Author: Daryl Posnett

ECS 154A - Lab 2 SS1 2020

Logistics

Submission

Due by 20:00 on Friday, 2020-07-17.

Turn in for the Logisim Evolution portion is on Gradescope. Submit the specified .circ files for each problem. The person submitting should specify their partner's name (if necessary) during the submission process.

Logisim Evolution, Grading, and Debugging

Need a reminder on how to download Logisim Evolution, how your circuits are autograded, or how to debug your circuits? Check the relevant sections of the `Lab 1 document`_.

Constraints

For these problems, unless specified otherwise, you must use designs relying on only the following:

- basic gates (NOT, AND, OR, NAND, NOR, XOR, XNOR)
- MUXes
- decoders
- the Logisim Evolution wiring library

Violating specified constraints will result in a 0 on the problem in question. While the autograder may give you credit even if you violate a constraint, we will check submissions after the due date and update grades appropriately. Note that for this lab you may not be warned of all of the autograder's grading criteria. Please read the instructions carefully and heed all constraints.

Logisim Evolution Problems [100]

1. Bit counting [10]

- Submission file for this part: *1.circ*
- Main circuit name: *bitcounting*
- Input pin(s): *twelvebitinput* [12]
- Output pin(s): *zeroes* [4]

Suppose we want to determine how many of the bits in a twelve-bit unsigned number are equal to zero. Implement the simplest circuit that can accomplish this task.

You may use any Logisim Evolution component for this problem except for the "bit adder." Note, however, your design must be combinational, you may not use any clocking or sequential solution.

2. Comparator implementation [20]

- Submission file for this part: *2.circ*
- Main circuit name: *comparison*
- Input pin(s): *inputa* [2], *inputb* [2]
- Output pin(s): *areequal* [1], *greaterthan* [1], *lessthan* [1]

Implement a 2-bit comparator.

Note that for greaterthan and lessthan, this means a>b, and a<b respectively. While already specified above, you may not use anything from the Arithmetic library for this problem as this defeats the purpose of the problem. Further, you may only use NAND gates to construct this circuit. I recommend doing the design first with standard gates (on paper) and then converting the design to NAND gates. For this problem your circuit must be minimal and efficient. I encourage you to think carefully about what is needed here. You must use kmaps to implement the basic comparison operations efficiently and you must also think about efficiency with respect to the whole circuit. Your goal will be to construct this circuit with the fewest number of NAND gates possible.

You must implement your comparator within the constraints specified for this lab.

3. Parity checker [5]

- Submission file for this part: *3.circ*
- Main circuit name: *parity*
- Input pin(s): *tenbitinput* [10]
- Output pin(s): *evenparity* [11]

Implement a simple even parity checker. Given a ten-bit number, output an 11th bit that ensures the total number of bits that are 1 is even. Concatenate this bit to the original number as the least significant bit.

There are parity gates for both type of parity. It defeats the purpose of this problem if you use those, so you may not use either of them. That said, it's possible to finish this problem using only a single gate.

4. 6-bit carry-lookahead unit [25]

- Submission file for this part: *4.circ*
- Main circuit name: *cla*
- Input pin(s): *inputa* [6], *inputb* [6], *carryin* [1]
- Output pin(s): *carryout* [6], *generator* [6], *propagator* [6]

Implement a 6-bit carry-lookahead unit (CLA). For the given *carryin* and each bit of the given inputs *inputa* and *inputb*, generate the relevant *carryout* bits. You will also need to output the corresponding *generator* and *propagator* bits.

Your CLA must be a true CLA. If your unit ripples the carry rather than calculating each carry based on the *generator* and *propagator* bits, you will get a 0. Your equations for C6 .. C1 should purely be in terms of C0, or *carryin*. (Note: In class we called carry in C-1, but that presents issues with logisim, so we use C0 here.) If a wire is feeding from your calculated C1 into your calculations for C2, or if you're duplicating the gates for C1 for use in C2, you're making a ripple-carry adder.

While already specified above, you may not use anything from the Arithmetic library for this problem.

5. 12-bit ALU [40]

- Submission file for this part: *5.circ*
- Main circuit name: *alu*
- Input pin(s): *inputa* [12], *inputb* [12], *operation* [3]
- Output pin(s): *aluout* [12], *overflow* [1], *carry* [1], *zero* [1]

I highly recommend that you finish the previous problem before starting this one.

Design a 12-bit ALU. Given the following input as the *operation* line, each bit cell of the ALU should perform the appropriate operation:

- 000 = AND
- 001 = OR
- 010 = XOR
- 011 = NOT A
- 100 = ADD (A + B)
- 101 = SUB (A - B)
- 110 = INC A
- 111 = DEC A

All arithmetic operations will be on 2's complement numbers. This only matters for the ADD/SUB/INC/DEC operations, since the others are performed bitwise. Note that you must use your six bit adder from the previous exercise to construct the 12 bit adder/subtractor for this exercise. Your 12 bit adder subtractor must be a "hybrid" circuit such that you make use of two 6 bit CLA units and combine them use ripple carry from one circuit into the other. If you create a pure ripple-carry adder instead of using your previous circuit, you will lose points.

Overflows are expected and you must indicate overflow by properly setting the overflow output bit for operations where overflow can occur. Similarly, you must set the carry output bit whenever the function has the potential to generate a carry. You do not have to do anything special in terms of interpreting carry for subtract operations, just make sure that the carry bit reflects the state of the ALU's carry bit when appropriate. Make sure that the carry bit is not set for operations that do not generate a carry. You must set the zero output bit whenever the result of an operation is equal to zero. This must work for all operations where it is appropriate.

The basic design of this ALU should be bit-parallel, that is, you will create bit-serial modules and combine them in a bit-parallel datapath. To be clear, you will have a 12 bit adder/subtractor and 12 bit logic modules. You must use a 12 bit MUX to route the outputs of these modules.

Your NOT inplementation must make use of the XOR logic circuit and your INC/DEC operations must make use of ADD/SUB. Thus your output MUX will only have four (12 bit) inputs, AND, OR, XOR, and ADD/SUB. This adds complexity to your mapper and various control circuits, but that is the point of this exercise. You will lose points if you do not heed these instructions.

I highly recommend the use of subcircuits to aid in testing and debugging. Create your 12 bit adder, then an adder subtractor, the logic circuits AND, OR, XOR, then combine these to make your ALU.

Note: your controller circuits must be minimal and efficient. You should use kmaps to make sure that this is the case. You will lose points if you attempt to brute force the controllers.

While already specified above, you may not use anything from the Arithmetic library for this problem. You will get a 0 if you use the built-in adder or subtractor; create the logic for those operations using gates.