CS250
VLSI Systems Design

Lecture 3: Hardware Design Languages

Fall 2011

Krste Asanović, John Wawrzynek
with
John Lazzaro
and
Brian Zimmer (TA)

Outline

- Background and History of Hardware Description
- Brief Introduction to Chisel Part 1
- Next Monday, Chisel Part 2

Design Entry

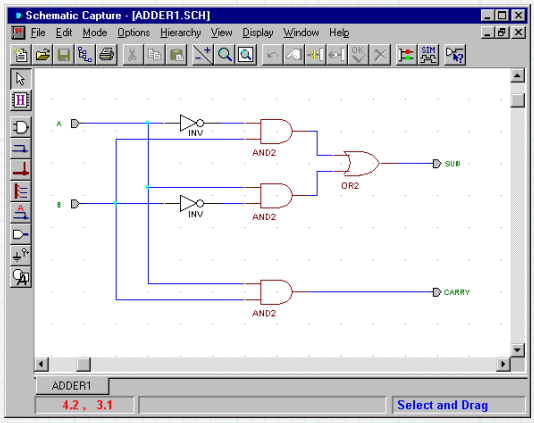
- Schematic entry/editing used to be the standard method in industry and universities.
- Used in commonly until ~2002

☺Schematics are intuitive. They match our use of gate-level or block diagrams.

☺Somewhat physical. They imply a physical implementation.

☹Require a special tool (editor).

☹Unless hierarchy is carefully designed, schematics can be confusing and difficult to follow on large designs.



- Hardware Description Languages (HDLs) are the new standard
  - except for PC board design, where schematics are still used.

Hardware Description Languages

- Originally invented for simulation.
  - Now "logic synthesis" tools exist to automatically convert from HDL source to circuits.
  - High-level constructs greatly improves designer productivity.
  - However, this may lead you to falsely believe that hardware design can be reduced to writing programs!*
- Basic Idea: language constructs describe circuits with two basic forms:
  - **Structural descriptions:** connections of components. Nearly one-to-one correspondence to with schematic diagram.
  - **Behavioral descriptions:** use high-level constructs (similar to conventional programming) to describe the circuit function.

"Structural" example:

```
Decoder(output x0,x1,x2,x3; inputs a,b)}
wire abar, bbar;
     inv(bbar, b);
     inv(abar, a);
     and(x0, abar, bbar);
     and(x1, abar, b   );
     and(x2, a,    bbar);
     and(x3, a,    b   );
  }
```

"Behavioral" example:

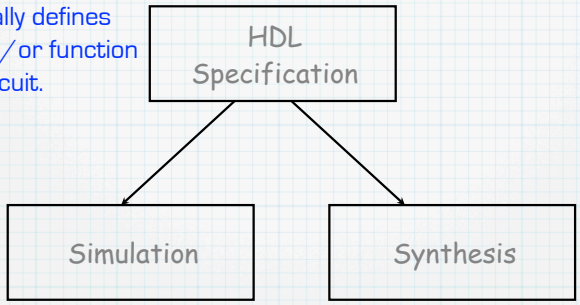
```
  Decoder(output x0,x1,x2,x3;
              inputs a,b)
  {
     case [a b]
          00: [x0 x1 x2 x3] = 0x1;
          01: [x0 x1 x2 x3] = 0x2;
          10: [x0 x1 x2 x3] = 0x4;
          11: [x0 x1 x2 x3] = 0x8;
      endcase;
  }
```

Warning: this is a fake HDL!

*Describing hardware with a language is similar, however, to writing a parallel program.

Standard Design Methodology

Hierarchically defines structure and/or function of circuit.



Verification: Does the design behave as required with regards to function (and timing, and power consumption)?

Maps specification to resources of implementation platform (FPGA or ASIC).

Note: This in not the entire story. Other tools are useful for analyzing HDL specifications. More on this later.

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.
- 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.
- Most major CAD frameworks now support both.
- Latest Verilog version is "System Verilog" .
- Other alternatives these days:
  - Bluespec (MIT spin-out) models digital systems using "guarded atomic actions"
  - C-to-gates Compilers (ex: Synfora PICO, AutoESL)

Verilog "Issues"

- 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
  - Your favorite complaint here!
  - In common use, most users ignore much of the language and stick to a very strict "style". Large companies post use rules and run lint style checkers. Nonetheless leads to confusion (particularly for beginners), and bugs.

- The real power of a textual representation of circuits is the ability to write circuit "compilers". Verilog has very weak "meta-programming" support". Simple parameter expressions, generate loops and case.
- Various hacks around this over the years, ex: embedded TCL scripting.

```
// Gray-code to binary-code converter
module gray2bin1 (bin, gray);
    parameter SIZE = 8;
    output [SIZE-1:0] bin;
    input  [SIZE-1:0] gray;

    genvar i;

    generate for (i=0; i<SIZE; i=i+1)
begin:bit
      assign bin[i] = ^gray[SIZE-1:i];
    end endgenerate
  endmodule
```


Chisel

Constructing Hardware In a Scala Embedded Language

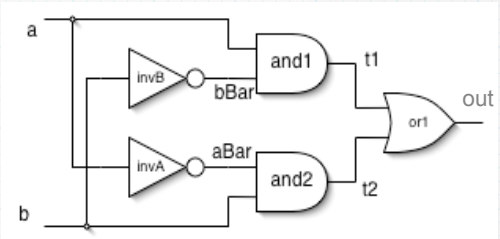
- Experimental attempt at a fresh start to address these issues.
  - Clean simple set of design construction primitives, just what is needed for RTL design (later support for UTL design)
  - Powerful "metaprogramming" model for building circuit generators
- Why embedded?
  - Avoid the hassle of writing 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.
- Still very new. Bugs will show up. Your feedback is needed.
- In class, brief presentation of basics. Ask questions.
- First tutorial document available online. Formal reference later.

Chisel Acknowlegements

Jonathan Bachrach	Principal developer
Huy Vo	Undergrad research assistant, Chisel Developer
Brian Richards	First actual user (translated FPU Verilog code to Chisel)
Yunsup Lee, Andrew Waterman	Early users for processor design mapped to FPGAs
Scott Beamer	Early user continues to write chisel routers and give feedback
Chris Celio	Pushed the design, given feedback, and written the most Chisel code in writing RiscV code
Chris Batten	Fast C++ template library that inspired Chisel fast simulator
James Martin and Alex Williams	Work on writing and translating network and memory controllers and non-blocking caches
Anonymous	Participants in first Chisel bootcamp

Simple Combinational Logic Example

```
// simple logic expression
(a & ~b) | (~a & b)
```

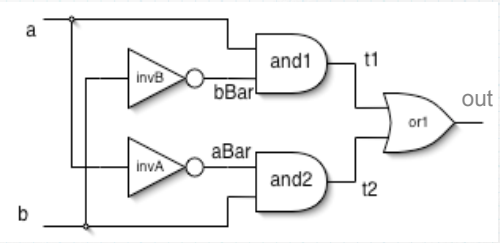


- Notes:
  - The associated logic circuits are not "executed". They are active always (like continuous assignment in Verilog).
  - Unlike Verilog, no built-in logic gates. Expressions instead.
  - The "variables", `a` and `b`, are "named wires", and were given names here because they are inputs to the circuit. Other wires don't need names.
  - Here we assumed that the inputs, and therefore all generated wires, are one bit wide, but the same expression would work for wider wires. The logic operators are "bitwise".
    - Chisel includes a powerful wire width inference mechanism.

Simple Combinational Logic Example

- In the previous example because the wires `a` and `b`, are named, each can be used in several places. Similarly we could name the circuit output:

```
// simple logic expression
val out =(a & ~b) | (~a & b)
```



  - The keyword `val` comes from Scala. It is a way to declare a program variable that can only be assigned once - a constant.
  - This way `out` will be generated at one place in the circuit and then "fanned-out" to other places where `out` appears.

```
// fan-out
val z =(a & out) | (out & b)
```

  - Another reason to name a wire is to help in debugging.

Functional Abstraction

- Naming wires and using fanout gives us a way to reuse an output in several places in the generated circuit. Function abstraction gives us a way to reuse a **circuit description**:

```
// simple logic function
def XOR (a: Bits, b: Bits) = (a & ~b) | (~a & b)
```

  - Here the function inputs and output are assigned the type `Bits`. More on types soon.
  - Now, wherever we use the `XOR` function, we get a copy of the associated logic. Think of the function as a "constructor".

```
// Constructing multiple copies
val z =(x & XOR(x,y)) | (XOR(x,y) & y)
```

  - Functions wrapping up simple logic are light-weight. This results hierarchy in your code, but no hierarchy in the Chisel output.
  - Later we'll see how to write polymorphic functions.
  - We'll see later that **Chisel Components** are used for building hierarchy in the resulting circuit.

Datatypes in Chisel

- **Chisel datatypes are used to specify the type of values held in state elements or flowing on wires.**
- Hardware circuits ultimately operate on vectors of binary digits, but more abstract representations for values allow clearer specifications and help the tools generate more optimal circuits.
- The basic types in Chisel are:

`Bits`	Raw collection of bits
`Fix`	Signed fixed-point number
`UFix`	Unsigned fixed-point number
`Bool`	Boolean

- Signed and unsigned integers are special cases of Fix and UFix, respectively.
- All signed numbers represented as 2's complement.
- Chisel supports several higher-order types: Bundles and Vecs.

Type Inference

- **Although it is useful to keep track of the types of your wires, because of Scala type inference, it is not always necessary to declare the type.**
- For instance in our earlier example:

```
// simple logic expression
val out =(a & ~b) | (~a & b)
```

the type of out was inferred from the types of `a` and `b` and the operators.

- If you want to make sure, or if there is not enough information around for the inference engine, you can always specify the type explicitly:

```
// simple logic expression
val out: Bits =(a & ~b) | (~a & b)
```

- Also, as we shall see, explicit type declaration is necessary in some situations.

Bundles

- **Chisel Bundles represent collections of wires with named fields.**
- Similar to "struct" in C. In chisel Bundles are defined as a class (similar to in C++ and Java):

```
class FIFOInput extends Bundle {
  val rdy = Bool('output)          // Indicates if FIFO has space
  val data = Bits(32, 'input)      // The value to be enqueued
  val enq = Bool('input)           // Assert to enqueue data
}
```

- Chisel has class methods for Bundle (i.e., automatic connection creation) therefore user created bundles need to "extend" class Bundle. (More later)
- Each field is given a name and defined with a constructor of the proper type and with parameters specifying width and direction.
- Instances of `FIFOInput` can now be made: `val jonsIO = new FIFOInput;`
- Bundle definitions can be nested and built into hierarchies,
- And are used to define the interface of "components" ...

Literals

- **Literals are values specified directly in your source code.**
- Chisel defines type specific constructors for specifying literals.

```
Bits("ha")      // hexadecimal 4-bit literal of type Bits
Bits("o12")     // octal 4-bit literal of type Bits
Bits("b1010")   // binary 4-bit literal of type Bits
Fix("5")        // signed decimal 4-bit literal of type Fix
Fix("-8")       // negative decimal 4-bit literal of type Fix
UFix("5")       // unsigned decimal 3-bit literal of type UFix
Bool(true)      // literals for type Bool, from Scala boolean literals
Bool(false)
```

- By default Chisel will size your literal to the minimum necessary width.
- Alternatively, you can specify a width value as a second argument:

```
Bits("ha", 8)   // hexadecimal 8-bit literal of type Bits, 0-extended
Fix("-5", 32)   // 32-bit decimal literal of type Fix, sign-extended
```

- Error reported if specified width value is less than needed.

Builtin Operators (1)

- Chisel defines a set of hardware operators for the builtin types.

Bitwise operators. Valid on Bits, Fix, UFix, Bool.	
`val invertedX = ~x`	Bitwise-NOT
`val hiBits = x & Bits("h_ffff_0000")`	Bitwise-AND
`val flagsOut = flagsIn | overflow`	Bitwise-OR
`val flagsOut = flagsIn ^ toggle`	Bitwise-XOR
Bitwise reductions. Valid on Bits, Fix, and UFix. Returns Bool.	
`val allSet = and(x)`	AND-reduction
`val anySet = or(x)`	OR-reduction
`val parity = xor(x)`	XOR-reduction
Equality comparison. Valid on Bits, Fix, UFix, and Bool. Returns Bool.	
`val equ = x === y`	Equality
`val neq = x != y`	Inequality
Shifts. Valid on Bits, Fix, and UFix.	
`val twoToTheX = Fix(''1'') << x`	Logical left shift.
`val hiBits = x >> 16`	Logical right shift.

Builtin Operators (2)

Bitfield manipulation. Valid on Bits, Fix, UFix, and Bool.	
`val xLSB = x(0)`	Extract single bit, LSB has index 0.
`val xTopNibble = x(15,12)`	Extract bit field from start to end bit position.
`val usDebt = Fill(3, Bits(''hA''))`	Repeats the given bit string multiple times.
`val float = Cat(exponent,mantissa)`	Concatenates bit fields, with first argument on left.
Logical operations. Valid on Bools.	
`val sleep = !busy`	Logical NOT.
`val hit = tagMatch && valid`	Logical AND.
`val stall = src1busy || src2busy`	Logical OR.
`val out = Mux(sel, inTrue, inFalse)`	Two-input mux where sel is a Bool.
Arithmetic operations. Valid on Nums: Fix and UFix.	
`val sum = a + b`	Addition.
`val diff = a - b`	Subtraction AND.
`val prod = a * b`	Multiplication.
`val div = a / b`	Division.
`val mod = a % b`	Modulus.
Arithmetic comparisons. Valid on Nums: Fix and UFix. Returns Bool.	
`val gt = a > b`	Greater than.
`val gte = a >= b`	Greater than or equal.
`val lt = a < b`	Less than.
`val lte = a <= b`	Less than or equal.

Bit-width Inference

- **A nice feature of the Chisel compiler is that it will automatically size the width of wires.**
- The bit-width of ports (of components) and registers must be specified, but otherwise widths are inferred with the application of the following rules:

`z = x + y`	wz = max(wx, wy) + 1
`z = x - y`	wz = max(wx, wy) + 1
`z = x <bitwise-op> y`	wz = max(wx, wy)
`z = Mux(c, x, y)`	wz = max(wx, wy)
`z = w * y`	wz = wx + wy
`z = x << n`	wz = wx + maxNum(n)
`z = x >> n`	wz = wx - minNum(n)
`z = Cat(x, y)`	wz = wx + wy
`z = Fill(n, x)`	wz = wx * maxNum(n)

- Bit-width inference is still under discussion. We might be adding more operators that preserve input widths (truncate to max of the input widths).

End of HDLs/Chisel Introduction part 1

Chisel Tutorial on website later today.

More Introduction in Class Monday:
Components and Circuit Hierarchy
More on Muxes
Registers
Conditional Update Rules
FSMs
Memory Blocks
More on Bundles, Arrays and Bulk Connections