MIPS Lab Environment Reference

The MIPS-Board

The lab board (see Figure 1) is the 79S361 from Integrated Device Technology (IDT). It uses the IDT79R36100, a MIPS R3000 based processor, running at 25 MHz (some boards at 33 MHz). The board is equipped with 4MB DRAM, 1MB SRAM and 2MB ROM.

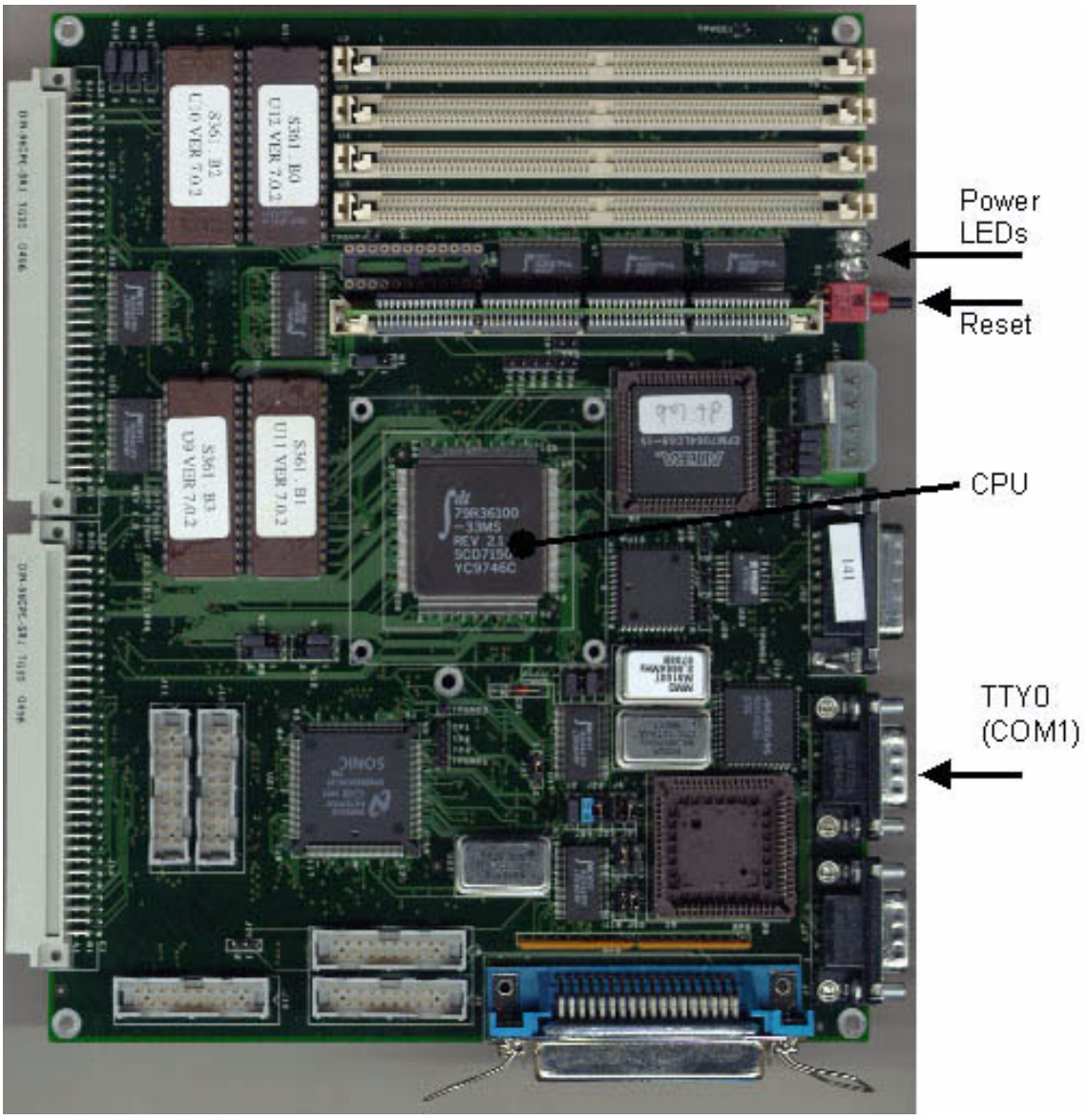

FIGURE 1 The lab board (not including the I/O expansion card).

In order to use the board for the lab, you have to connect it to the computer with a serial cable. Make sure it is connected to TTY0 (see Figure 1). Then turn on the power. The Power LEDs should shine with a green or red light. The board has a monitor in its ROM that you now can access from a terminal, either the one in MipsIt Studio2000 or through any other terminal program. If you wish to use a terminal program, you should set the serial parameters to 38400 bits/sec, 8 data bits, no parity, 1 stop bit and no flow control.

The Monitor

With the on-board monitor software, you can run programs, step through them, inspect registers and memory and much more. The section “Useful Monitor Commands” on page 3 has a list of the most common commands. To enter a command, you must be at an <IDT> prompt. To make sure you are at a prompt, you can press <ENTER> and see that you get a new prompt.

Here is a small example of how to use the monitor. Let us assume that you just uploaded the following little program to the board with MipsIt (see “MipsIt Studio2000 and the Build Process” on page 15 for details on how to do that). For a listing of symbols available, use the command `psym`, and to view the disassembly type `dis start/40`.

```
start:li                        a1, 5
      move                      a2, zero
      li                        a3, 0x80020030
loop: addi                      a2, a2, 2
      sb                        a2, 0(a3)
      addi                      a3, a3, 4
      addi                      a1, a1, -1
      bne                       zero, a1, loop
      nop
end:  nop
```

Note that the disassembly in the monitor often looks somewhat different from the code you wrote. There can even be some more/or different instructions. The reason for this will not be explained now.

You can now set a breakpoint at `loop` with the command `brk loop`. To list the breakpoints, just type `brk`. Now execute the program from `start` with `go start`. The execution will stop when it reaches `loop`. Now take a look at the registers with `dr`:

```
r0/zero=00000000  r1/at  =00000000  r2/v0  =00000000  r3/v1  =00000000
r4/a0  =00000000  r5/a1  =00000005  r6/a2  =00000000  r7/a3  =80020030
r8/t0  =00000000  r9/t1  =00000000  r10/t2 =00000000  r11/t3 =00000000
r12/t4 =00000000  r13/t5 =00000000  r14/t6 =00000000  r15/t7 =00000000
r16/s0 =00000000  r17/s1 =00000000  r18/s2 =00000000  r19/s3 =00000000
r20/s4 =00000000  r21/s5 =00000000  r22/s6 =00000000  r23/s7 =00000000
r24/t8 =00000000  r25/t9 =00000000  r26/k0 =00000000  r27/k1 =00000000
r28/gp =00000000  r29/sp =a00bc000  r30/fp =00000000  r31/ra =00000000
mdhi   =00000000  mdlo   =00000000
```

The contents of registers 5, 6 and 7 are as expected. Let us now step through a bit of the program, instruction by instruction. To do this just use `step`. Step four instructions with `step 4` or by using `step` four times. The contents of the registers 5 to 7 will now be:

```
r5/a1  =00000004  r6/a2  =00000002  r7/a3  =80020034
```

Register 5 is the counter for how many times the loop will run, so let us now make the loop one time extra by setting the register back to 5. This is done with the command `fr r5 0x5`. If you are interested you can step through some more of the program. To execute the rest of the program set a breakpoint at `end` and continue execution with `cont`. Note that you cannot use `go` to continue execution, because it will start from the beginning again.

Let us take a look at what the memory looks like at `0x80020030` and forward. Memory can be viewed as bytes, half words or words. We will use bytes in this example. `dump -b 0x80020030/32` will do just this, where `-b` indicates that we want to view it as bytes. We get the following output:

```
80020030: 42 00 00 00 44 00 00 00 46 00 00 00 48 00 00 00 *B'''D'''F'''H'''*
80020040: 4a 00 00 00 4c 00 00 00 00 00 00 00 00 00 00 00 *J'''L''''''''''''*
```

Useful Monitor Commands

ADDRESS = number | label
ADDRESSLIST = *ADDRESS* [*ADDRESS* [*ADDRESS* [*ADDRESS* ...]]]
BPNUM = breakpoint number
BPNUMLIST = *BPNUM* [*BPNUM* [*BPNUM* [*BPNUM* ...]]]
RANGE = *ADDRESS* - *ADDRESS* | *ADDRESS*/*count* | *ADDRESS*
REG# = r0 ... r31

`asm` *ADDRESS*

Examine and change memory interactively using standard assembler mnemonics.

`brk | b [`*ADDRESSLIST*`]`

Set/display breakpoints. There is a maximum of 16 breakpoints.

`cont | c`

Continues execution of the client process from where it last halted execution.

`clearsymbols | csym`

Clear symbol list.

`dis` *RANGE*

Disassemble target memory specified by *RANGE*.

`dr [`*REG#*`|`*NAME*`|cp0|fpr|fps|fpd]`

Print out the current contents of register(s).

`dump | d [-w|-h|-b]` *RANGE*

Dump the memory specified by *RANGE* to the display.

`fill | f [-w|-h|-b|-l|-r]` *RANGE* `[`*VALUE_LIST*`]`

Fills memory specified by range with *VALUE_LIST*.

`fr [-s|-d]` *REG#*`|`*NAME* *VALUE*

Fill *REG#*/*NAME* with *VALUE*.

`go | g [-n]` *ADDRESS*

Start execution at address *ADDRESS*.

`gotill | gt` *ADDRESS*

Continue execution until address *ADDRESS*.

`help | ? [`*COMMANDLIST*`]`

This command will print out a list of the commands available in the monitor. If a command list is supplied, only the syntax for the commands in the list is displayed.

`move | m [-w|-b|-h]` *RANGE DESTINATION*

Move the block of memory specified by *RANGE* to the address specified by *DESTINATION*.

`next | n`

Step over subroutine calls.

`printsymbols | psym [-a]`

Print list of symbols. Sorted by address if `-a` is specified, otherwise by name.

`regsel | rs [-c|-h]`

Select either the compiler names or the hardware names for registers.

`step |s` *COUNT*

Single step *COUNT* times.

`sub [-w|-h|-b|-l|-r] ADDRESS`

Examine and change memory interactively.

`unbrk | ub BPNUMLIST|all`

Clear breakpoints.

Useful Addresses

`0x80000000-0x800BFFFF`

1MB SRAM (first 64KB used by the monitor)

`0xA0000000-0xA00BFFFF`

1MB SRAM uncached, same physical memory as `0x80000000-0x800BFFFF`

`0x80400000-0x807FFFFF`

4MB DRAM

`0xA0400000-0xA07FFFFF`

4MB DRAM uncached, same physical memory as `0x80400000-0x807FFFFF`

`0x80020000`

Default start address

`0xBF900000`

8-bit I/O

`0xBFA00000`

Interrupt I/O (8-bit):
0 – K2 (input only)
1 – K1 (input only)
2 – Timer (input only)
3 – N/A (undefined value)
4 – K2 latched
5 – K1 latched
6 – Timer latched
7 – N/A (undefined value)

To acknowledge an interrupt, you have to write to `0xBFA00000`. The actual value is ignored; it is the write operation that will reset the state.

`0xBFB00000`

16-bit I/O

PROM Monitor Routines

The monitor has a collection of routines that may come in handy. Arguments to functions are placed in registers a0 to a3, if a function has more than four arguments they are placed on the stack. Return values are placed in register v0.

printf

Formatted print to standard output (the console).

Arguments:

a0	address of format string
a1...a3	arguments 1...3
(sp+n)	arguments 4...n

Return value:

none

Switches:

Linking a argument to the text is accompliced by inserting a %-sign followed by a character signifying the desired formate of the argument. Example of possible switches:

d	argument is formated to a decimal string
s	argument is interpreted as an address to an string.
f	argument is formated to a float string

Example:

Print the string “Number 12345 of 67890” to standard output.

```
string1:.asciiz "Number %d of %d"
...
la   a0, string1
li   a1, 12345
li   a2, 67890
jal  printf
```

putchar

Print a character to standard output (the console).

Arguments:

a0 character

Return value:

none

Example:

Print the character ‘A’ on standard output.

```
li   a0, 'A'
jal  putchar
```

getchar

Get a character from standard input (the keyboard).

Arguments:

none

Return value:

v0 character

Example:

none

puts

Print a null-terminated string to standard output (the console).

Arguments:

a0 address of the string

Return value:

none

Example:

Print the string “MIPS is fun!” to standard output.

```
string1:.asciiz "MIPS is fun!"
...
la   a0, string1
jal  puts
```

gets

Get a string from standard input (the keyboard).

Arguments:

a0 address of string

Return value:

v0 address of string

Example:

```
string1: .space 128
...
la   a0, string1
jal  gets
```

promatob

Convert a null-terminated string to integer.

Arguments:

a0 address of string
a1 address of integer
a2 radix 8, 10 or 16
a3 segment `0x0`, `0x80000000`, `0xA0000000` or `0xC0000000`

Return value:

v0 address of string where conversion stopped

Example:

The word at `integer1` will have the value 12345.

```
string1: .asciiz "12345"
         .align 4
integer1:.word
...
la   a0, string1
la   a1, integer1
li   a2, 10
li   a3, 0
jal  promatob
```

promstrcmp

Compare two null-terminated strings.

Arguments:

a0 address of string 1 (s)
a1 address of string 2 (t)

Return value:

v0 result: < 0 if s < t, > 0 if s > t, 0 if s = t

Example:

Register v0 will have a value less than zero.

```
string1:.asciiz "I'm bigger than you are"
string2:.asciiz "No, you aren't!"
...
la   a0, string1
la   a1, string2
jal  promstrcmp
```

promstrlen

Determine the number of characters in a null-terminated string.

Arguments:

a0 address of string

Return value:

v0 length of the string

Example:

Register v0 will be 14.

```
string1:.asciiz "How long am I?"
...
la   a0, string1
jal  promstrlen
```

promstrcpy

Copy one null-terminated string to another.

Arguments:

a0 address of destination string
a1 address of source string

Return value:

v0 address of destination string

Example:

The memory at `dst_string` will contain the null-terminated string “Copy me please!”.

```
src_string: .asciiz "Copy me please!"
dst_string: .space  32
...
la   a0, dst_string
la   a1, src_string
jal   promstrcpy
```

promstrcat

Concatenate two null-terminated strings.

Arguments:

a0 address of destination string (s)
a1 address of source string (t)

Return value:

v0 address of destination string (s = s + t)

Example:

The memory at `dst_string` will contain the null-terminated string “hello world”.

```
dst_string:
.asciiz  "hello"
.space   32
src_string:
.asciiz  " world"
...
la       a0, dst_string
la       a1, src_string
jal      promstrcat
```

install_normal_int

Install user exception/interrupt handler. The interrupt handler should return non-zero if it processed the interrupt, otherwise it should return zero. The return value is set in register v0.

Arguments:

a0 address of interrupt handler

Return value:

none

Example:

```
my_handler:
   # check if interrupt is NOT for me, if so return 0
li   v0, 0
j    ra
   # otherwise do something useful and return non-zero
   ...
li   v0, 1
j    ra
...
la   a0, my_handler
jal  install_normal_int
```

fptodp

Converts arguments from single precision floating-point to double precision. Needs to be called before passing a floating-point argument to `printf`.

Arguments:

a0 floating-point argument

Return value:

v0 msb of double precision
v1 lsb of double precision

Example:

Print the string “Number 3.5” to the standard output.

```
string1:.asciiz  "Number %f"
...
li.s a0, 3.5
```

```
jal  fptodp
la   a0, string1
move a1, v0
move a2, v1
jal  printf
```

The Simulator

The simulator (*mips.exe*) provides you with the possibility to run programs without the MIPS board. It basically includes the same facilities as the monitor (breakpoints, stepping, etc.) except with a graphical user interface and not by entering commands in a console.

The main view (Figure 2) consists of x units: CPU, RAM, Console, I/O, D-Cache and I-Cache. When you click on a unit, a window will appear (or disappear if it is already open). There is also an interrupt unit available from the View menu.

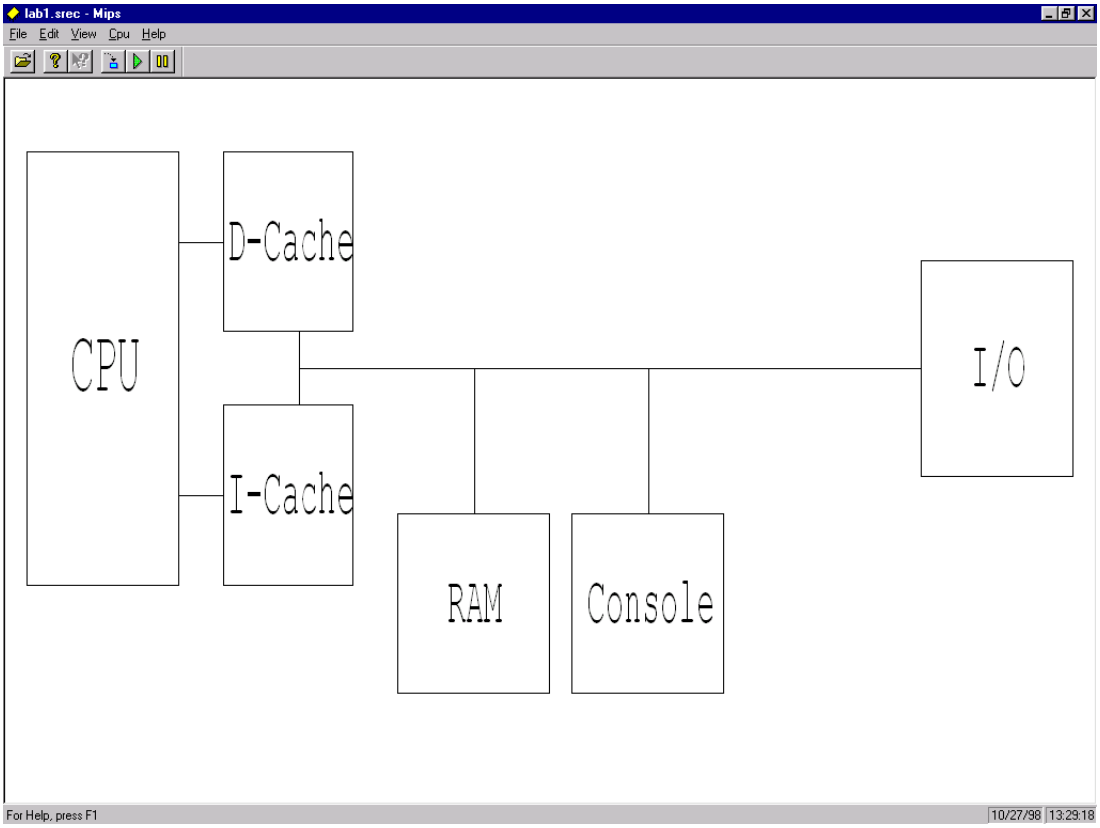

FIGURE 2 The Simulator.

Following is a brief description of each unit's window.

- CPU: View/modify the CPU registers.
- RAM: View/modify memory (also referred to as the MemView). This unit has the most functions; for a more detailed description see below.
- Console: Standard input/output for programs that use it.
- I/O: Simulates the 8-bit I/O unit, with 8 switches and 8 LEDs.
- D-Cache/I-Cache: Views of the data and instruction caches.
- Interrupt: Simulates the interrupt unit, with buttons K1, K2 and the timer.

Memory

The MemView, shown in Figure 3, is a list over the complete 32-bit address space (4GB). Each line is one word-aligned address, 0, 4, 8 and so on. The content of each line is displayed as four bytes in hexadecimal and some form of “translation” of the content (by default assembler). Since, of course, there is not 4GB of memory available and there is no point in allocating memory that is not used, most of the memory will be non-allocated. The non-allocated memory is displayed with a gray color and its content is filled with question marks (?? ?? ?? ??). This is not anything you have to worry about; it is just to explain why most of the memory contains question marks. Addresses can be displayed in two modes, virtual and physical mode. Usually only virtual mode is of interest. Physical mode has restrictions in the MemView and is not normally useful.

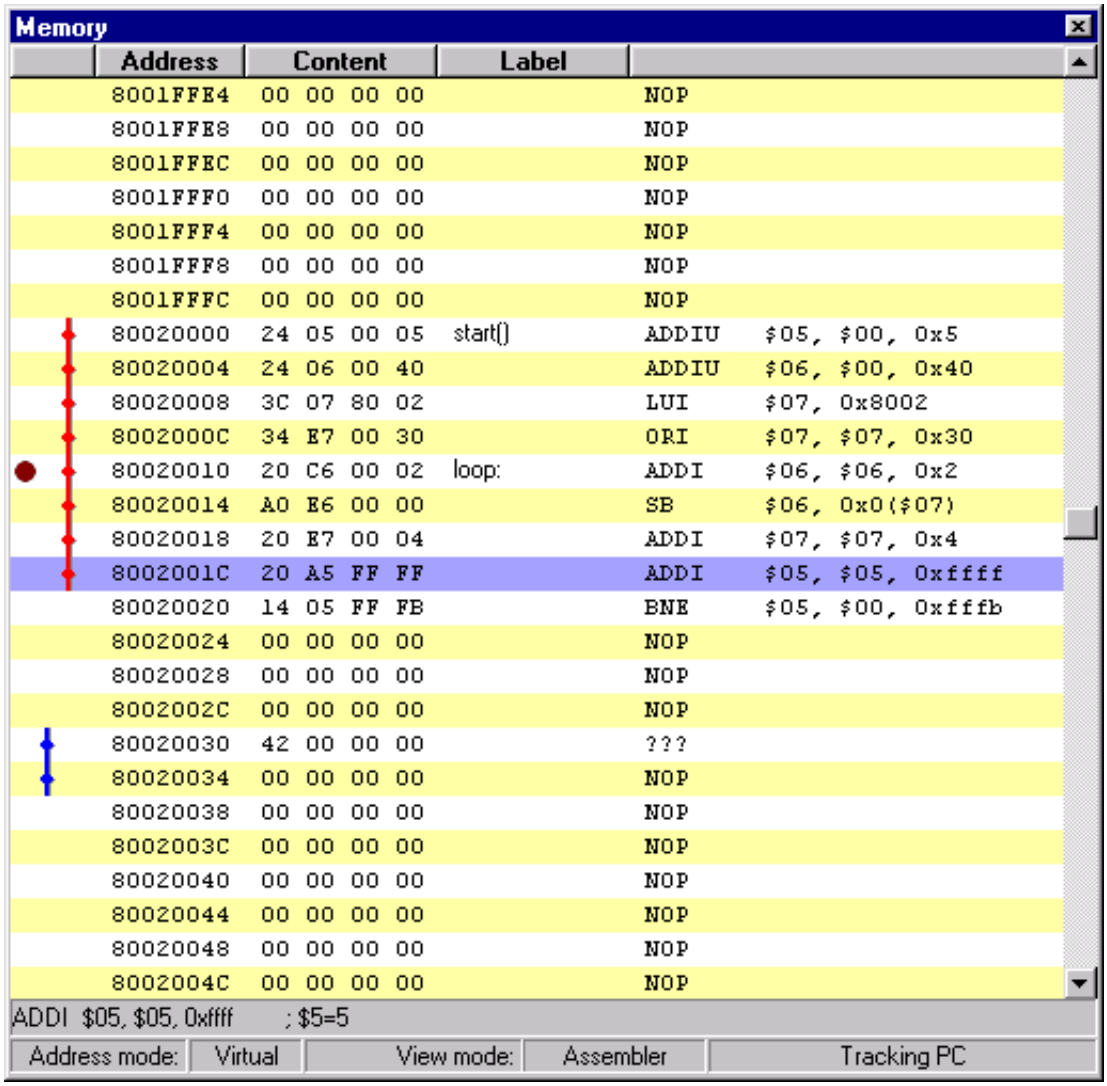

FIGURE 3 The MemView.

Most of the functions of the MemView can be accessed via the pop-up (Figure 4). To view the pop-up, click the right button in the MemView. In the first section you can select the address mode. The second section allows you to choose the translation of the content. The third section is related to the current position of the MemView. You can toggle Track PC. If Track PC is enabled, the view will always keep the address pointed to by the PC in the center. At bottom right of the status bar you can see if tracking is enabled or not (see Figure 3). You can also only Jump to PC; this will center the current address of the PC but it will not track it. Jump to SP does the same as Jump to PC, but it jumps to the address pointed to by the stack pointer (register 29). If any symbols are loaded, you can

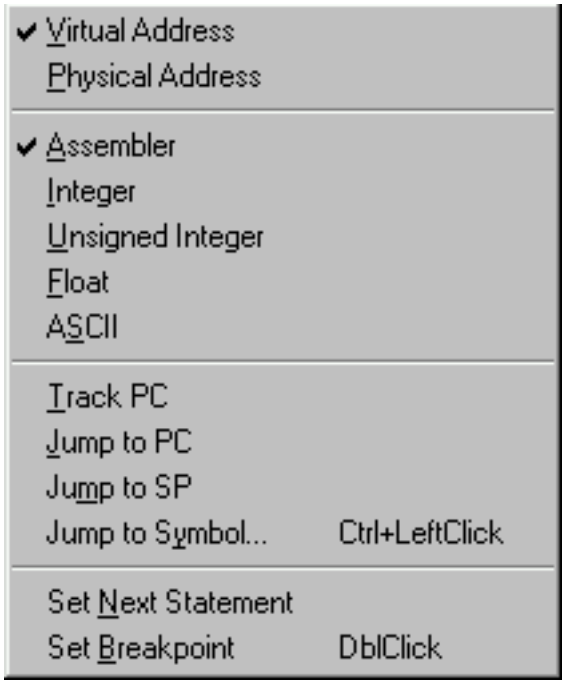

FIGURE 4 The MemView pop-up.

also jump to a symbol. When you do this, a list of symbols will be displayed for you to choose from. In the fourth and last section of the pop-up, you can Set Next Statement and Set/Clear Breakpoint. Set Next Statement will set the value of the PC to the address you clicked on when opening the pop-up. The address will have a focus-rectangle around it. Set Breakpoint/Clear Breakpoint will set or remove a breakpoint at the focused address.

You can also toggle breakpoints by double clicking on an address in the MemView. Breakpoints are displayed as dots in the left column (see Figure 3). The left column may also contain vertical blue or red lines with smaller dots on them. These line indicates if an address is cached: a red line if it is in the instruction cache, and a blue line if it is in the data cache. This is purely for informational purposes and can be ignored.

To jump to a certain address, just click in the Address column of the view. Using the scrollbar will be fairly difficult considering that it covers the whole address space. After entering the address press <ENTER>, or <ESC> if you wish to return to the old address. To edit the contents of an address, click the Content column and enter the data. Press <ENTER> to accept the new value, or <ESC> to keep the old value.

To load a program into the simulator, it is recommended to use the “Upload To Simulator” function in MipsIt. It is possible to load programs from the simulator. To do this, choose Open from the File menu. There are two different file types that can be loaded, .SREC and .OUT files. Note that SREC files contain no symbolic information. After a program is loaded, you run it or step through it, instruction by instruction. To run it at “full speed” choose Run from the CPU menu. To step one instruction, choose Step from the CPU menu.

Cache

The cache views show the contents of the caches, as well as some statistics. There is one view for each cache, data and instruction. Figure 5 below shows the data cache.

The view contains four parts: current address, the cache, an optional write buffer and cache statistics. If the write buffer size is 0, no write buffer will be display. The address displayed is conveniently divided into tag, index and word fields. Depending on the cache configuration, the field sizes will vary. The active row, selected by the index field, is marked gray. The active word, selected by the word field, is marked with blue text. The V column is the valid flag for a block. For a write-back cache, there will also be a dirty flag (D) column.

To configure the caches, go to the Edit menu and select Cache/Mem Config. The configuration dialog has three tabs: Inst. Cache, Data Cache and Memory. Inst. and Data are almost identical; the only difference is that the instruction cache has no write policy. Figure 6 shows the Data Cache config. In order to completely disable one of

the caches, check the Disable box. To just disable the penalty for a cache miss, check the Disable penalty box. Sizes have to be of the magnitude 2^n.

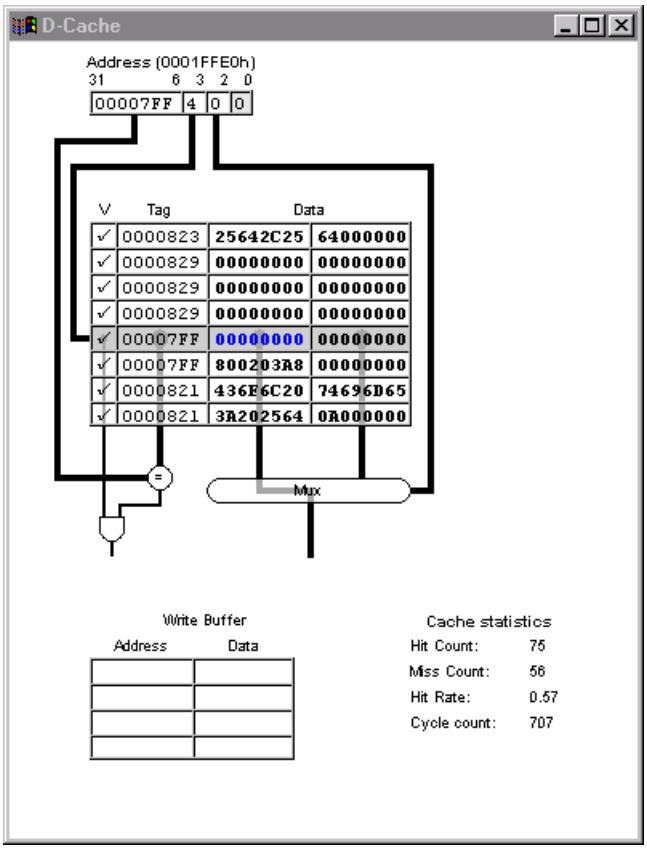

FIGURE 5 Data Cache view.

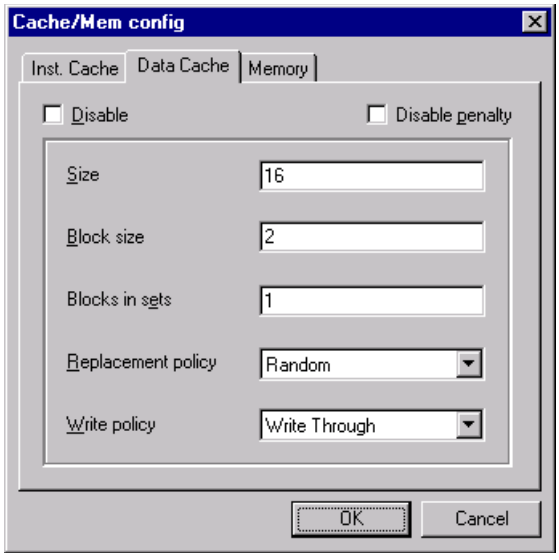

FIGURE 6 Data Cache config.

If penalty is not disabled, the penalty for memory reads and writes (in clock cycles) can be set in the Memory config (Figure 7). Here, you can also change the size of the write buffer. The size is the number of rows in the buffer.

Since the cache views give a performance hit in execution speed, there is another option to view cache statistics while running long programs. The data and instruction caches each have a statistics window, I-Cache Stats and D-Cache Stats, respectively, that can be accessed from the View menu. Figure 8 shows the D-Cache Stats window. Each window shows the current hit ratio, the overall hit ratio and the number of cycles executed.

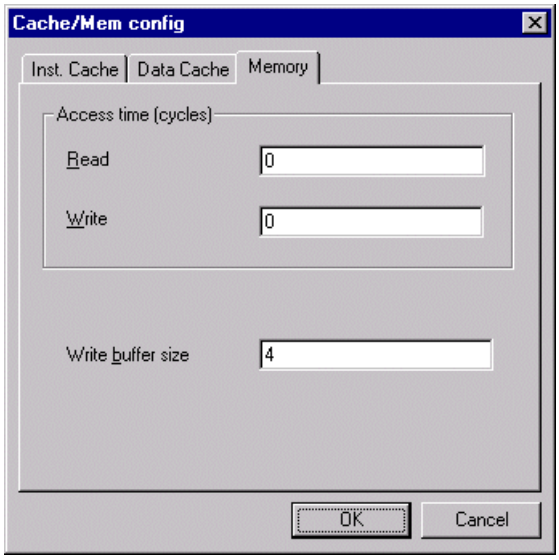

FIGURE 7 Memory config.

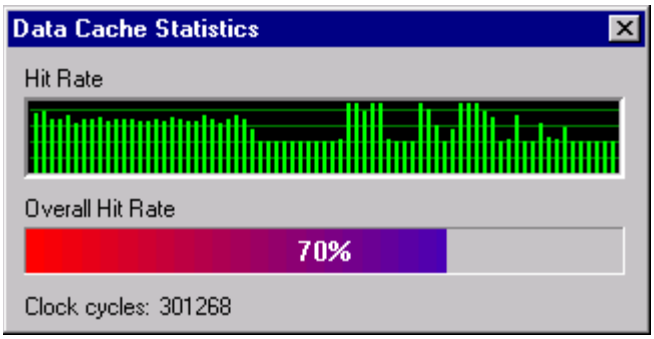

FIGURE 8 Data Cache statistics.

Pipeline

The pipeline simulator consists of a shell program named *Mipspipe2000.exe* that loads Java scripts containing structural information about the pipeline. There are two different scripts that contains different pipeline views of the CPU. One has a simpler pipeline model and is called *\S-script\s.dit*. The other script has a more complex pipeline and is called *\Xl-script\xl.dit*. The small pipeline has no forwarding or hazard detection. The XL version has those features and controller units that can be modified by the user. The controllers are programmed with a Java script.

There are a number of predefined variables that correspond to in/out signals to/from the controller.

Expression syntax:

`var = x;`	Assigns `x` to variable `var`
`x ::=`	
`x & x;`	Bitwise AND
`x | x;`	Bitwise OR
`x ^ x;`	Bitwise XOR
`~x;`	Bitwise complement
`x << x;`	Logical shift left
`x >> x;`	Logical shift right
`x == x;`	Boolean equal
`x != x;`	Boolean not equal
`!x;`	Boolean NOT
`(x);`	Encloses a sub-expression. Operator priorities might not be as expected, use this often.
`var[x];`	Bit extraction, returns bit x from variable var
`constant``;`	Numeric constant, for hex use `0x`-prefix

An example to demonstrate the syntax (note that this does not really do anything useful):

```
tmp = (0p & 0xaa) >> 3;
ALUSrc = !((tmp ^ 0xf0) == 0x05);
```

To open a controller dialog (Figure 9) choose from the Edit menu the desired controller unit to view. The available units are Control and Forward. To enable the user defined controller script, make sure the *Use this script* check box is checked. When the OK button is pressed, the script will be compiled, and any errors will be displayed in a new dialog box, along with the line and character number. *Remember to save your script after big changes. Software can crash, and all work will be lost.*

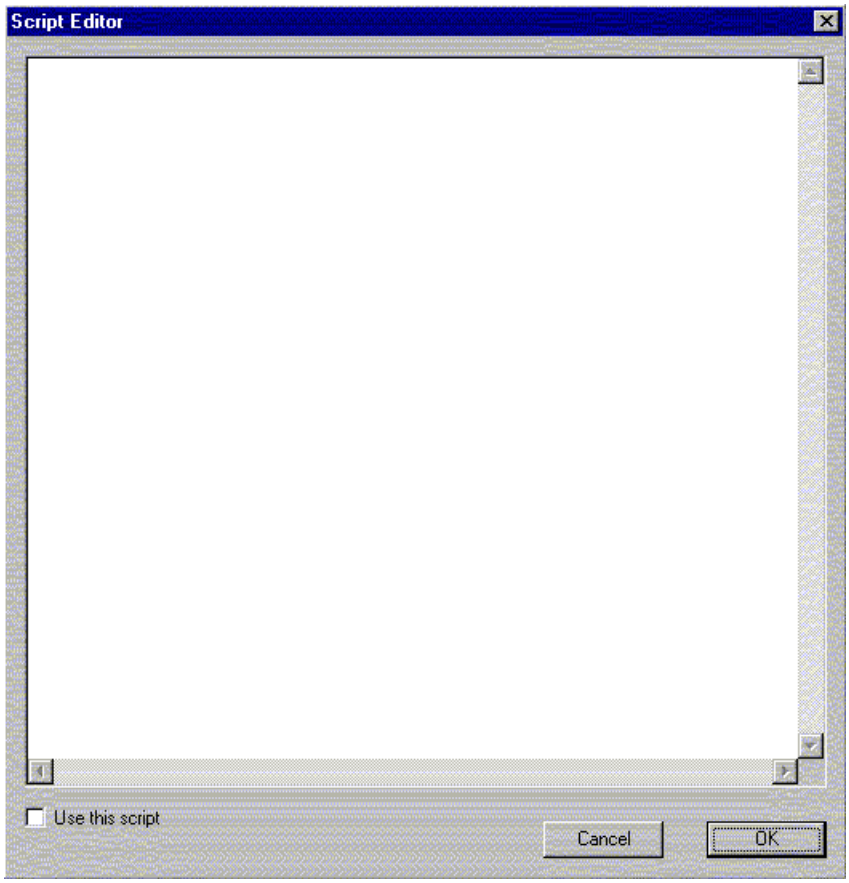

FIGURE 9 Pipeline controller.

MipsIt Studio2000 and the Build Process

MipsIt Studio is a Windows-hosted integrated development environment (IDE) for the IDT MIPS cards and simulator. Figure 10 shows the IDE in action.

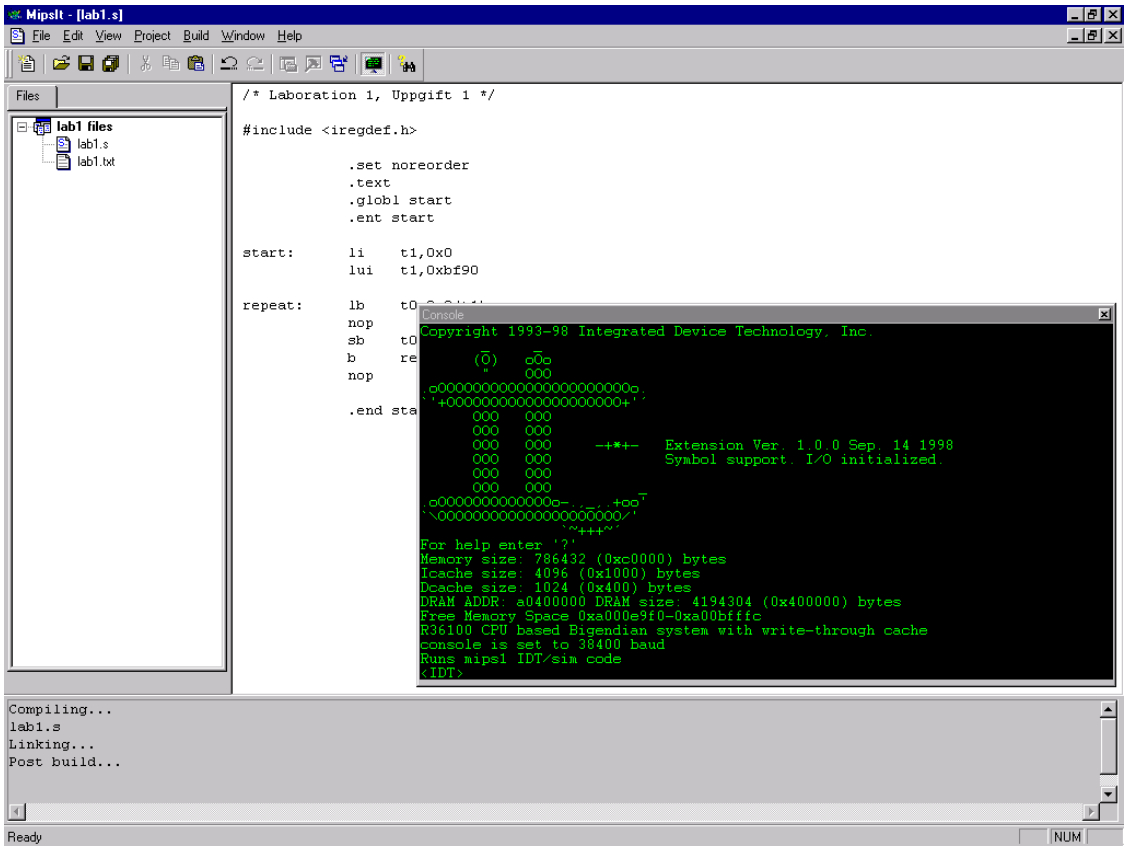

FIGURE 10 MipsIt Studio2000.

If you have used Microsoft Developer Studio/Visual C++, you should have a pretty good idea how MipsIt works. But if you are new to IDEs, you need to know what a project is. A project is a collection of interrelated source files that are compiled and linked to make up an executable file that can be uploaded to the simulator or to the MIPS hardware. A project may also include text files. These are only for informational purposes.

IDE Basics

The IDE consists of the following windows (see Figure 10):

- The project view, which contains a list of files included in a project. To open a file for editing, double click on it in the list.
- The output window, where all output from building, etc., is printed.
- The console, which makes it possible to communicate with the MIPS board. To make the console visible choose Console from the View menu. You can hide it by selecting it again or by pressing <ESC>.

Many commands also have hot-keys (like most Windows programs) to make work more efficient. There is also a toolbar with some of the commands. Some commands are disabled at all times. These are currently non-implemented.

To configure the IDE, choose Options from the File menu. You can change COM settings, compiler executable, paths, etc. When you start MipsIt the first time, it will normally auto-configure correctly, except for the COM-port. After changing any COM settings, MipsIt needs to be restarted.

Creating a Project

To create a new project follow these steps:

1. Choose New from the File menu. Then, in the resulting New dialog box (shown in Figure 11) click the Project tab (if it is not already selected).

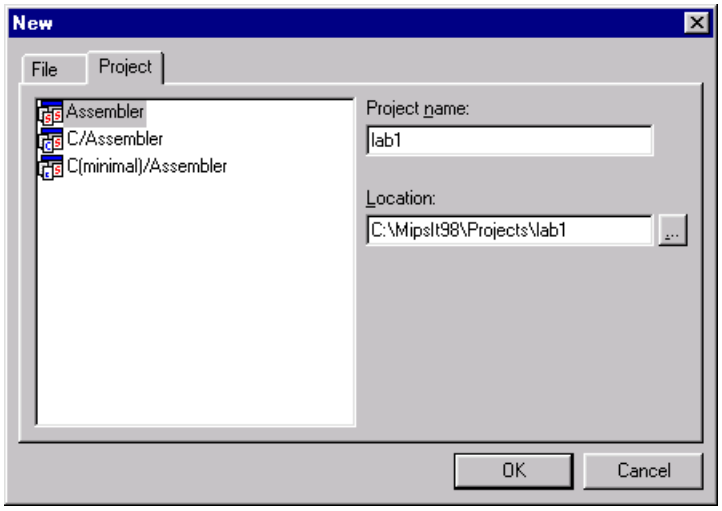

FIGURE 11 The New dialog, Project tab.

2. Select the type of project you want to create from the list. The project types are as follows:

   Assembler: if your project will only contain assembler files.

   C/Assembler: if you want a project that will contain only C or C and assembler files.

   C(minimal)/Assembler: same as C/Assembler, except with minimal libraries. This is your normal choice if you want a project that contains C files.

   The difference between the project types are the default libraries and modules. A C/Assembler project will link with a couple of libraries and will result in a bigger executable (which won't work with the simulator). A C(minimal)/Assembler project will link with only the absolute necessary libraries and will result in a smaller executable than with C/Assembler (and will work with the simulator).

3. Enter a name for the project and change the location if desired, and then click OK.

Adding Files to a Project

If you followed the steps for creating a new project, you should now have an empty project. You can add files to it by either creating new files or adding existing files. Creating a new file is very similar to creating a new project, except you select the File tab (see Figure 12) instead of the Project tab in the New dialog. If you want to add an existing file, choose Add File from the Project menu and then select the file you want to add.

Building and Uploading

In order to build your project, choose Build from the Build menu. Any files that need compilation will be compiled (or assembled), and the executable will be linked. Current status and results of the build process can be seen in the output window (Figure 10). When the project has been successfully built, you can upload the executable to the simulator or to the MIPS board by selecting Upload To Hardware or To Simulator in the Build menu.

If you want to re-compile all files, even those that have not been modified since the last build, choose Rebuild All from the Build menu.

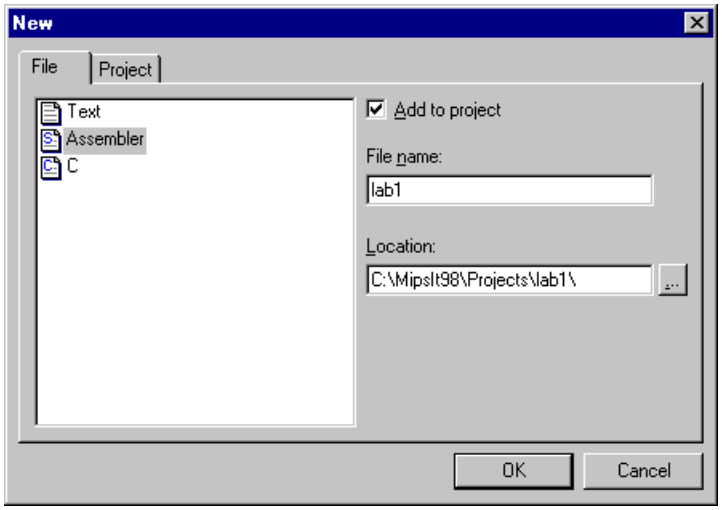

FIGURE 12 The New dialog, File tab.

Note: In order to successfully upload to hardware, the MIPS board must be "ready," i.e., connected to the computer, switched on and not running any program. The easiest way to make sure if the board is ready is to press <ENTER> in the console, after which you should get an <IDT> prompt. To reset the MIPS board press <CTRL>+<BREAK>. If that does not work, press the reset button on the board. If nothing happens after pressing the reset button, check that the COM settings are correct (choose Options in the File menu) and make sure the board is correctly connected to the computer and the power is turned on.

Viewing Assembler from C

It is possible to see the assembler code generated by the compiler from a C source file. To do this, open the C file (make sure it is the active window) and choose View Assembler from the Build menu. When successfully compiled, an assembler window will open with the output. The assembler output is not saved in any file.