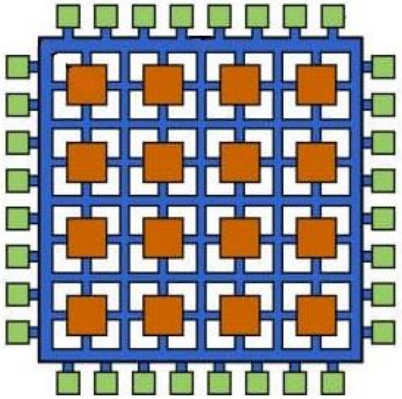
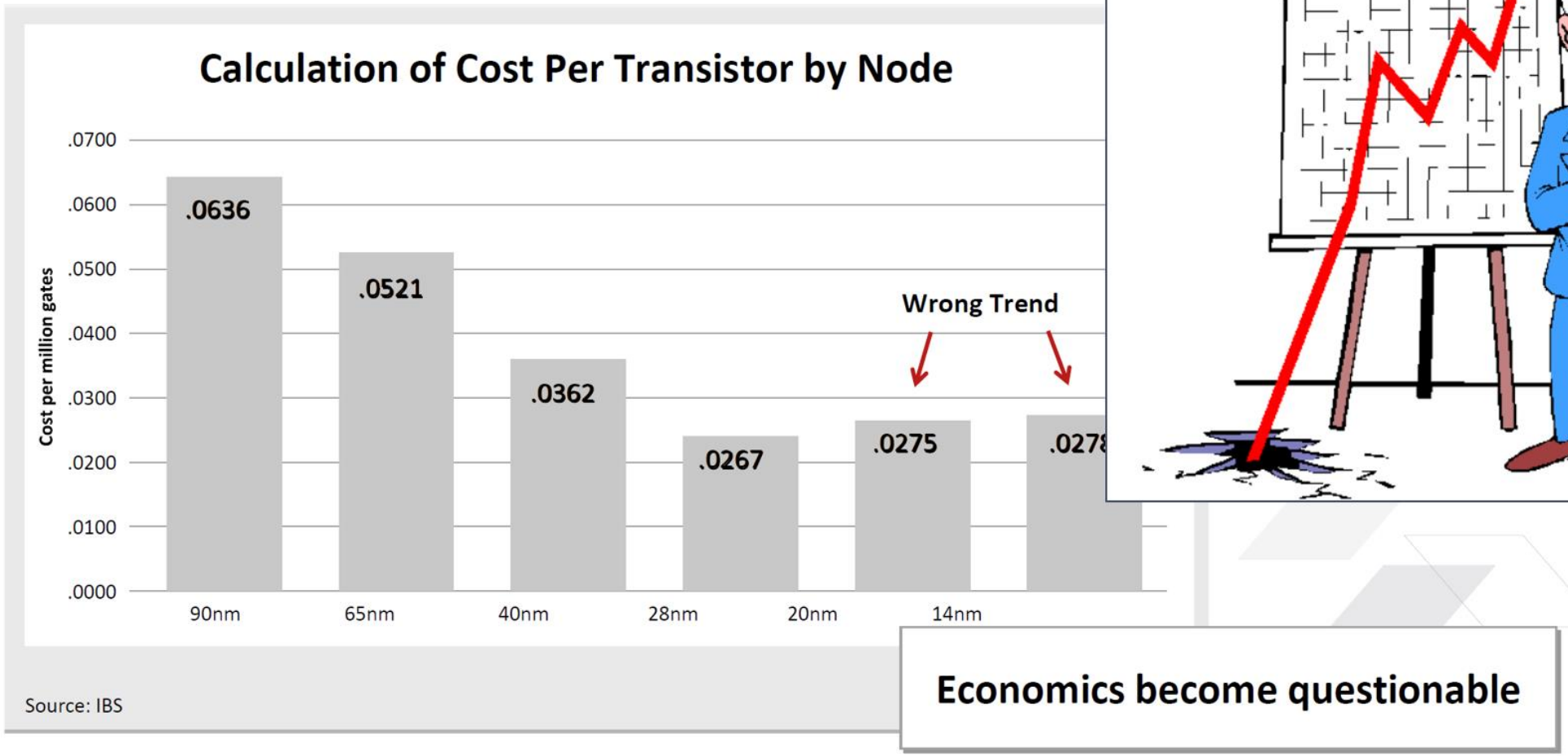
Part A.

Parallel programming on FPGAs using High-level Synthesis



Aggelos Ferikoglou
Dimitrios Danopoulos

End of Moore's Law



Economics become questionable

High-Performance Hardware

- 14 nm Intel Tri-Gate
  - 1 GHz
- 10 TF single precision
- 5.5M Logic Elements
  - 4-input LUT, register, carry, etc.
- Block RAM: 28.6 MiB
- Hardened DRAM controller DDR4
- Various options for memory

- 16 nm FinFET
  - ~600 MHz
- 6,840 DSPs (3.1 TF single prec.)

  2.5M Logic Elements
  - 1,182,000 5-input LUTs
  - 2,364,000 FFs
- Block RAM: 9.1 MiB
- TDP: 95 W (Amazon F1)

- 12 nm
  - 1455 MHz
- 5,120 cores (15.7 TF single prec.)
  - CUDA programming
- On-chip memory:
  - Registers: 20.8 MiB
  - L1/SM: 7.7 MiB
  - L2 Cache: 6.1 MiB
- TDP: 300W

FPGA are in the cloud for a while..

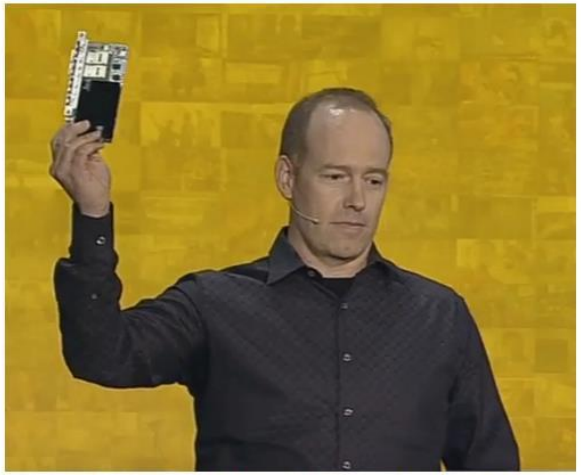
Microsoft Goes All in for FPGAs to Build Out AI Cloud

Michael Feldman | September 27, 2016 08:42 CEST

Software giant bets the (server) farm on reconfigurable computing

Microsoft has revealed that Altera FPGAs have been installed across every Azure cloud server, creating what the company is calling "the world's first AI supercomputer." The deployment spans 15 countries and represents an aggregate performance of more than one exa-op. The announcement was made by Microsoft CEO Satya Nadella and engineer Doug Burger during the opening keynote at the Ignite Conference in Atlanta.

The FPGA build-out was the culmination of more than five years of work at Microsoft to find a way to accelerate machine learning and other throughput-demanding applications and services in its Azure cloud. The effort began in earnest in 2011, when the company launched Project Catapult, the R&D initiative to design an acceleration fabric for AI services and applications. The rationale was that CPU evolution, a la Moore's Law, was woefully inadequate in keeping up with the demands of these new hyperscale applications. Just as in traditional high performance computing, multicore CPUs weren't keeping up with demand.



Doug Burger with Microsoft-designed FPGA card

Amazon Adds FPGA Instance to Public Cloud

Michael Feldman | December 7, 2016 07:17 CET

In a blog post published last week, Amazon Web Services Chief Evangelist Jeff Barr announced a new Elastic Compute Cloud (EC2) instance, known as the F1, which incorporates Xilinx FPGAs. The web giant says users will not only be able to build FPGA-accelerated applications for their own purposes with the new instance, but also resell them in the AWS Marketplace to third parties.

Barr makes the conventional pitch for FPGA acceleration, noting its inherent advantage in implementing parallel computation much more efficiently than general-purpose chips by being able to implement an application's dataflow at the level of the logic elements. Writes Barr:

"This highly parallelized model is ideal for building custom accelerators to process compute-intensive problems. Properly programmed, an FPGA has the potential to provide a 30x speedup to many types of genomics, seismic analysis, financial risk analysis, big data search, and encryption algorithms and applications."

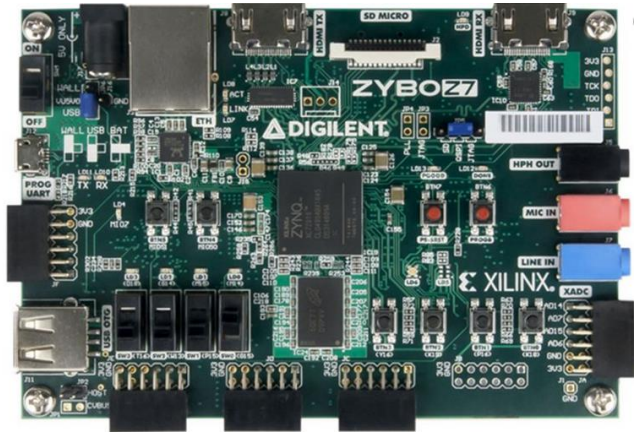
The F1 instance is equipped with Xilinx's Virtex Ultrascale+ VU9P, a 16nm device that contains about 2.5 million logic elements and over 6,800 DSP engines. The F1's host CPU is Intel's 18-core Broadwell E5 2686 v4 processor, which runs at 2.3 GHz. The instance may be configured with up to 8 FPGAs, 967 GiB of memory and 4 TB of NVMe flash storage. The PCIe fabric allows multiple FPGAs to communicate with one another directly, as well as share the same memory space.

Application developers will have access to the Xilinx Vivado Design Suite free of charge, but that will require that the application code be implemented in VHDL or Verilog. Third-party tools can also be used, including higher level frameworks like OpenCL, but that means users will be responsible for their own development environments.

At this point, FPGA application development is geared toward a rather niche audience, so this will hardly be a volume business for Amazon in the near-term. HPC cloud specialist Nimbix recently added Xilinx FPGAs to its own infrastructure in the hopes of building a customer base of FPGA computing enthusiasts. Much of the initial customer base will be Xilinx engineers themselves, who will be developing and testing software on their own product.

How to program FPGAs?

- **VHDL**: a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits.
- **Verilog**: another hardware description language standardized as IEEE 1364, used to model electronic systems.
- **LabVIEW** : LabVIEW utilizes the basic LabVIEW graphical interface but employs additional tools to enable it to provide the functionality required for programming FPGAs.
- **HLS**: High level synthesis (HLS), also known as algorithmic synthesis, is a design process in which a high level, functional description of a design (in C/C++) is automatically compiled into a RTL implementation that meets certain user specified design constraints.



HDL example (System Verilog)

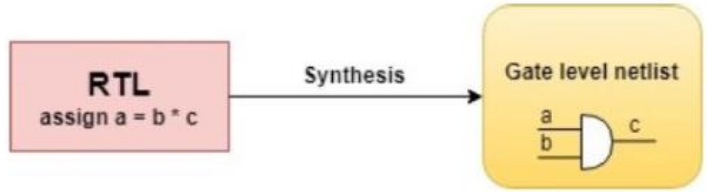
```
module exercise(
        start, clk, ready, rst
    );

    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
```

RTL (register transfer level) has building blocks which are adders, multipliers, flip-flops, etc.

Need to handle explicitly sequential logic signals

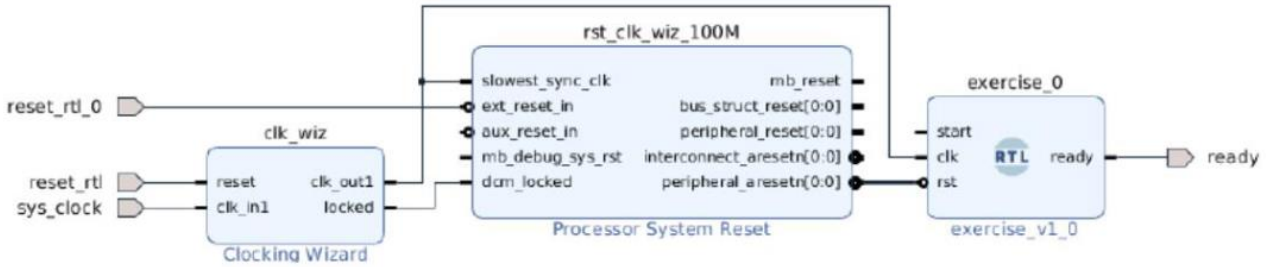
- Registers
- Flip-flops
- Control signals (e.g., reset)
- Clock

Synthesis is the process from which we obtain a gate-level netlist from our RTL description of the hardware



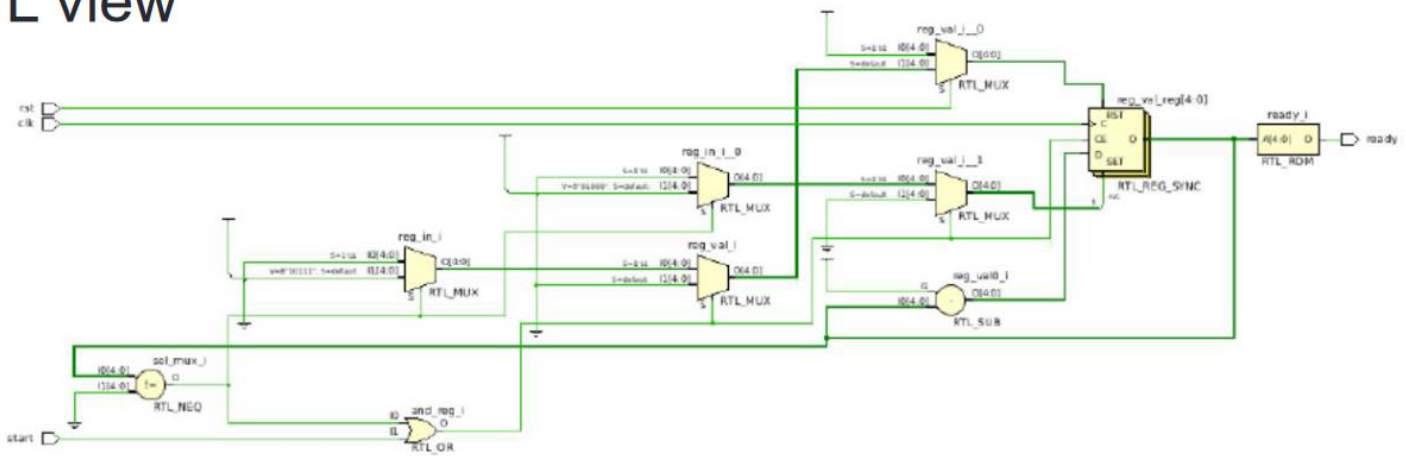
HDL example (System Verilog)

- HDL view



Hard and time consuming!

- RTL view



High-level synthesis (HLS)

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

- Xilinx and Intel (and other vendors) offer **HLS tools** for FPGA design:
  - C/C++/OpenCL code is transformed to the spatial paradigm in HW
  - Programming from the bit level to the word/datatype level

SDSoC™ Environment

bluespec

XILINX VITIS™

Microsoft Catapult

impulse accelerated technologies

High-level synthesis (HLS)

```
#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;
}
```

HLS characteristics:

- HW abstraction via a high-level language (C/C++)
- Usually repetitive C/C++ functions (for-loops) are mapped for HW
- Not all C/C++ constructs/functions can be used!
- Some non C/C++ standards can be used such as:
  - Custom data types (ap_uint, ap_fixed, ...)
  - Keywords (volatile,static,...)

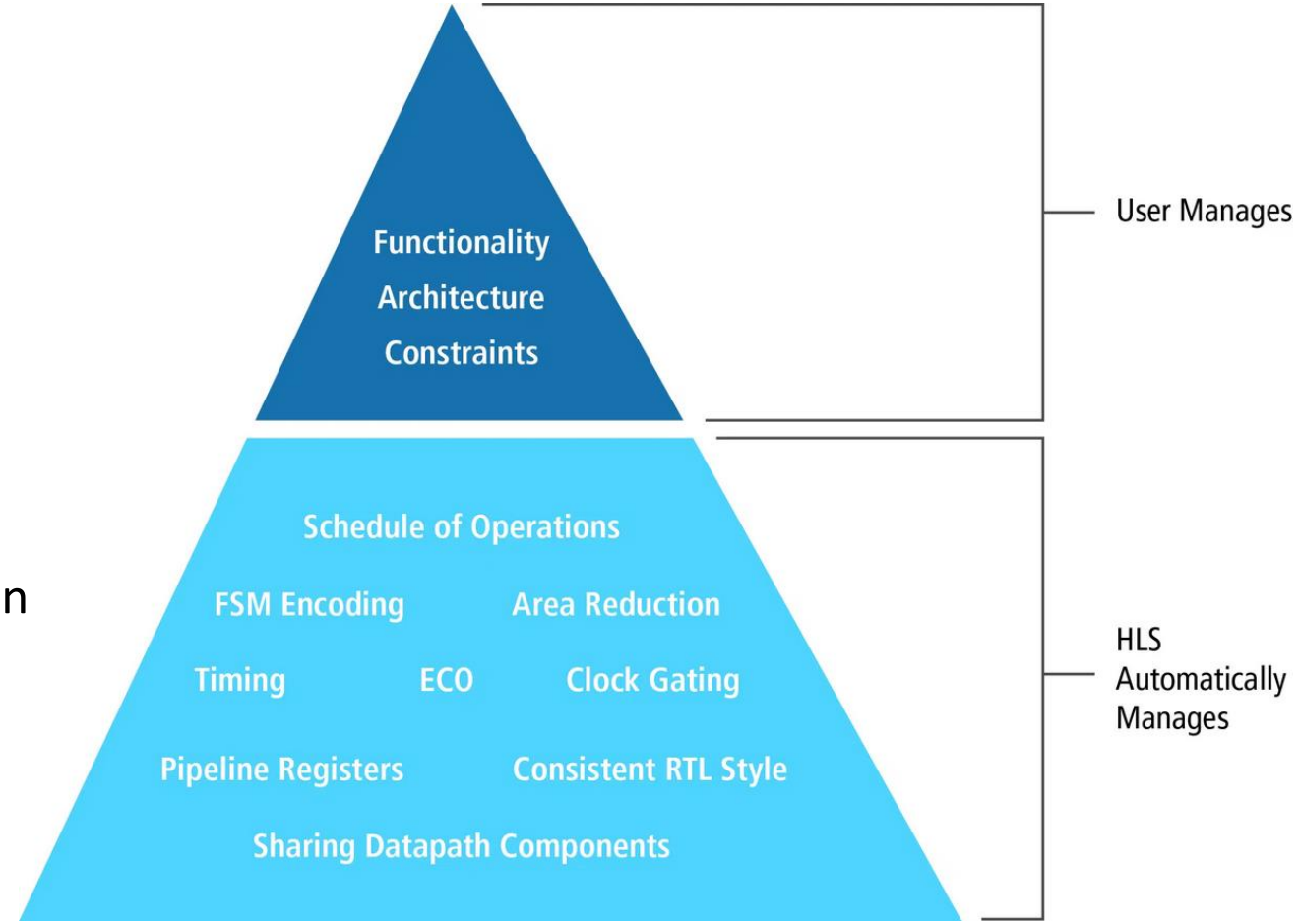
HLS directives:

We guide the compiler how to synthesize HW
with the use of “#pragmas” (i.e. #pragma HLS pipeline, unroll, etc.)

High-level synthesis (HLS)

Main benefits:

- ✓ Productivity: lower design complexity and faster simulation speed
- ✓ Portability: single source
  →multiple implementations, code re-use
- ✓ Faster Results: rapid design space exploration
  →higher quality of result (QoR)

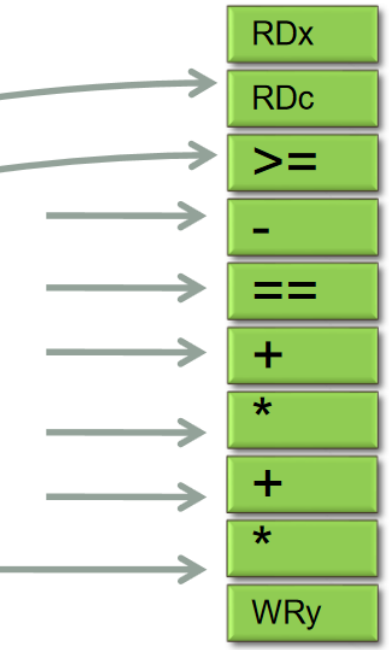


HLS operator types

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



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).

From any C code example ...

Operations are extracted…

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

HLS Functions and RTL hierarchy

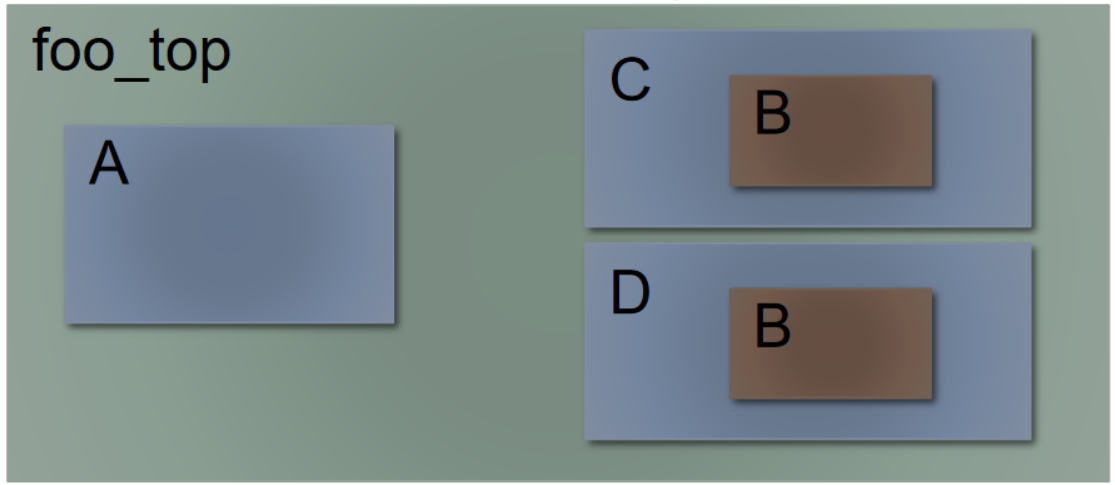
- Each function is translated into an RTL block
  - Verilog module, VHDL entity

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

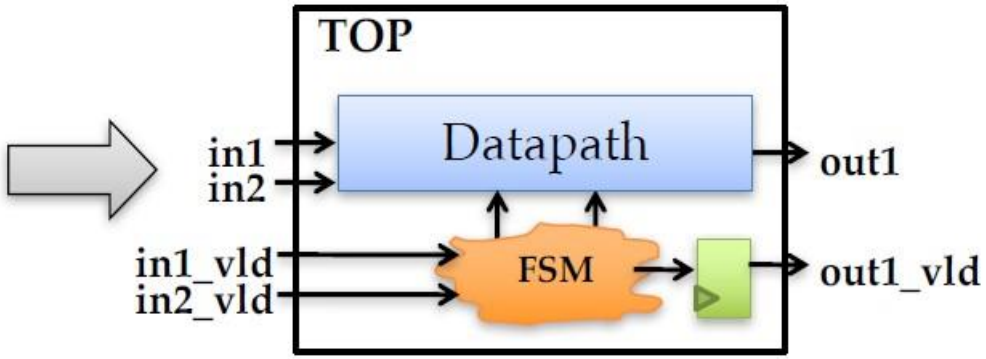
  - By default, each function is implemented using a common instance
  - Functions may be inlined to dissolve their hierarchy
    - Small functions may be automatically inlined

HLS Function arguments

Function arguments become ports on the RTL blocks

- Usually are pointers to arrays or scalars

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



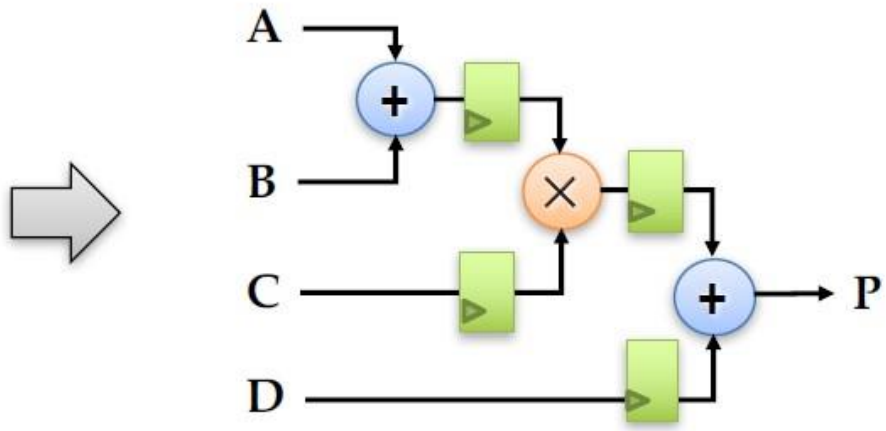
Input/output (I/O) protocols (axi_dma, etc.)

- They allow RTL blocks to synchronize data exchange

HLS expressions

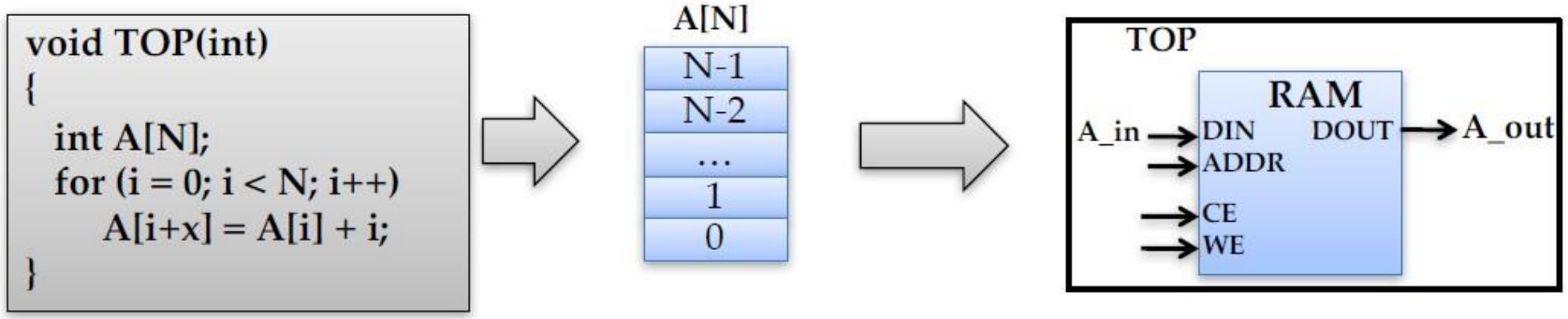
- In HLS we can write math expressions.
  - It generates datapath circuits from them
  - Timing constraints influence the use of registers

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

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



HLS Memory extraction

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



- An array can be partitioned into individual elements and mapped to registers
- This can enable parallel access

HLS pragmas

The HLS compiler provides pragmas that can be used to optimize the design:

- Reduce latency
- Improve throughput performance
- Reduce area and device resources
- Control the I/O ports of the kernels

```
void mmult (float A[N*N], float B[N*N], float C[N*N])
{
     float Abuf[N][N], Bbuf[N][N];
#pragma HLS array_partition variable=Abuf block factor=16 dim=2
#pragma HLS array_partition variable=Bbuf block factor=16 dim=1

     for(int i=0; i<N; i++) {
          for(int j=0; j<N; j++) {
#pragma HLS PIPELINE
               Abuf[i][j] = A[i * N + j];
               Bbuf[i][j] = B[i * N + j];
          }
     }

     for (int i = 0; i < N; i++) {
          for (int j = 0; j < N; j++) {
#pragma HLS PIPELINE
               float result = 0;
               for (int k = 0; k < N; k++) {
                    float term = Abuf[i][k] * Bbuf[k][j];
                    result += term;
               }
               C[i * N + j] = result;
          }
     }
}
```

HLS pragmas - Loop Unrolling

```
#pragma HLS unroll [factor=N]
```

Placed at the start of each loop

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

- **Pros**
  - Decrease loop overhead
  - Increase **parallelism** for scheduling
- **Cons**
  - Increase operation count, which may negatively impact *area*, *power*, and *timing*

```
int sum = 0;
for(int i = 0; i < 10; i++) {
#pragma HLS unroll factor=2
    sum += a[i];
}
```

unroll with a factor of 2

```
int sum = 0;
for(int i = 0; i < 10; i+=2) {
    sum += a[i];
    sum += a[i+1];
}
```

HLS pragmas - Loop Pipelining

```
#pragma HLS pipeline [II=N]
```

Placed at the start of each loop

Loop pipelining is the most important optimization in HLS. It allows a new iteration to begin processing before the previous iteration is complete.

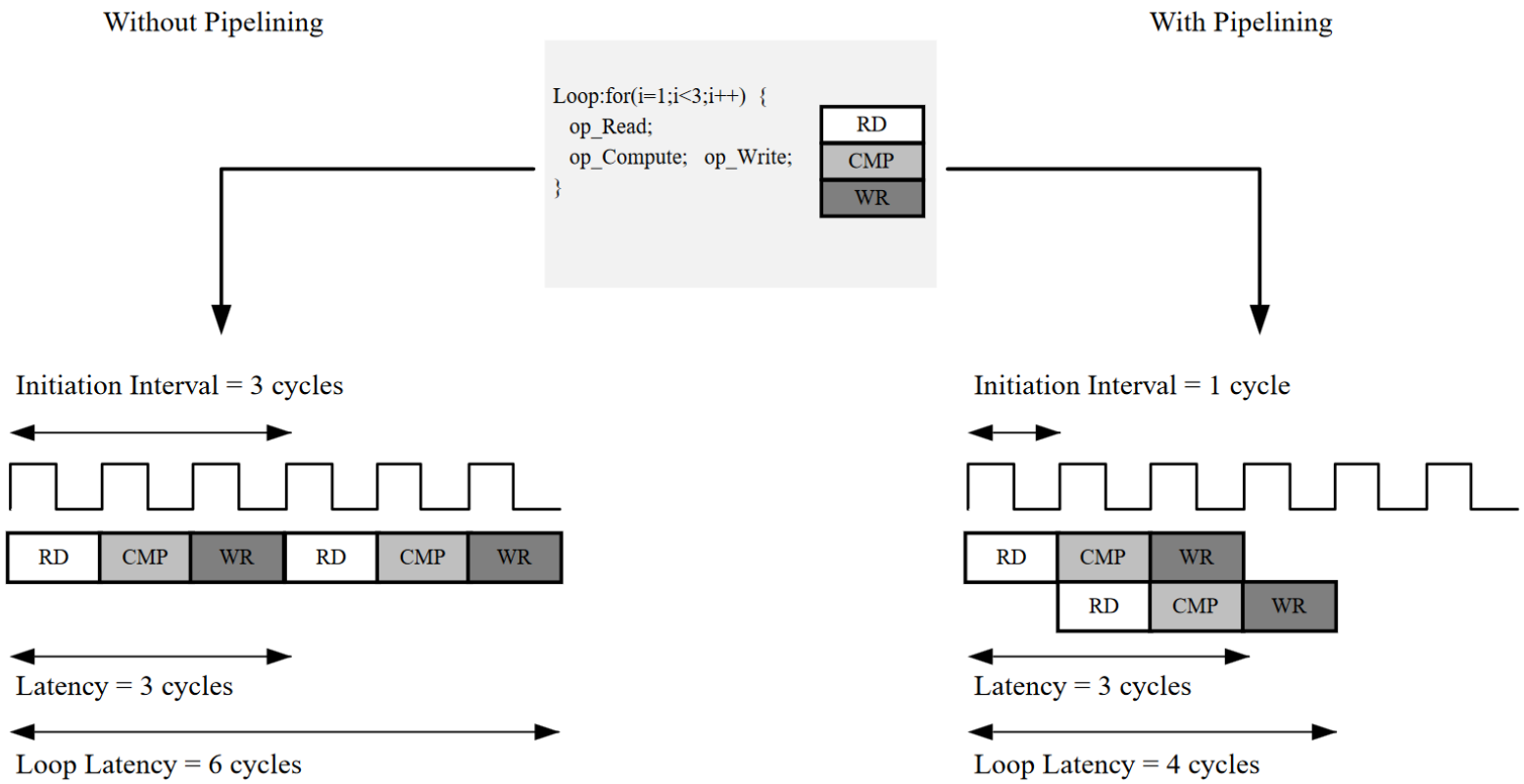
Key metric: **Initiation Interval (II)** expressed in number of cycles. It has many different meanings:

1. No. of cycles before we can accept new inputs
2. Longest delay between new input becoming available.
3. Factor slowdown of your application

```
void madd(float A[N*N], float B[N*N], float C[N*N])
{
  int i, j;

  for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
#pragma HLS PIPELINE II=1
      C[i*N+j] = A[i*N+j] + B[i*N+j];

}
```

HLS pragmas - Loop Pipelining

```
#pragma HLS pipeline [II=N]
```



If there are no data dependencies between loops we can have II=1 (ideal)

HLS pragmas – Increasing memory bandwidth

```
#pragma HLS partition variable=<variable> <block, cyclic, complete> factor=<int> dim=<int>
```

Arrays can be **partitioned** in smaller arrays. Memories have only a limited number of read ports and write ports, which can limit the throughput of a load/stores. The memory bandwidth is improved by splitting up the original array into multiple smaller effectively increasing the number of load/store ports.

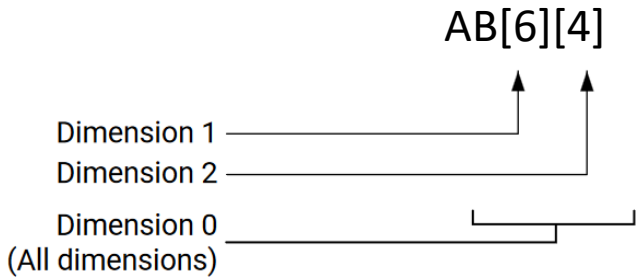
HLS provides three types of array partitioning:

- **block**: The original array is split into equally sized blocks of consecutive elements of the original array.
- **cyclic**: The original array is split into equally sized blocks interleaving the elements of the original array.
- **complete**: The default operation is to split the array into fully to its individual elements.

Example:

```
#pragma HLS array_partition variable=AB block factor=2 dim=2
```

Partitions dimension two of the two-dimensional array, AB[6][4] into two new arrays of dimension [6][2]

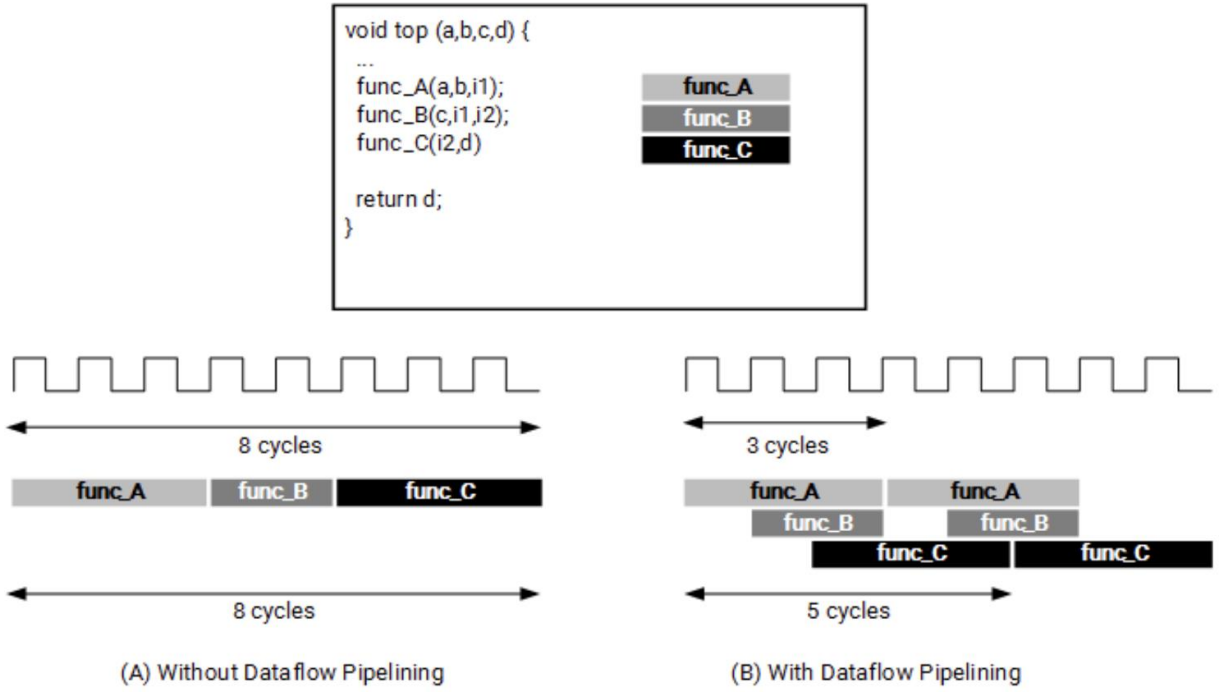


HLS pragmas – Task level pipelining

```
#pragma HLS dataflow
```

Placed before calling several loops/functions

The **DATAFLOW** pragma enables task-level pipelining, allowing functions or loops to start operation before the previous function or loop completes all its operations.



HLS pragmas – Controlling resources

```
#pragma HLS allocation instances=<list> limit=<value> <type>
```

Placed at start of functions or loops.

HLS **allocation** pragma specifies instance restrictions to limit resource allocation in the implemented kernel. This can limit the number of DPSs, LUTs or functions implemented in RTL.

Example 1: Limits the number of multiplier operators used in the implementation of the function my_func to 1.

```
void my_func(data_t angle) {
#pragma HLS allocation instances=mul limit=1 operation
...
}
```

Example 2: Given a design with multiple instances of function foo, this example limits the number of instances of foo in the RTL kernel to 2.

```
#pragma HLS allocation instances=foo limit=2 function
```

HLS pragmas – More accurate analysis

```
#pragma HLS loop_tripcount min=<int> max=<int> avg=<int>
```

Placed at start of a loop

HLS **tripcount** pragma can be applied to a loop to manually specify the total number of iterations performed by a loop. It sometimes helps HLS tool for analysis only, and does not impact the results of synthesis.

Example: In this example loop_1 in function foo is specified to have a minimum tripcount of 12 and a maximum tripcount of 16:

```
void foo (num_samples, ...) {
  int i;
  ...
  loop_1: for(i=0;i< num_samples;i++) {
   #pragma HLS loop_tripcount min=12 max=16
   ...
    result = a + b;
  }
}
```

HLS Optimization methodology

Baseline Hardware Function	• Determine performance with compiler defaults
1: Optimization for Metrics	• Define loop trip counts
2: Pipeline for Performance	• Pipeline and Dataflow
3: Optimize Structures for Performance	• Partition memories • Remove false dependencies
4: Reduce Latency	• Optionally specify latency requirements
5: Improve Area	• Optionally recover resources through sharing

```
void mmult (float A[N*N], float B[N*N], float C[N*N])
{
     float Abuf[N][N], Bbuf[N][N];
#pragma HLS array_partition variable=Abuf block factor=16 dim=2
#pragma HLS array_partition variable=Bbuf block factor=16 dim=1

     for(int i=0; i<N; i++) {
          for(int j=0; j<N; j++) {
#pragma HLS PIPELINE
               Abuf[i][j] = A[i * N + j];
               Bbuf[i][j] = B[i * N + j];
          }
     }

     for (int i = 0; i < N; i++) {
          for (int j = 0; j < N; j++) {
#pragma HLS PIPELINE
               float result = 0;
               for (int k = 0; k < N; k++) {
                    float term = Abuf[i][k] * Bbuf[k][j];
                    result += term;
               }
               C[i * N + j] = result;
          }
     }
}
```

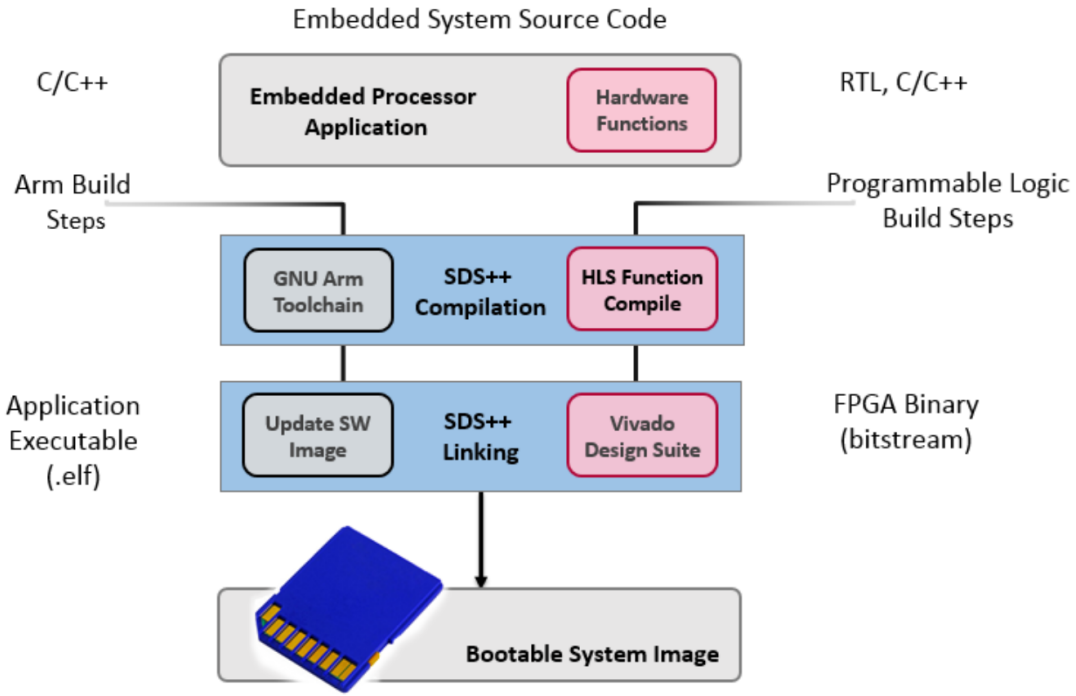
SDSoC

SDSoC is an Eclipse based development environment that you can accelerate C/C++ application for embedded Xilinx FPGAs.

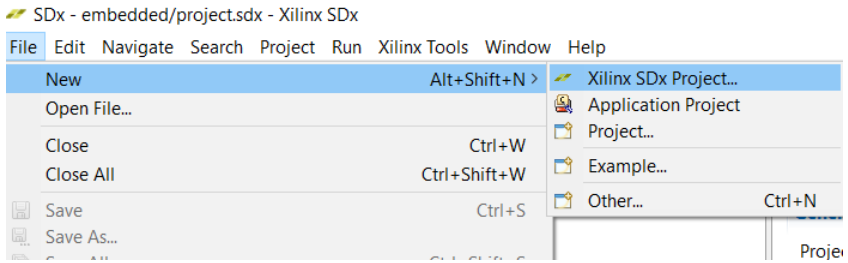
It provides:

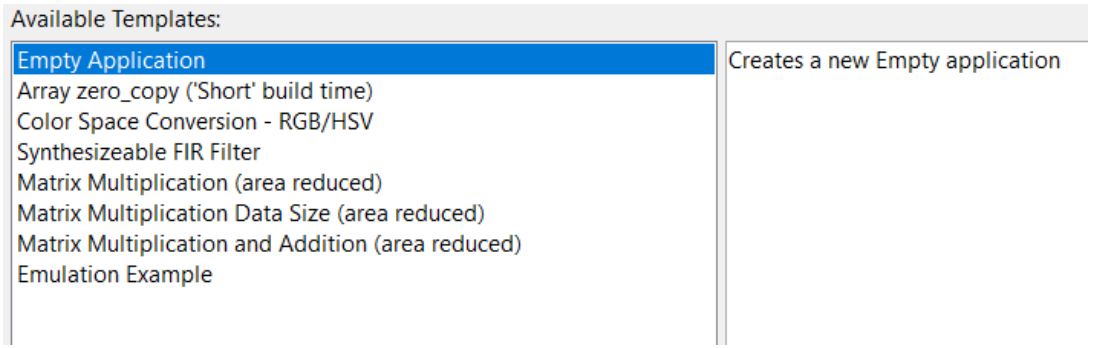
- System level profiling, debugging, and analysis
- Acceleration in programmable logic using HLS
- Hardware pre-compiled libraries for math and computer vision
- Supports bare metal, Linux and FreeRTOS as target OS

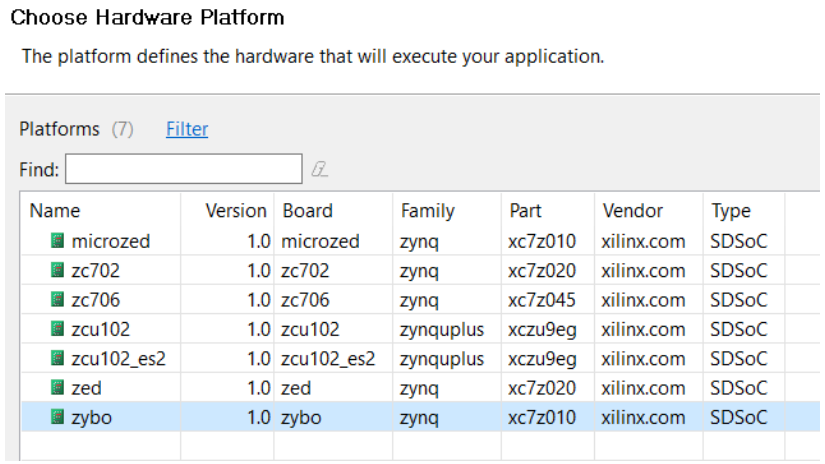
Figure 2: **SDSoC Build Process**

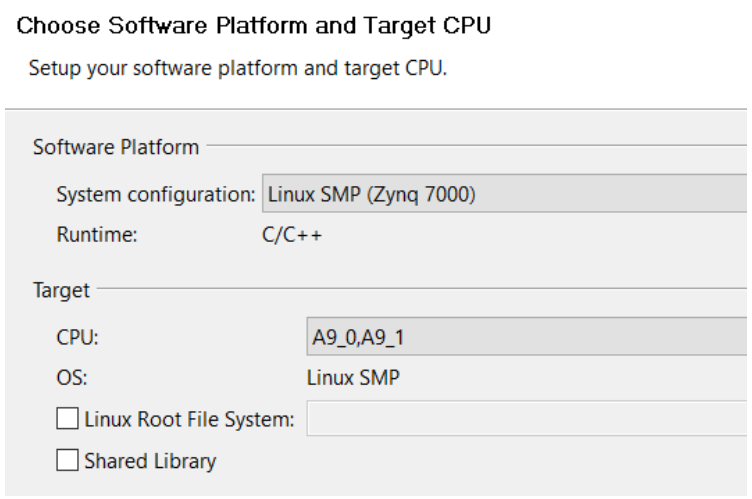


SDSoC – Create Project



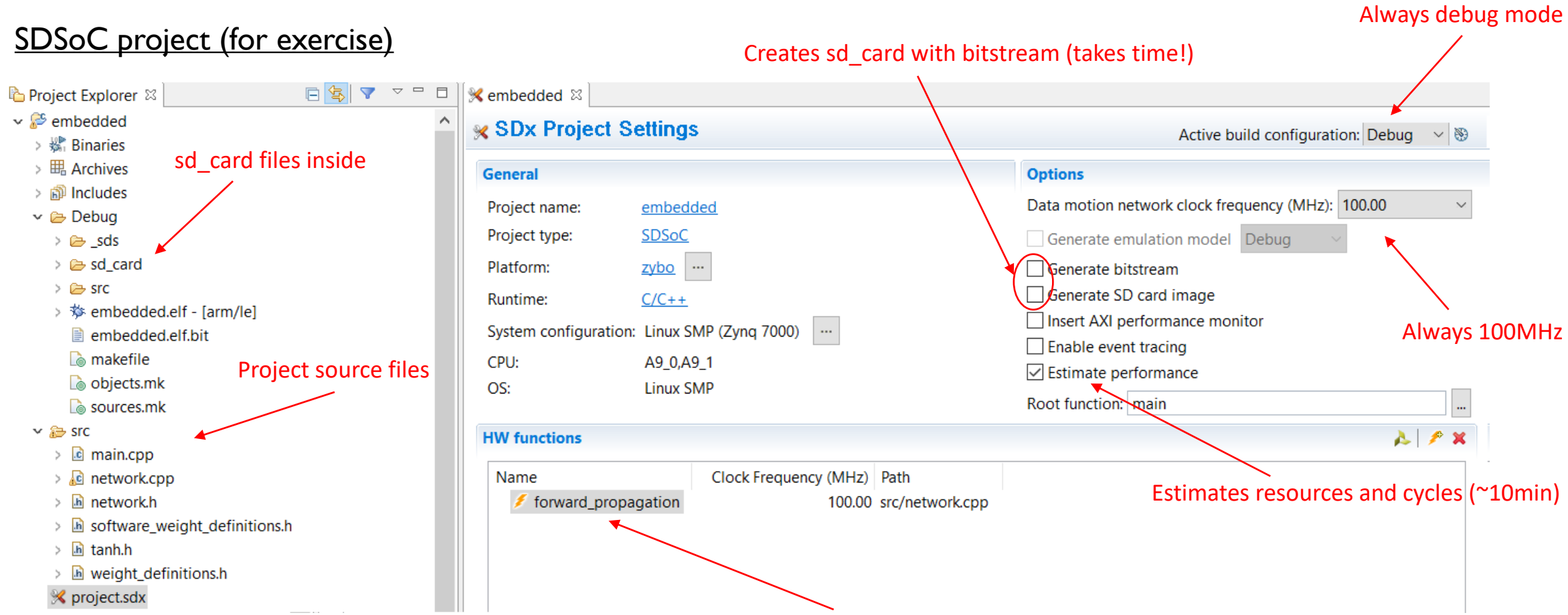






SDSoC - Tooflow

SDSoC project (for exercise)



Useful links:

SDSoC Intro Tutorial:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug1028-sdsoc-intro-tutorial.pdf

SDSoC Optimization Guide:
https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_4/ug1235-sdsoc-optimization-guide.pdf

HLS pragmas:
https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/okr1504034364623.html