AI ASIC: Design and Practice (ADaP)

Fall 2024

Latest Hardware Design Languages

燕博南

HDL History

Verilog originated at Automated Integrated Design Systems (renamed Gateway) in 1985. Acquired by Cadence in 1989.

Invented as simulation language. Synthesis was an afterthought. Many of the basic techniques for synthesis were developed at Berkeley in the 80's and applied commercially in the 90's.

Around the same time as the origin of Verilog, the US Department of Defense developed VHDL:

- A double acronym! VSIC (Very High-Speed Integrated Circuit) HDL
- Because it was in the public domain it began to grow in popularity.

HDL History

- Afraid of losing market share, Cadence opened Verilog to the public in 1990.
- An IEEE working group was established in 1993, and ratified IEEE Standard 1394 (Verilog) in 1995.
- Verilog is the language of choice of Silicon Valley companies, initially because of high-quality tool support and its similarity to C-language syntax.

- VHDL is still popular within the government, in Europe and Japan, and some Universities.

HDL History

- Latest Verilog version is “System Verilog”
- In 1997, Superlog (derived from Super and Verilog), for system specification, hardware design, hardware verification, and software development.
- In 2002, Co-Design Automation donated the Superlog language to Accellera, and the bulk of the verification functionality was based on the OpenVera language, which was donated by Synopsys.
- In 2005, SystemVerilog was first adopted in **IEEE standard**.
- Other alternatives these days:
  - Bluespec (MIT spin-out) models digital systems using “guarded atomic actions”
  - C-to-gates Compilers (ex: Cadence C-to-Silicon Compiler, Vivado HLS)

What are the problems with Verilog?

Hardware Construction Language

Part I

HDL is not enough - I

- Designed as a simulation language. “Discrete Event Semantics”
  - Many constructs don’t synthesize: ex: deassign, timing constructs
  - Others lead to mysterious results: for-loops
  - Difficult to understand synthesis implications of procedural assignment (always blocks), and blocking versus non-blocking assignments
  - In common use, most users ignore much of the language and stick to a very strict “style”. Companies use rules and run lint style checkers. Nonetheless leads to confusion (particularly for beginners), and bugs.
  - Few meta-programming support
    - Vs. embedded TCL scripting

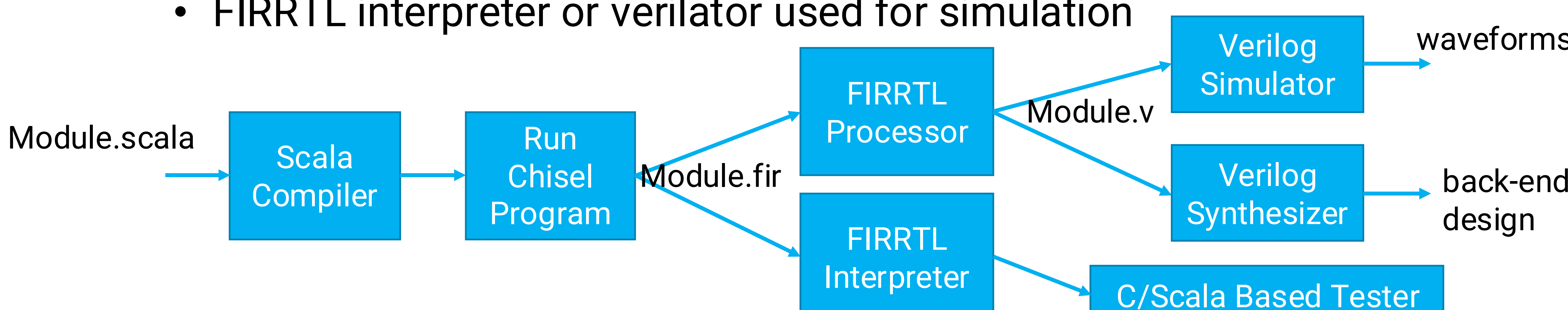
Chisel

- **Chisel: Constructing Hardware in a Scala Embedded Language**
  - Powerful “metaprogramming” model for building circuit generators
- Why embedded?
  - Avoid the hassle of writing and maintaining a new programming language (most of the work would go into the non-hardware specific parts of the language anyway)
- Why Scala?
  - Brings together the best of many others: Java JVM, functional programming, OO programming, strong typing, type inference

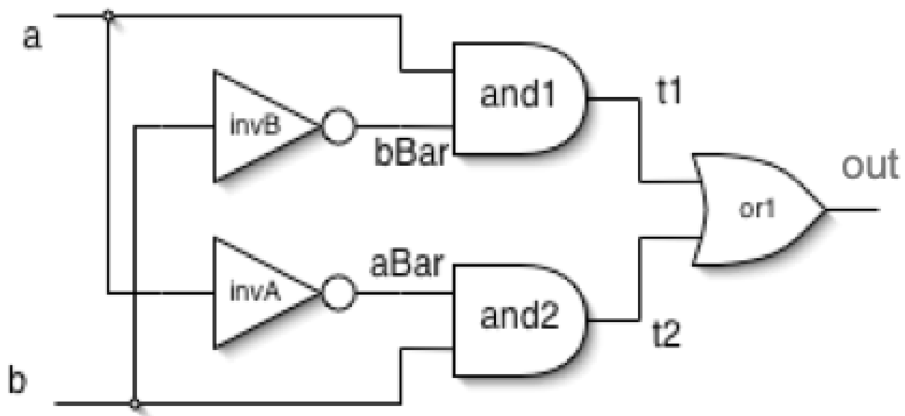
Chisel Workflow

Scala Compiler generates an executable (Chisel program)

- Execution of the Chisel program:
  - generates an internal data structure and output called FIRRTL (flexible intermediate representation for RTL)
  - FIRRTL "processor":
    - resolves wire widths
    - checks connectivity
    - generates target output (verilog for now)
    - FIRRTL interpreter or verilator used for simulation



Introduction to Chisel Language



- **Simple Logic**
- `val out = (a & ~b) | (~a & b)`

- **Function Abstraction**
- `def XOR (a:Bits, b:Bits)=(a&~b)|(~a&b)`
- `val z =(x & XOR(x,y)) | (XOR(x,y) & y)`

Datatypes in Chisel

- Chisel datatypes are used to specify the type of values held in state elements or flowing on wires.

Bits	Raw collection of bits (parent type)
SInt	Signed integer number
UInt	Unsigned integer number
Bool	Boolean

- All signed numbers represented as 2's complement

```
val out = (a & ~b) | (~a & b)
```

```
val out: UInt = (a & ~b) | (~a & b)
```

Bundles

```
class FIFOInput extends Bundle {
      val rdy = Output(Bool()) //Indicates if FIFO has space
      val data = Input(UInt(32.W)) //values to be enqueued
      val euq = Input(Bool()) //assert to enqueue data
}
```

Instantiation:

```
Val jonsIO = new FIFOInput
```

Literals

`”ha”.U`	Hexadecimal 4-bit literal of type Bits
`”o12”.U`	Octal 4-bit literal of type Bits
`”b1010”.U`	Binary 4-bit literal of type Bits
`5.S`	Signed decimal 4-bit literal of type Fix
`-8.S`	Negative decimal 4-bit literal of type Fix
`5.U`	Unsigned decimal 4-bit literal of type UFix
`true.B`	Literals for type Bool, from Scala Boolean literals
`false.B`	
`”ha”.asUInt(8.W)`	hexadecimal 8-bit literal of type Bits, 0-extended
`-5.asSInt(32.W)`	32-bit decimal literal of type Fix, signed-extended

Built-in Operators

Operators:

Chisel	Explanation	Width
`!x`	Logical NOT	1
`x && y`	Logical AND	1
`x || y`	Logical OR	1
`x(n)`	Extract bit, 0 is LSB	1
`x(n, m)`	Extract bitfield	`n - m + 1`
`x << y`	Dynamic left shift	`w(x) + maxVal(y)`
`x >> y`	Dynamic right shift	`w(x) - minVal(y)`
`x << n`	Static left shift	`w(x) + n`
`x >> n`	Static right shift	`w(x) - n`
`Fill(n, x)`	Replicate x, n times	`n * w(x)`
`Cat(x, y)`	Concatenate bits	`w(x) + w(y)`
`Mux(c, x, y)`	If c, then x; else y	`max(w(x), w(y))`
`~x`	Bitwise NOT	`w(x)`
`x & y`	Bitwise AND	`max(w(x), w(y))`
`x | y`	Bitwise OR	`max(w(x), w(y))`
`x ^ y`	Bitwise XOR	`max(w(x), w(y))`
`x === y`	Equality(triple equals)	1
`x =/= y`	Inequality	1
`x + y`	Addition	`max(w(x),w(y))`
`x +% y`	Addition	`max(w(x),w(y))`
`x +& y`	Addition	`max(w(x),w(y))+1`
`x - y`	Subtraction	`max(w(x),w(y))`
`x -% y`	Subtraction	`max(w(x),w(y))`
`x -& y`	Subtraction	`max(w(x),w(y))+1`
`x * y`	Multiplication	`w(x)+w(y)`
`x / y`	Division	`w(x)`
`x % y`	Modulus	`bits(maxVal(y)-1)`
`x > y`	Greater than	1
`x >= y`	Greater than or equal	1
`x < y`	Less than	1
`x <= y`	Less than or equal	1
`x >> y`	Arithmetic right shift	`w(x) - minVal(y)`
`x >> n`	Arithmetic right shift	`w(x) - n`

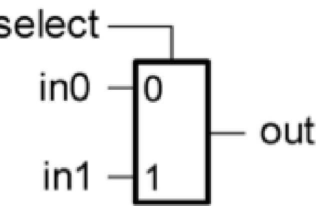
UInt bit-reduction methods:

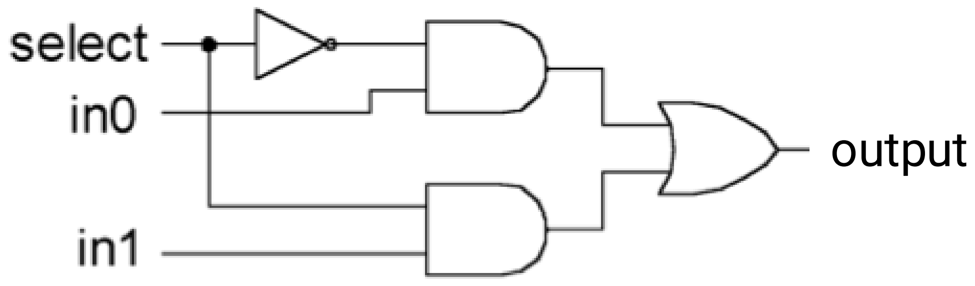
Chisel	Explanation	Width
`x.andR`	AND-reduce	1
`x.orR`	OR-reduce	1
`x.xorR`	XOR-reduce	1

As an example to apply the andR method to an SInt use x.asUInt.andR

Modules

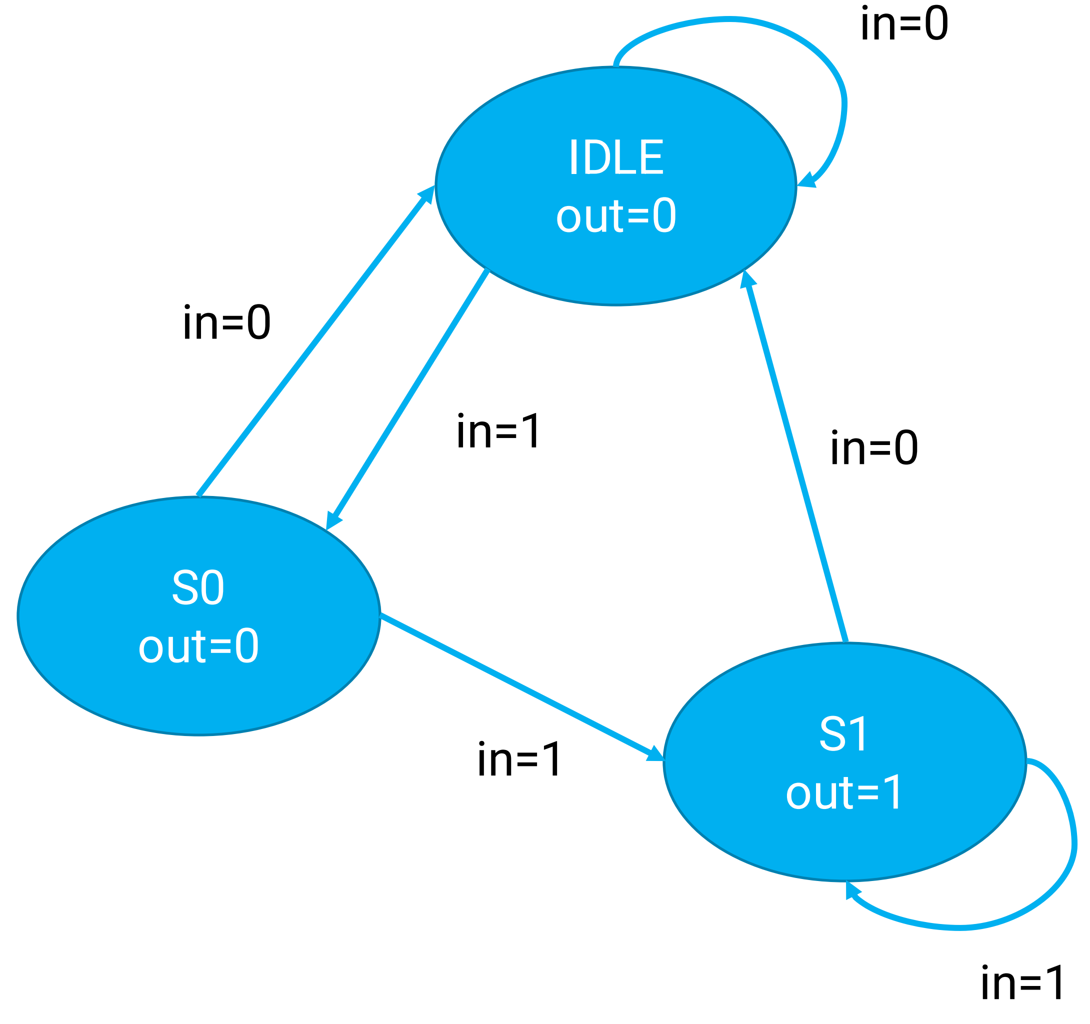
```
class Mux2 extends Module {
    val io = IO(new Bundle{
        val select = Input(UInt(1.W))
        val in0 = Input(UInt(1.W))
        val in1 = Input(UInt(1.W))
        val out = Output(UInt(1.W))
})

io.out := (io.select & io.in1) |
               (~io.select & io.in0)

}
```





Example-FSM

```
class MyFSM extends Module {
    val io = IO(new Bundle {
    val in = Input(Bool())
    val out = Output(Bool())
})
    val IDLE :: S0 :: S1 :: Nil = Enum(3)
    val state = Reg(init = IDLE);
    when (state === IDLE) {
        when (io.in) { state := S0 }
    }
    when (state === S0) {
        when (io.in) { state := S1 }
        .otherwise { state := IDLE }
    }
    when (state === S1) {
        when (!io.in) { state := IDLE }
    }
    io.out := state === S1;
}
```



Reference for Chisel

Chisel3 homepage: https://www.chisel-lang.org

Chisel Doc

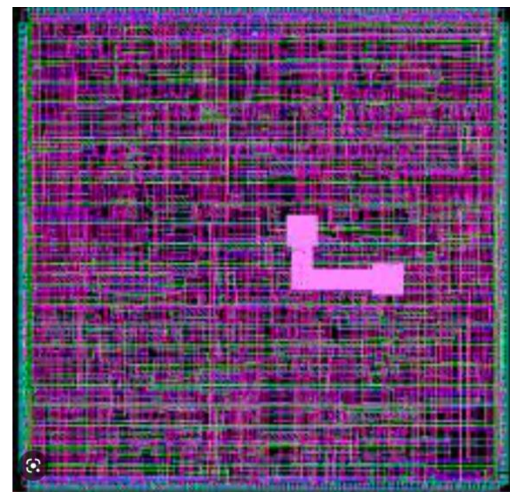
Chisel-Bootcamp

Chisel-Book Tutorial (中文版)

Chisel-Template

Chisel is still under development

- Debug is somewhat hard
  - With automatically generated intermediate wires
  - Still lack good back-annotation support
- Layout efforts are disturbing
  - With automatically generated intermediate wires
  - Parametrized front-end design does not propagate to back-end



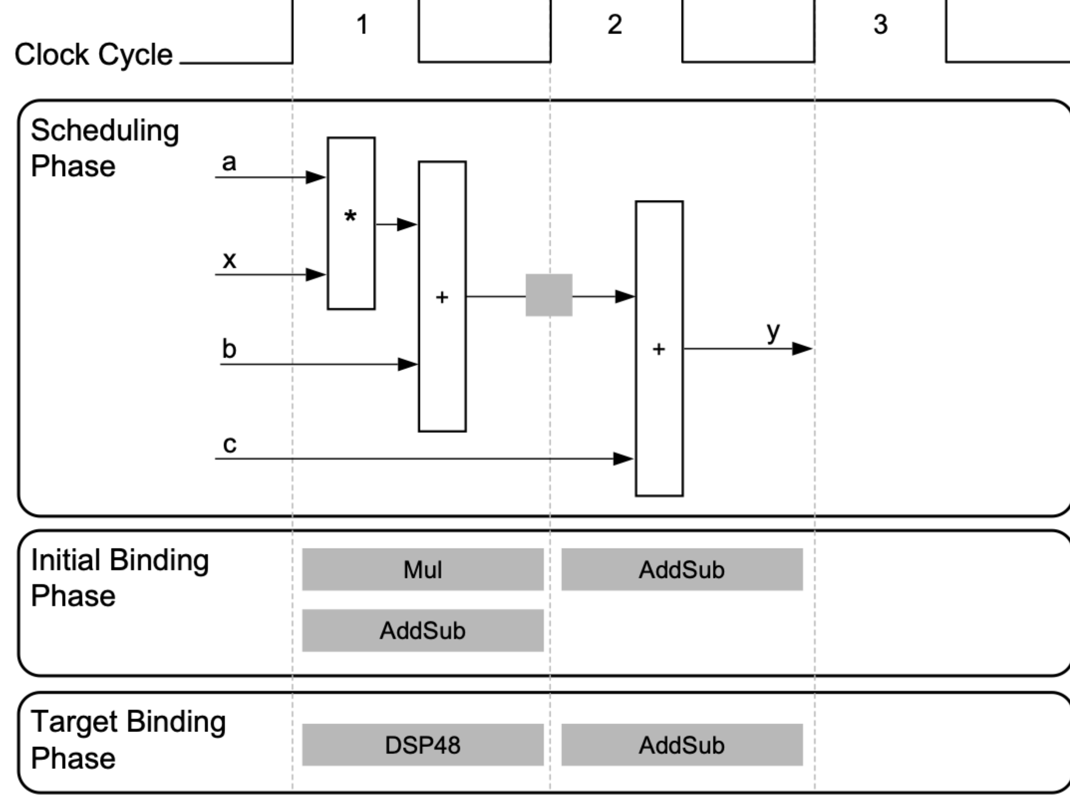
High-Level Synthesis

Part II

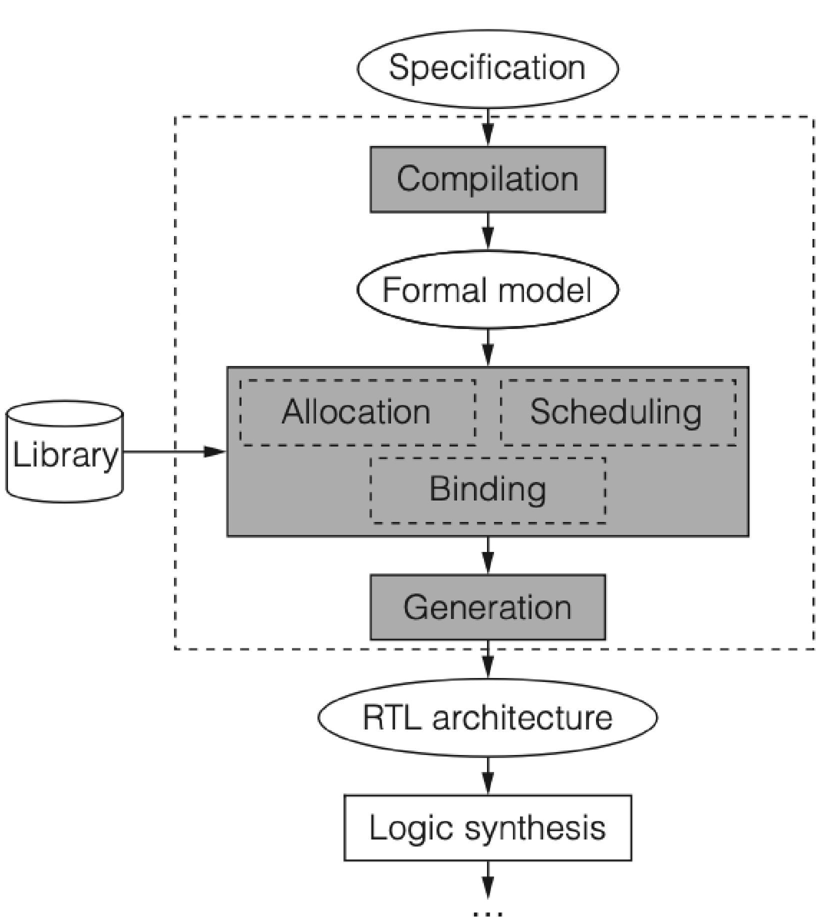
HDL is not enough - II

- Software algorithms are hard to be converted into RTL
  - There is no ”timing” in algorithms
  - Software don’t care wires and registers
- Think about mapping a CNN layer into Verilog
  - Too complicated

```
int foo(char x, char a, char b, char c) {
  char y;
  y = x*a+b+c;
  return y
}
```



High-Level Synthesis (HLS)



Key concepts

Starting from the high-level description of an application, an RTL component library, and specific design constraints, an HLS tool executes the following tasks (see Figure 1):

1. compiles the specification,
2. allocates hardware resources (functional units, storage components, buses, and so on),
3. schedules the operations to clock cycles,
4. binds the operations to functional units,
5. binds variables to storage elements,
6. binds transfers to buses, and
7. generates the RTL architecture.

Coussy, Philippe, et al. "An introduction to high-level synthesis." IEEE Design & Test of Computers 26.4 (2009): 8-17.

Typical Hardware Architectures

Control inputs

Control signals

State register (SR)

Next-state logic

Output logic

Status signals

Controller

Control outputs

RF/Scratch pad

Bus 1

Bus 2

ALU

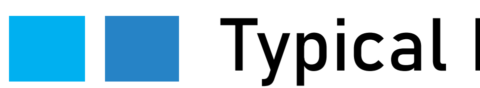
MUL

Memory

Bus 3

Data path

Coussy, Philippe, et al. "An introduction to high-level synthesis." IEEE Design & Test of Computers 26.4 (2009): 8-17.

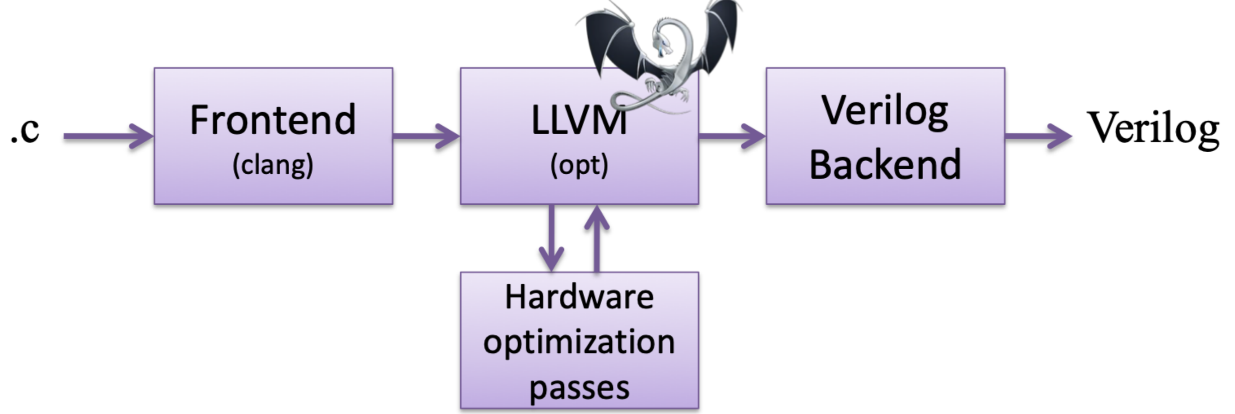


Typical HLS Compilers

LLVM Based Compiler	Xilinx	Intel FPGA

LLVM Based HLS

Write the C code and get it into the LLVM compiler



Intermediate Representation (IR)

```
entry:
    %A = add i32 %B, 5
    %C = icmp eq i32 %A, 0
    %br i1 %C, label %next, label %entry
    ...
```

```
case (state)
  ST0: begin
    A <= B + 5;
    state <= ST1;
  end
  ST1: begin
    C <= (A  == 0)
    state <= ST2;
  end
  ST2: begin
    if (C)
        state <= ST9;
    else
        state <= ST0;
    end
...
endcase
```

LLVM Based HLS

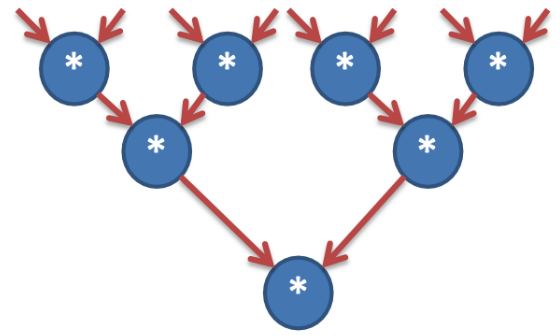
Time

13 cycles

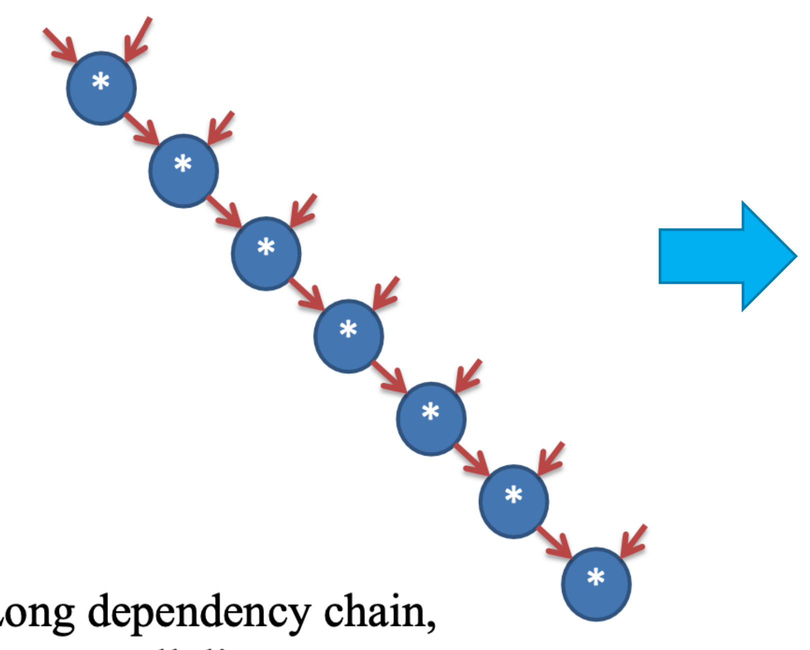
Mult
Mem
Other

Load A
Load B
Load C
Mult
Mult
Mult
Add
Xor
Store E
Store D

Arithmetic tree height reduction



Short dependency chain,
High parallelism

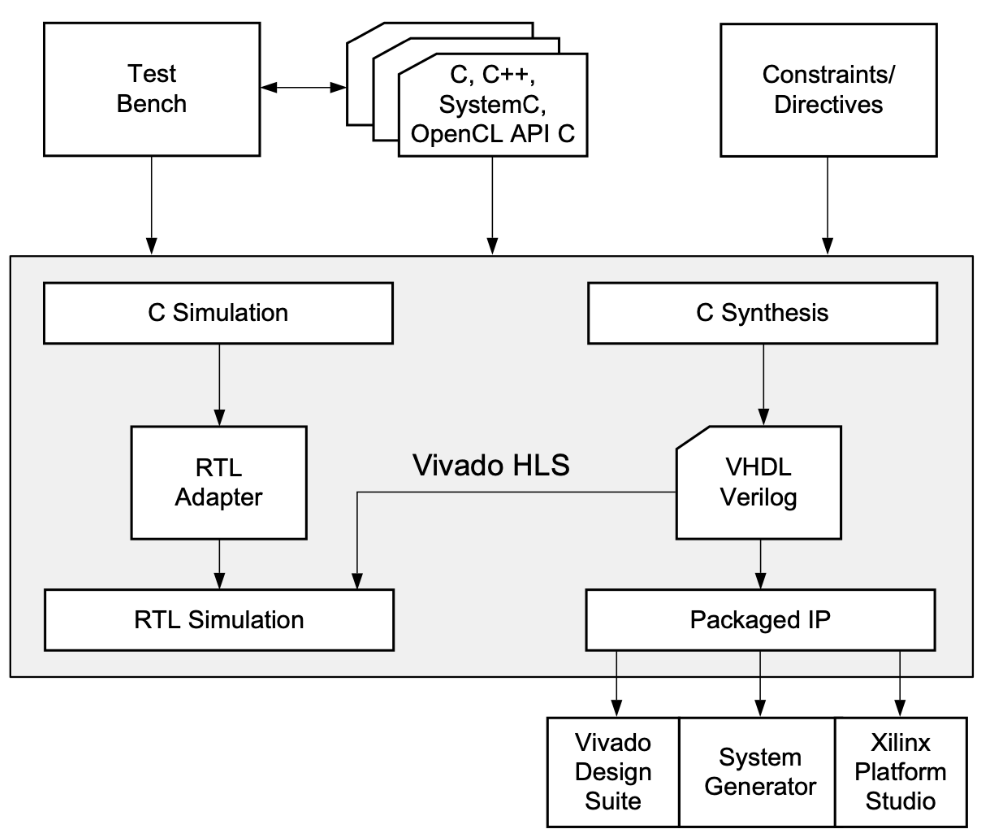


Long dependency chain,
Low parallelism

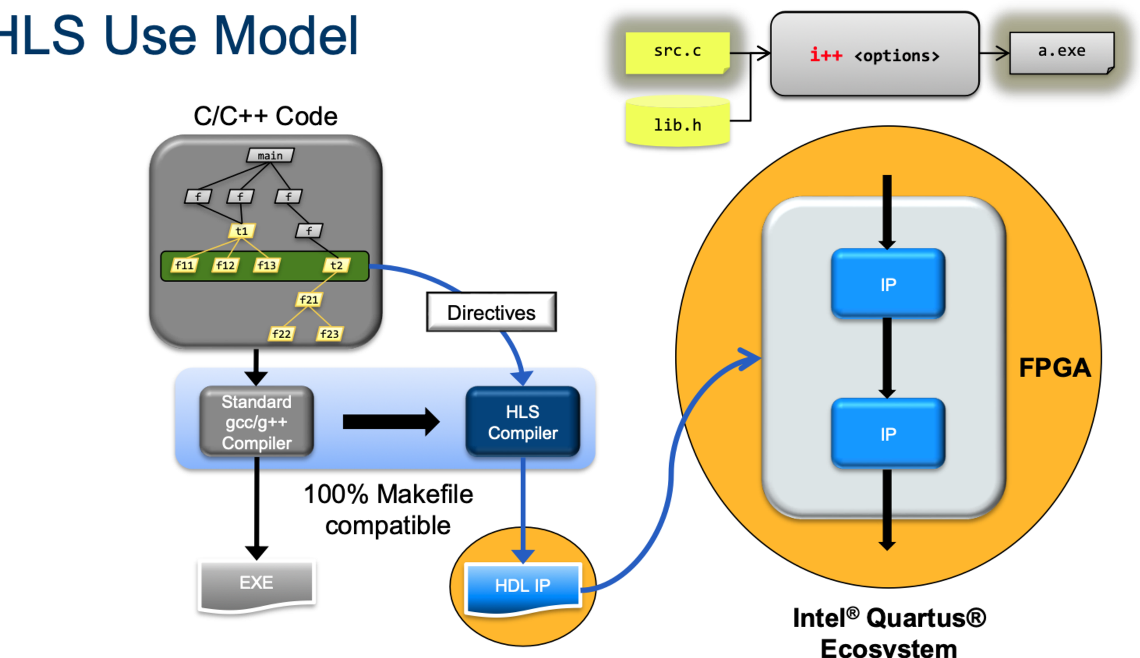
Binding can be done by the users or the compilers

Typical Flows/Tools of HLS

Xilinx HLS Flow:



Intel HLS Flow:



Example of Revising C Code For HLS :: Add HLS Tags

```
template<typename T, int K>
  static void convolution_strm(
  int width,
  int height,
  hls::stream<T> &src,
  hls::stream<T> &dst,
  const T *hcoeff,
  const T *vcoeff)
{
  #pragma HLS DATAFLOW
  #pragma HLS ARRAY_PARTITION variable=linebuf dim=1 complete
  hls::stream<T> hconv("hconv");
  hls::stream<T> vconv("vconv");
  // These assertions let HLS know the upper bounds of loops
  assert(height < MAX_IMG_ROWS);
  assert(width < MAX_IMG_COLS);
```

```
 assert(width < MAX_IMG_COLS);
  assert(vconv_xlim < MAX_IMG_COLS - (K - 1));
  // Horizontal convolution
  HConvH:for(int col = 0; col < height; col++) {
    HConvW:for(int row = 0; row < width; row++) {
  #pragma HLS PIPELINE
      HConv:for(int i = 0; i < K; i++) {
} }
  }
  // Vertical convolution
  VConvH:for(int col = 0; col < height; col++) {
    VConvW:for(int row = 0; row < vconv_xlim; row++)
{
  #pragma HLS PIPELINE
  #pragma HLS DEPENDENCE variable=linebuf inter false
      VConv:for(int i = 0; i < K; i++) {
    }
}
  Border:for (int i = 0; i < height; i++) {
    for (int j = 0; j < width; j++) {
  #pragma HLS PIPELINE
```

HLS is still under development

- Too many hidden techniques while use C++/SystemC
- ASIC HLS flow is harder than FPGA flow
  - Why?
- Tools under development!

Future

- What is the best design flow you think?
  - Integrate analog/digital
  - Automatically tune parameters
  - Readable (at least for LLM)
  - …