**LSU EE 4755** **Homework 5** Solution **Due: 7 November 2016**

**Problem 0:** This first problem provides background on the module used in this assignment. Please read the background and then solve the problems further below. The Verilog source can be found in directory hw05, however for this assignment there is no need to do anything with it.

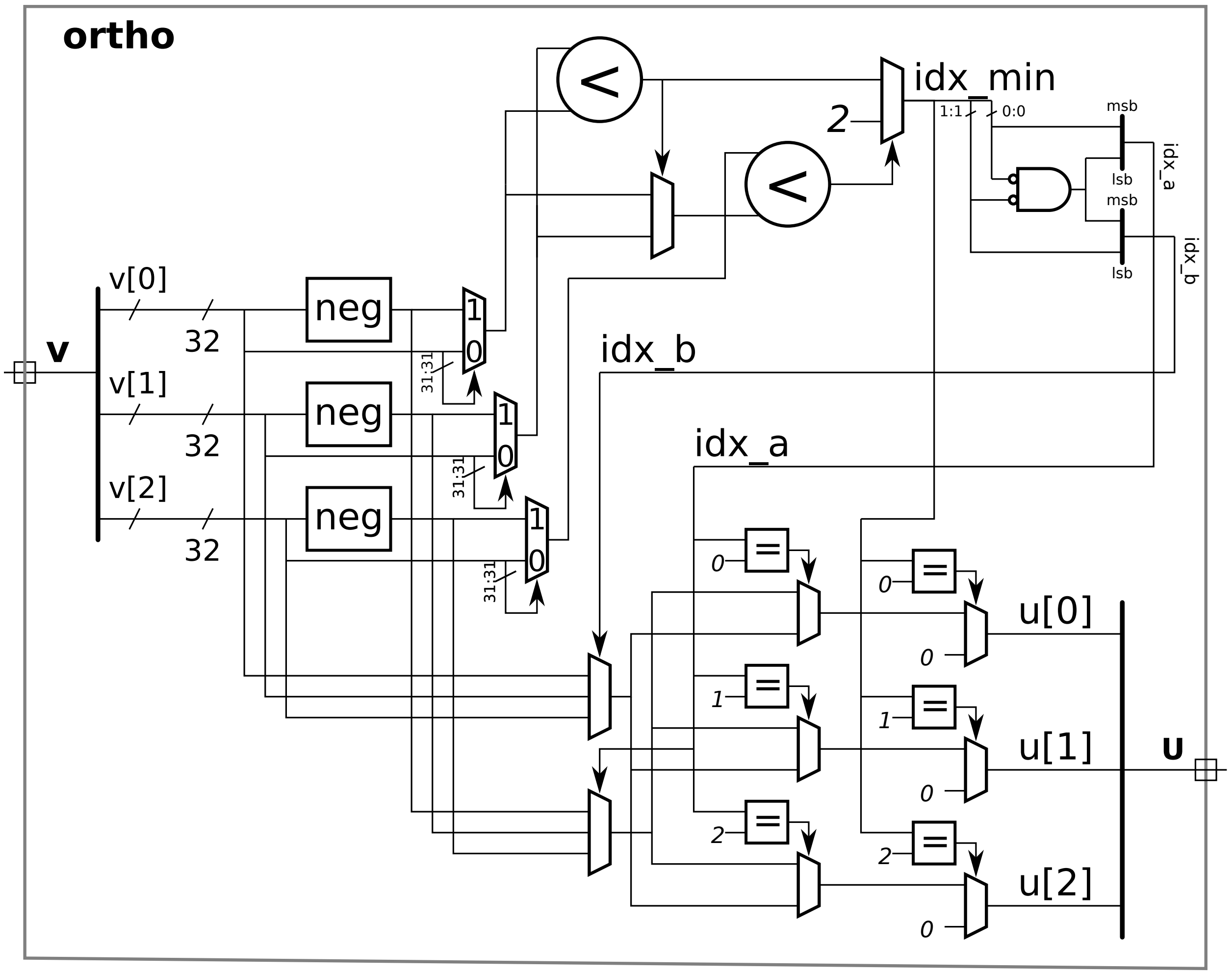
Module ortho has one input, v, a three-element vector of signed integers, and one output, u, also a three-element vector of signed integers. The output is computed so that u is orthogonal to v in the geometric sense. For those who are rusty on linear algebra, non-zero vectors $u$ and $v$ are orthogonal if $u \cdot v = 0$ or $u_x v_x + u_y v_y + u_z v_z = 0$. Using Verilog notation, u is computed so that u[0]*v[0]+u[1]*v[1]+u[2]*v[2]=0 and at least one element of u is not zero. It does so by finding the smallest element of v, setting the corresponding element in u to zero, swapping the to remaining two elements, and negating one of the two. For example, if $v = (4, 7, 55)$ then the module would set $u = (0, 55, -7)$.

```
module ortho #( int alternative = 1, int w = 32 )
   ( output logic signed [w-1:0] u [3],   input wire signed [w-1:0] v [3] );

   logic [1:0] idx_min, idx_a, idx_b;

   always_comb begin

      idx_min = 0;
      for ( int i=1; i<3; i++ ) if ( $abs(v[i]) < $abs(v[idx_min]) ) idx_min = i;

      idx_a = ( idx_min + 1 ) % 3;
      idx_b = ( idx_min + 2 ) % 3;

      if ( alternative == 1 ) begin

         // The loop below is a hint to synthesis program Cadence Encounter 14.28.
         for ( int i=0; i<3; i++ ) u[i] = 0;

         u[idx_min] = 0;
         u[idx_a] = v[idx_b];
         u[idx_b] = -v[idx_a];

      end else if ( alternative == 2 ) begin

         for ( int i=0; i<3; i++ )
           u[i] = idx_min == i ? 0 : idx_a == i ? v[idx_b] : -v[idx_a];

      end else $fatal(1);

   end

endmodule
```

**Important:** For all problems below in which hardware is shown:

- Clearly show inputs and outputs of `ortho`.
- Try to draw diagrams showing all hardware for `ortho` and refer to parts of the diagram in your answers below.

Complete solution appears below. See the problems for detail.

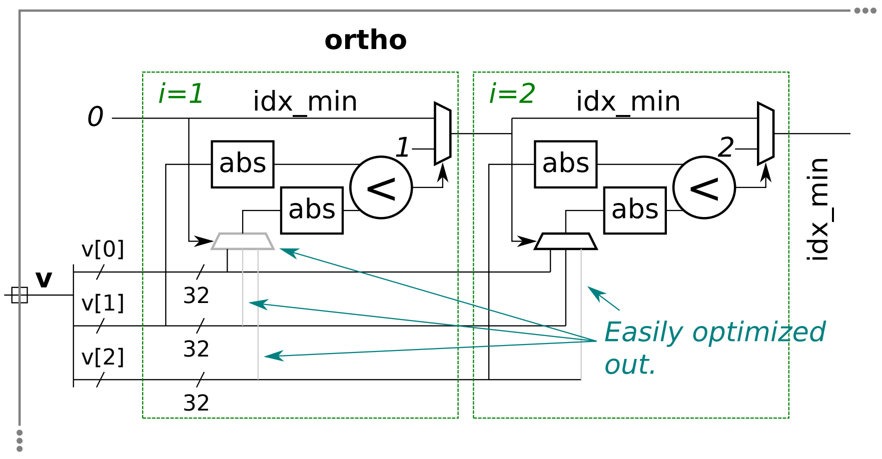


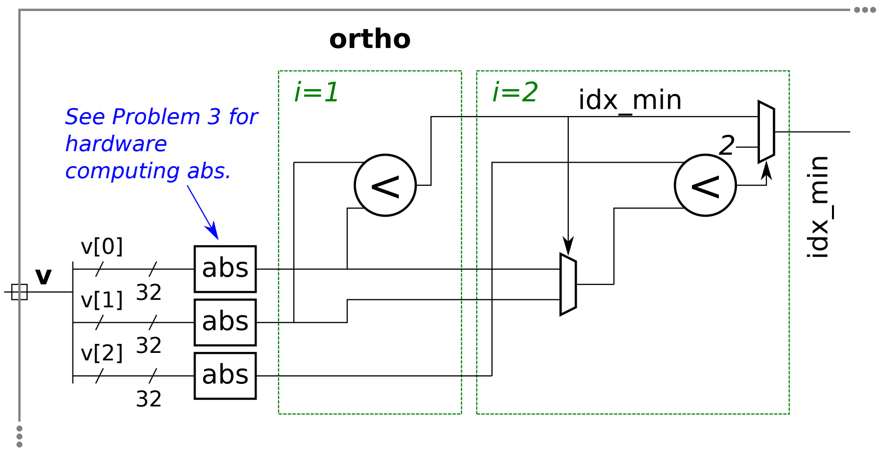
**Problem 1:** Consider the following part of the module:

```
idx_min = 0;
for ( int i=1; i<3; i++ )
  if ( $abs(v[i]) < $abs(v[idx_min]) ) idx_min = i;
```

(*a*) Show the hardware that will be synthesized for this fragment. (Please refer to the entire module when determining what will be synthesized.) Make reasonable optimizations. (See the next subproblem.) In this subpart show `abs` as a box.

Un-optimized and optimized solution appears below. In the un-optimized solution absolute value units appear at the output of the index operation multiplexors (the multiplexors implementing `v[idx_min]`), whereas in the optimized solution the absolute value is computed earlier. In the optimized version one index operation mux is removed entirely, in the other an input is eliminated. (See the midterm exam solution.) As shown in Problem 3, the absolute value hardware is shared with the hardware used for negation.





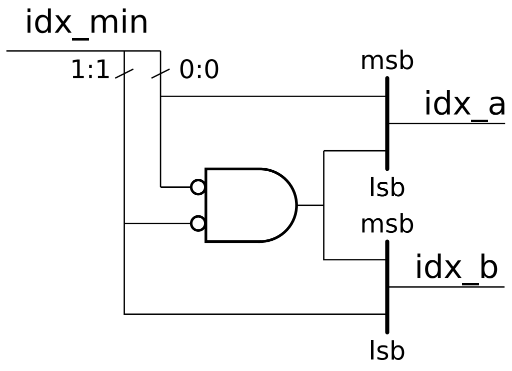
(*b*) The synthesis program synthesizes hardware that contains four absolute value units for this code, even with effort set to high. Explain why four is too many, perhaps by referring your own version that uses fewer absolute value units.

See the solution to the part above.

**Problem 2:** Consider the part of the module below: Show the hardware that will be synthesized for this code, taking into consideration that `idx_min` is two bits. *Hint: This is easy. Just consider all possible values of* `idx_min`.

```
idx_a = ( idx_min + 1 ) % 3;
idx_b = ( idx_min + 2 ) % 3;
```

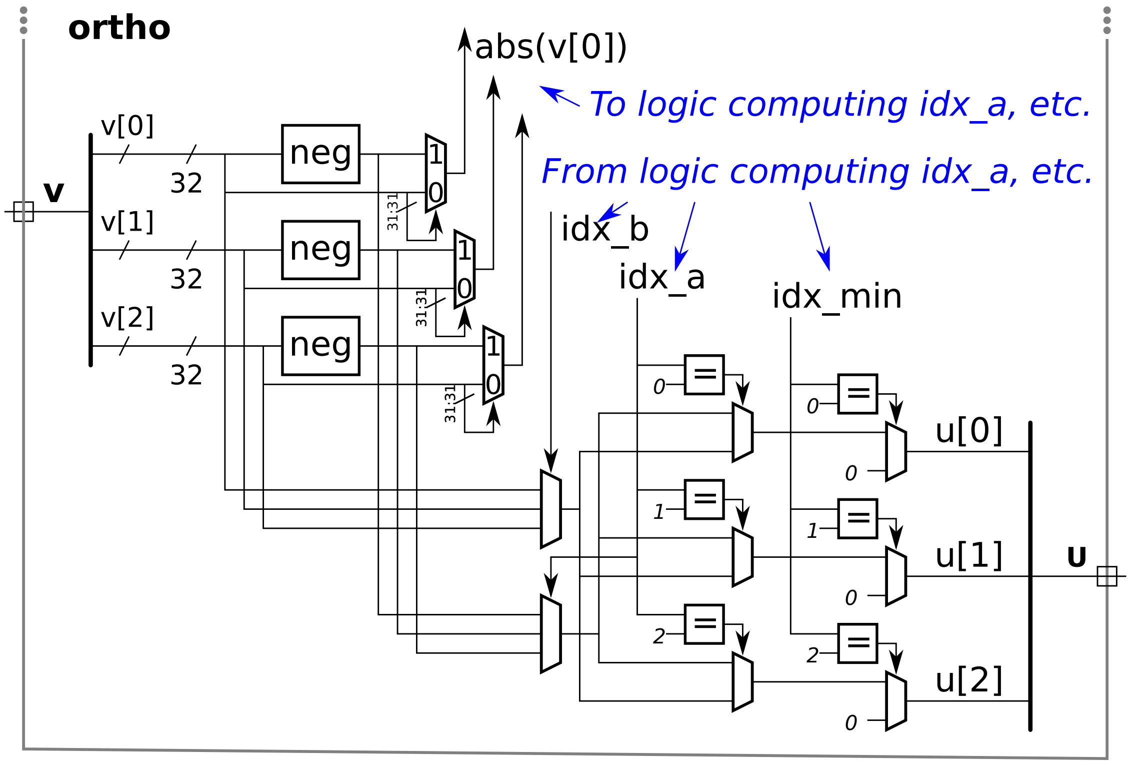
Solution appears below. The most important point is that there is no hardware to compute the remainder (modulo), which would be costly, nor are there adders. Drawing a truth table will show that only a single gate is needed.



**Problem 3:** Show the hardware that will be synthesized for the alternative 2 code, below, after optimization. As with the other problems, take into account the rest of the module. Look for opportunities to optimize `-v[idx_a]` taking advantage of hardware for `abs`.

```
    for ( int i=0; i<3; i++ )
      u[i] = idx_min == i ? 0 : idx_a == i ? v[idx_b] : -v[idx_a];
```

Solution appears below. Since this part needs negation (computing $-x$) and the hardware computing `idx_min` needs absolute value, which uses negation, this part computes the absolute value. Negation itself of a 2's complement value is computed by negating the bits and adding one. If the negated value were only needed for an adder, or adder-like hardware, then the adder could be eliminated.



**Problem 4:** As directed below, estimate the critical path in `ortho` for a $w$-bit instantiation. Do so using ripple-adder like implementations for absolute value, comparison, and negation. Use the performance model in which $n$-input AND and OR gates have delay $\lceil \lg n \rceil$ units.

(*a*) Find the critical path using the assumption that in hardware for an expression like $a + b < c$ the delay through the adder must be added to the delay through the comparison unit. The answer should be a function of $w$.

Solution appears in the diagram below in the upper timing number. See the last part for details.

(*b*) Find the critical path accounting for the fact that in ripple-like hardware for an expression like $a + b < c$ the low bits of the comparison can start as soon as the low bits of the sum are available. The answer should be a function of $w$.

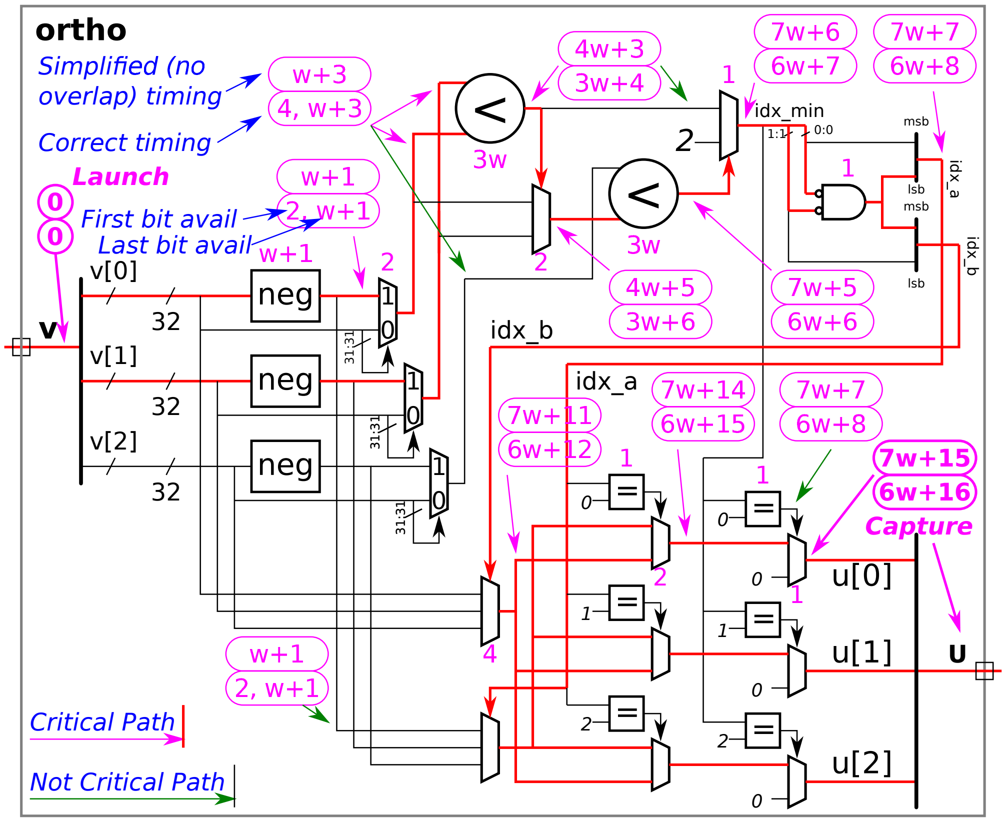
Solution appears in the diagram below in the lower timing number. See the last part for details.

(*c*) Show a sketch of the hardware with an arrow tracing the critical path through the hardware, from input to output. Annotating that arrow with intermediate delays will help in assigning partial credit.

The critical path appears in red in the figure below, the critical path (but not its length) is the same with both timing assumptions. The paired purple boxed numbers give the absolute time that the signal arrives at the labeled wire. The upper of the pair is under the assumption that one piece of ripple-like hardware must completely finish before a subsequent piece of ripple-like hardware can start. The lower number is computed under the correct assumption, that computation starts when data arrives. In the diagram this only affects the first comparison unit.

The delay of each component is shown as an unboxed purple number. The delay of the $\boxed{\text{neg}}$ unit is based on a ripple adder constructed with binary half adders. The carry chain consists only of AND gates.

Purple arrows point to wires carrying the critical path, green arrows point to non-critical wires.


ortho
Simplified (no overlap) timing
Correct timing
Launch
0
0
First bit avail
Last bit avail
w+3
4, w+3
w+1
2, w+1
w+1
v
v[0]
v[1]
v[2]
32
32
32
neg
neg
neg
2
1
0
31:31
3w
3w
<
<
2
4w+3
3w+4
4w+5
3w+6
7w+5
6w+6
2
1
idx_min
1:1
0:0
7w+6
6w+7
7w+7
6w+8
msb
lsb
msb
lsb
idx_a
idx_b
1
idx_b
idx_a
7w+11
6w+12
7w+14
6w+15
7w+7
6w+8
7w+15
6w+16
Capture
1
1
=
0
1
2
2
1
4
u[0]
u[1]
u[2]
U
w+1
2, w+1
Critical Path
Not Critical Path