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

FPGA module training

Lecture-8: March 24th 2026

Varun Sharma

University of Wisconsin – Madison, USA

Content

So far

- Motivation & Introduction
- Comparison: FPGAs/ASICs/GPU/CPU
- Domain specific Accelerators
- HLS setup and first example project
- Unsupported C/C++ constructs
- Data types
- HLS Pragmas
  - Interface
  - Array partition

Today

- HLS Pragmas
  - Array Reshape
  - Pipeline
  - DataFlow

HLS Pragmas

"Pragmas": Instructions to tell your compiler how to build the hardware

- HLS tool provides different set of pragmas that can be used to optimize the design, reduce latency, improve performance etc. These pragmas can be directly added to the source code for the kernel.

Type	Attributes
Kernel Optimization	• pragma HLS aggregate • pragma HLS alias • pragma HLS disaggregate • pragma HLS expression_balance • pragma HLS latency • pragma HLS performance • pragma HLS protocol • pragma HLS reset • pragma HLS top • pragma HLS stable
Function Inlining	• pragma HLS inline
Interface Synthesis	• pragma HLS interface • pragma HLS stream
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_partition • pragma HLS array_reshape
Structure Packing	• pragma HLS aggregate • pragma HLS dataflow
Resource Utilization	• pragma HLS allocation • pragma HLS bind_op • pragma HLS bind_storage • pragma HLS function_instantiate

https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/HLS-Pragmas

TAC-HEP 2026

#pragma HLS array_rehape

https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-array_reshape

Pragma HLS array_reshape

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

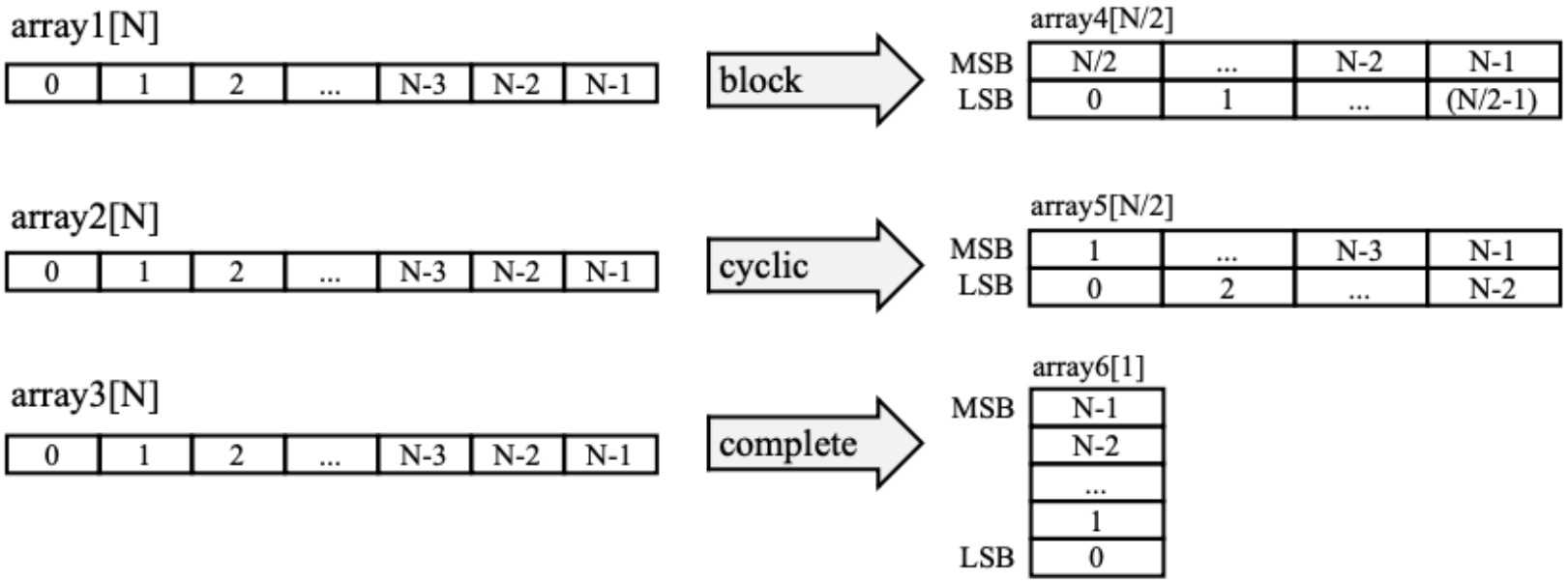
- ARRAY_RESHAPE pragma reforms the array with vertical remapping and concatenating elements of arrays by increasing bit-widths
- Reduces the number of block RAM consumed while providing parallel access to the data
- Pragma creates a new array with fewer elements but with greater bit-width, allowing more data to be accessed in a single clock cycle

Pragma HLS array_reshape

#pragma HLS array_reshape 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



Pragma HLS array_reshape: Complete

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

```
#include "example.h"

dout_t example(din_t mem[N]) {

#pragma HLS ARRAY_RESHAPE variable=mem complete

   dout_t total = 0;

   for (int i = 2; i < N; ++i)
      total += mem[i] + mem[i - 1] + mem[i - 2];

   return total;
}
```

```
Timing:
  * Summary:
  +--------+----------+----------+------------+
  |  Clock |  Target  | Estimated| Uncertainty|
  +--------+----------+----------+------------+
  |ap_clk  | 25.00 ns | 5.988 ns |   3.12 ns  |
  +--------+----------+----------+------------+

Latency:
  * Summary:
  +---------+---------+----------+----------+-----+-----+---------+
  |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
  |   min   |   max   |    min   |    max   | min | max |   Type  |
  +---------+---------+----------+----------+-----+-----+---------+
  |       63|       63| 1.575 us | 1.575 us |   63|   63|   none  |
  +---------+---------+----------+----------+-----+-----+---------+
```

```
* Loop:
+----------+---------+---------+----------+----------+----------+------+----------+
|          |  Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
| Loop Name|   min   |   max   |  Latency |  achieved |   target  | Count| Pipelined|
+----------+---------+---------+----------+----------+----------+------+----------+
|- Loop 1  |       62|       62|         1|         -|         -|    62|    no    |
+----------+---------+---------+----------+----------+----------+------+----------+
```

```
+---------------------+---------+-------+---------+---------+-----+
|         Name        | BRAM_18K| DSP48E|    FF   |   LUT   | URAM|
+---------------------+---------+-------+---------+---------+-----+
|DSP                  |        -|      -|        -|        -|    -|
|Expression           |        -|      -|        0|     5897|    -|
|FIFO                 |        -|      -|        -|        -|    -|
|Instance             |        -|      -|        -|        -|    -|
|Memory               |        -|      -|        -|        -|    -|
|Multiplexer          |        -|      -|        -|       33|    -|
|Register             |        -|      -|       19|        -|    -|
+---------------------+---------+-------+---------+---------+-----+
|Total                |        0|      0|       19|     5930|    0|
+---------------------+---------+-------+---------+---------+-----+
|Available SLR        |     1440|   2280|   788160|   394080|  320|
+---------------------+---------+-------+---------+---------+-----+
|Utilization SLR (%)  |        0|      0|    ~0   |        1|    0|
+---------------------+---------+-------+---------+---------+-----+
|Available            |     4320|   6840|  2364480|  1182240|  960|
+---------------------+---------+-------+---------+---------+-----+
|Utilization (%)      |        0|      0|    ~0   |    ~0   |    0|
+---------------------+---------+-------+---------+---------+-----+
```

TAC-HEP 2026

array_partition vs array_reshape

PARTITION vs RESHAPE

	Pragma HLS PARTITION			Pragma HLS RESHAPE		
	Complete	Block factor = 4	Cyclic factor = 4	Complete	Block factor = 4	Cyclic factor = 4
Estimated Clock (ns)	4.339	3.142	3.998	5.988	2.969	3.117
Latency (cycle)	63	187	187	63	187	187
Latency (μs)	1.575	4.675	4.675	1.575	4.675	4.675
Resources (FF)	19	42	37	19	46	42
Resources (LUT)	920	296	284	5930	488	476
Resource (Others)	0	0	0	0	0	0

PARTITION vs RESHAPE: Expression

PARTITION

* Expression:

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
i_fu_1056_p2	+	0	0	15	7	1
ret_V_1_fu_1040_p2	+	0	0	16	9	9
ret_V_fu_1026_p2	+	0	0	15	8	8
total_V_fu_1050_p2	+	0	0	17	10	10
icmp_ln9_fu_938_p2	icmp	0	0	11	7	8
ap_condition_486	or	0	0	2	1	1
Total		0	0	76	42	37

* Multiplexer:

Name	LUT	Input Size	Bits	Total Bits
agg_result_V_0_reg_656	9	2	10	20
ap_NS_fsm	15	3	1	3
ap_phi_mux_phi_ln215_2_phi_fu_811_p124	269	63	7	441
ap_phi_mux_phi_ln215_phi_fu_681_p124	269	63	7	441
i_0_reg_667	9	2	7	14
Total	571	133	32	919

RESHAPE

* Expression:

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
add_ln215_1_fu_154_p2	+	0	0	15	3	6
add_ln215_fu_103_p2	+	0	0	15	2	6
i_fu_211_p2	+	0	0	15	1	7
ret_V_1_fu_195_p2	+	0	0	16	9	9
ret_V_fu_144_p2	+	0	0	15	8	8
total_V_fu_205_p2	+	0	0	17	10	10
sub_ln215_1_fu_121_p2	-	0	0	16	9	9
sub_ln215_2_fu_172_p2	-	0	0	16	9	9
sub_ln215_fu_80_p2	-	0	0	16	9	9
icmp_ln9_fu_58_p2	icmp	0	0	11	7	8
lshr_ln215_1_fu_131_p2	lshr	0	0	1915	448	448
lshr_ln215_2_fu_182_p2	lshr	0	0	1915	448	448
lshr_ln215_fu_90_p2	lshr	0	0	1915	448	448
Total		0	0	5897	1411	1425

* Multiplexer:

Name	LUT	Input Size	Bits	Total Bits
agg_result_V_0_reg_36	9	2	10	20
ap_NS_fsm	15	3	1	3
i_0_reg_47	9	2	7	14
Total	33	7	18	37

PARTITION vs RESHAPE: Interface

PARTITION

* Summary:

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	example	return value
ap_rst	in	1	ap_ctrl_hs	example	return value
ap_start	in	1	ap_ctrl_hs	example	return value
ap_done	out	1	ap_ctrl_hs	example	return value
ap_idle	out	1	ap_ctrl_hs	example	return value
ap_ready	out	1	ap_ctrl_hs	example	return value
ap_return	out	10	ap_ctrl_hs	example	return value
mem_0_V	in	7	ap_none	mem_0_V	pointer
mem_1_V	in	7	ap_none	mem_1_V	pointer
mem_2_V	in	7	ap_none	mem_2_V	pointer
mem_3_V	in	7	ap_none	mem_3_V	pointer
mem_4_V	in	7	ap_none	mem_4_V	pointer
mem_5_V	in	7	ap_none	mem_5_V	pointer
mem_6_V	in	7	ap_none	mem_6_V	pointer
mem_7_V	in	7	ap_none	mem_7_V	pointer
mem_8_V	in	7	ap_none	mem_8_V	pointer
mem_9_V	in	7	ap_none	mem_9_V	pointer
mem_10_V	in	7	ap_none	mem_10_V	pointer
mem_11_V	in	7	ap_none	mem_11_V	pointer
mem_12_V	in	7	ap_none	mem_12_V	pointer
mem_13_V	in	7	ap_none	mem_13_V	pointer
mem_14_V	in	7	ap_none	mem_14_V	pointer
mem_15_V	in	7	ap_none	mem_15_V	pointer
mem_16_V	in	7	ap_none	mem_16_V	pointer
mem_17_V	in	7	ap_none	mem_17_V	pointer
mem_18_V	in	7	ap_none	mem_18_V	pointer
mem_19_V	in	7	ap_none	mem_19_V	pointer
mem_20_V	in	7	ap_none	mem_20_V	pointer
mem_21_V	in	7	ap_none	mem_21_V	pointer
mem_22_V	in	7	ap_none	mem_22_V	pointer
mem_23_V	in	7	ap_none	mem_23_V	pointer
mem_24_V	in	7	ap_none	mem_24_V	pointer
mem_25_V	in	7	ap_none	mem_25_V	pointer
mem_26_V	in	7	ap_none	mem_26_V	pointer
mem_27_V	in	7	ap_none	mem_27_V	pointer
mem_28_V	in	7	ap_none	mem_28_V	pointer
mem_29_V	in	7	ap_none	mem_29_V	pointer
mem_30_V	in	7	ap_none	mem_30_V	pointer
mem_31_V	in	7	ap_none	mem_31_V	pointer
mem_32_V	in	7	ap_none	mem_32_V	pointer

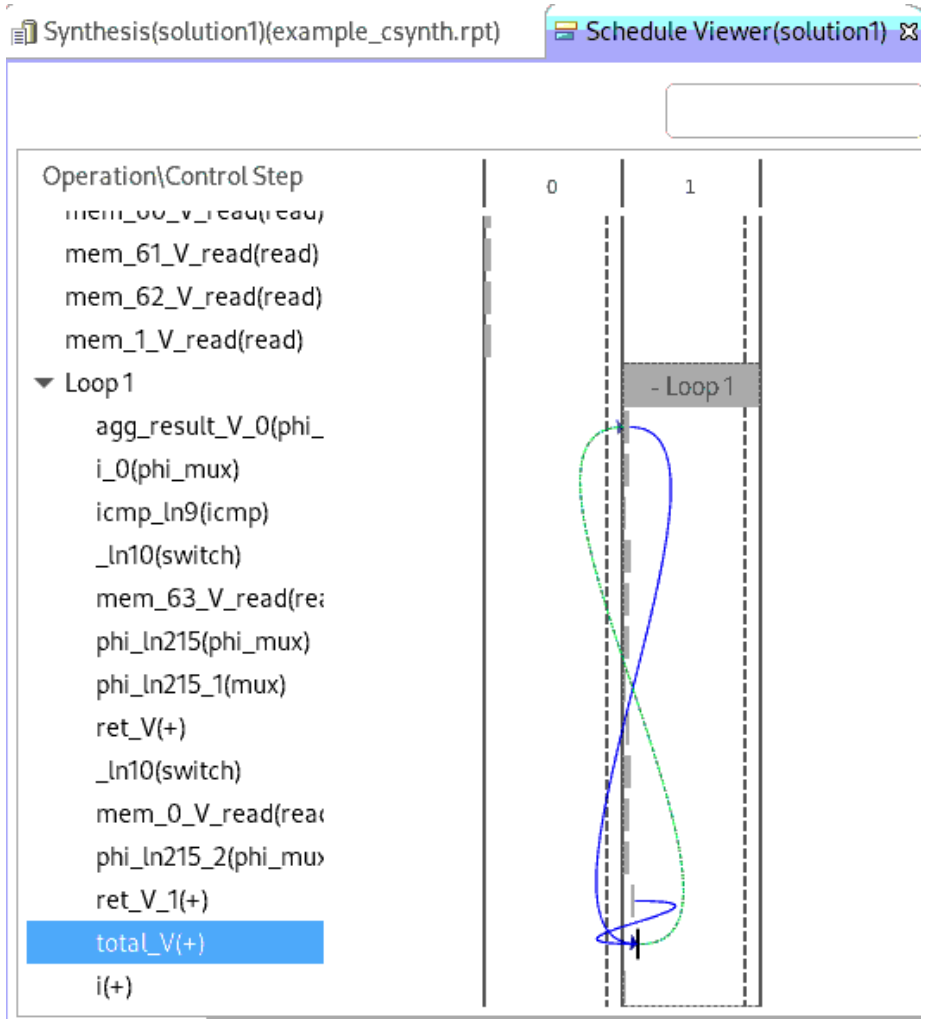
RESHAPE

* Summary:

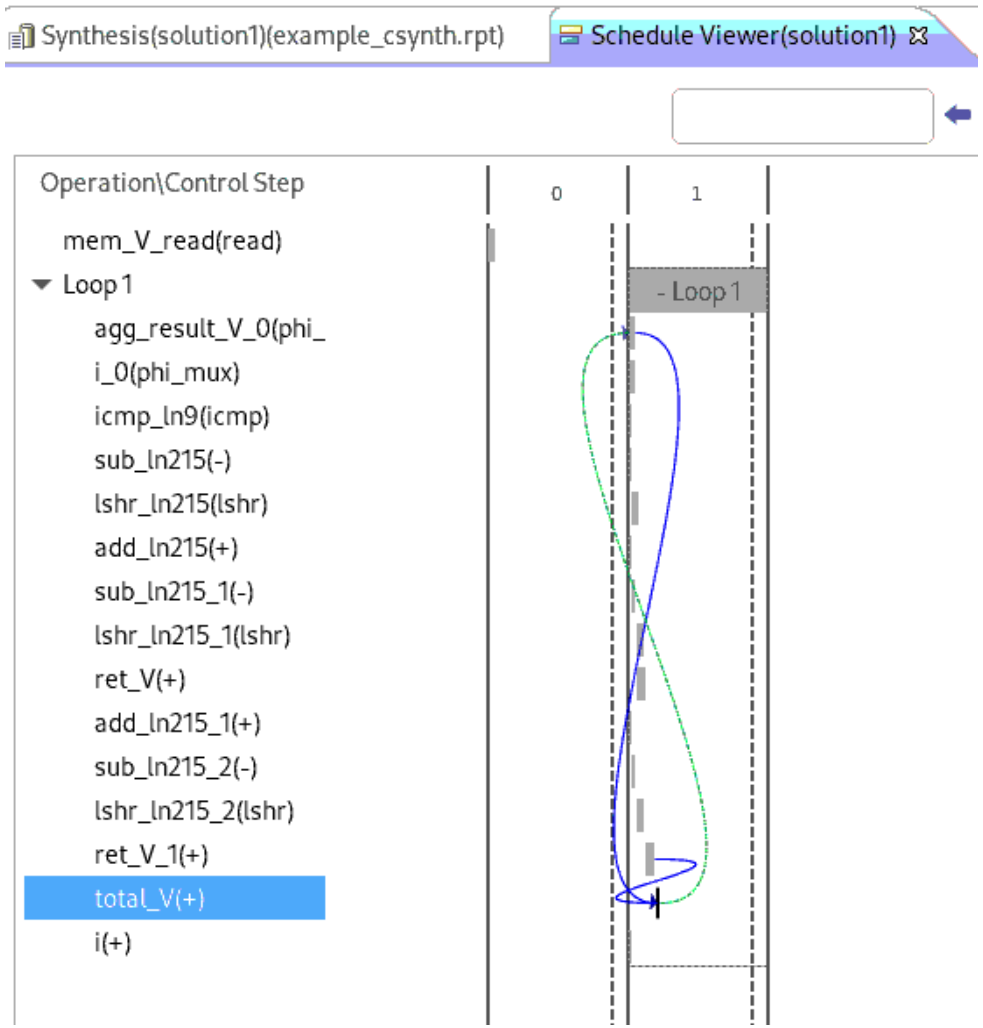
RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	example	return value
ap_rst	in	1	ap_ctrl_hs	example	return value
ap_start	in	1	ap_ctrl_hs	example	return value
ap_done	out	1	ap_ctrl_hs	example	return value
ap_idle	out	1	ap_ctrl_hs	example	return value
ap_ready	out	1	ap_ctrl_hs	example	return value
ap_return	out	10	ap_ctrl_hs	example	return value
mem_V	in	448	ap_none	mem_V	pointer

PARTITION vs RESHAPE

PARTITION



RESHAPE



Pragma HLS array_reshape: Block

```
#include "example.h"

dout_t example(din_t mem[N]) {

#pragma HLS ARRAY_RESHAPE variable=mem block factor=4

    dout_t total = 0;

    for (int i = 2; i < N; ++i)
        total += mem[i] + mem[i - 1] + mem[i - 2];

    return total;
}
```

Timing:

* Summary:

Clock	Target	Estimated	Uncertainty
ap_clk	25.00 ns	2.969 ns	3.12 ns

Latency:

* Summary:

Latency (cycles) min	Latency (cycles) max	Latency (absolute) min	Latency (absolute) max	Interval min	Interval max	Pipeline Type
187	187	4.675 us	4.675 us	187	187	none

* Loop:

Loop Name	Latency (cycles) min	Latency (cycles) max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- Loop 1	186	186	3	-	-	62	no

* Summary:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	428	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	60	-
Register	-	-	46	-	-
Total	0	0	46	488	0
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	0	0	~0	~0	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	0	0	~0	~0	0

Pragma HLS array_reshape: Cyclic

```
#include "example.h"

dout_t example(din_t mem[N]) {

#pragma HLS ARRAY_RESHAPE variable=mem cyclic factor=4

    dout_t total = 0;

    for (int i = 2; i < N; ++i)
        total += mem[i] + mem[i - 1] + mem[i - 2];

    return total;
}
```

```
Timing:
  * Summary:
  +--------+----------+----------+------------+
  |  Clock |  Target  | Estimated| Uncertainty|
  +--------+----------+----------+------------+
  |ap_clk  | 25.00 ns | 3.117 ns |   3.12 ns  |
  +--------+----------+----------+------------+

Latency:
  * Summary:
  +---------+---------+----------+----------+-----+-----+---------+
  |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
  |   min   |   max   |    min   |    max   | min | max |   Type  |
  +---------+---------+----------+----------+-----+-----+---------+
  |      187|      187| 4.675 us | 4.675 us |  187|  187|   none  |
  +---------+---------+----------+----------+-----+-----+---------+
```

```
* Loop:
+----------+---------+---------+----------+----------+----------+-----+----------+
|          |  Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
| Loop Name|   min   |   max   |  Latency | achieved |   target | Count| Pipelined|
+----------+---------+---------+----------+----------+----------+-----+----------+
|- Loop 1  |      186|      186|         3|         -|         -|    62|    no    |
+----------+---------+---------+----------+----------+----------+-----+----------+
```

```
* Summary:
+--------------------+---------+-------+---------+---------+-----+
|        Name        | BRAM_18K| DSP48E|    FF   |   LUT   | URAM|
+--------------------+---------+-------+---------+---------+-----+
|DSP                 |        -|      -|        -|        -|    -|
|Expression          |        -|      -|        0|      416|    -|
|FIFO                |        -|      -|        -|        -|    -|
|Instance            |        -|      -|        -|        -|    -|
|Memory              |        -|      -|        -|        -|    -|
|Multiplexer         |        -|      -|        -|       60|    -|
|Register            |        -|      -|       42|        -|    -|
+--------------------+---------+-------+---------+---------+-----+
|Total               |        0|      0|       42|      476|    0|
+--------------------+---------+-------+---------+---------+-----+
|Available SLR       |     1440|   2280|   788160|   394080|  320|
+--------------------+---------+-------+---------+---------+-----+
|Utilization SLR (%) |        0|      0|    ~0   |    ~0   |    0|
+--------------------+---------+-------+---------+---------+-----+
|Available           |     4320|   6840|  2364480|  1182240|  960|
+--------------------+---------+-------+---------+---------+-----+
|Utilization (%)     |        0|      0|    ~0   |    ~0   |    0|
+--------------------+---------+-------+---------+---------+-----+
```

TAC-HEP 2026

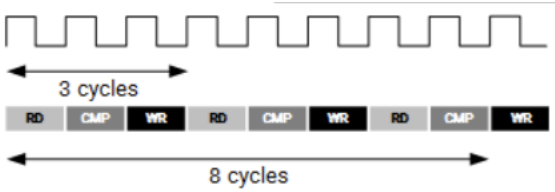
#pragma HLS Pipeline

https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-pipeline

Pragma HLS Pipeline

#pragma HLS pipeline II=<int>

- The PIPELINE pragma reduces the initiation interval (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



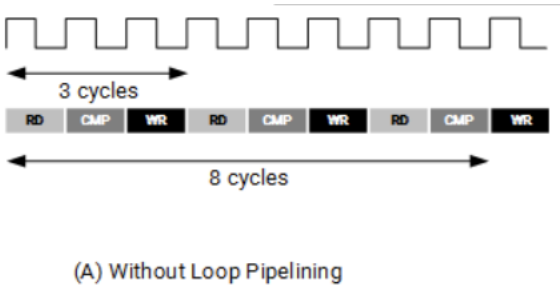
(A) Without Loop Pipelining

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

Pragma HLS Pipeline

#pragma HLS pipeline II=<int>

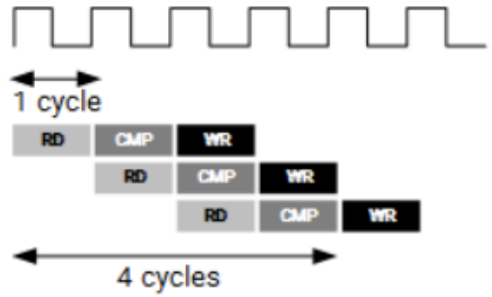
- The PIPELINE pragma reduces the initiation interval (II) for a function or loop by allowing the concurrent execution of operations
- A pipelined function or loop can process new inputs every <N> clock cycles
- If HLS can't create a design with the specified II, it issues a warning and creates a design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

```
#include "example.h"

dout_t example(din_t mem[N]) {

    dout_t total = 0;

    for (int i = 2; i < N; ++i)
        total += mem[i] + mem[i - 1] + mem[i - 2];

    return total;
}
```

Pragma HLS Pipeline: Example

```
#include "example.h"

dout_t example(din_t mem[N]) {

    #pragma HLS pipeline II=2

    dout_t total = 0;

    for (int i = 2; i < N; ++i)
        total += mem[i] + mem[i - 1] + mem[i - 2];

    return total;
}
```

Pragma HLS Pipeline: Example

```
#include "example.h"

dout_t example(din_t mem[N]) {

    #pragma HLS pipeline II=1

    dout_t total = 0;

    for (int i = 2; i < N; ++i)
        total += mem[i] + mem[i - 1] + mem[i - 2];

    return total;
}
```

```
Timing:
  * Summary:
  +--------+----------+----------+------------+
  |  Clock |  Target  | Estimated| Uncertainty|
  +--------+----------+----------+------------+
  |ap_clk  | 25.00 ns | 2.534 ns |   3.12 ns  |
  +--------+----------+----------+------------+

Latency:
  * Summary:
  +---------+---------+----------+----------+-----+-----+---------+
  |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
  |   min   |   max   |    min   |    max   | min | max |   Type  |
  +---------+---------+----------+----------+-----+-----+---------+
  |      126|      126| 3.150 us | 3.150 us |  126|  126|   none  |
  +---------+---------+----------+----------+-----+-----+---------+
```

```
* Loop:
+----------+---------+---------+----------+----------+----------+------+----------+
|          |  Latency (cycles) | Iteration|  Initiation Interval  | Trip |          |
| Loop Name|   min   |   max   |  Latency | achieved |   target | Count| Pipelined|
+----------+---------+---------+----------+----------+----------+------+----------+
|- Loop 1  |      124|      124|         3|         2|         1|    62|    yes   |
+----------+---------+---------+----------+----------+----------+------+----------+
```

```
* Summary:
+--------------------+---------+-------+---------+---------+-----+
|        Name        | BRAM_18K| DSP48E|    FF   |   LUT   | URAM|
+--------------------+---------+-------+---------+---------+-----+
|DSP                 |        -|      -|        -|        -|    -|
|Expression          |        -|      -|        0|      106|    -|
|FIFO                |        -|      -|        -|        -|    -|
|Instance            |        -|      -|        -|        -|    -|
|Memory              |        -|      -|        -|        -|    -|
|Multiplexer         |        -|      -|        -|       78|    -|
|Register            |        -|      -|       46|        -|    -|
+--------------------+---------+-------+---------+---------+-----+
|Total               |        0|      0|       46|      184|    0|
+--------------------+---------+-------+---------+---------+-----+
|Available SLR       |     1440|   2280|   788160|   394080|  320|
+--------------------+---------+-------+---------+---------+-----+
|Utilization SLR (%) |        0|      0|    ~0   |    ~0   |    0|
+--------------------+---------+-------+---------+---------+-----+
|Available           |     4320|   6840|  2364480|  1182240|  960|
+--------------------+---------+-------+---------+---------+-----+
|Utilization (%)     |        0|      0|    ~0   |    ~0   |    0|
+--------------------+---------+-------+---------+---------+-----+
```

Pragma HLS Pipeline: Example

```
#include "example.h"

dout_t example(din_t mem[N]) {

    #pragma HLS pipeline II=2

    dout_t total = 0;

    for (int i = 2; i < N; ++i)
        total += mem[i] + mem[i - 1] + mem[i - 2];

    return total;
}
```

Timing:

* Summary:

Clock	Target	Estimated	Uncertainty
ap_clk	25.00 ns	2.534 ns	3.12 ns

Latency:

* Summary:

Latency (cycles) min	Latency (cycles) max	Latency (absolute) min	Latency (absolute) max	Interval min	Interval max	Pipeline Type
126	126	3.150 us	3.150 us	126	126	none

* Loop:

Loop Name	Latency (cycles) min	Latency (cycles) max	Iteration Latency	Initiation Interval achieved	Initiation Interval target	Trip Count	Pipelined
- Loop 1	124	124	3	2	2	62	yes

* Summary:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	106	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	78	-
Register	-	-	46	-	-
Total	0	0	46	184	0
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	0	0	~0	~0	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	0	0	~0	~0	0

Pragma HLS Pipeline: Example

```
#include "example.h"

dout_t example(din_t mem[N]) {

   #pragma HLS pipeline II=3

   dout_t total = 0;

   for (int i = 2; i < N; ++i)
      total += mem[i] + mem[i - 1] + mem[i - 2];

   return total;
}
```

+ Timing:

* Summary:

Clock	Target	Estimated	Uncertainty
ap_clk	25.00 ns	2.534 ns	3.12 ns

+ Latency:

* Summary:

Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
188	188	4.700 us	4.700 us	188	188	none

* Loop:

Loop Name	Latency (cycles)		Iteration	Initiation Interval		Trip	Pipelined
	min	max	Latency	achieved	target	Count	
- Loop 1	186	186	3	3	3	62	yes

* Summary:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	104	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	66	-
Register	-	-	37	-	-
Total	0	0	37	170	0
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	0	0	~0	~0	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	0	0	~0	~0	0

A Comparison

	Default	Restructured	Pragma HLS Pipeline		
			II = 1	II = 2	II = 3
Estimated Clock (ns)	3.390	2.534	2.534	2.534	2.534
Latency (cycle)	187	126	126	126	188
Latency (μs)	4.675	3.150	3.150	3.150	4.7
Resources (FF)	35	50	46	46	37
Resources (LUT)	164	152	184	184	170
Resource (Others)	0	0	0	0	0

TAC-HEP 2026

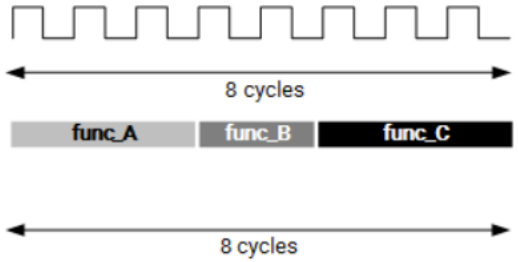
#pragma HLS Dataflow

https://docs.amd.com/r/en-US/ug1399-vitis-hls/pragma-HLS-dataflow

Pragma HLS Dataflow

#pragma HLS dataflow

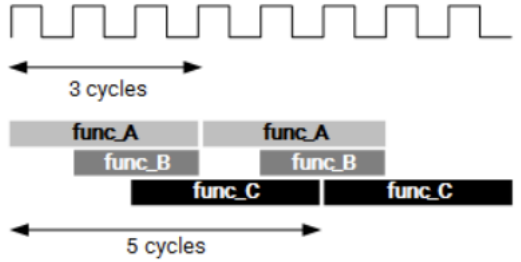
- **Enables task-level pipelining:** allow functions and loops to overlap in their operation
  - Increases the concurrency of the RTL implementation & thus the overall throughput of the design
- In the absence of any directives that limit resources (like pragma HLS allocation), HLS seeks to minimize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



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

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

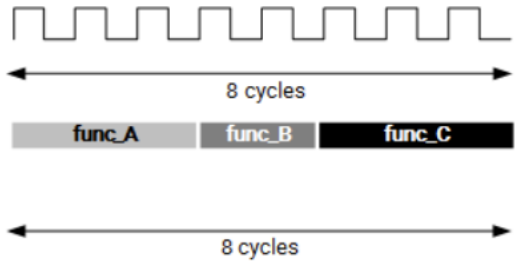
Example:

- 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

Task-level pipeline

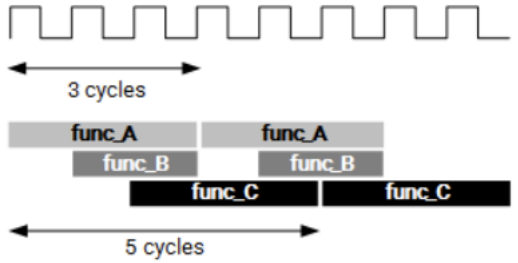
#pragma HLS dataflow



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;
}
```



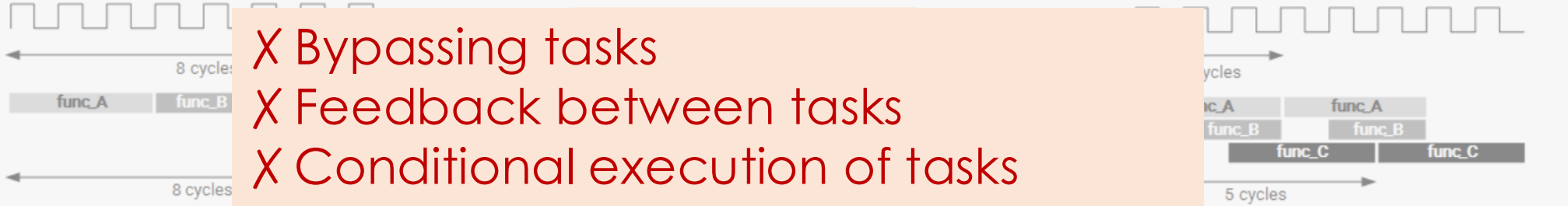
With DATAFLOW pipelining

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

Pragma HLS Dataflow - Example

Task-level pipeline

#pragma HLS dataflow



Without DATAFLO ATAFLOW pipelining

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

For t next

- ✓ 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

Pragma HLS Dataflow - Example

#pragma HLS dataflow

```
#include "example.h"

void example (
  unsigned int in[N],
  short a,
  short b,
  unsigned int c,
  unsigned int out[N]
  ) {

   unsigned int x, y;
   unsigned int tmp1, tmp2, tmp3;

for_Loop: for (unsigned int i=0 ; i < N; i++) {

        x = in[i];
        tmp1 = func(1, 2);
        tmp2 = func(2, 3);
        tmp3 = func(1, 4);

        y = a*x + b + squared(c) + tmp1 + tmp2 + tmp3;

        out[i] = y;
      }
}

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

unsigned int func(short a, short b){

  unsigned int res;
  res= a*a;
  res= res*b*a;
  res= res + 3;

  return res;
}
```

```
void example (
  unsigned int in[N],
  short a,
  short b,
  unsigned int c,
  unsigned int out[N]
  ) {

   unsigned int x, y;
   unsigned int tmp1, tmp2, tmp3;

#pragma HLS dataflow

for_Loop: for (unsigned int i=0 ; i < N; i++) {
#pragma HLS Pipeline

        x = in[i];
        tmp1 = func(1, 2);
        tmp2 = func(2, 3);
        tmp3 = func(1, 4);

        y = a*x + b + squared(c) + tmp1 + tmp2 + tmp3;

        out[i] = y;
      }
}

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

unsigned int func(short a, short b){

  unsigned int res;
  res= a*a;
  res= res*b*a;
  res= res + 3;

  return res;
}
```

Pragma HLS Dataflow

#pragma HLS dataflow

Without DATAFLOW pipelining

```
Timing:
  * Summary:
  +--------+----------+----------+------------+
  |  Clock |  Target  | Estimated| Uncertainty|
  +--------+----------+----------+------------+
  |ap_clk  | 25.00 ns | 7.401 ns |   3.12 ns  |
  +--------+----------+----------+------------+

Latency:
  * Summary:
  +---------+---------+----------+----------+-----+-----+---------+
  |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
  |   min   |   max   |    min   |    max   | min | max |   Type  |
  +---------+---------+----------+----------+-----+-----+---------+
  |      121|      121| 3.025 us | 3.025 us |  121|  121|   none  |
  +---------+---------+----------+----------+-----+-----+---------+
```

With DATAFLOW pipelining

```
Timing:
  * Summary:
  +--------+----------+----------+------------+
  |  Clock |  Target  | Estimated| Uncertainty|
  +--------+----------+----------+------------+
  |ap_clk  | 25.00 ns | 7.401 ns |   3.12 ns  |
  +--------+----------+----------+------------+

Latency:
  * Summary:
  +---------+---------+----------+----------+-----+-----+----------+
  |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline |
  |   min   |   max   |    min   |    max   | min | max |   Type   |
  +---------+---------+----------+----------+-----+-----+----------+
  |       62|       62| 1.550 us | 1.550 us |   63|   63| dataflow |
  +---------+---------+----------+----------+-----+-----+----------+
```

Pragma HLS Dataflow

#pragma HLS dataflow

Without DATAFLOW pipelining

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+---------------------+---------+-------+---------+---------+-----+
|         Name        | BRAM_18K| DSP48E|    FF   |   LUT   | URAM|
+---------------------+---------+-------+---------+---------+-----+
|DSP                  |        -|      -|        -|        -|    -|
|Expression           |        -|      5|        0|      154|    -|
|FIFO                 |        -|      -|        -|        -|    -|
|Instance             |        -|      -|        -|        -|    -|
|Memory               |        -|      -|        -|        -|    -|
|Multiplexer          |        -|      -|        -|       30|    -|
|Register             |        -|      -|      117|        -|    -|
+---------------------+---------+-------+---------+---------+-----+
|Total                |        0|      5|      117|      184|    0|
+---------------------+---------+-------+---------+---------+-----+
|Available SLR        |     1440|   2280|   788160|   394080|  320|
+---------------------+---------+-------+---------+---------+-----+
|Utilization SLR (%)  |        0|   ~0  |    ~0   |    ~0   |    0|
+---------------------+---------+-------+---------+---------+-----+
|Available            |     4320|   6840|  2364480|  1182240|  960|
+---------------------+---------+-------+---------+---------+-----+
|Utilization (%)      |        0|   ~0  |    ~0   |    ~0   |    0|
+---------------------+---------+-------+---------+---------+-----+
```

With DATAFLOW pipelining

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+---------------------+---------+-------+---------+---------+-----+
|         Name        | BRAM_18K| DSP48E|    FF   |   LUT   | URAM|
+---------------------+---------+-------+---------+---------+-----+
|DSP                  |        -|      -|        -|        -|    -|
|Expression           |        -|      -|        -|        -|    -|
|FIFO                 |        -|      -|        -|        -|    -|
|Instance             |        -|      5|      115|      214|    -|
|Memory               |        -|      -|        -|        -|    -|
|Multiplexer          |        -|      -|        -|        -|    -|
|Register             |        -|      -|        -|        -|    -|
+---------------------+---------+-------+---------+---------+-----+
|Total                |        0|      5|      115|      214|    0|
+---------------------+---------+-------+---------+---------+-----+
|Available SLR        |     1440|   2280|   788160|   394080|  320|
+---------------------+---------+-------+---------+---------+-----+
|Utilization SLR (%)  |        0|   ~0  |    ~0   |    ~0   |    0|
+---------------------+---------+-------+---------+---------+-----+
|Available            |     4320|   6840|  2364480|  1182240|  960|
+---------------------+---------+-------+---------+---------+-----+
|Utilization (%)      |        0|   ~0  |    ~0   |    ~0   |    0|
+---------------------+---------+-------+---------+---------+-----+
```

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
...
}
```

Example

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

```
#include "example.h"

void example (
  unsigned int in[N],
  short a,
  short b,
  unsigned int c,
  unsigned int out[N]
  ) {

   unsigned int x, y;
   unsigned int tmp1, tmp2, tmp3;

for_Loop: for (unsigned int i=0 ; i < N; i++) {
#pragma HLS allocation instances=func limit=1 function
        x = in[i];
        tmp1 = func(1, 2);
        tmp2 = func(2, 3);
        tmp3 = func(1, 4);

        y = a*x + b + squared(c) + tmp1 + tmp2 + tmp3;

        out[i] = y;
      }
}

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

Pragma HLS allocation

```
Timing:
  * Summary:
  +--------+----------+----------+------------+
  |  Clock |  Target  | Estimated| Uncertainty|
  +--------+----------+----------+------------+
  |ap_clk  | 25.00 ns | 7.401 ns |   3.12 ns  |
  +--------+----------+----------+------------+

Latency:
  * Summary:
  +---------+---------+----------+----------+-----+-----+---------+
  |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline|
  |   min   |   max   |    min   |    max   | min | max |   Type  |
  +---------+---------+----------+----------+-----+-----+---------+
  |      121|      121| 3.025 us | 3.025 us |  121|  121|   none  |
  +---------+---------+----------+----------+-----+-----+---------+
```

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+---------------------+---------+-------+---------+---------+-----+
|         Name        | BRAM_18K| DSP48E|    FF   |   LUT   | URAM|
+---------------------+---------+-------+---------+---------+-----+
|DSP                  |        -|      -|        -|        -|    -|
|Expression           |        -|      5|        0|      169|    -|
|FIFO                 |        -|      -|        -|        -|    -|
|Instance             |        -|      -|        -|        -|    -|
|Memory               |        -|      -|        -|        -|    -|
|Multiplexer          |        -|      -|        -|       30|    -|
|Register             |        -|      -|       85|        -|    -|
+---------------------+---------+-------+---------+---------+-----+
|Total                |        0|      5|       85|      199|    0|
+---------------------+---------+-------+---------+---------+-----+
|Available SLR        |     1440|   2280|   788160|   394080|  320|
+---------------------+---------+-------+---------+---------+-----+
|Utilization SLR (%)  |        0|   ~0  |    ~0   |    ~0   |    0|
+---------------------+---------+-------+---------+---------+-----+
|Available            |     4320|   6840|  2364480|  1182240|  960|
+---------------------+---------+-------+---------+---------+-----+
|Utilization (%)      |        0|   ~0  |    ~0   |    ~0   |    0|
+---------------------+---------+-------+---------+---------+-----+
```

Homework #4

- https://github.com/varuns23/TAC-HEP-FPGA/tree/main/2026/homeworks

Q5. Design and optimize an HLS kernel that performs element-wise arithmetic operations on two input arrays and produces multiple outputs. Given two input arrays A[N] and B[N] (N ≥ 64), compute:

- C[i] = A[i] + B[i]
- D[i] = A[i] * B[i]
- E[i] = (A[i] + B[i]) * (A[i] - B[i])
- a) Implement a simple version using a single loop. Use standard arrays (no pragmas). Measure latency and resource utilization after synthesis.
- b) Apply HLS Interface Pragmas: How does interface selection impact throughput and memory access?
- c) Apply loop pipelining (II=1): Report latency before and after pipelining
- d) Compare: No partitioning, Complete partitioning, Cyclic/block partitioning, reshape

TAC-HEP 2026

Questions?

TAC-HEP 2026

Extra Slides

TAC-HEP 2026

HLS Setup on cmstrigger02

https://github.com/varuns23/TAC-HEP-FPGA/blob/main/hls-setup/readme.md

Setup

Setup Port Forwarding for RDP

- From your local machine, run:
  - `ssh -L 3389:localhost:3389 -J <username>@login.hep.wisc.edu <username>@cmstrigger02.hep.wisc.edu`
- Keep this terminal open - it maintains the RDP tunnel

Connect Using Remote Desktop (RDP Client)

- Use Microsoft's RDP client (called `Windows App`) available for macOS and Windows.
- **Setup:** Download & install the `Windows App`, Open the app and click the **+** icon, then select `Add PC`
- **Configure:** Enter the IP address: `localhost:3389` or `127.0.0.1:3389`
- **Connect:** Double-click the PC icon, enter your UW computing and , and click **Connect**
- You should now see the remote desktop of `cmstrigger02`

https://github.com/varuns23/TAC-HEP-FPGA/blob/main/hls-setup/readme.md

Setup workign Directory

After logging in via RDP, open a terminal inside the remote desktop

- `mkdir -p /scratch/`whoami`` (if not there already)
- `cd /scratch/`whoami``
  - **For Vitis HLS:**
    - `Source /opt/Xilinx/Vitis/2020.1/settings64.sh`
    - `cd /scratch/`whoami``
    - `vitis_hls`
  - **For Vivado HLS:**
    - `Source /opt/Xilinx/Vivado/2020.1/settings64.sh`
    - `cd /scratch/`whoami``
    - `vivado_hls`

Homework# 2 (lecture-6)

- Run following C++ example using vitis and share your conclusion.
  - https://github.com/varuns23/TAC-HEP-FPGA/blob/main/2026/lecture06/lex6-ex01.cpp

- Write a progam to sort 16 object and with Vitis

Homework#1 (lecture-5)

```
1   #include <iostream>
2   #define N 10
3
4   void matrix_add(int A[N][N], int B[N][N], int C[N][N]) {
5
6       for (int i = 0; i < N; i++) {
7           for (int j = 0; j < N; j++) {
8               C[i][j] = A[i][j] + B[i][j];
9           }
10      }
11  }
```

1. Run on HLS for the above example of matrix addition and check the resource utilization
2. Compare with a vector addition and matrix multilplication.

Share your resource utilization for all three cases and your observation about the same.

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
- **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

Terminology

- **HLS file:** C/C++ code that will be synthesised and run on FPGA
- **Test bench (TB) file:** C/C++ code that is run to test the HLS code. It calls the HLS functions and can run tests on their output, e.g. C asserts
- **Tcl scripts:** set of tcl instructions executed by the Vivado HLS shell

- **Synthesis:** C/C++ ➞ HDL lang (VHDL/Verilog)
- **Project:** Collection of HLS and test bench (TB) files
  - Has a top-level function name that is the starting point for synthesis
- **Solution:** specific implementation of project
  - Runs on a specific device at a specific clock frequency
- **C simulation:** HLS+TB files are compiled with gcc against HLS headers and lib and plainly run as any other executable
- **C/RTL cosimulation:** synthesized HLS code is run on simulator and results tested on the C/C++ test bench

TAC-HEP 2026

Pragma HLS Interface

Advanced eXtensible Interface (AXI)

AXI is part of ARM AMBA, a family of micro controller buses first introduced in 1996

AXI4 is for memory-mapped interfaces and allows high throughput bursts of up to 256 data transfer cycles with just a single address phase.

There are three types of AXI4 interfaces:

- **AXI4 (m_axi):** For high-performance memory-mapped requirements
  - Specify on array and pointers (and references in C++)

- **AXI4-Lite (s_axilite):** For simple, low-throughput memory-mapped communication (for example, to and from control and status registers)
  - Can be used on any type of argument except streams

- **AXI4-Stream (axis):** For high-speed streaming data
  - Use this protocol on input arguments or output arguments only,

Control Signal: ap_start

- This signal controls the block execution and must be asserted to ***logic 1*** for the design to begin operation.
- It should be held at ***logic 1*** until the associated output handshake *ap_ready* is asserted.
- Keep *ap_start* = 1 until ap_ready becomes **1** (meaning the task is done, and new data can be processed)
- If *ap_start* is asserted low before *ap_ready* is high, the design might not have read all input ports and might stall operation on the next input read

Control Signal: ap_ready

- This output signal indicates when the design is ready for new inputs.
- The *ap_ready* signal is set to **logic 1** when the design is ready to accept new inputs, indicating that all input reads for this transaction have been completed.
- If the design has no pipelined operations, new reads are not performed until the next transaction starts.
- If *ap_start = 0*, the design will **stop** after finishing its current task.

Control Signal: ap_done

- This signal indicates when the design has completed all operations in the current transaction.
- *ap_done* = 1 means the design has **finished processing** all operations for the current task.
- If there is an *ap_return* output, the value is now **valid** and ready to be read.
- Not all functions have a function return argument and hence not all RTL designs have an ap_return port

Control Signal: ap_idle

- This signal indicates if the design is operating or idle (no operation).
- What *ap_idle* = 1 means the design is not doing anything (idle), It is waiting for *ap_start* = 1 to begin working
- This signal is asserted high when the design completes operation and no further operations are performed.