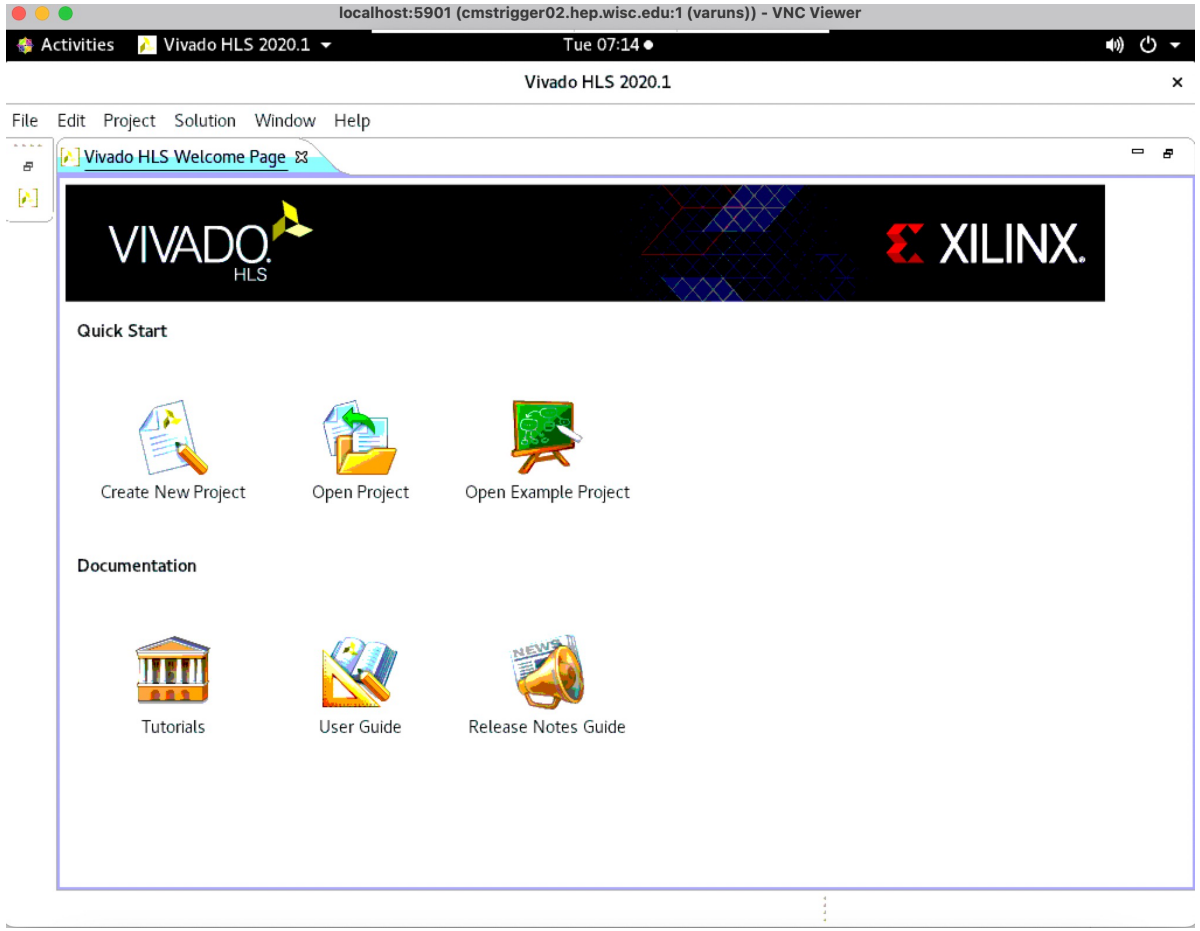
- **FPGA and its architecture**
  - Registor/Flip-Flops, LUTs/Logic Cells, DSP, BRAMs
  - Clock Frequency, Latency
  - Extracting control logic & Implementing I/O ports
- **Parallelism in FPGA**
  - Scheduling, Pipelining, DataFlow
- **Vivado HLS**
  - Introduction, Setup, Hands-on for GUI/CLI

Today:

- Continue with hands-on
- Introduction to Pragmas

Reminder: Steps to follow

- **Step-1: Creating a New Project/Opening an existing project**
- **Step-2: Validating the C-source code**
- **Step-3: High Level Synthesis**
- **Step-4: RTL Verification**
- **Step-5: IP Creation**



Ex: lec6Ex1

Eg.: lec6Ex1

```
#include "lec6Ex1.h"

void lec6Ex1 (
  unsigned int in[N],
  char a,
  char b,
  unsigned int c,
  unsigned int out[N]
  ) {

   unsigned int x, y;

for_Loop: for (unsigned int i=0 ; i < N; i++) {
        x = in[i];
        y = a*x + b + squared(c);
        out[i] = y;
      }
}

unsigned int squared(unsigned int a)
{
  unsigned int res = 0;
  res = a*a;
  return res;
}
```

```
#ifndef LEC6EX1_H_
#define LEC6EX1_H_

#include <stdio.h>
#include <math.h>
//#include <cmath>
//#include "hls_math.h"

#define N 20

void lec6Ex1 (
  unsigned int in[N],
  char a,
  char b,
  unsigned int c,
  unsigned int out[N]
  );

unsigned int squared(unsigned int );
//int get_sqrt(float x, float *ret);

#endif
~
~
```

Ex: lec6Ex1 -Synthesis Result

Eg.: lec6Ex1

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.152 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
61	61	0.610 us	0.610 us	61	61	none

Detail

Instance

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- for_Loop	60	60	3	-	-	20	no

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	5	0	146	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	30	-
Register	-	-	115	-	-
Total	0	5	115	176	0
Available	650	600	202800	101400	0
Utilization (%)	0	~0	~0	~0	0

Ex: lec6Ex1 -Synthesis Result

Eg.: lec6Ex1

Expression

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
mul_ln15_fu_128_p2	*	2	0	21	32	8
res_fu_99_p2	*	3	0	21	32	32
add_ln15_fu_105_p2	+	0	0	39	32	32
i_fu_117_p2	+	0	0	15	5	1
y_fu_133_p2	+	0	0	39	32	32
icmp_ln13_fu_111_p2	icmp	0	0	11	5	5
Total		**6**	**5**	**0**	**146**	**138**	**110**

Multiplexer

Name	LUT	Input Size	Bits	Total Bits
ap_NS_fsm	21	5	1	5
i_0_reg_80	9	2	5	10
Total	**30**	**7**	**6**	**15**

Register

Name	FF	LUT	Bits	Const Bits
add_ln15_reg_143	32	0	32	0
ap_CS_fsm	4	0	4	0
i_0_reg_80	5	0	5	0
i_reg_151	5	0	5	0
sext_ln15_reg_138	32	0	32	0
y_reg_166	32	0	32	0
zext_ln14_reg_156	5	0	64	59
Total	**115**	**0**	**174**	**59**

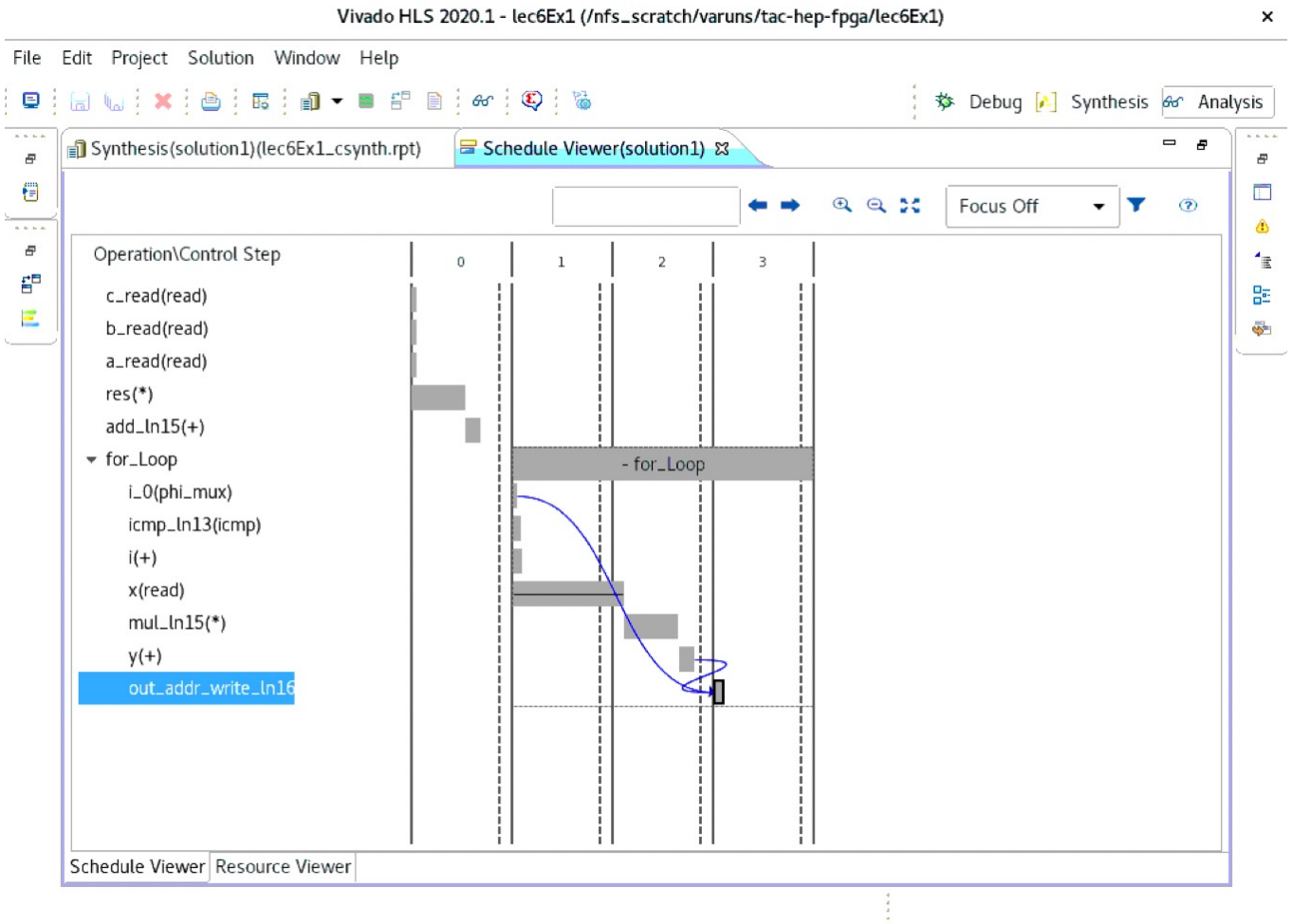
Interface

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	lec6Ex1	return value
ap_rst	in	1	ap_ctrl_hs	lec6Ex1	return value
ap_start	in	1	ap_ctrl_hs	lec6Ex1	return value
ap_done	out	1	ap_ctrl_hs	lec6Ex1	return value
ap_idle	out	1	ap_ctrl_hs	lec6Ex1	return value
ap_ready	out	1	ap_ctrl_hs	lec6Ex1	return value
in_r_address0	out	5	ap_memory	in_r	array
in_r_ce0	out	1	ap_memory	in_r	array
in_r_q0	in	32	ap_memory	in_r	array
a	in	8	ap_none	a	scalar
b	in	8	ap_none	b	scalar
c	in	32	ap_none	c	scalar
out_r_address0	out	5	ap_memory	out_r	array
out_r_ce0	out	1	ap_memory	out_r	array
out_r_we0	out	1	ap_memory	out_r	array
out_r_d0	out	32	ap_memory	out_r	array

Ex: lec6Ex1 -Synthesis Result

Eg.: lec6Ex1




```
#include "lec6Ex1.h"

void lec6Ex1 (
  unsigned int in[N],
  char a,
  char b,
  unsigned int c,
  unsigned int out[N]
  ) {

   unsigned int x, y;

for_Loop: for (unsigned int i=0 ; i < N; i++) {
        x = in[i];
        y = a*x + b + squared(c);
        out[i] = y;
      }
}

unsigned int squared(unsigned int a)
{
  unsigned int res = 0;
  res = a*a;
  return res;
}
```

Ex: lec6Ex2

Eg.: lec6Ex2

```
#include "lec6Ex2.h"

void lec6Ex2 (
  unsigned int in[NN],
  char a,
  char b,
  unsigned int c,
  unsigned int out[KK]
  ) {

   unsigned int x, y;
   unsigned int tmp;

Loop_j:for(unsigned int j=0; j < MM; j++){
         tmp = b*in[j];
Loop_i:for (unsigned int i=0 ; i < NN; i++) {
        unsigned int k = i+j*NN;
        x = in[i];
        y = a*x + tmp + squared(c);
        out[k] = y;
      }
   }
}

unsigned int squared(unsigned int a)
{
  unsigned int res = 0;
  res = a*a;
  return res;
}
```

```
#ifndef LEC6EX2_H_
#define LEC6EX2_H_

#include <stdio.h>
#include <math.h>

#define NN  30
#define MM  3
#define KK  90

void lec6Ex2(
  unsigned int in[NN],
  char a,
  char b,
  unsigned int c,
  unsigned int out[KK]
  );

unsigned int squared(unsigned int );

#endif
~
~
```

Ex: lec6Ex2 -Synthesis Result

Eg.: lec6Ex2

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.152 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
280	280	2.800 us	2.800 us	280	280	none

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- Loop_j	279	279	93	-	-	3	no
+ Loop_i	90	90	3	-	-	30	no

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	7	0	215	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	60	-
Register	-	-	195	-	-
Total	0	7	195	275	0
Available	650	600	202800	101400	0
Utilization (%)	0	1	~0	~0	0

Ex: lec6Ex2 -Synthesis Result

Eg.: lec6Ex2

Multiplexer

Name	LUT	Input Size	Bits	Total Bits
ap_NS_fsm	29	7	1	7
i_0_reg_120	9	2	5	10
in_r_address0	13	3	5	15
j_0_reg_108	9	2	2	4
Total	60	14	13	36

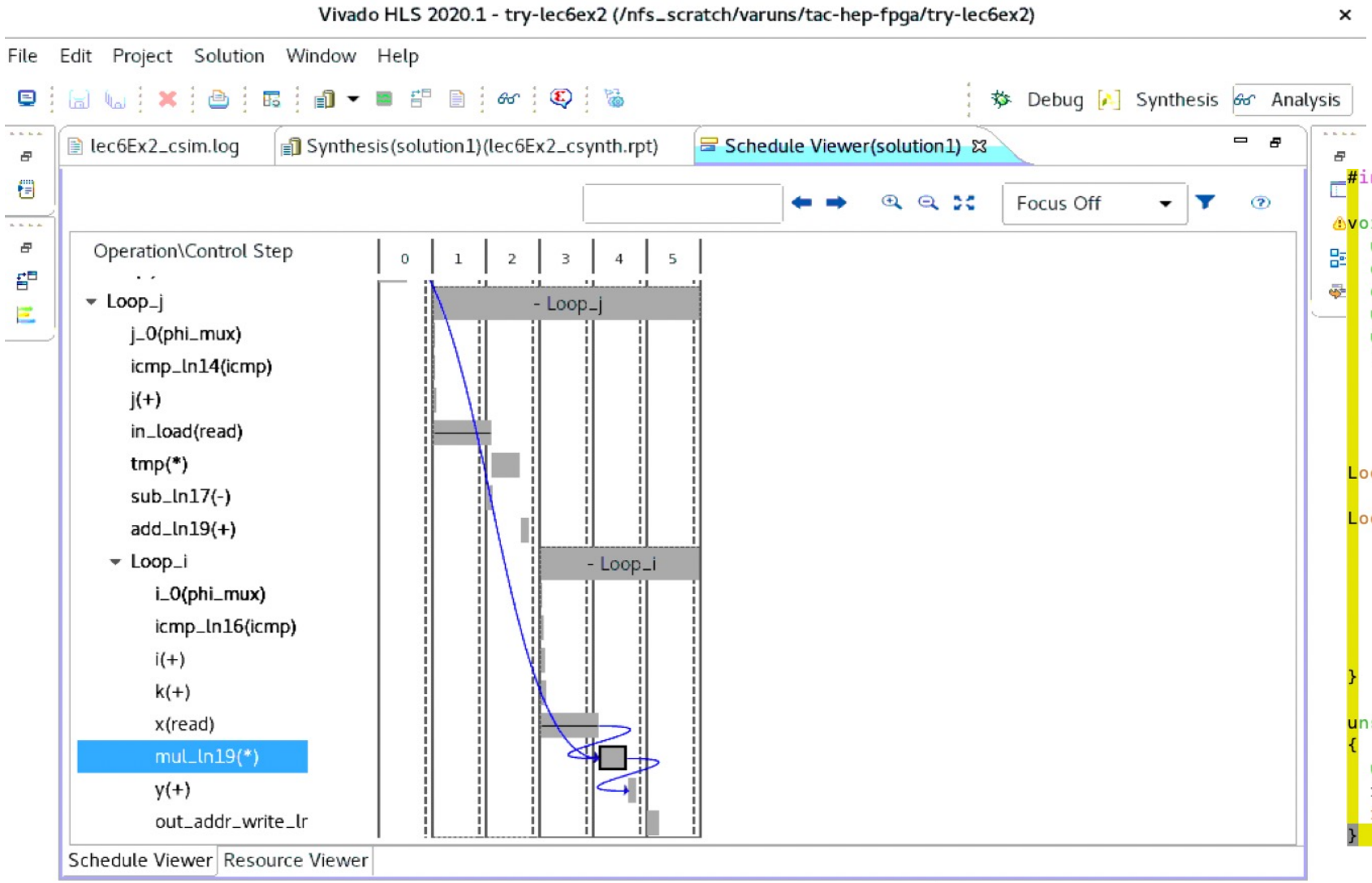
Register

Name	FF	LUT	Bits	Const Bits
add_ln19_reg_279	32	0	32	0
ap_CS_fsm	6	0	6	0
i_0_reg_120	5	0	5	0
i_reg_287	5	0	5	0
j_0_reg_108	2	0	2	0
j_reg_264	2	0	2	0
k_reg_292	8	0	8	0
res_reg_256	32	0	32	0
sext_ln15_reg_246	32	0	32	0
sext_ln19_reg_251	32	0	32	0
sub_ln17_reg_274	7	0	8	1
y_reg_302	32	0	32	0
Total	195	0	196	1

Summary

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	lec6Ex2	return value
ap_rst	in	1	ap_ctrl_hs	lec6Ex2	return value
ap_start	in	1	ap_ctrl_hs	lec6Ex2	return value
ap_done	out	1	ap_ctrl_hs	lec6Ex2	return value
ap_idle	out	1	ap_ctrl_hs	lec6Ex2	return value
ap_ready	out	1	ap_ctrl_hs	lec6Ex2	return value
in_r_address0	out	5	ap_memory	in_r	array
in_r_ce0	out	1	ap_memory	in_r	array
in_r_q0	in	32	ap_memory	in_r	array
a	in	8	ap_none	a	scalar
b	in	8	ap_none	b	scalar
c	in	32	ap_none	c	scalar
out_r_address0	out	7	ap_memory	out_r	array
out_r_ce0	out	1	ap_memory	out_r	array
out_r_we0	out	1	ap_memory	out_r	array
out_r_d0	out	32	ap_memory	out_r	array

Analysis Perspective




```
#include "lec6Ex2.h"

void lec6Ex2 (
  unsigned int in[NN],
  char a,
  char b,
  unsigned int c,
  unsigned int out[KK]
  ) {

   unsigned int x, y;
   unsigned int tmp;

Loop_j:for(unsigned int j=0; j < MM; j++){
         tmp = b*in[j];
Loop_i:for (unsigned int i=0 ; i < NN; i++) {
        unsigned int k = i+j*NN;
        x = in[i];
        y = a*x + tmp + squared(c);
        out[k] = y;
      }
   }
}

unsigned int squared(unsigned int a)
{
  unsigned int res = 0;
  res = a*a;
  return res;
}
```

TAC-HEP 2023

HLS Pragmas

Pragma HLS array_map

#pragma HLS array_map variable=<*name*> instance=<*instance*> <*mode*> offset=<*int*>

- Combines multiple smaller arrays into a single large array to help reduce block RAM resources
  - This larger array can then be targeted to a single larger memory (RAM or FIFO) resource
- Each array is mapped into a block RAM or UltraRAM, when supported by the device
  - The basic block RAM unit provided in an FPGA is 18K
  - If many small arrays do not use the full 18K, a better use of the block RAM resources is to map many small arrays into a single larger array

- The ARRAY_MAP pragma supports two ways of mapping small arrays into a larger one:
  - **Horizontal mapping:** this corresponds to creating a new array by concatenating the original arrays
    - Physically, this gets implemented as a single array with more elements.
  - **Vertical mapping:** this corresponds to creating a new array by concatenating the original words in the array
    - Physically, this gets implemented as a single array with a larger bit-width.

Pragma HLS array_map

#pragma HLS array_map variable=<*name*> instance=<*instance*> <*mode*> offset=<*int*>

variable=<*name*>: A required argument that specifies the array variable to be mapped into the new target array <*instance*>

instance=<*instance*>: Specifies the name of the new array to merge arrays into.

- <*mode*>: Optionally specifies the array map as being either horizontal or vertical.

offset=<*int*>: Applies to horizontal type array mapping only. The offset specifies an integer value offset to apply before mapping the array into the new array <*instance*>. For example:

- Element 0 of the array variable maps to element <*int*> of the new target
- Other elements map to <*int+1*>, <*int*+2>... of the new target.

Ex: Pragma HLS array_map

Arrays array1 and array2 in function foo are mapped into a single array, specified as array3 in the following example:

```
void foo (...) {
int8 array1[M];
int12 array2[N];
#pragma HLS ARRAY_MAP variable=array1 instance=array3 horizontal
#pragma HLS ARRAY_MAP variable=array2 instance=array3 horizontal
...
loop_1: for(i=0;i<M;i++) {
array1[i] = ...;
array2[i] = ...;
...
}
...
}
```

Ex: Pragma HLS array_map

This example provides a horizontal mapping of array A[10] and array B[15] in function foo into a single new array AB[25].

- Element AB[0] will be the same as A[0]
- Element AB[10] will be the same as B[0] because no offset= option is specified.
- The bit-width of array AB[25] will be the maximum bit-width of either A[10] or B[15]

```
#pragma HLS array_map variable=A instance=AB horizontal
#pragma HLS array_map variable=B instance=AB horizontal
```

The following example performs a vertical concatenation of arrays C and D into a new array CD, with the bit-width of C and D combined

The number of elements in CD is the maximum of the original arrays, C or D:

```
#pragma HLS array_map variable=C instance=CD vertical
#pragma HLS array_map variable=D instance=CD vertical
```

Pragma HLS array_partition

- Partitions an array into smaller arrays or individual elements and provides the following:
  - Results in RTL with multiple small memories or multiple registers instead of one large memory
  - Effectively increases the amount of read and write ports for the storage
  - Potentially improves the throughput of the design
  - Requires more memory instances or registers

Syntax:

Place the pragma in the C source within the boundaries of the function where the array variable is defined

#pragma HLS array_partition variable=<name> <type> factor=<int> dim=<int>

Pragma HLS array_partition

#pragma HLS array_partition variable=<name> <type> factor=<int> dim=<int>

variable=<name>: A required argument that specifies the array variable to be partitioned.
<type>: Optionally specifies the partition type (default type is complete)

- **cyclic:** Cyclic partitioning creates smaller arrays by interleaving elements from the original array
  - Partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. For example, if factor=3 is used:
    - Element 0 is assigned to the first new array
    - Element 1 is assigned to the second new array.
    - Element 2 is assigned to the third new array.
    - Element 3 is assigned to the first new array again.

- **block:** Block partitioning creates smaller arrays from consecutive blocks of the original array. This effectively splits the array into N equal blocks, where N is the integer defined by the factor= argument

- **complete:** Complete partitioning decomposes the array into individual elements
  - For a 1-D array, this corresponds to resolving a memory into individual registers (default <type>)

Pragma HLS array_partition

#pragma HLS array_partition variable=<name> <type> factor=<int> dim=<int>

factor=<*int*>: Specifies the number of smaller arrays that are to be created

NOTE: For complete type partitioning, the factor is not specified. Must for block and cyclic partitioning

dim=<*int*>: Specifies which dimension of a multi-dimensional array to partition. Specified as an integer from 0 to *N*, for an array with *N* dimensions:

- If a value of 0 is used, all dimensions of a multi-dimensional array are partitioned with the specified type and factor options.
- Any non-zero value partitions only the specified dimension. For example, if a value 1 is used, only the first dimension is partitioned.

Ex: Pragma HLS array_partition

#pragma HLS array_partition variable=AB block factor=4

- This example partitions the 13 element array, AB[13], into four arrays using block partitioning:
  - Because four is not an integer factor of 13:
  - Three of the new arrays have three elements each,
  - One array has four elements (AB[9:12])

#pragma HLS array_partition variable=AB block factor=2 dim=2

- This example partitions dimension two of the two-dimensional array, AB[6][4] into two new arrays of dimension [6][2]:

Pragma HLS unroll

- Unroll loops to create multiple independent operations rather than a single collection of operations
- **UNROLL** pragma transforms loops by creating multiples copies of the loop body in the RTL design, which allows some or all loop iterations to occur in parallel
- Loops in the C/C++ functions are kept rolled by default
  - When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence

- **UNROLL** pragma allows the loop to be fully or partially unrolled
  - Fully unrolling the loop creates a copy of the loop body in the RTL for each loop iteration, so the entire loop can be run concurrently
  - Partially unrolling a loop lets you specify a factor *N*

Pragma HLS unroll

#pragma HLS unroll factor=<N> region skip_exit_check

factor=<*N*>: Specifies a non-zero integer indicating that partial unrolling is requested.

- If factor= is not specified, the loop is fully unrolled.

region: An optional keyword that unrolls all loops within the body (region) of the specified loop, without unrolling the enclosing loop itself.

skip_exit_check: An optional keyword that applies only if partial unrolling is specified with factor=

- Fixed (known) bounds: No exit condition check is performed if the iteration count is a multiple of the factor. If the iteration count is not an integer multiple of the factor, the tool:
  - Prevents unrolling.
  - Issues a warning that the exit check must be performed to proceed.
- Variable (unknown) bounds: The exit condition check is removed as requested. You must ensure that:
  - The variable bounds is an integer multiple of the specified unroll factor.
  - No exit check is in fact require

Ex: Pragma HLS unroll

#pragma HLS unroll factor=<N> region skip_exit_check

The following example fully unrolls loop_1 in function *foo*

Place the pragma in the body of loop_1 as shown:

```
loop_1: for(int i = 0; i < N; i++) {
  #pragma HLS unroll
  a[i] = b[i] + c[i];
}
```

This example specifies an unroll factor of 4 to partially unroll loop_2 of function foo, and removes the exit check:

```
void foo (...) {
  int8 array1[M];
  int12 array2[N];
  ...
  loop_2: for(i=0;i<M;i++) {
    #pragma HLS unroll skip_exit_check factor=4
    array1[i] = ...;
    array2[i] = ...;
    ...
  }
  ...
}
```

Ex: Pragma HLS unroll

The following example fully unrolls all loops inside loop_1 in function foo, but not loop_1 itself due to the presence of the region keyword:

```
void foo(int data_in[N], int scale, int data_out1[N], int data_out2[N]) {
  int temp1[N];
  loop_1: for(int i = 0; i < N; i++) {
    #pragma HLS unroll region
    temp1[i] = data_in[i] * scale;
      loop_2: for(int j = 0; j < N; j++) {
        data_out1[j] = temp1[j] * 123;
      }
      loop_3: for(int k = 0; k < N; k++) {
        data_out2[k] = temp1[k] * 456;
      }
  }
}
```

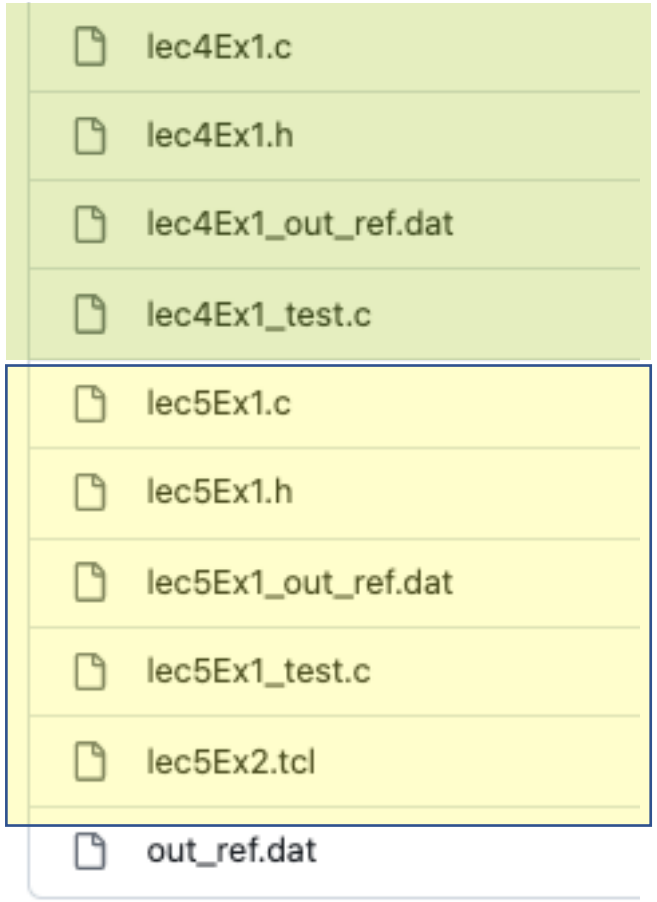
Ex: lec6Ex2a

```cpp
#include "lec6Ex2a.h"

void lec6Ex2a (
  ap_uint<8> in[NN],
  ap_uint<2> a,
  ap_uint<2> b,
  ap_uint<2> c,
  ap_uint<8> out[KK]
  ) {

#pragma HLS ARRAY_PARTITION variable=in complete dim=0
#pragma HLS ARRAY_PARTITION variable=out complete dim=0

   ap_uint<8> x, y;
   ap_uint<8> tmp;

Loop_j:for(ap_uint<2> j=0; j < MM; j++){
#pragma LOOP UNROLL
         tmp = b*in[j];

Loop_i:for (ap_uint<5> i=0 ; i < NN; i++) {
#pragma LOOP UNROLL
        ap_uint<7> k = i+j*NN;
        x = in[i];
        y = a*x + tmp + squared(c);
        out[k] = y;
      }
   }
}

ap_uint<8> squared(ap_uint<8> a)
{
#pragma HLS INLINE
  ap_uint<8> res = 0;
  res = a*a;
  return res;
}
~
~
```

```cpp
#ifndef LEC6EX2_H_
#define LEC6EX2_H_

#include <stdio.h>
#include <math.h>
#include "ap_int.h"

#define NN  30
#define MM  3
#define KK  90

void lec6Ex2a(
  ap_uint<8> in[NN],
  ap_uint<2> a,
  ap_uint<2> b,
  ap_uint<2> c,
  ap_uint<8> out[KK]
  );

ap_uint<8> squared(ap_uint<8> );

#endif
~
```

Exercise files

- Git clone: https://github.com/varuns23/TAC-HEP-FPGA-HLS.git



Assignment

- Use target device: **xc7k160tfbg484-2**
- Clock period of 10ns

1. Execute the code (lec5Ex2.tcl) using CLI (slide-25) and compare the results with GUI results for C-Simulation, C-Synthesis

2. Vary following parameters for two cases: high and very high values and compare with 1 for both CLI and GUI
    - Variable: "samples"
    - Variable: "N"
3. Run example lec3Ex2a

Assignment submission

- Where to submit:
  - https://pages.hep.wisc.edu/~varuns/assignments/TAC-HEP/
- Use your login machine credentials
- Submit one file per week
  - Week-2 & 3 can be merged together
- Try to submit by next Tuesday

TAC-HEP 2023

Questions?

TAC-HEP 2023

Additional material

Correct Time

From 03.28.2023 onwards

- Tuesdays: 9:00-10:00 CT / 10:00-11:00 ET / 16:00-17:00 CET
- Wednesday: 11:00-12:00 CT / 12:00-13:00 ET / 18:00-19:00 CET

Jargons

- **ICs - Integrated chip:** assembly of hundreds of millions of transistors on a minor chip
- **PCB:** Printed Circuit Board
- **LUT - Look Up Table aka 'logic'** - generic functions on small bitwidth inputs. Combine many to build the algorithm
- **FF - Flip Flops** - control the flow of data with the clock pulse. Used to build the pipeline and achieve high throughput
- **DSP - Digital Signal Processor** - performs multiplication and other arithmetic in the FPGA
- **BRAM - Block RAM** - hardened RAM resource. More efficient memories than using LUTs for more than a few elements
- **PCIe or PCI-E - Peripheral Component Interconnect Express**: is a serial expansion bus standard for connecting a computer to one or more peripheral devices
- **InfiniBand** is a computer networking communications standard used in high-performance computing that features very high throughput and very low latency
- **HLS -** High Level Synthesis - compiler for C, C++, SystemC into FPGA IP cores
- **DRCs -** Design Rule Checks
- **HDL** - Hardware Description Language - low level language for describing circuits
- **RTL** - Register Transfer Level - the very low level description of the function and connection of logic gates
- **FIFO** – First In First Out memory
- **Latency** - time between starting processing and receiving the result
  - Measured in clock cycles or seconds
- **II - Initiation Interval** - time from accepting first input to accepting next input