Douglas Money

CIS 451

Lab 4: Computer Instruction Types

1. What is the address of the first instruction that will be executed when running the program?

*4194304*

1. Where did you find this information?

*This is in the text segment area of MIPS. There is an area that holds all of the program instructions. The second column is address, which shows where that specific instruction is held in memory.*

1. Which memory locations contain the value for the variable val?

*Val is stored starting at address 268500992 until address 2685001023. This is because when we defined the variable val we gave it a declaration of .word. This is saying use a default word’s worth of space in MIPS, which is 4 bytes, or 32 bits.*

1. Diagram and label a process' address space in Mars. By this, I mean, draw a column labeled 0x00000000 at the bottom and 0xffffffff at the top. Then, show which portions of this address space are used used for instructions, user data, the stack, kernel data, etc. Hint: You will have to look in different MARS windows to find these different areas. Also, the stack grows "down", meaning that each data item added to the stack has a smaller address than the previous item.

|  |  |
| --- | --- |
| 0xFFFFFFFF | memory map limit address & kernel space high address |
| 0xFFFF0000 | MMIO base address |
| 0xFFFEFFFF | kernel data segment limit address |
| 0x90000000 | .kdata base address |
| 0x8FFFFFFC | kernel text limit address |
| 0x80000180 | exception handler address |
| 0x80000000 | kernel space base address & .ktext base address |
| 0x7FFFFFFFF | user space high address & data segment limit address |
| 0x7FFFFFFC | stack base address |
| 0x7FFFEFFC | stack pointer $sp |
| 0x10040000 | stack limit address & heap base address |
| 0x10010000 | .data base address |
| 0x10008000 | global pointer $gp |
| 0x10000000 | .extern base address & data segment base address |
| 0xFFFFFFC | text limit address |
| 0x00400000 | text base address |
| 0x00000000 |  |

1. What is the maximum program size for this configuration? (In other words, how many instructions can your program have before they overflow into data memory?)

*0x10000000 - 0x00400000 = 0xfc00000*

*0xfc00000 / 4 = 0x3F00000 instructions before it overflows into data memory*

1. What lines of machine language does the addi pseudo-instruction on line 11 produce?

***Hex:*** *0x20427fff*

***Binary:*** *0b0010 0000 0100 0010 0111 1111 1111 1111*

1. What does "lui" stand for? (Hint: Look on page A-57 in the SPIM guide.)

*Load upper immediate*

1. Why do the two addi instructions result in different numbers of actual instructions? (Hint: Write each constant out in hex.)

*The second addi instruction is trying to add 65538, which is an overflow outside*

*of the 16 bits available*.

1. Explain how the MIPS assembler applies the "Make the common case fast" principle to addi.

*Most of the time we will be adding integers that are between 0 and 65536,*

*therefore MIPS just adds the containing two registers together without doing any loading. When we get outside those bounds, MIPS must load into 2 separate registers in order to handle the overflow before adding*

1. What is the hex representation of the "immediate" parameter to the lui instruction generated for line 12?

*0x00001001 is the parameter value for the lui instruction.*

1. Why is this the immediate value that is used (i.e., what does it represent)? Hint: Look in the Data Segment window.

*That value in decimal is 4097. This is also the beginning 16 bits of the address*

*where our values are being loaded into.*

1. Now, look at the second machine instruction generated for line 12 (the first lw instruction). Notice that this instruction has three parameters. Describe the function of all three parameters. (Hint: Examine the machine language generated for lines 13 and 15 as well as the description of lw on page A-67.)

*The lw instruction uses an offset to access the correct part of a given register.*

*The offset is the digit on the outside of the parentheses, with the desired register*

*to access on the inside of the parentheses. The first parameter is the destination register.*

1. Are all three parameters necessary in order for lw to be able to access the entire 4GB memory space; or, could you eliminate the offset parameter? (Imagine a hypothetical lw instruction that did not have an offset parameter. Would there be memory locations that could not be accessed using this hypothetical lw?) If so, give an example. If not, give a sequence of machine instructions that could be used to loadval2 into $t1 with a 2-parameter version of lw.

*All three parameters aren’t necessary. Without all 3 parameters, though, you would need to load the entire 32-bit register value first. For example, something like this could work:*

*Lui $1, 0x00000001*

*Addi $1, $1, 0x0020*

*Lw $8, $1*

1. Explain why the three-parameter version of lw is useful. Include an explanation of how can it be used to "make the common case fast." (In other words, how it can be used to reduce the number of instructions needed by the program.) (Hint: Look for redundant code in the execute window for exampleCIT-3.s.)

*The three parameter version of lw is useful and quicker because it allows condensing of a 3 operation instruction into only 2 instructions*

1. Explain the cost of the three-parameter version of lw. In particular, include an explanation of how the third (i.e., offset) parameter can potentially slow the computer (as compared to the hypothetical two-parameter lw).

*Let’s say you don’t need to load any values from addresses that have any values in the*

*lower 16 bits. Then the 3 parameter version of lw would be doing an unnecessary step of*

*using the offset at* all.

1. How is the li pseudo-instruction implemented? In other words, which "real" instructions are used to implement the li pseudo-instruction? (Remember, register 0 always contains the value 0.)

*addiu $2, $0, 0x00000004*

*This instructions adds 4 (an immediate value) to the contents of the zero register (always*

*0), then puts the result into register 2, which is val1.*

1. How does MIPS implement the move pseudo-instruction?

*MIPS does this: addu $16, $0, $2*

*This instruction again adds the zero register together with register $2, then loads that*

*result into register 16, which is our s0 value.*

1. Would a built-in move be faster than the MIPS implementation? Why or why not? Consider the effects on both the time for the individual instruction, and the overall speed of the processor.

*The move instruction itself would be faster, so it may be beneficial but only for that case. The overall CPU speed would be somewhat hindered because adding the logic for a built-in move call would lengthen the critical path of the CPU.*

***See Attached***

1. Why does the la pseudo-instruction in line 17 generate two assembly instructions while the li pseudo instruction in line 18 generates only one?

*The la instruction is taking and moving a 32 bit register address to another*

*register. We have seen how this takes 2 instructions to complete due to it being a*

*32 bit number. The li instruction, on the other hand, is loading a 16 bit number*

*which only takes 1 instruction to complete because there is no loading of any upper bits*

1. What is the value of the immediate parameter for the beq instruction on line 30?

*0x00000018*

1. Where does this number come from (i.e., how does the assembler calculate it)?

*The branch command address is at 0x00400024, and the address of the exit command*

*comes in at 0x00400088. We can see that these two addresses live 24 address values apart from each other. This translates to 0x18 in hexadecimal. The immediate parameter for the beq instruction is the number of register values that the PC needs to jump to continue.*

1. What is the value of the immediate parameter for the j instruction on line 72? Be careful, you need to look at the actual hex value of the instruction, not the number in the "Basic" column.

*0x100004*

1. Where does this number come from (i.e., how does the assembler calculate it)?

*The value of the immediate parameter jumps to a target whose address is equal to the*

*last 26 bits of Js target value. The value of the immediate parameter is 0x00100004*

*which when binary shifted left 2 bits is equal to the address of line 22 0x00400010*