

CPU Sim

USER'S MANUAL

Version 3.4
January, 2007

by
Dale Skrien

assisted by Raymond Mazza III, Joshua Ladieu, Jonathan Weinberg, Andreea Olea, Thomas Cook, Patrick Rodjito, Tim Monahan, Mike Liedtke, Peter Landwehr, Taylor Snook, Charlie White, Brian Putnam, Tom Goth, Mike Wolk and Cadran Cowansage

Preface

This manual describes how to use version 3.4 of the CPU Sim simulation system. This simulator is based on the ideas in the STARTLE simulator developed by J.M. Kerridge of the Department of Computer Science, University of Sheffield, Sheffield, England. Report any problems or suggestions to Dale Skrien, Department of Computer Science, 5851 Mayflower Hill, Colby College, Waterville, Maine 04901, phone 207-859-5851. (Email: djskrien@colby.edu) (Web page: www.cs.colby.edu/~djskrien/CPUsim/)

Acknowledgments

The author and programmers would like to acknowledge the following people whose code was used in this package:

- Mark Fisher for his JavaHelp demo code¹,

¹ "JavaHelp for the Java Programmer" by Mark Fisher, JavaReport, June, 1999.

- David Flanagan for his PageableText class²,
- Mark Johnson for his HtmlEncoder and LAX classes³,
- Philip Milne for his TableSorter and TableMap classes⁴,
- Matthew Robinson and Pavel Vorobiev for their FindDialog, DialogLayout and PrintPreview classes⁵,
- Robb Shecter for his OOTableModel class⁶,
- Klaus Berg for his EFileChooser package⁷,
- Tom Tessier for his JScrollableDesktopPane package⁸,
- Slava Pestov for his JEditTextArea package⁹,
- Benjamin Michotte for his Text utility class¹⁰
- The ACM Java Task Force group for the IOConsole class¹¹.

INTRODUCTION

CPU Sim is an interactive computer simulation package in which the user specifies the details of the CPU to be simulated, including the register set, memory, the microinstruction set, machine instruction set, and assembly language instructions. Users can write machine or assembly language programs and run them on the CPU they've created.

CPU Sim simulates computer architectures at the register-transfer level. That is, the basic hardware units from which a hypothetical CPU is constructed consist of registers, condition bits, and memory (RAM). The user does not need to deal with individual transistors or gates on the digital logic level of a machine. The basic units used to define machine instructions consist of microinstructions of a variety of types. The details of how the microinstructions get executed by the hardware are not important at this level.

A user of CPU Sim who wants to simulate a new hypothetical CPU must first specify the hardware units and microinstructions for the CPU and then create the machine instruction set. Each machine instruction is implemented via an associated sequence of microinstructions specified by the user.

Once the instruction set has been specified, the user can write programs for the

² *Java Foundation Classes in a Nutshell* by David Flanagan, copyright (c) 1999 by O'Reilly & Associates.

³ "Programming XML in Java" by Mark Johnson, JavaWorld, April, 2000.

⁴ "The Java Tutorial" at <http://java.sun.com/docs/books/tutorial>.

⁵ *Swing* by Matthew Robinson and Pavel Vorobiev, Manning Publications Company, 1999.

⁶ "Object-Oriented Widgets" by Robb Shecter, JavaReport, December, 1999

⁷ <http://www.javaworld.com/javaworld/jvatips/jw-jvatip100.html>

⁸ http://www.javaworld.com/javaworld/jw-11-2001/jw-1130-jscroll_p.html

⁹ <http://www.jedit.org/>

¹⁰ <http://www.usixml.org/javadocs/michotte/be/michotte/util/Text.html>

¹¹ <http://jtf.acm.org/>

¹³ www.cs.colby.edu/~djskrien/CPUSim

new CPU, in either machine language or assembly language, which can be run in the CPU Sim environment.

CPU Sim executes a program through repeated execution of machine cycles. One cycle has two parts: the fetch sequence, which is a sequence of microinstructions that typically loads the next instruction to be executed into an instruction register and decodes it, and the execute sequence, which is the sequence of microinstructions associated with the newly-decoded instruction. The fetch sequence is the same for all cycles, but the execute sequence can, and usually does, vary from cycle to cycle. Execution of machine cycles halts when a “halt” bit is set to 1.

CPU Sim has many useful features for running programs and debugging them. For example, CPU Sim has a built-in text editor and assembler for creating and assembling programs. The editor will highlight the offending section of the program when the assembler finds an error. Assembled programs can be executed without stopping or they can be executed in debugging mode, which allows the users to step through the execution of a program, one machine instruction or microinstruction at a time. After each step, users can edit the contents of any register or memory and then continue stepping or even back up one microinstruction or machine instruction at a time, all the way back to the initial state of the machine.

The ideas for this simulator came from the “STARTLE” simulator developed by J.M. Kerridge of the University of Sheffield.

STARTING CPU Sim

CPU Sim is a Java application and so requires that you have a Java runtime environment (JRE) installed on your computer. CPU Sim should run with JRE 1.5 or later.

After having successfully downloaded and unzipped CPU Sim from the CPU Sim web page¹³, you can start up the program using a command line or terminal window, as described in the “InstallationInstructions.txt” file that is included in the download:

- Open a command or terminal window and navigate to the "CPUSim3.4.X" folder.
- Type in one of the following commands (all on one line):
 - (Windows users)


```
java -cp CPUSim3.4.jar;jhall.jar;CPUSimHelp3.4.jar
      cpusim.Main
```
 - (Mac or Linux users)


```
java -cp CPUSim3.4.jar:jhall.jar:CPUSimHelp3.4.jar
      cpusim.Main
```
- You can also add four optional arguments at the end of the command line in any order:


```
-m <machine file name>
-p <properties file name>
-t <text (assembly program) file name>
-c
```

If you specify a machine file using the `-m` flag, then that machine is loaded into CPU Sim during startup. If you specify a text file using the `-t` flag, then that file is opened during startup. If you specify the `-c` flag, which can only be used together with the `-t` and `-m` flags, then the corresponding text file and machine file will be loaded and run from the command line. If you specify a properties file using the `-p` flag, then the properties in that file are loaded into CPU Sim during startup. A properties file contains information regarding the preferred fonts for text, register, and RAM windows and a list of the recently opened machines and text files.

For example, to start up the Wombat1 machine and load the W1-0.a assembly language program that appear in the SampleAssignments folder, you could type in the following command (all on one line) on Windows:

```
java -cp CPUSim3.4.jar;jhall.jar;CPUSimHelp3.4.jar cpusim.Main -m
SampleAssignments/Wombat1.cpu -t SampleAssignments/W1-0.a
```

Notes:

- If the user does not specify a machine file, a new empty machine is opened.
- CPU Sim searches its directory for any files you specify on the command line, so you must include the directories relative to that location.
- If you use the `-c` flag, you must also specify a machine and text file using the `-m` and `-t` flags. In that case, the CPU Sim GUI windows never appear. Any input or output channels in the machine normally directed to a console window or to a dialog are redirected to the command line. Input and output channels directed to a file are not redirected to the command line.

As always, users can create short cuts to avoid typing such a long command. For example, Linux users can put the above command in a script file. Windows users can create a .bat file containing the command above that they can double-click to start up CPU Sim. Such a file, named "CPUSim.bat", is including in the CPU Sim installation package. Macintosh users can create AppleScript documents that do the same thing.

RUNNING PROGRAMS IN CPU SIM, A Tour using the Wombat1

This section demonstrates how to use CPU Sim to run a program on a hypothetical machine. In this tour, we will (a) load into CPU Sim the hypothetical machine "Wombat1" that has already been defined and saved in a file, (b) open an assembly language program for the Wombat1, (c) assemble the program, (d) load it into the Wombat1's memory, and (e) run it.

To begin, start up CPU Sim without any of the four optional flags mentioned in the preceding section. The window that will appear is the main "desktop" window for CPU Sim. Except for some dialog boxes, all windows used by CPU Sim are windows internal to the desktop window. Such windows include text windows for writing and editing assembly programs and display windows for viewing and editing the contents of registers and memory.

If you do not specify a machine on the command line when starting CPU Sim,

then the title of the main display window is “New” since, if no machine is specified, a new virtual machine is created that has no registers and no memory.

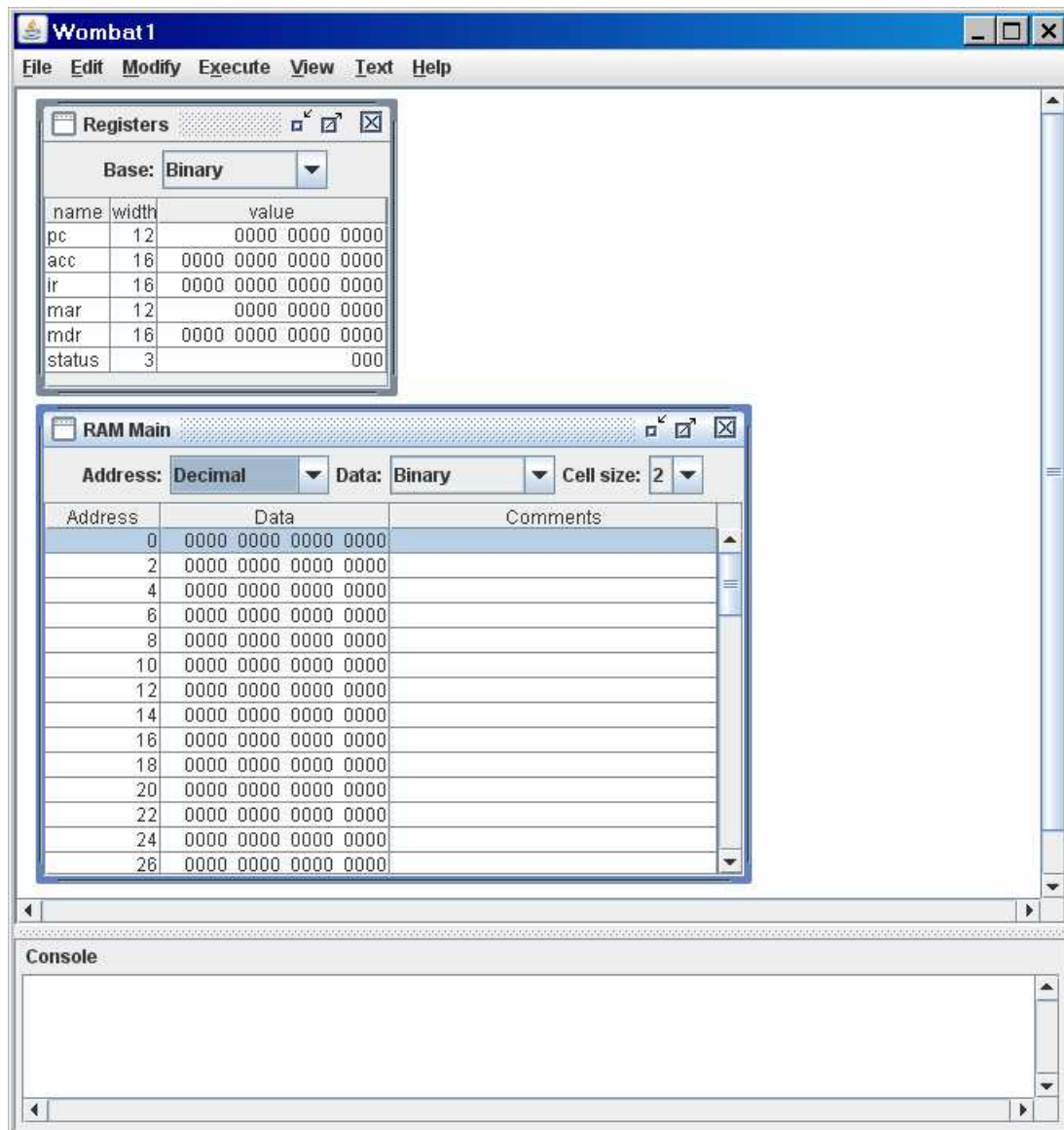


Figure 1. The main display after the Wombat1 has been loaded.

To load a previously saved machine, choose “Open machine...” from the **File** menu. You will be presented with a dialog box in which you are supposed to select a file. To continue with this tutorial, select and open the text file named “Wombat1.cpu”. It should be found in the SampleAssignments folder included with CPU Sim. (Note: The Wombat1.cpu file is a text file in XML format that contains the information for the Wombat1 machine.) If the machine loads without error, then the title of the main display should become “Wombat1” and two windows should appear inside the desktop (see Figure 1).

The window labeled “Registers” displays all the registers of the Wombat1

including their widths and current values. To see the values in every register displayed in binary (base 2), signed 2's complement decimal (base 10), unsigned decimal (base 10), hexadecimal (base 16), or as characters (ASCII or Unicode), select the appropriate item from the popup menu at the top of the window. Values in binary and hexadecimal format are displayed in four digit groups for readability. The values of the registers can be edited in any of these display formats by clicking in the appropriate cell of the table, typing in a new value, and then pressing the enter or return key. The edited values will be automatically reformatted into groups of four characters if the user is working in binary or hexadecimal.

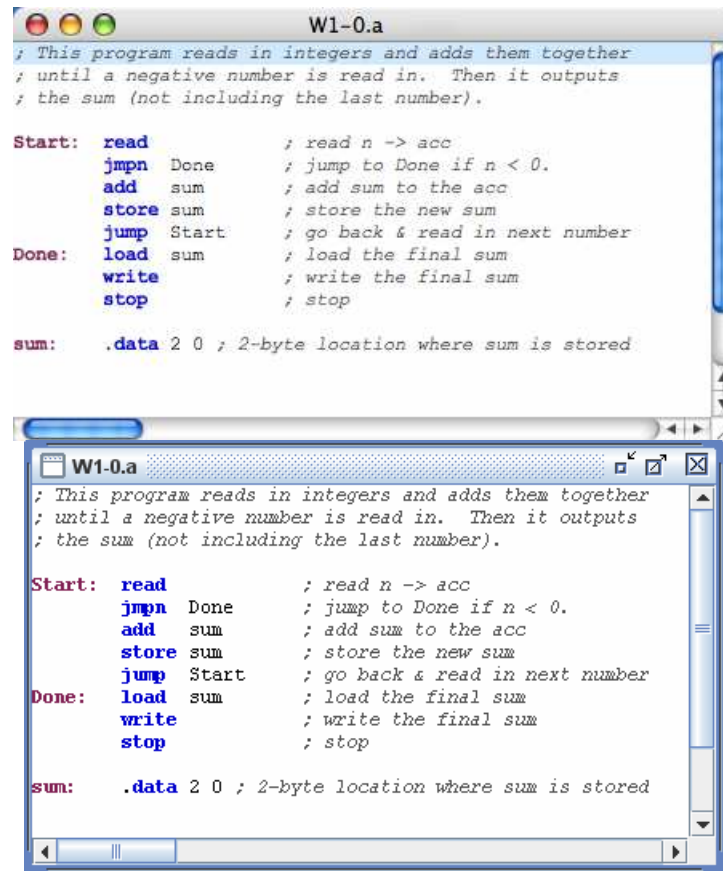
The window labeled "RAM Main" shows the contents of the RAM named "Main". As with the registers, you can display the values in base 2, 10 (signed or unsigned), 16, or in ASCII or Unicode, and, as with the registers, values in binary and hexadecimal format are grouped in four digit units. You can also view the addresses as positive integers in base 2, 10, or 16. You can view the contents of RAM in units of 1-8 bytes instead of just 1 byte. Since the Wombat1 uses 16-bit instructions, it is helpful to view RAM in 16-bit units, so we have selected a cell size of 2 in the popup menu in the RAM window. The values of all cells are editable except the cells in the address column.

Leave the registers and RAM windows open for the remainder of this tour.

Note: The table columns in any of these windows can be reordered by dragging the column headers left or right to a new position. The column widths can also be adjusted by dragging the line dividing the columns. In all windows except the RAM windows, the rows can be resorted by column by clicking in any column header.

The bottom panel of the main display is labeled "Console" and is used for console input and output. That is, the user can type data into the Console panel when prompted for input and CPU Sim can type data in the panel for output.

Now let's open a Wombat1 assembly language program. To do so, choose "Open text..." from the **File** menu and choose the file "W1-0.a". This file may be found in the same folder as the Wombat1.cpu file. After you have selected it, a window will appear containing the text of the file (see Figure 2). The first three lines are comments (comments begin with a semicolon and are shown in italics font in Figure 2). The remaining lines consist of instructions followed by comments. The instructions consist of the name of a Wombat1 machine instruction ("load", "store", etc., written in a bold blue font). Some of the instructions are followed by an argument and some of them have a label in front (labels end with a colon and are shown in a bold burgundy font). The ".data" instruction is actually a pseudo-instruction used by the assembler to allocate memory for the variable "sum". The assembler syntax is described in more detail below in the section entitled "SPECIFICATIONS OF ASSEMBLY LANGUAGE FOR CPU SIM MACHINES."



```

; This program reads in integers and adds them together
; until a negative number is read in. Then it outputs
; the sum (not including the last number).

Start: read          ; read n -> acc
      jmpn Done      ; jump to Done if n < 0.
      add  sum        ; add sum to the acc
      store sum       ; store the new sum
      jump Start     ; go back & read in next number
Done:  load sum        ; load the final sum
      write          ; write the final sum
      stop           ; stop

sum:   .data 2 0 ; 2-byte location where sum is stored

```

Figure 2. A Wombat1 assembly language program.

Before you can run this program, you must assemble it into machine language instructions that the Wombat1 can understand and then load those instructions into the Wombat1's memory. To do so, choose "Assemble & Load" from the **Execute** menu. (Note: If this menu item is disabled, the assembly language text window W1-0.a is not the currently-selected, i.e., highlighted, window. In that case, click on the text window before choosing "Assemble & Load".) You should see numbers and comments appear in the first few rows of the table in the RAM window. The numbers in the data column of the RAM window are the machine language instructions generated by the assembler from the assembly language program. The comments column of the RAM window displays the lines of assembly code from which the corresponding machine instruction was generated.

Now the program in the main memory is ready to be run. Make sure all the registers have been cleared (set to 0). If some of them are not 0, then either edit the values to make them 0 or choose "Clear all registers & arrays" from the **Execute** menu. Then choose "Run" from the **Execute** menu. The program will begin execution with the instruction whose address (namely, 0) is stored in the program counter *pc*. The machine runs by repeatedly executing machine cycles consisting of the fetch sequence, which loads into the *ir* the instruction whose address is in the *pc* and then decodes the instruction, followed by the execute sequence, which executes the machine instruction

that was just decoded. At this point, the console panel should ask you for input. Type a positive integer into the console panel and press the return key or enter key. Notice that the program will halt execution until an input value has been typed into the console. Repeat this process several times and then type in a negative number. The program will display an output message in the console panel giving the sum of all the positive numbers you typed in. This will be followed by a second output message in a new dialog box indicating that the program has ceased execution because a condition bit was set to 1. When execution is complete, you can see the final state of the registers and RAM in the registers and RAM windows.

If you wish to rerun the program with different input, you can proceed in two ways. You can either perform a two step process of selecting “Clear everything” from the **Execute** menu, and then choose “Assemble, load, & run” from the **Execute** menu, or you can select “Clear, assemble, load & run” from the **Execute** menu. While both of these methods perform the same function, “Clear everything” and “Assemble, load, & run” are divided into two separate operations in case you want to debug your program, as explained below.

When you are debugging assembly code, it is useful to be able to step through the execution, one instruction or microinstruction at a time, and to set break points in the code. To practice debugging using the Wombat1 and the program W1-0.a, first select “Clear everything” from the **Execute** menu, and then choose “Assemble & load” from the **Execute** menu. Then select “Debug Mode” from the **Execute** menu. You will see a toolbar appear at the top of the display (see Figure 3). You will also see check boxes appear in a new column on the left side in the Main RAM window. When you are in debug mode, you cannot edit the machine’s parameters and so the **Modify** menu is disabled. However, you can still edit the contents of registers or RAMs.

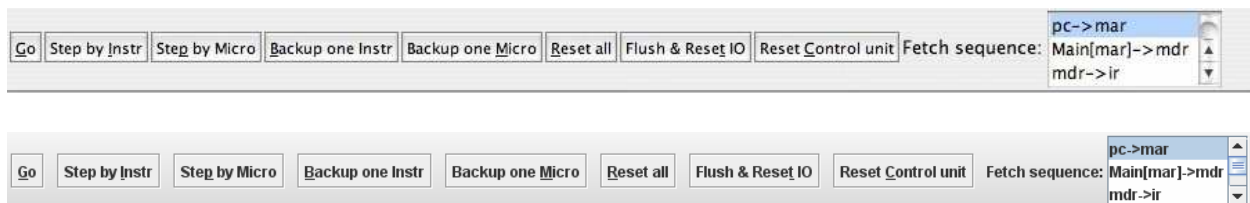


Figure 3. The debugging toolbar

Let us now use debugging mode to step through the execution of the W1-0.a program. Notice that, on its right end, the debugging toolbar says that the next instruction is the fetch sequence. If you click “Step by Instr” from the toolbar, then a complete machine cycle will be executed. Do so now. The fetch sequence will be executed, which will fetch and decode the “read” instruction, and then the “read” instruction's execute sequence will be executed, causing the machine to ask you for input in the console panel at the bottom of the desktop window. After typing the input in the console and pressing return, you will see that the value that you typed is now in the *acc* register. You can also see that the next instruction is again the fetch sequence. In

summary, each click of “Step by Instr” causes a full machine cycle to be executed, consisting of the fetch sequence followed by the execute sequence of the instruction that was fetched. As you step through the execution, note that the next machine instruction to be executed is highlighted in the RAM window. If you wish, you can edit the contents of any of the registers or RAM between steps.

Both the fetch sequence and the execute sequence of the instructions that are fetched are comprised of a series of smaller, more basic steps called “microinstructions”. The list on right side of the debug toolbar displays the microinstructions making up the current fetch sequence or instruction.

If you click “Step by Micro” from the toolbar, then only one microinstruction will execute, namely the microinstruction highlighted in the scrolling list on the right end of the debug toolbar. If the microinstruction that is executed changes a value in the RAM or Registers, then the data in the RAM or Registers that is changed during execution is outlined in green. When this microinstruction finishes execution, you will see that the next microinstruction that is to be executed becomes highlighted in the scrolling list on the right end of the debug toolbar, while the current machine instruction remains highlighted in the RAM window.

If the microinstruction that you execute is the final “end” microinstruction of a particular machine instruction, then the current machine cycle is ended. A new machine cycle will begin execution next with the fetch sequence and the scrolling list of microinstructions will be updated to display the microinstructions in the fetch sequence. The first microinstruction in the fetch sequence will be highlighted and the next machine instruction will be highlighted in the RAM window.

You can also return to any previous state by backing up one machine or microinstruction at a time. To do so, repeatedly choose “Backup one Instr” or “back up one Micro” from the toolbar. No matter what microinstruction you are currently executing, if you click “Backup one Instr”, the state of the CPU will revert back to the state at the beginning of the machine cycle. If you click “Back up one Micro”, the cells outlined in green in the Registers and RAM windows are updated to reflect those values that were changed in the previous microinstruction. To set break points in your code, check the box on the left of any line of code in the Main RAM window. When the CPU accesses that line of code (for example, when that line is loaded into the CPU for execution), the program will halt. At that point you can inspect or change any of the values in the register or RAM windows and resume execution or step forward or backward.

To learn more about debug mode, see the section of this manual entitled “EXECUTE MENU - RUNNING A PROGRAM ON THE SIMULATED MACHINE”.

At this point, you have many options. You can continue stepping through the program one instruction or one microinstruction at a time by clicking the appropriate button or you can continue execution without stopping by clicking the "Go" button. Alternatively, you can (a) backup up all the way and run the same program again with different input, (b) create or load a new assembly language program in a text window,

assemble it, and then run it, (c) create or load a new machine instead of the Wombat1, or (d) quit.

To quit CPU Sim, just choose “Quit” from the **File** menu. You may be asked whether you want to save the changes to the Wombat1 or an assembly language program. Click “No” for each of these and then CPU Sim will quit running.

CREATING NEW MACHINES IN CPU SIM, A Tour using the Wombat1

To demonstrate how to use CPU Sim to create a new hypothetical machine or CPU, we will outline the construction of the “Wombat1” machine used in the first tour. We will only demonstrate some of the principles of constructing a new machine since completing the whole machine here, especially all the microinstruction and machine instruction specifications, would be somewhat time consuming.

Start up CPU Sim or, if you left it running after the previous tour, choose “New machine” from the **File** menu. The default new machine has no registers, no memory, and no machine instructions.

The machine we will now construct, the Wombat1, is a single-accumulator machine using six registers: *acc* (the accumulator), *pc* (the program counter), *ir* (the instruction register), the *mar* (memory address register), *mdr* (the memory data register), and the *status* (the status register). The basic structure of the Wombat1 is shown in Figure 4. The *status* register is 3 bits wide, the *pc* and *mar* are 12 bits wide and the other three registers are 16 bits wide. The Wombat1 also has a main memory (RAM) consisting of 128 bytes. The arrows in the figure indicate the movement of data by the microinstructions.

The *pc* contains the address of the main memory location that contains the next instruction to be executed. In the fetch sequence (the first part of every machine cycle), the next instruction to be executed is copied into the *ir*, where the instruction is decoded. Then the instruction is executed, which completes the machine cycle. This is followed repeatedly by more machine cycles, each consisting again of the execution of the fetch sequence followed by the execution of a machine instruction.

All computations are done in the *acc*. The *mar* and *mdr* are the registers through which data is transferred to and from the main memory.

There are 12 machine instructions for the Wombat1, each associated with a 4-bit opcode. These instructions are: HALT (stop), READ (get input from the user), WRITE (send output to the user), LOAD (transfer data from the main memory to the *acc*), STORE (transfer data from the *acc* to the main memory), ADD (add a value from the main memory to the value in the *acc*, putting the result in the *acc*), SUBTRACT, MULTIPLY, DIVIDE (all similar to ADD), JMPZ (if the value in the *acc* is 0, jump to a new location to obtain the next instruction to be executed), JMPN (if the value in the *acc* is less than 0, jump to a new location to obtain the next instruction to be executed), JUMP (jump to a new location unconditionally).

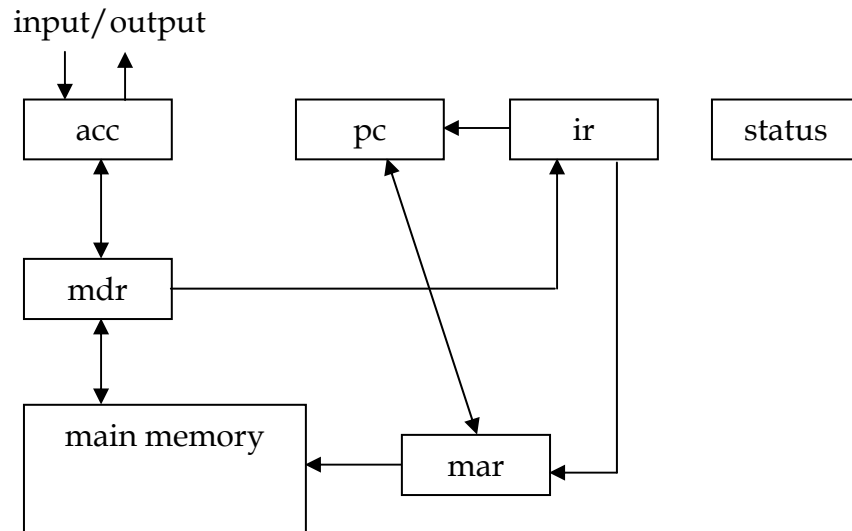


Figure 4. Diagram of data movement in the Wombat1.

We will now outline how one can create the Wombat1 machine by adding components and instructions to the default empty machine. To begin this process, it is always best to start at the lowest level, namely the basic hardware components. So choose “Hardware Modules...” from the **Modify** menu. The dialog box that appears allows you to edit the registers, register arrays, condition bits, and memory of a machine. You can add new components and modify or delete existing components. To modify a component, first select the type of the component in the popup menu at the top of the dialog. This causes the parameters associated with all existing components of that type to be displayed in the table in the center of the dialog box where they can be edited. The Wombat1 computer needs the six registers mentioned above, so select “Register” as the type of module and then click the “New” button six times. Finally edit the name and width of each of the six registers so that they match the description of the Wombat1 given above (see Figure 5).

The next piece of hardware to edit is the machine’s main memory. To do so, select “RAM” from the popup menu at the top of the Edit Modules dialog, click “New” and set the name to “Main” and length to 128 for the new memory component.

Finally, the Wombat1 needs to specify a halting condition bit and so select “ConditionBit” from the popup menu at the top of the dialog. Click “New” and then set the name of the condition bit to “halt-bit”, the register to the *status* register, the bit to 0, and check the “halt” box.

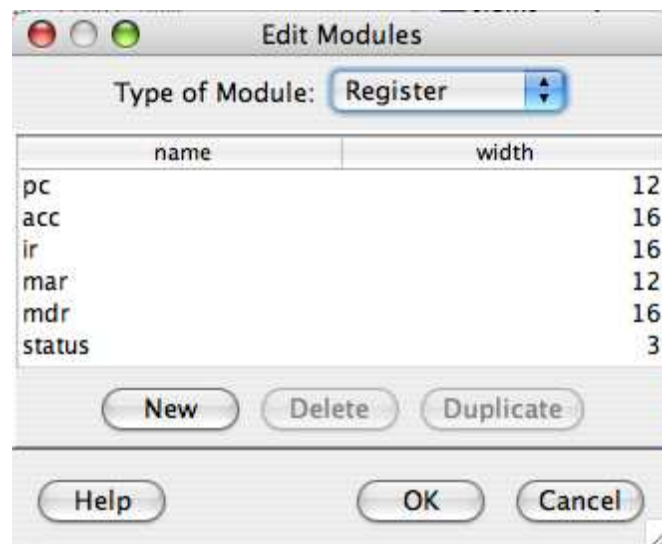
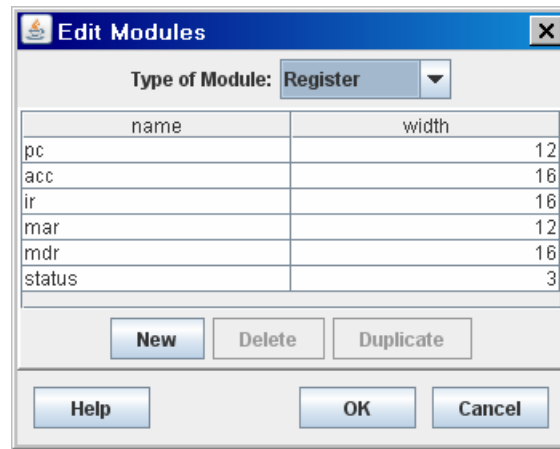


Figure 5. The dialog for editing the hardware modules.

The meanings of all the parameters of the hardware components are described in more detail at the end of this manual in the section entitled “SPECIFICATION OF THE SIMULATED HARDWARE UNITS”. You can also find out more information from the online help that appears when you click the “Help” button in this dialog box (and most other dialog boxes).

Close the Edit Modules dialog box by clicking the “OK” button.

Now we need to construct the necessary microinstructions that will be used to implement the machine instructions. There are 7 transfer, 4 arithmetic, 2 test, 1 increment, 1 decode, 2 io, 2 memory access, and 1 set condition-bit microinstructions that need to be created. The 7 transfer microinstructions are displayed by arrows between the registers in Figure 4 above.

To see how to create the transfer microinstructions, choose the menu item “Microinstructions...” from the **Modify** menu and then choose “TransferRtoR” in the popup menu at the top of the dialog that appears (see Figure 6). “TransferRtoR” type

microinstructions are for transferring a contiguous set of bits from one register to another register. To create the first transfer microinstruction, click the “New” button. We will edit this first microinstruction so that it transfers the contents of the *pc* to the *mar*. Click in the table cell with the “?” in it (in the column headed “name”) and type in “pc->mar” (you can give the microinstructions any name you want, but it helps to choose something descriptive). Select the “pc” as the “source” register and the “mar” as the “destination” register, with bit 0 as the start bit for both registers and give numBits the value 12. This means that we want to transfer all 12 bits between the registers. Now this microinstruction is complete. If you want to create the remaining 6 transfer microinstructions in the Wombat1, continue adding new microinstructions in the same manner (see Figure 6). The microinstructions of the other types needed for the Wombat1 can be created similarly by first selecting the appropriate type of microinstruction in the popup menu at the top of the dialog and then creating the microinstructions of that type. We will not go through the details in this tutorial. When you are finished, close the Edit Microinstructions dialog box by clicking “OK”.

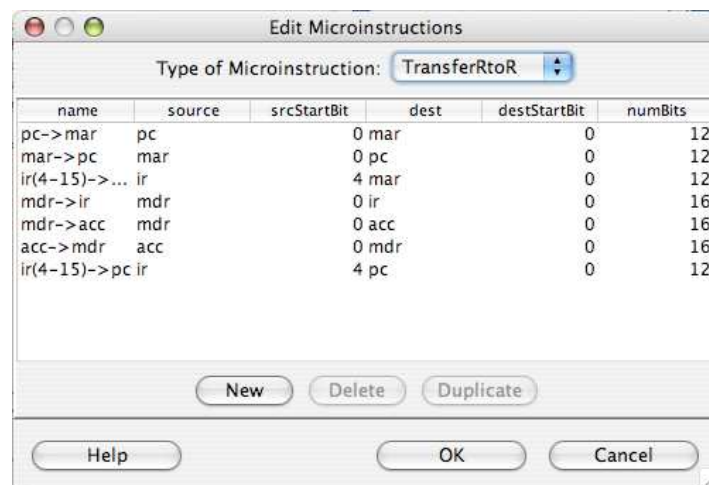
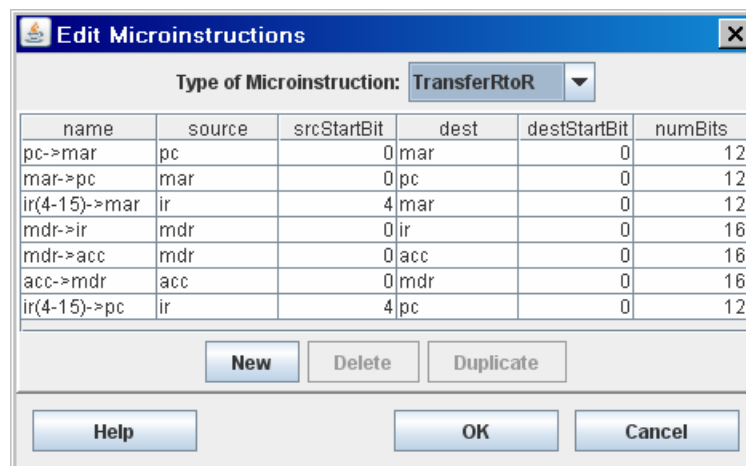
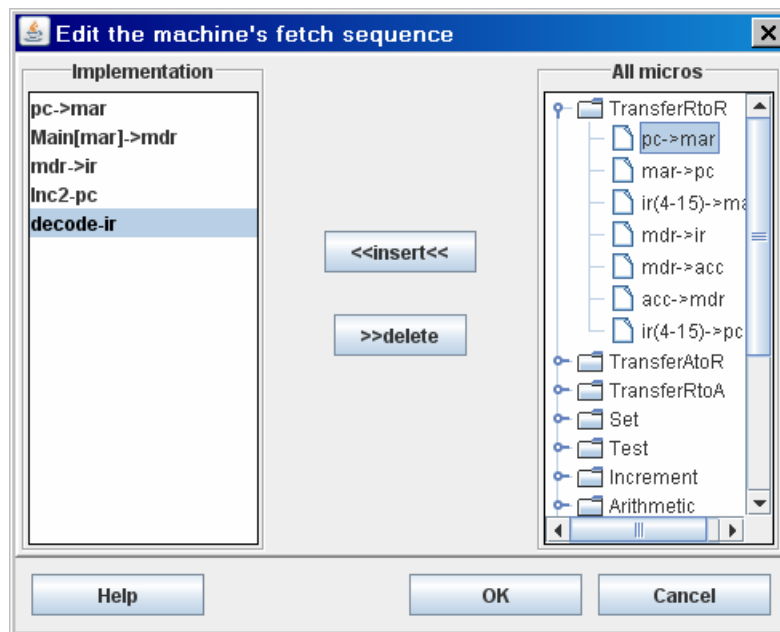


Figure 6. The dialog for editing microinstructions.

The detailed descriptions of all the parameters displayed in the microinstruction dialog boxes can be found in the section of this manual entitled “SPECIFICATION OF THE MICROINSTRUCTIONS”.

Once all the microinstructions have been created, the fetch sequence can be constructed. To do so, choose “Fetch Sequence” from the **Modify** menu. The fetch sequence is specified by a list of microinstructions. The default sequence is empty. The current fetch sequence list is displayed in the left scroll box entitled “Implementation”. To add microinstructions to it, first display the microinstructions of the type you want in the tree of microinstructions in the right side of the dialog by clicking on the dial to the right of the folder corresponding to the desired type. Then highlight the microinstruction you want by clicking on it. (Note: If you realize you need to create or modify a microinstruction before inserting it in the fetch sequence, you don’t need to close this dialog. Instead you can just double-click on the tree of microinstructions on the right and the dialog for editing and creating microinstructions will appear.) Next, to specify where the new microinstruction is to be inserted into the fetch sequence, click on the microinstruction in the left scroll box before which you want to insert the new microinstruction. Finally, click “<<insert<<” (see Figure 7).



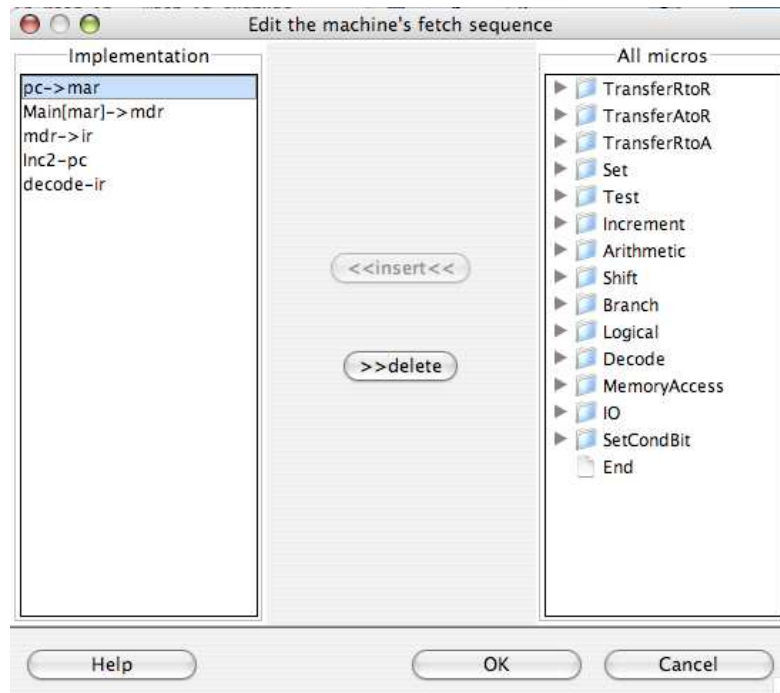


Figure 7. The dialog for editing the fetch sequence.

To remove a microinstruction from the fetch sequence, click on it in the left scroll box to highlight it, and then click “>>delete”. You can reorder the microinstructions in the list on the left by dragging them up or down to a new position. When you are done, click “OK”.

Now the 12 machine instructions of the Wombat1 need to be created. Choose “Machine Instructions...” from the **Modify** menu. The dialog box that appears allows you to edit machine instructions, including the name, opcode, field lengths, and the list of microinstructions that form the execution sequence of each instruction. The name you specify for a machine instruction is used in assembly language programs to execute that instruction. The opcode is specified in hexadecimal notation. The field lengths give the number of bits in each field of the instruction, including the opcode field (the first field) and all operand fields. For example, if an instruction has field lengths of 4 and 12, then the opcode occupies the leftmost 4 bits and the remaining 12 bits of the instruction form the only operand. All Wombat1 machine instructions have 4-bit and 12-bit fields, but for some of the instructions, the 12-bit field is ignored (such fields are denoted by putting parentheses around them in the fieldLengths column). As in other such dialog boxes, if you wish to create a new machine instruction, click “New” and then edit the parameters of that instruction (see Figure 8). Once all machine instructions have been created, click “OK”.

The detailed descriptions of all the parameters displayed in the machine instruction dialog box can be found in the section of this manual entitled “SPECIFICATION OF MACHINE INSTRUCTIONS”.

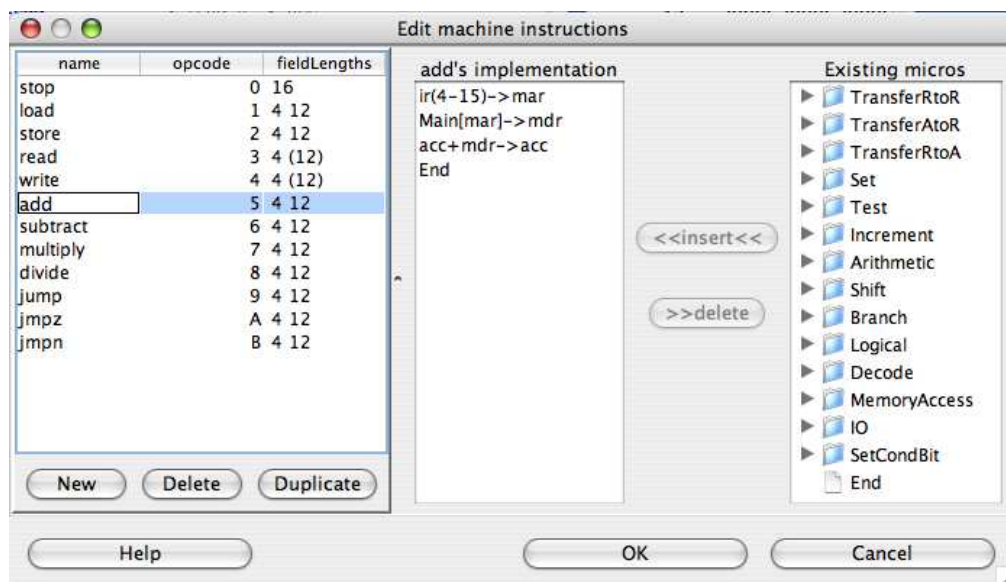
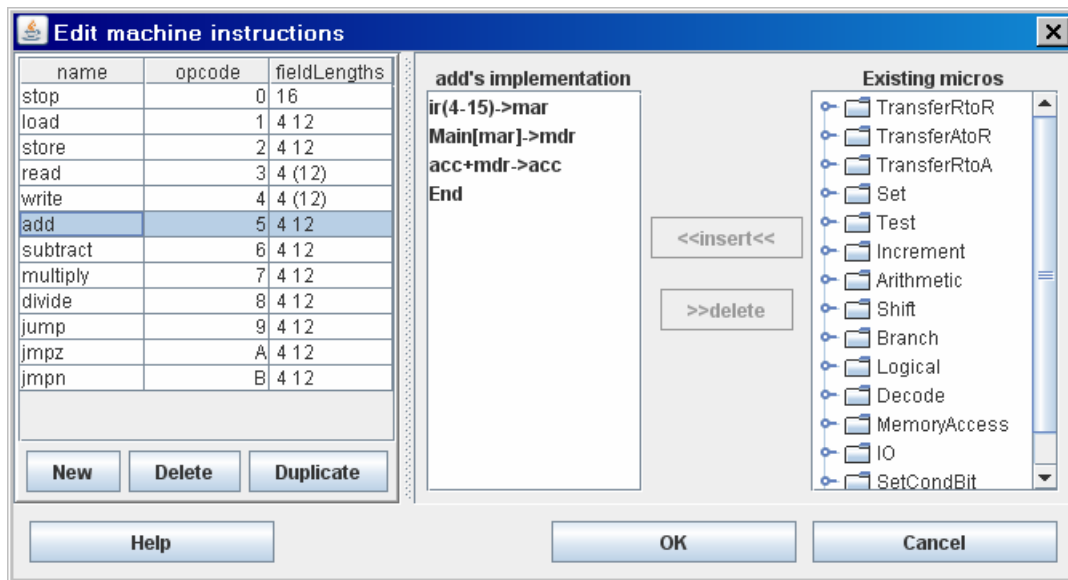


Figure 8. The dialog for editing machine instructions.

At this point, if you had created all the microinstructions, machine instructions and fetch sequence, the machine would be complete. You can at any time go back and change any part of the machine. To save the machine to a file, choose “Save machine” or “Save machine as...” from the **File** menu. CPU Sim saves the details of the machine as text in an XML file.

To see all the details of the machine in a more user-friendly format than XML, choose “Save machine in HTML...” from the **File** menu. You will be asked to type in the name of the HTML file into which the data will be saved. This file can be opened and viewed with any web browser. An HTML file displaying the details of the Wombat1 machine is included with CPU Sim in a file named “Wombat1.html”.

CREATING NEW MACHINES FROM EXISTING ONES

If you want to create a machine that is a modification of an existing machine, it is recommended that you create a copy of the existing machine by choosing “Save machine as...” from the **File** menu and typing in a new name for the file, and then modify the copy that is stored in the new file. CPU Sim does not allow you to create a new machine by cutting and pasting components from an existing machine.

While you are in the process of creating a new machine from an existing machine, you may find it frustrating in that you will not be allowed to change some parameters because of the uses to which they are currently being put. For example, you are not allowed to delete a register if one of its bits is being used as a condition bit. For another example, you cannot reduce the width of a register to less than 12 if that register is the source of a transfer microinstruction that transfers 12 of the register’s bits to another register.

To get around these problems, start by adding any new components (registers, register arrays, condition bits, and RAMs) you want. That is, first create new registers and other components without deleting any of the old ones. After this is done, adjust the microinstructions to use these new components instead of the old ones. Only after all this has been done should you delete any of the old components.

Once this task has been completed, you can change the names of the new components or instructions back to the names used by the old components if you wish. This will create no problems because CPU Sim uses the names of hardware items and microinstructions only for display purposes.

The remaining sections of this manual include descriptions of the main desktop window’s menu items, as well as a complete specification of the hardware, microinstructions, and machine instructions that can be created using CPU Sim.

FILE MENU - OPENING AND SAVING YOUR SIMULATED MACHINE AND PROGRAMS

The File menu (see Figure 9) provides the usual items for loading new or existing machines, opening or closing text windows containing assembly language programs, printing and displaying information about the machine or programs, and exiting CPU Sim.

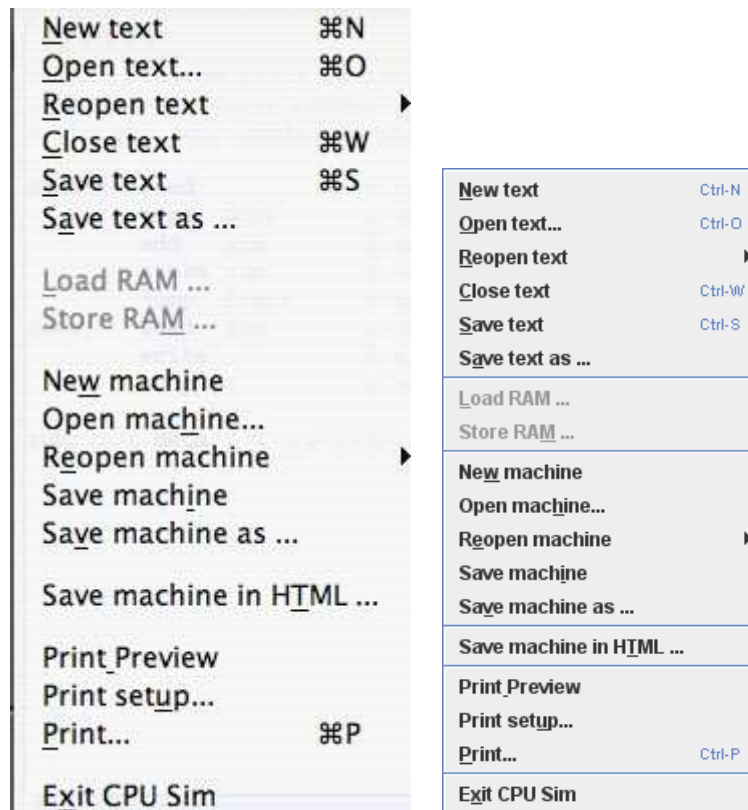


Figure 9. The File menu.

Note:

1. When a CPU Sim machine is saved to a file, all the parts are saved, including the hardware components (registers, register arrays, condition bits, RAMs), the microinstructions, the fetch sequence, and the machine instructions. The file also saves the global EQUs, the position and size of the register and RAM display windows, the highlighting, loading, and IO connection options chosen by the user (from the **Options** submenu under the **Execute** menu). However, the state of the machine is not saved; in particular, the current contents of the registers and RAMs are not saved.
2. CPU Sim machines are saved in a text file in XML format, which means that you can edit the saved machine from any text editor (including CPU Sim's built-in text editor). But you must be careful that the edited file still fulfills the data type definition (DTD) of a CPU Sim machine file or CPU Sim will not be able to load your edited machine. (Note: the DTD is included as part of the XML file.)

The "New text" menu item causes CPU Sim to create and open an empty text window in which the user can type a new assembly language program. The text window is like any basic text editor window, including the usual cutting and pasting and an arbitrary number of undo's and redo's.

The "Open text..." menu item displays a dialog box asking you to choose a file. You normally select a text file in which you had previously saved an assembly language program.

The "Reopen text" menu contains a list of the 10 most recently opened text files. This list is saved between sessions in the properties file specified on the command line when CPU Sim is started, so that when you quit CPU Sim and restart it later, the "Reopen text" menu will be populated with the same list as it had when you quit CPU Sim. If no properties file is specified when CPU Sim is started, then the "Reopen text" menu items will not be saved between sessions.

The "Close text" menu item closes the selected (topmost) text window. If the contents of the window have been changed since the window was last saved, the user is presented with a dialog box asking whether you wish to save the contents first.

The "Save text" menu item saves the contents of the selected (topmost) text window to a text file. If the user has not already saved the contents to a file, a dialog box is displayed asking the user to select a name and location for the file.

The "Save text as..." menu item displays a dialog box asking you to choose a file into which the contents of the selected (topmost) text window will be saved.

The "Load RAM..." menu item displays a dialog box asking you to choose a file. You should select a text file in which you had previously saved the contents of a RAM window using the "Store RAM..." menu item. The contents of the file will be loaded into the currently-selected RAM window.

The "Store RAM..." menu item displays a dialog box asking you to choose a text file into which the contents of the selected (topmost) RAM window will be saved. The data will be saved as text in binary format (that is, the data will be saved as strings of 0's and 1's) and the comments will be saved on the same line as the corresponding data.

The "New machine" menu item causes CPU Sim to create a new machine with no hardware (registers, condition bits, or RAMs), no machine instructions, and only one microinstruction ("End").

The "Open machine..." menu item displays a dialog box asking you to choose a file. You should select a file in which you had previously saved a machine through the use of the "Save machine" or "Save machine as..." menu items.

The "Reopen machine" menu contains a list of the 10 most recently opened machine files. This list is saved between sessions in the properties file specified on the command line when CPU Sim is started, so that when you quit CPU Sim and restart it later, the "Reopen machine" menu will be populated with the same list as it had when you quit CPU Sim. If no properties file is specified when CPU Sim is started, then the "Reopen machine" menu items will not be saved between sessions.

The "Save machine" menu item saves the current machine in a text file in XML format. If the current machine has never been saved to a file before, a dialog box appears asking you to type in the name of the file into which the machine will be saved.

The "Save machine as..." menu item presents you with a dialog box asking you to type in the name of the file into which the machine will be saved.

The "Save machine in HTML..." menu item creates a new HTML document containing a user-friendly description of the current machine, including all its details. The HTML document can be opened and viewed using any web browser.

The "Print preview..." menu item displays a thumbnail sketch of the pages as they would be printed for the currently selected text window.

The "Page Setup..." menu item brings up the usual page setup dialog, from which you can set the page size, reduction or enlargement, and other printing options.

The "Print..." menu item brings up a dialog for printing the contents of the selected window.

The "Exit CPU Sim" menu item causes CPU Sim to quit. If changes have been made to the currently loaded machine since the last time you saved it, CPU Sim asks you whether you want to save the current machine before quitting. Similarly, if changes have been made to any text window since the last time you saved it, CPU Sim asks you whether you want to save the contents before quitting.

EDIT MENU

The **Edit** menu items (see Figure 10) work with any text window. The user must first select (bring to the front) the text window to be edited.



Figure 10. The Edit menu.

"Undo" can undo the last change made in the selected text window. "Redo" will redo the last change that was undone. An arbitrary number of undo's and redo's can be done to the changes in any of the text windows.

"Cut", "Copy", "Paste", "Delete", "Select All", "Find", and "Replace" work in the usual fashion.

The "Preferences..." menu item brings up a dialog box in which you can change the font, font size, and font style for the text windows containing assembly code and for the display of the contents of Register and RAM windows. You can also change the

default syntax coloring used in text windows for the labels, keywords, and comments. The preferences you choose are saved between sessions in the properties file specified on the command line when CPU Sim is started, and so, for example, when you quit CPU Sim and restart it later, the Register and RAM windows will use the same font as when you quit CPU Sim. If no properties file is specified when CPU Sim is started, then the user's font choices will not be saved between sessions and the default font, size, and style will be used.

MODIFY MENU- CHANGING THE COMPONENTS

There are several dialog boxes that can be called up for the purpose of changing the components of the current machine. To call up these dialog boxes, just select the appropriate item from the **Modify** menu (see Figure 11).



Figure 11. The Modify menu.

Note that such modifications are not allowed when CPU Sim is in debug mode (that is, when the “Debug Mode” menu item is checked in the **Execute** menu).

Almost all of the dialog boxes display a table in which you can inspect and edit the parameters of existing components and in which you can add new components or delete components, as demonstrated in the tutorial in an earlier section of this manual. To delete a component displayed in the dialog box, select it in the table and then click the “Delete” button. To create a new component, click in the “New” button, at which point a new component will be added to the end of the table. To edit a cell, just click or double-click on it. Some cells are edited by typing new text in them, some cells are edited by selecting items from popup menus, and some cells are edited just by clicking in the check box in the cell. If you try to enter an illegal value in a cell, either the cell will become outlined in red (for example, if you type non-numeric values in a cell expecting a numeric value) or an error message will appear when you try to save the changes you made (for example, if you type in a numeric value that is out of range for that particular cell).

The columns in any of these tables can be reordered by dragging the column headers left or right to a new position. The column widths can also be adjusted by dragging the line dividing the columns. The rows can be resorted by column by clicking in any column header.

Names of Components, Microinstructions, and Machine Instructions

All hardware components, microinstructions, and machine instructions must be given names. You can name them anything you wish with the following conditions:

- You must use at least one non-space character in each name and you must have different names for each unit of the same type. It is strongly recommended that all components, microinstructions and machine instructions have unique names regardless of the type.
- Machine instruction names must be valid symbols in the assembly language syntax, which means they must consist of one or more letters, digits, underscores (_), plus signs (+), and minus signs (-), and they must start with a letter. Only ASCII characters are allowed.

Note that if you give a register array of length 4 the name "A", then the 4 registers in the array will automatically be given the names "A[0]", "A[1]", "A[2]", "A[3]".

Help buttons

Almost every dialog box in which parts of the machine can be modified contain a button labeled "Help". Clicking in the "Help" button causes the appropriate window from the online help to appear giving you some extra (hopefully helpful) information about the current dialog box.

Modifying registers, register arrays, condition bits, and RAMs

To change the registers, register arrays, condition bits or RAMs of the current machine, choose "Hardware modules..." from the **Modify** menu. You will see a dialog box with a popup menu at the top with "Register" currently selected. See Figure 5 above for an example. In the table in the dialog box, you will see a list of all existing registers. You can edit the parameters associated with any register or create new or delete old registers. The width of a register refers to the number of bits in the register. You cannot edit the contents of the registers (that is, the value stored in the registers) using this dialog box. To edit a register's contents, use the window that appears when you choose "Registers" from the **View** menu.

You can create, edit, and delete register arrays by selecting "Register Array" from the popup menu at the top of the dialog box. A register array is an indexed list of any number of registers. If the array has name "A" and if the array contains 4 registers, then those registers are denoted by "A[0]", "A[1]", "A[2]", and "A[3]".

By choosing "RAM" from the popup menu at the top of the dialog box, you can create, edit, or delete any number of RAMs of any length. The length refers to the number of bytes of data in the RAM. If you increase the length of a RAM, new memory locations containing the value 0 will be added at the end of the RAM. If you decrease the length of a RAM, some of the bytes at the end will be deleted and their contents will be lost. Note that all RAMs are byte-addressable.

By choosing "Condition Bit" from the popup menu at the top of the dialog box, you can edit the condition bits of the current machine. A condition bit is a specific bit of a register. The bit can be set to 0 or 1 by a SetCondBit microinstruction or it can be set

to 1 if an overflow or carry out occurs in an Arithmetic or Increment microinstruction. Also, you can specify whether you want the machine to halt execution when the bit gets set to 1.

The meanings of all the parameters of the hardware components are described in more detail in the section entitled “SPECIFICATION OF THE SIMULATED HARDWARE UNITS”.

Modifying the microinstructions

If you choose “Microinstructions...” in the **Modify** menu, a dialog box will appear displaying a table containing the parameters associated with the Set microinstructions. Edit their values in ways similar to the ways you edit the parameters of the registers, register arrays, RAMs, and condition bits as described above. To edit the parameters of other types of microinstructions, select the desired type from the popup menu at the top of the dialog box. See Figure 6 above for a picture of this dialog when TransferRtoR microinstructions are displayed for the Wombat1 machine. For more details on the meanings of the various parameters of each type of microinstruction, see the section entitled “SPECIFICATION OF THE MICROINSTRUCTIONS” below.

Note that CPU Sim creates only one copy of each microinstruction, regardless of how many places it is used in the fetch sequence and execute sequences of machine instructions. Therefore any changes you make to a microinstruction will affect all current uses of that microinstruction. As a result, it is usually a good idea to create a new microinstruction using the New or Duplicate buttons and edit that microinstruction rather than editing an existing microinstruction.

Modifying the fetch sequence

When you choose “Fetch Sequence...” from the **Modify** menu, you will be presented with a dialog box in which, on the left side, is listed the current sequence of microinstructions forming the fetch sequence. See Figure 7 above for a picture of this dialog when the Wombat1 machine has been loaded. If you wish to delete one of the microinstructions from the sequence displayed on the left in the dialog, select it by clicking on it. Then click the “>>delete” button. If you wish to insert a new microinstruction in the sequence, first click on the dial to the left of the type of microinstruction you want from the list in the right part of the dialog box. All existing microinstructions of that type will then be displayed. Select the one you want by clicking on it and then select in the list on the left of the dialog box the microinstruction before which you wish to insert the new microinstruction. Now click on the “Insert” button. If you want to add the new microinstruction to the end of the fetch sequence then don’t select any microinstruction in the list on the left before clicking the “<<insert<<” button. To reorder the microinstructions in the fetch sequence, just drag them up or down in the list on the left side of the dialog box.

A typical fetch sequence loads into an instruction register the contents of the memory location whose address is in a program counter register. Then it increments

the program counter and decodes the instruction in the instruction register. (An instruction register is specified as a parameter of each Decode microinstruction, as described below.) It is typically the case that the last microinstruction in the fetch sequence is a Decode microinstruction because the execution of a Decode microinstruction causes CPU Sim to (a) cease executing the fetch sequence, (b) decode the contents of the instruction register, and (c) begin executing the execute sequence of the machine instruction that was just decoded.

It is often the case that, when you are modifying the fetch sequence, you need to modify or create a new microinstruction and so you need to bring up the dialog box for modifying microinstructions. To do so, you can double-click on the list of current microinstructions on the right side of the dialog box, which will bring up the dialog for editing microinstructions. When you are finished editing the microinstructions, you can close that dialog box and can resume editing the fetch sequence. Important note: Any changes that you make to microinstructions are applied to the current machine, even if you cancel any changes to the fetch sequence.

Modifying the machine instructions

To modify machine instructions, choose “Machine Instructions...” from the **Modify** menu. The dialog box that appears (see Figure 8 above for a picture of this dialog box when the Wombat1 is loaded) is similar to the fetch sequence dialog box described above, except for the table on the left that displays all existing machine instructions’ names, their opcodes, and their field lengths.

To edit an instruction’s name, opcode, or field lengths, double-click on the entry in the table on the left and then type in the new value. For a discussion of what names are legal, see the section above regarding naming components, microinstructions, and machine instructions.

The opcode value is displayed in hexadecimal notation and a new value for the opcode must be entered as a non-negative hexadecimal value. If you type in characters that are not hex characters (the hex characters are 0-9 and a-f and A-F), the opcode box you are editing will become highlighted in red and a bell will sound when you attempt to save your changes.

The field lengths are entered as one or more positive (decimal) integers separated by one or more spaces. The sum of all field lengths must be a multiple of 8.

You may optionally surround some of the field lengths by parentheses. In doing so, you are telling CPU Sim that those fields are “don’t care” fields and are to be ignored. In this case, no operand needs to be specified in assembly language for that field when the instruction is executed. Instead, the assembler just fills that field with 0’s.

To modify the execute sequence of an instruction, first select the instruction you wish to edit by clicking on it in the table on the left. Its execute sequence or implementation (that is, the list of microinstructions that are executed when the machine instruction is executed), will appear in the center of the dialog box. Modify them as described above for the fetch sequence. To delete a machine instruction, first

select it in the table on the left. Then click the “Delete” button. To create a new machine instruction, click the “New” or “Duplicate” buttons.

The “Duplicate” button allows you to create easily new machine instructions that are variations of other machine instructions. To use this feature, select in the table on the left the machine instruction you wish to duplicate before you click the “Duplicate” button. A new machine instruction will be created with the same opcode, field lengths, and microinstructions as the instruction you had selected. You can then edit the name of the new instruction, its opcode, field lengths, and microinstructions as you would any other machine instruction.

Note that one microinstruction may appear in several machine instruction's execute sequences. As mentioned above, CPU Sim creates only one copy of that microinstruction, and so if you edit and change its parameters, then the changed microinstruction will be used in the fetch sequence and in all the machine instructions' execute sequences in which it appears.

Editing Microinstructions from the Fetch Sequence or Machine Instruction Dialogs

It is often the case that, when you are modifying the fetch sequence or the machine instructions, you will find that you need to modify or create a new microinstruction. The easiest way to do so is to double-click on the list of current microinstructions on the right side of the dialog box, which will bring up the dialog for editing microinstructions. When you are finished editing the microinstructions, you can close that dialog box and can resume editing the fetch sequence or the machine instructions. Important note: Any changes that you make to microinstructions are applied to the current machine, even if you cancel any changes to the fetch sequence or machine instructions.

Modifying the EQUs

Each machine can have some equates (EQUs) associated with them. These EQUs are global constants available for use in any assembly language program written for that machine. For example, if an EQU named “A0” with value 0 is available, then the user can type in “A0” wherever a numerical value is expected in assembly language programs and, during assembly, the symbol A0 will be replaced with the numeric value of 0. To create or edit the global EQUs, select “EQU’s” from the **Modify** menu, and then edit the cells of the table as you would any of the other tables described above. The values of EQUs must be specified as decimal integers.

Note that you can also type EQU declarations directly in your assembly language code. However, these EQU’s are local rather than global, in that they apply only to the program in which they appear. Therefore, they will not be saved with the current machine and will not appear in the global EQU table.

Note that the names of all EQUs must be valid symbols in the assembly language syntax described in the section “SPECIFICATION OF ASSEMBLY LANGUAGE FOR CPU SIM MACHINES” below. That is, they must consist of one or more letters or digits or '+' or '-' or '_' characters, and they must start with a letter. Only ASCII characters are

allowed.

EXECUTE MENU - RUNNING A PROGRAM ON THE SIMULATED MACHINE

Execution of an assembly language program in CPU Sim involves the following steps:

1. load into CPU Sim the virtual machine on which your program is to be run,
2. initialize the contents of the registers to the values you desire,
3. create or open a text file that contains the assembly code to be run,
4. assemble and load the program into the main memory (RAM) of your simulated machine,
5. run the program.

You have the choice of running the program without interruptions (other than those caused by errors in your code) by choosing either “Assemble, load & run” or “Run” from the **Execute** menu (see Figure 12), or you can step through the program one fetch sequence and execute sequence or one microinstruction at a time by using debug mode.

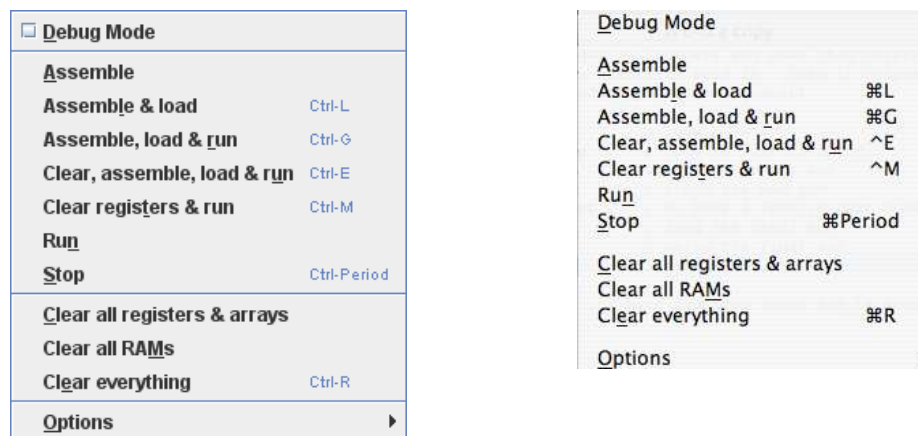


Figure 12. The Execute menu.

The "Debug mode" menu item causes the debug toolbar to be displayed (see Figure 3 above). The use of the debug toolbar is described below.

The “Assemble” menu item checks your code for legality. If the code is legal, nothing happens. If the code is illegal, an error message will appear and the offending line of assembly code will be highlighted for you to edit. Note that the code in any text window must be saved to a file before it can be assembled. If you try to assemble it before saving it, a dialog box will ask you to save it first. Note also that CPU Sim does not create executable files or object files. Every time you want to run a program, that program must be reassembled.

The “Assemble & load” menu item assembles your code and, if it is legal, loads

the assembled (machine) code into the simulated machine's memory. The RAM into which the code is loaded is not first cleared (set to 0), and so the only memory locations that are changed are those into which the new code is loaded. If you want to clear the rest of RAM, you need to do so first by choosing the appropriate menu item from the **Execute** menu. Note: You can use the Options submenu of the **Execute** menu (discussed below) to specify which RAM is to be loaded with the machine code and what starting address is to be used.

The "Assemble, load, & run" menu item is a convenience item that is equivalent to choosing the "Assemble & load" menu item followed by the "Run" menu item. Similarly, the "Clear, assemble, load & run" menu item is a convenience item that is equivalent to choosing the "Clear everything" menu item followed by "Assemble, load & run".

The "Run" menu item causes the machine to begin execution. The program will always begin execution with the first microinstruction of the fetch sequence. The program will stop when one of the following events occurs: (a) one or more condition bits that are designated as halt bits have their values sets to 1, (b) an error occurs, (c) the user chooses "Stop" from the **Execute** menu, (d) the user selects "Cancel" when an input dialog is displayed, (e) the user presses enter or return instead of a value when input is requested in the console panel.

The "Stop" menu item causes the currently executing program to halt. This menu item is particularly useful if the program gets into an infinite loop.

The "Clear <X>" menu items are convenience items for initializing registers or other components to 0 before beginning execution of a program. The "Clear everything" menu item causes all registers, register arrays, and RAMs to be cleared. You can also clear any individual register or cell of RAM by editing its contents in the register or RAM windows.

The "Options | Highlighting..." menu item brings up a dialog box in which the user can choose which lines of code are to be highlighted when the code is being executed. Note that this highlighting occurs only when in debug mode. The highlighting is specified via pairs consisting of a register and an RAM. The row of the RAM whose address corresponds to the value of the register is highlighted. For example, the user may wish to highlight the row of the stack corresponding to the address in a stack top pointer register. The rows of RAM are highlighted after each execution step. If the user wishes the highlighting to be updated after each microinstruction step, then the "dynamic" checkbox should be checked. Otherwise, the highlighting is updated only at the beginning of each machine cycle.

The "Options | Loading..." menu item brings up a dialog box in which the user can specify which RAM is the code store, i.e., the RAM in which assembled code is to be loaded. The user can also specify which address of the code store is to be the starting address for loading.

The "Options | IO connections..." menu item is discussed in the next section.

IO Options

The "Options | IO connections..." menu item brings up a dialog box in which the user can specify where each IO microinstruction will get or put data. The dialog box has a table with one row for each IO microinstruction in the current machine. The "name" column gives the name of the IO microinstruction. The "connection" column gives the current source or destination for the data when the microinstruction is executed. If the connection is "[User]" then the data is sent to or read from the user via a dialog box, one data value at a time. If the connection is "[Console]" then the data is sent to or read from the user via the console, one data value at a time. If the connection is a file name, then the data is sent to or read from that file. The file must be a text file. To change the connection for an IO microinstruction, click on the connection entry for that microinstruction to bring up a popup menu and select "[User]", "[Console]", or "File...". If you select "File...", a dialog box will appear in which you can choose a new text file to be the source or destination of the data. If two IO microinstructions both read from the same file or both write to the same file, they do so using the same data stream. That is, if one IO microinstruction reads a value from the file, then the other IO microinstruction cannot read that value from the file and instead will read the next value.

When CPU Sim attempts to read an integer from a text file, it reads past any white space (space, new line, carriage return, and tab characters), reads an optional '+' or '-' character, and then reads and appends digits (0-9) until a non-digit character is encountered. When CPU Sim attempts to read a character (ASCII or Unicode) from a file, it just reads the next character of the file. . If you want to store binary data in a text file to be read by an IO microinstruction, precede the binary value with a "0b". For example, if you want to store the binary value of "1101" then the text file should have the value "0b1101". If you want to store hexademical data in a file for later reading by an IO microinstruction, precede the hexadecimal value with the prefix "0x".

When CPU Sim attempts to write an integer to a text file, it first writes a space character so that successively written integers can be distinguished from each other. It then writes the integer in decimal format. When CPU Sim attempts to write a character (ASCII or Unicode) to a file, it writes the character immediately with no additional spacing.

When a program is run using the "Assemble, load, & run", "Run", or "Clear, assemble, load & run" menu items in the **Execute** menu, the data files are opened just before running and are closed when execution is halted (for whatever reason). If the user enters debug mode, the files are opened when execution begins for the first time (by clicking the "Go", "Step by Instr", or "Step by Micro" buttons). When in debug mode, the files are not closed until the user clicks the "Reset All" or "Flush & Reset IO" buttons in the debug toolbar, the user exits from debug mode, or the user selects a new connection (using the "Options | IO connections..." menu item) for the IO microinstruction connected to the file.

NOTE: Clicking the "Backup one Instr" or "Backup one Micro" buttons in the debug toolbar does not back up the reading or writing of text files. That is, the current position of reading from or writing to text files does not change when you click the "Backup one Instr" or "Backup one Micro" button.

Debug mode

Debug mode provides you with several tools for finding bugs in your code, including the following:

- You can step through the code one machine instruction or one microinstruction at a time, and, as you do so, the name of the current machine instruction being executed is displayed on the right end of the debug toolbar, along with the microinstructions composing it, which are displayed in a scroll box. The specific microinstruction being executed is highlighted in the scroll box. If you step by microinstruction, then any register and RAM cells that are updated during execution are outlined in green. After each step you can inspect and change the values stored in any register or RAM.
- You can back up one machine instruction at a time, which resets the values of the registers and RAMs to their state at the start of the last machine cycle.
- You can also back up one microinstruction at a time, which resets the values of the registers and RAMs to their state at the start of the last microinstruction.
- You can specify cells of RAMs to be highlighted based on the address stored in specified registers as you step through the code (using the "Options | Highlighting..." menu item, as discussed above). For example, you can cause the top cell of the stack to be highlighted after each step.
- You can specify break points at any cell in any RAM by checking the box on the left side of that cell in the RAM window. When such a cell is accessed (via a MemoryAccess microinstruction) for either reading or writing, the MemoryAccess microinstruction is executed and then execution halts. When the program halts, the address in RAM that caused the break is highlighted in red. Additionally, if the RAM window is not open at the time that the break is reached, it will be opened and, if the address in the RAM window where the break is located is outside of the view window, CPU Sim will scroll to the specified location. At that point, you can inspect and/or edit the contents of all registers and RAMs and back up or resume execution. Once execution is resumed, the RAM address where the break occurred will no longer be highlighted.

To enter debug mode, select the "Debug mode" menu item in the **Execute** menu. This action will cause (a) a debug toolbar to appear below the main menu (see Figure 3) and (b) a new column of checkboxes to appear on the left side of all RAM windows.

The debug toolbar contains 8 buttons and a text component and a scroll box. The text component near the right end of the toolbar always displays the machine instruction that is about to be or is currently being executed. At the start of a machine cycle, it displays the fetch sequence first. There is also a scroll box to the right of the text component that lists the microinstructions comprising the current machine instruction or fetch sequence.

The "Go" button causes execution to continue without interruption until one of

the following happens:

- a condition bit that is designated as halt bit has its value set to 1,
- a break point is reached,
- the user stops the current execution by choosing "Stop" from the **Execute** menu
- the user chooses Cancel when an input dialog appears
- the user presses enter or return before entering a value when the Console panel asks for input,
- an error occurs.

The "Step by Instr" button causes the execution of one full machine cycle (a fetch sequence followed by the execute sequence of the instruction decoded by the fetch sequence). If part of a machine cycle has already been executed (by means of the "Step by Micro" button or if the execution of a whole cycle was halted because of an error), then clicking this button causes the rest of that machine cycle to be executed. The name of the machine instruction currently being executed is displayed in the debug toolbar near the right end.

The "Step by Micro" button causes execution of one microinstruction in the current fetch or execute sequence. The name of the microinstruction to be executed is highlighted in the scroll box on the right end of the debug toolbar.

The "Backup one Instr" button causes the machine to back up to the state it was in at the beginning of the last machine cycle. You can continue backing up one machine instruction at a time all the way to the initial state of the machine when you began execution of the current program using the current machine in debug mode.

The "Backup one Micro" button causes the machine to back up to the state it was in at the beginning of the previous microinstruction. You can continue backing up one microinstruction at a time all the way to the initial state of the machine. If you have been backing up by microinstruction and then use the "Backup one Instr" button, the machine will back up to the state it was in at the beginning of the last machine cycle. From there you can back up by machine instruction or microinstruction, or you can step forward by machine instruction or microinstruction.

The "Reset All" button causes the machine to back up all at once to the initial state it was in when the user starting executing the current program in debug mode using the current machine. This action also causes the IO to be flushed and reset and the control unit to be reset. This buttons functions similarly to "Clear everything" in the **Execute** menu.

The column of checkboxes in the RAM windows that appears when CPU Sim is in debug mode is used for setting break points. CPU Sim will halt execution whenever a MemoryAccess microinstruction attempts to read or write to a cell of memory with a checked box. More precisely, the MemoryAccess microinstruction will be executed, the row of the RAM window with the checked box will be highlighted in red (if the RAM window is closed, it is opened automatically) and then execution will halt. At that point, you can inspect and/or change the contents of any register or RAM and continue execution or back up.

By checking the box in front of a memory cell containing a machine instruction, you can halt execution when that instruction is reached (since that instruction needs to be loaded into a register via a MemoryAccess microinstruction before it can be executed).

By checking the box in front of a memory cell containing data—for example, data on the system stack—you can halt execution when that data is read or written.

Note:

- Between every step forward or backward, you can inspect and edit any values in the registers or RAMs.
- One way to find errors in your programs is to step by machine instruction until you reach a point where an error has already occurred. Then back up to the instruction just before the error occurred. Finally step forward one microinstruction at a time until you reach the precise microinstruction in which the problem occurred.
- If your program seems to be in an infinite loop, you can choose "Stop" from the Execute menu to halt the execution.

VIEW MENU - DISPLAYING COMPONENTS OF THE CURRENT MACHINE

The **View** menu provides a way for the user to open and/or bring to the front the windows displaying the components (registers or RAMs) of the current machine (see Figure 13).

Registers
Register array A
RAM Main
RAM Stack

Figure 13. The View menu.

The "Registers" menu item will open a window displaying all the registers of the current machine, including their widths and values. You can display the registers' values in either binary, hexadecimal, decimal using 2's complement notation, unsigned decimal, ASCII, or Unicode.

Below the Registers menu item is a list of all the register arrays in the current machine. By selecting one of these menu items, you can display the registers of the selected register array and their values.

Below the register arrays are listed all the RAMs of the current machine. When you select one of these menu items, a window opens displaying the contents of the RAM. You can view the contents in binary, hexadecimal, decimal using 2's complement notation, unsigned decimal, ASCII, or Unicode and you can independently view the

addresses in binary, hexadecimal or unsigned decimal. You can also adjust the cell size and so view the contents of the RAM by individual bytes or by groups of bytes. For example, if you choose a cell size of 2, the RAM window will treat each pair of bytes as a 16-bit value and display it as such.

The "Comments" column of a RAM window can be used to store comments regarding the use of each memory cell. When an assembly program is assembled and loaded into the RAM, the rows in the "Comments" column next to each machine instruction contain the corresponding assembly language instruction from which that machine instruction was generated.

Note that every cell in the RAM window's table is editable except for the cells in the address column.

TEXT MENU - DISPLAYING TEXT EDITING WINDOWS

The **Text** menu provides a way for the user to open and/or bring to the front windows for displaying and editing text, usually consisting of assembly language programs (see Figure 14).

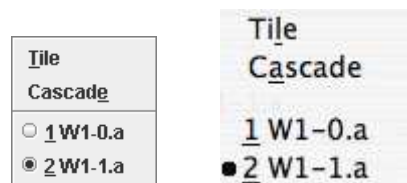


Figure 14. The Text menu.

The "Tile" menu item causes all text windows to expand to fill the desktop, with the space divided as evenly as possible among the windows.

The "Cascade" menu item causes all the text windows to overlap in a cascading manner from the upper left corner of the desktop.

The bottom part of the menu lists all currently opened text windows with the topmost one indicated by the radio button. You can select any of these items to bring that window up front.

THE HELP MENU - GETTING ON-LINE HELP

The "General CPU Sim Help" menu item brings up the online help window, which contains virtually the same material as this manual.

The "About CPU Sim..." menu item brings up a dialog box describing the CPU Sim application, including the current version and contact information.

KEYBOARD SHORTCUTS IN CPU SIM

There are many keyboard shortcuts for general actions – such as menu item actions – and many shortcuts specifically for editing assembly language programs in text windows. Here is a list of those shortcuts. Note that the Apple key is used instead of the Ctrl key on Macintosh computers.

General Keyboard Shortcuts

<i>Shortcut</i>	<i>Action</i>
Ctrl-N	New text editor
Ctrl-O	Open text editor
Ctrl-W	Close text editor
Ctrl-S	Save text editor
Ctrl-P	Print
Ctrl-Z	Undo
Ctrl-Y	Redo
Ctrl-X	Cut
Ctrl-C	Copy
Ctrl-V	Paste
Ctrl-A	Select all
Ctrl-F	Find
Ctrl-H	Replace
Ctrl-G	Assemble, load, & run
Ctrl-.	Stop
Ctrl-R	Clear everything
Alt-F	Open File menu
Alt-E	Open Edit menu
Alt-M	Open Modify menu
Alt-X	Open Execute menu
Alt-V	Open View menu
Alt-T	Open Text menu
Alt-H	Open Help menu

Text Editor Keyboard Shortcuts

<i>Shortcut</i>	<i>Action</i>
Ctrl-Backspace	Backspace a word
Ctrl-Delete	Delete a word
Ctrl-\	Toggle selection rect
Shift-Home	Select Home
Shift-End	Select End
Ctrl-Home	Document Home
Ctrl-End	Document End

Ctrl-Shift-Home	Select Document Home
Ctrl-Shift-End	Select Document End
Shift-Left	Select previous char
Shift-Right	Select next char
Ctrl-Left	Previous word
Ctrl-Right	Next word
Ctrl-Shift-Left	Select previous word
Ctrl-Shift-Right	Select next word
Shift-Up	Previous line
Shift-Down	Next line
Ctrl-Enter	Repeat

SPECIFICATION OF ASSEMBLY LANGUAGE FOR CPU SIM MACHINES

Syntax

The following context free grammar (CFG) gives the syntax of legal assembly language programs. The start symbol for the CFG is `Program`. `EOL` is an end-of-line token and `EOF` indicates the end of the file. Square brackets indicates an optional item. Items in parentheses followed by `"*"` indicate 0 or more copies of those items. Items in parentheses followed by `"+"` indicate 1 or more copies of those items. To separate tokens that the assembler would otherwise treat as one token, use one or more spaces or tab characters. Terminal symbols are displayed in bold and are surrounded by quotes. Terminal symbols are case sensitive. A range of characters, such as all small case letters of the alphabet, are indicated by the first character of the range and the last character separated by a dash, as in `"a-z"`. All characters in an assembly language program must be ASCII characters.

```

Program → [CommentsAndEOLs] EquMacroIncludePart InstructionPart EOF
CommentsAndEOLs → ([Comment] EOL)+
Comment → ";" <any-sequence-of-characters-not-including-EOF-or-EOL>
EquMacroIncludePart → ((EquDeclaration | MacroDeclaration |
    Include) CommentsAndEOLs)*
Include → ".include" <quoted-sequence-of-characters-not-including-
    EOL-or-EOF-surrounded >
EquDeclaration → Symbol "EQU" Operand
MacroDeclaration → "MACRO" Symbol [Symbol ([","] Symbol)*]
    CommentsAndEOLs InstructionPart "ENDM"
InstructionPart → ((RegularInstructionCall |
    DataPseudoinstructionCall | AsciiPseudoinstructionCall)
    CommentsAndEOLs)*
RegularInstructionCall → (Label CommentsAndEOLs)* [Label] Symbol
    [Operand ([","] Operand)*]
DataPseudoinstructionCall → (Label CommentsAndEOLs)* [Label]
    ".data" Operand [","] (Operand | [Operand [","]] "[" [Operand
    ([","] Operand)*] "]")
AsciiPseudoinstructionCall → (Label CommentsAndEOLs)* [Label]

```

```

    ".ascii" String
Label → Symbol ":"
Operand → Symbol | Literal
Literal → [ "-" | "+" ] ( ( 0-9 )+ | "0x"( 0-9a-fA-F )+ | "0b"( 0 |
    1 )+ | <single-quoted-character> )
Symbol → ( <letter> ) ( <letter or digit or - or + or _> ) *

```

Here is a summary of the parts of an assembly language program. The basic building blocks consist of the following items.

A *literal* is one of the following:

- (a) a decimal integer,
- (b) a hexadecimal integer (denoted by the prefix "0x" or "-0x" followed by one or more of the characters 0-9, a-f, A-F),
- (c) a binary integer (denoted by the prefix "0b" or "-0b" followed by one or more 0's or 1's),
- (d) a single character surrounded by single quotes.

Literals other than single-quoted characters can have an optional plus or minus sign in front. No commas or decimal points are allowed in literals. In the case of the single-quoted character, the value of the literal is the ASCII value of the character.

A *string* is any sequence of characters surrounded by double quotes, such as "abcde". Note that the sequence of characters inside the quotes cannot include the double-quote character and that the characters must all be on the same line.

A *symbol* consists of a sequence of characters, the first character of which must be an upper or lower case letter. The remaining characters must be letters or digits or + or - or _. CPU Sim distinguishes between upper and lower case letters; hence, "Data" and "data" are considered different symbols.

A *label* is a symbol followed immediately by a colon. The colon is just a separator and is not considered part of the label. The label and colon pair is an optional feature on every line of assembly language programs including those lines that are otherwise blank or contain only comments. In the latter two cases, the assembler will treat the label as if it referred to the next regular instruction or data pseudoinstruction. Labels can be used as operands in statements.

A *comment* is any sequence of characters preceded by a semicolon ";" and ending at the end of the line. Any line of a program can contain a comment. Blank lines and lines containing only comments are also allowed in assembly language programs, and are ignored by the assembler. When the program is assembled, regular instructions and data pseudoinstructions, including the comments on the ends of the lines, are saved and appear after each line of the assembled program in the Comments column of the RAM window into which the assembled program is loaded. However, remember that blank lines or lines with only comments are discarded when the program gets assembled.

A program consists of two parts. The first part contains any number of EQU declarations, include directives, and macro definitions in any order. The second part

contains any number of regular instructions and data pseudoinstructions, one per line.

Include directives

An *Include* directive is a pseudoinstruction used for temporarily inserting the text from one file into the text of another file just before assembly. The pseudoinstruction has the following parts:

label: .include text-file-name comment

The word “.include” and the text file name must be included, but the comment and the label are optional. The text file name must be a string of 1 or more characters surrounded by double quotes or chevron quotes. The string cannot include the end-of-line or return or double-quote characters. For example, suppose a file *F* contains the include directive

```
.include "W1-0.h"
```

Then, at the point in *F* at which the include directive appears, the contents of file “W1-0.h” are temporarily inserted for the purpose of assembly. The quoted part of the Include statement must be the pathname of the file to be inserted. This pathname is relative to the file *F*. For example, the Include directive above assumes that the file “W1-0.h” is in the same directory as *F*. If the file “W1-0.h” is in a subdirectory called “defs”, for example, then the include directive should say:

```
.include "defs/W1-0.h"
```

If the file “W1-0.h” is in the parent directory of *F*, then the include directive should say:

```
.include "../W1-0.h"
```

If the file *F* contains the include directive

```
.include <W1-0.h>
```

then CPU Sim will search for the file W1-0.h relative to the directory containing the current machine (that is, the .cpu file containing the current machine) instead of relative to the directory containing *F*.

Macro declarations and calls

Macro declarations give a name to a block of code. This block is called the macro *body*. A macro *call* is the use of a macro name as an instruction. The assembler textually inserts the macro body wherever a macro call occurs in an assembly language program. For example, consider the following macro declaration

```
MACRO foo
    add x, y
    load z
ENDM
```

Assume that the declaration appears in a file that includes the following 3 instruction statements:

```
add a, b
foo ; macro call
store b
```

In this case, the text that is assembled is the following:

```
add a, b
add x, y
load z
```

```
store b
```

Macros can also have parameters, which get textually replaced by the arguments in the macro call. For example, consider the following variation, which has one parameter *n*:

```
MACRO foo n
    add x, n
    load z
ENDM
```

Assume that this declaration appears in a file that includes the following 3 instruction statements:

```
add a, b
foo a ; macro call with argument a
store b
```

In this case, the text that is assembled is the following:

```
add a, b
add x, a
load z
store b
```

You can include arbitrary instructions inside a macro body, including labeled statements, data pseudoinstructions, and calls to other macros.

Note that macro declarations are always local, which means that these macros can only be called in the file in which they are defined. Therefore, it is often useful to put the macro definitions in files that are then included (using the `.include` statements) in all files in which you wish to make macro calls.

It is worth noting that, because of the textual substitution, labels in macro bodies must be treated specially. To understand the problem, consider a macro body in which one line has a label *L*, and suppose the macro is called twice in one file. If straight text substitution were performed, the label *L* would appear on two lines of that file, violating the necessary uniqueness of all labels in a program. Therefore, before the textual substitution of every macro call, the label and all references to the label in the macro body are replaced with a unique label. For this reason, labels in macro bodies cannot be referenced outside the macro body.

EQU declarations

EQU declarations in programs define names for integer constants. Those names can be used anywhere in the program where a number is expected. It is legal to declare an EQU name whose value is a previously-defined EQU name. For example, it is legal to make the following declarations:

```
A EQU 4
B EQU A
```

In this example, both *A* and *B* are now names for the constant 4.

Note that the names of all EQUs must be valid symbols and the values of all EQUs must be previously-defined EQU names or valid literals in the assembly language syntax. The literal value must be in the range -2^{63} to $2^{63}-1$ (–9223372036854775808 to 9223372036854775807).

Note that EQU declarations are always local, which means that the declared names can only be used in the file in which they are declared. Therefore, it is often

useful to put the EQU declarations in files that are then included (using the `.include` statements) in all files in which you wish to use the declared names.

You can also make global EQU declarations as described above in the section detailing the **Modify** menu. If a global EQU declaration and a local EQU declaration both use the same symbol (which is legal), the local EQU declaration has precedence and so hides the global EQU declaration.

Regular Instructions

Regular instructions have the following form (some of the parts are optional):

label: operator operands comments

Space and tab characters are only required between two parts if the assembler might otherwise treat the parts as one unit. Statements are terminated by the newline character. The label and comments are optional.

An *operator* is a symbol consisting of the name of a machine instruction of the current machine or the name of a macro. Every regular instruction must have an operator.

An *operand* is a symbol or literal. If it is a symbol, the same symbol must appear somewhere in the program as a label or it must be an EQU constant. For each instruction, there must be one operand for each field specified for that instruction, not counting the first field, which corresponds to the opcode, and not counting the "don't care" fields. That is, if a machine instruction has fields consisting of 4 2 10, then the opcode for the instruction must be 4 bits long and the instruction must have 2 operands, the first one having a 2-bit value and the second having a 10-bit value. If a machine instruction has fields consisting of 4 2 (10), where the "(10)" indicates a "don't care" field, then the opcode must be 4 bits long and the instruction must have one 2-bit operand. Operands must be separated from each other by a comma and/or one or more spaces and tabs.

Note that there are three types of sources of values for operands that are symbols: local EQUs (defined in the file), global EQUs (defined in the machine), and labels (defined in the file). A symbol might appear in all three types of sources. In that case, local EQUs have priority, followed by global EQUs, and finally labels.

Data pseudoinstruction

The word `".data"` signifies a pseudo-instruction in assembly language. It provides a way for the user to insert specific numerical values in specific locations in the assembled code. The pseudoinstruction has the following parts:

label: .data operands comment

The word `".data"` and the operands must be included, but the comment and the label are optional. The label can be used as an operand in assembly language statements to refer to the first address of the data that is being specified. The pseudoinstruction is terminated by the end-of-line character.

The first operand must have a positive numerical value *n* that indicates the number of bytes of data that are being given specific values.

This operand is followed by one of the following:

- one long value that will be stored using the given number of bytes.
- an integer between 1 and 8 indicating the cell size in bytes followed by a bracketed list of values to be put in successive cells of the given size. Therefore the number of values in the bracketed list must equal the total number of bytes (specified in the first operand) divided by the cell size (specified in the second operand). The list items are optionally separated by commas.
- a bracketed list of values whose length is equal to the number of bytes specified in the first operand. These values are optionally separated by commas. Note that this case is just a special case of the preceding case in which the cell size is omitted and is implicitly assumed to be 1 byte.

If any of the numbers are negative integers, they will be treated by CPU Sim in 2's complement form. Note that the values can be either integer literals, EQU names, or variables corresponding to labels. In the last case, the value of the variable is the address of the label.

For examples, consider the following pseudoinstructions:

```
.data 5 0 ;fills 5 bytes with 0's
.data 5 [0,0,0,0,0] ;fills 5 bytes with 0's
.data 5 1 [0,0,0,0,0] ;fills 5 bytes with 0's
.data 5 5 [0] ;fills 5 bytes with 0's
.data 5 -1 ;fills 5 bytes with all 1 bits
           ;(the 2's complement form for -1)
.data 5 3 ;fills the first 4 bytes with 0's and
           ;the 5th byte with the bits 00000011.
.data 5 [1,2,3,4,5] ;fills 5 bytes with the five 1-byte
                   ;values 1, 2, 3, 4, 5
.data 6 2 [4 2 7] ;fills 6 bytes with the three 2-byte
                 ;values 4, 2, 7
```

Ascii pseudoinstruction

The word ".ascii" signifies another pseudo-instruction in assembly language. It provides a way for the user to insert specific ascii values in specific locations in the assembled code. The pseudoinstruction has the following parts:

label: .ascii operand comment

The word ".ascii" and the operand must be included, but the comment and the label are optional. The label can be used as an operand in assembly language statements to refer to the first address of the data that is being specified. The pseudoinstruction is terminated by the end-of-line character.

The operand must be a string of 0 or more characters surrounded by double quotes. The string cannot include the end-of-line or return or double-quote characters:

```
.ascii "abcde" ; legal
.ascii "abc"de" ; illegal--contains a double quote character.
```

The effect of including an Ascii pseudoinstruction in your program is that the assembler inserts the ASCII numeric value of each of the characters in the string into the program at the point where the Ascii pseudoinstruction appears. So the first example pseudoinstruction above will be assembled into 5 bytes containing the 5 ASCII numeric

values for the characters 'a', 'b', 'c', 'd', and 'e'.

NAMES OF HARDWARE COMPONENTS, MICROINSTRUCTIONS AND MACHINE INSTRUCTIONS

All hardware components, microinstructions, and machine instructions must be given names. You can name them anything you wish under the following conditions. You must use at least one non-space character and you have different names for each unit of the same type. Machine instruction names must be valid symbols in the assembly language syntax, which means they must consist of one or more letters or digits or '+' or '-' or '_' characters and they must start with a letter. Only ASCII characters are allowed in Machine instruction names.

It is strongly recommended, but not required, that all units of all types have unique names.

Note that if you give a register array of length 4 the name "A", then the 4 registers will automatically be given the names "A[0]", "A[1]", "A[2]", "A[3]".

SPECIFICATION OF THE SIMULATED HARDWARE UNITS

There are four different types of hardware units that can be specified by the user:

1. Registers are storage locations for one value or bit string. They can hold one bit or many bits. Numbers are stored using 2's complement notation.
2. Register arrays are indexed lists of registers.
3. Condition bits are specific bits of registers.
4. RAMs are indexed lists of bytes. The index of a byte is referred to as the *address* of the data in that location. Data is accessed in the RAM through MemoryAccess microinstructions.

The parameters associated with each of these hardware components are discussed in each section below. Note that every components must also have a name as described above.

Register

width: a positive integer that specifies the number of bits in the register.

Register Array

width: a positive integer that specifies the number of bits in each register in the array.

All registers in an array must have the same width.

length: a positive integer specifying the number of registers in the register array.

Condition Bit

register: the register that contains the bit

bit: the bit of the register that is the condition bit. This is a number from 0 to one

less than the width of the specified register. Bit 0 is on the left-most bit of the register.

halt: a boolean value. If this value is true, then the machine will halt and display a message after the execution of any microinstruction that causes the bit's value to be set to 1.

RAM

length: a positive integer that specifies the number of bytes in the RAM. The first byte has index (address) 0.

SPECIFICATION OF THE FETCH SEQUENCE

The *fetch sequence* is a sequence of microinstructions that begins the execution of every machine cycle. A typical fetch sequence loads into an "instruction" register the contents of the memory location whose address is in a "program counter" register. Then it increments the program counter and decodes the instruction in the instruction register. It is typically the case that the last microinstruction in the fetch sequence is a Decode microinstruction, because the execution of a Decode microinstruction causes the machine to (a) cease executing the fetch sequence, (b) decode the instruction register, and (c) begin executing the execute sequence of the machine instruction that was just decoded. In other words, no microinstructions of the fetch sequence get executed after a Decode microinstruction has been executed.

SPECIFICATION OF MACHINE INSTRUCTIONS

A *machine instruction* consists of one or more bytes. The user specifies how the bits are to be divided into one or more fields. The first field stores the opcode of the instruction. The remaining fields form the operands of the instruction. For example, consider a 2-byte instruction named "load" with fields of length 4, 2, and 10. This instruction has a 4-bit opcode field, a 2-bit operand field, and a 10-bit operand field. When the user wants to execute this instruction in an assembly language program, the user must give the name of the instruction and 2 operands, each consisting of a symbol or number. The operands can optionally be separated by a comma. For the load instruction mentioned above, a legal assembly language instruction is "load 1, 96".

You may optionally designate some of the fields as "don't care" fields. To do so, you surround the field lengths by parentheses in the Machine Instruction dialog. By doing so, you are telling CPU Sim's assembler that those fields are to be ignored. In this case, no operand needs to be specified in assembly language for that field when the instruction is called and instead the assembler just fills the field with 0's.

For example, if the 2-bit field of the load instruction mentioned above is a "don't care" field, then, in an assembly language program, the programmer would type in the name of the instruction and one operand, such as "load 96". In that case, the assembler will put the opcode in the 4-bit first field, the operand 96 in the 10-bit field and will fill

the 2-bit field with 0's.

Note that the operands can have any value that fits in the field in either 2's complement or unsigned integer format. Therefore, the legal values for a 2-bit field are -2 to 3, and the legal values for a 4-bit field are -8 to 15.

Each machine instruction also specifies a list of microinstructions that form the implementation of the machine instruction and that are executed when the machine instruction is to be executed. This microinstruction list is called the *execute* sequence of the instruction. Every execute sequence should contain an End microinstruction, which causes the machine to stop executing the current execution sequence and start executing the fetch sequence, and so causes the start of a new machine cycle.

Parameters

Opcode: a non-negative integer that fits in the first field of the instruction

Field Lengths: a list of 1 or more positive integers whose sum is a multiple of 8.

The integers corresponding to lengths of "don't care" fields are surrounded by parentheses.

Implementation: a list of microinstructions that form the execute sequence of the instruction

SPECIFICATION OF THE MICROINSTRUCTIONS

CPU Sim provides a variety of types of microinstructions that can be used in fetch sequences or execute sequences. All these types are described in this section of the manual.

General Information

Sign Convention

The sign convention used in CPU Sim is 2's complement. All arithmetic calculations and tests are done using this convention. Therefore, whenever a sequence of bits is displayed in decimal, that decimal value corresponds to the two's complement value of the bits. However, when the sequence of bits is displayed in binary or hexadecimal, the bits are treated as an unsigned integer value and so negative signs don't appear in the display.

Floating point operations are not supported by CPU Sim.

Overflow and Carry

If an arithmetic operation results in a value that is too large (positive or negative) as a two's complement number to fit in the destination register, then the machine is in a state of signed overflow. When this occurs, a condition bit can optionally be set to 1. The choice of which condition bit (if any) to set for signed overflow can be specified in the Arithmetic and Increment microinstructions.

Similarly, if an arithmetic addition or subtraction microinstruction yields a value that results in a carry in or out of the destination register, then the machine is in a state of unsigned overflow. When this occurs, a condition bit can optionally be set to 1. The choice of which condition bit (if any) to set for unsigned overflow can be specified in the Arithmetic and Increment microinstructions.

Register Parts

Several microinstructions allow the user to specify parts of registers. In CPU Sim, the bits of a register are indexed from left (the most significant bit) to right (the least significant bit), starting with 0 and ending with one less than the width of the register. For example, if a register has 8 bits, the leftmost bit has index 0 and the rightmost bit has index 7.

The Microinstruction set

Transfer

There are three types of transfers of data allowed between registers and register arrays:

- (a) A register to a register, denoted by "TransferRtoR",
- (b) A register to a register array, denoted by "TransferRtoA",
- (c) A register array to a register, denoted by "TransferAtoR".

In all cases, the user is able to transfer data to or from any set of consecutive bits of a register. The old data in the part of the register receiving the data is lost and is replaced with the new data. The rest of the receiving register is unchanged. That is, if you transfer 4 bits into the first 4 bits of a 16-bit register, then the last 12 bits of the register are unchanged by the transfer.

In cases (b) and (c), the user must specify an index register and a sequence of consecutive bits of that register whose value will form the index of the source or destination register in the array to or from which the data is to be transferred. For example, a TransferRtoA microinstruction might be created to transfer bits from a register *r* to a register array *A* using bits 3-5 of an index register to specify which register in the array is to receive the data. If bits 3-5 of the index register contain the value 0, then the data in register *r* will be transferred to the first register in array *A*.

Parameters

source: a register or register array from which data is to be transferred

source start bit: this number designates the leftmost bit that is to be transferred. It must be a number from 0 to one less than the width of the source register or register array.

destination: a register or register array to which data is to be transferred

destination start bit: this number designates where the leftmost transferred bit is to be placed in the destination register.

number of bits: this specifies how many bits are to be transferred. The source start bit added to the number of bits must yield a sum that is at most the width of the source register or register array. Similarly for the destination register or register array.

index: a register which contains the index of the source or destination register in the array to or from which data is to be transferred. This parameter is only used in transfers of type TransferRtoA or TransferAtoR. The value of the bits in the index register that are used to compute the index are treated as an unsigned integer (that is, they are not treated as a two's complement representation of an integer).

index start bit: this number designates which bit is the leftmost bit of the index register used to compute the index into the array.

index number of bits: this specifies how many consecutive bits of the index register are to be used to compute the index into the array.

For example, if you specify 16-bit registers *r1* and *r2* as the source and destination registers, and if you specify a source start bit of 0, a destination start bit of 12, and the number of bits as 4, then the left-most 4 bits of *r1* will be transferred (more precisely, copied) into the right-most 4 bits of *r2*.

For another example, suppose you specify a TransferRtoA microinstruction with source register *r* and destination register array *A* containing 8 registers. If you specify an index register *i* and index start bit = 0 and index number of bits = 3, then the leftmost 3 bits of *i* are treated as an integer from 0 to 7 that is used to specify which of the 8 registers in array *A* is to receive the data.

Arithmetic

The arithmetic microinstructions use three registers and optionally two condition bits. The contents of the first two registers (the source registers) are added, subtracted, multiplied or divided and the result is placed in the third register (the destination register). If signed overflow occurs, that is, if the result is too big or too small to fit in the destination register using two's complement notation, then the overflow condition bit will be set to 1, if a condition bit is used. Otherwise the value of the overflow condition bit is left unchanged. In the case of overflow, the destination register will contain the rightmost *n* bits of the result in two's complement notation, where *n* is the width of the destination register. If there is a carry in/out, i.e., an unsigned integer overflow, in an addition or subtraction microinstruction, then the carry condition bit is set, if a condition bit is used.

Parameters:

Type: the type of operation, namely addition, subtraction, multiplication, or

division. Note that division is integer division, which truncates the result (i.e., it produces the floor of the result). That is, dividing 7 by 2 yields 3, whereas dividing -7 by 2 yields -4.

Source1, Source2, Destination: the three registers. The operation consists of combining Source1 and Source2 by the desired method (Source1+Source2, Source1-Source2, Source1*Source2, or Source1/Source2) and then placing the result in Destination. The three registers need not all be distinct. For example, the Destination could be the same register as Source1 and/or Source2. Parts of registers may not be specified. The three registers need not have the same width.

Overflow: an overflow condition bit. If not needed then "(none)" should be specified.

Carry: a carry condition bit. Used only in addition and subtraction microinstructions and ignored by the others. If not needed then "(none)" should be specified.

Shift

The shift microinstruction performs a bit-wise shift of the contents of the specified source register to either the left or the right and places the result in the destination register. The user specifies one of the following modes:

Logical shift - The newly emptied bits are given the value 0.

Arithmetic shift - If shifting right, then the leftmost bit is copied into the newly-emptied bits. If shifting left, then the emptied bits on the right are given the value 0.

Cyclic shift - The bits dropped off one end are placed in the emptied bits on the other end.

Parameters:

Type: either LOGICAL, ARITHMETIC, or CYCLIC.

Direction: either LEFT or RIGHT.

Distance: a non-negative integer indicating the number of bit locations to the left or right the values are to be shifted.

Source and Destination: the register whose contents are to be shifted and the register into which the shifted bits are placed. These two registers must have the same width and could be the same register.

Logical

The logical microinstructions perform the bit operations of AND, OR (inclusive or), NOT, NAND (negated AND), NOR (negated OR), or XOR (exclusive or) on the specified registers. In all operations but NOT, three registers are specified, two of which are source registers and the third of which is the destination register. NOT is monadic and so only the first source register is used and the second is ignored.

Parameters:

Type: which logical function (AND, OR, NAND, NOR, XOR, NOT) is to be carried out.

Source1, Source2, and Destination: the three registers to be used. Source1 and Source2 contain the values used in the specified operation and the result is placed in Destination. All registers must have the same width and all bits in the registers are affected by the operation. The three registers need not be distinct, i.e., the two source registers can be the same and a source register can be the same as the destination register. Source2 is ignored if the type of operation is NOT.

Test

The test microinstruction allows jumping to other microinstructions within the current fetch or execute sequence. The test microinstruction compares the value in a part of a register with a specific value. If the comparison succeeds, then a specified number of successive microinstructions in the current fetch or execute sequence are skipped over.

Parameters:

Comparison: the type of comparison to be done:

LT means "less than"

LE means "less than or equal to"

EQ means "equal to"

NE means "not equal to"

GE means "greater than or equal to"

GT means "greater than"

Comparison value: the integer to be compared with the part of the register. The Comparison value must be given in decimal notation.

Omission: an integer (positive, negative or zero) indicating the size of the relative jump. For example, if omission = 0, then no jump is performed. That is, the microinstruction following the test microinstruction is the next microinstruction to be executed. If omission = -1 and the comparison succeeds, then the test microinstruction is repeatedly executed in an infinite loop.

Register: the register whose value is to be tested.

Start: an integer indicating the leftmost bit to be tested in the register. The value must be an integer between 0 and one less than the width of the register.

Number of bits: a non-negative integer indicating the number of bits to be tested. The sum of the start and the number of bits must be at most the width of the register.

N.B. When you test a register or part of a register against a value, you can consider the part of the register to designate a two's-complement integer value or an

unsigned integer value. For example, if you are testing to see whether two bits are “11”, you can either test whether the value stored in the two bits is -1 or test whether the value is 3. Similarly, to test whether one bit of a register is a “1”, you can test to see whether that part of the register contains the value 1 or -1.

Increment

The Increment microinstruction adds an integer constant to the contents of a register. The constant can be positive or negative. The constant is specified in the microinstruction, so if the user needs to increment by different values, then different increment microinstructions need to be used. If signed overflow occurs, that is, if the result is too big or too small to fit in the register using two's complement notation, then the overflow condition bit will be set to 1, if a condition bit is used. Otherwise the value of the overflow condition bit is left unchanged. In the case of overflow, the register will contain the rightmost n bits of the result in two's complement notation, where n is the width of the register.

Typical uses of the increment microinstruction include incrementing the program counter during the fetch sequence or incrementing or decrementing the stack top pointer during pushes and pops.

Parameters:

Register: the register whose contents are to be incremented.

Value: the integer value that will be added to the register contents.

Overflow: an overflow condition bit. If not needed then "(none)" should be specified.

IO

An IO microinstruction simulates an IO port on a CPU. The microinstruction either (a) inputs into a buffer register a value from the user or from a text file or (b) outputs the value of a register to a text file or to the user. An IO microinstruction that outputs a value to the user causes a dialog box to appear with the value displayed. An IO microinstruction that inputs a value from the user causes a dialog box to appear in which the user is asked to type in the value.

To specify the source or destination of the data (either to/from the user or to/from a text file), use the **Options | IO connections...** menu item from the **Execute** menu.

Parameters:

direction: either “input” or “output” indicating whether to get data from the user or from a file (input) or send data to the user or to a file (output).

type: either "integer", "ascii", or "Unicode", indicating the type of data that is to be input or output.

buffer: the register to or from which data is to be input or output.

MemoryAccess

The MemoryAccess microinstruction transfers (copies) data between a register and a RAM. The number of bits transferred is equal to the width of the register.

Parameters:

direction: either “read” or “write” indicating whether data is to go from RAM to a register (read) or from a register to RAM (write).

memory: the RAM to or from which data is to be transferred.

data: the register to or from which data is to be transferred. The width of this register must be divisible by 8.

address: the register that contains the address indicating where in the RAM the data is to be read or written. If multiple bytes are to be read or written, the address is the first (smallest) address of the bytes.

Set

The Set microinstruction sets a sequence of consecutive bits of a register to a specified value.

Parameters:

register: the register whose bits are to be set.

start: the leftmost bit of the register that is to be set. This must be an integer from 0 to one less than the width of the register.

numBits: the number of consecutive bits in the register to be set

value: the value to which the bits are to be set. This value must fit in the designated bits as either an unsigned or signed (two’s complement) decimal. Therefore the value must be from -2^{w-1} to $2^{w-1}-1$, where w is the width of the register.

SetCondBit

The SetCondBit microinstruction sets the contents of a specified condition bit to 0 or 1. Note that this is just a convenience method since you can also set any bit of any register using the Set microinstruction.

Parameters:

bit: the condition bit to be set.

value: the value (0 or 1) to which the condition bit is to be set.

Branch

The branch microinstruction is identical to the Test microinstruction except that it is an unconditional jump. The microinstruction specifies how far to jump relative to the current (branch) microinstruction.

Parameter:

Distance: an integer (positive, negative or zero) indicating the size of the relative jump. For example, if distance = 0, then no jump is performed. That is, the microinstruction following the branch microinstruction is the next microinstruction to be executed, as normally happens with other microinstructions. If distance = -1, then the branch microinstruction is repeatedly executed in an infinite loop.

Decode

The decode microinstruction is usually the last microinstruction in the fetch sequence. It is used to simulate a decoder. An instruction register (*ir*) must be specified. CPU Sim will decode the instruction in the *ir*, i.e., determine which machine instruction is to be executed, using the following algorithm. CPU Sim initially inspects the leftmost bit of the *ir*. If there is a machine instruction with an opcode of one bit that matches the leftmost bit of the *ir*, then that machine instruction is chosen as the decoded instruction and the rest of the bits in the *ir* form its operands. If there is no such machine instruction, then the leftmost two bits of the *ir* are inspected and CPU Sim looks for a machine instruction with a 2-bit opcode that matches those two bits. CPU Sim continues inspecting more and more bits of the *ir* until it finds a machine instruction whose opcode matches those bits. Therefore, if you have one instruction with a one-bit opcode of 0 and another instruction with a two-bit opcode of 01, then the second instruction will never be decoded and executed. Once a machine instruction has been decoded, execution of that instruction begins, starting with the first microinstruction in its execute sequence.

Parameters:

ir: the register that contains the current instruction to be executed.

End

The end microinstruction is used to designate the end of an execute sequence and therefore the end of a machine cycle. It causes the current execute sequence to cease executing and the fetch sequence to begin executing, starting with the first microinstruction in that sequence.