|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Lesson 9 Screen04  The Screen04 lesson builds on Screen03, by teaching how to manipulate text. It is assumed you have the code for the [Lesson 8: Screen03](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen03.html) operating system as a basis.   |  | | --- | | **Contents**   * [1 String Manipulation](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen04.html#stringmanipulation) * [2 Division](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen04.html#division) * [3 Number Strings](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen04.html#numberstrings) * [4 Format Strings](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen04.html#formatstrings) * [5 Convert OS](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/screen04.html#convertos) |   1 String Manipulation  Variadic functions look much less intuitive in assembly code. Nevertheless, they are useful and powerful concepts.  Being able to draw text is lovely, but unfortunately at the moment you can only draw strings which are already prepared. This is fine for displaying something like the command line, but ideally we would like to be able to display and text we so desire. As per usual, if we put the effort in and make an excellent function that does all the string manipulation we could ever want, we get much easier code later on in return. Once such complicated function in C programming is sprintf. This function generates a string based on a description given as another string and additional arguments. What is interesting about this function is that it is variadic. This means that it takes a variable number of parameters. The number of parameters depends on the exact format string, and so cannot be determined in advance.  The full function has many options, and I list a few here. I've highlighted the ones which we will implement in this tutorial, though you can try to implement more.  The function works by reading the format string, and then interpreting it using the table below. Once an argument is used, it is not considered again. The return value of the function is the number of characters written. If the method fails, a negative number is returned.   | Table 1.1 sprintf formatting rules | | | --- | --- | | **Sequence** | **Meaning** | | *Any character except %* | Copies the character to the output. | | %% | Writes a % character to the output. | | %c | Writes the next argument as a character. | | %d *or* %i | Writes the next argument as a base 10 signed integer. | | %e | Writes the next argument in scientific notation using e**N** to mean ×10**N**. | | %E | Writes the next argument in scientific notation using E**N** to mean ×10**N**. | | %f | Writes the next argument as a decimal IEEE 754 floating point number. | | %g | Same as the shorter of %e and %f. | | %G | Same as the shorter of %E and %f. | | %o | Writes the next argument as a base 8 unsigned integer. | | %s | Writes the next argument as if it were a pointer to a null terminated string. | | %u | Writes the next argument as a base 10 unsigned integer. | | %x | Writes the next argument as a base 16 unsigned integer, with lowercase a,b,c,d,e and f. | | %X | Writes the next argument as a base 16 unsigned integer, with uppercase A,B,C,D,E and F. | | %p | Writes the next argument as a pointer address. | | %n | Writes nothing. Copies instead the number of characters written so far to the location addressed by the next argument. |   Further to the above, many additional tweaks exist to the sequences, such as specifying minimum length, signs, etc. More information can be found at [sprintf - C++ Reference](http://www.cplusplus.com/reference/clibrary/cstdio/sprintf/).  Here are a few examples of calls to the method and their results to illustrate its use.   | Table 1.2 sprintf example calls | | | | --- | --- | --- | | **Format String** | **Arguments** | **Result** | | "%d" | 13 | "13" | | "+%d degrees" | 12 | "+12 degrees" | | "+%x degrees" | 24 | "+1c degrees" | | "'%c' = 0%o" | 65, 65 | "'A' = 0101" | | "%d \* %d%% = %d" | 200, 40, 80 | "200 \* 40% = 80" | | "+%d degrees" | -5 | "+-5 degrees" | | "+%u degrees" | -5 | "+4294967291 degrees" |   Hopefully you can already begin to see the usefulness of the function. It does take a fair amount of work to program, but our reward is a very general function we can use for all sorts of purposes.  2 Division  Division is the slowest and most complicated of the basic mathematical operators. It is not implemented directly in ARM assembly code because it takes so long to deduce the answer, and so isn't a 'simple' operation.  While this function does look very powerful, it also looks very complicated. The easiest way to deal with its many cases is probably to write functions to deal with some common tasks it has. What would be useful would be a function to generate the string for a signed and an unsigned number in any base. So, how can we go about doing that? Try to devise an algorithm quickly before reading on.  The easiest way is probably the exact way I mentioned in [Lesson 1: OK01](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/ok01.html), which is the division remainder method. The idea is the following:   1. Divide the current value by the base you're working in. 2. Store the remainder. 3. If the new value is not 0, go to 1. 4. Reverse the order of the remainders. This is the answer.   For example:   | Table 2.1 Example base 2 conversion | | | | --- | --- | --- | | **Value** | **New Value** | **Remainder** | | 137 | 68 | 1 | | 68 | 34 | 0 | | 34 | 17 | 0 | | 17 | 8 | 1 | | 8 | 4 | 0 | | 4 | 2 | 0 | | 2 | 1 | 0 | | 1 | 0 | 1 |   So the answer is 100010012  The unfortunate part about this procedure is that it unavoidably uses division. Therefore, we must first contemplate division in binary.  For a refresher on long division expand the box below.  Long division explained  Let's suppose we wish to divide 4135 by 17.  0243 r 4  17)4135  0 0 × 17 = 0000  4135 4135 - 0 = 4135  34 200 × 17 = 3400  735 4135 - 3400 = 735  68 40 × 17 = 680  55 735 - 680 = 55  51 3 × 17 = 51  4 55 - 51 = 4  Answer: 243 remainder 4  First of all we would look at the top digit of the dividend. We see that the smallest multiple of the divisor which is less or equal to it is 0. We output a 0 to the result.  Next we look at the second to top digit of the dividend and all higher digits. We see the smallest multiple of the divisor which is less than or equal is 34. We output a 2 and subtract 3400.  Next we look at the third digit of the dividend and all higher digits. The smallest multiple of the divisor that is less than or equal to this is 68. We output 4 and subtract 680.  Finally we look at all remaining digits. We see that the lowest multiple of the divisor that is less than the remaining digits is 51. We output a 3, subtract 51. The result of the subtraction is our remainder.  To implement division in assembly code, we will implement binary long division. We do this because the numbers are stored in binary, which gives us easy access to the all important bit shift operations, and because division in binary is simpler than in any higher base due to the much lower number of cases.  1011 r 1  1010)1101111  1010  11111  1010  1011  1010  1  This example shows how binary long division works. You simply shift the divisor as far right as possible without exceeding the dividend, output a 1 according to the poisition and subtract the number. Whatever remains is the remainder. In this case we show 11011112 ÷ 10102 = 10112 remainder 12. In decimal, 111 ÷ 10 = 11 remainder 1.  Try to implement long division yourself now. You should write a function, DivideU32 which divides r0 by r1, returning the result in r0, and the remainder in r1. Below, we will go through a very efficient implementation.  function DivideU32(r0 is dividend, r1 is divisor)  set shift to 31 set result to 0 while shift ≥ 0  if dividend ≥ (divisor << shift) then  set dividend to dividend - (divisor <&lt shift) set result to result + 1  end if set result to result << 1 set shift to shift - 1  loop return (result, dividend)  end function  This code does achieve what we need, but would not work as assembly code. Our problem comes from the fact that our registers only hold 32 bits, and so the result of divisor << shift may not fit in a register (we call this overflow). This is a real problem. Did your solution have overflow?  Fortunately, an instruction exists called **clz** or count leading zeros, which counts the number of zeros in the binary representation of a number starting at the top bit. Conveniently, this is exactly the number of times we can shift the register left before overflow occurs. Another optimisation you may spot is that we compute divisor << shift twice each loop. We could improve upon this by shifting the divisor at the beginning, then shifting it down at the end of each loop to avoid any need to shift it elsewhere.  Let's have a look at the assembly code to make further improvements.  .globl DivideU32 DivideU32: result .req r0 remainder .req r1 shift .req r2 current .req r3  clz shift,r1 lsl current,r1,shift mov remainder,r0 mov result,#0  divideU32Loop$:  cmp shift,#0 blt divideU32Return$ cmp remainder,current  addge result,result,#1 subge remainder,current sub shift,#1 lsr current,#1 lsl result,#1 b divideU32Loop$  divideU32Return$: .unreq current mov pc,lr  .unreq result .unreq remainder .unreq shift  **clz dest,src** stores the number of zeros from the top to the first one of register **dest** to register **src**  You may, quite rightly, think that this looks quite efficient. It is pretty good, but division is a very expensive operation, and one we may wish to do quite often, so it would be good if we could improve the speed in any way. When looking to optimise code with a loop in it, it is always important to consider how many times the loop must run. In this case, the loop will run a maximum of 31 times for an input of 1. Without making special cases, this could often be improved easily. For example when dividing 1 by 1, no shift is required, yet we shift the divisor to each of the positions above it. This could be improved by simply using the new **clz** command on the dividend and subtracting this from the shift. In the case of 1 ÷ 1, this means shift would be set to 0, rightly indicating no shift is required. If this causes the shift to be negative, the divisor is bigger than the dividend and so we know the result is 0 remainder the dividend. Another quick check we could make is if the current value is ever 0, then we have a perfect division and can stop looping.  .globl DivideU32 DivideU32: result .req r0 remainder .req r1 shift .req r2 current .req r3  clz shift,r1 clz r3,r0 subs shift,r3 lsl current,r1,shift mov remainder,r0 mov result,#0 blt divideU32Return$  divideU32Loop$:  cmp remainder,current blt divideU32LoopContinue$  add result,result,#1 subs remainder,current lsleq result,shift  beq divideU32Return$  divideU32LoopContinue$:  subs shift,#1 lsrge current,#1 lslge result,#1 bge divideU32Loop$  divideU32Return$: .unreq current mov pc,lr  .unreq result .unreq remainder .unreq shift  Copy the code above to a file called 'maths.s'.  3 Number Strings  Now that we can do division, let's have another look at implementing number to string conversion. The following is pseudo code to convert numbers from registers into strings in up to base 36. By convention, a % b means the remainder of dividing a by b.  function SignedString(r0 is value, r1 is dest, r2 is base)  if value ≥ 0 then return UnsignedString(value, dest, base) otherwise  if dest > 0 then  setByte(dest, '-') set dest to dest + 1  end if return UnsignedString(-value, dest, base) + 1  end if  end function  function UnsignedString(r0 is value, r1 is dest, r2 is base)  set length to 0 do  set (value, rem) to DivideU32(value, base) if rem &gt 10 then set rem to rem + '0' otherwise set rem to rem - 10 + 'a' if dest > 0 then setByte(dest + length, rem) set length to length + 1  while value > 0 if dest > 0 then ReverseString(dest, length) return length  end function  function ReverseString(r0 is string, r1 is length)  set end to string + length - 1 while end > start  set temp1 to readByte(start) set temp2 to readByte(end) setByte(start, temp2) setByte(end, temp1) set start to start + 1 set end to end - 1  end while  end function  In a file called 'text.s' implement the above. Remember that if you get stuck, a full solution can be found on the downloads page.  4 Format Strings  Let's get back to our string formatting method. Since we're programming our own operating system, we can add or change formatting rules as we please. We may find it useful to add a %b operation that outputs a number in binary, and if you're not using null terminated strings, you may wish to alter the behaviour of %s to take the length of the string from another argument, or from a length prefix if you wish. I will use a null terminator in the example below.  One of the main obstacles to implementing this function is that the number of arguments varies. According to the ABI, additional arguments are pushed onto the stack before calling the method in reverse order. So, for example, if we wish to call our method with 8 parameters; 1,2,3,4,5,6,7 and 8, we would do the following:   1. Set r0 = 5, r1 = 6, r2 = 7, r3 = 8 2. Push {r0,r1,r2,r3} 3. Set r0 = 1, r1 = 2, r2 = 3, r3 = 4 4. Call the function 5. Add sp,#4\*4   Now we must decide what arguments our function actually needs. In my case, I used the format string address in r0, the length of the format string in r1, the destination string address in r2, followed by the list of arguments required, starting in r3 and continuing on the stack as above. If you wish to use a null terminated format string, the parameter in r1 can be removed. If you wish to have a maximum buffer length, you could store this in r3. As an additional modification, I think it is useful to alter the function so that if the destination string address is 0, no string is outputted, but an accurate length is still returned, so that the length of a formatted string can be accurately determined.  If you wish to attempt the implementation on your own, try it now. If not, I will first construct the pseudo code for the method, then give the assembly code implementation.  function StringFormat(r0 is format, r1 is formatLength, r2 is dest, ...)  set index to 0 set length to 0 while index < formatLength  if readByte(format + index) = '%' then  set index to index + 1 if readByte(format + index) = '%' then  if dest > 0 then setByte(dest + length, '%') set length to length + 1  otherwise if readByte(format + index) = 'c' then  if dest > 0 then setByte(dest + length, nextArg) set length to length + 1  otherwise if readByte(format + index) = 'd' or 'i' then  set length to length + SignedString(nextArg, dest, 10)  otherwise if readByte(format + index) = 'o' then  set length to length + UnsignedString(nextArg, dest, 8)  otherwise if readByte(format + index) = 'u' then  set length to length + UnsignedString(nextArg, dest, 10)  otherwise if readByte(format + index) = 'b' then  set length to length + UnsignedString(nextArg, dest, 2)  otherwise if readByte(format + index) = 'x' then  set length to length + UnsignedString(nextArg, dest, 16)  otherwise if readByte(format + index) = 's' then  set str to nextArg while getByte(str) != '\0'  if dest > 0 then setByte(dest + length, getByte(str)) set length to length + 1 set str to str + 1  loop  otherwise if readByte(format + index) = 'n' then  setWord(nextArg, length)  end if  otherwise  if dest > 0 then setByte(dest + length, readByte(format + index)) set length to length + 1  end if set index to index + 1  loop return length  end function  Although this function is massive, it is quite straightforward. Most of the code goes into checking all the various conditions, the code for each one is simple. Further, all the various unsigned integer cases are the same but for the base, and so can be summarised in assembly. This is given below.  .globl FormatString FormatString: format .req r4 formatLength .req r5 dest .req r6 nextArg .req r7 argList .req r8 length .req r9  push {r4,r5,r6,r7,r8,r9,lr} mov format,r0 mov formatLength,r1 mov dest,r2 mov nextArg,r3 add argList,sp,#7\*4 mov length,#0  formatLoop$:  subs formatLength,#1 movlt r0,length poplt {r4,r5,r6,r7,r8,r9,pc}  ldrb r0,[format] add format,#1 teq r0,#'%' beq formatArg$  formatChar$:  teq dest,#0 strneb r0,[dest] addne dest,#1 add length,#1 b formatLoop$  formatArg$:  subs formatLength,#1 movlt r0,length poplt {r4,r5,r6,r7,r8,r9,pc}  ldrb r0,[format] add format,#1 teq r0,#'%' beq formatChar$  teq r0,#'c' moveq r0,nextArg ldreq nextArg,[argList] addeq argList,#4 beq formatChar$  teq r0,#'s' beq formatString$  teq r0,#'d' beq formatSigned$  teq r0,#'u' teqne r0,#'x' teqne r0,#'b' teqne r0,#'o' beq formatUnsigned$  b formatLoop$  formatString$:  ldrb r0,[nextArg] teq r0,#0x0 ldreq nextArg,[argList] addeq argList,#4 beq formatLoop$ add length,#1 teq dest,#0 strneb r0,[dest] addne dest,#1 add nextArg,#1 b formatString$  formatSigned$:  mov r0,nextArg ldr nextArg,[argList] add argList,#4 mov r1,dest mov r2,#10 bl SignedString teq dest,#0 addne dest,r0 add length,r0 b formatLoop$  formatUnsigned$:  teq r0,#'u' moveq r2,#10 teq r0,#'x' moveq r2,#16 teq r0,#'b' moveq r2,#2 teq r0,#'o' moveq r2,#8  mov r0,nextArg ldr nextArg,[argList] add argList,#4 mov r1,dest bl UnsignedString teq dest,#0 addne dest,r0 add length,r0 b formatLoop$  5 Convert OS  Feel free to try using this method however you wish. As an example, here is the code to generate a conversion chart from base 10 to binary to hexadecimal to octal and to ASCII.  Delete all code after **bl SetGraphicsAddress** in 'main.s' and replace it with the following:  mov r4,#0 loop$: ldr r0,=format mov r1,#formatEnd-format ldr r2,=formatEnd lsr r3,r4,#4 push {r3} push {r3} push {r3} push {r3} bl FormatString add sp,#16  mov r1,r0 ldr r0,=formatEnd mov r2,#0 mov r3,r4  cmp r3,#768-16 subhi r3,#768 addhi r2,#256 cmp r3,#768-16 subhi r3,#768 addhi r2,#256 cmp r3,#768-16 subhi r3,#768 addhi r2,#256  bl DrawString  add r4,#16 b loop$  .section .data format: .ascii "%d=0b%b=0x%x=0%o='%c'" formatEnd:  Can you work out what will happen before testing? Particularly what happens for r3 ≥ 128? Try it on the Raspberry Pi to see if you're right. If it doesn't work, please see our troubleshooting page.  When it does work, congratulations, you've completed the Screen04 tutorial, and reached the end of the screen series! We've learned about pixels and frame buffers, and how these apply to the Raspberry Pi. We've learned how to draw simple lines, and also how to draw characters, as well as the invaluable skill of formatting numbers into text. We now have all that you would need to make graphical output on an Operating System. Can you make some more drawing methods? What about 3D graphics? Can you implement a 24bit frame buffer? What about reading the size of the framebuffer in from the command line?  The next series is the [Input](http://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/os/input01.html) series, which teaches how to use the keyboard and mouse to really get towards a traditional console computer. | 第九课 屏幕04  屏幕04的课程是以屏幕03课程为基础的。这一课将教授如何处理控制文本。这里假设你拥有了第八课：屏幕03操作系统的代码了。  目录   1. 字符串操作 2. 除法 3. 数字字符串 4. 格式化字符串 5. 改变操作系统 6. 字符串操作   在汇编语言代码中，可变参数函数看起来缺少直观。然后，它们却是非常有用且是个强有力的概念。  能够绘制文本却是有趣，但是，很不幸的是，此刻我们只能绘制已经准备好的字符串。显示命令行是足够了。但是我们的理想是要显示我们想要显示的任何事情。像往常一样，如果我们努力地去做一个超级棒的函数，来执行我们之前一直想要字符串处理，作为回报，之后我们的代码编写将变得非常容易。之前的C语言中如此复杂的函数是sprintf。这个函数根据另外一个字符串和额外的参数来生成一个字符串。这个函数有意思的地方是它的可变参数。这意味着该函数可以接受一个可变的参数列表来作为输入。参数的数量依赖于严格的格式化的字符串，并且不可以提前确定。  整个函数有很多选项，这里我仅仅列出了其中的一些。这里我把在整个课程中要实现的部分进行了高亮提示，但这并不妨碍你去实现更多的功能。  函数的工作过程是先读取格式化字符串，然后使用下面的表格中的信息去解释字符串。一旦一个参数被使用到，它就不会再次被考虑到。函数的返回值是被输入的字符串中字符的数量。如果函数失败了，它将会返回一个负数。  表1.1 sprintf格式规则   | Table 1.1 sprintf formatting rules | | | --- | --- | | **Sequence** | **Meaning** | | *Any character except %* | Copies the character to the output. | | %% | Writes a % character to the output. | | %c | Writes the next argument as a character. | | %d *or* %i | Writes the next argument as a base 10 signed integer. | | %e | Writes the next argument in scientific notation using e**N** to mean ×10**N**. | | %E | Writes the next argument in scientific notation using E**N** to mean ×10**N**. | | %f | Writes the next argument as a decimal IEEE 754 floating point number. | | %g | Same as the shorter of %e and %f. | | %G | Same as the shorter of %E and %f. | | %o | Writes the next argument as a base 8 unsigned integer. | | %s | Writes the next argument as if it were a pointer to a null terminated string. | | %u | Writes the next argument as a base 10 unsigned integer. | | %x | Writes the next argument as a base 16 unsigned integer, with lowercase a,b,c,d,e and f. | | %X | Writes the next argument as a base 16 unsigned integer, with uppercase A,B,C,D,E and F. | | %p | Writes the next argument as a pointer address. | | %n | Writes nothing. Copies instead the number of characters written so far to the location addressed by the next argument. |   实际上上述表格还有许多额外的调整，诸如明确字符串最小长度，正负号等。进一步学习，请参考C++参考手册中sprintf函数部分。  这里列举了一些调用该函数的例子，通过观察它们的返回值我们可以了解如何使用该函数。  表1.2 sprintf函数的调用示例   | **Format String** | **Arguments** | **Result** | | --- | --- | --- | | "%d" | 13 | "13" | | "+%d degrees" | 12 | "+12 degrees" | | "+%x degrees" | 24 | "+1c degrees" | | "'%c' = 0%o" | 65, 65 | "'A' = 0101" | | "%d \* %d%% = %d" | 200, 40, 80 | "200 \* 40% = 80" | | "+%d degrees" | -5 | "+-5 degrees" | | "+%u degrees" | -5 | "+4294967291 degrees" |   真心希望你以及了解到了整个函数的用途。编写这个函数的确需要大量的工作要做，但是给我们的奖励确是非常诱惑的——我们将得到一个通用的函数，可以大大地提高以后的工作效率。   1. 除法   除法是最慢的同时也是最复杂的基础数学类操作。在ARM汇编语言中并没有直接实现它，这是因为它将花费很长一段时间来得到答案，并且这不会是一个很简单的操作。  这个函数看起来很给力，不过也非常复杂。处理它的许多示例的最容易的办法或许是编写一个函数来处理一些普通的示例。一个很有帮助的想法是编写一个函数去生成一个基于任何进制的有符号数或者无符号数的字符串。那么，我们该怎么做呢？在继续阅读之前，请自己试着设计一个算法。  最简单的办法可能是我在第一课：OK01中提到的严格办法。我们称它为余数方法。这个想法可以表述以下：   1. 把当前值除以正在使用的进制的基数。 2. 存储余数。 3. 如果得到的新的值不是0，转到步骤1。 4. 反转余数序列，所得结果即是答案。   例如：  表2.1 2进制换算示例   | **Value** | **New Value** | **Remainder** | | --- | --- | --- | | 137 | 68 | 1 | | 68 | 34 | 0 | | 34 | 17 | 0 | | 17 | 8 | 1 | | 8 | 4 | 0 | | 4 | 2 | 0 | | 2 | 1 | 0 | | 1 | 0 | 1 |   那么，答案就是100010012。  这个过程不幸的部分就是它不可避免地要用到除法。因此，我们不得不首先考虑一下二进制的除法。  长除法的扩展参考在下面的方框里。  长除法扩展阅读。  为了在汇编语言中实现除法，我们将要实现二进制长除法。这么做的原因是数值存储都是以二进制为格式的，而二进制的所有重要的位移操作都很容易实现，并且二进制的除法要比其他任何进制的除法来的简单，这主要是因为示例的阶数。  1011 r 1  1010)1101111  1010  11111  1010  1011  1010  1  这个例子二进制长除法是如何工作的。只要没有超出被除数，那就简单的把除数一直往右移动。根据位置，来输出一个1，并且减去这个数。不管是什么都是余数。本例中，11011112÷ 10102 = 10112余数是12。十进制中，111 ÷ 10 = 11余数是1。  现在，请自己尝试着去实现长除法。你应该编写一个函数DivideU32，它用来把寄存器r0的数值除以r1的数值，结果方在r0中，而余数放在r1中。我们将经历一个非常有效的实现过程。  function DivideU32(r0 is dividend, r1 is divisor)  set shift to 31 set result to 0 while shift ≥ 0  if dividend ≥ (divisor << shift) then  set dividend to dividend - (divisor <&lt shift) set result to result + 1  end if set result to result << 1 set shift to shift - 1  loop return (result, dividend)  end function  这段代码确实实现了我们想要的功能，但是却并不是汇编语言级别的代码。我们的问题来自于一个事实——我们的寄存器只能存储32位数据，因而指令divisor << shift的执行结果将无法适配到寄存器中（我们称之为溢出）。这是个真实的问题。你的解决方案右溢出吗？  很幸运，存在一个叫clz或者统计头零的指令。该指令会从一个数值的二进制表示的最高位开始统计零的个数。这个指令可以严格地告知我们，在溢出之前，我们可以进行多少次左移。另一个你可以关注的优化举措是我们在每次循环中进行了两次divisor << shift操作。我们可以在开始的地方移动除数来改进这个操作，然后，在每次循环的结尾处把其移下来，以避免其他地方有移动需求。  让我们看一看下面的汇编代码，来做一些改进工作。  .globl DivideU32 DivideU32: result .req r0 remainder .req r1 shift .req r2 current .req r3  clz shift,r1 lsl current,r1,shift mov remainder,r0 mov result,#0  divideU32Loop$:  cmp shift,#0 blt divideU32Return$ cmp remainder,current  addge result,result,#1 subge remainder,current sub shift,#1 lsr current,#1 lsl result,#1 b divideU32Loop$  divideU32Return$: .unreq current mov pc,lr  .unreq result .unreq remainder .unreq shift  指令clz dest, src将把dest寄存器中开始的零的个数存储在寄存器src中。  或许，你已经看出来了。这个函数的效率看起来还不错。的确，这个函数确实很漂亮，但是你要谨记，除法操作是一个代价昂贵的操作。因为我们以后的操作系统和其上的应用会大量的使用到该函数，所以，对该函数的任何改进，效果将是非常棒的。当我们想要优化一段包含循环的代码，我们应该总是下意识地考虑这个循环要执行多少次。在本例中，当输入是1时，循环将执行最大的次数，31次。不用其他特殊的手段，改进这个循环还是很容易的。例如当计算1除以1时，根本不需要进行移动，也不需要在其上的每个位置都移动除数。可以使用新指令clz来改进这个操作。只需要把其放在除法的地方并且把移动操作删除即可。1除以1的情况下，移动的次数会是0，这正好证实了并不需要移动操作。如果其导致移动为负数时，那意味着除数大于被除数，这样商是0，余数就是被除数。另外一个快速测试的用例就是：被除数是0的情况。在此种情况下，我们将拥有一个完美的除法，并且我们能够把循环停下来。  .globl DivideU32 DivideU32: result .req r0 remainder .req r1 shift .req r2 current .req r3  clz shift,r1 clz r3,r0 subs shift,r3 lsl current,r1,shift mov remainder,r0 mov result,#0 blt divideU32Return$  divideU32Loop$:  cmp remainder,current blt divideU32LoopContinue$  add result,result,#1 subs remainder,current lsleq result,shift  beq divideU32Return$  divideU32LoopContinue$:  subs shift,#1 lsrge current,#1 lslge result,#1 bge divideU32Loop$  divideU32Return$: .unreq current mov pc,lr  .unreq result .unreq remainder .unreq shift  把上面的代码拷贝并复制文件“maths.s”里。   1. 数字字符串   既然我们可以做除法了，那就让我们来看看如何实现把数值转换成字符串吧。下面的伪代码可以把寄存器里的数值转换成包括36进制在内的任何进制字符串。提前声明一下，a % b的意思是a除以b的余数。  function SignedString(r0 is value, r1 is dest, r2 is base)  if value ≥ 0 then return UnsignedString(value, dest, base) otherwise  if dest > 0 then  setByte(dest, '-') set dest to dest + 1  end if return UnsignedString(-value, dest, base) + 1  end if  end function  function UnsignedString(r0 is value, r1 is dest, r2 is base)  set length to 0 do  set (value, rem) to DivideU32(value, base) if rem &gt 10 then set rem to rem + '0' otherwise set rem to rem - 10 + 'a' if dest > 0 then setByte(dest + length, rem) set length to length + 1  while value > 0 if dest > 0 then ReverseString(dest, length) return length  end function  function ReverseString(r0 is string, r1 is length)  set end to string + length - 1 while end > start  set temp1 to readByte(start) set temp2 to readByte(end) setByte(start, temp2) setByte(end, temp1) set start to start + 1 set end to end - 1  end while  end function  一个名叫“text.s”的文件实现了上述伪代码。再次告知一下，无论在任何地方困住了，你都可以在下载页中找到解答。   1. 格式化字符串   让我们回忆一下之前的字符串格式化函数。因为我们正在编写我们自己的操作系统，因此我们可以根据我们自己的愿望来增加或者改变格式化规则。我们或许发现增加a % b操作是非常有用的。该操作将会输出一个数值的二进制形式。如果没有使用空终止符，你或许想要改变%s的行为，以便从另外一个参数中获取字符串的长度信息，亦或者也可以从一个长度前缀中得到。我将会在下面的例子中使用空终止符。  实现这个函数的一个最主要的障碍是可变参数的数量。根据ABI规定，在调用函数之前，要把额外的参数以逆序压入堆栈中。据此我们举个例子。如果我们想要调用我们的函数，参数是8个，分别是1，2，3，4，5，6，7和8。那么下面是我们要做的：   1. Set r0 = 5, r1 = 6, r2 = 7, r3 = 8 2. Push {r0, r1, r2, r3} 3. Set r0 = 1, r1 = 2, r2 = 3, r3 = 4 4. Call the function 5. Add sp, #4\*4   现在，我们必须决定我们的函数实际需要什么样的参数。在我们的例子中，我把格式化字符串的地址放在r0中，它的长度信息放在r1中，目标字符串地址放在r2中，后面紧跟着我们需要的参数的列表，以r3开头且像上面那样在栈中延续排列。如果你想要使用带有空终止符格式字符串的话，寄存器r1中的参数可以移除。如果你想要使用缓存的最大长度的话，你可以把它存储在寄存器r3中。作为一个附加的修改，我认为改变函数是有用处的。当目标字符串的地址为0，且没有字符串输出，但准确的长度仍然返回时，因而一个格式化的字符串的长度可以被精确地计算出。  如果   1. 改变操作系统 |