HDLs Hardware Description Languages
as
DSLs Domain Specific Languages

Luca De Santis, June 2021, for Strumenta Community

Introduction

- In this presentation we are going to talk about **hardware description languages** in the perspective of language design, in syntactic and semantic aspects
- We will introduce the context in which HDLs are used and their typical features
- We will also introduce some trends today and a couple of proposals

A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity

Lenny Truong
Stanford University, USA
lenny@cs.stanford.edu

Pat Hanrahan
Stanford University, USA
hanrahan@cs.stanford.edu

SNAPL 2019

What is a Hardware Description Language

A HDL is a (formal) language aimed to represent

- Structure
- Behavior

of (digital) electronic systems, with the purpose of

- Simulate
- Synthesize

Note that:

	resembles
Structure	Class/Objects hierarchy
Behavior	Functions, methods
Simulation	Interpretation
Synthesis	Translation

Levels of description

Level	Tools and Methodology	Languages	Comment
Algorithmic	High-Level Synthesis	C/C++, System C, Matlab	Successful in DSP, inefficient in the general case
ISA (Instruction Set Architecture)	ISA/processor synthesis	DSLs (LISA), System-C	A re-vamping niche, the problem is effective compiling technologies (LLVM is on the run)
RTL (Register Transfer Level)	Logic Synthesis	Verilog/ SystemVerilog, VHDL	Well-established methodology, the problem of «wiring». So successful that is it slowing down further evolution ?
Gate Level Netlist	APR (Automatic Place and Route)	Verilog, VHDL	
Layout of Real Silicon			

Syntactic features

What we have to represent, with examples based on Verilog

Syntactic Feature #1 : Time

- HDLs represent real-world systems: they live in a time-frame
- In general, it's not necessary to represent floating-point values at full accuracy. Typically HDLs use integer values or fixed point values to represent «time units»
- Typically HDLs represent delays, so that it's implicit that «time 0» is the origin of everything

```
`timescale 1ns/1ps

initial
  begin
    #10 <…do something …>
    #40 <…do something …>
```

Happens at 10.000 ns

Happens at 50.000 ns

Syntactic Feature #2 : Bit Level Data Types and Operators

- Bit representation
  - High value → `1`
  - Low value → `0`
  - High-impedance value → `Z`
    - To model three-stateable busses
  - Undefined value → `X`
    - To model un-initialized registers, floating input effects, bus contention
- Operators on busses
  - Concatenation `{a,b,c}`
  - Sub-range Selection `a[3:5]`
- Bit level operators `~&` , `~|` , `~^`
- Arithmetic operators
- Arbitrary-length words
  - `` `12h0F5 ``
- Bit matrix
  - To model storage media (ROMs, RAMs initialization)

Syntactic Feature #3 : Hierarchical Composition

- Modules
- Instances
- Arrayed instances
- Parametric instantiation

```
module M1 (<io ports>) ;
…
endmodule

module M2 (<io ports>) ;
…
endmodule

module M3 (<io ports>) ;
…
M1 U1 (<connections>) ;
M2 U2 (<connections>) ;
…
endmodule
```

Unstructured modules are called «primitives»
Available set of primitives defines the «level of abstraction»

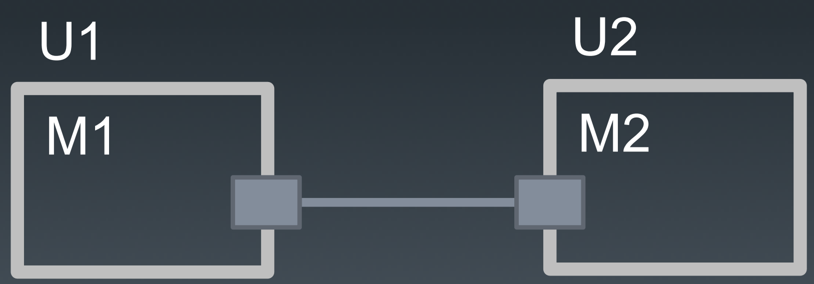
Syntactic Feature #4 : Parametric instantiation

- Useful to build reusable libraries of code
- Very uncomfortable syntax in Verilog
- Ambiguity : uses **`for`** instruction and variables, low readibility
- Useful only on simple arrayed instances

```
// Design for a half-adder
module ha ( input   a, b,
            output  sum, cout);

  assign sum  = a ^ b;
  assign cout = a & b;
endmodule

// A top level design that contains N instances of half adder
module my_design
    #(parameter N=4)
        (   input [N-1:0] a, b,
            output [N-1:0] sum, cout);

    // Declare a temporary loop variable to be used during
    // generation and won't be available during simulation
    genvar i;

    // Generate for loop to instantiate N times
    generate
        for (i = 0; i < N; i = i + 1) begin
          ha u0 (a[i], b[i], sum[i], cout[i]);
        end
    endgenerate
endmodule
```

Syntactic Feature #5 : Connections and Wiring

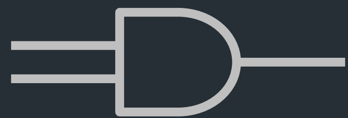
- Modules have «ports» (input, output, in-out)
- Instances inside a module are connected by «wires»
  - map by order: `mymod U1(y,a,b);`
  - map by name: `mymod U1(.out(y),.in1(a),.in2(b));`
- Need of an unambigous semantics :
  - What does it mean that two ports are connected ?



Syntactic Feature #6 : Wires and Continuous Assignment


```
assign y = a & b ;
```

- At any change of a or b, y must be updated
- Different mind-set with respect to general programming
- Models combinational logic



```
assign y = #10  a & b ;
```

- Assignment has effect after 10 time units in the future

Syntactic Feature #7 : States and Storage


```
reg [7:0] x ;
reg [15:0] M [1023:0] ;
```

Definition of states and storages characterizes digital systems more than boolean logic. Sources of ambiguity:

- Variables as in general programming
- Proper state variables (as in FSMs)
- Sampling devices, synchronizers
- Update on edge or on level (d-ff vs. d-latch)
- Shift registers, SIPO, PISO, LFSR
- Different kind of memories
- LIFO, FIFO access

Here is the battle between increasing abstraction and preserving fundamental details

Syntactic Feature #8 : Events and Sensitivity


```
always @ ( a or b )
y = a + b ;
```

```
always @ ( posedge clk )
q = d ;
```

An **event** is «something that happens».
In the digital domain, any change of any signal is an event.
Positive edge 0 → 1 must be distinguished by negative edge 1 → 0

Full sensitivity list → combinational logic
Partial sensitivity list → sequential logic

User defined events

```
→ my_event ;
. . .
@ ( my_event ) <…do something …>
```

Event triggering

Event serving

Syntactic Feature #9 : Behavioral Modeling

- Needed to introduce general programming primitives in the context of hardware modeling
- Very useful in the modeling and verification phase (stimuli abstraction)
- Greatest source of confusion:
  - Generates the false illusion that hardware design is like programming
  - Generates bad abstractions and ambiguous semantics
  - Discrepancies between simulation and synthesis

```
module while_example ();
  integer i=0;
  reg [7:0] r_data[15:0];

  initial begin
    r_data[i] = i*i;
    while (r_data[i] < 100)
        begin
        i = i + 1;
        r_data[i] = i*i;
        #10;
        end
    end
endmodule
```

What «i» and «r_data» are in the physical world ?
In a time-frame, when they are updated ?

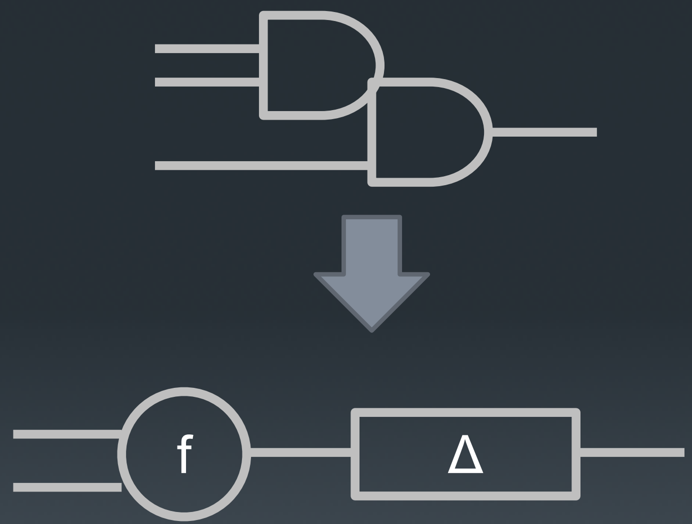
Semantic features

How descriptions map on the physical world

Fundamental semantics of digital systems representations

- delayed functions

```
always @ ( a or b )
y = #(delta) a + b ;
```

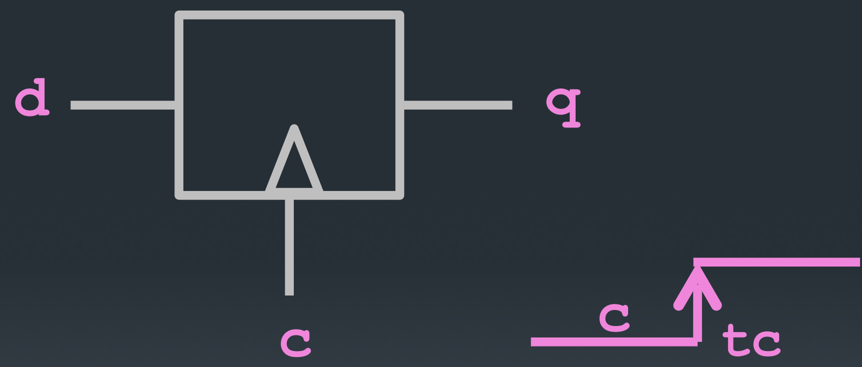


$$y(t+\Delta) = a(t)+b(t)$$

Fits perfectly FP-style

- sampling devices

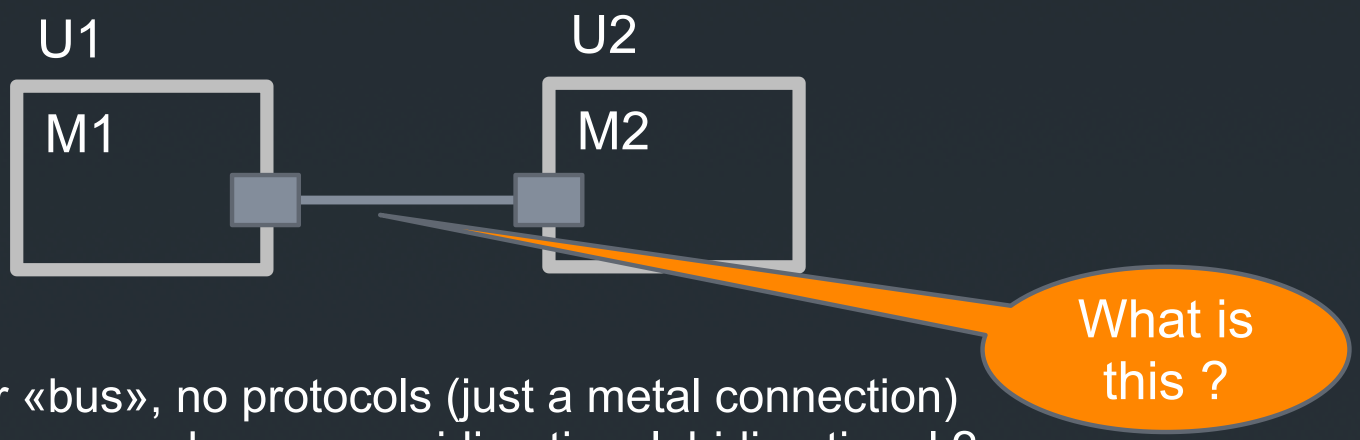
```
always @ ( posedge clk )
q = #(delta) d ;
```



$$q(tc+\Delta) = d(tc)$$

Issues to fit FP-style

Semantics of connections



- Simple «wire» or «bus», no protocols (just a metal connection)
    - Synchronous, asynchronous, unidirectional, bidirectional ?
- Very basic synchronism, adding «data valid» signal
- Hand-shaking (request, acknowledge)
- Synchronous data channel
- Queued message passing (blocking, non-blocking)
- Memory access protocols (arbitrated or not)

Semantics of connections is strictly connected to the description level.

By experience, most of design bugs are at the interface between blocks: a tight control of interconnection semantics is fundamental to design success

Verilog only defines a basic directionality: one input port can be connected to only one output port

Simulation engines

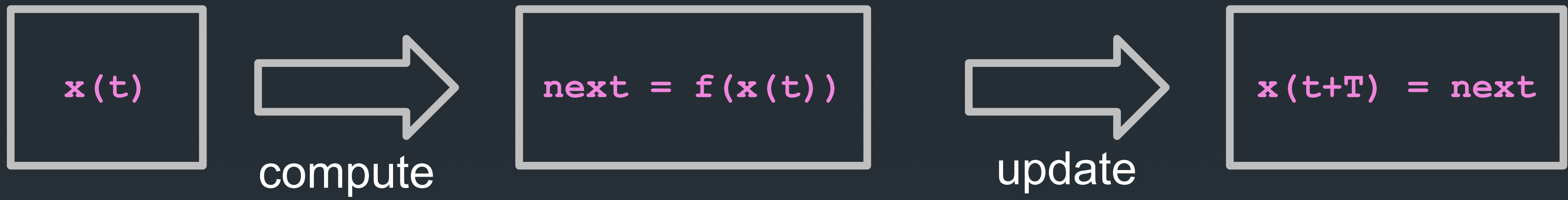
How to check design correctness

Instruction Accurate Simulation

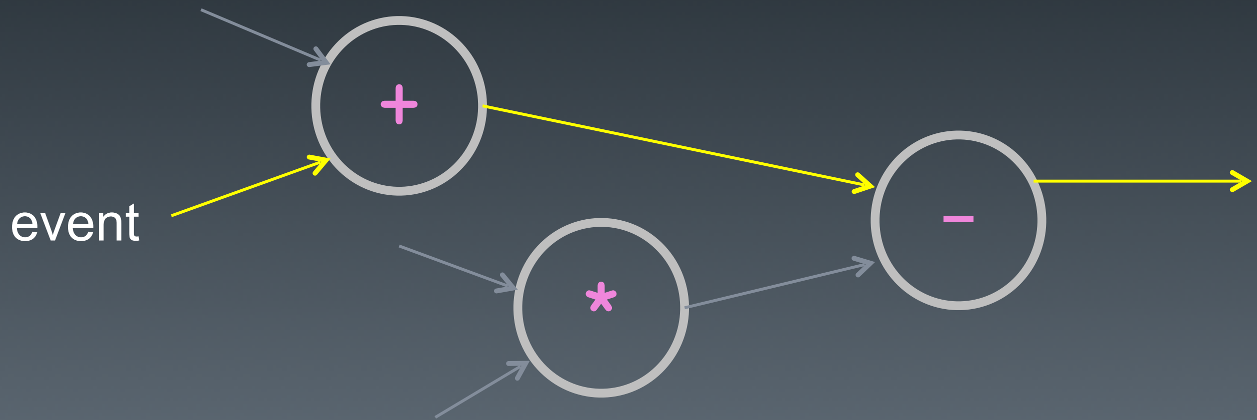
- Virtual Machine Methodology
- Useful to validate the I.S. at an upper level
- Abstracting instruction set details
- If I.S. is the target of design activity, the compiler is the bottleneck : parameterized compilation of SW being executed on the machine
- The problem of resources «boundness»: hardware-aware compilers
- Growing popularity of LLVM/MLIR tool-chain

Clock/Cycle Accurate Simulation

- FSM-like execution
  - If we are not interested to intra-clock-cycle propagation



- Untimed data-flow
  - If we are not interested to accurate timing inside clock cycle

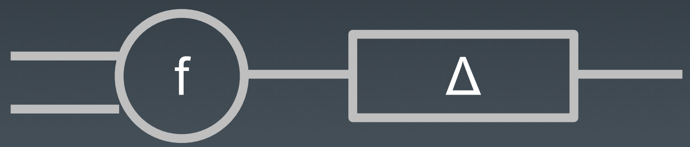


Each block is sensible to input wires and drives output wires
Each wire has a list of connected blocks
Keep a «Touched blocks list»

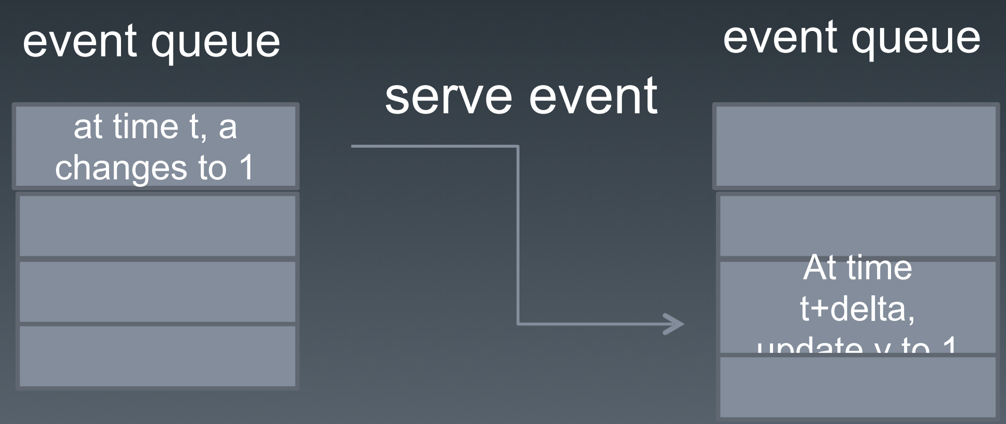
On clock event :
Update storage devices:
Propagate events
While (touchedList is not empty)

Timing Accurate Simulation

- **Timed Data-Flow** model, events cascade
- Discrete event engines based on:
  - Priority Queue (Heap with time tag as the key)
  - Calendar Queue (Priority Queue implemented as a hash-table)
  - Time Wheel (Circular buffer of events-list with a pointer advancing at each time step)
  - Local Counters (Each module has its own time counter initialized when an event arrives, counted down at any unit time step)



$$y(t+\Delta) = a(t)+b(t)$$



A few words about synthesis

- **High-Level Synthesis**

Starting from an algorithm code, allocates hardware resources according to area/timing constraints; very successful on implementing DSP algorithms, not so effective on control-dominated architectures

- **Logic Synthesis**

Very well established tool, based on boolean function reduction and state machines optimization. Unexpected results if designer is not disciplined (linting tools).

- **Instruction Set synthesis**

Only apparent abstraction. Based on predefined pipelined architectures, customized on designer input parameters.

- **HW/SW co-synthesis**

No well established tools. Based on heuristics and designers experience.

- **Place & Route Algorithms**

Complex topic, lot of heuristics, based on search for an optimal placement plus simulated annealing to wire blocks in an optimal way. Predictable results and optimal metrics are the big challenges.

Some issues

Some issues

- Great success of digital systems in last thirty years is mostly due to the implementation of the **synchronous model** : separate logic functionality from timing and verify both in parallel (similar to HW/SW partitioning in processors)
- The most successful HDL (Verilog) doesn't have any syntactic or semantic way to define explicitely clock domains : this is done at an upper level, typically in the synthesis tool environment
- Clock is like any other wire : is this freedom ? No, it just increases the probability of introducing bugs

- The freedom of «wiring» allowed by Verilog is strictly connected to the «goto» problem. Software engineers solved this problem long time ago by structured programming, hardware engineers don't do it yet. No «structured hardware design» on the shelf.

Some issues

Again...

- The most successfull theoretical model in HW design is the FSM (Finite State Machine): Verilog has no way to define a FSM in a formally structured way.
- The freedom of defining a FSM in a no formal way just increases the probability of introducing bugs.

Again…

- Verilog has no semantics of interconnections other than basic in/out declarations. High-level synthesis tools have, but they skip many critical low-level details

- Exponential increase of complexity of digital design, increase of parallelism, necessity of predictable metrics are pushing Verilog to its limits.
- This is why there is a new trend in HDL design : Software engineering methods applied to hardware design to raise the abstraction level and avoid «wild wiring»

Alternative tools aimed at raising the abstraction level (just a sample)

- **Chisel** : based on Scala, adopts test-driven design model, hard learning curve for an expert hardware engineer with none or little «software engineering mindset»
- **Cλash** : pure functional approach, same problem of the engineer's mindset
- **TV-Verilog** : easy syntax to define pipelining schemes, very promising tool. This is an example of how well-defined syntax features associated to a well-defined semantics, makes things easier and more productive for the user

A proposal : MyProcessor

- Philosophy : Processor as a tool, not only as a component
- Searching for the optimal PPA trade-off
- Limited use of «wire» concept through FP-like syntax
- Avoid micro-details of instruction set through IS abstraction
- Standard storage primitives and «behavioral tasks»
- Easy parameterization and arrayed instantiation
- Explicit clock and reset domains
- Explicit clock scheme
- Not only for processors but useful for a generic design

- Status:
  - Proof of concept in Python
  - Working on ANTLR+Kotlin flow as described by Federico Tomassetti

Some Details

Clock scheme defined at HDL level


```
domain clk_reg :
        reg R1 [32]
        reg A1 [8]

domain clk_mem :
        mem M1 [ 32, 1024 ]


R1 <= M1.read(A1)
```

Memory access task
This syntax is enough for the compiler to understand how to connect things: no use of «wires»

This substitutes dozens code lines without loosing details on clock behavior.

Trends and conclusion

Trends Today

- Raise the level of hardware **abstraction** without loosing critical details, possibly not towards strictly «software mindset» solutions. The problem is that hardware engineers and software engineers have a different mind-set; Typically hardware engineers don't know what is a **monad**, while software engineers don't know what is **asynchronous metastability** ☺ (joking!)
- Dealing with massive heterogeneous **parallelism**
- Tight control of PPA **metrics**
- Formal **correctness**: verification cost exploded, observability of state space, correct-by-design methodology
- **Democratization** of design: open source EDA, digital design as a cloud service
- HW-SW **co-design** and **co-synthesis** are still open topics, no well-established methodologies
- The problem of «parametric» and **hardware-aware compilers** (LLVM, MLIR…)
- **Neuromorphic** processing, **non-VonNeumann** architectures: ML/AI on synthesized HW
- **Projectional editing** applied to HW design : a possibility ?

Conclusion

- Big opportunities for software engineers to help hardware designers in embracing a growing complexity
- A famous Dijkstra's citation enlightened me

  *"The purpose of **abstraction** is not to be vague, but to create a new semantic level in which one can be absolutely precise"*
- Reinforce the concept of what is the «semantics» of digital systems representations at any level of description

"This is an exciting time to be a researcher interested in PL and hardware"

Lenny Truong, Pat Hanrahan "A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity" SNAPL 2019

Thanks for your attention!

ldesantis@ymail.com