Homework #2

Due 11:59pm Sunday, 18 September, 2016
Multiple submissions accepted.
25% penalty per day per full or partial .

Problems are from chapter 2. But one clear goal of this assignment is to get you very familiar with section 10 of Appendix A. You will need it.

The book's authors suggest installing SPIM. Details on how to do that follow the homework. I did not need SPIM at all to complete the problems, so I would recommend trying without it first.

Also note that the problems are changed from last year – so although the problem numbers are the same, I've made adjustments to the actual instructions and equations in many places. WORK THE PROBLEMS AS THEY ARE WRITTEN HERE. Do not just take the problem numbers and go back to the exercise in the book. Many are slightly different.

Problems:

2.1 [5] <§2.2> For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables f , g , & h are given and could be considered 32-bit integers as declared in a C program. Use a minimal number of MIPS assembly instructions.

```
f = (g – 9) + h;
```

Hint: avoid using pseudo instructions. Page A-51 in the text lists all the MIPS commands and indicates the ones that are pseudo instructions. *Pseudoinstructions*, appear as real instructions in assembly language programs. The hardware, however, knows nothing about *pseudoinstructions*, so the assembler must translate them into equivalent sequences of actual machine instructions. A *pseudoinstruction* expands to several machine instructions.

2.2 [5] <§2.2> For the following MIPS assembly instructions below, what is a corresponding one-line C statement?

```
add  f, g, h
sub  f, f, i
```

2.3 [5] <§2.2, §2.3> For the following C statement, what is the corresponding MIPS assembly code? Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address

of the arrays A and B are in registers $s5 and $s6, respectively. Choose an appropriate register to store the result.

```
B[8] = A[i-j];
```

2.4 [5] <§2.2, §2.3> For the MIPS assembly instructions below, what is the corresponding C statement? Assume that the variables f, g, h, i, and j are assigned to registers $s0, $s1, $s2, $s3, and $s4, respectively. Assume that the base address of the arrays A and B are in registers $s6 and $s7, respectively.

```
sll $t0, $s0, 2   # $t0 = f * 4
add $t0, $s6, $t0 # $t0 = &A[f]
sll $t1, $s1, 2   # $t1 = g * 4
add $t1, $s7, $t1 # $t1 = &B[g]
lw  $s0, 0($t0)   # f = A[f]
addi $t2, $t0, 8
lw  $t0, 0($t2)
add $t0, $t0, $s0
sw  $t0, 0($t1)
```

2.5 [5] <§2.2, §2.3> For the MIPS assembly instructions in Exercise 2.4, rewrite the assembly code to minimize the number if MIPS instructions (if possible) needed to carry out the same function.

2.6 The table below shows 32-bit values of an array stored in memory.
Hint: Note that there is a typographical error in the book which shows A[1] as address 38. It is not. It is 28.

Array Index	Byte Address	Present Value	New Value	Array Index
A[0]	&24	2		
A[1]	&28	4		
A[2]	&32	3		
A[3]	&36	6		
A[4]	&40	1		

2.6.1 [5] <§2.2, §2.3> For the memory locations in the table above, write C code to sort the data from lowest value to highest value, placing the lowest value in the smallest memory location (&24).

Assume that the data shown represents the C variable called **`Array`**, which is an array of type **`int`**, and that the first number in the array shown is the first element in the array. Assume that this particular machine is a byte-addressable machine and a word consists of four bytes.

Hint: Byte-addressable memory in this problem is just like what we've discussed in class. You have to convert words to byte numbers. Also, for this first part, don't try and write code for a sorting algorithm; just write code to move the numbers as needed to sort them, using temporary variables as needed. Don't move constants, move Array[m] to Array[n], for whatever array index is appropriate. To make things easier I included a table where you can fist list the new values in the order being requested by the problem, and then list their associated Array Index you will need your code to move.

2.6.2 [5] <§2.2, § 2.3> For the memory locations in the table above, write MIPS code to sort the data from lowest to highest, placing the lowest value in the smallest memory location. Use a minimum number of MIPS instructions. Assume the base address of **`Array`** is stored in register **`$s6`**.

Hint: Remember you'll be repeatedly moving the contents around of a word of memory starting at some 4-byte multiple added to **`$s6`**. So the first is **`0($s6)`**, and putting that value into **`$t0`**, for example, would be a line of MIPS code that looks like this:

```
lw $t0, 0($s6)
```

Minimize instructions, not temporary variables. I was able to sort the array in 8 instructions, and the line above was my first line of code.

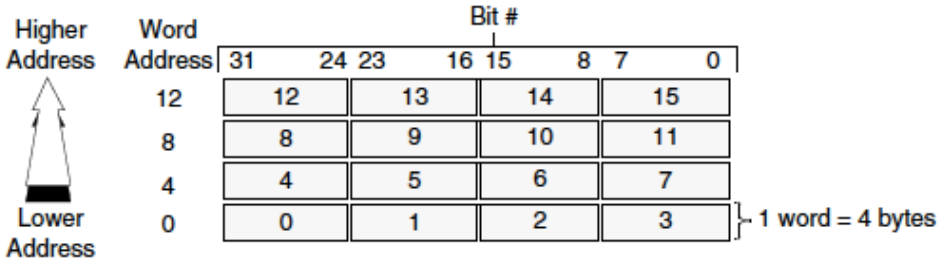
2.7 [5] <§2.3> Show how the value `0x`**`12efcdab`** (a.k.a. **`12efcdab`**$_{16}$) would be arranged in memory of a little-endian and a big-endian machine. Assume the data is stored starting at address 0.

Hint: Two diagrams below may help understand big- and little-endian orders. They're from the maker of MIPS hardware, not from our text. I will point out three things that are sometimes confusing:

- The bit order of bytes never changes. It's always right to left, 0 to 7. In a word, therefore, the bits are always 0 to 31, right to left. That never changes.
- The word address is always bottom-up in these diagrams. Word 0 is the bottom-most row, then word 1, and so on. Similarly, Byte 0-3 is always in the bottom row, 4-7 the next row up, etc.
- The difference is in how the bytes are numbered *within* a word. Is it *never* how you number bits with the word, or bits within the byte. So in Big-Endian, byte #2 contains bits 8 through 15, numbered right to left. In Little-Endian, byte #2 contains bits 16-23, also numbered right to left.

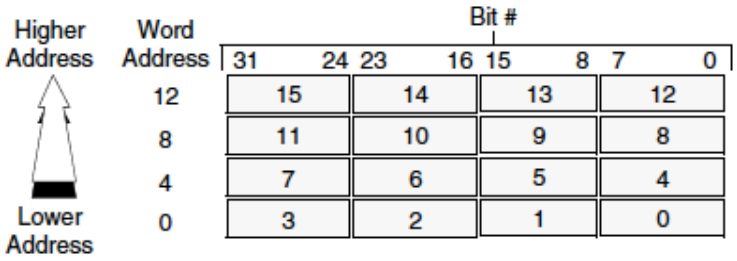
> When configured in **big-endian order**, byte 0 is the most-significant (left-hand) byte. Figure 4.2 shows this configuration.

Figure 4.2 Big-Endian Byte Ordering



When configured in **little-endian order**, byte 0 is always the least-significant (right-hand) byte. Figure 4.3 shows this configuration.

Figure 4.3 Little-Endian Byte Ordering



Use the following table for your answer:

Little-Endian		Big -Endian	
Address	Data	Address	Data
12		12	
8		8	
4		4	
0		0	

2.11 [15] <§2.2,§2.5> **This problem is heavily modified from the book.** Consider the following assembly code, based roughly on the C code for

```
i = N*N + 3*N
```

I've converted it to the following MIPS Code instructions:

```
lw      $t0, 4($gp)       # fetch N
mul     $t0, $t0, $t0     # N*N
lw      $t1, 4($gp)       # fetch N
ori     $t2, $zero, 3     # 3
mul     $t1, $t1, $t2     # 3*N
add     $t2, $t0, $t1     # N*N + 3*N
sw      $t2, 0($gp)       # i = ...
bgez    $t2, $zero 8      # if i≥0, skip jump
jal     0x7FFC            # else jump by 32764
```

Break the instructions into I-type, R-type or J-type. Some MIPS exercises have FR and FI type instructions, but we aren't using any of those here.

- Begin by placing the instruction into the appropriate table. Use Appendix A to guide you.
- Then fill out the fields based on the instruction type.
- Any numeric values you place in a field will be assumed to be in decimal unless you specify otherwise.

I've converted the first instruction for you, and included some annotations. Remember, the order of operands in a MIPS instruction represented in these tables is DIFFERENT than the way they're written down in assembler for us. Specifically, in a 3-operand command like an R-type command, rd is written first, then rs and rt.

R-Type Instructions	opcode (6 bits)	rs (5 bits)	rt (5 bits)	rd (5 bits)	shamt (5 bits)	funct (6 bits)
`mul $t0, $t0, $t0`	MULTIP 28	t0 8	t0 8	t0 8	shift 0	MUL 2

I-Type Instructions	opcode (6 bits)	rs (5 bits)	rt (5 bits)	Immed (16 bits)

J-Type Instructions	opcode (6 bits)	pseudo address (26 bits)

2.12 Assume that registers `$s0` and `$s1` hold the values `0x80000000` and `0xD0000000`, respectively.

2.12.1 [5] <§2.4> What is the value of `$t0` for the following assembly code?

```
add $t0, $s0, $s1
```

2.12.2 [5] <§2.4> Is the result in `$t0` the desired result, or has there been overflow?

2.12.3 [5] <§2.4> For the contents of registers `$s0` and `$s1` as specified above, what is the value of `$t0` for the following assembly code?

```
sub $t0, $s0, $s1
```

2.12.4 [5] <§2.4> Is the result in $t0 the desired result, or has there been overflow?

2.12.5 [5] <§2.4> For the contents of registers $s0 and $s1 as specified above, what is the value of $t0 for the following assembly code?

```
add $t0, $s0, $s1
add $t0, $t0, $s0
```

2.12.6 [5] <§2.4> Is the result in `$t0` the desired result, or has there been overflow?

Hint: Remember that these are numbers (not text or a line of code in this case) so 2's complement representation applies. You're given hex, so first convert it to a 32-bit binary number. If the resulting binary number is positive, you can convert it to decimal using any normal binary-to-decimal converter. If it's negative, though, you'll need to convert it to decimal a different way – by using a 2's complement-to-decimal converter. I've listed a few below, but you can find others I'm sure.

Once in decimal, and only once in decimal, do the math. If your decimal result is positive, convert the answer back to binary using a normal decimal-to-binary converter, but use a 2's complement decimal-to-binary converter if it's negative.

Is the binary string longer than 32 bits? If it is, you're in overflow.
Is the 32nd bit (technically bit 31) what you expect it to be? (1 if negative, 0 if positive)? If not, that's another sign (no pun intended) that you have overflowed (overflown?).

Whatever the rightmost 32 bits are, copy only those 32 bits into a binary-to-hex converter, and supply that as your answer.

Converters I found:
http://www.binaryhexconverter.com/hex-to-binary-converter
http://www.binaryhexconverter.com/binary-to-decimal-converter

http://www.exploringbinary.com/twos-complement-converter/

2.14 [5] <§2.2, §2.5> State the instruction type and provide the assembly language instruction for the following binary value:

```
1010 1101 0100 1001 0000 0000 0010 0000two
```

Hint: You are going to want to get very good at working these problems. Also, this problem is not the same as the one in the book (different binary).

2.15 [5] <§2.2, §2.5> Provide the type and hexadecimal representation of following instruction:

```
sw $s4, 16($t0)
```

Hint:

- Use Appendix A or your green card (both of which are on blackboard with the lecture slides) to begin to break this command apart:
- Start by slotting the commands into the blocks that make up that type of instruction. You know you'll have opcode, and a couple of registers and a number.
- Mark each entry with the # of bits it is required to take
- Convert each entry to binary with that many bits
- Combine the result into a single 32-bit binary string
- Break that string back up, this time into 8 sets of 4 bits
- Convert each set of 4 bits into a hexadecimal character
- Report the answer

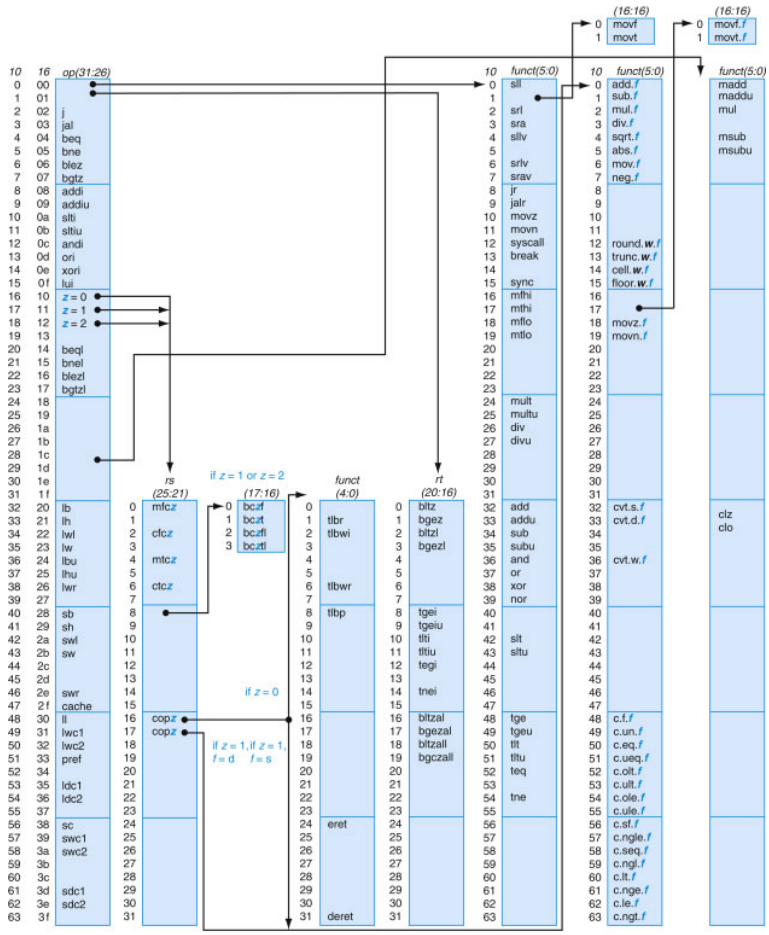
2.16 [5] <§2.5> Provide the type, assembly language instruction, and binary representation of instruction described by the following MIPS fields:

```
op=0, rs=2, rt=3, rd=4, shamt=0, funct=35
```

2.17 [5] <§2.5> Provide the type, assembly language instruction, and binary representation of instruction described by the following MIPS fields:

```
op=0x23, rs=1, rt=2, const=0x4
```

Hint: you could use SPIM for this, but I used appendix A and our class discussion to decipher the string. A diagram that helps you decode the first 6 bits and the function bits (if present) into your command, and therefore your instruction format type can be found in figure A.10.2 in Appendix A, page A-50:



Read the caption for that image carefully to see how to use this table.

The SPIM Simulator

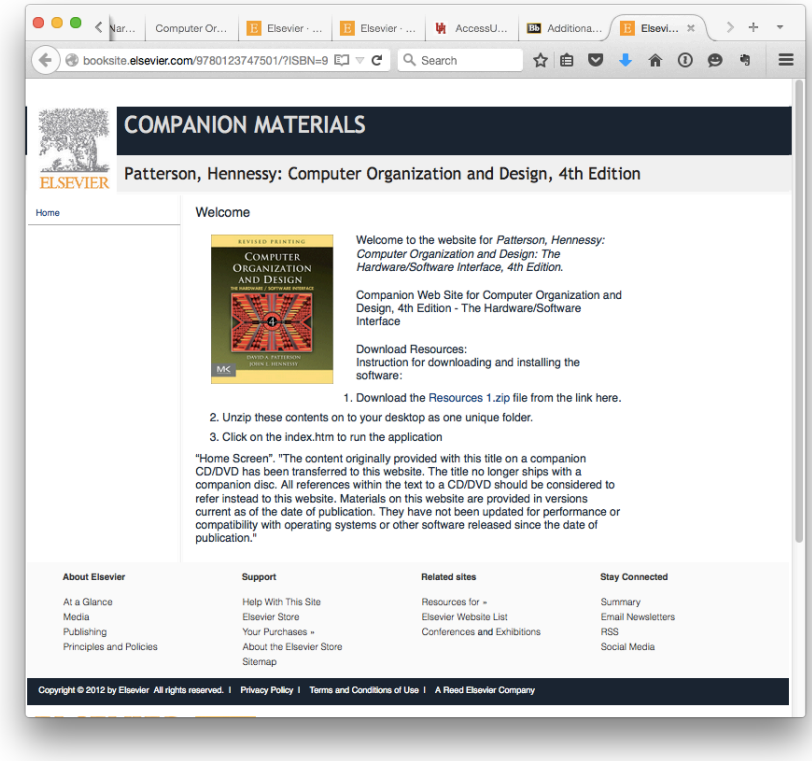
This is a little long, please hang in there.

The authors have suggested that the MIPS simulator (called SPIM) mentioned in Appendix A may be useful to the homework. It's an interesting package and is now available on Linux, Windows and on Macs (that's a new development).

If you don't want to read all of this, just skip to the end to download and install the software.

The simulator is described in section A.9, on page A-40 in Appendix A.
Follow these steps to get it up and running:

First, try this link to the companion web site for the 4th edition of our book:
http://booksite.elsevier.com/9780123747501/?ISBN=9780123747501
If it works for you like it did for me, you'll get a page that looks like this:

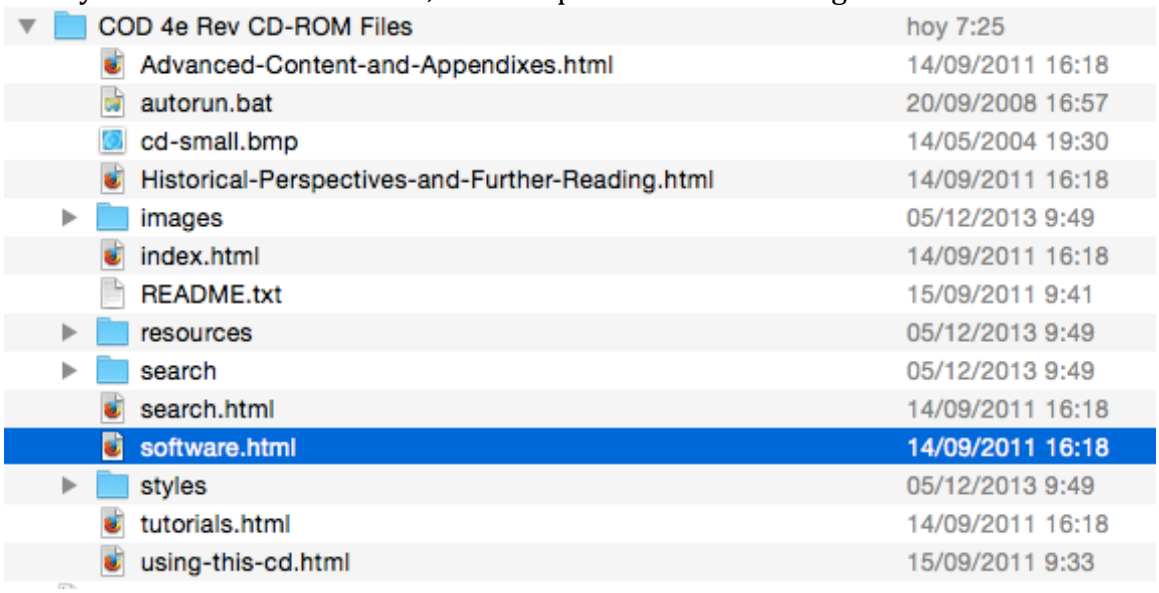


It has a link to a Resources_1.zip file, which you should download.

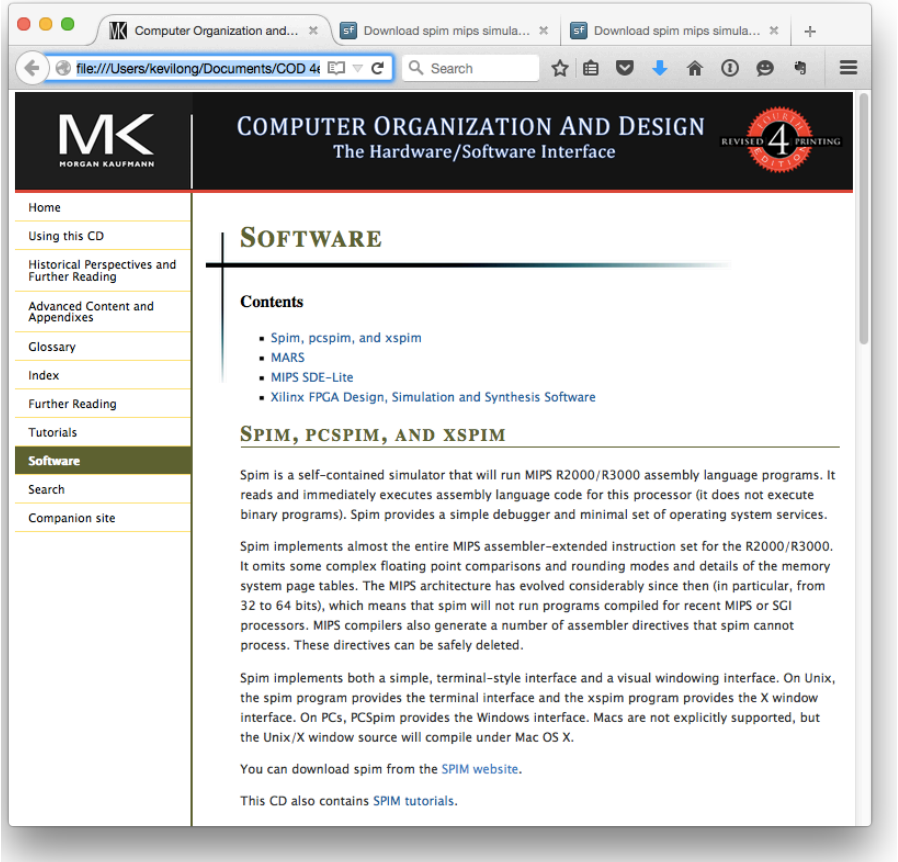
If you can't access this page or the download, you can pull the ZIP file directly from our blackboard site. Look in the **Contents** folder on blackboard, under **Additional Textbook Publisher Content**. Look for a file called

"SPIM Simulator info and other 4th edition materials – ZIP file"

Once you download the ZIP file, it will expand to the following folder:

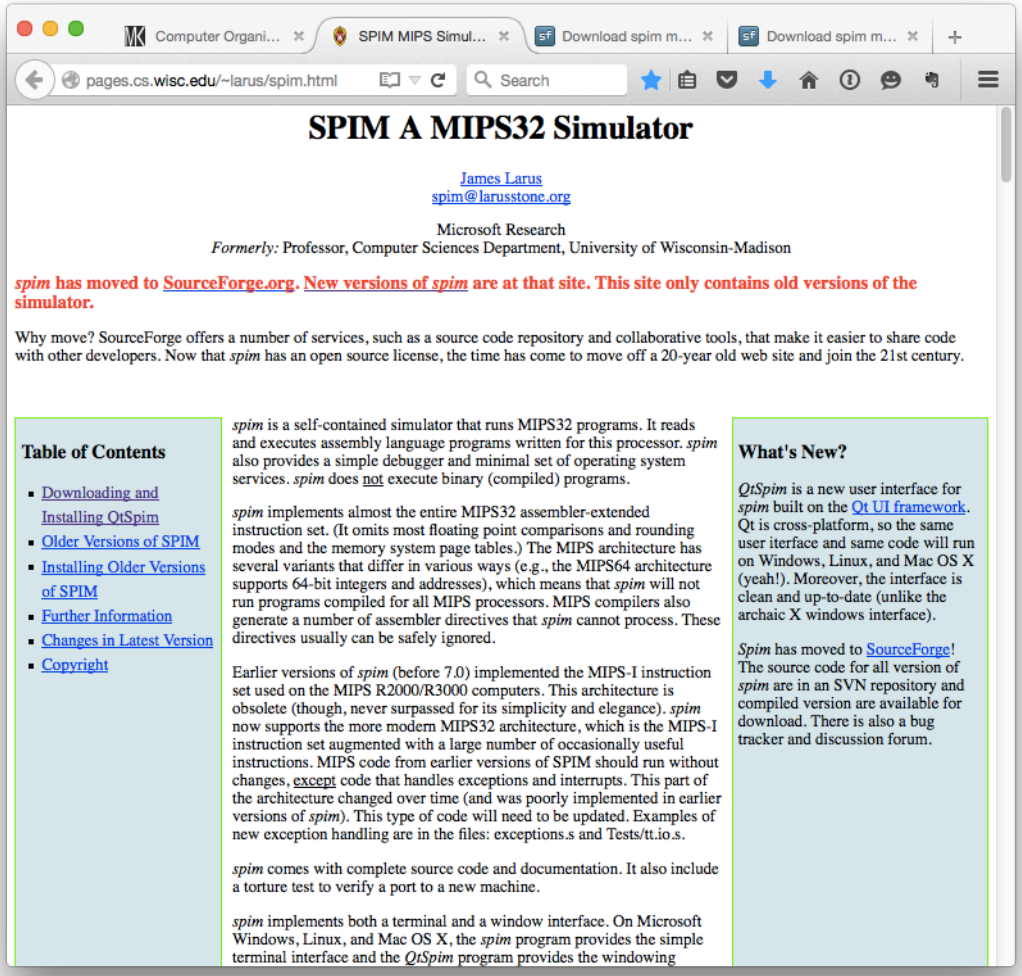


There's a bunch of stuff in here that has nothing to do with SPIM. Open the "software.html" document:



You'll find a link to the "SPIM website" in the first bullet. It points here:
http://pages.cs.wisc.edu/~larus/spim.html

James Larus from UW-Madison (and Microsoft Research) is the author of SPIM.



Note the "**Downloading and Installing QtSpim**" link on the left. Follow that link to scroll down to this paragraph:

> A compiled, immediately installable version of *QtSpim* is available for Microsoft Windows, Mac OS X, and Linux can be downloaded from: https://sourceforge.net/projects/spimsimulator/files/.

Follow that link to sourceforge, a code sharing site. Links to the downloads are found there. The newest versions are at the top. There are versions for 32- and 64-bit Linux architectures, one for Windows and one for Mac.

Here are the links to the downloads you need. I'm sure there are some tricks to getting this all up and running, I've included some for Mac, since I tested that.

32-bit Linux: qtspim_9.1.16_linux32.deb

http://sourceforge.net/projects/spimsimulator/files/qtspim_9.1.16_linux32.deb/download

Mac: QtSpim_9.1.16_mac.mpkg.zip

http://sourceforge.net/projects/spimsimulator/files/QtSpim_9.1.16_mac.mpkg.zip/download

Hint: when you download and expand the ZIP file, you'll have a QtSpim.mpkg file that may require you to control-click the package and selecting "Open" since the Mac's Gatekeeper package may prevent it from opening by just double-clicking. You'll get a QtSim program installed in your Applications folder.

64-bit Linux: qtspim_9.1.16_linux64.deb

http://sourceforge.net/projects/spimsimulator/files/qtspim_9.1.16_linux64.deb/download

Windows: QtSpim_9.1.16_Windows.exe

http://sourceforge.net/projects/spimsimulator/files/QtSpim_9.1.16_Windows.exe/download