Mini-EDSAC User Guide

Bill Purvis, August 2017

Introduction

The Mini-EDSAC emulation on the **BlackIce** board is a much-simplified version of the original Cambridge EDSAC computer. The emulated machine is reduced from 35-bit words, to 16-bit words, with a single order per word, as opposed to to on the original. I have also switched the character set from the obscure encoding used on the Cambridge machine to the basic ASCII set. I have also simplified some of the orders. For example, the shift orders used a rather unusual method to indicate the number of places to be shifted, based on the number of trailing zeros in the order. My version simply specifies the number in the conventional manner, as the numeric value of the address bits in the order. I have also simplified the two multiply orders into one that simply multiplies the Accumulator by the operand. The original orders used a separate Multiplier register and added or subtracted the product to or from the Accumulator. The Load Multiplier order (H) is therefore not implemented.

Installation

The Verilog code is easily compiled by running the **make** command. This builds a file called **mini.bin** which can be downloaded onto the BlackIce board by whatever means you are accustomed to using.

In order to run programs on the emulator you will need a server program to run on your host computer. This is provided as the **miniserve.c** file, which will also be compiled by running **make**. You will need an RS232 link between the BlackIce board and the host machine, either by means of an RS232 adapter or using the **icedfu-uart** firmware on the BlackIce board. You will probably need to modify the blackice.pcf file to tailor the RX and TX ports to suit whichever link you are going to use. If you have already run make, you will need to re-run it after changing the .pcf file, though it should only invoke the final stages of the compilation.

Running Programs

Once the code has been installed on the board, you should be able to run the supplied programs to confirm that it is working.

This is done by running the **miniserve** program, built by make, as follows.

The server program has a number of options, which can be seen by entering:

```
./miniserve -h
```

The simplest test is provided by test4.txt. This is a very basic program which simply prints some digits. It assumes that it will be loaded into the emulated store from location 0 onwards, and is treated as if it was a set of initial orders. To run this simply type:

```
./miniserve -i test4.txt
```

At the prompt ‘:’ type ‘r’ and <enter>.

The -i option indicates that the following argument is the name of a text file which should be converted to binary by the server and downloaded to the emulator as a memory image, and then executed, starting at 0. It should produce the following output:

```
987654321
/012345678
```

This is not very impressive, but does confirm that the emulator is working.

A more impressive test is to run the *squares* program. This is based on the first program that was run on the original machine, and consists of a table of squares of all the integers from 0 to 99. To run this, type:

```
./miniserve -t squares.txt
```

Here, the server assumes the file **initialorders.txt** is to be downloaded. This is an adaptation of the original initial orders, which was hard-wired onto a bank of uniselectors on the original machine. It’s function was to read a program from paper tape, convert it to binary, and load it into store and then run it. In this case, the -t option specifies the name of the file to contain the program to be read by the initial orders. If all goes well, the emulator should print out a table of ten rows of ten four-digit numbers, being the squares of the integers from 0 to 99.

Debugging Programs

While working with the emulator I became aware that it is sometime difficult to know what is really going on. By using the -g option on *miniserve* the emaultor will send trace information to the server, and this gets written into the **session.log** file. This amounts to a line for each executed order, with the values of the **SCT** (program counter), the **IR** (instruction register) and, where relevant, the value in the accumulator on completion. This can be helpful, and has certainly helped me to track down bugs in both the emulator and in test programs.

A further useful tool is to specify an address of 1 on the **Z** (halt) order. This causes the emulator to send a memory dump to the server. Only non-zero words are sent, and these are recorded in the **session.log** file. In order to make better use of this, the server program now records the values sent to record an image of the memory and this can be examined using the **p** command to the server. The syntax of the command is

```
p[l][d] <address> <count>
```

where the l option treats the value in memory as a 32-bit word, and the d option selects decimal format rather than the default hexadecimal.

Note that this is only useful if the program has halted, and if the address on the **Z** order is 1.

Sometimes a program may fail in such a way that the halt order is never executed. There are three possibilities: the program may encounter an invalid order code, or it may get into a loop which never ends, or it may simply hang. In the first case, this is reported as an invalid order, and the memory is dumped. The other cases can provide some information: if you press the button next to S3 on the board, this is detected and a memory dump is sent. The emulator then resets into the idle state to await a further run.

Appendix 1: Emulated Orders

The following list describes the various orders implemented. These are based on those used in the original machine, but with minor variations.

Note that the values are 16 bits, but the Accumulator is 32 bits wide. The original design treated the values as fractions (-1<=acc<+1). Add and Subtract therefore operate only on the most significant 16 bits. Multiply produces a 31-bit value extended by a single zero bit at the least significant end to 32 bits.

A m	Add the value in store[**m**] to the Accumulator
C m	Collate (AND) the value in store[**m**] with the Accumulator
E m	Test the sign of the Accumulator, jump to **m** if not negative
G m	Test the sign of the Accumulator, jump to **m** if negative
I m	Read a character(8 bits) from tape into the Accumulator
L m	Shift the Accumulator left by **m** bits
O m	Print out the character stored in store[**m**]
R m	Shift the Accumulator right by **m** bits
S m	Subtract the value in store[**m**] from the Accumulator
T m	Store the most significant 16 bits of the Accumulator in store[**m**] and clear the Accumulator
U m	Store the most significant 16 bits in store[**m**], leave Accumulator intact
V m	Multiply the Accumulator by store[**M**], result is 31 bits
Z m	Halt the computer

The paper tape format for this emulator consists of a single capital letter, specifying the opcode from the above table, followed by a decimal integer value. For the I and Z orders, the value is not relevant, and is usually 0. To specify constants the character @ is treated as a valid letter, but with value zero. As the order code only has five bits for the operation, the least significant five bits of the ASCII code are used.

See the .txt files for examples of the format. I have include corresponding files with the .etc qualifier which include comments and symbolic identifiers. I have an assembly program which resolves the addresses and outputs the resulting code in virtual tape format. If you want a copy just ask!