https://inst.eecs.berkeley.edu/~eecs251b/sp23/

EECS251B
Advanced Digital Circuits and Systems

Lecture 3 – Design Productivity

Vladimir Stojanović

Tuesdays and Thursdays 9:30-11am

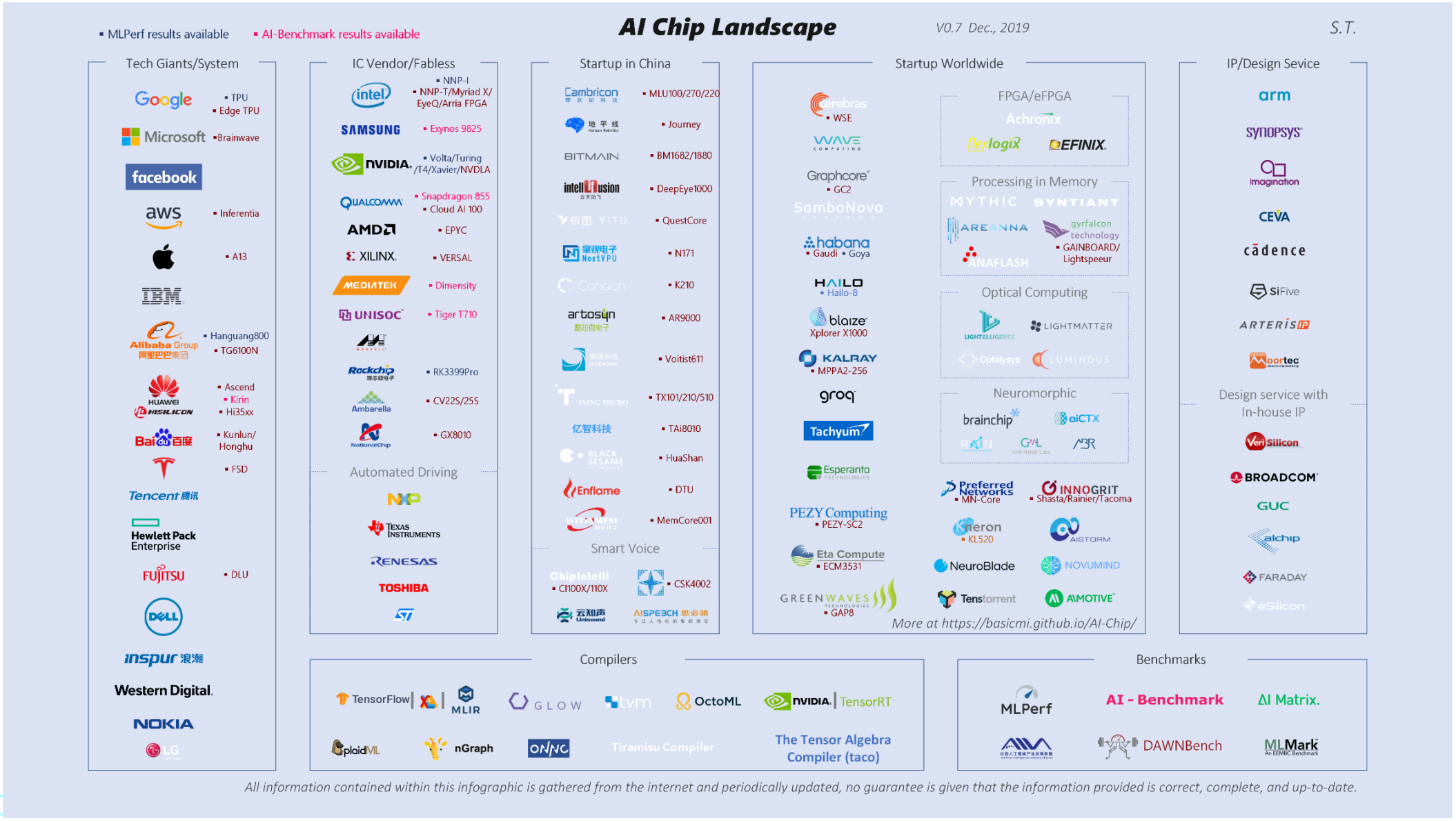
Cory 521

- **Design Productivity**
- High-Level Synthesis
- Chisel
- SystemVerilog (next lecture)

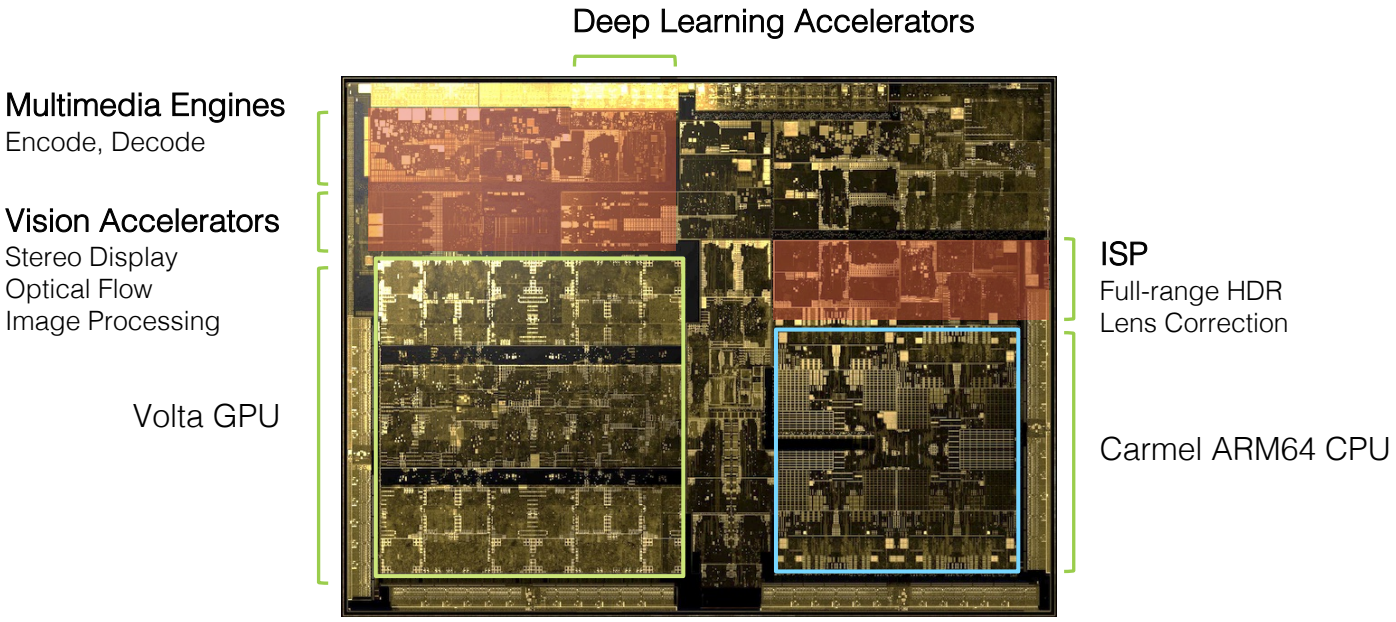
Resources

- Chisel
  - Chisel bootcamp: https://github.com/freechipsproject/chisel-bootcamp
  - Professor Scott Beamer's lectures on Agile Hardware Design in Jupyter notebooks (https://github.com/agile-hw/lectures)
  - Resources: https://www.chisel-lang.org/chisel3/docs/resources/resources.html
  - Scala course: https://www.coursera.org/specializations/scala
- High-Level Synthesis
  - Professor Zhiru Zhang's course on High-Level Synthesis (https://www.csl.cornell.edu/courses/ece5775/schedule.html)
  - High-Level Synthesis Blue Book
  - Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer, Parallel Programming for FPGAs, arXiv, 2018.

Golden Age of Computer Architecture



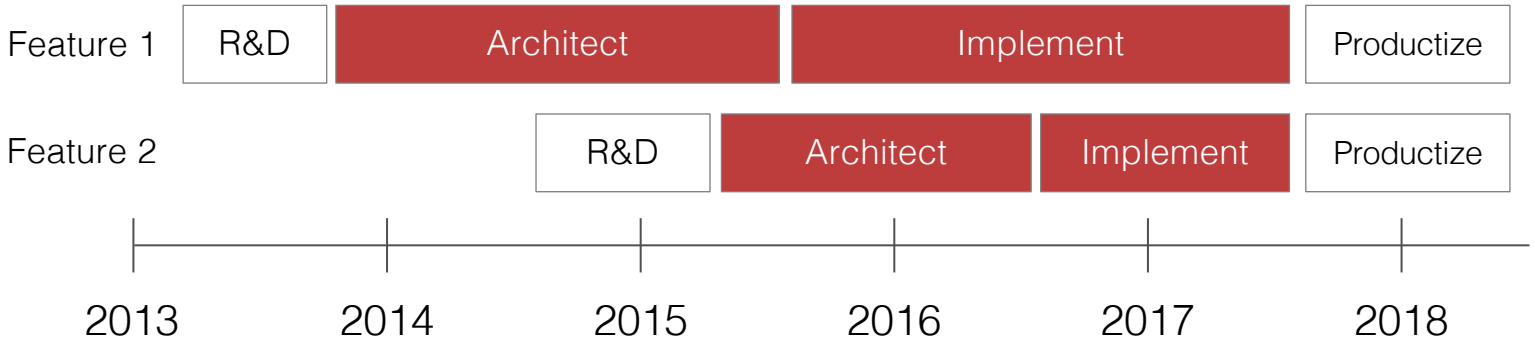
Today's Systems-on-Chip are complex



"Most Complex SoC Ever Made
~8,000 Engineering Years"

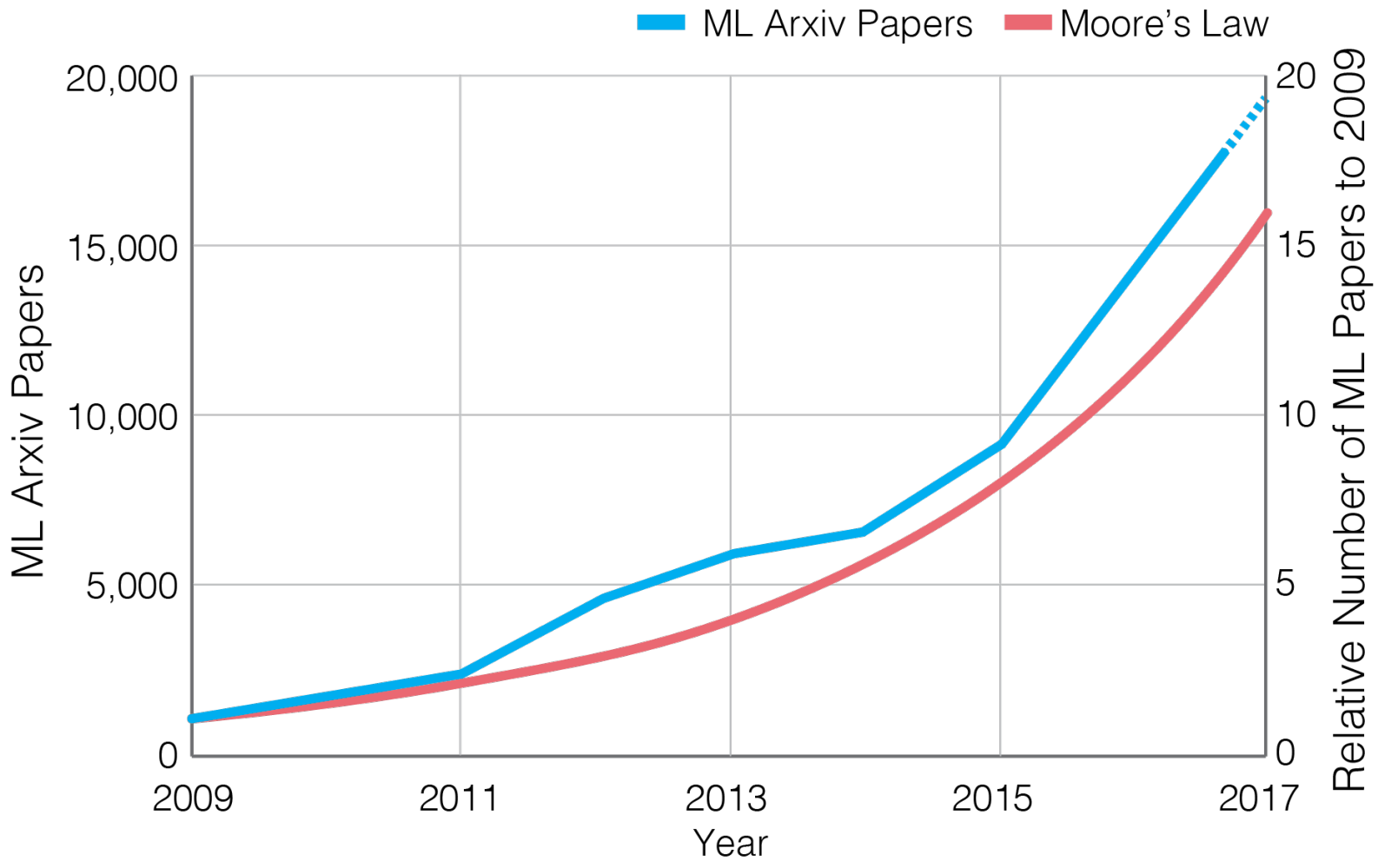
* NVIDIA Xavier Announcement, CES'2018

Hardware design efforts are relatively slow



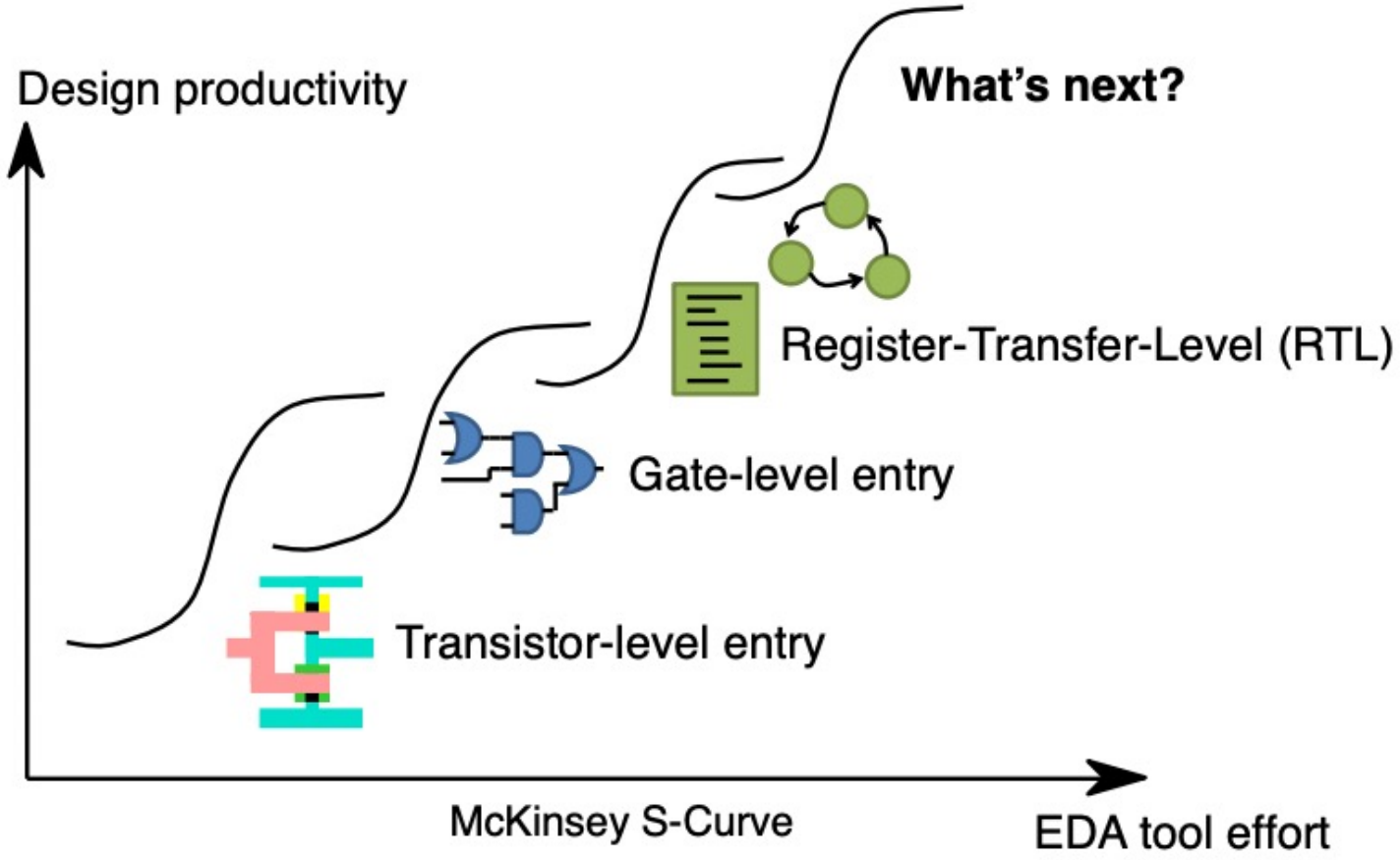
- The number of unique features in hardware is limited by design effort

While algorithms are changing relatively quickly



- Design Productivity
- **High-Level Synthesis**
- Chisel
- SystemVerilog (next lecture)

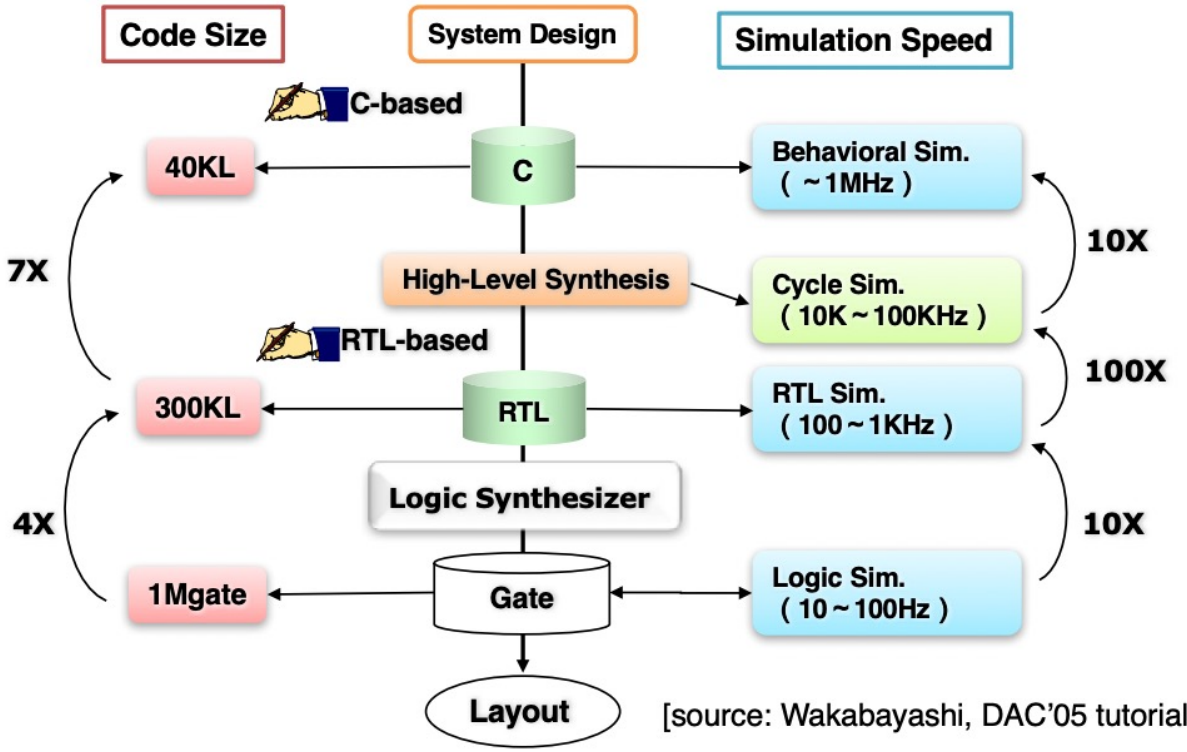
Design Abstraction Evolution



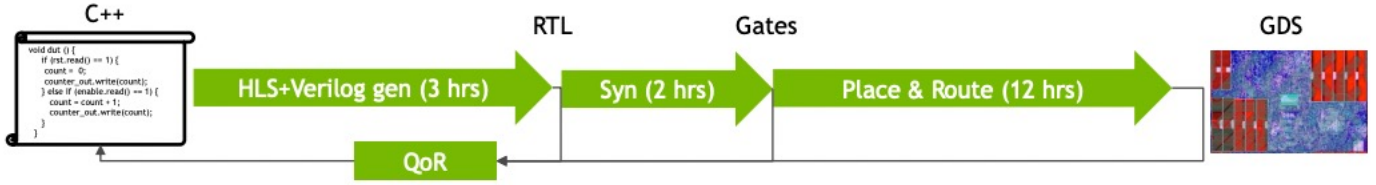
Source: Kurt Keutzer

High-Level Synthesis (HLS)

- **Automated** design process that transforms a high-level functional specification to optimized register-transfer level (RTL) descriptions for efficient hardware implementation
- Why HLS?
  - Productivity
    - Lower design complexity and faster simulation speed
  - Portability
    - Single source -> multiple implementations
  - Permutability
    - Rapid design space exploration -> higher quality of result (QoR)



[source: Wakabayashi, DAC'05 tutorial]



[Source: NVIDIA, 2019]

High-Level Synthesis (HLS)

- **Automated** design process that transforms a high-level functional specification to optimized register-transfer level (RTL) descriptions for efficient hardware implementation
- Why HLS?
  - Productivity
    - Lower design complexity and faster simulation speed
  - Portability
    - Single source -> multiple implementations
  - Permutability
    - Rapid design space exploration -> higher quality of result (QoR)

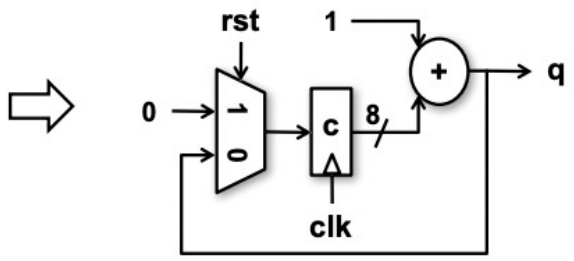
```
module dut(rst, clk, q);
  input rst;
  input clk;
  output q;
  reg [7:0] c;

  always @ (posedge clk)
  begin
    if (rst == 1b'1) begin
      c <= 8'b00000000;
  end
  else begin
    c <= c + 1;
  end

  assign q = c;
endmodule
```

RTL Verilog

vs.

```
uint8 dut() {
  static uint8 c;
  c+=1;
}
```

High-Level Synthesis



An 8-bit counter

Case study: FIR Filter

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

$x[n]$	input signal
$y[n]$	output signal
N	filter order
b_i	ith filter coefficient

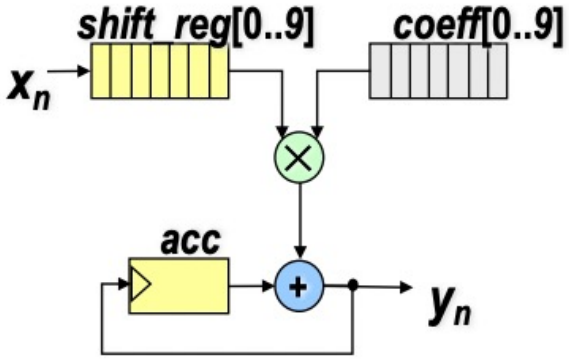
```
//original FIR
#define SIZE 128
#define N 10
void fir(int input[SIZE], int out[SIZE]) {
  // FIR coefficients
  int coeff[N] = {13,-2,9,…,74};
  // exact translation from the FIR equation
  for (int n = 0; n < SIZE; n++) {
    int acc = 0;
    for (int i = 0; i < N; i++) {
      if (n - i >=0)
        acc += coeff[i] * input[n-i];
    }
    output[n] = acc;
  }
}
```

[Example adapted from Prof. Zhiru Zhang's HLS tutorial.]

Case study: FIR Filter: Default Microarchitecture

```
void fir(int input[SIZE], int out[SIZE]) {
  // FIR coefficients
  int coeff[N] = {13,-2,9,…,74};
  // shift registers
  int shift_reg[N] = {0};
  for (int n = 0; n < SIZE; n++) {
    int acc = 0;
    for (int j = N - 1; j>0; j--) {
      shift_reg[j] = shift_reg[j-1];
    }
    shift_reg[0] = input[n];
    for (int i = 0; i < N; i++) {
      acc += coeff[i] * shift_reg[i];
    }
    output[n] = acc;
  }
}
```

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



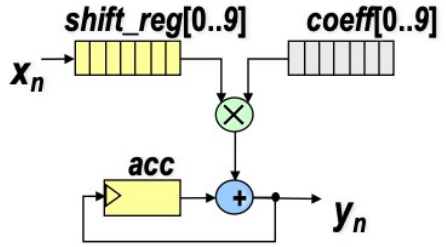
Possible optimizations

- Loop unrolling
- Array partitioning
- Pipelining

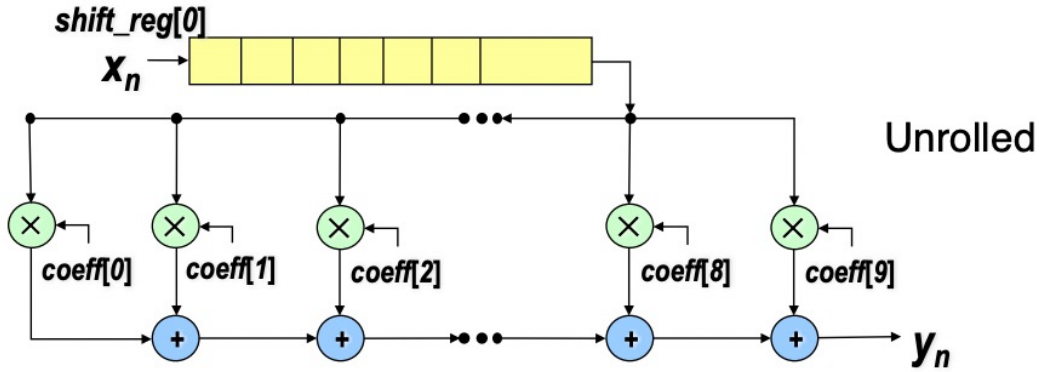
[Example adapted from Prof. Zhiru Zhang's HLS tutorial.]

Case study: FIR Filter: Unrolling

```
void fir(int input[SIZE], int out[SIZE]) {
  // FIR coefficients
  int coeff[N] = {13,-2,9,…,74};
  // shift registers
  int shift_reg[N] = {0};
  for (int n = 0; n < SIZE; n++) {
    int acc = 0;
    for (int j = N - 1; j>0; j--) {
      #pragma HLS unroll
      shift_reg[j] = shift_reg[j-1];
    }
    shift_reg[0] = input[n];
    for (int i = 0; i < N; i++) {
      #pragma HLS unroll
      acc += coeff[i] * shift_reg[i];
    }
    output[n] = acc;
  }
}
```



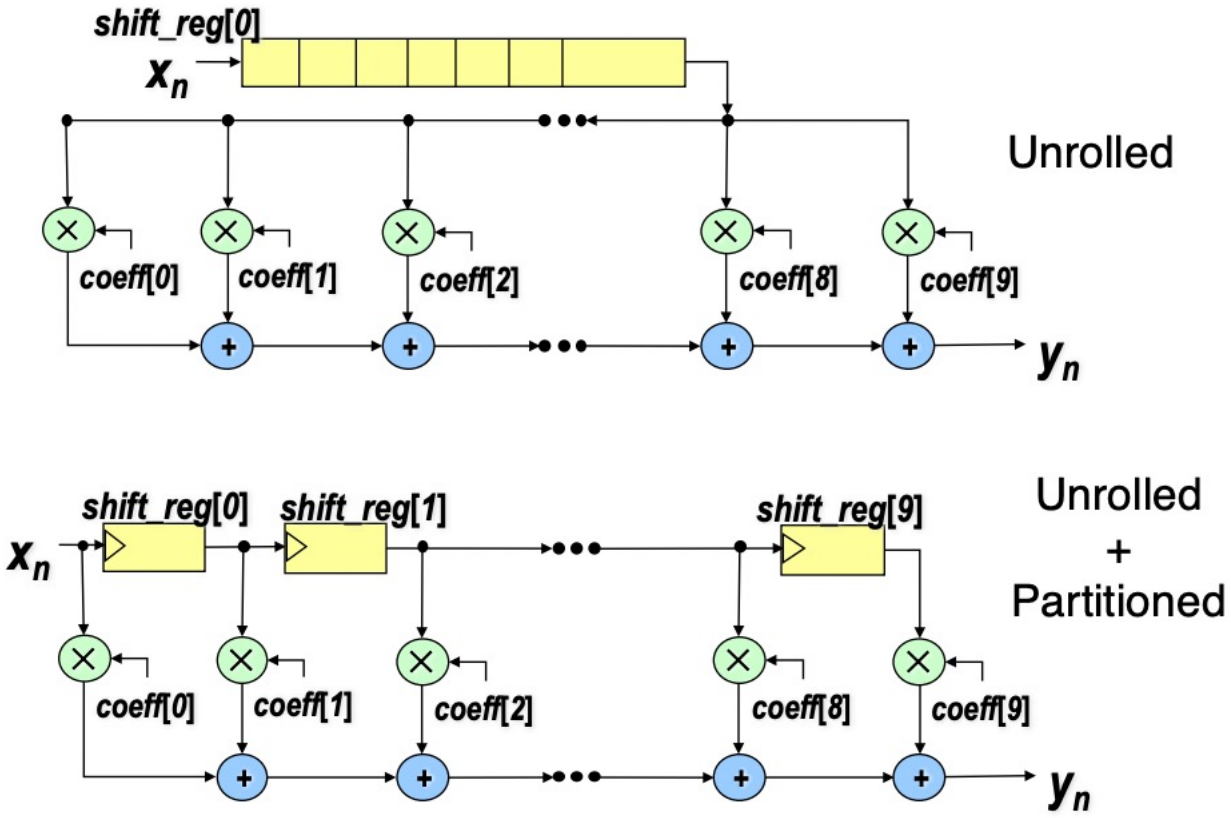
Default



Unrolled

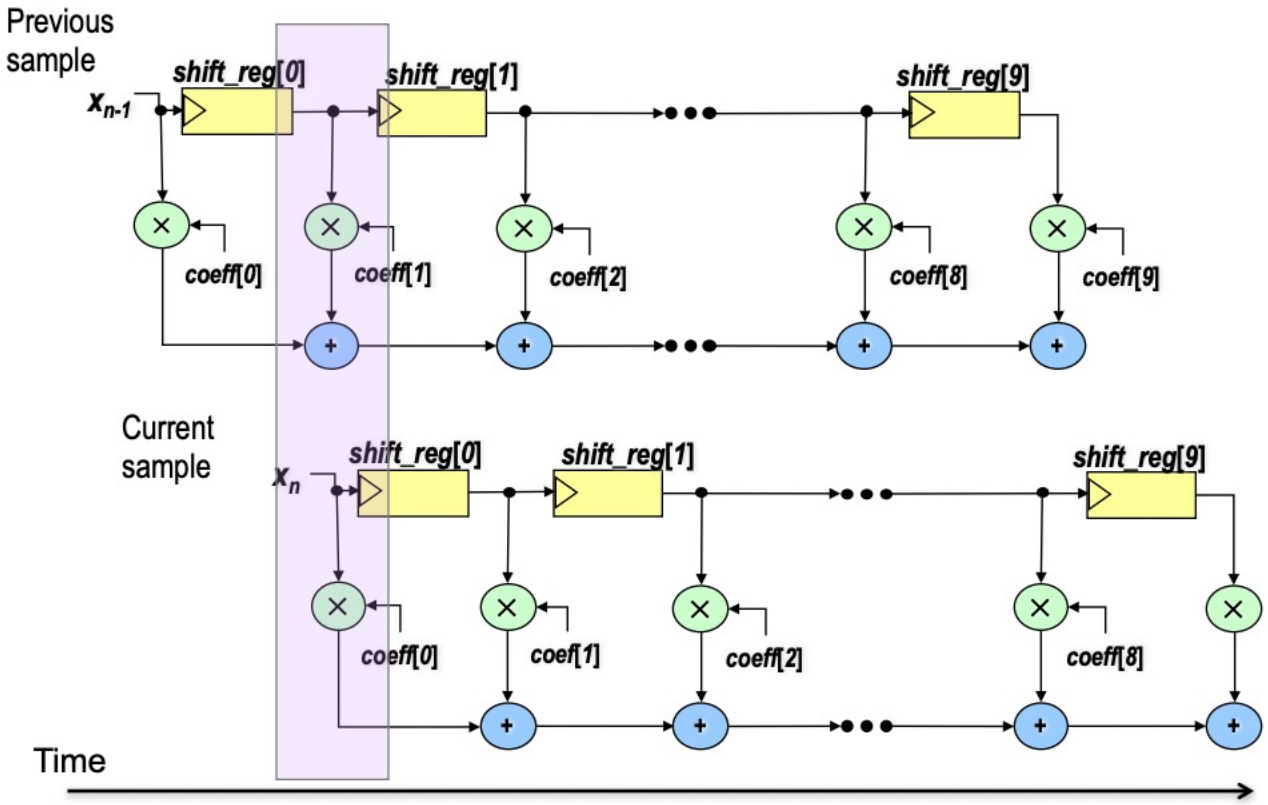
Case study: FIR Filter: Partitioning

```
void fir(int input[SIZE], int out[SIZE])
{
  // FIR coefficients
  int coeff[N] = {13,-2,9,…,74};
  // shift registers
  int shift_reg[N] = {0};
  #pragma HLS ARRAY_PARTITION
variable=shift_reg complete dim=0
  for (int n = 0; n < SIZE; n++) {
    for (int j = N - 1; j>0; j--) {
      …
    }
    …
    for (int i = 0; i < N; i++) {
      …
    }
  }
}
```



Case study: FIR Filter: Pipelining

```
void fir(int input[SIZE], int
out[SIZE]) {
  // FIR coefficients
  int coeff[N] = {13,-2,9,…,74};
  // shift registers
  int shift_reg[N] = {0};
  for (int n = 0; n < SIZE; n++) {
    #pragma HLS pipeline II=1
    int acc = 0;
    …
    output[n] = acc;
  }
}
```



High-Level Synthesis

- Exponential growth in HW design complexity calls for higher level of design abstraction.
- To raise hardware design level of abstraction
  - Use **high-level languages,**
    - e.g., C++ instead of Verilog
  - Use **automation,**
    - e.g., HLS tools to generate Verilog
  - Use **libraries/generators,**
    - e.g., MatchLib from NVIDIA

- Design Productivity
- High-Level Synthesis
- **Chisel**
- SystemVerilog (next lecture)

Adapted from slides by Hasan Genc and Brendan Sweeney

Verilog/VHDL

- Old, extremely popular
- If you get a job writing RTL, it will probably be in one of these
- Cons
  - Poor parameterizability
    - Some companies write scripts to generate Verilog
  - Weak type-safety (esp. Verilog)
    - Weak notion of signed vs unsigned
    - Easy to accidentally truncate bits
  - Easy to generate non-synthesizable hardware by mistake
    - Originally a simulation language
      - Hardware primitives must be *inferred*
    - Huge market for Verilog linters!

Old school hardware engineers will tell you that it's fine. It's fine that the language is so counter-intuitive that almost all people who initially approach Verilog write code that's not just wrong but nonsensical.

Dan Luu

SystemVerilog

- Superset of Verilog
- New features
  - Especially for verification
  - More expressive
    - Clearer datatypes, better abstractions, etc.
- More about SystemVerilog from Vighnesh (next lecture)
- Cons:
  - Also possible to generate non-synthesizable hardware by mistake
  - Some advanced features are not supported by all vendors

Chisel

- Embeds an HDL in Scala
  - Verilog/VHDL/SystemVerilog are standalone languages
- Chisel describes the hardware…
  - …and Scala parameterizes it
- Pros:
  - No confusing separation between synthesizable and simulation-only hardware
  - Easy to define new types and *typeclasses*
    - You don’t have to always think in terms of “bits”
    - Operator overloading, implicit datatype conversion, etc.
  - Powerful metaprogramming and parameterizability
    - Don’t just write hardware!
    - Write *programs* that generate hardware!
    - Just look at *Diplomacy* for proof

But surely there must be a catch?

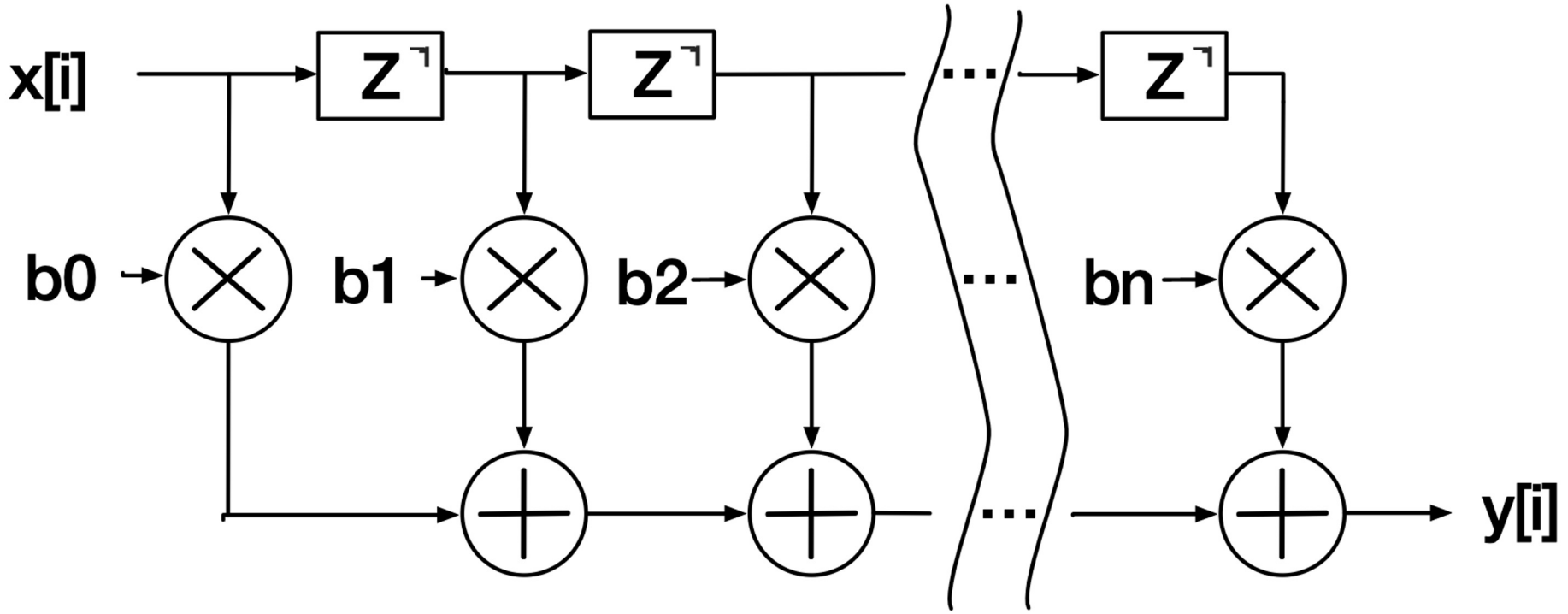
- Chisel cons:
  - Awkward support for latches and asynchronous hardware
  - Verification flow is not (yet) as advanced as with SystemVerilog
    - Some examples of Chisel working with UVM and SVA exist/being worked on (ChiselVerify, Dynamic Verification Library (verif), CHA)
  - Some people are scared of Scala
    - Some parts of the Scala ecosystem are painful to use
      - SBT, JVM, …
- We will also talk about issues specific to the Chisel **compiler** later

What about other options?

- There are *lots* of HDLs
- Embedded in lots of different languages
  - Including Python and Haskell
- Examples
  - BlueSpec
  - SpinalHDL
  - Magma
  - PyRTL
  - Clash
  - MyHDL

Let’s Make an FIR Filter

(example from the Chisel website)



Prettier Verilog

This is how beginner typically use Chisel:

```
// 3-point moving sum implemented in the style of a FIR filter
class MovingSum3(bitWidth: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))
    val out = Output(UInt(bitWidth.W))
  })

  val z1 = RegNext(io.in)
  val z2 = RegNext(z1)

  io.out := (io.in * 1.U) + (z1 * 1.U) + (z2 * 1.U)
}
```

Where Chisel Shines

- Write *generators*, not *circuits*
- Chisel makes functional programming *easy*
  - Maps, reduces, tabulates…
- But it doesn't *force* functional programming on you
  - For-loops and stateful statements are OK too!

```
// Generalized FIR filter parameterized by the convolution coefficients
class FirFilter(bitWidth: Int, coeffs: Seq[UInt]) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))
    val out = Output(UInt(bitWidth.W))
  })
  // Create the serial-in, parallel-out shift register
  val zs = Reg(Vec(coeffs.length, UInt(bitWidth.W)))
  zs(0) := io.in
  for (i <- 1 until coeffs.length) {
    zs(i) := zs(i-1)
  }

  // Do the multiplies
  val products = VecInit.tabulate(coeffs.length)(i => zs(i) * coeffs(i))

  // Sum up the products
  io.out := products.reduce(_ + _)
}
```

Where Chisel *Especially* Shines

- Types!
  - Powerful typeclasses
  - Great type-checking
    - Fewer bugs!

```
// Generalized FIR filter parameterized by the convolution coefficients
class FirFilter[T <: Data : Ring](t: T, coeffs: Seq[T]) extends Module {
  val io = IO(new Bundle {
    val in = Input(t)
    val out = Output(t)
  })
  // Create the serial-in, parallel-out shift register
  val zs = Reg(Vec(coeffs.length, t))
  zs(0) := io.in
  for (i <- 1 until coeffs.length) {
    zs(i) := zs(i-1)
  }

  // Do the multiplies
  val products = VecInit.tabulate(coeffs.length)(i => zs(i) * coeffs(i))

  // Sum up the products
  io.out := products.reduce(_ + _)
}
```

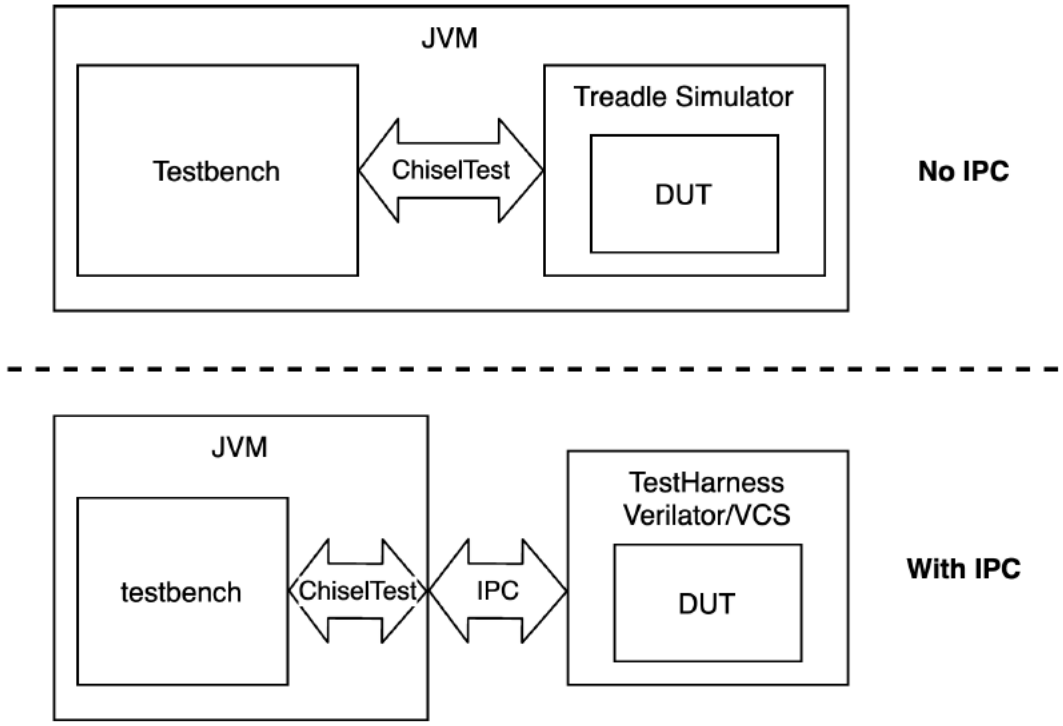
Generators Make Hardware Reusable

Not unique to Chisel, but Chisel makes it *easier*

```
val movingSum3Filter = Module(new FirFilter(UInt(8.W) Seq(1.U, 1.U, 1.U)))
val delayFilter = Module(new FirFilter(SInt(16.W) Seq(0.S, 1.S)))
val triangleFilter = Module(new FirFilter(Float(8,24), Seq(1.Fl, 2.Fl, 3.Fl, 2.Fl, 1.Fl)))
```

ChiselTest – A test harness for CHISEL RTL

```
import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec

  class SimpleTestExpect extends AnyFlatSpec
      with ChiselScalatestTester {
    "DUT" should "pass" in {
      test(new DeviceUnderTest) { dut =>
        dut.io.a.poke(0.U)
        dut.io.b.poke(1.U)
        dut.clock.step()
        dut.io.out.expect(0.U)
        dut.io.a.poke(3.U)
        dut.io.b.poke(2.U)
        dut.clock.step()
        dut.io.out.expect(2.U)
      }
    }
  }
```



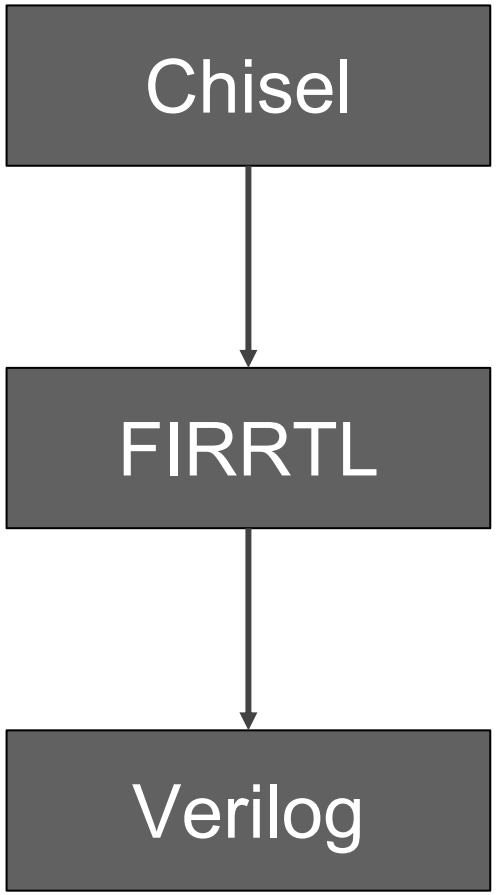
- Based on ScalaTest
  - Set input values with *poke*
  - Read output values with *peek*
  - Compare the values with *expect*

Chisel high-points

- List manipulations are easy
  - Map, reduce, scan, etc.
- Parameterizing across *types* is easy
  - With an excellent type system to make sure you don't make mistakes
- Reusable hardware
  - Very convenient standard library
    - Queues, handshake-based interfaces, bit-manipulation functions, counters, etc.
  - RocketChip components are (kind-of) plug-and-play
    - TLBs, caches, entire CPUs, etc.
- Excellent IDE support
  - Write hardware in IntelliJ

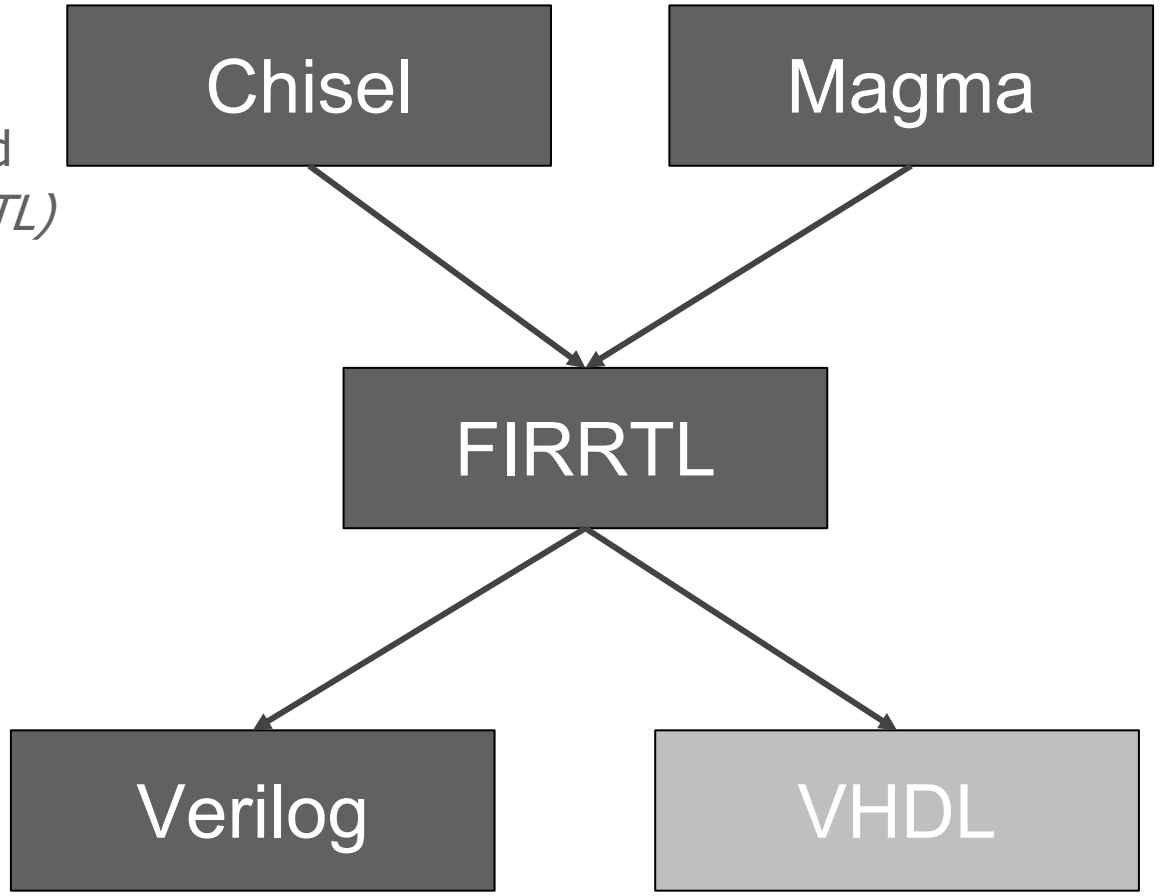
Chisel Compiler

- Chisel has an intermediate representation (IR) called *FIRRTL (Flexible Intermediate Representation for RTL)*
  - *Chisel:* Clang
  - *FIRRTL:* LLVM
  - *Verilog:* Machine code
- FIRRTL supports *transformations* on FIRRTL data structures (abstract syntax tree – AST)
  - Early area estimations
  - Add hardware counters and instrumentation
  - Optimize RTL for ASIC vs FPGA



Chisel Compiler

- Chisel has an intermediate representation (IR) called *FIRRTL (Flexible Intermediate Representation for RTL)*
  - *Chisel:* Clang
  - *FIRRTL:* LLVM
  - *Verilog:* Machine code
- FIRRTL supports *transformations* on FIRRTL data structures (abstract syntax tree – AST)
  - Early area estimations
  - Add hardware counters and instrumentation
  - Optimize RTL for ASIC vs FPGA



CHISEL -> FIRRTL -> Verilog

CHISEL

```
// 3-point moving sum implemented in the style of a FIR filter
class MovingSum3(bitWidth: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(bitWidth.W))
    val out = Output(UInt(bitWidth.W))
  })

  val z1 = RegNext(io.in)
  val z2 = RegNext(z1)

  io.out := (io.in * 1.U) + (z1 * 1.U) + (z2 * 1.U)
}
```

FIRRTL

```
1  circuit FirFilter :
2    module FirFilter :
3      input clock : Clock
4      input reset : UInt<1>
5      output io : {flip in : UInt<8>, out : UInt<8>}
6
7      reg zs : UInt<8>[3], clock
8      zs[0] <= io.in
9      zs[1] <= zs[0]
10     zs[2] <= zs[1]
11     node _T = mul(zs[0], UInt<1>("h01"))
12     node _T_1 = mul(zs[1], UInt<1>("h01"))
13     node _T_2 = mul(zs[2], UInt<1>("h01"))
14     wire products : UInt<9>[3]
15     products[0] <= _T
16     products[1] <= _T_1
17     products[2] <= _T_2
18     node _T_3 = add(products[0], products[1])
19     node _T_4 = tail(_T_3, 1)
20     node _T_5 = add(_T_4, products[2])
21     node _T_6 = tail(_T_5, 1)
22     io.out <= _T_6
```

Verilog

```
1  module FirFilter(
2    input        clock,
3    input        reset,
4    input  [7:0] io_in,
5    output [7:0] io_out
6  );
7    reg [7:0] zs_0;
8    reg [7:0] zs_1;
9    reg [7:0] zs_2;
10   wire [8:0] products_0 = zs_0 * 1'h1;
11   wire [8:0] products_1 = zs_1 * 1'h1;
12   wire [8:0] products_2 = zs_2 * 1'h1;
13   wire [8:0] _T_4 = products_0 + products_1;
14   wire [8:0] _T_6 = _T_4 + products_2;
15   assign io_out = _T_6[7:0];
16   always @(posedge clock) begin
17     zs_0 <= io_in;
18     zs_1 <= zs_0;
19     zs_2 <= zs_1;
20   end
21 endmodule
```

- FIR Filter Example

Advantages of the Chisel Compiler

- Optimization passes can be shared across hardware designs
  - A similar concept exists for software compilers
- Attach arbitrary metadata for any Chisel expression
  - The FIRRTL compiler can use that metadata later
    - E.g. don't const-propagate, FPGA register preset
- FIRRTL makes it easy to build *tooling* for RTL designs
  - Automatically create:
    - Test harness generators
    - Macro compilers
      - Useful for SRAMs!
    - Simulators
  - Can benefit from other people's FIRRTL optimizers

Problems With the Chisel Compiler

- Generated Verilog can be unreadable (e.g. fully unrolled arrays, …)
  - This can make debugging and verification more difficult
    - Especially for some automated verification tools
  - Some HDLs do a much better job with this
- Slow
- Memory hungry
  - Makes it hard to run this on a low-end laptop

Chisel/FIRRTL is an *integral* part of the “Berkeley stack”

- CPUs
  - *RocketChip:* In-order CPU
    - 2k GitHub stars, 406 citations
  - *BOOM:* Out-of-order CPU
  - *Sodor:* Educational
- Accelerators
  - *Hwacha:* Vectors
  - *Gemmini:* DNNs
  - *Genesis:* Genomics
  - *Protobuf accelerator*
- Hardware design frameworks
  - *Chipyard*
- Simulation frameworks
  - *FireSim*
  - *MIDAS*

People outside Berkeley use Chisel too!

- Industry
  - Google, IBM, SiFive, Lampro Mellon, Intel
    - Google’s Edge TPU was designed with Chisel
- Academia
  - Cornell, Stanford, Technical University of Denmark, LBL, Boston University, UC Davis, Peking

Some questions to think about

- Is Chisel focusing on the right problems?
- Why do so many people complain about Verilog/VHDL, but they still keep using it?
- Are HDLs even worth creating anymore?
  - Why not focus on HLS instead?
  - What other abstractions for generating hardware are possible?