Magma

Pat Hanrahan

CS448H: Agile Hardware Design
Winter 2017

Verilog

Combinational logic: and4.v, andn4.v

Sequential logic: dff.v, tff.v

Procedural assignment: dffr.v

Combinational logic: always_comb1.v

Latch inference: latch.v, inferred.v

Blocking: qqblocking.v

Nonblocking: qqnonblocking.v

Combinational always: always_comb2.v

Race conditions: race*.v

Operator Type	Operator Symbol	Operation Performed	Number of Operands
Arithmetic	*	Multiply	2
	/	Divide	2
	+	Add	2
	-	Subtract	2
	%	Modulus	2
Logical	!	Logical negation	1
	&&	Logical and	2
	||	Logical or	2
Rational	>	Greater than	2
	<	Less than	2
	>=	Greater than or equal	2
	<=	Less than or equal	2
Equality	==	Equality	2
	!=	Inequality	2
	===	Case equality	2
	!==	Case inequality	2
Bitwise	~	Bitwise negation	1
	&	Bitwise and	2
	|	Bitwise or	2
	^	Bitwise xor	2
	^~ or ~^	Bitwise xnor	2
Reduction	&	Reduction and	1
	~&	Reduction nand	1
	|	Reduction or	1
	~|	Reduction nor	1
	^	Reduction xor	1
	^~ or ~^	Reduction xnor	1
Shift	>>	Right shift	2
	<<	Left shift	2
Concatenation	{ }	Concatenation	any
Replication	{ { } }	Replication	any
Conditional	? :	Conditional	3

Verilog

1. **Variables in `always` blocks must always be declared as `reg`, but may or may not be a `reg`.**
   - `always @(*)` **is combinational, i.e. lhs is not a `reg`, but beware of `reg` inference(!). Use blocking assignment (=)**
   - `always @(posedge clk)` **is sequential, i.e. lhs is a `reg`. Use nonblocing assignments (<=)**
2. **"Procedural assignment" inside an `always` block looks like C, but is not sequentially executed like C, since it runs in parallel.**
3. **Allows multiple assignments to a `wire` or `reg`**

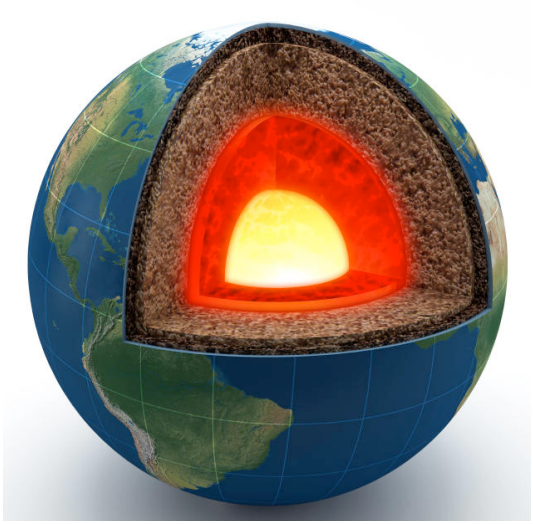
Verilog

1. **Designed for simulation. "testbenches" used to test code.**
2. **These parts do not generate hardware. E.g. `for` and `while` statements**
3. **Easy to write programs that can't be synthesized (because sequential code can't be synthesized)**
4. **Does not use modern PL concepts**
   - **Poor type system**
   - **Poor support for module and abstraction**
   - **Poor support for meta-programming**
   - **...**
5. **Yet another programming language ...**

Meta-programming

```
wire [`N-1:0] fasum;
wire [`N-1:0] facout;

FA fa(J1[0], J3[0], 1'b0, fasum[0],
facout[0]);

genvar i;
generate
for (i = 1; i < `N ; i = i + 1) begin:add
    FA fa(J1[i], J3[i], facout[i-1],
fasum[i], facout[i]);
end

assign D5 = facout[`N-1];
```

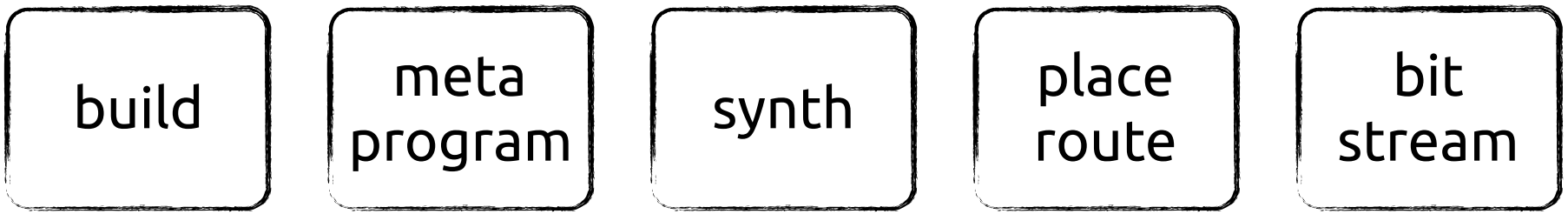
```
wire [`N-1:0] fasum;
wire [`N-1:0] facout;

FA fa0(J1[0], J3[0], 1'b0,
         fasum[0], facout[0]);

% for i in range(1,N):
FA fa${i}(J1[${i}], J3[${i}], facout[${i-1}],
        fasum[${i}], facout[${i}]);
% endfor

assign D5 = facout[`N-1];
```



Magma

Python meta-programs Circuits

Multi-Stage Meta-Programming



`make`	`cpp`	`map`	`par`	`bitgen`
`bake`	`magma`	`yosis`	`arachne`	`icepack`

Combinational Logic

Circuit Abstraction

1. **Instantiate circuits**

```
lut4 = LUT4(I0&I1)
ff = DFF()
```

Circuits are "like" functions

Circuit Abstraction

2. Wire circuits to other circuits

Explicitly

```
wire(I, ff.I)
wire(ff.O, O)
```

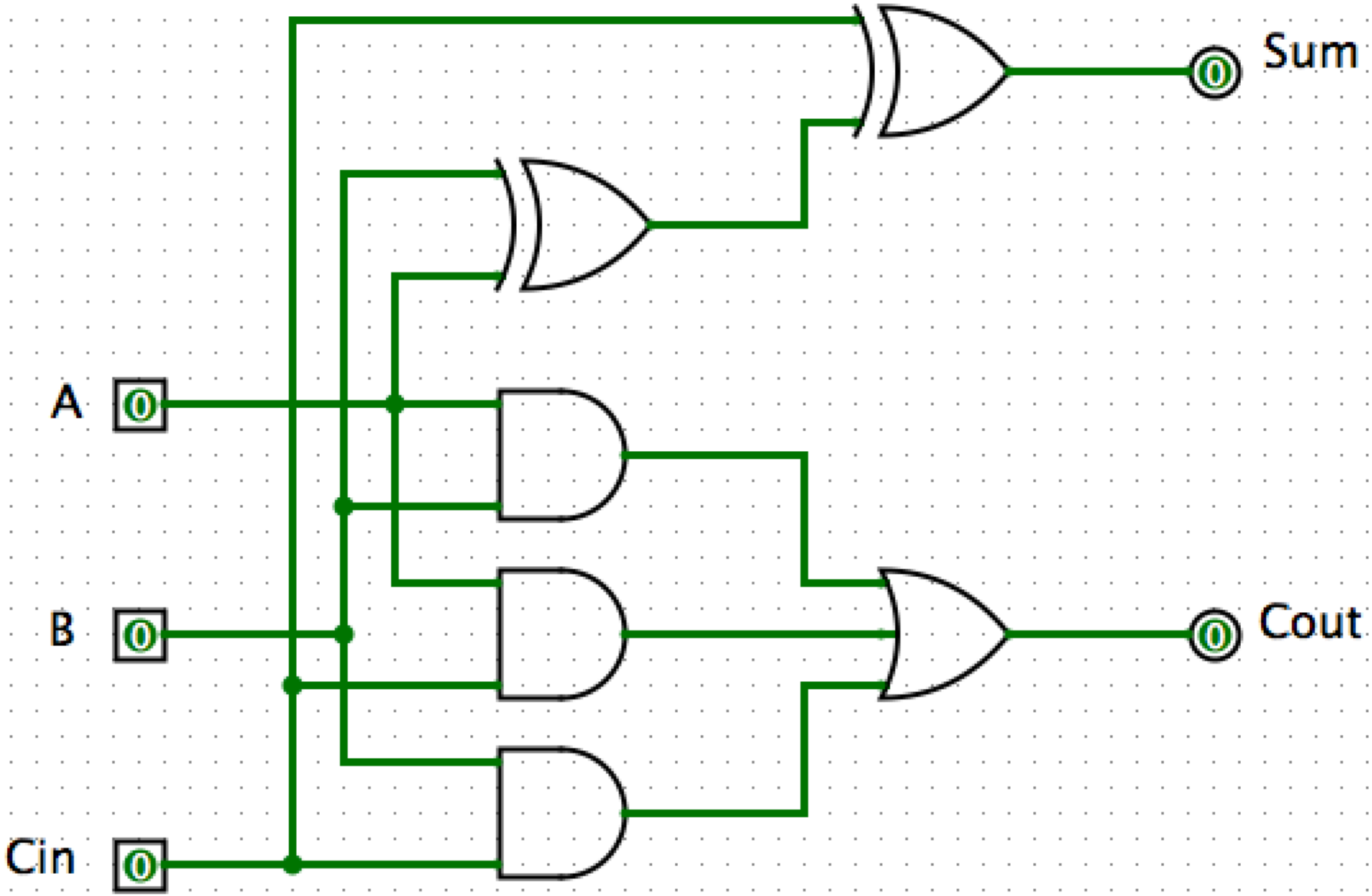
Using function calls

```
ff(lut(I))
```

NB. Circuits can only be "called" once

```
// And4

lut4 = LUT4(I0&I1&I2&I3)

lut4(main.J1[0],
     main.J1[1],
     main.J1[2],
     main.J1[3])

wire( lut4, main.D5 )
```

Full Adder



fulladder.py

```
main = icestick.main()

def and2(a, b):
    return And2()(a,b)

def or3(a,b,c):
    return Or3()(a,b,c)

def xor3(a,b,c):
    return Xor3()(a,b,c)

def FullAdder(a,b,c):
    S = xor3(a,b,c)
    C = or3(and2(a,b),and2(b,c),and2(c,a))
    return S, C

S, C = FullAdder(main.J1[0],main.J1[1],main.J1[2])

wire( S, main.D1 )
wire( C, main.D2 )
```

```
// fulladder.v
module main (input [2:0] J1, output  D2, output  D1);
wire  inst0_O;
wire  inst1_O;
wire  inst2_O;
wire  inst3_O;
wire  inst4_O;
SB_LUT4 #(.LUT_INIT(16'h9696)) inst0
(.I0(J1[0]), .I1(J1[1]), .I2(J1[2]), .I3(1'b0), .O(inst0_O));
SB_LUT4 #(.LUT_INIT(16'h8888)) inst1
(.I0(J1[0]), .I1(J1[1]), .I2(1'b0), .I3(1'b0), .O(inst1_O));
SB_LUT4 #(.LUT_INIT(16'h8888)) inst2
(.I0(J1[1]), .I1(J1[2]), .I2(1'b0), .I3(1'b0), .O(inst2_O));
SB_LUT4 #(.LUT_INIT(16'h8888)) inst3
(.I0(J1[2]), .I1(J1[0]), .I2(1'b0), .I3(1'b0), .O(inst3_O));
SB_LUT4 #(.LUT_INIT(16'hFEFE)) inst4
(.I0(inst1_O), .I1(inst2_O), .I2(inst3_O), .I3(1'b0), .O(inst4
_O));
assign D2 = inst4_O;
assign D1 = inst0_O;
endmodule
```

Magma Product Types

```
T = Bit | Array(n,T) | Tuple(T1, T2, …, Tn)
```

- **Recursive product type (not algebraic data type)**
- **All types have fixed size**

```
Array(n,T)
```

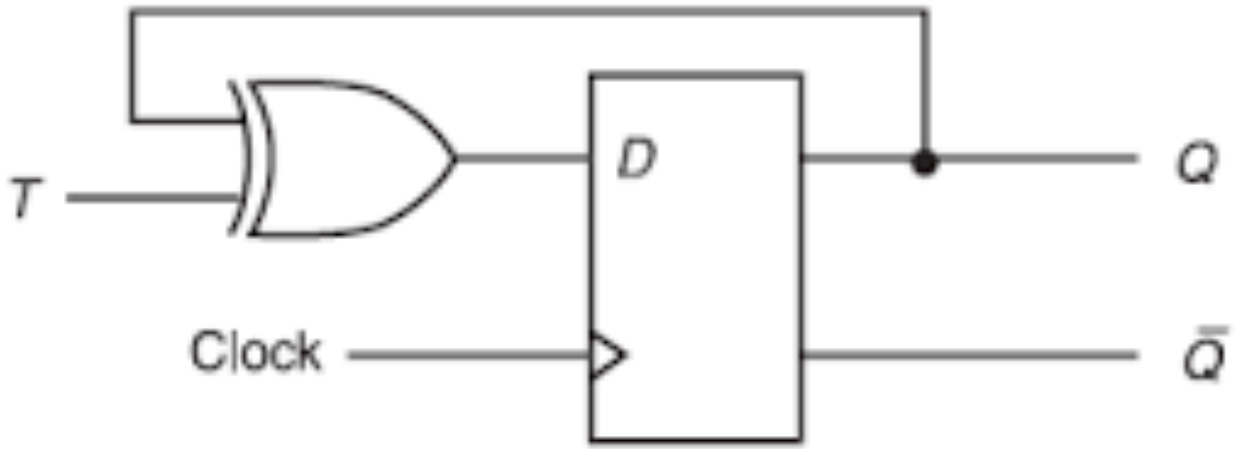
```
Tuple(T1, T2, …, Tn)
```

- **Calling these functions returns a type (class)**
- **Generalizes *higher-kinded* types**

Behavioral Specification

`behavioral`

Sequential Logic

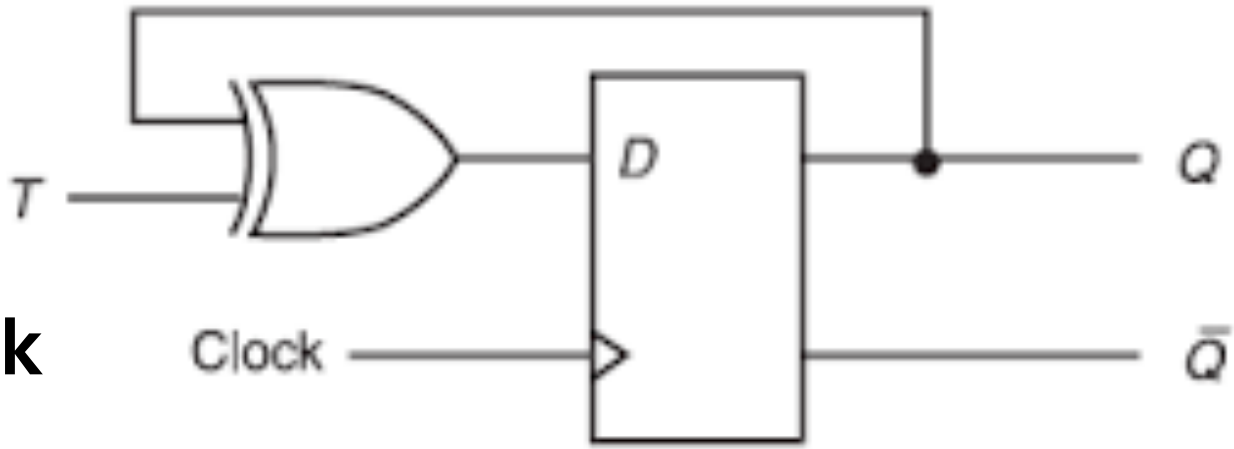


Toggle Flip-Flop

Sequential Logic

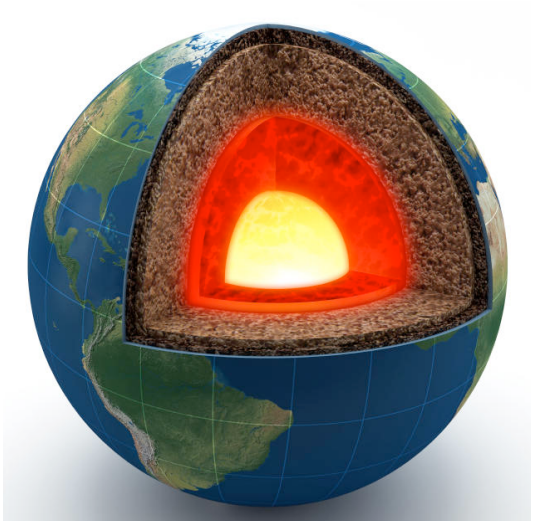
1. Flip-flop/Register
   - Circuit which has an input I, an output O, and internal state
   - CLK and CE
   - RESET and SET
2. Sequential logic
   - Compute next value using combinational logic as a function of the inputs and the previous state (FF output)
   - Wire next value to FF input
   - Results in a cyclic graph of circuits

```
# create ff holding state first
ff = FF()

# lut computes the next state ...
lut = LUT2( I0^I1 )
lut(T, ff)

# lut computes the next state
ff(lut)

wire( ff, O )

# Implicit clock
```



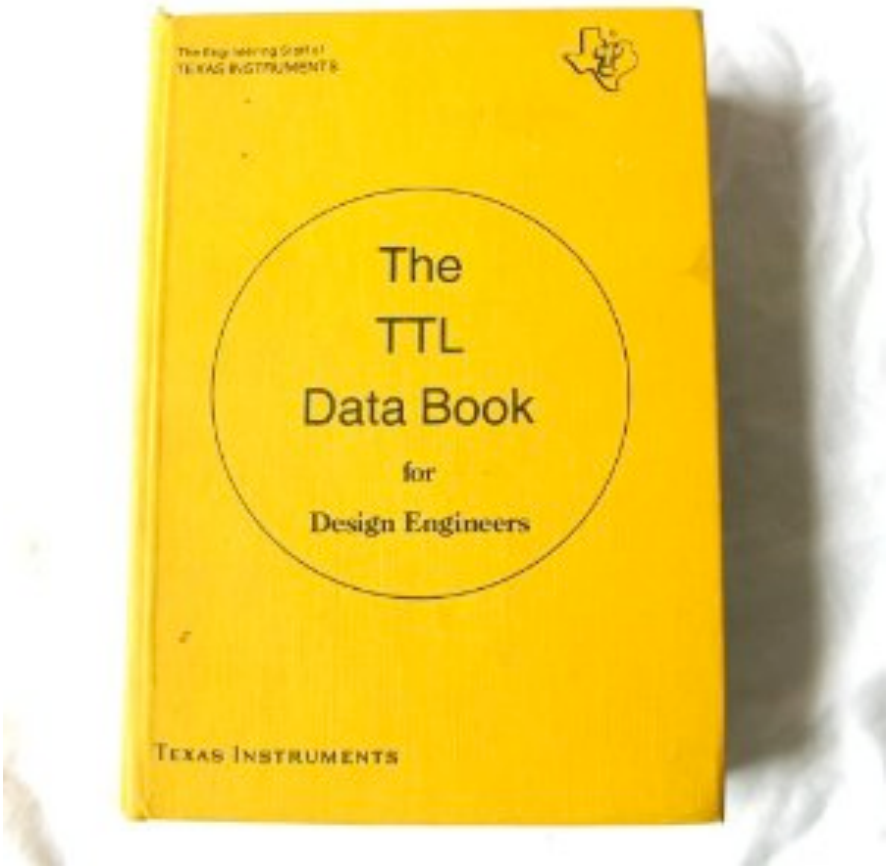
add

counter



Mantle

Standard Low-Level Hardware Library
(think libc and libm, TTL 7400 / CMOS 4000)



Mantle
`libc` for hardware

`And, Or, Xor, …`
`Add, Sub, …`
`Mux,`
`Registers`
`Shift registers`
`Counters`
`Memories`
`…`