High-Level Synthesis

Xilinx Vivado HLS

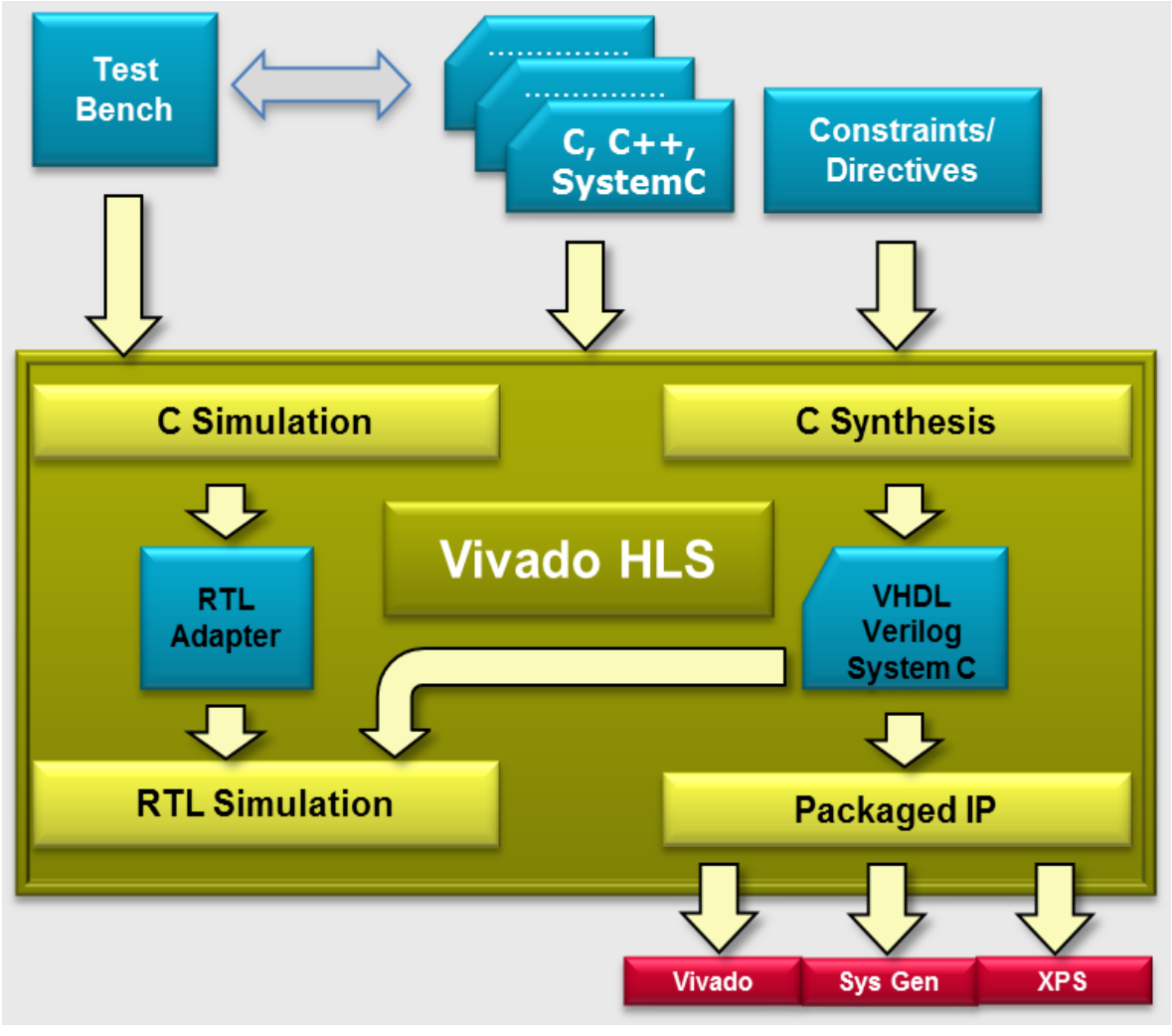
Hao Zheng
Comp Sci & Eng
University of South Florida

Reading

- **The Zynq Book**, chapter 14, 15

- **Vivado Design Suite Tutorial: High-Level Synthesis**

Overview


Test Bench
C, C++, SystemC
Constraints/ Directives
C Simulation
C Synthesis
RTL Adapter
Vivado HLS
VHDL Verilog System C
RTL Simulation
Packaged IP
Vivado
Sys Gen
XPS

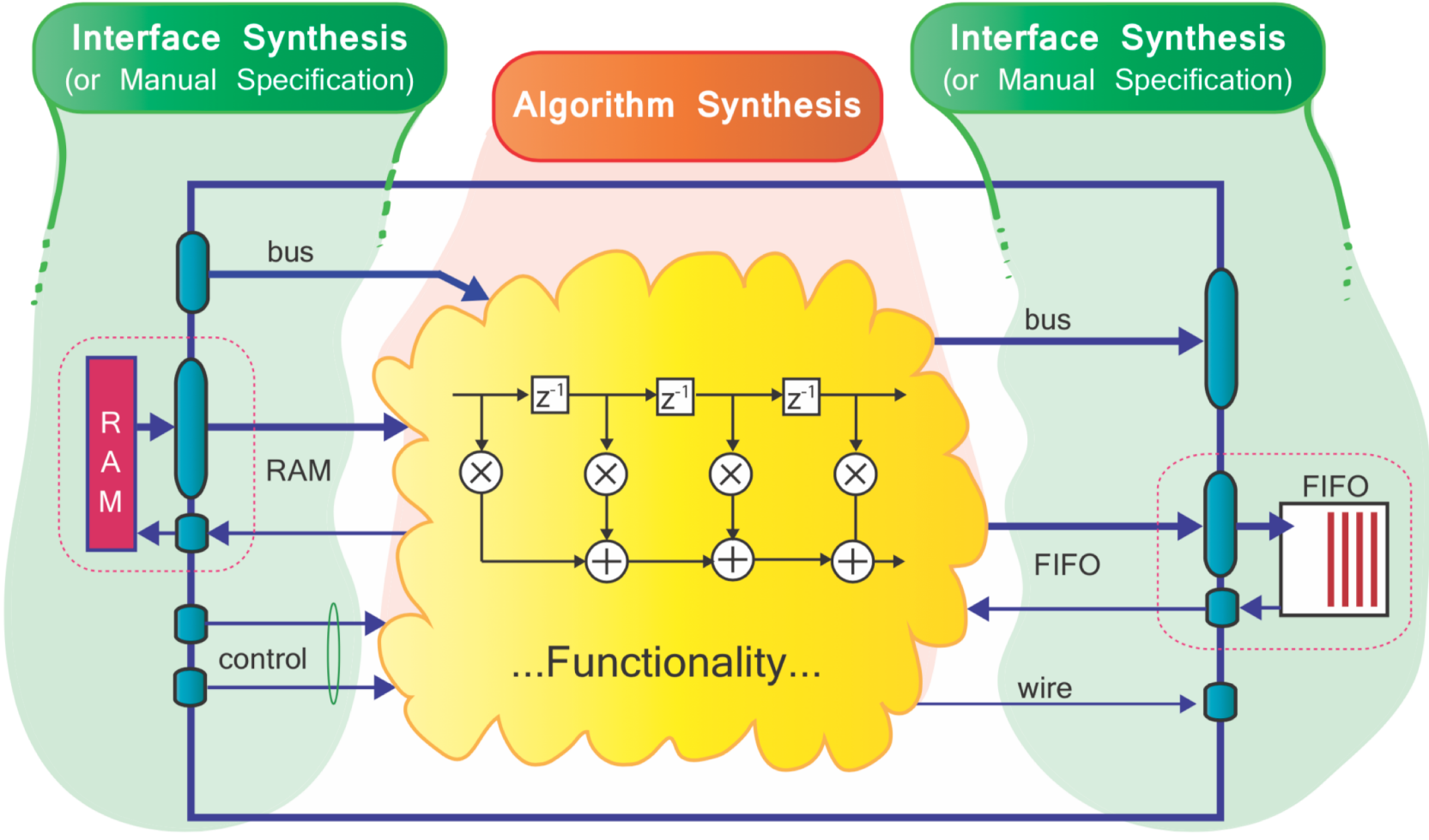
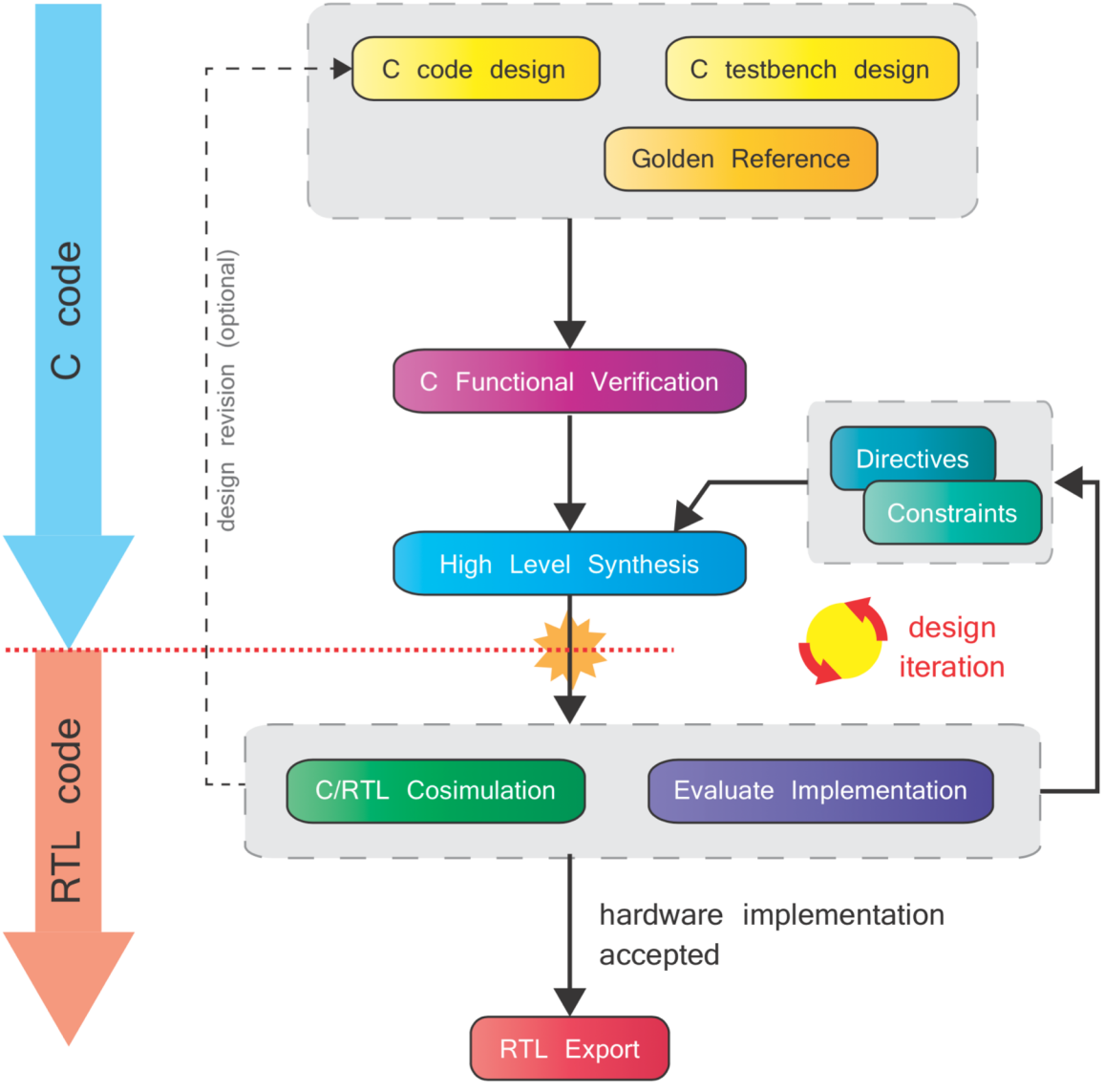


Figure 14.4: Clarification of the algorithm and interface, and showing a subset of interface types


C code
RTL code
design revision (optional)
C code design
C testbench design
Golden Reference
C Functional Verification
Directives
Constraints
High Level Synthesis
design iteration
C/RTL Cosimulation
Evaluate Implementation
hardware implementation accepted
RTL Export

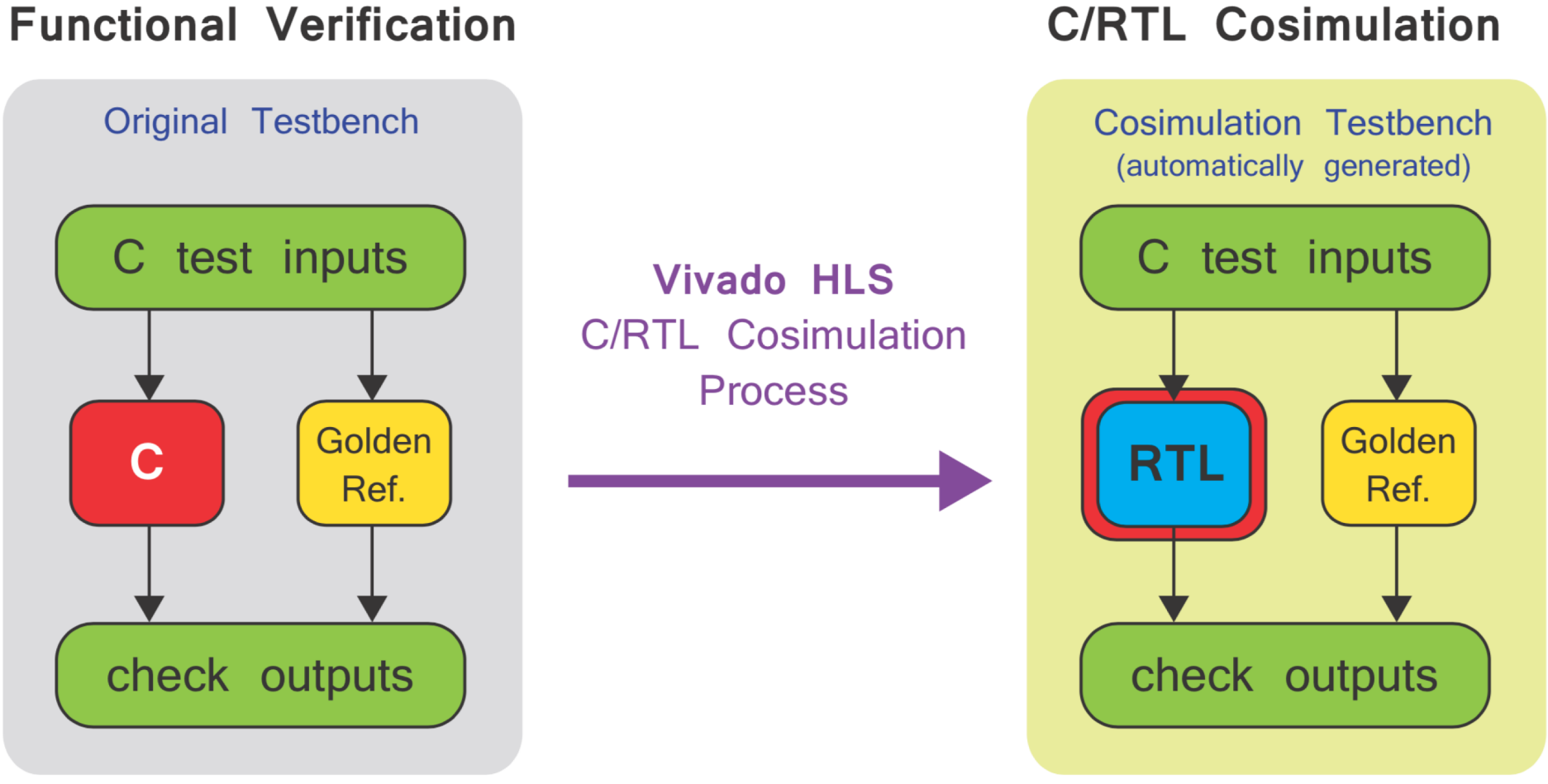


Figure 14.6: C functional verification and C/RTL cosimulation in Vivado HLS

Implementation Considerations

- → ***Resources / area***
- → ***Throughput***
- → ***Clock frequency***
- → ***Latency***
- → ***Power consumption***
- → ***I/O requirements***

Controlled by synthesis directives

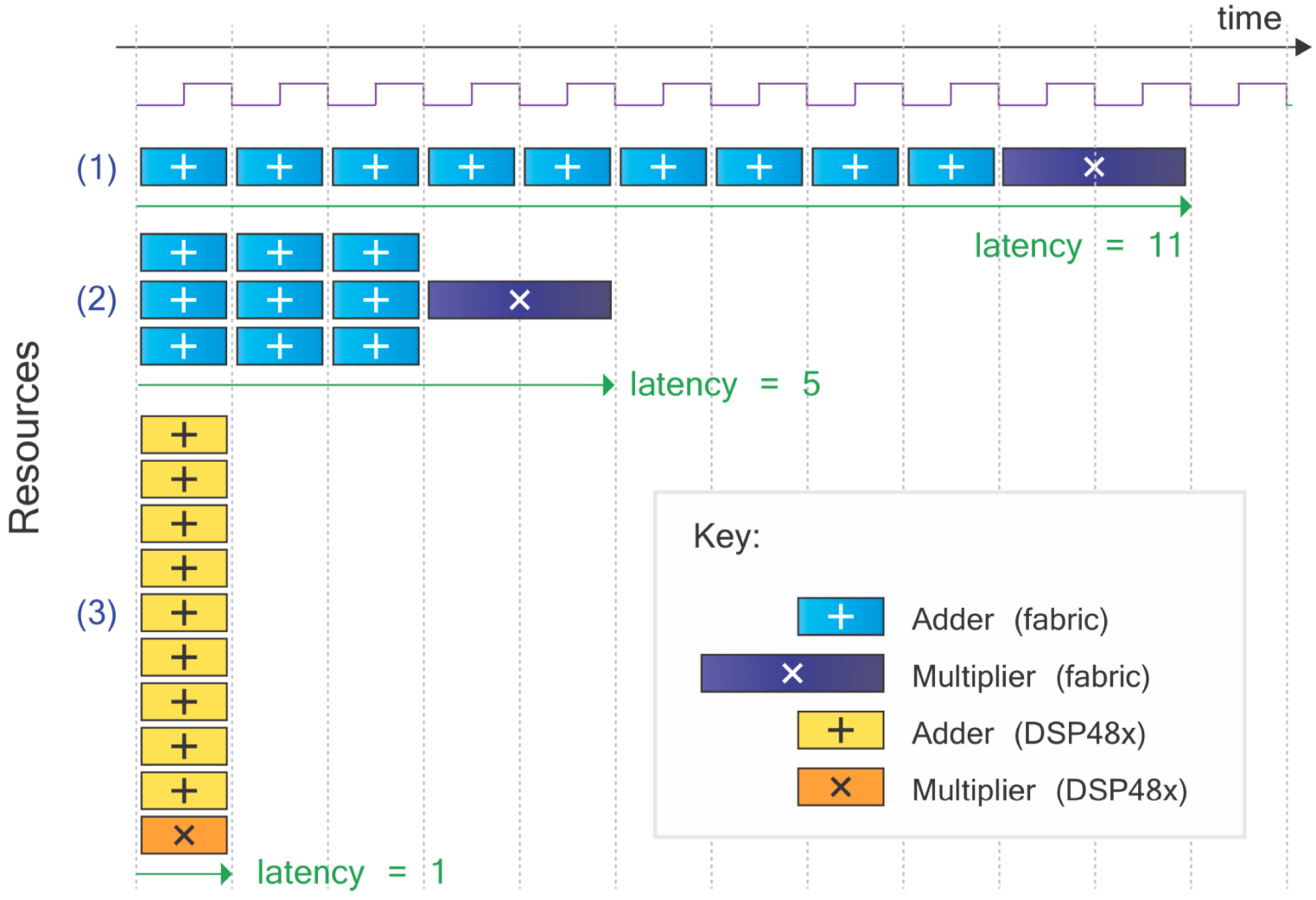


Figure 14.8: Comparison of three possible outcomes from HLS for an example function

Native Types in C/C++

Type	Description	Number of Bits[a]	Range[b]
`char`	Representation of the basic character set.	8	-128 to 127
`signed char`		8	-128 to 127
`unsigned char`		8	0 to 255
`short int`	A reduced precision version of `int`, requiring less storage.	16	-32,768 to 32,767
`unsigned short int`		16	0 to 65,535
`int`	The basic integer data type.	32	-2,147,483,648 to 2,147,483,647
`unsigned int`		32	0 to 4,294,967,295
`long int`	In many cases the `long int` type will be the same length as `int`, i.e. 32 bits.	32	-2,147,483,648 to 2,147,483,647
`unsigned long int`		32	0 to 4,294,967,295
`long long int`	An extended precision integer type.	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned long long int`		64	0 to 18,446,744,073,709,551,615
`float`	Single precision floating point (IEEE 754)	32	$-3.403e^{+38}$ to $3.403e^{+38}$
`double`	Double precision floating point (IEEE 754)	64	$-1.798e^{+308}$ to $1.798e^{+308}$

Arbitrary Precision – Integer

Table 15.2: Arbitrary precision integer data types for use in C and C++ Vivado HLS designs

Language	Integer Data Type	Description	Required Header
C	int*N* (e.g. `int7`)	signed integer of *N* bits precision	`#include "ap_cint.h"`
	uint*N* (e.g. `uint7`)	unsigned integer of *N* bits precision	
C++	ap_int<*N*> (e.g. `ap_int<7>`)	signed integer of *N* bits precision	`#include "ap_int.h"`
	ap_uint<*N*> (e.g. `ap_uint<7>`)	unsigned integer of *N* bits precision	

1 ≤ N ≤ 1024

Typical C/C++ Construct to RTL Mapping

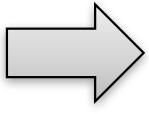
C Constructs		HW Components
Functions	→	**Modules**
Arguments	→	**Input/output ports**
Operators	→	**Functional units**
Scalars	→	**Wires or registers**
Arrays	→	**Memories**
Control flows	→	**Control logics**

Function Hierarchy

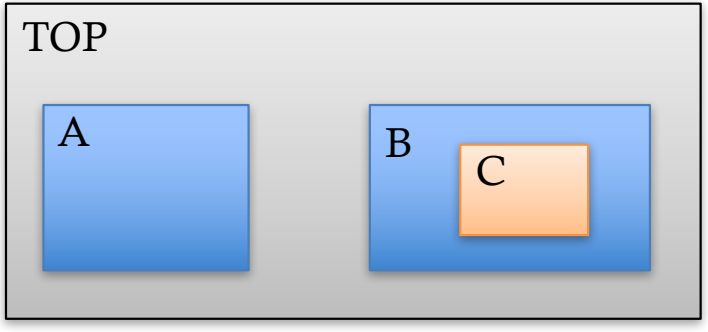
- Each function is synthesized to a RTL module
  - Function inlining eliminates hierarchy
- The function **main()** cannot be synthesized
  - Used to develop C-testbench

Source code

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

void TOP( ) {
        A(…);
        B(…);
}
```



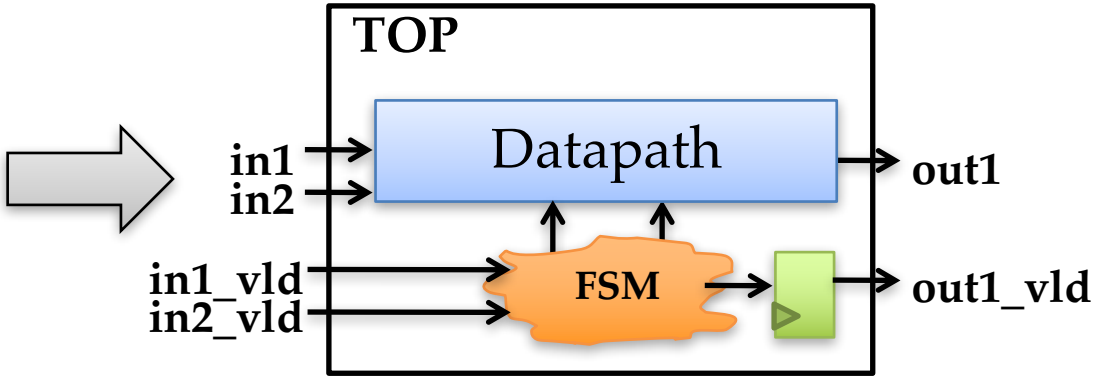
RTL hierarchy



Function Arguments

- Function arguments become module ports
  - Interface follows certain protocol to synchronize data exchange

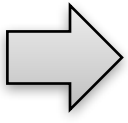
```
void TOP(int* in1, int* in2,
         int* out1)
{
    *out1 = *in1 + *in2;
}
```

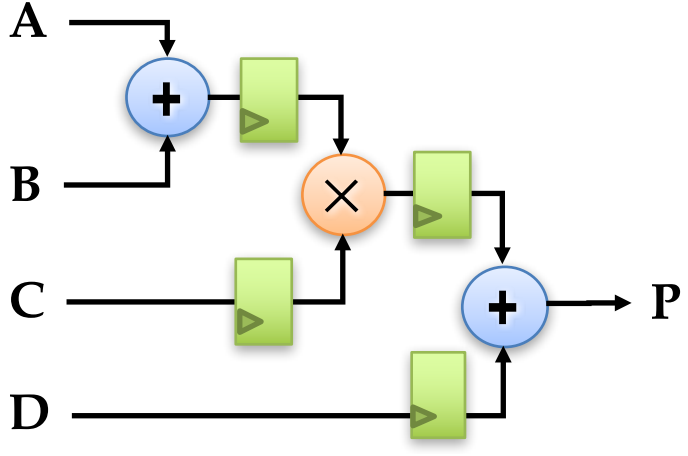


Expressions

- Expressions and operations are synthesized to datapath components
  - Timing constraints influence the degree of registering

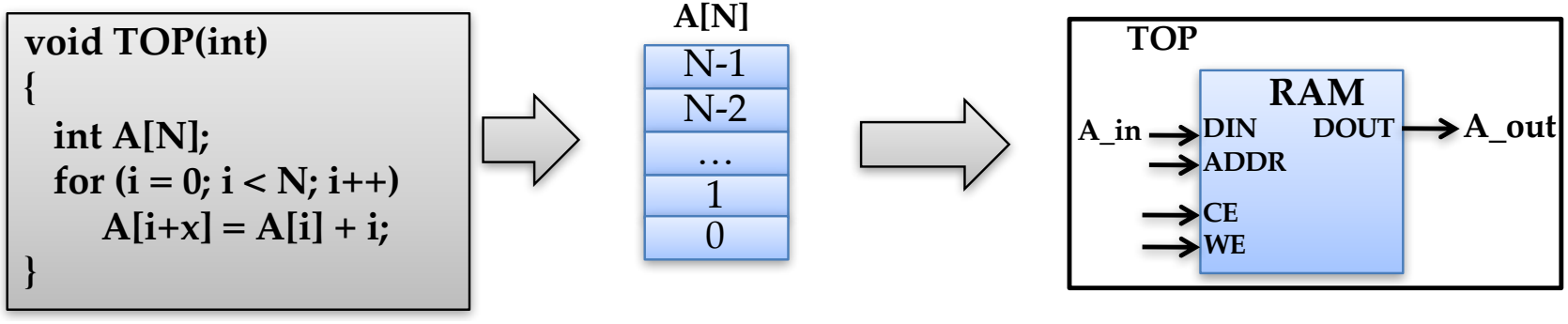
```
char A, B, C, D,
int P;

P = (A+B)*C+D
```





Arrays

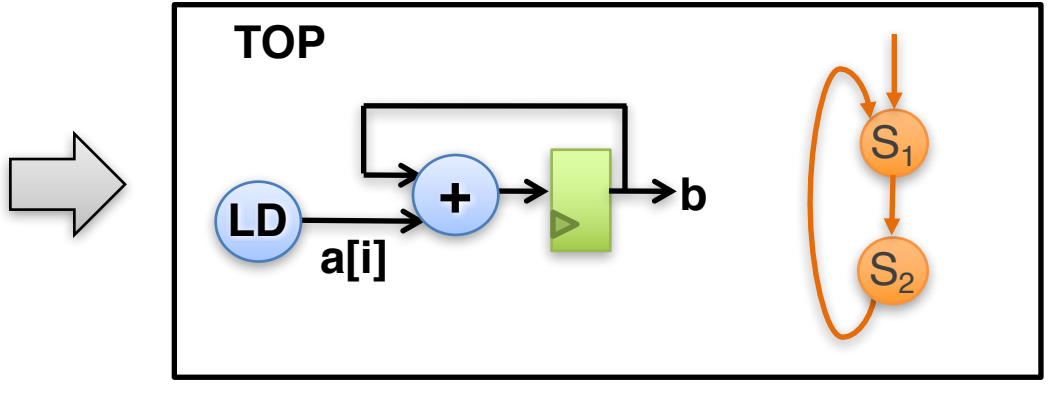
- An array is typically implemented by a mem block
  - Read & write array -> RAM; Constant array -> ROM
- An array can be partitioned and map to multiple RAMs
- Multiples arrays can be merged and map to one RAM
- An array can be partitioned into individual elements and map to registers



Loops

- By default, loops are rolled
  - Each loop iteration corresponds to a “sequence” of states (possibly a DAG)
  - This state sequence will be repeated multiple times based on the loop trip count

```
void TOP (…) {
  …
  for (i = 0; i < N; i++)
       b += a[i];
}
```



Loop Unrolling

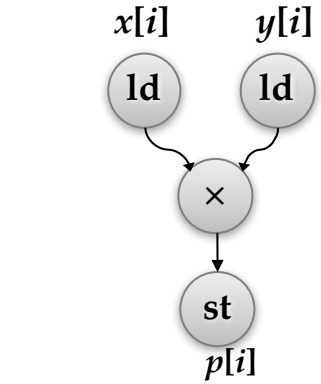
- To expose higher parallelism and achieve shorter latency
  - Pros
    - Decrease loop overhead
    - Increase parallelism for scheduling
    - Facilitate constant propagation and array-to-scalar promotion
  - Cons – increase operation count, which may negatively impact area,

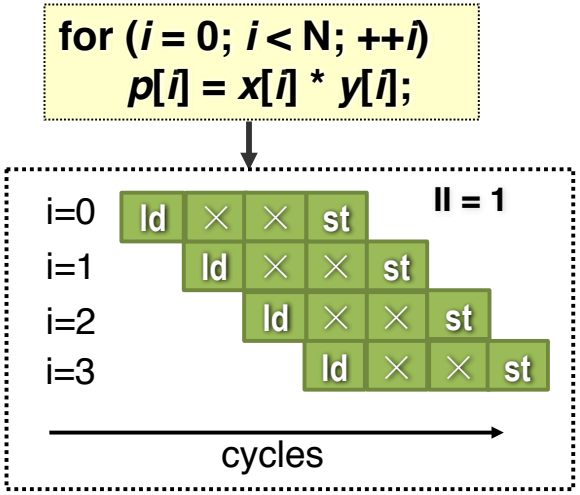
```
for (int i = 0; i < N; i++)
  A[i] = C[i] + D[i];
```

```
A[0] = C[0] + D[0];
A[1] = C[1] + D[1];
A[2] = C[2] + D[2];
.....
```

Loop Pipelining

- Loop pipelining is one of the most important optimizations for high-level synthesis
  - Allows a new iteration to begin processing before the previous iteration is complete
  - Key metric: **Initiation Interval (II)** in # cycles





Synthesis of Loops – Case Study

```
void add_array (short c[12], short a[12], short b[12])
{
  short j;                                  // loop variable

  add_loop: for (j=0;j<12;j++) {     // loop through elements (x12)
        c[j] = a[j] + b[j];          // addition operation
    }
}
```

By default, Vivado intends to optimize area, so loops are rolled

Synthesis of Loops – Case Study

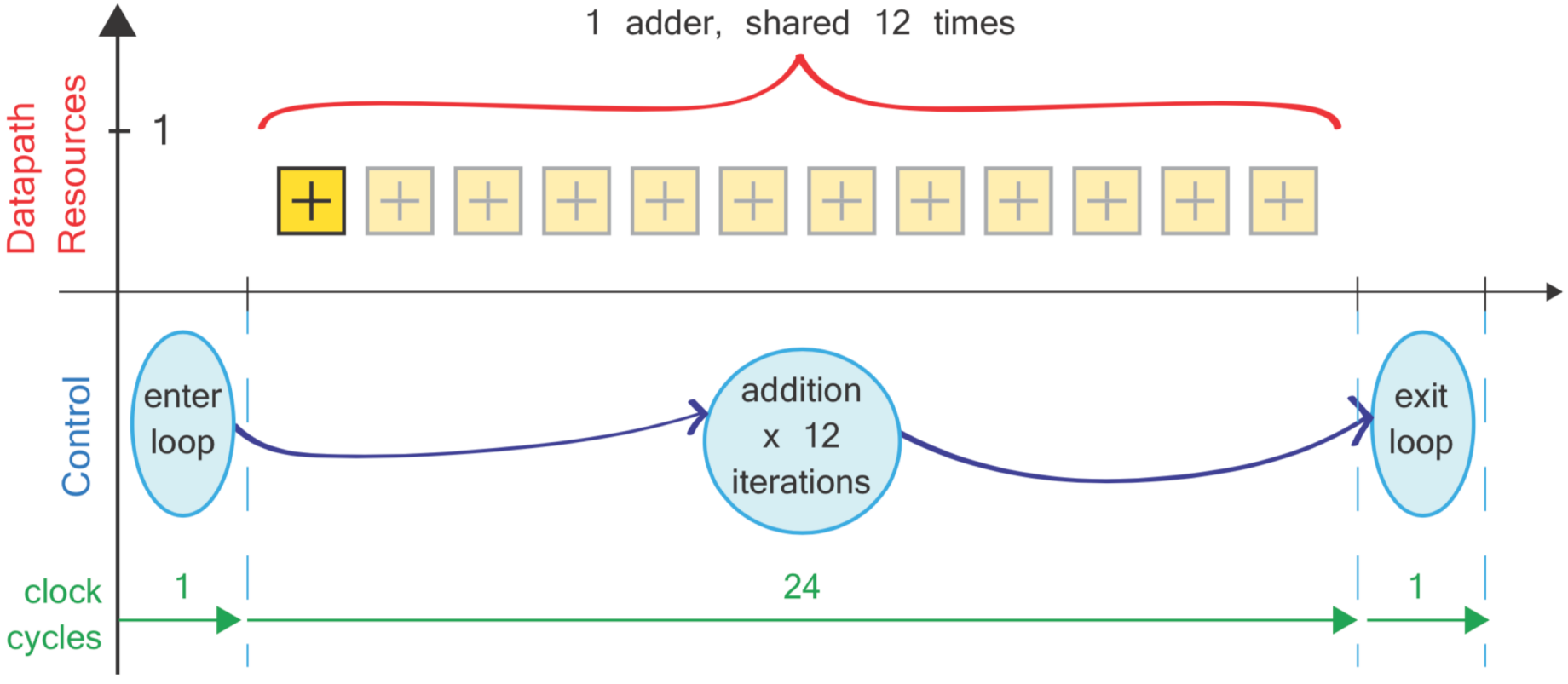


Figure 15.23: Extraction of addition loop into datapath and control logic

Merging Loops

```
void add_mult (short c[12], short m[12], short a[12], short b[12])
{
  short j;


  add_loop: for (j=0;j<12;j++) {
        c[j] = a[j] + b[j];
    }


  mult_loop: for (j=0;j<12;j++) {
        m[j] = a[j] * b[j];
    }
}
```

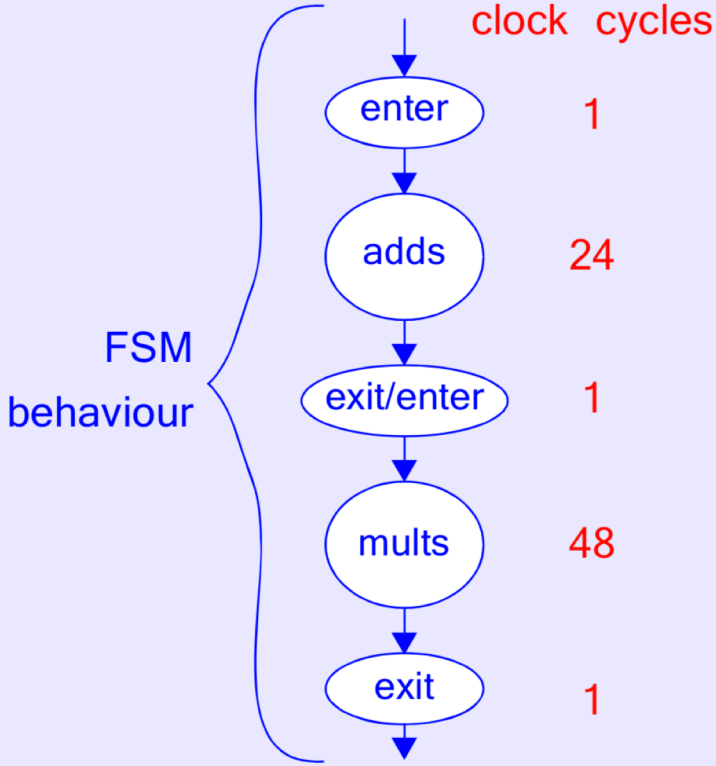


Figure 15.24: Consecutive loops for addition and multiplication within a function

Merging Loops

```
void add_mult (short c[12], short m[12], short a[12], short b[12])
{
  short j;


  add_mult_loop: for (j=0;j<12;j++) {
        c[j] = a[j] + b[j];
        m[j] = a[j] * b[j];
    }
}
```

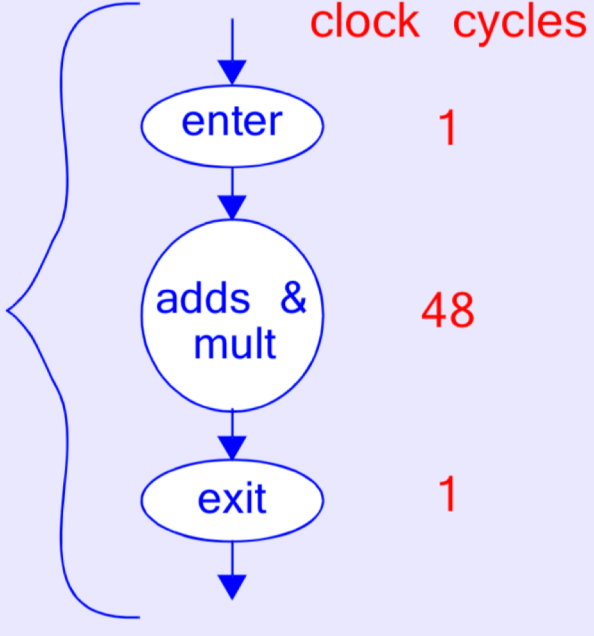


Figure 15.25: Merged addition and multiplication loops

Interface Synthesis

```
void find_average_of_best_X (int *average, int samples[8], int X)

{
     // body of function (statements, sub-function calls, etc.)

}
```

Port Directions

Table 15.6: Synthesis of port directions

C/C++ Function Argument	RTL Port Type
An argument which is read from and never written to	in
An argument which is written to and never read from	out
A value output by the function return statement	out
An argument which is both written to and read from	inout (bidirectional)

Port Protocols

- ***Simple: ap_none, ap_stable, ap_ack***
- ***Ports with validation: ap_vld, ap_ovld, ap_hs***
- ***Memory Interface: ap_memory, bram***
- ***ap_fifo—***
- ***ap_bus—***
- ***AXI: axis, s_axilite, m_axi.***

Backup