

Answer Key

balsa code

Mel Chua, Jon Tse

December 18, 2005

1 gates.balsa

```
import [balsa.types.basic]

(--  
All of the following gates are Muller-C elements.  
They won't fire until they've gotten a transition on all of their inputs.  
They do this by doing a handshake on each of the input channels  
and storing the data in intermediary variables.  
Then the desired op is run on the intermediary variables, and another  
handshake is performed on the output channel to make sure the receiving  
module is ready for input.  
--)

(-- One bit AND gate --)
procedure gate_and_1 (input A : bit;
                      input B : bit;
                      output O : bit) is
    variable tmpA, tmpB : bit
begin
    loop
        A -> tmpA || B -> tmpB;
        O <- tmpA and tmpB
    end --Loop
end --gate_and_1

(-- One bit OR gate --)
procedure gate_or_1 (input A : bit;
                      input B : bit;
                      output O : bit) is
    variable tmpA, tmpB : bit
begin
    loop
        A -> tmpA || B -> tmpB;
        O <- tmpA or tmpB
    end --Loop
end --gate_or_1

(-- One bit NOT gate --)
procedure gate_not_1 (input A : bit;
                      output O : bit) is
    variable tmpA : bit
begin
    loop
        A -> tmpA;
        O <- not tmpA
    end --Loop
end --gate_not_1

(-- One bit XOR gate --)
procedure gate_xor_1 (input A : bit;
                      input B : bit;
                      output O : bit) is
    variable tmpA, tmpB : bit
begin
    loop
        A -> tmpA || B -> tmpB;
        O <- tmpA xor tmpB
    end --Loop
end --gate_xor_1
```

2 muxes.balsa

```
import [balsa.types.basic]
import [gates]

(--  
2 to 1 Multiplexer, 1 bit wide.  
  
This is a traditional 2->1 mux, implemented in "structural" balsa, using  
the gates defined in gates.balsa.  
--)  
procedure mux2_1 (input A : bit;  
                  input B : bit;  
                  input SEL : bit;  
                  output O : bit) is  
    channel notSEL, innermux0, innermux1 : bit  
begin  
    gate_not_1(SEL,notSEL) ||  
    gate_and_1(A,notSEL,innermux0) ||  
    gate_and_1(B,SEL,innermux1) ||  
    gate_or_1(innermux0,innermux1,O)  
end --procedure mux2_1  
  
(--  
2 to 1 Multiplexer, 32 bits wide  
  
This is a 32 bit wide mux. Instead of implementing it structurally,  
this was done in "behavioral" balsa. Balsa is a strongly typed language  
and doesn't allow for easy referencing of bit positions in a 32 bit number.  
--)  
  
procedure mux2x32_1x32 (input A : 32 bits;  
                        input B : 32 bits;  
                        input SEL : bit;  
                        output O : 32 bits) is  
    variable tmpA, tmpB : 32 bits  
    variable tmpSEL : bit  
begin  
    loop  
        A -> tmpA || B -> tmpB || SEL -> tmpSEL;  
        if tmpSEL = false then  
            O <- tmpA  
        else  
            O <- tmpB  
        end --if  
    end --loop  
end --procedure mux2x32_1x32  
  
(--  
procedure mux2x32_1x32 (input A : 32 bits;  
                        input B : 32 bits;  
                        input SEL : bit;  
                        output O : 32 bits) is  
    variable tmpA, tmpB, innermux0, innermux1 : 32 bits  
    variable tmpSEL, notSEL: bit  
begin  
    loop  
        A -> tmpA || B -> tmpB || SEL -> tmpSEL;  
        notSEL := not tmpSEL;  
        innermux0 := tmpA and notSEL || innermux1 := tmpB and tmpSEL;  
        O <- innermux0 or innermux1  
    end  
end  
--)  
  
--The following were our attempt to do this structurally. None of them compiled,  
--and all are in various states of disarray.  
  
(--  
procedure mux2x32_1x32 (input A : 32 bits;  
                        input B : 32 bits;  
                        input SEL : bit;  
                        output O : 32 bits) is  
begin  
    A -> tmpA || B -> tmpB;  
    arrA := (tmpA as array 32 of bit) || arrB := (tmpB as array 32 of bit);  
    chanA <- (tmpA as array 32 of bit);  
  
    chanA[0] <- arrA[0] || chanB[0] <- arrB[0] ||  
    chanA[1] <- arrA[1] || chanB[1] <- arrB[1] ||
```

```

chanA[2] <- arrA[2] || chanB[2] <- arrB[2] ||
chanA[3] <- arrA[3] || chanB[3] <- arrB[3] ||
chanA[4] <- arrA[4] || chanB[4] <- arrB[4] ||
chanA[5] <- arrA[5] || chanB[5] <- arrB[5] ||
chanA[6] <- arrA[6] || chanB[6] <- arrB[6] ||
chanA[7] <- arrA[7] || chanB[7] <- arrB[7] ||
chanA[8] <- arrA[8] || chanB[8] <- arrB[8] ||
chanA[9] <- arrA[9] || chanB[9] <- arrB[9] ||
chanA[10] <- arrA[10] || chanB[10] <- arrB[10] ||
chanA[11] <- arrA[11] || chanB[11] <- arrB[11] ||
chanA[12] <- arrA[12] || chanB[12] <- arrB[12] ||
chanA[13] <- arrA[13] || chanB[13] <- arrB[13] ||
chanA[14] <- arrA[14] || chanB[14] <- arrB[14] ||
chanA[15] <- arrA[15] || chanB[15] <- arrB[15] ||
chanA[16] <- arrA[16] || chanB[16] <- arrB[16] ||
chanA[17] <- arrA[17] || chanB[17] <- arrB[17] ||
chanA[18] <- arrA[18] || chanB[18] <- arrB[18] ||
chanA[19] <- arrA[19] || chanB[19] <- arrB[19] ||
chanA[20] <- arrA[20] || chanB[20] <- arrB[20] ||
chanA[21] <- arrA[21] || chanB[21] <- arrB[21] ||
chanA[22] <- arrA[22] || chanB[22] <- arrB[22] ||
chanA[23] <- arrA[23] || chanB[23] <- arrB[23] ||
chanA[24] <- arrA[24] || chanB[24] <- arrB[24] ||
chanA[25] <- arrA[25] || chanB[25] <- arrB[25] ||
chanA[26] <- arrA[26] || chanB[26] <- arrB[26] ||
chanA[27] <- arrA[27] || chanB[27] <- arrB[27] ||
chanA[28] <- arrA[28] || chanB[28] <- arrB[28] ||
chanA[29] <- arrA[29] || chanB[29] <- arrB[29] ||
chanA[30] <- arrA[30] || chanB[30] <- arrB[30] ||
chanA[31] <- arrA[31] || chanB[31] <- arrB[31];

mux2_1(chanA[0],chanB[0],SEL,chan0[0]) ||
mux2_1(chanA[1],chanB[1],SEL,chan0[1]) ||
mux2_1(chanA[2],chanB[2],SEL,chan0[2]) ||
mux2_1(chanA[3],chanB[3],SEL,chan0[3]) ||
mux2_1(chanA[4],chanB[4],SEL,chan0[4]) ||
mux2_1(chanA[5],chanB[5],SEL,chan0[5]) ||
mux2_1(chanA[6],chanB[6],SEL,chan0[6]) ||
mux2_1(chanA[7],chanB[7],SEL,chan0[7]) ||
mux2_1(chanA[8],chanB[8],SEL,chan0[8]) ||
mux2_1(chanA[9],chanB[9],SEL,chan0[9]) ||
mux2_1(chanA[10],chanB[10],SEL,chan0[10]) ||
mux2_1(chanA[11],chanB[11],SEL,chan0[11]) ||
mux2_1(chanA[12],chanB[12],SEL,chan0[12]) ||
mux2_1(chanA[13],chanB[13],SEL,chan0[13]) ||
mux2_1(chanA[14],chanB[14],SEL,chan0[14]) ||
mux2_1(chanA[15],chanB[15],SEL,chan0[15]) ||
mux2_1(chanA[16],chanB[16],SEL,chan0[16]) ||
mux2_1(chanA[17],chanB[17],SEL,chan0[17]) ||
mux2_1(chanA[18],chanB[18],SEL,chan0[18]) ||
mux2_1(chanA[19],chanB[19],SEL,chan0[19]) ||
mux2_1(chanA[20],chanB[20],SEL,chan0[20]) ||
mux2_1(chanA[21],chanB[21],SEL,chan0[21]) ||
mux2_1(chanA[22],chanB[22],SEL,chan0[22]) ||
mux2_1(chanA[23],chanB[23],SEL,chan0[23]) ||
mux2_1(chanA[24],chanB[24],SEL,chan0[24]) ||
mux2_1(chanA[25],chanB[25],SEL,chan0[25]) ||
mux2_1(chanA[26],chanB[26],SEL,chan0[26]) ||
mux2_1(chanA[27],chanB[27],SEL,chan0[27]) ||
mux2_1(chanA[28],chanB[28],SEL,chan0[28]) ||
mux2_1(chanA[29],chanB[29],SEL,chan0[29]) ||
mux2_1(chanA[30],chanB[30],SEL,chan0[30]) ||
mux2_1(chanA[31],chanB[31],SEL,chan0[31]);;

chan0[0] -> arr0[0] ||
chan0[1] -> arr0[1] ||
chan0[2] -> arr0[2] ||
chan0[3] -> arr0[3] ||
chan0[4] -> arr0[4] ||
chan0[5] -> arr0[5] ||
chan0[6] -> arr0[6] ||
chan0[7] -> arr0[7] ||
chan0[8] -> arr0[8] ||
chan0[9] -> arr0[9] ||
chan0[10] -> arr0[10] ||
chan0[11] -> arr0[11] ||
chan0[12] -> arr0[12] ||
chan0[13] -> arr0[13] ||
chan0[14] -> arr0[14] ||
chan0[15] -> arr0[15] ||
chan0[16] -> arr0[16] ||

```

```

chan0[17] -> arr0[17] ||
chan0[18] -> arr0[18] ||
chan0[19] -> arr0[19] ||
chan0[20] -> arr0[20] ||
chan0[21] -> arr0[21] ||
chan0[22] -> arr0[22] ||
chan0[23] -> arr0[23] ||
chan0[24] -> arr0[24] ||
chan0[25] -> arr0[25] ||
chan0[26] -> arr0[26] ||
chan0[27] -> arr0[27] ||
chan0[28] -> arr0[28] ||
chan0[29] -> arr0[29] ||
chan0[30] -> arr0[30] ||
chan0[31] -> arr0[31]
end --procedure mux2x32_1x32
--)

(--  

mux2_1(A[0],B[0],SEL,0[0]) ||
mux2_1(A[1],B[1],SEL,0[1]) ||
mux2_1(A[2],B[2],SEL,0[2]) ||
mux2_1(A[3],B[3],SEL,0[3]) ||
mux2_1(A[4],B[4],SEL,0[4]) ||
mux2_1(A[5],B[5],SEL,0[5]) ||
mux2_1(A[6],B[6],SEL,0[6]) ||
mux2_1(A[7],B[7],SEL,0[7]) ||
mux2_1(A[8],B[8],SEL,0[8]) ||
mux2_1(A[9],B[9],SEL,0[9]) ||
mux2_1(A[10],B[10],SEL,0[10]) ||
mux2_1(A[11],B[11],SEL,0[11]) ||
mux2_1(A[12],B[12],SEL,0[12]) ||
mux2_1(A[13],B[13],SEL,0[13]) ||
mux2_1(A[14],B[14],SEL,0[14]) ||
mux2_1(A[15],B[15],SEL,0[15]) ||
mux2_1(A[16],B[16],SEL,0[16]) ||
mux2_1(A[17],B[17],SEL,0[17]) ||
mux2_1(A[18],B[18],SEL,0[18]) ||
mux2_1(A[19],B[19],SEL,0[19]) ||
mux2_1(A[20],B[20],SEL,0[20]) ||
mux2_1(A[21],B[21],SEL,0[21]) ||
mux2_1(A[22],B[22],SEL,0[22]) ||
mux2_1(A[23],B[23],SEL,0[23]) ||
mux2_1(A[24],B[24],SEL,0[24]) ||
mux2_1(A[25],B[25],SEL,0[25]) ||
mux2_1(A[26],B[26],SEL,0[26]) ||
mux2_1(A[27],B[27],SEL,0[27]) ||
mux2_1(A[28],B[28],SEL,0[28]) ||
mux2_1(A[29],B[29],SEL,0[29]) ||
mux2_1(A[30],B[30],SEL,0[30]) ||
mux2_1(A[31],B[31],SEL,0[31])
--)

```

3 alu.balsa

```

import [balsa.types.basic]

(--  

Because balsa is strongly typed, we have to have a way of getting individual bits  

out of the various variables. Defining records is one way of doing this. A record  

can look like a continuous array of bit positions, but you can reference the sections  

you defined too.  

--)  

type addssplit is record
    bottom : 31 bits;
    top : bit
end

type addressresult31 is record
    result : 31 bits;
    carry : bit
end

type addressresult32 is record
    result : 32 bits;
    carry : bit
end

```

```

type alu_ctrl is record
    low : bit;
    high : bit
end

(--  

Traditional 32-bit full adder. Because adding two 32 bit numbers always carrys out either  

a one or a zero, balsa treats the result of an addition as a 33 bit number. By passing it  

to one of the records above, we can easily obtain the carry out bit, as show below.  

--)  

procedure fulladder_32 (input A : 32 bits; input B : 32 bits; input sub_flag : bit;  

                        output O : 32 bits; output carry_out : bit; output overflow_flag : bit) is
    variable tmpA, tmpB, xorB : 32 bits
    variable splitA, splitB : addsplit
    variable tmpSub_flag : bit
    variable arrSub : array 32 of bit
    variable i : byte
    variable result31 : addresult31
    variable result32 : addresult32
begin
    A -> tmpA || B -> tmpB || sub_flag -> tmpSub_flag;
    for || i in 0..31 then
        arrSub[i] := tmpSub_flag
    end
    ;
    xorB := tmpB xor (arrSub as 32 bits)
    ;
    splitA := (tmpA as addsplit) || splitB := (xorB as addsplit)
    ;
    result31 := (splitA.bottom + splitB.bottom + tmpSub_flag as addresult31) ||
    result32 := (tmpA + xorB + tmpSub_flag as addresult32)
    ;
    O <- result32.result ||
    carry_out <- result32.carry ||
    overflow_flag <- result32.carry xor result31.carry
end

(--  

Traditional 32-bit SLT.

Uses the full adder.
--)
procedure slt_32 (input A : 32 bits; input B : 32 bits; output O : bit) is
    channel sumout : 32 bits
    channel carry_out, overflow_flag : bit
    variable tmpOverflow_flag, tmpCarry_out : bit
    variable tmpSumout : 32 bits
    variable splitSumout : addsplit
begin
    fulladder_32(A,B,<- 1,sumout,-> tmpCarry_out,-> tmpOverflow_flag) ||
    sumout -> tmpSumout; splitSumout := (tmpSumout as addsplit);
    O <- splitSumout.top xor tmpOverflow_flag
end

(--  

32-bit ALU

Uses behavioral balsa instead of muxes because of syntactical problems.
--)

procedure alu_32 (input A : 32 bits; input B : 32 bits; input CTRL : 2 bits;  

                  output O : 32 bits; output carry_out : bit;  

                  output overflow_flag : bit; output zero_flag : bit) is

variable bufA, bufB, adder_out, xor_out, bufOut : 32 bits
variable bufCarry, bufOverflow, bufZero, slt_out : bit
variable bufCTRL : 2 bits
variable CTRLbits : alu_ctrl
begin
    A -> bufA || B -> bufB || CTRL -> bufCTRL;
    CTRLbits := (bufCTRL as alu_ctrl) || xor_out := bufA xor bufB;
    fulladder_32(<- bufA,<- bufB,<- CTRLbits.low,-> adder_out,-> bufCarry,-> bufOverflow)
    ||
    slt_32(<- bufA,<- bufB,-> slt_out);
    case bufCTRL of
        0 then bufOut := xor_out
        | 1 then bufOut := (slt_out as 32 bits)
        | 2..3 then bufOut := adder_out
    end
    ;
    if( bufOut = 0 ) then

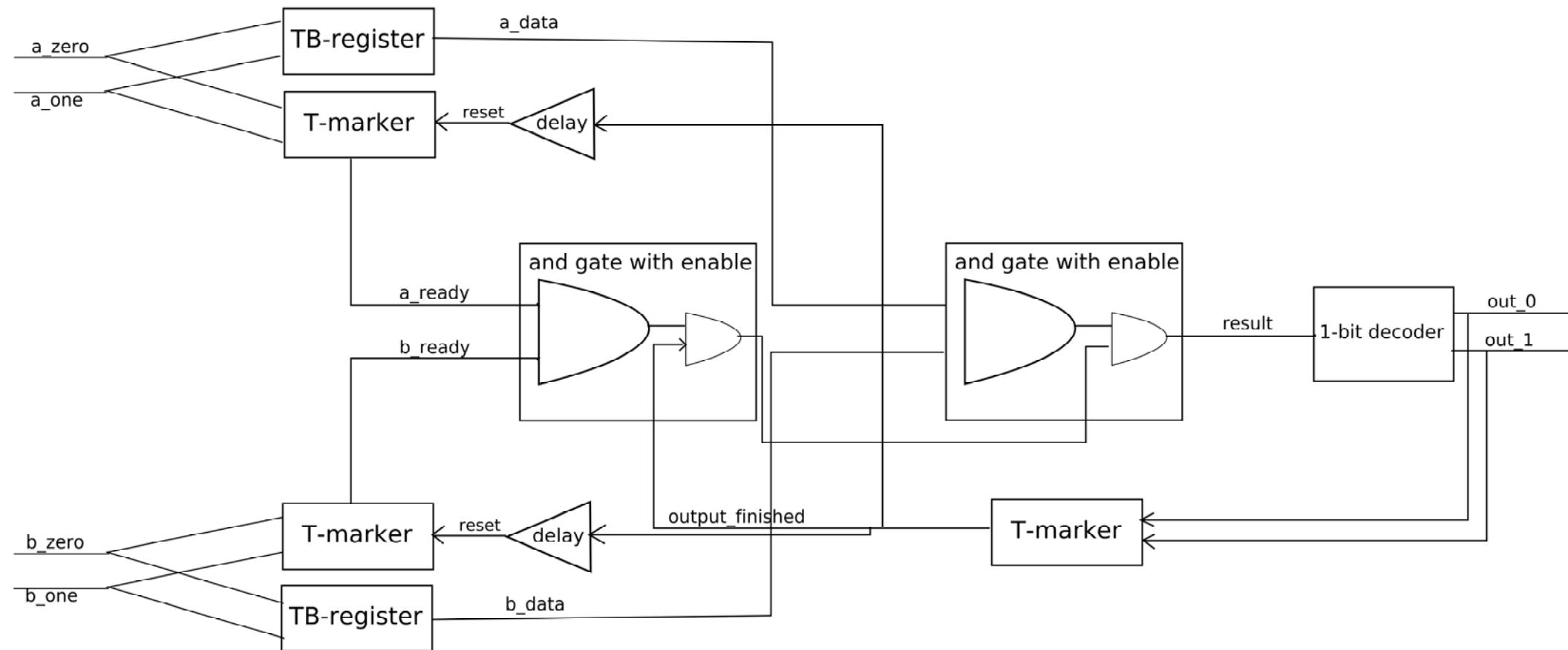
```

```
        bufZero := 1
else      bufZero := 0
end
;
0 <- bufOut || carry_out <- bufCarry || overflow_flag <- bufOverflow || zero_flag <- bufZero
end
```

jEdit - Untitled-1

```
1 /* Pseudocode for asynchronous AND-gate.
2 /*
3 /* Works by storing data in buffer registers until all components are ready
4 /* to transmit to the gate in question, as indicated by the T-markers.
5 /* Upon completion of a cycle, the output triggers a reset of all the T-markers
6 /* and tells the components to wait for the next round of inputs to be ready.
7 /* Uses two wires per bit for input and output (4 input, 2 output wires).
8 /* All components are standard except for the ones described below.
9 */
10
11 /*TB (transition-based) Register
12 /*Stores a 0 or 1 in outvalue, depending on which input wire has transitioned.
13 /*
14 inputs: in0, in1
15 outputs: outvalue
16 always @ edge(in0)
17   outvalue = 0
18 always @ edge(in1)
19   outvalue = 1
20
21
22 /* T (transition) Marker
23 /* Has the gate in question transitioned (finished with previous cycle) yet?
24 /* At any transition of any input, change has_transitioned to 1.
25 /* At reset, change has_transitioned to 0.
26 /*
27 inputs: in0, in1
28 outputs: has_transitioned
29 always @ edge(in0)
30   has_transitioned = 1
31 always @ edge(in1)
32   has_transitioned = 1
33 always @ edge(in0)
34   has_transitioned = 0
```

Asyncrhonous AND-gate



Note: all data flows from left to right unless otherwise indicated.