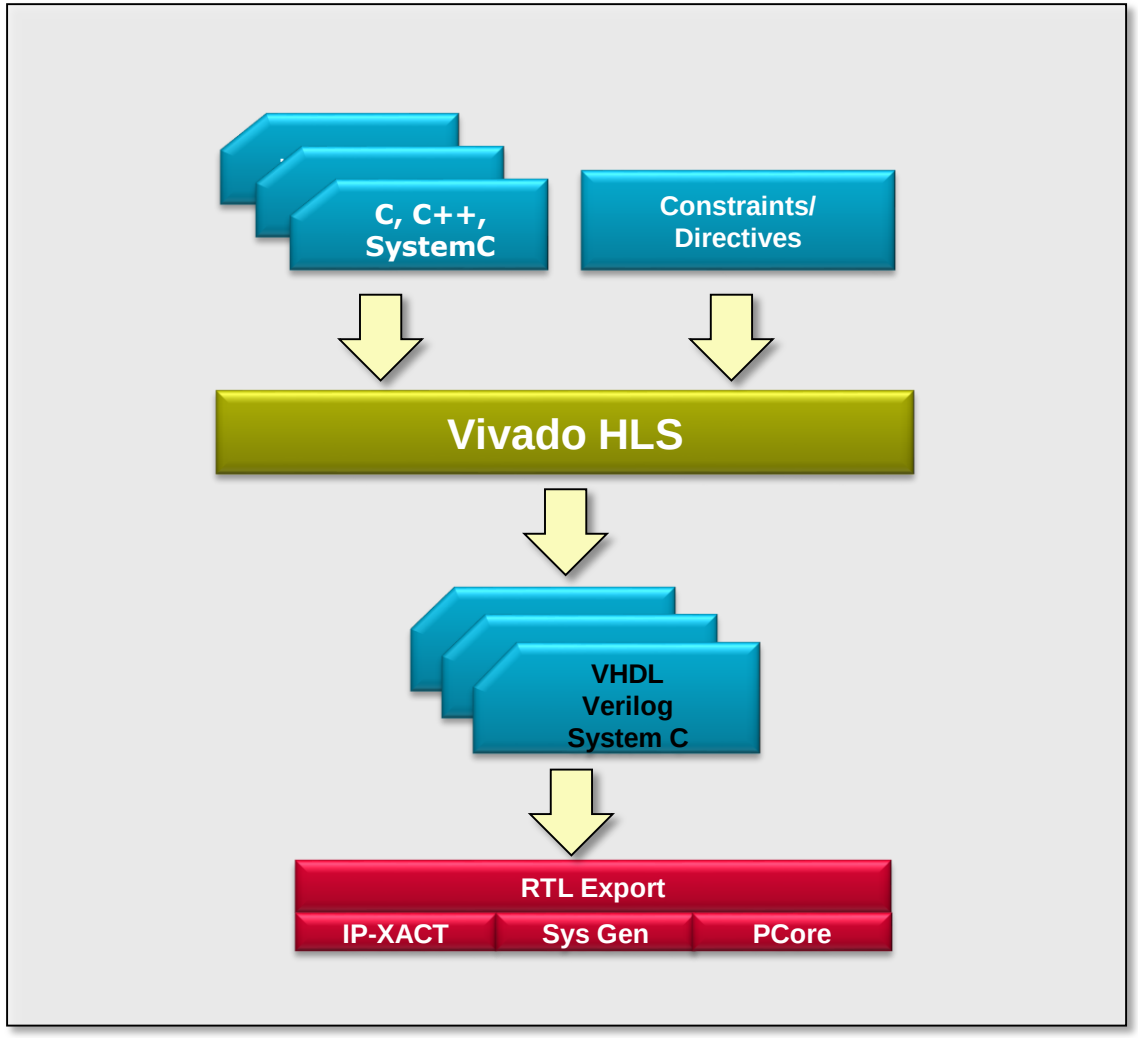
High-Level Synthesis with Vivado HLS

High-Level Synthesis: HLS

- **High-Level Synthesis**
  - Creates an RTL implementation from C level source code
  - Extracts control and dataflow from the source code
  - Implements the design based on defaults and user applied directives
- **Many implementation are possible from the same source description**
  - Smaller designs, faster designs, optimal designs
  - Enables design exploration



Design Exploration with Directives

One body of code:
Many hardware outcomes

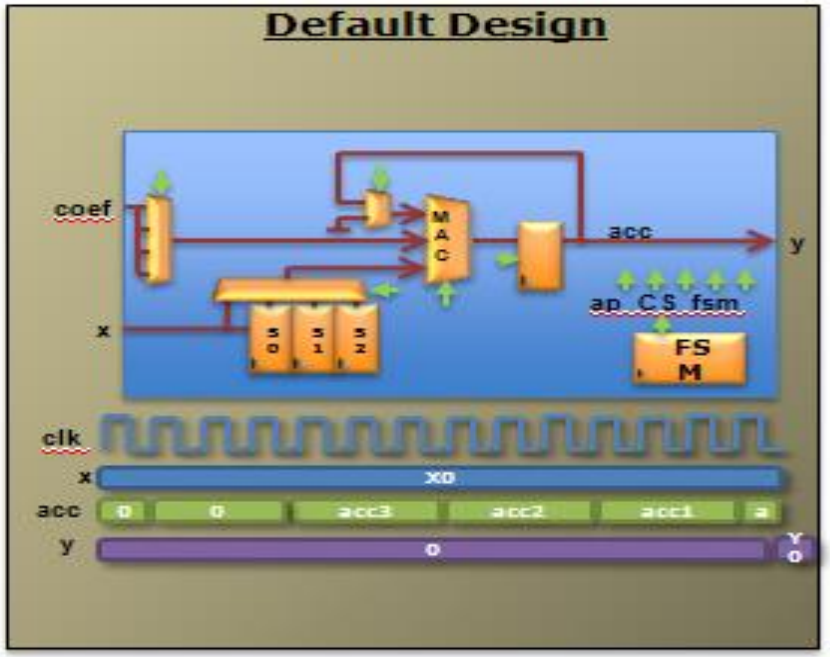
```
…
loop:  for (i=3;i>=0;i--) {
   if (i==0) {
      acc+=x*c[0];
      shift_reg[0]=x;
   } else {
      shift_reg[i]=shift_reg[i-1];
      acc+=shift_reg[i]*c[i];
   }
}
….
```

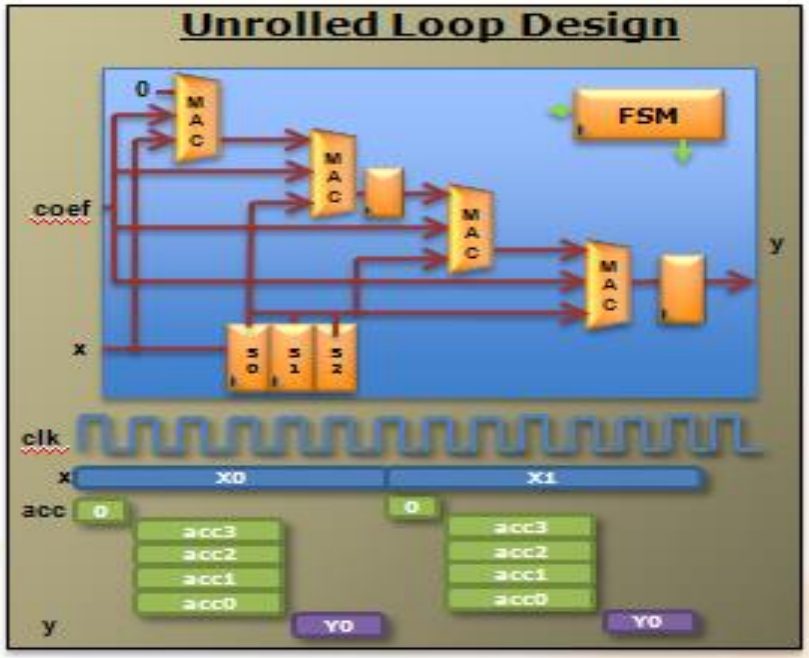
Before we get into details, let's look under the hood ….

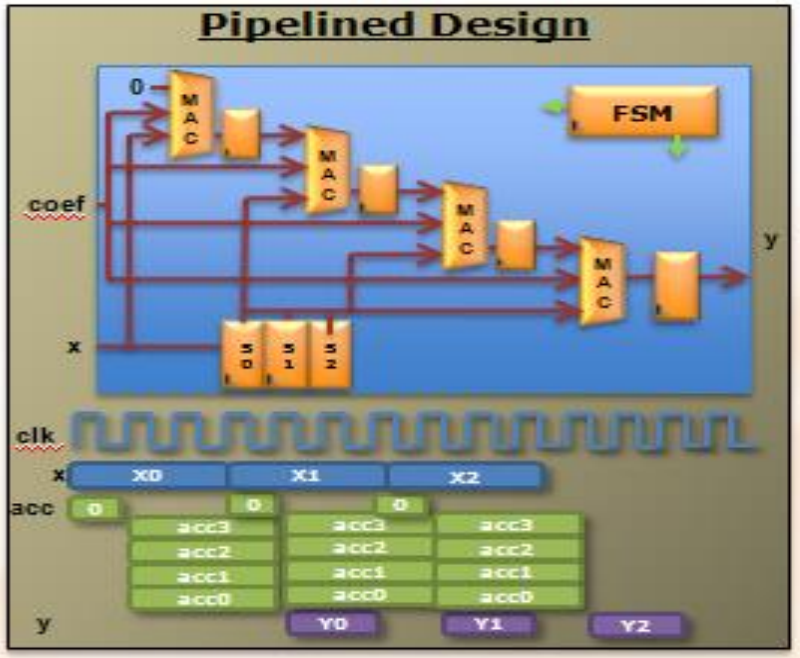
The same hardware is used for each iteration of the loop:
- **Small area**
- **Long latency**
- **Low throughput**

Different hardware is used for each iteration of the loop:
- **Higher area**
- **Short latency**
- **Better throughput**

Different iterations are executed concurrently:
- **Higher area**
- **Short latency**
- **Best throughput**







Introduction to High-Level Synthesis

- **How is hardware extracted from C code?**
  - Control and datapath can be extracted from C code at the top level
  - The same principles used in the example can be applied to sub-functions
    - At some point in the top-level control flow, control is passed to a sub-function
    - Sub-function may be implemented to execute concurrently with the top-level and or other sub-functions
- **How is this control and dataflow turned into a hardware design?**
  - Vivado HLS maps this to hardware through scheduling and binding processes
- **How is my design created?**
  - How functions, loops, arrays and IO ports are mapped?

HLS: Control Extraction

Code

```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
  ) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

  acc=0;
  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];
    }
  }
  *y=acc;
}
```

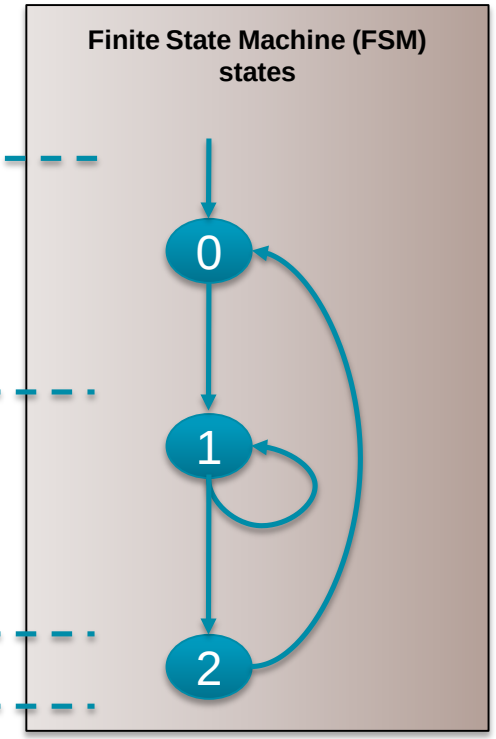
Function Start

For-Loop Start

For-Loop End

Function End

Control Behavior



From any C code example ..

The loops in the C code correlated to states of behavior

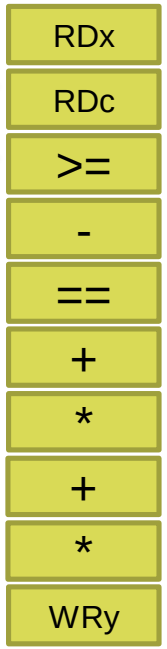
This behavior is extracted into a hardware state machine

HLS: Control & Datapath Extraction

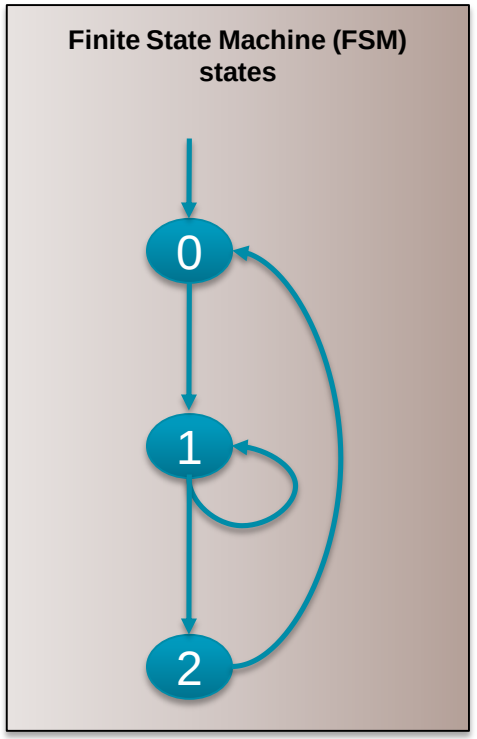
Code

```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
  ) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

  acc=0;
  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];
    }
  }
  *y=acc;
}
```

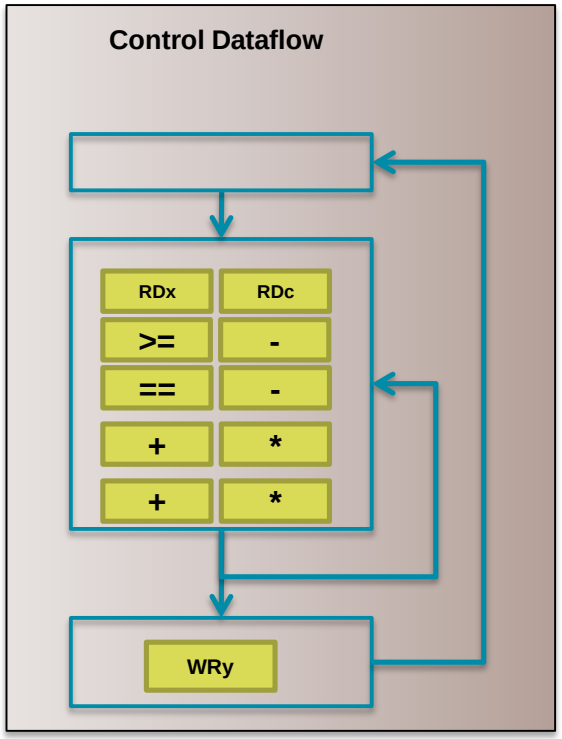
Operations



Control Behavior



Control & Datapath Behavior



From any C code example ..

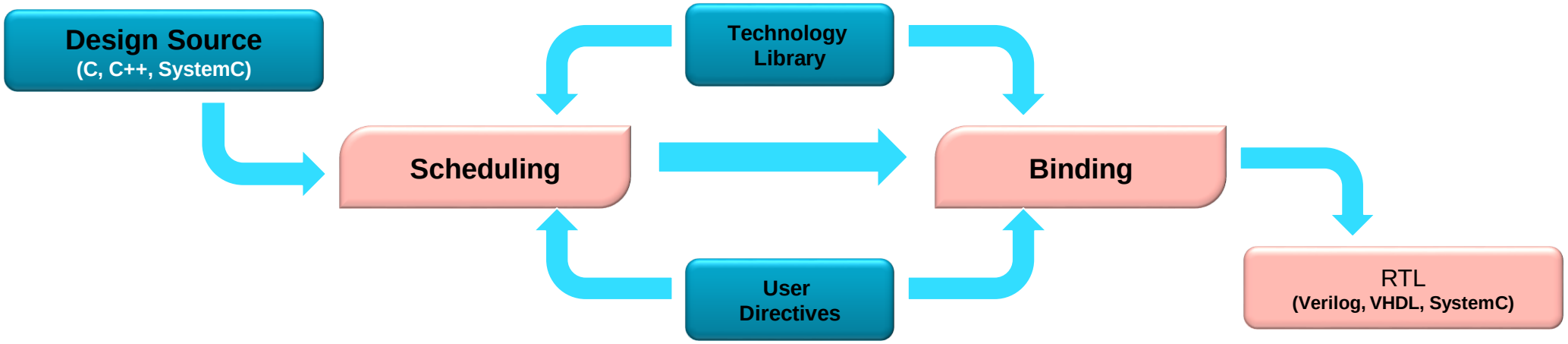
Operations are extracted…

The control is known

A unified control dataflow behavior is created.

High-Level Synthesis: Scheduling & Binding

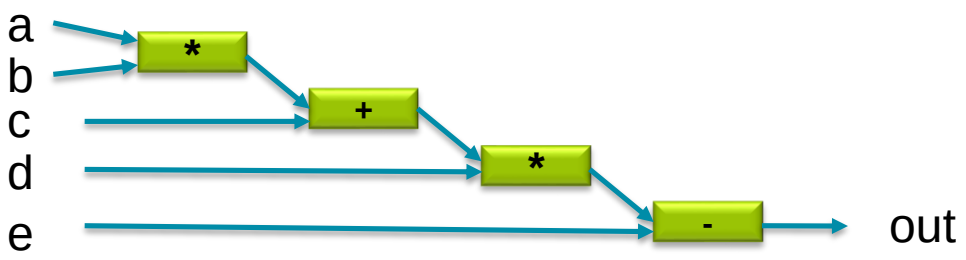
- **Scheduling & Binding**
  - Scheduling and Binding are at the heart of HLS
- **Scheduling determines in which clock cycle an operation will occur**
  - Takes into account the control, dataflow and user directives
  - The allocation of resources can be constrained
- **Binding determines which library cell is used for each operation**
  - Takes into account component delays, user directives



Scheduling

- **The operations in the control flow graph are mapped into clock cycles**

```
void foo (
 …
  t1 = a * b;
  t2 = c + t1;
  t3 = d * t2;
  out = t3 – e;
}
```



Schedule 1

- **The technology and user constraints impact the schedule**
  - A faster technology (or slower clock) may allow more operations to occur in the same clock cycle

Schedule 2

- **The code also impacts the schedule**
  - Code implications and data dependencies must be obeyed

Binding

- **Binding is where operations are mapped to cores from the hardware library**
  - Operators map to cores

- **Binding Decision: to share**
  - Given this schedule:

    - Binding must use 2 multipliers, since both are in the same cycle
    - It can decide to use an adder and subtractor or one addsub

- **Binding Decision: or not to share**
  - Given this schedule:

    - Binding may decide to share the multipliers (each is used in a different cycle)
    - Or it may decide the cost of sharing (muxing) would impact timing and it may decide not to share them
    - It may make this same decision in the first example above too

Understanding Vivado HLS Synthesis

- **HLS**
  - Vivado HLS determines in which cycle operations should occur (scheduling)
  - Determines which hardware units to use for each operation (binding)
  - It performs HLS by :
    - Obeying built-in defaults
    - Obeying user directives & constraints to override defaults
    - Calculating delays and area using the specified technology/device
- **Understand Vivado HLS defaults**
  - Key to understanding the initial design created by Vivado HLS
- **Understand the priority of directives**
  1. Meet Performance (clock & throughput)
     - Vivado HLS will allow a local clock path to fail if this is required to meet throughput
     - Often possible the timing can be met after logic synthesis
  2. Then minimize latency
  3. Then minimize area

The Key Attributes of C code

```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
  ) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

  acc=0;
  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];
}
  }
  *y=acc;
}
```

Functions: All code is made up of functions which represent the design hierarchy: the same in hardware

Top Level IO : The arguments of the top-level function determine the hardware RTL interface ports

Types: All variables are of a defined type. The type can influence the area and performance

Loops: Functions typically contain loops. How these are handled can have a major impact on area and performance

Arrays: Arrays are used often in C code. They can influence the device IO and become performance bottlenecks

Operators: Operators in the C code may require sharing to control area or specific hardware implementations to meet performance

Let's examine the default synthesis behavior of these …

Functions & RTL Hierarchy

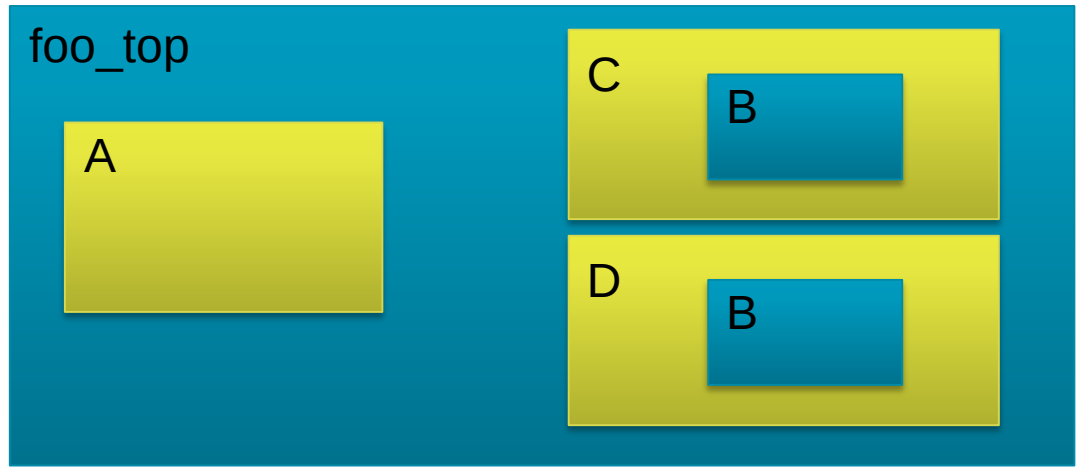
- **Each function is translated into an RTL block**
  - Verilog module, VHDL entity

Source Code

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


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

my_code.c

RTL hierarchy



Each function/block can be shared like any other component (add, sub, etc) provided it's not in use at the same time

  - By default, each function is implemented using a common instance
  - Functions may be inlined to dissolve their hierarchy
    - Small functions may be automatically inlined

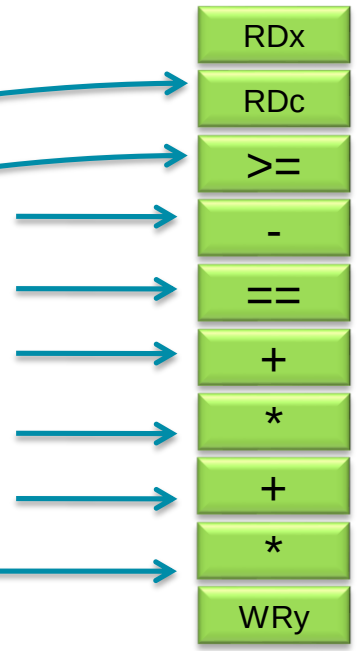
Types = Operator Bit-sizes

Code

```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
  ) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

  acc=0;
  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];
    }
  }
  *y=acc;
}
```

From any C code example ...

Operations



Operations are extracted…

Types

Standard C types

long long (64-bit)	short (16-bit)	unsigned types
int (32-bit)	char (8-bit)	
float (32-bit)	double (64-bit)	

Arbitary Precision types

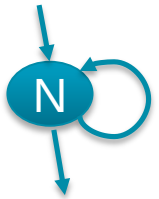
C:	ap(u)int types (1-1024)
C++:	ap_(u)int types (1-1024) ap_fixed types
C++/SystemC:	sc_(u)int types (1-1024) sc_fixed types

Can be used to define any variable to be a specific bit-width (e.g. 17-bit, 47-bit etc).

The C types define the size of the hardware used: handled automatically

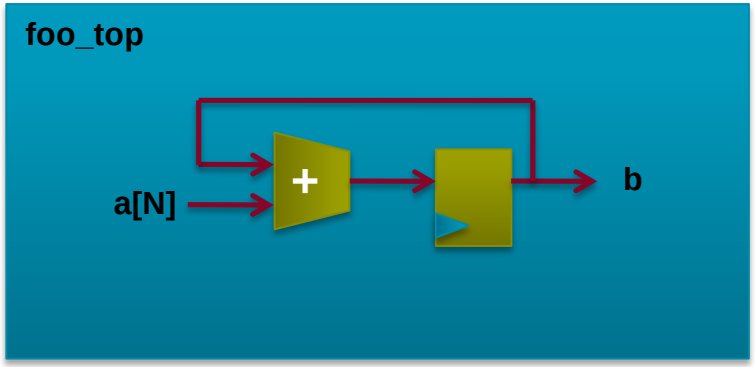
Loops

- **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;
  ...
  }
```

Synthesis



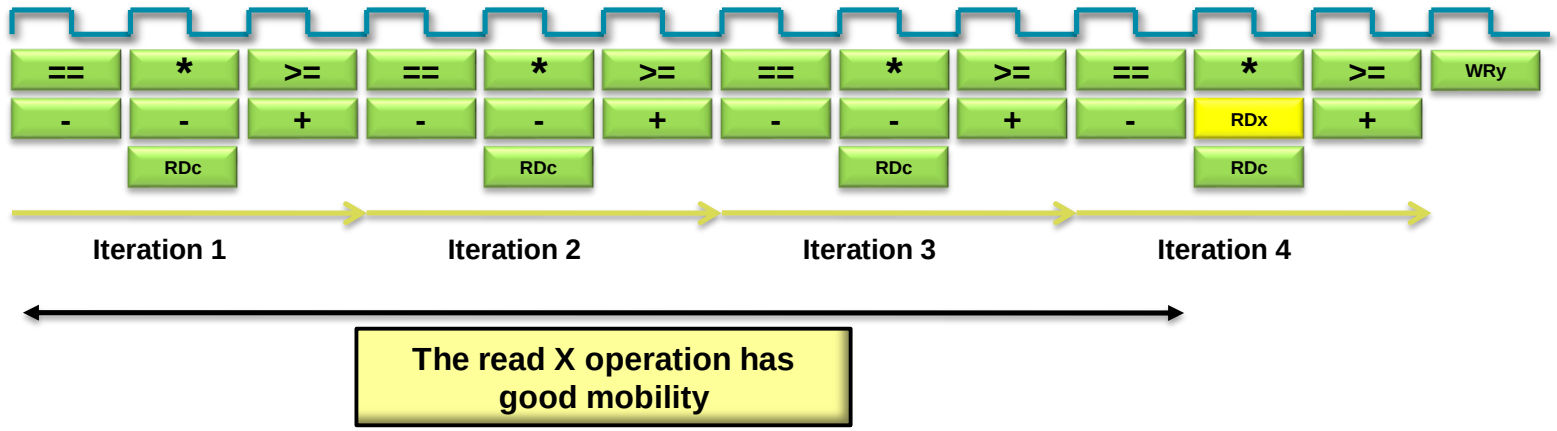
Loops require labels if they are to be referenced by Tcl directives
(GUI will auto-add labels)

  - Loops can be unrolled if their indices are statically determinable at elaboration time
    - Not when the number of iterations is variable
  - Unrolled loops result in more elements to schedule but greater operator mobility
    - Let's look at an example ….

Data Dependencies: Good

```
void fir (
 …
acc=0;
  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];
    }
  }
  *y=acc;
}
```

Default Schedule

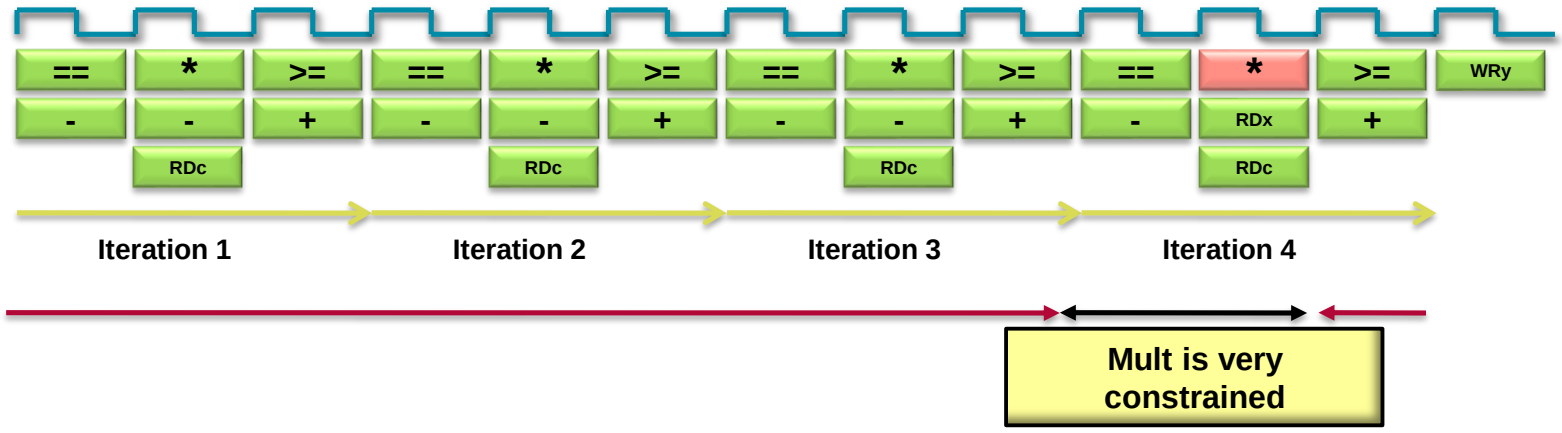


- **Example of good mobility**
  - The read on data port X can occur anywhere from the start to iteration 4
    - The only constraint on RDx is that it occur before the final multiplication
  - Vivado HLS has a lot of freedom with this operation
    - It waits until the read is required, saving a register
    - There are no advantages to reading any earlier (unless you want it registered)
    - Input reads can be optionally registered
  - The final multiplication is very constrained…

Data Dependencies: Bad

```
void fir (
 …
acc=0;
  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];
    }
  }
  *y=acc;
}
```

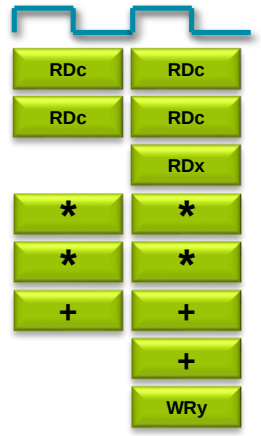
Default Schedule



- **Example of bad mobility**
  - The final multiplication must occur before the read and final addition
    - It could occur in the same cycle if timing allows
  - Loops are rolled by default
    - Each iteration cannot start till the previous iteration completes
    - The final multiplication (in iteration 4) must wait for earlier iterations to complete
  - The structure of the code is forcing a particular schedule
    - There is little mobility for most operations
  - Optimizations allow loops to be unrolled giving greater freedom

Schedule after Loop Optimization

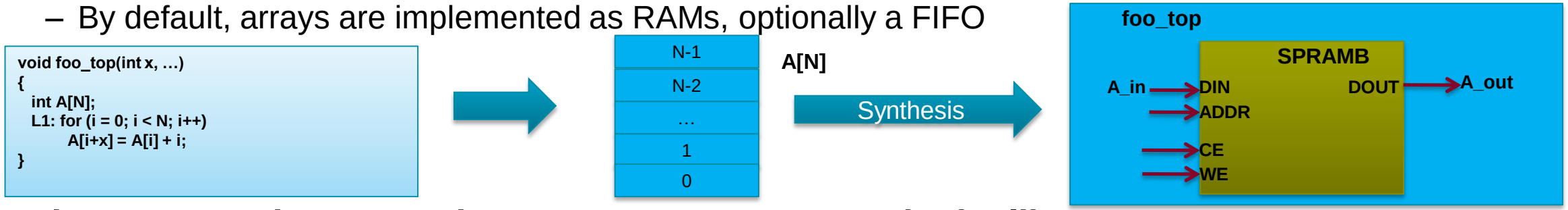
- **With the loop unrolled (completely)**
  - The dependency on loop iterations is gone
  - Operations can now occur in parallel
    - If data dependencies allow
    - If operator timing allows
  - Design finished faster but uses more operators
    - 2 multipliers & 2 Adders
- **Schedule Summary**
  - All the logic associated with the loop counters and index checking are now gone
  - Two multiplications can occur at the same time
    - All 4 could, but it's limited by the number of input reads (2) on coefficient port C
  - Why 2 reads on port C?
    - The default behavior for arrays now limits the schedule…




```
void fir (
 …
acc=0;
  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];
    }
  }
  *y=acc;
}
```

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



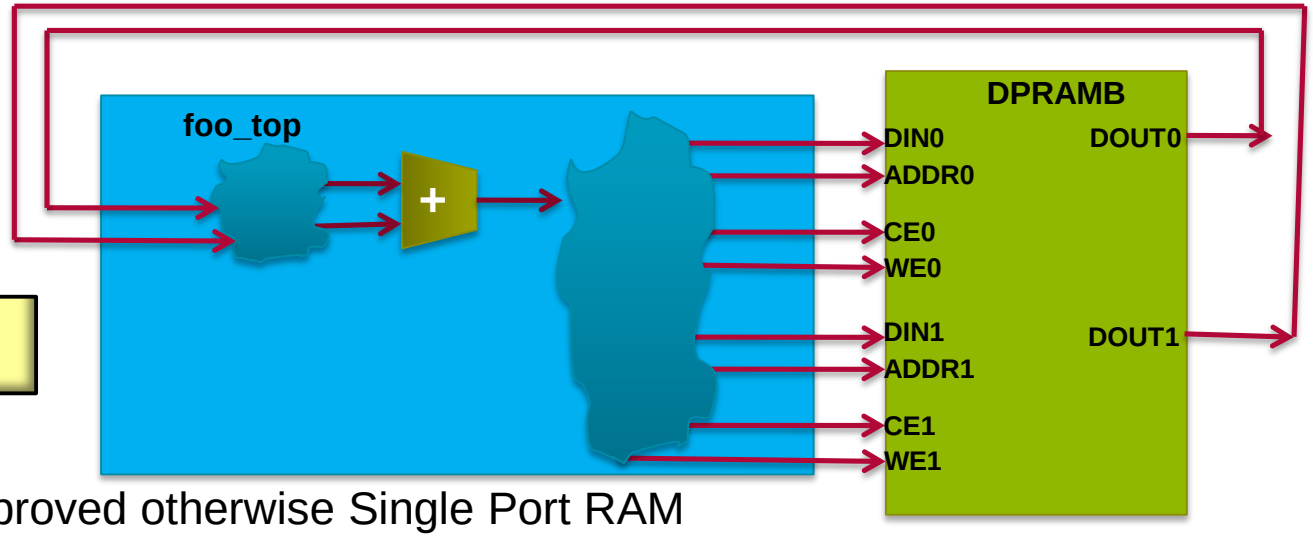
- **The array can be targeted to any memory resource in the library**
  - The ports (Address, CE active high, etc.) and sequential operation (clocks from address to data out) are defined by the library model
  - All RAMs are listed in the Vivado HLS Library Guide
- **Arrays can be merged with other arrays and reconfigured**
  - To implement them in the same memory or one of different widths & sizes
- **Arrays can be partitioned into individual elements**
  - Implemented as smaller RAMs or registers

Top-Level IO Ports

- **Top-level function arguments**
  - All top-level function arguments have a default hardware port type
- **When the array is an argument of the top-level function**
  - The array/RAM is “off-chip”
  - The type of memory resource determines the top-level IO ports
  - Arrays on the interface can be mapped & partitioned
    - E.g. partitioned into separate ports for each element in the array

```
void foo_top( int A[3*N] , int x)
{
  L1: for (i = 0; i < N; i++)
        A[i+x] = A[i] + i;
}
```

Synthesis



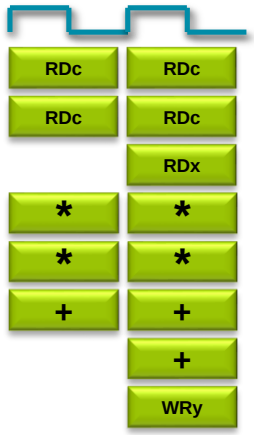
Number of ports defined by the RAM resource

- **Default RAM resource**
  - Dual port RAM if performance can be improved otherwise Single Port RAM

Schedule after an Array Optimization

- **With the existing code & defaults**
  - Port C is a dual port RAM
  - Allows 2 reads per clock cycles
    - IO behavior impacts performance

Note: It could have performed 2 reads in the original rolled design but there was no advantage since the rolled loop forced a single read per cycle

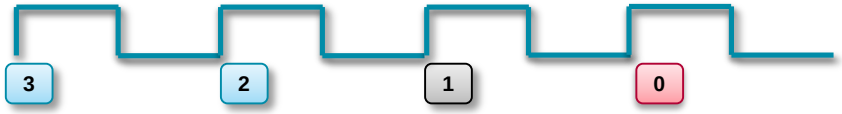



```
loop: for (i=3;i>=0;i--) {
  if (i==0) {
    acc+=x*c[0];
    shift_reg[0]=x;
  } else {
    shift_reg[i]=shift_reg[i-1];
    acc+=shift_reg[i]*c[i];
  }
}
*y=acc;
```

- **With the C port partitioned into (4) separate ports**
  - All reads and mults can occur in one cycle
  - If the timing allows
    - The additions can also occur in the same cycle
    - The write can be performed in the same cycles
    - Optionally the port reads and writes could be registered

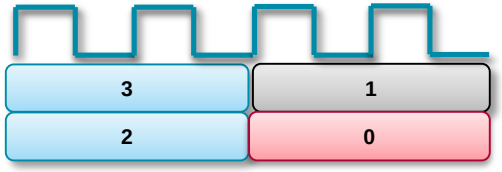
Operators

- **Operator sizes are defined by the type**
  - The variable type defines the size of the operator
- **Vivado HLS will try to minimize the number of operators**
  - By default Vivado HLS will seek to minimize area after constraints are satisfied
- **User can set specific limits & targets for the resources used**
  - Allocation can be controlled
    - An upper limit can be set on the number of operators or cores allocated for the design: This can be used to force sharing
    - e.g limit the number of multipliers to 1 will force Vivado HLS to share



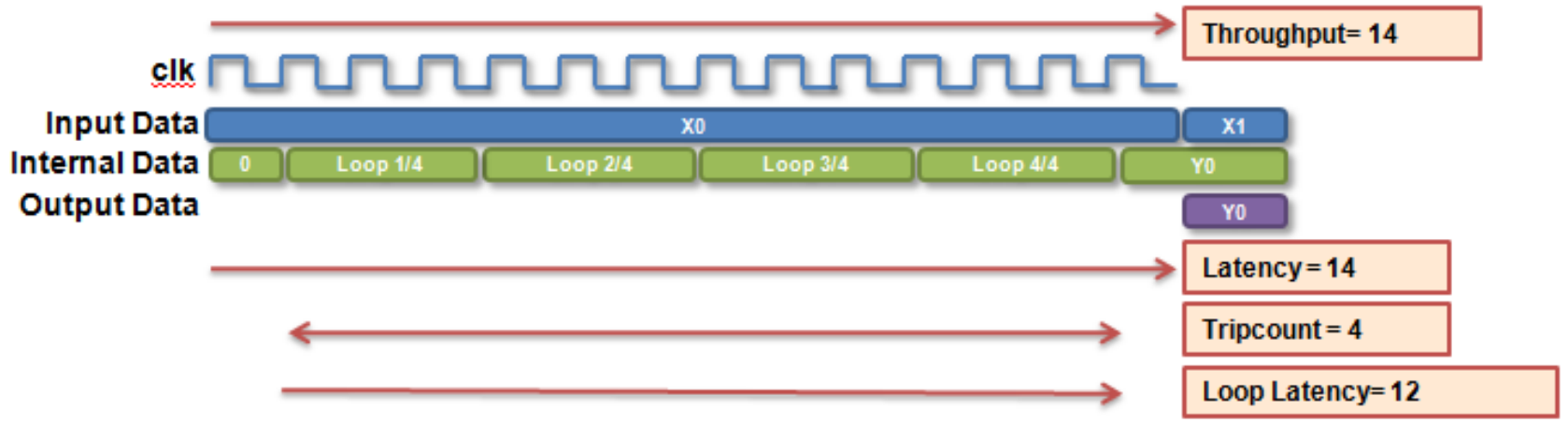
Use 1 mult, but take 4 cycle even if it could be done in 1 cycle using 4 mults

  - Resources can be specified
    - The cores used to implement each operator can be specified
    - e.g. Implement each multiplier using a 2 stage pipelined core (hardware)



Same 4 mult operations could be done with 2 pipelined mults (with allocation limiting the mults to 2)

Vivado HLS Terminology for measuring in Clock Cycles

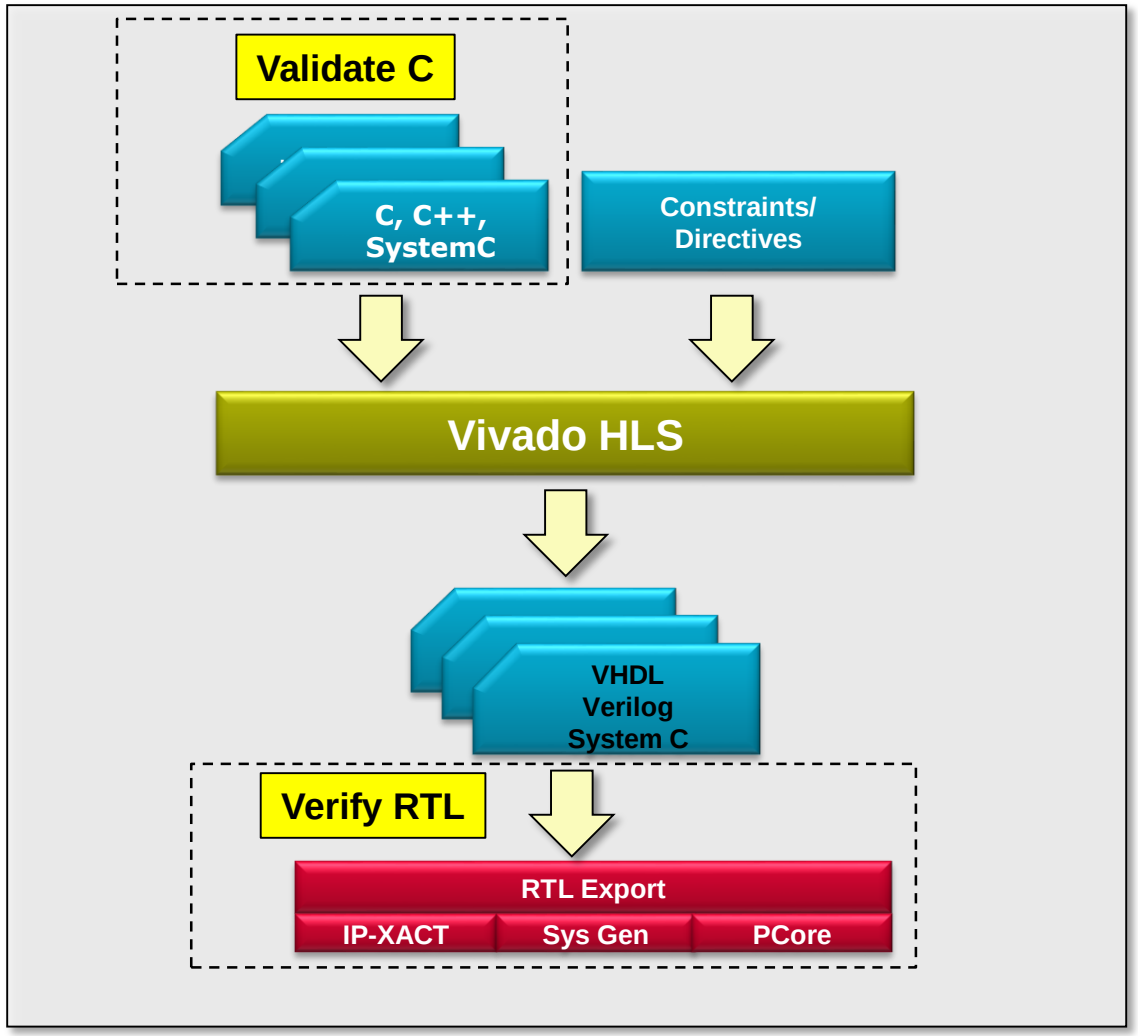


Latency	The number of cycles from input to output (final output of an array write)	14 cycles
Throughput	The number of cycle between new input samples (in this example it must wait for all operations to complete before it can read a new input)	14 cycles
Initiation Interval (II)	The number of cycles between new inputs to a pipeline (the same as throughput, but this term is used with pipelines).	Not shown in this example.
Data Rate	The 1/throughout * clock frequency	10ns clock => 7.14 Mhz, ((1/10e9)*14)
Trip count	The number of iterations in a loop	4
Loop Latency	The latency of the entire loop (divide by tripcount to get the latency for each loop iteration)	12 cycles

You may have your own terminology: this is AutoESL's

C Validation and RTL Verification

- **There are two steps to verifying the design**
  - Pre-synthesis: C ***Validation***
    - Validate the algorithm is correct
  - Post-synthesis: RTL ***Verification***
    - Verify the RTL is correct
- **C validation**
  - A **HUGE** reason users want to use HLS
    - Fast, free verification
  - Validate the algorithm is correct **before** synthesis
    - Follow the test bench tips given over
- **RTL Verification**
  - Vivado HLS can co-simulate the RTL with the original test bench



C Function Test Bench

- **The test bench is the level above the function**
  - The main() function is above the function to be synthesized
- **Good Practices**
  - The test bench should compare the results with golden data
    - Automatically confirms any changes to the C are validated and verifies the RTL is correct
  - The test bench should return a 0 if the self-checking is correct
    - Anything but a 0 (zero) will cause RTL verification to issue a FAIL message
    - Function main() should expect an integer return (non-void)

```
int main () {
  int ret=0;
  …
  ret = system("diff --brief  -w output.dat output.golden.dat");
  if (ret != 0) {
        printf("Test failed  !!!\n");
        ret=1;
  } else {
        printf("Test passed !\n");
  }
  …
  return ret;
}
```

Determine or Create the top-level function

- **Determine the top-level function for synthesis**
- **If there are Multiple functions, they must be merged**
  - There can only be 1 top-level function for synthesis

Given a case where functions func_A and func_B are to be implemented in FPGA

main.c

```
int main () {
        ...
        func_A(a,b,*i1);
        func_B(c,*i1,*i2);
        func_C(*i2,ret)

        return ret;
}
```

func_A
func_B
func_C

Re-partition the design to create a **new** single top-level function inside main()

main.c

```
#include func_AB.h
int main (a,b,c,d) {
        ...
        // func_A(a,b,i1);
        // func_B(c,i1,i2);
        func_AB (a,b,c, *i1, *i2);
        func_C(*i2,ret)

        return ret;
}
```

func_AB
func_C

func_AB.c

```
#include func_AB.h
func_AB(a,b,c, *i1, *i2) {
        ...
        func_A(a,b,*i1);
        func_B(c,*i1,*i2);
        …
}
```

func_A
func_B

Recommendation is to separate test bench and design files

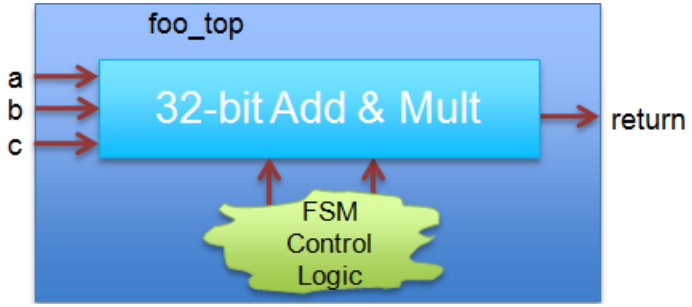
Data Types

Why is arbitrary precision Needed?

- **Code using native C int type**

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

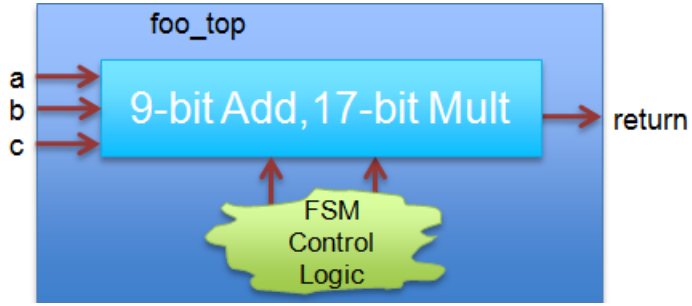
Synthesis



- **However, if the inputs will only have a max range of 8-bit**
  - Arbitrary precision data-types should be used

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

Synthesis



  - It will result in smaller & faster hardware with the full required precision
  - With arbitrary precision types on function interfaces, Vivado HLS can propagate the correct bit-widths throughout the design

HLS & C Types

- **There are 4 basic types you can use for HLS**
  - Standard C/C++ Types
  - Vivado HLS enhancements to C: apint
  - Vivado HLS enhancements to C++: ap_int, ap_fixed
  - SystemC types

Type of C	C(C99) / C++	Vivado HLS apint (bit-accurate with C)	Vivado HLS ap_int (bit-accurate with C++)	OSCI SystemC (IEEE 1666-2005 :bit-accurate)
Description		Used with standard C	Used with standard C++	IEEE standard
Requires		#include "ap_cint.h"	#include "ap_int.h" #include "ap_fixed.h" #include "ap_stream.h"	#include "systemc.h"
Pre-Synthesis Validation	gcc/g++	apcc	g++	g++
Fixed Point	NA	NA	ap_fixed	#define SC_INCLUDE_FX sc_fixed
Signal Modeling	Variables	Variables	Variables Streams	Signals, Channels, TLM (1.0)

Outline

- C and C++ Data Types
- *Arbitrary Precision Data Types*
- System C Data Types
- Floating Point Support
- Summary

Arbitrary Precision : C++ ap_int types

- **For C++**
  - Vivado HLS types ap_int can be used
  - Range: 1 to 1024 bits
    - Signed: ap_int<W>
    - Unsigned: ap_uint<W>
  - The bit-width is specified by W

```
#include ap_int.h

void foo_top (…) {

   ap_int<9>      var1;   // 9-bit
   ap_uint<10>    var2;   // 10-bit unsigned
```

Include header file

- **C++ compilation**
  - Use g++ at the Vivado HLS CLI (shell)
    - Include the path to the Vivado HLS header file

```
shell> g++ –o my_test test.c test_tb.c -I$VIVADO_HLS_HOME/include
```

AP_INT operators & conversions

Fully Supported for all Arithmetic operator

Operations	
Arithmetic	+ - * / % ++ --
Logical	~ !
Bitwise	& | ^
Relational	> < <= >= == !=
Assignment	*= /= %= += -= <<= >>= &= ^= |=

Methods for type conversion

Methods		Example
To integer	Convert to a integer type	res = var.to_int();
To unsigned integer	Convert to an unsigned integer type	res = var.to_uint();
To 64-bit integer	Convert to a 64-bit long long type	res = var.to_int64();
To 64-bit unsigned integer	Convert to an unsigned long long type	res = var.to_uint64();
To double	Convert to double type	res = var.double();

AP_INT Bit Manipulation methods

Methods		Example
Length	Returns the length of the variable.	res=var.length;
Concatenation	Concatenation low to high	res=var_hi.concat(var_lo); Or res= (var_hi,var_lo)
Range or Bit-select	Return a bit-range from high to low or a specific bit.	res=var.range(high bit,low bit); Or res=var[bit-number]
(n)and_reduce	(N)And reduce all bits.	bool t = var.and_reduce();
(n)or_reduce	(N)Or reduce all bits	bool t = var.or_reduce();
X(n)or_reduce	X(N)or reduce all bits	bool t = var.xor_reduce();
Reverse	Reserve the bits in the variable	var.reverse();
Test bit	Tests if a bit is true	bool t = var.test(bit-number)
Set bit value	Sets the value of a specific bit	var.set_bit(bit-number, value)
Set bit	Set a specific bit to one	var.set(bit-number);
Clear bit	Clear a specific bit to zero	var.clear(bit-number);
Invert Bit	Invert a specific bit	var.invert(bit-number);
Rotate right	Rotate the N-bits to the right	var.rrotate(N);
Rotate left	Rotate the N-bits to the left	var.lrotate(N);
Bitwise Invert	Invert all bits	var.b_not();
Test sign	Test if the sign is negative (return true)	bool t = var.sign();

Arbitrary Precision : C++ ap_fixed types

- **Support for fixed point datatypes in C++**
  - Include the path to the ap_fixed.h header file
  - Both signed (ap_fixed) and unsigned types (ap_ufixed)

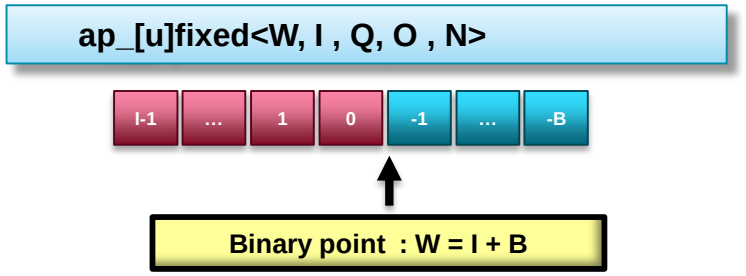
```
#include ap_fixed.h        $VIVADO_HLS_HOME/include/ap_fixed.h

void foo_top (…) {

 ap_fixed<9, 5, AP_RND_CONV, AP_SAT>  var1;        // 9-bit,
                                                   // 5 integer  bits, 4 decimal places

 ap_ufixed<10, 7, AP_RND_CONV, AP_SAT>  var2;      // 10-bit unsigned
                                                   //  7 integer bits, 3 decimal places
```

- **Advantages of Fixed Point types**
  - The result of variables with different sizes is automatically taken care of
  - The binary point is automatically aligned
    - Quantization: Underflow is automatically handled
    - Overflow: Saturation is automatically handled

Alternatively, make the result variable large enough such that overflow or underflow does not occur

Definition of ap_fixed type

- **Fixed point types are specified by**
  - Total bit width (W)
  - The number of integer bits (I)
  - The quantization/rounding mode (Q)
  - The overflow/saturation mode (O)



	Description	
W	Word length in bits	
I	The number of bits used to represent the integer value (the number of bits above the decimal point)	
Q	Quantization mode (modes detailed below) dictates the behavior when greater precision is generated than can be defined by the LSBs.	
	AP_Fixed Mode	**Description**
	AP_RND	Rounding to plus infinity
	AP_RND_ZERO	Rounding to zero
	AP_RND_MIN_INF	Rounding to minus infinity
	AP_RND_INF	Rounding to infinity
	AP_RND_CONV	Convergent rounding
	AP_TRN	Truncation to minus infinity
	AP_TRN_ZERO	Truncation to zero (default)
O	Overflow mode (modes detailed below) dictates the behavior when more bits are required than the word contains.	
	AP_Fixed Mode	**Description**
	AP_SAT	Saturation
	AP_SAT_ZERO	Saturation to zero
	AP_SAT_SYM	Symmetrical saturation
	AP_WRAP	Wrap around (default)
	AP_WRAP_SM	Sign magnitude wrap around
N	The number of saturation bits in wrap modes.	

Quantization Modes

- **Quantization mode**
  - Determines the behavior when an operation generates more precision in the LSBs than is available
- **Quantization Modes (rounding):**
  - AP_RND, AP_RND_MIN_IF, AP_RND_IF
  - AP_RND_ZERO, AP_RND_CONV
- **Quantization Modes (truncation):**
  - AP_TRN, AP_TRN_ZERO

Quantization Modes: Rounding

- **AP_RND_ZERO: rounding to zero**
  - For positive numbers, the redundant bits are truncated
  - For negative numbers, add MSB of removed bits to the remaining bits.
  - The effect is to round towards zero.
    - 01.01 (1.25 using 4 bits) rounds to 01.0 (1 using 3 bits)
    - 10.11 (-1.25 using 4 bits) rounds to 11.0 (-1 using 3 bits)

- **AP_RND_CONV: rounded to the nearest value**
  - The rounding depends on the least significant bit
  - If the least significant bit is set, rounding towards plus infinity
  - Otherwise, rounding towards minus infinity
    - 00.11 (0.75 using 4-bit) rounds to 01.0 (1.0 using 3-bit)
    - 10.11 (-1.25 using 4-bit) rounds to 11.0 (-1.0 using 3-bit)

Quantization Modes: Truncation

- **AP_TRN: truncate**
  - Remove redundant bits. Always rounds to minus infinity
  - This is the default.
    - 01.01(1.25) ➔ 01.0 (1)

- **AP_TRN_ZERO: truncate to zero**
  - For positive numbers, the same as AP_TRN
    - For positive numbers: 01.01(1.25) ➔ 01.0(1)
  - For negative numbers, round to zero
    - For negative numbers: 10.11 (-1.25) ➔ 11.0(-1)

Overflow Modes

- **Overflow mode**
  - Determines the behavior when an operation generates more bits than can be satisfied by the MSB
- **Overflow Modes (saturation)**
  - AP_SAT, AP_SAT_ZERO, AP_SAT_SYM
- **Overflow Modes (wrap)**
  - AP_WRAP, AP_WRAP_SM
  - The number of saturation bits, N, is considered when wrapping

Overflow Mode: Saturation

- **AP_SAT: saturation**
  - This overflow mode will convert the specified value to MAX for an overflow or MIN for an underflow condition
  - MAX and MIN are determined from the number of bits available
- **AP_SAT_ZERO: saturates to zero**
  - Will set the result to zero, if the result is out of range
- **AP_SAT_SYM: symmetrical saturation**
  - In 2's complement notation one more negative value than positive value can be represented
  - If it is desirable to have the absolute values of MIN and MAX symmetrical around zero, AP_SAT_SYM can be used
  - Positive overflow will generate MAX and negative overflow will generate -MAX
    - 0110(6) => 011(3)
    - 1011(-5) => 101(-3)

Overflow Mode: Wrap

- **AP_WRAP, N = 0**
  - Simple wrapping, keeping the LSBs
    - ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 19.0; Result: 1_0011 → 0011 (3.0)
    - ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = 31.0; Result: 1_1111 → 1111 (-1.0)
    - ap_ufixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0; Result: 10_1101 → 1101 (13.0)
    - ap_fixed<4, 4, AP_RND, AP_WRAP> UAPFixed4 = -19.0; Result: 10_1101 → 1101 (-3.0)
- **AP_WRAP, N = 1**
  - Behaves similar to case where N = 0, except that positive numbers stay positive and negative numbers stay negative (keeps original „sign bit”)

AP_FIXED operators & conversions

Fully Supported for all Arithmetic operator

Operations	
Arithmetic	+ - * / % ++ --
Logical	~ !
Bitwise	& | ^
Relational	> < <= >= == !=
Assignment	*= /= %= += -= <<= >>= &= ^= |=

Methods for type conversion

Methods		Example
To integer	Convert to a integer type	res = var.to_int();
To unsigned integer	Convert to an unsigned integer type	res = var.to_uint();
To 64-bit integer	Convert to a 64-bit long long type	res = var.to_int64();
To 64-bit unsigned integer	Convert to an unsigned long long type	res = var.to_uint64();
To double	Convert to double type	res = var.double();
To ap_int	Convert to an ap_int	res = var.to_ap_int();

AP_FIXED methods

Methods for bit manipulation

Methods		Example
Length	Returns the length of the variable.	res=var.length;
Concatenation	Concatenation low to high	res=var_hi.concat(var_lo); Or res= (var_hi,var_lo)
Range or Bit-select	Return a bit-range from high to low or a specific bit.	res=var.range(high bit,low bit); Or res=var[bit-number]

Floating Point Support

- **Synthesis for floating point**
  - Data types (IEEE-754 standard compliant)
    - Single-precision
      - 32 bit: 24-bit fraction, 8-bit exponent
    - Double-precision
      - 64 bit: 53-bit fraction, 11-bit exponent
- **Support for Operators**
  - Vivado HLS supports the Floating Point (FP) cores for each Xilinx technology
    - If Xilinx has a FP core, Vivado HLS supports it
    - It will automatically be synthesized
  - If there is no such FP core in the Xilinx technology, it will not be in the library
    - The design will be still synthesized

Floating Point Cores

Core	7-Series	Virtex 6	Virtex 5	Virtex 4	Spartan 6	Spartan 3
FAddSub	X	X	X	X	X	X
FAddSub_nodsp	X	X	X	-	-	-
FAddSub_fulldsp	X	X	X	-	-	-
FCmp	X	X	X	X	X	X
FDiv	X	X	X	X	X	X
FMul	X	X	X	X	X	X
FMul_nodsp	X	X	X	-	X	X
FMul_meddsp	X	X	X	-	X	X
FMul_fulldsp	X	X	X	-	X	X
FMul_maxdsp	X	X	X	-	X	X
FRSqrt	X	X	X	-	-	-
FRSqrt_nodsp	X	X	X	-	-	-
FRSqrt_fulldsp	X	X	X	-	-	-
FRecip	X	X	X	-	-	-
FRecip_nodsp	X	X	X	-	-	-
FRecip_fulldsp	X	X	X	-	-	-
FSqrt	X	X	X	X	X	X
DAddSub	X	X	X	X	X	X
DAddSub_nodsp	X	X	X	-	-	-
DAddSub_fulldsp	X	X	X	-	-	-
DCmp	X	X	X	X	X	X
DDiv	X	X	X	X	X	X
DMul	X	X	X	X	X	X
DMul_nodsp	X	X	X	-	X	X
DMul_meddsp	X	X	X	-	-	-
DMul_fulldsp	X	X	X	-	X	X
DMul_maxdsp	X	X	X	-	X	X
DRSqrt	X	X	X	X	X	X
DRecip	X	X	X	-	-	-
DSqrt	X	X	X	-	-	-

Support for Math Functions

- **Vivado HLS provides support for many math functions**
  - Even if no floating-point core exists
  - These functions are implemented in a bit-approximate manner
  - The results may differ within a few Units of Least Precision (ULP) to the C/C++ standards
- **Use math.h (C) or cmath.h (C++)**
  - The functions will be synthesized automatically
  - The C simulation results may differ from the RTL simulation results
  - Use a test bench which checks for ranges: not == or !=
- **Replace math.h or cmath.h with VHLS header file “hls_math.h” Or keep math/cmath and “add_files hls_lib.c”**
  - The C simulation will match the RTL simulation
  - The C simulation may differ from the C simulation using math/cmath (or math/cmath without hls_lib.c)

Supported Math Functions

- **Floating C point functions ***f**
  - There is no double-precision implementation
  - C++ functions will overload as per the C++ standard
    - Can be used with double or single precision
- **More specific details are in the User Guide**
  - Refer to the Coding Style Guide chapter: C Libraries

Function	Float	Double	Accuracy (ULP)	LogicCore
ceilf	Supported	Not Applicable	Exact	Not Supported
copysignf	Supported	Not Applicable	Exact	Not Supported
fabsf	Supported	Not Applicable	Exact	Not Supported
floorf	Supported	Not Applicable	Exact	Not Supported
logf	Supported	Not Applicable	1 to 5	Not Supported
cosf	Supported	Not Applicable	1 to 100	Not Supported
sinf	Supported	Not Applicable	1 to 100	Not Supported
abs	Supported	Supported	Exact	Not Supported
ceil	Supported	Supported	Exact	Not Supported
copysign	Supported	Supported	Exact	Not Supported
cos	Supported	Supported	2 for float. 5 for double	Not Supported
fabs	Supported	Supported	Exact	Not Supported
floor	Supported	Supported	Exact	Not Supported
fpclassify	Supported	Supported	Exact	Not Supported
isfinite	Supported	Supported	Exact	Not Supported
isinf	Supported	Supported	Exact	Not Supported
isnan	Supported	Supported	Exact	Not Supported
isnormal	Supported	Supported	Exact	Not Supported
log	Supported	Supported	1 for float, 16 for double	Not Supported
log10	Supported	Supported	1 for float, 16 for double	Not Supported
recip	Supported	Supported	Exact	Supported
round	Supported	Supported	Exact	Not Supported
rsqrt	Supported	Supported	Exact	Supported
signbit	Supported	Supported	Exact	Not Supported
sin	Supported	Supported	2 for float, 5 for double	Not Supported
sqrt	Supported	Supported	Exact	Supported
trunc	Supported	Supported	Exact	Not Supported

For more information on floating point refer to Application Note **Floating Point Design with Vivado HLS**

Example on using Floating Point Types

- **The following highlights some typical use scenarios**
  - Example values

```
double              foo_d   = 3.1459;
float               foo_f   = 3.1459;
ap_fixed<14,4>      foo_fx  =  -1.4142;
int                 foo_i   = 42;
```

Using ap_fixed requires:
• C++
• $Vivado HLS_HOME/include/ap_fixed.h

- **When using sqrt() function**
  - It is from math.h which is a C function, not C++

```
extern "C" float sqrtf(float);
```

Required if it's a C++ function

- **Understand that sqrt() is 64-bit and sqrtf() is 32-bit**

```
double      var_d = sqrt(foo_d);      // 64-bit sqrt core
float       var_f = sqrtf(foo_f);     // This will lead to a single precision sqrt core

            var_f = sqrt(foo_f);      // Still 64-bit, with format conversion cores (single to double and back)
```

- **Type conversions can be used**

```
ap_fixed<14,4>      var_fx = sqrtf(foo_fx);   // fixed-point to single precision conversion
                                              // Fixed → 32-bit sqrt core → float to fixed conversion
int                 var_i = sqrtf(foo_i);     // int to float conversion
                                              // Int → 32-bit sqrt → float to int
```

Using sqrt instead of sqrtf would imply a single to double conversion and back

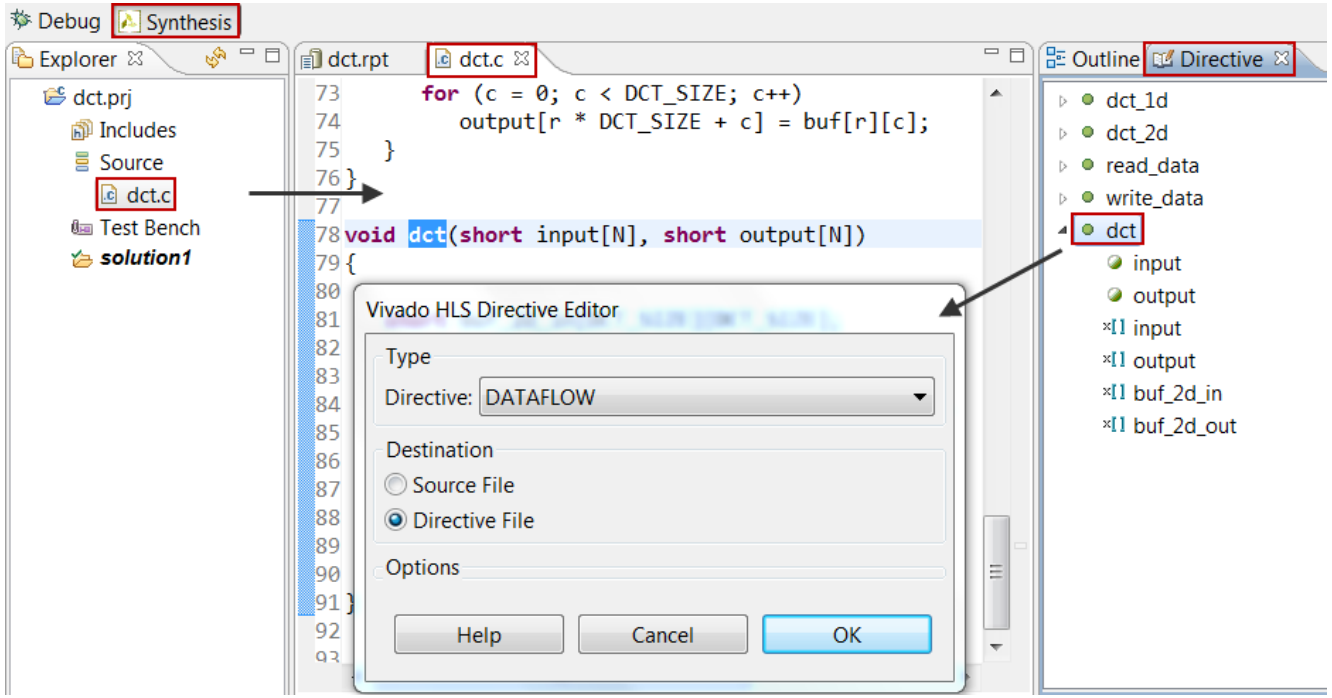
Improving Performance

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

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 this 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

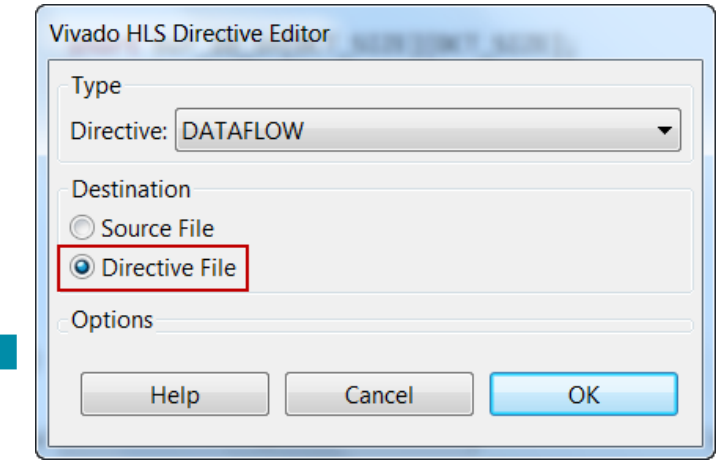


Optimization Directives: Tcl or Pragma

- **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)

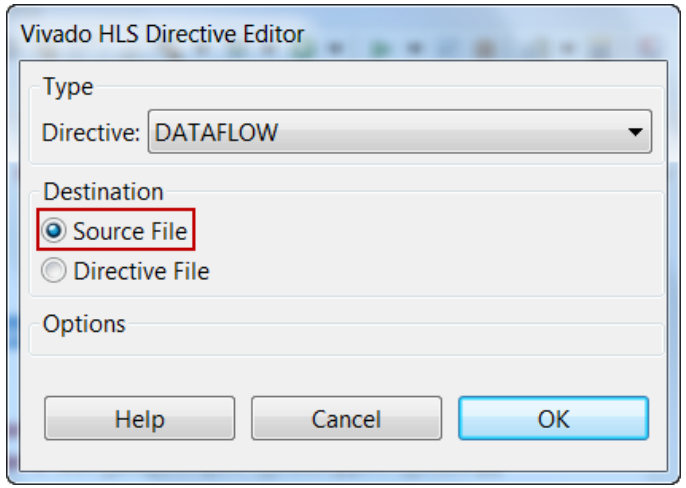
dct
% HLS DATAFLOW



- **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

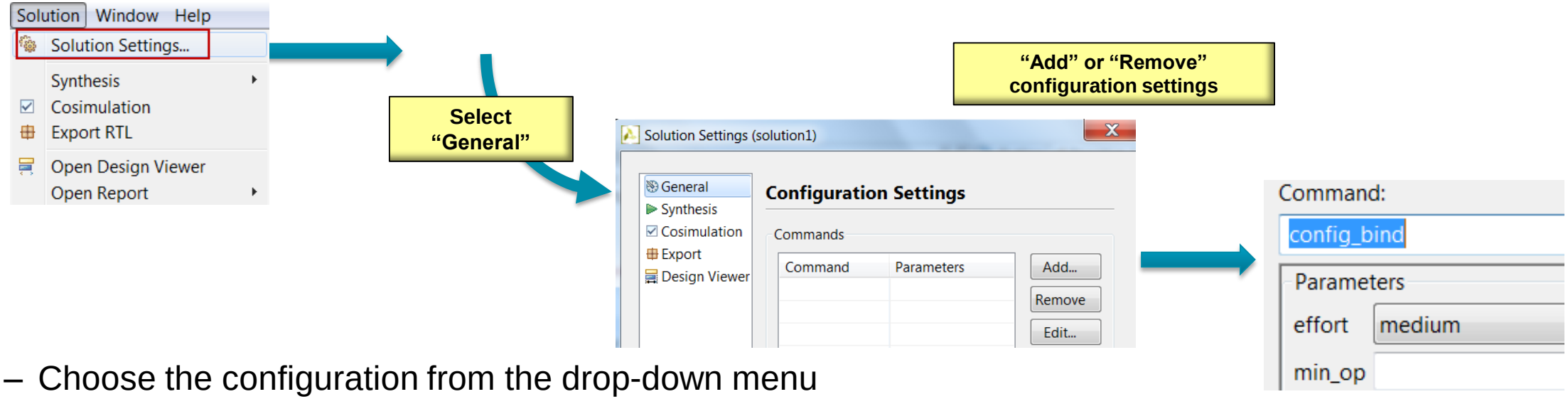
dct
HLS DATAFLOW

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



Solution Configurations

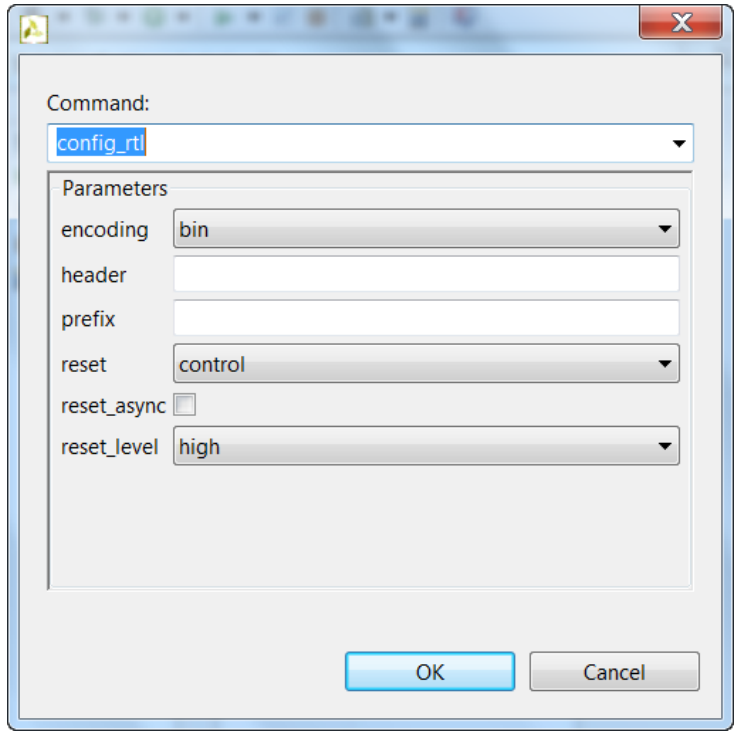
- **Configurations can be set on a solution**
  - Set the default behavior for that solution
    - Open configurations settings from the menu (Solutions > Solution Settings…)



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

Example: Configuring the RTL Output

- **Specify the FSM encoding style**
  - By default the FSM is binary
- **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

Outline

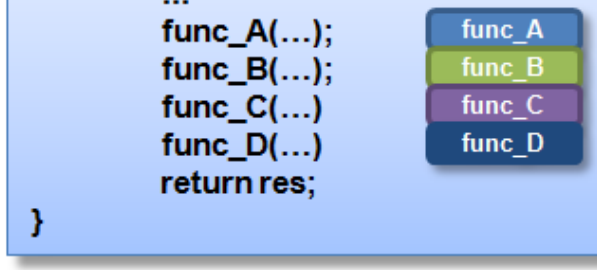
- Adding Directives
- *Improving Latency*
- Improving Throughput
- Performance Bottleneck
- Summary

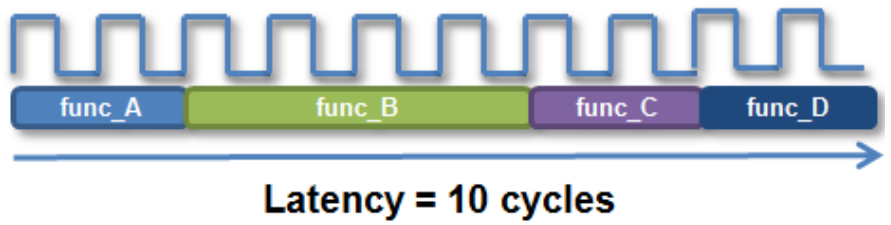
Latency and Throughput

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

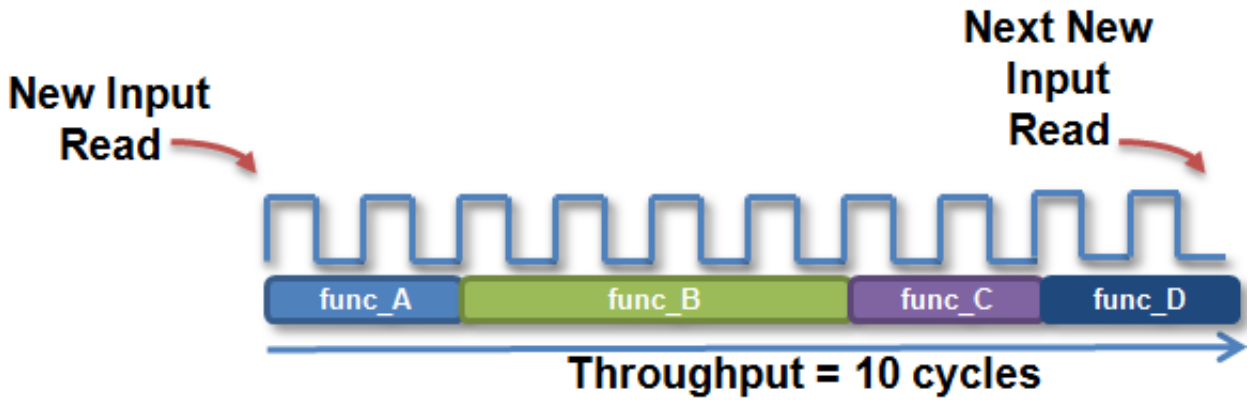
```
void foo_top (a,b,c,d, *x, *y) {
        ...
        func_A(...);
        func_B(...);
        func_C(...)
        func_D(...)
        return res;
}
```





Design Throughput

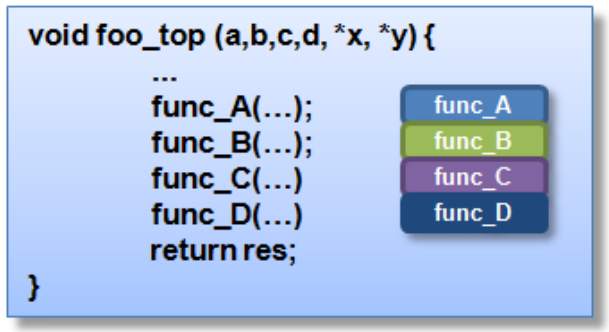
- The throughput of the design is the number of cycles between new inputs
  - By default (no concurrency) this is the same as latency
  - Next start/read is when this transaction ends

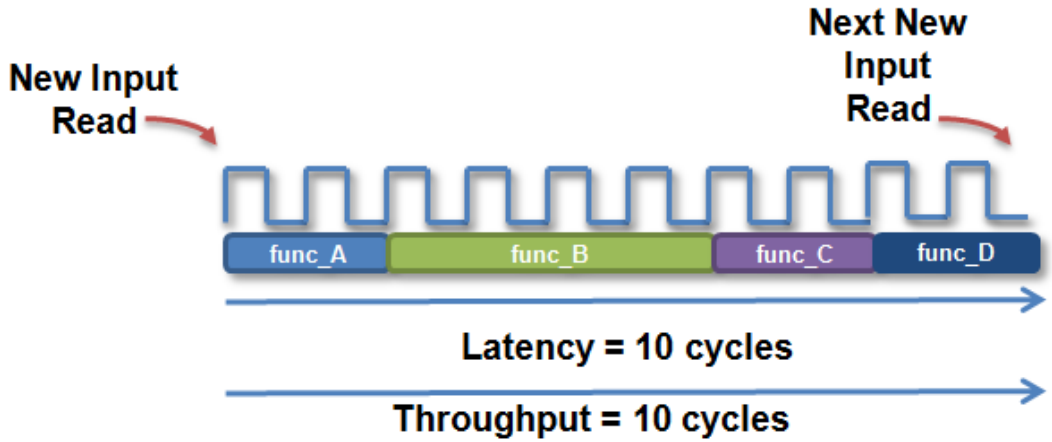


Latency and Throughput

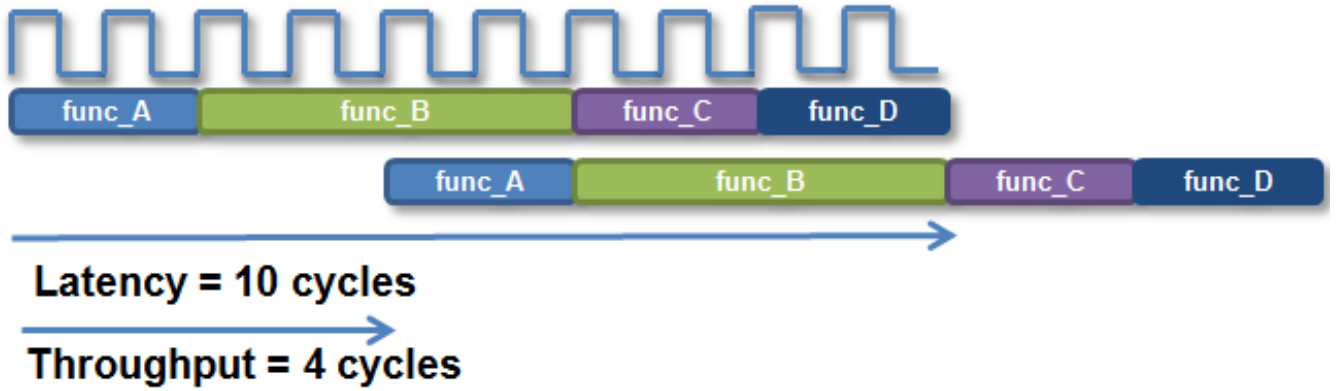
- **In the absence of any concurrency**
  - Latency is the same as throughput

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





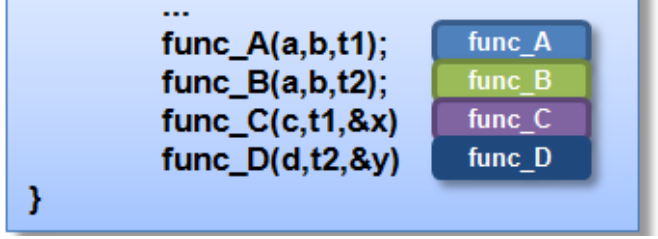
- **Pipelining for higher throughput**
  - Vivado HLS can pipeline functions and loops to improve throughput
  - Latency and throughput are related
  - We will discuss optimizing for latency first, then throughput

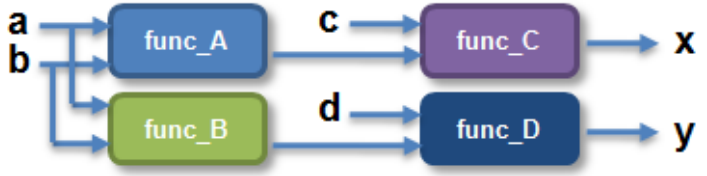


Vivado HLS: Minimize latency

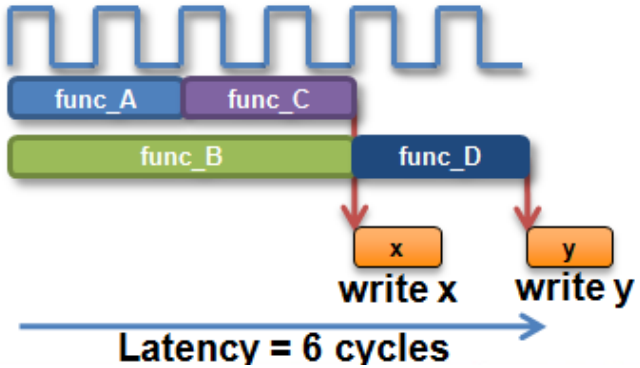
- **Vivado HLS will by default minimize latency**
  - Throughput is prioritized above latency (no throughput directive is specified here)
  - In this example
    - The functions are connected as shown
    - Function B takes longer than any other functions

```
void foo_top (a,b,c,d, *x, *y) {
    ...
    func_A(a,b,t1);
    func_B(a,b,t2);
    func_C(c,t1,&x)
    func_D(d,t2,&y)
}
```





- **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

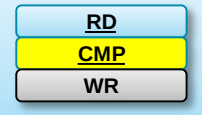


- func_A and func_B can start at the same time
- func_C can start as soon as func_A completes
- func_D must wait for func_B to complete
- Outputs are written as soon as they are ready

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

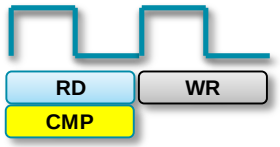
```
Loop:for(i=1;i<3;i++) {
  op_Read;
  op_Compute;
  op_Write;
}
```




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

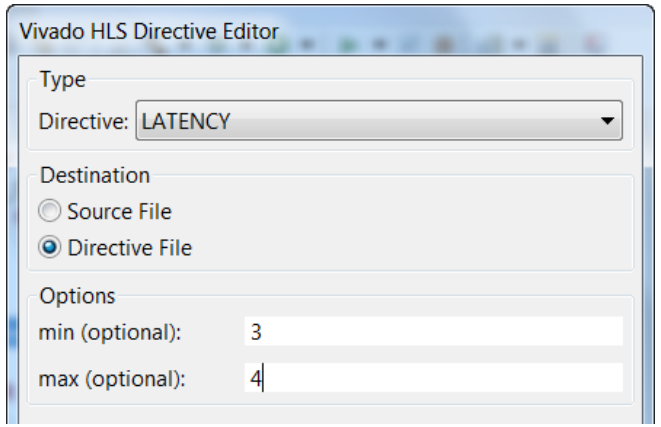
Example with Sequential Operations

Example of Minimizing latency with Parallel Operations

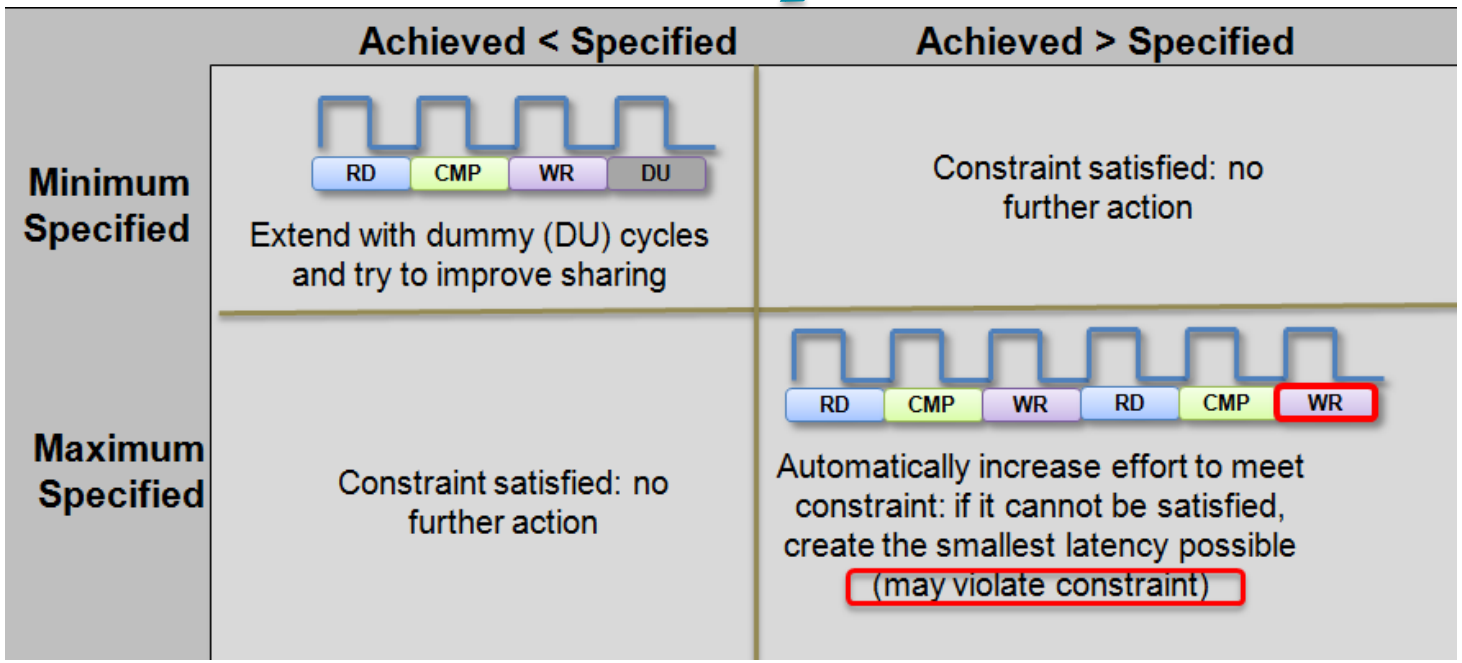


Latency Constraints

- **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

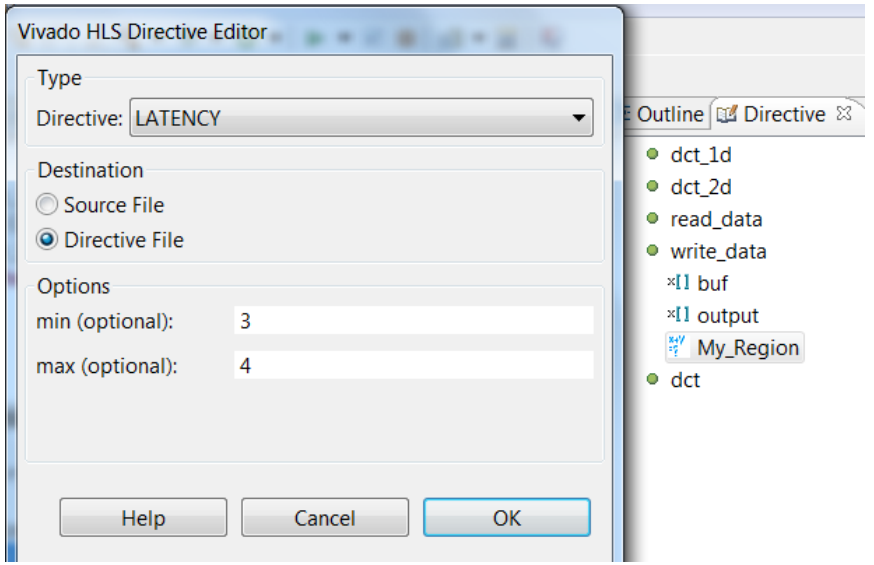


Regions: Specific Latency Constraints

- **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

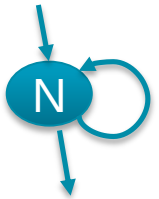
```
int write_data (int buf, int output) {
 if (x < y)  {
       return (x + y);
 } else  {
       My_Region: {
            return (y – x) * (y + x);
       }
}
```

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



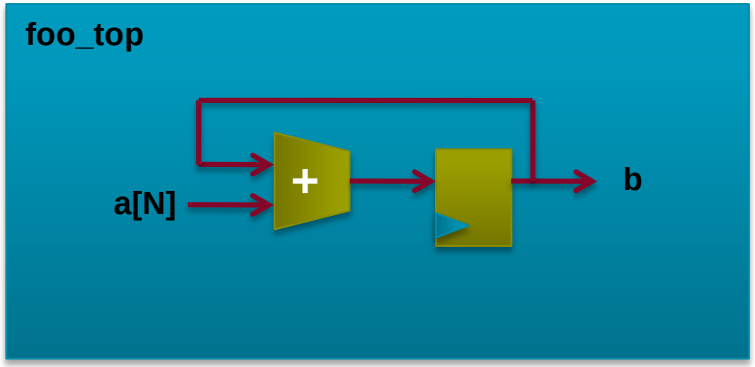
Review: Loops

- **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;
  ...
  }
```

Synthesis



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

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

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

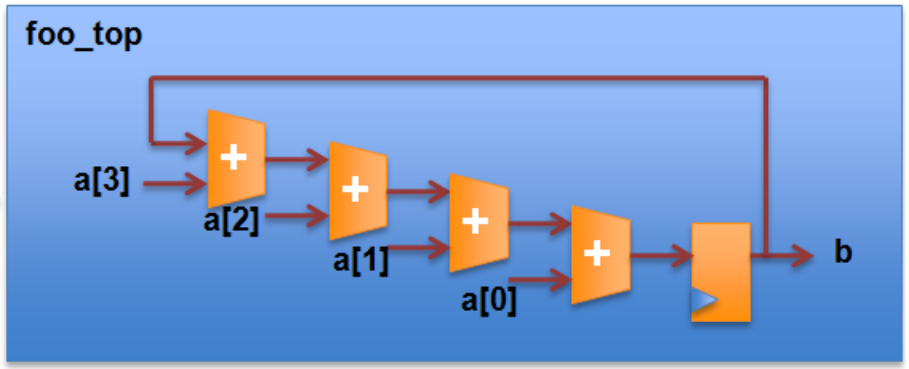
```
void foo_top (…) {
  ...
  Add: for (i=3;i>=0;i--) {
      b = a[i] + b;
  ...
  }
```

  - 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

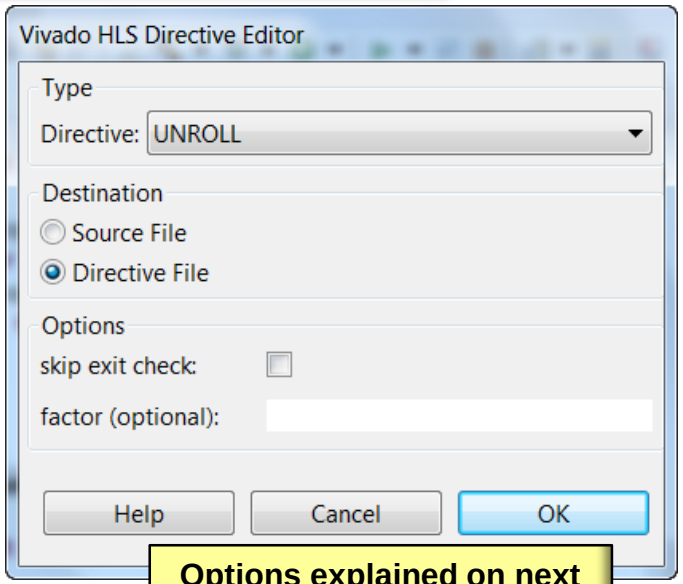
Unrolled Loops can Reduce Latency

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

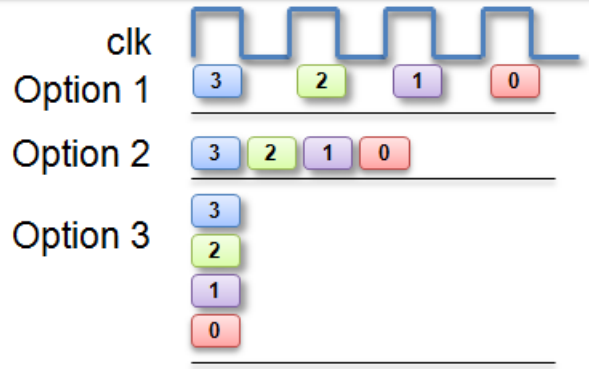
Unrolled



Select loop “Add” in the directives pane and right-click



Options explained on next slide



Unrolled loops allow greater option & exploration

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

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 unroll by a factor 2, 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 remove the exit check (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];
}
```

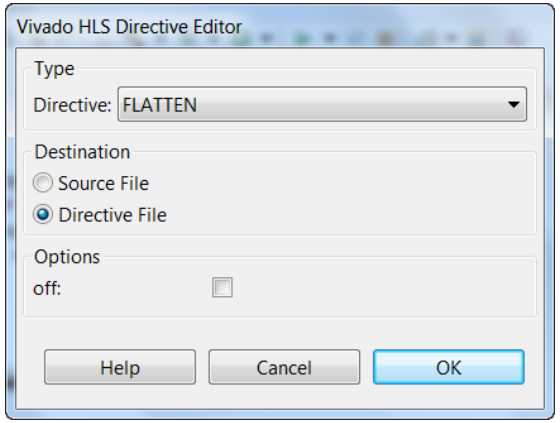
Effective code after compiler transformation

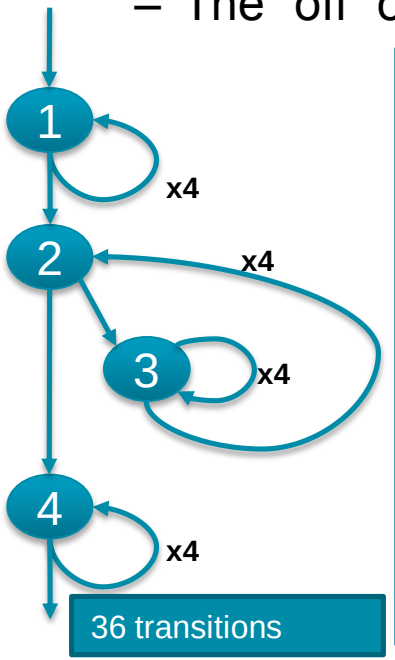
```
for(int i = 0; i < N; i += 2) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
}
```

An extra adder for N/2 cycles trade-off

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

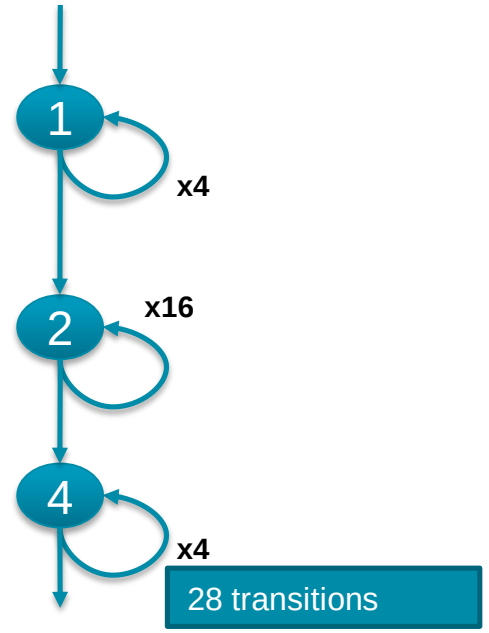





```
void foo_top (…) {
  ...
  L1: for (i=3;i>=0;i--) {
      [loop body L1 ]
  }

  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



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

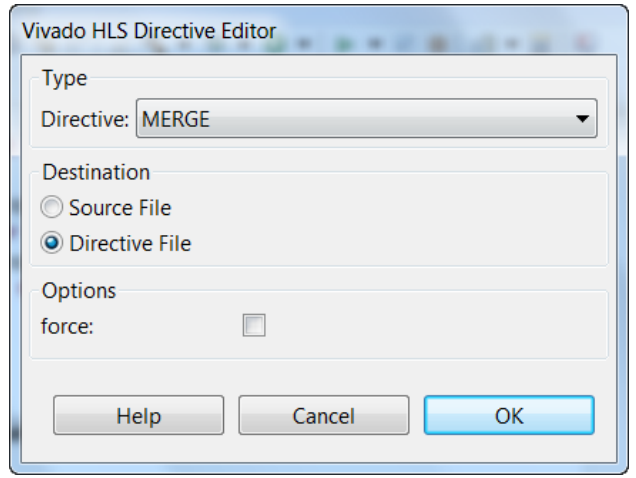
```
Loop_outer: for (i=3;i>=0;i--) {
   Loop_inner: for (j=3;j>=0;j--) {
       [loop body]
   }
}
```

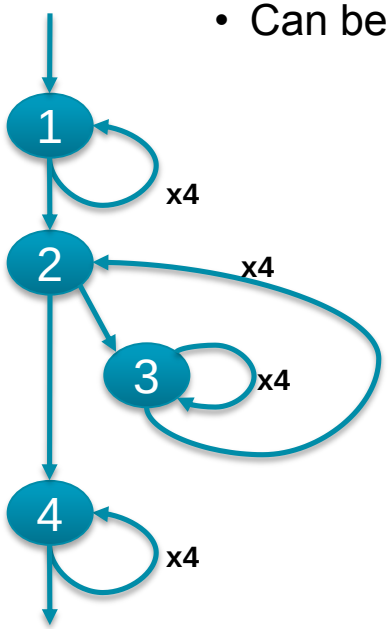
```
Loop_outer: for (i=3;i>N;i--) {
   Loop_inner: for (j=3;j>=0;j--) {
       [loop body]
   }
}
```

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

Loop Merging

- **Vivado HLS can automatically merge loops**
  - A faster approach than manually changing the code
  - Allows for more efficient architecture explorations
  - FIFO reads, which must occur in strict order, can prevent loop merging
    - Can be done with the “force” option : user takes responsibility for correctness



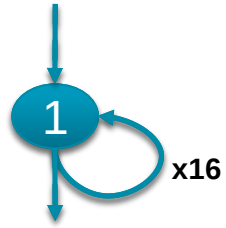


36 transitions

```
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 ]
  }
```

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 ]
  }
```



18 transitions

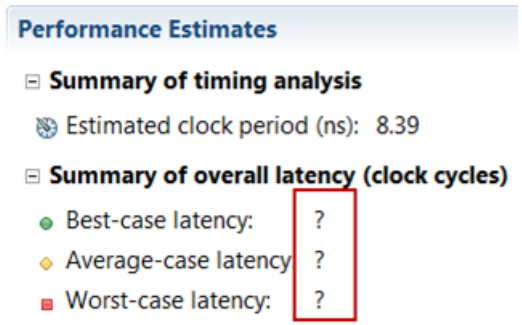
Loop Merge Rules

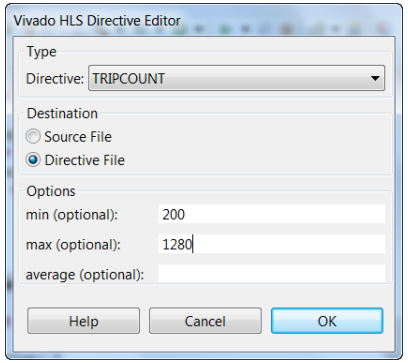
- **If loop bounds are all variables, they must have the same value**
- **If loops bounds are constants, the maximum constant value is used as the bound of the merged loop**
  - As in the previous example where the maximum loop bounds become 16 (implied by L3 flattened into L2 before the merge)
- **Loops 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

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





Outline

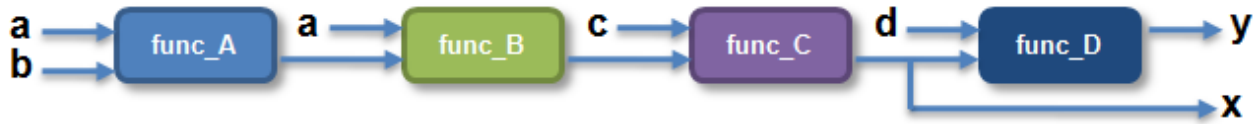
- Adding Directives
- Improving Latency
- *Improving Throughput*
- Performance Bottleneck
- Summary

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_B(a,t1,t2);
        func_C(c,t2,&x)
        func_D(d,x,&y)
}
```

func_A func_B func_C func_D

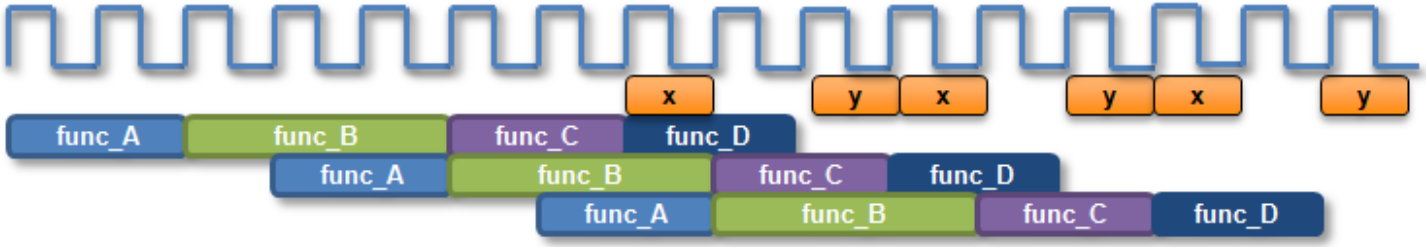


- **Vivado HLS will schedule the design**

The latency is 9 cycles

The throughput is also 9 cycles

- **It can also automatically optimize the dataflow for throughput**

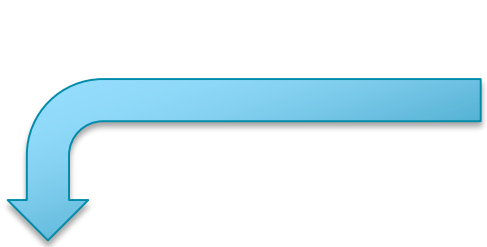


The latency is still 9 cycles

The throughput is now 3

Function Pipelining

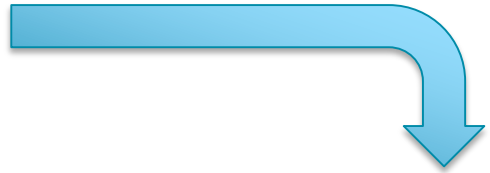
Without Pipelining

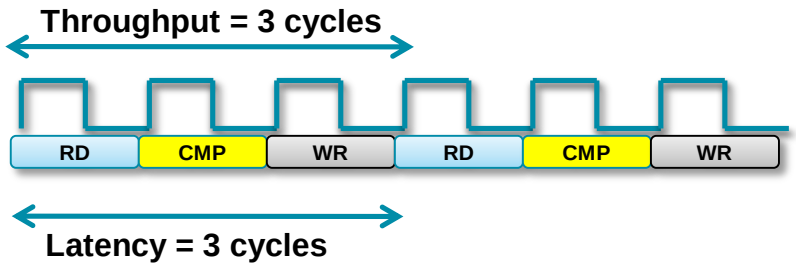


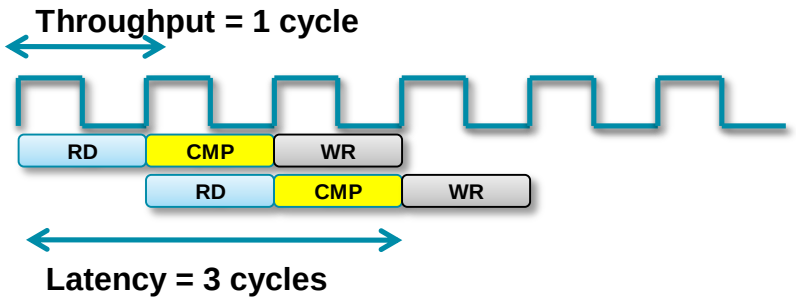
```
void foo(...) {
   op_Read;
   op_Compute;
   op_Write;
}
```

RD
CMP
WR

With Pipelining



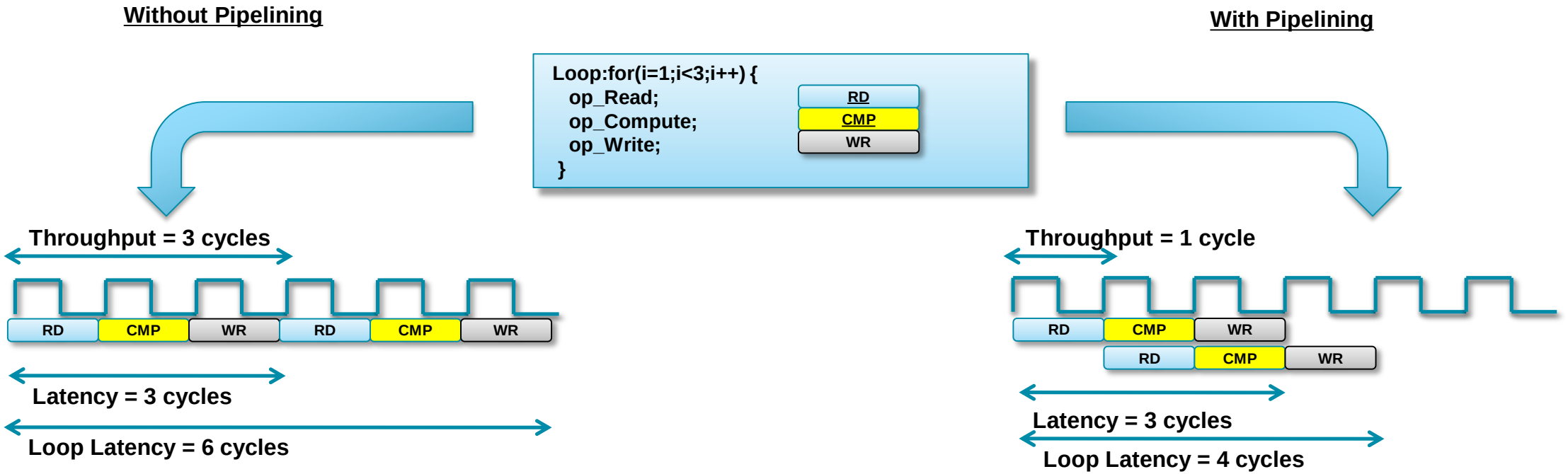




- There are 3 clock cycles before operation RD can occur again
  - Throughput = 3 cycles
- There are 3 cycles before the 1st output is written
  - Latency = 3 cycles

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

Loop Pipelining



- 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

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];
     }
  }
}
```

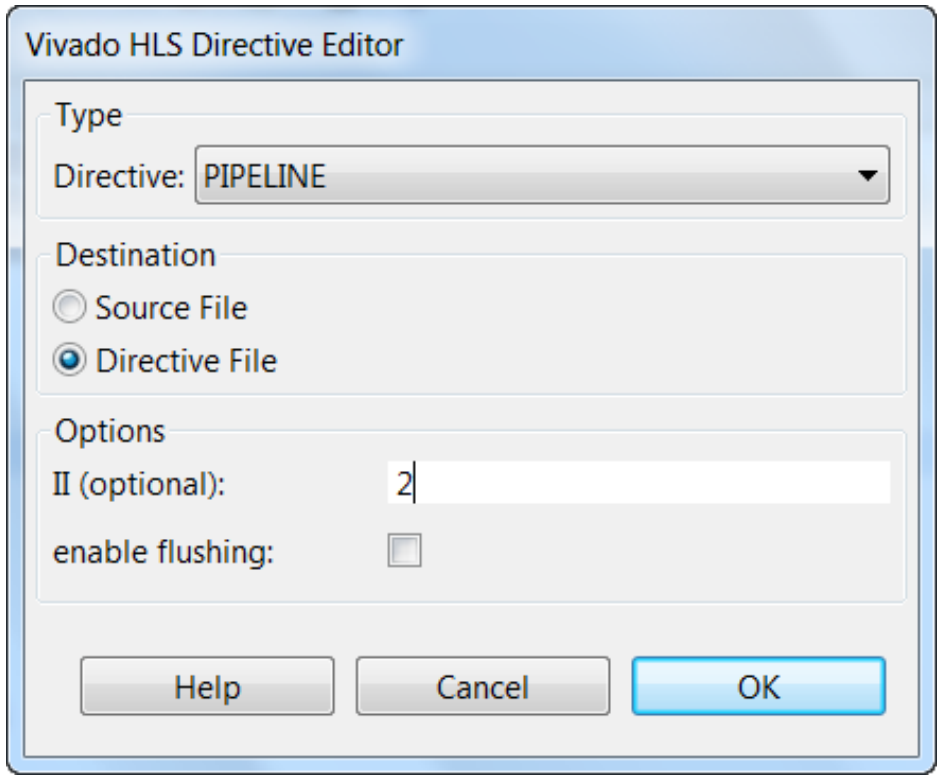
Unrolls L1 and L2
N*M adders, 3(N*M) accesses

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

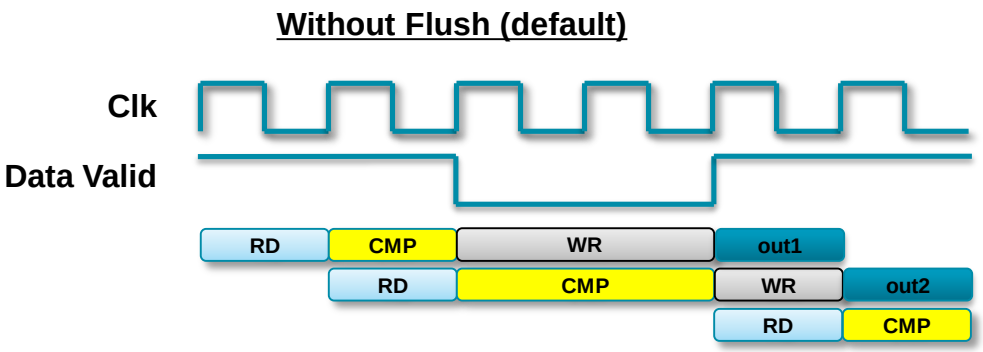
Initiation Interval (or II)

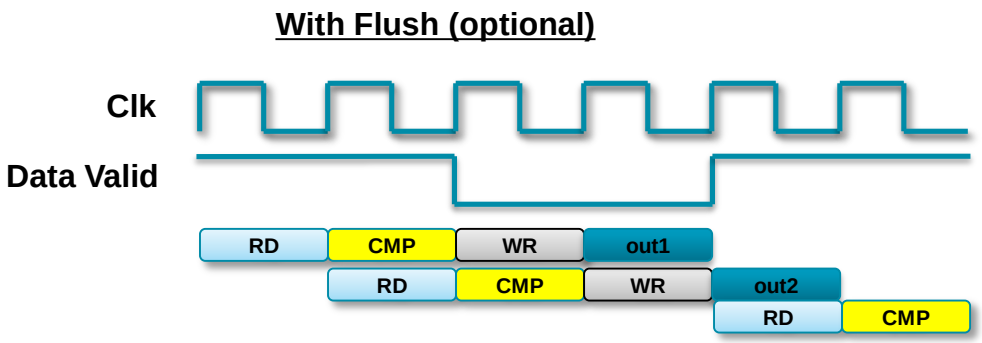
- **Omit the target II and Vivado HLS will Automatically pipeline for the fastest possible design**
  - Specifying a more accurate maximum may allow more sharing (smaller area)



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



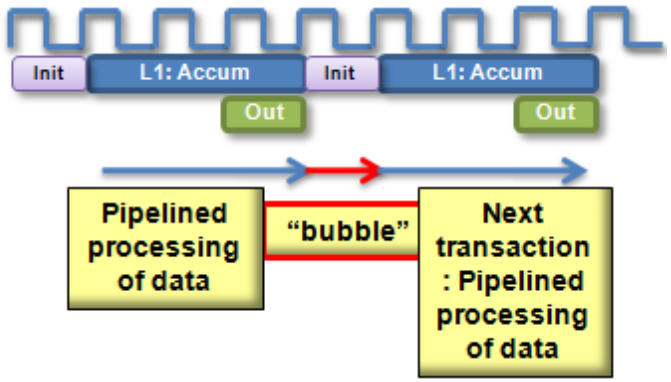


- **With Flush**
  - When no new input reads are performed
  - Values already in the pipeline are flushed out

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;
  ...
}
```

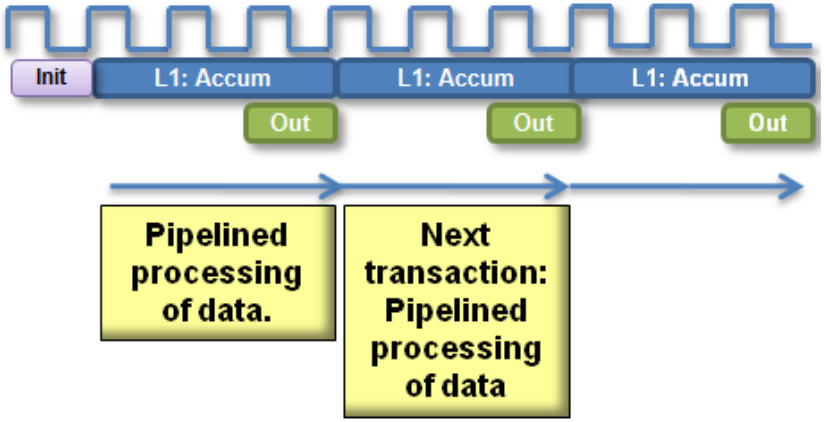


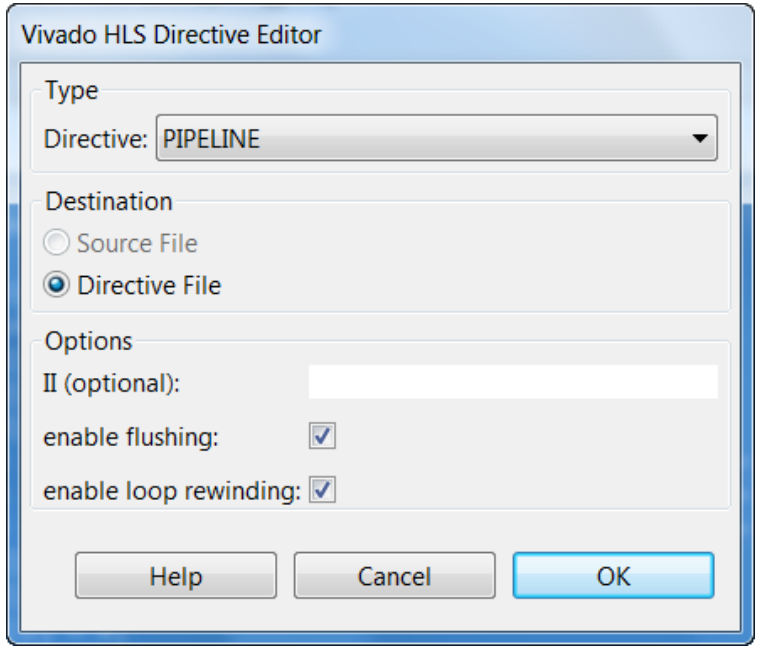
  - The function will process a stream of data
  - The next time the function is called, it still needs to execute the initial (init) operations
    - These operations are any which occur before the loop starts
    - These operations may include interface start/stop/done signals
  - This can result in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

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;
 ...
}
```

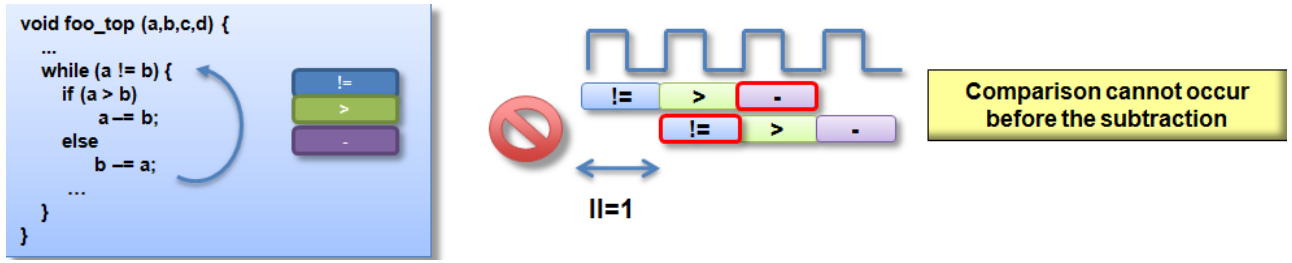




- **The rewind portion only effects top-level loops**
  - Ensures the operations before the loop are never re-executed when the function is re-executed

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)



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

Resource Contention: Unfeasible Initiation Intervals

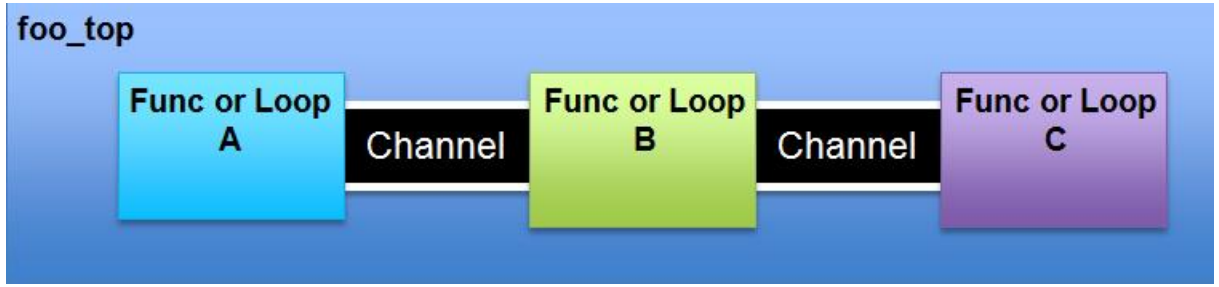
- **Sometimes the II specification cannot be met**
  - In this example there are 2 read operations on the same port

  - 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

Dataflow Optimization

Dataflow Optimization

- Can be used at the top-level function
- Allows blocks of code to operate concurrently
  - The blocks can be functions or loops
  - Dataflow allows loops to operate concurrently
- It places channels between the blocks to maintain the data rate



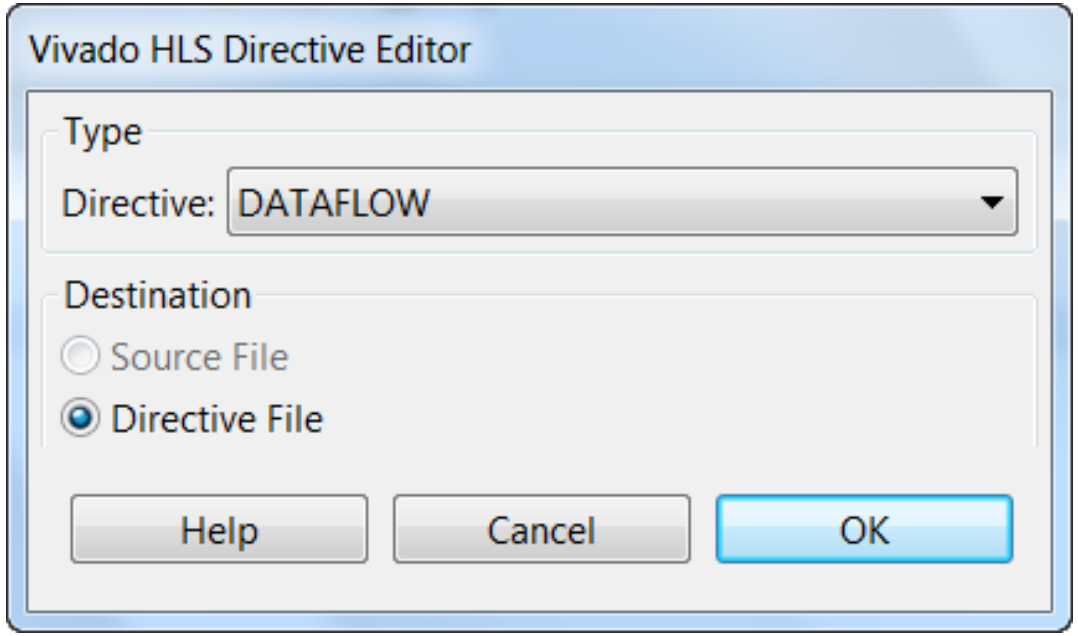
  - For arrays the channels will include memory elements to buffer the samples
  - For scalars the channel is a register with hand-shakes

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

Dataflow Optimization Commands

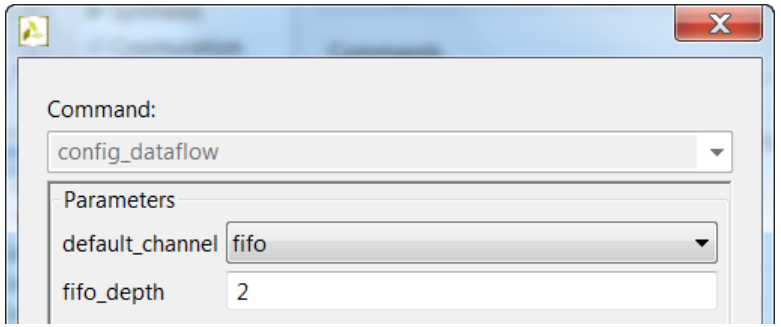
- **Dataflow is set using a directive**
  - Vivado HLS will seek to create the highest performance design
    - Throughput of 1



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

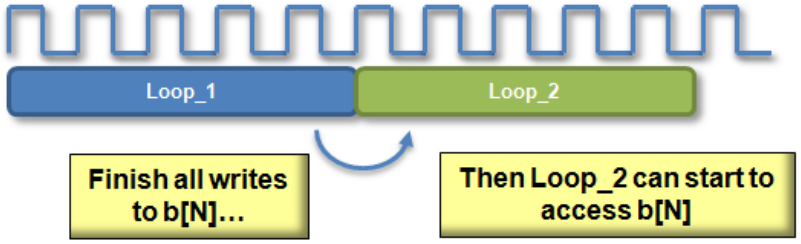
- When the default is ping-pong
  - Select an array and mark it as Streaming (directive STREAM) to implement the array as a FIFO
- When the default is FIFO
  - Select an array and mark it as Streaming (directive STREAM) with option “off” to implement the array as a ping-pong

To use FIFO’s the access must be sequential

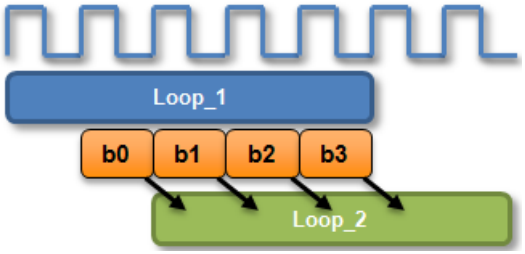
Dataflow : Ideal for streaming arrays & multi-rate functions

- **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++) {
    b[i] = a[i] + in1;
}

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



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



- **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)

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

Outline

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

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

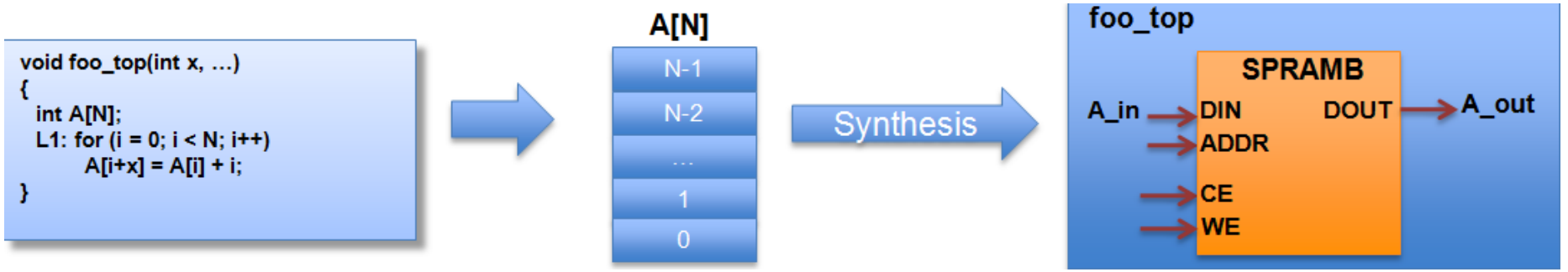
```
void foo_top (...) {
  ...
  for (i = 2; i < N; i++)
    mem[i] = mem[i-1] +mem[i-2];

  }

}
```

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

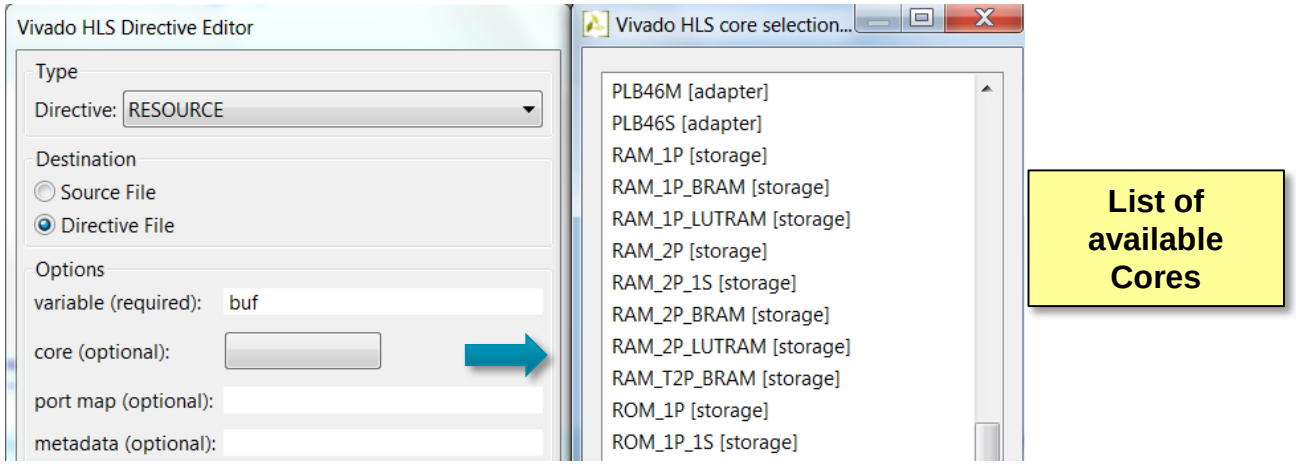
    - Cannot pipeline with a throughput of 1
- **Vivado HLS allows arrays to be partitioned and reshaped**
  - Allows more optimal configuration of the array
  - Provides better implementation of the memory resource

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



- **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



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

Arrays to RAMs : Initialization

- **Static and Const have great impact on array implementation**
  - Const, as shown on previous slide, implies a ROM
  - Static can impact how RAMs are initialized
- **A typical coding style changes**

```
int coeff[8] = {-2, 8, -4, 10, 14, 10, -4, 8, -2};
```

  - Implies coeff is set each time the function is executed
  - Resets in the RAM being explicitly loaded each transaction
    - Costs clock cycles
  - The array can be initialized as a static

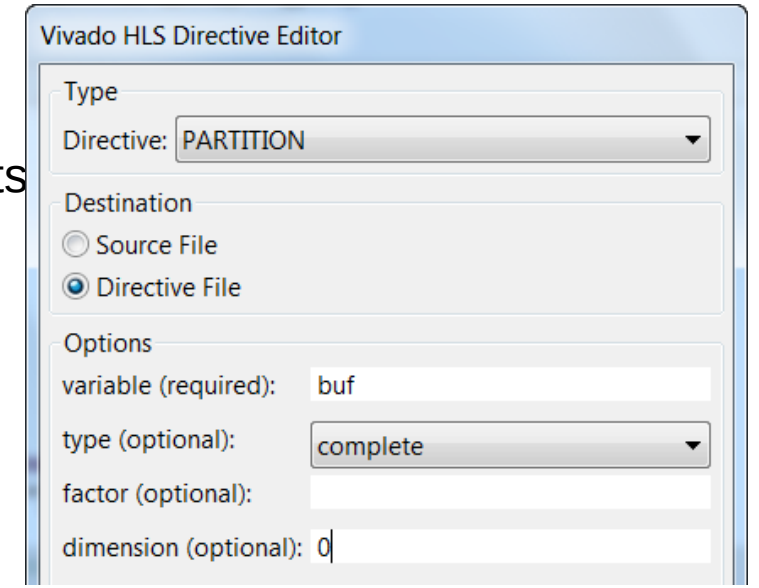
```
static int coeff[8] = {4, -2, 8, -4, 10, 14, 10, -4, 8, -2, 4};
```

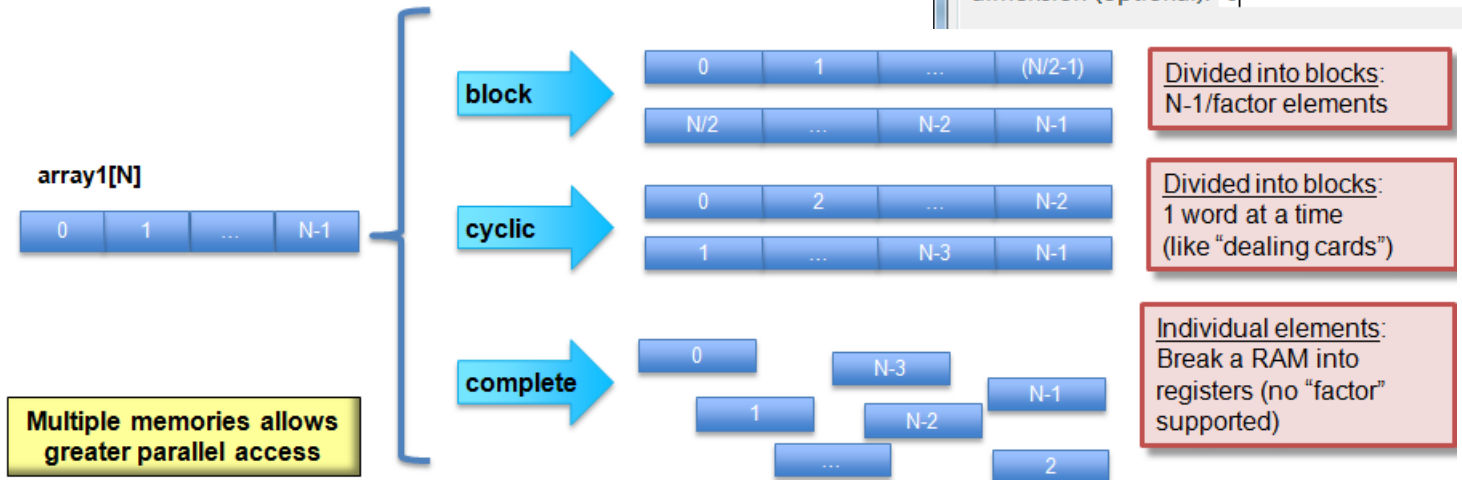
  - Statics are initialized at “start up” only
    - In the C, in the RTL via bitstream

A coefficient array does not need to be initialized every time but doing that costs “nothing” in software, unlike hardware

Array Partitioning

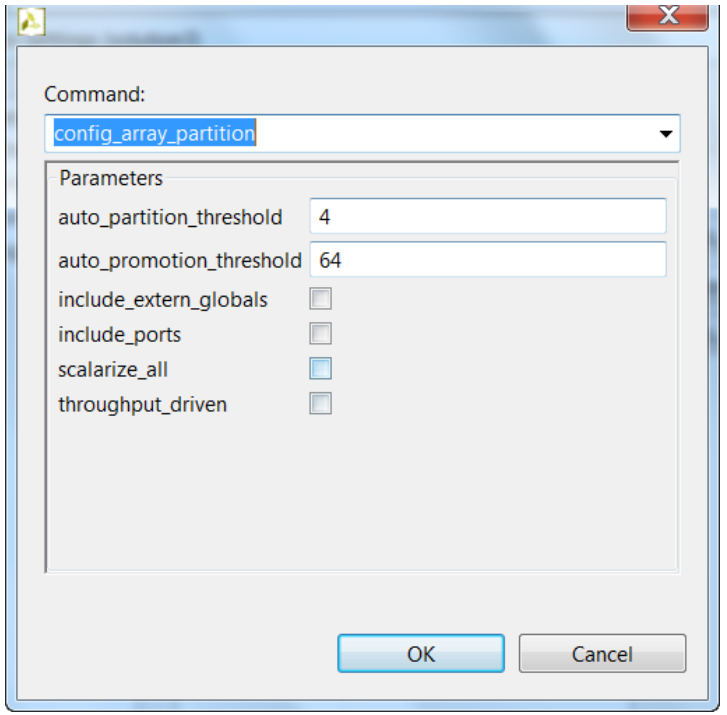
- **Partitioning breaks an array into smaller elements**
  - If the factor is not an integer multiple the final array has fewer elements
  - Arrays can be split along any dimension
    - If none is specified dimension zero is assumed
    - Dimension zero means all dimensions
  - All partitions inherit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





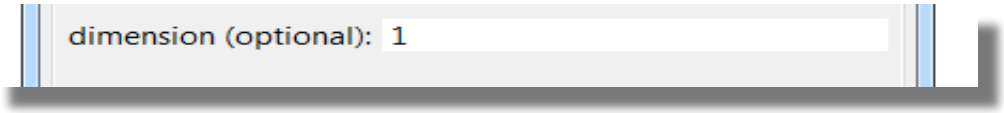
Configuring Array Partitioning

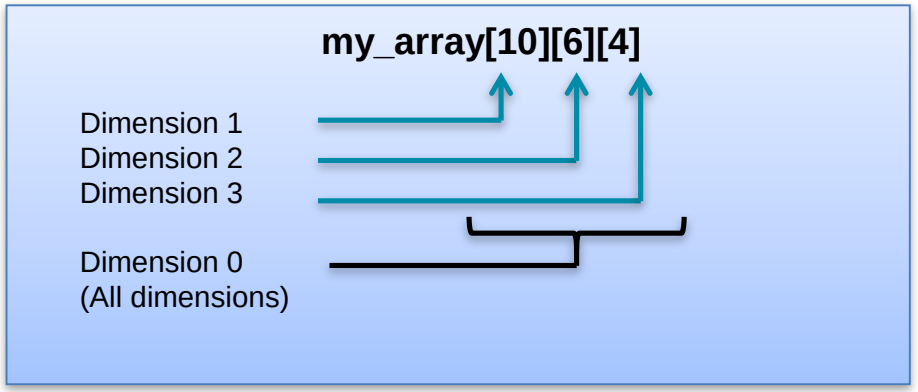
- **Vivado HLS can automatically partition arrays to improve throughput**
  - This is controlled via the array configuration command
  - Enable mode throughput_driven
- **Auto-partition arrays with constant indexing**
  - When the array index is not a variable
  - Arrays below the threshold are auto-partitioned
  - 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



Array Dimensions

- The array options can be performed on dimensions of the array





- Examples

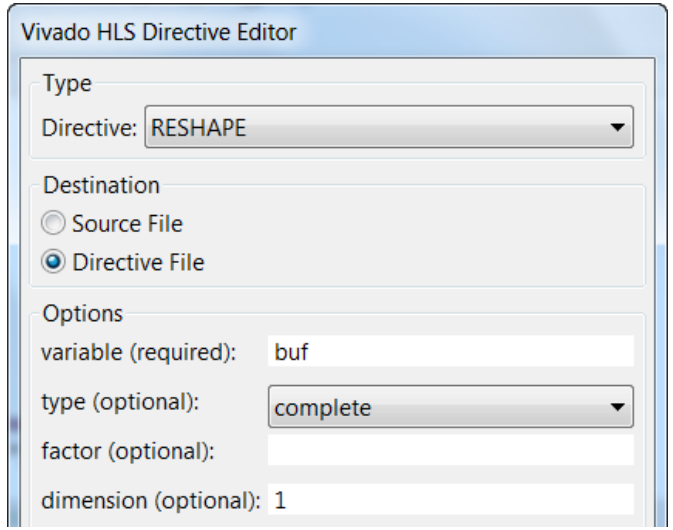
my_array[10][6][4] ➔ partition dimension 3 ➔
my_array_0[10][6]
my_array_1[10][6]
my_array_2[10][6]
my_array_3[10][6]

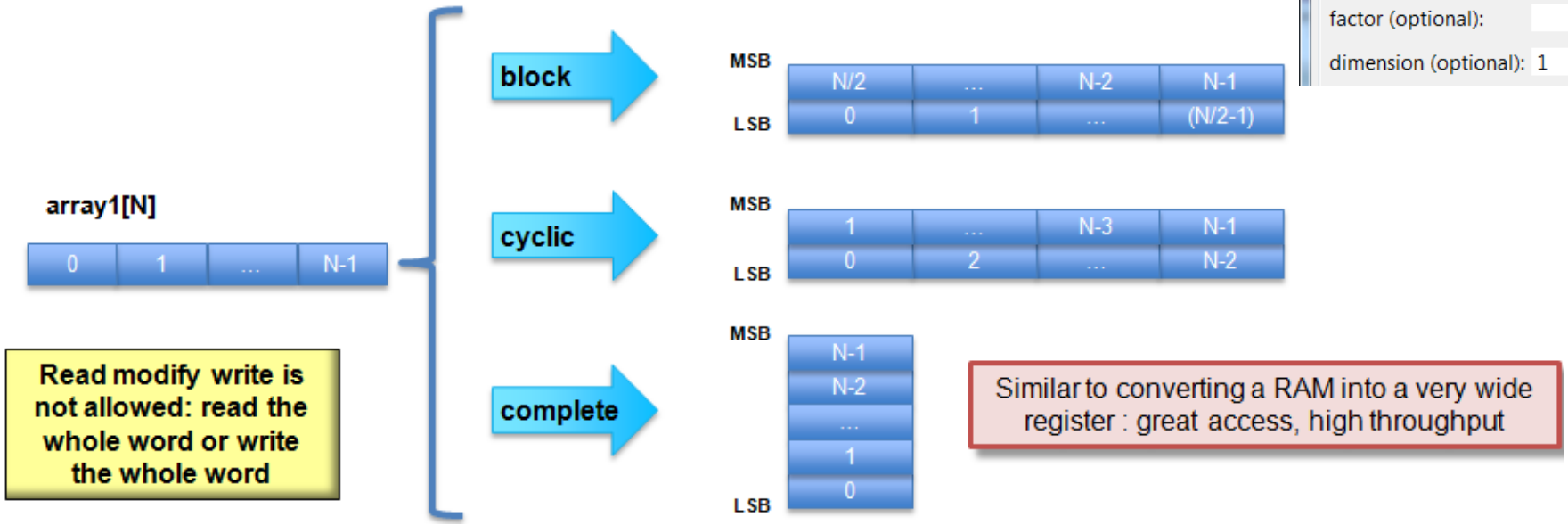
my_array[10][6][4] ➔ partition dimension 1 ➔
my_array_0[6][4]
my_array_1[6][4]
my_array_2[6][4]
my_array_3[6][4]
my_array_4[6][4]
my_array_5[6][4]
my_array_6[6][4]
my_array_7[6][4]
my_array_8[6][4]
my_array_9[6][4]

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

Array Reshaping

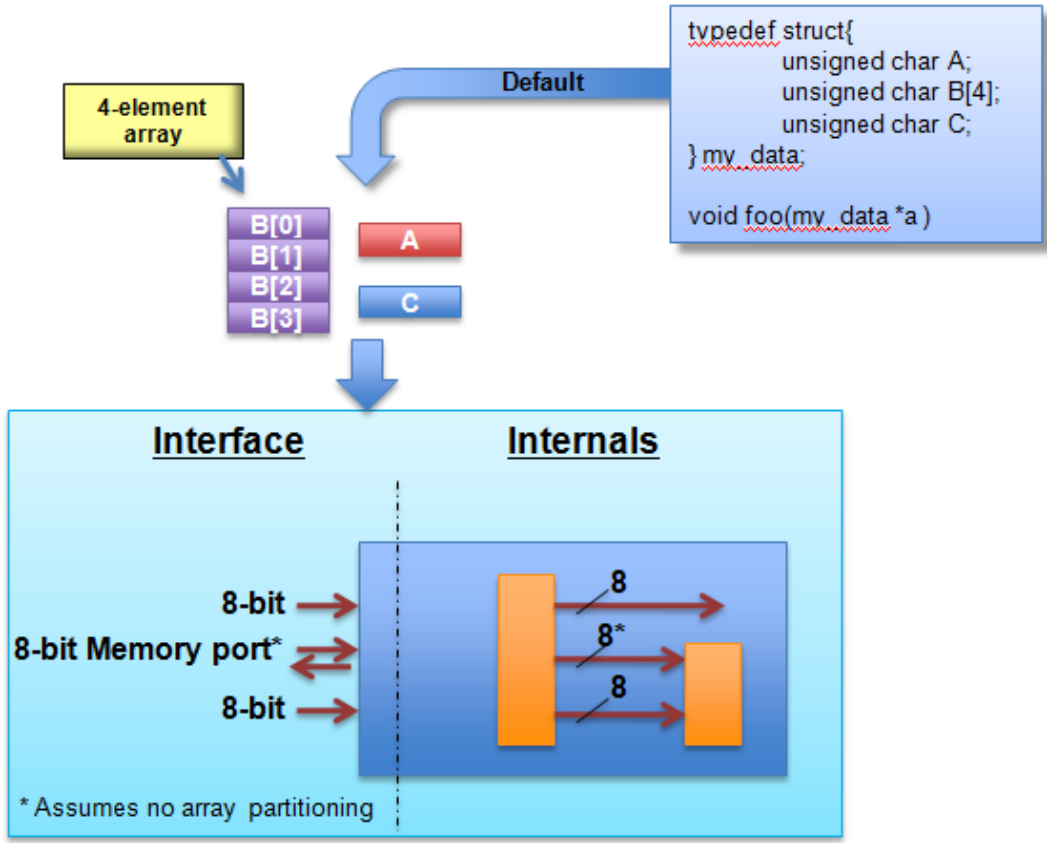
- **Reshaping recombines partitioned arrays back into a single array**
  - Same options as array partition
  - However, reshape automatically recombines the parts back into a single element
  - The “new” array has the same name
    - Same name used for resource targeting





Structs and Arrays: The Default Handling

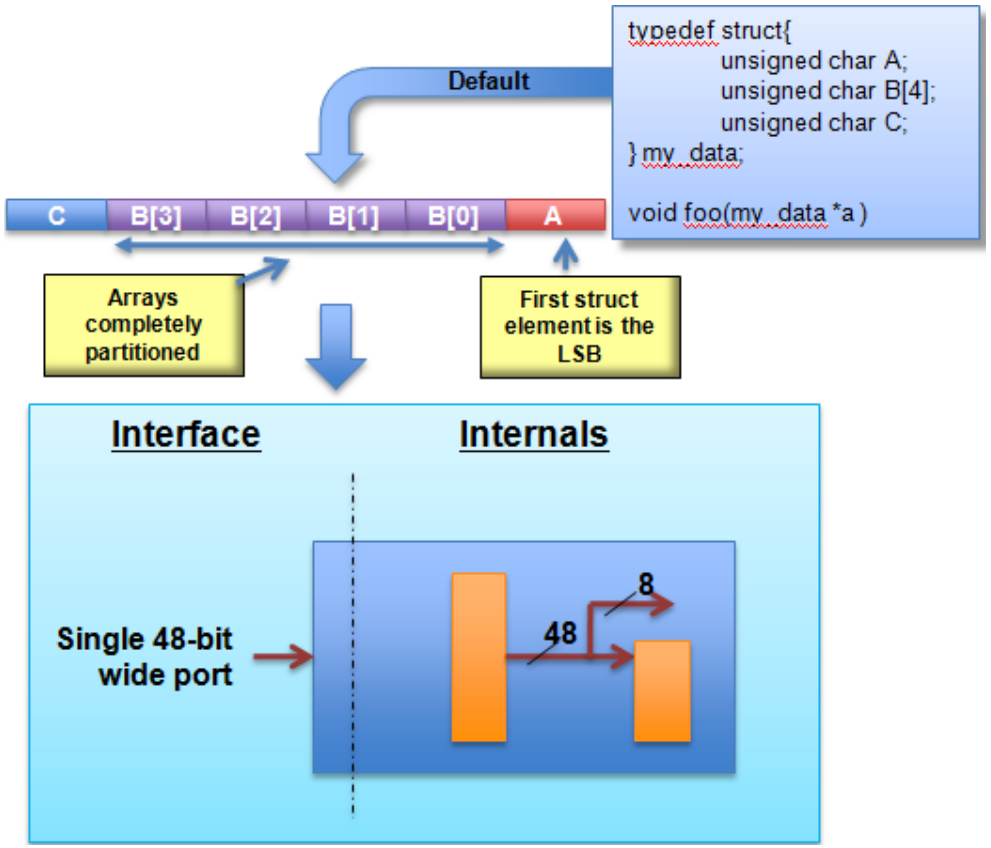
- **Structs are a commonly used coding construct**
  - By default, structs are separated into their separate elements



- **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

Data Packing

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

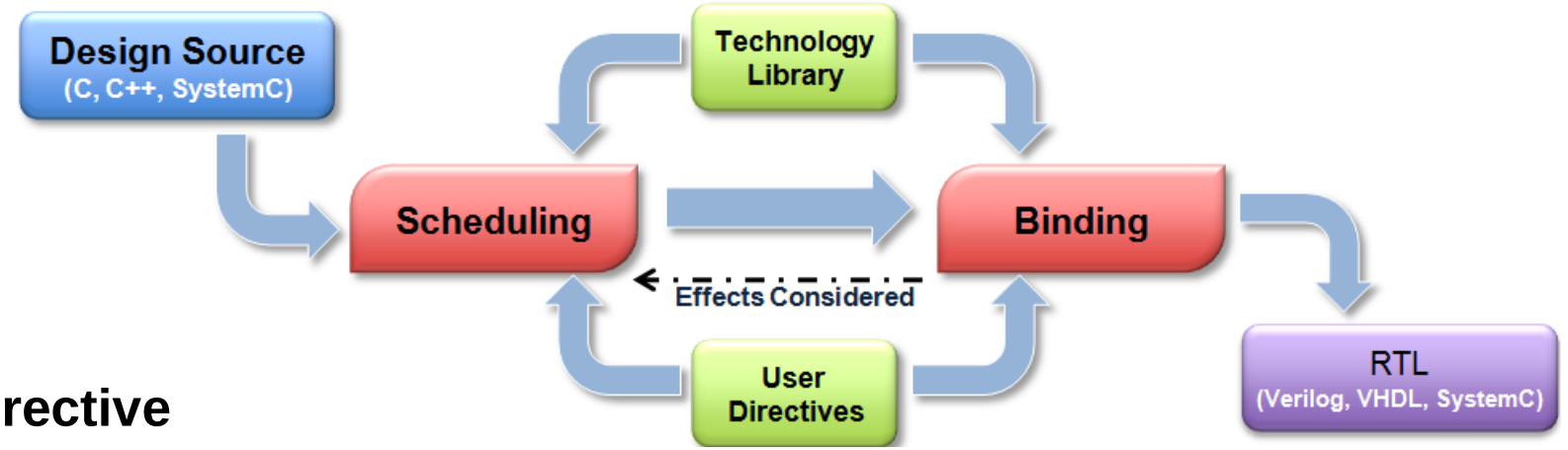


- **Grouped structure**
  - First element in the struct becomes the LSB
  - Last struct element becomes the MSB
  - Arrays are partitioning completely
- **On the Interface**
  - This means a single port
- **Internally**
  - Single bus
  - May result in simplified control logic, faster and lower latency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

Improving Area and Resources

Review: Control Scheduling & Binding

- **Scheduling & Binding**
  - Scheduling and Binding are the processes at the heart of HLS



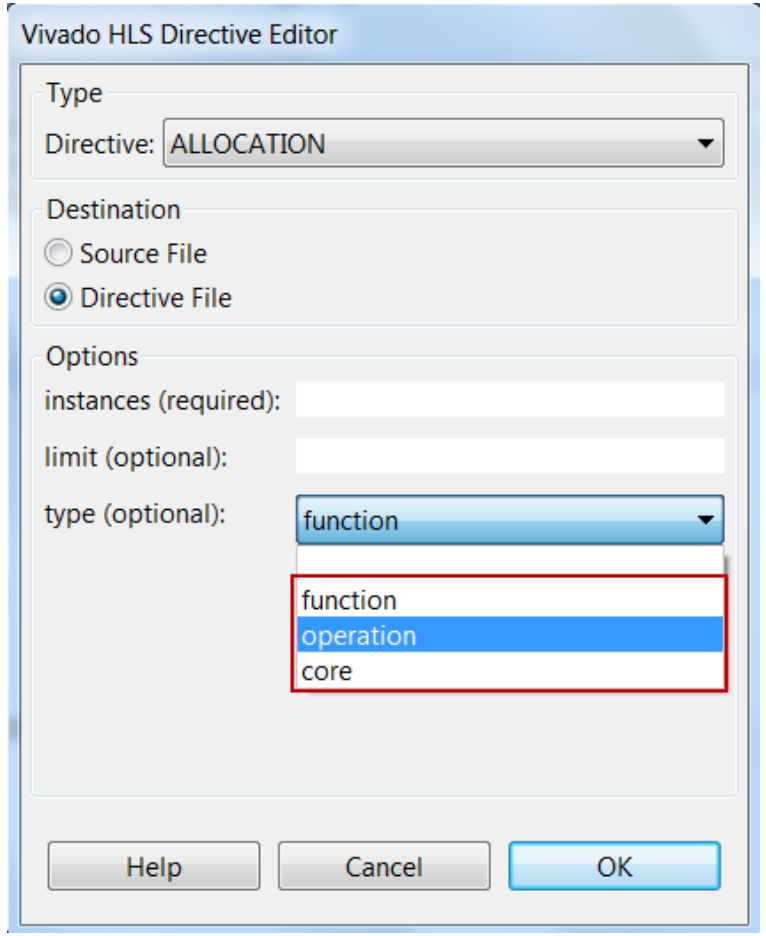
- **The allocation directive**
  - Can be used to limit the number of operation in scheduling & binding stages
- **The resource directive**
  - Can be used to specify which cores are to be used during binding
- **Binding configuration**
  - Can be used to minimize the number of operations

Allocation: Limit the Numbers

- **Allocation directive limits different types**
  - Type: Operation
    - The instances are the operators
    - Add, mul, urem, etc.
  - Type: Core
    - The instances are the cores

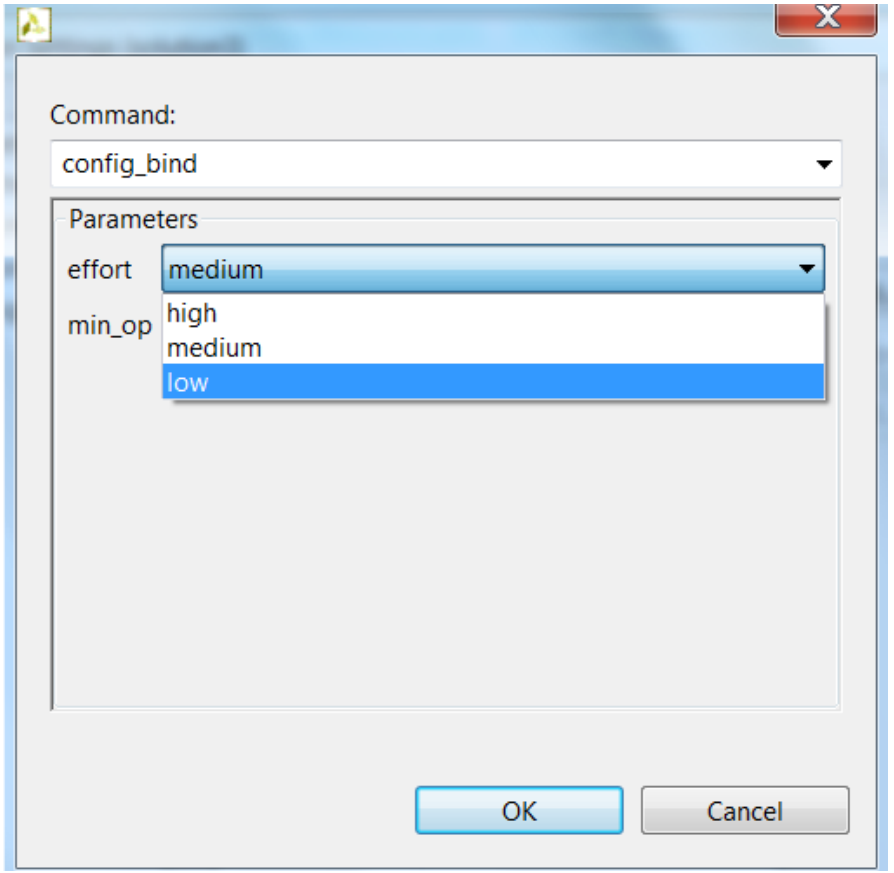
Operators and Cores are listed in the Vivado HLS Library Guide

  - Type: Functions
    - The functions in the code
    - Discussed in more detail later
- **Allocations are defined for a scope**
  - Like all directives, allocations are set for the scope they are applied in
    - If the directive is applied to a function, loop or region, it does not include objects outside that scope



Configuring Binding

- **Binding is controlled via a configuration command**
  - The effort levels determine how much time is spent trying to map many operators onto fewer cores
  - As with all effort levels, they are worth using if you can see the design close to what is required
    - Else the tool will spend time exploring for possibilities
    - And simply increase run time
    - Use efforts judiciously
- **Binding can be configured to minimize specific operators**
  - Can be used to direct Vivado HLS to synthesize with the minimum number of operations
  - The configuration command overrides muxing costs and can be used to force sharing
    - Works on all scopes in a design

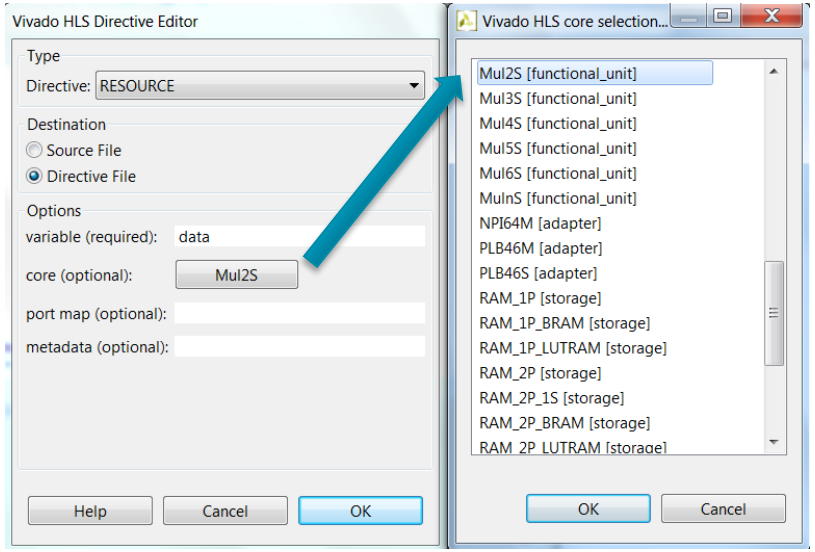


Additional Control: Specify Resources

User control of Resources

- The resource directive gives user control over the specific resource (core) used to implement operations
  - Select the scope & right-click to apply the directive
  - Select “core” for a list of resources
  - Specify the variable

In this example, “data” is implemented with a 2-stage pipelined multiplier



Multiple line coding caveat

- If multiple operations occur on a single line
- A temporary variable is required to isolate the specific operation

```
a=b*c*d
```

```
data= c*d;
a = b*data;
```

Improving Area

- **Control the number of elements**
  - Directives can be used to control scheduling and binding
- **Control the design hierarchy**
  - Like RTL synthesis, removing the hierarchy can help optimize across function and loop boundaries
    - Functions can be inlined
    - Loops can be unrolled
- **Array implementation**
  - Vivado HLS provides directives for combining memories
    - Allowing a single large memory to be used instead of multiple smaller memories
- **Bit-width optimization**
  - Arbitrary precision types ensure correct operator sizing

Review: Functions & RTL Hierarchy

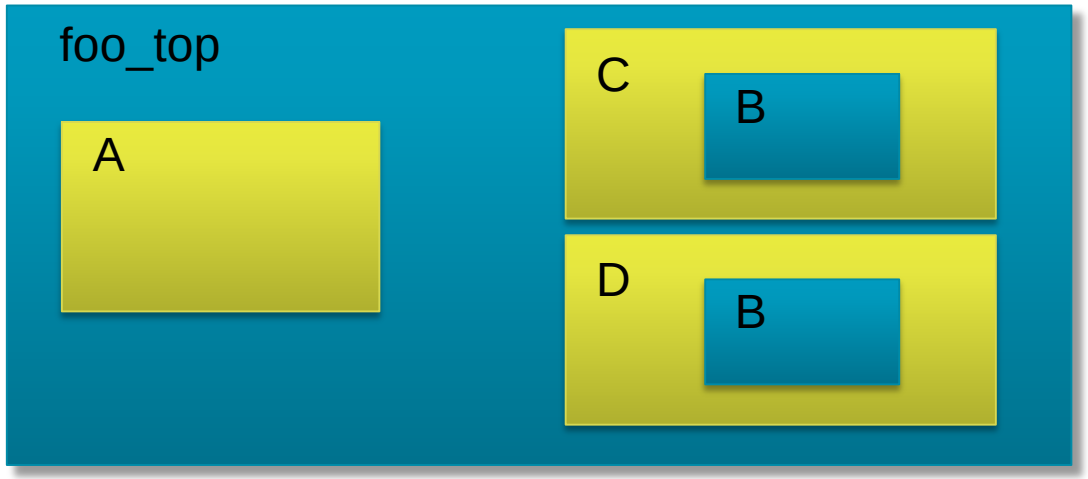
- **Each function is translated into an RTL block**
  - Verilog module, VHDL entity

Source Code

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


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

my_code.c

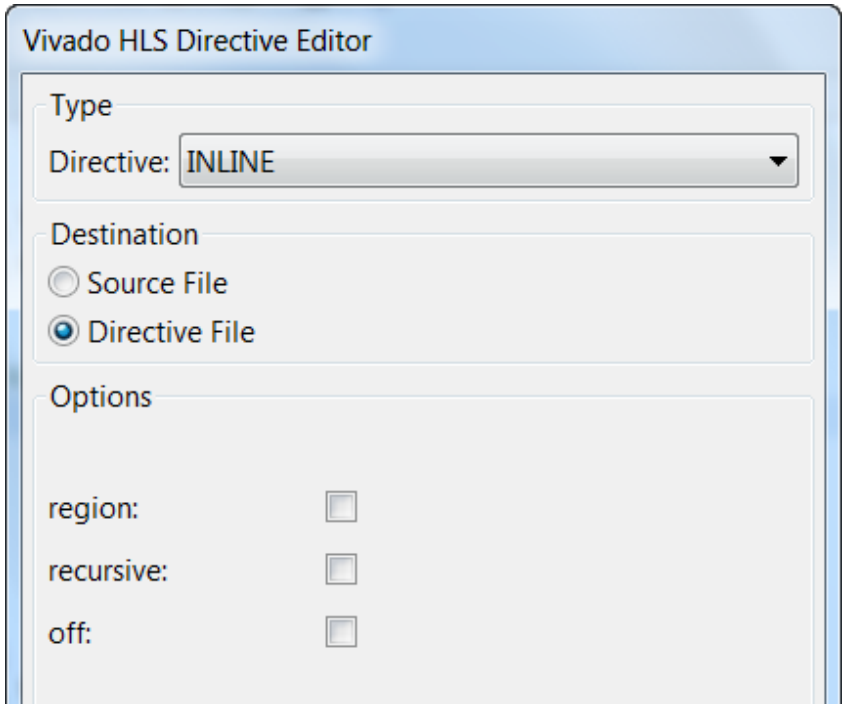
RTL hierarchy



Functions can be inlined – the hierarchy removed & the function dissolved into the surrounding function

Controlling Inlining

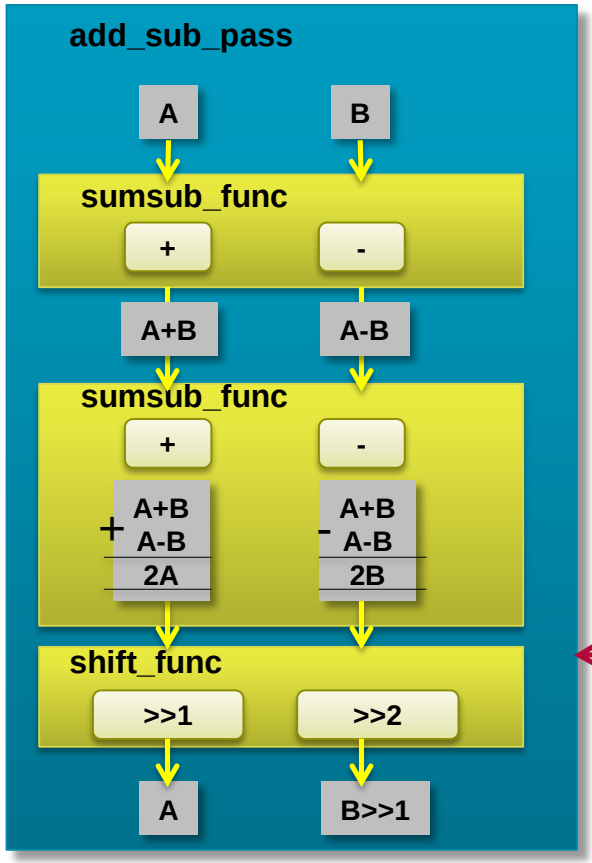
- **Vivado HLS performs some inlining automatically**
  - This is performed on small logic functions if Vivado HLS determines area or performance will benefit
- **User Control**
  - Functions can be specifically inlined
    - The function itself is inlined
  - Optionally recursively down the hierarchy
  - Optionally everything within a region can be inlined
    - Everything named region or a function or a loop
  - Optionally inlining can be explicitly prevented
    - Turn inlining off
- **Inlining functions allows for greater optimization**
  - Like ungrouping RTL hierarchies: optimization across boundaries
  - Like ungrouping RTL hierarchies it can result in lots of operations & impact run time



Function Inlining

- Inlining can be used to remove function hierarchy

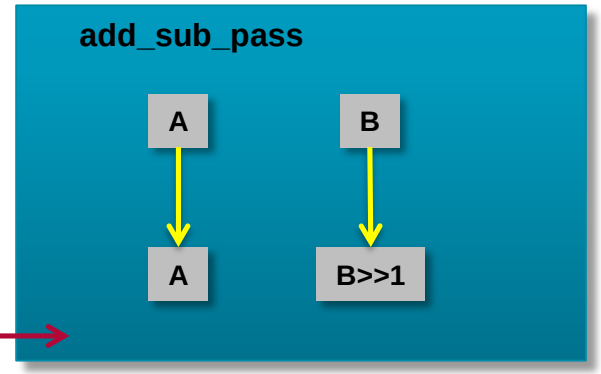
No Inlining




```
int sumsub_func (int *in1, int *in2, int *outSum, int *outSub) {
*outSum = *in1 + *in2;
*outSub = *in1 - *in2;
}

int shift_func (int *in1, int *in2, int *outA, int *outB) {
   *outA = *in1 >> 1;
   *outB = *in2 >> 2;
}

void add_sub_pass(int A, int B, int *C, int *D) {
   int apb, amb;
   int a2, b2;

   sumsub_func(&A,&B,&apb,&amb);
   sumsub_func(&apb,&amb,&a2,&b2);
   shift_func(&a2,&b2,C,D);

}
```

2 Adders
2 Subtractors

Inlining



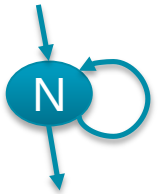
Zero Area

Inlining allows optimization to be performed across function hierarchies

Like RTL ungrouping, too much inlining can create a lot of logic and slow runtime

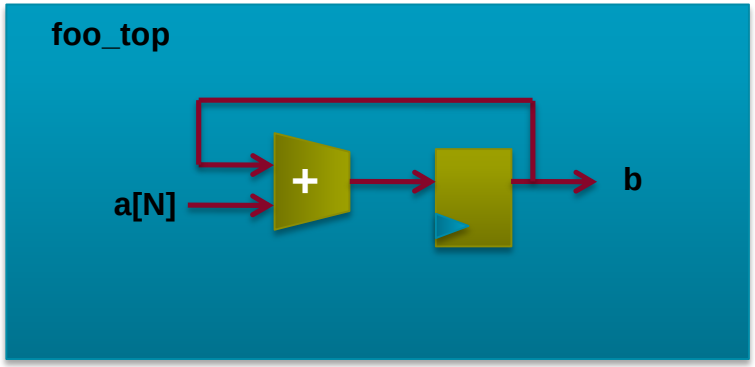
Loops

- **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;
  ...
  }
```

Synthesis



- **For Area optimization**

Keeping loops rolled maximizes sharing across loop iterations: each iteration of the loop uses the same hardware resources

Loop Merging & Flattening

- **Loop merging & flattening can remove the redundant computation among multiple (related) loops**
  - Improving area (and sometimes performance)

```
My_Region: {
#pragma HLS merge loop
for (i = 0; i < N; ++i)
    A[i] = B[i] + 1;
for (i = 0; i < N; ++i)
    C[i] = A[i] / 2;
}
```

Merge

```
for (i = 0; i < N; ++i) {
    A[i] = B[i] + 1;
    C[i] = A[i] / 2;
}
```

Effective code after compiler transformation

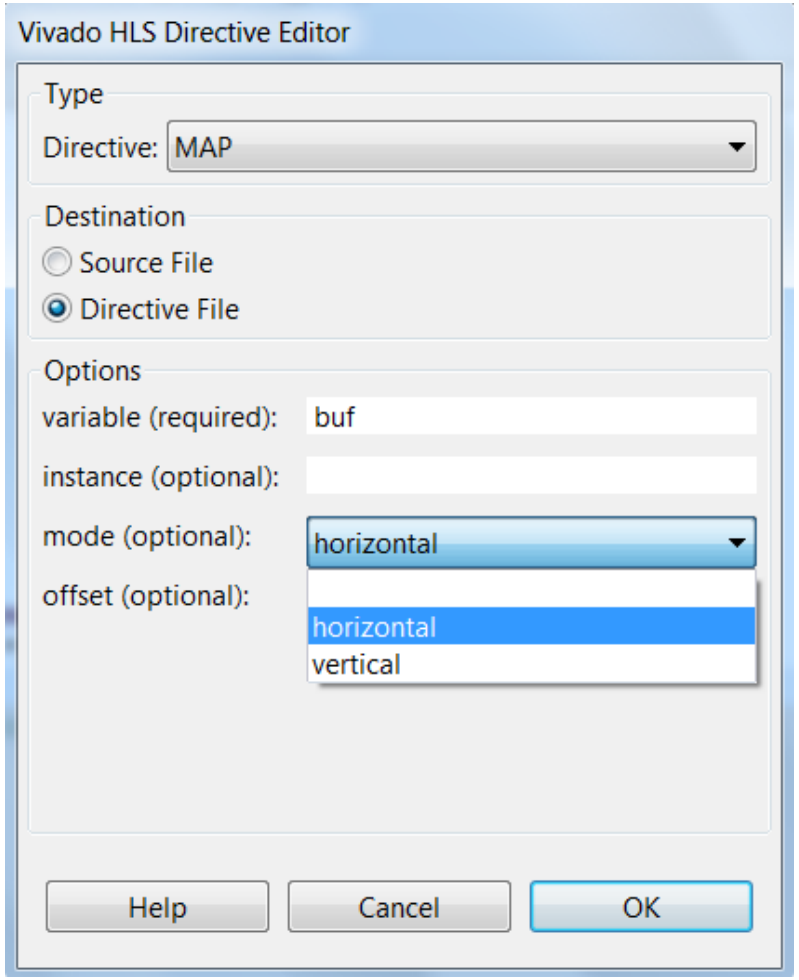
- **Allows Vivado HLS to perform optimizations**
  - Optimization cannot occur across loop boundaries

```
for (i = 0; i < N; ++i)
    C[i] = (B[i] + 1) / 2;
```

Removes A[i], any address logic and any potential memory accesses

Mapping Arrays

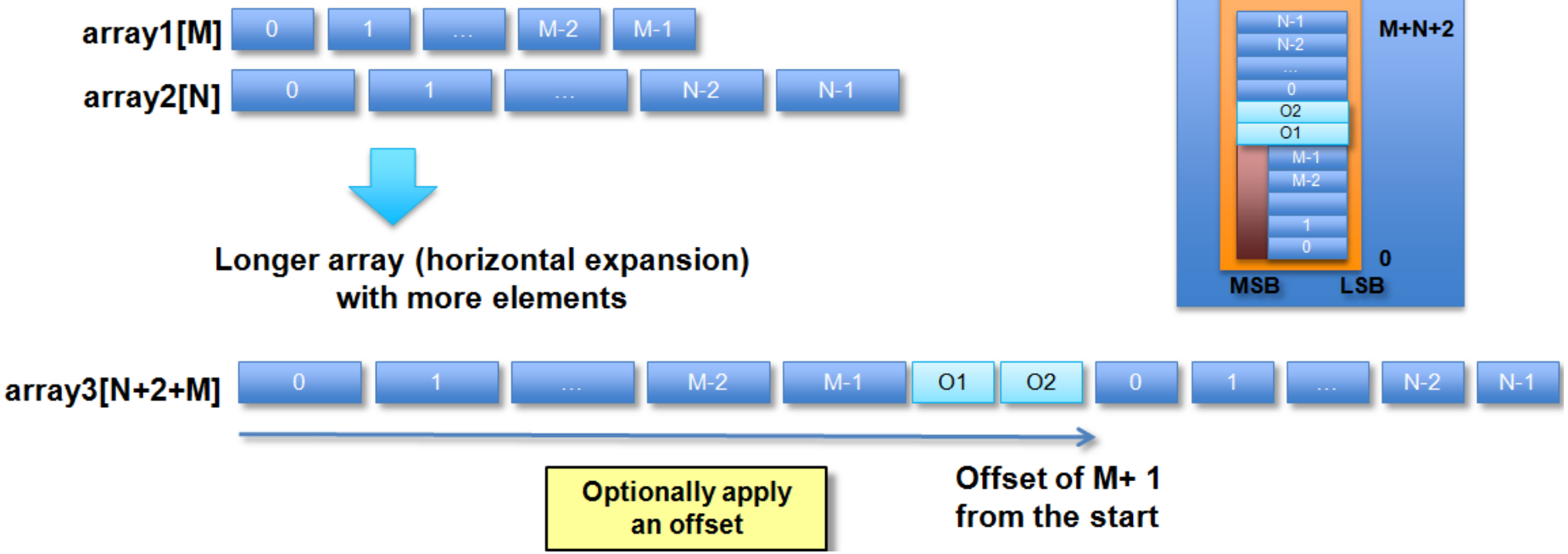
- **The arrays in the C model may not be ideal for the available RAMs**
  - The code may have many small arrays
  - The array may not utilize the RAMs very well
- **Array Mapping**
  - Mapping combines smaller arrays into larger arrays
    - Allows arrays to be reconfigured without code edits
  - Specify the array variable to be mapped
  - Give all arrays to be combined the same instance name
- **Vivado HLS provides options as to the type of mapping**
  - Combine the arrays without impacting performance
    - Vertical & Horizontal mapping
- **Global Arrays**
  - When a global array is mapped all arrays involved are promoted to global
  - When arrays are in different functions, the target becomes global
- **Arrays which are function arguments**
  - All must be part of the same function interface



Horizontal Mapping

Horizontal Mapping

- Combines multiple arrays into longer (horizontal) array
- Optionally allows the arrays to be offset
  - The default is to concatenate after the last element

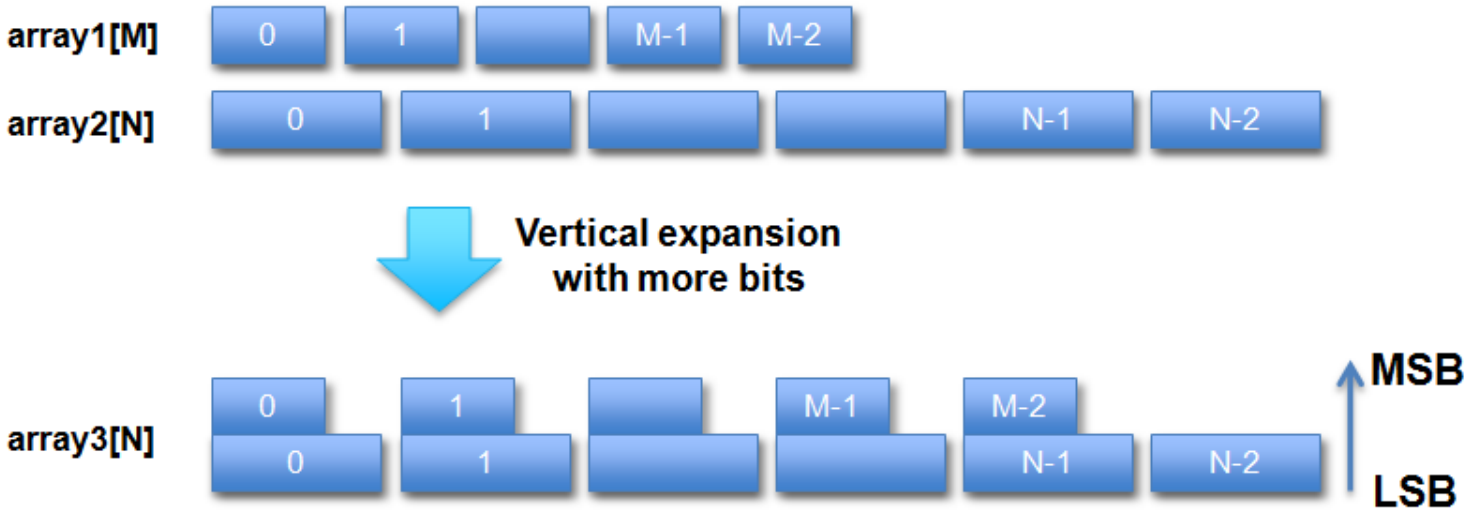


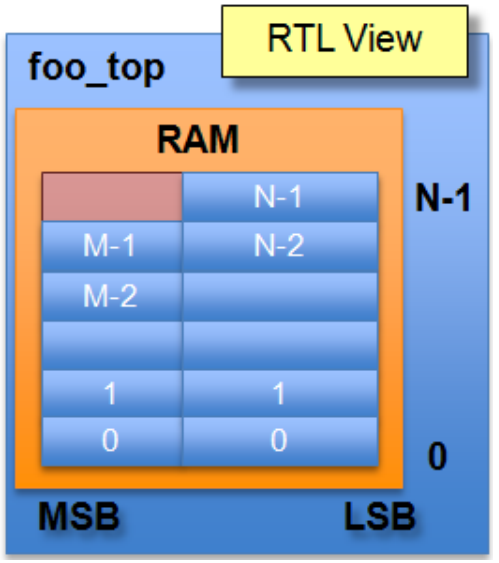
  - The first array specified (in GUI or Tcl script) starts at location zero

Vertical Mapping

Vertical Mapping

– Combines multiple arrays in to an array with more bits





– The first array specified (in Tcl or GUI) starts at the LSB

Vertical Mapping for performance

– Creates RAMs with wide words ➔ Parallel accesses

Block and Port Level Protocols

MicroBlaze
Vivado HLS 2012.2 Version

The Key Attributes of C code : IO

Code

```
void fir (

  data_t *y,
  coef_t c[4],
  data_t x

 ) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

  acc=0;
  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];
}
  }
  *y=acc>>2;
}
```

Functions: All code is made up of functions which represent the design hierarchy: the same in hardware

Input & Outputs: The arguments of the top-level function must be transformed to hardware interfaces with an IO protocol

Types: All variables are of a defined type. The type can influence the area and performance

Loops: Functions typically contain loops. How these are handled can have an impact on area and performance.

Arrays: Arrays are used often in C code. They can impact the device area and become performance bottlenecks.

Operators: Operators in the C code may require sharing to control area or be assigned to specific hardware implementations to meet performance

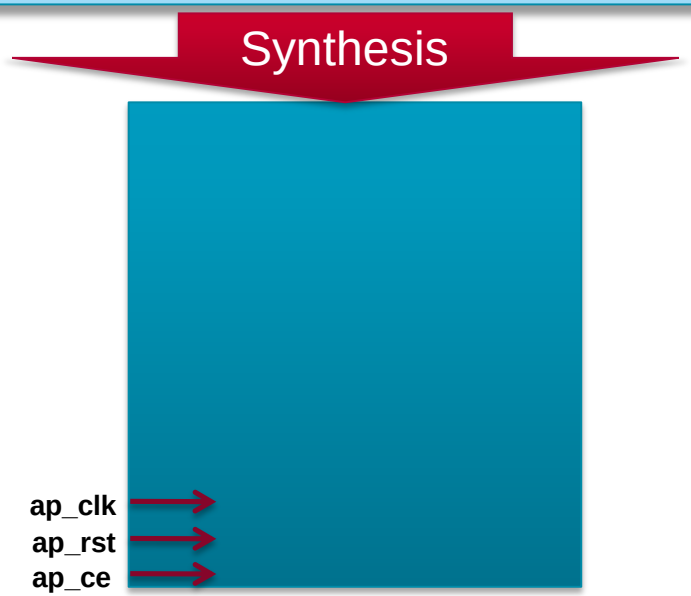
Vivado HLS IO Options

- **Vivado HLS has four types of IO**
  1. Data ports created by the original C function arguments
  2. IO protocol signals added at the Block-Level
  3. IO protocol signals added at the Port-Level
  4. IO protocol signals added externally as Pcore Interfaces
- **Data Ports**
  - These are the function arguments/parameters
- **Block-Level Interfaces (optional)**
  - An interface protocol which is added at the block level
  - Controls the addition of block level control ports: start, idle, done, and ready
- **Port-Level interfaces (optional)**
  - IO interface protocols added to the individual function arguments
- **Pcore interfaces (optional)**
  - Added as external adapters when a pcore for use in EDK is generated

Vivado HLS Basic Ports

Adder Example

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

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

        return  temp;
}
```

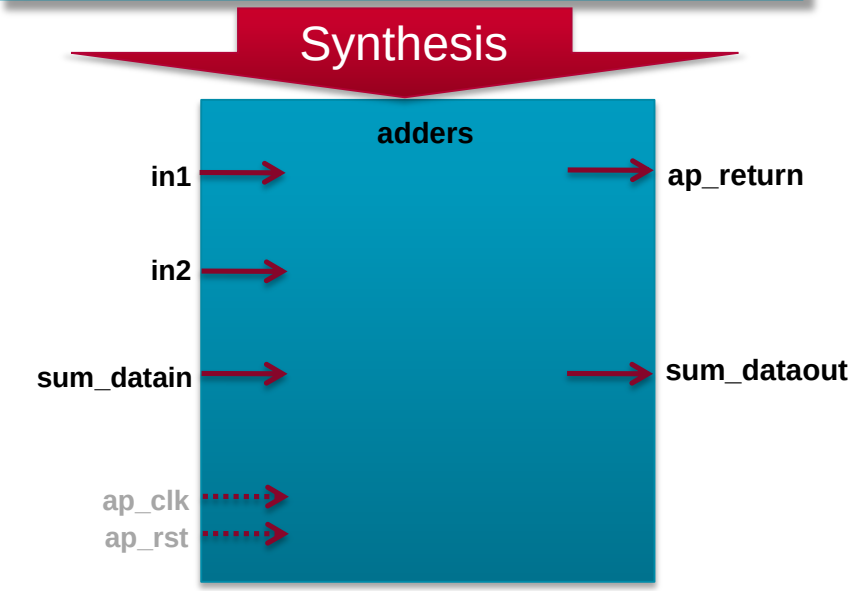


- **Clock added to all RTL blocks**
  - One clock per function/block
  - SystemC designs may have a unique clock for each CTHREAD
- **Reset added to all RTL blocks**
  - Only the FSM and any variables initialized in the C are reset by default
  - Reset and polarity options are controlled via the RTL Configuration
    - Solutions/Solution Settings...
- **Optional Clock Enable**
  - An optional clock enable can be added via the RTL configuration
  - When de-asserted it will cause the block to “freeze”
    - All connected blocks are assumed to be using the same CE
    - When the IO protocol of this block freezes, it is expected other blocks do the same
    - Else a valid output may be read multiple times

Vivado HLS IO Options: Function Arguments

Adder Example

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

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

        return  temp;
}
```



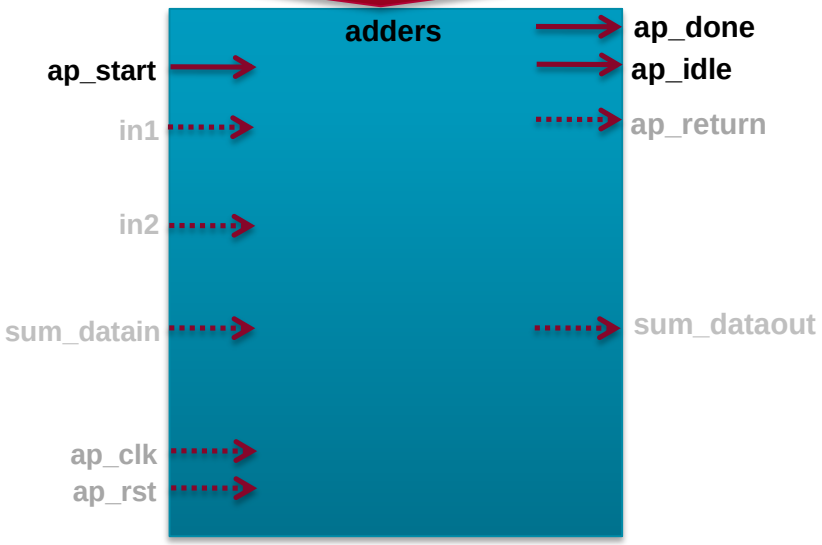
- **Function Arguments**
  - Synthesized into data ports
- **Function Return**
  - Any return is synthesized into an output port called ap_return
- **Pointers (& C++ References)**
  - Can be read from and written to
  - Separate input and output ports for pointer reads and writes
- **Arrays (not shown here)**
  - Like pointers can be synthesized into read and/or write ports

Vivado HLS IO Options: Block Level Protocol

Adder Example

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

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

        return  temp;
}
```

Synthesis

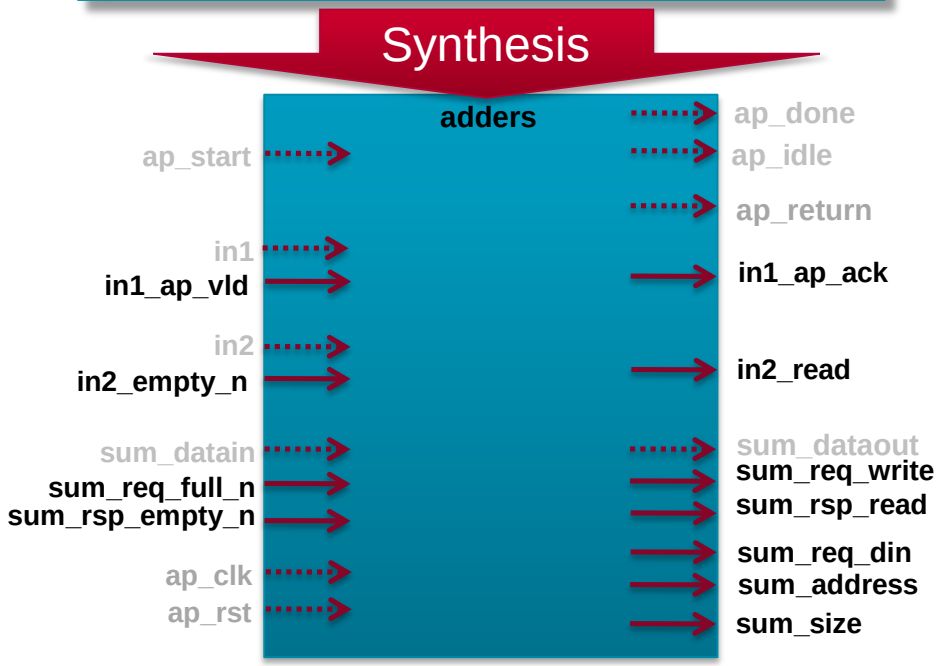


- **Block Level Protocol**
  - An IO protocol added at the RTL block level
  - Controls and indicates the operational status of the block
- **Block Operation Control**
  - Controls when the RTL block starts execution (ap_start)
  - Indicates if the RTL block is idle (ap_idle) or has completed (ap_done)
- **Complete and function return**
  - The ap_done signal also indicates when any function return is valid
- **Ready (not shown here)**
  - If the function is pipelined an additional ready signal (ap_ready) is added
  - Indicates a new sample can be supplied before done is asserted

Vivado HLS IO Options: Port IO Protocols

Adder Example

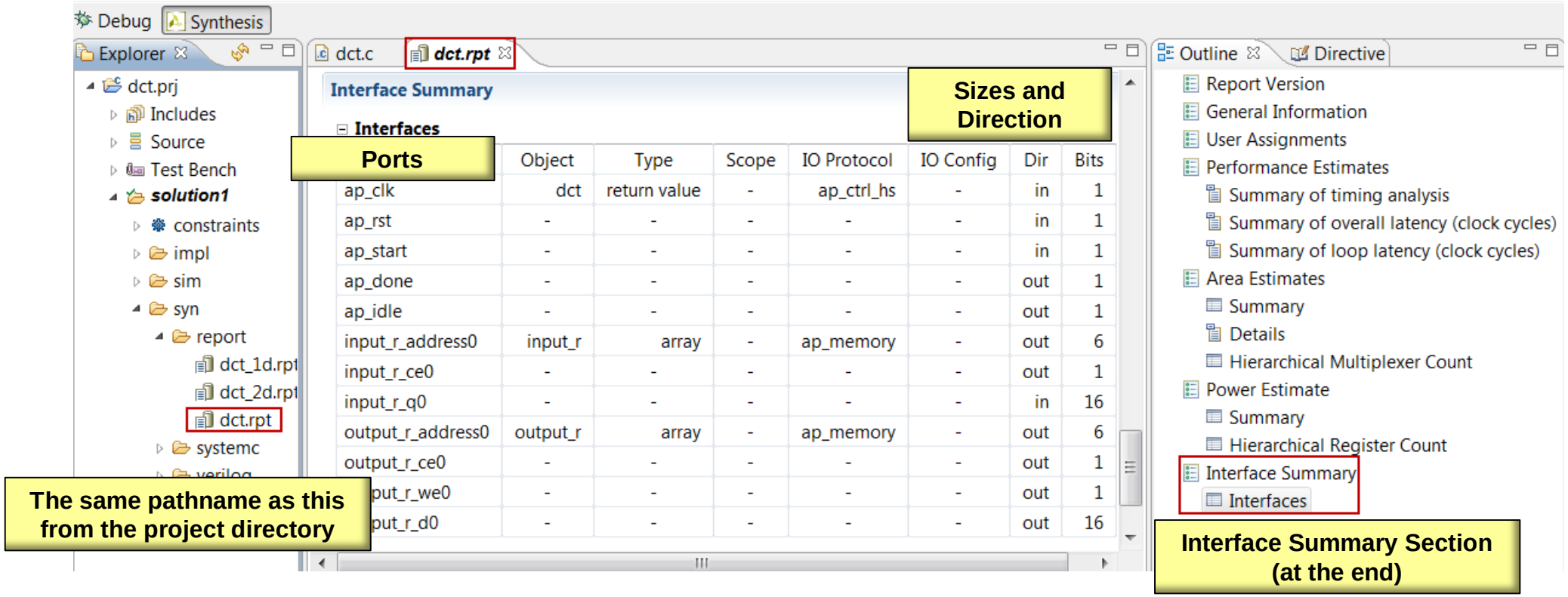
```
#include "adders.h"
int adders(int in1, int in2, int *sum) {

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

        return  temp;
}
```



- **Port IO Protocols**
  - An IO protocol added at the port level
  - Sequences the data to/from the data port
- **Interface Synthesis**
  - The design is automatically synthesized to account for IO signals (enables, acknowledges etc.)
- **Select from a pre-defined list**
  - The IO protocol for each port can be selected from a list
  - Allows the user to easily connect to surrounding blocks
- **Non-standard Interfaces**
  - Supported in C/C++ using an arbitrary protocol definition
  - Supported natively in SystemC

Vivado HLS Interfaces

- **Where do you find the summary?**
  - In the Synthesis report

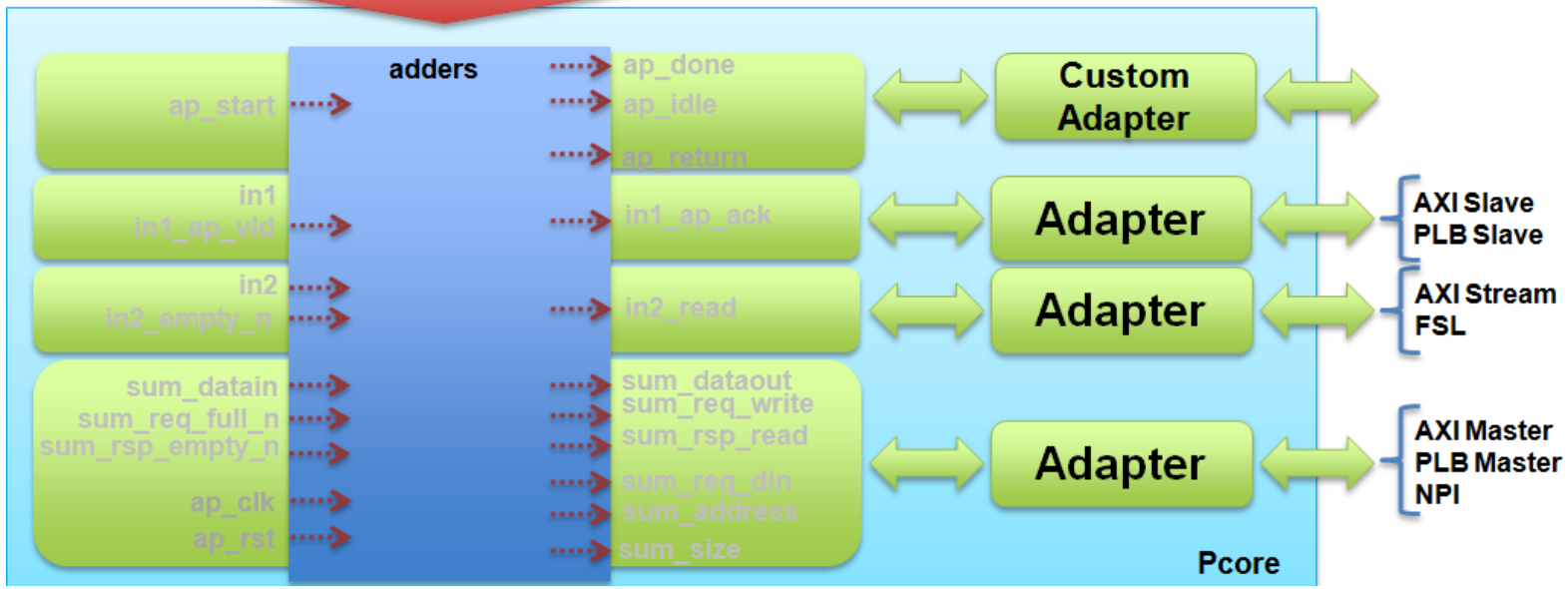


Vivado HLS IO Options: Pcore Adapters

Adder Example

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

        int temp;

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

        return  temp;
}
```

Synthesis



- Added when the IP block is exported
- Available in the IP directory
  - "ip", "sysgen" or "pcore"

Outline

- Introduction
- *Block Level Protocols*
- Port Level Protocols
- Summary

Example 101 : Combinational Design

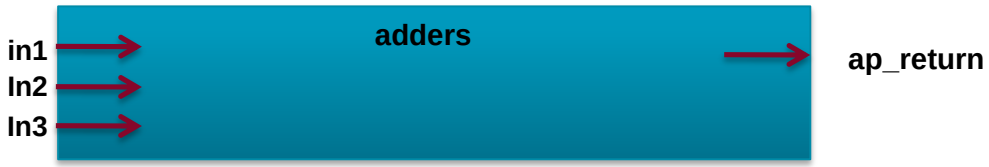
Simple Adder Example

- Output is the sum of 3 inputs

```
#include "adders.h"
int adders(int in1, int in2, int in3) {

        int sum;
        sum = in1 + in2 + in3;
        return sum;
}
```

Synthesized with 100 ns clock

- All adders can fit in one clock cycle
- **Combinational design**



- The function return becomes RTL port ap_return
- No handshakes are required or created in this example

```
================================================================
== Performance Estimates
================================================================
+ Summary of timing analysis:
    * Estimated clock period (ns): 3.45
+ Summary of overall latency (clock cycles):
    * Best-case latency:    0
    * Average-case latency: 0
    * Worst-case latency:   0
```

Example 102: Sequential Design

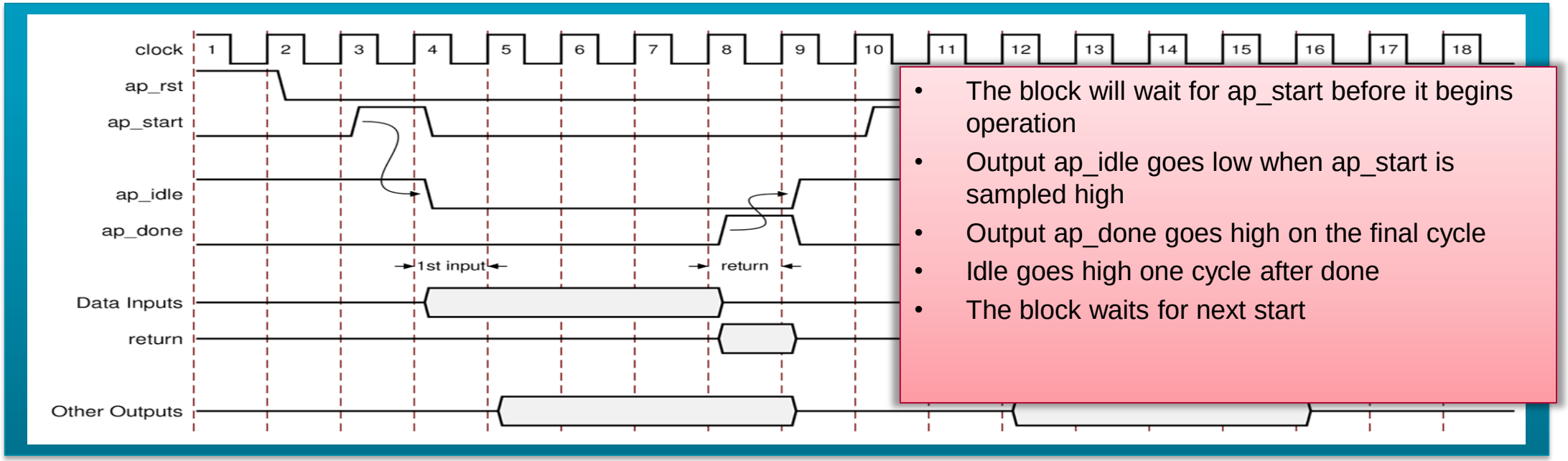
- **The same adder design is now synthesized with a 3ns clock**
  - The design now takes more than 1 cycle to complete
  - Vivado HLS creates a **sequential** design with the default port types

```
#include "adders.h"
int adders(int in1, int in2, int in3) {

        int sum;
        sum = in1 + in2 + in3;
        return sum;
}
```

Synthesis

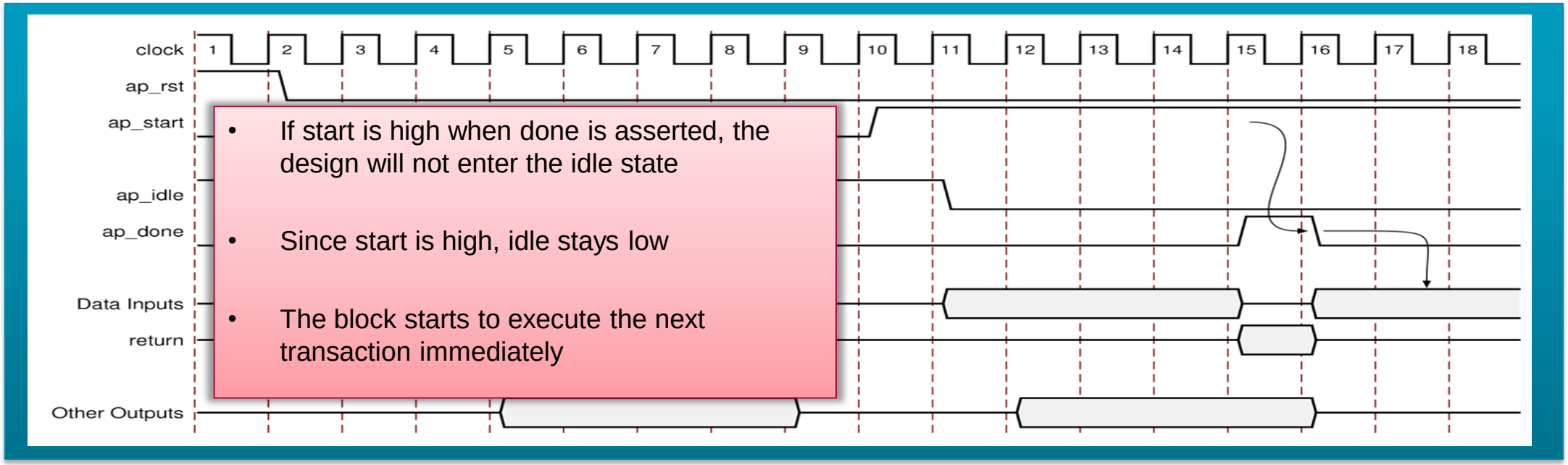
- **By Default ..**
  - Block level handshake signals are added to the design
    - These tell the design when to start operation (ap_start)
    - Indicate when the design has completed (ap_done) and is idle (ap_idle)
    - In a pipelined design, they also indicate when the design can accept new data (ap_ready) : covered later

AP_START: Pulsed



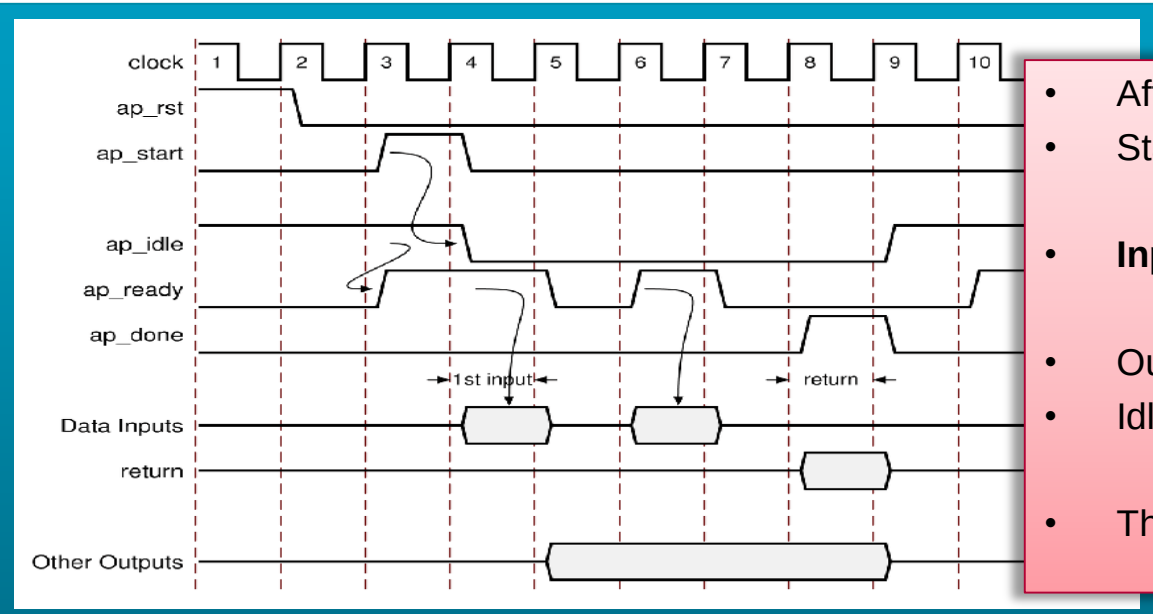
- **Input Data**
  - Can be applied when ap_idle goes low
    - The first read is performed 1 clock cycle after idle goes low
  - Input reads can occur in any cycle up until the last cycle
- **Output Data**
  - Any function return is valid when ap_done is asserted high
  - Other outputs may output their data at any time after the first read
    - The output can only be guaranteed to be valid with ap_done if it is registered
    - It is recommended to use a port level IO protocol for other outputs

AP_START: Constant High



- **Input and Output data operations**
  - As before
- **The key difference here is that the design is never idle**
  - The next data read is performed immediately

Pipelined Designs

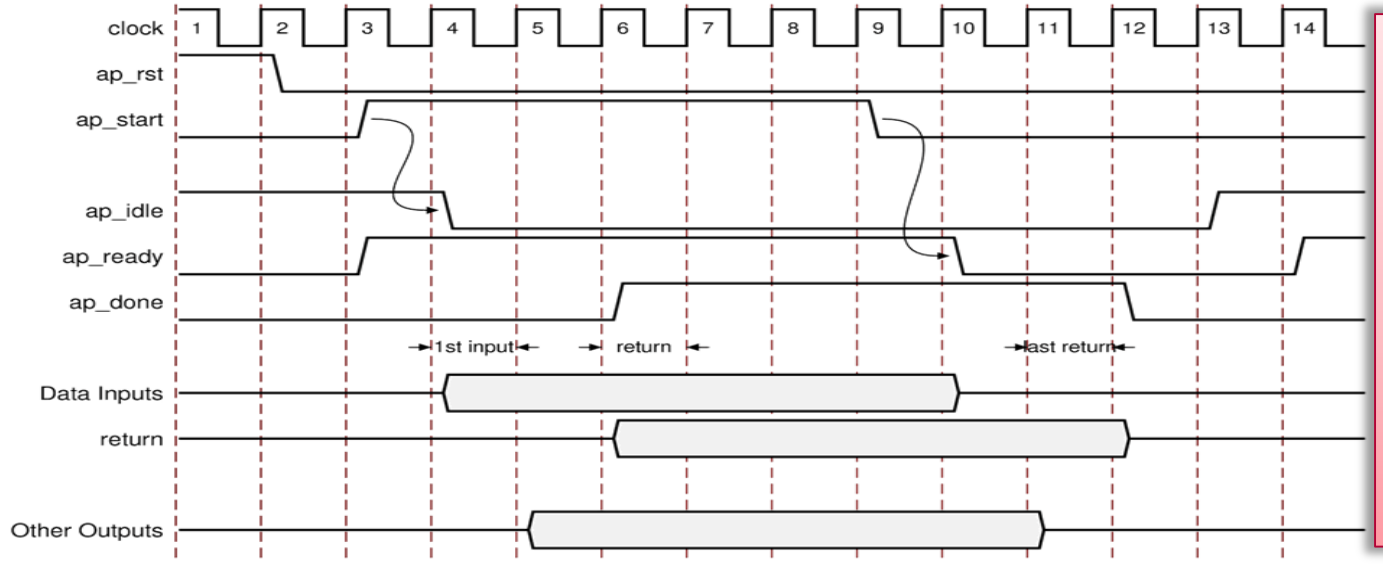


- After reset, ap_ready goes high if idle is high
- Start applied : Idle goes low, apply first data
- **Inputs will be read when ap_ready is high and idle is low**
- Output ap_done goes high on the final cycle
- Idle goes high one cycle after done
- The block waits for next start

- **Input Data**
  - Will be read when ap_ready is high and ap_idle is low
    - Indicates the design is ready for data
  - Signal ap_ready will change at the rate of the throughput
- **Output Data**
  - As before, function return is valid when ap_done is asserted high
  - Other outputs may output their data at any time after the first read
    - It is recommended to use a port level IO protocol for other outputs

This example shows an II of 2

Pipelined Designs: Throughput = 1

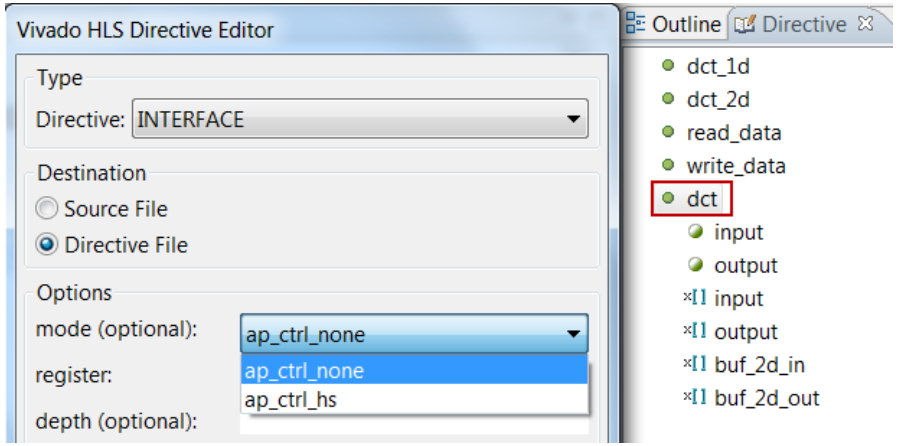


- Will start with ap_start
- Will keep reading input data since throughput = 1
- After the first output, ap_done will remain high
- If ap_start goes is sampled low, ap_ready will go low immediately

- **Input Data when TP=1**
  - It can be expected that ap_ready remains high and data is continuously read
  - The design will only stop processing when ap_start is de-asserted
- **Output Data when TP=1**
  - After the first output, ap_done will remain high while there are samples to process
    - Assuming there is no data decimation (output rate = input rate)

Disabling Block Level Handshakes

- **Block Level Handshakes can be disabled**
  - Select the function in the directives tab, right-click
    - Select Interface & then **ap_ctrl_none** for no block level handshakes
    - Select Interface & then **ap_ctrl_hs** to re-apply the default




```
# Tcl commands

set_directive_interface -mode ap_ctrl_none “dct“
# Default is on
# set_directive_interface -mode ap_ctrl “dct“
```

- **Requirement: Manually verify the RTL**
  - Without block level handshakes autosim cannot verify the design
    - Will only work in simple combo and TP=1 cases
    - Handshakes are required to know when to sample output signals

It is recommended to leave the default and use Block Level Handshakes

Block Level Handshake Summary

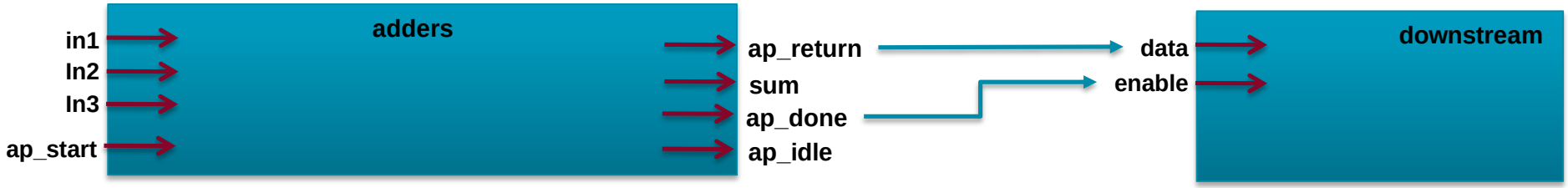
- **Block Level handshakes are added to the RTL design**
  - These are in addition to any data arguments
  - These signals help to interface the RTL block with other blocks
  - Can be optionally disabled
- **Enables system level control & sequencing**
  - Is the only indication (ap_done) the function return (ap_return) is valid for reading
  - Is the only way to prevent data being processed (make ap_start=0)
  - The only overhead is an additional clock cycle at the beginning to sample the start signal
    - It can be tied high and this overhead will only be incurred once, after reset
- **Recommendation**
  - It is safer and more productive to make use of block level handshakes

Outline

- Introduction
- Block Level Protocols
- *Port Level Protocols*
- Summary

Port Level IO protocols

- **We've seen how the function return is validated**
  - The block level output signal ap_done goes high to indicate the function return is valid
  - This allows downstream blocks to correctly sample the output port



- **For other outputs, Port Level IO protocols can be added**
  - Allowing upstream and downstream blocks to synchronize with the other data ports
  - The type of protocol depends on the type of C port
    - Pass-by-value scalar
    - Pass-by-reference pointers (& references in C++)
    - Pass-by-reference arrays

The starting point is the type of argument used by the C function

Let's Look at an Example

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

        int temp;

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

        return  temp;
}
```

"Sum" is a pointer which is read and written to : an Inout

Argument Type	Variable			Pointer Variable			Array			Reference Variable		
	Pass-by- value			Pass-by-reference			Pass-by-reference			Pass-by-reference		
Interface Type	I	IO	O	I	IO	O	I	IO	O	I	IO	O
ap_none	D			D						D		
ap_stable												
ap_ack												
ap_vld						D						D
ap_ovld					D						D	
ap_hs												
ap_memory							D	D	D			
ap_fifo												
ap_bus												
ap_ctrl_none												
ap_ctrl_hs			D									

The port for "Sum" can be any of these interface types

The default for port for "Sum" will be type ap_ovld

Now, let's look at what these interfaces are …

Key:

I	: input
IO	: inout
O	: output
D	: Default Interface

Supported Interface

Unsupported Interface

Default IO Protocols

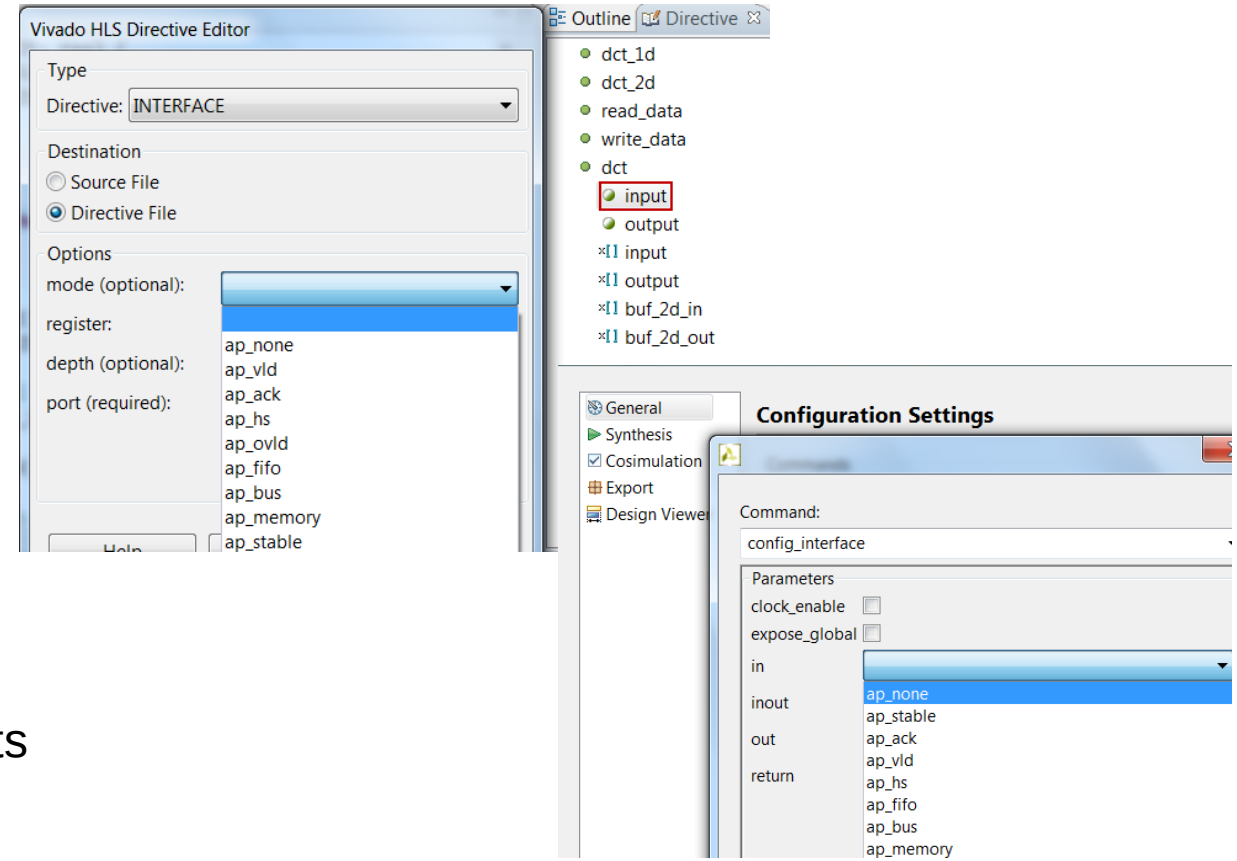
- **The default port protocols**
  - Inputs: ap_none
  - Outputs: ap_vld
  - Inout: ap_ovld
    - In port gets ap_none
    - Out port gets ap_vld
  - Arrays: ap_memory
  - All shown as the default (D) on previous slide
- **Result of the default protocols**
  - No protocol for input ports
    - They should be held stable for the entire transaction
    - There is no way to know when the input will be read
  - Output writes have an accompanying output valid signal which can be used to validate them
  - Arrays will default to RAM interfaces (2-port RAM is the default RAM)

The Vivado HLS shell/console always shows the results of interface synthesis

```
@I [RTGEN-500] Setting IO mode on port 'adders|in1' to 'ap_none'.
@I [RTGEN-500] Setting IO mode on port 'adders|in2' to 'ap_none'.
@I [RTGEN-500] Setting IO mode on port 'adders|in3' to 'ap_none'.
@I [RTGEN-500] Setting IO mode on port 'adders|return' to 'ap_ctrl_hs'.
```

Specifying IO Protocols

- **Select the port in the Directives pane to specify a protocol**
  - Select the port
  - Right-click and choose Interface
  - Select the protocol from the drop-down menu
- **Or apply using Tcl or Pragma**
- **Specify as Configurations**
  - A configuration can be used to set the default
  - Useful when the function has MANY arguments
    - Set the default for <u>all</u> inputs etc.



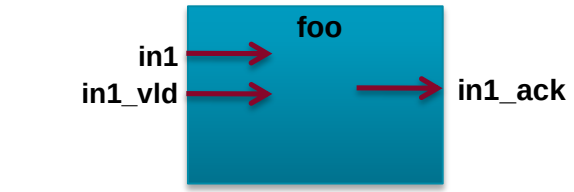
Wire Protocols

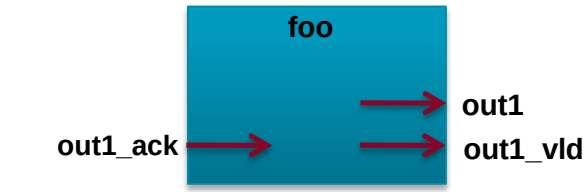
- **The wire protocols add a valid and/or acknowledge port to each data port**
- **The wire protocols are all derivatives of protocol ap_hs**
  - ap_ack: add an acknowledge port
  - ap_vld: add a valid port
  - ap_ovld: add a valid port to an output
  - ap_hs: adds both
- **Output control signals are used to inform other blocks**
  - Data has been read at the input by this block (ack)
  - Data is valid at the output (vld)
  - The other block must accept the control signal (this block will continue)
- **Input control signals are used to inform this block**
  - The output data has been read by the consumer (ack)
  - The input from the producer is valid (vld)
  - **This block will stall** while waiting for the input controls

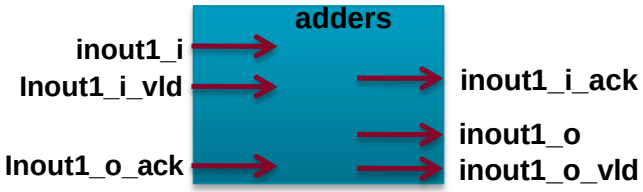
Wire Protocols: Ports Generated

- **The wire protocols are all derivatives of protocol ap_hs**
  - Inputs
    - Arguments which are only read
    - The valid is input port indicating when to read
    - Acknowledge is an output indicating it was read
  - Outputs
    - Arguments which are only written to
    - Valid is an output indicating data is ready
    - Acknowledge is an input indicating it was read
  - Inouts
    - Arguments which are read from and written to
    - These are split into separate in and out ports
    - Each half has handshakes as per Input and Output

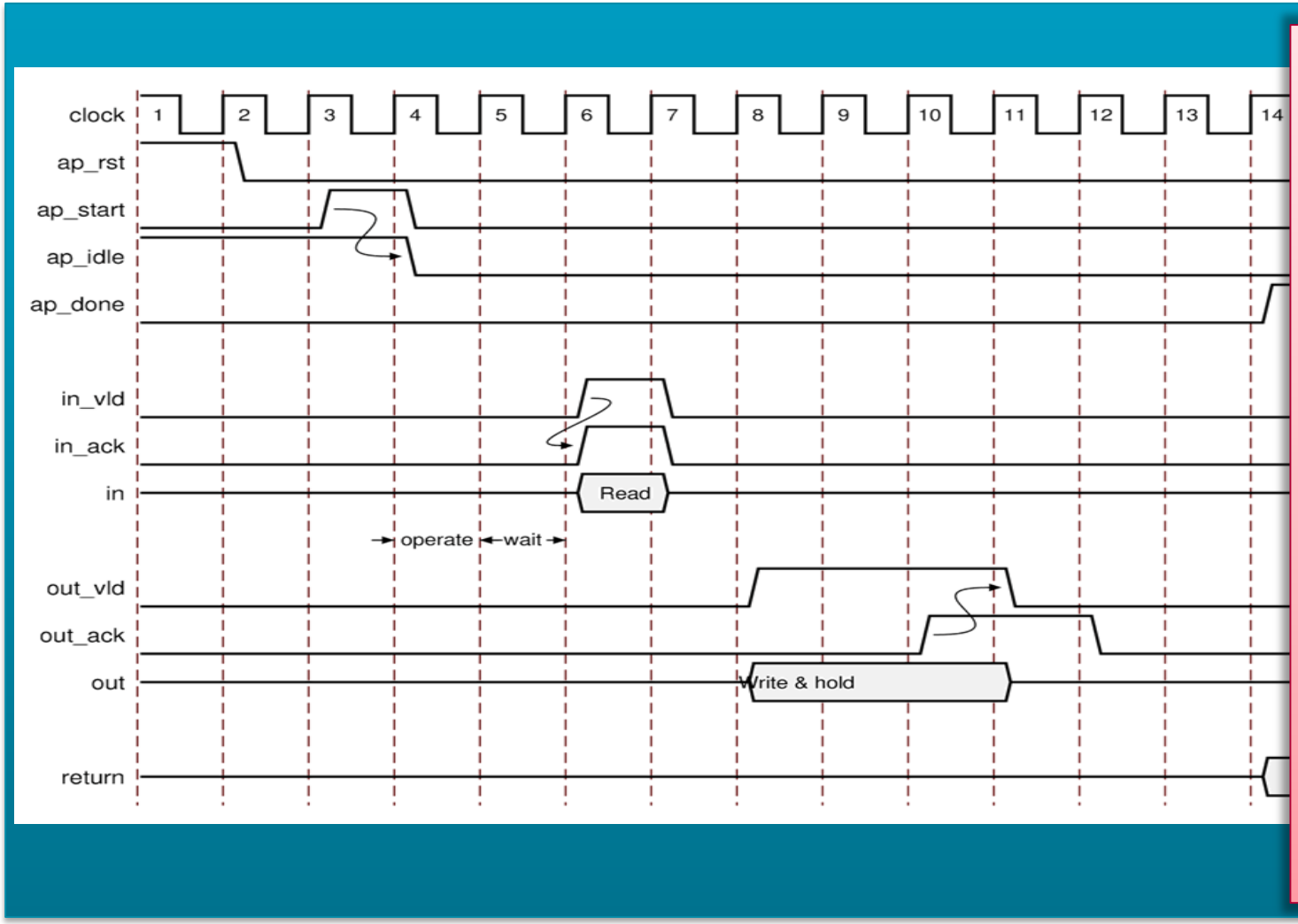
ap_hs is compatible with AXI-Stream







Handshake IO Protocol



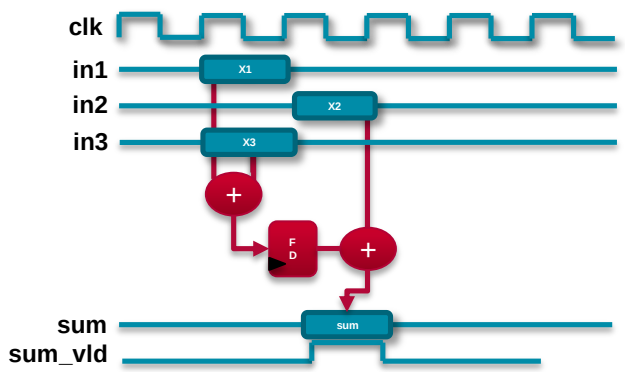
- *After start, idle goes low and the RTL operates <u>until a read is required</u>*
- The RTL will stall (wait) until the input valid is asserted
- If there is more than one input valid, each can stall the RTL
- It will acknowledge on the same cycle it reads the data (reads on next clock edge)
- An output valid is asserted when the port has data
- The RTL will stall (hold the data and wait) until an input acknowledge is received
- *Done will be asserted when the function is complete*

Other Handshake Protocols

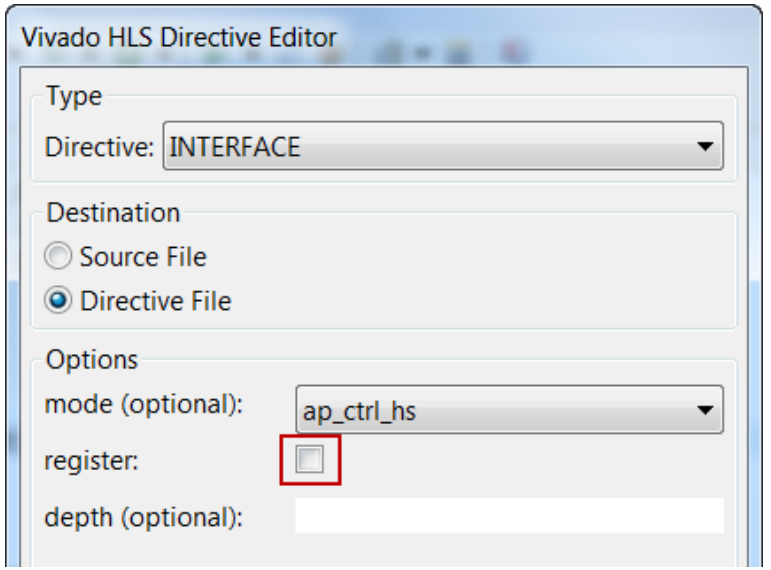
- **The other wire protocols are derivatives of ap_hs in behavior**
  - Some subtleties are worth discussing when the full two-way handshake is **not** used
- **Using the Valid protocols (ap_vld, ap_ovld)**
  - Outputs: Without an associated input acknowledge it is a requirement that the consumer takes the data when the output valid is asserted
    - Protocol ap_ovld only applies to output ports (is ignored on inputs)
  - Inputs: No particular issue.
    - Without an associated output acknowledge, the producer does not know when the data has been read (but the done signal can be used to update values)
- **Using the Acknowledge Protocol (ap_ack)**
  - Outputs: Without an associated output valid the consumer will not know when valid data is ready but the design will stall until it receives an acknowledge (**Dangerous**: Lock up potential)
  - Inputs: Without an associated input valid, the design will simply read when it is ready and acknowledge that fact.

Registering IO Reads and Writes

- **Vivado HLS does not register Input and Outputs by default**
  - It will chain operations to minimize latency
  - The inputs will be read when the design requires them
  - The outputs will be written as soon as they are available



- **Inputs and outputs can be registered**
  - Inputs will be registered in the first cycle
    - Input pointers and partitioned arrays will be registered when they are required
  - Outputs will be registered and held until the next write operation
    - Which for scalars will be the next transaction (can't write twice to the same port) unless the block is pipelined



Wire Handshake Summary

- **Wire Handshakes are flexible and the defaults**
  - They can be used for all C arguments except arrays
  - They are the default for all C arguments except arrays
  - Inputs default to none, outputs default to output valid
- **Input valid and acknowledge signals will stall the RTL**
  - The RTL will wait until an input is valid
  - After writing an output to the port, it will wait for the output to be acknowledged
  - Any port can stall the designs : each can block
- **Recommendation**
  - It is safer and more productive to make use of full two-way handshake
    - It's also similar to AXI4-Steam
  - Be very careful about using an acknowledge only protocol on output ports

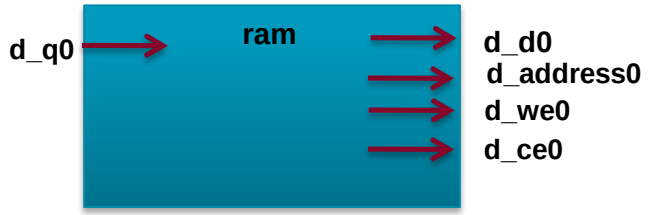
Memory IO Protocols

- **Memory protocols can be inferred from array and pointer arguments**
  - Array arguments can be synthesized to RAM or FIFO ports
    - When FIFOs specified on array ports, the ports must be read-only or write-only
  - Pointer (and References in C++) can be synthesized to FIFO ports
- **RAM ports**
  - Support arbitrary/random accesses
  - May be implemented with Dual-Port RAMs to increase the bandwidth
    - The default is a Dual-Port RAM interface
  - Requires two cycles read access: generate address, read data
    - Pipelining can reduce this overhead by overlapping generation with reading
- **FIFO ports**
  - Require read and writes to be sequential/streaming
  - Always uses a standard FIFO (single-port) model
  - Single cycle for both reads and writes

Memory IO Protocols: Ports Generated

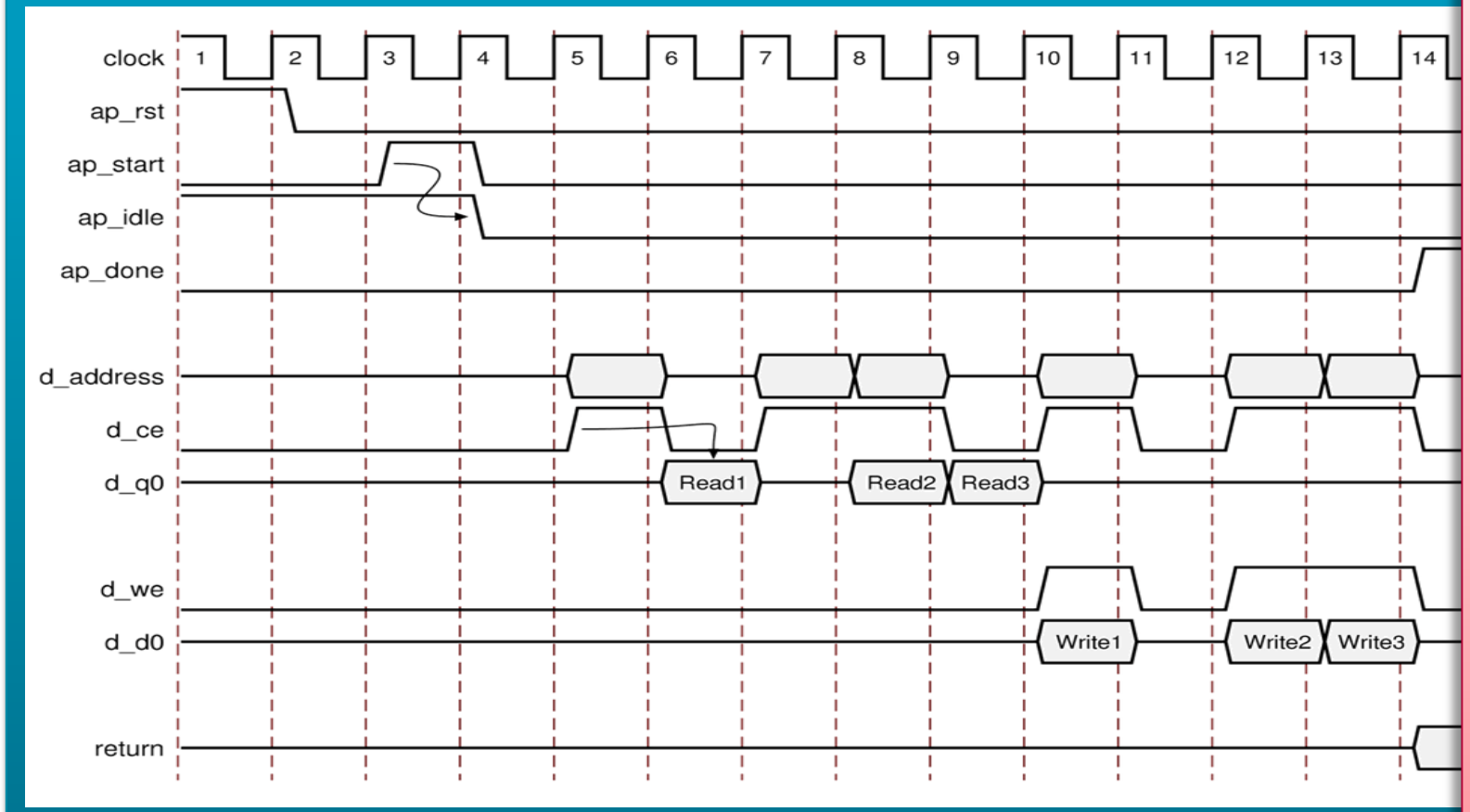
- **RAM Ports**
  - Created by protocol ap_memory
  - Given an array specified on the interface
  - Ports are generated for data, address & control
    - Example shows a single port RAM
    - A dual-port resource will result in dual-port interface
  - Specify the off-chip RAM as a Resource
    - Use the RESOURCE directive on the array port
- **FIFO Ports**
  - Created by protocol ap_fifo
  - Can be used on arrays, pointers and references
  - Standard Read/Write, Full/Empty ports generated
    - Must use separate arrays for read and write
    - Pointers/References: split into In and Out ports

```
#include "ram.h"
void ram (int d[DEPTH], …) {
  …
}
```




```
#include "fifo.h"
void fifo (int d_o[DEPTH],
           int d_i[DEPTH]) {

  …
}
```

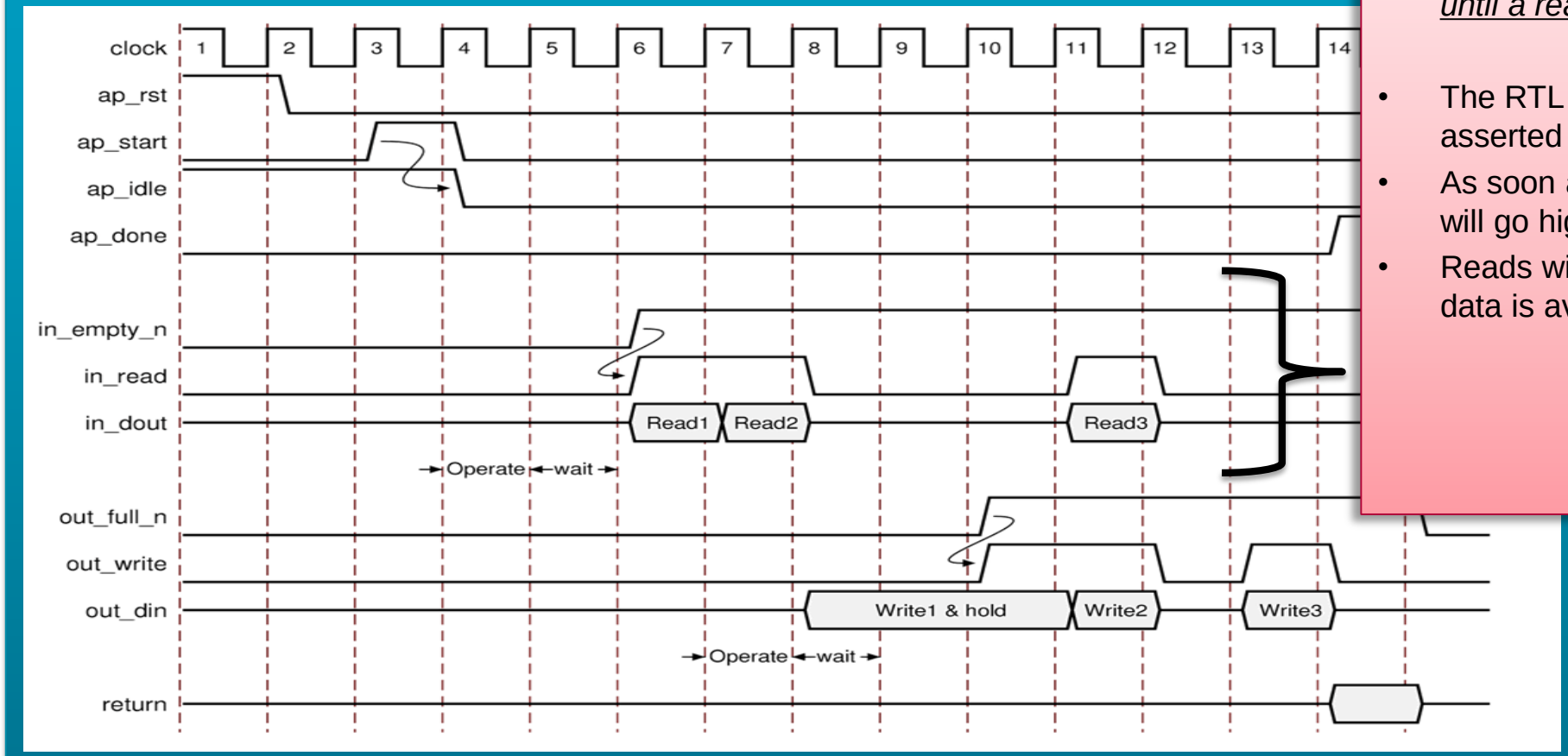
Memory IO Protocol



- *After start, idle goes low and the RTL operates*
- When a read is required, an address is generated and CE asserted high
- Data is available on data input port d_q0 in the next cycle
- The read operations may be pipelined
- When a write is required, the address and data are placed on the output ports
- Both CE &WE are asserted high
- Writes may be pipelined
- *Done will be asserted when the function is complete*

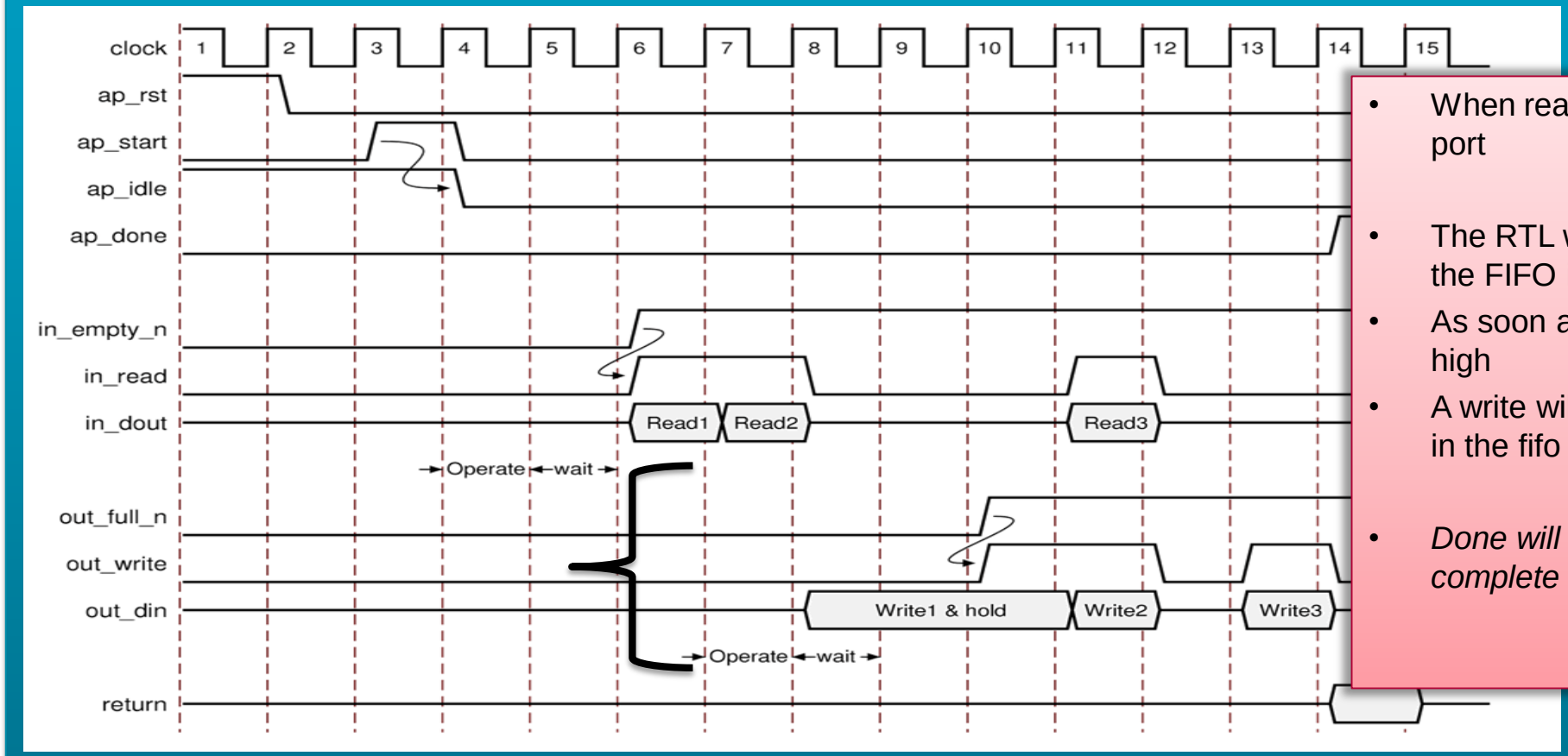
There is no stall behavior initiated by a RAM interface

FIFO IO Protocol (Read)



- *After start, idle goes low and the RTL operates <u>until a read is required</u>*
- The RTL will stall (wait) until the empty_n is asserted high to indicate data is available
- As soon as data is available, the fifo read port will go high
- Reads will occur when required, so long as data is available (empty_n is high)

FIFO IO Protocol (Write)



- When ready, data will be written to the output port
- The RTL will then stall and hold the data until the FIFO is no longer full (full_n high)
- As soon as data can written, write is asserted high
- A write will occur when required if there is room in the fifo (full_n high)
- *Done will be asserted when the function is complete*

Memory IO Protocol Summary

- **Vivado HLS can interface with both RAMs and FIFOs**
  - Code changes *may* be required
  - Only array arguments can be implemented as RAM ports
  - Only array, pointer and references can be implemented as FIFO ports
    - If input scalars are used, use ap_hs if streaming is required
  - If an array argument is to be implemented as a FIFO, separate read and write arrays are required
- **RAM ports**
  - The ports are defined by the RAM resource assigned to the array
  - Single and dual-port RAMs are available
- **FIFO ports**
  - Read and Write operations on FIFO ports must be sequential
  - FIFO ports can stall the RTL design
    - If the FIFOs are empty or full

Bus IO Protocol

- **Vivado HLS supports a Bus IO protocol**
  - The IO protocol is that of a generic bus
  - The IO protocol is not an industry standard
  - The Vivado HLS bus protocol is principally used in the Pcore flow, allowing Vivado HLS to connect to adapters and industry standard buses
- **The Bus IO protocol supports memcpy**
  - The bus IO protocol supports the C function memcpy
  - This provides a high performance interface for bursting data in a DMA like fashion
- **The Bus IO protocol supports complex pointer arithmetic at the IO**
  - Pointers at the IO can be synthesized to ap_fifo or ap_bus
  - If using ap_fifo, the accesses must be sequential
  - If pointer arithmetic is used, the port must use ap_bus

Standard and Burst Mode

Standard Mode

- Each access to the bus results in a request then a read or write operation
- Multiple read or writes can be performed in a single transaction

Single read and write in Standard Mode

```
void foo (int *d) {
  static int acc = 0;
  int i;

    acc += d[i];
    d[i] = acc;
}
```

Multiple reads and writes in Standard Mode

```
#in
void foo (int *d) {
  static int acc = 0;
  int i;

  for (i=0;i<4;i++) {
    acc += d[i];
    d[i] = acc;
  }
}
```

```
void foo (int *d) {
  static int acc = 0;
  int i;

  for (i=0;i<4;i++) {
    acc += *(d+i);
    *(d+i) = acc;
  }
}
```

Burst Mode

- Use the memcpy command
- Copies data between array & a pointer argument
- The pointer argument can be a bus interface
  - This example uses a size of 4
  - This is more efficient for higher values

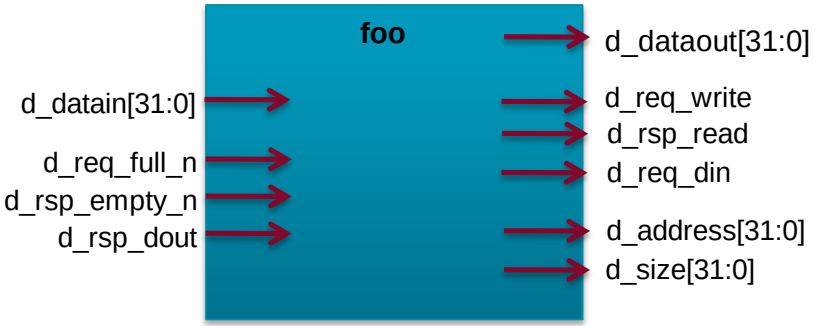
Burst Mode

```
void foo (int *d) {
  int buf1[4], buf2[4];
  int i;

  memcpy(buf1,d,4*sizeof(int));

  for (i=0;i<4;i++) {
    buf2[i] = buf1[3-i];
  }

  memcpy(d,buf2,4*sizeof(int));
}
```

Bus IO Protocol: Ports Generated

- **Bus request then access protocol**
  - The protocol will request a read/write bus access
  - Then read or write data in single or burst mode

```
#include “foo.h"

void foo (int *d) {
  …
}
```

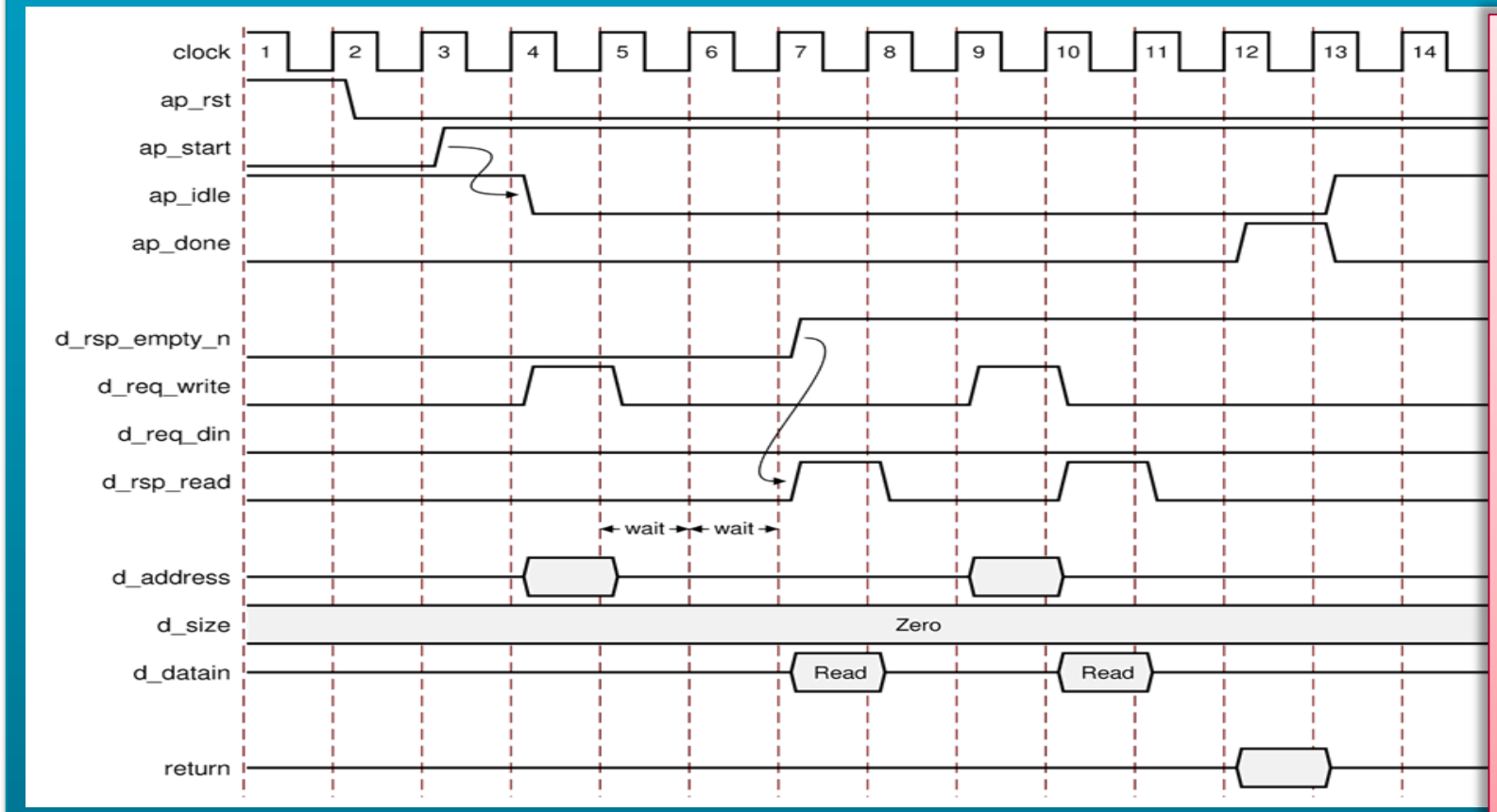


Input Ports	Description
d_datain	Input data.
d_req_full_n	Active low signal indicates the bus bridge is full. The design will stall, waiting to write.
d_rsp_empty_n	Active low signal indicates the bus bridge is empty and can accept data.

Output Ports	Description
d_dataout	Output data.
d_req_write	Asserted to initiate a bus access.
d_req_din	Asserted if the bus access is to write. Remains low if the access it to read.
d_rsp_read	Asserted to start a bus read (completes a bus access request and starts reading data)
d_address	Offset for the base address.
d_size	Indicates the burst (read or write) size.

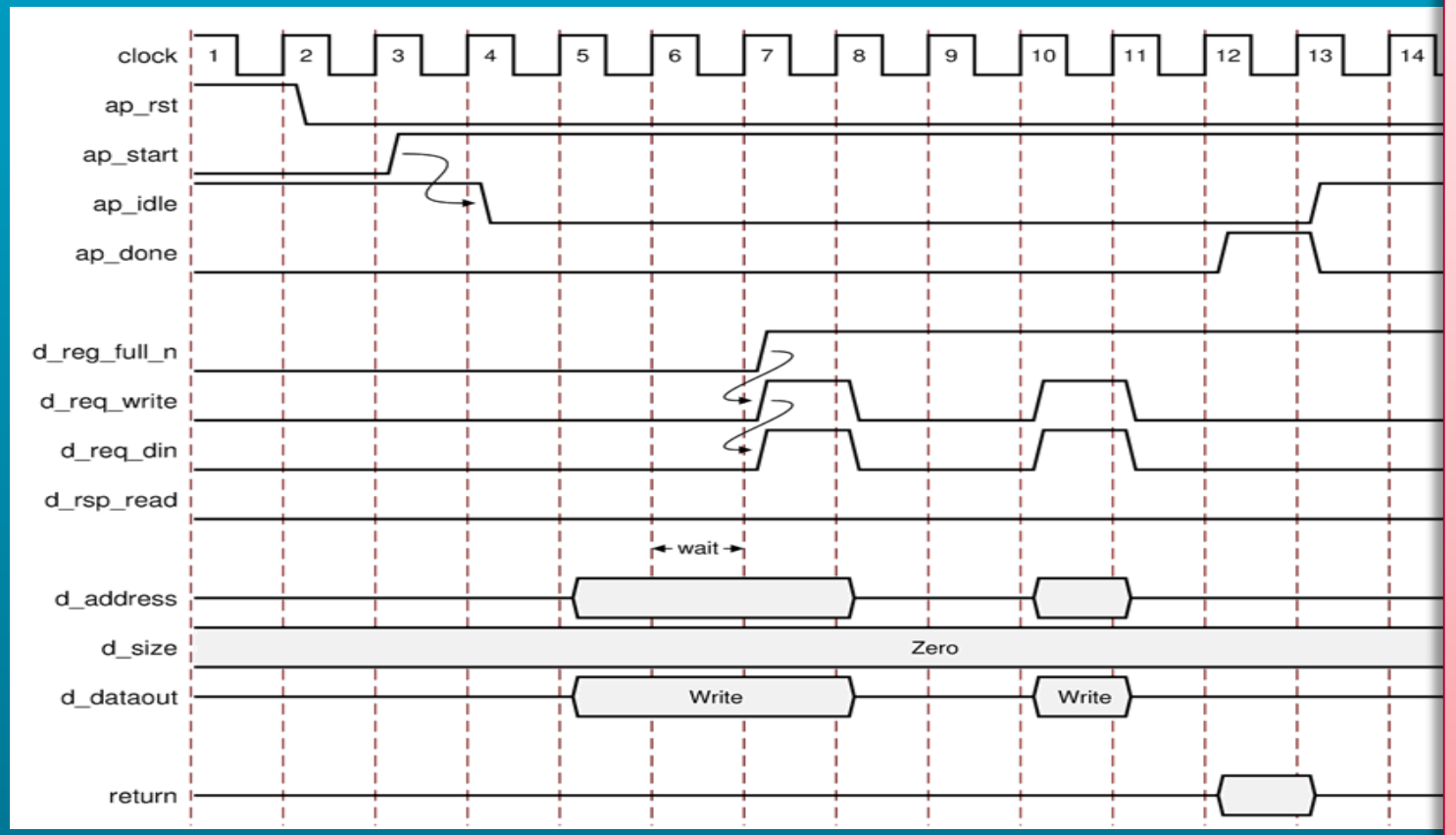
Standard Mode: The address port gives the index address e.g. *(d+i), addr = value of “i”

Bus IO Protocol (Standard, Read)



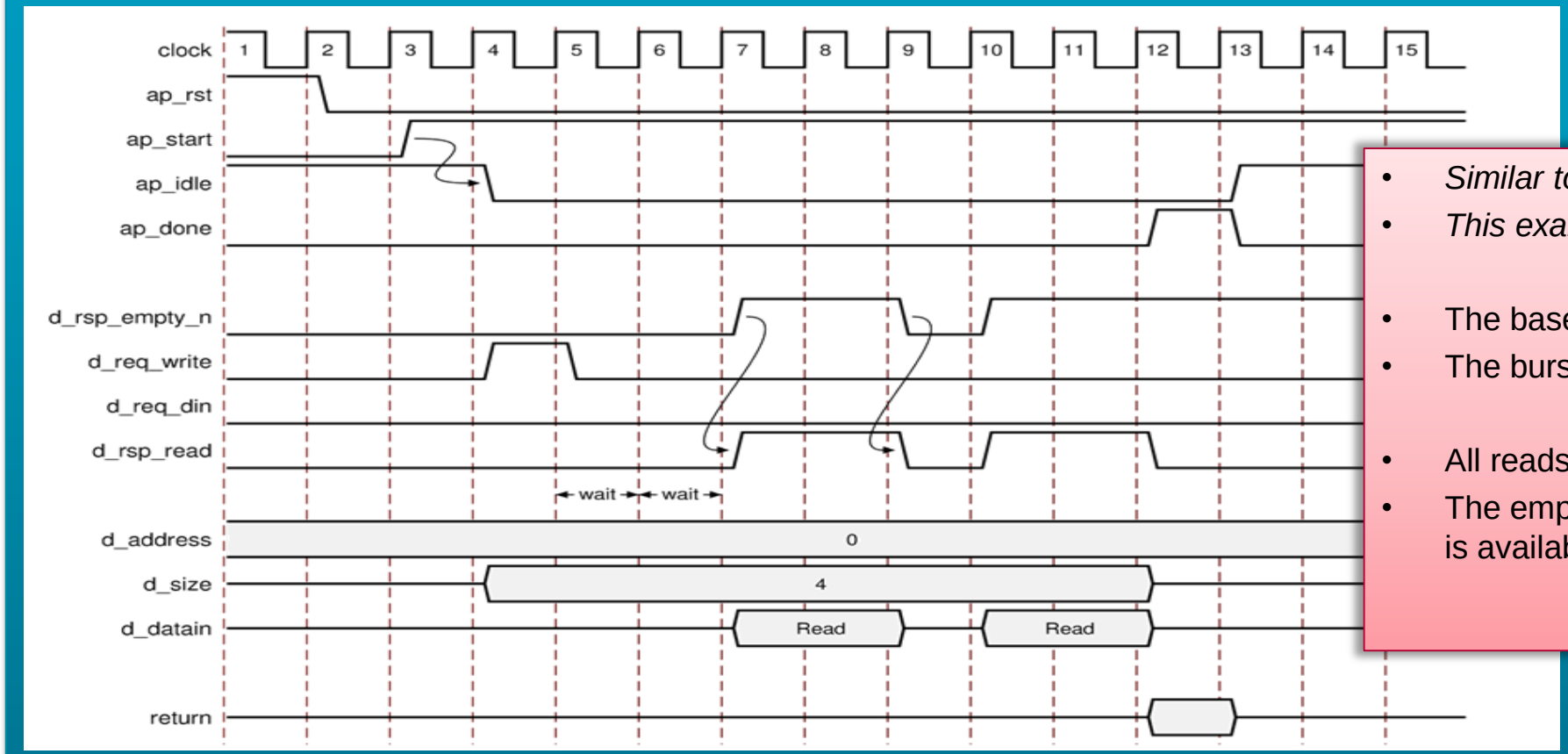
- *After start, idle goes low and the RTL operates <u>until a read is required</u>*
- A req_write high with req_din low indicates a read request
- A read address is supplied: value is the pointer index value
- The RTL will stall (wait) until the empty_n is asserted high to indicate data is available
- As soon as data is available, rsp_read will go high
- Reads will occur when required, so long as data is available (empty_n is high)

Bus IO Protocol (Standard, Write)



- *After start, idle goes low and the RTL operates <u>until a write is required</u>*
- When a write is required, data and address are applied
- The RTL will stall (wait) until the full_n is asserted high to indicate space is available
- As soon as data is available, rsp_write and req-din will go high
- A req_write high with req_din high indicates a write request
- There is no acknowledge for writes in this interace
- Write will occur when required, so long as data is available (full_n is high)

Bus IO Protocol (Burst, Read)

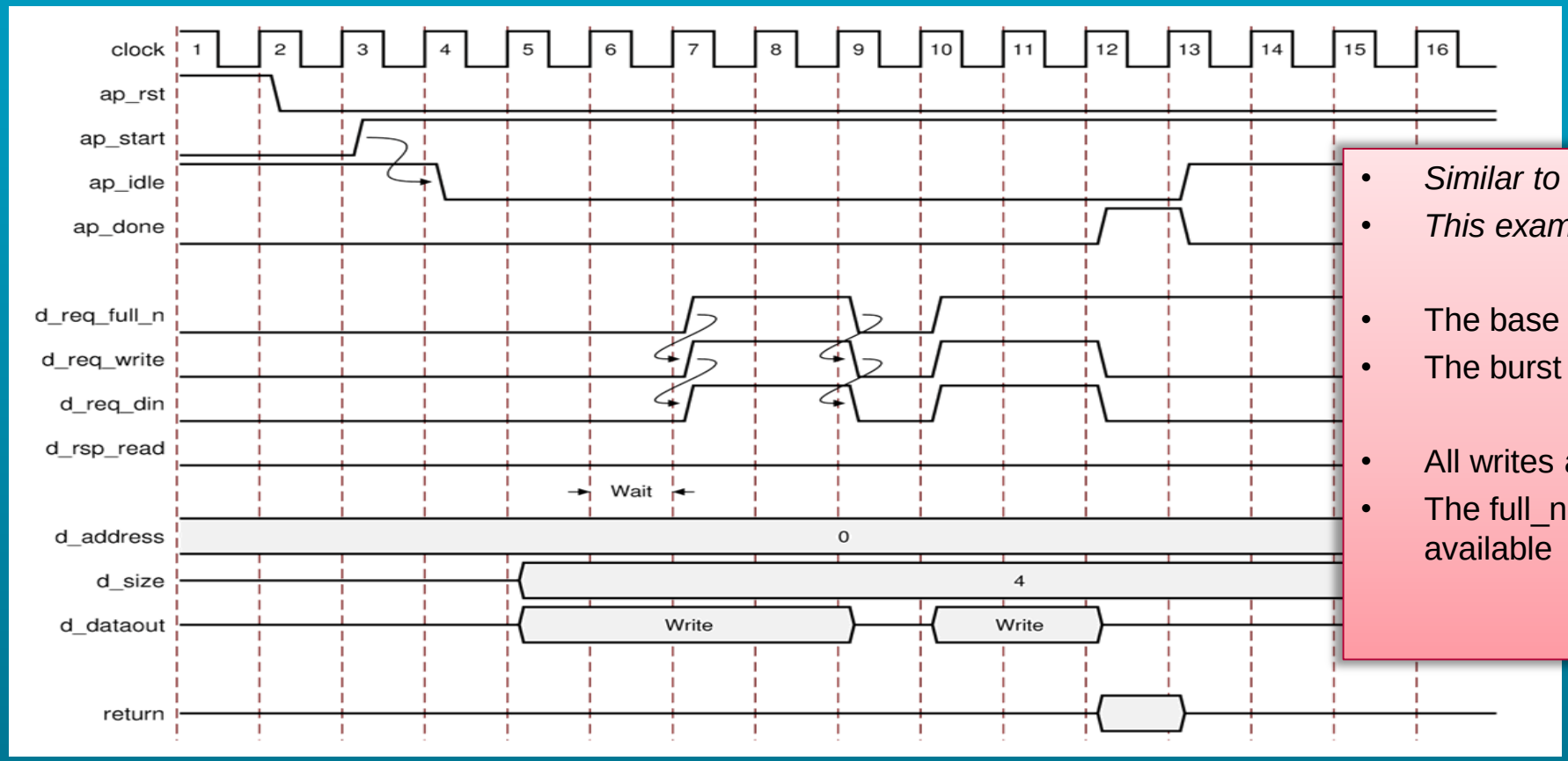


- *Similar to a Standard Mode*
- *This example uses a burst=4*

- The base address is zero
- The burst size is placed on port size

- All reads are done consecutively
- The empty_n signal will stall the RTL until data is available

Bus IO Protocol (Burst, Write)



- *Similar to a Standard Mode*
- *This example uses a burst=4*

- The base address is zero
- The burst size is placed on port size

- All writes are done consecutively
- The full_n signal will stall the RTL until data is available