Hardware description, a language perspective

dram

2021-11-20

碎碎念

“dram 是学业繁忙的大三学生”

– “所以本周的预告呢？ ”

“dram 是学业繁忙的大三学生”

– “所以本周的预告呢？”

Pwe 会长
周三或周四 1 12:08 PM
dram 是学业繁忙的大三学生 12:08 PM

Figure 1: 大家以为 dram 在做的事情

“dram 是学业繁忙的大三学生”

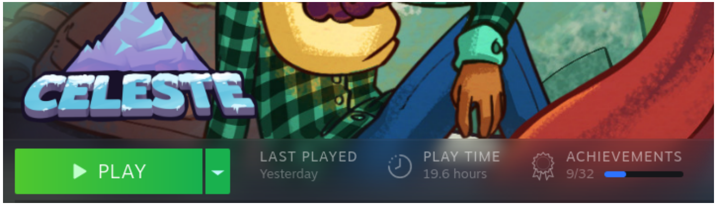
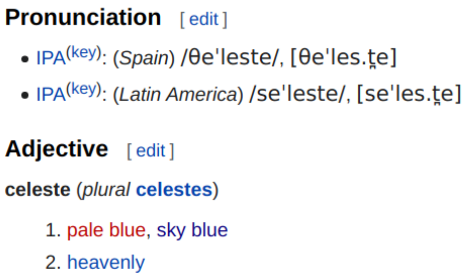


Figure 2: dram 实际上在做的事情

‘Celeste’


Pronunciation [edit]

- IPA(key): (*Spain*) /θeˈleste/, [θeˈles.t̪e]
- IPA(key): (*Latin America*) /seˈleste/, [seˈles.t̪e]

Adjective [edit]

celeste (*plural* **celestes**)

1. pale blue, sky blue
2. heavenly

Figure 3: ‘Celeste’ in Spanish

Celeste

– 天蓝色?

Celeste

– 天蓝色?

  无端联想

– Sequencer 又想让我讲 Bluespec 了
– 然后给喵喵托梦，让喵喵给我疯狂暗示
– 喵喵给我安利爬山模拟器

Celeste

– 天国?

Celeste

- 天国?
- Sequencer：科学的数字电路设计方法学
- 来自天国的 HDL

今天的内容

- 由于疫情原因，Sequencer 今天没有到场
- 我可以讲 Sequencer 听不懂的东西了

今天的内容

- 由于疫情原因，Sequencer 今天没有到场
- 我可以讲 Sequencer 听不懂的东西了

- Hardware description language
- Sequencer 讲过 Hardware 了
- 我来讲 Language

Language

Lies and abstractions

- Computer science is all about lies
- A good lie is called an *abstraction*

Lies and abstractions

- Xilinx University Program
- 74LSxx on FPGA[1]

[1] `https://github.com/xupgit/Basys3`

Lies and abstractions

- Xilinx University Program
- 74LSxx on FPGA[1]

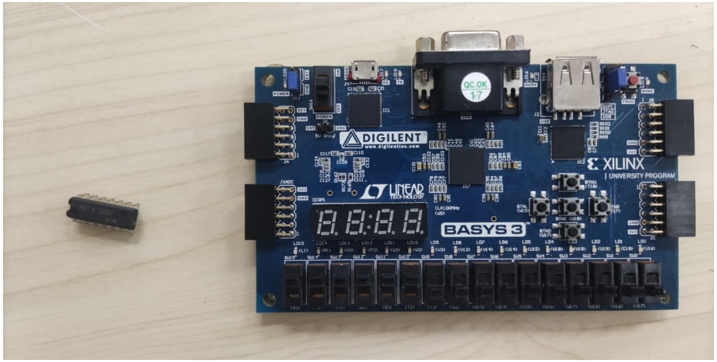


Figure 4: 74LS86 and Basys 3

[1]https://github.com/xupgit/Basys3

Preprocessor and language

- 74-series 和 LUT
  - 在画 “原理图”[2]的时候，我们：
  - 看到的是 74-series 的框
  - 可以按照 74-series 的逻辑分析
  - 那它就是 74-series 的一个实现

[2]指 Block Design

Preprocessor and language

- 74-series 和 LUT
  - 在画 “原理图”[2]的时候，我们：
  - 看到的是 74-series 的框
  - 可以按照 74-series 的逻辑分析
  - 那它就是 74-series 的一个实现
- 我们一般不会认为 C 语言是一个 macro assembler
  - C17: ISO/IEC 9899:2018
  - 作为一个语言，它有其单独定义的语义
  - 我们按照 C 语言的语义写的程序，可以正确在 CPU（with 操作系统）上运行

[2]指 Block Design

Turtles all the way down

- Electronics
- Transistors
- Gates
- Clocked waveforms
- Machine binary
- Assembler
- ...

Turtles all the way down

- Electronics
- Transistors
- Gates
- Clocked waveforms
- Machine binary
- Assembler
- ...

- 限制我们考虑的底层实现的细节
- 获得一种更接近我们最终要解决的问题的思路

Turtles all the way down

- Electronics
- Transistors
- Gates
- Clocked waveforms
- Machine binary
- Assembler
- ...

- 限制我们考虑的底层实现的细节
- 获得一种更接近我们最终要解决的问题的思路

- 设计一个更高层次的语言，然后给它定义一个不涉及我们不关心的细节的语义

Clash

组合逻辑

- 一段组合逻辑是一个输入到输出的函数

$$f : P \to Q$$

- 对于在同步时序逻辑中的一个组合逻辑电路来说，它的输出完全只和输入有关

Functional programming

– Combinatorial logic modules *are* pure functions

Functional programming

– Combinatorial logic modules *are* pure functions

```
fullAdder :: Bit -> Bit -> Bit -> (Bit, Bit)
fullAdder a b ci = (co, s)
  where
    ab = a `xor` b
    s = ab `xor` ci
    co = (a .&. b) .|. (ab .&. ci)
```

Functional programming

- Combinatorial logic modules *are* pure functions

```
fullAdder :: Bit -> Bit -> Bit -> (Bit, Bit)
fullAdder a b ci = (co, s)
  where
    ab = a `xor` b
    s = ab `xor` ci
    co = (a .&. b) .|. (ab .&. ci)
```

```
module full_adder(
  input a, input b, input ci,
  output co, output s
);

// ...
```

Clash

- An HDL that happens to be a subset of Haskell
- `https://clash-lang.org/`

Clash

- An HDL that happens to be a subset of Haskell
- `https://clash-lang.org/`

Comparison with Chisel

- Chisel *runs* Scala code to generate Verilog. Clash *compiles* Haskell source to Verilog.
  - No Haskell code is really ‘run’ in the synthesis flow.
  - The Haskell code is *not* a hardware generator. It *is* the hardware.

Functional programming

- 一个函数式程序确实是一个结构化描述
- 拍平成文本的图

```haskell
fullAdder :: Bit -> Bit -> Bit -> (Bit, Bit)
fullAdder a b ci = (co, s)
  where
    ab = a `xor` b
    s = ab `xor` ci
    co = (a .&. b) .|. (ab .&. ci)
```

Clash

- Haskell 的 “可综合子集”
- 与 Haskell 相交的部分语义与 Haskell 相同

```
ghci> fullAdder 1 1 0
(1,0)
ghci> fullAdder 1 0 0
(0,1)
ghci> fullAdder 1 1 1
(1,1)
```

Clash

- Haskell 的 “可综合子集”
- 与 Haskell 相交的部分语义与 Haskell 相同

```
ghci> fullAdder 1 1 0
(1,0)
ghci> fullAdder 1 0 0
(0,1)
ghci> fullAdder 1 1 1
(1,1)
```

Haskell-synthesis mismatch?

- 结构化描述的翻译很直接
- 不是所有 Haskell 结构都能翻译成硬件
  - 比如 inline 不掉的 higher-order function
- (Clash 现状：编译会静态地知道发生了问题，但是没有办法给出一个人类可读的报错)

时序逻辑

- $[A]_d$: Discrete time-step signal, synchronized to clock domain d
- $f : [A]_d \to [B]_d$
- `f :: _ => Signal dom A -> Signal dom B`

时序逻辑

- `map` : $(A \to B) \to ([A]_d \to [B]_d)$
- `delay` : $A \to [A]_d \to [A]_d$
  - Implemented as D flip-flops
  - (First param is initial value)
- `liftA2` : $(A \to B \to C) \to ([A]_d \to [B]_d \to [C]_d)$
  - ($[-]_d$ is applicative)

Mealy machine

```haskell
debounceEvent :: forall dom ctr. _
    => ctr -> Signal dom Bool -> Signal dom Bool
debounceEvent ctrMax inp = snd <$> next
    where
        state :: Signal dom ctr
        state = delay 0 (fst <$> next)

        next :: Signal dom (ctr, Bool)
        next = liftA2 go state inp

        go :: ctr -> Bool -> (ctr, Bool)
        go 0 False = (0, False)
        go 0 True = (ctrMax, True)
        go ctr False = (ctr - 1, False)
        go _ True = (ctrMax, False)
```

Signals work like lists

```
map :: (a -> b) -> [a] -> [b]

map (\x -> x + 1) [2, 0, 2, 1, ...] = [3, 1, 3, 2, ...]

delay :: a -> [a] -> [a]

delay 0 [1, 1, 2, 0, ...] = [0, 1, 1, 2, ...]
```

Signals work like lists

```
map :: (a -> b) -> [a] -> [b]

map (\x -> x + 1) [2, 0, 2, 1, ...] = [3, 1, 3, 2, ...]

delay :: a -> [a] -> [a]

delay 0 [1, 1, 2, 0, ...] = [0, 1, 1, 2, ...]
```

– Denotational semantics: mental model for designing

What we have so far

- `a`: Signal without clock domain
- `Signal dom a`: Signal with specified clock domain

What we have so far

- `a`: Signal without clock domain
- `Signal dom a`: Signal with specified clock domain

- `a -> b`: Unclocked module
- `Signal dom a -> Signal dom b`: Clocked module

Fun with Types

Case study: Delayed signals

- DSP pipeline
- Computational geometry: pipeline registers
- Block-RAM access: 1-cycle or 2-cycle delay

Example: Rendering

- Vintage-style graphics processor
  - Tiles
  - Sprites

Example: Rendering

- Vintage-style graphics processor
  - Tiles
  - Sprites
- Coordinate (x, y)
- Tile rendering:
  - Get tile ID from RAM
  - Get tile pixel data from RAM
- Match (x, y) with sprite
  - Get sprite pixel data from RAM
- Blend

Pipelining delay

- Tracking delays in pipelines
  - dsptools[3]
  - `DSignal`[4]
- $[A]_d^{(n)}$
- `DSignal (dom :: Domain) (delay :: Nat) a`

[3] https://github.com/ucb-bar/dsptools

[4] https://hackage.haskell.org/package/clash-prelude-1.4.6/docs/Clash-Signal-Delayed.html

Delayed signal primitives

- $\texttt{map} : (A \to B) \to ([A]_d^{(n)} \to [B]_d^{(n)})$
- $\texttt{delay} : A \to [A]_d^{(n)} \to [A]_d^{(n+1)}$
- $\texttt{ram} : [\texttt{Addr}]_d^{(n)} \to [\texttt{Data}]_d^{(n+2)}$

Delayed signal primitives

- $\texttt{map} : (A \to B) \to ([A]_d^{(n)} \to [B]_d^{(n)})$
- $\texttt{delay} : A \to [A]_d^{(n)} \to [A]_d^{(n+1)}$
- $\texttt{ram} : [\texttt{Addr}]_d^{(n)} \to [\texttt{Data}]_d^{(n+2)}$
- $\texttt{del} : A \to [A]_d^{(n)} \to [A]_d^{(n+m)}$
- $\texttt{feedback} : [A]_d^{(n+m)} \to [A]_d^{(n)}$

Quaternion rotation

$$q = x\mathbf{i} + y\mathbf{j} + z\mathbf{k} + s$$
$$\mathsf{qv} = (x, y, z)$$

Quaternion rotation

$$q = x\mathbf{i} + y\mathbf{j} + z\mathbf{k} + s$$
$$\mathtt{qv} = (x, y, z)$$

```
qv :: DSignal dom 0 Vec3
qs :: DSignal dom 0 Scalar
v :: DSignal dom 0 Vec3

tmp :: DSignal dom 3 Vec3
tmp = 2 * (qv `cross` v)

result :: DSignal dom 6 Vec3
result = (del qv `cross` tmp) + del (del s `sv` tmp) + del v
```

Computation with delay information

- `cross` : $[V]_d^{(n)} \to [V]_d^{(n)} \to [V]_d^{(n+3)}$
- `sv` : $[s]_d^{(n)} \to [V]_d^{(n)} \to [V]_d^{(n+2)}$
- $+$: $[V]_d^{(n)} \to [V]_d^{(n)} \to [V]_d^{(n)}$

Type inference

– Type inference → delay inference

```
       ...
6     (del qv `cross` tmp) + del (del s `sv` tmp) + del v
6      del qv `cross` tmp
3      del qv
0          qv
```

– Inferred:

```
del :: DSignal dom 0 Vec3 -> DSignal dom 3 Vec3
```

Type inference

- Type inference → delay inference

```
tmp :: DSignal dom 3 Vec3

                                     ...
6     (del qv `cross` tmp) + del (del s `sv` tmp) + del v
6                            del (del s `sv` tmp)
6 - m                             del s `sv` tmp
6 - m - 2 = 3                               tmp
    => m = 1
```

- Inferred:

```
del :: DSignal dom 5 Vec3 -> DSignal dom 6 Vec3
```

Implementation

- Based on helper functions, no need for special compiler support

```haskell
newtype DSignal (dom :: Domain) (delay :: Nat) a =
    DSignal { toSignal :: Signal dom a }
```

Haskell type system is not omnipotent

– Synthesizability?

Haskell type system is not omnipotent

- Synthesizability?

  $\texttt{delay} : A \to [A]_d^{(n)} \to [A]_d^{(n+1)}$

- Register initial value must be static constant
- Unchecked by Haskell type system
- Caught by Clash compiler, with horrible error messages
  - Synthesis is done on IR (*Core*), post-optimization

Haskell type system is not omnipotent

- Synthesizability?

  $\texttt{map} : (A \to B) \to ([A]_d^{(n)} \to [B]_d^{(n)})$

- `map` is not synthesizable
- If `f` is synthesizable, `map f` is synthesizable
- Complicated relationships between circuit-generating helpers and circuits

Haskell type system is not omnipotent

- Synthesizability?

  $\texttt{map} : (A \to B) \to ([A]_d^{(n)} \to [B]_d^{(n)})$

- `map` is not synthesizable
- If `f` is synthesizable, `map f` is synthesizable
- Complicated relationships between circuit-generating helpers and circuits

- Differentiate between synthesizable/generator/non-synthesizable code
- Can we modify the type system for it?

Verilog is a mess

```
// Generator code
integer i;

// Synthesizable code
logic [31:0] foo [SIZE - 1 : 0];
logic [31:0] bar [SIZE - 1 : 0];
logic [31:0] baz [SIZE - 1 : 0];

always_comb begin
    // Generator code
    for (i = 0; i < SIZE; i ++) begin

        // Synthesizable code
        foo[i] = bar[i] + baz[i];
    end
```

Staging

HDL semantics

- “Verilog 项目都是仿真驱动开发的。” —— zyx

HDL semantics

- “Verilog 项目都是仿真驱动开发的。” —— zyx
- Modeling semantics
  - Verilog: Simulation semantics
  - Bluespec: One rule at a time
  - Clash: Normal Haskell semantics
- Circuit semantics
  - Synchronous sequential logic
  - Send to digital backend engineer

HDL semantics

- “Verilog 项目都是仿真驱动开发的。” —— zyx
- Modeling semantics
  - Verilog: Simulation semantics
  - Bluespec: One rule at a time
  - Clash: Normal Haskell semantics
- Circuit semantics
  - Synchronous sequential logic
  - Send to digital backend engineer
- RTL HDL 编程和软件开发的本质区别
  - 软件开发人员有硬件用，硬件开发人员没有硬件用

HDL semantics

$$\begin{tikzcd}
\text{Source code} \arrow[r, "\text{Synthesis}"] \arrow[d, "\text{Simulator}", "\text{and Humans}"'] & \text{Synthesis result} \arrow[d, "\text{Backend}"] \\
\text{Modeling semantics} & \text{Circuit semantics}
\end{tikzcd}$$

Figure 5: The two semantic domains of HDL

HDL semantics

$$\begin{tikzcd}
\text{Source code} \arrow[r, "\text{Synthesis}"] \arrow[d, "\text{Simulator}"', "\text{and Humans}"] & \text{Synthesis result} \arrow[d, "\text{Backend}"] \\
\text{Modeling semantics} \arrow[r, dash, dashed, "\text{Match?}"] & \text{Circuit semantics}
\end{tikzcd}$$

Figure 5: The two semantic domains of HDL

Domain mismatch

‘Synthesizable subset’

- Verilog: Event-based behavioral simulation
- Clash: Structural description, reuses Haskell semantics
  - Static checks are limited

Domain mismatch

‘Synthesizable subset’

- Verilog: Event-based behavioral simulation
- Clash: Structural description, reuses Haskell semantics
    - Static checks are limited
- Generator looks the same as synthesizable!

Solution 1: Separate ‘generator’ stage

- Chisel: eDSL, completely different language
  - Run Scala code, spits out verilog
- Needs separate versions of all data types
  - One for Scala, one for synthesis

Solution 1.1: Single language, multiple contexts

- Bluespec: Monad

```
module mkRamController(RamController);
    // In module context
endmodule
```

Solution 1.1: Single language, multiple contexts

- Bluespec: Monad

```
module mkRamController(RamController);
    // In module context
endmodule
```

- `mkRamController :: module#(RamController)`

Solution 1.1: Single language, multiple contexts

```
module FooModule(Foo);
    // Control in module context, generator code
    for (Integer i = 0; i < n; i = i + 1) begin

        // Run mkRamController, calls other generator code
        RamController ctrl <- mkRamController
        // ...
    end

    rule foo;
        // In rule context, synthesizable
        if (b > 3)
            a <= b + 2;
    end
```

Solution 2: Staging with static types

- `delay` : $A \to [A] \to [A]$
- In `reg = delay init next`
    - `init` is a ‘generator’ constant
    - `next` is a synthesizable signal input
    - `reg` is a synthesizable signal output

Solution 2: Staging with static types

- `delay` $: A \to [A] \to [A]$
- In `reg = delay init next`
  - `init` is a ‘generator’ constant
  - `next` is a synthesizable signal input
  - `reg` is a synthesizable signal output
- `delay` $: A_{\texttt{gen}} \to_{\texttt{gen}} ([A]_{\texttt{syn}} \to [A]_{\texttt{syn}})_{\texttt{gen}}$

Labelled primitives

- $\texttt{map} : (A_{\mathsf{syn}} \to B_{\mathsf{syn}})_{\mathsf{gen}} \to_{\mathsf{gen}} ([A]_{\mathsf{syn}} \to [B]_{\mathsf{syn}})_{\mathsf{gen}}$
- $\texttt{delay} : A_{\mathsf{gen}} \to_{\mathsf{gen}} ([A]_{\mathsf{syn}} \to [A]_{\mathsf{syn}})_{\mathsf{gen}}$
- $\texttt{liftA2} : (A_{\mathsf{syn}} \to B_{\mathsf{syn}} \to C_{\mathsf{syn}})_{\mathsf{gen}} \to_{\mathsf{gen}} ([A]_{\mathsf{syn}} \to [B]_{\mathsf{syn}} \to [C]_{\mathsf{syn}})_{\mathsf{gen}}$

Synthesizability

- Synthesizable modules

$$([A]_{\mathsf{syn}} \to [B]_{\mathsf{syn}} \to \ldots \to [R]_{\mathsf{syn}})_{\mathsf{gen}}$$

- Not synthesizable: $\mathtt{delay} : \mathsf{u32}_{\mathsf{gen}} \to_{\mathsf{gen}} ([\mathsf{u32}]_{\mathsf{syn}} \to [\mathsf{u32}]_{\mathsf{syn}})_{\mathsf{gen}}$
- Synthesizable: $\mathtt{delay}\ 0 : ([\mathsf{u32}]_{\mathsf{syn}} \to [\mathsf{u32}]_{\mathsf{syn}})_{\mathsf{gen}}$

Polymorphism

- Non-synthesizable: $\texttt{mux} : \forall_{\textsf{gen}} A.(\texttt{Bool}_{\textsf{syn}} \to A_{\textsf{syn}} \to A_{\textsf{syn}} \to A_{\textsf{syn}})_{\textsf{gen}}$
- Non-synthesizable: $\texttt{mux}[\texttt{u32}] : (\texttt{Bool}_{\textsf{syn}} \to \texttt{u32}_{\textsf{syn}} \to \texttt{u32}_{\textsf{syn}} \to \texttt{u32}_{\textsf{syn}})_{\textsf{gen}}$

Polymorphism

- Non-synthesizable: $\texttt{mux} : \forall_{\textsf{gen}} A.(\texttt{Bool}_{\textsf{syn}} \rightarrow A_{\textsf{syn}} \rightarrow A_{\textsf{syn}} \rightarrow A_{\textsf{syn}})_{\textsf{gen}}$
- Non-synthesizable: $\texttt{mux}[\texttt{u32}] : (\texttt{Bool}_{\textsf{syn}} \rightarrow \texttt{u32}_{\textsf{syn}} \rightarrow \texttt{u32}_{\textsf{syn}} \rightarrow \texttt{u32}_{\textsf{syn}})_{\textsf{gen}}$

- Generic/parameterized circuits still have static checks

Synthesizability

– Well-typed programs don’t get stuck

Synthesizability

- Well-typed programs don’t get stuck
- Well-typed hardware models don’t get stuck
- Well-typed, synthesizably-typed hardware generators can emit netlists

Is Verilog salvagable?

– Labelled, staged behavioral modeling

```
// This is NOT REAL VERILOG CODE

// Force mark as generator-only
genvar integer i;

// Synthesizable code
logic [31:0] foo [SIZE - 1 : 0];
logic [31:0] bar [SIZE - 1 : 0];
logic [31:0] baz [SIZE - 1 : 0];
```

Is Verilog salvagable?

```
always_comb begin
    // Generator only
    generate for (i = 0; i < SIZE; i ++) begin
        foo[i] = bar[i] + baz[i];

        // Not permitted: cannot access signal from generator
        // i = foo[i];
    end
end
```

Conclusion

Conclusion

- Abstractions are good
- Type theory is hard
- We need better HDL

Conclusion

- Abstractions are good
- Type theory is hard
- We need better HDL

- Can we have an HDL with an expressive enough type system for both modelling and synthesis?

Thank you