Digital Techniques 2

L5: Digital Arithmetic

UNIVERSITY OF OULU
CIRCUITS and SYSTEMS

Fixed-Point Representation of Numbers

- Fixed-point (FXP) numbers have a fixed number of digits on the right hand side of the radix point. e.g.
  $0010_2 = 2_{10}$
  $00.10_2 = 0.5_{10}$
  $01.01_2 = 1.25_{10}$
- The radix point is not represented in hardware, so you have to keep track of it!

- Signed FXP numbers are usually encoded in two'2 complement format

Two's complement encoding method:

1. Select the number of bits in codewords (N)
2. Encode numbers **X** ϵ $[-2^{N-1}, 2^{N-1}-1]$ as $\mathbf{2^N + X}$ and **drop the N+1:th** bit, using codes [10..00, 10..01, ..., 01..11]
3. Example: N = 4 bits,
   X = -1
   $X_{2's\ C} = 2^4 + (-1)$
   $=16 - 1 = 10000 - 0001 = 01111$
4. We can now add positive and negative numbers:
   2 + (-1) = 0010 + 1111 = 10001

To do: Check that this works for all positive and negative numbers. Study 2's complement number encoding until you understand how it works!

Floating-Point Representation of Number

- Floating point number representation: **significant * base** $^{\text{exponent}}$
- In digital circuits, significant (S) and exponent (E) are (signed) fixed-point numbers, and base = 2
- FLP numbers are usually encoded by...
  - S = sign bit and unsigned value
  - Normalization: leading zeros in S value are omitted and E is adjusted accordingly
  - Exponent biasing: a constant value is added to exponent to make it positive (sign bit does not have to be stored)

- Floating point numbers have a (much) **larger dynamic range** [min, max] but **lower precision** than fixed point numbers
- In SystemVerilog the *float* type is **real**
- In general, floating point hardware is more complex than fixed-point
- Synthesis tools in general do not support floating point numbers, but that may be changing
- **In this course, we only use fixed-point numbers**

Working with Fixed-Point Numbers

- You have to take care that radix points stay aligned in operations
- Example: Assume **A** has radix point at right end, and B at left end

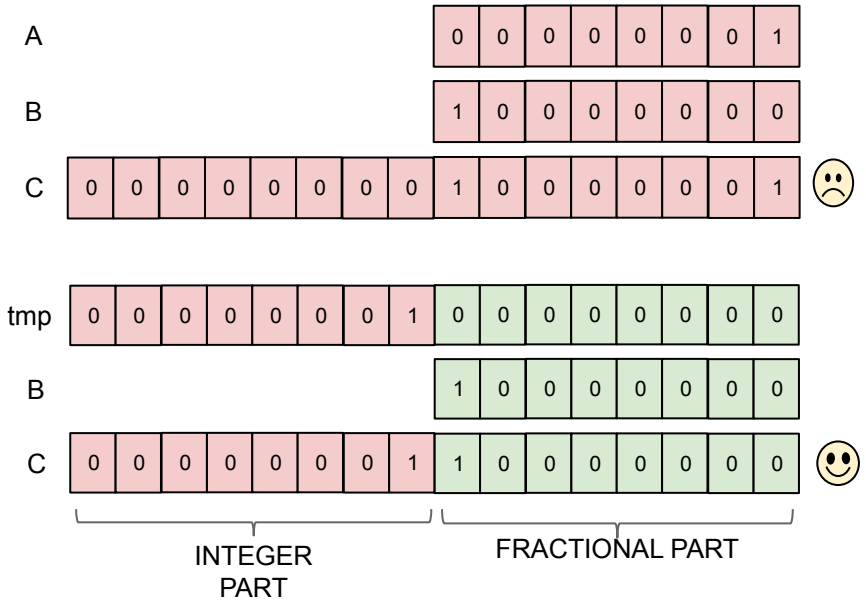
```
logic [7:0]  A, B;
logic [15:0] C, tmp;
A = 8'b00000001;
B = 8'b10000000;

C = A + B;

tmp = A << 8;
C = tmp + B
```

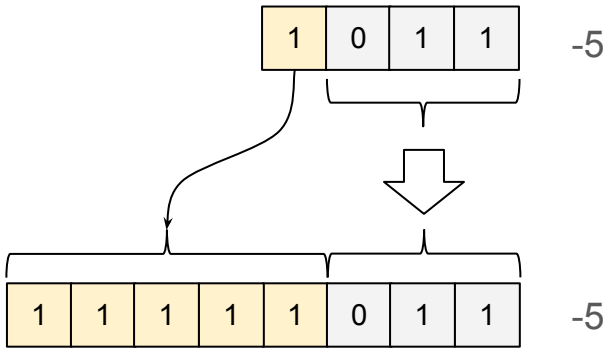
Tip: It's a good idea to use "temp" variables to make sure sign and bit-length are handled correctly,

Let A = 00000001. (1_{10}) and B = .10000000 (0.5_{10})



Manipulation of Signed FXP Numbers

- To obtain the **complement** of a 2's complement number, first invert all bits and then add a '1' to the inverted value: -111101 = 000010 + 1 = 000011 = 3
- To make a signed FXP number longer, **sign extension** must be done:



- Shortening a number from left end:
  Value may be outside the new range, is often **saturated** (clipped to max possible value) before shortening.
  Example: Shorten 01111, 10001 and 00111 to four bits (to range [-8, 7])
  $01111\ (\ 15_{10}) \Rightarrow 0111\ (\ 7_{10})$
  $10001\ (-15_{10}) \Rightarrow 1000\ (-8_{10})$
  $00111\ (\ \ 7_{10}) \Rightarrow 0111\ (\ 7_{10})$
- Shortening from the right end:
  - **Truncation** (just drop bits from the end)
  - **Rounding** + truncation (better, but requires comparator hardware)

Handling of Signed FXP Numbers in SystemVerilog Designs

- **logic** vector works like an unsigned binary number by default, but you can change the encoding if necessary:

```
logic          [7:0] a;
logic unsigned [7:0] b;
logic signed   [7:0] c;
```

- SystemVerilog is a *weakly typed* language: operations on data items that are of incompatible types are allowed **according to language rules**
- Signedness handling is an area where you certainly have to know the rules

Assume a = 1110 (14)

```
module confused (input logic [3:0] a,
                 output logic less_than_1,
                 output logic less_than_minus_1);
  always_comb
    begin
      if (a < 1)                    // (14 < 1)
       less_than_1= '1;
      else
       less_than_1= '0;

      if (a < -1)                   // (14 < 15)
       less_than_minus_1= '1;
      else
       less_than_minus_1= '0;
    end
endmodule
```

SV Standard 11.4.4, "When one or both operands of a relational expression are unsigned, the expression shall be interpreted as a comparison between unsigned values."

Some of SystemVerilog's Signedness Rules

- Decimal number literals (such as -1 or 4) are signed
- Based literal constants (**4'b1111**) are unsigned unless indicated with an 's' (**4'sb1111**)
- Packed logic arrays (**logic [3:0] x**) are unsigned unless declared *signed*
- Bit selects (**y[3]**) and range selects (**y[3:0]**) are always unsigned (gotcha!)
- Concatenation {y, x} result is unsigned
- Sign of an assignment depends only on operands (z = x + y), not on left side
- An expression is signed only if all operands are signed

How to be sure? Use **$signed** and **$unsigned** conversion functions, e.g.
instead of `if (a < -1)` write `if ($signed(a) < -1)` if **a** should be treated as signed

Survival guide: If you are uncertain, write a simple testbench program and try it out! (like in **labs/lab_datatypes**)

Some of SystemVerilog's Bit-Length Rules

- **int** and unspecified-length literals: 32 bits
- Operation **i op j** where **op** is **+, -, *, /, %, ^** etc, (L(x) = length of x)
  L(op) = max(L(i), L(j))
- Operation **i op j**, where **op** is **<<, >>, **, <<<, or >>>**
  L(op) = L(i)
- In assignments, right-hand side length also depends on left hand side length

```
logic [3:0] p, q;
logic [3:0] r;
p = 7;
q = 9;

r = (p + q) >> 1; // r == 0
```

L = max(4,4) => 7+9 overflows

```
logic [3:0] p, q;
logic [4:0] r;
p = 7;
q = 9;

r = (p + q) >> 1; // r == 8
```

L(r) = 5 => no overflow

```
logic [3:0] p, q;
logic [3:0] r;
p = 7;
q = 9;

r = (7 + q) >> 1; // r == 8
```

L(7) = 32 => no overflow

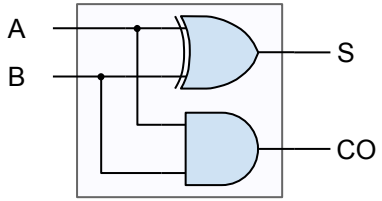
Arithmetic Hardware: Addition

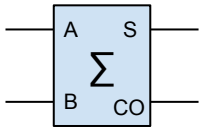
- Binary addition works just like they taught you in school:
  A = 0110, B = 1101, A+B = 10011

```
  1 1 0 0      ← Carry bits
    0 1 1 0
+   1 1 0 1
-----------
  1 0 0 1 1
```

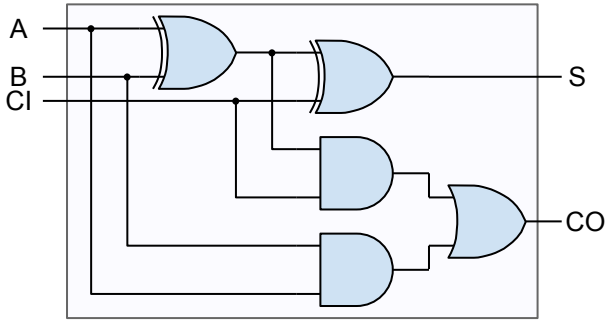
- Number of bits in sum A + B is **max(bits in A, bits in B) + 1**
- To compute A - B, first form a 2's complement of B, then add

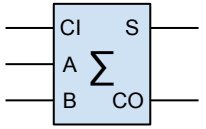
Building blocks:





Half adder

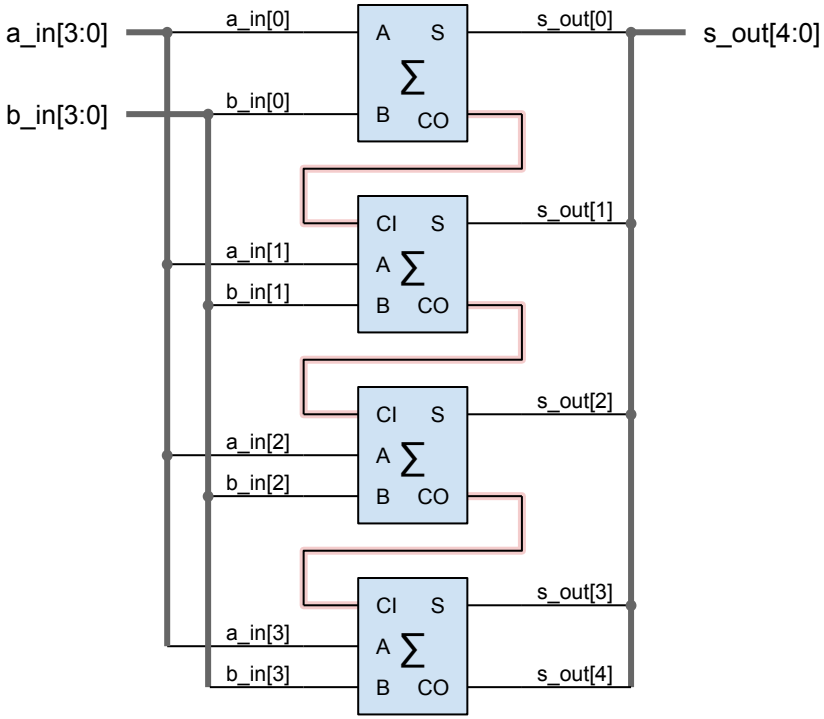




Full adder

Ripple-Carry Adder

- Simplest binary adder circuit
- 'Ripple' refers to the basic feature and limitation: the carry bit ripples from adder stage to the next
- Also known as a **carry-propagate adder** (see also: adder/subtractor)
- Delay and complexity are proportional to the number of bits in the addends
- Speeding up carry propagation is the main objective in developing faster adder architectures (at the cost of additional hardware)



Area: 17 gates
Delay: 7 gates

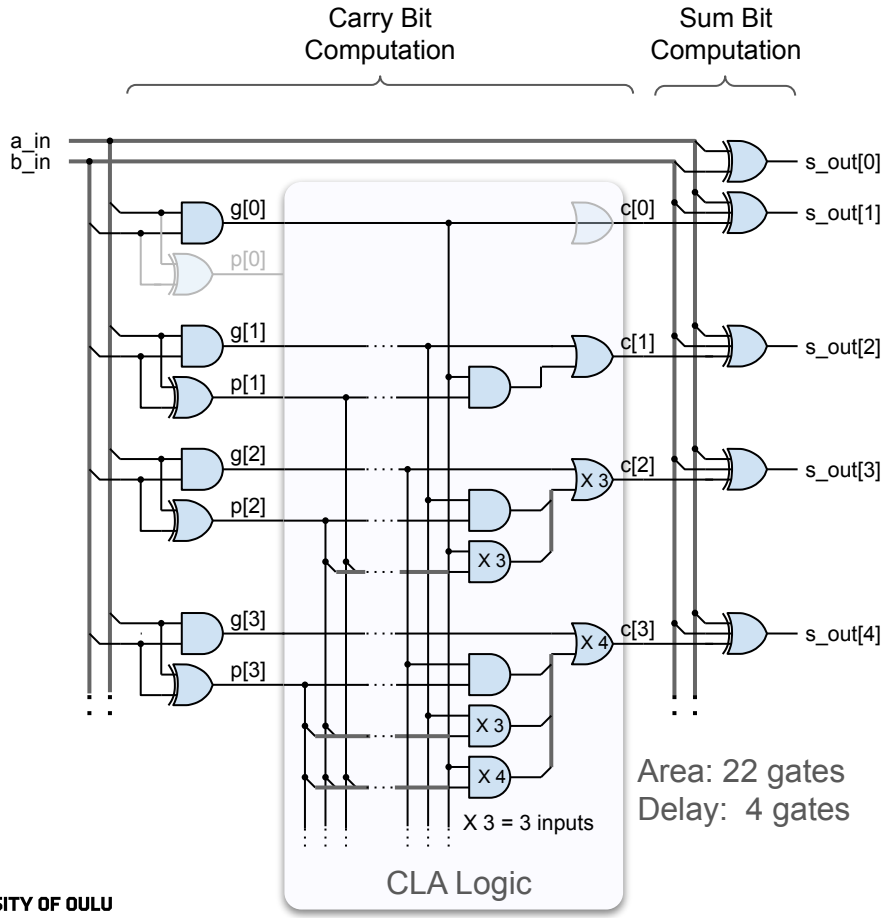
Carry Look-Ahead Adder

g[i] = Carry is generated from bits a_in[i] and b_in[i] if
$g[i] = a\_in[i] \cdot b\_in[i]$ (both are 1)
p[i] = Bits a_in[i] and b_in[i] propagate a 'lower' carry if
$p[i] = a\_in[i] \oplus b\_in[i]$ (one is 1)

The i:th carry bit then has the logic function:
$c[i] = g[i] + p[i] \cdot c[i-1]$

Three first carry bits of an adder:
$c[0] = g[0]$
$c[1] = g[1] + p[1] \cdot g[0]$
$c[2] = g[2] + p[2] \cdot g[1] + p[2] \cdot p[1] \cdot g[0]$

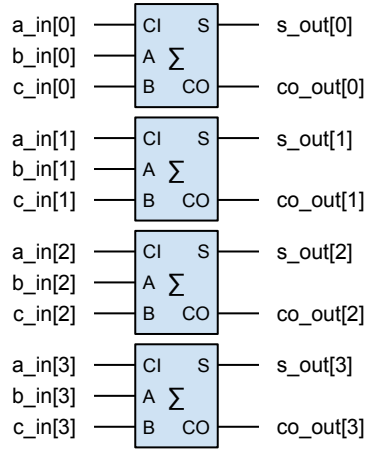
Sum bit s_out[i] is given by the XOR function:
$s\_out[i] = a\_in[i] \oplus b\_in[1] \oplus c[i-1]$

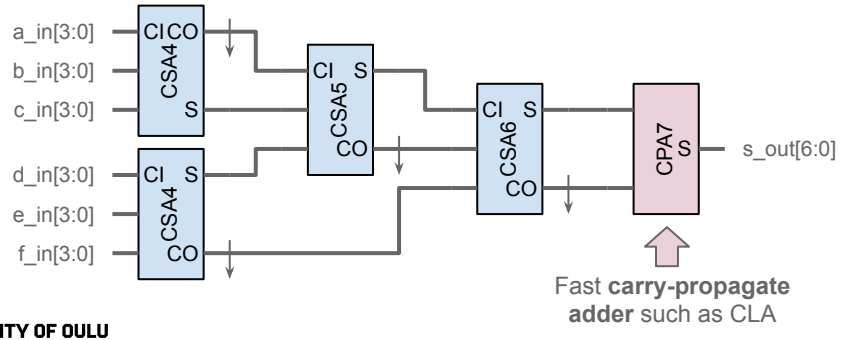


Carry-Save Addition

- Observation: A+B =
  (Sum bits of A+B) + (Carry bits of A + B) << 1
- Sum and carry bits can be generated with **carry save adder** (CSA) that does not propagate carry
- Final sum is generated by adding sum bits with left-shifted carry bits in a carry propagate adder (CPA)
- To compute a **sum of many numbers** A+B+C+... we can delay the propagation of carry by creating intermediate sum and carry vectors in CSA:s and using a CPA as a last stage
- Optimal CSA tree is called a **Wallace tree**

CSA:

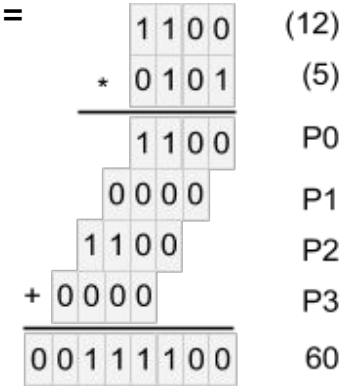




Arithmetic Hardware: Multiplication

- Works like in school, again, except that partial products (P0 - P3) are obtained with the AND operation (multiplication by 0 or 1)
- Implementation:
  - Generate partial products P with AND gates
  - Add partial products in a Wallace tree CSA
- Signed multiplication:
  - Convert operands to unsigned and fix the sign of the result
  - Use special algorithms, such as, Booth
- Number of bits in A * B is **bits in A + bits in B**

Example: 5 x 12 =



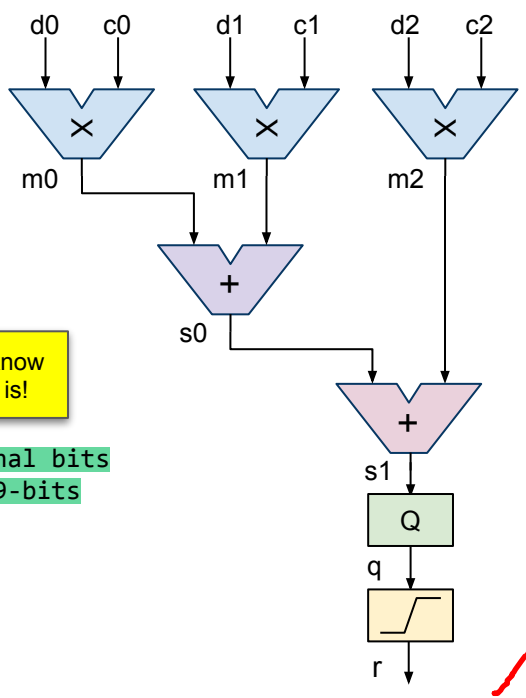
How about other arithmetic operations?
There are countless logic architectures for implementing arithmetic operations (+,-,*,/,>,<,sin, cos,...) in combinational or sequential logic, in bit-parallel and bit-serial format, with binary and 2's complement numbers etc. Consult the vast **computer arithmetic** literature to learn more.

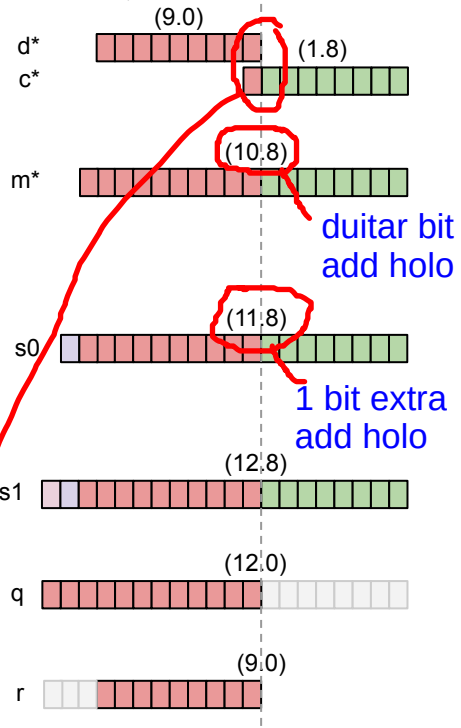
Fixed-Point Datapath Sizing Example

Input and output formats:
d's and **r**: 9-bit signed integers
c's: 9 bit FXP numbers with the point after MSB. Decim range is [-1.0, 0.99609375]

```
module fxp (input logic signed [8:0] c0, c1, c2,
            input logic signed [8:0] d0, d1, d2,
            output logic signed [8:0] r);
   logic signed [17:0]        m0, m1, m2;
   logic signed [18:0]        s0;
   logic signed [19:0]        s1;
   logic signed [11:0]        q;
   always_comb
     begin
       m0 = c0 * d0;
       m1 = c1 * d1;
       m2 = c2 * d2;
       s0 = m0 + m1;
       s1 = m2 + s0;
       q = s1 >>> 8;                  // drop fractional bits
       if (q > 9'sb011111111)         // saturate to 9-bits
         r = 9'sb011111111;
       else if (q < 9'sb100000000)
         r = 9'sb100000000;
       else
         r = q;
     end
endmodule
```

Here we must know where the point is!





duitar bit add holo

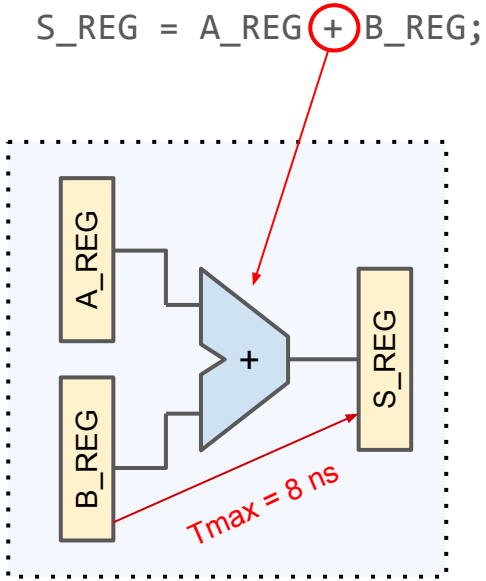
1 bit extra add holo

aida line tana hoise jee point ar age and pore kondar koi bit ase dekhar jonno

UNIVERSITY OF OULU
CIRCUITS and SYSTEMS

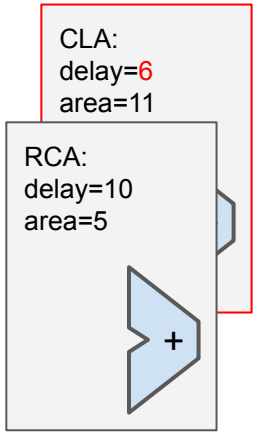
Implementation of Arithmetic Operations in Synthesis

- Logic synthesis programs' RTL component libraries contain a selection of arithmetic components (+, -, *, / etc)
- In *Design Compiler*, the RTL library is called *DesignWare*
- In RTL code translation, the synthesis program implements arithmetic operations detected in the code by inserting a library part in the unmapped design
- Library part implementation is automatically selected according to timing constraints in optimization
- Don't try to optimize the logic of an arithmetic part: it can make it slower (e.g. flattening to two-level will "optimize" CLA logic away!)



RTL architecture generated from code.

RTL COMPONENT LIBRARY



References

1. IEEE Standard for SystemVerilog
   11.6 Expression bit lengths
   11.8 Expression evaluation rules (signedness rules)
2. Harris (2007) Digital Design And Computer Architecture
   Ch 5.2 - 5.3