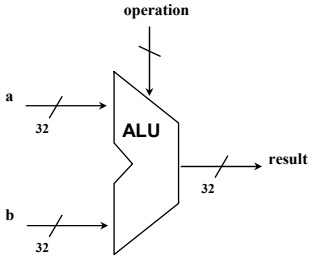
Chapter Four

4.1 Arithmetic

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



4.2 Numbers

- Bits are just bits (no inherent meaning)
  - conventions define relationship between bits and numbers
- Binary numbers (base 2): with each digit d of 0 and 1
  - 0000 0001 0010 0011 0100 0101 0110 0111 (32 bits)
    decimal value can be 0...2^n-1; see example in p.211
- Of course the real case is more complicated:
  - fixed point numbers that represent integer are finite: overflow may occur
  - Floating point numbers can represent real numbers
  - negative numbers: Sign Magnitude, One's Complement,Two's Complement
    e.g., no MIPS subi instruction; addi can add a negative number)
- How do we represent negative numbers?
  i.e., which bit patterns will represent which numbers?

Possible Negative Number Representations

- The concept of sign bit

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?
- The value represented by two's complement with n bits

$$b_{n-1} \times (-2^{n-1}) + \sum_{i=0}^{n-2} b_i \times 2^i$$

32 bits in MIPS

- **32 bit signed numbers:**

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

maxint
minint

- **unsigned integer: address**
  - **slt, slti ; unsigned version :sltu, sltiu**
  - **Example: P215**

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: "sign extension"**

```
0010  -> 0000 0010(+2)
1010  -> 1111 1010 (-6)
```

- **Fig.4.2 for MIPS assembly language with unsigned operation**

4.3 Addition & Subtraction

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

```
  0111        0111        0110
+ 0110      - 0110      - 0101
------      ------      ------
  1101        0001        0001
```

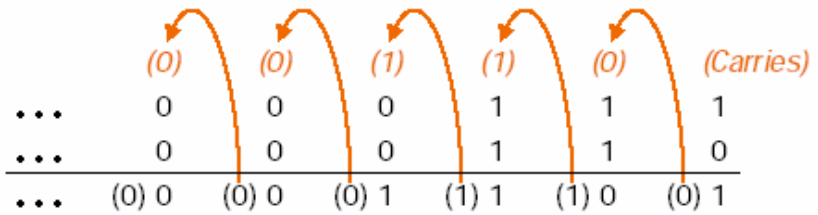
- **Two's complement operations easy**
  - **subtraction using addition of negative numbers: 7-6=7+(-6)**

```
  0111
+ 1010
------
  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"-> The overall result is wrong



Detecting Overflow

- No overflow when adding a positive and a negative number
- No overflow when signs are the same for subtraction

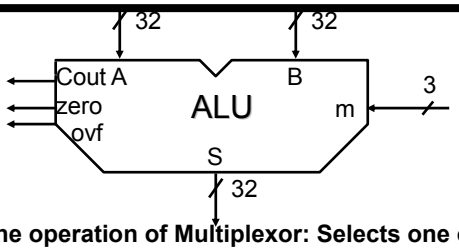
$S = A + B$ with overflow bit: $O = a_{n-1}\overline{b_{n-1}}\overline{s_{n-1}} + \overline{a_{n-1}}b_{n-1}s_{n-1}$

- 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
- Effects of Overflow
  - An exception (interrupt) occurs
    - Control jumps to predefined address for exception
    - Interrupted address is saved for possible resumption
  - Don't always want to detect overflow
    — new MIPS instructions: `addu, addiu, subu`
    note: `addiu` *still sign-extends!*
    note: `sltu, sltiu` *for unsigned comparisons*

4.4 Boolean Algebra & Gates

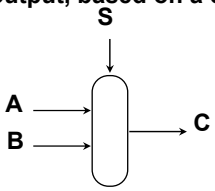
- Logic operation:
  - Shift left (<<) : sll, shift right (>>) : srl
  - bit-by-bit AND (&) : and, andi, Bit-by-Bit OR (|) (bitwise operation): or, ori
- A general logic or (control) problem : Consider a logic function with three inputs: A, B, and C.
  Output D is true if at least one input is true
  Output E is true if exactly two inputs are true
  Output F is true only if all three inputs are true
  - Show the truth table for these three functions.
  - Show the Boolean equations for these three functions.
  - Show an implementation consisting of inverters, AND, and OR (fig.4.8) gates or only NAND gates

4.5 An ALU (arithmetic logic unit)



A simplified ALU

- **Review the operation of Multiplexor: Selects one of the inputs to be the output, based on a control input : If s=0, c=a or s=1, c=b**



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

- **Let's build an 32-bits ALU to support the `andi`, `ori` and add instructions**
  - **we'll just build a 1 bit ALU, and use 32 of them**
  - **Let's first look at a 1-bit ALU for addition:**

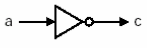
1. AND gate (c = a · b)

a	b	c = a · b
0	0	0
0	1	0
1	0	0
1	1	1

2. OR gate (c = a + b)

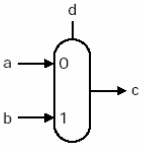
a	b	c = a + b
0	0	0
0	1	1
1	0	1
1	1	1

3. Inverter (c = ā)



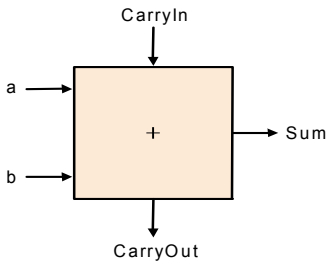
a	c = ā
0	1
1	0

4. Multiplexor
(if d = = 0, c = a;
else c = b)



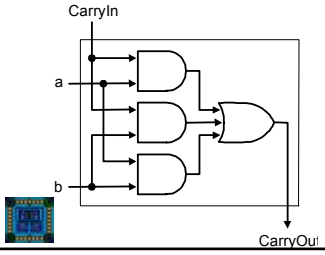
d	c
0	a
1	b

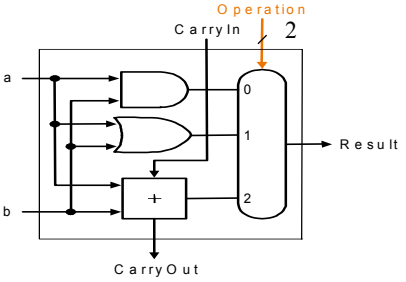
1-bit Adder and 1-bit ALU



CarryIn	A	B	AB[G]	A+B[P]	A⊕B	SUM	CarryOut
0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	0
0	1	0	0	1	1	1	0
0	1	1	1	1	0	0	1
1	0	0	0	0	0	1	0
1	0	1	0	1	1	0	1
1	1	0	0	1	1	0	1
1	1	1	1	1	0	1	1

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

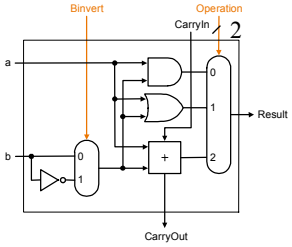


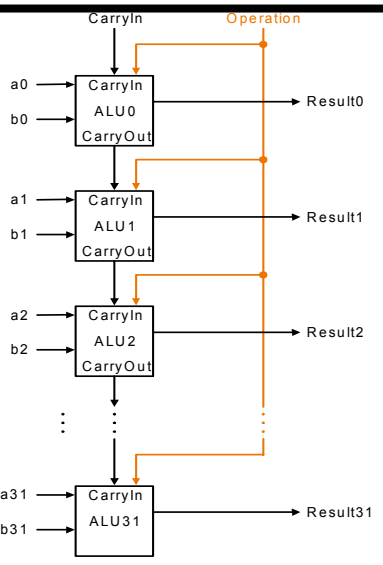


Building a 32 bit ALU

- **A 32 bit ALU constructed from 32 1-bit ALUs**
- **What about subtraction (a – b) ?**
  - **Two's complement approch: just negate b and add.**
  - **How do we negate?**

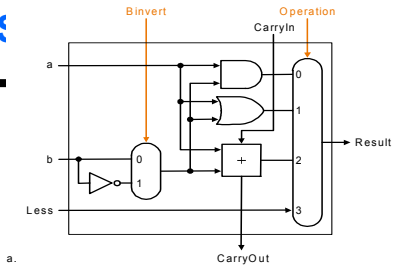
A very clever solution:

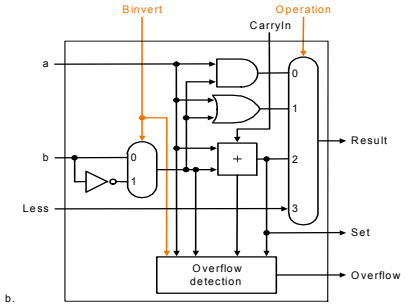


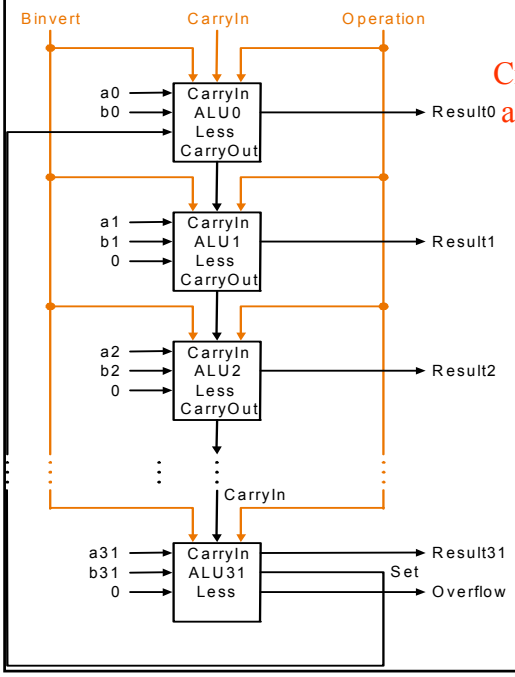


Tailoring the ALU to the MIPS

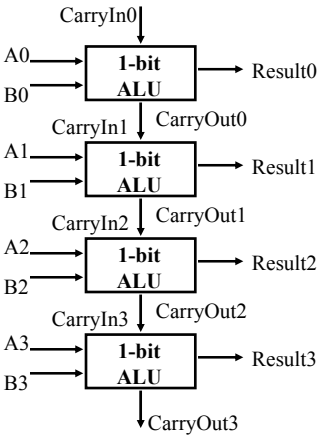
- Need to support the set-on-less-than instruction (slt)
  - remember: slt is an arithmetic instruction
  - produces a 1 if rs < rt and 0 otherwise
  - use subtraction: (a-b) < 0 implies a < b
- Need to support test for equality (beq $t5, $t6, $t7)
  - use subtraction: (a-b) = 0 implies a = b







Critical Path of n-bit Rippled-carry adder is n*Tcarryout

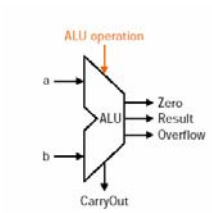


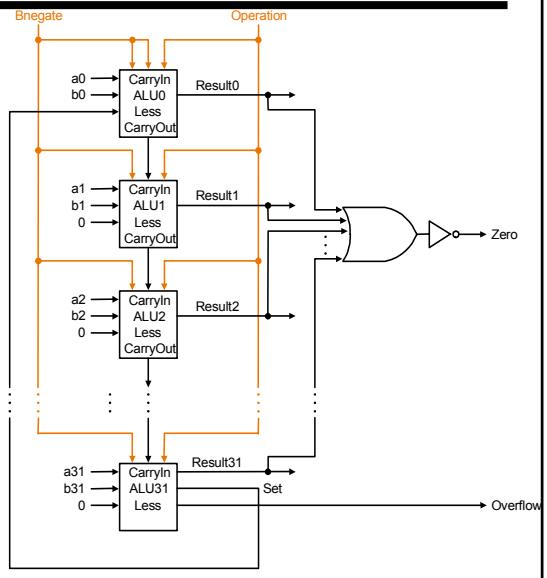
Test for equality

- **Notice of control lines:**

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

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





Conclusions

- **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")**
  - **In today's 0.18 um CMOS technology: 2 inputs NAND (NOR)gate:0.14 ns**
- **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**

Problem: ripple carry adder is slow-> Carry-lookahead adder

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

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

$c_1 = b_0c_0 + a_0c_0 + a_0b_0$

$c_2 = b_1c_1 + a_1c_1 + a_1b_1$

$c_3 = b_2c_2 + a_2c_2 + a_2b_2$

$c_4 = b_3c_3 + a_3c_3 + a_3b_3$

Not feasible! Why?

- Motivation:
  - If we didn't know the value of carry-in, what could we do?
  - When would we always generate a carry? $g_i = a_i b_i$
  - When would we propagate the carry? $p_i = a_i + b_i$
- Did we get rid of the ripple?

$c_1 = g_0 + p_0c_0$

$c_2 = g_1 + p_1c_1$

$c_3 = g_2 + p_2c_2$

$c_4 = g_3 + p_3c_3$

$C_1 = g_0 + p_0c_0$

$C_2 = g_1 + p_1g_0 + p_1p_0c_0$

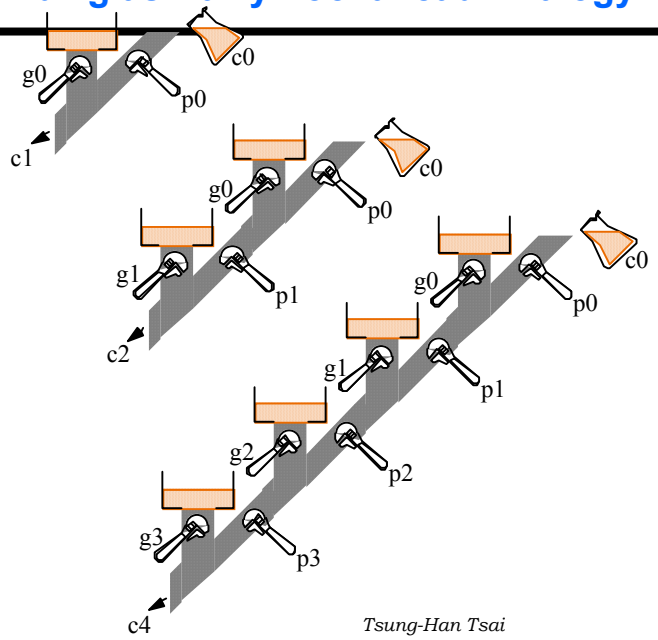
$C_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_0$

$C_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_0$

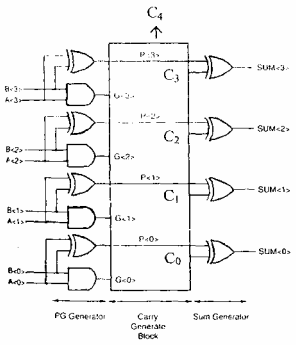
$= g_3 + p_3(g_2 + p_2(g_1 + p_1(g_0 + p_0c_0)))$

Feasible! Why?

Plumbing as Carry Lookahead Analogy



One-Level Carry-lookahead adder



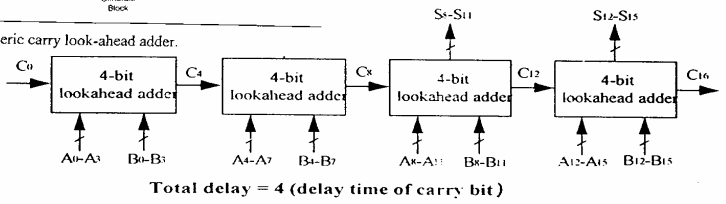
$$C_{i+1} = a_i b_i + c_i(a_i + b_i) = g_i + c_i p_i$$

where $p_i = a_i + b_i$ ...*propagation* signal,
$g_i = a_i b_i$ ...*generation signal*

so $c_{i+1} = g_i + p_i c_i = \ldots\ldots = g_i + p_i g_{i-1} + \ldots\ldots + p_i p_{i-1} \ldots\ldots p_0 c_0$

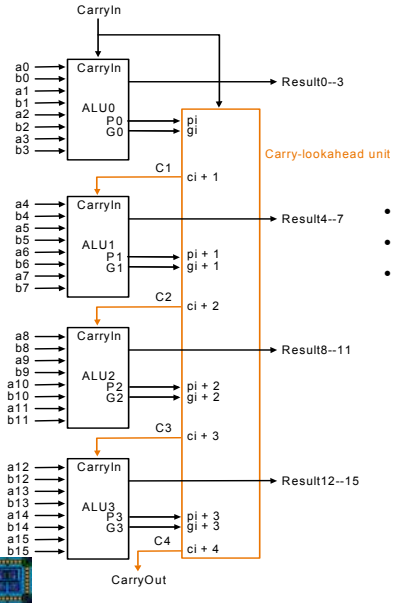
and $s_i = c_i \oplus a_i \oplus b_i = c_i \oplus p_i$ ($p_i = a_i \oplus b_i$ instead of $a_i + b_i$)

Figure 8.15 Generic carry look-ahead adder.



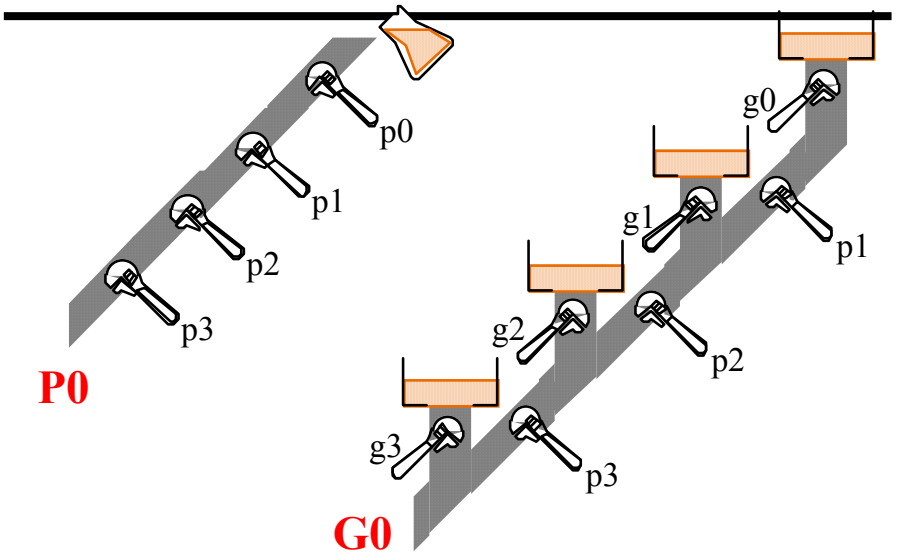
16-bits one-level carry look-ahead adder.

Use principle to build bigger adders



- **Can't build a 16 bit adder this way... (too big)**
- **Could use ripple carry of 4-bit CLA adders**
- **Better: use the CLA principle again!**

2nd level Carry, Propagate as Plumbing



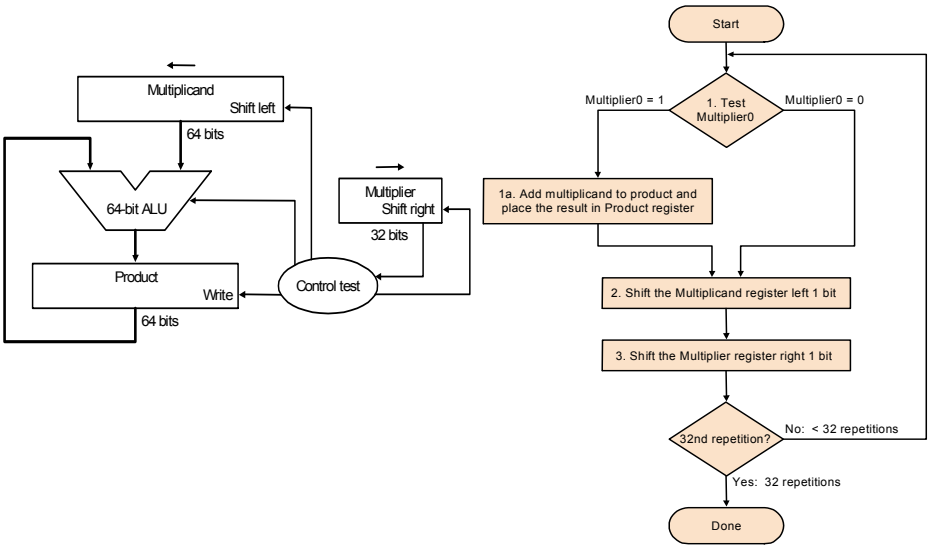
4.6 Multiplication

- More complicated than addition
  - accomplished via several shifting and addition oprations
- More cycle time and more area as compared to adder
- Let's look at 3 versions based on grade school algorithm

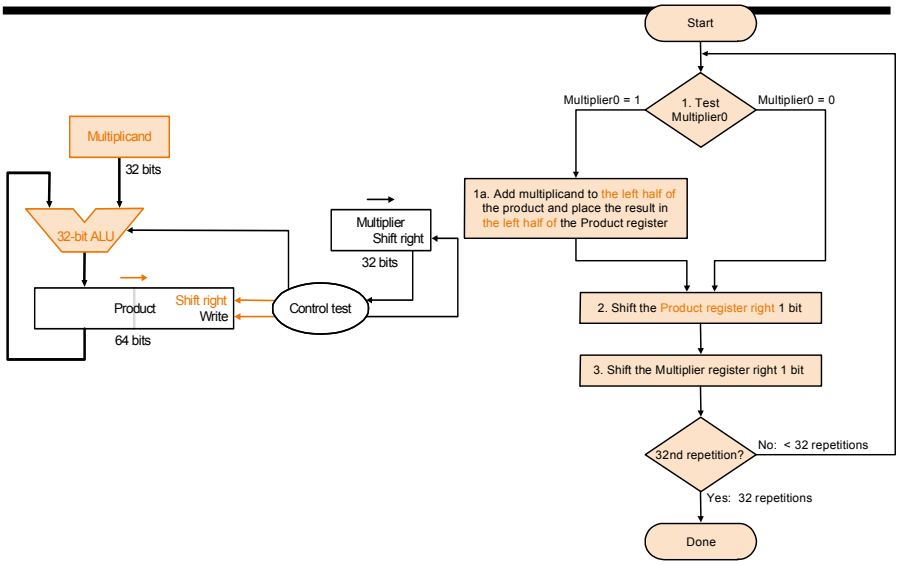
```
          0010   (multiplicand)
        x 1011   (multiplier)
          0010
         0010
        0000
       0010
       0010110
```

- Negative numbers: convert and multiply
  - there are better techniques, we won't look at them

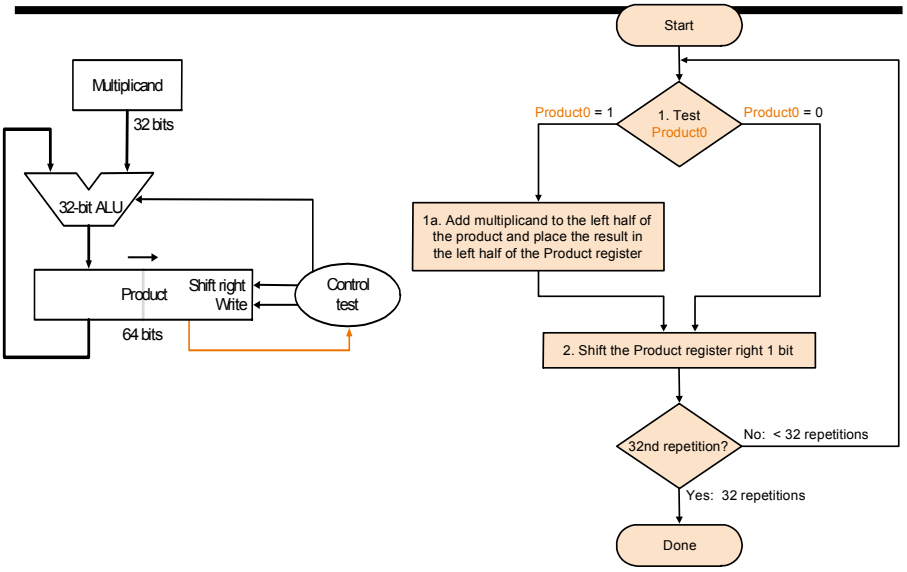
Multiplication: Implementation



Second Version



Final Version



Divider

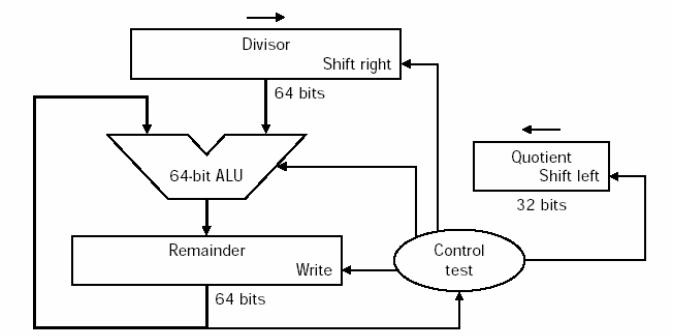


Fig 4.39

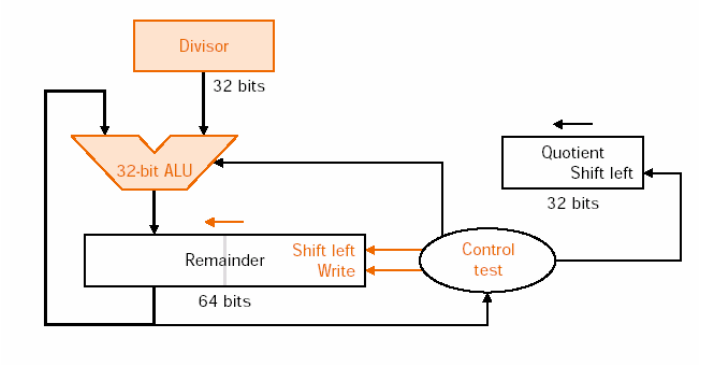
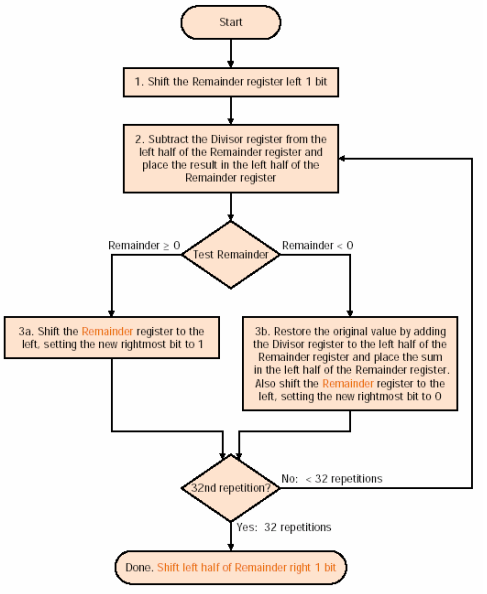
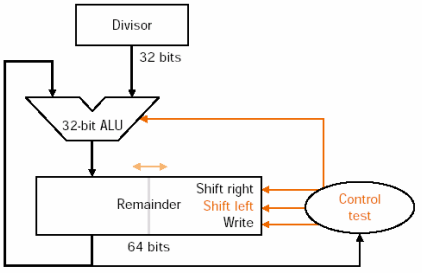


Fig 4.40 & 4.41



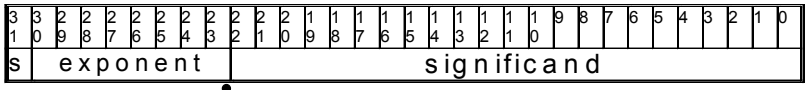


Floating Point (a brief look)

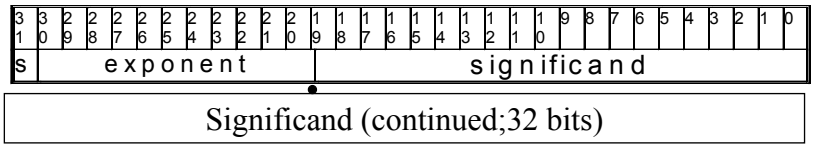
- We need a way to represent
  - numbers with fractions, e.g., 3.1416
  - very small numbers, e.g., .000000001
  - very large numbers, e.g., $3.15576 * 10^9$
- Scientific notation:
  - Base 10: $1.0*10^{-9}$, $0.1*10^{-8}$, $3.15576 * 10^9$, $31.5576*10^8$
    - No leading 0s is called a normalized number
- Scientific notation in binary number: $a = 0.11*2^1$ (= 1.5) = $(-1)^S*F*2^E$, where s = 0, F=0.11, E = 1 and . is called binary point; (sign bit;F;E)
  - sign, exponent (E), significand (F): $(-1)^{sign}$ *significand* $2^{exponent}$
  - $a = 0.11*2^1$ is called the floating point notation where floating point means the position of floating point is not fixed.
  - The more bits for significand gives more accuracy
  - more bits for exponent increases range

IEEE 754 floating-point standard

- IEEE 754 floating point standard:
  - single precision: 1 bit for sign, 8 bit exponent, 23 bit significand



  - double precision: 1 bit for sign, 11 bit exponent, 52 bit significand



- Leading "1" bit of significand is implicit
- Exponent is "biased" to make sorting easier
  - all 0s is smallest exponent all 1s is largest
  - bias of 127 for single precision and 1023 for double precision
  - range of floating-point: $(-1)^{sign} \times (1+\text{significand})_2 \times 2^{exponent - bias}$
  - If significant equals s1s2s3,… -> $(-1)^{sign} \times (1+s1 \times 2^{-1} + s2 \times 2^{-2} \ldots) \times 2^{exponent - bias}$
- Overflow and underflow can still occur

IEEE 754 Floating-Point Representation

- **Example:**
  - **decimal: -.75 = -3/4 = -3/2²**
  - **binary: -.11 = -1.1 x 2⁻¹**
  - **floating point: exponent = 126 = 01111110**
  - **IEEE single precision:** `10111111010000000000000000000000`
- **Example P.280**
- **Floating-point addition: Example: the four steps in P.281**

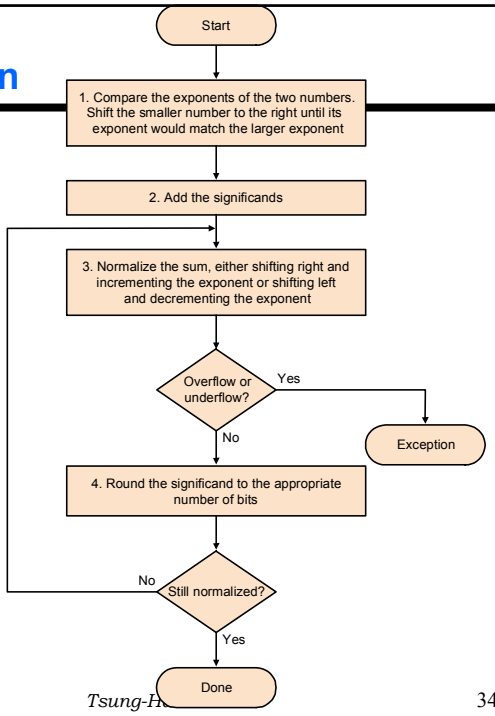
  - $x = (s,F,E) = (-1)^s(1+F)*r^E$ with $\frac{1}{r} \le |F| \le 1 - r^{-p} < 1$, $F : p$ (23) bits

    $0 \le |E| \le r^q - 1$, $E : q$ (8) bits

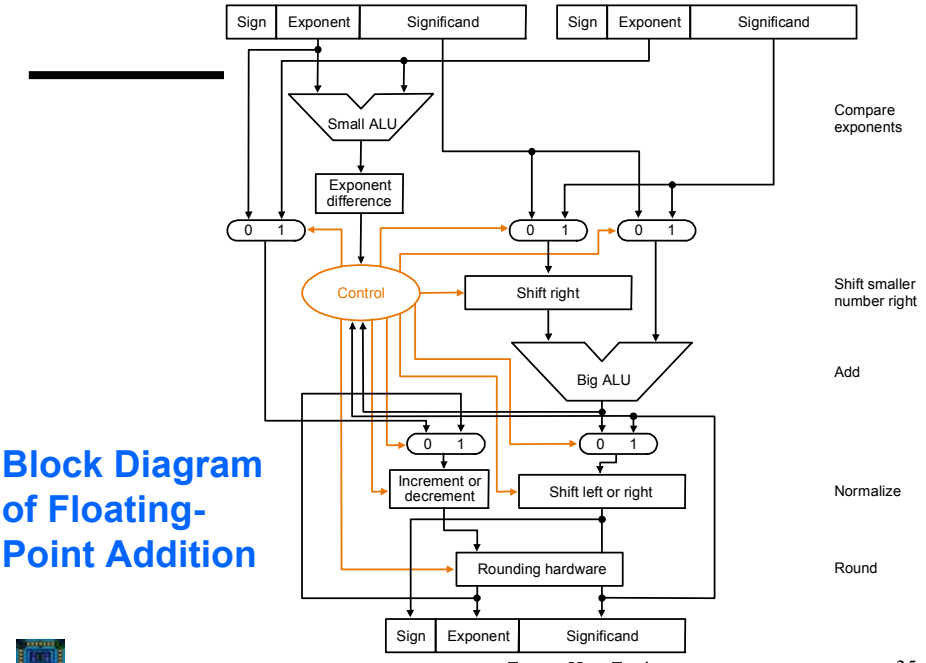
  - Addition : $(s_1,F_1,E_1) \pm (s_2,F_2,E_2) = \begin{cases} (((-1)F_1 \pm (-1)F_2 * r^{-(E_1-E_2)}), E_1) & \text{if } E_1 > E_2 \\ ((F_1 * r^{-(E_1-E_2)} \pm F_2), E_2) & \text{if } E_1 \le E_2 \end{cases}$

    ☑ radix point of x_1, x_2 must be aligned => shifting the mantissa with a smaller exponent $|e_1-e_2|$ places to the right.
    ☑ Normalize the sum
    ☑ Round the significant to the appropriate number of bits

Flow Chart of Floating-Point Addition



Block Diagram of Floating-Point Addition



Floating point Multiplication

- FLP multiplication and division :

$$(s, F, E) = (s_1, F_1, E_1) * (s_2, F_2, E_2) = (s_1 \oplus s_2, F_1 * F_2, E_1 + E_2 - bias)$$
$$(s, F, E) = (s_1, F_1, E_1) / (s_2, F_2, E_2) = (s_1 \oplus s_2, F_1 / F_2, E_1 - E_2 + bias)$$

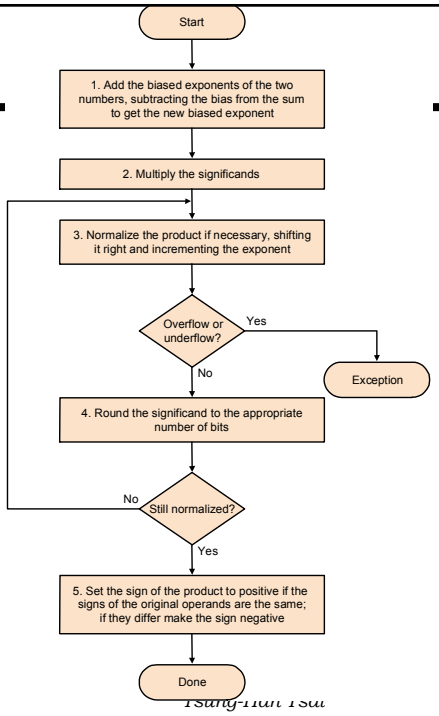
  - ⇨ $F_1 * F_2$ and $E_1 + E_2$ -bias can be executed simultaneously.
    - ➠ The same amount of execution time as corresponding FXP operation.

$$\frac{1}{r^2} \le |F_1 * F_2| < 1 \begin{cases} \frac{1}{r} \le |F_1 * F_2| < 1 : ok \\ \frac{1}{r^2} \le |F_1 * F_2| < \frac{1}{r} \Rightarrow (r * F, E - 1) \end{cases}$$

$$\frac{1}{r} \le |F_1 / F_2| < r \begin{cases} \frac{1}{r} < |F_1 / F_2| < 1 : ok \quad when\ F1 < F2 \\ 1 \le |F_1 / F_2| < r : (F * r^{-1}, E + 1) \quad when\ \ F1 \ge F2 \ne 0 \end{cases}$$

- **Example: Step 1 to Step 5 in p.283 to P.286**

Flow Chart of Floating-Point Multiplication



MIPS Floating-point Operation

- **MIPS supports the IEEE 754 single-precision and double-precision formats**
  - **add.s, add.d; sub.s, sub.d**
  - **mul.s, mul.d;div.s,div.d**
  - **Comparison Ins: c.x.s or c.x.d where x can be eq, neq, lt, le, gt, ge**
    - Sets a bit to true or false
  - **Floating branch: true (bclt) and false (bclf)**
- **Have separate 32 floating-point register: $f0, $f1, $f2…$f31**
  - **Each has 32 bits andare used in pairs for double precision numbers**
  - **Loads and stores for floating-point registers: lwc1 and swc1**
  - **Example: P288**
- **Summary: Fig.4.47 on P.291**
- **Example: P.293**

Floating Point Complexities

- Operations are somewhat more complicated than integer operation
- In addition to overflow we can have “underflow”
- Accuracy can be a big problem
  - IEEE 754 keeps two extra bits, guard and round: example in P.297
  - four rounding modes
  - positive divided by zero yields “infinity”
  - zero divide by zero yields “not a number”
  - other complexities
- C data types and MIPS Ins for them : see table on p.299
- Implementing the standard can be tricky
- Not using the standard can be even worse
  - see text for description of 80x86 and Pentium bug!

Chapter Four 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)