Improving Performance

Vivado HLS 2013.3 Version

Objectives

> **After completing this module, you will be able to:**

– Add directives to your design

– List number of ways to improve performance

– State directives which are useful to improve latency

– Describe how loops may be handled to improve latency

– Recognize the dataflow technique that improves throughput of the design

– Describe the pipelining technique that improves throughput of the design

– Identify some of the bottlenecks that impact design performance

Improving Performance

- **Vivado HLS has a number of way to improve performance**
  - Automatic (and default) optimizations
  - Latency directives
  - Pipelining to allow concurrent operations
- **Vivado HLS support techniques to remove performance bottlenecks**
  - Manipulating loops
  - Partitioning and reshaping arrays
- **Optimizations are performed using directives**
 - Let's look first at how to apply and use directives in Vivado HLS

XILINX ➤ ALL PROGRAMMABLE.

Outline

- ***Adding Directives***
- **Improving Latency**
  - Manipulating Loops
- **Improving Throughput**
- **Performance Bottleneck**
- **Summary**

XILINX ➤ ALL PROGRAMMABLE.

Applying Directives

▶ **If the source code is open in the GUI Information pane**

– The Directive tab in the Auxiliary pane shows all the locations and objects upon which directives can be applied (in the opened C file, not the whole design)

• Functions, Loops, Regions, Arrays, Top-level arguments

– Select the object in the Directive Tab

• "dct" function is selected

– Right-click to open the editor dialog box

– Select a desired directive from the drop-down menu

• "DATAFLOW" is selected

– Specify the Destination

• Source File

• Directive File

**XILINX** ➤ ALL PROGRAMMABLE.
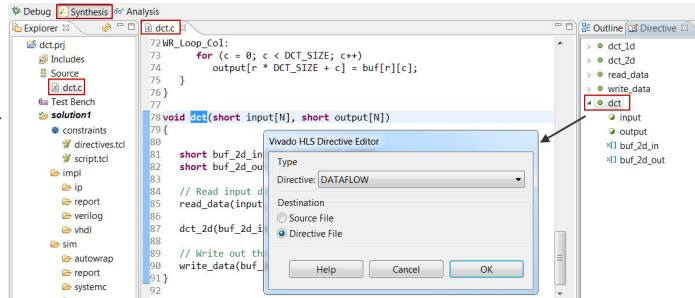
Optimization Directives: Tcl or Pragma

▶ **Directives can be placed in the directives file**

– The Tcl command is written into directives.tcl

– There is a directives.tcl file in each solution

• Each solution can have different directives

> Once applied the directive will be shown in the Directives tab (right-click to modify or delete)

● dct
% HLS DATAFLOW
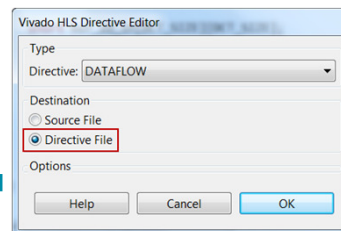
▶ **Directives can be place into the C source**

– Pragmas are added (and will remain) in the C source file

– Pragmas (#pragma) will be used by every solution which uses the code

● dct
HLS DATAFLOW

```
78 void dct(short input[N], short output[N])
79 {
80 #pragma HLS DATAFLOW
81
```

XILINX ➤ ALL PROGRAMMABLE.
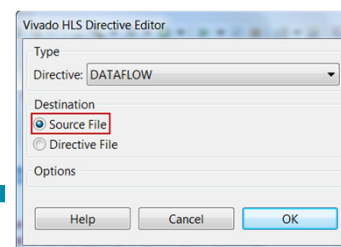
Solution Configurations

- **Configurations can be set on a solution**
  - Set the default behavior for that solution
    - Open configurations settings from the menu (Solutions > Solution Settings…)

"Add" or "Remove" configuration settings

Select "General"
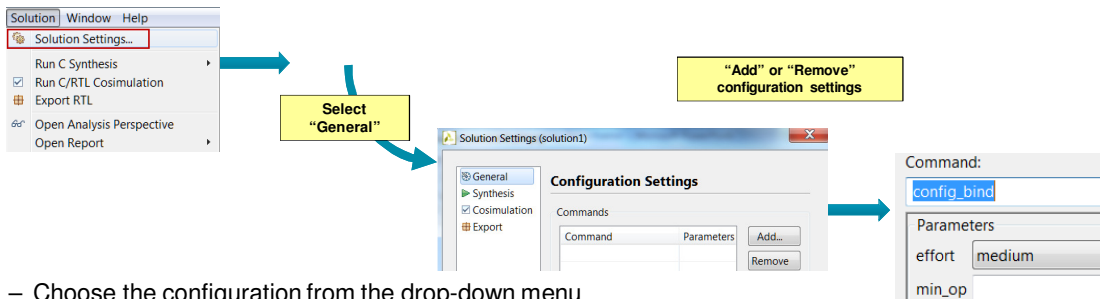
  - Choose the configuration from the drop-down menu
    - Array Partitioning, Dataflow Memory types, Default IO ports, RTL Settings, Operator binding, Schedule efforts

XILINX ➤ ALL PROGRAMMABLE.

Example: Configuring the RTL Output

- **Specify the FSM encoding style**
  - By default the FSM is auto
- **Add a header string to all RTL output files**
  - Example: Copyright Acme Inc.
- **Add a user specified prefix to all RTL output filenames**
  - The RTL has the same name as the C functions
  - Allow multiple RTL variants of the same top-level function to be used together without renaming files
- **Reset all registers**
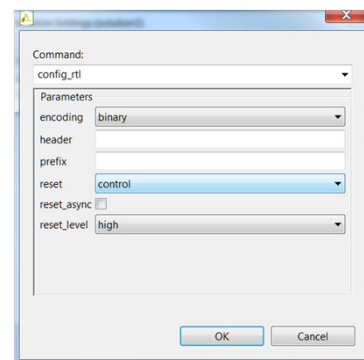 - By default only the FSM registers and variables initialized in the code are reset
  - RAMs are initialized in the RTL and bitstream
- **Synchronous or Asynchronous reset**
  - The default is synchronous reset
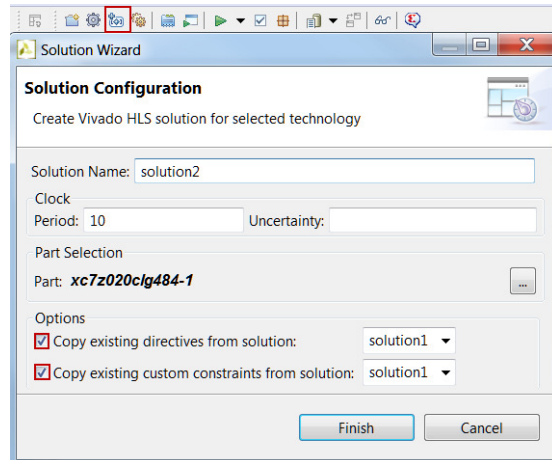- **Active high or low reset**
  - The default is active high

The remainder of the configuration commands will be covered throughout the course

XILINX ➤ ALL PROGRAMMABLE.

Copying Directives into New Solutions

- **Select the New Solution Button**
- **Optionally modify any of the settings**
  - Part, Clock Period, Uncertainty
  - Solution Name
- **Copy existing directives**
  - By default selected
  - Uncheck if do not want to copy
  - No need to copy pragmas, they are in the code
- **Copy any existing custom commands in to the new script.tcl**
  - By default selected
  - Uncheck if do not want to copy

Solution Wizard

Solution Configuration
Create Vivado HLS solution for selected technology

Solution Name: solution2

Clock
Period: 10   Uncertainty:

Part Selection
Part: *xc7z020clg484-1*

Options
☑ Copy existing directives from solution: solution1 ▼
☑ Copy existing custom constraints from solution: solution1 ▼

Finish   Cancel

© Copyright 2013 Xilinx

Outline

- **Adding Directives**
- *Improving Latency*
  - Manipulating Loops
- **Improving Throughput**
- **Performance Bottleneck**
- **Summary**

© Copyright 2013 Xilinx

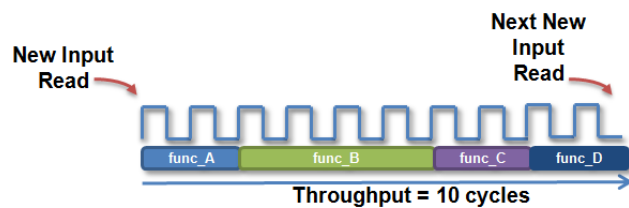# Latency and Throughput – The Performance Factors

Design Latency

– The latency of the design is the number of cycle it takes to output the result

- In this example the latency is 10 cycles

```
void foo_top (a,b,c,d, *x, *y) {
    ...
    func_A(...);
    func_B(...);
    func_C(...)
    func_D(...)
    return res;
}
```

func_A
func_B
func_C
func_D

func_A   func_B   func_C   func_D

Latency = 10 cycles
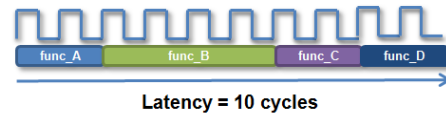
Design Throughput

– The throughput of the design is the number of cycles between new inputs

- By default (no concurrency) this is the same as latency
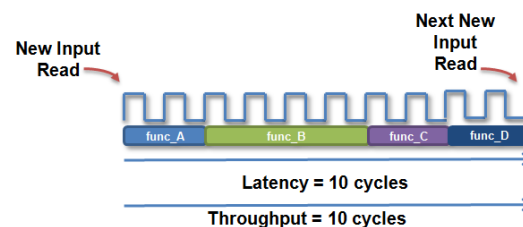- Next start/read is when this transaction ends

New Input Read

Next New Input Read

func_A   func_B   func_C   func_D

Throughput = 10 cycles

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Latency and Throughput

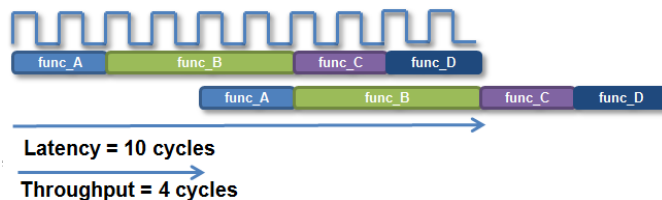## In the absence of any concurrency

– Latency is the same as throughput

```
void foo_top (a,b,c,d, *x, *y) {
    ...
    func_A(...);
    func_B(...);
    func_C(...)
    func_D(...)
    return res;
}
```

func_A
func_B
func_C
func_D

New Input Read

Next New Input Read

func_A   func_B   func_C   func_D

Latency = 10 cycles

Throughput = 10 cycles

Pipelining for higher throughput

– Vivado HLS can pipeline functions and loops to improve throughput

– Latency and throughput are related

– We will discuss optimizing for latency first, then throughput

func_A   func_B   func_C   func_D

func_A   func_B   func_C   func_D

Latency = 10 cycles

Throughput = 4 cycles

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Vivado HLS: Minimize latency

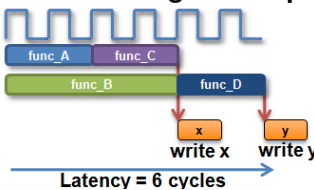- **Vivado HLS will by default minimize latency**
  - Throughput is prioritized above latency (no throughput directive is specified here)
  - In this example
    - The functions are connected as shown
    - Assume function B takes longer than any other functions

```
void foo_top (a,b,c,d, *x, *y) {
    ...
    func_A(a,b,t1);          func_A
    func_B(a,b,t2);          func_B
    func_C(c,t1,&x);         func_C
    func_D(d,t2,&y)          func_D
}
```
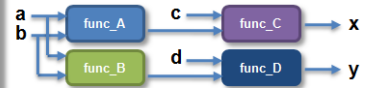
- **Vivado HLS will automatically take advantage of the parallelism**
  - It will schedule functions to start as soon as they can
    - Note it will not do this for loops within a function: by default they are executed in sequence

  write x   write y
  Latency = 6 cycles

  - func_A and func_B can start at the same time
  - func_C can start as soon as func_A completes
  - func_D must wait for func_B to complete
  - Outputs are written as soon as they are ready

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Default Behavior: Minimizing Latency

- **Functions**
  - Vivado HLS will seek to minimize latency by allowing functions to operate in parallel
    - As shown on the previous slide
- **Loops**
  - Vivado HLS will not schedule loops to operate in parallel by default
    - Dataflow optimization must be used or the loops must be unrolled
    - Both techniques are discussed in detail later

```
Loop:for(i=1;i<3;i++) {
    op_Read;               RD
    op_Compute;            CMP
    op_Write;              WR
}
```

- **Operations**
  - Vivado HLS will seek to minimize latency by allowing the operations to occur in parallel
  - It does this within functions <u>and</u> within loops

```
void foo(...) {
    op_Read;               RD
    op_Compute;            CMP
    op_Write;              WR
}
```
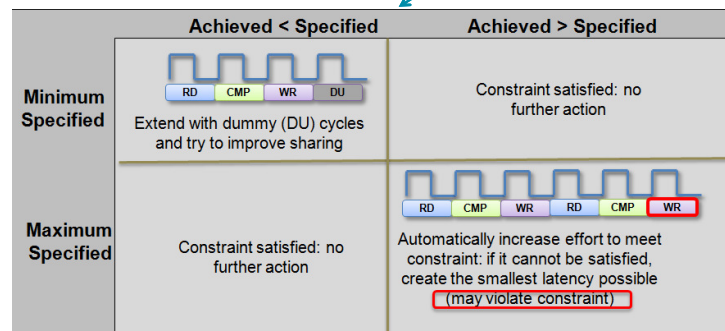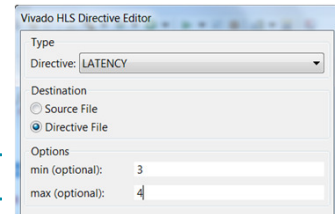
Example with Sequential Operations

RD   CMP   WR

Example of Minimizing latency with Parallel Operations

RD   WR
CMP

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Latency Constraints

Latency constraints can be specified

- Can define a minimum and/or maximum latency for the location
  - This is applied to all objects in the specified scope
- No range specification: schedule for minimum
  - Which is the default

Impact of ranges

	Achieved < Specified	Achieved > Specified
Minimum Specified	RD CMP WR DU — Extend with dummy (DU) cycles and try to improve sharing	Constraint satisfied: no further action
Maximum Specified	Constraint satisfied: no further action	RD CMP WR RD CMP WR — Automatically increase effort to meet constraint: if it cannot be satisfied, create the smallest latency possible (may violate constraint)

Vivado HLS Directive Editor
Type
Directive: LATENCY
Destination
Source File
Directive File
Options
min (optional): 3
max (optional): 4

XILINX ALL PROGRAMMABLE.

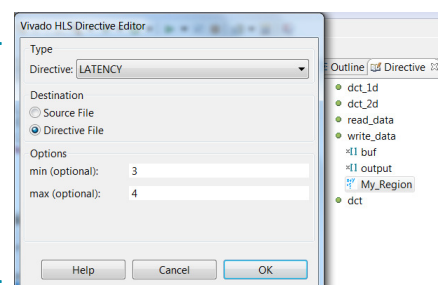# Region Specific Latency Constraint

Latency directives can be applied on functions, loops and regions

Use regions to specify <u>specific</u> locations for latency constraints

- A region is any set of named braces {…a region…}
  - The region My_Region is shown in this example
- This allows the constraint to be applied to a specific range of code
  - Here, only the else branch has a latency constraint

```
int write_data (int buf, int output) {

if (x < y) {
    return (x + y);
} else {

    My_Region: {

        return (y – x) * (y + x);

    }

}
```

Select the region in the Directives tab & right-click to apply latency directive

Vivado HLS Directive Editor
Type
Directive: LATENCY
Destination
Source File
Directive File
Options
min (optional): 3
max (optional): 4
Help   Cancel   OK

Outline Directive
dct_1d
dct_2d
read_data
write_data
buf
output
My_Region
dct

XILINX ALL PROGRAMMABLE.

Outline

> **Adding Directives**
> **Improving Latency**
>   – *Manipulating Loops*
> **Improving Throughput**
> **Performance Bottleneck**
> **Summary**

© Copyright 2013 Xilinx
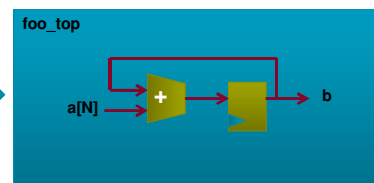
XILINX ➤ ALL PROGRAMMABLE.

Review: Loops

> **By default, loops are rolled**
>   – Each C loop iteration ➔ Implemented in the same state
>   – Each C loop iteration ➔ Implemented with same resources

```
void foo_top (…) {
  ...
  Add: for (i=3;i>=0;i--) {
      b = a[i] + b;
  ...
  }
```

**Loops require labels if they are to be referenced by Tcl directives
(GUI will auto-add labels)**

Synthesis

foo_top

a[N] → + → b

N

>   – Loops can be unrolled if their indices are statically determinable at elaboration time
>   • Not when the number of iterations is variable

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Rolled Loops Enforce Latency

➤ **A rolled loop can only be optimized so much**

– Given this example, where the delay of the adder is small compared to the clock frequency

```
void foo_top (…) {
    ...
    Add: for (i=3;i>=0;i--) {
        b = a[i] + b;
    ...
    }
```
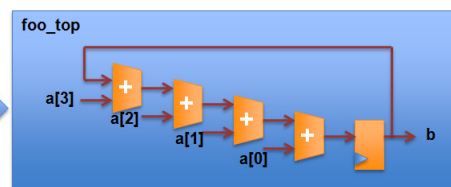
Clock

Adder Delay   3   2   1   0

– This rolled loop will never take less than 4 cycles

• No matter what kind of optimization is tried

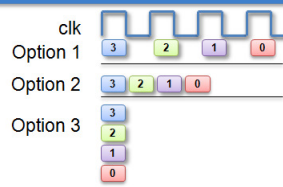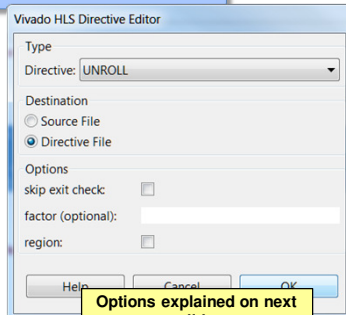• This minimum latency is a function of the loop iteration count

£ XILINX ➤ ALL PROGRAMMABLE.

Unrolled Loops can Reduce Latency

```
void foo_top (…) {
    ...
    Add: for (i=3;i>=0;i--) {
        b = a[i] + b;
    ...
    }
```

Unrolled

foo_top

a[3]   +
        a[2]   +
                a[1]   +
                        a[0]   +   b

clk

Option 1   3   2   1   0

Option 2   3 2 1 0

Option 3   3
            2
            1
            0

Unrolled loops allow greater option & exploration

Select loop "Add" in the directives pane and right-click

Vivado HLS Directive Editor

Type
Directive: UNROLL

Destination
○ Source File
● Directive File

Options
skip exit check: ☐
factor (optional):
region: ☐

Help   Cancel   OK

Options explained on next slide

Unrolled loops are likely to result in more hardware resources and higher area

£ XILINX ➤ ALL PROGRAMMABLE.

Partial Unrolling

> **Fully unrolling loops can create a lot of hardware**

> **Loops can be partially unrolled**

– Provides the type of exploration shown in the previous slide

> **Partial Unrolling**

– A standard loop of N iterations can be unrolled to by a factor

– For example <u>unroll by a factor 2,</u> to have N/2 iterations

• Similar to writing new code as shown on the right ➔

• The break accounts for the condition when N/2 is not an integer

– If "i" is known to be an integer multiple of N

• The user can <u>remove the exit check</u> (and associated logic)

• Vivado HLS is not always be able to determine this is true (e.g. if N is an input argument)

• User takes responsibility: verify!

```
Add: for(int i = 0; i < N; i++) {
   a[i] = b[i] + c[i];
}
```

```
Add: for(int i = 0; i < N; i += 2) {
   a[i] = b[i] + c[i];
   if (i+1 >= N) break;
   a[i+1] = b[i+1] + c[i+1];
}
```

Effective code after compiler transformation

```
for(int i = 0; i < N; i += 2) {
   a[i] = b[i] + c[i];
   a[i+1] = b[i+1] + c[i+1];
}
```
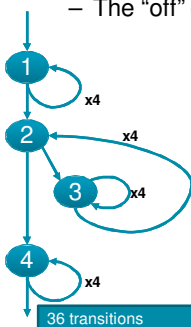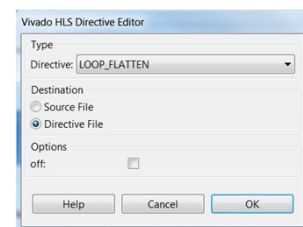
An extra adder for N/2 cycles trade-off

XILINX ➤ ALL PROGRAMMABLE.

Loop Flattening

> **Vivado HLS can automatically flatten nested loops**

– A faster approach than manually changing the code

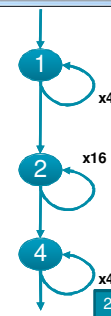> **Flattening should be specified on the inner most loop**

– It will be flattened into the loop above

– The "off" option can prevent loops in the hierarchy from being flattened

Vivado HLS Directive Editor

Type

Directive: LOOP_FLATTEN

Destination
○ Source File
● Directive File

Options
off: ☐

Help   Cancel   OK

```
void foo_top (…) {
   ...
   L1: for (i=3;i>=0;i--) {
      [loop body l1 ]
   }

   L2: for (i=3;i>=0;i--) {
   L3: for (j=3;j>=0;j--) {
      [loop body l3 ]
      }
   }

   L4: for (i=3;i>=0;i--) {
      [loop body l4 ]
   }
}
```

```
void foo_top (…) {
   ...
   L1: for (i=3;i>=0;i--) {
      [loop body l1 ]
   }

   L2: for (k=15,k>=0;k--) {

      [loop body l3 ]
   }

   L4: for (i=3;i>=0;i--) {
      [loop body l1 ]
   }
}
```

Loops will be flattened by default: use "off" to disable

36 transitions

28 transitions

XILINX ➤ ALL PROGRAMMABLE.

Perfect and Semi-Perfect Loops

❯ Only perfect and semi-perfect loops can be flattened
  – The loop should be labeled or directives cannot be applied
  – Perfect Loops
    – Only the inner most loop has body (contents)
    – There is no logic specified between the loop statements
    – The loop bounds are constant
  – Semi-perfect Loops
    – Only the inner most loop has body (contents)
    – There is no logic specified between the loop statements
    – The outer most loop bound can be variable
  – Other types
    – Should be converted to perfect or semi-perfect loops

```
Loop_outer: for (i=3;i>=0;i--) {
   Loop_inner: for (j=3;j>=0;j--) {
      [loop body]
   }
}
```

```
Loop_outer: for (i=3;i>N;i--) {
   Loop_inner: for (j=3;j>=0;j--) {
      [loop body]
   }
}
```

```
Loop_outer: for (i=3;i>N;i--) {
   [loop body]           🚫
   Loop_inner: for (j=3;j>=M;j--) {
      [loop body]         🚫
   }
}
```

☰ XILINX ❯ ALL PROGRAMMABLE.

Loop Merging

❯ Vivado HLS can automatically merge loops
  – A faster approach than manually changing the code
  – Allows for more efficient architecture explorations
  – <u>FIFO reads, which must occur in strict order, can prevent loop merging</u>
    • Can be done with the "force" option : user takes responsibility for correctness
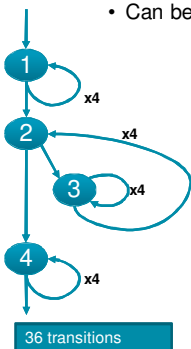

```
void foo_top (…) {
   ...
L1: for (i=3;i>=0;i--) {
   [Loop body l1 ]
}

L2: for (i=3;i>=0;i--) {
   L3: for (j=3;j>=0;j--) {
      [Loop body l3 ]
   }
}

L4: for (i=3;i>=0;i--) {
   [Loop body l4 ]
}
```

Already flattened

```
void foo_top (…) {
   ...
L123: for (l=16,l>=0;l--) {
   if (cond1)
      [loop body l1 ]

   [Loop body l3 ]

   if (cond4)
      [loop body l4 ]
}
```

36 transitions

18 transitions

☰ XILINX ❯ ALL PROGRAMMABLE.
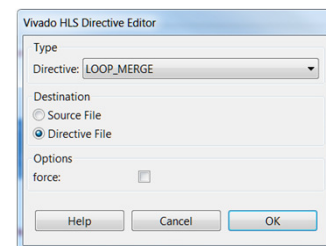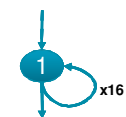
Loop Merge Rules

- **If loop bounds are all variables, they must have the same value**
- **If loops bounds are constants, the maximum constant value is used as the bound of the merged loop**
  - As in the previous example where the maximum loop bounds become 16 (implied by L3 flattened into L2 before the merge)
- **Loops with both variable bound and constant bound cannot be merged**
- **The code between loops to be merged cannot have side effects**
  - Multiple execution of this code should generate same results
    - A=B is OK, A=B+1 is not
- **Reads from a FIFO or FIFO interface must always be in sequence**
  - A FIFO read in one loop will not be a problem
  - FIFO reads in multiple loops may become out of sequence
    - This prevents loops being merged

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Loop Reports

- **Vivado HLS reports the latency of loops**
  - Shown in the report file and GUI
- **Given a variable loop index, the latency cannot be reported**
  - Vivado HLS does not know the limits of the loop index
  - This results in latency reports showing unknown values
- **The loop tripcount (iteration count) can be specified**
  - Apply to the loop in the directives pane
  - Allows the reports to show an estimated latency

Impacts reporting – not synthesis

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Techniques for Minimizing Latency

Constraints

– Vivado HLS accepts constraints for latency

Loop Optimizations

– Latency can be improved by minimizing the number of loop boundaries
  - Rolled loops (default) enforce sharing at the expense of latency
  - The entry and exits to loops costs clock cycles

ΣXILINX ➤ ALL PROGRAMMABLE.

Outline

Adding Directives
Improving Latency
– Manipulating Loops
Improving Throughput
Performance Bottleneck
Summary

ΣXILINX ➤ ALL PROGRAMMABLE.

Improving Throughput

❯ **Given a design with multiple functions**
  – The code and dataflow are as shown


```
void foo_top (a,b,c,d, *x, *y) {
    ...
    func_A(a,b,t1);        func_A
    func_B(a,t1,t2);       func_B
    func_C(c,t2,&x);       func_C
    func_D(d,x,&y)         func_D
}
```
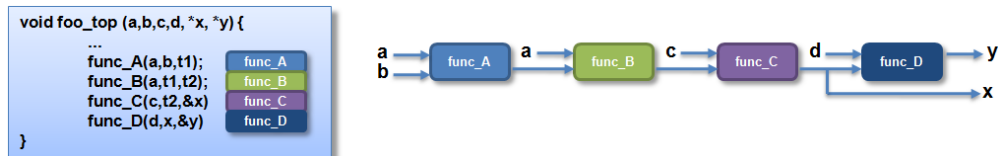
❯ **Vivado HLS will schedule the design**

> The latency is 9 cycles

> The throughput is also 9 cycles

❯ **It can also automatically optimize the dataflow for throughput**
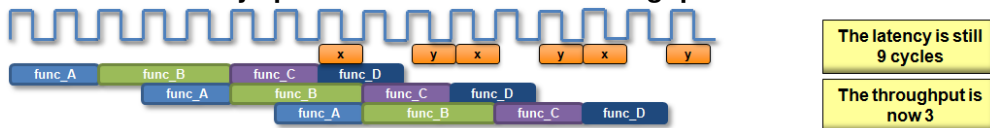
> The latency is still 9 cycles

> The throughput is now 3

XILINX ❯ ALL PROGRAMMABLE.

Dataflow Optimization

❯ **Dataflow Optimization**
  – Can be used at the <u>top-level function</u>
  – Allows blocks of code to operate concurrently
    • The blocks can be <u>functions or loops</u>
    • Dataflow allows loops to operate concurrently
  – It places channels between the blocks to maintain the data rate
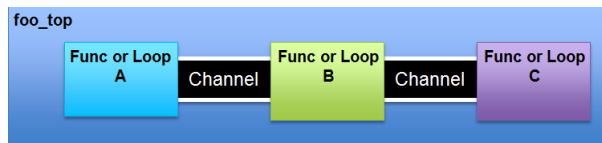
    • For arrays the channels will include memory elements to buffer the samples
    • For scalars the channel is a register with hand-shakes

❯ **Dataflow optimization therefore has an area overhead**
  – Additional memory blocks are added to the design
  – The timing diagram on the previous page should have a memory access delay between the blocks
    • Not shown to keep explanation of the principle clear

XILINX ❯ ALL PROGRAMMABLE.

Dataflow Optimization Commands

> **Dataflow is set using a directive**
> - Vivado HLS will seek to create the highest performance design
>   - Throughput of 1

£ XILINX ➤ ALL PROGRAMMABLE.

Dataflow Optimization through Configuration Command

> **Configuring Dataflow Memories**
> - Between functions Vivado HLS uses ping-pong memory buffers by default
>   - The memory size is defined by the maximum number of producer or consumer elements
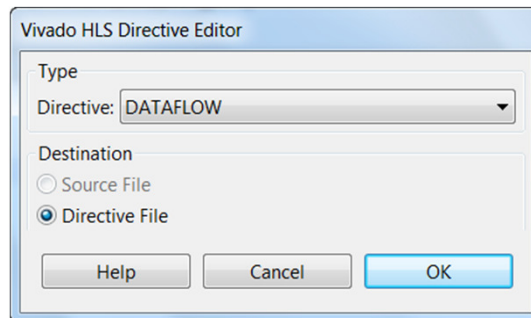> - Between loops Vivado HLS will determine if a FIFO can be used in place of a ping-pong buffer
> - The memories can be specified to be FIFOs using the Dataflow Configuration
>   - Menu: Solution > Solution Settings > config_dataflow
>   - With FIFOs the user can override the default size of the FIFO
>   - Note: Setting the FIFO too small may result in an RTL verification failure

> **Individual Memory Control**
> - When the default is ping-pong
>   - Select an array and mark it as Streaming (directive STREAM) to implement the array as a FIFO
> - When the default is FIFO
>   - Select an array and mark it as Streaming (directive STREAM) with option "off" to implement the array as a ping-pong
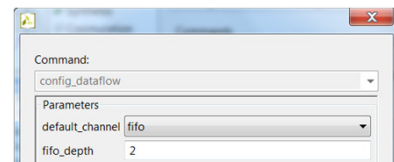
> **To use FIFO's the access must be sequential. If HLS determines that the access is <u>not sequential</u> then it will halt and issue a message. If HLS can not determine the sequential nature then it will issue warning and continue.**

£ XILINX ➤ ALL PROGRAMMABLE.

Dataflow : Ideal for streaming arrays & multi-rate functions

- **Arrays are passed as single entities by default**
  - This example uses loops but the same principle applies to functions

```
int a[N], b[N], c[N];

Loop_1: for (i=0;i<=N-1;i++) {        Loop_1
    b[i] = a[i] + in1;
}

Loop_2: for (i=0;i<=N-1;i++) {        Loop_2
    c[i] = b[i] * in2;
}
```
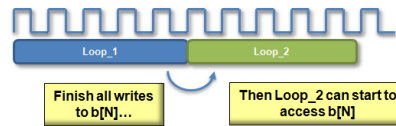
Loop_1   Loop_2

Finish all writes to b[N]...   Then Loop_2 can start to access b[N]

- **Dataflow pipelining allows loop_2 to start when data is ready**
  - The throughput is improved
  - Loops will operate in parallel
    - If dependencies allow

Loop_1

b0 b1 b2 b3

Loop_2
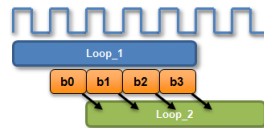
- **Multi-Rate Functions**
  - Dataflow buffers data when one function or loop consumes or produces data at different rate from others

- **IO flow support**
  - To take maximum advantage of dataflow in streaming designs, the IO interfaces at both ends of the datapath should be streaming/handshake types (ap_hs or ap_fifo)

XILINX ALL PROGRAMMABLE.

Pipelining: Dataflow, Functions & Loops

- **Dataflow Optimization**
  - Dataflow optimization is "coarse grain" pipelining at the function and loop level
  - Increases concurrency between functions and loops
  - Only works on functions or loops at the top-level of the hierarchy
    - Cannot be used in sub-functions

- **Function & Loop Pipelining**
  - "Fine grain" pipelining at the level of the operators (*, +, >>, etc.)
  - Allows the operations inside the function or loop to operate in parallel
  - Unrolls all sub-loops inside the function or loop being pipelined
    - Loops with variable bounds cannot be unrolled: This can prevent pipelining
    - Unrolling loops increases the number of operations and can increase memory and run time

XILINX ALL PROGRAMMABLE.

Function Pipelining

Without Pipelining
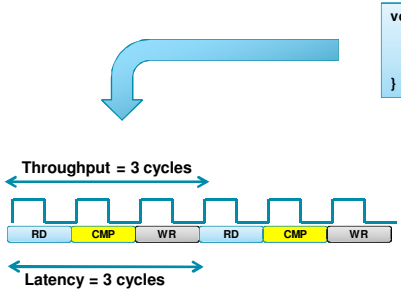
With Pipelining

```
void foo(...) {
    op_Read;          RD
    op_Compute;       CMP
    op_Write;         WR
}
```

Throughput = 3 cycles

| RD | CMP | WR | RD | CMP | WR |

Latency = 3 cycles

Throughput = 1 cycle

| RD | CMP | WR |
| | RD | CMP | WR |

Latency = 3 cycles

> There are 3 clock cycles before operation RD can occur again
  - Throughput = 3 cycles
> There are 3 cycles before the 1st output is written
  - Latency = 3 cycles

> The latency is the same
> The throughput is better
  - Less cycles, higher throughput

XILINX > ALL PROGRAMMABLE.

Loop Pipelining

Without Pipelining

With Pipelining

```
Loop:for(i=1;i<3;i++) {
    op_Read;          RD
    op_Compute;       CMP
    op_Write;         WR
}
```

Throughput = 3 cycles

| RD | CMP | WR | RD | CMP | WR |

Latency = 3 cycles

Loop Latency = 6 cycles

Throughput = 1 cycle

| RD | CMP | WR |
| | RD | CMP | WR |

Latency = 3 cycles

Loop Latency = 4 cycles
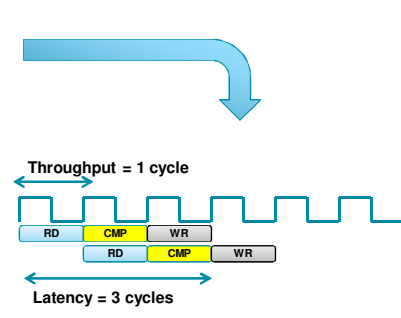
> There are 3 clock cycles before operation RD can occur again
  - Throughput = 3 cycles
> There are 3 cycles before the 1st output is written
  - Latency = 3 cycles
  - For the loop, 6 cycles
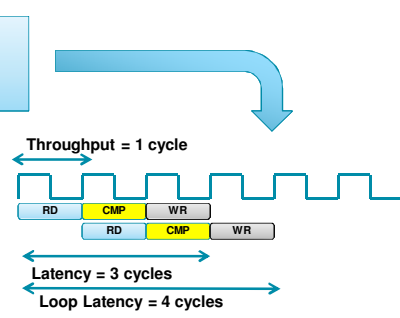
> The latency is the same
- The throughput is better
  - Less cycles, higher throughput
> The latency for all iterations, the loop latency, has been improved

XILINX > ALL PROGRAMMABLE.

Pipelining and Function/Loop Hierarchy

> **Vivado HLS will attempt to unroll all loops nested below a PIPELINE directive**

– May not succeed for various reason and/or may lead to unacceptable area
  - Loops with variable bounds cannot be unrolled
  - Unrolling Multi-level loop nests may create a lot of hardware

– Pipelining the inner-most loop will result in best performance for area
  - Or next one (or two) out if inner-most is modest and fixed
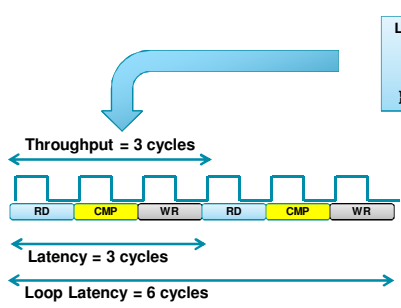   ▪ e.g. Convolution algorithm
  - Outer loops will keep the inner pipeline fed

```
void foo(in1[ ][ ], in2[ ][ ], …) {
…
 L1:for(i=1;i<N;i++) {
  L2:for(j=0;j<M;j++) {
#pragma AP PIPELINE
   out[i][j] = in1[i][j] + in2[i][j];
  }
 }
}
```

1adder, 3 accesses

```
void foo(in1[ ][ ], in2[ ][ ], …) {
…
 L1:for(i=1;i<N;i++) {
#pragma AP PIPELINE
  L2:for(j=0;j<M;j++) {
   out[i][j] = in1[i][j] + in2[i][j];
  }
 }
}
```

Unrolls L2 M adders, 3M accesses

```
void foo(in1[ ][ ], in2[ ][ ], …) {
#pragma AP PIPELINE
…
 L1:for(i=1;i<N;i++) {
  L2:for(j=0;j<M;j++) {
   out[i][j] = in1[i][j] + in2[i][j];
  }
 }
}
```

Unrolls L1 and L2 N*M adders, 3(N*M) accesses

XILINX > ALL PROGRAMMABLE.

Pipelining Commands

> **The pipeline directive pipelines functions or loops**

– This example pipelines the function with an Initiation Interval (II) of 2
  - The II is the same as the throughput but this term is used exclusively with pipelines

RD	CMP	WR

	RD	CMP	WR

Initiation Interval (or II)

> **Omit the target II and Vivado HLS will Automatically pipeline for the fastest possible design**

– Specifying a more accurate maximum may allow more sharing (smaller area)

Vivado HLS Directive Editor

Type
Directive: PIPELINE

Destination
○ Source File
● Directive File

Options
II (optional): 2
enable flushing: ☐

Help | Cancel | OK
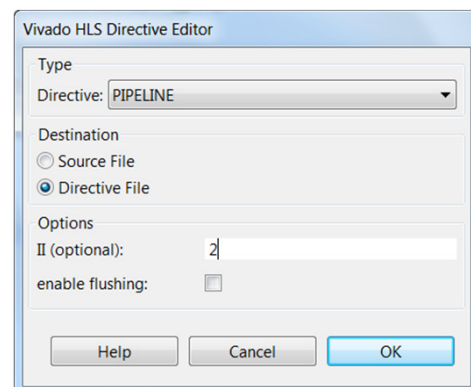
XILINX > ALL PROGRAMMABLE.

Pipeline Flush

> **Pipelines can optionally be flushed**
- – Flush: when the input enable goes low (no more data) all existing results are flushed out
  - • The input enable may be from an input interface or from another block in the design
- – The default is to stall all existing values in the pipeline

Without Flush (default)

Clk							
Data Valid							

RD	CMP	WR		out1		
	RD	CMP		WR	out2	
		RD		CMP		

With Flush (optional)

Clk							
Data Valid							

RD	CMP	WR	out1		
	RD	CMP	WR	out2	
			RD	CMP	

> **With Flush**
- – When no new input reads are performed
- – Values already in the pipeline are flushed out

XILINX ➤ ALL PROGRAMMABLE.

Pipelining the Top-Level Loop

> **Loop Pipelining top-level loop may give a "bubble"**
- – A "bubble" here is an interruption to the data stream
- – Given the following

```
void foo_top (in1, in2, …) {

    static accum=0;
    …
    L1:for(i=1;i<N;i++) {
        accum = accum + in1 + in2;
    }
    out1_data = accum;
    …
}
```

Init	L1: Accum	Init	L1: Accum
	Out		Out

Pipelined processing of data	"bubble"	Next transaction : Pipelined processing of data
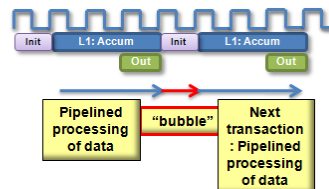
- – The function will process a stream of data
- – The next time the function is called, it still needs to execute the initial (init) operations
  - • These operations are any which occur before the loop starts
  - • These operations may include interface start/stop/done signals
- – This can result in an unexpected interruption of the data stream

XILINX ➤ ALL PROGRAMMABLE.

Continuous Pipelining the Top-Level loop

> **Use the "rewind" option for continuous pipelining**
- Immediate re-execution of the top-level loop
- The operation rewinds to the start of the loop
  - Ignores any initialization statements before the start of the loop

```
void foo_top (in1, in2, ...) {

    static accum=0;
    ...
    L1:for(i=1;i<N;i++) {
        accum = accum + in1 + in2;
    }
    out1_data = accum;
    ...
}
```

| Init | L1: Accum | L1: Accum | L1: Accum |

| Out | Out | Out |

Pipelined processing of data. **Next transaction: Pipelined processing of data**
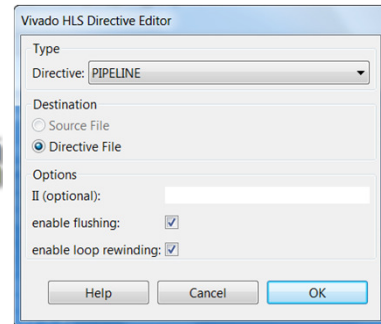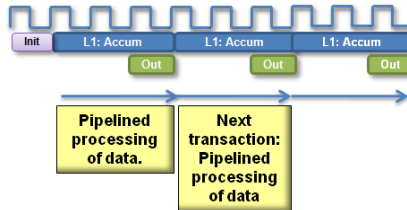
Vivado HLS Directive Editor

Type
Directive: PIPELINE

Destination
○ Source File
● Directive File

Options
II (optional):
enable flushing: ☑
enable loop rewinding: ☑

Help   Cancel   OK

> **The rewind portion _only_ effects top-level loops**
- Ensures the operations before the loop are never re-executed when the function is re-executed

© Copyright 2013 Xilinx

XILINX ► ALL PROGRAMMABLE.

Issues which prevent Pipelining

> **Pipelining functions unrolls all loops**
- Loops with variable bounds cannot be unrolled
- This will prevent pipelining
  - Re-code to remove the variables bounds: max bounds with an exit

> **Feedback prevent/limits pipelines**
- Feedback within the code will prevent or limit pipelining
  - The pipeline may be limited to higher initiation interval (more cycles, lower throughput)

```
void foo_top (a,b,c,d) {
    ...
    while (a != b) {
        if (a > b)
            a -= b;
        else
            b -= a;
        ...
    }
}
```

| != | > | - |

| != | > | - |

Comparison cannot occur before the subtraction

II=1
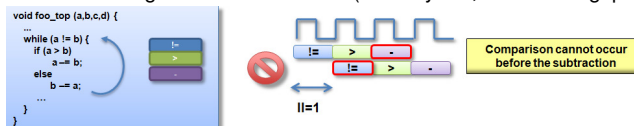
> **Resource Contention may prevent pipelining**
- Can occur within input and output ports/arguments
- This is a classis way in which arrays limit performance

© Copyright 2013 Xilinx

XILINX ► ALL PROGRAMMABLE.

Resource Contention: Unfeasible Initiation Intervals

- **Sometimes the II specification cannot be met**
  - In this example there are 2 read operations on the same port

(A) Pipeline with II=1


```
void foo(m[2]...) {

    op_Read_m[0];     RD
    op_Read_m[1];     RD
    op_Compute;       CMP
    op_Write;         WR

}
```

(B) Pipeline with II=2
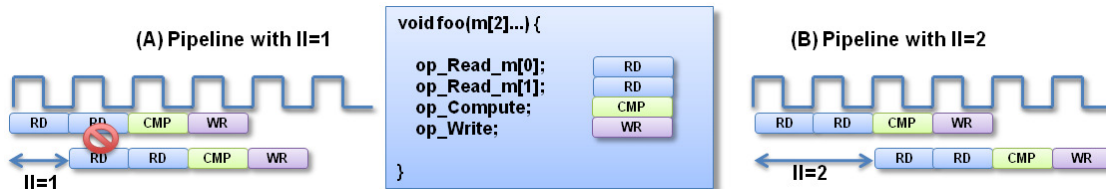
  - An II=1 cannot be implemented
    - The same port cannot be read at the same time
    - Similar effect with other resource limitations
    - For example if functions or multipliers etc. are limited
- **Vivado HLS will automatically increase the II**
  - Vivado HLS will always try to create a design, even if constraints must be violated

ΣXILINX ➤ ALL PROGRAMMABLE.

Outline

- **Adding Directives**
- **Improving Latency**
  - Manipulating Loops
- **Improving Throughput**
- *Performance Bottleneck*
- **Summary**

ΣXILINX ➤ ALL PROGRAMMABLE.

Arrays : Performance bottlenecks

Arrays are intuitive and useful software constructs
- They allow the C algorithm to be easily captured and understood

Array accesses can often be performance bottlenecks
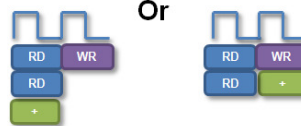- Arrays are targeted to a default RAM
  - May not be the most ideal memory for performance

```
void foo_top (…) {
    ...
    for (i = 2; i < N; i++)
        mem[i] = mem[i-1] +mem[i-2];

    }

}
```

RD RD + WR

Or

RD WR
RD
+

RD WR
RD
+

Even with a dual-port RAM, we cannot perform all reads and writes in one cycle

  - Cannot pipeline with a throughput of 1

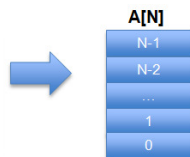**Vivado HLS allows arrays to be partitioned and reshaped**
- Allows more optimal configuration of the array
- Provides better implementation of the memory resource

XILINX ➤ ALL PROGRAMMABLE.

Review: Arrays in HLS

An array in C code is implemented by a memory in the RTL
- By default, arrays are implemented as RAMs, optionally a FIFO

```
void foo_top(int x, …)
{
    int A[N];
    L1: for (i = 0; i < N; i++)
        A[i+x] = A[i] + i;
}
```

A[N]

N-1
N-2
...
1
0

Synthesis

foo_top

SPRAMB

A_in → DIN   DOUT → A_out
→ ADDR

→ CE
→ WE
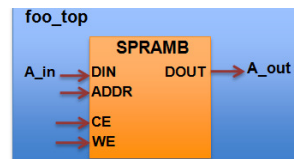
The array can be targeted to any memory resource in the library
- The ports and sequential operation are defined by the library model
  - All RAMs are listed in the Vivado HLS Library Guide

Vivado HLS Directive Editor

Type
Directive: RESOURCE

Destination
○ Source File
● Directive File

Options
variable (required):   buf_2d_in

core (required):

port map (optional):

metadata (optional):

Help   Cancel   OK

Vivado HLS core selection...

MulnS [functional_unit]
RAM_1P [storage]
RAM_1P_BRAM [storage]
RAM_1P_LUTRAM [storage]
RAM_2P [storage]
RAM_2P_1S [storage]
RAM_2P_BRAM [storage]
RAM_2P_LUTRAM [storage]
RAM_T2P_BRAM [storage]
ROM_1P [storage]
ROM_1P_1S [storage]
ROM_1P_BRAM [storage]
ROM_1P_LUTRAM [storage]
ROM_2P [storage]
ROM_2P_BRAM [storage]
ROM_2P_LUTRAM [storage]

OK   Cancel

List of available Cores

XILINX ➤ ALL PROGRAMMABLE.

Array and RAM selection

➤ **If no RAM resource is selected**
  – Vivado HLS will determine the RAM to use
    • It will use a Dual-port if it improves throughput
    • Else it will use a single-port
➤ **BRAM and LUTRAM selection**
  – If none is made (e.g. resource RAM_1P used) RTL synthesis will determine if RAM is implemented as BRAM or LUTRAM
  – If the user specifies the RAM target (e.g. RAM_1P_BRAM or RAM_1P_LUTRAM is selected) Vivado HLS will obey the target
    • If LUTRAM is selected Vivado HLS reports registers not BRAM

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Array Partitioning

➤ **Partitioning breaks an array into smaller elements**
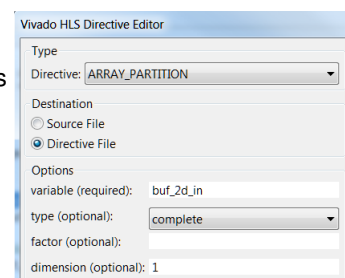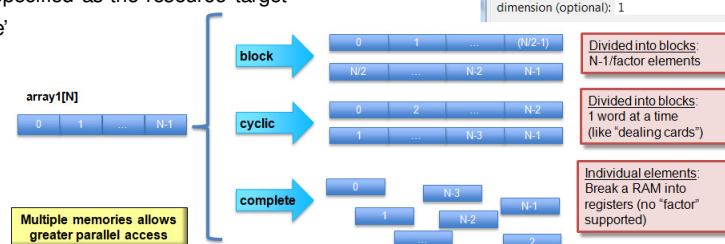 – If the factor is not an integer multiple the final array has fewer elements
  – Arrays can be split along any dimension
    • If none is specified dimension zero is assumed
    • Dimension zero means all dimensions
  – All partitions inherit the same resource target
    • That is, whatever RAM is specified as the resource target
    • Except of course "complete"

Vivado HLS Directive Editor

Type
Directive: ARRAY_PARTITION

Destination
○ Source File
● Directive File

Options
variable (required): buf_2d_in
type (optional): complete
factor (optional):
dimension (optional): 1

array1[N]

block — Divided into blocks: N-1/factor elements

cyclic — Divided into blocks: 1 word at a time (like "dealing cards")

complete — Individual elements: Break a RAM into registers (no "factor" supported)

Multiple memories allows greater parallel access

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Configuring Array Partitioning

> **Vivado HLS can automatically partition arrays to improve throughput**
  - This is controlled via the array configuration command
  - Enable mode throughput_driven

> **Auto-partition arrays with constant indexing**
  - When the array index is not a variable
  - Arrays below the threshold are auto-partitioned
  - Set the threshold using option elem_count_limit

> **Partition all arrays in the design**
  - Select option scalarize_all

> **Include all arrays in partitioning**
  - The include_ports option will include any arrays on the IO interface when partitioning is performed
    - Partitioning these arrays will result in multiple ports and change the inteface
    - This may however improve throughput
  - Any arrays defined as a global can be included in the partitioning by selecting option include_extern_globals
    - By default, global arrays are not partitioned

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.
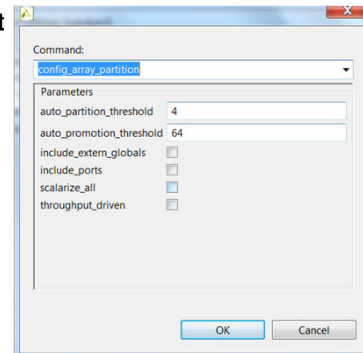
Array Dimensions

> **The array options can be performed on dimensions of the array**

dimension (optional): 1

my_array[10][6][4]

Dimension 1
Dimension 2
Dimension 3

Dimension 0
(All dimensions)
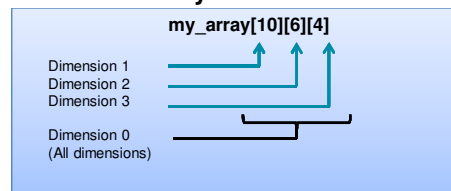
> **Examples**

my_array[10][6][4] ➔ partition dimension 3 ➔

my_array_0[10][6]
my_array_1[10][6]
my_array_2[10][6]
my_array_3[10][6]

my_array[10][6][4] ➔ partition dimension 1 ⟶

my_array_0[6][4]
my_array_1[6][4]
my_array_2[6][4]
my_array_3[6][4]
my_array_4[6][4]
my_array_5[6][4]
my_array_6[6][4]
my_array_7[6][4]
my_array_8[6][4]
my_array_9[6][4]

my_array[10][6][4] ➔ partition dimension 0 ➔ 10x6x4 = 240 individual registers

© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Array Reshaping

➤ **Reshaping recombines partitioned arrays back into a single array**
  – Same options as array partition
  – However, reshape automatically recombines
    the parts back into a single element
  – The "new" array has the same name
    • Same name used for resource targeting

Vivado HLS Directive Editor

Type
Directive: ARRAY_RESHAPE ▼

Destination
○ Source File
◉ Directive File

Options
variable (required):   buf_2d_in
object: ☐
type (optional):   complete ▼
factor (optional):
dimension (optional): 1

block

MSB | N/2 | ... | N-2 | N-1
LSB | 0 | 1 | ... | (N/2-1)

array1[N]

| 0 | 1 | ... | N-1 |

cyclic

MSB | 1 | ... | N-3 | N-1
LSB | 0 | 2 | ... | N-2

Read modify write is not allowed: read the whole word or write the whole word

complete

MSB | N-1 |
| N-2 |
| ... |
| 1 |
LSB | 0 |

Similar to converting a RAM into a very wide register : great access, high throughput
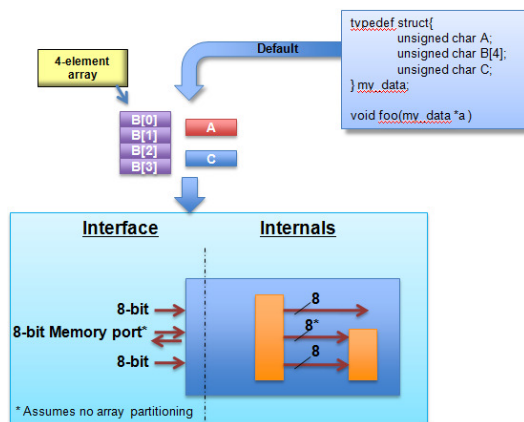
© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Structs and Arrays: The Default Handling

➤ **Structs are a commonly used coding construct**
  – By default, structs are separated into their separate elements

4-element array

Default

```
typedef struct{
        unsigned char A;
        unsigned char B[4];
        unsigned char C;
} my_data;

void foo(my_data *a )
```

B[0]
B[1]
B[2]
B[3]

A

C

Interface   Internals

8-bit →
8-bit Memory port* ⇄
8-bit →

8
8*
8

* Assumes no array partitioning

• **Treated as separate elements**

• **On the Interface**
  – This means separate ports

• **Internally**
  – Separate buses & wires
  – Separate control logic, which may be more complex, slower and increase latency
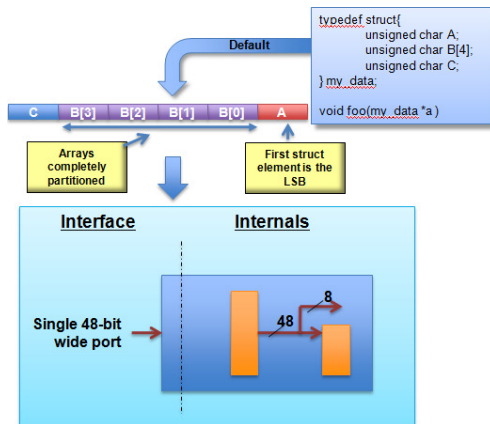
© Copyright 2013 Xilinx

XILINX ➤ ALL PROGRAMMABLE.

Data Packing

> **Data packing groups structs internally and at the IO Interface**
> – Creates a single wide bus of all struct elements

```
typedef struct{
        unsigned char A;
        unsigned char B[4];
        unsigned char C;
} my_data;

void foo(my_data *a )
```

Default

| C | B[3] | B[2] | B[1] | B[0] | A |

Arrays completely partitioned

First struct element is the LSB

Interface **Internals**

Single 48-bit wide port

8

48

- **Grouped structure**
  - First element in the struct becomes the LSB
  - Last struct element becomes the MSB
  - Arrays are partitioning completely
- **On the Interface**
  - This means a single port
- **Internally**
  - Single bus
  - May result in simplified control logic, faster and lower latency designs

ΣXILINX ➤ ALL PROGRAMMABLE.

Outline

> **Adding Directives**
> **Improving Latency**
> – Manipulating Loops
> **Improving Throughput**
> **Performance Bottleneck**
> *Summary*

ΣXILINX ➤ ALL PROGRAMMABLE.

Summary

- **Directives may be added through GUI**
  - Tcl command is added into script.tcl file
  - Pragmas are added into the source file
- **Latency is minimized by default**
  - Constraints can be set
- **Loops may have impact on the latency**
- **Throughput may be improved by pipelining at**
  - The task, function, and loop level
- **Arrays may create performance bottleneck if not handled properly**

XILINX ➤ ALL PROGRAMMABLE.

Summary

- **Optimizing Performance**
  - Latency optimization
    - Specify latency directives
    - Unroll loops
    - Merge and Flatten loops to reduce loop transition overheads
  - Throughput optimization
    - Perform Dataflow optimization at the top-level
    - Pipeline individual functions and/or loops
    - Pipeline the entire function: beware of lots of operations, lots to schedule and it's not always possible
  - Array Optimizations
    - Focus on bottlenecks often caused by memory and port accesses
    - Removing bottlenecks improves latency and throughput
      - Use Array Partitioning, Reshaping, and Data packing directives to achieve throughput

XILINX ➤ ALL PROGRAMMABLE.

Lab2 Intro
Improving Performance

Vivado HLS 2013.3 Version
ZedBoard

Objectives

➤ **After completing this lab, you will be able to:**

– Add directives to your design
– Understand the effect of INLINE-ing functions
– Observe the effect of PIPELINE-ing functions
– Improve the performance using various directives

The Design

> **The design consists of YUV filter typically used in video processing. The design consists of three functions – rgb2yuv, yuv_scale, and yuv2rgb**

  – Each of these functions iterates over the entire source image, requiring a single source pixel to produce a pixel in the result image

  – The scale function simply applies individual scale factors, supplied through top-level arguments

```c
// The top-level function
void yuv_filter (
        image_t *in,
        image_t *out,
        yuv_scale_t Y_scale,
        yuv_scale_t U_scale,
        yuv_scale_t V_scale
        )
{
// Internal image buffers
#ifndef __SYNTHESIS__
    image_t *yuv = (image_t *)malloc(sizeof(image_t));
    image_t *scale = (image_t *)malloc(sizeof(image_t));
#else // Workaround malloc() calls w/o changing rest of code
    image_t _yuv;
    image_t _scale;
    image_t *yuv = &_yuv;
    image_t *scale = &_scale;
#endif

    rgb2yuv    (in, yuv);
    yuv_scale (    yuv, scale, Y_scale, U_scale, V_scale);
    yuv2rgb    (         scale, out);
}
```

XILINX > ALL PROGRAMMABLE.

Procedure

> **Create a Vivado HLS project by executing script from Vivado HLS command prompt**
> **Open the created project in Vivado HLS GUI and analyze**
> **Apply TRIPCOUNT directive using PRAGMA**
> **Apply PIPELINE directive, generate solution, and analyze output**
> **Apply DATAFLOW directive to improve performance**
> **Export and Implement the design**

XILINX > ALL PROGRAMMABLE.

Summary

In this lab you learned that even though this design could not be pipelined at the top-level, a strategy of pipelining the individual loops and then using dataflow optimization to make the functions operate in parallel was able to achieve the same high throughput, processing one pixel per clock. When DATAFLOW directive is applied, the default memory buffers (of ping-pong type) are automatically inserted between the functions. Using the fact that the design used only sequential (streaming) data accesses allowed the costly memory buffers associated with dataflow optimization to be replaced with simple 2 element FIFOs using the Dataflow command configuration

XILINX ➤ ALL PROGRAMMABLE.