Approaching Project 2 – System Design and Using Verilog

(8-bit) 16-to-1 multiplexer

S3

S2

S1

S0

switches[3:0]

switches[1]

switches[0]

switches[3]

switches[2]

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

Out

? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?

LEDS[7:0]

Datapath

Memory

ENABLE (KEY[1])

CLEAR (KEY[0])

operandB[7:0]

operandA[7:0]

opcode[3:0]

result[7:0]

status[3:0]

It should go without saying that understanding Verilog will go a long way in understanding how to do Project 2, but understanding the language might not help you to understand the structure of the circuit you are trying to make, and what its place is in the larger system where it is doing its work. However, once you understand the larger system and what the datapath does in that system, I think that certain elements of the language will make more sense to you.

Your goal in this assignment is to make a digital system called a datapath. I have shown a block diagram of the entire system. The diagram contains a block called a datapath. For the time being, you should ignore the “Memory” unit that is supplying values to the datapath, and the 16-to-1 multiplexer that is looming ominously at the bottom of the diagram.

A datapath is a circuit that performs operations of various kinds (arithmetic, logic, shift, *etc*.) on operands. Different operations require different numbers of operands. For example, an addition operation to require two operands to serve as the addends, while an increment operation only requires one operand, since “increment” implies that we are always adding the same constant value to the operand. Our datapath has two 8-bit operands, one or both might represent the values that a particular operation will use.

A datapath is designed to perform multiple operations, so providing operands to the datapath does not provide enough information for the datapath to “know” what operation it should be doing. To “know” what operation it should be doing, it needs another piece of information. We call this piece of information an opcode. Each opcode uniquely identifies one of the datapath’s operation. Together, the opcode and the operands provide the datapath with all of the information it needs to produce a correct result.

Suppose that we were designing a simple datapath capable of performing four operations: addition, subtraction, bitwise-AND, and bitwise-OR. Observe that since the datapath performs four operations, its opcodes have to have at least two bits, since two bits of information is the smallest number that allows us to distinguish four cases: 00, 01, 10, 11. Which opcode performs which operation depends on how we design the datapath, but for this example, imagine that we have assigned the opcodes as follows:

|  |  |
| --- | --- |
| Opcode | Operation |
| 00 | Addition (A + B) |
| 01 | Subtraction (A – B) |
| 10 | Bitwise-AND (A AND B) |
| 11 | Bitwise-OR (A OR B) |

Suppose that the datapath operates on 4-bit operands. Note that 10 bits of information are needed to specify an operation and its operands. For example, imagine that the 10 bits form a “word” having the following layout:

|  |  |  |
| --- | --- | --- |
| Opcode (2 bits) | Operand A (4 bits) | Operand B (4 bits) |

Any 10-bit word should be enough for the datapath to provide a result. In effect, the 10-bit word is a rudimentary “computer instruction.” Using the format described above

* The word 00 0101 0010 (spaces inserted for convenience) means “ADD +5 to +2”. The result should be 0111.
* The word 01 0011 1110 means “SUBTRACT -2 from +3”. (Notice that we are using signed 2’s complement operands.) The result should be 0101.
* The word 10 1110 0111 means “AND 1110 and 0111”. The result should be 0110.
* The word 11 1010 0110 means “OR 1010 and 0110”. The result should be 1110.

I have not provided any details as to how this datapath should perform these operations, but since all of these operations either exist in Verilog or can be described in terms of other operations that exist in Verilog, we can write enough “code” to implement the logic that performs each operation. If we were to build a “circuit” that performed each of the four operations, the job of the opcode would be to choose which of the operations gets to be the final output of the circuit. *What kind of circuit element does that job?*

As the block diagram indicates, our datapath uses 8-bit operands and uses a 4-bit opcode. This means that there will be certain differences between your datapath and the one in the previous example:

* Each of the operations that your datapath has to accept 8-bit operands instead of 4-bit operands (as the example showed).
* Each of the operations must (in general) produce an 8-bit result instead of a 4-bit result.
* Each of the operations must receive a unique 4-bit opcode instead of a unique 2-bit code.
* I haven’t discussed this (I did include it in the block diagram though) but your datapath must also produce a 4-bit “status” output as described in the project specification.

With all of that being said, you should still be able to describe a “circuit” that performs each of the operation, and then use the opcode (and another “circuit”) to choose which of the operations gets to be the final output of the circuit.

But this is the point where the limitations of our DE0-Nano board start to create complexities for us…

Our Nano board only has four switches, which isn’t nearly enough to simultaneously supply a 4-bit opcode and two 8-bit operands. This is why the memory is present in our system. The memory’s job will be to hold the 20-bit words that represent the “instructions” of our datapath. Each address of the memory holds one 20-bit value that represents an opcode and two (potential) operands. (Remember, not every instruction requires two operands, but each “instruction” holds the values, even if the operation doesn’t use both.) By pressing KEY1, the memory advances by one address; this means that each time you press KEY1, the datapath is receiving a new “instruction” word. Pressing KEY0, resets the memory; this means that the memory returns to its initial address, which causes the datapath to see the first “instruction” that the memory contains.

*To test your datapath, you will eventually have to modify the text file that represents the memory’s contents. Each 5-digit hexadecimal value represents a 20-bit binary value; the first hex digit is the 4-bit opcode, the second and third hex digits are the 8-bit operand A, and the fourth and fifth hex digits are the 8-bit operand B.*

Using a memory solves the problem of not having enough input switches to manipulate the datapath, but this isn’t the only problem we have. Our Nano board only has 8 LEDs on it. We can only look at one 8-bit value at a time, but there are many 8-bit values that we might want to see at any given time. Of course, we might want to look at the result of an operation. But we might want to see either of the operands, or the opcode, or the value of the status output. Since we only have enough LEDs to look at one of them at a time, we can use a multiplexer to decide which one of them will appear on the LEDs at any given time. *The complete system that you implement will actually use the switches on your Nano board to manipulate this output multiplexer.*

You should therefore view your design process as follows:

1. Design the *datapath*: Use Verilog to describe the workings of a circuit that accepts a 4-bit opcode and two 8-bit operands and produces an 8-bit result that is consistent with the opcode and operands. Your datapath might consist of multiple modules, each of which is responsible for some hardware task or function.
2. Place the *datapath* in the *system*: Use Verilog syntax to instantiate your datapath in the top-level module, and to connect the correct sets of signals to the inputs of the multiplexers, so that manipulating the switches will in turn manipulate which value we see on the LEDs. There is a table in the specification that tells you which value has to be visible on the LEDs for each switch combination; this should guide you in making connections to the multiplexer inputs.

You should view your testing process as follows – *this is important, because your simulation should literally be a simulation of the way that you will physically manipulate your board*:

1. Press KEY0 to reset the memory.
2. With the first 20-bit instruction on the datapath, change the switch values to view the various system values of interest. In particular, view the result to make certain that your datapath works.
3. Press KEY1 to advance the memory to the next 20-bit instruction.
4. Repeat Step 2 for the current instruction.
5. Repeat Steps 3 and 4 for each instruction you want to test.

A VERILOG FREEBIE!

I am posting an assignment to give you a chance to practice your Verilog design and to prepare you for how to use Quartus in Project 2. In class, I mentioned that we will be able to perform operations on vectors in a “vector-like” manner.

Suppose that I want to perform the bitwise-OR of the operands, as described in the initial example. I could create a module whose only job is to perform this operation:

module or\_function\_structural(A, B, or\_result);

input [3:0] A, B;

output [3:0] or\_result;

or or1(or\_result[3], A[3], B[3]),

or1(or\_result[2], A[2], B[2]),

or1(or\_result[1], A[1], B[1]),

or1(or\_result[0], A[0], B[0]);

endmodule

In structural Verilog, we have to describe one gate for every one-bit binary operation we want to perform.

In dataflow Verilog, we can use single operators to describe the manner in which multi-bit operations occur. All of the bitwise-operators that exist as structural gates also exist as dataflow operators. (Many other dataflow operators exist, as we will see.)

So here is the same module, re-written to use continuous assignment and dataflow operators:

module or\_function\_dataflow(A, B, or\_result);

input [3:0] A, B;

output [3:0] or\_result;

assign or\_result = A | B;

endmodule

The vertical bar, or pipe symbol (|) is the dataflow symbol for the bitwise-OR operator. Since A, B, and or\_result have the same number of bits, the dataflow operation performs the OR operation on corresponding bits of A and B – A[3] with B[3], A[2] with B[2], and so forth – and places the result of each bitwise operation into the corresponding bit of or\_result.

As you might imagine, this is ***much*** easier than using structural Verilog to do the same thing. In fact, you might wonder what the point is to making an entire module for one line of Verilog code is. Really, there is no point. We could line up a datapath’s worth of operations:

assign some\_result = …

assign some\_other\_result = …

assign still\_another\_result = …

assign yet\_another\_result = …

Then, with all of the individual results “calculated” we could deal with the problem of how to obtain the final result. It might not come as a surprise to you that we could use another dataflow operation with continuous assignment to derive the final result:

assign final\_result = …

The final result should be a function of the intermediate results. Here is something to guide your thinking:

module dataflow\_example (A, B, C, F);

input [7:0] A, B, C;

output [7:0] F;

(something is missing here!)

assign notA = ~A; // notA is the bitwise complement of A.

assign product1 = notA & B // product1 is the AND of notA and B.

assign product2 = A & C // product2 is the AND of A and C.

assign F = product1 | product2 // The output is the OR of product1 and product2

endmodule

That is, we have computed the logic function F = A’B + AC – and not just for individual bit values of A, B, and C, but for 8-bit vectors A, B, and C!

*How should notA, product1, and product2 be incorporated into the module?*

Time for you to have some fun with all of this!