Vitis HLS Optimization

AMD
together we advance_

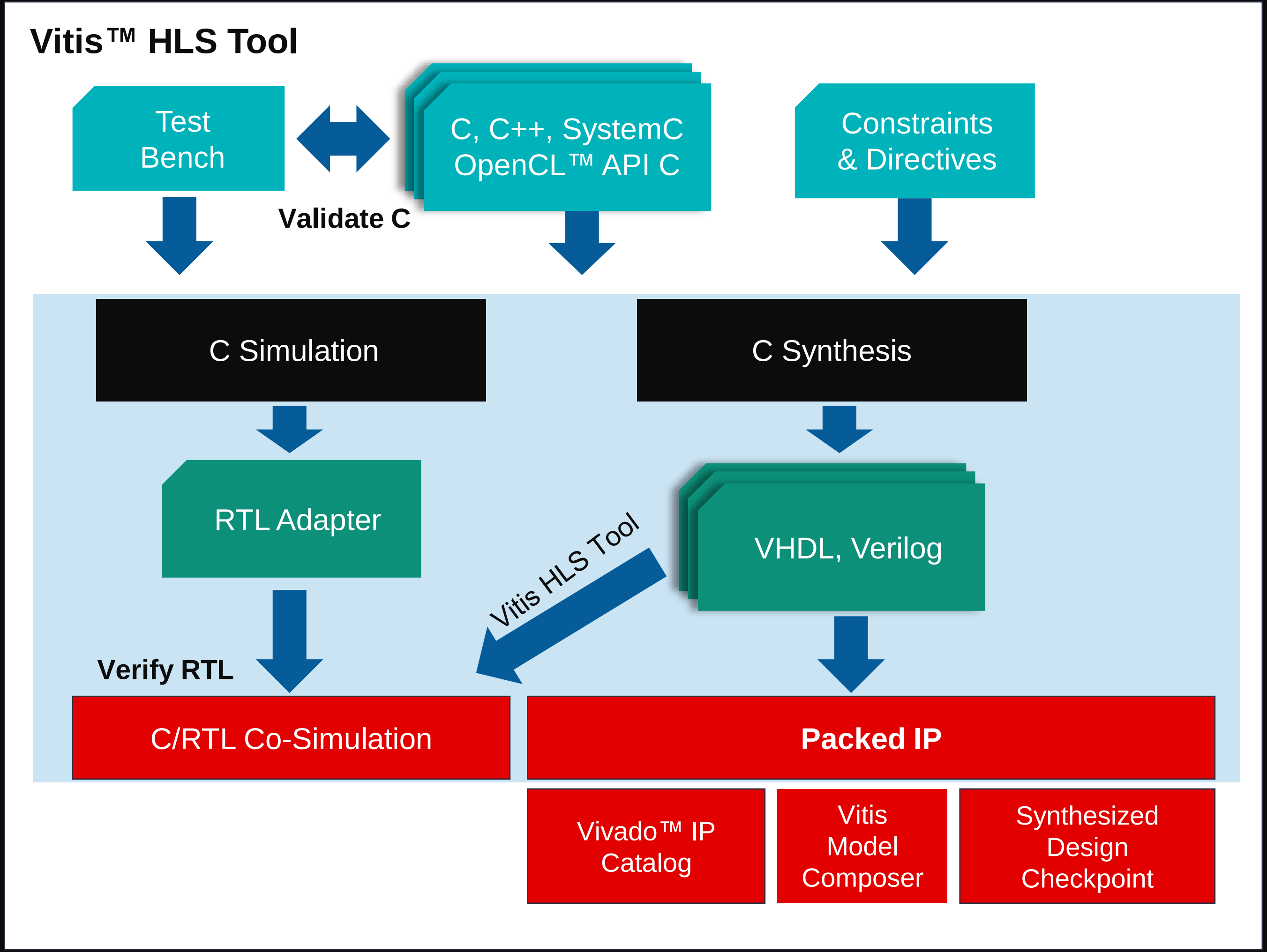
C Validation and RTL Verification

Pre-synthesis stage

- C simulation checks the functionality of the C algorithm
- C validation uses the C test bench

Post-synthesis stage

- Verification is automated through the C/RTL co-simulation feature
- Reuses the C test bench to perform verification on the output RTL



© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Basics of High-Level Synthesis

Vitis HLS tool

- Allows C, C++, and OpenCL™ functions to become hard wired onto the device logic fabric and RAM/DSP blocks
- Implements hardware kernels in the Vitis application acceleration development flow and develops RTL IP for FPGA designs in Vivado Design Suite

Vitis™ HLS tool performs three phases

1 Scheduling

Binding 2

3 Control Logic Extraction

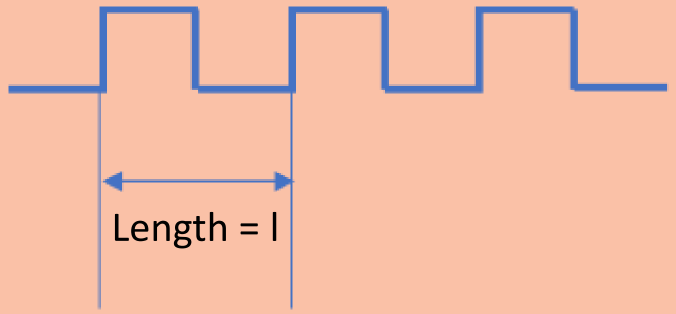
© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

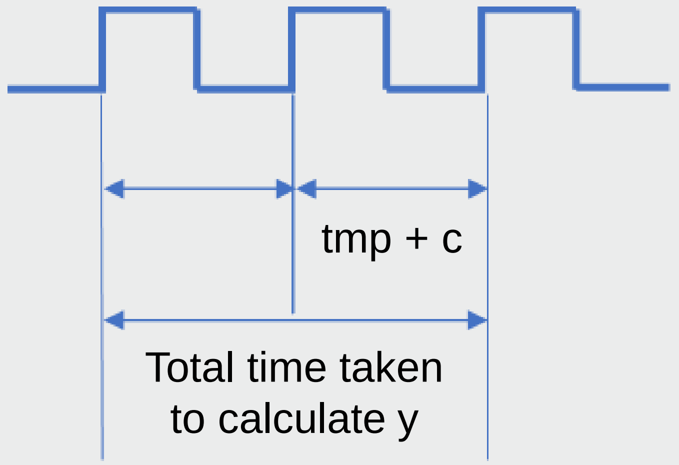
Scheduling

Scheduling determines which operations occur during each clock cycle based on:

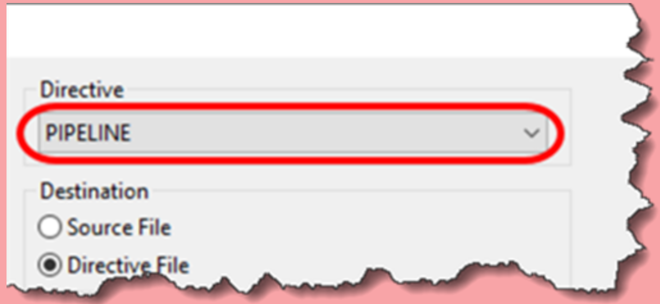
Length of the clock cycle or clock frequency



Time it takes for the operation to complete as defined by the target device



User-specified optimization directives



Directive: PIPELINE

Destination: ( ) Source File [✓ Directive File]

Pipeline, dataflow, interface, etc.

© Copyright 2023 Advanced Micro Devices, Inc.

together we advance_

Scheduling

Determines which operations occur during each clock cycle based on:

If the clock period is longer or a faster FPGA is targeted:	If the clock period is shorter or a slower FPGA is targeted:
More, if not all, operations might complete in one clock cycle	HLS automatically schedules the operations over more clock cycles

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Scheduling and Binding Example

<u>Initial binding phase</u>:

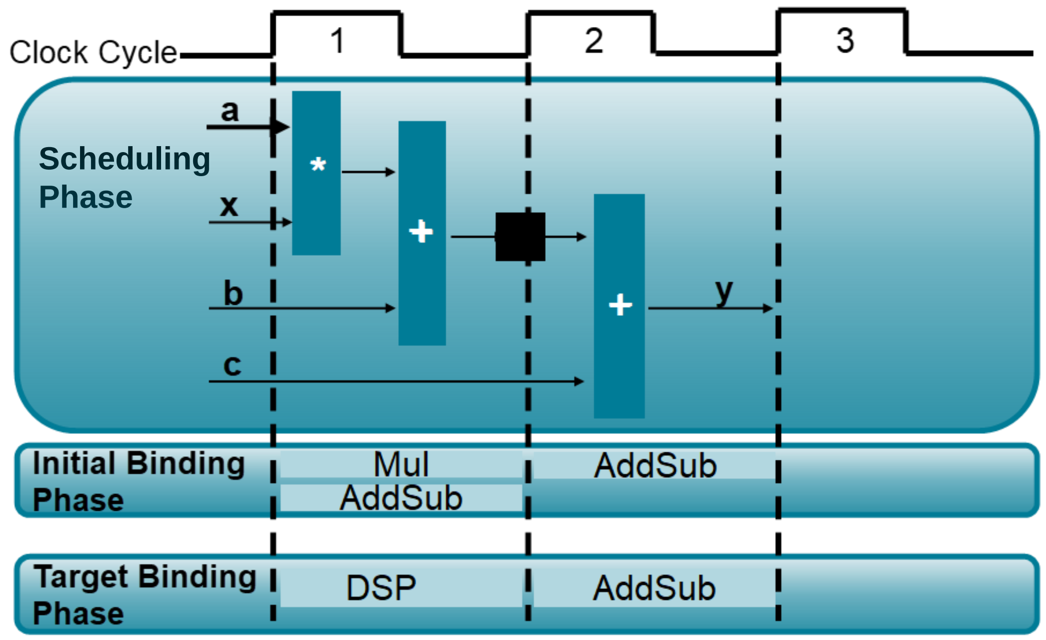
Implements the multiplier operation using a combinational multiplier (Mul)

Implements both add operations using a combinational adder/subtractor (AddSub)

<u>Target binding phase</u>:

Implements both the multiplier and one of the addition operations using a DSP module (computational block that provides the ideal balance of high performance and efficient implementation)

```
int foo(char x, char a, char b, char c) {
    char y;
    y = x*a+b+c;
    return y
}
```



© Copyright 2023 Advanced Micro Devices, Inc.

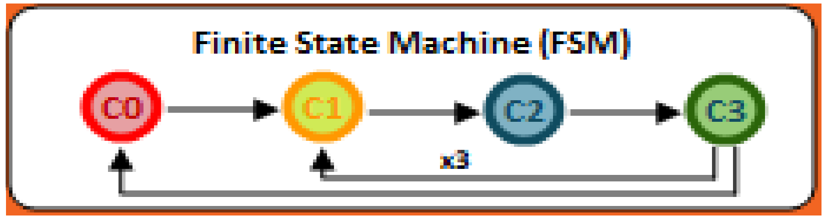
Control Logic Extraction & I/O Port Implementation Example

Example performs the same multiplication and addition operations, but inside a ‘for’ loop

Two of the function arguments are arrays

HLS automatically extracts the control logic from the C code and creates an FSM in the RTL design to sequence these operations

```
void foo (int in [3], char a, char b, char c, int out [3]) {
  int x,y;
  for (int i = 0; i < 3; i++) {
    x = in[ i ];
    y = a*x + b + c;
    out [ i ] = y;
  }
}
```



© Copyright 2023 Advanced Micro Devices, Inc.

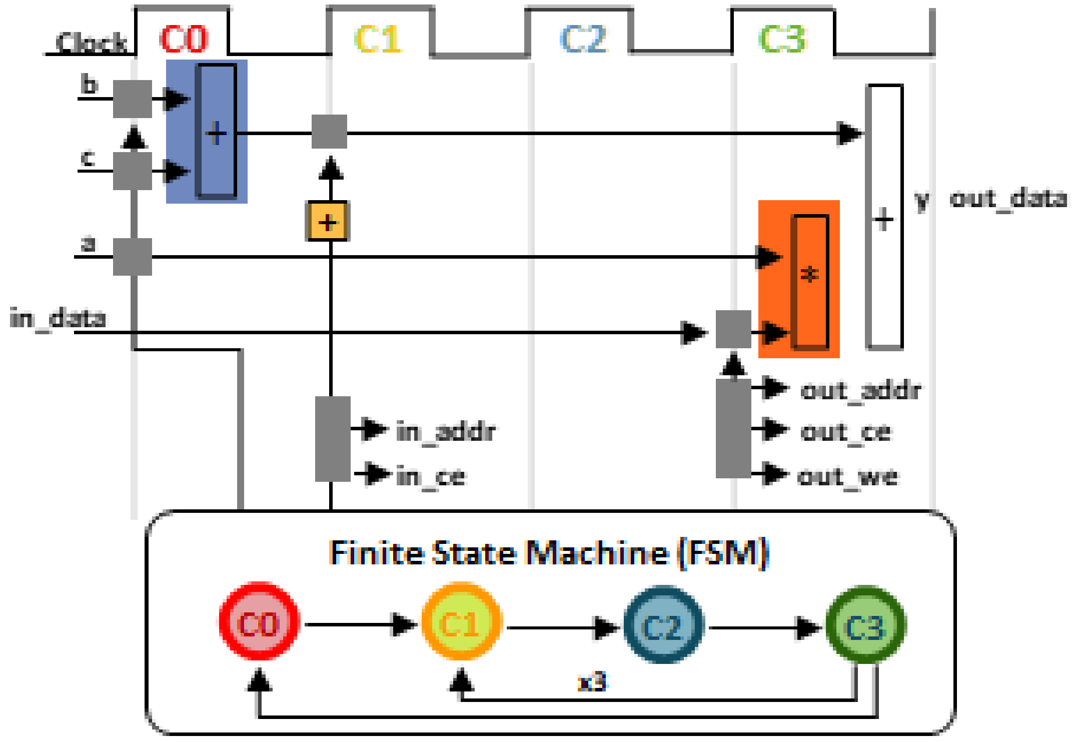
AMD
together we advance_

Control Logic Extraction & I/O Port Implementation Example

FSM controls when the registers store data and the state of any I/O control signals

Starts in state C0—on the next clock, it enters state C1, then state C2, and then C3

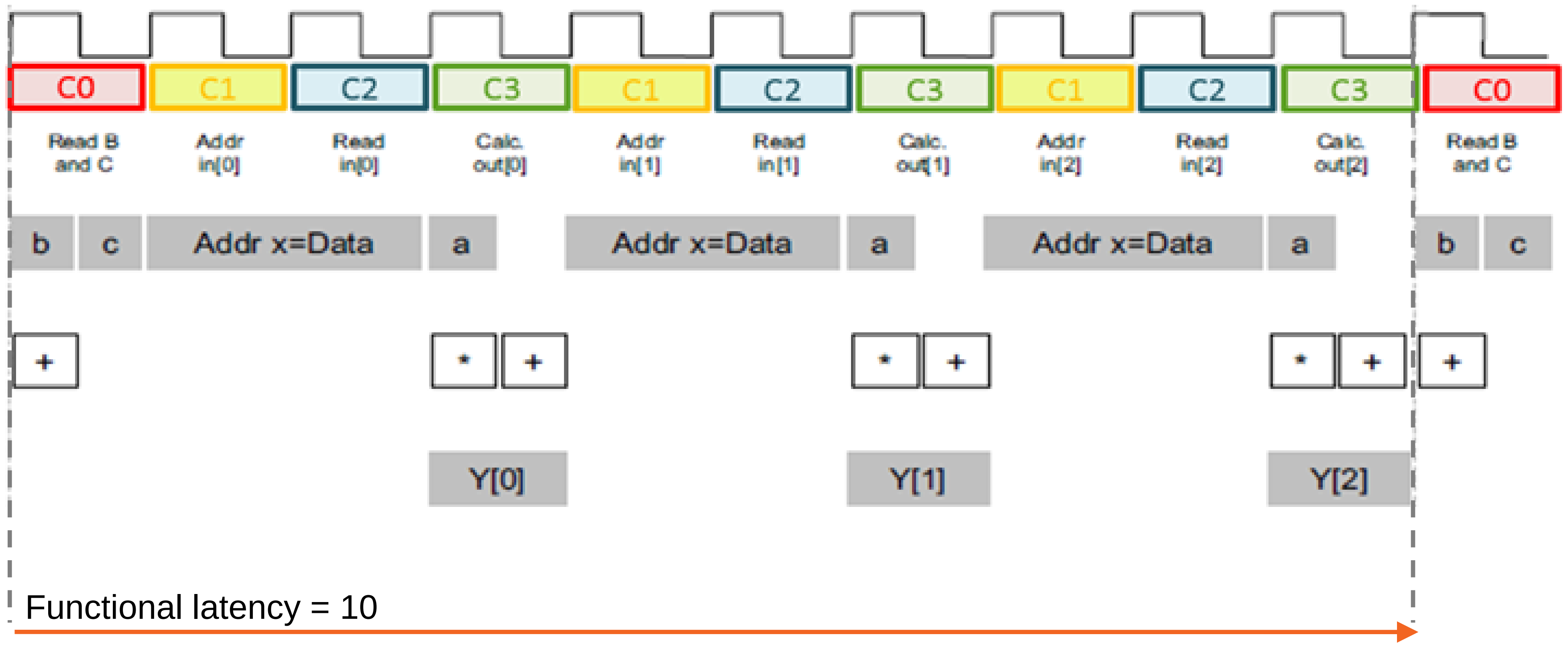
```
void foo (int in [3], char a, char b, char c, int out [3]) {
  int x,y;
  for (int i = 0; i < 3; i++) {
    x = in[ i ];
    y = a*x + b + c;
    out [ i ] = y;
  }
}
```



© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Terminology for Measuring in Clock Cycles



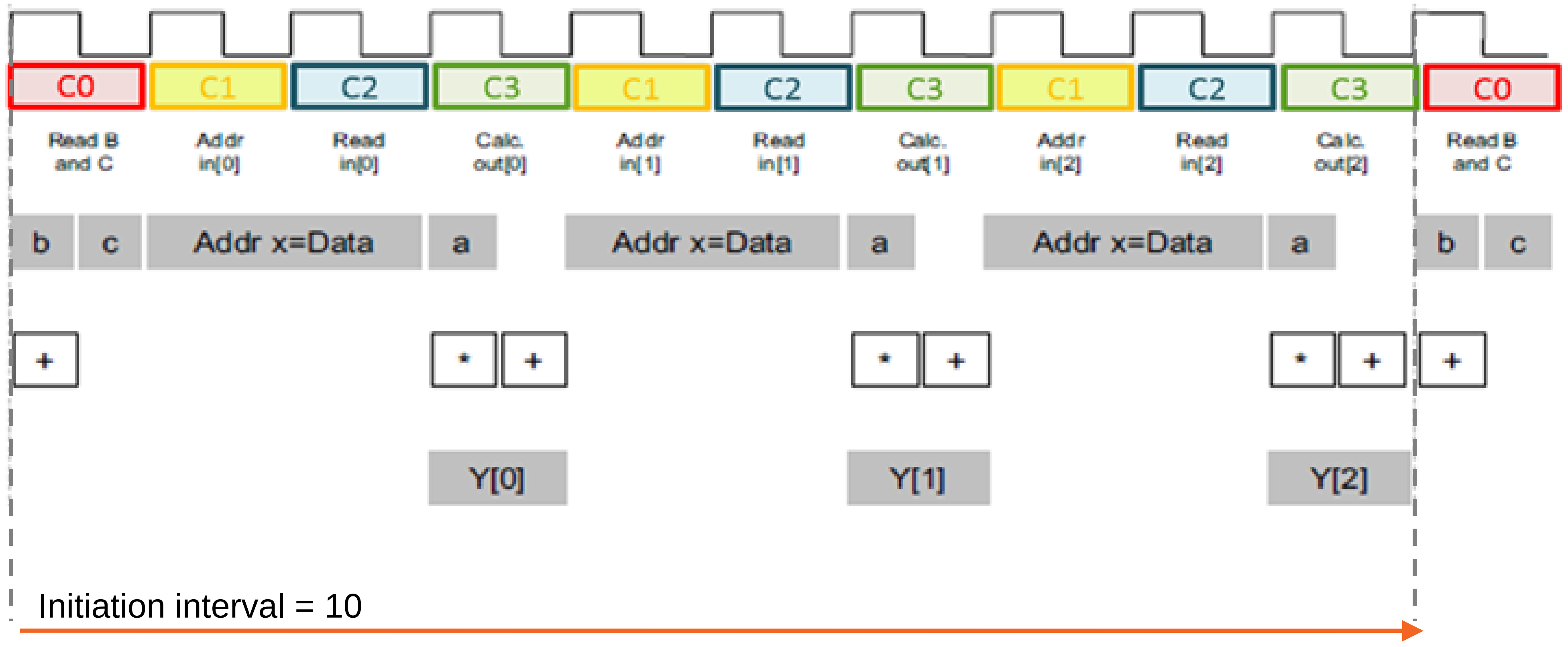
Latency: Number of clock cycles required for the function to go from input to output generation

When the output is an array, the latency is measured to the last array value output

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Terminology for Measuring in Clock Cycles

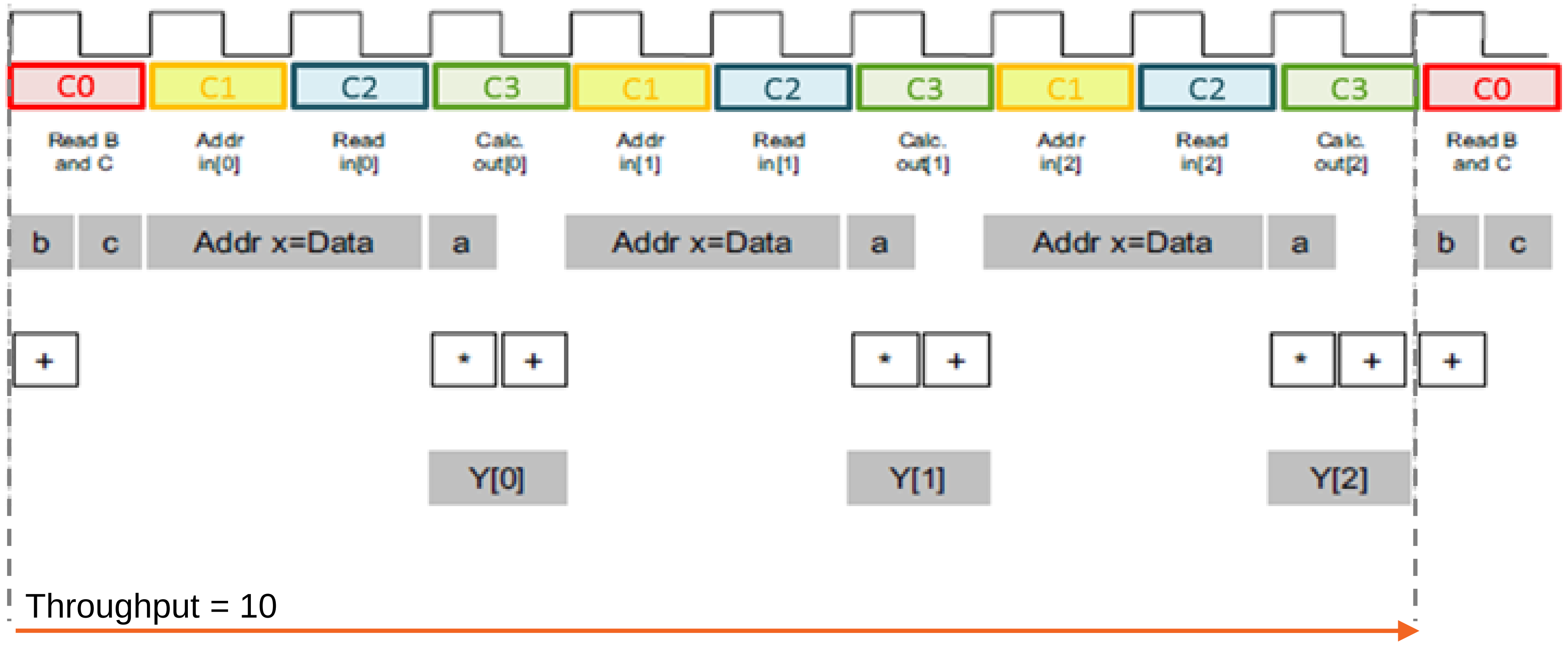


Initiation interval (II): Number of clock cycles before the function can accept new input data

© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

Terminology for Measuring in Clock Cycles

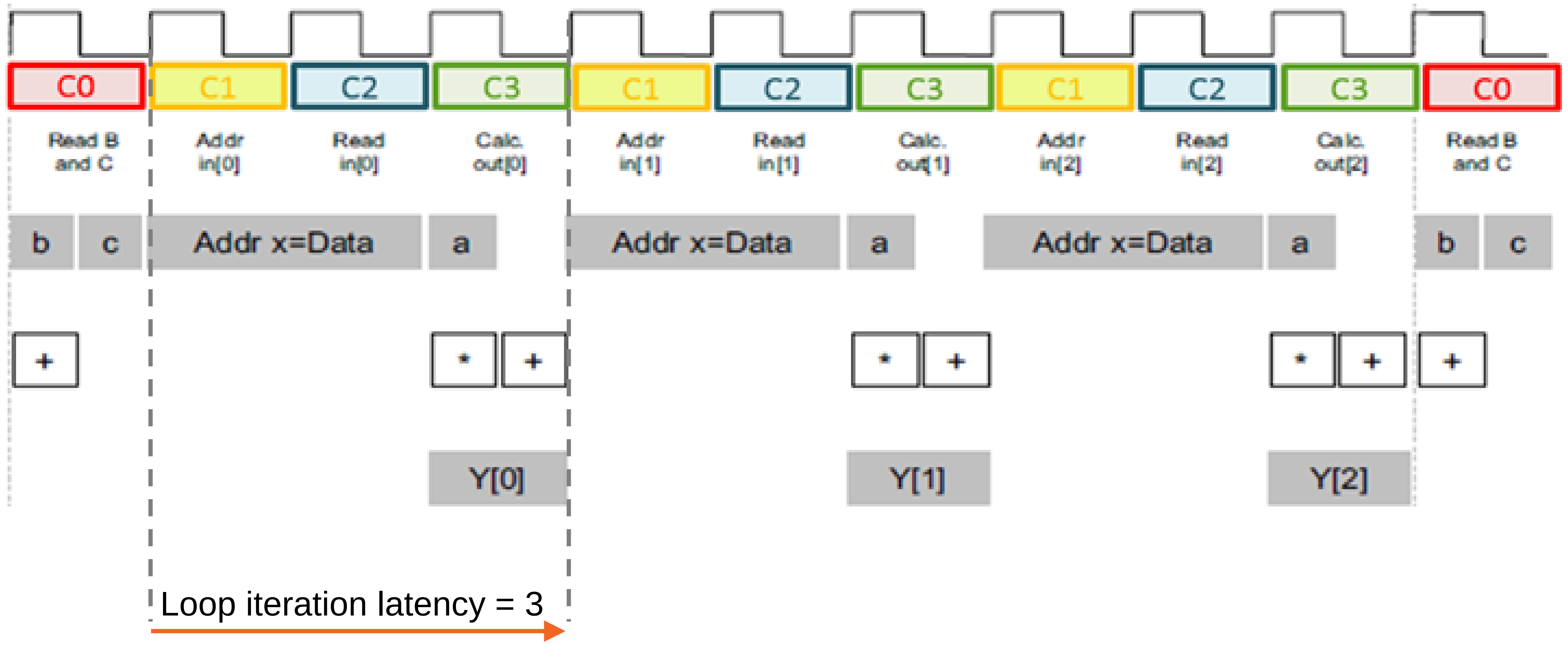


Throughput: Number of cycles between the new input samples

© Copyright 2023 Advanced Micro Devices, Inc.

together we advance_

Terminology for Measuring in Clock Cycles

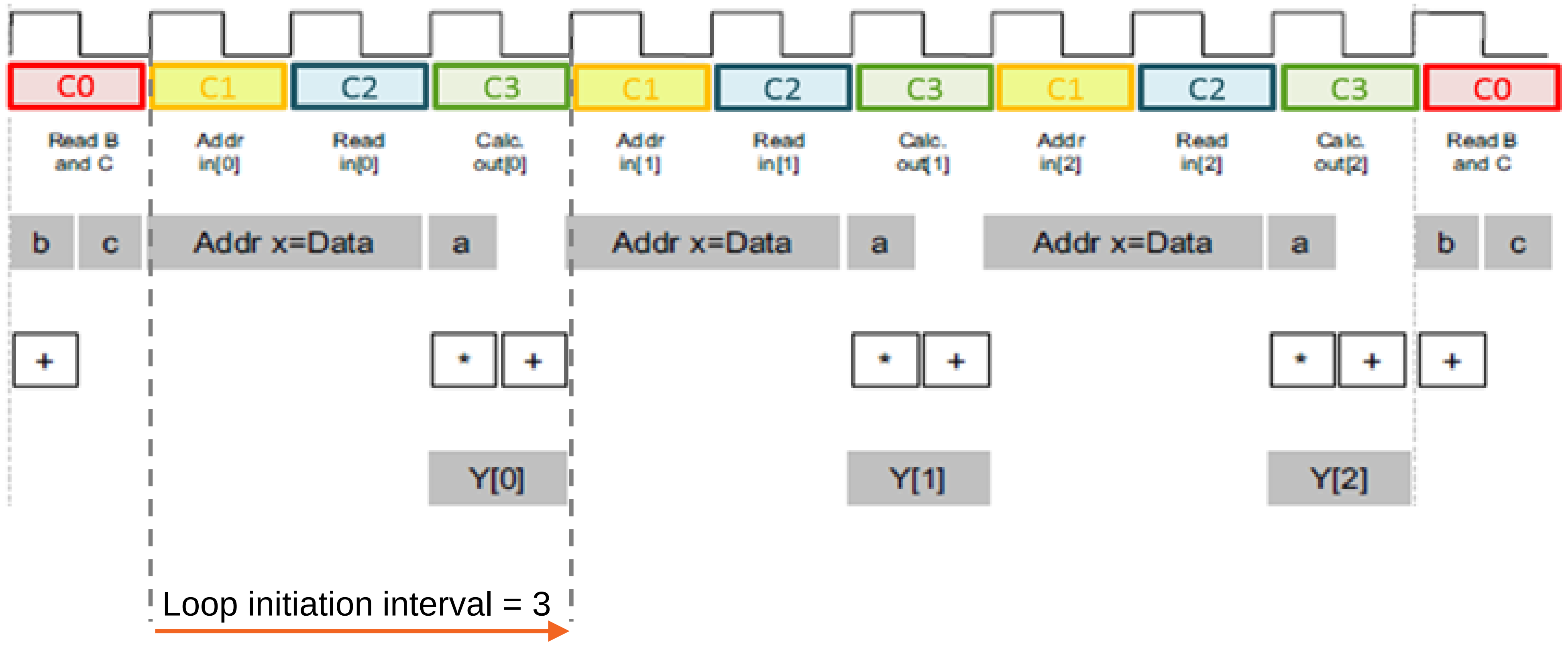


Loop iteration latency: Number of clock cycles it takes to complete one iteration of the loop

© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

Terminology for Measuring in Clock Cycles

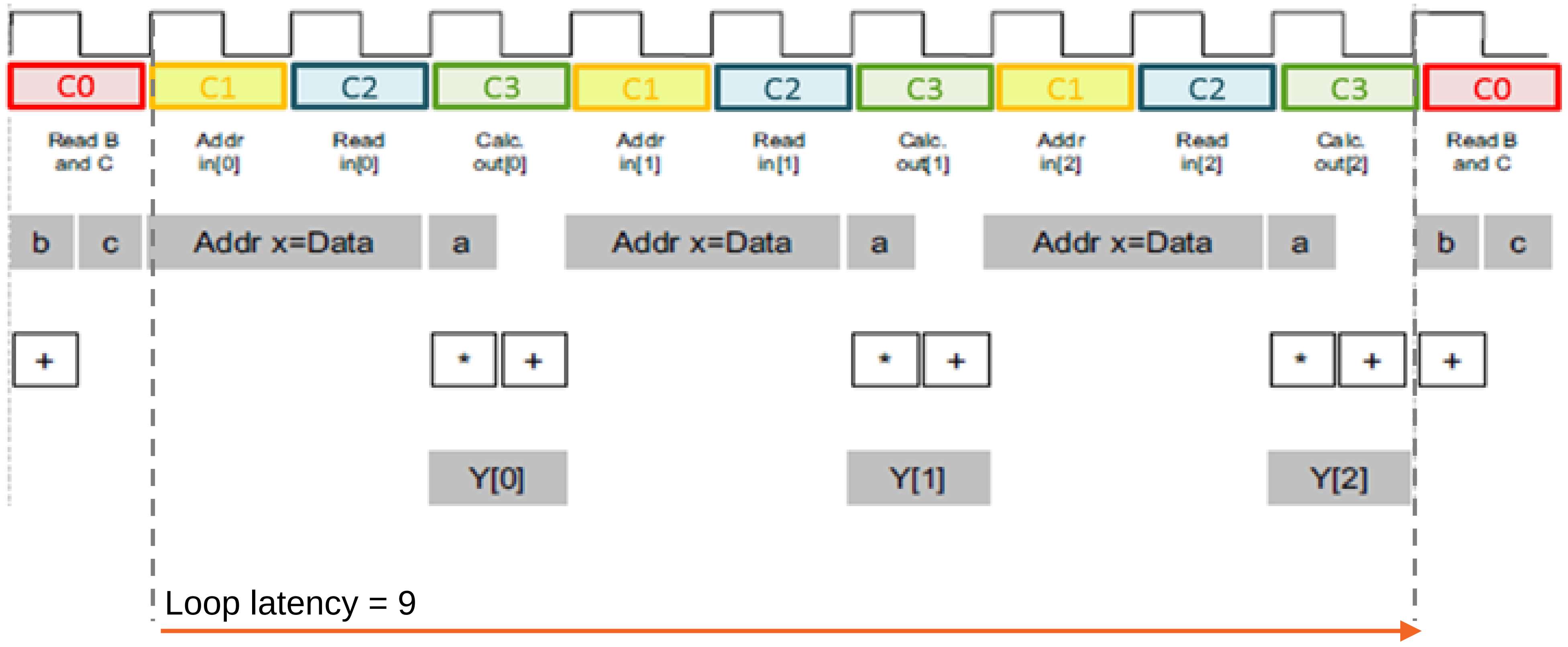


Loop initiation interval: Number of clock cycles before the next iteration of the loop starts to process data

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Terminology for Measuring in Clock Cycles

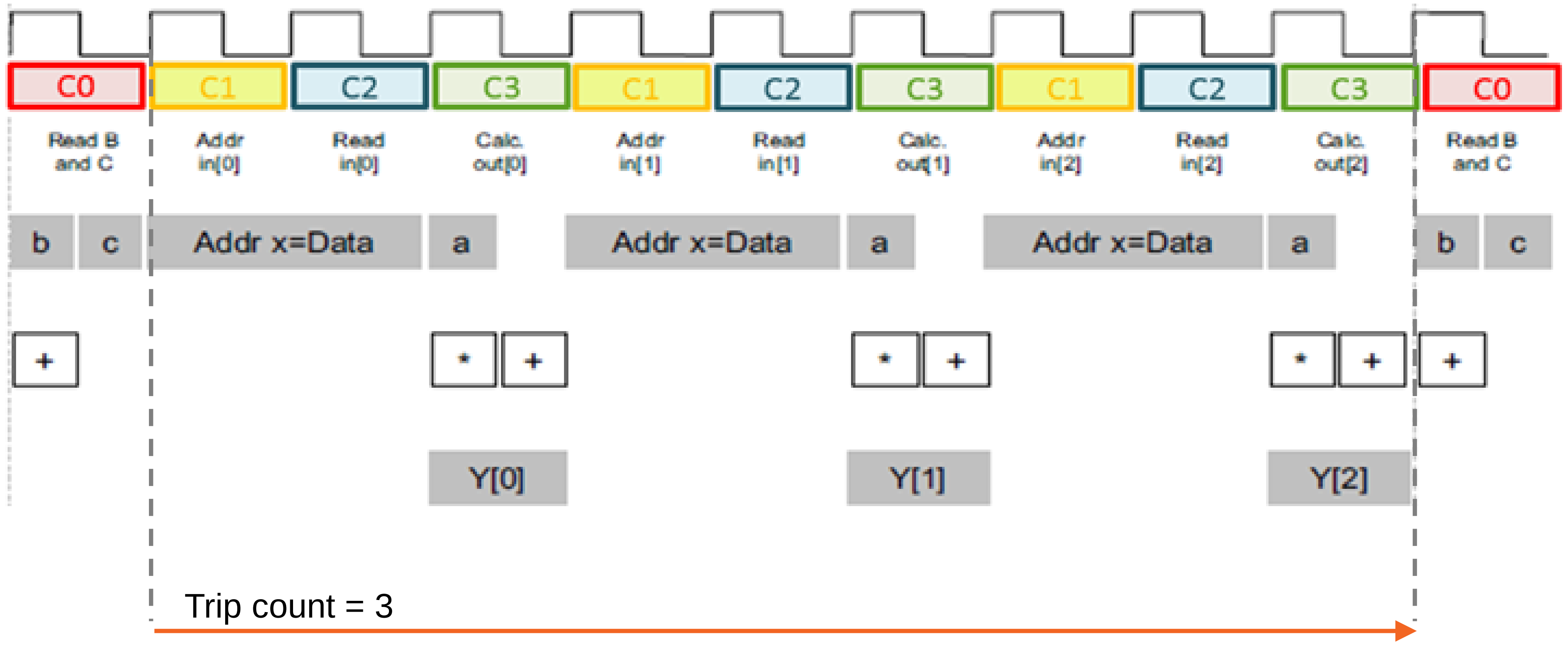


Loop latency: Number of cycles to execute all iterations of the loop

© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

Terminology for Measuring in Clock Cycles



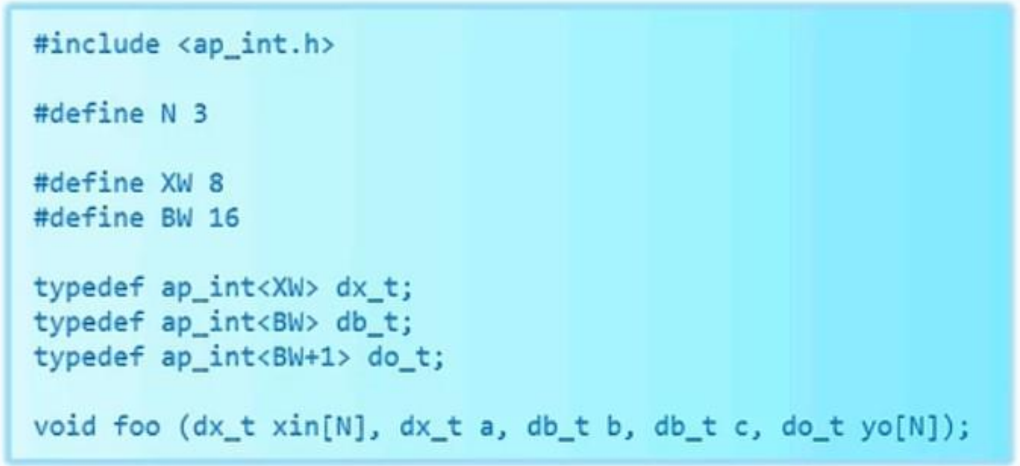
Trip count: Number of iterations in the loop; three in this case

Data rate: Equal to the 1/throughput * clock frequency

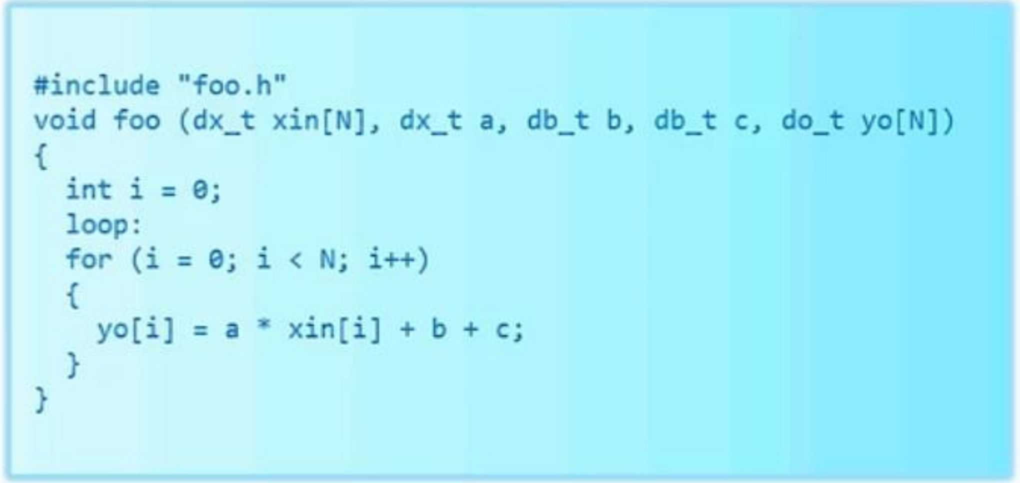
© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

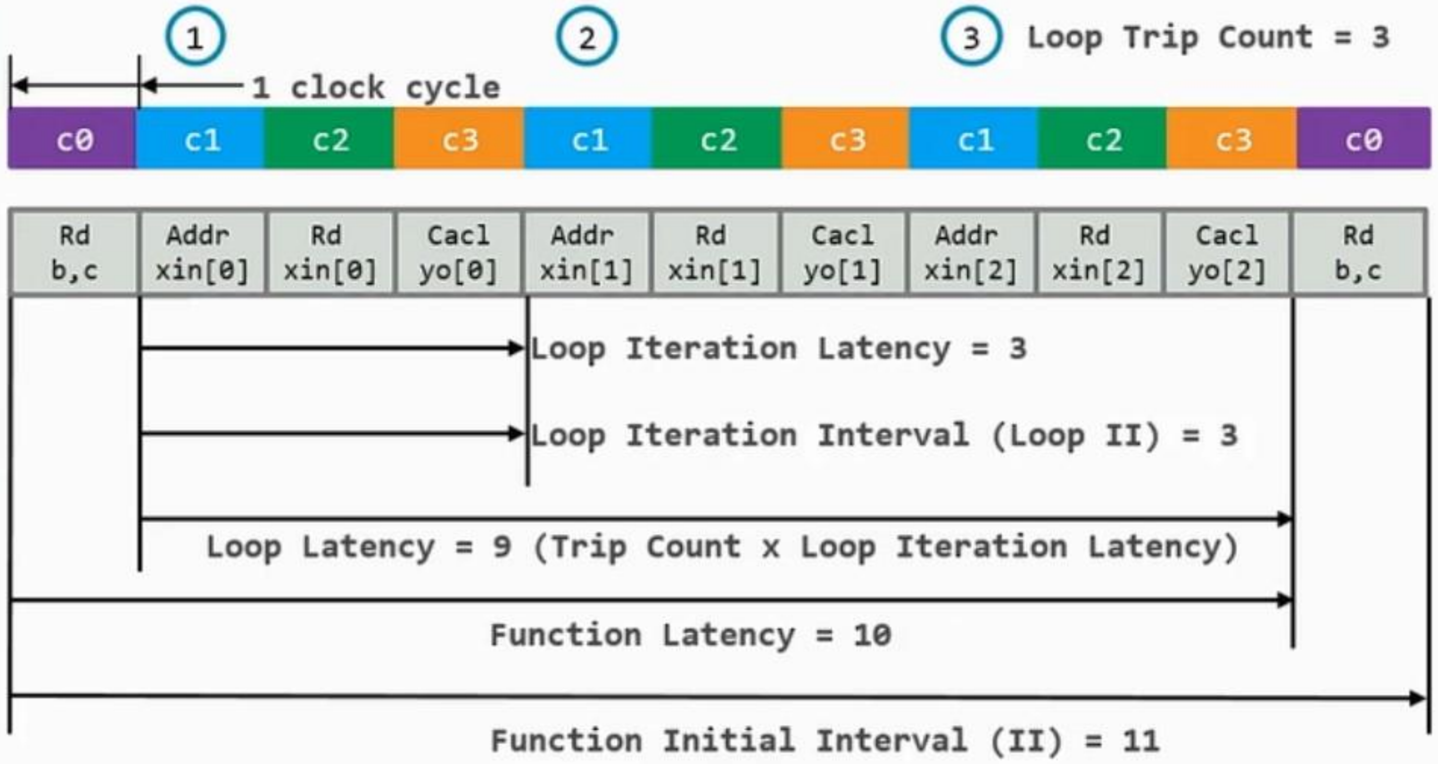
Terminology for Measuring in Clock Cycles



```
#include <ap_int.h>

#define N 3

#define XW 8
#define BW 16

typedef ap_int<XW> dx_t;
typedef ap_int<BW> db_t;
typedef ap_int<BW+1> do_t;

void foo (dx_t xin[N], dx_t a, db_t b, db_t c, do_t yo[N]);
```



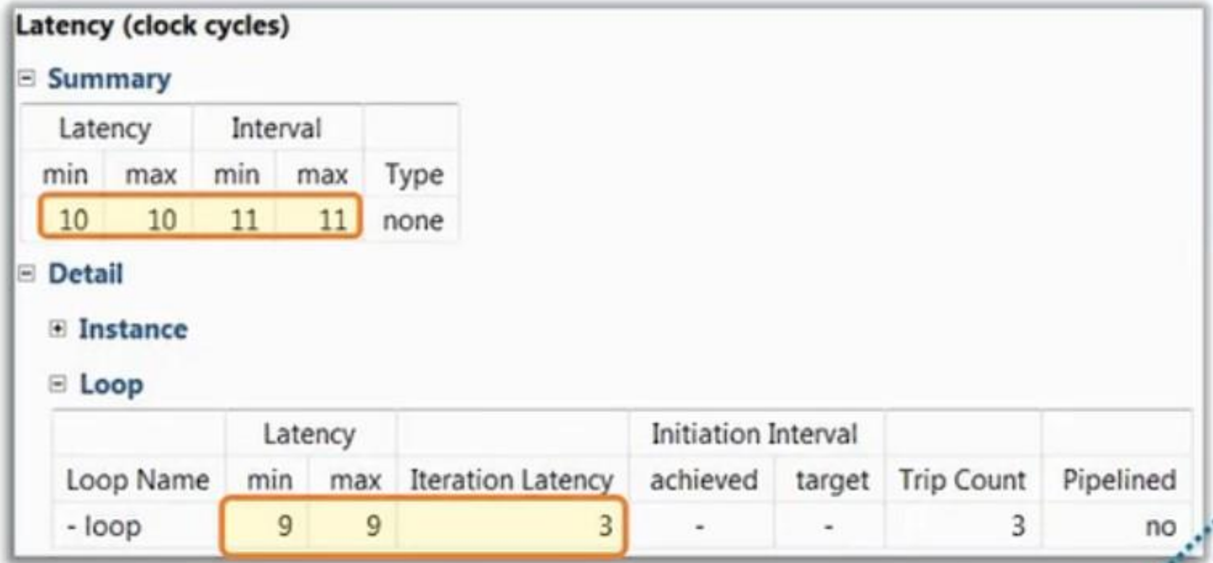
```
#include "foo.h"
void foo (dx_t xin[N], dx_t a, db_t b, db_t c, do_t yo[N])
{
  int i = 0;
  loop:
  for (i = 0; i < N; i++)
  {
    yo[i] = a * xin[i] + b + c;
  }
}
```



© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

Terminology for Measuring in Clock Cycles



Latency (clock cycles)

⊟ **Summary**

Latency		Interval		
min	max	min	max	Type
10	10	11	11	none

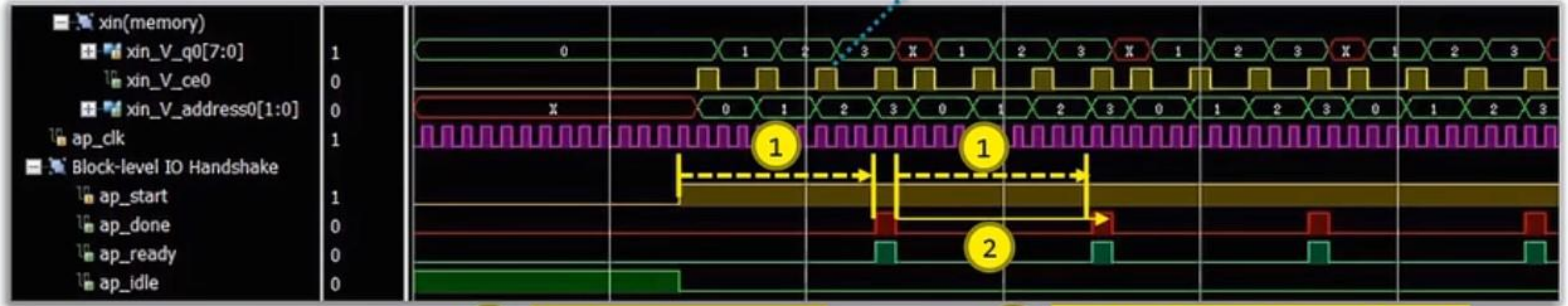
⊟ **Detail**

⊞ **Instance**

⊟ **Loop**

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop	9	9	3	-	-	3	no

Although the last data is available here, it's necessary for HLS to check the loop condition. This means extra cycles will be taken.

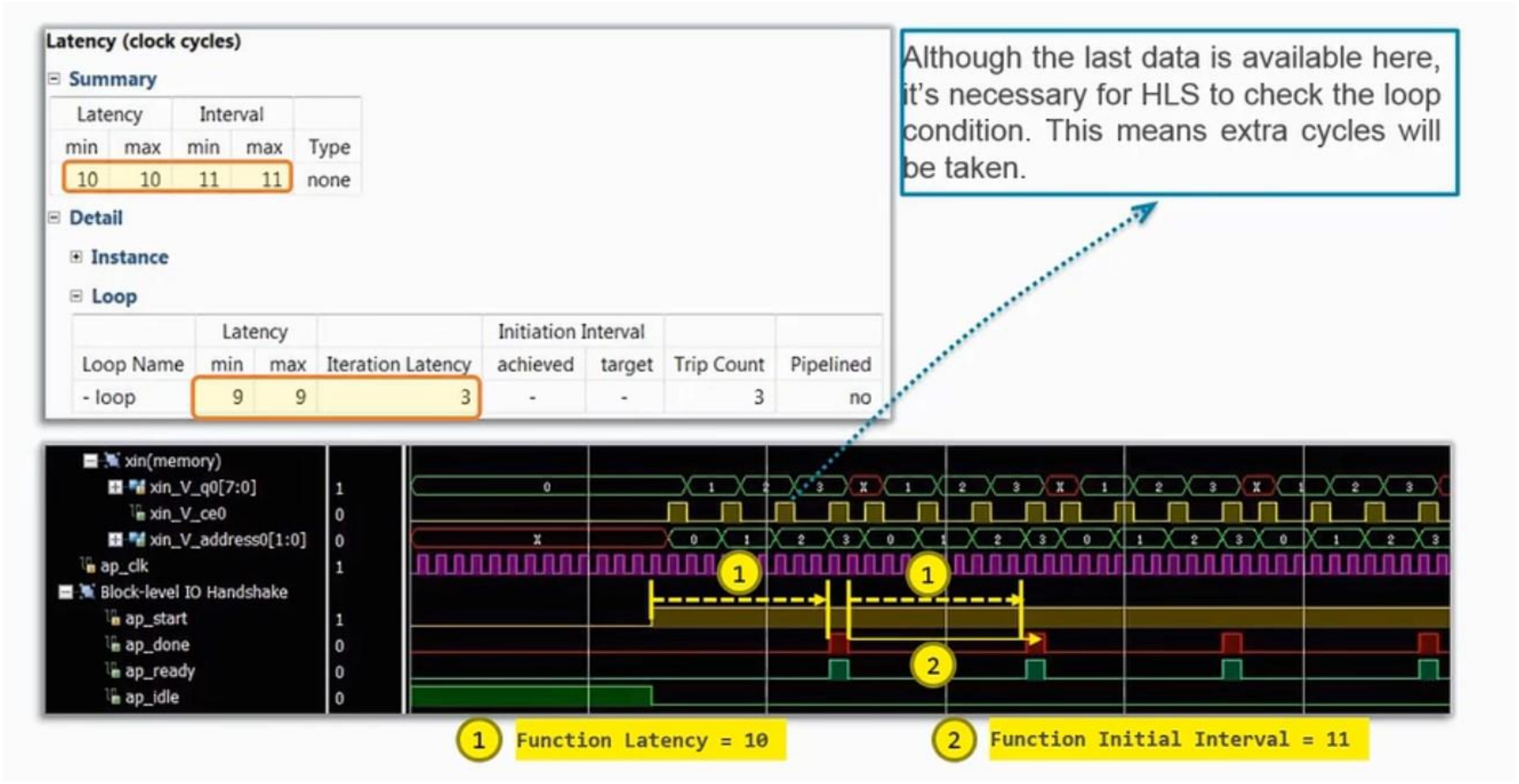


1 Function Latency = 10

2 Function Initial Interval = 11

© Copyright 2023 Advanced Micro Devices, Inc.

Terminology for Measuring in Clock Cycles



Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
10	10	11	11	none

Detail

Instance

Loop

	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop	9	9	3	-	-	3	no

Although the last data is available here, it's necessary for HLS to check the loop condition. This means extra cycles will be taken.

1 Function Latency = 10

2 Function Initial Interval = 11

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Unsupported Constructs: Overview

Some constructs are not synthesizable or can result in errors further down the design flow

- System Calls
- Dynamic Memory Usage
- Data Types
- Pointer Limitations

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Unsupported Constructs: Overview

System Calls

- System calls cannot be synthesized
  - Performing some tasks on the OS in which the C program is running
- HLS tool ignores these commonly used system calls
- Vitis™ HLS tool defines the macro __SYNTHESIS__, which allows excluding non-synthesizable code from the design

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Unsupported Constructs: Overview

Dynamic Memory Usage

- All the constructs of C are supported in HLS, provided they are statically defined at compile time
- If a function is not fully realized, it cannot be synthesized
  - Example: malloc (), alloc (), free ()
- HLS tool does not support C++ objects that are dynamically created or destroyed
- Polymorphism and virtual function calls are not supported

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Unsupported Constructs: Overview

Data Types

Forward declared types and recursive types are not supported for synthesis

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Unsupported Constructs: Overview

Pointer Limitations

- No support for general pointer casting but supports pointer casting between native C/C++ types
- Supports pointer arrays for synthesis, provided that each pointer points to a scalar or an array of scalars—arrays of pointers cannot point to additional pointers
- Function pointers are not supported

© Copyright 2023 Advanced Micro Devices, Inc.
AMD
together we advance_

Design Exploration with Directives

AMD
together we advance_

Design Exploration with Pragmas

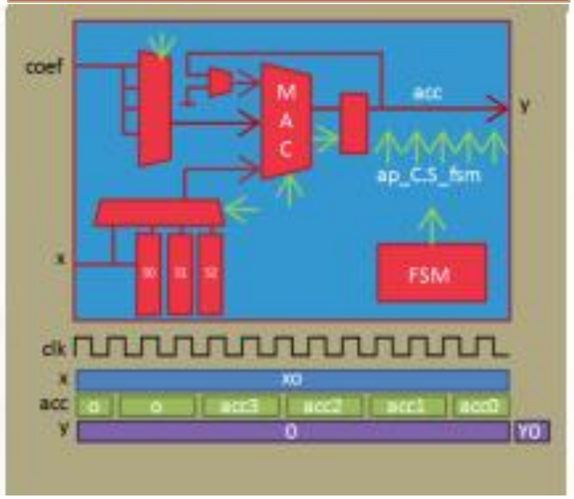
Multiplication and accumulation happen inside the “for” loop

Example

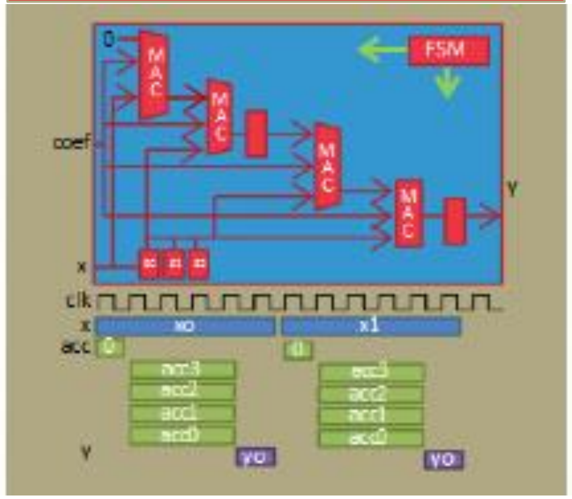
```
loop:  for  (i=3;i>=0;i--) {
  if (i==0) {
    acc+=X*C[0];
    shift_reg[0]=X;
  } else {
     shift_reg[i]=shift_reg[i-1];
     acc+=shift_reg[i]*C[i];
  } }
```

Loop can result in three types of hardware, depending upon the pragmas used

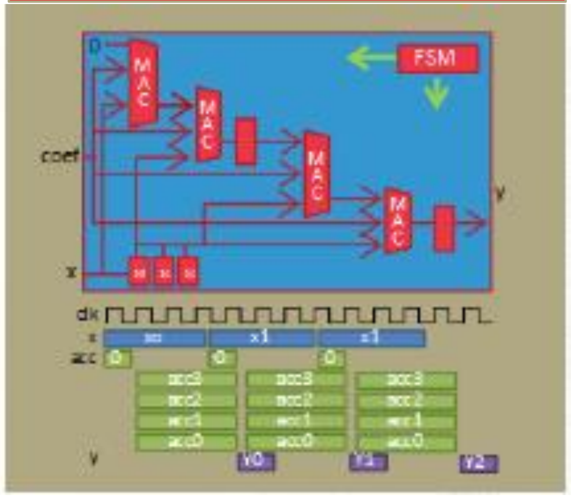
Default Design



Unroll Pragma



Pipeline Pragma



© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

HLS Directives

Loops: Unrolling

Loops: Pipelining

Loops or Function Dataflow

© Copyright 2023 Advanced Micro Devices, Inc.

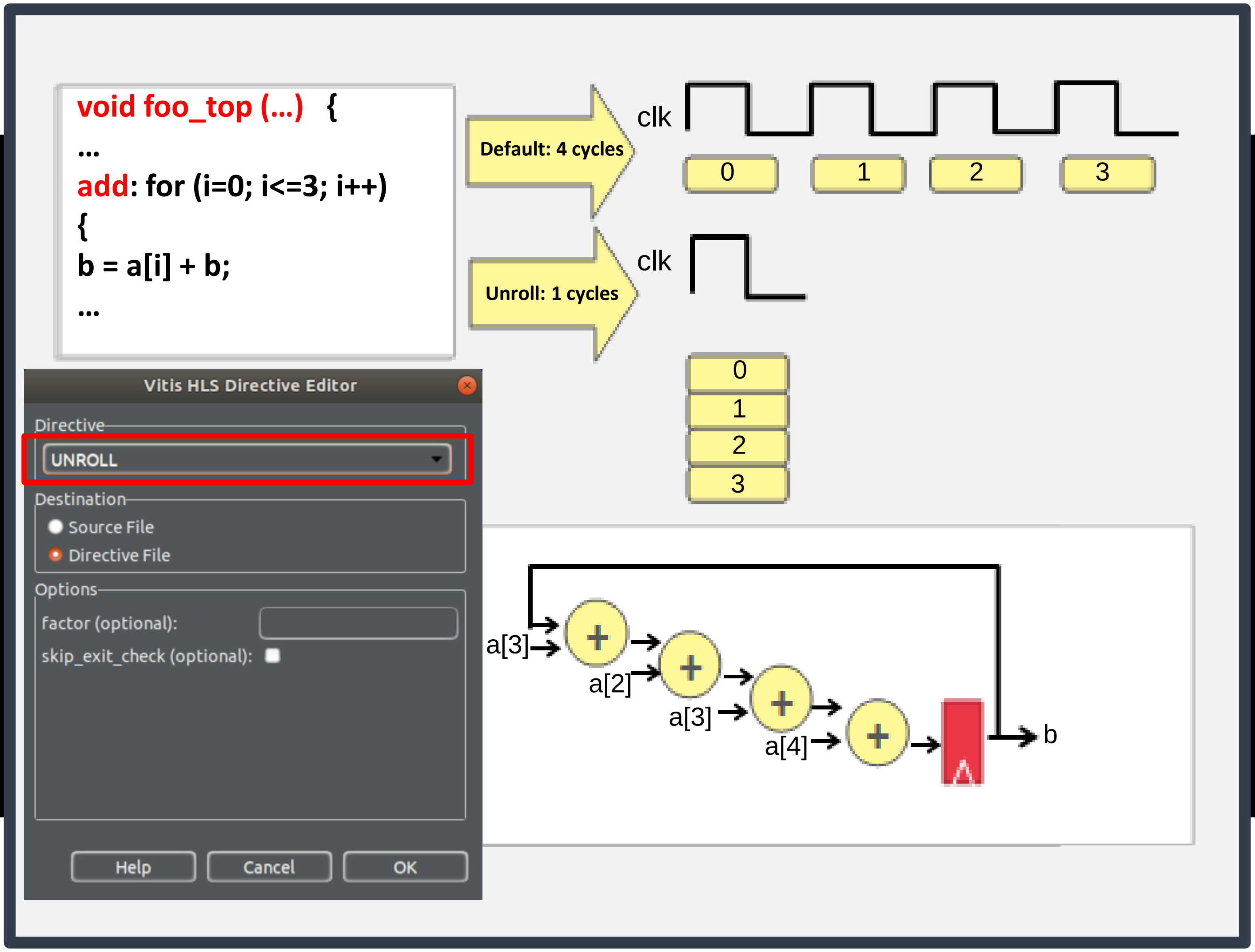
AMD
together we advance_

Loops: Unrolling

These independent operations can then be performed in one clock cycle

- Instead of four cycles, in case of default execution

Fully unrolled loop requires more resources, more area but gives better throughput



```
void foo_top (…)  {
…
add: for (i=0; i<=3; i++)
{
b = a[i] + b;
…
```

Vitis HLS Directive Editor

Directive: UNROLL

Destination:
- Source File
- [✓ Directive File]

Options:
- factor (optional):
- skip_exit_check (optional): ☐

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

PIPELINING

PIPELINE directive allows parallel execution of the operations

Loop Pipelining

Function Pipelining

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

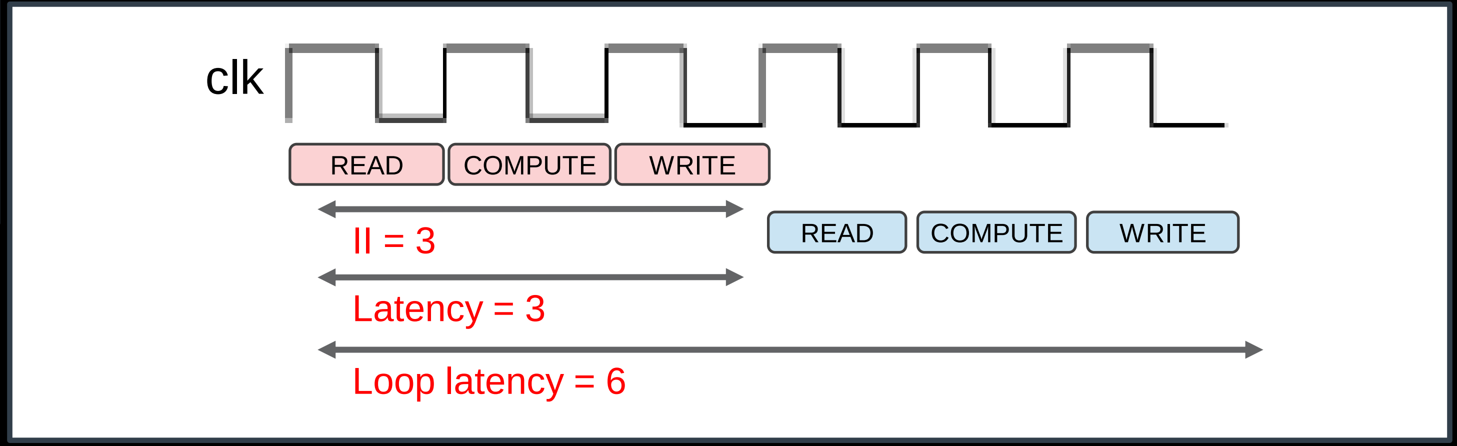
Loop Pipelining

During pipelining, the initiation interval defaults to 1 if not specified but can be explicitly specified as well

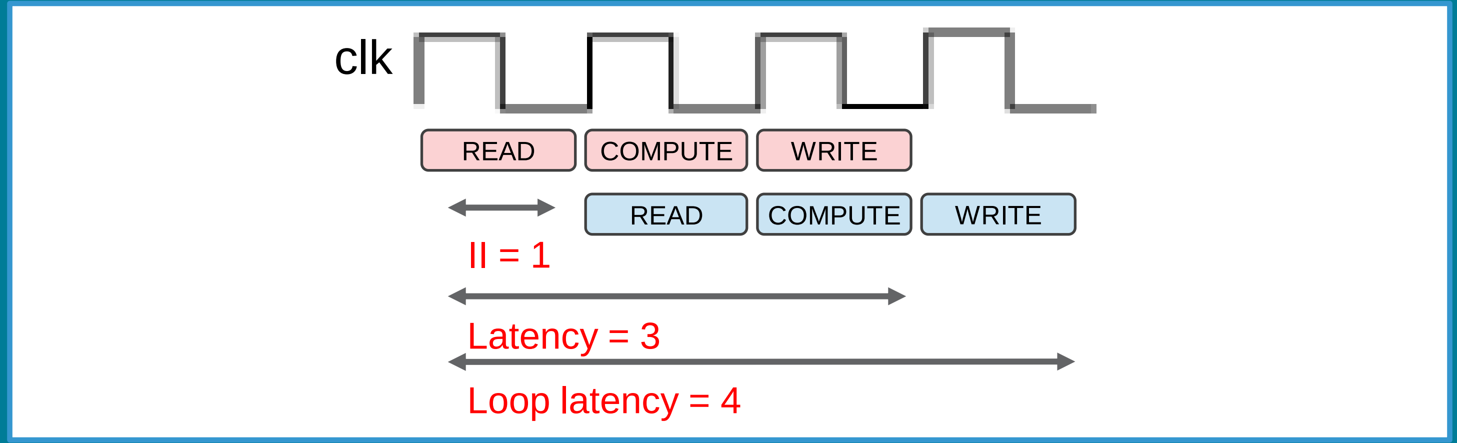
Function reads an input and outputs a value every three clock cycles

```
void foo(...)  {
...
add: for (i=1; i=<2; i++)
{
op_READ;
op_COMPUTE;
op_WRITE;
}
...
```

Without pipeline



With pipeline



Compute operation from first iteration and the reading operation from the second iteration happens parallelly

© Copyright 2023 Advanced Micro Devices, Inc.

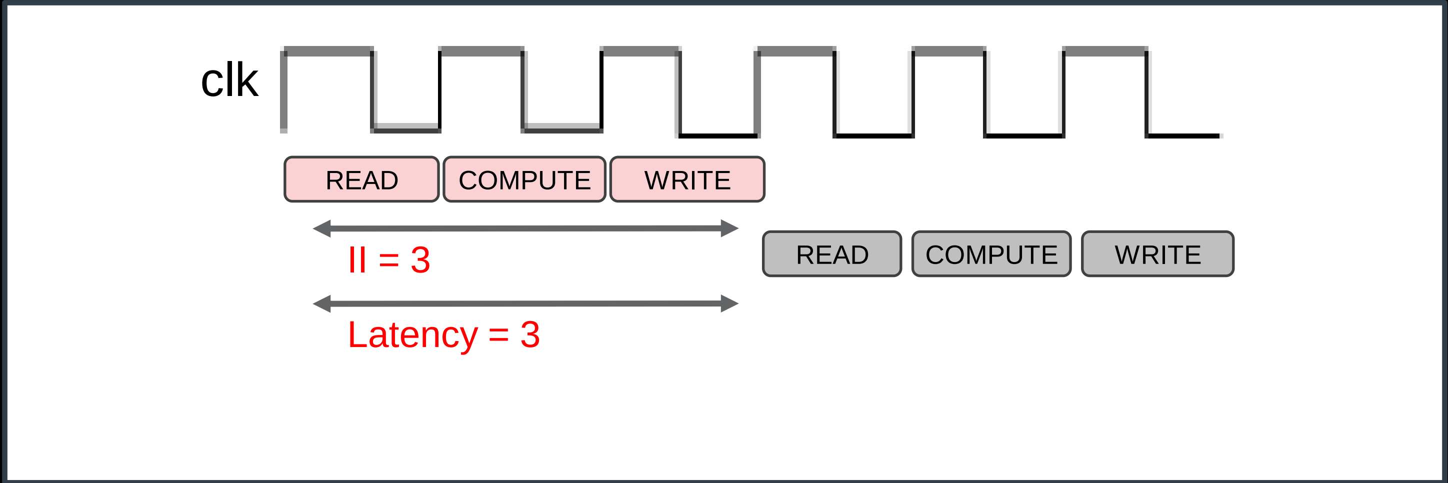
AMD
together we advance_

Function Pipelining

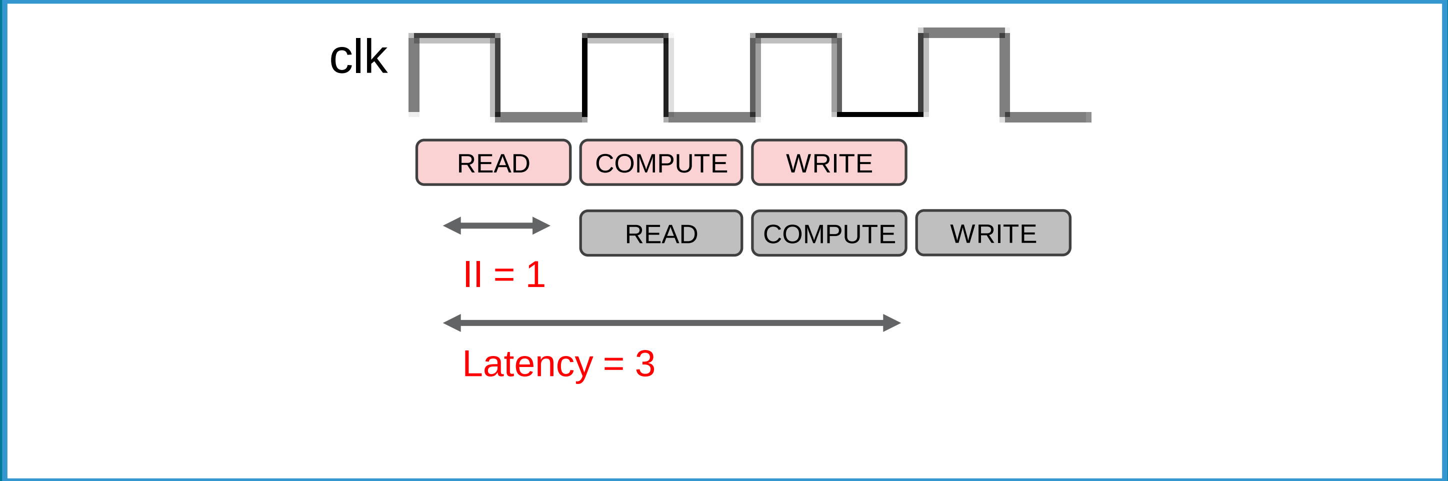
All the operations happen in a default sequential manner

```
void foo(…)  {
op_READ;
op_COMPUTE;
op_WRITE;
}
```

Without pipeline



With pipeline



Operations happen in parallel

© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

Pipelining: Be Careful Where You Put It!

In general, **pipelining the inner-most loop will result in best performance for area**

- If the inner most loop is modest and fixed, try the next one (or two) out
    - Outer loops will keep the inner pipeline fed

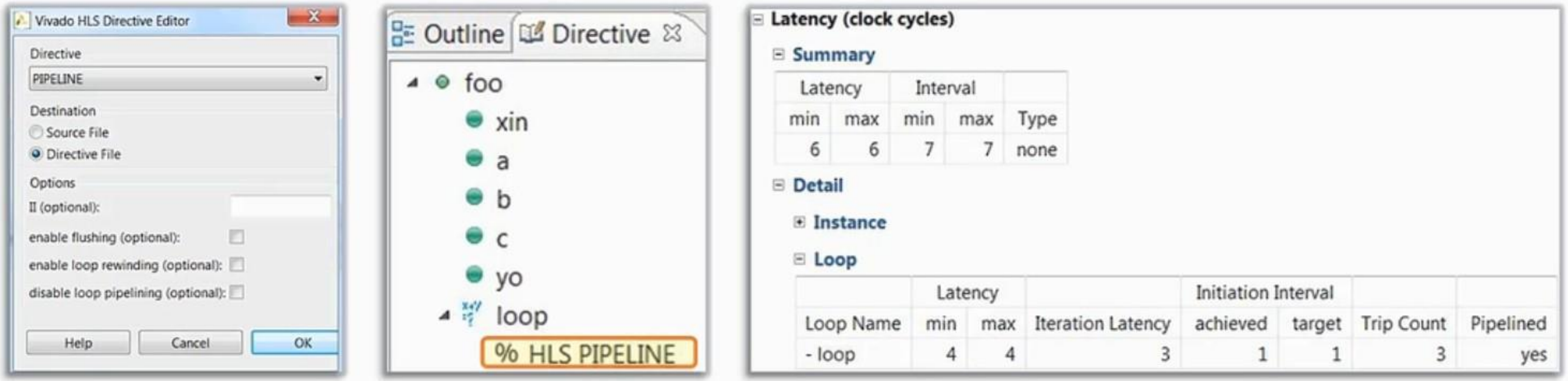
```
void foo(in1[][], in2[][], ...) {
...
  L1:for(i=1;i<N;i++) {
    L2:for(j=0;j<M;j++) {
#pragma AP PIPELINE
      out[i][j] = in1[i][j] + in2[i][j];
    }
  }
}
```

1 adders, 3 accesses

© Copyright 2023 Advanced Micro Devices, Inc.

together we advance_

Pipelining: Be Careful Where You Put It!



Vivado HLS Directive Editor

Directive: PIPELINE

Destination: Source File [✓ Directive File]

Options

II (optional):

enable flushing (optional): ☐

enable loop rewinding (optional): ☐

disable loop pipelining (optional): ☐

Help | Cancel | OK

Outline | Directive

- foo
  - xin
  - a
  - b
  - c
  - yo
  - loop
    - % HLS PIPELINE

Latency (clock cycles)

Summary

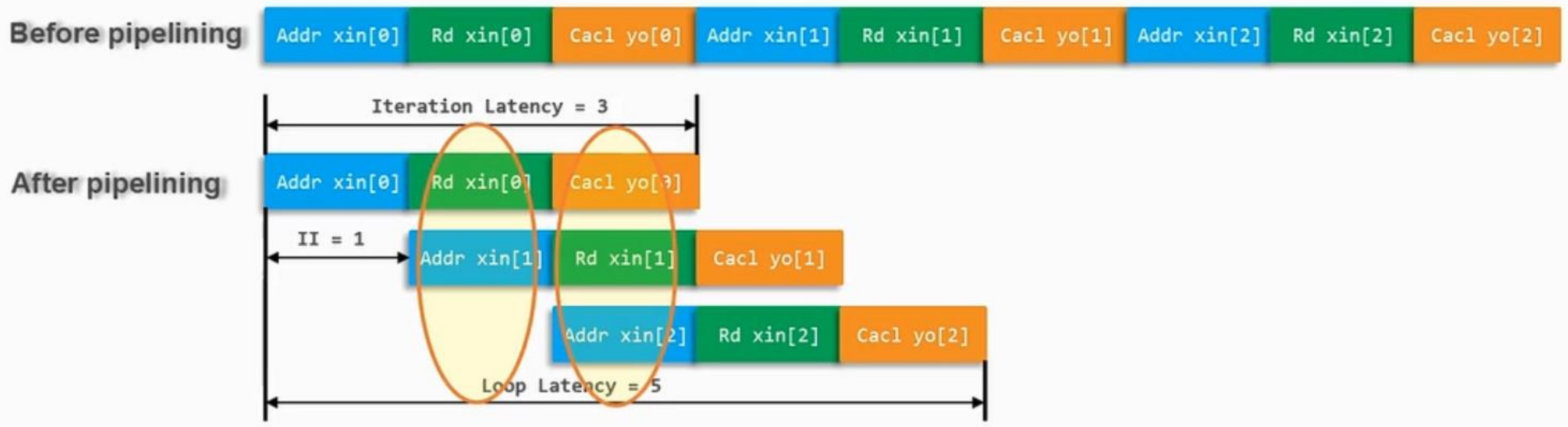
Latency		Interval		
min	max	min	max	Type
6	6	7	7	none

Detail

Instance

Loop

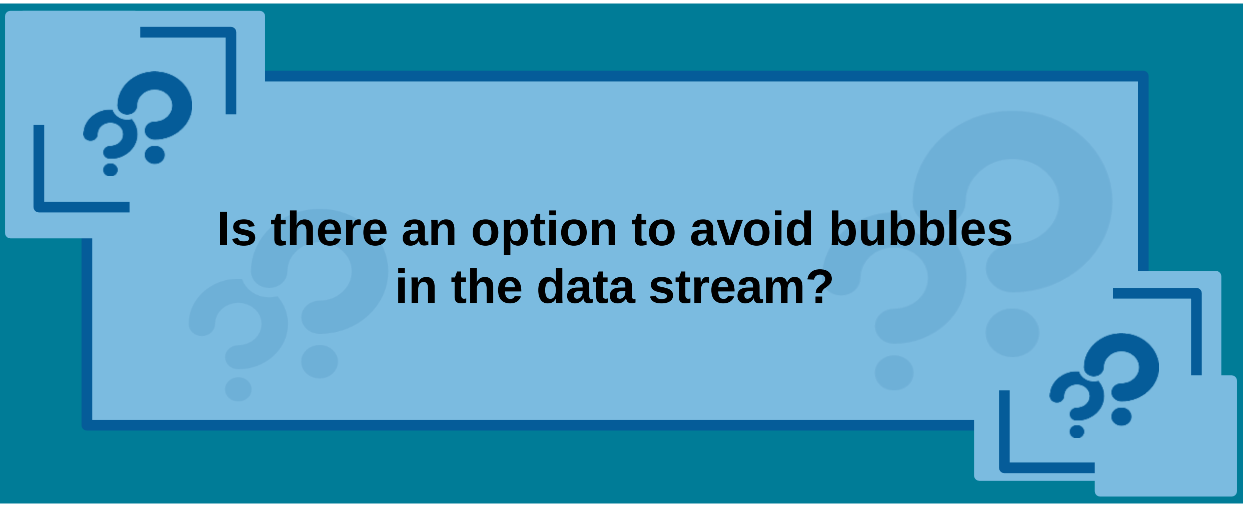
	Latency			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- loop	4	4	3	1	1	3	yes



© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

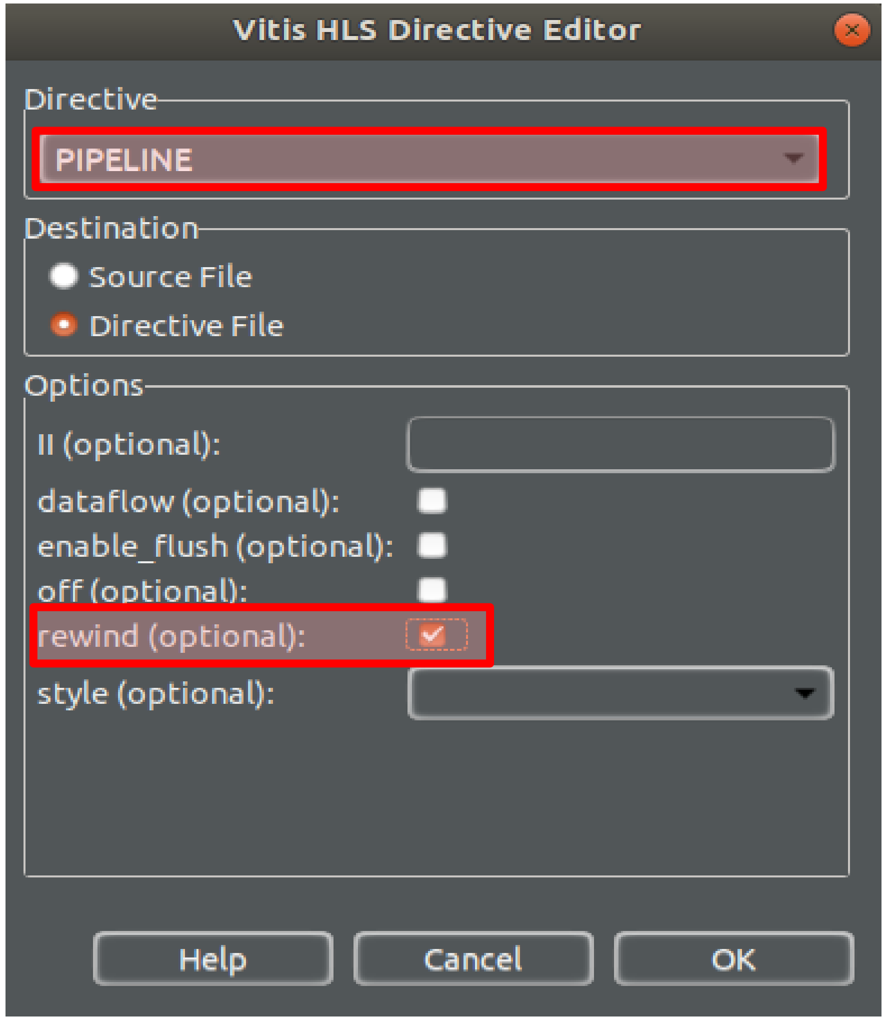
Continuous Pipelining of the Top-Level Loop



Is there an option to avoid bubbles in the data stream?

“Rewind”

Helps continuously execute the pipeline



Vitis HLS Directive Editor

Directive: PIPELINE

Destination:
- Source File
- [✓ Directive File]

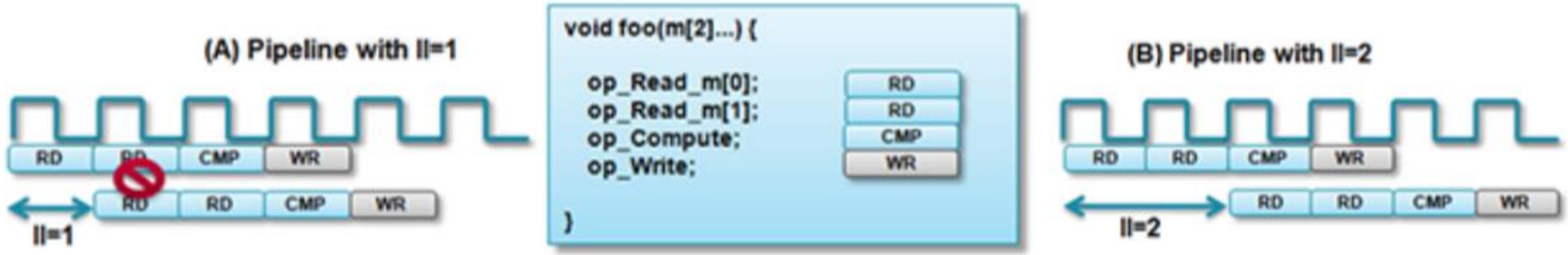
Options:
- II (optional):
- dataflow (optional): ☐
- enable_flush (optional): ☐
- off (optional): ☐
- rewind (optional): ✓
- style (optional):

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Resource Contention: Unfeasible Initiation Intervals

Vitis™ HLS tool always tries to improve the initiation interval (II), but sometimes this specification cannot be met



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

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

}
```

Two read operations happening on the same port

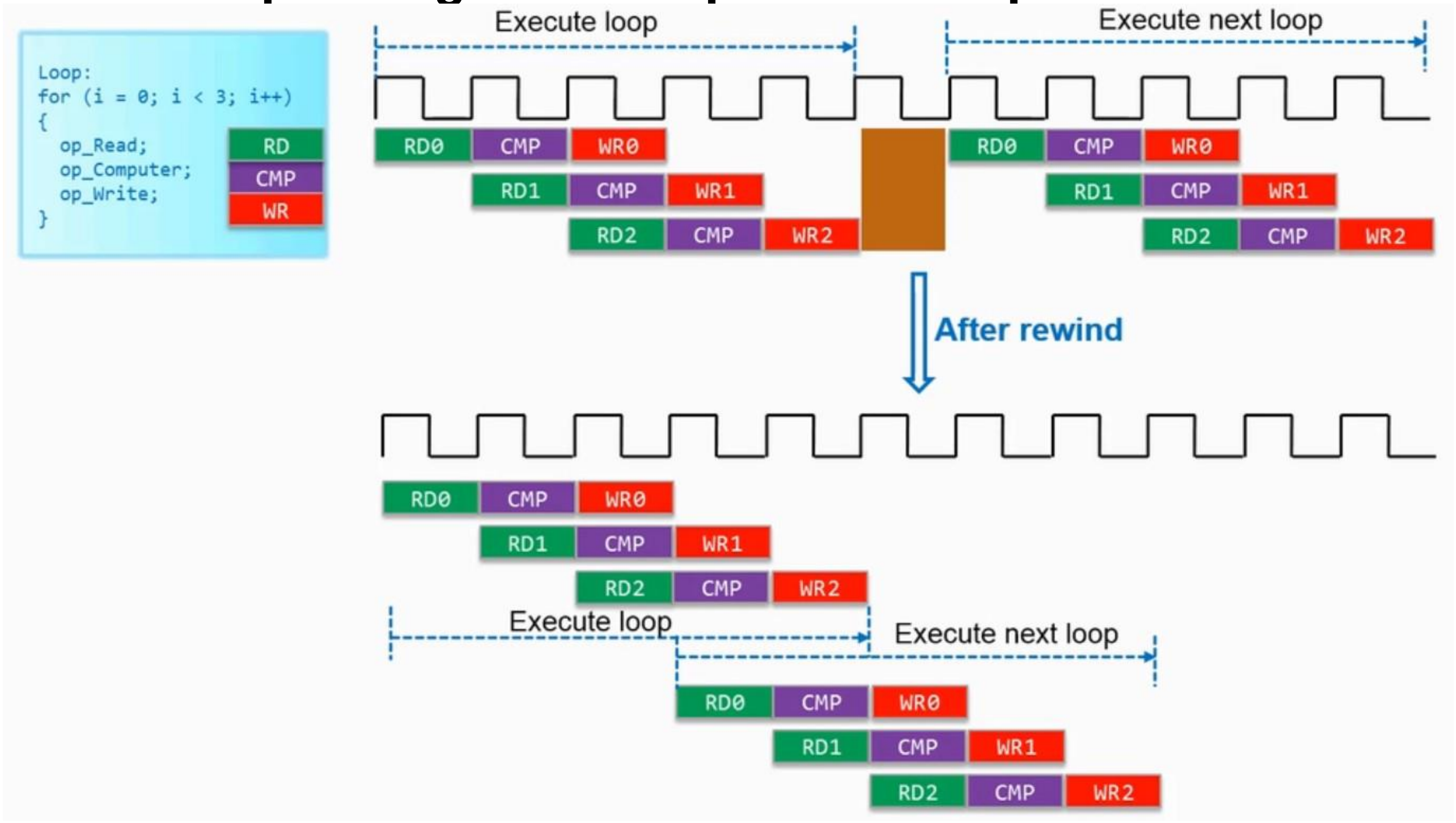
HLS tool cannot make II = 1 as

- Same port cannot be read at the same time
- Similar effect with other resource limitations

HLS tool will automatically increase the II to 2 to create a design even if constraints are violated

© Copyright 2023 Advanced Micro Devices, Inc.

Continuous Pipelining of the Top-Level Loop



```
Loop:
for (i = 0; i < 3; i++)
{
  op_Read;
  op_Computer;
  op_Write;
}
```

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Reduce Latency

Directives and Configurations	Description
LATENCY	Allows a minimum and maximum latency constraint to be specified
LOOP_FLATTEN	Allows nested loops to be collapsed into a single loop with improved latency
LOOP_MERGE	Merge consecutive loops to reduce overall latency, increase sharing, and improve logic optimization

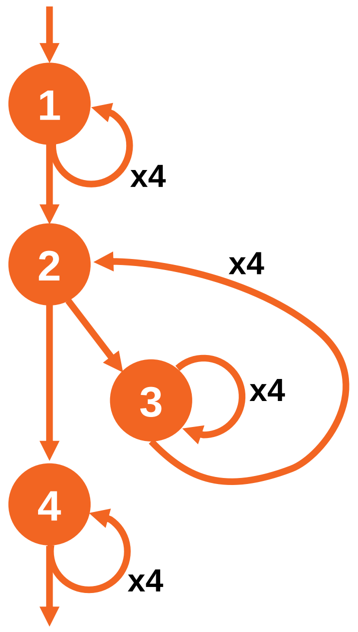
Benefit to the latency because it typically costs a clock cycle to enter and leave a loop

The fewer the transitions between loops, the smaller the number of clock cycles

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

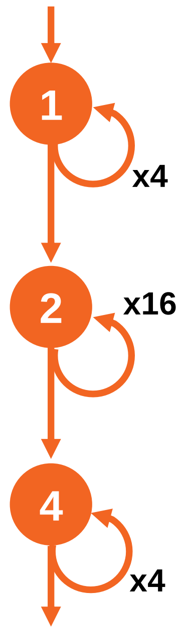
Loop Flattening



36 transitions

```
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 L4]
  }
```



28 transitions

© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

Perfect, Semi-Perfect, and Imperfect Loops

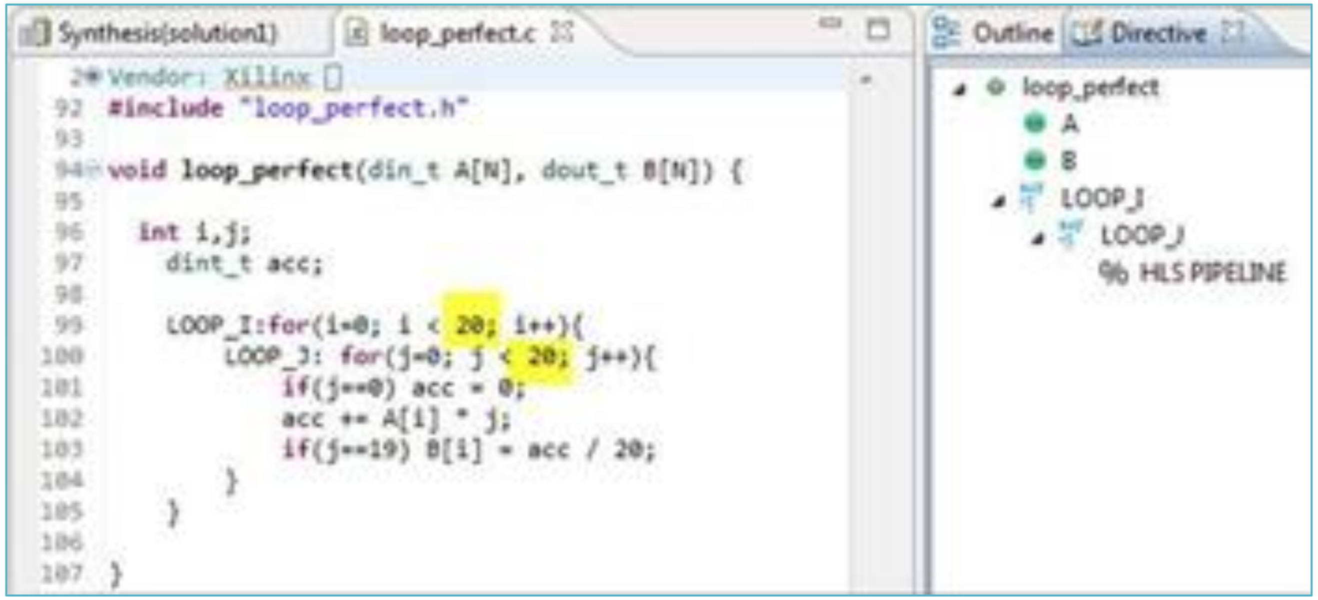
Perfect Loops | **Semi-Perfect Loops** | **Imperfect Loops**

Only the innermost loop has the loop body content

No logic specified between the loop statements

All the loop bounds are constant

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



```
/* Vendor: Xilinx
#include "loop_perfect.h"

void loop_perfect(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++){
            if(j==0) acc = 0;
            acc += A[i] * j;
            if(j==19) B[i] = acc / 20;
        }
    }

}
```

© Copyright 2023 Advanced Micro Devices, Inc.
AMD
together we advance_

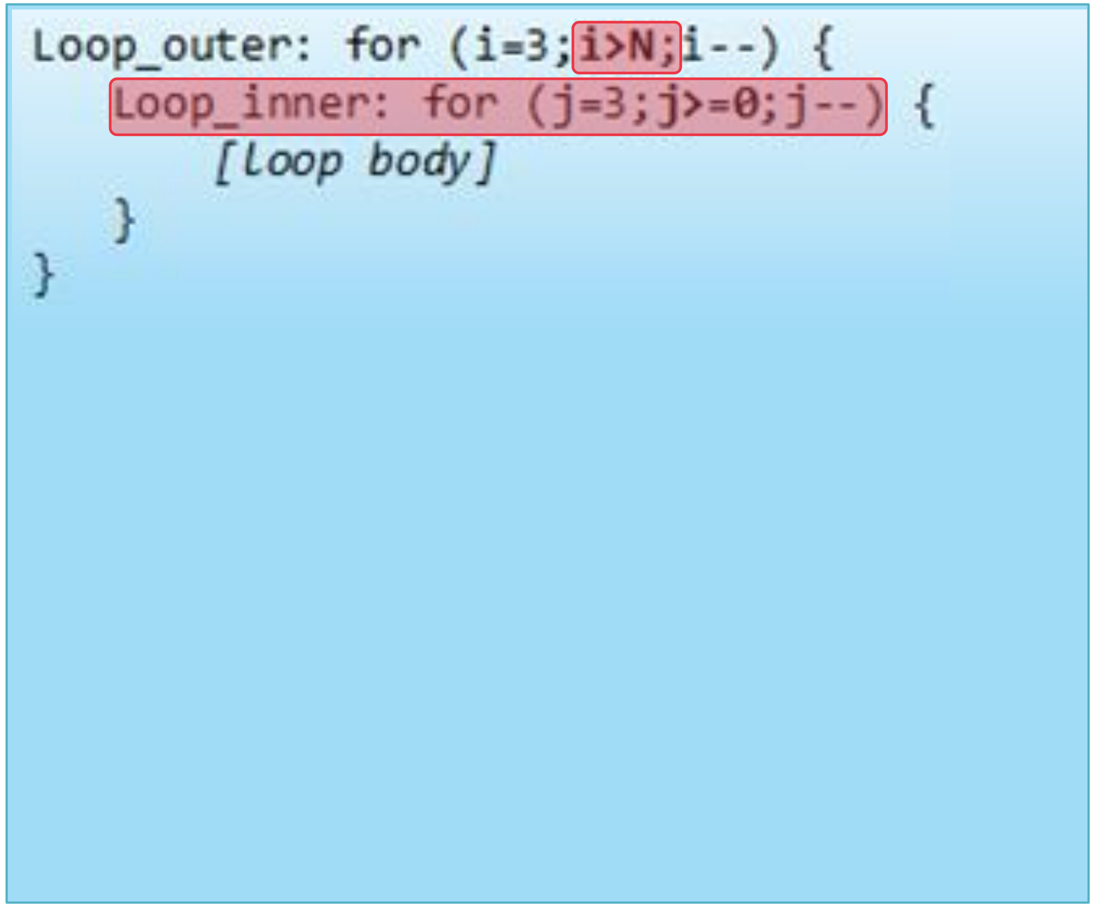
Perfect, Semi-Perfect, and Imperfect Loops

Perfect Loops	Semi-Perfect Loops	Imperfect Loops

Only the innermost loop has the loop body content

No logic specified between the loop statements

Outermost loop bound can be a variable



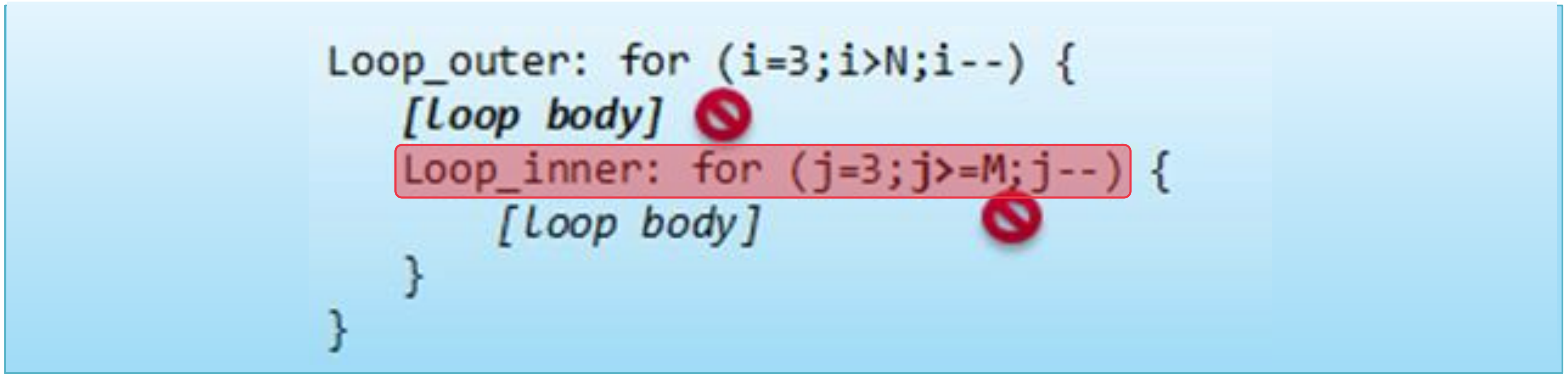
```
Loop_outer: for (i=3;i>N;i--) {
    Loop_inner: for (j=3;j>=0;j--) {
        [loop body]
    }
}
```

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Perfect, Semi-Perfect, and Imperfect Loops

Perfect Loops | **Semi-Perfect Loops** | **Imperfect Loops**



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

Inner loop has variables bounds or the loop body is not exclusively inside the inner loop

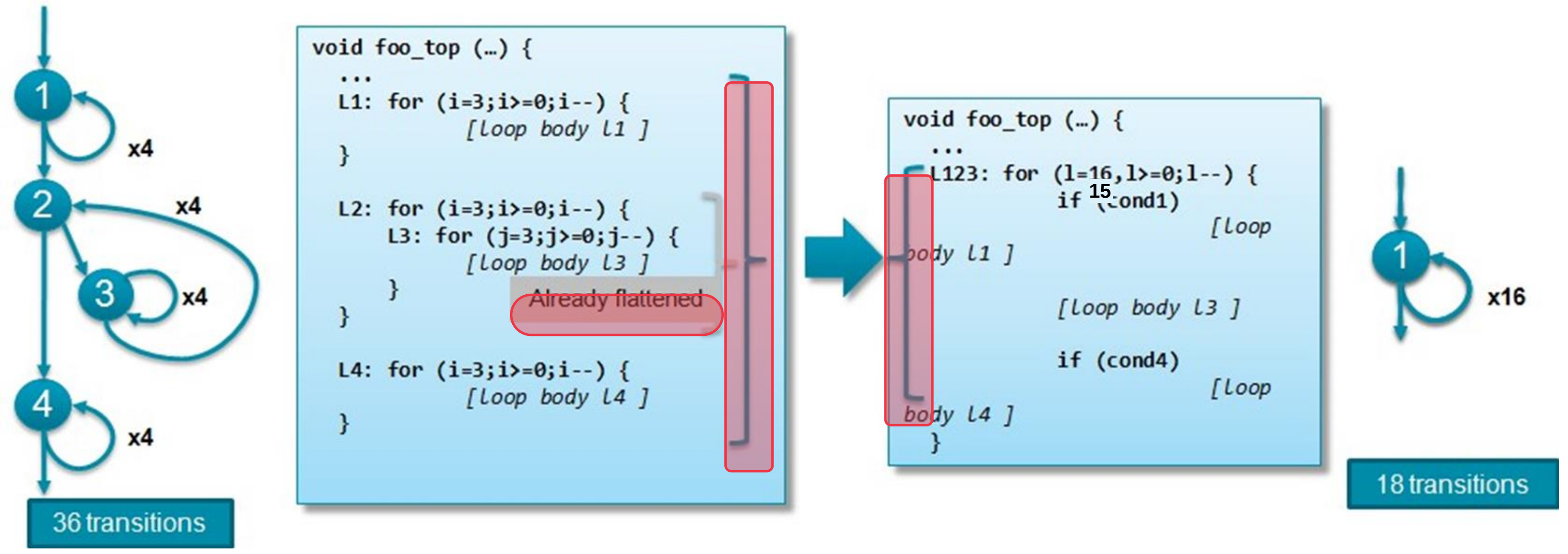
- Designers should try to restructure the code or unroll the loops to create a perfect loop nest

Trivial transformation from imperfect to perfect loop is made automatically

© Copyright 2023 Advanced Micro Devices, Inc.
AMD together we advance_

Loop Merging

- Merging the loops allow the logic within the loops to be optimized together
- Allows for more efficient architecture explorations



```
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 (…) {
  ...
  L123: for (l=16,l>=0;l--) {
          if (cond1)
                  [Loop
body L1 ]

          [Loop body L3 ]

          if (cond4)
                  [Loop
body L4 ]
  }
```

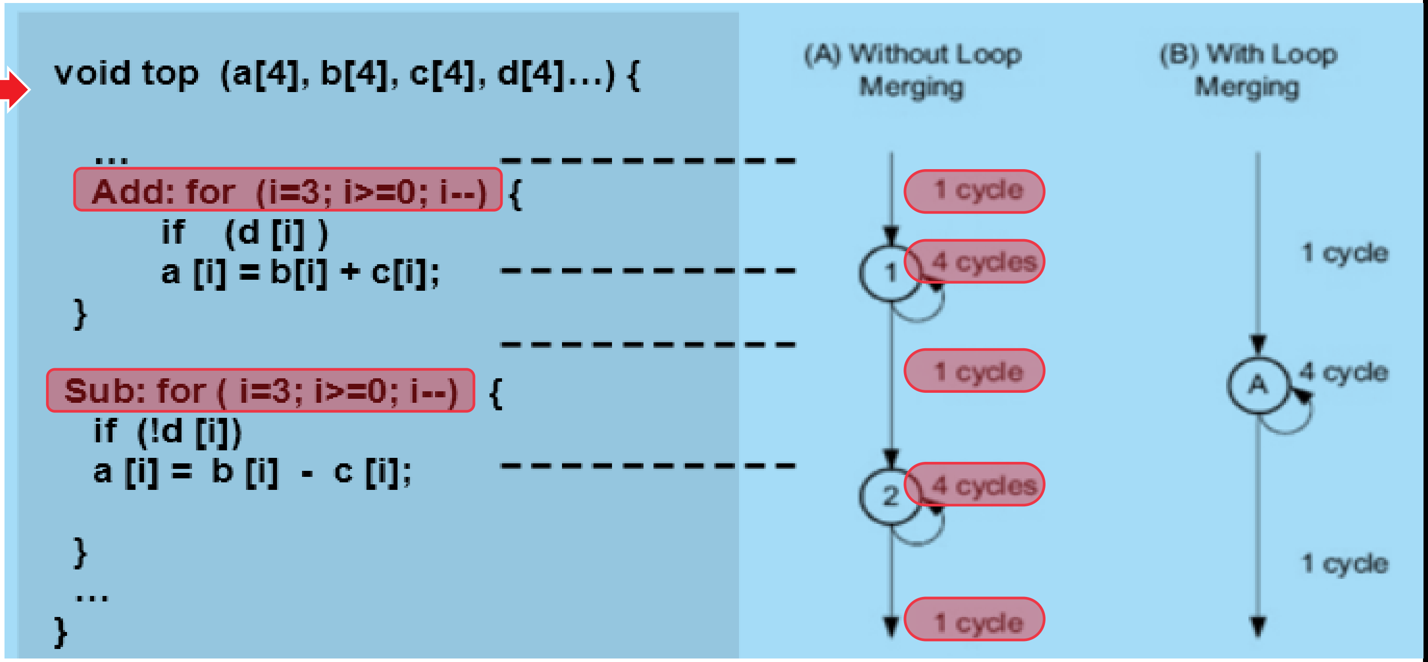
© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Loop Merging

Without merging the loops:

With merging of the loops:



```
void top  (a[4], b[4], c[4], d[4]...) {

   ...
   Add: for  (i=3; i>=0; i--) {
         if   (d [i] )
         a [i] = b[i] + c[i];
   }

 Sub: for ( i=3; i>=0; i--)  {
    if  (!d [i])
    a [i] =  b [i]  -  c [i];

   }
   ...
}
```

11 cycles

6 cycles

Loop Merging Reduces Latency

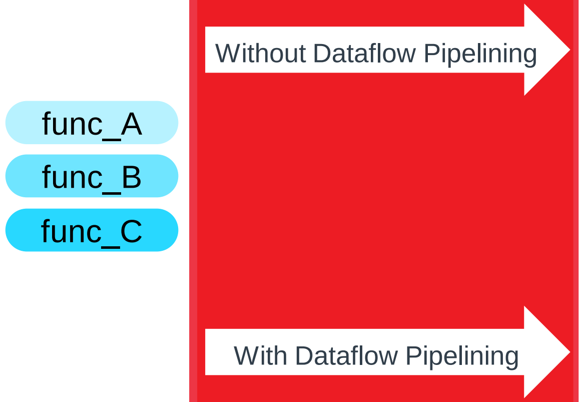
© Copyright 2023 Advanced Micro Devices, Inc.

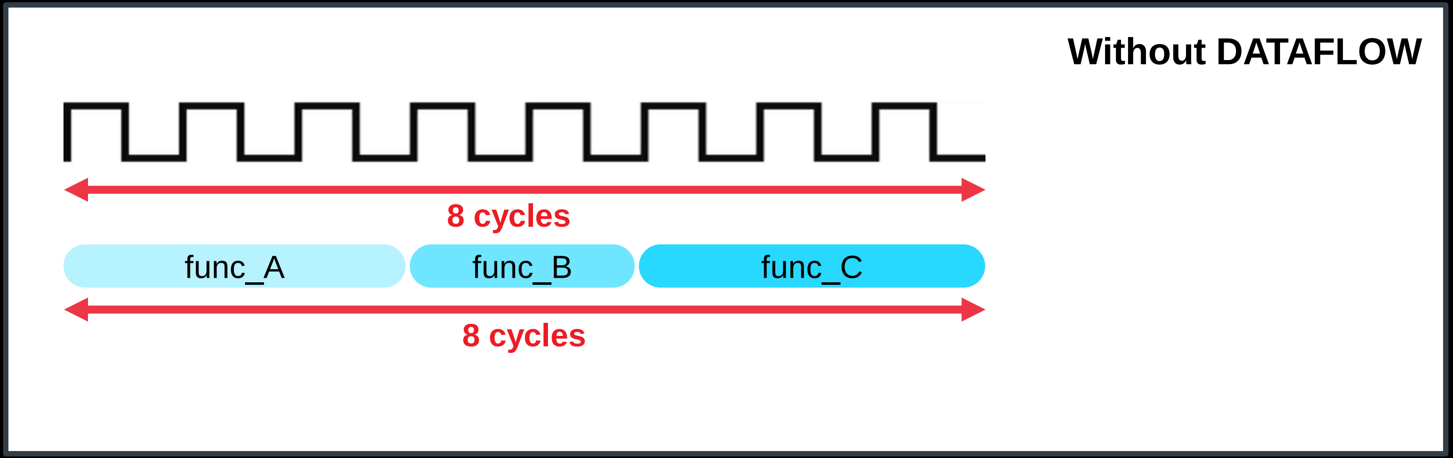
AMD together we advance_

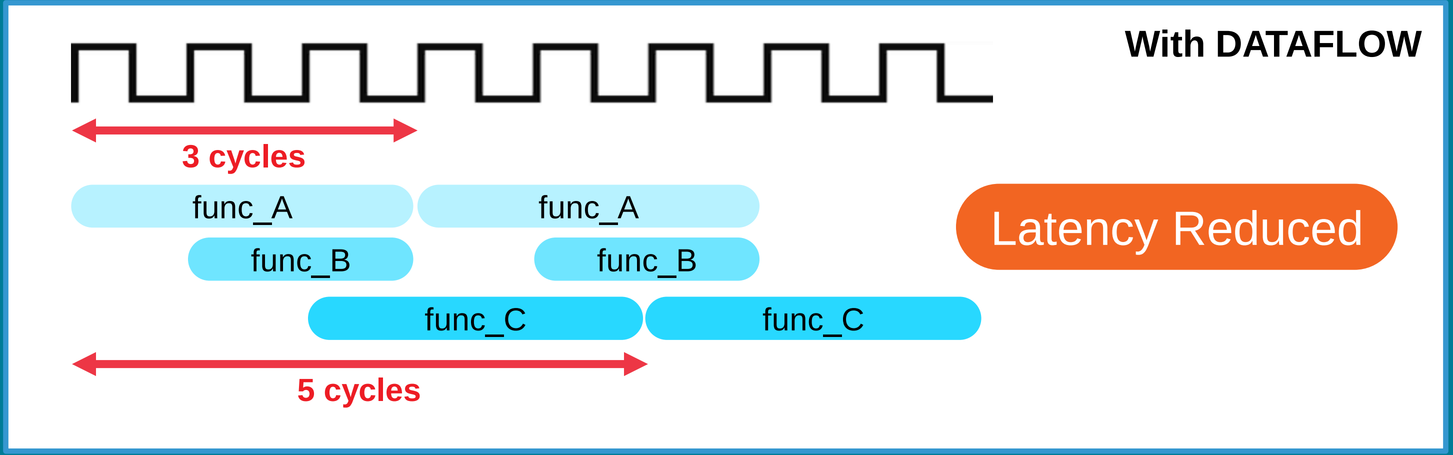
Dataflow

Implementation requires eight cycles before a new input can be processed by func_A and eight cycles before an output is written by func_C

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

   return d;
}
```







func_A can begin processing a new input every three clock cycles (giving lower initiation interval); requires five clocks to output a final value

© Copyright 2023 Advanced Micro Devices, Inc.

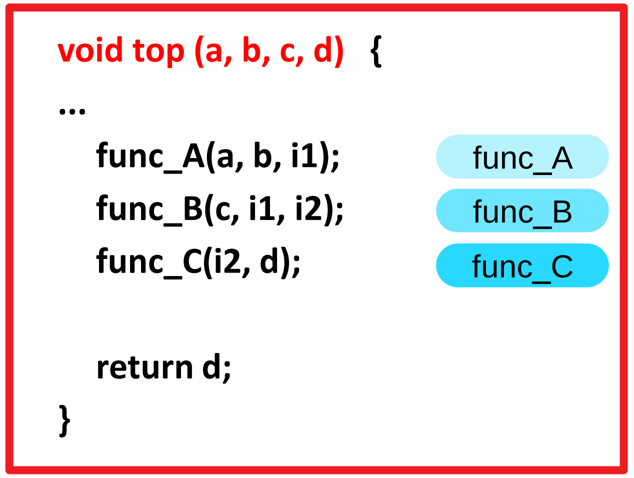
AMD together we advance_

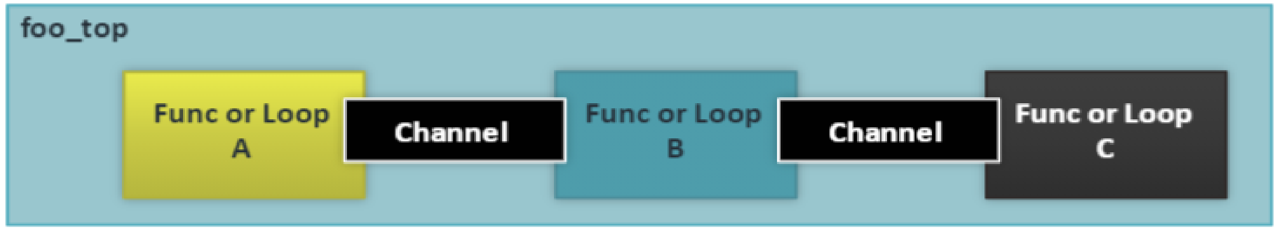
Configuring the Dataflow Channel

Vitis™ HLS tool analyzes the function or a loop body

Creates individual channels that model the dataflow to store the results of each task in the dataflow region

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

   return d;
}
```



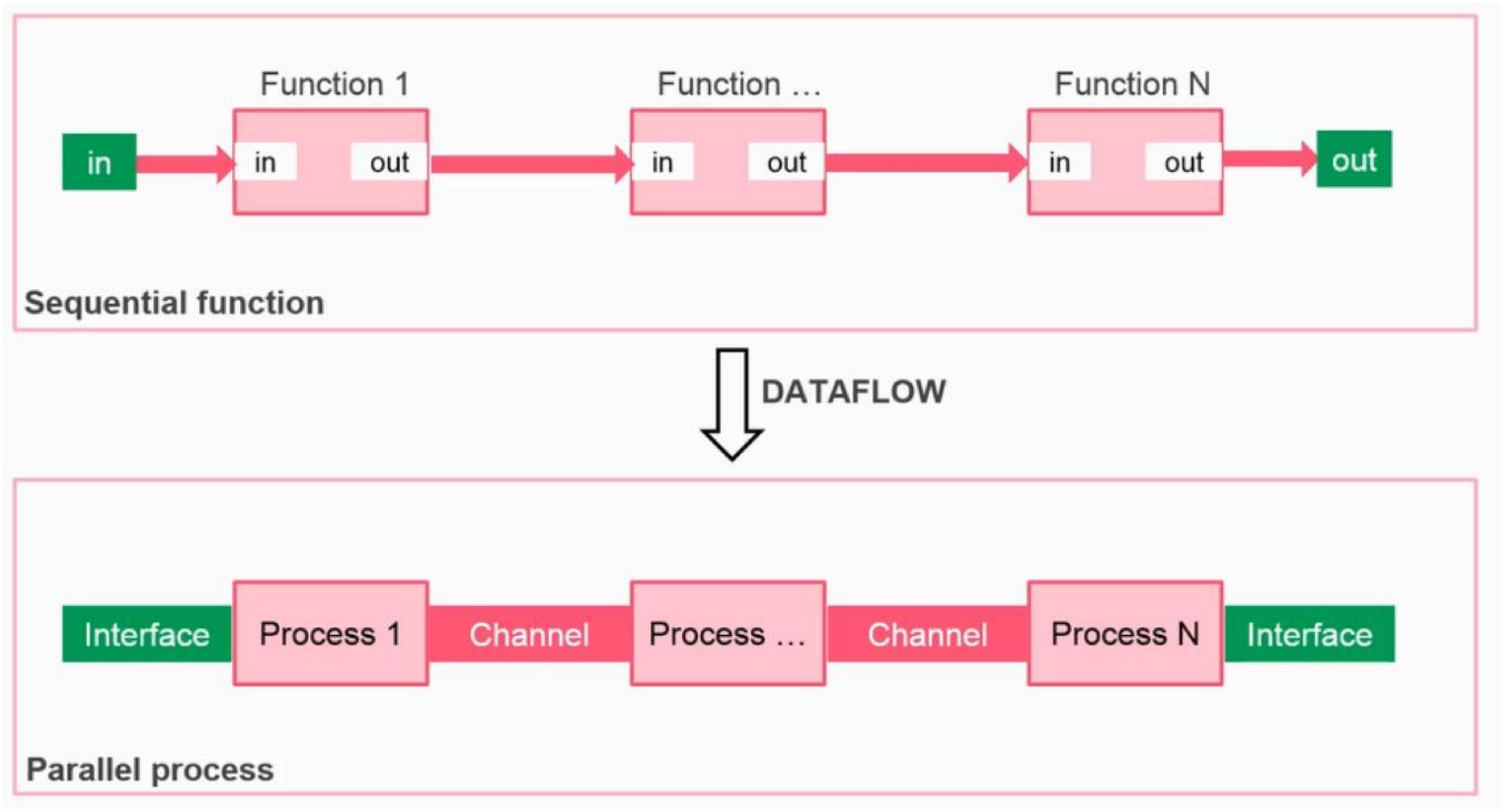


Places these channels between the blocks to maintain the data rate

© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

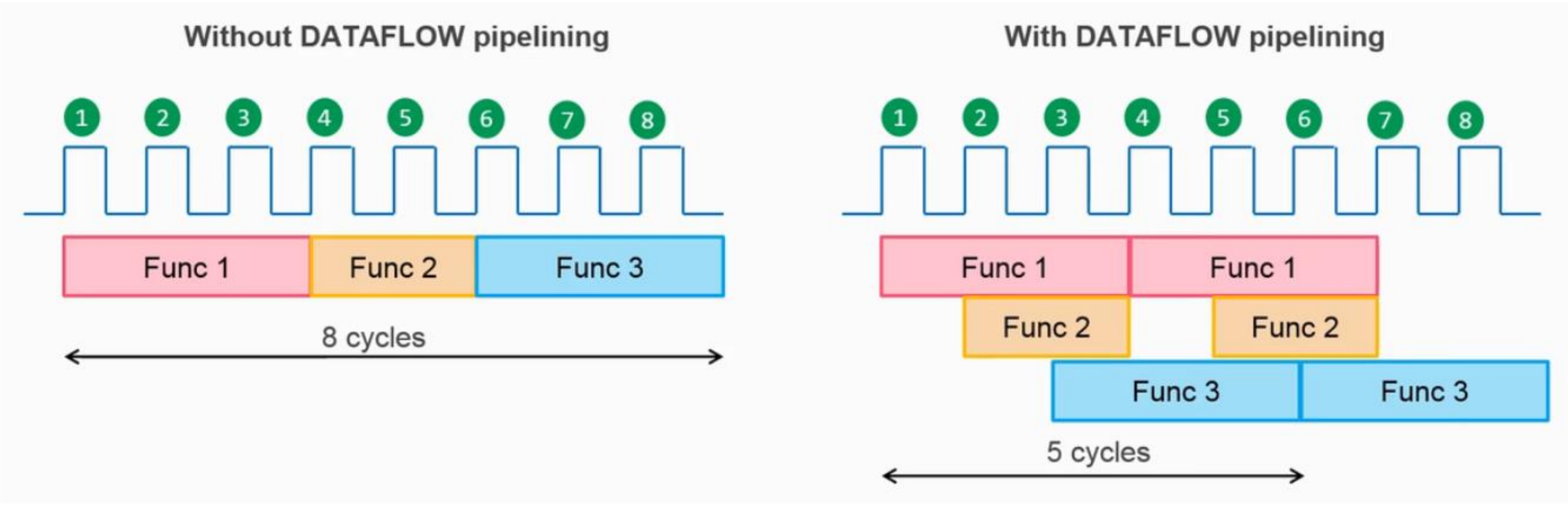
Configuring the Dataflow Channel



© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

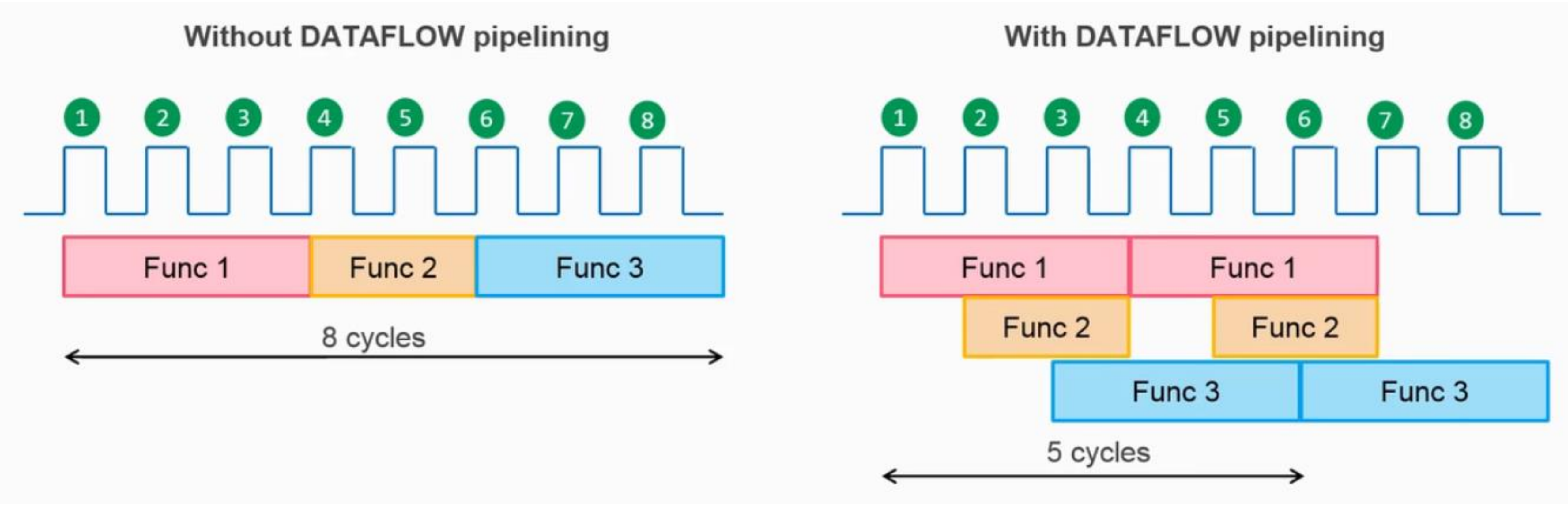
Dataflow: Ideal for Streaming Arrays and Multi-Rate Functions



© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Dataflow: Ideal for Streaming Arrays and Multi-Rate Functions



© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Interface Types

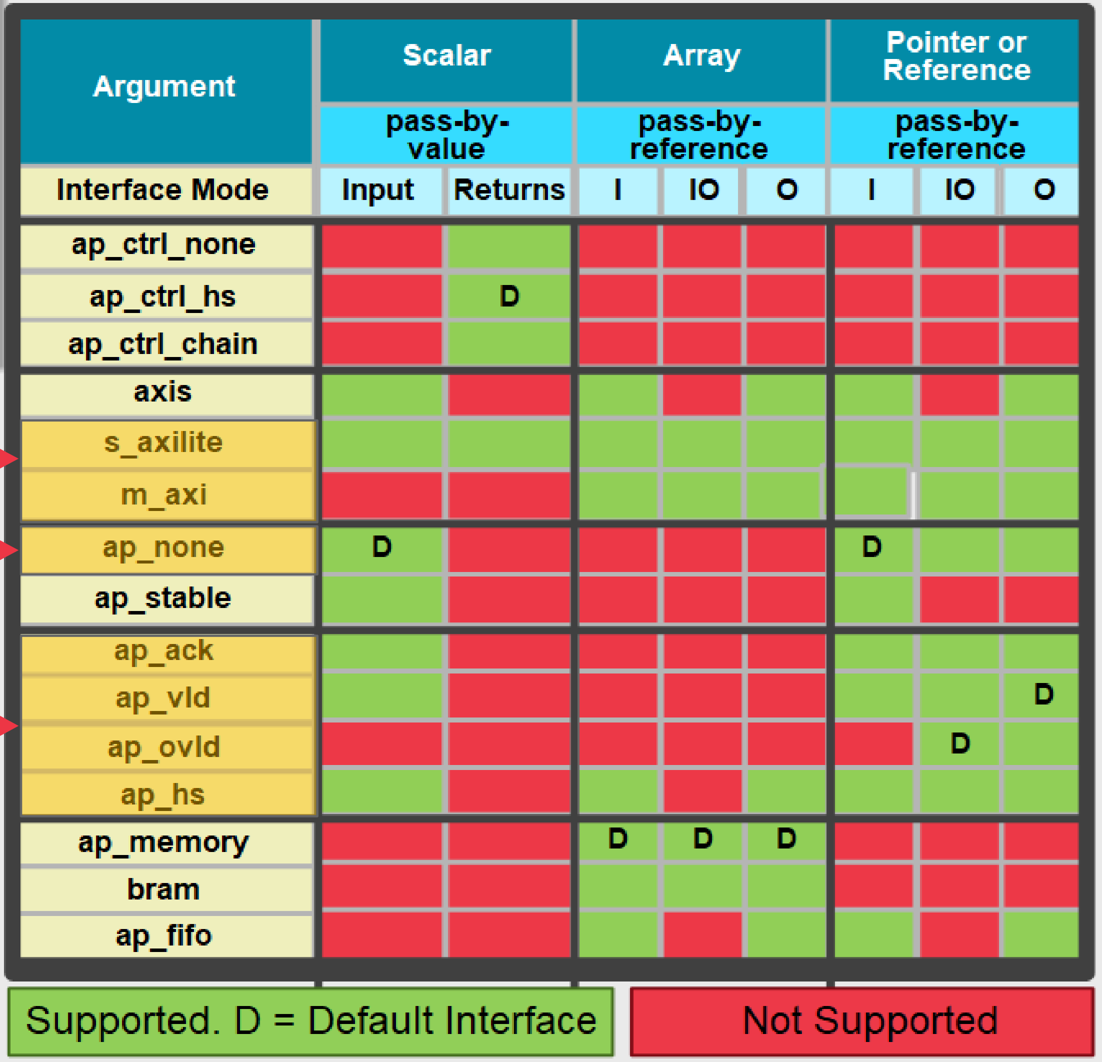
```
# include "adders.h"
int adders(int in1, int in2, int *sum) {
     int temp;

    *sum = in1 + in2 + *sum;
     temp = in1 + in2;

     return temp;
}
```

Default interface type for the sum port will be type ap_ovld

AXI Interface Protocol

No I/O Protocol

Wire Handshake Protocol



Argument	Scalar		Array			Pointer or Reference		
	pass-by-value		pass-by-reference			pass-by-reference		
Interface Mode	Input	Returns	I	IO	O	I	IO	O
ap_ctrl_none	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
ap_ctrl_hs	🟥	🟩 D	🟥	🟥	🟥	🟥	🟥	🟥
ap_ctrl_chain	🟥	🟩	🟥	🟥	🟥	🟥	🟥	🟥
axis	🟩	🟥	🟩	🟥	🟩	🟩	🟥	🟩
s_axilite	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
m_axi	🟥	🟥	🟩	🟩	🟩	🟩	🟩	🟩
ap_none	🟩 D	🟥	🟥	🟥	🟥	🟩 D	🟩	🟩
ap_stable	🟩	🟥	🟥	🟥	🟥	🟩	🟥	🟥
ap_ack	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩
ap_vld	🟩	🟥	🟥	🟥	🟥	🟩	🟩	🟩 D
ap_ovld	🟥	🟥	🟥	🟥	🟥	🟥	🟩 D	🟩
ap_hs	🟩	🟥	🟩	🟥	🟩	🟩	🟩	🟩
ap_memory	🟥	🟥	🟩 D	🟩 D	🟩 D	🟥	🟥	🟥
bram	🟥	🟥	🟩	🟩	🟩	🟥	🟥	🟥
ap_fifo	🟥	🟥	🟩	🟥	🟩	🟩	🟥	🟩

🟩 Supported. D = Default Interface

🟥 Not Supported

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Default I/O Protocols

For every type of C argument in the function code, there is a default I/O protocol associated with it

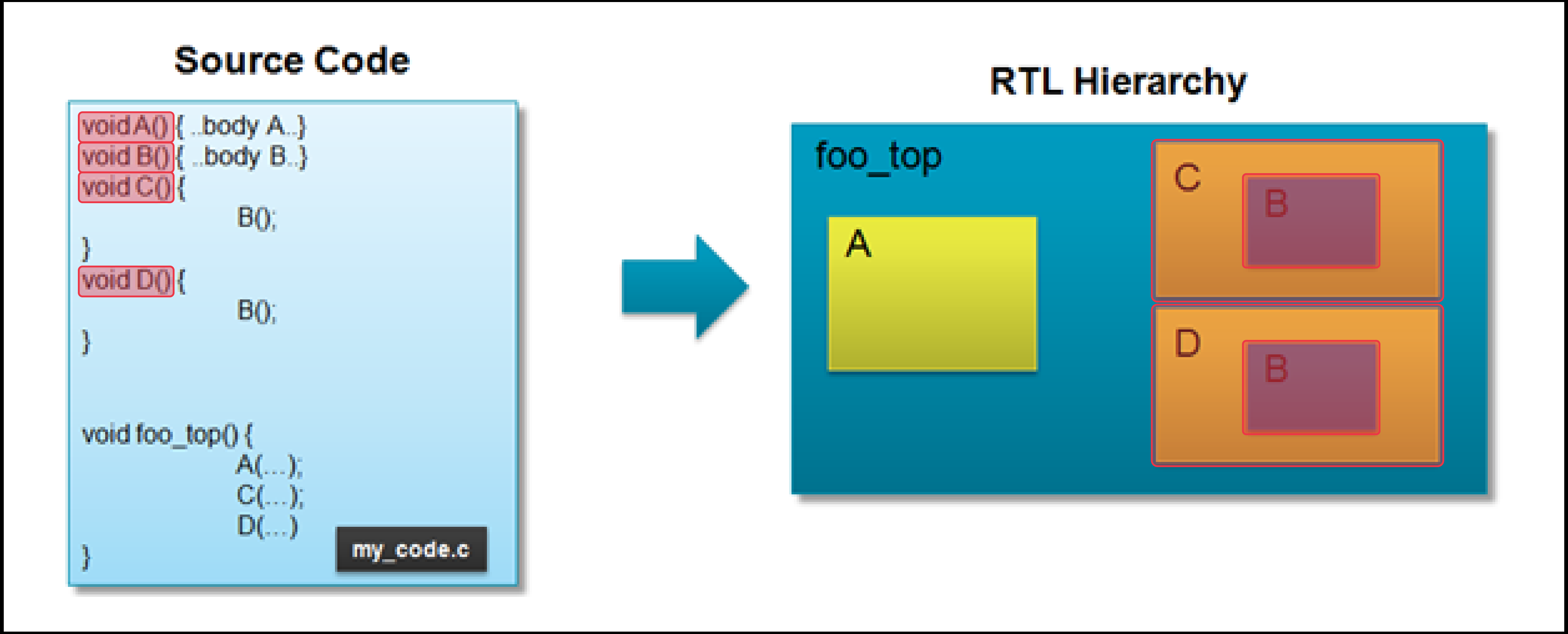
C Argument Type	Default I/O Protocol
Input	ap_none
Output	ap_vld
Inout In port of inout Out port of inout	ap_ovld ap_none ap_vld
Arrays	ap_memory

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Function Inlining

Each function in the design gets synthesized into an RTL block, maintaining the design hierarchy



Source Code

```
void A() { ..body A..}
void B() { ..body B..}
void C() {
                B();
}
void D() {
                B();
}


void foo_top() {
                A(...);
                C(...);
                D(...)
}
```

© Copyright 2023 Advanced Micro Devices, Inc.

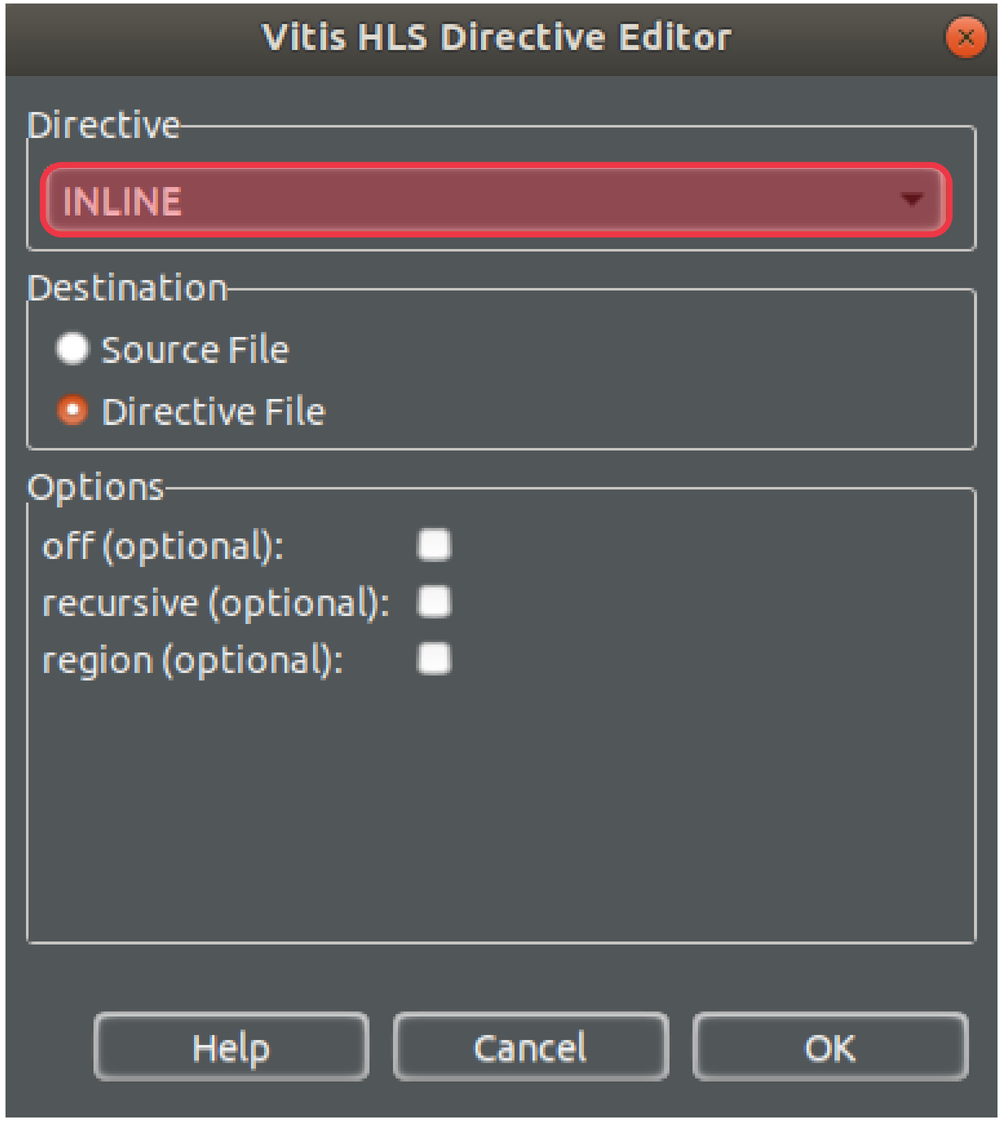
AMD
together we advance_

Function Inlining

INLINE directive → Removes function hierarchy

Improves the area by allowing the components within the function to be better shared or optimized with the logic in the calling function

Vitis™ HLS tool performs some inlining automatically on small logic functions


Vitis HLS Directive Editor

Directive: INLINE

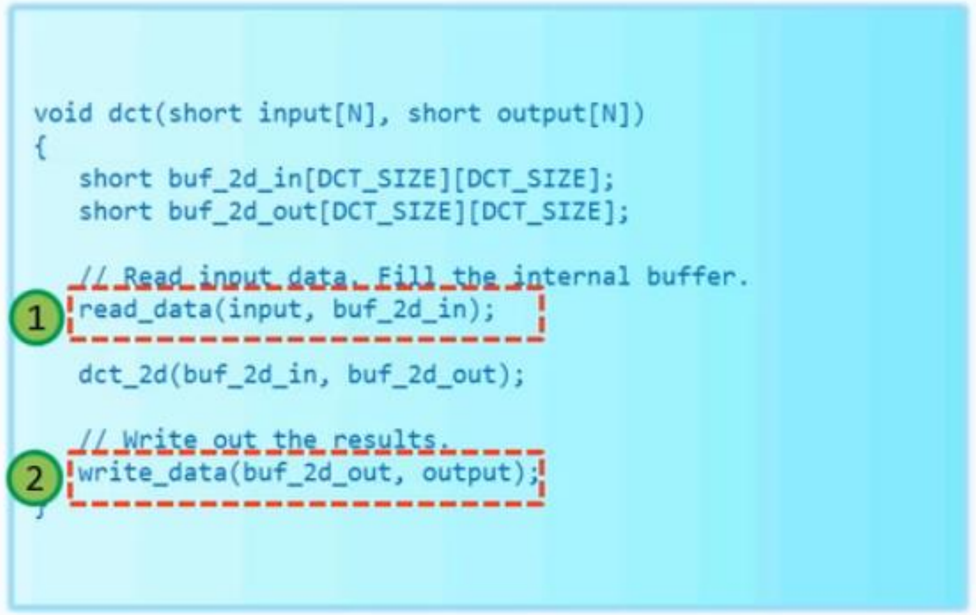
Destination: Source File [✓ Directive File]

Options:
off (optional): ☐
recursive (optional): ☐
region (optional): ☐

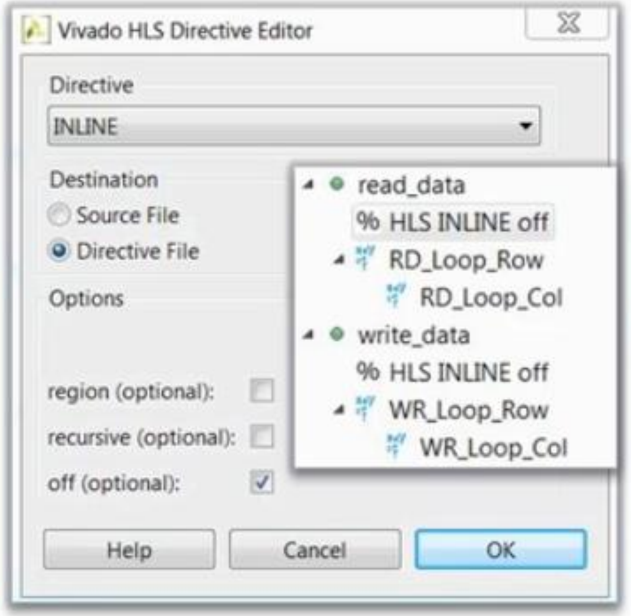
© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

Function Inlining



```
void dct(short input[N], short output[N])
{
   short buf_2d_in[DCT_SIZE][DCT_SIZE];
   short buf_2d_out[DCT_SIZE][DCT_SIZE];

   // Read input data. Fill the internal buffer.
   read_data(input, buf_2d_in);

   dct_2d(buf_2d_in, buf_2d_out);

   // Write out the results.
   write_data(buf_2d_out, output);
}
```



Directive: INLINE

Destination: ( ) Source File [✓ Directive File]

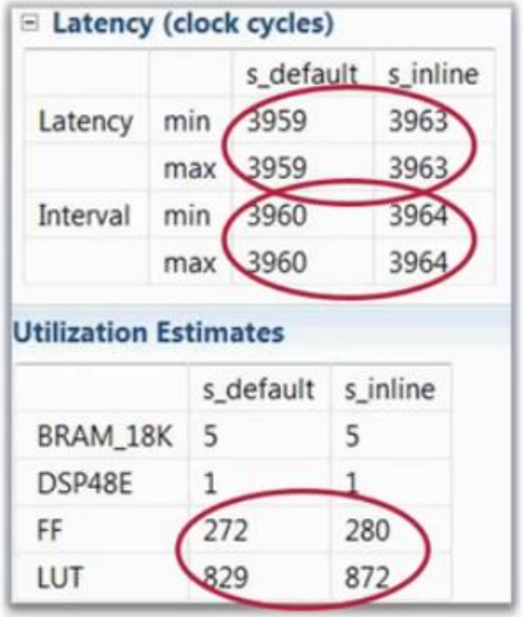
Options

region (optional): ☐

recursive (optional): ☐

off (optional): ✓

- read_data
  - % HLS INLINE off
  - RD_Loop_Row
    - RD_Loop_Col
- write_data
  - % HLS INLINE off
  - WR_Loop_Row
    - WR_Loop_Col

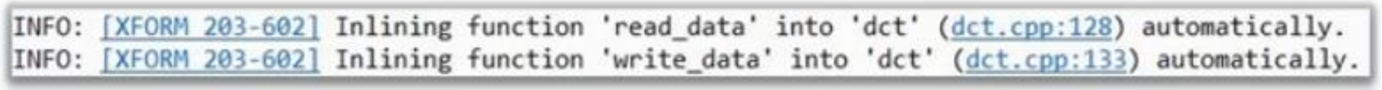


Latency (clock cycles)

		s_default	s_inline
Latency	min	3959	3963
	max	3959	3963
Interval	min	3960	3964
	max	3960	3964

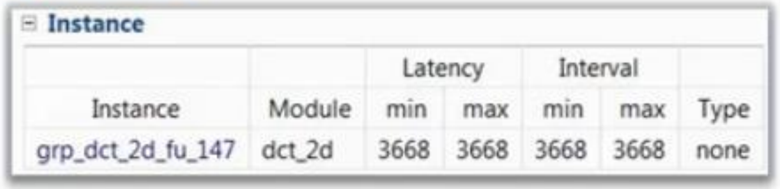
Utilization Estimates

	s_default	s_inline
BRAM_18K	5	5
DSP48E	1	1
FF	272	280
LUT	829	872



```
INFO: [XFORM 203-602] Inlining function 'read_data' into 'dct' (dct.cpp:128) automatically.
INFO: [XFORM 203-602] Inlining function 'write_data' into 'dct' (dct.cpp:133) automatically.
```

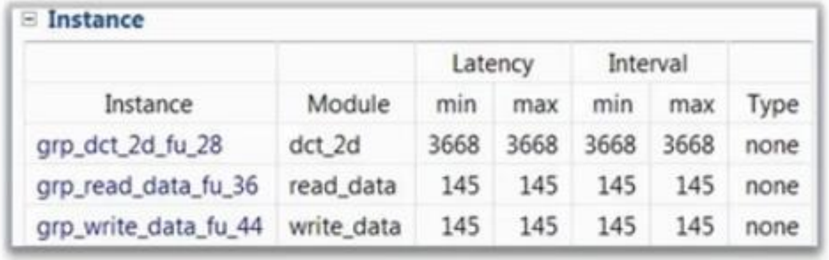
By default



Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
grp_dct_2d_fu_147	dct_2d	3668	3668	3668	3668	none

Inline off



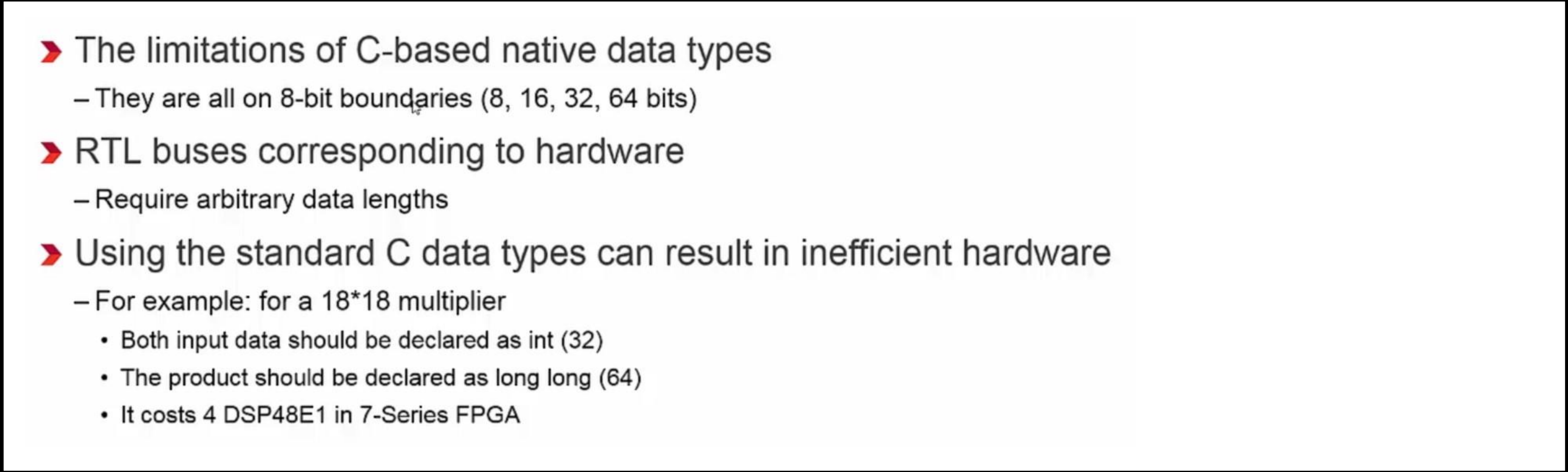
Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
grp_dct_2d_fu_28	dct_2d	3668	3668	3668	3668	none
grp_read_data_fu_36	read_data	145	145	145	145	none
grp_write_data_fu_44	write_data	145	145	145	145	none

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Arbitrary Precision Integers



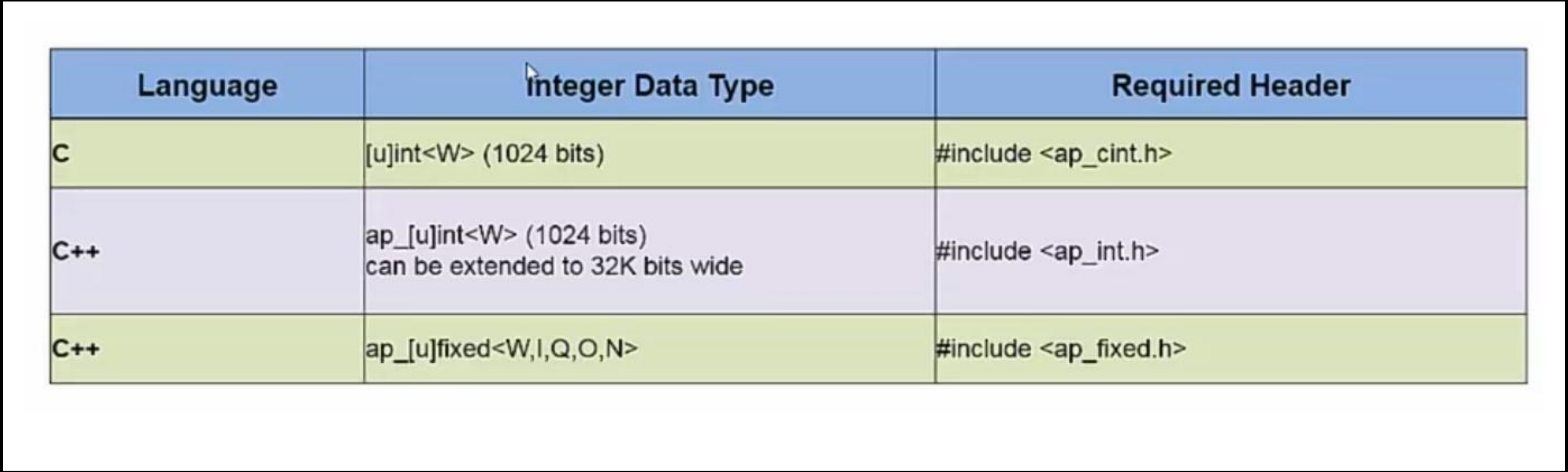
- The limitations of C-based native data types
  - They are all on 8-bit boundaries (8, 16, 32, 64 bits)
- RTL buses corresponding to hardware
  - Require arbitrary data lengths
- Using the standard C data types can result in inefficient hardware
  - For example: for a 18*18 multiplier
    - Both input data should be declared as int (32)
    - The product should be declared as long long (64)
    - It costs 4 DSP48E1 in 7-Series FPGA

C and C++ languages have standard types created on the 8-bit boundary

- Usually have fixed size (character = 8 bits, integer = 32 bits, and long = 64 bits)
- Implemented hardware sometimes need different sizes of data types
- Results may not be bit accurate and can give sub-standard QoR

© Copyright 2023 Advanced Micro Devices, Inc.

Arbitrary Precision Integers



Language	Integer Data Type	Required Header
C	[u]int<W> (1024 bits)	#include <ap_cint.h>
C++	ap_[u]int<W> (1024 bits) can be extended to 32K bits wide	#include <ap_int.h>
C++	ap_[u]fixed<W,I,Q,O,N>	#include <ap_fixed.h>

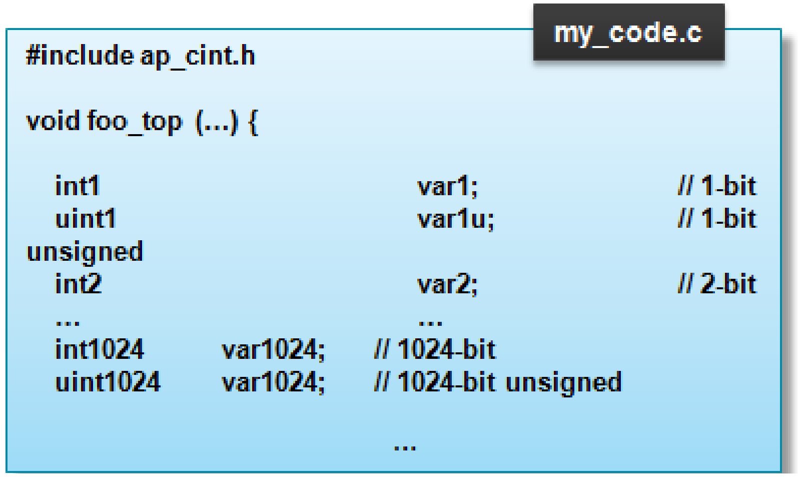
C and C++ languages have standard types created on the 8-bit boundary

- Usually have fixed size (character = 8 bits, integer = 32 bits, and long = 64 bits)
- Implemented hardware sometimes need different sizes of data types
- Results may not be bit accurate and can give sub-standard QoR

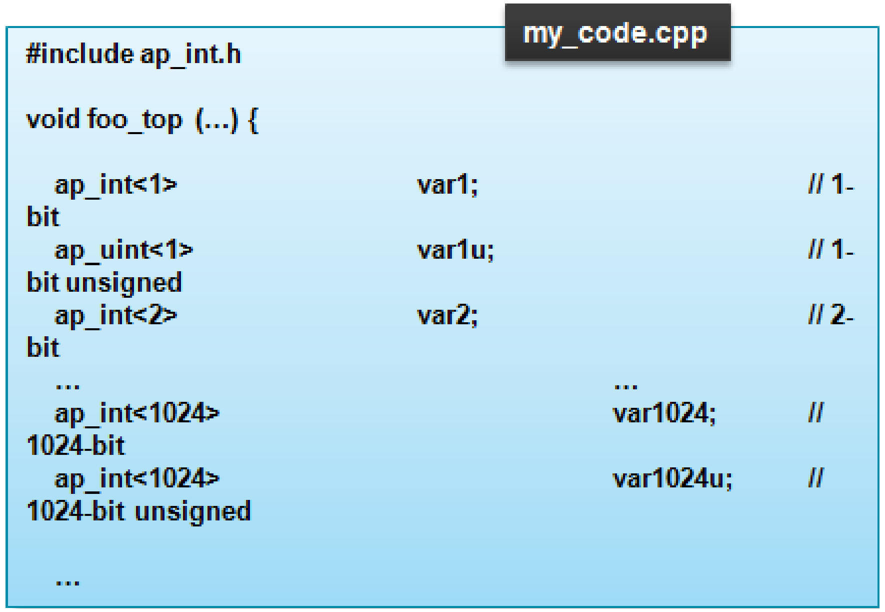
© Copyright 2023 Advanced Micro Devices, Inc.

together we advance_

Arbitrary Precision Integers



```
#include ap_cint.h

void foo_top  (…) {

   int1                          var1;          // 1-bit
   uint1                         var1u;         // 1-bit
unsigned
   int2                          var2;          // 2-bit
   …                             …
   int1024      var1024;    // 1024-bit
   uint1024     var1024;    // 1024-bit unsigned

                           …
```



```
#include ap_int.h

void foo_top  (…) {

   ap_int<1>                  var1;                    // 1-
bit
   ap_uint<1>                 var1u;                   // 1-
bit unsigned
   ap_int<2>                  var2;                    // 2-
bit
   …                                      …
   ap_int<1024>                           var1024;     //
1024-bit
   ap_int<1024>                           var1024u;    //
1024-bit unsigned

   …
```

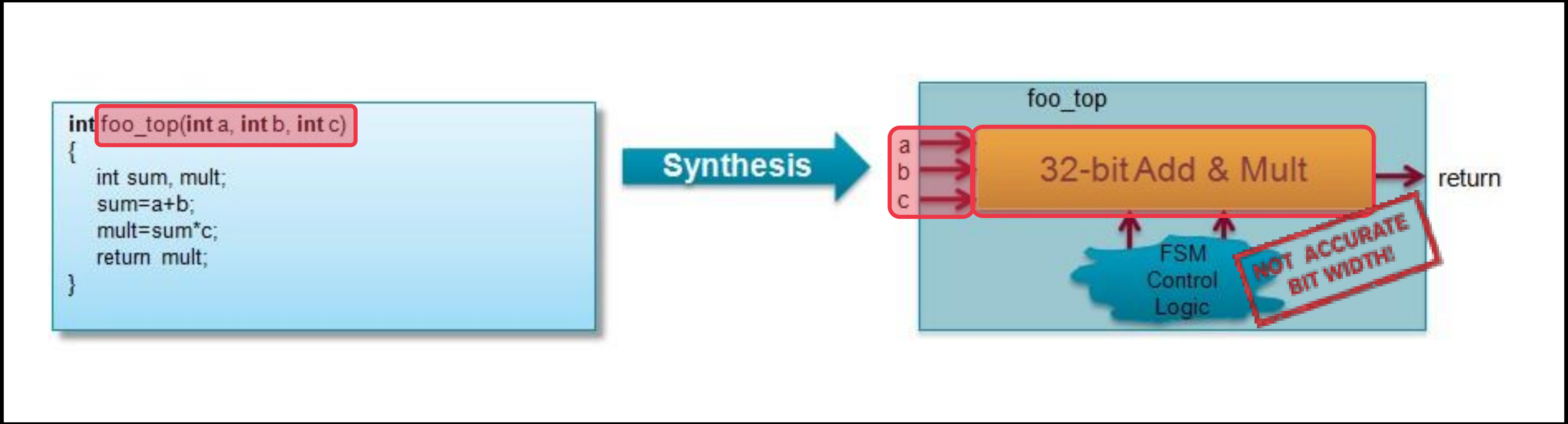
C and C++ languages have standard types created on the 8-bit boundary

- Usually have fixed size (character = 8 bits, integer = 32 bits, and long = 64 bits)
- Implemented hardware sometimes need different sizes of data types
- Results may not be bit accurate and can give sub-standard QoR

© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

Arbitrary Precision Integers



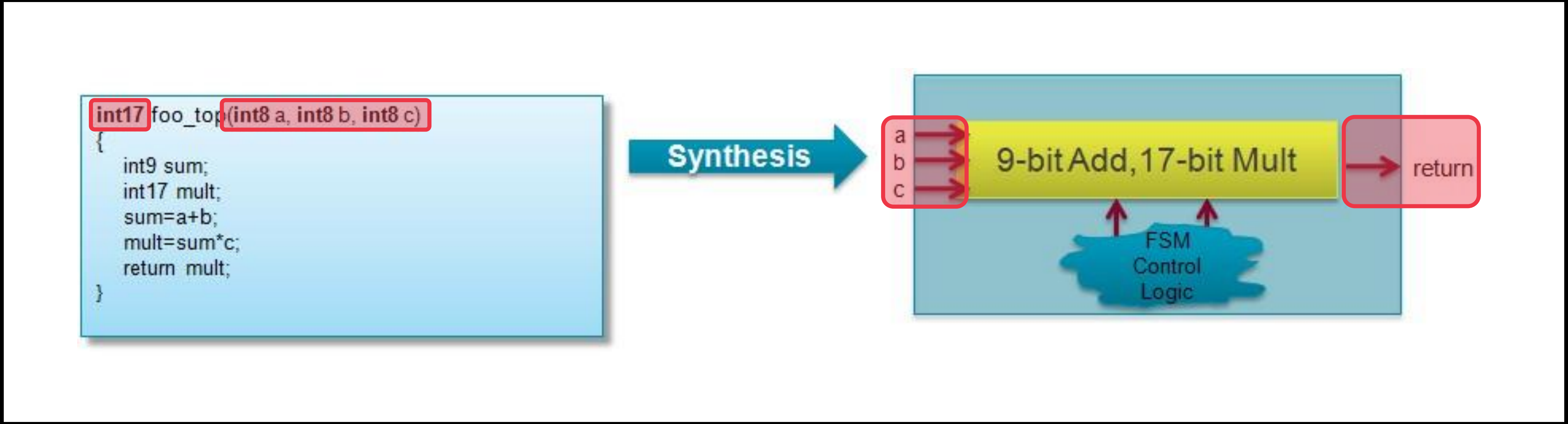
```
int foo_top(int a, int b, int c)
{
    int sum, mult;
    sum=a+b;
    mult=sum*c;
    return  mult;
}
```

Signals with arbitrary widths are also converted into fixed sized widths when native C data types are used

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Arbitrary Precision Integers



```
int17 foo_top(int8 a, int8 b, int8 c)
{
    int9 sum;
    int17 mult;
    sum=a+b;
    mult=sum*c;
    return mult;
}
```

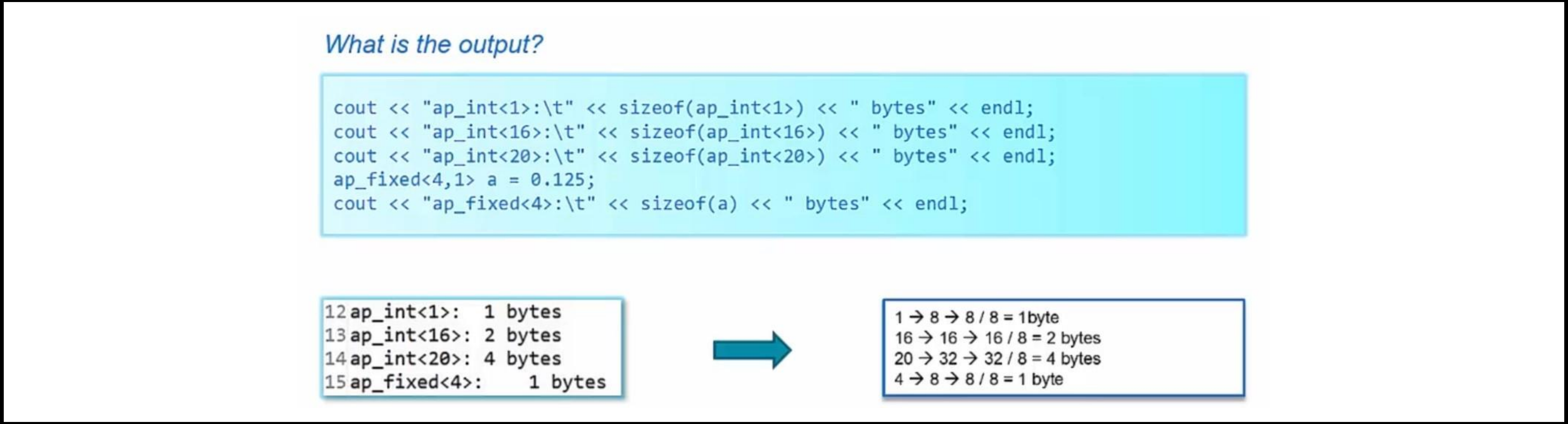
Usage of bit-accurate widths results in smaller and faster hardware with full precision

Full precision can be simulated/validated with C simulation and hardware will behave the same

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Arbitrary Precision Integers



What is the output?

```
cout << "ap_int<1>:\t" << sizeof(ap_int<1>) << " bytes" << endl;
cout << "ap_int<16>:\t" << sizeof(ap_int<16>) << " bytes" << endl;
cout << "ap_int<20>:\t" << sizeof(ap_int<20>) << " bytes" << endl;
ap_fixed<4,1> a = 0.125;
cout << "ap_fixed<4>:\t" << sizeof(a) << " bytes" << endl;
```

```
ap_int<1>:   1 bytes
ap_int<16>:  2 bytes
ap_int<20>:  4 bytes
ap_fixed<4>:     1 bytes
```

1 → 8 → 8 / 8 = 1byte
16 → 16 → 16 / 8 = 2 bytes
20 → 32 → 32 / 8 = 4 bytes
4 → 8 → 8 / 8 = 1 byte

Usage of bit-accurate widths results in smaller and faster hardware with full precision

Full precision can be simulated/validated with C simulation and hardware will behave the same

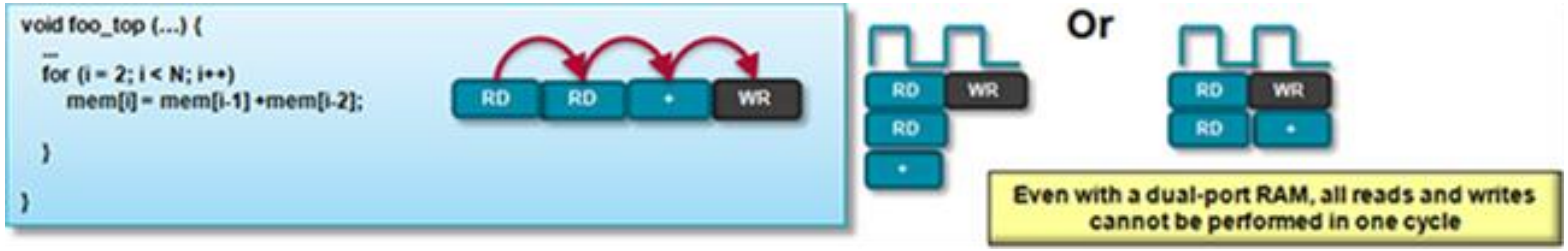
© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

Arrays: Performance Bottlenecks

Arrays are intuitive and useful software constructs

Allow the C algorithm to be easily captured and understood



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

  }

}
```

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

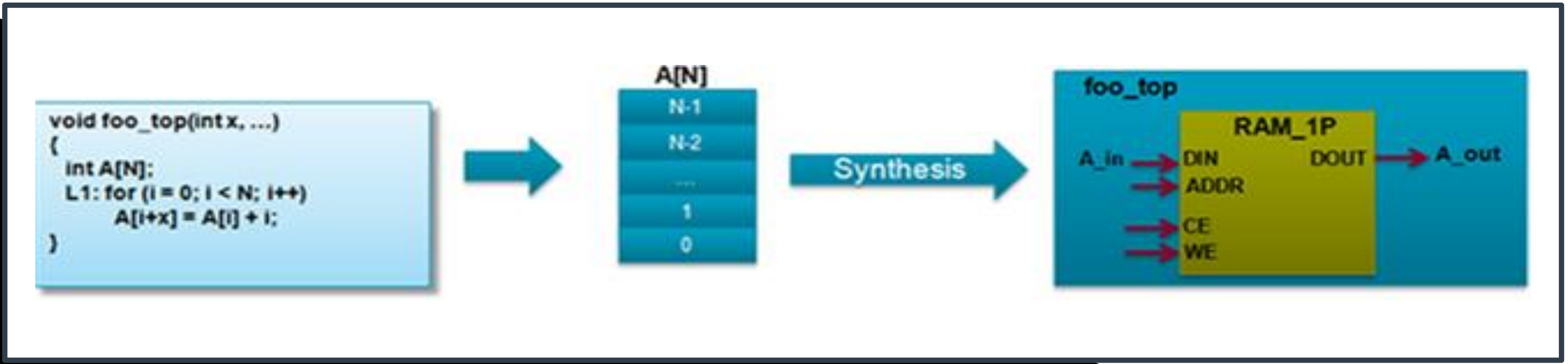
Solution:

- Array can be partitioned and reshaped to produce higher data bandwidth
- Allows more optimal configuration of the array
- Provides a better implementation of the memory resource

© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

Arrays in HLS



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

Arrays in the C code → Block RAM elements in RTL

To use a FIFO instead of a block RAM → STREAM directive

Arrays are automatically specified as streaming:

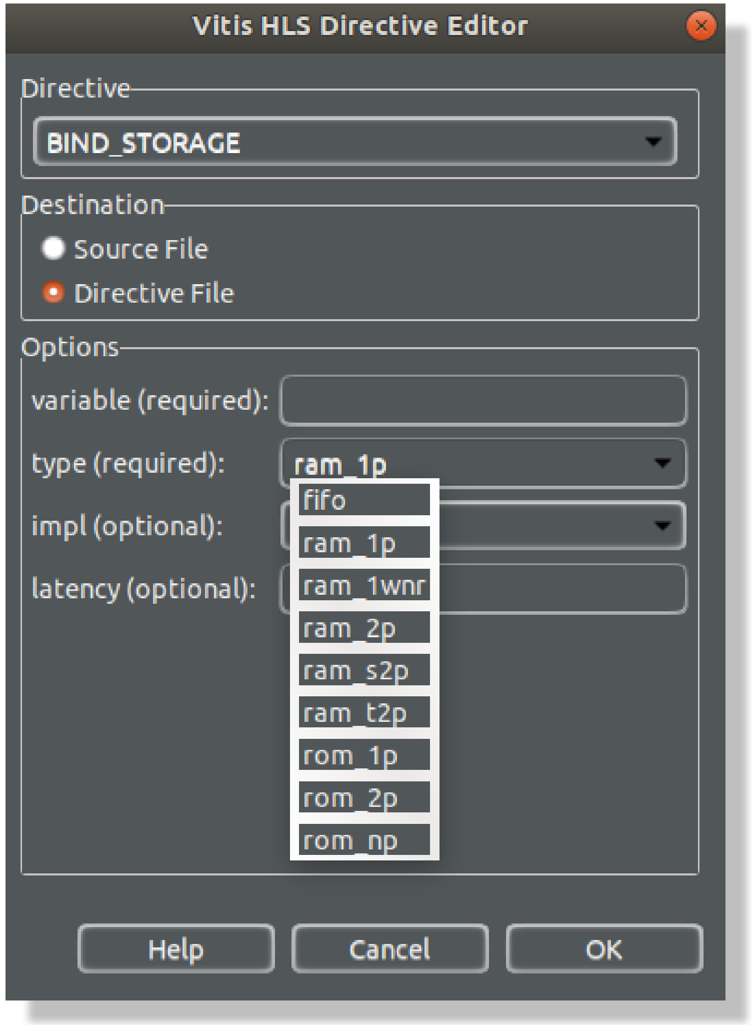
- If an array is set as interface type ap_fifo, axis, or ap_hs, etc
- If the arrays are used in a region where the DATAFLOW optimization is applied

All other arrays must be specified as streaming using the STREAM directive if a FIFO is required for the implementation

© Copyright 2023 Advanced Micro Devices, Inc.

Arrays in HLS

Any memory resource in the library can be targeted while using the arrays



Vitis HLS Directive Editor

Directive: BIND_STORAGE

Destination:
Source File [✓ Directive File]

Options:
variable (required):
type (required): ram_1p
impl (optional):
latency (optional):

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Array and RAM Selection

BIND_STORAGE directive

Type of RAM

RAM port: Single port or dual port

If no directive is specified	If no RAM target is specified	If RAM target is specified
• Single-port RAM by default • Dual-port RAM if it reduces the II or latency	RTL synthesis will determine if RAM is implemented as block RAM or LUTRAM	Vitis™ HLS tool will obey the target selected

© Copyright 2023 Advanced Micro Devices, Inc.

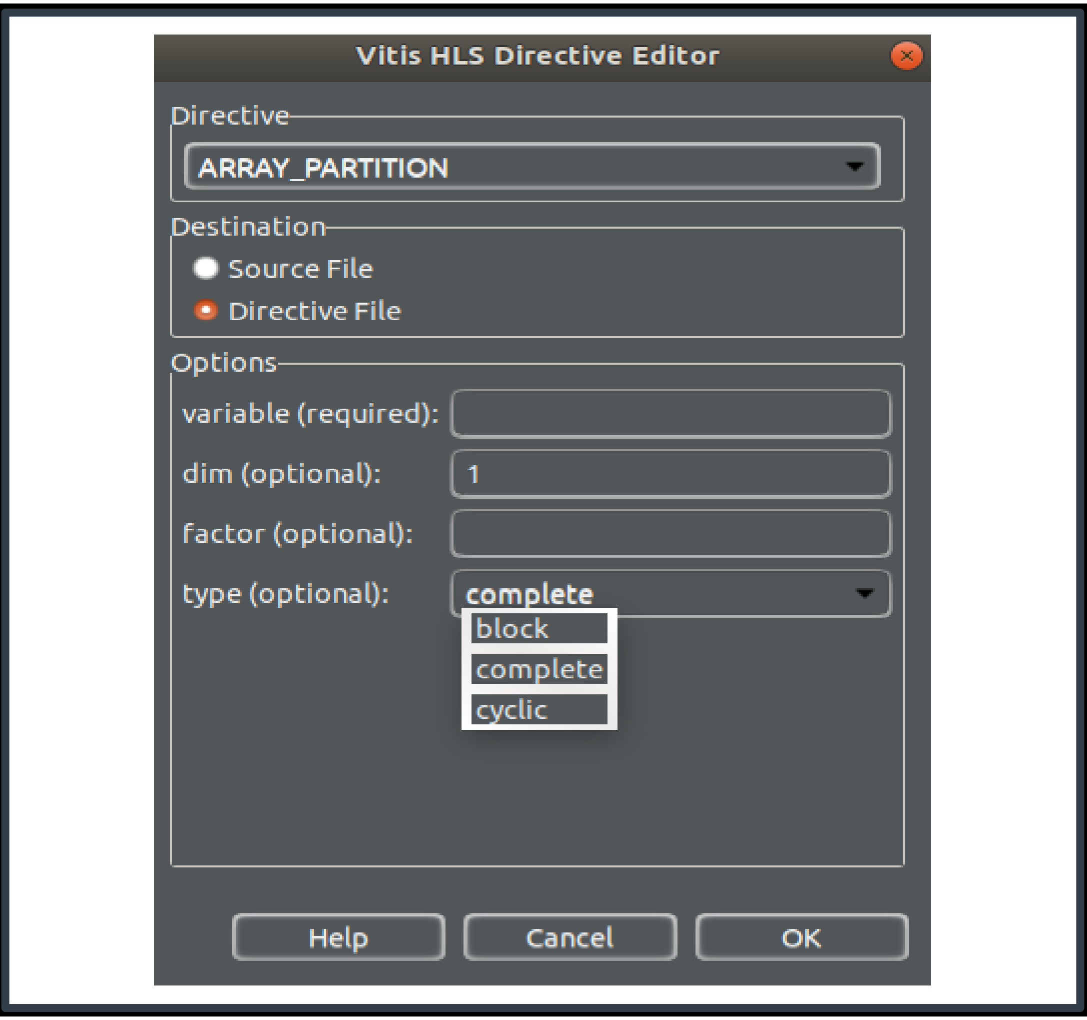
AMD
together we advance_

Array Partitioning

Partitions the large arrays into multiple smaller arrays or individual registers to improve parallel access to data and remove block RAM bottlenecks

Types of array partitioning:

- **Block:** Original array is split into equally sized blocks of consecutive elements of the original array
- **Complete:** Default operation is to split the array into its elements. This corresponds to resolving a memory into registers
- **Cyclic:** Original array is split into equally sized blocks, interleaving the elements of the original array



Vitis HLS Directive Editor

Directive: ARRAY_PARTITION

Destination: Source File [✓ Directive File]

Options

variable (required):

dim (optional): 1

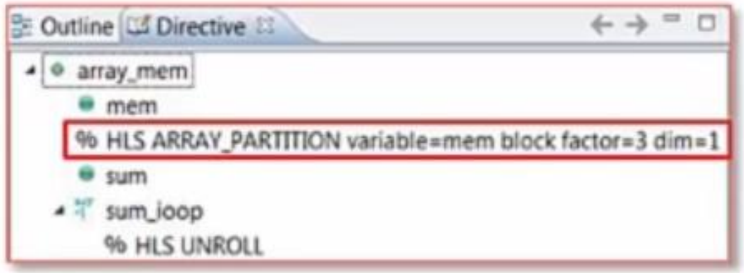
factor (optional):

type (optional): complete

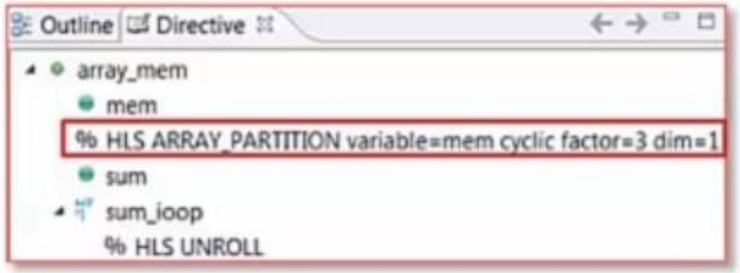
© Copyright 2023 Advanced Micro Devices, Inc.

AMD together we advance_

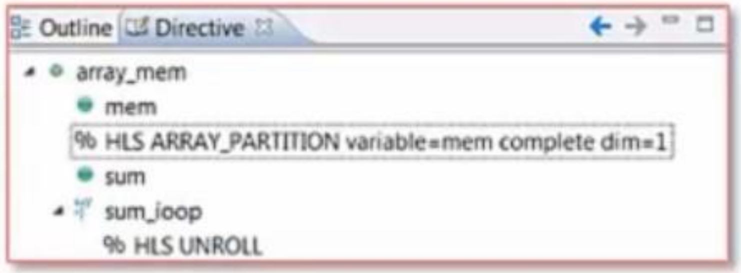
Array Partitioning



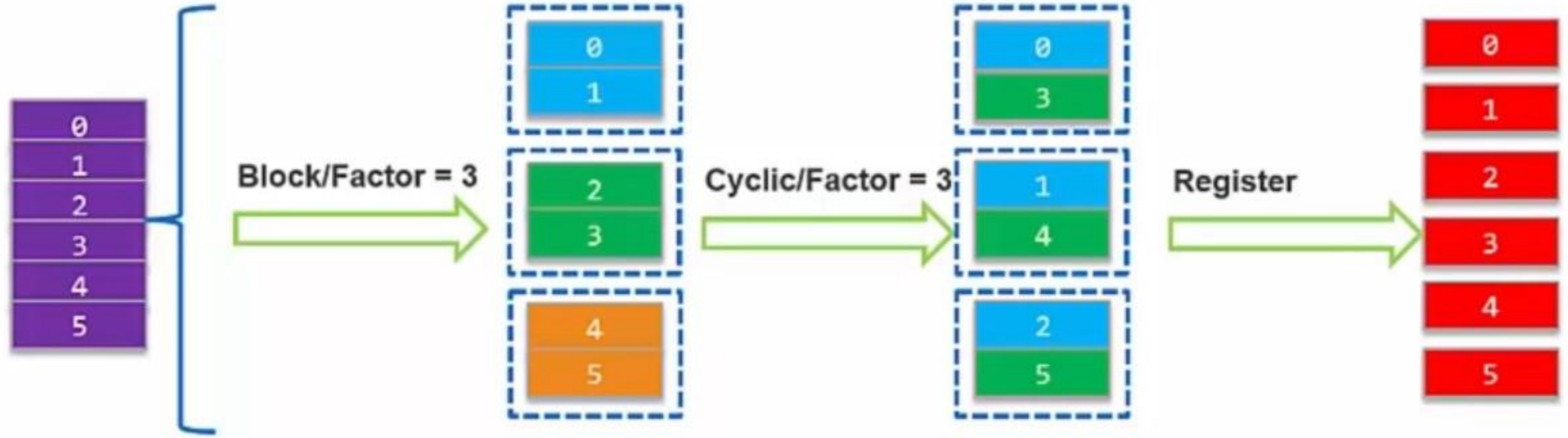
```
array_mem
  mem
  % HLS ARRAY_PARTITION variable=mem block factor=3 dim=1
  sum
  sum_loop
    % HLS UNROLL
```



```
array_mem
  mem
  % HLS ARRAY_PARTITION variable=mem cyclic factor=3 dim=1
  sum
  sum_loop
    % HLS UNROLL
```



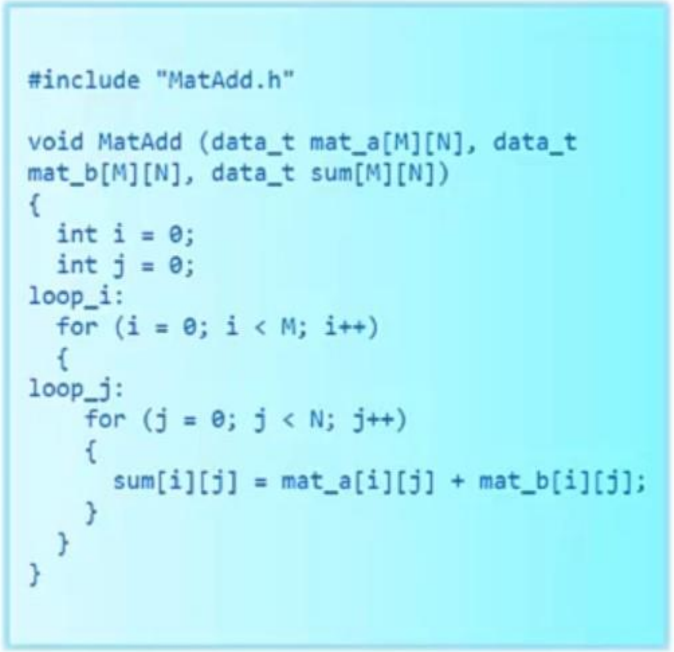
```
array_mem
  mem
  % HLS ARRAY_PARTITION variable=mem complete dim=1
  sum
  sum_loop
    % HLS UNROLL
```



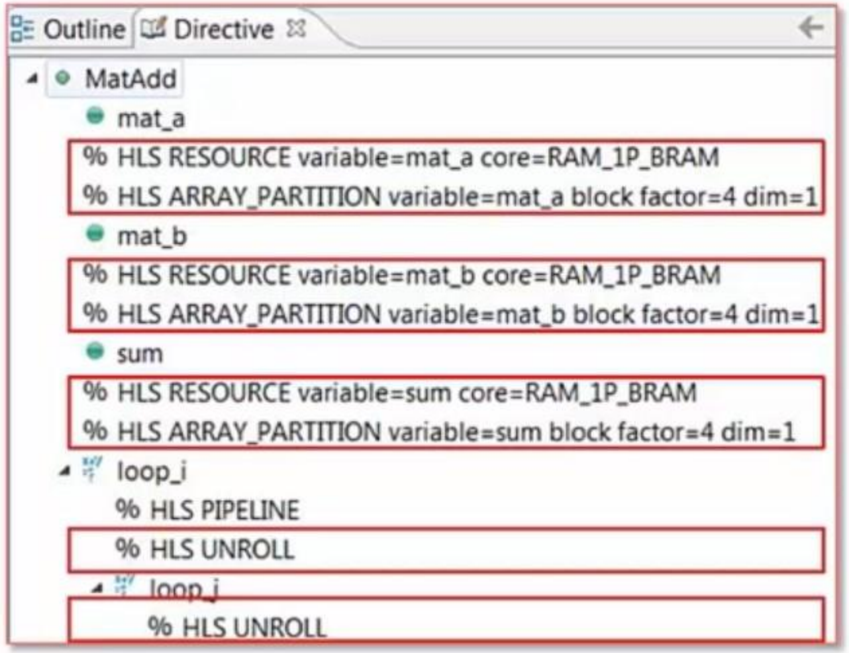
© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

Array Partitioning

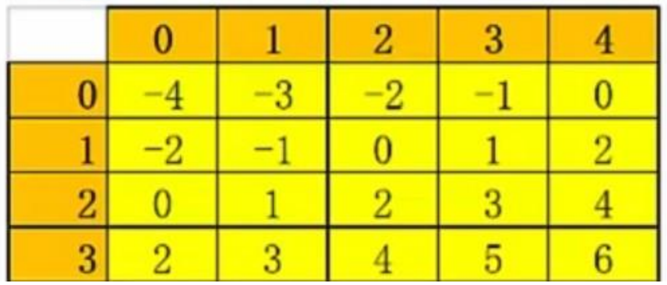


```
#include "MatAdd.h"

void MatAdd (data_t mat_a[M][N], data_t
mat_b[M][N], data_t sum[M][N])
{
  int i = 0;
  int j = 0;
loop_i:
  for (i = 0; i < M; i++)
  {
loop_j:
    for (j = 0; j < N; j++)
    {
      sum[i][j] = mat_a[i][j] + mat_b[i][j];
    }
  }
}
```



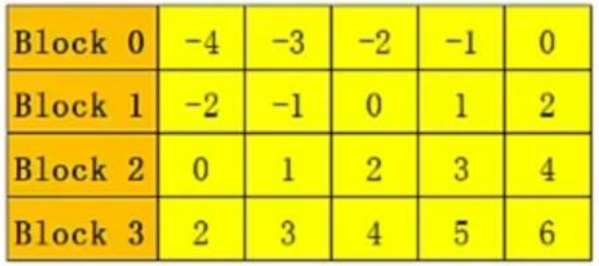
```
MatAdd
  mat_a
    % HLS RESOURCE variable=mat_a core=RAM_1P_BRAM
    % HLS ARRAY_PARTITION variable=mat_a block factor=4 dim=1
  mat_b
    % HLS RESOURCE variable=mat_b core=RAM_1P_BRAM
    % HLS ARRAY_PARTITION variable=mat_b block factor=4 dim=1
  sum
    % HLS RESOURCE variable=sum core=RAM_1P_BRAM
    % HLS ARRAY_PARTITION variable=sum block factor=4 dim=1
  loop_i
    % HLS PIPELINE
    % HLS UNROLL
    loop_j
      % HLS UNROLL
```

mat_a: 4x5



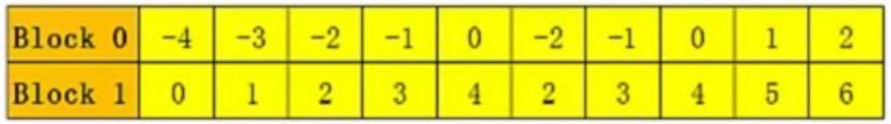
	0	1	2	3	4
0	-4	-3	-2	-1	0
1	-2	-1	0	1	2
2	0	1	2	3	4
3	2	3	4	5	6

Block factor = 4, dim = 1



Block 0	-4	-3	-2	-1	0
Block 1	-2	-1	0	1	2
Block 2	0	1	2	3	4
Block 3	2	3	4	5	6

Block factor = 2, dim = 1



Block 0	-4	-3	-2	-1	0	-2	-1	0	1	2
Block 1	0	1	2	3	4	2	3	4	5	6

© Copyright 2023 Advanced Micro Devices, Inc.

AMD
together we advance_

AMD