Jason Leake
26/6/95

Test programs for the Phase 1 Demonstrator BCID Card

The main BCID test software is implemented as a program called BCID. It provides a command line interface capable of carrying out simple operations on the BCID boards.

More complicated operations can be built up by issuing a series of commands. These may be contained in a script file which can be read by the program. The script can contain conditional branches and use of variables on which some simple arithmetic can be performed.

There are two other simpler test programs available, which are described at the en of this document.

Accessing registers on the BCID board

You can refer to the the various registers on the BCID board, and fields within them using acronyms. For example, the control and status register is referred to as `CSR`, and the External Clock Enable bit within it is referenced as `CSR.ECE`. The registers may be directly manipulated using the `EXAMINE` and `DEPOSIT` commands. In most of the examples below, the program's prompt `BCID>` is shown; obviously you should not type this yourself as input either interactively or inside script files. All values used by the program are unsigned hexadecimal.

For example,

```
BCID>EXAMINE CSR.ACE
  CSR.ACE Auto-inc counter enable = 0
```

Displays the current contents of the Auto-Increment Counter enable bit. `EXAMINE ALL` will display the contents of all registers on the card.

```
BCID>DEPOSIT FCR 50
```

Deposits 50 (hexadecimal) into the FPGA Control Register.

There are three other commands for modifying the values of registers which are intended for use within program scripts. These are

```
BCID>INCREMENT register
BCID>DECREMENT register
BCID>ADD register value
```

For example,

```
BCID>ADD FRCR 81
```

will cause the FPGA RAM Counter Register to be read, 81 hexadecimal added to it and the new value written back.

Other essential commands

`HELP` displays a list of the command which the program will recognise. `HELP` followed by a command will display a description of that command.

You can use `EXIT` or `QUIT` to escape from the program. `QUIT` instantly terminates it, and `EXIT` has the same effect when used interactively.

All commands are case insensitive, although tables and variables retain the case in their names for display purposes although upper or lower case may be used to identify them.

Commands can also be abbreviated, for example:

```
BCID>E CS
```

means the same thing as `EXAMINE CSR`

Handling the FPGAs

The FPGAs are controlled by manipulating the FPGA Control Register. In addition they can be loaded with configuration programs held on disk (via the FPGA RAM Counter and Memory Registers) via a static storage area in the program. This is known as the FPGA Configuration Memory (which should not be confused with the memory on the board).

Because of compiler restrictions, this memory area is limited to 10000 16-bit words (= 2710 words in hexadecimal—*ALL* values displayed by the program are in hexadecimal). This will not be a problem as the configuration memory inside the FPGA is only 2406 words long.

The program's configuration memory can be loaded either from disk or from successive reads of the FPGA RAM Access Register (which is used during readback). It can similarly be written to disk or to the FPGA RAM Access Register. Note that in the case of accesses to the FRAR, the FPGA Counter Register must be set up in accordance with the programming model to allow the desired areas of memory to be addressed.

The following is used to load the configuration memory from a file on disk:

```
BCID>LOAD CONFIGURATION filename
This file contained a valid Xilinx PROM header which has been skipped over.
```

Where the filename has a default filetype of .bit unless otherwise specified. When the file is loaded, the program examines the data at the start of it and if it appears to be a valid Xilinx PROM header then it is skipped over and the data following it read into the memory. If it is not a valid header, then the following message is displayed.

```
BCID>LOAD CONFIGURATION fred.dat
The configuration file does not contain a valid Xilinx PROM header, so the whole file has been loaded.
```

The configuration area may be loaded from the FPGA RAM by not specifying a filename. It will be loaded using the address following the one currently in the FPGA Counter Register and will fill the complete 10000 word memory.

```
BCID>LOAD CONFIGURATION
```

The converse of these commands, to write the contents of configuration memory to a file ot RAM:

```
BCID>WRITE CONFIGURATION filename
BCID>WRITE CONFIGURATION
```

Note that the configuration memory as written back to a file will not have a header if it is reloaded into the program.

Usually, you will not need to directly access the FPGA RAM Access Memory using the `LOAD` and `WRITE` commands because there is also a command to load a specified FPGA with the contents of the configuration memory, and a command to write the FPGA configuration program to the configuration memory.

Thus, to load FPGAs #2 and 3 with the program contained in FPGAPROG.BIT, use the following three commands:

```
BCID>LOAD CONFIGURATION FPGAPROG.BIT
BCID>PROGRAM FPGA 2
BCID>PROGRAM FPGA 3
```

If the file is more that 2710 (hex) words long, further words are ignored.

And to readback the configuration program from FPGA 1 and write it into a file, you would use the following:

```
BCID>READBACK FPGA 1
BCID>WRITE CONFIGURATION FPGAPROG.BIT
```

Finally, you can see what is currently loaded into the program's configuration memory using the command

```
BCID>SHOW CONFIGURATION
```

This will display the size of the configuration program currently loaded and from where is was obtained - the name of the file or other source. For example, after the previous `READBACK` command, it will show:

```
BCID>SHOW CONFIGURATION
FPGA configuration loaded from "bcid22.bit", size = 2406 words
Note that the compiler limits the configuration memory is to 2710 words
although the card has an adressable memory of 4000 words. However, the
FPGA only uses 2406 of them...

The configuration data starts with the following:
FF 04 24 A0 F6 EE 7F 7F FF FD
24 A0 F6 EE 7F 7F FF FD F7 DF
F6 EE 7F 7F FF FD F7 DF 7F FD
7F 7F FF FD F7 DF 7F FD F5 D7
FF FD F7 DF 7F FD F5 D7 5F 7F
F7 DF 7F FD F5 D7 5F 7F FB EB
7F FD F5 D7 5F 7F FB EB EF BF
F5 D7 5F 7F FB EB EF BF FF FE
5F 7F FB EB EF BF FF FE FB AF
FB EB EF BF FF FE FB AF BF FE
```

You will see that it produces a little message about the size of the configuration memory lest you forget, plus the contents of the first 100 bytes.

Handling the other memories

As well as the FPGA configuration memory, there are another thirty-six 256-byte memories on the card, paired into eighteen 256-word memories. These are handled using tables. There are 16 of these 256-word arrays. They each have a number from 0 to 15 and you can also give them names. You can use anything as a name, provided that it does not clash with the hexadecimal numbers 0 to F as the program would then be unable to tell whether you are referring to a table by number or possibly another one by name.

Tables are loaded with data, which can be typed in, read from a file or read from the Memory Access Register (or FPGA RAM Access Register) and similarly output.

You can obtain a list of the tables using the command:

```
BCID>SHOW TABLES
```

```
Table 0, unnamed, all zeroes
Table 1, unnamed, in use
Table 2, unnamed, all zeroes
Table 3, unnamed, all zeroes
Table 4, unnamed, all zeroes
Table 5, name OUTPUT_TABLE, in use
Table 6, unnamed, all zeroes
Table 7, unnamed, all zeroes
Table 8, unnamed, all zeroes
Table 9, unnamed, all zeroes
Table A, unnamed, all zeroes
Table B, unnamed, all zeroes
Table C, unnamed, all zeroes
Table D, unnamed, all zeroes
Table E, unnamed, all zeroes
Table F, unnamed, all zeroes
Use the EXAMINE TABLE command to look at contents of tables
```

Initially, all tables are unnamed and all values in them are set to zero. The expression "in use" here means that at least one value is not zero–though of course you may be using a table which happens to be all zeroes!

Tables can be given names, for example:

```
BCID>SET TABLE 2 NAME INPUT_TABLE
```

And as described in the output look at the contents with the `EXAMINE` table command. You can either specify the table number, or the table name if there is one.

```
BCID>EXAMINE TABLE OUTPUT_TABLE
```

Tables may not have names which clash with keywords. You can load tables by several means. The most obvious is to type them at the keyboard.

```
BCID>LOAD TABLE OUTPUT_TABLE
Type QUIT when you get bored
0>2 4 5 2 4 6 3389
7>
```

And so on for 256 hexadecimal numbers. Note that you may specify one or more numbers per line. It is extremely unlikely that you will ever do this again after you have done it once! If you are overcome with boredom during this process, you can abort the input by typing the command `QUIT`. This will leave the first elements of the table filled with whatever new values you have entered, and the rest unchanged.

A better way of doing the same job is to create a file containing input in the same format and reading it using the command:

```
BCID>LOAD TABLE 4 FROM FRED
```

Note that the default filetype is .bcd, so the file read here is FRED.BCD

You can also modify a single value in a table using the `DEPOSIT` command.

```
BCID>DEPOSIT TABLE OUTPUT_TABLE 3 45
```

This changes the fourth element (i.e. the one with index 3) of output_table to be 45 (fourth because the first index is zero). You can actually change it to be the contents of a register or variable. If you are feeling perverse, you can issue the command

```
BCID>DEPOSIT TABLE INPUT_TABLE 5 MIDR
```

to load the contents of the Module ID Register into the sixth element of the table.

For many tests, it is only necessary to set tables to arbitrary values. For example,

```
BCID>LOAD TABLE INPUT_TABLE WITH 0
```

Will clear the table. There are two other variations,

```
BCID>LOAD TABLE INPUT_TABLE WITH 4 INCREMENT
```

and

```
BCID>LOAD TABLE INPUT_TABLE WITH 4 DECREMENT
```

The first sets the first element of input_table to 4, the next to 5 and so on. The second, sets the elements in the table to the series 4, 3, 2, 1, 0, FFFF, FFFE and so on.

Tables are copied to and from the Memory Access Register or FPGA RAM Access Register using:

```
BCID>LOAD TABLE LUT_OUTPUT WITH MAR
BCID>WRITE TABLE TEST_INPUT TO FRAR
```

When you read back the contents of a memory into a table, you can check that the contents increase monotonically by 1 using the `CHECK_INC TABLE` *`number`* command. This is sometimes useful for checking memories - you load a table with an incrementing value, write it to the memory and read it back..

Finally, the most useful table command will write the lower byte of each entry to the corresponding memory address in a specified channel without affecting the paired byte from the other channel. This is the `PROGRAM` memory command. For example:

```
BCID>PROGRAM LUT_MEMORY 5 WITH TABLE TEST_PEDESTALS
```

This loads the LUT memory on channel 5 by manipulating the CSR, CR and MAR. Other types of memory are `INPUT_MEMORY` and `OUTPUT_MEMORY`. The converse operation, reading back the contents of the memories into a table, is done using commands such as:

```
BCID>READBACK LUT_MEMORY 5 TO TABLE TEST_PEDESTALS
```

FPGA emulation

In order to confirm that the FPGA's are working correctly, the program is capable of emulating their operation. It takes a table and processes it through a software version of the algorithm which is also being run on an FPGA, and writes the results to an output table. It assumes that the FPGA has received a stream of zeroes before the input table to clear any internal memory.

The idea is that the results of this software emulation are compared to the real output of an FPGA. The easiest way of doing this is to clock a table of zeroes through the FPGA followed by the same table which is put through the software version of the algorithm.

The command to process a table through the algorithm is:

```
BCID>PROCESS TABLE input_table TO output_table [ticks]
```

This causes the specified input table to be fed through the currently selected algorithm and written to the output table. Normally, the algorithm will be run until the output table is filled, but you can restrict the number of cycles by specifying the number of simulated clock ticks to run for. Note that there is some latency through both the real and simulated FPGAs, so the first few elements of *output_table* will normally be zero.

There are several algorithms to choose from, and the settings of the Peak Finder Override and FIR Filter Enable flags can also be changed. This is controlled by the `SET ALGORITHM` command. This has the following format:

```
BCID>SET ALGORITHM value
BCID>SET ALGORITHM FFE value
BCID>SET ALGORITHM PFO value
```

For the FIR Filter Enable and Peak Finder Override, the least significant bit is used to select the value. This means that if the value is even, then the flag is set to zero. Note that variables (or even registers) can be specified instead of a value. Note that as elsewhere in the program, values are interpreted as hexadecimal.

You can see what the current selection is, and what algorithms are available using the command:

```
BCID>SHOW ALGORITHM
```

There is currently provision for 16 algorithms. The procedures for coding the algorithms are of the format:

```
word algorithm_0(word input,int pfo, int ffe, int reset);
```

to

```
word algorithm_F(word input,int pfo, int ffe, int reset);
```

The parameters are the data into the simulated FPGA for one clock tick, the settings of the PFO and FFE flags and a reset flag. The reset flag is set when the procedure is called to process the first element of a table to cause it to reset any static variables. The return value is the output from the FPGA at the same clock tick as the input is received. To process a complete table, the procedure will be called 256 times with each element of the table in turn and `reset` set when it is called with the first element.

Writing test scripts

Although the interactive commands are moderately useful, the real power of the test program lies in the ability to write scripts to perform more complex test operations. There are executed using the `RUN` command.

```
BCID>RUN MIDR.BCD
```

The default filetype is again .bcd

Test scripts can be nested up to a level of ten deep. A script can also recursively run itself. A reasonable amount of support for them has is in the program to make these at least reasonably easy to write. These are discussed in the rest of this document.

Comments

`#` to end of line is interpreted as a comment and ignored by the program

Conditional statements

These use the `IF` keyword to perform a comparison. Scalar variables such as registers and variables are compared using `>`, `<`, `=`, `>=`, `<=`, `<>` and `!=` –the last two being synonymous and meaning not equal.

If the test is true, then the command following the `IF` statement on the same line is executed. For example, this command will clear the Protocol Error bit if it is set.

```
BCID>IF CSR.PE = 1 DEPOSIT CSR.CPE 1
```

It is also possible to test tables. They may only be compared for equality or inequality with other tables. For example,

```
BCID>IF TABLE INPUT_TABLE = OUTPUT_TABLE THEN SAY Test passed

BCID>IF TABLE INPUT_TABLE <> INPUT_TABLE THEN SAY This is never output
```

Branching

This is the main use of conditional statements. The `GOTO` keyword causes program execution to move to the statement following the label. Labels are strings which appear at the start of the line and are terminated by a colon. Any arbitrary string may be used as a label, as long as it contains no spaces, tabs, semicolons or commas and ends in a colon.

An example of a label in use is

```
LABEL:    INCREMENT FRAR
          IF FRAR != 0 GOTO LABEL
```

Variables

The program supports the use of unsigned integer variables. These are just like registers on the BCID board, but do not have any physical significance. They must be declared before they can be used and cannot have names which clash with other keywords.

There are two types of variables–global and local. Global variables are declared using the `GLOBAL` keyword and remain in existence after the script which declared them exits. They are visible at the interactive level and from all scripts. There is one pre-defined global variable, which is called `status`. It is used by the `EXIT` command.

For example, to declare a global variable called channel_number, use the command:

```
BCID>GLOBAL CHANNEL_NUMBER
```

Local variables are visible only to the script which they are declared in and all subsequently called scripts unless hidden by another local variable of the same name declared in a subsequent script.

```
BCID>LOCAL LOCAL_VARIABLE
```

Local variables cannot be re-declared once they are in existence except at a lower level; for example a script can declare a local variable and then call another one or itself to declare a variable with the same name. If you declare a local variable twice at the same level you will get an error message and further execution of the script will be aborted (unless you have disabled the handling of errors).

```
BCID>local fred
BCID>local fred
Variable fred already exists
```

Global variables may be re-declared as they are used to pass values back to callers so a script may be re-executed several times. Local variables take precedence over global variables of the same name.

You can list all of the variables declared so far using the `SHOW VARIABLES` command. For example:

```
BCID>global fred
BCID>deposit fred 5
BCID>global fred
BCID>local joe
BCID>deposit joe 6
BCID>local fred
BCID>deposit fred 5
BCID>show variables
status (global) = 0
fred (global) = 5 HIDDEN
joe (local) = 6
fred (local) = 5
```

As with registers, `DEPOSIT`, `EXAMINE`, `INCREMENT`, `DECREMENT`, `IF` tests and `ADD` can be used on them. For example,

```
BCID>LOCAL FRED
BCID>DEPOSIT fred 4
BCID>GLOBAL NEW_value
BCID>DEPOSIT NEW_VALUE FF43
BCID>IF fred = csr say fred = csr
BCID>DEPOSIT CSR NEW_VALUE
```

Error processing

If the program detects an error in the format of a command or the execution of commands which perform complex functions such as programming the Xilinx FPGA (`PROGRAM FPGA` *n)*, then it will report the error and cease execution of any script which it is running. You can override this behaviour using the command:

```
BCID>SET ERROR OFF
```

To re-enable it again, you must issue the command:

```
BCID>SET ERROR ON
```

You can find out the current status is, for example:

```
BCID>SHOW ERROR
Error checking is enabled.
```

Terminal input and output

The commands `SAY` and `GET` perform the terminal I/O function from a script.

```
BCID>SAY This is a line of text
```

One quirk is that any multiple spaces will be collapsed to single ones. The values of registers and variables may be displayed by preceding the name with a dollar. For example:

```
BCID>SAY The value in the CSR is $CSR
The value in the CSR is 66
```

`GET` is similar to `SAY`, except the first parameter is the variable name to be read from the user. Any remaining words on the line will be displayed in the same way as for `SAY`.

EXIT

To return control from a script to the one which called it, use the `EXIT` command. You can specify a return code as a parameter. This return code is either a value or a register or variable, the contents of which will be copied into the global variable `status`. If it is not specified, then `status` is set to zero. You can also return control just by letting control reach the end of the script. In this case, the value returned in `status` will also be zero.

Other commands

The `RESET` command sets all of the registers to their power-up state (except for the Clear Program bit of the FPGA Control Register which does not get set). It does not touch the contents of any of the on-board memories. Sometimes you want to look at the CSR after every command that is executed which reads or writes to the board. One reason for doing this is to find out where the protocol error bit is set.

You can pule the Generate Single Tick bit of the CSR using the command:

`BCID>TICK` *`number`*

Where *number* is an optional count for the number of times the bit is to be pulsed. The default, if a number is not specified, is for the bit to be pulsed once. The number can either be an actual value, or a register or variable.

The command:

```
BCID>SET MONITOR ON
```

Enables an implicit `EXAMINE CSR` after any such command.

The command:

```
BCID>SHOW BCID
```

Displays the address being used to access the Module ID Register, which is the base address if the UA1 crate controller. It also obtains the currently selected module number from the UA1 crate controller and displays that.

The currently selected module can be changed using the command:

`BCID>SET BCID` *`number`*

For example, to select the BCID card at module address 4, use the following command. As always, numeric values are always displayed and interpreted as hexadecimal.

```
BCID>SET BCID 4
```

The default address is 3, which corresponds to the lowest numbered slot used for BCID cards.

You can find out which version of the program you are running by looking at the messages displayed when you run the program.

```
rd27_VME2> bcid
Phase 1.5 BCID demonstrator card diagnostic program, version of 16/6/1995
```

```
10:30
Atlas Level 1 Bit-Parallel Calorimeter Trigger collaboration
This is the OS9 version. Selecting UA1 crate station 3.
BCID>
```

Writing scripts

At the moment, the current BCID board verification scripts comprise a master script called TESTS.BCID and a number of subsidiary ones which are selected by the user specifying a particular test to run. This is the script which TESTS.BCID calls to test the memories (other than the configuration memory). It tests the memories by loading them and reading back the values loaded. The values tried are 0; FFFF; a pattern with the first word being 0, the next word being 1 up to FF; and a pattern with the first word being FFFF, the next word FFFE and down to FF00. The latter two tests are to check that auto-incrementing is working.

Note that it returns a status to the calling script; 0 meaning that the test failed and 1 means that it succeeded. You can use the command:

```
BCID>SET VERIFY ON
```

to help you to debug scripts. It displays each command before it is executed preceded by a string "-->" so that you can easily tell it from output produced by the script. No prizes for guessing how to turn it off again.

```
#
# Test the LUT/input/output memories
# Jason Leake 25 October 1994
#
    say Test LUT/input/output memories

# Write a test pattern of 0000 to all memories, then FFFF, then 0 in first
# memory cell, 1 in second and so on; then FFFF in first memory cell, 1 in
# second and so on
#
    local CHANNEL_PAIR       # Channel to use
    local CURRENT_TEST       # Which test are we doing?
    set table 1 name INPUT_PATTERN
    set table 2 name OUTPUT_PATTERN

    decrement CURRENT_TEST   # Next increment will set back to 0
DO_ANOTHER_TEST:
    increment CURRENT_TEST
    if CURRENT_TEST < 1 goto TEST1
    if CURRENT_TEST < 2 goto TEST2
    if CURRENT_TEST < 3 goto TEST3
    if CURRENT_TEST < 4 goto TEST4
    if CURRENT_TEST < 5 goto TEST5
    say **********************************
    say Test passed - LUT, Input and Output memories all ok
    say **********************************
    exit 1

TEST1:
# Load memories with 0 and read back
    load table INPUT_PATTERN with 0
    goto DO_THE_TEST

TEST2:
# Load memories with FFFF and read back
```

```
    load table INPUT_PATTERN with FFFF
    goto DO_THE_TEST

TEST3:
# Load memories with 0, 1, 2 and read back
    load table INPUT_PATTERN with 0 increment
    goto DO_THE_TEST

TEST4:
# Load memories with FFFF, FFFE, and read back
    load table INPUT_PATTERN with FFFF decrement
    goto DO_THE_TEST

DO_THE_TEST:
    deposit CHANNEL_PAIR 0

CHANNEL_LOOP:
    deposit CSR.CN CHANNEL_PAIR
    deposit CSR.ACE 1       # Enable autoincrementing of counter
    deposit CR FF           # Load with address before first to use
    write table INPUT_PATTERN to MAR
# Now read it back to make sure that it is the same as the one written
    deposit CSR.CN CHANNEL_PAIR
    deposit CSR.ACE 1       # Enable autoincrementing of counter
    deposit CR FF           # Load with address before first to use
    load table OUTPUT_PATTERN with MAR
    if table INPUT_PATTERN != OUTPUT_PATTERN goto FAILED
    increment CHANNEL_PAIR  # Go on to the next pair of channels
    if CHANNEL_PAIR < 6 goto CHANNEL_LOOP
    goto DO_ANOTHER_TEST

FAILED:
# Here if a test fails
    say @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    say Test failed - Input pattern does not match output pattern
    say @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
    say
    say Input pattern is:
    examine table INPUT_PATTERN
    say Output pattern is
    examine table OUTPUT_PATTERN
    exit 0
```

Command summary

The commands undestood by the program are as follows. Italic text within angle brackets means that this is something which must be specified; italic text within square brackets means that this is something which can be specified. Non italic text is used for keywords.

```
ADD <register> <register or value>
CHECK_INC TABLE <name or number>
DECREMENT <register>
DEPOSIT <register> <register or value>
DEPOSIT TABLE <name or number> <index> <register or value>
EXAMINE <register>
EXAMINE ALL
EXAMINE TABLE <name or number>
EXIT
GET <register> [text]
GLOBAL <variable name>
GOTO label
HELP <topic>
IF <register or value> <predicate> <register or value> <command>
IF TABLE <name or number> <= or !=> <name or number> <command>
INCREMENT <register>
LOAD CONFIGURATION [filename]
LOAD TABLE <name or number> [FROM <filename>]
LOAD TABLE <name or number> WITH <value> [INCREMENT or DECREMENT]
LOAD TABLE <name or number> WITH <FRAR or MAR>
LOCAL <variable name>
PROCESS TABLE <from> TO <to> [ticks]
PROGRAM FPGA <fpga_number>
PROGRAM <LUT_MEMORY or INPUT_MEMORY or OUTPUT_MEMORY> <channel> WITH TABLE <name or number>
QUIT
READBACK FPGA <fpga_number>
READBACK <LUT_MEMORY or INPUT_MEMORY or OUTPUT_MEMORY> <channel> TO TABLE <name or number>
RESET
RUN <filename>
SAY <text (can include $register)>
SET ALGORITHM <value or variable>


SET ALGORITHM FFE <value or variable
SET ALGORITHM PFO <value or variable>
SET BCID <UA1 crate station number>
SET MONITOR <ON or OFF>
SET TABLE number NAME tablename
SET VERIFY <ON or OFF>
SHOW BCID
SHOW CONFIGURATION
SET ERROR <ON or OFF>
SHOW TABLES
SHOW VARIABLES
TICK [number]
WRITE TABLE <name or number> TO [FRAR or MAR]
WRITE CONFIGURATION [filename]
```

The default filetype for all files is .bcd, except for FPGA configuration files which are bit. All numbers are expressed in hexadecimal; there is no decimal mode.

Registers are MIDR, CSR, CR, CCR, MAR, FCR, FRAR, and FRCR; or counters C0-C3 or fields of MIDR, CSR or FCR - e.g. CSR.PE is Protocol Error

Up to 20 variables may exist, but they must be declared `GLOBAL` or `LOCAL` first.

Predicates are =, >, >=, <, <=, !=, <>
Commands may be abbreviated (e.g. `EXAMINE` can be abbreviated to `E`)
Labels are a name followed by a colon.

If in doubt use the online help.

Program source

The source of the program is contained in BCID.C, with the definition of the data structure describing the BCID card register layout contained in BCID_REGISTERS.H.

Other test programs

There are two other test programs available for debugging the BCID cards. These are:

VERIFY_BCID ***station***

This allows you to carry out a range of simple tests to check various aspects of the BCID card are working, such as many repeated memory tests with or without the autoincrment function. The real purpose of the program is to quickly verify that a card is working or if it is not, which part is failing. Because it is a computer-based diagnostic it is not able to check the flow of data through the board in the "normal" running mode—data coming in on the front panel and leaving the card via the backplane.

The station parameter specified in the command line is the UA1 crate station number of the board which you wish to test. For example:

```
rd27_VME2> verify_bcid 3
Various simple tests on the BCID card at selected station.
This is version 2/3/1995 15:15 running under OS9.
Selecting UA1 crate station 3.

-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 The tests are:

1. Test Module ID register
2. Test Counter Register
3. Test autoincrementing of Counter Register
4. Test in/out/lut memories with range of test patterns without autoincre-
ment
5. Test in/out/lut memories with autoincrement
6. Test FPGA RAM Counter Register
7. Test FPGA RAM Counter Register autoincrementing
8. Test FPGA RAM with a range of test patterns using autoincrement
Enter test number, or A for all above tests:
```

BLOOP

This is a very simple test program which allows you to repeatedly read or write to a single register in a tight loop. The purpose of it is to allow signals on the board to be examined using an oscilloscope or logic analyser. The BCID test program itself is not really suitable for this because its complex command processing and relatively slow loops do not display a continuous trace. The program is used in conjunction with BCID however, you set up the conditions you want to observe on the card using BCID and then run BLOOP just to access a particular register in a loop.

```
rd27_VME2> bloop
BCID station number? 4
Register? mar
Will access MAR
Read or Write register? w
What value to you want to write? (hex) f0f0
```