A Graphical Interface for CHIP

Lorenzo Alvisi
The University of Texas at Austin
Department of Computer Sciences
Austin, TX

Fred B. Schneider
Cornell University
Department of Computer Science
Ithaca, NY

6 June 1996

Abstract

CHIP (Cornell Hypothetical Instructional Processor) [BBDS83] is a computer system designed as an educational tool for teaching undergraduate courses in operating system and machine architecture. This document describes CHIP's graphical interface and covers in a tutorial how the interface is used to debug and execute CHIP programs.

A Graphical Interface for CHIP

Lorenzo Alvisi
The University of Texas at Austin
Department of Computer Sciences
Austin, TX

Fred B. Schneider
Cornell University
Department of Computer Science
Ithaca, NY

6 June 1996

Abstract

CHIP (Cornell Hypothetical Instructional Processor) is a computer system designed for use in teaching undergraduate courses in operating system and machine architecture. This document describes CHIP's graphical interface and contains a tutorial describing how the interface is used to debug and execute CHIP programs.

1 Introduction

CHIP (Cornell Hypothetical Instructional Processor) [BBDS83] is a computer system designed for use in teaching undergraduate courses in operating system and machine architecture. This document describes CHIP's graphical interface and includes a tutorial describing how this interface is used to debug and execute CHIP programs. The document is organized as follows. Section 2 describes how to compile and generate assembler listings for programs targeted for CHIP. Section 3 describes the CHIP console, which is the primary means a user interacts with CHIP. Section 4 contains the tutorial.

2 Compiling C programs

The C compiler for CHIP, called `pcc`, is a slightly modified PDP-11 C compiler. The command line syntax for this compiler is very similar to that of the compiler for the PDP-11. Consult the manual page for details by typing `man pcc`. There are several differences between a C program running on the PDP-11 and one running on CHIP. In CHIP:

- the load module produced by `pcc` (the `a.out` file) has the program origin at address 1024 (decimal) rather than 0;
- the symbolic name `start` is associated with address 1024;
- the program prologue causes a transfer of control to the function `main()` through the `INPRG` instruction;
- none of the library function that request UNIX system services (e.g. `printf()`, `scanf()`, `open()`, etc.) are supported.

3 The CHIP Console

The CHIP console offers tools that assist the user in the development of CHIP programs. The tools serve two main purposes:

1. They provide comprehensive control over the program execution.

   The user can stop execution either by setting breakpoints at arbitrary locations in the program or by setting watchpoints for arbitrary locations in memory. In addition, the user can set the speed at which the processor executes, including single stepping.

2. They provide a facility to inspect and modify the internal state of CHIP.

   The user can read or write any of CHIP's registers, as well as any region of CHIP's memory. In addition, registers and memory locations can be *traced*, allowing the user to see how their contents change as CHIP executes.

3.1 Running CHIP

The console is invoked whenever the CHIP simulator is started. To start the CHIP simulator, type the UNIX command line

`chip [-s] [`*file*`]`

where optional argument *file* is an object file in PDP-11 format produced by the `pcc` command. If *file* is omitted, then file `a.out` in the current directory is used. The `-s` flag causes the symbol table for the object file to be written to file *file*`.symbols` in the current directory.

As part of its initialization phase, the CHIP simulator loads the object file (*file* or `a.out`) into CHIP's physical memory starting at location 1024. The CHIP processor state is initialized as follows:

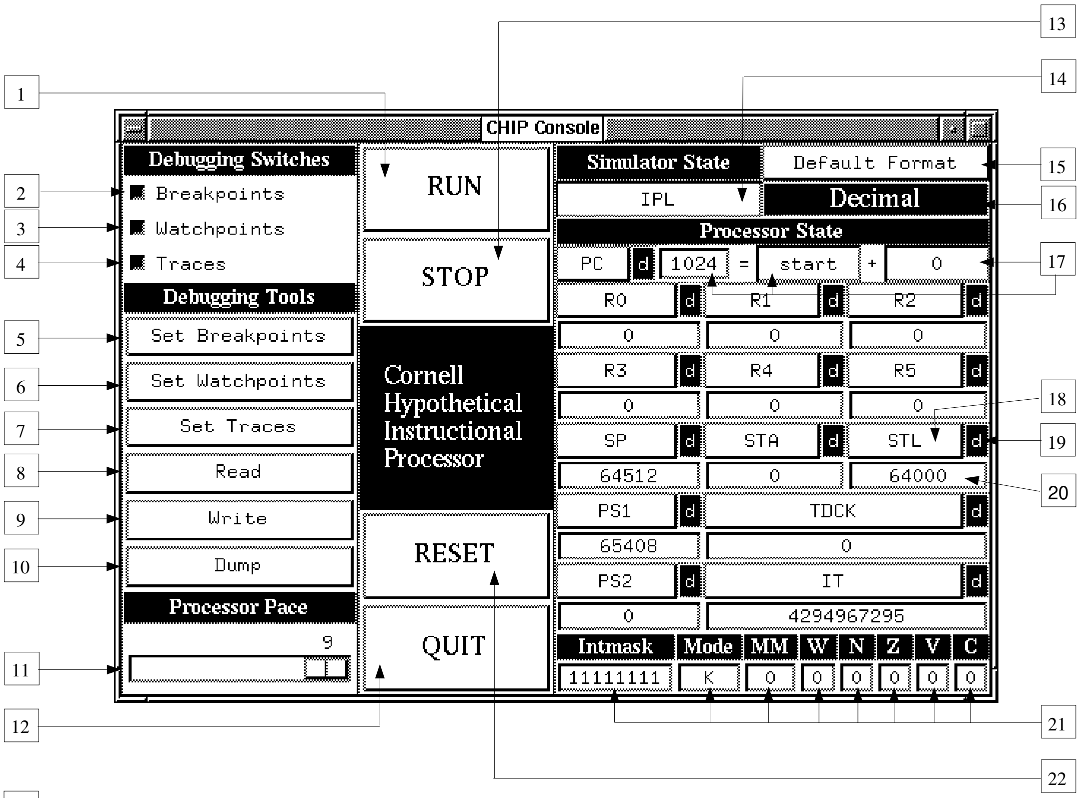
- Program Counter (PC) points to the first executable instruction, i.e. $PC = 1024$.
- Stack Pointer (SP) points to the first word beyond the end of available physical memory.
- Stack Limit Register (STL) points to the first word of the last page of available physical memory.
- Processor Status Word 1 ($PSW1$) is set so that all interrupts are masked, memory mapping is not in effect and the machine is in KERNEL mode.
- All other registers are set to 0.

When initialization is complete, a small window will appear. The window contains a button, labeled **Chip Hypothetical Instructional Processor** — click on this button to open the console main window.

3.2 The Console Windows

Figure 1 shows **CHIP Console**, the console main window, as it appears upon invocation. Refer to the sections indicated in boldface in Figure 1 for details about each part.

In addition to the console main window, the CHIP user interface defines several other console windows, called *console tool windows*, that are associated with tools offered by CHIP. Figures 3, 6, 7 and 9 through 13 show the console tool windows as they initially appear upon invocation. Console tool windows are described in Section 3.4.



1 RUN button.
Press to start execution. **2.6**

2 Breakpoints switch.
Set to enable/disable breakpoints. **2.3.1**

3 Watchpoints switch.
Set to enable/disable watchpoints. **2.3.2**

4 Traces switch.
Set to enable/disable traces. **2.3.3**

5 Set Breakpoints button. **2.4.1**

6 Set Watchpoints button. **2.4.3**

7 Set Traces button. **2.4.4**

8 Read button.
Press to read from memory. **2.4.5**

9 Write button.
Press to write to memory. **2.4.6**

10 Dump Button.
Press to save CHIP state to a file. **2.4.7**

11 Processor Pace Slider.
Set to determine the processor speed. **2.5**

12 QUIT button.
Press to leave CHIP. **2.9**

13 STOP button.
Press to suspend execution. **2.6**

14 Simulator State. **2.10.**

15 Default Format button.
Press to set the default display format. **2.11**

16 Default display format. **2.11**

17 Program Counter value. **2.12.2**

18 Register display format button. **2.12.1**

19 Register's current display format. **2.12.1**

20 Register's current value. **2.12.1**

21 Binary representation of register PS1. **2.12.3**

22 RESET button.
Press to restore CHIP to startup state. **2.10**

Figure 1: The Chip Console window at startup.

3.2.1 Using the Console

To modify the state of CHIP, the user interacts with the console in four ways:

1. Pressing a console button.
2. Selecting an entry of a console menu.
3. Typing a value into a console entry box.
4. Dragging a console slider.

3.2.2 Using Console Buttons

The quickest way to identify the console buttons is to move the mouse pointer across a console window. When the mouse pointer moves over a console button, the button changes color to indicate that it has become *active*. If mouse button 1 (i.e. the leftmost button) is pressed when a button is active, then the console button is said to be *pressed* and the action that is associated with that button is performed.

3.2.3 Using Console Menus

The console defines special buttons, called *menubuttons*, that allow entries from a menu to be selected. If mouse button 1 is pressed when a menubutton is active, the corresponding menu appears underneath the menubutton. To select an entry from such a menu, keep mouse button 1 pressed and move the mouse pointer down the menu until the desired entry is highlighted; then, release mouse button 1.

3.2.4 Using the Console Entry Boxes

Entry boxes allow a user to inspect and modify the value of a CHIP simulator variable.

3.2.4.1 The Display Format

Each entry box displays and accept values according to a *display format*. The display format for an entry box is set using the menubutton that is located just above the entry box. Typically, the display format is changed by the user:

- to declare the format of a value to be entered in the entry box;
- to convert the value displayed in the entry box to a new format.

Acceptable formats are:

Symbolic: The symbolic name of a global variable defined in the loaded program. The format is available only in entry boxes used to specify a region of CHIP's memory (see Section 3.4.3). It is possible to convert a value entered as **Symbolic** to any other format. The converse, however, is not true, because the mapping from addresses to symbolic names is not unique. For instance, if symbols `okbuf` and `start` correspond to addresses 4830 and 1024 respectively, address 7052 could be interpreted both as `okbuf` + 2222 and as `start` + 6028.

Binary: A binary integer (i.e. bits). It is possible to convert a value entered as **Binary** to any other format except **Symbolic**.

Octal: An octal integer. The value is displayed prefixed by a '0', but it is not necessary — although it is allowed — to type the leading '0' when entering a value in octal format. It is possible to convert a value entered as **Octal** to any other format except **Symbolic**.

Decimal: A decimal integer. It is possible to convert a value entered as **Decimal** to any other format except **Symbolic**.

Hex: A hexadecimal integer. The value is displayed prefixed by a leading '0x', but it is not necessary — although it is allowed — to type the leading '0x' when entering a value in hexadecimal format. It is possible to convert a value entered as **Hex** to any other format except **Symbolic**.

ASCII: An ASCII string. Since there is a corresponding ASCII character for each 7-bit pattern, any ASCII character can be stored in an 8-bit byte. However, not all the 8-bit patterns that can be stored in a byte correspond to ASCII characters. Therefore, a two-character string can be stored in a CHIP word or in a 16-bit register, and similarly a four-character string can be stored in a 32-bit register, such as IT or TDCK. However, not all 16 or 32 bit patterns can be represented as strings of ASCII characters. CHIP displays and allows the user to enter only strings composed of ASCII characters that are printable. When converting to **ASCII** from another format, 8-bit patterns corresponding to non-printable characters are represented by the character '@'. It is possible to convert a value entered as **ASCII** to any other format except **Symbolic**.

The dark label to the right of the menubutton indicates the current display format of the entry box. The label will indicate one the following:

s for **Symbolic**

b for **Binary**

o for **Octal**

d for **Decimal**

x for **Hexadecimal**

a for **ASCII**

Attempts to enter values in a format different from the current display format will generate an error messages.

Figure 2 shows the entry box, menubutton, and display label for register *SP* as they initially appear in the console main window. Notice that the display format for *SP* is given as *x* to denote **Hex**.

3.2.4.1 Editing an Entry Box

To enter a value in an entry box, move the mouse pointer over the entry box and press mouse button 1. A blinking vertical-cursor will appear, indicating the position in the entry box where the new value should be typed. Once you are finished typing in the value, type `<Return>`.

Figure 2: The entry box, menubutton, and display label of register *SP*.

To delete characters inside an entry box, move the vertical cursor to the immediate right of the characters to be deleted and type `<Control-h>`. Alternatively, select the characters to be deleted by dragging the mouse pointer over them with mouse button 1 pressed; then press the `<Delete>` key.

3.2.5 Using Console Sliders

A *slider* is a rectangular object that can be *dragged* within a given horizontal or vertical guide. To drag a slider, move the mouse pointer over the slider and press mouse button 1. Then, keeping mouse button 1 pressed, move the mouse in a direction allowed by the guide.

3.3 Debugging Switches

Set these switches to enable or disable the various console tools. Each switch is a flat button accompanied by a rectangular *selector box* to its left. When the switch is on, the selector box appears dark and sunken; when the switch is off, the selector box appears empty. For example, the selector boxes for the **Breakpoints**, **Watchpoints** and **Traces** switches in Figure 1 are all on.

3.3.1 Breakpoints

If the **Breakpoints** switch is on, then all defined breakpoints are enabled. If the **Breakpoints** switch is off, then defined breakpoints will remain on the breakpoint list, but encountering a breakpoint during execution does not cause CHIP to stop. Default: on.

3.3.2 Watchpoints

If the **Watchpoints** switch is on, all defined watchpoints are enabled. If the **Watchpoints** switch is off, then defined watchpoints will remain on the watchpoint list, but referencing a watchpoint during execution does not cause CHIP to stop. Default: on.

3.3.3 Traces

If the **Traces** switch is on, then the contents of the following are displayed as they change during program execution:

- all CHIP registers that have their *trace bit* set (see Section 3.12);
- all memory locations selected with the **Set Traces** tool (see Section 3.4.5) that are associated with non-iconified windows.

If the **Traces** switch is off, changes to registers and traced memory locations are not displayed until CHIP stops. Switching **Traces** off can substantially speed up program execution, since it relieves the simulator from the task of updating the display constantly. Default: on.

3.4 Debugging Tools

The buttons in this part of the console are associated with tools that allow control of program execution. The tools also allow the content of CHIP's memory to be inspected and modified.

3.4.1 The Console Tool Windows

Each of the debugging tools is associated with one or more console tool windows. These windows serve two main purposes:

1. They allow the user to specify how the tools they are associated with should be used.
2. They allow the user to inspect the result of using the tools.

To invoke the console tool window associated with tool a *t* :

1. Press the *t* button in the console main window.

   When the *t* button is pressed, the associated console tool window appears on the screen and the *t* button becomes disabled.

To dismiss the console tool window associated with a tool *t* :

1. Press the **Dismiss** button in the console tool window or type `<Control-C>` while the mouse pointer is on the tool window.

   The console tool window will disappear from the screen, and the *t* button in the console main window is re-enabled. Dismissing a console tool window is not equivalent to iconifying one:

   - If a console tool window is dismissed, then the corresponding button in the console main window is re-enabled.
   - If a console tool window is iconified, then the corresponding button in the console main window is left disabled.

3.4.2 Set Breakpoints

Press this button to insert one or more breakpoints in the *breakpoint list*.

A *breakpoint* is a marked instruction in the program. It is specified by giving an optional byte offset to the symbolic name of a function defined in the program. Whenever such a marked instruction is about to be executed, CHIP stops. The user can inspect the state of CHIP at that time. When the **Set Breakpoints** button is pressed, a new **Breakpoint List** tool window pops up. Figure 3 shows the window as it first appears.

To add an entry to the breakpoint list:

1. Enter the symbolic name of a program function.

   Type the name of the function in the entry below the label **Function Name** and press `<Return>`. This action will enable the **Add** button in the **Breakpoint List** window.

2. Set the sign of the offset (optional).

   Press the + flat button if the sign of the offset is positive; otherwise, press the − flat button. The current selection for the sign of the offset is indicated with a dark and sunken diamond-shaped selector to the left of the corresponding flat button.

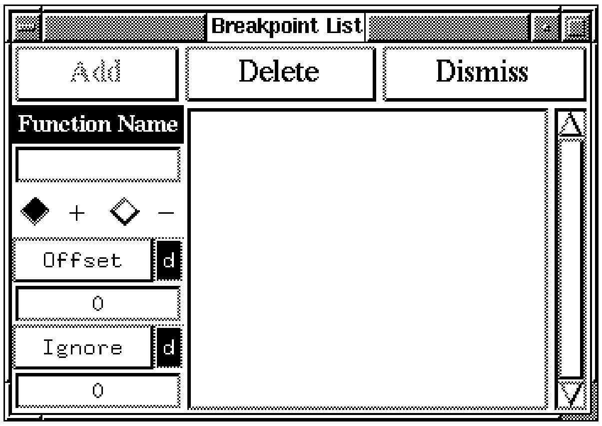


Figure 3: The **Breakpoint List** window at startup...

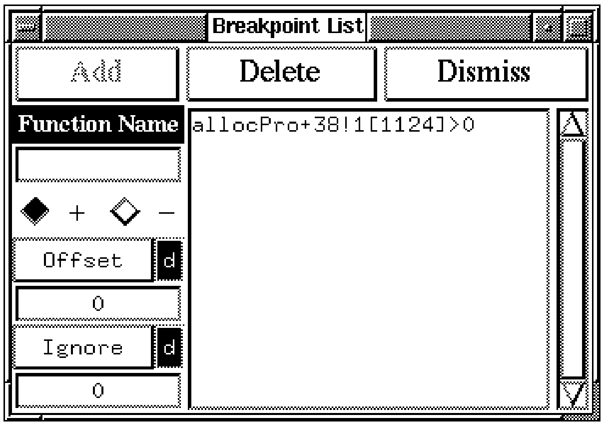


Figure 4: ...and after a breakpoint has been inserted.

3. Set the desired input format for the offset (optional).

   The dark label to the right of the **Offset** menubutton shows the current input format. To change format, press the **Offset** menubutton and select the menu entry corresponding to the desired format (see also Section 3.2.3 and 3.2.4).

4. Enter a byte offset (optional).

   Type the *offset* value in the entry box below the menubutton **Offset** and press `<Return>` (see also Section 3.2.4).

5. Set the desired input format for the **Ignore** value (optional).

   The dark label to the right of the **Ignore** menubutton shows the current input format. To change format, press the **Ignore** menubutton and select the menu entry corresponding to the desired format (see also Sections 3.2.3 and 3.2.4).

6. Enter the **Ignore** value (optional).

   Type the *ignore* value in the entry box below the menubutton **Ignore**, and press `<Return>` (see also Section 3.2.4). CHIP will not stop at the breakpoint for the first *ignore* times the breakpoint is hit. The default for *ignore* is 0.

7. Press the **Add** button.

   When the **Add** button is pressed, the breakpoint is added to the breakpoint list. The format of the entry is:

$$\textit{function\_name sign offset!ignore}[\textit{address}] > \textit{hitcount}$$

   where *hitcount* is the number of times the breakpoint has been encountered. Figure 4 shows window **Breakpoint List** right after a breakpoint has been set at the instruction located 38 bytes after the address associated with the symbolic function name `allocPro`. The decimal number displayed between square brackets is the address for the breakpoint. The decimal number after the '!' character indicates the number of times the breakpoint will be ignored. Thus, execution will stop the second time this breakpoint is encountered.

To delete an entry from the breakpoint list:

1. Select the entry.

Click on the entry with mouse button 1: the selected entry will be highlighted. Multiple entries can be selected by dragging the mouse while keeping mouse button 1 pressed.

2. Press the **Delete** button.

As a shortcut, clicking mouse button 1 twice on an entry in the breakpoint list causes that entry to be deleted.

To disable breakpoints temporarily:

1. Switch off the **Breakpoints** switch (see Section 3.3.1).
   Note that when the **Breakpoints** switch is on, all defined breakpoints are enabled even if the breakpoint list has been dismissed or iconified.

When a breakpoint is encountered:

If the breakpoint's *hitcount* is greater than the breakpoint's *ignore*, then:

1. Execution stops.
2. The corresponding entry in the breakpoint list is highlighted and *hitcount* is updated for all the entries in the breakpoint list.
3. The simulator state (see Section 3.10) is set to *BKPT*.
4. The displayed value of the registers and of the traced memory locations (see Section 3.4.5) is updated.

If the breakpoint's *hitcount* is less than or equal to the breakpoint's *ignore*, then CHIP continues execution undisturbed.

3.4.3 Specifying a region of memory

Set Watchpoints, **Set Traces**, **Read**, and **Write** each require the user to specify a region in memory. Figure 5 shows the dialog box used to specify a region of CHIP's memory. A region consists of a block of memory words defined by a lower and an upper-bound. The address of the lower bound is computed by adding to a low-base address the value of an optional offset.

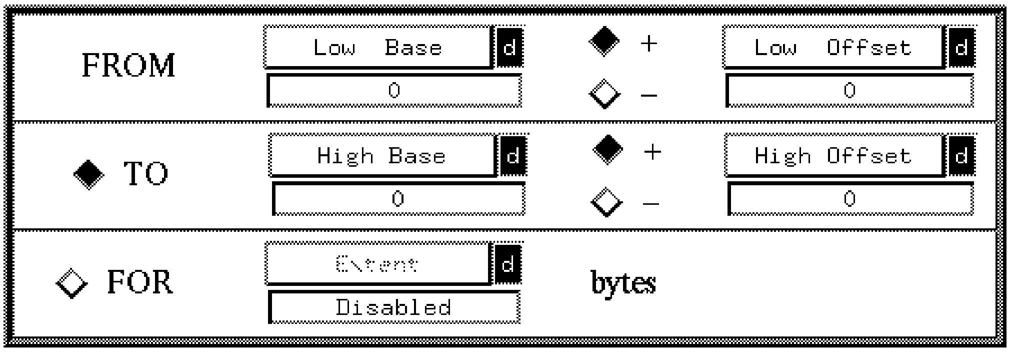


Figure 5: The dialog box used to specify intervals in CHIP memory.

The upper-bound can be specified in one of two ways:

1. By adding to a high-base address the value of an optional offset. — or —

2. By adding to the lower bound the extent of the desired interval.

To set the address of the lower bound:

1. Set the desired input format for the low-base address.

   The dark label to the right of the **Low Base** menubutton shows the current input format (see Section 3.2.4). To change format, press the menubutton **Low Base** and select the menu entry corresponding to the desired format (see also Sections 3.2.3 and 3.2.4).

2. Enter the low-base address.

   Type the value of the low-base address in the entry box below the menubutton **Low Base** and press `<Return>` (see also Section 3.2.4).

3. Set the sign of the offset (optional).

   Press the + flat button if the sign of the offset is positive; otherwise, press the − flat button. The current selection for the sign of the offset is indicated with a dark and sunken diamond-shaped selector to the left of the corresponding flat button.

4. Set the desired input format for the low offset.

   The dark label to the right of the **Low Offset** menubutton shows the current input format. To change format, press the menubutton **Low Offset** and select the menu entry corresponding to the desired format (see also Sections 3.2.3 and 3.2.4).

5. Enter the low offset.

   Type the value of the low offset in the entry box below the menubutton **Low Base** and press `<Return>` (see also Section 3.2.4).

To set the address of the upper-bound:

1. Select the way to determine the upper-bound.

   Press the **TO** flat button to enter the upper-bound as a high-base address plus an optional offset; alternatively, press the **FOR** flat button to enter the extent of the interval in memory from the lower bound.

2. If **TO** was pressed:

   (a) Set the desired input format for the high-base address.
   The dark label to the right of the **High Base** menubutton shows the current input format (see Section 3.2.4). To change format, press the menubutton **High Base** and select the menu entry corresponding to the desired format (see also Sections 3.2.3 and 3.2.4).

   (b) Enter the high-base address.
   Type the value of the low-base address in the entry box below the menubutton **High Base** and press `<Return>` (see also Section 3.2.4).

   (c) Set the sign of the offset (optional).
   Press the + flat button if the sign of the offset is positive; otherwise, press the − flat button otherwise. The current selection for the sign of the offset is indicated with a dark and sunken diamond-shaped selector to the left of the corresponding flat button.

(d) Set the desired input format for the high offset.
The dark label to the right of the **High Offset** menubutton shows the current input format. To change format, press the menubutton **High Offset** and select the menu entry corresponding to the desired format (see also Sections 3.2.3 and 3.2.4).

(e) Enter the low offset.
Type the value of the high offset in the entry box below the menubutton **Low Base** and press `<Return>` (see also Section 3.2.4).

If **FOR** was pressed:

(a) Set the desired format for entering the extent of the region.
The dark label to the right of the **Extent** menubutton shows the current input format (see Section 3.2.4). To change format, press the menubutton **Extent** and select the menu entry corresponding to the desired format (see also Sections 3.2.3 and 3.2.4).

(b) Enter the extent of the region.
Type the extent of the region in the entry box below the menubutton **Extent** and press `<Return>` (see also Section 3.2.4).

Note that if the **TO** option has been selected, then the entry box **Extent** will contain the value *Disabled*. Similarly, if the **FOR** option has been selected, then entry boxes **High Base** and **High Offset** will contain the value *Disabled*. Note also that if the **TO** option has been selected, then the values entered in the **Low Base** and **Low Offset** entry boxes are automatically copied to the **High Base** and **High Offset** entry boxes, respectively. This makes it easier to select an interval consisting of a single memory word. It also diminishes the chances of mistakenly entering an upper-bound that is lower than the lower bound.

3.4.4 Set Watchpoints

Press this button to insert watchpoints in the *watchpoint list*.

A *watchpoint* is a marked address in memory. Whenever a watchpoint is about to be referenced, CHIP stops. The user can inspect the state of CHIP at that time. When the **Set Watchpoints** button is pressed, two new tool windows — **Set Breakpoints** and **Watchpoint List** — pop up. Figures 6 and 7 show the windows as they first appear upon invocation.

To add an entry to the watchpoint list:

1. Enter the memory region to be watched.
   Enter the desired region through the dialog box (see Section 3.4.3) in the **Set Watchpoints** window.
2. Select the kind of memory reference that should stop execution.
   Press the **Reference** menubutton in the **Set Watchpoints** window to select one of the following:

   Read: Stops execution whenever an instruction is about to read (but not write) a value stored at one of the addresses in the selected region

   Write: Stops execution whenever an instruction is about to write (but not read) a value stored at one of the addresses in the selected region.

   R/W: Stops execution whenever an instruction is about to read and/or write a value stored at one of the addresses in the selected region.

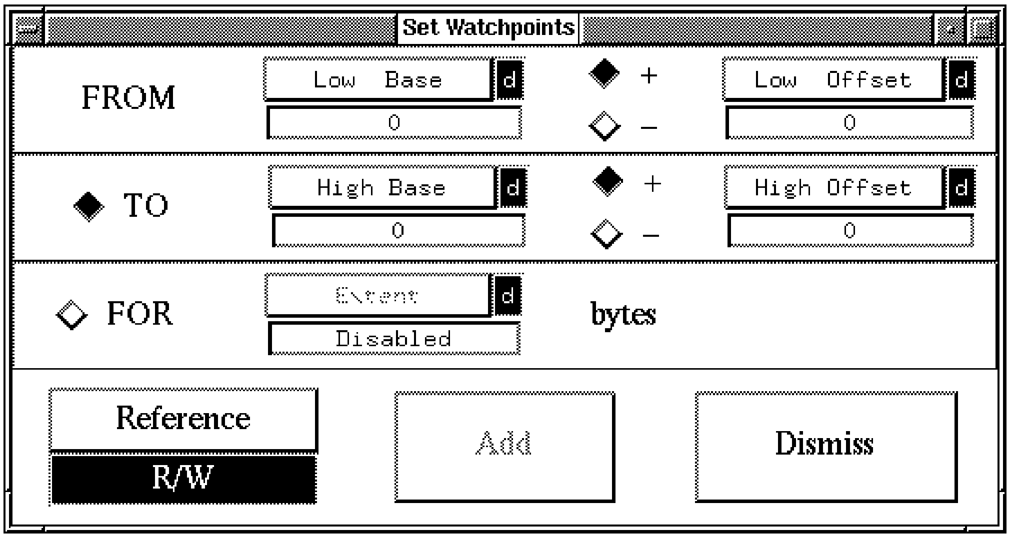


Figure 6: The **Set Watchpoint** window at startup.

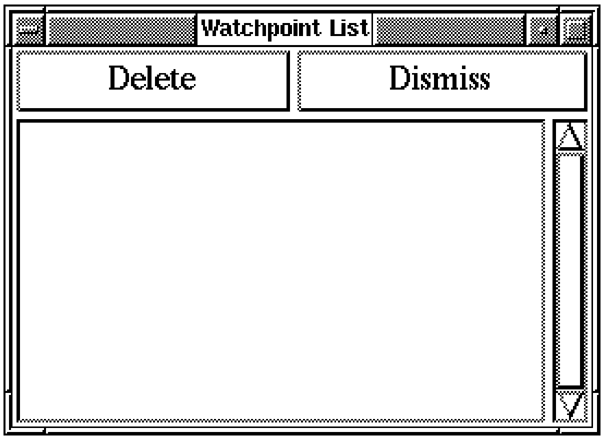


Figure 7: The **Watchpoint List** window at startup...

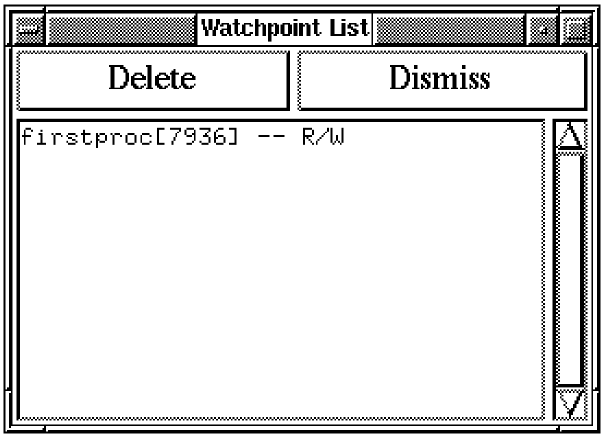


Figure 8: ...and after a watchpoint has been inserted.

3. Press the **Add** button in the **Set Watchpoints** window.

   When the button is pressed, the watchpoint is added to the list in the **Watchpoint List** window. An address can be listed only once in the watchpoint list.

Figure 8 shows the **Watchpoint List** window after a watchpoint has been set at the address corresponding to symbolic name `firstproc`. The decimal number displayed between square brackets is the address where the watchpoint has been set. Execution will stop whenever an instruction is about to read and/or write location 7936.

To delete an entry from the watchpoint list:

1. Select the entry.

   Click on the entry with mouse button 1: the selected entry will be highlighted. Multiple entries can be selected by dragging the mouse while keeping mouse button 1 pressed.

2. Press the **Delete** button in the **Watchpoint List** window.

As a shortcut, click twice with mouse button 1 on an entry in the watchpoint list to be deleted.

To disable watchpoints temporarily:

1. Switch off the **Watchpoints** switch (see Section 3.3.2).
   Note that when the **Watchpoints** switch is on, all defined watchpoints are enabled even if the watchpoint list has been dismissed or iconified.

When a watchpoint is encountered:

1. Execution stops.
2. The corresponding entry in the watchpoint list is highlighted.
3. The simulator state (see Section 3.10) is set to *WATCH*.
4. If the **Breakpoint List** window is active, then the displayed value of *hitcount* for all the entries in the breakpoint list is updated.
5. The displayed value of the registers and of the traced memory locations (see Section 3.4.5) is updated.

3.4.5 Set Traces

Press this button to observe the contents of a region in memory change during execution.

When the **Set Traces** button is pressed the **Trace Memory** tool window appears. Figure 9 shows the window as it first appears upon invocation.

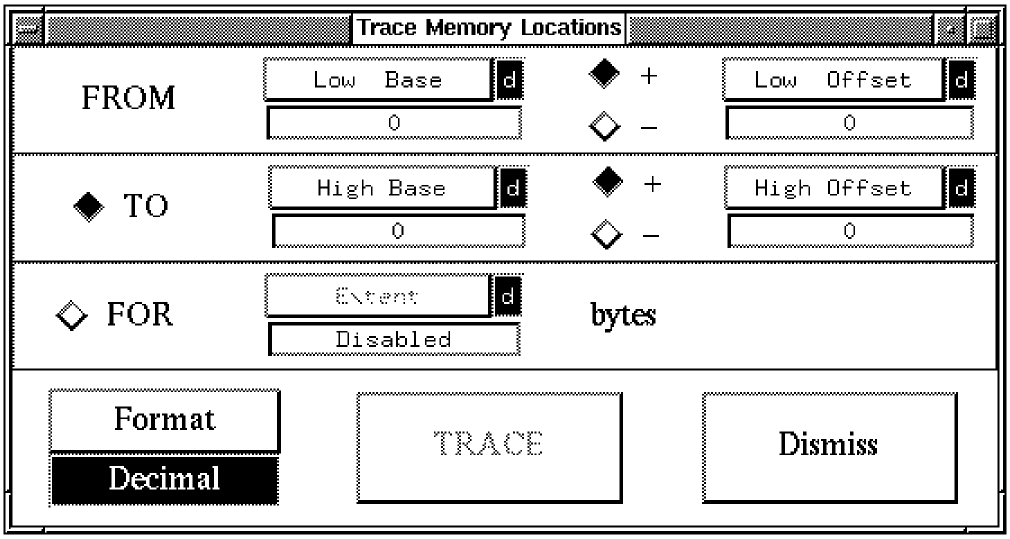


Figure 9: The **Trace Memory** window at startup.

To trace a region in memory:

1. Enter the memory region to be traced.

   Type the desired region through the dialog box (see Section 3.4.3) in the **Trace Memory** window.

2. Select the display format of the data.

   The dark label below the **Format** menubutton shows the current format in which the data to be traced are going to be displayed. To change format, press the **Format** menubutton and select the menu entry corresponding to the desired format (see also Sections 3.2.3 and 3.2.4).

3. Press the **TRACE** button in the **Trace Memory** window.

   A new tool window appears, displaying the contents of the selected memory region. Figure 10 shows a snapshot of the window that traces the content of a buffer. In this example, a variable `okbuf` is being used to record the actions taken by the program. As execution proceeds, the displayed values are updated to reflect changes that take place in memory. To temporarily suspend updating the displayed values, switch off the **Traces** switch (see Section 3.3.3).

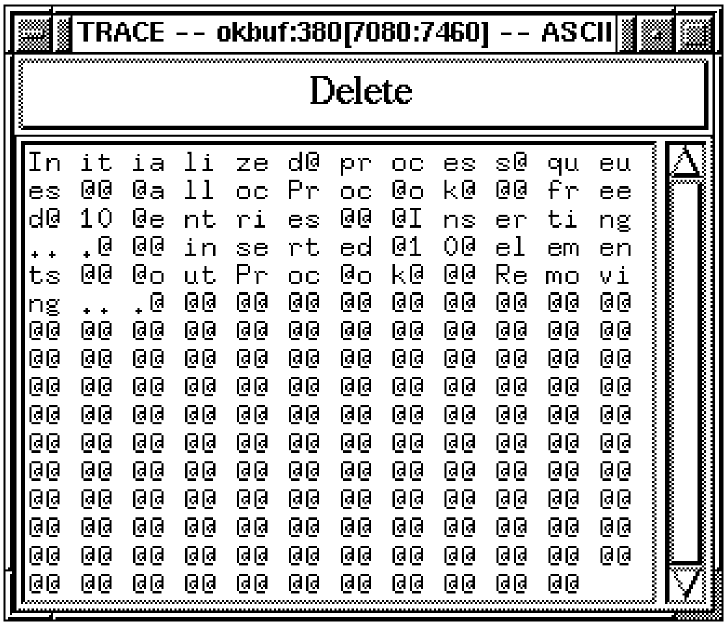


Figure 10: A window tracing the contents of buffer `okbuf`.

To dismiss the Trace Memory window:

1. Press the **Dismiss** button in the **Trace Memory** window.

To delete a window displaying traced data:

1. Press the **Delete** button in the window to be deleted.

3.4.6 Read

Press this button to display the contents of a memory region.

When the **Read** button is pressed, a new tool window — **Read from Memory** — appears. Figure 11 shows the window as it first appears upon invocation.

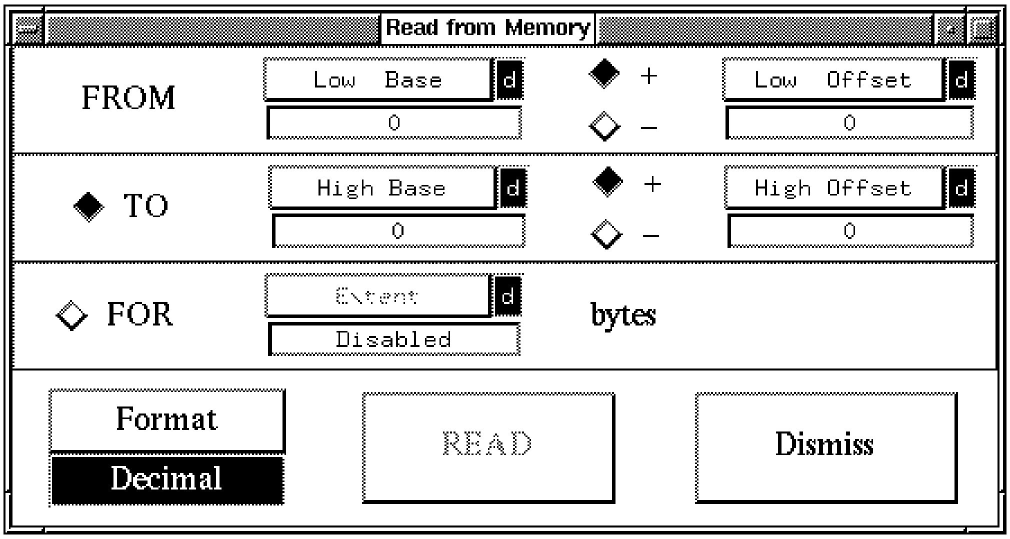


Figure 11: The **Read from Memory** window at startup.

To read a region in memory:

1. Enter the memory region to be read.
   Enter the desired region through the dialog box (see Section 3.4.3) in the **Read from Memory** window.
2. Select the display format of the data.
   The dark label below the **Format** menubutton shows the current format in which the data to be read are going to be displayed. To change format, press the **Format** menubutton and select the menu entry corresponding to the desired format (see also Sections 3.2.3 and 3.2.4).
3. Press the **READ** button in the **Read from Memory** window.
   A new tool window pops up, displaying the contents of the selected memory region. The new window is identical to the one invoked when the **TRACE** button is pressed in the **Trace Memory** window (see 3.4.5 and Figure 10).

To delete a window displaying read data:

1. Press the **Delete** button in the window to be deleted.

3.4.7 Write

Press this button to write a memory region.

When the **Write** button is pressed, a new tool window — **Write to Memory**— appears. Figure 12 shows the window as it first appears upon invocation. The value to be written in the memory region can be written either as an integer (specified as **Binary**, **Octal**, **Decimal**, or **Hex**) or as an **ASCII** string. If an integer is given, then its value is written to each of the words in the memory region. If an **ASCII** string is given, successive bytes are assigned characters from the string until either the string or the region is exhausted.

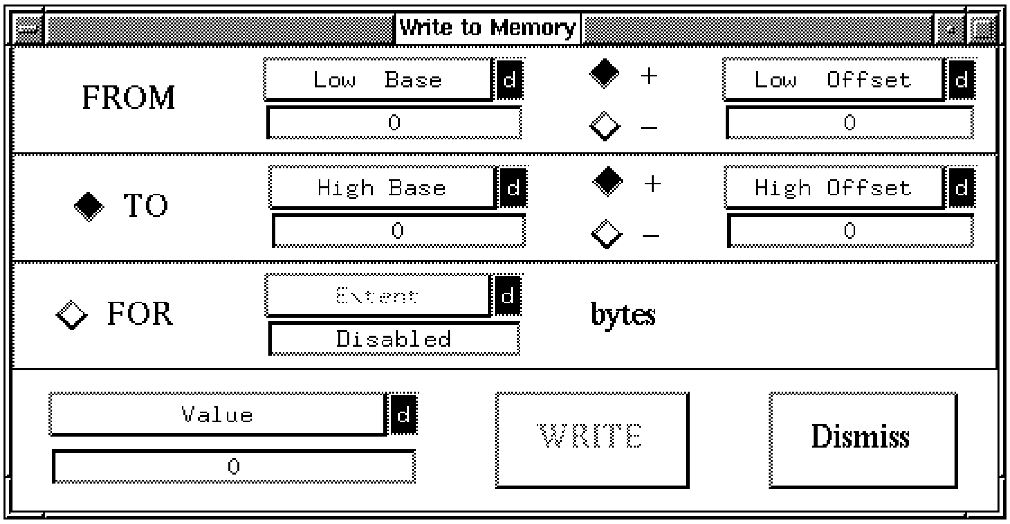


Figure 12: The **Write to Memory** window at startup.

To write to an region in memory:

1. Enter the memory region to be written.

   Enter the desired region through the dialog box (see Section 3.4.3) in the **Write to Memory** window.

2. Select the display format of the data.

   The dark label to the right of the **Value** menubutton shows the current format. To change format, press the **Format** menubutton and select the menu entry corresponding to the desired format (see also Sections 3.2.3 and 3.2.4). Notice that the **Value** menubutton does *not* support automatic conversion of the data displayed in its entry box when a new format is selected.

3. Press the **WRITE** button in the **Write to Memory** window.

3.4.8 Dump

Press this button to save in a file information about the current state of the simulation.

The following information is saved:

- the content of all the registers.
- the content of all regions of memory that are currently being traced.
- the content of any window that displays the result of a **READ** in a memory region.

When the **Dump** button is pressed, a new window — **Select File** — appears. Figure 13 shows the **Select File** window: in addition to an entry box for the name of the selected file, the window displays the contents of the current directory. The dark label underneath the **File** entry box shows the path corresponding to the current selection.

To save the state of the simulation:

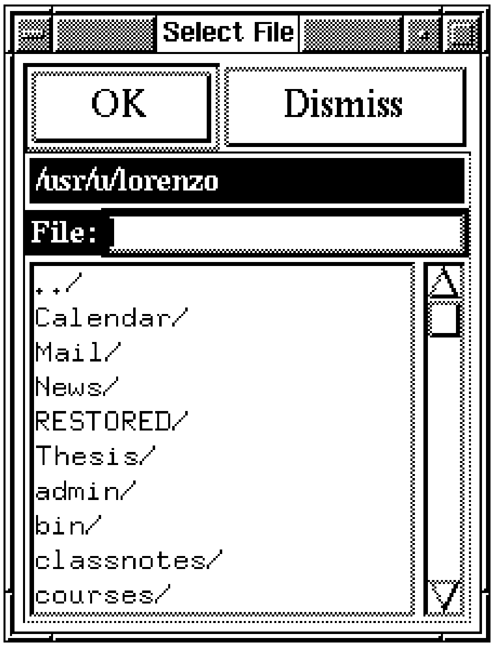


Figure 13: The **Select File** window.

1. Select the name of the file where the information will be saved.

   Type the file name in the entry box to the right of the **File** label, or, if the file already exists, then press mouse button 1 once on the entry corresponding to the file name in the directory listing.
2. Press the **OK** button or the `<Return>` key

   As a shortcut, if the file already exists, then the result of following steps 1 and 2 can be achieved by pressing mouse button 1 twice on the entry corresponding to the file name in the directory listing.

3.5 Processor Pace

Drag this slider to set the pace of the processor.
The processor pace can be set to any integer value between 0 and 9. Setting the pace to 0 causes CHIP to execute in single steps; setting the pace to 9 allows CHIP to execute at full speed. The current pace value is displayed as a decimal integer above the slider.

3.6 RUN

Press this button to start or resume execution.
When the **RUN** button is pressed and the processor pace is set to 0, the simulator state (see Section 3.10) is set to *SINGLE STEP*; otherwise, it is set to *RUN*.

3.7 STOP

Press this button to temporarily stop execution.
When the **STOP** button is pressed the simulator state (see Section 3.10) is set to *STOP*.

3.8 RESET

Press this button to restart the loaded program from its initial state.
When the **RESET** button is pressed, the simulator is re-initialized (see Section 3.1). The simulator state (see Section 3.10) is set to *IPL*.

3.9 QUIT

Press this button to exit the CHIP simulator.

3.10 Simulator State

The value displayed below the label **Simulator State** can be one of the following:

IPL: This is displayed at startup and whenever the **RESET** button (see Section 3.8) is pressed.

WAIT: This is displayed when the *W* bit of *PSW1* (see Section 3.12.3) is set, there are no unmasked interrupt requests, and there are no possible sources of unmasked future interrupts.

BKPT: This is displayed when a breakpoint (see Section 3.4.2) is encountered.

WATCH: This is displayed when a memory location on which a watchpoint is established (see Section 3.4.4) is about to be referenced.

SINGLE STEP: This is displayed when CHIP executes with the the processor pace set to zero (see Section 3.5). When in this state, CHIP executes in single steps; that is, execution suspends after each instruction.

HALT: This is displayed when a `HALT` instruction is executed.

STOP: This is displayed when the **STOP** button is pressed.

3.11 Default Format

The **Default Format** menubutton controls the default display format for:

- all entry boxes,
- the label that displays the difference between the current value of the *PC* and the address of the function the current instruction is in (see Section 3.12.2),
- the output display formats by the **Format** menubuttons defined in the **Set Traces** and **Read** windows.

The current default format is displayed in the label below the menubutton.

To change the default display format:

1. Press the menubutton **Display Format** and select the menu entry corresponding to the desired format (see Section 3.2.3).

3.12 Processor State

This section of the console allows the user to inspect and modify the contents of CHIP's registers, each of which can be accessed through a dedicated entry box and the corresponding menubutton (see Figure 1).

3.12.1 The Register Menubuttons

The format of the data displayed and accepted by each of the register entry boxes is determined by menubuttons, labeled with the names of the corresponding registers. When the register menubutton is pressed, the menu that appears contains the following entries:

Trace: Select this entry to toggle the register's *trace bit*. If both the trace bit and the **Traces** switch (see Section 3.3.3) are on, the content of the register is displayed in the corresponding entry box as it changes during execution. Default: on.

Binary: (see Section 3.2.4);

Octal: (see Section 3.2.4);

Decimal: (see Section 3.2.4);

Hexadecimal: (see Section 3.2.4);

ASCII: (see Section 3.2.4);

3.12.2 The Program Counter

In addition to being displayed in the corresponding entry box, the address contained in the program counter is also displayed in the form *function name* + *offset*, where *function name* is the symbolic name of the function that contains the instruction about to be executed.

3.12.3 The *PS1* Register

The binary representation of the Processor Status Word 1 (*PS1*) register is always available through the row of entry boxes located at the bottom of the **Processor State** section of the console (see Figure 1). Each of the entry boxes (**Intmask, Mode, MM, W, N, Z, V, C**) can be modified independently to change the value of the register. Note that the **Mode** bit can be set by entering either '1' or 'K' (for Kernel mode), and reset by entering either '0' or 'U' (for User mode). The contents of *PS1* can also be inspected and globally modified through the **PS1** entry box.

4 CHIP Console Tutorial

Before you begin:

We assume that you know how to interact with a user interface that uses buttons, menus, and entry boxes. In particular, you should know how to press a button, how to select an entry from a menu, and how to enter a value in an entry box. We also assume that you know what menubuttons, breakpoints and watchpoints are. If you are uncomfortable with any of these assumptions, you should first read Section 3.

Conventions:

Throughout this section we employ the following notational conventions:

- `typewriter` font is used for:
  - text typed by the user.
  - names of UNIX commands.
  - names of UNIX files.
- *italics* font is used for:
  - meta-variables (as in "Open file *file*").
  - symbolic names defined in the CHIP program, such as variable and function names.
  - CHIP registers.
- **boldface** font is used for:
  - Names of console buttons, windows, labels, menu entries, and entry boxes.

For instance, the following describes how a user enters the symbolic name *count* in the **Function Name** entry box defined in the **Breakpoint List** window:

> Enter `count` in entry box **Function Name** of window **Breakpoint List**.

We will also use the expression "Press *button*" as a shortcut for "Press button *button*".

4.1 The Program

The program you will use in this tutorial is contained in file `~cs415/test/console_test.c`. The program is shown in Figure 14. Function *count* computes the sum of the integers from 0 to 8. The value returned by *count* is assigned to variable *size* and then function *MOVBCK* is called to move a block of *size* bytes from *source* to *destin*. Function *MOVBCK* is described in detail in section 4.2 of [BBDS83].

4.2 Getting ready to run CHIP

In this section of the tutorial you will set up your environment so you can complete the three sessions comprising this tutorial.

First, you need to create a new subdirectory `test` in your home directory. To do so:

1. Type `cd` to make your home directory the current directory.
2. Type `mkdir test` to create subdirectory `test` in your home directory.

Then, move to directory `test`, copy file `~cs415/test/console_test.c` into it, and compile `console_test.c`. To do so:

1. Type `cd test` to make `test` the current directory.
2. Type `cp ~cs415/test/console_test.c .` to copy file `console_test.c` to directory `test`.
3. Type `pcc -S console_test.c` to create assembly file `console_test.s`.
4. Type `pcc console_test.s -o console_test` to create object file `console_test`.

```
#include"/usr/u/cs415/hoca/h/types.h"

char destin[45];
int sum;
int size;
int count(){
  int i;

  sum = 0;
  for(i=0; i<9; i++) {
      sum = sum + i;
  }
  return sum;
}

main(){
  char *source;
        size = count();
        source = "Cornell Hypothetical Instructional Processor";
        MOVBCK(source, destin, size);
}
```

Figure 14: The test program `console_test.c`

Figure 15 shows the contents of `console_test.s`. Note that `pcc` generates integers in octal format.

5. Type `ls` to display the contents of `test`.

   Directory `test` should contain the following files:

   - `console_test.c`
   - `console_test.s`
   - `console_test`

4.3 Learning to use CHIP's console

In this section of the tutorial you will learn how to use CHIP's console to monitor program execution. This section is organized in three sessions. Each session explores different features of CHIP's console, using `console_test` as a common test program. Session 1 shows how to start CHIP, execute a program at different speeds, set the format used to display the contents of CHIP's registers, and exit CHIP. Session 2 shows how to monitor execution by tracing regions in CHIP's memory. Finally, Session 3 shows how to use breakpoints and watchpoints, how to write to memory and registers, and how to save to a file the current state of the console.

4.3.1 Session 1

The first thing you need to know is how to start CHIP:

1. Type `chip console_test`.

   Window **CHIP** appears on the screen. The window contains a button, labeled **Cornell Hypothetical Instructional Processor** (see Figure 16).

2. Press button **Cornell Hypothetical Instructional Processor**.

```
.comm   _destin,56
.comm   _sum,2
.comm   _size,2
.globl _count
.text
_count:
~~count:
CSV
jbr     L1
L2:~i=177770
clr     _sum
clr     177770(r5)
L4:cmp  $11,177770(r5)
jle     L5
mov     _sum,r0
add     177770(r5),r0
mov     r0,_sum
L6:inc  177770(r5)
jbr     L4
L5:mov  _sum,r0
jbr     L3
L3:CRET
L1:tst  -(sp)
jbr     L2
.globl _main
.text
_main:
~~main:
CSV
jbr     L7
L8:~source=177770
jsr     pc,_count
mov     r0,_size
mov     $L10,177770(r5)
mov     _size,(sp)
mov     $_destin,-(sp)
mov     177770(r5),-(sp)
MOVBCK
cmp     (sp)+,(sp)+
L9:CRET
L7:tst  -(sp)
jbr     L8
.globl
.data
L10:.byte 103,157,162,156,145,154,154,40,110,171,160,157,164,150
.byte 145,164,151,143,141,154,40,111,156,163,164,162,165,143,164
.byte 151,157,156,141,154,40,120,162,157,143,145,163,163,157,162,0
```

Figure 15: `console_test.s`

Window **CHIP** disappears, and a new window — **CHIP Console** — appears on the screen (see Figure 17). **CHIP Console** is the console's main window. To indicate that the console is in its initial state, the value displayed below the **Simulator State** label is *IPL*, for Initial Process Load.

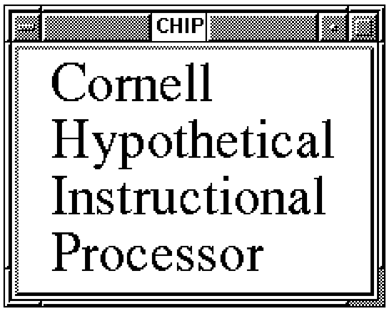


Figure 16: The **CHIP** window.

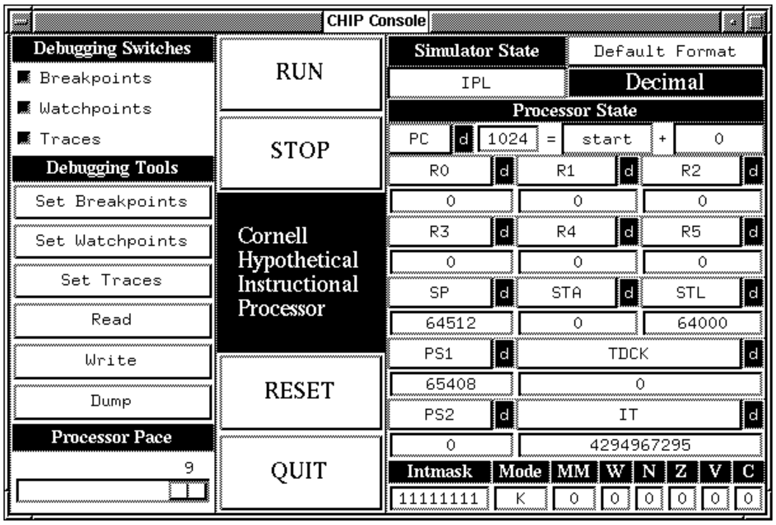


Figure 17: The **CHIP Console** window at startup

3. Press the **RUN** button.

   CHIP starts executing `console_test`, and the value displayed below the **Simulator State** label changes to *RUN*. The values of CHIP's registers are updated as execution proceeds. When execution terminates, the value displayed below the **Simulator State** label changes to *HALT*.

Figure 18 shows window **CHIP CONSOLE** when execution of `console_test` terminates.

In the execution just completed, CHIP processed `console_test` at full speed. To follow execution more closely, it is often desirable to execute a program at a slower speed. You can set the processor pace using the slider below the label **Processor Pace**. First, though, you need to restart CHIP from its initial state:

4. Press **RESET**.

   CHIP is re-initialized. The value displayed below the **Simulator State** label changes back to *IPL*.

Now set the processor pace, and restart execution. To begin with, set the pace to 1.

5. Drag to the left the slider below the label **Process Pace**, until the pace is set to 1.

6. Press **RUN**, and wait for execution to terminate.

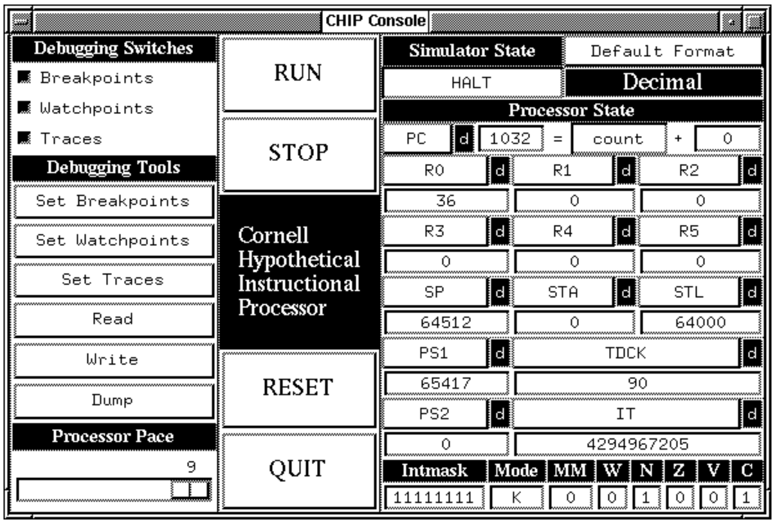


Figure 18: The Chip Console window after `console_test` terminates.

Repeat steps 4, 5 and 6, experimenting with different values for the processor pace. An interesting value to try is 0: with this setting, CHIP single-steps through the program. Try the following:

7. Press **RESET**.

8. Press **RUN**.

9. Press **STOP**.

   Execution is temporarily suspended. The value displayed below the **Simulator State** label changes to *STOP*.

10. Set **Processor Pace** to 0.

11. Press **RUN** a few times.

    CHIP steps through the program. The value displayed below the **Simulator State** label is now *SINGLE STEP*.

12. Set **Processor Pace** to 9.

13. Press **RUN**.

    Execution resumes at full speed.

The processor pace is not the only factor that determines how long it takes CHIP to execute a program. A further factor is the time necessary to update the console display as execution proceeds. By default, the console *traces* CHIP's registers (i.e. it shows how the registers' contents change during execution). However, tracing can be disabled to speed up execution. To execute `console_test` with tracing disabled:

14. Press **RESET**.

15. Press **Traces**.

    The appearence of the square-shaped selector associated with button **Traces** changes from sunken to raised.

16. Press **RUN**.

If you decide to trace CHIP's registers, the console allows you to choose the format in which their contents is displayed. The default format — **Decimal** — is shown in the label under button **Default Format**. You can change the default format at any time by pressing menubutton **Default Format**, and selecting the desired menu entry. You can also change the display format of individual registers by using their private menubuttons. Try the following:

17. Press **Reset**.

18. Set **Processor Pace** to 2.

19. Press **Run**.

20. While CHIP is executing, press **Default Format** and select **Octal**.

    The display format of all registers changes to **Octal**.

21. Press **R0**, and select **Hex**.

    The contents of *R0* are displayed in hexadecimal.

Repeat the above steps experimenting with different formats.

When you change the default display format, the size of the console main window may change. This effect, which is particulalrly visible changing to and from format **Binary**, can cause CHIP to apparently "hang" while the console main window is being resized. To avoid this problem, press **STOP** to temporarily suspend execution before changing default format. When resizing is completed, press **RUN** to resume execution.

Finally, here is how to exit CHIP:

22. Press **QUIT**.

    Window **CHIP CONSOLE** disappears, and CHIP exits.

4.3.2 Session 2

In addition to registers, CHIP's console allows to trace and read regions in CHIP's memory. Since standard I/O functions such as *printf()* cannot be used in C programs targeted for CHIP, examining memory is often the only way to find out the value of a program variable during execution. In this session you'll learn how to trace both local and global variables.

Restart CHIP as in Session 1:

1. Repeat steps 1 and 2 of Session 1.

Tracing gloabal variables in CHIP is straightforward: the address of a global variable can be specified by entering the variable's symbolic name. For instance, here is how to trace variable *sum*:

2. Press button **Set Traces**.

   Window **Trace Memory** appears on the screen.

3. Press menubutton **Low Base** in window **Trace Memory**, and select the menu entry **Symbolic**.

4. Enter `sum` in entry box **Low Base**.

5. Press button **TRACE** in window **Trace Memory**

   Window **TRACE – sum[2240] – Decimal** appears on the screen. This window displays the current value of variable *sum*.

You can also trace larger regions in CHIP's memory. Here is how you can trace buffer *destin*:

6. Enter `destin` in entry box **Low Base**.

7. Enter `45` in entry box **Extent**.

8. Press **Format** and select the menu entry **ASCII**.

9. Press **TRACE**.

   Window **TRACE – destin:45[2194:2239] – ASCII** appears on the screen. This window displays the current contents of buffer *destin*.

Tracing a local variable is a little more tricky, since the address of a local variable cannot be specified symbolically. In order to determine the address, you need to refer to the assembly version of your program, which in our case is file `console_test.s`, shown in Figure 15. The address of a local variable is expressed in CHIP assembly as an offset from the value of register *R5*, the *frame pointer* register. To determine the address of a local variable *var* you need first to add the offset for *var* to the value of *R5*, and then to take the least significant 16-bits of the binary representation of the addition's result.

Finding the value of the offset for *var* is easy: for each local variable *var*, `console_test.s` contains a line of the format:

~*var*=*offset*

where *offset* is the octal representation of the offset's value. The value of *R5* is the value that *R5* assumes upon executing instruction *CSV* in the function where *var* is defined. *CSV* is described in section 4.2 of [BBDS83].

For instance, here is how to trace variable *i*, the index of the loop in function *count*:

10. Find the offset for *i* in `console_test.s`.

    The offset for *i* is 177770.

11. Set **Processor Pace** to 0.

12. Single-step through `console_test`, until the **PC** entry box shows that CHIP is about to execute the instruction at address $count + 2$.

13. Press **R5** and select the menu entry **Octal**.

Entry box **R5** should display the value 0175756.

14. Add 177770 to the contents of **R5**.

    The result of adding the octal numbers 177770 and 175756 is the octal number 375746. Variable *i*'s address is therefore the octal number 175746, obtained by taking the least-significant 16 bits of the binary representation of 375746.

15. Press **Low Base**, and select the menu entry **Octal**.

16. Enter `175746` in entry box **Low Base**.

17. Press **TRACE**.

    Window **TRACE – 0175746[64486] – Decimal** appears on the screen. This window displays the current value of variable *i*.

Since you won't need the window **Trace Memory** for the rest of this session, you can dismiss it.

10. Press **Dismiss** in window **Trace Memory**.

By now there are four console windows on the screen, and you may want to take a moment to lay them out properly. Figure 19 shows a possible layout.

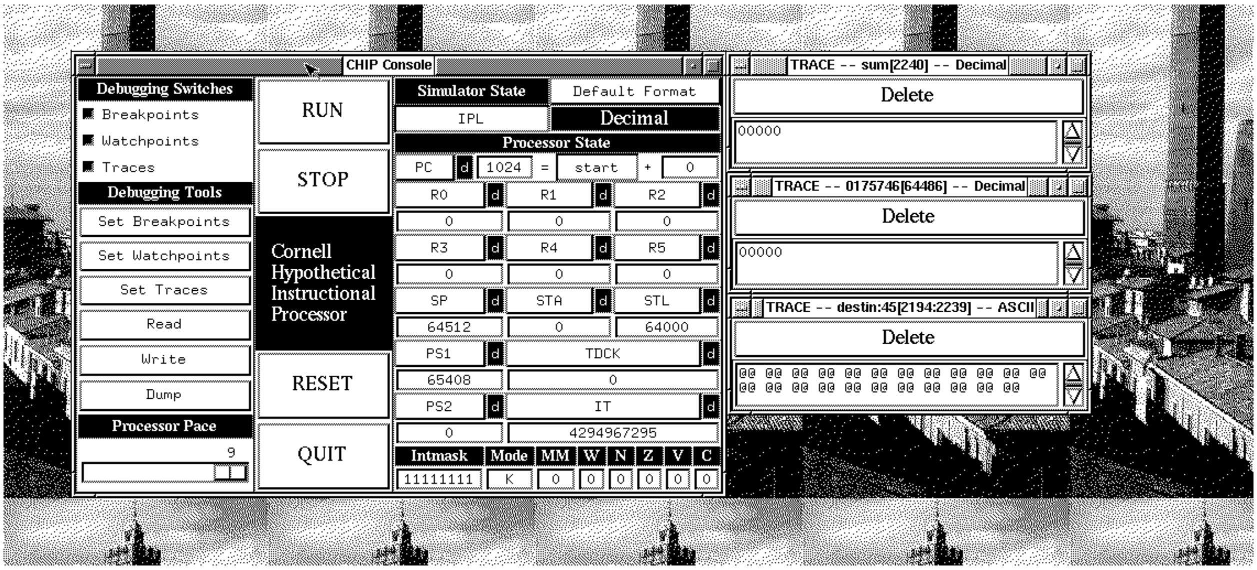


Figure 19: Console windows before running `console_test`.

Now reset CHIP's state, and execute `console_test` at a pace that allows you to examine the contents of the traced memory locations as they change:

11. Press **RESET**.

12. Set **Process Pace** to 1.

13. Press **RUN**.

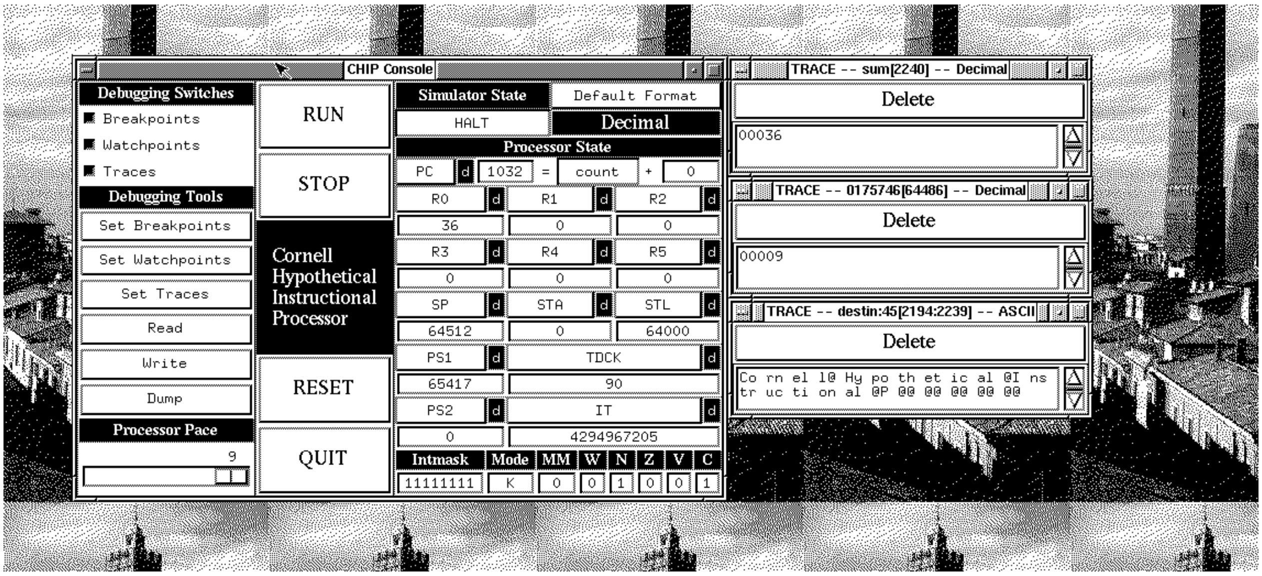


Figure 20: Console windows after running `console_test`.

Figure 20 shows the console windows when execution terminates.

Tracing memory regions is a powerful tool, but, as you saw in Session 1, can considerably slow down execution. Try to execute `console_test`, at full speed, two more times, first with tracing enabled, and then with tracing disabled. Then, exit CHIP to end Session 2.

4.3.3 Session 3

In this session you will learn how to use breakpoints and watchpoints, how to modify the state of CHIP during the execution of a program, and how to save CHIP's state to a file. The first thing you need to do after starting CHIP is to set up windows that allow you to trace the global variables *destin* and *size*, and local variable *i*. Start with the global variables:

1. Restart CHIP.

2. Create the windows **TRACE – destin:45[2194:2239] – ASCII** and **TRACE – size[2242] – Decimal** as you saw in Session 2.

As you saw in Session 2, computing the address of local variable *i* requires kowing the value of register *R5* when CHIP's *PC* is at address $count + 2$. To read *R5*'s value, in Session 2 you single-stepped through `console_test` up to address $count + 2$. Now we'll show you a less tedious approach: just set a breakpoint at address $count + 2$. When CHIP encounters the breakpoint, it will suspend execution, allowing you to read *R5*'s value. Here is how to set a breakpoint at address $count + 2$:

3. Press **Set Breakpoints** in the console's main window.

   Window **Breakpoint List** appears on the screen.

4. Enter *count* in the **Function Name** entry box.

5. Enter 2 in the **Offset** entry box.

6. Press **Add**.

Figure 21 shows window **Breakpoint List** after the breakpoint has been added.

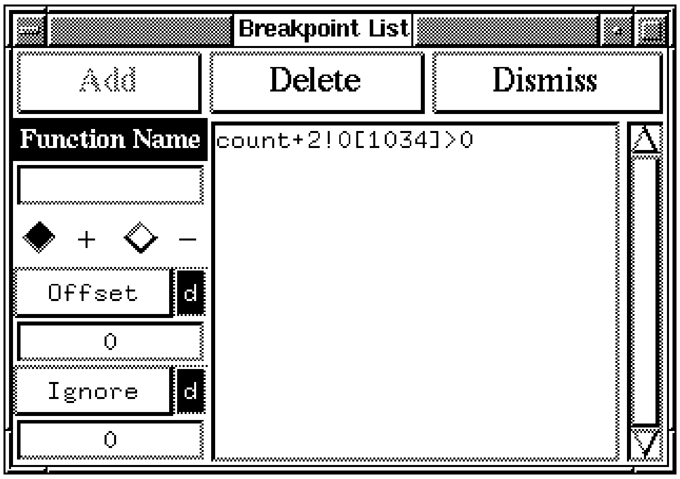


Figure 21: Window **Breakpoint List**.

7. Press **RUN**.

   CHIP suspends execution at the breakpoint.

8. As you did in Session 2, use the value of *R5* to determine the address of *i*, and proceed to create window **TRACE – 0175746[64486] – Decimal**.

Since you won't need to suspend execution at address $count + 2$ for the rest of this Session, you can delete the breakpoint, and dismiss window **Breakpoint List**:

9. Double-click on `count+2!0[1034]>0` in window **Breakpoint List**.

10. Press **Dismiss** in window **Breakpoint List**.

Now continue execution until `console_test` terminates:

11. Press **RUN**.

When execution completes, buffer *destin* does not contain all the 44-character-long string "Cornell Hypothetical Instructional Processor", but only its first 36 characters. This is to be expected, since the number of bytes that are copied starting from address *sorg* to buffer *destin* is determined by the value of variable *size*, which, as shown in window **TRACE – size[2242] – Decimal**, is indeed 36. For the entire string to be copied to buffer *destin*, *size*'s value must be at least equal to 44. If we want to copy the entire string and we don't want to edit and recompile `console_test.c`, then we need to modify the value of *size* during execution, making sure that *size* has the right value — say 44 — before it is passed to function *MOVBCK*. A glance at the code for function *main()* in `console_test.s` suggests that two approaches are possible:

1. We can set *R0* to 44 *before* instruction `mov r0,_size` is executed.

2. We can overwrite *size* *after* instruction `mov r0,_size` has been executed.

In the following we are going to pursue both approaches.

We start by resetting CHIP, and establishing a read-and-write watchpoint for variable *size*. This guarantees that CHIP will suspend execution just before processing the first instruction that references *size*, which happens to be precisely `mov r0,_size`. Here is how to set the watchpoint:

12. Press **Reset**.

13. Press **Set Watchpoints**.

    Windows **Set Watchpoints** and **Watchpoint List** appear on the screen..

14. Press **Low Base** in window **Set Watchpoints** and select menu entry **Symbolic**.

15. Enter `size` in entry box **Low Base**.

16. Press **Add**.

Figure 22 shows window **Watchpoint List** after the watchpoint has been added.

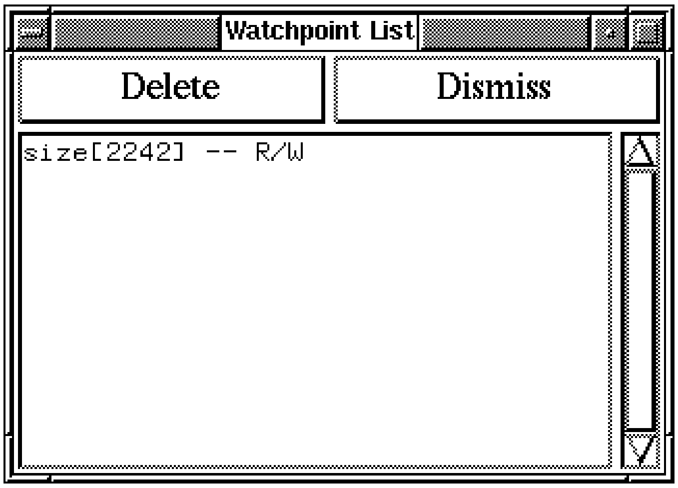


Figure 22: Window **Breakpoint List**.

You won't need to set any other watchpoints in this Session, so you can dismiss window **Set Watchpoints** before starting execution:

17. Press **Dismiss** in window **Set Watchpoints**.

18. Press **RUN**.

    CHIP suspends execution just before executing instruction `mov r0,_size`in `console_test.s`.

Now all you need to do is set *R0* to 44 and restart execution at top speed. First though, you need to take care of the watchpoint for *size*. On the one hand, keeping the watchpoint active is inconvenient, since it would force CHIP to stop execution one more time just before instruction `mov _size,(sp)`. On the other hand, deleting the watchpoint now would require you to enter it again later, when we will explore the second approach to the problem of setting the value of *size*. The right thing to do in this kind of situations is to temporarily disable watchpoints, so that the watchpoint for *size* is kept on the watchpoint list, but it is not active. In practice, the above discussion translates to:

18. Press **Watchpoints**

    The appearence of the square-shaped selector associated with button **Watchpoints** changes from sunken to raised.

19. Enter `44` in entry box **R0**.

20. Set **Processor Pace** to 9.

21. Press **RUN**.

As expected, when execution completes the entire string "Cornell Hypothetical Instructional Processor" has been copied to buffer *destin*. To explore the second approach to the problem of setting *size*. reset CHIP and start execution of `console_test`.

22. Press **RESET**.

    As part of resetting CHIP, watchpoints are re-enabled.

23. Press **RUN**.

    Once again CHIP suspends execution just before executing instruction `mov r0,_size`

Then single-step once to execute instruction `mov r0,_size`:

24. Set **Processor Pace** to 0.

25. Press **RUN**.

    Window **TRACE – size[2242] – Decimal** shows that the value of *size* is now 36.

Now delete the watchpoint for *size*, and dismiss window **Watchpoint List**:

26. Double-click on `size[2242] -- R/W` in window **Watchpoint List**.

27. Press **Dismiss** in window **Watchpoint List**.

    Window **Watchpoint List** disappears.

Finally, overwrite the current value of *size*:

28. Press **Write**.

    Window **Write to Memory** appears on the screen.

29. Press **Low Base** in window **Write to Memory**, and select menu entry **Symbolic**.

30. Enter `size` in entry box **Low Base**.

31. Enter `44` in entry box **Value**.

32. Press **WRITE**.

    Window **TRACE – size[2242] – Decimal** shows that the value of *size* is now 44.

33. Press **Dismiss** in window **Write to Memory**.

```
*************
*           *
* REGISTERS *
*           *
*************

PC:    1032  =  count  +  0
R0:      36    R1:       0    R2:       0
R3:       0    R4:       0    R5:       0
SP:   64512    STA:      0  STL:  64000
PS1:  65417    PS2:      0
TDCK:          90  IT:  4294967205

*****************
*               *
* TRACE WINDOWS *
*               *
*****************

****TRACE -- destin:45[2194:2239] -- ASCII****

Co rn el l@ Hy po th et ic al @I ns tr uc ti on al @P ro ce ss or @@
-----------------------------------------------

****TRACE -- size[2242] -- Decimal****

00044
-----------------------------------------------

****TRACE -- 0175746[64486] -- Decimal****

00009
-----------------------------------------------
```

Figure 23: `console_test.s`

Window **Write to Memory** disappears.

Reset the processor pace at its maximum value, and restart execution:

34. Set **Processor Pace** to 9.
35. Press **RUN**.

Before terminating the session, save the contents CHIP's registers and of all the windows that trace regions in CHIP's memory to file `console_test.sv`

36. Press **Dump**

    Window **Select File** appears on the screen.

37. Enter `console_test.sv` in entry box **File**.
38. Press **OK**.

    Figure 23 shows the contents of file `console_test.sv`.

39. Press **Dismiss**

    Window **Select File** disappears.

That's it!

40. Press **QUIT**

References

[BBDS83] O. Babaoğlu, M. Bussan, R. Drummond, and Fred B. Schneider. Documentation for the chip computer system (version 1.1). Technical Report TR83-584, Cornell University Department of Computer Science, December 1983.