FPGA
Field-Programmable Gate Arrays

Paolo Burgio
paolo.burgio@unimore.it

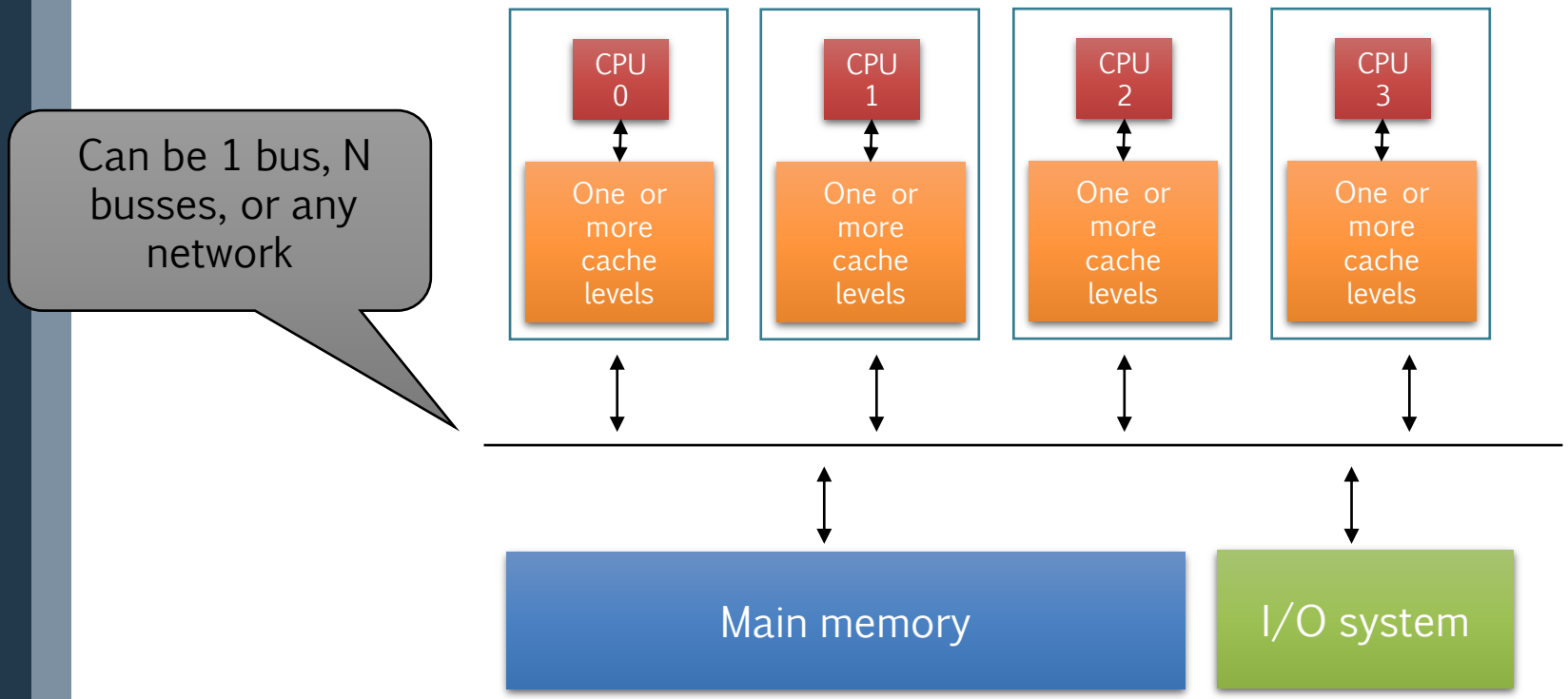

MACHINE
LEARNING
acer
acer
acer

Outline

- Introduction to FPGAs
- How to use them
- Heterogeneous programming
- FPGA-based heterogeneous programming
- How to program it
- Xilinx: now and soon…

The world, till now

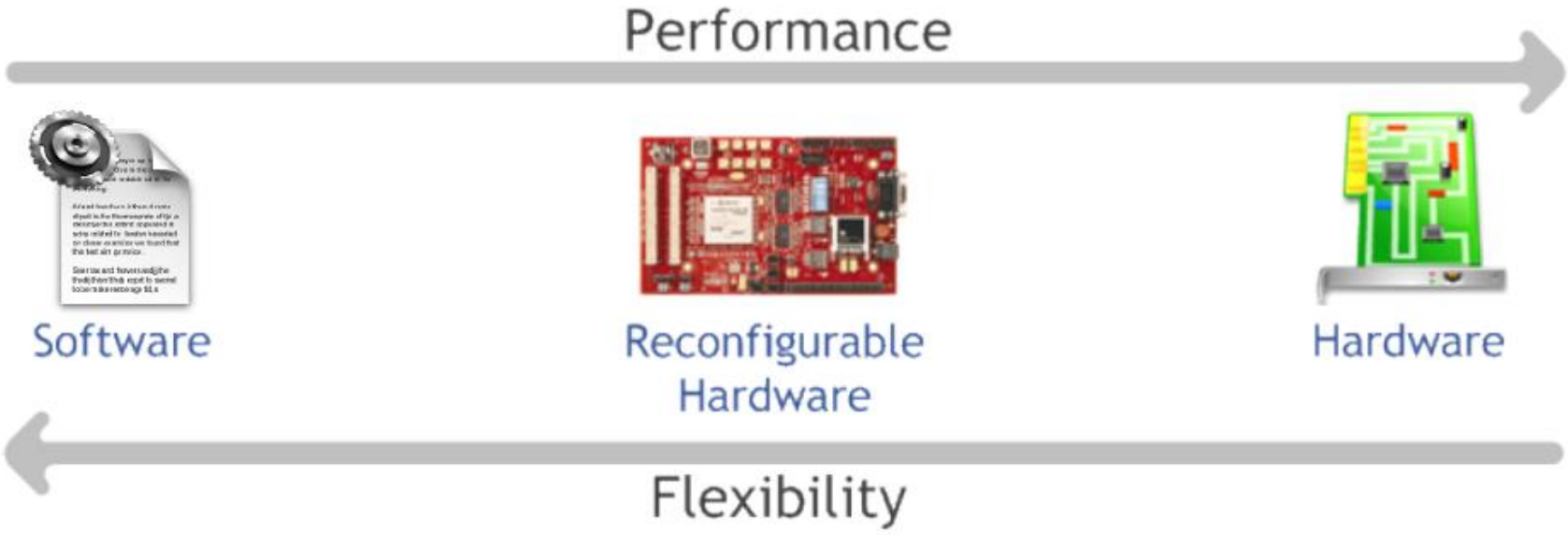
- (A)Symmetric multi-processing
  - Single or multi-core



Reconfigurable Hardware

"Reconfigurable computing is intended to fill the gap between hardware and software, achieving potentially much higher performance than software, while maintaining a higher level of flexibility than hardware

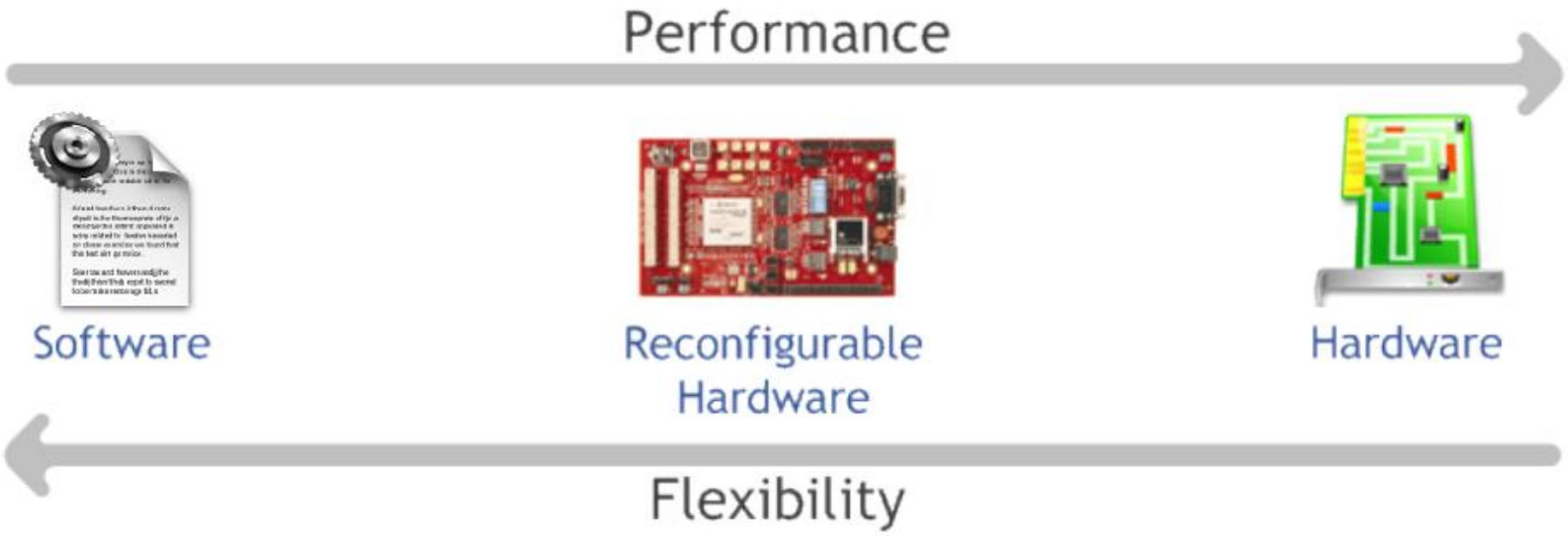
(K. Compton and S. Hauck, *Reconfigurable Computing: a Survey of Systems and Software*, 2002)



…eh?

We are used to have

› On one (left) side, full programmable artifacts (software)
  – Run on single or multi-cores, designed for General Purpose computing
› On one (right) side, hardware blocks to perform specific (subset of) operations



What if?

…we had a "sea" of hardware blocks that we can program as we want

- We can build cores
- We can build co-processors
- We can build what we want

What would you use them?

- For prototyping!

Hardware developent process is long and cumbersome

- Imagine a full-fledged cores
- Typically, years of development
- You can "try, and see whether it works"

History of reconfigurable technologies

- Logic gates (1950s-60s)
- Regular structures for two-level logic (1960s-70s)
  - Muxes and decoders, PLAs
- Programmable sum-of-products arrays (1970s-80s)
  - PLDs, complex PLDs
- Programmable gate arrays (1980s-90s)
  - densities high enough to permit entirely new class of application, e.g., prototyping, emulation, acceleration

Field-programmable gate arrays

- *"A field-programmable gate array (FPGA) is an integrated circuit designed to be configured by a customer or a designer after manufacturing."*

- Traditionally used for prototyping
  - Takes minutes vs years for "real" hardware

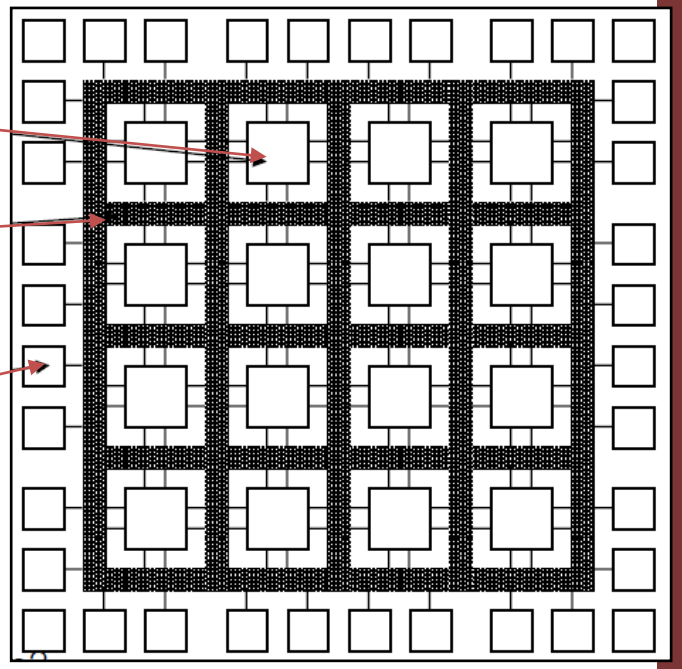
- Tech has evolved so they are actively used in production settings
  - Less powerful, yet more energy-efficient than a GPU
  - Way more flexible

Integrated into System-on-chips

- As reconfigurable accelerator
- We'll see later…

FPGAs

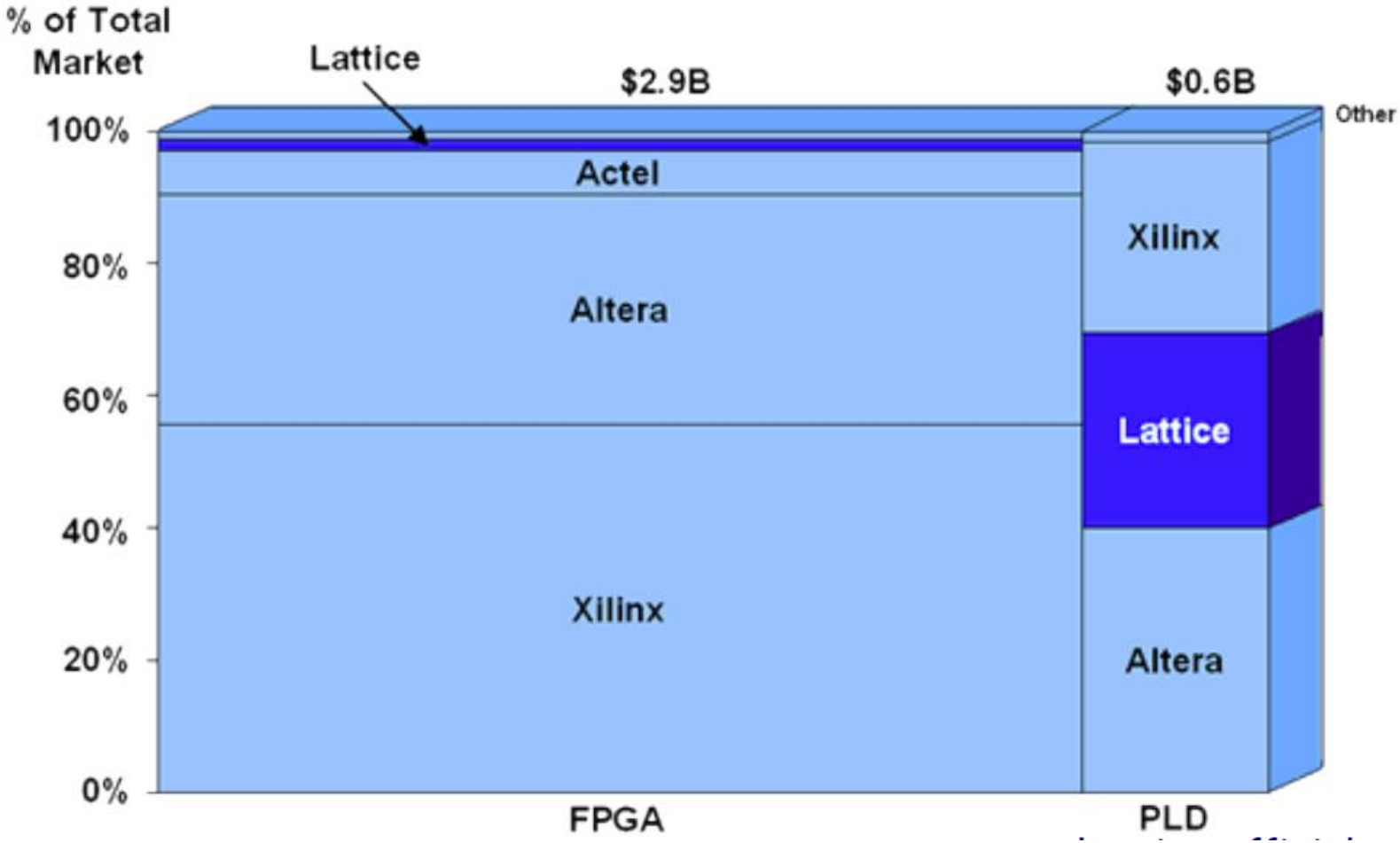
- Logic blocks
  - to implement combinational and sequential logic
- Interconnect
  - wires to connect inputs and outputs to logic blocks
- I/O blocks
  - special logic blocks at periphery of device for external connections



Key questions:

- how to make logic blocks programmable?
- how to connect the wires?
- after the chip has been fabbed

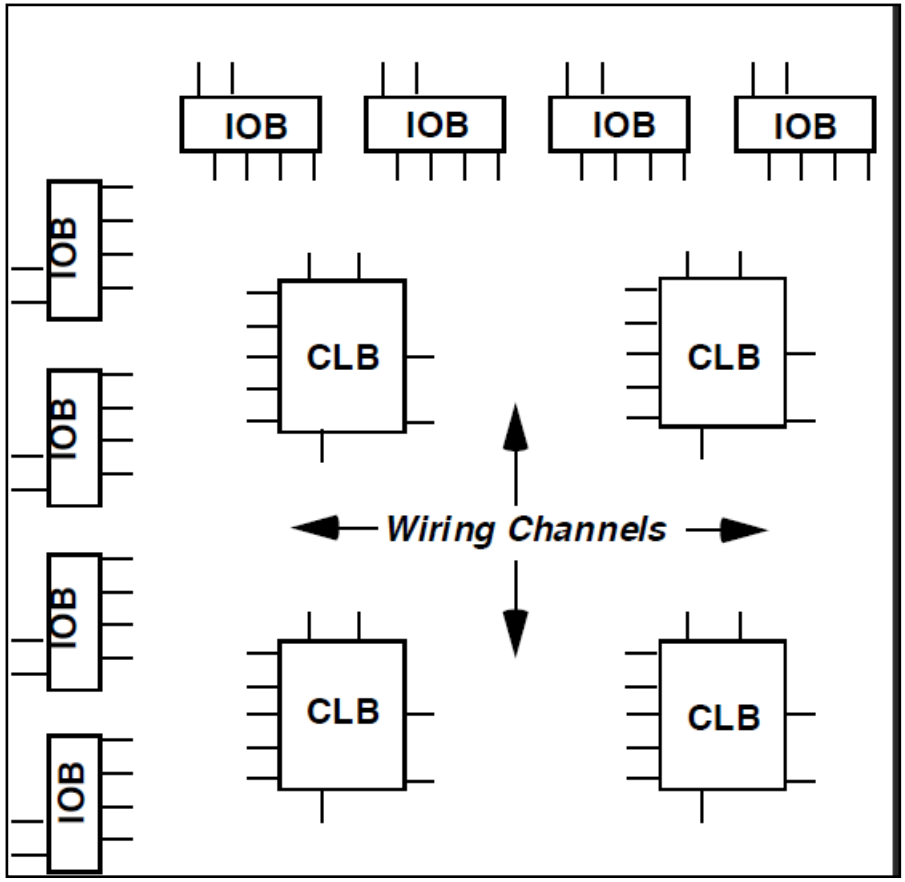
Commercial FPGA companies



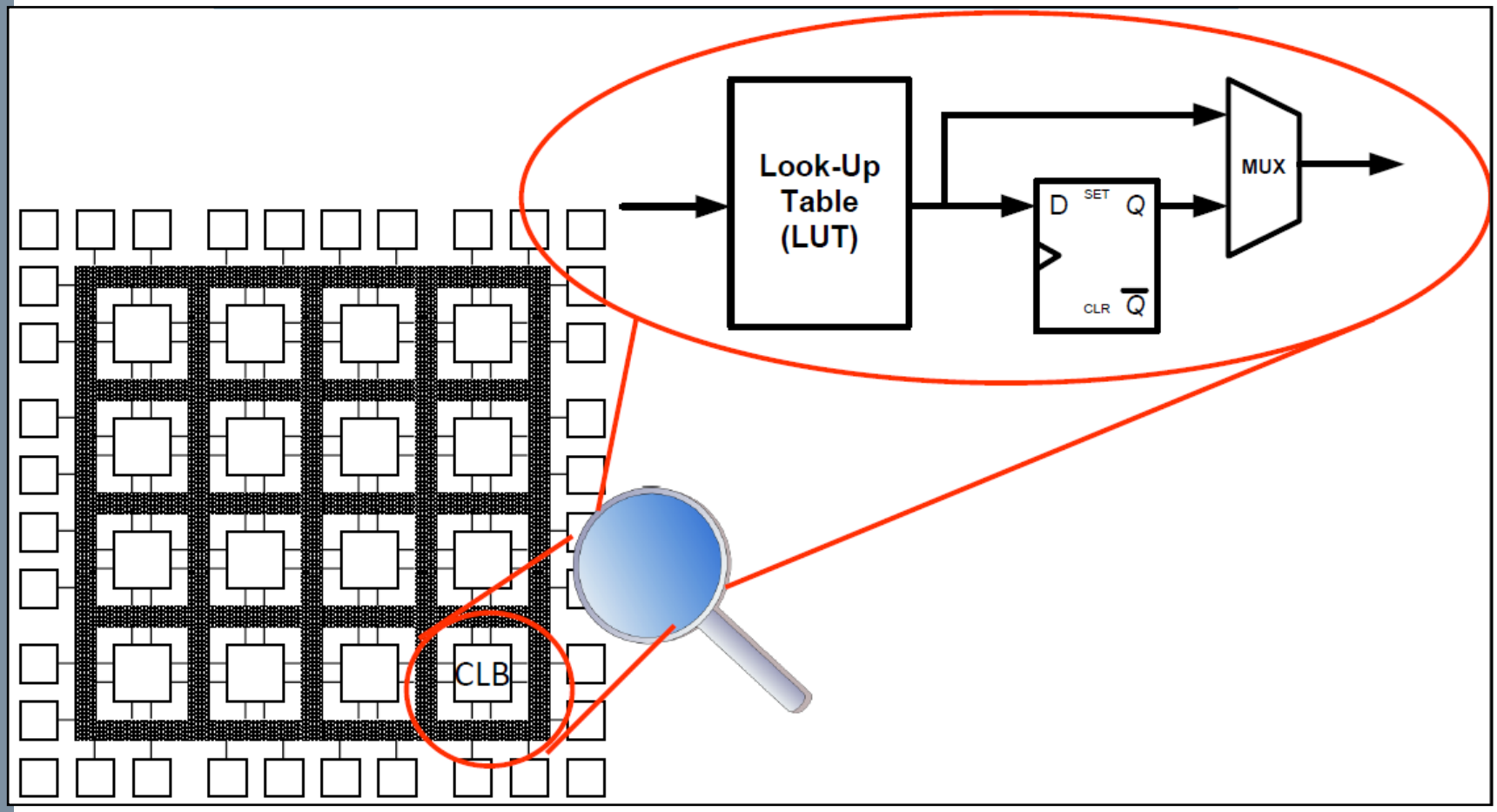
(Xilinx) Programmable Gate Arrays

CLB - Configurable Logic Block

- Built-in fast carry logic
- Can be used as memory
- Three types of routing
  - direct
  - general-purpose
  - long lines of various lengths
- RAM-programmable
  - can be reconfigured



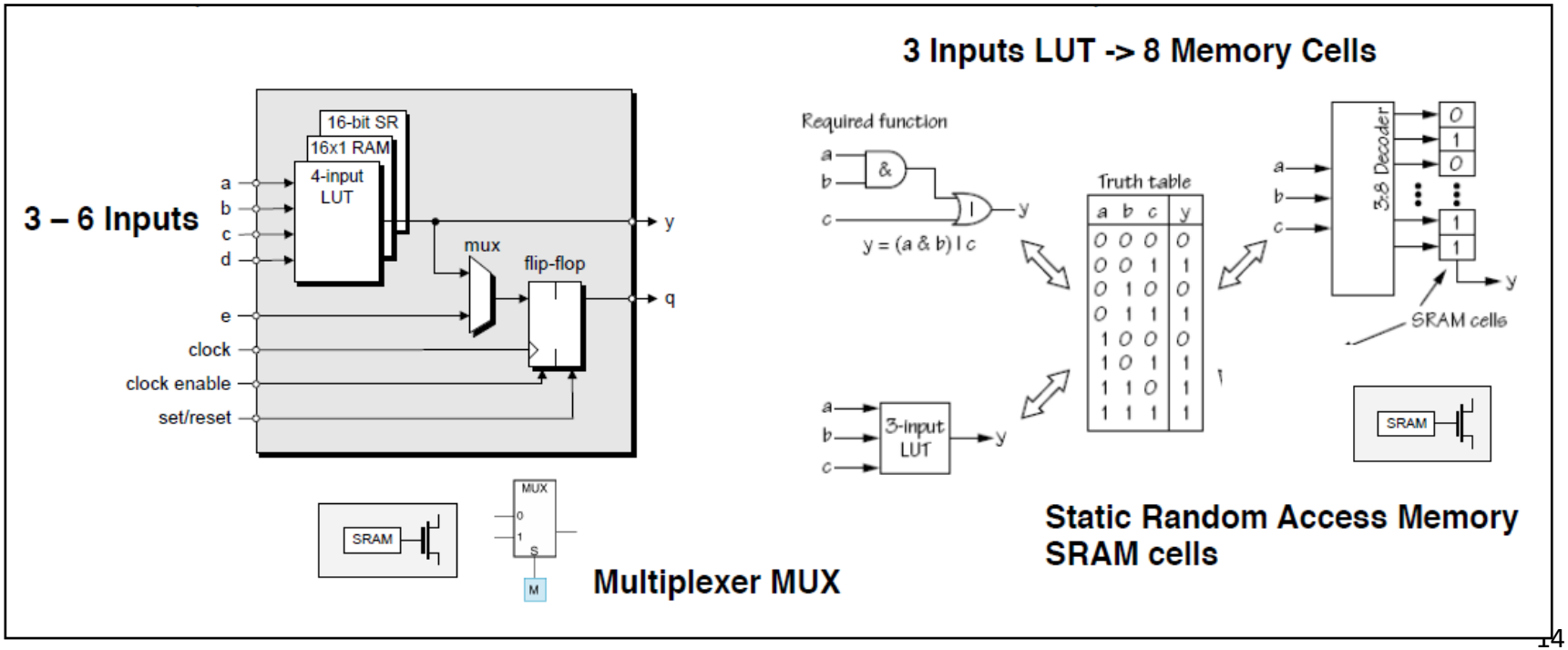
Simplified CLB Structure



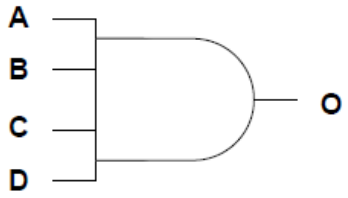
LookUp Tables

LUT contains Memory Cells to implement small logic functions

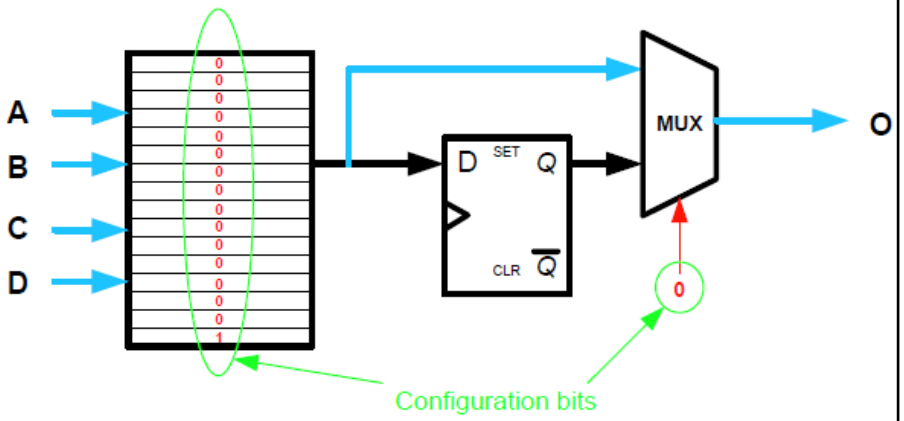
- Each cell holds ‘0’ or ‘1’
- Programmed with outputs of Truth Table
- Inputs select content of one of the cells as output



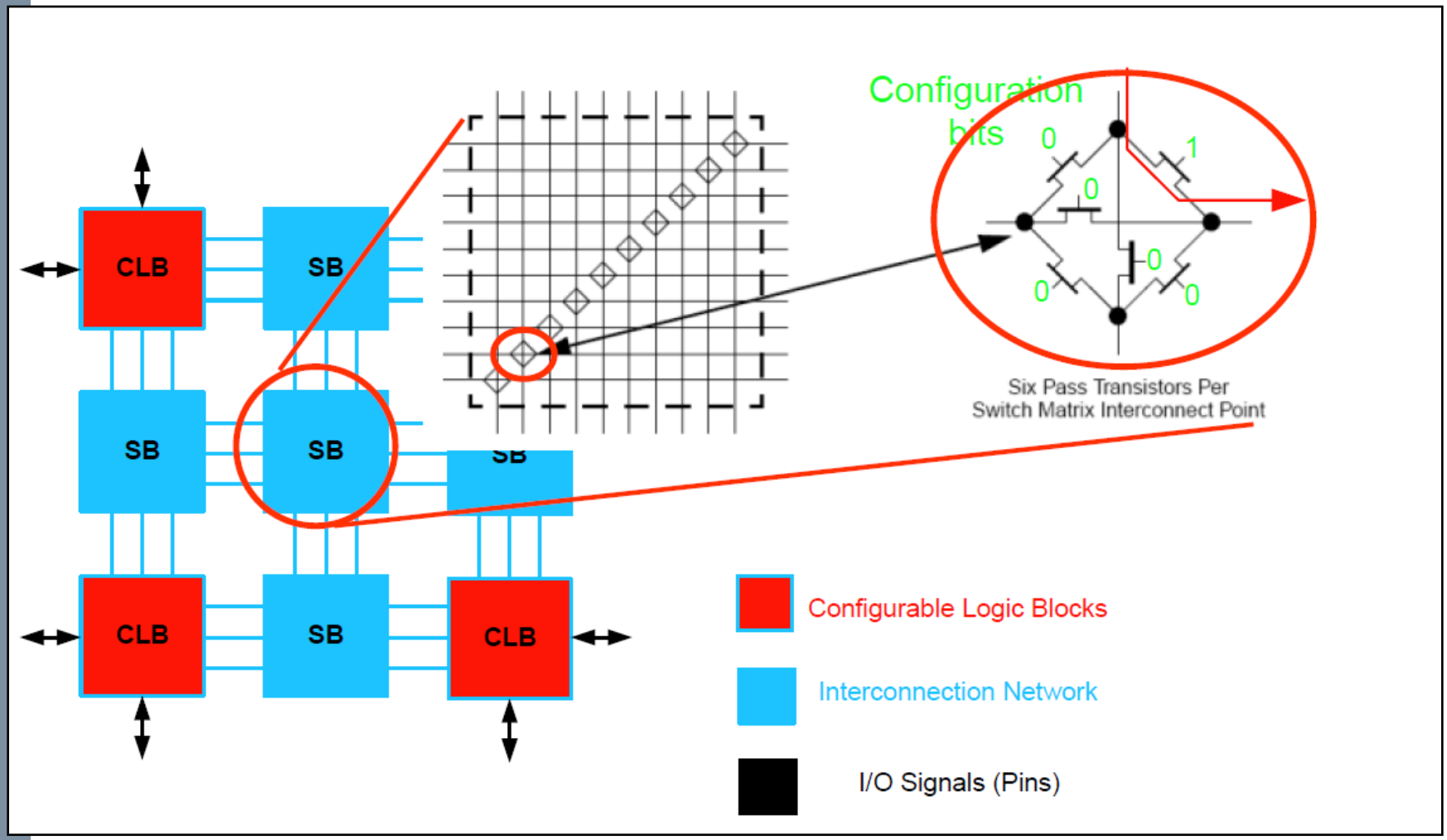
Example: 4-input AND gate



A	B	C	D	O
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

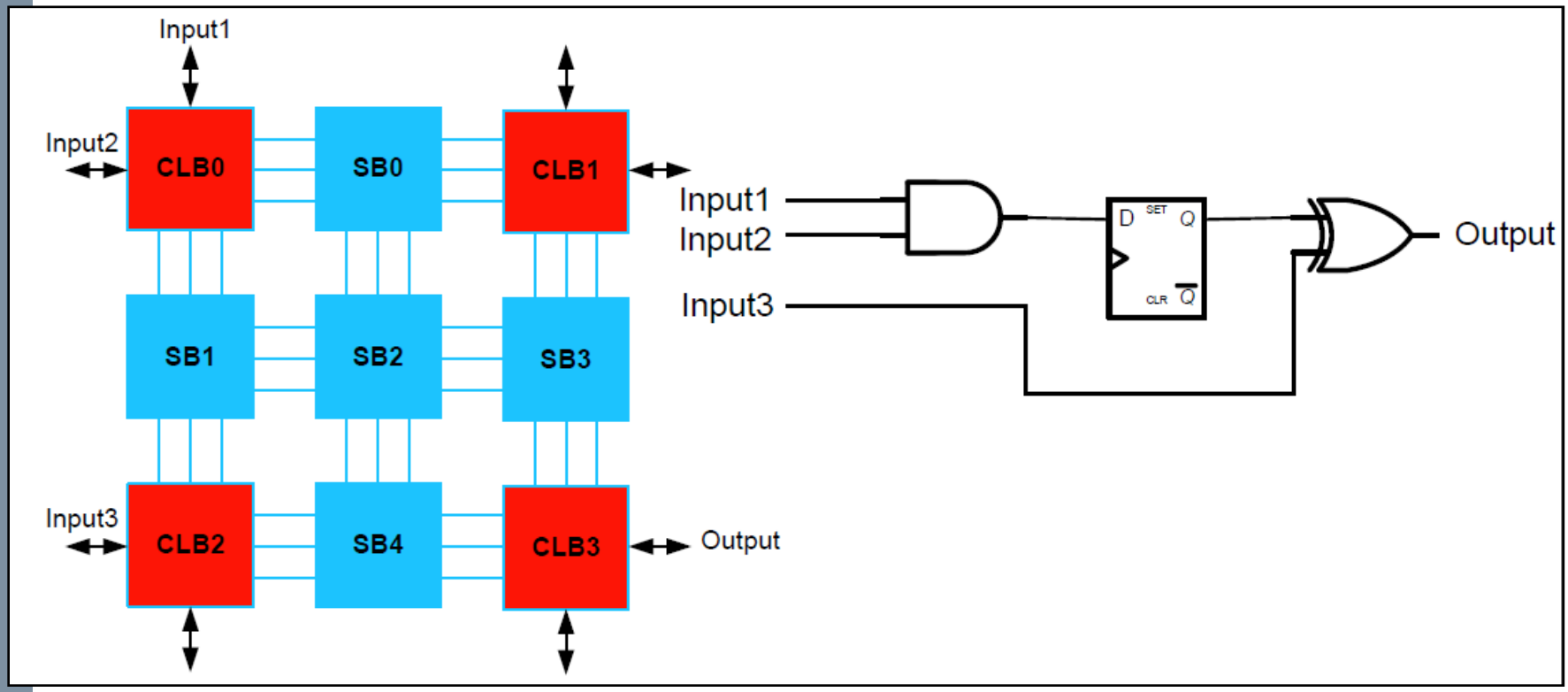


Interconnection Network

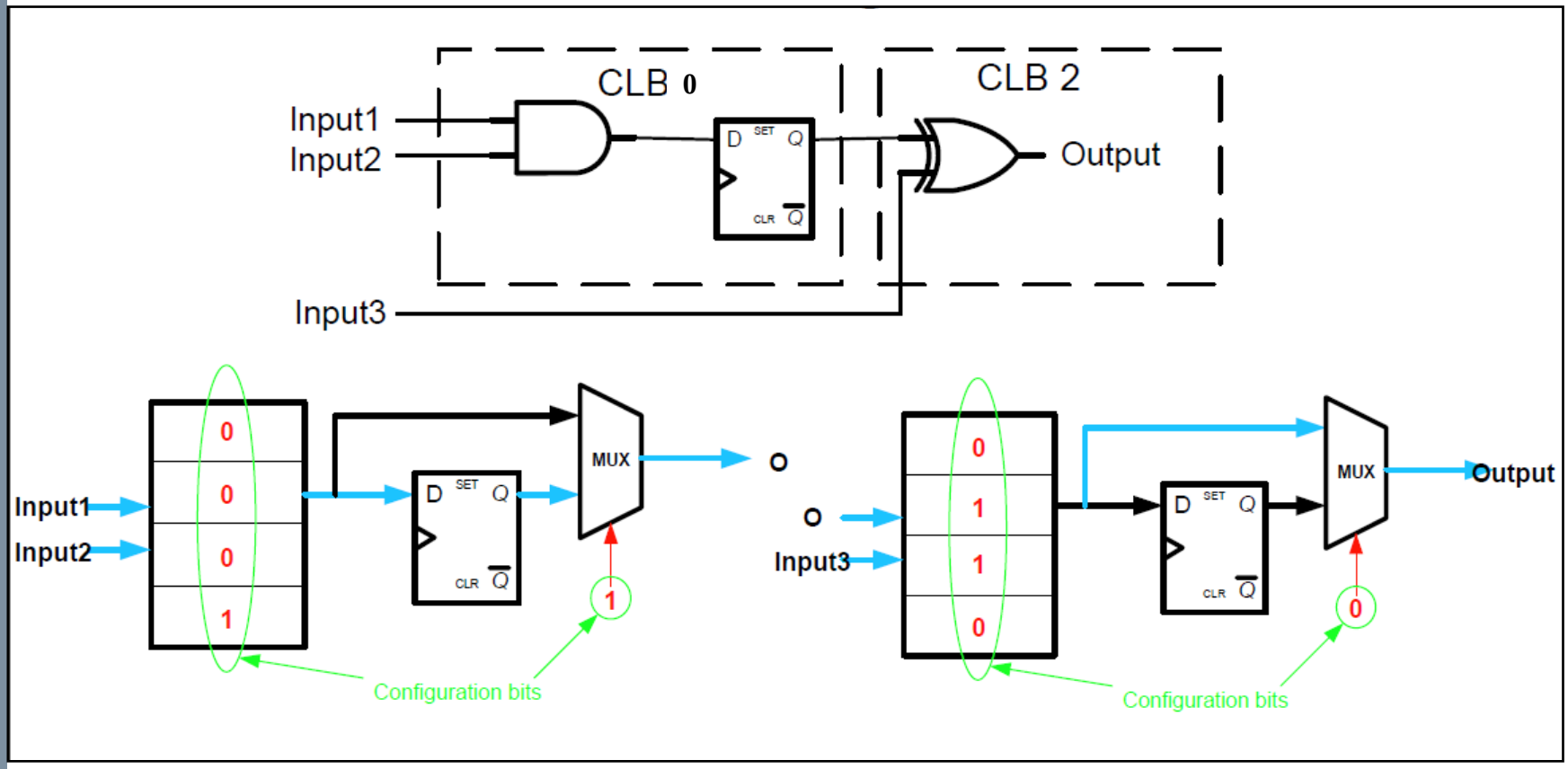


Example

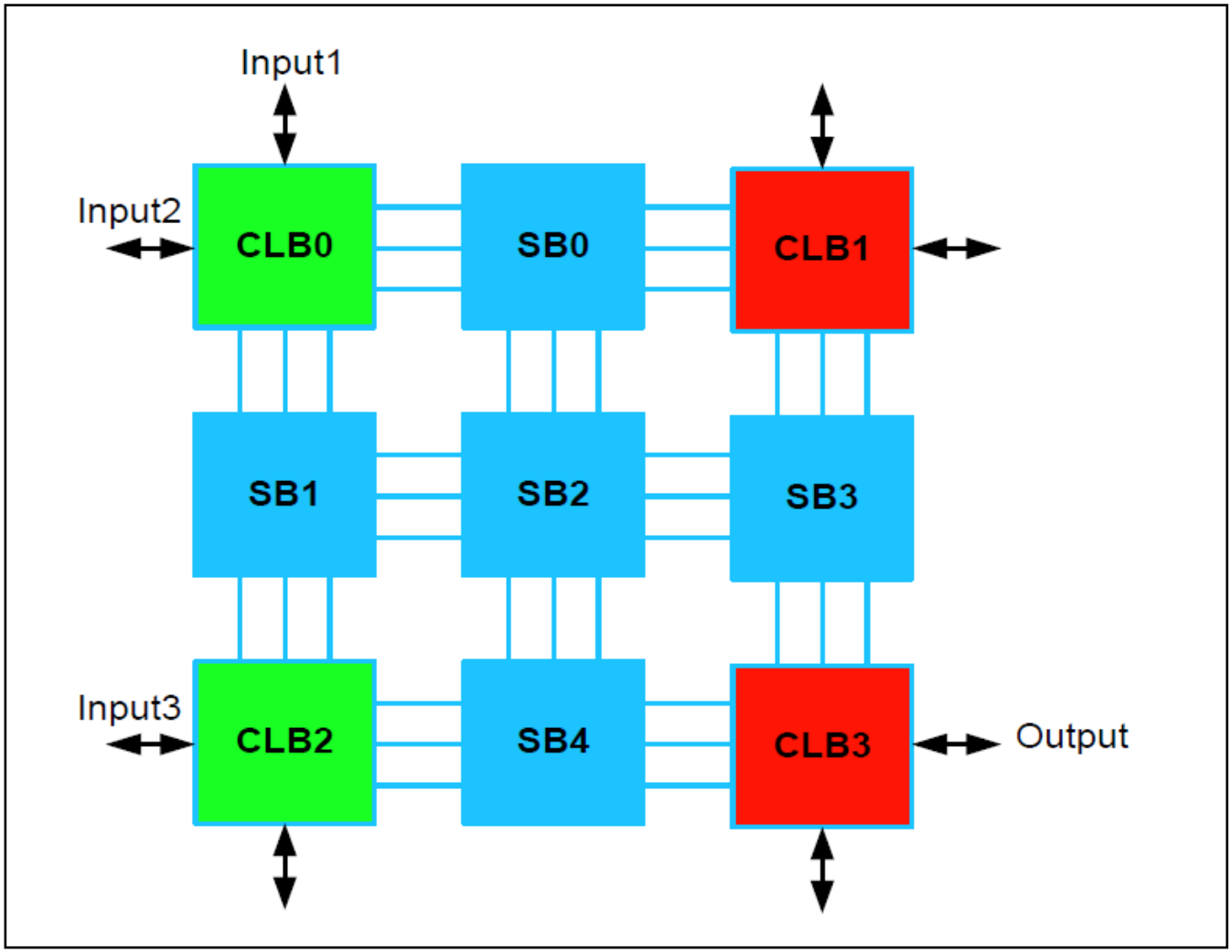
› Determine the configuration bits for the following circuit implementation in a 2x2 FPGA, with I/O constraints as shown in the following figure. Assume 2-input LUTs in each CLB



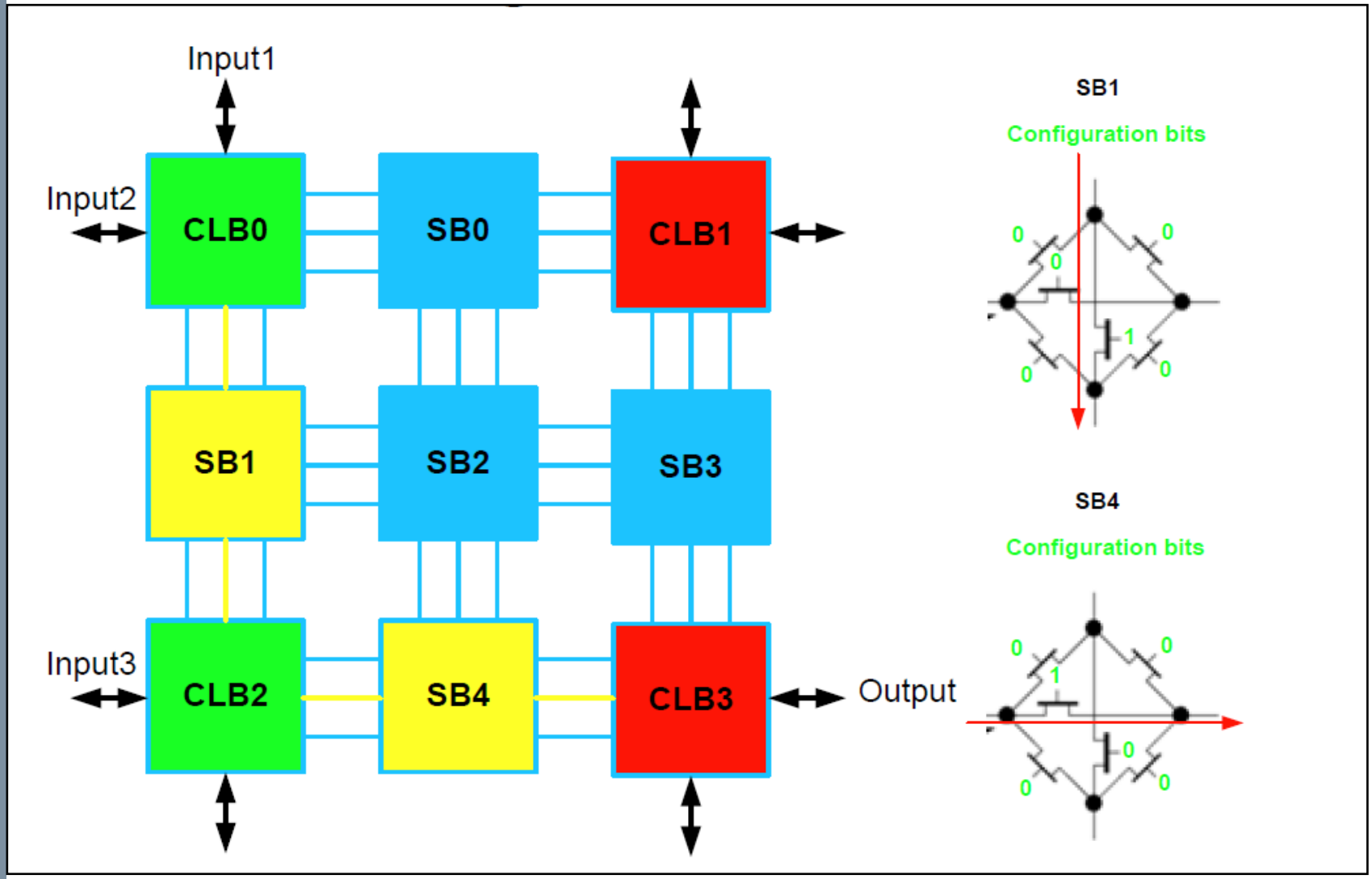
Configure CLBs



Placement: select CLBs



Routing: Select path



Configuration Bitstream

- The configuration bitstream must include ALL CLBs and SBs, even unused ones
- CLB0: 00011
- CLB1: ?????
- CLB2: 01100
- CLB3: XXXXX
- SB0: 000000
- SB1: 000010
- SB2: 000000
- SB3: 000000
- SB4: 000001

Some Definitions

- Object Code (aka "Bitstream): the executable active physical (either HW or SW) implementation of a given functionality
- Core: a specific representation of a functionality. It is possible, for example, to have a core described in VHDL, in C or in an intermediate representation (e.g. a DFG)
- IP-Core: a core described using a HD Language combined with its communication infrastructure (i.e. the bus interface)
- Reconfigurable Functional Unit: an IP-Core that can be plugged and/or unplugged at runtime in an already working architecture
- Reconfigurable Region: a portion of the device area used to implement a reconfigurable core

Computer-Aided Design

Can't design FPGAs by hand!

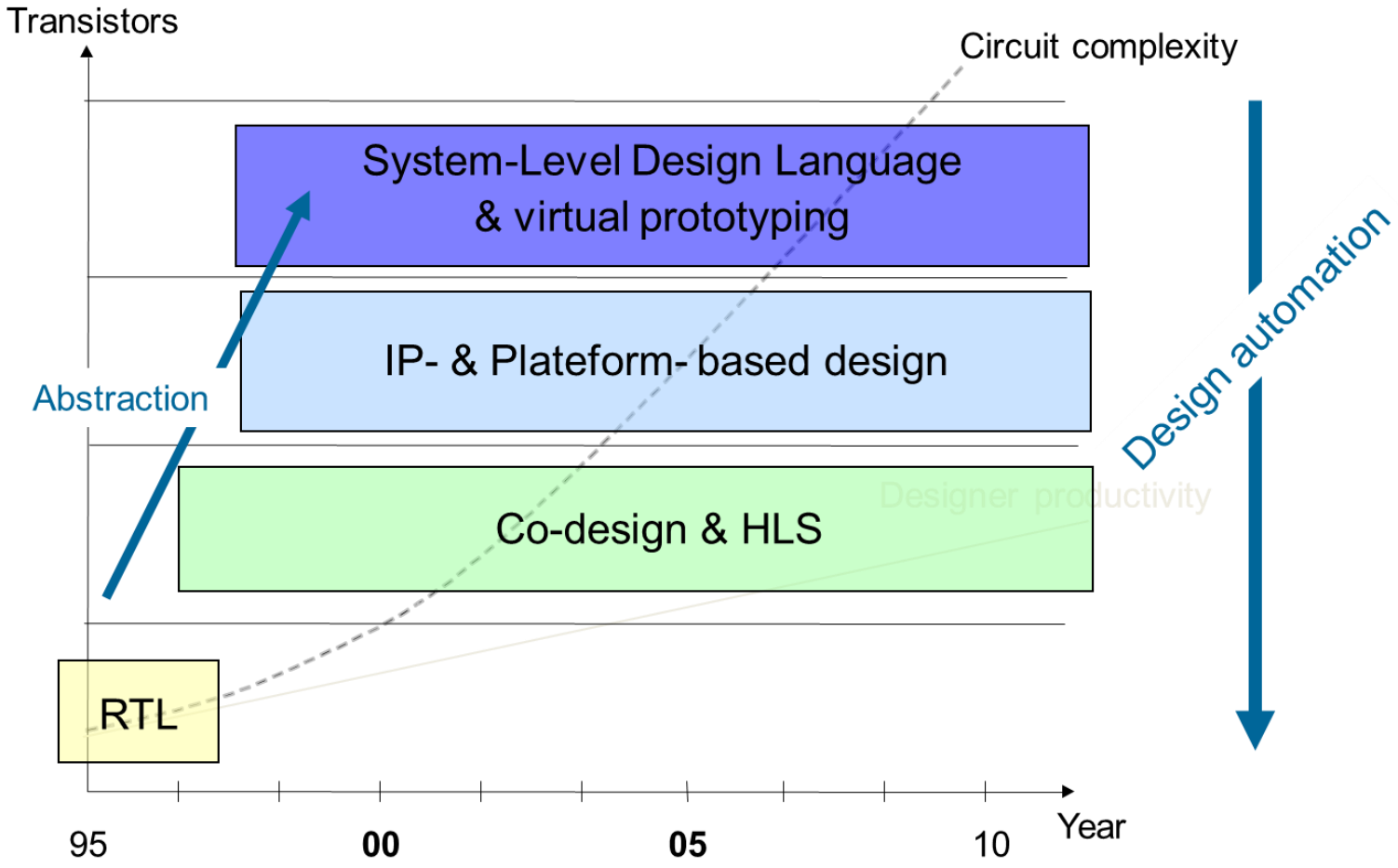
- way too much logic to manage, hard to make changes
- Hardware description languages (HDL), es: Verilog, VHDL
  - specify functionality of logic at a high level
- Logic synthesis
  - process of compiling HDL program into logic gates and flip-flops
- Validation - high-level simulation to catch specification errors
  - verify pin-outs and connections to other system components
  - low-level to verify and check performance

CAD Tool Path (cont'd)

- Technology mapping
  - map the logic onto elements available in the implementation technology (LUTs for Xilinx FPGAs)
- Placement and routing
  - assign logic blocks to functions
  - make wiring connections
- Partitioning and constraining
  - if design does not fit or is unroutable as placed split into multiple chips
  - if design it too slow prioritize critical paths, fix placement of cells, etc.
  - few tools to help with these tasks exist today
- Generate programming files - bits to be loaded into chip for configuration

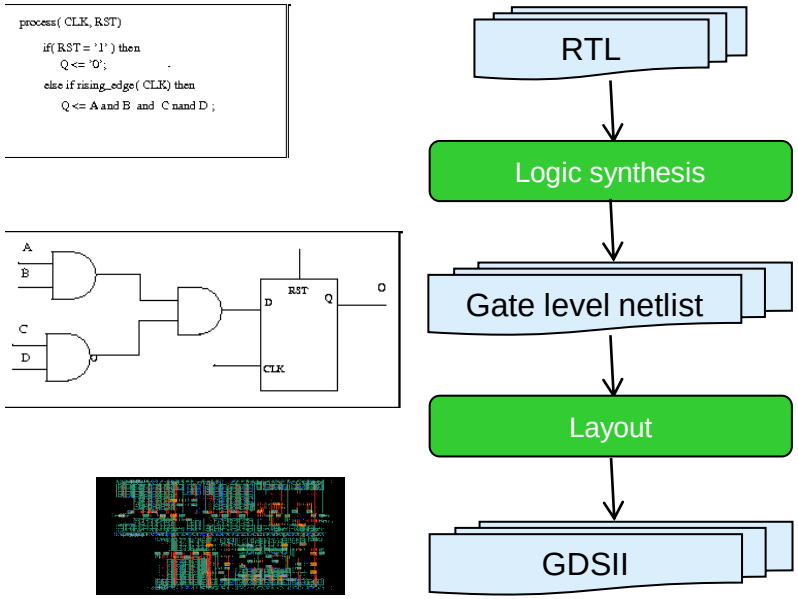
High-level synthesis

Electronic System Level Design (ESLD)



Typical HW design flow

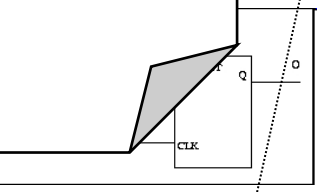
› Starting from a Register Transfer Level description, generate an IC layout

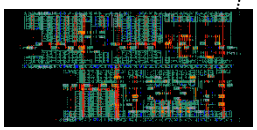


Typical HW design flow

› Starting from a Register Transfer Level description, generate an IC layout

```
#define N 2

typedef int matrix[N][N];

int main(const matrix A, matrix C)
{

   const matrice B ={{1, 2},{ 3, 4}};
   int tmp;
   int i,j,k;

      for (i=0;i<N;i++)
           for (j=0;j<N;j++){
              tmp = A[i][0]*B[0][j];

           for (k=1;k<N - 1;k++)
             tmp =  tmp + A[i][k] * B[k][j];

           C[i][j] = tmp + A[i][N-1] * B[N-1][j];
        }

     return 0;
}
```





Algorithm

High-Level synthesis

RTL

SystemC simulation models (CABA/TLM)

Logic synthesis

Virtual prototyping

Gate level netlist

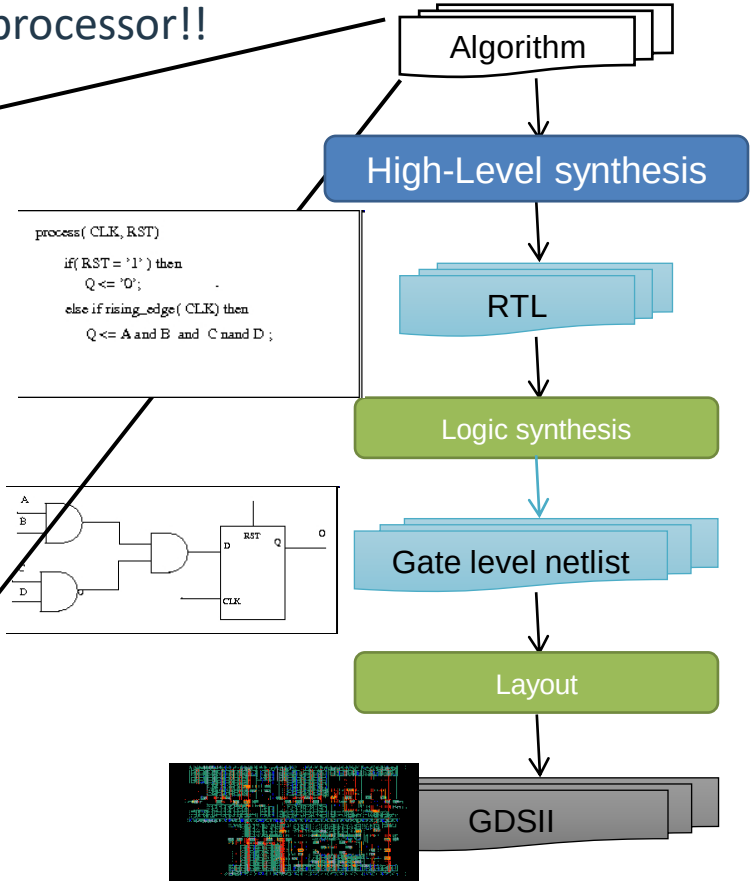
Layout

GDSII

High-Level Synthesis

- Starting from C code, generate RTL, and then, layout
  - Typically, HW accelerators (FFT?) Ips
  - Can also synthesize a fully fledged processor!!
  - Soft-cores

```
#define N 2
typedef int matrix[N][N];

int main(const matrix A, matrix C)
{
  const matrice B ={{1, 2},{ 3, 4}};
  int tmp;
  int i,j,k;
  for (i=0;i<N;i++)
    for (j=0;j<N;j++) {
      tmp = A[i][0]*B[0][j];
      for (k=1;k<N - 1;k++)
        tmp = tmp + A[i][k] * B[k][j];
        C[i][j] = tmp + A[i][N-1] *
                  B[N-1][j];
    }
  return 0;
}
```



It is basically, a compiler!

- From C code
  - Generates the "physical" representation of Hardware modules
  - Registry Transfer Level, RTL
  - That will be deployed on the board

- Automatically

Synthesizable C subset

- No pointers
  - Statically unresolved
  - Arrays are allowed!
- No standard function call
  - printf, scanf, fopen, malloc…
- Function calls are allowed
  - Can be in-lined or not
- Nearly all datatypes are allowed
  - Specific datatypes are encouraged
  - Bit accurate integers, fixed point, signed, unsigned…

Example #1: a simple C code

```c
#define N 16

int main(int data_in, int *data_out) {
  static const int Coeffs [N] = { 98, -39, -327, 439, 950, -2097, -1674, 9883,
                                  9883, -1674, -2097, 950, 439, -327, -39, 98 };

  int Values[N];
  int temp;
  int sample,i,j;

  sample = data_in;
  temp = sample * Coeffs[N-1];

  for(i = 1; i<=(N-1); i++) {
    temp += Values[i] * Coeffs[N-i-1];
  }

  for(j=(N-1); j>=2; j-=1 ) {
    Values[j] = Values[j-1];
  }

  Values[1] = sample;
  *data_out=temp;

  return 0;
}
```

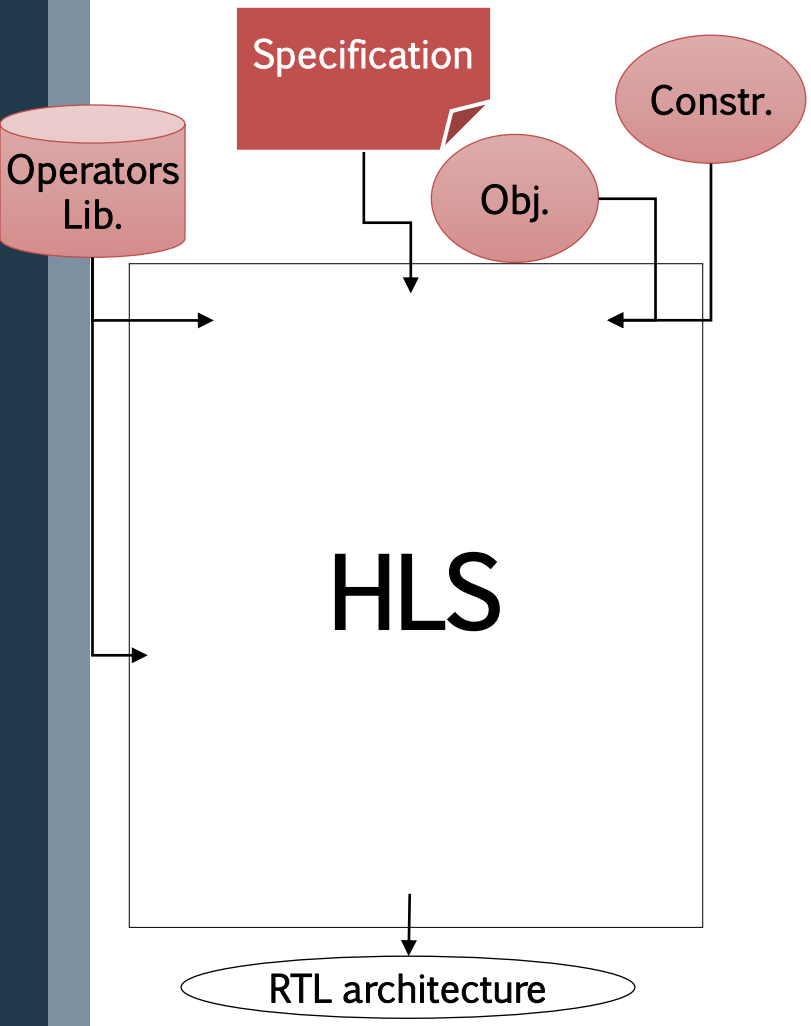
Example #2: bit accurate C++ code

```
#include "ac_fixed.h" // From Mentor Graphics
#define PORT_SIZE ac_fixed<16, 12, true, AC_RND,AC_SAT> // 16 bits, 12 bits after the \
   point, quantization = rounding, overflow = saturation
#define N 16

int main(PORT_SIZE data_in, PORT_SIZE &data_out) {
  static const PORT_SIZE Coeffs [N]= { 1.1, 1.5, 1.0, 1.0, 1.7, 1.8, 1.2, 1.0,
                                       1.6, 1.0, 1.5, 1.1, 1.9, 1.3, 1.4, 1.7 };
  PORT_SIZE Values[N];
  PORT_SIZE temp;
  PORT_SIZE sample;

  sample= data_in;
  temp = sample * Coeffs[N-1];

  for(int i = 1; i<=(N-1); i++) {
    temp = Values [i] * Coeffs[N-i-1] + temp;
  }

  for(int j=(N-1); j>=2; j-=1 ) {
    Values[j] = Values [j-1];
  }

  Values[1] = sample;
  data_out=temp;

  return 0;
 }
```

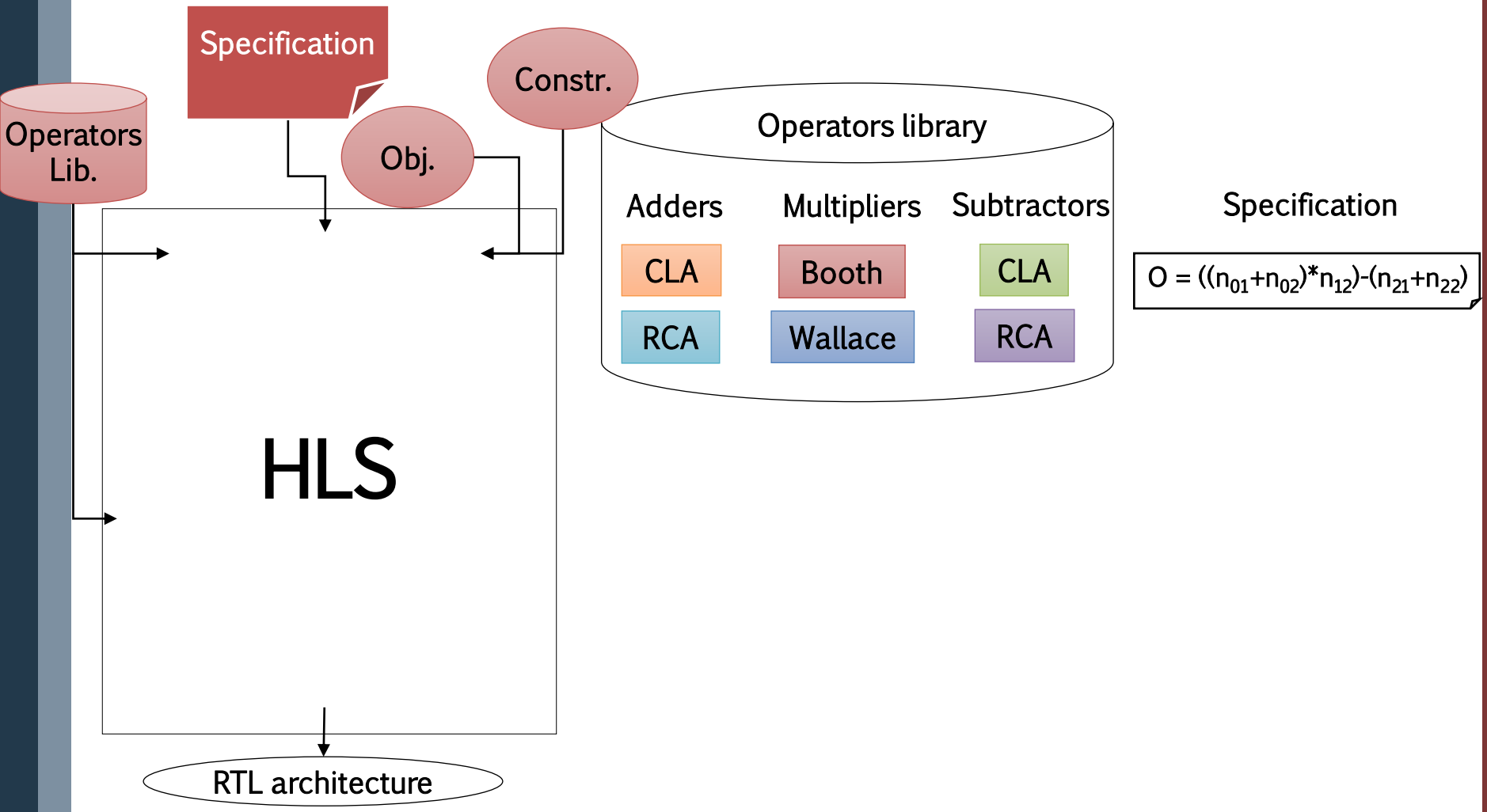
High-level transformations

- Loops
  - Loop pipelining,
  - Loop unrolling
  - Loop merging
  - Loop tiling
  - …
- Arrays mapping
  - Arrays can be mapped on memory banks
  - Arrays can be synthesized as registers
  - Constant arrays can be synthesized as logic
  - …
- Functions
  - Function calls can be in-lined
  - Function is synthesized as an operator
    - Sequential, pipelined, functional unit…
  - Single function instantiation
  - …

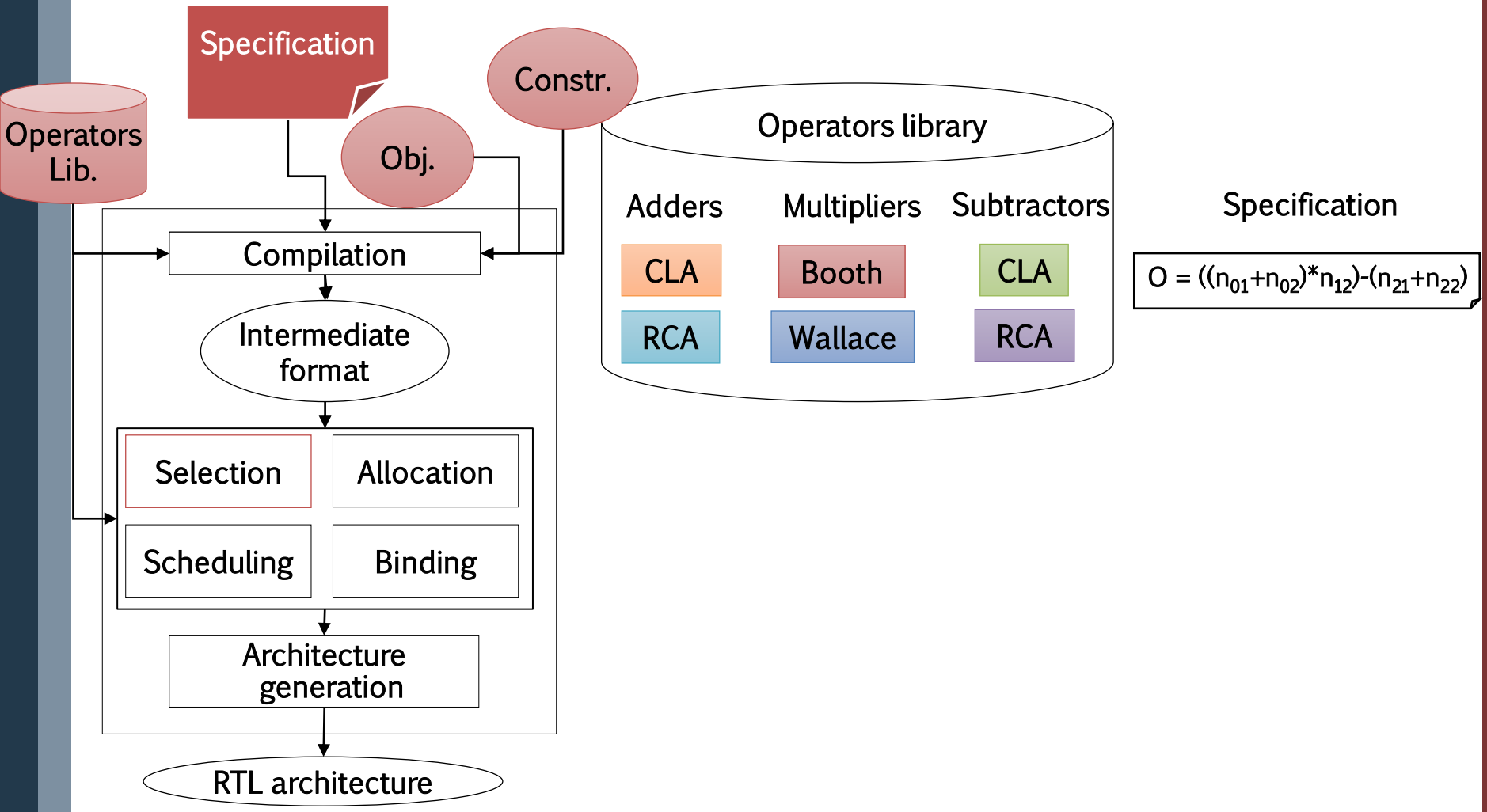
HLS steps: Inputs



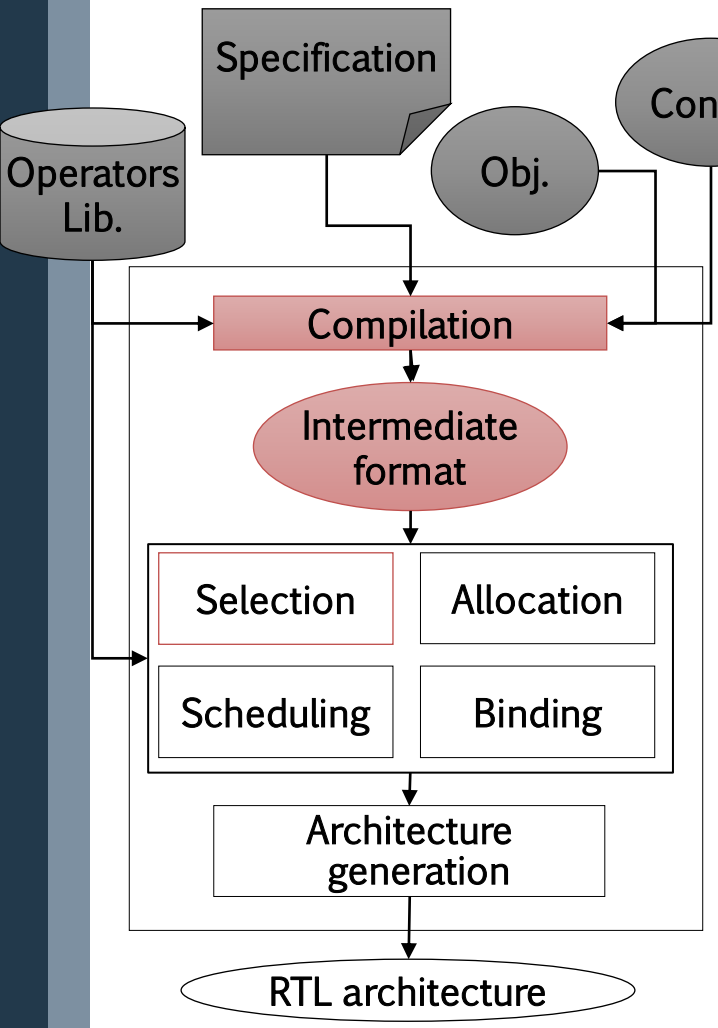
HLS steps: Inputs

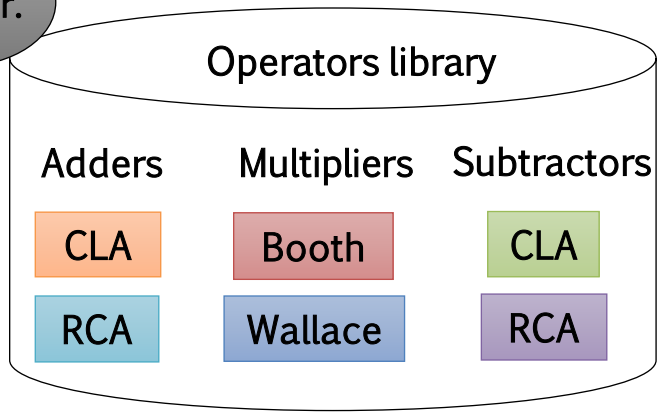


HLS steps: Inputs



HLS steps: Compilation

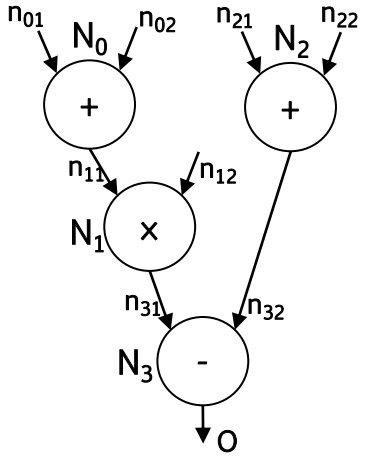




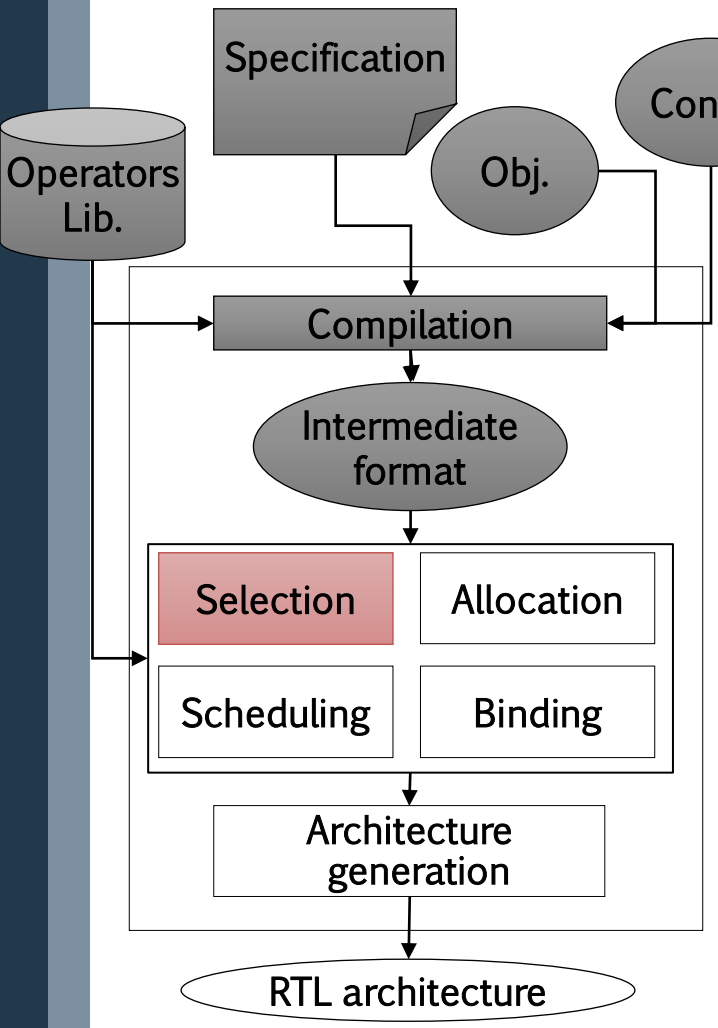
Specification

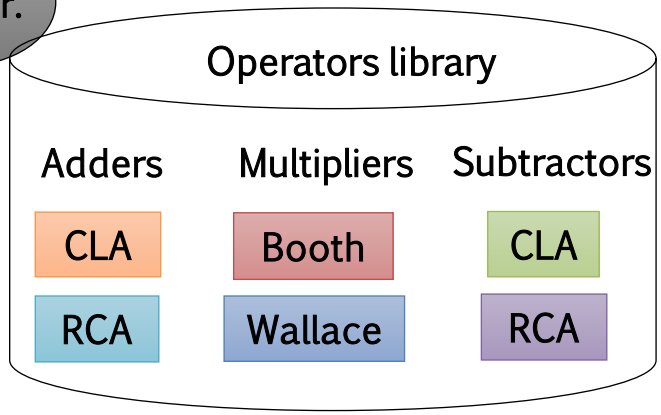
$O = ((n_{01}+n_{02})*n_{12})-(n_{21}+n_{22})$

Intermediate representation



HLS steps: Selection

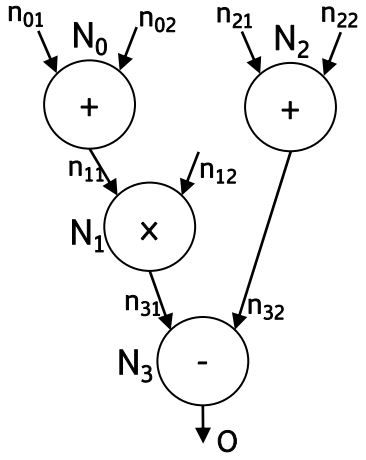




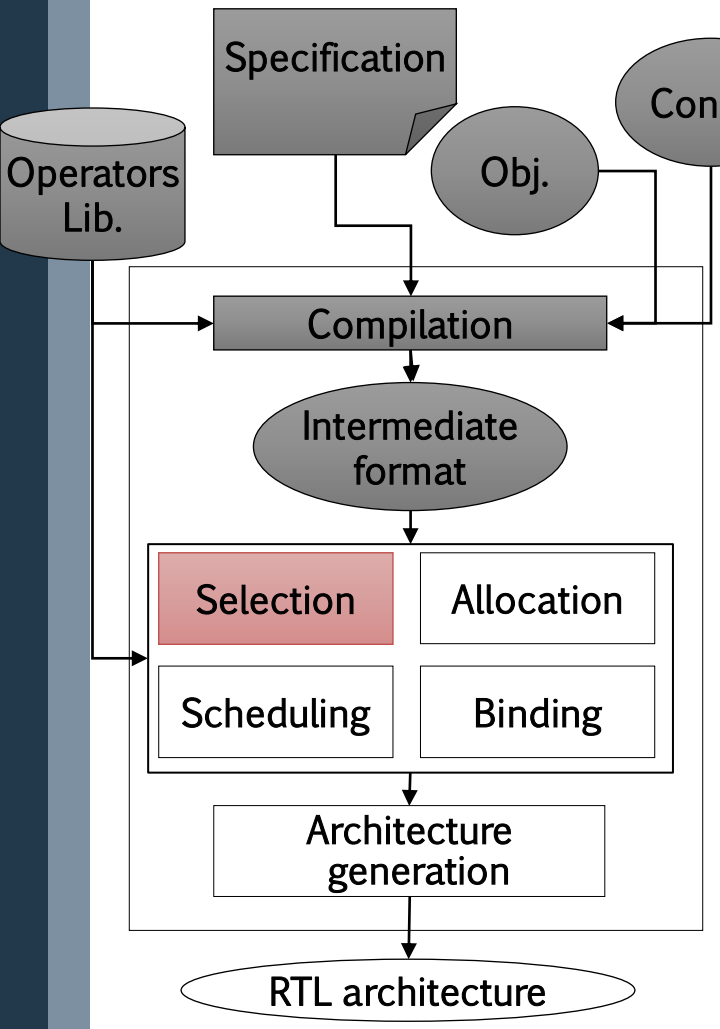
Specification

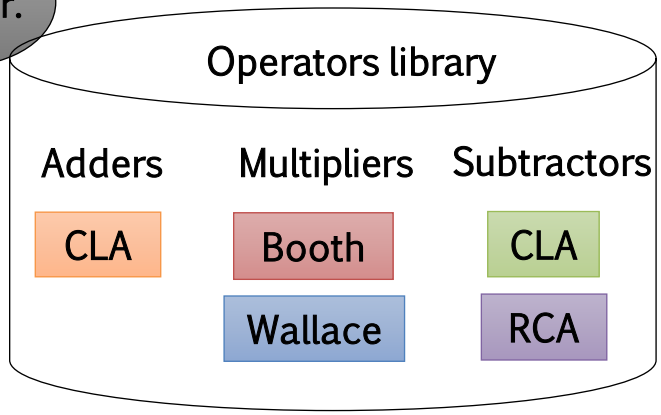
$O = ((n_{01}+n_{02})*n_{12})-(n_{21}+n_{22})$

Intermediate representation



HLS steps: Selection



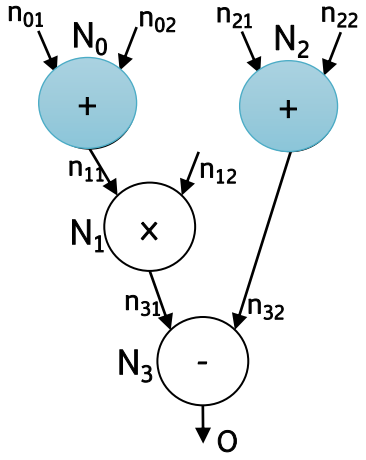


RCA

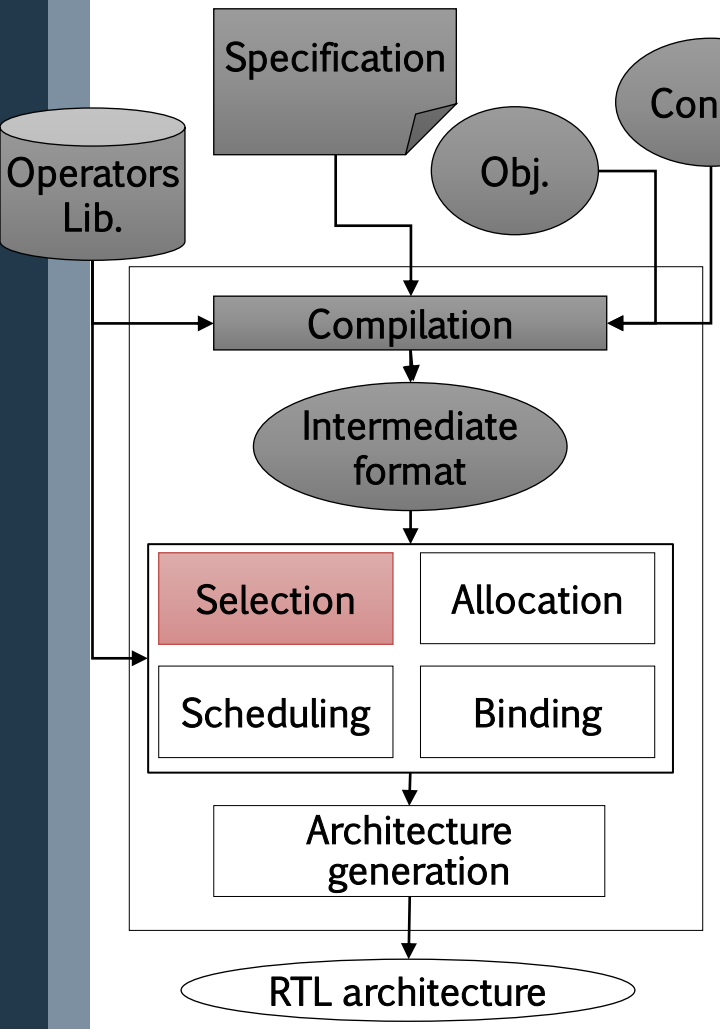
Specification

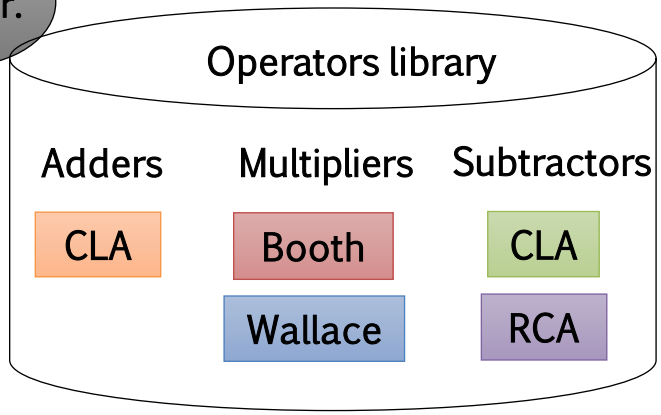
$O = ((n_{01}+n_{02})*n_{12})-(n_{21}+n_{22})$

Intermediate representation



HLS steps: Selection



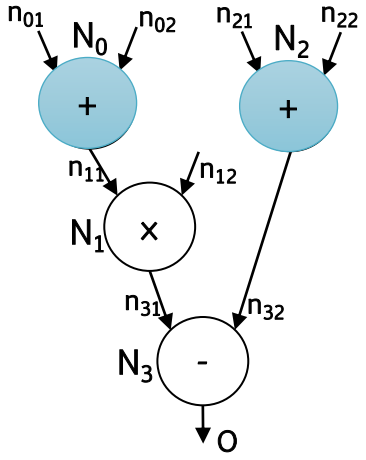


RCA

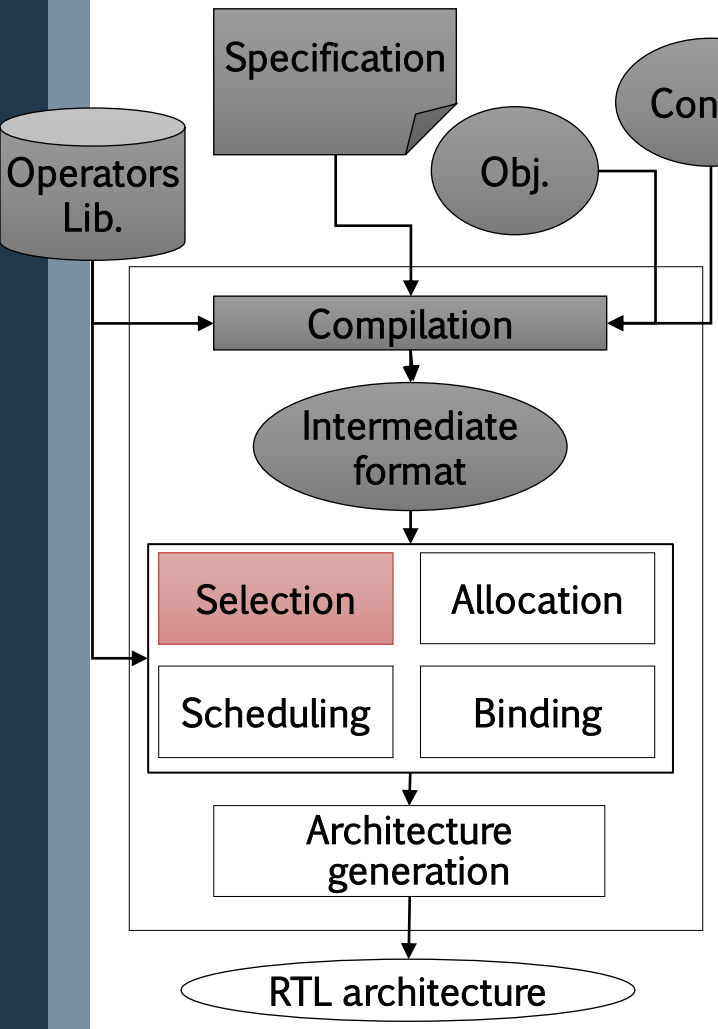
Specification

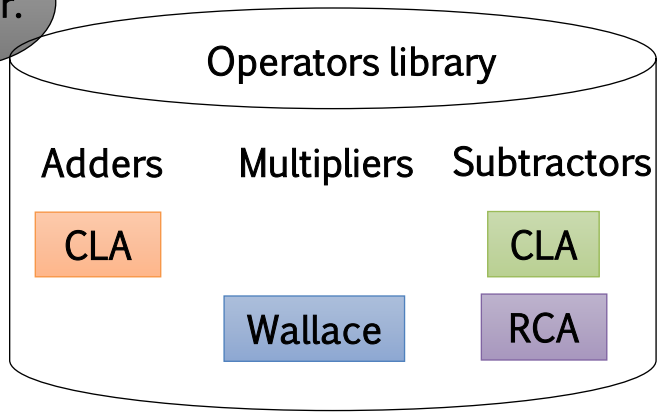
$O = ((n_{01}+n_{02})*n_{12})-(n_{21}+n_{22})$

Intermediate representation



HLS steps: Selection





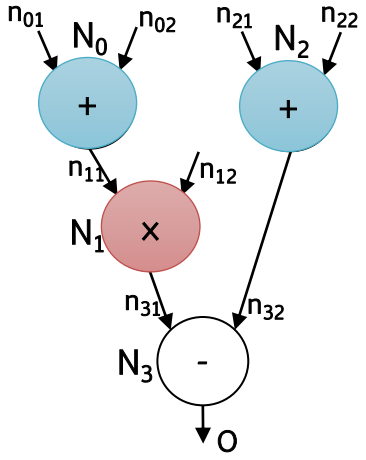
Booth

RCA

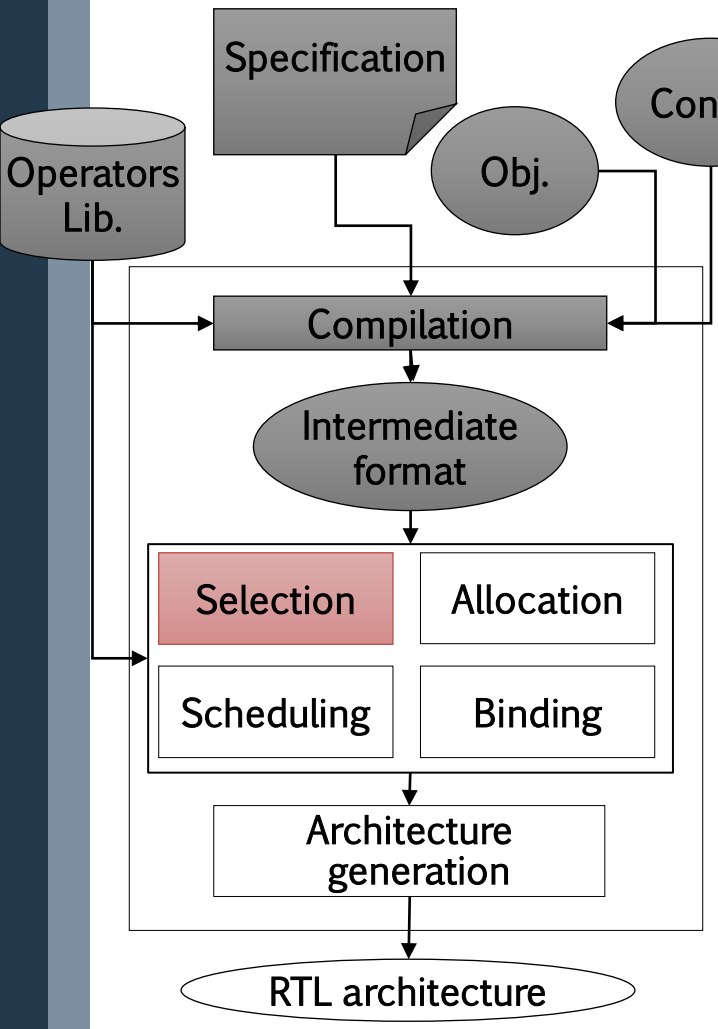
Specification

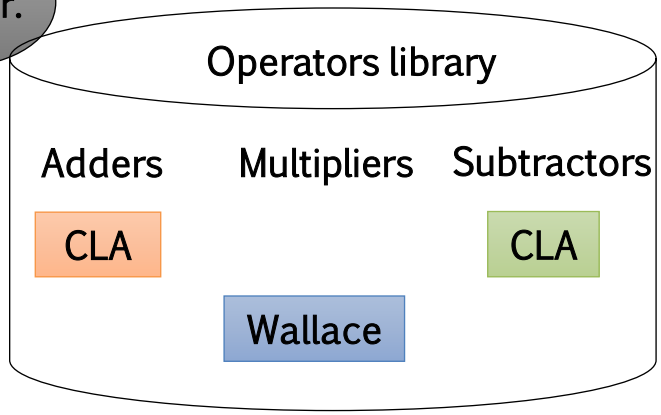
$O = ((n_{01}+n_{02})*n_{12})-(n_{21}+n_{22})$

Intermediate representation



HLS steps: Selection

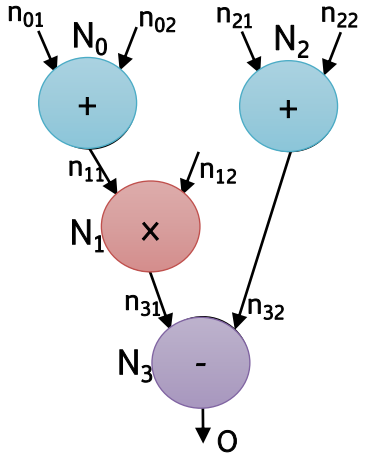




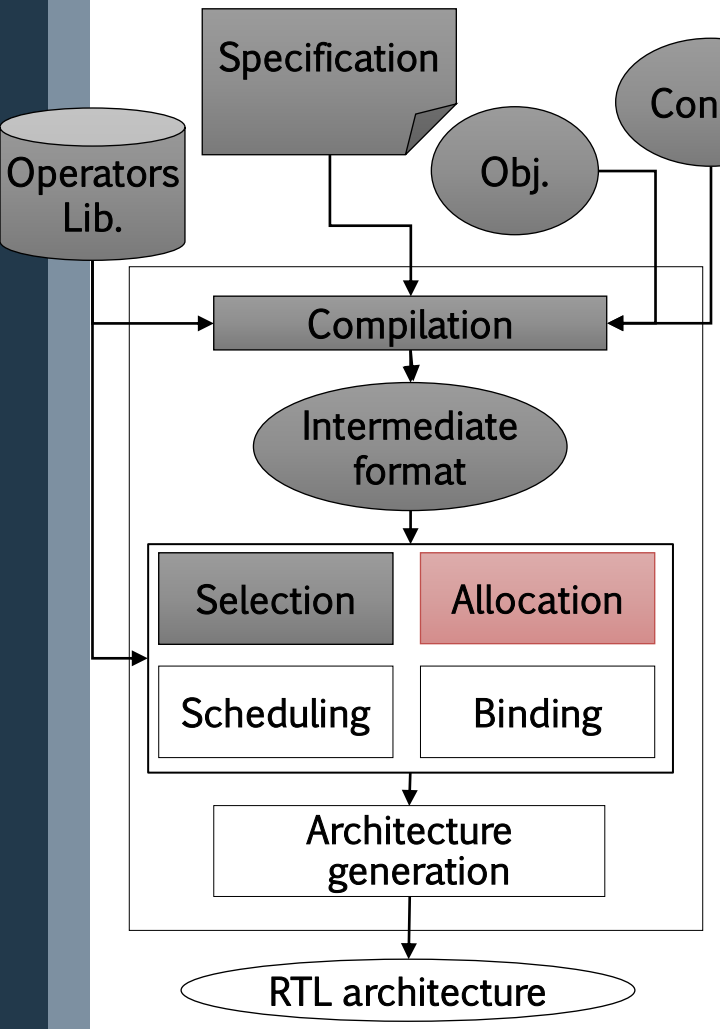
Specification

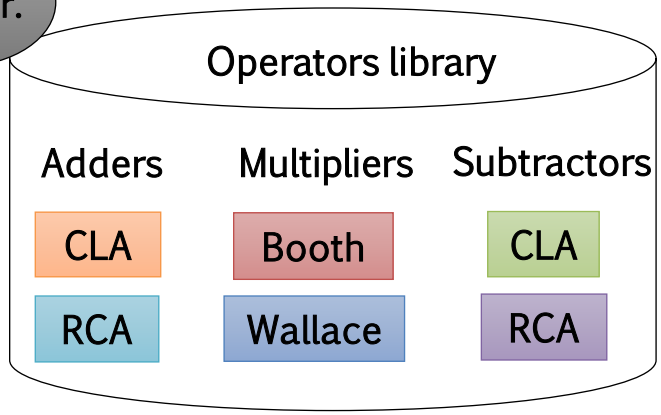
$O = ((n_{01}+n_{02})*n_{12})-(n_{21}+n_{22})$

Intermediate representation



HLS steps: Allocation

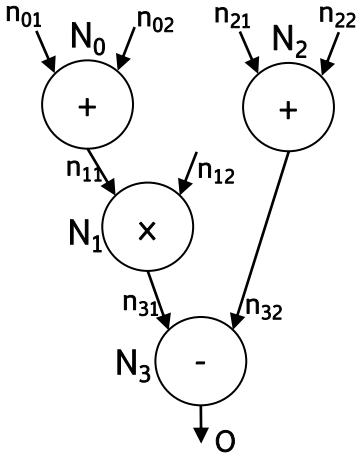




Specification

$O = ((n_{01}+n_{02})*n_{12})-(n_{21}+n_{22})$

Intermediate representation

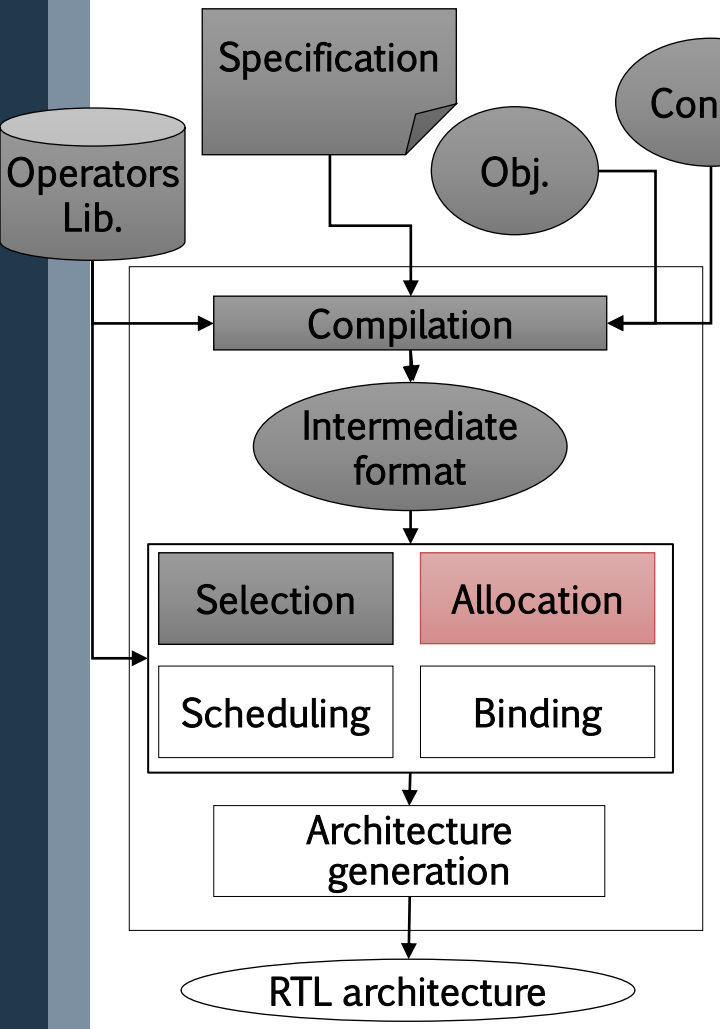


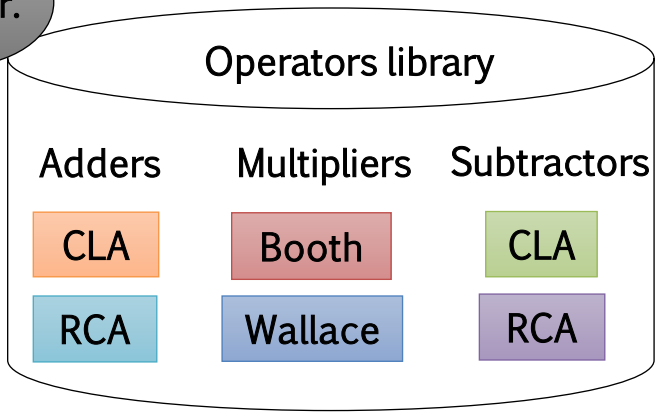
Booth

RCA

RCA

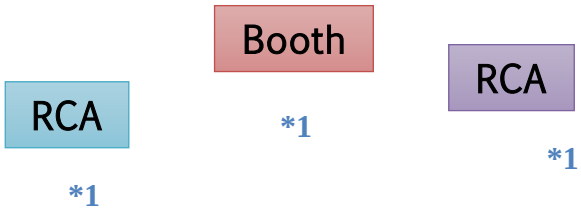
HLS steps: Allocation



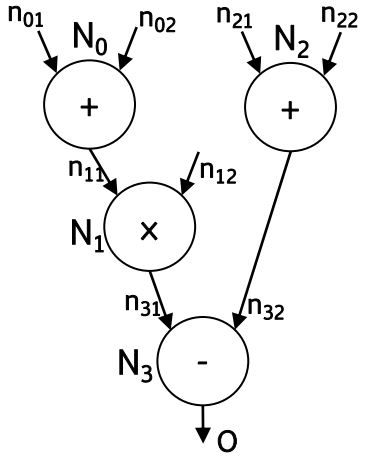


Specification

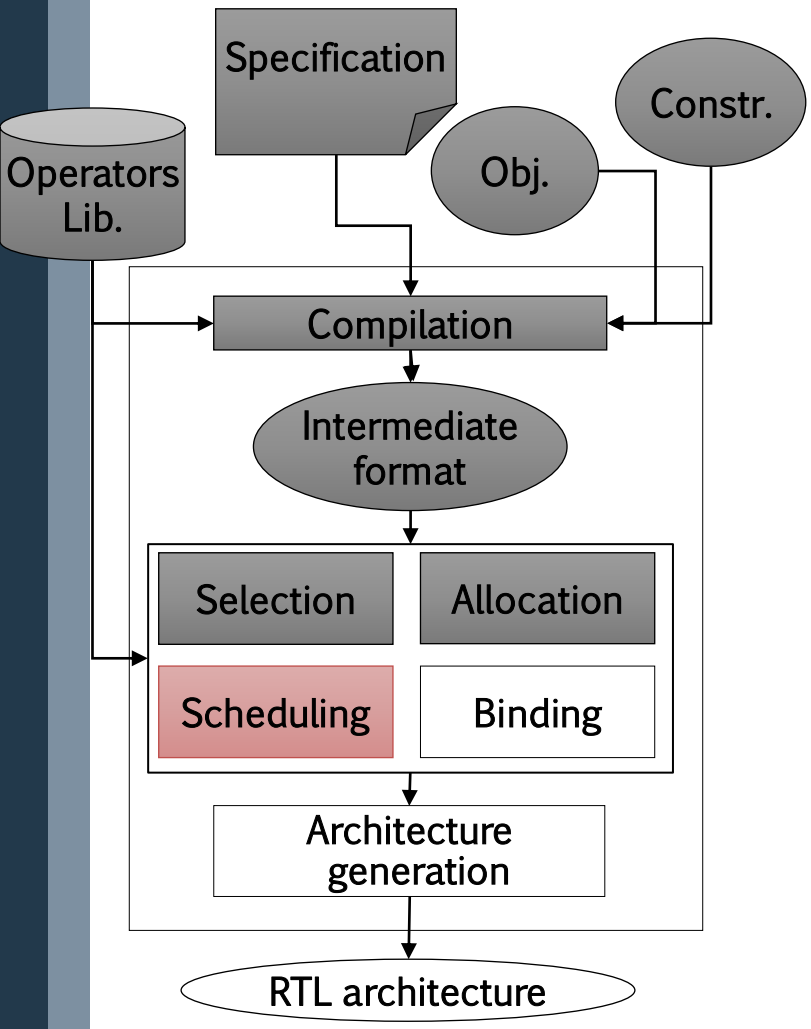
$O = ((n_{01}+n_{02})*n_{12})-(n_{21}+n_{22})$



Intermediate representation



HLS steps: Scheduling



RCA

*1

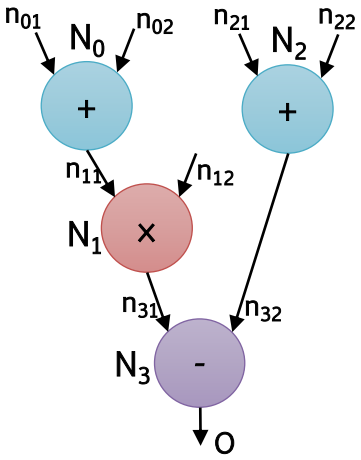
Booth

*1

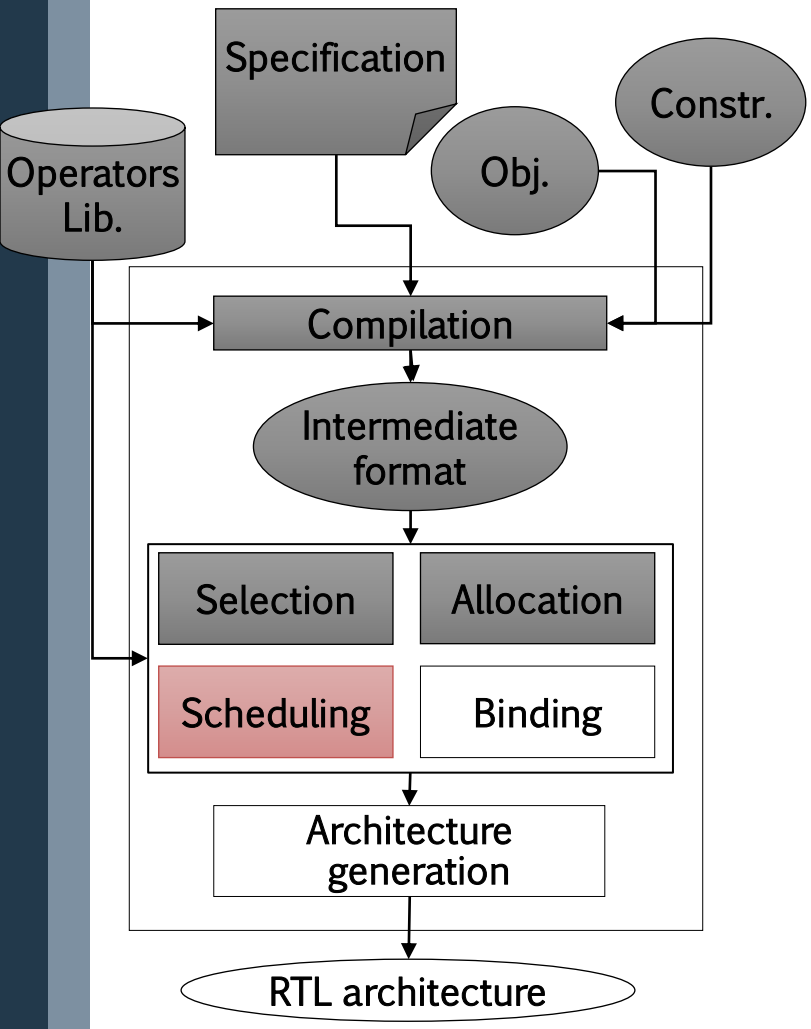
RCA

*1

Intermediate representation



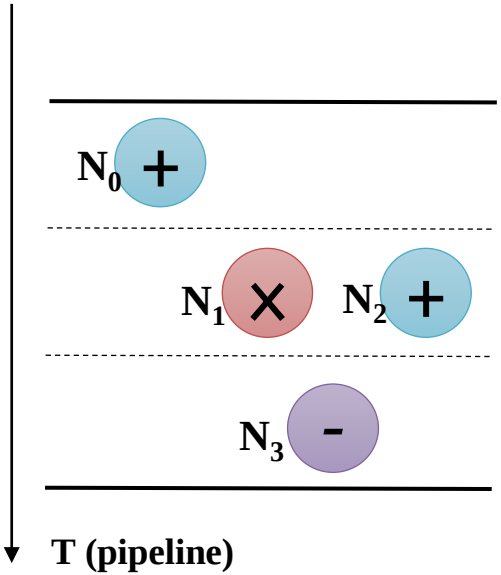
HLS steps: Scheduling



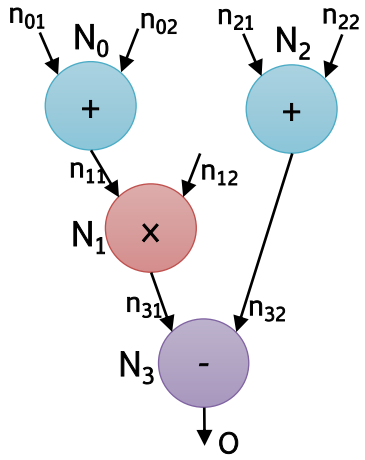
RCA *1

Booth *1

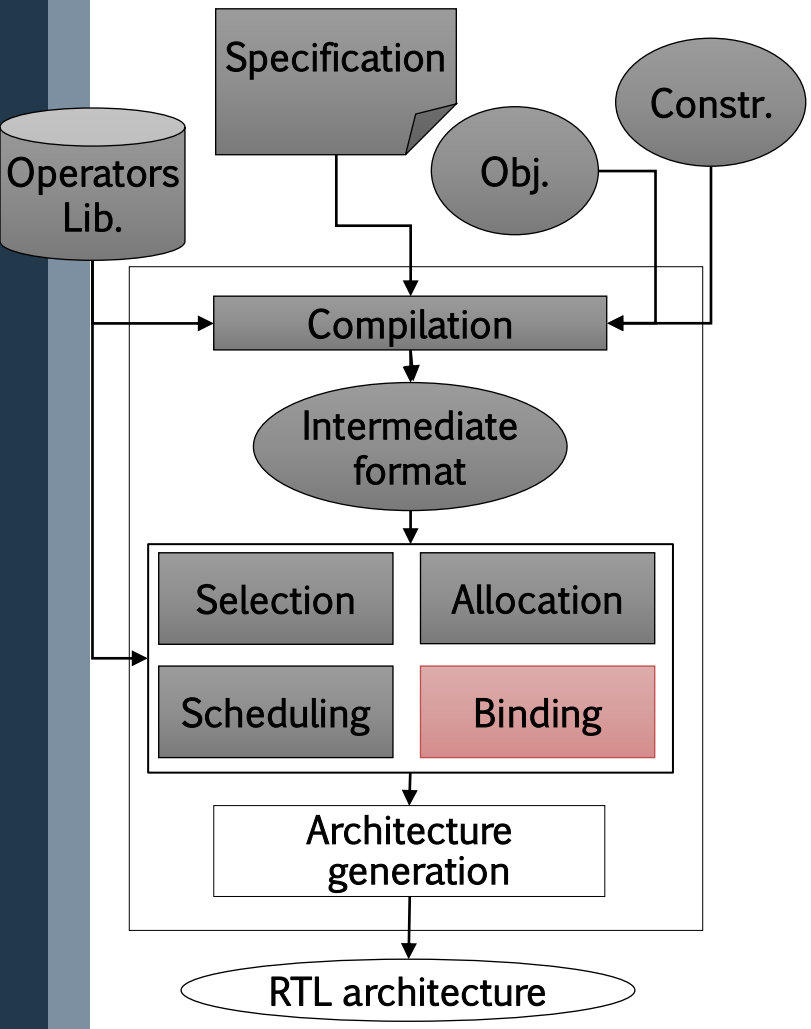
RCA *1



Intermediate representation



HLS steps: Binding

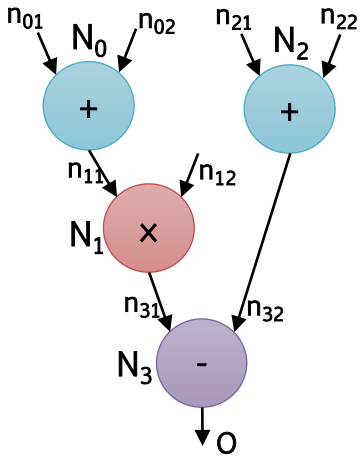


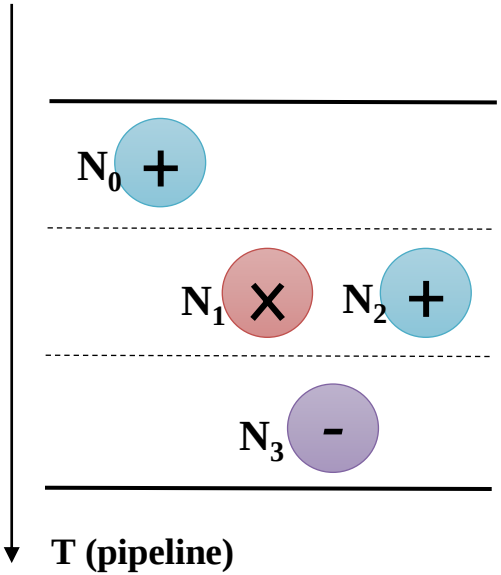
RCA *1

Booth *1

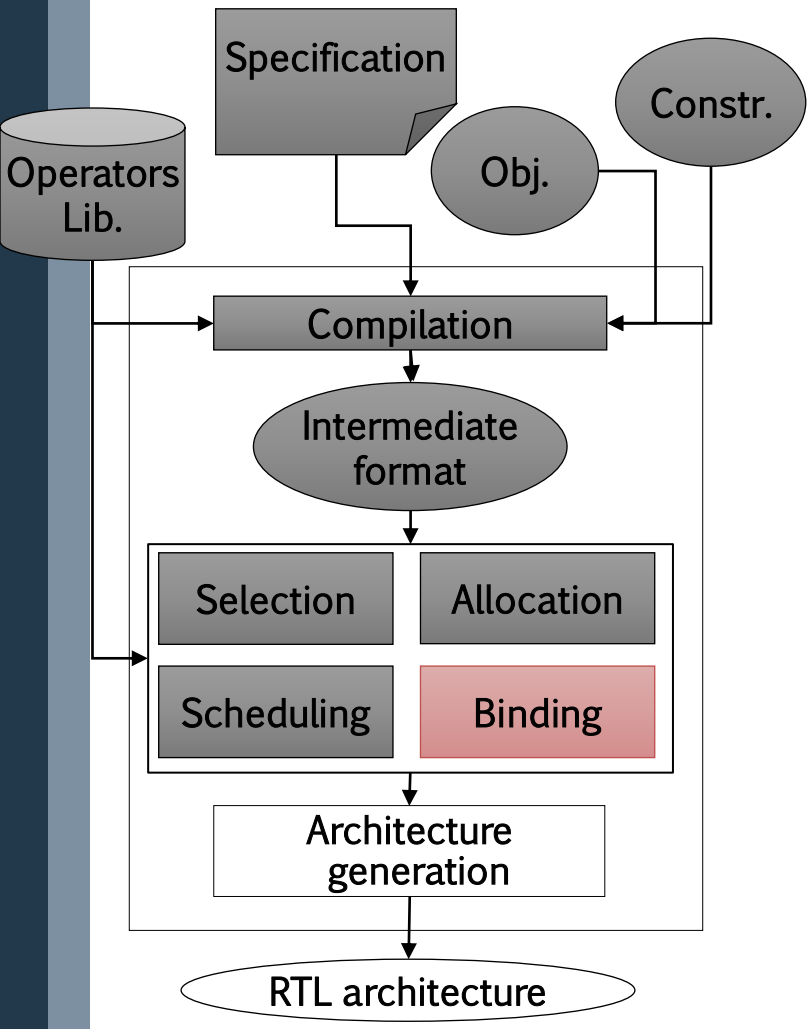
RCA *1

Intermediate representation





HLS steps: Binding

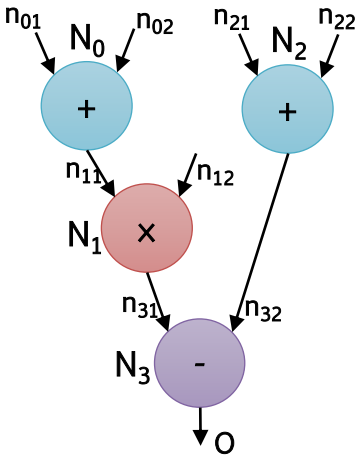


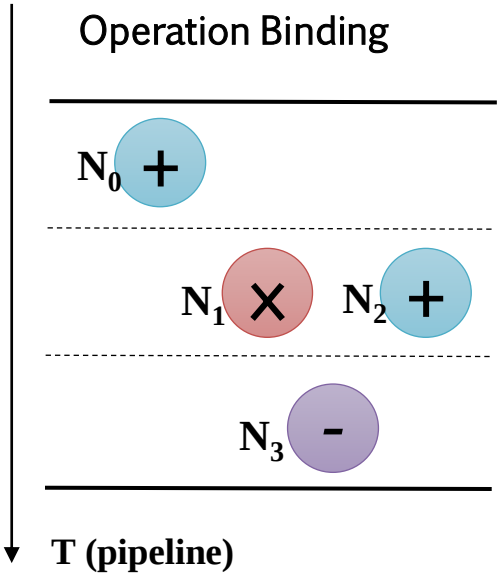
RCA *1

Booth *1

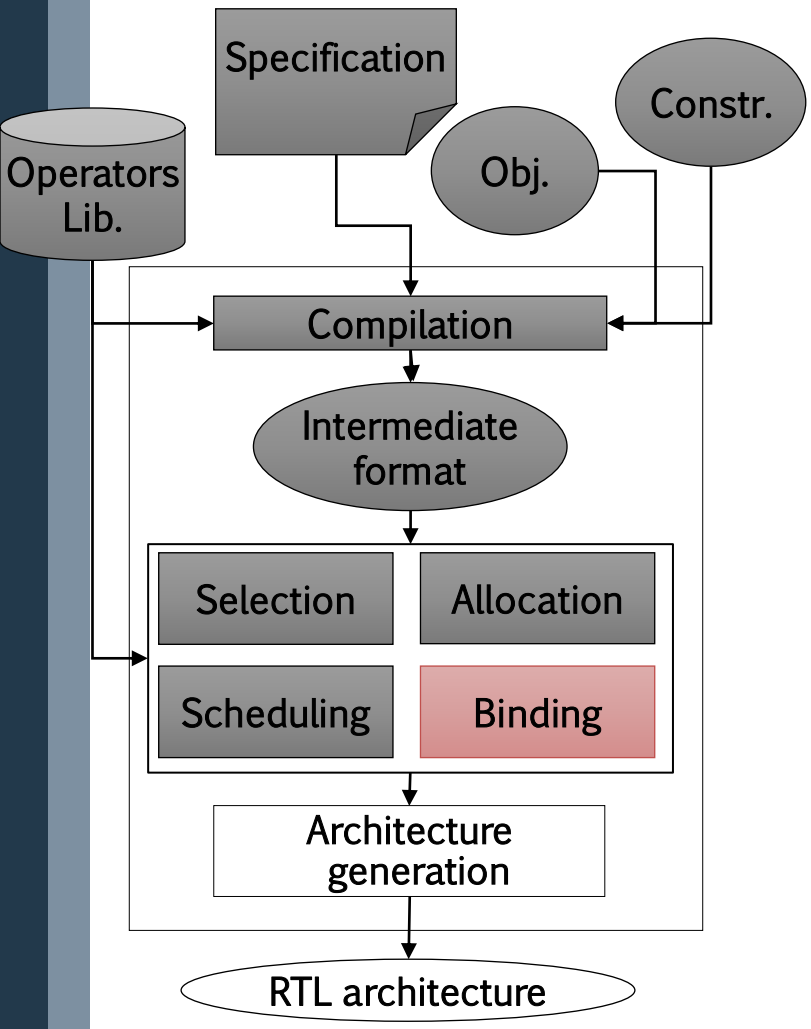
RCA *1

Intermediate representation





HLS steps: Binding

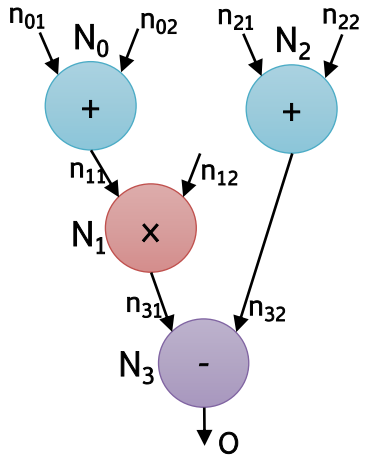


RCA *1

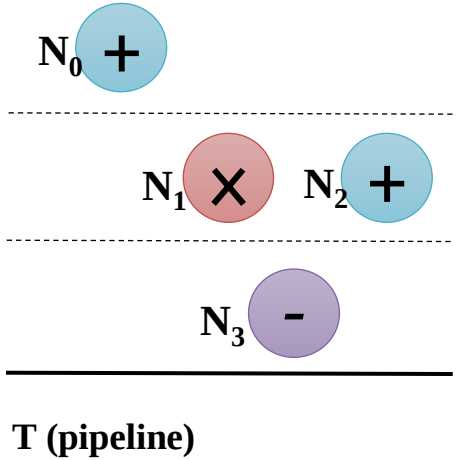
Booth *1

RCA *1

Intermediate representation



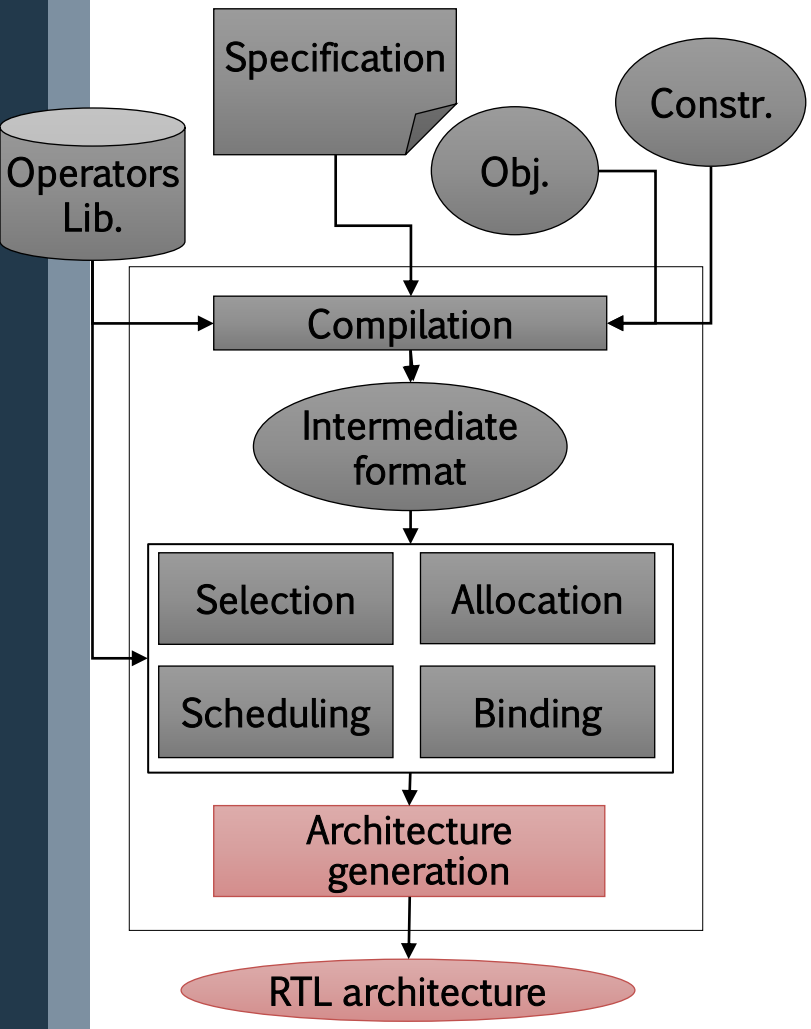
Operation Binding

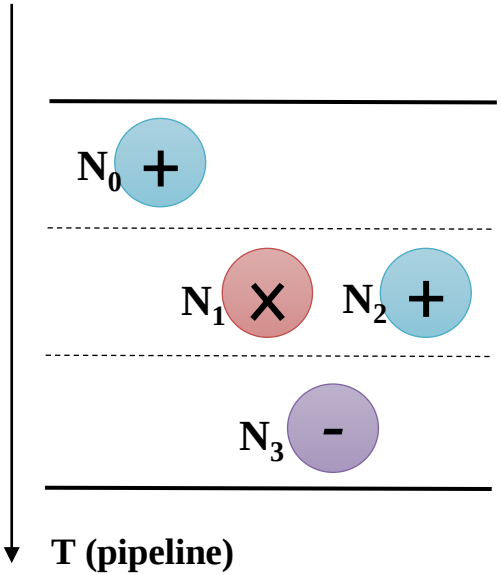


Data Binding

n_{01}	→	R_1
n_{02}	→	R_2
n_{21}, n_{11}	→	R_3
n_{22}, n_{12}	→	R_4
n_{31}	→	R_5
n_{32}	→	R_6

HLS steps: Binding





Data Binding

$n_{01} \rightarrow R_1$

$n_{02} \rightarrow R_2$

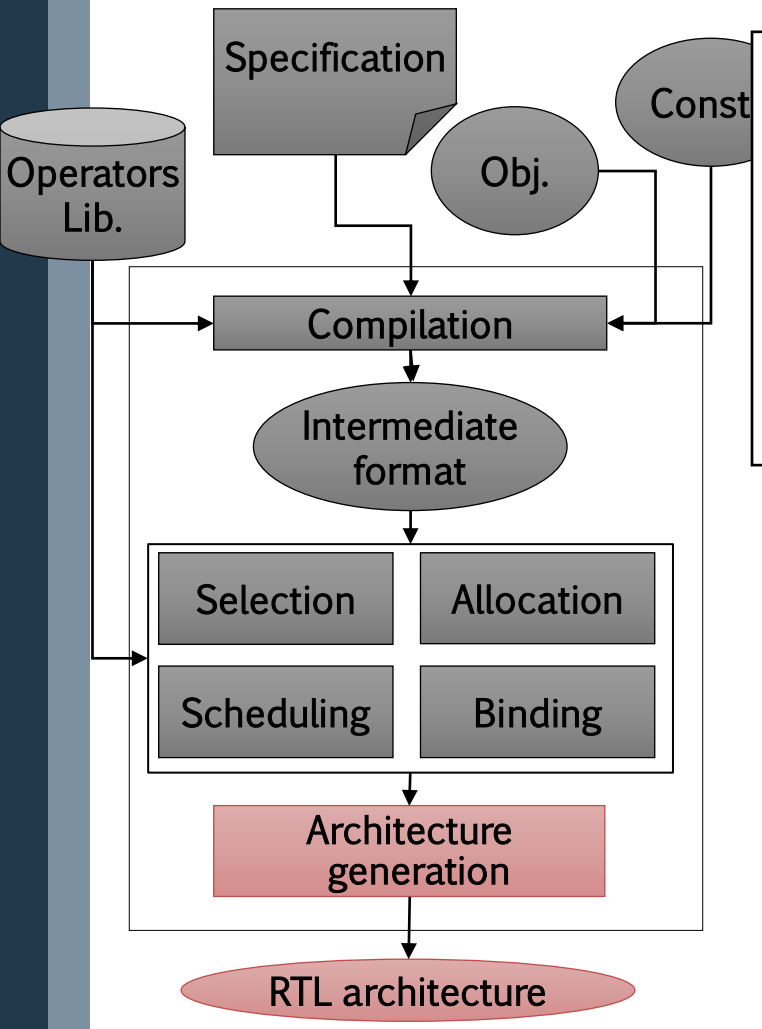
$n_{21}, n_{11} \rightarrow R_3$

$n_{22}, n_{12} \rightarrow R_4$

$n_{31} \rightarrow R_5$

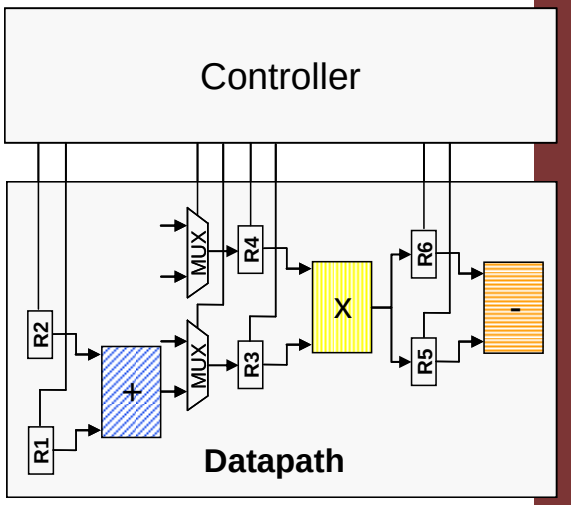
$n_{32} \rightarrow R_6$

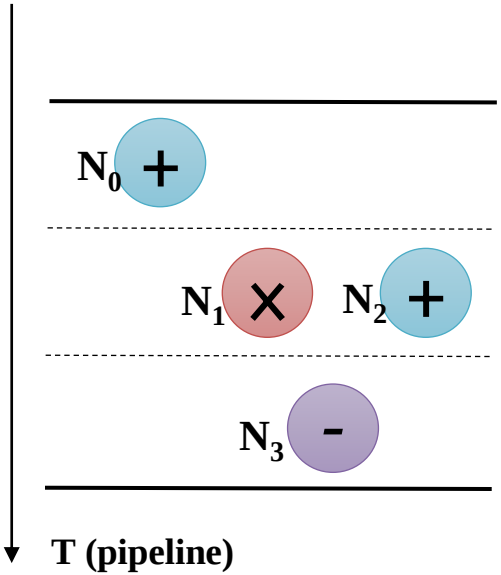
HLS steps: Binding



Controller
- FSM controller
- Programmable controller

Datapath components
- Storage components
- Functional units
- Connection components





Data Binding

n_{01}	→	R_1
n_{02}	→	R_2
n_{21}, n_{11}	→	R_3
n_{22}, n_{12}	→	R_4
n_{31}	→	R_5
n_{32}	→	R_6

Xilinx's Vivado SDK

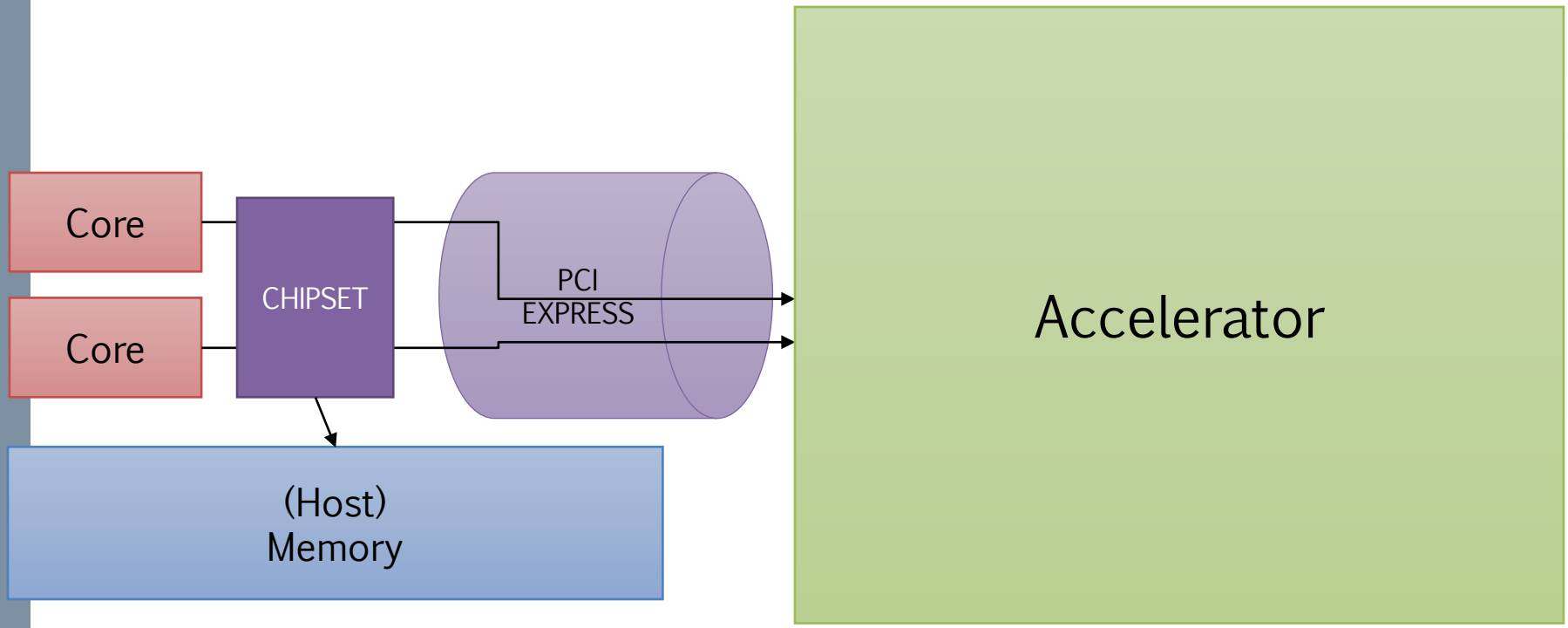
The FPGA development tool

- Starting from C or RTL…
- …generates and deploys the IP on the FPGA
- ..as well as SW artifacts to interact with them (drivers)

Heterogeneous systems

Host-accelerator model

- Multi-core General purpose host
  - The "traditional" core
- Coupled with a co-processor/accelerator



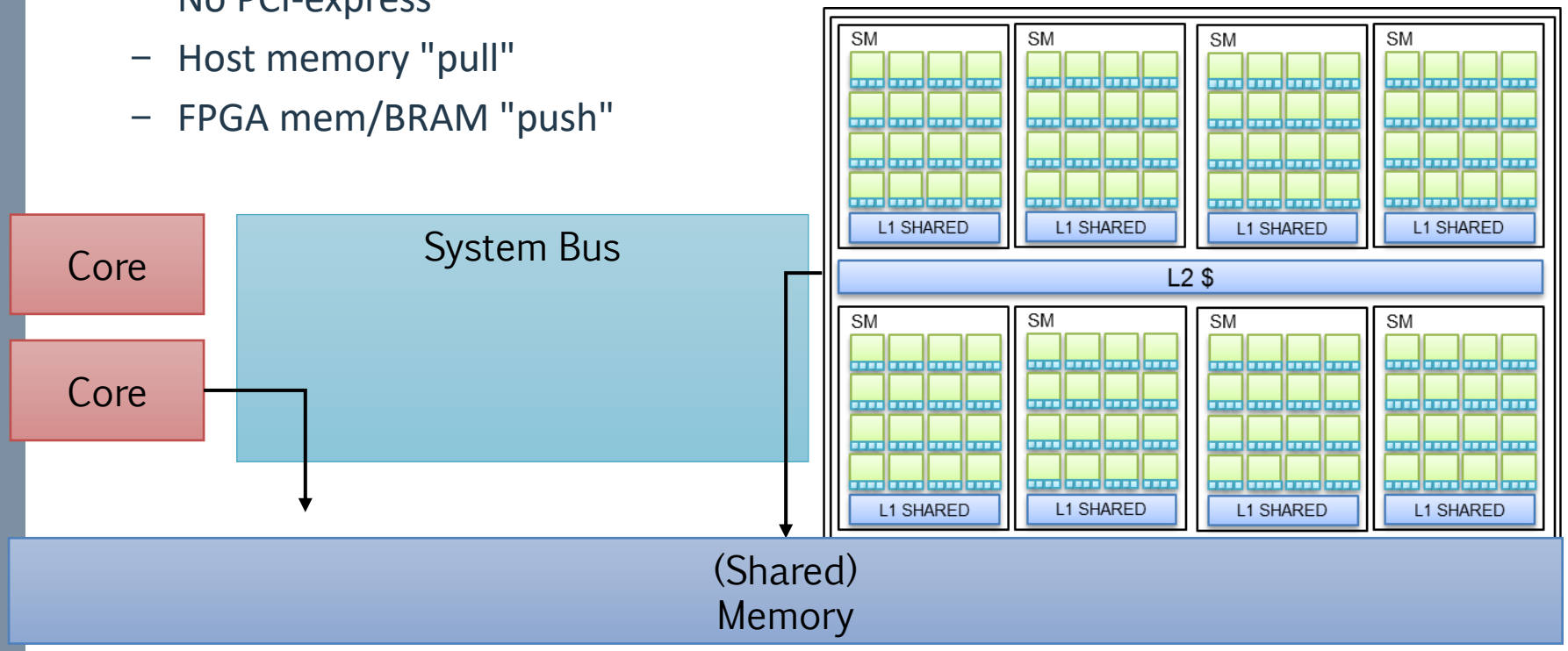
Do you remember (i)GPGPUs?

Integrated GP-GPUs for embedded platforms

› Host + accelerator model

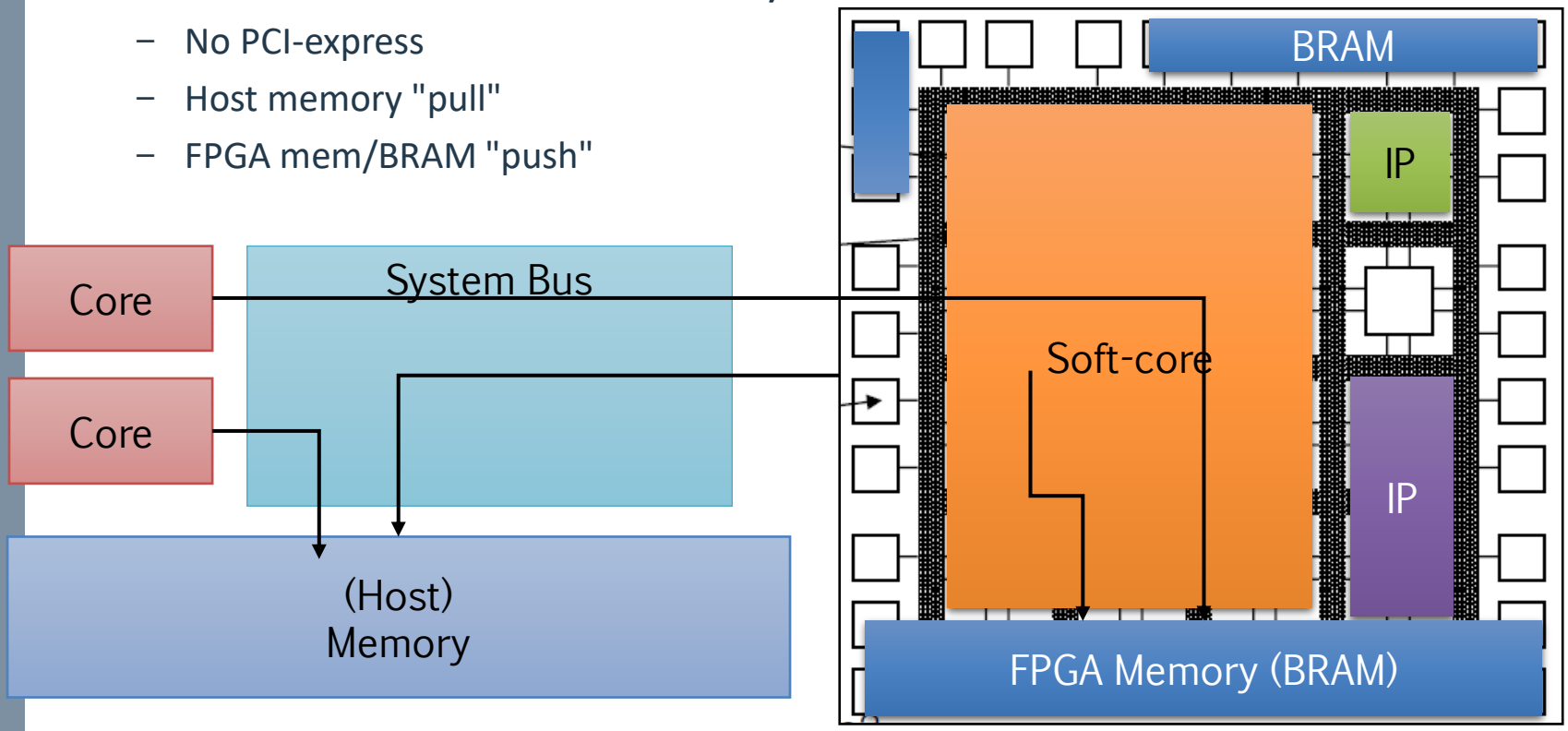
› Communicate via shared memory

- No PCI-express
- Host memory "pull"
- FPGA mem/BRAM "push"



FPGA-based accelerators

- Can create hundreds of small HW accelerators (de/crypt, de/coders)
- Can even create a single core (as co-processor)
  - Soft-cores
- Communicate via shared memory
  - No PCI-express
  - Host memory "pull"
  - FPGA mem/BRAM "push"



Example: Xilinx Zynq

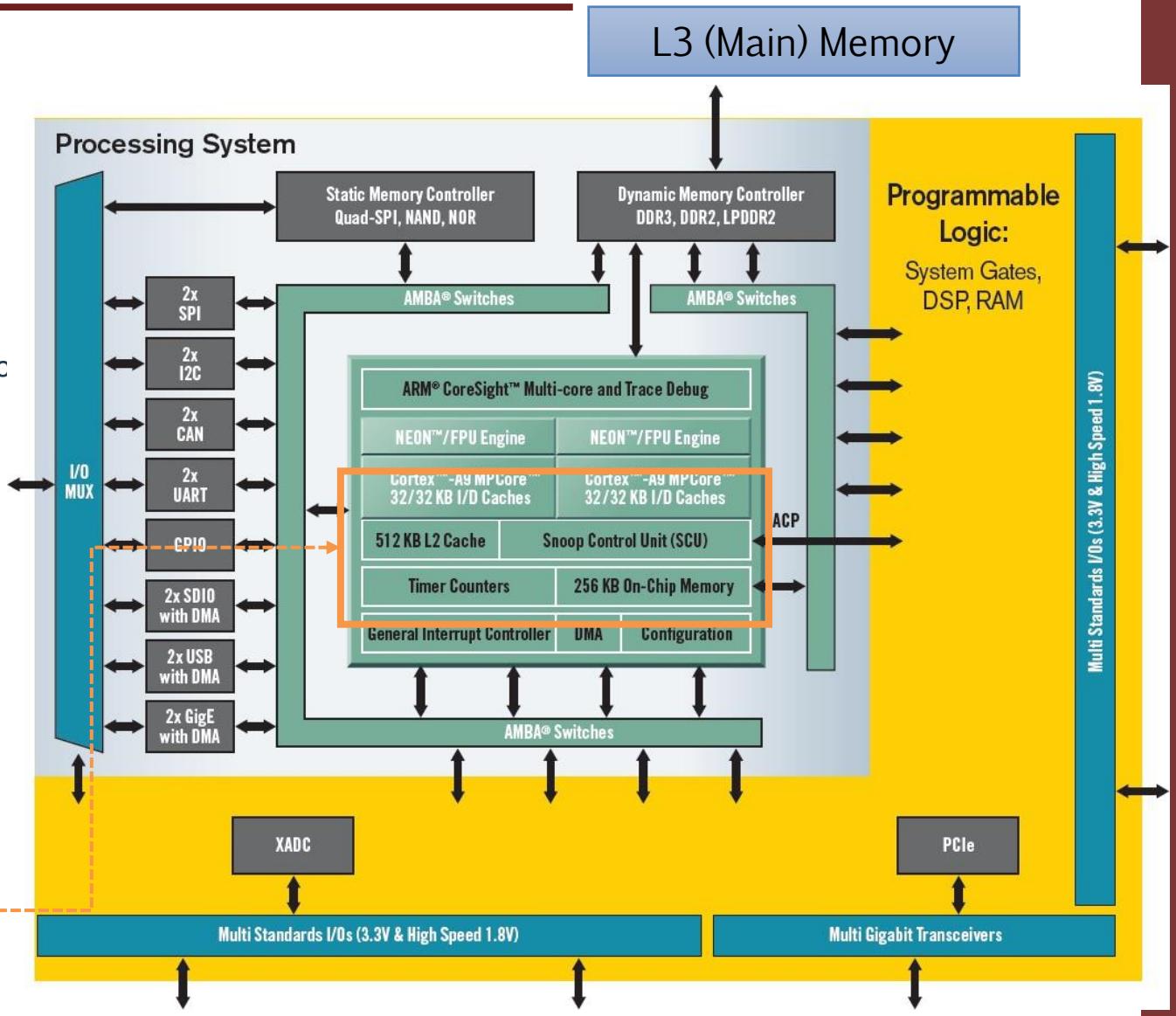
- Dual-core ARM host
- Programmable Logic

Shared memory

- Host caches
- Host SPM (On-chip Memc
- External DDR banks

Connectivity

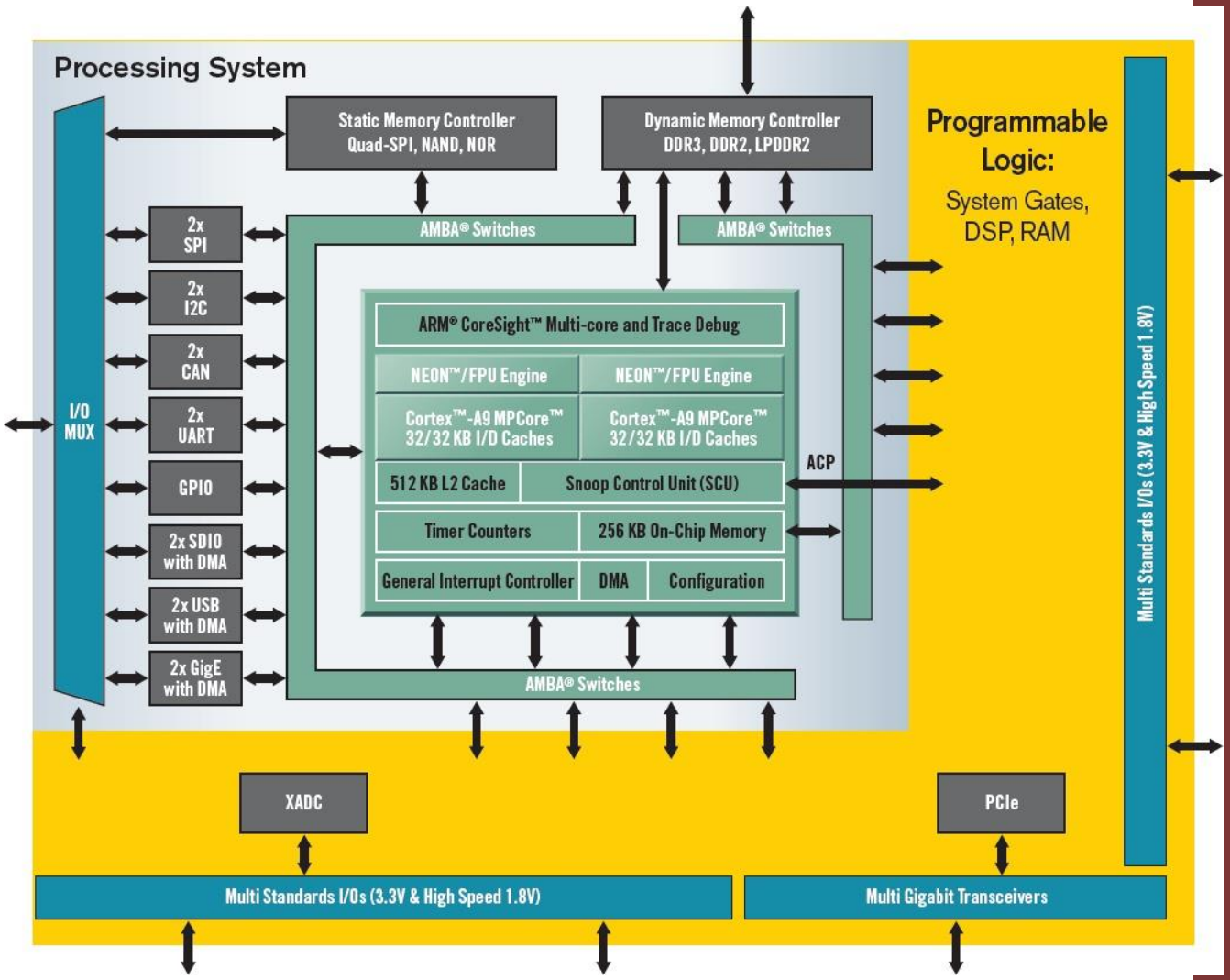
- AMBA AXI connector
- Highly scalable
- Cache coherency port ACP



Xilinx FPGA SoCs (NON ARGOMENTO D'ESAME PER A.A. 2020/2021)

Xilinx Zynq-7000

- Dual-core ARM host
- Programmable Logic
- Shared memory



Xilinx Zynq-7000

FAMILY	PART	Logic Cells (K)	Block RAM (Mb)	DSP Slices	Maximum I/O Pins	Maximum Transceiver (	Video Code Unit (VCU)
ZYNQ-7000							
	Z-7010	28	2,1	80	100	-	-
	Z-7015	74	3,3	160	150	4	-
	Z-7020	85	4,9	220	200	-	-
	Z-7030	125	9,3	400	250	4	-
	Z-7035	275	17,6	900	362	16	-
	Z-7045	350	19,1	900	362	16	-
	Z-7100	444	26,5	2020	400	16	-

Zedboard

- Complete development kit with Xilinx Zynq-7000 SoC
- Basic support for rapid prototyping and proof-of-concept development
- Small ☺

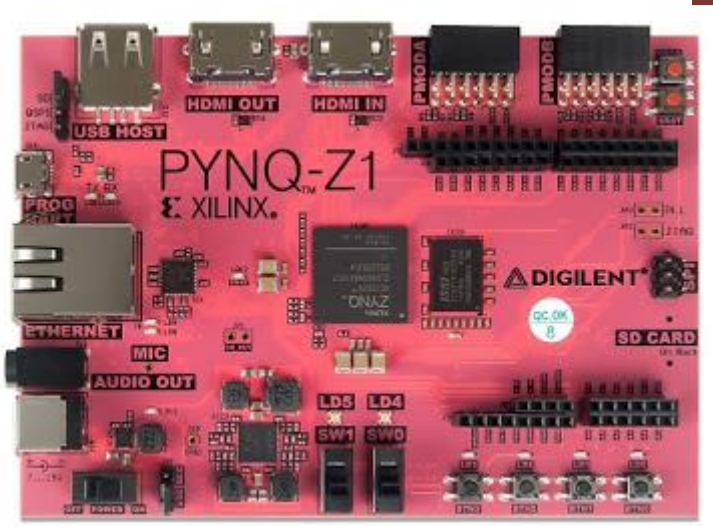
UltraZed

- System-On-Module (SOM)
- Based on the Ultrascale architecture: no host!
- Packages system memory, Ethernet, USB, and configuration memory needed for an embedded processing system

- UltraZed EG
- Ultrazed EV

Xilinx Pynq: Python for Zynq

- Open-source project from Xilinx for design
- Uses Python language and libraries
- Maximizes productivity



Processor: Dual-Core ARM® Cortex®-A9

FPGA: 1.3 M reconfigurable gates

Memory: 512MB DDR3 / FLASH

Storage: Micro SD card slot

Video: HDMI In and HDMI Out

Audio: Mic in, Line Out

Network: 10/100/1000 Ethernet

Expansion: USB Host connected to ARM PS

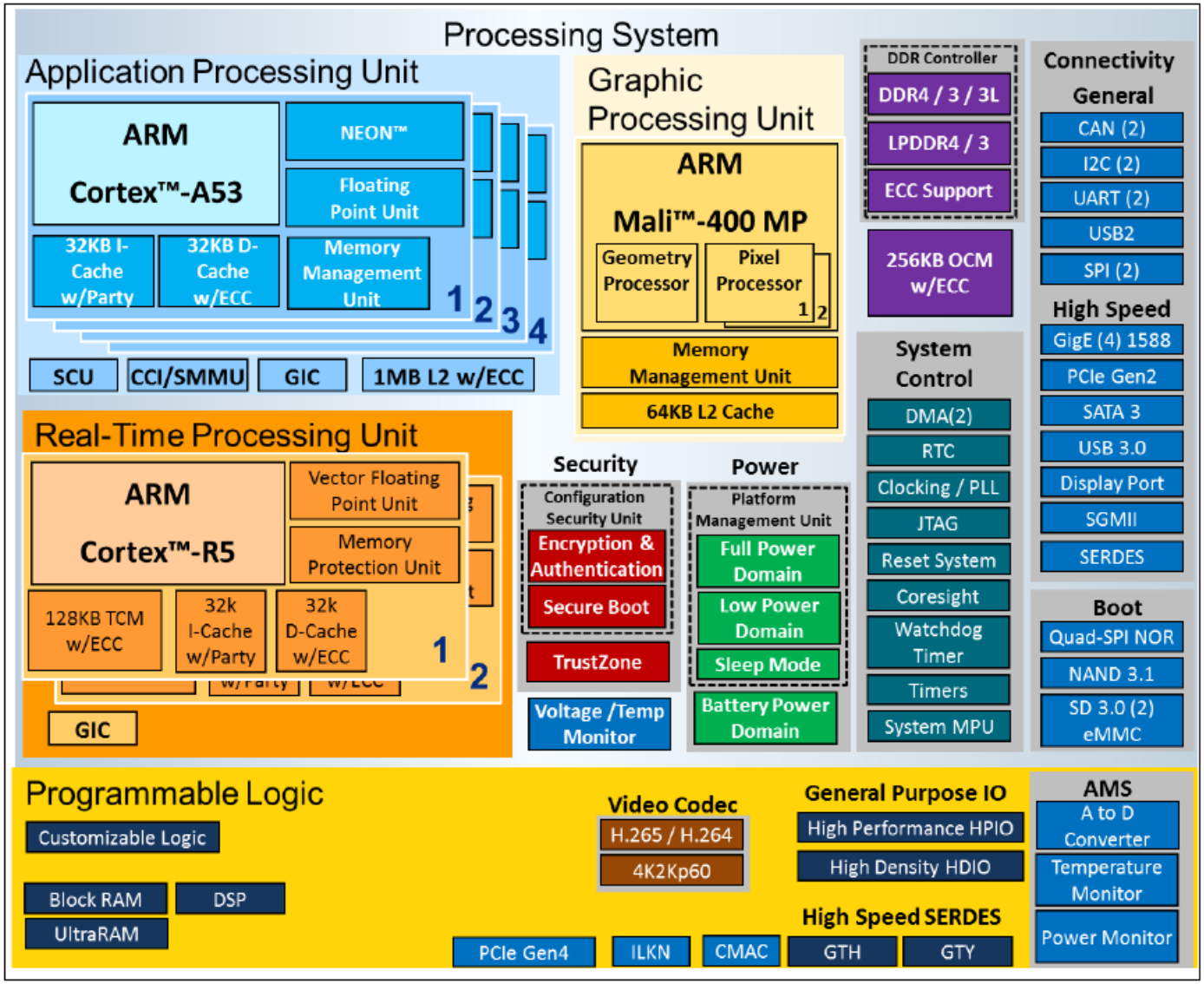
Interfaces: 1x Arduino Header, 2x Pmod (49 GPIO)

GPIO: 16 GPIO (65 in total with Arduino and Pmods)

Other I/O: 6x User LEDs, 4x Pushbuttons, 2x Switches

Dimensions: 3.44” x 4.81” (87mm x 122mm)

Xilinx Zynq Ultrascale+



Xilinx Zynq Ultrascale portfolio

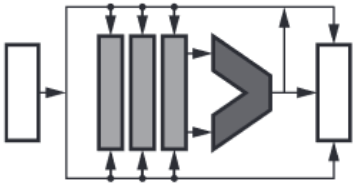
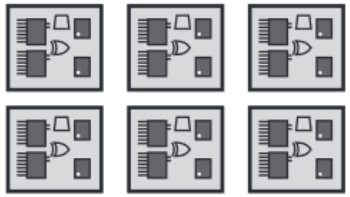
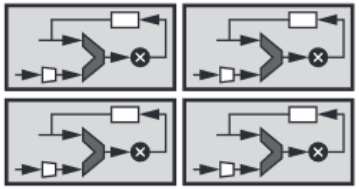
- Zynq UltraScale+ CG
  - Dual-core Cortex-A53 and a dual-core Cortex-R5 real-time processor
  - Programmable logic
  - Optimized for industrial motor control, sensor fusion, and industrial IoT applications
- Zynq UltraScale+ EG
  - Quad-core Cortex-A53 and dual-core Cortex-R5 real-time processors
  - Mali-400 MP2 graphics processing unit + programmable logic
  - Next-generation wired and 5G wireless infrastructure, cloud computing, and Aerospace and Defense applications
- Zynq UltraScale+ EV
  - EG platform + integrated H.264 / H.265 video codec
  - Multimedia, automotive ADAS, surveillance, and other embedded vision applications

Xilinx Zynq Ultrascale+

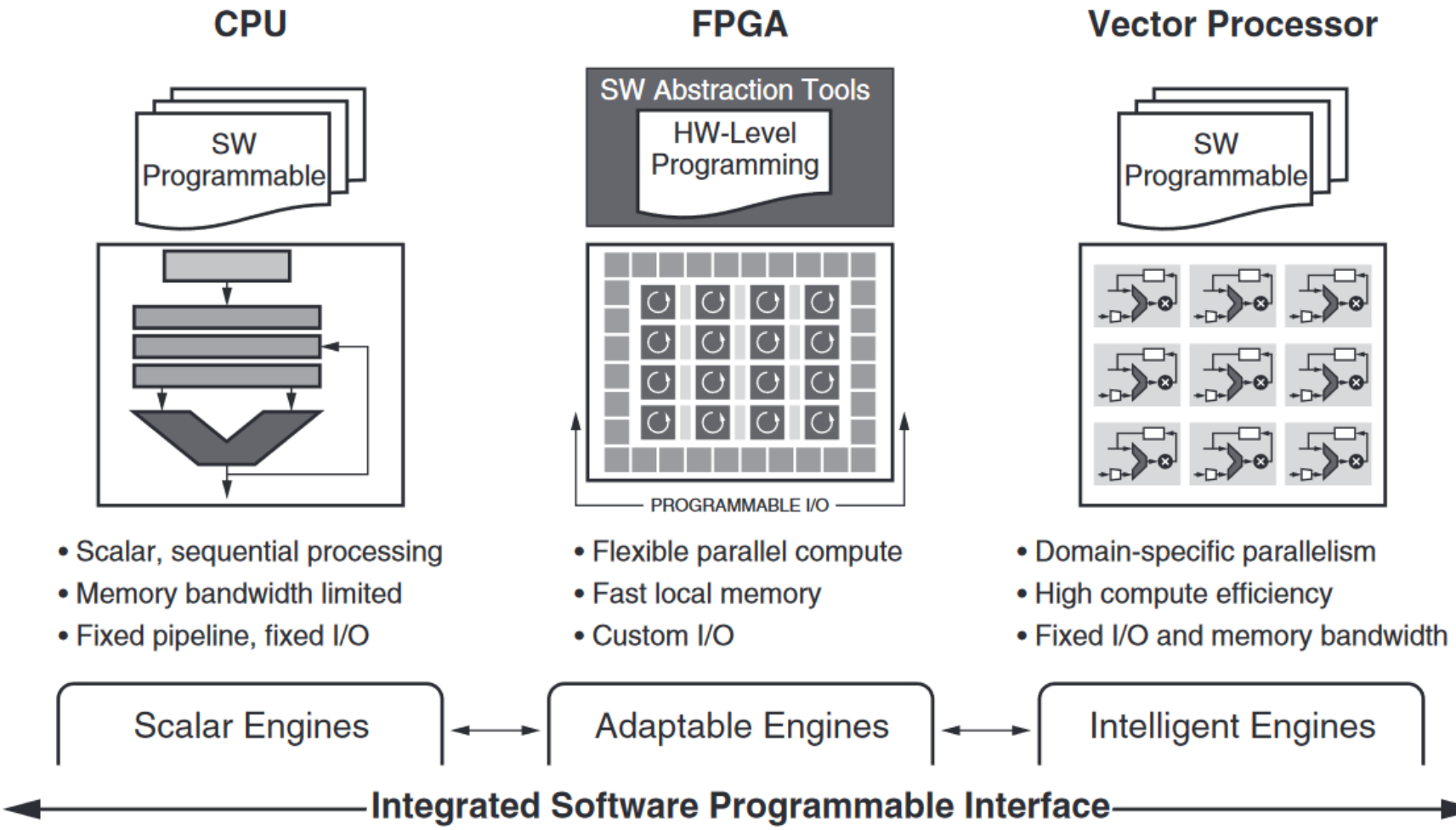
FAMILY	PART	Logic Cells (K)	Block RAM (Mb)	DSP Slices	Maximum I/O Pins	Maximum Transceiver (	Video Code Unit (VCU)
ZYNQ UltraScale+ CG							
	ZU2CG	103	5,3	240	252	-	-
	ZU3CG	154	7,6	360	252	-	-
	ZU4CG	192	18,5	728	252	-	-
	ZU5CG	256	23,1	1248	252	-	-
	ZU6CG	469	25,1	1973	328	-	-
	ZU7CG	504	38	1728	464	-	-
	ZU9CG	600	32,1	2520	328	-	-
ZYNQ UltraScale+ EG							
	ZU2EG	103	5,3	240	252	-	-
	ZU3EG	154	7,6	360	252	-	-
	ZU4EG	192	18,5	728	252	-	-
	ZU5EG	256	23,1	1248	252	-	-
	ZU6EG	469	25,1	1973	328	-	-
	ZU7EG	504	38	1728	464	-	-
	ZU9EG	600	32,1	2520	328	-	-
	ZU11EG	653	43,6	2928	512	-	-
	ZU15EG	747	57,7	3528	328	-	-
	ZU17EG	926	56,7	1590	668	-	-
	ZU19EG	1143	70,6	1968	668	-	-
ZYNQ UltraScale+ EV							
	ZU4EV	192	18,5	728	252	-	1
	ZU5EV	256	23,1	1248	252	-	1
	ZU7EV	504	38	1728	464	-	1

Next generation: Versal

› Adaptive Compute Acceleration Platform (ACAP)

Scalar Processing	Adaptable Hardware	Vector Processing *(e.g., GPU, DSP)*
		
Complex Algorithms and Decision Making	Processing of Irregular Data Structures *Genomic Sequencing*	Domain-specific Parallelism
	Latency Critical Workloads *Real-Time Control*	Signal Processing *Complex Math, Convolutions*
	Sensor Fusion *Pre-processing, Programmable I/O*	Video and Image Processing

Programming Versal

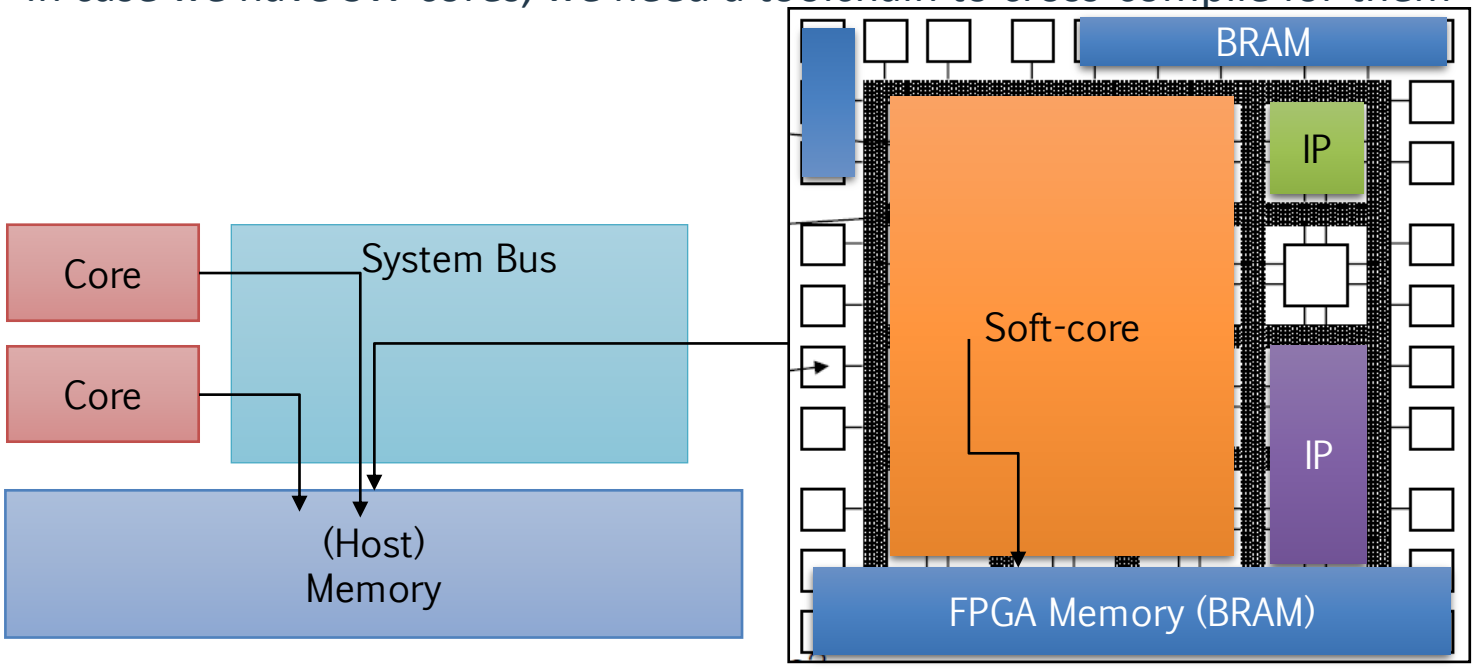


Programming heterogeneous systems

Heterogeneous programming

Besides a tool to generate the actual IPs, we need

- A way to efficiently offload (pre-compiled) bitcode on the FPGA
  - On-the-fly Dynamic Partial Rreconfiguration (DPR)
- Simple offloading subroutines to the newly created HW blocks
  - To increase productivity
- In case we have SW cores, we need a toolchain to cross-compile for them



1) custom/"by hand"/CAD

Code generated by logic synthesis tool

- Step 1 – generate the bitcode of the accelerator
  - Vivado HLS
- Step 2 – plug the accelerator in a design
  - Vivado
  - Include processing system (ARM host) + accelerator + IC + …
- Step 3 – generate the design
  - Bitcode ready to be installed of the IP
  - Architecture configuration files (memory maps…)
  - Software for host + drivers to communicate with the IP

2) offload-based programming

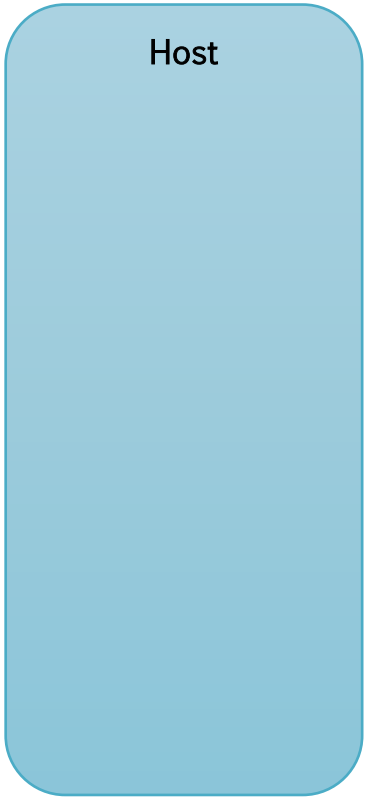
Offload-based programming models

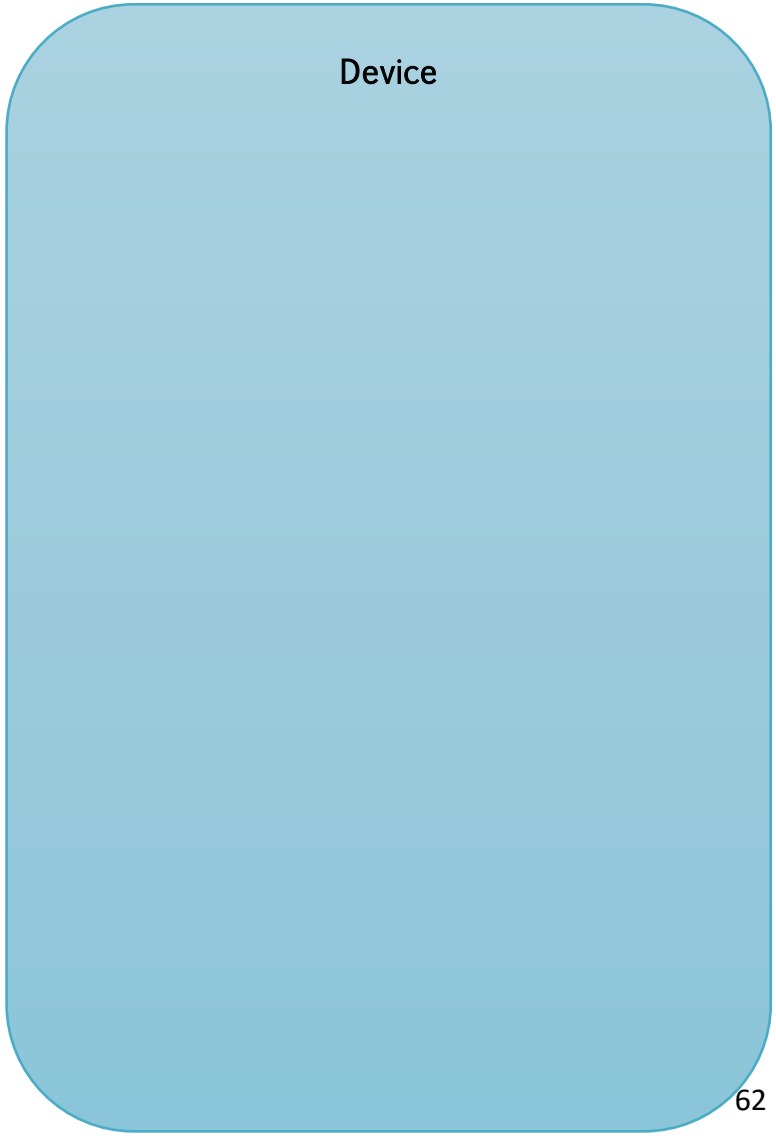
- CUDA (for NVIDIA GPUs)
- OpenCL (for "generic" accelerators)
- OpenMP 4.5

2) offload-based programming

Offload-based programming models

- CUDA (for NVIDIA GPUs)
- OpenCL (for "generic" accelerators)
- OpenMP 4.5

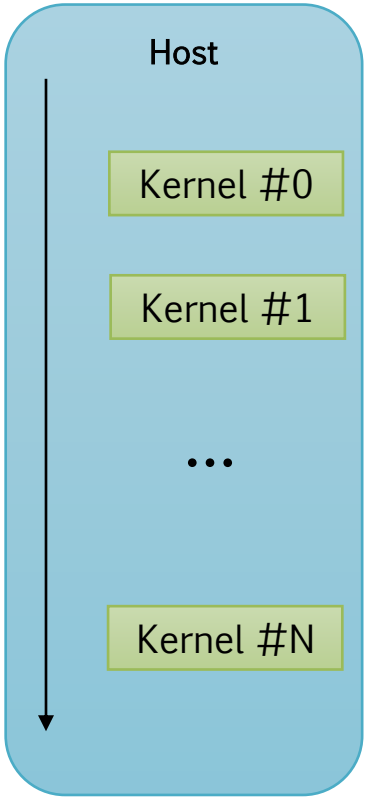


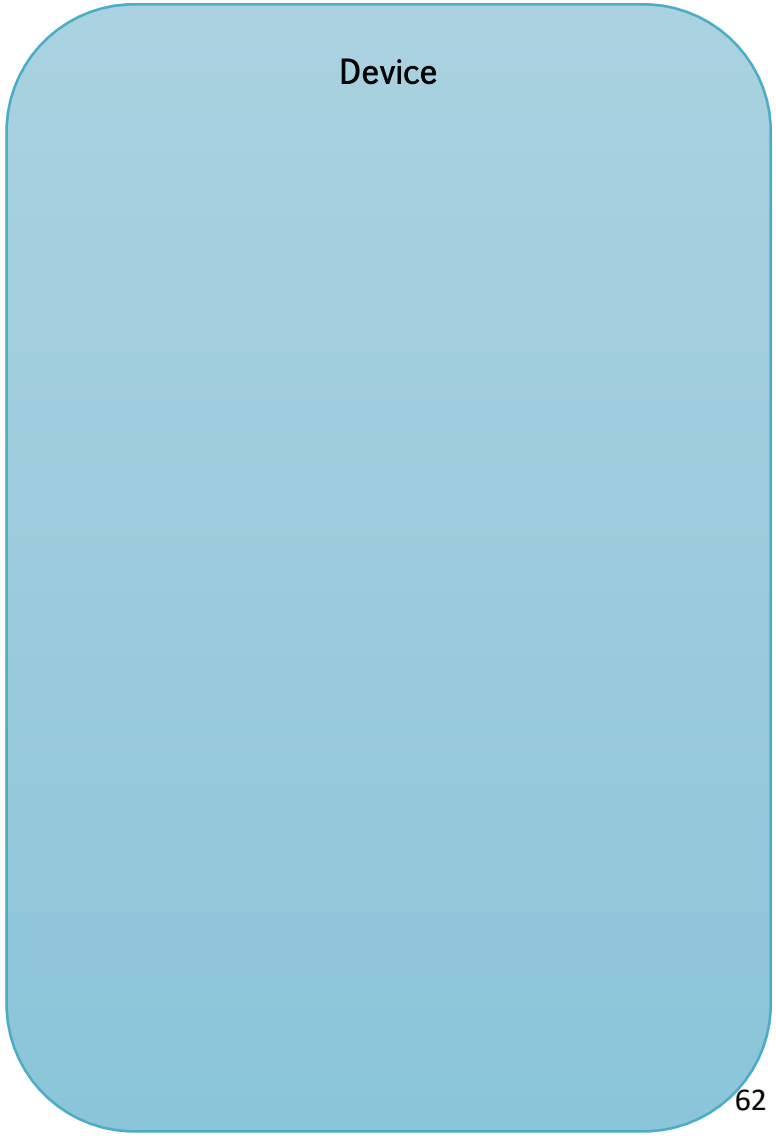


2) offload-based programming

Offload-based programming models

- CUDA (for NVIDIA GPUs)
- OpenCL (for "generic" accelerators)
- OpenMP 4.5

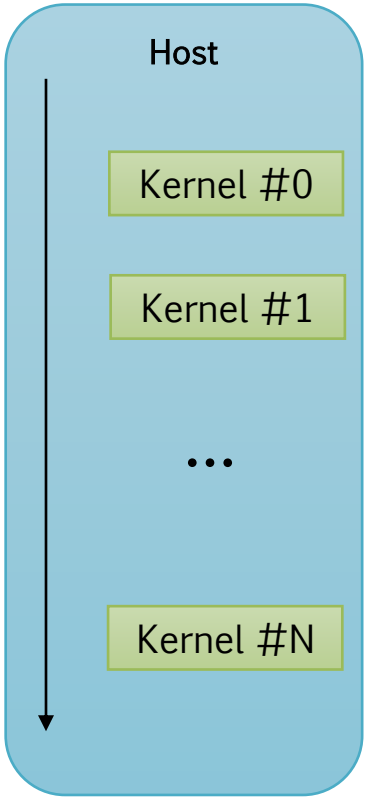


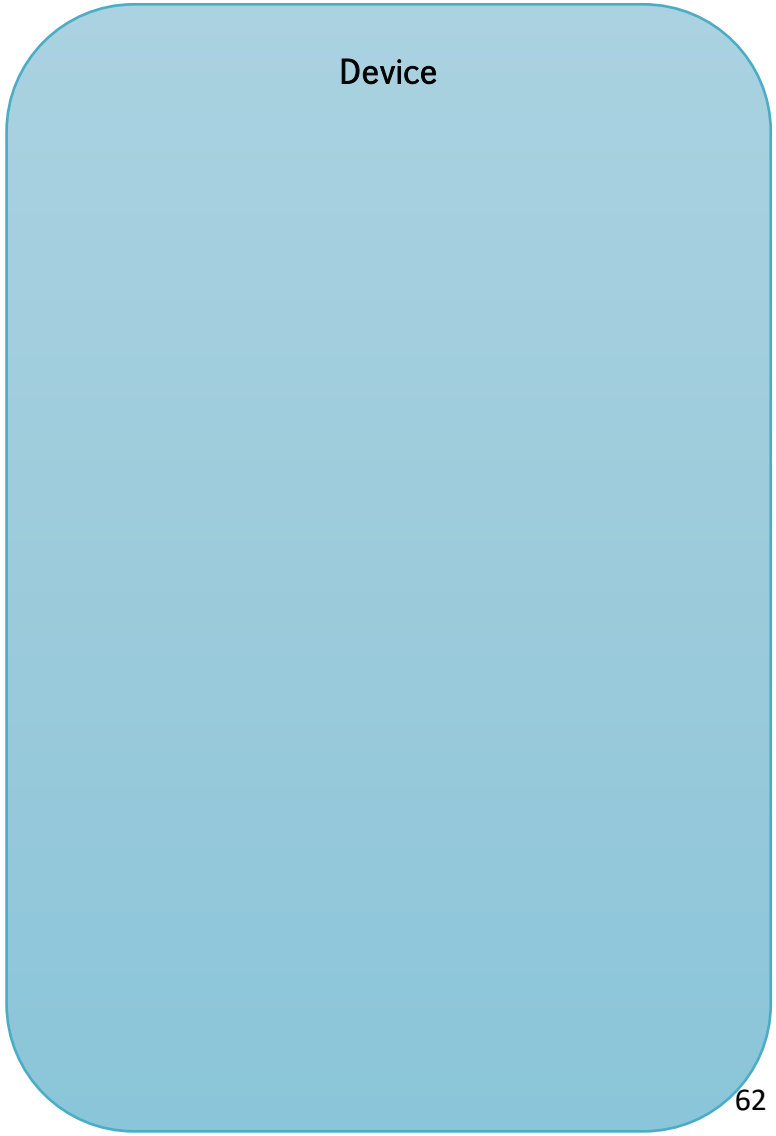


2) offload-based programming

Offload-based programming models

- CUDA (for NVIDIA GPUs)
- OpenCL (for "generic" accelerators)
- OpenMP 4.5

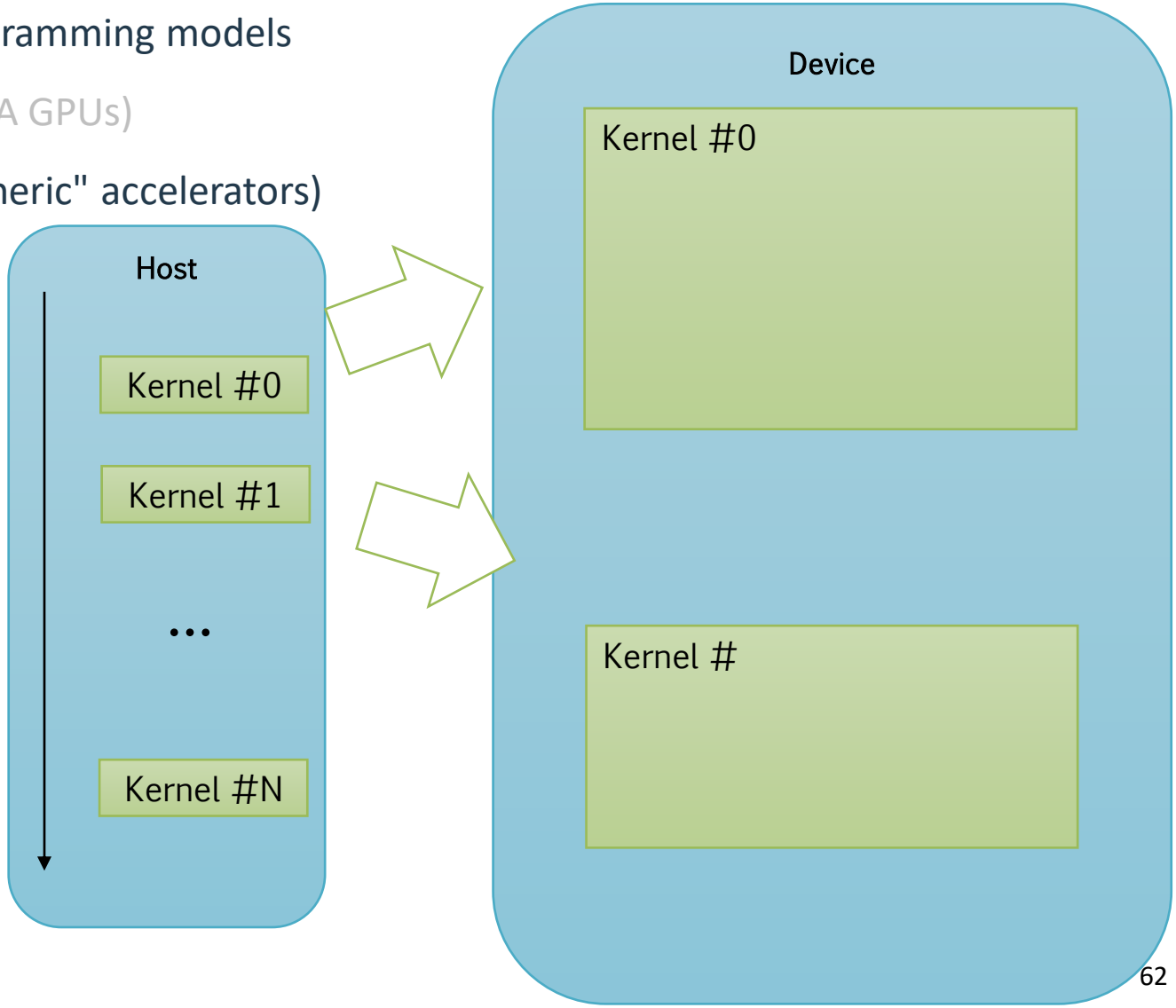




2) offload-based programming

Offload-based programming models

- CUDA (for NVIDIA GPUs)
- OpenCL (for "generic" accelerators)
- OpenMP 4.5

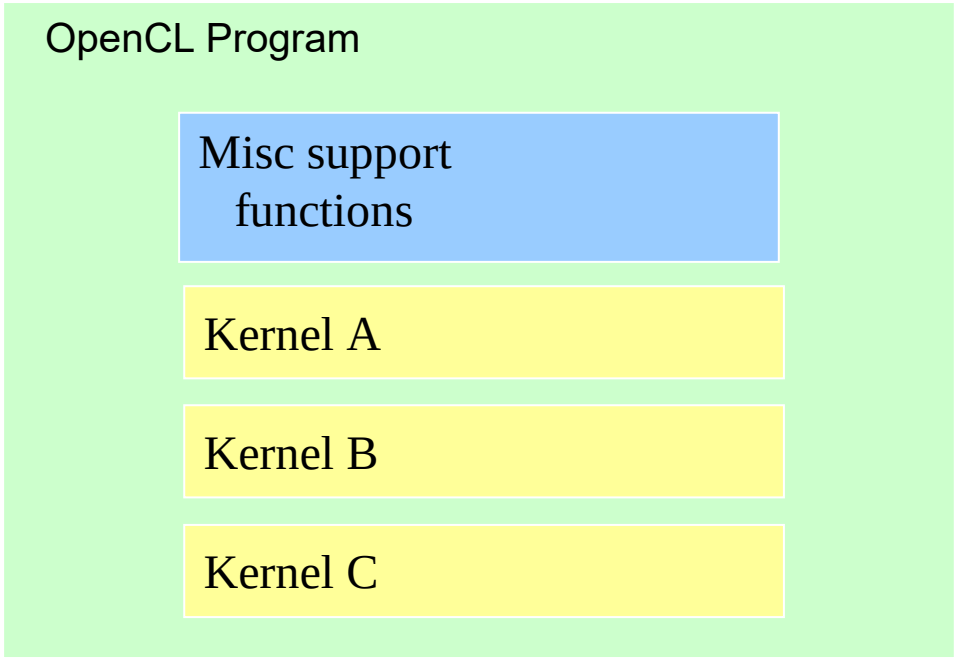


OpenCL

- OpenCL was initiated by Apple and maintained by the Khronos Group (also home of OpenGL) as an industry standard API
  - For cross-platform parallel programming in CPUs, GPUs, DSPs, FPGAs,…
- OpenCL host code is much more complex and tedious due to desire to maximize portability and to minimize burden on vendors

OpenCL program

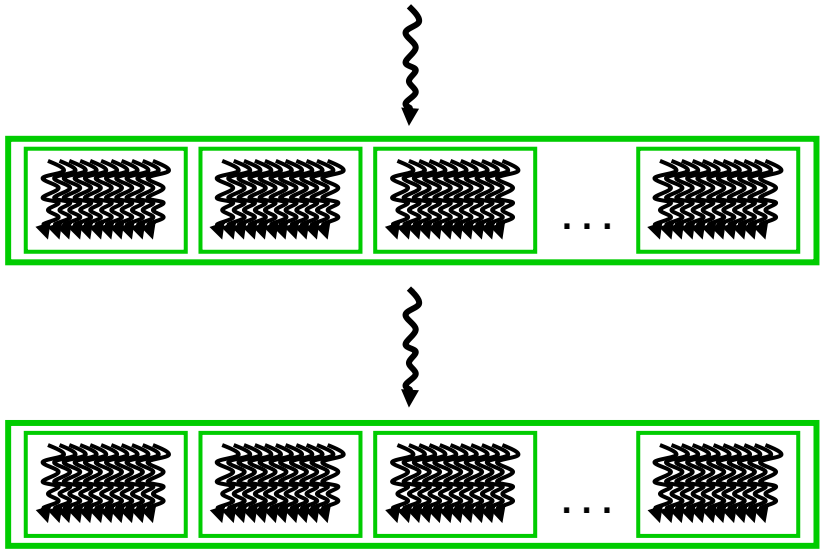
- An OpenCL “program” is a C program that contains one or more “kernels” and any supporting routines that run on a target device
- An OpenCL kernel is the basic unit of parallel code that can be executed on a target device
- In our case, an FPGA



OpenCL execution model

- Integrated host+device app C program
  - Serial or modestly parallel parts in host C code
  - Highly parallel parts in device SPMD kernel C code

- Queues of command/data transfer to be executed on the device



OpenCL kernels – software version

- Code that executes on target devices
- Kernel body is instantiated N times (data parallel) – work items
- Each OpenCL work item gets a unique index

- In the FPGA case, we use IP drivers instead of this

```
__kernel void  vadd(__global const float *a,
                    __global const float *b,
                    __global float *result)
{
    int id = get_global_id(0);
    result[id] = a[id] + b[id];
}
```

Host code – create exec ctx

```
cl_int clerr = CL_SUCCESS;
cl_context clctx = clCreateContextFromType(0, CL_DEVICE_TYPE_ALL, NULL, NULL,
&clerr);

size_t parmsz;
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, 0, NULL, &parmsz);

cl_device_id* cldevs = (cl_device_id *) malloc(parmsz);
clerr = clGetContextInfo(clctx, CL_CONTEXT_DEVICES, parmsz, cldevs, NULL);

cl_command_queue clcmdq = clCreateCommandQueue(clctx, cldevs[0], 0, &clerr);
```

Host code – create data buffers

```
float *h_A = …,   *h_B = …;
    // allocate device (GPU) memory
   cl_mem d_A, d_B, d_C;
   d_A = clCreateBuffer(clctx, CL_MEM_READ_ONLY |
          CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_A, NULL);
   d_B = clCreateBuffer(clctx, CL_MEM_READ_ONLY |
          CL_MEM_COPY_HOST_PTR, N *sizeof(float), h_B, NULL);
   d_C = clCreateBuffer(clctx, CL_MEM_WRITE_ONLY,
  N *sizeof(float), NULL, NULL);
```

Host code – device config setting

```
clkern=clCreateKernel(clpgm, "vadd", NULL);
  …
  clerr= clSetKernelArg(clkern, 0, sizeof(cl_mem),(void *)&d_A);
  clerr= clSetKernelArg(clkern, 1, sizeof(cl_mem),(void *)&d_B);
  clerr= clSetKernelArg(clkern, 2, sizeof(cl_mem),(void *)&d_C);
  clerr= clSetKernelArg(clkern, 3, sizeof(int), &N);
```

Host code – kernel launch

```
cl_event event=NULL;
clerr= clEnqueueNDRangeKernel(clcmdq, clkern, 2, NULL,
     Gsz, Bsz, 0, NULL, &event);
clerr= clWaitForEvents(1, &event);
clEnqueueReadBuffer(clcmdq, d_C, CL_TRUE, 0,
     N*sizeof(float), h_C, 0, NULL, NULL);
clReleaseMemObject(d_A);
clReleaseMemObject(d_B);
clReleaseMemObject(d_C);
}
```

The Hercules framework HERCULES


Application


Application


Application

OpenCL

OpenMP

Programming model

NVIDIA CUDA

Platform Abstraction

Offloading mechanism and communication API


«Big» cores

ERIKA ENTERPRISE

«Little» cores

Host subsystem

1

CUDA fware & runtime

Processing Clusters/cores

GPU subsystem

2

ERIKA ENTERPRISE

Many-core subsystem

3

HW IP

Softcores + accelerators

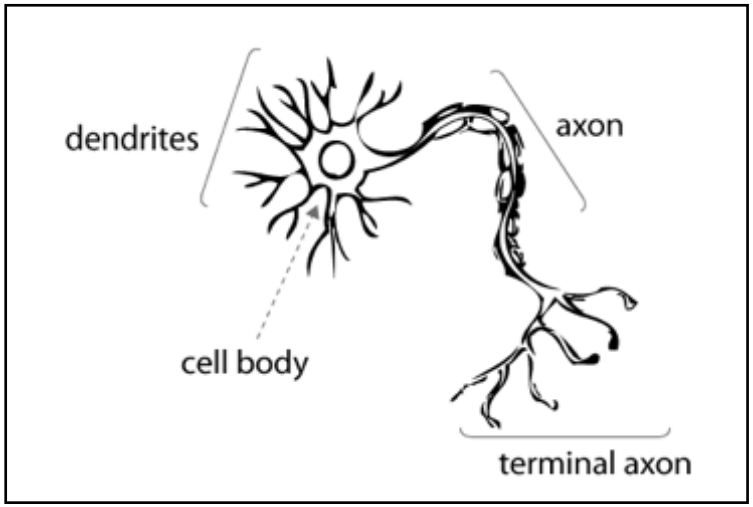
FPGA subsystem

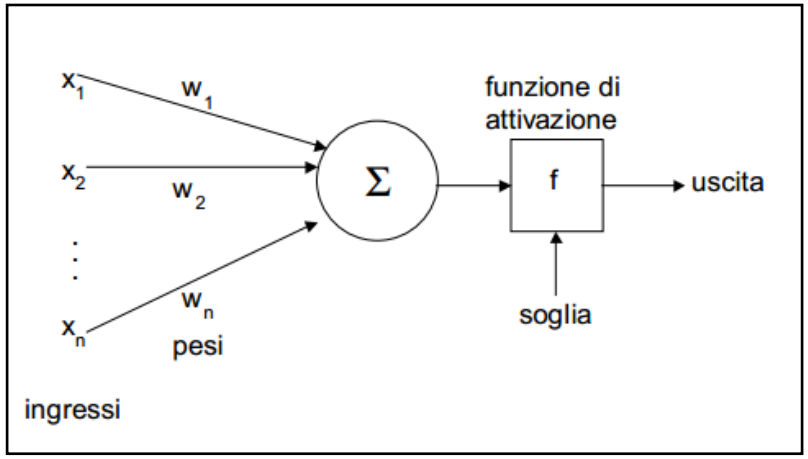
Several accelerators «flavours»

Neural Networks on FPGA accelerators

Neural networks

- Bio-inspired
- Based on neurons arranged in layers
  - And sub-layers
- Convolutional neural network
  - Perform Convolutions

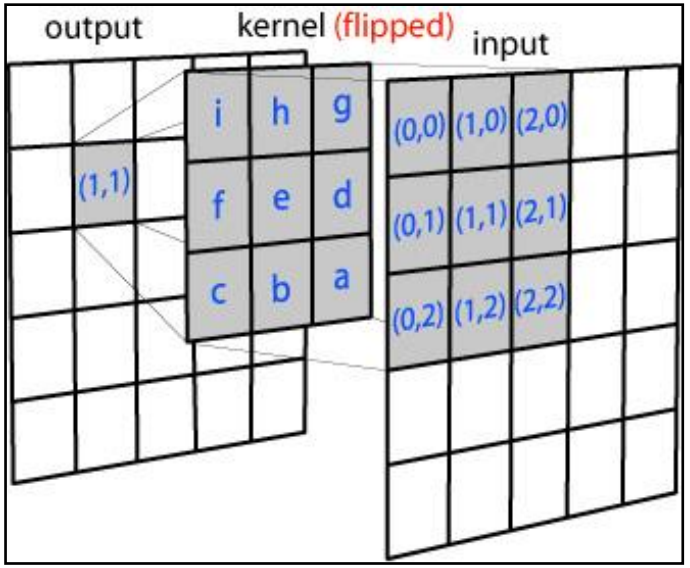




Convolution

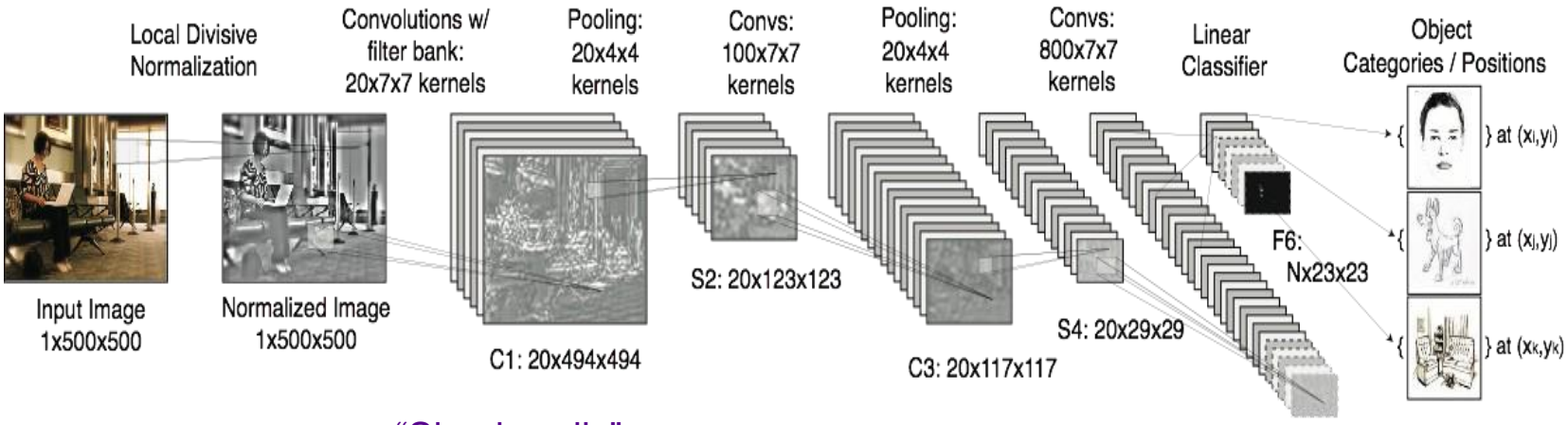
- Computation-intensive
- Suitable for implementation in hardware
- In computer vision, blurring

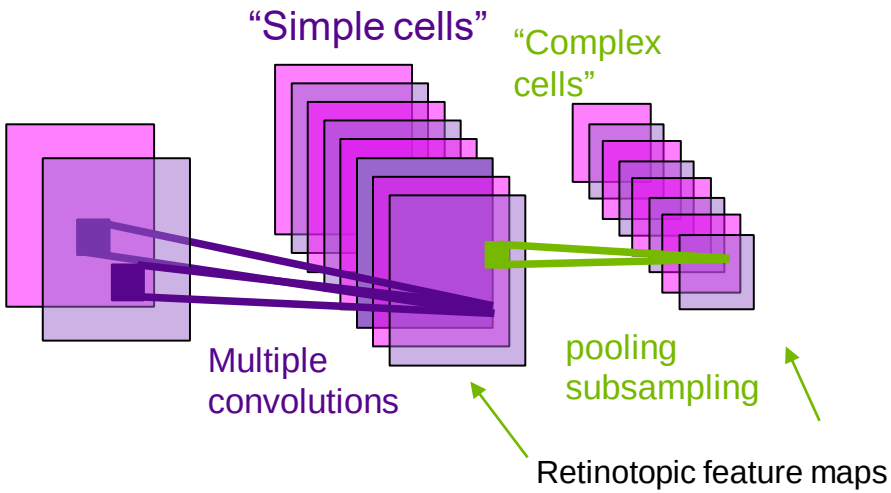
$$(f * g)(t) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau) g(t-\tau)\, d\tau$$
$$= \int_{-\infty}^{\infty} f(t-\tau) g(\tau)\, d\tau.$$



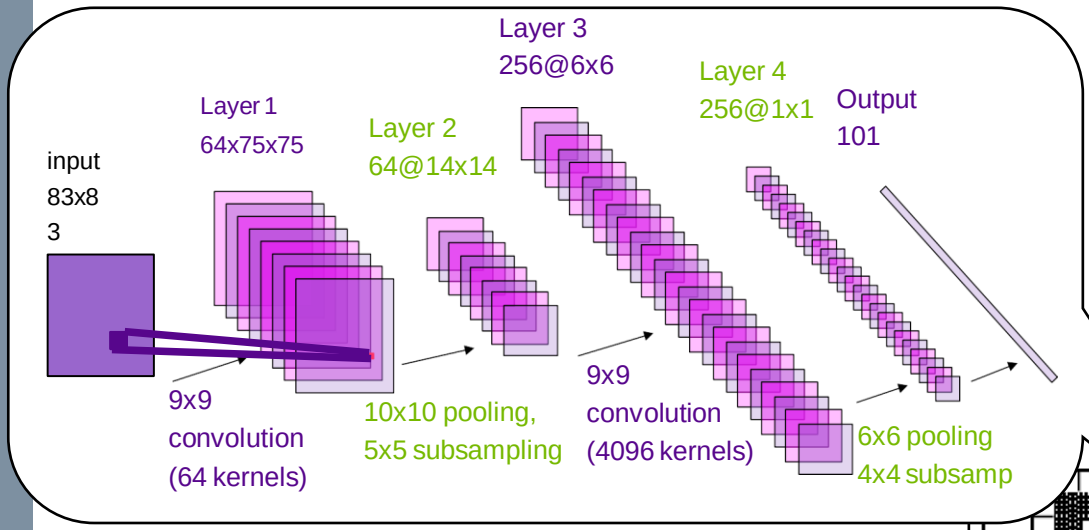
The convolutional net model

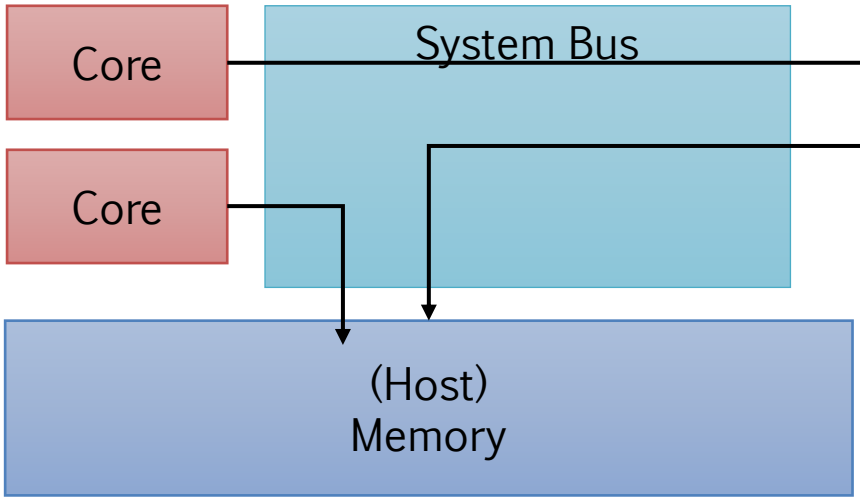
› (Multistage Hubel-Wiesel system)

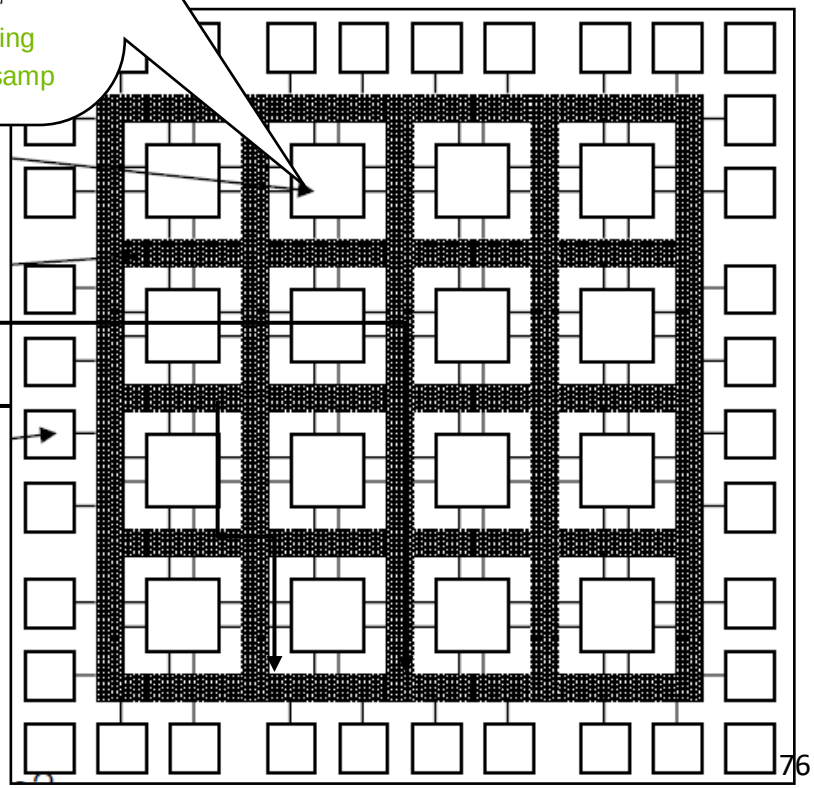




The convolutional net model (cont'd)







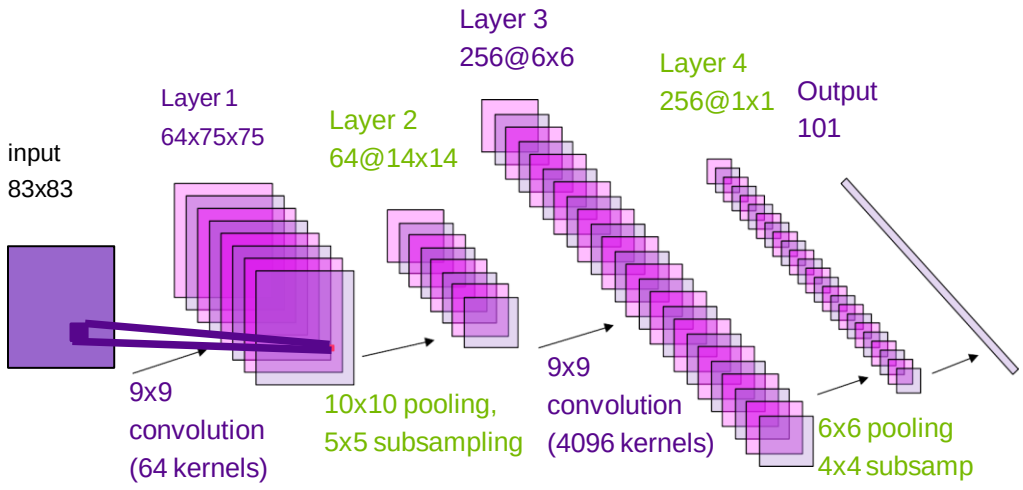
Network parameters

Network topology

- How many layers and sublayers?
- How big they are?
- How are they connected?

Neuron type

- CNN
- int/float datatypes
- How to perform pooling?



Training a network

"The training problem"

- To set the weights/CNN kernels
- Training set must be huge

A "big data" problem

- Why do you think Google does self-driving cars?
- Why do you think big cloud players want our data?

References

Course website

- http://hipert.unimore.it/people/paolob/pub/Industrial_Informatics/index.html

My contacts

- paolo.burgio@unimore.it
- http://hipert.mat.unimore.it/people/paolob/

Resources

- Xilinx Zynq-7000 All Programmable SoC => https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html
- Pynq => http://www.pynq.io/
- Xilinx Ultrascale => https://www.xilinx.com/products/technology/ultrascale.html
- A "small blog« => http://www.google.com