The Abdus Salam International Centre for Theoretical Physics

IAEA International Atomic Energy Agency

Joint ICTP-IAEA School on Zynq-7000 SoC and its Applications for Nuclear and Related Instrumentation

Introduction to High-level Synthesis

Fernando Rincón

fernando.rincon@uclm.es

Contents

- What is High-level Synthesis?
- Why HLS?
- How Does it Work?
- HLS Coding
- An example: Matrix Multiplication
- Validation Flow
- RTL Export
- Design analysis

What is High-level Synthesis?

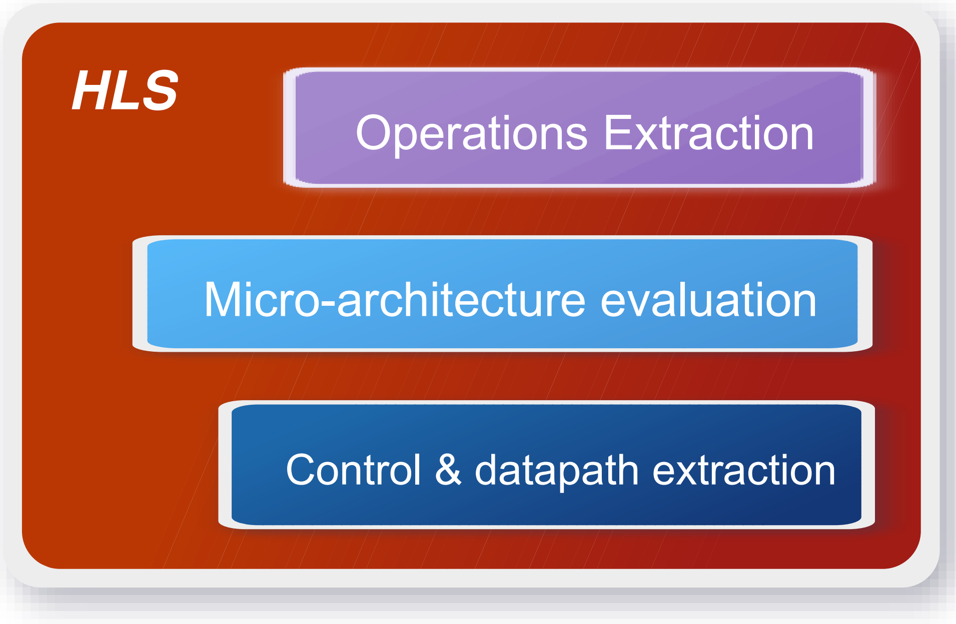
- Compilation of behavioral algorithms into RTL descriptions

Behavioral Description

Algorithm

Constraints

I/O description
Timing
Memory



RTL Description

Finite State Machine

Datapath

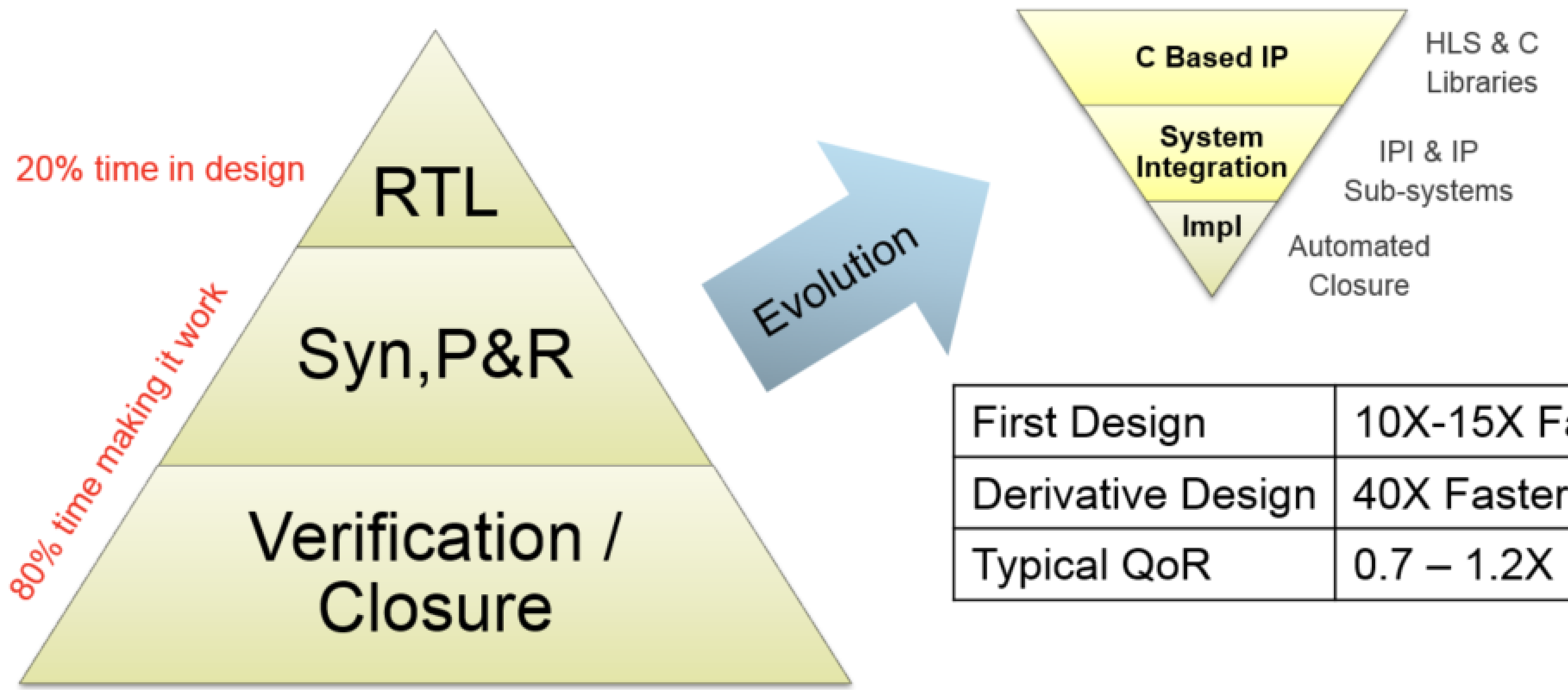
Why HLS?

- Need for productivity improvement at design level
  - Design Space Exploration
  - Reduced Time-to-market
  - Trend to use FPGAs as Hw accelerators
- Electronic System Level Design is based in
  - Hw/Sw Co-design
    - SystemC / SystemVerilog
    - Transaction-Level Modelling
  - One common C-based description of the system
  - Iterative refinement
  - Intregration of models at a very different level of abstraction
  - But need an efficient way to get to the silicon
- Rising the level of abstraction enables Sw programmers to have access to silicon

Why HLS?

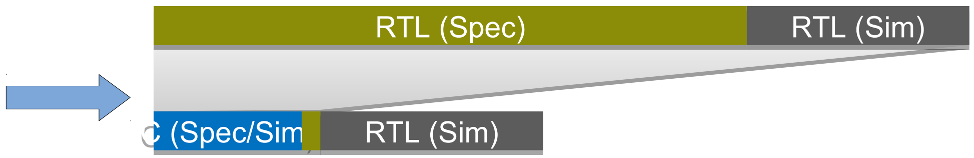
Vivado RTL-Based Design

Vivado C and IP-Based Design



First Design	10X-15X Faster
Derivative Design	40X Faster
Typical QoR	0.7 – 1.2X

Video Design Example			
Input	C Simulation Time	RTL Simulation Time	Improvement
10 frames 1280x720	10s	~2 days (ModelSim)	~12000x



HLS Benefits

- Design Space Exploration
  - Early estimation of main design variables: latency, performance, consumption
  - Can be targeted to different technologies
- Verification
  - Reuse of C-based testbenches
  - Can be complemented with formal verification
- Reuse
  - Higher abstraction provides better reuse opportunities
  - Cores can be exported to different bus technologies

Design Space Exploration

```
…
  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];
     }
```

Ssame hardware is used for each loop iteration :

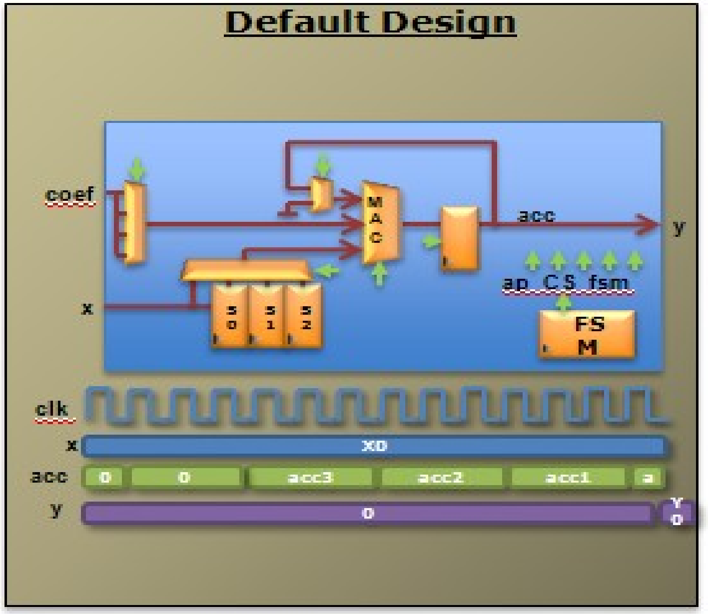
- Small area
- Long latency
- Low throughput

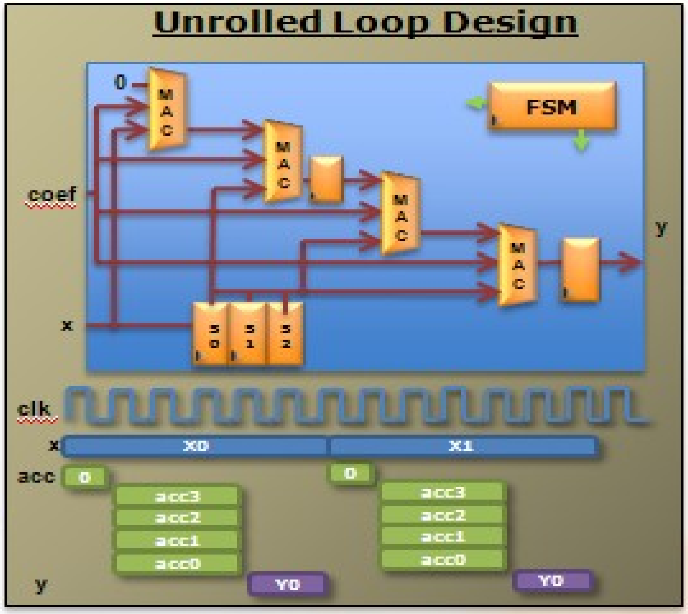
Different hardware for each loop iteration :

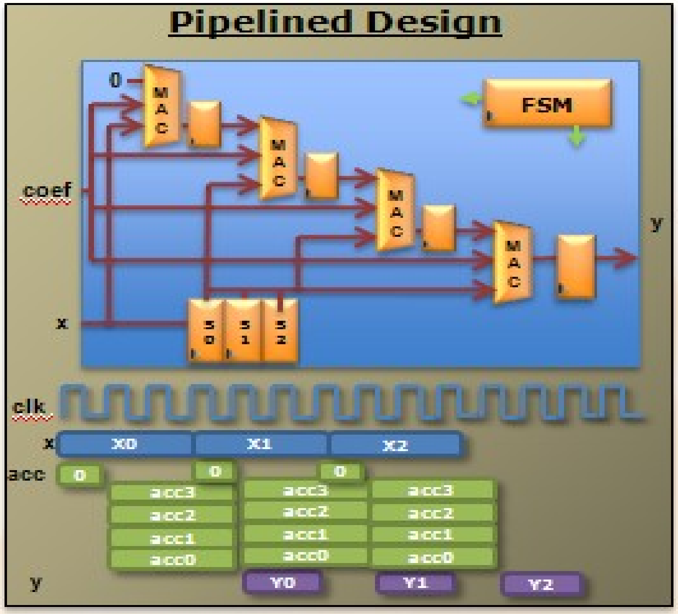
- Higher area
- Short latency
- Better throughput

Different iterations executed concurrently:

- Higher area
- Short latency
- Best throughput







How Does it Work? - Control Extraction

Code

```c
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;
}
```

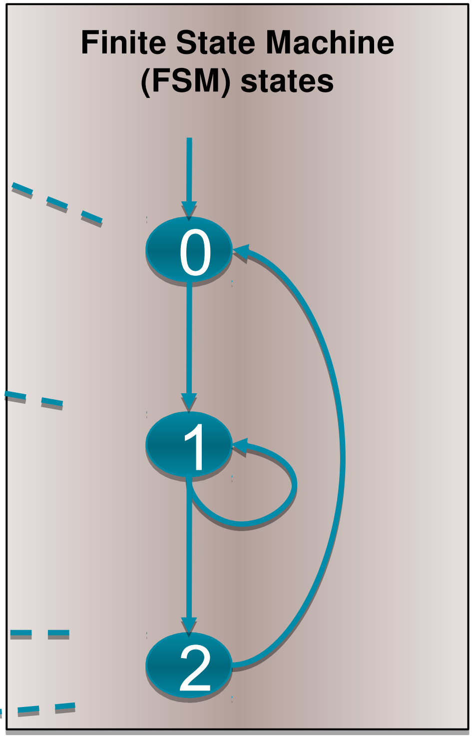
Control Behavior

Function Start

For-Loop Start

For-Loop End

Function End



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

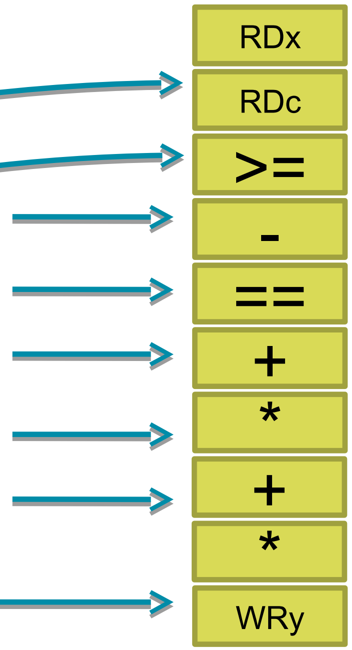
How does it work? - Datapath Extraction

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;
}
```

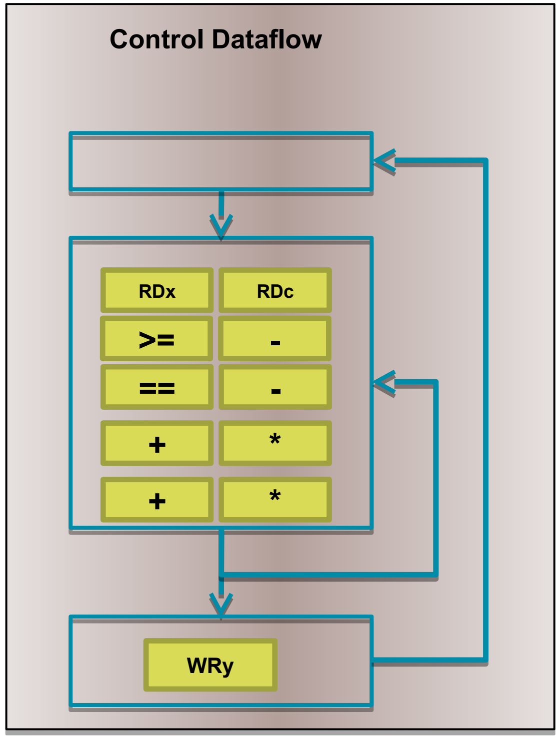
From any C code example ..

Operations



Operations are extracted...

Control & Datapath Behavior

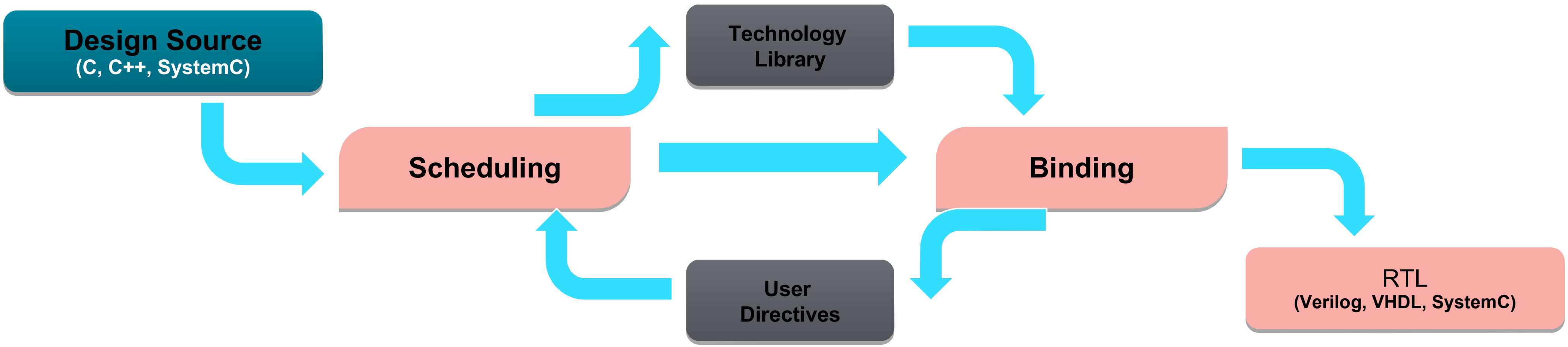


A unified control dataflow behavior is created.

Scheduling + Binding

How Does it Work? - Scheduling & Binding

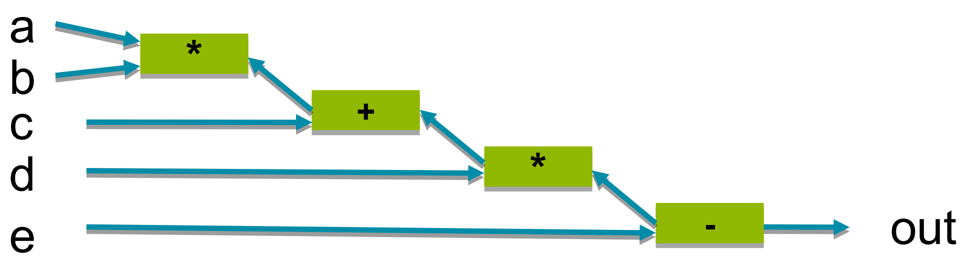
- Scheduling and Binding are at the heart of HLS
- Scheduling determines in which clock cycle an operation will occur
  - Takes into account the control, dataflow and user directives
  - The allocation of resources can be constrained
- Binding determines which library cell is used for each operation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



How Does it Work? - Scheduling

- Operations are mapped into clock cycles, depending on timing, resources, user directives, ...

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



When a faster technology or slower clock ...

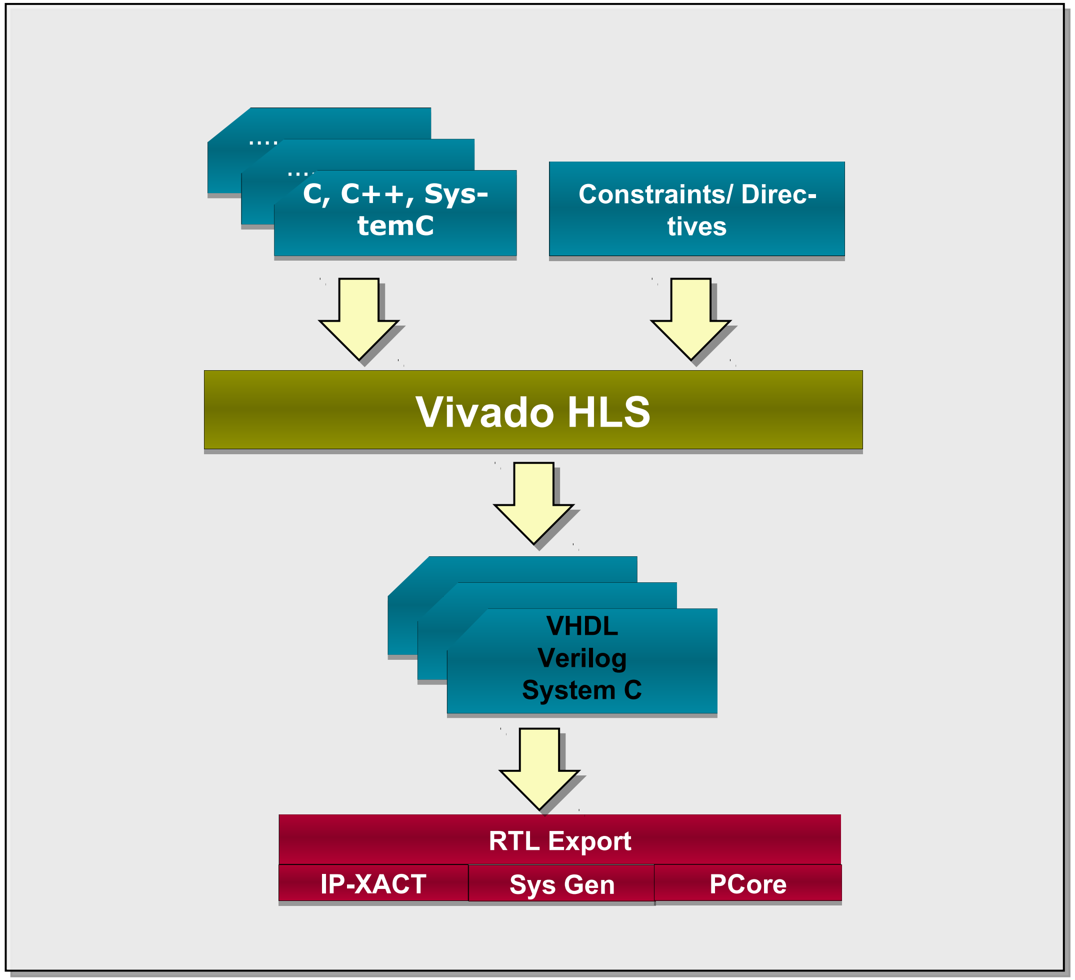
How Does it Work? - Allocation & Binding

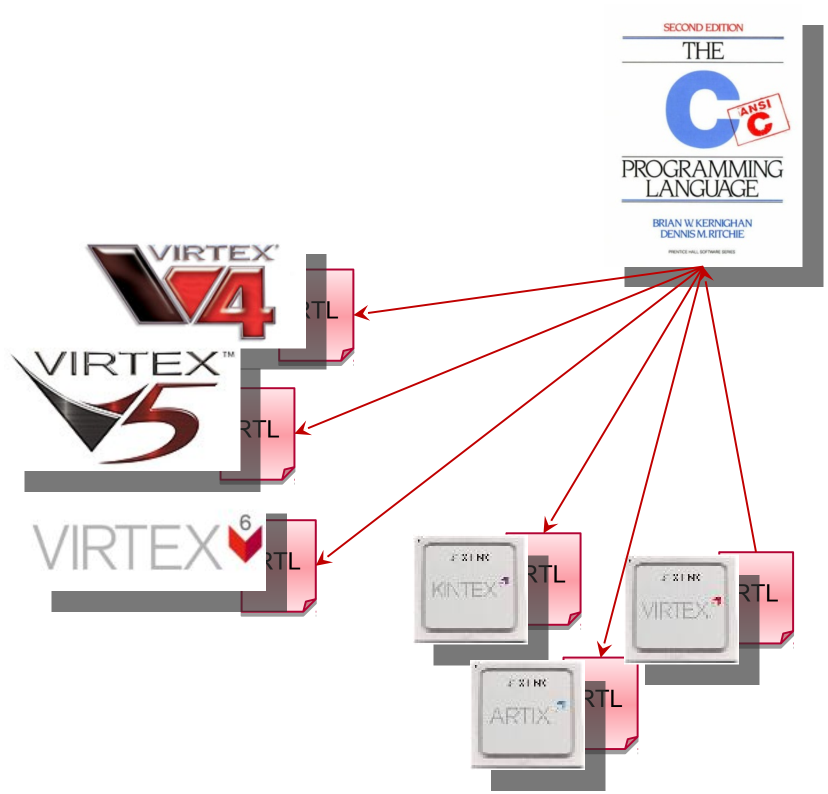
Operations are assigned to functional units available in the library

Vivado HLS

- High-level Synthesis Suite from Xilinx

TODO: transparencia de transición
Visualizar el flujo





Source Code: Language Support

- Vivado HLS supports C, C++, SystemC and OpenCL API C kernel
  - Provided it is statically defined at compile time
  - Default extensions: .c for C / .cpp for C++ & SystemC
- Modeling with bit-accuracy
  - Supports arbitrary precision types for all input languages
  - Allowing the exact bit-widths to be modeled and synthesized
- Floating point support
  - Support for the use of float and double in the code
- Support for OpenCV functions
  - Enable migration of OpenCV designs into Xilinx FPGA
  - Libraries target real-time full HD video processing

Source Code: Key Attributes

- Only one top-level function is allowed

```
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: Represent the design hierarchy

Top Level IO : Top-level arguments determine Interface ports

Types: Type influences area and performance

Loops: Their scheduling has major impact on area and performance

Arrays: Mapped into memory. May become main performance bottlenecks

Operators: Can be shared or replicated to meet performance

Functions & RTL Hierarchy

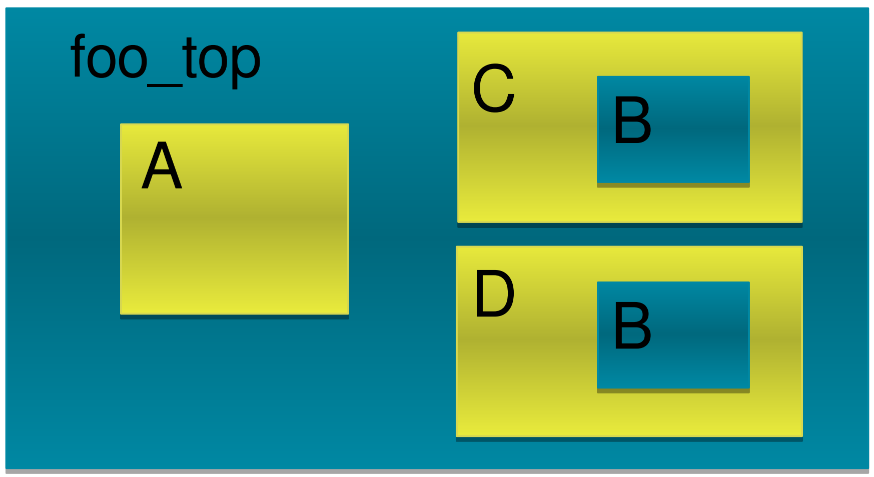
- Each function is translated into an RTL block.
- Can be shared or inlined (dissolved)

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



Operator Types

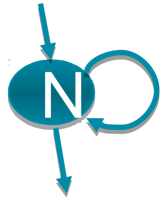
- They define the size of the hardware used

- **Standard C Types**
  - Integers:
    - `long long` => 64 bits
    - `int` => 32 bits
    - `short` => 16 bits
  - Characters:
    - char => 8 bits
  - Floating Point
    - Float => 32 bits
    - Double => 64 bits

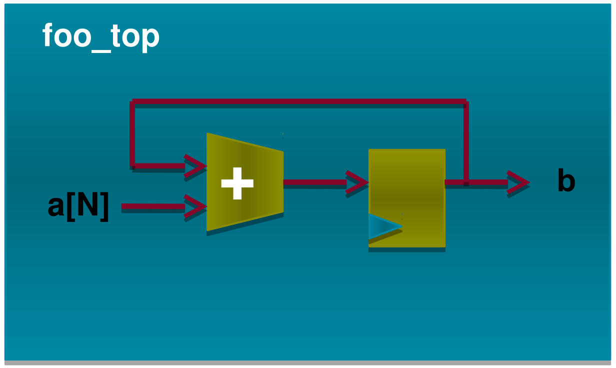
- **Arbitrary Precission Types**
  - C
    - `ap(u)int` => (1-1024)
  - C++:
    - `ap_(u)int` => (1-1024)
    - `ap_fixed`
  - C++ / SystemC:
    - `sc_(u)int` => (1-1024)
    - `sc_fixed`

Loops

- Rolled by default
  - Each iteration implemented in the same state
  - Each iteration implemented with the same resources




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



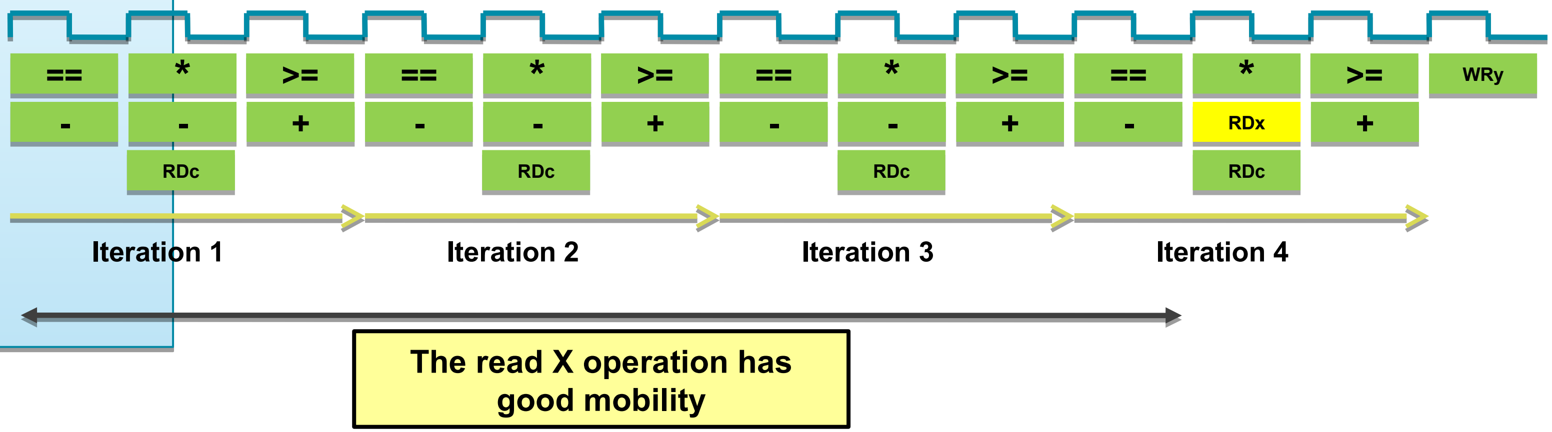
- Loops can be unrolled if their indices are statically determinable at elaboration time
  - Not when the number of iterations is variable
  - Result in more elements to schedule but greater operator mobility

Data Dependencies: Good

- 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
    - Input reads can be optionally registered

```
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;
}
```

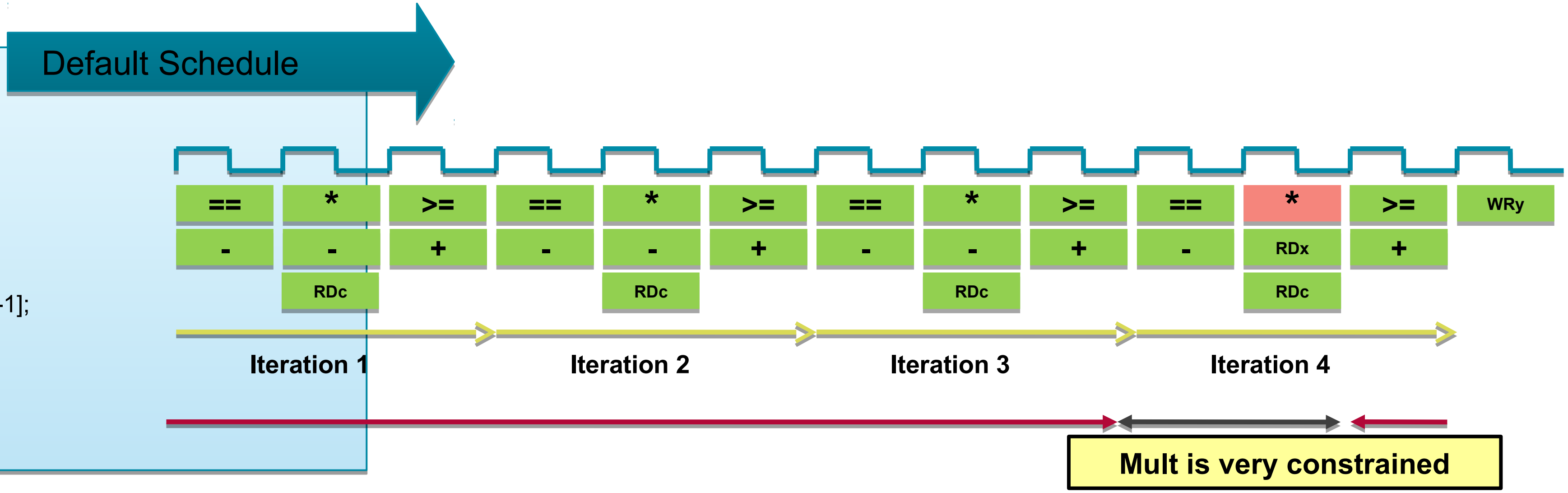
Default Schedule



Data Dependencies: Bad

- The final multiplication must occur before the read and final addition
- 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

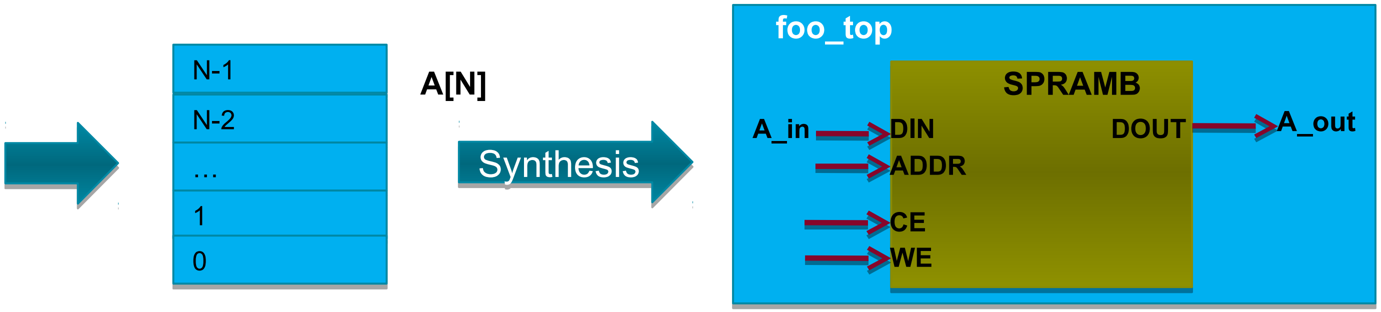
```
void fir (
 …
acc=0;
  loop: for (i=3;i>=0;i--) {
     if (i==0) {
       acc+=x*c[0];
       shift_reg[0]=x;
     } else {
        shift_reg[i]=shift_reg[i-1];
        acc+=shift_reg[i]*c[i];
     }
  }
  *y=acc;
}
```



Arrays

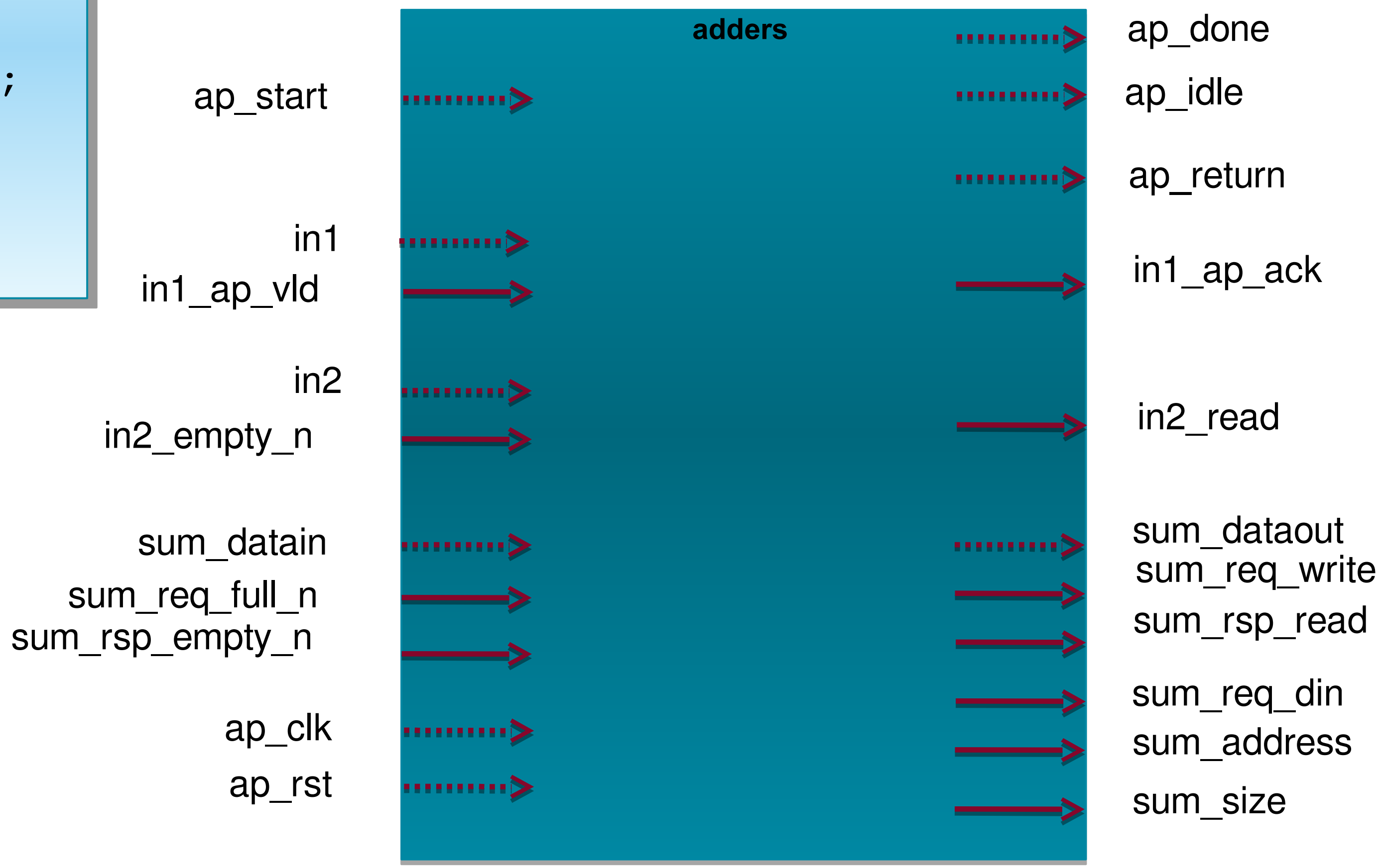
- By default implemeted as RAM
  - Dual port if performance can be improved otherwise Single Port RAM
  - optionally as a FIFO or registers bank
- Can be targeted to any memory resource in the library
- Can be merged with other arrays and reconfigured
- Arrays can be partitioned into individual elements
  - Implemented as smaller RAMs or registers

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



Top-Level IO Ports

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

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

    return  temp;
}
```



An example: Matrix Multiply

```
#define A_ROWS 16
#define A_COLS 16
#define B_ROWS 16
#define B_COLS 16

void mm(int in_a[A_ROWS][A_COLS],
        int in_b[A_COLS][B_COLS],
        int out_c[A_ROWS][B_COLS])
{
  int i,j, k;

  a_row_loop: for (i = 0; i < A_ROWS; i++) {

    b_col_loop: for (j = 0; j < B_COLS; j++) {
      int sum_mult = 0;

    a_col_loop: for (k = 0; k < A_COLS; k++)
      sum_mult += in_a[i][k] * in_b[k][j];

    out_c[i][j] = sum_mult;
    }
  }
}
```

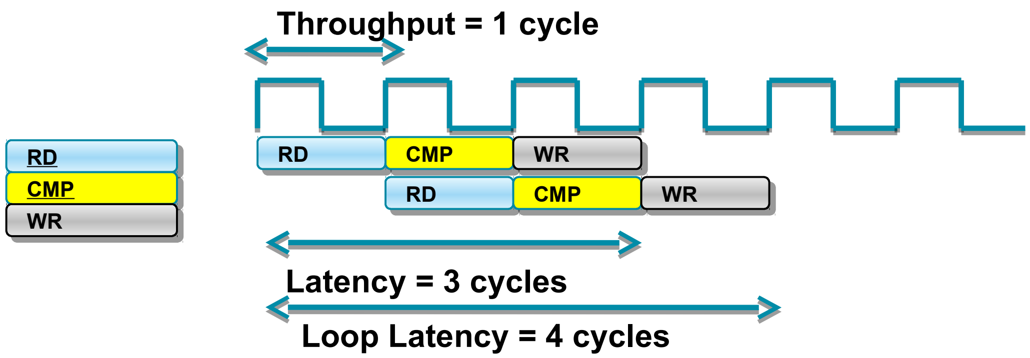
Clock cycle: 6.68 ns

Loop	Latency	Iteration latency	Trip count	Initiation interval
a_row_loop	37408	2338	16	0
b_col_loop	2336	146	16	0
a_col_loop	144	9	16	0

Resources	BRAM	DSP	FF	LUT
Total	0	4	207	170

Pipelined version

```
#define A_ROWS 16
#define A_COLS 16
#define B_ROWS 16
#define B_COLS 16

void mm(int in_a[A_ROWS][A_COLS],
        int in_b[A_COLS][B_COLS],
        int out_c[A_ROWS][B_COLS])
{
  int i,j, k;

  a_row_loop: for (i = 0; i < A_ROWS; i++) {

    b_col_loop: for (j = 0; j < B_COLS; j++) {
      int sum_mult = 0;

    a_col_loop: for (k = 0; k < A_COLS; k++)
      #pragma HLS pipeline
      sum_mult += in_a[i][k] * in_b[k][j];

    out_c[i][j] = sum_mult;
    }
  }
}
```



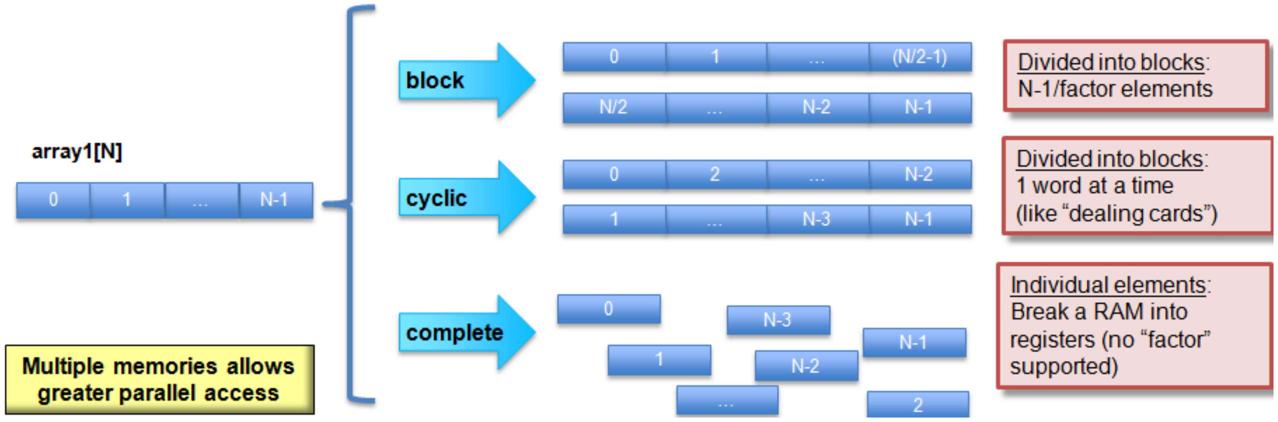
Clock cycle: 7.83 ns

Loop	Latency	Iteration latency	Trip count	Initiation interval
all_fused	4105	11	4096	1

Resources	BRAM	DSP	FF	LUT
Total	0	4	45	21

Parallel Dot-Product MM

```
#define A_ROWS 16
#define A_COLS 16
#define B_ROWS 16
#define B_COLS 16

void mm(int in_a[A_ROWS][A_COLS],
        int in_b[A_COLS][B_COLS],
        int out_c[A_ROWS][B_COLS])
{
#pragma HLS ARRAY_PARTITION DIM=2 VARIABLE=in_a complete
#pragma HLS ARRAY_PARTITION DIM=1 VARIABLE=in_b complete

  int i,j, k;

  a_row_loop: for (i = 0; i < A_ROWS; i++) {

    b_col_loop: for (j = 0; j < B_COLS; j++) {
      #pragma HLS pipeline
      int sum_mult = 0;

    a_col_loop: for (k = 0; k < A_COLS; k++)
      sum_mult += in_a[i][k] * in_b[k][j];

    out_c[i][j] = sum_mult;
    }
  }
}
```



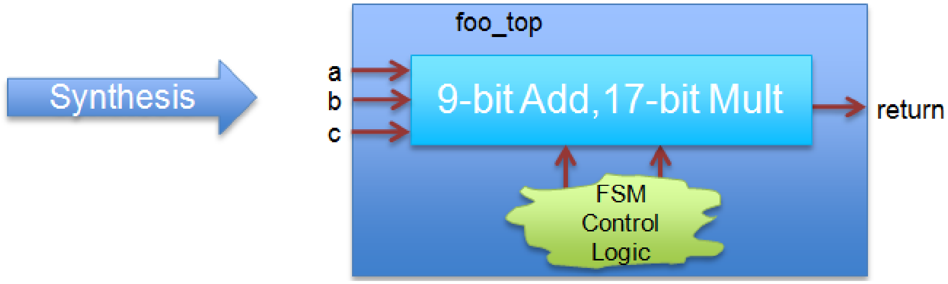
Clock cycle: 7.23 ns

Loop	Latency	Iteration latency	Trip count	Initiation interval
all_fused	264	10	256	1

Resources	BRAM	DSP	FF	LUT
Total	0	64	720	336

18-bit Parallel Dot-Product MM

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




```
#define A_ROWS 16
#define A_COLS 16
#define B_ROWS 16
#define B_COLS 16

#include <ap_cint.h>
void mm(int18 in_a[A_ROWS][A_COLS],
        int18 in_b[A_COLS][B_COLS],
        int18 out_c[A_ROWS][B_COLS])
{
#pragma HLS ARRAY_PARTITION DIM=2 VARIABLE=in_a complete
#pragma HLS ARRAY_PARTITION DIM=1 VARIABLE=in_b complete

  int i,j, k;

  a_row_loop: for (i = 0; i < A_ROWS; i++) {

    b_col_loop: for (j = 0; j < B_COLS; j++) {
     #pragma HLS pipeline
     int18 sum_mult = 0;

    a_col_loop: for (k = 0; k < A_COLS; k++)
     sum_mult += in_a[i][k] * in_b[k][j];

    out_c[i][j] = sum_mult;
    }
  }
}
```

Clock cycle: 7.64 ns

Loop	Latency	Iteration latency	Trip count	Initiation interval
all_fused	260	6	256	1

Resources	BRAM	DSP	FF	LUT
Total	0	16	560	214

Pipelined Floating-Point MM

```
#define A_ROWS 16
#define A_COLS 16
#define B_ROWS 16
#define B_COLS 16

void mm(float in_a[A_ROWS][A_COLS],
        float in_b[A_COLS][B_COLS],
        float out_c[A_ROWS][B_COLS])
{
  int i,j, k;

  a_row_loop: for (i = 0; i < A_ROWS; i++) {

    b_col_loop: for (j = 0; j < B_COLS; j++) {
      #pragma HLS pipeline
      float sum_mult = 0;

    a_col_loop: for (k = 0; k < A_COLS; k++)
      sum_mult += in_a[i][k] * in_b[k][j];

    out_c[i][j] = sum_mult;
    }
  }
}
```

Clock cycle: 8.03 ns

Loop	Latency	Iteration latency	Trip count	Initiation interval
all_fused	2125	8	256	8

Resources	BRAM	DSP	FF	LUT
Total	0	10	696	1424

MM Interface Synthesis

Function activation interface

Can be disabled ap_control_none

Synthesized memory ports

Also dual-ported

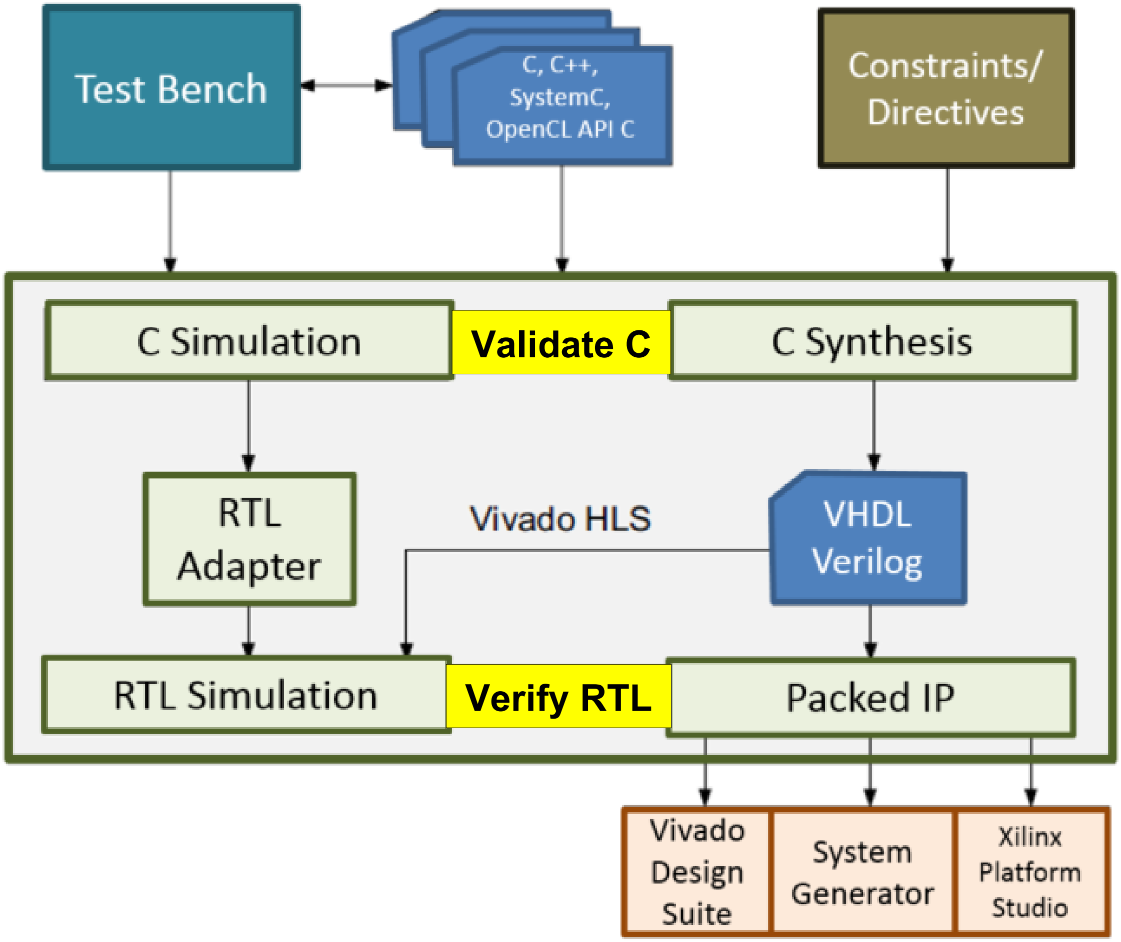
In the array partitioned Version, 16 mem ports. One per partial product

RTL ports	dir	bits	Protocol	C Type
ap_clk	in	1	ap_ctrl_hs	return value
ap_rst	in	1	ap_ctrl_hs	return value
ap_start	in	1	ap_ctrl_hs	return value
ap_done	out	1	ap_ctrl_hs	return value
ap_idle	out	1	ap_ctrl_hs	return value
ap_ready	out	1	ap_ctrl_hs	return value
in_a_address0	out	8	ap_memory	array
in_a_ce0	out	1	ap_memory	array
in_a_q0	in	32	ap_memory	array
in_b_address0	out	8	ap_memory	array
in_b_ce0	out	1	ap_memory	array
in_b_q0	in	32	ap_memory	array
in_c_address0	out	8	ap_memory	array
in_c_ce0	out	1	ap_memory	array
in_c_we0	out	1	ap_memory	array
in_c_d0	out	32	ap_memory	array

Validation Flow

- 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



Test benches

- The test bench should be in a separate file
- Or excluded from synthesis
  - The Macro __SYNTHESIS__ can be used to isolate code which will not be synthesized

```c
// test.c
#include <stdio.h>
void test (int d[10]) {
  int acc = 0;
  int i;
  for (i=0;i<10;i++) {
    acc += d[i];
    d[i] = acc;
  }
}
#ifndef __SYNTHESIS__
int main () {
  int d[10], i;
  for (i=0;i<10;i++) {
    d[i]   = i;
  }
  test(d);
  for (i=0;i<10;i++) {
    printf("%d %d\n", i, d[i]);
  }
  return 0;
}
#endif
```

Design to be synthesized

Test Bench
Nothing in this ifndef will be read by Vivado HLS
(will be read by gcc)

Test benches

- Ideal test bench
  - Should be self checking
    - RTL verification will re-use the C test bench
  - If the test bench is self-checking
    - Allows RTL Verification to be run without a requirement to check the results again
  - RTL verification “passes” if the test bench return value is 0 (zero)
    - Actively return a 0 if the simulation passes

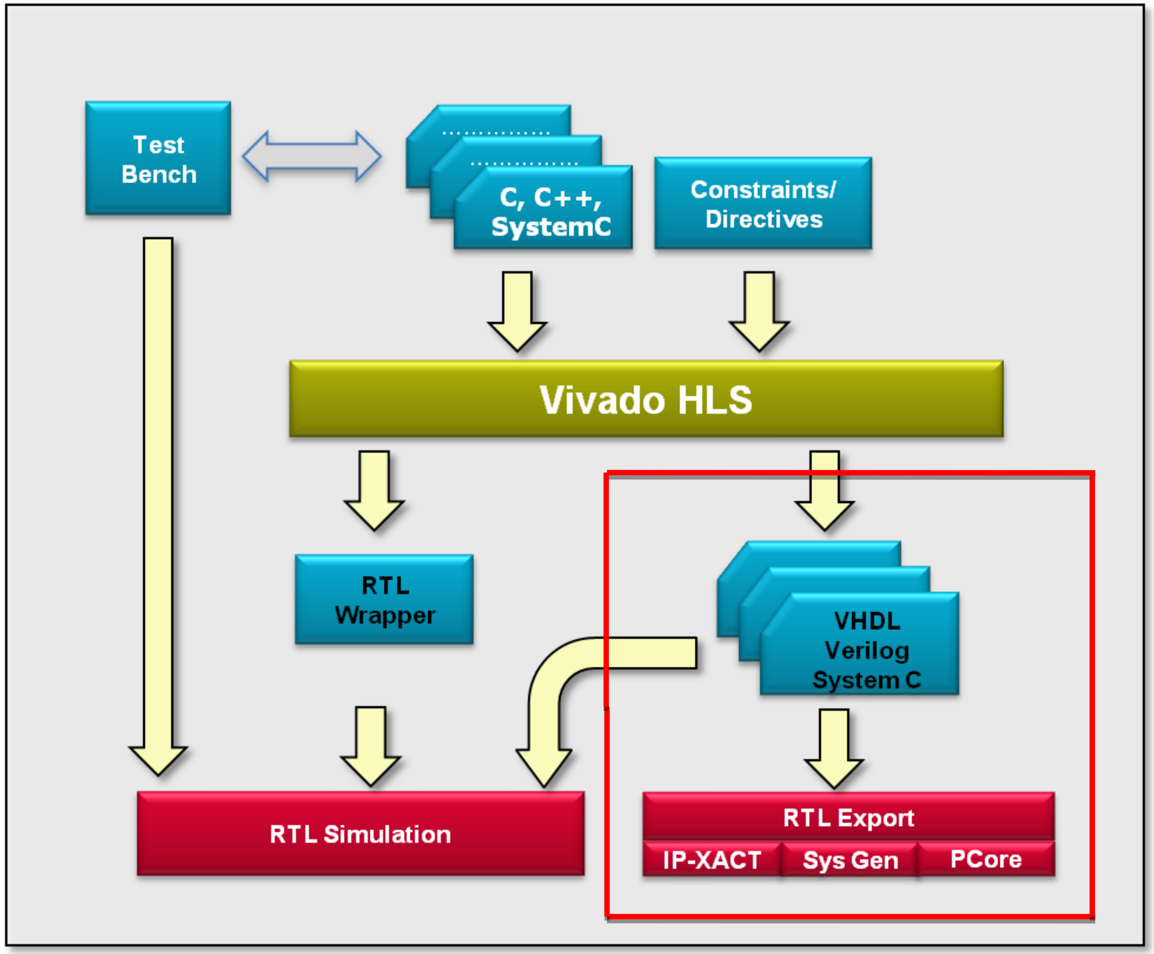
```
int main () {
  // Compare results
  int ret = system("diff --brief -w test_data/output.dat test_data/output.golden.dat");
  if (ret != 0) {
        printf("Test failed !!!\n", ret); return 1;
  } else {
        printf("Test passed !\n", ret); return 0;
  }
```

The –w option ensures the “new-line” does not cause a difference between Windows and Linux files

  - Non-synthesizable constructs may be added to a synthesize function if __SYNTHESIS__ is used

```
#ifndef __SYNTHESIS__
   image_t *yuv = (image_t *)malloc(sizeof(image_t));
#else // Workaround malloc() calls w/o changing rest of code
   image_t _yuv;
#endif
```

RTL Export



RTL output in Verilog, VHDL and SystemC

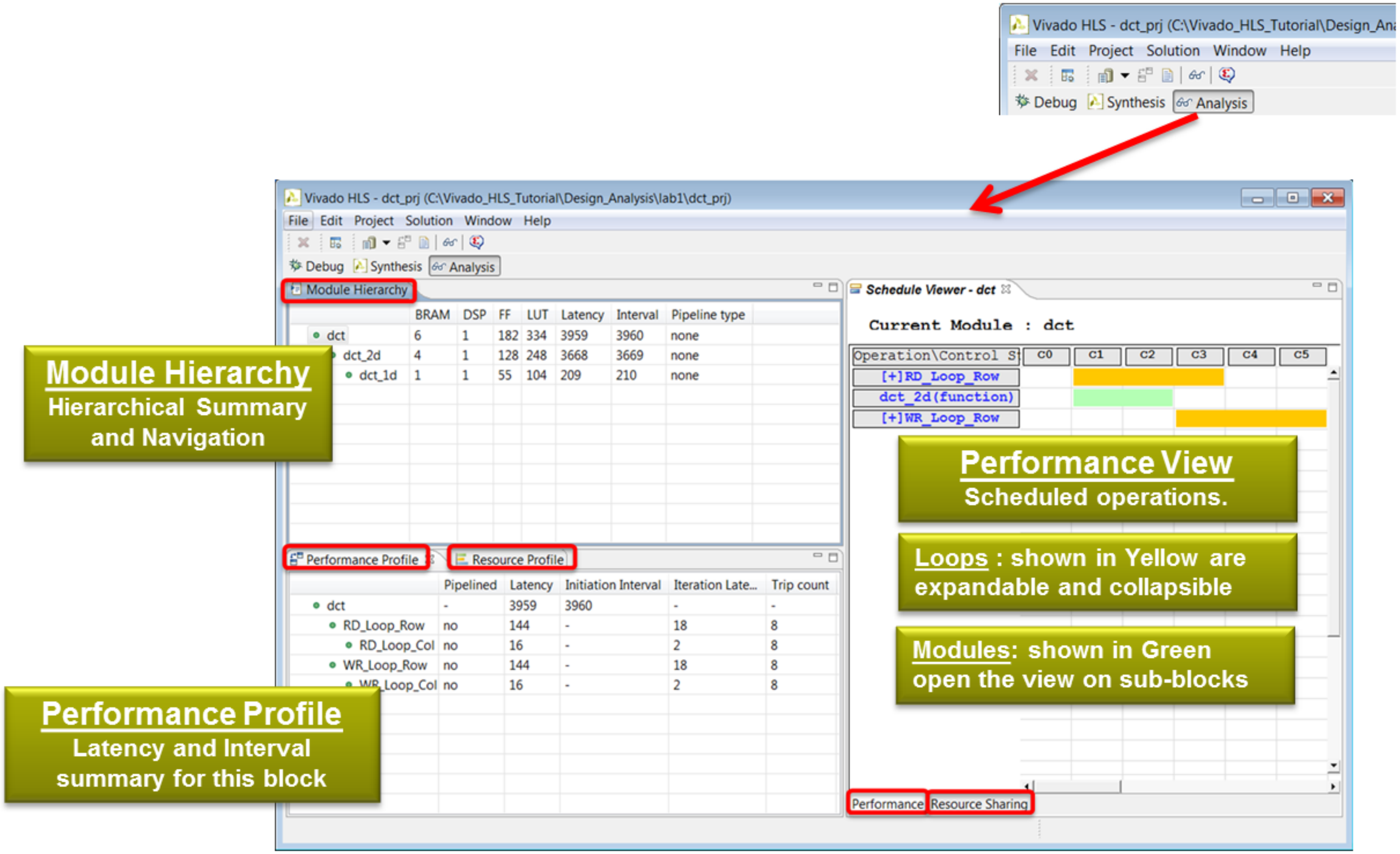
Scripts created for RTL synthesis tools

RTL Export to IP-XACT, SysGen, and Pcore formats

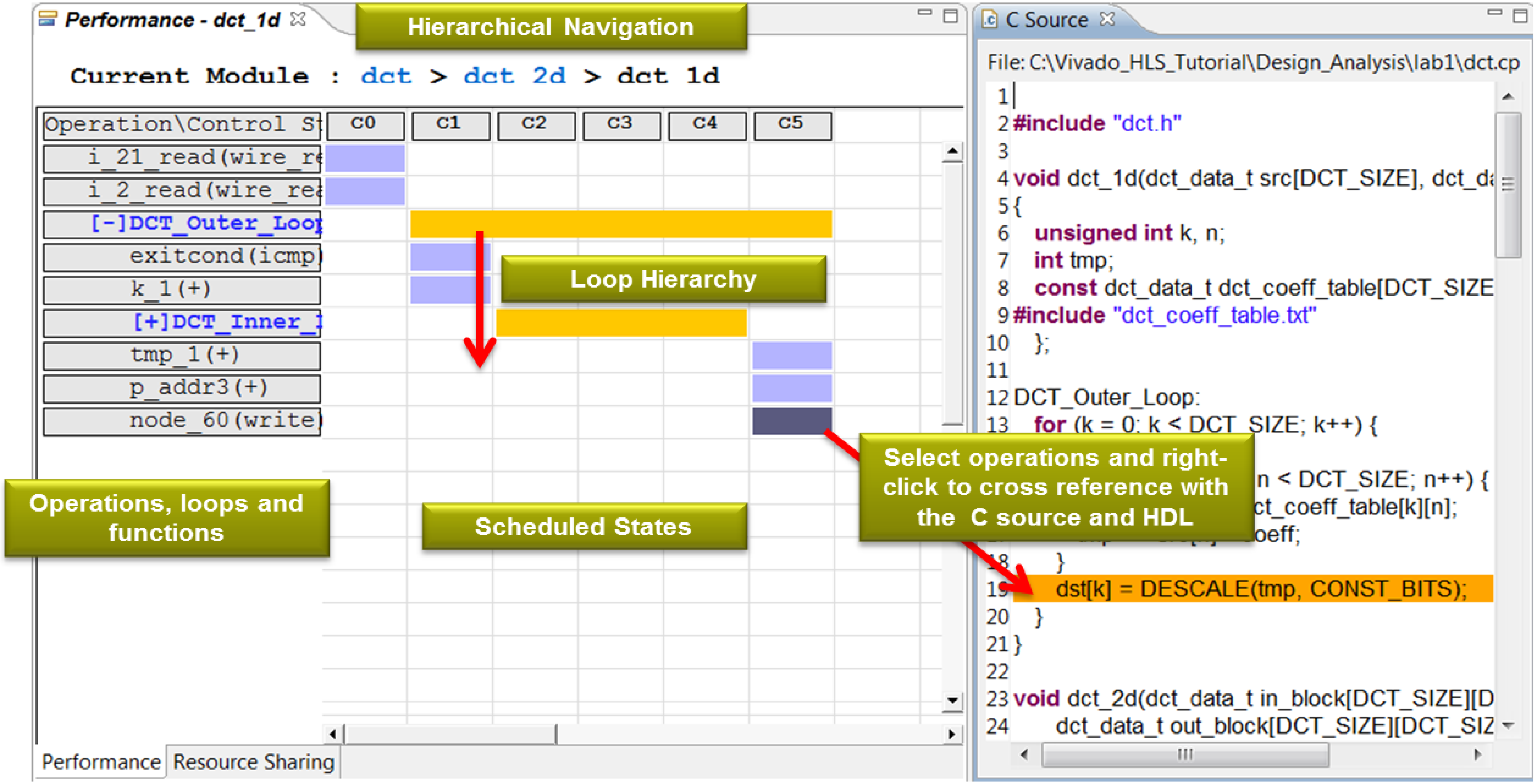
IP-XACT and SysGen => Vivado HLS for 7 Series and Zynq families
PCore => Only Vivado HLS Standalone for all families

Design Analysis

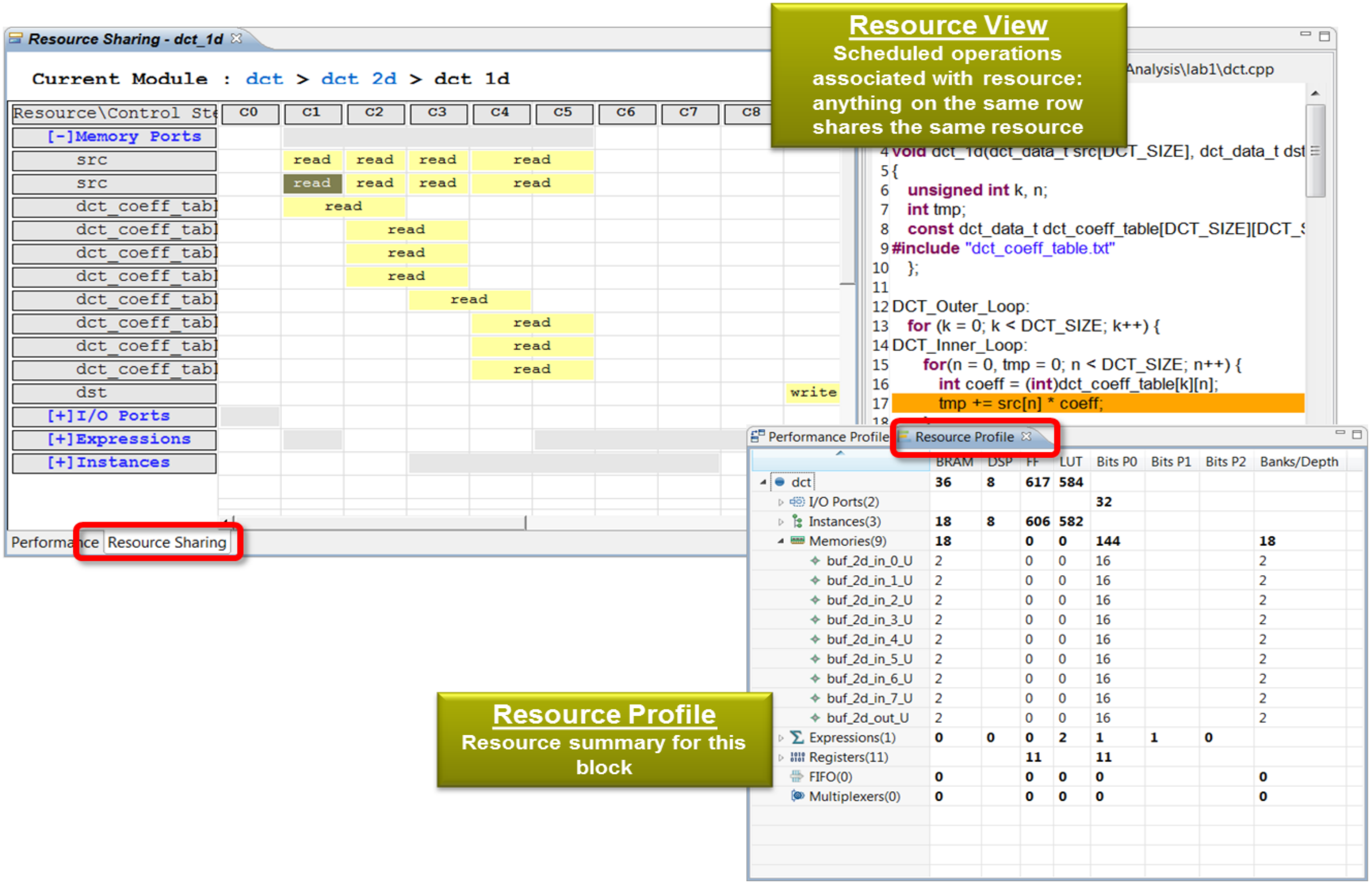
- Perspective for design analysis
  - Allows interactive analysis



Performance Analysis



Resources Analisys



References

- M. Fingeroff, “High-Level Synthesis Blue Book”, X libris Corporation, 2010
- P. Coussy, A. Morawiec, “High-Level Synthesis:

  from Algorithm to Digital Circuit”, Springer, 2008
- “High-Level Synthesis Flow on Zynq” Course materials

  from the Xilinx University Program, 2016