Introduction to CMOS VLSI Design

Quick Verilog

Lecture by Peter Kogge
Fall 2009, 2011, 2012, 2015, 2018
University of Notre Dame
Using some slides by Jay Brockman Notre Dame 2008,
and David Harris, Harvey Mudd College
http://www.cmosvlsi.com/coursematerials.html

Levels of Design Modeling

- **Structural**: how a module is "wired together" from simpler modules
  - modules defined in terms of input and output **ports**
  - "Calling" a module creates a new instance
    - With "arguments" that are named wires.
- **Behavioral**: how outputs change as function of inputs
  - Key statement type: **assignment**
    - **Continuous** assignment – for driving a wire
    - **Always** assignment – for driving a latch
- **Physical**: layout of devices on chip
- **Netlist**: list of all ports connected to each wire

Verilog

- **Verilog**: one of two popular **Hardware Description Languages** (**HDL**)
  - VHDL is the other
- Naturally modular:
  - Define new module type in terms of simpler types
  - When used, creates a new "**instance**" of module
  - With wire names between "**ports**" of instance
- Suitable for all levels of models:
  - **Behavioral**: with no other resources needed
  - **Structural**: with library of technology-specific modules
  - **Physical**: as with structural but with "place & route" to
    - ***Place*** individual instances on 2D chip surface
    - ***Route*** wires between instance ports using chip metal

In Following

- `Courier` text = sample Verilog code
- `Green courier` = sample use of a keyword
- <*xxx*> = a general syntactic term

MIPS8 Overview

MIPS Architecture

- ❑ Example: subset of MIPS processor architecture
  - – Drawn from Patterson & Hennessy
- ❑ MIPS is a 32-bit architecture with 32 registers
  - – Consider 8-bit subset using 8-bit datapath
  - – Only implement 8 registers ($0 - $7)
  - – $0 hardwired to 00000000
  - – 8-bit program counter
- ❑ David Harris has developed labs to implement
  - – Uses Electric CAD tools
  - – Illustrate the key concepts in VLSI design

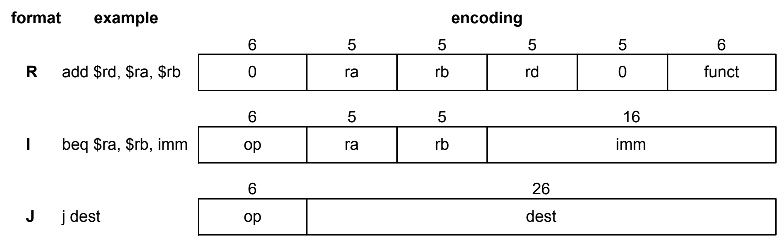
Instruction Set

Table 1.7 MIPS instruction set (subset supported)

Instruction	Function		Encoding	op	funct
`add $1, $2, $3`	addition:	$1 → $2 + $3	R	000000	100000
`sub $1, $2, $3`	subtraction:	$1 → $2 – $3	R	000000	100010
`and $1, $2, $3`	bitwise and:	$1 → $2 and $3	R	000000	100100
`or $1, $2, $3`	bitwise or:	$1 → $2 or $3	R	000000	100101
`slt $1, $2, $3`	set less than:	$1 → 1 if $2 < $3 $1 → 0 otherwise	R	000000	101010
`addi $1, $2,`	add immediate:	$1→ $2 + imm	I	001000	n/a
`beq $1, $2, imm`	branch if equal:	PC → PC + imm[a]	I	000100	n/a
`j destination`	jump:	PC_destination[a]	J	000010	n/a
`lb $1, imm($2)`	load byte:	$1 → mem[$2 + imm]	I	100000	n/a
`sb $1, imm($2)`	store byte:	mem[$2 + imm] → $1	I	110000	n/a

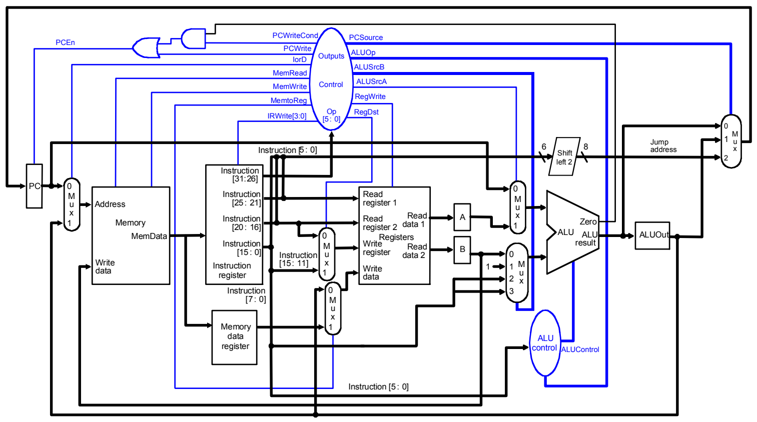
Instruction Encoding

- 32-bit instruction encoding
  - Requires four cycles to fetch on 8-bit datapath

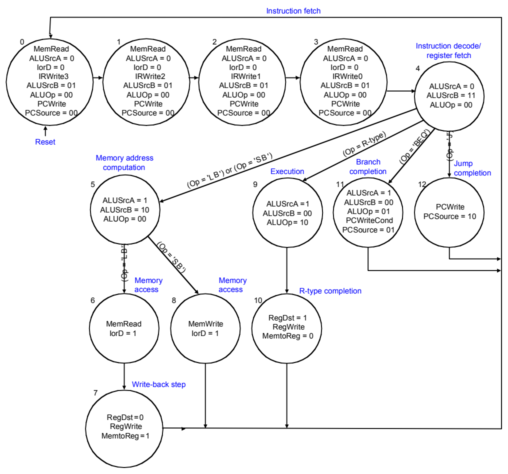


MIPS Microarchitecture

- Multicycle μarchitecture from Patterson & Hennessy

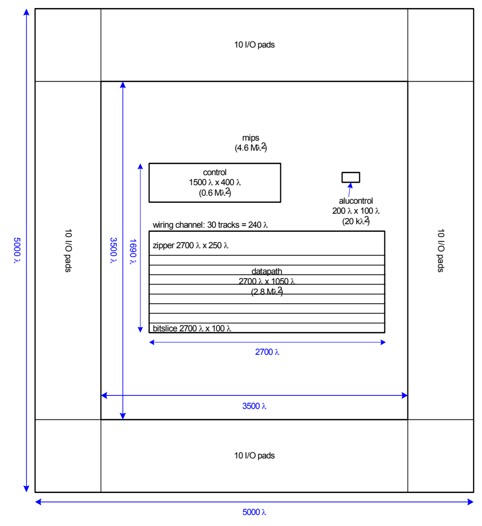


Multicycle Controller

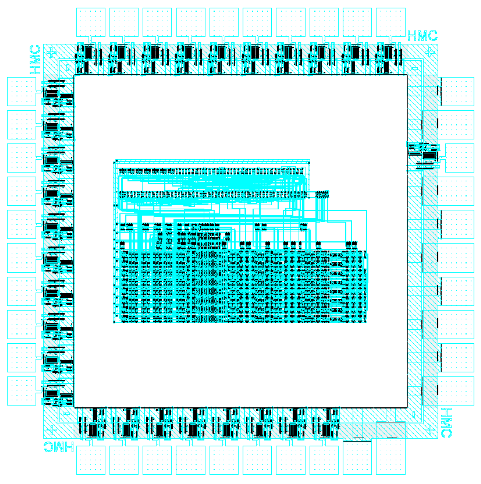


Instruction fetch
0 MemRead ALUSrcA = 0 IorD = 0 IRWrite3 ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
1 MemRead ALUSrcA = 0 IorD = 0 IRWrite2 ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
2 MemRead ALUSrcA = 0 IorD = 0 IRWrite1 ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
3 MemRead ALUSrcA = 0 IorD = 0 IRWrite0 ALUSrcB = 01 ALUOp = 00 PCWrite PCSource = 00
Reset
Instruction decode/ register fetch
4 ALUSrcA = 0 ALUSrcB = 11 ALUOp = 00
Memory address computation
(Op = 'LB') or (Op = 'SB')
(Op = R-type)
(Op = 'BEQ')
(Op = 'J')
5 ALUSrcA = 1 ALUSrcB = 10 ALUOp = 00
Execution
9 ALUSrcA =1 ALUSrcB = 00 ALUOp= 10
Branch completion
11 ALUSrcA = 1 ALUSrcB = 00 ALUOp = 01 PCWriteCond PCSource = 01
Jump completion
12 PCWrite PCSource = 10
(Op = 'LB')
(Op = 'SB')
Memory access
Memory access
6 MemRead IorD = 1
8 MemWrite IorD = 1
R-type completion
10 RegDst = 1 RegWrite MemtoReg = 0
Write-back step
7 RegDst=0 RegWrite MemtoReg=1

MIPS Floorplan

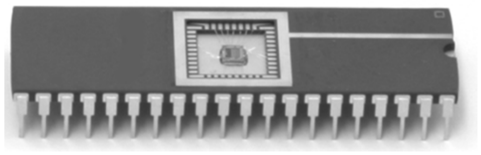


MIPS Layout



Fabrication & Packaging

- ❑ Tapeout final layout
- ❑ Fabrication
  - – 6, 8, 12” wafers
  - – Optimized for throughput, not latency (10 weeks!)
  - – Cut into individual dice
- ❑ Packaging
  - – Bond gold wires from die I/O pads to package



Modules, Structural Design, and Netlists

Signal Declarations

- `wire <list-of-signal-names>;`
  - Each name in list is a separate net
    - Treat as an electrical "wire"
    - No "stored value"
    - echoes whatever value is currently driving it to all other terminals with no time delay
- `reg <list-of-signal-names>;`
  - Each name treated like a "latch"
  - "Stores" value last "assigned" to it
- `parameter <name> = <value>`
  - Defines a constant value as in C `const`

Defining Modules in Verilog

- **`Module`**: defines a basic circuit with some input and output signals
- `module <module-name>(<port-list>);`
  <definitions of internal signals>;
  <assignments to internal signals and output ports>;
  `endmodule`
- `<port-list>:` list of circuit I/Os
  - Each port is a named "wire" from/to a "pad"
  - Input signals: **input** <*name*>,<*name*>….,
  - Output signals: **output** <*name*>,<*name*>…,
  - Signals that can be both: **inout** <*name*>,<*name*>…

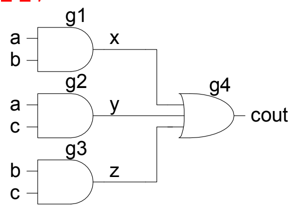
Example

defines new module type

port description

3 internal nets

```
module carry(input  a, b, c,
             output cout)

    wire   x, y, z;

    and g1(x, a, b);
    and g2(y, a, c);
    and g3(z, b, c);
    or  g4(cout, x, y, z);
endmodule
```



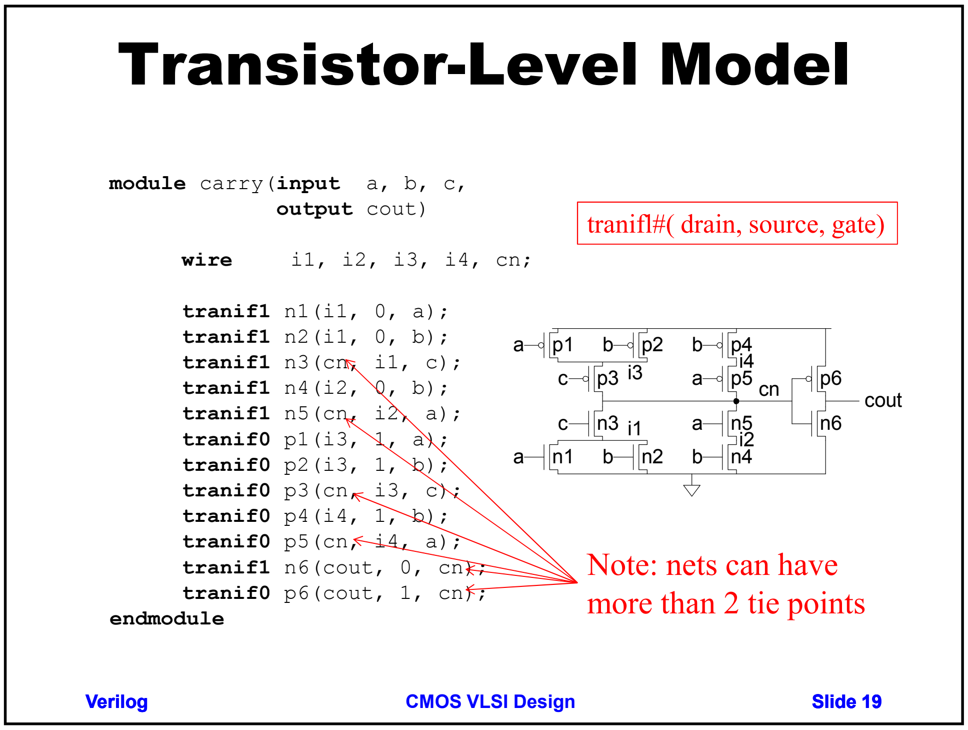
g1,…g4 instances of other module types

associating net names with ports builds internal netlist

Alternative Port List

```
module carry(a, b, c, cout)
    input a, b, c;
    output cout;

    wire   x, y, z;

    and g1(x, a, b);
    and g2(y, a, c);
    and g3(z, b, c);
    or  g4(cout, x, y, z);
endmodule
```

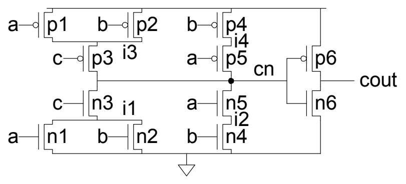
Sometimes easier to read



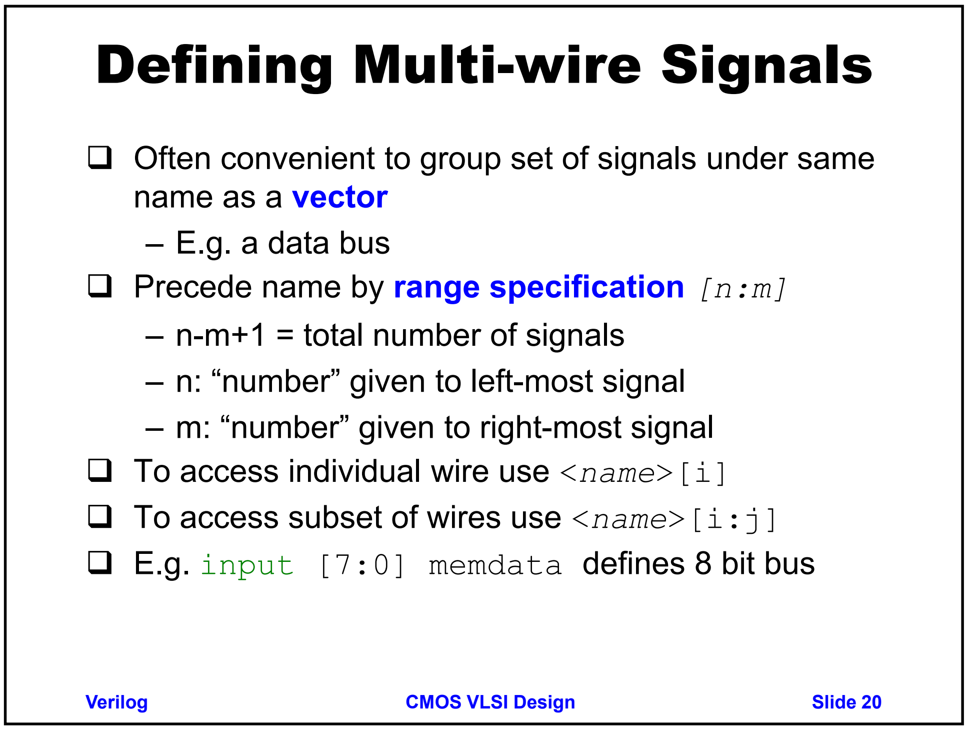
Transistor-Level Model

```
module carry(input  a, b, c,
             output cout)

      wire     i1, i2, i3, i4, cn;

      tranif1 n1(i1, 0, a);
      tranif1 n2(i1, 0, b);
      tranif1 n3(cn, i1, c);
      tranif1 n4(i2, 0, b);
      tranif1 n5(cn, i2, a);
      tranif0 p1(i3, 1, a);
      tranif0 p2(i3, 1, b);
      tranif0 p3(cn, i3, c);
      tranif0 p4(i4, 1, b);
      tranif0 p5(cn, i4, a);
      tranif1 n6(cout, 0, cn);
      tranif0 p6(cout, 1, cn);
endmodule
```

tranif1#(drain, source, gate)



Note: nets can have more than 2 tie points



Defining Multi-wire Signals

- ❑ Often convenient to group set of signals under same name as a **vector**
  - – E.g. a data bus
- ❑ Precede name by **range specification** `[n:m]`
  - – n-m+1 = total number of signals
  - – n: “number” given to left-most signal
  - – m: “number” given to right-most signal
- ❑ To access individual wire use `<name>[i]`
- ❑ To access subset of wires use `<name>[i:j]`
- ❑ E.g. `input [7:0] memdata` defines 8 bit bus

Example 4-input Ripple Adder

```
module four_bit_adder(input [3:0] a, b, input cin;
                      output [3:0] s, output cout);

      wire [3:1] c;

      fulladder fa0(a[0], b[0], cin,    s[0], c[1]);

      fulladder fa1(a[1], b[1], c[1],   s[1], c[2]);

      fulladder fa2(a[1], b[1], c[2],   s[2], c[3]);

      fulladder fa3(a[1], b[1], c[3],   s[3], cout);
endmodule;
```

Note: “order” of instance statements in code is irrelevant

Types of Nets in Verilog

- ❑ **Wire**: a normal metal or poly line
  - • driven by some single output port
  - • Replicates signal at arbitrary # of input ports
- ❑ **Tri**: a tri-stated wire
  - – May be driven by multiple tri-stated output ports
- ❑ **Supply0**: ground
- ❑ **Supply1**: V_{dd}
- ❑ Wand: wired AND
  - – May be driven my multiple open collector drivers
- ❑ Wor: wired OR
  - – May be driven my multiple emitter-coupled drivers
- ❑ Triand, trior: similar to above but may be three-stateTri0: resistor to ground
- ❑ Tri1: resistor to Vdd
- ❑ Trireg: models charge stored on a net

Verilog Value Types

- **Scalar**: (i.e. a single bit) may have values
  - 0: signal is driven to ground
  - 1: signal is driven high (typically to Vdd)
  - X: signal value is unknown or conflicted
  - Z: signal is “high impedance”
- **Sized numbers** written as ***<number>’<base><digits>***
  - <*number*> is the number of digits
  - <*base*> is the base for the number representation
    - B or b: binary
    - O or o: octal
    - H or h: hexadecimal
    - D or d: decimal
  - <*digits*>: string of digits in the specified base notation
- **Unsized numbers**: eliminate the <*number*>’

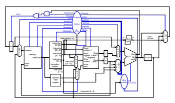
Value Examples

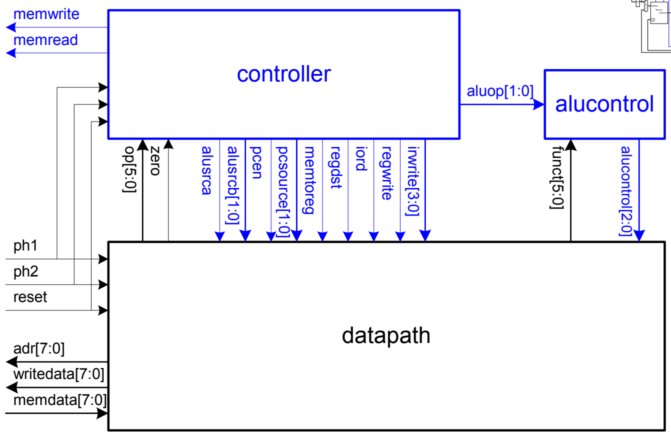
Table A.2 Constants

Number	# Bits	Base	Decimal Equivalent	Stored
`3’b101`	3	Binary	5	101
`‘b11`	unsized	Binary	3	000000..00011
`8’b11`	8	Binary	3	00000011
`8’b1010_1011`	8	Binary	171	10101011
`3’d6`	3	Decimal	6	110
`6’o42`	6	Octal	34	100010
`8’hAB`	8	Hexadecimal	171	10101011
`42`	unsized	Decimal	42	0000…00101010

from W&H

MIPS Block Diagram





High Level Verilog MIPS

```
// simplified MIPS processor
module mips #(parameter WIDTH = 8, REGBITS = 3)
             (input              clk, reset,
              input  [WIDTH-1:0] memdata,
              output             memread, memwrite,
              output [WIDTH-1:0] adr, writedata);

   wire [31:0]  instr;
   wire         zero, alusrca, memtoreg, iord, pcen, regwrite, regdst;
   wire [1:0]   aluop,pcsource,alusrcb;
   wire [3:0]   irwrite;
   wire [2:0]   alucont;

   controller   cont(clk, reset, instr[31:26], zero, memread, memwrite,
                     alusrca, memtoreg, iord, pcen, regwrite, regdst,
                     pcsource, alusrcb, aluop, irwrite);
   alucontrol   ac(aluop, instr[5:0], alucont);
   datapath     #(WIDTH, REGBITS)
                dp(clk, reset, memdata, alusrca, memtoreg, iord, pcen,
                   regwrite, regdst, pcsource, alusrcb, irwrite, alucont,
                   zero, instr, adr, writedata);
endmodule
```

Verilog Execution Model

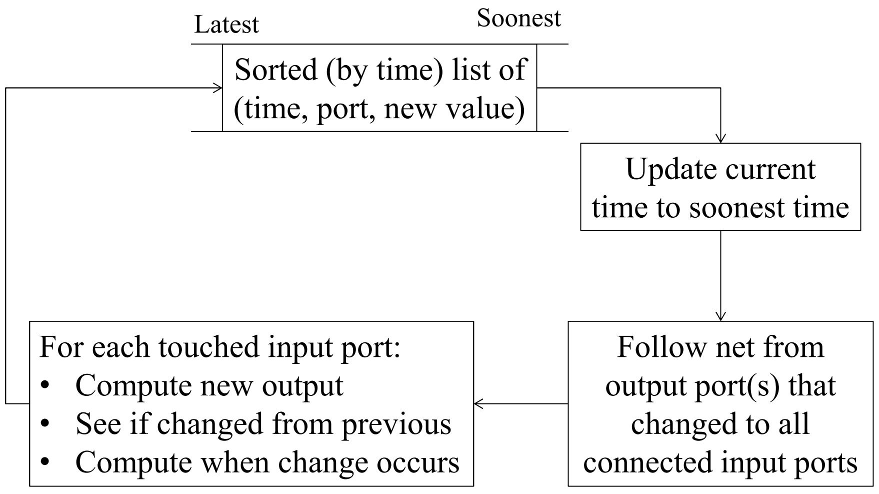
Verilog Execution Model (1)

- ❑ Module <u>definitions</u> define
  - – **Input** and **output ports** that can be wired to other module
  - – **Behavior**: What happens in any instance when actual inputs change
- ❑ Model <u>construction</u> defines multiple **instances** of modules
  - – How they are wired together
  - – What to do "at time 0"

Verilog Execution Model (2)

- At any point in time
  - Compute values for all expressions that have been "activated"
  - Tag with time when changes should happen
  - Sort updates by time
- To advance to next step
  - Find updates with smallest time – make that the "current time"
  - Apply all updates
  - See (via appropriate netlists) which circuits see changes in inputs
  - Compute new values, tag the updates, and repeat

Discrete Event Execution


Latest
Soonest
Sorted (by time) list of (time, port, new value)
Update current time to soonest time
Follow net from output port(s) that changed to all connected input ports
For each touched input port:
• Compute new output
• See if changed from previous
• Compute when change occurs

Verilog Behavioral Models & Expressions

Behavioral Models in Verilog

- ❑ Normal Purpose:
  - – Simplify design process for "simple" logic
    - *ESPECIALLY STATE MACHINES*
  - – Avoid doing detailed structural design early in the design cycle
- ❑ Behavioral models act just like structural
  - – Mixed model designs can be simulated simply
- ❑ Tools exist for **synthesizing** a structural design from behavior module

A Behavioral Module

```
module <module-name>(<port-list>);

<declarations>

assign #<delay> <net-name> = <expression>;

endmodule
```

repeated multiple times

Distributing Signals: Nets

- **Net**: changes value whenever circuit driving it changes
  - Carries a signal
  - Most common – a **wire**
- `assign` #<*delay*> <*net-name*> = <*expression*>;
  - ***Any change in expression changes value assigned to net***
  - <*expression*>: functional combination of constants and other signals
  - < *net-name*>: name of net that "tracks" value of expression
  - #<*delay*>: optional "delay" from time expression changes until time net signal changes

Operators in Expressions

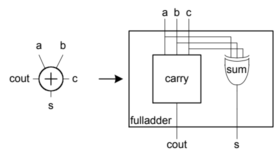
Table A.1 Operator Precedence

Symbol	Meaning	Precedence
~	NOT	Highest
*, /, %	MUL, DIV, MODULO	
+, -	PLUS, MINUS	
<<, >>,	Logical Left/Right Shift	
<<<, >>>	Arithmetic Left/Right Shift	
<, <=, >, >=	Relative Comparison	
==, !=	Equality Comparison	
&, ~&	AND, NAND	
^, ~^	XOR, XNOR	
|, ~|	OR, NOR	
?:	Conditional	Lowest

from W&H

Motivating Example

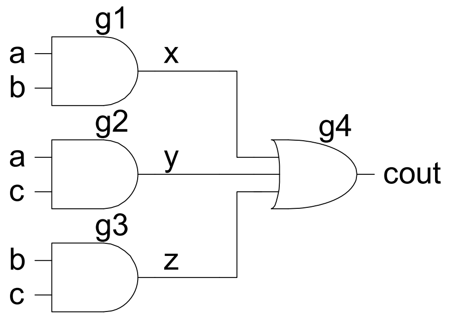
```
module fulladder(input  a, b, c,
                 output s, cout);

Xor2        sum_1(a, b, s1);
Xor2        sum_2(s1,c, s);
carry       c1(a, b, c, cout);
endmodule

module carry(input  a, b, c,
             output cout)

assign cout = (a&b) | (a&c) | (b&c);
endmodule
```



Which is:
- Structural?
- Behavioral?

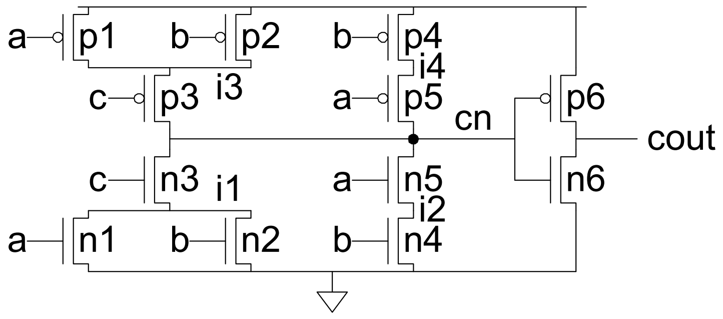
Example: Carry Logic – Standard Cell Gates

- **assign** `cout = (a&b) | (a&c) | (b&c);`



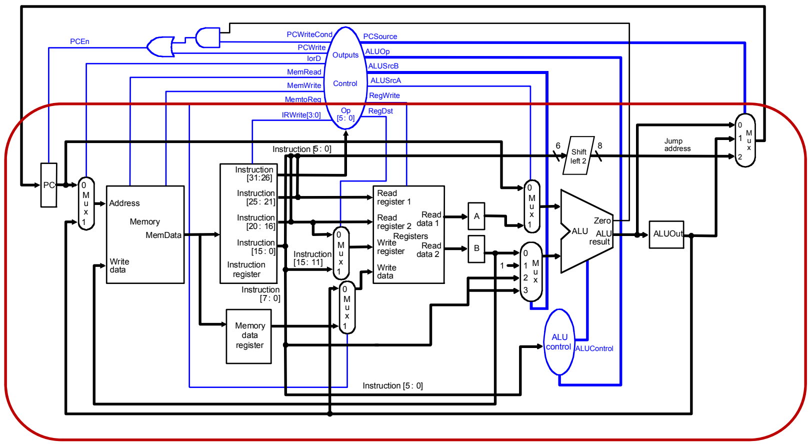
Example: Carry Logic - Transistors

- **assign** `cout = (a&b) | (a&c) | (b&c);`



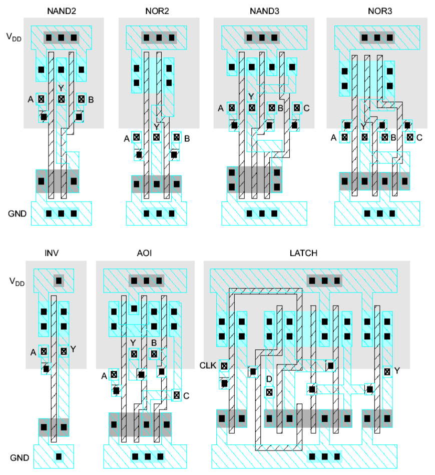
MIPS Datapath

- Multicycle μarchitecture from Patterson & Hennessy



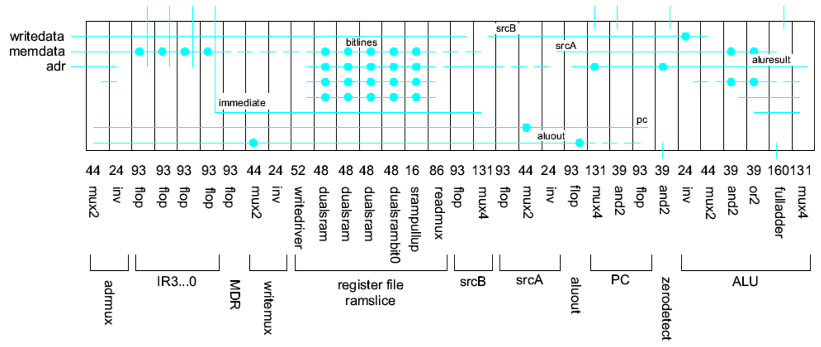
Standard Cell Library

- Uniform cell height
- Uniform well height
- M1 V_{DD} and GND rails
- M2 Access to I/Os
- Well / substrate taps
- Exploits regularity



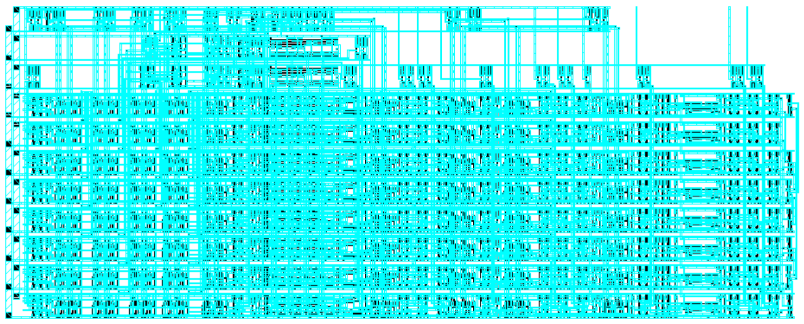
Slice Plans

- Slice plan for bitslice
  - Cell ordering, dimensions, wiring tracks
  - Arrange cells for wiring locality



MIPS Datapath

- 8-bit datapath built from 8 bitslices (regularity)
- Zipper at top drives control signals to datapath



MIPS Verilog DataPath (1)

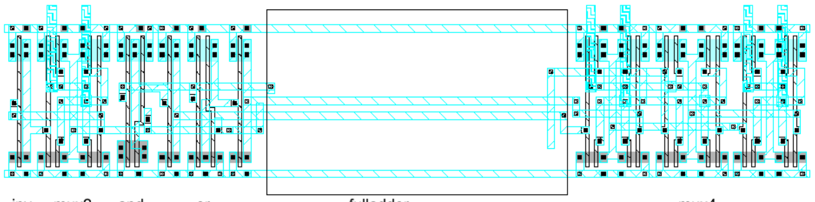
```
module datapath #(parameter WIDTH = 8, REGBITS = 3)
                 (input              clk, reset,
                  input  [WIDTH-1:0] memdata,
                  input                        alusrca, memtoreg, iord, pcen,
regwrite, regdst,
                  input  [1:0]       pcsource, alusrcb,
                  input  [3:0]       irwrite,
                  input  [2:0]       alucont,
                  output             zero,
                  output [31:0]      instr,
                  output [WIDTH-1:0] adr, writedata);

   // size of the parameters must be changed to match the WIDTH parameter
   parameter CONST_ZERO = 8'b0;
   parameter CONST_ONE =  8'b1;

   wire [REGBITS-1:0] ra1, ra2, wa;
   wire [WIDTH-1:0] pc, nextpc, md, rd1, rd2, wd, a, src1, src2, aluresult,
                       aluout, constx4;

   // shift left constant field by 2
   assign constx4 = {instr[WIDTH-3:0],2'b00};

   // register file address fields
   assign ra1 = instr[REGBITS+20:21];
   assign ra2 = instr[REGBITS+15:16];
   mux2        #(REGBITS) regmux(instr[REGBITS+15:16], instr[REGBITS+10:11],
regdst, wa);
```

MIPS Verilog Data Path (2)

```
   // independent of bit width, load instruction into four 8-bit registers
over four cycles
   flopen      #(8)      ir0(clk, irwrite[0], memdata[7:0], instr[7:0]);
   flopen      #(8)      ir1(clk, irwrite[1], memdata[7:0], instr[15:8]);
   flopen      #(8)      ir2(clk, irwrite[2], memdata[7:0], instr[23:16]);
   flopen      #(8)      ir3(clk, irwrite[3], memdata[7:0], instr[31:24]);

   // datapath
   flopenr     #(WIDTH)  pcreg(clk, reset, pcen, nextpc, pc);
   flop        #(WIDTH)  mdr(clk, memdata, md);
   flop        #(WIDTH)  areg(clk, rd1, a);
   flop        #(WIDTH)  wrd(clk, rd2, writedata);
   flop        #(WIDTH)  res(clk, aluresult, aluout);
   mux2        #(WIDTH)  adrmux(pc, aluout, iord, adr);
   mux2        #(WIDTH)  src1mux(pc, a, alusrca, src1);
   mux4        #(WIDTH)  src2mux(writedata, CONST_ONE, instr[WIDTH-1:0],
                                 constx4, alusrcb, src2);
   mux4            #(WIDTH)   pcmux(aluresult, aluout, constx4, CONST_ZERO,
pcsource, nextpc);
   mux2        #(WIDTH)  wdmux(aluout, md, memtoreg, wd);
   regfile       #(WIDTH,REGBITS) rf(clk, regwrite, ra1, ra2, wa, wd, rd1,
rd2);
   alu         #(WIDTH) alunit(src1, src2, alucont, aluresult);
   zerodetect #(WIDTH) zd(aluresult, zero);
endmodule
```

MIPS ALU

- Arithmetic / Logic Unit is part of bitslice



Behavioral ALU

```
module alu #(parameter WIDTH = 8)
            (input      [WIDTH-1:0] a, b,
             input      [2:0]       alucont,
             output reg [WIDTH-1:0] result);

   wire      [WIDTH-1:0] b2, sum, slt;

   assign b2 = alucont[2] ? ~b:b;
   assign sum = a + b2 + alucont[2];
   // slt should be 1 if most significant bit of sum is 1
   assign slt = sum[WIDTH-1];

   always@(*)
      case(alucont[1:0])
         2'b00: result <= a & b;
         2'b01: result <= a | b;
         2'b10: result <= sum;
         2'b11: result <= slt;
      endcase
endmodule
```

Verilog Procedural Assignments

Continuous Assignments

- `assign` statements are "continuous"
- Written once, executed "continuously"
- Any changes in right hand side immediately reflected in left-hand signal
  - Delayed in time perhaps by **`delay`**
- If multiple assigns in a block,
  - execution is *order independent*

Procedural Assignments

- ❑ Sometime we want to execute statements **procedurally** (like a C program), i.e.
  - – Start statements at some point in time
  - – And then in some order
  - – Especially important for controlling registers
- ❑ Two kinds of such behavior
  - – **Single pass behavior** is executed only once
    - • At initialization
  - – **Cyclic behavior** is triggered whenever "something" happens
    - • Where "something" can be defined in advance
- ❑ Need **registers** to capture values between changes

Registers

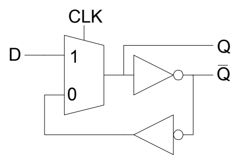
- ❑ Verilog **registers** much like C variables
  - – Once assigned a value, they keep it
  - – Until another value is assigned
- ❑ Defined by: **`reg`** *`[<msb>:<lsb>] <reg-name`*>;
  - – If no [:] then default size is 1 bit
  - – Initial value is "x"
- ❑ Typically assignments tied into events like clocks

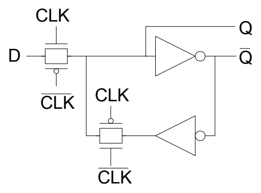
Procedural Assignments

<reg-name> = *<expression>*;

- Note: no "`assign`"
- When statement executed, register takes on value from expression
  - Actually an update event is scheduled
- Update event is done BEFORE next statement in program order

A Level-Sensitive D Latch





```
module D_latch(input D, CLK, output Q);

Q = CLK ? D : Q;

endmodule
```

A D Latch with a Reset

```
module D_latch_wReset(input D, CLK, Reset,
                      output Q);

Q = Reset ? 0 (CLK ? D : Q);

endmodule
```

Program Blocks

- Often more than 1 statement needs to be executed in a time-sensitive assignment
- Such statements grouped into **blocks**
  - <*statement*>; … <*statement*>;
- **`begin`** <*statement-block*> **`end`**;
  - All statements in block are executed in order
    - i.e. one at a time top to bottom
  - Assignment statements within block are said to be **blocking assignments**
    - Next one cannot start until last completes
- **`fork`** <*statement-block*> **`join`**;
  - All statements in block are executed in parallel
    - i.e. in any order

Single Execution Behavior

- Execute *only* at specific time, i.e. time=0
- Typically used by test benches for startup

    initial <statement>;

- Example:

```
initial begin
  Reset = 0;
  #10 Reset = 1;
  #5 Reset =0;
end
```

What is the waveform?

Controlling Cyclic Behavior

- Statement to be executed at specific times
  - E.g. at clock pulses

  always @(<*sensitivity list*>) <*statement*>
- @ is optional **event control operator**
- <*sensitivity-list*> is list of signal names
  - @* means use all nets & variables read by procedure's statements
- Whenever any one of signals in sensitivity-list changes, <*statement*> is executed
  - And only then
- What kind of change is immaterial, unless signal name preceded by:
  - **posedge**: execute only on a 0 to 1
  - **negedge**: execute only on a 1 to 0

Non-blocking Assignments

- ❑ Use of "<=" instead of "=" in assignment makes statement execute *concurrently* with later statements

  <register> <= *<expression>*;

  – left-hand side MUST be a register
- ❑ Order of such assignments is immaterial;
  – They ALL HAPPEN AT ONCE!
- ❑ Better match to real circuits

```
always @(posedge clock)
  begin
  reg1 <= expr1;
  reg2 <= expr2;
  …
  end
```

All expressions evaluated <u>at same time;</u> then all assignments happen <u>at same time</u>

Example: Clock Generation

```
Reg clk;

always
      begin
          clk <= 1;
          # 5;
          clk <= 0;
          # 5;
      end
```

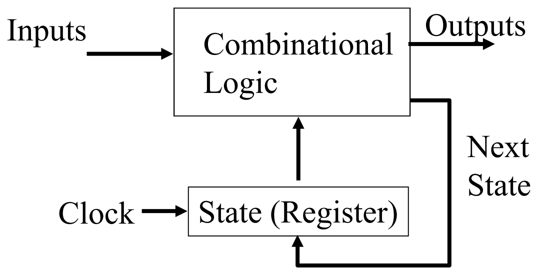
schedules a change to clk "now"

schedules a change to clk in 5 time units

Assignment Summary

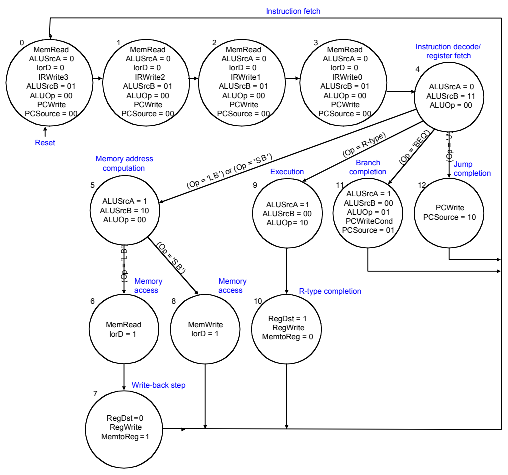
- Continuous assignment:
  - `assign #<delay> <net-name> = <expression>`
- Procedural assignment:
  - *<register>* = *<expression>*;
- Non-blocking assignment:
  - *<register>* **<=** *<expression>*;

Finite State Machines



```
module FSM(input clock, In1, …Inn, output O1, …Om);

reg state, next_state;

always @(posedge clock)
  begin
   state <= next_state;
   …
   Code that does something like next_state <= <expression>;
   …
  end
```

Multicycle Controller



MIPS VERILOG CONTROL LOGIC

```
module controller(input clk, reset,
                  input      [5:0] op,
                  input            zero,
                  output reg memread, memwrite, alusrca, memtoreg, iord,
                  output           pcen,
                  output reg       regwrite, regdst,
                  output reg [1:0] pcsource, alusrcb, aluop,
                  output reg [3:0] irwrite);
```

- ❑ Next: list of constants to define machine states
- ❑ Then an “always” sensitive to rising edge of clock
  - – Embedded case statement
  - – One case per state
  - – Determines what new state should be at end of this clock
- ❑ Then an “always” sensitive to any change in values
  - – Embedded case statement
  - – Each case specifies control signals to dataflow & memory

More Syntax

Memory

reg <*word-size*> <*memory-name*> <*mem-size*>…<*mem-size*>;

- <*word-size*> is `[`<*msb*>:<*lsb*>`]`
- <mem-size> is `[`<*low-index*>:<*high-index*>`]`
- Example: define register d_array as 2D array of 16 bit words

  `reg [15:0] d_array [0:127][0:255]`
- Selecting a word from an array:
  - Assume a_byte is an 8 bit set of wires

    `a_byte = d_array[64][32][12:5];`
  - takes 8 bits from 32nd column of 64th row

Other Verilog Constructs

- **Comments**:
  - **Single line**: // …
  - **Block**: /* …. */
- **Identifiers:**
  - start with letter or _
  - follow by letters, digits, _, or $
  - case sensitive

If Statement

```
if (<expression>)

        <then_statement>

else <else_statement>

end
```

- else is optional
- nesting permitted, as in else if ….

Case Statement

```
case (<expression>)
        <case_item>: <case_item_statement>
        …
        default: <case_item_statement>
endcase
```

- ❑ <case_item> is a constant that is matched against that from the <*expression*>)
  - – If match, then do the <*case_item_statement*>
    - • And exit case statement after completion
  - – If not, try next case
- ❑ `default` is optional for no matches

For Statement

```
for (<initial_assignment>;
       <condition-expression>;
        <update_statement>)
begin
<loop-statements>
end
```

- ❑ <*initial_assignment*> and <*update_statement*> are some form of assignment
- ❑ All iterations of <*loop-statements*> conceptually done AT SAME TIME

Other Loops

```
repeat (<expression>)
begin
<loop-statements>
end
```

```
while (<condition>)
begin
<loop-statements>
end
```

`wait` (<*expression*>) <*statement*>

- ❑ All iterations of <*loop-statements*> conceptually done AT SAME TIME

Timescale

- ❑ `` `timescale `` <*time_unit*>/<*time_precision*>
  - <*time_unit*> and <*time_precision*> of form:
    - <*number*> <*space*> <*unit*>
    - with <*unit*> as `s ms us ns ps fs`
- ❑ From this point on:
  - for delays: 1 time unit = <*time_unit*>
  - for precision of internal time calculations: use <*time_precision*>

Module Instantiation Options

- Previously

  <module-type> <instance-name> (<*list-of-signal-names*>);
- In place of <*list-of-signal-names*>, use list of

  . <*port-name*>(<*net-name*>)
- Optionally add #(<parameter_assignment_list>) after <*module-type*>
  - Associate values with parameters within module definition
  - Overwrites `parameter` <name> = <value> inside definition

More
(See Quick Reference Card)

- Builtin modules: and, or, nand, nor, xor,xnor
  - One output (first argument)
  - One or more inputs (rest oft arguments)
- `$display` statement: verilog eqvt of printf
  - `$display` (<format string>);
    - (" … text …", <list of var-names>)
      - %<format-code> in text says how to format "next" var in list
      - `$time` is current time
  - executed as any other statement in a block
  - references variable values AS THEY ARE THEN

see http://www.hpcc.ecs.soton.ac.uk/hpci/tools/vlogref.pdf

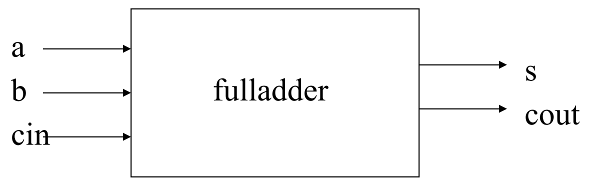
More Examples

Typical Design

- ❑ File with list of module definitions
- ❑ Typically first one is a **testbench**
  - – Glues together other pieces in a way that allows appropriate operation to be demonstrated
  - – Includes `initial` code to start up & reset system
  - – Includes code to generate clocks & other common control signals

```
module TestBench(); \\ no ports
reg <in-list>; \\ all input ports to DUT
wire <out-list>; \\ all output ports from DUT

module Device_Under_Test(input <in-list>, output <outlist>);
    …
endmodule;

initial begin
    code to change values of regs in <in-list>
end;

endmodule;
```

A Behavioral Full Adder



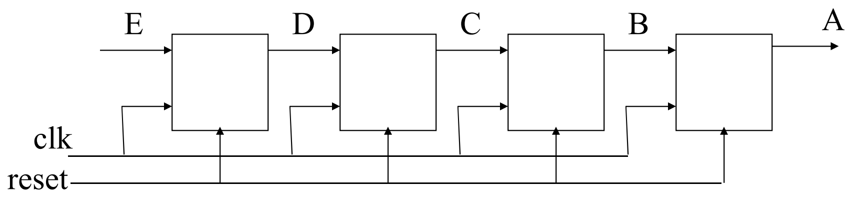
```
module fulladder(input a, b, cin, output s, cout);
        wire prop;
        assign prop = a ^ b;
        assign s = prop ^ cin;
        assign cout = ( a & b) | (cin & (a | b);
endmodule
```

AND

OR

Note: “order” of *assigns* in code is irrelevant

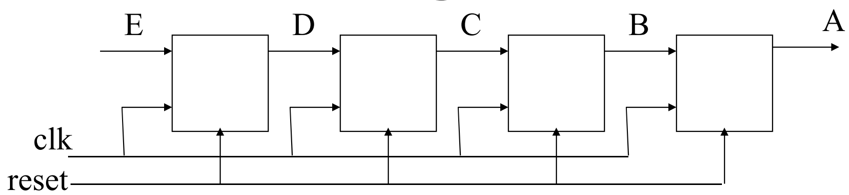
Example: A 4 bit Shift Register



```
Module shiftreg(input E, clk, reset,
                output D, C, B, A);
        reg A, B, C, D; // note these output port defined to be regs
        always @(posedge clk or posedge reset)
        begin
        if (reset) begin A=0;B=0;C=0;D=0:E=0; end
        else begin
                A = B;
                B = C;
                C = D;
                D = E;
                end
        end
```

What happens if we reorder these?

Repeat Shift Register Example:

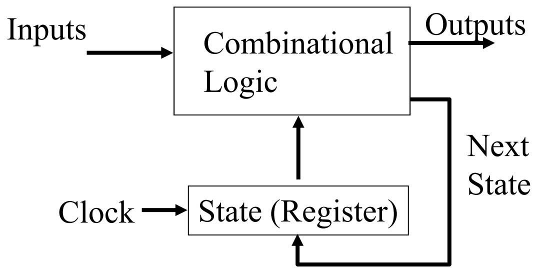


```
Module shiftreg(input E, clk, reset,
                output reg D, C, B, A);
        always @(posedge clk or posedge reset)
        begin
        if (reset) begin A<=0;B<=0;C<=0;D<=0:E<=0; end
        else begin
                A <= B;
                B <= C;
                C <= D;
                D <= E;
                end
        end
```

Note simplified syntax

Now what happens if we reorder these?

Finite State Machines



```
module FSM(input clock, In1, …Inn, output O1, …Om);
reg state, next_state;
parameter … statei = 'bxxxx;
always @(posedge clock)
begin
state <= next_state;
case (state)
        …
        statei: begin next_state<=statej; Ok <= … end;
        …
end
```