The Abdus Salam
International Centre
for Theoretical Physics

2499-20

International Training Workshop on FPGA Design for Scientific Instrumentation and Computing

11 - 22 November 2013

High-Level Synthesis: how to improve FPGA design productivity

RINCON CALLE Fernando
Universidad de Castilla la Mancha
Escuela Superior de Informatica
Departamento de Tecnologias y Sistemas de la Informacion
Ciudad Real
SPAIN

UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

International Training Workshop on FPGA Design For Scientific Instrumentation And Computing

High-Level Synthesis: how to improve FPGA design productivity

Fernando.Rincon@uclm.es

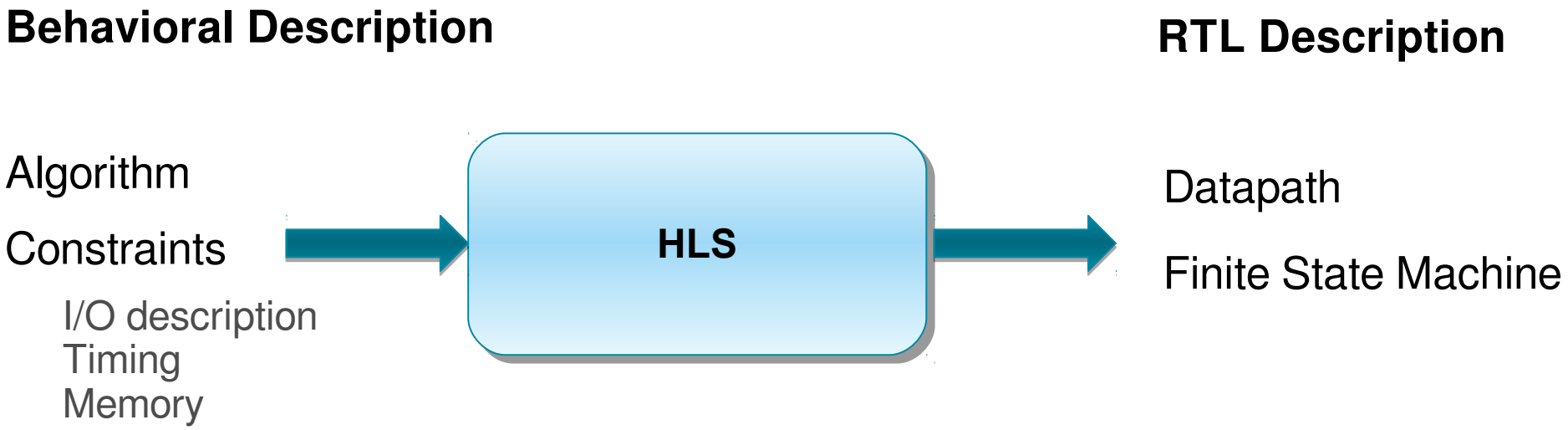
UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

Contenidos

- **Introduction to High-Level Synthesis**
- **Vivado HLS**
  - **Key elements**
  - **Matrix Multiplication example**
  - **Validation & Verification**
  - **RTL export**

What is High-Level Synthesis?

- Compilation of behavioral algorithms into RTL descriptions



UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

Why HLS?

- Need for productivity improvement at design level
- 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

UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

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

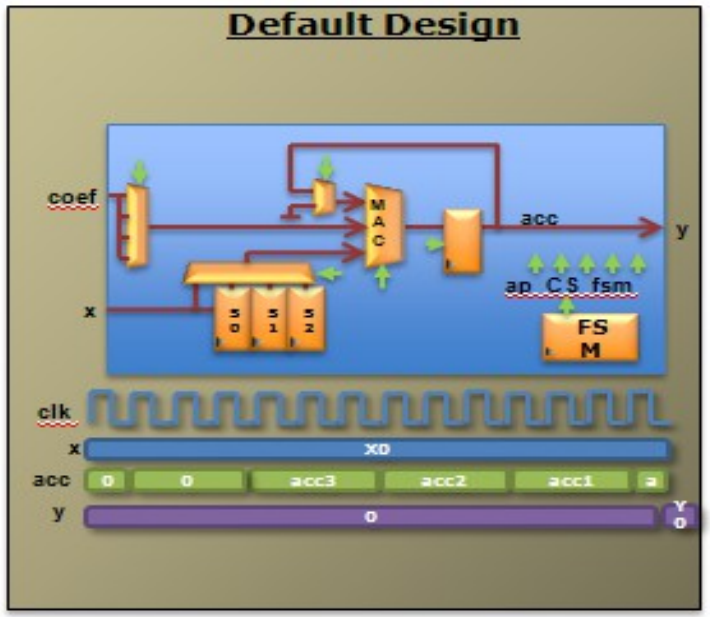
Design Space Exploration

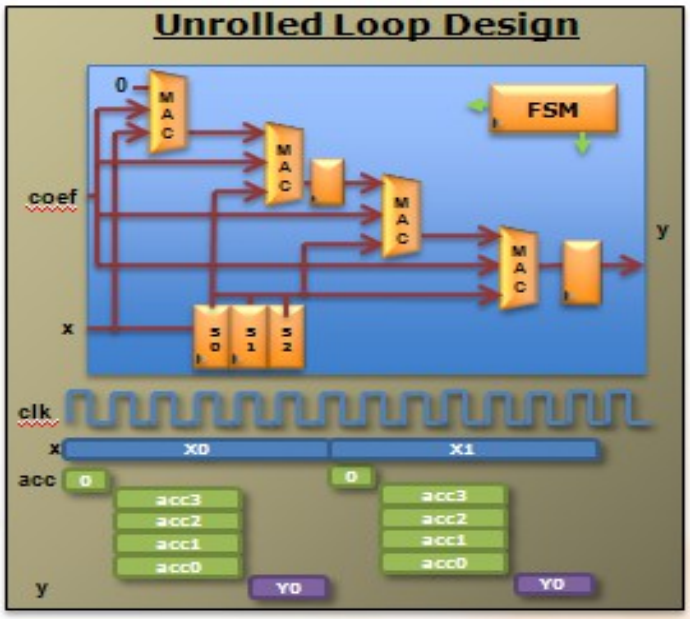
```
…
 loop: for (i=3;i>=0;i--) {
    if (i==0) {
      acc+=x*c[0];
      shift_reg[0]=x;
    } else {
       shift_reg[i]=shift_reg[i-1];
       acc+=shift_reg[i]*c[i];
    }
```

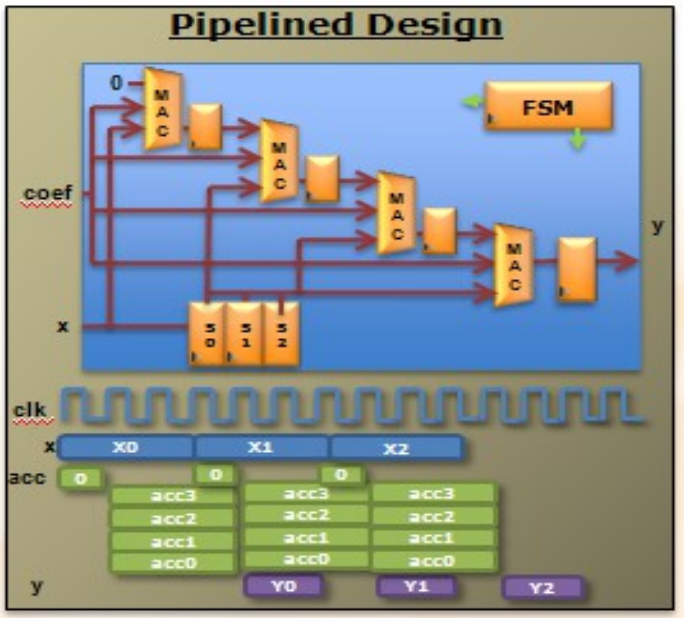
Ssame hardware is used for each loop iteration :
- **Small area**
- **Long latency**
- **Low throughput**

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

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;
}
```

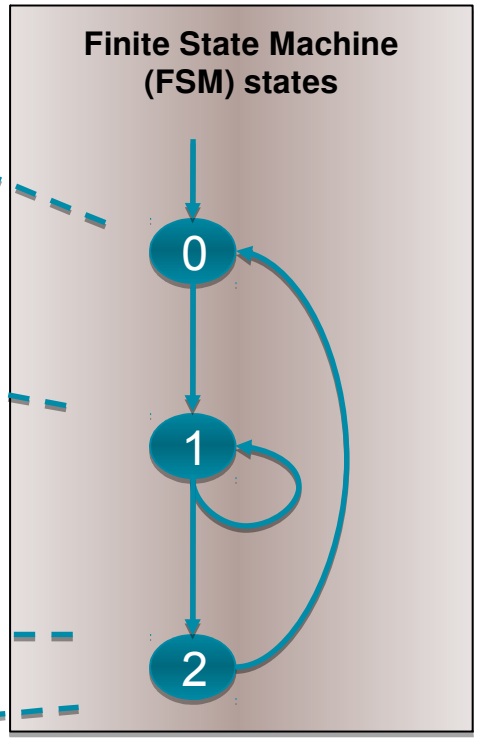
Function Start

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

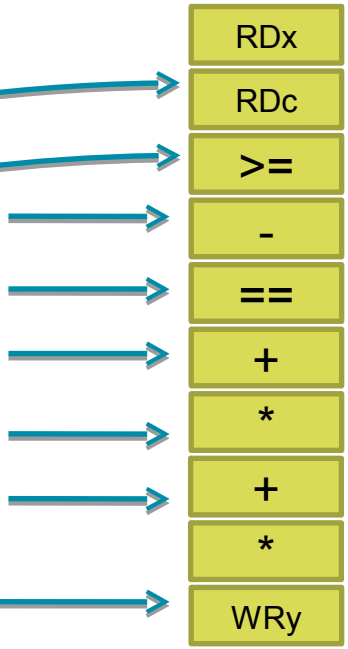
This behavior is extracted into a hardware state machine

How does it work II

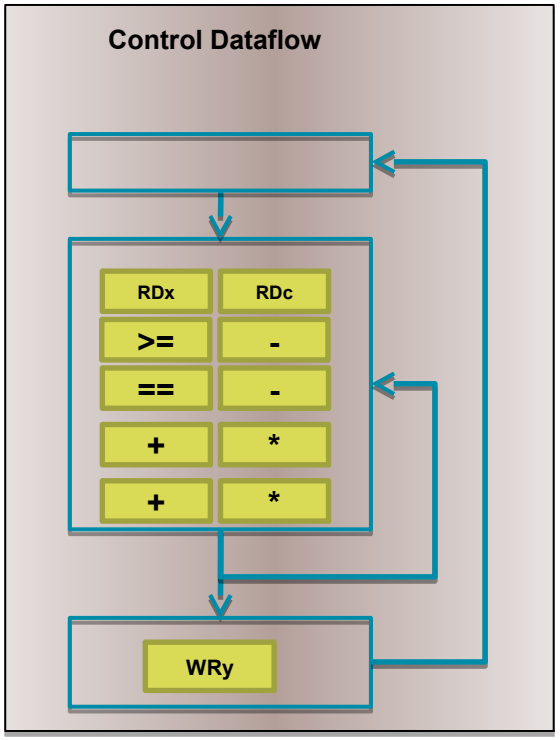
Code

```
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 & Datapath Behavior



From any C code example ..

Operations are extracted...

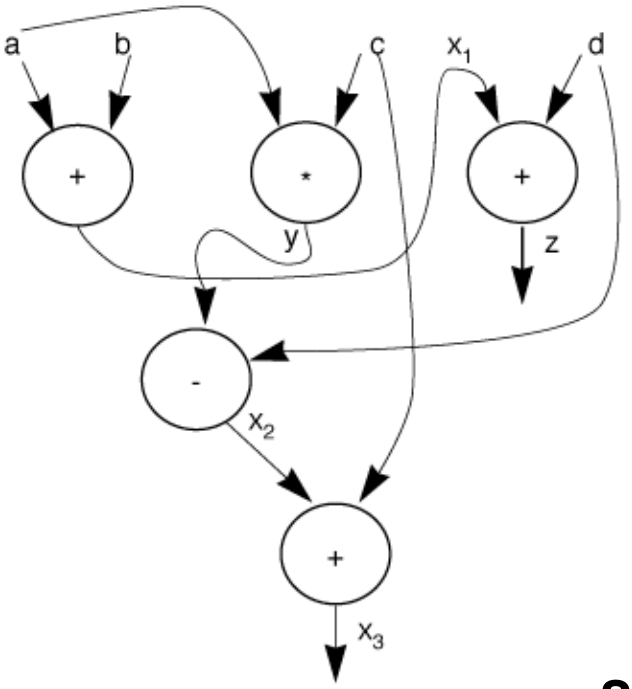
A unified control dataflow behavior is created.

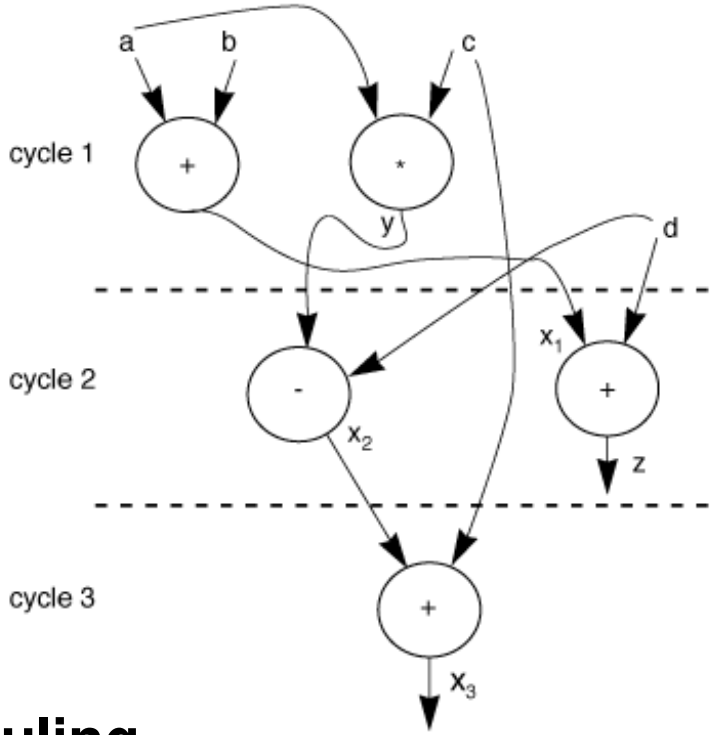
Scheduling

Original code:

```
x <= a + b;
y <= a * c;
z <= x + d;
x <= y - d;
x <= x + c;
```

Operations are mapped into clock cycles

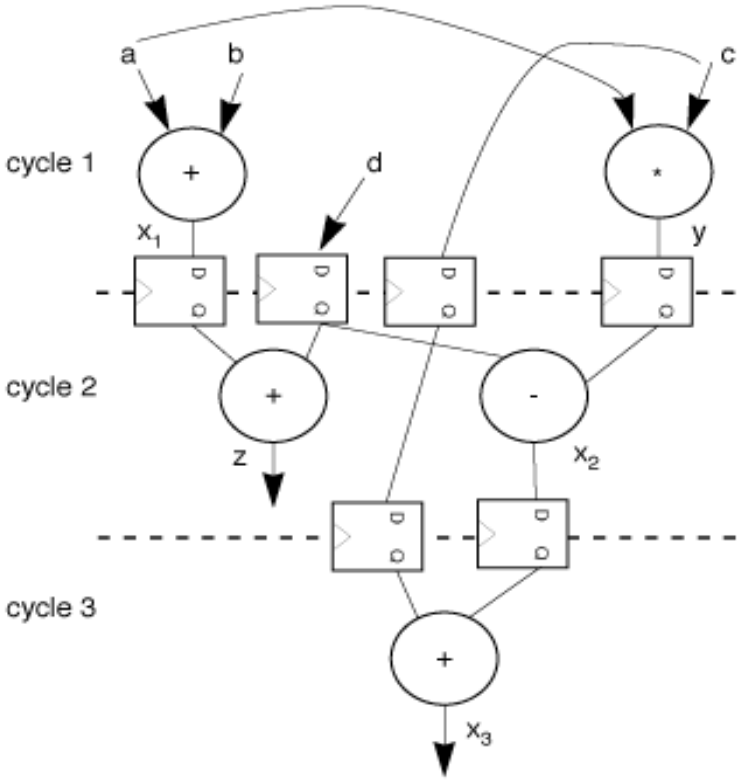


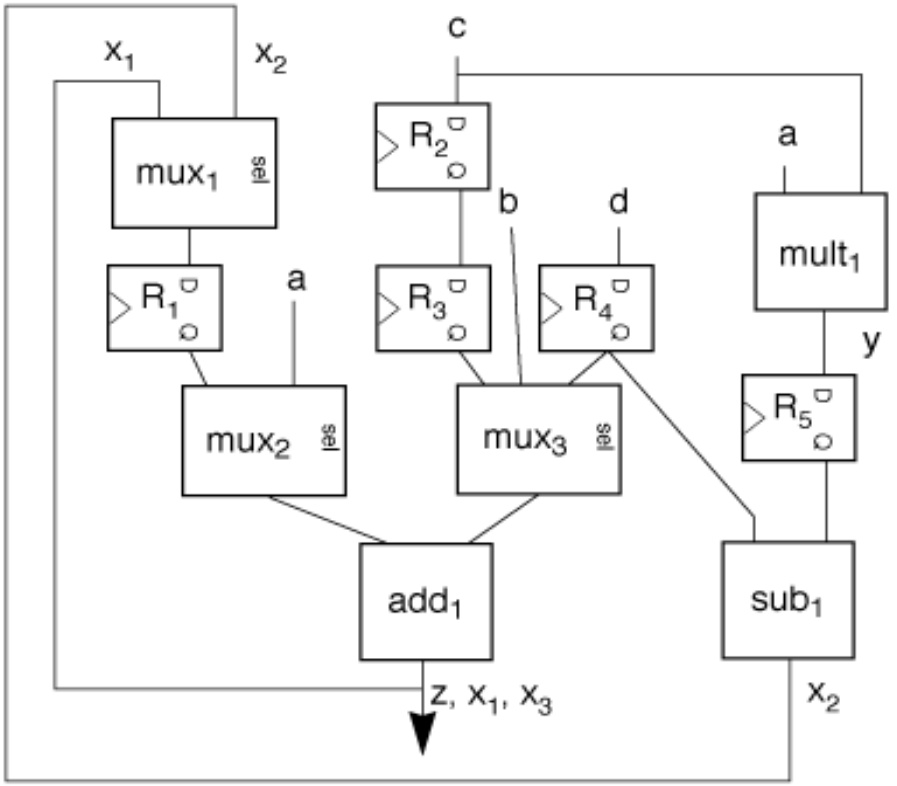


scheduling

Allocation & Binding

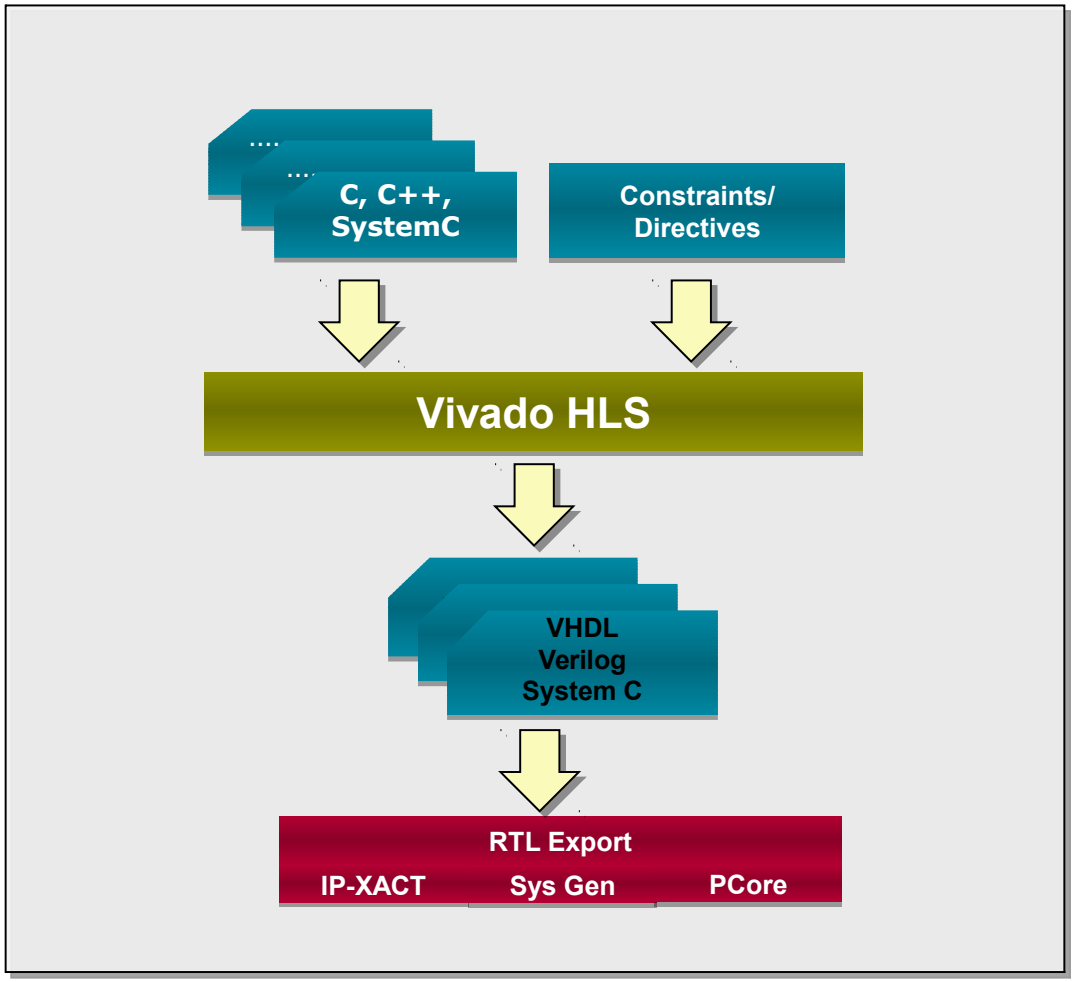
Operations are assigned to available functional units

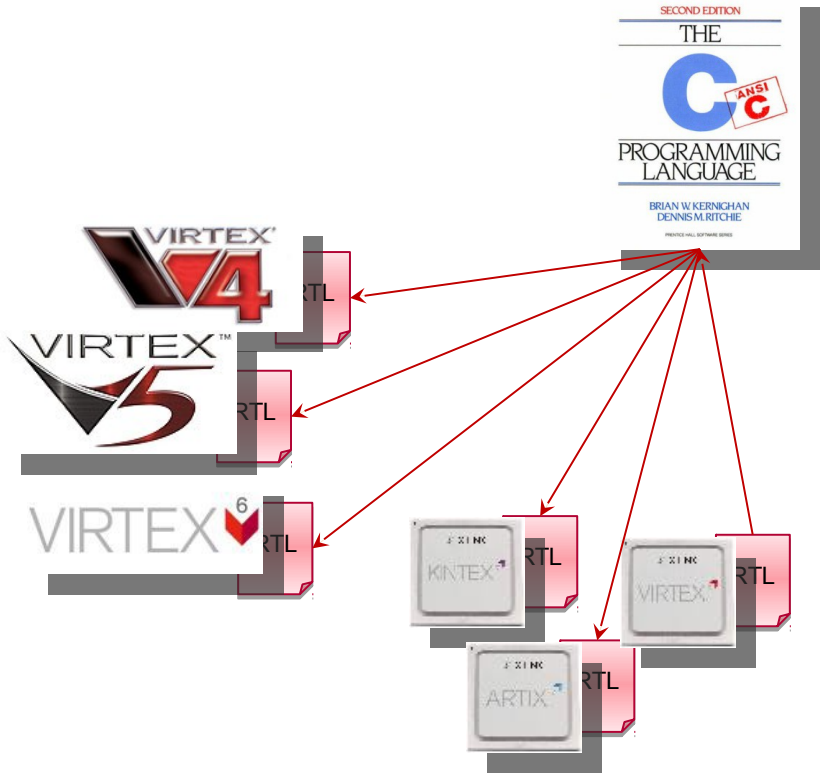




Vivado HLS

- High Level Synthesis Suite from Xilinx





Key attributes of 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;
}
```

Only one top-level function is allowed

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

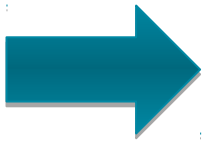
Functions & RTL Hierarchy

Each function is translated into an RTL block

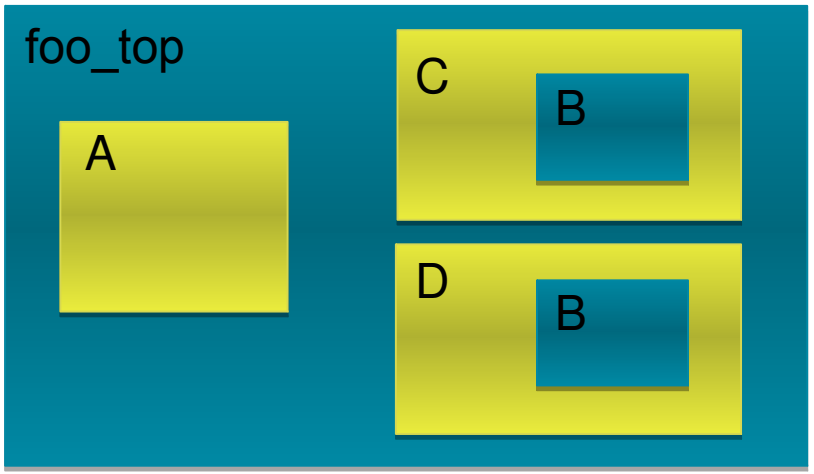
Source Code

```
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



Tip

Functions can be shared or inlined

Types

Define the size of the hardware used

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

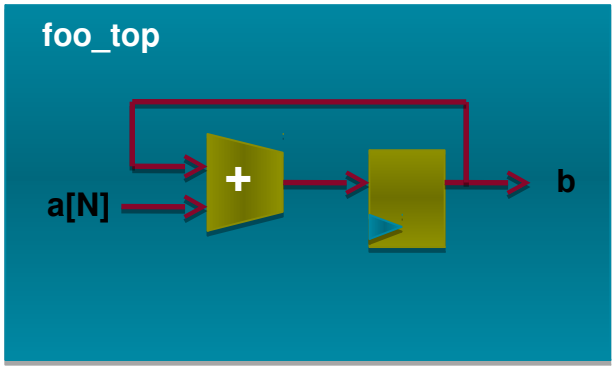
C:	ap(u)int types (1-1024)
C++:	ap_(u)int types (1-1024) ap_fixed types
C++/SystemC:	sc_(u)int types (1-1024) sc_fixed types

Loops

Loops are rolled by default


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

Synthesis



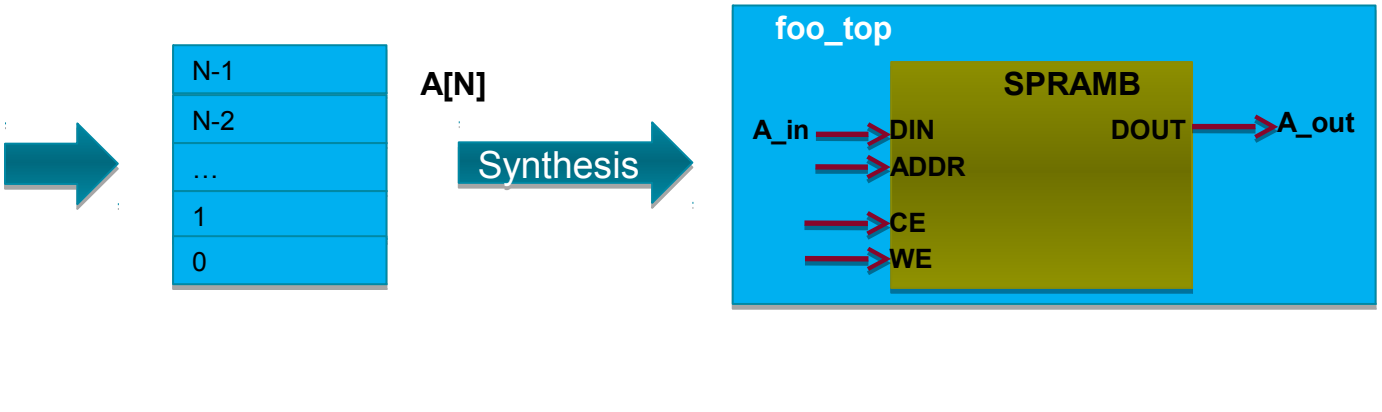
Tip

Unrolling is possible if the number of iterations is resolved at compile time

Arrays

By default implemeted as RAM, optionally as a FIFO or registers bank

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

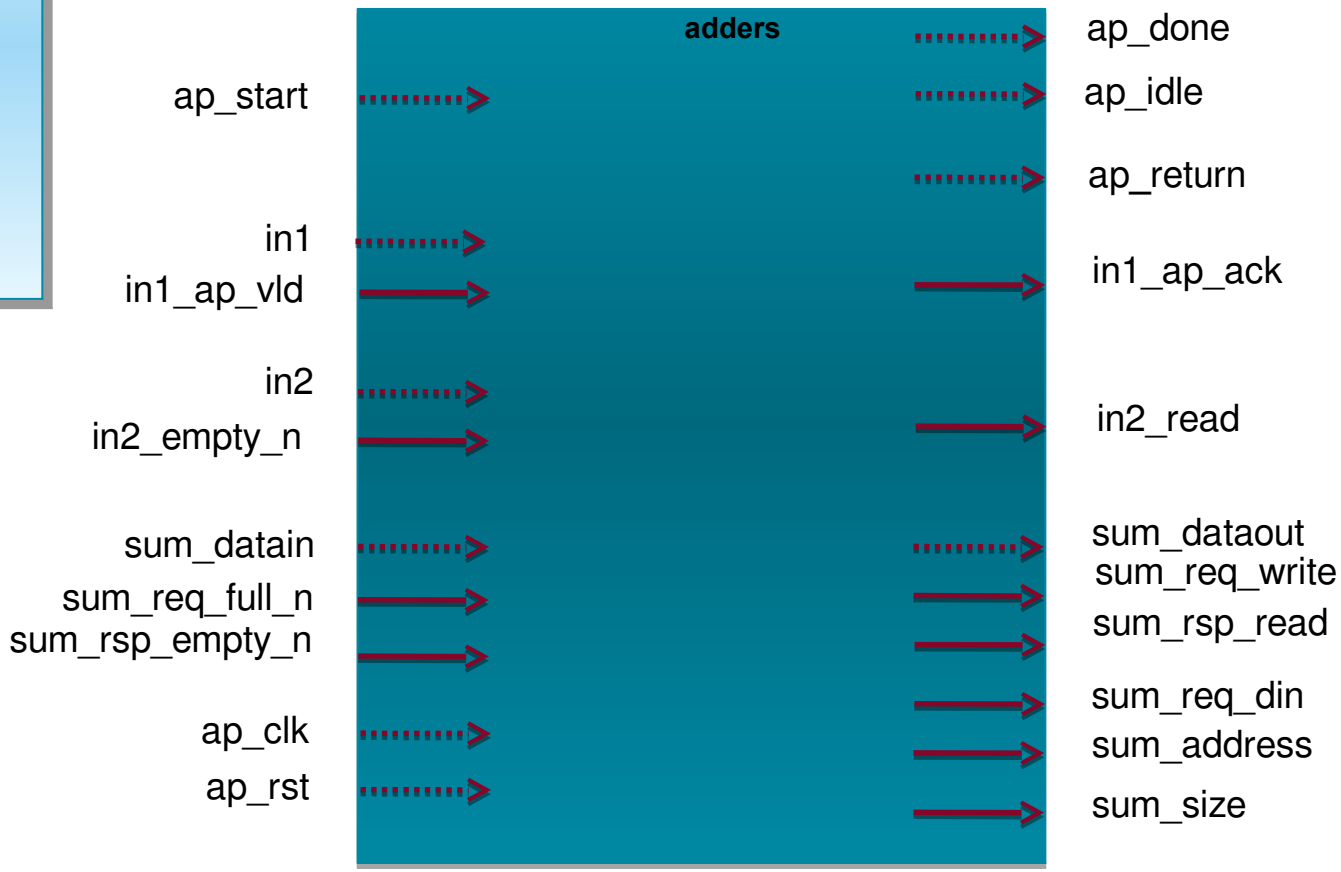


Tip

Can be merged and partitioned

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;
    }
  }
}
```

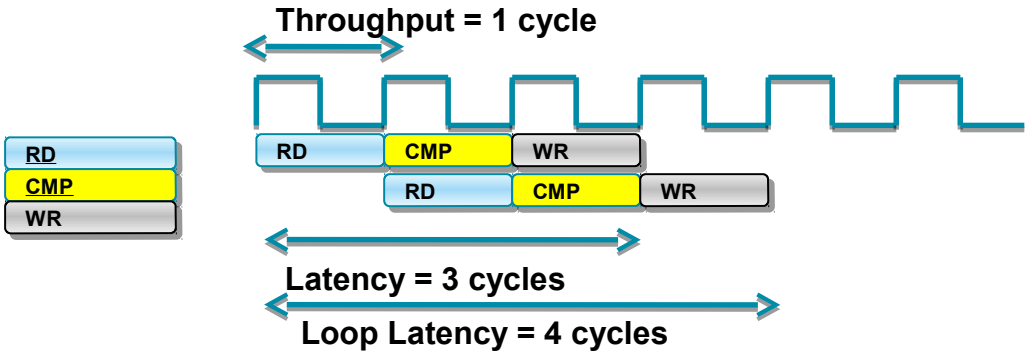
Clock cycle: 6.68 ns

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;
    }
  }
}
```

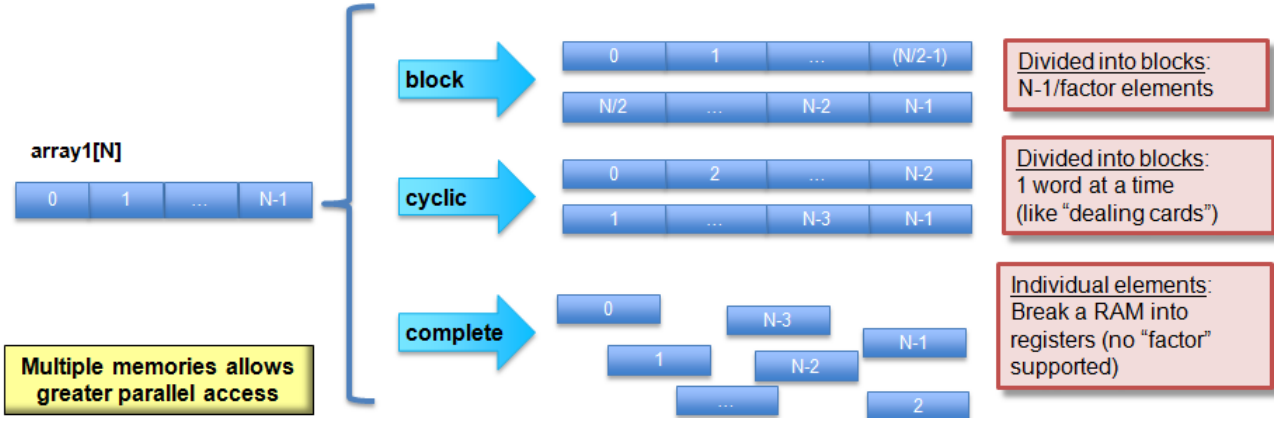


Clock cycle: 7.83 ns

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

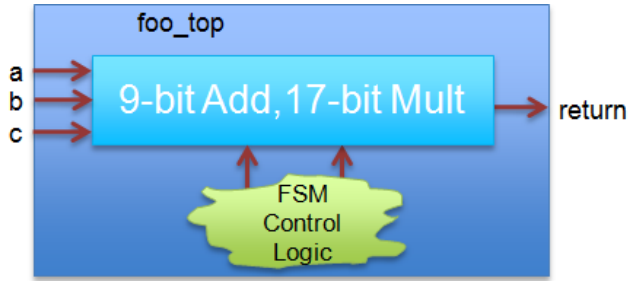
Resources	BRAM	DSP	FF	LUT
Total	0	64	720	336

18-bit Parallel Dot-Product MM

```
#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;
    }
  }
}
```

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

Synthesis



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

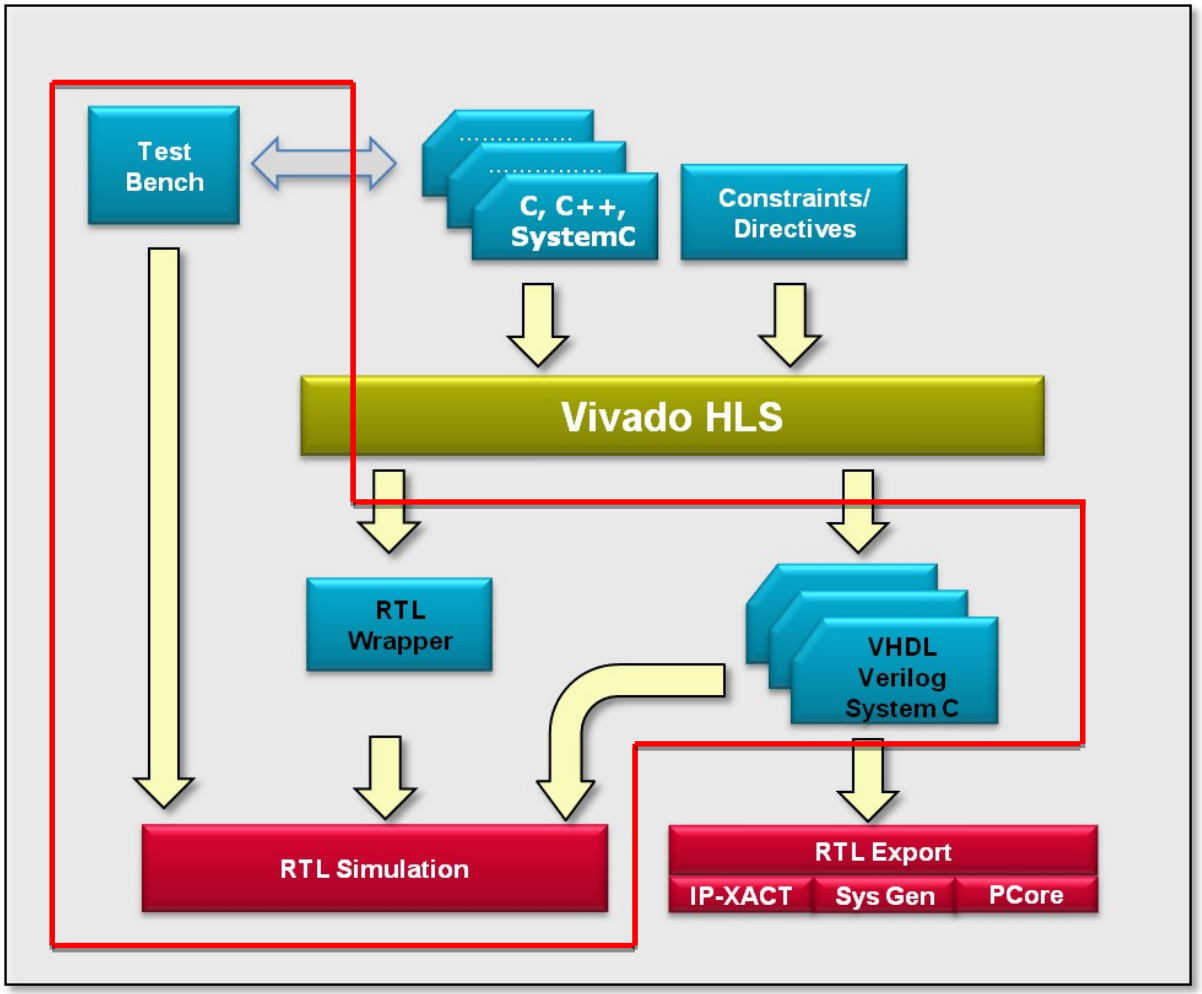
Synthesized memory ports

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

Verification & Validation



- RTL output in Verilog, VHDL and SystemC
- Automatic re-use of the C-level test bench
- RTL verification can be executed from within Vivado HLS
- Support for 3rd party HDL simulators in automated flow

RTL cosimulation

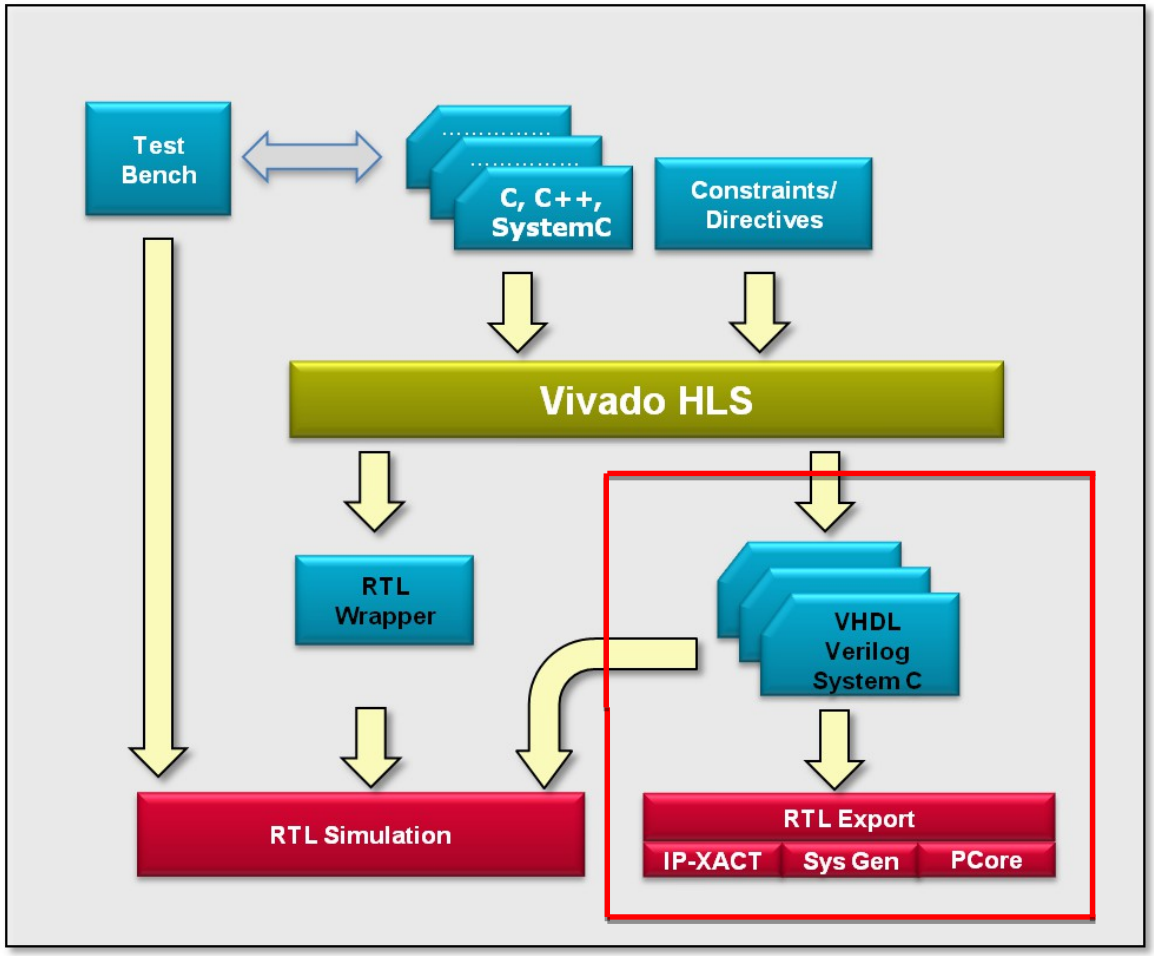
Automatic SystemC wrappers are created to reuse the C test-bench

Tip

No RTL test-bech is created

RTL export

Synthesized designs can be exported to pcores integrables into the system-level design



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 and Standalone for 7-Series families
PCore => Only Vivado HLS Standalone for all families

System integration

Different types of bus interfaces are supported, depending on the type of access

Supported Interface | Unsupported Interface

Pcore Interface							Argument Type	Variable			Pointer Variable			Array			Reference Variable		
NPI	FSL	PLB		AXI				Pass-by- value			Pass-by-reference			Pass-by-reference			Pass-by-reference		
		Slave	Master	Stream	Slave	Master	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									

UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

Conclusions

- HLS has already reached maturity
- Increased design productivity
  - code reuse vs. core reuse
  - Integration with high-level verification flows
  - quick design space exploration
- But there is still margin for improvement
  - Hw/Sw integration
  - Automate most of design decisions currently done by the designer

UCLM
UNIVERSIDAD DE CASTILLA-LA MANCHA

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 Workshop” Course materials from the Xilinx University Program, 2013