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...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?

Possible Representations

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

- **Issues: balance, number of zeros, ease of operations**
- **Which one is best? Why?**

MIPS

- **32 bit signed numbers:**

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ — *maxint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$ — *minint*
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

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 gets converted to 32 bits for arithmetic**
  - **copy the most significant bit (the sign bit) into the other bits**

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

  - **"sign extension" (lbu vs. lb)**

Addition & Subtraction

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

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

- **Two's complement operations easy**
  - **subtraction using addition of negative numbers**

```
  0111
+ 1010
```

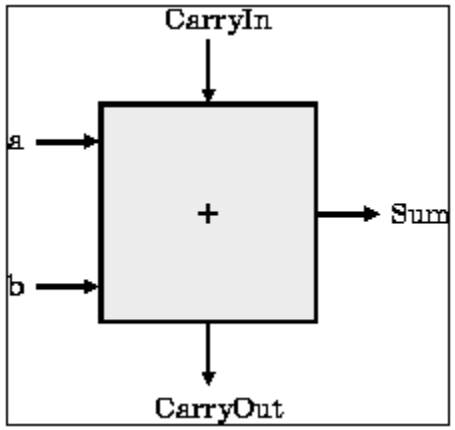
- **Overflow (result too large for finite computer word):**
  - **e.g., adding two n-bit numbers does not yield an n-bit number**

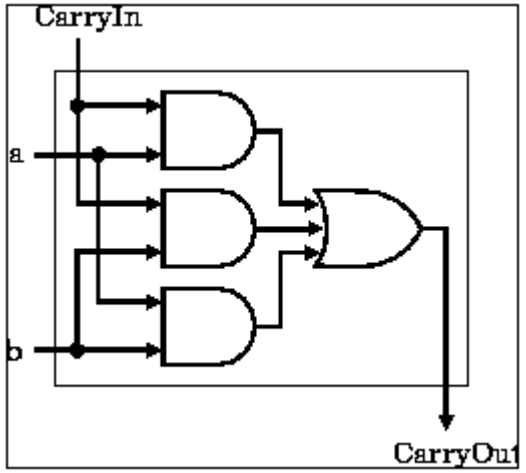
```
  0111
+ 0001
  1000
```

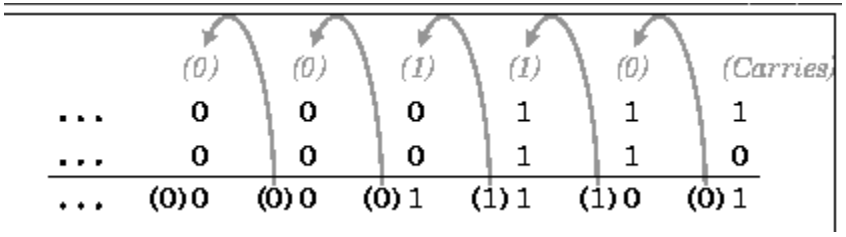
note that overflow term is somewhat misleading, it does not mean a carry "overflowed"

One-Bit Adder

- **Takes three input bits and generates two output bits**
- **Multiple bits can be cascaded**







Adder Boolean Algebra

A	B	CI	CO	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

C = A.B + A.CI+ B.CI

S = A.B.CI + A'.B'.CI+A'.B.CI'+A.B'.CI'

Detecting Overflow

- No overflow when adding a positive and a 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 ?

Effects of Overflow

- An exception (interrupt) occurs
  - Control jumps to predefined address for exception
  - Interrupted address is saved for possible resumption
- Details based on software system / language
  - example: flight control vs. homework assignment
- Don't always want to detect overflow
  — new MIPS instructions: `addu, addiu, subu`

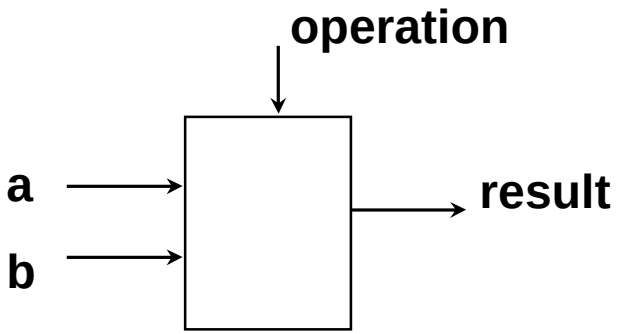
  note: `addiu` *still sign-extends!*
  note: `sltu, sltiu` *for unsigned comparisons*

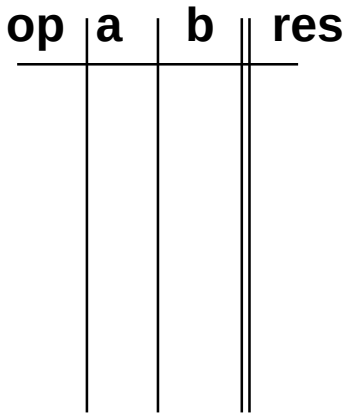
Real Design

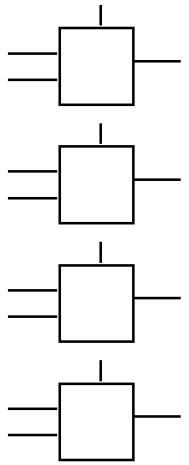
A	B	C	D	E	F	
0	0	0	0	0	0	
0	0	1	1	0	0	
0	1	0	1	0	0	D = A + B + C
0	1	1	1	1	0	
1	0	0	1	0	0	E = A'.B.C + A.B'.C + A.B.C'
1	0	1	1	1	0	
1	1	0	1	1	0	F = A.B.C
1	1	1	1	0	1	

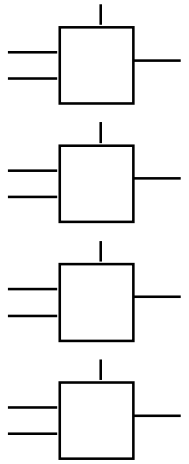
An ALU (arithmetic logic unit)

- Let's build an ALU to support the `andi` and `ori` instructions
  - we'll just build a 1 bit ALU, and use 32 of them





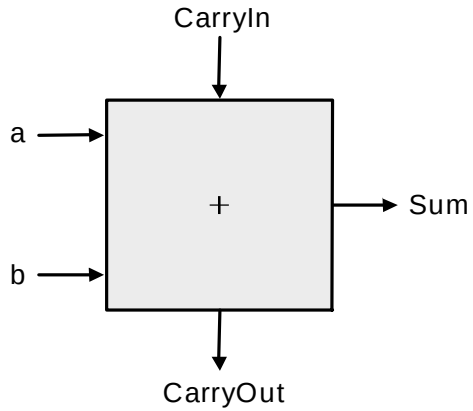




- Possible Implementation (sum-of-products):

Different Implementations

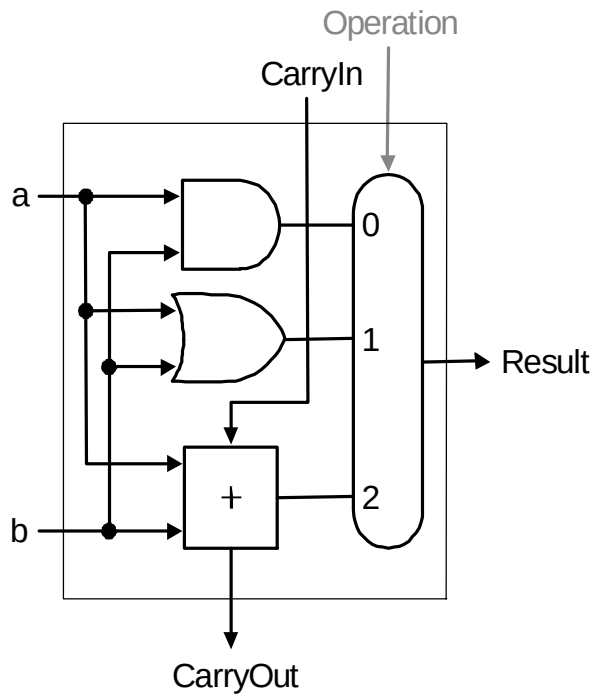
- Not easy to decide the “best” way to build something
  - Don't want too many inputs to a single gate
  - Don’t want to have to go through too many gates
  - for our purposes, ease of comprehension is important
- Let's look at a 1-bit ALU for addition:

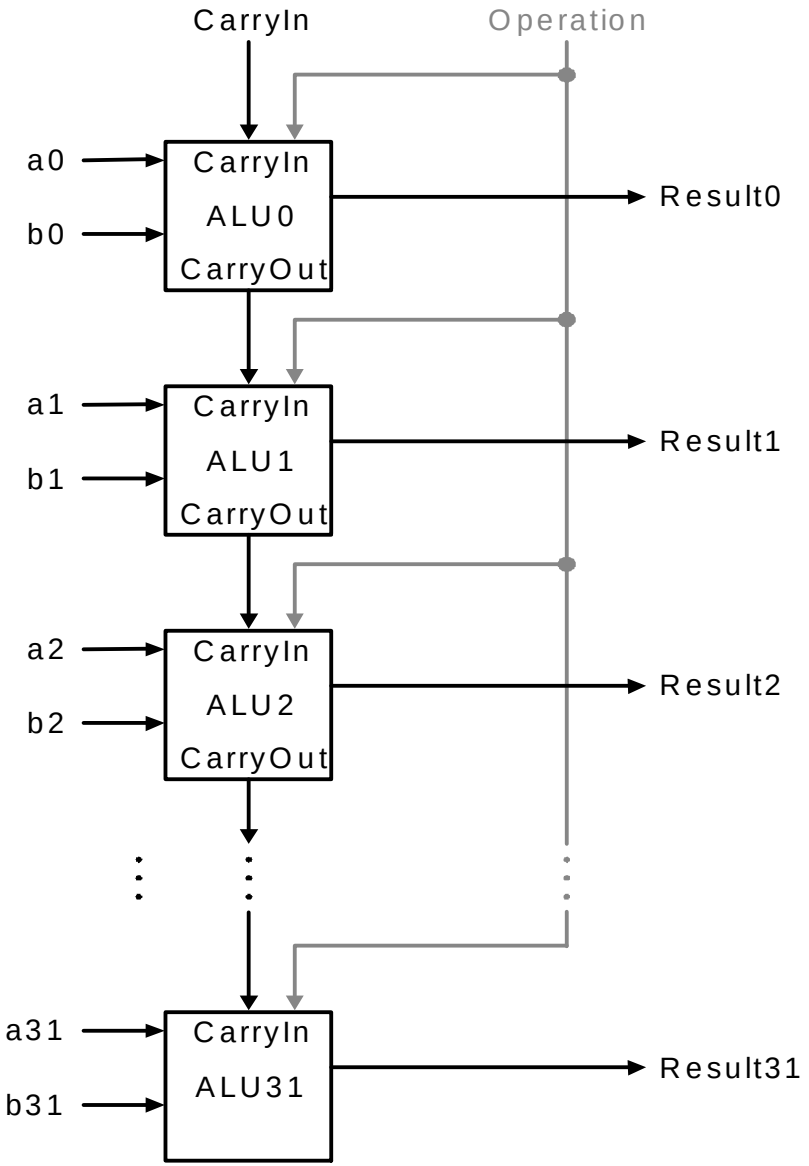


$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- How could we build a 1-bit ALU for add, and, and or?
- How could we build a 32-bit ALU?

Building a 32 bit ALU

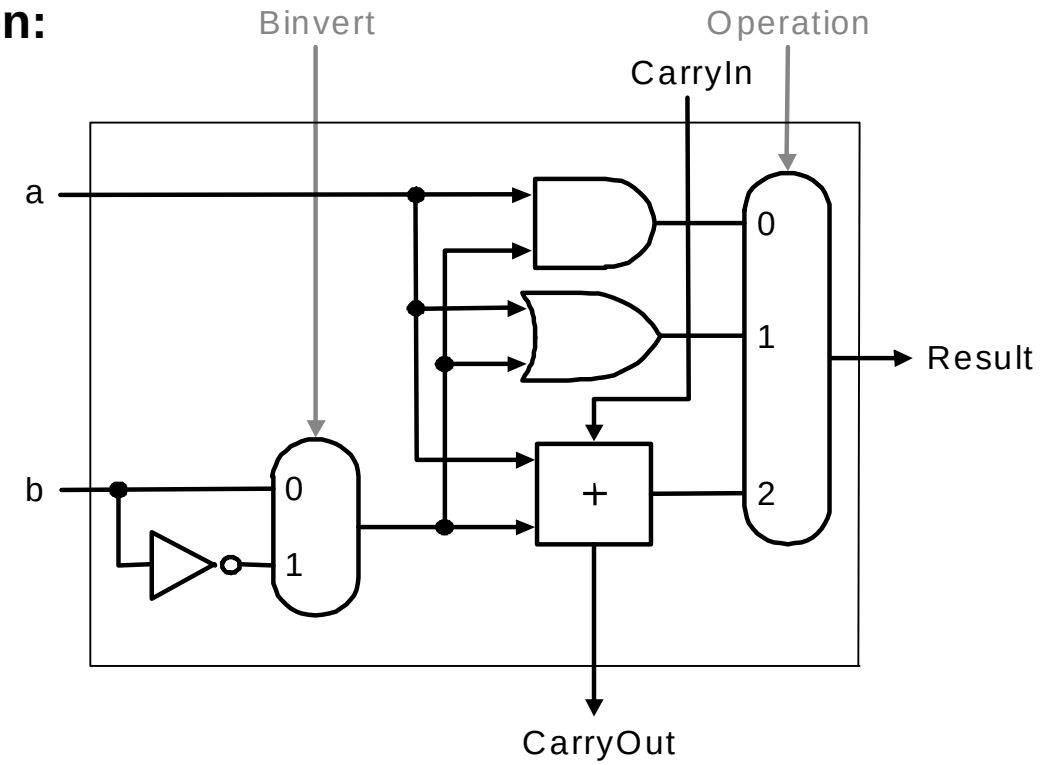




What about subtraction (a – b) ?

- **Two's complement approach: just negate b and add.**
- **How do we negate?**

- **A very clever solution:**

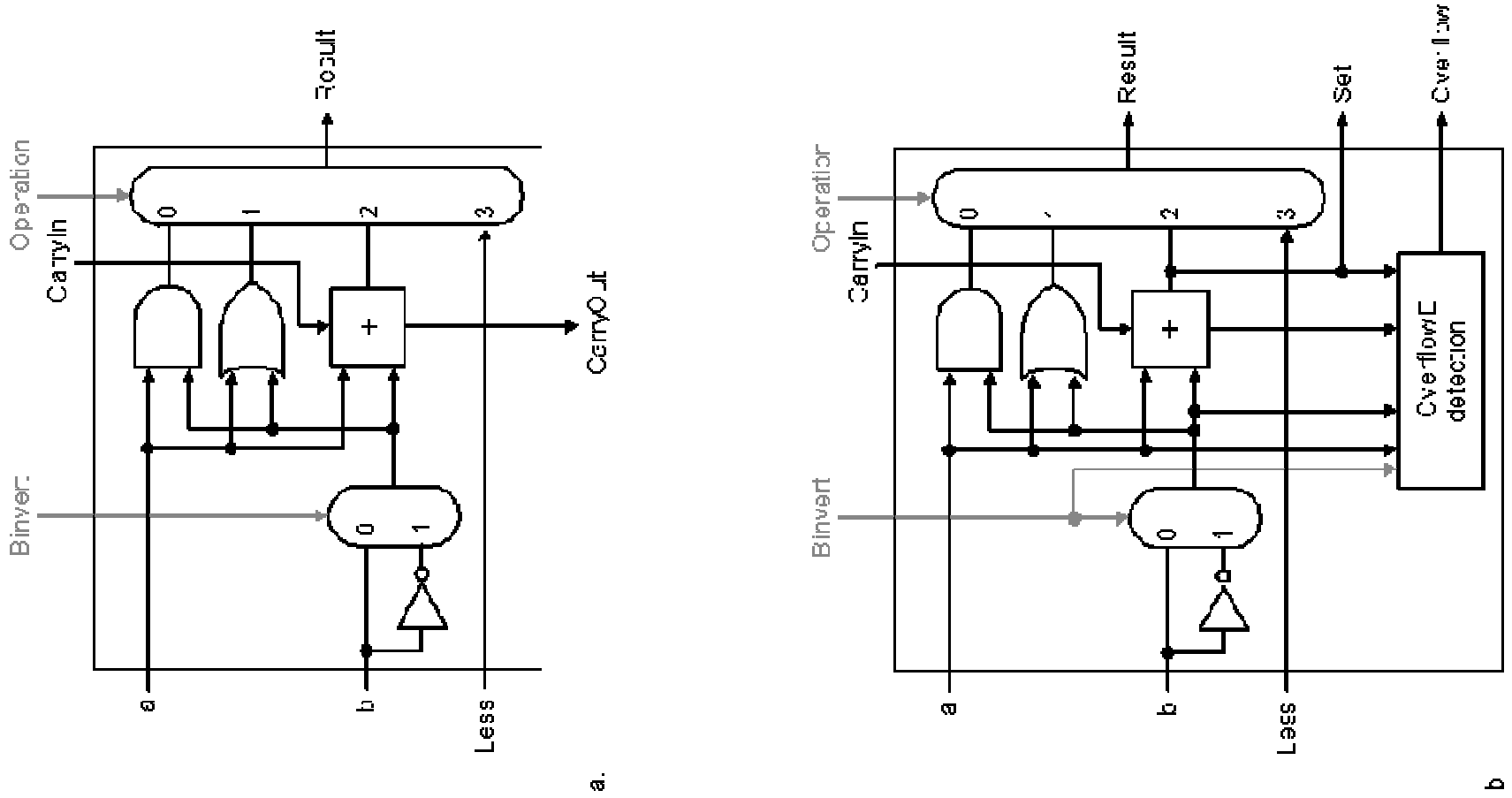


Tailoring the ALU to the MIPS

- **Need to support the set-on-less-than instruction (slt)**
  - **remember: slt is an arithmetic instruction**
  - **produces a 1 if rs < rt and 0 otherwise**
  - **use subtraction: (a-b) < 0 implies a < b**
- **Need to support test for equality (beq $t5, $t6, $t7)**
  - **use subtraction: (a-b) = 0 implies a = b**

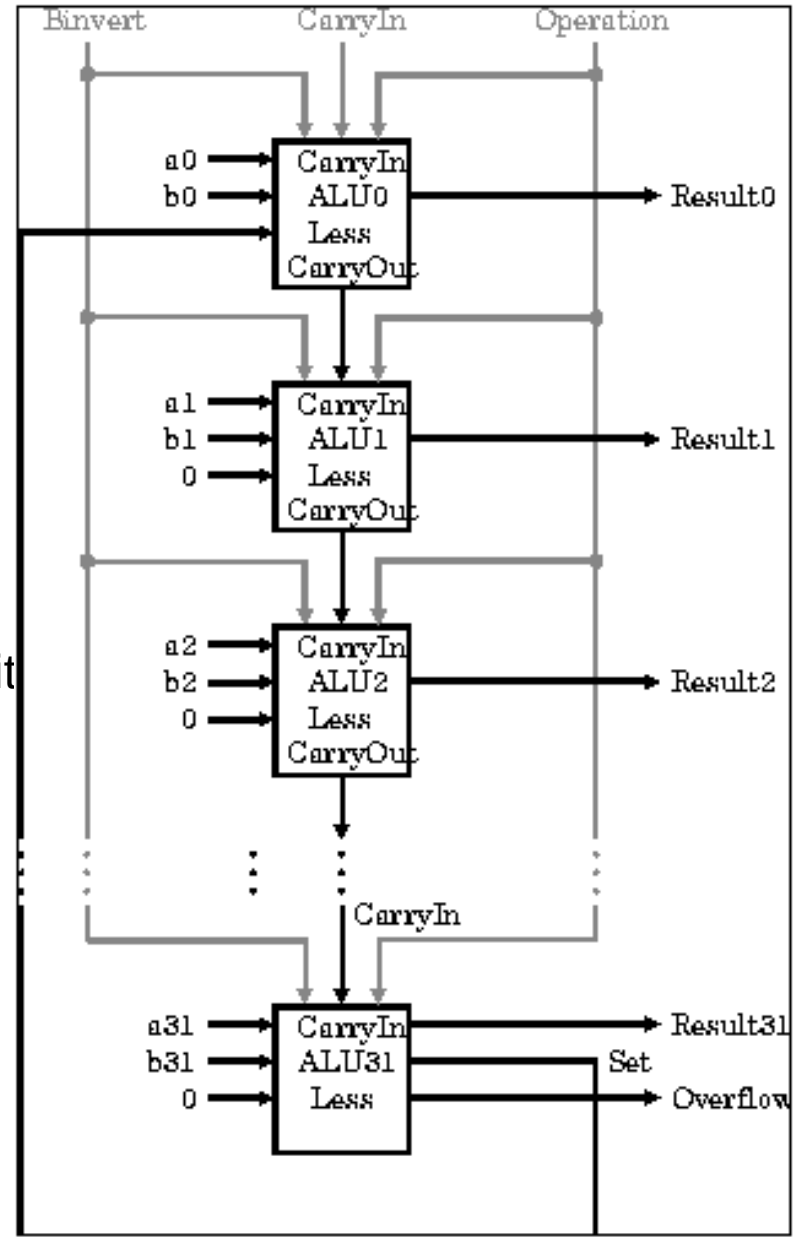
Supporting slt

- **Can we figure out the idea?**



A 32-bit ALU

- A Ripple carry ALU
- Two bits decide operation
  - Add/Sub
  - AND
  - OR
  - LESS
- 1 bit decide add/sub operation
- A carry in bit
- Bit 31 generates overflow and set bit

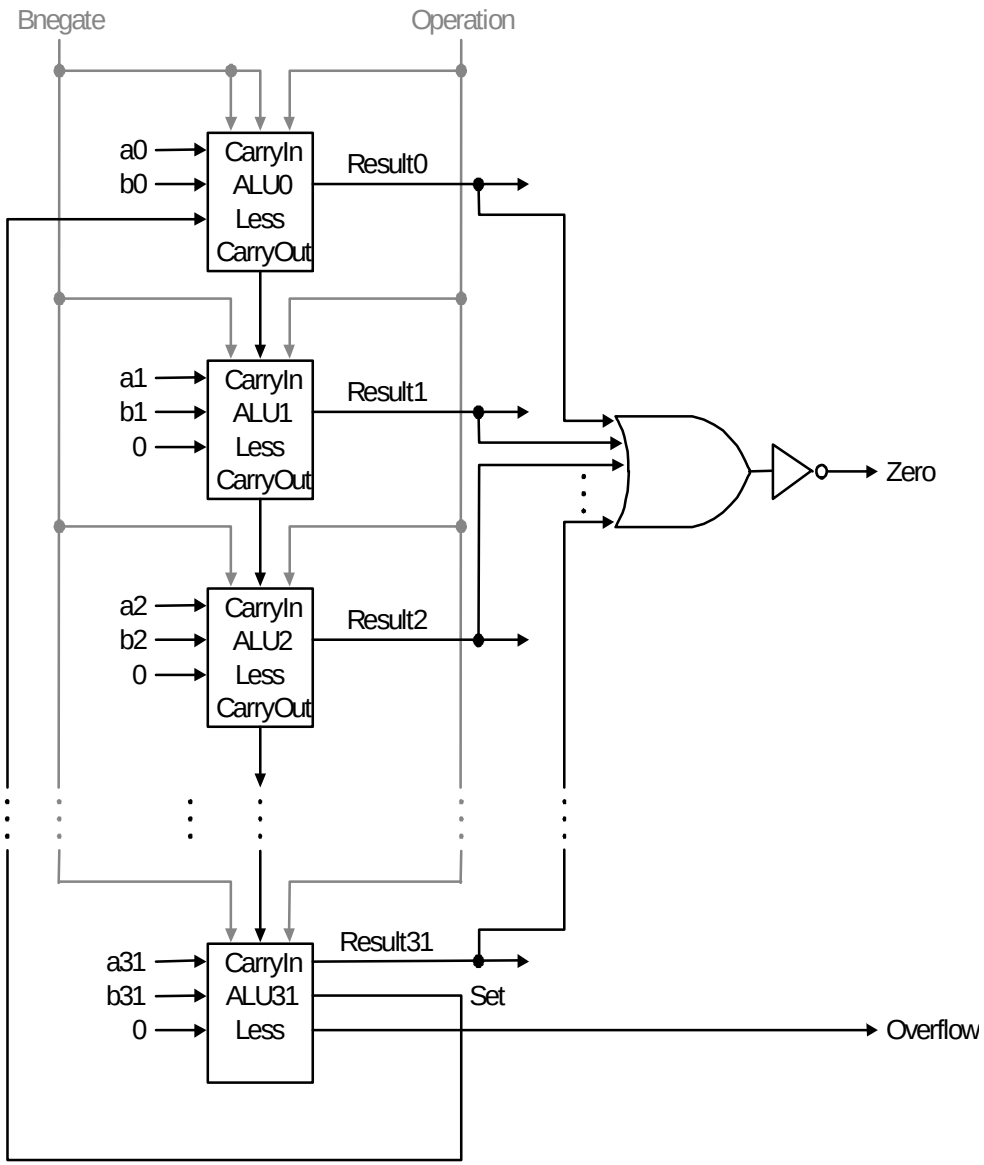


Test for equality

- **Notice control lines:**

```
000 = and
001 = or
010 = add
110 = subtract
111 = slt
```

•Note: zero is a 1 when the result is zero!



Problem: ripple carry adder is slow

- Is a 32-bit ALU as fast as a 1-bit ALU?
- Is there more than one way to do addition?
  - two extremes: ripple carry and sum-of-products

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

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

Not feasible! Why?

Carry-look-ahead adder

- 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?

$c_1 = g_0 + p_0c_0$

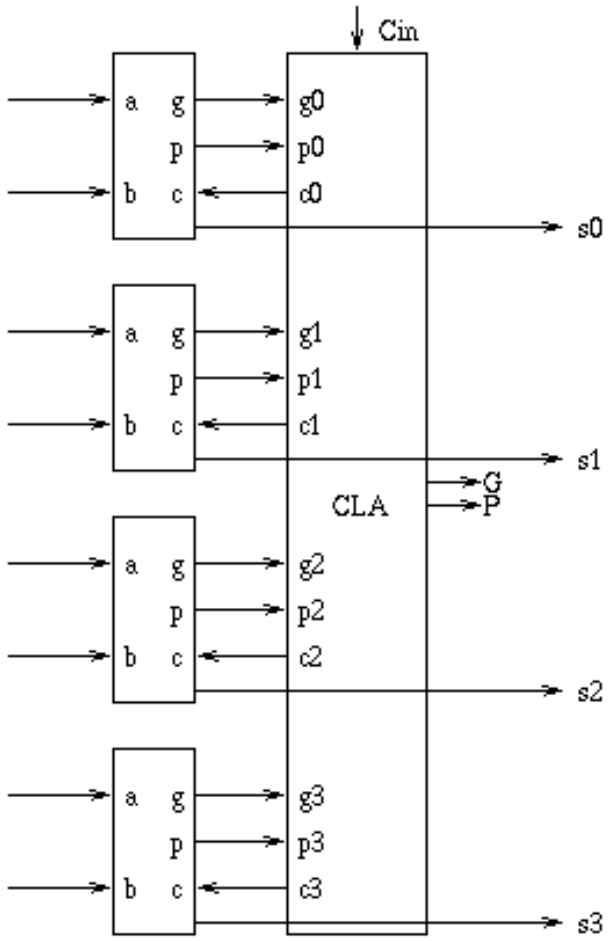
$c_2 = g_1 + p_1c_1$ $\quad c_2 = g_1 + p_1g_0 + p_1p_0c_0$

$c_3 = g_2 + p_2c_2$ $\quad c_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$

$c_4 = g_3 + p_3c_3$ $\quad c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$

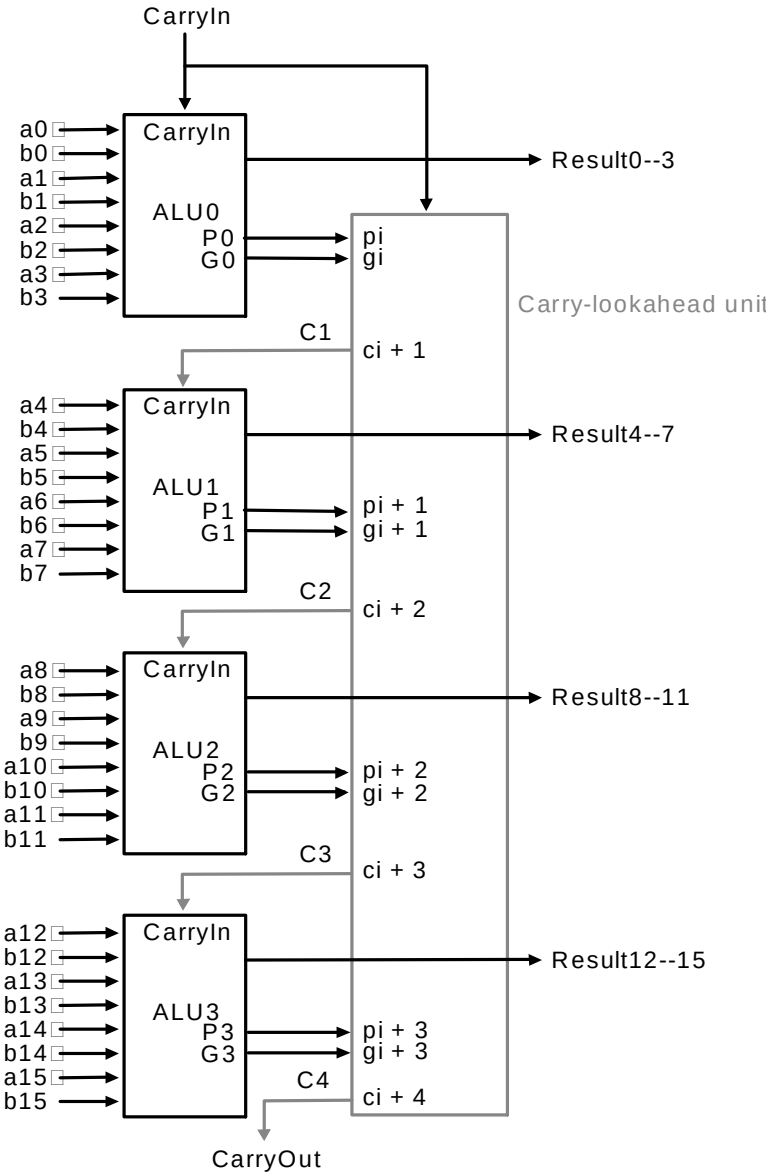
Feasible! Why?

A 4-bit carry look-ahead adder



- **Generate g and p term for each bit**
- **Use g's, p's and carry in to generate all C's**
- **Also use them to generate block G and P**
- **CLA principle can be used recursively**

Use principle to build bigger adders



- **A 16 bit adder uses four 4-bit adders**
- **It takes block g and p terms and cin to generate block carry bits out**
- **Block carries are used to generate bit carries**
  - **could use ripple carry of 4-bit CLA adders**
  - **Better: use the CLA principle again!**

Delays in carry look-ahead adders

- **4-Bit case**
  - **Generation of g and p: 1 gate delay**
  - **Generation of carries (and G and P): 2 more gate delay**
  - **Generation of sum: 1 more gate delay**
- **16-Bit case**
  - **Generation of g and p: 1 gate delay**
  - **Generation of block G and P: 2 more gate delay**
  - **Generation of block carries: 2 more gate delay**
  - **Generation of bit carries: 2 more gate delay**
  - **Generation of sum: 1 more gate delay**
- **64-Bit case**
  - **12 gate delays**

Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Let's look at 3 versions based on grade school algorithm

```
 01010010   (multiplicand)
x01101101   (multiplier)
```

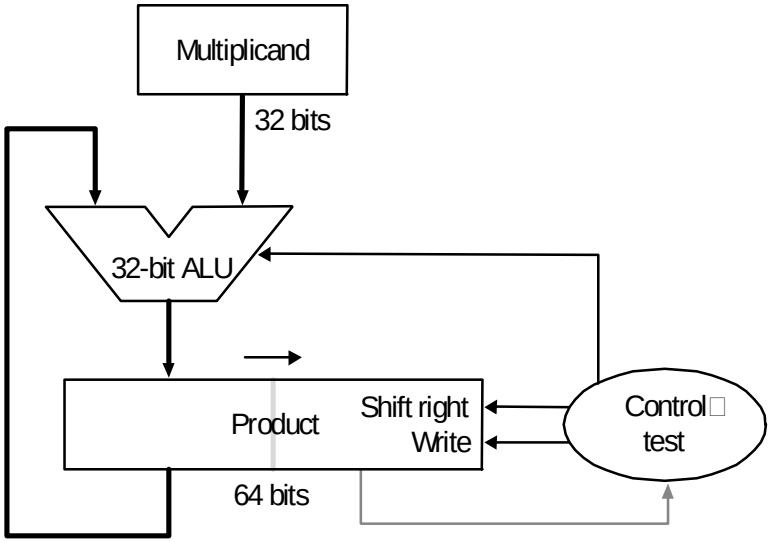
- Negative numbers: convert and multiply
- Use other better techniques like Booth's encoding

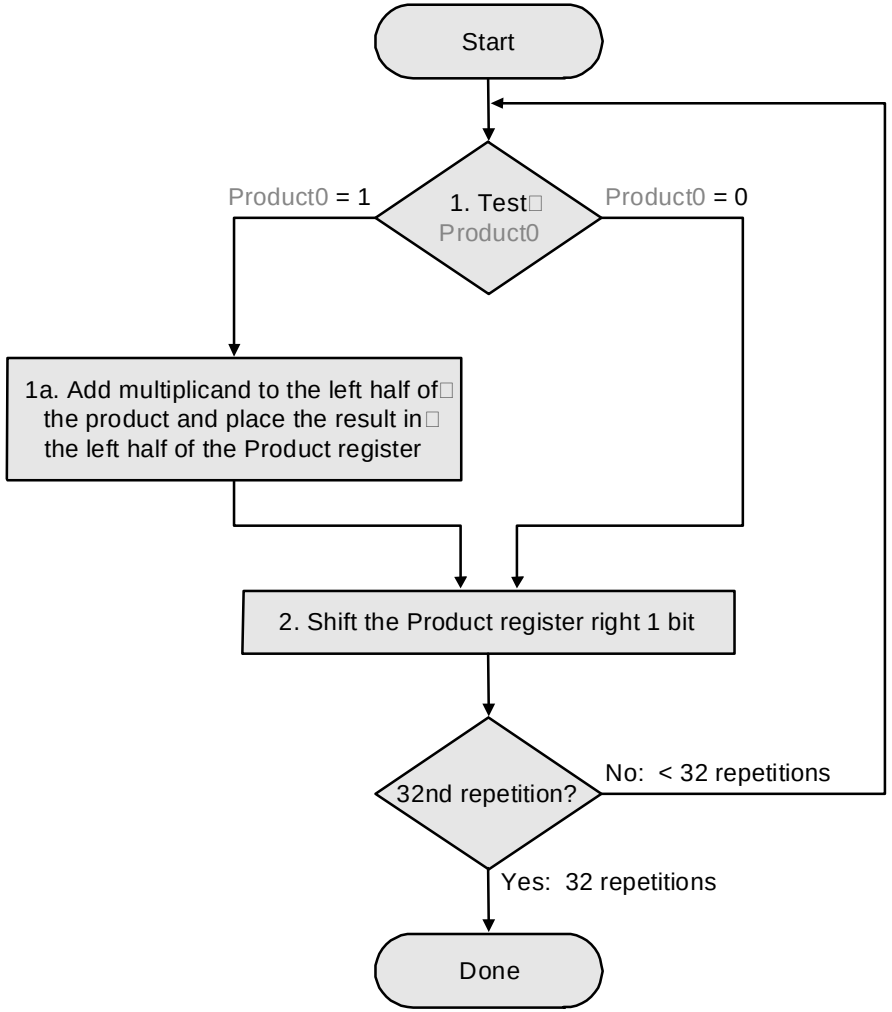
Multiplication

```
  01010010  (multiplicand)
x01101101  (multiplier)
--------
  00000000
  01010010   x1
--------
   01010010
  000000000   x0
--------
   001010010
  0101001000   x1
--------
   0110011010
  01010010000   x1
--------
   1000010 1010
  000000000000   x0
--------
   0100001 01010
  0101001000000   x1
--------
   0111001101010
  0101001000000   x1
--------
   1000101101010
  000000000000000   x0
--------
 0100010111101010
```

```
  01010010  (multiplicand)
x01101101  (multiplier)
--------
  00000000
  01010010   x1
--------
   01010010
  000000000   x0
--------
   001010010
  0101001000   x1
--------
   0110011010
  01010010000   x1
--------
   1000010 1010
  000000000000   x0
--------
   0100001 01010
  0101001000000   x1
--------
   0111001101010
  0101001000000   x1
--------
   1000101101010
  000000000000000   x0
--------
 0100010111101010
```

Multiplication: Implementation





Multiplication Example

Itera-tion	multi-plicand	Orignal algorithm	
		Step	Product
0	0010	Initial values	0000 0110
1	0010	1:0 ⇒ no operation	0000 0110
	0010	2: Shift right Product	0000 0011
2	0010	1a:1⇒ prod = Prod + Mcand	0010 0011
	0010	2: Shift right Product	0001 0001
3	0010	1a:1⇒ prod = Prod + Mcand	0011 0001
	0010	2: Shift right Product	0001 1000
4	0010	1:0 ⇒ no operation	0001 1000
	0010	2: Shift right Product	0000 1100

Signed Multiplication

- **Let Multiplier be Q[n-1:0], multiplicand be M[n-1:0]**
- **Let F = 0 (shift flag)**
- **Let result A[n-1:0] = 0….00**
- **For n-1 steps do**
  - **A[n-1:0] = A[n-1:0] + M[n-1:0] x Q[0] /* add partial product */**
  - **F<= F .or. (M[n-1] .and. Q[0]) /* determine shift bit */**
  - **Shift A and Q with F, i.e.,**
  - **A[n-2:0] = A[n-1:1]; A[n-1]=F; Q[n-1]=A[0]; Q[n-2:0]=Q[n-1:1]**
- **Do the correction step**
  - **A[n-1:0] = A[n-1:0] - M[n-1:0] x Q[0] /* subtract partial product */**
  - **Shift A and Q while retaining A[n-1]**
  - **This works in all cases excepts when both operands are 10..00**

Booth's Encoding

- Numbers can be represented using three symbols, 1, 0, and -1
- Let us consider -1 in 8 bits
  - One representation is 1 1 1 1 1 1 1 1
  - Another possible one 0 0 0 0 0 0 0 -1
- Another example +14
  - One representation is 0 0 0 0 1 1 1 0
  - Another possible one 0 0 0 1 0 0 -1 0
- We do not explicitly store the sequence
- Look for transition from previous bit to next bit
  - 0 to 0 is 0; 0 to 1 is -1; 1 to 1 is 0; and 1 to 0 is 1
- Multiplication by 1, 0, and -1 can be easily done
- Add all partial results to get the final answer

Using Booth's Encoding for Multiplication

- Convert a binary string in Booth's encoded string
- Multiply by two bits at a time
- For n bit by n-bit multiplication, n/2 partial product
- Partial products are signed and obtained by multiplying the multiplicand by 0, +1, -1, +2, and -2 (all achieved by shift)
- Add partial products to obtain the final result
- Example, multiply 0111 (+7) by 1010 (-6)
- Booths encoding of 1010 is -1 +1 -1 0
- With 2-bit groupings, multiplication needs to be carried by -1 and -2
-

```
1 1 1 1 0 0 1 0   (multiplication by -2)
1 1 1 0 0 1 0 0   (multiplication by -1 and shift by 2 positions)
```

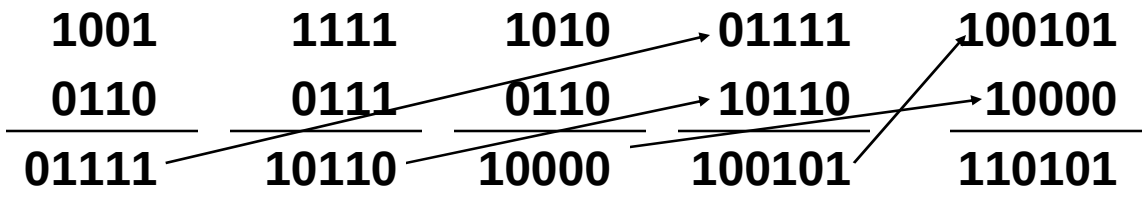
- Add the two partial products to get 11010110 (-42) as result

Booth's algorithm (Neg. multiplier)

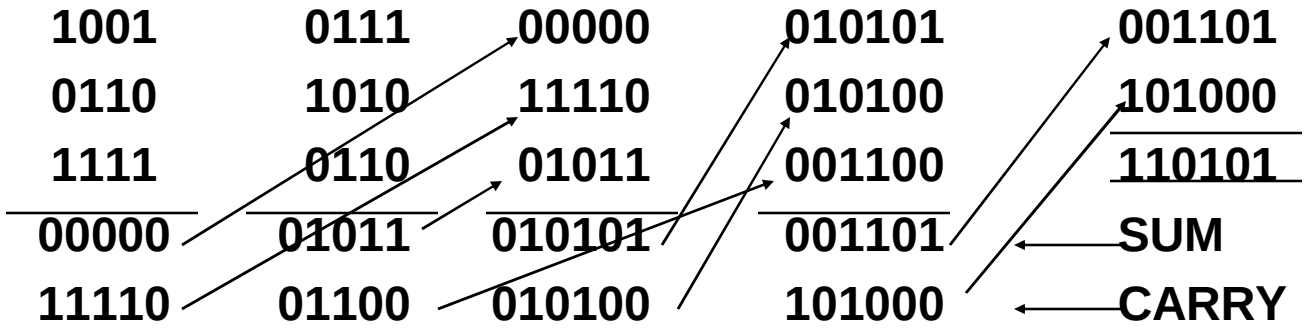
Itera-tion	multi-plicand	Booth's algorithm	
		Step	Product
0	0010	Initial values	0000 1101 0
1	0010	1c: 10⇒ prod = Prod - Mcand	1110 1101 0
	0010	2: Shift right Product	1111 0110 1
2	0010	1b: 01⇒ prod = Prod + Mcand	0001 0110 1
	0010	2: Shift right Product	0000 1011 0
3	0010	1c: 10⇒ prod = Prod - Mcand	1110 1011 0
	0010	2: Shift right Product	1111 0101 1
4	0010	1d: 11 ⇒ no operation	1111 0101 1
	0010	2: Shift right Product	1111 1010 1

Carry-Save Addition

- Consider adding six set of numbers (4 bits each in the example)
- The numbers are 1001, 0110, 1111, 0111, 1010, 0110 (all positive)
- One way is to add them pair wise, getting three results, and then adding them again



- Other method is add them three at a time by saving carry



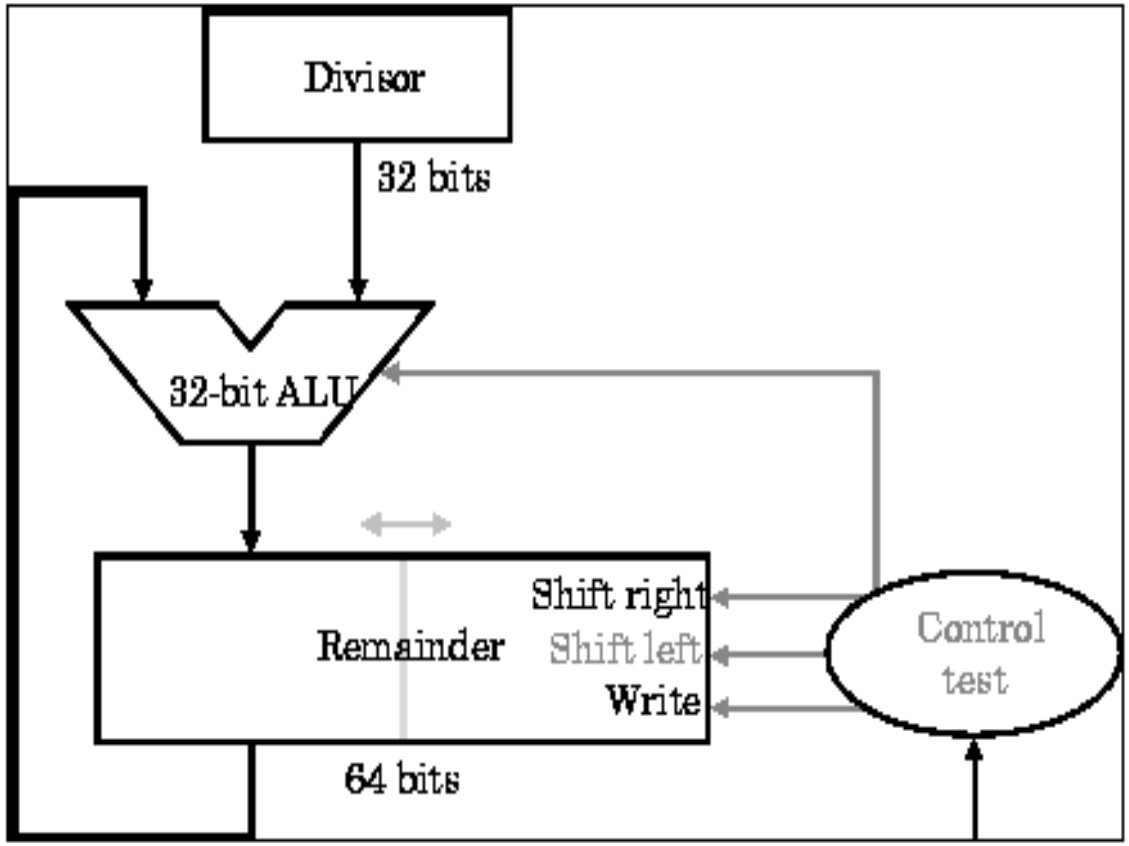
Division

- Even more complicated
  - can be accomplished via shifting and addition/subtraction
- More time and more area
- We will look at 3 versions based on grade school algorithm

```
 0011 | 0010 0010   (Dividend)
```

- Negative numbers: Even more difficult
- There are better techniques, we won't look at them

Division


Divisor
32 bits
32-bit ALU
Remainder
Shift right
Shift left
Write
Control
test
64 bits

Restoring Division

Iteration	Divisor	Divide algorithm	
		Step	Remainder
0	0010	Initial values	0000 0111
	0010	Shift Rem left 1	0000 1110
1	0010	2: Rem = Rem - Div	1110 1110
	0010	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0001 1100
2	0010	2: Rem = Rem - Div	1111 1100
	0010	3b: Rem < 0 ⇒ + Div, sll R, R0 = 0	0011 1000
3	0010	2: Rem = Rem - Div	0001 1000
	0010	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0011 0001
4	0010	2: Rem = Rem - Div	0001 0001
	0010	3a: Rem ≥ 0 ⇒ sll R, R0 = 1	0010 0011
Done	0010	shift left half of Rem right 1	0001 0011

Non-Restoring Division

Iteration	Divisor	Divide algorithm	
		Step	Remainder
0	0010	Initial values	0000 1110
1	0010	1: Rem = Rem - Div	1110 1110
	0010	2b: Rem < 0 ⇒,sll R, R0 = 0	1101 1100
	0010	3b: Rem = Rem + Div	1111 1100
2	0010	2b: Rem < 0 ⇒ sll R, R0 = 0	1111 1000
	0010	3b: Rem = Rem + Div	0001 1000
3	0010	2a: Rem > 0 ⇒ sll R, R0 = 1	0011 0001
	0010	3a: Rem = Rem - Div	0001 0001
4	0010	2a: Rem > 0 ⇒ sll R, R0 = 1	0010 0011
Done	0010	shift left half of Rem right 1	0001 0011