CS4100: 計算機結構

Computer Arithmetic

國立清華大學資訊工程學系
九十三學年度第一學期

Adapted from class notes of D. Patterson

Outline

Problem: Designing MIPS ALU

- Requirements: must support the following arithmetic and logic operations
  - add, sub: two's complement adder/subtractor with overflow detection
  - and, or, nor : logical AND, logical OR, logical NOR
  - slt (set on less than): two's complement adder with inverter, check sign bit of result

Functional Specification

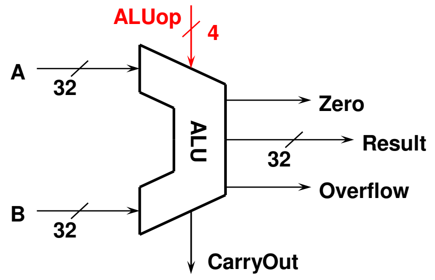


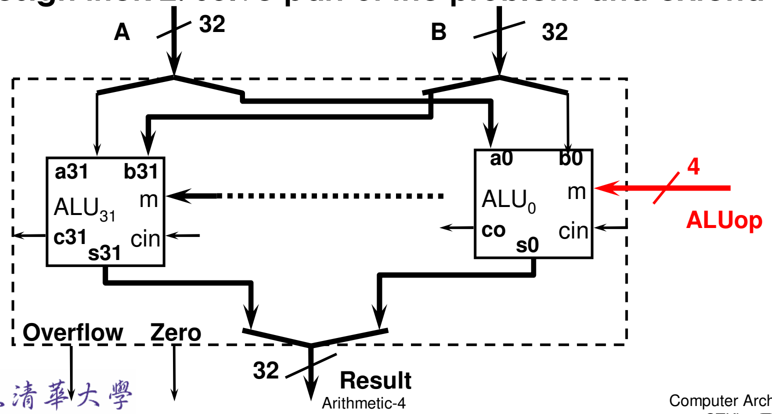
Fig. B.5.14

ALU Control (ALUop)	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-on-less-than
1100	nor

Fig. B.5.13

A Bit-slice ALU

- Design trick 1: divide and conquer
  - Break the problem into simpler problems, solve them and glue together the solution
- Design trick 2: solve part of the problem and extend



A 1-bit ALU

- Design trick 3: take pieces you know (or can imagine) and try to put them together

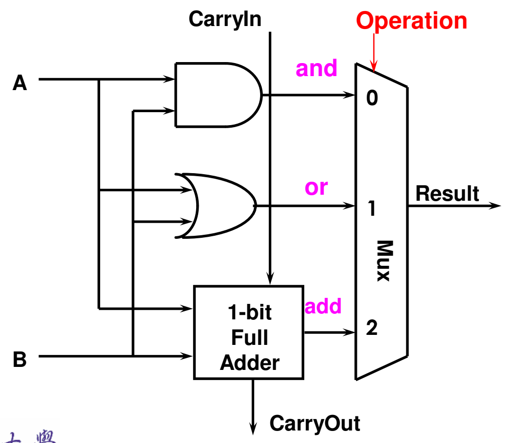
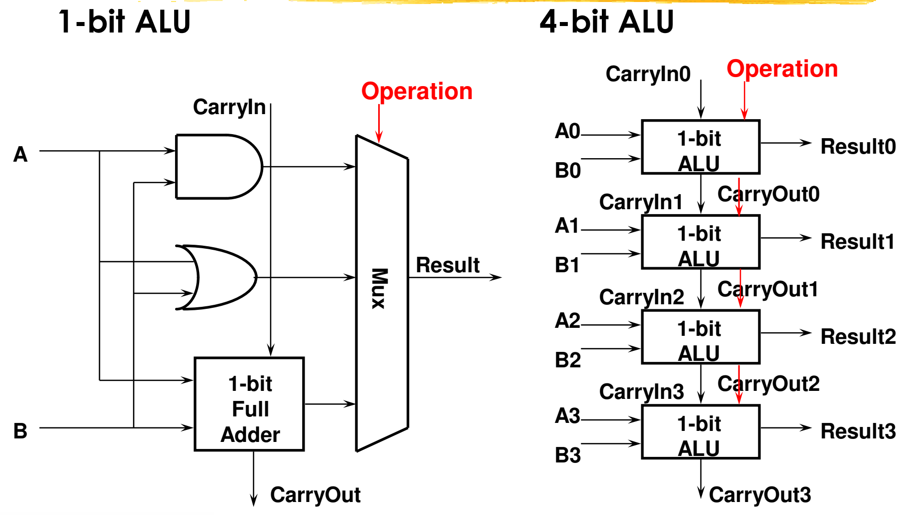


Fig. B.5.6

A 4-bit ALU



How about Subtraction?

- 2's complement: take inverse of every bit and add 1 (at c_{in} of first stage)
  - A + B' + 1 = A + (B' + 1) = A + (-B) = A - B
  - Bit-wise inverse of B is B'

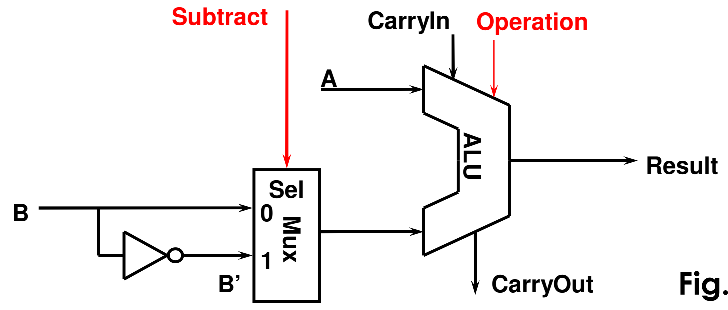
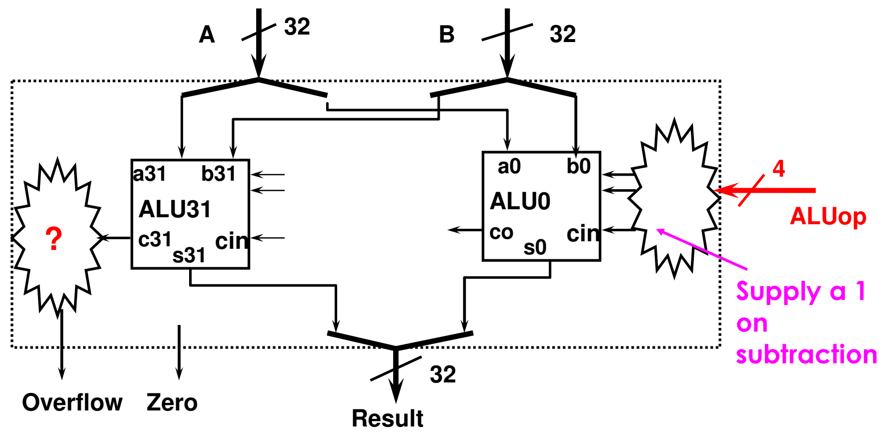


Fig. B.5.8

Revised Diagram

- LSB and MSB need to do a little extra



Nor Operation

- A nor B = (not A) and (not B)

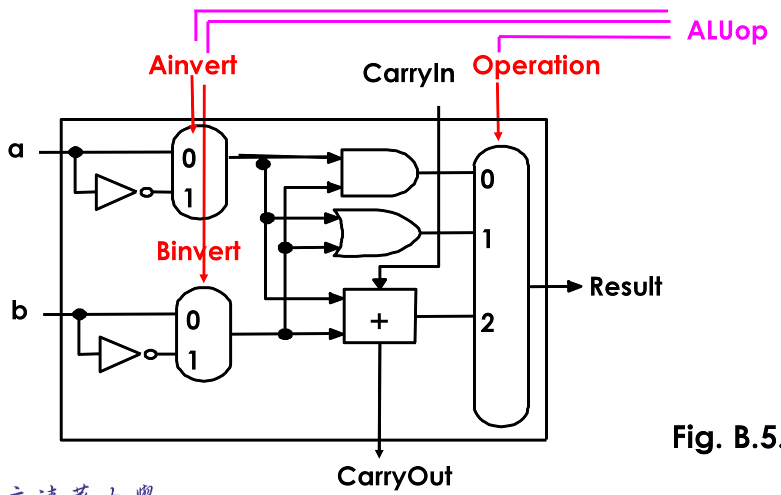


Fig. B.5.9

Overflow

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
		-8	1000

```
Ex:  7  +  3  =  10   but ...       - 4  -  5  =  - 9   but  ...
     0   1   1   1                     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
```

Overflow Detection

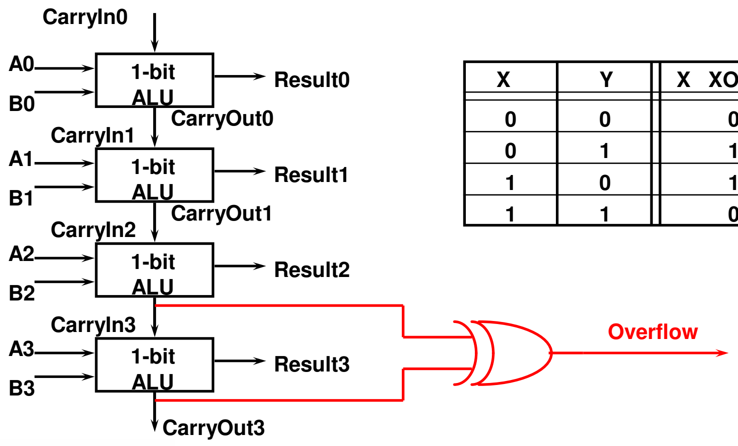
- Overflow: result too big/small to represent
  - $-8 \leq$ 4-bit binary number ≤ 7
  - When adding operands with different signs, overflow cannot occur!
  - Overflow occurs when adding:
    - 2 positive numbers and the sum is negative
    - 2 negative numbers and the sum is positive

    => sign bit is set with the value of the result
  - Overflow if: Carry into MSB $\neq$ Carry out of MSB

```
[0   1]  1   1                  [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
```

Overflow Detection Logic

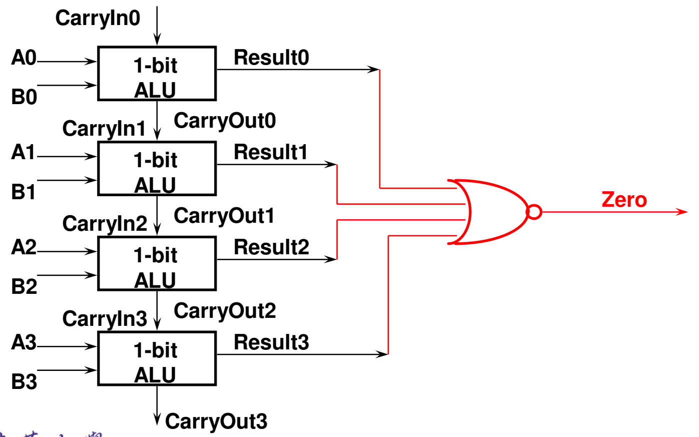
- Overflow = CarryIn[N-1] XOR CarryOut[N-1]



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

Zero Detection Logic

- Zero Detection Logic is a one BIG NOR gate



Set on Less Than (I)

- 1-bit in ALU

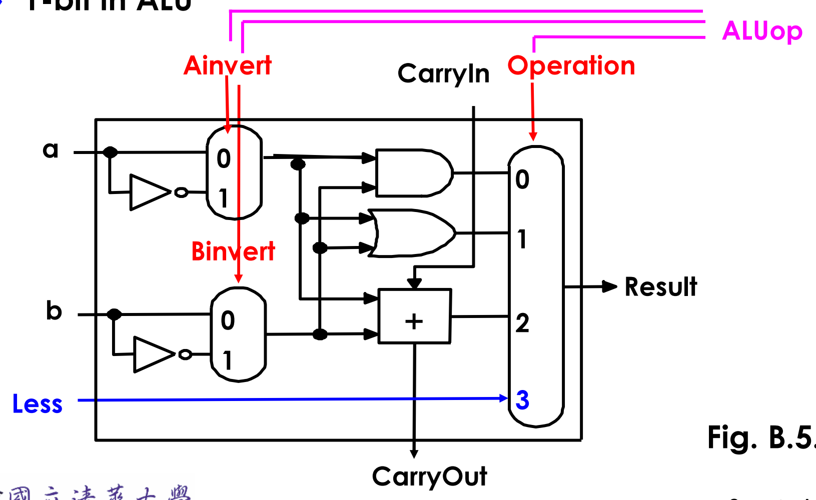


Fig. B.5.10a

Set on Less than (II)

- Sign bit in ALU

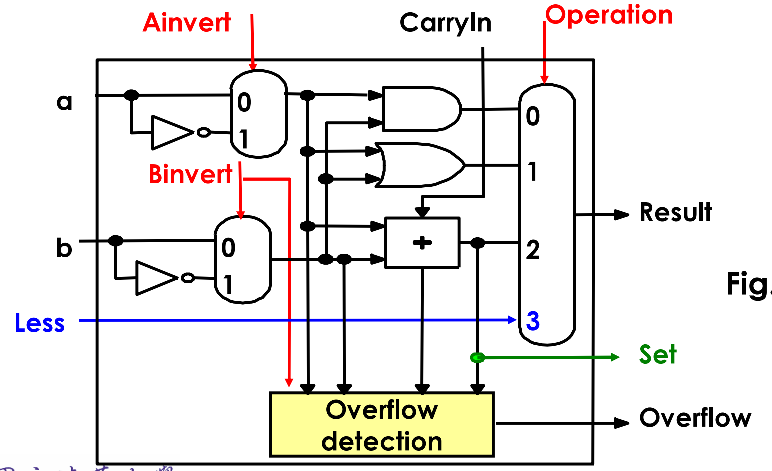
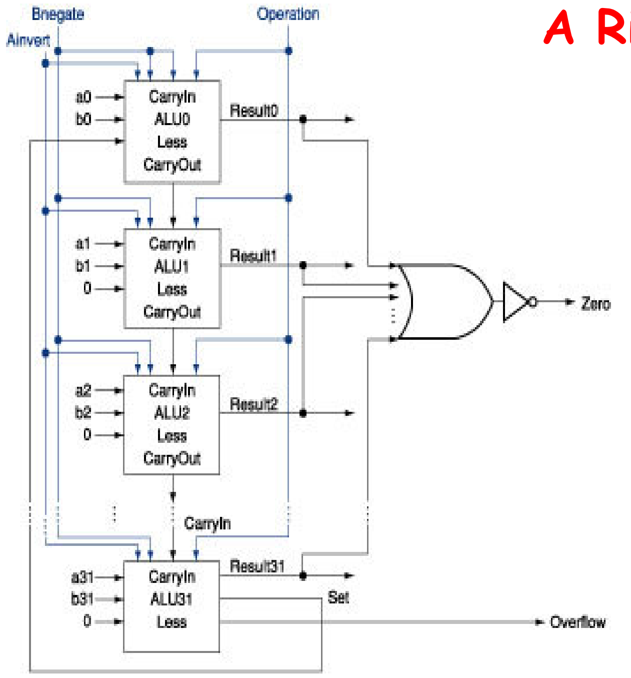


Fig. B.5.10b

A Ripple Carry Adder

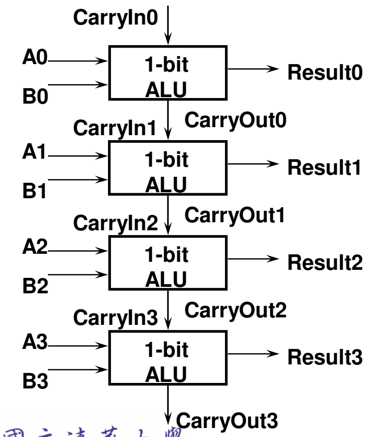


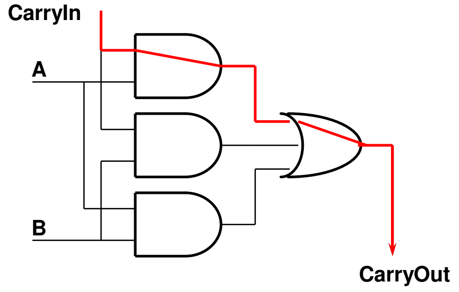
ALUop	Function
0000	and
0001	or
0010	add
0110	subtract
0111	set-less-than
1100	nor

Fig. B.5.12

Problems with Ripple Carry Adder

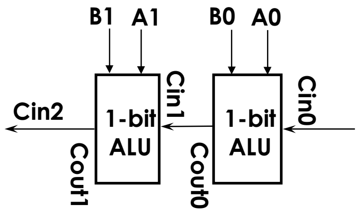
- Carry bit may have to propagate from LSB to MSB => worst case delay: N-stage delay





Design Trick: look for parallelism and throw hardware at it

Carry Lookahead: Theory (I) (Appendix B.6)



- CarryOut=(B*CarryIn)+(A*CarryIn)+(A*B)
  - Cin2=Cout1= (B1 * Cin1)+(A1 * Cin1)+ (A1 * B1)
  - Cin1=Cout0= (B0 * Cin0)+(A0 * Cin0)+ (A0 * B0)
- Substituting Cin1 into Cin2:
  - Cin2=(A1*A0*B0)+(A1*A0*Cin0)+(A1*B0*Cin0)
    +(B1*A0*B0)+(B1*A0*Cin0)+(B1*B0*Cin0)
    +(A1*B1)

Carry Lookahead: Theory (II)

- Now define two new terms:
  - Generate Carry at Bit i: gi = Ai * Bi
  - Propagate Carry via Bit i: pi = Ai + Bi
- We can rewrite:
  - Cin1=g0+(p0*Cin0)
  - Cin2=g1+(p1*g0)+(p1*p0*Cin0)
  - Cin3=g2+(p2*g1)+(p2*p1*g0)+(p2*p1*p0*Cin0)
- Carry going into bit 3 is 1 if
  - We generate a carry at bit 2 (g2)
  - Or we generate a carry at bit 1 (g1) and bit 2 allows it to propagate (p2 * g1)
  - Or we generate a carry at bit 0 (g0) and bit 1 as well as bit 2 allows it to propagate .....

A Plumbing Analogy for Carry Lookahead (1, 2, 4 bits)

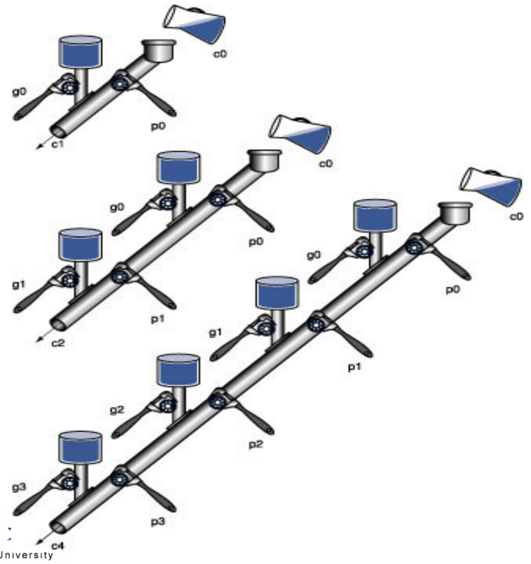
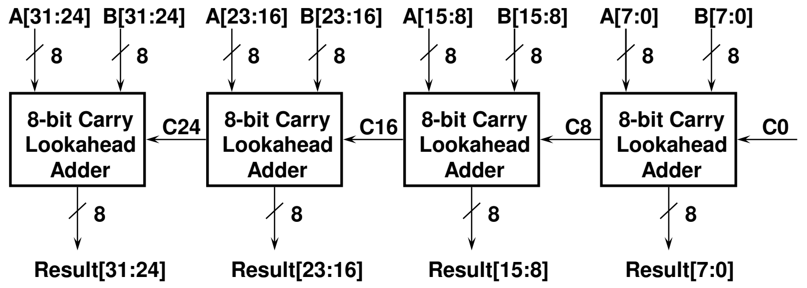


Fig. B.6.1

Cascaded Carry Lookahead

- Expensive to build a "full" carry lookahead adder
  - Just imagine length of the equation for Cin31
- Common practices:
  - Connects several N-bit lookahead adders to form a big one



A Plumbing Analogy for Carry Lookahead (Next Level P0 and G0)

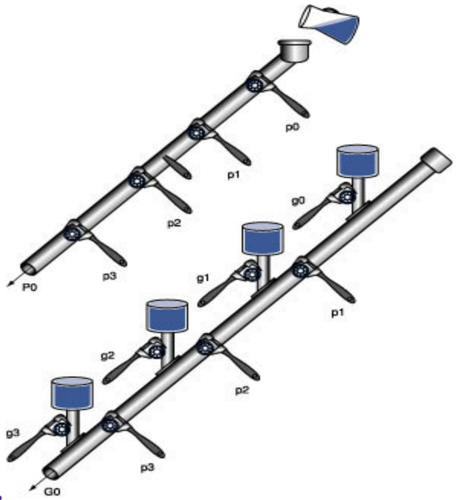
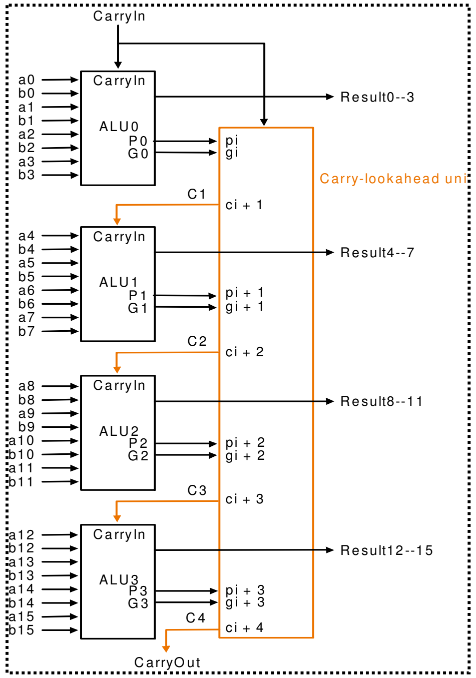


Fig. B.6.2

A Carry Lookahead Adder



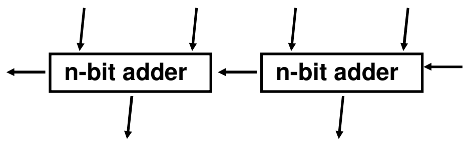
A	B	Cout	
0	0	0	kill
0	1	Cin	propagate
1	0	Cin	propagate
1	1	1	generate

G = A * B
P = A + B

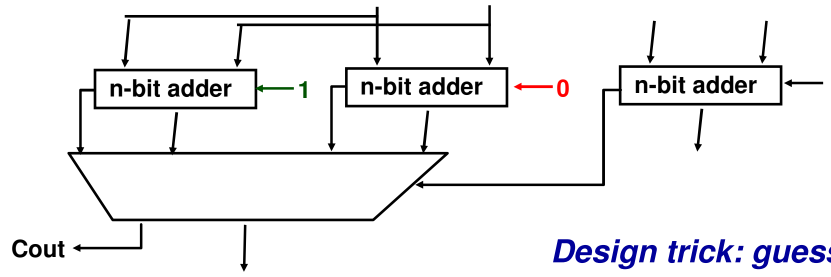
Fig. B.6.3

Carry-select Adder

CP(2n) = 2*CP(n)



CP(2n) = CP(n) + CP(mux)



Design trick: guess

Outline

- Signed and unsigned numbers (Sec. 3.2)
- Addition and subtraction (Sec. 3.3)
- Constructing an arithmetic logic unit (Appendix B.5, B6)
- Multiplication (Sec. 3.4 ,CD: 3.23 In More Depth)
- Division (Sec. 3.5)
- Floating point (Sec. 3.6)

Multiplication in MIPS

```
mul $t1, $t2        # t1 * t2
```

- No destination register: product could be ~2^{64}; need two special registers to hold it
- 3-step process:

```
  $t1   01111111111111111111111111111111
X $t2   01000000000000000000000000000000
```

00011111111111111111111111111111 11000000000000000000000000000000

Hi Lo

```
mfhi  $t3      $t3   00011111111111111111111111111111

mflo  $t4      $t4   11000000000000000000000000000000
```

Division in MIPS

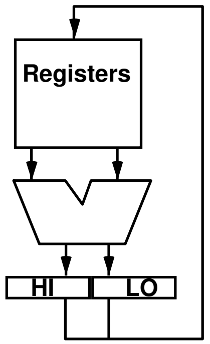
```
div $t1, $t2      # t1 / t2
```

- Quotient stored in Lo, remainder in Hi
```
mflo $t3      #copy quotient to t3
mfhi $t4      #copy remainder to t4
```
- 3-step process

- Unsigned multiplication and division:
```
mulu $t1, $t2      # t1 * t2
divu $t1, $t2      # t1 / t2
```
  - Just like mul, div, except now interpret t1, t2 as unsigned integers instead of signed
  - Answers are also unsigned, use mfhi, mflo to access

MIPS Multiply/Divide Summary

- Start multiply, divide
  - MULT rs, rt   HI-LO = rs × rt // 64-bit signed
  - MULTU rs, rt   HI-LO = rs × st // 64-bit unsigned
  - DIV rs, rt LO = rs ÷ rt; HI = rs mod rt
  - DIVU rs, rt
- Move result from multiply, divide
  - MFHI rd   rd = HI
  - MFLO rd   rd = LO
- Move to HI or LO
  - MTHI rd   HI = rd
  - MTLO rd   LO = rd



Unsigned Multiply

- Paper and pencil example (unsigned):

```
Multiplicand        1000ten
Multiplier        X  1001ten
                     1000
                    0000
                   0000
                  1000
Product           01001000ten
```

- 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)
- 2 versions of multiply hardware and algorithm

Unisigned Multiplier (Ver. 1)

- 64-bit *multiplicand register* (with 32-bit multiplicand at right half), 64-bit ALU, 64-bit *product register*, 32-bit *multiplier register*

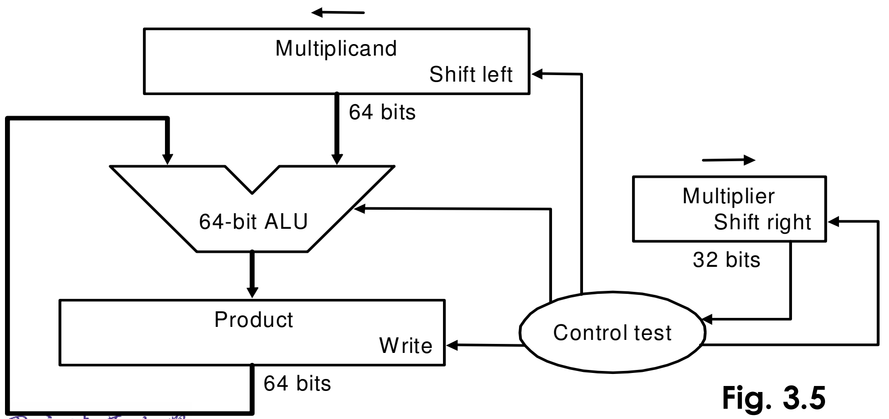


Fig. 3.5

Multiply Algorithm (Ver. 1)

0010 x 0011

Product	Multiplier	Multiplicand
0000 0000	0011	0000 0010
0000 0010	0001	0000 0100
0000 0110	0000	0000 1000
0000 0110	0000	0001 0000
0000 0110	0000	0010 0000

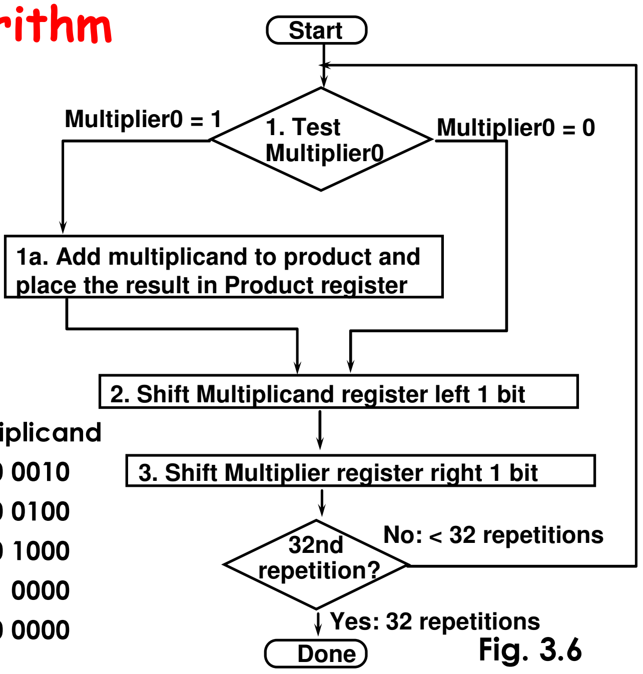

Start
1. Test Multiplier0
Multiplier0 = 1
Multiplier0 = 0
1a. Add multiplicand to product and place the result in Product register
2. Shift Multiplicand register left 1 bit
3. Shift Multiplier register right 1 bit
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done

Fig. 3.6

Observations: Multiply Ver. 1

- 1 clock per cycle => ~100 clocks per multiply
  - Ratio of multiply to add 5:1 to 100:1
- Half of the bits in multiplicand always 0
  => 64-bit adder is wasted
- 0's inserted in left of multiplicand as shifted
  => least significant bits of product never changed once formed
- Instead of shifting multiplicand to left, shift product to right?
- Product register wastes space => combine Multiplier and Product register

Unisigned Multiplier (Ver. 2)

- 32-bit Multiplicand register, 32 -bit ALU, 64-bit Product register (HI & LO in MIPS), (0-bit Multiplier register)

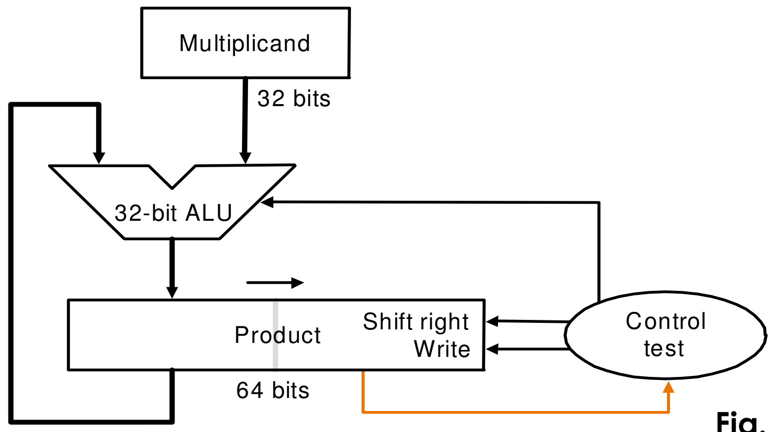
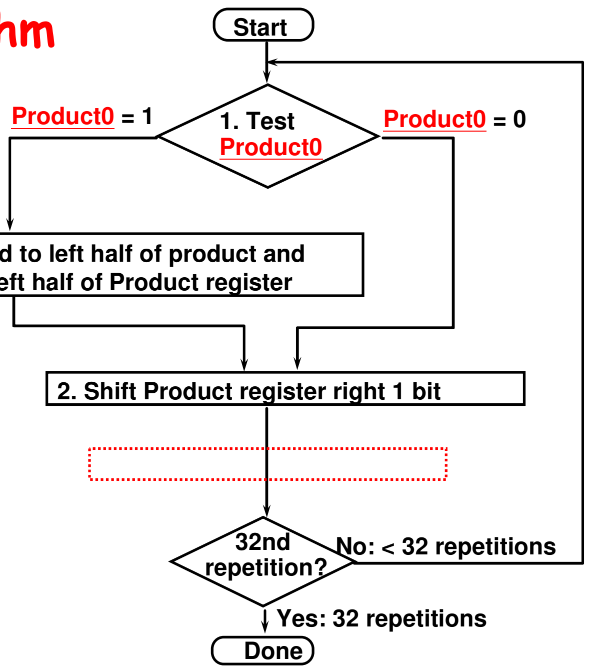


Fig. 3.7

Multiply Algorithm (Ver. 2)


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

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

Observations: Multiply Ver. 2

- 2 steps per bit because multiplier and product registers combined
- MIPS registers Hi and Lo are left and right half of Product register
  => this gives the MIPS instruction MultU
- What about signed multiplication?
  - The easiest solution is to make both positive and remember whether to complement product when done (leave out sign bit, run for 31 steps)
  - Apply definition of 2's complement
    - sign-extend partial products and subtract at end
  - Booth's Algorithm is an elegant way to multiply signed numbers using same hardware as before and save cycles

Booth's Algorithm: Motivation (CD: 3.23 In More Depth)

- Example: 2 x 6 = 0010 x 0110:

```
            0010two
x           0110two
+           0000     shift (0 in multiplier)
+          0010      add (1 in multiplier)
+         0010       add (1 in multiplier)
+        0000        shift (0 in multiplier)
         0001100two
```

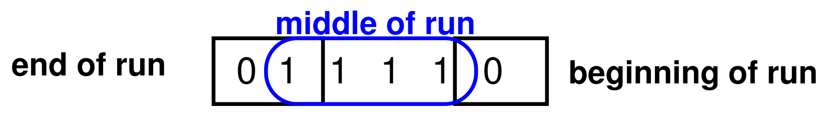
- Can get same result in more than one way:

```
  6 = -2 + 8         0110 = -00010 + 01000
```

- Basic idea: replace a string of 1s with an initial subtract on seeing a one and add after last one

```
            0010two
x           0110two
            0000 shift (0 in multiplier)
-         0010   sub (first 1 in multiplier)
         0000    shift (mid string of 1s)
+       0010     add (prior step had last 1)
        00001100two
```

Booth's Algorithm: Rationale



Current bit	Bit to right	Explanation	Example	Op
1	0	Begins run of 1s	0000111**10**00	sub
1	1	Middle run of 1s	00001**11**1000	none
0	1	End of run of 1s	000**01**111000	add
0	0	Middle run of 0s	0**00**01111000	none

Originally for speed (when shift was faster than add)

- Why it works?

```
   -1
+ 10000
  01111
```

Booth's Algorithm

1. Depending on the current and previous bits, do one of the following:
   00: Middle of a string of 0s, no arithmetic op.
   01: End of a string of 1s, so add multiplicand to the left half of the product
   10: Beginning of a string of 1s, so subtract multiplicand from the left half of the product
   11: Middle of a string of 1s, so no arithmetic op.
2. As in the previous algorithm, shift the Product register right (arithmetically) 1 bit

Booths Example (2 x 7)

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 0111 0	10 -> sub
1a. P = P - m	1110	+1110	
		1110 0111 0	shift P (sign ext)
1b.	0010	1111 0011 1	11 -> nop, shift
2.	0010	1111 1001 1	11 -> nop, shift
3.	0010	1111 1100 1	01 -> add
4a.	0010	+0010	
		0001 1100 1	shift
4b.	0010	0000 1110 0	done

Booths Example (2 x -3)

Operation	Multiplicand	Product	next?
0. initial value	0010	0000 1101 0	10 -> sub
1a. P = P - m	1110	+1110	
		1110 1101 0	shift P (sign ext)
1b.	0010	1111 0110 1	01 -> add
		+0010	
2a.		0001 0110 1	shift P
2b.	0010	0000 1011 0	10 -> sub
		+1110	
3a.	0010	1110 1011 0	shift
3b.	0010	1111 0101 1	11 -> nop
4a		1111 0101 1	shift
4b.	0010	1111 1010 1	done

Faster Multiplier

- A combinational multiplier
- Carry save adder

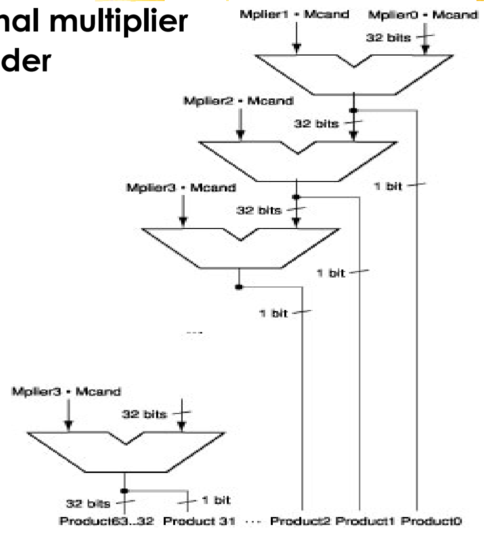


Fig. 3.9

Outline

- Signed and unsigned numbers (Sec. 3.2)
- Addition and subtraction (Sec. 3.3)
- Constructing an arithmetic logic unit (Appendix B.5, B6)
- Multiplication (Sec. 3.4, CD: 3.23 In More Depth)
- Division (Sec. 3.5)
- Floating point (Sec. 3.6)

Divide: Paper & Pencil

$$
\begin{array}{lrrl}
 & & 1001_{ten} & \text{Quotient} \\
\text{Divisor} & 1000_{ten} & \big|\,1001010_{ten} & \text{Dividend} \\
 & & \underline{-1000} & \\
 & & 10 & \\
 & & 101 & \\
 & & 1010 & \\
 & & \underline{-1000} & \\
 & & 10_{ten} & \text{Remainder}
\end{array}
$$

- See how big a number can be subtracted, creating quotient bit on each step
  Binary => 1 * divisor or 0 * divisor
- Two versions of divide, successive refinement
- *Both dividend and divisor are 32-bit positive integers*

Divide Hardware (Version 1)

- 64-bit *Divisor register* (initialized with 32-bit divisor in left half), 64-bit ALU, 64-bit *Remainder register* (initialized with 64-bit dividend), 32-bit *Quotient register*

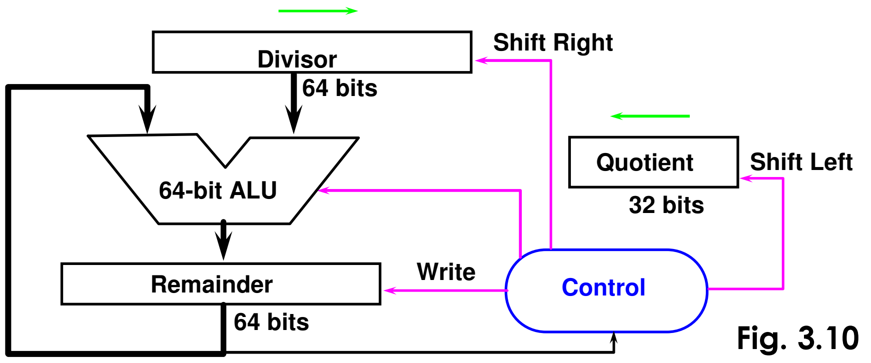


Fig. 3.10

Divide Algorithm (Version 1)

Quot.	Divisor	Rem.
0000	00100000	00000111
		11100111
		00000111
0000	00010000	00000111
		11110111
		00000111
0000	00001000	00000111
		11111111
		00000111
0000	00000100	00000111
		00000011
0001		00000011
0001	00000010	00000011
		00000001
0011		00000001
0011	00000001	00000001

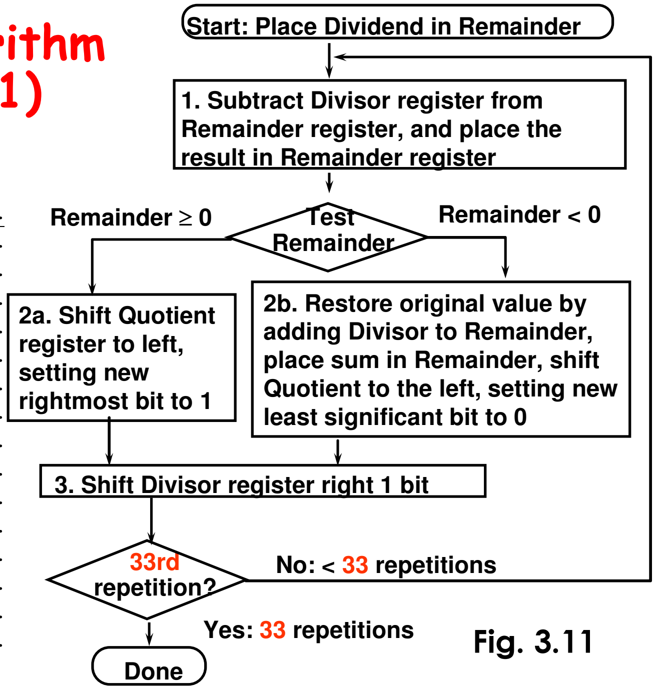

Start: Place Dividend in Remainder
1. Subtract Divisor register from Remainder register, and place the result in Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
2a. Shift Quotient register to left, setting new rightmost bit to 1
2b. Restore original value by adding Divisor to Remainder,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

Fig. 3.11

Observations: Divide Version 1

- Half of the bits in divisor register always 0
  => 1/2 of 64-bit adder is wasted
  => 1/2 of divisor is wasted
- Instead of shifting divisor to right, shift remainder to left?
- 1st step cannot produce a 1 in quotient bit (otherwise quotient is too big for the register)
  => switch order to shift first and then subtract
  => save 1 iteration
- Eliminate Quotient register by combining with Remainder register as shifted left

Divide Hardware (Version 2)

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

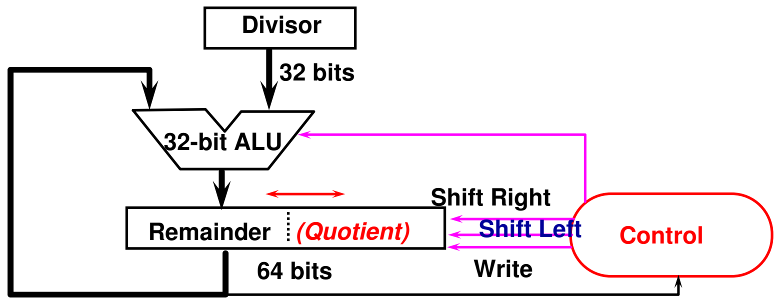
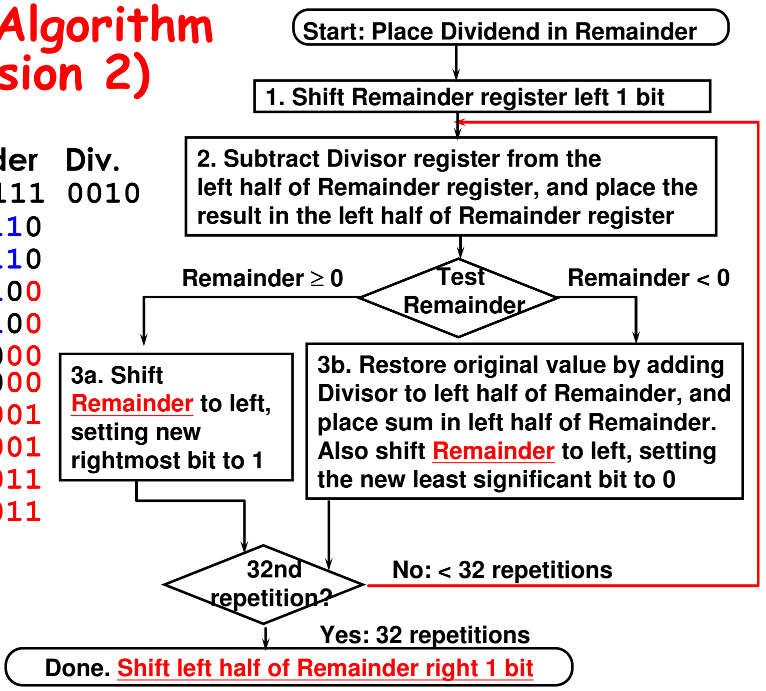


Fig. 3.13

Divide Algorithm (Version 2)

Step	Remainder		Div.
0	0000	0111	0010
1.1	0000	1110	
1.2	1110	1110	
1.3b	$\overline{0}$001	1100	
2.2	1111	1100	
2.3b	$\overline{0}$011	1000	
3.2	0001	1000	
3.3a	$\overline{0}$011	0001	
4.2	0001	0001	
4.3a	$\overline{0}$010	0011	
	0001	0011	


Start: Place Dividend in Remainder
1. Shift Remainder register left 1 bit
2. Subtract Divisor register from the left half of Remainder register, and place the result in the left half of Remainder register
Remainder ≥ 0
Test Remainder
Remainder < 0
3a. Shift Remainder to left, setting new rightmost bit to 1
3b. Restore original value by adding Divisor to left half of Remainder, and place sum in left half of Remainder. Also shift Remainder to left, setting the new least significant bit to 0
32nd repetition?
No: < 32 repetitions
Yes: 32 repetitions
Done. Shift left half of Remainder right 1 bit

Divide

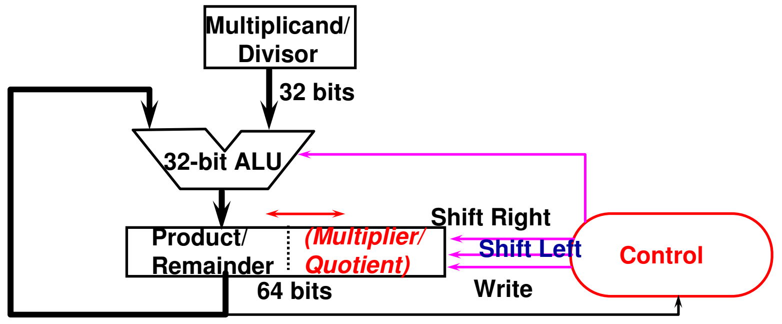
- Signed Divides:
  - Remember signs, make positive, complement quotient and remainder if necessary
  - Alternative: let Dividend and Remainder have same sign and negate Quotient if Divisor sign & Dividend sign disagree, e.g., -7÷ 2 = -3, remainder = -1
- Possible for quotient to be too large:
  if divide 64-bit integer by 1, quotient is 64 bits

Observations: Multiply and Divide

- Same hardware as multiply: just need ALU to add or subtract, and 64-bit register to shift left or shift right
- Hi and Lo registers in MIPS combine to act as 64-bit register for multiply and divide

Multiply/Divide Hardware

- 32-bit Multiplicand/Divisor register, 32 -bit ALU, 64-bit Product/Remainder register, (0-bit Multiplier/Quotient register)



Outline

- ♦ Signed and unsigned numbers (Sec. 3.2)
- ♦ Addition and subtraction (Sec. 3.3)
- ♦ Constructing an arithmetic logic unit (Appendix B.5, B6)
- ♦ Multiplication (Sec. 3.4, CD: 3.23 In More Depth)
- ♦ Division (Sec. 3.5)
- ♦ Floating point (Sec. 3.6)

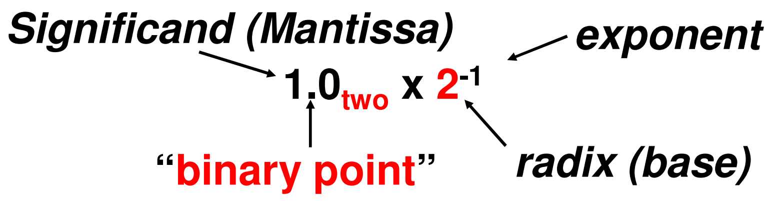
Floating-Point: Motivation

- ♦ What can be represented in N bits?

Unsigned	0	to	$2^n - 1$
2's Complement	-2^{n-1}	to	$2^{n-1} - 1$
1's Complement	$-2^{n-1}+1$	to	2^{n-1}
Excess M	$-M$	to	$2^n - M - 1$

- ♦ But, what about ...
  - very large numbers?
    9,349,398,989,787,762,244,859,087,678
  - very small number?
    0.0000000000000000000000045691
  - rationals 2/3
  - irrationals $\sqrt{2}$
  - transcendentals e, π

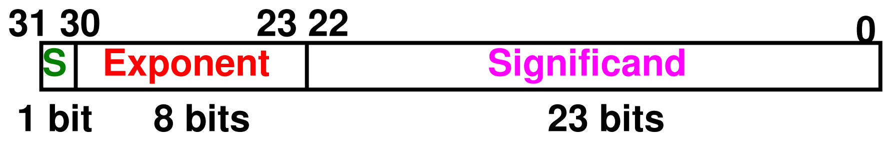
Scientific Notation: Binary



- ♦ Computer arithmetic that supports it is called floating point, because the binary point is not fixed, as it is for integers
- ♦ Normalized form: no leading 0s
  (exactly one digit to left of decimal point)
- ♦ Alternatives to represent 1/1,000,000,000
  - Normalized: 1.0×10^{-9}
  - Not normalized: 0.1×10^{-8}, 10.0×10^{-10}

FP Representation

- ♦ Normal format: $1.xxxxxxxxxx_{two} \times 2^{yyyy_{two}}$
- ♦ Want to put it into multiple words: 32 bits for *single-precision* and 64 bits for *double-precision*
- ♦ A simple single-precision representation:



S represents sign
Exponent represents y's
Significand represents x's

- ♦ Represent numbers as small as 2.0×10^{-38} to as large as 2.0×10^{38}

Double Precision Representation

- Next multiple of word size (64 bits)

31	30 20	19 0
S	Exponent	Significand
1 bit	11 bits	20 bits

Significand (cont'd)
32 bits

- Double precision (vs. single precision)
  - Represent numbers almost as small as 2.0×10^{-308} to almost as large as 2.0×10^{308}
  - But primary advantage is greater accuracy due to larger significand

IEEE 754 Standard (1/4)

- Regarding single precision, DP similar
- Sign bit:
  - 1 means negative
  - 0 means positive
- Significand:
  - To pack more bits, leading 1 implicit for normalized numbers
  - 1 + 23 bits single, 1 + 52 bits double
  - always true: 0 < Significand < 1
    (for normalized numbers)
- Note: 0 has no leading 1, so reserve exponent value 0 just for number 0

IEEE 754 Standard (2/4)

- Exponent:
  - Need to represent positive and negative exponents
  - Also want to compare FP numbers as if they were integers, to help in value comparisons
  - If use 2's complement to represent?
    e.g., 1.0×2^{-1} versus $1.0 \times 2^{+1}$ (1/2 versus 2)

1/2	0	1111 1111	000 0000 0000 0000 0000 0000
2	0	0000 0001	000 0000 0000 0000 0000 0000

If we use integer comparison for these two words, we will conclude that 1/2 > 2!!!

IEEE 754 Standard (3/4)

- Instead, let notation 0000 0000 be most negative, and 1111 1111 most positive
- Called biased notation, where bias is the number subtracted to get the real number
  - IEEE 754 uses bias of 127 for single precision:
    Subtract 127 from Exponent field to get actual value for exponent
  - 1023 is bias for double precision

1/2	0	0111 1110	000 0000 0000 0000 0000 0000
2	0	1000 0000	000 0000 0000 0000 0000 0000

IEEE 754 Standard (4/4)

- Summary (single precision):

31	30 … 23	22 … 0
S	Exponent	Significand
1 bit	8 bits	23 bits

$(-1)^S \times (1.\text{Significand}) \times 2^{(\text{Exponent}-127)}$

- Double precision identical, except with exponent bias of 1023

Example: FP to Decimal

0	0110 1000	101 0101 0100 0011 0100 0010

- Sign: 0 => positive
- Exponent:
  - 0110 1000_{two} = 104_{ten}
  - Bias adjustment: 104 - 127 = -23
- Significand:
  - $1+2^{-1}+2^{-3}+2^{-5}+2^{-7}+2^{-9}+2^{-14}+2^{-15}+2^{-17}+2^{-22}$ $= 1.0 + 0.666115$
- Represents: $1.666115_{ten}\times 2^{-23} \approx 1.986 \times 10^{-7}$

Example 1: Decimal to FP

- Number = - 0.75
  = $-\ 0.11_{two} \times 2^{0}$ (scientific notation)
  = $-\ 1.1_{two} \times 2^{-1}$ (normalized scientific notation)

- Sign: negative => 1
- Exponent:
  - Bias adjustment: -1 +127 = 126
  - 126_{ten} = 0111 1110_{two}

1	0111 1110	100 0000 0000 0000 0000 0000

Example 2: Decimal to FP

- A more difficult case: representing 1/3?
  = $0.33333\ldots_{10}$ = $0.0101010101\ldots_{2} \times 2^{0}$
  = $1.0101010101\ldots_{2} \times 2^{-2}$
  - Sign: 0
  - Exponent = -2 + 127 = 125_{10}=01111101_2
  - Significand = 0101010101…

0	0111 1101	0101 0101 0101 0101 0101 010

Zero and Special Numbers

- What have we defined so far? (single precision)

Exponent	Significand	Object
0	0	???
0	nonzero	???
1-254	anything	+/- floating-point
255	0	???
255	nonzero	???

Representation for 0

- Represent 0?
  - exponent all zeroes
  - significand all zeroes too
  - What about sign?
  - `+0: 0 00000000 00000000000000000000000`
  - `-0: 1 00000000 00000000000000000000000`
- Why two zeroes?
  - Helps in some limit comparisons

Special Numbers

- What have we defined so far? (single precision)

Exponent	Significand	Object
0	0	0
0	nonzero	???
1-254	anything	+/- floating-point
255	0	???
255	nonzero	???

- Range:

  $1.0 \times 2^{-126} \approx 1.8 \times 10^{-38}$

  What if result too small? (>0, < 1.8×10^{-38} => Underflow!)

  $(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$

  What if result too large? (> 3.4×10^{38} => Overflow!)

Representation for +/- Infinity

- In FP, divide by zero should produce +/- infinity, not overflow
- Why?
  - OK to do further computations with infinity, e.g., X/0 > Y may be a valid comparison
- IEEE 754 represents +/- infinity
  - Most positive exponent reserved for infinity
  - Significands all zeroes

S	1111 1111	0000 0000 0000 0000 0000 000

Representation for Not a Number

- What do I get if I calculate sqrt(-4.0) or 0/0?
  - If infinity is not an error, these should not be either
  - They are called *Not a Number* (NaN)
  - Exponent = 255, Significand nonzero
- Why is this useful?
  - Hope NaNs help with debugging?
  - They contaminate: op(NaN,X) = NaN
  - OK if calculate but don't use it

Special Numbers (cont'd)

- What have we defined so far? (single-precision)

Exponent	Significand	Object
0	0	0
0	nonzero	denom (???)
1-254	anything	+/- fl. pt. #
255	0	+/- infinity
255	nonzero	NaN

Floating-Point Addition

Basic addition algorithm:

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

(2) right shift the smaller number, say Xm, that many positions to form Xm $\times 2^{Xe-Ye}$

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

if demands normalization, then normalize:

(4) left shift result, decrement result exponent
right shift result, increment result exponent
(4.1) check overflow or underflow during the shift
(4.2) round the mantissa
continue until MSB of data is 1
(NOTE: Hidden bit in IEEE Standard)

(5) if result is 0 mantissa, set the exponent

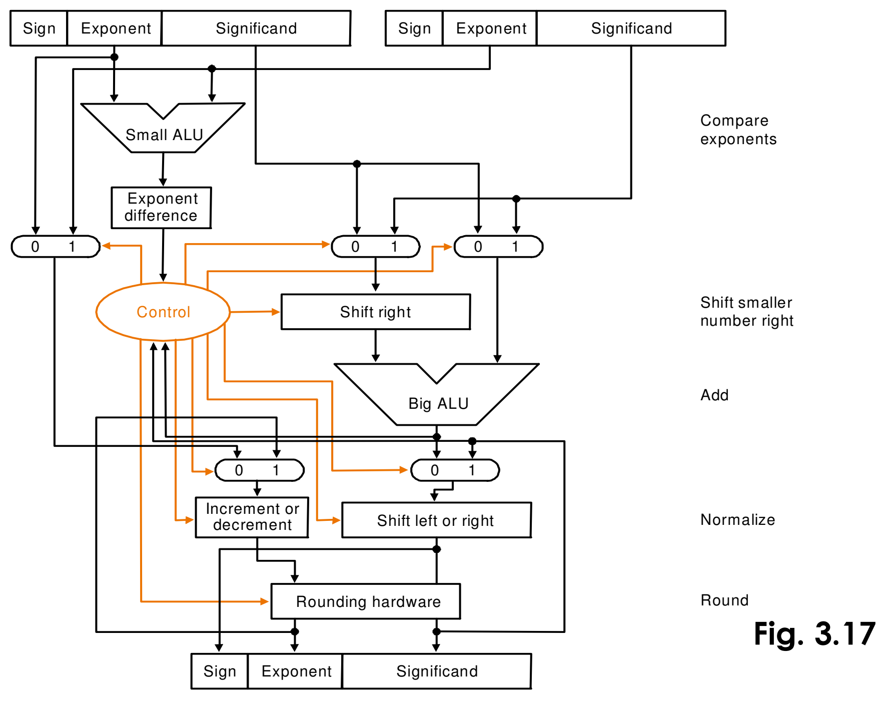


Fig. 3.17

Floating-Point Multiplication

Basic multiplication algorithm

(1) add exponents of operands to get exponent of product

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 + 8

need extra subtraction step of the bias amount

(2) multiplication of operand mantissa

(3) normalize the product

(3.1) check overflow or underflow during the shift

(3.2) round the mantissa

continue until MSB of data is 1

(4) set the sign of product

MIPS Floating Point

- Separate floating point instructions:
  - Single precision: `add.s`, `sub.s`, `mul.s`, `div.s`
  - Double precision: `add.d`, `sub.d`, `mul.d`, `div.d`
- FP part of the processor:
  - contains 32 32-bit registers: `$f0`, `$f1`, …
  - most registers specified in .s and .d instruction refer to this set
  - separate load and store: `lwc1` and `swc1`
  - Double Precision: by convention, even/odd pair contain one DP FP number: `$f0`/`$f1`, `$f2`/`$f3`
  - Instructions to move data between main processor and coprocessors:
    - `mfc0`, `mtc0`, `mfc1`, `mtc1`, etc.
  - See CD A-73 to A-80

Floating Point Fallacy

- FP add, subtract associative?
  FALSE!
  - $x = -1.5 \times 10^{38}$, $y = 1.5 \times 10^{38}$, $z = 1.0$
  - $x + (y + z) = -1.5\text{x}10^{38} + (1.5\text{x}10^{38} + 1.0)$
    $= -1.5\text{x}10^{38} + (1.5\text{x}10^{38}) = 0.0$
  - $(x + y) + z = (-1.5\text{x}10^{38} + 1.5\text{x}10^{38}) + 1.0$
    $= (0.0) + 1.0 = 1.0$
- Therefore, Floating Point add, subtract are not associative!
  - Why? FP result approximates real result!
  - This example: 1.5×10^{38} is so much larger than 1.0 that $1.5 \times 10^{38} + 1.0$ in floating point representation is still 1.5×10^{38}

Summary

- MIPS arithmetic: successive refinement to see final design
  - 32-bit adder and logic unit
  - 32-bit multiplier and divisor, with HI and LO
  - Booth's algorithm to handle signed multiplies
- Floating point numbers *approximate* values that we want to use
  - IEEE 754 Floating Point Standard is most widely accepted to standardize their interpretation
  - New MIPS registers (`$f0`-`$f31`) and instructions:
    - Single-precision (32 bits, $2\text{x}10^{-38} \ldots 2\text{x}10^{38}$): `add.s`, `sub.s`, `mul.s`, `div.s`
    - Double-precision (64 bits , $2\text{x}10^{-308} \ldots 2\text{x}10^{308}$): `add.d`, `sub.d`, `mul.d`, `div.d`
- Type is not associated with data, bits have no meaning unless given in context