Half adder

```
module ha(sum, carry, a, b);
output sum;
output carry;
input a;
input b;
xor g1(sum,a,b);
and g2(carry,a,b);
endmodule
```

Full adder

```
module fa(sum, carry, a, b, cin);
output sum;
output carry;
input a;
input b;
input cin;
wire w1,w2,w3;
ha h1(w1,w2,a,b);
ha h2(sum,w3,cin,w1);
or h3(carry,w3,w2);
endmodule
```

RCA

```
module ripple(sum, carry, a,b, cin);
input [3:0] sum;
input carry;
input [3:0] a,b;
input cin;
wire c1,c2,c3;
fa f1(sum[0],c1,a[0],b[0],cin);
fa f2(sum[1],c2,a[1],b[1],c1);
fa f3(sum[2],c3,a[2],b[2],c2);
fa f4(sum[3],carry,a[3],b[3],c3);
endmodule
```

2X1MUX

```
module mux_2x1(y, i0,i1,s);
output y;
input i0,i1,s;
wire sbar;
assign sbar=~s;
assign y=(sbar&i0) | (s&i1);
endmodule
```

4X1MUX

```
module mux_4x1(out, i0,i1,i2,i3, s0,s1);
output out;
input i0,i1,i2,i3;
input s0,s1;
assign out=(~s1&~s0&i0)|(~s1&s0&i1)|(s1&~s0&i2)|(s1&s0&i3);
endmodule
```

8X1MUX

```
module mux_8x1(out, i, s);
output out;
input [7:0] i;
input [2:0] s;
wire w1,w2;
mux_4x1 m1(w1,i[0],i[1],i[2],i[3],s[0],s[1]);
mux_4x1 m2(w2,i[4],i[5],i[6],i[7],s[0],s[1]);
mux_2x1 m3(out,w1,w2,s[2]);
endmodule
```

SRFF

```
module srff(q, qbar, s, r, clk);
output q;
output qbar;
input s;
input r;
input clk;
wire nand1_out,nand2_out;
nand n1(nand1_out,clk,s);
nand n2(nand2_out,clk,r);
nand n3(q,nand1_out,qbar);
nand n4(qbar,nand2_out,q);
endmodule
```

JKFF

```
module jkff(q,qbar, clk,clr, j,k);
output q,qbar;
input clk,clr;
input j,k;
reg q;
initial
q=0;
always@(negedge clk)
```

```
begin
q=(j & (~q)) | ((~k)&q);
end
assign qbar=~q;
endmodule
```

3X8DECODER

```
module dec3x8(d, a, e);
output [7:0] d;
input [2:0] a;
input e;
wire x,y,z;
not g1(z,a[0]);
not g2(y,a[1]);
not g3(x,a[2]);
and g4(d[0],x,y,z,e);
and g5(d[1],x,y,a[0],e);
and g6(d[2],x,a[1],z,e);
and g7(d[3],x,a[1],a[0],e);
and g8(d[4],a[2],y,z,e);
and g9(d[5],a[2],y,a[0],e);
and g10(d[6],a[2],a[1],z,e);
and g11(d[7],a[2],a[1],a[0],e);
endmodule
```

4X16 using 3x8

```
module dec4x16(out, i);
output [15:0] out;
input [3:0] i;
dec3x8 g1(out[7:0],i[2:0],~i[3]);
dec3x8 g2(out[15:8],i[2:0],~i[3]);
endmodule
```

DFF

```
module dff(q, qbar, d, clk, clear);
output q;
output qbar;
input d;
input clk;
input clear;
reg q,qbar;
always@(posedge clk or posedge clear)
begin
if(clear==1)
begin
q <=0;
qbar <= 1;
end
```

```
else
begin
q <=d;
qbar <=!d;
end
end
endmodule
```

TFF

```
module tff(q, clk, clr, t);
output q;
input clk;
input clr;
input t;
reg q;
initial
q=0;
always@(posedge clk)
begin
case({clr,t})
2'b10: q=0;
2'b00: q=q;
2'b01: q=~q;
endcase
end
endmodule
```

UPDOWN COUNTER

```
module updown(q, clr, clk, mod);
output reg [3:0] q;
input clr;
input clk;
input mod;
always@(posedge clk)
begin
case({clr,mod})
2'b11 : q=0;
2'b10 : q=0;
2'b01 : q=q+1;
2'b00:q=q-1;
endcase
end
endmodule
```

MODN COUNTER

```
module modn # (parameter N=10,
```

```
parameter WIDTH=4)
(input clk,
input rstn,
output reg[WIDTH-1:0] out);
always@ (posedge clk)
begin
if(rstn)
begin
out <= 0;
end
else
begin
if(out==N-1)
out <=0;
else
out <= out+1;
end
end
endmodule
```

SISO

```
module siso(shift_out, shift_in, clk);
output shift_out;
input shift_in;
input clk;
reg shift_out;
reg [2:0] data;
always @(posedge clk)
begin
data[0] <= shift_in;
data[1] <= data[0];
data[2] <= data[1];
shift_out <= data[2];
end
endmodule
```

INVERTER USING SWITCH

```
module inverter(out, in);
output out;
input in;
//declare power and ground
supply1 pwr;
supply0 gnd;
//instantiate nmos and pmos switches
pmos(out,pwr,in);
nmos(out,gnd,in);
endmodule
```

CMOS NAND

```
module cmos_nand(out, a,b);
output out;
input a,b;
wire w;
supply1 vcc;
supply0 gnd;
pmos p1(out,vcc,a);
pmos p2(out,vcc,b);
nmos n1(w,gnd,b);
nmos n2(out,w,a);
endmodule
```

2X1MUX USING CMOS

```
module mux2x1_cmos(out, s,i0,i1);
output out;
input s,i0,i1;
//internal wire
wire sbar;
not n1(sbar, s);
//cmos switches
cmos c1(out, i0, sbar, s);
cmos c2(out, i1, s, sbar);
endmodule
```

SIPO

```
module sipo(q, clk,clr,d);
output reg [3:0] q;
input clk,clr,d;
reg [3:0] tmp;
always @(posedge clk)
if(clr==1)
q=4'b0000;
else
begin
tmp = q>>1;
q={d,tmp[2:0]};
end
endmodule
```

PIPO

```
module pipo(q, d,clk,clr);
output reg [3:0] q;
input [3:0] d,clk,clr;
```

```
always @(posedge clk)
if(clr==1)
q=4'b0000;
else
q=d;
endmodule
```

FSM111

```
module fsm_111(o, reset,a,clk);
output reg o;
input reset,a,clk;
reg [1:0]pre_s,nxt_s;
initial begin
pre_s=2'b00;
end
always @(posedge clk)
begin
if(reset==1)
begin
pre_s=2'b00;
o=0;
end
else
begin
case(pre_s)
2'b00:
begin
if(a==1)
begin
=0;
nxt_s=2'b01;
pre_s=nxt_s;
end
else
begin
o=0;
nxt_s=2'b00;
pre_s=nxt_s;
end
end
2'b01:
begin
if(a==1)
begin
o=0;
nxt_s=2'b10;
pre_s=nxt_s;
end
else
begin
```

```
o=0;
nxt_s=2'b00;
pre_s=nxt_s;
end
end
2'b10:
begin
if(a==1)
begin
o=1;
nxt_s=2'b00;
pre_s=nxt_s;
end
else
begin
o=0;
nxt_s=2'b00;
pre_s=nxt_s;
end
end
default: pre_s= 2'b00;
endcase
end
end
endmodule
```

CLA

```
module claadder(sum, cout, cin, a, b);
output [3:0]sum;
output cout;
input cin;
input [3:0]a;
input [3:0]b;
wire p0,p1,p2,p3,g0,g1,g2,g3;
wire c1,c2,c3,c4;
assign
p0=(a[0]^b[0]),
p1=(a[1]^b[1]),
p2=(a[2]^b[2]),
p3=(a[3]^b[3]);
assign
g0=(a[0]&b[0]),
g1=(a[1]&b[1]),
g2=(a[2]&b[2]),
g3=(a[3]&b[3]);
assign c0=cin,
c1=g0|(p0&cin),
c2=g1|(p1&g0)|(p1&p0&cin),
c3=g2|(p2&g1)|(p2&p1&g0)|(p2&p1&p0&cin),
```

```
c4=g3|(p3&g2)|(p3&p2&g1)|(p3&p2&p1&g0)|(p3&p2&p1&p0&cin);
assign
sum[0]=p0^cin,
sum[1]=p1^c1,
sum[2]=p2^c2,
sum[3]=p3^c3;
assign
cout=c4;
endmodule
```

CSA

```
module csadder(a,b, cin, sum, cout);
input [3:0] a,b;
input cin;
output [3:0] sum;
output cout;
wire[3:0]p;
wire c0;
wire bp;
ripple g1(sum[3:0],c0,a[3:0],b[3:0],cin);
propogate_p p1(p,bp,a,b);
mux_2x1 g2(cout,c0,cin,bp);
endmodule
module propogate_p(p,bp,a,b);
input [3:0]a,b;
output [3:0]p;
output bp;
assign p=a^b;
assign bp=&p;
endmodule
```

4-BIT BRAUN ARRAY

```
module braun( a, b,p);
output [7:0] p;
input [3:0] a;
input [3:0] b;
wire [32:1]w;
and a1(p[0],a[0],b[0]);
and a2(w[1],a[1],b[0]);
and a3(w[2],a[2],b[0]);
and a4(w[3],a[3],b[0]);
and a5(w[4],a[0],b[1]);
and a6(w[5],a[1],b[1]);
and a7(w[6],a[2],b[1]);
and a8(w[7],a[3],b[1]);
and a9(w[8],a[0],b[2]);
and a10(w[9],a[1],b[2]);
```

```
and a11(w[10],a[2],b[2]);
and a12(w[11],a[3],b[2]);
and a13(w[12],a[0],b[3]);
and a14(w[13],a[1],b[3]);
and a15(w[14],a[2],b[3]);
and a16(w[15],a[3],b[3]);
fa f1(p[1],w[16],w[1],w[4],1'b0);
fa f2(w[32],w[17],w[5],w[2],1'b0);
fa f3(w[31],w[18],w[3],w[6],1'b0);
fa f4(p[2],w[19],w[8],w[32],w[16]);
fa f5(w[30],w[20],w[31],w[9],w[17]);
fa f6(w[29],w[21],w[7],w[10],w[18]);
fa f7(p[3],w[22],w[30],w[12],w[19]);
fa f8(w[26],w[23],w[29],w[13],w[20]);
fa f9(w[25],w[24],w[11],w[14],w[21]);
fa f10(p[4],w[27],w[26],1'b0,w[22]);
fa f11(p[5],w[28],w[25],w[27],w[23]);
fa f12(p[6],p[7],w[15],w[28],w[24]);
endmodule
```

BOOTH MULTIPLIER

```
module BoothMul(clk,rst,start,X,Y,valid,Z);
input clk;
input rst;
input start;
input signed [3:0]X,Y;
output signed [7:0]Z;
output valid;
reg signed [7:0] Z,next_Z,Z_temp;
reg next_state, pres_state;
reg [1:0] temp,next_temp;
reg [1:0] count,next_count;
reg valid, next_valid;
parameter IDLE = 1'b0;
parameter START = 1'b1;
always @ (posedge clk or negedge rst)
begin
if(!rst)
begin
Z <= 8'd0;
valid <= 1'b0;
pres_state <= 1'b0;
temp <= 2'd0;
count <= 2'd0;
end
else
begin
Z <= next_Z;
valid <= next_valid;
pres_state <= next_state;
```

```
temp <= next_temp;
count <= next_count;
end
end
always @ (*)
begin
case(pres_state)
IDLE:
begin
next_count = 2'b0;
next_valid = 1'b0;
if(start)
begin
next_state = START;
next_temp = {X[0],1'b0};
next_Z = {4'd0,X};
end
else
begin
next_state = pres_state;
next_temp = 2'd0;
next_Z = 8'd0;
end
end
START:
begin
case(temp)
2'b10: Z_temp = {Z[7:4]-Y,Z[3:0]};
2'b01: Z_temp = {Z[7:4]+Y,Z[3:0]};
default: Z_temp = {Z[7:4],Z[3:0]};
endcase
next_temp = {X[count+1],X[count]};
next_count = count + 1'b1;
next_Z = Z_temp >>> 1;
next_valid = (&count) ? 1'b1 : 1'b0;
next_state = (&count) ? IDLE : pres_state;
end
endcase
end
endmodule
```

WALLACE TREE

```
module wtm(p, a,b);
output [7:0] p;
input [3:0] a,b;
wire s11,s12,s13,s14,s15,s22,s23,s24,s25,s26,s32,s33,s34,s35,s36,s37;
wire c11,c12,c13,c14,c15,c22,c23,c24,c25,c26,c32,c33,c34,c35,c36,c37;
wire [6:0] p0,p1,p2,p3;
assign p0 = a & {4{b[0]}};
assign p1 = a & {4{b[1]}};
assign p2 = a & {4{b[2]}};
```

```
assign p3 = a & {4{b[3]}};
assign p[0] = p0[0];
assign p[1] = s11;
assign p[2] = s22;
assign p[3] = s32;
assign p[4] = s34;
assign p[5] = s35;
assign p[6] = s36;
assign p[7] = s37;
//1st stage
ha ha11(s11,c11,p0[1],p1[0]);
fa fa12(s12,c12,p0[2],p1[1],p2[0]);
fa fa13(s13,c13,p0[3],p1[2],p2[1]);
fa fa14(s14,c14,p1[3],p2[2],p3[1]);
ha ha15(s15,c15,p2[3],p3[2]);
//2nd stage
ha ha22(s22,c22,c11,s12);
fa fa23(s23,c23,p3[0],c12,s13);
fa fa24(s24,c24,c13,c32,s14);
fa fa25(s25,c25,c14,c24,s15);
fa fa26(s26,c26,c15,c25,p3[3]);
//3rd stage
ha ha32(s32,c32,c22,s23);
ha ha34(s34,c34,c23,s24);
ha ha35(s35,c35,c34,s25);
ha ha36(s36,c36,c35,s26);
ha ha37(s37,c37,c36,c26);
endmodule
```

4-BIT BAUGHLY

```
module bm(p, a,b);
output [7:0] p;
input [3:0] a,b;
wire w0,w1,w2,w3,w4,w5,w6,w7,w8,w9,w10,w11,w12,w13,w14,w15,w16,w17,
w18, w19,w20,w21,w22,w23,w24,w25,w26,w27,w28,w29;
and g0 (w0, a[3], b[0]);
and g1 (w1, a[2], b[0]);
and g2 (w2, a[1], b[0]);
and g3 (p[0],a[0], b[0]);
and g4 (w6, a[3], b[1]);
and g5 (w3, a[2], b[1]);
and g6 (w4, a[1], b[1]);
and g7 (w5, a[0], b[1]);
and g8 (w12, a[3], b[2]);
and g9 (w13, a[2], b[2]);
and g10(w14, a[1], b[2]);
and g11(w15, a[O], b[2]);
and g12(w21, a[3], b[3]);
and g13(w22, a[2], b[3]);
```

```
and g14(w23, a[1], b[3]);
and g15(w24, a[0], b[3]);
fa_df f0 (w8, w7, w0, w3, 1'b0);
fa_df f1 (w10, w9, w1 , w4, 1'b0);
fa_df f2 (p[1], w11 , w2, w5, 1'b0);
fa_df f3 (w17 , w16 , w6 , w13, w7);
fa_df f4 (w19, w18 , w8 , w14, w9);
fa_df f5 (p[2], w20, w10, w15, w11);
fa_df f6 (w26 , w25, w12, w22, w16);
fa_df f7 (w28 , w27, w17, w23, w18);
fa_df f8 (p[3], w29 , w19, w24, w20);
fa_df f9 (p[6], p[7], w21, w25, w31);
fa_df f10(p[5], w31 , w26, w27, w30);
fa_df f11(p[4], w30 , w28, w29, 1'b0);
endmodule
module fa_df(output s,output cout, input a, input b,input cin);
wire z1,z2,z3;
xor(z1,a,b);
xor(s,z1,cin);
and(z2,z1,cin);
and(z3,a,b);
or(cout,z2,z3);
endmodule
```

16X4 RAM

```
module ram_16x4(data, RD,WR, Address, Output);
input[3:0]data;
input RD,WR;
input[3:0] Address;
output reg[3:0]Output;
reg[3:0]RAM[15:0];
reg[3:0]Temp;
always@(Address)begin
if(WR)
begin
RAM[Address]=data;Temp=RAM[Address];
end
if(RD)
begin
Output=RAM[Address];
end
end
endmodule
```

TESTBENCH

```
data=4'b0000; RD = 0;WR=1; Address= 4'b0000;#30;

data=4'b0001; RD = 0;WR=1; Address= 4'b0001;#30;
data=4'b0010; RD = 0;WR=1; Address= 4'b0010;#30;
```

```
data=4'b0011; RD = 0;WR=1; Address= 4'b0011;#30;
data=4'b0100; RD = 0;WR=1; Address= 4'b0100;#30;
data=4'b0101; RD = 0;WR=1; Address= 4'b0101;#30;
data=4'b0110; RD = 0;WR=1; Address= 4'b0110;#30;
data=4'b0111; RD = 0;WR=1; Address= 4'b0111;#30;
data=4'b1000; RD = 0;WR=1; Address= 4'b1000;#30;
data=4'b1001; RD = 0;WR=1; Address= 4'b1001;#30;
data=4'b1010; RD = 0;WR=1; Address= 4'b1010;#30;
data=4'b1011; RD = 0;WR=1; Address= 4'b1011;#30;
data=4'b1100; RD = 0;WR=1; Address= 4'b1100;#30;
data=4'b1101; RD = 0;WR=1; Address= 4'b1101;#30;
data=4'b1110; RD = 0;WR=1; Address= 4'b1110;#30;
data=4'b1111; RD= 0; WR =1;Address=4'b1111;#30;

#50;

RD = 1;WR=0; Address= 4'b0000;#30;
RD = 1;WR=0; Address= 4'b0001;#30;
RD = 1;WR=0; Address= 4'b0010;#30;
RD = 1;WR=0; Address= 4'b0011;#30;
RD =1; WR=0; Address= 4'b0100;#30;
RD = 1;WR=0; Address= 4'b0101;#30;
RD = 1;WR=0; Address= 4'b0110;#30;
RD = 1;WR=0; Address= 4'b0111;#30;
RD = 1;WR=0; Address= 4'b1000;#30;
RD = 1;WR=0; Address= 4'b1001;#30;
RD = 1;WR=0; Address= 4'b1010;#30;
RD = 1;WR=0; Address= 4'b1011;#30;
RD = 1;WR=0; Address= 4'b1100;#30;
RD = 1;WR=0; Address= 4'b1101;#30;
RD = 1;WR=0; Address= 4'b1110;#30;
RD = 1;WR=0; Address= 4'b1111;#30;
end
endmodule
```

16X4 ROM

```
module rom_16x4(Output, Address, RD);
output reg[3:0] Output;
input [3:0] Address;
input RD;
reg[3:0] ROM[15:0];
initial
begin
ROM[4'b0000]=4'b1111;
ROM[4'b0001]=4'b1110;
ROM[4'b0010]=4'b1101;
ROM[4'b0011]=4'b1100;
ROM[4'b0100]=4'b1011;
ROM[4'b0101]=4'b1010;
```

```
ROM[4'b0110]=4'b1001;
ROM[4'b0111]=4'b1000;
ROM[4'b1000]=4'b0111;
ROM[4'b1001]=4'b0110;
ROM[4'b1010]=4'b0101;
ROM[4'b1011]=4'b0100;
ROM[4'b1100]=4'b0011;
ROM[4'b1101]=4'b0010;
ROM[4'b1110]=4'b0001;
ROM[4'b1111]=4'b0000;
end
always@(RD , Address)
begin
if(RD)
begin
Output=ROM[Address];
end
end
endmodule
```

Testbench:-

```
initial begin

Address = 4'b0000; RD = 1; #50;
Address = 4'b0001; RD = 1; #50;
Address = 4'b0010; RD = 1; #50;
Address = 4'b0011; RD = 1; #50;
Address = 4'b0100; RD = 1; #50;
Address = 4'b0101; RD = 1; #50;
Address = 4'b0110; RD = 1; #50;
Address = 4'b0111; RD = 1; #50;
Address = 4'b1000; RD = 1; #50;
Address = 4'b1001; RD = 1; #50;
Address = 4'b1010; RD = 1; #50;
Address = 4'b1011; RD = 1; #50;
Address = 4'b1100; RD = 1; #50;
Address = 4'b1101; RD = 1; #50;
Address = 4'b1110; RD = 1; #50;
Address = 4'b1111; RD = 1; #50;
```

FIFO

```
module fifo_mem(data_out,fifo_full, fifo_empty, fifo_threshold, fifo_overflow,
fifo_underflow,clk, rst_n, wr,
rd, data_in);
input wr, rd, clk, rst_n;
input[7:0] data_in; // FPGA projects using Verilog/ VHDL
output[7:0] data_out;
output fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow;
```

```
wire[4:0] wptr,rptr;
wire fifo_we,fifo_rd;
write_pointer top1(wptr,fifo_we,wr,fifo_full,clk,rst_n);
read_pointer top2(rptr,fifo_rd,rd,fifo_empty,clk,rst_n);
memory_array top3(data_out, data_in, clk,fifo_we, wptr,rptr);
status_signal top4(fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow, wr,
rd, fifo_we, fifo_rd,
wptr,rptr,clk,rst_n);
endmodule
module memory_array(data_out, data_in, clk,fifo_we, wptr,rptr);
input[7:0] data_in;
input clk,fifo_we;
input[4:0] wptr,rptr;
output[7:0] data_out;
reg[7:0] data_out2[15:0];
wire[7:0] data_out;
always @(posedge clk)
begin
if(fifo_we)
data_out2[wptr[3:0]] <=data_in ;
end
assign data_out = data_out2[rptr[3:0]];
endmodule
module read_pointer(rptr,fifo_rd,rd,fifo_empty,clk,rst_n);
input rd,fifo_empty,clk,rst_n;
output[4:0] rptr;
output fifo_rd;
reg[4:0] rptr;
assign fifo_rd = (~fifo_empty)& rd;
always @(posedge clk or negedge rst_n)
begin
if(~rst_n) rptr <= 5'b000000;
else if(fifo_rd)
rptr <= rptr + 5'b000001;
else
rptr <= rptr;
end
endmodule
module status_signal(fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow, wr,
rd, fifo_we,
fifo_rd, wptr,rptr,clk,rst_n);
input wr, rd, fifo_we, fifo_rd,clk,rst_n;
input[4:0] wptr, rptr;
output fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow;
wire fbit_comp, overflow_set, underflow_set;
wire pointer_equal;
wire[4:0] pointer_result;
reg fifo_full, fifo_empty, fifo_threshold, fifo_overflow, fifo_underflow;
assign fbit_comp = wptr[4] ^ rptr[4];
assign pointer_equal = (wptr[3:0] - rptr[3:0]) ? 0:1;
```

```
assign pointer_result = wptr[4:0] - rptr[4:0];
assign overflow_set = fifo_full & wr;
assign underflow_set = fifo_empty&rd;
always @(*)
begin
fifo_full =fbit_comp & pointer_equal;
fifo_empty = (~fbit_comp) & pointer_equal;
fifo_threshold = (pointer_result[4]||pointer_result[3]) ? 1:0;
end
always @(posedge clk or negedge rst_n)
begin
if(~rst_n) fifo_overflow <=0;
else if((overflow_set==1)&&(fifo_rd==0))
fifo_overflow <=1;
else if(fifo_rd)
fifo_overflow <=0;
else
fifo_overflow <= fifo_overflow;
end
always @(posedge clk or negedge rst_n)
begin
if(~rst_n) fifo_underflow <=0;
else if((underflow_set==1)&&(fifo_we==0))
fifo_underflow <=1;
else if(fifo_we)
fifo_underflow <=0;
else
fifo_underflow <= fifo_underflow;
end
endmodule
module write_pointer(wptr,fifo_we,wr,fifo_full,clk,rst_n);
input wr,fifo_full,clk,rst_n;
output[4:0] wptr;
output fifo_we;
reg[4:0] wptr;
assign fifo_we = (~fifo_full)&wr;
always @(posedge clk or negedge rst_n)
begin
if(~rst_n) wptr <= 5'b000000;
else if(fifo_we)
wptr <= wptr + 5'b000001;
else
wptr <= wptr;
end
endmodule
```