PROCESSOR DESIGN AND DATAPATH DESIGN IN VERILOG

Anurag Pallaprolu
2012B5A3405P
SK 207, BITS Pilani

April 8, 2014

Contents

1 Introduction

This document deals with the different types of processor designs and how the respective datapaths can be simulated in Verilog. It is clasiified into components needed to build a datapath for a relatively simple MIPS instruction.

Each component will be described and a sample Verilog implementation is displayed. It might not be the most efficient design but the implementation is assured to work. Then, I would look at designing processors in three basic types of design.

Single Cycle

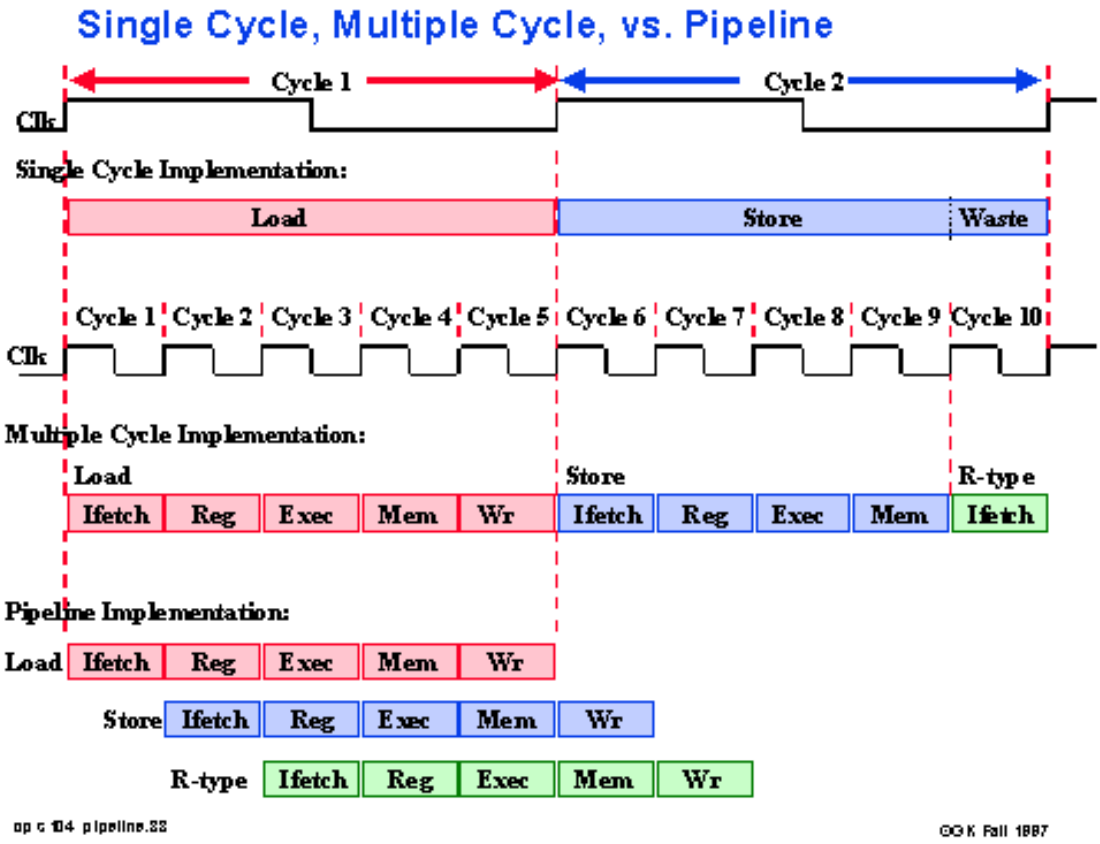
This design basically tries running the entire instruction in one cycle always. This is the chronologically first type of processors that were made. It is inefficient for the obvious reason that, for the entire instruction to run in one cycle, the cycle length should be larger than the slowest instruction. Hence, the computation efficiency will also go by the same proportion.

Multi Cycle

As the name suggests, multicycle or multiple cycle implementation essentially implements each of the sub-implementations in their respective cycles. It still appears as if this still has the "cycle time greater than the time of slowest instruction" problem but care must be taken as the instruction that we are talking about here is the subimplementation of the actual instruction. So, it is not an actual disadvantage but the cycle time is assured to be shorter than the single cycle design.

Pipelined

This is a radical shift from the above two types. It essentially implements an assembly line type of processing the sub-implementations of the given instruction. But, this time, the cycle length depends on the number of so called "pipelines". I shall explain more about the associated terminology later. But I suppose the picture below should suffice.



In the upcoming sections, I would review my basic digital devices to build a datapath in verilog. But first, something traditional related to Verilog.

2 The Language: Testbenches

Any Verilog code written will have to be tested on a given number of input values. The place where you (virtually) go out to test your device is called the Testbench. In this case, it is a Verilog file with a module assigned to it. As an example, take a look at the Verilog code for a D Flip Flop and the Testbench for the corresponding device. The details of the specific device will be dealt with later.

The Module

```
module dff(b,clk,q,q_0);
```

```
        input b, clk;
        output q,q_0;
        reg q;
        reg q_0;

        always@(posedge clk)
        begin
                q <= b;
                q_0 <= !b;
        end
endmodule
```

The Testbench

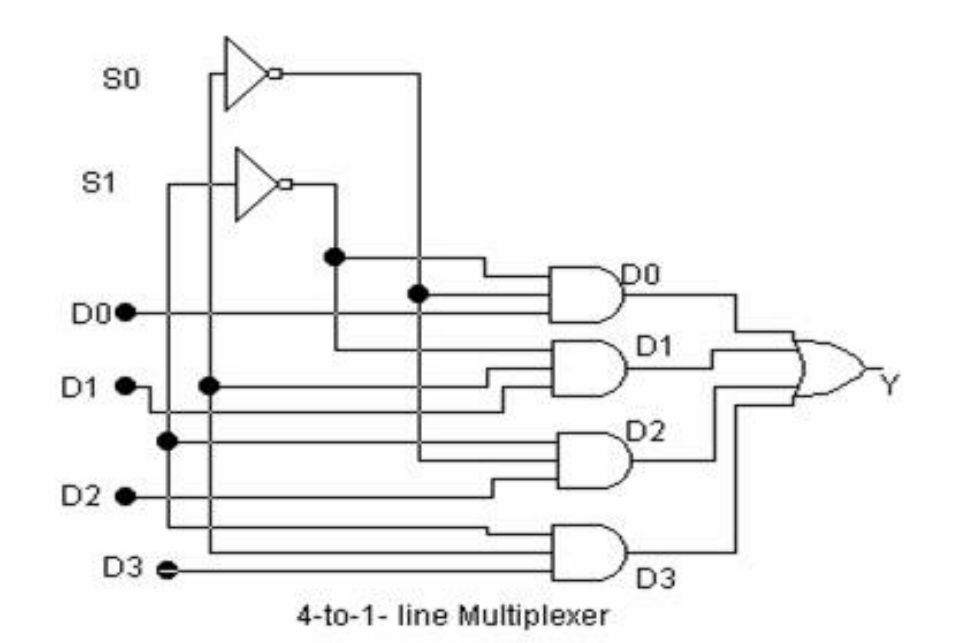
```
module dff_tb;

reg clock, reset, d;
wire q, q_0;

initial begin
        $dumpfile("dff_tb.vcd");
        $dumpvars(1,dff_tb);
        $monitor("clk=%b, d=%b,q=%b,q_0=%b", clock, d, q, q_0);
        clock = 0;
        d = 1;
        #10 d = 0;
        #12 d = 1;
        #18 d = 0;
        #20 $finish;
end

always
        begin #5 clock = !clock;
end

dff d0( .b(d), .clk(clock), .q(q), .q_0(q_0));

endmodule
```

So it is quite clear to see what a Testbench does. It sets up a loop of values which are ran on the given device and its output is recorded. Note: The commands $dumpfile()$ and $dumpvars()$ are NOT related to Verilog. They are parameters for viewing the wave diagrams and are needed for waveform viewers like GTKWave etc. If you are using a standard verilog compiler like $iVerilog$ then the dump file can be created easily by the following command $> iverilog - o * enterexecname * *entertbname * *entersrcname * .$

3 The Components

3.1 Multiplexor

The multiplexor or more commonly known as a "mux" is a basic digital circuit which essentially lets the user choose one input from the given array of several inputs. This is implemented quite easily in a behavioral manner in Verilog, but the circuit is also quite simple. The multiplexor is a very versatile device when it comes to decision making situations in circuits quite easy to implement. The schematic is given below and then the Verilog code(s) to implement this design.



The essential idea is to take in the values of the select lines and allow the corresponding data lines through. The behavioral Verilog code is given

below.

```
module mux(select,d,q);
        input[1:0] select; //Select wire
        input[3:0] d; //array of select choices
        output q;//out

        wire q;
        wire[1:0] select;
        wire[3:0] d;
        assign q = d[select];//give the user what he needs!
endmodule
```

The Verilog code for the gate level design schematic shown before is given below.

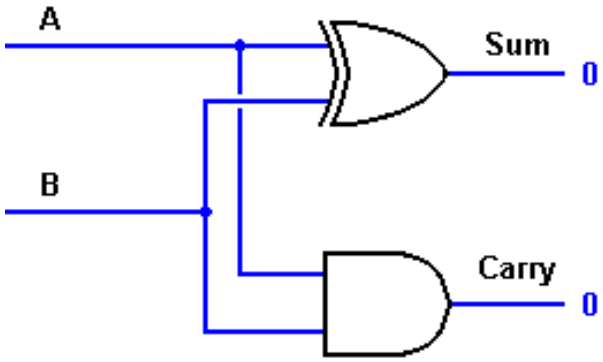
```
module mux7( select, d, q );

input[1:0] select;
input[3:0] d;
output     q;

wire       q, q1, q2, q3, q4, NOTselect0, NOTselect1;
wire[1:0]  select;
wire[3:0]  d;

not n1( NOTselect0, select[0] );
not n2( NOTselect1, select[1] );

and a1( q1, NOTselect0, NOTselect1, d[0]  );
and a2( q2,  select[0], NOTselect1, d[1]  );
and a3( q3, NOTselect0,  select[1], d[2]  );
and a4( q4,  select[0],  select[1], d[3]  );

or o1( q, q1, q2, q3, q4 );

endmodule
```

3.2 Adders and Subtractors: The basic Arithmetic Logic Unit

An Arithmetic Logic Unit(ALU) has the basic function of performing arithmetic on given input bits. The type of operations to be done are specified by the appropriate control lines. The most basic of arithmetic operations that can be done are addition and subtraction. Devices which perform such tasks are called "adders". Another control line to the adder(called the "binvert")can be summoned to make it sign-invert one of the bits and then add them, a process equivalent to subtraction.

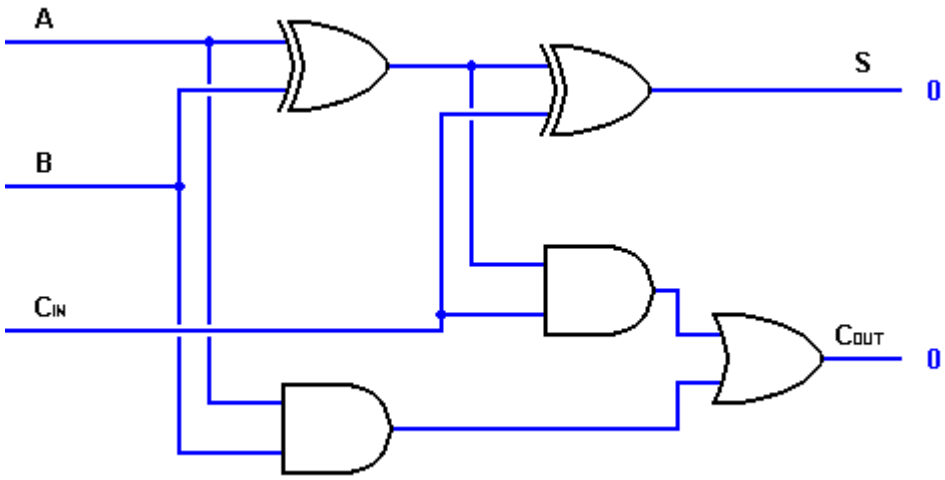
There are essentially two types of adders. Ones which cannot accomodate for overflow(sometimes also known as carry out/carry in) are called as half-adders. Adders which have three input pins and two output pins two extra for the two overflow pins are called full adders.

The design for a half adder is given below.




```
module ha(a,b,sum,carry);
        input a,b;
        output sum,carry;

        assign sum = (a&(~b))|((~a)&(b));
        assign carry = a&b;
endmodule
```

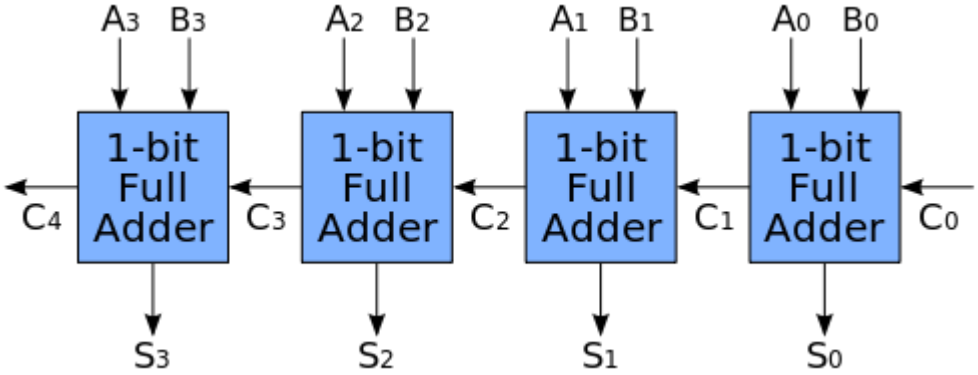
The design for the full adder is given in the next page. Notice the slight similarity in the first stage of arrangement.




```
module fa(a,b,cin,sum,cout);
        input a,b,cin;
        output sum, cout;

        assign sum = cin ^ a ^ b;
        assign cout = ~cin & a & b| cin & (a|b);
```

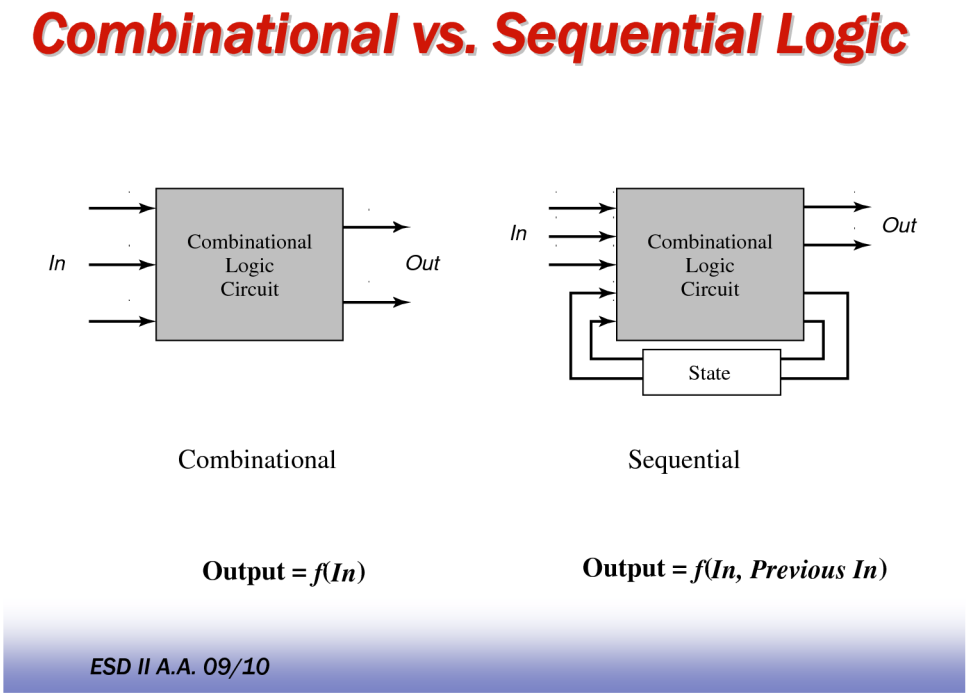
endmodule

Making a subtractor need not be dealt in detail because the only change in both the circuits is just at inverting a signal(if its a one bit scenario then by a not gate). I can now extend the number of input bits required by the adder. The schematic for the "ripple-carry adder" is given below.

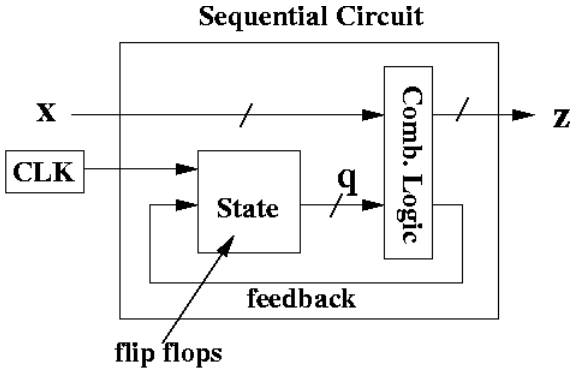


3.3 Latches, Flip Flops and Registers: Sequential Digital Devices

In the entirety of the digital devices that we will see, we can divide them into two categories: Combinational and Sequential Devices. The diagram given below should make things clear regarding the differences between combinational and sequential circuits.



To repeat, combinational circuits do not involve any sort of storage or memory of sorts, given a live data it works with it on the spot and shoots out the result. A sequential circuit involves some sort of a memory of the given data, the output depends recursively on previous inputs. A combinational logic is neccesary for a sequential circuit to exist. Refer to the figure in the next page.



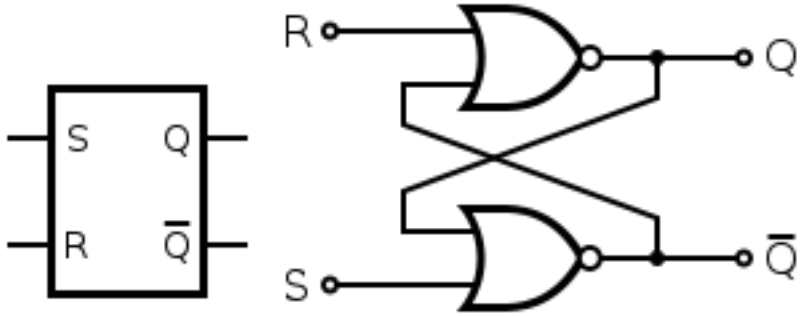
Let us start our memory design section with the simplest of all devices, the latch.

3.3.1 Latch

As the name suggests a latch is a single bit storage device which has two states. In more technical language, it is said to be a bistable multivibrator, which means, it has only two stable existant states. Latches can be memory devices, and can store one bit of data for as long as the device is on. As the name suggests, latches are used to "latch onto" information and hold in place. Latches are very similar to flip-flops, but are not synchronous devices, and do not have a visible clocking methodology. There are two basic types of latches manufactured.

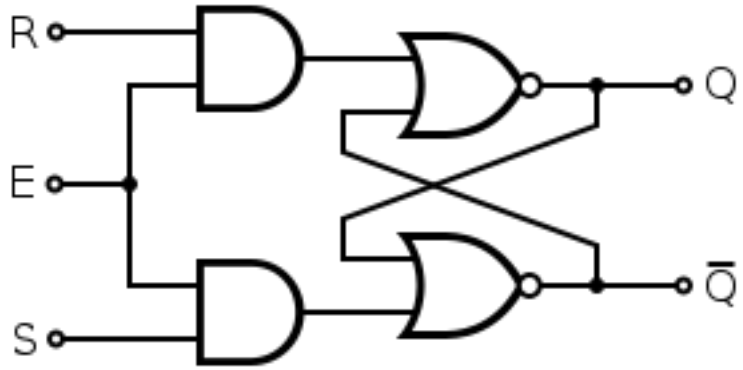
SR Latch: The Set/Reset Latch

An SR Latch is an asynchronous i.e, non clocked device. It works independently of control signals and relies only on the state of the S and R inputs. The schematics for the SR Latch are given below. Starting from the block diagram to the gate level implementation are presented.



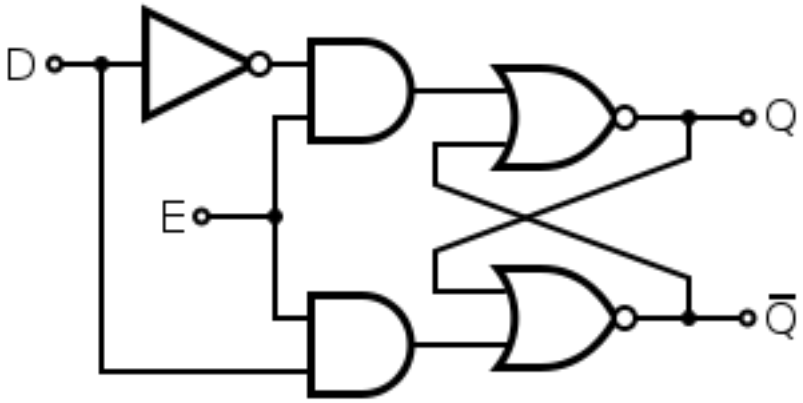
Gated SR Latch

In simple words, the Gated SR latch is a slight improvement, more rather a modification in which the latch's working condition i.e, its control could be enabled or disabled as per usage. There is an extra input port for enabling and disabling the latch. The schematic is given below.



D Latch

The D-Latch is an abbreviation of Data Latch. It is a more data secured kind of a gated SR Latch. There are two input pins, one for data and other for enabling the latch. There are no more set and reset pins. The schematic is as follows:



The Verilog code for the corresponding types of latches is given below.

D Latch

```
module dlatchmod(e, d, q);
```

```
        input e;
        input d;
        output q;
        reg q;
                always @(e or d) begin
                        if (e)
                                q<=d;
                end

endmodule
```

Gated S-R Latch

```
module gatedsr(r,e,s,q,q0);
        input r,e,s;
        output q, q0;
        wire w,x;
        assign w = r&e;
        assign x = s&e;
        nor(q,w,q0);
        nor(q0,q,x);
endmodule
```

3.3.2 Flip Flops

Flip flops are traditional to memory devices. They can be thought of simply as one bit registers. In the very essence, a flip flop is similar to a latch, with the only stark difference being the fact that Flip Flops have a kind of a feedback mechanism which helps them hold onto the assigned value. Changes in data stored will be visible only at positive edges of clocks. I have mentioned before a type of a flip flop(the DFF) at the beginning while talking about testbenches. I would explain two more types of flip flops. First the schematics and then the Verilog.

SR Flip Flop

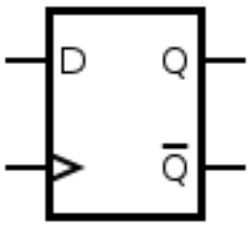
An SR(Set/Reset) flip-flop is perhaps the simplest flip-flop, and is very similar to the SR latch, other than for the fact that it only transitions on clock edges. SR flip-flops are extremely uncommon because they retain the illegal state when both S and R are asserted. Generally

when people refer to SR flip-flops, they mean SR latches. The Verilog code is easy to understand.

```
module srcase(s,r,clk,q,qb);
     input s,r;
     input clk;
     output q,qb;
reg q,qb;
initial
  begin
        q=1'b0;
    q=1'b0;
 end
always@(posedge clk)
begin
     case({s,r})
     {1'b0,1'b0}: begin q=q; qb=qb; end
   {1'b1,1'b0}: begin q=1'b1; qb=0; end
   {1'b0,1'b1}: begin q=0; qb=1'b1; end
   default : begin q="x";qb="x"; end
   endcase
 end
endmodule
```

D Flip Flop

A D Flip Flop is the most common kind of flip flop which is used mostly to make registers and register files. They are the edge triggered variant of the transparent latch. It has a standard symbol given below.



The Verilog code for the DFF has been given in section 2 itself. Refer it for the testbench too.

JK Flip Flop

The JK flip flop is just an advanced SR type flip flop. The only improvement is in terms of the stability. Note the fact that, when

both the inputs of the SR flip flop are 1, then the device goes into a metastable state, whereas in a JK flip flop it does the job of inverting the signal. Refer to the table given below for clarity. The first table is about SR flip flops

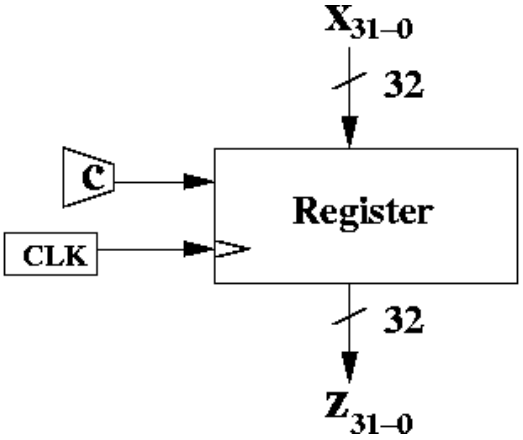
S	R	Q	Comment
0	0	0	Hold State
0	1	0	Reset
1	0	1	Set
1	1	Meta	stable

The table below is for JK flip flops. Q* refers to the complement of Q.

J	K	Q	Comment
0	0	Q(previous)	Hold
0	1	0	Reset
1	1	$Q*$	Toggle

3.3.3 Registers

I had mentioned before that D flip flops can be used to create register memories. Now, we shall actually build a 32 bit register using 32 individual flip flops. You might think that synchronization of those many bits of data would be difficult, but it is all managed thanks to the simultaneous clock line that runs through all of them. The schematic and the Verilog implementation are given below.

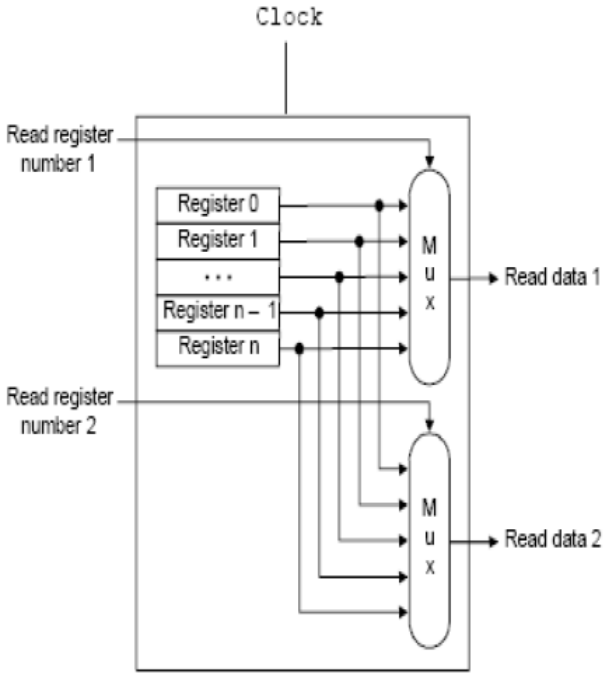


```
module dff(b,clk,q,q_0);
        input b, clk;
        output q,q_0;
        reg q;
        reg q_0;

        always@(posedge clk)
        begin
                q <= b;
                q_0 <= !b;
        end
endmodule

module register(d,clk,q,q_0);
        input wire[31:0] d;
        input clk;
        output wire[31:0] q;
        output wire[31:0] q_0;
        genvar i;
        generate
                for(i=0; i<=31; ++i) begin
                        dff d (d[i],clk,q[i],q_0[i]);
                        end
        endgenerate
endmodule
```

Here, again, the language comes to the aid. The command *genvar* sets up a running variable which then generates parallel instances of the same device. *generate* and *endgenerate* mark out the replication part of the code.

3.3.4 Register Files

Now, I move up the strucure of data storage sizes and we can see clearly that the next obvious choice of such devices is to collect many 32 bit registers into a sort of a "mega-register". This is nothing but a plain old register file. A simple diagram is given next page which explains the concept.



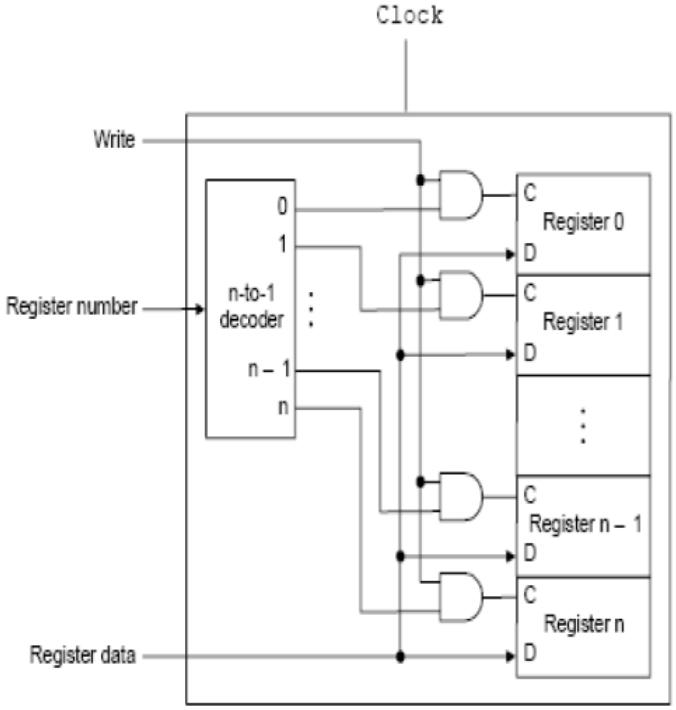
The register files are stacked up and each one can be summoned by calling their appropriate 5 bit register number. The multiplexors help us in selecting the right register number and supplying us with the data in the register. It is no surprise that the writing of data into the registers is very similar to the reading procedure and this will be discussed in the next few pages. Now, for the larger picture, the Verilog code.

```
module dff(b,clk,q,q_0);
        input b, clk;
        output q,q_0;
        reg q;
        reg q_0;

        always@(posedge clk)
        begin
                q <= b;
                q_0 <= !b;
```

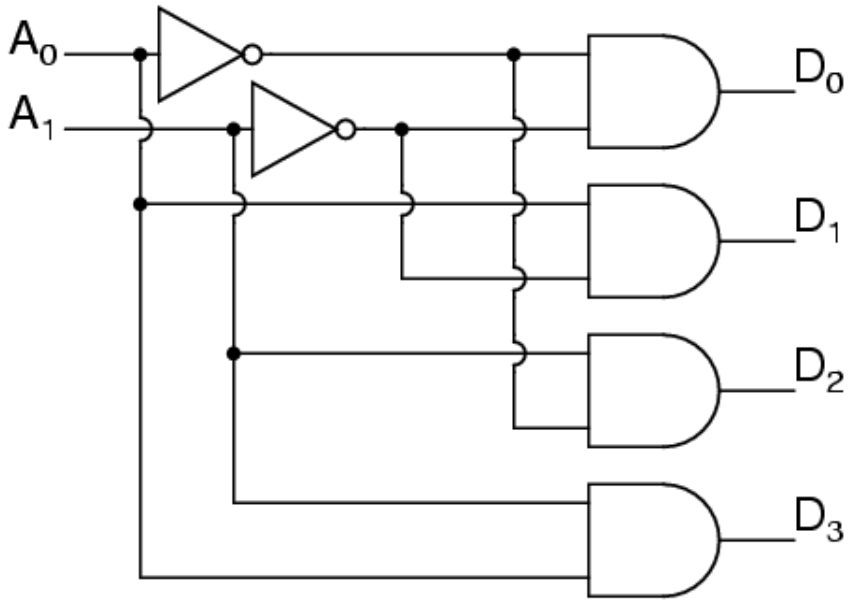
```
        end
endmodule

module register(d,clk,q,q_0);
        input wire[31:0] d;
        input clk;
        output wire[31:0] q;
        output wire[31:0] q_0;
        genvar i;
        generate
                for(i=0; i<=31; i=i+1) begin
                        dff d(d[i],clk,q[i],q_0[i]);
                        end
        endgenerate
endmodule


module smux(d,s,q);
        input[31:0] d;
        input[4:0] s;
        output q;
        wire[31:0] d;
        wire[4:0] s;
        wire[4:0] s1;
        reg q;
        integer i,w;
        always @ (s1 or d)
        begin
                for(i=0; i<=31; i=i+1)
                begin
                        if(s1==s)
                                w = s1;
                                q = d[w];
                        end
        end
endmodule
```

I have not written the *regfile* module altogether because the above script contains all the subcomponents of the file. All that is left to implement is keeping track of registers and implementing the correct

select lines. The block diagram schematic makes this point clear The write procedure for a register file, does however involve some differences, like the write synchronization with the clocking etc,. The block diagram for the writing method is given below. It involves the same stack to be rewritten into the data from the given register number.



Note: The component n to 1 decoder or simply a *decoder* is a digital device which has the task of setting one of the outputs in the given array of outputs as high and the remaining low. Technical jargon aside, it is very similar to the operation of a multiplexor. For a much simpler understanding, look at the schematic and the truth table followed by it.



A1	A0	D3	D2	D1	D0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

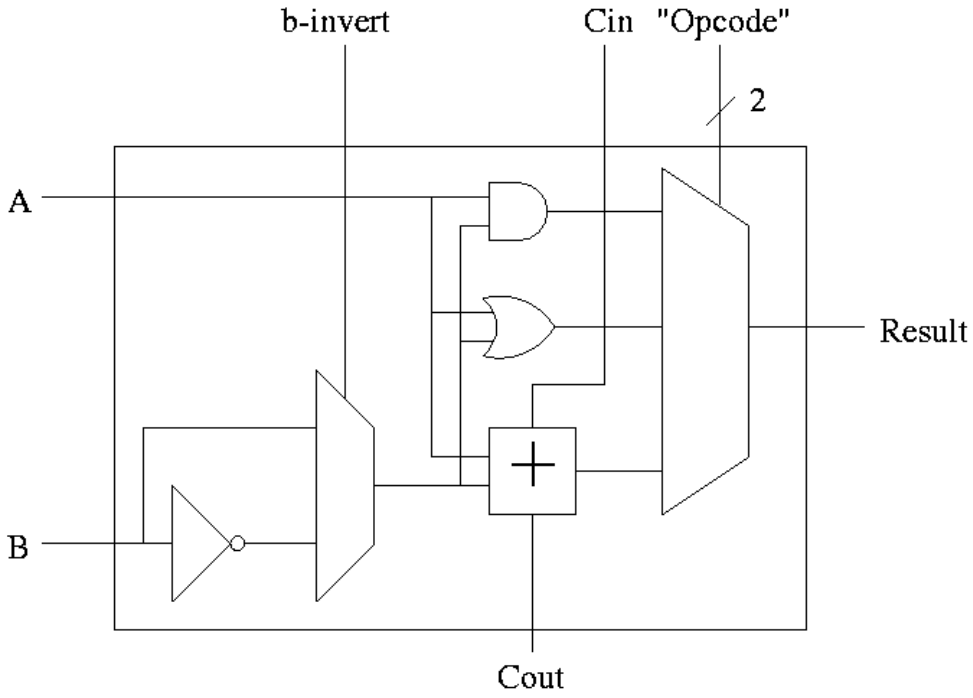
A simple script for the gate level design of the decoder is given below.

```
module dec(a,d);
        input[1:0] a;
        output[3:0] d;
        wire[1:0] a;
        wire[3:0] d;
        assign d[0] = ~(a[0])&~(a[1]);
        assign d[1] = a[0]&~(a[1]);
        assign d[2] = ~a[0]&a[1];
        assign d[3] = a[0]&a[1];
endmodule
```

I have *almost* come to an end to the description of the toolchain we would use to build a basic datapath. *However*, I would now describe the fully functional Arithmetic Logic Unit.

3.4 Arithmetic Logic Unit: A mix of things

As explained in a previous section, the ALU is the mothership for all arithmetic operations on data incoming from a register file. The methods in which data flows along through the ALU will be covered in more detail when I explain what a *datapath* is. For now, I shall explain the 1 bit ALU. Look at the schematic below.



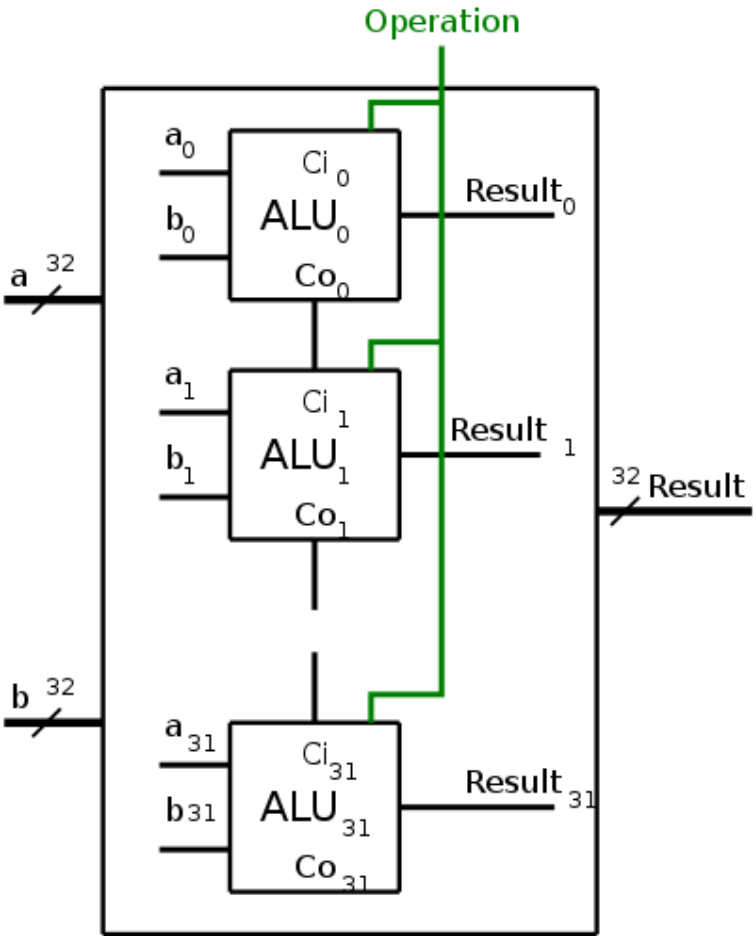
There are two inputs A and B for logical operations, the b-invert for switching from addition mode to subtraction mode(quite simply by adding the negated value of B), the Cin for any carry-ins entering the full adder inside the ALU. The input *opcode* (a 2 bit input) is the control line for the mux inside to select the operation the ALU has to perform. The result and cout are the standard output bits as usual.

The *block* module in the Verilog script on the next page represents the ALU block shown above.

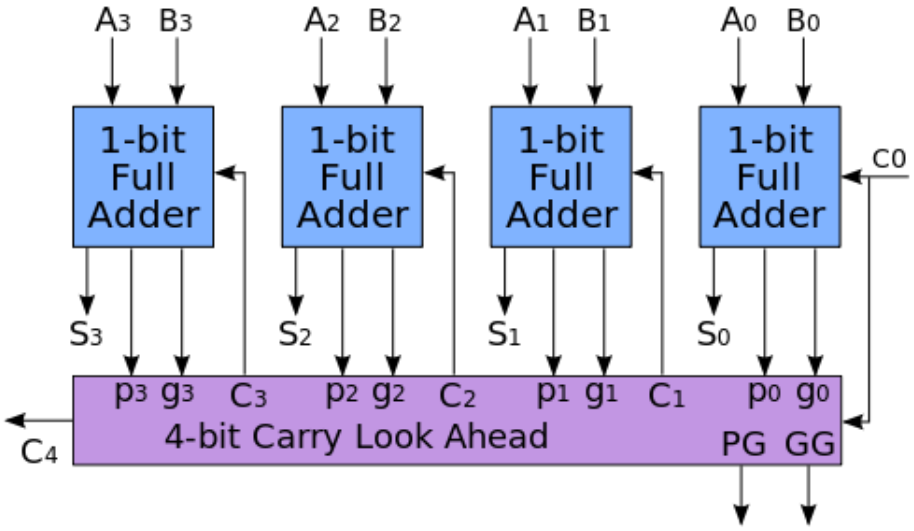
```
module mux(in1, in2, in3, select, out);
```

```
        input in1;
        input in2;
        input in3;
        input[1:0] select;
        output reg out;

        always@(in1 or in2 or select)
        begin
                if(select==2'b00) out=in1;
                else if(select==2'b01) out=in2;
                else if(select==2'b10) out=in3;
        end
endmodule


module fa(a,b,cin,sum,cout);
        input a,b,cin;
        output sum, cout;

        assign sum = cin ^ a ^ b;
        assign cout = ~cin & a & b| cin & (a|b);
//assigns the carryout and sum bits. Next step is to develop a ripple
endmodule



module block(in1, in2, carryin, control, result, carryout);
        input in1,in2,carryin;
        input[1:0] control;
        output result, carryout;

        wire w0,w1,w2;
        assign w0 = in1&in2;
        assign w1 = in|in2;
        fa f0(.a(in1),.b(in2),.cin(carryout),.sum(w2),.cout(carryout))
        mux m0(.in1(w0),.in2(w1),.in3(w2),.select(control),.out(result
endmodule
```

Now we look into the extension of this block unit. Let us see the schematic of a 32 bit ALU below.

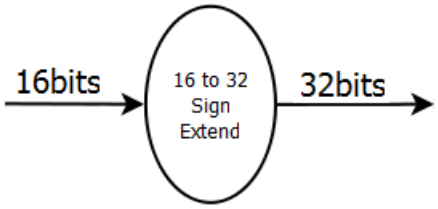


It is just following the general pattern of clustering digital devices to make a device of larger utility. Programming this is Verilog is also not difficult, noting the use of the generate command and running it through all the, luckily, independent inputs. People generally try to vary the design of the interconnection of the full adders inside the ALU for maximum efficiency. One such design, which we have seen before is the ripple carry adder. There is another eponymous design called the carry lookahead adder whose schematic is shown on the next page.



3.5 Sign Extender/Zero Extender and Logical Left Shifter

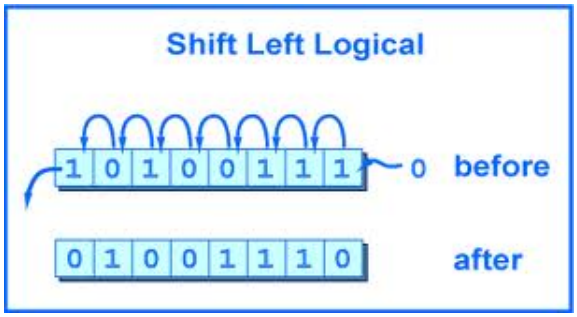
The operation of sign extension is a general term which refers to the process of increasing the number of bits of a binary number while keeping the sign and value preserved. Simply put, it is the process of adding more bits after the most significant bit(MSB). It is used while processing branching instructions for the datapath. The Verilog script for the sign extender is given below. The schematic is not neccesary right now. The symbol for the sign extender is given below.




```
module SignExt(in, out);

  input[15:0] in;
  output[31:0] out;
  reg [31:0] out;
```

```
    always@(in)
      begin

        out[31:0] ={{16{in[15]}}, in[15:0]};


      end
endmodule
```

The Shift Left Logical operation(*sll*) is to shift the left most bit in the bit pattern one bit leftwards. This can be clearly seen in the figure below



The Verilog script:

```
module LeftShift(in,out);
      input [31:0]in;
      output [31:0]out;
      reg [31:0]out;
      always@(in)
      begin
            out[31:0]={in[29:0],2'b00};
      end
endmodule
```

3.6 A few other useful Verilog scripts

Now, I am going to show you a few more Verilog scripts which were omitted before due to constraints. I start off with the proper *regfile* module and include two Verilog scripts from Stanford released open source which contain (maybe) refined versions of my components.

3.6.1 RegFile Module

```
module RegisterFile(readReg1, readReg2, writeReg, writeData, readData1
    input [4:0]readReg1, readReg2, writeReg;
    input [31:0]writeData; //address of the register to be written on
    input RegWrite; //RegWrite - register write signal; writeReg-the

    output [31:0]readData1, readData2;
    reg [31:0]readData1, readData2;

    reg [31:0]RegMemory[0:31];

    integer placeVal, i, j, writeRegINT=0, readReg1INT=0, readReg2INT

    initial
    begin
      for(i=0 ; i<32 ; i=i+1)
      begin
            for(j=0 ; j<32 ; j= j+1)
              RegMemory[i][j] = 1'b0;
      end
    end

    always@ (RegWrite or readReg1 or readReg2 or writeReg or writeDat
    begin

       if(RegWrite == 1)
       begin

         placeVal = 1;
         readReg1INT=0;
         readReg2INT=0;
         for(i=0 ; i<5 ; i=i+1)
         begin
              if(readReg1[i] == 1)
                 readReg1INT = readReg1INT + placeVal;

              if(readReg2[i] == 1)
                 readReg2INT = readReg2INT + placeVal;
```

```
        placeVal = placeVal * 2;
    end


    for(i=0 ; i<32 ; i=i+1)
    begin
        readData1[i] = RegMemory[readReg1INT][i];
        readData2[i] = RegMemory[readReg2INT][i];
    end



    //binary to decimal address translation.
    placeVal = 1;
    writeRegINT=0;
    for(i=0 ; i<5 ; i=i+1)
    begin
        if(writeReg[i] == 1)
            writeRegINT = writeRegINT + placeVal;

        placeVal = placeVal * 2;
    end

    $display("before writing %d at %d", writeData, writeRegINT);
    for(i=0 ; i<32 ; i=i+1)
    begin
         RegMemory[writeRegINT][i] = writeData[i];
    end
    $display("after writing %d at %d", writeData, writeRegINT);

end // Register Write

if(RegWrite == 0)
begin
    //binary to decimal address translation.
    placeVal = 1;
    readReg1INT=0;
    readReg2INT=0;
    for(i=0 ; i<5 ; i=i+1)
    begin
```

```
            if(readReg1[i] == 1)
               readReg1INT = readReg1INT + placeVal;

            if(readReg2[i] == 1)
               readReg2INT = readReg2INT + placeVal;

            placeVal = placeVal * 2;
        end


        for(i=0 ; i<32 ; i=i+1)
        begin
            readData1[i] = RegMemory[readReg1INT][i];
            readData2[i] = RegMemory[readReg2INT][i];
        end


    end// Register Read

  end //always@

endmodule
```

3.6.2 Stanford Standard Blocks 1

```
// building blocks for EE108 Lecture 4
// (c) Bill Dally - 2003 - All Rights Reserved
// 1/15/2003
//-----------------------------------------------------------------------

// test script
module test ;
  parameter n=4 ;
  parameter m=16 ;

  reg [5:0] a ;
  wire [m-1:0] b ;
  wire [3:0] c ;
  wire [63:0] d ;
  wire [n-1:0] e,g ;
```

```
  reg [m-1:0] f ;
  wire [1:0] h ;
  wire [5:0] j ;
  wire [2:0] k ;

  Decoder #(n,m) dec(a[n-1:0], b) ;
  Dec24 d24(a[1:0],c) ;
  Dec664 d64(a,d) ;
  //Encoder #(m,n) enc(f, e) ;
  Enc164 e2(f,g) ;
  //Mux4 #(2) mx(2'd0, 2'd1, 2'd2, 2'd3, f, h) ;
  Arb #(6) arb(a,j) ;
  Encoder #(6,3) enc2(j,k) ;

  initial begin
    a = 0 ; f = 1 ;
    repeat (64) begin
      // #100 $display("%b %b %b %h",a,b,c,d) ;
      #100 $display("%b %b %d",a,j,k) ;
      a = a+ 1 ; f = f<<1;
    end
  end
endmodule

//-----------------------------------------------------------------------------

// arbitrary width decoder
module Decoder(a, b) ;
  parameter n = 3 ;
  parameter m = 8 ;
  input  [n-1:0] a ;
  output [m-1:0] b ;

  reg [m-1:0] b ;
  integer i ;

  always @(a) begin
    for(i=0;i<m;i=i+1) begin
      b[i] = (a == i) ? 1 : 0 ;
    end
```

```
  end
endmodule

//------------------------------------------------------------------------

// fixed width decoder
module Dec24(a, b) ;
  input  [1:0] a ;
  output [3:0] b ;
  wire   [3:0] b ;

  assign b[0] = !a[0] & !a[1] ;
  assign b[1] =  a[0] & !a[1] ;
  assign b[2] = !a[0] &  a[1] ;
  assign b[3] =  a[0] &  a[1] ;
endmodule

//------------------------------------------------------------------------

// factored decoder
module Dec664(a, b) ;
  input  [5:0]  a ;
  output [63:0] b ;

  reg    [63:0] b ;
  wire   [3:0]  c, d, e ;

  integer i ;

  Dec24 d0(a[1:0],c) ;
  Dec24 d1(a[3:2],d) ;
  Dec24 d2(a[5:4],e) ;

  always @(c or d or e) begin
    for(i=0;i<64;i=i+1) begin
      b[i] = c[i & 3] & d[(i>>2) & 3] & e[(i>>4) & 3] ;
    end
  end
endmodule
```

```
//-----------------------------------------------------------------------

// encoder
module Encoder(a, b) ;
  parameter n = 8 ;
  parameter m = 3 ;
  input  [n-1:0] a ;
  output [m-1:0] b ;
  reg    [m-1:0] b ;
  integer i ;

  always @(a) begin
    for(i=0;i<n;i=i+1) begin
      if(a[i] == 1) b = i ;
    end
    if(a == 0) b = 0 ;
  end
endmodule

//-----------------------------------------------------------------------

// encoder - fixed width
module Enc42(a, b) ;
  input  [3:0] a ;
  output [1:0] b ;
  wire   [1:0] b ;

  assign b[1] = a[3] | a[2] ;
  assign b[0] = a[3] | a[1] ;
endmodule
//-----------------------------------------------------------------------

// encoder - fixed width - with summary output
module Enc42a(a, b, c) ;
  input  [3:0] a ;
  output [1:0] b ;
  output c ;
  wire   [1:0] b ;
  wire   c ;
```

```
  assign b[1] = a[3] | a[2] ;
  assign b[0] = a[3] | a[1] ;
  assign c = |a ;
endmodule

//-------------------------------------------------------------------------

// factored encoder
module Enc164(a, b) ;
  input [15:0] a ;
  output[3:0]  b ;
  wire [3:0] b ;
  wire [7:0] c ; // intermediate result of first stage
  wire [3:0] d ; // if any set in group of four

  // four LSB encoders each include 4-bits of the input
  Enc42a e0(a[3:0],  c[1:0],d[0]) ;
  Enc42a e1(a[7:4],  c[3:2],d[1]) ;
  Enc42a e2(a[11:8], c[5:4],d[2]) ;
  Enc42a e3(a[15:12],c[7:6],d[3]) ;

  // MSB encoder takes summaries and gives msb of output
  Enc42 e4(d[3:0], b[3:2]) ;

  // two OR gates combine output of LSB encoders
  assign b[1] = c[1]|c[3]|c[5]|c[7] ;
  assign b[0] = c[0]|c[2]|c[4]|c[6] ;
endmodule
//-------------------------------------------------------------------------

// two input mux with one-hot select (arbitrary width)
module Mux2(a0, a1, s, b) ;
  parameter k = 1 ;
  input [k-1:0] a0, a1 ;  // inputs
  input [1:0]   s ; // one-hot select
  output[k-1:0] b ;
  wire [k-1:0] b = ({k{s[0]}} & a0) |
                   ({k{s[1]}} & a1) ;
endmodule
```

```
//------------------------------------------------------------------------

// four input mux with one-hot select (arbitrary width)
module Mux4(a0, a1, a2, a3, s, b) ;
  parameter k = 1 ;
  input [k-1:0] a0, a1, a2, a3 ;  // inputs
  input [3:0]   s ; // one-hot select
  output[k-1:0] b ;
  wire [k-1:0] b = ({k{s[0]}} & a0) |
                   ({k{s[1]}} & a1) |
                   ({k{s[2]}} & a2) |
                   ({k{s[3]}} & a3) ;
endmodule

//------------------------------------------------------------------------

// five input mux with one-hot select (arbitrary width)
module Mux5(a0, a1, a2, a3, a4, s, b) ;
  parameter k = 1 ;
  input [k-1:0] a0, a1, a2, a3, a4 ;  // inputs
  input [4:0]   s ; // one-hot select
  output[k-1:0] b ;
  wire [k-1:0] b = ({k{s[0]}} & a0) |
                   ({k{s[1]}} & a1) |
                   ({k{s[2]}} & a2) |
                   ({k{s[3]}} & a3) |
                   ({k{s[4]}} & a4) ;
endmodule
//------------------------------------------------------------------------

// eight input mux with one-hot select (arbitrary width)
module Mux8(a0, a1, a2, a3, a4, a5, a6, a7, s, b) ;
  parameter k = 1 ;
  input [k-1:0] a0, a1, a2, a3, a4, a5, a6, a7 ;
// inputs
  input [7:0]   s ; // one-hot select
  output[k-1:0] b ;
  wire [k-1:0] b = ({k{s[0]}} & a0) |
                   ({k{s[1]}} & a1) |
                   ({k{s[2]}} & a2) |
```

```
                    ({k{s[3]}} & a3) |
                    ({k{s[4]}} & a4) |
                    ({k{s[5]}} & a5) |
                    ({k{s[6]}} & a6) |
                    ({k{s[7]}} & a7) ;
endmodule
//-------------------------------------------------------------------------
// funnel shifter - takes an n-bit input and shifts it by up to n-m
// bits to generate an m-bit output
// one hot input s selects how much to shift.
// if s[i] is high bit in[i+j] is routed to out[j].
// if s is zero, out is undefined
//-------------------------------------------------------------------------

module FunnelShift(in, s, out) ;
  parameter n = 12 ;
  parameter m = 5 ;
  input [n-1:0] in ;
  input [n-m-1:0] s ;
  output [m-1:0] out ;
  reg [m-1:0] out ;
  integer i ;

  always @(in or s) begin
    for(i=0;i<(n-m);i=i+1) begin
      if(s[i] == 1) out = in>>i ;
    end
    if(s == 0) out = {m{1'bx}} ;
  end
endmodule
//-------------------------------------------------------------------------

// mux with one-hot select (arbitrary ports - one bit wide)
module Mux(a,s,b) ;
  parameter n = 8 ;
  input [n-1:0] a, s ;
  output b ;
  wire [n-1:0] p = a & s ; // product of input and select
  wire b = |p ;
endmodule
```

```
//-----------------------------------------------------------------------

// mux with binary select (arbitrary width)
module BMux(a, s, b) ;
  parameter n = 8 ;
  parameter m = 3 ;
  input[n-1:0] a ;
  input[m-1:0] s ;
  output b ;
  wire [n-1:0] ds ; // decoded select

  Decoder #(m,n) dec(s, ds) ;
  Mux #(n) mux(a, ds, b) ;
endmodule
//-----------------------------------------------------------------------

// arbiter (arbitrary width) - LSB is highest priority
module Arb(r, g) ;
  parameter n=8 ;
  input  [n-1:0] r ;
  output [n-1:0] g ;
  wire   [n-1:0] c = {(~r[n-2:0] & c[n-2:0]),1'b1} ;
  wire   [n-1:0] g = r & c ;
endmodule
//-----------------------------------------------------------------------
// arbiter (arbitrary width) - MSB is highest priority
module RArb(r, g) ;
  parameter n=8 ;
  input  [n-1:0] r ;
  output [n-1:0] g ;
  wire   [n-1:0] c = {1'b1,(~r[n-1:1] & c[n-1:1])} ;
  wire   [n-1:0] g = r & c ;
endmodule
//-----------------------------------------------------------------------

// priority encoder (arbitrary width)
module PriorityEncoder(r, b) ;
  parameter n=8 ;
  parameter m=3 ;
  input  [n-1:0] r ;
```

```
  output [m-1:0] b ;
  wire   [n-1:0] g ;
  Arb #(n) a(r, g) ;
  Encoder #(n,m) e(g, b) ;
endmodule
//------------------------------------------------------------------------
```

3.6.3 Stanford Standard Blocks 2

```
// arithmetic circuits for EE108 Lecture 5
// (c) Bill Dally - 2003 - All Rights Reserved
// 1/18/2003
//------------------------------------------------------------------------

// test script
module test4 ;

  reg [15:0] in0, in1 ;
  wire [15:0] out ;
  reg cin ;
  wire cout ;

  Add16 a(in0,in1,cin,cout,out) ;

  initial begin
    in0 = 16'd12345 ; in1 = 16'd43210 ; cin=0 ;
    #100 $display("%05h + %05h = %05h cin = %b cout = %b",
    in0, in1, out, cin, cout) ;
    $display("p=%b g=%b c=%b pp=%b gg=%b cc=%b",
    a.p, a.g, a.c, a.pp, a.gg, {a.c[12],a.c[8],a.c[4]}) ;

    in0 = 16'd12345 ; in1 = 16'd43210 ; cin=1 ;
    #100 $display("%05h + %05h = %05h cin = %b cout = %b",
    in0, in1, out, cin, cout) ;
    $display("p=%b g=%b c=%b pp=%b gg=%b cc=%b",
    a.p, a.g, a.c, a.pp, a.gg, {a.c[12],a.c[8],a.c[4]}) ;

    in0 = 16'h7fff ; in1 = 16'h8000 ; cin = 0 ;
    #100 $display("%05h + %05h = %05h cin = %b cout = %b",
    in0, in1, out, cin, cout) ;
```

```
    $display("p=%b g=%b c=%b pp=%b gg=%b cc=%b",
    a.p, a.g, a.c, a.pp, a.gg, {a.c[12],a.c[8],a.c[4]}) ;

    in0 = 16'h7fff ; in1 = 16'h8000 ; cin = 1 ;
    #100 $display("%05h + %05h = %05h cin = %b cout = %b",
    in0, in1, out, cin, cout) ;
    $display("p=%b g=%b c=%b pp=%b gg=%b cc=%b",
    a.p, a.g, a.c, a.pp, a.gg, {a.c[12],a.c[8],a.c[4]}) ;
  end
endmodule
//----------------------------------------------------------------------

// test script
module test3 ;

  reg [3:0] in0, in1 ;
  wire [7:0] out ;

  Mul4 mul(in0,in1,out) ;

  initial begin
    in0 = 0 ;
    repeat (16) begin
      in1 = 0 ;
      repeat (in0+1) begin
        #100 $display("%03d * %03d = %03d",in0,in1,out) ;
        in1 = in1 + 1 ;
      end
      in0 = in0 + 1 ;
    end
  end
endmodule
//----------------------------------------------------------------------

// test script
module test2 ;

  reg [7:0] in0, in1 ;
  wire [7:0] out ;
  reg sub ;
```

```
  wire ovf ;

  AddSub #(8) a(in0,in1,sub,out,ovf) ;

  initial begin
    in0 = 8'd87  ; in1 = 8'd40 ; sub = 0 ;
    #100 $display("%03h + %03h = %03h ovf = %b", in0, in1, out, ovf)
    in0 = 8'd87  ; in1 = 8'd40 ; sub = 1 ;
    #100 $display("%03h - %03h = %03h ovf = %b", in0, in1, out, ovf)
    in0 = 8'd88  ; in1 = 8'd40 ; sub = 0 ;
    #100 $display("%03h + %03h = %03h ovf = %b", in0, in1, out, ovf)
    in0 = 8'he9  ; in1 = 8'h17 ; sub = 1 ; /* -23 - 23 */
    #100 $display("%03h - %03h = %03h ovf = %b", in0, in1, out, ovf)
    in0 = 8'he9  ; in1 = 8'h97 ; sub = 0 ; /* -23 - 105 = -128
no ovf */
    #100 $display("%03h + %03h = %03h ovf = %b", in0, in1, out, ovf)
    in0 = 8'he9  ; in1 = 8'd106 ; sub = 1 ; /* overflow */
    #100 $display("%03h - %03h = %03h ovf = %b", in0, in1, out, ovf)
  end
endmodule
//------------------------------------------------------------------------

// test script
module test1 ;

  reg [7:0] in0, in1 ;
  wire [7:0] out ;
  reg cin ;
  wire cout ;

  Adder1 #(8) a(in0,in1,cin,cout,out) ;

  initial begin
    in0 = 8'd87  ; in1 = 8'd139 ; cin = 0 ;
    #100 $display("%03d + %03d + %b = %03d c = %b", in0, in1, cin, out
    in0 = 8'd87  ; in1 = 8'd139 ; cin = 1 ;
    #100 $display("%03d + %03d + %b = %03d c = %b", in0, in1, cin, out
    in0 = 8'd127  ; in1 = 8'd128 ; cin = 0 ;
    #100 $display("%03d + %03d + %b = %03d c = %b", in0, in1, cin, out
    in0 = 8'd127  ; in1 = 8'd128 ; cin = 1 ;
```

```
    #100 $display("%03d + %03d + %b = %03d c = %b", in0, in1, cin, out
    in0 = 8'd123  ; in1 = 8'd99 ; cin = 0 ;
    #100 $display("%03d + %03d + %b = %03d c = %b", in0, in1, cin, out
    in0 = 8'd123  ; in1 = 8'd99 ; cin = 1 ;
    #100 $display("%03d + %03d + %b = %03d c = %b", in0, in1, cin, out
  end
endmodule
//-----------------------------------------------------------------------

module test ;

  reg [2:0] in ;
  wire [1:0] o1, o2, o3 ;

  HalfAdder ha(in[0],in[1],o1[1],o1[0]) ;
  FullAdder1 fa1(in[0],in[1],in[2],o2[1],o2[0]) ;
  FullAdder2 fa2(in[0],in[1],in[2],o3[1],o3[0]) ;

  initial begin
    in = 0 ;
    repeat (8) begin
      #100 $display("%b %b %b %b",in,o1,o2,o3) ;
      in = in+ 1 ;
    end
  end
endmodule
//-----------------------------------------------------------------------

// half adder
module HalfAdder(a,b,c,s) ;
  input a,b ;
  output c,s ;  // carry and sum
  wire s = a ^ b ;
  wire c = a & b ;
endmodule

//-----------------------------------------------------------------------

// full adder - from half adders
module FullAdder1(a,b,cin,cout,s) ;
```

```
  input a,b,cin ;
  output cout,s ;  // carry and sum
  wire g,p ;     // generate and propagate
  wire cp ;
  HalfAdder ha1(a,b,g,p) ;
  HalfAdder ha2(cin,p,cp,s) ;
  or o1(cout,g,cp) ;
endmodule
//-------------------------------------------------------------------------

// full adder - logical
module FullAdder2(a,b,cin,cout,s) ;
  input a,b,cin ;
  output cout,s ;  // carry and sum
  wire s = a ^ b ^ cin ;
  wire cout = (a & b)|(a & cin)|(b & cin) ; // majority
endmodule
//-------------------------------------------------------------------------

// multi-bit adder - structural
module Adder(a,b,cin,cout,s) ;
  parameter n = 8 ;
  input [n-1:0] a, b ;
  input cin ;
  output [n-1:0] s ;
  output cout ;
  wire [n:0] c ;
  genvar i ;
  assign c[0] = cin ;
  assign cout = c[n] ;

  generate
    for(i=0;i<n;i=i+1) begin:bit
      FullAdder2 a(a[i],b[i],c[i],c[i+1],s[i]) ;
    end
  endgenerate
endmodule
//-------------------------------------------------------------------------

// multi-bit adder - behavioral
```

```
module Adder1(a,b,cin,cout,s) ;
  parameter n = 8 ;
  input [n-1:0] a, b ;
  input cin ;
  output [n-1:0] s ;
  output cout ;
  wire [n-1:0] s;
  wire cout ;

  assign {cout, s} = a + b + cin ;
endmodule
//-------------------------------------------------------------------------

// add a+b or subtract a-b, check for overflow
module AddSub(a,b,sub,s,ovf) ;
  parameter n = 8 ;
  input [n-1:0] a, b ;
  input sub ;             // subtract if sub=1, otherwise add
  output [n-1:0] s ;
  output ovf ;            // 1 if overflow
  wire c1, c2 ;           // carry out of last two bits
  wire ovf = c1 ^ c2 ;    // overflow if signs don't match

  // add non sign bits
  Adder1 #(n-1) ai(a[n-2:0],b[n-2:0]^{n-1{sub}},sub,c1,s[n-2:0]) ;
  // add sign bits
  Adder1 #(1)   as(a[n-1],b[n-1]^sub,c1,c2,s[n-1]) ;
endmodule
//-------------------------------------------------------------------------

// 4-bit multiplier
module Mul4(a,b,p) ;
  input [3:0] a,b ;
  output [7:0] p ;

  // form partial products
  wire [3:0] pp0 = a & {4{b[0]}} ; // x1
  wire [3:0] pp1 = a & {4{b[1]}} ; // x2
  wire [3:0] pp2 = a & {4{b[2]}} ; // x4
  wire [3:0] pp3 = a & {4{b[3]}} ; // x8
```

```
  // sum up partial products
  wire cout1, cout2, cout3 ;
  wire [3:0] s1, s2, s3 ;
  Adder1 #(4) a1(pp1, {1'b0,pp0[3:1]}, 1'b0, cout1, s1) ;
  Adder1 #(4) a2(pp2, {cout1,s1[3:1]}, 1'b0, cout2, s2) ;
  Adder1 #(4) a3(pp3, {cout2,s2[3:1]}, 1'b0, cout3, s3) ;

  // collect the result
  wire [7:0] p = {cout3, s3, s2[0], s1[0], pp0[0]} ;
endmodule
//------------------------------------------------------------------------

// 4:1 pg combine for fast adder carry chain
module PG4(pin,gin,pout,gout) ;
  input [3:0] pin, gin ;
  output pout, gout ;

  wire pout = &pin ;
  wire gout = gin[3] | (pin[3] & (gin[2] | (pin[2]
                                  & (gin[1] | (pin[1] & gin[0]))))) ;
endmodule
//------------------------------------------------------------------------

// 1:4 carry expand for fast adder carry chain
module Carry4(p,g,cin,cout) ;
  input [2:0] p, g ;
  input cin ;
  output [2:0] cout ;
  wire [2:0] cout = g | (p & {cout[1:0],cin}) ;
endmodule
//------------------------------------------------------------------------

module Add16(a,b,cin,cout,s) ;
  input [15:0] a, b ;
  input cin ;
  output cout ;
  output [15:0] s ;

  // single bit p and g
```

```
  wire [15:0] p = a^b ;
  wire [15:0] g = a&b ;

  // four bit p and g
  wire [3:0] pp, gg ;
  wire ppp, ggg ;
  PG4 pg0(p[3:0],g[3:0],pp[0],gg[0]) ;
  PG4 pg1(p[7:4],g[7:4],pp[1],gg[1]) ;
  PG4 pg2(p[11:8],g[11:8],pp[2],gg[2]) ;
  PG4 pg3(p[15:12],g[15:12],pp[3],gg[3]) ;
  PG4 pgtop(pp[3:0],gg[3:0],ppp,ggg) ;

  // four bit carry expand
  wire [15:0] c ;
  assign c[0] = cin ;
  Carry4 ctop(pp[2:0],gg[2:0],cin,{c[12],c[8],c[4]}) ;
  Carry4 c0(p[2:0],g[2:0],cin,c[3:1]) ;
  Carry4 c1(p[6:4],g[6:4],c[4],c[7:5]) ;
  Carry4 c2(p[10:8],g[10:8],c[8],c[11:9]) ;
  Carry4 c3(p[14:12],g[14:12],c[12],c[15:13]) ;

  // form sum
  wire [15:0] s = p^c ;

  // carry out
  assign cout = ggg | (ppp & cin) ;
endmodule
//-----------------------------------------------------------------------------
```