

COMP 122/L Week 6 Part 2

Kyle Dewey

Outline

- Exploiting *don't cares* in Karnaugh maps
- Multiplexers
- Arithmetic Logic Units (ALUs)

Exploiting *Don't Cares* in Karnaugh-Maps

Don't Cares

- Occasionally, a circuit's output will be unspecified on a given input
 - Occurs when an input's value is invalid
- In these situations, we say the output is a *don't care*, marked as an X in a truth table

Example: Binary Coded Decimal

- Occasionally, it is convenient to represent decimal numbers directly in binary, using 4-bits per decimal digit
 - For example, a digital display



Example: Binary Coded Decimal

- Not all binary values map to decimal digits

Binary	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

Binary	Decimal
1000	8
1001	9
1010	X
1011	X
1100	X
1101	X
1110	X
1111	X

Significance

- Recall that in a K-map, we can only group 1s
- Because the value of a *don't care* is irrelevant, we can treat it as a 1 if it is convenient to do so (or a 0 if that would be more convenient)

Example

- A circuit that calculates if the binary coded decimal input $\% 2 == 0$

Example

- A circuit that calculates if the binary coded decimal input $\% 2 == 0$

I_3	I_2	I_1	I_0	R
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	1
0	1	1	1	0

I_3	I_2	I_1	I_0	R
1	0	0	0	1
1	0	0	1	0
1	0	1	0	X
1	0	1	1	X
1	1	0	0	X
1	1	0	1	X
1	1	1	0	X
1	1	1	1	X

Example

As a K-map

		$I_1 I_0$			
		00	01	11	10
$I_3 I_2$	00	1	0	1	0
	01	1	0	1	0
	11	X	X	X	X
	10	1	0	X	X

Example

If we don't exploit *don't cares*...

		$I_1 I_0$			
		00	01	11	10
$I_3 I_2$	00	1	0	1	0
	01	1	0	1	0
	11	X	X	X	X
	10	1	0	X	X

Example

If we **do** exploit *don't cares*...

		$I_1 I_0$			
		00	01	11	10
$I_3 I_2$	00	1	0	1	0
	01	1	0	1	0
	11	X	X	X	X
	10	1	0	X	X

Example

If we **do** exploit *don't cares*...

$$R = !I_1!I_0 + I_1I_0$$

		I_1I_0			
		00	01	11	10
I_3I_2	00	1	0	1	0
	01	1	0	1	0
	11	X	X	X	X
	10	1	0	X	X

Multiplexers

Motivation

- At this point, you've seen a lot of straightline circuits
- However, this doesn't quite match up with respect to what a processor does. Why?

Motivation

- At this point, you've seen a lot of straightline circuits
- However, this doesn't quite match up with respect to what a processor does. Why?
 - We don't always do the same thing - it depends on the instruction
 - What do we need here?

Motivation

- At this point, you've seen a lot of straightline circuits
- However, this doesn't quite match up with respect to what a processor does. Why?
 - We don't always do the same thing - it depends on the instruction
 - What do we need here?
 - Some form of a conditional

Conditional

- Assume `selector`, `A`, `B`, and `R` all hold a single bit
- How can we implement this using what we have seen so far? (Hint: what does the truth table look like?)

$$R = (\text{selector}) ? A : B$$

R = (selector) ? A : B

S	A	B	R
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

$R = (\text{selector}) ? A : B$

S	A	B	R
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Unreduced sum-of-products:

$$R = \neg S \neg A B + \neg S A B + S \neg A \neg B + S A B$$

$R = (\text{selector}) ? A : B$

S	A	B	R
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Unreduced sum-of-products:

$$R = \neg S \neg A B + \neg S A B + S A \neg B + S A B$$

Reduced sum-of-products:

$$R = \neg S B + S A$$

Slight Modification

Original

`R = (selector) ? A : B`

Modified

`R = (selector) ? doThis() : doThat()`

Slight Modification

Original

$$R = (\text{selector}) ? A : B$$

Modified

$$R = (\text{selector}) ? \text{doThis}() : \text{doThat}()$$

Intended semantics: either `doThis()` or `doThat()` is executed. Our formula from before doesn't satisfy this property:

$$R = !S * \text{doThat}() + S * \text{doThis}()$$

Slight Modification

Original

`R = (selector) ? A : B`

Modified

`R = (selector) ? doThis() : doThat()`

- Fixing this is hard, but possible
- Involves circuitry we'll learn later
- Oddly enough, this isn't as big of a problem as it seems, and it's ironically *faster* than doing just one or the other. Why?

Slight Modification

Original

`R = (selector) ? A : B`

Modified

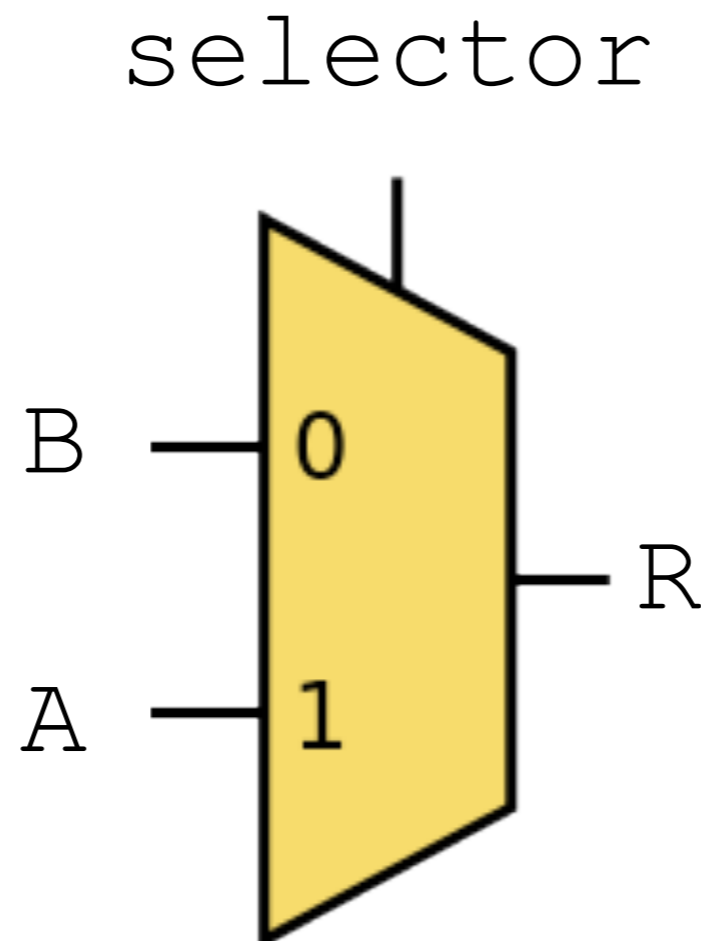
`R = (selector) ? doThis() : doThat()`

- Oddly enough, this isn't as big of a problem as it seems, and it's ironically *faster* than doing just one or the other. Why? - branches executed in parallel at the hardware level. Faster because extra circuitry is extra.

Multiplexer

- Component that does exactly this:

$$R = (\text{selector}) ? A : B$$



Question

- Recall the arithmetic logic unit (ALU), which is used to add, subtract, shift, perform bitwise operations, etc.
- How might a multiplexer be useful for an ALU?

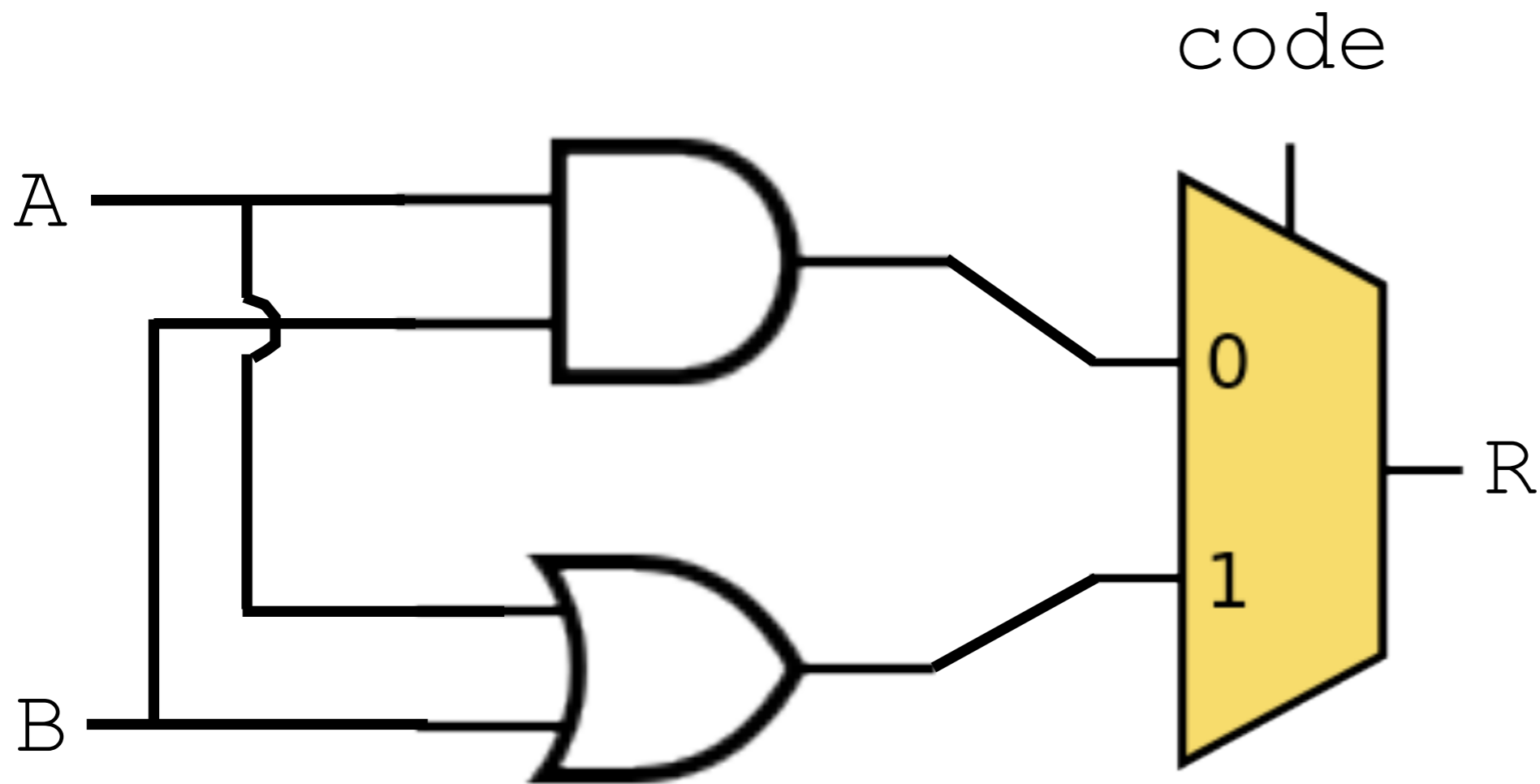
Question

- Recall the arithmetic logic unit (ALU), which is used to add, subtract, shift, perform bitwise operations, etc.
- How might a multiplexer be useful for an ALU? - Do all operations at once in parallel, and then use a multiplexer to select the one you want

Example

- Let's design a one-bit ALU that can do bitwise AND and bitwise OR
- It has three inputs: A , B , and S , along with one output R
- S is a code provided indicating which operation to perform; 0 for AND and 1 for OR

Example

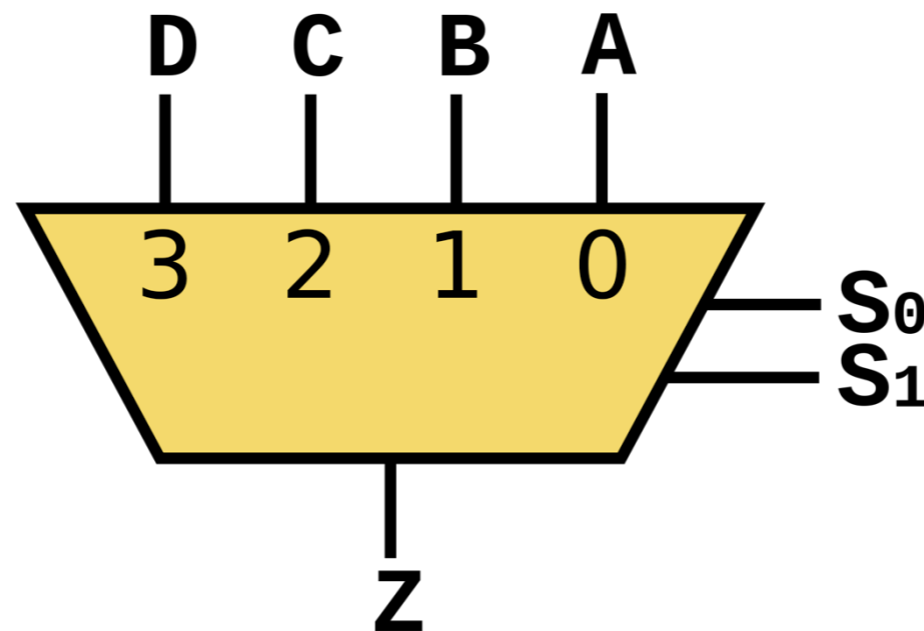


Bigger Multiplexers

- Can have a multiplexer with more than two inputs
- Need multiple select lines in this case
- Question: how many select lines do we need for a 4 input multiplexer?

Bigger Multiplexers

- Can have a multiplexer with more than two inputs
- Need multiple select lines in this case
- Question: how many select lines do we need for a 4 input multiplexer? - 2. Values of different lines essentially encode different binary integers.



Bigger Multiplexers

- We can build up bigger multiplexers from 2-input multiplexers. How?