**Debugging Memory Operations**

In the Class Project we will need to debug memory operations (loads and stores) by observing the data which is actually in memory. This document describes this process, which will occur in the Debug Perspective.

1. **Displaying Memory Contents**

When a program is executing, there will be a tab in the lower Console Window called Memory, as shown in Figure 1. This is the area where the system memory contents may be observed. This tab can contain one or two Monitors. In order to add a Monitor, click the small green Plus Sign, and the Dialog Box shown in Figure 2 will appear. Select “Enter memory space and address” with the radio button. The memory space will typically be “standardname.as\_data”, where standardname is the name of the current project – select this from the pulldown choices. The box to the right of the memory space, which is the starting address, MUST have 0 entered (a blank entry will not work), as shown in Figure 3. Click OK.

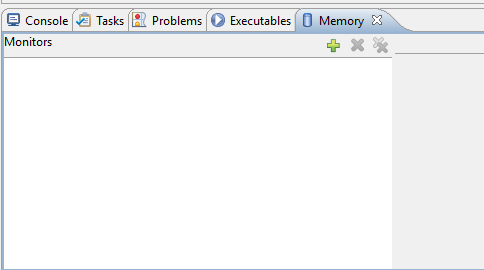


Figure 1

A screenshot of a cell phone

Description automatically generated

Figure 2

A screenshot of a cell phone

Description automatically generated

Figure 3

This will then add a Rendering to the Monitor. Each rendering shows the memory data in a slightly different way, and the initial value is the Codasip Hex Rendering as shown in Figure 4. In this Rendering, the address is shown followed by the 32-bit data (in hex) at that address, that address +1, that address + 2 and that address + 3. Clicking the green Plus sign next to New Renderings will show a number of other possible Renderings. “Traditional” shows eight hex values and may be more useful. Within the Monitor, scrolling moves forward and backward through the address space.

A screenshot of a social media post

Description automatically generated

Figure 4

If the simulation is stopped, the Monitor will go away. If the simulation is run again, the Monitor should be restored. In that case you will have to double click Codasip Hex Rendering (or another desired Rendering) to select it.

1. **Little Endian Memory**

The first test program which uses memory is phase2\_test, so initiate that simulation (this will show the data in Figure 4 above). If the test passes, scroll down until you can see addresses starting at 0x7C0. The highlighted value is in address 0x7E4. Because RISC-V is a Little Endian architecture, the first two digits (the first two nibbles – “12”) are the least significant byte of the 32-bit word at address 0. The next two digits are the second least significant byte, and so on. As a result, the value which was written from the processor has the bytes reversed – 0x78564312.

Note that the actual value is NOT all eight nibbles reversed – the 8-bit bytes are reversed.

1. **Separate Memory Spaces**

RISC-V is a Harvard architecture, which means that there should be separate instruction and data memories, each with their own address space. In Codasip we have two memory spaces defined as “as\_code” (the Instruction Memory) and “as\_data” (the Data Memory). We will not need to look at the Instruction Memory since it doesn’t change, which is why we selected the memory space as\_data in Section 1.

1. **Observing Memory Changes**

The first memory location phase2\_test writes to is in line 554, which is a store word (sw) to a base address of x26 (which has been set to 0) and an immediate offset of 0x7CC to create the address 0x7CC. Scroll down in the Memory Monitor until the line with 7C0 at the start appears. The initial data at that location is 0, which is the initial value of all of data memory.

Run the test to line 554, either by stepping or by setting a breakpoint. At that point the memory value at address 0x7CC is still 0. However, one step changes the value to 0x32000000 and highlights the word to indicate that it was written, as shown in Figure 5. Note that because of the Little Endianness, the actual register value written (from x10) was 0x00000032, which indicates case 50.

A screenshot of a social media post

Description automatically generated

Figure 5

Observing the values in memory is a good way to debug failures with memory operations. On a load, the destination register should be loaded with the value from the selected memory location.