ANSWERS

A8.7 Half adders and full adders

Boolean algebra and logic circuits · A level · OCR H446 1.4.3, AQA 7517 4.6.4.1 · about 20 min

What this lesson is about

The full adder's expressions, building it from half adders, and chaining full adders into a ripple carry adder.

- The half adder, again
- Why "half"?
- The full adder
- Two half adders and an OR gate
- Adding whole numbers: the ripple carry adder

Questions

5 marks in all

1. Which expressions give a half adder's outputs? [1 mark]

- ☐ **A** S = A ⊻ B, C = A ∧ B
- ☐ **B** S = A ∨ B, C = A ∧ B
- ☐ **C** S = A ∧ B, C = A ⊻ B
- ☐ **D** S = A ⊻ B, C = A ∨ B

> **Answer:** A. The sum is 1 when the inputs differ (XOR); the carry is 1 when both are 1 (AND).

2. Why is a half adder not enough to add multi-bit numbers? [1 mark]

- ☐ **A** It has no input for the carry from the previous column
- ☐ **B** It cannot add 1 and 1
- ☐ **C** It has no sum output
- ☐ **D** It only works on the leftmost bit

> **Answer:** A. Every column except the first adds three bits, A, B and the carry in, and a half adder has only two inputs.

3. A full adder has A = 1, B = 0 and Cin = 1. Give Cout and S as two digits, Cout first. [1 mark]

> **Answer:** 10. 1 + 0 + 1 = 2, which is 10 in binary: carry out 1, sum 0.

4. How is a full adder built from half adders? [1 mark]

- ☐ **A** Two half adders, with their two carry outputs joined by an OR gate
- ☐ **B** Two half adders, with their carry outputs joined by an AND gate
- ☐ **C** Three half adders in a row
- ☐ **D** One half adder and a NOT gate

> **Answer:** A. The first adds A and B, the second adds Cin to that sum, and the two carries (never both 1) are ORed.

5. What does this program print? [1 mark]

```
def full_adder(a, b, cin):
    s1, c1 = a ^ b, a & b
    return s1 ^ cin, c1 | (s1 & cin)

carry = 0
bits = ""
for a, b in [(1, 1), (1, 0), (0, 1), (1, 0)]:
    s, carry = full_adder(a, b, carry)
    bits = str(s) + bits
print(bits, carry)
```

Answer:

```
0000 1
```

This adds 1011 + 0101 from the right: 11 + 5 = 16, so the four sum bits are 0000 with a carry out of 1.

The task: a 4-bit ripple carry adder

Write three functions. Every bit is the integer 0 or 1. - `half_adder(a, b)` returns a tuple `(sum, carry)`. - `full_adder(a, b, carry_in)` returns a tuple `(sum, carry_out)`, and must be built from two calls to `half_adder` and an OR. - `add4(x, y)` takes two strings of four characters, each `"0"` or `"1"`, with the most significant bit first. It adds them with a ripple of `full_adder` calls, starting from the rightmost bit with a carry in of 0, and returns a tuple `(total, carry)`: `total` a 4-character string of the sum bits, most significant first, and `carry` the final carry out (0 or 1). Your program may not use `int()`, `bin()`, `format()` or `sum()`. For each pair `("0110", "0111")`, `("1011", "0110")`, `("1111", "0001")` and `("0101", "1010")`, in that order, print one line in exactly this form: `0110 + 0111 = 1101 carry 0`

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

def half_adder(a, b):
    return 0, 0

def full_adder(a, b, carry_in):
    return 0, 0

def add4(x, y):
    return "0000", 0
```

The hint students can ask for: Add from the rightmost bit, which is the last character of each string, and pass each carry out into the next column's carry in. The first column has a carry in of 0. Build the answer string as you go, remembering that you are working right to left.

A solution

```
# the two lines every program starts with: the commands, then the robot
from bugbot import *
connect()

def half_adder(a, b):
    return a ^ b, a & b

def full_adder(a, b, carry_in):
    s1, c1 = half_adder(a, b)
    total, c2 = half_adder(s1, carry_in)
    return total, c1 | c2

def add4(x, y):
    carry = 0
    result = ""
    for i in range(3, -1, -1):
        a = 1 if x[i] == "1" else 0
        b = 1 if y[i] == "1" else 0
        bit, carry = full_adder(a, b, carry)
        result = str(bit) + result
    return result, carry

for x, y in [("0110", "0111"), ("1011", "0110"), ("1111", "0001"), ("0101", "1010")]:
    total, carry = add4(x, y)
    print(f"{x} + {y} = {total} carry {carry}")
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.