Addition & Subtraction

Just like in grade school (carry/borrow 1s)

```
   0111         0111         0110
 + 0110       - 0110       - 0101
 ------       ------       ------
```

Addition & Subtraction

Two's complement operations easy

– subtraction using addition of negative numbers

```
  0111   (7)           0111 (7)
+ 1010   (-6)        - 0110 (6)
```

Addition & Subtraction

Overflow (result too large for finite computer word):

- adding two n-bit numbers does not yield an n-bit number

```
    1111
+   0001
   10000
```

- How about -1 + -1?

Detecting Overflow

No overflow when adding positive to negative number

No overflow when signs are the same for subtraction

Overflow occurs when the value affects the sign:

- overflow when adding two positives yields a negative
- or, adding two negatives gives a positive
- or, subtract a negative from a positive and get a negative
- or, subtract a positive from a negative and get a positive

Consider the operations A + B, and A – B

- Can overflow occur if B is 0 ?
- Can overflow occur if A is 0 ?

Unsigned Multiplication

- Binary multiplication follows the same basic process as decimal multiplication
  - Multiplicand is multiplied by the current digit of the multiplier.
    Partial-product terms are put in proper position:

```
          0110           # A =  6
        x 1011           # B = 11
   -------------
          0110
         0110
        0000
       0110
   -------------
       1000010           # Product = 66
```

Unsigned Multiplication

– For fixed-digit multiply (computers), all digits are present in multiplier, multiplicand, partial-product terms. We just don't show all the zeros when doing hand multiplication:

```
    00000110           # A =  6
  x 00001011           # B = 11
  ----------
    00000110
    00001100
    00000000
    00110000
  ----------
    01000010           # Product = 66
```

Partial Product Terms

- Partial product terms are either zero, or the multiplicand times a power of 2
  - Recall that each power of 2 is one left shift

```
  00000110        # A =  6
x 00001011        # B = 11
----------
  00000110        # A x 2^0
  00001100        # A x 2^1
  00000000        # 0
  00110000        # A x 2^3
----------
  01000010        # Product= 66
```

Partial Product Terms (cont.)

- We can add the partial product terms to the product as they are generated:

```
  00000110         # A =  6
x 00001011         # B = 11
----------
  00000000
 +00000110
----------
  00000110         # Product =  6
 +00001100
----------
  00010010         # Product = 18
 +00000000
----------
  00010010         # Product = 18
 +00110000
----------
  01000010         # Product = 66
```

Standard Multiply Algorithm

- Multiply can be done using a series of shift and add operations:

```
product = 0
while multiplier is non-zero
  if multiplier LSB = 1
    product = product + multiplicand
  multiplier = multiplier >> 1             # look at next bit
  multiplicand = multiplicand << 1     # times 2
```

Multiply Algorithm Example

```
   00000110  # A is the multiplicand
 x 00001011  # B is the multiplier
 ----------

Cycle 1: A= 00000110, B = 00001011
        If condition is true, Product =  00000110

Cycle 2: A= 00001100, B = 00000101
        If condition is true, Product =  00010010

Cycle 3: A= 00011000, B = 00000010
        If condition is false, Product = 00010010

Cycle 4: A= 00110000, B = 00000001
        If condition is true, Product =  01000010

Cycle 5: A= 01100000, B = 00000000
        Loop ends,            Product =  01000010
```

Signed Multiplication

- Standard shift and add algorithm only works for positive numbers
- To include negative numbers with standard method must:
  - Save XOR of sign bits to get product sign bit
  - Convert multiplier/multiplicand to positive
  - Do shift and add algorithm
  - Negate result if product sign bit is 1

Truth Table for Adder Bit Slice

3 inputs (A, B, Cin); 2 outputs (Sum, Cout)

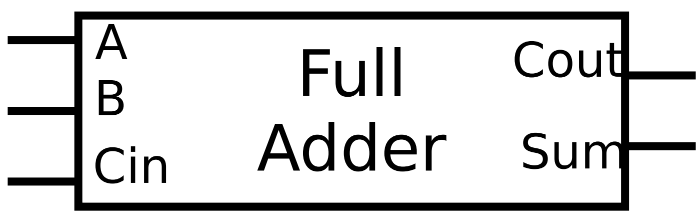
A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Adder Equations

Sum = $(A \oplus B) \oplus Cin$

Carry = AB + ACin + BCin

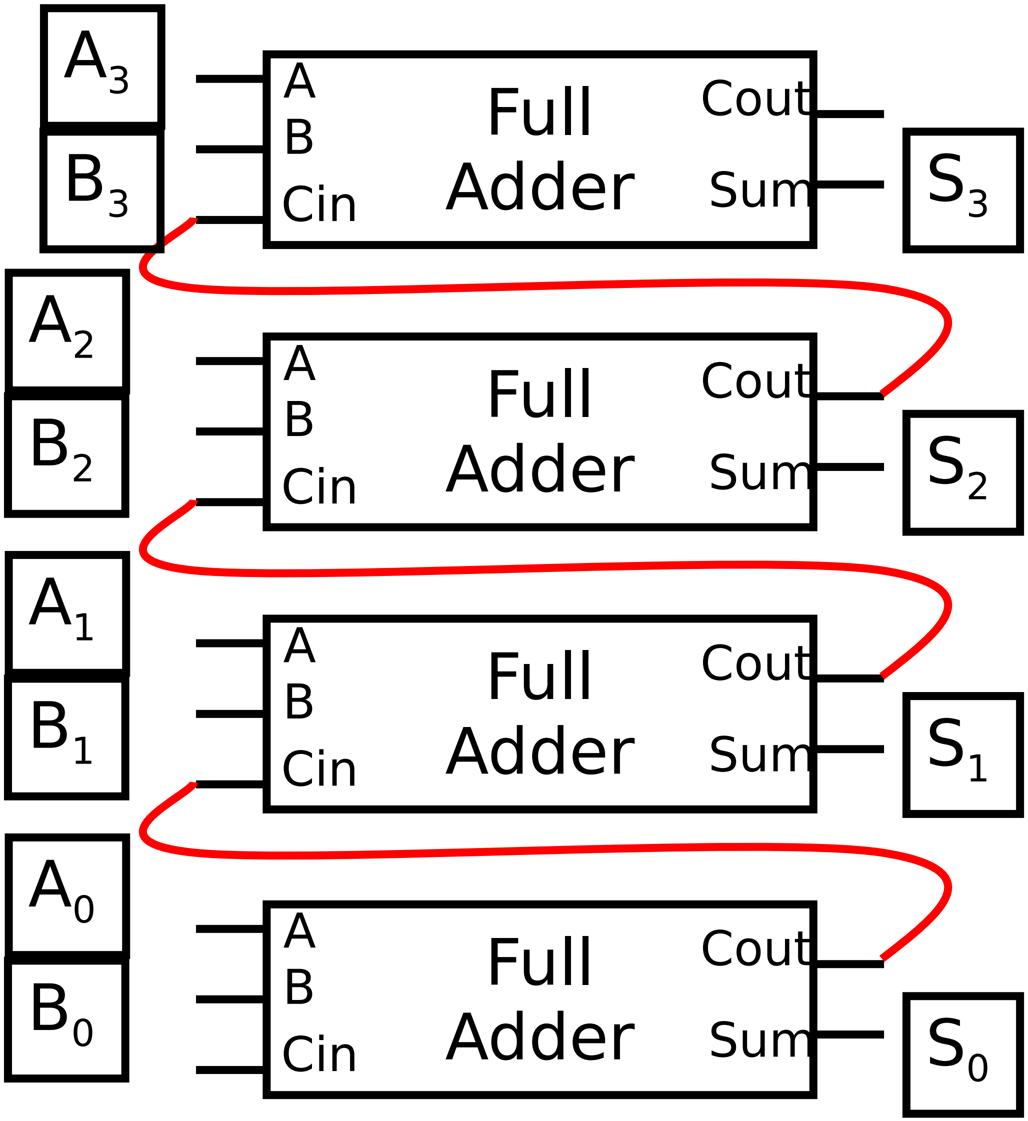
Abstract as “Full Adder”:



Cascading Adders

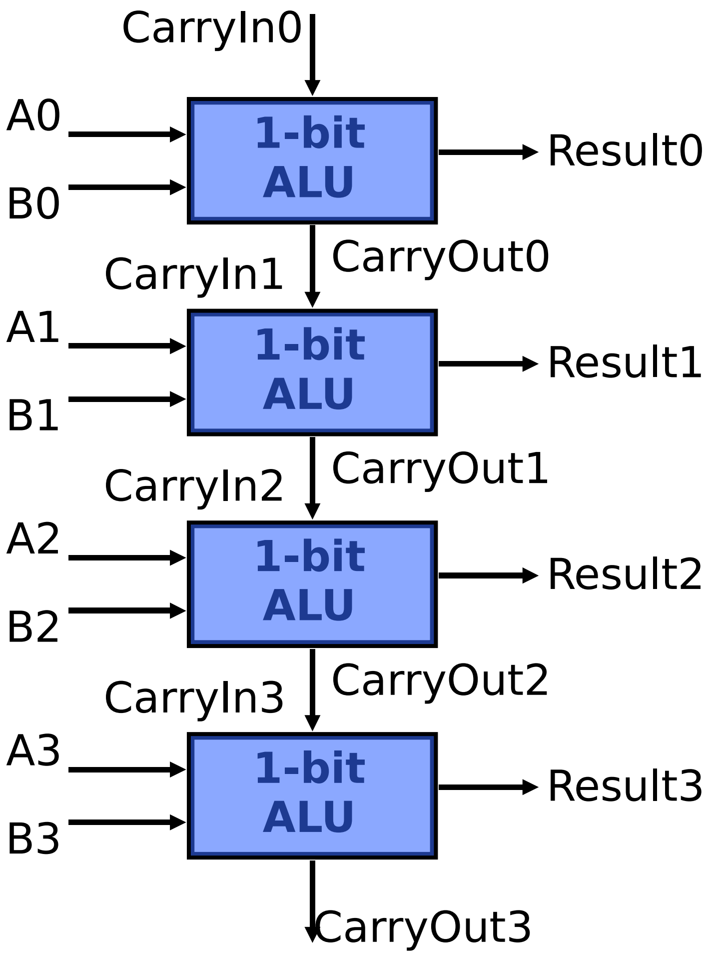
Cascade Full Adders to make multi-bit adder:

A+B=S



But What About Performance?

- Critical path of one bit-slice is CP
- Critical path of n-bit rippled-carry adder is n*CP
- Design Trick:
  - Throw hardware at it

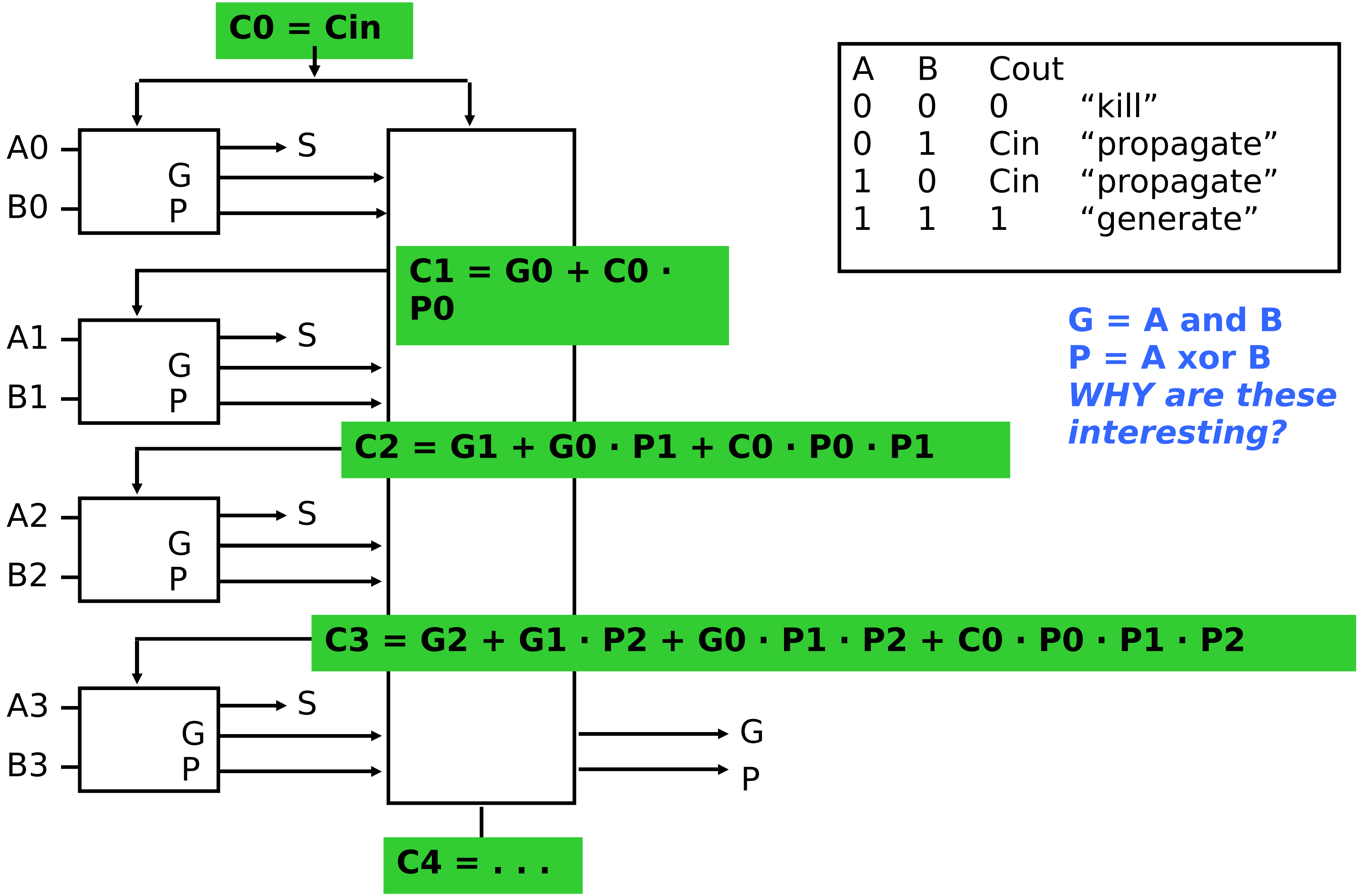


Truth Table for Adder Bit Slice

3 inputs (A, B, Cin); 2 outputs (Sum, Cout)

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0=Cin
0	1	1	0	1=Cin
1	0	0	1	0=Cin
1	0	1	0	1=Cin
1	1	0	0	1
1	1	1	1	1

Carry Look Ahead (Design trick: peek)



A	B	Cout	
0	0	0	"kill"
0	1	Cin	"propagate"
1	0	Cin	"propagate"
1	1	1	"generate"

G = A and B
P = A xor B
WHY are these interesting?

CLA vs. Ripple