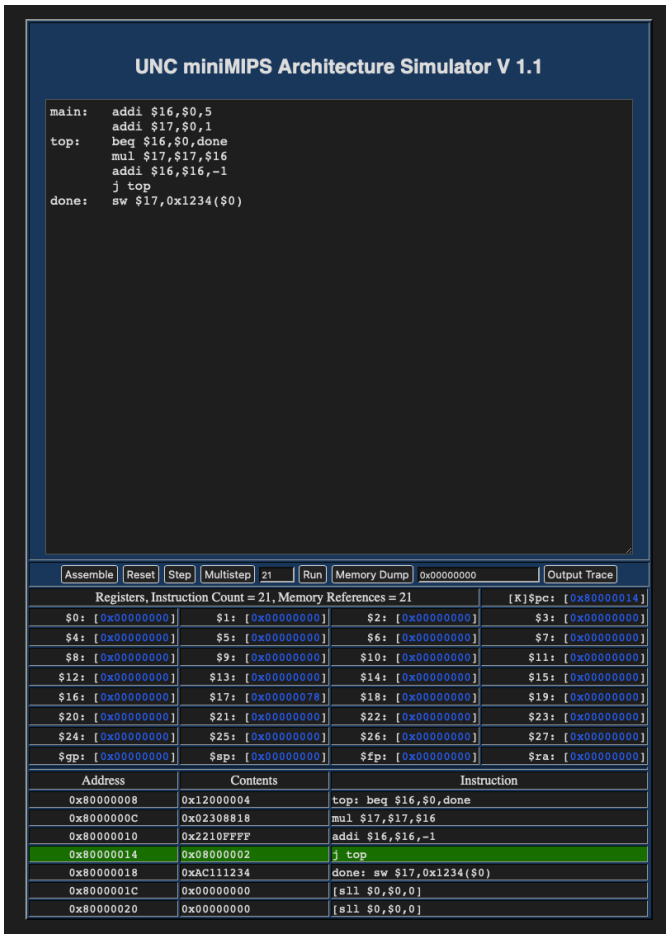
Max Leroy
Soren Marius
10 March, 2024

Lab 3 Assembly

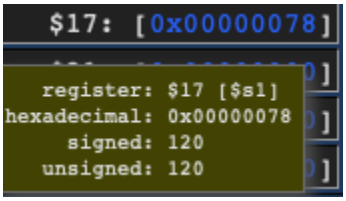
This lab explores MIPS using the UNC MiniMIPS simulator to help us visualize and interact with how low-level instructions execute on a processor. The given assembly program calculates the factorial of 5 by multiplying values in a loop and storing the final result in memory. We get to analyze the assembly code by stepping through the execution, seeing register values change, and observing memory changes. After this, we translated this assembly to a higher-level language, modified a variable in this higher level language, and then updated the assembly code to reflect this change. Below is our documentation of this, along with a few answered questions.

Original Assembly language

Below is the original assembly code after we ran it for 21 steps. As was mentioned earlier, this snippet calculates the factorial of 5 within a loop.



Below is the memory dump of the final register (address 0x1234) after this code ran.



Memory Dump: 0x1234 = 4660

Address	:	Contents	
0x00001234 :	0x00000078120		
0x00001238 :	0x00000000	0	
0x0000123C :	0x00000000	0	
0x00001240 :	0x00000000	0	
0x00001244 :	0x00000000	0	
0x00001248 :	0x00000000	0	
0x0000124C :	0x00000000	0	
0x00001250 :	0x00000000	0	
0x00001254 :	0x00000000	0	
0x00001258 :	0x00000000	0	
0x0000125C :	0x00000000	0	
0x00001260 :	0x00000000	0	
0x00001264 :	0x00000000	0	
0x00001268 :	0x00000000	0	
0x0000126C :	0x00000000	0	

Close

Higher Level Equivalent

Below is the equivalent original assembly in C++. Within the main execution thread, we store the values of 5 and 1 in local variables (similarly to the registers $16 and $17). We then run a loop from 5 down to 1 which we multiply by the current result to calculate the factorial. We finally print out this variable to simulate it being written to memory (like the final sw line in the original assembly).

```cpp
int main() {
    int x = 5;  // Equivalent to $16 = 5
    int result = 1;  // Equivalent to $17 = 1

    while (x > 0) {  // Equivalent to the "top" loop
        result *= x;  // Equivalent to mul $17, $17, $16
        x--;  // Equivalent to addi $16, $16, -1
    }

    // Equivalent to sw $17, 0x1234($0) (storing result in memory)
    std::cout << "Final result: " << result << std::endl;
    return 0;
}
```

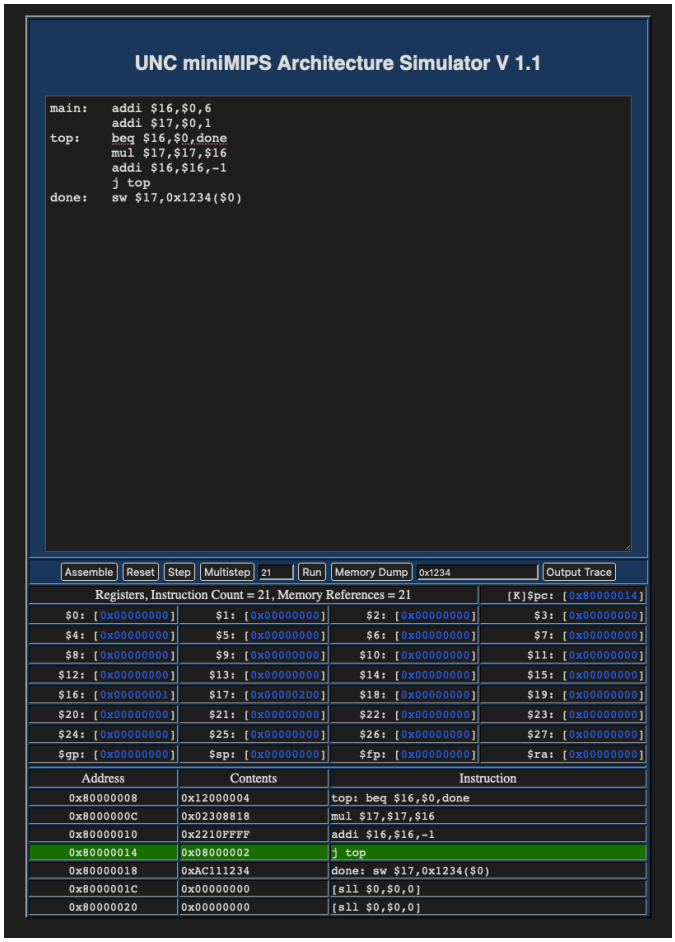
Then, we made a slight change to this high level code by changing the value of the original variable ‘x’ to 6. This means we are now calculating the factorial of 6 instead of 5. Below is the high level code that reflects this change, with a simple change to the declaration of x.

```cpp
int main() {
    int x = 6;  // Changed from 5 to 6 (Equivalent to $16 = 6)
    int result = 1;  // Equivalent to $17 = 1

    while (x > 0) {  // Loop continues until x is 0
        result *= x;  // Equivalent to mul $17, $17, $16
        x--;  // Equivalent to addi $16, $16, -1
    }

    // Equivalent to sw $17, 0x1234($0) (storing result in memory)
    std::cout << "Final result: " << result << std::endl;
    return 0;
}
```

Finally, we changed the original assembly to reflect this change. This was done by simply changing the original value stored in register $16 from 5 to 6. The memory dump below shows the value at the final address 0x1234 to be 720, which equals 6!.



Memory Dump: 0x1234 = 4660

Address	:	Contents	
0x00001234	:	0x000002D0	720
0x00001238	:	0x00000000	0
0x0000123C	:	0x00000000	0
0x00001240	:	0x00000000	0
0x00001244	:	0x00000000	0
0x00001248	:	0x00000000	0
0x0000124C	:	0x00000000	0
0x00001250	:	0x00000000	0
0x00001254	:	0x00000000	0
0x00001258	:	0x00000000	0
0x0000125C	:	0x00000000	0
0x00001260	:	0x00000000	0
0x00001264	:	0x00000000	0
0x00001268	:	0x00000000	0
0x0000126C	:	0x00000000	0

Contributions

Max did most of the lab first and then Soren did it as well afterwards. Max documented the majority of the lab and answered the lab questions. Soren ran through and completed some of the memory dumps that Max wasn't able to get.

Lab Questions

1. **Include answers to questions posed in steps 6 and 8 of the assembly simulation.**
    a. 6. The Program Counter (PC) in MIPS stores the address of the next instruction to be executed. During normal execution, it increments by 4 for each instruction, since MIPS instructions are 4 bytes long. However, when a jump (j top) or branch (beq) instruction is encountered, the PC updates to the target address instead of the next sequential instruction.
    b. 8. The Reset button reloads the instructions into memory but does not reset the contents of memory or registers because MIPS processors do not automatically clear memory or registers upon restarting a program. This behavior mimics real hardware, where memory and register values persist unless explicitly overwritten, allowing debugging and testing without losing previous execution data. It helps track how data changes across runs, making it useful for analyzing program behavior and troubleshooting issues.

2. **In plain English, what is this assembly program doing? You can reference your high-level code to help describe the operation if it is helpful.**
    a. This program calculates the factorial of a given number stored in register $16 (which is initialized to 5 in this case). It does so using a loop that multiplies a running product (stored in $17) by decreasing values stored in $16 until it reaches zero. It then stores the computed factorial in memory at the address 0x1234.

3. **What are the MIPS names of the registers used in this program?**
    a. Three registers are used: $16, $17, and $0. Register $16 corresponds to $s0, which is typically used to store a saved value, and in this case, it holds the number whose factorial is being computed. Register $17 corresponds to $s1, another saved register, which stores the running factorial result throughout the loop. Lastly, $0 corresponds to $zero, a special register that always holds the constant value 0, used here for comparison and memory storage operations.