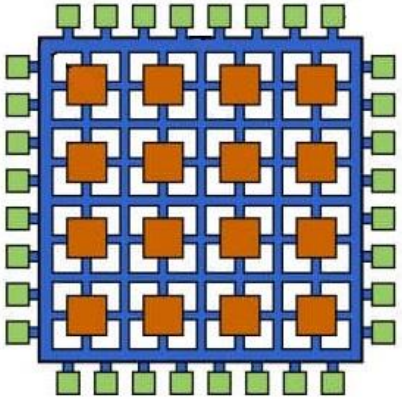
Part A.

Programming on FPGAs using High-level Synthesis



Aggelos Ferikoglou, Dimitrios Danopoulos, Sotirios Xydis

HLS: What and Why

- What
  - *Automated* design process that transforms a **high-level functional specification to optimized register-transfer level (RTL)** descriptions for efficient hardware implementation

The CMU Design Automation System

An Example of Automated Data Path Design

A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Leive, J. Kim

Carnegie-Mellon University
Departments of Electrical Engineering and Computer Science
Pittsburgh, Pennsylvania 15213

Abstract

This paper illustrates the methodology of the CMU Design Automation System by presenting an automated design of the PDP-8/E data paths from a functional description. This automated design (using synthesis techniques) is compared both to DEC's implementation and the Intersil single chip implementation.

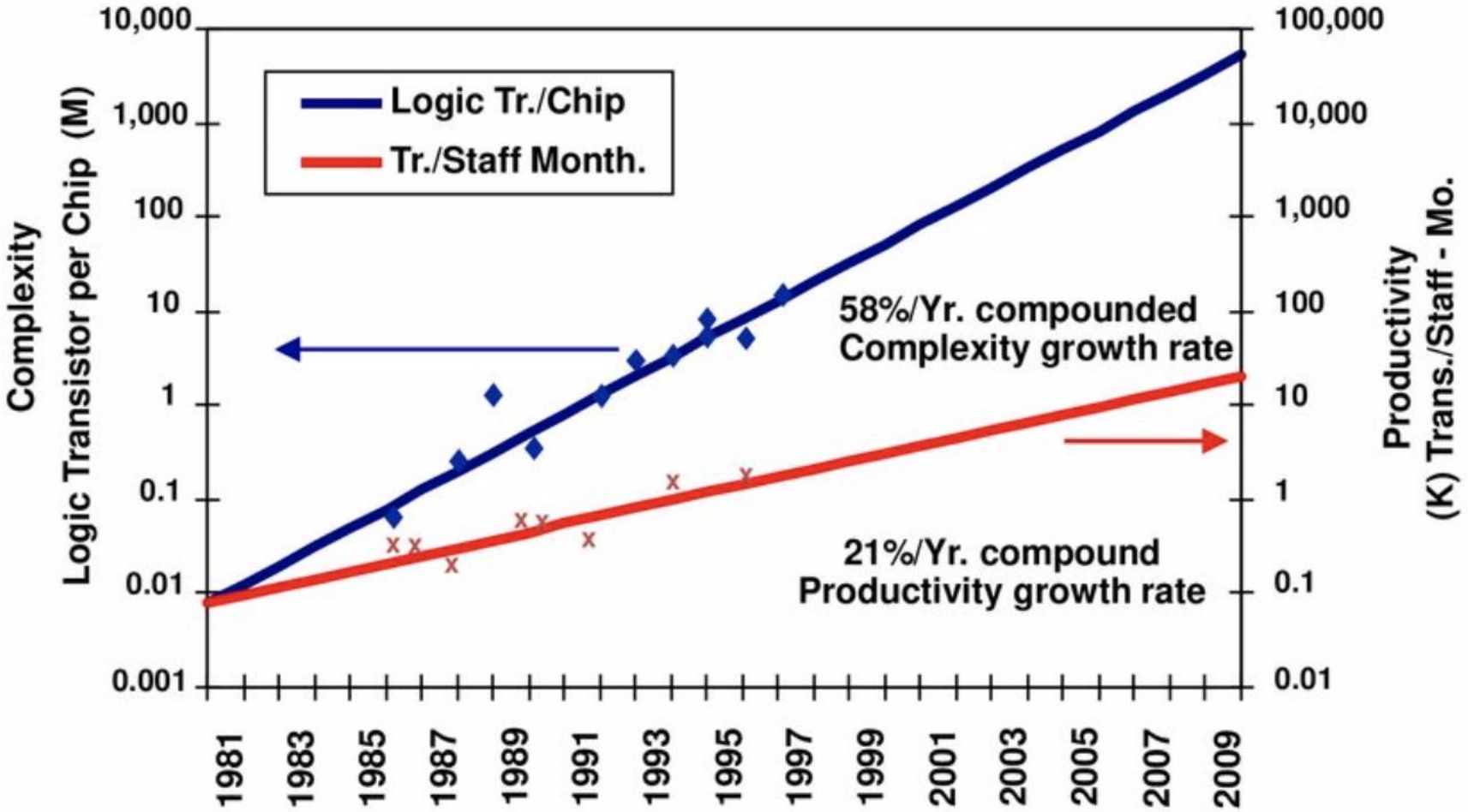
1. Introduction

As it is becoming possible to integrate larger numbers of logic components on a single chip, the need for more powerful design aids is becoming apparent. Indeed, these aids must be capable of supporting a designer from the system level of design down to the mask level. In this way the systems level designer can become more aware of the implications of higher-level design tradeoffs on implementation properties such as silicon area, power consumption, testability, and speed, and be able to make more timely use of new technologies. The ultimate goal of the Carnegie-Mellon University Design Automation (CMU-DA) System [12] is to provide a technology-relative, structured-design aid to help the

The CMU-DA system differs from other design automation systems because the input design description is a functional specification. Such a specification provides a model that, while accurately characterizing the input-output behavior desired for the implementation, does not necessarily specify its internal structure. The system software collectively performs the synthesis function by transforming the input functional description into a structural description. The design process involves binding implementation decisions in a top-down manner as a design proceeds through the design system. More structural decisions are made at each level until a complete hardware specification is obtained, with the most influential design trade offs being performed first in order to cut down the design search space.

The purpose of this paper is to illustrate the methodology of the CMU-DA system. The results given here are worst case - many optimizations which are straightforward have not been implemented yet; research is in progress on others. The design of the data part of a DEC PDP-8/E [5] from the ISP level through to a TTL and standard cell design will be discussed. Only the subset of the full DA system which is presently implemented has been used for this example. The

Design – Productivity Gap

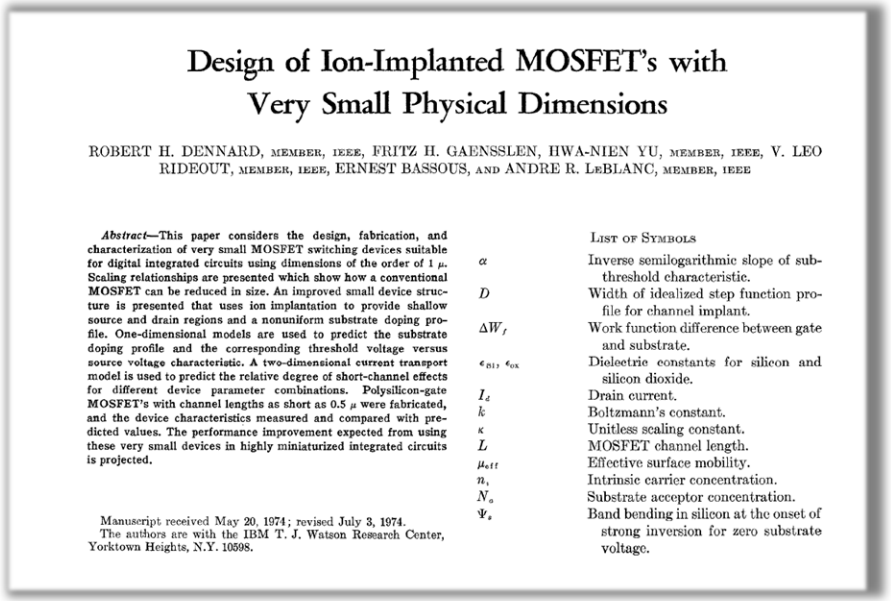


Complexity outpaces design productivity

Introduction – Energy Scaling

✓Moore & Dennard: industry roadmap for 40 years


Cramming more components onto integrated circuits


Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions

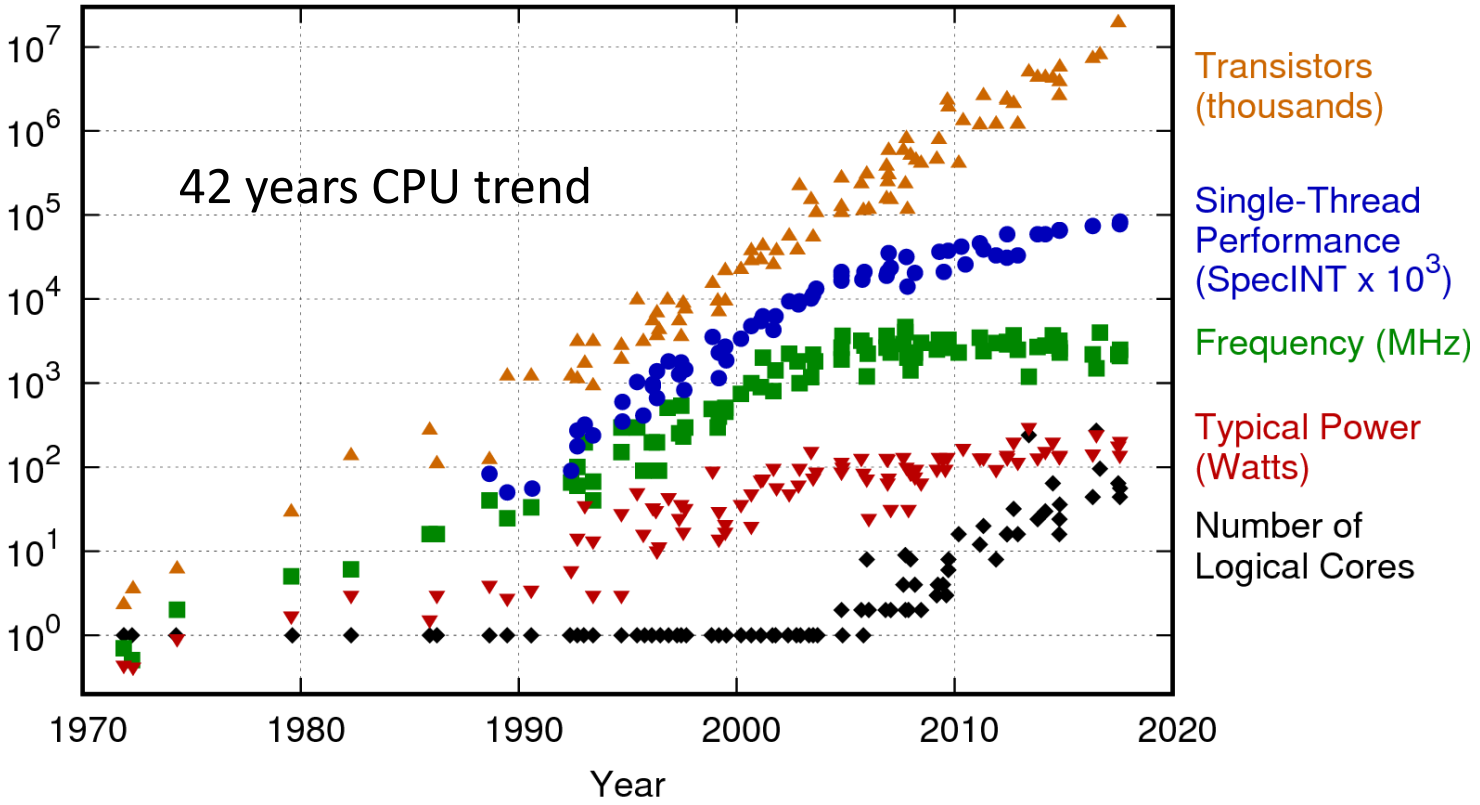
ROBERT H. DENNARD, MEMBER, IEEE, FRITZ H. GAENSSLEN, HWA-NIEN YU, MEMBER, IEEE, V. LEO RIDEOUT, MEMBER, IEEE, ERNEST BASSOUS, AND ANDRE R. LEBLANC, MEMBER, IEEE

Dennard scaling made them useful!

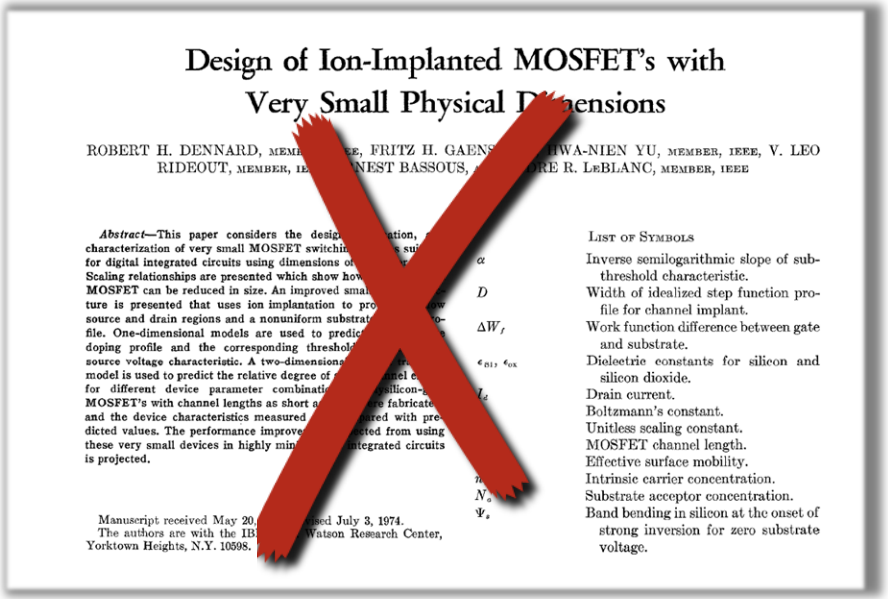
Moore's law gives us more transistors...

Introduction – Energy Scaling

- Mid 2000s Dennard's Scaling broke down



Source: https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/



Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions

ROBERT H. DENNARD, FRITZ H. GAENSSEN, HWA-NIEN YU, V. LEO RIDEOUT, ERNEST BASSOUS, ANDRE R. LEBLANC

Abstract—This paper considers the design, characterization of very small MOSFET switching for digital integrated circuits using dimensions Scaling relationships are presented which show how MOSFET can be reduced in size. An improved structure is presented that uses ion implantation to provide source and drain regions and a nonuniform substrate file. One-dimensional models are used to predict doping profile and the corresponding threshold source voltage characteristic. A two-dimensional model is used to predict the relative degree of for different device parameter combinations MOSFET's with channel lengths as short as and the device characteristics measured dicted values. The performance improvement these very small devices in highly integrated circuits is projected.

List of Symbols

Inverse semilogarithmic slope of subthreshold characteristic.
Width of idealized step function profile for channel implant.
Work function difference between gate and substrate.
Dielectric constants for silicon and silicon dioxide.
Drain current.
Boltzmann's constant.
Unitless scaling constant.
MOSFET channel length.
Effective surface mobility.
Intrinsic carrier concentration.
Substrate acceptor concentration.
Band bending in silicon at the onset of strong inversion for zero substrate voltage.

Manuscript received May 20, ... July 3, 1974.
The authors are with the IB... Watson Research Center, Yorktown Heights, N.Y. 10598.

Dennard scaling made them useful!

The rise of custom computing -- High-Performance Accelerators

- 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..

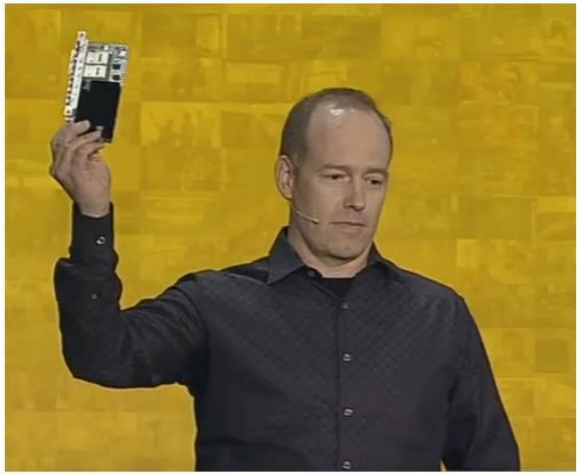
Microsoft Goes All in for FPGAs to Build Out AI Cloud

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."

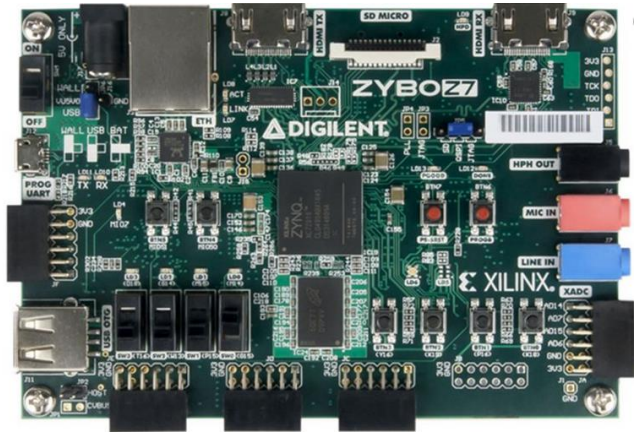
The F1 instance is equipped with Xilinx's Virtex Ultrascale+ VU9P, a 16nm device that contains about 2.5 million logic elements and over 6,800 DSP engines. The F1's host CPU is Intel's 18-core Broadwell E5 2686 v4 processor, which runs at 2.3 GHz. The instance may be configured with up to 8 FPGAs, 967 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.

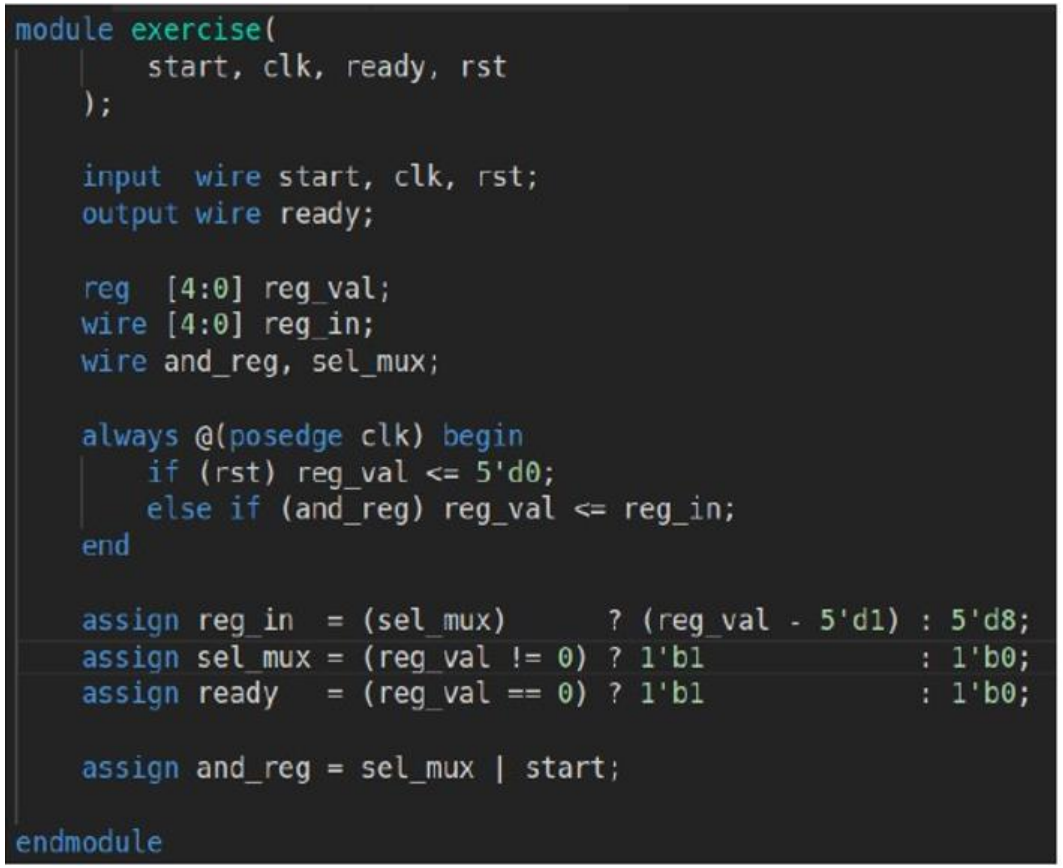
At this point, FPGA application development is geared toward a rather niche audience, so this will hardly be a volume business for Amazon in the near-term. HPC cloud specialist Nimbix recently added Xilinx FPGAs to its own infrastructure in the hopes of building a customer base of FPGA computing enthusiasts. 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
```

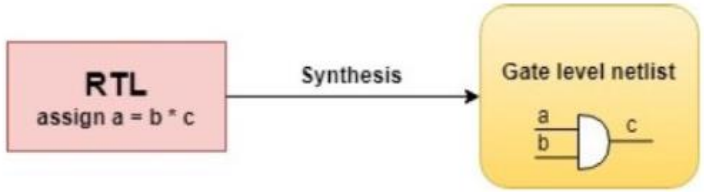
Hard and time consuming!

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



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,...)

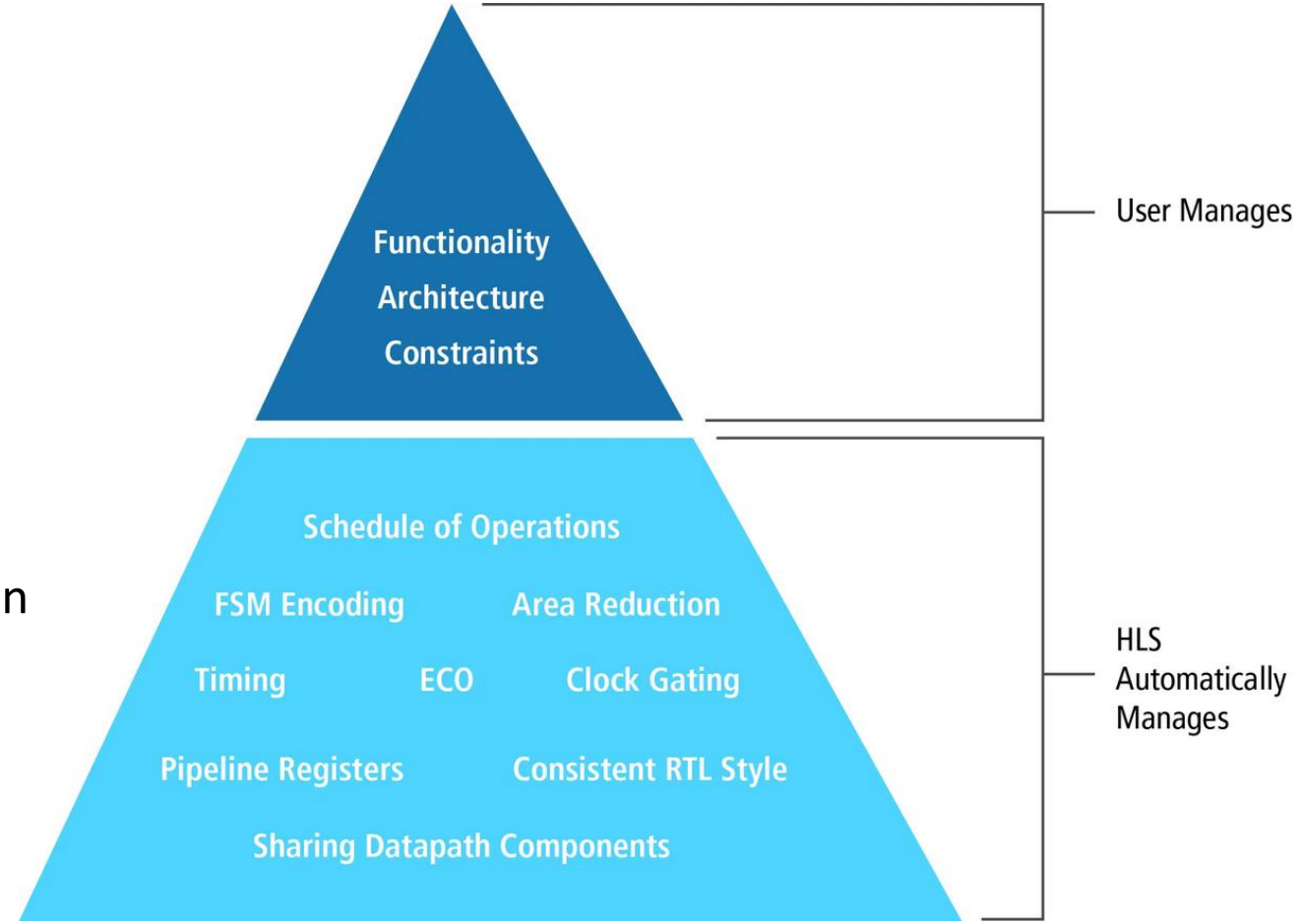
HLS directives:

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

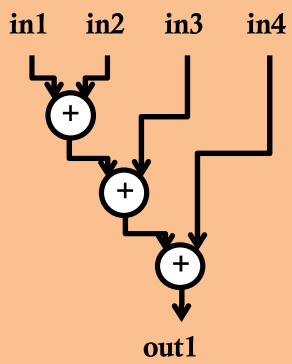
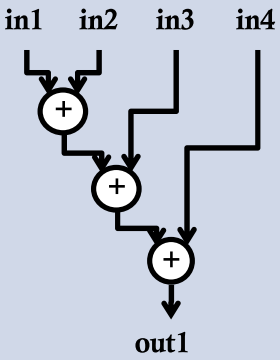
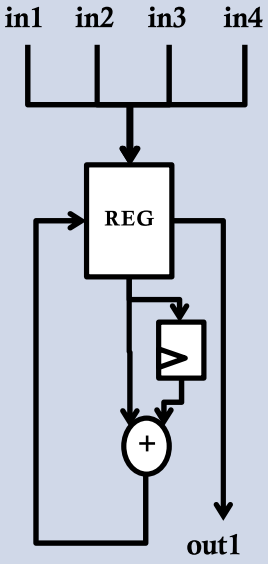
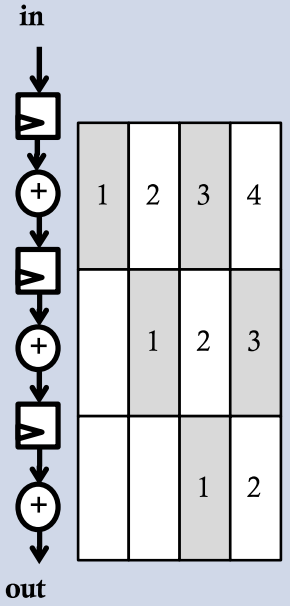
High-level synthesis (HLS)

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)



Design space exploration: Automatic evaluation of HW micro-architectures

	Latency	Area	Throughput
 **Control-Data Flow Graph**			
$out1 = f(in1, in2, in3, in4)$	$t_{clk} \quad 3 \quad d_{add}$ $T_1 = 1 / t_{clk}$ $A_1 = 3 * A_{add}$	$t_{clk} \approx d_{add} + d_{setup}$ $T_2 = 1 / (3 * t_{clk})$ $A_2 = A_{add} + 2 * A_{reg}$	$t_{clk} \quad d_{add} + d_{setup}$ $T_3 = 1 / t_{clk}$ $A_3 = 3 * A_{add} + 6 * A_{reg}$
Untimed	**Combinational**	**Sequential**	**Pipelined**

HLS from the practical point view: Mapping C constructs to HW

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

HLS Functions and RTL hierarchy

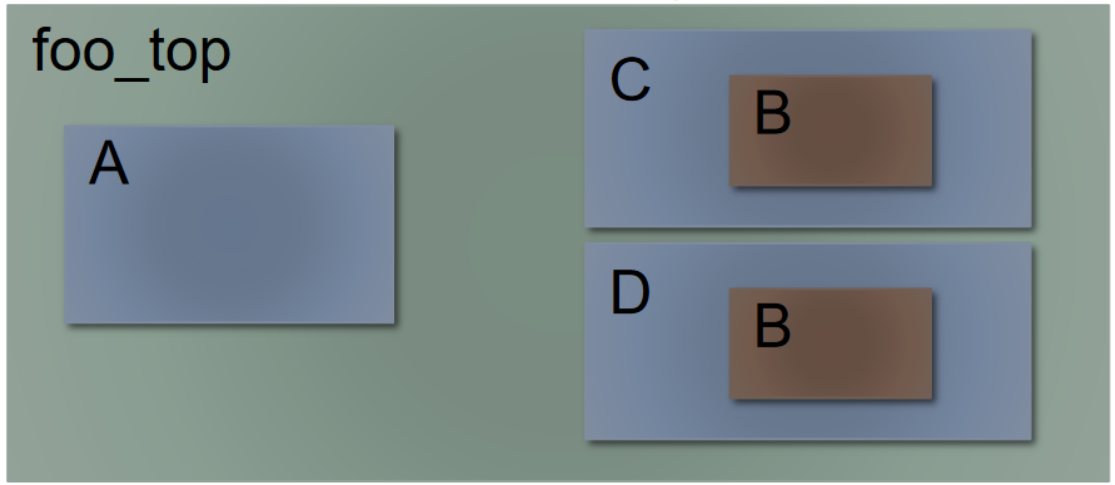
- Each function is translated into an RTL block
  - Verilog module, VHDL entity

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

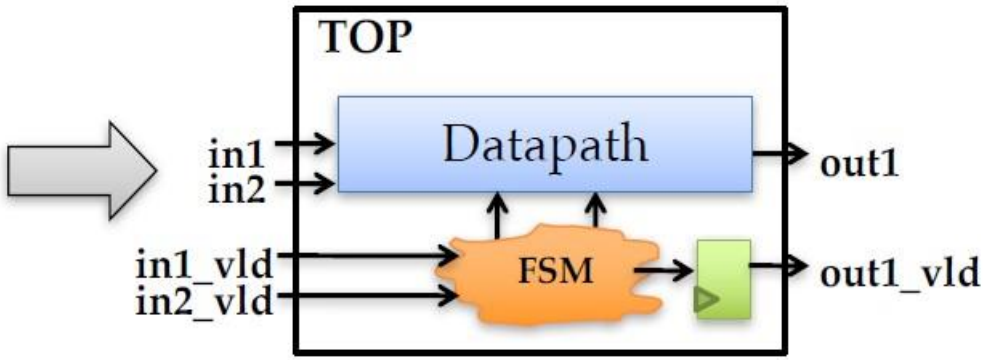
  - By default, each function is implemented using a common instance
  - Functions may be inlined to dissolve their hierarchy
    - Small functions may be automatically inlined

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;
}
```



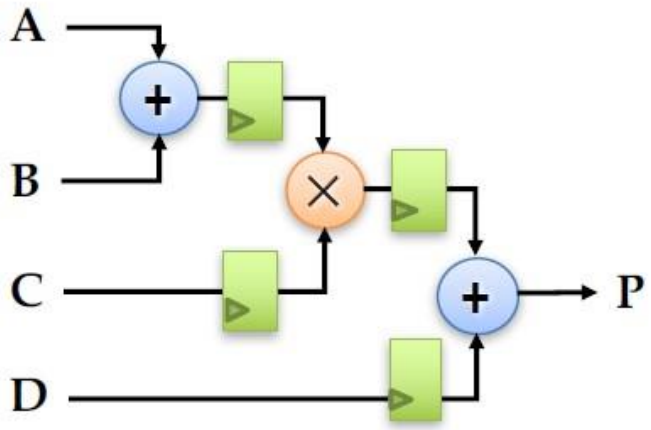
Input/output (I/O) protocols (axi_dma, etc.)

- 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
```



Algorithmica (1991) 6: 5–35

Algorithmica
© 1991 Springer-Verlag New York Inc.

Retiming Synchronous Circuitry[1]

Charles E. Leiserson[2] and James B. Saxe[3]

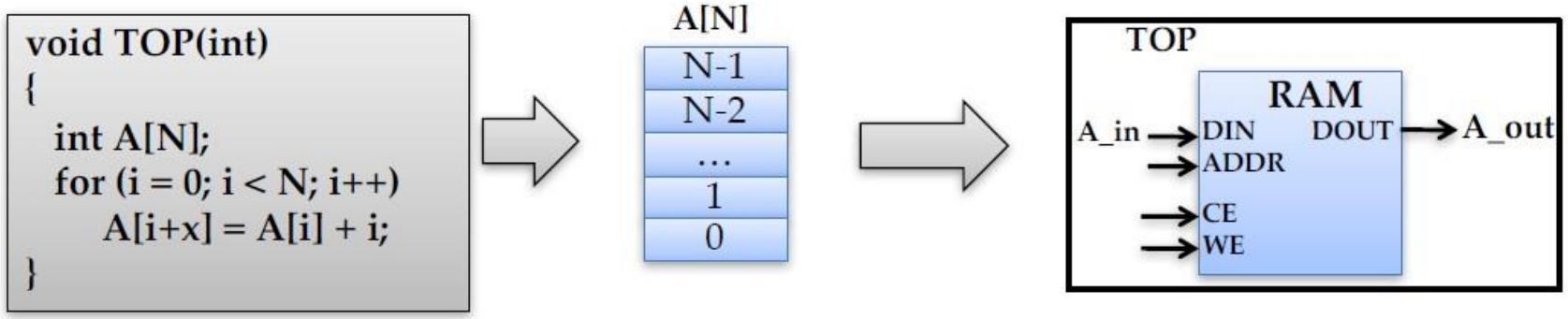
Abstract. This paper describes a circuit transformation called *retiming* in which registers are added at some points in a circuit and removed from others in such a way that the functional behavior of the circuit as a whole is preserved. We show that retiming can be used to transform a given synchronous circuit into a more efficient circuit under a variety of different cost criteria. We model a circuit as a graph in which the vertex set V is a collection of combinational logic elements and the edge set E is the set of interconnections, each of which may pass through zero or more registers. We give an $O(|V||E|\lg|V|)$ algorithm for determining an equivalent retimed circuit with the smallest possible clock period. We show that the problem of determining an equivalent retimed circuit with minimum state (total number of registers) is polynomial-time solvable. This result yields a polynomial-time optimal solution to the problem of pipelining combinational circuitry with minimum register cost. We also give a chacterization of optimal retiming based on an efficiently solvable mixed-integer linear-programming problem.

Key Words. Digital circuitry, Graph theory, Linear programming, Network flow, Optimization, Pipelining, Propagation delay, Retiming, Synchronous circuitry, Systolic circuits, Timing analysis.

1. Introduction. The goal of VLSI design automation is to speed the design of a system without sacrificing the quality of implementation. A common means of achieving this goal is through the use of optimization tools that improve the quality of a quickly designed circuit. In this paper we show how to optimize clocked circuits by relocating registers so as to reduce combinational rippling. Unlike pipelining, this technique, which we call *retiming*, does not increase circuit latency. In order to illustrate retiming, consider the problem of designing a digital

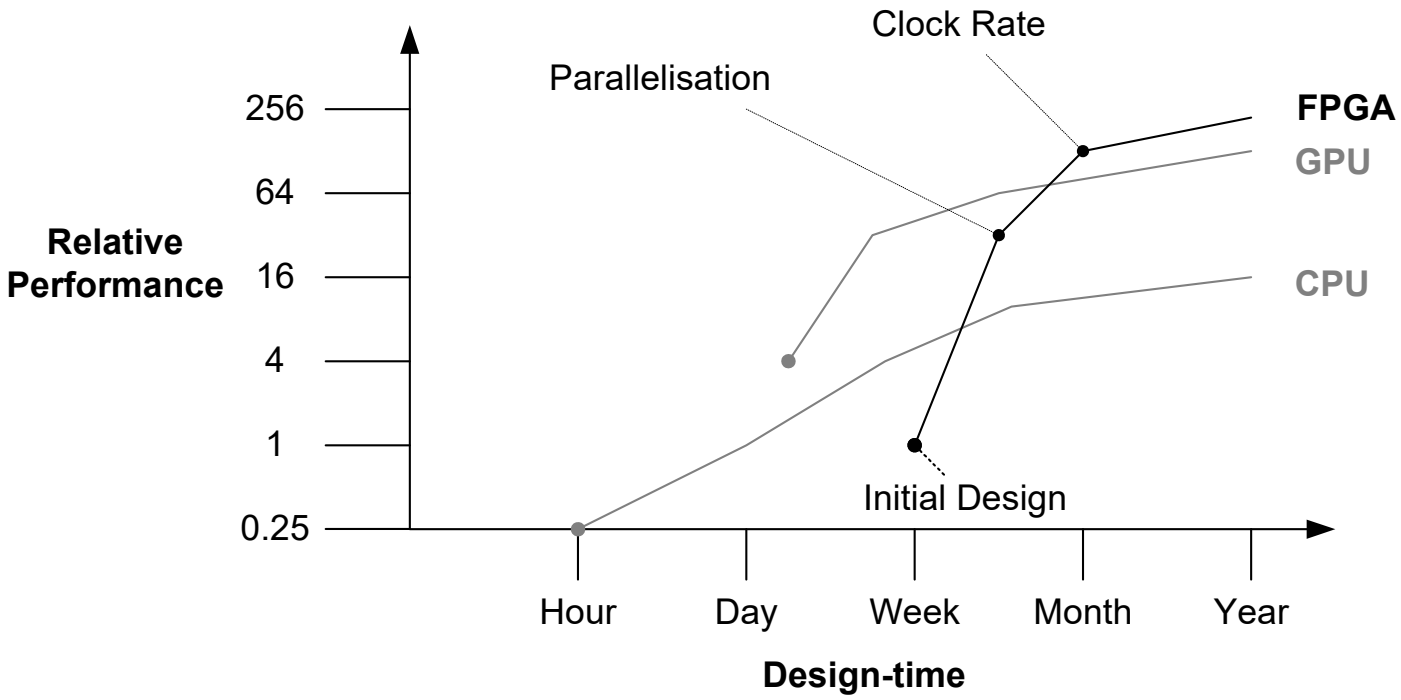
HLS Memory extraction

- 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

Getting performance out of HLS … not a free lunch



Performance through specialization

- **FPGAs provide large speed-up and power savings – *at a price!***
  - Days or weeks to get an initial version working
  - Multiple optimisation and verification cycles to get high performance

Table 1. Vivado HLS Pragmas by Type

Type	Attributes
Kernel Optimization	• pragma HLS allocation • pragma HLS clock • pragma HLS expression_balance • pragma HLS latency • pragma HLS reset • pragma HLS resource • pragma HLS top
Function Inlining	• pragma HLS inline • pragma HLS function_instantiate
Interface Synthesis	• pragma HLS interface • pragma HLS protocol
Task-level Pipeline	• pragma HLS dataflow • pragma HLS stream
Pipeline	• pragma HLS pipeline • pragma HLS occurrence
Loop Unrolling	• pragma HLS unroll • pragma HLS dependence
Loop Optimization	• pragma HLS loop_flatten • pragma HLS loop_merge • pragma HLS loop_tripcount
Array Optimization	• pragma HLS array_map • pragma HLS array_partition • pragma HLS array_reshape
Structure Packing	• pragma HLS data_pack

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.

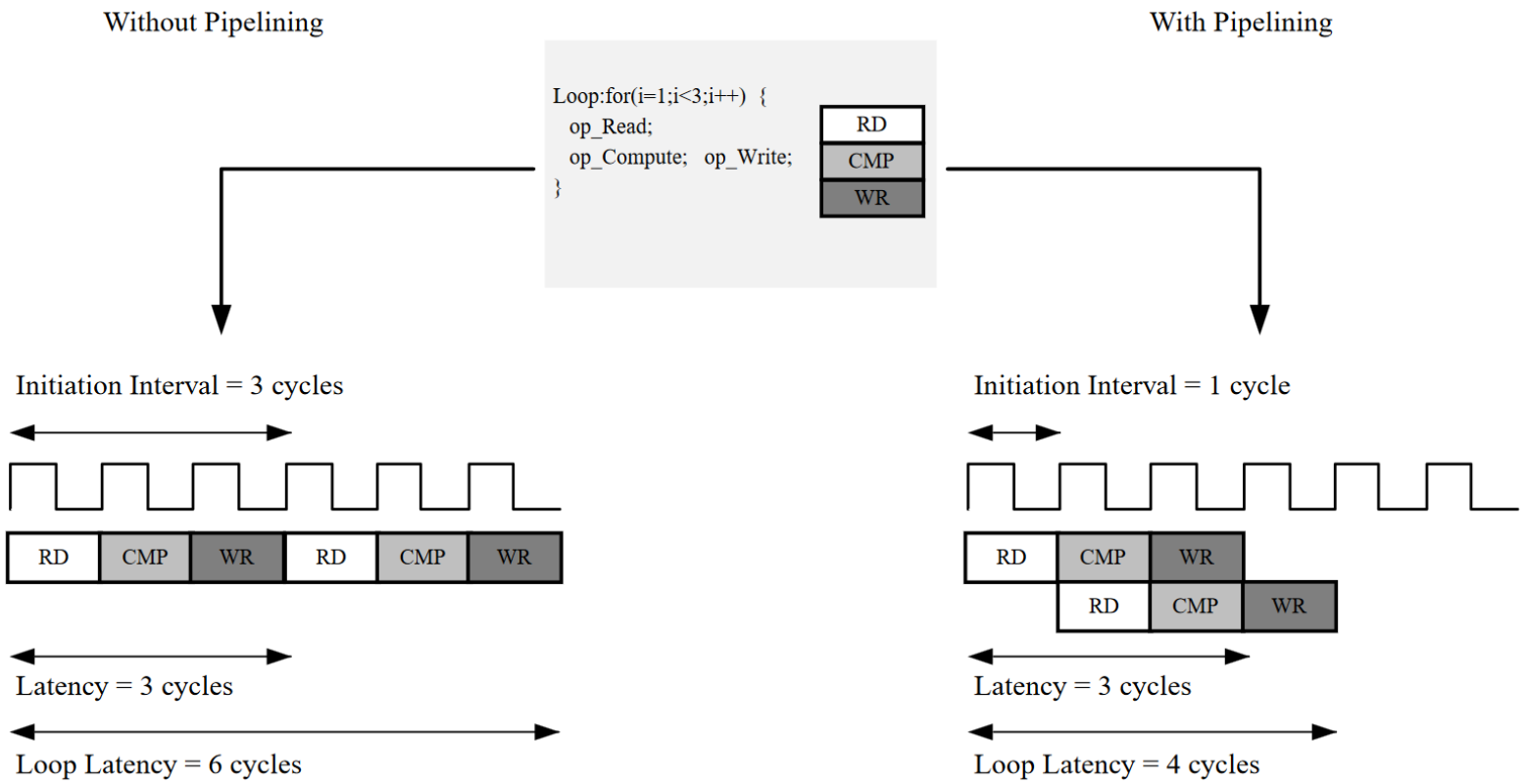
Key metric: **Initiation Interval (II)** expressed in number of cycles. It has many different meanings:

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.

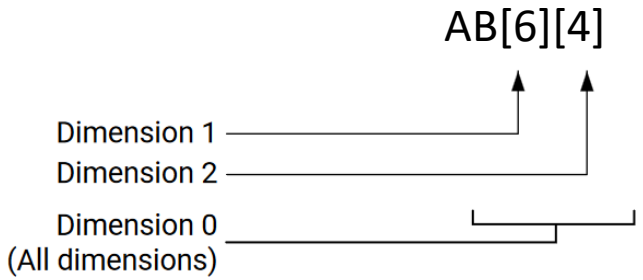
HLS provides three types of array partitioning:

- **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]



Direct HLS memory implementation (1/3)

C/C++ Kernel

```
int buffer[16] ;
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

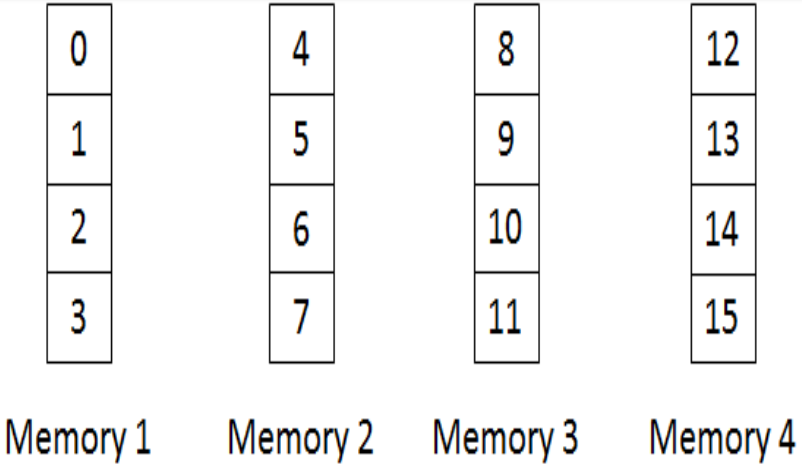
buffer

- Buffer implemented in 1 physical memory
- Buffer can sustain 2 concurrent transactions
- Reading all values of buffer can take 8 clock cycles

Direct HLS memory implementation (2/3)

C/C++ Kernel

```
int buffer[16];
#pragma HLS ARRAY_PARTITION variable=buffer block factor=4 dim=1
```

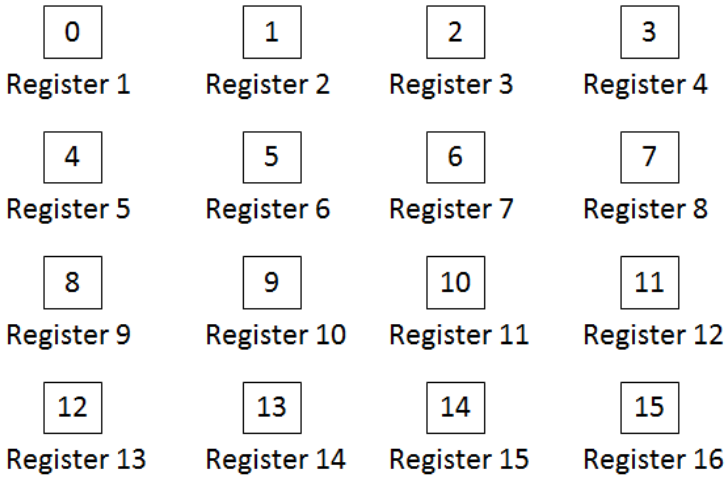


- Buffer implemented in 4 physical memories
- Buffer can sustain 8 concurrent transactions
- Compiler handles all address translations to physical memories
- Reading all values of buffer can take 2 clock cycles

Direct HLS memory implementation (3/3)

C/C++ Kernel

```
int buffer[16];
#pragma HLS ARRAY_PARTITION variable=buffer complete dim=1
```



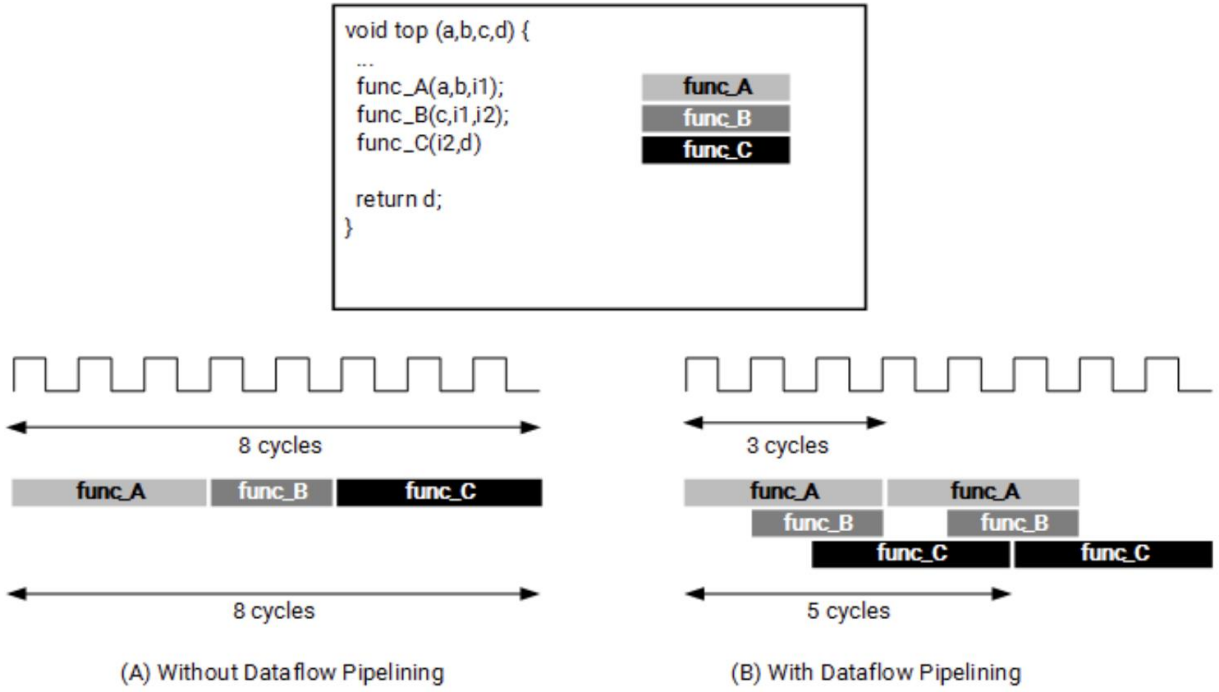
- Buffer implemented in 16 registers
- Buffer can sustain 16 concurrent transactions
- Compiler handles all address translations to physical memories
- Reading all values of buffer can take 1 clock cycles

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;
  }
}
```

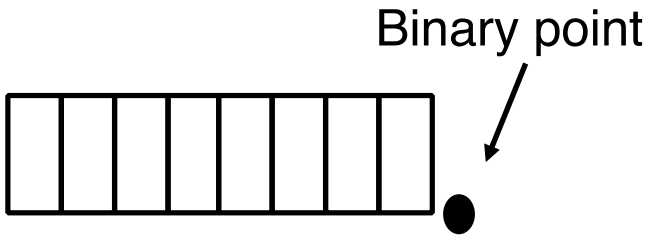
Binary Number Representation

Unsigned number

- MSB has weight 2^{n-1}
- Range of an n-bit unsigned number: ?

Two's complement

- MSB has weight -2^{n-1}
- Range of an n-bit two's complement number: ?



Examples: assuming integers here

2^3	2^2	2^1	2^0	unsigned
1	0	1	1	= 11

-2^3	2^2	2^1	2^0	2'c
1	0	1	1	= -5

Arbitrary precision integer

- C/C++ only provides a limited set of native integer types
  - char (8b), short (16b), int (32b), long (?), long long (64b)
  - Byte aligned: efficient in processors

- Arbitrary precision integer in Vivado HLS
  - Signed: **ap_int**; Unsigned **ap_uint**
  - Templatized class ap_int<W> or ap_uint<W>
    - W is the user-specified bitwidth
  - Two's complement representation for signed integer

```
#include "ap_int.h"
…
ap_int<9>       x;  // 9-bit
ap_uint<24>     y;  // 24-bit unsigned
ap_uint<512>    z;  // 512-bit unsigned
```

Representing fractional numbers

- Binary representation can also represent fractional numbers, usually **called fixed-point numbers**, by simply extending the pattern to include negative exponents
  - Less convenient to use compared to floating-point types
  - Efficient and cheap in application-specific hardware

2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	unsigned
1	0	1	1 .	0	1	= 11.25

Binary point

Overflow and underflow

- **Overflow** occurs when a number is larger than the largest number that can be represented in a given number of bits

2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	unsigned
0	0	1	0	1	1	0	1	= 11.25
	0	1	0	1	1	0	1	= 11.25
		1	0	1	1	0	1	= 11.25
			0	1	1	0	1	= 3.25

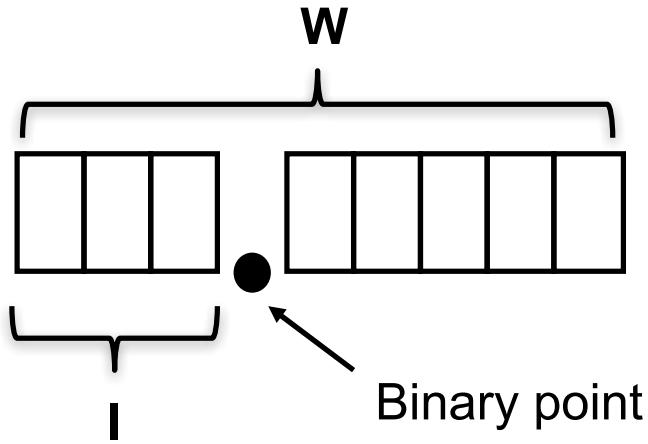
Drop MSB

Overflow occurs

- **Underflow** occurs when a number is smaller than the smallest number that can be represented

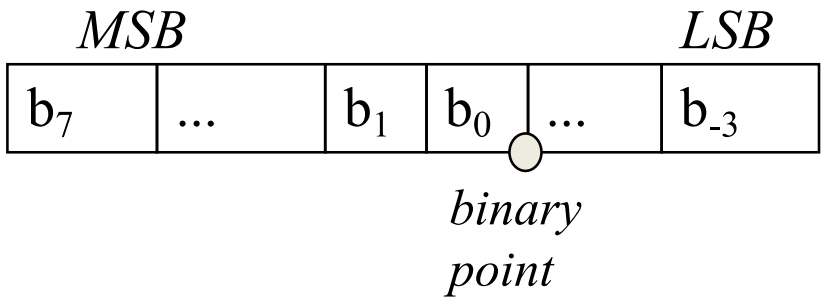
Fixed-point type in Vivado

- Arbitrary precision fixed-point type
  - Signed: **ap_fixed**; Unsigned **ap_ufixed**
  - Templatized class ap_fixed<W, I, Q, O>
    - W: total word length
    - I: integer word length
    - Q: quantization mode
    - O: overflow mode



Example: Fixed-point modelling

- ap_ufixed<11, 8, AP_TRN, AP_WRAP> x;

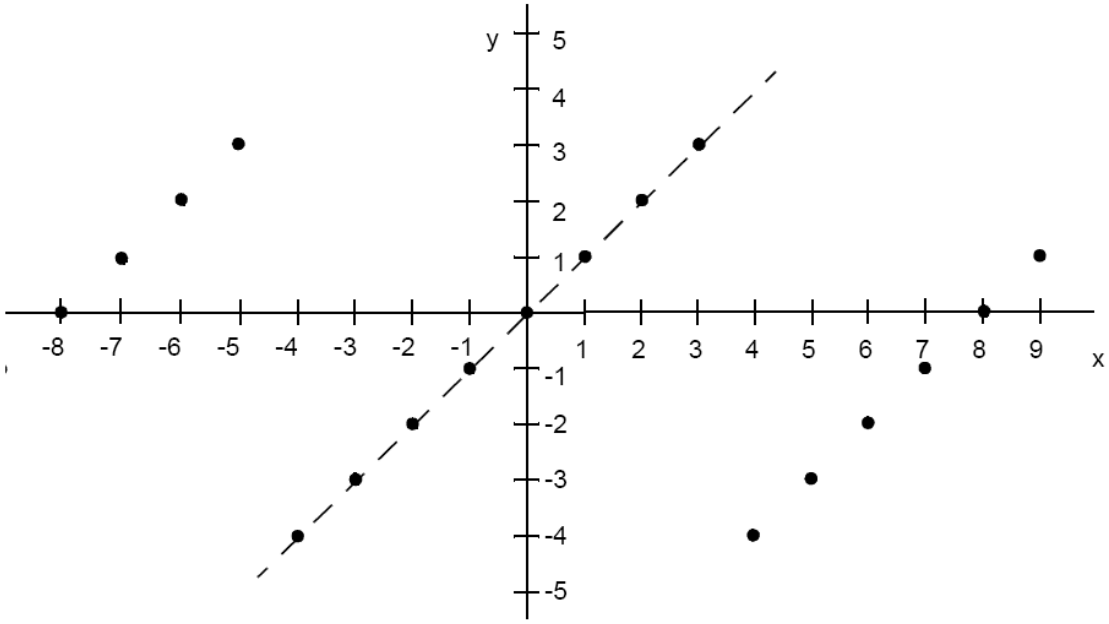


  - 11 is the total number of bits in the type
  - 8 bits to the left of the decimal point
  - AP_TRN defines truncation behavior for quantization
  - AP_WRAP defines wrapping behavior for overflow

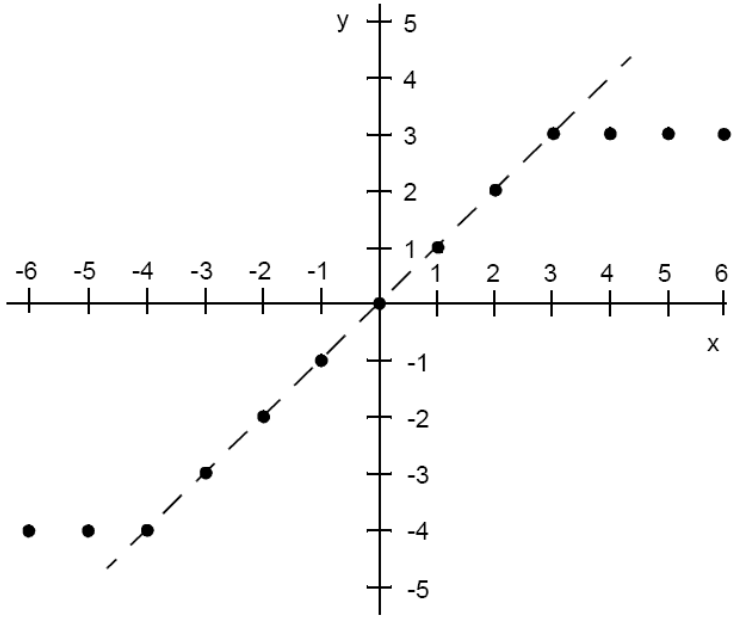
Fixed-point type: Overflow behavior

- ap_fixed overflow mode
  - Determines the behavior of the fixed point type when the result of an operation generates more precision in the **MSBs** than is available

```
ap_fixed<W, IW_X> x;
ap_fixed<W, IW_Y> y = x; /* IW_Y < IW_X */
```



Default: AP_WRAP (wrapping mode)



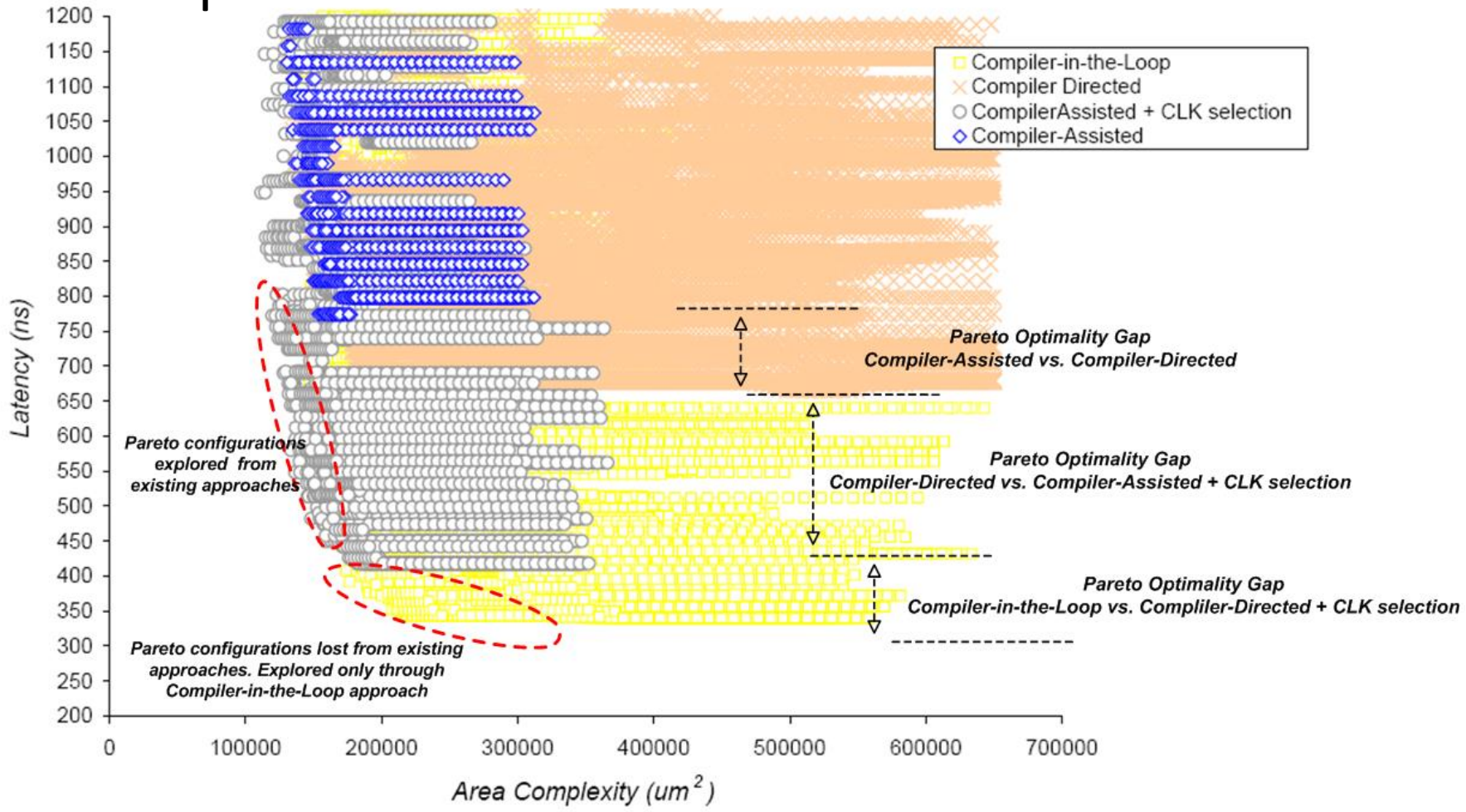
AP_SAT (saturation mode)

Fixed-point type: Quantization behavior

- ap_fixed quantization mode
  - Determines the behavior of the fixed point type when the result of an operation generates more precision in the **LSBs** than is available
  - Default mode: AP_TRN (truncation)
  - Other rounding modes: AP_RND, AP_RND_ZERO, AP_RND_INF, …

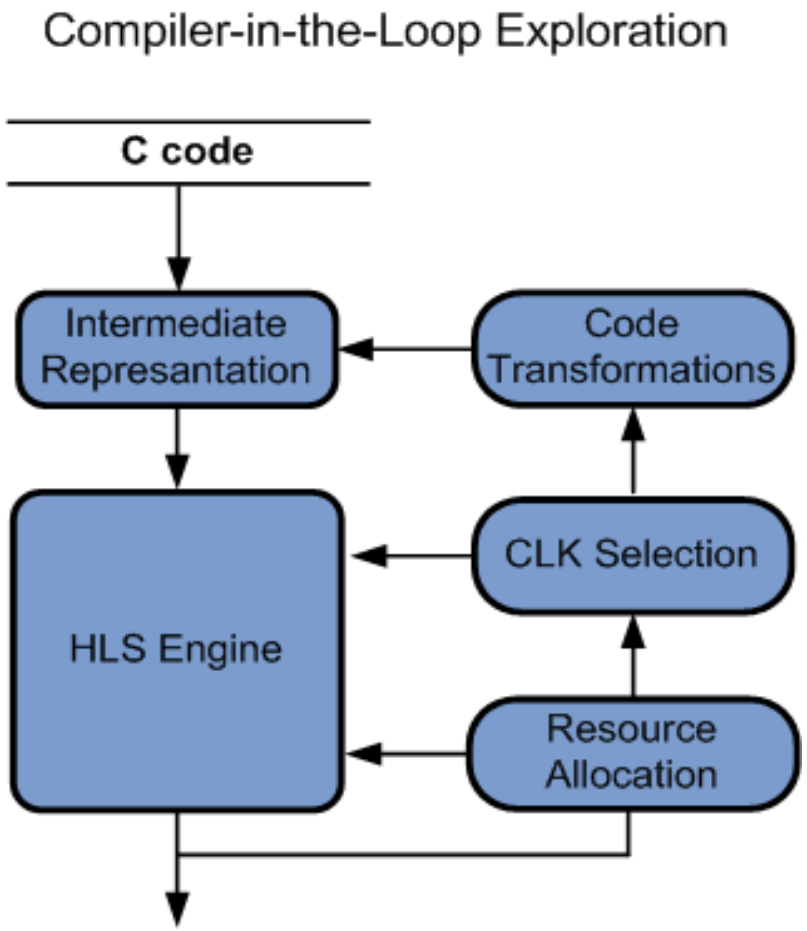
```
ap_fixed<4, 2, AP_TRN>   x = 1.25;     (b'01.01)
ap_fixed<3, 2, AP_TRN>   y = x;
                           └──→ 1.0     (b'01.0)

ap_fixed<4, 2, AP_TRN>   x = -1.25;    (b'10.11)
ap_fixed<3, 2, AP_TRN>   y = x;
                           └──→ -1.5    (b'10.1)
```

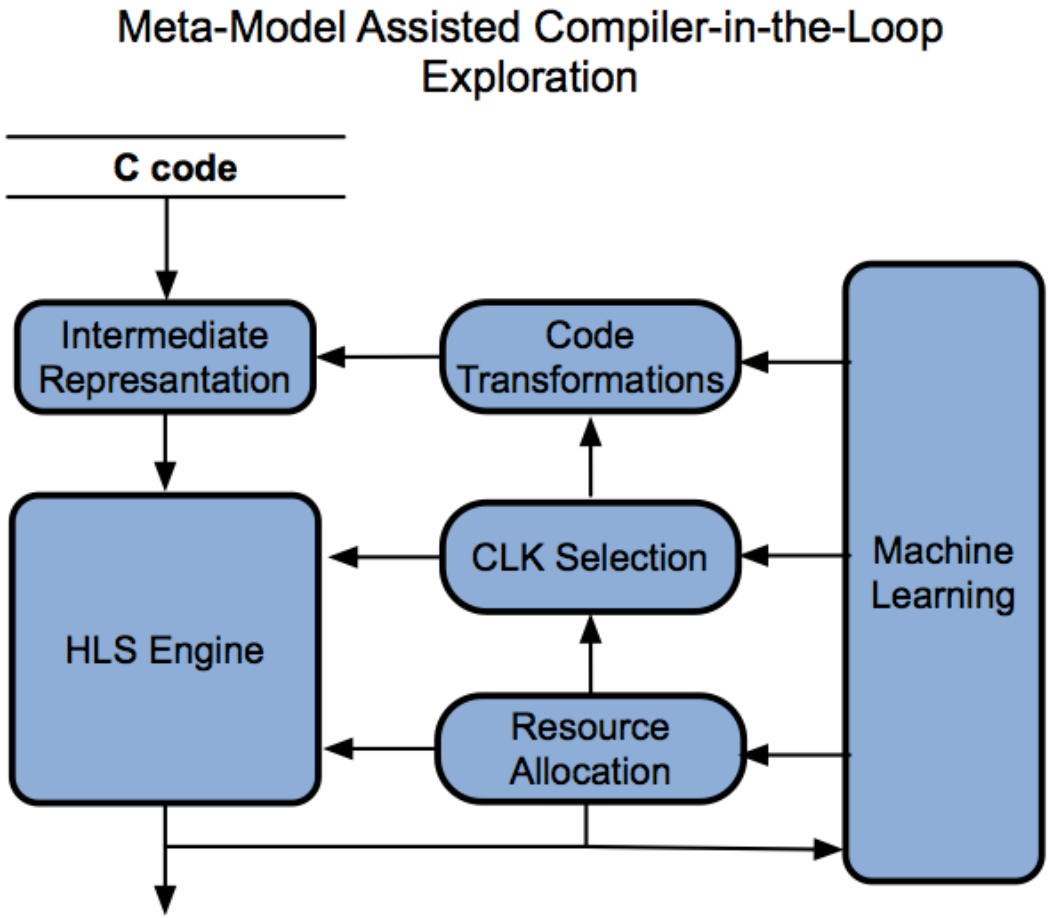
COMPINLOOP expansion OF the “visible” solution space



[**Source**: **Sotirios Xydis**, Kiamal Pekmestzi, Dimitrios Soudris, George Economakos, “Compiler-in-the-Loop Exploration During Datapath Synthesis for Higher Quality Delay-Area Trade-offs”, **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, 18, 1, article 11, 11:1-11:35, January 2013]

EXPANSION: Compiler-In-the-Loop Explorative HLS





Scaled up problem sizes:
50K < Solution spaces < 1M