Michael Winkler

Senior Research

Professor Joel Hollingsworth

Change Log

4/8/17:

* Implemented the quantum counter (referred to from here on as the QC). For now, the quantum value is hardcoded into the chip itself, but I’m not committed to this approach; in the future, the QC could read the value from somewhere else.

4/9/17-4/10/17:

* Began integrating the QC into my CPU. More specifically:
  + Added the QC chip to the CPU
  + Added two or gates that or the output of the QC with the signals that determine with the D register should load a value and whether the program counter should load a value. I’ve included these because when the quantum is reached, I want both the D register and the PC to load a new value.
  + Added a not gate that takes the QC’s output as input and has as output ~qcOut. This is fed into the increment input of the program counter, which determines whether the PC increments during a given clock cycle, because I want it to pause for one clock cycle when the quantum is reached.
  + Added a register that holds
  + Added a 16-bit multiplexor that chooses between the output of the A register and the address at which the code that jumps to the control function begins. That multiplexor feeds into the program counter’s input and is controlled by the output of the quantum counter. The result of this setup is that when the quantum is reached, the program counter loads as its next value the address of the beginning of the code that jumps to the control function.
  + Added another 16-bit multiplexor that chooses between the ALU output and the address output by the program counter. Its output is fed into the D register and it’s controlled by the quantum counter’s output. Before I made any modifications, the D register always took the output of the ALU into its input; now, when the quantum is reached, it stores the current PC value. This is important because the code that jumps to the control function expects the previous PC value to be stored in the D register so that it can be pushed onto the stack as a return address.

4/11/17:

4/13/17:

* Realized that I can use the existing function call method in my compiler to call the program control method, rather than writing the code to call it from scratch. However, the method did require some modifications; to that end, I added a function called writeControllCall() into my codeWriter class. I also modified the writeInit() function to call the new function.

4/15/17:

* Focused on testing today. I’ve been testing each component as I go along, but today I made sure everything works together. Specifically, I ensured that the following steps function correctly:
  1. The compiler completes the jack -> virtual machine language step correctly, including the Sys file which contains my control method.
  2. The compiler also completes the virtual machine language -> assembly language step correctly, including the additions I made that allow the jump to the control function to occur.
  3. When I step through a program running on my modified CPU, an interrupt occurs after the number of steps specified by the quantum and a jump to the control function is affected.
  4. After the control function executes, control is correctly returned to the function that was executing before the interrupt occurred.

4/16/17:

* Implemented the “disk” chip, which is not really a separate disk but will be treated as such for the purposes of this project

4/20/17:

* Began integrating my new “disk” chip into the computer. It turns out that this is a much taller task than I’d anticipated because the computer, as it exists, is 16-bit, which means that its registers aren’t big enough to store all the addresses that exist within the disk. I’ll also need to modify the instruction set to contain a ‘write to disk’ command in addition to the existing ‘write to memory (RAM)’ instruction. I’m going to set this aside until I make some more progress on modifying the compiler to compile multiple programs to be run simultaneously.
* Also, I set up the file/folder structure that I’ll use to test the next set of modifications I’m going to make to the compiler, which will allow it to compile multiple programs to be run in parallel. Due to the way the computer is structured, all the programs to be run simultaneously will have to be compiled together. It’s not ideal, and obviously a real computer wouldn’t do it this way, because users need to be able to start and stop programs at their discretion. But adding that level of functionality would require changing the computer to a degree that it would be beyond the scope of this project.

4/22/17:

* Modified the compiler to compile multiple files to be run in parallel. This involved:
  + Changing the compiler to expect a certain file structure (one folder containing subfolders that each represent an individual program and hold the .vm files for that program)
  + Modified the compiler to keep track of which program it’s currently writing. The compiler uses this information in the follow ways:
    - Any time the compiler writes the code for a function, it adds a modifier specific to the current program to the function’s identifier so that if multiple programs have functions with the same name, they aren’t all translated to the same location in the ROM.
    - Also, the compiler sometimes uses file names when creating labels for code segments. To avoid conflicts if multiple programs have files with the same name (e.g. each program must have a main.vm file) the compiler also adds a program-specific modifier to labels created using file names.
  + Designated R17 to store the number of programs currently being executed, and modified the compiler to store the number of programs it compiles to be run simultaneously in R17 in its writeInit() method

4/23/17-4/26/17:

* Modified the compiler to store the starting address of each program (on disk) in the computer’s RAM starting at address 22 (the first address after the designated addresses)
* Integrated the disk chip into the computer. This involved making the following modifications:
  + I had to modify the processor’s instruction set to add the ability to do anything with the disk that it can do with RAM memory. So for example, since the instruction D=M exists in the processor’s assembly language (it means: write into the D register the value in RAM pointed to by the value in the A register) D=E should also exist. I should note that since the letter D is already used in the processor’s assembly instruction set to represent the D register, I chose E to represent the disk. I will discuss the changes I made to the instruction set in greater detail in a different file
  + Added an ‘inE’ (input from disk) input to the CPU in addition to its ‘inM’ (input from RAM) input
  + Added an ‘outE’ (to disk) output to the CPU
  + Added a ‘writeE’ output from the CPU that indicates whether the CPU’s outE output should be written to the disk
  + I also modified the processor to actually affect the new instructions involving the disk. This meant giving it the ability to work with addresses that are longer than 15 bits, because the disk has more addresses than can be encoded in 15 bits. The problem was that the A register, which holds values that are treated as addresses in memory, was only 16 bits, which was still too short to handle the disk addresses. In order to deal with this issue (and similar issues that cropped up when I started attempting to deal with this one), I refactored the whole computer to be 32-bit rather than 16-bit. That meant refactoring the following components:
    - And16 -> And32
    - Mux16 -> Mux32
    - Mux4Way16 -> Mux4Way32
    - Mux8Way16 -> Mux8Way32
    - Not16 -> Not32
    - Or16 -> Or32
    - Add16 -> Add32
    - Inc16 -> Inc32
    - ALU -> ALU32Bit
    - Register -> Register32Bit
    - RAM8 -> RAM832Bit\*
    - RAM64 -> RAM6432Bit\*
    - RAM512 -> RAM51232Bit\*
    - RAM4K -> RAM4K32Bit\*
    - RAM16K -> RAM16K32Bit\*
    - RAM32K -> RAM32K32Bit\*
    - RAM256K -> RAM32K32Bit\*
    - ROM32K -> ROM32K32Bit\*
    - Disk -> Disk32Bit
    - PC -> PC32Bit
    - QC -> QC32Bit
    - Screen -> Screen32Bit
    - Keyboard -> Keyboard32Bit
    - CPU -> CPU32Bit
    - Computer -> Computer32Bit
    - \*The unfortunate naming convention here was mandated by the fact that chip names, according to the Nand2tetris testing framework, can neither start with numbers nor contain special characters
  + In addition, I had to restructure the inputs to ALU within the processor so that it could accept instructions involving the disk. The ALU itself is unmodified; it still takes two 32-bit inputs plus 6 instruction bits. However, the input to and output from the ALU now differ based on two instruction bits that were previously unused: one to specify whether the result of an operation should be written to the disk, and one to specify whether the operation itself should be executed on the value currently pointed to on the disk, instead of the value pointed to in the RAM.
  + Finally, I had to add the ability to recognize instructions involving the disk to the assembler

To-do:

* The actual code for process switching that gets called when the interrupt occurs
* Virtual memory