Chapter 3

Arithmetic for Computers

Arithmetic for Computers

- Operations on integers
  - Addition and subtraction (+,-)
  - Multiplication and division (*,/)
  - Dealing with overflow
- Floating-point real numbers
  - Representation and four operations (+,-,*,/)

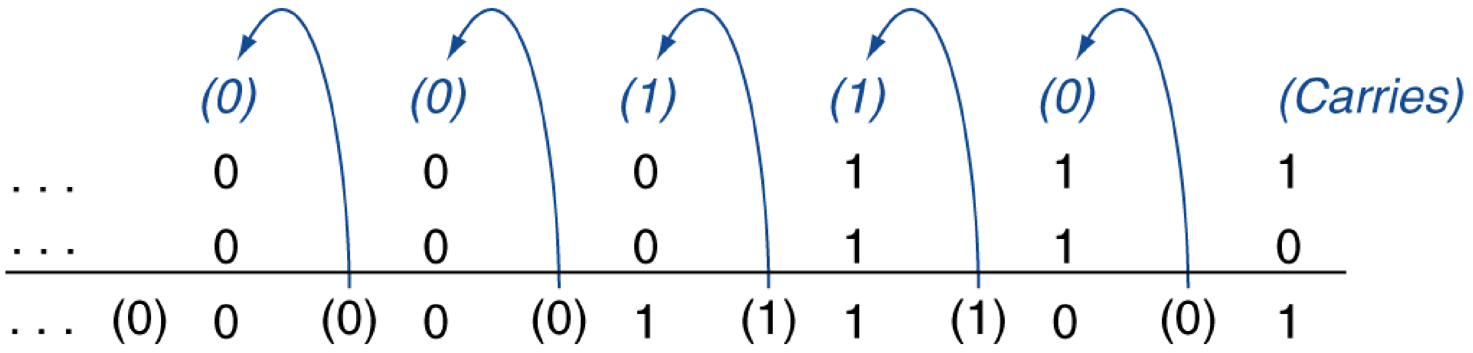
Review 2's complement numbers

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
no corresponding positive number ➔ No +8		-8	1000

2's complement range is asymmetric

Integer Addition

- Example: 7 + 6



- Overflow if result out of range
  - Adding positive and negative operands, no overflow
  - Adding two positive operands
    - Overflow if result sign is 1
  - Adding two negative operands
    - Overflow if result sign is 0

Integer Subtraction

- Add negation of second operand
- Example: 7 – 6 = 7 + (–6)

```
+7:     0000 0000 … 0000 0111
–6:     1111 1111 … 1111 1010
-----------------------------
+1:     0000 0000 … 0000 0001
```

- Overflow if result is out of range
  - Subtracting two positive or two negative operands, no overflow
  - Subtracting positive from negative operand
    - Overflow if result sign is 0
  - Subtracting negative from positive operand
    - Overflow if result sign is 1

Overflow

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

```
              0111
discard     + 1010
            ------
          1  0001
```

- 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"

Two positive numbers added gives a
negative result
➔ Overflow occurred

Overflow

- Overflow if result *out of range*
  - No overflow when adding a positive and a negative number
  - No overflow when signs of operands 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 ?

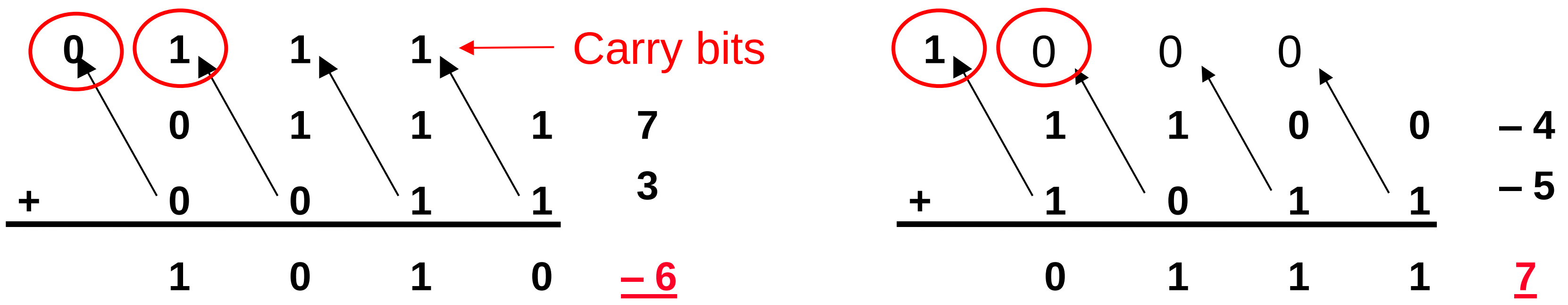
Overflow summary

Operation	Operand A	Operand B	Result indicating overflow
A + B	≥ 0	≥ 0	< 0
A + B	< 0	< 0	≥ 0
A - B	≥ 0	< 0	< 0
A - B	< 0	≥ 0	≥ 0

Overflow example

Examples: 7 + 3 = 10 but ...

- 4 - 5 = - 9 but ...

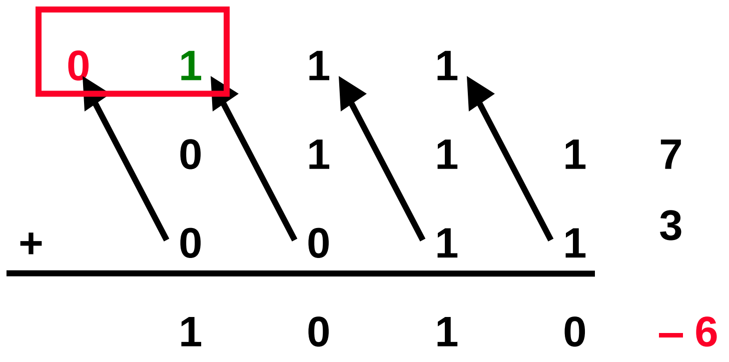


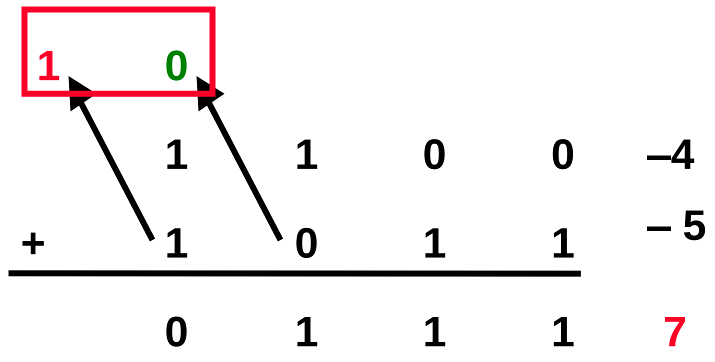
```
  0   1   1   1        Carry bits             1   0   0   0
      0   1   1   1     7                         1   1   0   0    – 4
+     0   0   1   1     3                  +      1   0   1   1    – 5
  ------------------                          ------------------
      1   0   1   0   – 6                         0   1   1   1      7
```

Carry in to MSB and carry out of MSB are different

Overflow Detection

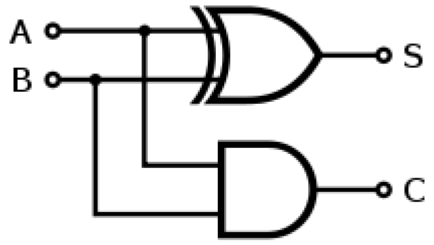
- Overflow: the result is too large (or too small) to represent properly
  - Example: - 8 ≤ 4-bit binary number ≤ 7
- When adding operands with different signs, overflow cannot occur!
- Overflow occurs when adding:
  - 2 positive numbers and the sum is negative
  - 2 negative numbers and the sum is positive
- On your own: Convince yourselves we can detect overflow by:
  - Carry into MSB ⊕ Carry out of MSB
  - An easy way to build overflow detection hardware!



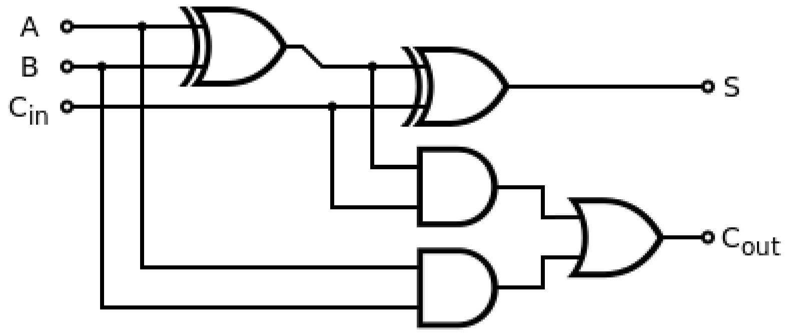


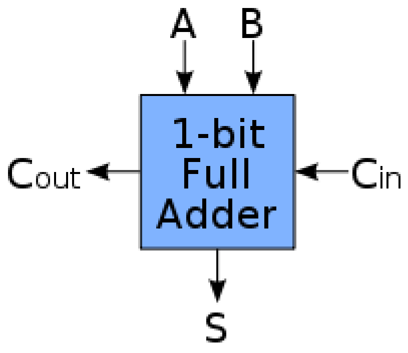
1-bit Integer addition in hardware

- All of this can be implemented in simple boolean logic circuits:
- 1-bit half adder circuit:



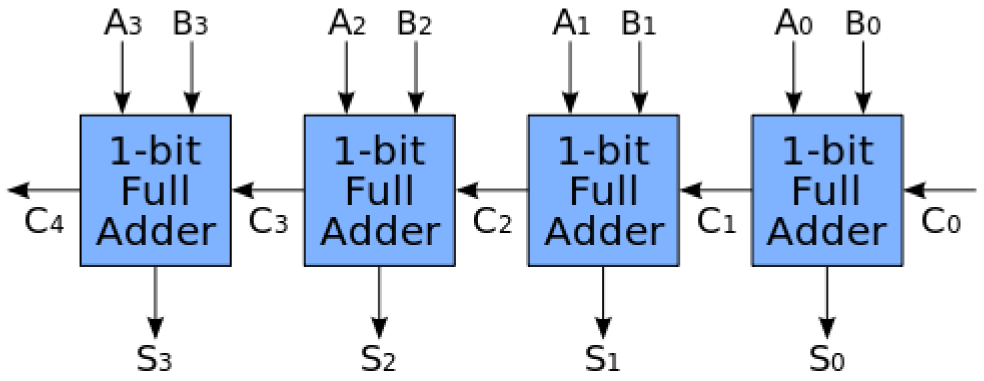
- 1-bit full adder circuit:





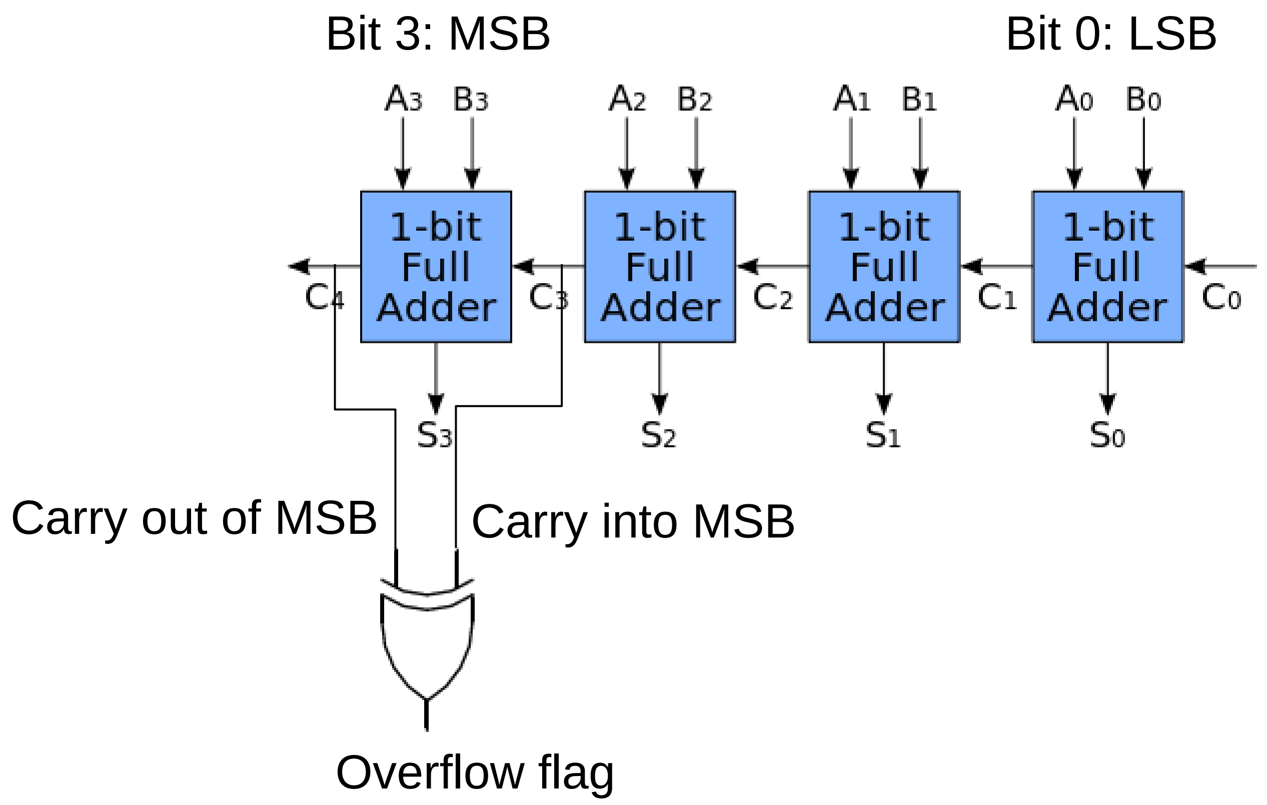
n-bit adder hardware

- String n 1-bit full adders with carry propagation to obtain an n-bit adder:



4-bit adder example

Overflow for the 4 bit adder



Dealing with Overflow

- Some languages (e.g., C) ignore overflow
  - Use MIPS `addu`, `addui`, `subu` instructions
- Other languages (e.g., Ada, Fortran) require raising an exception
  - Use MIPS `add`, `addi`, `sub` instructions
  - On overflow, invoke exception handler
    - Save PC in exception program counter (EPC) register
    - Jump to predefined handler address
    - `mfc0` (move from coprocessor reg) instruction can retrieve EPC value, to return after corrective action

Integer Multiplication (unsigned)

- Paper and pencil example (unsigned):

```
Multiplicand             1000
Multiplier               1001
                         ----
                         1000
                        0000
                       0000
                      1000
                      ----
Product               01001000
```

- m bits x n bits = m+n bit product
- Binary makes it easy:

  0 => place 0 (0 x multiplicand)

  1 => place a copy (1 x multiplicand)

- We will see 2 versions of multiply hardware & algorithm:
  - successive refinement

Multiplication algorithms

- If we had enough hardware, we can build the entire multiplication circuit as one giant combinational circuit that generates the result in one shot.
- Normally this requires a lot of hardware. So, the traditional implementation is an iterative approach

  Multiply two n bit numbers A and B to generate the product P

  P ← A*B

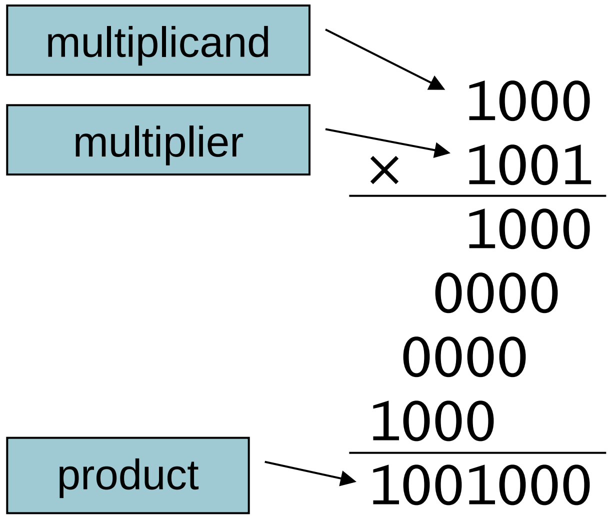
  Algorithm

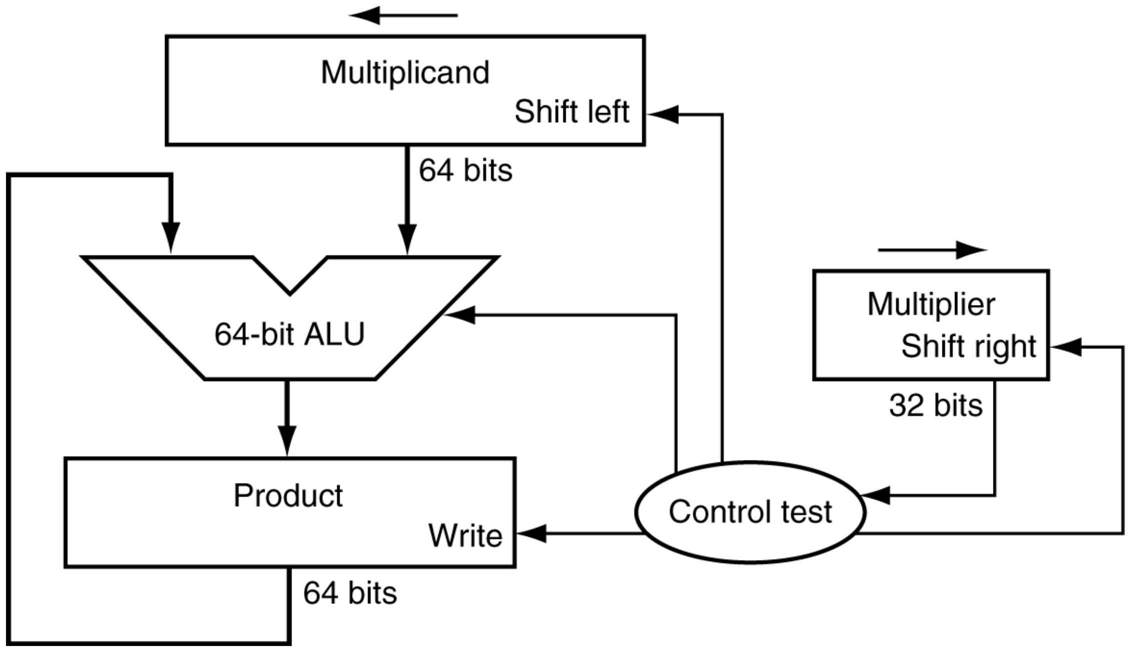
```
P ← 0 // initialize P to 0; will hold partial product
for i=0 to n-1 {
     // examine bit i of B
     if ($B_i$ == 1) {
        P ← P + A
     }
     else { /* do nothing */}
     A ← 2*A // shift A to the left by 1 bit for the next iteration
}
```

- Basic hardware solutions are variations of this algorithm.

Multiplication (attempt 1)

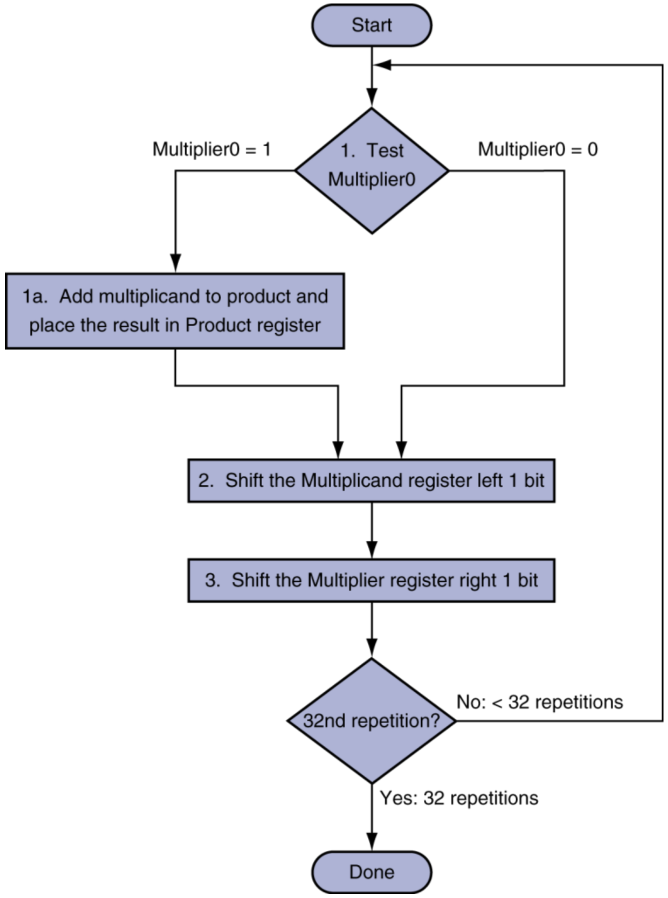
- 64-bit Multiplicand register, 64-bit ALU, 64-bit Product register, 32-bit multiplier register



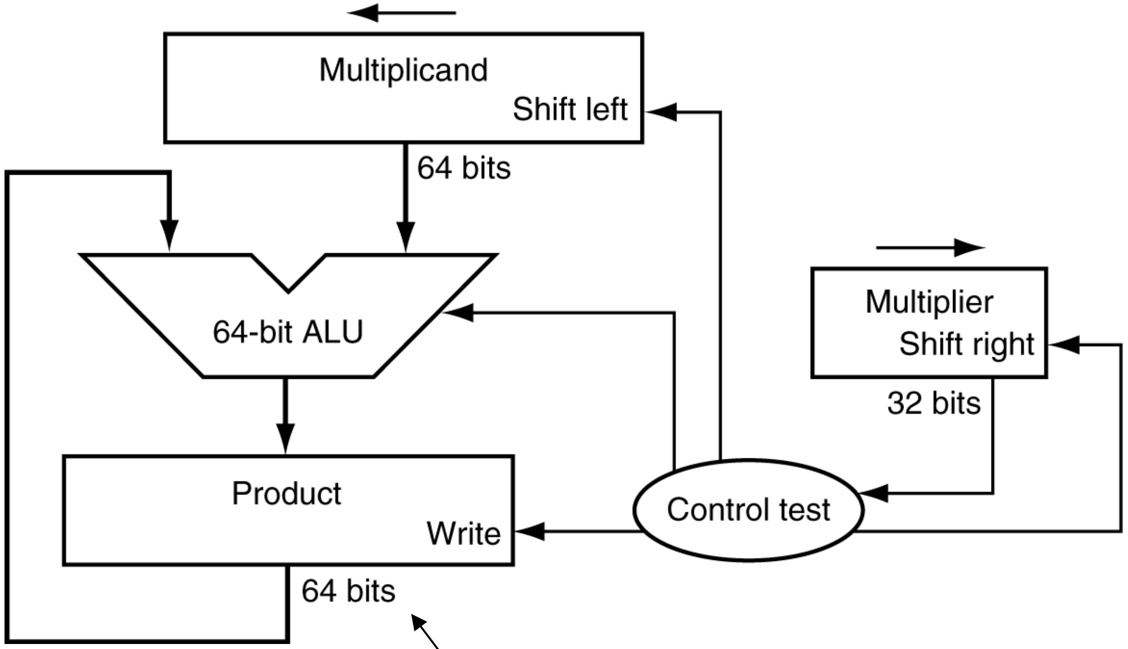


Length of product is the sum of operand lengths

Multiplication Hardware



Start
Multiplier0 = 1
1. Test Multiplier0
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

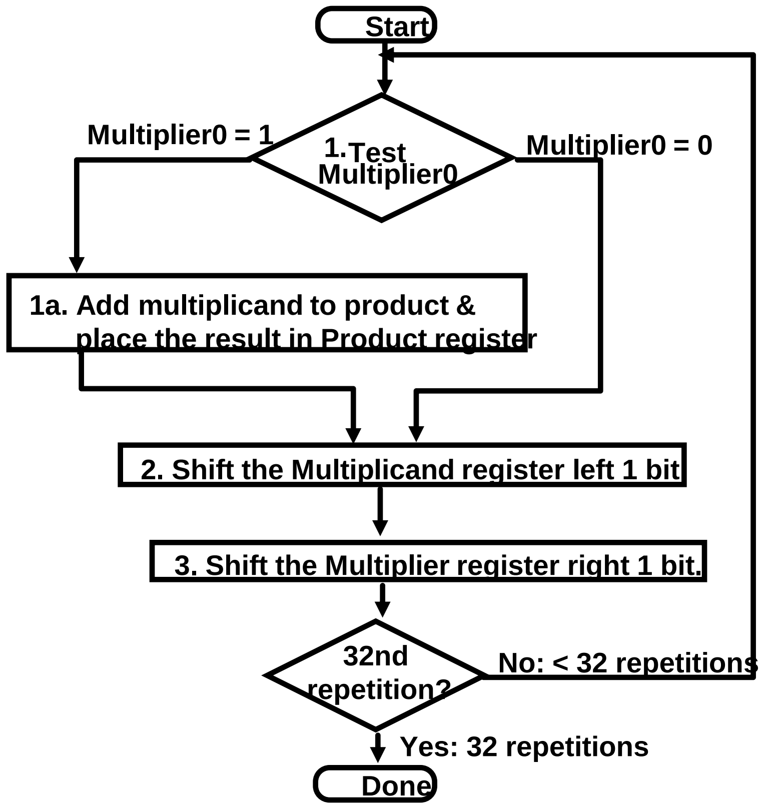


Initially 0

Multiply Algorithm Version 1

Example with 4 bit numbers multiplied instead of 32 bit ones.

Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	P=P+Mcand	
Shift Mcand left; Shift Multiplier right		
0000 0010	0001	0000 0100
0000 0110	P=P+Mcand	
Shift Mcand left; Shift Multiplier right		
0000 0110	0000	0000 1000
No add		
Shift Mcand left; Shift Multiplier right		
0000 0110	0000	0001 0000
No add		
Shift Mcand left; Shift Multiplier right		
0000 0110	0000	0010 0000
4 iterations. Done.		


Start
Multiplier0 = 1
1.Test Multiplier0
Multiplier0 = 0
1a. Add multiplicand to product & 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

Observations on Multiply Version 1

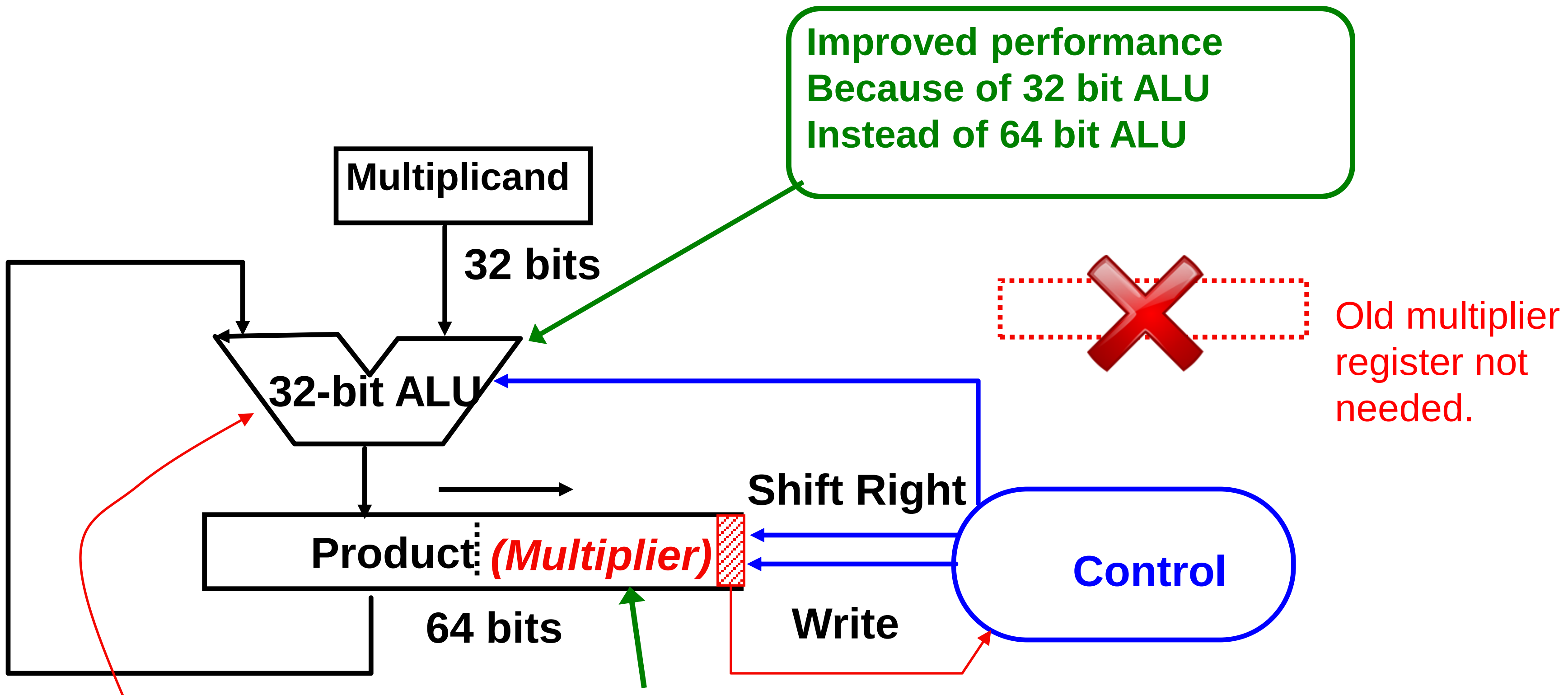
- 1/2 bits in multiplicand always 0
  - 64-bit adder is wasted
  - 64-bit adder delay is twice as long (more bits to propagate carry)
- 0’s inserted in left of multiplicand as shifted
  - least significant bits of product never changed once formed

Improvement:

- Instead of shifting multiplicand to left, shift product to right?
  - This allows us to use 32-bit ALU instead of 64-bit ALU.
  - savings in hardware and less delay.
- Start multiplier in lower half of product register
  - only one register to shift.

MULTIPLY HARDWARE Version 2

- 32-bit Multiplicand register, 32 -bit ALU, 64-bit Product register, (0-bit of Multiplier register tested)



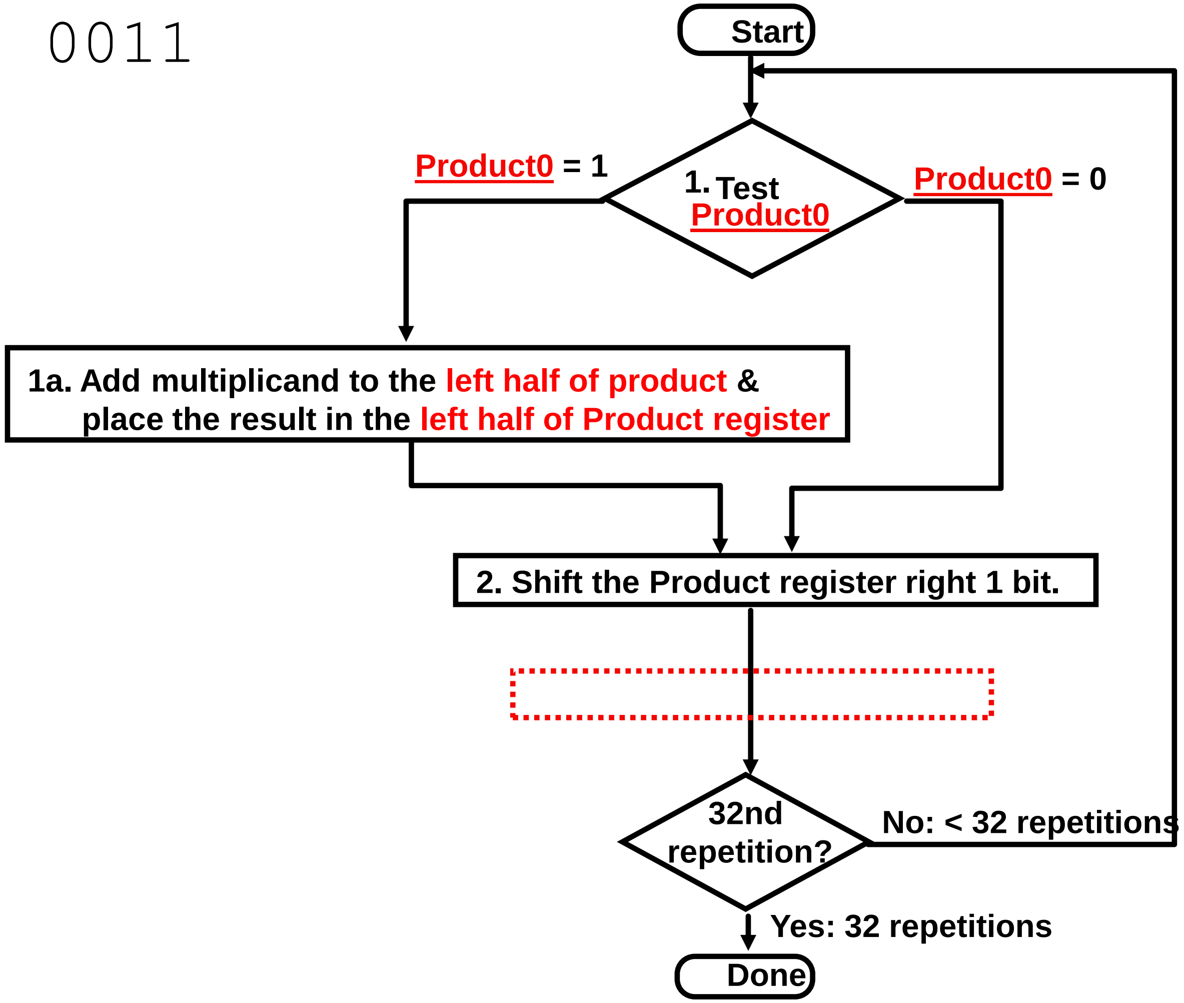
32 bit Add is done on the left half of Product register

Start multiplier in lower half of product register
➔ only one register to shift.

Multiply Algorithm Version 2

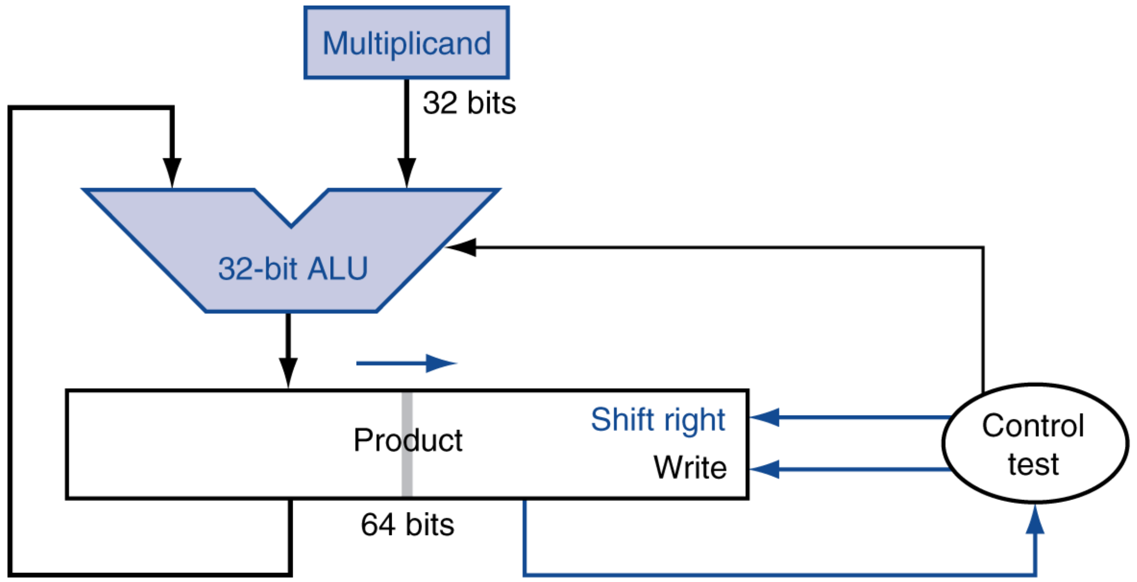
Multiplicand	Product
0010	0000 0011

Multiplicand	Product
0010	0000 0011
0010	0010 0011
0010	0001 0001
0010	0011 0001
0010	0001 1000
0010	0000 1100
0010	0000 0110


Start
Product0 = 1
1. Test Product0
Product0 = 0
1a. Add multiplicand to the left half of product & place the result in the left half of Product register
2. Shift the Product register right 1 bit.
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Optimized Multiplier

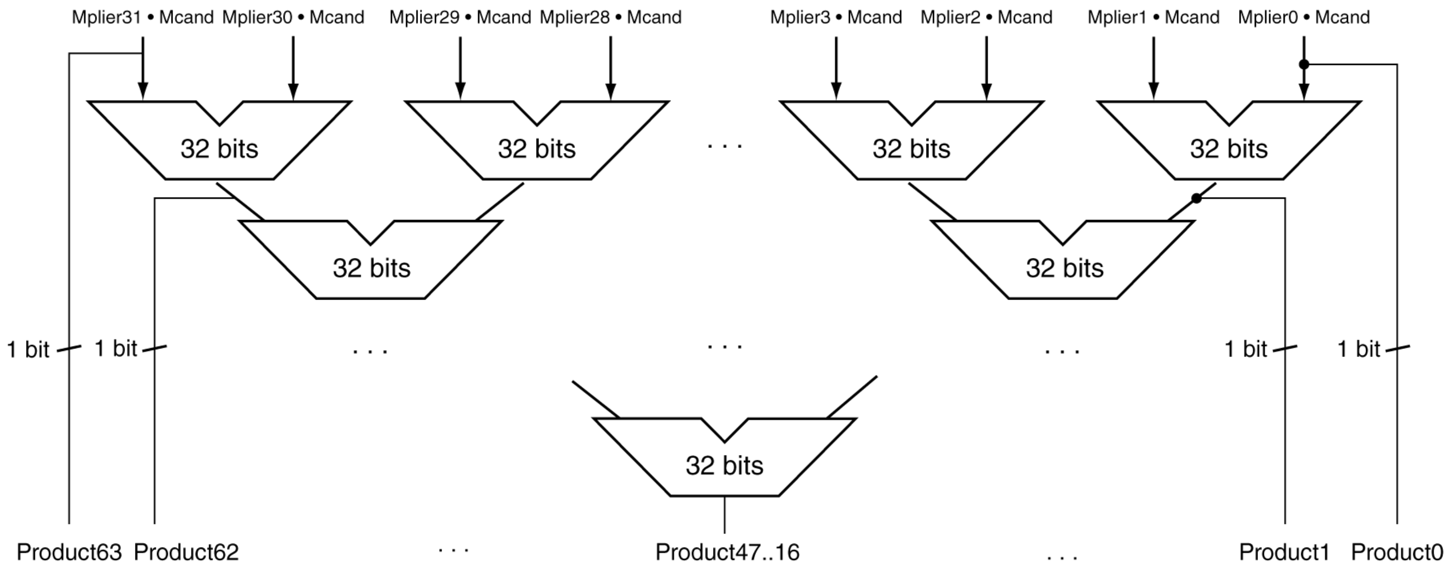
- Perform steps in parallel: add/shift



- One cycle per partial-product addition
  - That’s ok, if frequency of multiplications is low

Faster Multiplier

- Uses multiple adders
  - Cost/performance tradeoff

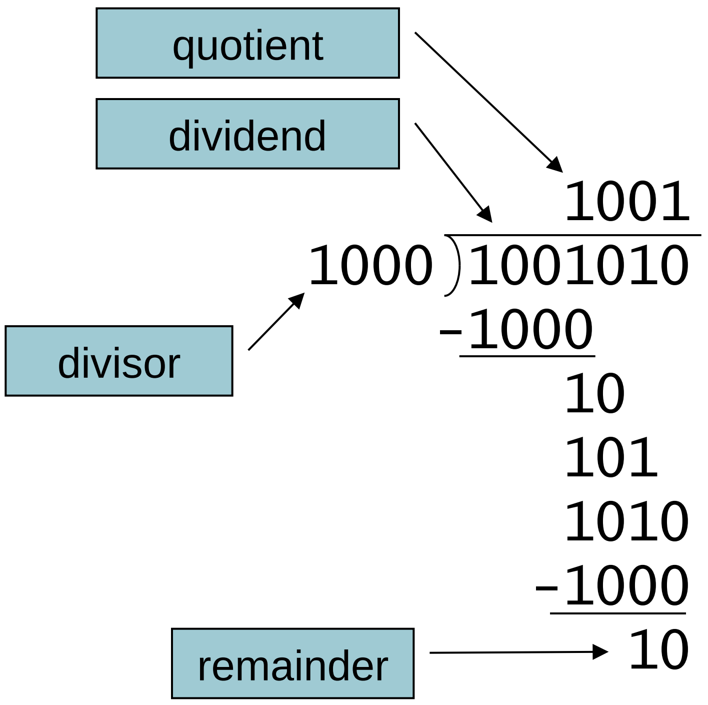


- Can be pipelined
  - Several multiplication performed in parallel

MIPS Multiplication

- Two 32-bit registers for product
  - HI: most-significant 32 bits
  - LO: least-significant 32-bits
- Instructions
  - `mult rs, rt  /  multu rs, rt`
    - 64-bit product in HI/LO
  - `mfhi rd  /  mflo rd`
    - Move from HI/LO to rd
    - Can test HI value to see if product overflows 32 bits
  - `mul rd, rs, rt`
    - Least-significant 32 bits of product –> rd

Division

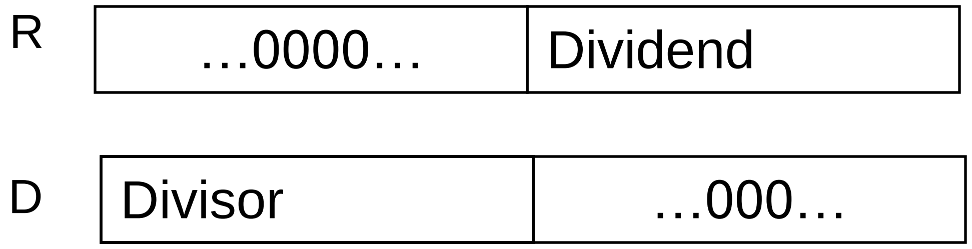


n-bit operands yield *n*-bit quotient and remainder

- 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

Registers

- Remainder register R (initially place dividend in R) (64 bits)
- Divisor register D (place divisor in left half of D) (64 bits)
  - D is twice the precision (64 bits for 32 bit division). We place it in the left half so we can start subtracting from the most significant bits of the dividend as we shift it to the right.
- Quotient register Q (32 bits)



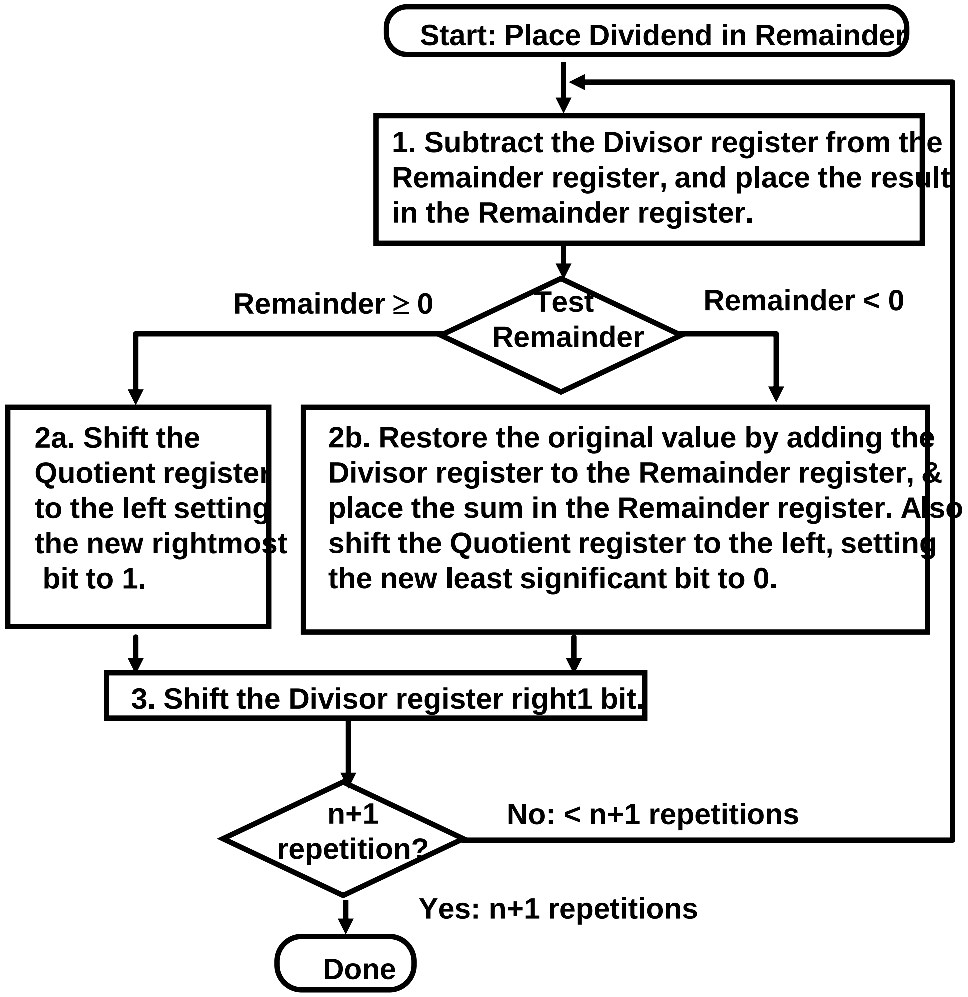
Divide Algorithm Version 1

- Takes n+1 steps for n-bit Quotient & Rem.

Place divisor in left half of divisor register then subtract entire D from entire R

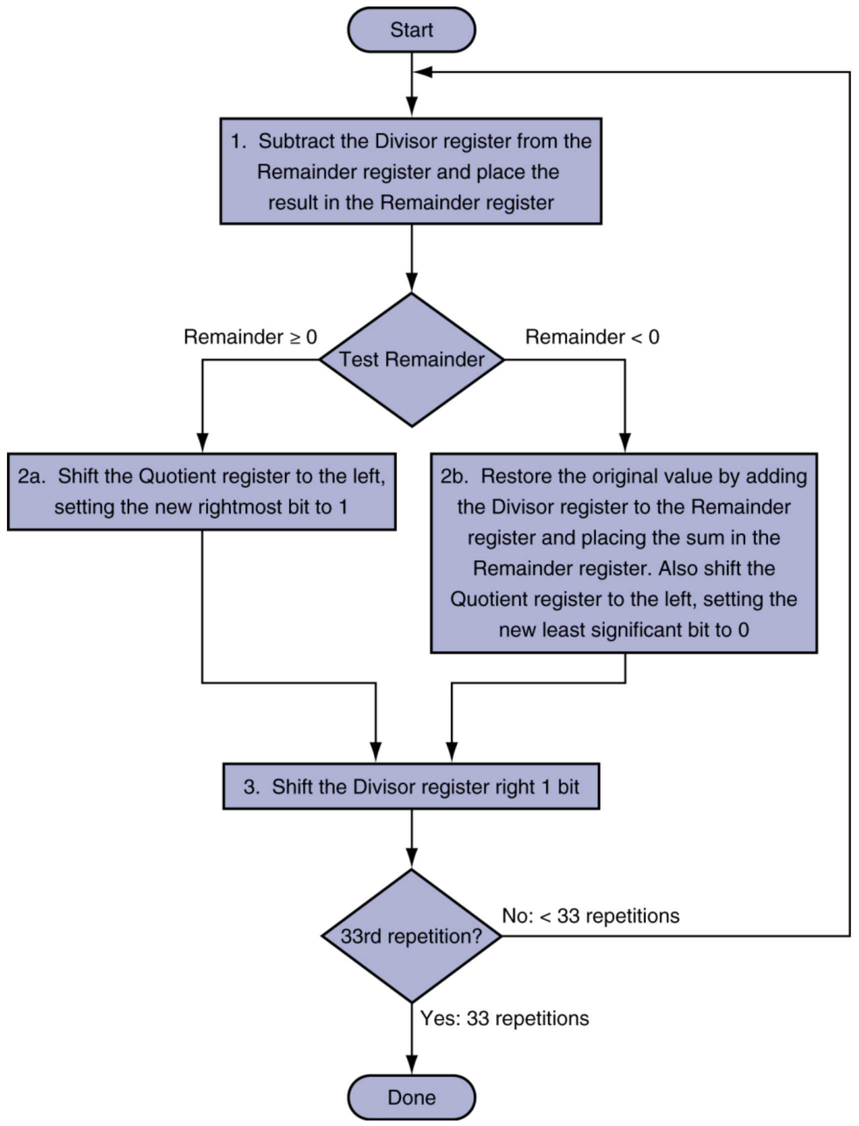
Right half of D will = 0 initially.

First subtraction will never result in a qutient bit of 1. Always 0.

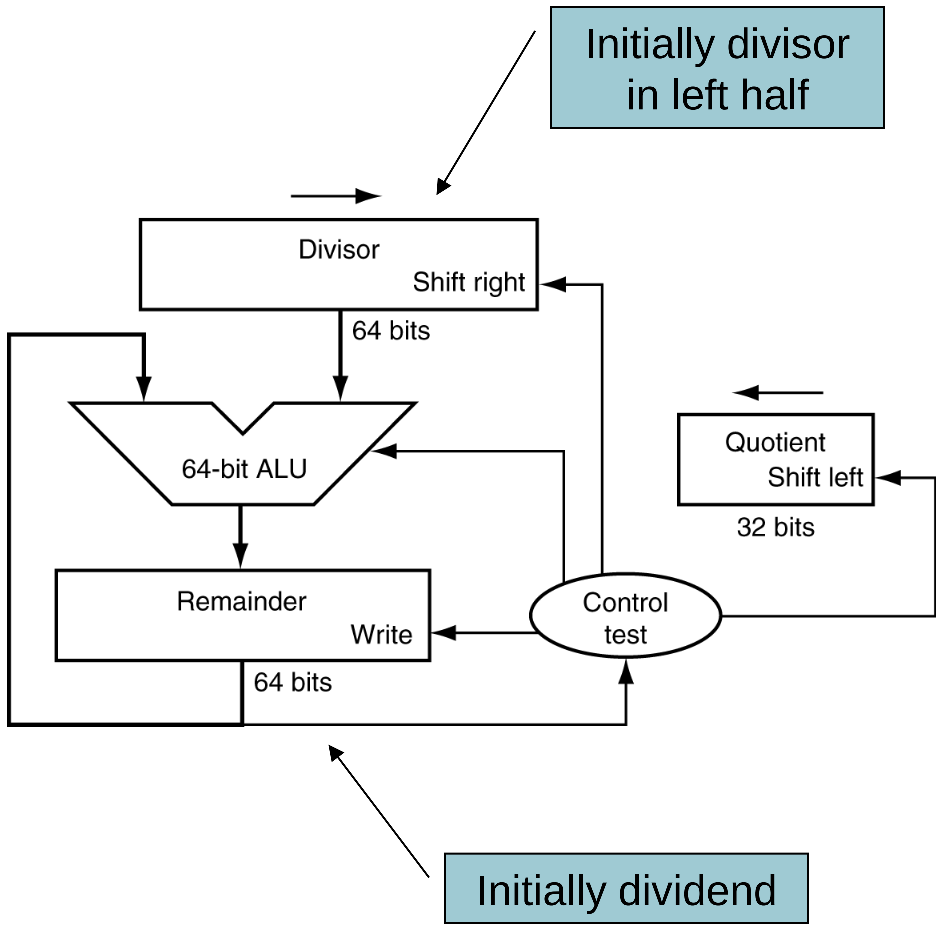


Start: Place Dividend in Remainder
1. Subtract the Divisor register from the Remainder register, and place the result in the Remainder register.
Test Remainder
Remainder ≥ 0
Remainder < 0
2a. Shift the Quotient register to the left setting the new rightmost bit to 1.
2b. Restore the original value by adding the Divisor register to the Remainder register, & place the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0.
3. Shift the Divisor register right1 bit.
n+1 repetition?
No: < n+1 repetitions
Yes: n+1 repetitions
Done

Division Hardware version 1



Start
1. Subtract the Divisor register from the Remainder register and place the result in the Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift the Quotient register to the left, setting the new rightmost bit to 1
2b. Restore the original value by adding the Divisor register to the Remainder register and placing the sum in the Remainder register. Also shift the Quotient register to the left, setting the new least significant bit to 0
3. Shift the Divisor register right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done



Example

7/2➔ quotient 3 remainder 1

iteration			divisor	Dividend in remainder
	Initial values	Q: 0000	D: 0010 0000	R: 0000 0111 –D = 1110 0000
1	1: R = R–D	Q: 0000	D: 0010 0000	R: 1110 0111 ← R – D < 0
	2b: +D, sl Q, 0	Q: 0000	D: 0010 0000	R: 0000 0111
	3: Shr D	Q: 0000	D: 0001 0000	R: 0000 0111 –D = 1111 0000
2	1: R = R–D	Q: 0000	D: 0001 0000	R: 1111 0111
	2b: +D, sl Q, 0	Q: 0000	D: 0001 0000	R: 0000 0111
	3: Shr D	Q: 0000	D: 0000 1000	R: 0000 0111 –D = 1111 1000
3	1: R = R–D	Q: 0000	D: 0000 1000	R: 1111 1111
	2b: +D, sl Q, 0	Q: 0000	D: 0000 1000	R: 0000 0111
	3: Shr D	Q: 0000	D: 0000 0100	R: 0000 0111 –D = 1111 1100
4	1: R = R–D	Q: 0000	D: 0000 0100	R: 0000 0011
	2a: sl Q, 1	Q: 0001	D: 0000 0100	R: 0000 0011
	3: Shr D	Q: 0001	D: 0000 0010	R: 0000 0011 –D = 1111 1110
5	1: R = R–D	Q: 0001	D: 0000 0010	R: 0000 0001
	2a: sl Q, 1	Q: 0011	D: 0000 0010	R: 0000 0001
	3: Shr D	Q: 0011	D: 0000 0001	R: 0000 0001

4 bits ➔ 5 iterations: first iteration will always generate Q = 0 bit, because we are subtracting 0-D

Observations on Divide Version 1

- 1/2 bits in divisor always 0
  ➔ 1/2 of 64-bit adder is wasted
  ➔ 1/2 of divisor is wasted
- 1st step cannot produce a 1 in quotient bit
  (otherwise, quotient would be too large for the register)

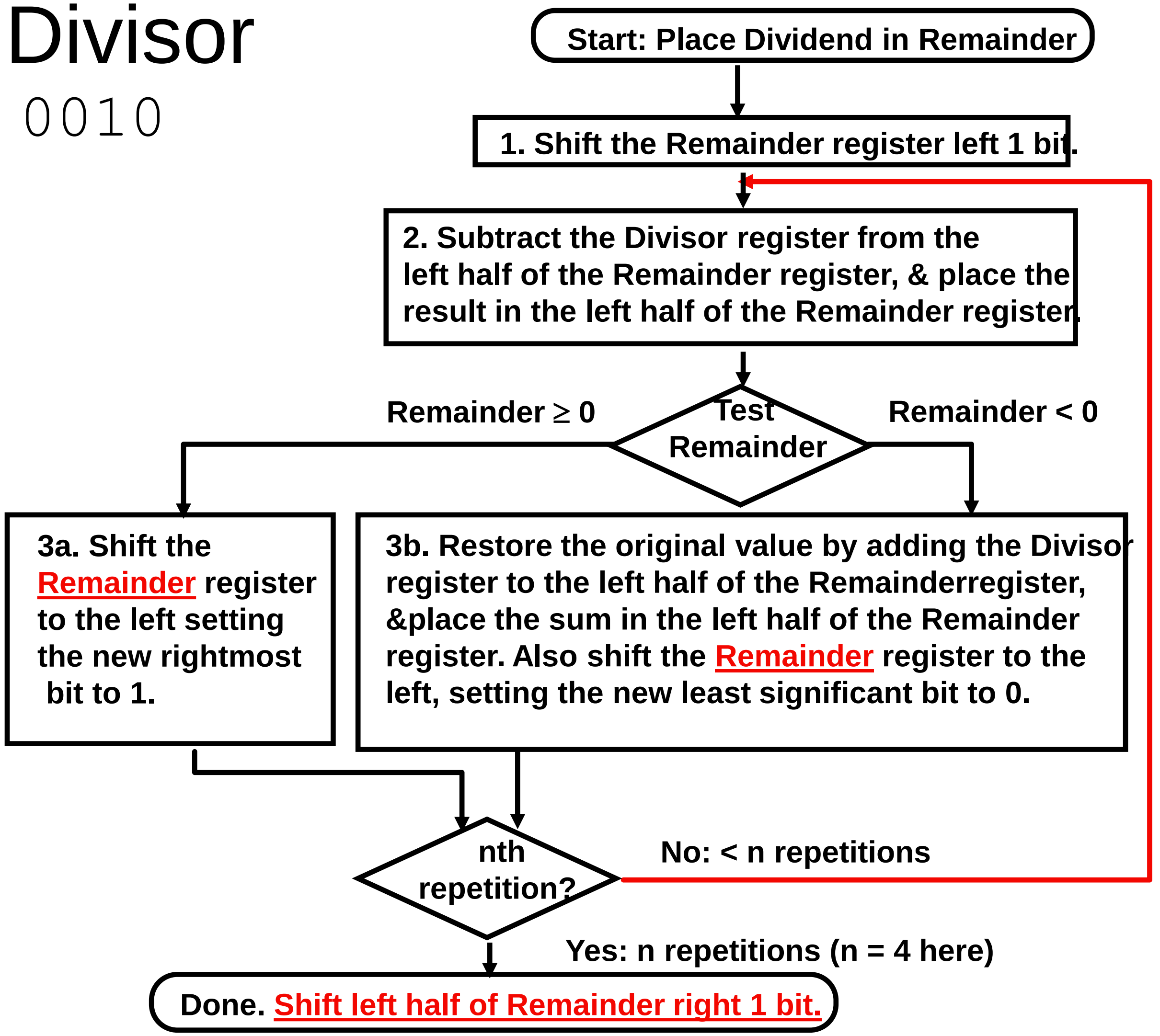
Improved on Divide Version 2

- Instead of shifting divisor to right, shift remainder to left?
- 1 step quotient bit always zero, then
  ➔ switch order to shift first and then subtract, can save 1 iteration
- Eliminate Quotient register by combining with Remainder as shifted left
  - Start by shifting the Remainder left as before.
  - Thereafter loop contains only two steps because the shifting of the Remainder register shifts both the remainder in the left half and the quotient in the right half
  - The consequence of combining the two registers together and the new order of the operations in the loop is that the remainder will be shifted left one time too many.
  - Thus the final correction step must shift back only the remainder in the left half of the register

Divide Algorithm Version 2

Remainder Divisor

```
0000 0111     0010
```


Start: Place Dividend in Remainder
1. Shift the Remainder register left 1 bit.
2. Subtract the Divisor register from the left half of the Remainder register, & place the result in the left half of the Remainder register.
Remainder ≥ 0
Test Remainder
Remainder < 0
3a. Shift the Remainder register to the left setting the new rightmost bit to 1.
3b. Restore the original value by adding the Divisor register to the left half of the Remainderregister, &place the sum in the left half of the Remainder register. Also shift the Remainder register to the left, setting the new least significant bit to 0.
nth repetition?
No: < n repetitions
Yes: n repetitions (n = 4 here)
Done. Shift left half of Remainder right 1 bit.

Division Example

	Divisor	Dividend in remainder	
	D: 0010	R: 0000 0111	Initialize
0: Shl R	D: 0010	R: 0000 1110	
1: R = R–D	D: 0010	R: 1110 1110	Iteration 1
2b: +D, sl R, 0	D: 0010	R: 0001 1100	
1: R = R–D	D: 0010	R: 1111 1100	Iteration 2
2b: +D, sl R, 0	D: 0010	R: 0011 1000	
1: R = R–D	D: 0010	R: 0001 1000	Iteration 3
2a: sl R, 1	D: 0010	R: 0011 0001	
1: R = R–D	D: 0010	R: 0001 0001	Iteration 4
2a: sl R, 1	D: 0010	R: 0010 0011	
Shr R(rh)	D: 0010	R: 0001 0011	

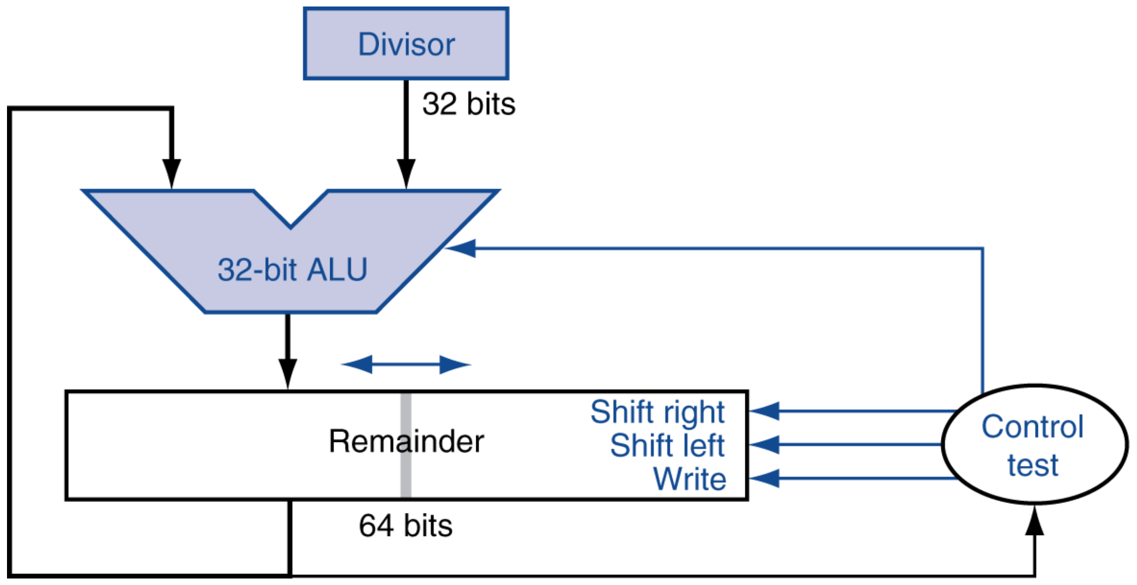
4 iterations

Quotient

Final correction

7/2➔ quotient 3 remainder 1

Optimized Divider



- One cycle per partial-remainder subtraction
- Looks a lot like a multiplier!
  - Same hardware can be used for both

Faster Division

- Can’t use parallel hardware as in multiplier
  - Subtraction is conditional on sign of remainder
- Faster dividers (e.g. SRT division) generate multiple quotient bits per step
  - Still require multiple steps

MIPS Division

- Use HI/LO registers for result
  - HI: 32-bit remainder
  - LO: 32-bit quotient
- Instructions
  - `div rs, rt  /  divu rs, rt`
  - No overflow or divide-by-0 checking
    - Software must perform checks if required
  - Use `mfhi`, `mflo` to access result

Real number representation in binary

- Usual representation (just like decimal numbers):
  - Use negative powers of the radix after the fractional point: radix = 10 for decimal and 2 for binary
  - Decimal $0.55 = 5 \times 10^{-1} + 5 \times 10^{-2}$
  - Binary $0.101 = 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$
- With fixed precision (e.g., 32 bits), how do we represent these real numbers in computers?

Floating Point numbers

- Representation for non-integral numbers
  - Including very small and very large numbers
- Like scientific notation
  - -2.34×10^{56} ← normalized
  - $+0.002 \times 10^{-4}$ ← not normalized
  - $+987.02 \times 10^{9}$
- In binary
  - $\pm 1.xxxxxxx_2 \times 2^{yyyy}$
- Types `float` and `double` in C

Floating Point Standard

- Defined by IEEE Std 754-1985
- Developed in response to divergence of representations
  - Portability issues for scientific code
- Now almost universally adopted
- Two representations
  - Single precision (32-bit)
  - Double precision (64-bit)

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)
  - Sign magnitude representation for whole number ➔ two zeros: ±0.

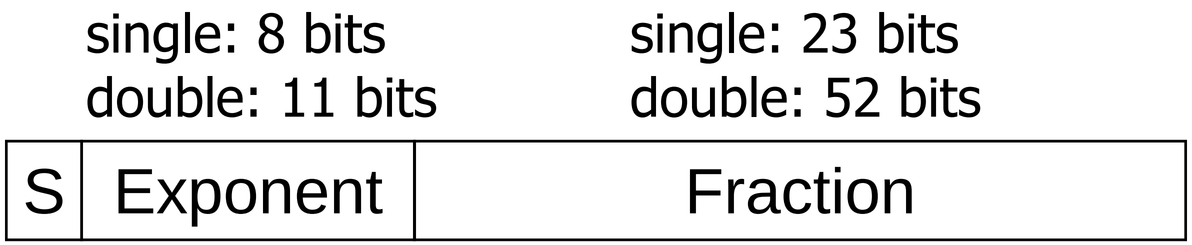
IEEE Floating-Point Format

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

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

- Normalized significand: 1.0 ≤ |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

IEEE Floating-Point Format



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

- Exponent: excess representation: actual exponent + Bias
  - Single: Bias = $(0111\ 1111)_2 = 2^7 - 1 = 127;$
  - Double: Bias = $(011\ 1111\ 1111)_2 = 2^{10} - 1 = 1023$
    - For single precision, exponent of 0111 1111 corresponds to 0 after bias is subtracted: 127-127 = 0.
    - For single precision, exponent of 1000 0000 corresponds to +1 after bias is subtracted: 128-127 = 1.
  - Ensures exponent is unsigned
  - Can do magnitude comparison without converting.
    - 1000 0000 > 0111 1111

Single-Precision Range

- Exponents 00000000 and 11111111 reserved as special cases
- Smallest value
  - Exponent: 00000001
    $\Rightarrow$ actual exponent = 1 – 127 = –126
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-126} \approx \pm 1.2 \times 10^{-38}$
- Largest value
  - exponent: 11111110
    $\Rightarrow$ actual exponent = 254 – 127 = +127
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+127} \approx \pm 3.4 \times 10^{+38}$

Double-Precision Range

- Exponents 0000…00 and 1111…11 reserved
- Smallest value
  - Exponent: 00000000001
    $\Rightarrow$ actual exponent = 1 – 1023 = –1022
  - Fraction: 000…00 $\Rightarrow$ significand = 1.0
  - $\pm 1.0 \times 2^{-1022} \approx \pm 2.2 \times 10^{-308}$
- Largest value
  - Exponent: 11111111110
    $\Rightarrow$ actual exponent = 2046 – 1023 = +1023
  - Fraction: 111…11 $\Rightarrow$ significand $\approx$ 2.0
  - $\pm 2.0 \times 2^{+1023} \approx \pm 1.8 \times 10^{+308}$

Floating-Point Precision

- Relative precision
  - all fraction bits are significant
  - Single: approx 2^{-23}
    - Equivalent to $23 \times \log_{10}2 \approx 23 \times 0.3 \approx 6$ decimal digits of precision
  - Double: approx 2^{-52}
    - Equivalent to $52 \times \log_{10}2 \approx 52 \times 0.3 \approx 16$ decimal digits of precision

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 Example

- What number is represented by the single-precision float

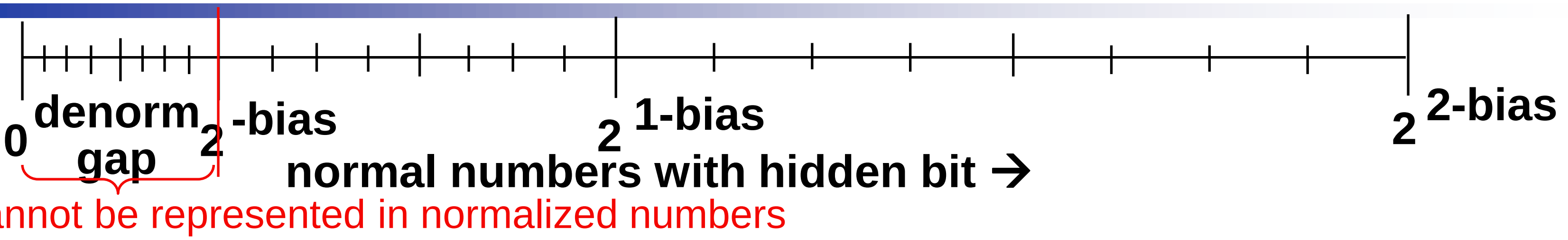
  11000000101000…00

  - S = 1
  - Fraction = $01000\ldots00_2$
  - Exponent = 10000001_2 = 129
- $x = (-1)^1 \times (1 + .01_2) \times 2^{(129 - 127)}$

  $= (-1) \times 1.25 \times 2^2$

  $= -5.0$

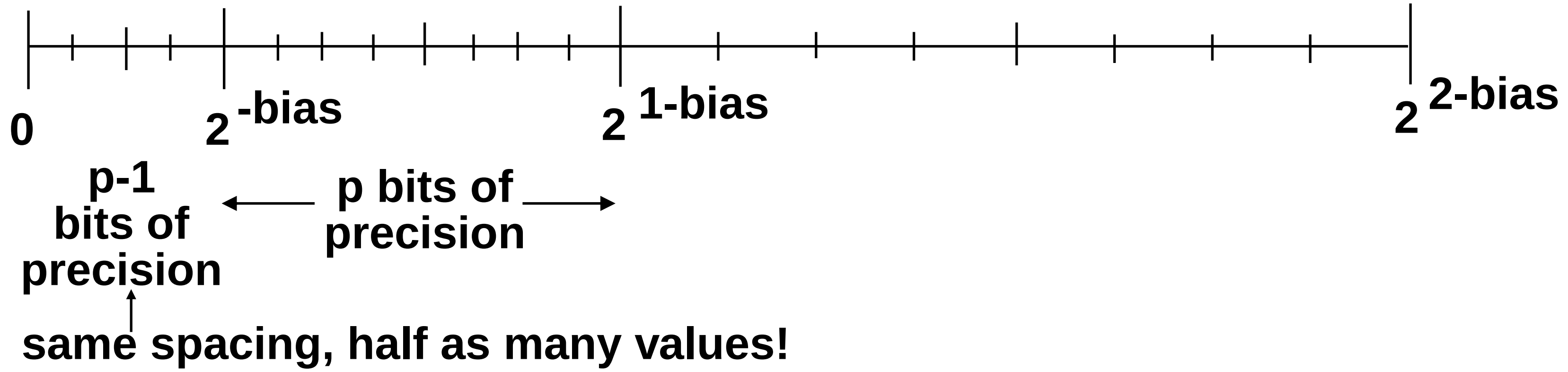
"Denormalized" Numbers



0
denorm gap
2^{-bias}
2^{1-bias}
2^{2-bias}
normal numbers with hidden bit →
Cannot be represented in normalized numbers

The gap between 0 and the next representable number is much larger than the gaps between nearby representable numbers.

IEEE standard uses denormalized numbers to fill in the gap, making the distances between numbers near 0 more alike.



NOTE: PDP-11, VAX cannot represent subnormal numbers (because it didn't use IEEE standard for FP numbers). These machines underflow to zero instead.

Denormalized Numbers

- Exponent = 000...0 $\Rightarrow$ hidden bit before fractional point is 0

$$x = (-1)^S \times (0 + \text{Fraction}) \times 2^{-\text{Bias}}$$

- Smaller than normalized numbers
  - allow for gradual underflow, with diminishing precision

- Denormalized with fraction = 000...0

$$x = (-1)^S \times (0 + 0) \times 2^{-\text{Bias}} = \pm 0.0$$

Two representations of 0.0!

Infinities and NaNs

- Exponent = 111...1, Fraction = 000...0
  - ±Infinity
  - Can be used in subsequent calculations, avoiding need for overflow check
- Exponent = 111...1, Fraction ≠ 000...0
  - Not-a-Number (NaN)
  - Indicates illegal or undefined result
    - e.g., 0.0 / 0.0
  - Can be used in subsequent calculations

Floating-Point Addition

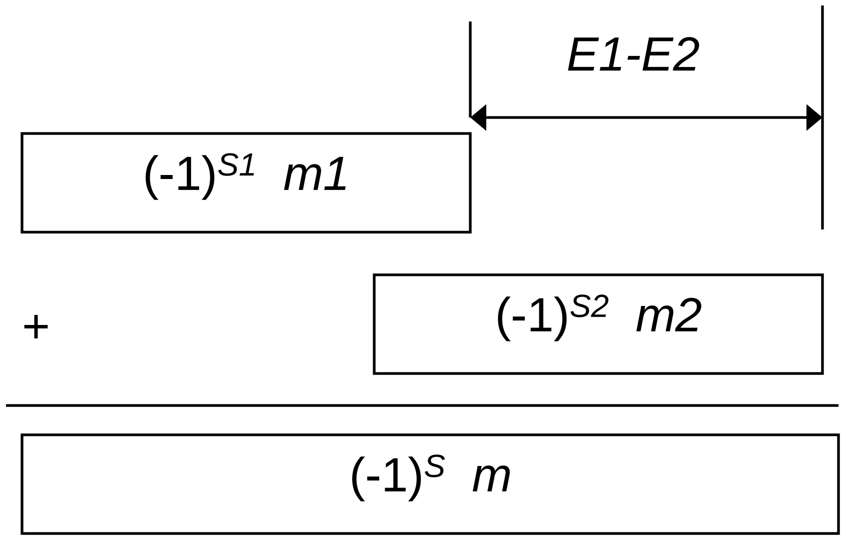
- Consider a 4-digit decimal example
  - $9.999 \times 10^{1} + 1.610 \times 10^{-1}$
- 1. Align decimal points
  - Shift number with smaller exponent
  - $9.999 \times 10^{1} + 0.016 \times 10^{1}$
- 2. Add significands
  - $9.999 \times 10^{1} + 0.016 \times 10^{1} = 10.015 \times 10^{1}$
- 3. Normalize result & check for over/underflow
  - 1.0015×10^{2}
- 4. Round and renormalize if necessary
  - 1.002×10^{2}

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) $= 0.0625$

FP addition

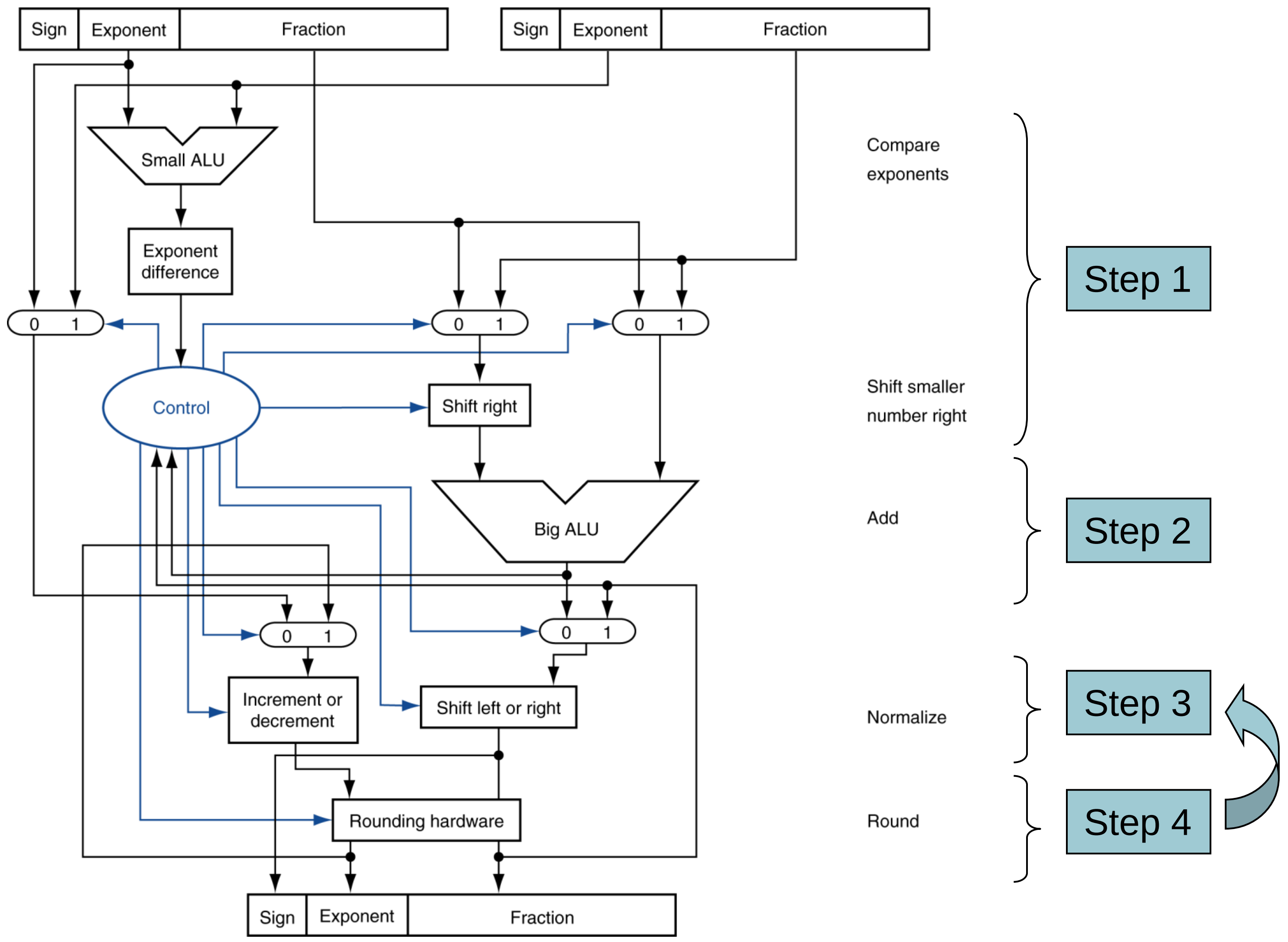
- Operands
  - $(-1)^{S1}\ m1\ 2^{E1}$
  - $(-1)^{S2}\ m2\ 2^{E2}$
  - Assume $E1 \geq E2$
- Exact Result
  - $(-1)^{S}\ m\ 2^{E}$
  - Sign *s,* Mantissa *m:*
    - Result of signed align & add
  - Exponent *E:* $E1 - E2$
- Fixing
  - Shift *m* right, increment *E* if $m \geq 2$
  - Shift *m* left *k* positions, decrement *E* by *k* if $m < 1$
  - Overflow if *E* out of range
  - Round *m* to fit significand precision



FP Adder Hardware

- Much more complex than integer adder
- Doing it in one clock cycle would take too long
  - Much longer than integer operations
  - Slower clock would penalize all instructions
- FP adder usually takes several cycles
  - Can be pipelined

FP Adder Hardware



FP multiplication

- Operands
  - $(-1)^{S1}\, m1\ 2^{E1}$
  - $(-1)^{S2}\, m2\ 2^{E2}$
- Exact Result
  - $(-1)^{S}\, m\ 2^{E}$
  - Sign *s*: $s1 \neq s2 \rightarrow (s = 1 \text{ if } s1 \neq s2 \text{ and } s = 0 \text{ otherwise})$
  - Mantissa *m*: $m1 * m2$
  - Exponent *E*: $E1 + E2$
- Fixing
  - Overflow if *E* out of range
  - Round *m* to fit significand precision
- Implementation
  - Biggest chore is multiplying mantissas

Floating-Point Multiplication

- Consider a 4-digit decimal example
  - $1.110 \times 10^{10} \times 9.200 \times 10^{-5}$
- 1. Add exponents
  - For biased exponents, subtract bias from sum
  - New exponent = 10 + –5 = 5
- 2. Multiply significands
  - $1.110 \times 9.200 = 10.212 \Rightarrow 10.212 \times 10^{5}$
- 3. Normalize result & check for over/underflow
  - 1.0212×10^{6}
- 4. Round and renormalize if necessary
  - 1.021×10^{6}
- 5. Determine sign of result from signs of operands
  - $+1.021 \times 10^{6}$

Floating-Point Multiplication

- Now consider a 4-digit binary example
  - $1.000_2 \times 2^{-1} \times -1.110_2 \times 2^{-2}$ (0.5 × –0.4375)
- 1. Add exponents
  - Unbiased: –1 + –2 = –3
  - Biased: (–1 + 127) + (–2 + 127) = –3 + 254 – 127 = –3 + 127
- 2. Multiply significands
  - $1.000_2 \times 1.110_2 = 1.110_2 \Rightarrow 1.110_2 \times 2^{-3}$
- 3. Normalize result & check for over/underflow
  - $1.110_2 \times 2^{-3}$ (no change) with no over/underflow
- 4. Round and renormalize if necessary
  - $1.110_2 \times 2^{-3}$ (no change)
- 5. Determine sign: +ve × –ve ⇒ –ve
  - $-1.110_2 \times 2^{-3} = -0.21875$

FP Arithmetic Hardware

- FP multiplier is of similar complexity to FP adder
  - But uses a multiplier for significands instead of an adder
- FP arithmetic hardware usually does
  - Addition, subtraction, multiplication, division, reciprocal, square-root
  - FP $\leftrightarrow$ integer conversion
- Operations usually takes several cycles
  - Can be pipelined

FP Instructions in MIPS

- FP hardware is coprocessor 1
  - Adjunct processor that extends the ISA
- Separate FP registers
  - 32 single-precision: $f0, $f1, … $f31
  - Paired for double-precision: $f0/$f1, $f2/$f3, …
    - Release 2 of MIPS ISA supports 32 × 64-bit FP reg's
- 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
  - `lwc1`, `ldc1`, `swc1`, `sdc1`
    - e.g., `ldc1 $f8, 32($sp)`

FP Instructions in MIPS

- Single-precision arithmetic
  - `add.s`, `sub.s`, `mul.s`, div.s
    - e.g., `add.s $f0, $f1, $f6`
- Double-precision arithmetic
  - `add.d`, `sub.d`, `mul.d`, `div.d`
    - e.g., `mul.d $f4, $f4, $f6`
- Single- and double-precision comparison
  - `c.`*`xx`*`.s`, `c.`*`xx`*`.d` (*xx* is eq, `lt`, `le`, …)
  - Sets or clears FP condition-code bit
    - e.g. `c.lt.s $f3, $f4`
- Branch on FP condition code true or false
  - `bc1t`, `bc1f`
    - e.g., `bc1t TargetLabel`

FP Example: °F to °C

- C code:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - `fahr` in $f12, result in $f0, literals in global memory space

- Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp)
     lwc2  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

Accurate Arithmetic

- IEEE Std 754 specifies additional rounding control
  - Extra bits of precision (guard, round, sticky)
  - Choice of rounding modes
  - Allows programmer to fine-tune numerical behavior of a computation
- Not all FP units implement all options
  - Most programming languages and FP libraries just use defaults
- Trade-off between hardware complexity, performance, and market requirements

Interpretation of Data

The BIG Picture

- Bits have no inherent meaning
  - Interpretation depends on the instructions applied
- Computer representations of numbers
  - Finite range and precision
  - Need to account for this in programs

Associativity

- Parallel programs may interleave operations in unexpected orders
  - Assumptions of associativity may fail

		(x+y)+z	x+(y+z)
x	-1.50E+38		-1.50E+38
y	1.50E+38	0.00E+00	
z	1.0	1.0	1.50E+38
		1.00E+00	0.00E+00

- Need to validate parallel programs under varying degrees of parallelism

Concluding Remarks

- ISAs support arithmetic
  - Signed and unsigned integers
  - Floating-point approximation to reals
- Bounded range and precision
  - Operations can overflow and underflow
- MIPS ISA
  - Core instructions: 54 most frequently used
    - 100% of SPECINT, 97% of SPECFP
  - Other instructions: less frequent