L. BIT MANIPULATION pdf

1: Bit Manipulation - LeetCode

A big advantage of bit manipulation is that it can help to iterate over all the subsets of an N-element set. As we all know there are 2 N possible subsets of any given set with N elements. What if we represent each element in a subset with a bit.

Chapter Five Bit Manipulation 5. Indeed, one of the reasons people claim that the "C" programming language is a "medium-level" language rather than a high level language is because of the vast array of bit manipulation operators that it provides. This chapter will discuss how to manipulate strings of bits in memory and registers using 80x86 assembly language. This chapter begins with a review of the bit manipulation instructions covered thus far and it also introduces a few new instructions. This chapter reviews information on packing and unpacking bit strings in memory since this is the basis for many bit manipulation operations. Finally, this chapter discusses several bit-centric algorithms and their implementation in assembly language. Before describing how to manipulate bits, it might not be a bad idea to define exactly what this text means by "bit data. For our purposes, bit manipulation refers to working with data types that consist of strings of bits that are non-contiguous or are not an even multiple of eight bits long. Generally, such bit objects will not represent numeric integers, although we will not place this restriction on our bit strings. Note that a bit string does not have to start or end at any special point. For example, a bit string could start in bit seven of one byte in memory and continue through to bit six of the next byte in memory. In memory, the bits must be physically contiguous i. In registers, if a bit string crosses a register boundary, the application defines the continuation register but the bit string always continues in bit zero of that second register. A bit set is a collection of bits, not necessarily contiguous though it may be , within some larger data structure. For example, bits Normally, we will deal with bit sets that are part of an object no more than about 32 or 64 bits in size. Note that bit strings are special cases of bit sets. A bit run is a sequence of bits with all the same value. A run of zeros is a bit string containing all zeros, a run of ones is a bit string containing all ones. The first set bit in a bit string is the bit position of the first bit containing a one in a bit string, i. A similar definition exists for the first clear bit. A similar definition exists for the last clear bit. A bit offset is the number of bits from some boundary position usually a byte boundary to the specified bit. As noted in Volume One, we number the bits starting from zero at the boundary location. If the offset is less than 32, then the bit offset is the same as the bit number in a byte, word, or double word value. Likewise, if you use this same value with the OR instruction, it can force bits four through seven to ones in the destination operand. Indeed, on the earliest 80x86 processors, these were the only instructions available for bit manpulation. The following paragraphs review these instructions, concentrating on how you could use them to manipulate bits in memory or registers. The AND instruction provides the ability to strip away unwanted bits from some bit sequence, replacing the unwanted bits with zeros. This instruction is especially useful for isolating a bit string or a bit set that is merged with other, unrelated data or, at least, data that is not part of the bit string or bit set. For example, suppose that a bit string consumes bit positions 12 through 24 of the EAX register, we can isolate this bit string by setting all other bits in EAX to zero by using the following instruction: In theory, you could use the OR instruction to mask all unwanted bits to ones rather than zeros, but later comparisons and operations are often easier if the unneeded bit positions contain zero. You can also use the OR instruction to mask unwanted bits around a set of bits. However, the OR instruction does not let you clear bits, it allows you to set bits to ones. In some instances setting all the bits around your bit set may be desirable; most software, however, is easier to write if you clear the surrounding bits rather than set them. The OR instruction is especially useful for inserting a bit set into some other bit string. To do this, there are several steps you must go through: Clear all the bits surrounding your bit set in the source operand. Clear all the bits in the destination operand where you wish to insert the bit set. OR the bit set and destination operand together. For example, suppose you have a value in bits You would begin by stripping out bits 13 and above from EAX; then you would strip out bits Next, you would shift the bits in EAX so the bit string occupies bits When working with bit masks, it is incredibly poor programming style to use literal numeric constants as in the past few examples. Combined with some constant expressions, you can

produce code that is much easier to read and maintain. The current example code is more properly written as: Notice the use of the compile-time not operator "! This saves having to create another constant in the program that has to be changed anytime you modify the BitMask constant. Having to maintain two separate symbols whose values are dependent on one another is not a good thing in a program. Of course, in addition to merging one bit set with another, the OR instruction is also useful for forcing bits to one in a bit string. By setting various bits in a source operand to one you can force the corresponding bits in the destination operand to one by using the OR instruction. The XOR instruction, as you may recall, gives you the ability to invert selected bits belonging to a bit set. Of course, if you want to invert all the bits in some destination operand, the NOT instruction is probably more appropriate than the XOR instruction; however, to invert selected bits while not affecting others, the XOR is the way to go. Although this might seem like a waste, since you can easily force this four-bit string to zero or all ones using AND or OR, the XOR instruction has two advantages: Remember, however, that this trick only works if you know the current value of a bit set within the destination operand. These instructions affect the flags as follows: These instructions always clear the carry and overflow flags. These instructions set the sign flag if the result has a one in the H. These instructions set the parity flag if there are an even number of set bits in the L. The first thing to note is that these instructions always clear the carry and overflow flags. This means that you cannot expect the system to preserve the state of these two flags across the execution of these instructions. A very common mistake in many assembly language programs is the assumption that these instructions do not affect the carry flag. This simply will not work. One of the more interesting aspects to these instructions is that they copy the H. This means that you can easily test the setting of the H. For this reason, many assembly language programmers will often place an important boolean variable in the H. Indeed, earlier volumes have done little more than acknowledge its existence. However, since this is a chapter on bit manipulation and parity computation is a bit manipulation operation, it seems only fitting to provide a brief discussion of the parity flag at this time. Parity is a very simple error detection scheme originally employed by telegraphs and other serial communication schemes. The idea was to count the number of set bits in a character and include an extra bit in the transmission to indicate whether that character contained an even or odd number of set bits. The receiving end of the transmission would also count the bits and verify that the extra "parity" bit indicated a successful transmission. An important fact bears repeating here: The instruction set only uses the L. Although the need to know whether the L. Indeed, programs reference this flag so often after the AND instruction that Intel added a separate instruction, TEST, whose main purpose was to logically AND two results and set the flags without otherwise affecting either instruction operand. Use 1 is actually a special case of 2 where the bit set contains only a single bit. A common use for the AND instruction, and also the original reason for the inclusion of the TEST instruction in the 80x86 instruction set, is to test to see if a particular bit is set in a given operand. These clears all the other bits in the second operand leaving a zero in the bit position under test the bit position with the single set bit in the constant operand if the operand contains a zero in that position and leaving a one if the operand contains a one in that position. Since all of the other bits in the result are zero, the entire result will be zero if that particular bit is zero, the entire result will be non-zero if that bit position contains a one. The following instruction sequence demonstrates how to test to see if bit four is set in EAX: Simply supply a constant that has one bits in all the positions you want to test and zeros everywhere else. ANDing such a value with an unknown quantity will produce a non-zero value if one or more of the bits in the operand under test contain a one. The following example tests to see if the value in EAX contains a one in bit positions one, two, four, and seven: To accomplish this, you must first mask out the bits that are not in the set and then compare the result against the mask itself. If the result is equal to the mask, then all the bits in the bit set contain ones. The following example checks to see if all the bits corresponding to a value this code calls bitMask are equal to one: We can check for any combination of values by specifying the appropriate value as the operand to the CMP instruction. This suggests another way to check for all ones in the bit set: Then if the zero flag is set, you know that there were all ones in the original bit set, e. This was for purposes of example only. In fact, you can use a variable or other register here, if you prefer. The BTx instructions allow the following syntactical forms: If the second operand is a memory location, then the bit count is not limited to values in the range If the first

operand is a constant, it can be any eight-bit value in the range If the first operand is a register, it has no limitation. The BT instruction copies the specified bit from the second operand into the carry flag. For example, the "bt 8, eax ;" instruction copies bit number eight of the EAX register into the carry flag. You can test the carry flag after this instruction to determine whether bit eight was set or clear in EAX. In general, the BT instruction is, perhaps, not the best instruction for testing individual bits in a register.

L. BIT MANIPULATION pdf

2: Bit Manipulation

Bit manipulation in the C programming language The programming language C has direct support for bitwise operations that can be used for bit manipulation. In the following examples, n is the index of the bit to be manipulated within the variable bit_fld, which is an unsigned char being used as a bit field.

B digital pin 8 to 13 C analog input pins D digital pins 0 to 7 Each port is controlled by three registers, which are also defined variables in the arduino language. The maps of the ATmega8 and ATmega chips show the ports. The newer Atmegap chip follows the pinout of the Atmega exactly. PIN registers correspond to the state of inputs and may only be read. For a complete mapping of Arduino pin numbers to ports and bits, see the diagram for your chip: Examples Referring to the pin map above, the PortD registers control Arduino digital pins 0 to 7. Be aware that this can interfere with program download or debugging. The bits in this register control whether the pins in PORTD are configured as inputs or outputs so, for example: PIND is the input register variable It will read all of the digital input pins at the same time. Why use port manipulation? From The Bitmath Tutorial Generally speaking, doing this sort of thing is not a good idea. Here are a few reasons: The code is much more difficult for you to debug and maintain, and is a lot harder for other people to understand. Your time is valuable, right? Usually it is much better to write code the most obvious way. The code is less portable. If you use digitalRead and digitalWrite , it is much easier to write code that will run on all of the Atmel microcontrollers, whereas the control and port registers can be different on each kind of microcontroller. It is a lot easier to cause unintentional malfunctions with direct port access. Pin 0 is the receive line RX on the serial port. It would be very easy to accidentally cause your serial port to stop working by changing pin 0 into an output pin! So you might be saying to yourself, great, why would I ever want to use this stuff then? Here are some of the positive aspects of direct port access: You may need to be able to turn pins on and off very quickly, meaning within fractions of a microsecond. Each machine instruction requires one clock cycle at 16MHz, which can add up in time-sensitive applications. Direct port access can do the same job in a lot fewer clock cycles. Sometimes you might need to set multiple output pins at exactly the same time. It requires a lot fewer bytes of compiled code to simultaneously write a bunch of hardware pins simultaneously via the port registers than it would using a for loop to set each pin separately. In some cases, this might make the difference between your program fitting in flash memory or not! See Reference Home Corrections, suggestions, and new documentation should be posted to the Forum. Code samples in the reference are released into the public domain.

L. BIT MANIPULATION pdf

3: Tutorials - Bitwise Operators and Bit Manipulations in C and C++ - www.enganchecubano.com

Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a byte. C language is very efficient in manipulating bits. Here are following operators to perform bits manipulation: & Binary AND Operator copies a bit to the result if it exists in both operands.

Exclusive-or encryption is one example when you need bitwise operations. Another example comes up when dealing with data compression: In principle, this means taking one representation and turning it into a representation that takes less space. One way of doing this is to use an encoding that takes less than 8 bits to store a byte. In order to encode and decode files compressed in this manner, you need to actually extract data at the bit level. Finally, you can use bit operations to speed up your program or perform neat tricks. Understanding what it means to apply a bitwise operator to an entire string of bits is probably easiest to see with the shifting operators. Regardless of underlying representation, you may treat this as true. As a consequence, the results of the left and right shift operators are not implementation dependent for unsigned numbers for signed numbers, the right shift operator is implementation defined. The leftshift operator is the equivalent of moving all the bits of a number a specified number of places to the left: This is the number 32 -- in fact, left shifting is the equivalent of multiplying by a power of two. Note that a bitwise right-shift will be the equivalent of integer division by 2. Why is it integer division? Consider the number 5, in binary, When you perform a right shift by one: Note that this only holds true for unsigned integers; otherwise, we are not guaranteed that the padding bits will be all 0s. Generally, using the left and right shift operators will result in significantly faster code than calculating and then multiplying by a power of two. The shift operators will also be useful later when we look at how to manipulating individual bits. For instance, working with a byte the char type: The only time where both bits are 1, which is the only time the result will be 1, is the fifth bit from the left. If both bits are a 1, the result will also have a 1 in that position. The symbol is a pipe: Again, this is similar to boolean logical operator, which is. Well, we need to isolate the one bit that corresponds to that car. The strategy is simple: Consider trying to extract the fifth bit from the right of a number: What about the question mark? XXXXX and , then the result will be This procedure works for finding the bit in the nth position. The only thing left to do is to create a number with only the one bit in the correct position turned on. These are just powers of two, so one approach might be to do something like: It obscures the fact that what we want to do is shift a bit over a certain number of places, so that we have a number like -- a couple of zeros, a one, and some more zeros. The one could also be first or last -- or There are two cases to consider: In one case, we need to turn a bit on, and in the other, turn a bit off. What does this suggest we should do? If we have a bit set to zero, the only way we know right now to set it to 1 is to do a bitwise OR. Again we need to move a single bit into the correct position: Take the case of setting the rightmost bit to 1: The shift is the same as before; the only difference is the operator and that we store the result. Setting a car to be no longer in use is a bit more complicated. A useful way to remember this is that the tilde is sometimes called a twiddle, and the bitwise complement twiddles every bit: This turns out to be a great way of finding the largest possible value for an unsigned number: Once we twiddle 0, we get all 1s: We know that all 1s is the largest possible number. This non-equivalence principle holds true for bitwise AND too, unless you know that you are using strictly the numbers 1 and 0. Now that we have a way of flipping bits, we can start thinking about how to turn off a single bit. We know that we want to leave other bits unaffected, but that if we have a 1 in the given position, we want it to be a 0. Take some time to think about how to do this before reading further. We need to come up with a sequence of operations that leaves 1s and 0s in the non-target position unaffected; before, we used a bitwise OR, but we can also use a bitwise AND. Now, to turn off a bit, we just need to AND it with 0: How can we get that number? This is where the ability to take the complement of a number comes in handy: If we turn one bit on and take the complement of the number, we get every bit on except that bit: We actually need to know whether a car is in use or not if the bit is on or off before we can know which function to call. There is an easier way, but first we need the last bitwise operator: Bitwise Exclusive-Or XOR There is no boolean operator counterpart to bitwise exclusive-or, but there is a simple explanation. The exclusive-or operation

takes two inputs and returns a 1 if either one or the other of the inputs is a 1, but not if both are. That is, if both inputs are 1 or both inputs are 0, it returns 0. Exclusive-or is commonly abbreviated XOR. For instance, if you have two numbers represented in binary as and then taking the bitwise XOR results in So you can think of the XOR operation as a sort of selective twiddle: As an exercise, can you think of a way to use this to exchange two integer variables without a temporary variable? How does that help us? Well, remember the first principle: XORing a bit with 0 results in the same bit. This leaves everything unchanged, but flips the bit instead of always turning it on: Bitwise operators are good for saving space -- but many times, space is hardly an issue. And one problem with working at the level of the individual bits is that if you decide you need more space or want to save some time -- for instance, if we needed to store information about 9 cars instead of 8 -- then you might have to redesign large portions of your program. There are also times when you need to use bitwise operators: Summary You should now be familiar with six bitwise operators: And you now should have a better sense of what goes on at the lowest levels of your computer. A Parting Puzzle One final neat trick of bitwise operators is that you can use them, in conjunction with a bit of math, to find out whether an integer is a power of two. Take some time to think about it, then check out the solution.

L. BIT MANIPULATION pdf

4: Bit manipulation in C# using a mask - Stack Overflow

I did provide o(n) solution with hash table but he seemed to be asking for a bit manipulation or bit masking technique for 32 bit address. - akira November

If you wish, you can download the program source code, project files, and binaries here. Characters can represent values from 0 to , if we use unsigned characters. To represent these values, they use a series of bits. We know it takes 8 bits to make a byte because we know that computers work in a binary number system, and can only count in powers of 2, and if we take 8 numeric positions, we get 2 to the 8th power base 2 raised to the 8 number positions and get , but since we start counting at 0, that gives us This is all fine and good you may say, but how does that help me in C programming? Well, it all comes down to efficiency concerns, as usual. Imagine we needed to store a yes or no value. We could represent that with the values 1 for yes, and 0 for no. Imagine we had 8 such values we needed to store. Now we are wasting 56 bits. Is this an efficient design? So, what is the solution? Bit manipulation, of course. C has built-in methods for manipulating data a single bit at a time. For example, in our last lesson, when we talked about unions, we talked about the Symbol Entry structure, which stored a flag register to keep track of various items related to a symbol file. They had 16 different flags, all with yes or no values, and they used a single short integer 2 bytes to store all 16 flags. This means they only had to use 2 bytes to store 16 different properties about their files. So, what does this semicolon one mean? This is a bit-field. However, it does not have to be a single bit. We can use any number from 1 to 16 remember that a short integer is 2 bytes, or 16 bits. However, there is little point in using values of 8 or 16, since we can use characters for 8 bits, and short integers for 16 bits. You might be thinking that there are 2 unsigned shorts here, so why is it 16 bits and not The reason is because the compiler will always pack bit-fields into the smallest possible space. You could actually have put each of those declarations on a single line and it would still take only 16 bits. Note that you can only declare bit-fields inside structures and unions. So, if we wanted to represent small values, like the values for cards, which have face values no higher than 13 13 faces in a poker deck per suit , and 2 suits, we could represent it all with a single character. This means we can store any values from 0 to 2 raised to the 4th power - 1 2 to the 4th being 16, that puts our range at To make it less confusing, we could define an enumeration to handle our constants with more meaningful names, like this: Now, if you remember our last definition, which looked more like this: Contrast this with our new definition, and we only use 52 bytes for an entire deck. This would mean our new definition would only use bytes, but our old definition would use bytes. On a platform with such limited memory, this is a significant savings. It is important to note however that though you save memory here in the storage space used, it is more difficult for the processor to work with single bits than with bytes. This means more code is generated to work with bit-fields than with standard variables. You will want to try both to see if you actually end up saving memory at all. A better use of bit-fields than memory saving concerns are flagsets like the one in the AMS symbol pointer struct. Remember in our card example, we used 4 bits for the face, and 2 bits for the suit. This adds up to only 6 bits, but an unsigned character uses 8 bits. Does this mean we have to use 8 bits, or do we only need 6? Well, the answer is we have to use 8. This is because there is no way to reserve less than 8 bits of memory at a time. This is a limitation of the processor. Create a new C Source File named bitcard. Modify the file so that it looks like this: Send it to TiEmu. It will look something like this: Step 2b - Program Analysis This program is nearly identical to the card program from lesson 7, but there are some important changes. The const keyword just means constant. Just remember the important difference was our use of the bit structure to declare the card. Although the program takes up roughly the same amount of memory as the card program from lesson 7, since most of the size of the program is embedded in the string literal constants of the card name arrays and the code itself, the program will require only a fourth as much memory to allocate space for the card array. Assuming this is the only thing we need to use from the dynamic memory, we could run this card game even if the calculator had only a couple hundred bytes of free memory available. This is of huge importance on a TI, where memory is so very limited. Step 3 - Bit Masks and the Bitwise Operations Another very important concept in C is the idea of bit masks in bitwise operations. They can use be

represented by a map of binary numbers, representing that a pixel should either be drawn, or not be drawn. We can also use the numbers to mean if a sprite should be drawn, like a block on a wall.

L. BIT MANIPULATION pdf

5: Bit Manipulation - Javaâ„¢ How to Program, Seventh Edition [Book]

Level up your coding skills and quickly land a job. This is the best place to expand your knowledge and get prepared for your next interview.

Left circular shift or rotate Right circular shift or rotate Another form of shift is the circular shift or bit rotation. In this operation, the bits are "rotated" as if the left and right ends of the register were joined. The value that is shifted in on the right during a left-shift is whatever value was shifted out on the left, and vice versa. This operation is useful if it is necessary to retain all the existing bits, and is frequently used in digital cryptography. Rotate through carry[edit] Left rotate through carry Right rotate through carry Rotate through carry is similar to the rotate no carry operation, but the two ends of the register are separated by the carry flag. The bit that is shifted in on either end is the old value of the carry flag, and the bit that is shifted out on the other end becomes the new value of the carry flag. A single rotate through carry can simulate a logical or arithmetic shift of one position by setting up the carry flag beforehand. With rotate-through-carry, that bit is "saved" in the carry flag during the first shift, ready to shift in during the second shift without any extra preparation. The number of places to shift is given as the second argument to the shift operators. Shifts can result in implementation-defined behavior or undefined behavior , so care must be taken when using them. If the first operand is of type uint or ulong, the right-shift is a logical shift. Care must be taken to ensure the statement is well formed to avoid undefined behavior and timing attacks in software with security requirements. A second try might result in: However, the branch adds an additional code path and presents an opportunity for timing analysis and attack, which is often not acceptable in high integrity software. To avoid the undefined behavior and branches under GCC and Clang, the following should be used. The pattern is recognized by many compilers, and the compiler will emit a single rotate instruction: Clang provides some rotate intrinsics for Microsoft compatibility that suffers the problems above. Intel also provides x86 Intrinsics. More details of Java shift operators: The type of the shift expression is the promoted type of the left-hand operand. If the promoted type of the left-hand operand is int, only the five lowest-order bits of the right-hand operand are used as the shift distance. If the promoted type of the left-hand operand is long, then only the six lowest-order bits of the right-hand operand are used as the shift distance. In bit and shift operations, the type byte is implicitly converted to int. If the byte value is negative, the highest bit is one, then ones are used to fill up the extra bytes in the int. Shifts in JavaScript[edit] JavaScript uses bitwise operations to evaluate each of two or more units place to 1 or 0. The number of places to shift is given as the second argument. For example, the following assigns x the result of shifting y to the left by two bits:

L. BIT MANIPULATION pdf

6: Bit Manipulation Interview Questions | CareerCup

Lesson 8: Bit Manipulation Step 1 - An Introduction to Bits in C As you may or may not know, the smallest data type in C is the character, or char for short.

Assume that the first value is no larger than 31 so that it has at most five significant bits and at least three leading 0 bits. Similarly assume that the second value is no larger than 15 four significant bits and the third value is no larger than seven bits. Give code to pack all three of these numbers into a bit word in the AX register, copying the low order five bits from value1 to bits 11â€”15 of AX, the low order four bits from value2 to bits 7â€”10 of AX, and the low-order seven bits from value3 into bits 0â€—6 of AX. Give code to unpack the 16 bit number in the AX register into fivebit, four-bit, and seven-bit numbers, padding each value with zeros on the left to make eight bits, and storing the resulting bytes at value1, value2, and value3 respectively. Write similar code sequences that use shift and addition instructions to efficiently multiply by 5, 7, 9, and The procedure will have two parameters, passed on the stack: Use a rotate instruction to extract the bits one at a time, left-to-right, recalling that jc or jnc instructions look at the carry bit. This exercise is the same as Programming Exercise 3 in Section 7. An eight-bit number can be represented using three octal digits. Bits 7 and 6 determine the left octal digit, which is never larger than 4, bits 5, 4, and 3 the middle digit, and bits 2, 1, and 0 the right digit. For instance, is 11 or The value of a bit number is represented in split octal by applying the 2â€—3â€—3 system to the high-order and low-order bytes separately. Write a NEAR32 procedure split-Octal which converts an bit integer to a string of exactly six characters representing the value of the number in split octal. These macros and the procedures they call are very similar. This section uses atod as an example. The atod macro expands into the following sequence of instructions. The actual source identifier is used in the expanded macro, not the name source. The assembled version of this procedure is contained in the file IO. Source code for atodproc is shown in Fig. The procedure begins with standard entry code. The flags are saved so that flag values that are not explicitly set or reset as promised in the comments can be returned unchanged. The popf and pop instructions at AToDExit restore these values; however, the word on the stack that is popped by popf will have been altered by the body of the procedure, as discussed below. Possible error conditions are: This is implemented with a straightforward while loop. Following the while loop, ESI points at some nonblank character. The main idea of the procedure is to compute the value of the integer by implementing the following left-to-right scanning algorithm. The second job of the procedure, after skipping blanks, is to store this multiplier, 1 for a positive number or 1 for a negative number. The multiplier, stored in local variable space on the stack, is given the default value 1 and changed to 1 if the first nonblank character is a minus sign. If the first nonblank character is either plus or a minus sign, then the address in ESI is incremented to skip over the sign character. Now the main design is executed. The value is accumulated in the EAX register. If multiplication by 10 produces an overflow, then the result is too large to represent in EAX. The jc overflowD instruction transfers control to the code at overflowD that takes care of all error situations. To convert a character to a digit, the character is loaded into the BL register and the instruction and ebx,Fh clears all bits except the low-order four in the EBX register. If adding the digit to the accumulated value produces a carry, the sum is too large for EAX; the jc instruction transfers control to overflowD. The main loop terminates as soon as ESI points at any character code other than one for a digit. Thus an integer is terminated by a space, comma, letter, null, or any nondigit. In order to determine if a valid integer has been entered, the main loop keeps a count of decimal digits in the CX register. When the loop terminates, this count is checked. If it is zero, there was no digit and the jz instruction jumps to overflowD for error handling. There is no need to check for too many digits; this would already have been caught by overflow in the main loop. Otherwise, the instruction test eax,eax is used to see if the accumulated value is larger than ; the sign bit will be 1 for a value of this magnitude. If any of the error conditions occur, the instructions starting at overflowD are executed. The original flags are popped into the AX register. The bit corresponding to the overflow flag is set to 1 to indicate an error, and a value of will be returned in EAX; other flags are set or reset to correspond to the zero value. The zero flag is set since the result returned will be zero; the parity flag is set since has even

parity an even number of 1 bits. The instruction and ax,b ; reset sign and carry flags clears bit 7 sign flag since is not negative and bit 0 carry , which is always cleared. The bit pattern resulting from these or and and instructions is pushed back on the stack to be popped into the flags register by popf before exiting the procedure. The new flag values are then pushed back on the stack with another pushf to be recovered by the normal popf in the exit code. The test instruction leaves AC undefined; this is why the comments at the beginning of the procedure mention AC. The procedure atodproc checks for zero digits in the number it is converting, but not for too many digits. Show why this is unnecessary by tracing the code for ,,,, the smallest possible digit number. Another valid reason not to limit the number of digits is that any number of leading zeros would be valid. The procedure should skip leading blanks and then accumulate a value until a character that does not represent a hex digit is encountered. Valid characters are 0 through 9, A through F, and a through f. If there are no hex digits or the result is too large to fit in EAX, then return 0 and set OF; these are the only possible errors. Clear OF if no error occurs. The Hardware Level Logic Gates Digital computers contain many integrated circuits and many of the components on these circuits are logic gates. A logic gate performs one of the elementary logical operations described in Section 8. Each type of gate has a simple diagram that represents its function. These diagrams are pictured in Fig. Logic Gates These simple circuits operate by getting logic 0 or 1 inputs and putting the correct value on the output. For example, if the two inputs of the or circuit are 0 and 1, then the output will be 1. Logic values 0 and 1 are often represented by two distinct voltage levels. The logic values at inputs x and y of this circuit can be thought of as two bits to add. These are exactly the results given by a half adder circuit. Half adder circuit Exercises 8. To do this takes a series of full adder circuits. One full adder circuit has three inputs x, y, and carry in, and two outputs, sum and carry out. Make a chart similar to the one in Fig. The chart will have five columns x, y, carry in, sum, and carry out and eight rows below the header row. Draw a full adder circuit. Use two half adders and an or gate to combine their carry outputs. Use three full adders and a half adder to draw a circuit that can add two four bit numbers. This circuit will have eight inputs four pairs of bits and five outputs four sum bits and a carry bit. For simplicity, you can draw each adder or half adder as a block diagram, without showing all its gates. Summary This chapter has explored the various 80x86 instructions that allow bits in a byte, word, or doubleword destination to be manipulated. The logical instructions and, or, and xor perform Boolean operations using pairs of bits from a source and destination. Applications of these instructions include setting or clearing selected bits in a destination. The test instruction is the same as the and instruction except that it only affects flags; the destination operand is unchanged. Shift instructions move bits left or right within a destination operand. These instructions come in single-bit and multiple-bit versions. Single-bit shifts use 1 for the second operand; multiple-bit versions use CL or an immediate value for the second operand and shift the destination the number of positions specified. Vacated positions are filled by 0 bits in all single shift operations except for the arithmetic right shift of a negative number, for which 1 bits are used. Shift instructions can be used for efficient, convenient multiplication or division by 2, 4, 8 or some higher power of two. Double shift instructions get bits to shift in from a source register. Rotate instructions are similar to shift instructions. However, the bit that falls off one end of the destination fills the void on the other end. Shift or rotate instructions can be used in combination with logical instructions to extract groups of bits from a location or to pack multiple values into a single byte or word. The atod macro generates code that calls the procedure atodproc. This procedure scans a string in memory, skipping leading blanks, noting a sign if any , and accumulating a doubleword integer value as ASCII codes for digits are encountered. Logical instructions are used in several places in the procedure. Logic gates are the primitive building blocks for digital computer circuits. Each gate performs one of the elementary Boolean operations.

7: Bit manipulation - Wikipedia

Refer Find most significant set bit of a number for details.; We can quickly check if bits in a number are in alternate pattern (like). We compute n ^ (n >> 1). If n has an alternate pattern, then n ^ (n >> 1) operation will produce a number having set bits only. '^' is a bitwise XOR operation.

L. BIT MANIPULATION pdf

8: Chapter Five Bit Manipulation

Bit Manipulation. Welcome to the latest post on Software Engineering , the publication where I aim to post weekly(ish) deep-dive articles on algorithms, design patterns, data structures and more!

9: C - Bits Manipulations

Learn about bit manipulation. This video is a part of HackerRank's Cracking The Coding Interview Tutorial with Gayle Laakmann McDowell. www.enganchecubano.com

L. BIT MANIPULATION pdf

Forty Fun Bible Puzzles for Kids #03 the First the Biggest and the Best Making a playground map. Taming the paper tiger and other dejunking adventures The diatoms in four postglacial deposits in Greenland. Fish cognition and behavior Everyone a Teacher (The Ethics of Everyday Life) Cats on a chandelier Sempre forever jm darhower Creo parametric 4.0 advanced tutorial Machine design reviewer Planning and organizing for curriculum renewal Dancing on my grave Ssc cgl tier 1 solved papers kiran prakashan 2007 lexus rx 350 manual Your hunting buddies are everything Love to the highest bidder Broadway musicals, 1943-2004 Story of the census. 1790-1915. Mcgraw hill ryerson data management 12 solutions OS/2 LAN server certification handbook Gate exam 2018 syllabus The environment, pristine, and common sense A study in economic history. Unpreparedness : Bull Run and its aftermath Johansson, G. Projective transformations as determining visual space perception. Communicating mathematics Ch. 6. The rationale and limits of living donor organ transplantation 3 XVI-XVIII century. Hearsay exception/records of regularly conducted activity business records MCSE Windows 98 study guide (exam 70-98) Nuclearisation of divided nations Check Your Own I.Q. The english language history On the Sex of Fish and the Gender of Scientists Burmese Refugees in Thailand Significantly significant : what it means for you and me A big guy took my ball Sienese quattrocento painting. Drama of the Apocalypse Philosophy after Hegel