Episode 6.10 – Demultiplexers

Welcome to the Geek Author series on Computer Organization and Design Fundamentals. I'm David Tarnoff, and in this series, we are working our way through the topics of Computer Organization, Computer Architecture, Digital Design, and Embedded System Design. If you're interested in the inner workings of a computer, then you're in the right place. The only background you'll need for this series is an understanding of integer math, and if possible, a little experience with a programming language such as Java. And one more thing. Due to the computational nature of this episode, you might want to visit the transcript page found at intermation.com to download the episode worksheet.

Welcome to the Geek Author series on Computer Organization and Design Fundamentals. I'm David Tarnoff, and in this series, we are working our way through the topics of Computer Organization, Computer Architecture, Digital Design, and Embedded System Design. If you're interested in the inner workings of a computer, then you're in the right place. The only background you'll need for this series is an understanding of integer math, and if possible, a little experience with a programming language such as Java. And one more thing. Due to the computational nature of this episode, you might want to visit the transcript page found at intermation.com to download the episode worksheet.

In Episode 6.09 – Multiplexers, we designed a circuit that would select one digital stream from a group of inputs and connect it to the single output of the circuit. In this episode, we're going to flip that around and take a single input signal and route it to a selected channel from a group of output channels. The inputs to this circuit are simple. Like the multiplexer, there are n select lines labeled S0, S1, S2, and so on. The number associated with each select line identifies the power of two position it holds in a binary number. That binary number identifies a channel number. For example, if S2 equals one, S1 equals one, and S0 equals zero, we've selected channel six, 1-1-0 in binary. In the case of the demultiplexer, we use these select lines to identify the output to which the data will be routed. After the select lines, there is only one additional input, a single data line containing the data to be sent to the selected output channel.

The simplest demultiplexer has two inputs: a single select bit, S0, and a single data bit, D. Since the select bit can take on one of two values, zero or one, there are two output lines, Y0 and Y1, to which the data can be routed. If the select bit, S0, equals zero, Y0 follows the values that appear on D, in other words, D is routed to Y0, while Y1 remains inactive. If S0 equals one, Y1 follows the values that appear on D, while Y0 remains inactive. This demultiplexer configuration is referred to as a one-to-two demultiplexer.

The term "inactive" here does not mean that the outputs that haven't been selected don't contain a one or a zero. For an active-high circuit, one where we drive the output with AND gates, the inactive state is zero. For an active-low circuit, one where we drive the output with NAND gates, the inactive state is one.

A one-to-four demultiplexer requires two select bits, S1 and S0, to route the single data input to one of four outputs, Y0, Y1, Y2, or Y3. The decimal equivalent of the binary pattern found on the bits S1 and S0 identify the output selected. If S1 and S0 both equal zero, the circuit routes the data to Y0, while Y1, Y2, and Y3 take on their inactive levels. If S1 and S0 equal zero and one respectively, the circuit routes the

data to Y1, while the other outputs are inactive. S1 and S0 equal to one and zero respectively routes the data to Y2. Finally, both S1 and S0 equal to one route the data to Y3.

Now for our design. For a demultiplexer with n select bits, there will be 2^n outputs. This means that there will be 2^n separate circuits, one driving each output. Don't worry, though, they'll be simple. Remember that if the decimal digit represented by the n select bits equals some value, k for instance, then the output Y_k equals the value at the data input, D. All we need to do is determine when the select bits represent k, and then pass the data bit, D, through to the Y_k output. This sounds a bit like the decoder circuits we discussed in Episode 6.08. Those little devices had one job – watch for a specific pattern of ones and zeros and output an active signal when they find it. The only difference here is that we will be replacing the active signal with D. And how did we implement decoders? Why, with a single AND or NAND gate.

Let's build one of these demultiplexers, specifically, an active-high one-to-eight demultiplexer with three select bits, S2, S1, and S0. Now remember, active-high circuits output logic zeros in the inactive state, so if the output we're designing for is not selected to output the data stream, then it should output a zero. That said, we implement active-high decoder circuits using AND gates.
When the three select bits, S2, S1, and S0, all equal zero, we want to route the data to the output Y0. Let's describe this in a sentence. Y0 will output a one if S2 equals zero and S1 equals zero and S0 equals zero and the data, D, equals one. If all three select bits equal zero and the data bit equals zero, the output follows D, and is a zero. Because we're trying to design an active-high circuit, when the select bits are anything other than zeros for this circuit, we want the output to go inactive and output a zero. It sounds like the only time we want to output a one is if the data bit equals one and the select bits all equal zero. Just like the decoder circuit, this is an AND gate, the Boolean expression being S2-bar AND S1-bar AND S0-bar AND D.

If we want to route the data to Y1, the select bits need to be 0-0-1. In this case, we want to output a one if S2 equals zero and S1 equals zero and S0 equals one and D equals one. Otherwise, we output a zero. This makes the Boolean expression for our Y1 circuit S2-bar AND S1-bar AND S0 AND D.

A pattern is emerging here for our active-high one-to-eight demultiplexer. For the output Y_k to equal our data input, we need to place a set of inverters at the inputs to our AND gate that identify when the select bits represent k. Add D as an additional input to our AND gate, and we have the Y_k circuit for our demultiplexer. For example, to represent five with our three select bits, we need the pattern 1-0-1. Since the middle bit, S1, is the only one that needs to be zero, it's the only one that gets inverted. This gives us our Y5 circuit for our active-high one-to-eight demultiplexer: S2 AND S1-bar AND S0 AND D.
What about the circuits for the active-low demultiplexer? Remember that it is the AND operations that decide when to make the circuit active. Replacing the AND gate with the NAND gate makes an active-low circuit. Going back to our Y5 circuit, if we replace the AND gate with a NAND gate, then the output will be active, a logic zero, only when S2 is a one, S1 is a zero, S0 is a one, and D is a one. This is close to what we want, but not quite. Notice that the output is the inverse of D, but we want to output D. That means that before D goes into our NAND gate, we need to invert it. That's it, just invert D before it goes into the NAND gate.

In our next episode, we're going to step away from digital circuitry, and get back to binary operations on the computer. For episode transcripts, worksheets, links, or other podcast notes, please visit us at intermation.com where you will also find links to our Instagram, Twitter, Facebook, and Pinterest pages.

Until the next episode, remember that while the scope of what makes a computer is immense, it's all just ones and zeros.

Welcome to the Geek Author series on Computer Organization and Design Fundamentals. I'm David Tarnoff, and in this series, we are working our way through the topics of Computer Organization, Computer Architecture, Digital Design, and Embedded System Design. If you're interested in the inner workings of a computer, then you're in the right place. The only background you'll need for this series is an understanding of integer math, and if possible, a little experience with a programming language such as Java. And one more thing. Due to the computational nature of this episode, you might want to visit the transcript page found at intermation.com to download the episode worksheet.

In Episode 6.09 – Multiplexers, we designed a circuit that would select one digital stream from a group of inputs and connect it to the single output of the circuit. In this episode, we're going to flip that around and take a single input signal and route it to a selected channel from a group of output channels. The inputs to this circuit are simple. Like the multiplexer, there are n select lines labeled S0, S1, S2, and so on. The number associated with each select line identifies the power of two position it holds in a binary number. That binary number identifies a channel number. For example, if S2 equals one, S1 equals one, and S0 equals zero, we've selected channel six, 1-1-0 in binary. In the case of the demultiplexer, we use these select lines to identify the output to which the data will be routed. After the select lines, there is only one additional input, a single data line containing the data to be sent to the selected output channel.

The simplest demultiplexer has only two inputs: a single select bit, S0, and a single data bit, D. Since the select bit can take on one of two values, zero or one, there are two output lines, Y0 and Y1, to which the data can be routed. If the select bit, S0, equals zero, Y0 follows the values that appear on D, in other words, D is routed to Y0, while Y1 remains inactive. If S0 equals one, Y1 follows the values that appear on D, while Y0 remains inactive. This demultiplexer configuration is referred to as a one-to-two demultiplexer.

The term "inactive" here does not mean that Y1 doesn't contain a one or a zero. For an active-high circuit, one where we drive the output with AND gates, the inactive state is zero. For an active-low circuit, one where we drive the output with NAND gates, the inactive state is one.

A one-to-four demultiplexer requires two select bits, S1 and S0, to route the single data input to one of four outputs, Y0, Y1, Y2, or Y3. The decimal equivalent of the binary pattern found on the bits S1 and S0 identify the output selected. If S1 and S0 both equal zero, the circuit routes the data to Y0, while Y1, Y2, and Y3 take on their inactive levels. If S1 and S0 equal zero and one respectively, the circuit routes the data to Y1, while the other outputs are inactive. S1 and S0 equal to one and zero respectively routes the data to Y2. Finally, both S1 and S0 equal to one route the data to Y3.

Now for our design. For a demultiplexer with n select bits, there will be 2^n outputs. This means that there will be 2^n separate circuits, one driving each output. Don't worry, though, they'll be simple. Remember that if the decimal digit represented by the n select bits equals some value, k for instance, then the output Y_k equals the value at the data input, D. All we need to do is determine when the select bits represent k, and then pass the data bit, D, through to the Y_k output. This sounds a bit like the decoder circuits we discussed in Episode 6.08. Those little devices had one job – watch for a specific pattern of ones and zeros and output an active signal when they find it. The only difference here is that

we will be replacing the active signal with D. And how did we implement decoders? Why, with a single AND or NAND gate.

Let's build one of these demultiplexers, specifically, an active-high one-to-eight demultiplexer with three select bits, S2, S1, and S0. Now remember, active-high circuits output logic zeros in the inactive state, so if the output we're designing for is not selected to output the data stream, then it should output a zero. That said, we implement active-high decoder circuits using AND gates.

When the three select bits, S2, S1, and S0, all equal zero, we want to route data to the output Y0. Let's describe this in a sentence. Y0 will output a one if S2 equals zero and S1 equals zero and S0 equals zero and the data, D, equals one. If all three select bits equal zero and the data bit equals zero, the output follows D, and is a zero. Because we're trying to design an active-high circuit, when the select bits are anything other than zeros for this circuit, we want the output to go inactive and output a zero. It sounds like the only time we want to output a one is if the data bit equals one and the select bits all equal zero. Just like the decoder circuit, this is an AND gate, the Boolean expression being S2-bar AND S1-bar AND S0-bar AND D.

If we want to route the data to Y1, the select bits need to be 0-0-1. In this case, we want to output a one if S2 equals zero and S1 equals zero and S0 equals one and D equals one. Otherwise, we output a zero. This makes the Boolean expression for our Y1 circuit S2-bar AND S1-bar AND S0 AND D.

A pattern is emerging here for our active-high one-to-eight demultiplexer. For the output Y_k to equal our data input, we need to place a set of inverters at the inputs to our AND gate to identify when the select bits represent k. Add D as an additional input to our AND gate, and we have the Y_k circuit for our demultiplexer. For example, to represent five with our three select bits, we need the pattern 1-0-1. Since the middle bit, S1, is the only one that needs to be zero, it's the only one that gets inverted. This gives us our Y5 circuit for our active-high one-to-eight demultiplexer: S2 AND S1-bar AND S0 AND D. What about the circuits for the active-low demultiplexer? Remember that it is the AND operations that decides when to make the circuit active. Replacing the AND gate with the NAND gate makes an active-low circuit. Going back to our Y5 circuit, if we replace the AND gate with a NAND gate, then the output will be active, a logic zero, only when S2 is a one, S1 is a zero, S0 is a one, and D is a one. This is close to what we want, but not quite. Notice that the output is the inverse of D, but we want to output D. That means that before D goes into our NAND gate, we need to invert it. That's it, just inverted D before it goes into the NAND gate.

In our next episode, we're going to step away from digital circuitry, and get back to binary operations on the computer. For episode transcripts, worksheets, links, or other podcast notes, please visit us at intermation.com where you will also find links to our Instagram, Twitter, Facebook, and Pinterest pages. Until the next episode, remember that while the scope of what makes a computer is immense, it's all just ones and zeros.