Frequently Asked Questions

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

As in, an array of words:
mem[0] = 32 bit word
mem[1] = 32 bit word
mem[2] = 32 bit word

Or, 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.

In the case of an array of bytes, if we want to read in an instruction from memory we would need to 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 project guidelines 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[16383] = 0xFFFC - 0xFFFF (65532 - 65536)

Therefore, memory is an array of words.

It also says that all program starts at memory location 0x4000 (see definition 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.

Q. 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 Appendix A, Figure 2 of the project description color-codes each function to be implemented with dotted lines.

Q. How does *instruction_fetch()* work?

A. The data you are given is in Mem. Mem is an array that is populated by functions in spimcore.c. The data is supplied to the program by the .asc input file. PC is the index of Mem[], but with a little twist: You have to use (PC >> 2) whenever you use it. The information that is in this location referenced by Mem[(PC >> 2)] is the decimal value of the instruction that was in Hex in the file.

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

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

A. These are control signals established by the control unit.

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 of the second input of the ALU. This can be 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 Data 2 of the register file. For most I-type instructions, however, the second input to ALU is coming from the Sign Extend unit. BEQ is an exception here – being an I-Type instruction that relies on Read Data 2 to determine whether or not to branch. Therefore based on the diagram in Figure 2 of the project description, R-Types and Branch utilize ALUSrc=0 while other I-Types us ALUSrc=1.

Q. What R-type instructions are we supposed to set when ALUOp is equal to "111"? It just says "instruction is an R-type".

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 what to do by looking at the funct field (bits[5-0]) of the instruction. This is called "multiple levels of decoding": the instruction_decode(…) function sets the initial ALUOp value of 7 and the

ALU_operations(…) function should update the ALUOp value based on the funct parameter.

Q. Should we halt in rw_memory() in the event that ((ALUresult % 4)!=0)?

A. Yes, but **only if** ALUresult represents an address. ALUresult should be an address if MemRead or MemWrite is asserted.

Q. 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.

Q. 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 is primarily used for conditional branching.

Q. 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.

Q. 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[].

In this project spimcore.c handles the command line arguments for and contains the main method. All project files will be tested with the original spimcore file.