THE UNIVERSITY of WISCONSIN MADISON

CS/ECE 552: Arithmetic and Logic

Matthew D. Sinclair

Lecture notes based in part on slides created by Mark Hill, David Wood, Mikko Lipasti, Guri Sohi, John Shen and Jim Smith

Basic Arithmetic and the ALU

- Number representations: 2’s complement, unsigned
- Addition/Subtraction
- Add/Sub ALU
  - Full adder, ripple carry, subtraction
- Logical operations
  - and, or, xor, nor, shifts
- Overflow

Basic Arithmetic and the ALU

- Covered later in the semester:
  - Integer multiplication, division
  - Floating point arithmetic
- These are not crucial for the project

Background

- Recall
  - n bits enables 2^n unique combinations
- Notation: $b_{31}\ b_{30} \ldots b_3\ b_2\ b_1\ b_0$
- No inherent meaning
  - $f(b_{31} \ldots b_0)$ => integer value
  - $f(b_{31} \ldots b_0)$ => control signals

Background

- 32-bit types include
  - Unsigned integers
  - Signed integers
  - Single-precision floating point
  - MIPS instructions (refer to book)

Unsigned Integers

- $f(b_{31}...b_0) = b_{31} \times 2^{31} + ... + b_1 \times 2^1 + b_0 \times 2^0$
- Treat as normal binary number

  E.g. 0…01101010101

  $= 1 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^1 + 1 \times 2^0$

  $= 128 + 64 + 16 + 4 + 1 = 213$
- Max $f(111...11) = 2^{32} – 1 = 4{,}294{,}967{,}295$
- Min $f(000...00) = 0$
- Range $[0,2^{32}-1]$ => # values $(2^{32}-1) – 0 + 1 = 2^{32}$

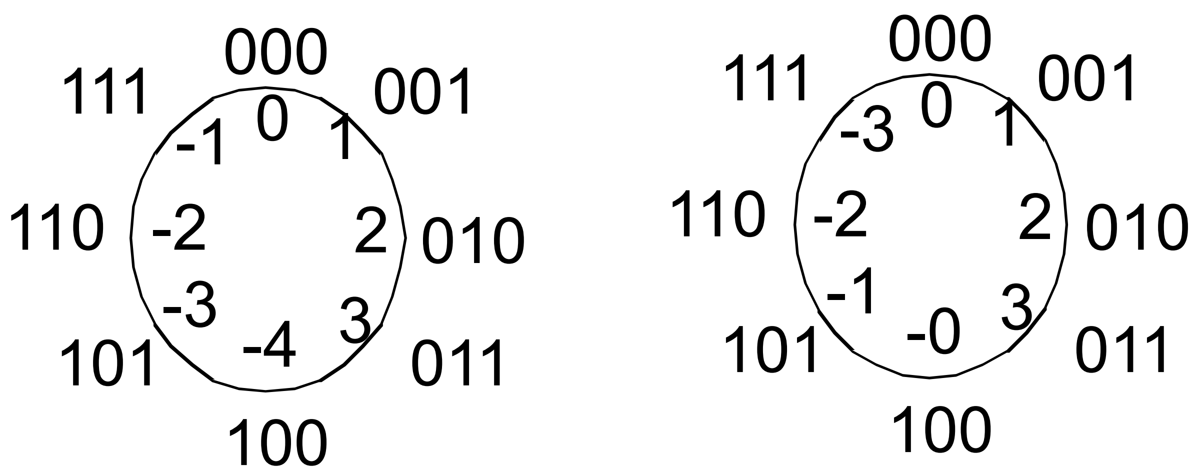
Signed Integers

- 2's complement

  $f(b_{31}...b_0) = -b_{31} \times 2^{31} + ... b_1 \times 2^1 + b_0 \times 2^0$
- Max $f(0111...11) = 2^{31} - 1 = 2147483647$
- Min $f(100...00) = -2^{31} = -2147483648$ (asymmetric)
- Range$[-2^{31}, 2^{31}-1]$ => # values$(2^{31}-1 - -2^{31}) = 2^{32}$
- Invert bits and add one: e.g. –6
  - 000…0110 => 111…1001 + 1 => 111…1010

Why 2's Complement

- Why not use sign-magnitude?
- 2's complement makes hardware simpler
- Just like humans don't work with Roman numerals
- Representation affects ease of calculation, not correctness of answer



Addition and Subtraction

- 4-bit unsigned example

0	0	1	1		3
1	0	1	0		10
1	1	0	1		13

- 4-bit 2's complement – ignoring overflow

0	0	1	1		3
1	0	1	0		-6
1	1	0	1		-3

Subtraction

- A – B = A + 2's complement of B
- E.g., 3 – 2

0	0	1	1		3
1	1	1	0		-2
0	0	0	1		1

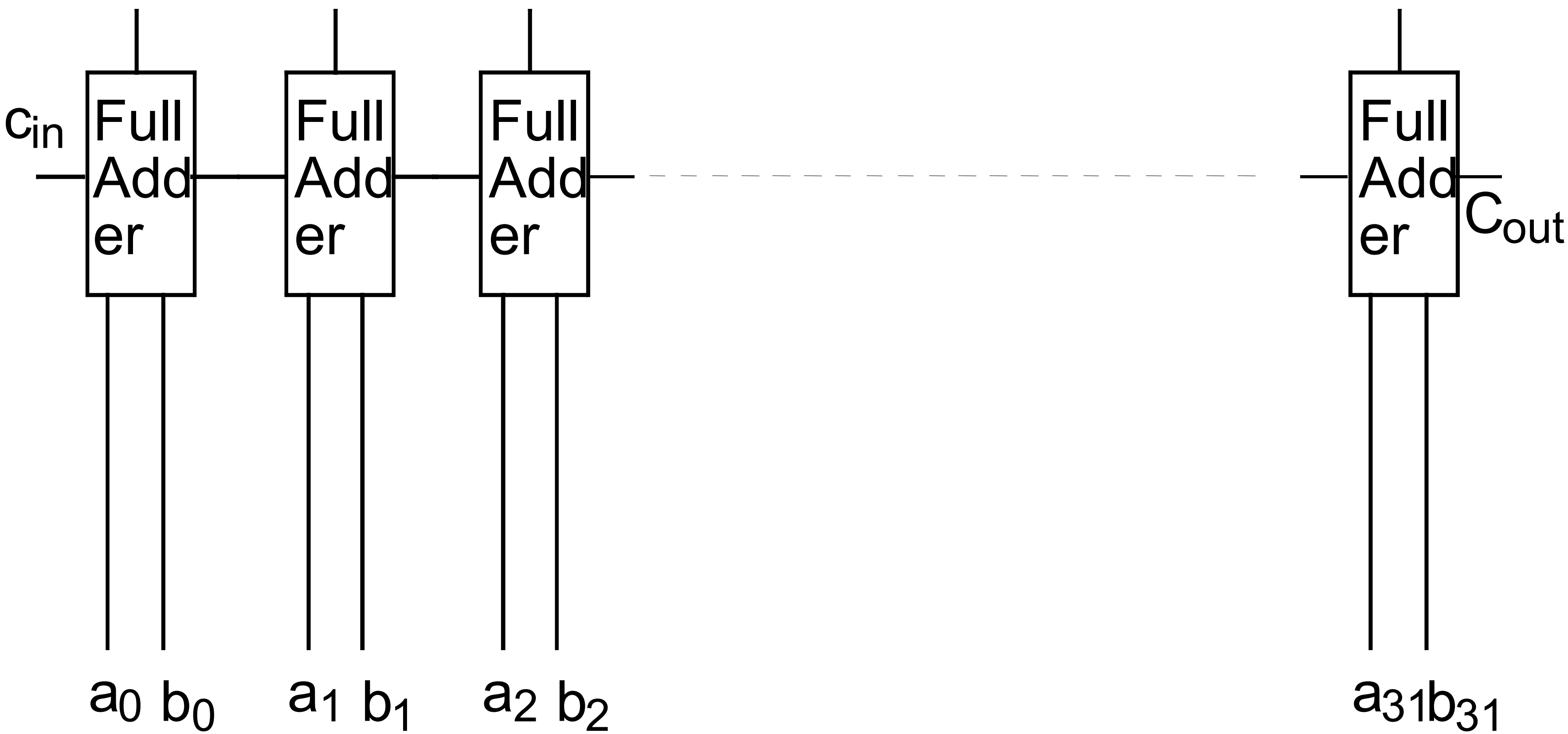
Full Adder

- Full adder $(a,b,c_{in}) \Rightarrow (c_{out}, s)$
- c_{out} = two or more of (a, b, c_{in})
- s = exactly one or three of (a,b,c_{in})

a	b	c_{in}	c_{out}	s
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

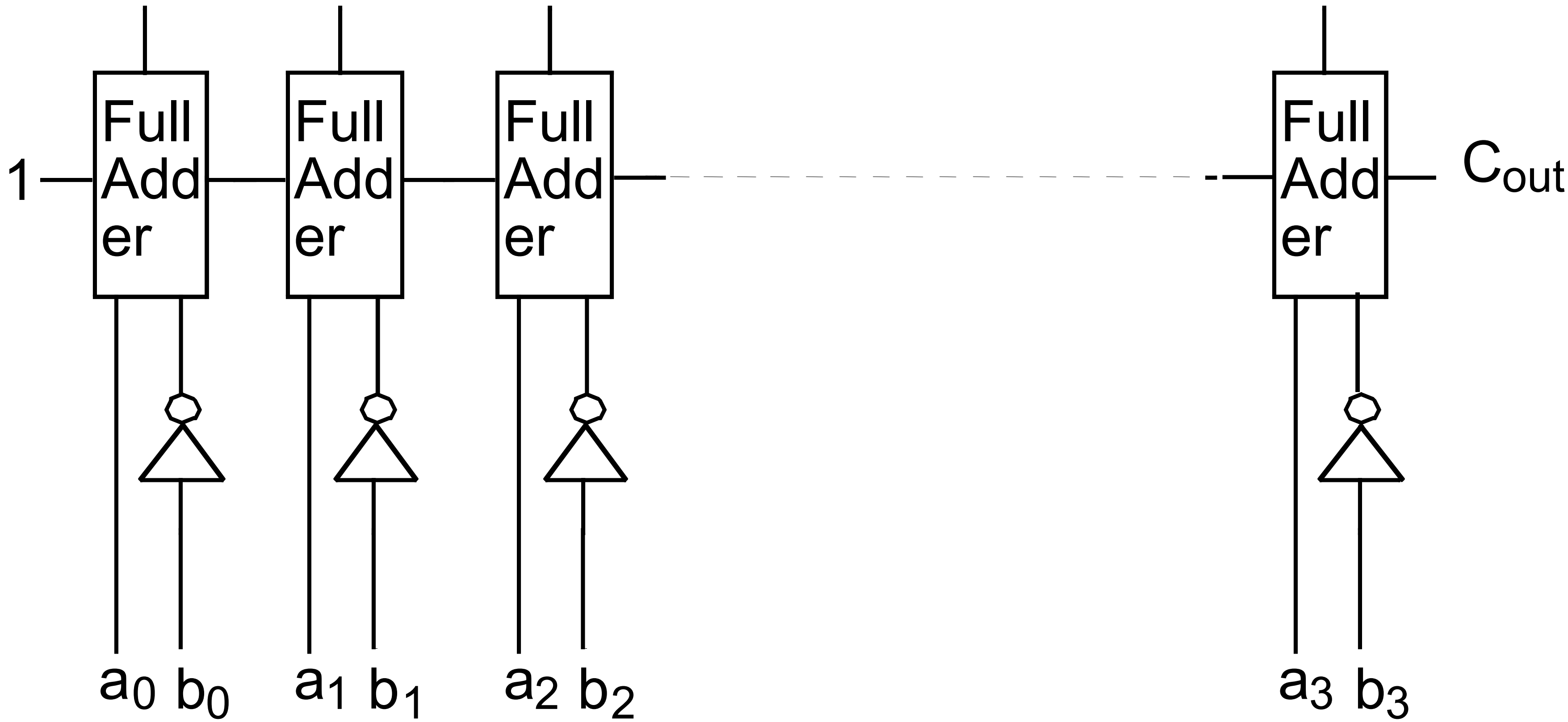
Ripple-carry Adder

- Just concatenate the full adders



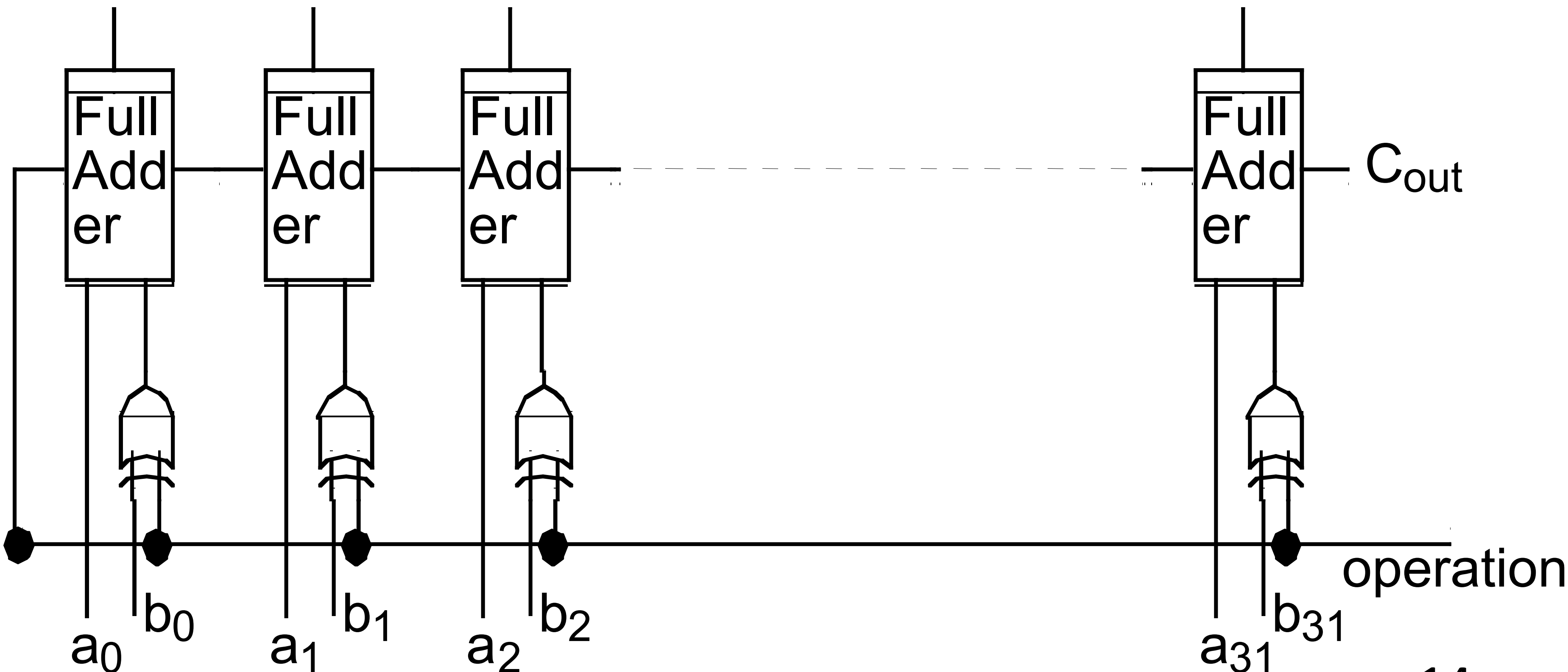
Ripple-carry Subtractor

- A – B = A + (-B) => invert B and set c_{in} to 1



Combined Ripple-carry Adder/Subtractor

- Control = 1 => subtract
- XOR B with control and set c_{in0} to control

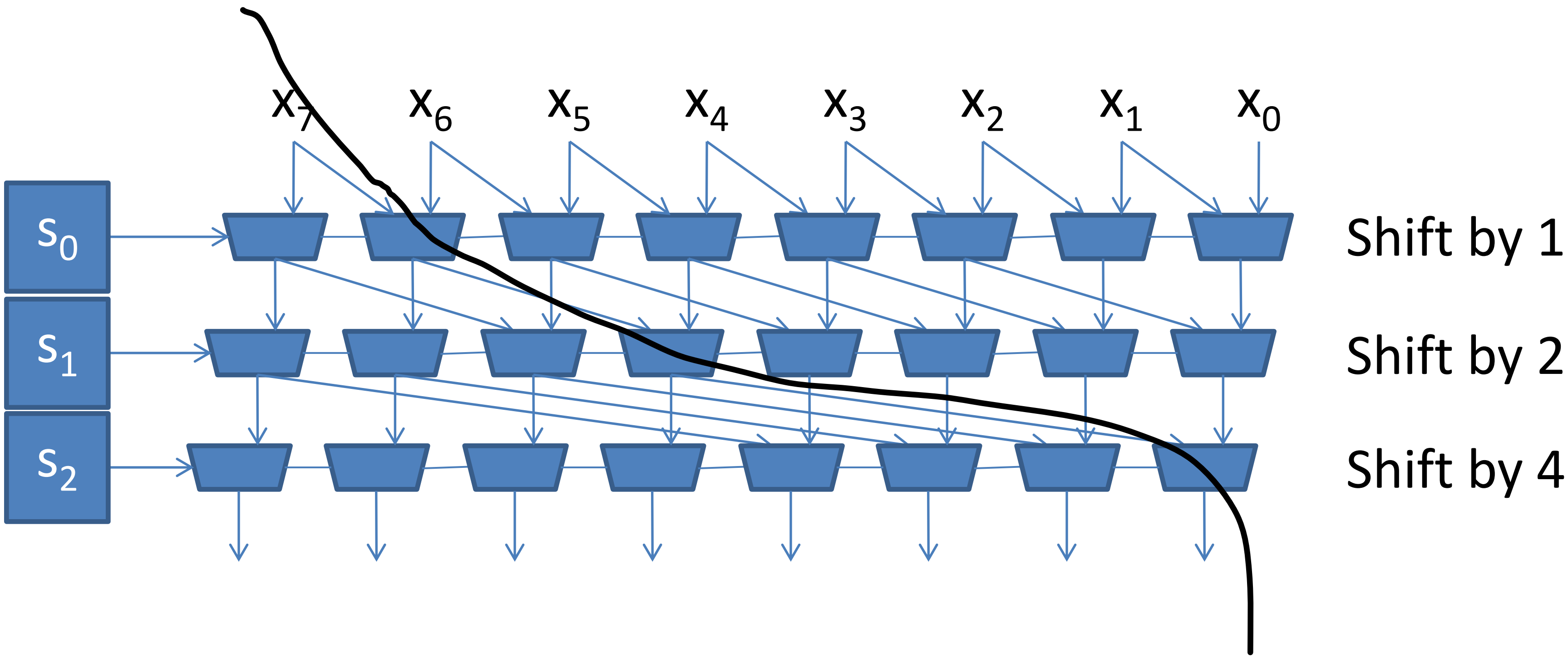


Logical Operations

- Bitwise AND, OR, XOR, NOR
  - Implement w/ 32 gates in parallel

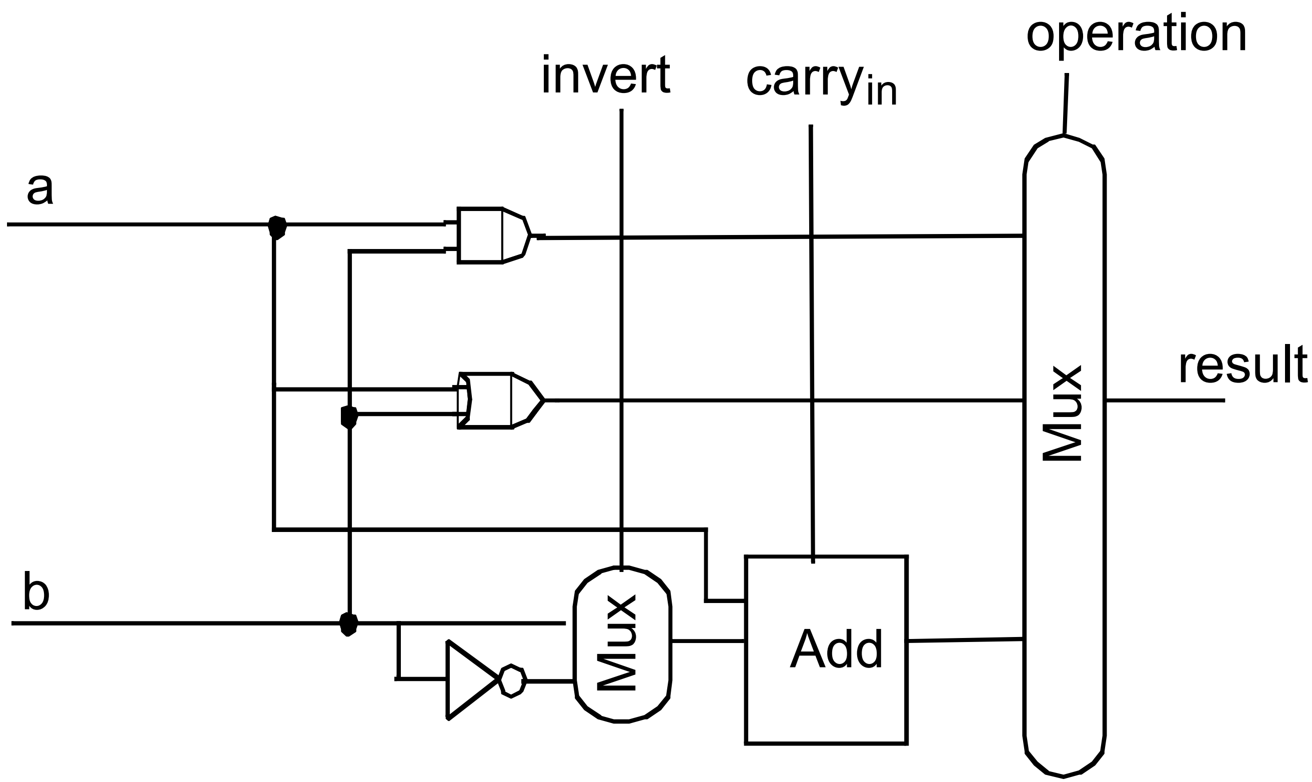
- Shifts and rotates
  - rol => rotate left (MSB->LSB)
  - ror => rotate right (LSB->MSB)
  - sll -> shift left logical (0->LSB)
  - srl -> shift right logical (0->LSB)
  - sra -> shift right arithmetic (old MSB->new MSB)

Shifter



- Right shift logic shown: missing inputs are 0
  - Left shift logic similar
- Rotate: wraparound instead of 0 inputs

All Together



Overflow

- With n bits only 2^n combinations
  - Unsigned range $[0, 2^n-1]$
  - 2's complement range $[-2^{n-1}, 2^{n-1}-1]$
- Unsigned Add

  5 + 6 > 7: 101 + 110 => 1011

  f(3:0) = a(2:0) + b(2:0) => overflow = f(3)

  Carryout from MSB

Overflow

- More involved for 2's complement

  -1 + -1 = -2:

  111 + 111 = 1110

  110 = -2 is correct
- Can't just use carry-out to signal overflow

Addition Overflow

- When is overflow NOT possible?

  $(p1, p2) > 0$ and $(n1, n2) < 0$

  $p1 + p2$

  $p1 + n1$ not possible

  $n1 + p2$ not possible

  $n1 + n2$
- Just checking signs of inputs is not sufficient

Addition Overflow

- 2 + 3 = 5 > 4: 010 + 011 = 101 =? –3 < 0
  - Sum of two positive numbers should not be negative
    - Conclude: overflow
- -1 + -4: 111 + 100 = 011 > 0
  - Sum of two negative numbers should not be positive
    - Conclude: overflow

Overflow = f(2) * ~(a2)*~(b2) + ~f(2) * a(2) * b(2)

Subtraction Overflow

- No overflow on a-b if signs are the same
- Neg – pos => neg ;; overflow otherwise
- Pos – neg => pos ;; overflow otherwise

Overflow = f(2) * ~(a2)*(b2) + ~f(2) * a(2) * ~b(2)

What to do on Overflow?

- Ignore ! (C language semantics)
  - What about Java? (try/catch?)
- Flag – condition code
- Sticky flag – e.g. for floating point
  - Otherwise gets in the way of fast hardware
- Trap – possibly maskable
  - MIPS has e.g. add that traps, addu that does not
  - Useful for extended precision in software

Zero and Negative

- Zero = ~[f(2) + f(1) + f(0)]
- Negative = f(2) (sign bit)

Zero and Negative

- May also want correct answer even on overflow
- Negative = (a < b) = (a – b) < 0 even if overflow
- E.g. is –4 < 2?

  100 – 010 = 1010 (-4 – 2 = -6): Overflow!

- Work it out: negative = f(2) XOR overflow

Summary

- Binary representations, signed/unsigned
- Arithmetic
  - Full adder, ripple-carry, adder/subtractor
  - Overflow, negative
- Logical
  - Shift, and, or
- Next: high-performance adders

– Later: multiply/divide/FP