Bridge of Life Education

SOC Design

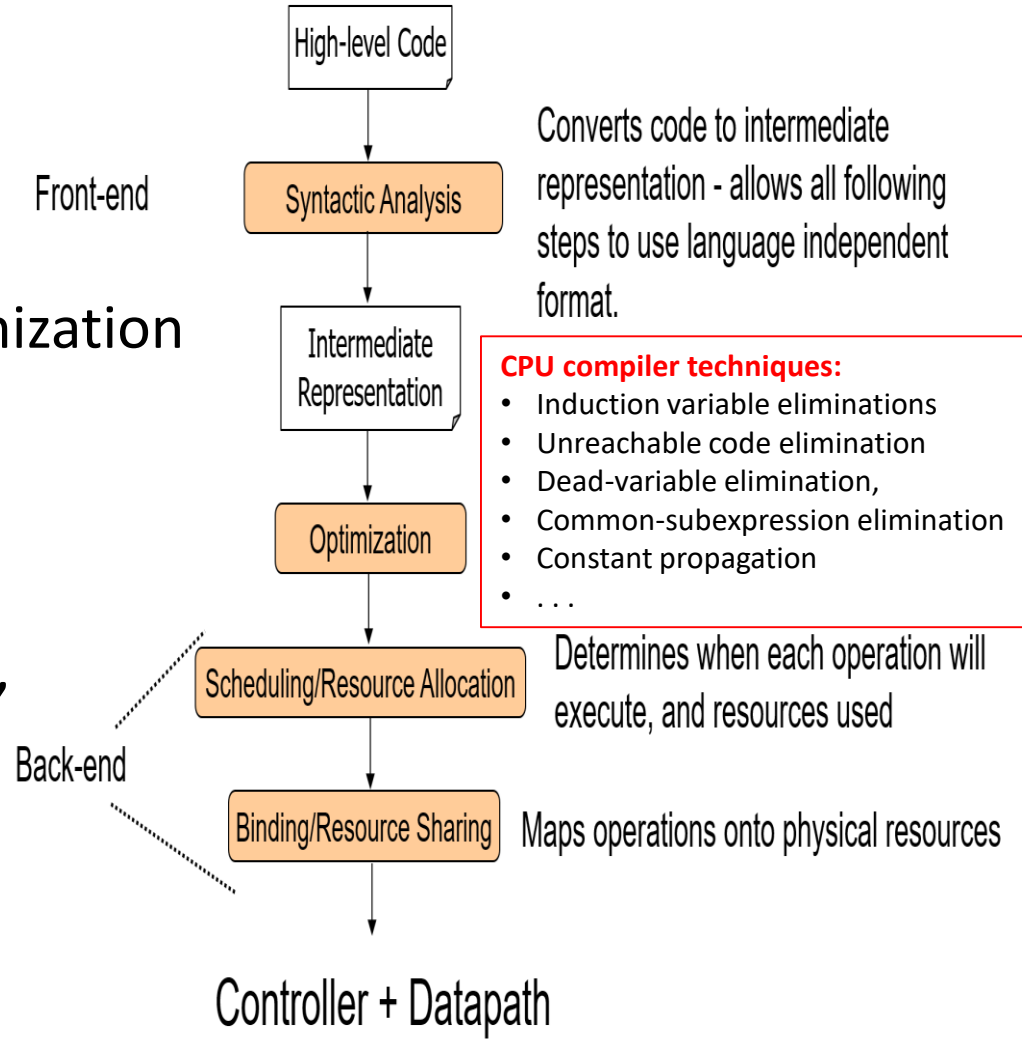
HLS Introduction

Topics

- Unsupported C/C++ Constructs
- C to RTL Mapping

High Level Synthesis - HLS

- Convert (C/C++/OpenCL) into a RTL circuit
  - Optimize for power, performance, area, timing
  - Use Directives (Pragma) to direct compile/optimization process
- Vendors – FPGA vendors, IC
  - Xilinx Vitis-HLS, Intel HLS Compiler
  - Siemens/Mentor Catapult, Cadence Stratus HLS, (Synopsys Synphony HLS)
- Focus on the Back-end part
  - Scheduling/Resource Allocation
  - Binding/Resource Sharing



Code generation from a Domain-Specific Language for C-based HLS of hardware accelerators
https://dl.acm.org/doi/pdf/10.1145/2656075.2656081

Unsupported C/C++ Constructs

Unsupported C/C++ Constructs

- System Calls
- Dynamic Memory Usage (malloc)
- No C++ dynamic polymorphism nor dynamic virtual function call
  - Static/Compile-time polymorphism (function/operator overloading) is ok
- Pointer Limitation
- Recursive Functions

All resource must be statically allocated at compilation stage

Refer to ug902

System Calls

- e.g., printf(), malloc(), getc(), time(), sleep()
- HLS defined macro **__SYNTHESIS__** to exclude non-synthesized code
- **__SYNTHESIS__** is only defined in HLS
- Maintain the same copy of the source code for C-simulation and C/RTL co-simulation

```
void hier_func4(din_t A, din_t B, dout_t *C, dout_t *D)
{
   dint_t apb, amb;

   sumsub_func(&A,&B,&apb,&amb);
#ifndef __SYNTHESIS__
                FILE *fp1;
                char filename[255];
                 sprintf(filename,"Out_apb_%03d.dat",apb);
                fp1=fopen(filename,"w");
                fprintf(fp1, "%d \n", apb);
                fclose(fp1);
#endif
   shift_func(&apb,&amb,C,D);
}
```

Dynamic Memory Usage

- Memory allocation: malloc(), alloc(), and free()
- User-defined macro NO_SYNTH

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

dout_t malloc_removed(din_t din[N], dsel_t width) {
#ifdef NO_SYNTH
   long long *out_accum = malloc (sizeof(long long));
   int* array_local = malloc (64 * sizeof(int));
#else
   long long _out_accum;
   long long *out_accum = &_out_accum;
   int _array_local[64];
   int* array_local = &_array_local[0];
#endif
……
}
```

C to RTL Mapping

Mapping of Key Attributes of C Code

Function: design hierarchy, mapped to MODULE

Arguments : mapped to Input/output interface of the hardware

Types: All variables are of a defined type, influence the area and the performance

Loops: impact on area and performance, HLS opt with Directive Pragma

Control flow: Control logic

Arrays: impact the device area, and performance bottleneck

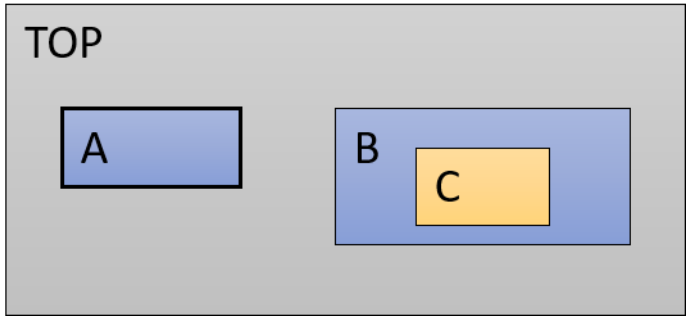
Expression/Operators: Function unit. Allocation/Scheduling (Sharing) to meet performance and area

```
46  #include "fir.h"
47
48  void fir (
49      data_t *y,
50      coef_t c[N],
51      data_t x
52  ) {
53
54      static data_t shift_reg[N];
55      acc_t acc;
56      data_t data;
57      int i;
58
59      acc=0;
60      Shift_Accum_Loop: for (i=N-1;i>=0;i--) {
61          if (i==0) {
62              shift_reg[0]=x;
63              data = x;
64          } else {
65              shift_reg[i]=shift_reg[i-1];
66              data = shift_reg[i];
67          }
68          acc+=data*c[i];;
69      }
70      *y=acc;
71  }
```

Function Hierarchy

- Top-level function becomes the top level of the RTL
- Sub-functions are synthesized into blocks in the RTL design
- Inlined to dissolve the hierarchy
  - Provide greater optimization opportunity
- Vitis requires C++ kernels to be declared as extern “C” to avoid name mangling issues

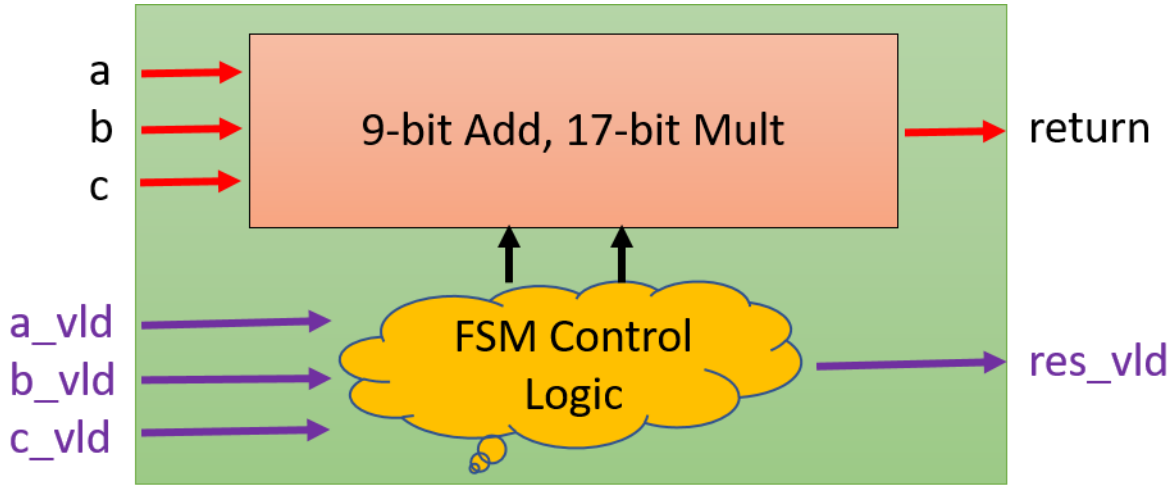
```
void A { ... Body A ...}
void C { ... Body C ...}
void B {   C; }
void TOP() {
                    A (  ...  )
                    B   ( ... )
}
```



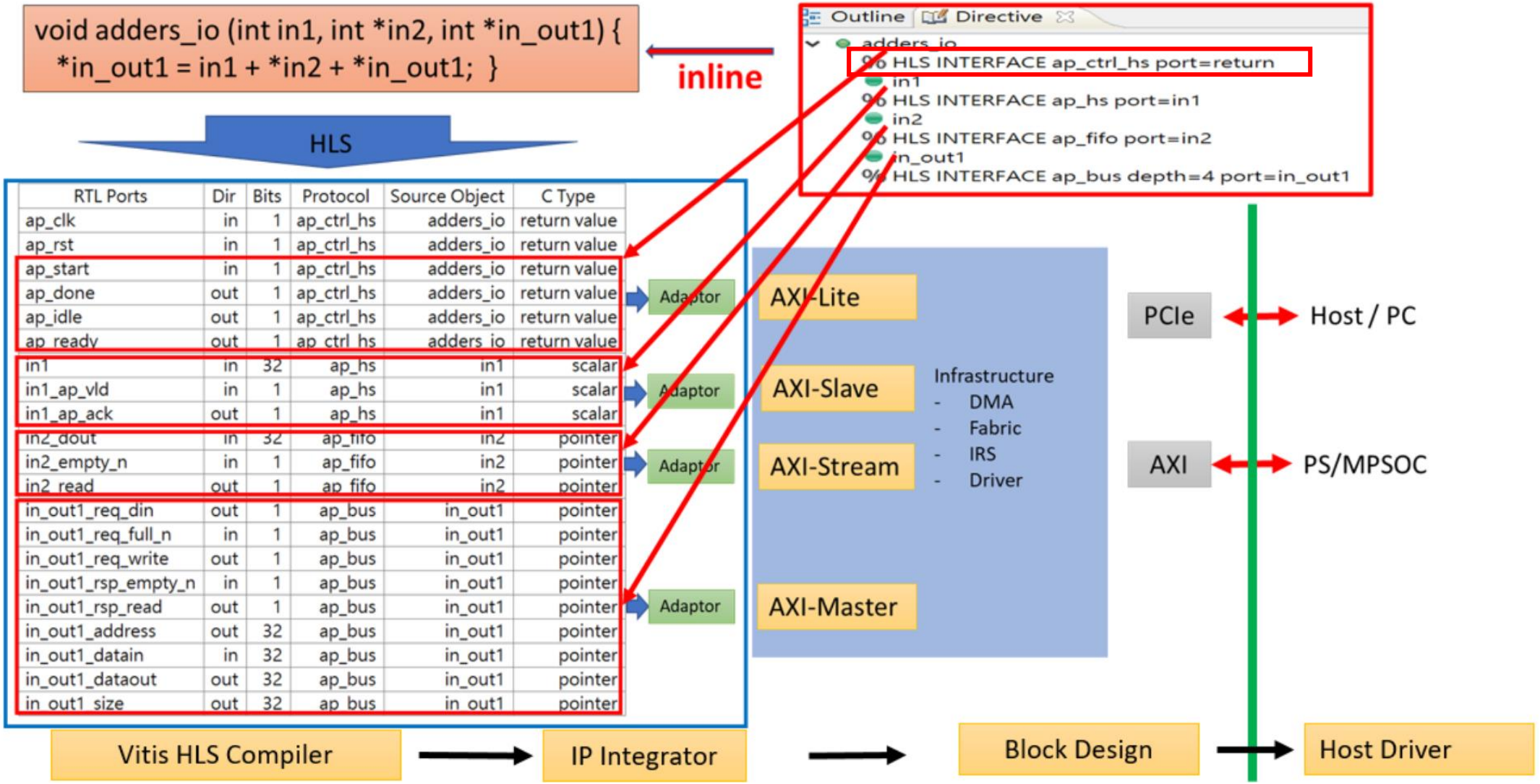
Function Arguments

- Function arguments mapped to ports on the RTL blocks
- **Global variable** if accessed only local to the function, no io port created.
- Insert control ports (Port-level Protocol) to automatically synchronize data exchange among blocks
- Insert Block-level Protocol on Top level function to communicate with Host
- Arbitrary precision bit-width to reduce resource and latency

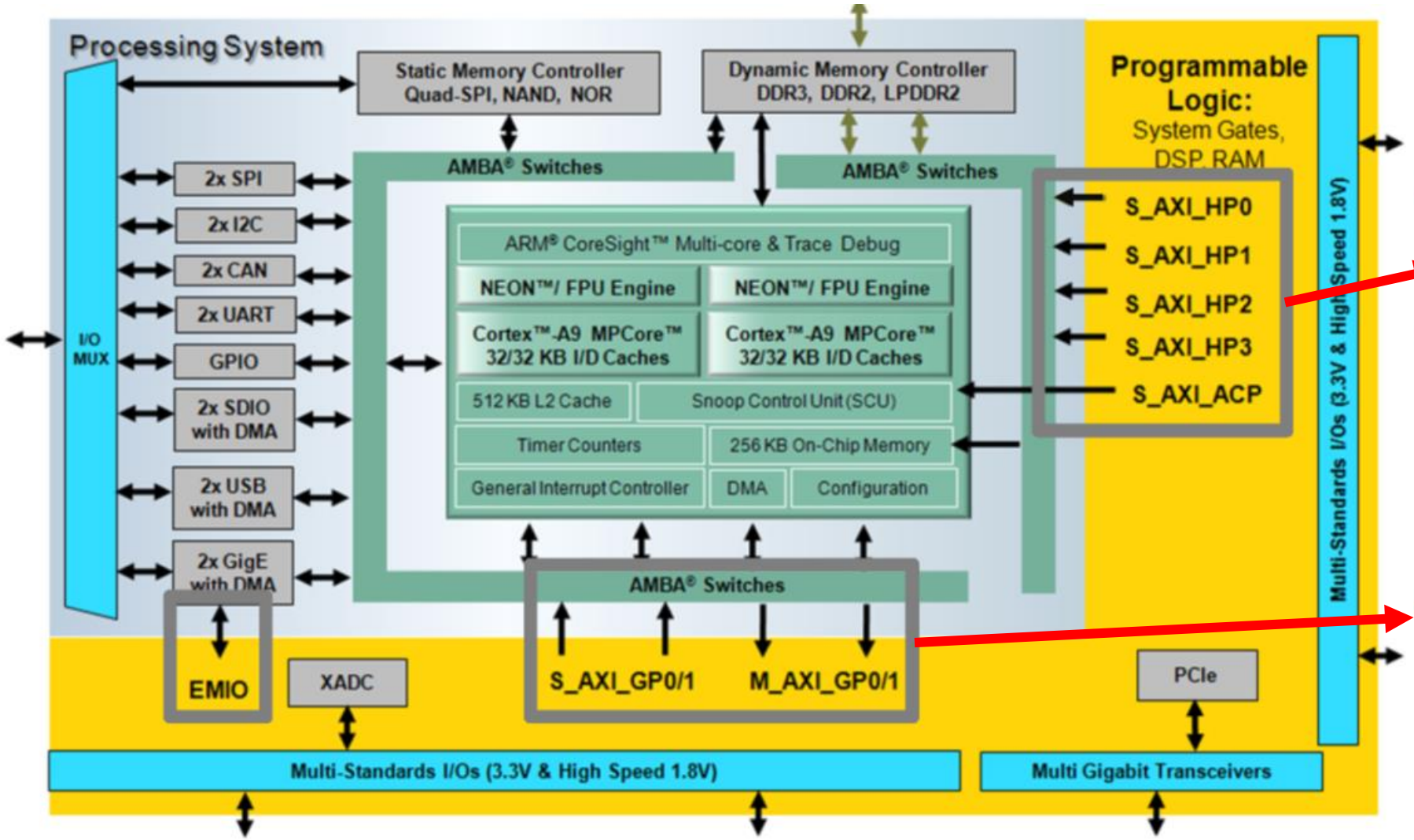
```
int17  foo_top(int8* a, int8* b, int8* c, int17* ret)
{
        int sum, multi;

        sum = *a + *b;
        multi = sum * *c;
        return multi;
}
```



Host/Kernel Communication



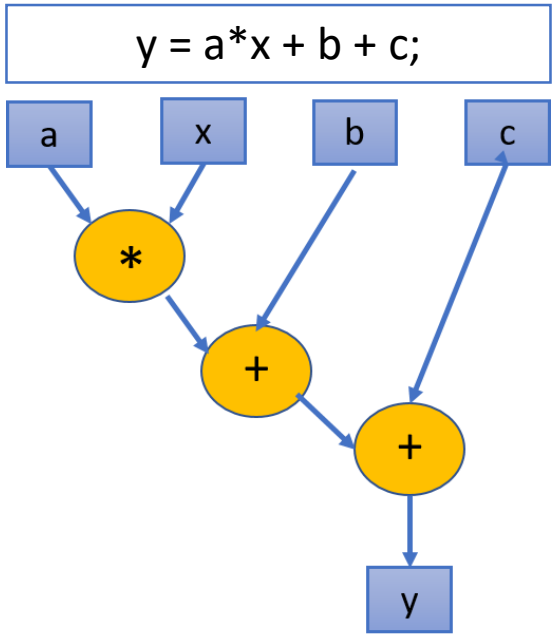
Zynq Block Diagram



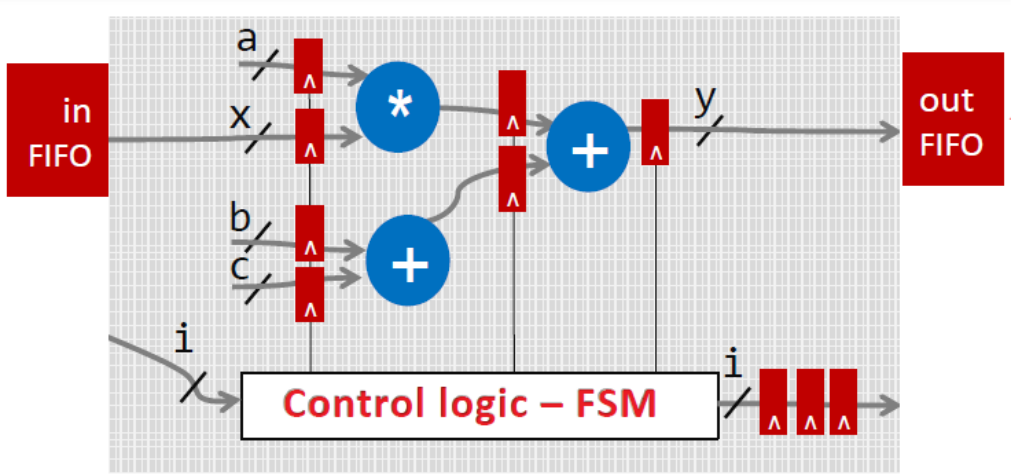
- **AXI High Performance**
  - 4x Slave (64-bit)
    - 1K FIFOs
- **ACP**
  - 1x Slave (64-bit)
  - Cache access

- **AXI General Purpose ports**
  - 2x Master (32-bit)
  - 2x Slave (32-bit)

Expressions – Data Flow Graph



- Start by analyzing the data dependencies between the various steps in the expression shown above. This analysis leads to a Data Flow Graph (DFG)
- Expression is translated to datapath and its control path (FSM)



Resource Allocation, Scheduling, Binding

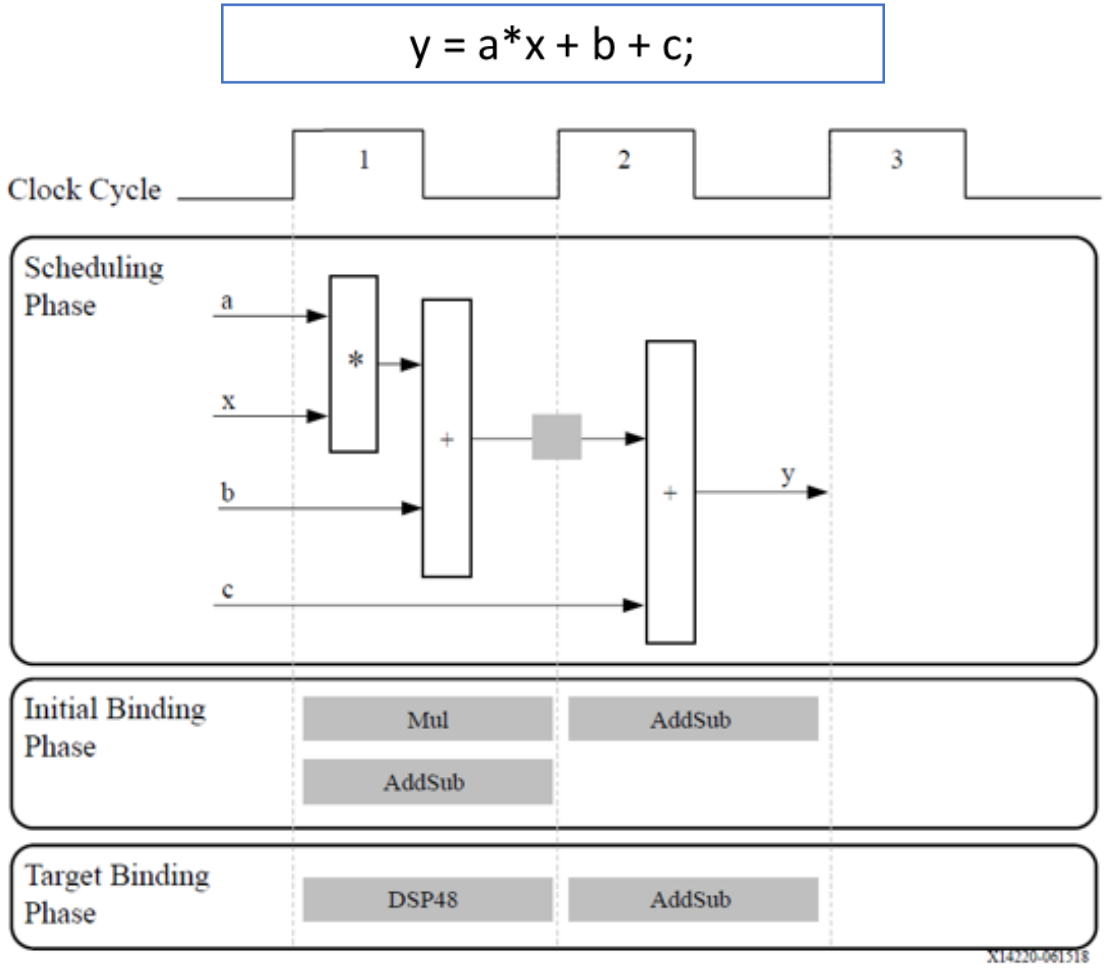
- **Resource allocation**: Each operation is mapped to a hardware resource, annotated with both timing and area information

  #pragma HLS allocation operation instance = add limit = 1

- **Scheduling**: decide which clock cycle to perform what operations

- **Binding**: mapped to the hardware resource.

  #pragma HLS bind_op variable=<variable> op=<type> impl=<value> latency=<int>



Arrays

- Typically implemented by a memory block
  - Read & write array mapped to RAM
  - Constant array mapped to ROM
- Memory access is often the performance bottleneck
  - HLS default memory model assumes 2-port BRAM
  - Array can be reshaped and/or partitioned to remove bottleneck


```
void foo (...) {
...
SUM_LOOP:for(i=2;i<N;++i) {
  sum += mem[i] + mem[i-1] + mem[i-2];
  ...
  }
}
```

See UG902 to get full throughput on this example
- *(Chap 3 – Array Accesses and Performance)*

Example: Code implies three reads from a RAM, prevents full throughput

Control Flow: Loop

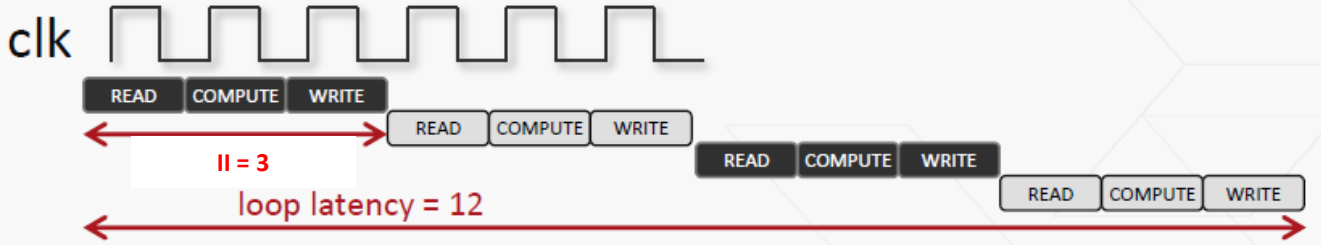
- Loops are the main area of parallelism in an algorithm
- Loops can be
  - pipelined,
  - Unrolled, Partially unrolled,
  - Merged
  - Flattened
- HLS generates the datapath and control logic

Loop - Pipeline

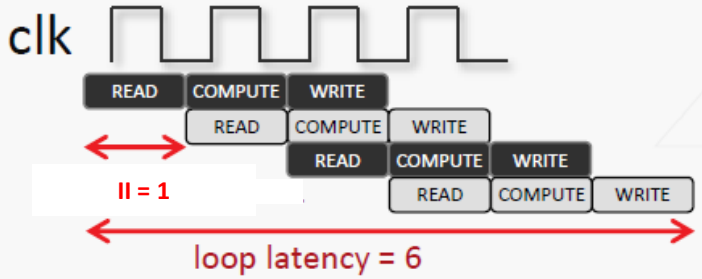
- One of the most important optimization
- Allow a new iteration to begin before the previous iteration is complete
- Key matric: Initiation Interval (II)

```
void F (...)  {
...
add: for (i=0;i<=3;i++) {
# PRAGMA HLS PIPELINE
      op_READ;
      op_COMPUTE;
      op_WRITE;
}
...
```

default



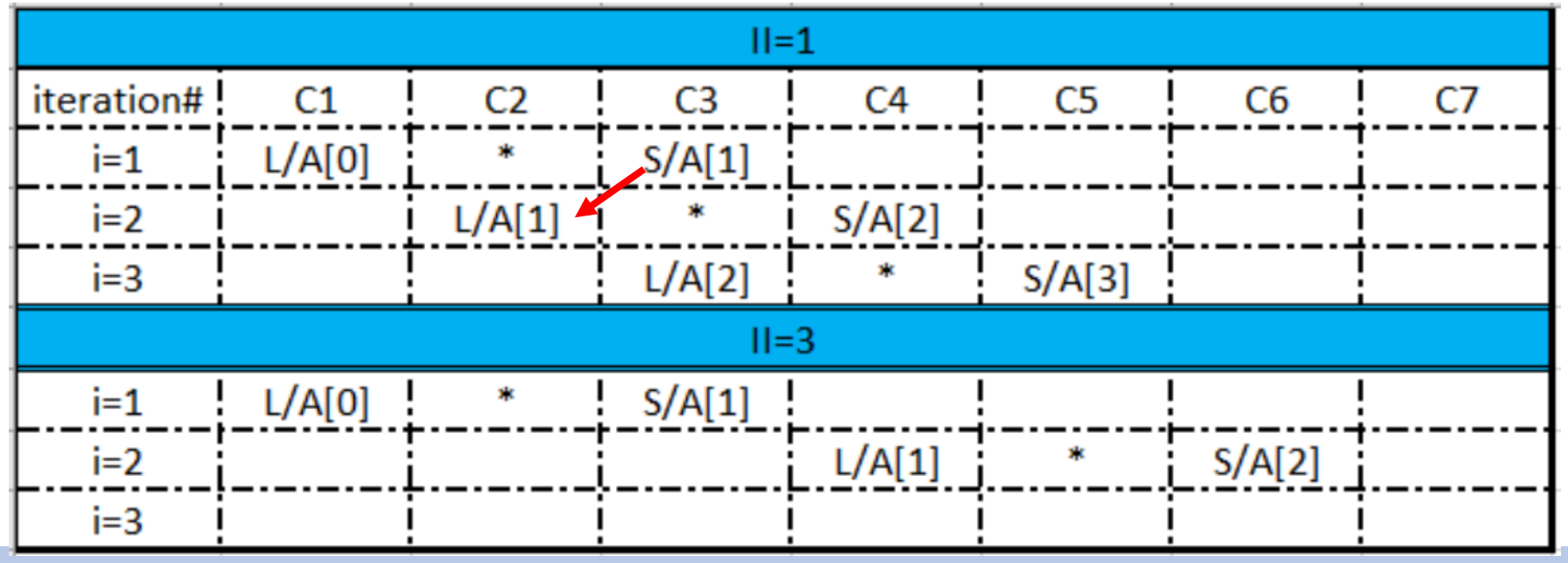
PIPELINE



Data Dependency – RAW (Read After Write)

- **True dependency**
- S1-iteration(u) -> S2-iteration (v)
- S1 computes a value S2 uses

```
for(i=0; I < N; i++)
{
   A[i] = A[i-1] * a;
 }
```

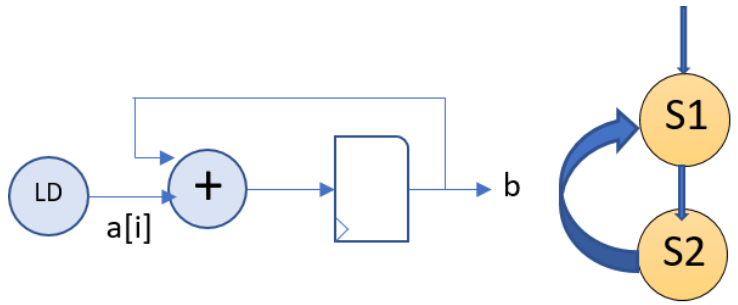


II=1							
iteration#	C1	C2	C3	C4	C5	C6	C7
i=1	L/A[0]	*	S/A[1]				
i=2		L/A[1]	*	S/A[2]			
i=3			L/A[2]	*	S/A[3]		
II=3							
i=1	L/A[0]	*	S/A[1]				
i=2				L/A[1]	*	S/A[2]	
i=3							

Control Flow – Rolled

- By default, loops are rolled
  - Each loop iteration corresponds to a “sequence” of states (DAG)
  - The state sequence will be repeated multiple times based on the loop trip count.
  - The resource (adder) is repeatedly used in the loop iteration.
  - Efficient use the resource, but longer latency

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



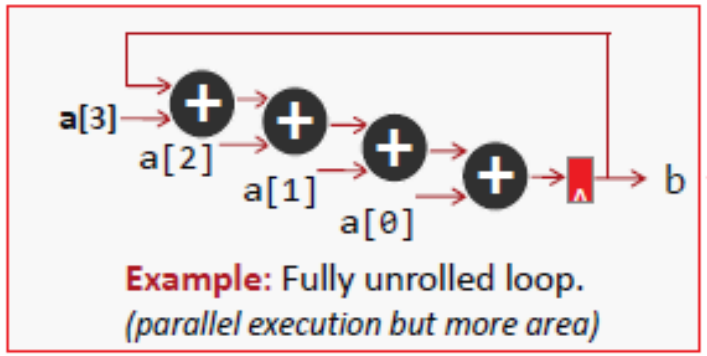
Loop - Unroll

- Rolled loops can be made unrolled or partially unrolled by

  #pragma UNROLL [factor = n]

- Pros
  - Decrease loop overhead
  - Increase parallelism for scheduling
- Cons
  - Increase operator count, negatively impact area, power and timing

```
void F ( . . . ) {
. . .
add: for (i=0; i <= 3; i++) {
   #pragma UNROLL
   b += a[i];
. . .
```

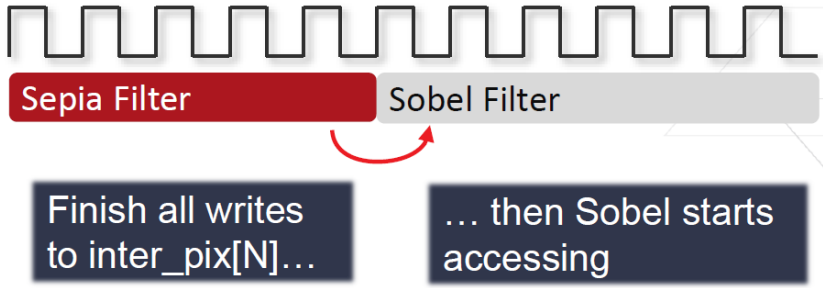


Task-Level Parallelism - Dataflow

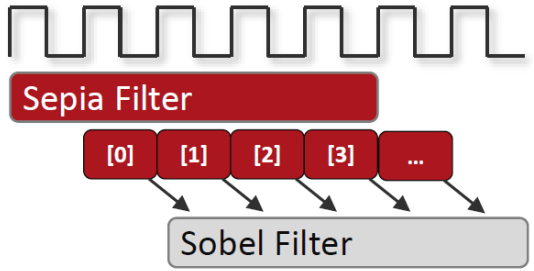
- **By default a C function producing data for another is fully executed first**

```
// This memory can be a FIFO during optimization
rgb_pixel inter_pix[MAX_HEIGHT][MAX_WIDTH];

// Primary processing functions
sepia_filter(in_pix,inter_pix);
sobel_filter(inter_pix,out_pix2);
```

Sepia Filter
Sobel Filter



- **Dataflow allows Sobel to start as soon as data is ready**
  - Functions operate concurrently and continuously
  - The interval (hence throughput) is improved
  - Channel buffer has to be filled before consumed for ping-pong



- **Dataflow creates memory channels**
  - Created between loops or functions to store data samples
  - “Ping-pong” channel holds all the data
  - “FIFO” for sequential access, no need to store all the data

Supplement

Pointer Limitation

- **General Pointer Casting**
  - Not support general pointer casting
  - Support pointer casting between native C/C++ types.
- **Pointer Arrays**
  - Support pointer array to a scalar or an array of scalars
  - Not support array of pointers point to additional pointers
- **Function Pointer – not supported**

Recursive Function – A GCD Example

- Recursive functions cannot be synthesized because their function call depth is data-dependent, thus non-determined at compiler time.

- **Tail-recursion is a loop in disguise**, the simple function can easily be transformed as right.

```
unsigned foo (unsigned m, unsigned n) {
   if (m == 0) return n;
   if (n == 0) return m;
   return foo(n, m%n);
}
```

```
unsigned foo (unsigned m, unsigned n) {
   while( m!=0 & n!=0 ) {
      unsigned int mmodn=m%n;
      m=n;
      n=mmodn;
   }
   if (m == 0) return n;
   else return m;
}
```

https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/Recursive-Functions

Using C++ Templates for Recursion

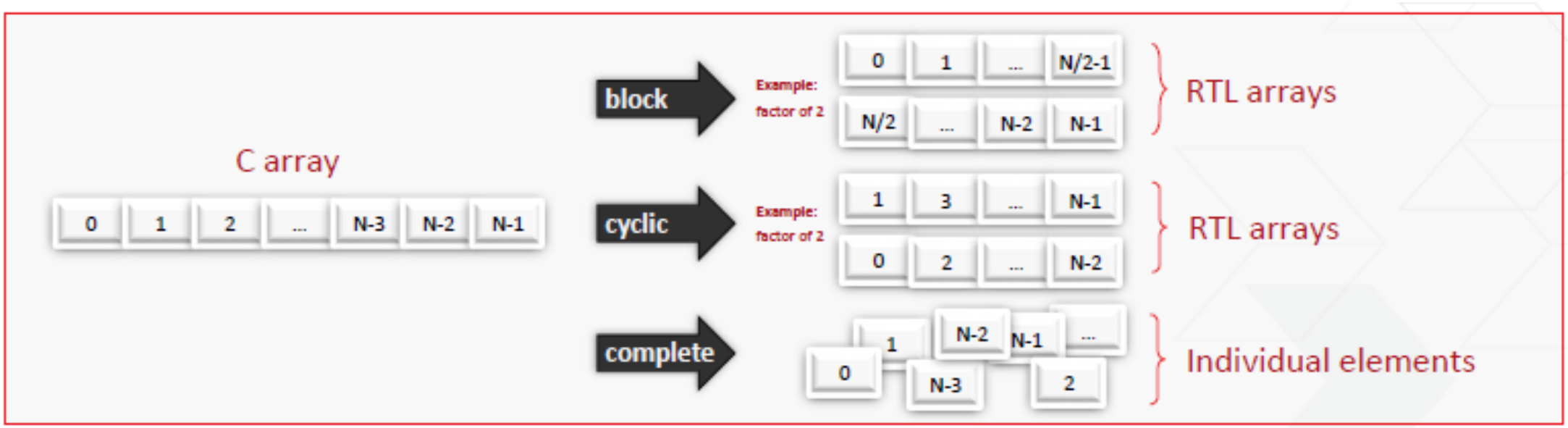
- Fibonacci number
  - Fn= Fn-1 + Fn-2 n > 2
  - F1 = F2 = 1
- The key to performing synthesis is a termination class (template<> struct **fibon_s<1>**) is used to implement the final call in the recursion where a template size of one is used.

```
// Tail recursive call
template<data_t N>
   struct fibon_s {
      template<typename T>
      static T fibon_f(T a, T b) {
            return fibon_s<N-1>::fibon_f(b, (a+b));
     }
};
// Termination condition
template<> struct fibon_s<1> {
    template<typename T>
       static T fibon_f(T a, T b) {
       return b;
    }
};
void cpp_template(data_t a, data_t b, data_t &dout){
  dout = fibon_s<FIB_N>::fibon_f(a,b);
}
```

Function, Top function (Kernel) explained

Partition, Reshape Your Arrays

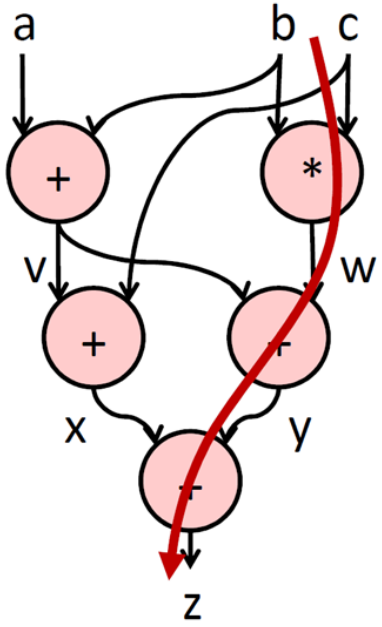
- Partitioning splits an array into independent arrays
  - Array can be partitioned on any of their dimensions for better throughput
- Reshaping combines array elements into wider containers
  - Different arrays into a single physical memory
  - New RTL memories are automatically generated without changes to C code

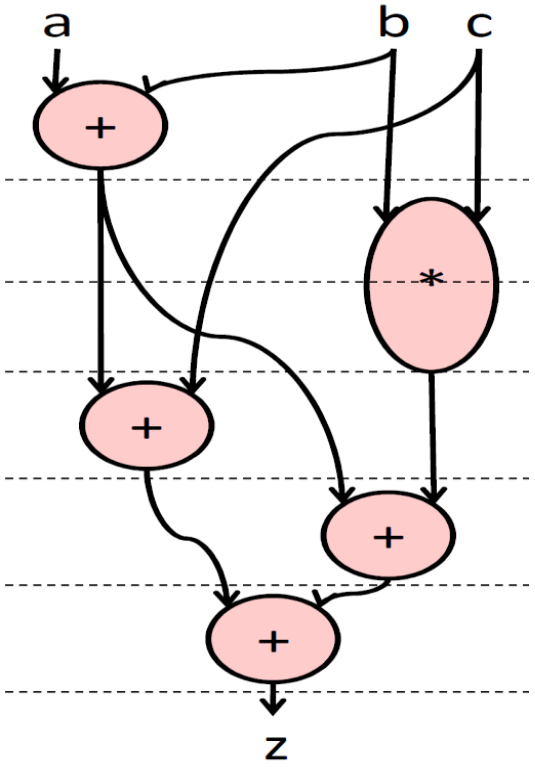


Examples Demonstrate HLS Process

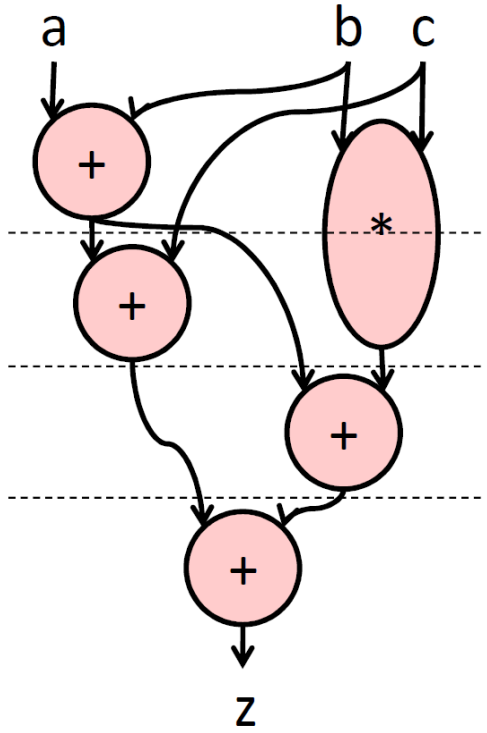
Example - Expression Datapath Resource allocation & Scheduling

```
v = a + b;
w = b * c;
x = v + c;
y = v + w;
z = x + y;
```



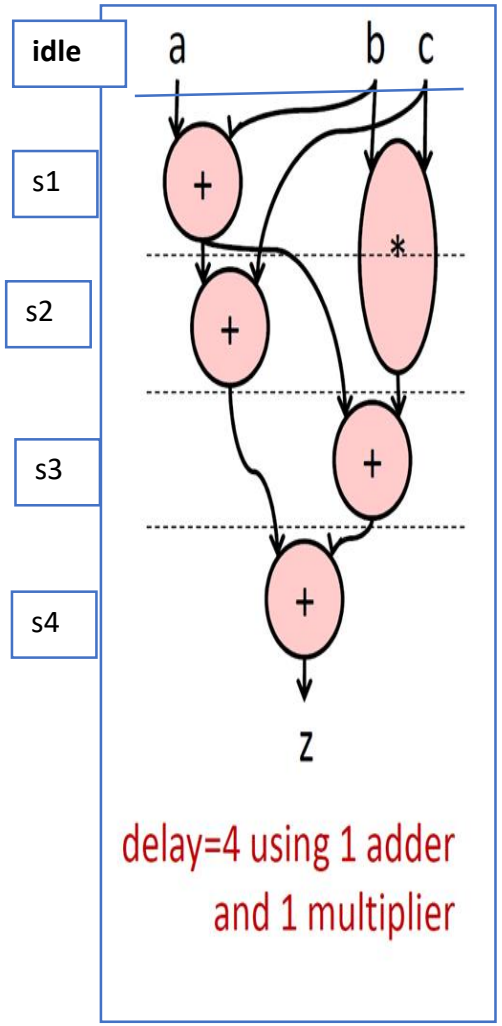


delay=6 using 1 MAC

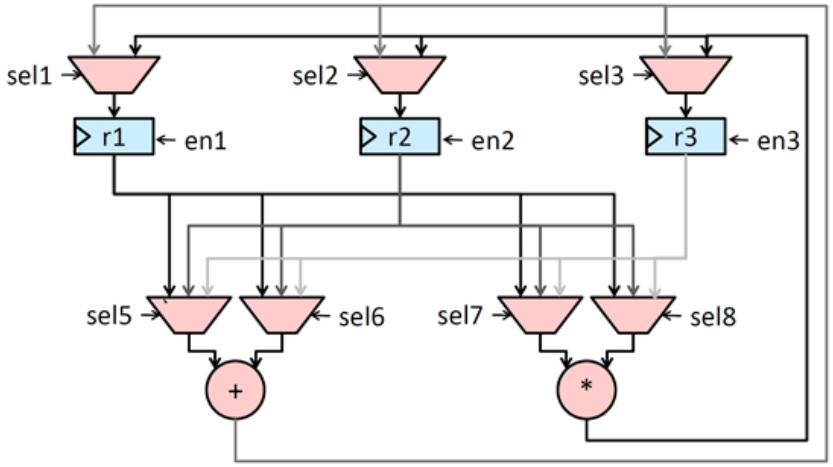


delay=4 using 1 adder and 1 multiplier

Example – Expression Datapath Binding & Resource Sharing



Generating Datapath



Assign State for Control Signals

- Assume initially a in r1; b in r2; c in r3

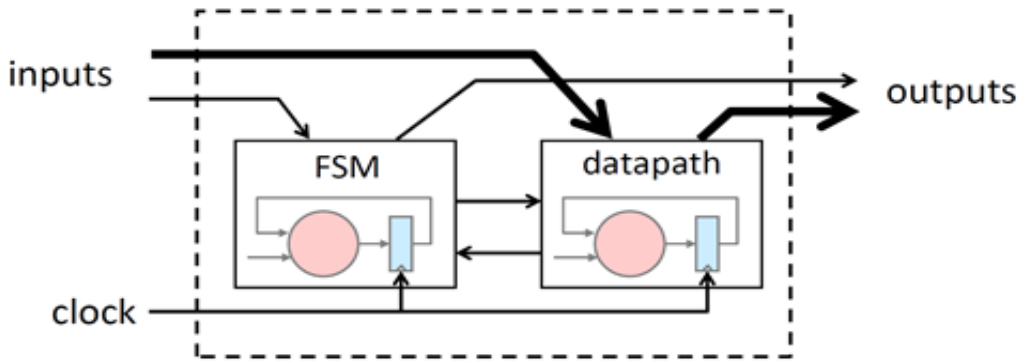
	r1		r2		r3		add		mult	
	sel1	en1	sel2	en2	sel3	en3	sel5	sel6	sel7	sel8
S1	add	1	-	0	-	0	r1	r2	r2	r3
S2	-	0	add	1	mul	1	r1	r3	r2	r3
S3	add	1	-	0	-	-	r1	r3	-	-
S4	add	1	-	-	-	-	r2	r1	-	-

Generating State Machines

- Multiple cycles
- States: (idle, s1, s2, s3, s4
- Encode States: 000,100,101,110,111)

Generate Control Signals

- Combines the Control Signal State & State Machine, e.g.
- e.g. en1 = s1 | s3 | s4;

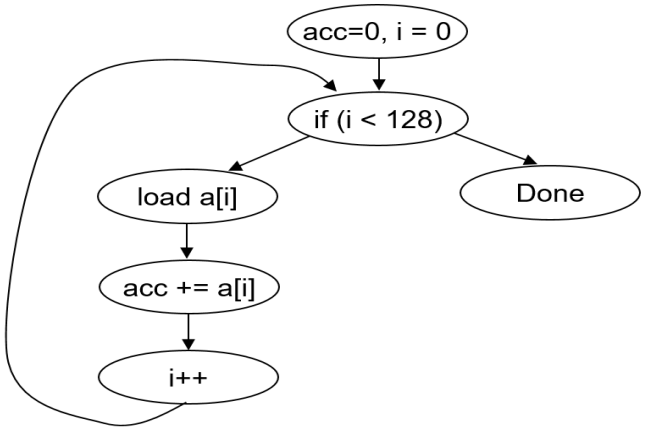


Example – Control Flow

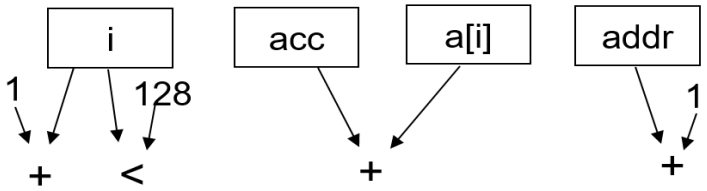
0: Loop – add array

```
int controlflow(int a[N]) {
    int i, acc;
    acc = 0;
    for(i = 0; i < N; i++) {
        acc += a[i];
    }
    return acc;
}
```

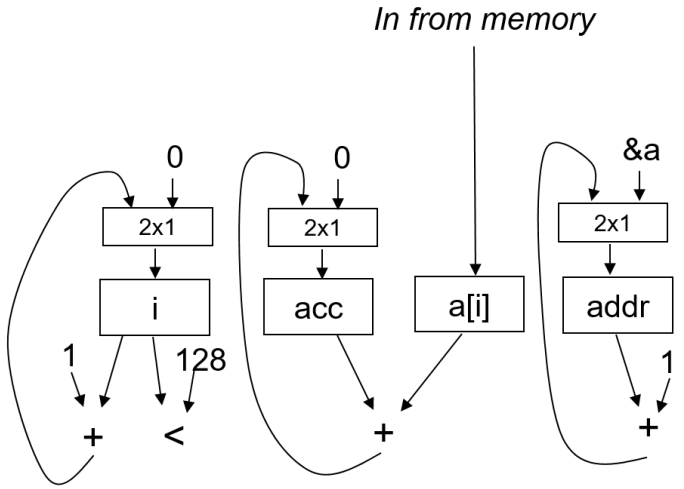
1: Decompose into states



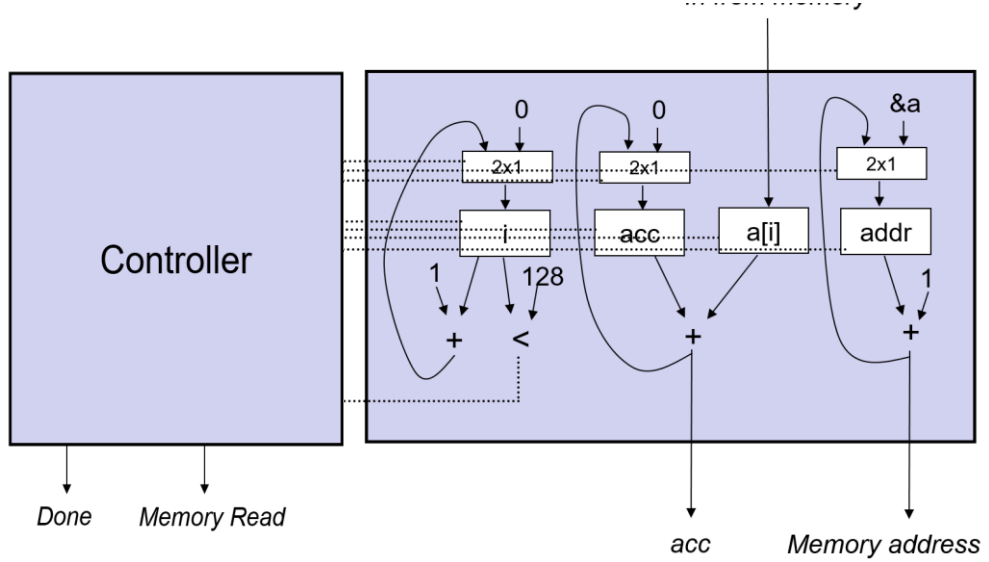
2: Identify state variables & operands



3: Determine sources of state variable input



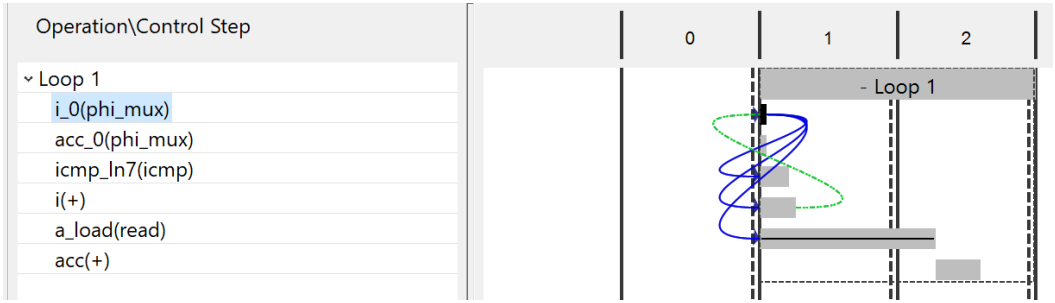
4: add input/output
5: Generate Controller (State Machines) for datapath control

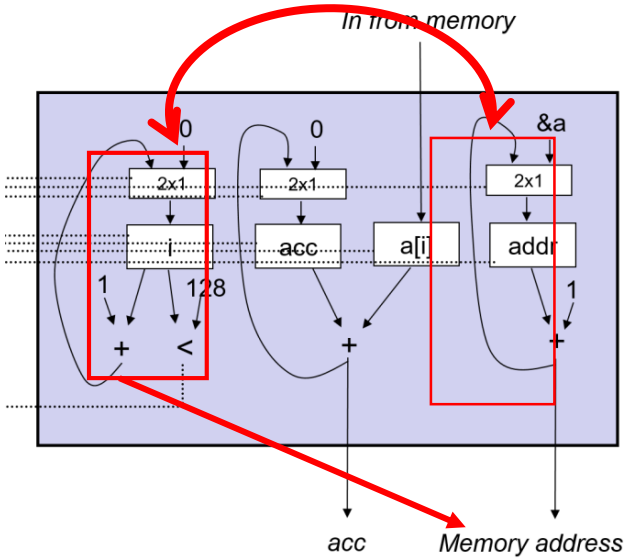


Example – Control Flow (Vivado HLS)

```
int controlflow(int a[N]) {
    int i, acc;
    acc = 0;
    for(i = 0; i < N; i++) {
        acc += a[i];
    }
    return acc;
}
```

Scheduling





Interface

RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	controlflow	return value
ap_rst	in	1	ap_ctrl_hs	controlflow	return value
ap_start	in	1	ap_ctrl_hs	controlflow	return value
ap_done	out	1	ap_ctrl_hs	controlflow	return value
ap_idle	out	1	ap_ctrl_hs	controlflow	return value
ap_ready	out	1	ap_ctrl_hs	controlflow	return value
ap_return	out	32	ap_ctrl_hs	controlflow	return value
a_address0	out	4	ap_memory	a	array
a_ce0	out	1	ap_memory	a	array
a_q0	in	32	ap_memory	a	array

Resource

Expression

Variable Name	Operation	DSP48E	FF	LUT	Bitwidth P0	Bitwidth P1
acc_fu_75_p2	+	0	0	39	32	32
i_fu_64_p2	+	0	0	13	4	1
icmp_ln7_fu_58_p2	icmp	0	0	9	4	4
Total	3	0	0	61	40	37

Multiplexer

Name	LUT	Input Size	Bits	Total Bits
acc_0_reg_46	9	2	32	64
ap_NS_fsm	21	4	1	4
i_0_reg_35	9	2	4	8
Total	39	8	37	76

- Memory address is the same as “variable i”

Question#1:

- When we run the synthesis on the right kernel function. Is the synthesis able to run? If not, what is wrong?

```
#include "malloc_removed.h"
#include <stdlib.h>
#define NO_SYNTH

dout_t malloc_removed(din_t din[N], dsel_t width) {
#ifdef NO_SYNTH
   long long *out_accum = malloc (sizeof(long long));
   int* array_local = malloc (64 * sizeof(int));
#else
   long long _out_accum;
   long long *out_accum = &_out_accum;
   int _array_local[64];
   int* array_local = &_array_local[0];
#endif
……
}
```

Question#2

- Given the code below, and assuming the following resources
  - a[], and b[] arrays are in the same memory block (a single port memory)
  - c[] array has its own memory block
  - One multiplier available with 2 T latency

What is the II ?

```
void TOP(...) {
    ...
    for(i = 0; i < 4; i++)
        c[i] = a[i] * b[i];
}
```