Carleton University

Department of Systems and Computer Engineering

SYSC 3006

Computer Organization

Debugger User Guide

Debugger: V4.5

Guide Version: 4.3

The Debugger circuit is distributed in a zipped folder. **You MUST UNZIP the folder** (i.e. extract all files) before trying to load these circuit files into Logisim. If the files are not unzipped, they may load into Logisim, but may not simulate properly.

Introduction

The distributed Debugger circuit includes a working implementation of a computer system (Processor, Memory and I/O) as shown in Figure 1.

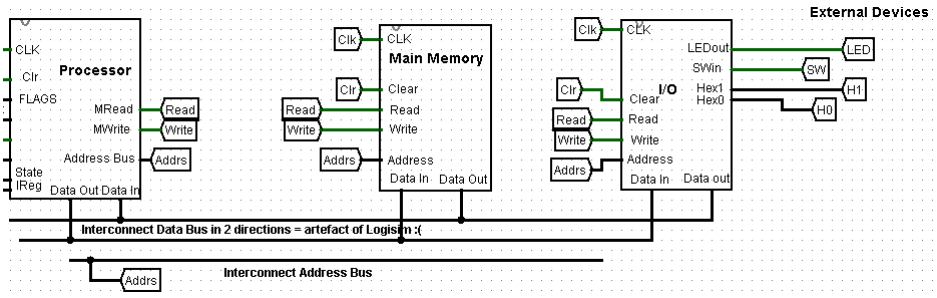

Figure 1 Computer System Circuit

The **Processor** supports all of the software instructions discussed in class; however, the underlying hardware may have a few minor differences from that discussed in class. All of these hardware differences are irrelevant to the software behaviour, and therefore, are of no concern when working on the software side of the hardware/software interface. When working on the software side of the hardware/software interface there is no need to look at the Processor's internal circuits.

The **Main Memory** is a 4 k x 32-word memory. Loading programs into Main Memory will require accessing the memory hardware (and will be discussed below).

The **I/O** component will be used in later labs and can be ignored until it is needed.

The **Interconnection Bus** accommodates a Logisim "quirk". The control signals (Clk, Read and Write) and Interconnection Address Bus are as expected from class, but the data bus is different. Logisim does not allow the "pins" on components to support bi-directional transfer (i.e. a pin cannot support transferring a signal both into and out of the component). This complicates the use of pins to interface the Data Bus,

since the direction of transfer depends on whether the bus is being used to Read or Write. The solution in the Debugger circuit introduces a separate Interconnection Data Bus for transfer in each direction (e.g. one for data going into the Processor and one for data going out of the Processor). Having these two data buses works around the pin problem and the resulting Interconnection Bus circuit performs as expected.

The distributed Debugger circuit also includes an integrated Debug Control Panel to control software execution. The control panel is shown in Figure 2, and discussed further below.

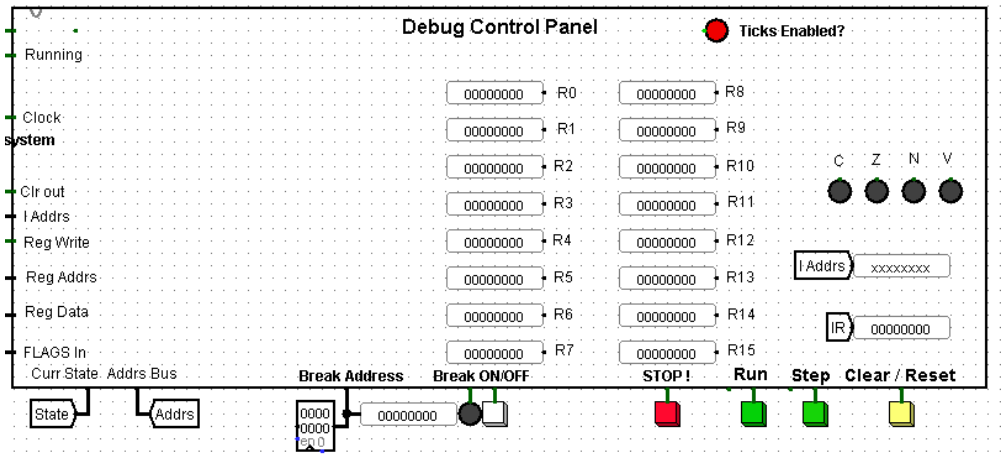

Figure 2 Debug Control Panel

A program's execution state in the Processor (sometimes called the program's execution *context*) consists of the register values and FLAG values. The current values in the Processor's registers are shown in the probes labeled R0 through R15. The current values of the ALU FLAGS are displayed using the LEDs labeled C, Z, N, and V. The gray values shown in the figure indicated the current values of the FLAGS are 0. When a flag is 1, its corresponding LED will be red. When the circuit is initially loaded or reset (discussed below) the register values and ALU FLAGS are all 0 (as shown in the figure).

The panel uses IAddrs and IR to show the current instruction. The current instruction is the instruction that has been fetched by the current iteration of the instruction cycle. The control panel allows the instruction cycle to be stopped following the fetching of an instruction (actually the cycle is paused until it is resumed by pressing the Step or Run buttons, discussed below). When execution has been stopped, IAddrs shows the address that the current instruction was fetched from, and IR shows the fetched instruction. The IAddrs value of all x's (shown in the figure) indicates that the circuit has been initially loaded but an instruction has not yet been fetched. The initial value of 0x00000000 in IR is the value in IR when the Processor is reset (discussed below).

The panel has 5 buttons to help control software execution. The yellow Clear/Reset button resets the Processor, but has no effect on the Main Memory. Resetting the Processor forces all register values and the FLAGS to 0. If the ticking clock is enabled (discussed below), pressing the Clear/Reset button will also cause the first instruction to be fetched, but not executed. The green Step button will execute the current instruction only. After executing that instruction the next instruction will be fetched (to become the current instruction) but will not be executed. The green RUN button will execute the current

instruction and then continue fetching and executing instructions until some future stopping condition is encountered. The red STOP button will stop the execution of instructions. The STOP button is included to regain control should the execution of your program "run away" in an unexpected manner (for example, when an infinite loop is entered). The white Break ON/OFF switch allows a breakpoint address to be set (breakpoints are discussed later).

Note: Logisim's Poke tool must be used to press the buttons during simulation.

Starting and Initializing the Debugger

When the Debugger circuit is loaded, the canvas will appear as shown in Figure 3, which has been (mostly) shown before in Figure 1 and Figure 2.

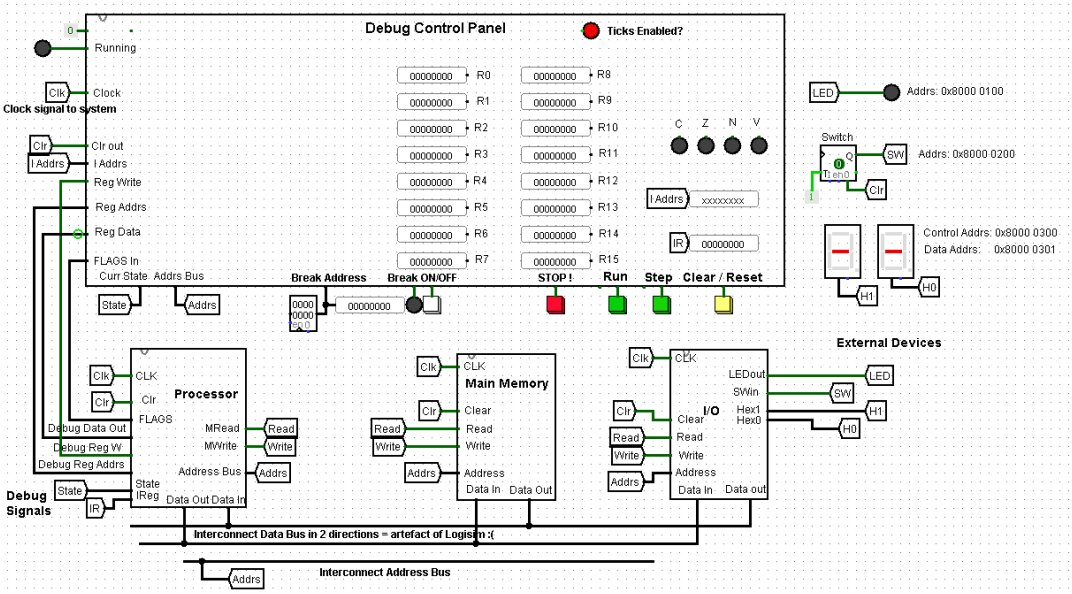

Figure 3 Debugger Circuit Loaded in Logisim

The circuit has an internal ticking clock that drives the system. Logisim loads the circuit with the ticking clock disabled. **NOTE: The ticking clock MUST be enabled to cause any activity in the circuit.** The current status of the ticking clock can be seen and controlled using the Simulate → Ticks Enabled menu item. Figure 4 shows the initial Simulate menu when the ticking clock is disabled.

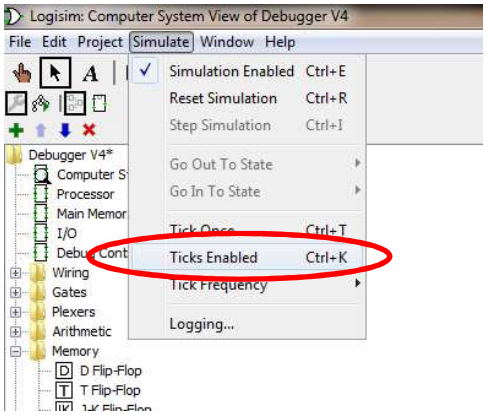

Figure 4 Ticking Clock Disabled

The Debugger circuit can initially sense that the ticking clock is disabled and uses the Ticks Enabled? LED (see Figure 5) as a reminder that the clock has yet to be enabled.

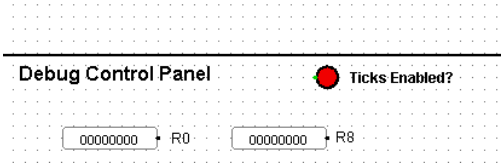

Figure 5 Ticks Disabled Reminder

The clock can be enabled using Simulate → Ticks Enable (i.e. pull down the Simulate menu and click on Ticks Enable). As seen in Figure 6, a check mark appears to the left of "Ticks Enabled" in the menu when the clock is enabled.

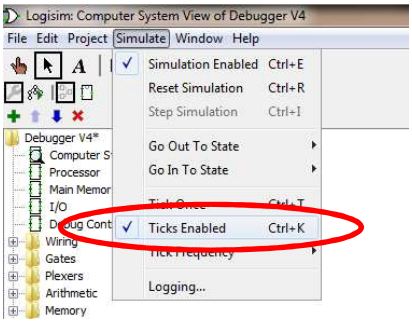

Figure 6 Ticking Clock Enabled

At this point, the Debugger is ready to execute instructions; however, none have been loaded in Main Memory. When Main Memory initially loads, each word contains 0x00000000 (which happens to correspond to NOP instructions).

Executing Instructions and Programs

The Debugger circuit has been distributed with a Test Program to allow initial validation that the circuit is working. The assembly language source code is in the file TestProgram SRC. The assembled code is in the file TestProgram OBJ, and is compatible with loading as a memory image into a Logisim Memory component.

This is the assembly language source code for the Test Program:

```
        B Go                ; unconditional Branch over the Minus1 variable
Minus1 DCD #-1              ; a variable named Minus1 and containing the value "-1"
Go                          ; the target label declaration for the B instruction
        LDR R0, [ Minus1 ]  ; R0 = -1
        DCD #0xFFFFFFFF     ; breakpoint instruction is encoded as 0xFFFFFFFF
```

The test program loads R0 with -1 and then executes a breakpoint (breakpoints are discussed later).

To load the assembled version of the code into Main Memory, right-click on the Main Memory component and select View Main Memory as seen in Figure 7. **Note:** Care must be taken in how the

Main Memory component is accessed for loading. Other approaches may open a similar view of Main Memory, but may not bring the loaded image into the simulation.

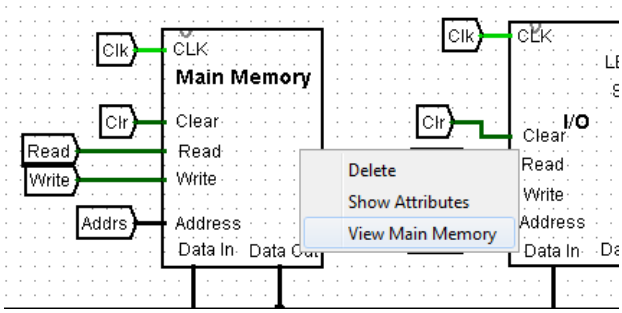

Figure 7 Right-Click on Main Memory Component

The Main Memory component view will then open on the canvas, as shown in Figure 8.

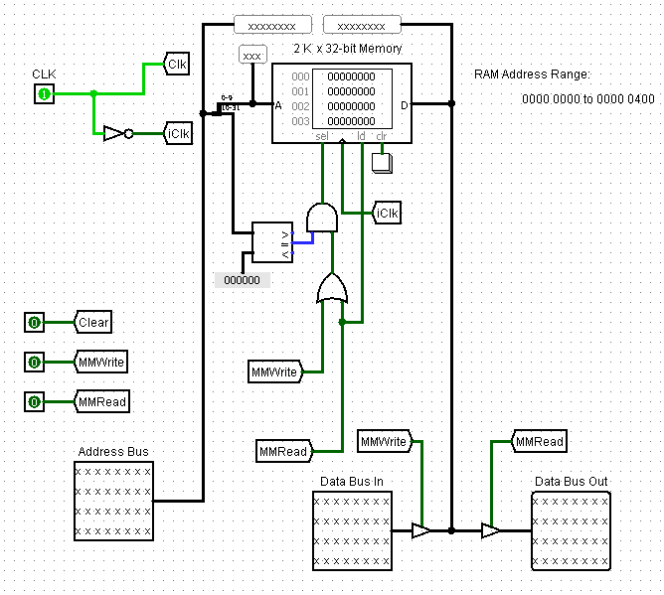

Figure 8 Main Memory Component Circuit

Right-click on the RAM component and then select Load Image, as seen in Figure 9. Navigate to the folder containing the Test Program and select TestProgram OBJ.

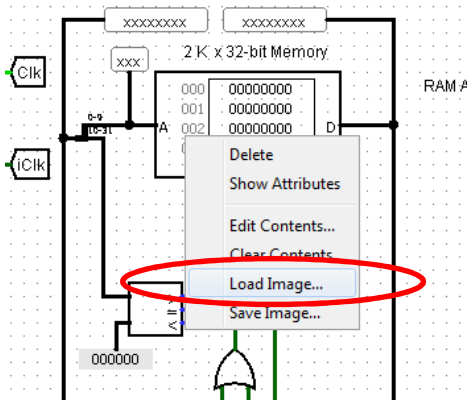

Figure 9 Right-Click on RAM Component

After loading, the Main Memory will contain the program, as shown in Figure 10.

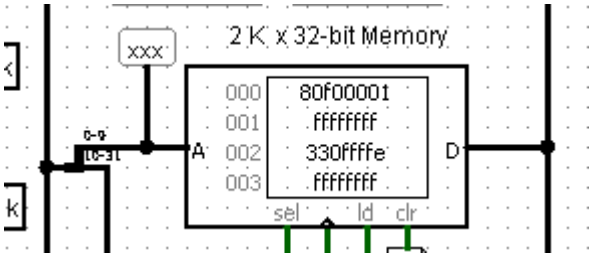

Figure 10 Test Program Loaded in Main Memory

Now that the program has been loaded, return to the Computer System Level by double-clicking on the Computer System View in the Explorer Pane (see Figure 11).

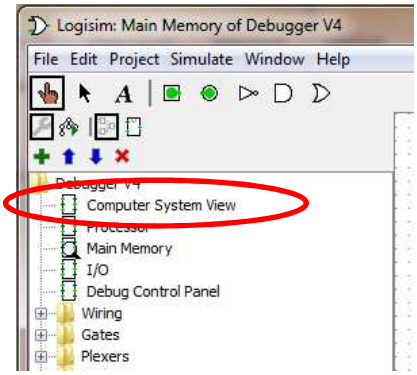

Figure 11 Computer System View in Explorer Pane

Now that the program has been loaded, the Computer System must be reset to start debugging. Press the yellow Clear/Rest button to reset the system. The reset should result in the first instruction being fetched, but not yet executed. Figure 12 shows the relevant parts of the Debug Control Panel. Note that the IR now contains the instruction stored at Main Memory address 0, i.e. 0x80F00001. This is the first instruction of the Test Program (i.e. B Go), and it is now the *current instruction*. Recall that the PC (R15) is incremented while fetching an instruction. The panel shows the updated (incremented) value, in this

case 0x00000001. The instruction cycle dealing with the first instruction has progressed through the fetch stage and the current instruction is ready for execution.

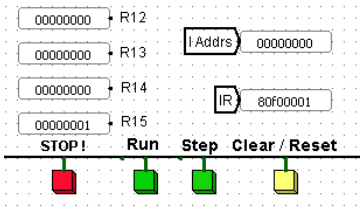

Figure 12 After Loading Test Program and Pressing Reset

There are two ways to continue executing the program. Instructions can be executed one-at-a-time by pressing the green Step button to advance one instruction. Pressing the Step button will execute the current instruction and fetch the next instruction. (Don't forget that Logisim's Poke tool must be used to press the buttons during simulation.) Executing one instruction at a time using the Step button is called *single-stepping*. For example, Figure 13 shows the resulting panel after single-stepping the first instruction. The first instruction is a Branch (unconditional) instruction to address 0x00000002. The panel shows that the execution has advanced the PC to 0x2, then fetched the next instruction (0x330FFFFE) and incremented the PC (R15 now contains 0x3).

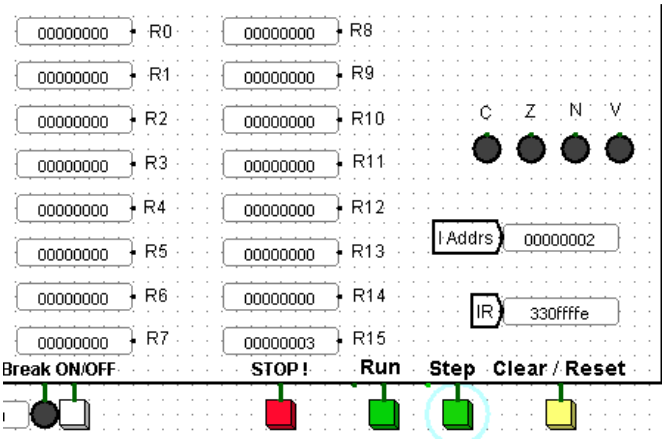

Figure 13 After Single-Stepping 1st Instruction

The green Run button can be used as an alternative to single-stepping. Pressing the Run button causes program execution to continue from the current instruction. **Note:** After pressing Run, execution will not stop unless a breakpoint is encountered (breakpoints are discussed later).

Figure 14 shows the Debug Control Panel that results from pressing the Run button to execute the Test Program. Recall that the test program set R0 = -1, which is confirmed in the figure (R0 = 0xFFFFFFFF which is the 2's complement encoding for -1). The breakpoint instruction (0xFFFFFFFF) has been loaded as the current instruction, and execution has stopped.

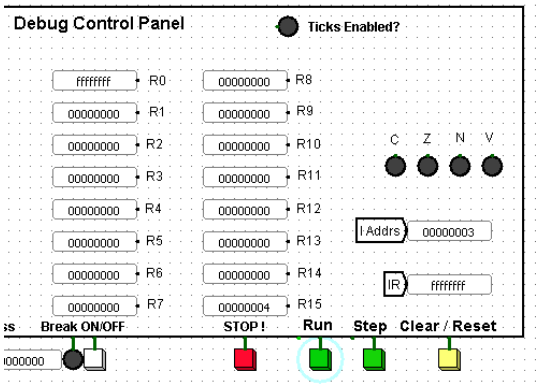

Figure 14 Test Program Execution After Pressing Run

The Debugger circuit supports two forms of breakpoints: a breakpoint instruction and a breakpoint address. The **breakpoint instruction** is a special instruction (encoded with opcode = 0xFF) that is recognized by the Debugger circuit. Executing the breakpoint instruction terminates all system activity and puts the system in a "halted" state. **Note:** After executing a breakpoint instruction, the system must be Reset before it will execute instructions again. The Test Program uses the breakpoint instruction to terminate execution.

The breakpoint address can be used to stop (but not terminate) execution once an instruction is fetched from a particular address. To set a breakpoint address, the Poke tool must be used to program the Break Address Register. By itself, entering a Break Address is not enough to cause a break in execution (should an instruction ever be fetched from that address. To enable the Break Address functionality, the white Break ON/OFF button must be used to toggle the breakpoint address monitoring ON (or OFF). When the Break Address functionality is enabled, the associated LED is ON. Figure 15 shows the Break Address functionality enabled for address 0x0000000A. Now, if an instruction is fetched from address 0xA, then execution will stop before the instruction is executed.

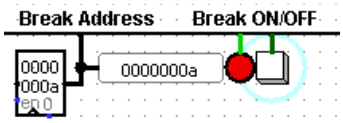

Figure 15 Break Address Set for Address 0xA

Debugging using provided capabilities is an acquired art. Single-stepping is a good way to watch how a program is functioning one instruction at a time. This is essential for debugging programs. However, single-stepping can be tedious when a lot of instructions must execute before an "interesting" part of the program is reached (i.e. a part that needs debugging). A Break Address is a good way to execute to an "interesting" point in a program, and then stop so that single-stepping can be used to debug what happens after that point. Breakpoint instructions are an easy way to ensure that a program stops (terminates) when it has completed. Remember: the instruction cycle will repeat "forever" unless something is done to stop it.

Note: It should not be necessary to use Simulate → Reset Simulation, but if you do remember that is will also clear the contents of the Main Memory RAM component! I.E. After doing this level of Reset, the program must be reloaded.