Arithmetic

Introduction

- Representation of numbers
- Limits on number size
- Fractions and real numbers
- Arithmetic operations

Signed and unsigned numbers

- Radix or base of a number system
  - Base is indicated by subscript to actual number
- If b is the radix, the value of ith digit d is given by: $d \times b^i$, with i starting at zero and increasing from right to left
- Example: 1011_2

$$\begin{aligned} 1011_2 &= ((1 \times 2^0) + (1 \times 2^1) + (0 \times 2^2) + (1 \times 2^3))_{10} \\ &= ((1 \times 1) + (1 \times 2) + (0 \times 4) + (1 \times 8))_{10} \\ &= (1 + 2 + 0 + 8)_{10} \\ &= 11_{10} \end{aligned}$$

- Bits are numbered $0, 1, 2, \ldots$ from right to left in a word, with bit 0 being the least significant bit (LSB), and the highest numbered bit being the most significant bit (MSB)
- With n bits, the maximum number that can be represented is given by $2^n - 1$
- ASCII vs binary numbers
  - Internal representation
  - Leading zeroes are not generally shown
  - Overflow as a result of arithmetic operation
    - Overflow handled by OS or application
  - Sign and magnitude representation for negative numbers
    - Where to put sign bit? MSB or LSB?
    - Adders may need an extra step to set the sign bit
    - Both positive and negative zero
  - 1's complement
    - Flip all the bits
    - Still have positive and negative zero
  - 2's complement
    - Used in all processors designed today
    - Sum of an n-bit number and its negative is 2^n
    - Leading bit (MSB) as 1 indicates a negative number
    - Positive numbers are represented as normal with leading bit zero
    - Only one zero representation

  - For 8-bit numbers, the range is from +127 to -128, with just one 0
  - The decimal representation for a number is found by

$$(-1) \times x_{n-1} \times 2^{n-1} + x_{n-2} \times 2^{n-2} + x_{n-3} \times 2^{n-3} + \cdots + x_0 \times 2^0$$

    - Only the MSB is multiplied by -1
  - Example: 11110100_2 (original number is 8-bit)

$$\begin{aligned} 11110100_2 &= ((-1 \times 2^7) + (1 \times 2^6) + (1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0))_{10} \\ &= ((-1 \times 128) + (1 \times 64) + (1 \times 32) + (1 \times 16) + (0 \times 8) + (1 \times 4) + (0 \times 2) + (0 \times 1))_{10} \\ &= (-128 + 64 + 32 + 16 + 0 + 4 + 0 + 0)_{10} \\ &= -12_{10} \end{aligned}$$

- Signed load operation (load word)
  - Repeatedly copy the sign bit to fill the rest of the register
  - Also known as *sign extension*
  - Unsigned load simply fills the left of data with 0s
  - `lb` treats the byte as a signed number and performs sign extension into the register
  - `lbu` (load byte unsigned) works with unsigned integers
- Overflow on 2's complement numbers
  - Overflow occurs when the MSB is not the same as what it would be if we had infinite bits
- Memory addresses are always unsigned; same with some other data element types
  - C has `int` and `unsigned int`
  - Depending on our intention, the number $F4_{16}$ may be less than (-12_{10}) or greater than (244_{10}) 0
  - MIPS handles this distinction by providing two versions of *set on less than* instruction
    1. `slt` and `slti` work with signed integers
    2. `sltu` and `sltiu` work with unsigned integers
- Example
  - Signed vs unsigned comparison
    - Let `$s0` contain the number `FFFF FFFF`$_{16}$, and register `$s1` contain the number `0000 0001`$_{16}$
    - Execute the following instructions

```
slt     $t0, $s0, $s1           # signed comparison
sltu    $t1, $s0, $s1           # unsigned comparison
```

    - `$t0` has value 1 while `$t1` has 0
- Finding the 2's complement representation
  - Find the representation of positive number
  - Flip the bits (change 1 to 0 and 0 to 1)
  - Add 1
  - Representation of -42_{10} in 8-bits
  - $42_{10} = 00101010_2$
  - Flip the bits: 11010101_2
  - Add 1: 11010110_2
- We add 1 to account for the fact that $x + \bar{x} \equiv -1$
- Verify the result above by negating the number again
  - Flip the bits: 00101001_2
  - Add 1: 00101010_2
- Operations involving numbers with different bit size representation

    - To add a 16-bit number to a 32-bit number, perform sign extension
    - Simply replicate the MSB into the new bits of 32-bit equivalent of 16-bit number
- Binary to hex conversion

Addition and subtraction

- Addition is performed bit-by-bit with carry being passed to next digit to the left
- Subtraction is performed by adding two numbers
- Example: Add 6_{10} to 7_{10}

	7_{10}	00000111_2
+	6_{10}	00000110_2
=	13_{10}	00001101_2

- Example: Subtract 6_{10} from 7_{10}, or add -6_{10} to 7_{10}

	7_{10}	00000111_2
+	6_{10}	11111010_2
=	1_{10}	00000001_2

- Overflow
  - No overflow when adding numbers of *different* sign (effectively, subtracting)
  - No overflow when subtracting numbers of *same* sign
  - Overflow if addition of two positive numbers gives a negative number, or addition of two negative numbers gives a positive number
    - The sign bit is set with the value of the result instead of the sign
  - In MIPS
    - `add`, `addi`, and `sub` cause exceptions on overflow
    - `addu`, `addiu`, and `subu` do not cause exceptions on overflow
  - C ignores overflows; MIPS compiler account for this by using the unsigned version of instructions
- Exception (*should not be called interrupt*)
  - Unscheduled procedure call
  - MIPS has a register called *exception program counter* (EPC)
    - Contains address of the instruction that caused exception
  - Instruction `mfc0` (*move from system control*) copies `epc` into a general purpose register to provide for return to offending instruction
    - Problem: The contents of general purpose register will be destroyed, and we will not have the actual value in *all* the registers before the problem instruction when we return
    - Problem is solved by using two registers `$k0` and `$k1` for the OS and their contents are not restored on exceptions

Logical operations

- May need to operate on groups of bits within a word
- Instructions to simplify the packing and unpacking of bits in a word

- *Shift* instructions
  - Move the bits in a word to the left or right
  - Original contents of a 8-bit register: `0010 1010`
  - Contents after a shift left by 2: `1010 1000`
  - MIPS instructions are called *shift left logical* (`$sll`) and *shift right logical* (`srl`)
  - Example:

```
sll     $t2, $s0, 8          # $t2 = $s0 << 8;
```

	op	rs	rt	rd	shamt	funct
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
`sll`	0x00	0x00	0x10	0x0A	0x08	0x00

  - The `shamt` field in the *R*-format stands for *shift amount*
  - The encoding of `sll` is 0 in both the `op` and `funct` fields; `rs` is unused
  - Why do we have only five bits assigned to shift operation?
- Bitwise AND and OR
  - Useful to extract the contents of a certain number of bits
  - Let register `$t1` contain `0010 1010`$_2$ and register `$t2` contain `0011 0000`; then, the instruction

```
and     $t0, $t1, $t2                # $t0 = $t1 & $t2
```

    yields in register `$t0` the value `0010 0000`$_2$
  - `and` is usually applied with a mask to extract a set of bits
  - `or` is the dual to `and`
  - Applying `or` to the registers defined above

```
or      $t0, $t1, $t2                # $t0 = $t1 | $t2
```

    we have in register `$t0` the value `0011 1010`$_2$
- C bit fields
  - Look at the parsing of datagram

Constructing an Arithmetic Logic Unit (ALU)

- Part of a computer that performs arithmetic and logic operations
- 32-bit operands in MIPS ALU
- Four building blocks
  1. AND gate
     - `c = a & b`
  2. OR gate
     - `c = a | b`
  3. Inverter, or NOT
     - `c = ~a`
  4. Multiplexer, or switch
     - `c = ( ! d ) ? a : b`

- Building a 1-bit ALU
  - Logical operations map directly into hardware
  - Look at Figure 4.9 – logical unit for AND and OR
  - Single bit adder
    - Two inputs for operands
    - Third input for carry-in
    - Single bit output for sum
    - Second output for carry-out
  - Make an Input/Output table

Inputs			Outputs		Operation
a	b	carry-in	carry-out	sum	
0	0	0	0	0	0+0+0 = 00
0	0	1	0	1	0+0+1 = 01
0	1	0	0	1	0+1+0 = 01
0	1	1	1	0	0+1+1 = 10
1	0	0	0	1	1+0+0 = 01
1	0	1	1	0	1+0+1 = 10
1	1	0	1	0	1+1+0 = 10
1	1	1	1	1	1+1+1 = 11

  - Full adder, or (3,2) adder – three inputs, two outputs
  - Half adder, or (2,2) adder – two inputs, two outputs
  - The outputs can be expressed as logical equations using truth tables
  - Values of inputs when carry-out is a 1

Inputs		
a	b	carry-in
0	1	1
1	0	1
1	1	0
1	1	1

  - Logical equation for carry-out

```
carry-out = ( b & carry-in) | ( a & carry-in ) | ( a & b ) | ( a & b & carry-in)
```

  - The above equation can be simplified to

```
carry-out = ( b & carry-in) | ( a & carry-in ) | ( a & b )
```

  - Figure 4.13
  - Sum bit is set when any of the three input bits is 1, with the other two being 0; or when all three input bits are 1
  - Logical equation for sum is

```
sum = ( a & ~b & ~carry-in ) | ( ~a & b & ~carry-in ) | ( ~a & ~b & carry_in )
        | ( a & b & carry-in )
```

- A 32-bit ALU
  - A 32-bit ALU is created by connecting adjacent 1-bit ALUs
  - A single carry-out of the LSBs ripples through the 1-bit ALUs
  - *Ripple carry adder*

  - Subtraction is achieved by adding the 2's complement number
    * Add an inverter (a multiplexer) that will invert the signal on b and send a 1 as carry-in bit for the LSB
    * Effectively, we have
    $$a + \bar{b} + 1 = a - b$$
    * Figure 4.16

- Tailoring the 32-bit ALU to MIPS
  - Most of the design is already done but we need to implement the `slt` instruction
  - `slt` is defined as

    ```
    ( rs < rt ) ? 1 : 0
    ```

  - All bits (except for LSB) are set to 0, with LSB value based on the comparison result
  - Expand the 3-input multiplexor of Figure 4.16 to add an input for `slt`; new input is called `less` and used only for `slt` (Figure 4.17)
    * Top drawing (Fig 4.17) shows the new 1-bit ALU with expanded multiplexor
    * Connect 0 to `less` input for most significant 31 bits of ALU
    * Compare and set the LSB for `slt` instruction
    $$\begin{aligned} (a-b) < 0 &\Rightarrow ((a-b)+b) < (0+b) \\ &\Rightarrow a < b \end{aligned}$$
    * Sign bit indicates the result
      · 1 means true, or $a < b$
      · 0 means false, or $a \not< b$
    * Connect the sign bit from adder output to LSB
      · Result out from MSB in ALU for `slt` is not the output of adder (Fig 4.17)
      · ALU output for `slt` is input value `less`
      · Need a new 1-bit ALU for MSB that has one extra output bit for adder output
      · New output line is called `set` and is used only for `slt`
      · Special ALU for MSB requires the overflow detection logic associated with that bit
    * For subtraction, we set both `carry-in` and `binvert` to 1; for adds or logical, both lines are 0
      · Both lines can be combined into a single control line called `bnegate`
  - Conditional branch instruction
    * Branch either if two registers are equal, or they are unequal
    * Test equality by subtracting one register from the second and see if result is 0
    * Testing for 0
      · OR all outputs and send the signal through an inverter
    $$0 \equiv \neg(r_{31}|r_{30}|\cdots|r_2|r_1|r_0)$$
  - Revised 32-bit ALU in Figure 4.19
  - Control lines (`bnegate` and `operation`) tell the ALU to perform add, subtract, AND, OR, or `slt`
  - Complete ALU can be represented as in Figure 4.21

- Carry lookahead
  - Carry has to be *rippled* in sequence from LSB to MSB, resulting in a slow sequence of operations to add two 32-bit numbers

- Improve the speed by anticipating the carry
- Fast carry using "infinite" hardware
  - Only external inputs are two operands and carry-in to LSB of adder
  - In theory, carry-in to all remaining bits of adder can be calculated in just two levels of logic
  - Carry-in for bit 2 (c_2) is the same as carry-out of bit 1

$$c_2 = (b_1 \& c_1) | (a_1 \& c_1) | (a_1 \& b_1)$$

  - Similarly, c_1 is defined by

$$c_1 = (b_0 \& c_0) | (a_0 \& c_0) | (a_0 \& b_0)$$

  - Substituting c_1's value in first equation, we have

$$c_2 = (a_1 \& a_0 \& b_0) | (a_1 \& a_0 \& c_0) | (a_1 \& b_0 \& c_0) | (b_1 \& a_0 \& b_0) | (b_1 \& a_0 \& c_0) | (b_1 \& b_0 \& c_0) | (a_1 \& b_1)$$

  - As the number of bits increases, complexity becomes unmanageable, making this technique prohibitively expensive for fast adders
- Fast carry using first level of abstraction: Propagate and Generate
  - Carry-lookahead adder
  - Relies on levels of abstraction in its implementation
  - Original equation

$$\begin{aligned} c_{i+1} &= (a_i \cdot b_i) + (a_i \cdot c_i) + (b_i \cdot c_i) \\ &= (a_i \cdot b_i) + (a_i + b_i) \cdot c_i \end{aligned}$$

  - For c_2, we have

$$\begin{aligned} c_2 &= (a_1 \cdot b_1) + (a_1 + b_1) \cdot c_1 \\ &= (a_1 \cdot b_1) + (a_1 + b_1) \cdot ((a_0 \cdot b_0) + (a_0 + b_0) \cdot c_0) \end{aligned}$$

  - Product (AND) and sum (OR) of a_i and b_i above are known as *generate* (g_i) and *propagate* (p_i)

$$\begin{aligned} g_i &= a_i \cdot b_i \\ p_i &= a_i + b_i \\ c_{i+1} &= g_i + p_i \cdot c_i \end{aligned}$$

  - Generate
    - Let g_i be 1
    - Then, we have

$$\begin{aligned} c_{i+1} &= g_i + p_i \cdot c_i \\ &= 1 + p_i \cdot c_i \\ &= 1 \end{aligned}$$

    - Adder *generates* a carry-out independent of the value of carry-in
  - Propagate
    - Let p_i be 1 and g_i be 0
    - If g_i is 1, the generate-case applies
    - Then, we have

$$\begin{aligned} c_{i+1} &= g_i + p_i \cdot c_i \\ &= 0 + 1 \cdot c_i \\ &= c_i \end{aligned}$$

  - The value of carry-in is *propagated* to carry-out
  * Carry-out is a 1 if g_i is 1, or both p_i and carry-in are 1
  * A carry-out can be made true by a generate far away if all the propagates between them are true
  * Propagate and generate become first levels of abstraction
  * Example with four bits

$$\begin{aligned}
c_1 &= g_0 + p_0 \cdot c_0 \\
c_2 &= g_1 + p_1 \cdot c_1 \\
&= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\
c_3 &= g_2 + p_2 \cdot c_2 \\
&= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \\
c_4 &= g_3 + p_3 \cdot c_3 \\
&= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0
\end{aligned}$$

  * Still considerable logic

- Fast carry using second level of abstraction
  * Use the above 4-bit adder with carry-lookahead logic as a building block
  * Connect a set of them in ripple carry-adder fashion to create a 16-bit adder
  * A second option will be to create a second level of carry-lookahead abstraction, resulting in *super-propagate* signal

$$\begin{aligned}
P_0 &= p_3 \cdot p_2 \cdot p_1 \cdot p_0 \\
P_1 &= p_7 \cdot p_6 \cdot p_5 \cdot p_4 \\
P_2 &= p_{11} \cdot p_{10} \cdot p_9 \cdot p_8 \\
P_3 &= p_{15} \cdot p_{14} \cdot p_{13} \cdot p_{12}
\end{aligned}$$

  * For *super-generate* signal, the carry-out from the MSB of the 4-bit group is the only one of concern

$$\begin{aligned}
G_0 &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 \\
G_1 &= g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4 \\
G_2 &= g_{11} + p_{11} \cdot g_{10} + p_{11} \cdot p_{10} \cdot g_9 + p_{11} \cdot p_{10} \cdot p_9 \cdot g_8 \\
G_3 &= g_{15} + p_{15} \cdot g_{14} + p_{15} \cdot p_{14} \cdot g_{13} + p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12}
\end{aligned}$$

  * The higher level abstraction is described by

$$\begin{aligned}
C_1 &= G_0 + P_0 \cdot c_0 \\
C_2 &= G_1 + P_1 \cdot G_0 + P_1 \cdot P_0 \cdot c_0 \\
C_3 &= G_2 + P_2 \cdot G_1 + P_2 \cdot P_1 \cdot G_0 + P_2 \cdot P_1 \cdot P_0 \cdot c_0 \\
C_4 &= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0
\end{aligned}$$

  * Figure 4.24
  * Example: Both levels of propagate and generate; Determine the g_i, p_i, G_i and P_i values of the following 16-bit numbers

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

Also determine the carry-out C_4
First, determine $g_i = a_i \cdot b_i$ and $p_i = a_i + b_i$

a:	0001	1010	0011	0011
b:	1110	0101	1110	1011
g_i	0000	0000	0010	0011
p_i	1111	1111	1111	1011

Super-propagates are simply the AND of lower-level propagates

$$\begin{aligned} P_3 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\ P_2 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\ P_1 &= 1 \cdot 1 \cdot 1 \cdot 1 = 1 \\ P_0 &= 1 \cdot 0 \cdot 1 \cdot 1 = 0 \end{aligned}$$

Super-generates are given by

$$\begin{aligned} G_0 &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 \\ &= 0 + 1 \cdot 0 + 1 \cdot 0 \cdot 1 + 1 \cdot 0 \cdot 1 \cdot 1 \\ &= 0 + 0 + 0 + 0 \\ &= 0 \\ G_1 &= g_7 + p_7 \cdot g_6 + p_7 \cdot p_6 \cdot g_5 + p_7 \cdot p_6 \cdot p_5 \cdot g_4 \\ &= 0 + 1 \cdot 0 + 1 \cdot 1 \cdot 1 + 1 \cdot 1 \cdot 1 \cdot 0 \\ &= 0 + 0 + 1 + 0 \\ &= 1 \\ G_2 &= g_{11} + p_{11} \cdot g_{10} + p_{11} \cdot p_{10} \cdot g_9 + p_{11} \cdot p_{10} \cdot p_9 \cdot g_8 \\ &= 0 + 1 \cdot 0 + 1 \cdot 1 \cdot 0 + 1 \cdot 1 \cdot 1 \cdot 0 \\ &= 0 + 0 + 0 + 0 \\ &= 0 \\ G_3 &= g_{15} + p_{15} \cdot g_{14} + p_{15} \cdot p_{14} \cdot g_{13} + p_{15} \cdot p_{14} \cdot p_{13} \cdot g_{12} \\ &= 0 + 1 \cdot 0 + 1 \cdot 1 \cdot 0 + 1 \cdot 1 \cdot 1 \cdot 0 \\ &= 0 + 0 + 0 + 0 \\ &= 0 \\ C_4 &= G_3 + P_3 \cdot G_2 + P_3 \cdot P_2 \cdot G_1 + P_3 \cdot P_2 \cdot P_1 \cdot G_0 + P_3 \cdot P_2 \cdot P_1 \cdot P_0 \cdot c_0 \\ &= 0 + 1 \cdot 0 + 1 \cdot 1 \cdot 1 + 1 \cdot 1 \cdot 1 \cdot 0 + 1 \cdot 1 \cdot 1 \cdot 0 \cdot 0 \\ &= 0 + 0 + 1 + 0 + 0 \\ &= 1 \end{aligned}$$

  - Speed of ripple adder vs carry-lookahead adder
    - Computed by comparing the number of gate delays along the longest path through a piece of logic
    - Left as reading assignment

Multiplication

- Review the multiplication of two numbers in long hand

Multiplicand					1	0	0	0
Multiplier	×				1	0	0	1
					1	0	0	0
				0	0	0	0	
			0	0	0	0		
		1	0	0	0			
Product		1	0	0	1	0	0	0

- Observation: Number of digits in product is considerably larger than either multiplicand or multiplier
  - If multiplicand has n bits and multiplier has m bits, the product requires $n + m$ bits
- Assume that we are multiplying only positive numbers

- First version of multiplication algorithm and hardware
  - Mimics the algorithm used above
  - Figure 4.25 and 4.26
  - Multiplier in 32-bit *multiplier register*
  - Product in a 64-bit register initialized to 0
  - Multiplicand and ALU also 64-bit wide
  - Multiplicand is shifted left by 1 bit at every step
  - Control decides when to shift multiplicand and multiplier registers and when to write new values into product
  - Example multiplying 2 by 3 (4-bits)
  - Problem: Too many clock cycles for shift and add operations
- Second version of multiplication algorithm and hardware
  - Half the bits in the multiplicand are always zero
    - Multiplicand is 32-bit but lives in a 64-bit register
  - 64-bit ALU is wasteful as half the adder bits add 0 to intermediate sum
  - Instead of shifting multiplicand left, shift product right
  - The sum at every step is 32-bit, so only the most significant 32-bits of the product get modified
  - Figures 4.29 and 4.28
  - Example multiplying 2 by 3 (4-bits)
- Final version of multiplication algorithm and hardware
  - Product register has wasted space that equals the number of bits used up in multiplier at every step
  - Combine rightmost half of product with multiplier
  - Initialize by putting multiplier in the least significant 32-bits of product, with most significant 32-bits initialized to 0
  - Figures 4.32 and 4.31
  - Example multiplying 2 by 3 (4-bits)
- Signed Multiplication
  - Possible to multiply the two numbers as positive and adjust the sign at the end
  - The last algorithm works as long as we take care of sign extension
- Booth's algorithm
  - More elegant approach to multiply signed numbers
  - With the ability to add and subtract, there are multiple ways to compute a product
  - Multiply 2_{10} by 6_{10}

					0	0	1	0		
×					0	1	1	0		
+					0	0	0	0	shift	(0 in multiplier)
+				0	0	1	0		add	(1 in multiplier)
+			0	0	1	0			add	(1 in multiplier)
+		0	0	0	0				shift	(0 in multiplier)
	0	0	0	0	1	1	0	0		

- Booth's observation: A CPU that can add or subtract can get the same result in more than one way

$$\begin{aligned} 6_{10} &= -2_{10} + 8_{10} \\ 0110_2 &= -0010_2 + 1000_2 \end{aligned}$$

- Replace a string of 1's in the multiplier (0110_2) with an initial subtract when you see a 1 (0010_2) and later add when you see the bit *after* the last 1 (1000_2)
- Back to multiply 0010_2 by 0110_2

					0	0	1	0		
×					0	1	1	0		
+					0	0	0	0	shift	(0 in multiplier)
-				0	0	1	0		subtract	(first 1 in multiplier)
+			0	0	0	0			shift	(middle of string of 1s)
+		0	0	1	0				add	(prior step had last 1)
	0	0	0	0	1	1	0	0		

- Remember that when you subtract, you are doing 2's complement arithmetic, and take care of sign extension in the intermediate step
- Classify bits into beginning, middle, and end of a run of 1s
- Classification is simplified by examining only two bits at a time – current and previous
  - Since there is no previous bit in the beginning, we add a new bit – variously called as *a mythical bit* or bit_{-1} – to the right of multiplier/product register, and initialize it to 0
- Four possibilities based on current and previous bits
  1. 00 – Middle of string of 0s, just shift
  2. 01 – End of string of 1s, add multiplicand to the left half of product
  3. 10 – Beginning of string of 1s, subtract multiplicand from left half of product
  4. 11 – Middle of string of 1s, just shift
- *Caution: Initialize the left half of product register to all 0's, even if the multiplier is negative; Do not sign extend the multiplier in the product register*
- Example: Multiply 2_{10} by 6_{10} (Figure 4.34)
- Multiply by powers of 2 (2^i) can be simply achieved by shifting the bits (`sll`) in MIPS

- Multiply in MIPS
  - Separate pair of 32-bit registers (`HI` and `LO`) to hold the 64-bit product
  - Two instructions – `mult` and `multu`
  - `mflo` and `mfhi` – Multiply from low/high
    - `mfhi` is used to transfer the contents of `HI` to a general-purpose register to check for overflow

Division

- Long division using binary numbers
  - Divide 84_{10} by 8_{10}, or 1010100_2 by 1000_2

								1	0	1	0		Quotient
1	0	0	0		1	0	1	0	1	0	0		Dividend
				−	1	0	0	0					
							1	0					
							1	0	1				
							1	0	1	0			
						−	1	0	0	0			
									1	0			
									1	0	0		Remainder

  - Assume that both dividend and divisor are positive
  - Division operands (dividend and divisor) and both results (quotient and remainder) are 32-bit
- First version of division algorithm and hardware
  - Mimics the above algorithm
  - Start with 32-bit quotient register set to 0
  - Put divisor in the left half of 64-bit divisor register and shift right at each iteration to align with dividend
  - Initialize the 64-bit remainder register with dividend
  - 64-bit ALU
  - Algorithm requires $n + 1$ steps to divide n bit operands
  - Example: $7 \div 2$
  - Figures 4.37 and 4.36
- Second version of the division algorithm and hardware
  - Cut divisor and ALU in half
  - Shift remainder to left instead of shifting divisor to right
  - Figure 4.39
  - Algorithm cannot produce a 1 in the quotient bit in the first step
    * Resolved by switching the order of shift and subtract
    * Also removes one iteration of the algorithm
- Final version of the division algorithm and hardware
  - Quotient register can be eliminated by shifting the bits of quotient into the remainder instead of shifting in 0s
  - Figure 4.40 and 4.41
  - Shift the remainder left as before in the first step
  - Later steps require only one shift as remainder is in left half and quotient is in right half
  - Final step: shift remainder right by 1 bit in the left half of the register
- Signed division
  - Simplest solution: Remember the signs and negate the quotient if signs disagree
  - Must also set the sign of the remainder with the constraint

$$\texttt{Dividend} = \texttt{Quotient} \times \texttt{Divisor} + \texttt{Remainder}$$

  - Dividing combinations of ± 7 by ± 2 (verify by plugging into above formula)

$$\begin{aligned} +7 \div +2 &: \text{Quotient: } +3, \text{ Remainder: } +1 \\ -7 \div +2 &: \text{Quotient: } -3, \text{ Remainder: } -1 \\ +7 \div -2 &: \text{Quotient: } -3, \text{ Remainder: } +1 \\ -7 \div -2 &: \text{Quotient: } +3, \text{ Remainder: } -1 \end{aligned}$$

  - Negate the quotient if the operands are of opposite signs
  - Match the sign of remainder to that of dividend
- Divide in MIPS
  - Use the 32-bit `HI` and `LO` registers for both multiply and divide

  - `HI` contains remainder
  - `LO` contains quotient
  - Instructions `div` and `divu`

Floating point

- Fractions, or *reals*
- Exponential or scientific notation
  - Single digit to the left of decimal point
  - Normalized scientific notation with decimal numbers
    - Only one digit, with no leading zeroes, to the left of decimal point
    - 1.86×10^5 – Normalized
    - 186×10^3 – Not normalized
    - 0.186×10^6 – Not normalized
  - Normalized numbers in binary
    - 1.010×2^{101}
    - Binary point (instead of decimal point)
  - Advantages of normalized form
    1. Simpler exchange of data that includes floating point numbers
    2. Standardized algorithms for floating point arithmetic
    3. Improved accuracy by removing leading zeros in the fraction
- Notation
  - General format: $1.xxxxxxxxx_2 \times 2^{yyyy}$
  - $1.xxxxxxxxx$ part is called *mantissa* or *significand*
  - $yyyy$ part is called *exponent*
  - We'll show the $yyyy$ part in decimal just to simplify the system
- Floating point representation
  - Compromise between size of mantissa and exponent
  - Tradeoff gives you accuracy or range but not both
  - MIPS floating point number – sign and magnitude representation
    - Bit 31 – Sign bit
    - Bits 23–30 – 8-bit exponent
    - Bits 0-22 – Mantissa or significand
    - Known as *sign-and-magnitude* representation because sign is separate from magnitude (mantissa)
  - General representation: $(-1)^S \times F \times 2^E$
  - There is extraordinary but limited range of numbers that can be represented
  - Overflow means that exponent is too large to be represented in the exponent field
  - If negative exponent's magnitude is too large to be represented, it is known as *underflow*
  - `double` vs `float`
    - 11-bit exponent in `double`
    - 52-bit mantissa

- The two representations – `float` and `double` – as described are part of the *IEEE 754 floating point standard*
- The leading bit of normalized binary numbers can be made implicit, yielding 24-bit mantissa in single precision and 53-bit mantissa in double precision
- 0 has no leading one, and hence, gets the reserved exponent value 0
  - If the sign bit is 0, it is implicitly given a value 1 for all practical purposes
- Representation is given by: $(-1)^S \times (1 + \text{Significand}) \times 2^E$
  - Bits of significand represent the fraction between 0 and 1
  - Numbering the bits from left to right as $s_1, s_2, \ldots$, the value is given by

$$(-1)^S \times (1 + (s_1 \times 2^{-1}) + (s_2 \times 2^{-2}) + (s_3 \times 2^{-3}) + (s_4 \times 2^{-4}) + \cdots) \times 2^E$$

- Putting sign in MSB allows for comparison tests to be performed quickly, especially for sorting applications
- Representing $1.0 \times 2^{\pm 1}$

$$\begin{aligned} 1.0 \times 2^{-1} &= 0\ 11111111\ 00000000000000000000000 \\ 1.0 \times 2^{-1} &= 0\ 00000001\ 00000000000000000000000 \end{aligned}$$

- *Biased notation*
  - Represent most negative exponent as 00000000_2 and the most positive as 11111111_2
  - *Bias* is given by the number to be *subtracted* from the normal unsigned number to get the true number
  - IEEE uses a bias of 127 for single precision, and 1023 for double precision
  - The new representation for $1.0 \times 2^{\pm 1}$ is

$$\begin{aligned} 1.0 \times 2^{-1} &= 0\ 01111110\ 00000000000000000000000 \\ 1.0 \times 2^{-1} &= 0\ 10000000\ 00000000000000000000000 \end{aligned}$$

  - Value represented by floating point number is $(-1)^S \times (1 + \text{Significand}) \times 2^{E-\text{Bias}}$
- Floating point representation
  - -0.75 in single and double precision

$$\begin{aligned} -0.75_{10} &= (3/4)_{10} \\ &= (3/2^2)_{10} \\ &= (11/100)_2 \\ &= (0.11 \times 2^0)_2 \\ &= (1.1 \times 2^{-1})_2 \end{aligned}$$

  - A better algorithm is given by

```
b = 0;                  // Number of bits; precision desired
d = 0.75;               // Floating point mantissa to be converted to binary

while ( d > 0 && i < nbits )    // nbits is the number of bits desired
{
    if ( ( d *= 2 ) >= 1.0 )
    {
        write ( '1' );
        d -= 1.0;
    }
    else
        write ( '0' );
    i++;
}
```

  - Use general representation from above with bias, the exponent should be $-1 + 127 = 126$

    - Therefore, single precision representation is
      $$1\ 01111110\ 10000000000000000000000$$
    - The double precision representation is
      $$1\ 01111111110\ 1000000000000000000000000000000000000000000000000000$$
  - Converting binary to decimal floating point
    - Binary number: 1 10000001 01000000000000000000000
    - Sign bit is set, so number is negative
    - Exponent: 129 - 127 = 2
    - Mantissa: $(1 + (0 \times 2^{-1}) + (1 \times 2^{-2})$, or $(1 + 0.25)$, or 1.25
    - The number is: $-1^1 \times 1.25 \times 2^2$, or -1.25×4, or -5.0

- Floating point addition
  - Before adding numbers, there exponents must be the same (aligned)
  - The significand must be normalized after addition
  - The significand may need to be rounded
  - See by adding $9.999_{10} \times 10^1$ and $1.610_{10} \times 10^{-1}$
    - Align the exponent, making $1.610_{10} \times 10^{-1}$ into $0.016_{10} \times 10^1$
    - Adding two together, we have $10.015_{10} \times 10^1$
    - Normalizing, we get $1.0015_{10} \times 10^2$
    - Rounding, we get $1.002_{10} \times 10^2$
  - Algorithm in Figure 4.44
  - Add 0.5_{10} and -0.4375_{10} in binary using above algorithm
  - Look at the hardware figure in the book

- Floating point multiplication
  - Exponent of two operands is simply added together (remember to take care of bias)
  - Multiply the significands
  - Normalize result
  - Round the result
  - Fix the sign
  - Algorithm for multiplication: Figure 4.46

- Floating point instructions in MIPS
  - Addition, single and double: `add.s` and `add.d`
  - Subtraction, single and double: `sub.s` and `sub.d`
  - Multiplication, single and double: `mul.s` and `mul.d`
  - Division, single and double: `div.s` and `div.d`
  - Comparisons, single and double: `c.eq.s`, `c.eq.d`, `c.neq.s`, `c.neq.d`, `c.lt.s`, `c.lt.d`, `c.le.s`, `c.le.d`, `c.gt.s`, `c.gt.d`, `c.ge.s`, and `c.ge.d`
  - Branch, true or false: `bc1t` or `bc1f`
  - Separate floating point registers: `FP0`, `FP2`, `FP4`, ..., `FP30` (all `double`)
  - Registers `FP0`, `FP8`, `FP16`, and `FP24` are also used for single precision
  - Load and store is achieved by: `lwc1` and `swc1`