FPGA programming
An Introduction to High-Level Synthesis (HLS)

Prof. Andrea Marongiu
(andrea.marongiu@unimore.it)

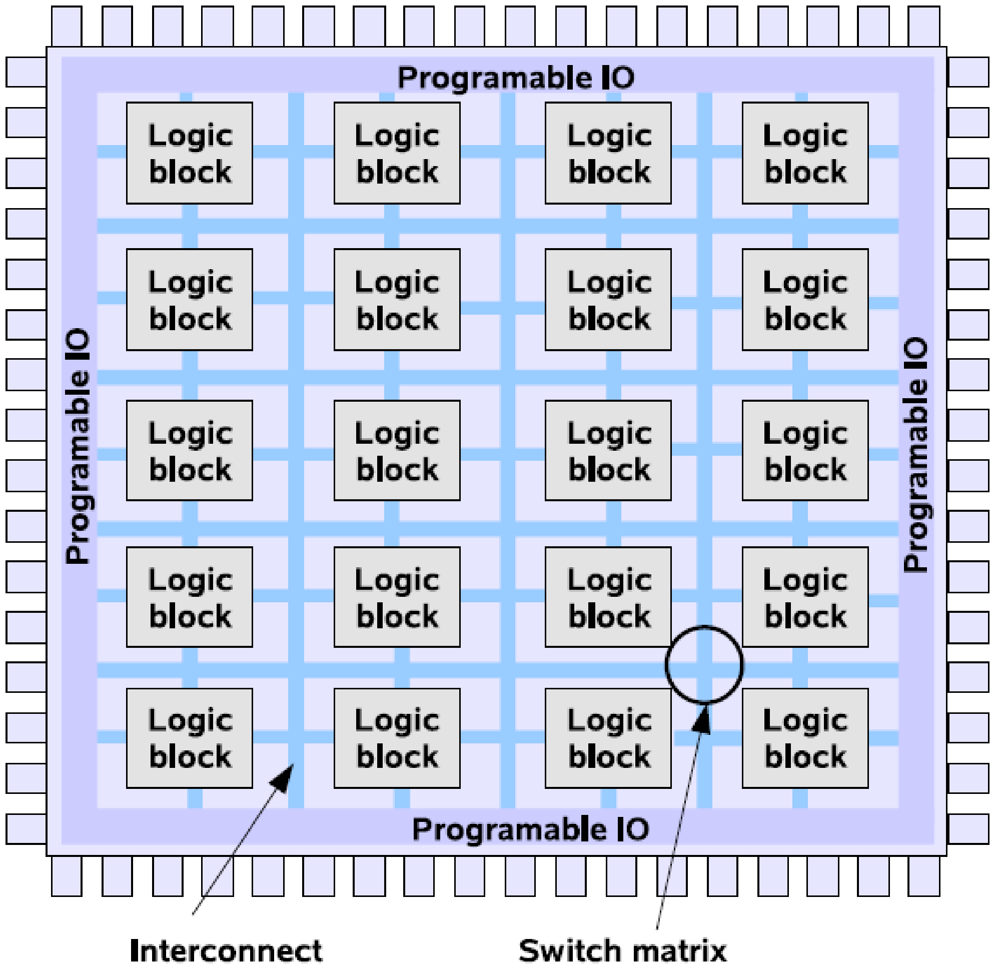
Courtesy:
Mark Manwaring
Kia Bazargan
Giovanni De Micheli
Gupta
Youn-Long Lin
Giuseppe Tagliavini

Also includes slides and contents from:
"*Introduction to High-Level Synthesis with Vivado HLS*" by Xilinx
"High-level Synthesis and System Synthesis" by Camposano, J. Hofstede, Knapp, MacMillen Lin
"Software/Hardware Codesign" course at George Mason University by prof. Kris Gaj

FPGA Architecture (recap)

The basic structure of an FPGA is composed of the following elements:

- **Look-up table (LUT)**: This element performs **logic** operations
- **Flip-Flop (FF)**: This register element **stores** the result of the LUT
- **Wires**: These elements connect elements to one another, both logic and clock
- **Input/Output (I/O) pads**: These physically available ports get signals in and out of the FPGA.

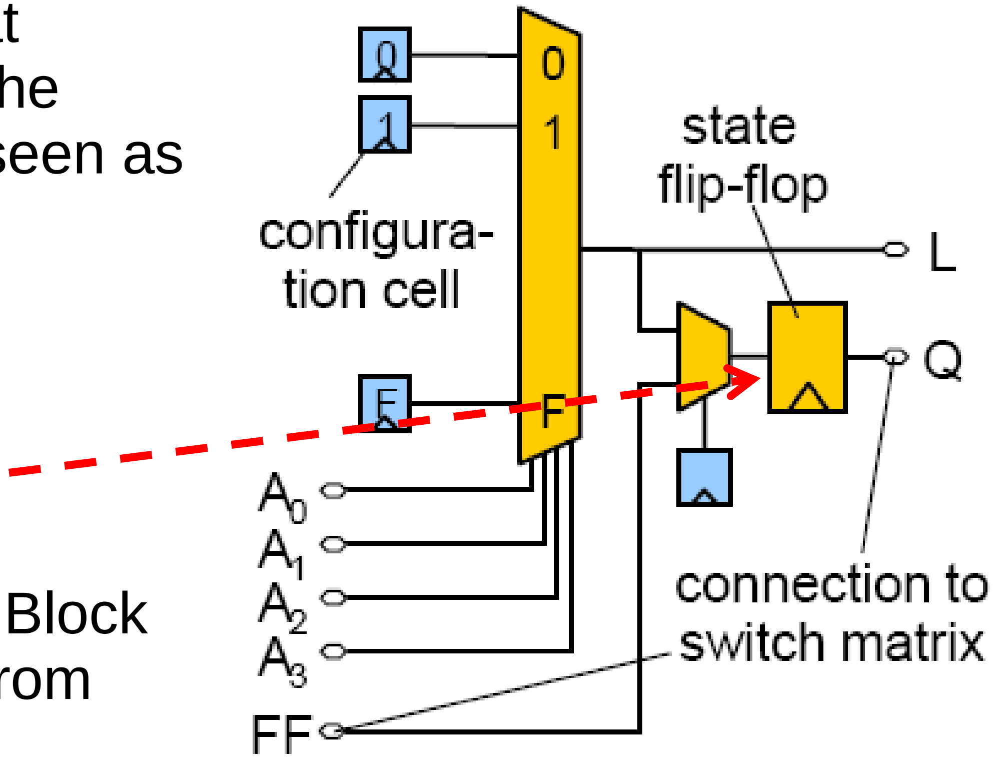


FPGA Components: Logic

A LUT is basically a multiplexer that evaluates the truth table stored in the configuration SRAM cells (can be seen as a one bit wide ROM).

How to handle sequential logic?
Add a flip-flop to the output of LUT (Clocked Storage element).

This is called a Configurable Logic Block (CLB): circuit can now use output from LUT or from FF.

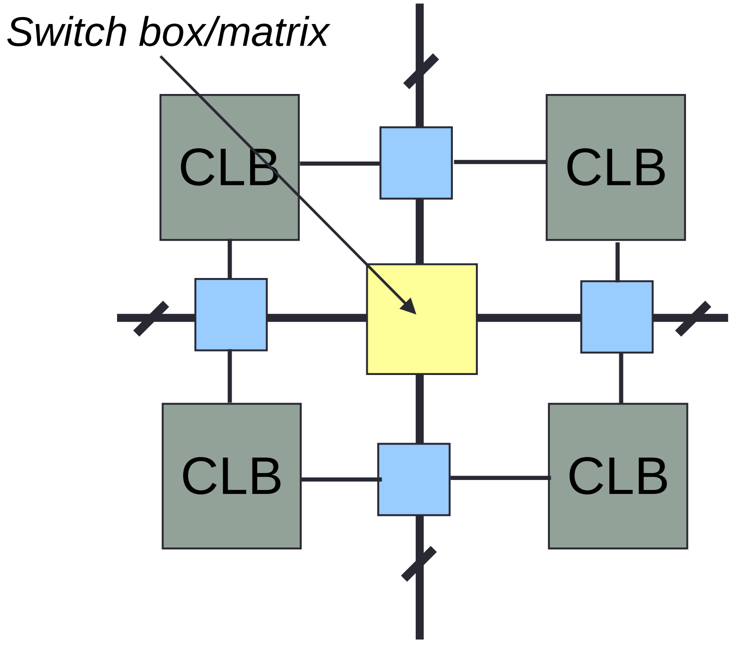


FPGA Components: wires

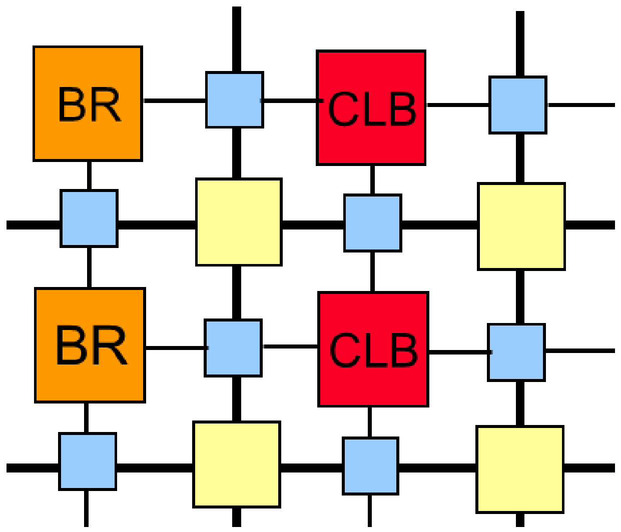
Connection boxes allow CLBs to connect to routing wires but that only allows to move signals along a single wire; to connect wires together Switch boxes (switch matrices) are used: these connect horizontal and vertical routing channels. The flexibility defines how many wires a single wire can connect into the box.

ROUTABILITY is a measure of the number of circuits that can be routed

HIGHER FLEXIBILITY
=
BETTER ROUTABILITY



FPGA Components: memory



The FPGA fabric includes embedded memory elements that can be used as random-access memory (RAM), read-only memory (ROM), or shift registers. These elements are block RAMs (BRAMs), LUTs, and shift registers.

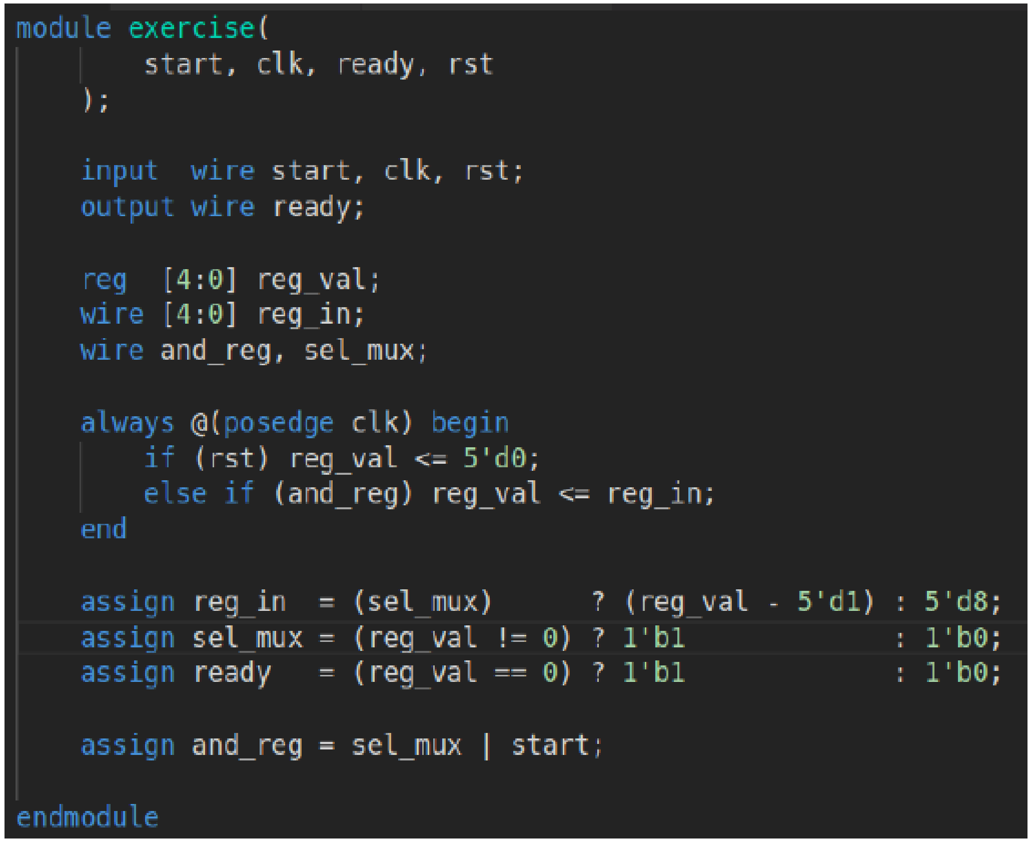
Using LUTs as SRAM, this is called
DISTRIBUTE RAM

Included dedicated RAM components in the FPGA fabric are called
BLOCKs RAM

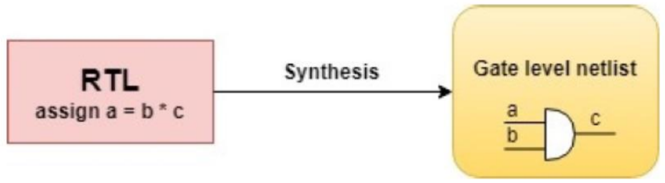
Designing with FPGA

- FPGAs are configured using a HW design flow
  - Describe the desired behavior in a **Hardware Description Language (HDL)**
  - Use the FPGA design automation tools to turn the HDL description into a configuration bitstream
- After configuration, the FPGA operates like dedicated hardware
- HW design expertise needed, low abstraction level, much slower than SW design on processors!

HDL Example (System Verilog)



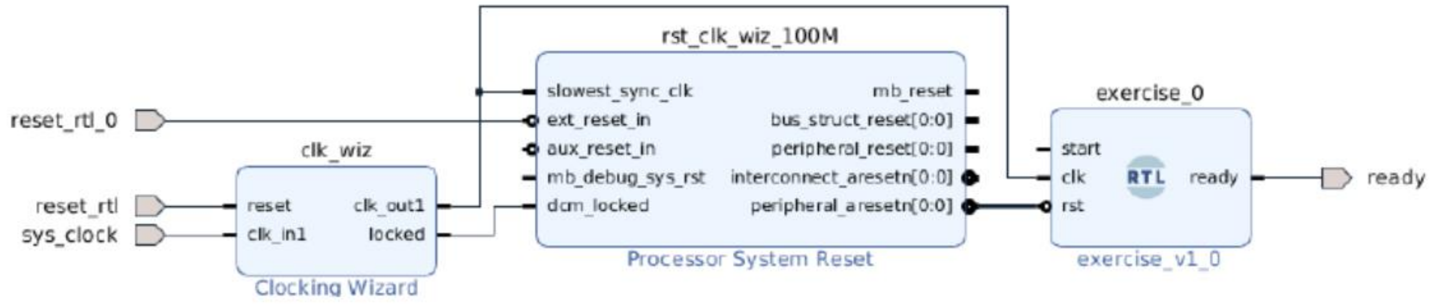
```
module exercise(
        start, clk, ready, rst
    );

    input  wire start, clk, rst;
    output wire ready;

    reg  [4:0] reg_val;
    wire [4:0] reg_in;
    wire and_reg, sel_mux;

    always @(posedge clk) begin
        if (rst) reg_val <= 5'd0;
        else if (and_reg) reg_val <= reg_in;
    end

    assign reg_in  = (sel_mux)       ? (reg_val - 5'd1) : 5'd8;
    assign sel_mux = (reg_val != 0) ? 1'b1             : 1'b0;
    assign ready   = (reg_val == 0) ? 1'b1             : 1'b0;

    assign and_reg = sel_mux | start;

endmodule
```



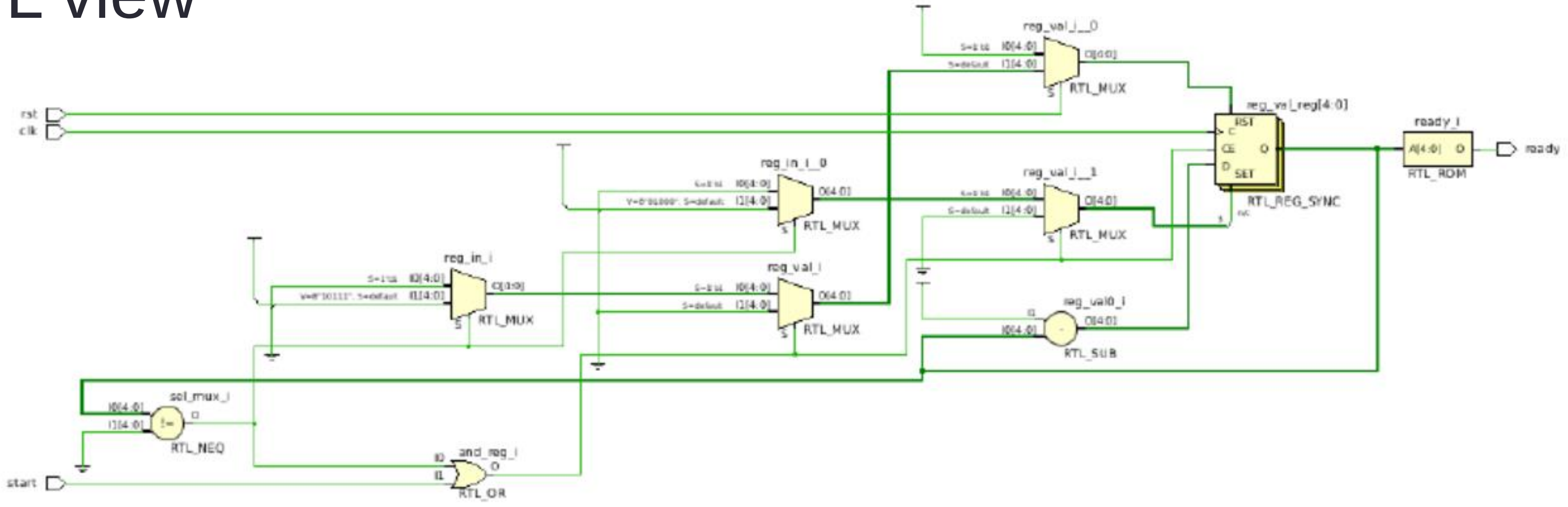
- Level of abstraction is **RTL (register transfer level)**, where building blocks are adders, multipliers, flip-flops, etc.
- Need to handle explicitly sequential logic signals
  - Registers
  - Flip-flops
  - Control signals (e.g., reset)
  - Clock
- **Synthesis** is the process from which we obtain a **gate-level netlist** from our RTL description of the hardware

HDL Example (System Verilog)

- HDL view

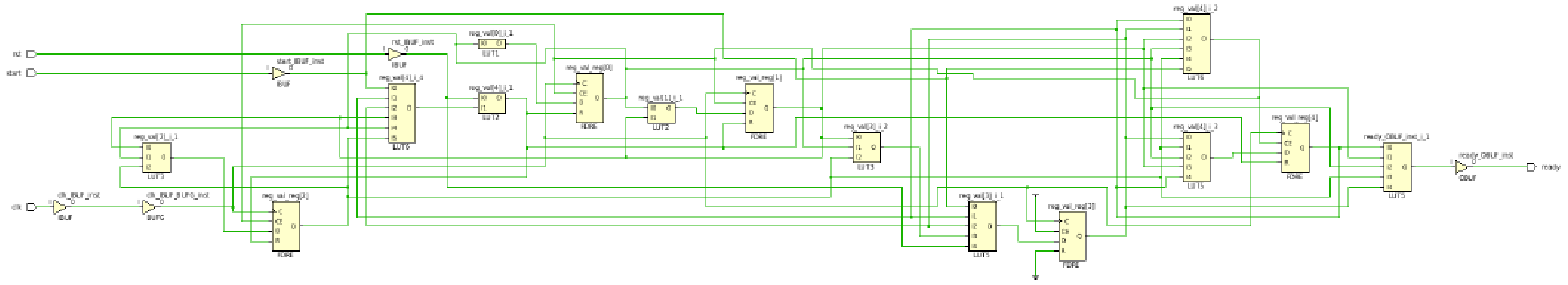


- RTL view

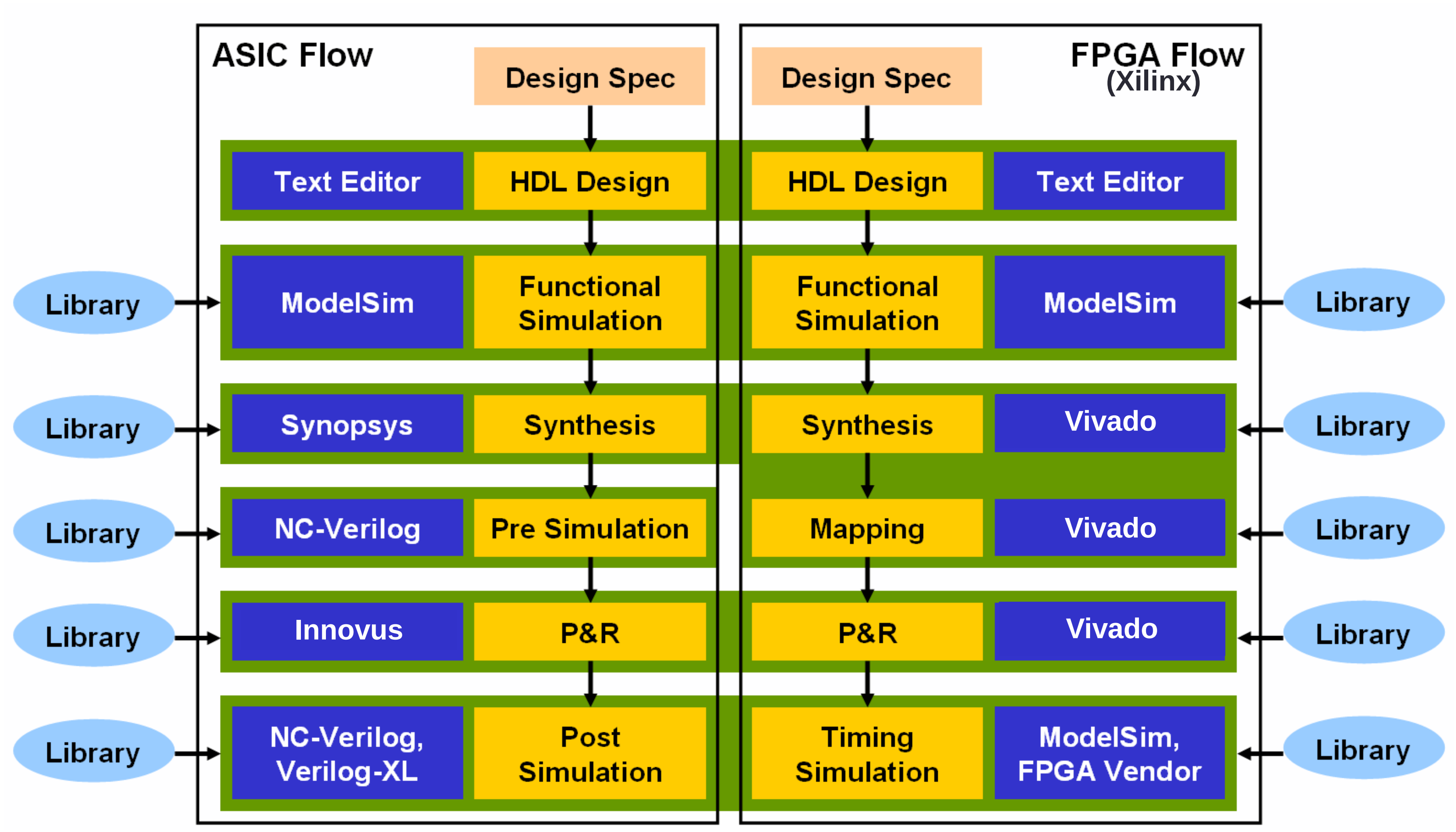


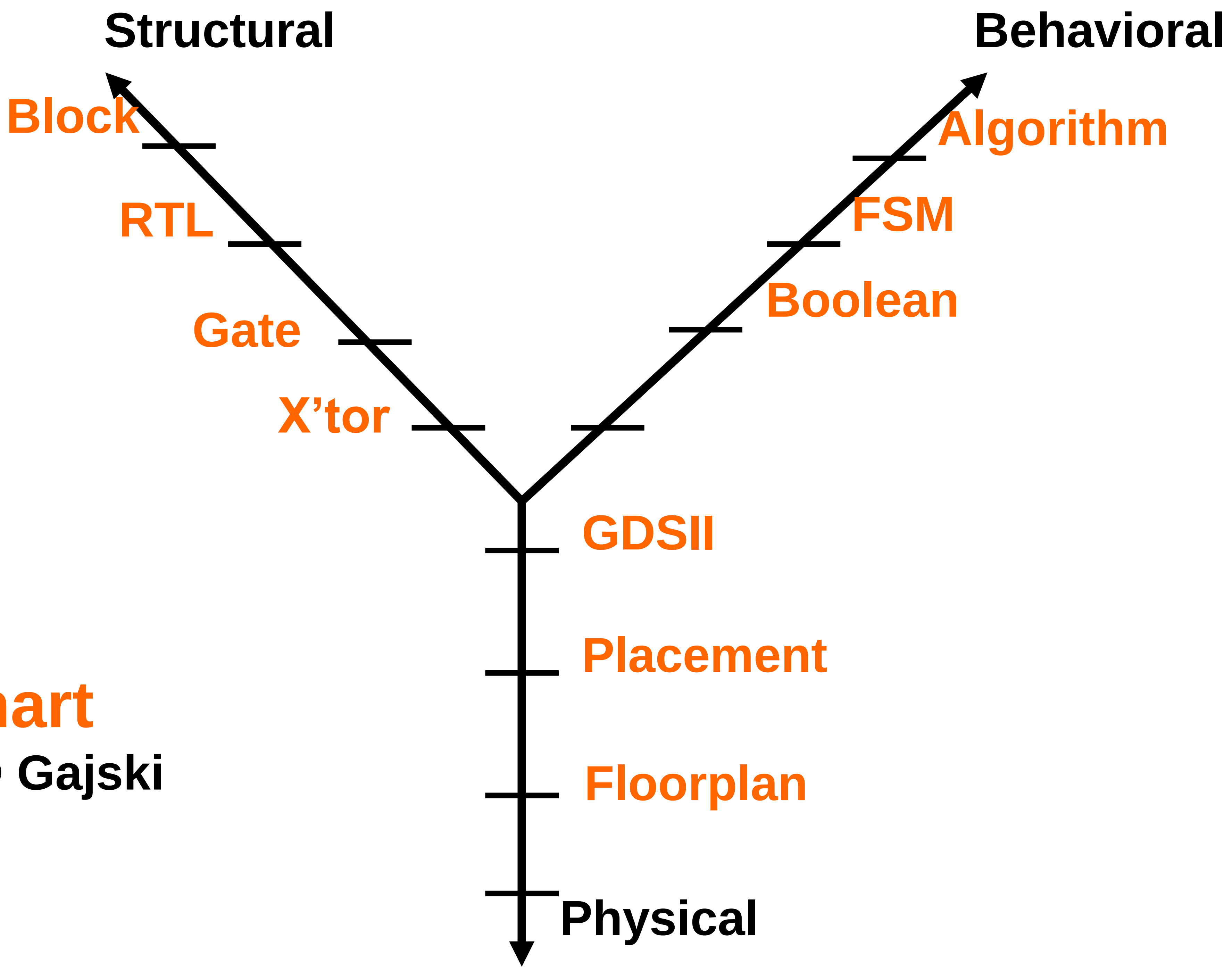
HDL Example (System Verilog)

- Netlist



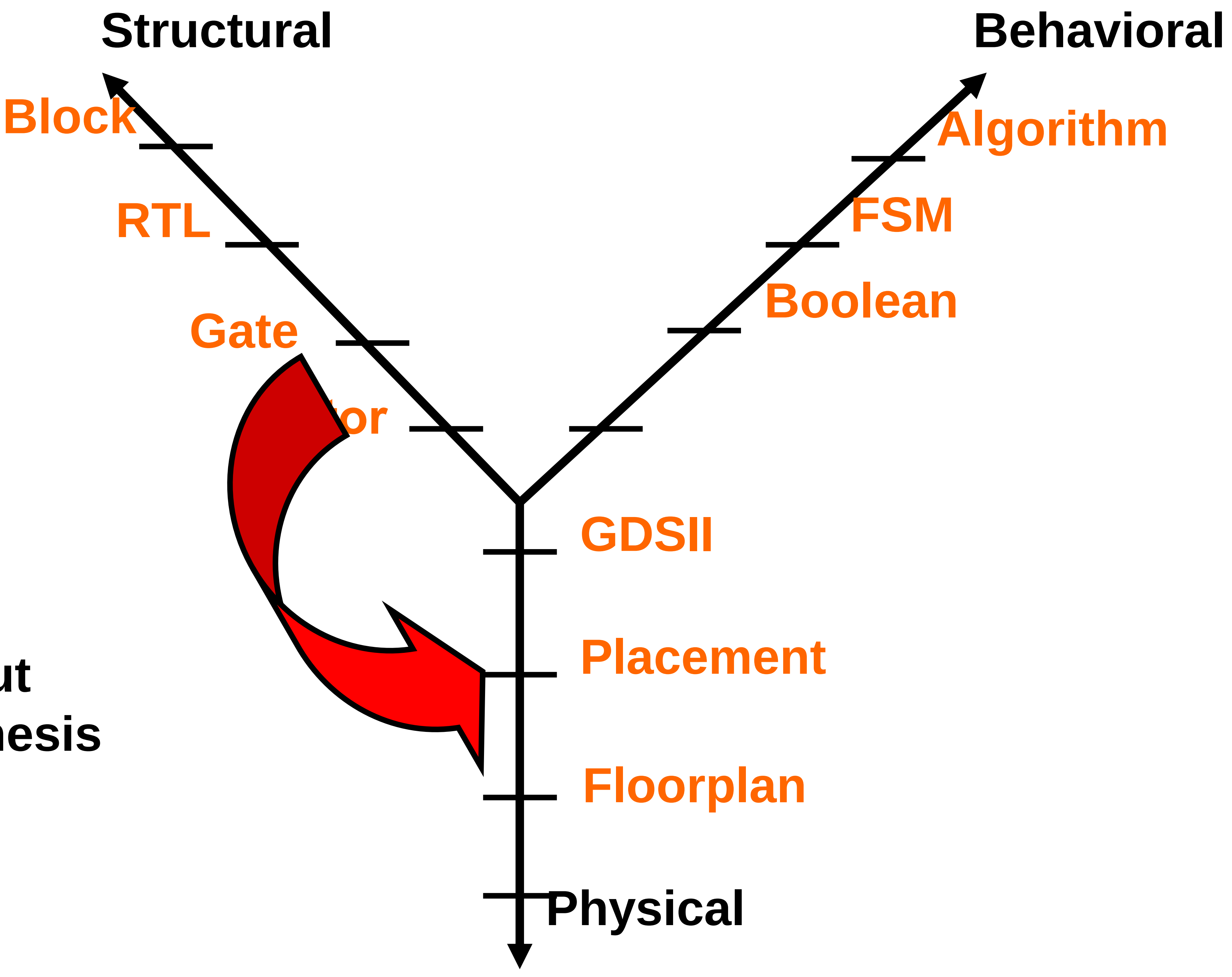
HW Design flow



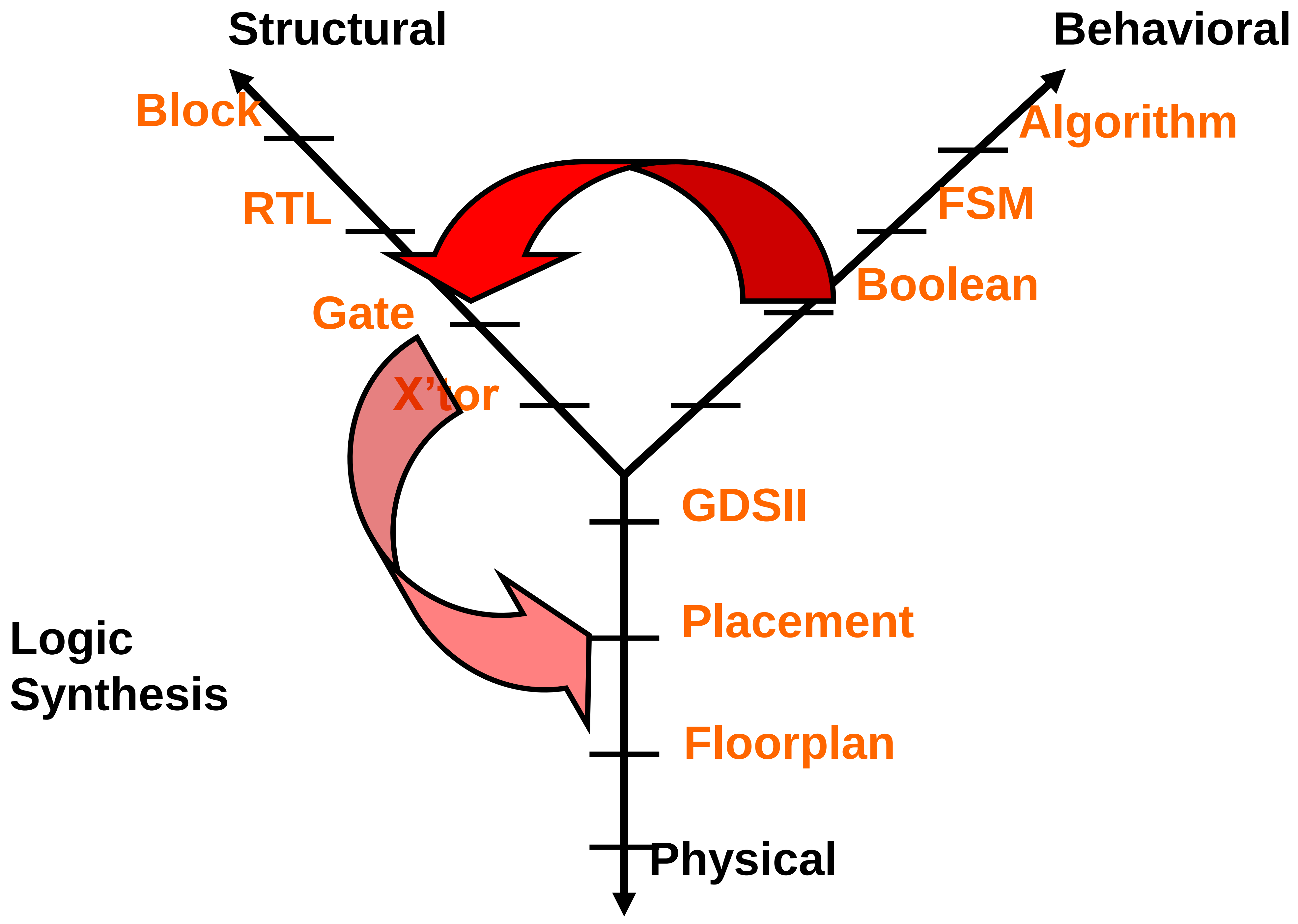


Y-Chart

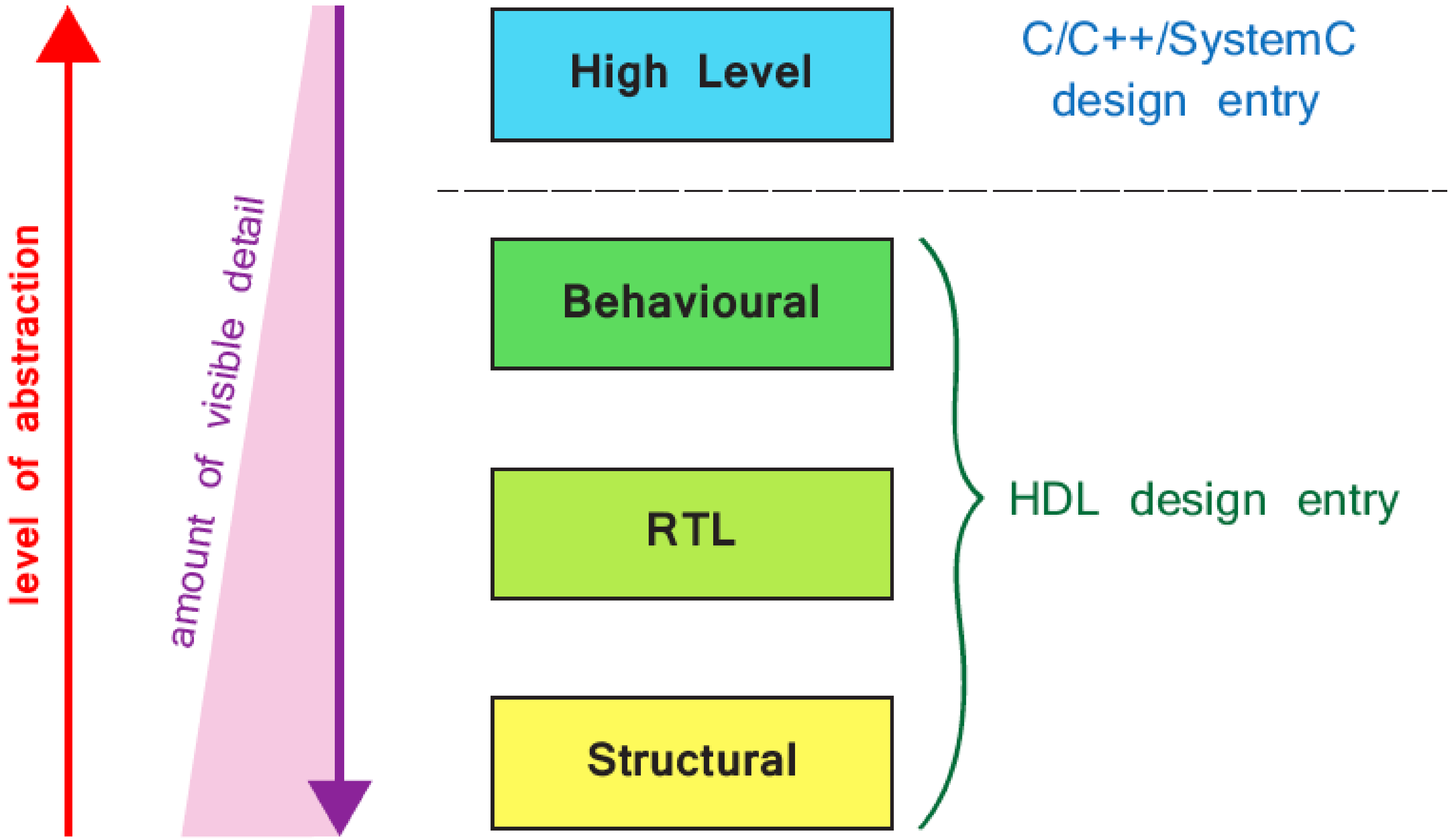
Dan D Gajski



Layout
Synthesis


Structural
Behavioral
Block
RTL
Gate
X'tor
Algorithm
FSM
Boolean
GDSII
Placement
Floorplan
Physical
Logic Synthesis

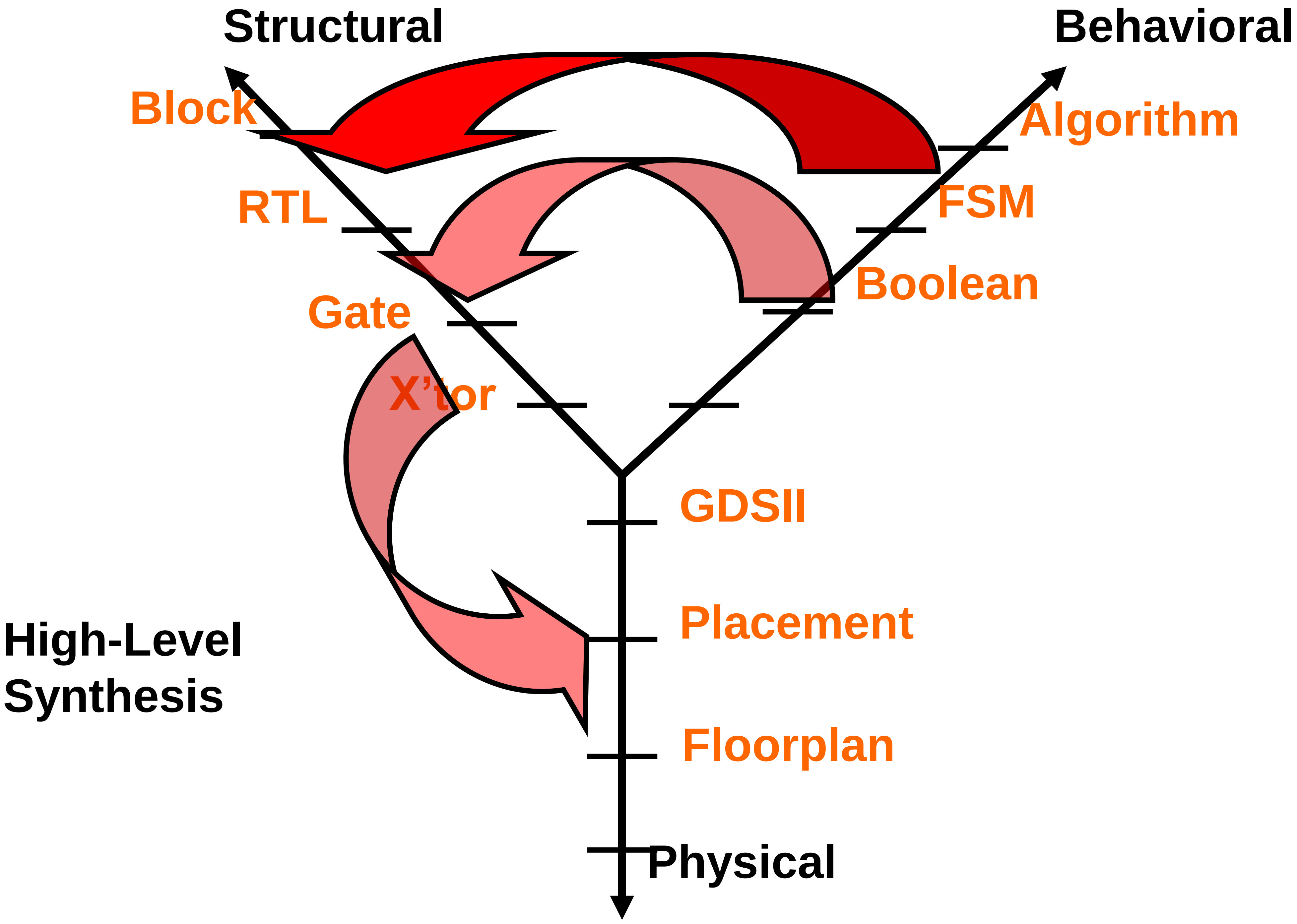
Level of abstractions in FPGA design



Source: The Zynq Book

Level of abstractions in FPGA design

Synthesis step	Level	Behavior	Structure
System ↓	Specification	System specification	
High-level ↓	System	Algorithms	CPU's, MEM's BUS's
Logic ↓	Register (RTL)	Register transfers	REG's, ALU's, MUX's
Physical ↓	Logic	Boolean expressions	Gates, flip-flops
	Circuit	Transfer functions	Transistors


Structural
Behavioral
Block
Algorithm
RTL
FSM
Gate
Boolean
X'tor
GDSII
Placement
High-Level
Synthesis
Floorplan
Physical

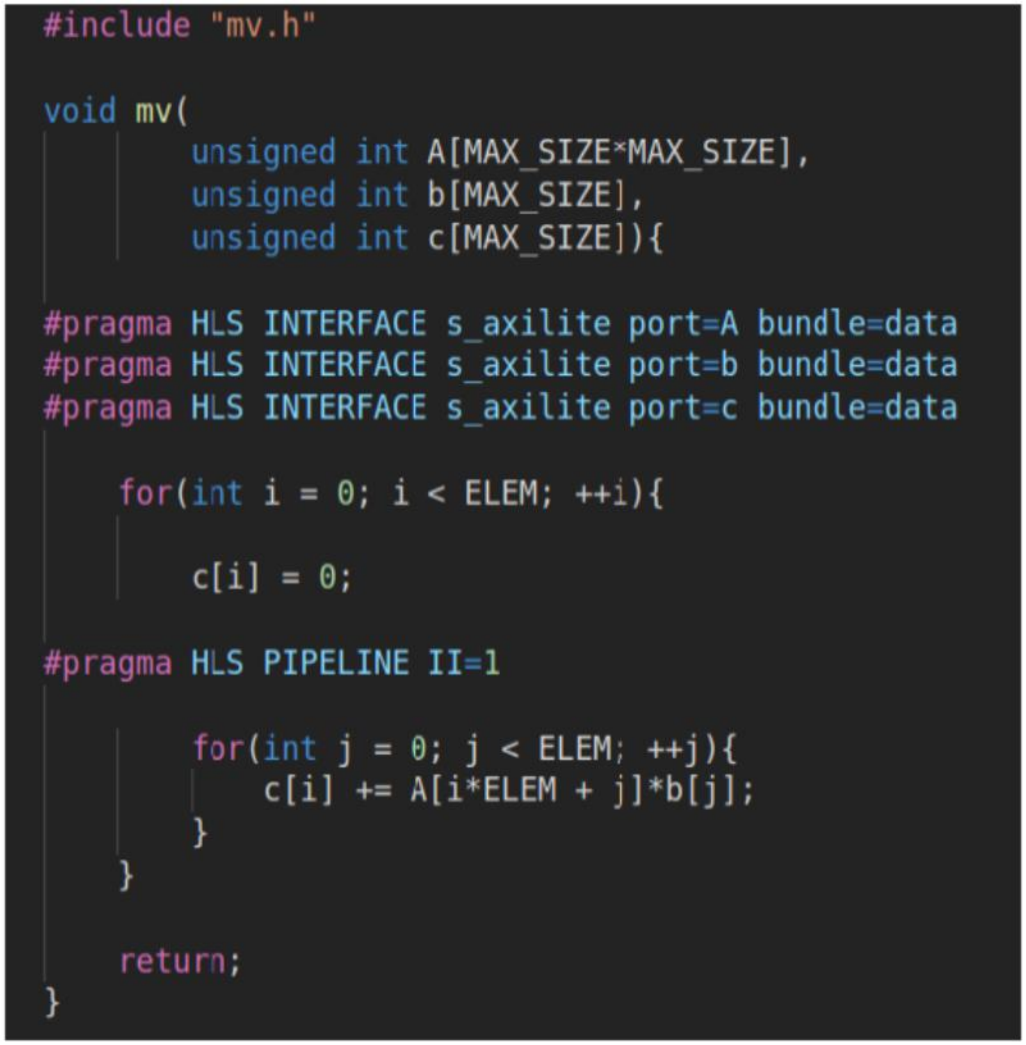
High-Level Synthesis (HLS)

- HLS is an **automated design process** that transforms a high-level functional specification to a [optimized] register-transfer level (RTL) description suitable for hardware implementation

- HLS tools are widely used for complex ASIC and **FPGA design**

- Main benefits
  - **Productivity:** lower design complexity and faster simulation speed
  - **Portability:** single source → multiple implementations
  - **Permutability:** rapid design space exploration → higher quality of result (QoR)

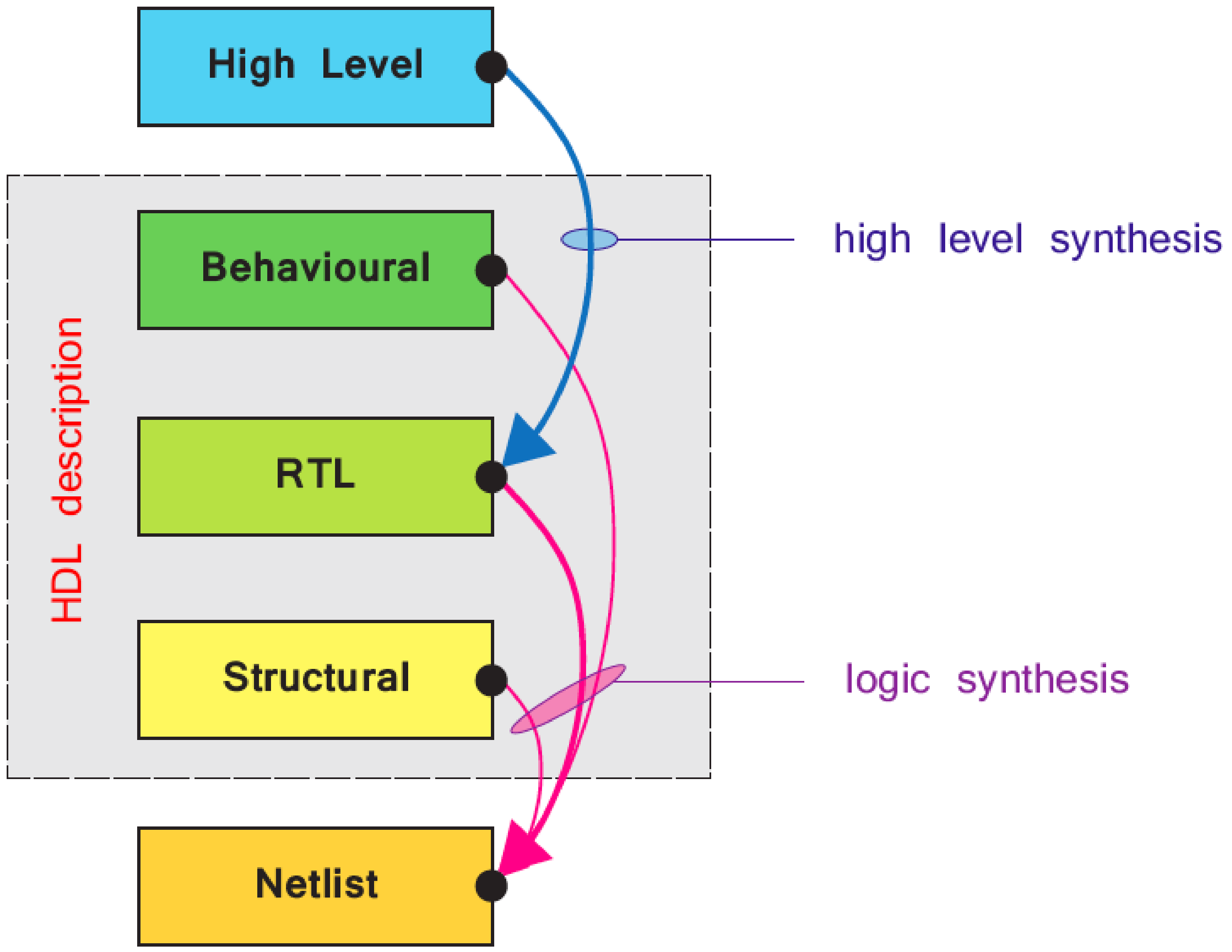
HLS Example (C/C++)



```
#include "mv.h"

void mv(
        unsigned int A[MAX_SIZE*MAX_SIZE],
        unsigned int b[MAX_SIZE],
        unsigned int c[MAX_SIZE]){

#pragma HLS INTERFACE s_axilite port=A bundle=data
#pragma HLS INTERFACE s_axilite port=b bundle=data
#pragma HLS INTERFACE s_axilite port=c bundle=data

    for(int i = 0; i < ELEM; ++i){

        c[i] = 0;

#pragma HLS PIPELINE II=1

        for(int j = 0; j < ELEM; ++j){
            c[i] += A[i*ELEM + j]*b[j];
        }
    }

    return;
}
```

- Behavioral description of the HW via a procedural, high-level language.
- Not all C/C++ constructs can be used, and a few “non-standard things” to get used to
  - Use of compiler directives (#pragma) to steer design decisions
  - Custom data types (ap_uint, ap_fixed, …)
  - Keywords (volatile, static, …)

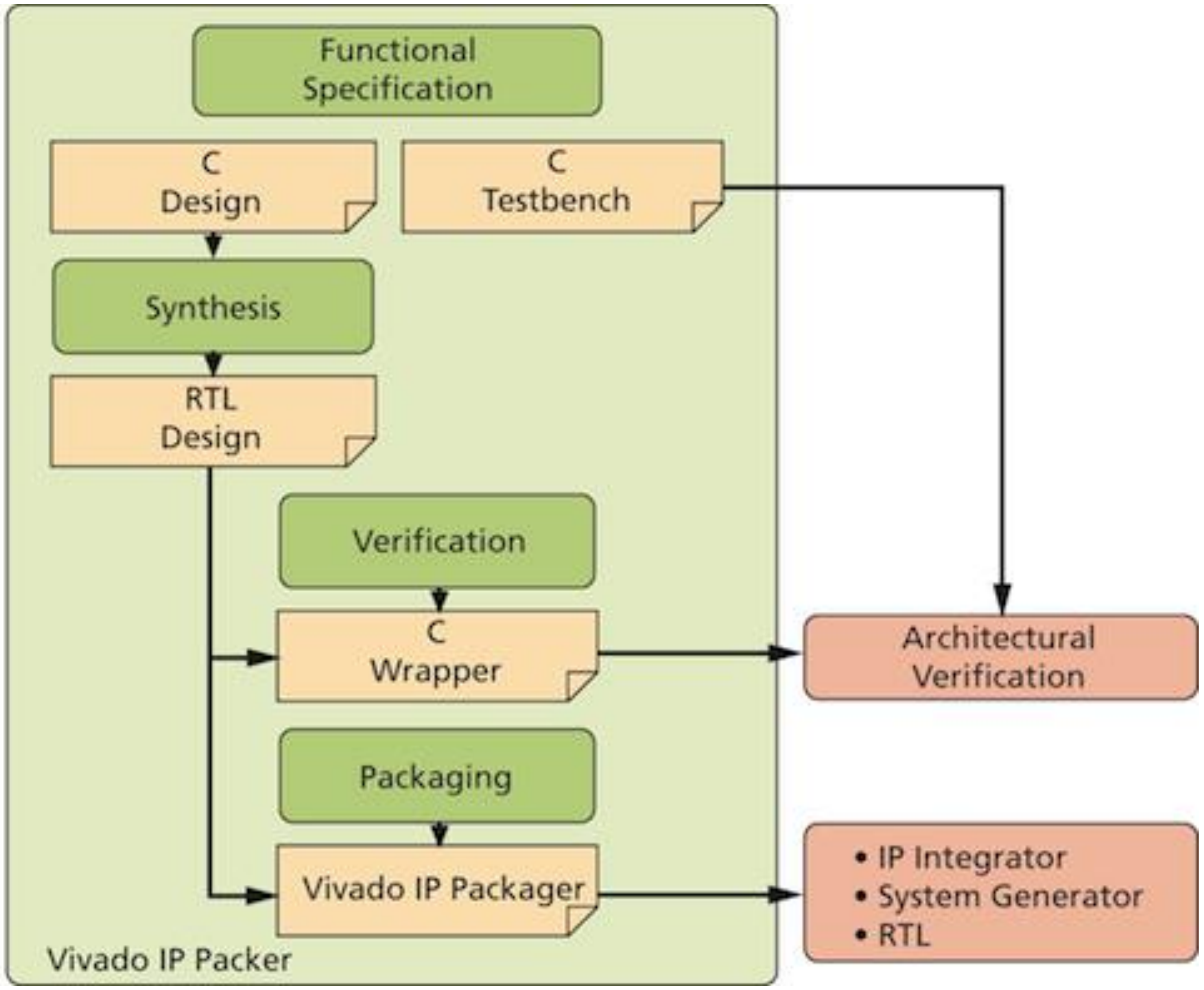
High-level VS logic synthesis



Source: The Zynq Book

Xilinx Vivado HLS

- Starts at C
  - C
  - C++
  - SystemC
- Produces RTL
  - Verilog
  - VHDL
  - SystemC
- Automates Flow
  - Verification
  - Implementation



Permutability – Design Space Exploration

One body of code:
Many hardware outcomes

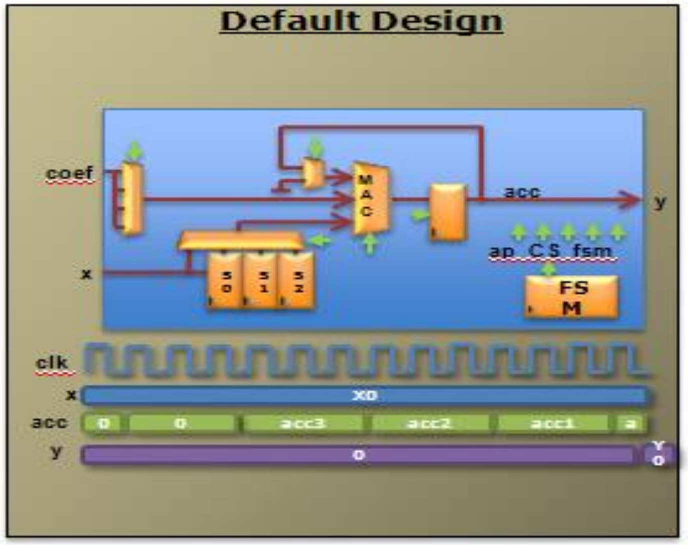
```
…
loop: for (i=3;i>=0;i--) {
  if (i==0) {
    acc+=x*c[0];
    shift_reg[0]=x;
  } else {
     shift_reg[i]=shift_reg[i-1];
     acc+=shift_reg[i]*c[i];
  }
}
….
```

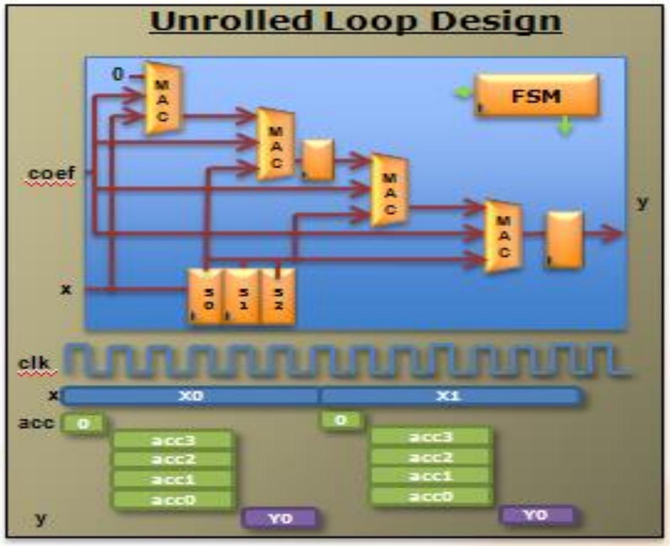
Before we get into details, let's look under the hood ….

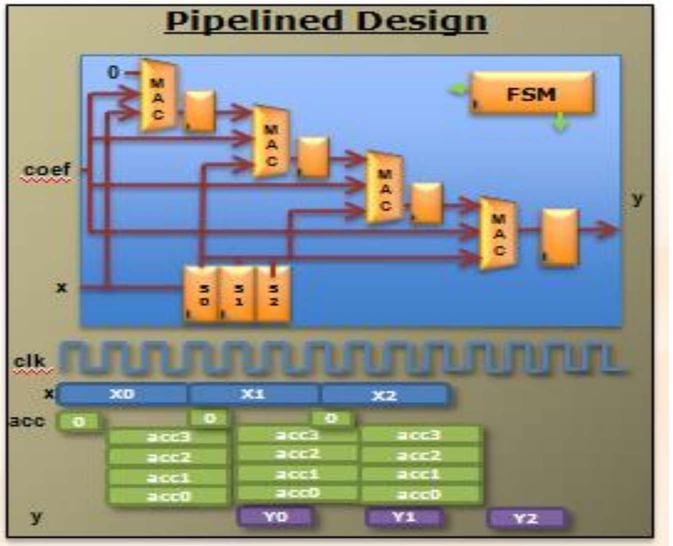
The same hardware is used for each iteration of the loop:
•Small area
•Long latency
•Low throughput

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

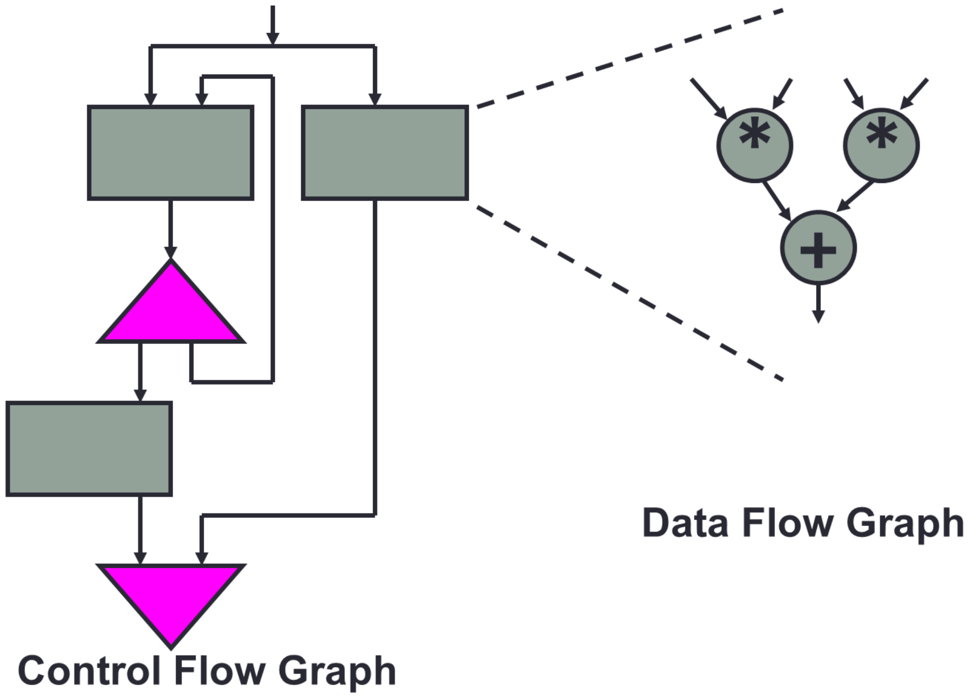




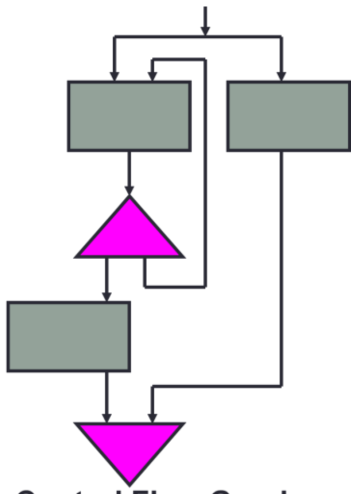


How does HLS work?

- 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?



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;
}
```

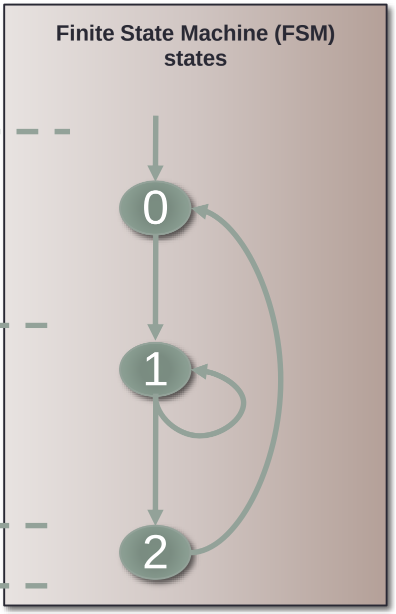
Function Start

For-Loop Start

For-Loop End

Function End

Control Behavior

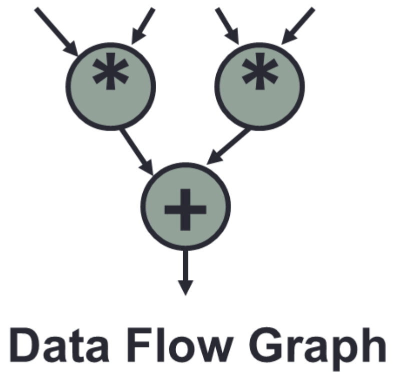


From any C code example ..

The loops in the C code correlated to states of behavior

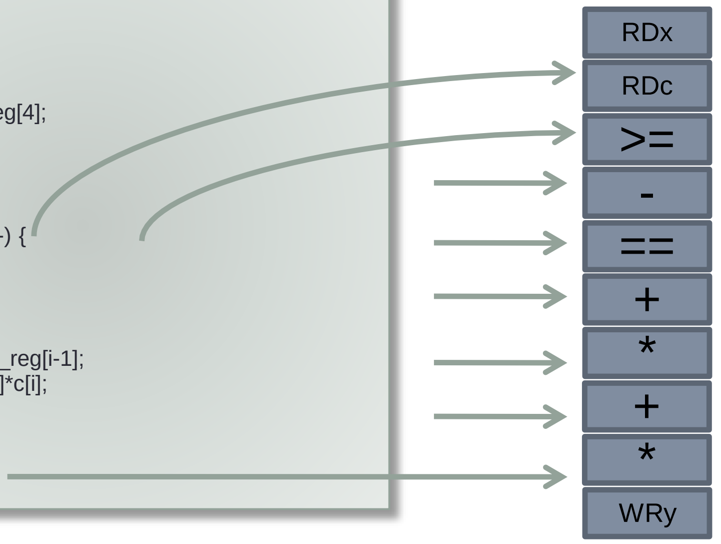
This behavior is extracted into a hardware state machine

Datapath extraction



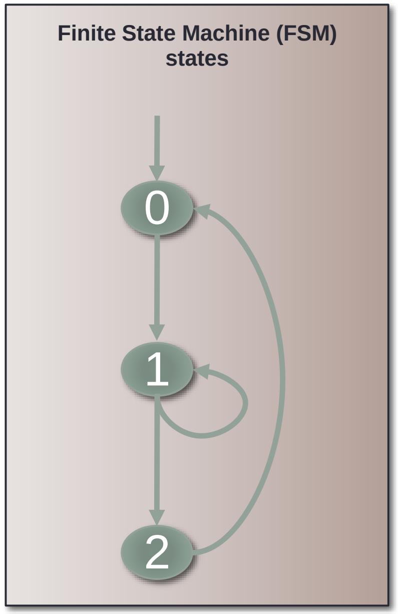
Code

```
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;
}
```

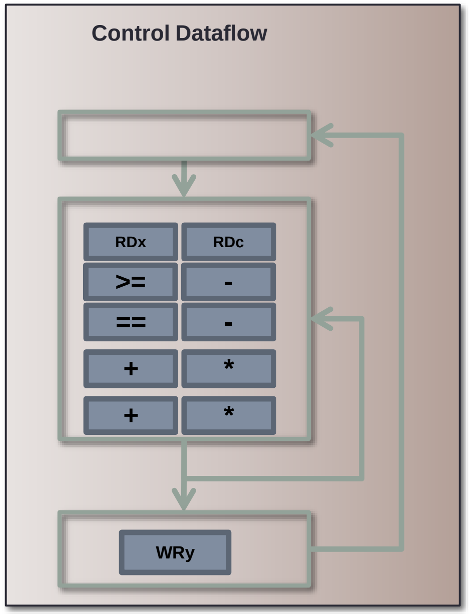


Operations

Control Behavior



Control & Datapath Behavior



From any C code example ..

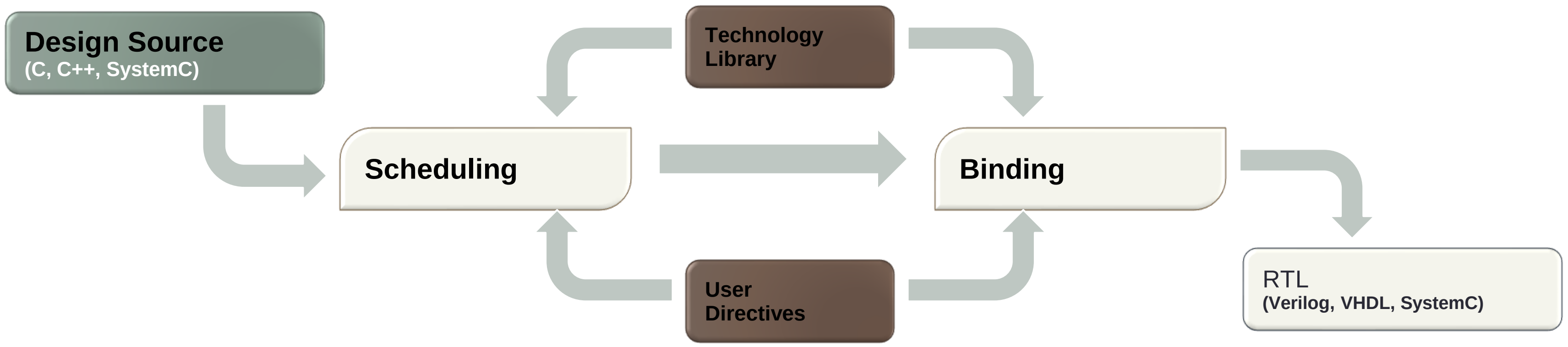
Operations are extracted…

The control is known

A unified control dataflow behavior is created.

Scheduling and Binding

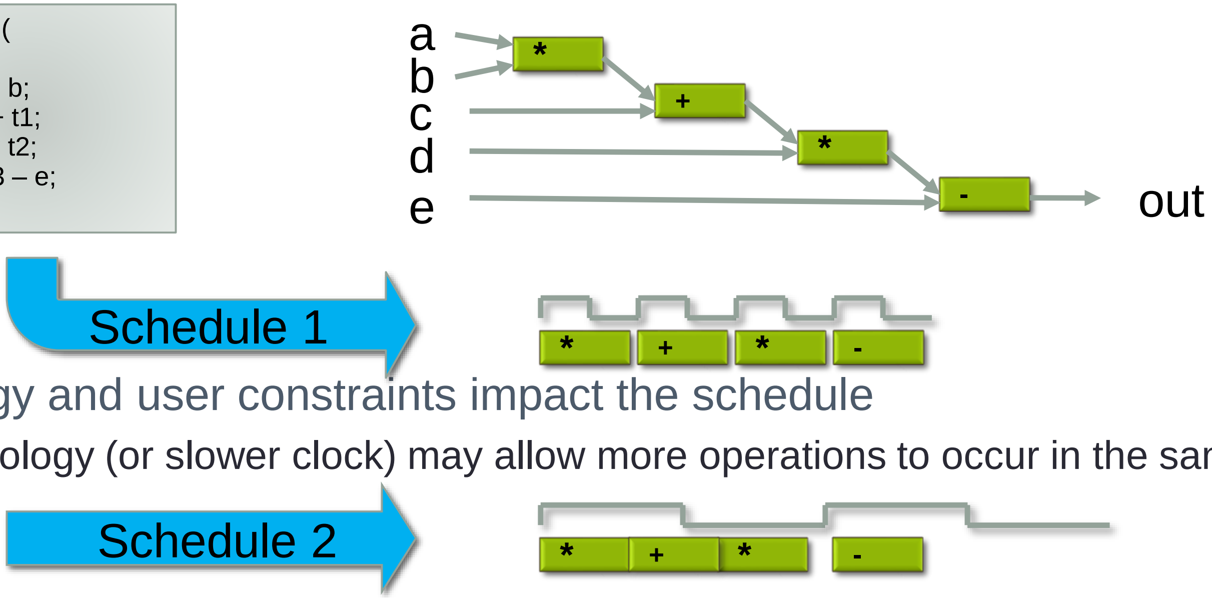
- Scheduling & Binding
  - Scheduling and Binding are at the heart of HLS
- Scheduling determines in which clock cycle an operation will occur
  - Takes into account the control, dataflow and user directives
  - The allocation of resources can be constrained
- Binding determines which library cell is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



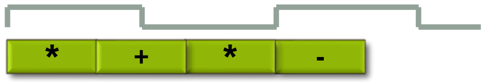
- The technology and user constraints impact the schedule
  - A faster technology (or slower clock) may allow more operations to occur in the same clock cycle

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

Binding

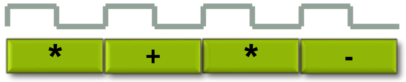
- Binding is where operations are mapped to cores from the hardware library
  - Operators map to cores

- 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 *share* 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

Mapping of C/C++ constructs to RTL

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

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

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

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

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

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

Functions & RTL hierarchy

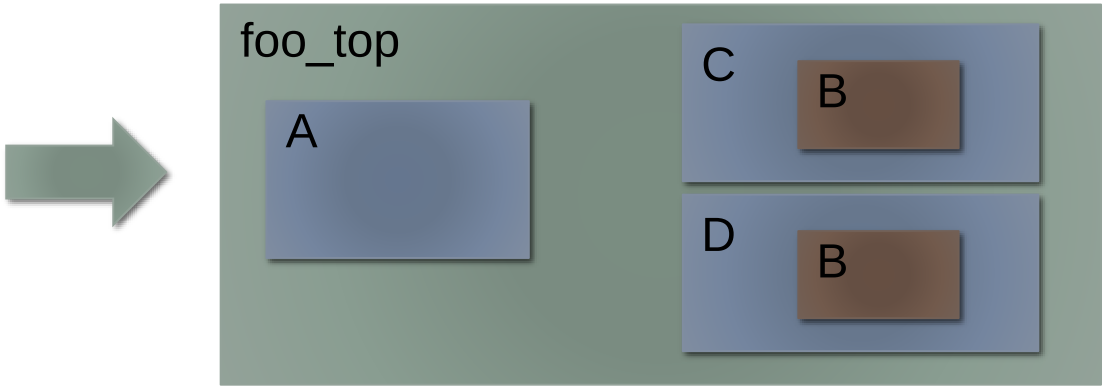
- Each function is translated into an RTL block
  - Verilog module, VHDL entity

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

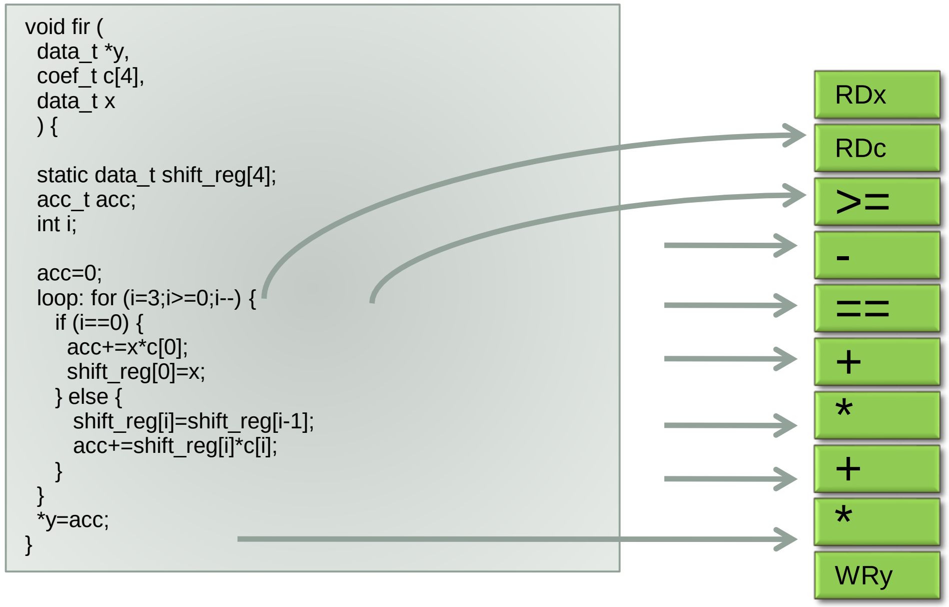
- By default, each function is implemented using a common instance
- Functions may be inlined to dissolve their hierarchy
  - Small functions may be automatically inlined

Types = Operator bit sizes

Code

Operations

Types



```
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;
}
```

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).

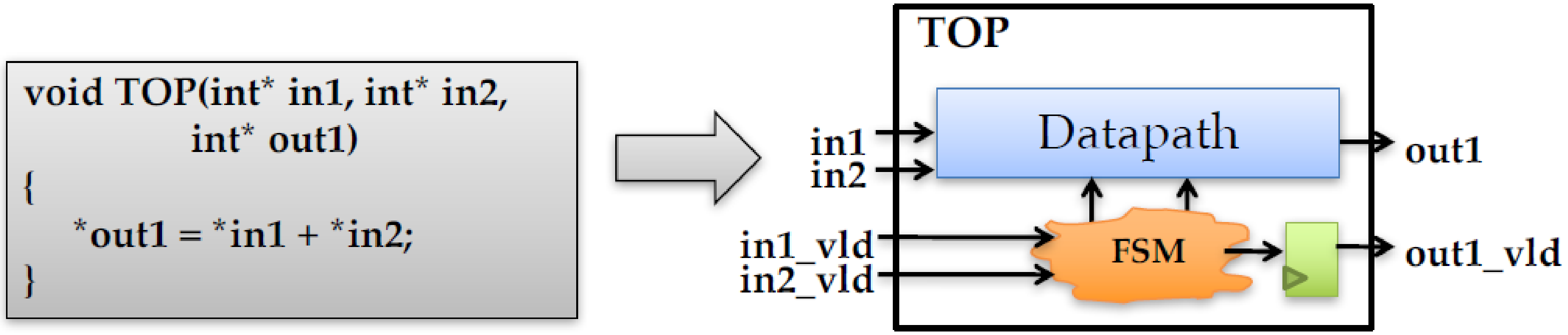
From any C code example ...

Operations are extracted…

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

Function arguments

- Function arguments become ports on the RTL blocks
  - Additional control ports are added to the design



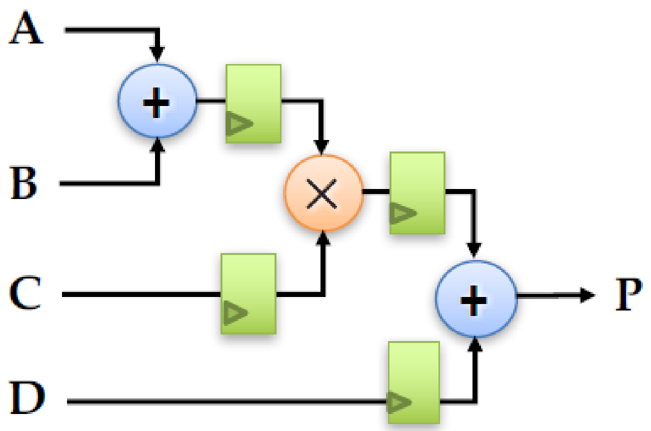
```
void TOP(int* in1, int* in2,
              int* out1)
{
    *out1 = *in1 + *in2;
}
```

- Input/output (I/O) protocols
  - They allow RTL blocks to synchronize data exchange

Expressions

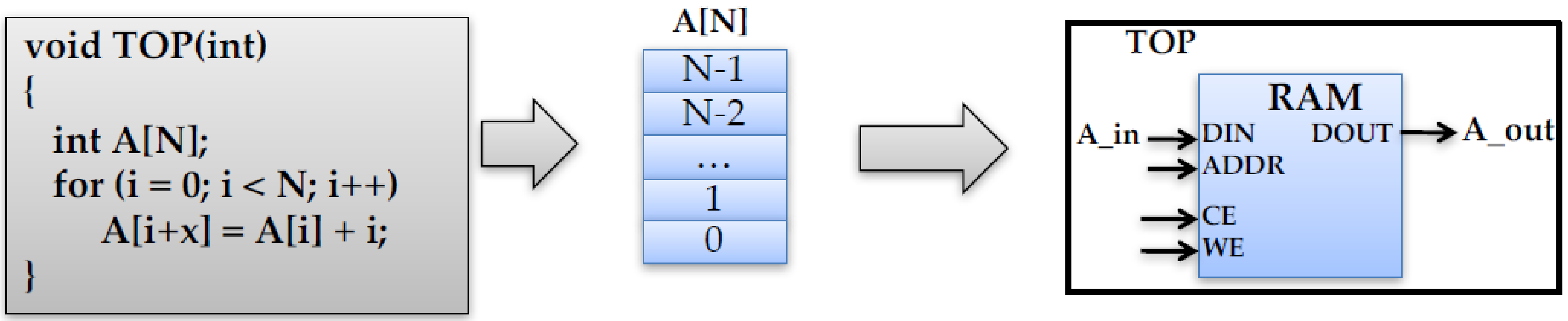
- HLS generates datapath circuits mostly from expressions
  - Timing constraints influence the use of registers

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

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



Arrays

- By default, an array in C code is typically implemented by a memory block in the RTL
  - Read & write array → RAM; Constant array → ROM

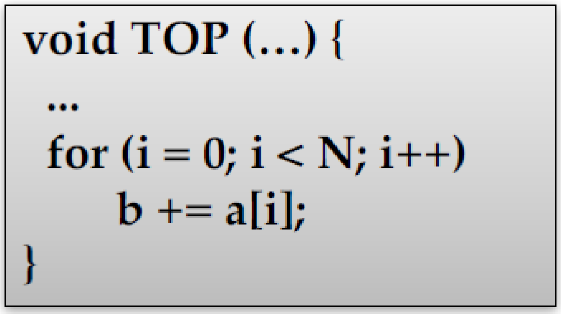


```
void TOP(int)
{
  int A[N];
  for (i = 0; i < N; i++)
      A[i+x] = A[i] + i;
}
```

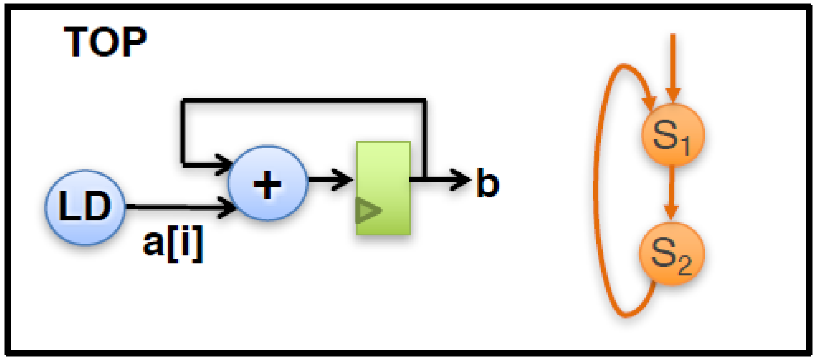
- An array can be partitioned and map to multiple RAMs
- Multiples arrays can be merged and map to one RAM
- An array can be partitioned into individual elements and mapped to registers

Loops

- By default, loop iterations are executed in order
  - 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

- Loop unrolling to expose **higher parallelism** and achieve shorter latency

- **Pros**
    - Decrease loop overhead
    - <u>Increase **parallelism** for scheduling</u>
    - Facilitate constant propagation and array-to-scalar promotion
- **Cons**
    - Increase operation count, which may negatively impact *area*, *power*, and *timing*

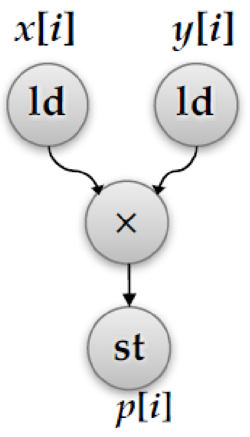
```
for (int i = 0; i < N; i++)
  A[i] = C[i] + D[i];
```

↓

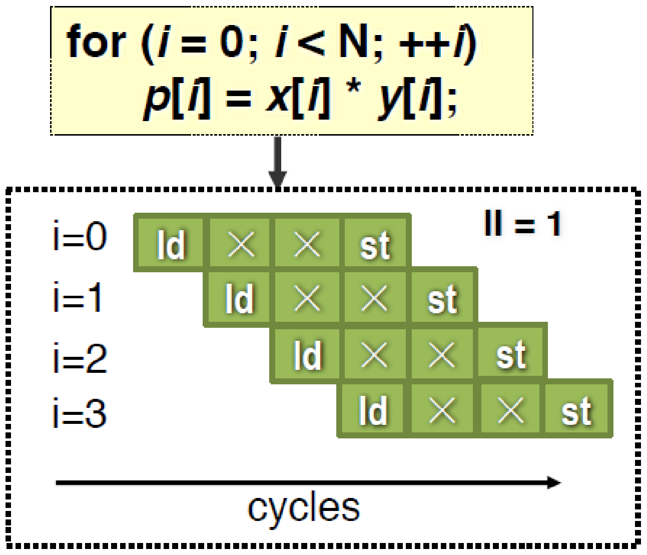
```
A[0] = C[0] + D[0];
A[1] = C[1] + D[1];
A[2] = C[2] + D[2];
     .....
```

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)** expressed in number of cycles



ld – Load
st – Store



Example: FIR filter

$$y[n] = \sum_{i=0}^{N} b_i x[n-i]$$

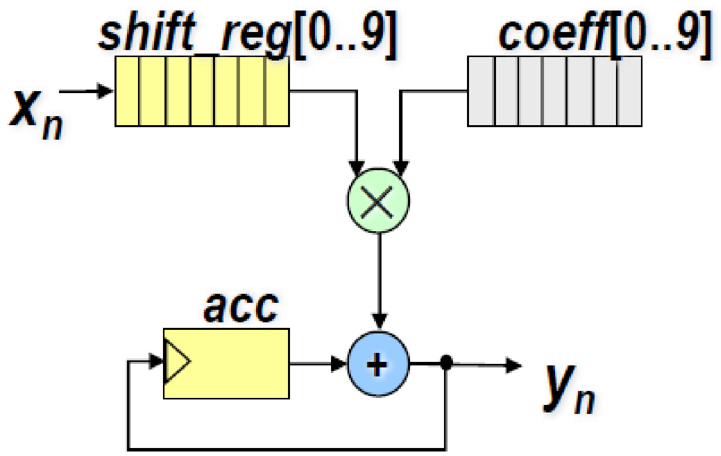
$x[n]$	input signal
$y[n]$	output signal
N	filter order
b_i	*i*th filter coefficient

```
// original, non-optimized version of FIR

#define SIZE 128
#define N 10

void fir(int input[SIZE], int output[SIZE]) {

  // FIR coefficients
  int coeff[N] = {13, -2, 9, 11, 26, 18, 95, -43, 6, 74};

  // exact translation from FIR formula above
  for (int n = 0; n < SIZE; n++) {
    int acc = 0;
    for (int i = 0; i < N; i++ ) {
      if (n - i >= 0)
        acc += coeff[i] * input[n - i];
    }
    output[n] = acc;
  }
}
```

HLS code for a FIR Filter

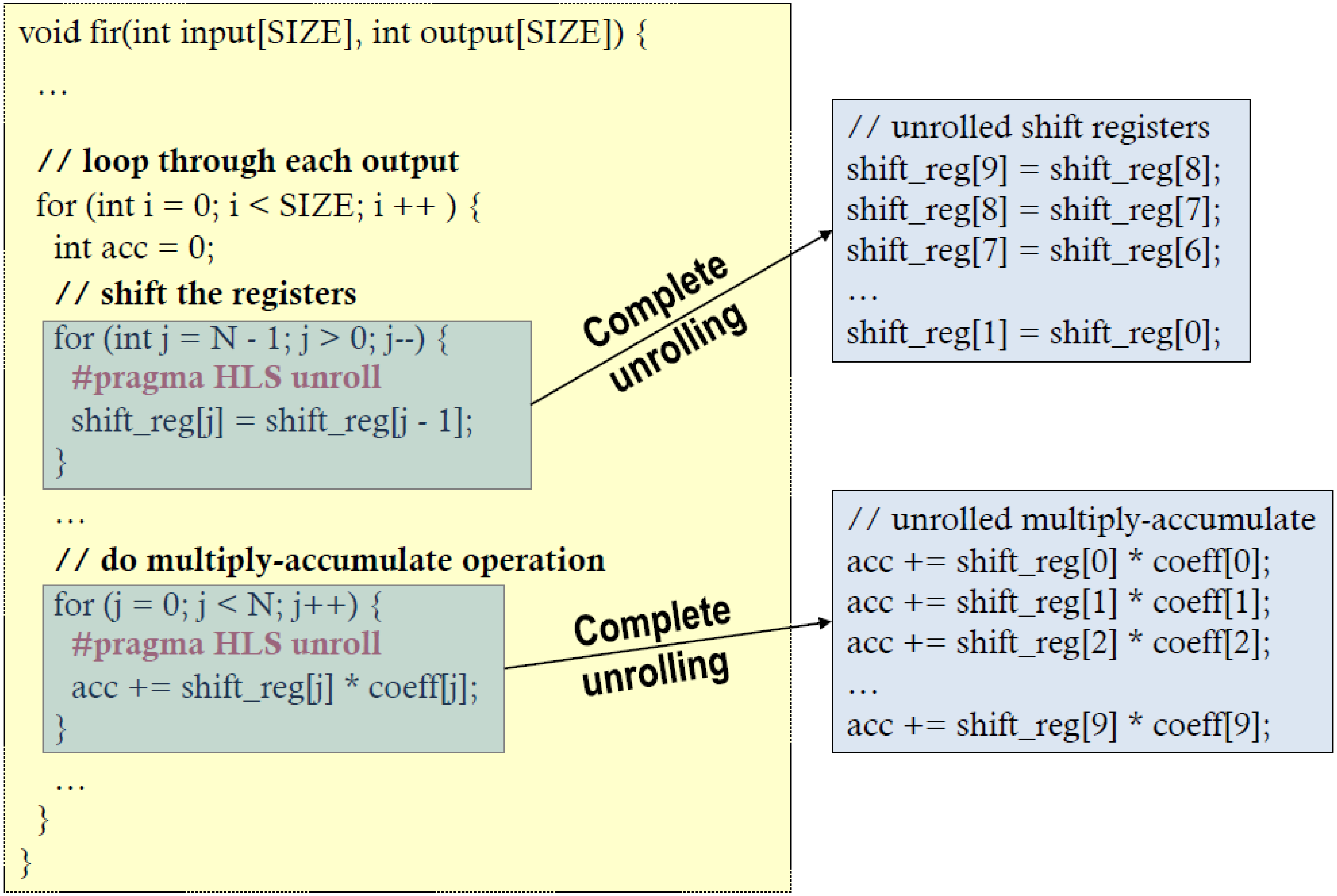
```
void fir(int input[SIZE], int output[SIZE]) {
  // FIR coefficients
  int coeff[N] = {13, -2, 9, 11, 26, 18, 95, -43, 6, 74};
  // Shift registers
  int shift_reg[N] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
  // loop through each output
  for (int i = 0; i < SIZE; i ++ ) {
    int acc = 0;
    // shift registers
    for (int j = N - 1; j > 0; j--) {
      shift_reg[j] = shift_reg[j - 1];
    }
    // put the new input value into the first register
    shift_reg[0] = input[i];
    // do multiply-accumulate operation
    for (j = 0; j < N; j++) {
      acc += shift_reg[j] * coeff[j];
    }
    output[i] = acc;
  }
}
```

$$y[n] = \sum_{i=0}^{N} b_i x[n-i]$$



- Further optimizations are possible
  - Loop unrolling
  - Pipelining

Loop unrolling

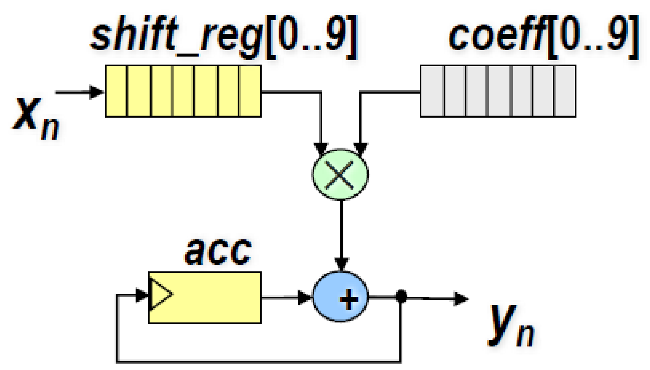


```
void fir(int input[SIZE], int output[SIZE]) {
  …

  // loop through each output
  for (int i = 0; i < SIZE; i ++ ) {
    int acc = 0;
    // shift the registers
    for (int j = N - 1; j > 0; j--) {
      #pragma HLS unroll
      shift_reg[j] = shift_reg[j - 1];
    }
    …
    // do multiply-accumulate operation
    for (j = 0; j < N; j++) {
      #pragma HLS unroll
      acc += shift_reg[j] * coeff[j];
    }
    …
  }
}
```

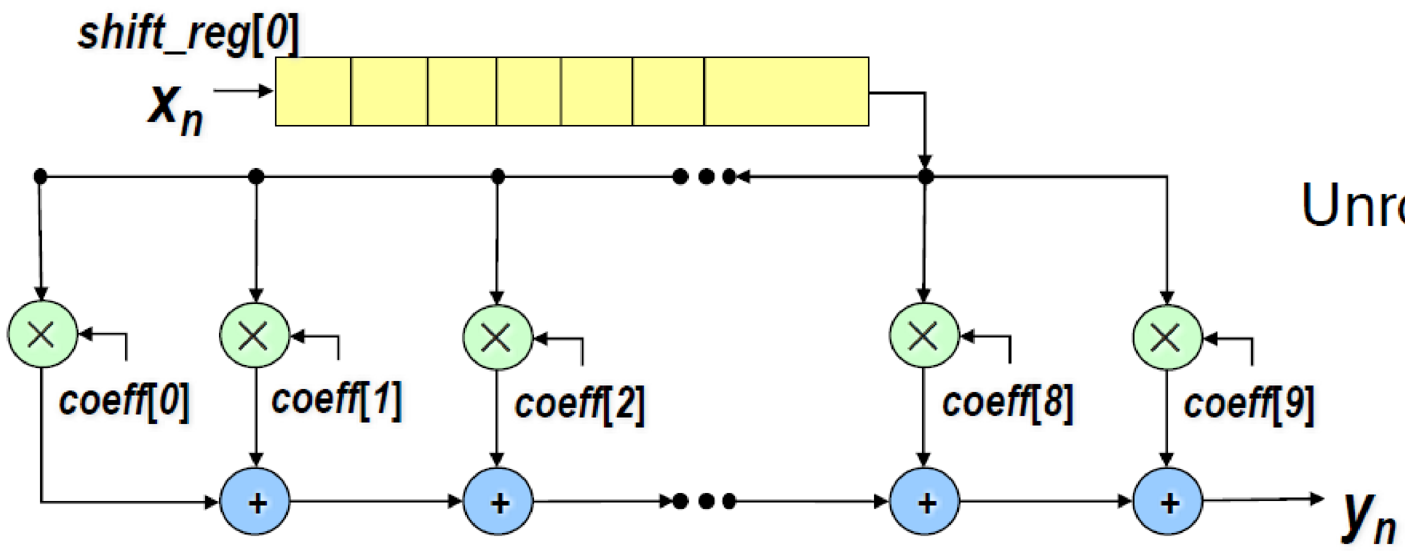
```
// unrolled shift registers
shift_reg[9] = shift_reg[8];
shift_reg[8] = shift_reg[7];
shift_reg[7] = shift_reg[6];
…
shift_reg[1] = shift_reg[0];
```

```
// unrolled multiply-accumulate
acc += shift_reg[0] * coeff[0];
acc += shift_reg[1] * coeff[1];
acc += shift_reg[2] * coeff[2];
…
acc += shift_reg[9] * coeff[9];
```

Architecture after unrolling



Default



Unrolled

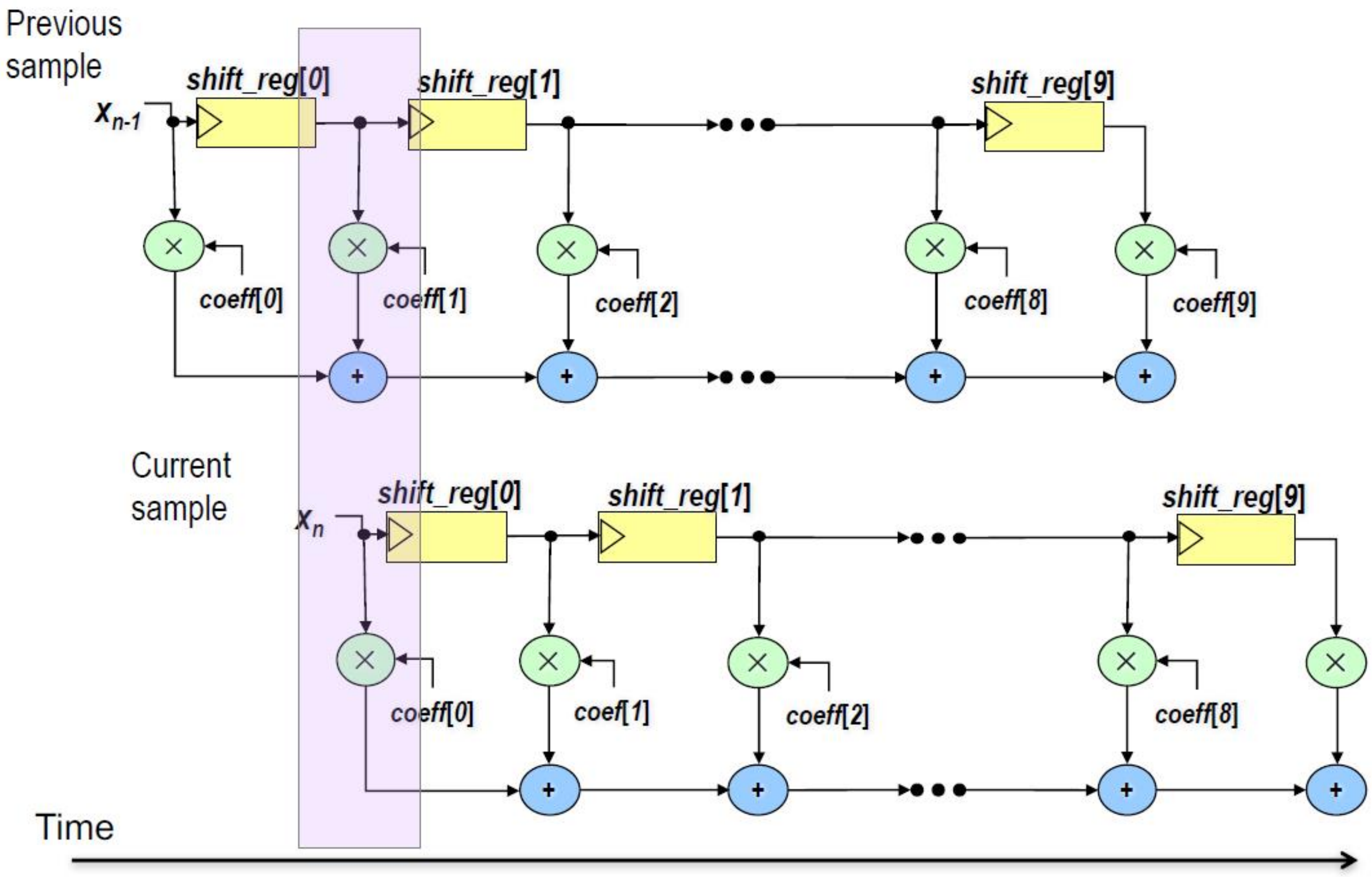
Pipelining

```
void fir(int input[SIZE], int output[SIZE]) {
  …

  // loop through each output
  for (int i = 0; i < SIZE; i ++ ) {
    #pragma HLS pipeline II=1
    int acc = 0;
    // shift the registers
    for (int j = N - 1; j > 0; j--) {
      #pragma HLS unroll
      shift_reg[j] = shift_reg[j - 1];
    }
    …
    // do multiply-accumulate operation
    for (j = 0; j < N; j++) {
      #pragma HLS unroll
      acc += shift_reg[j] * coeff[j];
    }
    …
  }
}
```

Pipeline the entire outer loop

```
// loop through each output
for (int i = 0; i < SIZE; i ++ ) {
  #pragma HLS pipeline II=1
  int acc = 0;

  …

  // put the new input value into the
  // first register
  shift_reg[0] = input[i];

  …
}
```

Architecture after pipelining



Online resources

- VIVADO HLS https://www.xilinx.com/video/hardware/getting-started-vivado-high-level-synthesis.html
- G. Martin and G. Smith, “High-Level Synthesis: Past, Present, and Future,” IEEE Design & Test of Computers, IEEE, vol. 26, no. 4, pp. 18–25, July 2009.
- Vivado Design Suite Tutorial, High-Level Synthesis, UG871, Nov. 2014
- Vivado Design Suite User Guide, High-Level Synthesis, UG902, Oct. 2014
- Introduction to FPGA Design with Vivado High-Level Synthesis, UG998, Jul. 2013.