U. Wisconsin CS/ECE 552
Introduction to Computer Architecture

Prof. Karu Sankaralingam

Arithmetic Part 1 (Chapter 3.1-3.5, B.5-B.6)

www.cs.wisc.edu/~karu/courses/cs552/

Slides combined and enhanced by Karu Sankaralingam from work by Falsafi, Hill, Marculescu, Nagle, Patterson, Roth, Rutenbar,Schmidt, Shen, Sohi, Sorin, Thottethodi, Vijaykumar, & Wood

Outline

- Representing Integers
  - Unsigned, 2's Complement
- Addition and subtraction
- Add/Sub ALU
  - full adder, ripple carry, subtraction,
- Carry lookahead
- Overflow
- Barrel shifter
- Defer: multiplication, division, floating-point

Unsigned Integers

- Recall:
  - n bits give rise to 2^n combinations
  - let us call a string of 32 bits as "b_{31} b_{30} . . . b_3 b_2 b_1 b_0"
- $f(b_{31} \ldots b_0) = b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0$
- Treat as normal binary number
  - e.g., 0 . . .011010101
  - = $1 \times 2^7+1 \times 2^6+0 \times 2^5+1 \times 2^4+0 \times 2^3+1 \times 2^2+0 \times 2^1+1 \times 2^0$
  - = 128 + 64 + 16 + 4 + 1 = 213
- max f (111 . . . 11) = 2^{32} - 1 = 4, 294, 967, 295
- min f(000 . . . 00) = 0
- range [0, 2^{32}-1] => # values (2^{32} -1) - 0 + 1 = 2^{32}

More Generally

- Bits have no inherent meaning
- Conventions define meaning
  - E.g., represent negative integers?
  - Seek circuit simplicity & speed
- n bits can represent finite possibilities: 2^n
  - Integers countably infinite → overflow
  - Reals uncountably infinite → overflow, underflow, imprecise

Integer Representation

- Sign Magnitude: / One's Complement / Two's Complement

Sign Magnitude	One's Complement	Two's Complement
000 = +0	000 = +0	000 = +0
001 = +1	001 = +1	001 = +1
010 = +2	010 = +2	010 = +2
011 = +3	011 = +3	011 = +3
100 = -0	100 = -3	100 = -4
101 = -1	101 = -2	101 = -3
110 = -2	110 = -1	110 = -2
111 = -3	111 = -0	111 = -1

- Balance, number of zeros, ease of arithmetic

Two's Complement Integers

- $f(b_{31}\, b_{30} \ldots b_1\, b_0) = -b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0$
  - max f(0111 . . . 11) = $2^{31} - 1$ = 2147483647
  - min f(100 . . . 00) = -2^{31} = -2147483648 (asymmetric)
- range $[-2^{31}, 2^{31}-1]$ => #values $(2^{31}-1 - -2^{31} + 1) = 2^{32}$
- E.g., -6
- 000 . . . 0110 --> 111 . . 1001 + 1 --> 111 . . .1010

Two's Complement Operations

- Negating a two's complement number: invert all bits and add 1
  - 1010 -> 0101 + 1 = 0110
  - 0110 -> 1001 + 1 = 1010
- Converting n bit numbers into numbers with more than n bits:
  - copy the most significant bit (the sign bit)

    0010 -> 0000 0010

    1010 -> 1111 1010
  - Called "sign extension"

Sign extension

- Consider representation of -2:

3bit (decimal)	2-bit (decimal)
011 (+3)	
010 (+2)	
001 (+1)	01 (+1)
000 (0)	00 (0)
111 (-1)	11 (-1)
110 (-2)	10 (-2)
101 (-3)	
100 (-4)	

Addition and Subtraction

- Similar to decimal (carry/borrow twos instead of tens)
- Identical operation for signed and unsigned
  - E.g. Unsigned vs Signed

0011	3	3
1010	10	-6
1101	13	-3

Interesting cases

- Show computation in 4-bit 2's complement representation

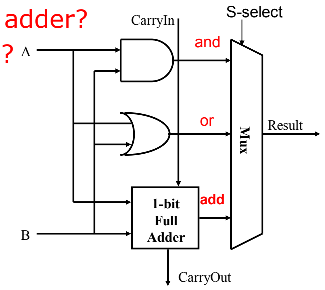
  4+4

  (-4) + (-4)

- Overflow: later

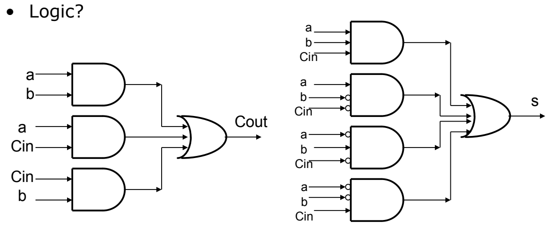
ALU bit-slice

- Bit-wise operation
  - and, or, add
  - Full adder?
  - Sub?



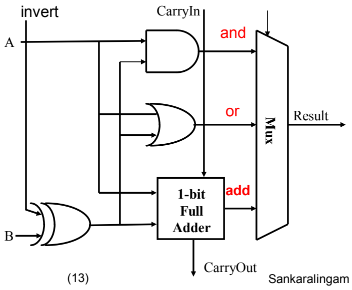
Full adder

- Three inputs and two outputs
- Cout, s = F(a,b,Cin)
  - Cout : only if at least two inputs are set
  - S : only if exactly one input or all three inputs are set
- Logic?

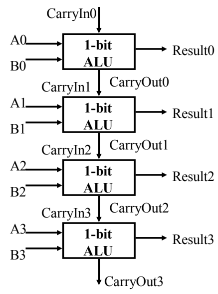


Subtract

- A - B = A + (– B)
  - form two complement by invert and add one



Ripple-carry adder



Problem : Slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
  - Delay = 32x CP(Fast adder) + XOR
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products
  - Flatten expressions to two levels

Can you see the ripple? How could you get rid of it?

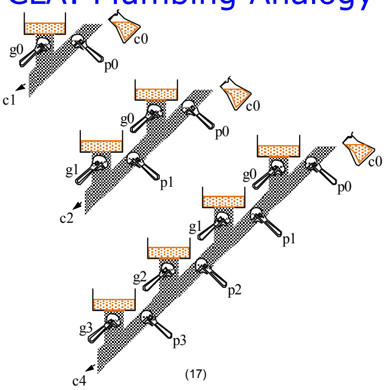
$c_1 = b_0c_0 + a_0c_0 + a_0b_0$
$c_2 = b_1c_1 + a_1c_1 + a_1b_1$
$c_3 = b_2c_2 + a_2c_2 + a_2b_2$
$c_4 = b_3c_3 + a_3c_3 + a_3b_3$

$c_2 = b_1(b_0c_0 + a_0c_0 + a_0b_0) + a_1(b_0c_0 + a_0c_0 + a_0b_0) + a_1b_1$
$c_2 = b_1b_0c_0 + b_1a_0c_0 + b_1a_0b_0 + a_1b_0c_0 + a_1a_0c_0 + a_1a_0b_0 + a_1b_1$
$c_3 = ?$
$c_{31} = ?$ Not feasible! Why? Exponential fanin

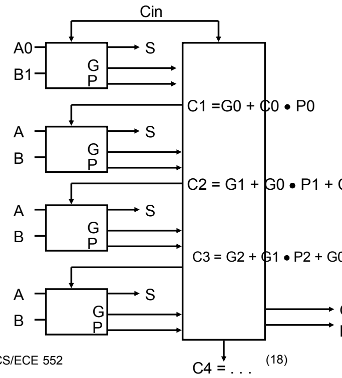
Carry look-ahead

- An approach in-between our two extremes
- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry?
    - $g_i = a_i \, b_i$
  - When would we propagate the carry?
    - $p_i = a_i + b_i$
- Did we get rid of the ripple?

CLA: Plumbing Analogy



Carry-lookahead adder



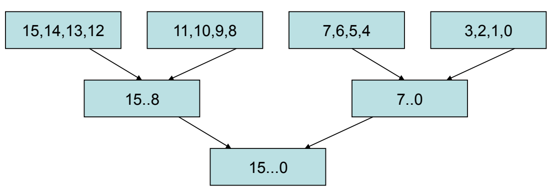
A	B	C-out	
0	0	0	"kill"
0	1	C-in	"propagate"
1	0	C-in	"propagate"
1	1	1	"generate"

G = A and B
P = A xor B

Carry-Lookahead Adder

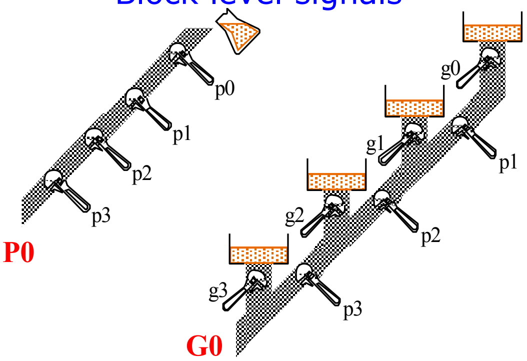
- Waitaminute!
  - Nothing has changed
  - Fanin problems if you flatten!
    - Linear fanin, not exponential
  - Ripple problem if you don't!
- Enables divide-and-conquer
- Figure out Generate and Propagate for 4-bits together
- Compute hierarchically

Carry Lookahead adder

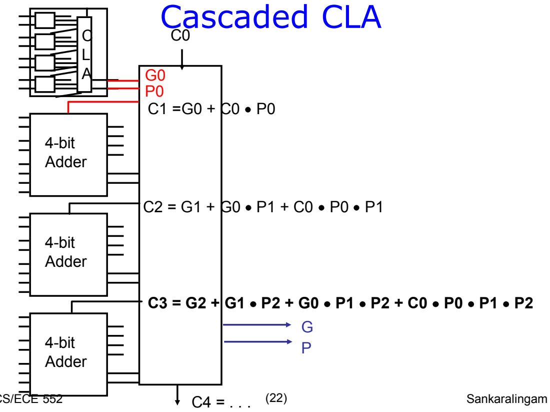


- Height of tree = O(lg(n))
- 32 bit addition : k*lg(32) = k*5

Block level signals



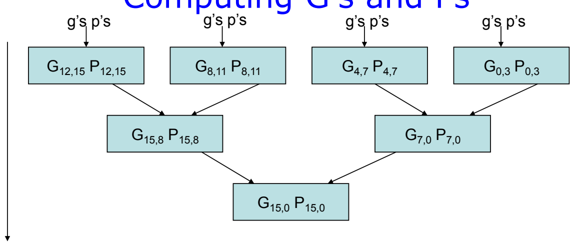
Cascaded CLA



Carry-Lookahead Adder

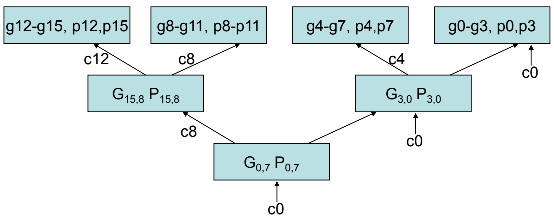
- Hierarchy
  - $G_{i,k} = G_{j+1,k} + P_{j+1,k} * G_{i,j}$ (assume $i < j+1 < k$)
  - $P_{i,k} = P_{i,j} * P_{j+1,k}$
  - $G_{0,7} = G_{4,7} + P_{4,7} * G_{0,3}$
  - $P_{0,7} = P_{0,3} * P_{4,7}$
  - $G_{8,15} = G_{12,15} + P_{12,15} * G_{8,11}$
  - $P_{8,15} = P_{8,11} * P_{12,15}$
  - $G_{0,15} = G_{8,15} + P_{8,15} * G_{0,7}$
  - $P_{0,15} = P_{0,7} * P_{8,15}$

Computing G's and Ps



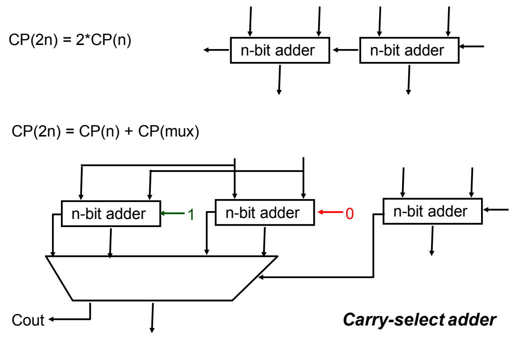
- Parallel algorithm in hardware

Computing C's



- Different tree.
- Note propagation order
- Worth spending some time to think about the intricacies

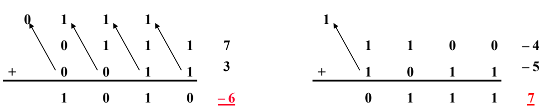
Carry-selection: Guess



Overflow

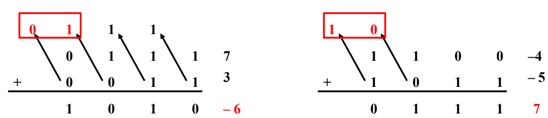
Decimal	Binary	Decimal	2's Complement
0	0000	0	0000
1	0001	-1	1111
2	0010	-2	1110
3	0011	-3	1101
4	0100	-4	1100
5	0101	-5	1011
6	0110	-6	1010
7	0111	-7	1001
		-8	1000

- **Examples: 7 + 3 = 10 but ...**
  - 4 - 5 = - 9 but ...



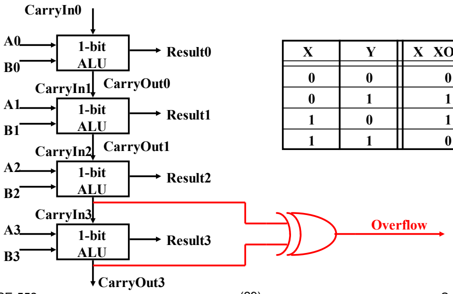
Overflow

- Overflow: the result is too large (or too small) to represent properly
  - Example: - 8 < = 4-bit binary number <= 7
- When adding operands with different signs, overflow cannot occur!
- Overflow occurs when adding:
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive
- On your own: Prove you can detect overflow by:
  - Carry into MSB ⊕ Carry out of MSB



Overflow detection

- Carry into MSB ⊕ Carry out of MSB
  - For N-bit ALU: Overflow = CarryIn[N - 1] XOR CarryOut[N - 1]



X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

Negative, Zero

- Required for conditional branches
- Zero
  - How?
  - NOR all 32 bits
  - Avoid 33rd bit (carry out)
- Negative may be required on overflow
  - If (a<b) jump : jump taken if a-b is negative
- Tempting to consider MSB
  - E.g. if (-5 < 4) branch
  - Branch should be taken, but (-5-4) computation results in overflow… so MSB is 0
  - E.g. if (7 < -3) branch
  - Branch should not be taken but (7- (-3)) results in overflow… so MSB is 1.

Shift

- E.g., Shift left logical for d<7:0> and shamt<2:0>
  - Using 2-1 muxes called Mux(select, in0, in1)
  - stage0<7:0> = Mux(shamt<0>,d<7:0>, 0 || d<7:1>)
  - stage1<7:0> = Mux(shamt<1>, stage0<7:0>, 00 || stage0<6:2>)
  - dout<7:0) = Mux(shamt<2>, stage1<7:0>, 0000 || stage1<3:0>)
- Other operations
  - Right shift
  - Arithmetic shifts
  - Rotate

Barrel Shifter