Lecture 4: Arithmetic for Computers

CSCE 2610 Computer Organization

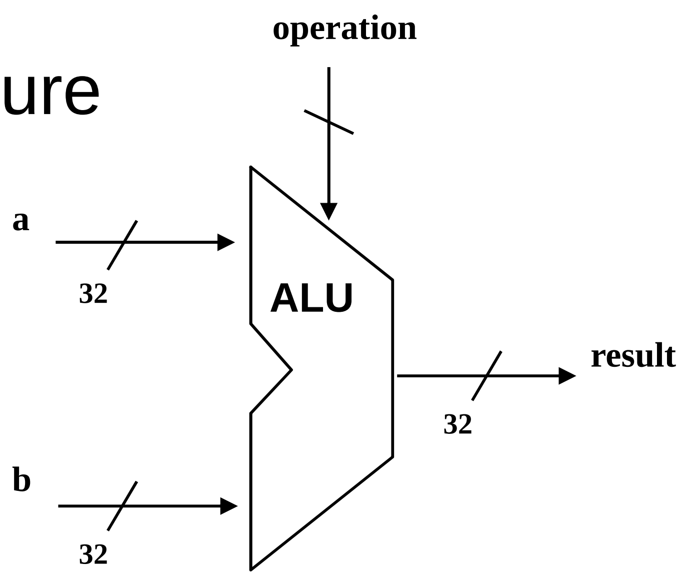
Instructor: Saraju P. Mohanty, Ph. D.

Outline of this Lecture

- Addition/Subtraction operation
- Logic operation
- Design of arithmetic and logic unit (ALU)
- Multiplication operation
- Design of hardware for multiplication
- Division operation
- Design of hardware for division
- Floating point operation
- Design of hardware for floating point operation

Arithmetic

- Where we've been:
  - Performance (seconds, cycles, instructions)
  - Abstractions:
    Instruction Set Architecture
    Assembly Language and Machine Language
- What's up ahead:
  - Implementing the Architecture



Let us first learn numbers!!

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?

Value of a Digit or Number

- In any number base, the value of ith digit d is: d x Base^i
- "i" starts at 0 and increases from right to left.
- For decimal base is 10, for binary base is 2.
- For clarity decimals will have subscript 10 and binary will have subscript 2 and so on….
- Example: 1011^2 represents

$$\begin{aligned} & (1x2^3) + (0x2^2) + (1x2^1) + (1x2^0)_{10} \\ &= (1x8) + (0x4) + (1x2) + (1x1)_{10} \\ &= 8 + 0 + 2 + 1_{10} \\ &= 11_{10} \end{aligned}$$

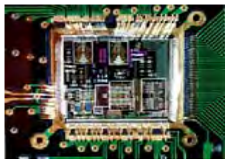
Why Don’t Computers Use Decimals?

- Easy hardware implementation
- The building block of digital computers, the transistors, act as a switch. A switch has two states ON or OFF.
- Converting back and forth between binary and decimal can be for infrequent input/output events can be inefficient.

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$_{ten}$ ***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 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)

Decimal Value of a 2's Complement Binary

- 32-bit 2's complement number

  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
- Decimal value:

  $(1x\text{-}2^{31}) + (1x2^{30}) + (1x2^{29}) + \ldots. + (0x2^1) + (0x2^0)_{10}$

  $= -2^{31} \quad + 2^{30} \quad + 2^{29} \quad + \ldots. \quad + 0 \quad + 0_{10}$

  $= -2{,}147{,}483{,}648_{10} \quad + -2{,}147{,}483{,}644_{10}$

  $= -4_{10}$

Negation

- Negate 2_{10}
  $2_{10} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2$
  Inverting bits:
  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2$
  Adding 1:
  $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 = -2_{10}$

- Negate -2_{10}
  $-2_{10} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2$
  Inverting bits:
  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2$
  Adding 1:
  $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$

Memory Space for Different Data Type

Type	Description	Size
char	Character or small integer.	1byte
short int (short)	Short Integer.	2bytes
int	Integer.	4bytes
long int (long)	Long integer.	4bytes
bool	Boolean value. It can take one of two values: true or false.	1byte
float	Floating point number.	4bytes
double	Double precision floating point number.	8bytes
long double	Long double precision floating point number.	8bytes

Source: http://www.cplusplus.com/doc/tutorial/variables.html

Addition and Subtraction

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

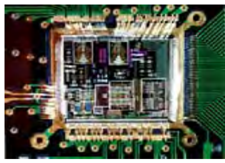
```
  0111            0111            0110
+ 0110          - 0110          - 0101
```

- 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  note that overflow term is somewhat misleading,
_ 1000          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

Operation	Operand A	Operand B	Result
A + B	>= 0	>= 0	< 0
A + B	< 0	< 0	>=0
A - B	>= 0	< 0	< 0
A - B	< 0	>= 0	>= 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

- MIPS instructions:add, addi, sub cause exceptions on overflow

- Don't always want to detect overflow

  — MIPS instructions: addu, addiu, subu

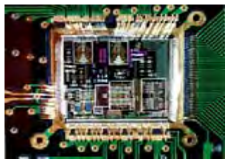
  do not cause exceptions on overflow

Exception and Interrupt

- Exception: An unscheduled event that disrupts program execution.
- Interrupt: An exception that comes from outside of the processor.
- Some architectures use the term interrupt for all exceptions.

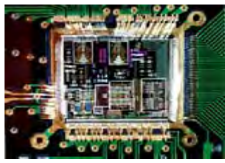
Exception in MIPS

- MIPS has a register called "exception program counter" (EPC) to contain the address of the instruction that caused exception.
- The instruction "move from system control" (mfc0) copies EPC into a GPR so that program can return to the offending instruction via a "jump register" (jr) instruction.



What Happens in a Computer When Interrupt/Exception Occurs?

- States of the associated registers are saved.
- Subroutines in an operating system or device driver called interrupt handlers or an interrupt service routines (ISRs), is triggered for execution.
- Interrupt service routines (ISRs), have a several functions to handle different types of interrupt/exception.
- ISRs serve the interrupt.
- Registers are loaded back.
- Execution of the program that caused exception continues.



Logical Operations

- Shift left logical (sll)

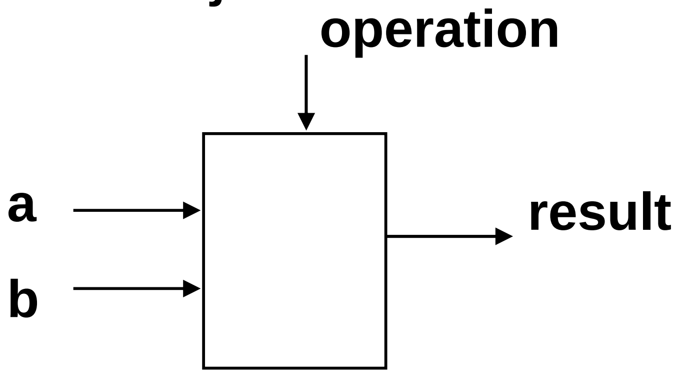
  Sll $10, $16, 8 # reg $10 = reg $16 << 8 bits

op	rs	rt	rd	shamt	funct
0	0	16	10	8	0

- Shift right logical (srl)
- AND, OR operations (and, andi, or , ori)

An ALU (arithmetic logic unit)

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

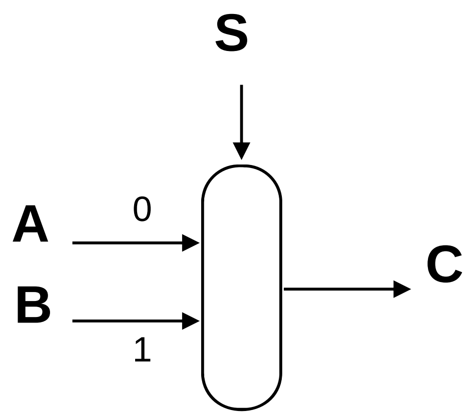


op	a	b	result

- Possible Implementation (sum-of-products):

Review: The Multiplexor

- Selects one of the inputs to be the output, based on a control input

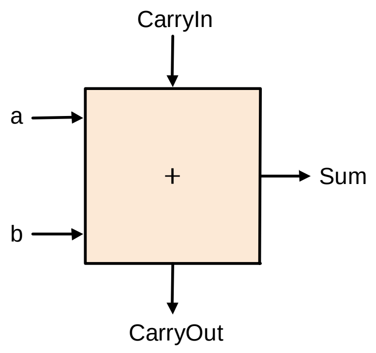


note: we call this a 2-input mux
even though it has 3 inputs!

- Lets build our ALU using MUXes:

Different Implementations

- Not easy to decide the “best” way to build something
  - Don't want too many inputs to a single gate
  - Dont 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:

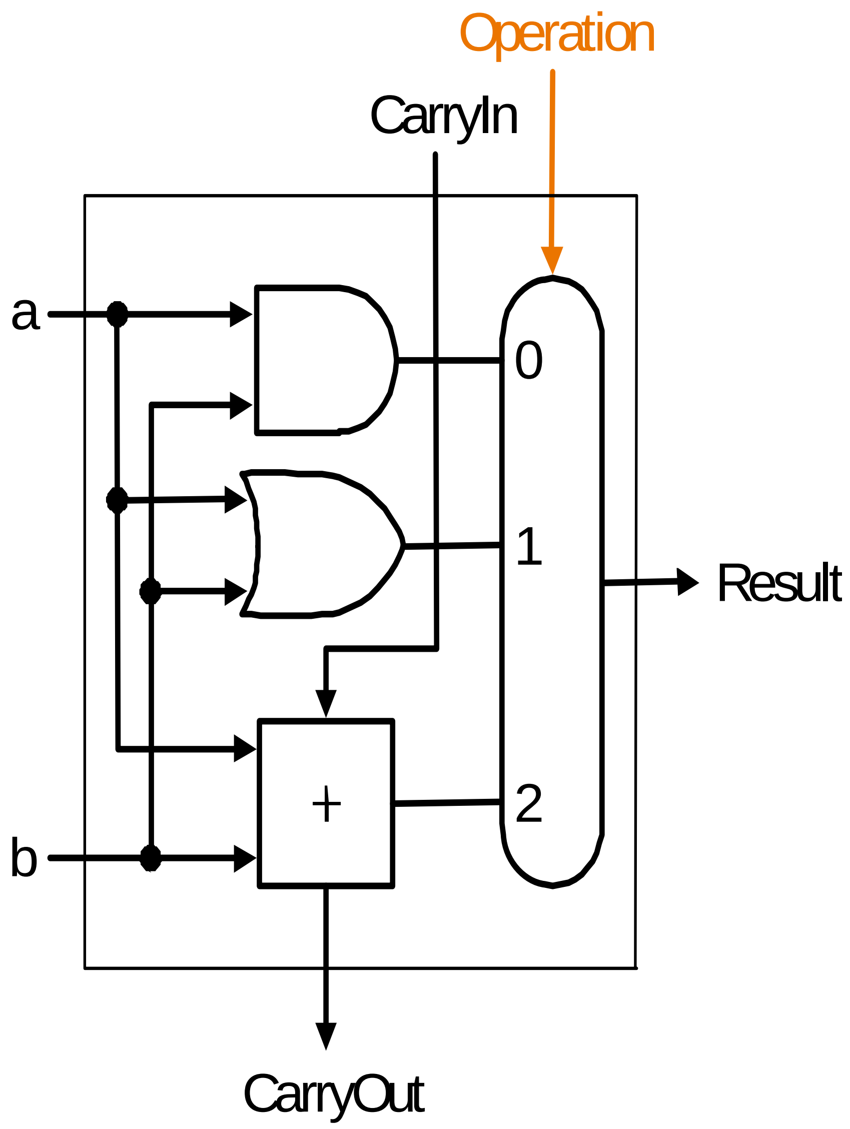


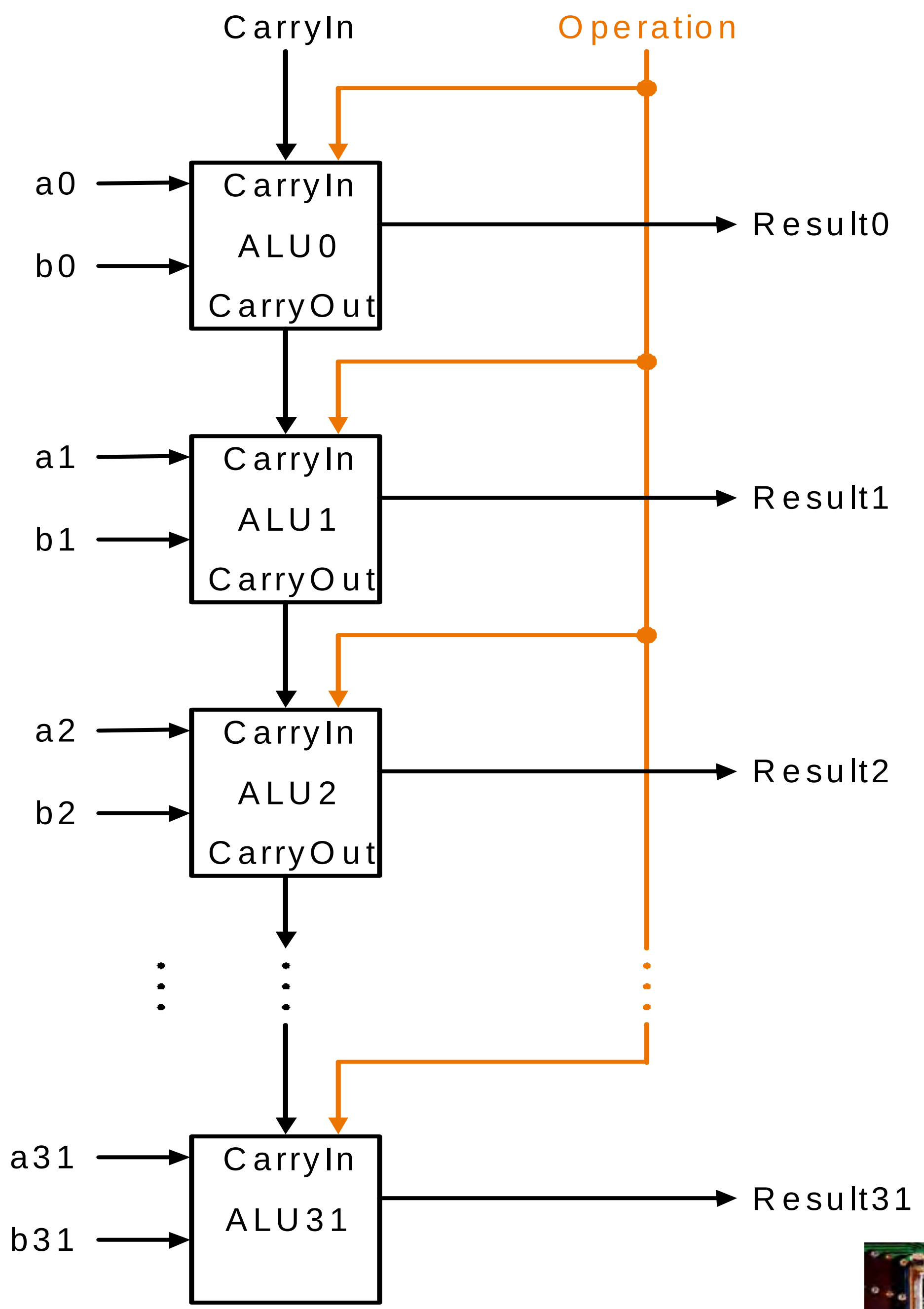
c_{out} = a b + a c_{in} + b c_{in}
sum = a xor b xor c_{in}

Different Implementations …

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

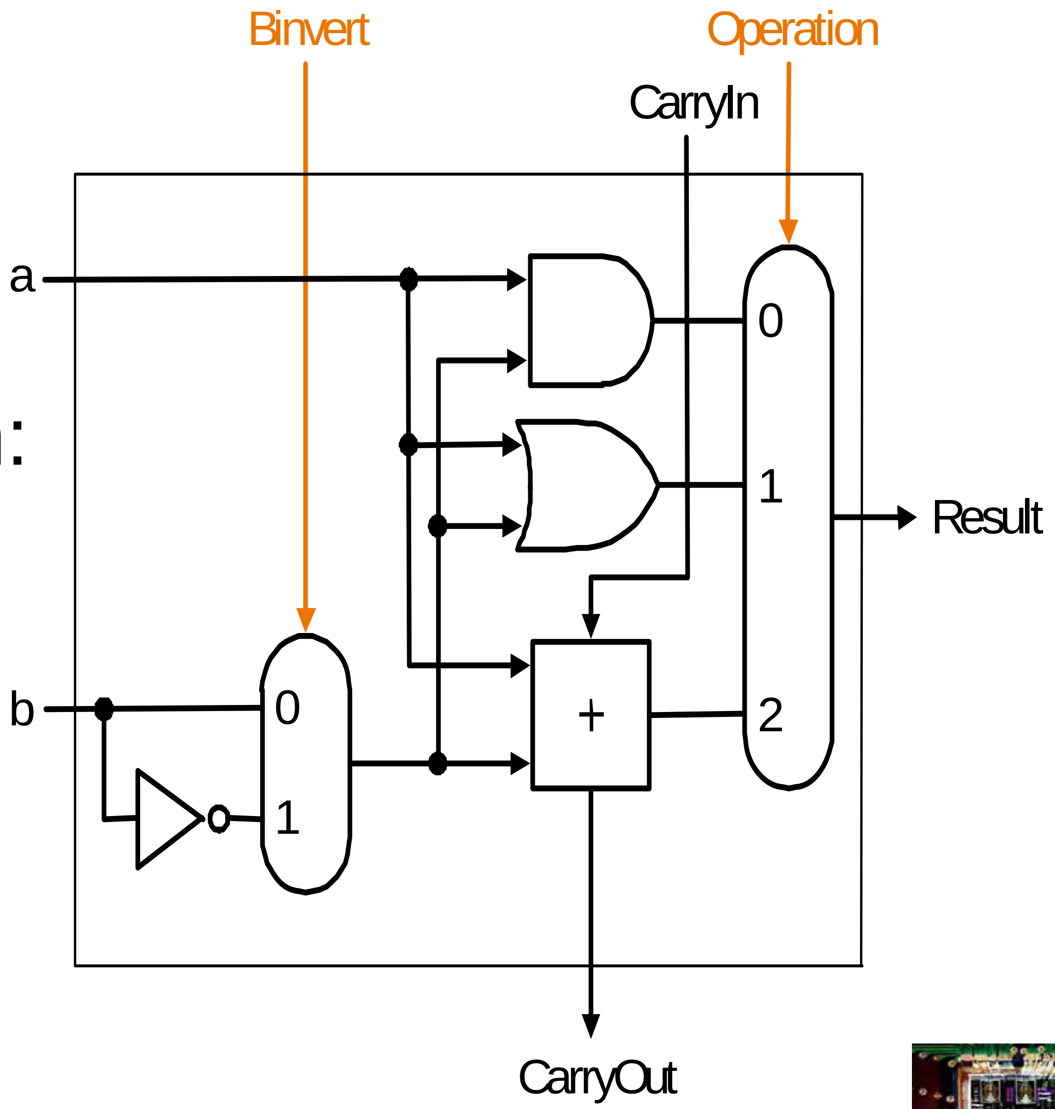
Building a 32 bit ALU





What about subtraction (a – b) ?

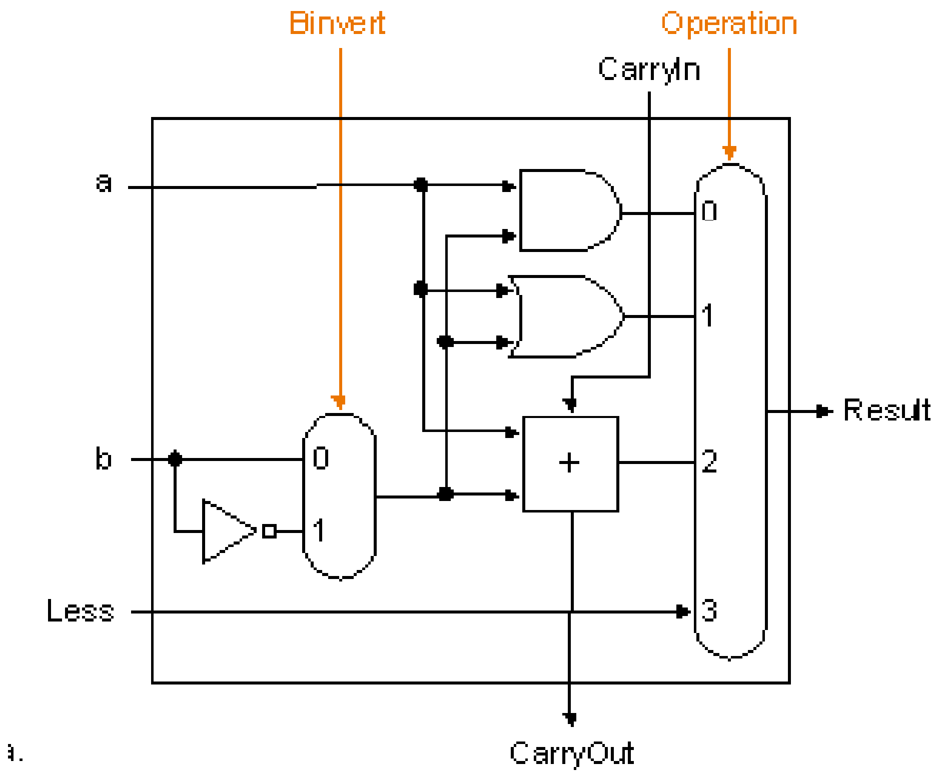
- Two's complement approach: just negate b and add.
- How do we negate?
- A very clever solution:



Tailoring the ALU to the MIPS

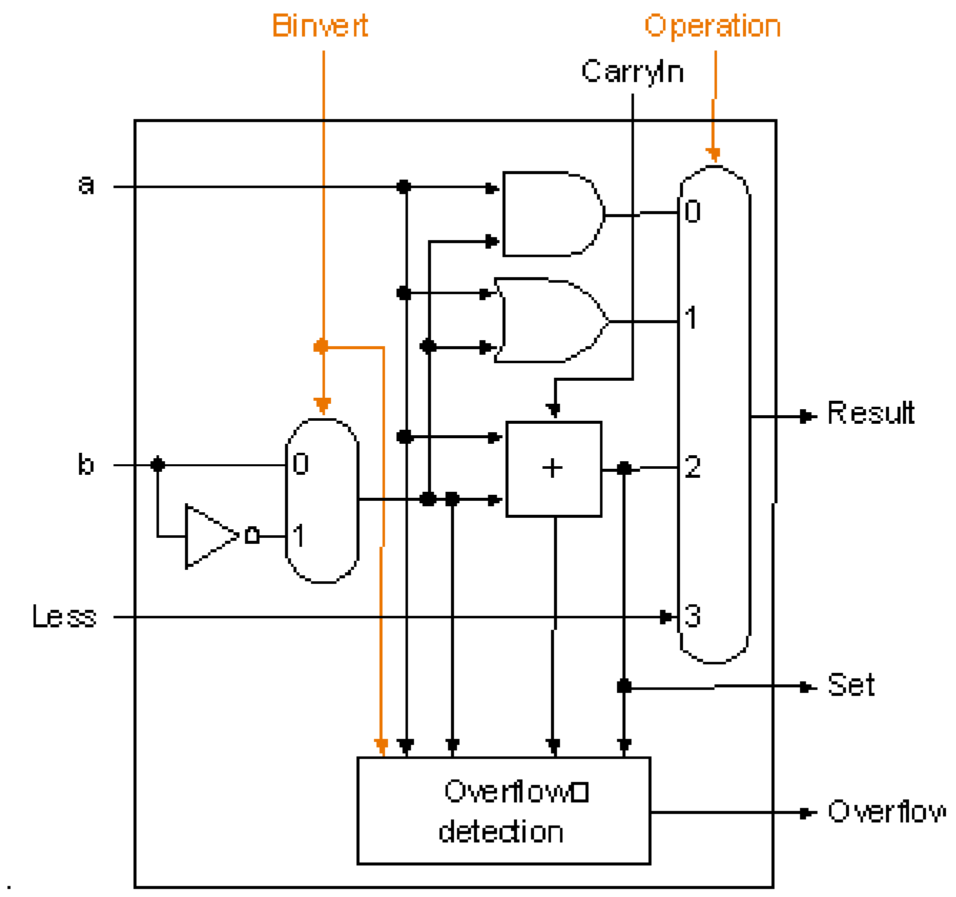
- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (beq $t5, $t6, $t7)
  - use subtraction: (a-b) = 0 implies a = b

Supporting slt for MIPS



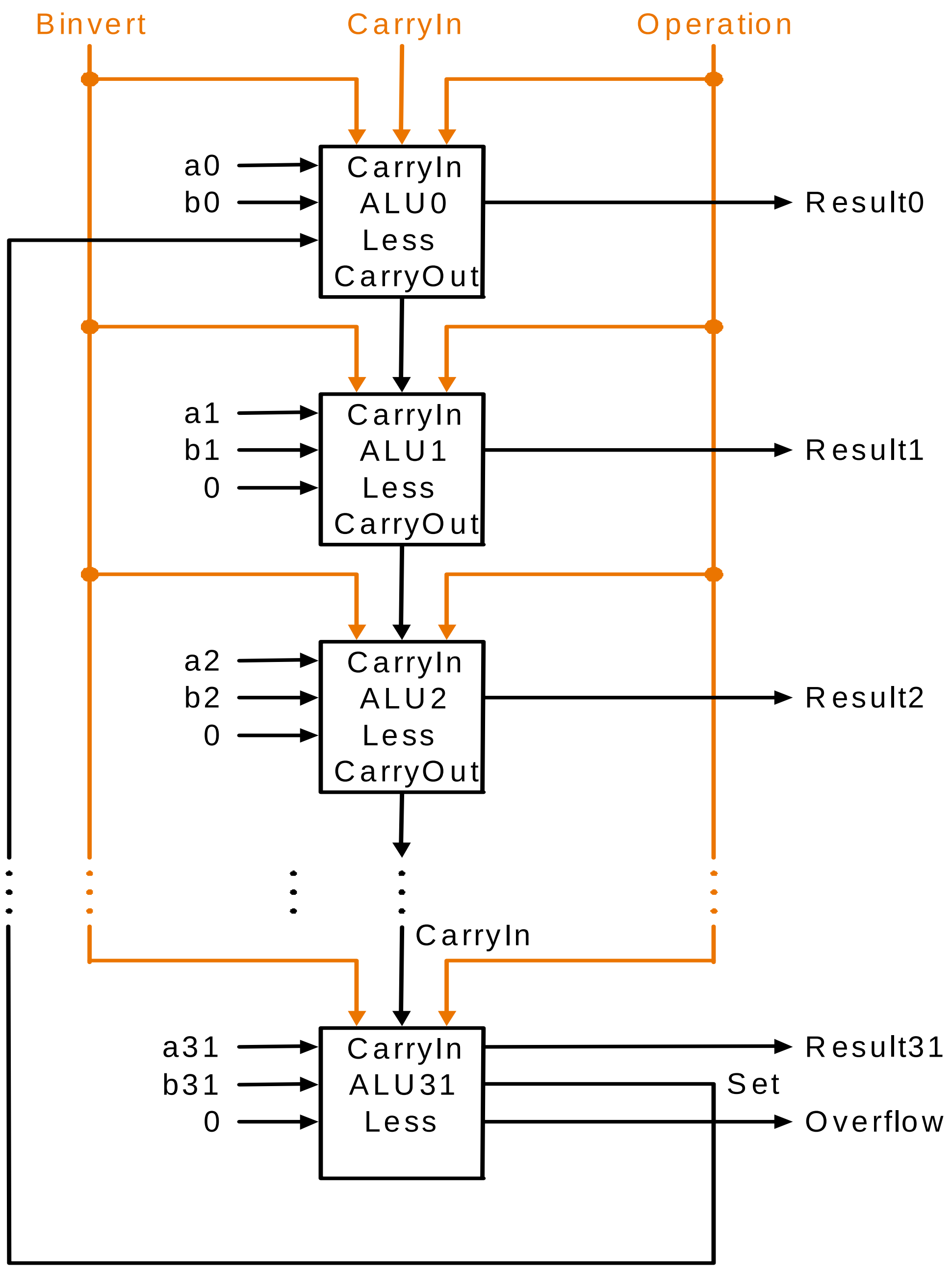
Less will be zero for all bits other than LSB which will be 0 or 1 coming from the “set” output of MSB.

Supporting slt and Overflow: 1-bit ALU for MSB

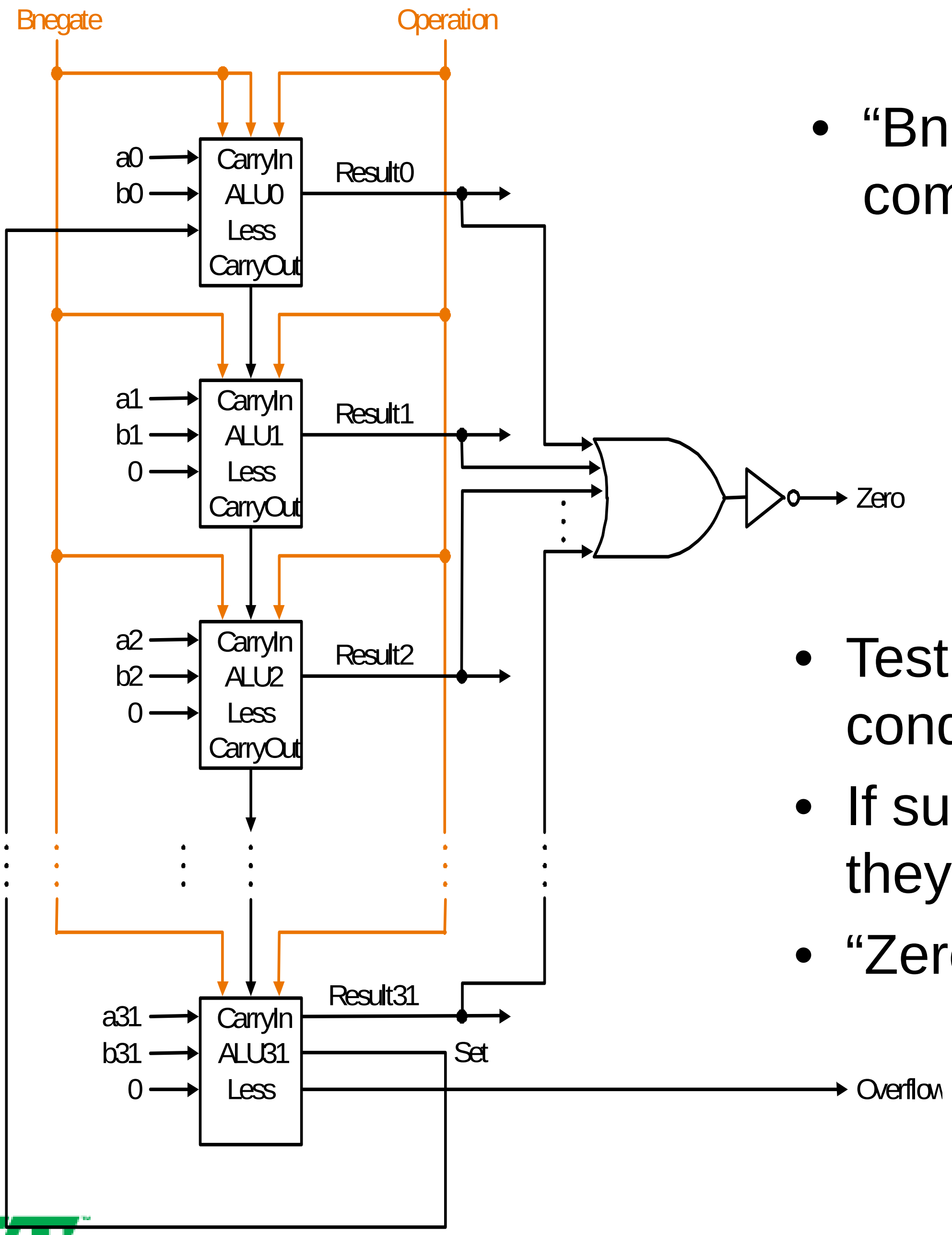


Overflow detection logic at the most significant bit (MSB) ALU.

32-bit ALU for MIPS: Using 32 1-bit ALUs



32-bit ALU for MIPS: Using 32 1-bit ALUs



- “Bnegate” is a single control line combining CarryIn and Binvert.

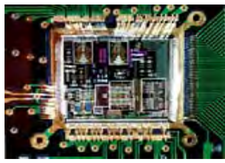
- Testing for equality needed for conditional branch instructions.
- If subtraction results is 0, then they are equal.
- “Zero” is a 1 when the result is 0!

ALU Design: Summary

- We can build an ALU to support the MIPS instruction set
  - key idea: use multiplexor to select the output we want
  - we can efficiently perform subtraction using two's complement
  - we can replicate a 1-bit ALU to produce a 32-bit ALU
- Important points about hardware
  - all of the gates are always working
  - the speed of a gate is affected by the number of inputs to the gate
  - the speed of a circuit is affected by the number of gates in series (on the "critical path" or the "deepest level of logic")
- Our primary focus: comprehension, however,
  - Clever changes to organization can improve performance (similar to using better algorithms in software)
  - we'll look at two examples for addition and multiplication

Binary Multiplication

- More complicated than addition
  - accomplished via shifting and addition
- More time and more area
- Negative numbers: convert and multiply
  - there are better techniques.



Binary Multiplication

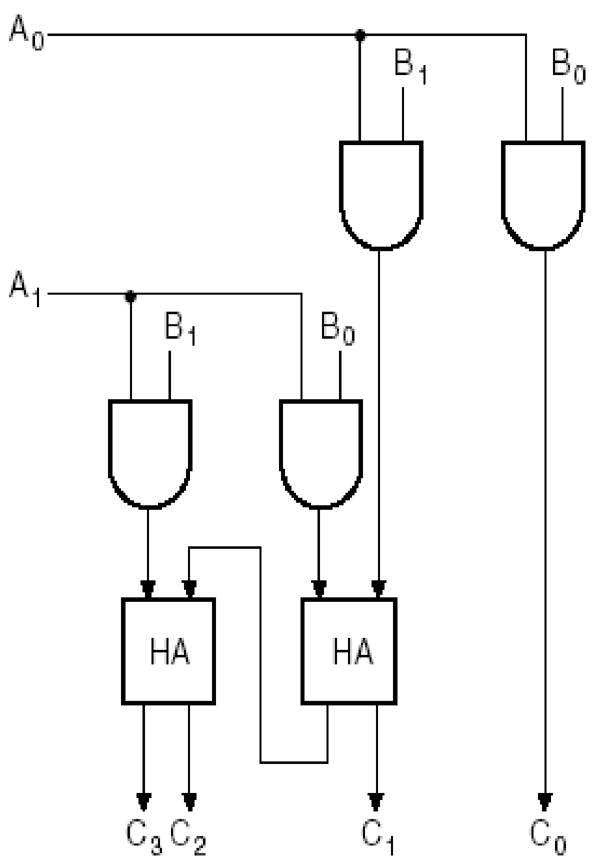
- Example:

```
Multiplicand:      1011
Multiplier:      x  101
                 ------
                   1011
                  0000
                 1011
                 ------
Product:         110111
```

- Observation : The multiplier bits are always 1 or 0, therefore the partial products are equal to either the multiplicand or to 0.
- The above fact has been exploited in various ways, and many time and hardware efficient multiplication algorithms have been developed.
- Booth's multiplier and Wallace-Tree multiplier are two examples.

Binary Multipliers: A 2-bit example

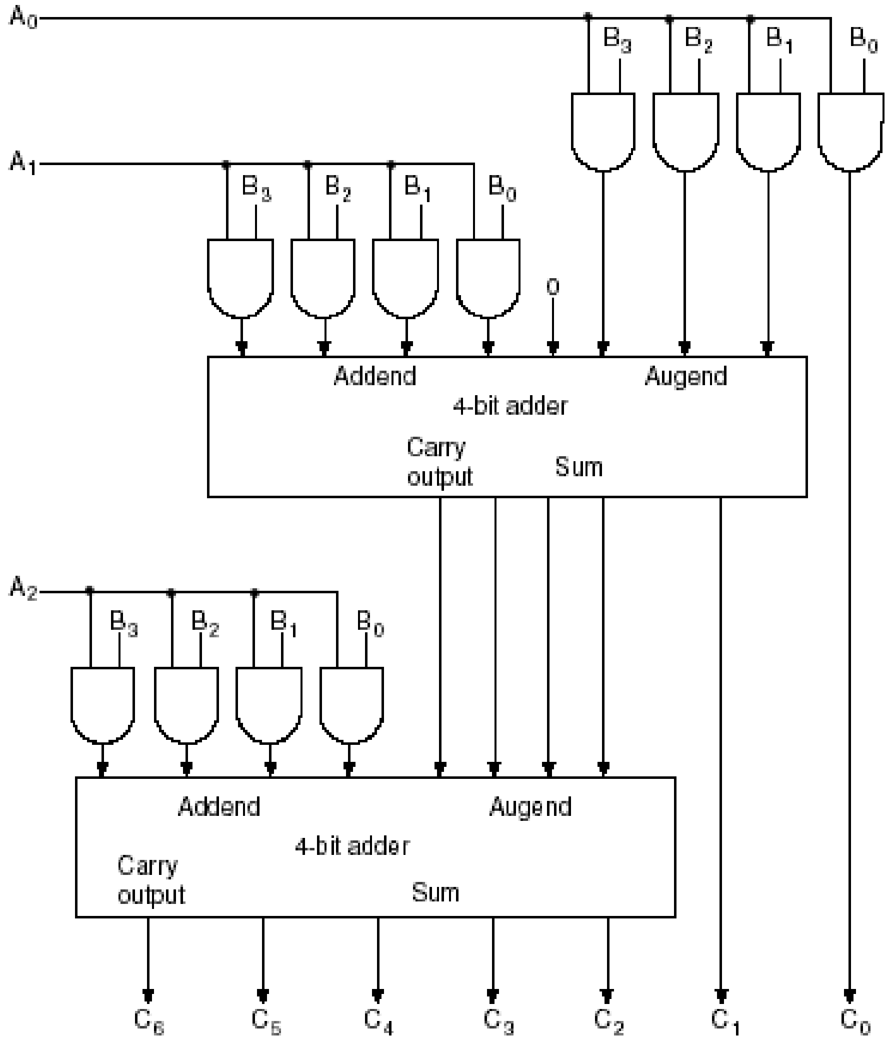
$$\begin{array}{cccc}
 & & B_1 & B_0 \\
 & & A_1 & A_0 \\
\hline
 & & A_0B_1 & A_0B_0 \\
 & A_1B_1 & A_1B_0 & \\
\hline
C_3 & C_2 & C_1 & C_0
\end{array}$$



A 2-Bit by 2-Bit Binary Multiplier

Product A_0 and B_0 is 1 if both are 1, else it is 0. Thus, the product is same as AND operation.

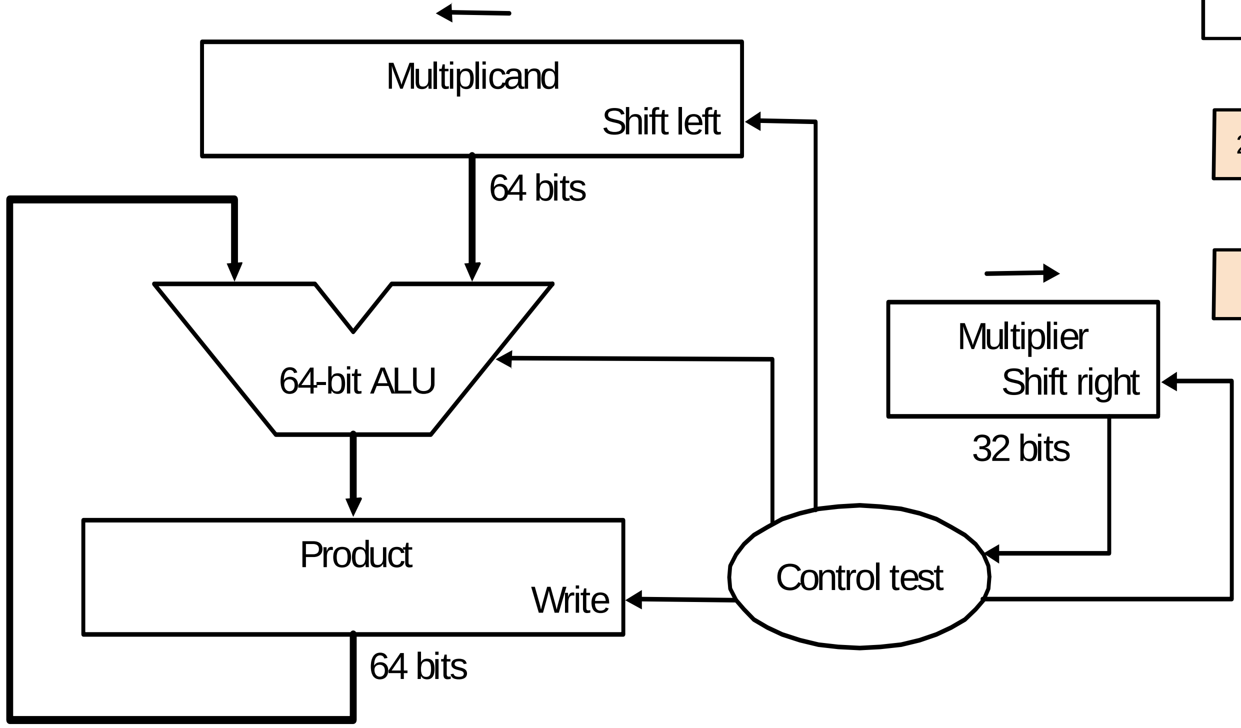
Binary Multipliers: A 4-bit by 3-bit example

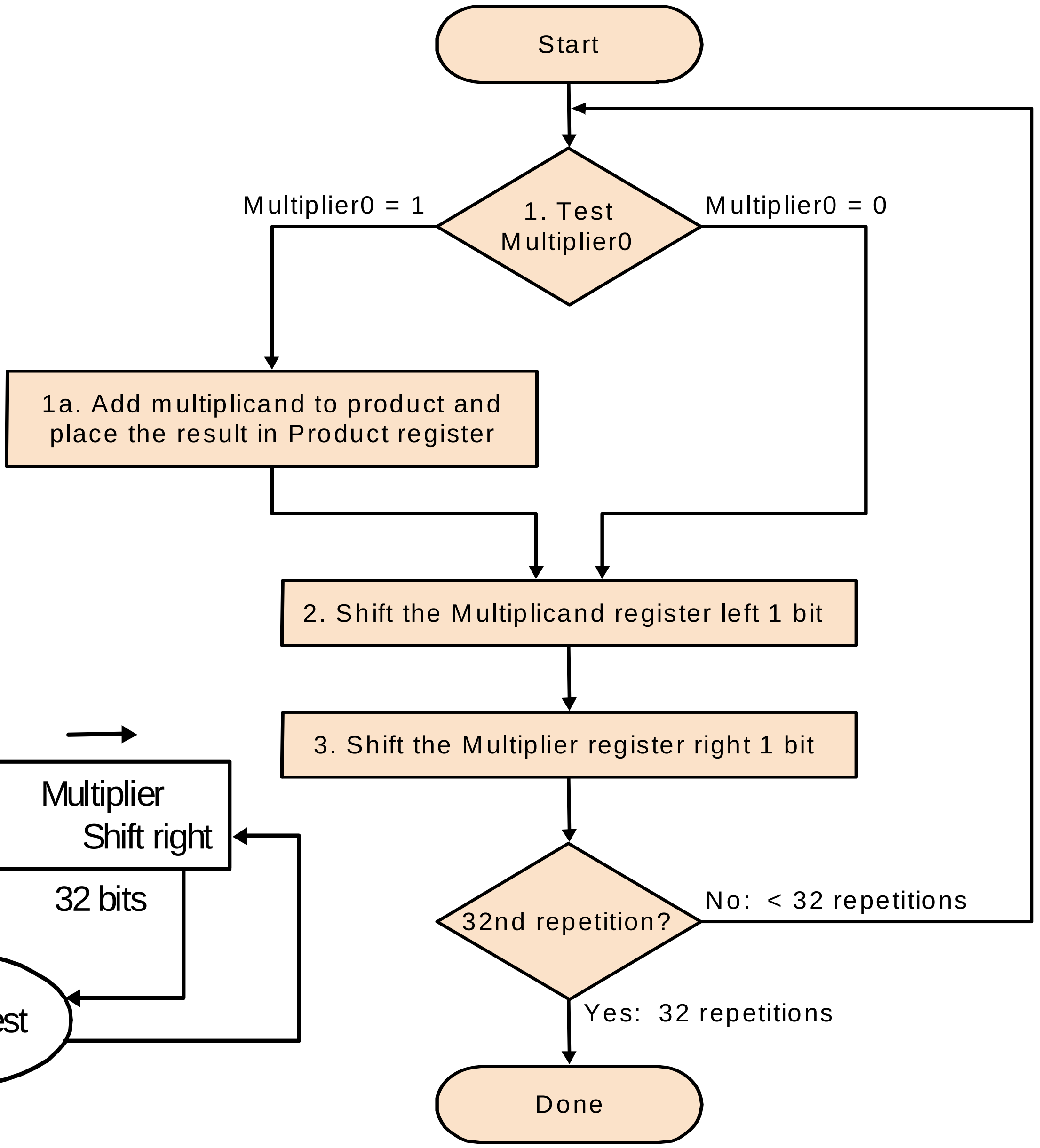


A 4-Bit by 3-Bit Binary Multiplier

For J multiplier bits and K multiplicand bits, we need JxK AND gates and (J-1) K-bit adders to produce a product of J+K bits.

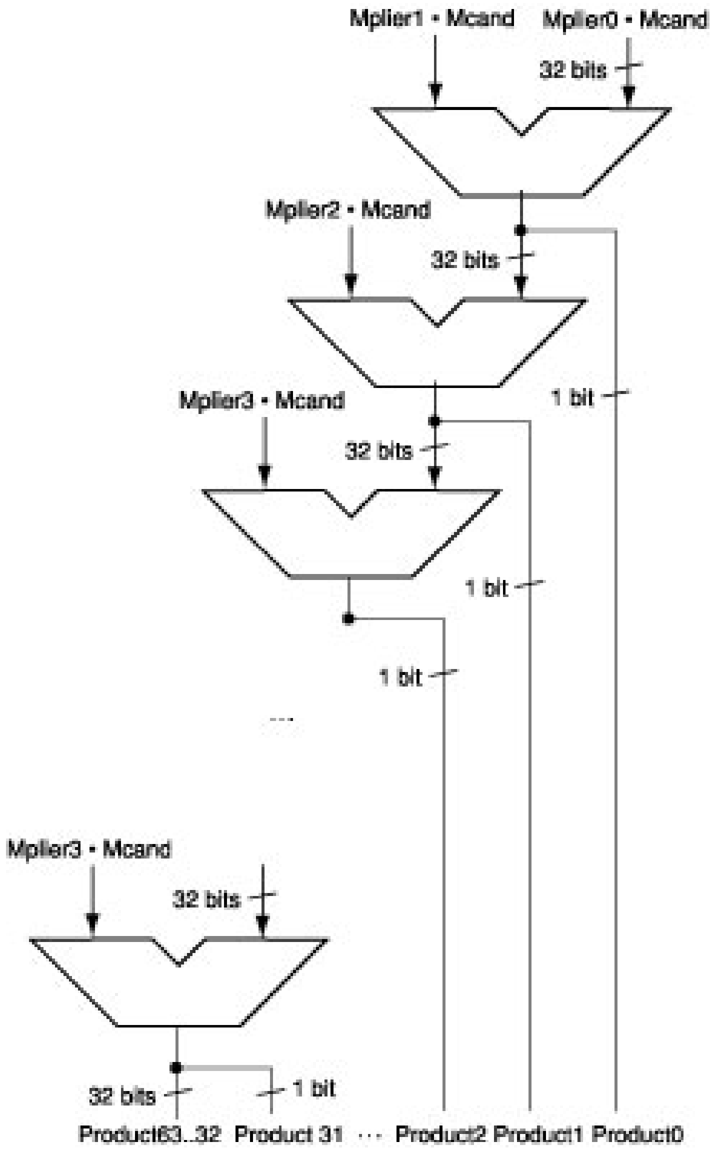
Multiplication Implementation: v1





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

Fast Multiplication Hardware: Unrolls the Loop



- Rather than using a single 32-bit adder 32 times, this hardware “unrolls the loop” to use 32 adders.
- Each adder produces a 32-bit sum and a carry out.
- 1st input: multiplicand ANDed with a multiplier bit.
- The LSB bit is a bit of the product.
- The carry out and the upper 31bits of the sum are passed along the next adder as 2nd input.

Multiplication: MIPS Instructions

- A pair of 32-bit registers Hi and Lo available for 64-bit product.
- Two instructions: mult and multu
- Both instructions ignore overflow.
- Pseudo-instructions mflo mfhi are used to place products into registers.

Division

- Example:

1001_{ten} Quotient

Divisor 1000_{ten} | 1001010_{ten} Dividend

$\underline{-\ \ 1000}$

10

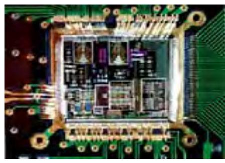
101

1010

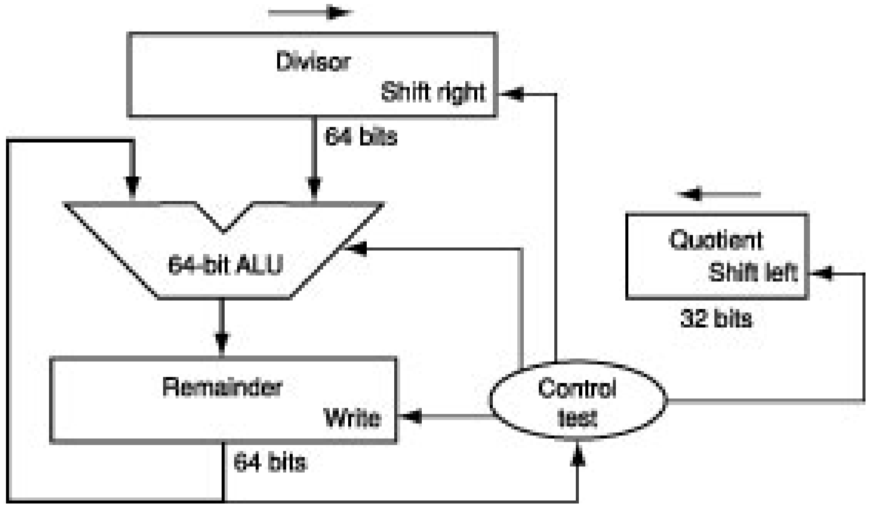
$\underline{-1000}$

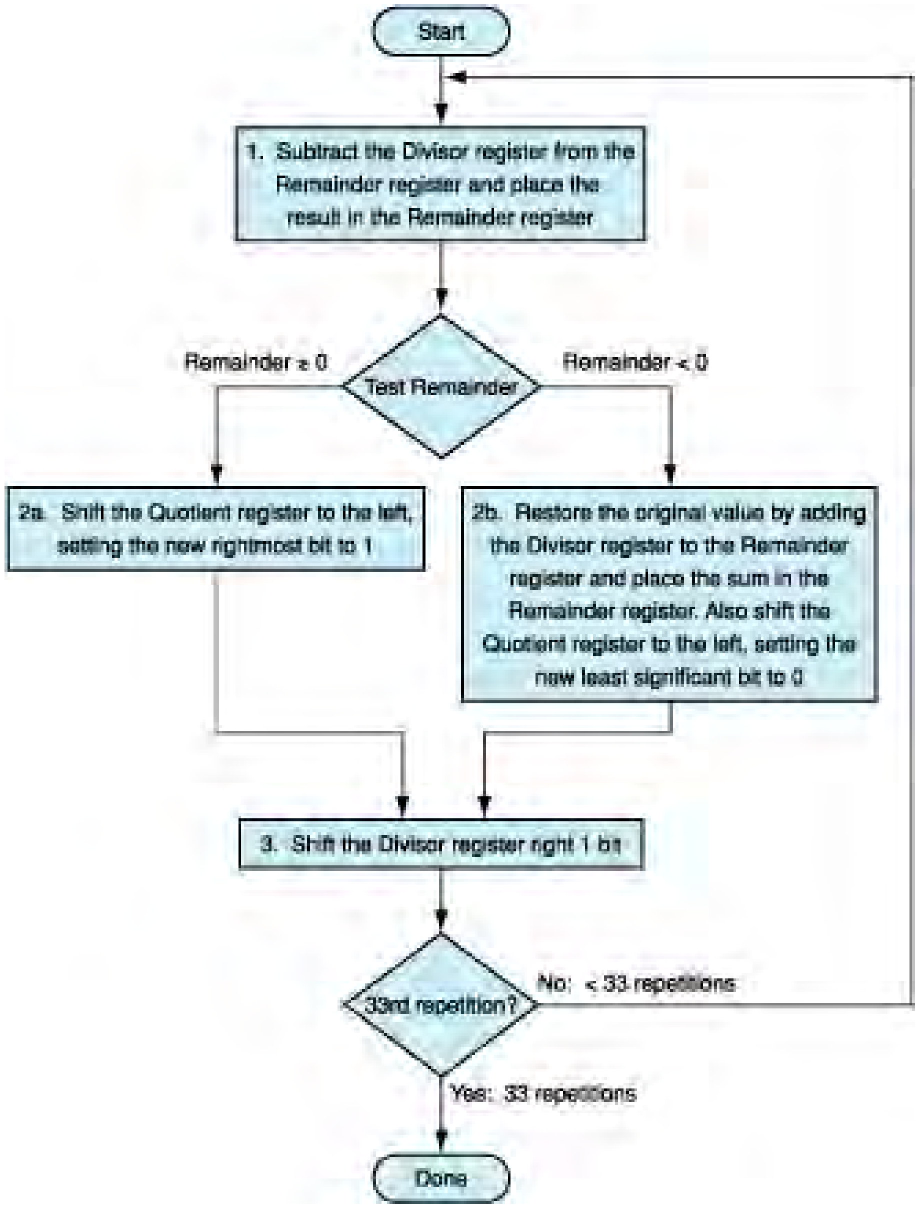
10_{ten} Remainder

- Observation : Dividend = Quotient x Divisor + Remainder



Division Implementation: v1





Start
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 and 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 right 1 bit
33rd repetition?
No: < 33 repetitions
Yes: 33 repetitions
Done

Division: MIPS Instructions

- The pair of 32-bit registers Hi and Lo are used.
- Two instructions: div and divu
- Hi contains the remainder and Lo contains the quotient after the divide instruction is complete.
- Pseudo-instructions mflo mfhi are used to place results into registers.

Floating Point : a brief look

- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., 0.000000001
  - very large numbers, e.g., 3.15576 E 10^{9}
- Representation:
  - sign, exponent, significand: $(-1)^{\text{sign}}$ X significant X 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 754 floating-point standard

- Leading "1" bit of significand is implicit.

- Exponent is "biased" to make sorting easier (as only positive numbers are to be dealt with)
  - all 0s is smallest exponent all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - summary: $(-1)^{\text{sign}}$ X (1+fraction) X $2^{\text{exponent} - \text{bias}}$

- Example:
  - decimal: $-0.75 = -3/4 = -3/2^2$
  - binary: $-0.11 = -1.1 \times 2^{-1} = -1.1 \times 2^{(126-127)}$
  - floating point: exponent = 126 = 01111110
  - IEEE single precision: 10111111010000000000000000000000

Float-Point Representation: Single Precision

- A floating-point value is represented in a single 32-bit word.
- Bias value for single precision is 127.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	Exponent								Fraction																						
1	8 bits								23 bits																						

- Decimal number -0.75 is represented as follows:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s	Exponent								Fraction																						
1	8 bits								23 bits																						

Float-Point Representation: Double Precision

- A floating-point value is represented in two 32-bit words.
- Bias value for single precision is 1023.
- Decimal number -0.75 is represented as follows:

Register - 1

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	1	1	1	1	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s	Exponent											Fraction																			
1	11 bits											20 bits																			

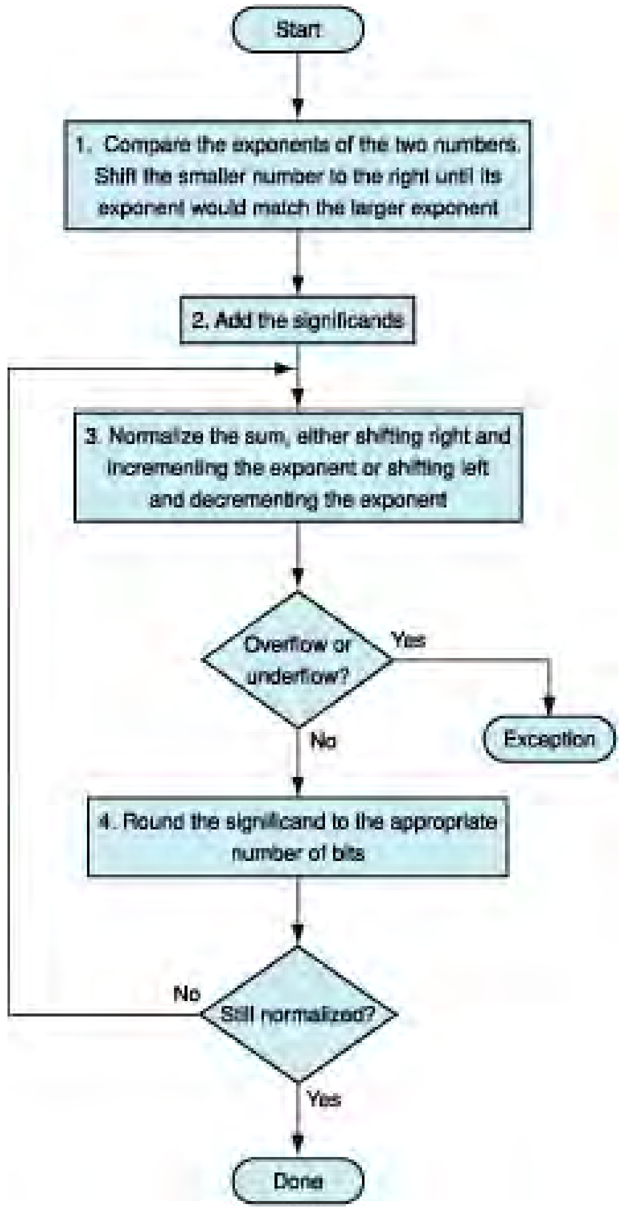
Register - 2

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
32 bits																															

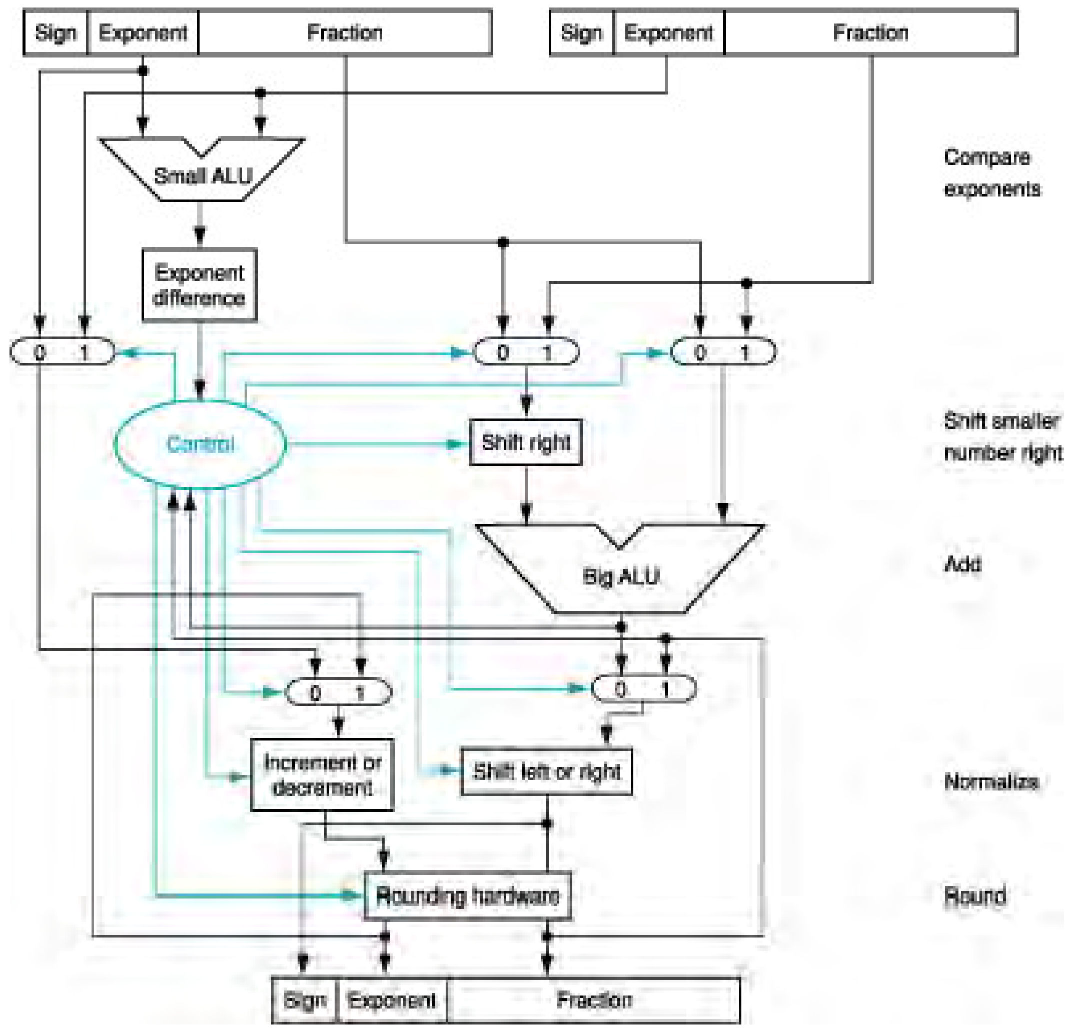
Floating Point Complexities

- Operations are somewhat more complicated
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
  - other complexities
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

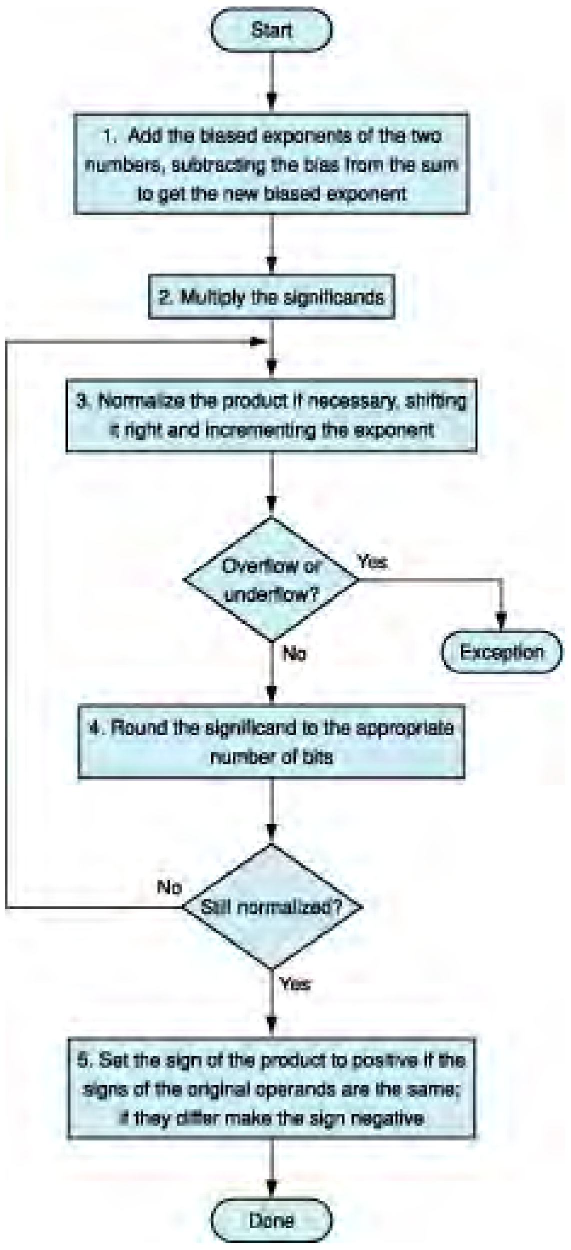
Floating Point Addition



Start
1. Compare the exponents of the two numbers. Shift the smaller number to the right until its exponent would match the larger exponent
2. Add the significands
3. Normalize the sum, either shifting right and incrementing the exponent or shifting left and decrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
Still normalized?
No
Yes
Done



Floating Point Multiplication


Start
1. Add the biased exponents of the two numbers, subtracting the bias from the sum to get the new biased exponent
2. Multiply the significands
3. Normalize the product if necessary, shifting it right and incrementing the exponent
Overflow or underflow?
Yes
Exception
No
4. Round the significand to the appropriate number of bits
No
Still normalized?
Yes
5. Set the sign of the product to positive if the signs of the original operands are the same; if they differ make the sign negative
Done

Floating-Point Instruction in MIPS

- Addition: add.s (single) and add.d
- Subtraction: sub.s and sub.d
- Multiplication: mul.s and mul.d
- Division: div.s and div.d

Summary

- Computer arithmetic is constrained by limited precision
- Bit patterns have no inherent meaning but standards do exist
  - two’s complement
  - IEEE 754 floating point
- Computer instructions determine “meaning” of the bit patterns
- Performance and accuracy are important so there are many complexities in real machines (i.e., algorithms and implementation).
- We are ready to move on (and implement the processor)