Chapter 3

Arithmetic for Computers

Tien-Fu Chen

Dept. of Computer Science

National Chiao Tung Univ.

Material source:
COD RISC-v slides

Numbers: Possible Representations

- Bits are just bits (no inherent meaning)
  — data meaning depends on interpretation of bits
-
-

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?

Two’s complement

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

Two's Complement Operations

- Negating a number: invert all bits and add 1

$$A - B = A + (-B) = A + \overline{B} + 1$$

remember: “negate” and “invert” are quite different!

- Converting n bit numbers into m bit numbers (m > n):
  - Convert 16 bit immediate 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"

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”

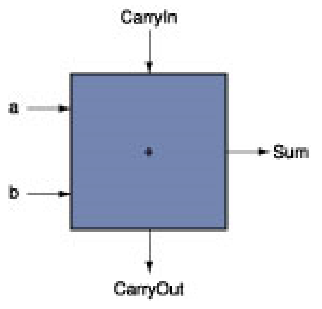
Detecting Overflow

- 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
- Detecting Overflow

	A	B	Result
A+B	+	+	-
A+B	-	-	+
A-B	+	-	-
A-B	-	+	+

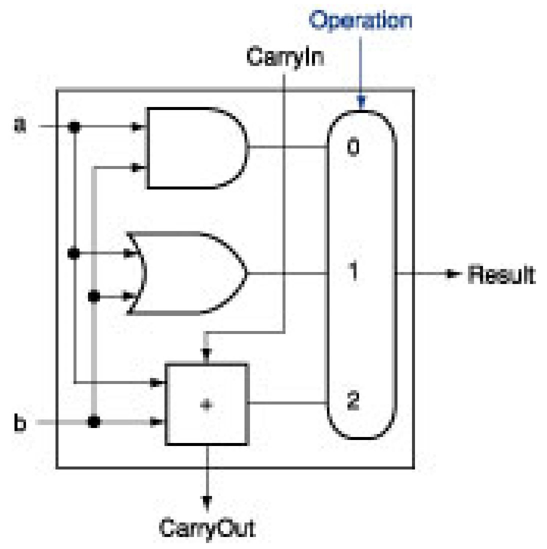
ALU (arithmetic logic unit) – **Ref B.25~**

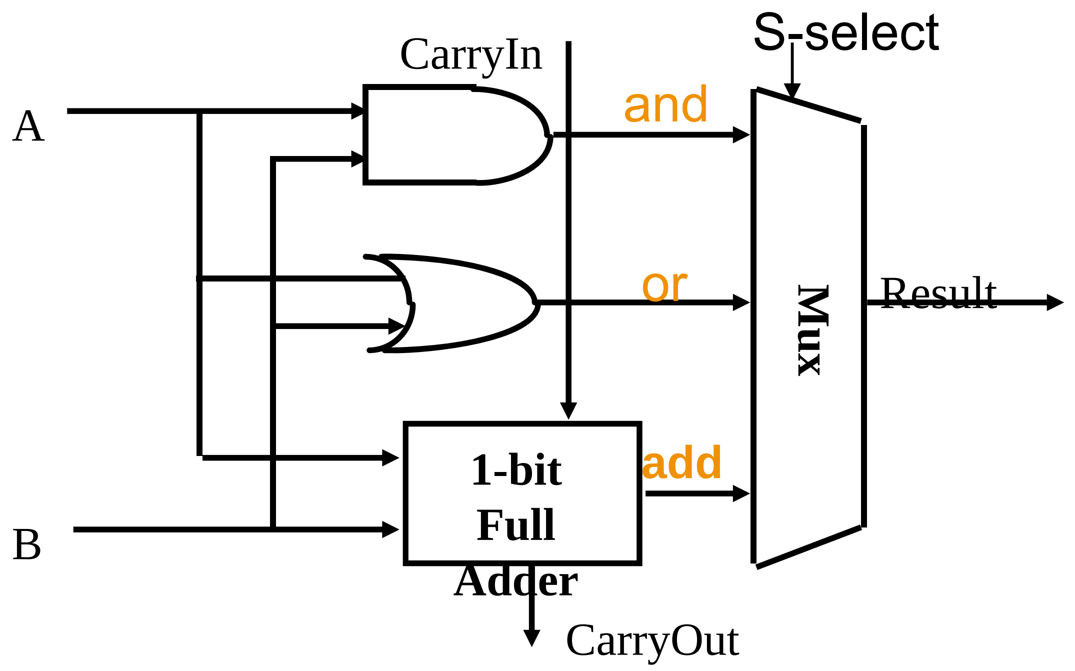
- 1-bit adder



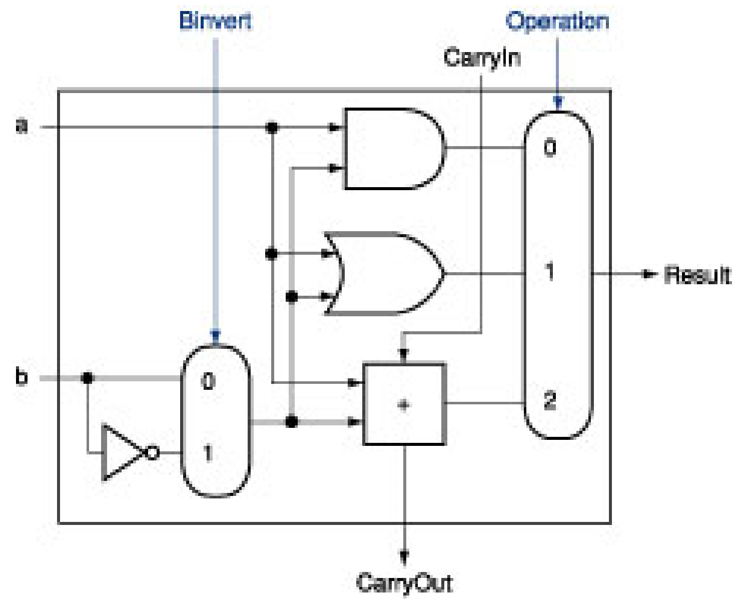
$$c_{out} = a\ b + a\ c_{in} + b\ c_{in}$$
$$sum = a \text{ xor } b \text{ xor } c_{in}$$

- 1-bit ALU





Combine add/sub in 32-bit ALU

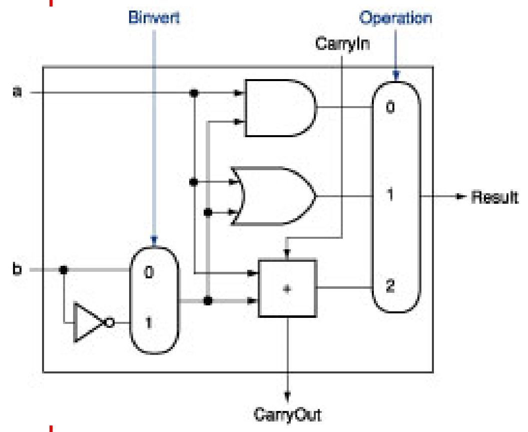


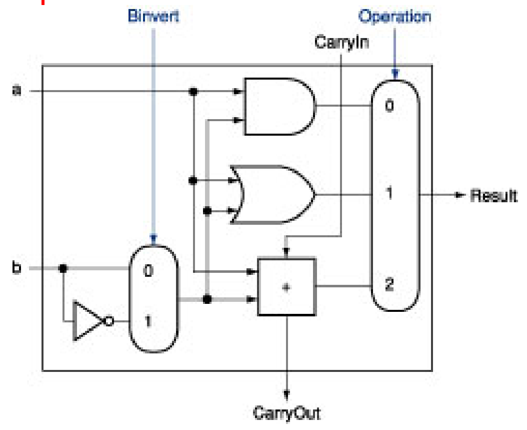
	Binvert	Carryin	Operation	
and				
or				
add				
sub				
slt				

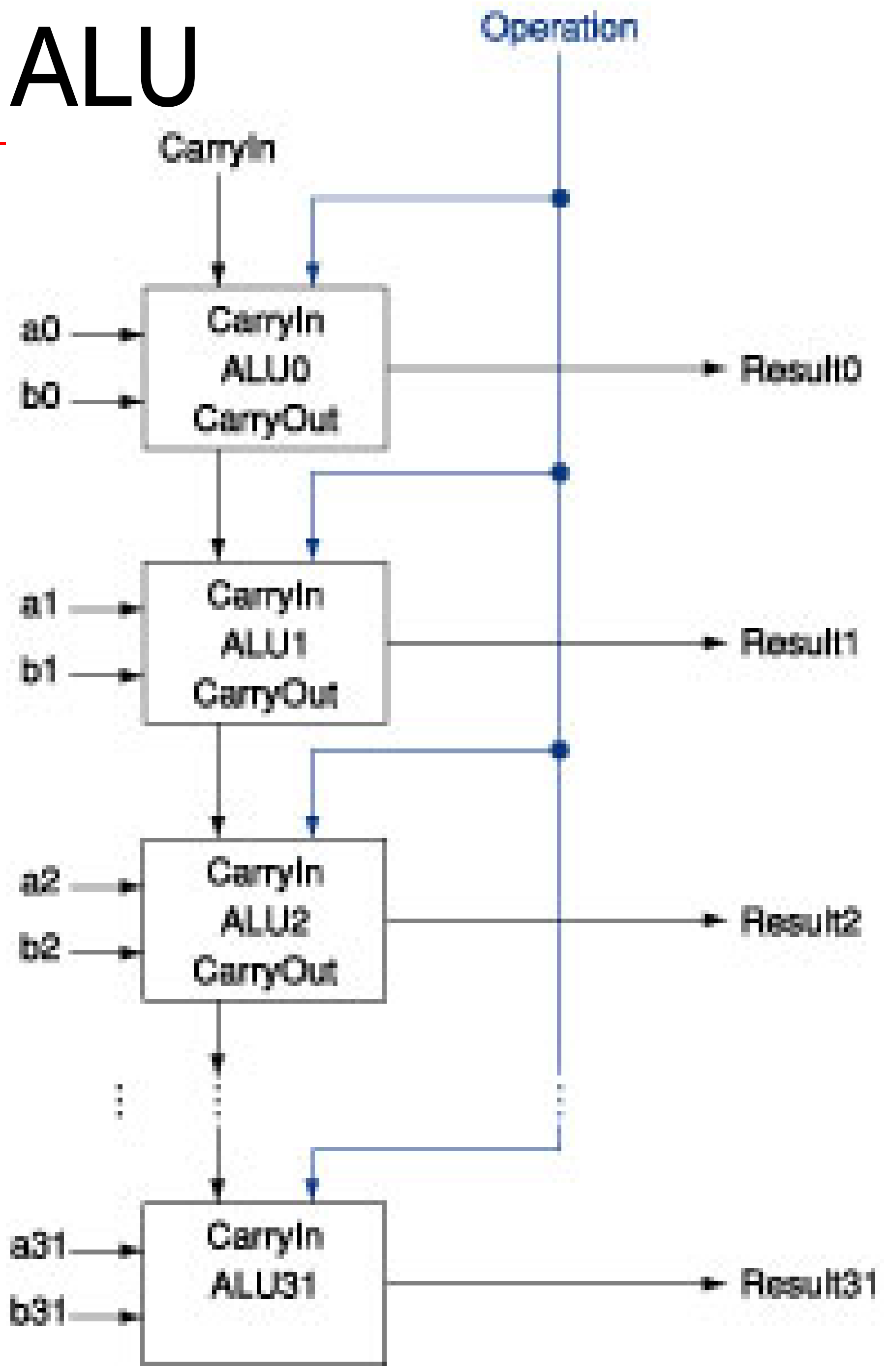
$$A - B = A + (-B) = A + \overline{B} + 1$$

Support add/sub in 32-bit ALU

$$A - B = A + (-B) = A + \overline{B} + 1$$





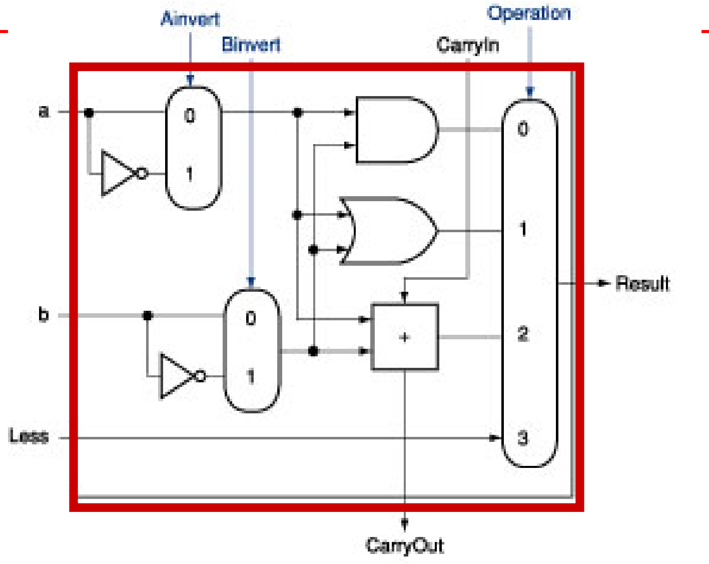


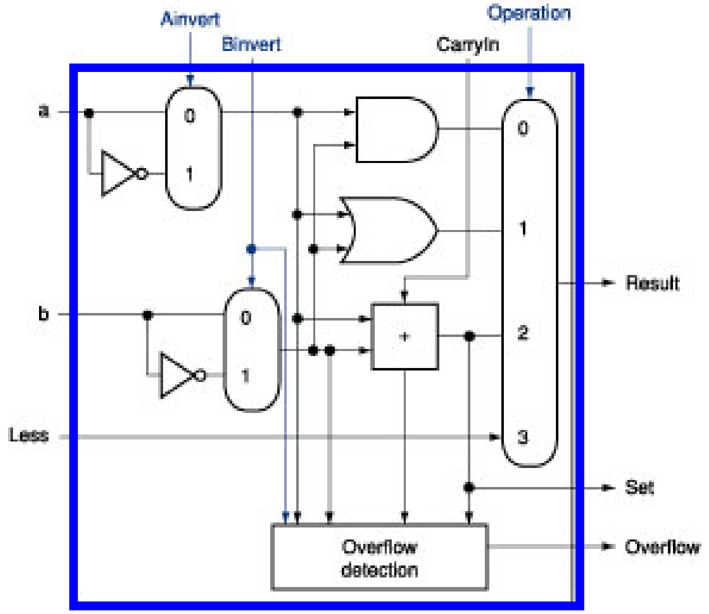
•Control lines
000 = and
001 = or
010 = add
110 = sub
111 = slt

Supporting less-than instruction

- set-on-less-than instruction (slt)

  (a-b) < 0 implies a < b
- (Top) A 1-bit ALU that performs AND, OR, and addition on a and b or b,
  - The top drawing includes a direct input that is connected to perform the set on less than operation
- (bottom) a 1-bit ALU for the most significant bit.
  - has a direct output from the adder for the less than comparison called Set.



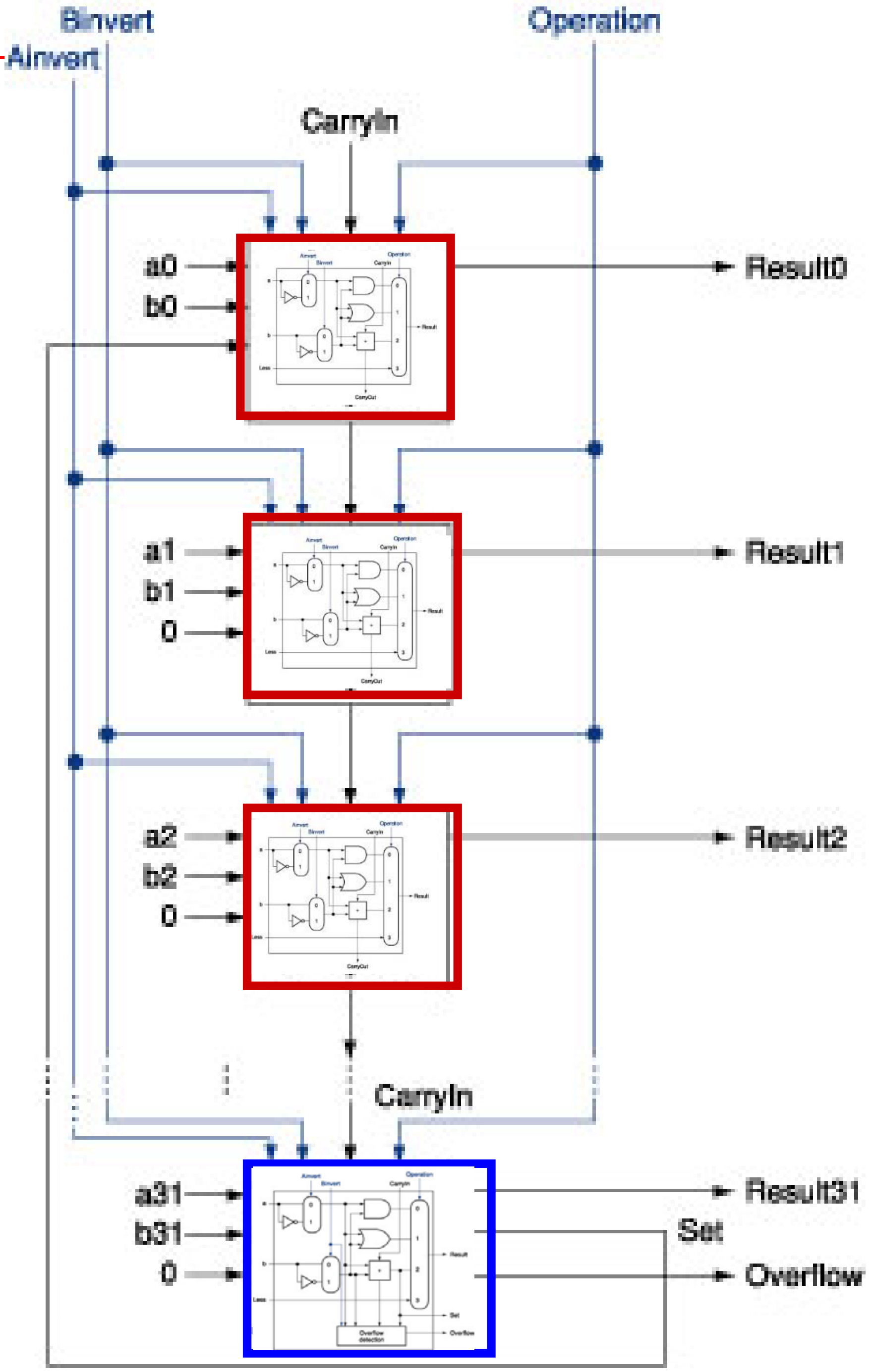


Set less than: result=1 when (A<B)

C = (A < B)   is syntax correct?

- A 32-bit ALU constructed from
  - the 31 copies of the 1-bit ALU in the top
  - one 1-bit ALU in the bottom
  - The Less inputs are connected to 0 except for the least significant bit,
- ALU performs a – b

  Result = 0 . . . 001 if a < b,
  Result = 0 . . . 000 otherwise.

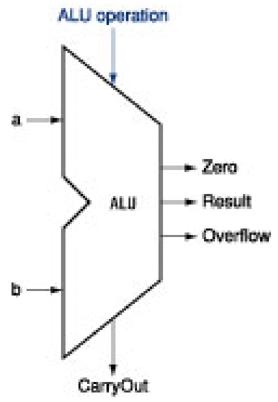


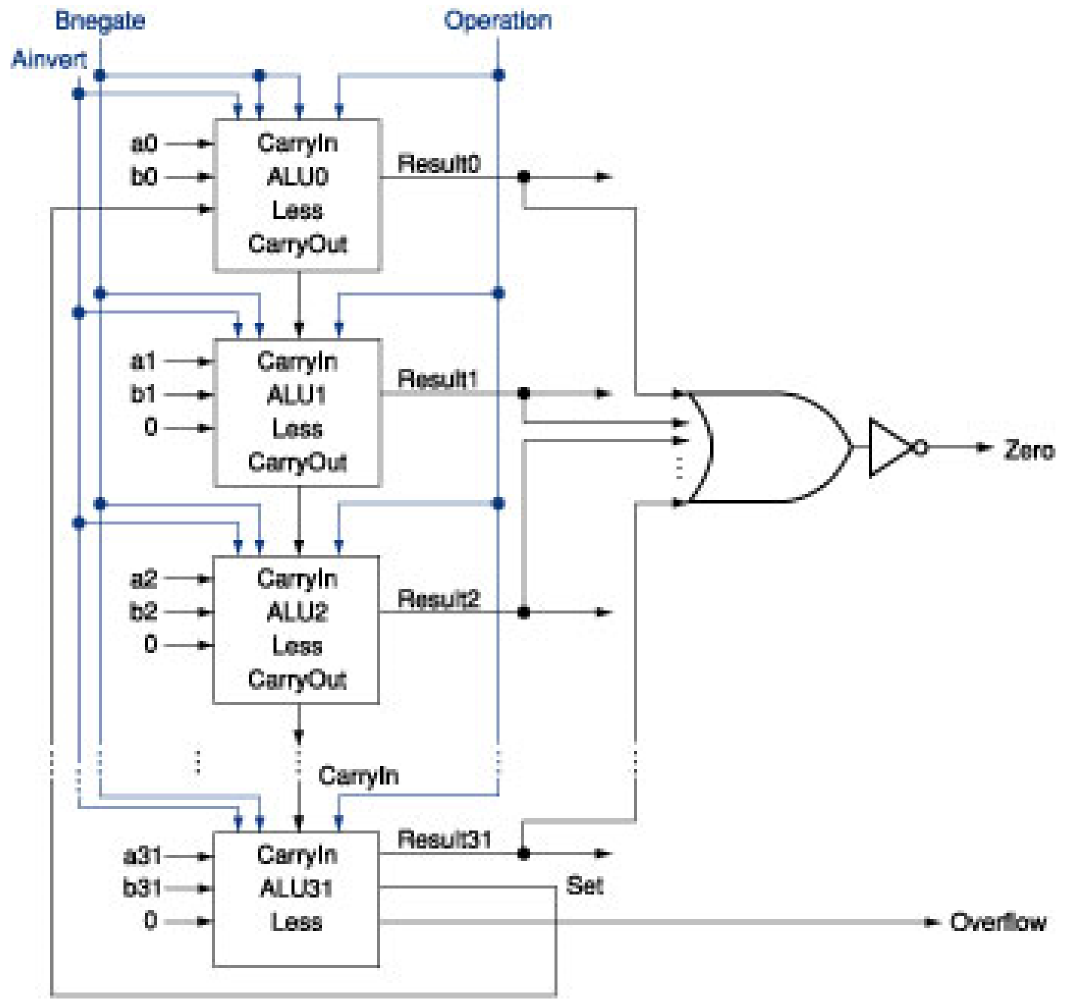
Supporting EQ instructions

- test for equality (beq \$1, \$2, L)

  (a-b) = 0 implies a = b

zero is a 1 when the result is 0!





- control lines:

0000 AND
0001 OR
0010 add
0110 subtract
0111 set on less than
1100 NOR

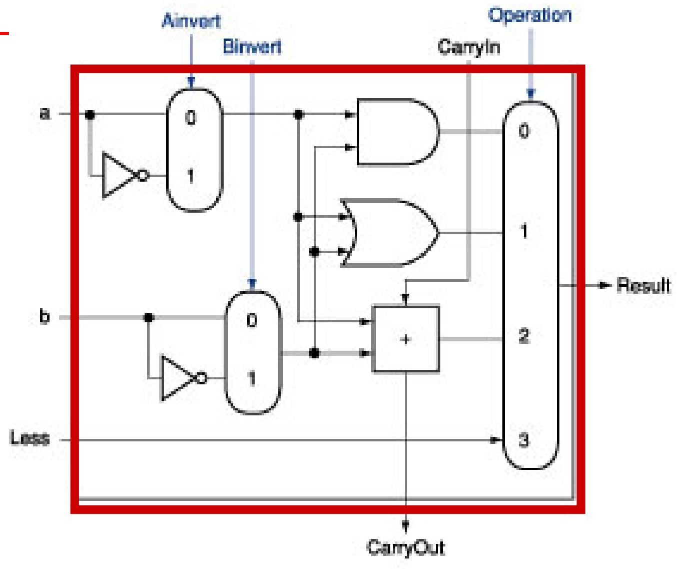
Support more comparisons

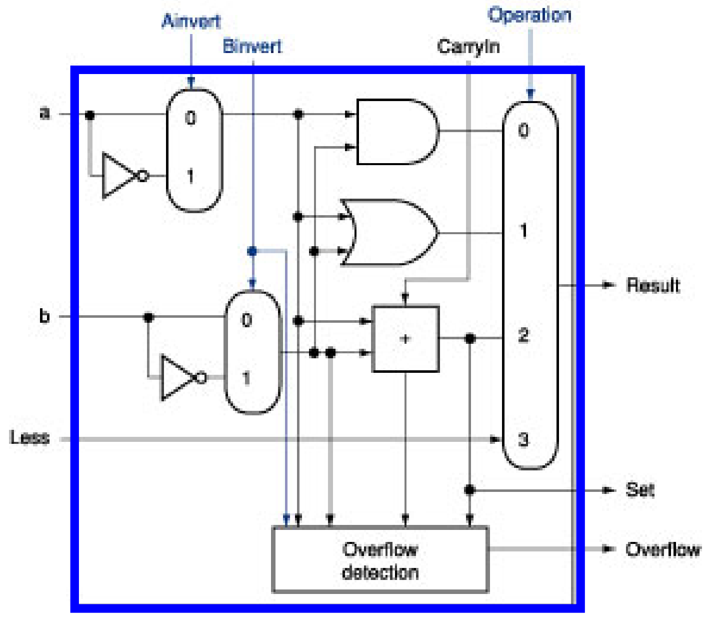
If we have LT & EQ:

- < LT (less than)
- > GT (greater than)
- <= LE (less and equal)
- >= GE (greater and equal)
- = EQ (equal)
- != NE (not equal)

Bonus: how can we generate…

- SEQ
- SNE
- SGT
- SGE
- SLE





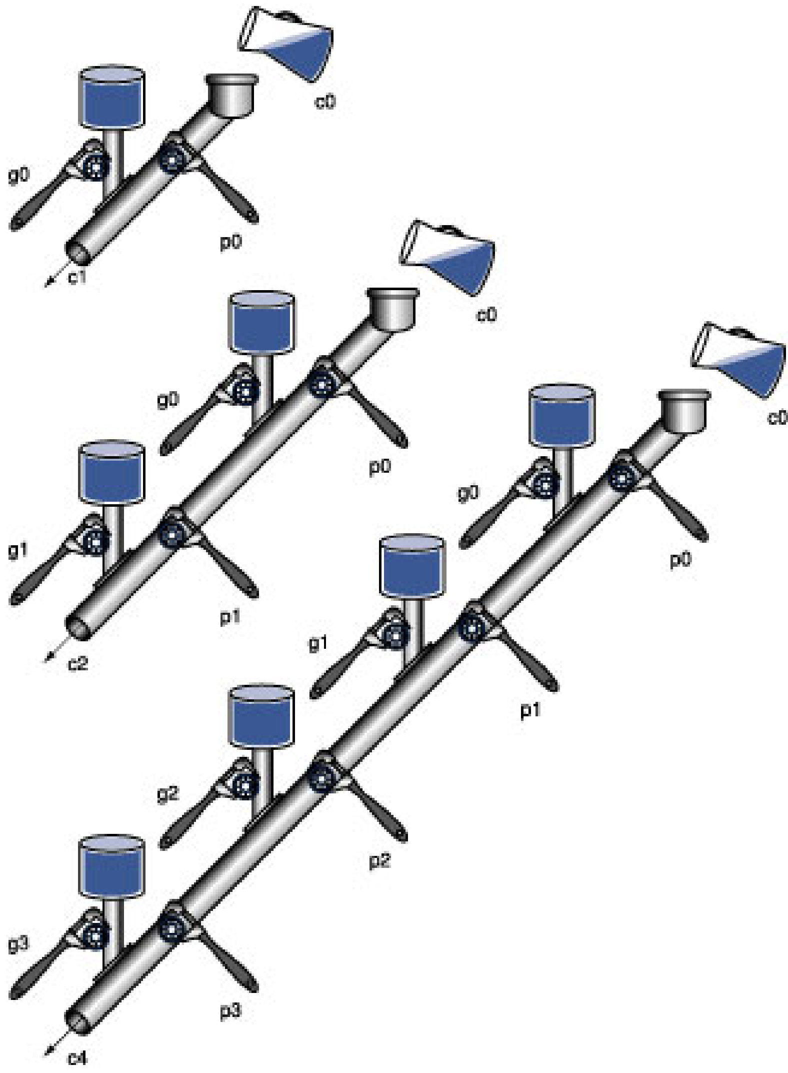
Improve adder by Carry lookahead

- ripple carry adder is slow
- sum-of-products is too expensive
- Carry-lookahead adder

  When generate a carry? gi = ai*bi

  When propagate the carry? pi = ai + bi

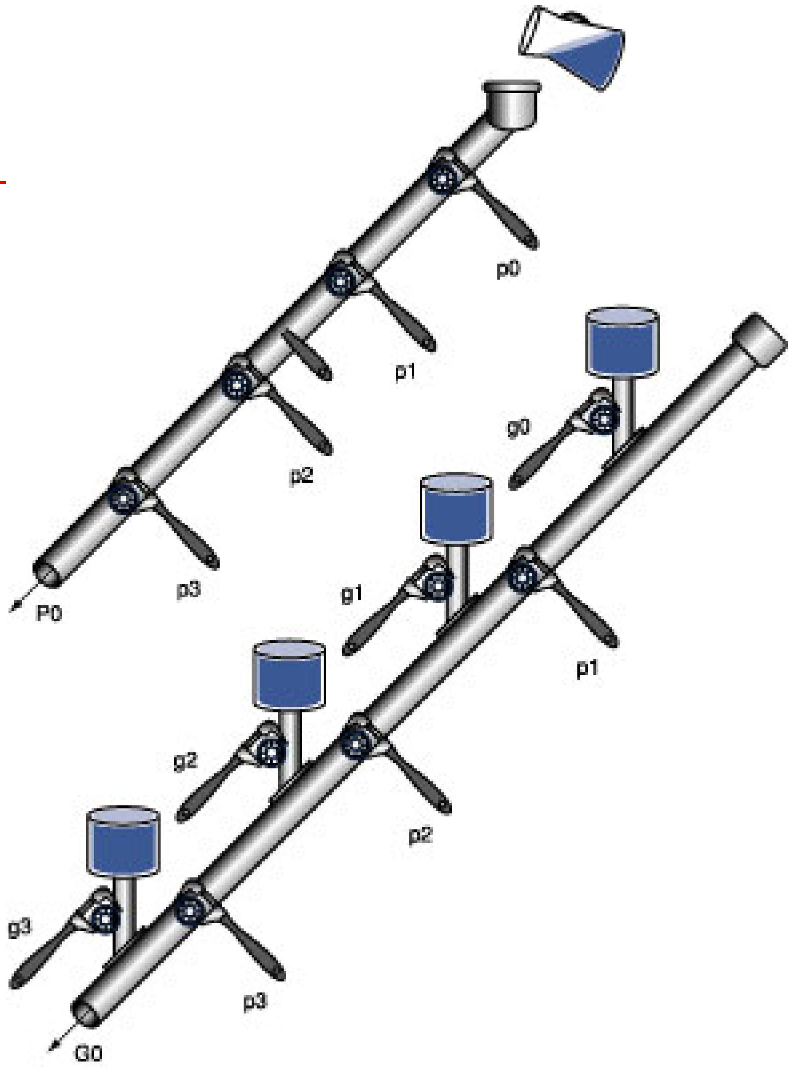
$$
\begin{aligned}
c1 &= g0 + (p0 \cdot c0) \\
c2 &= g1 + (p1 \cdot g0) + (p1 \cdot p0 \cdot c0) \\
c3 &= g2 + (p2 \cdot g1) + (p2 \cdot p1 \cdot g0) + (p2 \cdot p1 \cdot p0 \cdot c0) \\
c4 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\
&\quad + (p3 \cdot p2 \cdot p1 \cdot p0 \cdot c0)
\end{aligned}
$$



2nd level abstraction -

- super propagate

$$
\begin{aligned}
P0 &= p3 \cdot p2 \cdot p1 \cdot p0 \\
P1 &= p7 \cdot p6 \cdot p5 \cdot p4 \\
P2 &= p11 \cdot p10 \cdot p9 \cdot p8 \\
P3 &= p15 \cdot p14 \cdot p13 \cdot p12
\end{aligned}
$$



$$
\begin{aligned}
G0 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\
G1 &= g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4) \\
G2 &= g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8) \\
G3 &= g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12)
\end{aligned}
$$

Super-carry

- Example in textbook

$$
\begin{aligned}
C1 &= G0 + (P0 \cdot c0) \\
C2 &= G1 + (P1 \cdot G0) + (P1 \cdot P0 \cdot c0) \\
C3 &= G2 + (P2 \cdot G1) + (P2 \cdot P1 \cdot G0) + (P2 \cdot P1 \cdot P0 \cdot c0) \\
C4 &= G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) \\
&\quad + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0)
\end{aligned}
$$

Both Levels of the Propagate and Generate

Determine the *gi*, *pi*, *Pi*, and *Gi* values of these two 16-bit numbers:

```
a:      0001 1010 0011 0011two
b:      1110 0101 1110 1011two
```

Also, what is CarryOut15 (C4)?

Aligning the bits makes it easy to see the values of generate *gi* $(ai \cdot bi)$ and propagate *pi* $(ai + bi)$:

```
a:      0001 1010 0011 0011
b:      1110 0101 1110 1011
gi:     0000 0000 0010 0011
pi:     1111 1111 1111 1011
```

where the bits are numbered 15 to 0 from left to right. Next, the "super" propagates (P3, P2, P1, P0) are simply the AND of the lower-level propagates:

$$
\begin{aligned}
P3 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\
P2 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\
P1 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\
P0 &= 1 \cdot 0 \cdot 1 \cdot 1 = 0
\end{aligned}
$$

$$
\begin{aligned}
G0 &= g3 + (p3 \cdot g2) + (p3 \cdot p2 \cdot g1) + (p3 \cdot p2 \cdot p1 \cdot g0) \\
&= 0 + (1 \cdot 0) + (1 \cdot 0 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) = 0 + 0 + 0 + 0 = 0 \\
G1 &= g7 + (p7 \cdot g6) + (p7 \cdot p6 \cdot g5) + (p7 \cdot p6 \cdot p5 \cdot g4) \\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 1 + 0 = 1 \\
G2 &= g11 + (p11 \cdot g10) + (p11 \cdot p10 \cdot g9) + (p11 \cdot p10 \cdot p9 \cdot g8) \\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0 \\
G3 &= g15 + (p15 \cdot g14) + (p15 \cdot p14 \cdot g13) + (p15 \cdot p14 \cdot p13 \cdot g12) \\
&= 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0) = 0 + 0 + 0 + 0 = 0
\end{aligned}
$$

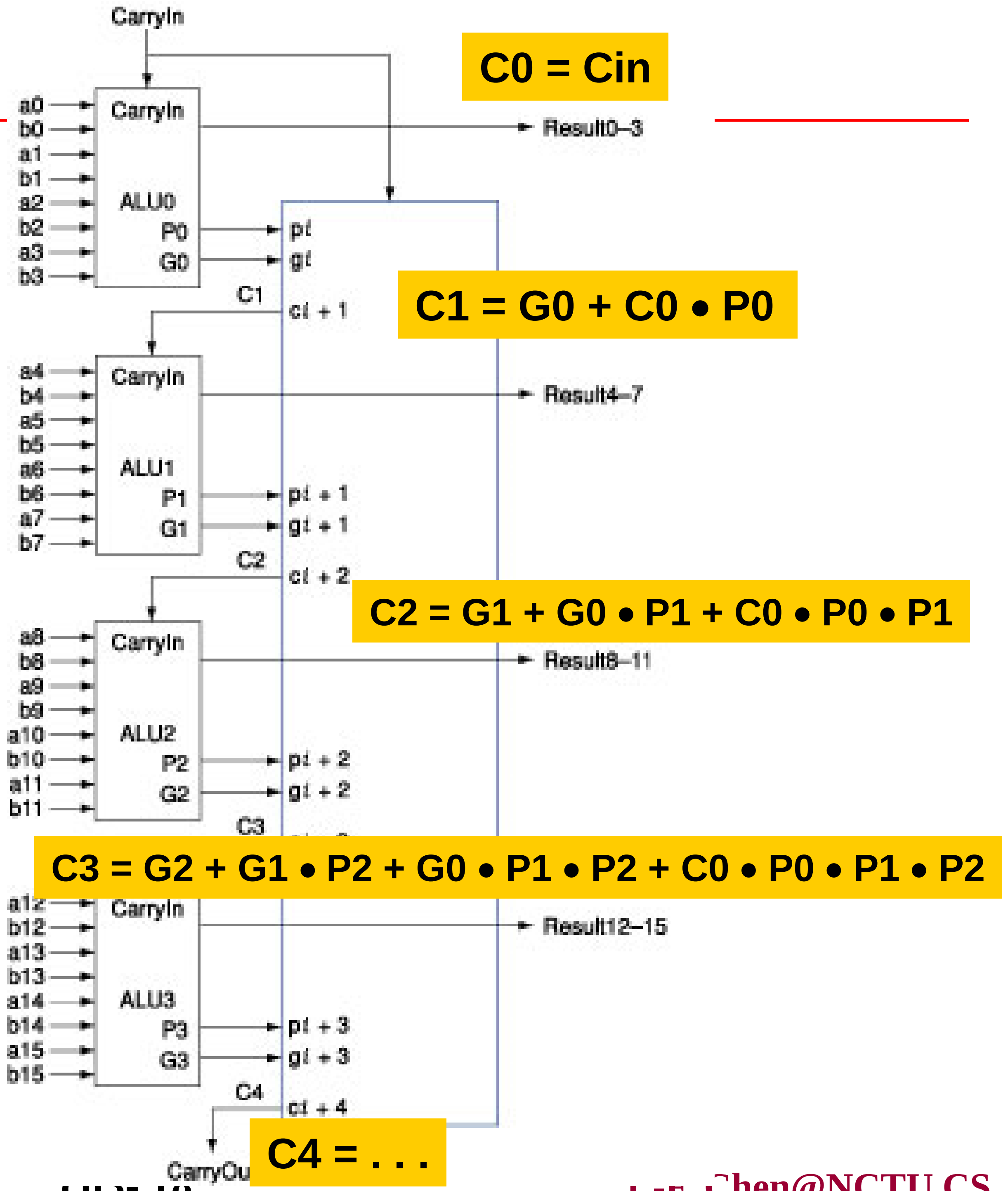
$$
\begin{aligned}
C4 = {} & G3 + (P3 \cdot G2) + (P3 \cdot P2 \cdot G1) + (P3 \cdot P2 \cdot P1 \cdot G0) \\
& + (P3 \cdot P2 \cdot P1 \cdot P0 \cdot c0) \\
= {} & 0 + (1 \cdot 0) + (1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 0 \cdot 0) \\
= {} & 0 + 0 + 1 + 0 + 0 = 1
\end{aligned}
$$

Four 4-bit ALUs

- Could use ripple carry of 4-bit CLA adders
- Better: use the CLA principle again
- Gate delay
  - Ripple carry adder:

    16*2=32
  - W/ CLA

    2+2+1=5



C0 = Cin

C1 = G0 + C0 • P0

C2 = G1 + G0 • P1 + C0 • P0 • P1

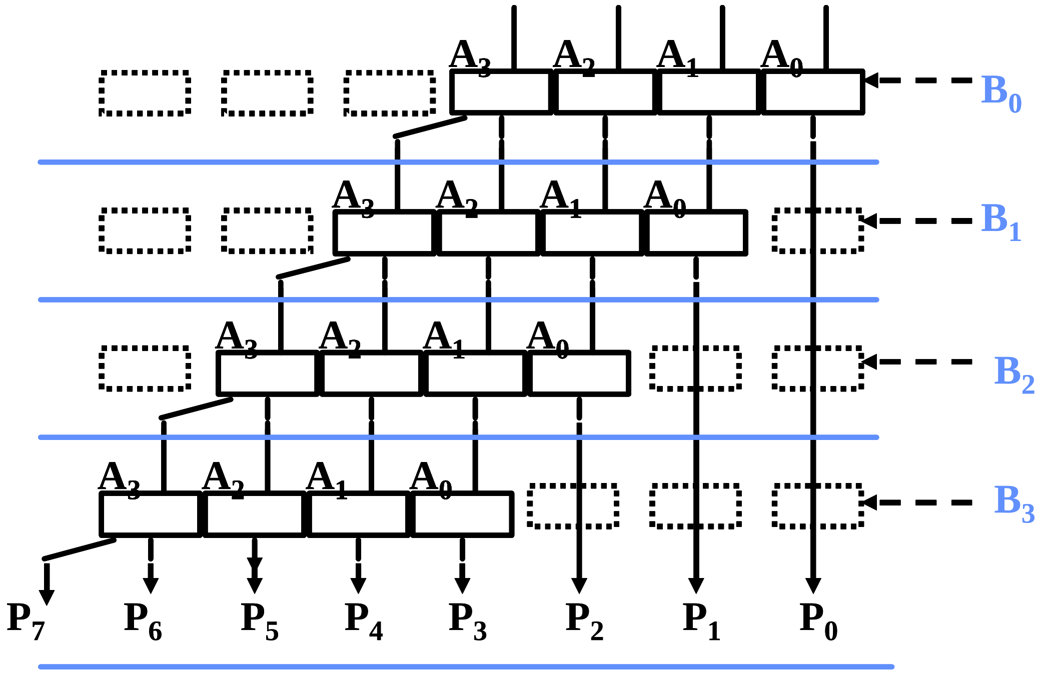
C3 = G2 + G1 • P2 + G0 • P1 • P2 + C0 • P0 • P1 • P2

C4 = . . .

Multiplication

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

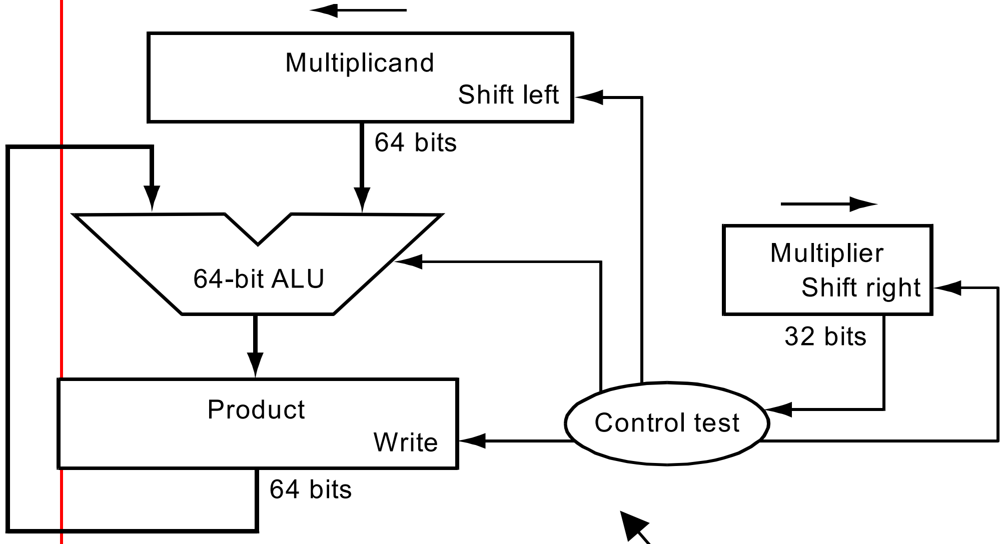
```
       0010   (multiplicand)
   __x_1011   (multiplier)
```



- Negative numbers:

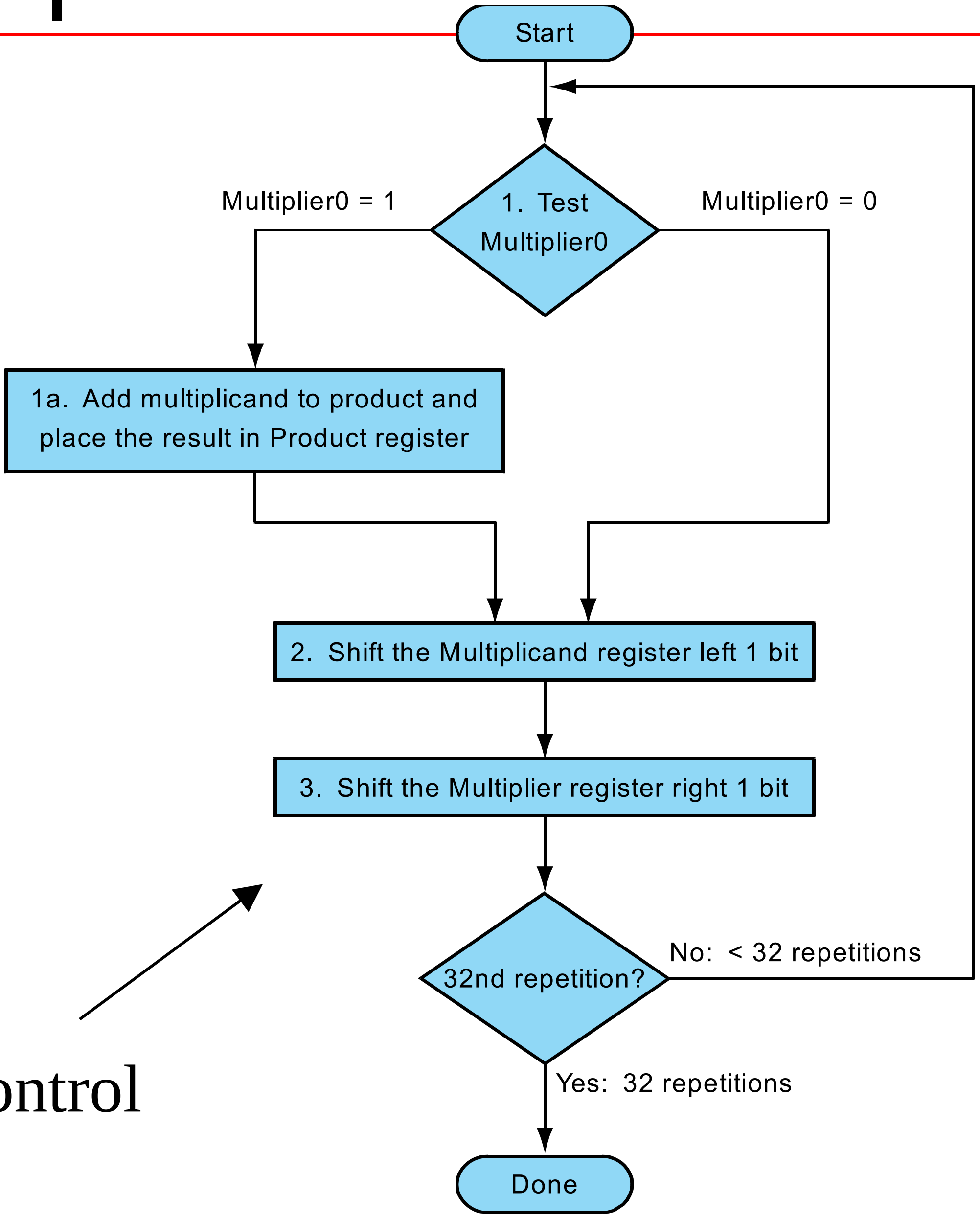
  convert and multiply

Multiplication: 1st sequential version



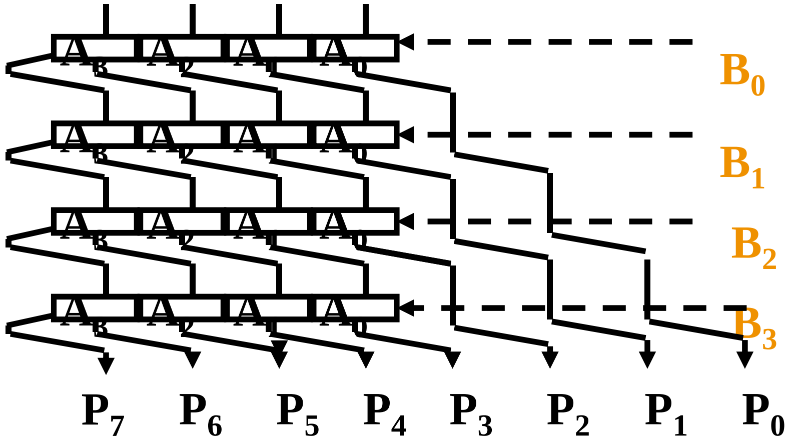
Datapath

Control

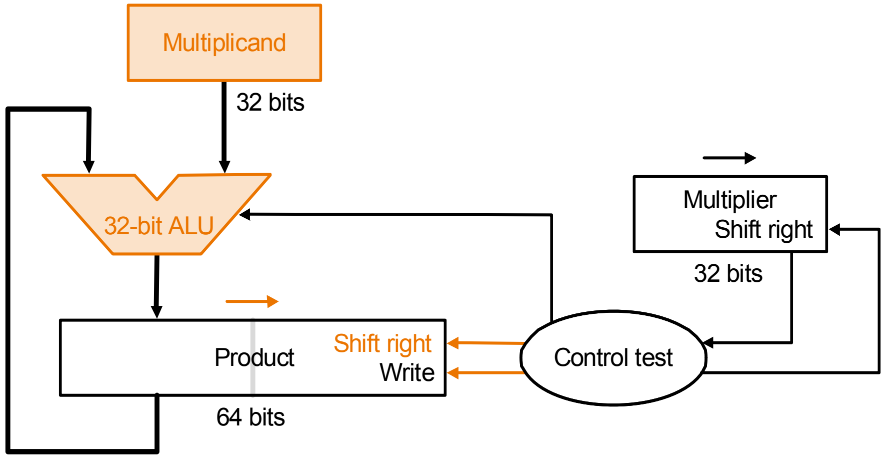

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift the Multiplicand register left 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

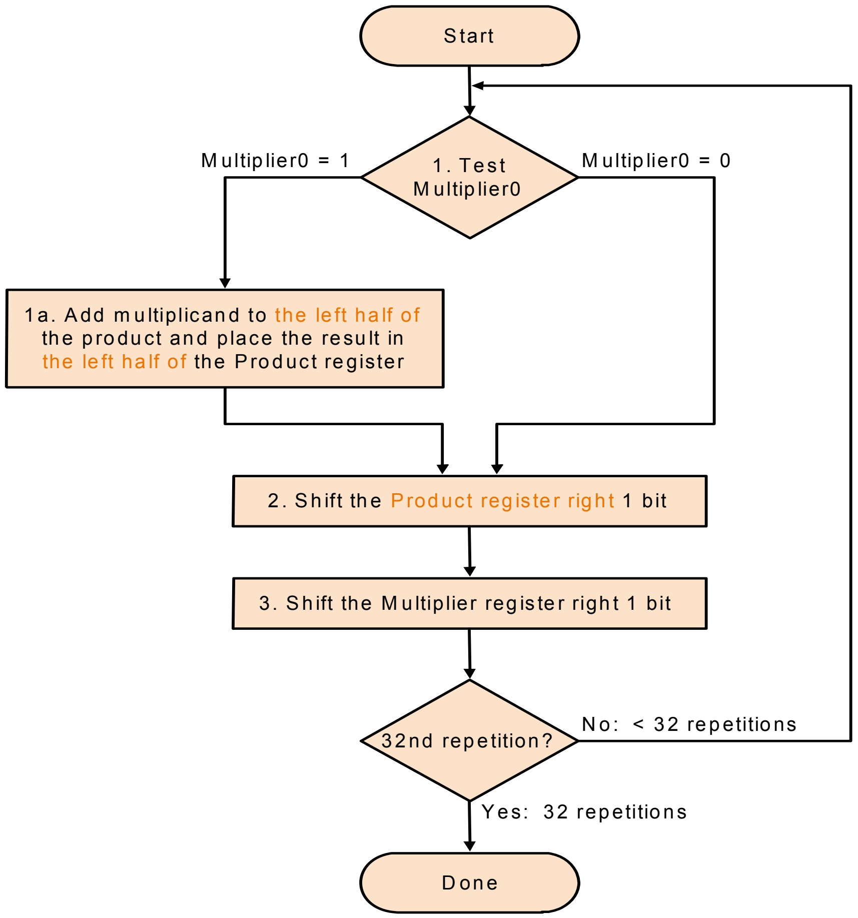
2nd version of multiplication

- Half of multiplicand is 0



- why not Shift product right?

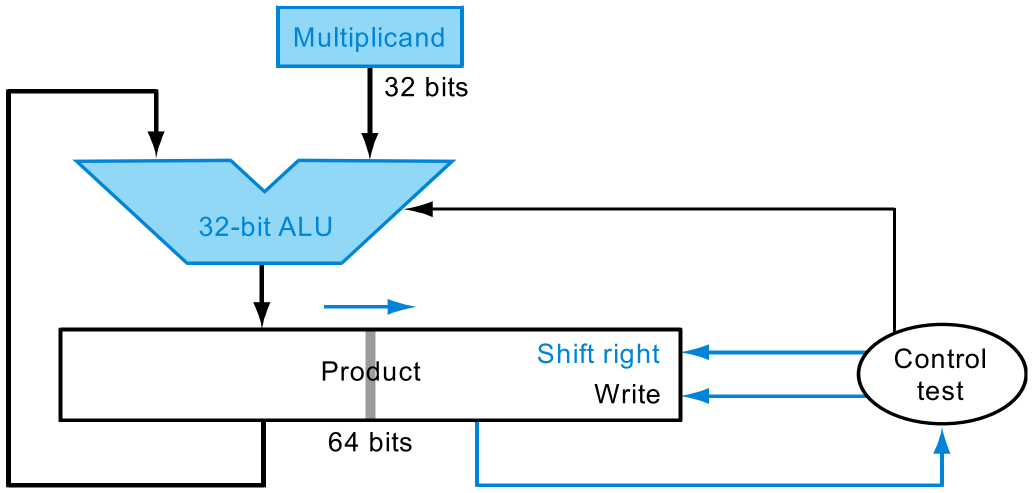


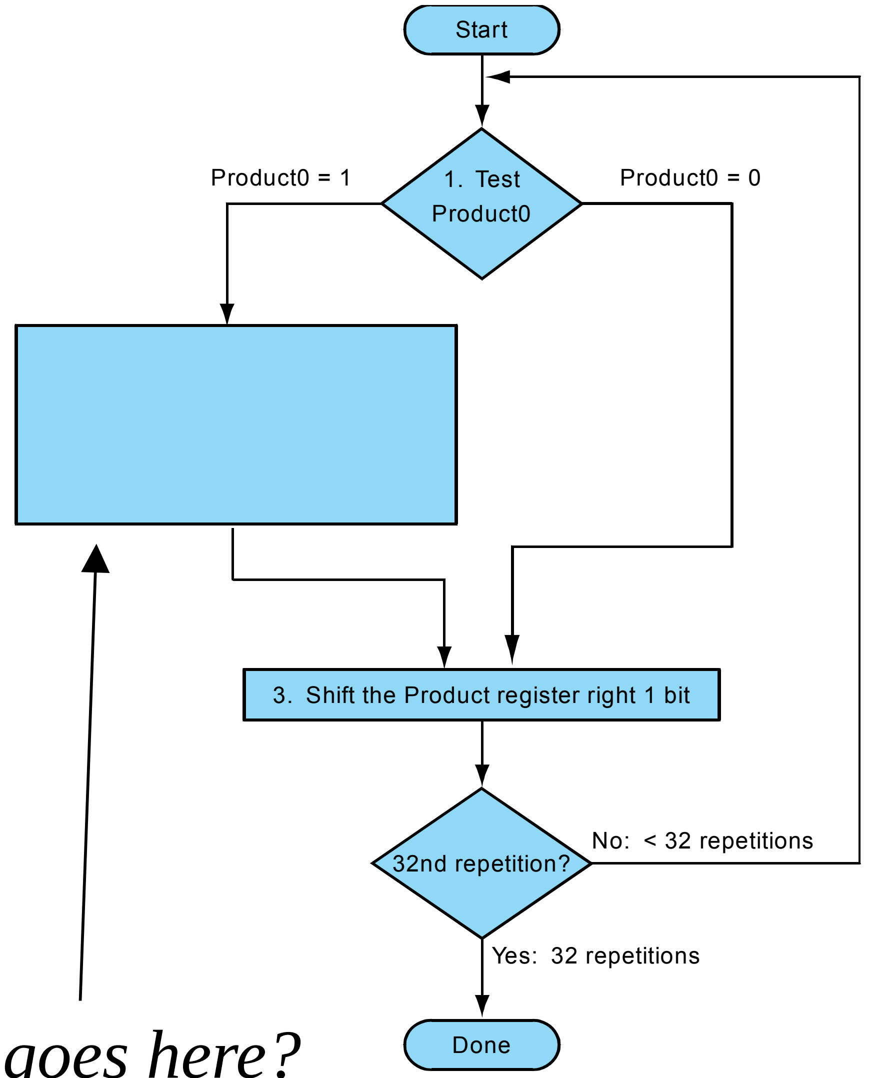


Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to the left half of the product and place the result in the left half of the Product register
2. Shift the Product register right 1 bit
3. Shift the Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Final Version

- Combine right part of the product with multiplier
- Multiplier starts in right half of product



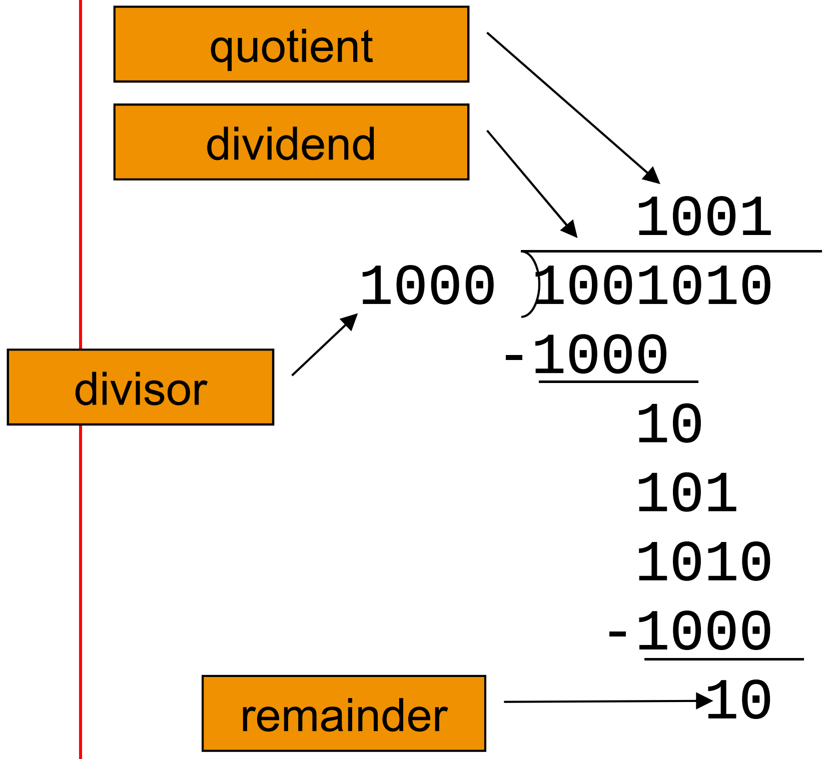


What goes here?

RISC-V Multiplication

- Four multiply instructions:
  - mul: multiply
    - Gives the lower 64 bits of the product
  - mulh: multiply high
    - Gives the upper 64 bits of the product, assuming the operands are signed
  - mulhu: multiply high unsigned
    - Gives the upper 64 bits of the product, assuming the operands are unsigned
  - mulhsu: multiply high signed/unsigned
    - Gives the upper 64 bits of the product, assuming one operand is signed and the other unsigned
  - Use mulh result to check for 64-bit overflow

Division

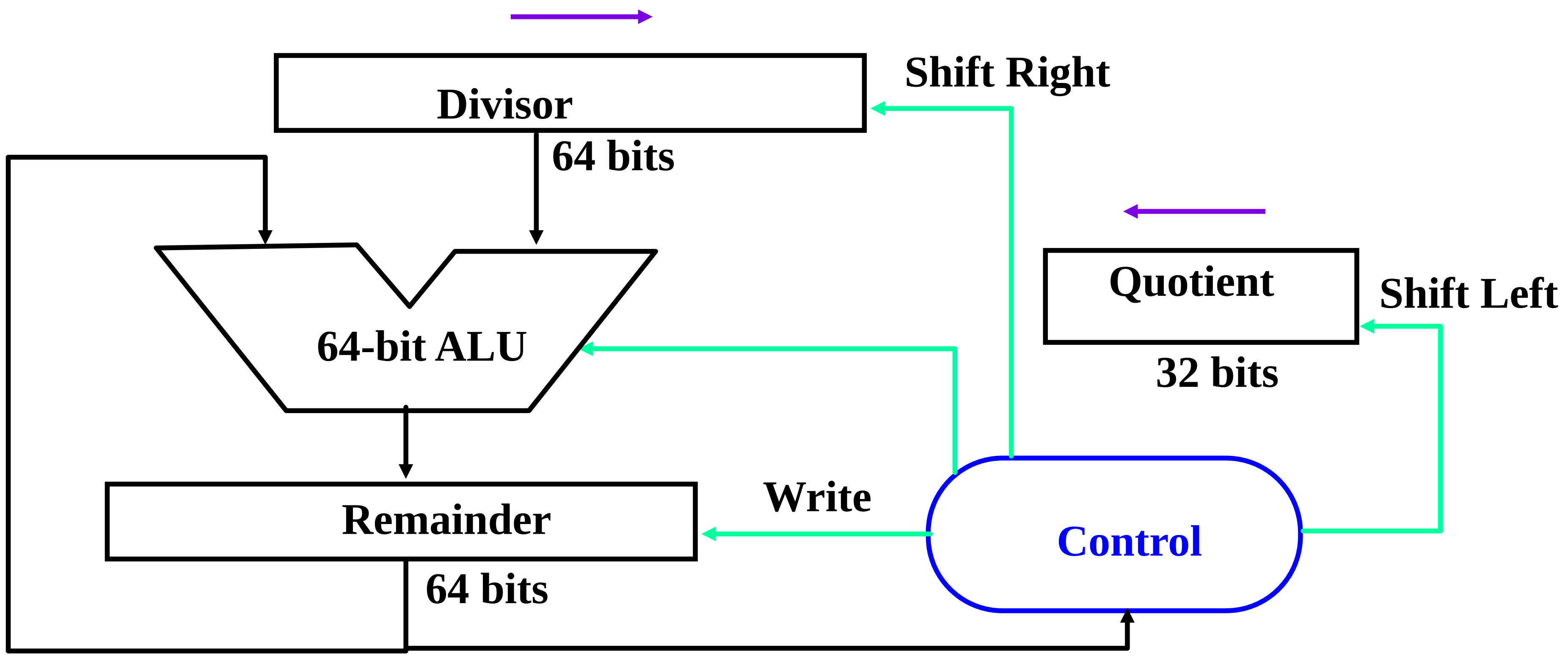


n-bit operands yield *n*-bit quotient and remainder

- Check for 0 divisor
- Long division approach
  - If divisor ≤ dividend bits
    - 1 bit in quotient, subtract
  - Otherwise
    - 0 bit in quotient, bring down next dividend bit
- Restoring division
  - Do the subtract, and if remainder goes < 0, add divisor back
- Signed division
  - Divide using absolute values
  - Adjust sign of quotient and remainder as required

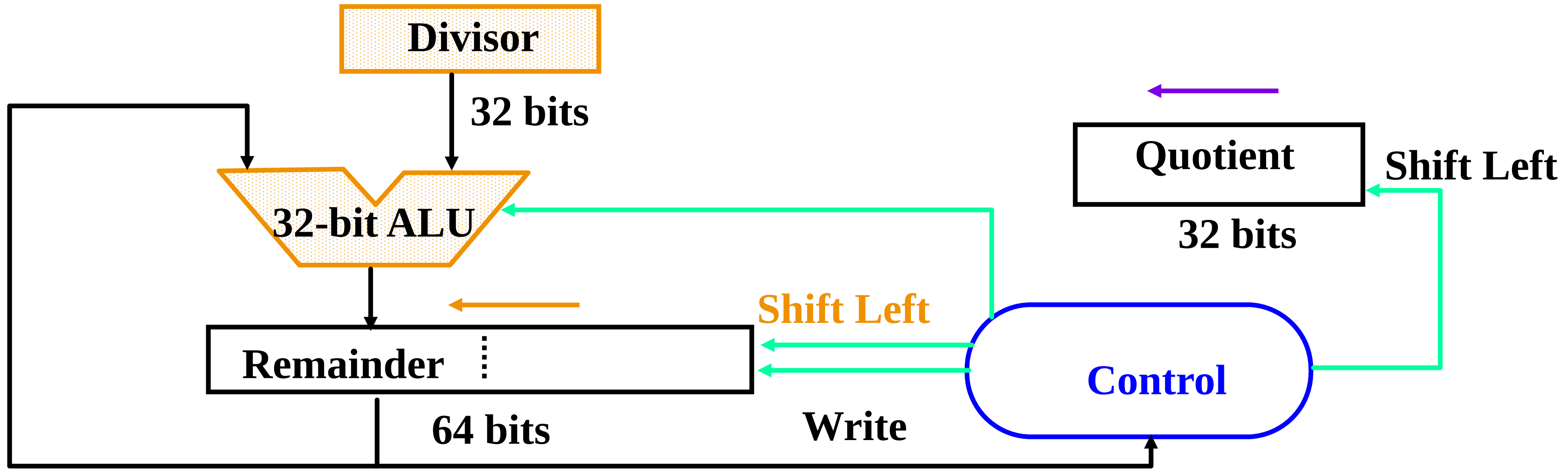
DIVIDE HARDWARE Version 1

- 64-bit Divisor reg, 64-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



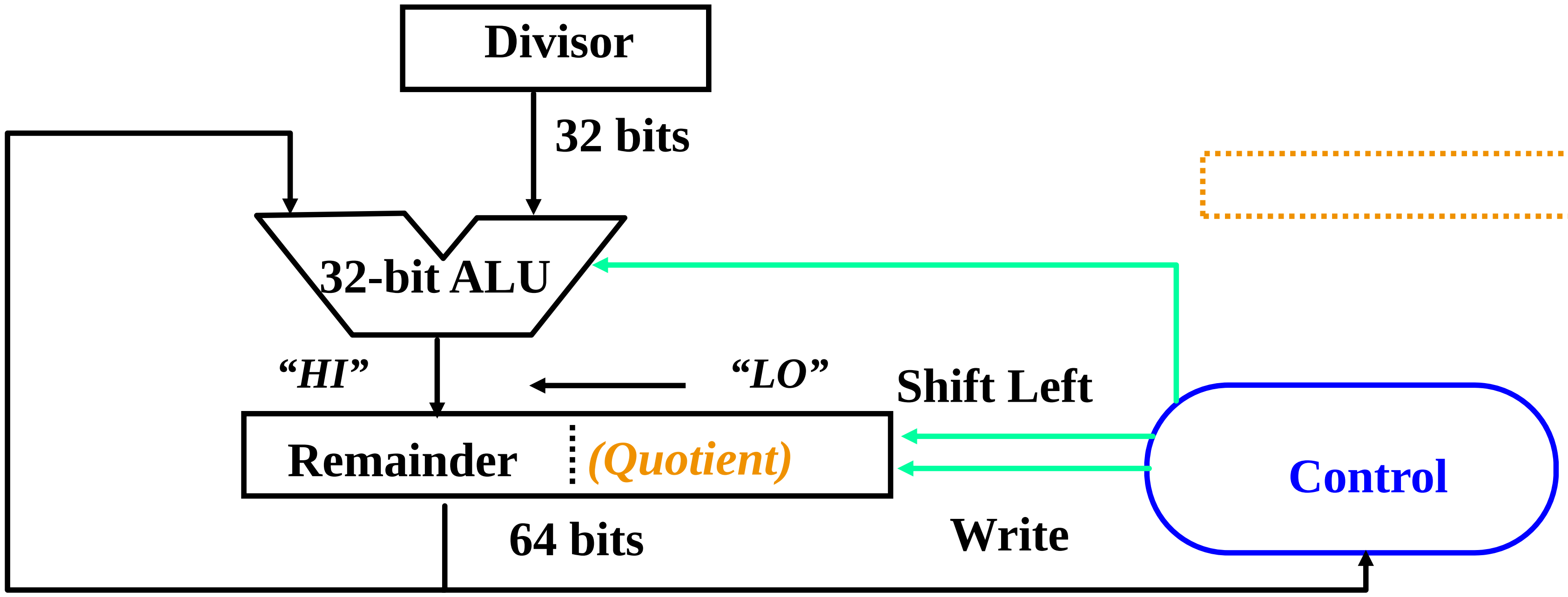
DIVIDE HARDWARE Version 2

- 32-bit Divisor reg, 32-bit ALU, 64-bit Remainder reg, 32-bit Quotient reg



DIVIDE HARDWARE Version 3

- 32-bit Divisor reg, 32 -bit ALU, 64-bit Remainder reg, (0-bit Quotient reg)



Floating Point (a brief look)

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., 3.15576 ´ 10^9
- Representation:
  - sign, exponent, significand: $(–1)^{sign}$ ´ significand ´ $2^{exponent}$
  - more bits for significand gives more accuracy
  - more bits for exponent increases range
- IEEE 754 floating point standard:
  - single precision: 8 bit exponent, 23 bit significand
  - double precision: 11 bit exponent, 52 bit significand

IEEE Floating-Point Format

	single: 8 bits double: 11 bits	single: 23 bits double: 52 bits
S	Exponent	Fraction

$$x = (-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$$

- S: sign bit (0 $\Rightarrow$ non-negative, 1 $\Rightarrow$ negative)
- Normalize significand: $1.0 \leq |\text{significand}| < 2.0$
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the “1.” restored
- Exponent: excess representation: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1203

Floating-Point Example

- Represent –0.75
  - $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
  - S = 1
  - Fraction = $1000\ldots00_2$
  - Exponent = –1 + Bias
    - Single: $-1 + 127 = 126 = 01111110_2$
    - Double: $-1 + 1023 = 1022 = 01111111110_2$
- Single: 1011111101000…00
- Double: 1011111111101000…00

Floating-Point Addition

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)
- 1. Align binary points
  - Shift number with smaller exponent
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$
- 2. Add significands
  - $1.000_2 \times 2^{-1} + -0.111_2 \times 2{-}1 = 0.001_2 \times 2^{-1}$
- 3. Normalize result & check for over/underflow
  - $1.000_2 \times 2^{-4}$, with no over/underflow
- 4. Round and renormalize if necessary
  - $1.000_2 \times 2^{-4}$ (no change) = 0.0625

Basic Addition Algorithm/Multiply issues

For addition (or subtraction) this translates into the following steps:

(1) compute Ye - Xe *(getting ready to align binary point)*

(2) right shift Xm that many positions to form Xm 2^{Xe-Ye}

(3) compute Xm 2^{Xe-Ye} + Ym

if representation demands normalization, then normalization step follows:

(4) left shift result, decrement result exponent (e.g., 0.001xx…)
right shift result, increment result exponent (e.g., 101.1xx…)
continue until MSB of data is 1 (NOTE: Hidden bit in IEEE Standard)

(5) for multiply, doubly biased exponent must be corrected:

Xe = 7	Xe = 1111	= 15	= 7 + 8
Ye = -3	Ye = 0101	= 5	= -3 + 8
Excess 8	10100	20	$4 + 8 + \underbrace{8}$

extra subtraction step of the bias amount

(6) if result is 0 mantissa, may need to zero exponent by special step

FP Instructions in RISC-V

- Separate FP registers: f0, …, f31
  - double-precision
  - single-precision values stored in the lower 32 bits
- FP instructions operate only on FP registers
  - Programs generally don't do integer ops on FP data, or vice versa
  - More registers with minimal code-size impact
- FP load and store instructions
  - `flw, fld`
  - `fsw, fsd`
- Single-precision arithmetic
  - `fadd.s, fsub.s, fmul.s, fdiv.s, fsqrt.s`
    - e.g., `fadds.s f2, f4, f6`