Hierarchical Design

• The objective is to produce a complete design.
• A design will consist of several components at various levels of hierarchy.
• A complete design will be the full hierarchy, not a collection of independent files.
Design Hierarchy

Controller Top

CLK GEN

3

Controller

CLK

RESET

RST
Verilog Code to Gates

• The whole point of Verilog design is to end up with a circuit.
• After creating a functional model, the next step is logic synthesis (ECE 527).
• Synthesis is a combination of multi-level logic minimization and technology mapping.
Circuit Timing

• Behavioral code normally contains no timing information.
  – #10 assign A = B & C; //legal but pointless
• Real circuits (silicon) must obey physical laws: capacitance, resistance and inductance all cause propagation delays.
• Library components and delay files are used to model expected performance.
Technology Mapping

• Synthesis algorithms depend on having a library of logic components—standard and complex gates, registers of various types.
• The library is technology-dependent. A new library must be created for every semiconductor process.
• Libraries are a link between logical and physical design.
HDL Design Flow

1. Design Specification
2. HDL Coding
3. Simulation Vectors
4. HDL Simulation
   - Simulation Passed?
     - No: Return to HDL Coding
     - Yes: Logic Synthesis
Concept & Market Research

Design Specification

RTL Coding & Simulation

Logic Synthesis

Scan Insertion

Static Timing Analysis

Timing OK?

No

Yes

Floorplanning, CT Insertion, Global P&R

Static Timing Analysis

Go to P&R
Detailed Routing

Post-layout STA

Timing OK?

Design Rule Check

Passed?

Yes

Tapeout

Collect Bonus, Vacation

Back to Salt Mine

Back to Salt Mine

Yes
Design Flow

• The synthesizer uses the selected library to make a gate-level design that meets specified constraints (area and speed).
• Once synthesis is complete, the resulting gate-level structural design can be simulated with the same vectors used to verify the original design.
Synthesis Source Code

module counter(CLK, RST, COUNT);
    input CLK, RST;
    output [3:0] COUNT;
    reg [3:0] COUNT;

    always @(posedge CLK or negedge RST) begin
        if (!RST) COUNT <= 0;
        else COUNT <= COUNT + 1;
    end
endmodule
module tb_counter();

    wire [3:0] COUNT;
    reg CLK, RST;

    counter   C1(CLK, RST, COUNT);

    initial begin
        CLK = 1'b0;
        forever #10 CLK <= ~CLK;
        end

    initial begin
        RST <= 1'b1;
        #15 RST <= 1'b0;
        #40 RST <= 1'b1;
        end
    endmodule
Mentor Graphics Modelsim output: use only Cadence NC Verilog in class.
Turn Behavioral Code into Gates

• Once the behavior of an HDL description has been verified, turn it into gates.
• Constrain synthesizer for area, speed.
• Point it to a library.
• Check synthesis reports.
• Verify netlist.
  – Often only done for performance (static analysis) as gate-level simulation is slow.
module counter (CLK, RST, COUNT);
    output [3:0] COUNT;
    input CLK, RST;
    wire N2, N3, N4, n5, n9, n10;

    FD2 \COUNT_reg[3] ( .D(N3), .CP(CLK), .CD(RST), .Q(COUNT[3]) );
    FD2 \COUNT_reg[1] ( .D(N2), .CP(CLK), .CD(RST), .Q(COUNT[1]) );
    FD2 \COUNT_reg[2] ( .D(N4), .CP(CLK), .CD(RST), .Q(COUNT[2]), .QN(n5) );
    FT2 \COUNT_reg[0] ( .CP(CLK), .CD(RST), .Q(COUNT[0]) );
    EO U3 ( .A(n9), .B(n5), .Z(N4) );
    EO U4 ( .A(COUNT[3]), .B(n10), .Z(N3) );
    NR2 U5 ( .A(n9), .B(n5), .Z(n10) );
    ND2 U6 ( .A(COUNT[1]), .B(COUNT[0]), .Z(n9) );
    EO U7 ( .A(COUNT[1]), .B(COUNT[0]), .Z(N2) );
endmodule
Logic Synthesis Requires A Library
Post-Synthesis Analysis

Verilog Simulation may be run with original functional verification vectors, the synthesized netlist and a file of extracted parasitic delay data. Static timing analysis is also frequently used.
Simulation-Synthesis-Simulation

Thus a library of components that accurately models the physical characteristics of a device that can be built is essential for the simulation-synthesis-simulation design flow.

Anyone using Verilog/VHDL needs a library. Someone has to make them.
Standard Cell Gate: Physical Model
Verilog Gate Level Models

- Post-synthesis, Verilog gate level models are used for simulation.
- They are the link between abstract RTL (Verilog or VHDL) and a less-abstract netlist.
- Verilog models contain less information than synthesis models: don’t need to know physical size, orientation.
Libraries and Modeling

- Delay Modeling
- Specify Blocks
- Timing Checks
- User-Defined Primitives
Delay Models

In Verilog, there are three delay models:

1. **Distributed Delay**: The delay is distributed across each gate.

2. **Lumped Delay**: The delay is lumped at the last gate.

3. **Path delay (Pin-to-Pin)**: A specify block is used to specify pin-to-pin delays for each path.
Delay Modeling

- Lump delay on the last gate driving the output.
  - Easy to model.
  - Inaccurate.

- Distribute delays across the gates.
  - More accurate than lumped delay methodology.
  - More work.

- Path delays (specify block)
  - Exactly match the delay specification.
Distributed delays can be modeled by assigning delay values to individual gates or by using delay values in individual `assign` statements.

With post-layout back annotation data or pre-layout routing estimates, gates of the same type can have different delays.
// Distributed delays in gate-level modules
module M (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
// Delay is distributed to each gate.
    nand #10 nl (e, a, b);
    nand #6 n2 (f, c, d);
    nand #5 n3 (out, e, f);
endmodule

// Distributed delays in data flow definition of a module
module M (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
// Distributed delay in each expression
    assign #10 e = ~(a & b);
    assign #6 f = ~(c & d);
    assign #5 out = ~(e & f);
endmodule
Lumped Delay

• The cumulative delay of all paths is lumped at one location.
• Delays can be specified as a single delay on the output gate of the module.
• This only works if all paths have the same delay: not usually the case.
Lumped Delay

//Lumped delay module
module M (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;

//Delay is lumped at the output gate.
    nand    nl    (e, a, b);
    nand    n2    (f, c, d);
    nand    #15   n3    (out, e, f);
endmodule
Path delay (Pin-to-Pin delay)

- A delay between a source pin/port (input or inout) and a destination pin/port (output or inout) of a module is called a module path delay. In Verilog, path delays are specified using Specify Blocks.
Features of a Specify Block

• Typical tasks you perform inside a specify block:
  - Describe various paths across the module and assign delays to those paths.
  - Describe timing checks that would ensure that the timing constraints of the device are met.

• A specify block is bounded by the keywords `specify` and `endspecify`, and must appear within a module boundary.
Features of a Specify Block

• Parameters are declared in a specify block using the `specparam` keyword.

• The specify block is a `separate` block in the module and does not appear under any other block, such as `initial` or `always`.

• Inside Specify blocks, delays could be specified as `Parallel Connection` or `Full Connection` Module Paths.
Parallel Connection Module Paths

- A parallel connection is specified by the symbol => and can be between two scalars or two vectors of the same size. It is used as shown below.

  \[( <source\_field> => <destination\_field> ) = <delay\_value>; \]

- In a parallel connection, each bit in source field connects to its corresponding bit in the destination field.

- If the source and the destination fields are vectors, they must have the same number of bits; otherwise, there is a mismatch.

- Thus, a parallel connection specifies delays from each bit in source to its corresponding bit in destination.
//Pin-to-pin delays
module M (out, a, b, c, d);
  output out;
input a, b, c, d;
wire e, f;
//Specify block with path delay statements

specify
  (a => out) = 15;
  (b => out) = 15;
  (c => out) = 11;
  (d => out) = 11;
endspecify
//gate instantiations
nand nl (e, a, b);  nand n2 (f, c, d);  nand n3 (out, e, f);
endmodule
Parallel Connection Module Paths

module noror (O, A, B, C);
output O;
input A, B, C;

nor nl (netl, A, B);
or ol (O, C, netl);

specify
(A => O) = 3;
(B => O) = 3;
(C => O) = 1;
endspecify
endmodule
Min:Typ:Max

• Semiconductor performance is typically specified as a timing triplet: best case, typical case and worst case.
• These are combinations of process, voltage and temperature.
• The three together define the envelope in which the device can be called upon to operate.
Process

• Users have no control over process. Some days the line produces faster chips, some days slower.

• The fab guarantees that the chips they ship will fit someplace between what they do on their best days and what they do on the worst.

• On really bad days no chips are acceptable.
Voltage

- A chip will have a nominal operating voltage, such as 5 for legacy components, less for modern ones.
- Five does not mean 5.000000000
- It’s 5 +/- something, usually 0.5 or 0.25 V. You need some slack on the power supply, it drops out a bit with loading.
Performance and Supply Voltage

• Performance tends to go up (delays go down) with increasing voltage, to a point.
• Thus a circuit will be faster when it has a best-case power supply (i.e. 5.5 Volts) than when it has a worst-case one (4.5 Volts).
Temperature

- Performance also changes with temperature.
- CMOS delays tend to be linear with temperature for both rise and fall times.
- TTL is not, fall times decrease with rising temperatures while rise time increase.
- Each component will have a specified temperature operating range.
Temperature Ranges

• Military temperature range is -55 C to +125 C.
• Commercial is 0 C to 70 C.
• Some processes also have “industrial,” between commercial and military.
• Delay characteristics are specified as a Min:Typ:Max triplet.
Parallel Connection Module Paths

• Examples:

(a => out) = 2.2; // Path delay specified from a to out.

• Specify rise and fall delays.

(r => x) = (1, 2); // Rise and fall delay from r to x.

• Specify rise, fall, and turnoff delays.

(in => out) = (3, 4, 1);

• Specify delays in min:typ:max format.

(in => out) = (1.2:1.4:1.6, 2.1:2.3:2.5);
Full Connection Module Paths

• A **full connection** is specified by the symbol `*>` and is used as shown below.

\[
(\text{<source	extunderscore field>} \ast \text{<destination	extunderscore field>}) = \text{<delay	extunderscore value>};
\]

• In a **full connection**, each bit in the source field connects to every bit in the destination field.

• If the source and the destination are vectors, then they need not have the same number of bits.

• A **full connection** describes the delay between each bit of the source and every bit in the destination.
Full Connection Module Paths

//Pin-to-pin delays
module M (out, a, b, c, d);
  output out;
  input a, b, c, d;
  wire e, f;
//Full connection

specify
  (a, b *> out) = 15;
  (c, d *> out) = 11;
endspecify
//gate instantiations
nand nl (e, a, b); nand n2 (f, c, d); nand n3 (out, e, f);
endmodule
Path Delays

• Examples:

1. \((a, b \rightarrow q, qb) = 12:15:18; \)  //is equivalent to
   
   \((a \Rightarrow q) = 12:15:18;\)
   
   \((b \Rightarrow q) = 12:15:18;\)
   
   \((a \Rightarrow qb) = 12:15:18;\)
   
   \((b \Rightarrow qb) = 12:15:18;\)

• // Path delay specified from a and b to out.
   
   \((a, b \rightarrow out) = 2.2;\)
Path Delays

- // Rise and Fall delay specified from r to o1 and from r to o2.
  \[(r \rightarrow o1, o2) = (1, 2);\]

- // path delays specified from a[l] to b[l] and from a[0] to b[0].
  \[(a[1:0] \Rightarrow b[1:0]) = 3; // parallel connection\]

- // path delays specified for all paths from a to o.
  \[(a[7:0] \rightarrow o[7:0]) = 6.3; // full connection\]
specparam Statements

- Special parameters can be declared for use inside a *specify* block.
- They are declared by the keyword *specparam*.
- Instead of using hardcoded delay values to specify pin-to-pin delays, it is common to define specify parameters by using *specparam* and then to use those parameters inside the *specify* block.
module M (out, a, b, c, d);
output out;
input a, b, c, d;
wire e, f;
//Full connection

specify
// define parameters inside the specify block
specparam in_to_out1 = 15;
specparam in_to_out2 = 11;
    (a, b *> out) = in_to_out1 ;
    (c, d *> out) = in_to_out2 ;
endspecify
NAND Gate Model

`timescale 1 ns / 1 ns
module nand(A, B, Z); //note input/output order reversal
input A, B;
output Z;
nand(Z, A, B);
specify
specparam
  tAZrise = 5,
tAZfall = 3,
tBZrise = 4.5,
tBZfall = 4;
(A => Z) = (tAZrise, tAZfall);
(B => Z) = (tBZrise, tBZfall);
endspecify
endmodule
Specify Block Parameters

- The keyword `specparam` declares parameters within a specify block. Specify parameters (or `specparams`) differ from module `parameters` in scope, and are never interchangeable. The following summarizes the differences between the two types of parameter declarations.

- **Specify Parameters**
  - Use keyword `specparam`
  - Must be declared inside specify blocks
  - May only be used inside specify blocks
  - Cannot use `defparam` to override values
  - Save memory because they are not replicated with each module instance

- **Module Parameters**
  - Use keyword `parameter`
  - Must be declared outside specify blocks
  - Cannot be used inside specify blocks
  - Use `defparam` to override values
  - Use memory because they are replicated with each module instance
Conditional path delays

• Based on the states of input signals to a circuit, the pin-to-pin delays might change.

• Verilog allows path delays to be assigned *conditionally*, based on the value of the signals in the circuit.

• A *conditional path delay* is expressed with the *if* conditional statement.

• The operands can be scalar or vector, module input or inout ports or their bit-selects or part-selects, locally defined registers or nets or their bit-selects or part-selects, or compile time constants (constant numbers and specify block parameters).
Conditional path delays

- The conditional expression can contain any logical, bitwise, reduction, concatenation, or conditional operator.
- The *else* construct cannot be used.
- Conditional path delays are also known as *state dependent path delays (SDPD).*
Conditional path delays

module M (out, a, b, c, d); output out;

input a, b, c, d;
wire e, f;

    //specify block with conditional pin-to-pin timing
specify
    //different pin-to-pin timing based on state of signal a.
if (a) (a => out) = 9;
if (!a) (a => out) = 10;
    // Conditional expression contains two signals b and c.
    // If b & c is true, delay = 9, otherwise delay = 13.
if (b && c) (b => out) = 9;
if (!(b && c)) (b => out) = 13;
    //Use concatenation operator and full connection.
if ({c,d} == 2’b0l) (c,d *> out) = 15;
if ({c,d} != 2’b0l) (c,d *> out) = 13;
endspecify
    nand nl (e, a, b);  nand n2 (f, c, d);  nand n3 (out, e, f);
endmodule
Short Delays

• Short pulses (glitches) may be short relative to the intrinsic delay of a gate.
• Do glitches at the input cause transitions at the output?
• It’s up to the modeling mode. Maybe yes, maybe no.
Glitch Transmission

A = 1, B = 1, SEL transitions from 1 to 0.
Is there a 1 to 0 to 1 glitch on OUT? Assume delay of inverter is less than the delay of a two-input gate.
Inertial Versus Transport Delay Modes

• In **inertial delay mode**, Verilog does not transmit the pulses with a duration that is **shorter** than the switching time of the circuit. This characterizes the behavior of **switching circuits**.

• In **transport delay mode**, every change at the input is reflected at the output, after an amount of time equal to the path delay. This characterizes the behavior of **transmission lines**.

• Since Verilog-XL version 2.1, module path delays have unlimited transport delay functionality. To make use of this functionality, you must invoke Verilog with the option: `+transport-path-delays`
Inertial Versus Transport Delay Modes

- Verilog simulation defaults to inertial delay mode.
- Transport delay must be specified for a simulation session.

\[ \text{Delay} = 2 \text{ ns} \]

Note rejection of pulses that are shorter than intrinsic delay

Note the transport of all input changes to the output.
Sequential Cell Modeling

• Up to now all models have been for Boolean type gates.
• Not much can go wrong with a basic gate.
• Sequential cells are much more complicated.
• Timing constraints may cause a cell to behave not in accord to its Boolean functions.
Timing Checks in Verilog

. Use timing checks to verify the timing of the design.
. In Verilog, timing checks perform the following steps:
  1. Determine the elapsed time between two events
  2. Compare the elapsed time to a specified limit
  3. If the elapsed time is less than the specified limit, report a timing violation.
. Timing checks performed by Verilog are:
  - data setup time
  - data hold time
  - pulse width
  - clock period
  - skew
  - recovery
System Tasks for Timing Checks in Verilog

- $setup(data\_event, \text{ref\_event}, \text{limit}, \text{notifier})$
- $\text{hold(ref\_event, data\_event, limit, notifier)}$
- $\text{setup\_hold(ref\_event, data\_event, s\_limit, h\_limit, notifier)}$
- $\text{width(ref\_event, limit, threshold, notifier)}$
- $\text{period(ref\_event, limit, notifier)}$
- $\text{skew(ref\_event, data\_event, limit, notifier)}$
- $\text{recovery(ref\_event, data\_event, limit, notifier)}$
Timing Checks in Verilog

- **ref event** is the transition of a control signal that establishes the reference time for tracking timing violations on the **data-event**.
- **data-event** is the signal change that initiates the timing check and is monitored for timing check.
- **limit** is the time limit used to detect timing violations.
- **notifier** is a Verilog register that toggles its value every time a violation is reported. It is an optional argument.
- **threshold** is an optional argument that filters out spikes and glitches.
Notifiers in Timing Checks

- When a timing-check violation occurs, Verilog reports a violation and the notifier toggles.
- You can use the notifier to affect the value of the output.
- For example, you can make the output become undefined when a timing violation occurs.
- You can use the notifier to affect the value of the output in two ways:
  - You can make the notifier an input port to a user-defined primitive (UDP).
  - Without declaring a port, a high level behavioral module can still act on the value of the notifier.
$setup(data_event, reference_event, setup_limit, notifier);
$hold(reference_event, data_event, hold_limit, notifier);
$setuphold(reference_event, data_event, setup_limit, hold_limit, notifier);
$skew(reference_event, data_event, skew_limit, notifier);
$recovery(reference_event, data_event, limit, notifier);
$period(reference_event, period_limit, notifier);
$width(reference_event, width_limit, width_threshold, notifier);
$\text{setup}(\text{data}, \text{posedge clk, 20, notifier});$

- The $\text{setup}$ system task reports a violation if the period that elapsed from a change in data to $\text{posedge clk}$ is less than 20.
Timing Checks in Verilog

$\text{hold}(\text{posedge clk, data, 11, notifier});$

- The $\text{hold}$ system task reports a violation if the period that elapsed from $\text{posedge clk}$ to change in data is less than 11.
Timing Checks in Verilog

$setuphold(posedge clk, data, 20, 11, notifier);

- *The $setuphold* system task is a combination of $setup and $hold. This statement creates a setup check of 20 and a hold check of 11.
Timing Checks in Verilog

$\text{skew}(\text{posedge } \text{clk1}, \text{negedge } \text{clk2}, 11, \text{flag});$

- The $\text{skew}$ system task reports a violation if the period that elapsed from $\text{posedge } \text{clk1}$ to $\text{negedge } \text{clk2}$ is greater than 11.
Timing Checks in Verilog

\$\text{recovery}(\text{posedge reset, posedge clk2, 11, flag});

- The $\text{recovery}$ system task reports a violation if the period that elapsed from \text{posedge reset} to \text{posedge clk2} is less than 11.
Timing Checks in Verilog

module dff (data, clock, reset, q, qb);
input data, clock, reset;
output q, qb;
//instantiate the primitives for the basic flip-flop
udp_dff(q_int, data, clock, reset);
buf bl(q, q_int);
not nl(qb, q_int);

//create timing checks
specify
    $setup(data, posedge clock, 12);
    $hold(posedge clock, data , 5);
    $width(posedge clock, 25);
endspecify
endmodule
Conditional Timing Checks

- The special qualifier `&&&` places a condition on a timing check.
- The evaluation of a timing check can be limited to the evaluation of a conditional expression.
- The timing check is performed only when the condition is true.
- The conditional expression must involve only one signal.
Conditional Timing Checks

module dff (data, clock, reset, q, qb);
input data, clock, reset;
output q, qb;

// instantiate the primitives for the basic flip-flop
udp_dff(q_int, data, clock, reset);
buf bl(q, q_int);
not nl(qb, q_int);

// create timing checks
specify
  $setup(data, posedge clock &&& reset, 12);
  $hold(posedge clock, data &&& (!reset), 5);
  $width(posedge clock, 25);
endspecify
endmodule

- The setup check is performed only when \texttt{reset} is high.
- The hold check is performed only when \texttt{reset} is low.
- The width check is performed regardless of the value of \texttt{reset}.
User-Defined Primitives (UDPs)

- UDPs permit the user to augment the set of predefined primitive elements.
- UDPs can represent sequential as well as combinational elements.
- Their behavior is described in a truth table (which takes less time to evaluate than to built-in accelerated primitives).
- A UDP can replace 10 to 20 built-in primitives. Thus, the simulation time and the memory requirements can be reduced significantly, especially if there are a number of instances of that module.
UDP Features

- **UDPs** can have only one output. Thus if the functionality requires more than one output, then the additional primitives need to be connected to the output of the **UDP** or several **UDPs** can be used together.

- **UDPs** can have 1 to 10 inputs. However, the memory requirements increase dramatically as the number of inputs increase from 5.

- All ports must be scalar and no bidirectional ports are allowed.
UDPs **Features**

- The output port must be the **first** port in the UDP port list.
- *UDP* definitions occur **outside** of a module.
- In *UDPs* defining tables, 0, 1, x are the only allowed logic values. The z logic value is not supported. If used, will be replaced by x.
- In *UDP’s* defining tables, columns define the input ports in the same order as in the UDP port list.
- Any combination of inputs that is not specified in the table will produce an x at the output.
UDP Symbols

- In addition to 0, 1 and x, UDP tables can use
  - $\textit{?}$ iteration over 0, 1 and x
  - $b$ iteration of 0 and 1
  - no change

- UDPs also have five edge symbols, covered next
Edge Symbols

- Rising edge: r
  - (01)
- Falling edge: f
  - (10)
- Potential rising edge: p
  - (01), (0x), (x1)
- Potential falling edge: n
  - (10), (1x), (x0)
- Any change: *
  - (??)
Combinational UDP Example: 2-1 Multiplexer

primitive multiplexer ( o, a, b, s);
 output o;
 input a, b, s;

 table
   // a  b  s   :  o
   0 ? 1  : 0;
   1 ? 1  : 1;
   ? 0 0  : 0;
   ? 1 0  : 1;
   0 0  x  : 0;
   1 1  x  : 1;
endtable
endprimitive
Combinational UDP Example: 2-1 Multiplexer

• The first two entries say that if $s = 1$ then, irrespective of the value of the input $b$, the value on the output $o$, will be the same as that of input $a$.

• The next two entries say that if $s = 0$ then, irrespective of the value of the input $a$, the value on the output $o$, will be the same as that of input $b$.

• The last two entries are put in to reduce pessimism. We say that if both inputs, $a$ and $b$, have the same logic value, then even if $sel = x$, the output $o$ will have the same value as inputs $a$ or $b$. This behavior cannot be modeled using Verilog built-in primitives.
Combinational UDP Example: Full Adder

You can implement the full adder with only two combinational UDPS.

// FULL ADDER SUM TERM
primitive U_ADDR2_S (S, A, B, CI);
output S;
input A, B, CI;
table
//A  B  CI : S
0 0 0 : 0;
0 0 1 : 1;
0 1 0 : 1;
0 1 1 : 0;
1 0 0 : 1;
1 0 1 : 0;
1 1 0 : 0;
1 1 1 : 1;
endtable
endprimitive

// FULL ADDER CARRY TERM
primitive U_ADDR2_C (Co, A, B, CI);
output Co;
input A, B, CI;
table
//A  B  CI : Co
0 0 ? : 0;
0 ? 0 : 0;
? 0 0 : 0;
? 1 1 : 1;
1 ? 1 : 1;
1 1 ? : 1;
endtable
endprimitive

For a large number of instantiations of the full adder, the memory requirements are greatly reduced since the full adder now consists of only two primitives (versus five or six if gates are used).
Tristate Cell With UDP

```verilog
`timescale 1 ns / 1 ns
module z_udp(A, B, C, Z);
    input A, B, C;
    output Z;
    bufif1 (Z, U, C);
    udp_AB(U, A, B);

    specify
        (A => Z) = (1,2,3);
        (B => Z) = (2,3,4);
    endspecify
endmodule

primitive udp_AB(U, A, B);
    output U;
    input A, B;
    table
        0 0 : 1;
        0 1 : 1;
        1 0 : 0;
        1 1 : 0;
    endtable
endprimitive
```
UDP Tables and Hi-Z

- UDP tables can use 0, 1 or X
- Z is not a legal value. Z’s are treated as X’s.
- To make a tristate cell, an external primitive is used in the module that instantiates the UDP.
- Options are bufif1, bufif0, notif1, notif0.
- Odd inconsistency: table output of a combinational UDP is neither a reg nor a wire. Making it either would be a syntax error. Output of module may be declared type wire.
Sequential Circuit UDP

• An n-input sequential circuit UDP is declared by n+2 columns table. The first n columns represent the n inputs. The next column represents the present state of the circuit, while the last column represents the next state of the circuit.

• It is necessary to declare the output port as a reg data type.

• There are two types of sequential circuit UDP:
  – Level-Sensitive Sequential UDP and
  – Edge-Sensitive Sequential UDP.
Sequential Circuit UDP

• The output terminal of a sequential UDP can be initialized to a known value at the start of the simulation using *initial* statement.

• Since this does not correspond to any physical device known by your instructor, and the point of a UDP is to model a library component, this construct will never be used.
Level-Sensitive Sequential UDP: Latch

primitive latch (q, Enable, data);
  output q;
  reg q;
  input Enable, data;
  table
    // Enable data present state next state
    //
    0 1 : ? : 1 ;
    0 0 : ? : 0 ;
    1 ? : ? : - ;
  endtable
  endprimitive

Note the use of a register for storage.

Notice the additional field used to specify a next state.

The ? is used to represent don't care conditions in the inputs and current state.
Level-Sensitive Sequential UDP: Latch

The latch behaves as follows:

- When the Enable input is 0, the value on the data input is put on the output.
- When the Enable input is 1, there is no change in the output.
- If the output has '-', then it means that there is no change in the output.
- The output needs to be declared as a `reg` to store the previous value.
Edge-Sensitive Sequential UDP: D Flip Flop

• Transition value are placed between a pair of parenthesis. e.g. (01) represents a transition from 0 to 1.
• Only one input transition can be specified in any table entry statement.
• If any input transition is specified, then all other transitions and the corresponding action must also be specified.
Edge-Sensitive Sequential UDP: D Flip Flop

primitive ff (q, clock, data);
  output q;
  input clock, data;
  reg q;
  table
  // clock data present next
  // state state
  (01) 1 : ? : 1 ;
  (01) 0 : ? : 0 ;
  //reduce pessimism for 0 → x transition
  (0?) 1 : 1 : 1 ;
  (0?) 0 : 0 : 0 ;
  // ignore negative edge of clock
  (?0) ? : ? : - ;
  // ignore data changes on steady clock
endtable
endprimitive
**Edge-Sensitive Sequential UDP: D Flip Flop**

primitive dff1 (q, clk, d);
   output q; reg q;
   input clk, d;
initial q = 1'b1;
   table
      // clk   d   q   q+
      (01) 1   :  ?   : 1 ;
      (01) 0   :  ?   : 0 ;
      (?0)  ?   :  ?   :  - ;
endtable
endprimitive

`timescale 1 ns / 1 ns
module dff (q, qb, clk, d);
   input clk, d;
   output q, qb;
   dff1 #5 df (qi, clk, d);
   buf #(5,6) (q,qi);
   not #(6,5) (qb,qi);
endmodule

The output q has an initial value of 1 at the start of the simulation; a delay specification in the UDP instance does not delay the simulation time of the assignment of this initial value to the output. When simulation starts, this value is the current state in the state table.

Instances of user-defined primitives are specified inside modules in the same manner as for gates. The instance name is optional, just as for gates. The terminal order is as specified in the UDP definition. Only two delays can be specified, because z is not supported for UDPs.
Edge-Sensitive Sequential UDP
Mixing Level-sensitive and Edge-Sensitive

• UDP definitions allow a mixing of the level-sensitive and the edge-sensitive constructs in the same description. An edge-triggered JK flip-flop with asynchronous preset and clear needs such a mixture.

• In the following example, the preset and clear logic is level-sensitive. Whenever the preset and clear combination is 0 1, the output has value 1. Similarly, whenever the preset and clear combination has value 1 0, the output has value 0.

• The remaining logic is sensitive to edges of the clock. In the normal clocking cases, the flip-flop is sensitive to the rising clock edge as indicated by an r in the clock field in those entries.
Mixing Level-sensitive and Edge-Sensitive

primitive jk_edge_FF (q, clock, j, k, pre, clr);
    output q; reg q;
    input clock, j, k, pre, clr;

table
    // clock  j  k  pre  clr  :  q  :  q+
    ?  ?  ?  0  1  :  ?  :  1 ; //preset logic
    ?  ?  ?  1  0  :  ?  :  0 ; //clear logic
    ?  ?  ?  1  *  :  0  :  0 ;
    r  0  0  1  1  :  ?  :  - ; //Normal clocking cases
    r  0  1  1  1  :  ?  :  0 ;
    r  1  0  1  1  :  ?  :  1 ;
    r  1  1  1  1  :  0  :  1 ;
    r  1  1  1  1  :  1  :  0 ;
    b  ?  *  ?  ?  :  ?  :  - ; // b is iteration of 0 and 1

endtable
endprimitive
DFF Using Timing Checks and Notifier

**primitive** dff_udp(q, clock, data, notifier);
**output** q; **reg** q;
**input** clock, data, notifier;
**table** // clock data notifier: state: q

```
  r  0  ? : ? : 0 ; // r is rising edge: 0 to 1
  r  1  ? : ? : 1 ; // ? is iteration over 0, 1 and x
  n  ?  ? : ? : - ; // n is potential falling edge
  ?  *  ? : ? : - ; // * is any transition
```

**endtable**

**endprimitive**
'timescale 100 fs / 1 fs
module dff(q, clock, data);
output q;
input clock, data;
reg notifier;  //notifier is input to udp
dff_udp(q1, clock, data, notifier);
buf(q, q1);
specify
specparam tSU = 5, tH = 1, tPW = 20, tPLH = 4:5:6,
tPHL=4:5:6;
(clock => q) = (tPLH, tPHL);
$setup(data, posedge clock, tSU, notifier);  //setup: data to clock
$hold(posedge clock, data, tH, notifier);  // hold: clock to data
$period(posedge clock, tPW, notifier);  // clock:period
endspecify
endmodule