# LLAMA

## LOADABLE LOGIC ADAPTABLE MACHINE ARCHITECTURE

---

## FINAL REPORT

Wendy Benbow
wbenbow@ee.ualberta.ca

Rob Behm
rbehm@ualberta.ca

Craig Joly
craig@taipan.mudshark.org

APRIL 2, 2001

# DECLARATION OF ORIGINAL CONTENT

The design elements of the project and report are entirely the work of the authors and have not been submitted for credit in any other course except as follows:

- Table 1 from User Interface section is from Reference [2]
- Application circuit for ML2653 (Ethernet Interface Section) is from Reference [8]
- Drawing 12 & Drawing 13 in the SRAM section are from Reference [9]
- VHDL code for keypad from StudentAppNotes for Tetris Group (1999_w)
- VHDL code for LCD based on code from Reference [2]
- Drawing 4 is from Reference [4]
- Table 3 & Drawing 15 are based on table 13 and figure 3 in Reference [12]

Craig Joly  Wendy Benbow  Rob Behm

# ABSTRACT

In this document, the design and simulation of an an architecture for a portable FPGA configurater are explained. Called LLAMA, Loadable Logic Adaptable Machine Architecture, this configurater reads hardware programming/configuration files via a serial or Ethernet link, stores them locally, then programs/configures any FLEX6000, 8000, 10K or APEX20K FPGA. LLAMA has been implemented on the FLEX10K20 FPGA on the Altera UP1 board and uses an LCD and keypad for the user to control its operation and for providing status and feedback to the user.

# LLAMA DATASHEET

The LLAMA chip implements a FPGA programmer using both Rs−232 and Ethernet (or raw bit stream) to acquire programs. These programs are stored in SRAM until needed for programming. A simple user interface, consisting of a two−line LCD display and a 16−key keypad, provide control and status display.

## FEATURES

- 25.175 MHz external frequency
- Support for Ethernet, Serial, or synchronous bitstream data input
- 2−line LCD and 16−key keypad User Interface
- Stores data on up to 4x32kB SRAM chips(can be non−volatile)
- Programs Altera FLEX 6000, 8000, 10K and APEX 20K FPGAs using Passive Serial transfer mode.

## DEVICE UTILIZATION

LLAMA has been implemented on an FLEX10K20. The utilization of the FPGA is shown below:

| *Module* | *Number of Logic Blocks* | *Percent of Total LBs* |
|---|---|---|
| RS232 | 126 | 10% |
| Ethernet | 256 | 22% |
| User Interface | 381 | 33% |
| RBF Configurater | 115 | 9% |
| **Total** | **981** | **85%** |

## DEVICE PINOUT

LLAMA was implemented on the Altera UP1 board, which provides an internal clock of 25.175 Mhz. Three banks of external connections are available for connecting to external hardware, denoted FLEX_EXPAN_A, FLEX_EXPAN_B and FLEX_EXPAN_C. For simplicity, pins on these connectors will be referred to Axx (xx denotes pin number)

| Module | Pin Name | UP1 Pin | Module | Pin Name | UP1 Pin |
|---|---|---|---|---|---|
| Main | nReset | PB1 | Memory | A0 − 14 | C15 − 29 |
| Comm | RS232In | B30 | Memory | D0 − 7 | C30 − 37 |
| Comm | RS232Out | B32 | Memory | RE | C39 |
| Comm | RxE | B23 | Memory | WE | C38 |
| Comm | RxD | B25 | Memory | CS0 − 3 | C40 − 43 |
| Comm | RxC | B19 | Keypad | Row0 − 3 | A39 − 42 |
| Comm | TxE | B29 | Keypad | Col0 − 3 | A35 −38 |
| Comm | TxD | B31 | RBF | Data | A49 |
| Comm | TxC | B27 | RBF | DCLK | A51 |
| Comm | Col | B33 | RBF | NStatus | A53 |
| Comm | FD | B35 | RBF | ConfDone | A55 |
| Comm | Lpbk | B36 | RBF | nConfig | A50 |
| LCD | Enable | A18 | RBF | initDone | A52 |
| LCD | Select | A16 | | | |
| LCD | R/W | A17 | | | |
| LCD | Data0 − 7 | A19 − 26 | | | |

## TARGET DEVICE CONNECTIONS

| Signal | EPF10K20RC240−4 Pin Name |
|---|---|
| Data | 180 |
| nConfig | 121 |
| conf_done | 2 |
| dclk | 179 |
| nstatus | 60 |

msel0, msel1 and nCE should be grounded on the target device.

NC0 chould be left unconnected.

# TABLE OF CONTENTS

# OVERVIEW

FPGAs are often used for prototyping and for research projects. They are very useful and convenient because hardware can be written to them and debugged without spending large sums of money to fabricate an ASIC, only to discover that there's a mistake in the design. Unfortunately, several popular types of FPGAs are volatile, so the programming or configuration information must be reloaded every time the device is powered up. The LLAMA project aims to fix that, by designing a portable FPGA configurater that will both allow FPGAs to be truly reconfigured "in the field", without the presence of a PC, and to allow quick reconfiguration of an FPGA after power−on.

A prototype LLAMA device has been implemented on a FLEX10K20 FPGA. This is a volitle implemention, but if time and resources permitted, the LLAMA could be implemented on a non−volitle FPGA which would make it much more useful in the field.

# ACHIEVEMENTS

## COMMUNICATIONS

The hardware for the ethernet was connected as shown in the application note, and worked on the first attempt. A link light was established, and valid data was clocked into LLAMA, only for the data portion of the packet. When a RBF file was split into 1500 byte pieces, and transmitted in ethernet frames to LLAMA, it was read correctly into SRAM, and then programmed onto the target board.

The RS232 receiver was simple to connect, and also functioned as expected, once a correct pinout was found for the RS232 connector (there was some confusion between Data Terminal Equipment and Data Communica−tions Equipment in our references). Serial bytes could be programmed into SRAM one at a time by typing on the keyboard, or an entire RBF file could be transferred into SRAM using a common terminal program, such as *HyperTerm*.

The top level communications file detects activity on both the ethernet and the serial, and requests a memory lock. The data is then read from the source (ethernet or serial) into SRAM, until the user indicates that the transmission is finished (recommended to wait for about 5 seconds for a 28K file). The rational for this design was the random nature of the transfer; there was no way of determining whether the transmission was complete or not, since another packet (either serial or ethernet) could still be coming when the current packet was done. It was simpler and less resource intensive to let the user signal the user interface when the transmission was complete than to parse the file to find an end of file.

## MEMORY

Memory simulates and runs flawlessly, including a locking mechanism to only allow only the module that owns the lock to access memory. The configuration module will send a request for memory and wait for an acknowledgment before it attempts to retrieve data for configuring a target. Due to the inability of the ethernet device to pause transmission, the ethernet module does not wait for an acknowledgment. The memory module itself keeps the ethernet module from writing to main memory and those bytes will be lost. The memory module successfully accesses multiple SRAM chips based on requests from the UI.

## CONFIGURATER

The configurater was originally envisioned to be a JTAG programmer and configurater that could work with any IEEE1149.1 compatible device. It would load JBC (Jam ByteCode) files, interpret them and send JTAG commands over a JTAG serial interface to configure/program the target device. Due to the nature of the JBC file, this was much more difficult to implement than originally expected. The JBC file is set up as instructions to a virtual machine[10], which should simplify the interpreting, but the data referencing seemed very complicated. It was decided that time would be more effectively spent making a simple, more limited configuration scheme work than deciphering the intricacies of data referencing in the JBC file.

The latest iteration uses RBF files in a passive parallel synchronous configuration scheme for Altera FLEX 6000, 8000, 10K and APEX 20K devices. The module simulated perfectly, but there were problems programming to a target device. The target would immediately give an error when configuration was attempted. It was assumed that the documentation's[12] reference to a "low to high transition on the *nCONFIG* pin" mean an active high pulse. Study of a waveform included (see Drawing 15) in the document showed that this actually referred to the rising edge of an active low pulse.

The module's state machine was also refined to give it more possible waiting states to better deal with signals from the target device. Configuration of a target device has been demonstrated and verified.

## UI

The keypad and LCD both work as desired. The messages for displaying on the LCD are stored on an EAB ROM instead of the initial implementation that used combinational logic. This nearly halved the size of the UI module and allows messages to be more easily changed.

# DESIGN DETAILS

The LLAMA architecture allows a hardware configuration file to be created on a computer in a program such as Max+Plus II, uploaded to the LLAMA, then used to configure another FPGA whenever desired. If more than one configuration file has been loaded onto the LLAMA board, the user will be able to select between the different files via a simple user interface, consisting of an LCD display and a small keypad.

*Drawing 1: Design Hierarchy*

This architecture is broken into different modules, each of which interfaces with the other modules via the core as shown in the design hierarchy in Drawing 1. The core is responsible for transferring data between and providing control signals to the other modules. Configuration files are transferred into the LLAMA board either by an RS−232 serial or Ethernet connection. A small application for the host computer will need to be created to support either of these data transfer methods, and implement a protocol compatible with LLAMA. Once the configuration file is read from the computer, it is stored in local RAM. After the transfer is complete and veri−

fied with an 8−bit checksum, it can be written to permanent storage. The user can then connect a target board, and select the configuration file to be used via the user interface (LCD and keypad). Configuration of the target board is then carried out using the RBF configuration module. A block diagram showing this arrangement is shown in Drawing 2.

*Drawing 2: Block Diagram of LLAMA System*

In total, the LLAMA device consists of four main modules:

- Communications (Serial and Ethernet)
- External Memory (SRAM)
- RBF Configurater
- User Interface

## COMMUNICATIONS

### RS−232

One simple method of loading the configuration data from a computer onto the LLAMA board is via a serial connection utilizing the RS−232 protocol. This is an established and reliable technology that will provide a common interface to almost all current computers. In general, both 9−pin and 25−pin

serial cables exist, but our device will implement only the 9−pin standard, as is most common for non−modem serial applications.

The RS−232 standard specifies a "non return to zero" encoding scheme for the serial data. In specific, logic ’1’ levels are encoded with negative voltages, and positive voltages correspond to a logic ’0’.[3] The exact specifications are in Table 1.

| *Logic Level* | *RS−232 Transmit Specification* | *RS−232 Receive Specification* |
|---|---|---|
| ’0’ | +5 to +15 V | +3 to +15 V |
| ’1’ | −5 to −15 V | −3 to −15 V |

*Table 1: RS232 Level Specifications*

Clearly, these logic levels are not compatible with those used in the FPGA, and a conversion will be necessary. The LT1080 was chosen as a good solution, due to its use of charge pumps to generate the required voltage levels from a single +5 V supply. The schematic for this connection may be seen in Drawing 16 (Page 19).

To properly interface with a PC, some form of flow control must be used. Hardware signals CTS, RTS, DCD, etc are used for this. Since this is a low performance application (ie. Not strongly bi−directional) these lines are merely connected in a loopback function, as shown in Drawing 3[4]. In this manner, the remote computer takes care of its own flow control, freeing us from dealing with it.

*Drawing 3: Serial Loopback Connections*

When a transmission is not in progress, the data line is held at logic 1 (negative voltage). The start of a transmission is signaled by the data line switching to a logic 0. This is called the "start bit", and data will follow this bit at a predefined rate (typically 9600 bits/sec). The end of a transmission is indicated by one or more stop bits[3][5] (one for LLAMA). This is summarized in Drawing 4.

*Drawing 4: Serial Bitstream*

We chose a transfer rate of 115200 bits/second to reduce transfer times. The reception of data is triggered by the arrival of the start bit. Once the start bit is detected, data follows at the data bit rate [3]. A 115200 Hz counter with an enable is used to time transmission. Once the start bit is detected, the counter is started, and the data bits are shifted into an 8 bit serial−parallel converter for conversion to bus width. Once a whole byte is accumulated, it is sent to permanent storage.

### RS−232 Receiver

The receiver takes the ingress bitstream, converts it to a 8−bit parallel stream, and places it on the data bus. Appropriate addressing and flow

control will be handled by the core. A block diagram of the receiver is as follows:

The combinational logic in the state machine controls the flow of bits to the shift register, ensuring that only valid data is copied to the bus. When the system is idle or a complete byte of information has not been received, then no data is wrote to the bus (controlled via signals to the core).

A clock is provided to both the shift register (which also has an enable, controlled by the state machine) and the state machine at 16 times the bit rate. For example, for 115200 baud, a 184 MHz clock is provided.

*Drawing 5: RS232 Receiver − Block Diagram*

The state machine is structured as shown in Drawing 6. When there is no transmission in progress, the sender holds the data line high (from a logic level − the actual voltage on the RS232 line is negative). This corresponds to a *wait_start* state. When the sender begins a transmission, it is indicated by a "start bit". The detection of the leading edge of a start bit triggers a state transition to the *got_start*. The state machine waits in this state for 24 clock cycles (8 clocks cycles to get to the approximate middle of the start bit, then 16 more to get to the middle of the first data bit) then transitions to the *data_xfer* state. At this point the shift register is enabled, and eight bits are shifted in, at 1/16th of the clock rate (the bit rate). After eight bits have been received, the state machine transitions to the *wait_stop* state. If a valid stop bit is detected, the bus is signaled that there is valid data. Whether a stop bit was received or not, the state machine transi−tions to the *wait_start* state.

*Drawing 6: RS232 Receiver − State Machine*

### RS232 Transmitter

A transmitter was implemented to allow feedback to the serial user, as well as for sending configuration data to a BitBlaster connected to a target device. Currently, it is attached to the receiver, and echoes any data sent to the FPGA back out. There is a slight delay between reception of the original signal and transmission of the signal, approximately 8 clock cycles, to allow the transmitting software on the PC time to catch up. The transmitter is more straightforward than the receiver − a start bit and a stop bit are added to the input data stream, and then they are shifted out of a shift register. This is summarized in the following block diagram:

*Drawing 7: RS232 Transmit Block Diagram*

The State Machine controls the shift register, allowing it to shift out data only when there is valid data coming in from the bus. This is controlled by the core.

*Drawing 8: RS232 Transmit State Machine*

As currently connected, the bus connection is made to the output from the receive shift register, resulting in an echo configuration.

## ETHERNET INTERFACE

An Ethernet interface is to be provided with LLAMA for a more accessible (and faster) means of loading configuration files into permanent storage. Ethernet is a popular networking technology and comes in a variety of speeds and cable specifications. The most common varieties are 10 Base−T and 100 Base−T, where the preceding number indicates the speed, in megabits per second, and the "−T" indicates the use of twisted pair cable. For LLAMA, 10 Base−T was chosen as the access method, as it requires a much slower clock speed than 100 Base−T.

The 10 Base−T standard, contained in IEEE 802.3, specifies a total of four wires in the twisted pair cable. Receive and transmit lines are separated into two half−duplex differential signals to allow elimination of noise.[6]

An Ethernet transceiver derives its clock from the Ethernet packet itself, which is encoded using Manchester encoding, a bit scheme that requires a clock at double the data rate. Hence, a 20 Mhz clock is provided by the Ethernet packet. Rather than perform Manchester Decoding in the FPGA (which would consume a great deal of logic cells, assuming the FPGA is fast enough) an external IC was chosen for this task, the ML2653. This IC is connected via isolation transformers to the twisted pair cable as shown in Appendix D.

The receive state machine controls the data flow. When RxE (Receive enable) is first asserted, the state machine skips the first 12 bytes of data (96 bits), ignoring the source and destination address, assuming point−to−point connection. It then records the packet length, so it knows when to stop reading data. All data after the length field is assumed to be data, and is copied to the serial−parallel converter for transfer to memory. If at any time data flow is interrupted, or RxE goes low, a data fault is sent to the memory controller, and the packet is erased. This process is summarized in Drawing 9.

*Drawing 9: Ethernet Transmit State Machine*

*Drawing 10: Ethernet Transmit Flow*

Transmit data is shifted out to the ML2653 at a rate determined by the TxC (Transmit clock) line that is provided from the ML2653.

For simplicity, a point to point connection is assumed, with all packets being broadcast. This eliminates the need to filter packets, a job easiest done in software. Again, for simplicity, the transmit section is not necessary. A no−collision environment will be assumed (point−to−point connection) and no interference or re−sends will be necessary (will be verified at a later date).

There will be a synchronizer register between the input shift register and the bus in order to ensure that signals are clocked to the main clock domain properly. A simple state machine will control shift register as well as signal the bus that there is data. This is shown in Drawing 11.

*Drawing 11: Ethernet Receive Blocks*

The flow of a typical packet transmission is:

1. Host initiates transmission. A preamble of alternating 1's and 0's is sent on the twisted wire pair, Manchester encoded
2. The ML2653 recognizes this as the start of a packet, and prepares to receive data
3. When the preamble is done, the ML2653 asserts RxE (the receive enable) and starts decoding bits from the line. These bits are then sent on the RxD line on the rising edge (this is programmable) of RxC, which turned on at the same time as RxE was asserted
4. When the receiver (FPGA) detects a rising edge on RxE, it starts the shift register, enabling it to receive data on the rising edge of RxE. When the eighth bit is shifted in, the synchronizer register is also loaded.
5. When RxE goes low again, the data transmission is presumed complete, and the last 4 bytes are the checksum to match against the calculated one, if time and space permits.

# EXTERNAL MEMORY

## SRAM

LLAMA has permanent storage of RBF configuration files in the form of SRAM. The SRAM system consists on four MOSEL P51256SL−10 chips. They each require 15 address pins (shared) and 2 control pins (one shared).

To read and write data onto the SRAM, the following sequences can be used[9]:

| Write Cycle | Read Cycle |
|---|---|
| 1. Assert Address to address bus | 1. Assert Address to address bus |
| 2. Assert Write Enable to active state | 2. Assert Output Enable to active state |
| 3. Write Data to data bus | 3. Read Data from data bus |
| 4. Pull Write Enable to inactive state | 4. Pull Output Enable back to inactive state |

*Table 2: Read and Write Cycles*

These sequences are illustrated by the state machines below:

*Drawing 12: SRAM Read Cycle*

*Drawing 13: SRAM Write Cycle*

The memory control module has several purposes. It implements a hand−shaking mechanism so that only one of the configurater or transfer modules can access the device. The device that wishes to access memory asserts a request signal high. When memory is free, the control module will assert an acknowledgment. At this time, the requesting device has access to the memory.

Since all memory access is to be sequential (start at an address and either read or write consecutive bits), addresses are also calculated in the memory module to simplify the logic in modules that require memory and to reduce redundant processes. When a device has locked the memory, it simply sends either a data valid or data request pulse and the control module increments the address and sends a command to the memory module to read the data into or out of the SRAM.

Do to hardware availability, each configuration file will be stored on a separate 256kbit SRAM. The user interface will select the desired file (based on user input) and then select the appropriate chip select to enable/disable the chips.

## RBF CONFIGURATER

The RBF configurater module will configure a target Altera FLEX10K, FLEX8000, FLEX6000 or APEX2000 device with an RBF configuration file. The RBF file can be created from an SOF (SRAM Object File) using the Max+Plus II software.

The RBF configurater is controlled by two signals from the UI module, a start signal and a stop (abort) signal. The abort signal should be rarely used because target device configuration takes approximately 1.1s at the RBF module's frequency (224kHz).

Three signals are used to configure a target device and another two are monitored for status signals. The three output signals are a signal to put the target into configuration mode (*nconfig*) the configuration data (*data*), a configuration clock (*dclk*). The signals being monitored are *conf_done* and *n_status*, a complete signal and an error signal.

Once the RBF module receives an active high pulse on the start signal, it sends asserts its memory request signal high. After an acknowledgment is received, the *nconfig* pin is pulsed low and the module waits for the target to release *nstatus*. When *nstatus* returns high, the RBF module starts pulsing the configuration clock (*dclk*) and sending configuration bits, least significant bit first, onto the *data* pin. Once the target device finishes configuring, it pulls *conf_done* high. At this signal, the RBF module release memory, stops placing shifting bits onto the *data* pin and pulses *dclk* 16 more times before returning to a wait state. This sequence is shown in Drawing 14.

The target device timing waveforms are shown in Drawing 15 and the timing parameters are in Table 3. Due to the RBF module's low frequency compared to the maximum frequency of the target device, all of the values in the table are honored with no special timing modifications to the code.

*Drawing 14: RBF Configuration State Machine*

*Drawing 15: RBF Timing Diagram*

| *Symbol* | *Parameter* | *Min* | *Max* | *Units* |
|---|---|---|---|---|
| $t_{CF2CD}$ | *nCONFIG* **low to** ***CONF_DONE*** **low** | | 200 | ns |
| $t_{CF2ST0}$ | *nCONFIG* low to *nSTATUS* low | | 200 | ns |
| $t_{CF2ST1}$ | *nCONFIG* high to *nSTATUS* high | | 4 | µs |
| $t_{CFG}$ | *nCONFIG* low pulse width | 2 | | µs |
| $t_{STATUS}$ | *nSTATUS* low pulse width | 1 | | µs |
| $t_{CF2CK}$ | *nCONFIG* high to first rising edge on *DCLK* | 5 | | µs |
| $t_{ST2CK}$ | *nSTATUS* high to first rising edge on *DCLK* | 1 | | µs |
| $t_{DSU}$ | Data setup time before rising edge on *DCLK* | 10 | | ns |
| $t_{DH}$ | Data hold time after rising edge on *DCLK* | 0 | | ns |
| $t_{CH}$ | *DCLK* high time | 30 | | ns |
| $t_{CL}$ | *DCLK* low time | 30 | | ns |
| $t_{CLK}$ | *DCLK* period | 60 | | ns |
| $f_{MAX}$ | *DCLK* maximum frequency | | 16.7 | MHz |

*Table 3: RBF Timing Parameters*

## USER INTERFACE

The user interface provides a means for the user to control the LLAMA board, and will consist of an Emerging Technologies 16x2 LCD display and a 4x4 Series 83 keypad. In order to use the LLAMA, the user must be able to select the desired file to load, and then start the loading process.

The UI is run using a state machine that asks the user for input at nearly every step. This state machine can be found in Drawing 16 below. By looking at the table of messages (Table 4) and the state machine diagram one can easily understand the system. Essentially, the user can see the current active file (for both loading and downloading), select a new file, load the active file and receive a message when the file is loaded on the target board. The user

must tell the system when the the system is done loading each configuration file into the RAM.

*Drawing 16: UI State Machine*

| Msg_code | Message |
|---|---|
| ’000’ | "Active File: [#]<br>New File or Load" |
| ’001’ | "Load File [#]?<br>[OK] [Cancel]" |
| ’010’ | "Now Loading<br>[CANCEL]" |
| ’011’ | "ABORTED!! Press<br>[OK] to Continue" |
| ’101’ | "Downloading File [#]<br>[OK] when Done" |
| The file number [#] is dynamically changed by the system | |

*Table 4: UI States*

## KEYPAD

The purpose of the key pad is to give the user a means of communicating with the LLAMA. This keypad consists of a grid of 4 rows and 4 columns. When a button is pushed, it acts as a switch and completes the circuit between one of the rows and one of the columns. By using pull up resistors on the rows, and driving the columns low, when a button is pushed, the input will be low for detection. The columns can be driven low one at a time as the voltage of rows is checked. The keypad’s schematic is fairly trivial and can be found in Appendix D.

The bounce is 4ms at make and 10ms at break[1]. Using a debouncing period greater than 4ms will ensure that the keypad is read properly.

For the Llama system, the top four keys are designated for selecting the files, the bottom right key is [OK], the key to the left of that is [CANCEL] and the bottom right key is [LOAD]. The remaining keys can be used for more file selects when the system is expanded to hold more configuration files.

## LCD

The LCD display will be used in conjunction with the keypad to enable the user to communicate with the LLAMA and to see the status of the LLAMA. The user will be able to see the current active file, select a new file, and run the configurater. Based on available supplies, the Optrex DMC16207 LCD has been selected for this project. The VHDL code is based on the code from the CDMA group, and modified to better suit our needs[2] .

The LCD requires a 2.5kHz clock, therefore, a clock divider was imple−mented.[2] . The enable signal is pulsed at a rate similar to the clock rate. This is accomplished by an inverted clock signal.

To initialize the LCD Display, the steps in Table 5 were used: (thanks to the CDMA group for this!)[2]

| RS | R/W | Data | Time Held | Purpose |
|---|---|---|---|---|
| 0 | 0 | 00000000 | 7 cycles (2.8ms) | Allow LCD time to warm up |
| 0 | 0 | 00110000 | 15 cycles (6ms) | Set display to 1 line, 8 bit data interface |
| 0 | 0 | 00001000 | 1 cycle (0.4ms) | Turn display off |
| 0 | 0 | 00000001 | 4 cycles (1.6ms) | Clear display |
| 0 | 0 | 00000110 | 1 cycle (0.4ms) | Set display to increment on write, no shift |
| 0 | 0 | 00001111 | 4 cycles (1.6ms) | Move cursor to home position |

*Table 5: LCD Initialization*

The messages have all been stored in the internal ROM and are selected based on the message code received from the system. By saving the messages in the ROM the number of logic cells utilized by the UI were cut in half.

# EXPERIMENTS

## CONFIGURING WITH RBF

Getting the RBF configurater working required a bit of experimentation. Since there was little documentation on the UP1 board − no schematics or description of operation − the jumper settings were determined by trial and error. For configuring the target board using RBF files, the correct setting is the same as for configuring the Flex10K only (see UP1 documentation)

Even with the correct jumper settings, configuring the target board seemed very hit and miss − when it was working, it would work for a long time. When it wasn't working, it wouldn't work at all. Eventually, it was deter−mined that there was a great deal of cross−talk between DCLK and Data on the RBF configuration interface. Keeping these wires as far apart as the wire lengths permit, and not allowing them to ever cross, allows programming to nearly always succeed.

## UI

The LCD display was originally designed with the messages being gener−ated combinationally. The UI took up over half of the total available logic blocks with only 4 messages. A redesign to use the EABs as a PROM to store the messages reduced the UI module to approximately a third of the total available logic block with 10 messages.

## JBC PROGRAMMING

The original design of the project called for JBC files to be used to program the target device instead of RBF files. The advantage of JBC files is

that they support a wide variety of possible target devices and are an industry standard[10]. Unfortunately, they are bytecode for a virtual machine that actually does the programming/configuration. VHDL code to interpret the JBC file was started and reached slightly over 1000 lines. It could follow and decode the op−code part of the file. However, being a virtual machine, the data part of the file was not accessed in a way that was easy to convert to hardware and it was determined that a JBC interpreter would not fit in our available number of logic blocks, even as a stand−alone, externally controlled chip.

The RBF file takes the configuration system to the other extreme. It only supports a narrow range of Altera devices and does not require any inter−preting. The data is already formatted suitably for the target device and simply needs to be pulled off of a storage device and streamed to the target with appropriate control signals.

# REFERENCES

[1] Keypad data sheet

[2] Smith, Jessamyn (CDMA Group) (2000). *Optrex DMC16207 LCD Driver* [Online]. Available: http://www.ee.ualberta.ca/~elliott/552/studentAppNotes/2000_f/ inte−facing/lcd/ [2001, February 8].

[3] Bensler, Tim and Eric Chan. (1999). *RS232 Serial Port* [Online]. Available: http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1999_w/RS232/ [2001, February 1].

[4]Airborn Electronics, *RS232 Connections, and wiring up serial devices* [Online] Available: http://www.airborn.com.au/rs232.html

[5] ARC Electronics. *RS232 Data Interface* [Online]. Available: http://www.arce−lect.com/rs232.htm [2001, February 1].

[6] *Caplan, Stephen (1998). Ethernet (IEEE 802.3) Overview* [Online]. Available: http://www.ee.ualberta.ca/~elliott/ee552/studentAppNotes/1998f/ethernet/ethernet.html [2001, February1].

[7] Gilbert, H. (1995). *Basic Tutorial on 10 Mbps Ethernet* [Online]. Available: http://pclt.cis.yale.edu/pclt/COMM/ETHER.HTM [2001, February 1]. © 1995 PCLT.

[8] Micro Linear Devices. (2000). *ML2652/ML2653 10Base−T Physical Interface Chip (Datasheet)* [Online]. Available: http://www.microlinear.com/DS/DS2652_53−01.pdf [2001, February 3]. © 1998 Micro Linear.

[9] Cheung, Eric, Felicia Cheng, David Li and Tim Kwan. (2000). *SRAM Interfacing Basics* [Online]. Available: http://www.ee.ualberta.ca/~elliott/ee552/studen−tAppNotes/2000_w/interfacing/sram_basics/sram.html [2001, February 10].

[10] Altera Corporation (2000). *JAM STAPL Player Source Code ver. 2.1* (see ref 12)

[11]Altera Corporation. (2001). *JAM* [Online]. Available: http://www.jamisp.com [2001 February 13].

[12]Altera Corporation. (2000). *Configuring APEX 20K, FLEX 10K & FLEX 6000 Devices, ver. 1.03* [Online]. Available: http://www.altera.com/literature/an/an116.pdf [2001, March 17].