Frequently Asked Questions

Q1. Where do I get information about SPIM?
A. http://www.cs.wisc.edu/~larus/spim.html

Q2. Is memory supposed to be an array of words, or an array of bytes?

As in.
mem[1] = 32 bit word
mem[4] = 32 bit word
mem[8] = 32 bit word

Or if it's an array of bytes
mem[0] = 8 bit byte
mem[1] = 8 bit byte
mem[2] = 8 bit byte
mem[3] = 8 bit byte
mem[4] = 8 bit byte //the first byte of a new word.

So if we want to read in an instruction from memory we would read it in 4 8-bit chunks.

A. Functions that require it are instruction_fetch and rw_memory and the array Mem is passed as an argument.
The little table from the first page of the spec gives the answer:
Notice the "Mem[0]" in the first row, so the memory goes like this
Mem[0] = 0x0000 - 0x0003
Mem[1] = 0x0004 - 0x0007
...
Mem[16384] = 0xFFFC - 0xFFFF (65532 - 65536)

It also says that all program starts at memory location 0x4000 (see def of PCINIT in spimcore.c) which corresponds to Mem[4096]. (4096 = 0x1000)
The trick is to test for word alignment while it is still in address form, then reference the proper index in the Mem array by shifting the address right twice.

Q3. What are the inputs and outputs of the program?

A. Most of the functionality of this simulator is provided by spimcore. The only input that you need to provide to spimcore is a text file with extension .asc, which should contain the 32-bit instructions as ASCII in hexadecimal format (see the example in the project description). You can write a sequence of instructions manually in your .asc file, or optionally write a simple assembler to do that for you (i.e. convert a program to its hex sequence).

There is no output. Spimcore takes care of the simulation. But you need to make sure that the datapath/control are working correctly. For your convenience, the diagram in Figure 2 of the project description color-codes each function to be implemented with dotted lines.

Q4. How does *instruction_fetch()* work?

A. The data you are given is in Mem which is an array that is filled in the main() from the data supplied to the program in your .asc file. PC is the full 32-bit address of the next instruction in Mem[]. But with a little twist: You have to use (PC >> 2) whenever you use it with the Mem array (or you need to use the pre-defined macro MEM(PC), which shifts its argument right by 2 for you). The info that is in this location ref: by Mem[(PC >> 2)] is the decimal value of the instruction that was in Hex in the file. As of for checking to see if it's word is aligned ...maybe you should read the book!

Note that 64k (bytes 0-65,536) of memory are available, but again they are accessed as words (unsigned Mem[]). Therefore for a given address, you have to determine its word offset, which is the index to the array Mem[].

Q5. What do "RegDst" and "ALUSrc" represent?

A.
RegDst defines which register is the destination register of an instruction. If the instruction is an R-type then the destination register is given by bits 15-11 of the instruction and if the instruction is I-type or branch then the destination register is given by bits 20-16 of the instruction. If you look at the diagram in Figure 2 of the project description, then you can see that for the latter case RegDst=0 and for the former RegDst=1.

ALUSrc determines the source input of the ALU. Again this can be readily determined from the diagram in Figure 2 of the project description. For R-type instructions the second input to the ALU is given by the register specified by bits 20-16, which appear directly at the output Read Data2 of the register file. For I-type and branching, however, the second input to ALU is coming from the Sign Extend unit. Therefore based on the diagram in Figure 2 of the project description, in the former case ALUSrc=0 and in the latter ALUSrc=1.

Q6. Can you explain what the input file to the assembler should contain? I know the output .asc file is hex. Are we supposed to convert things like addi $t0 $t1 5 to their hex format?

A. You shouldn't worry about the assembler at first, but yes, it is supposed to take in MIPS instructions in plain text and output these to their hex counterparts.

The instruction part would need a switch statement with 14 cases and then you would need another switch statement with 32 cases for the register values. And that is just the "simple" part. The jump and branch translations are more complicated. jump uses an

absolute address to tell the PC what it should be next. The branch instructions work with PC-relative branching (see the book or lecture notes).

If you just want to be able to run spimcore and you only need one example .asc file then you may want to write one manually (using Appendix B) with all the instructions that you need to implement!! See the project description for an example.

Q7. Are the functions in the project described in better detail anywhere in the book? Or is there somewhere where I can better understand what the arguments for each function are supposed to do?

A. Ok... I think CH4 in the book is the magic chapter, for those of you who are still running around in circles. Yes read the first few sections of chapter 4.

If you want a better understanding of what the functions do, take a close look at the diagram in Figure 2 of the project description. Each function is identified and color-coded and represents a certain section of schematic. If you look carefully, you will see that each argument in a function actually represents some piece of physical hardware in the diagram.

Here is how the arguments work;
Pointers, such as unsigned* and char* are values that you need to set, besides *Mem and *Reg, which are arrays, can go either way. The other values are what you use to determine how to assign the pointers.

A really easy function that uses all of these argument types is read_register.
Here are the arguments;
unsigned r1
unsigned r2
unsigned *Reg
unsigned *data1
unsigned *data2

All this function consists of are two assignment statements, just two lines of code. Just remember that *Reg is actually an array, so you can think of it as unsigned Reg[] instead.

Q8. What R-type instructions are we supposed to set when ALUOp is equal to "111"? It just says "instruction is an R-type". I'm confused....

A. When the ALUOp control signal is 7 or 111, this tells the ALU control that it needs to figure out what operation to tell the ALU to do by looking at the funct field (bits[5-0]) of the instruction. This is called "multiple levels of decoding".

Q9. What are argc and argv for?

A. The argc stands for "argument count" and it is also the number of elements in the argv

array. char **argv could also be written as char *argv[].

But that doesn't matter for this project. spimcore.c handles everything for you, it also contains the main method.

Unless you changed something in spimcore.c, it should work fine. Don't forget to compile it according to the directions.

Q10. Should we halt in rw_memory() in the event that ((ALUresult % 4)!=0)?
A. Yes that is correct, but only halt when ALUresult is an address. ALUresult should be an address if MemRead or MemWrite is asserted. For instance, on an I-type command, the ALUresult would hold a memory address.

Q11. Under the ALU_operations function, what are we supposed to do with the arguments, extended_value and ALUsrc ?

A. If you look at the diagram in Figure 2 of the project description, your ALUsrc is a control signal to a multiplexer, which chooses between a sign extended_value, or data2 in order to send the outcome to the ALU for operation.

Q12. What are we supposed to do with the Zero parameter being passed into many of the functions?

A. Zero parameter indicates that the result of the operation performed by ALU was zero. This can be for instance used for conditional branching.

Q13. What exactly does "sign-extend" mean, and how can I figure out the sign extended value of a number?

A. If you still do not know after all these lectures how to sign-extend then… O.K. All it means is "convert a 16-bit binary number into 32-bits", while keeping its sign as it is.

Q14. How do we know when a control signal is a “don’t care”?

A. A control signal is a “don’t care” for an instruction, if its value has no effect on the correct operation of datapath for that instruction. For instance, the ALUOp signals have no impact on the correct operation of the datapath for the jump instruction. Also, since we will not write to a destination register, the value of RegDst would be irrelevant for jump.