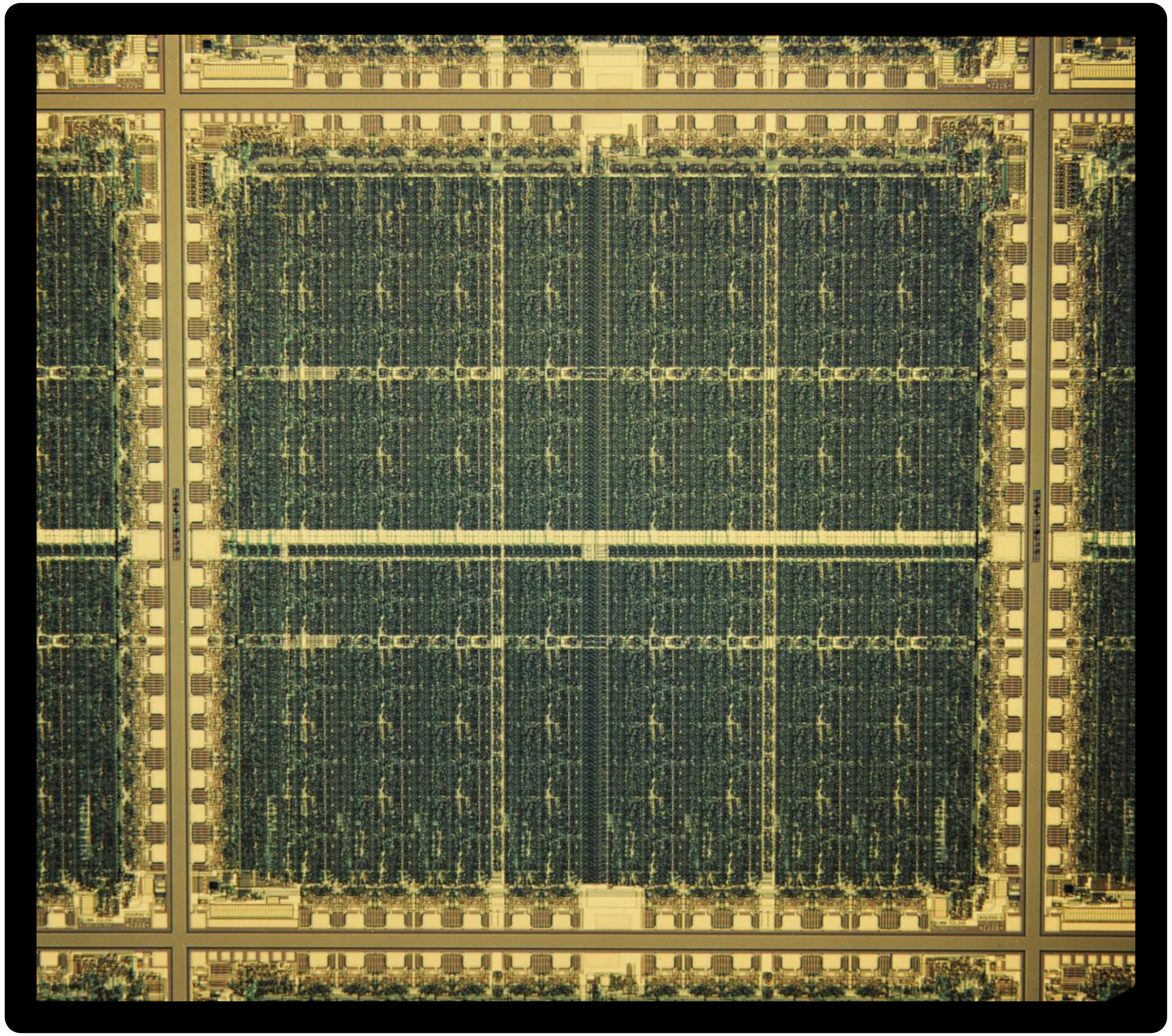
ALL PROGRAMMABLE
ANY MEDIA
ANY MACHINE
5G
4K/8K
ANY STANDARD
ANY NETWORK
5G Wireless • SDN/NFV • Video/Vision • ADAS • Industrial IoT • Cloud Computing

XILINX
ALL PROGRAMMABLE™

Optimizing OpenCL applications on Xilinx FPGA

Jeff Fifield, Ronan Keryell, Hervé Ratigner, Henry Styles, Jim Wu

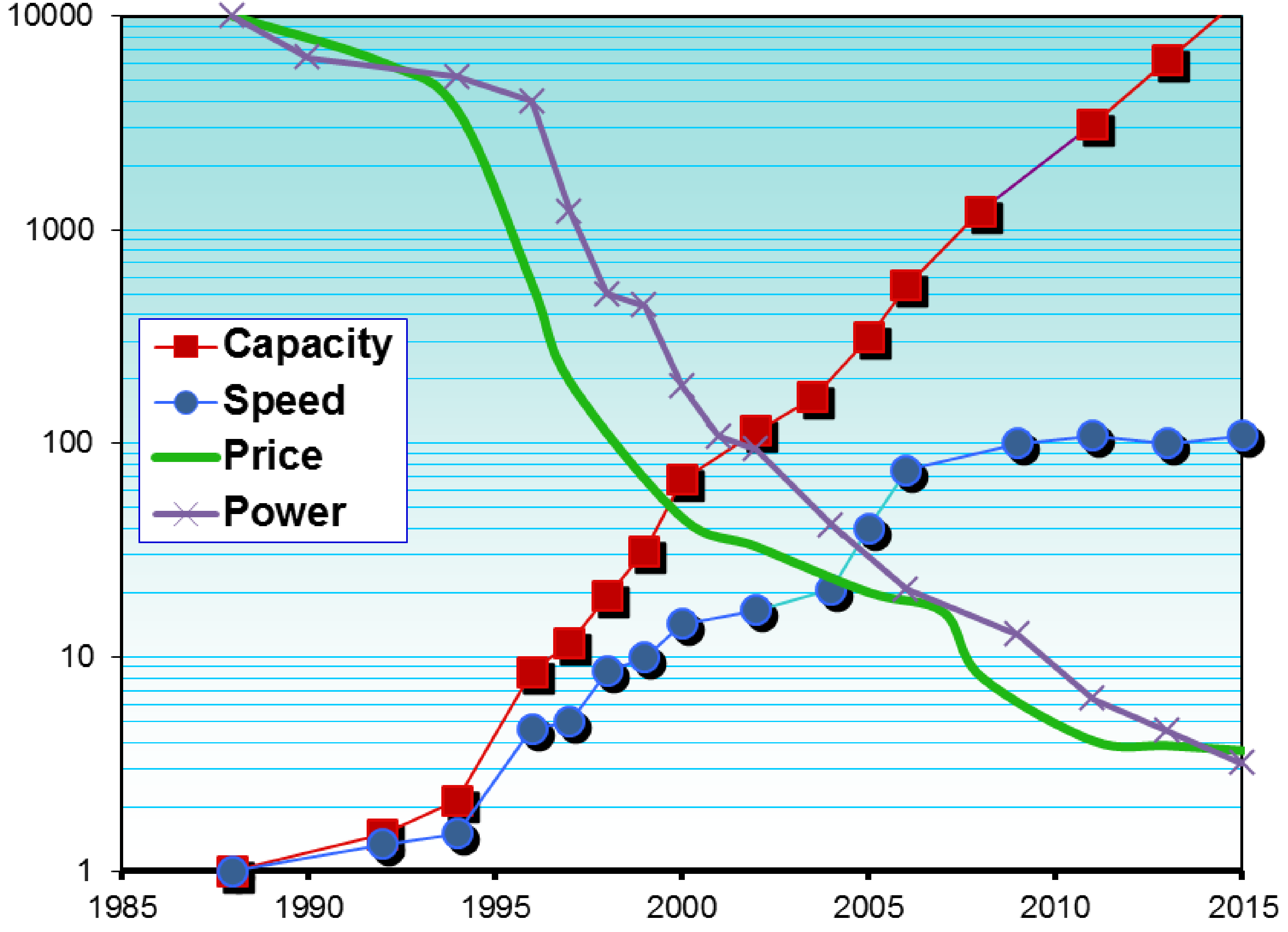
Once upon a time the XC2064…



1985: the First FPGA

- 64 flip-flops
- 128 3-LUT
- 58 I/O pins
- 18 MHz (toggle)
- 2 µm 2LM

Since then…



- **10,000x more logic…**
  - Plus embedded IP
    - Memory
    - Microprocessor
    - DSP
    - Gigabit Serial I/O
- **100x faster**
- **5,000x lower power/gate**
- **10,000x lower cost/gate**

Three Ages of the FPGAs: Trimberger, S, Proceedings of the IEEE | Vol. 103, No. 3, March 2015

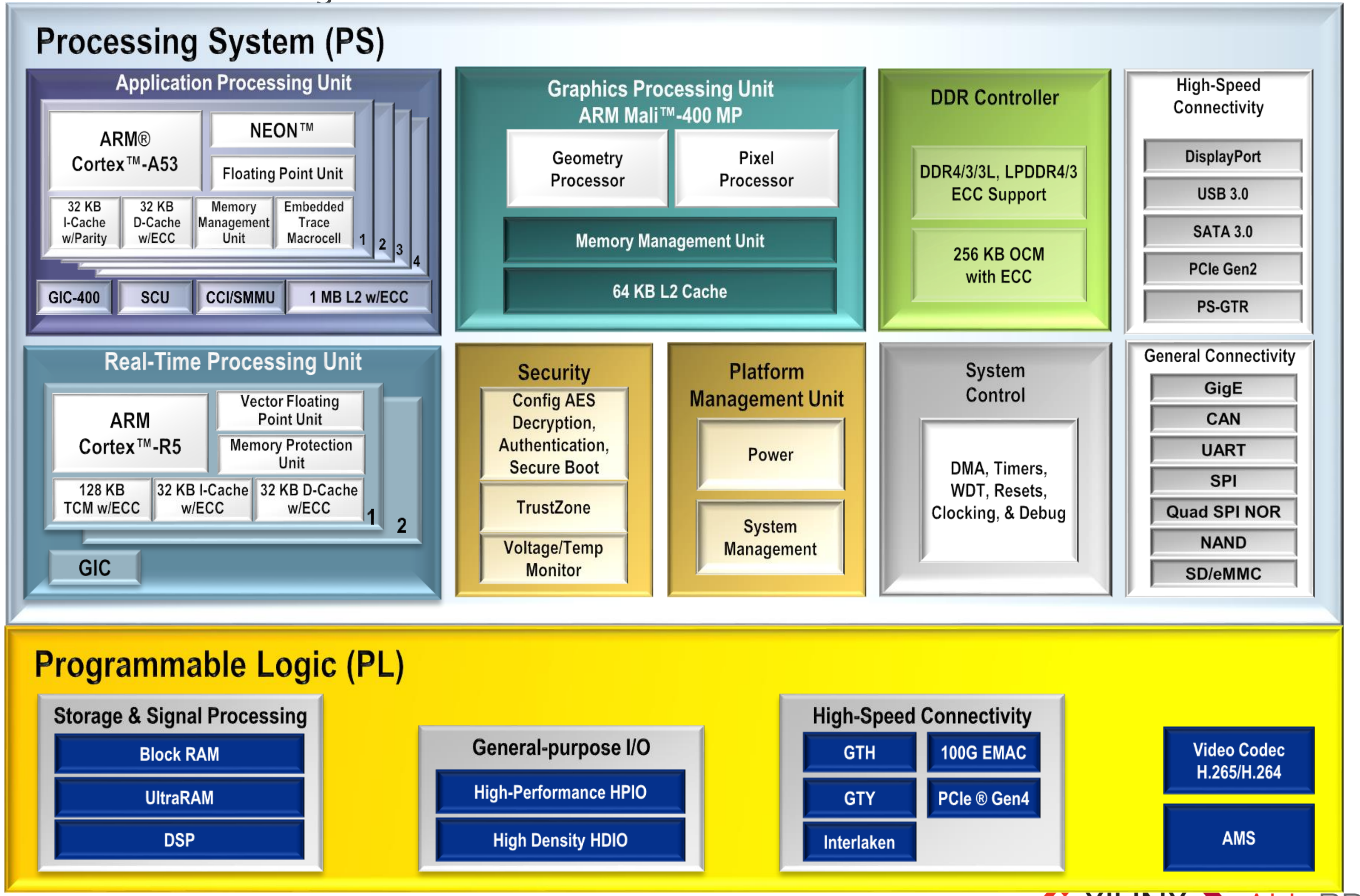
Next generation challenges

- **Power**
- **Performance**
- **Cost drivers**
- **Power management**
- **64bit processing**
- **Real-time processing**
- **Video and graphics processing**
- **Pervasive safety and security**
- **Higher levels of processor-fabric integration**

XILINX ALL PROGRAMMABLE.

Zynq UltraScale+ MPSoC Overview

Heterogeneous Multi-Processing

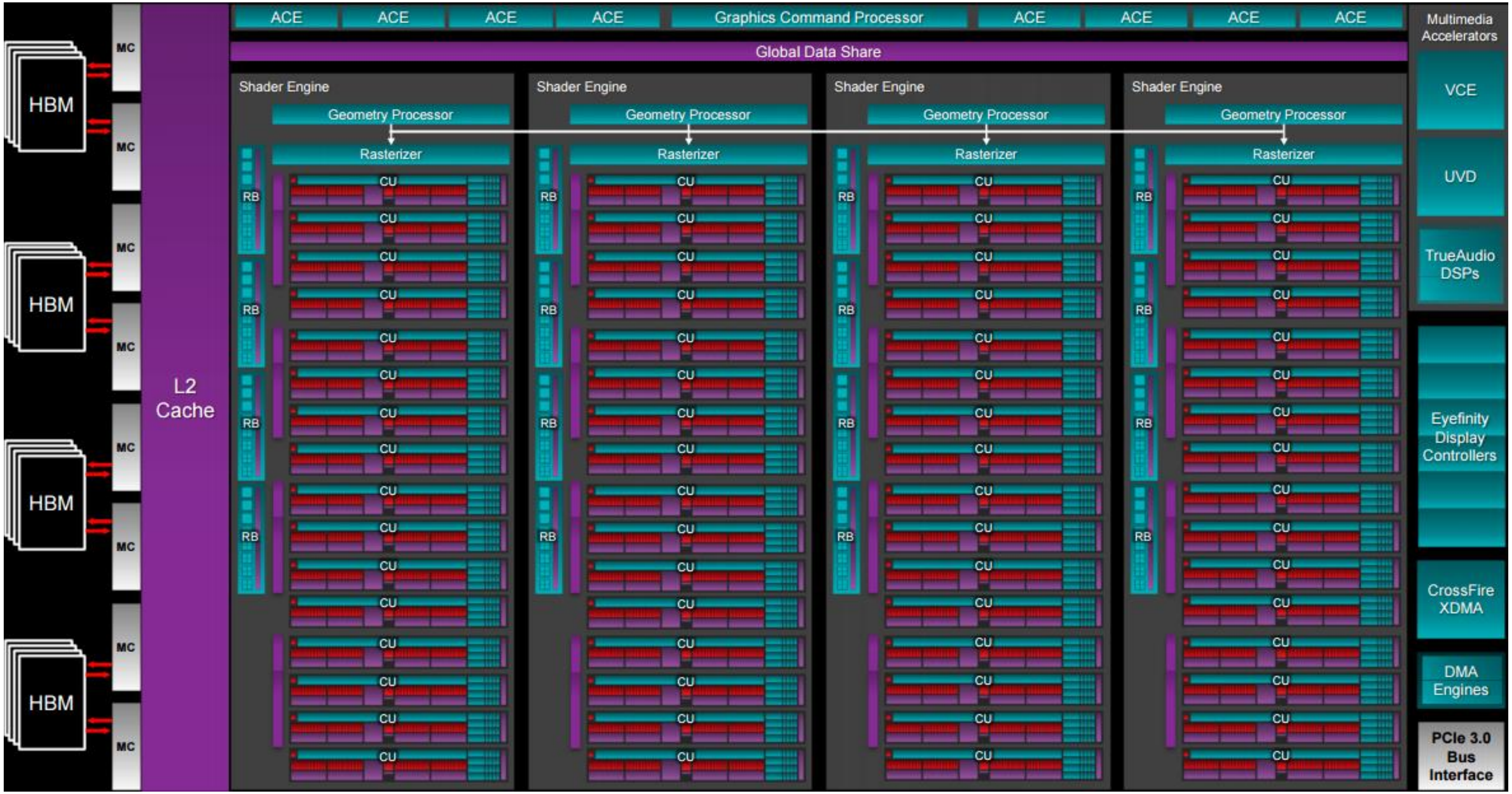


Agenda

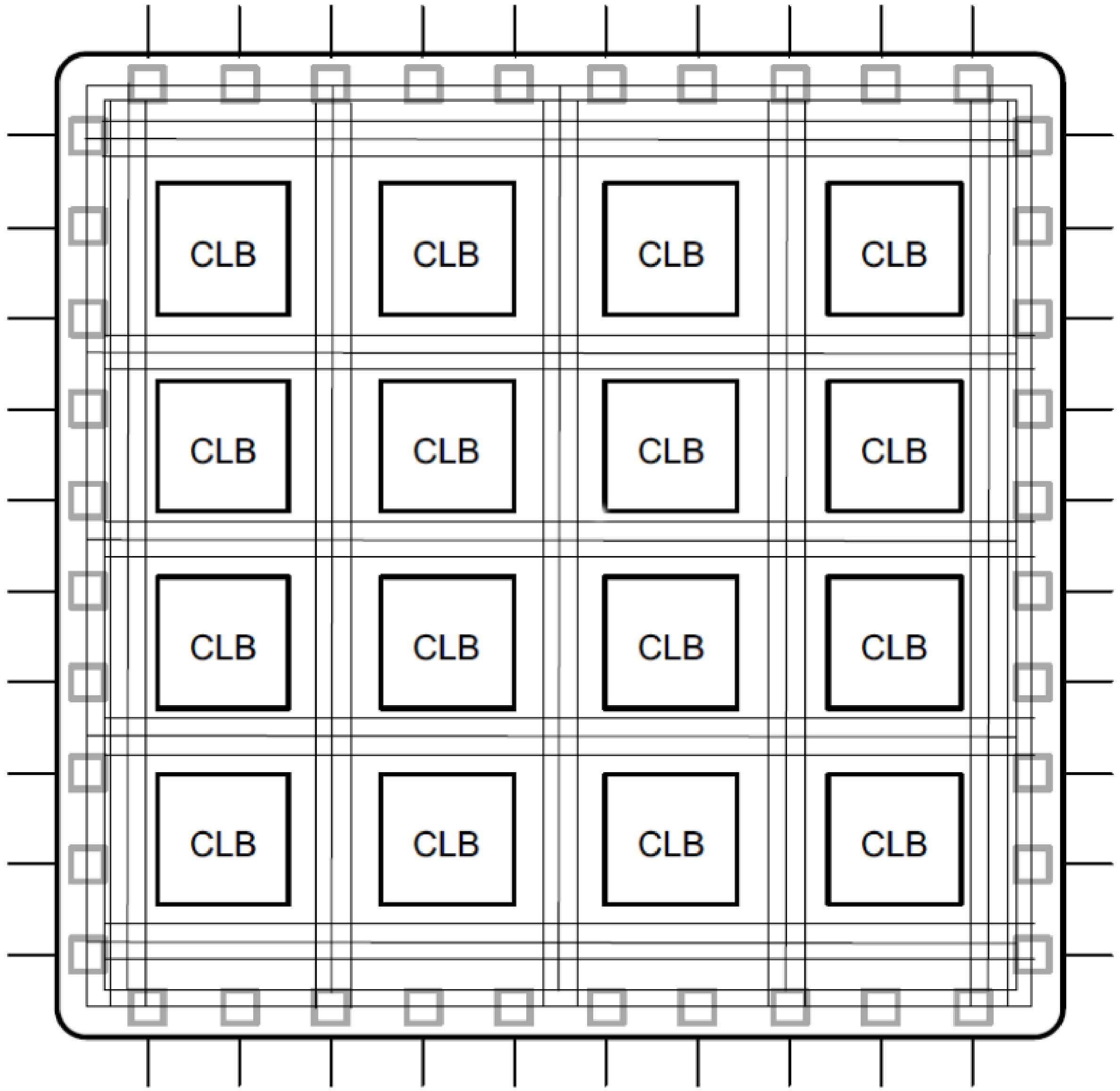
- FPGA architecture vs CPU/GPU
- The programming challenge
- OpenCL optimization on FPGA
- Using I/O and other IP blocks

Architecture

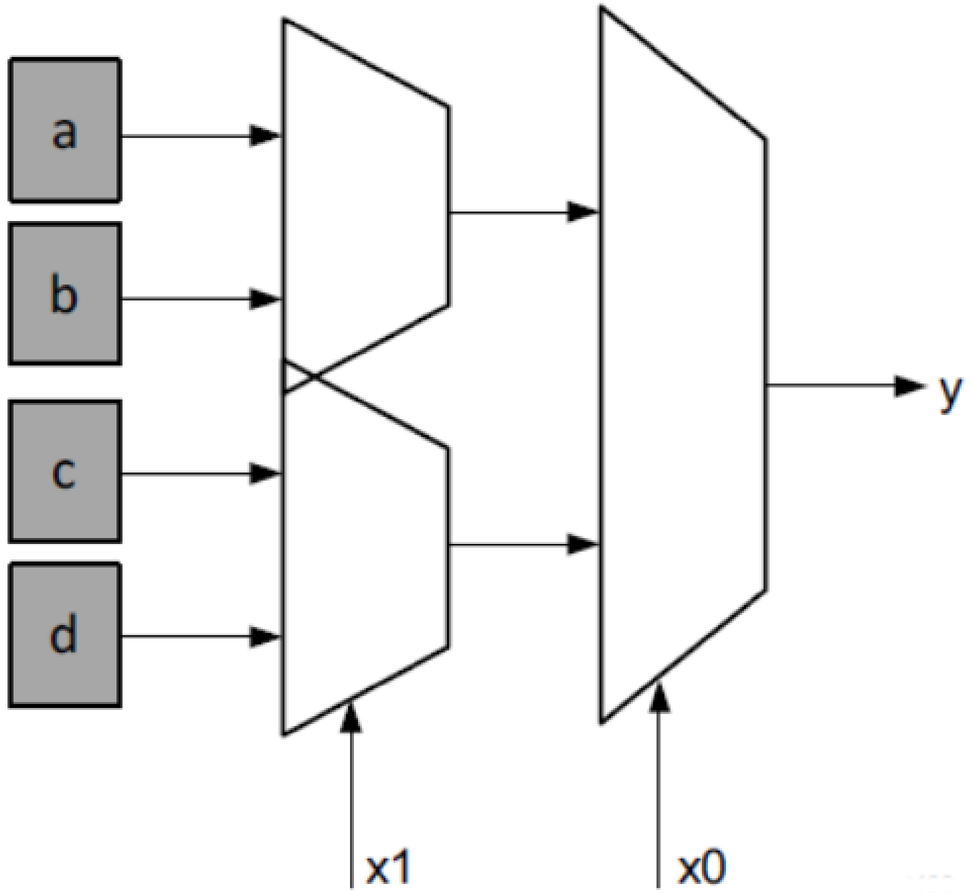
From AMD Fiji XT GPU (2015)…

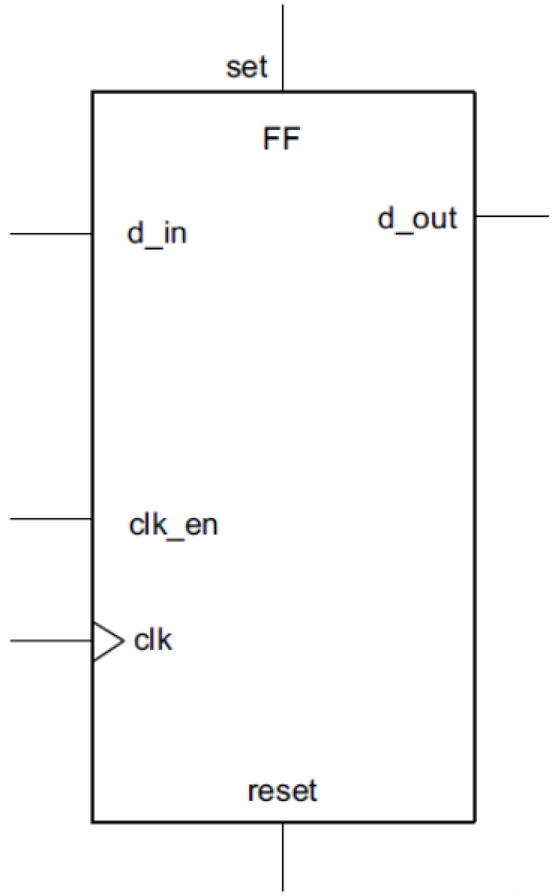


… to Field-Programmable Gate Array (FPGA)



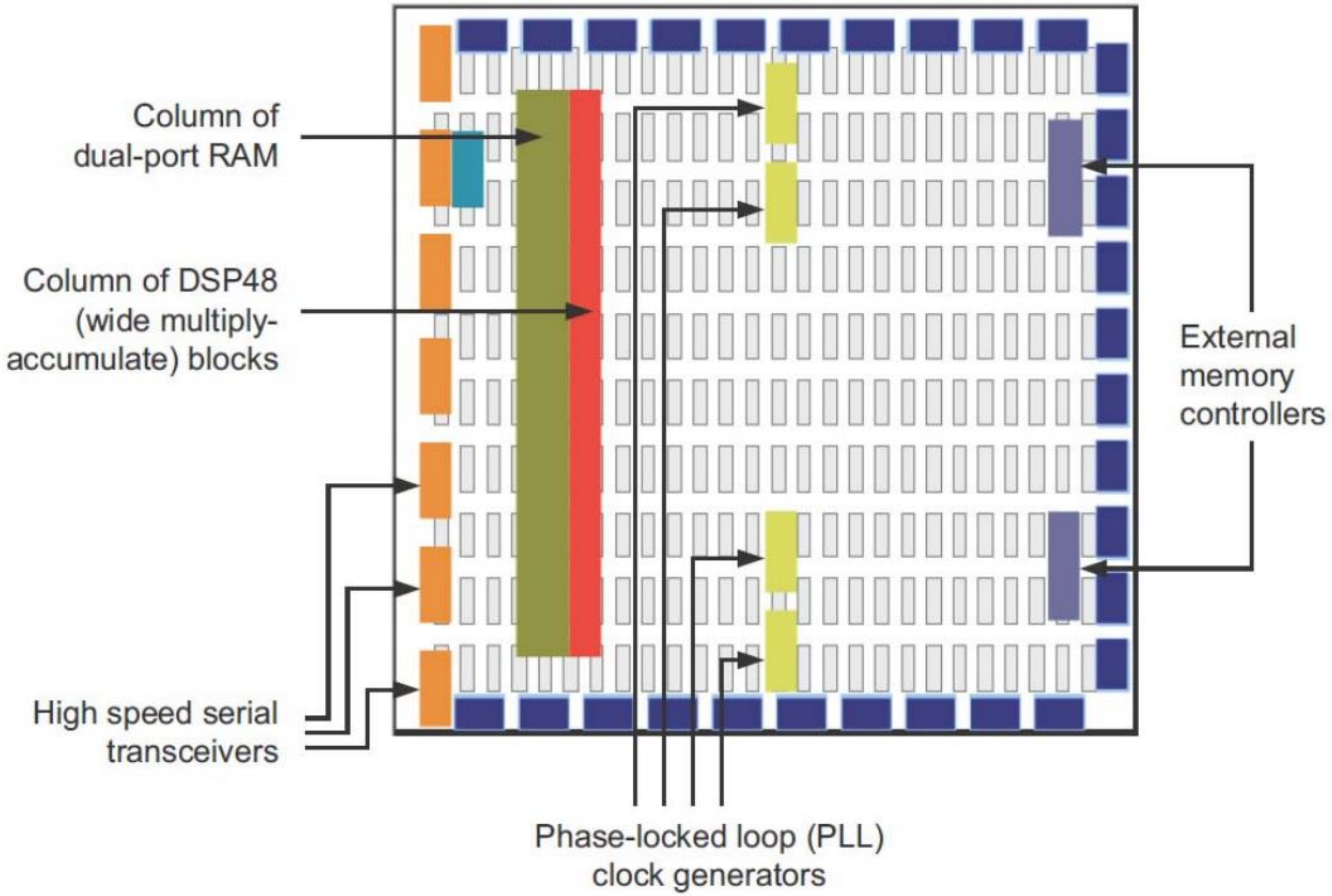
Basic architecture =
Lookup Table + Flip-Flop storage + Interconnect



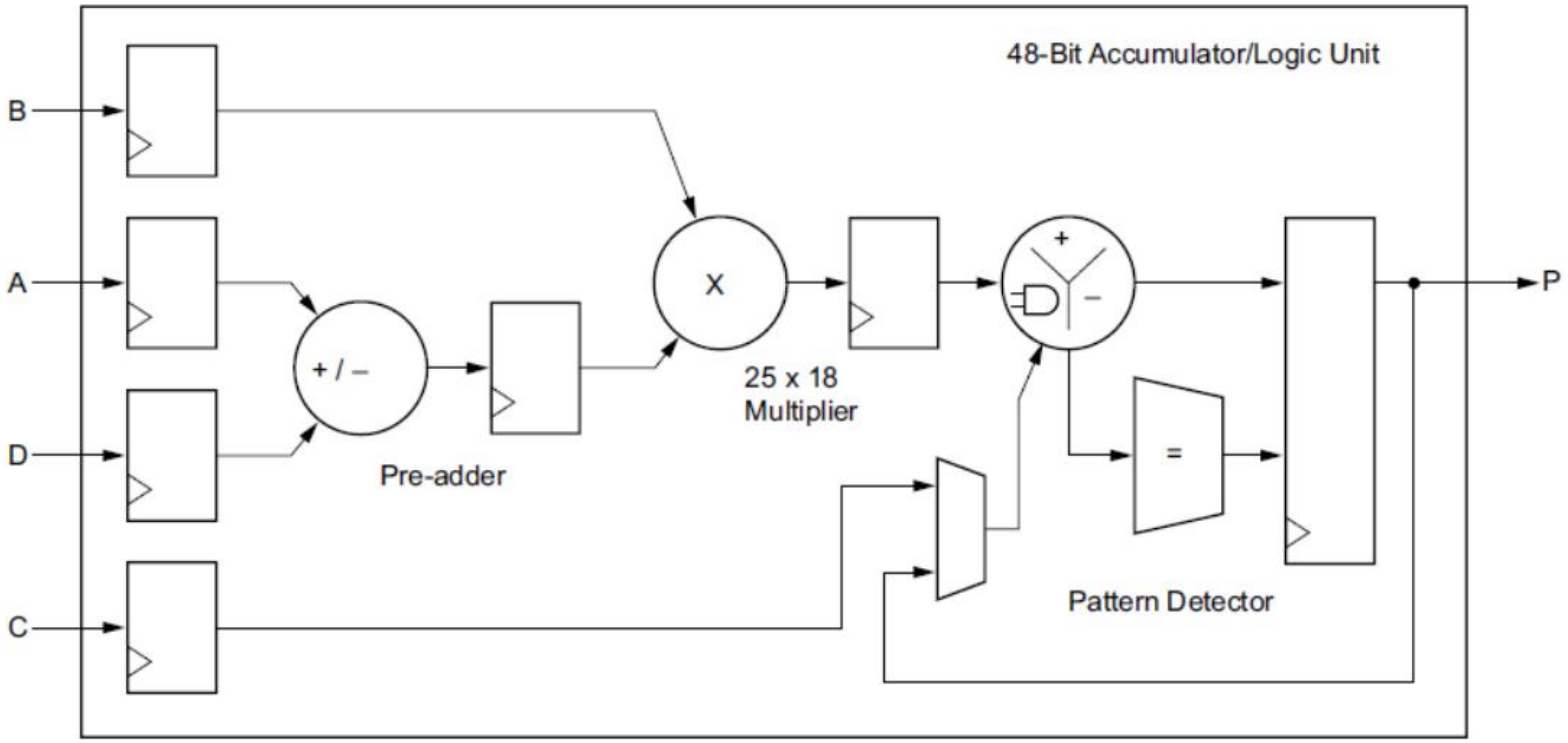


Typical Xilinx LUT have 6 inputs

Global view of programmable logic part

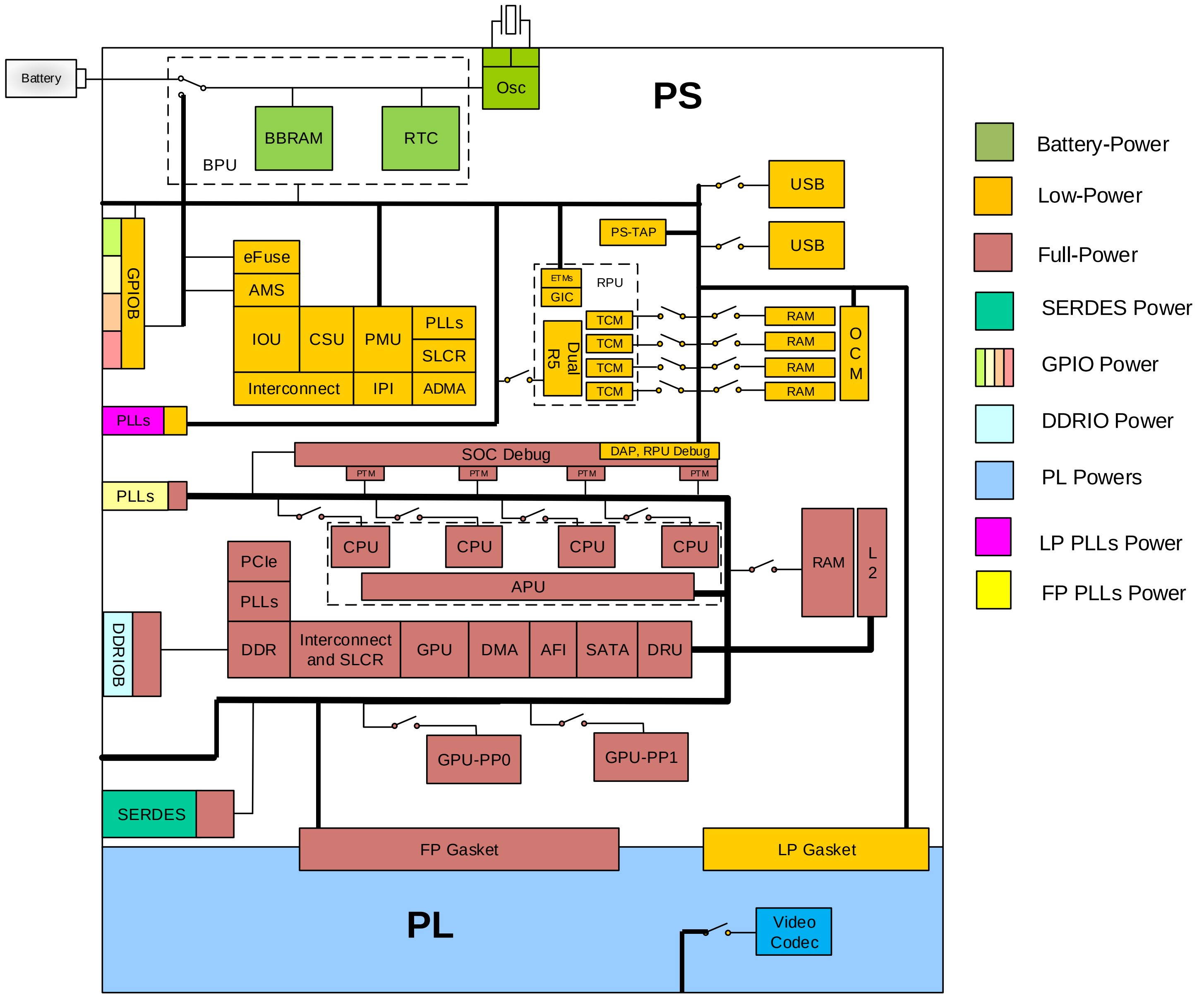

Column of dual-port RAM
Column of DSP48 (wide multiply-accumulate) blocks
High speed serial transceivers
External memory controllers
Phase-locked loop (PLL) clock generators

DSP48 block overview



Power wall… Power-Domains and Power-Gating!

- **Multiple power domains**
  - Low power domain
  - Full power domain
  - PL power domain
- **Power gating**
  - A53 per core
  - L2 and OCM RAM
  - GPU, USB
  - R5s & TCM
  - Video CODEC
- **Sleep Mode**
  - 35mW sleep mode
  - Suspend to DDR with power off



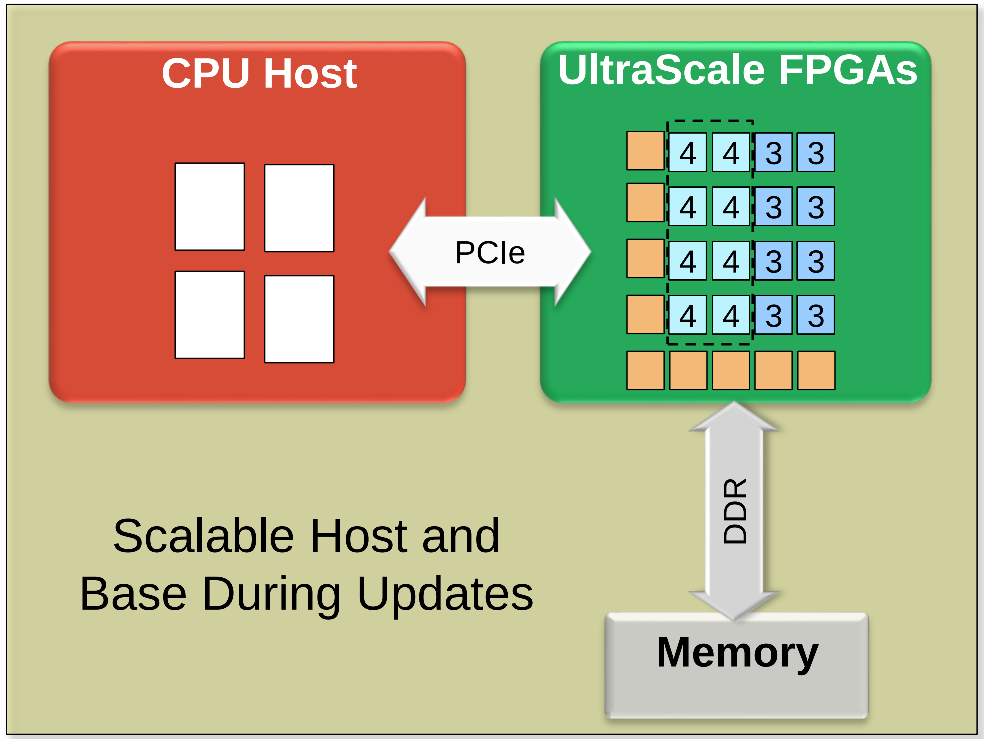
XILINX ALL PROGRAMMABLE™

∃Programming challenges…

Xilinx FPGA programming

- FPGA used to be programmed by electrical engineers…
  - Typically at Register-Transfer Level (RTL)
  - Hardware description languages (VHDL, Verilog)
  - Full control for extreme performance but very low productivity
- Need to attract a wider audience of programmers to new markets
  - Evolution towards normal languages (C & C++/SystemC) through High-Level Synthesis
  - SDSoC for programming Xilinx Zynq UltraScale+ MPSoC with #pragma
  - SDNet to implement network appliances
  - SDAccel to bring OpenCL accelerator experience

CPU/GPU-like Runtime Experience on FPGAs



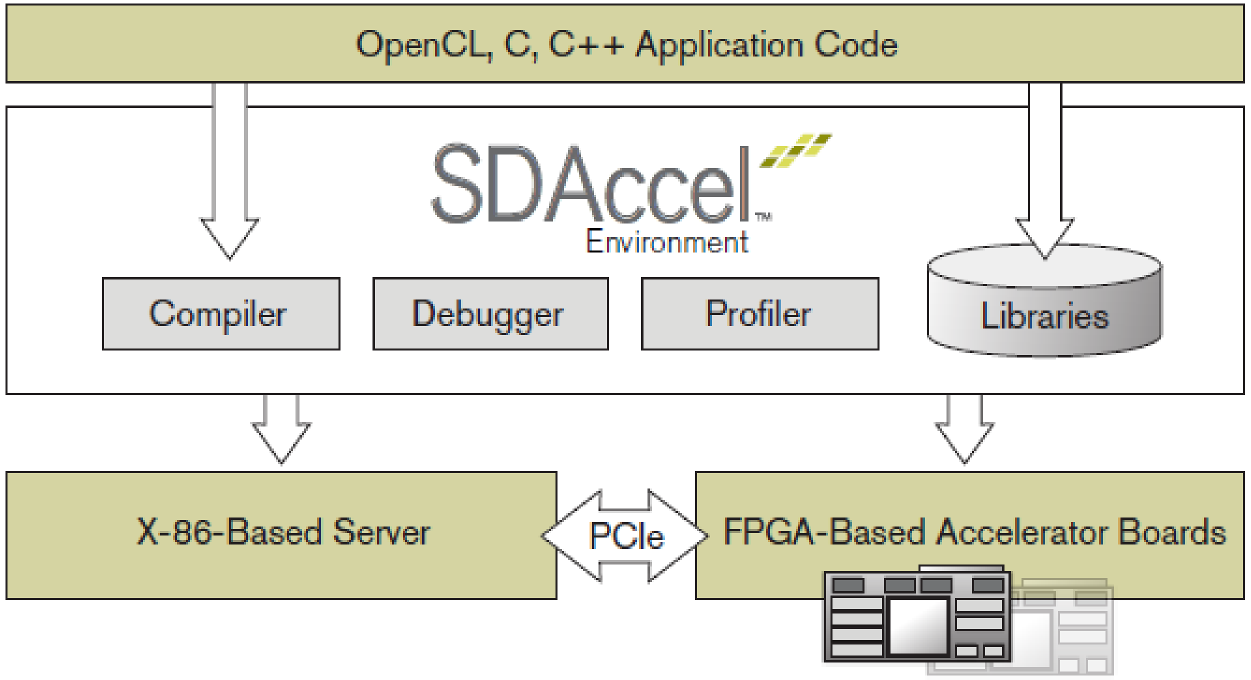
CPU/GPU Runtime Experience

- On-demand loadable acceleration units
- Always-on interfaces (Memory, Ethernet PCIe, Video)
- Optimize resources through hardware reuse

SDAccel Environment and Ecosystem

Application Developers

SDAccel - CPU/GPU Development Experience on FPGAs



Library Providers

COTS Board Partners

XILINX ALL PROGRAMMABLE.

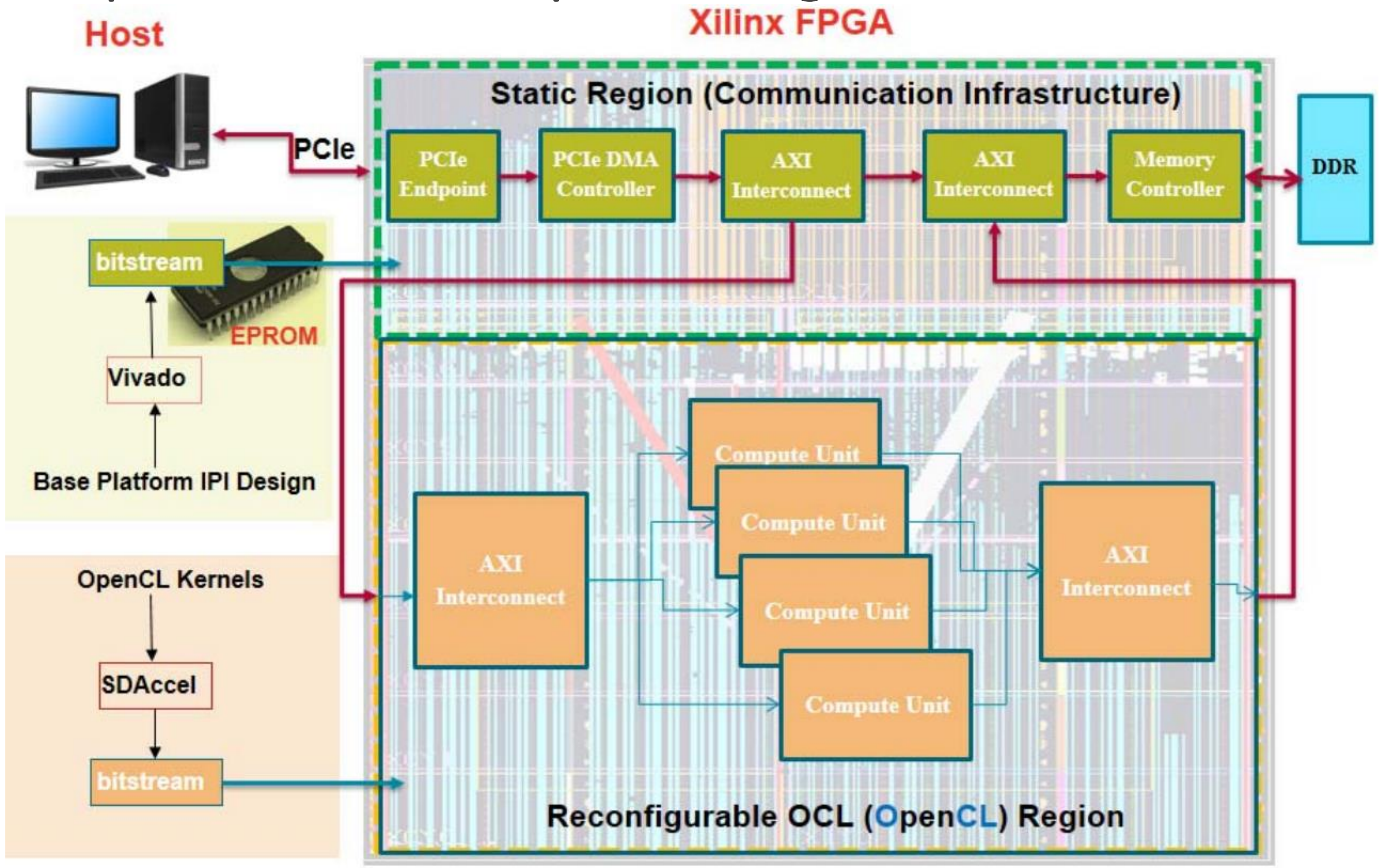
SDAccel OpenCL implementation

- **Each OpenCL work-group mapped on 1 IP block**
  - 1 physical CU/work-group
- **OpenCL host API handles accelerator invocation**
- **Support several kernels per program**
- **Pipelining of work-items in a work-group possible with #pragma or attributes**
- **Off-line kernel compilation**
- **Generate FPGA container with bitstream + metadata**
- **Different compilation/execution modes**
  - CPU-only mode for host and kernels
  - RTL version of kernels for co-simulation
  - Real FPGA execution
- **Estimation of resource utilization by the tool**

Typical deployment workflow

- **Optimize on x86 platform with emulator and auto generated cycle accurate models**
  - Identify application for acceleration
  - Program and optimize kernel on host
  - Compile and execute application for CPU
  - Estimate performance
  - Debug FPGA kernels with cycle accurate models on CPU
- **Deployment on FPGA**
  - Compile for FPGA (longest step)
  - Execute and validate performance on card

SDAccel platform with OpenCL region



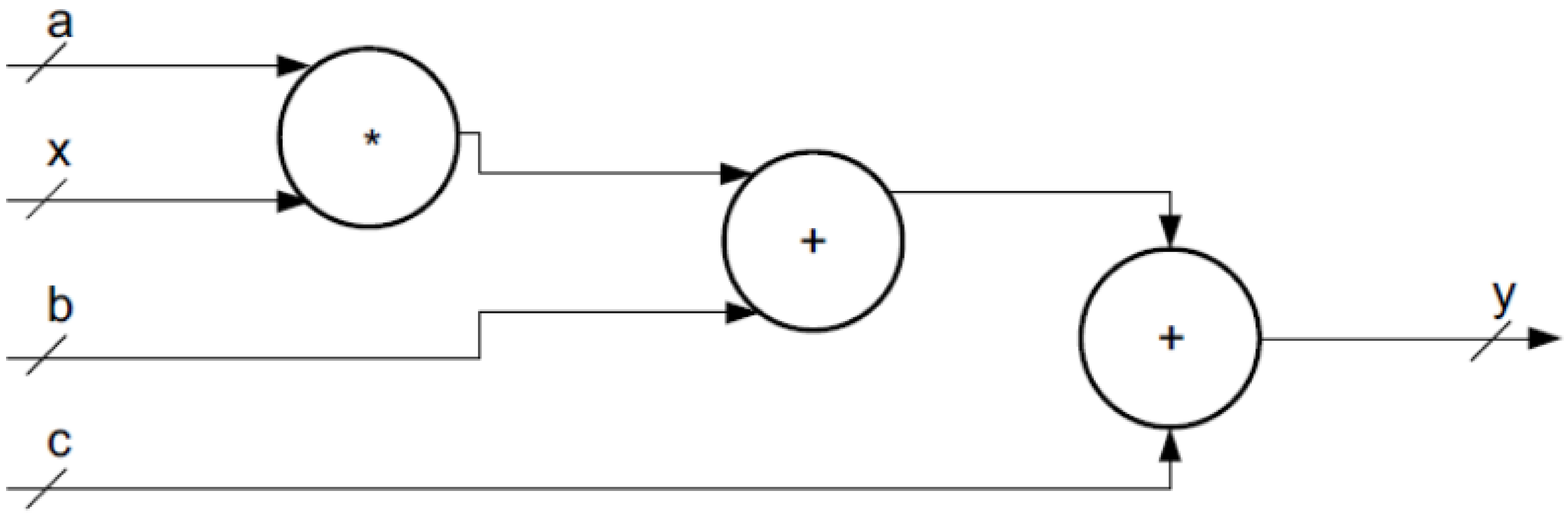
We all want Independent Forward Progress!

- **In OpenCL 2.x specification: no guaranty of independent forward progress between kernels, between work-groups or work-items**
  – Portable program cannot implement producer-consumer algorithm… ☹
- **On FPGA everything is synthesized to real hardware**
  – Since physical existence, independent forward progress possible by construction! ☺
- **⇒ Data flow applications without host control possible**
  – Even with cycles in the graph! ☺

Behind the scene: understanding High-Level Synthesis

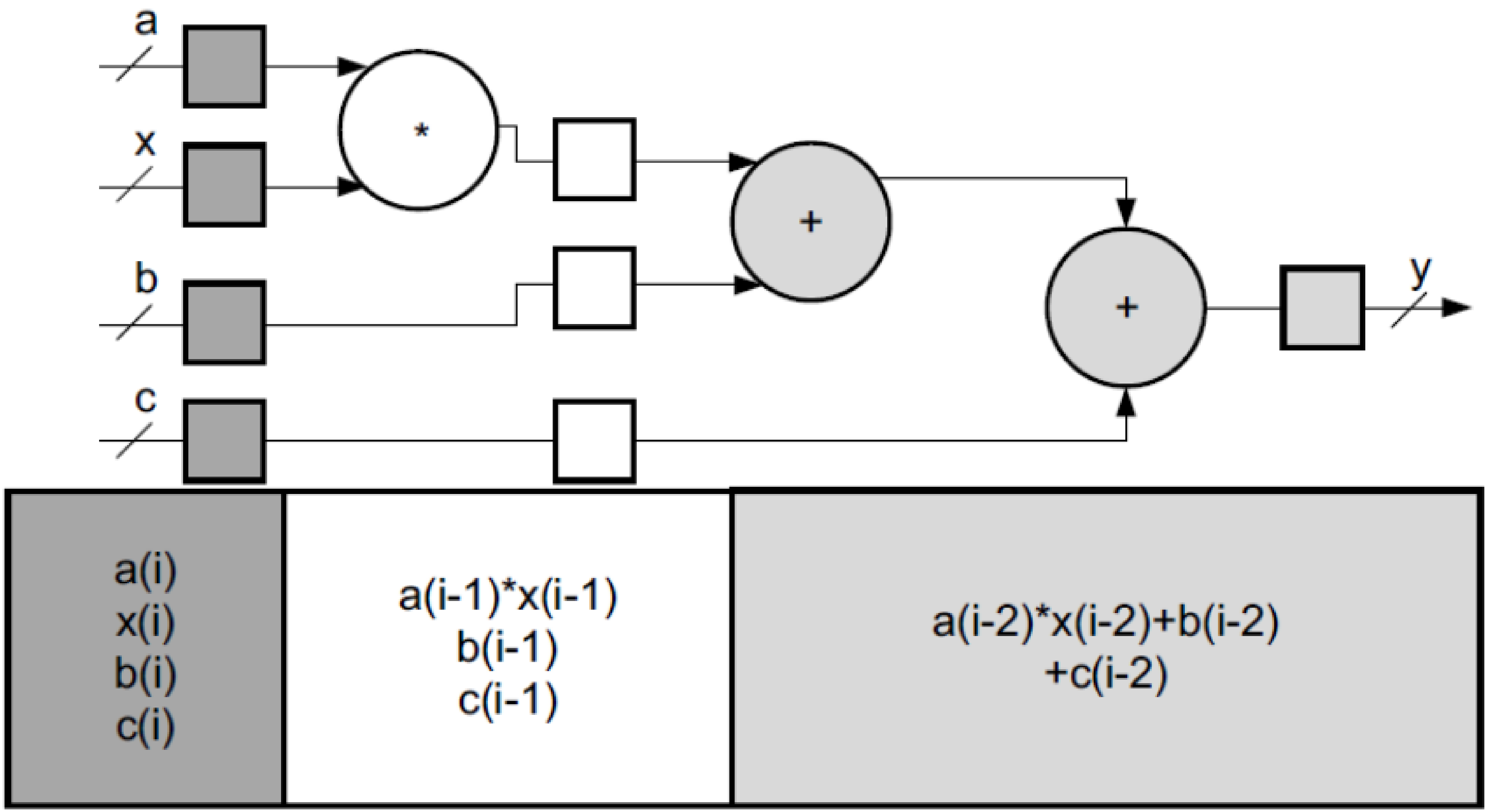
Compilation of expressions

- y = a*x + b + c;



Compilation of a pipelined expression

- y = a*x + b + c;

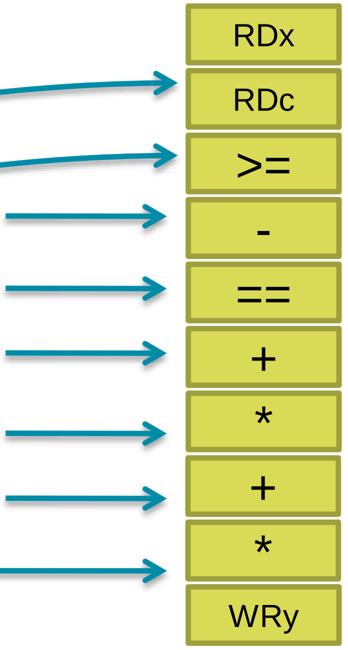


HLS: Control & Datapath Extraction

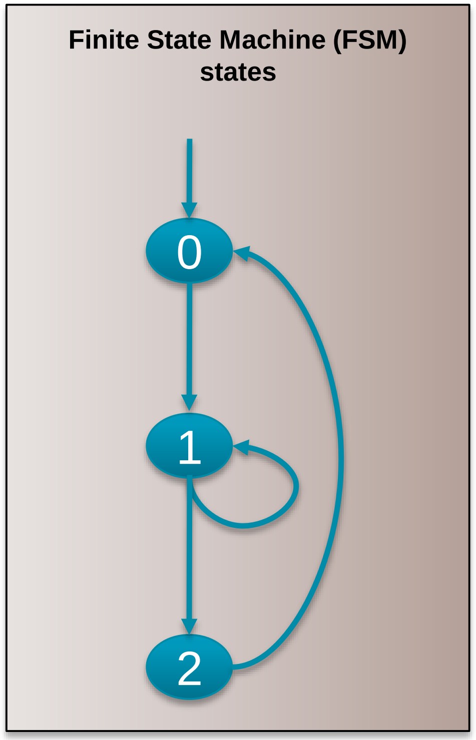
Code

```
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;
}
```

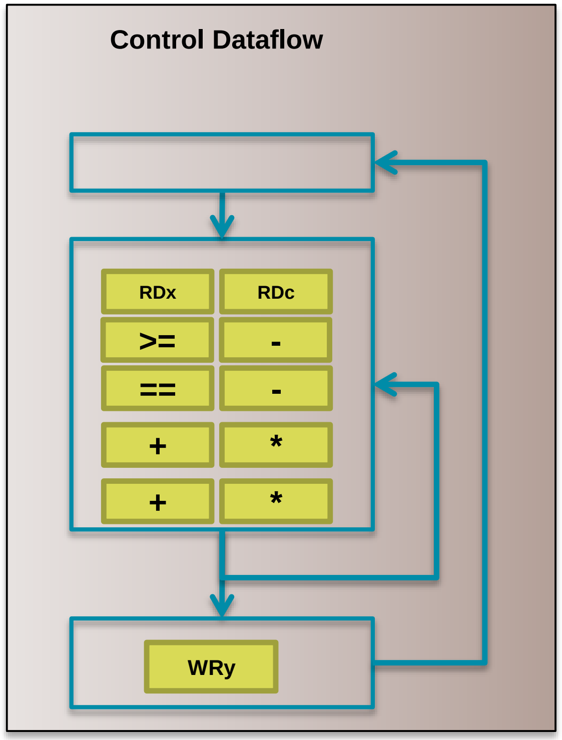
Operations



Control Behavior



Control & Datapath Behavior



From any C code example ..

Operations are extracted…

The control is known

A unified control dataflow behavior is created.

XILINX ALL PROGRAMMABLE.

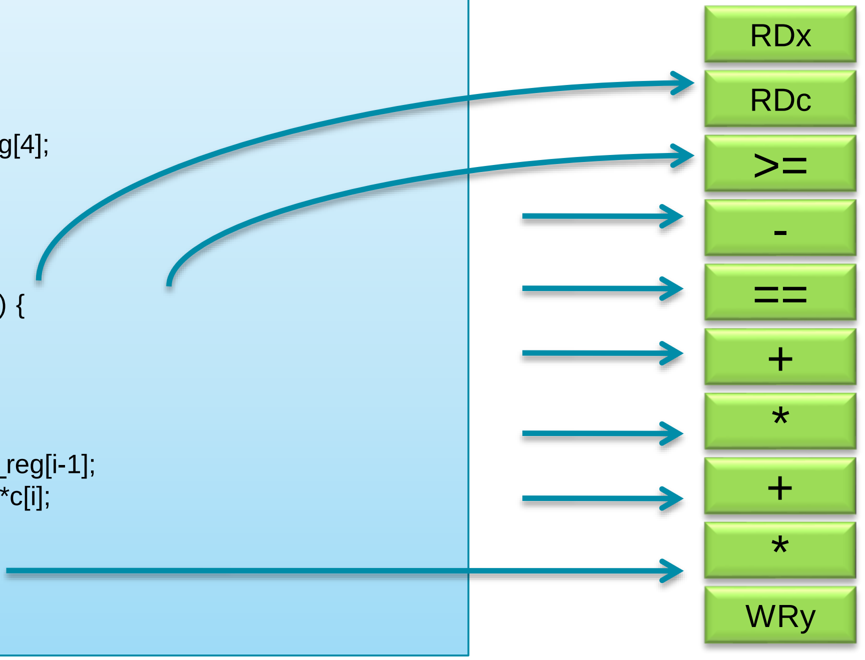
Types = Operator Bit-sizes = Hardware efficiency

Code

```
void fir (
  data_t *y,
  coef_t c[4],
  data_t x
  ) {

  static data_t shift_reg[4];
  acc_t acc;
  int i;

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

From any C code example ...

Operations



Operations are extracted…

Types

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

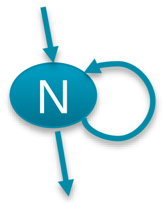
Can be used to define any variable to be a specific bit-width (e.g. 17-bit, 47-bit etc).

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

XILINX ALL PROGRAMMABLE.

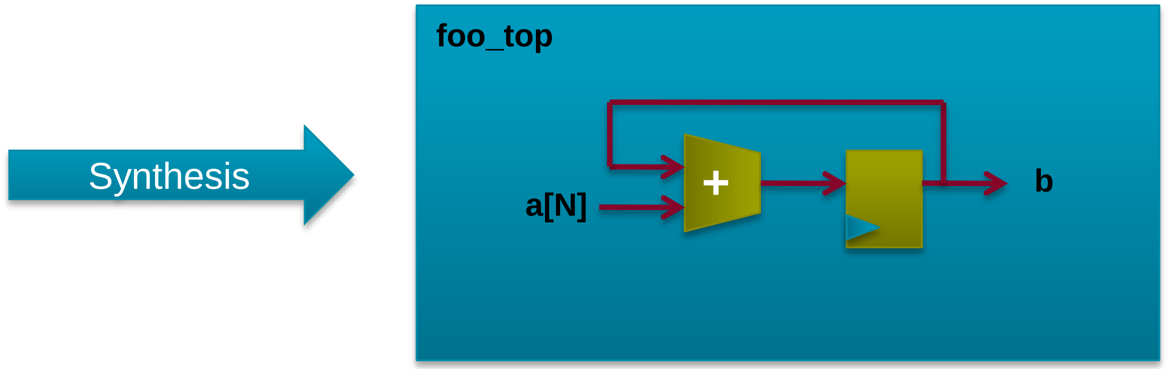
Loops

- **By default, loops are rolled**
  - Each C loop iteration ➔ Implemented in the same state
  - Each C loop iteration ➔ Implemented with same resources



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

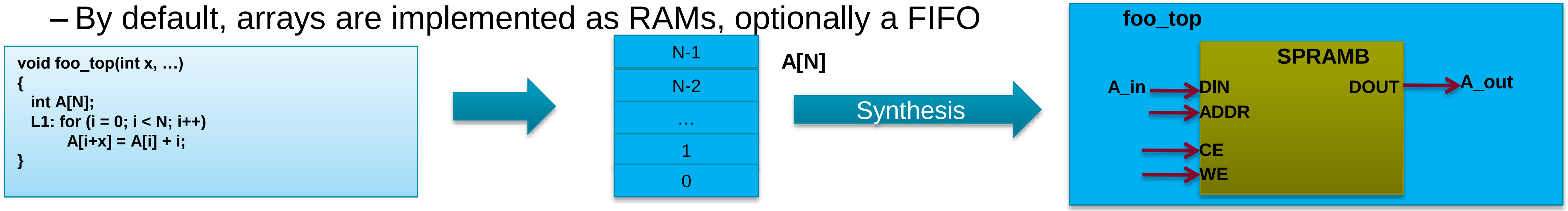
Loops require labels if they are to be referenced by Tcl directives
(GUI will auto-add labels)



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

Arrays in HLS

- **An array in C code is implemented by a memory in the RTL**
  - By default, arrays are implemented as RAMs, optionally a FIFO



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

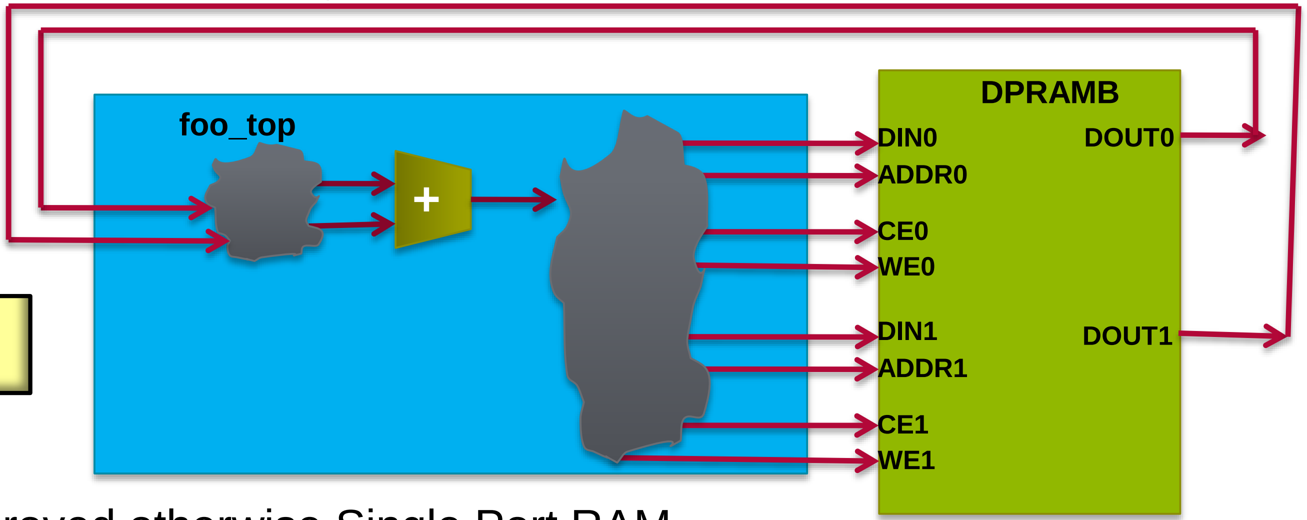
- **The array can be targeted to any memory resource in the library**
  - The ports (Address, CE active high, etc.) and sequential operation (clocks from address to data out) are defined by the library model
  - All RAMs are listed in the Vivado HLS Library Guide
- **Arrays can be merged with other arrays and reconfigured**
  - To implement them in the same memory or one of different widths & sizes
- **Arrays can be partitioned into individual elements**
  - Implemented as smaller RAMs or registers

Top-Level IO Ports

- **Top-level function arguments**
  - All top-level function arguments have a default hardware port type
- **When the array is an argument of the top-level function**
  - The array/RAM is "off-chip"
  - The type of memory resource determines the top-level IO ports
  - Arrays on the interface can be mapped & partitioned
    - E.g. partitioned into separate ports for each element in the array

```
void foo_top( int A[3*N] , int x)
{
  L1: for (i = 0; i < N; i++)
        A[i+x] = A[i] + i;
}
```

Synthesis



Number of ports defined by the RAM resource

- **Default RAM resource**
  - Dual port RAM if performance can be improved otherwise Single Port RAM

XILINX ALL PROGRAMMABLE.

Design Exploration with Directives

```
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];
   }
}
```

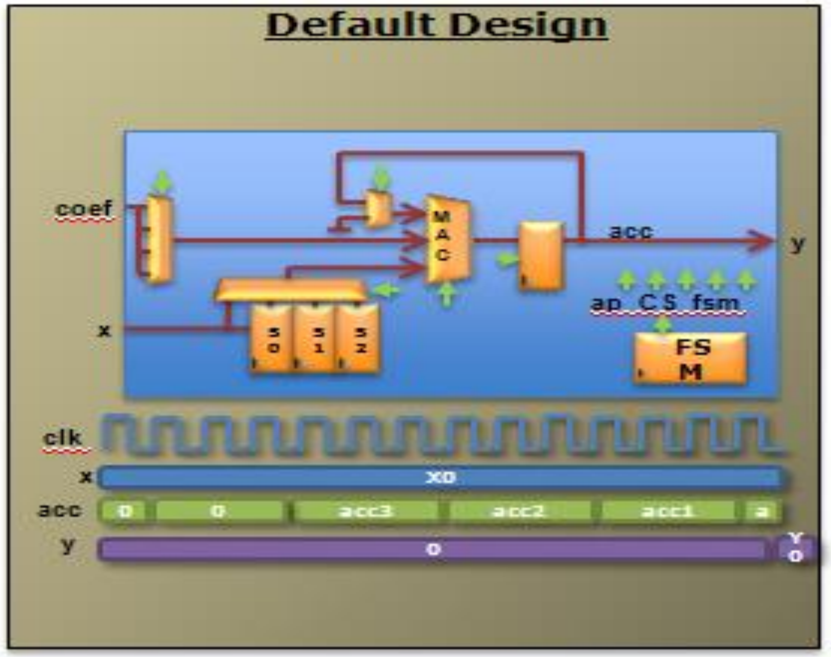
One body of code:
Many hardware outcomes

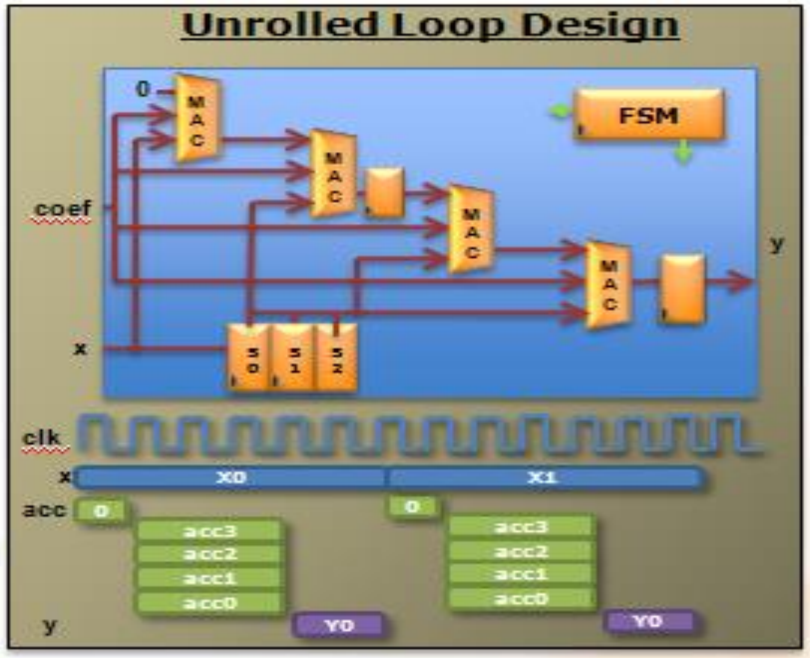
Before we get into details, let's look under the hood ….

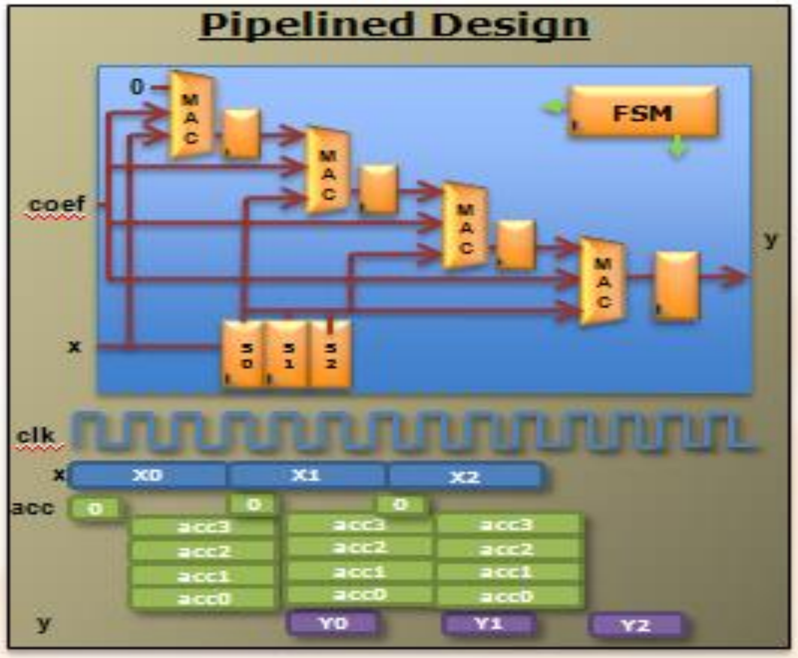
The same hardware is used for each iteration of the loop:
- **Small area**
- **Long latency**
- **Low throughput**

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**







XILINX ALL PROGRAMMABLE™

Optimizing OpenCL for Xilinx FPGA

Memory mapping

- Host memory: attached to the host only
  - Memory accessed by kernels needs to be moved by openCL API
- Constant and global memory
  - Typically placed into external SDRAM connected to FPGA
  - But can also be mapped to on-chip BlockRAM
- Local memory
  - CU-local mapped on chip, in registers or BlockRAM
- Private memory
  - Work-item-private mapped on chip, in registers or BlockRAM

→ Select carefully for the right size/latency/bandwidth trade-off

Specify work-group size for better implementation

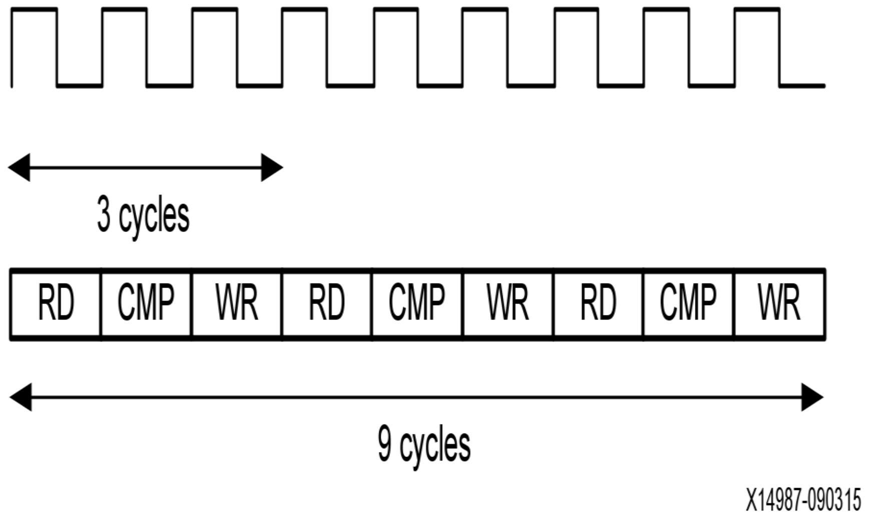
```
__kernel
__attribute__((reqd_work_group_size(4,4,1)))
void mmult32(__global int* A, __global int* B,
__global int* C) {
  // 2D Thread ID
  int i = get_local_id(0);
  int j = get_local_id(1);
  __local int B_local[256];
  int result=0, k=0;
  B_local[i*16 + j] = B[i*16 + j];
  barrier(CLK_LOCAL_MEM_FENCE);
  for(k = 0; k < 16; k++)
    result += A[i*16 + k]*B_local[k*16 + j];
  C[i*16 + j] = result;
}
```

- **Is compiled into**

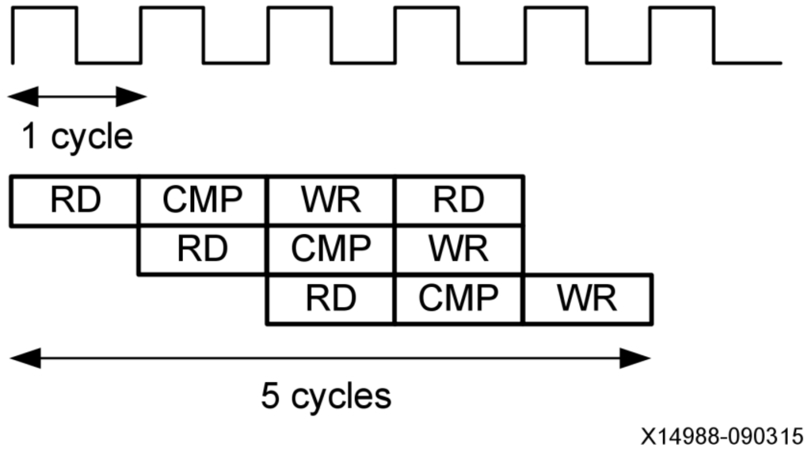
```
__kernel void mmult32(global int* A, global int* B,
global int* C) {
  localid_t id;
  int B_local[16*16];
  for(id[2] = 0; id[2] < 1; id[2]++)
    for(id[1] = 0; id[1] < 4; id[1]++)
      for(id[0] = 0; id[0] < 4; id[0]++) {
        ...
      }
  ...
}
```

Work-item pipelining

```
__attribute__ ((reqd_work_group_size(3,1,1)))
kernel void foo(...) {
  int tid = get_global_id(0);
  op_Read(tid);
  op_Compute(tid);
  op_Write(tid);
}
```



```
__attribute__ ((reqd_work_group_size(3,1,1)))
kernel void foo(...) {
  __attribute__((xcl_pipeline_workitems)) {
    int tid = get_global_id(0);
    op_Read(tid);
    op_Compute(tid);
    op_Write(tid);
  }
}
```



Loop pipelining

- Can also be applied on loops to improve throughput

```
__kernel __attribute__ ((reqd_work_group_size(1, 1, 1)))
void vaccum(__global const int *a,
            __global const int *b,
            __global int *result) {
  int tmp = 0;
  __attribute__((xcl_pipeline_loop))
  for (int i=0; i < 32; i++)
    tmp += a[i] * b[i];
  *result = tmp;
}
```

Loop unrolling

```
kernel void
vmult(local int* a, local int* b, local int* c) {
  int tid = get_global_id(0);
  __attribute__((opencl_unroll_hint(2)))
  for (int i = 0; i < 4; i++) {
    int idx = tid*4 + i;
    a[idx] = b[idx]*c[idx];
  }
}
```

- Use more hardware resources for higher throughput
- Also required to have pipelining of loops at outer scope

Program scope global variables using on-chip memories

- **OpenCL 2.0 program-scope global variables**

```
global int g_var0[1024];
global int g_var1[1024];
kernel __attribute__ ((reqd_work_group_size(256,1,1)))
void input_stage (global int *input) {
  __attribute__((xcl_pipeline_workitems)) {
    g_var0[get_local_id(0)] = input[get_local_id(0)];
  }
}
kernel __attribute__ ((reqd_work_group_size(256,1,1)))
void adder_stage(int inc) {
  __attribute__ ((xcl_pipeline_workitems)) {
    int input_data = g_var0[get_local_id(0)];
    int output_data = input_data + inc;
    g_var1[get_local_id(0)] = output_data;
  }
}
kernel __attribute__ ((reqd_work_group_size(256,1,1)))
void output_state(global int *output) {
  __attribute__ ((xcl_pipeline_workitems)) {
    output[get_local_id(0)] = g_var1[get_local_id(0)];
  }
}
```

- **Program-scope arrays ≥ 4096 B allocated on-chip**
- **⇒ Use program-scope array to lower external DDR memory pressure**

Partitioning memories inside of compute units

- **Useful to avoid memory bank access conflict**
  - Remember writing `a[16][17]` instead of `a[16][16]` on GPU or on… Cray 1 (1978)?
- **__local int buffer[16] __attribute__((xcl_array_partition(cyclic,4,1)));**
  - Cyclic partition on 4 memories along dimension 1
- **___local int buffer[16] __attribute__((xcl_array_partition(block,4,1)));**
  - Bloc partition on 4 memories along dimension 1
- **__local int buffer[16] __attribute__((xcl_array_partition(complete,1)));**
  - Complete partitioning along dimension 1: 1 register/element ☺

Asynchronous copy for burst memory transfers

- Cache global memory with local or private memory
- Use OpenCL copy operations to generate more efficient burst transfers

```
__attribute__ ((reqd_work_group_size(1, 1,
1)) kernel
void smithwaterman(global int *matrix,
                   global int *maxIndex,
                   global const char *s1,
                   global const char *s2) {
```

```
// Local memories using BlockRAMs
local char localS1[N];
local char localS2[N];
local int localMatrix[N*N];
async_work_group_copy(localS1, s1, N, 0);
async_work_group_copy(localS2, s2, N, 0);
async_work_group_copy(localMatrix,
                      matrix, N*N, 0);
__attribute__((xcl_pipeline_loop)
for (short index = N;
     index < N * N;
     index++)
  // Use localX instead of X variables
}
```

Improving data path performance with vectorization

- Kernel using only 32-bit DDR memory accesses

```
__kernel __attribute__
((reqd_work_group_size(1, 1, 1)))
void vadd(__global int *a,
          __global int *b,
          __global int *c) {
  for (int i = 0; i < 256; ++i)
    c[i] = a[i] + b[i];
}
```

- Use OpenCL copy operations to generate more efficient burst transfers

```
__kernel __attribute__
((reqd_work_group_size(1, 1, 1)))
void vadd(__global int16 *a,
          __global int16 *b,
          __global int16 *c) {
  for (int i = 0; i < 256/16; ++)
    c[i] = a[i] + b[i];
}
```

Using pipes to optimize dataflow algorithms

- **OpenCL 2.0 pipes: FIFO connecting kernels**
  - Useful to stream data from kernel to kernel
  - Very power and performance efficient on FPGA
    - Direct FIFO without using external memory
  - Only program-scope static pipes supported
  - Blocking pipe extension to avoid spinning (power…)

```
pipe int p0 __attribute__((xcl_reqd_pipe_depth(512)));
pipe int p1 __attribute__((xcl_reqd_pipe_depth(512)));
// Read
kernel __attribute__ ((reqd_work_group_size(256, 1,
1)))
void input_stage(__global int *input) {
  write_pipe_block(p0, &input[get_local_id(0)]);
}
// Transform
kernel __attribute__ ((reqd_work_group_size(256, 1,
1)))
void adder_stage(int inc) {
  int input_data, output_data;
  read_pipe_block(p0, &input_data);
  output_data = input_data + inc;
  write_pipe_block(p1, &output_data);
}
// Write back
kernel __attribute__ ((reqd_work_group_size(256, 1,
1)))
void output_stage(__global int *output) {
  read_pipe_block(p1, &output[get_local_id(0)]);
}
```

- **Implicit assumption: sequential execution of work-items**

Using multiple external memory DDR banks

- **Possible to select different DDR banks to maximize bandwidth**
- **Xilinx vendor extension**
  - Allows to map a buffer on a given
    - XCL_MEM_DDR_BANK0, XCL_MEM_DDR_BANK1, XCL_MEM_DDR_BANK2, XCL_MEM_DDR_BANK3

```
#include <CL/cl_ext.h>
/* This defines
   typedef struct {
     unsigned flags;
     void *obj;
     void *param;
   } cl_mem_ext_ptr_t;
 */
int main(int argc, char *argv[]) {
  [...]
      int a[DATA_SIZE];
      cl_mem input_a;
      cl_mem_ext_ptr_t input_a_ext;
      input_a_ext.flags = XCL_MEM_DDR_BANK0;
      input_a_ext.obj = a; // Initialized from a
      input_a_ext.param = 0; // Reserved for the future
      input_a = clCreateBuffer(context, CL_MEM_READ_ONLY
                                 | CL_MEM_USE_HOST_PTR
                                 | CL_MEM_EXT_PTR_XILINX,
                                sizeof(int)*DATA_SIZE,
                                &input_a_ext, NULL);

      [...]
}
```

Using High-Level Synthesis and RTL kernels

Using HLS C/C++ as OpenCL kernel

- Need to use parameter interface compatible with SDAccel OpenCL

```
void matrix_multiplication(int *a, int *b, int *output) {
#pragma HLS INTERFACE m_axi port=a offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=b offset=slave bundle=gmem
#pragma HLS INTERFACE m_axi port=output offset=slave bundle=gmem
#pragma HLS INTERFACE s_axilite port=a bundle=control
#pragma HLS INTERFACE s_axilite port=b bundle=control
#pragma HLS INTERFACE s_axilite port=output bundle=control
#pragma HLS INTERFACE s_axilite port=return bundle=control
  // Matrices of size 16*16
  const int rank = 16;
  int sum = 0;
  // Cache the external matrices
  int bufa[rank*rank];
  int bufb[rank*rank];
  int bufc[rank*rank];
  memcpy(bufa, a, sizeof(bufa));
  memcpy(bufb, b, sizeof(bufb));
  for (unsigned int c = 0; c < rank; c++) {
    for (unsigned int r = 0; r < rank; r++) {
      sum = 0;
      for (int index = 0; index < rank; index++) {
#pragma HLS pipeline
        int aIndex = r*rank + index;
        int bIndex = index*rank + c;
        sum += bufa[aIndex]*bufb[bIndex];
      }
      bufc[r*rank + c] = sum;
    }
  }
  memcpy(output, bufc, sizeof(bufc));
  return;
}
```

Using anything as an OpenCL kernel

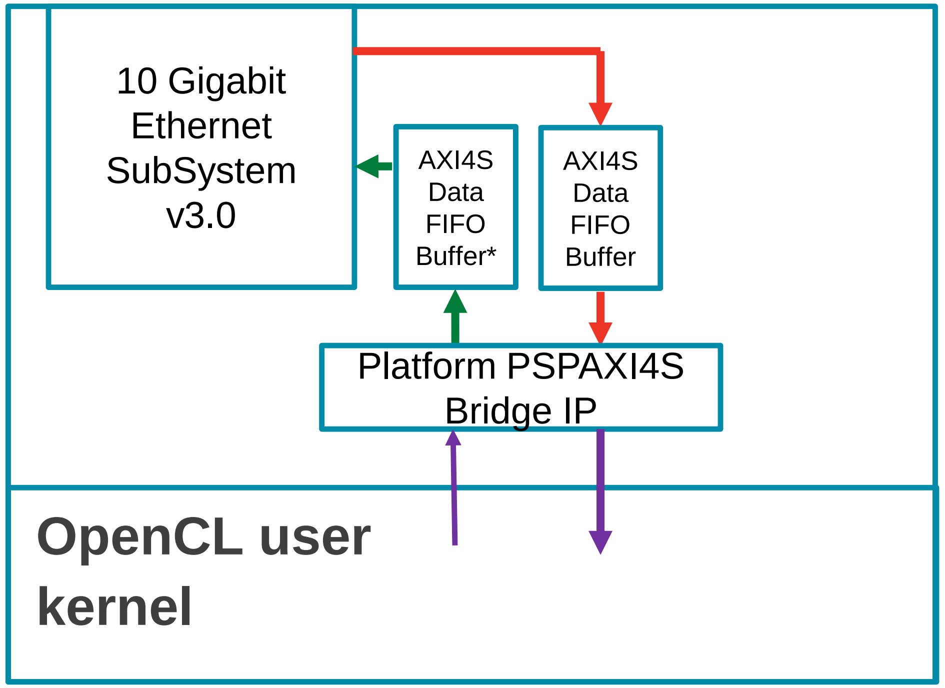
- **Require XML description of kernel interface**

```
<?xml version="1.0" encoding="UTF-8"?>

<root versionMajor="1" versionMinor="0">

<kernel name="input_stage" language="ip"

vlnv="xilinx.com:hls:input_stage:1.0" attributes=""

preferredWorkGroupSizeMultiple="0" workGroupSize="1">

<ports>

<port name="M_AXI_GMEM" mode="master"
range="0x3FFFFFFF" dataWidth="32"

portType="addressable" base="0x0"/>

<port name="S_AXI_CONTROL" mode="slave" range="0x1000"
dataWidth="32"

portType="addressable" base="0x0"/>

<port name="AXIS_P0" mode="write_only" dataWidth="32"
portType="stream"/>

</ports>

<args>

<arg name="input" addressQualifier="1" id="0"
port="M_AXI_GMEM"

size="0x4" offset="0x10" hostOffset="0x0"
hostSize="0x4" type="int*" />

<arg name="__xcl_gv_p0" addressQualifier="4" id=""
port="AXIS_P0"

size="0x4" offset="0x18" hostOffset="0x0"
hostSize="0x4" type=""

memSize="0x800"/>

</args>

</kernel>

<pipe name="xcl_pipe_p0" width="0x4" depth="0x200"
linkage="internal"/>

<connection srcInst="input_stage" srcPort="p0"
dstInst="xcl_pipe_p0"

dstPort="S_AXIS"/>

</root>
```

Use High-Speed I/O

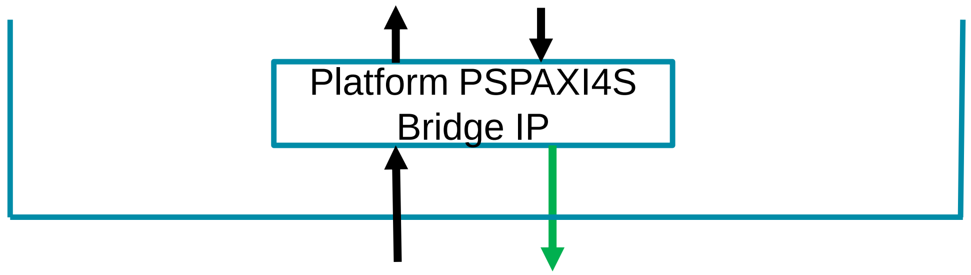
Using 10 Gb/s Ethernet as OpenCL 2.0 pipes

10 Gigabit Ethernet Subsystem v3.0
http://www.xilinx.com/support/documentation/ip_documentation/axi_10g_ethernet/v3_0/pg157-axi-10g-ethernet.pdf



Available through Ethernet DSA : xilinx_adm-pcie-7v3_1ddr-1eth_2_0.dsa

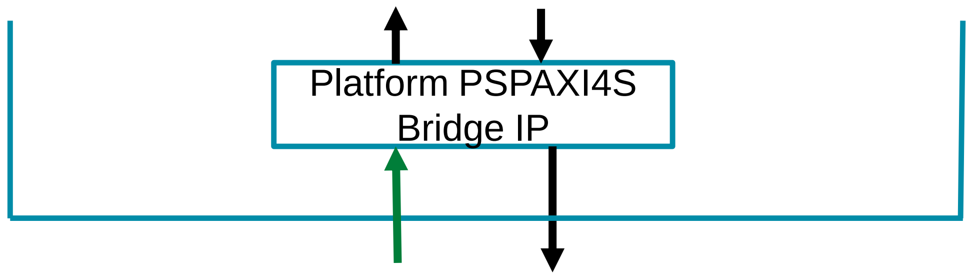
RX OpenCL 2.0 pipe



RX pipe bits	Width	Purpose
tdata[63:0]	64	10GE subsystem tdata [data beat]
tdata[71:64]	8	10GE subsystem tkeep [byte enables]
tdata[72]	1	10GE subsystem tuser [good/bad packet]
tdata[73]	1	10GE tlast [framing]
tdata[127:74]	54	Unused

RX is 128-bit Pipe with framing "in-band"

TX OpenCL 2.0 pipe



RX pipe bits	Width	Purpose
tdata[63:0]	64	10GE subsystem tdata [data beat]
tdata[71:64]	8	10GE subsystem tkeep [byte enables]
tdata[72]	1	10GE subsystem tuser [explicit underrun (abort)]
tdata[73]	1	10GE tlast [framing]
tdata[127:74]	54	Unused

TX is 128-bit pipe with framing “in-band”

Conclusion

Performance check list

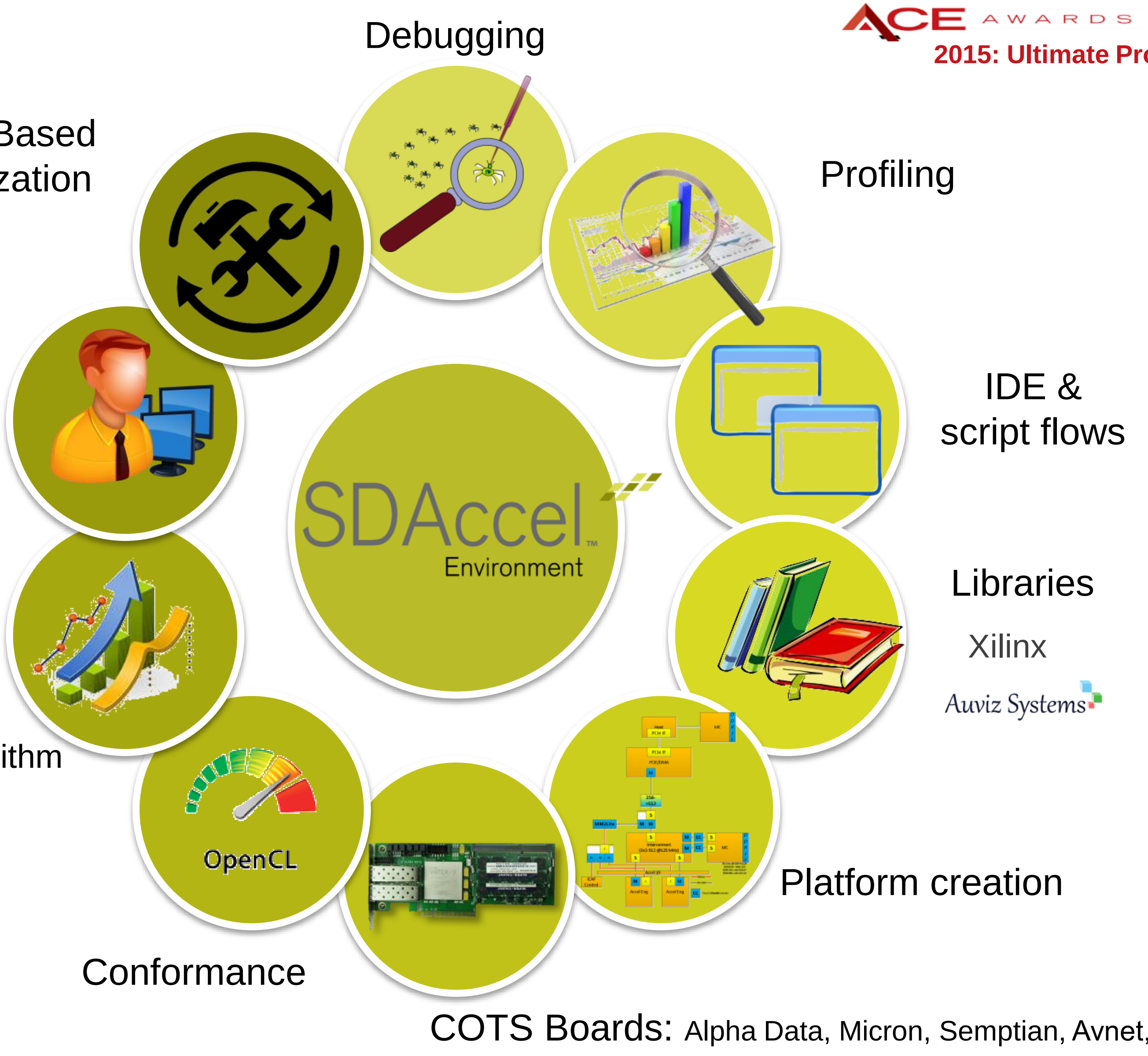
- Verify functional correctness ☺
- Use profiling summary
- Use timeline trace
- Use pipelining
- Unroll loops
- Use burst data transfers
- Isolate data transfer from computation
- Use local and private memories for burst and scratch pads
- Use on-chip global memories
- Use on-chip pipes
- Use work-groups
- Use multiple memory ports
- Use the entire port width

Conclusion

- **Modern FPGAs are complex MP-SoC with programmable logic, CPU, GPU, peripherals, high-performance I/O… → the ultimate accelerator ☺**
- **SDAccel: OpenCL for host-accelerator style programming on Xilinx FPGA**
  - Add specific optimizations for FPGA power & efficiency trade-offs
    - Optimized data sizes (DNN with very small data size compared to CPU/GPU…)
    - #CU
    - Pipelining
    - Bus
    - Several different memories for better locality and bandwidth
    - Hardware pipes
    - Specific interfaces & IO…
- **See poster "OpenCL meets Open Source Streaming Analytics", Robin Grosman (Huawei)**
- ***En route* for the GP-FPGA revolution like we had the GP-GPU!**
- **Possible to have kernels in HLS C/C++, VHDL, Verilog for demanding users and I/O…**
- **http://www.xilinx.com/support/documentation-navigation/development-tools/software-development/sdaccel.html**

SDAccel: Award winning Whole Product

2015: Ultimate Products



Applications & Benchmarks

- **Machine Learning**: Image classification (**Auviz**), Vehicle prediction (**MulticoreWare**)
- **Security**: SHA1 encryption algorithm
- **Genomics**: Smithwaterman gene sequencing algorithm
- **Video**: FFMPEG video scaling acceleration on IBM Power8
- **Imaging**: FAST image feature extraction algorithm (**ArrayFire**)

XILINX

ALL PROGRAMMABLE™