The Arctic Switch Fabric Control Interface System

by

Elizabeth F. Y. Ogston

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Computer Science and Engineering and
Master of Engineering in Electrical Engineering and Computer Science
at the Massachusetts Institute of Technology

February 1996

© Massachusetts Institute of Technology 1996. All rights reserved.
The Arctic Switch Fabric Control Interface System

by

Elizabeth F. Y. Ogston

Submitted to the Department of Electrical Engineering and Computer Science
on February 9, 1996, in partial fulfillment of the
requirements for the degrees of
Bachelor of Science in Computer Science and Engineering and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis describes the design and implementation of ANCIS, a control interface program for an Arctic router network. ANCIS creates a simple read/write interface to a set of registers located in each Arctic router chip of the network. These registers control router configuration, keep track of a variety of statistics and errors, and can be used for placing test signals directly on the router pins. They are accessed through several layers of protocol using a JTAG scan ring. ANCIS will be used for configuration and testing of the Arctic Switch Fabric Network in the StarT-NG, for testing individual Arctic routing chips, and for testing 4 Arctic network boards.

Thesis Supervisor: George A. Boughton
Title: Principle Research Engineer, MIT Laboratory for Computer Science
Acknowledgments

Thank you Andy.
## Contents

1  Introduction .......................................................... 7

2  ANCIS Design ......................................................... 15
   2.1  Design Outline .................................................. 15
   2.2  Script Interface ................................................ 16
   2.3  Arctic Interface ................................................ 16
   2.4  Extest, Arcintest, and Low Level Scan ......................... 19
   2.5  JTAG Board Low Level Control ................................. 21

3  ANCIS Implementation ............................................. 22
   3.1  Scripts .......................................................... 22
   3.2  Arctic Interface ................................................ 24
   3.3  Extest, Arcintest, and Low Level Scan ........................ 31
   3.4  JTAG Board Low Level Control ................................. 32

4  ANCIS Testing and Use ............................................. 34
   4.1  Example ........................................................ 35

5  Future Work .......................................................... 41

6  Conclusion ............................................................. 43

A  ANCIS Code ........................................................... 45
   A.1  ancis.h ............................................................ 46
   A.2  data_stream.h .................................................... 47
List of Figures

1-1 Block diagram of the StarT-NG ...................................... 8
1-2 A StarT-NG Node ....................................................... 9
1-3 An 8 leaf sub-network of the switch fabric fat tree ................. 9
1-4 The Arctic Router ..................................................... 10
1-5 The Registers within an Arctic .................................... 11
1-6 Boundary and Low Level Mode Scan Rings .......................... 12
1-7 The JTAG Scan Ring .................................................. 12
1-8 The JTAG Controller Board ......................................... 13
1-9 ANCIS Structure .................................................... 14

2-1 The Arctic JTAG Interface .......................................... 18
2-2 The JTAG TAP Controller .......................................... 20

3-1 A Sample Script File ................................................ 24
3-2 Arctic Interface Functions .......................................... 25

4-1 Verilog/ANCIS Interaction .......................................... 35
4-2 Script File Used in the Text Example ................................ 36
4-3 Verilog Grwaves Status Window .................................... 40
Chapter 1

Introduction

The Arctic Network Control Interface System (ANCIS) is a set of programs designed to allow easy access to configuration, statistics, error, scan, and control registers in the Arctic Switch Fabric Network of the StarT-NG. This information is stored within the Switch Fabric Network, and is accessed through an IEEE standard JTAG scan ring. ANCIS is concerned with hiding the complexity of the JTAG interface by providing a simple read/write register interface to the StarT-NG. In addition, ANCIS can be used for testing single Arctic routers, and 4 Arctic boards.

StarT-NG is a joint project between MIT’s Computation Structures Group and Motorola to develop a message-passing parallel machine using a Motorola 620 based commercial SMP. The StarT-NG consists of eight SMP sites, each with four processors. (Fig 1-1) These processors are connected by the Switch Fabric Network, a fat tree of 4x4 routers. The network connects to the processors through a Network Interface Unit on each processor’s L2 Cache Interface. (Fig 1-2)

The Arctic Switch Fabric Network is a 32 leaf fat tree constructed from 16 circuit boards with 4 Arctics on each board. Figure 1-3 shows the fat tree configuration for an 8 leaf branch of the network. Arctic is a 4x4 200 MB/link/second packet-switched router developed by MIT. Each Arctic has an expanded IEEE standard JTAG interface used for accessing various registers within the chip. These registers are used for configuring the router, collecting statistics and error information, and for testing.
Figure 1-1: Block diagram of the StarT-NG
Figure 1-2: A StarT-NG Node

Figure 1-3: An 8 leaf sub-network of the switch fabric fat tree
The JTAG interface (Fig 1-4) to an Arctic consists of a clock (tclk), a Test Mode Select (tms) line, a Test Data In (tdi) line, and a Test Data Out (tdo) line. The tms line is used to cycle through a finite state machine, the states of which define whether the JTAG interface is idle or accessing a register. In some of these states the tdi and tdo lines are connected to a selected register within the Arctic and the register’s contents can be serially written or read. (Fig 1-5) There are two classes of registers accessible through the JTAG port. First are the JTAG Data registers which are selected using the JTAG Interface Instruction Register. Among the JTAG Data Registers is a Status and Data register. The Data part of this register can be one of a number of Arctic Data Registers. Which Arctic Data Register is accessed by the Data Register is determined by the value in the Access Operation Address Register. The Arctic User’s Manual [2] describes in detail how Arctic registers are accessed.

JTAG Data Registers also include two scan cell registers. The Low Level Mode Scan Register connects to a series of six scan rings that can be used to test the Arctic chips. The Boundary Scan Cells register allows bits to be input to or read from each of the Arctic’s pins. (Fig 1-6)

The Arctic chips in the Switch Fabric Network are connected into one large JTAG
Figure 1-5: The Registers within an Arctic
scan ring allowing all the chips to be controlled together. (Fig 1-7) In the scan ring the same clock and tms signals are sent to all the Arctics, and the tdo line of each Arctic in the ring is connected to the tdi line of the next. In this way registers on all the routers can be accessed though a single JTAG interface to the network. This interface is connected to a JTAG Controller Board (Fig 1-8) which consists of three FIFOs that accept and buffer the tms, tdi, and tdo bits. The Controller Board sits on the ISA bus of a Power PC. The ANCIS system runs on this computer and access the Controller Board FIFOs using memory mapped writes and reads.

ANCIS itself was designed with several layers of abstraction corresponding to the underlying hardware interfaces. (Fig 1-9) Within the StarT-NG the highest level is a set of C commands that can be called by a script interface written by Motorola. For testing purposes a different script interface was written that allows tests to be input to a single Arctic or to the 4 Arctic network board directly through the JTAG Controller
Board. The next layer of abstraction deals with the somewhat complicated protocol required by Arctic for accessing registers through the JTAG interface. Finally, the lowest abstraction layer masks the protocol required by the JTAG Controller Board.
Figure 1-9: ANCIS Structure
Chapter 2

ANCIS Design

The basic ANCIS structure consists of three layers of programs. At the highest level a script interface is used for writing test and configuration sequences. This is followed by the Arctic Interface which is a set of read/write register commands used by the script interface and accessed by Motorola’s script interface within the StarT-NG. Finally at the lowest level is a set of JTAG board commands used by the Arctic Interface to actually send bits to the network.

2.1 Design Outline

The basic ANCIS structure is shown in figure 1-9. The program layers depicted are:

Script Interface :
Reads testing script files and translates them to Arctic Interface commands.

Arctic Interface :
Translates commands from the Script Interface commands, or those received from the Motorola Script Interface, into bits to be sent to the controller board.

JTAG board low level control :
Runs the controller board. Sends bits from the Arctic Interface and deals with controller board commands and controller board status.
2.2 Script Interface

Script files are used to list a set of Arctic Interface Commands to be run using the command \texttt{run_script}. Within a script command arguments can either be listed or replaced with a reference to a separate file. Scripts can output log files, and compare log files to check files. Finally, scripts are made more general by allowing for variables that are replaced by command line arguments. Details are given in section 3.1.

2.3 Arctic Interface

The layering of the Arctic Interface corresponds to that of the Arctic Access Operations, Access Macros, and JTAG Transaction Macros described in the Arctic User's Manual [2]. The design is essentially the same as that described by Mark Nodine [5] for his proto-type Arctic Device Drive Design. However, it allows for the chain of Arctics rather than a single Arctic. It also includes instructions for extest and arcintest.

Upper Level

The upper level of the Arctic Interface consists of the commands accessible to the Script Interface. These include separate reads and writes for each register within Arctic, and resets for the JTAG port, and JTAG controller board. Arctic Registers are listed in figure 1-5.

Each register within Arctic has a different length and format, making separate read/write commands for each necessary. In addition, each read and write can be done on a selected set of Arctics in the scan chain, bypassing the others. Upper Level write commands accept the data to be written directly into the JTAG scan chain. Read commands output whichever bits are returned from the scan chain on the read. Thus it is up to the user to know the data format for the register being accessed.

Upper Level commands also take an argument defining which Arctics in the scan chain are to be operated on. Arctics that are not required for the command are put
into a bypass mode where the one bit bypass register is switched into the scan chain. The data argument given to the Upper Level commands must contain the bits for all of the Arctics being accessed, in scan chain order. However, ANCIS automatically inserts and removes the one bit bypasses. The format of data arguments is discussed in more detail in section 3.2.

**Arctic Access Operations**

Arctic Access Operations are used to issue reads and writes of the Arctic Data Registers. (Fig 2-1) Reads and writes to Arctic Data registers are two-phase operations. This is because the Arctic Data Register desired must first be selected by writing the Access Operation Address Register. This connects the Arctic Data Register to the AOS_and_Dreg (Arctic Operation Status and Data Register) path. Next, the operation desired must be written to the Access Operation Instruction Register. After a read or write has been called in this manner, some time later a status and returned data will appear in the AOS_and_Dreg. The read or write is not valid until a COMPLETED status is returned. There are however some registers that are guaranteed to be read or written immediately and thus do not return a status. These registers are the port/statistics control register, the system-reset control register, the error counters, and the buffer-free counters. A status check is performed by calling the get_status instruction which repeatedly reads the AOS_and_Dreg until a completed status is returned for all Arctics being accessed. The status check is placed at this level because any lower level would need to be passed the location of the status bits within the data chain for each Arctic being accessed, as well as a flag indicating whether status is required for the operation.

**Access Macros**

The Access Macros package Arctic Access Operations into JTAG Transaction Macros. The Arctic Mode, Access Operation Address, Access Operation Instruction, Status and Data, and Low Level Mode Scan Registers are all accessed by placing a select-register command in the JTAG Instruction Register. (Fig 2-1) This connects the
Figure 2-1: The Arctic JTAG Interface
selected register to the tdo and tdi lines so it can be read or written. Access Macros deal with this process of selecting the correct register to be read or written, and then actually accessing that register.

**Lowest Level: JTAG Transaction Macros**

All JTAG registers are accessed by cycling through the states in the JTAG TAP controller. (Fig 2-2) The tms line changes these states and the tdi line carries the data to be shifted into the registers in the Shift-DR and Shift-IR states.

This level creates the tms and tdi streams. It inserts bypass instructions into the instruction part of the tdi stream and inserts single bypass bits between the register data arguments. It also pulls out bypass bits while reading back the data and before returning it.

### 2.4 Extest, Arcintest, and Low Level Scan

The Extest and Arcintest instructions are used to access the boundary scan rings on the Arctics. The boundary scan register allows bits to be read directly from, or written to, the pins of Arctic. Additionally, it includes 16 extra bits for each input and output port since the links between Arctics run twice as fast as the routers themselves. The scan ring also contains the status control register. (Please see the Arctic Manual for more details.) Extest disconnects the pins from the rest of Arctic so that tests of the board interconnect can be run. Arcintest disconnects Arctic from the pins so tests of the Arctic circuitry can be run.

The Low Level Scan instruction is used to access the manufacturing test rings within Arctic. These are scan cells used for running test vectors generated from the HDL model of the chip. There are 6 manufacturing test rings, each corresponding to one bit in the Low Level Mode Scan Register. Each write to the Low Level Mode Scan Register writes one bit to each manufacturing test ring. Each ring is 525 bits long, and thus to do a scan the Low Level Mode Scan Register must be written 525 times before the system clock is cycled. Similarly, the boundary scan register is 338
Figure 2-2: The JTAG TAP Controller
bits long and the system clock must not be cycled while the register is being written. Because of this Extest, Arcintest and Low Level Scan use an additional set of JTAG Transaction Macros to allow them to pause in the Pause-DR state rather than the Run-Test-Idle state (which advances the system clock).

Since the Low Level Mode Scan Register and Boundary Scan Cells are JTAG Data Registers instead of Arctic Data Registers, Extest, Arcintest and Low Level Scan do not follow the layers of abstraction used by the rest of the Arctic Interface. Extest and Arcintest both deal directly with the Pause-DR JTAG Transaction Macros. Low Level Scan uses its own set of Access Macros as well as these JTAG Transaction Macros. (Fig 3-2)

2.5 JTAG Board Low Level Control

The JTAG board low level controller receives sets of tdi and tms bits from the Arctic Interface, sends them to the JTAG board, sends “go” commands at the appropriate times, and reads back the tdo data coming out of the board. Detail on how the JTAG controller board works can be found in Professor Moona’s paper [4].

The board_send_bits instruction treats the data it is given as a single set that can be sent to the JTAG board along with a “go” command. The JTAG board repeats the last bit in a set when it is idle and thus each set of bits must end in a pause state. In addition the JTAG board’s FIFOs are 4k bytes long so that each set of bits must be less than 4k bytes.
Chapter 3

ANCIS Implementation

ANCIS was implemented in C as a set of library functions. At the top level these are called by Motorola’s scripting interface. For testing, these functions are accessible through a testing script interface also implemented in C. At the lowest level ANCIS interacts with a simple device driver for the JTAG Controller Board.

3.1 Scripts

The following section describes the script interface used to test the Arctic chip and 4 Arctic board. Scripts are used to list and run a set of ANCIS commands. The output received when the commands are run can be written to a log file for examination. Moreover, these log files can be checked against compare files containing the correct outputs so that any errors are automatically caught. Scripts are made more general by the use of command line arguments that can replace <arg#> strings within the script file, and the ability to replace ANCIS command arguments with the name of a file containing the correct arguments.
Scripts are run using the command:

```bash
run_script<file_name> <arg1> <arg2> .... :
```

Tells the Interpreter to run the commands listed in the given file_name. Commands listed here can be any of the Direct Arctic Interface commands. These are described in section 3.2.

The `ar1`, `arg2`, etc. arguments to `run_script` can be used to replace the text `<arg#>` within the script with the text given in the argument.

ANCIS commands should be listed within scripts as described here. Each command in a script should be on a separate line followed by a list of the command's arguments, also each on separate lines. Alternatively, argument lists can be replaced by the keyword `FILENAME` followed by the name of a file containing the replaced arguments. Details of each ANCIS command and the arguments they take are given in section 3.2.

For each command given, data must be provided separately for each Arctic in the scan chain. Arctics are labeled by scan chain number from 0 to 63. If data is not provided for an Arctic that Arctic is bypassed while the command is run. Data for individual Arctics should be listed as the keyword `ARCTIC` followed by an Arctic number on a line by itself immediately preceding the actual data. Data for Arctics should be listed in increasing Arctic number order. For commands that do not take inputs, the Arctics operated on can be defined by listing just the `ARCTIC <Arctic#>` lines.

In addition to the ANCIS commands, there are three log commands allowed in a script:

```bash
start_log<filename>:
```

Writes the command’s outputs to a log file. If this command hasn’t been called outputs are ignored.

```bash
end_log:
```

Must be called sometime after starting a log, and before opening another.

```bash
compare_logs <log_file> <comparison_file> <results_file>:
```

Can be used when running a test to see if the correct data was received. Writes any discrepancies to the results file.

A sample script file is shown in Figure 3-1.
3.2 Arctic Interface

This section describes the details of the functions used in each layer of the Arctic Interface. All functions are implemented in C. Figure 3-2 gives a block diagram of the functions and their interaction.

The JTAG interface to the Arctic Scan Chain receives data one bit per clock cycle on the tms and tdi lines. Data to be written into a register is given to the ANCIS upper level functions as a set of bits. ANCIS adds bypass bits and padding to be entered on the tdi line while the JTAG controller cycles states. It also creates the corresponding set of bits to be entered on the tms line.

These sets of bits are passed around ANCIS as data_streams; structs used to abstract the idea of sets of data in which bits need to be inserted, changed, or removed. Data_streams are implemented as a list of integers that contain the bits of the stream and a size variable that keeps track of the stream’s overall size. Data_stream structs are accessed through a set of functions for data_stream creation, deletion, and bit or
Figure 3-2: Arctic Interface Functions
byte reads and write.

**Upper Level**

Upper Level functions are listed below. Upper Level functions are used to access each register within the Arctics. In addition there are reset commands for the JTAG ports and the JTAG controller board. Register data arguments are given to, and returned by the Upper Level as data.streams. These are written as if the Arctic chain only contained those Arctics that are operated on, bypass bits are not included. The int Arctics[] argument defines which Arctics in the scan chain a command is to operate on. Arctic[] is a list of integers the size of the scan chain. Arctics[i] contains a 0 if Arctic i in the scan chain should be bypassed, and a 1 if Arctic i should execute the function.

Upper Level functions include:

```c
void read_psc (struct data_stream *data, int Arctics[]);
    // Reads the port/statistics control registers for the specified Arctics.
    inputs: arctics
    outputs: data out

void write_psc (struct data_stream *data, int Arctics[]);
    // Writes the given data into the port/statistics control registers for the specified Arctics.
    inputs: data
    outputs: acknowledgement

void reset (int Arctics[]);
    // Resets the specified Arctics. Clears the error counters and disables all input and output ports.
    inputs: arctics
    outputs: acknowledgement

void reset_controller ();
    // Resets the JTAG controller board and all of the Arctics’ TAP controllers.
    inputs: none
    outputs: acknowledgement
```
void port_stat (int port, enum STATS stat, struct data_stream *data,
    int clear, int Arctics[]);
    Reads the given statistics register for the given port in the specified Arctics.
    Clears the statistic register after reading if clear is set to 1.
    inputs: port, statistic, arctics
    outputs: data out

void port_stats (int port, struct data_stream *data[6], int clear,
    int Arctics[]);
    Reads all the statistics for a given port in the specified Arctics. Clears the
    statistics read if clear is set to 1.
    inputs: port, arctics
    outputs: data out

void all_stats (struct data_stream *data[4][6], int clear, int Arctics[]);
    Reads all statistics for all ports in the specified Arctics. Clears the statistics
    read if clear is set to 1.
    inputs: arctics
    outputs: data out

void clear_stats (int Arctics[]);
    Clears all the statistics counters on all ports in the specified Arctics.
    inputs: arctics
    outputs: acknowledgement

void errors (struct data_stream *data, int clear, int Arctics[]);
    Reads the error counters for the specified Arctics. Clears the counters after
    reading them if clear is set to 1.
    inputs: arctics
    outputs: data out

void bufs_free (struct data_stream *data, int Arctics);  
    Reads the buffer free counters for the specified Arctics.
    inputs: arctics
    outputs: data out

void cfg_read_port (int port, struct data_stream *data[5],
    int Arctics[]);
    Reads the configuration registers for the given port in the specified Arctics.
    inputs: port, arctics
    outputs: data out

void cfg_write_port (int port, struct data_stream *data[5],
    int Arctics[]);
    Writes the configuration registers for the given port in the specified Arctics.
    inputs: port, data in
    outputs: acknowledgement
void cfgwrite_all (struct data_stream *data[5], int Arctics[]);
Writes the configuration registers for all ports in the specified Arctics with the
same data.
inputs: data in
outputs: acknowledgement

void change_mode (enum MODES mode, int Arctics[]);
Changes the mode in the given Arctics to mode. Changing to configuration
mode requires that all input and output sections of the chip be disabled for
480ns prior to entering configuration mode. It is the user’s responsibility to
make sure this is done.
inputs: mode, arctics
outputs: acknowledgement

void extest (FILE *data_in, FILE *data_out, int size, int Arctics[]);
Runs an extest sequence on the given Arctics using the data in data_in and
writing the output to data_out.
inputs: sequence size, data in
outputs: data out

void arcintest (FILE *data_in, FILE *data_out, int size, int Arctics[]);
Runs an arcintest sequence on the given Arctics using the data in data_in and
writing the output to data_out.
inputs: sequence size, data in
outputs: data out

void do_llm_scan (FILE *data_in, FILE *data_out, int Arctics[]);
Runs a low level scan on the given Arctics using the data in data_in and writing
the output to data_out.
inputs: data in
outputs: data out

Note: extest, arcintest and low level scan use a different set of lower level commands
and thus are described in separate sections.

Arctic Access Operations

Arctic Access Operations are used to issue reads and writes to the Arctic Registers.
Since these are split phase operations, the get_status instruction must be called after
calling the read_reg instruction to retrieve the requested data, or after calling the
write_reg instruction to confirm the write. However, there are some registers that are
guaranteed to write quickly and do not use the get_status instruction after a write. These are the port/statistics control register, the system-reset control register, the error counters, and the buffer-free counters.

Arctic Access Operation functions include:

```c
void read_reg (enum REGISTERS address, int Arctics[]);
Sets up the register to be read. Actual reading is done by a call to get_status.
```

```c
void write_reg (enum REGISTERS address, struct data_stream *ds, int data_len, int Arctics[]);
Writes the data in ds to the given register. The write isn’t completed until a call to get_status returns COMPLETED. (unless you are writing one of the fast registers listed above)
```

```c
int get_status (struct data_stream *ds, int data_len, int Arctics[], int status?);
Continues to read the AOSandDreg until a COMPLETED or error status is returned by all the Arctics. Returns the status. Setting the status? argument to 0 indicates that the get_status is being called on one of the fast registers and the status does not need to be checked.
```

Access Macros

Access Macros are used to package Arctic Access Operations into JTAG Transaction Macros. These select the correct register to be accessed, then do the access.

Access Macro functions include:

```c
void write_AOAreg (enum REGISTERS address, int Arctics[]);
Writes the AOAreg with address for each Arctic specified.
```

```c
void write_AOSandDreg (struct data_stream *ds, int data_len, int Arctics[]);
Writes ds to the AOSandDreg.
```

```c
void write_AOIreg (enum INSTRS instruction, int Arctics[]);
Writes the AOI reg with instruction for each Arctic specified.
```

```c
void read_AOSandDreg (struct data_stream *ds, int Arctics[], int datalen);
Reads the AOSandDreg into ds.
```

```c
void write_Mreg (short mode, int Arctics[]);
Writes the given mode for each Arctic specified.
```
Lowest level: JTAG Transaction Macros

JTAG Transaction Macros create the tms line bits used to cycle through the JTAG TAP controller states. They also pad the tdi line so that the desired data is entered in the correct JTAG controller states.

JTAG Transaction Macros include:

```c
void loadJTAGIR (enum INSTRS instr, enum INSTRS alternate, int Arctics[]);

Loads the JTAG instruction Register with instr, ignores returned bits. The instr argument is a single instruction that is repeated for each Arctic to be operated on. All other Arctics are given the instruction named by alternate. Alternate will be bypass for most instructions and clamp for arcintest and extest instructions.
```

TAP Controller States:
- Run-Test-Idle
- Select DR Scan
- Select IR Scan
- Capture IR
- Shift IR
- Exit1 IR
- Update-IR
- Run Test Idle

```c
void load_JTAGDR (struct datastream *ds, int Arctics[],
int data_len);

Loads the JTAG Data Register with ds, puts returned bits into ds. The ds argument does not contain any data for Arctics to be bypassed. Thus load_JTAGDR puts in bypass bits for all bypassed Arctics when sending ds to the Controller Board and removes them again before returning the data received.
```

TAP Controller States:
- Run-Test-Idle
- Select DR Scan
- Capture DR
- Shift DR
- Exit1 DR
- Update-DR
- Run Test Idle

Separate JTAG Transaction Macros may need to be written for specialized func-
tions. Extest, Arcintest and Low Level Scan use an additional set of JTAG Transaction Macros. These are described in section 3.3.

3.3 Extest, Arcintest, and Low Level Scan

As described in section 2.4 Extest, Arcintest and Low Level Scan are accessed in a different manner than the other registers on the Arctics, and thus use different sets of commands.

Extest, Arcintest, and Low Level Scan Upper Level Commands include:

```c
void extest (FILE *data_in, FILE *data_out, int size, int Arctics[]);
   1. Writes the extest instruction to all Arctics specified in Arctics[], writes the clamp instruction to all others.
   2. Feeds in the data in the data_in file. Size indicates how many sets of extest scans there are. Writes the data returned into the data_out file.

void arcintest (FILE *data_in, FILE *data_out int size, int Arctics[]);
   1. Writes the arcintest instruction to all Arctics specified in Arctics[], writes the clamp instruction to all others.
   2. Feeds in the data in the data_in file. Size indicates how many sets of arcintest scans there are. Writes the data returned into the data_out file.

void do_llmscan (FILE *data_in, FILE *data_out, int Arctics[]);
   Does a complete low level scan (525 cycles) using the data in the file data_in and writing the data returned to the file data_out.
```

Low Level Scan Access Macros

```c
void set_up_llmscan (struct data_stream *ds, int arctics[]);
   Writes the JTAG Instruction Register and reads the first scan. Writes the data that is returned into ds.

void llm_scan (struct data_stream *ds, int arctics[]);
   Cycles the system clock. Reads ds into the scan rings, puts returned data into ds.

void end_llmscan ();
   Returns the TAP controller to Run-Test-Idle.
```
Extest, Arcintest, and Low Level Scan JTAG Transaction Macros

```c
void load_JTAGDR2_beg (struct data_stream *ds, int Arctics[],
    int data_length);
```

Does first scan and ends in the Pause-DR state. Fills in bypass bits before scanning and removes them again before returning data.

```
TAP controller states:
    Run Test Idle
    Select DR Scan
    Capture DR
    Shift DR
    Exit1 DR
    Pause DR
```

```c
void load_JTAGDR2 (struct data_stream *ds, int Arctics[],
    int data_length);
```

Cycles the system clock then does the next scan. Fills in bypass bits before scanning and removes them again before returning data.

```
TAP controller states:
    Pause DR
    Exit2-DR
    Update_DR
    Run Test Idle
    Select DR Scan
    Capture_DR
    Shift DR
    Exit1-DR
    Pause DR
```

```c
void load_JTAGDR2_end ();
```

Returns the TAP controller to Run-Test-Idle.

```
TAP controller states:
    Pause DR
    Exit2-DR
    Update_DR
    Run Test Idle
```

### 3.4 JTAG Board Low Level Control

There are two functions that allow access to the JTAG Controller Board. The first, `board_send_bits` is used to send a set of accesses to the Controller Board and receive
the data returned by those accesses. The second is used to reset the board.

```c
int board_send_bits (struct data_stream *tdi, struct data_stream *tms);
   1. Sends the given tms and tdi bits out to the board. Bits must end in a pause
      state and begin in the state the last instruction left them in.
   2. Sends a go instruction to the board.
   3. Waits until the board is finished, and reads the data from TDO into the tdi
      data_stream.

int board_reset (){
   1. Resets the JTAG board
   2. Writes two idle bytes and reads one to fill up the byte left in the serial to
      parallel converter and the bit left in the tdi buffer.
Chapter 4

ANCIS Testing and Use

A simplified version of the script interface has been implemented. In addition, the Arctic Interface Commands to read and write the statistics, mode, and configuration register commands have been implemented and debugged. The current system is being tested using a Verilog model of the Arctic chip as shown in Fig. 4-1. The testing setup is somewhat convoluted since Verilog provides no easy way of communicating with other programs while running a simulation. However ANCIS needs ongoing communication with the chip model since it must know the results of each get_status instruction before generating its next set of commands.

To allow ANCIS to interface to the Verilog model of Arctic the JTAG controller board functions are simulated by reading and writing to a set of files. There is one file for each of the tms and tdi fifos. ANCIS generates sets of tms and tdi bits and writes them to these files. The data is then read into memories within the Verilog model. Verilog enters the bits in the tms and tdi memories into the Arctic Verilog model using a modified JTAG interface of an Arctic test program written by Richard Davis [3]. Tdo bits returned from the Arctic model are written to another memory, which is written to a file which ANCIS reads after Verilog has completed each stage of the simulation.

Control between ANCIS and Verilog is maintained with another set of files. Once ANCIS has generated a set of tms and tdi bits it increments a counter and indicates the number bytes in the fifo files in a signal file. Verilog continuously reads this file
until the counter is increased, at which point it reads the fifo files and begins it’s simulation. Meanwhile ANCIS waits for a Verilog signal file to be written. Verilog creates this file when it’s simulation is finished, ANCIS erases it again, and goes on to generate the next set of fifo bits.

4.1 Example

Figure 4-2 shows a script file that runs the startup procedure for a single Arctic then reads the port statistic control register to show a read operation. The following section goes over the ANCIS debugging log file for this script to demonstrate how ANCIS works.

Configuring an Arctic involves five steps. First the JTAG controller is initialized using the JTAG reset line. This is done by the JTAG Controller Board when it is reset. Second Arctic itself is reset using the reset command which writes to the system-reset register. Third Arctic is put into configuration mode and the configuration registers are written. Fourth, the mode it changed back to normal mode. Finally, the statistics
reset_controller
reset
ARCTIC 0
change_mode
config
ARCTIC 0
cfg_write_all
reg1
ARCTIC 0
0ffbdff8
reg2
ARCTIC 0
040f7bfe
reg3
ARCTIC 0
00000000
reg4
ARCTIC 0
00000000
reg5
ARCTIC 0
000300ff
change_mode
normal
ARCTIC 0
clear_stats
ARCTIC 0
write_psc
ARCTIC 0
333333
read_psc
ARCTIC 0

Figure 4-2: Script File Used in the Text Example
counters are cleared. This is done by reading the clear statistics registers. In the example the port statistics control register is then written to enable the inputs and outputs.

In the example the ANCIS level that issued each command is indicated by the indentation. Script commands are not shown since they mirror the Upper Level of the Arctic Interface. Arctic Interface Upper Level commands and their arguments are not indented. Arctic Access Operations are indented three spaces, Access Macros are indented six spaces, JTAG Transaction Macros are indented nine spaces, and controller board commands are indented twelve spaces. Commands at each Arctic Interface level are diagramed in figure 3-2. Figure 2-1 diagrams the different registers within Arctic.

The symbol < .... > indicates that lines have been deleted from the example to make it readable. Most of the commands are only shown through the Arctic Access Operations; read_reg, write_reg, and get_status which are used to access the Arctic Registers. Notice that the the get_status command is used for reading after the read_psc command and for checking status after writing the configuration registers, but is not required after the write_psc command.

The write_psc command demonstrates how the Arctic Access Operations are broken down into Access Macros which write the sequence of JTAG registers required to access an Arctic Register. The write AOReg within the read_psc command is used to show how Access Macros are further broken down into a sequence of JTAG Transaction Macros which create the sequence of load JTAGIR and load JTAGDR instructions required to write a JTAG register. Finally the write AOIReg instruction within the read_psc command shows the actually bits written to the board by the JTAG Transaction Macros. It also demonstrates the board go command which sends the bits in the board’s FIFOs to Arctic.

Figure 4-3 is the grwaves window of the Verilog simulation after the example has been run showing that the registers within the chip are written correctly.
reset_controller
  board_write addr:122 data:2

reset
  write reg addr: 341 data:0
    < .... >

change_mode
config
  write Mreg: 0
    < .... >

cfg_write_all
  data =
    0001111111111101111011111111111111000
    011111111111011111000000100000
    00000000000000000000000000000000
    00000000000000000000000000000000
    11111111100000000110000000000000

write reg addr: le0 data:0001111111111011110111111111000
< .... >

get status
< .... >

write reg addr: le1 data:0111111111111011111000000100000
< .... >

get status
< .... >

write reg addr: le2 data:00000000000000000000000000000000
< .... >

get status
< .... >

write reg addr: le3 data:00000000000000000000000000000000
< .... >

get status
< .... >

write reg addr: le4 data:11111111100000000110000000000000
< .... >

get status
< .... >

change_mode
normal
  write Mreg: 1
    < .... >
Writing an Arctic Register is done in three stages
First, the Arctic Operation Address Register is written to indicate which Arctic Register is being accessed
Second, the data is written to the Arctic Operation Status and Data Register
Finally, a “write” command is placed in the Arctic Operation Instruction Register

Accessing a JTAG register is done in two stages
First, the JTAG Instruction Register is written to indicate which JTAG Register is being accessed
Next, the JTAG Data Register is written with the data
Writing the JTAGIR or JTAGDR registers consists of sending a set of bits for the TDI and TDO lines to the Controller Board, one byte at a time
An additional byte of TDI and TMS bits must be sent to fill the serial to parallel conversion space in the Controller Board
The go command tells the controller board to send the bits in its FIFO’s to Arctic.

For testing the go command is simulated by Verilog.

A read of the controller board status tells ANCIS when the board’s FIFO’s are empty.

The Controller Board must be told to stop after each go command.

ANCIS can then read the data returned in the Controller Board’s TDO FIFO.

---

The go command tells the controller board to send the bits in its FIFO’s to Arctic.

For testing the go command is simulated by Verilog.

A read of the controller board status tells ANCIS when the board’s FIFO’s are empty.

The Controller Board must be told to stop after each go command.

ANCIS can then read the data returned in the Controller Board’s TDO FIFO.

---

The go command tells the controller board to send the bits in its FIFO’s to Arctic.

For testing the go command is simulated by Verilog.

A read of the controller board status tells ANCIS when the board’s FIFO’s are empty.

The Controller Board must be told to stop after each go command.

ANCIS can then read the data returned in the Controller Board’s TDO FIFO.
Chapter 5

Future Work

Enough of ANCIS has been implemented to demonstrate that it will work, however there are still parts left unwritten. All of the lower level functions to read and write Arctic Registers are in place so the remaining read and write Upper Level functions should be easy to implement. The scan functions (do.llm.scan, extest and arcintest) have been sketched out, but still need to be fully implemented and tested.

There are several features that could be added to ANCIS as well. Upper Level functions can easily check that the length of the data arguments passed to them, and create an error if it does not match the length of the register to be read or written. The scripting language does not yet contain all the desired features. Currently none of the log file commands, or the commands to substitute in text or files to a script file have been written.

A future version of ANCIS will want to change where the board.go commands are issued. Currently they are issued after each JTAG Transaction Macro. This is a mistake, at the JTAG Transaction Macro level a board.go command only needs to be issued after each load.JTAGDR command. Board.go commands really only need to be issued after each get.status command or when the Controller Boards FIFO’s are filled. However, this would necessitate that ANCIS to keep track of the number of bytes sent to the FIFOs and require that the get.status command has access to the Controller Board to issue the board.go commands. Moreover, as the Controller Board is not yet available, the exact controller board interface has yet to be tested.
Future versions of ANCIS could also keep track of the data in the JTAG Instruction Registers and Arctic Operation Instruction Registers for each chip and not rewrite them if the preceding command used the same instruction as the current one.

Finally, the current Verilog model only allows testing of single Arctic length scan chains. Longer scan chains have been checked by hand. However, to make sure bypass instructions are issued correctly a new Verilog model with several Arctics in the scan chain needs to be written.
Chapter 6

Conclusion

ANCIS has been architected so that it provides a simple and usable read/write interface to registers in the Arctic Switch Fabric Network for testing procedures and the StarT-NG. At the lowest level it interfaces to the JTAG Controller Board using commands to a specified device driver. Moreover, the architecture allows ANCIS to be implemented easily in a clean, modular fashion.

Enough of the ANCIS read, write and reset commands have been written to initialize an Arctic. These commands have been tested on a Verilog HDL model of the chip to demonstrate their successful performance. In addition the basic design of the remaining commands has been laid out to facilitate future implementation.

The implementation work on ANCIS will be continued next semester by another student following the architecture specified in this thesis. The completed ANCIS system will be used in March to test stand alone Arctic chips, and subsequently to set up the Arctic Switch Fabric Network when it is completed.
Bibliography


Appendix A

ANCIS Code

The Appendix contains the code written so far for the ANCIS system. These files are all currently in the /home/jj/elth/ANCIS/code2/ directory. Verilog simulation code is in the /home/prj4/arcticdf/arctic20testelth/ directory.

Arctic files include:

Scripting Language files: scripts.c, scripts.h
Upper Level commands: upper_level.c, upper_level.h
Arctic Access Operations: arctic_access_ops.c arctic_access_ops.h
Access Macros: access_macs.c access_macs.h
JTAG Transaction Macros: jtag_trans.c jtag_trans.h
Controller Board interface: board.c board.h
Verilog simulation interface: sim.c sim.h
Datastream commands: data_stream.c data_stream.h
A.1 ancis.h

/* header file for all parts of the ANCIS program and scripts */

#include <stdio.h>
#include <stdlib.h>

#define NA 1
/* number of arctics in the scan chain */
#define TEST 1

#include "data_stream.h"
#include "sim.h"
#include "board.h"
#include "jtag_trans.h"
#include "access_macs.h"
#include "arctic_access_ops.h"
#include "upper_level.h"
A.2 data_stream.h

/*header file for data_stream abstraction
Elth 11/28/95*/

#include <math.h>

enum bit {ZERO, ONE};

struct data_stream
{
    int size;
    unsigned short *data;
};

/*data kept in 16 bit chunks, space allocated when the data_stream is created
bits are labeled starting at 0*/

struct data_stream *create_data_stream(int size);
/*returns an empty data stream with size space in it*/
/*size = number of bits*/

void delete_data_stream(struct data_stream *ds);
/*deletes the given data stream, deallocates memory used*/

void set_bit(struct data_stream *ds, long bitnumber, enum bit value);
/*sets the given bit to value*/

enum bit get_bit(struct data_stream *ds, long bitnumber);
/*returns the value of given bit*/

int set_byte(struct data_stream *ds, long start, short byte);
/*adds given byte to data stream beginning at start*/
/*start is the bit number, not the byte number*/

int get_byte(struct data_stream *ds, long start, short *byte);
/*returns the next byte in the stream. successive get_next_byte commands will return successive bytes
returns 0 if there are no bytes left
returns number of valid bits if a byte is returned*/

int get_ds_length(struct data_stream *ds);

struct data_stream *append_streams(struct data_stream *ds1,
                                 struct data_stream *ds2);
/*returns a new data stream that consists of ds1 followed by ds2
assumes both streams are full*/

/*more functions, to be taken out later??*/
struct data_stream *parse_to_ds(int data1,
    int size1,
    int arctics[NA]);
/*creates a data stream with data1 for each specified
    arctic and no data for each bypasses*/

int add_bits(struct data_stream *ds, int num_bits, int start,
    int bits);  
/*add num_bits number of bits to ds starting at start (bit number)
    from the number bits*/

/* for debugging */
void print_stream(struct data_stream *ds);
void bin_print(int digits, int number);
A.3  data_stream.c

/* File containing all data stream functions
data_stream.h contains declarations.

Elth 11/28/95
*/

/* data streams are abstractions for the bit streams to be
fed into the scan chain. At upper levels they are used to
pass around data to be read into, or read out of the arctic
registers, at lower levels they contain control bits as well
data streams can be written and read bit by bit, or byte by
byte. Their correct length must be defined when they are
created. Functions are also provided to get a streams length,
create and destroy streams, and combine streams.
*/

#include "ancis.h"

void bin_print(int digits, int number)
{
    int i;
    for(i=0; i<digits; i++)
        printf("%d", ((number & (1<< i)) >> i));
    printf("\n");
}

struct data_stream *create_data_stream(int size)
/*creates an empty data_stream of size size. returns a pointer to it
empty data streams have 0's in all their data bytes*/
{
    struct data_stream *ds;
    int i;
    if(
        ds = (struct data_stream *)malloc(sizeof(struct data_stream))) == NULL)
        printf("uh oh, can't get enough memory \n");
    ds->size = size;
    size = size/16+1;
    if(
        (ds->data = (short *)calloc(size, sizeof(short))) == NULL)
        printf("uh oh, can't get enough memory \n");
    for(i=0; i<size; i++)
        ds->data[i] = 0;
    return(ds);
}
void delete_data_stream(struct data_stream *ds)
/*deletes the given stream, frees up memory*/
{
    free(ds->data);
    free(ds);
}

void set_bit(struct data_stream *ds, long bit_number, enum bit value)
/*sets the given bit address to the given value*/
{
    int i, j;

    if(bit_number >= ds->size)
    {
        printf("attempt to set bit outside data stream: %d, %d\n",
                bit_number, ds->size);
        return;
    }

    if (value == ZERO)
        ds->data[bit_number/16] = ds->data[bit_number/16] & ~(1 << (bit_number%16));
    else
        ds->data[bit_number/16] = ds->data[bit_number/16] | (1 << (bit_number%16));
}

enum bit get_bit(struct data_stream *ds, long bit_number)
/*returns bit bit_number from data_stream*/
{
    return((ds->data[bit_number/16] & (1 << (bit_number%16))) >> (bit_number%16));
}

int set_byte(struct data_stream *ds, long start, short byte)
{
    /*adds given byte to data stream begining at start*/
    /*start is a bit number, not a byte number*/
    /*if the byte goes over the length of the stream,
     truncates the number by filling in to the end of the stream*/
    /*returns number of bits set*/

    int word_num;
    int offset;
    int mask;
    short new_byte;

    /*if byte outside the data_stream*/
    if (start > ds->size)
    {
        printf("attempt to add byte to full stream: %d, %d\n",

/* find the offset from the beginning of a word and the word number within the stream */
offset = start % 16;
word_num = start / 16;

/* if the entire byte falls within one word */
if(offset <= 8)
{
    /* clear the correct byte within the word */
    mask = ~((255 << offset));
    ds->data[word_num] = ds->data[word_num] & mask;
    /* splice in the new byte */
    byte = byte << offset;
    ds->data[word_num] = ds->data[word_num] | byte;
    return(8);
}

/* if the byte is spread over two words */
else
{
    /* first word, lsb of byte */
    mask = ~((255 << offset));
    ds->data[word_num] = ds->data[word_num] & mask;
    new_byte = byte << offset;
    ds->data[word_num] = ds->data[word_num] | new_byte;

    /* second word, msb of byte */
    /* if next byte is over the end of the data_stream stop */
    /* check this bound!!! */
    if((word_num+2) * 8 > ds->size)
        return(offset);
    mask = ~(255 >> (16 - offset));
    ds->data[word_num+1] = ds->data[word_num+1] & mask;
    new_byte = byte >> (16 - offset);
    ds->data[word_num+1] = ds->data[word_num+1] | new_byte;
    return(8);
}

int get_byte(struct data_stream *ds, long start, short *byte)
{
    /* returns the 8 bits starting at bit number start as the arg byte
     returns number of bits return. If they byte goes over the end
     of the data_stream then the lsb's are filled in with 0's */
}
int word_num;
int offset;
short mask;
short new_byte;

/*if byte outside the data stream*/
if (start > ds->size)
{
    printf("attempt to get byte not in data_stream: %d, %d\n", 
        ds->size, start);
    return(0);
}

/* find the offset from the beginning of a word and the word number within the stream */
offset = start % 16;
word_num = start / 16;

/* if the entire byte falls within one word */
if(offset <= 8)
{
    /* mask off the correct byte within the word, and shift it over to the correct position*/
    mask = (255 << offset);
    *byte = ((ds->data[word_num] & mask) >> offset);
    return(8);
}

/*if the byte is spread over two words*/
else
{
    /*first word, lsb of byte*/
    mask = (255 << offset);
    *byte = (ds->data[word_num] & mask) >> offset;

    /*second word, msb of byte*/
    /*if next byte is over the end of the data_stream stop*/

    /*check this bound!!!*/
    if((word_num+2) *8 > ds->size)
        return(offset);

    mask = (255 >> (16-offset));
    *byte = (*byte |
        ((ds->data[word_num+1] & mask) << (16-offset)));

    return(8);
}
```c
int get_ds_length(struct data_stream *ds)
/*returns length of data_stream*/
{
    return(ds->size);
}

struct data_stream *append_streams(struct data_stream *ds1,
                                  struct data_stream *ds2)
{
    int size;
    int i, j;
    struct data_stream *new_ds;
    int offset;
    unsigned short mask1, mask2;
    int ds_offset;

    /*create a new data stream*/
    size = ds1->size + ds2->size;
    new_ds = create_data_stream(size);

    /*transfer over data from ds1*/
    for(i=0; i<= ds1->size/16; i++)
        new_ds->data[i] = ds1->data[i];

    /*transfer over data from ds2*/
    offset = ds1->size % 16;
    mask1 = 65535 >> offset;
    mask2 = 65535 << (16 - offset);
    ds_offset = ds1->size/16;

    /*clear overflow bits from ds1*/
    new_ds->data[ds_offset] = new_ds->data[ds_offset] & ~(65535 << offset);

    /*for each word in ds2, split it in half at offset, and
    add two halves one at a time*/
    for(i=0; i<= ds2->size /16; i++)
    {
        /*first half of ds2 word*/
        new_ds->data[ds_offset+i] = new_ds->data[ds_offset+i] |
                                (ds2->data[i] & mask1) << offset);

        /*second half of ds2 word*/
        new_ds->data[ds_offset+i+1] =
                                (ds2->data[i] & mask2) >> (16 - offset));
    }

    return(new_ds);
}
```
void print_stream(struct data_stream *ds)
{
    /*print each bit of stream, in binary*/
    int i;
    for(i=0; i< ds->size; i++)
        printf("%d", get_bit(ds, i));
    printf("\n");
}

int add_bits(struct data_stream *ds, int num_bits, int start, int bits)
/*add num_bits number of bits to ds starting at start (bit number) from the number bits*/
/*for now this is cludged using set byte and set bit*/
{
    int i;
    int j;
    for(i = 0; i<num_bits/8; i++)
    {
        set_byte(ds, i*8, (((255<<(i*8)) & bits)>> (i*8)));
    }
    j = (num_bits- (num_bits%8));
    for(i=0; i<num_bits%8; i++)
        set_bit(ds, j+i, ((1<<(j+i)) & bits)>> j+i);
}

struct data_stream *parse_to_ds(int datal, int sizel, int arctics[NA])
/*creates a data stream with data1 for each specified arctic and no data for each bypasses*/
{
    struct data_stream *ds;
    int i,j,k;
    int count =0;
    int place =0;
    for(i=0; i<NA; i++)
    {
        if (arctics[i] == 1)
            count++;
    }
    ds = create_data_stream(count*sizel);
    for(i=0; i<NA; i++)
if (arctics[i] == 1)
{
    for(j=0; j<size1; j++)
    {
        set_bit(ds, place++, ((data1 & (1<<j)) >> j ));
    }
}

return(ds);
}
A.4 scripts.h

```c
#include <stdio.h>

int do_start_log(char log_name[]);
int do_end_log();
int do_read_psc();
int do_write_psc();
int do_reset();
int do_reset_controller();
int do_port_stat();
int do_port_stats();
int do_all_stats();
int do_clear_stats();
int do_errors();
int do_bufs_free();
int do_cfg_read_port();
int do_cfg_write_port();
int do_cfg_write_all();
int do_change_mode();
int do_extest();
int do_arcintest();
int do_ctlm_scan();

int get_next_line(char data[6][60]);

struct datastream *read_arctic_and_data_lines(int arctics[], int num_bits);

int read_arctic_lines(int arctics[]);
```

56
A.5 scripts.c

/* Elth Ogston
   11–16–95
   Sample Scripting Language to Interface with ANCIS commands */

/* just now all that's needed is write_psc and read_psc, max arctics = 1*/
#include "ancis.h"
#include "scripts.h"

/* at present this scripting language does NOT
   provide for $? input commands
   deal with FILENAME data replacement files
   allow for data check files
   allow for comment lines starting with a #

   It DOES
   allow each ANCIS command to be entered
   read commands and ARCTIC/data inputs */

/* global variables for these functions, the script file to
   read from, and the current line number being read */

int line_number = 0;
FILE *script;

main(int argc, char *argv[])
{
    char line_parses[6][60];

    /* check for the correct number of arguments to the run_script
       command (i.e. one, the file name) */
    if (argc != 2)
    {
        printf("to run type run_script <script file> \n");
        return 1;
    }

    /* open the script file for reading*/
    if ((script = fopen(argv[1], "r")) == NULL)
    {
        printf("can\’t open file %s\n", argv[1]);
        return 1;
    }

    /* the main loop of the program goes through the script file lines
       one at a time, does a big if-then lookup of the command names
       and calls a do_that_command function. Each do_command function

57
reads the data/ARCTIC lines after the command (until it reaches a new command) and then calls the function itself with the given data.

lines are always read with the get_next_line command. This parses out the data on that line, be it a command name, input data, or ARCTIC number. It returns (as arguments) the data as a set of strings, in order, of the strings that were separated by white space in the line.

If the next line is the end of the file, get_next_line returns EOF, otherwise it returns TRUE.

the longest possible data field is 60 bits. Since C ints(longs) are 32 bits any data longer than 32 bits will be divided into two strings. Data is written in the script files as hex numbers.

max string size will be from file names. These will be limited to 60 character length strings. Lines never have more than 3? strings on them?? */

```c
while(get_next_line(line_parses) != EOF)
{
    /* yep, it's the great big lookup table of commands*/

    if (strcmp(line_parses[0], "start_log") == 0)
        do_start_log(line_parses[1]);

    else if (strcmp(line_parses[0], "end_log") == 0)
        do_end_log();

    else if (strcmp(line_parses[0], "read_psc") == 0)
        do_read_psc();

    else if (strcmp(line_parses[0], "write_psc") == 0)
        do_write_psc();

    else if (strcmp(line_parses[0], "reset") == 0)
        do_reset();

    else if (strcmp(line_parses[0], "reset_controller") == 0)
        do_reset_controller();

    else if (strcmp(line_parses[0], "port_stat") == 0)
        do_port_stat();

    else if (strcmp(line_parses[0], "port_stats") == 0)
        do_port_stats();

    else if (strcmp(line_parses[0], "all_stats") == 0)
        do_all_stats();
```
else if (strcmp(line_parses[0], "clear_stats") == 0)
    do_clear_stats();

else if (strcmp(line_parses[0], "errors") == 0)
    do_errors();

else if (strcmp(line_parses[0], "bufs_free") == 0)
    do_bufs_free;

else if (strcmp(line_parses[0], "cfg_read_port") == 0)
    do_cfg_read_port();

else if (strcmp(line_parses[0], "cfg_write_port") == 0)
    do_cfg_write_port();

else if (strcmp(line_parses[0], "cfg_write_all") == 0)
    do_cfg_write_all();

else if (strcmp(line_parses[0], "change_mode") == 0)
    do_change_mode();

else if (strcmp(line_parses[0], "extest") == 0)
    do_extest();

else if (strcmp(line_parses[0], "arcintest") == 0)
    do_arcintest();

else if (strcmp(line_parses[0], "do_llmscan") == 0)
    do_do_llmscan();

else
    printf("syntax error or unrecognized function line \%d; \%s \n",
           line_number, &line_parses[0]);

/*close script file*/

int do_start_log(char log_name[])
{
}

int do_end_log()
{
}

int do_read_psc()
{
    int arctics[NA];
    int count;
    struct data_stream *ds;

    printf("read_psc\n \n");
count = read_arctic_lines(arctics);

ds = create_data_stream(count*18);
/*psc is 18 bits long*/

read_psc(ds, arctics);

printf("psc read = ");
print_stream(ds);
}

int do_write_psc()
{
  struct data_stream *data;
  int arctics[NA];
  int i;

  data = read_arctic_and_data_lines(arctics, 18);

  if(TEST==1)
  {
    printf("write_psc \\
    data = \\
    arctics = \\
    for(i=0; i<NA; i++)
    printf("%d", arctics[i]);
    printf("\n \n");
  }

  write_psc(data, arctics);
}

int do_reset()
{
  int arctics[NA];
  int i;

  read_arctic_lines(arctics);

  if(TEST==1)
  {
    printf("reset \\
    arctics = \\
    for(i=0; i<NA; i++)
    printf("%d", arctics[i]);
    printf("\n \n");
  }

  reset(arctics);
}
int do_reset_controller()
{
    if (TEST == 1)
    {
        printf("reset_controller \n");
    }

    board_reset();
}

int do_port_stat()
{
}

int do_port_stats()
{
}

int do_all_stats()
{
}

int do_clear_stats()
{
    int arctics[NA];

    read_arctic_lines(arctics);

    clear_stats(arctics);
}

int do_errors()
{
}

int do_bufs_free()
{
}

int do_cfg_read_port()
{
}

int do_cfg_write_port()
{
}

int do_cfg_write_all()
{
    struct data_stream *data[5];
    int arctics[NA];
    int i;
    char next_line[8][60];
for(i=0; i<5; i++)
{
    get_next_line(next_line);
    data[i] = read_arctic_and_data_lines(arctics, 32);
}

if(TEST==1)
{
    printf("cfg_write_all \n");
    printf("data = \n");
    for(i=0; i<5; i++)
    {
        print_stream(data[i]);
        printf("arctics = ");
        for(i=0; i<NA; i++)
        {
            printf("%d", arctics[i]);
            printf("\n \n");
        }
    }
    cfg_write_all(data, arctics);
}

int do_change_mode()
{
    int arctics[NA];
    char next_line[6][60];
    int i;

    get_next_line(next_line);
    read_arctic_lines(arctics);

    if(TEST==1)
    {
        printf("change_mode \n");
        printf("%s \n", next_line[0]);
        printf("arctics = ");
        for(i=0; i<NA; i++)
        {
            printf("%d", arctics[i]);
            printf("\n \n");
        }
    }

    if(strcmp(next_line[0], "normal") == 0)
        change_mode(MODE_normal, arctics);
    else if (strcmp(next_line[0], "config") == 0)
        change_mode(MODE_config, arctics);
    else if (strcmp(next_line[0], "low level test") == 0)
        change_mode(MODE_low_level_test, arctics);
    else
        printf("unrecognized mode %s \n", next_line[0]);
}

int do_extest()
{
}
int do_arcintest()
{
}

int do_do_llm_scan()
{
}

/*
lines are always read with the get_next_line command
this parses out the data on that line, be it a command name,
input data, or ARCTIC number. It returns (as arguments)
the data as a set of strings, in order, of the strings that were
seperated by white space in the line.

If the next line is the end of the file, get_next_line returns
EOF, otherwise it returns TRUE.

the longest possible data field is 60 bits. Since C ints(longs)
are 32 bits any data longer then 32 bits will be divided into
two strings. Data is written in the script files as hex numbers.

max string size will be from file names. These will be limited
to 60 character length strings. Lines never have more then
3? strings on them??
*/

int get_next_line(char data[6][60])
{
    /*read in the next line*/
    /*parse out the strings within it, assumes there are only 3 data objects*/
    /*returns the number of data objects read, or EOF if the end of file
    was reached*/

    int i = 0;
    char temp_string[301];

    /*if the line was blank, ignore it*/
    while(i <= 0)
    {
        /*get next line up to end line*/

        if( fgets(temp_string, 290, script) == NULL)
            return(EOF);

        /*parse it*/
        i = sscanf(temp_string, "%s %s %s %s %s", &data[0], &data[1], &data[2],
                   &data[3], &data[4], &data[5]);

        /*update line count*/
        line_number++;
    }

    /*return number of data objects read*/
struct data_stream *read_arctic_and_data_lines(int arctics[], int num_bits) {
    /* read in ARCTICS lines and data to go with each
    each Arctic used gets one line, Arctics are listed
    in increasing order, data for each arctic follows
    on line immediately after the arctic.

    read_arctic_and_data_lines, returns number of arctics read,
    returns as args arctics list and pointer to data_stream

    prints an error if no Arctic lines are found
    */
    /*read arctic lines (up to NA of them) and
    one data line per arctic line*/

    struct data_stream *data;
    int temp_data[NA];
    char next_line[6][60];
    int i;
    int c;
    int count = 0;

    /*initialize arctics array*/
    for(i=0; i<NA; i++)
        arctics[i] = 0;

    /*read arctic and data lines*/
    for(i=0; i<NA; i++) {
        /*if next line isn't an arctics line, stop here*/
        /*an arctic line will always start with an A
        nothing else will (except data which would be
        proceeded by an arctic line)??*/

        if((c = getc(script)) != 'A')
            ungetc(c, script);
        ungetc(c, script);

        /*read Arctic line*/
        get_next_line(next_line);
        i = strtol(next_line[1], (char**)NULL, 16);
        if(i >= NA)
            printf("error: arctic number larger then scan chain
                    entered line number %d \n", line_number);
        arctics[i] = 1;

        /*read data line*/
        get_next_line(next_line);
    }

    return(i);
}

temp_data[count] = strtol(next_line[0], (char**)NULL, 16);
count++;
}

/*create data stream*/
data = create_data_stream(count*num_bits);
/*add data to data stream*/
for(i=0; i<count; i++)
    add_bits(data, num_bits, i*num_bits, temp_data[i]);
return(data);
}

int read_arctic_lines(int arctics[])
/*same as for read_arctic_and_data_lines, except for functions like reads that don't take any data*/
{
    /* read in ARCTICS lines assuming no data follows
        each Arctic used gets one line, Arctics are listed
        in increasing order
        read_arctic_lines, returns number of arctics read,
        returns as args arctics list
    */
    /*read arctic lines (up to NA of them) and
     any data line per arctic line*/
    char next_line[6][60];
    int i;
    int c;
    int count = 0;
    /*initialize arctics array*/
    for(i=0; i<NA; i++)
        arctics[i] = 0;
    /*read arctic*/
    for(i=0; i<NA; i++)
    {
        /*if next line isn't an arctics line, stop here*/
        /*an arctic line will always start with an A
         nothing else will (except data which would be
         proceeded by an arctic line)?*/
        if((c = getc(script)) != 'A')
            {
                ungetc(c, script);
                break;
            }
        ungetc(c, script);
        /*read Arctic line*/
get_next_line(next_line);
i = strtol(next_line[1], (char**)NULL, 16);
if(i >= NA)
    printf("error: arctic number larger then scan chain
          entered line number %d \n", line_number);
arctics[i] = 1;
count++;
}

return(count);
}
A.6 upper_level.h

```
enum STAT {
    PACKETS = STAT_PACKETS,
    PRIORITY = STAT_PRIORITY,
    UP = STAT_UP,
    DOWN = STAT_DOWN,
    IDLE = STAT_IDLE,
    WAIT= STAT_WAIT};

void read_psc(struct data_stream *data, int arctics[]);
void write_psc(struct data_stream *data, int arctics[]);
void reset(int arctics[]);
void reset_controller();
void port_stat(int port, enum STAT stat, struct data_stream *data, int clear, int arctics[]);
void port_stats(int port, struct data_stream *data[6], int clear, int arctics[]);
void all_stats(struct data_stream *data[4][6], int clear, int arctics[]);
void clear_stats(int arctics[]);
void errors(struct data_stream *data, int clear, int arctics[]);
void bufs_free(struct data_stream *data, int arctics[]);
void cfg_read_port(int port, struct data_stream *data[5], int arctics[]);
void cfg_write_port(int port, struct data_stream *data[5], int arctics[]);
void cfg_write_all(struct data_stream *data[5], int arctics[]);
void change_mode(enum MODE mode, int arctics[]);
void extest(FILE *data_in, FILE *data_out, int size, int arctics[]);
void arcintest(FILE *data_in, FILE *data_out, int size, int arctics[]);
void do_llm_scan(FILE *data_in, FILE *data_out, int arctics[]);
```
#include "ancis.h"

void old_change_mode(enum MODE mode, int arctics[]);
int Global_Mode[NA];

void read_psc(struct data_stream *data, int arctics[])
{
    int i, len=0;
    if(TEST)
    {
        printf("read psc \n");
    }
    /*check for correct length data stream*/
    for(i=0; i<NA; i++)
        if (arctics[i] == 1)
            len++;
    if (get_ds_length(data) != ((NA-len) + len*18))
        {
            printf("wrong data stream length given to read_psc\n");
            return;
        }
    read_reg(CONTROL_PORT_STATS, arctics);
    get_status(data, 18, arctics, 0);
}

void write_psc(struct data_stream *data, int arctics[])
{
    struct data_stream *temp_ds;
    if(TEST)
    {
        printf("write psc data: ");
        print_stream(data);
    }
    temp_ds = create_data_stream(12+NA);
    write_reg(CONTROL_PORT_STATS, data, 18, arctics);
    /*no status required*/
}

void reset(int arctics[])
{
    /*reset is done by writting the one bit long reset register
     0x341*/
    int i;
}
struct data_stream *ds;
    /* create a junk data_stream to write*/
ds = create_data_stream(NA);

write_reg(SYSTEM_RESET, ds, 1, arctics);
    /* no status required*/
    /* set Global_Mode to normal*/
for(i=0; i<NA; i++)
    Global_Mode[i] = MODE_normal;
}

void reset_controller()
{
    if(TEST)
        printf("reset_controller\n");
    // reset the controller and cycle all the TAP controllers
    // back to RTI
    // uses board level functions*/
    board_reset;
}

void port_stat(int port, enum STAT stat, struct data_stream *data,
    int clear, int arctics[])
{
    /* stats start at 280 +1 for each one, +6 for each port*/
    /* +0x40 to clear */
    /* stats do need status*/
    /* register size is 36 + two status bits*/
    read_reg(stat+(6*port)+(0x40*clear), arctics);
    get_status(data, 38, arctics, 1);
}

void port_stats(int port, struct data_stream *data[6], int clear, 90
    int arctics[])
{
    int i;
    for(i=0; i<6; i++)
    {
        read_reg(STAT_PACKETS+i+(6*port)+(0x40*clear), arctics);
        get_status(data[i], 38, arctics, 1);
    }
}

void all_stats(struct data_stream *data[4][6], int clear, int arctics[])
{
    int i, j;
    for(j=0; j<4; j++)
    {  

for(i=0; i<6; i++)
{
    read_reg(STAT_PACKETS+i+(6*j)+(0x40*clear), arctics);
    get_status(data[j][i], 38, arctics, 1);
}

void clear_stats(int arctics[])
{
    /*stats are cleared by reading the clear stats register
    size 36, + 2 status, status is needed*/

    struct data_stream *temp_ds;
    int len = 0;
    int i;

    if(TEST == 1)
        printf("clear_stats \n");

    for(i=0; i<NA; i++)
        if(arctics[i] == 1)
            len++;

    len = (len*38)+ NA-len;
    temp_ds = create_data_stream(len);

    read_reg(STAT_CLEAR_ALL, arctics);
    get_status(temp_ds, 38, arctics, 1);
}

void errors(struct data_stream *data, int clear, int arctics[])
{
    /*one register, 60 bits long. no status required*/
    /*to clear read ERROR+1*/

    read_reg(ERROR + clear, arctics);
    get_status(data, 60, arctics, 0);
}

void bufs_free(struct data_stream *data, int arctics[])
{
    /*one register, 16 bits long. no status*/
    read_reg(BF_COUNTERS, arctics);
    get_status(data, 20, arctics, 0);
}

void cfg_read_port(int port, struct data_stream *data[5],
                    int arctics[])
{
    /*each cfg register is 32 bits, status is needed*/
    /*registers start at 0, + one for next in port,
     + 40(hex) for each port*/

/ you must be in configuration mode before running any of the configuration commands*/

int i;

/*check to make sure the mode is ok*/

for(i=0; i<NA; i++)
  if((arctics[i] == 1) &&
     (Global_Mode[i] != MODE_config))
    {
      printf("ERROR, attempt to access configuration registers when not in config mode\n");
      return;
    }

for(i=0; i<5; i++)
  {
    read_reg(CONFIG_PORT + (0x40*port) + i, arctics);
    get_status(data[i], 34, arctics, 1);
  }

void cfg_write_port(int port, struct data_stream *data[5],
                      int arctics[])
{
  int i;
  struct data_stream *temp_ds;
  int len =0;

  /*check to make sure the mode is ok*/

  for(i=0; i<NA; i++)
    if((arctics[i] == 1) &&
       (Global_Mode[i] != MODE_config))
      {
        printf("ERROR, attempt to access configuration registers when not in config mode\n");
        return;
      }

  for(i=0; i<NA; i++)
    if(arctics[i] == 1)
      len++;

  len = (len*34)+ NA-len;
  temp_ds = create_data_stream(len);

  for(i=0; i<5; i++)
    {
      write_reg(CONFIG_PORT + (0x40*port) + i, data[i], 34, arctics);
      get_status(temp_ds, 34, arctics, 1);
    }
}
void cfg_write_all(struct data_stream *data[5], int arctics[])
{
    /*write to the CONFIG_BROADCAST reg*/
    int i;
    struct data_stream *temp_ds;
    int len = 0;

    /*check to make sure the mode is ok*/
    for (i = 0; i < NA; i++)
        if ((arctics[i] == 1) &&
            (Global_Mode[i] != MODE_config))
        {
            printf("ERROR, attempt to access configuration registers 
when not in config mode\n");
            return;
        }

    for (i = 0; i < NA; i++)
        if (arctics[i] == 1)
            len++;

    len = (len * 34) + NA - len;
    temp_ds = create_data_stream(len);

    for (i = 0; i < 5; i++)
        write_reg(CONFIG_BROADCAST + i, data[i], 34, arctics);
        get_status(temp_ds, 34, arctics, 1);
}

void change_mode(enum MODE mode, int arctics[])
{
    /*must disable all inputs and outputs using psc
    before entering configuration mode*/
    /*mode changed by writting the M_reg, 2 bits,
    it's an access macro, not an arctic access op.*/
    /*Global_Mode keeps track of what mode each arctic
    is in*/

    /* FOR NOW CHANGE MODE DOES NOT DEAL WITH ENABLEING
    OR DISABLEING INPUT PORTS, IT IS UP TO THE USER TO
    DEAL WITH THESE ARCTIC RULES */

    int i;

    printf("CHANGE MODE %d \n", mode);
    write_Mreg(mode, arctics);

    /*correct the Global_Mode*/

    for (i = 0; i < NA; i++)
if(arctics[i] == 1)
    Global_Mode[i] = mode;
}

void old_change_mode(enum MODE mode, int arctics[])
{
    /*must disable all inputs and outputs using psc
     before entering configuration mode*/
    /*mode changed by writing the M_reg, 2 bits,
     it's an access macro, not an arctic access op.*/
    /*Global_Mode keeps track of what mode each arctic
     is in*/

    /*VERSION OF CHANGE MODE THAT ATTEMPTS TO DEAL WITH
     INPUT PORTS, NOT DEBUGGED YET*/
    int i,j;
    struct data_stream *temp_ds;
    int len =0;
    int place =0;

    for(i=0; i<NA; i++)
        if(arctics[i] == 1)
            len++;

    printf("CHANGE MODE %d \n", mode);

    /*psc is 18 ibts*/
    len = (len*18)+ NA-len;
    temp_ds = create_data_stream(len);

    if (mode == MODE_config)
    {
        for(i=0; i<NA; i++)
            if(arctics[i] == 1)
                for(j=0; j<NA; j++)
                    if(arctics[j] == 0)
                        place++;
                    else
                        set_bit(temp_ds, place, ONE);
                        set_bit(temp_ds, place+1, ONE);
                        set_bit(temp_ds, place+4, ONE);
                        set_bit(temp_ds, place+5, ONE);
                        set_bit(temp_ds, place+8, ONE);
                        set_bit(temp_ds, place+9, ONE);
                        set_bit(temp_ds, place+13, ONE);
                        break;
```c
set_bit(temp_ds, place+14, ONE);
place = place + 18;
}

/*write new psc*/
write_psc(temp_ds, arctics);
}
}

write_Mreg(mode, arctics);

/*if modes was config, and changed to normal or llm_test reenable inputs and outputs.*/
if(mode != MODE_config)
{
  for(i=0; i<NA; i++)
    if ((GlobalMode[i] == MODE_config) & (arctics[i] == 1))
      {
        /*enable inputs outputs*/
        /*read current psc*/
        read_psc(temp_ds, arctics);
        /*set enable bits*/

        for(j=0; j<NA; j++)
        {
          if(arctics[j] == 0)
            place++;
          else
            {
              set_bit(temp_ds, place, ZERO);
              set_bit(temp_ds, place+1, ZERO);
              set_bit(temp_ds, place+4, ZERO);
              set_bit(temp_ds, place+5, ZERO);
              set_bit(temp_ds, place+8, ZERO);
              set_bit(temp_ds, place+9, ZERO);
              set_bit(temp_ds, place+13, ZERO);
              set_bit(temp_ds, place+14, ZERO);
              place = place + 18;
            }
        }

        /*write new psc*/
        write_psc(temp_ds, arctics);
      }
}

/*correct the Global_Mode*/
for(i=0; i<NA; i++)
  if(arctics[i] == 1)
    Global_Mode[i] = mode;
```
void extest(FILE *data_in, FILE *data_out, int size, int arctics[])
{
}

void arcintest(FILE *data_in, FILE *dataout, int size, int arctics[])
{
}

void do_llmscan(FILE *data_in, FILE *data_out, int arctics[])
{
    int i;
    struct data_stream *temp_ds;
    int len =0;
    int place;

    /*create a data stream to send things in on*/
    for(i=0; i<NA; i++)
        if(arctics[i] == 1)
            len++;

    /*llm scan reg is 6 ibts*/
    len = (len*6)+ NA-len;
    temp_ds = create_data_stream(len);

    /*open files—hmmm, already open, is this right?*/
    /*read in next stream*/
    for(i=0; i<len; i++)
    {
        /*set_bit(temp_ds, place, atoi(getc(data_in)));*/
    }

    /*send to arctics*/
    /*write out to file*/
    /*change mode*/
    change_mode(MODE_low_level_test, arctics);

    /*return mode to normal*/
    change_mode(MODE_normal, arctics);
}
A.8  arctic_access_ops.h

void read_reg (enum REGISTERS address, int arctics[]);

void write_reg (enum REGISTERS address, struct data_stream *ds,
                int data_len, int arctics[]);

int get_status (struct data_stream *ds, int data_len, int arctics[],
                int status);
#include "ancis.h"

void read_reg (enum REGISTERS address, int arctics[]) 
{ 
    if(TEST) 
        printf(" read reg: \%x\n", address); 

    write_AOAreg(address, arctics); 
    write_AOIrreg(AOI_read_reg, arctics); 
} 

void write_reg (enum REGISTERS address, struct data_stream *ds, 
        int data_len, int arctics[]) 
{ 
    if(TEST) 
    { 
        printf(" write reg addr: \%x data:", address); 
        print_stream(ds); 
    } 

    write_AOAreg(address, arctics); 
    write_AOSandDreg(ds, arctics, data_len); 
    write_AOIrreg(AOI_write_reg, arctics); 
} 

int get_status (struct data_stream *ds, int data_len, int arctics[], int status) 
{ 
    int place = 0; 
    enum AOS_values current_status; 
    enum AOS_values total_status = IN_PROGRESS; 
    int i; 
    
    if(TEST) 
        printf(" get status\n"); 

    read_AOSandDreg(ds, arctics, data_len); 

    if(status == 1) 
        /*status is needed, check it and continue to read 'till completed*/ 
    { 
        while(total_status == IN_PROGRESS) 
        { 
            total_status == COMPLETED; 
            for(i=0; i<NA; i++) 
            { 
                if(arctics[i] == 0) 
                    /*bypassed*/ 
                    place++; 
                else 
...
```c
{
    /* look at first two bits, well last two since it comes out backwards */
    place = place + data_len - 2;
    current_status =
        get_bit(ds, place++);
    current_status = current_status +
        get_bit(ds, place++)*2;

    if(current_status == IN_PROGRESS)
        total_status = IN_PROGRESS;

    else if ((current_status == MODE_ERROR)
                 ^ (current_status == BROADCAST_READ_ERROR))
    }
    return(current_status);
```
A.10 access_macs.h

```c
enum AOS_values {
    COMPLETED = 0,
    IN_PROGRESS = 1,
    MODE_ERROR = 2,
    BROADCAST_READ_ERROR = 3};

enum REGISTERS{
    CONTROL_PORT_STATS = 0x200,
    SYSTEM_RESET = 0x341,
    STAT_PACKETS = 0x280,
    STAT_PRIORITY = 0x281,
    STAT_UP = 0x282,
    STAT_DOWN = 0x283,
    STAT_IDLE = 0x284,
    STAT_WAIT = 0x285,
    STAT_CLEAR_ALL = 0x2FF,
    ERROR = 0x300,
    BF_COUNTERS = 0x240,
    CONFIG_PORT = 0x000,
    CONFIG_BROADCAST = 0x100,
    MANUF_TEST_RING = 0x380};

enum INSTRS{
    AOI_write_reg = 0,
    AOI_read_reg = 1};

enum MODE {
    MODE_config = 0,
    MODE_normal = 1,
    MODE_low_level_test = 2};

void write_AOAreg(enum REGISTERS address,
    int arctics[NA]);
/*writes the AOAreg with address for each arctic
 specified, otherwise writes a 0 for the bypass reg*/

void write_AOSandDreg(struct data_stream *ds,
    int arctics[NA], int data_len);
/*writes the ds to the AOSandDreg*/

void write_AOIreg(enum INSTRS instruction,
    int arctics[NA]);
/*writes the AOI reg with instruction for each arctic
 specified, otherwise writes the bypass instruction*/

void read_AOSandDreg(struct data_stream *ds,
    int arctics[NA], int data_len);
/*reads the AOSandDreg into ds
 if data_len >0 continues to reread until the status
 for all arctics specified is COMPLETED*/
```
void writeMreg(short mode, int arctics[NA]);
/*writes the given mode for each arctic specified*/

void set_up_llm_scan(struct data_stream *ds, int arctics[]);

void llm_scan(struct data_stream *ds, int arctics[]);

void end_llm_scan();
void write_AOArege(enum REGISTERS address,  
               int arctics[NA])  
{
    if(TEST)
        printf("write_AOArege: %x \n", address);
    load_JTAGIR(JTAG_sel_AOArege, JTAG_bypass, arctics);
    load_JTAGDR(parse_to_ds(address, 10, arctics), arctics, 10);
}

void write_AOIreg(enum INSTRS instruction,  
                  int arctics[NA])  
{
    if(TEST)
        printf("write_AOIreg instr: %x \n", instruction);
    load_JTAGIR(JTAG_sel_AOIreg, JTAG_bypass, arctics);
    load_JTAGDR(parse_to_ds(instruction, 2, arctics), arctics, 2);
}

void write_AOSandDreg(struct data_stream *ds, int arctics[NA],  
                        int datalen)  
{
    if(TEST)
        {  
            printf("write_AOSandDreg data:");
            print_stream(ds);
        }
    load_JTAGIR(JTAG_sel_AOSandDreg, JTAG_bypass, arctics);
    load_JTAGDR(ds, arctics, datalen);
}

void write_Mreg(short mode, int arctics[NA])  
{
    if(TEST)
        printf("write_Mreg \n");
    load_JTAGIR(JTAG_sel_Mreg, JTAG_bypass, arctics);
    load_JTAGDR(parse_to_ds(mode, 2, arctics), arctics, 2);
}

void read_AOSandDreg(struct data_stream *ds,  
                      int arctics[NA], int datalen)  
{
    if(TEST)
        printf("read_AOSandDreg \n");
load_JTAGIR(JTAG_sel_AOSandDreg, JTAG_bypass, arctics);
load_JTAGDR(ds, arctics, data_len);
}

void set_up_llmscan(struct data_stream *ds, int arctics[])
{
    load_JTAGIR(JTAGselAOAreg, JTAG_bypass, arctics);
    load_JTAGDR(parse_to_ds(MANUF.TEST_RING, 10, arctics), arctics, 10);
    load_JTAGIR(JTAG_sel_AOSandDreg, JTAG_bypass, arctics);

    load_JTAGDR2_beg(ds, arctics, 6);
}

void llm_scan(struct data_stream *ds, int arctics[])
{
    load_JTAGDR2(ds, arctics, 6);
}

void end_llm_scan()
{
    load_JTAGDR2_end;
}
A.12  jtag_trans.h

```c
enum JTAGINST{
    JTAG_bypass = 0xF,
    JTAG_extest = 0x0,
    JTAG_preload = 0x1,
    JTAG_arcntest = 0x2,
    JTAG_sel_Mreg = 0x4,
    JTAG_sel_AOAregr = 0x5,
    JTAG_sel_AOIrreg = 0x6,
    JTAG_sel_AOSandDreg = 0x7,
    JTAG_clamp = 0x8};

void load_JTAGIR(enum JTAGINST instr, enum JTAGINST alternate,
                 int arctics[NA]); /*loads IR with instr, alternate for bypassed*/

void load_JTAGDR(struct data_stream *ds, int arctics[NA],
                 int data_len); /*loads DR with ds, adds in the 0 for bypassed bits
for the bypassed arctics. removes them again before returning the tdo*/

void load_JTAGDR2_beg(struct data_stream *ds, int arctics[],
                      int data_length);  

void load_JTAGDR2_end();
```
A.13 jtag_trans.c

#include "ancis.h"

void load_JTAGIR(enum JTAG_INST instr, enum JTAG_INST alternate,
                  int arctics[NA])

{
    struct data_stream *tms_ds, *tdi_ds;
    int place = 0;
    int i;

    if(TEST)
        printf("load JTAGIR instr: \x \n", instr);

    /*length of tdi and tms data streams: all jtag instructions are 4 bits long so length = NA*4*/
    tms_ds = create_data_stream((NA*4)+8);
    tdi_ds = create_data_stream((NA*4)+8);

    /*initialize streams so that tms is 0 when not set otherwise (stays in pause states)
     and tdi is also (junk bits)
     new data streams automatically have 0 in all bits*/
    set_bit(tms_ds, place++, ONE);
    set_bit(tms_ds, place++, ZERO);
    set_bit(tms_ds, place++, ZERO);

    for(i=0; i<NA; i++)
    {
        if(arctics[i] == 0)
            /*put in alternate*/
            {
                set_bit(tdi_ds, place++, (alternate & 1));
                set_bit(tdi_ds, place++, ((alternate>>1) & 1));
                set_bit(tdi_ds, place++, ((alternate>>2) & 1));
                set_bit(tdi_ds, place, ((alternate>>3) & 1));
            }
        else
            /*put in instr*/
            {
                set_bit(tdi_ds, place++, (instr & 1));
                set_bit(tdi_ds, place++, ((instr>>1) & 1));
                set_bit(tdi_ds, place++, ((instr>>2) & 1));
                set_bit(tdi_ds, place, ((instr>>3) & 1));
            }
    }
}

84
/* go to RTI */
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ZERO);

board_send_bits(tdi_ds, tms_ds);

/* board interface needs to deal with sticky and lost
   bits at the end */
}

void load_JTAGDR(struct data_stream *ds, int arctics[NA],
                  int data_len)
{
    struct data_stream *tms_ds, *tdi_ds;
    int place = 0;
    int ds_place = 0;
    int i, j;
    int len = 0;

    if(TEST)
    {
        printf(" load JTAGDR data: ");
        print_stream(ds);
    }

    /* figure out length of streams */
    for(i=0; i<NA; i++)
        if(arctics[i] == 1)
            len++;

    len = len*data_len + (NA-len) + 6;

    /* create streams */
    tms_ds = create_data_stream(len);
    tdi_ds = create_data_stream(len);

    /* initialize streams so that tms is 0 when not set otherwise
       (stays in pause states)
       and tdi is also (junk bits)
       not needed, newly create ds's are all 0's */

    /* get to shift DR stat from RTI */
    set_bit(tms_ds, place++, ONE);
    set_bit(tms_ds, place++, ZERO);
    set_bit(tms_ds, place++, ZERO);
/ *shift DR*/
for(i=0; i<NA; i++)
    if(arctics[i] == 0)
        /*bypass*/
        place++;
    else
        {
            /*copy in data*/
            for(j=0; j<data_len; j++)
                set_bit(tdi_ds, place++, get_bit(ds, ds_place++));
        }
place--;
/*go to RTI*/
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ZERO);
board_send_bits(tdi_ds, tms_ds);
/*bits returned in tdi_ds, remove extra bits, and
send back in ds,
3 extra
1 for each bypass
data_len for each actual
3 more extra*/
place = 3;
ds_place = 0;
for(i=0; i<NA; i++)
    if(arctics[i] == 0)
        /*bypass*/
        place++;
    else
        {
            /*copy over data*/
            for(j=0; j<data_len; j++)
                set_bit(ds, ds_place++, get_bit(tdi_ds, place++));
        }
/*
printf("bits returned from load_JTAGDR:");
print_stream(ds);
*/
}

void load_JTAGDR2_beg(struct data_stream *ds, int arctics[],
    int data_length)
{
    struct data_stream *tms_ds, *tdi_ds;
    int place = 0;
    int ds_place = 0;
    int i, j,
```c
int len = 0;

if(TEST)
    printf("load JTAGDR2_beg\n");

/*figure out length of streams*/
for(i=0; i<NA; i++)
    if(arctics[i] == 1)
        len++;
len = len*data_length + (NA-len) + 5;

/*create streams*/
tms_ds = create_data_stream(len);
tdi_ds = create_data_stream(len);

/*initialize streams so that tms is 0 when not set otherwise
   (stays in pause states) and tdi is also (junk bits)*/
for(i=0; i<(len); i=i+8)
    { 
        set_byte(tms_ds, i, 0);
        set_byte(tdi_ds, i, 0);
    }

/*get to shift DR stat from RTI*/
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ZERO);
set_bit(tms_ds, place++, ZERO);

/*shift DR*/
for(i=0; i<NA; i++)
    if(arctics[i] == 0)
        /*bypass*/
        place++;
else
    { /*copy in data*/
        for(j=0; j<data_length; j++)
            set_bit(tdi_ds, place++, get_bit(ds, ds_place++));
    }

/*go to Pause DR*/
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ZERO);

board_send_bits(tdi_ds, tms_ds);
```
void loadJTAGDR2(struct data_stream *ds, int arctics[],
        int data_length)
{
    struct data_stream *tms_ds, *tdi_ds;
    int place = 0;
    int ds_place = 0;
    int i, j;
    int len = 0;

    if(TEST)
        printf("load JTAGDR2_beg\n");

    /*figure out length of streams*/
    for(i=0; i<NA; i++)
        if(arctics[i] == 1)
            len++;
    len = len*data_length + (NA-len) + 8;

    /*create streams*/
    tms_ds = create_data_stream(len);
    tdi_ds = create_data_stream(len);

    /*initialize streams so that tms is 0 when not set otherwise
     *stays in pause states
     *and tdi is also (junk bits)
     */
    for(i=0; i<(<len); i=i+8)
    {
        set_byte(tms_ds, i, 0);
        set_byte(tdi_ds, i, 0);
    }

    /*get to shift DR stat from Pause-DR*/
    set_bit(tms_ds, place++, ONE);
    set_bit(tms_ds, place++, ONE);
    set_bit(tms_ds, place++, ZERO);
    set_bit(tms_ds, place++, ONE);
    set_bit(tms_ds, place++, ZERO);
    set_bit(tms_ds, place++, ZERO);

    /*shift DR*/
    for(i=0; i<NA; i++)
        if(arctics[i] == 0)
            /*bypass*/
            place++;
        else
            { /*copy in data*/
                for(j=0; j<data_length; j++)
set_bit(tdi_ds, place++, get_bit(ds, ds_place++));
}

/*go to Pause DR*/
set_bit(tms_ds, place++, ONE);
set_bit(tms_ds, place++, ZERO);

board_send_bits(tdi_ds, tms_ds);

}

void load_JTAGDR2_end()
{
    struct data_stream *tms_ds, *tdi_ds;
    int place = 0;
    int ds_place = 0;
    int i;

    /*create streams*/
tms_ds = create_data_stream(3);
    tdi_ds = create_data_stream(3);

    /*initialize streams so that tms is 0 when not set otherwise
     (stays in pause states)
     and tdi is also (junk bits)
     */
    set_byte(tms_ds, 0, 0);
    set_byte(tdi_ds, 0, 0);

    /*get to RTI stat from Pause–DR*/
    set_bit(tms_ds, place++, ONE);
    set_bit(tms_ds, place++, ONE);
    set_bit(tms_ds, place++, ZERO);

    board_send_bits(tdi_ds, tms_ds);
}
A.14  board.h

enum ports{TDI, TMS};
enum board_status{B_COMPLETED, B_IN_PROGRESS};

int board_send_bits(struct data_stream *tdi,
                    struct data_stream *tms);
/*
1. sends given chunk of bits out to the board (in one
   piece. bits must end in a pause state. bits must
   begin in the state the last instruction left them in.

2. sends a go instruction to the board.

3. waits till the board is finished, and reads the data
   from TDO into the tdi data_stream.

***doesn't look at status***
*/

int board_reset();
/*resets the JTAG board

   sends/reads two data words to fill up the extra space
   in the fifo
*/

/*** lower level functions ***/

int board_send_stream(enum ports st, struct data_stream *ds);
/*sends stream to board. fills in to an even
   number of bytes with the last bit in the stream.
   returns last bit of stream*/

int board_go(struct data_stream *tdo, enum bit last_bit_tms);
/*sends and captures data to tdo
   fills in and strips off byte of idle repeats of last_bit
   in both tms and tdi
   reads tdo
   returns board to receiving state when done*/
A.15 board.c

#include "ancis.h"

int board_send_bits(struct data_stream *tdi, 
                    struct data_stream *tms)
{
    int last_bit;

    /*check to make sure there are less then 4k–1 bytes of data*/
    if(get_ds_length(tdi) > 3999*8)
    {
        printf("uh oh, attempt to send chunk bigger then the buffers \n");
        return(0);
    }

    /*fill buffers, send stream fills buffers to an even number of bits*/
    if (TEST == 1)
    {
        printf(" Bits Sent To Board \n");
        printf(" TDI\n");
    }
    board_send_stream(TDI, tdi);

    if (TEST == 1)
        printf(" TMS\n");
    last_bit = board_send_stream(TMS, tms);

    /*send a go signal: go sends bits, waits till done then returns whatever is sent back. It also fills in and strips off the extra byte that fills the serial to parallel converter.*/
    /*returned bits are put back into tdi*/
    board_go(tdi, last_bit);
}

int board_send_stream(enum ports st, struct data_stream *ds)
{
    int i, j, k, len;
    int place = 0;
    unsigned short byte;
    struct data_stream *temp;

    /*send the main stream to the board*/
    /*length of stream to be sent*/
    len = get_ds_length(ds);
/ *send full bytes */
for(i=0; i < (len/8); i++)
{
    get_byte(ds, place, &byte);
    if(st == TDI)
        board_write(TDI_ADDR, byte);
    else
        board_write(TMS_ADDR, byte);
    place = place + 8;
}

/* figure out what the last bit of the stream was */
j = get_bit(ds, len);

/* if there wasn't an even number of bytes, pick up the last one */
if( (len % 8) > 0 )
{
    get_byte(ds, place, &byte);

    /* fill in extra bits of last byte with the last bit, then send it */
    if (j == 0)
        /* clear end of byte */
        byte = byte & (255 >> (8-len%8));
    else
        /* fill in last bits with 1's */
        byte = byte | (255 << (len%8));

    if(st == TDI)
        board_write(TDI_ADDR, byte);
    else
        board_write(TMS_ADDR, byte);
}
return(j);

int board_reset()
{
    short s;
    /*reset*/
    board_write(COMMAND, RESET);

    /*for now ignore the rest*/
    /*tell to stop*/
    /* board_write(COMMAND, STOP);
    */
    /*write some stuff to fill up gaps*/
    /* board_write(TDI_ADDR, 0);
    board_write(TMS_ADDR, 0);
    */
int board_go(struct data_stream *tdo, enum bit last_bit_tms)
{
    /*add one extra byte of info to end of the input streams,
    ignores the first byte coming out*/
    int status;
    int place;
    int i;
    int ds_len;
    short s, j;

    /*fill in extra tdi byte*/
    if (TEST == 1)
        printf(" TDI
");
    board_write(TDI_ADDR, 0);

    /*fill in extra tms byte*/
    if (TEST == 1)
        printf(" TMS
");
    if (last_bit_tms == 0)
    {
        board_write(TMS_ADDR,0);
    }
    else
    {
        board_write(TMS_ADDR, 255);
    }

    /*tell board to go*/
    if (TEST == 1)
        printf(" go command
");
    board_write(COMMAND, START);

    /*wait for it to finish*/
    while(board_read(STATUS, &s))
    {
        if (s == DONE)
            break;
    }
/*tell board to stop*/
board_write(COMMAND, STOP);

/*read data out*/

/* if the board was actually there and saving the last byte to send later you would ignore the first byte returned. as it we ignore the last byte instead */
board_read(TDO_ADDR, &s);

/*figure out data stream length rounding up to the nearest byte*/

ds_len = get_ds_length(tdo);

if ((ds_len % 8) == 0)
    ds_len = ds_len / 8;
else
    ds_len = (ds_len / 8) + 1;

/*write rest into tdo*/
for(i=0; i < ds_len; i++)
{
    board_read(TDO_ADDR, &j);
    set_byte(tdo, (i*8), j);
}

/*last byte ignored, for simulation only*/
board_read(TDO_ADDR, &s);
A.16 sim.h

/* currently writes two files, tms, and tdi to be read by
verilog into memories and fed into the simulated chip.
verilog will return bits in a new file, tdo, in which
contains the bits returned on the tdo. issuing a start command
gets verilog running */

enum ADDR{
    TDI_ADDR = 0x120,
    /* for writing to the tdi stream*/
    TMS_ADDR,
    /* for writing to the tms stream*/
    COMMAND,
    /* for writing commands*/
    TDO_ADDR = 0x120,
    /* for reading bits returned by tdo*/
    STATUS = 0x122};
    /* for checking board status*/

enum B_STAT{
    DONE,
    NOT_DONE
};

enum COM{
    START,
    /* start sending bits, closes files, tells verilog to go */
    STOP,
    /* must be called after start, and before the next start*/
    RESET
    /* must be called on startup to clear FIFOs (well, actually
to open the fifo files*/
};

enum JTAG_STATES{
    TLR,
    RTI,
    SDRS,
    CDR,
    SDR,
    E1DR,
    PDR,
    E2DR,
    UDR,
    SIRS,
    CIR,
    SIR,
    E1IR,
    PIR,
    E2IR,
    UIR
};
void board_write(enum ADDR address, short byte);
/*writes a byte to a board register*/
/* can write data to tdi or tms, or commands,
   start, stop, and reset*/

int board_read(enum ADDR address, short *byte);
/*reads a byte from a board register*/
/*can read status, or tdo*/

void do_command(enum COM c);
/*opens and closes files */
#include "ancis.h"
#include <time.h>

enum B_STAT board_status = DONE;
short bytes_returned = 0;
    *cycle_file;
int size_data_in_fifo;
int cycles = 0;

void board_write(enum ADDR address, short byte)
{
    int i;

    if (TEST == 1)
        printf(" board_write addr:%x data:%x \n", address, byte);

    /*deal with what board_status should be??*/

    if (address == TDI_ADDR)
    {
        board_status = NOT_DONE;
        fprintf(tdi_file, "%02x ", byte);
        size_data_in_fifo++;
    }

    if (address == TMS_ADDR)
    {
        board_status = NOTDONE;
        fprintf(tms_file, "%02x ", byte);
    }

    if (address == COMMAND)
    do_command(byte);

}

int board_read(enum ADDR address, short *byte)
{
    int i;
    int c;

    /*returns size of data written to byte*/
    /*returns 0 if no bytes left*/
    /*once you start reading the tdo you must keep reading
     till you've got all of it*/

    if (address == TDO_ADDR)
    {
        /*read bytes from tdo_file, they are in hex
         so you need 4 for a byte*/
/*WHAT DIRECTION DO THEY COME BACK IN*/ left
in the same direction as they are read out of the
file*/
/*clear byte*/

*byte = 0;

for(i=1; i>=0; i--)
{
c = getc(tdo_file);

/*there are returns between each byte*/
if (c == 'n')
c = getc(tdo_file);

/*end if c = EOF*/
if (c == EOF)
return( (i-1) * -4);

/*convert c from hex characters*/
c = c - '0';
if (c > 9)
c = c-39;

/*add c to byte*/
*byte = *byte | (c << (i*4));
}
printf(" board_read: %x\n", *byte);
return(8);

if(address == STATUS)
{
*byte = board_status;
printf(" board_read: status\n");
return(0);
}

void do_command(enum COM c)
{
if (c == RESET)
{
/*open files for writing to*/
/*for now take whatever bits are written, and write them to
files to be sent to verilog as the TMS stream and the
TDI stream*/

/*open files*/
if ((tms_file = fopen("fifo_tms_out", "w")) == NULL)
{
printf("can't open file tms_out\n");
}
return;
}

if ((tdi_file = fopen("fifo_tdi_out", "w")) == NULL)
{
    printf("can't open file tdi_out
")
    return;
}

bytes Returned = 0;
board_status = DONE;
size_data_in_fifo = 0;
}

else if (c == START)
{
    board_status = DONE;
    /*close files*/
    fclose(tdi_file);
    fclose(tms_file);
    fclose(tdo_file);
    /*reopen files*/
    if ((size_file = fopen("file_size", "w")) == NULL)
    {
        printf("can't open file file_size
")
        return;
    }
    fprintf(size_file, "%02x
", size_data_in_fifo);
    fclose(size_file);

    printf("-----------------Please wait for Verilog to run----------------
");
    / *tell verilog to go*/
    cycle_file = fopen("cycle_file", "w");
    fprintf(cycle_file, "%04x
", cycles);
    cycles++;
    fclose(cycle_file);
    /*
     getchar();
     */
    /*wait till verilog is done*/
    /*verilog is done reading /writing when it writes the file "done_file", wait for it to be readable*/
    while((done_file = fopen("done_file", "r")) == NULL)
    {
        /*wait for a bit (10 seconds)*/
        sleep(10);
    }
/ * delete done_file so it's not around 'till verilog
   writes it again*/
   
   remove("done_file");
   fclose(done_file);
   /*
   unlink("done_file");
   */
   
   size_data_in_fifo = 0;

else if (c == STOP)
{
    board_status = DONE;

    /*reopen files*/
    if ((tms_file = fopen("fifo_tms_out", "w")) == NULL)
    {
        printf("can't open file tms_out\n");
        return;
    }

    if ((tdi_file = fopen("fifo_tdi_out", "w")) == NULL)
    {
        printf("can't open file tdi_out\n");
        return;
    }

    if ((tdo_file = fopen("fifo_tdo_in", "r")) == NULL)
    {
        printf("meep, can't open file tdo_in \n");
        while(1 == 1)
            sleep(100);
    }
}