CS64 Week 6 Lecture 1

Kyle Dewey

Overview

- Tail call optimization
- Introduction to circuits
- Digital design: single bit adders
- Circuit minimization
  - Boolean algebra
  - Karnaugh maps
  - Exploiting *don't cares*

More Recursion

- What's special about the following recursive function?

```
int recFac(int n, int accum) {
  if (n == 0) {
    return accum;
  } else {
    return recFac(n - 1, n * accum);
  }
}
```

More Recursion

- What's special about the following recursive function?
  - It is *tail recursive* - with the right optimization, uses constant stack space
  - We can do this in assembly - `tail_recursive_factorial.asm`

```
int recFac(int n, int accum) {
  if (n == 0) {
    return accum;
  } else {
    return recFac(n - 1, n * accum);
  }
}
```

Dispelling the Magic: Circuits

Why Binary?

- Very convenient for a circuit
  - Two possible states: on and off
  - `0` and `1` correspond to on and off

Relationship to Bitwise Operations

- You’re already familiar with bitwise OR, AND, XOR, and NOT
- These same operations are fundamental to circuits
    - Basic building blocks for more complex things

Single Bits

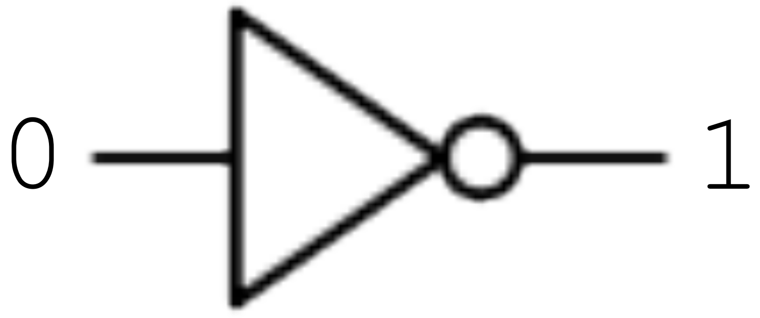
- For the moment, we will deal only with individual bits
- Later, we'll see this isn't actually that restrictive

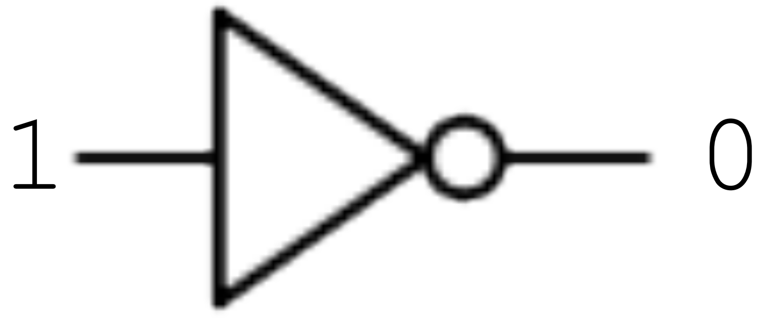
Operations on Single Bits: AND

Operations on Single Bits: OR

Operations on Single Bits: XOR

Operations on Single Bits: NOT



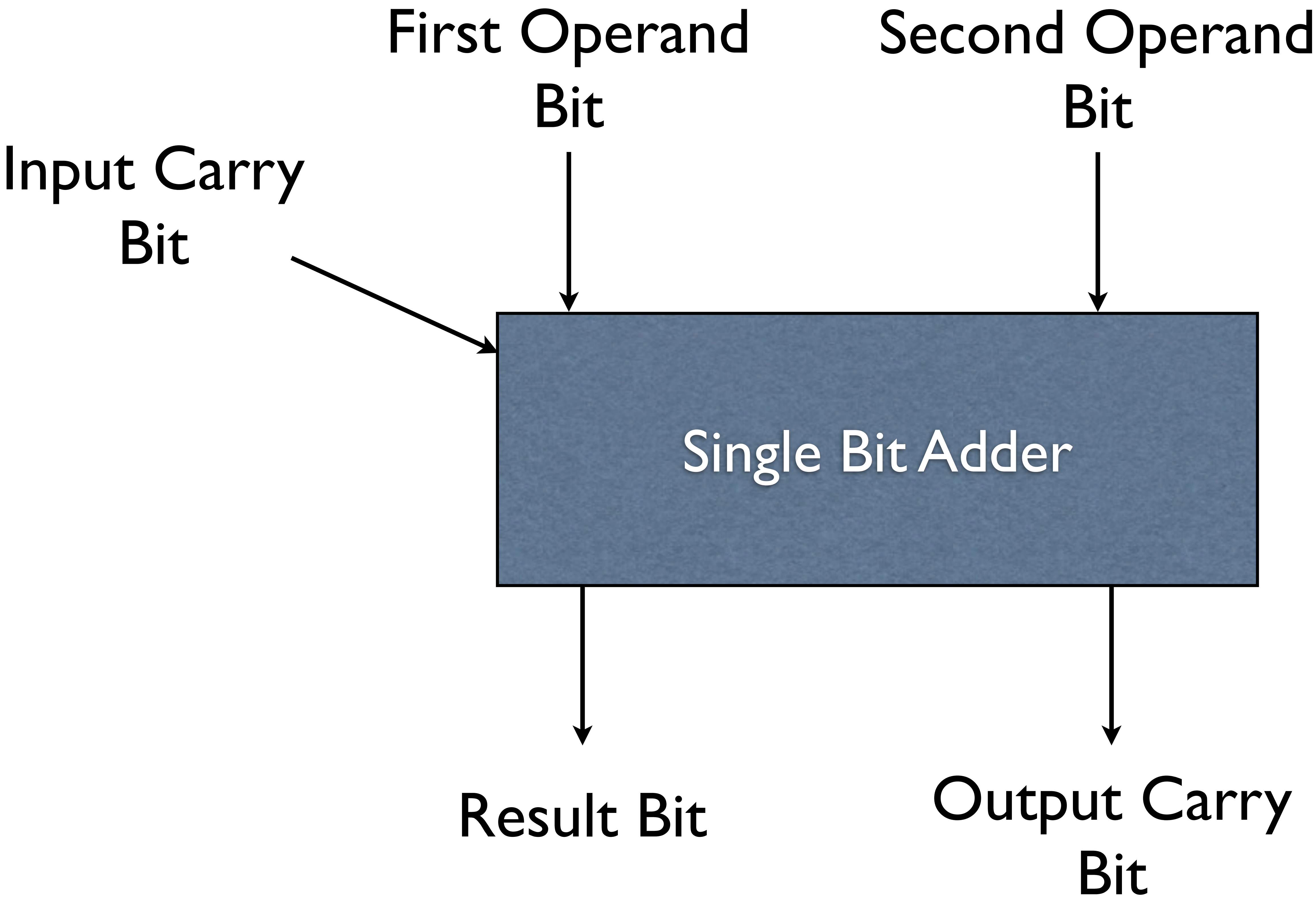


Recall: Addition

Addition with Single Bits

0 0 +0 -- 0	0 0 +1 -- 1	0 1 +0 -- 1	0 1 +1 -- 0 Carry: 1
1 0 +0 -- 1	1 0 +1 -- 0 Carry: 1	1 1 +0 -- 0 Carry: 1	1 1 +1 -- 1 Carry: 1

In Summary



First Operand Bit

Second Operand Bit

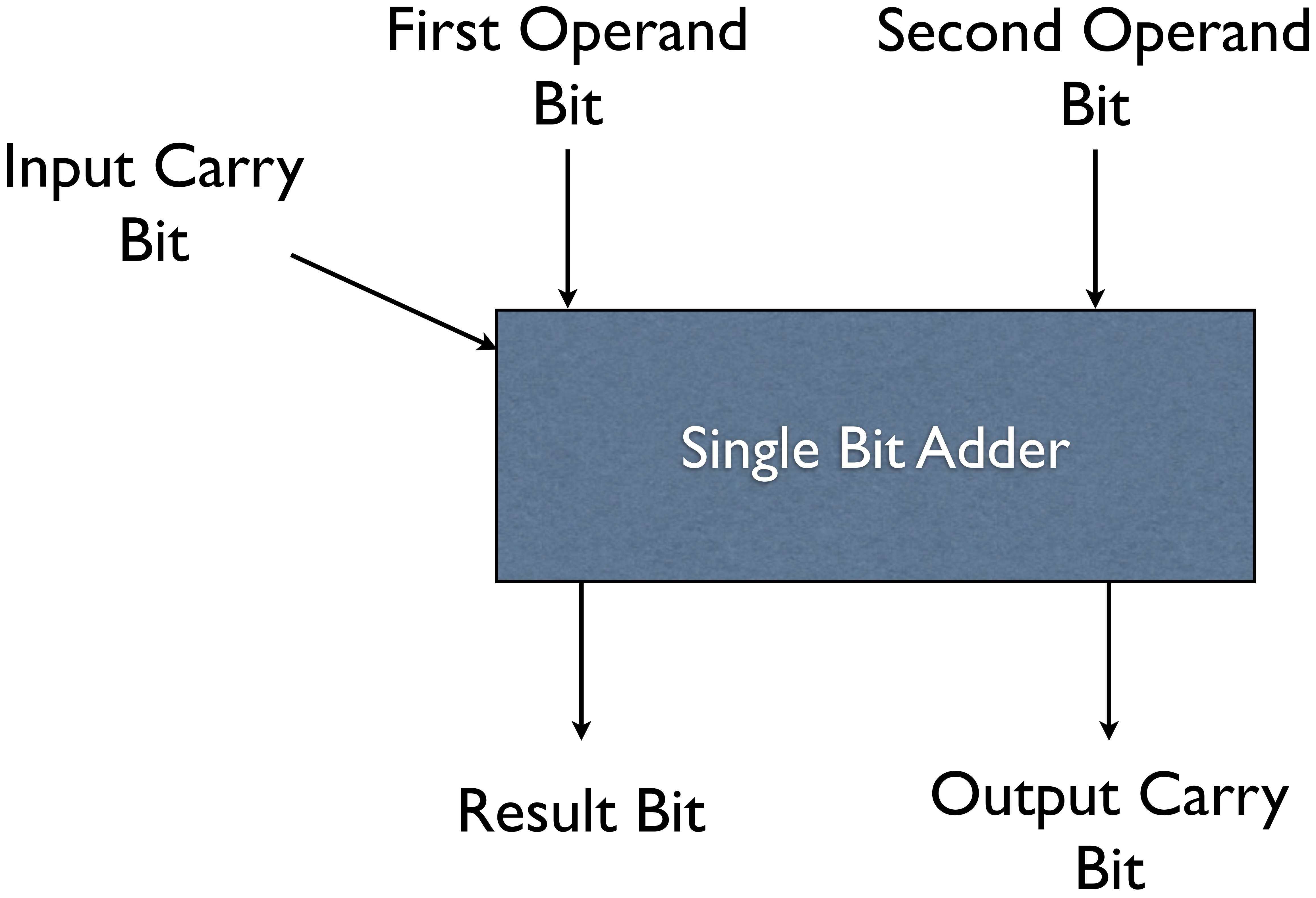
Input Carry Bit

Single Bit Adder

Result Bit

Output Carry Bit

- How can we adapt this to add multi-digit binary numbers together?



First Operand Bit

Second Operand Bit

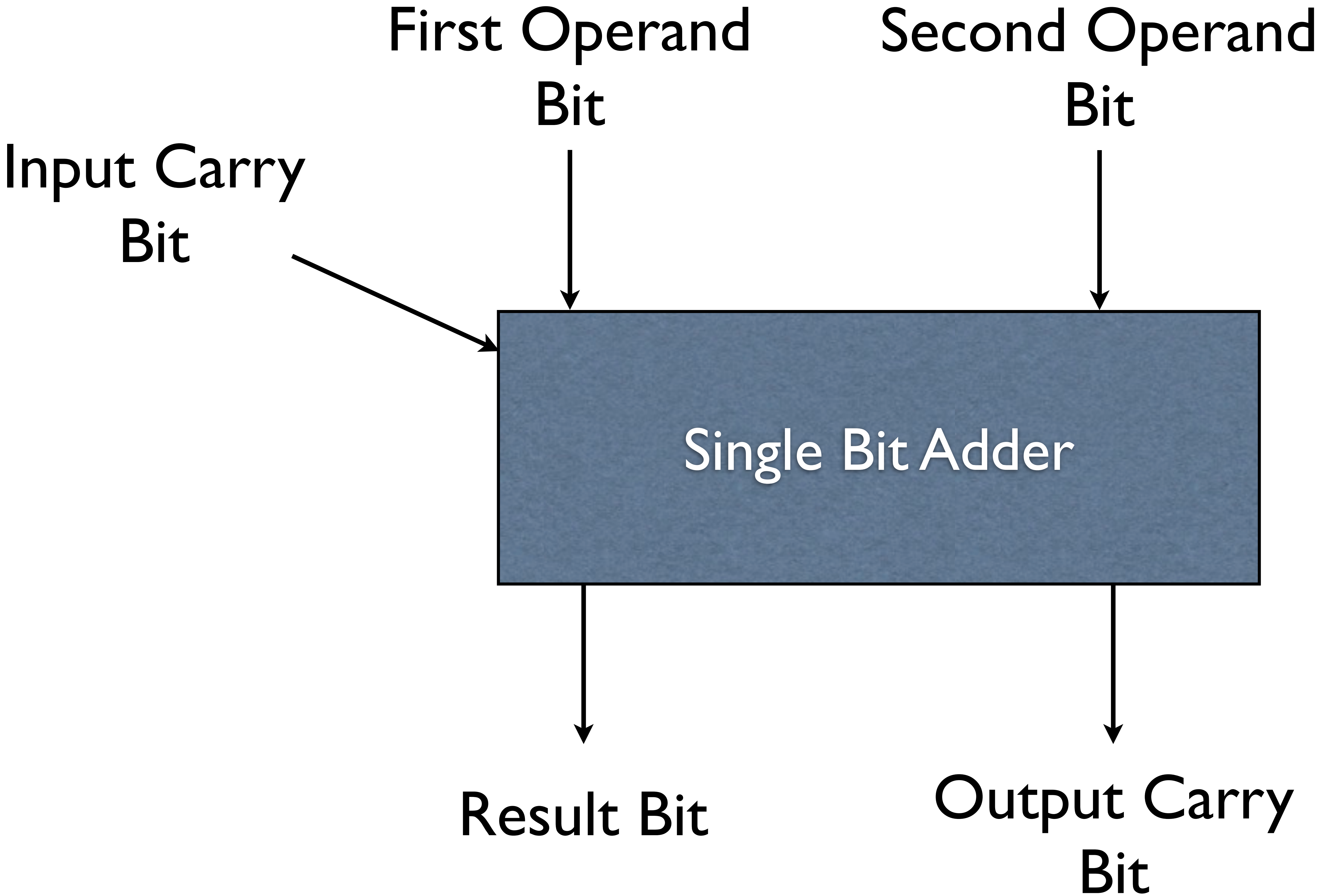
Input Carry Bit

Single Bit Adder

Result Bit

Output Carry Bit

Putting it Together



First Operand Bit

Second Operand Bit

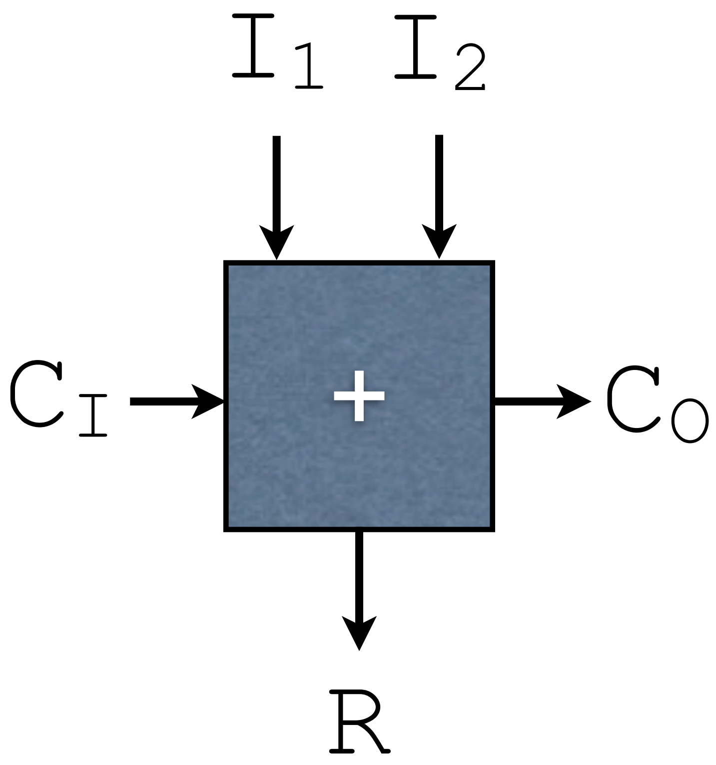
Input Carry Bit

Single Bit Adder

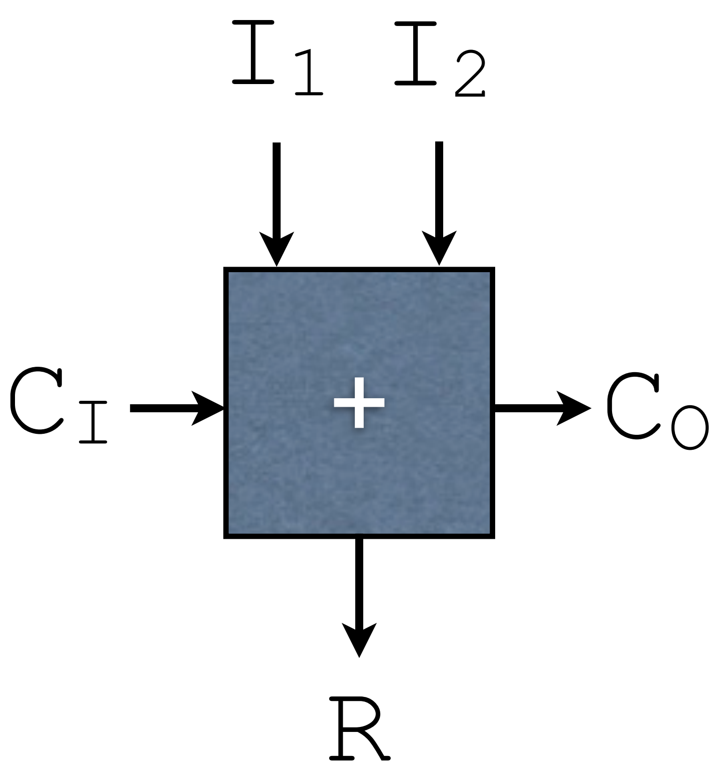
Result Bit

Output Carry Bit

Putting it Together

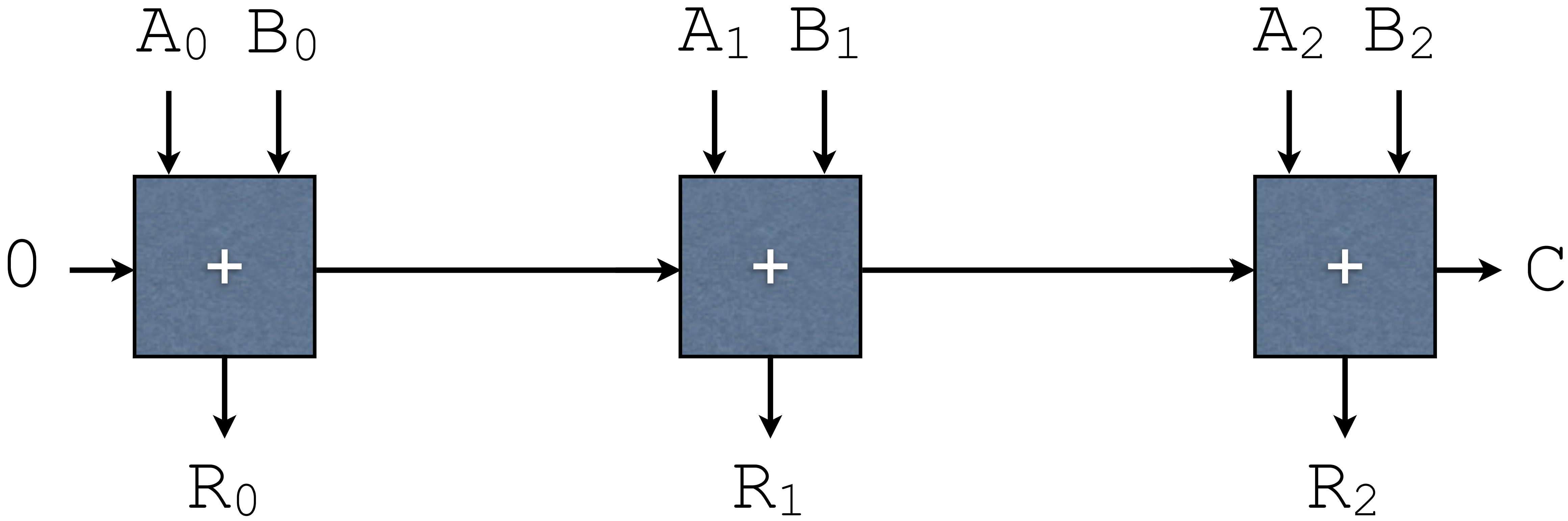


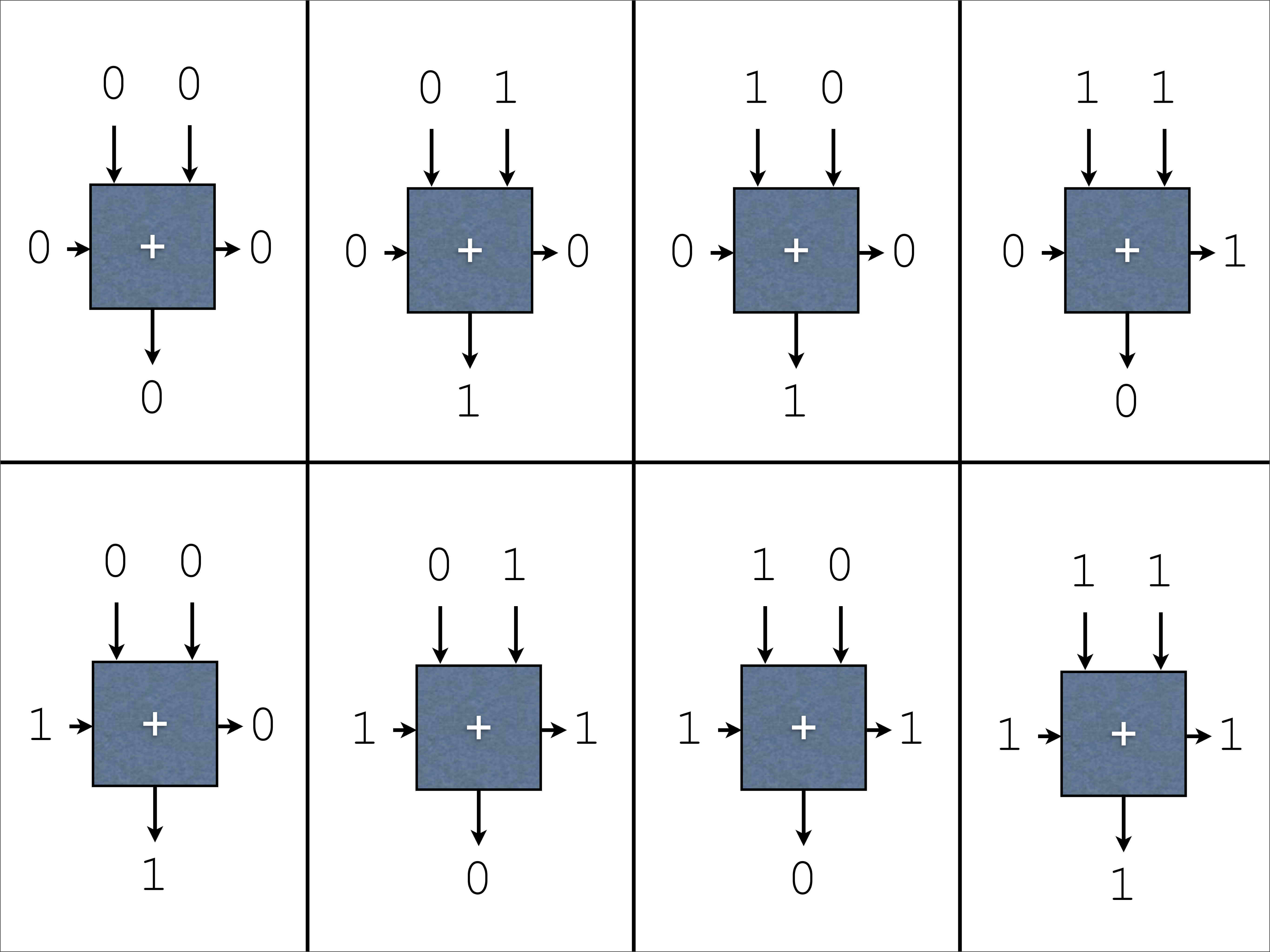
Recall: Single Bit Adders



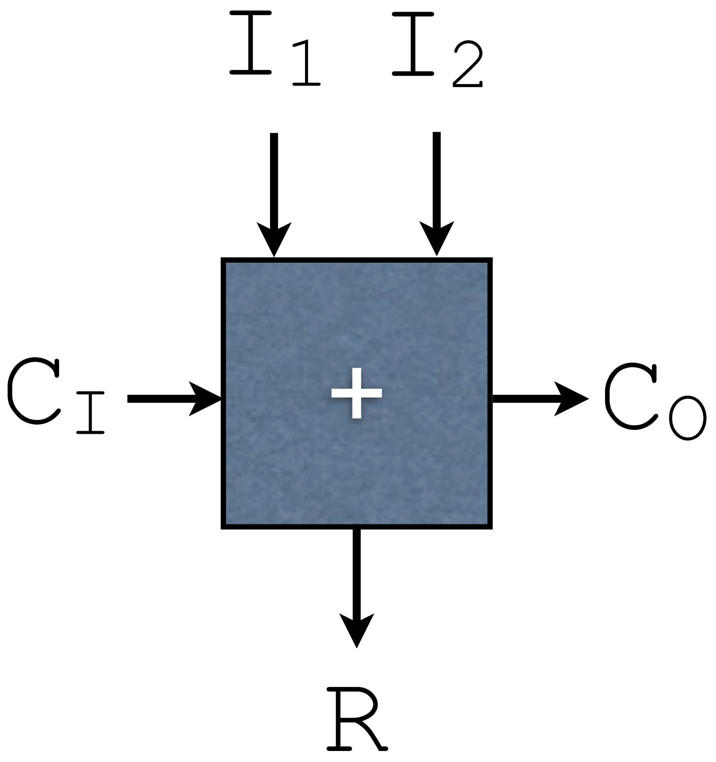
Stringing them Together

For two three-bit numbers, `A` and `B`, resulting in a three-bit result R




0 0
0 + 0
0
0 1
0 + 0
1
1 0
0 + 0
1
1 1
0 + 1
0
0 0
1 + 0
1
0 1
1 + 1
0
1 0
1 + 1
0
1 1
1 + 1
1

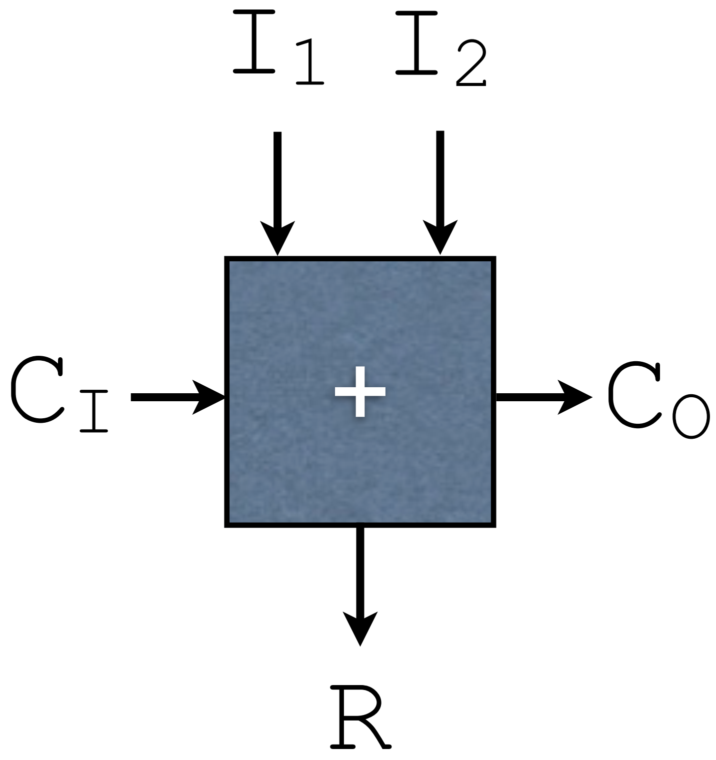
As a Truth Table



C_I	I_1	I_2	C_O	R
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

As a Truth Table

Question: how can this be turned into a circuit?



C_I	I_1	I_2	C_O	R
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Sum of Products

- Variables: `A`, `B`, `C`...
- Negation of a variable: $\overline{\texttt{A}}$, $\overline{\texttt{B}}$, $\overline{\texttt{C}}$...

Sum of Products

- Another way to look at OR: sum (+)

`A + B`

- Another way to look at AND: multiplication (*)

`A * B` `AB`

Sum of Products Example

A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

Sum of Products Example

A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

Sum of Products Example

A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

$O = \boxed{\overline{A} * B}$

Sum of Products Example

A	B	O
0	0	0
0	1	1
1	0	1
1	1	0

$$O = \overline{A}*B + \boxed{A*\overline{B}}$$

Sum of Products

C_I	I_1	I_2	C_O	R
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Question: What would the sum of products look like for this table?

(Note: need one equation for each output.)

Sum of Products

C_I	I_1	I_2	C_O	R
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Question: What would the sum of products look like for this table?

(Note: need one equation for each output.)

Answer in the presenter notes.

In-Class Example: Shift Left by 1

Circuit Minimization

Motivation

- Unnecessarily large programs: bad
- Unnecessarily large circuits: Very Bad™
    - Why?

Motivation

- Unnecessarily large programs: bad
- Unnecessarily large circuits: Very Bad™
    - Why?
        - Bigger circuits = bigger chips = higher cost (non-linear too!)
        - Longer circuits = more time needed to move electrons through = slower

Simplification

- Real-world formulas can often be simplified, according to algebraic rules
  - How might we simplify the following?

```
R = A*B + !A*B
```

Simplification

- Real-world formulas can often be simplified, according to algebraic rules
  - How might we simplify the following?

```
R = A*B + !A*B
R = B(A + !A)

R = B(true)

R = B
```

Simplification Trick

- Look for products that differ only in one variable
  - One product has the original variable (`A`)
  - The other product has the other variable (`!A`)

`R = A*B + !A*B`

Additional Example 1

`!ABCD + ABCD + !AB!CD + AB!CD`

Additional Example 1

`!ABCD + ABCD + !AB!CD + AB!CD`

`BCD(A + !A) + !AB!CD + AB!CD`

Additional Example 1

```
!ABCD + ABCD + !AB!CD + AB!CD

BCD(A + !A) + !AB!CD + AB!CD

BCD + !AB!CD + AB!CD
```

Additional Example 1

`!ABCD + ABCD + !AB!CD + AB!CD`

`BCD(A + !A) + !AB!CD + AB!CD`

`BCD + !AB!CD + AB!CD`

`BCD + B!CD(!A + A)`

Additional Example 1

!ABCD + ABCD + !AB!CD + AB!CD

BCD(A + !A) + !AB!CD + AB!CD

BCD + !AB!CD + AB!CD

BCD + B!CD(!A + A)

BCD + B!CD

Additional Example 1

`!ABCD + ABCD + !AB!CD + AB!CD`

`BCD(A + !A) + !AB!CD + AB!CD`

`BCD + !AB!CD + AB!CD`

`BCD + B!CD(!A + A)`

`BCD + B!CD`

`BD(C + !C)`

Additional Example 1

```
!ABCD + ABCD + !AB!CD + AB!CD

BCD(A + !A) + !AB!CD + AB!CD

BCD + !AB!CD + AB!CD

BCD + B!CD(!A + A)

BCD + B!CD

BD(C + !C)
BD
```

Additional Example 2

`!A!BC + A!B!C + !ABC + !AB!C + A!BC`

Additional Example 2

`!A!BC + A!B!C + !ABC + !AB!C + A!BC`

`!A!BC + A!BC + A!B!C + !ABC + !AB!C`

Additional Example 2

`!A!BC + A!B!C + !ABC + !AB!C + A!BC`

`!A!BC + A!BC + A!B!C + !ABC + !AB!C`

`!BC(A + !A) + A!B!C + !ABC + !AB!C`

Additional Example 2

```
!A!BC + A!B!C + !ABC + !AB!C + A!BC

!A!BC + A!BC + A!B!C + !ABC + !AB!C

!BC(A + !A) + A!B!C + !ABC + !AB!C

!BC + A!B!C + !ABC + !AB!C
```

Additional Example 2

```
!A!BC + A!B!C + !ABC + !AB!C + A!BC

!A!BC + A!BC + A!B!C + !ABC + !AB!C

!BC(A + !A) + A!B!C + !ABC + !AB!C

!BC + A!B!C + !ABC + !AB!C

!BC + A!B!C + !AB(C + !C)
```

Additional Example 2

!A!BC + A!B!C + !ABC + !AB!C + A!BC

!A!BC + A!BC + A!B!C + !ABC + !AB!C

!BC(A + !A) + A!B!C + !ABC + !AB!C

!BC + A!B!C + !ABC + !AB!C

!BC + A!B!C + !AB(C + !C)

!BC + A!B!C + !AB

Scaling Up

- Performing this sort of algebraic manipulation by hand can be tricky
- We can use *Karnaugh maps* to make it immediately apparent as to what can be simplified

Example

```
R = A*B + !A*B
```

Example

`R = A*B + !A*B`

A	B	O
0	0	0
0	1	1
1	0	0
1	1	1

Example

`R = A*B + !A*B`

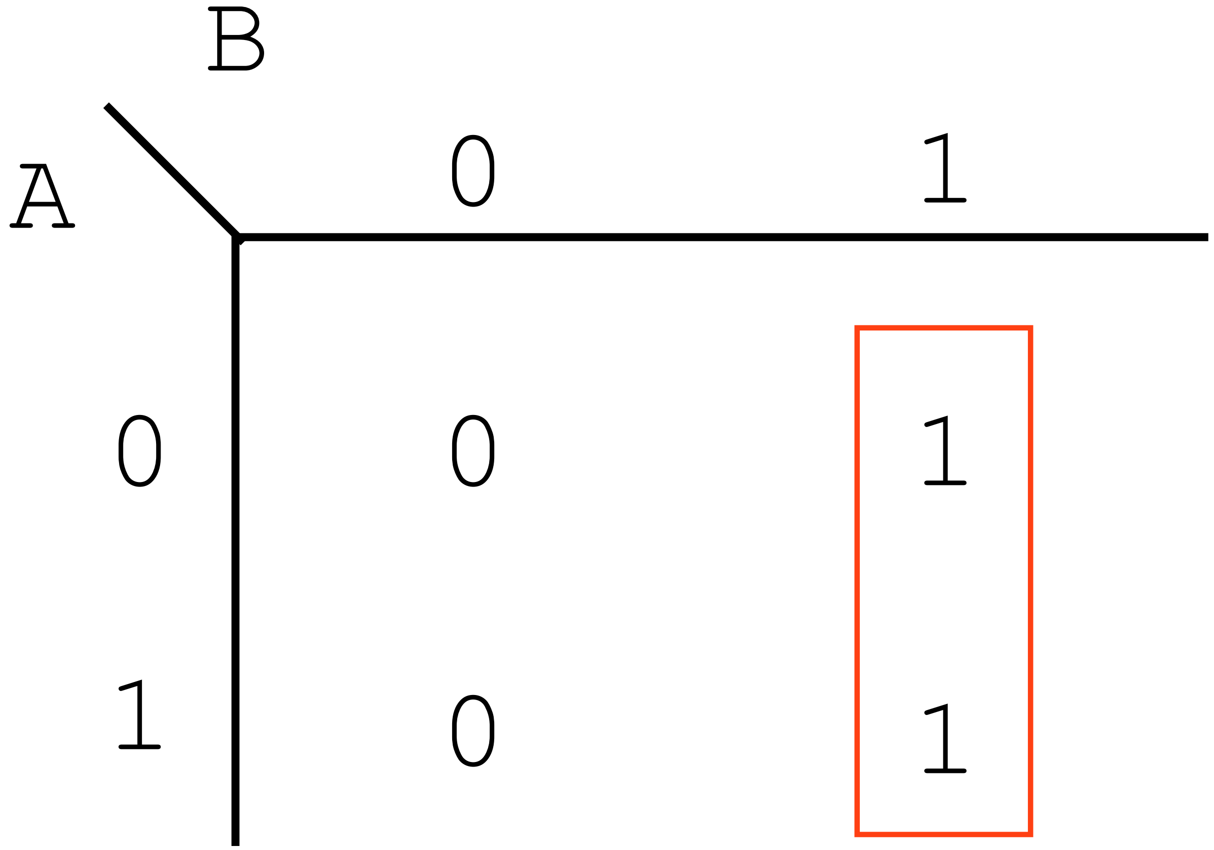
A	B	O
0	0	0
0	1	1
1	0	0
1	1	1

$$\begin{array}{c|cc} {}_{A}\diagdown^{B} & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 0 & 1 \end{array}$$

Example

`R = A*B + !A*B`

A	B	O
0	0	0
0	1	1
1	0	0
1	1	1

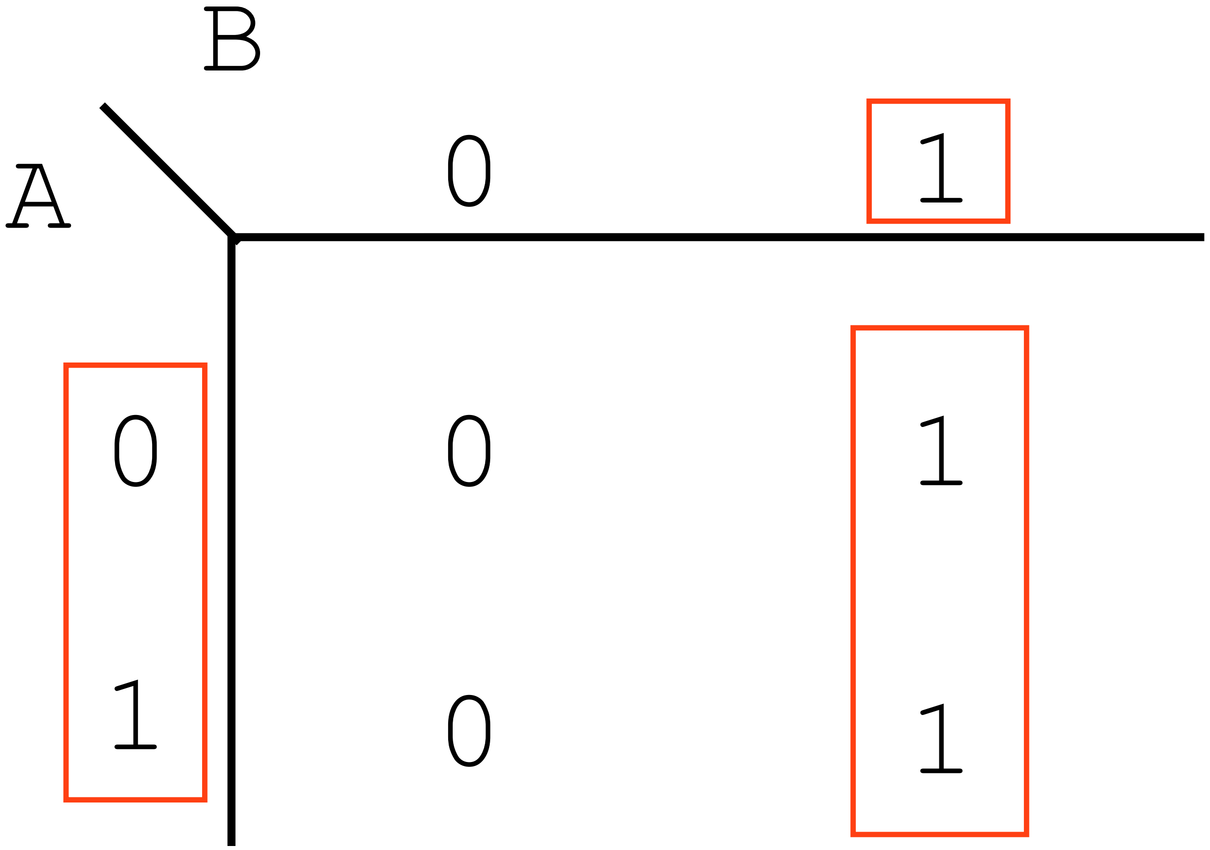


A B	0	1
0	0	1
1	0	1

Example

`R = A*B + !A*B`

A	B	O
0	0	0
0	1	1
1	0	0
1	1	1



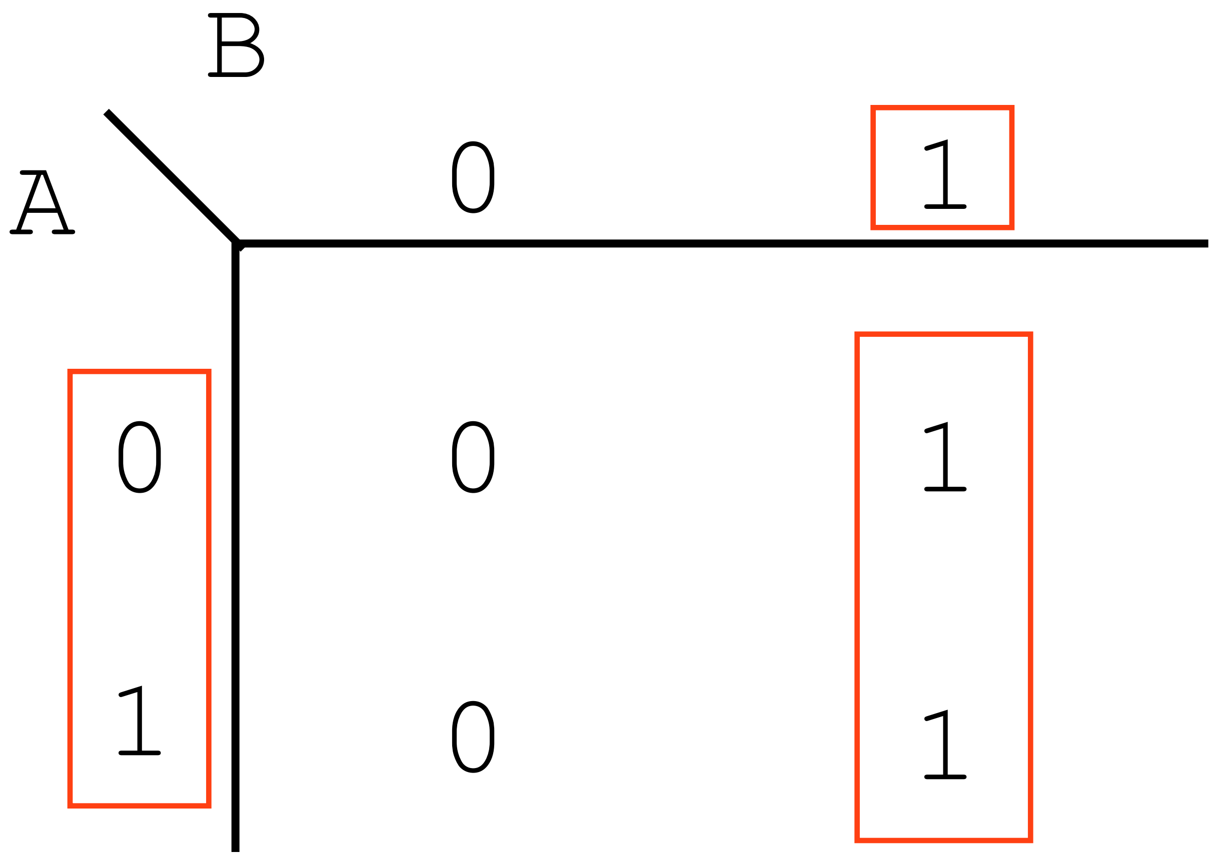
A \ B	0	1
0	0	1
1	0	1

Example

`R = A*B + !A*B`

A	B	O
0	0	0
0	1	1
1	0	0
1	1	1

`R = B`



A \ B	0	1
0	0	1
1	0	1

Three Variables

- We can scale this up to three variables, by combining two variables on one axis
- The combined axis must be arranged such that only one bit changes per position

A \ BC	00	01	11	10
0	?	?	?	?
1	?	?	?	?

Three Variable Example

`R = !A!BC + !ABC + A!BC + ABC`

R = !A!BC + !ABC + A!BC + ABC

A	B	C	R
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

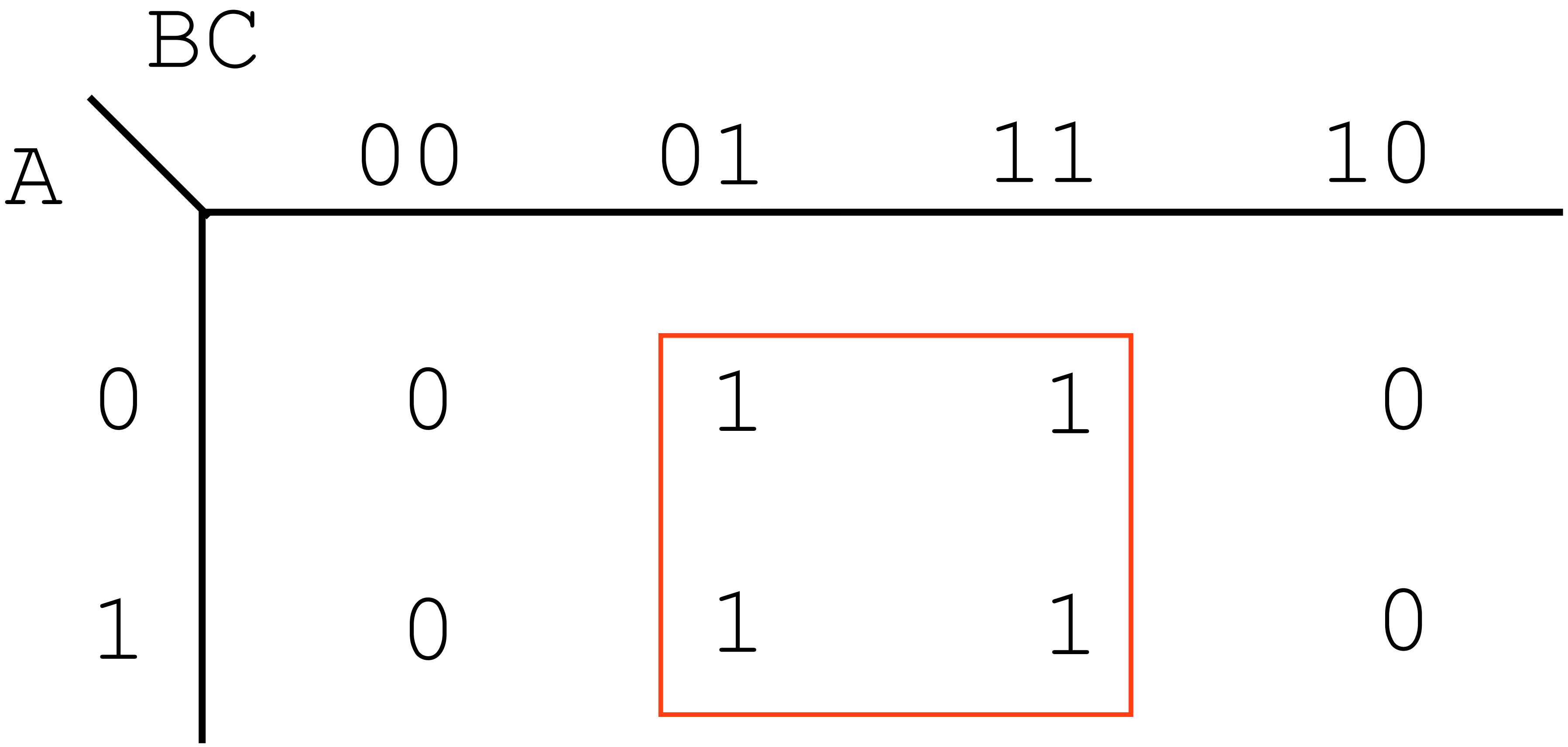
R = !A!BC + !ABC + A!BC + ABC

A	B	C	R
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

BC A	00	01	11	10
0	0	1	1	0
1	0	1	1	0

R = !A!BC + !ABC + A!BC + ABC

A	B	C	R
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

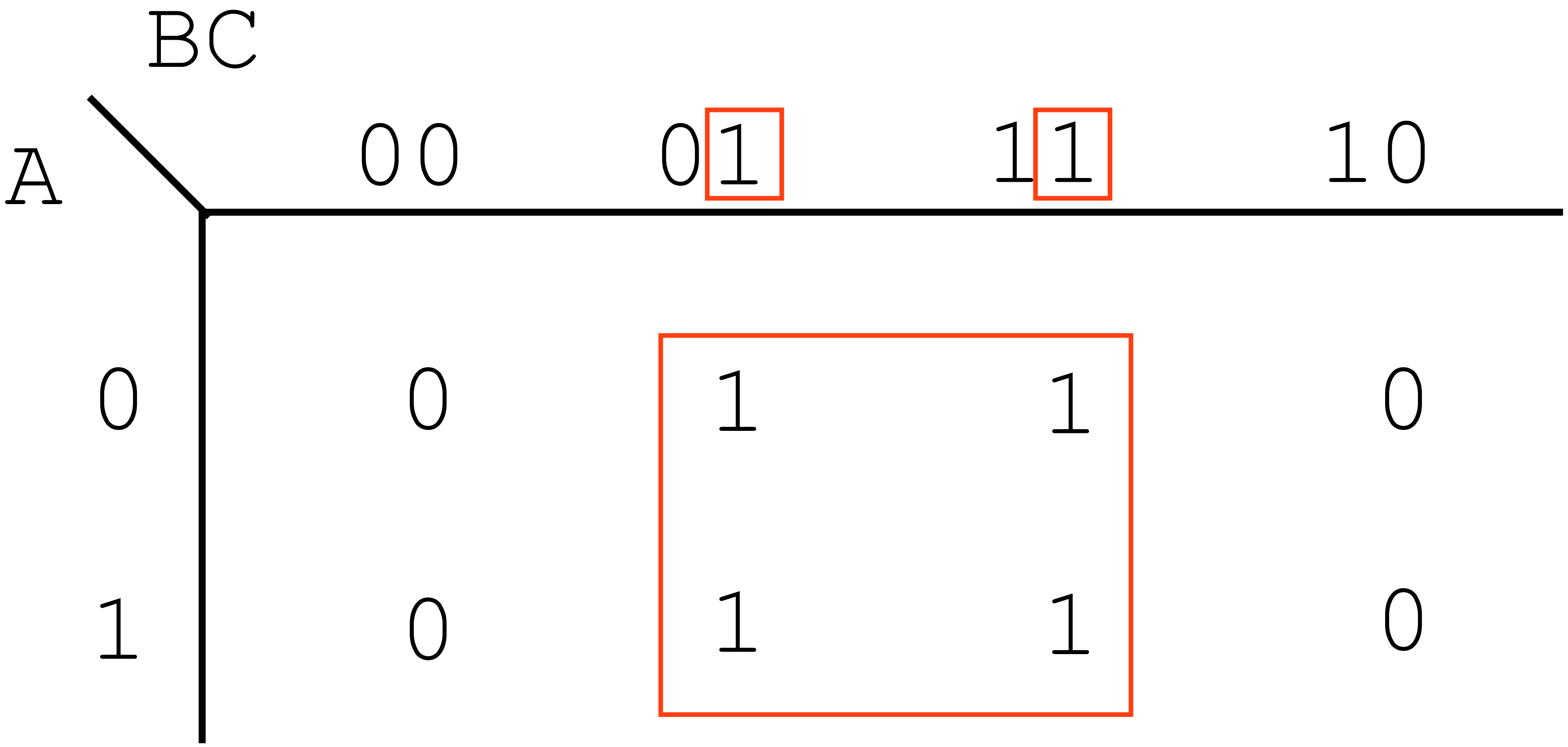


A \ BC	00	01	11	10
0	0	1	1	0
1	0	1	1	0

R = !A!BC + !ABC + A!BC + ABC

A	B	C	R
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

R = C



A \ BC	00	01	11	10
0	0	1	1	0
1	0	1	1	0

Another Three Variable Example

R = !A!B!C + !A!BC + !ABC + !AB!C + A!B!C + AB!C

R = !A!B!C + !A!BC + !ABC +
!AB!C + A!B!C + AB!C

A	B	C	R
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

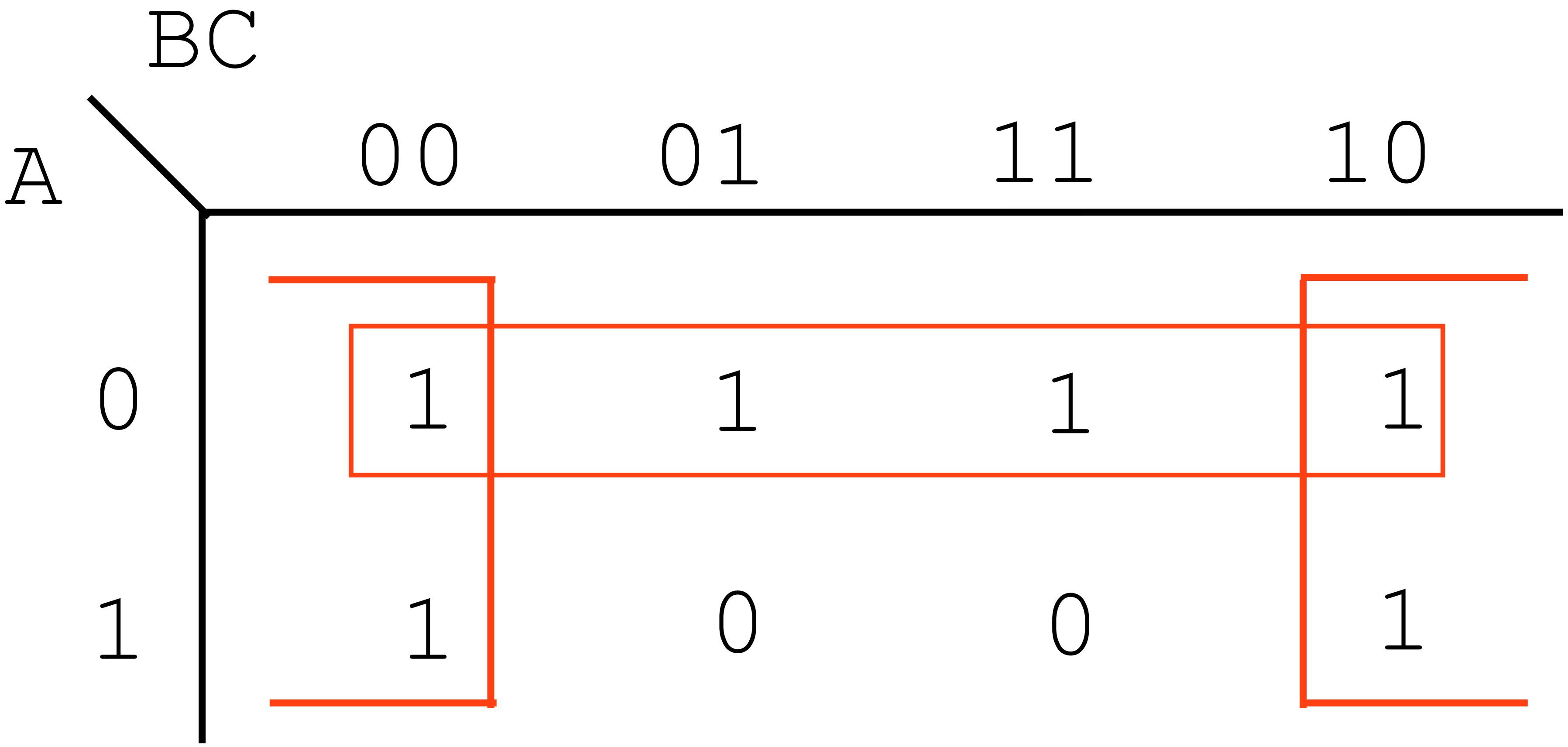
R = !A!B!C + !A!BC + !ABC +
!AB!C + A!B!C + AB!C

A	B	C	R
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

A \ BC	00	01	11	10
0	1	1	1	1
1	1	0	0	1

R = !A!B!C + !A!BC + !ABC +
!AB!C + A!B!C + AB!C

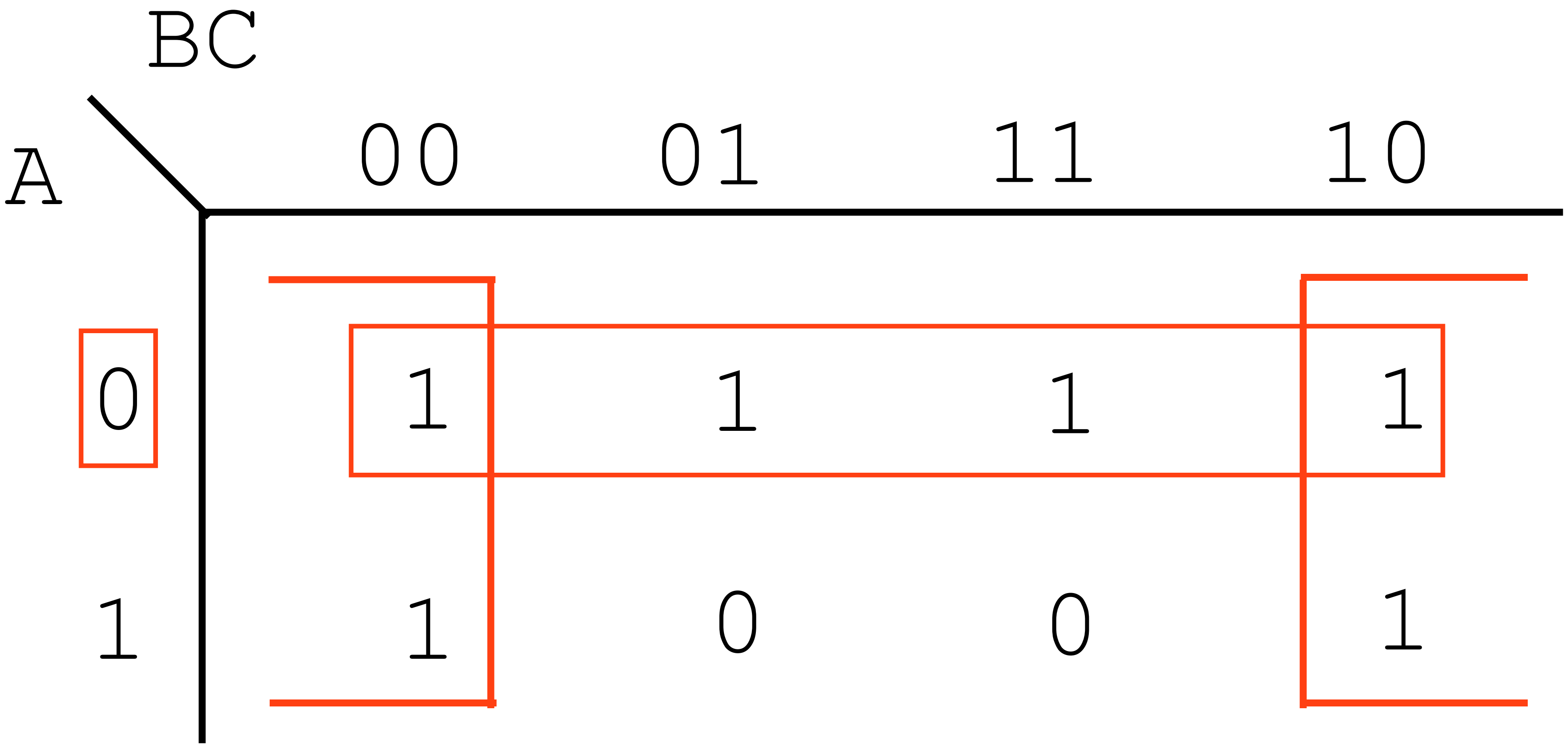
A	B	C	R
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0



A \ BC	00	01	11	10
0	1	1	1	1
1	1	0	0	1

R = !A!B!C + !A!BC + !ABC + !AB!C + A!B!C + AB!C

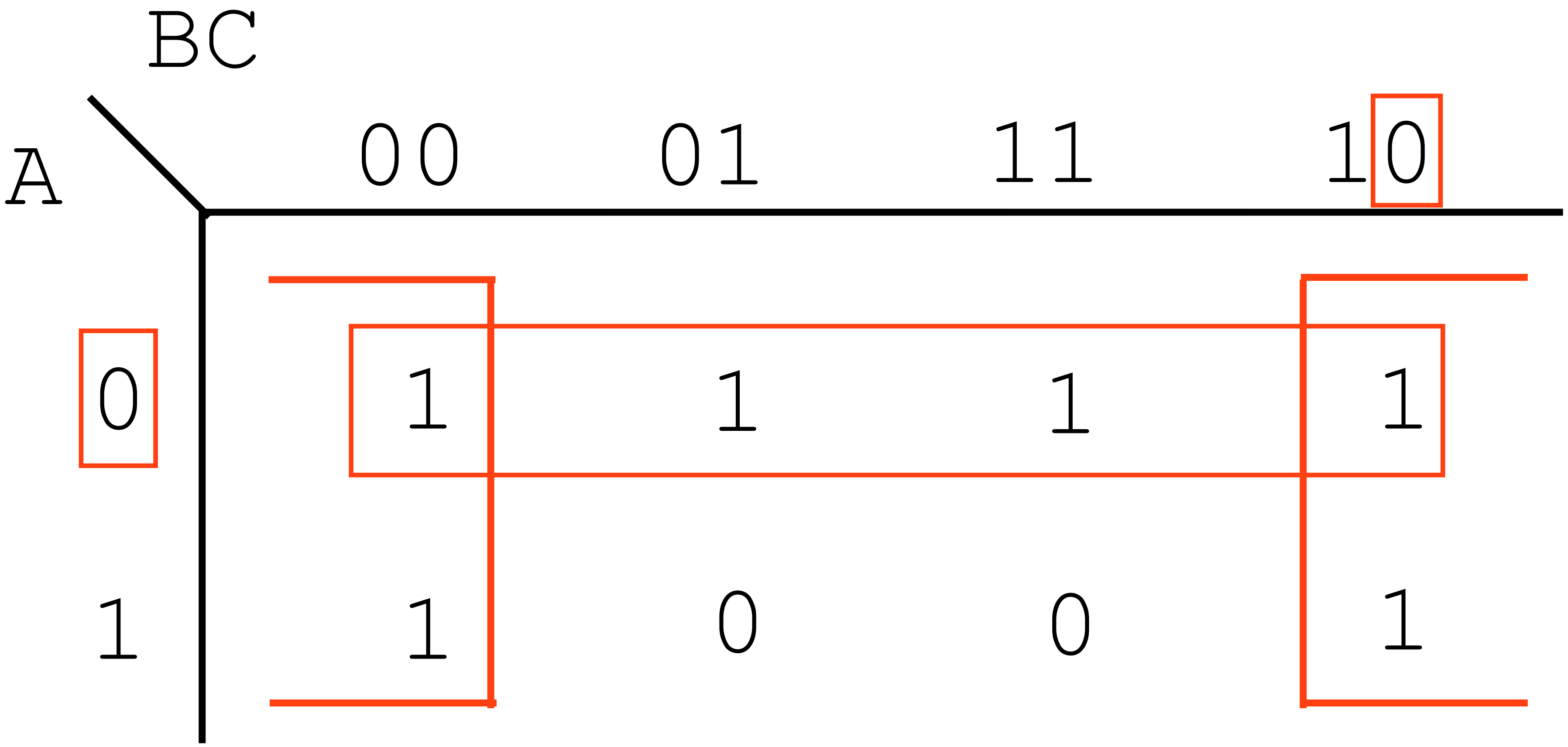
A	B	C	R
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0



A \ BC	00	01	11	10
0	1	1	1	1
1	1	0	0	1

R = !A!B!C + !A!BC + !ABC +
!AB!C + A!B!C + AB!C

A	B	C	R
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0



BC

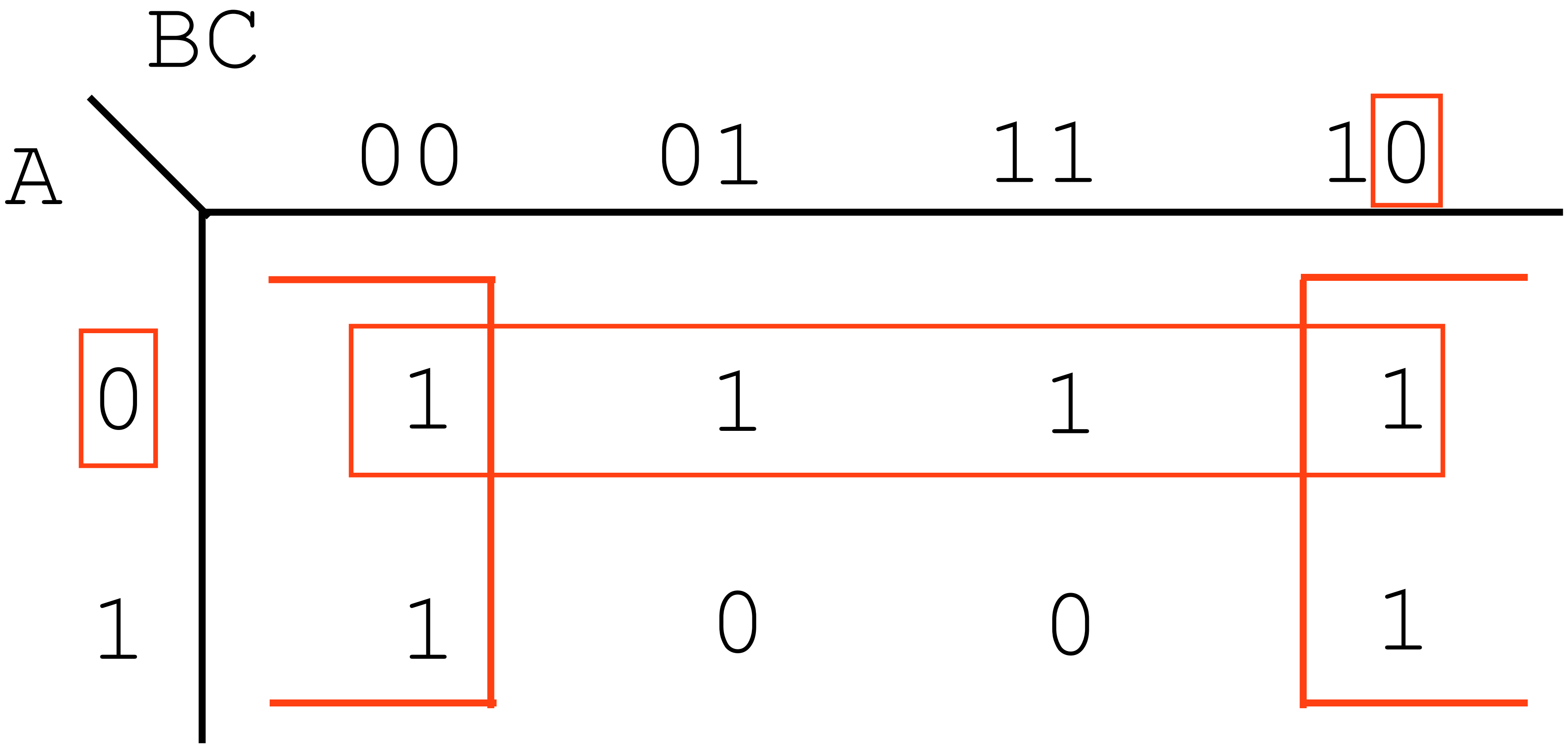
A

	00	01	11	10
0	1	1	1	1
1	1	0	0	1

R = !A!B!C + !A!BC + !ABC +
!AB!C + A!B!C + AB!C

A	B	C	R
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

R = !A + !C



A \ BC	00	01	11	10
0	1	1	1	1
1	1	0	0	1

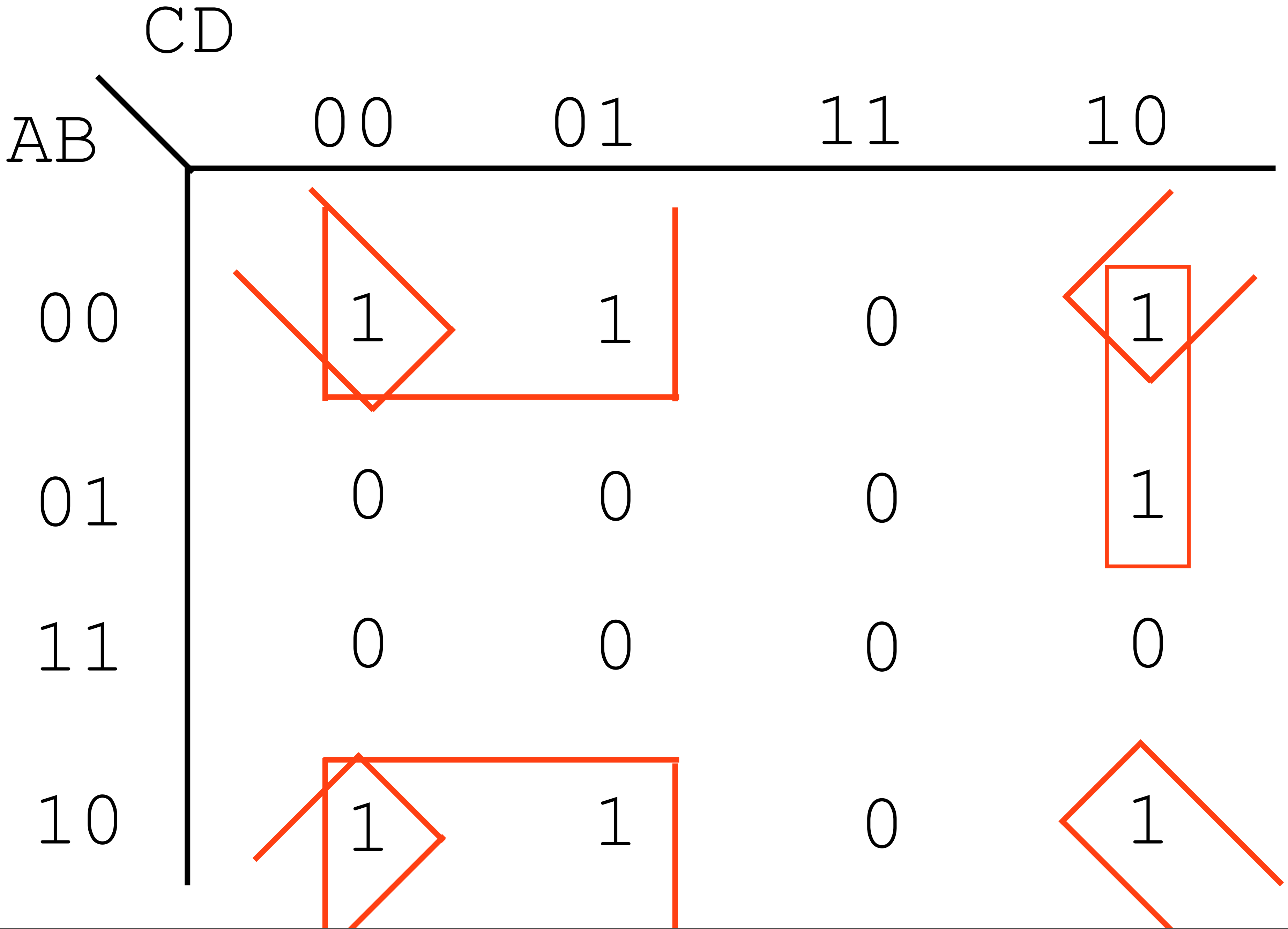
Four Variable Example

R = !A!B!C!D + !A!B!CD + !A!BC!D + !ABC!D + A!B!C!D + A!B!CD + A!BC!D

R = !A!B!C!D + !A!B!CD + !A!BC!D + !ABC!D + A!B!C!D + A!B!CD + A!BC!D

AB \ CD	00	01	11	10
00	1	1	0	1
01	0	0	0	1
11	0	0	0	0
10	1	1	0	1

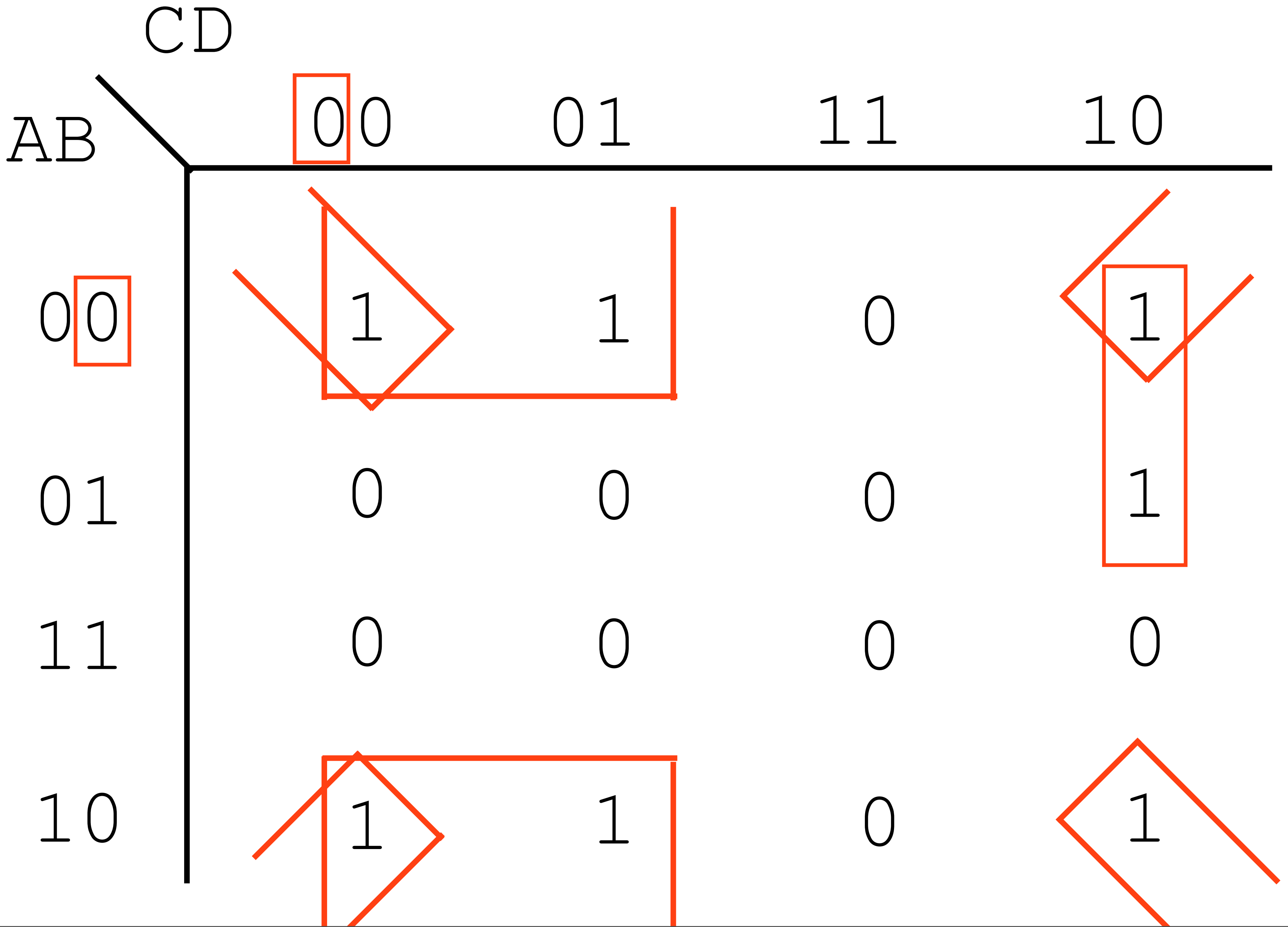
R = !A!B!C!D + !A!B!CD + !A!BC!D + !ABC!D + A!B!C!D + A!B!CD + A!BC!D



AB \ CD	00	01	11	10
00	1	1	0	1
01	0	0	0	1
11	0	0	0	0
10	1	1	0	1

R = !A!B!C!D + !A!B!CD + !A!BC!D + !ABC!D + A!B!C!D + A!B!CD + A!BC!D

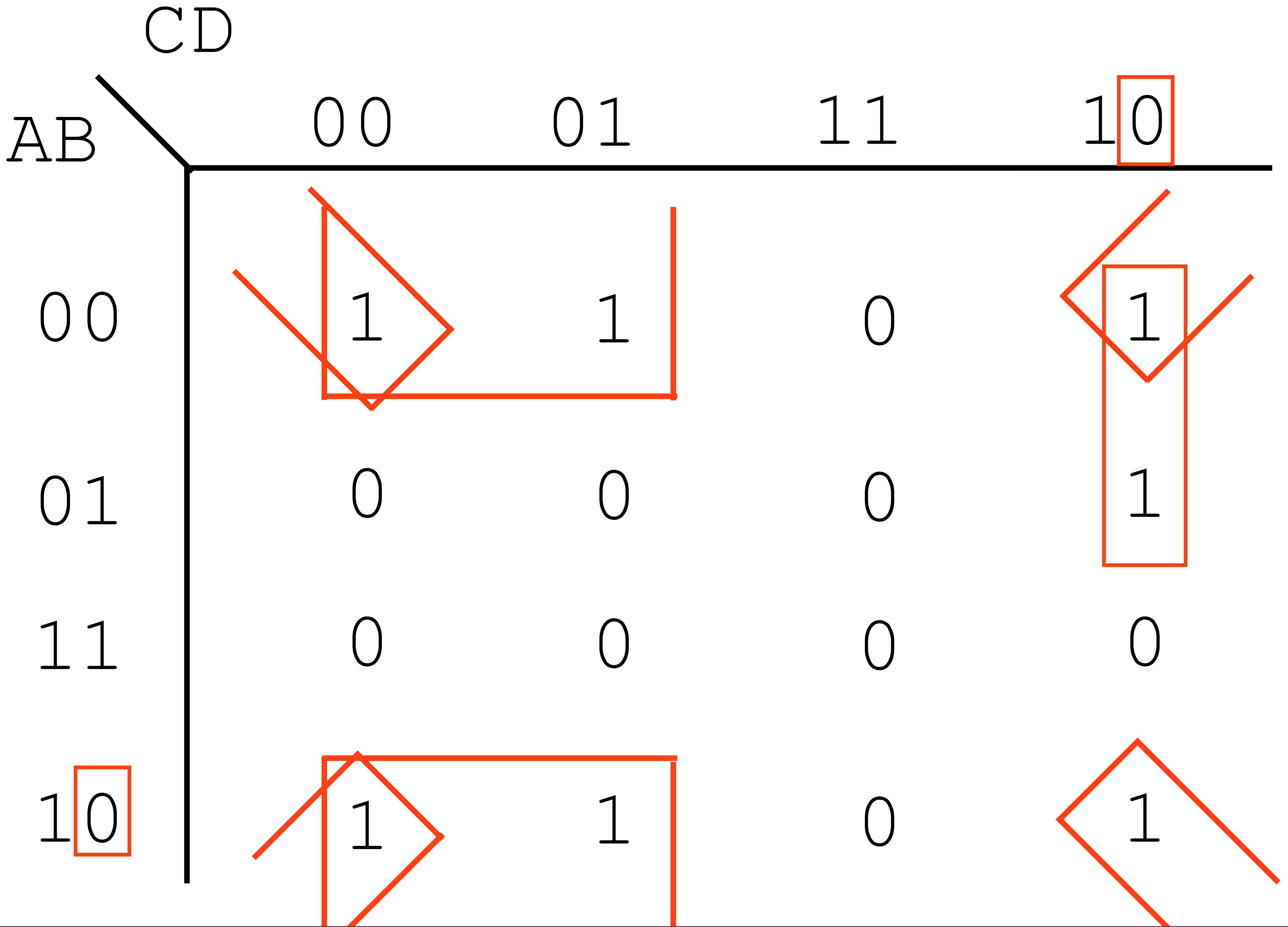
R =!B!C



AB \ CD	00	01	11	10
00	1	1	0	1
01	0	0	0	1
11	0	0	0	0
10	1	1	0	1

R = !A!B!C!D + !A!B!CD + !A!BC!D + !ABC!D + A!B!C!D + A!B!CD + A!BC!D

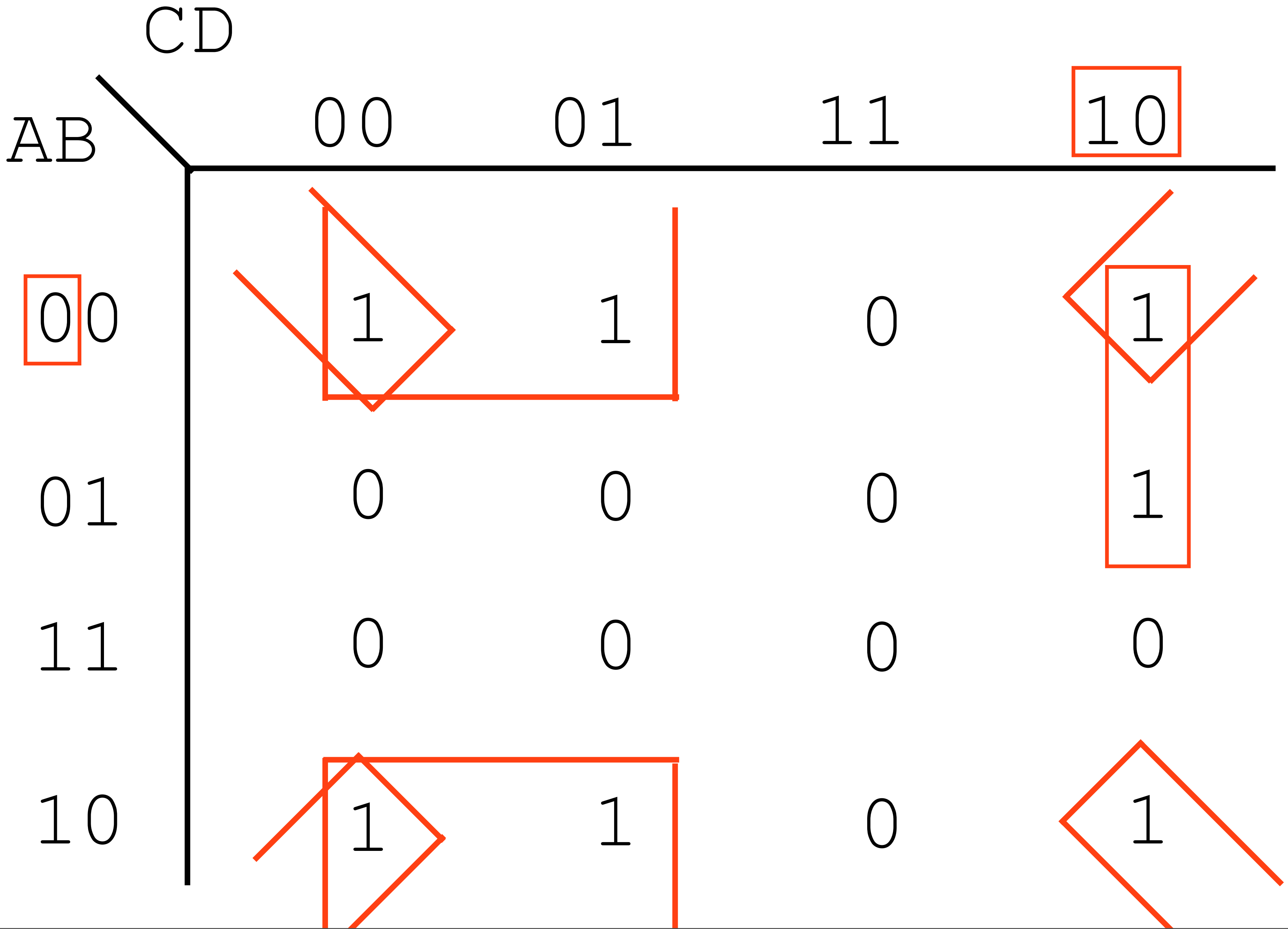
R =!B!C + !B!D



AB \ CD	00	01	11	10
00	1	1	0	1
01	0	0	0	1
11	0	0	0	0
10	1	1	0	1

R = !A!B!C!D + !A!B!CD + !A!BC!D + !ABC!D + A!B!C!D + A!B!CD + A!BC!D

R =!B!C + !B!D + !AC!D



AB \ CD	00	01	11	10
00	1	1	0	1
01	0	0	0	1
11	0	0	0	0
10	1	1	0	1

K-Map Rules in Summary (1)

- Groups can contain only `1`s
- Only `1`s in adjacent groups are allowed (no diagonals)
- The number of `1`s in a group must be a power of two (1, 2, 4, 8...)
- The groups must be as large as legally possible

K-Map Rules in Summary (2)

- All `1`s must belong to a group, even if it's a group of one element
- Overlapping groups are permitted
- Wrapping around the map is permitted
- Use the fewest number of groups possible

Revisiting Problem

`!A!BC + A!B!C + !ABC + !AB!C + A!BC`

Revisiting Problem

`R = !A!BC + A!B!C + !ABC + !AB!C + A!BC`

A	B	C	R
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Revisiting Problem

`R = !A!BC + A!B!C + !ABC + !AB!C + A!BC`

A	B	C	R
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

A \ BC	00	01	11	10
0	0	1	1	1
1	1	1	0	0

Revisiting Problem

R = !A!BC + A!B!C + !ABC + !AB!C + A!BC

A	B	C	R
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

R = !AC

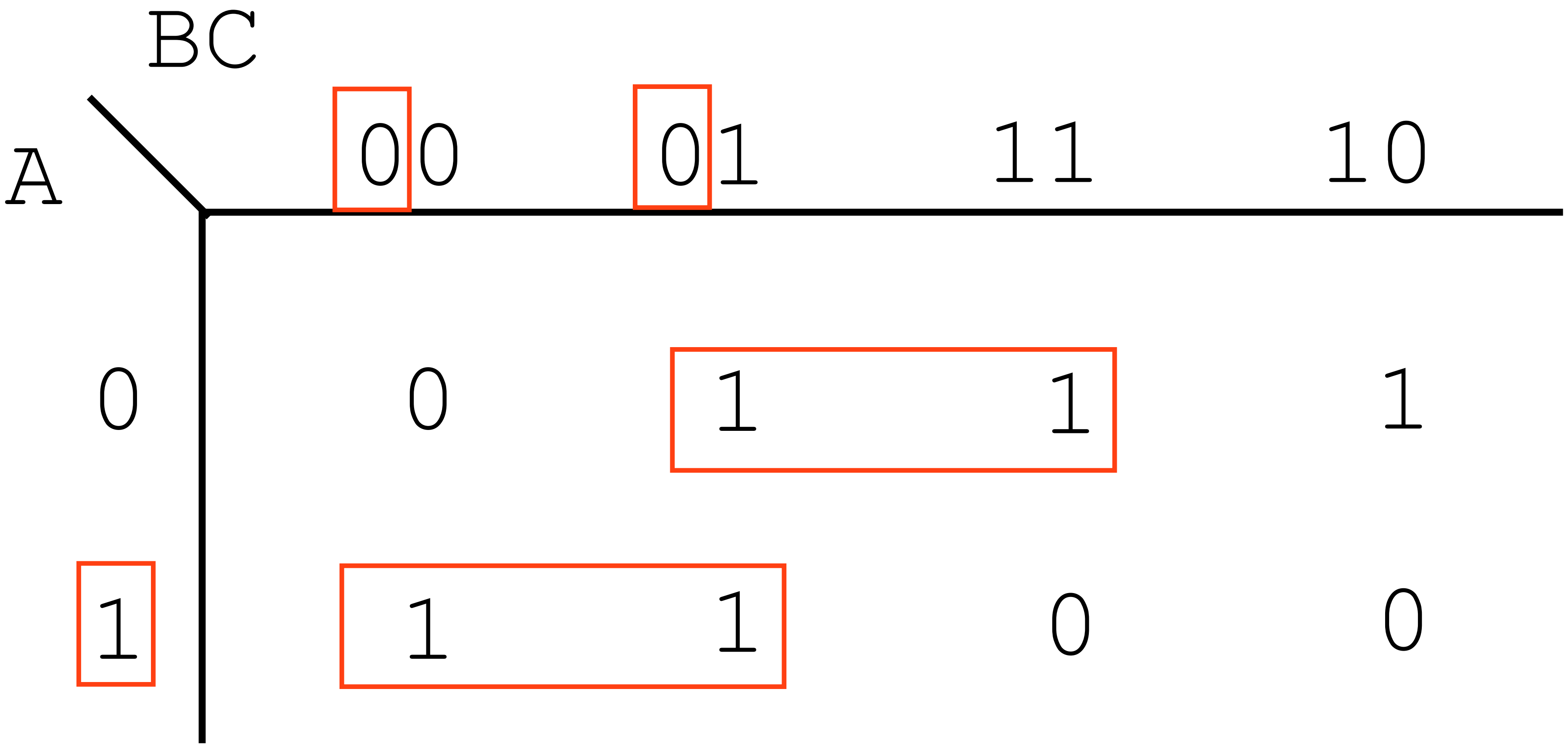
A \ BC	00	01	11	10
0	0	1	1	1
1	1	1	0	0

Revisiting Problem

R = !A!BC + A!B!C + !ABC + !AB!C + A!BC

A	B	C	R
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

R = !AC + A!B



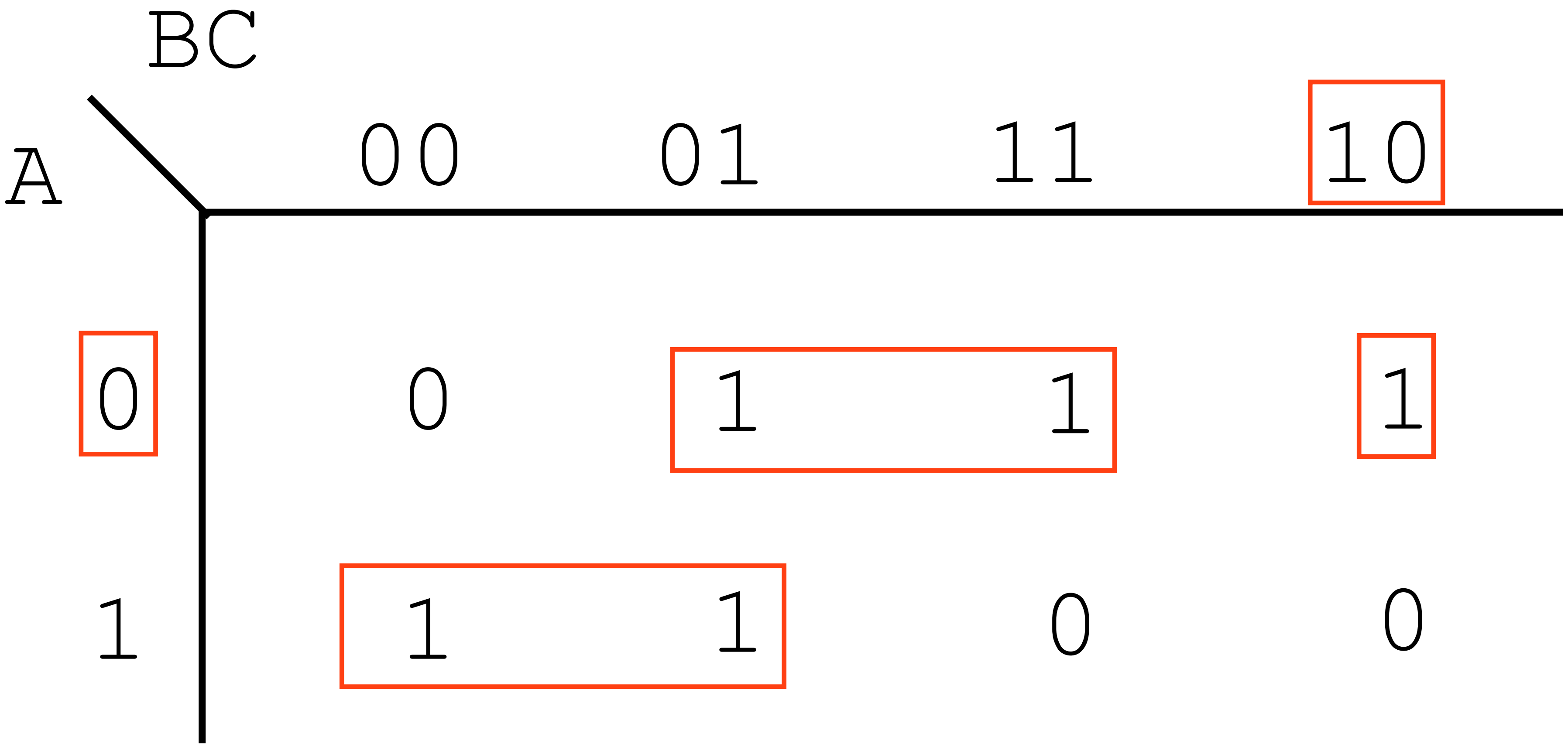
A \ BC	00	01	11	10
0	0	1	1	1
1	1	1	0	0

Revisiting Problem

R = !A!BC + A!B!C + !ABC + !AB!C + A!BC

A	B	C	R
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

R = !AC + A!B + !AB!C



A \ BC	00	01	11	10
0	0	1	1	1
1	1	1	0	0

Difference

- Algebraic solution: `!BC + A!B!C + !AB`
- K-map solution: `!AC + A!B + !AB!C`
- Question: why might these differ?

Difference

- Algebraic solution: `!BC + A!B!C + !AB`
- K-map solution: `!AC + A!B + !AB!C`
- Question: why might these differ?
  - Both are *minimal*, in that they have the fewest number of products possible
  - Can be multiple minimal solutions

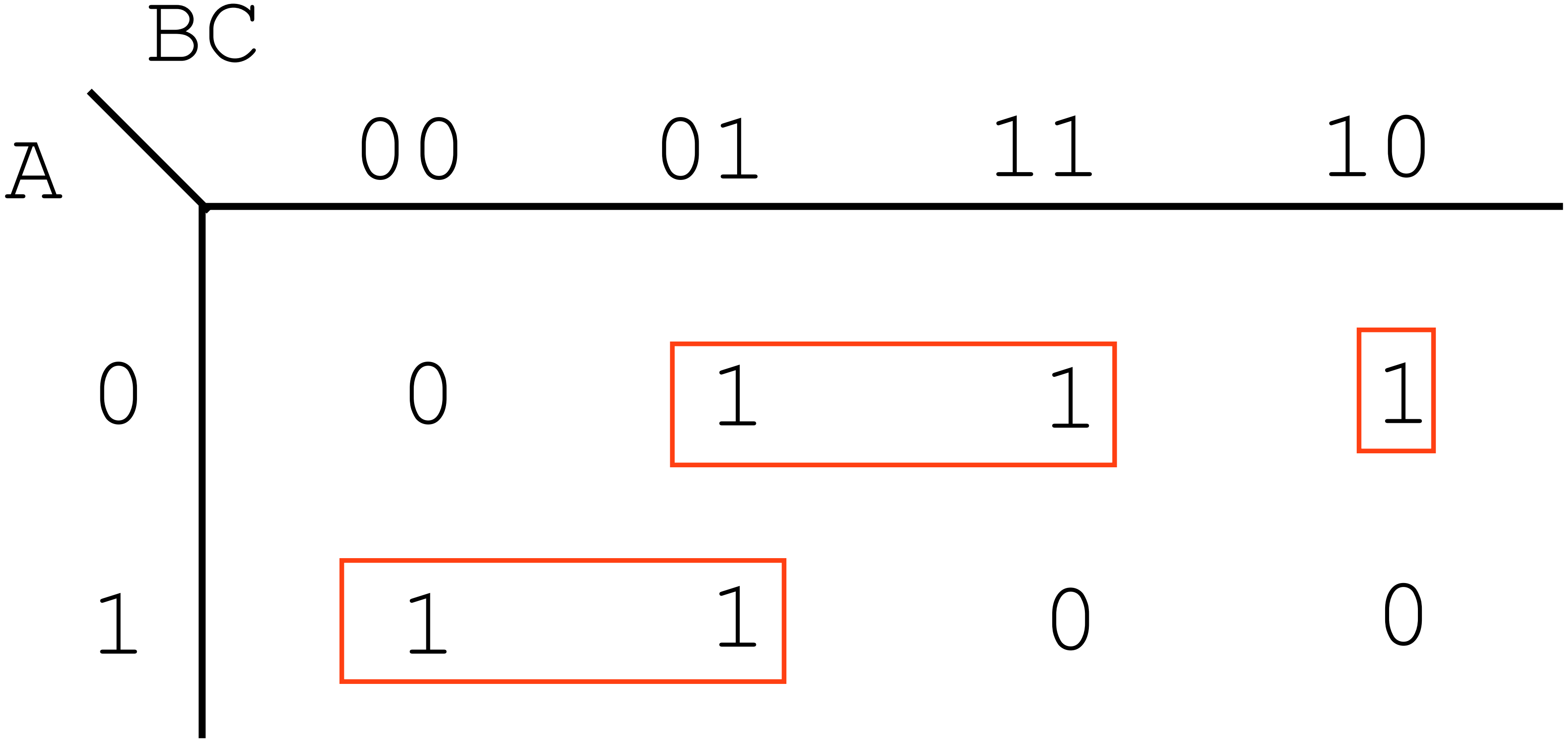
Difference

- Algebraic solution: `!BC + A!B!C + !AB`
- K-map solution: `!AC + A!B + !AB!C`
- Question: why might these differ?
  - Both are *minimal*, in that they have the fewest number of products possible
  - Can be multiple minimal solutions

Difference

Algebraic solution: `!BC + A!B!C + !AB`

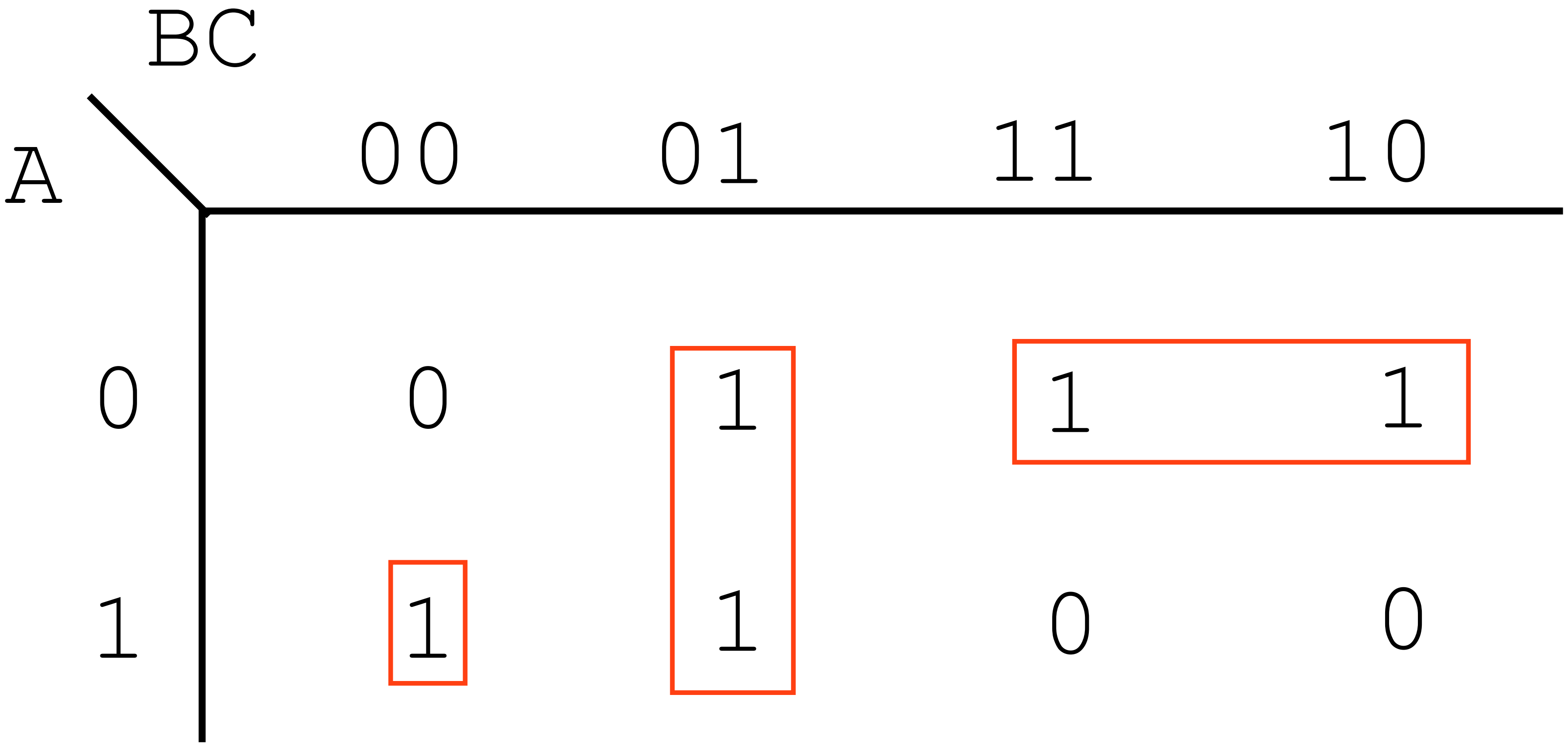
K-map solution: `!AC + A!B + !AB!C`



A \ BC	00	01	11	10
0	0	1	1	1
1	1	1	0	0

Difference

Algebraic solution: `!BC + A!B!C + !AB`
K-map solution: `!BC + A!B!C + !AB`



A \ BC	00	01	11	10
0	0	1	1	1
1	1	1	0	0

Exploiting *Don't Cares* in K-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 `1`s
- 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

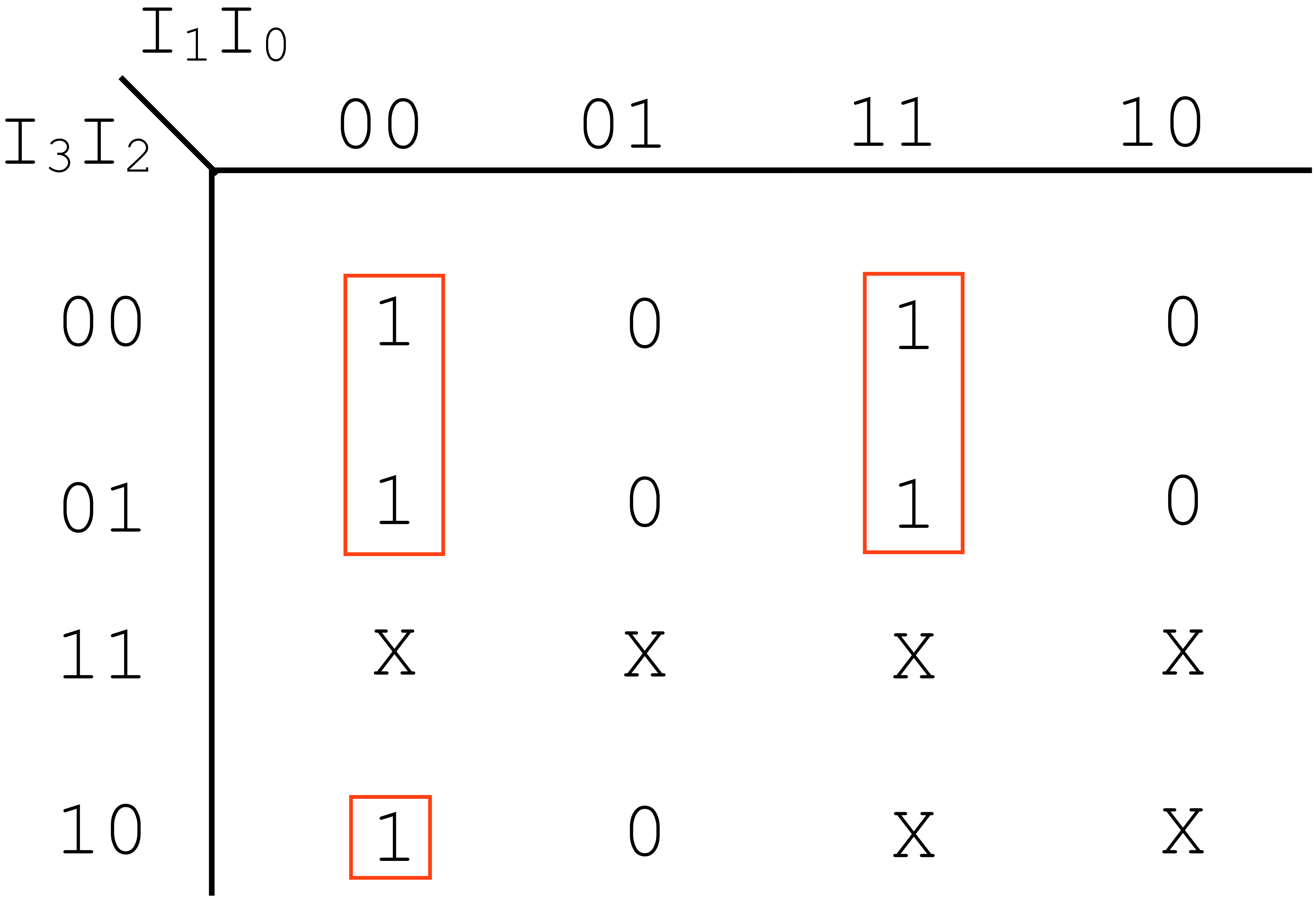
Example

As a K-map

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

Example

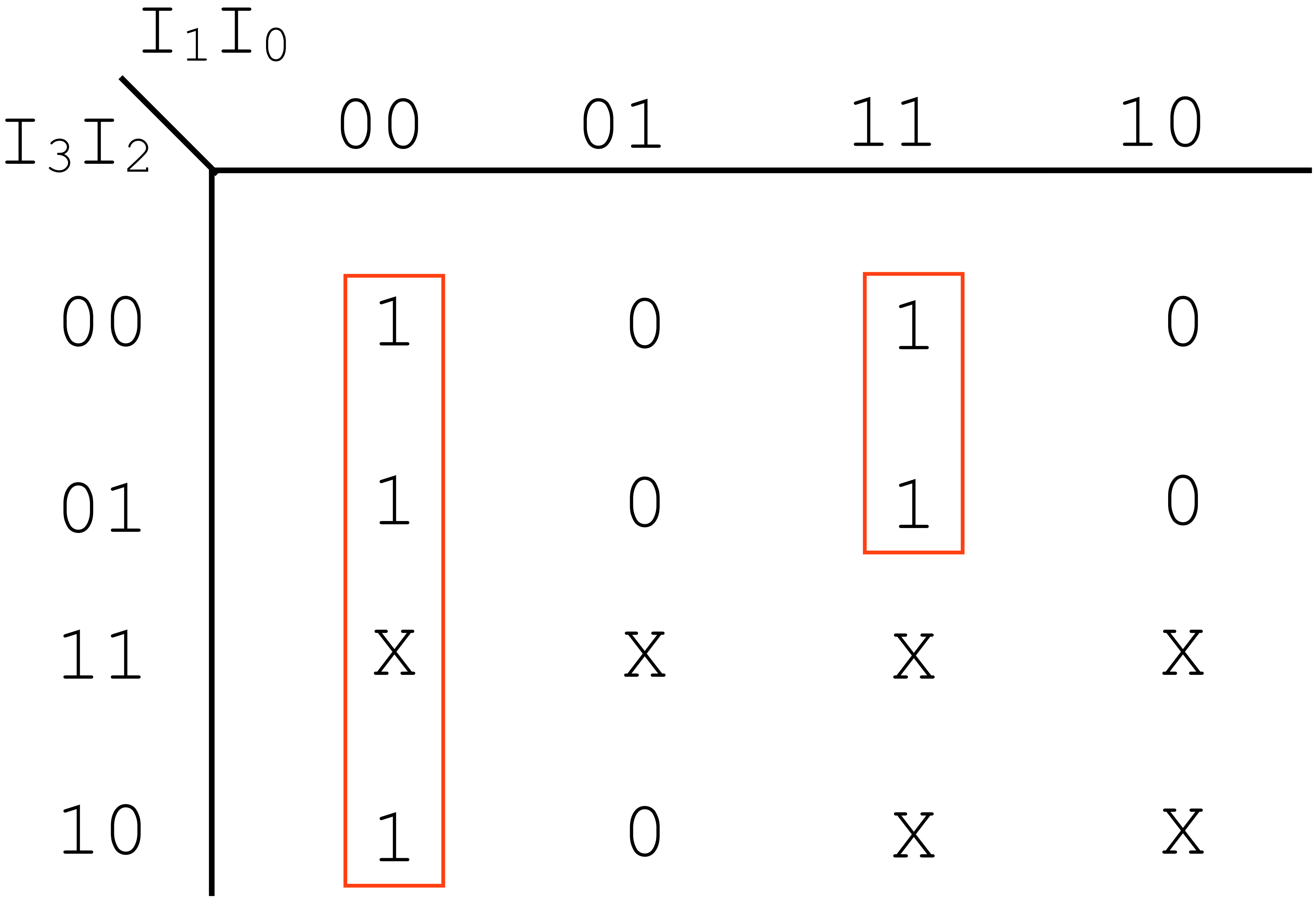
If we don't exploit *don't cares*...



I_3I_2 \ I_1I_0	00	01	11	10
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...*

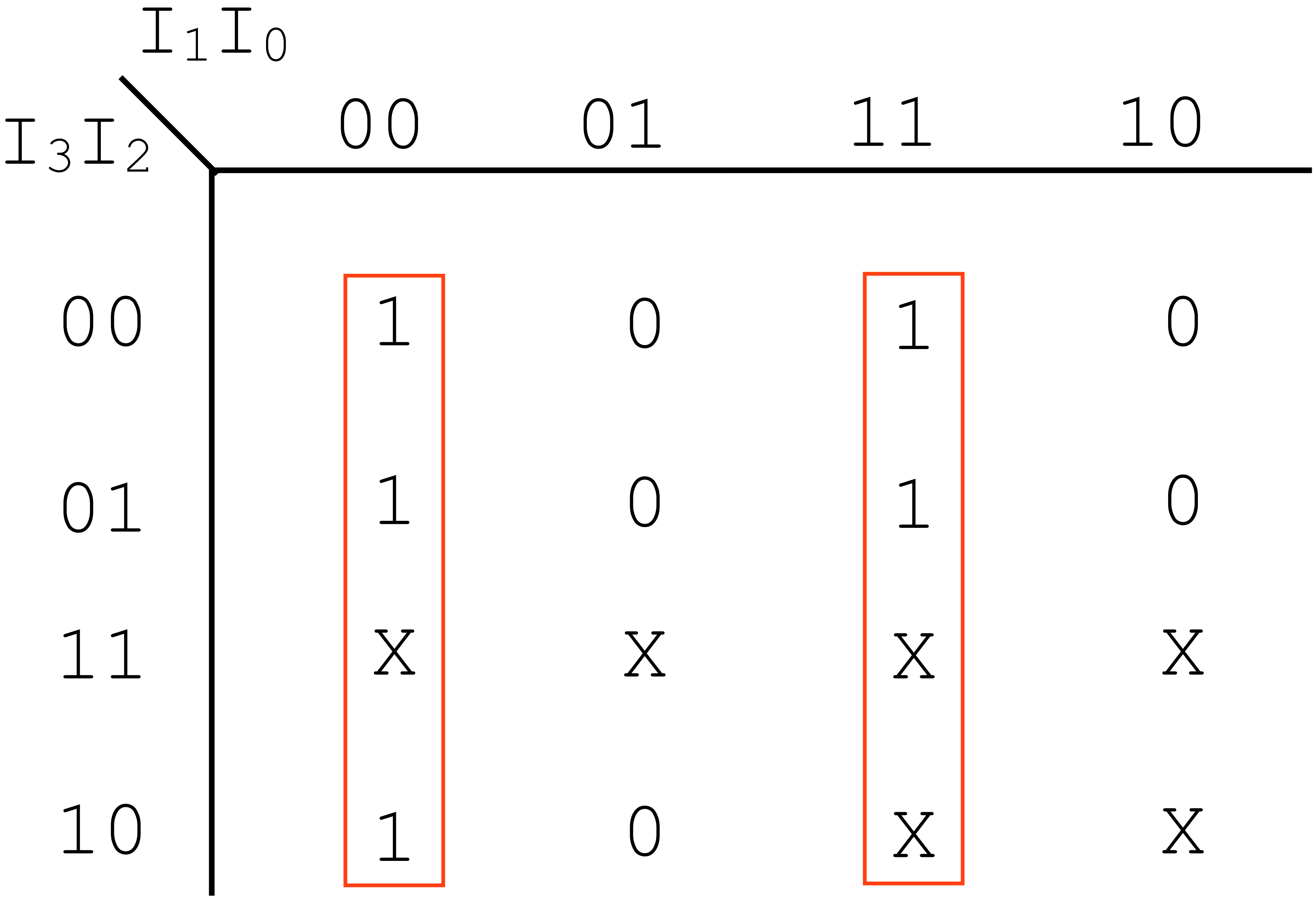


I_3I_2 \ I_1I_0	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	X	X	X	X
10	1	0	X	X

Example

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

$R = !I_1!I_0 + I_1I_0$



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