ECE437: Introduction to Digital Computer Design

Chapter 3a (addition)
Fall 2016

Midterm 2

- Date: 11/21
- Time: 8-10pm
- Place: ME 1061

- Chapter 5a+b 5.1-5.4 5.8-5.15 (not including Virtual Memory)
- Chapter 5a (all) + 5b slides (1-57 73-83) – includes parallel programming, coherence, consistency but not synchronization
- Chapter 3 3.1-3.2 + slides (all of Ch. 3a)
- Slides cover more than the book

Outline

- Basic arithmetic (Ch 3.1-3.3)
  - Representing numbers
  - 2's Complement, unsigned
  - Addition and subtraction
  - Add/Sub ALU
    - full adder, ripple carry, subtraction, together
  - Logical operations
    - and, or, xor, nor, shifts - barrel shifter
  - Carry lookahead, overflow

More on Chapter 3

- Later in semester
  - Integer multiplication, division
  - floating point representation/arithmetic
- not crucial for the lab

Unsigned Integers

- Recall:
  - n bits give rise to 2^n combinations
  - let us call a string of 32 bits as "$b_{31}\ b_{30} \ldots b_3\ b_2\ b_1\ b_0$"
- $f(b_{31} \ldots b_0) = b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0$
- 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 $(2^{32} - 1) - 0 + 1 = 2^{32}$

Numbers

- Bits are just bits (no inherent meaning)
  — conventions define relationship between bits and numbers
- Binary numbers (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: $0 \ldots 2^n - 1$
- Of course it gets more complicated:
  numbers are finite (overflow)
  fractions and real numbers ***
  negative numbers
  e.g., no MIPS subi instruction; addi can add a negative number)
- How do we represent negative numbers?
  i.e., which bit patterns will represent which numbers?

Number Representation

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

Signed Integers

- 2's complement
- $f(b_{31}\ b_{30} \ldots b_1\ b_0) = -b_{31} \times 2^{31} + \ldots + b_1 \times 2 + b_0 \times 2^0$
  - 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
  - remember: "negate" and "invert" are quite different!

- Converting n bit numbers into numbers with more than n bits:
  - MIPS 16 bit immediate converted to 32 bits for arithmetic
  - copy the most significant (the sign) bit into the other bits

```
0010  -> 0000 0010
1010  -> 1111 1010
```

  - "sign extension"
    - (remember ORI vs. LW, zero extend vs sign extend)

Negation in 2's complement

- Negation: Invert all bits, add 1
  - Why?
- If the (k+1) bit 2's complement representation of a number N is $<b_k\, b_{k-1} \ldots b_1\, b_0>$

  $N_k = -b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
- Show that the negation procedure is correct

Negation in 2's complement

- Key trick:
  - Complement of bit b can be written as (1-b)
- $N_k = -b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
- Inversion gives us

  $-(1-b_k) \times 2^k + (1-b_{(k-1)}) \times 2^{(k-1)} + \ldots + 2^1 \times (1-b_1) + 2^0 \times (1-b_0)$
- Separating the red and blue terms and adding 1

  $- 2^k + 2^{(k-1)} + \ldots + 2^1 + 2^0 + 1 - Nk$
- Blue terms plus 1 goes to zero. Q.E.D.

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)	

Mathematical basis

- Inductive proof
  - if the (k+1)-bit 2's complement representation of a number N is $\langle b_k\ b_{k-1} \ldots b_1\ b_0 \rangle$
    - $N_k = -b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
  - Then the (k+2)-bit 2's complement representation $\langle b_k\ b_k\ b_{k-1} \ldots b_1\ b_0 \rangle$ also represents N
    - $N = -b_k \times 2^{k+1} + b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
    - $= (-2 \times b_k + b_k) \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$
    - $= -b_k \times 2^k + b_{(k-1)} \times 2^{(k-1)} + \ldots + 2^1 \times b_1 + 2^0 \times b_0$

Addition and Subtraction

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

	Unsigned	signed
0011	3	3
1010	10	-6
1101	13	-3

Interesting cases

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

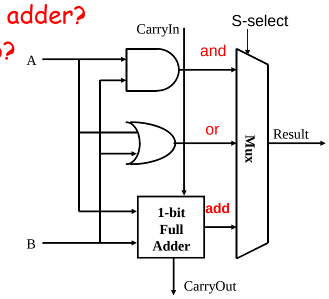
  4+4

  (-4) + (-4)

- Overflow: later

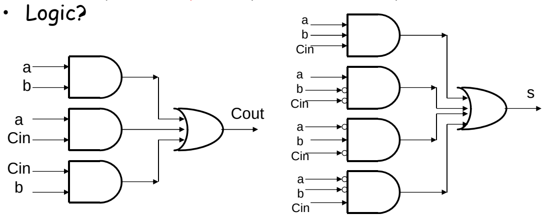
ALU bit-slice

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



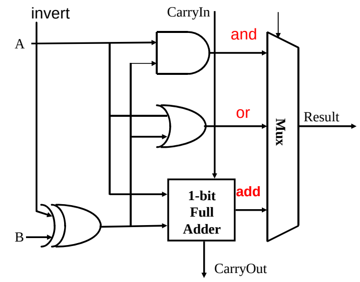
Full adder

- Three inputs and two outputs
- Cout, s = F(a,b,Cin)
  - Cout : only if atleast 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



Exercise

- How do I convert 237 to binary?
- What is the 2's complement representation (3-bit) of:
  -3:
  +5:
- If a number X is represented as b_3 b_2 b_1 b_0 in 4 bit 2's complement arithmetic, what is the 8-bit 2's complement representation of X?
- Show the computation in 4-bit 2's complement arithmetic:
  5 - (-3)

Outline

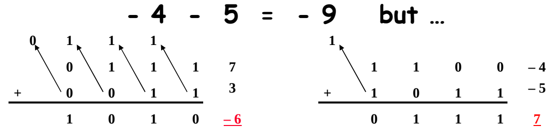
- Wind up number representation issues
  - Overflow
  - Negative
- Advanced addition techniques
  - The problem with ripple carry
  - Blocked ripple carry
  - Carry Look-ahead adder
  - Carry select adder
  - Barrel shifter

Overflow

Decimal	Binary
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

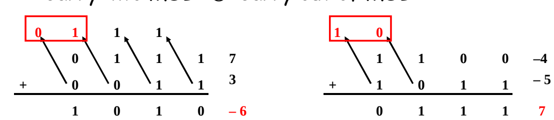
Decimal	2's Complement
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

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



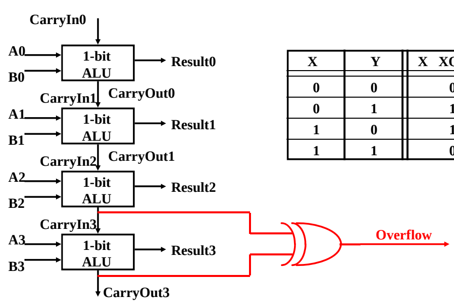
Overflow

- 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]

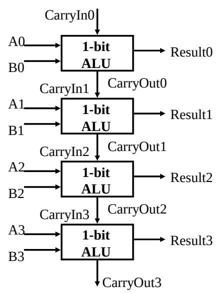


X	Y	X XOR Y
0	0	0
0	1	1
1	0	1
1	1	0

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 for 3 bits… so MSB is 0
  - E.g. if (7 < -3) branch
  - Branch should not be taken but (7- (-3)) results in overflow for 3 bits … so MSB is 1.
  - Negative = ??

Ripple-carry adder



Problem : Slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
  - Delay = 32 x Critical-path(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_2 = <7 min-terms>

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$ c_3 = <15 min-terms>

$c_4 = b_3c_3 + a_3c_3 + a_3b_3$ c_4 = <31 min-terms>

Not feasible! Why? Exponential fanin

Blocked Ripple Carry

- Flatten Logic in blocks of "k" say 4
  - But not the naïve version
  - Block of 4 requires 31-input OR gate
- Ripple carry from one block to the next
- Delay through N-bit addition
  - (N/4) * 2 + 1 (XOR)
- Reduction by a constant factor
  - Still linear in number of inputs
  - Can do better: Logarithmic delay

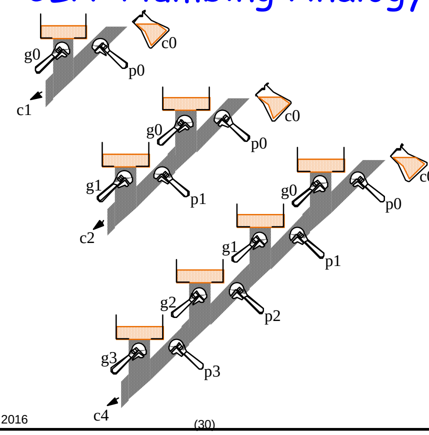
Carry Lookahead Adder

- Reformulate addition
  - Facilitates block computation
  - Facilitates hierarchical, parallel computation
  - Key concepts: Generate and Propagate
- An approach in-between our two extremes
  - Ripple carry
  - Flattened 2-level

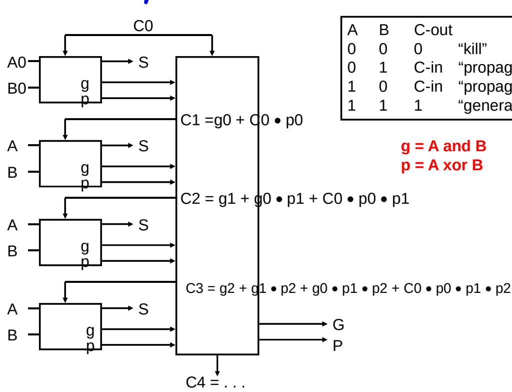
Carry look-ahead

- 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?
    - $p_i = a_i + b_i$ (slightly corrected later)
- Did we get rid of the ripple?

CLA: Plumbing Analogy



Carry-lookahead adder



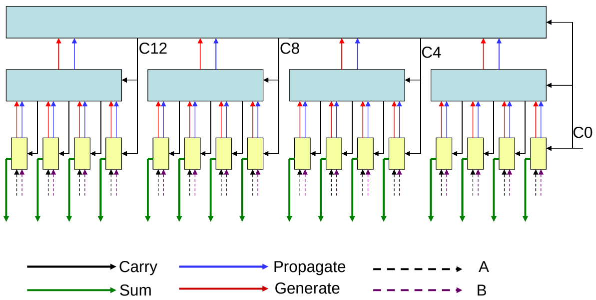
A	B	C-out	
0	0	0	"kill"
0	1	C-in	"propagate"
1	0	C-in	"propagate"
1	1	1	"generate"

g = A and B
p = A xor B

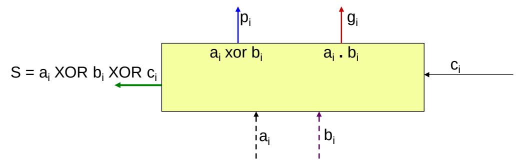
Carry-Lookahead Adder

- Waitaminute!
  - Nothing has changed
  - Fanin problems if you flatten!
    - Not really, Linear fanin, not exponential
  - Ripple problem if you don't!
- Enables divide-and-conquer
- Figure out Generate and Propagate for k-bits together
- Compute hierarchically
- Instead of linearly rippling thru blocks of k-bits (our previous idea) go up a tree of k-bit blocks (logarithmic)

2-level 16-bit CLA



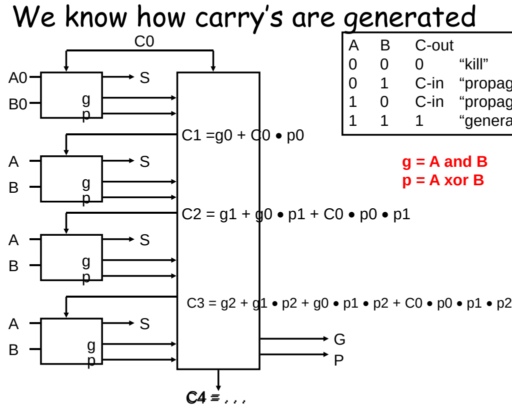
Leaf Node



- Zero level gi and pi generation
  - One gate delay
- Part of Full adder (only sum bit)
  - Two gate delays (ignoring inverters)

Intermediate nodes

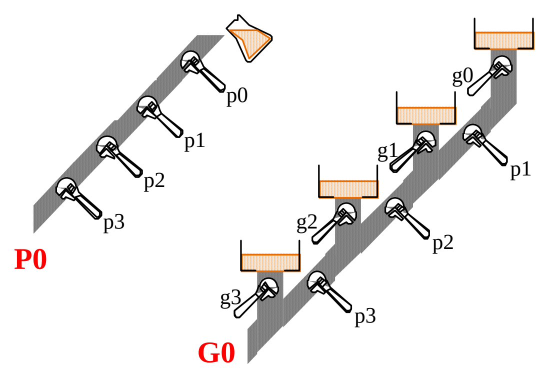
- We know how carry's are generated



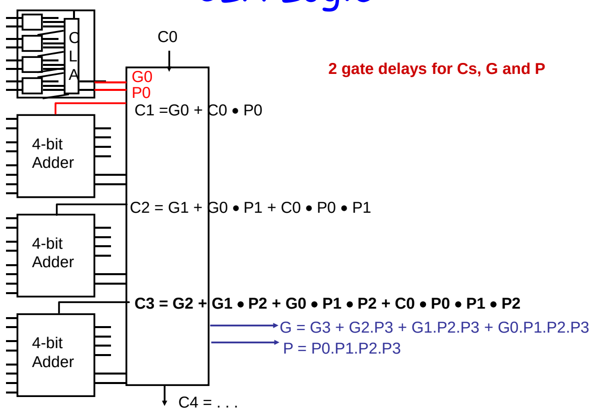
A	B	C-out	
0	0	0	"kill"
0	1	C-in	"propagate"
1	0	C-in	"propagate"
1	1	1	"generate"

g = A and B
p = A xor B

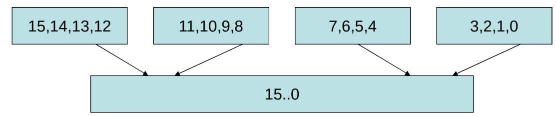
Block level signals



CLA Logic

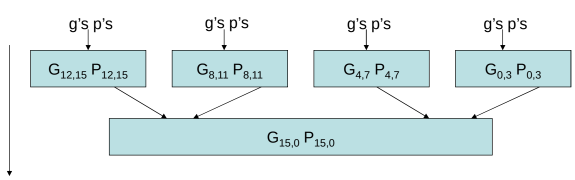


CLA: Delay Analysis



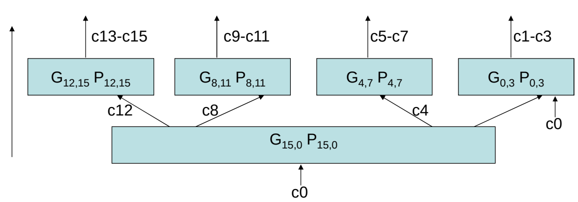
- Height of tree = $O(\log_4(n))$
- 16 bit addition : $k*\log_4(16) = k*2$ (slightly inaccurate)

Computing G's and Ps



- UP the tree
- Parallel algorithm in hardware
  - g's and p's : 1
  - First level Gs and Ps : 1 + 2 = 3
  - Second level Gs and Ps : 1 + 2 + 2 = 5
    - (not needed for 16 bit adder example)

Computing C's



- DOWN the tree
- Worth spending some time to think about the intricacies

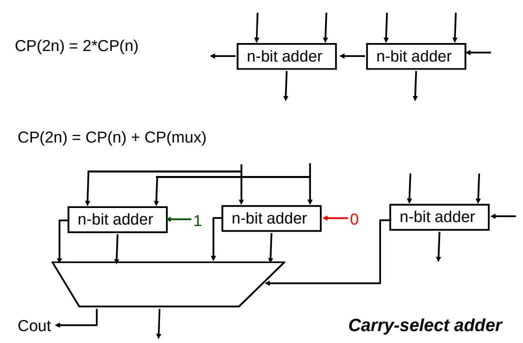
Delays

- c1—c3 : 1 + 2
- c4 : 1 + 2 + 2 **
- c5-c7 : 1 + 2 +2 + 2 = 7
- c8 : 1 + 2 + 2
- c9-c11 : 1 + 2 + 2 + 2 = 7
- c12: 1 + 2 + 2
- c13-c15 : 1 + 2 + 2 + 2 = 7

- What about sum-bits?

**** Can be 1 + 2 if computed in first level CLA block. No overall improvement by doing this**

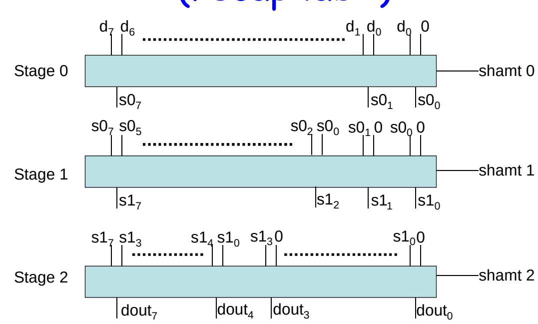
Carry-selection: Guess



Shift

- Recap lab2
- 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>, d<6:0> || 0)
  - stage1<7:0> = Mux(shamt<1>, stage0<7:0>, stage0<5:0> || 00)
  - dout<7:0> = Mux(shamt<2>, stage1<7:0>, stage1<3:0> || 0000)
- Other operations
  - Right shift
  - Arithmetic shifts
  - Rotate

Barrel Shifter (recap lab2)



Extensions

- Design a barrel shifter unit that can do
  - Right shift
  - Left shift
- Design a shifter/rotator combo that can do
  - Right rotate
  - Left rotate
  - In addition to shifts