Verilog's Purpose

Words or Pictures

A Hardware Description Language (HDL)

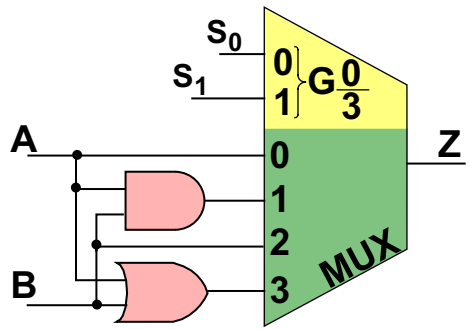
Are words better than pictures?

- For digital design, the words seem to be ahead. (for now)
- In Simulink the pictures are ahead.

```
always @(S0 or S1 or A or B)
 case({S0, S1})
   1: Z = A;
   2: Z = A&B;
   3: Z = B;
   4: Z = A|B;
 endcase
```

Words

Pictures



Two Purposes for an HDL

1. Simulation
   - Simulate parallel components.
   - Simulate timing.
   - Describe operation with higher-level concept (like If and Case).
   - Also describe circuits at gate-level.
2. Synthesis
   - Generates the gate-level description from the high-level one.
   - Acts as input for a logic minimizer.

Verilog For Synthesis[1]

What is the clearest way to describe a circuit

Differential encoding

1. **In words:**

If both inputs are 1, change both outputs.
If one input is 1 change an output as follows:
If the previous outputs are equal
change the output with input 0;
If the previous outputs are unequal
change the output with input 1.
If both inputs are 0, change nothing.

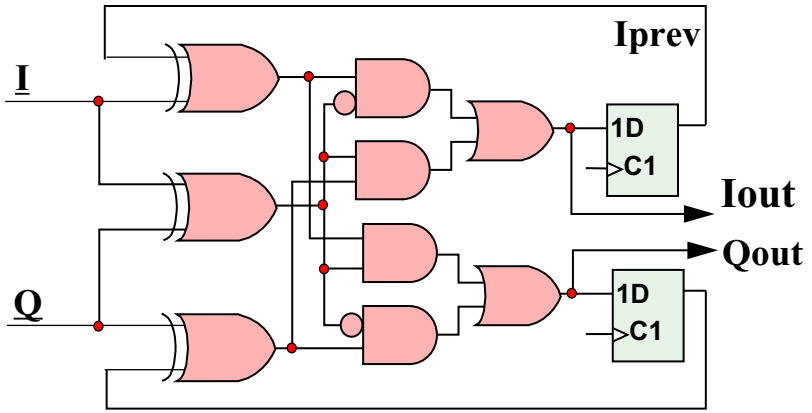
2. **With equations**:

$Iout=(\overline{I \oplus Q})(I \oplus Iprev + (I \oplus Q)(Q \oplus Qprev)$
$Qout=(\overline{I \oplus Q})(Q \oplus Qprev + (I \oplus Q)(I \oplus Iprev)$

3. **By a schematic;** does not give much insight.

4. **By tables;** two forms are shown.
An output vs. input table.
An output-change vs. input table.

Which representation would you choose?



IQ \ (I,Q)prev	00	01	11	10
00	00	01	11	10
01	10	00	01	11
11	11	10	00	01
10	01	11	10	00

Iout,Qout

IQ \ (I,Q)prev	=	≠
00	- -	- -
01	Δ -	- Δ
11	Δ Δ	Δ Δ
10	- Δ	Δ -

ΔIout,ΔQout

1. File Ver1SynM.fm6

Two Purposes: Simulation and Synthesis

Simulation Was Original Purpose

1984 Verilog-XL simulator and language developed by Gateway Design Automation

Synthesis Adopted The Language Later

1987 Verilog based synthesis tool introduced by Synopsys.

What's Good for Simulation May Be So-so For Synthesis

- Computers have different constraints than other hardware.
  They have huge cheap storage.
  They do not mind wasting a few instructions
- Hardware uses small expensive storage (flip-flops)
  Custom hardware is usually chosen for high speed (don't waste operations).
- Compiling as for a computer is an inefficient way to compile to hardware.

calculate y and store

```
y=A&B
```

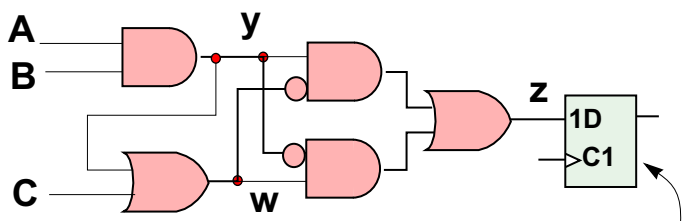
calculate w and store

```
w=C|y
```

calculate z and store

```
z=w(~y) | (~w)y;
```

Will simulate easily, but uses extra storage



Hardware only stores at end

Only a Subset of Verilog is Synthesizable

Developed for Simulation, Used for Synthesis

Verilog was developed as an input language for simulation. In one sense a compiling for simulation is a construction of a logic circuit. It yields a digital machine that gives the right outputs. However it is not an implementation people normally want for synthesis.

The development costs of implementing logic in a microcomputer are much smaller than other methods. Furthermore simple microcontrollers are produced in such quantity that one must produce many millions of a specialized chip to beat their cost. Thus one will normally only do custom logic to gain speed, reduce size, or reduce power. However, if the extra design costs can be justified, one can gain one to three orders of magnitude in speed. size or power by custom logic design.

The implementation for a circuit compiled for simulation is quite different from the implementation for a custom chip. In simulation all intermediate results are saved in memory or an internal register. For a circuit, this practice would waste memory and would slow down the circuit, because each save represents a clock cycle. A compiler for synthesis must remove all unnecessary saves. This is fairly difficult.

Also, when running the logic in simulation, only one logic operation is done at a time in the computer. This is because the computer normally has only one arithmetic-logic unit (ALU). A compilation for synthesis will want to generate parallel hardware. However the change from serial to parallel operations is normally easy.

In summary, although they use the same Verilog input, synthesis is a different, and harder job than simulation.

The Synthesis Subset of Verilog

Not all Verilog commands synthesize easily. For example initial initializing variables) is easy to do in a program where all variables are stored. However in hardware only variables stored in flip-flops are easy to initialize. For this reason only a subset of Verilog is synthesizable. These notes will concentrate on that subset.

Lexicography

Character Set

- 0123456789ABCD..YZabcd...yz_$
  Cannot start with a number or $
- Verilog is case sensitive.
- All keywords are in lower case.

White Space

- The white space characters are:
  space (\b), *tabs* (\t), *newlines* (\n).
- These are ignored except in strings and tokens.
- The space is also used as a delimiter

Comments

- Two types:
  - single line comments starting with

```
// Comment
```

  - multiple line comments delimited by

```
/*  Comment     */
```

- Comments cannot be nested.

```
Tom,   Tom-Jones        No hyphen
Really_Long_Winded_Type_Names
_2cows,  __2bulls,   2guys
abc2 /* is not the same as*/ ABC2

                Key word
assign x = a&b | c&(~d) |
           e&(~g)&f;
// is the same as
assign
x=a&b|c(~d)|e&(~g)&f

A=3+2;  // Comment starts
//  with "||" Good for
//   commenting code.

/*  These comments extend
over multiple lines.  Good
for commenting out code  */
```

Lexicography

Character Set

No hyphen is allowed

There are escaped identifiers which are- \<any ascii characters except white space><white space>. Thus **\#ba** is a legal name but **\#ba,** is not. However **\#ba ,** is all right. The white space placement is critical. This feature seems to have been included for asserted low variables like **\reset ,** that is "reset when low."

Comments

Sometimes /* . . . */ is used to put a comment in the middle of a line of code.

1.• PROBLEMS[1]

Which of the following are valid?

OK$ OK_flip OK#flop OK$latch _OK $OK _ _All_Right_ 23OK June-Bug

```
/* Comment out code
assign  _OK= a|b;    /* "|" is a logic OR  */
assign  Fill= _OK & c;
End of removed code */
assign x = /* If y is over 3 or the gastric resonance will overflow the bedorfulls! */
           y + bedorfull;
```

1. The bad tokens are OK#flop, $OK, 23OK and June-Bug. The nested comment will fail, but the two line statement with an embedded comment is all right. Note a statement ends when a semicolon is reached. A new line does not end a statement.

Statements

Statement Delimiters

- End of Statement is a ";"
- Returns usually do not matter.
- Multiple statements of the same kind can be grouped, and separated by commas.

Data Values

Constants

Specified by number of bits and value.

Integer values are truncated to fit variable size.

Strings

Store in reg, 8 bits per character.

Treated like any other number.

Parameters

Values used during compilation but not synthesized or simulated.

```
assign   Long_Count = A + B + C + D + E
+ F + G + H + I + J;

assign   X=1, Y=2, Z=3;



5'b10111;   //5 bits, binary value 10111.
5'd23;      //5 bits, decimal value 23
5'h17;       //5bit, hex value 17

wire [3:0]  tom, dick;
assign  tom=23;     // is the same value as
assign  dick=4'b0111;


reg [8*5:1] hi;
  initial hi = 'Hello';


parameter n=4;
reg [n-1:0] tom, dick, harry;
parameter Reset_state = 0, state_B = 1;
   Run_state = 2, finish_state = 3;
```

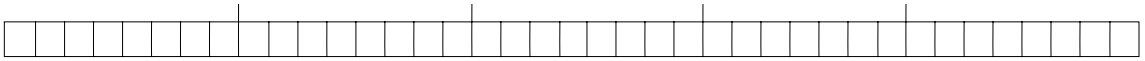
Data Values

Strings

Note that:-

```
reg (8*5:1) hi
```

makes a 40 x1 dimensional array.



Parameters

Can give states meaningful names instead of digits. Alternately use the macro definition **`define**:

```
`define Reset_state = 0, state_B = 1, Run_state = 2, finish_state = 3;
```

Add a backquote when using a macro i.e. `if(state == `Run_state)`

2.• PROBLEMS

ASCII

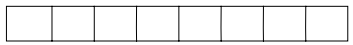
'A'=8'h41
'B'=8'h42
'C'=8'h43
'D'=8'h44

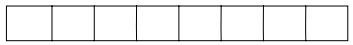
'J'=8'h4a

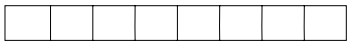
Continuation lines

how do you make them?

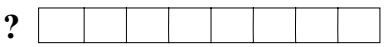
Binary representation of Constants

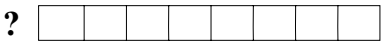
`4'b10` gives ?  1

`4'd10` gives ? 

`reg [8:0] A; initial A=16;` gives ? 

```
reg [8:0] B, C;  initial
                   begin
                     B = 'B'; gives ?
                     C = B+1; gives ?
                   end
```





1. 4'b10 is 

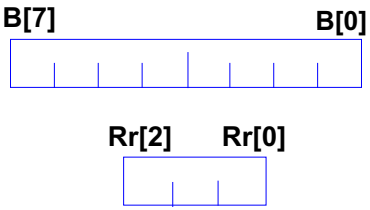
Verilog Variables

Data Types for Synthesis

- ***Wire***
  Used for left hand side of structural code
  Wire **variables synthesize into wires.**
  Inputs are of type wire.
- ***Reg***
  Used for left hand side of procedural code.
  May synthesis into latches and flip flops.
  May also synthesize into wires.
  Most (not all) outputs are of type reg.

```
module typ_sample(W,B,Row,Rr);
input W, B;
wire W, Wx;
wire [7:0] B;

output Rot,Rr;
reg Rot;
```



Two Paradigms

1. Procedural

Think like C code

Example:

```
reg c, d;
wire a, b, e;
always @ ... //Starts a Verilog procedure
  begin
    c = a & b;   // Store c in a "reg"
    d = c | e;   // Store d in a "reg"
```

More later

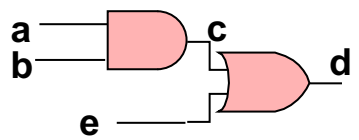
2. Structural

Think like a circuit

Example:

```
wire c, d, a, b;
  assign c = a & b; // Circuit
  assign d = c | e; // Circuit
```

Assign is used with structural coding.



Verilog Data Types

Wires

These always correspond to a wire or a bus of wires in a circuit.
Unfortunately **reg** may also correspond to wires.
However wires never store. They just transfer inputs to outputs.

Note the "dimension" of the bus (vector) comes before the name of the bus.

Reg

These correspond to variables in the C language. In Verilog for simulation they are always stored.

In synthesis the compiler will generate latches or flip-flops for them. However if it can be sure their output does not need to be stored it will synthesise them into wires. It can be sure they do not have to store if their outputs is based only on their present inputs.

Rule for reg and wire.

A variable is declared type **`reg`** if it appears on the left hand side of an equal sign in a procedure.
A procedure starts with the word **`always`** or **`initial`**.

A variable is declared of type wire if it appears on the left side of an equal sign in structural code.
Structual code statements start with the word **`assign.`**

3.• PROBLEMS

```
reg [3:0] AA;  always . . .
    AA=15; // Will AA synthesize into wires, flip-flops, or latches?[1]
wire [7:0] BB;  assign BB = AA + BB; // Will this statement generate storage?
```

1. Clearly AA does not need to be stored since it is always a constant. Just hard wire it to four 1s.
The second statement will not generate storage because wire variables are never stored. The statement is an asynchronous feedback loop where BB keeps incrementing by 15 continuously. No working circuit like that could ever be synthesized.

Data Types for Synthesis

- ***Integer***
  Normally used for things like loop indexes which do not synthesize.

  Converted to right number of bits automatically if stored in a scalar or vector

- ***Scalar***
  A single `wire` or `reg` (like W or Wr) is a scalar. It can contain only 1 bit .

- ***Vector***
  A *wire* or *reg* made of multiple bits.
  `reg [7:0] B;`

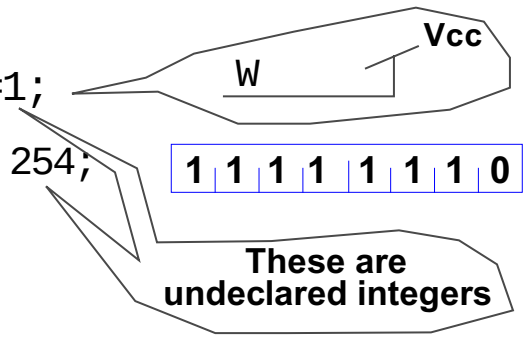
- ***Array***
  A 2-dimensional array of *wire* or *reg*.

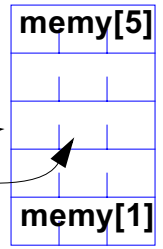
  Operations are very restricted

```
wire W;
wire [7:0] B;
reg Wr;
reg [2:0] Rr;
integer IMe; initial IMe=4012;

// Integer "1" converted to 1 bit and assigned
// to a scalar
assign W=1;
assign B= 254;

// Array
reg [2:0] memy [5:1]
  . . .
Rr=memy[3];
Wr=Rr[1];
. . .
```





Data Types

Integers

Integers do not synthesize to physical hardware, unless they are synthesized as power and ground connections as shown.

Beware; integers are the only two's complement numbers in Verilog. Recall -1 as bit vector is all ones. Thus-

```
assign B = -1;    // would put a value of 255 in B. since on the slide B is defined with eight bits.
```

Vectors

Conventionally arrays are dimensioned [7:0] (left down to right) so the most-significant bit of an 8-bit bus is number 7. One can go [0:7] or [8:1] or [1:8] or even [15:8] if one wishes.

Array

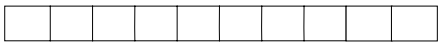
The example shows a **reg** array which is far the most common.
However **wire** arrays can be made.

Verilog is very restrictive for arrays. Their is little programming convenience in using them. The advantage of arrays is in specifying embedded RAM. Arrays must be accessed like a memory, that is only one word at a time.

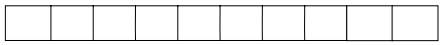
```
reg [7:0] memry [0:1023];  //Storage
  // One can only get at rows of the array directly. Define a vector to extract bits.
  reg [7:0] Mem_Word;  //Not storage
  Mem_Word = memry[997];
```

Problems: what are the bit patterns?

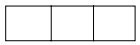
```
reg [9:0] B; initial B =  1;
```



```
reg [9:0] B; initial B = -1;
```



```
reg [5:7] H; initial H=8'ha6;
```



hex
8'ha6 =1010_0110

Operators

Verilog has three types of operators,

They take either one, two or three operands.

1. **Unary operators appear on the left of their operand:**

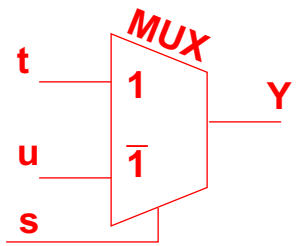
```
clock = ~clock;     // ~ is the unary bitwise negation operator,
                    // clock is the operand
```

2. **binary in the middle,**

```
c = a || b;         // || is the logical or, a and b are the operands
```

3. **and ternary separates its three operands by two operators.**

```
r = (s) ? t : u;    // ?: is the ternary conditional operator, which
                    // reads r = [if s is true then t else u]
                    // Verilog has only one ternary operator
```



Operators, General

Unary operators

They are:

~, **!**, sometimes **-** or **+** as in **-2** or **+3,** and in one sense the reduction operators (next slide).

Ternary operator

The only one is:

r = s ? t : u

One can see binary operators are by far the most common.

1

1. From the previous page: r**eg** [5:7] H; **initial** H=8'ha6 gives H=110. The numbers do not care what subscripts you put on their bits.

Operators

Arithmetic

+, -, *

% (modulus) / (divide)

Relational

<, <=, >, >=, == !=

Logical

! (not), &&, ||

Shift

A<<3 B>>1

Bitwise

~(not), &, |, ^ (xor), ~^ or ^~

Reduction

&, ~&, |, ~|, ^(xor), ~^ or ^~

Concatenation

{ }

Conditional

(condition)? if true: if false;

- Modulus and division are for test benches only. Not for synthesis.

  12%5 ==> 2

- Logical: The whole variable is treated as false (0) zero, or true (1) for anything but zero.

  ```
  27 && -3 ==> 1
  27 && 0 ==> 0
  A || 33 ==> 1      (for any A)
  ```

- Shift A left 3 bits and zero fill
  Shift B right 1 place and zero fill

- Bit-by-Bit operations between two variables.

  ```
  5'b11001 ^ 5'b01101 ==> 5'b10100
  5'b11001 & 5'b01101 ==> 5'b01001
  ```

- Reduction: Between the bits of one variable.

  ```
  & 5'b01101 ==> 0
  ^ 5'b01101 ==> 1
  ```

- Concatenate:

  ```
  wire [2:1] A, wire [3:1] B;
  wire [5:1] C;
   C={A, B};
  ```

- Conditional:

  ```
  // Increment A if C==D, else decrement A.
    assign  A = (C==D) ? A+1 : A-1:
  ```

Operators

modulus

a%b is the remainder of a/b. **7%3=> 1, 13%15 => 13**

Other Operators

Replication

A useful, but not widely used operation.

Concatenation of n copies of the same thing can be written {n{X}} instead of {X,X,X, . . .X}

Thus to fill an 8-bit word Z with 8 copies of the least-significant bit of word W, use:

```
wire [7:0] W, Z;
assign Z = {8{W[1]}};
```

4.• PROBLEMS

```
reg [2:0] A, B;
initial begin  A=3'b111; B=3'b101; end
```

To what value would the following expressions evaluate?

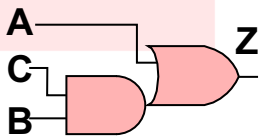
a) **A && B;**
b) **A & B;**
c) **& A;**
d) **& B;**
e) **B>>1**
f) **B<<2**
g) **{A,B}**
h) **(A~^B)= =(B^~A)**

Verilog Structure

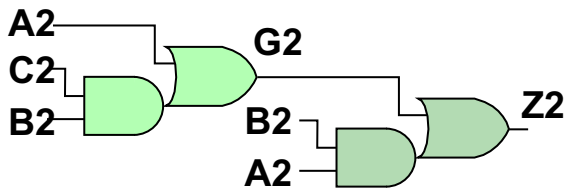
The Module

- The subroutines[1] of verilog
- All code is contained in modules.
- Modules can invoke other modules.
- Modules are never defined inside other modules.
  Think: any module definition can be made a separate file.
- In C one thinks of calling one module (procedure) and then calling it again in sequence.
- In Verilog one builds two instances of a module. They both exist and run at the same time.

```
module gate(Z, A,B,C);
   input A,B,C;
   output Z;
      assign Z = A|(B&C);
endmodule
```




```
module two_gates(Z2, A2,B2,C2)
   input A2,B2,C2;
   output Z2;
   gate gate_1(G2, A2,B2,C2);
   gate gate_2(Z2, G2,A2,B2);
endmodule
```



Two instances of "gate."

1. Procedure has a different meaning in Verilog.

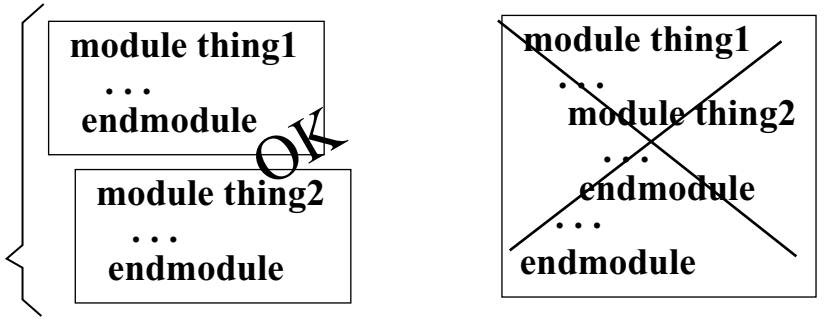
Verilog Structure

Modules

All statements in Verilog are contained between:

module
//and
endmodule

Modules cannot contain another module's definition.

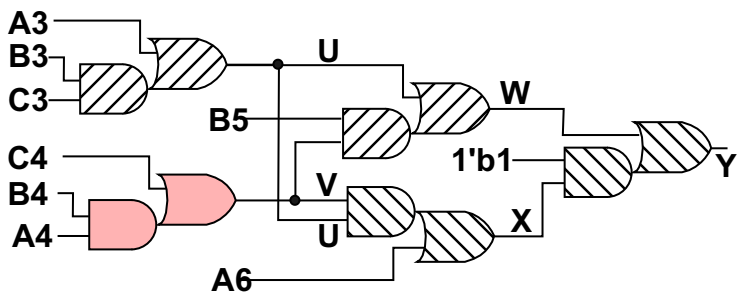


Modules can contain invoke another module. On the slide, module *two-gates* invokes module *gate* twice. Once as gate_1 and once as gate_2.
These are called two different *instances* of *gate*.
Instances are not recursive.

5.• PROBLEM,: complete the module using submodules

```
module five-gates(Y, A3, B3, C3, C4, B4,
        A4, B5, A6);
input  A3, B3, C3, C4, B4, A4, B5, A6;
wire  A3, B3, C3, C4, B4, A4, B5, A6,
      U, V, W, X;          //[1]
output Y;                  // Is Y reg or wire?
assign U=gate gate_u( ...


endmodule
```



1. Module interconnections default to type Scalar **wir**. However making a wir declaration reminds you check the defaults are correct.

A Hierarchy of Modules

The *main* module is the *test_bench*

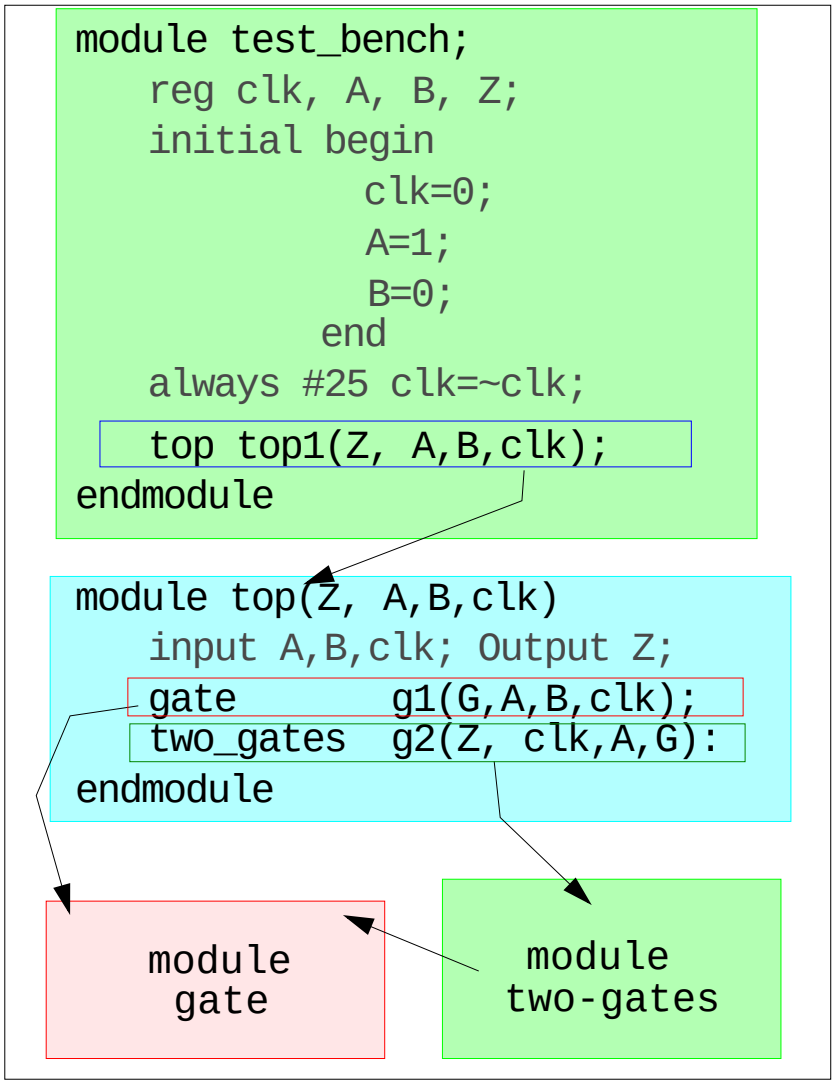
- It generates all signals to feed the module, like the `clk`.
- It likely prints out the outputs (not shown here).
- It is never synthesized. It is only simulated.
- It drives the simulation of the other modules before synthesis.
  - Then they are synthesized into gates
  - Then it drives a check simulation on the synthesized gates.

The *top* module

- Is the top of the synthesized code.
- It often collects the other module invocations.
- The chip I/O signals pass thru *top.*

The other modules

- They collect at the bottom



Module Hierarchy

Common hierarchy

The hierarchy on the slide is very common.
The test bench only invokes one module.
This corresponds to the "pins" on an IC.

Note module A will have three *instances* but only one *module definition*.

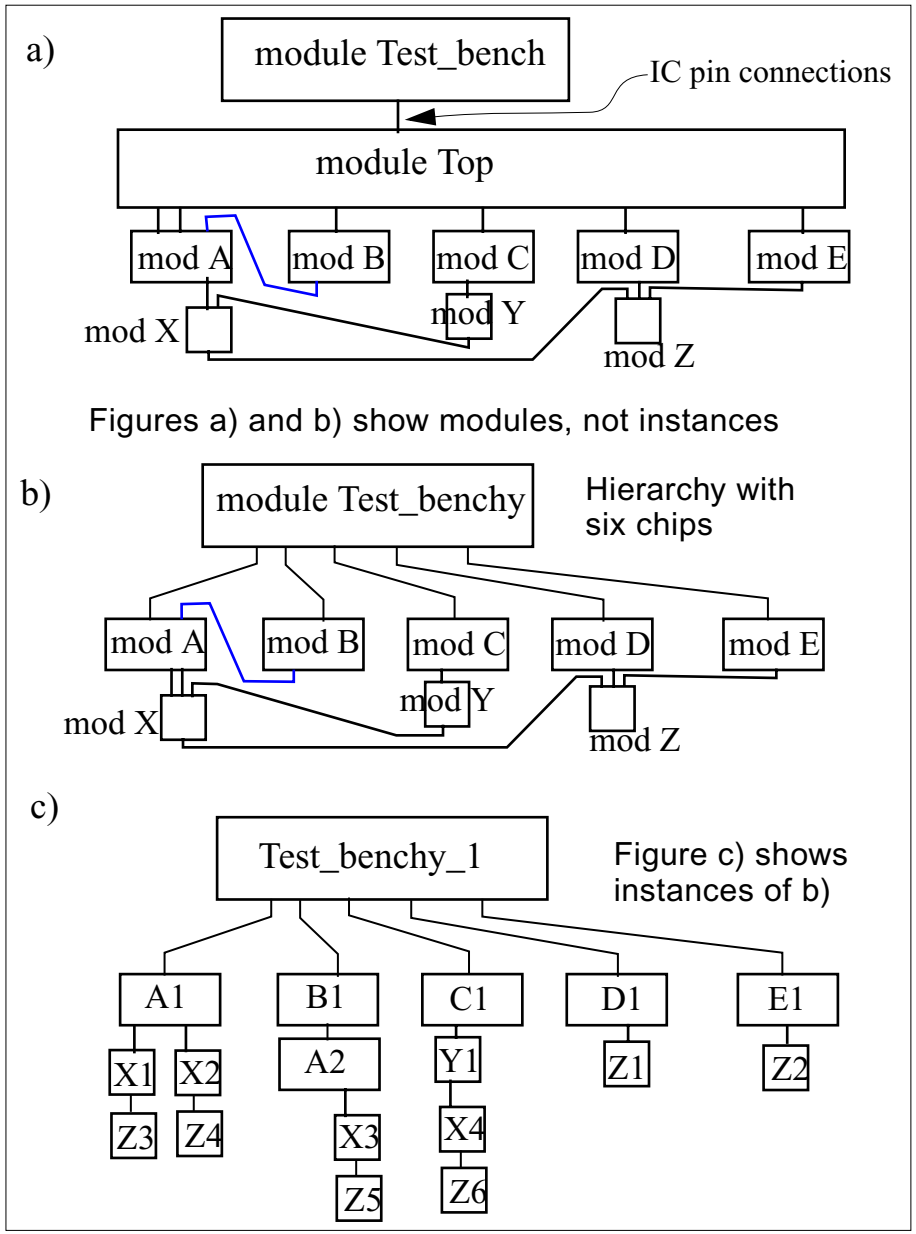
Another hierarchy

If one was designing a system of say six chips, one might want to structure the modules differently.

I said six chips because one has two *instances* of the chip A defined by module A.

6.• PROBLEM

If one line entering the top of a block represents an instance, list the number of instances each module in the figure a) will have.



Structural vs Procedural Verilog

Two Paradigms

1. Procedural

Think like C code

Example:

```
reg c, d;

always @.... /* Starts a Verilog procedure,
                more later. */
  c = a & b;
        // Store c in a register
  d = c | e;
        // Store d in a register
```

- After c is stored, changing a, b, or e does not change c or d.
- Order of statements is important.

```
d = c | e;
c = a & b;
```

- Every statement requires storage by default.
  - This is all right for simulation where storage is cheap.

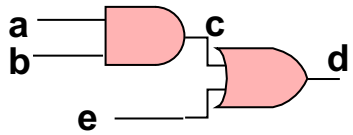
2. Structural

Think like a circuit

Example:

```
wire  c, d;

assign c = a & b; // Circuit
assign d = c | e; // Circuit
```



- Changing a, b, or e any time may immediately change c, d.
- Reversing statement order does nothing

```
assign d = c | e;
assign c = a & b;
```

- Add flip-flops for storage but only when needed.
  - Minimizes expensive flip flops.

Structural vs Procedural Verilog

Structural Verilog

Structural Verilog code looks like a *netlist*, a textual description of the schematic.
Structural code is written with some combination of :

- **`assign`** statements as is shown, and/or
- interconnections of modules.

Statement order in the code has no more meaning than where on the page one puts a schematic symbol.

Procedural Verilog

The Verilog code looks much like c code. Procedures always start with
initial, or
always.

`Initial` procedures start at time = 0, run sequentially through the statements in the procedure block.
`Always` procedures start at time = 0.[1] They run sequentially through the block. However at the end of the block they come back and run through the block again. That is the reason for the word “always.”

Memory

When running a computer, one has huge amounts of dynamic RAM but little parallel calculation ability. Thus all calculation results are stored. When running logic there is a large amount of parallel computation ability, but storage is in expensive flip-flops.

1. However they can be combined with an @(*some_time*) statement which starts them later.

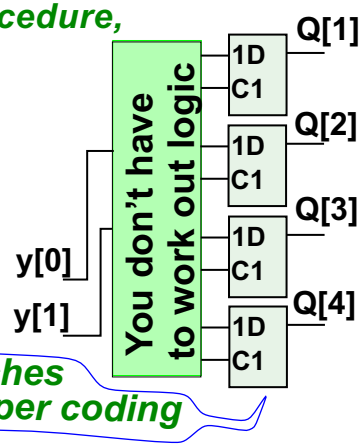
Two Paradigms (cont.)

1. Procedural

- Describe how a function works, but not how to build it.
  Usually easier to code.

Example: 2-bit to one-of-4 decoder

```
reg [4:1] Q;
wire [1:0] y;
always @ ... // Start of procedure, more later.
  case(y)
   2'b00:  Q[1]=1;
   2'b01:  Q[2]=1;
   2'b10:  Q[3]=1;
   2'b11:  Q[4]=1;
  endcase
```



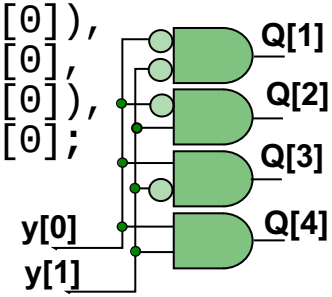
- Easier to code.
  Let the synthesizer do the logic.
- In c-like code we depend on storing each result
- The synthesizer may add latches, but they can be avoided by proper coding.

2. Structural

- Describes how to build a circuit usually at gate level, hard to follow.

Example: 2-bit to one-of-4 decoder

```
wire [4:1] Q;
wire [1:0] y;
assign
  Q[1]=(~y[1]) & (~y[0]),
  Q[2]=(~y[1]) &   y[0],
  Q[3]=  y[1]  & (~y[0]),
  Q[4]=  y[1]  &   y[0];
```



- Have to work out the logic to write code.
- All Q values are calculated in parallel by hardware. No need to remember.

Two Paradigms for Synthesis

Structual code

Structual code is like a schematic in words. Little thought is required to converted to hardware. However a large program of structural code is hard to write and hard to read.

Procedural code

This code is easier to write. The problem is computers have much memory and only one arithmetic logic unit. Thus they tend to store all intermediate results. One would expect to store C in:

```
C = A & B;
F = C ^ E;
```

In hardware one can have plenty of extra gates, and one would feed the output of the AND directly into the XOR.

The compiler must decide when to insert storage and when not to. This can get complicated.

Two ways to write procedural code

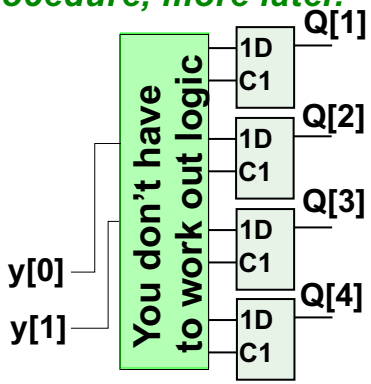
1. Procedural with latches

- The synthesized circuit must give the same result as the simulation.

Example: 2-bit to one-of-4 decoder

```
reg [4:1] Q;
wire [1:0] y;
always @ ... // Start of procedure, more later.

case(y)
   2'b00:  Q[1]=1;
   2'b01:  Q[2]=1;
   2'b10:  Q[3]=1;
   2'b11:  Q[4]=1;
endcase
```



- When y changes the code executes.
- Only one of the cases is selected. One new Q is calculated.
- Must remember the 3 other Q values?
- In C they are stored in memory. In synthesis they are stored in latches.

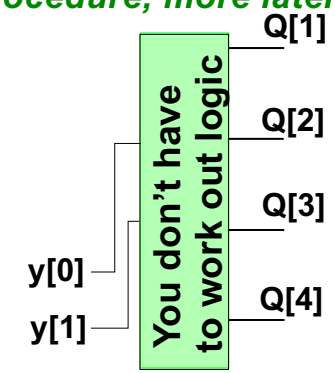
2. Procedural without latches

- The synthesized circuit must give the same result as the simulation.

Example: 2-bit to one-of-4 decoder

```
reg [4:1] Q;
wire [1:0] y;
always @ ... // Start of procedure, more later.

Q=4'b0000;
case(y)
   2'b00:  Q[1]=1;
   2'b01:  Q[2]=1;
   2'b10:  Q[3]=1;
   2'b11:  Q[4]=1;
endcase
```



- When y changes the code executes.
- Q=4'b0000 calculates four Q values.
- Case overwrites one of them.
- Nothing has to be remembered.
- This synthesis does not need latches.

Structural vs Procedural Verilog (cont.)

The procedure must evaluate all four Q values each time it is run through.
If it does not then it must maintain the old values calculated at some earlier time.

One can remove the need for latches by adding one statement to the code on the slide.

```
always @ ... // Start of procedure
Q=4'b0000; // Statement to add
 case(y)
   2'b00:  Q[1]=1;
   2'b01:  Q[2]=1;
   2'b10:  Q[3]=1;
   2'b11:  Q[4]=1;
   endcase
```

Without `Q=4'b0000;`

Here only one-of-four Q values is calculated so three latches are needed on any pass through. Four since a different three are needed on different passes.

With `Q=4'b0000;`

Here all four are calculated each time the procedure is run. This will synthesize to a circuit without latches that will give the same result as if the procedure was executed in C code.

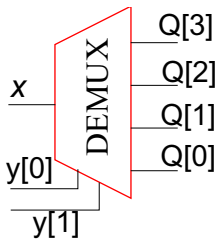
7.• PROBLEMS

a. Write a code segment for a 2-bit to one-of-4 decoder which includes the statements:
reg [3:0] Q; Q=4'b0000; Q[y]=1;

b. Write a code segment, for a 4-output demux, in procedural Verilog.
It will look very much like the 2-to-4 decoder.

c. Write it in structual code using the replicate operator. See “Replication” on page 16.

```
assign  Q = {4{x}} & {y[0]&y[1], ...
```



Why Procedural is Better ⇔ Why Structural is Better

1. Procedural

Easier To Code

- Don't have to work out logic.
- Can code like we learned to do in C.
- Can use *case, while, for, if* . . . Conceptually difficult in structured coding.

40

Synthesis is Harder

- C compilers code into hardware which:
  - runs one instruction at a time.
  - they store every result.
- Synthesizers build a machine which:
  - calculate results in parallel.
  - feed results forward without storing.
- Synthesizer must substitute parallel calculation for storage.

2. Structural

Harder to Code

- Designer has to know what circuit will do at a logical level.
- Synthesizer will only minimize logic.
- Coding style is unlike what one learned in Programming 101.
- Unclear how to incorporate high level concepts: *case, while, for,* . . .

Synthesis is Easier

- First pass of converting to logic is done.
- Synthesis is logic optimization. Can optimize for:
  - area
  - speed
  - testability
  - power

Why Procedural is Better ⇔ Why Structural is Better

Procedural Synthesis is Harder

- Compilers generate code for hardware which:
  - runs one instruction at a time.
  - latches every result.
- Hard to compile into a machine which:
  - runs many operations in parallel.
  - continuously calculates parallel results.
  - can feed the results of one calculation directly into another without storing.
- Hardware memory is flip-flops and is expensive. Synthesizer must avoid using much of the storage a compiler would tend to put in.

Only always@ is Synthesizable

`always` without an @ condition repeats continuously. It is an infinite loop with no delay between loops. It is tamed in simulation by placing delays inside the procedure, but one cannot synthesize numerical delays. One can place the @ command as a separate statement inside the procedure. See “Time In Verilog” on page 37.

A Verilog Procedure

Starts with `always` or `initial`.

```
always@(some condition)
    begin
      . . .
     Statements including
       if
       case
       for
       while
       . . .
    end
```

```
always
    begin
      . . .
     Statements inc'd
       if
       case
       for
       while
      . . .
    end
```

```
initial
    begin
      . . .
     Statements inc'd
       if
       case
       for
       while
      . . .
    end
```

- `begin` and `end` bracket the procedure.
- `initial` is not synthesizable and is used for test benches. `always` without @ condition, is normally only used in test benches.
- Variables on the left-hand side should be of type `reg`, or at least not of type `wir`.

type reg

```
w = a+ c;
```

Verilog Procedures

Commands only Usable Inside a Procedure

for
while
forever
repeat
disable
if. . . else . . . elseif
case, casex, casez

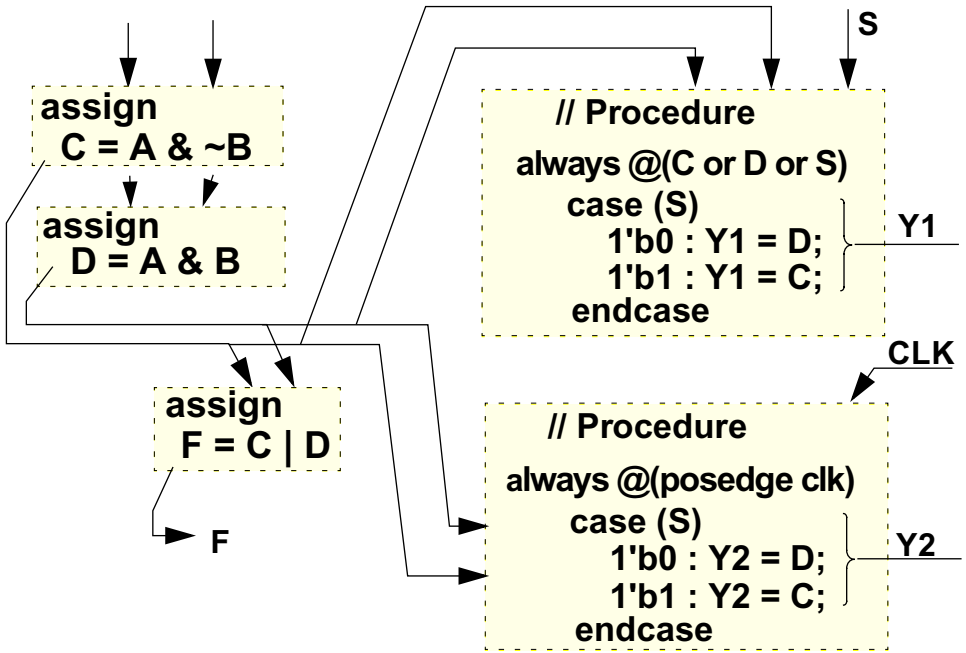
Procedures run Concurrently

Several or all of the procedures may be active at the same time, just as different parts of a logic circuit can execute at the same time. Being active here means changing values.

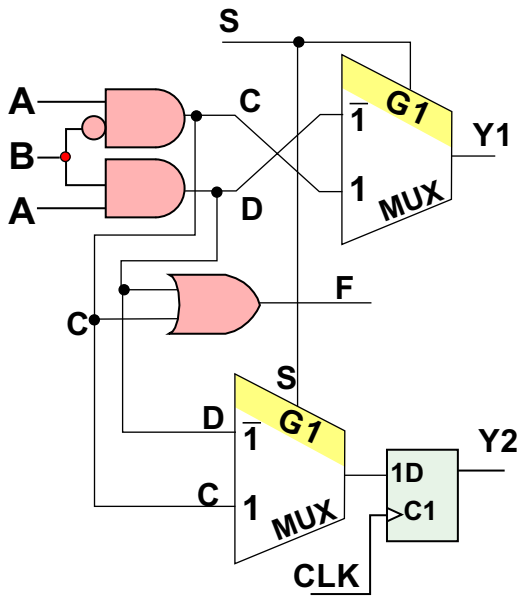
The HDL Compromise

1. Make Procedures a Subunit of Structural Code

- Inside a procedure one codes like in C.
- Outside, a procedure it is a chunk of hardware.
- All procedures run in parallel with other procedures



How to think of Verilog (especially procedures) as a circuit



What will be synthesised from the code

Procedure as Blocks in a Circuit

Procedures are less formally denoted than in many languages.

In hardware, the whole procedure is a block of circuitry, just a big complex gate.

8.• PROBLEM

Does this code segment require the synthesizer to generate storage?

```
reg [3:0] Q;
wire [1:0] y;

always @(y)
 case(y)
   2'b00:  Q[0]=1;
   2'b01:  Q[1]=1;
   2'b10:  Q[2]=1;
   2'b11:  Q[3]=1;
   endcase
```

The HDL Compromise

2. Only Put in Latches If Necessary

- Many procedures do not need to store values.
- If all left-hand values can be calculated from a single procedure entry, nothing needs to be stored.

Example: Enabled 2-bit to 1-of-4 decoder.

```
reg [4:1] Q;
wire [1:0] x; wire Enb;
// Procedure
always @(x or Enb)
 begin
   if(Enb) Q[x]=1;
    else  Q=4'b0000;
 end
```

- If Enb=1, only Q[x] calculated.
- The other three Q[i]s must be remembered from when Enb = 0.
- To remember, insert latches.
- To behave like C code this circuit needs latches.
- NOT A GOOD WAY TO WRITE CODE

```
reg [4:1] Q;
wire [1:0] x; wire Enb;
// Procedure
always @(x or Enb)
 begin
    Q=4'b0000;
    if (Enb) Q[x]=1;
 end
```

- If Enb=1, Q[x] immediately overwrites Q=4'b0000.
- All Q[i]s are calculated in one entry.
- No need to remember.
- This circuit behaves like C code but uses no latches.
- BETTER WAY TO WRITE CODE

Avoid Unnecessary Latches

To avoid unnecessary latches, one must code carefully.

One of the rules, illustrated above, is:-
Every time one executes a procedure all of the variables defined anywhere in the procedure must be calculated. Otherwise latches will be generated.

9.• PROBLEMS:

Assuming the following statements are alone inside a procedure,[1] will they generate latches?

a)
```
if (a>1) y=1; else y=0;
```

b)
```
if (a = = 3) x=1; else y=1;
```

c)
```
if (a)  begin x=1; y=1; end
```
// else do nothing

d)
```
if (a)  begin x=0; y=0; end
   else begin x=1; y=1; end
```

e)
```
x=1; y=1;
if (a)  x=0; else y=0;
```

1. For those who know about the trigger list, assume it is **`@(a )`**.

Avoiding Unwanted Latches: Rule 1

If the procedure has several paths,
every path must evaluate all outputs

Else synthesis will insert latches to hold the old values of those unevaluated outputs.

Methods

Method 1:

Set all outputs to some value at the start of the procedure.
Later on different values can overwrite those values.

```
always @(. . .
  begin
    x=0; y=0; z=0;
       if (a) x=2; elseif (b) y=3; else z=4;
end
```

Method 2:

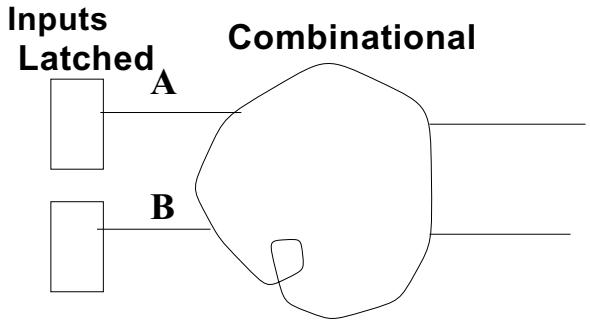
Be sure every branch of every `if` and `case` generate every output.

```
always @(. . .
      begin
        if (a)           begin   x=2; y=0; z=0;  end
            elseif (b)   begin   x=0; y=3; z=0;  end
                 else    begin   x=0; y=0; z=4;  end
    end
```

Writing Procedural Code Without Latches

Eliminating Latches

Let the inputs to a combinational logic block be held by latches, flip flops, or by input switches. Then the outputs only change if an input(s) change.



To duplicate the behaviour of a combinational block with sequential code, one need only be sure the outputs are re-evaluated every time any input changes. Then nothing needs to be stored inside the combinational block.

All outputs must be evaluated on all paths.

If the procedure has several paths, every path must evaluate all outputs. Otherwise latches must be inserted to hold the previous values of those unevaluated outputs.

Methods

Method 1:

Set all outputs to some value at the start of the procedure.

Later on different values can overwrite those values. In simulation, with delays, one could get glitches by doing this. For synthesis one does not use delays within procedures and the glitches are at worst 0 ns.
The synthesizer will take out the extra “writes” during logic minimization.

Method 2:

Be sure every branch of every *if* and *case* generate every output.

This is usually a lot more work for the coder. The synthesizer can handle either method.

Time In Verilog

There are three concepts of time.

1. *# <delay> <event>*
   This is not useful for synthesis.
   One cannot synthesize a given delay.

```
Q = #5 x&y;
// Q changes 5 units after x or y changes
```

2. *@<edge-triggered event>*
   For procedural code.
   Proceed past this point when the correct edge happens.

```
@(posedge clock) //
                 // Respond to rise
@(A or B)  // Respond to rise and fall.
@negedge clock // Respond to fall
```

3. *wait(<signal_has_high_level>)*
   Proceed when signal has a high logic level.

   Not supported for synthesis
   Handy in test-benches.

```
wait( enable)
  #10 count = count + 1;
// Count every 10 time units while enable =1
```

Time in Verilog

The difference between wait and @

In some cases edge and level triggered act the same. In some places they are not.

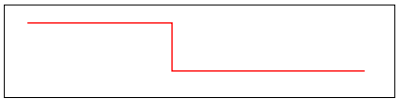
`wait(x)` //will trigger if x starts out high. It will continuously trigger if x stays high.

`@(posedge x)` //will need an edge.

A common reset problem

Reset generation in test bench.

```
initial begin  reset=1; #1 reset=0; end
```

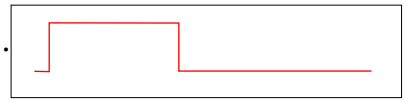


Possible reset implementations inside flip-flops

```
always @(posedge reset) Q=0; //will not reset.
```

```
always wait (reset) Q1=0;  //This will reset, but wait causes an infinite loop (see below).
```

Better make your test-bench reset-pulse a true pulse rather than a step.



An infinite loop

```
always
    begin
     wait(!reset)  x=1;   // As soon as reset goes low, this is a zero delay loop.
    end                      // This will stop a simulator from advancing the simulation..
```

Timing and Procedures

Execution of Procedures

- Two procedure types:
  initial
  always>
- `initial` implicitly starts at t=0.
- `always` must explicitly state <u>when</u> in will be executed

Initial

1. `Initial`
   Starts running at t=0.
   Continues until told to stop.
2. `Initial` is not synthesizable
   - Used mainly for test-benches.
3. Initializes test bench variables
4. Simulation must have an initial procedure that ends with `$finish`

```
//Typical initial procedure
Initial
  begin                          t=0
        clk =0;                  t=0
  #5    A=1;                     t=5
  #5    B=0;                     t=10
  end    // end after 10         t=10
units.

//Common test-bench clock.
 initial clk=0;
 always  #50 clk=~clk;


Initial
  #5000 a=1;
  #5000 $finish;
  // Simulation will run t=0 to 10,000.
```

Procedure Timing

Multiple Procedures

One may have many initials procedures all running in parallel. They all start at time t=0.

$finish

$finish ends the simulation. Without it one must manually abort the simulator.

10.•PROBLEMS:

```
initial
  begin
       aa=0;bb=0
   #50 aa=1;
   #50 aa=0;
   #50 $finish
  end
always @(aa)  bb=~bb;
```

a. How long will the simulation run?

b. When will the value of bb change?

Procedure Timing

Always

Must explicitly state when the procedure will be executed.

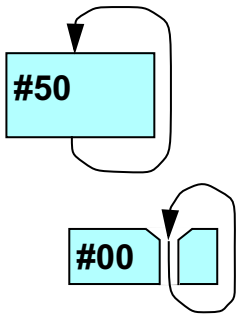
1. Always
   This runs all the time.
   When it finishes it starts again.

2. Always @(A or B)
   Runs when A or B change value.
   Used for combinational logic.

3. Always @(posedge clk)
   Runs after each rising clk
   Synthesizes to flip-flops.
   If you don't want FF don't use *posedge /negedge*.

4. Always @(clk)
   Runs when either edge of clk changes
   Can be made to give latches.

```
//Common test-bench clock.
 initial clk=0;
 always  #50 clk=~clk;
 // always reruns every 50 ns.
```




```
always  A=~A;
// Zero delay loop. Kills computer.

always @ (A or B)
  Y= A | B; // will synthesize to an
OR gate.

always @ (posedge clk)
  Q=d;
// a Flip-flop is implied to hold the old Q
// in between rising clock edges.

always @ (clk)
  if (clk) Q=d;
// implied else;
//      a latch is inserted to hold the old Q
```

Procedure Timing

Multiple Procedures

One may have many always procedures all running in parallel. Typically they start on the same clock edge.

always wait

always @(. . .) is commonly used. However
always wait can be useful in test benches. Also one can have several @ checks inside an always loop.

```
always @(a)
     #5 x= ...;
      @(b)
```

An alternate clock loop

This uses a forever loop which can only be used inside a procedure.

```
initial
    begin
       clk=0;
       forever #5 clk=~clk;
    end
```