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

GPU & FPGA module training: Part-2

Week-5: Vivado HLS: More pragmas and *Do's & Don'ts*

Lecture-9: April 18th 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, Introduction to Pragmas
  - Different Pragmas and their effects on performance

Today:

- Some more pragmas
- Good practices to write HLS codes
  - Does & Don'ts

TAC-HEP 2023

HLS Pragmas

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	

HLS Loop_flatten

#pragma HLS loop_flatten off

- Allow nested loops to be flattened into a single loop hierarchy with improved latency
- In RTL implementation, takes on clock-cycle to move from outer loop to an inner loop & vice-versa
- Flattening nested loops allow them to be optimized as a single loop
  - Saves clock cycles
  - Allows for greater optimization of the loop body logic
- *LOOP_FLATTEN* pragma is applied to the loop body of inner-most loop in the hierarchy
  - Only perfect & semi-perfect loops can be flattened

Different for-loops

- Perfect loop nest:
  - Only innermost loop body has content
  - No logic between loop statements
  - All loop bounds are constant
- Semi-perfect loop nest:
  - Only innermost loop body has content
  - No logic between loop statements
  - Outermost loop bound can be a variable
- Imperfect loop nest:
  - Loop body is not exclusively inside the inner loop
  - Try restructure or UNROLL

Examples

#pragma HLS loop_flatten

Perfect for loop

```
#include "lec9Ex3.h"

void lec9Ex3(din_t A[N], dout_t B[N]) {
  int i,j;
  dint_t acc;

LOOP_I:for(i=0; i < 20; i++){
LOOP_J: for(j=0; j < 20; j++){
#pragma HLS loop_flatten
          if(j==0) acc = 0;
          acc += A[j] * j;
          if(j==19) {
            if (i%2 == 0)
              B[i] = acc / 20;
            else
              B[i] = 0;
          }
        }
       }
}
```

Ex: lec9Ex3.c

Imperfect for loop

```
#include "lec9Ex4.h"

void lec9Ex4(din_t A[N], dout_t B[N]) {

    int i,j;
    dint_t acc;

    LOOP_I:for(i=0; i < 20; i++){
        acc = 0;
        LOOP_J: for(j=0; j < 20; j++){
#pragma HLS loop_flatten
            acc += A[j] * j;
        }
        if (i%2 == 0)
            B[i] = acc / 20;
        else
            B[i] = 0;
    }

}
```

Ex: lec9Ex4.c

Perfect loop – Performance Estimates

#pragma HLS loop_flatten

Ex: lec9Ex3.c

Without Loop_flatten pragma

Performance Estimates

- Timing
  - Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.902 ns	1.25 ns

- Latency
  - Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1241	1641	12.410 us	16.410 us	1241	1641	none

  - Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- LOOP_I	1240	1640	62 ~ 82	-	-	20	no
+ LOOP_J	60	80	3 ~ 4	-	-	20	no

With Loop_flatten pragma

- Timing
  - Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.902 ns	1.25 ns

- Latency
  - Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
1201	1601	12.010 us	16.010 us	1201	1601	none

  - Detail
    - Instance
    - Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- LOOP_I_LOOP_J	1200	1600	3 ~ 4	-	-	400	no

Perfect loop – Resource Utilization

#pragma HLS loop_flatten

Ex: lec9Ex3.c

Without Loop_flatten pragma

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	2	-	-	-
Expression	-	-	0	152	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	65	-
Register	-	-	76	-	-
Total	0	2	76	217	0
Available	650	600	202800	101400	0
Utilization (%)	0	~0	~0	~0	0

With Loop_flatten pragma

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	2	-	-	-
Expression	-	-	0	180	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	70	-
Register	-	-	94	-	-
Total	0	2	94	250	0
Available	650	600	202800	101400	0
Utilization (%)	0	~0	~0	~0	0

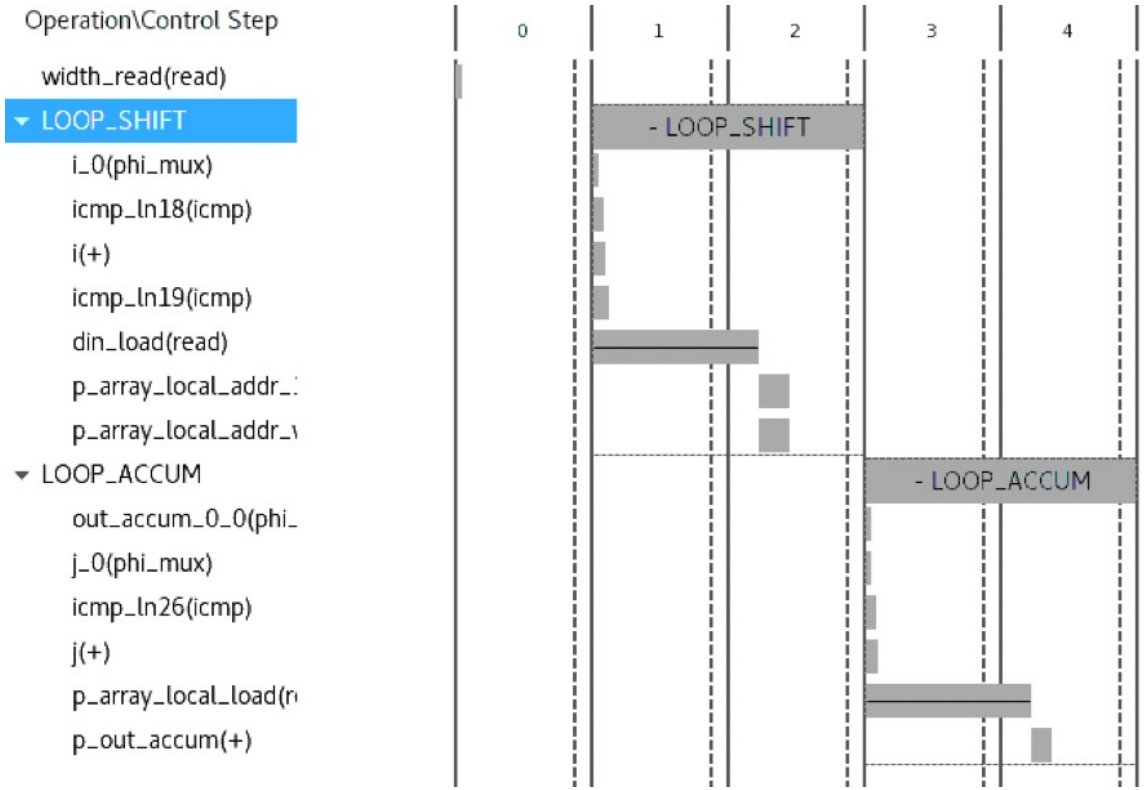
Perfect loop

Perfect loop – Analysis Perspective

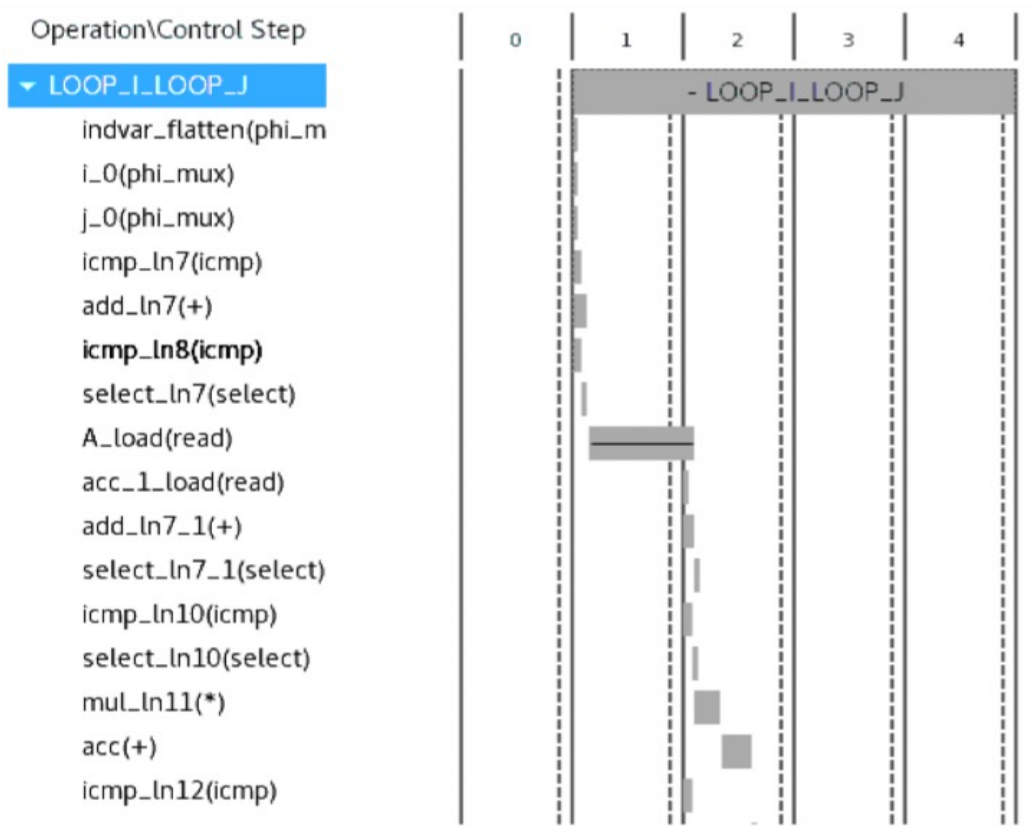
#pragma HLS loop_flatten

Ex: lec9Ex3.c

Without Loop_flatten pragma



With Loop_flatten pragma



Imperfect loop

#pragma HLS loop_flatten

Ex: lec9Ex4.c

Without Loop_flatten pragma

- Timing
  - Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.902 ns	1.25 ns

- Latency
  - Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
861	861	8.610 us	8.610 us	861	861	none

  - Detail
    - Instance
    - Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- LOOP_I	860	860	43	-	-	20	no
+ LOOP_J	40	40	2	-	-	20	no

With Loop_flatten pragma

- Timing
  - Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.902 ns	1.25 ns

- Latency
  - Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
861	861	8.610 us	8.610 us	861	861	none

  - Detail
    - Instance
    - Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- LOOP_I	860	860	43	-	-	20	no
+ LOOP_J	40	40	2	-	-	20	no

Imperfect loop

#pragma HLS loop_flatten

Ex: lec9Ex4.c

Without Loop_flatten pragma

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.902 ns	1.25 ns

```
WARNING: [XFORM 203-542] Cannot flatten a loop nest 'LOOP_I' (lec9Ex4.c:8:34) in function 'lec9Ex4' :

the outer loop is not a perfect loop because there is nontrivial logic before entering the inner loop.
```

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- LOOP_I	860	860	43	-	-	20	no
+ LOOP_J	40	40	2	-	-	20	no

With Loop_flatten pragma

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.902 ns	1.25 ns

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- LOOP_I	860	860	43	-	-	20	no
+ LOOP_J	40	40	2	-	-	20	no

HLS Loop_merge

#pragma HLS loop_merge force

- Merge consecutive loops into a single loop to reduce overall latency, increase sharing, and improve logic optimization

- Merging loops:
  - Reduces the number of clock cycles required in the RTL to transition between the loop-body implementations
  - Allows the loops be implemented in parallel (if possible)

- *force*: optional keyword to force loops to be merged even when HLS tool issues a warning

HLS Loop_merge

#pragma HLS loop_merge force

- **Some rules before thinking to merge:**
  - For variable loop bounds, must have same value (# of iterations)
  - For constant loop bounds, the max. constant value is used as the bound of the merged loop
  - Loops with both variable bounds and constant bounds CANNOT be merged.
  - The code between loops to be merged cannot have side effects
    - Multiple execution of this code should generate the same results (a=b is allowed, a=a+1 is not).
  - Loops cannot be merged when they contain FIFO reads
    - Reads from a FIFO or FIFO interface must always be in sequence

TAC-HEP 2023

HLS Coding Styles

Un-supported C contructs

- HLS supports a wide range of C-language

✗ Some constructs are **NOT Synthesizable**

- Can result in errors further down the design flow

✓ **To be synthesized successfully:**

- C-function must contain entire functionality of the design
- No functionality can be performed by system calls to the Operating System
- C-construct must be of a bounded size
- Implementation must be un-ambiguous

System Calls

- System calls can't be synthesized as they are performed upon Operating System

- HLS ignores common system calls such as:
  - *printf()*, *fprintf(stdout,)*, *getc()*, *time()*, *sleep()* etc..

FORBIDDEN

- Vivado HLS defines macro *__SYNTHESIS__* when synthesis is performed
  - Only use macro in the code to be **synthesized** and not in the test bench

  - Example on next slide: macro *__SYNTHESIS__* is used to ensure the non-synthesizable files writes are ignored during synthesis

System Calls

Ex: lec9Ex1.c

```
#include "lec9Ex1.h"

void lec9Ex1 (
  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;
      }

#ifndef __SYNTHESIS__
//Following code will be ignored for synthesis
          FILE *f1;
          f1=fopen("out_temp","w");
          for(unsigned int j=0 ; j < N; j++){
              fprintf(f1,"%u\n",out[j]);
          }
          fclose(f1);
#endif
}
```

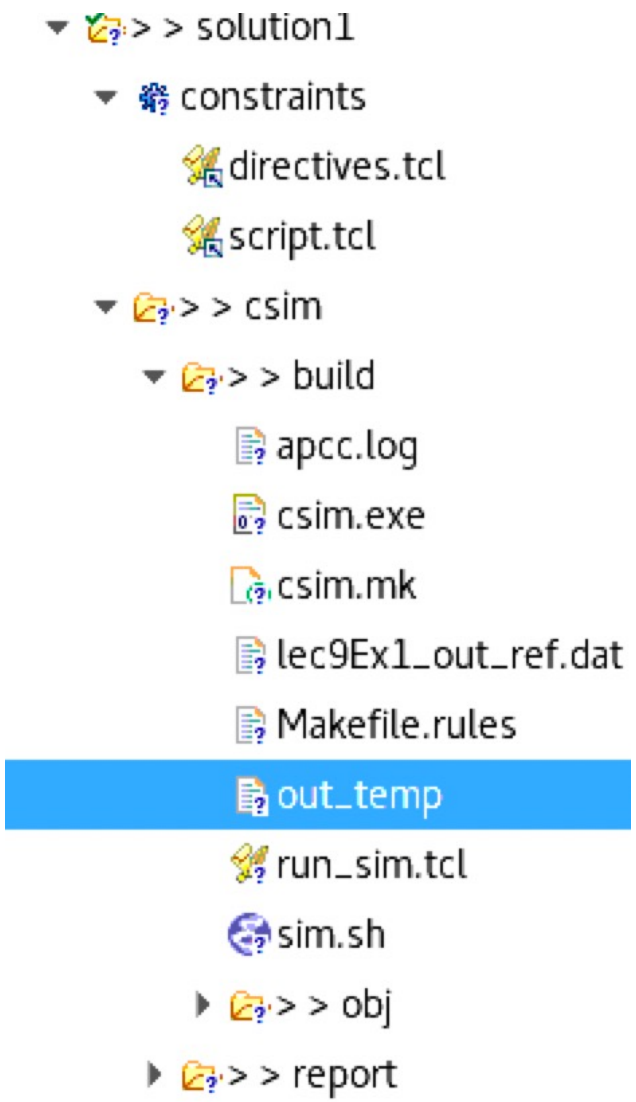
Code to be synthesized

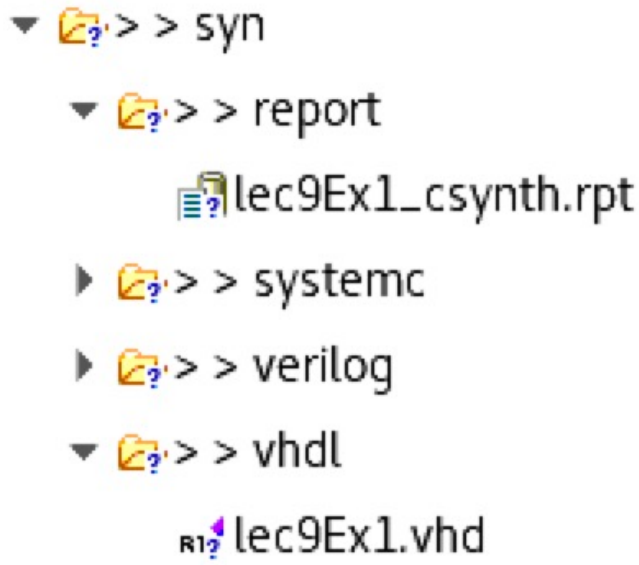
```
#include "lec9Ex1.h"
#include <stdlib.h>


int main () {

  unsigned int input[N];
  unsigned int output[N];

  short a = 2;
  short b = 3;
  unsigned int c = 5;


  for(int irnd=0; irnd<N; irnd++){
    input[irnd] = rand() % 20;
    output[irnd] = 0;
    printf("%i, input: %u", irnd, input[irnd]);
  }


  // Execute the function with latest input
  lec9Ex1(input, a, b, c, output);

  for(int i=0; i<N; i++){
    printf("%i %u %u\n",i,input[i],output[i]);
  }
  return 0;
}
```

Test bench

System Calls

Ex: lec9Ex1.cc





Dynamic Memory Usage

- Memory allocation system calls MUST be removed from the design code before synthesis

- System calls that manages memory allocations, such as:
  - malloc(), alloc(), free()
  - Uses resources from OS memory

FORBIDDEN

- Dynamic memory operations must be transformed into equivalent bounded representations

- Example on next slide: how a design using malloc() can be transformed

Dynamic Memory Usage

Ex: lec9Ex2.c

```
#include "lec9Ex2.h"
#include <stdlib.h>
//#define NO_SYNTH

dout_t lec9Ex2(din_t din[N], dsel_t width) {

#ifdef NO_SYNTH
        long long *out_accum = malloc (sizeof(long long));
        int* array_local = malloc (64 * sizeof(int));
#else
        long long _out_accum;
        long long *out_accum = &_out_accum;
        int _array_local[64];
        int* array_local = &_array_local[0];
#endif
        int i,j;

  LOOP_SHIFT:for (i=0;i<N-1; i++) {
    if (i<width)
                        *(array_local+i)=din[i];
                else
                        *(array_local+i)=din[i]>>2;
  }

        *out_accum=0;
  LOOP_ACCUM:for (j=0;j<N-1; j++) {
      *out_accum += *(array_local+j);
  }

  return *out_accum;
}
```

Allocates a region in memory to store 64 values of 32 bits each

Although this coding example clearly states a constant memory allocation

HLS code does not analyze the contents of the malloc statement

Dynamic Memory Usage

Ex: lec9Ex2.c

```
#include "lec9Ex2.h"
#include <stdlib.h>
//#define NO_SYNTH

dout_t lec9Ex2(din_t din[N], dsel_t width) {

#ifdef NO_SYNTH
        long long *out_accum = malloc (sizeof(long long));
        int* array_local = malloc (64 * sizeof(int));
#else
        long long _out_accum;
        long long *out_accum = &_out_accum;
        int _array_local[64];
        int* array_local = &_array_local[0];
#endif
        int i,j;

  LOOP_SHIFT:for (i=0;i<N-1; i++) {
    if (i<width)
                        *(array_local+i)=din[i];
                else
                        *(array_local+i)=din[i]>>2;
  }

        *out_accum=0;
  LOOP_ACCUM:for (j=0;j<N-1; j++) {
      *out_accum += *(array_local+j);
  }

  return *out_accum;
}
```

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	4.532 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
126	126	1.260 us	1.260 us	126	126	none

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	114	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	1	-	0	0	0
Multiplexer	-	-	-	82	-
Register	-	-	68	-	-
Total	1	0	68	196	0
Available	650	600	202800	101400	0
Utilization (%)	~0	0	~0	~0	0

Dynamic Memory Usage

Snippet from logs

```
INFO: [RTGEN 206-500] Setting interface mode on port 'lec9Ex2/din' to 'ap_memory'.
INFO: [RTGEN 206-500] Setting interface mode on port 'lec9Ex2/width' to 'ap_none'.
INFO: [RTGEN 206-500] Setting interface mode on function 'lec9Ex2' to 'ap_ctrl_hs'.
INFO: [SYN 201-210] Renamed object name 'lec9Ex2_p_array_local' to 'lec9Ex2_p_array_lbkb' due to the length limit 20
...
INFO: [RTMG 210-278] Implementing memory 'lec9Ex2_p_array_lbkb_ram (RAM)' using block RAMs.
INFO: [HLS 200-111] Finished generating all RTL models Time (s): cpu = 00:00:31 ; elapsed = 00:00:33 . Memory (MB): peak = 1775.266 ;
INFO: [VHDL 208-304] Generating VHDL RTL for lec9Ex2.
INFO: [VLOG 209-307] Generating Verilog RTL for lec9Ex2.
```

- The design does not use *__SYNTHESIS__* macro
- User-defined macro *NO_SYNTH* is used to select between the synthesizable and non- synthesizable versions
- Fixed sized resources can be created & the existing pointer can simply be made to point to the fixed sized resources
  - Prevent manual recoding of the existing design

Dynamic Memory Usage

Ex: lec9Ex2.c

```
#include "lec9Ex2.h"
#include <stdlib.h>
//#define NO_SYNTH

dout_t lec9Ex2(din_t din[N], dsel_t width) {

#ifdef NO_SYNTH
        long long *out_accum = malloc (sizeof(long long));
        int* array_local = malloc (64 * sizeof(int));
#else
        long long _out_accum;
        long long *out_accum = & out_accum;
        int
        int
#endif
        int
  LOOP_SHIFT:for (i=0;i<N-1; i++) {
    if (i<width)
                        *(array_local+i)=din[i];
                else
                        *(array_local+i)=din[i]>>2;
  }

        *out_accum=0;
  LOOP_ACCUM:for (j=0;j<N-1; j++) {
      *out_accum += *(array_local+j);
  }

  return *out_accum;
}
```

Assignment exercise1: See what happens if you don't use NO_SYNTH macro and remove alternate code

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	4.532 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type

DSP	-	-	-	-	-
Expression	-	-	0	114	-
FIFO	-	-	-	-	-
Instance	-	-	-	-	-
Memory	1	-	0	0	0
Multiplexer	-	-	-	82	-
Register	-	-	68	-	-
Total	1	0	68	196	0
Available	650	600	202800	101400	0
Utilization (%)	~0	0	~0	~0	0

Dynamic Memory Usage

- As above example had coding changes which MAY impact the functionality of the design, Xilinx **DOES NOT** recommend using the __SYNTHESIS__ macro

- Xilinx recommends that you perform the following steps:

1. Add used-defined macro __NO_SYNTH__ to code & modify
2. Enable macro __NO_SYNTH__ , execute the C-simulation, and save the results
3. Disable the macro __NO_SYNTH__ and execute the C simulation to verify that the results are identical
4. Perform synthesis with the user-defined macro disabled

This will ensure the design functionality is retained even after synthesis

Dynamic Memory Usage

- Give restrictions on dynamic memory usage in C
- HLS does not support (for synthesis) C++ objects that are dynamically created or destroyed such as
  - Polymorphism, dynamic virtual function calls

```
Class A {
public:
   virtual void bar() {a;}
};
void fun(A* a) {
   a->bar(); }
   A* a = 0;
if (base)
  a = new A();
  else
  a = new B();
  foo(a);
```

Cannot be synthesized because it creates a new function at run time

Pointer Limitations

- Pointers in a C/C++ program are function parameters, array handling, pointer to pointer, and type casting

- HLS compiler supports pointer usage that can be completely analyzed at compile time

```
int *A = malloc(10*sizeof(int)); //Not allowed
```

Pointer to reference a dynamically allocated region in memory

```
int A[10];
int *pA;

pA = A;
```

Valid coding style

Array access with a pointer

- All uses of pointer pA can be analyzed & mapped back to array A.
- As array A is created by automatic memory allocation, HLS can fully determine the properties of A

Pointer Limitations

- Another supported model for memories and pointers is in accessing external memory
- HLS: Any pointer access on function parameters implies either a variable or an external memory.
- External memory:any memory outside of the scope of the compiler-generated RTL
  - Memory might be located in another function in the FPGA or in part of an off-chip memory, such as DDR

Pointer to extrenal memory

```
void foo(int *data_in,...)
{
    int item1, item2, item3;

  item1 = *data_in;
  item2 = *(data_in + 1);
  item3 = *(data_in + 2);
  ...
}
```

Example: Function *foo* is a top-level module for HLS with *data_in* as a parameter

- Based on the multiple pointer access on *data_in*, HLS infers that this function parameter is an external memory module, which must be accessed through a bus protocol at the hardware level

Standard Template Libraries

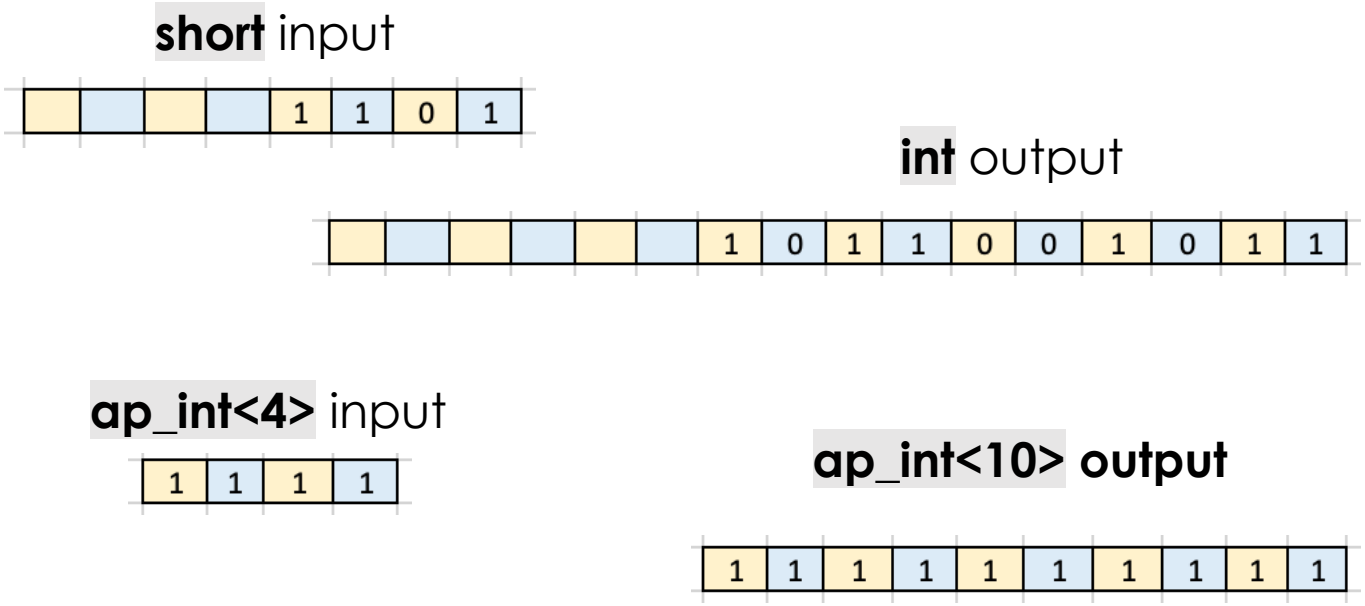
- Many of the C++ Standard Template Libraries (STLs) contain function recursion and use dynamic memory allocation
- For this reason, the STLs cannot be synthesized
- *Solution:* Create a local function with identical functionality that does not exhibit these characteristics of recursion, dynamic memory allocation or the dynamic creation and destruction of objects

**Standard data types, such as std::complex, are supported for synthesis

Arbitrary precision

Creating hardware, it is useful to use more accurate bit-widths

For ex: a case in which the input to a filter is 4-bit and the yielded results requires a maximum of 10-bits



C/C++ data types	Bit-width
(unsigned) char	4
(unsigned) short	8
(unsigned) int	16
(unsigned) long	32
(unsigned) long long	64
float	32
double	64
IntN_t	N=8/16/32/64

Arbitrary precision

Using standard C data types for hardware design results in unnecessary hardware costs.

Operations can use more LUTs and registers than needed for the required accuracy, and delays might even exceed the clock cycle, requiring more cycles to compute the result

C/C++ data types	Bit-width
(unsigned) char	4
(unsigned) short	8
(unsigned) int	16
(unsigned) long	32
(unsigned) long long	64
float	32
double	64
IntN_t	N=8/16/32/64

Assignment Week-5

1. Do exercise mention on slide-24
2. A matrix multiplication using two for loops and compare results for pragma loop_flatten & unroll
3. Write a simple program doing arithmetic operations(+, -, *, /, %) between two variable use of arbitrary precision to compare results between stand c/c++ data types and using ap_(u)int<N>

TAC-HEP 2023

Questions?

TAC-HEP 2023

Acknowledgement

Lectures are compiled using content from Xilinx's public pages/examples or different user guides

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

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

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

2. Compare the analysis perspective (Performance) for different case shared today
3. For Array_partitioning, instead of using complete, use block and cyclic with different factors