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

GPU & FPGA module training: Part-2

Week-4: Vivado HLS: *Pragmas & more examples*

Lecture-7: April 11th 2023

Varun Sharma

University of Wisconsin – Madison, USA

WISCONSIN
UNIVERSITY OF WISCONSIN-MADISON

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, Introduction to Pragmas

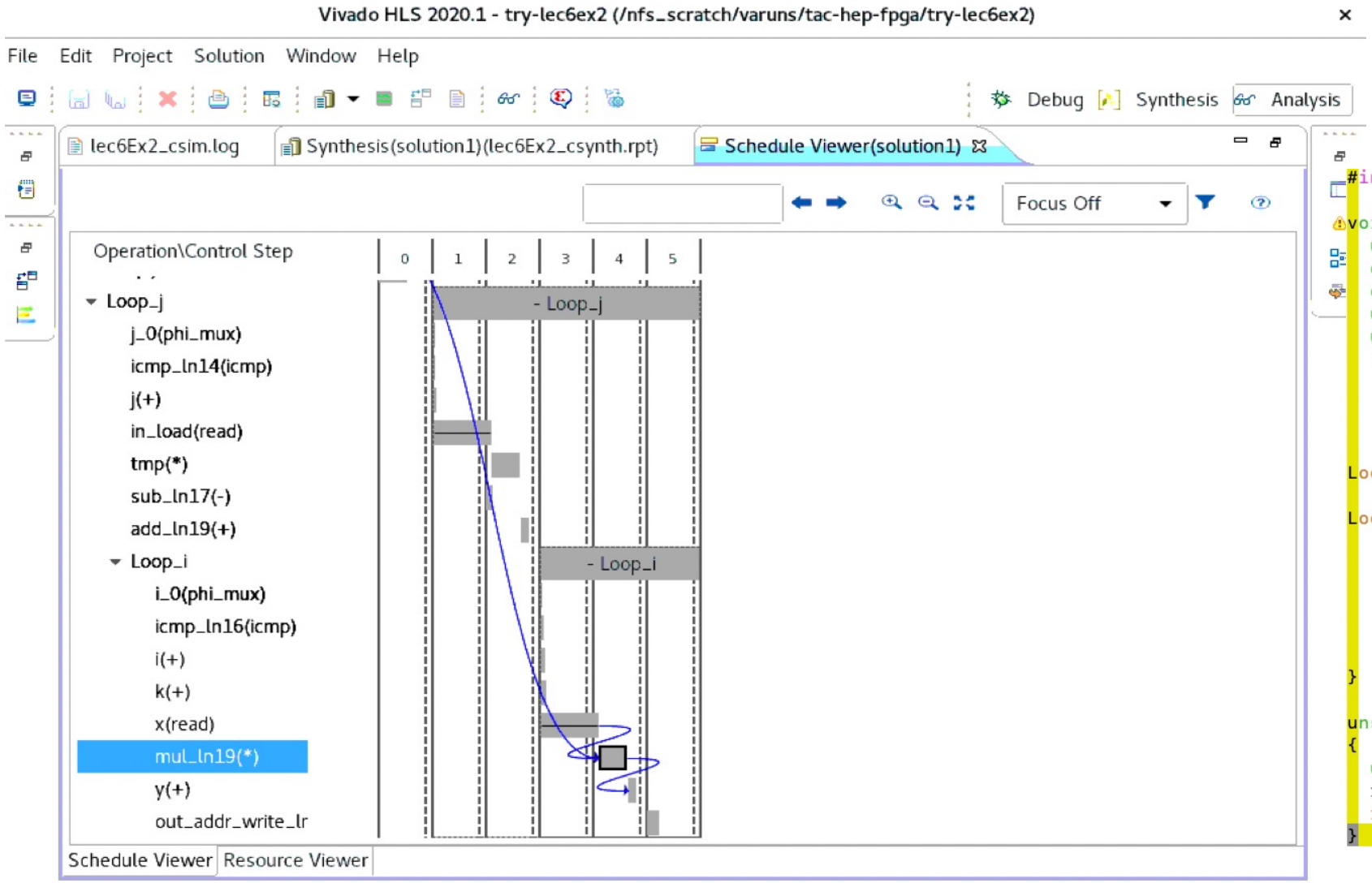
Today:

- Continue with Pragmas with some examples

TAC-HEP 2023

HLS Pragmas [Ref]

Why are they needed?




```
#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;
}
```

Pragmas by type

Type	Attributes	
Kernel Optimization	pragma HLS allocation pragma HLS expression_balance pragma HLS latency	pragma HLS reset pragma HLS resource pragma HLS stable
Function Inlining	pragma HLS inline pragma HLS function_instantiate	
Interface Synthesis	pragma HLS interface	
Task-level Pipeline	pragma HLS dataflow pragma HLS stream	
Pipeline	pragma HLS pipeline pragma HLS occurrence	
Loop Unrolling	pragma HLS unroll pragma HLS dependence	
Loop Optimization	pragma HLS loop_flatten pragma HLS loop_merge	pragma HLS loop_tripcount
Array Optimization	pragma HLS array_map pragma HLS array_partition	pragma HLS array_reshape
Structure Packing	pragma HLS data_pack	

Pragma HLS allocation

- Specifies instance restrictions to limit resource allocation in the implemented kernel
- Defines & can limit the number of RTL instances and hardware resources used to implement specific functions, loops, operations or cores

- Example: c-source code has 4 instances of a function *my_func*
  - ALLOCATION pragma can ensure that there is only one instance of of *my_func*
  - All 4 instances are implemented using the same RTL block
    - Reduces resource used by function but may impact performance

- **Operations:** additions, multiplications, array reads, & writes can be limited by ALLOCATION pragma

Pragma HLS allocation - Syntax

#pragma HLS allocation instances=<list> limit=<value> <type>

- **Instance<list>*:** Name of the function, operator, or cores
- **limit=<value>*:** Specifies the limit of instances to be used in kernel
- **<type>*:** Specifies the allocation applies to a function, an operator or a core (hardware component) used to create the design (such as adder, multiplier, BRAM)
  - *Function:* allocation applies to the functions listed in the instances=
  - *Operation:* applies to the operations listed in the instances=
  - *Core:* applies to the cores

Pragma HLS allocation - Example

#pragma HLS allocation instances=<list> limit=<value> <type>

Example1: Limits the number of instances of my_func in the RTL for hardware kernel to 1

```
void top { a, b, c, d} {
#pragma HLS ALLOCATION instances=my_func limit=1 function
 ...
 my_func(a,b); //my_func_1
 my_func(a,c); //my_func_2
 my_func(a,d); //my_func_3
 ...
}
```

Example2: Limits the number of multiplier operation used in the implementation of the function *my_func* to 1

- Limit does NOT apply outside the function
- Alternatively, inline the sub-function can also do similar job

```
void my_func(data_t angle) {
#pragma HLS allocation instances=mul limit=1 operation
...
}
```

Pragma HLS Latency

#pragma HLS latency min=<int> max=<int>

- Specifies a minimum or maximum latency value, or both, for the completion of functions, loops, and regions
  - *min=<int>*: minimum latency for the function, loop, or region of code
  - *max=<int>*: maximum latency for the function, loop, or region of code

- **Latency:** # of CLK cycles required to produce an output

- **Function latency:** # of CLK cycles required for the function to computee all output values and return

- **Loop latency**: # of CLK cycles to execute all iterations of the loop

Pragma HLS Latency

#pragma HLS latency min=<int> max=<int>

- HLS always tries to minimize latency in the design

- When LATENCY pragma is specified
  - ***Min < Latency < Max***: Constraint is satisfied, No further optimization
  - ***Latency < min***: It extends latency to the specified value, potentially increasing sharing
  - ***Latency > max***: Increases effort to achieve the constraints
    - Still unsuccessful: issue a warning & produce design with the smallest achievable latency in excess of maximum

Pragma HLS Latency - Example

#pragma HLS latency min=<int> max=<int>

Example-1: Function foo is specified to have a minimum latency of 4 and a maximum latency of 8

```
int foo(char x, char a, char b, char c) {
  #pragma HLS latency min=4 max=8
  char y;
  y = x*a+b+c;
  return y
}
```

Example-2: loop_1 is specified to have a maximum latency of 12

```
void foo (num_samples, ...) {
  int i;
  ...
  loop_1: for(i=0;i< num_samples;i++) {
  #pragma HLS latency max=12
    ...
    result = a + b;
  }
}
```

Example-3: Creates a code region and groups signals that need to change in the same clock cycle by specifying zero latency

```
// create a region { } with a latency = 0
{
  #pragma HLS LATENCY max=0 min=0
  *data = 0xFF;
  *data_vld = 1;
}
```

Pragma HLS Resource

#pragma HLS resource variable=<variable> core=<core> latency=<int>

- Specifies that a specific library resource (core) is used to implement a variable (array, arithmetic operation, or function argument) in RTL
  - If not specified: HLS determines for you
  - Specially useful, when multiple cores in the library can implement the operation

- ***variable=<variable>*:*** *Specifies the array, arithmetic operation, or function argument to assign the RESOURCE pragma to*

- ***core=<core>*:*** Specifies the core, as defined in the technology library

- ***latency=<int>*:** Specifies the latency of the core

Pragma HLS Resource - Example

Example-1:

- <coeffs[128]> variable is an argument to the top-level function *top*
- Specifies that coeffs is implemented with core *RAM_1P* from the library

```
void top (int in_[128], int coeffs[128], int out_[128] ) {
#pragma HLS resource variable=coeffs core=RAM_1P
...
}
```

Example-2:

- Two-stage pipelined multiplier is specified to implement the multiplication for variable <c> of the function foo
- The HLS tool selects the core to use for variable <d>

```
int foo (int a, int b) {
int c, d;
#pragma HLS RESOURCE variable=c latency=2
c = a*b;
d = a*c;
return d;
}
```

Pragma HLS Dataflow

#pragma HLS dataflow

- Enables task-level pipelining: allow functions and loops to overlap in their operation
  - Increases the concurrency of the RTL implementation & thus the overall throughput of the design
- In the absence of any directives that limit resources (like pragma HLS allocation), HLS seeks to minimize latency & improve concurrency
  - Data dependencies can limit this, hence proper dataflow is needed

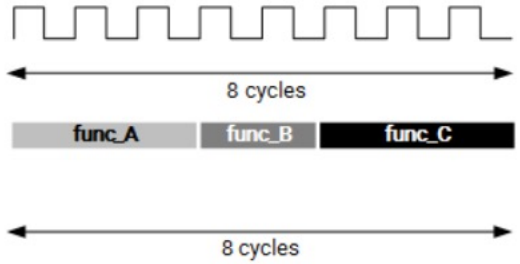
Example:

- Functions/loops that access arrays must finish all read/write accesses to the arrays before they complete
- Prevent the next function or loop that consumes the data from starting operation
- The DATAFLOW optimization enables the operations in a function or loop to start operation before the previous function or loop completes all its operations

Pragma HLS Dataflow - Example

Task-level pipeline

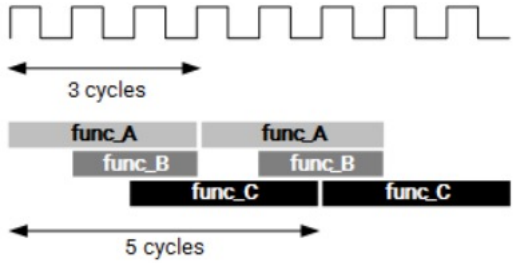
#pragma HLS dataflow



Without DATAFLOW pipelining

```
void top(a, b, c, d){
    ...
    func_A(a,b,i1);
    func_B(c,i1,i2);
    func_C(i2,d);

    ...
    return d;
}
```



With DATAFLOW pipelining

For the DATAFLOW optimization to work, the data must flow through the design from one task to the next

```
wr_loop_j: for (int j = 0; j < TILE_PER_ROW; ++j) {
#pragma HLS DATAFLOW
   wr_buf_loop_m: for (int m = 0; m < HEIGHT; ++m) {
      wr_buf_loop_n: for (int n = 0; n < WIDTH; ++n) {
      #pragma HLS PIPELINE
      // should burst WIDTH in WORD beat
         outFifo >> tile[m][n];
      }
   }
   wr_loop_m: for (int m = 0; m < HEIGHT; ++m) {
      wr_loop_n: for (int n = 0; n < WIDTH; ++n) {
      #pragma HLS PIPELINE
         outx[HEIGHT*TILE_PER_ROW*WIDTH*i+TILE_PER_ROW*WIDTH*m+WIDTH*j+n] = tile[m][n];
      }
   }
}
```

- ✗ Bypassing tasks
- ✗ Feedback between tasks
- ✗ Conditional execution of tasks
- ✗ Loops with multiple exit conditions

Pragma HLS Dataflow - Example

Task-level pipeline

#pragma HLS dataflow

```
void top(a, b, c, d){
```

- ✓ HLS tool issues a message and does not perform DATAFLOW optimization
- ✓ Use the STABLE pragma to mark variables within DATAFLOW regions to be stable to avoid concurrent read or write of variables.
- ✓ No hierarchial implementation

```
wr_loop_j: for (int j = 0; j < TILE_PER_ROW; ++j) {
#pragma HLS DATAFLOW
   wr_buf_loop_m: for (int m = 0; m < HEIGHT; ++m) {
      wr_buf_loop_n: for (int n = 0; n < WIDTH; ++n) {
      #pragma HLS PIPELINE
      // should burst WIDTH in WORD beat
         outFifo >> tile[m][n];
      }
   }
   wr_loop_m: for (int m = 0; m < HEIGHT; ++m) {
      wr_loop_n: for (int n = 0; n < WIDTH; ++n) {
      #pragma HLS PIPELINE
         outx[HEIGHT*TILE_PER_ROW*WIDTH*i+TILE_PER_ROW*WIDTH*m+WIDTH*j+n] = tile[m][n];
      }
   }
}
```

- ✗ Bypassing tasks
- ✗ Feedback between tasks
- ✗ Conditional execution of tasks
- ✗ Loops with multiple exit conditions

Pragma HLS Stable

#pragma HLS stable variable=<name>

- The STABLE pragma marks variables within a DATAFLOW region as being stable
- Applies to both scalar and array variables whose content can be written/read by the process inside the DATAFLOW region
- Eliminates the extra synchronization involved for DATAFLOW region

Example:

- Specifies the array A as stable
- If A is read by proc2, then it will not be written by another process while the DATAFLOW region is being executed

```
void foo(int A[...], int B[...]){
#pragma HLS dataflow
#pragma HLS stable variable=A
  proc1(...);
  proc2(A, ...);

...
}
```

Pragma HLS Inline

#pragma HLS inline <region | recursive | off>

- Removes a function as a separate entity in the hierarchy
- The function is dissolved into the calling function and no longer appears as a separate level of hierarchy in RTL design
- May improve area by allowing the components within the function to be better shared or optimized with the logic in the calling function

- ***Region***: Optionally, all functions (sub-functions) in the specified region are to be inlined

- ***Recursive***: Inlines all functions recursively within the specified function or region
  - By default, only one level of function inlining is performed

- ***Off:*** Disables function inlining to prevent specified functions from being inlined
  - For example, HLS automatically inlines small functions & with the off option, automatic inlining can be prevented

Pragma HLS Inline - Example

#pragma HLS inline <region | recursive | off>

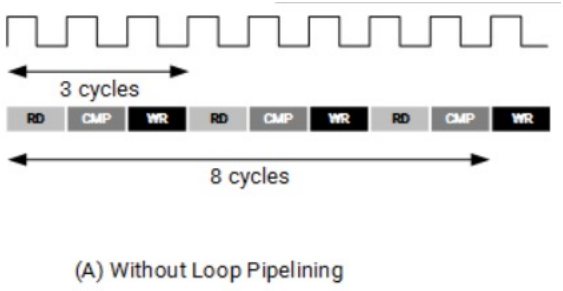
- Inlines all functions within the body of *foo_top*
- Inlining recursively down through the function hierarchy, except function foo_sub is not inlined.
- The recursive pragma is placed in function *foo_top*
- The pragma to disable inlining is placed in the function foo_sub

```
foo_sub (p, q) {
#pragma HLS inline off
  int q1 = q + 10;
  foo(p1,q);// foo_3
  ...
}
void foo_top { a, b, c, d} {
#pragma HLS inline region recursive
  ...
  foo(a,b);//foo_1
  foo(a,c);//foo_2
  foo_sub(a,d);
  ...
}
```

Pragma HLS Pipeline

#pragma HLS pipeline II=<int>

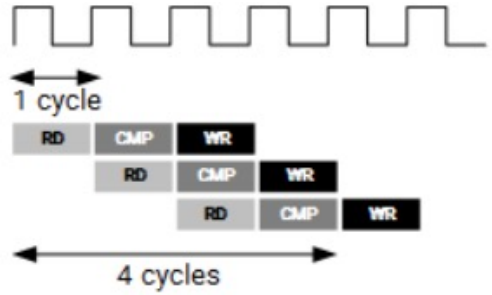
- The PIPELINE pragma reduces the II for a function or loop by allowing the concurrent execution of operations
- A pipelined function or loop can process new inputs every <N> clock cycles
- If HLS can't create a design with the specified II, it issues a warning and creates a design with the lowest possible II



(A) Without Loop Pipelining

Without Loop pipelining

```
void func(input, output){
...
  for(i=0; i>=N; i++){
#pragma HLS pipeline II=2
    op_read;
    op_compute;
    op_write;
  }
...
}
```



With Loop pipelining

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.

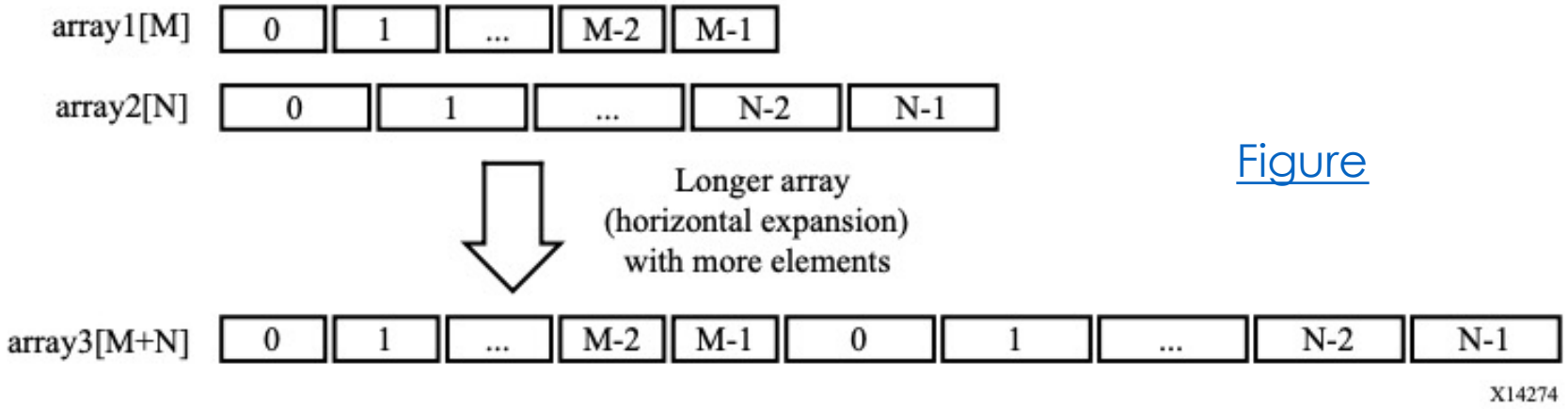
Pragma HLS array_map: Horizontal

#pragma HLS array_map variable=<*name*> instance=<*instance*> **horizontal**

Horizontal mapping: Creating a new array by concatenating the original arrays

Implemented as a single array with more elements

```
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] = ...;
 ...
}
...
}
```



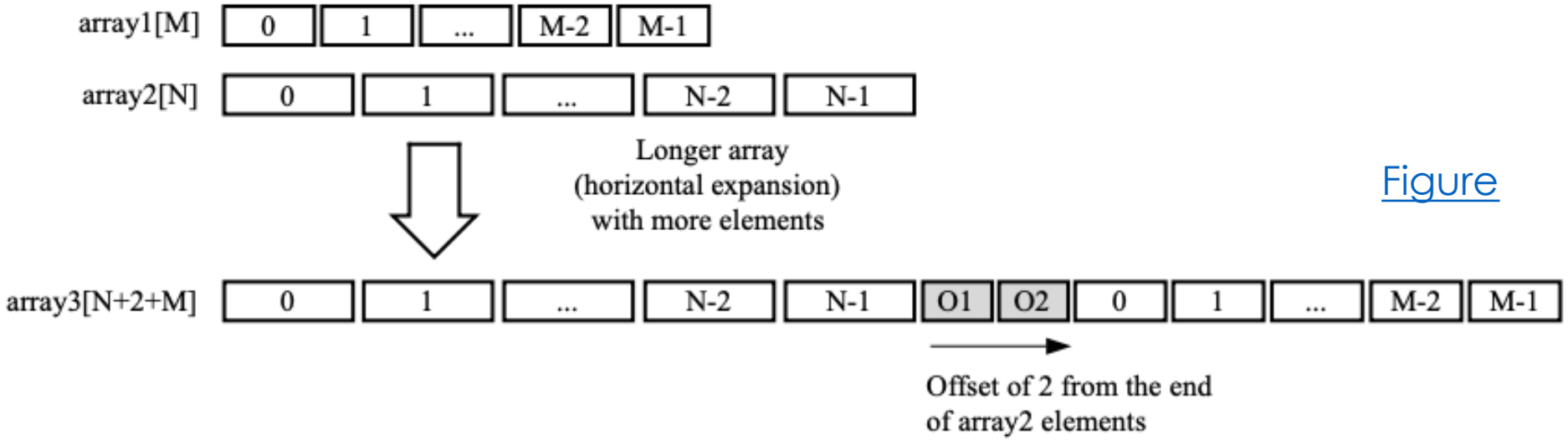
Figure

Pragma HLS array_map: Horizontal

#pragma HLS array_map variable=<*name*> instance=<*instance*> **horizontal offset=2**

The **offset** option to the ARRAY_MAP directive is used to specify at which location subsequent arrays are added when using the horizontal option

```
void foo (...) {
 int8 array1[M];
 int12 array2[N];
#pragma HLS ARRAY_MAP variable=array2 instance=array3 horizontal
#pragma HLS ARRAY_MAP variable=array1 instance=array3 horizontal offset=2
 ...
loop_1: for(i=0;i<M;i++) {
 array1[i] = ...;
 array2[i] = ...;
 ...
}
...
}
```



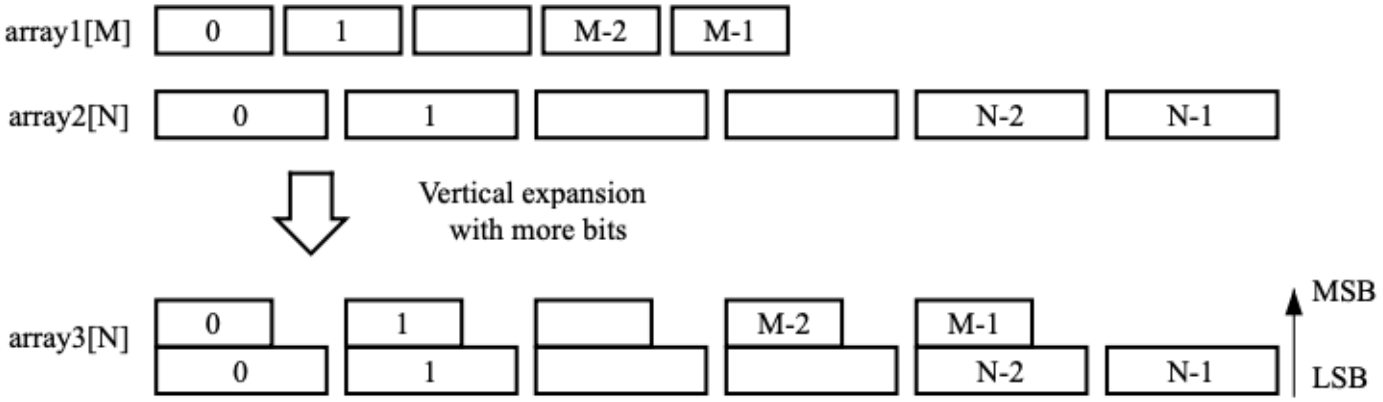
Figure

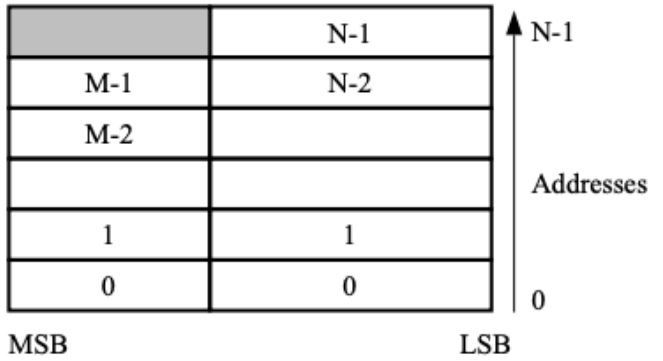
Pragma HLS array_map: Vertical

#pragma HLS array_map variable=<*name*> instance=<*instance*> **vertical**

Vertical mapping: Arrays are concatenated by to produce an array with higher bitwidths

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



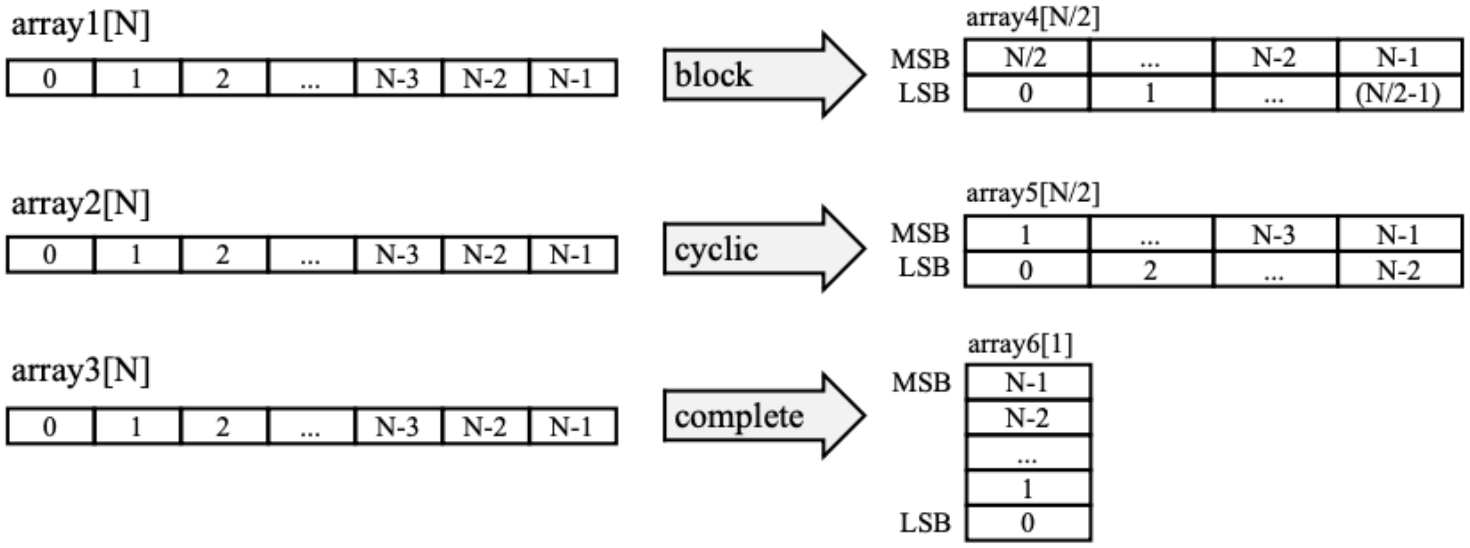


Figure

Pragma HLS array_partition

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

- **Cyclic:** Cyclic partitioning creates smaller arrays by interleaving elements from the original array
- **Block:** Block partitioning creates smaller arrays from consecutive N-blocks of the original array
- **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>)




```
void foo (...) {
int array1[N];
int array2[N];
int array3[N];
#pragma HLS ARRAY_PARTITION variable=array1 block factor=2 dim=1
#pragma HLS ARRAY_PARTITION variable=array2 cycle factor=2 dim=1
#pragma HLS ARRAY_PARTITION variable=array3 complete dim=1
...
}
```

Figure

Summary

- Pragmas are important to implement a design in best possible ways
- There are a lot of pragmas to help the design implementation
- Be careful with the choice of pragma's to avoid conflicts

Assignment Week-4

1. Do a matrix multiplication of two 1-dimensional arrays – A[N]*B[N], where N > 5
   a) Report synthesis results without any pragma directives
   b) Add as many pragma directives possible
      i. Report any conflicts (if reported in logs) between two pragmas

TAC-HEP 2023

Questions?

TAC-HEP 2023

Additional material

Assignment submission

- Where to submit:
  - https://pages.hep.wisc.edu/~varuns/assignments/TAC-HEP/
- Use your login machine credentials
- Submit one file per week
- Try to submit by following week's Tuesday

Correct Time

From 03.28.2023 onwards

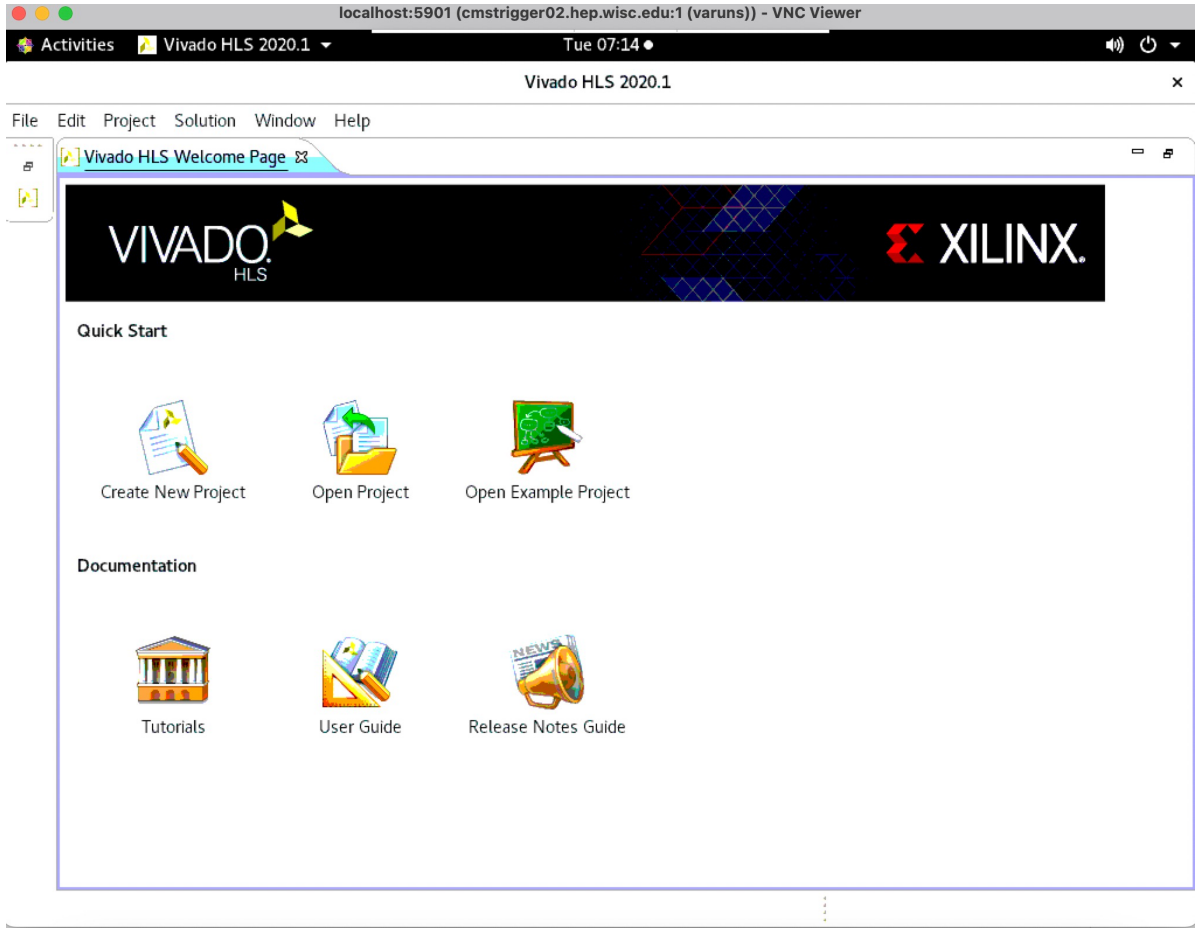
- Tuesdays: 9:00-10:00 CT / 10:00-11:00 ET / 16:00-17:00 CET
- Wednesday: 11:00-12:00 CT / 12:00-13:00 ET / 18:00-19:00 CET

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

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**



Assignment Week-3

- 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