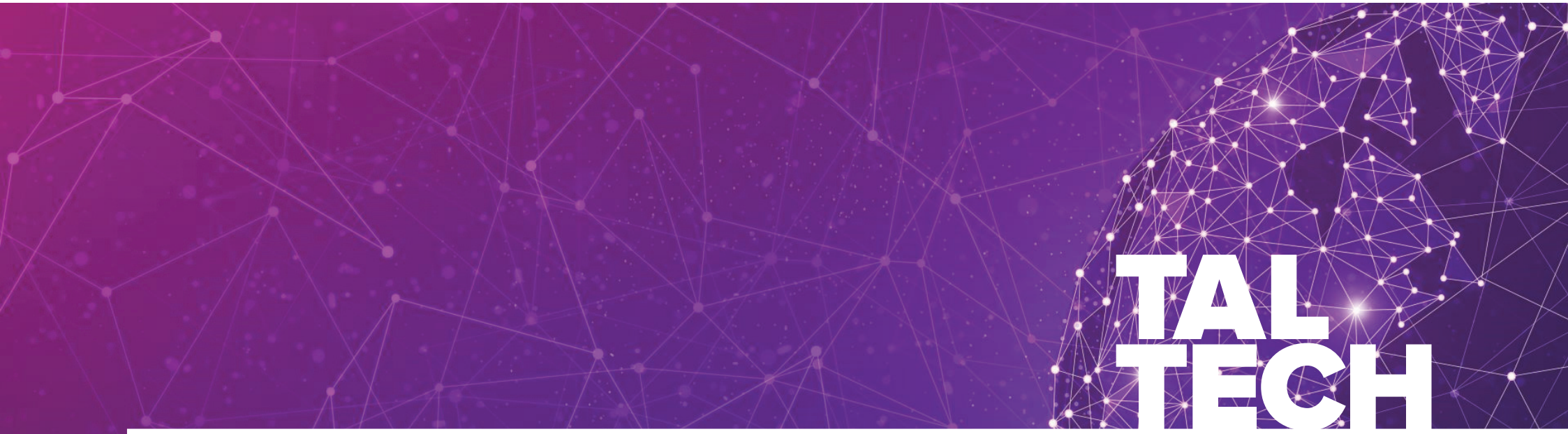



# MICROPROCESSOR SYSTEMS (IAS0430)

Department of Computer Systems
Tallinn University of Technology

10.12.2021

# COMPUTER ARITHMETIC

- **Operations on Binary**
  - **Addition**:
    - Addition is a simple operation to preform on integers.
    - The binary addition table is
      - 0 + 0 = 0
      - 0 + 1 = 1
      - 1 + 0 = 1
      - 1 + 1 = 0 and 1 as carry out

| + | 0 | 1 |
|---|---|---|
| **0** | 0 | 1 |
| **1** | 1 | 0/1 c |

    - Simply put, adding a 1 to 1 is in fact 2 in binary
      - $2_{10}$ is $10_2$ in binary
      - Which means that one bit value of 1 is carried to the next power of 2 magnitude
    - Addition is a very simple operation that can be performed using a full adder

# COMPUTER ARITHMETIC

- **Operations on Binary**
  - **Subtraction**
    - Subtraction is a simple operation to preform on integers.
    - The binary subtraction table
      - 0 - 0 = 0
      - 1 - 0 = 1
      - 1 - 1 = 0
      - 0 - 1 = 1 and 1 as borrow

| - | 0 | 1 |
|---|---|---|
| **0** | 0 | 1 |
| **1** | 1/1 b | 0 |

    - Simply put, subtracting a 1 from 0 is not possible, so we use the value found in the higher magnitude to raise the value of the 0 to 10.
    - Subtraction can be done using 2 methods:
      - Direct subtraction if the minuend is larger than the subtrahend
      - Or using addition → (4 – 10) is equivalent to (4 + (-10))
    - No need to build a subtractor if you can build a Positive to Negative and an adder.

# COMPUTER ARITHMETIC

- **Addition – Example**

```
                0011000.   carry
  13            00001101
+ 24          + 00011000
  37            00100101   result
```

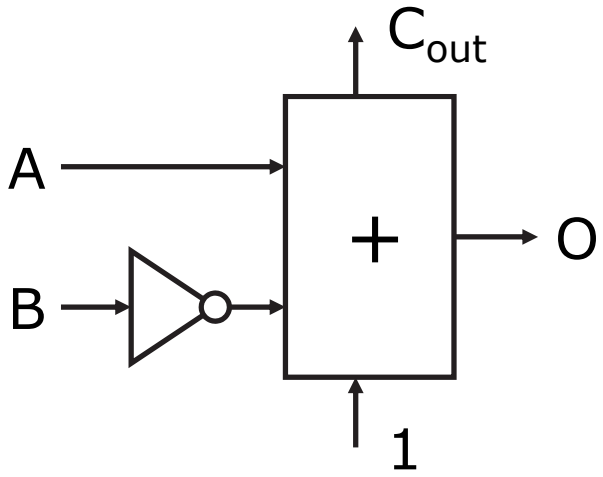
- **Subtraction – Examples**

```
                1110000.   borrow                                  1
  13            00001101                 00001101        00001101
- 24          - 00011000               + 11101000      + 11100111
 -11            11110101   result        11110101        11110101
                [ A - B ]                [ A + -B ]      [A+(~B)+1]
```

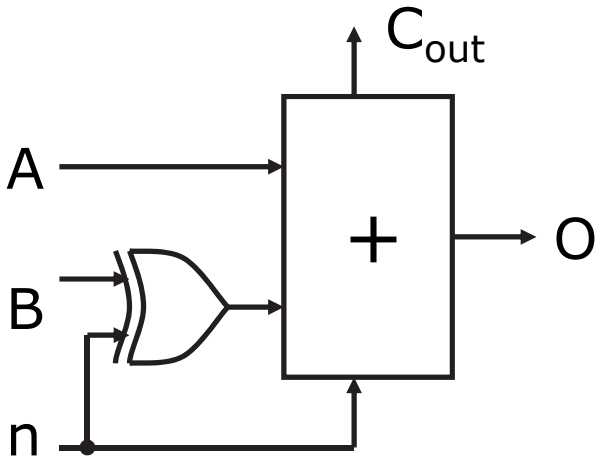
# COMPUTER ARITHMETIC IN HARDWARE

- **Adding & Subtracting**

$$O = A - B = A + (\sim B) + 1$$

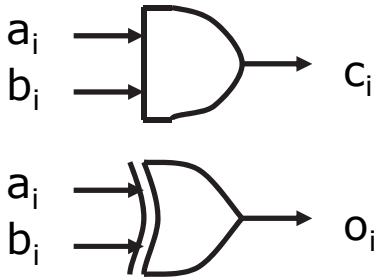



$$O = n\ ?\ (A - B) : (A + B)$$



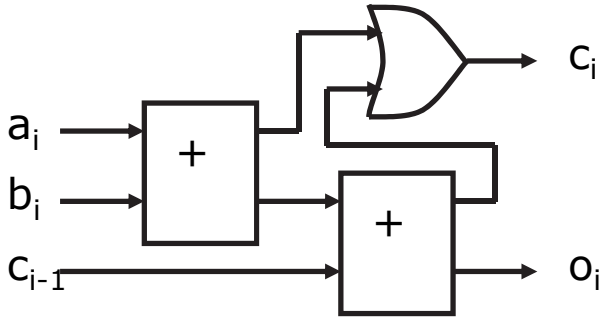
# COMPUTER ARITHMETIC IN HARDWARE

- **Half adder**
  - $(c_i,o_i) = a_i + b_i$
  - $o_i = a_i \oplus b_i$ ; $c_i = a_i \cdot b_i$




| a + b | c o |
|---|---|
| 0 + 0 | 0 0 |
| 0 + 1 | 0 1 |
| 1 + 0 | 0 1 |
| 1 + 1 | 1 0 |

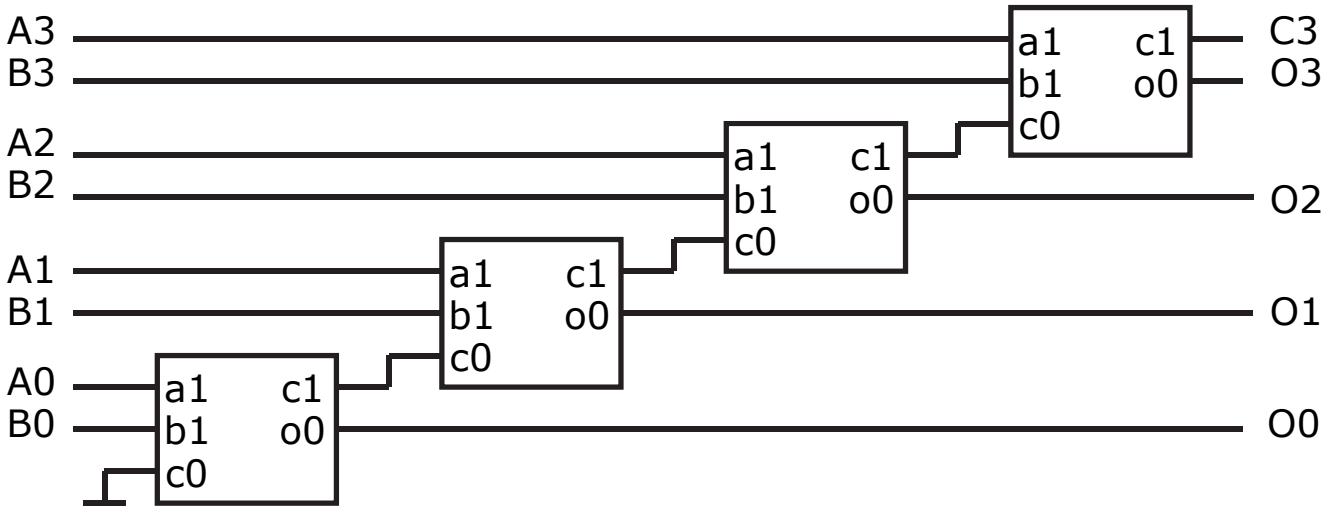
- **Full adder == two half adders**
  - $(c_i,o_i) = a_i + b_i + c_{i-1}$
  - $o_i = a_i \oplus b_i \oplus c_{i-1}$ ;
    $c_i = a_i \cdot b_i + a_i \cdot c_{i-1} + b_i \cdot c_{i-1}$




| a + b + c | c o |
|---|---|
| 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 |

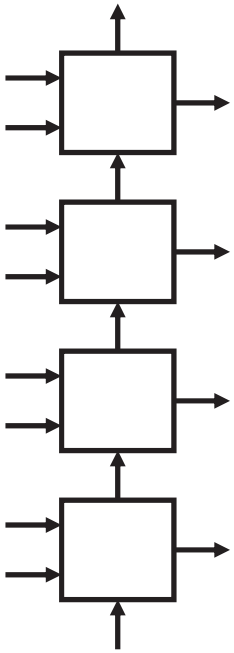
# COMPUTER ARITHMETIC IN HARDWARE

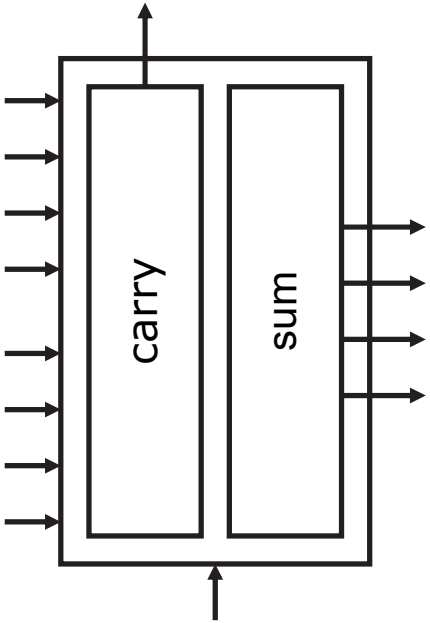
- **Adders**
  - **Ripple-carry adder**
    - The carry bits are propagated through the adder by connecting the carry out bit from a full adder to the carry in to the next full adder.
    - The carry propagation is slow

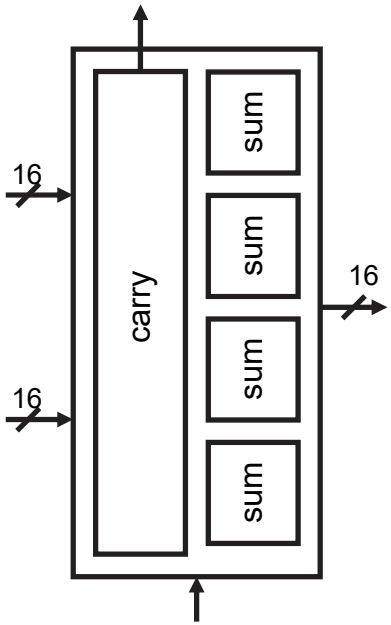


# COMPUTER ARITHMETIC IN HARDWARE

- Adders
  - Making carry propagation faster?

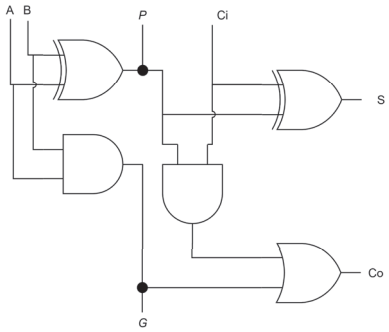




# COMPUTER ARITHMETIC

- **Adders**
  - **Carry Look-ahead Adder**
    - This adder type performs calculation of the carry bit. The Propagate (P) and the Generate (G) bits are used explicitly to generate each of the carry out of the adder.
    - Each carry out from any adder requires knowledge of the ***P*** and ***G*** values.
      - If a carry is **propagated (P)** = 1 and **Carry in** = 1 then Carry out must be 1
      - **OR** if a carry is **generated (G)** then **Carry out must be 1**
    - This can be all performed in one cycle if **P** and **G** are generated for all the adders.
    - This will give us:
      - Carry in of $0^{th}$ adder ($C_0$) will be the value of Carry in ($C_i$) to the $0^{th}$ adder
        - $C_0 = C_i$
      - Carry in of $1^{st}$ adder ($C_1$) will be 1 **<u>if</u>** $0^{th}$ adder generated a Carry ($G_0$) **OR** ($0^{th}$ adder propagated a $P_0$ a carry and its $C_0$ was 1
        - $C_1 = G_0 + P_0 \cdot C_i$

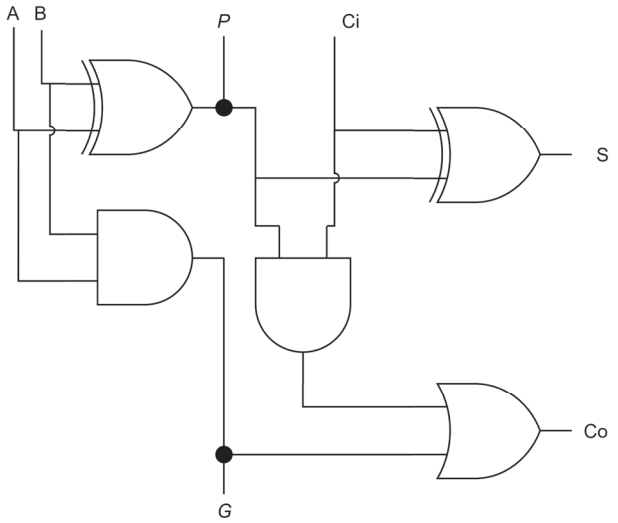


# COMPUTER ARITHMETIC

- **Adders**
  - **Carry Look-ahead Adder**
    - Following the logic from previous slide:
      - A 4-bit Carry Look-ahead Adder will have the following Carry bits:
        - $C_0 = C_i$
        - $C_1 = G_0 + P_0 C_0$
        - $C_2 = G_1 + P_1 C_1$
        - $C_3 = G_2 + P_2 C_2$
      - Since each of the carry bits are dependent on the carry bit from previous adder, the equations can be unpacked to produce the logical expression of the circuit:
        - $C_0 = C_i$
        - $C_1 = G_0 + P_0 C_i$
        - $C_2 = G_1 + P_1 G_0 + P_1 P_0 C_i$
        - $C_3 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_i$
      - A digital circuit can be constructed using the logical expression

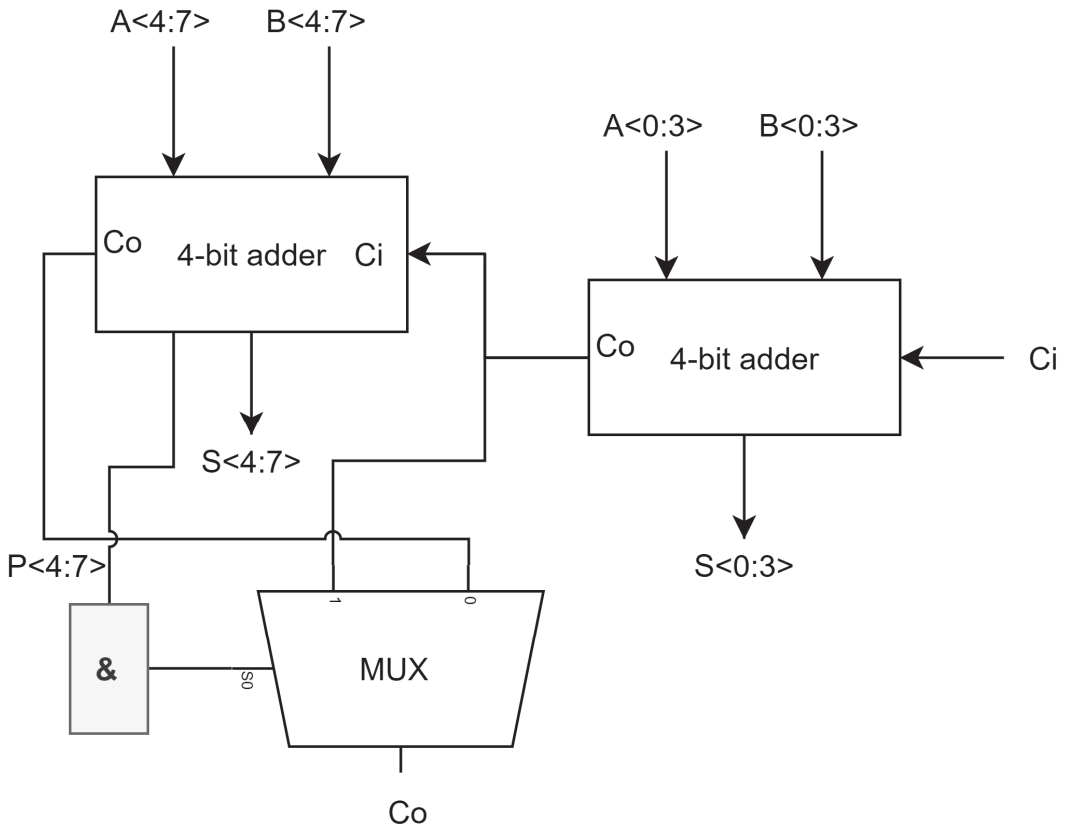
# COMPUTER ARITHMETIC

- **Adders**
  - **Carry Skip Adder**
    - This adder type allows prediction of the carry bit.
    - Two additional outputs are taken from the adder Propagate (P) and Generate (G)
      - ***P*** tells us if a carry value needs to be **propagated** to the next adder.
      - ***G*** tells us if a carry out value was **generated** in the adder regardless of the initial value of the carry in bit.
    - Using the value of **P**, we can determine if a carry bit needs to be sent into the next adder.
    - By ANDing all the P outputs in an adder series, we can determine the value of the carry bit for the next adder series faster than ripple carry.
    - Let us convert an 8-bit ripple carry adder into an 8-bit skip carry adder:



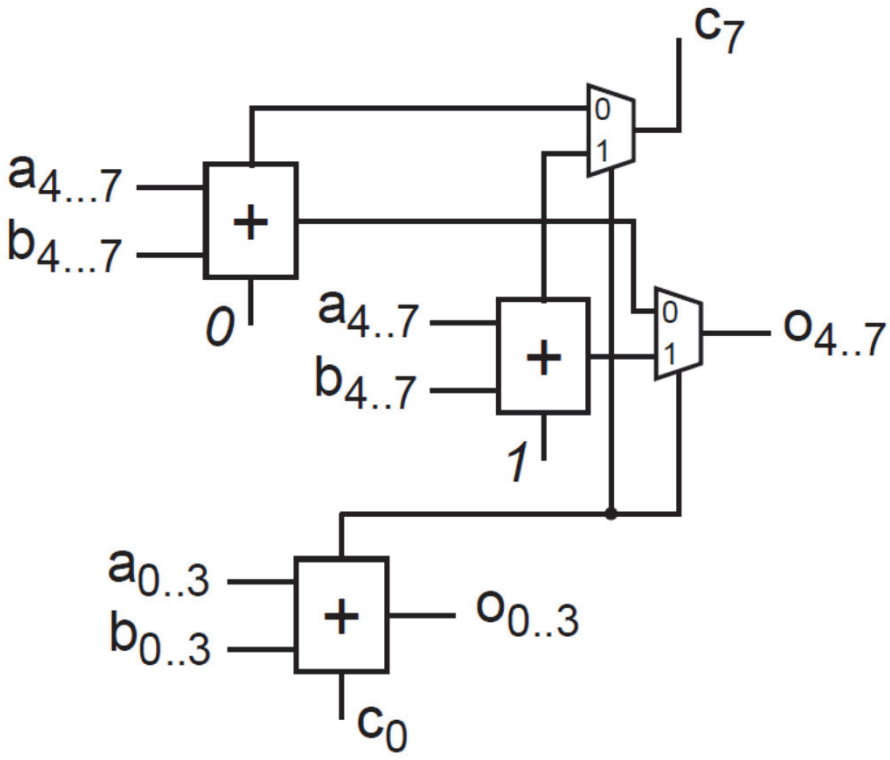
# COMPUTER ARITHMETIC

- **Adders**
  - **Carry Skip Adder**
    - If P values for all the adders on the second series is 1, then the output of the AND gate is 1. The carry from the previous adder is used as carry out.
    - If P values for the adders on the second series are not all 1, then the output of the AND gate is 0. The carry from the second adder is used as carry out.



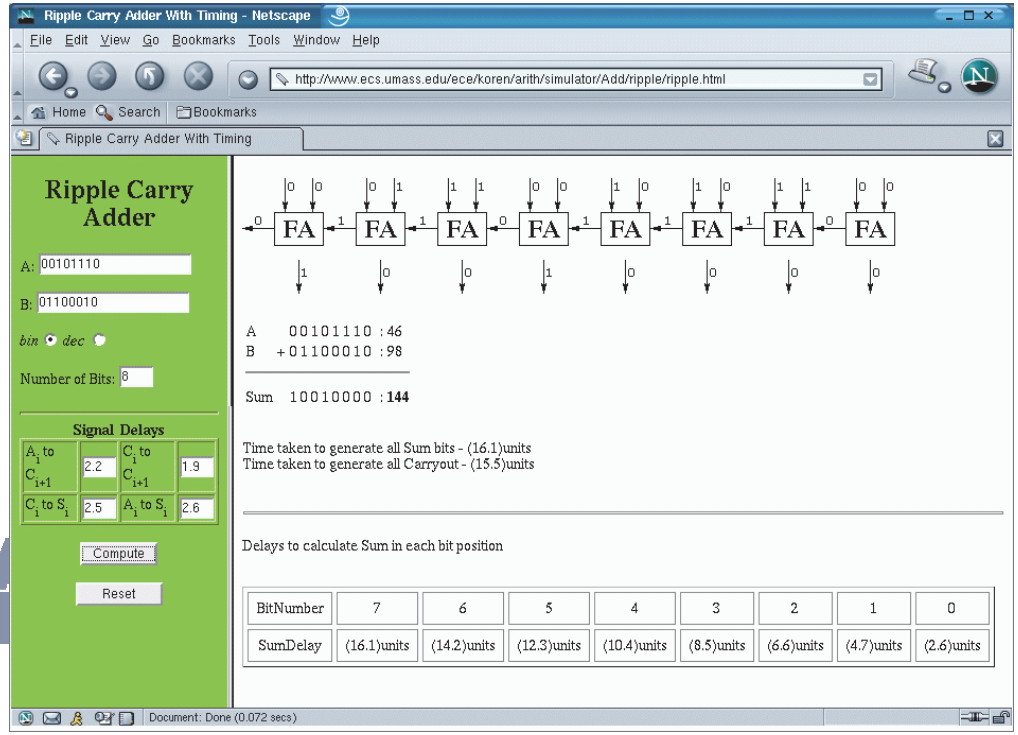
# COMPUTER ARITHMETIC

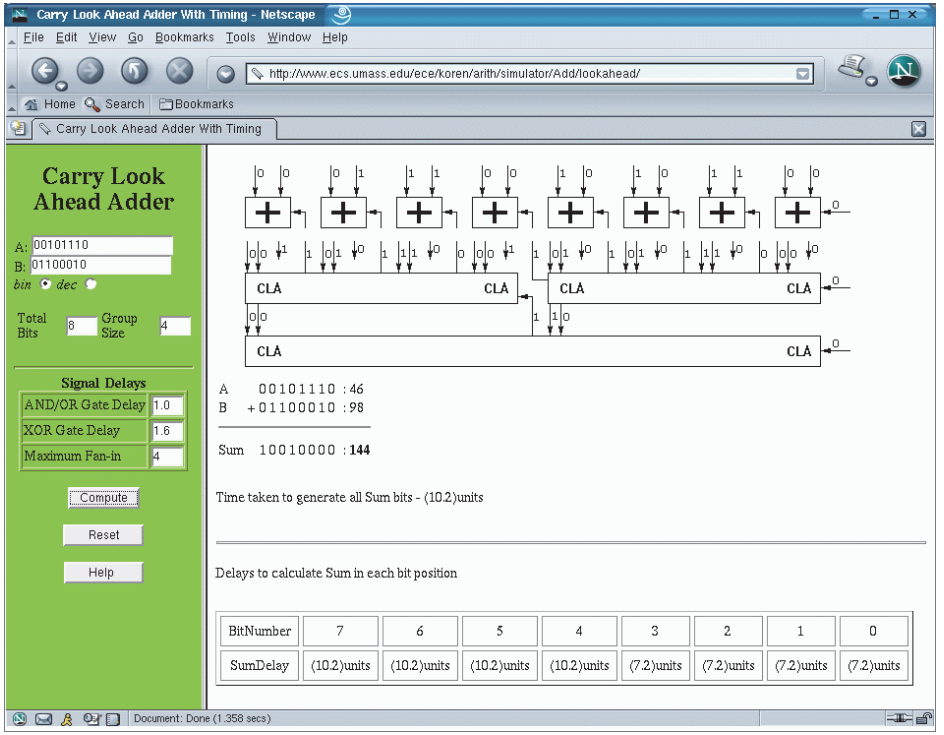
- **Adders**
  - **Carry Select Adder**
    - Speculative calculation.
    - For every group two sums and carry-outs are calculated – one with carry-in 0 and other with carry-in 1.
    - The result is selected when the actual carry-in arrives.



# COMPUTER ARITHMETIC IN HARDWARE

- **Adders**
  - Ripple-carry – the smallest but slowest.
  - Carry look-ahead, carry skip & carry select – faster but larger.
  - The size and speed depends on the block sizes and used technology.
  - Computer Arithmetic Algorithms Simulator
    http://www.ecs.umass.edu/ece/koren/arith/simulator/

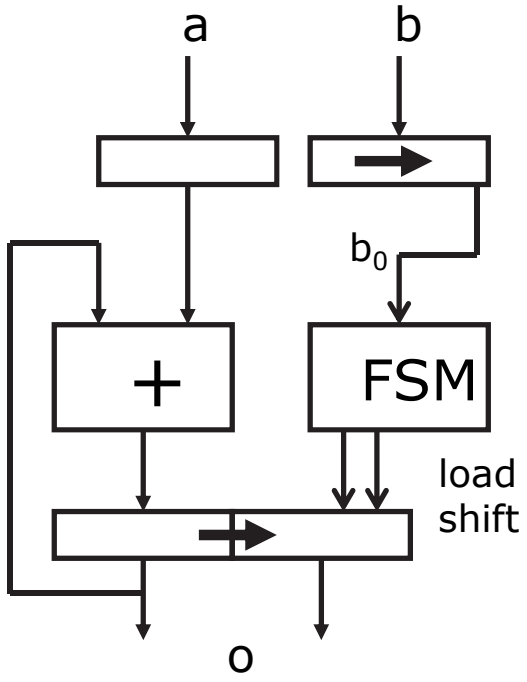


# COMPUTER ARITHMETIC IN HARDWARE

- **Multiplication**
  - **Sequential multiplier**
    - Adding & shifting – all implementations are based on this simple algorithm

```
   00011000 * 00001101     [ 24 * 13 ]
   --------------------
1.            00000000
2.           00000000.
3.          00000000..
4.         00001101...
5.        00001101....
6.       00000000.....
7.      00000000......
8.     00000000.......
   --------------------
       000000100111000     [ 312 ]
```



# COMPUTER ARITHMETIC IN HARDWARE

- **Multiplication**
  - **Sequential multiplier**
    - Making it faster?
    - Radix-4 – two bits at a time

```
   00011000 * 00001101     [ 24 * 13 ]
   ------------------
1.           00000000     00011000 == * 0
2.         00011010..     00011000 == * 2
3.       00001101....     00011000 == * 1
4.     00000000......     00011000 == * 0
   ------------------
        00000100111000     [ 312 ]
```

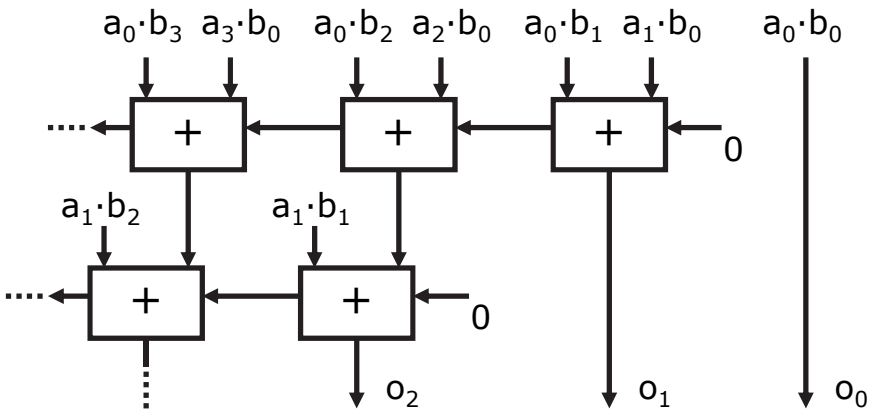
    - Adding "+ [0,1,2,3]*a" where 3*a=4*a–a – this must be remembered!

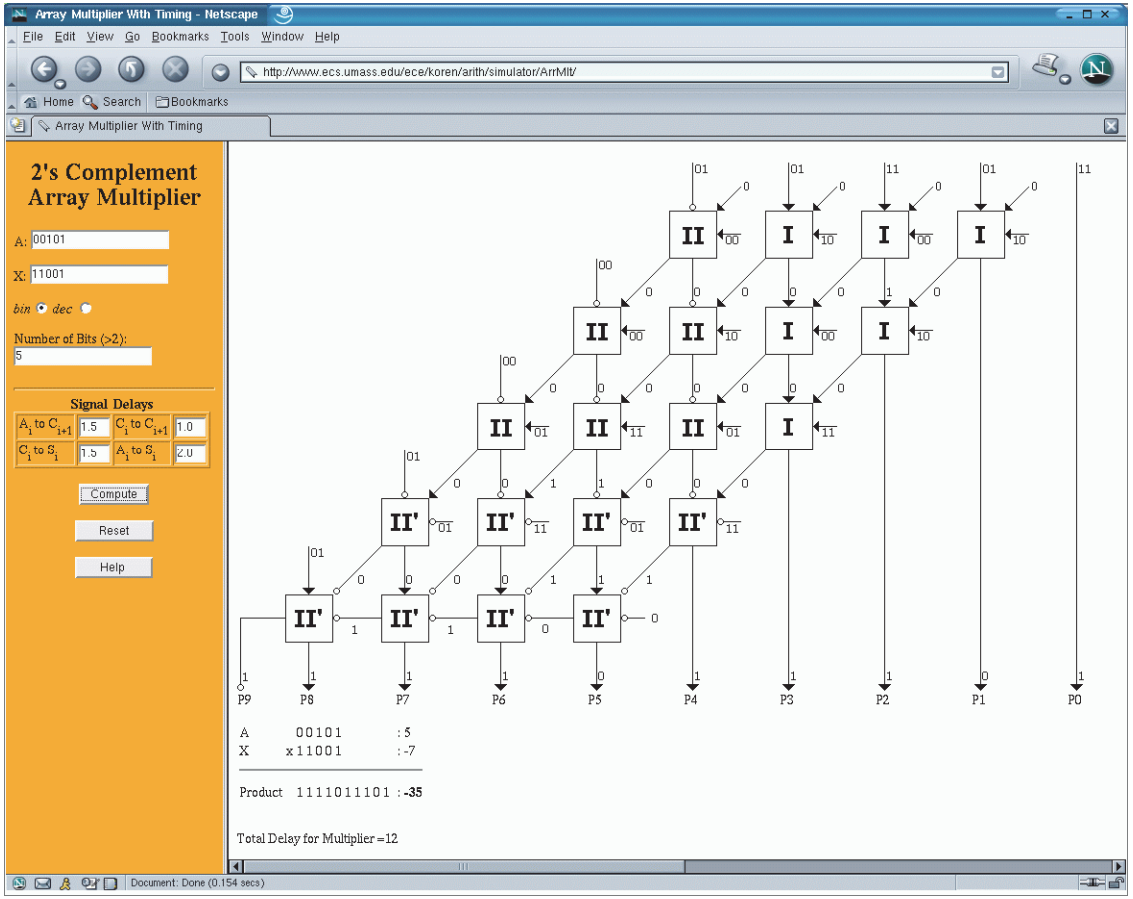
# COMPUTER ARITHMETIC IN HARDWARE

- **Multiplication**
  - **Sequential multiplier**
    - Radix-4 – two bits at a time
    - Adding "+ [0,1,2,3]*a" where 3*a=4*a–a – this must be remembered!
    - A special flag is used ~ "carry" from previous iteration
    - The flag is 0 (by default in the beginning)
      - $b_{10}$=00 ? → nothing
      - $b_{10}$=01 ? → o+=a
      - $b_{10}$=10 ? → o+=2*a [ o+=(a<<1) ]
      - $b_{10}$=11 ? → o-=a [ Flag is set to 1 ]
    - The flag is 1 (subtraction in the previous iteration)
      - $b_{10}$=00 ? → o+=a [ 4-1==3 ]
      - $b_{10}$=01 ? → o+=2*a [ 2==1+1 ]
      - $b_{10}$=10 ? → o-=a [ 3==2+1, the flag is set to 1 again ]
      - $b_{10}$=11 ? → do nothing [ 4==3+1, the flag is set to 1 again ]

# COMPUTER ARITHMETIC IN HARDWARE

- **Multiplication**
  - **Parallel multiplier**
    - Array multiplier
    - AND-gates & full adders
    - Good for pipelining



# COMPUTER ARITHMETIC IN HARDWARE

- **Division**
  - **Sequential divider**

```
312 / 13 = 24        0100111000 / 01101 = 011000
26                 - 01101
---                  ----------
 52                  0001101000
 52                 -  01101
---                  ----------
  0                  0000000000  [reminder]
===                  ==========
```

    - Subtracting, checking & shifting – all implementations are based on this algorithm
      - A binary divider can be implemented using a sequence of subtractions

# FINAL EXAM

- **Topics / Tasks**
  - Overall setup like in the midterm exam
  - Duration 60 minutes
  - Tasks are bit more complex because all materials are allowed to use
    - Help form other people is not allowed!

  - T/F questions about memories
  - Comparison of different memories
  - Mapping a short memory access trace into cache
  - Short process queue scheduling
  - Binary number representations and some arithmetic operations